UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Eduardo Mobilon

100 GBIT/S AES-GCM CRYPTOGRAPHY ENGINE FOR OPTICAL TRANSPORT NETWORKS: ARCHITECTURE, DESIGN, AND 40 NM SILICON PROTOTYPING

MÓDULO CRIPTOGRÁFICO AES-GCM OPERANDO A 100 GBIT/S PARA REDES DE TRANSPORTE ÓPTICO: ARQUITETURA, PROJETO E PROTOTIPAGEM EM 40 NM

Campinas
2023

Eduardo Mobilon

100 Gbit/s AES-GCM Cryptography Engine for Optical Transport Networks: Architecture, Design, and 40 nm Silicon Prototyping

Módulo Criptográfico AES-GCM Operando a 100 Gbit/s Para Redes de Transporte Óptico: Arquitetura, Projeto e Prototipagem em 40 nm

Thesis presented to the School of Electrical and Computer Engineering of the University of Campinas, as partial fulfillment of the requirements for the degree of Doctor in Electrical Engineering, in the area of Telecommunications and Telematics.

Tese apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas, como parte dos requisitos exigidos para a obtenção do título de Doutor em Engenharia Elétrica, na área de Telecomunicações e Telemática.

Supervisor/Orientador: Prof. Dr. Dalton Soares Arantes

Este trabalho corresponde à versão final da tese defendida pelo aluno Eduardo Mobilon, orientada pelo Prof. Dr. Dalton Soares Arantes.

Campinas

2023

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Rose Meire da Silva - CRB 8/5974

M713g

Mobilon, Eduardo, 1973-
100 Gbit/s AES-GCM cryptography engine for optical transport networks : architecture, design, and 40 nm silicon prototyping / Eduardo Mobilon. – Campinas, SP : [s.n.], 2023.

Orientador: Dalton Soares Arantes.
Tese (doutorado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Criptografia. 2. Autenticação. 3. Redes ópticas. 4. Circuitos digitais. 5. Circuitos integrados. I. Arantes, Dalton Soares, 1946-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações Complementares

Título em outro idioma: Módulo criptográfico AES-GCM operando a 100 Gbit/s para redes de transporte óptico : arquitetura, projeto e prototipagem em 40 nm
Palavras-chave em inglês:
Cryptography
Authentication
Optical networks
Digital circuits
Integrated circuits
Área de concentração: Telecomunicações e Telemática
Titulação: Doutor em Engenharia Elétrica
Banca examinadora:
Dalton Soares Arantes [Orientador]
Max Henrique Machado Costa
Kayol Soares Mayer
João Batista Rosolem
Monica de Lacerda Rocha
Data de defesa: 24-02-2023
Programa de Pós-Graduação: Engenharia Elétrica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-6101-2940
- Currículo Lattes do autor: http://lattes.cnpq.br/9699996941745210

UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

100 Gbit/s AES-GCM Cryptography Engine for Optical Transport Networks: Architecture, Design, and 40 nm Silicon Prototyping

Módulo Criptográfico AES-GCM Operando a 100 Gbit/s Para Redes de Transporte Óptico: Arquitetura, Projeto e Prototipagem em 40 nm

Autor: Eduardo Mobilon **RA:** 994155

Orientador: Prof. Dr. Dalton Soares Arantes

Data da Defesa: 24 de fevereiro de 2023

A Comissão Examinadora composta pelos membros abaixo aprovou esta tese.

Prof. Dr. Dalton Soares Arantes (Presidente)
FEEC/Unicamp

Prof. Dr. Max Henrique Machado Costa
FEEC/Unicamp

Dr. Kayol Soares Mayer
FEEC/Unicamp

Dr. João Batista Rosolem
CPQD

Profa. Dra. Monica de Lacerda Rocha
EESC/USP

A Ata de Defesa, com as respectivas assinaturas dos membros da Comissão Examinadora, encontra-se no SIGA (Sistema de Fluxo de Dissertação/Tese) e na Secretaria de Pós-Graduação da Faculdade de Engenharia Elétrica e de Computação.

I dedicate this work to all of my family.

To my beloved wife and daughters, with special gratitude for their support and comprehension, to my mother, who educated me with simplicity and wisdom to righteousness and honesty, to the memory of my father, who taught me and showed the value of work, and to my brothers, who have always supported me in my goals.

ACKNOWLEDGMENTS

First and foremost, I thank God for the countless blessings, health, and opportunities granted to me and my family.

My special gratitude to my beloved wife and daughters, for all the encouragement, support, comprehension, and sacrifices throughout my dedication to this project.

To my supervisor and committee chair, Prof. Dr. Dalton Soares Arantes, for the way in which he allowed me to conduct the work and for the ever-relevant observations in all textual and content revisions. Further, I extend my gratitude to my committee members, Prof. Dr. Max Henrique Machado Costa (FEEC/Unicamp), Dr. Kayol Soares Mayer (FEEC/Unicamp), Dr. João Batista Rosolem (CPQD), and Profa. Dra. Mônica de Lacerda Rocha (EESC/USP), for their careful instructions and constructive advice.

To CPQD, for the opportunities that allowed me to build my professional career, developing and coordinating advanced and innovative technological projects.

My appreciation, at long last, to everyone who contributed directly and indirectly to the realization of this work, especially to my colleagues in the development of the 100G OTN Processor ASSP, a challenging project for the most advanced Brazilian integrated circuit at the time.

This work is part of a research and development project for an integrated circuit designed for the Brazilian telecom industry, led by the CPQD Foundation and financed by the Technical-Scientific Development Fund (FUNTEC) of the National Bank for Economic and Social Development (BNDES), and by the technical intervening company Padtec S.A.

The function of cryptographic protocols is
to minimize the amount of trust required.

Niels Ferguson

ABSTRACT

The optical transport network (OTN) technology enables transparent multiservice high-speed data transmission. All digital processing operations involving client data manipulation for digital wrapping and transport through the optical network are performed by OTN processor devices, implemented in either high-end programmable logic devices or application-specific integrated circuits. Although highly efficient for data transport and monitoring, the OTN communication channel is vulnerable to a series of attacks in the optical layer, which can lead to the extraction and/or manipulation of confidential information. Cryptography mechanisms address the information security issue in a communication link, ensuring data confidentiality. Authentication strategies, on the other hand, are used to provide integrity by allowing the detection of possible manipulations in the transmitted information. OTN systems were not designed to cope with this technology. The use of block ciphers, for instance, requires some adaptations to handle packet generation and the transport of auxiliary cryptographic information, which are needed in the data encryption and decryption processes. This work presents and describes in detail the conception, architecture design, implementation, and 40 nm silicon prototyping of an Advanced Encryption Standard with Galois/Counter Mode (AES-GCM) Cryptography Engine operating at 100 Gbit/s, specifically developed to protect high-speed OTN communication links, providing confidentiality and integrity to transmitted data.

Key Words: optical transport network; OTN; cryptography; confidentiality; integrity; encryption; decryption; AES-GCM.

RESUMO

A tecnologia de redes de transporte óptico (*Optical Transport Network* – OTN) permite a transmissão de dados multisserviços de forma transparente e em alta velocidade. Suas operações de processamento digital envolvendo manipulação de dados dos sinais clientes para empacotamento e transporte pela rede óptica são realizadas por processadores OTN, implementados em avançados dispositivos de lógica programável ou em circuitos integrados de aplicação específica. Embora altamente eficiente em suas funções de transporte e monitoração dos dados, o canal de comunicação OTN é vulnerável a uma série de ataques na camada óptica que podem permitir a extração e/ou manipulação de informações confidenciais. Mecanismos de criptografia endereçam a questão da segurança da informação em um enlace de comunicação, garantindo a confidencialidade dos dados. Estratégias de autenticação, por outro lado, são empregadas para garantir sua integridade ao permitir a detecção de eventuais manipulações na informação transmitida. Sistemas OTN não foram projetados para lidar com essa tecnologia. O uso de cifradores de bloco, por exemplo, exige algumas adaptações para tratar a geração de pacotes e o transporte de informações criptográficas auxiliares, necessárias ao processo de cifragem e decifração dos dados. Este trabalho apresenta e descreve de forma detalhada a concepção, projeto de arquitetura, implementação e prototipagem em silício na tecnologia de 40 nm de um Módulo Criptográfico utilizando o Padrão de Criptografia Avançada com Modo Galois/Contador (*Advanced Encryption Standard with Galois/Counter Mode* – AES-GCM) operando a 100 Gbit/s, desenvolvido especificamente para proteger enlaces de comunicação OTN de alta velocidade, proporcionando integridade e confidencialidade aos dados transmitidos.

Palavras Chave: rede de transporte óptico; OTN; criptografia; confidencialidade; integridade; cifragem; decifração; AES-GCM.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

Acronym	Description
AAD	Additional Authenticated Data
AES	Advanced Encryption Standard
AIS	Alarm Indication Signal
ASIC	Application Specific Integrated Circuit
ASSP	Application Specific Standard Product
ATE	Automated Test Equipment
ATM	Asynchronous Transfer Mode
ATPG	Automatic Test Pattern Generation
BER	Bit Error Rate
BERT	Bit Error Rate Tester
BIP	Bit-Interleaved Parity
BIST	Built-In Self-Test
BGA	Ball Grid Array
CBR	Constant Bit Rate
CBC	Cipher Block Chaining
CBID	Crypto Block ID
CFP	C Form-Factor Pluggable
CLI	Command-Line Interface
CPID	Crypto Packet ID
CPU	Central Processing Unit
CSID	Crypto Session ID
CSKS	Crypto Session Key Selection
CTS	Clock Tree Synthesis
DES	Data Encryption Standard
DFT	Design for Verification
DH	Diffie–Hellman
DoS	Denial of Service
DRC	Design Rule Checking
DUT	Design Under Test
DWDM	Dense Wavelength Division Multiplexing
ECB	Electronic Code Book
ECDH	Elliptic Curve Diffie–Hellman
EFEC	Enhanced Forward Error Correction
ETSI	European Telecommunications Standards Institute
EVB	Evaluation Board
FA	Frame Alignment
FAS	Frame Alignment Signal
FCBGA	Flip Chip Ball Grid Array
FEC	Forward Error Correction
FIFO	First In, First Out
FIPS	Federal Information Processing Standards
FPGA	Field Programmable Gate Array
FS	Frame Start
FSM	Finite State Machine

Acronym	Description
GCC	General Communication Channel
GCM	Galois/Counter Mode
GCD	Greatest Common Divisor
GDSII	Graphic Design System II
GFP	Generic Framing Procedure
HCF	Highest Common Factor
IEEE	Institute of Electrical and Electronics Engineers
IC	Integrated Circuit
IP	Intellectual Property / Internet Protocol
ITU-T	International Telecommunication Union Telecommunication Standardization Sector
IV	Initialization Vector
JTAG	Joint Test Action Group
KDF	Key Derivation Function
LCAS	Link Capacity Adjustment Scheme
LEC	Logic Equivalence Checking
LFSR	Linear Feedback Shift Register
LOA	Loss of Authentication
LUT	Lookup Table
LVS	Layout Versus Schematic
MAC	Message Authentication Code
MCU	Microcontroller Unit
MDIO	Management Data Input/Output
MFAS	Multi-Frame Alignment Signal
MPO	Multi-Fiber Push On
MFS	Multi-Frame Start
MLM	Multi-Layer Mask
NIST	National Institute of Standards and Technology
OCh	Optical Channel
ODTN	Open and Disaggregated Transport Network
ODU	Optical Channel Data Unit
OH	Overhead
ONF	Open Networking Foundation
OPU	Optical Channel Payload Unit
OSI	Open Systems Interconnection
OTL	Optical Transport Layer
OTN	Optical Transport Network
OTP	One-Time Pad
OTU	Optical Channel Transport Unit
PCB	Printed Circuit Board
PDH	Plesiochronous Digital Hierarchy
PKC	Public-Key Cryptography
PKS	Public-Key System
PLL	Phase Locked Loop
PRBS	Pseudo-Random Binary Sequence
QKD	Quantum Key Distribution
QEC	Quantum Error Correction
QRE	Quantum-Resistant Encryption
QRNG	Quantum Random Number Generation

Acronym	Description
RES	Reserved
RTL	Register Transfer Level
RSA	Rivest–Shamir–Adleman
SBC	Single Board Computer
SDH	Synchronous Digital Hierarchy
SDN	Software-Defined Networking
SDON	Software-Defined Optical Networking
SED	Self-Encrypting Drive
SerDes	Serializer/Deserializer
SHA	Secure Hash Algorithm
SMA	Subminiature Version A
SNDL	Store Now and Decrypt Later
SNR	Signal-to-Noise Ratio
SONET	Synchronous Optical Network
SSF	Server Signal Fail
STA	Static Timing Analysis
TDM	Time Division Multiplexing
T-SDN	Transport-Software-Defined Networking
TSF	Trail Signal Fail
USB	Universal Serial Bus
UVM	Universal Verification Methodology
VCAT	Virtual Concatenation
WDM	Wavelength Division Multiplexing
XOR	Exclusive OR

CONTENTS

Chapter 1
INTRODUCTION

1.1 BACKGROUND AND PURPOSE

The ever-growing convergence of Internet video and mobile data services has been pushing optical network infrastructure to expand the available transmission capacity, using new modulation formats and increasingly higher transmission data rates [1–4]. On top of the physical media advances, optical transport network (OTN) technology [5–8] emerged as an evolution of the synchronous optical network (SONET) and synchronous digital hierarchy (SDH) systems [9], enabling carriers to fully utilize each fiber wavelength by transparently mapping client signals according to a standardized digital wrapping technique [10] in a scalable and cost-optimized way.

OTN has become the de facto technology for long-distance transport networking and has been extensively deployed across the globe, providing carrier-class protection and carrying mission-critical traffic from the edge into the metro and core of the network. The global optical transport network market size is expected to grow from $23.91 billion in 2023 to $35.18 billion in 2027, at a compound annual growth rate (CAGR) of 10.1% [11].

All the digital processing operations involving client data manipulation for wrapping and transmission over the optical network are performed by OTN processor devices, with functional logic blocks or intellectual property (IP) cores implemented in either high-end field programmable gate arrays (FPGAs) or application specific integrated circuits / standard products (ASICs/ASSPs).

Although highly efficient and error-protected [12], the OTN communication channel is insecure in terms of allowing an intruder to tap sensitive information from the link and/or manipulate and change the transmitted data. The rise of cybercrimes and the consequent costs of data breaches drive network communication security to the cutting edge of encryption technology, including quantum cryptography [13–17].

Data can be encrypted at different individual layers of the open systems interconnection (OSI) model [18] or even at multiple layers, in an approach known as layered security. In optical transport networking, the wavelength division multiplexing (WDM) technology is considered a layer 0 — an additional level in the base of the OSI model stack, dealing with physical optics.

In this context, OTN is a physical layer protocol carrying out the OSI layer 1 functions in a transport network, and the great advantage of encrypting at this layer is that it enables a transparent secure transmission at the payload level, after the information data have passed through the different upper layer protocols.

In the scope of microelectronics, many contributions have been presented with alternatives for implementing cryptographic algorithms, such as the well-known Advanced Encryption Standard with Galois/Counter Mode (AES-GCM) [19–24]. However, because of the data block processing nature of this algorithm, its use in OTN systems requires adaptations such as a packet generation mechanism, ensuring data segmentation and aggregation, as well as a transport container for cryptographic auxiliary information.

This work presents the development of a 40 nm silicon-prototyped architecture for a 100 Gbit/s AES-GCM Cryptography Engine solution. The design was specifically conceived for OTN cryptosystems (OSI layer 1), with emphasis on systemic aspects addressing important issues such as the generation of cryptographic packets (with OTN frame aggrupation and insertion of the encryption overhead), the composition of the initialization vector sequence (concatenating four distinct fields with specific functions), and the transport of the encryption overhead within the OTN infrastructure (using reserved fields of the OTN frames).

Additional features designed and implemented in this work are also covered, such as an optional scrambling of the encryption overhead data for enhanced security, a synchronization and hitless cipher key change strategy with a protection mechanism based on forward error correction (with no acknowledgment signaling), the retention of the received cryptographic packets until authentication confirmation, and the generation of programmed consequent actions as a result of user selected triggering events (fail signals, errors, authentication mismatching, etc.).

The register transfer level (RTL) design of the proposed solution was fully verified by simulations, and the implemented IP core was integrated into an 8M gate 40 nm 100 Gbit/s OTN processor ASSP developed by CPQD [25] for the Brazilian telecom industry, integrating third-party mixed-signal solutions with previously developed functional logic blocks [26], enabling the establishment of secure optical communication links.

1.2 THESIS OUTLINE

Chapter 2 presents introductory concepts related to optical transport and encryption technologies, including some relevant mathematical background. Key elements of the hierarchical construction of OTN frames and the main definitions relating to encryption, authentication, and symmetric/asymmetric cryptosystems are discussed, as well as some details of the AES algorithm and modes of operation, such as the Galois/Counter Mode. Quantum cryptography, quantum computing threats, and post-quantum cryptography are also covered.

The architecture of the developed OTN cryptographic system is described in Chapter 3, with emphasis on the concepts of cryptographic session, block, and packet, as well as details about the role of the hardware and software layers, the structure of the encryption overhead framing, the error or failure recovery mechanisms, and the session management.

Chapter 4 presents the 100G AES-GCM Cryptography Engine, with its features, characteristics, operation modes, configuration procedures, hardware functional architecture, partitioning of the logic building blocks, and implementation details.

The design, verification, and 40 nm silicon prototyping of the proposed solution, integrated into a 100G OTN Processor developed by CPQD, is shown in Chapter 5.

Chapter 6 brings the conclusions and final remarks about the conducted work, highlighting the main contribution and suggestions for future research.

The interested reader is referred to Appendix A for a complete description of the 100G AES-GCM Cryptography Engine configuration register map, to Appendix B for an explanation of the digital integrated circuit design flow used by CPQD, and finally to Appendix C for the entire RTL code listing of one of the constituting logic blocks of the complete cryptographic solution.

1.3 AUTHOR CONTRIBUTIONS

This work presents and describes in detail a system and logic architecture of an AES-GCM Cryptography Engine operating at 100 Gbit/s, conceived and designed for use in OTN communication links originally not prepared to cope with a block cipher.

The topic is of great relevance, and the information presented, including some design ideas, provides a solid foundation for the development of similar applications in the areas of both cryptography and OTN systems or even of a different nature in the scope of microelectronics.

1.4 SUBJECT RELATED AUTHOR'S WORKS

Several author's works are directly related to the 100G AES-GCM Cryptography Engine development, among which the following stand out.

1.4.1 PATENT

Deposited at the Brazilian National Institute of Industrial Property (INPI) on behalf of CPQD Foundation and Padtec S.A.:

- Process number: BR 10 2016 0112605;
- Deposit date: 18/05/2016;
- Title: Hardware Method and Architecture for Implementing AES-GCM Encryption in Optical Transport Networks (*Método e Arquitetura de Hardware para Implementação de Criptografia AES-GCM em Redes de Transporte Óptico*);
- Inventors: Eduardo Mobilon and Rodolfo Soares Caproni.

1.4.2 JOURNAL PUBLISHED PAPER

Eduardo Mobilon and Dalton Soares Arantes, "100 Gbit/s AES-GCM Cryptography Engine for Optical Transport Network Systems: Architecture, Design and 40 nm Silicon Prototyping". *Microelectronics Journal*, vol. 116, 105229, ISSN 0026-2692, 2021. https://doi.org/10.1016/j.mejo.2021.105229.

1.4.3 CONFERENCE PROCEEDING PAPERS

E. Mobilon, R. Bernardo, and L. R. Monte, "100 Gbit/s Optical Transport Network 40 nm Test Chip Design and Prototyping". *SBMO/IEEE MTT-S International Microwave and Optoelectronics Conference (IMOC)*, Águas de Lindóia, pp. 1–5, 2017. https://doi.org/10.1109/IMOC.2017.8121108;

A. Salvador, D. Carvalho, C. Nakandakare, E. Mobilon, J. C. de Oliveira, and D. S. Arantes, "100 Gbit/s FEC for OTN Protocol: Design Architecture and Implementation Results". *International Telecommunications Symposium (ITS)*, São Paulo, Brazil, pp. 1–5, 2014. https://doi.org/10.1109/ITS.2014.6947951;

R. Bernardo, A. H. Salvador, E. Mobilon, L. R. Monte, S. Boisclair, and A. Warshawsky, "Design and FPGA Implementation of a 100 Gbit/s Optical Transport Network Processor". *23rd International Conference on Field Programmable Logic and Applications*, Porto, Portugal, pp. 1–4, 2013. https://doi.org/10.1109/FPL.2013.6645601.

1.4.4 COMPUTER PROGRAM REGISTRATION

Deposited at the Brazilian National Institute of Industrial Property (INPI) on behalf of CPQD Foundation:

- Process number: BR 51 2017 000827-2;
- Creation date: 14/06/2016;
- Expedition date: 01/08/2017;
- Title: CPQD3407 – AES-256 GCM Cryptographic Module for OTN Protocol (*Módulo Criptográfico AES-256 GCM para Protocolo OTN*) – Crypto4OTN – V.05.09.00;
- Language: System Verilog;
- Authors: Eduardo Mobilon and Arley Henrique Salvador.

Chapter 2

OPTICAL TRANSPORT TECHNOLOGIES AND CRYPTOGRAPHY

This chapter briefly describes optical transport technologies, highlighting their frame structure and protection mechanisms. Security threats in optical networks are also overviewed, with a few considerations regarding software-defined and disaggregated optical networks. Cryptographic systems are then presented after some relevant mathematical background, starting with the main concepts related to data encryption and authentication, briefing over their historical evolution, and stepping into quantum cryptography. The final sections show some of the cryptographic modes of operation, with emphasis on the most relevant in the scope of this work, the quantum computing threats endangering classical encryption algorithms, as well as their quantum-resistant counterparts of post-quantum cryptography.

2.1 INTRODUCTION

The history of transport network technologies begins with the interconnection of telegraph and telephone exchanges, providing communications between regional, national, and international centers through analog networks using coaxial cables [6, 27].

Digital transmission systems using the time division multiplexing (TDM) technique emerged in the 1970s, significantly increasing transmission distances as a result of the higher signal-to-noise ratio (SNR). This technique was conceived to transport voice over telephone networks using plesiochronous digital hierarchy (PDH) technology. It is organized into several levels (orders), and the aggregated signal of each order is formed by grouping (with sequential byte interleaving) four tributary signals from the previous level.

The main difficulty of PDH systems is the insertion and extraction (add/drop) of these tributaries in intermediate network sections due to channel interleaving. This process requires demultiplexing the aggregated beam, which makes this an inflexible and expensive standard.

With the advent and evolution of semiconductor lasers and optical fibers, a more advanced and still PDH-compatible system evolved from the synchronous optical network (SONET), created at Bellcore (Bell Communications Research, Inc.), to its standardization by the ITU-T in the 1980s as the new synchronous digital hierarchy (SDH) [9].

SDH technology has brought a significant increase in the transmission rates of transport networks, integrating them with optical transmission systems, in addition to the possibility of broad and efficient management, making them highly reliable. Its interfaces were designed for the transport of legacy telephone traffic, but with the emergence and popularization of packet networks (especially Ethernet), SDH systems were adapted to carry this type of traffic. New mechanisms were then created, such as virtual concatenation (VCAT), link capacity adjustment scheme (LCAS), and generic framing procedure (GFP).

Because of the development and maturity of optical technologies and the exponential growth of data traffic (resulting from Internet services), it has become essential to improve the integration between data networks and optical networks. Thus, in the 1990s, studies began on a new standard to maximize the efficiency of transmission systems — the optical transport network (OTN).

This system defines a digital envelope to encapsulate numerous protocols such as SDH, ATM, and Ethernet and transport them efficiently and securely across optical networks using wavelength division multiplexing (WDM) technology [28].

Optical transport networks handle circuit switching, establishing a dedicated communication path between two nodes and thus guaranteeing low latency and high bandwidth. These circuit links are set up with a dedicated and fixed capacity to accommodate a deterministic peak traffic, being then underutilized in some circumstances.

Packet switching, on the other hand, creates a connectionless network with messages being segmented into packets that are dynamically routed between the source and destination nodes, being the primary basis for transferring data across computer networks. Resources are allocated as needed, and the network channels are occupied only when packets are transmitted.

Due to the explosive growth in data traffic driven by video and multimedia applications, cloud services, and the emergence of high-speed mobile Internet access technologies, packet traffic dominates metropolitan area networks.

In this context, the multi-protocol label switching (MPLS) technology [29] routes and delivers data packets to their destinations quickly and efficiently based on labels rather than network addresses, working in conjunction with the Internet protocol (IP) and providing class-specific traffic engineering features.

Since its inception, OTN has evolved and optimized to support Ethernet traffic and flexible packet technologies. Typically, multi-layer network architectures are used with an MPLS over OTN approach, where the label-switching routers are attached to optical cross-connects, which are then interconnected by point-to-point WDM links creating end-to-end circuit-switched optical channels called lightpaths. In this way, Ethernet and IP/MPLS traffic are aggregated to be efficiently and reliably transported across the OTN backbone.

Beyond data transport and administration functionalities, information security services are crucial in communication systems, including modern optical telecommunications. The most important ones can be grouped as follows [30]:

- Confidentiality: protection of data from unauthorized disclosure;
- Integrity: assurance that data are received exactly as transmitted;
- Authentication: assurance of the communicating party identity;
- Nonrepudiation: protection against denial by one of the parties of having participated in all or part of the communication.

These elementary security functions are covered in the field of cryptography, originally defined as the art and science of encryption [31], but today featuring a much broader scope. Cryptography has a long and fascinating history, starting back thousands of years ago, changing the outcome of both world wars and being ubiquitously available in current communication and storage systems. Throughout its existence, it has transitioned from an ancient art to a modern principle-driven science.

2.2 OPTICAL TRANSPORT NETWORKS

The OTN protocol is an optimum converged transport technology for both legacy and emerging client signals, enabling high-speed transparent multi-service data transport with additional functionalities of multiplexing, switching, management, and supervision. It also incorporates a forward error correction (FEC) code [32–34] that allows greater reach between optical network nodes and/or higher bit rates on the same fiber.

In the electrical domain, an OTN signal is obtained by a hierarchical frame construction, with the client signal being mapped into progressively larger data structures, which make up the optical channel payload unit (OPU), the optical channel data unit (ODU), and, finally, the optical channel transport unit (OTU), as illustrated in Figure 2.1.

This TDM hierarchy allows for lower-order ODU payloads to be combined and/or switched to add/drop lower rate signals into/from higher-order ODU frames. The OTN switching technology [35, 36] supports new services, virtualizing the optical network bandwidth between nodes to the ODU bit rate granularities of ODU0 (1.25 Gbit/s), ODU1 (2.5 Gbit/s), ODU2/2e (10 Gbit/s), ODU3 (40 Gbit/s), or ODU4 (100 Gbit/s).

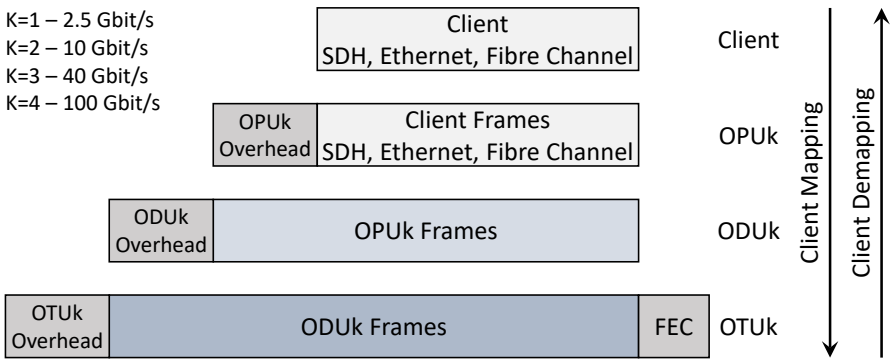


Figure 2.1 – OTN signal hierarchical frame construction.

Due to the process or method of mapping (encapsulating) and transparently transporting client signals within the payload of a larger frame over an optical network, the OTN system is also called “digital wrapper” [5].

The resulting G.709 OTN frame structure, shown in Figure 2.2, is made up of four lines of 4080 bytes, with reserved areas for the frame alignment (FA) word, the different layer overheads (OH), the payload data, and the FEC redundancy bytes [34].

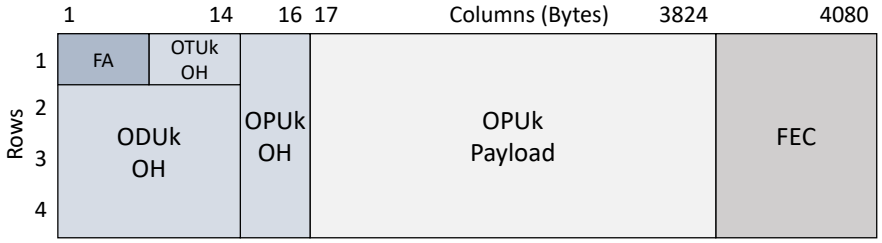


Figure 2.2 – G.709 OTN frame structure.

The 7-byte frame alignment word comprises two important signals: a 6-byte frame alignment signal (FAS), defined in row 1 and columns 1 to 6 of the OTN frame as a fixed pattern sequence (0xf6f6f6282828) and a single byte multi-frame alignment signal (MFAS), defined in row 1 and column 7, with its value being incremented after each frame providing a 256-frame multi-frame.

The OTU4 level was incorporated into the OTN hierarchy from the 2009 revision of the ITU-T G.709 standard [10], motivated by IEEE initiatives to develop a 100 Gbit/s Ethernet signal standard. The technological and economic efforts to maintain the rate of the OTU4 level equivalent to four times that of the OTU3 (as in the previous levels) were not justified at the time, since the only client signal data rate was 100 Gbit/s [37].

The bit manipulation and processing operations necessary for the hierarchical construction of OTN frames, from the client signal mapping to the calculation of the redundancy bytes of the error correcting code, are performed by digital circuits implemented in ASICs or FPGAs. Typically, the entire process is divided into distinct functional logic blocks dedicated to mapping, multiplexing, framing, and FEC, among others.

Data synchronization between the numerous blocks that make up a given OTN solution is guaranteed by digital signals generated (or extracted) from the frame alignment markers (FAS and MFAS). Thus, usually a digital frame start (FS) pulse precedes the first bit of the first line of an OTN frame. Similarly, a multi-frame start (MFS) pulse precedes the first bit of the first line of the first OTN frame (MFAS = 0) of a multi-frame.

In the optical domain, dense wavelength division multiplexing (DWDM) technology [28] is adopted at the physical layer to allow for the use of all available bandwidth in the optical fiber. As illustrated in Figure 2.3, each OTUk frame is mapped into a wavelength, which constitutes the optical channel (OCh) [5, 7].

In optical networks, OTN-compatible line cards can be commonly found either in the network nodes, as terminal modules called transponders that map/demap client signals, or along the links, as regenerator units used in long reach communication channels, also allowing for client data switching and aggregation.

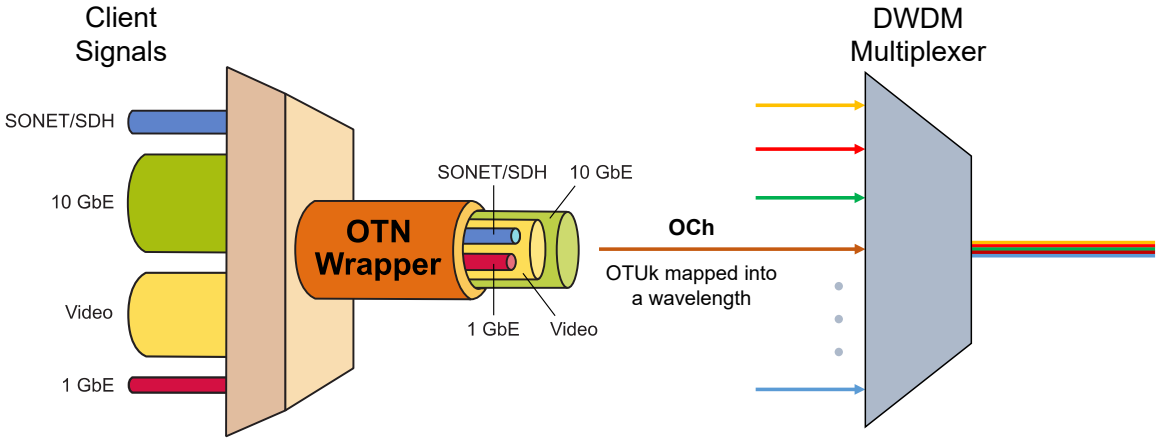


Figure 2.3 – Client signals being mapped into an OTUk structure transported in a single wavelength optical channel.

The standard OTN FEC, based on the Reed-Solomon RS (255, 239) code [34], or even stronger enhanced FEC (EFEC) codes [38, 39] ensure protection against communication errors caused by transmission impairments in the optical fiber, categorized into linear (attenuation, noise, and dispersion) [40] and nonlinear (four-wave mixing, self-phase modulation, and cross-phase modulation) [41] effects. This error control strategy also brings early warning capabilities, since the analysis of the number of corrected errors can assist on the early detection of some network element degradation.

Although real-time optical power analysis can also be used for fault detection and localization [42], including some attempts to tap the communication link, only a cryptography solution can address current network stringent security demands.

2.3 Security Threats in Optical Networks

The optical layer of a transport network is vulnerable to a number of attacks [43, 44]. Information security threats can be classified into the following main categories [16]:

- Espionage, characterized as an attempt to gain unauthorized access to transported data without disrupting network traffic;

- Disturbance of the optical channel, aiming at degrading or completely interrupting the communication service in the optical layer through cuts in the optical fibers, insertion of interfering signals (reducing the signal-to-noise ratio), etc.;
- Denial of service (DoS) attack, through the flooding of packets on the network, causing loss of service or degradation in data transfer rate;
- Network intrusion, allowing the attacker to exercise control over resources and manipulate the network operation;
- Quantum attacks, resulting from the emerging technology of quantum computing, aimed at breaking the keys used in cryptographic algorithms.

Regarding data confidentiality, the biggest concern is the possibility of espionage through optical fiber tapping. This can be done intrusively, with the fiber cut and a reconnection using a signal splitter, or non-intrusively, through techniques that allow extracting a part of the light without generating traffic disturbances. One of the main non-intrusive tapping methods consists of bending the fiber, causing some of the light to escape through its cladding (due to the angle of incidence being smaller than the critical angle). This can be easily done using a clip-on optical coupler specially designed for this purpose [45].

On top of the physical media threats, network control, management, and operation can also be compromised, bringing new challenges to maintaining the security of the entire system.

Traditional packet networks have distributed control and transport protocols running on their routers and switches, making them complex and hard to manage. High-level network policies are implemented by the configuration of each device using low-level and vendor-specific commands. The control and data planes, responsible for handling and forwarding network traffic, are also embedded in the networking devices, reducing the flexibility of the infrastructure operation and evolution.

Software-defined networking (SDN) [46] has emerged as a new paradigm that became a highly adopted technology in packet networks, separating the control and data planes and allowing the control operations to be implemented in a logically centralized network controller.

These same architectural principles were brought to single-domain optical networks, leveraging the flexibility of SDN control into software-defined optical networking (SDON)

[47], providing network services over an underlying high-capacity optical infrastructure with specific transmission and switching characteristics. Transport-SDN (T-SDN) [48] also came out as a natural evolution into multiple domain optical networks, with more stringent demands for the networking element orchestration and control, mainly due to new physical attributes such as wavelength continuity, non-linear effects, signal-to-noise ratio, and dispersion compensation.

Recent efforts are now being directed towards the disaggregation of optical communication equipment, breaking the vertically integrated (single-vendor) solutions with the approach of using open application program interfaces (APIs) and open-source software to allow transparent communication with vendor-independent equipment for network configuration and management workflows. Open and Disaggregated Transport Network (ODTN) project [49] is an Open Networking Foundation (ONF) platform and operator-led initiative to build datacenter interconnections using multi-vendor disaggregated optical transport equipment orchestrated and managed by an SDN controller, starting with point-to-point and expanding to multi-point networks.

Despite the numerous advantages of all these SDN-based concepts applied to network management and control, new flanks of cyber attacks are opened up. The paradigm of a logically centralized network controller allows for programmability, automation, and run-time deployment of security procedures and policies, but it may, in fact, become a single point of failure, leading to potentially catastrophic consequences for the whole network.

Many security challenges and threats in SDN can be grouped according to the plane being attacked (application, control, or data plane) [50], and all the respective security solutions are also implemented in the software layer.

Hardware encryption mechanisms remain a good countermeasure for securing the data plane traffic, and, in the context of the evolving T-SDN and ODTN systems, they must comply with the open APIs necessary for transparent communication with the SDN-based controller. Additionally, with the segregation of the control and data planes, it is also important to monitor and protect the communication channel between the separated layers of the SDN-based optical network, applying the already studied security solutions [50] to guarantee the expected network-wide security.

2.4 MATHEMATICAL BACKGROUND FOR CRYPTOGRAPHY

This section brings a quick review of some mathematical concepts that help in understanding cryptographic algorithms and systems. Further details can be found in any textbook on number theory [51, 52] and abstract algebra or even in introductory sections of cryptography books [30, 31, 53, and 54].

2.4.1 NUMBER THEORY

Number theory is a branch of mathematics concerned with the study of the set of positive integers, often called natural numbers, with a wide range of applications in cryptography.

2.4.1.1 DIVISIBILITY

A nonzero b divides a if $a = mb$ for some m, where a, b, and m are integers. It is denoted by $b \mid a$, indicating that b is a divisor of a with no remainder on division.

Some simple properties of divisibility for integers are shown below:

- If $a \mid 1$, then $a = \pm 1$;
- If $a \mid b$ and $b \mid a$, then $a = \pm b$;
- Any $b \neq 0$ divides 0;
- If $a \mid b$ and $b \mid c$, then $a \mid c$;
- If $b \mid g$ and $b \mid h$, then $b \mid (mg+nh)$, for arbitrary integers m and n.

Given any positive integer b and any nonnegative integer a, dividing a by b results in an integer quotient q and an integer remainder (or residue) r that conform to the following relationship:

$$a = qb + r \quad 0 \leq r < b; q = \lfloor a/b \rfloor \tag{2.1}$$

where $\lfloor x \rfloor$ is the largest integer less than or equal to x. Equation (2.1) expresses a theorem but, by tradition, is referred to as the division algorithm.

2.4.1.2 PRIME NUMBERS

An integer $p > 1$ is called a prime number if its only positive divisors are 1 and p. Any integer greater than 1 that is not prime is called a composite number.

The fundamental theorem of arithmetic, also called the unique factorization theorem, states that every positive integer greater than 1 can be written in exactly one way (apart from rearrangement) as the product of one or more primes.

For example, $15 = 3 \times 5$, $255 = 3 \times 5 \times 17$, and $60 = 2 \times 2 \times 3 \times 5$.

The greatest common divisor (GCD) of two or more integers (not all zero), also called the highest common factor (HCF), is the largest positive integer that divides each of them. Two numbers, a and b, are called coprime, relatively prime, or mutually prime if $\gcd(a, b) = 1$.

For small numbers, the GCD is easy to calculate by factoring them and finding their HCF. As an example, for $a = 84$ and $b = 30$, factoring yields:

$a = 84 = 2 \times 2 \times 3 \times 7$ $\quad$ $b = 30 = 2 \times 3 \times 5$

The GCD is then the product of all common prime factors — $\gcd(84, 30) = 2 \times 3 = 6$.

2.4.1.3 THE EUCLIDEAN ALGORITHM

The most efficient method known for finding the greatest common divisor of two numbers is called the Euclidean algorithm, which consists of doing a sequence of divisions with a remainder r until $r = 0$.

From Equation (2.1) it follows that any common divisor of b and r is also a divisor of a. Moreover, any common divisor of a and b is also a divisor of r, since $r = a - qb$. Accordingly, the common divisors of a and b are the same as the ones of b and r.

The problem of finding the common divisors of a and b is then reduced to the same problem for the numbers b and r, which are respectively less than a and b. The essence of the also called Euclid's algorithm lies in the repetition of this argument, finding the GCD of two integers a and b by recursively reducing the operands.

As an example, for the calculation of the $\gcd(36, 132)$ the algorithm runs as follows:

> The first step is to divide 132 by 36, which gives a quotient of 3 and a remainder of 24, written as $132 = 3 \times 36 + 24$. The next step is to take 36 and divide it by the remainder 24 from the previous step, giving $36 = 1 \times 24 + 12$. Repeating this process,

when dividing 24 by 12 the remainder becomes zero or 24 = 2 × 12 + 0. The remainder from the previous step is then the greatest common divisor of the original two numbers, so gcd(36, 132) = 12.

Applying the Euclidean algorithm to large numbers shows how it is far more efficient than factorization to compute the GCD. Table 2.1 shows the algorithm steps for the calculation of gcd(1160718174, 316258250) = 1078.

1160718174	=	3 × 316258250	+	211943424
316258250	=	1 × 211943424	+	104314826
211943424	=	2 × 104314826	+	3313772
104314826	=	31 × 3313772	+	1587894
3313772	=	2 × 1587894	+	137984
1587894	=	11 × 137984	+	70070
137984	=	1 × 70070	+	67914
70070	=	1 × 67914	+	2156
67914	=	31 × 2156	+	**1078**
2156	=	2 × 1078	+	0

Table 2.1 – Euclidean algorithm steps for the calculation of gcd(1160718174, 316258250) = 1078.

2.4.1.4 Modular Arithmetic and Congruence

If a is an integer and b is a positive integer, a mod b is defined as the remainder when a is divided by b. The integer b is called the modulus. Thus, for any integer a, the Equation (2.1) can be rewritten as follows:

$$a = qb + r \quad 0 \leq r < b; q = \lfloor a/b \rfloor$$

$$a = \lfloor a/b \rfloor \times b \ + (a \ mod \ b) \tag{2.2}$$

Two integers a and b are said to be congruent modulo n if (a mod n) = (b mod n). This is written as $a \equiv b \pmod{n}$, with the parentheses indicating that (mod n) applies to the entire equation and not just to the right-hand side. When a is congruent to b with respect to the modulus n, a and b differ by a multiple of n or, in other words, n divides ($a - b$). It follows that if $a \equiv 0 \pmod{n}$, then $n \mid a$. Some examples are given below:

11 mod 7 = 4 → 11 = 7 × 1 + 4
–11 mod 7 = 3 → –11 = 7 × (–2) + 3
73 ≡ 4 (mod 23)
21 ≡ –9 (mod 10)

Congruences have the following properties:

- $a \equiv b \pmod{n}$ if $n \mid (a - b)$;
- $a \equiv b \pmod{n}$ implies $b \equiv a \pmod{n}$;
- $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$ imply $a \equiv c \pmod{n}$.

The modulo operator (mod n) maps all integers into the set of integers $\{0, 1, \ldots, (n - 1)\}$. Modular arithmetic can be performed within this finite set, with the following properties:

- $[(a \bmod n) + (b \bmod n)] \bmod n = (a + b) \bmod n$;
- $[(a \bmod n) - (b \bmod n)] \bmod n = (a - b) \bmod n$;
- $[(a \bmod n) \times (b \bmod n)] \bmod n = (a \times b) \bmod n$.

These are some examples:

$11 \bmod 8 = 3$
$15 \bmod 8 = 7$
$[(11 \bmod 8) + (15 \bmod 8)] \bmod 8 = 10 \bmod 8 = 2$
$(11 + 15) \bmod 8 = 26 \bmod 8 = 2$
$[(11 \bmod 8) - (15 \bmod 8)] \bmod 8 = -4 \bmod 8 = 4$
$(11 - 15) \bmod 8 = -4 \bmod 8 = 4$
$[(11 \bmod 8) \times (15 \bmod 8)] \bmod 8 = 21 \bmod 8 = 5$
$(11 \times 15) \bmod 8 = 165 \bmod 8 = 5$

One familiar example of modular arithmetic is the "clock arithmetic". In spite of new hours being added, the result always wraps around 12 (or 24, depending on the clock style).

2.4.1.5 FERMAT'S AND EULER'S THEOREMS

Fermat's theorem states that if p is prime and a is a positive integer not divisible by p, then:

$$a^{p-1} \equiv 1 \ (mod\ p) \qquad (2.3)$$

These are some examples for $a = 7$ and $p = 19$:

$7^2 = 49 \equiv 11 \pmod{19}$
$7^8 \equiv 49 \equiv 11 \pmod{19}$
$a^{p-1} = 7^{18} = 7^{16} \times 7^2 \equiv 7 \times 11 \equiv 1 \pmod{19}$
$7^4 \equiv 121 \equiv 7 \pmod{19}$
$7^{16} \equiv 121 \equiv 7 \pmod{19}$

A useful alternative form of Fermat's theorem states that if p is a prime and a is a positive integer, then:

$$a^p \equiv a \ (mod\ p) \tag{2.4}$$

The first form of this theorem in Equation (2.3) requires that a be relatively prime to p. This does not hold for Equation (2.4).

Two examples are:

$p = 5$ and $a = 3 \rightarrow a^p = 3^5 = 243 \equiv 3 \pmod 5 = a \pmod p$
$p = 5$ and $a = 10 \rightarrow a^p = 10^5 = 100000 \equiv 10 \pmod 5 = a \pmod p$

An important quantity in number theory is referred to as Euler's totient function, written as $\varphi(n)$ and defined as the number of positive integers less than n and relatively prime to n. By convention, $\varphi(1) = 1$, although it has no meaning.

For example, 37 is a prime number and so all of the positive integers from 1 through 36 are relatively prime to 37. Consequently, $\varphi(37) = 36$. To determine $\varphi(35)$, on the other hand, all of the positive integers less than 35 that are relatively prime to it must be listed. As shown below, since there are 24 numbers on this list, $\varphi(35) = 24$.

1, 2, 3, 4, 6, 8, 9, 11, 12, 13, 16, 17, 18, 19, 22, 23, 24, 26, 27, 29, 31, 32, 33, 34

As observed in the previous example, for a prime number p, $\varphi(p) = p - 1$.

Now for two prime numbers p and q, with $p \neq q$ and $n = pq$,

$$\varphi(n) = \varphi(pq) = \varphi(p) \times \varphi(q) = (p-1) \times (q-1) \tag{2.5}$$

As an example, $\varphi(21) = \varphi(3) \times \varphi(7) = (3 - 1) \times (7 - 1) = 2 \times 6 = 12$.

Euler's theorem states that for every a and n that are relatively prime:

$$a^{\varphi(n)} \equiv 1 \ (mod\ n) \tag{2.6}$$

Two examples are:

$a = 3$, $n = 10$, and $\varphi(10) = 4 \rightarrow a^{\varphi(n)} = 3^4 = 81 \equiv 1 \pmod{10} = 1 \pmod n$
$a = 2$, $n = 11$, and $\varphi(11) = 10 \rightarrow a^{\varphi(n)} = 2^{10} = 1024 \equiv 1 \pmod{11} = 1 \pmod n$

A useful alternative form of this theorem is also available:

$$a^{\varphi(n)+1} \equiv a \ (mod\ n) \qquad (2.7)$$

Again, as in the case of Fermat's theorem, the first form of Euler's theorem in Equation (2.6) requires that *a* be relatively prime to *n*. This does not hold for Equation (2.7).

2.4.2 Abstract Algebra and Finite (Galois) Fields

Fields are a subset of a larger class of algebraic structures called rings, which are in turn a subset of the larger class of groups, all of them being fundamental elements of a branch of mathematics known as abstract algebra. Finite fields are a subset of fields, consisting of those with a finite number of elements.

These structures are sets of elements which can be algebraically combined by operations subjected to specific rules that define the nature of the set. By convention, the notation for these operations is usually the same as the ones for addition and multiplication on ordinary numbers.

2.4.2.1 Groups

A group *G* is a set of elements with a binary operation denoted by • that associates to each ordered pair (*a*, *b*) of elements in *G* an element (*a* • *b*) in *G*, such that the following axioms are obeyed. The operator • is generic and can refer to addition, multiplication, or some other mathematical operation.

(A1) Closure: if *a* and *b* belong to *G*, then *a* • *b* is also in *G*;
(A2) Associative: *a* • (*b* • *c*) = (*a* • *b*) • *c* for all *a*, *b*, *c* in *G*;
(A3) Identity element: there is an element *e* in *G* such that
a • *e* = *e* • *a* = *a* for all *a* in *G*;
(A4) Inverse element: for each *a* in *G*, there is an element *a*' in *G* such that
a • *a*' = *a*' • *a* = *e*.

A group is called abelian or commutative if it satisfies the following additional condition:

(A5) Commutative: *a* • *b* = *b* • *a* for all *a*, *b* in *G*.

Examples of abelian groups include the set of integers (positive, negative, and 0) under addition as well as the set of nonzero real numbers under multiplication.

A cyclic group G has every element being a power a^k (with k integer) of some fixed element $a \in G$. This element a is said to generate the group or to be a generator of G. A cyclic group is always abelian and may be finite or infinite.

2.4.2.2 RINGS

A ring R is a set of elements with two binary operations, called addition and multiplication, such that for all a, b, c in R the following axioms are obeyed.

(A1–A5) R is an abelian group with respect to addition, satisfying axioms A1 through A5. For the case of an additive group, the identity element is denoted as 0 and the inverse of a as $-a$;

(M1) Closure under multiplication: if a and b belong to R, then ab is also in R;

(M2) Associativity of multiplication: $a(bc) = (ab)c$ for all a, b, c in R;

(M3) Distributive laws: $a(b + c) = ab + ac$ for all a, b, c in R and
$(a + b)c = ac + bc$ for all a, b, c in R.

A ring is called commutative if it satisfies the following additional condition:

(M4) Commutativity of multiplication: $ab = ba$ for all a, b in R.

An example of commutative ring is the set of integers with their standard addition and multiplication operations.

An integral domain is a commutative ring that obeys the following axioms.

(M5) Multiplicative identity: there is an element 1 in R such that
$a1 = 1a = a$ for all a in R.

(M6) No zero divisors: if a and b belong to R and $ab = 0$, then either $a = 0$ or $b = 0$.

2.4.2.3 FIELDS

A field F is a set of elements with two binary operations, called addition and multiplication, such that for all a, b, c in F the following axioms are obeyed.

(A1–M6) F is an integral domain, satisfying axioms A1 through A5 and M1 through M6;

(M7) Multiplicative inverse: for each a in F, except 0, there is an element a^{-1} in F such that $a(a^{-1}) = (a^{-1})a = 1$.

The rational, real, and complex numbers are usual examples of fields.

A field can also be defined as a set which contains an additive and a multiplicative group, such that the following conditions hold:

- F forms an abelian group with respect to addition;
- The nonzero elements of F form an abelian group with respect to multiplication;
- The distributive law holds, that is, for all a, b, c in F: $a(b + c) = ab + ac$ and $(a + b)c = ac + bc$.

Finite fields, also called Galois fields after Évariste Galois who studied them in the nineteenth century, are algebraic structures with a finite number of elements. This number defines the order or cardinality of the field.

An important theorem states that a field with order m only exists if m is a prime power, i.e., $m = p^n$, for some positive integer n and prime integer p, which is called the characteristic of the finite field.

As a consequence, there are, for instance, finite fields with 11 elements, with 81 elements (since $81 = 3^4$), or with 256 elements (since $256 = 2^8$). On the other hand, there is no finite field with 12 elements since $12 = 2^2 \times 3$, which is not a prime power.

Finite fields of order p^n, generally written $GF(p^n)$, play a crucial role in many cryptographic algorithms.

Prime fields are the ones with a prime order, i.e., with $n = 1$. Elements of a prime field $GF(p)$ can be represented by integers $0, 1, \ldots, p - 1$, and the two field operations are integer addition and multiplication modulo p.

The arithmetic in a prime field follows the rules for integer rings:

- Addition and multiplication are done modulo p;

- The additive inverse of any element a is given by $a + (-a) \equiv 0 \bmod p$;
- The multiplicative inverse of any nonzero element a is defined as $a(a^{-1}) = 1$.

Table 2.2 shows the arithmetic modulo 2 for the smallest finite field $GF(2)$, with the elements {0, 1}, which is ubiquitous in computer science technology and, consequently, important for the advanced encryption standard algorithm implementation.

Addition

+	0	1
0	0	1
1	1	0

Multiplication

×	0	1
0	0	0
1	0	1

Table 2.2 – Arithmetic modulo 2 for the finite field GF(2).

It can be noticed that the modulo 2 addition is equivalent to an XOR logic operation, while the multiplication is equivalent to an AND.

When the order p^n of a finite field is not a prime number, the addition and multiplication operations cannot be represented by an arithmetic modulo p^n. These so-called extension fields require a different notation for their elements, as well as different arithmetic rules. The elements of an extension field are represented by polynomials, and computations in such fields are done by polynomial arithmetic.

In extension fields $GF(2^n)$, elements are represented by polynomials with coefficients in $GF(2)$ with a maximum degree of $n - 1$. In the field $GF(2^8)$, used in the advanced encryption standard, each element $A \in GF(2^8)$ is then represented as:

$$A(x) = a_7x^7 + \cdots + a_1x + a_0, \quad a_i \in GF(2) = \{0, 1\} \tag{2.8}$$

A polynomial in $GF(2^n)$ can be uniquely represented by its n binary coefficients. In this way, each element of the set of all the $2^8 = 256$ polynomials that makes up the finite field $GF(2^8)$ can be represented by an 8-bit number.

2.4.3 Polynomial Arithmetic in Extension Fields

Addition and subtraction in extension fields are performed by the respective standard polynomial operations, adding and subtracting coefficients with equal powers of x in the underlying finite field $GF(2)$.

Given two elements of an extension field, $A(x)$ and $B(x) \in GF(2^n)$, their sum and difference are computed as follows:

$$C(x) = A(x) + B(x) = \sum_{i=0}^{n-1} c_i x^i, \quad c_i = a_i + b_i \; mod \; 2 \tag{2.9}$$

$$C(x) = A(x) - B(x) = \sum_{i=0}^{n-1} c_i x^i, \quad c_i = a_i - b_i = a_i + b_i \; mod \; 2 \tag{2.10}$$

Addition and subtraction modulo 2 are the same operations, equivalent to a bitwise XOR logic function. The example below shows the sum or difference of two elements of $GF(2^8)$:

$$\begin{array}{rcl} A(x) & = & x^7 + x^6 + x^4 \quad + \quad 1 \\ B(x) & = & \qquad\qquad x^4 + x^2 + 1 \\ \hline C(x) & = & x^7 + x^6 \quad + \quad x^2 \end{array}$$

Multiplication in an extension field follows a similar approach of the respective operation in a prime field $GF(p)$, where the result is modulo p. In $GF(2^n)$, however, an irreducible polynomial is necessary for the modulo reduction. This also called prime polynomial is roughly comparable to a prime number, since it cannot be expressed as a product of two polynomials. In other words, its only factors are 1 and the polynomial itself.

According to the fundamental theorem of algebra, every non-zero, single-variable, degree n polynomial with complex coefficients has, counted with multiplicity, exactly n complex roots. This does not hold for finite fields, since they are not algebraically closed. As an example, the polynomial $P(x) = x^8 + x^4 + x^3 + x + 1$, used for multiplications in the AES algorithm, is irreducible over $GF(2^8)$.

Conversely, the polynomial $P(x) = x^4 + x^3 + x + 1$ with coefficients over $GF(2)$ is reducible since $x^4 + x^3 + x + 1 = (x^2 + x + 1)(x^2 + 1)$. Therefore, it cannot be used to construct the extension field $GF(2^4)$.

Given two elements of an extension field, $A(x)$ and $B(x) \in GF(2^n)$ and an irreducible polynomial $P(x)$, their multiplication is computed as follows:

$$C(x) = A(x) \cdot B(x) \; mod \; P(x) \tag{2.11}$$

$$P(x) = \sum_{i=0}^{n-1} p_i x^i, \quad p_i \in GF(2) \tag{2.12}$$

As an example, $A(x) = x^3 + x^2 + 1$ and $B(x) = x^2 + x$ are elements of a finite field $GF(2^4)$ with the irreducible polynomial $P(x) = x^4 + x + 1$. First, their plain product is obtained as $C'(x) = A(x) \cdot B(x) = x^5 + x^3 + x^2 + x$. Then, by (2.11), $C(x) = x^5 + x^3 + x^2 + x \bmod P(x)$, with the remainder obtained by long polynomial division as shown below.

$$\begin{array}{rl|l} & & x \\ x^4 + x + 1 & \big| & x^5 + x^3 + x^2 + x \\ & - & (x^5 + \quad x^2 + x) \\ & & 0 + x^3 \quad 0 \quad 0 \end{array}$$

The final polynomial product over the extension field $GF(2^4)$ is $C(x) = A(x) \cdot B(x) = x^3$.

With regards to the inversion operation in extension fields, the inverse A^{-1} of a nonzero element $A \in GF(2^n)$ with an irreducible polynomial $P(x)$ is defined as:

$$A^{-1}(x) \cdot A(x) = 1 \; mod \; P(x) \tag{2.13}$$

For small fields (with typically 2^{16} or fewer elements) lookup tables with the precomputed inverses of all field elements are often used. Otherwise, inverses can also be explicitly computed using the extended Euclidean algorithm.

2.4.4 One-Way and Trapdoor Functions

Given two sets, X and Y, a function from X (domain) to Y (codomain) is an assignment of an element of Y to each element of X. This relationship is declared by the notation $f: X \rightarrow Y$. For any $x \in X$, $f(x) = y$ is called the image of x under f. The image of a function, denoted by Im(f), is the set of all output values it may produce.

A function $f : X \rightarrow Y$ is one-way if $f(x)$ is easy to compute for all $x \in X$, but for a random $y \in$ Im(f) it is computationally infeasible (with known algorithms) to find any $x \in X$ such that $f(x) = y$. In other words, one-way functions are easy to compute but intractable to invert. Their existence is still an open conjecture.

Trapdoor (or trapdoor one-way) functions are a special case of one-way functions for which finding their inverse is also easy giving an extra information (the trapdoor).

Two cryptographically relevant candidates for one-way functions are the integer factorization and the discrete logarithm problems. The security of many cryptographic techniques depends upon their intractability.

2.4.4.1 INTEGER FACTORIZATION PROBLEM

As stated in Section 2.4.1.2, by the fundamental theorem of arithmetic every positive integer greater than 1 can be written in exactly one way (apart from rearrangement) as the product of one or more prime numbers.

No efficient non-quantum algorithm is known for the prime factorization of sufficiently large numbers, and many cryptographic techniques depend upon the intractability of this problem. A recent factorization experiment showed that a 240-digit (795-bit) number was factored using approximately 900 core-years of computing power [55].

The integer factorization problem presents a one-way function: multiplying two large primes is computationally easy, but factoring the resulting product is very difficult. It can also be a source of many trapdoor one-way functions when the prime factors are provided as a trapdoor (extra) information.

As an example, given two prime numbers p and q, together with their product $n = pq$, a function defined as $f(x) = x^3 \bmod n$ is relatively easy to compute but intractable to invert (i.e., to find x given n) for very large distinct prime factors. However, if p and q are provided as a trapdoor information, available algorithms can efficiently compute this modular cube.

2.4.4.2 DISCRETE LOGARITHM PROBLEM

The discrete logarithm problem is defined in the so-called cyclic groups — the algebraic structures defined in Section 2.4.2.1. Given the general expression

$$a^m \equiv 1 \ (mod\ n) \tag{2.14}$$

in which a and n are relatively prime, there is at least one integer m that satisfies the equation, namely $m = \varphi(n)$, according to Euler's theorem in Equation (2.6).

The least positive exponent *m* that satisfies Equation (2.14) is called the order of *a* (mod *n*), which also corresponds to the period length of the group generated by *a*.

As an example, the powers of 7 modulo 19 are given below.

$7^1 \equiv 7 \pmod{19}$
$7^2 = 49 = 2 \times 19 + 11 \equiv 11 \pmod{19}$
$7^3 = 343 = 18 \times 19 + 1 \equiv 1 \pmod{19}$
$7^4 = 2401 = 126 \times 19 + 7 \equiv 7 \pmod{19}$
$7^5 = 16807 = 884 \times 19 + 11 \equiv 11 \pmod{19}$

The order of 7 modulo 19 = 3, and, consequently, the output sequence repeats with this period length, generating a cyclic group.

A number *g* is a primitive root modulo *n* if every number *a* relatively prime to *n* is congruent to a power of *g* modulo *n*. In other words, *g* is a primitive root modulo *n* if, for every integer *a* relatively prime to *n*, there is some integer *k* for which

$$g^k \equiv a \ (mod\ n) \qquad (2.15)$$

Such a value *k* is called the index or discrete logarithm of *a* to the base *g* modulo *n*.

In this way, *g* is a primitive root modulo *n* if and only if *g* is a generator of the multiplicative group of integers modulo *n*. In particular, for a prime number *p*, if *a* is a primitive root of *p*, then $a, a^2, \ldots, a^{p-1}$ are distinct elements (mod *p*), generating a cyclic group with a period length of $p - 1$.

As an example, 3 is a primitive root of 7, but 2 is not.

$3^1 \bmod 7 = 3$	$3^4 \bmod 7 = 4$	$2^1 \bmod 7 = 2$	$2^4 \bmod 7 = 2$
$3^2 \bmod 7 = 2$	$3^5 \bmod 7 = 5$	$2^2 \bmod 7 = 4$	$2^5 \bmod 7 = 4$
$3^3 \bmod 7 = 6$	$3^6 \bmod 7 = 1$	$2^3 \bmod 7 = 1$	$2^6 \bmod 7 = 1$

Considering now the exponential equation

$$y = g^x \ mod\ p \qquad (2.16)$$

it is easy to calculate *y* given *g*, *x*, and *p*, but it is intractable to find *x* (the discrete logarithm) given *y*, *g*, and *p*, thereby establishing a one-way function.

2.4.5 HASH FUNCTIONS

A hash function H computes a digest of a message, accepting a variable-length block of data M as input and producing a fixed-size hash value $h = H(M)$. It can be seen as the fingerprint or a unique representation of a message. Practical hash functions have output lengths between 128–512 bits.

A cryptographic hash function is an algorithm for which it is computationally infeasible to find a data object that maps to a pre-specified hash result (one-way property) or two data objects that map to the same hash result (collision-free property). It is widely used in security applications such as message authentication and digital signatures.

For a hash value $h = H(x)$, x is called the preimage of h. A collision occurs when $x \neq y$ and $H(x) = H(y)$. Generally accepted security requirements for cryptographic hash functions are listed below:

- Variable input size: H can be applied to a block of data of any size;
- Fixed output size: H produces a fixed-length output;
- Efficiency: $H(x)$ is relatively easy to compute for any given x;
- Preimage resistance (one-way property): for any given hash value h, it is computationally infeasible to find y such that $H(y) = h$;
- Second preimage resistance (weak collision resistance): for any given block x, it is computationally infeasible to find $y \neq x$ with $H(y) = H(x)$;
- Collision resistance (strong collision resistance): it is computationally infeasible to find any pair (x, y) with $x \neq y$, such that $H(x) = H(y)$;
- Pseudo-randomness: the output of H meets standard tests for pseudo-randomness.

In order to handle an arbitrary-length message, the hash function segments input data into a series of equal size blocks, which are then processed sequentially by a compression function. This iterated design is known as Merkle–Damgård construction.

Many dedicated (custom designed) hash functions have been proposed in the last decades, and the most popular ones have been part of the so-called MD4 family. This message digest algorithm was developed by Ronald Rivest using 32-bit variables and bitwise Boolean operations to allow very efficient software implementations. A strengthened version,

named MD5, are still used in Internet security protocols for computing file checksums or for storing of password hashes.

After early signs of potential weaknesses, a new algorithm was published in 1993, called secure hash algorithm (SHA). In 1995 it was modified to SHA-1. Both are 160-bit hash functions based on the MD5 algorithm. Further improvements led to SHA-2 in 2004 and SHA-3 in 2015. SHA-2 is still an industry standard, with variants generating hash values in the range of 256–512 bits.

2.5 CRYPTOGRAPHIC SYSTEMS

A vulnerable communication system transmits data from node A (Alice) to B (Bob) through an insecure channel. Secret and valuable information can be intercepted by an eavesdropper (Eve). As depicted in Figure 2.4, this security issue can be addressed by cryptographic systems using an encryption mechanism on the transmitter side to convert the original message *x*, known as plaintext, into an encrypted version *y*, called ciphertext, which becomes useless and unintelligible to an intruder. A decryption function on the receiver side reverts this operation and recovers the original message.

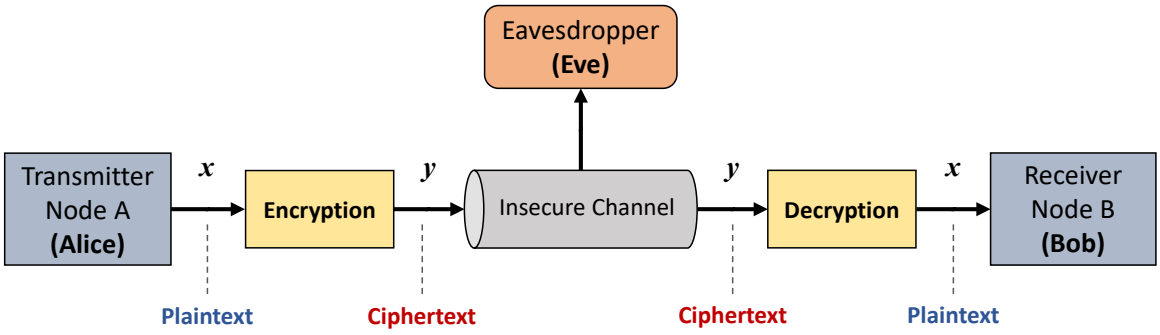


Figure 2.4 – Ciphered communication system.

Encryption mechanisms address the important issue of information security in a communication link, obscuring data that could be inspected by an intruder tapping one or both of the lines of a bidirectional communication channel without the knowledge of the terminal units. Authentication strategies are used to ensure data integrity and detect a possible manipulation as the intruder could also insert malicious regenerator modules in the communication link, being able not only to intercept traffic, but also to modify the information contents. In an OTN system, for instance, as long as the intruder correctly

updates the BIP-8 (bit-interleaved parity — purely an error detection scheme) values in the OTN frames, data terminal equipment will not notice any modifications in the payload.

Cryptographic systems apply special algorithms with a set of rules for data manipulation (based on either substitutions and permutations or complex mathematical operations) to provide the expected data security functions. They all depend on the secrecy of a key, used to parameterize the publicly known encryption and decryption functions.

Cryptanalysis is the science of breaking cryptosystems and, together with cryptography, makes up the field of cryptology. Many types of attacks are enforced against encryption algorithms as an attempt to breaching security.

In a brute-force attack, every possible key is tried until a meaningful ciphertext to plaintext correspondence is achieved. Analytical attacks, on the other hand, rely on the algorithm structural properties, the knowledge of the plaintext characteristics, or even the availability of sample plaintext–ciphertext pairs in order to deduce a specific plaintext or the key being used by the cryptosystem. Some of these cryptanalytic attack models [30] are listed in Table 2.3.

Attack Model	Known to Cryptanalyst
Ciphertext Only	▪ Ciphertext.
Known Plaintext	▪ Ciphertext; ▪ One or more plaintext–ciphertext pairs formed with the secret key.
Chosen Plaintext	▪ Ciphertext; ▪ Plaintext message chosen by cryptanalyst, together with its corresponding ciphertext generated with the secret key.
Chosen Ciphertext	▪ Ciphertext; ▪ Ciphertext chosen by cryptanalyst, together with its corresponding decrypted plaintext generated with the secret key.
Chosen Text	▪ Ciphertext; ▪ Plaintext message chosen by cryptanalyst, together with its corresponding ciphertext generated with the secret key; ▪ Ciphertext chosen by cryptanalyst, together with its corresponding decrypted plaintext generated with the secret key.

Table 2.3 – Cryptanalytic attack models.

Two other important generic approaches are the birthday and the meet-in-the-middle attacks, which are called collision attacks [31] and are based on the birthday paradox [53]

which shows a surprisingly large probability of appearance of duplicate values (also called collisions), such as the reuse of keys in a cryptosystem.

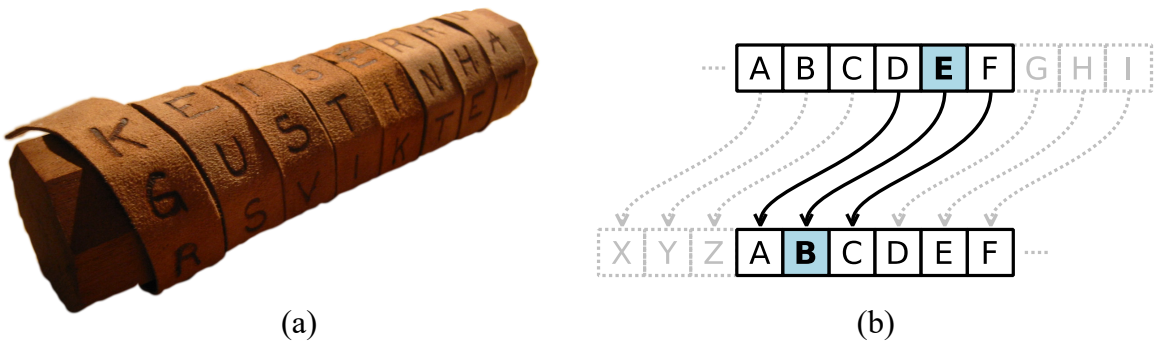
2.5.1 HISTORICAL EVOLUTION

In ancient Egypt, around 4000 years ago, writing by symbols (hieroglyphics) started the history of cryptography. It was not a secret writing system, as in the modern concept, but through some substitution of usual symbols for unusual ones (probably aiming to convey dignity and authority), it incorporated one of the essential elements of cryptography: a deliberate transformation of writing [56].

As early as 500 BC, in ancient Greece, the Spartan military used the "Scytale" — a method of scrambling messages through a transposition process obtained by helically winding a ribbon around a stick [57], as illustrated in Figure 2.5 (a) [58]. The written message then became unintelligible when the ribbon was unrolled, being deciphered only by the receiver who had a stick of the same diameter.

Later, one of the simplest and most well-known encryption techniques based on character substitution appeared in ancient Rome. An example is Caesar's famous monoalphabetic cipher, used around 50 BC by Emperor Gaius Julius Caesar. For the encryption, each letter in the clear message is replaced by another one from the same alphabet after shifting some positions, as illustrated in Figure 2.5 (b) [59].

Using a single alphabet, the Caesar cipher key space is quite limited (only 25 characters), making the code easily decipherable.



(a) (b)

Figure 2.5 – Scytale (a) used by the Spartan and the representation of the monoalphabetic substitution used in the Caesar cipher (b).

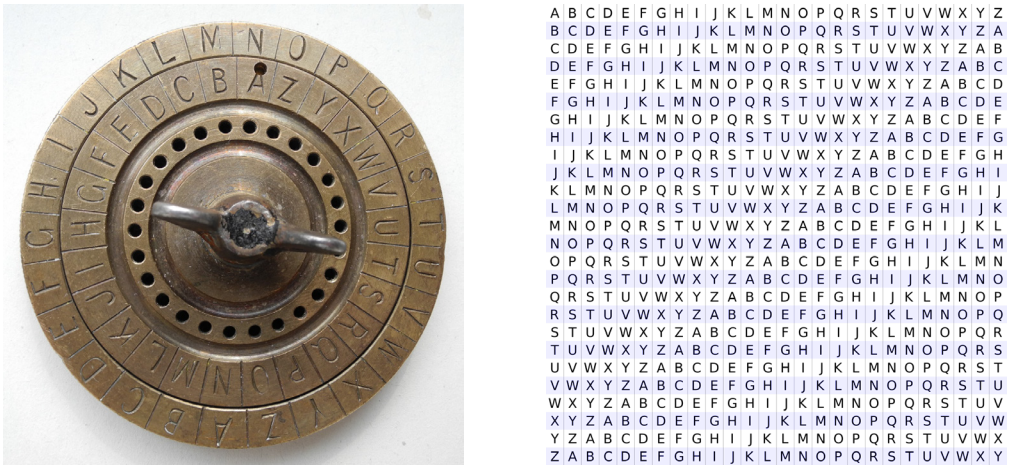
The first well-documented polyalphabetic cipher is believed to have been created by Leon Battista Alberti around 1467 [56]. He also developed a device for encoding and decoding, illustrated in Figure 2.6 (a) [60]. This cipher disk consisted of two concentric disks with printed alphabets mounted one on top of the other. The smaller disk rotated on top of the larger one, allowing the alphabets to shift.

When a fixed offset (or a fixed key) was used, the cipher became monoalphabetic, equivalent to the Caesar cipher. When the offset was changed throughout the encoding process, it became a polyalphabetic cipher.

In 1518, the book "Polygraphia" was published, considered the first printed work on cryptology. It was written by a German Benedictine abbot named Johannes Trithemius, who created a square table of alphabets in which rows are formed by the previous ones shifted one position to the left. This was called "Tabula Recta", illustrated in Figure 2.6 (b) [61], and served as a polyalphabetic cipher (consisting essentially of 26 Caesar ciphers).

The first letter of the clear message was encrypted by substitution using the first alphabet, the second letter, using the second alphabet, and so on, returning to the first after 26 letters.

The Trithemius cipher is also the first instance of a progressive key system, in which all cipher alphabets are exhaustively used before any repetition [56]. Modern cryptographic systems also make use of this key progression.



(a) (b)

Figure 2.6 – Cipher disk (a) and the "Tabula Recta" (b) used by Trithemius.

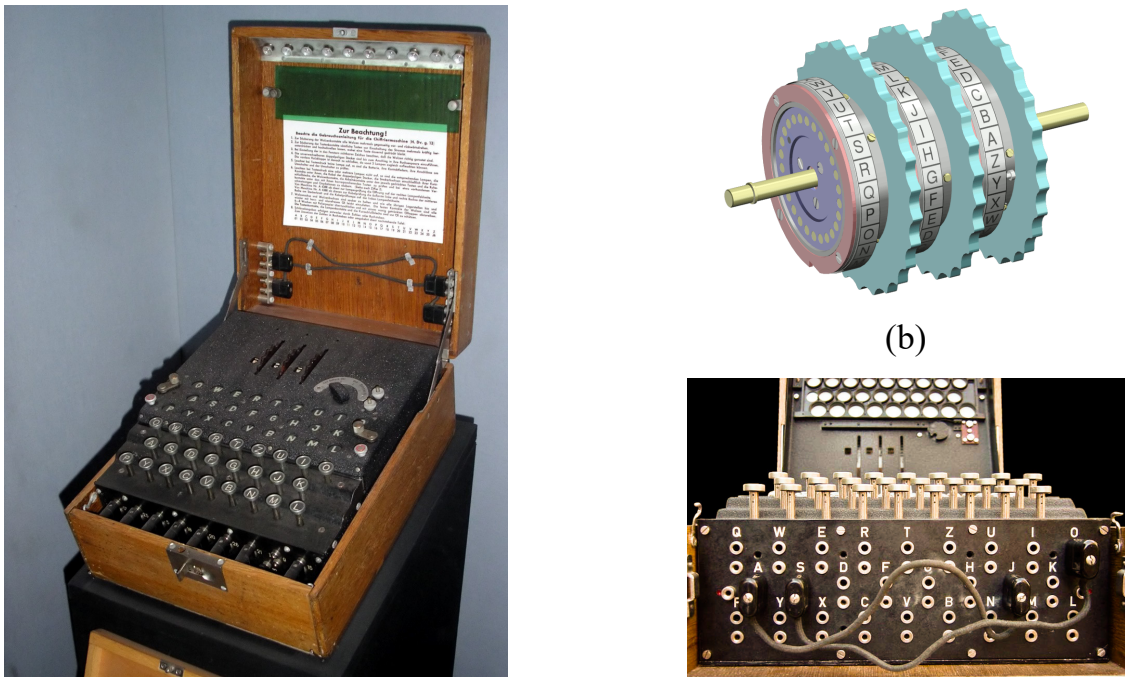
In 1586, Blaise de Vigenère published a type of polyalphabetic cipher called an autokey cipher, which used secret keys iteratively derived from both clear and encrypted messages [56]. It constitutes one of the first major cryptographic systems, gaining a reputation for being exceptionally strong and remaining indecipherable for about 300 years [62].

After the World War I, Arthur Scherbius invented in Germany a polyalphabetic substitution cipher based on electromechanical rotors — a machine called Enigma [57, 63], shown in Figure 2.7 (a) [64].

Each letter of the clear message was typed on a keyboard, and the corresponding ciphered letter lit up in a specific panel. Three rotors (later increased to four) of 26 steps with electrical contacts, as illustrated in Figure 2.7 (b) [65], generated the coding. One of the cylinders rotated one step for each typed letter. The second rotated one step for each complete revolution of the first, and the third, one step for each complete revolution of the second. The result is a set of $26 \times 26 \times 26 = 17{,}576$ different substitution alphabets used before the system returns to its initial state [30].

When the armed forces of Nazi Germany adopted the Enigma, a front plug board was added to the electromechanical apparatus, as shown in Figure 2.7 (c) [66], allowing permutations of letters between the keyboard and the first cipher. The number of rotors also increased to five (of which three were chosen and installed in the machine). With the rotor order arbitrarily adjustable and the number of cables for the plug board increased from six to ten, the encryption key space was approximately 1.6×10^{20} [62].

The UK created a secret department, based in Bletchley Park, to break codes and decipher military messages. Among the cryptanalysts, formed by brilliant mathematicians, engineers, linguists, and others, was Alan Turing, considered the father of computer science. Aided by Polish mathematicians, such as Marian Rejewski et al., who had already broken the Enigma code in its commercial version, Turing developed a machine called “Bombe” that could neutralize the effect of the plug board, thus reducing the effort to code breaking by a factor of about 10^{14} [57, 62].



(a) (b) (c)

Figure 2.7 – Enigma machine (a), 3D illustration of its rotors (b), and the plug board (c) with connections creating the substitutions S ↔ O and A ↔ J.

Claude Shannon, the father of information theory, also laid a solid theoretical foundation for both cryptography and cryptanalysis [62]. The concept that a secure cryptographic system depends on the secrecy of the key and not only on the complexity or secrecy of the encryption algorithm (Kerckhoffs principle) had already been enunciated by Auguste Kerckhoffs in 1883 [67]. Shannon rephrased it to "the enemy knows the system being used", establishing principles [68] still used today:

- Substitution (*S box*) – generating confusion and hiding the connection between the encrypted message and the key;
- Transposition (*T box*) – generating diffusion and obscuring the statistical relationship between the clear and encrypted messages;
- Multistage – a super encipherment with serial stages of substitution and transposition, also known as a product cipher.

These principles were used in the development of the first encryption standard, published in 1977. The data encryption standard (DES) evolved from the result of a competition in the United States, in which the IBM system was the winner.

Using a 64-bit block cipher and 16 stages of substitution and transposition, it was placed in the public domain and then broken by various groups using the exhaustive key search method, also called a brute-force attack [62].

After a further selection process, it was then replaced in 2001 by a new advanced encryption standard (AES) that used 128, 192, and 256-bit keys, thus generating a virtually unbreakable number of combinations.

Shannon also proved that a single-use cipher is information-theoretically secure, as long as it is used correctly, even for an attacker with infinite computing power. However, its key must be random, at least as long as the clear message to be encrypted, and used only once.

This one-time pad (OTP) cryptosystem was first described by Frank Miller in 1882, 35 years before the patent issued to Gilbert S. Vernam, who developed an electromechanical system using similar principles to encrypt teletype communications. In the early implementations, the key material was distributed as a thick pad with hundreds of single-use paper sheets (hence the name), each bearing a unique key in the form of lines of randomly sequenced letters or numbers.

Historical uses of one-time pad systems starting back in the early 1900s [56, 63] include communications between diplomatic and military organizations, espionage agencies such as the KGB, and even the hotline between Moscow and Washington D.C. established in 1963 — a teleprinter arrangement, popularly known as the red telephone, protected by a one-time tape system.

Practical OTP systems experience two fundamental difficulties — the generation of truly random single-use keys and their distribution across all the communication parties.

In 1976, Whitfield Diffie and Martin Hellman published a classic paper [69] with a secure key exchange mechanism and an asymmetric-key encryption concept that opened the door to public-key systems (PKS). In this approach, the receiver has a secret or private key and a public key. The private key is used by the decryption algorithm, while the public key is used by the sender to encrypt the clear message to be transmitted.

Two years later, in 1977, Massachusetts Institute of Technology (MIT) scholars Ronald Rivest, Adi Shamir, and Len Adleman invented a public-key system named after their

initials — RSA, which is based on the difficulty of factoring products of large prime numbers [70].

The inversion of the PKS channel, using the private (secret) key on the transmitter side and the public key on the receiver side, makes it possible to implement the concept of digital signature, in which the transmitter can authenticate itself to the receiver [53].

With the incipient practical implementation of quantum computing technology [71], it is possible that, in a few decades, codes that would take billions of years to be broken in brute-force attacks by advanced conventional computers will be more quickly deciphered.

The same concepts of quantum mechanics that guided the development of quantum computers also gave rise to the so-called "quantum cryptography" [72], which so far actually presents a secure solution for exchanging keys between parties [73]. Many of these quantum communication methods, and particularly the so-called quantum key distribution (QDK) protocols, use a subset of photon polarization states to represent a binary '0' and another for '1' [14]. The first demonstration of an experimental prototype carried out in 1992 [74] established the technological feasibility of this concept.

2.5.2 Symmetric and Asymmetric Cryptography

Classical modern cryptography can be split into two major branches: secret or symmetric-key systems and public or asymmetric-key distribution methods.

In a symmetric cryptography approach, also known as symmetric-key or single-key scheme, the same cipher key is used by the two communication nodes (Alice and Bob) to encrypt and decrypt messages, as shown in Figure 2.8.

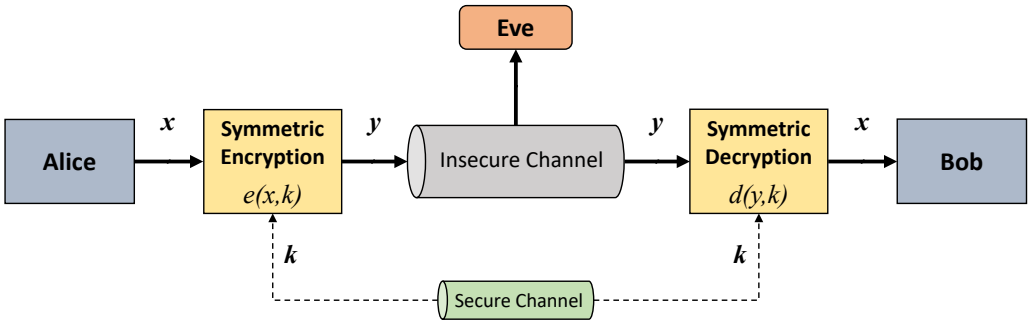


Figure 2.8 – Symmetric-key cryptosystem.

The implementation of this cryptosystem presupposes the existence of a secure channel through which both nodes exchange their secret cipher key.

A symmetric encryption function *e(x,k)* accepts two inputs: the clear message to transmit (*x*), or plaintext, and a cipher key (*k*). The output of *e(x,k)* is an encrypted message (*y*), or ciphertext. The original message can only be found by applying the decryption function *d(y,k)* on the encrypted message with the correct cipher key (*k*).

The selected encryption function, based on substitutions and permutations, must make it difficult for intruders (such as Eve) to guess the cipher key from the encrypted message and, the larger the key size (in bits), the longer it will take to be discovered via brute-force or some other mathematical attack. With 256 bits, for instance, there are $2^{256} \approx 1.1 \cdot 10^{77}$ different keys to be tried in a brute-force attack.

Assuming, for instance, that each possible combination could be checked in just one single floating-point operation (flop), a supercomputer capable of performing at 1 exaflop/s, for example, would need $36.7 \cdot 10^{48}$ years to check just 1% of this huge key space. Interestingly enough, the current estimate for the age of the universe is approximately $13.77 \cdot 10^{9}$ years.

Fast and secure symmetric-key algorithms are widely used in communication and storage systems. They have, however, two important shortcomings related to the key distribution problem (the need of a secure channel) and the possibility of repudiation (a communication node cheatingly denying the transmission or reception of a message).

These drawbacks were overcome by a revolutionary idea proposed by Whitfield Diffie and Martin Hellman [69] in 1976, with contributions from Ralph Merkle [75] — an astounding breakthrough with the concept of public-key cryptography (PKC).

This radically different asymmetric cryptography approach, depicted in Figure 2.9, uses a pair of related keys (public and private) and moves from the elementary tools of substitution and permutation to the use of one-way and trapdoor mathematical functions, which are easy to compute but intractable to invert, unless using some private extra information in the case of trapdoor functions. Public keys can be published and do not need to be obscured. Only the private keys must be secretly maintained.

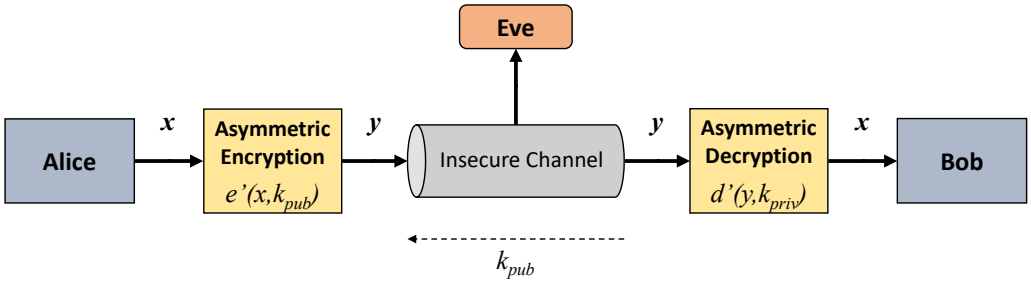


Figure 2.9 – Asymmetric-key cryptosystem.

The asymmetric encryption function $e'(x,k_{pub})$ ciphers Alice's plaintext (x) using the previously shared Bob's public key (k_{pub}), generating the ciphertext (y). The original message is then recovered by applying the decryption function $d'(y,k_{priv})$ using the related private key (k_{priv}), secretly maintained by Bob.

The inversion of the public-key system channel — using the private key for message encryption and the public one for decryption — provides a means for nonrepudiation using digital signature algorithms [53].

In these scenario, Bob, for instance, digitally signs a message by applying an algorithm taking the message itself and his private key as inputs. The resulting digital signature is then transmitted appended to its respective message. Alice later applies a signature verification algorithm using the received message and the appended digital signature as inputs. The result is a true/false or valid/invalid signature indication, thus legitimating the origin of the message.

Three major families of public-key algorithms of practical relevance can be classified based on their underlying computational problem [53]:

- Integer Factorization Schemes – such as the RSA, based on the difficulty to factor large integers;
- Discrete Logarithm Schemes – such as the Diffie–Hellman (DH) key exchange, based on what is known as the discrete logarithm problem in finite fields;
- Elliptic Curve (EC) Schemes – such as the Elliptic Curve Diffie–Hellman (ECDH) key exchange, being a generalization of the discrete logarithm algorithm.

These public-key algorithms require long operand arithmetic, based on number theoretic functions. The longer these operands and keys, the more secure the algorithms become.

Their security level, which expresses the cryptography strength, can be defined by a number of bits, resulting in the number of steps required in an attack. A security level of *n* bits indicates that the best-known attack for a particular algorithm requires 2^n steps.

A symmetric cryptosystem with an *n*-bit key has a security level of *n* bits, but this relationship is not as straightforward in the asymmetric case. Table 2.4 shows the recommended key bit lengths for security levels of 80, 128, 192, and 256 bits for both asymmetric and symmetric cryptosystems.

Algorithm Family	**Cryptosystem**	**Security Level (bit)**			
		80	**128**	**192**	**256**
Integer Factorization	RSA	1024	3072	7680	15360
Discrete Logarithm	DH	1024	3072	7680	15360
Elliptic Curves	ECDH	160	256	384	512
Symmetric-key	AES	80	128	192	256

Table 2.4 – Recommended key bit lengths for security levels of 80, 128, 192, and 256 bits for both asymmetric and symmetric cryptosystems.

These results show that for a given cryptographic strength, asymmetric cryptosystems require longer keys when compared to the symmetric case. EC schemes offer the closest asymmetric equivalent to symmetric encryption in terms of performance.

The computational complexity for the calculations needed in the asymmetric cryptosystems does not scale linearly with the key length, but instead grows much more rapidly than that. Consequently, the proposed key sizes are long enough to provide resistance to a brute-force attack but result in a processing speed too slow for a high amount of data encryption [30]. Public-key operations can be 2–3 orders of magnitude slower than symmetric-key encryption [53].

This is the reason why these asymmetric cryptosystems are used only for key management and digital signature applications. Data are commonly encrypted/decrypted by a faster symmetric system (such as the AES) using a key that is exchanged with an asymmetric algorithm.

Figure 2.10 shows two OTN communication nodes A and B using a hybrid cryptosystem, with an asymmetric-key scheme for the key exchange (eliminating the need of a secure

channel) and a symmetric-key system for data encryption (providing secure and fast data processing).

In node A side, the asymmetric encryption function $e'(k,k_{pub})$ ciphers the symmetric key (k) using the previously shared public key (k_{pub}). In node B side, the asymmetric decryption function $d'(z,k_{priv})$ receives the ciphered key (z) and decrypts the symmetric key (k) using the related private key (k_{priv}). After being securely exchanged, the symmetric key (k) can be used by the symmetric encryption $e(x,k)$ and decryption $d(y,k)$ functions to provide data confidentiality over the communication channel.

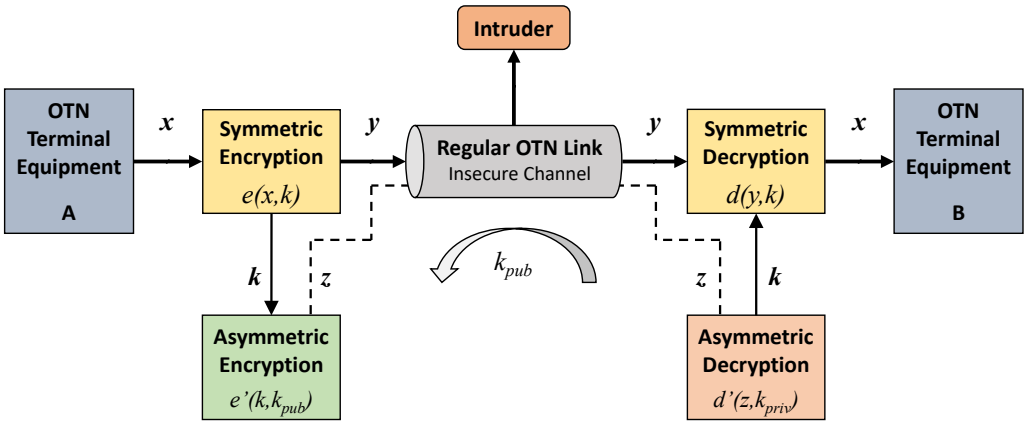


Figure 2.10 – Hybrid (symmetric and asymmetric) OTN cryptosystem.

2.5.3 Quantum Cryptography

Although symmetric encryption algorithms can provide a fast and secure means of ensuring data confidentiality, as discussed in the previous section, the fundamental problem of key distribution persists. Public-key systems (using asymmetric encryption) were designed to handle this limitation, also addressing the non-repudiation issue.

Nonetheless, as soon as sufficient computational power becomes available or new mathematical algorithms are designed (with fast and clever procedures for factoring large numbers, for instance), encryption key security will eventually be at risk, compromising the whole cryptography solution.

Quantum information theory is a recently emerged area of physics and technology, with astonishing implications in computer science and communication systems. In contrast with the use of discrete classical deterministic bits, a quantum information processing system encodes the elementary units of information using non-classical properties of a quantum

system in a superposition state, yielding revolutionary properties and possibilities in the new domain of Quantum Computing.

The basic unit of quantum information is a two-state quantum-mechanical system called Qubit (quantum bit), which can be engineered as the spin of the electron, trapped atoms and ions, photons, and superconducting circuits. Following a fundamental principle of quantum mechanics, qubits represent not only the '0' and '1' binary states but also their superposition.

One application of these new information processing techniques is called Quantum Cryptography [76], which so far in fact does not directly encrypt data, but rather provides quantum key distribution (QKD) solutions based on concepts from quantum mechanics such as the Heisenberg's uncertainty principle, no-cloning theorem, and quantum entanglement.

The first QKD protocol — the still widely used BB84 — was invented in 1984 by Charles Bennett of IBM Research and Gilles Brassard of the University of Montreal [77], being then demonstrated with an experimental prototype in 1992 [74] using a quantum channel to transmit qubits in the form of polarized photons.

In accordance with the laws of quantum mechanics, polarized photons behave deterministically only when they pass through a parallel or perpendicular filter, being transmitted or absorbed depending on the correspondence of their polarization axes. When diagonally polarized photons pass through a vertically oriented polarizing filter, however, they emerge randomly repolarized in either the vertical or the horizontal direction. Consequently, the receiver can distinguish between either rectilinear (0 and 90°) or diagonal (45 and 135°) polarizations, but not between both.

In this way, because of the uncertainty principle, an eavesdropper cannot measure the photon polarizations without modifying them in such wise that it allows the action to be detected by the communication parties. This is what ensures unconditional absolute security.

The BB84 QKD protocol can be divided in the following steps:

1. Alice choses a random bit sequence as well as a random set of rectilinear (0 and 90°) or diagonal (45 and 135°) polarization bases;
2. Alice encodes each bit polarizing photons according to the corresponding chosen bases — for instance, a horizontal or 45° polarization for a binary '0' and a vertical or 135° for a binary '1';
3. Bob receives the photons and, for each one, randomly decides to measure its rectilinear or diagonal polarization, converting the results back into a bit sequence;
4. Bob and Alice then communicate through a public but authenticated channel. Bob tells her which type of measurement (rectilinear or diagonal) was used for each photon (but not the measurement result), and Alice tells him whether he has made the right kind of measurement;
5. They both discard all cases in which Bob has made the wrong measurement;
6. The remaining polarizations can then be decoded and sifted as a new shared key sequence to be used for secure encryption purposes.

Figure 2.11 illustrates these protocol steps, with the highlighted columns indicating the correspondence between the sending and measurement bases.

Alice's random bits	0	1	1	0	1	1	0	0	1	0	1	1	0	0	1
Alice's random sending bases	+	+	×	+	×	×	×	+	+	×	+	×	+	+	×
Alice's photon polarizations	↔	↕	⤡	↔	⤡	⤡	⤢	↔	↕	⤢	↕	⤡	↔	↔	⤡
Bob's random measuring bases	×	+	+	+	+	+	×	+	×	×	×	×	×	+	×
Bob's photon polarizations	⤢	↕	↕	↔	↔	↕	⤢	↔	⤢	⤢	⤡	⤡	⤡	↔	⤡
Bob's received bits	0	1	1	0	0	1	0	0	0	0	1	1	1	0	1
Public discussion of bases															
Sifted key		**1**		**0**			**0**	**0**		**0**		**1**		**0**	**1**

Figure 2.11 – BB84 protocol illustration.

Eavesdropping can be detected by publicly comparing (and then discarding) a randomly selected subset of the sifted key or, with much more efficiency, by a parity calculation within a random subset containing about half the bits of this key.

Since this detection is done after qubit transmission, there is no point in using this technology to transmit valuable information other than an encryption key, which will be used for encipherment only after the security is verified.

QKD could be used with one-time pad encryption to achieve perfect secrecy. This requires, however, a truly random key (which can also be generated with quantum techniques) at least as long as the message to be encrypted, what may impose bandwidth limitations considering the difficulties of maintaining an error-free quantum communication channel.

Being typically 1.000 to 10.000 times slower than conventional optical communications [78], practical QKD systems are commonly combined with classical symmetric encryption mechanisms (such the AES algorithm) with short-term (frequently replaced) keys to provide quantum-safe security.

Several commercially available QKD systems are supplied by companies such as MagiQ (USA) [79], KETS (UK) [80], ID Quantique (Switzerland) [81], and Toshiba (Japan) [82].

Additionally, there have been also initiatives to standardize a QKD interface, such as the European OpenQKD project [83] and the work of the European Telecommunications Standards Institute (ETSI) [84], which will certainly facilitate the introduction of QKD for OTN vendors. Some are indeed already taking part in this new technology, joining efforts with telecom operators to make demonstrations combining symmetric encryption solutions with QKD systems in multiple testbed networks [85, 86].

2.6 ADVANCED ENCRYPTION STANDARD – AES

A well-known, widely adopted and worldwide standardized symmetric encryption algorithm is the AES (Rijndael) [87, 88], developed by the Belgian cryptographers Joan Daeman and Vincent Rijimen [89].

Following a five-year standardization process, Rijndael was selected between fifteen competing designs and was announced by the National Institute of Standards and Technology (NIST) as the Federal Information Processing Standards (FIPS) publication 197 on November 26, 2001.

This 20+ year algorithm is still largely used in current digital communication and computer/storage systems, being implemented for instance in commercially available OTN processor ASSPs and self-encrypting drives (SEDs) due to its robust, fast, and secure encryption capabilities. Some other applications include file transfer and Wi-Fi security

protocols, programming language libraries, mobile apps, file compression tools, and disk partition encryption.

AES is an iterative block cipher that encrypts 128 bits of data using a 128/192/256-bit cipher key. Data are processed by repeatedly combining substitution (bringing in some confusion) and permutation computations (adding diffusion) within several rounds (10, 12, or 14, depending on the key size) to produce the ciphertext.

Each 128-bit input block is arranged into a 4 × 4 matrix of 16 bytes, called a state matrix, and data are processed within each round using operations in the Galois field $GF(2^8)$, following a set of standardized steps: substitute bytes, shift rows, mix columns, and add round key. A key expansion process generates each 128-bit round key from the original cipher key (of 128, 192, or 256 bits). Decryption is achieved by performing these operations in the reverse order.

The sequence of operations listed in Figure 2.12 shows the encryption process using a pseudo-code notation [89]. Encryption consists of an initial key addition, denoted by *AddRoundKey*, followed by $Nr - 1$ rounds with the application of the transformation denoted by *Round* and, finally, one application of the final transformation denoted by *FinalRound*. The number of rounds (*Nr*) is 10, 12, or 14, depending on the length of the cipher key (128, 192, or 256 bits).

The initial key addition and each subsequent transformation take as input the *State* matrix and a round key, which is denoted by *ExpandedKey[i]* in the round *i*. The operation of deriving the round keys (*ExpandedKey*) from the main cryptographic key (*CipherKey*) is denoted by *KeyExpansion*.

```
procedure AES_Rijndael(State,Cipherkey)
  KeyExpansion(CipherKey,ExpandedKey)
  AddRoundKey(State,ExpandedKey[0])
  for i = 1 to Nr - 1 do
     Round(State,ExpandedKey[i])
  end for
  FinalRound(State,ExpandedKey[Nr])
end procedure
```

Figure 2.12 – AES encryption algorithm pseudo-code.

The *Round* transformation consists of four different operations or steps: Substitute Bytes (generating confusion), Shift Rows and Mix Columns (generating diffusion), and Add Round Key [30]. The *FinalRound* is identical to the previous ones, except for the removal of the Mix Columns step.

Figure 2.13 shows a simplified flowchart of the AES encryption algorithm.

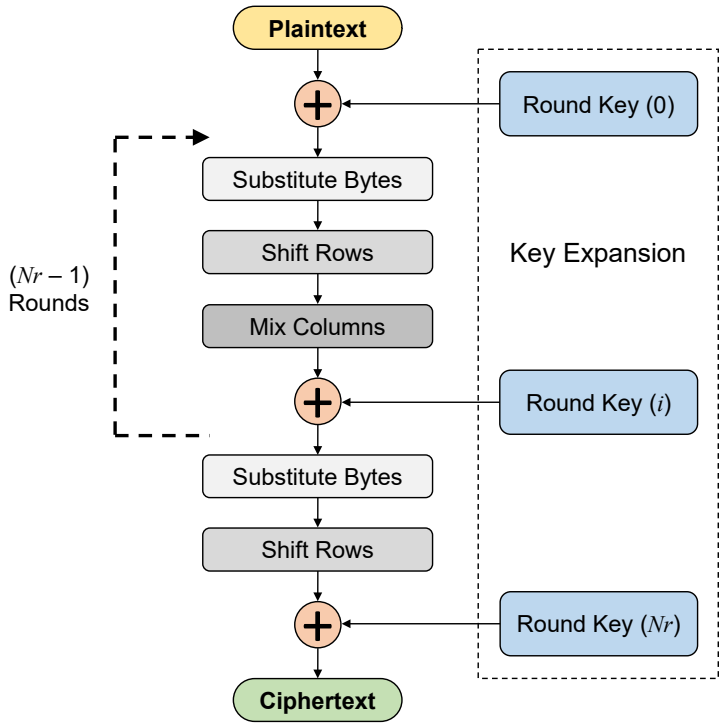


Figure 2.13 – Simplified flowchart of the AES encryption algorithm.

2.6.1 SUBSTITUTE BYTES

The Substitute Bytes step, illustrated in Figure 2.14 [90], constitutes a non-linear transformation in which each byte $a_{i,j}$ of the input state matrix is replaced by another, $b_{i,j}$, according to a substitution box (S-Box) constructed from a multiplicative inverse in the finite field $GF(2^8)$ and a bitwise affine transformation.

The S-Box is a 16 × 16 matrix with a permutation of all 256 possible 8-bit values. The replacement process is typically implemented from a lookup table (LUT) in which the four most significant bits of $a_{i,j}$ address the row, while the four least significant bits address the columns. In this way, the hexadecimal value 95, for instance, addresses the position of row 9 and column 5 of this table, containing the value 2A, thus generating the substitution 95 → 2A.

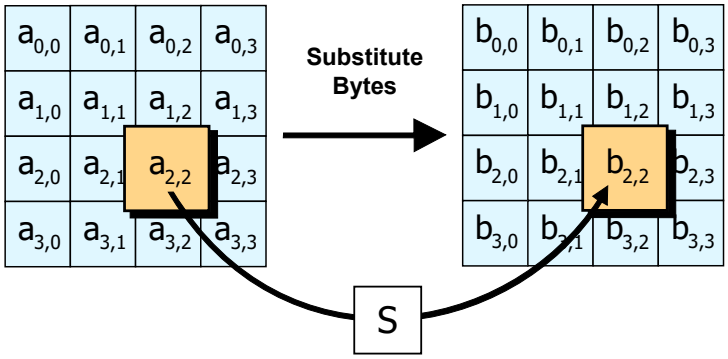


Figure 2.14 – Illustration of the AES algorithm Substitute Bytes step.

This transformation generates confusion, causing each bit of the ciphered message to depend on several parts of the key, thus making it difficult to be discovered from known plaintext attacks in which pairs of clear and encrypted messages are analyzed.

2.6.2 SHIFT ROWS

In the Shift Rows step, illustrated in Figure 2.15 [91], the first row of the state matrix is preserved while the second, third, and fourth rows are cyclically shifted 1, 2, and 3 bytes to the left, respectively.

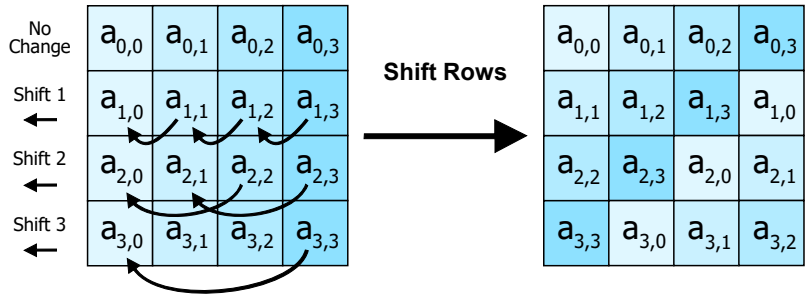


Figure 2.15 – Illustration of the AES algorithm Shift Rows step.

2.6.3 MIX COLUMNS

The Mix Columns step, illustrated in Figure 2.16 [92], operates on each column individually by mapping each of its bytes into new values that are a function of all four bytes in that column. This can be represented in terms of polynomial arithmetic by considering each column of the state matrix as a four-term polynomial with coefficients in $GF(2^8)$. Each column is then multiplied (modulo $\{01\}x^4 + \{01\}$) by a fixed polynomial $C(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$, where the $\{n\}$ notation indicates n in the hexadecimal number base.

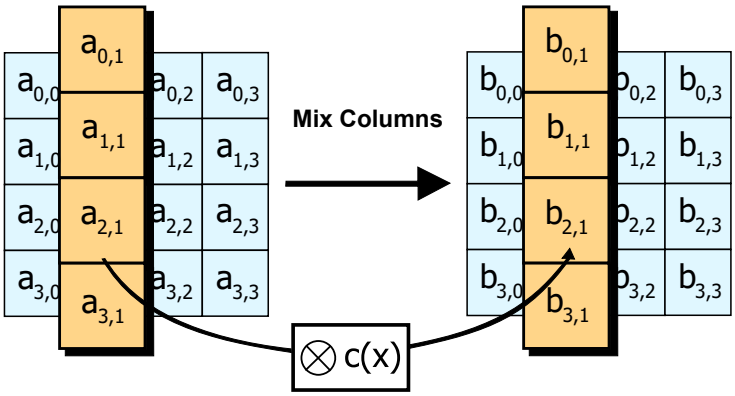


Figure 2.16 – Illustration of the AES algorithm Mix Columns step.

This transformation, together with the Shift Rows step, generates diffusion, so that a casual change in one bit of the clear message produces a change in several bits of the ciphered message, thus obscuring their statistical relationship.

2.6.4 ADD ROUND KEY

In this step, illustrated in Figure 2.17 [93], the round key is added to the state matrix by a bitwise eXclusive OR (XOR) logic operation.

As depicted in Figure 2.13, this addition happens at both the input and output of the AES algorithm and is sometimes referred to as key whitening. It helps to increase the resistance against exhaustive key search attacks.

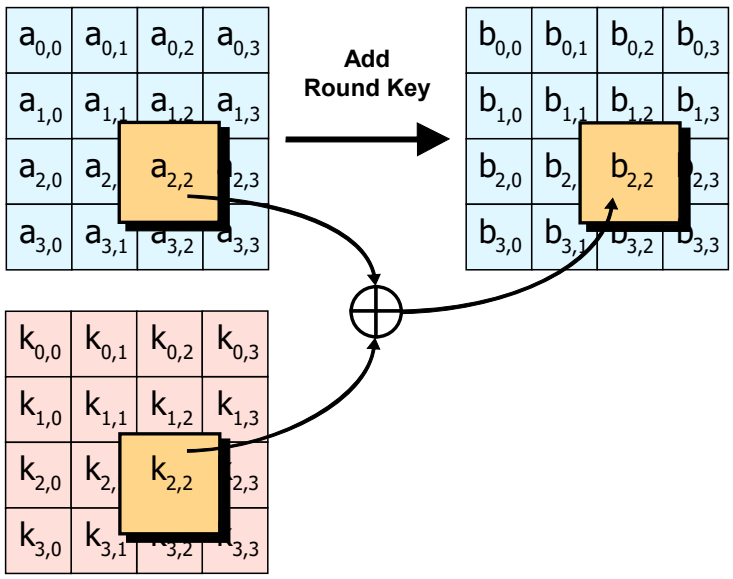


Figure 2.17 – Illustration of the AES algorithm Add Round Key step.

2.6.5 ROUND KEY DERIVATION

The round keys (128 bits) are derived from the main cryptographic key (128, 192, or 256 bits) from an expansion process, generating a linear vector of 128 × (*Nr* + 1) bits through operations performed on 4-byte words involving substitutions (S-Box), cyclic permutations, and recursive XOR operations. The selection of round keys is done by consecutively reading, at each new round, the necessary number of bits from the expanded vector.

Alternatively, key derivation can be done in real time, without expanding the full array — a solution particularly suited to memory-constrained systems.

2.7 MODES OF OPERATION

In order to encrypt a larger than 128-bit message, the AES block cipher needs to be repeatedly applied in a way defined by different modes of operation [94]. A straightforward approach would be a direct use of the block cipher, breaking the original message into 128-bit chunks, for instance, encrypting each one independently, and then concatenating the resulting ciphertext blocks together.

Although highly parallelizable in hardware implementations, this Electronic Code Book (ECB) mode of operation is not semantically secure [95, 96], as similar 128-bit plaintext blocks will look very similar at the output of the encryption function, helping an attacker to build up a codebook and extract the message contents.

An alternative solution called Cipher Block Chaining (CBC) mode involves connecting the cipher blocks in such a way that the resulting ciphertext becomes dependent on all the previously processed plaintext blocks. This also requires an additional primitive called Initialization Vector (IV) sequence, which is essentially a cryptographic “nonce” (number only used once), to randomize the encryption of the first block. This chaining connection, however, prevents hardware parallelization.

The counter (CTR) mode of operation uses a set of counters as input blocks for the cipher. The resulting encrypted blocks are then combined with those of the clear message through a logic XOR operation, generating confidentiality. In this architecture, which is highly parallelizable in hardware, randomization for subsequent message blocks is achieved as

this counter is incremented for each new cipher instance. Obviously, care must be taken with its initialization mechanism so that values do not repeat between different instances of the encryption function using the same key. An initialization vector (IV) can then be concatenated with the counter, ensuring this uniqueness requirement [53, 94]. Figure 2.18 illustrates the CTR mode of operation using an AES block cipher and an IV sequence.

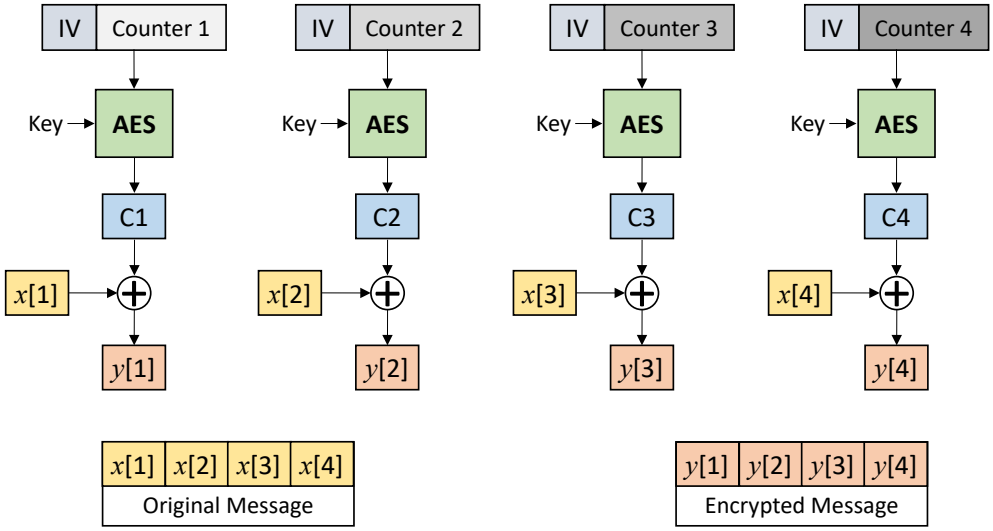
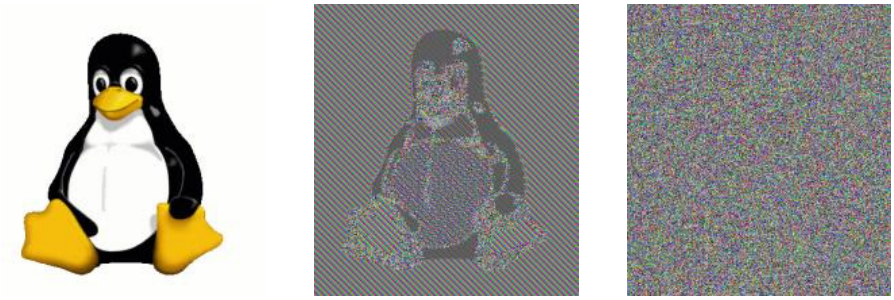


Figure 2.18 – Illustration of the CTR mode of encryption using an AES block cipher and an IV sequence.

If the length of the chosen IV sequence is 96 bits, for example, 32 bits are left for the counter (totaling the 128 input bits of the AES block cipher). For each new message block encrypted with the same key, the counter must be incremented while the IV remains fixed. In this example, 2^{32} message blocks can be encrypted with the same key. After that, a new IV sequence must be generated, which must also be different from the previous ones until the key is changed.

In this architecture, the block cipher always perform only the encryption operation, as it encrypts just counter values (already concatenated to IV) that are synchronized in both the transmitter and receiver with respect to each corresponding encrypted block. Consequently, decryption naturally occurs when ciphered data are fed into the algorithm, since if $C \oplus X = Y$, then $C \oplus Y = X$.

Figure 2.19 [97–99] shows the encryption of an image (a) in the ECB (b) and CBC or CTR (c) modes of operation, highlighting the fragility of the ECB mode in keeping clear message patterns present in the ciphered message.



(a) (b) (c)

Figure 2.19 – Examples of the encryption of an original image (a) using ECB (b) and CBC or CTR (c), highlighting the fragility of the ECB mode.

2.7.1 GALOIS/COUNTER MODE – GCM

A standardized algorithm called Galois/Counter Mode [100] provides not only a mode of encryption operation but also an authentication mechanism. It is a combination of the counter mode of encryption with the Galois mode of authentication.

This architecture relies on a 128-bit block cipher and uses multiplications in the 128-bit Galois field $GF(2^{128})$ to realize authenticated encryption and decryption functions, with the computation of a message authentication code (MAC). Galois field multiplication computations are easily parallelized in hardware, allowing for high processing throughputs [101–103].

On the transmitter side, data are encrypted using the CTR mode and a MAC_{TX} is computed and appended to the message. On the receiver side, a MAC_{RX} is also computed and then compared with the received MAC_{TX} to ensure message authentication and integrity. The MAC is also known as authentication TAG and can be thought of as a cryptographic checksum [53].

The GCM algorithm also protects the authenticity of an additional block of data, called additional authenticated data (AAD). However, it is left in clear (not encrypted) and may serve as a container for parameters of a network protocol, for instance.

Figure 2.20 illustrates an AES-GCM algorithm architecture. In the encryption section, five 128-bit AES block ciphers running in parallel (in CTR mode) handle a plaintext data path of 640 bits. In the authentication section, hash multiplications are performed in the Galois

field. The hash sub key (*H*) is obtained by the encryption of a 128-bit zero sequence and the IV sequence is also ciphered before being included in the authentication process.

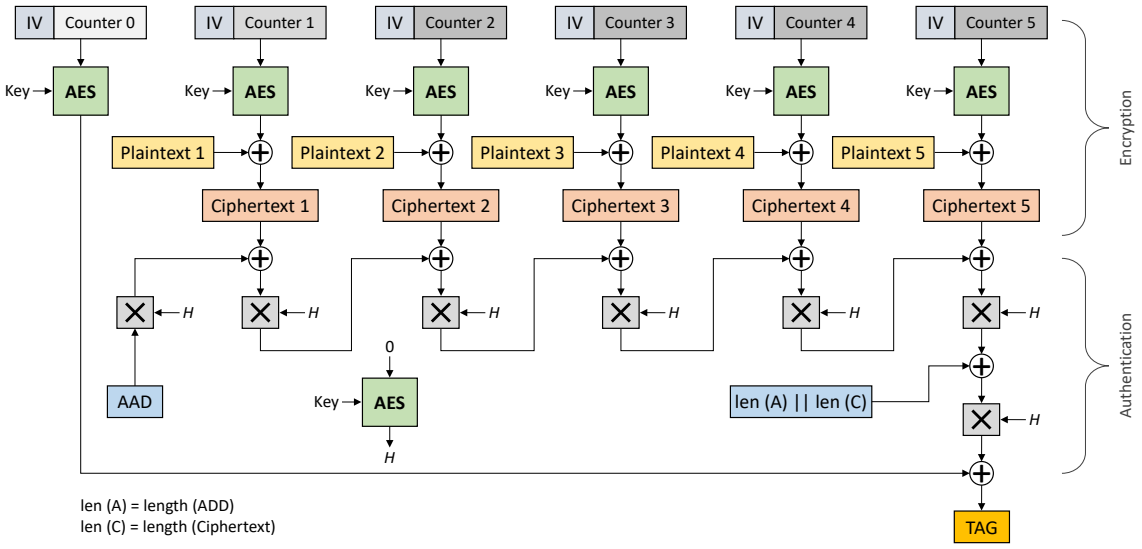


Figure 2.20 – Illustration of the AES-GCM architecture.

2.7.2 GCM Security Aspects

There is a crucial requirement of "uniqueness" [100] for IV sequences in the GCM, in order to ensure the security of this mode of operation [104, 105]:

> *The probability that the authenticated encryption function ever will be invoked with the same IV and the same key on two (or more) distinct sets of input data shall be no greater than* 2^{-32}.

If a single IV is repeated in a set of instances of the authenticated encryption function that uses a given key, the cryptographic solution may be vulnerable to forgery attacks [106].

Although the IV sequence can have any number of bits between 1 and $2^{64} - 1$, the length of 96 bits is recommended to promote interoperability, efficiency, and simplicity of design [100].

The length of the resulting TAG sequence is limited by the size of the finite field within which the hash multiplications are performed (128 bits). Although some applications may benefit from the use of shorter TAGs (such as those involving audio and video, which require lower latency), this may lead to security vulnerabilities and a 128-bit sequence is recommended [100, 107].

2.8 QUANTUM THREAT AND POST QUANTUM CRYPTOGRAPHY

Quantum computing [108] has emerged as a technology capable of solving certain types of problems of huge importance for humankind, which today are practically prohibitive or even impossible to be solved by classical computers.

It represents a new and fundamentally different computing paradigm, carrying tremendous advantages for solving optimization, simulation, and mathematical problems which impact biomolecular systems, pharmaceutical industries, material science such as superconductors, machine learning and artificial intelligence, financial services and technology, and many other areas of human knowledge.

As mentioned before, in Section 2.5.3, instead of digital deterministic states represented by bits, quantum computers rely on encoding information in any coherent two-level system with superposition states requiring probabilistic measurements. These qubits or quantum bits are much more powerful because of the associated quantum mechanical phenomena such as superposition, interference, and entanglement.

In a classical computer, calculations can be done either sequentially in time or in parallel, with multiple copies of the calculating machine. With n bits, there are 2^n states that are processed one at a time by logic gates performing Boolean operations.

Similarly, a quantum computer has qubit gates that can be arranged in circuits to perform any quantum logic. However, it can bring together all those 2^n states and put them all in one superposition state, leading to quantum effects of parallelism and interference, which underlies the processing exponential speedup [109, 110].

Recent quantum supremacy demonstrations include the Sycamore 53-qubit programmable superconducting processor from Google [111], which actually has been beaten by a new classical algorithm [112], and a non-programmable photonic platform with 50 squeezed states [113].

The Canadian company Xanadu Quantum Technologies also presented Borealis, their newest and largest quantum computer ever built. With 216 squeezed-state qubits, it became the first computer capable of quantum computational advantage to be deployed on the cloud. With a runtime advantage that is over 50 million times as extreme as that reported

from earlier photonic machines, it requires only 36 μs to produce a single sample of a particular mathematical problem, which would take on average more than 9.000 years to be done by the best available classical algorithms and supercomputers [114].

Nonetheless, quantum computers need to be engineered to cope with decoherence and errors caused by environmental factors that cause qubits to lose their properties [115].

Coherence time is the lifetime of the quantum-mechanical properties of a qubit, corresponding to the duration of time in which the qubit is in a viable state before it decays due to environmental disruptions. Gate time corresponds to the time required for a single gate operation. A figure of merit is then how many gates can be applied before qubit decoherence. Various techniques to control these phenomena are gathered under the name of quantum error correction (QEC).

Different qubit modalities such as trapped ions, silicon quantum dots, and superconducting qubits show varying performances in terms of the number of operations before errors, as well as the gate speed, which ranges from kilohertz up to a few gigahertz [116].

Special algorithms have been developed to explore the advantages of quantum computers in solving specific problems such as simulation, factoring, linear systems, and search [110]. Two of most importance for encryption-based security systems are Shor's factoring and Grover's search algorithms.

In 1994, Peter Shor developed an efficient algorithm for finding discrete logarithms and factoring integers that runs in a polynomial time on a quantum computer with a sufficient number of qubits [117]. This is the most famous and possibly most important quantum algorithm discovered to date.

Given a number N, the goal is to factor it and find its prime factors p and q. The difficulty of doing that with classical computers underpins the security basis of the RSA public-key algorithm. The runtime of the best-known classical algorithm for factoring an n-bit number is roughly $e^{n^{1/3}}$ steps, which is not polynomial time and thus inefficient, while in a quantum computer running Shor's algorithm, it is $n^2 \log n \log \log n$ steps, reducing a sub-exponential problem to a polynomial problem [109].

Another mundane problem in information processing is searching for an entry that satisfies a given condition in an unsorted database of $N = 2^n$ items. The most efficient way of achieving that with a classical algorithm is to examine all the N items in the database until finding the correct answer, which on average will require $N/2$ trials.

Lov K. Grover devised in 1996 [118] a quantum search algorithm that finds the expected entry with a quadratic speedup of $\sqrt{N}$ steps [109]. This means that a symmetric cryptosystem with keys of 128 bits, for instance, could be brute-force attacked with 2^{64} steps by a quantum computer, which is still a huge effort.

The current question is how this quantum leap in computing power and new algorithms will ever challenge and change the way information is securely communicated.

Whether large-scale cryptographically relevant quantum computers become available, Shor's contributions would destroy the security basis for most real deployed public-key (asymmetric) cryptosystems, while Grover's could threaten the symmetric solutions.

The Global Risk Institute has published a 2021 Quantum Threat Timeline Report [119] focusing on estimates for the timeline of the threat posed to cybersecurity by quantum computers. This survey provides the most recent opinions of almost fifty experts from academia and industry on several aspects of quantum computing.

Table 2.5, reproduced from this report, shows what to expect for different timeframes based on the expert's estimates of the likelihood of a quantum computer being able to break the RSA-2048 algorithm in 24 hours.

This report also indicates that superconducting systems and trapped ions are the most promising technology for realizing a cryptography-threatening quantum computer.

As mentioned before, all qubit modalities are subject to environmental factors that turn into different noise sources, reducing the operational fidelity of the quantum bits. In order to cope with decoherence, many physical qubits are needed to make up a single logical qubit, providing stability, error correction, and fault tolerance to quantum gates and circuits.

Timeframe	What to Expect
Next 5 Years	Most experts (25/46) judged that the threat to current public-key cryptosystems in the next 5 years is "<1% likely". About a quarter of them (11/46) judged it relatively unlikely ("<5% likely"). The rest selected "<30%" (9/46) or "about 50%" (1/46) likely, suggesting there is a non-negligible chance of an impactful surprise within what would certainly be considered a very short-term future.
Next 10 Years	Still more than half of the respondents (24/46) judged the event was "<1%" or "<5%" likely, but already 15/46 felt it was "about 50%" or ">70%" likely, suggesting there is a significant chance that the quantum threat becomes concrete in this timeframe.
Next 15 Years	More than half (28/46) of the respondents indicated "about 50%" likely or more likely, among whom 13 indicated a ">70%" likelihood, and 5 an even higher ">95%" likelihood. This timeframe appears as a tipping point, as the number of respondents estimating a likelihood of "about 50%" or larger become the majority.
Next 20 Years	Roughly 90% (41/46) of respondents indicated "about 50%" or more likely, with 21/46 pointing to ">95%" or ">99%" likely. This indicates there is a significant bias toward viewing the realization of the quantum threat as substantially more likely than not within this timeframe.
Next 30 Years	Forty experts out of 46 indicated that the quantum threat has a likelihood of 70% or more this far into the future, with 16/44 experts indicating a likelihood greater than 99%. Thus, there appears to be a relatively low expectation of any fundamental showstoppers or other reasons that a cryptographically relevant quantum computer would not be realized in the long run.

Table 2.5 – What to expect for different timeframes based on the expert's estimates of the likelihood of a quantum computer able to break the RSA-2048 algorithm in 24 hours.

A consensus study report published in 2019 by The National Academies of Sciences, Engineering, and Medicine [120] helps clarify the current state of the art, likely progress toward, and ramifications of, a general-purpose quantum computer.

Table 2.6, partially extracted from this publication, shows some literature-reported estimates of quantum resilience for current cryptosystems under different assumptions of error rates and error-correcting codes.

Cryptosystem	Category	Key Size	Security Level	Threatening Quantum Algorithm	Logical Qubits Required	Physical Qubits Required	Time Required to Break the Cryptosystem
AES-GCM	Symmetric Encryption	128 192 256	128 192 256	Grover's algorithm	2,953 4,449 6,681	4.61×10^6 1.68×10^7 3.36×10^7	2.61×10^{12} years 1.97×10^{22} years 2.29×10^{32} years
RSA	Asymmetric Encryption	1024 2048 4096	80 112 128	Shor's algorithm	2,050 4,098 8,194	8.05×10^6 8.56×10^6 1.12×10^7	3.58 hours 28.63 hours 229 hours
ECDH	Asymmetric Encryption	256 384 512	128 192 256	Shor's algorithm	2,330 3,484 4,719	8.56×10^6 9.05×10^6 1.13×10^6	10.5 hours 37.67 hours 55 hours

Table 2.6 – Literature-reported estimates of quantum resilience for current cryptosystems.

These rough estimates, based on underlying assumptions for construction architecture, error rate, and gate speed, show that the number of physical qubits required to implement the threatening algorithms is thousands of times larger than what is provided by current quantum computers, and, moreover, the time required to break symmetric cryptosystems is still astronomical.

As shown in Figure 2.21 [121], IBM has just unveiled its 433-qubit Osprey quantum processor and expects to scale to 4,158 qubits in 2025, going beyond single chips and introducing classical parallelized quantum computing.

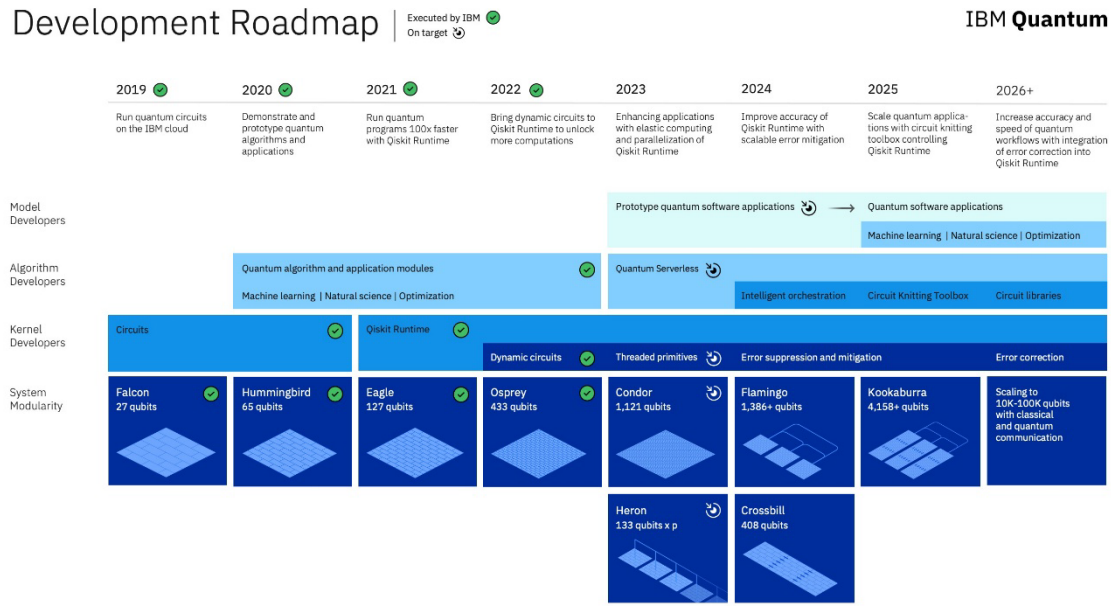


Figure 2.21 – IBM quantum roadmap.

A great deal of technological evolution is needed for quantum computers to achieve the necessary capacity to endanger classical cryptography. Despite all the engineering challenges, though, asymmetric cryptosystems will eventually be compromised. As a response to this quantum crisis, many efforts have been made towards the so-called post-quantum cryptography (PQC) [122, 123], which is believed to resist quantum computer attacks.

This relatively young research area focuses on devising mathematical operations and algorithms for quantum-safe classical cryptography. There are many important classes of cryptographic systems beyond those threatened by quantum computing, such as hash-based, code-based, lattice-based, and multivariate-quadratic-equations cryptography [123].

The capability of breaking RSA and elliptic curve cryptography together with a store-and-break threat known as store now and decrypt later (SNDL) — a cyber attack in which valuable encrypted data are stored and maintained to be decrypted later, once sufficiently large and fault-tolerant quantum computers are available — drives organizations and standardization bodies into a transition to a new suite of quantum-resistant encryption (QRE) algorithms [124].

As the quantum threat is expected to have a large and disruptive impact on the current digitally dependent economy, the World Economic Forum also prepared a study [125] based on in-depth discussions between senior leaders and quantum experts from its quantum security working group, which provides guidance for organizations to achieve a secure quantum transition setting short, medium, and long-term goals to manage the risk.

The report also highlights emerging technologies that can help mitigate the quantum threat: post-quantum cryptography, quantum key distribution, and quantum random number generation (QRNG), which is used as a source of genuine randomness to maximize the strength of a key.

In summary, as previously discussed and according to the estimates shown in Table 2.6, although public-key systems may be ruined within a few decades, the AES-GCM and other symmetric cryptosystems are in the class of quantum-resistant encryption algorithms. Therefore, assuming a secure key exchange between the parties, symmetric encryption with large keys could be used to protect information privacy, as quantum computers are not expected to be able to feasibly reduce the brute-force attack time.

Chapter 3

OTN CRYPTOGRAPHIC SYSTEM ARCHITECTURE

The previous chapter introduced optical transport technologies and key concepts related to data encryption and authentication. This chapter describes the entire architecture of the cryptographic system developed for the establishment of secure links in OTN systems, with emphasis on the hardware and software layer division and on the generation and transport of the encryption overhead data necessary for the link operation.

3.1 OTN CRYPTOGRAPHIC LINK

Although highly efficient and protected against transmission errors, OTN systems were not originally prepared to establish a secure communication channel with data confidentiality and authenticity. The implementation of a cryptography solution demands adaptations that allow the transport of additional data corresponding to the encryption overhead, as well as the design of mechanisms for the generation of IV sequences, establishment and maintenance of cryptographic sessions, management and exchange of keys, among others.

The high-level system architecture diagram depicted in Figure 3.1 illustrates how to enable cryptography in a 100 Gbit/s optical transport network using the developed 100G AES-GCM Cryptography Engine. It must be integrated into the OTN processing chain (either in an ASIC/ASSP or in an FPGA) at the ODU layer.

The developed architecture is based on a hybrid cryptosystem, as previously shown in Figure 2.10, with the encryption and authentication functions implemented in a hardware layer using the AES-GCM algorithm. The generation and exchange of large symmetric keys are implemented in a software layer (running in the Host CPU), and the use of an asymmetric protocol such as Diffie–Helman or RSA, or even quantum key distribution, can provide a secure method for key exchange over any channel, including the inherently insecure OTN link.

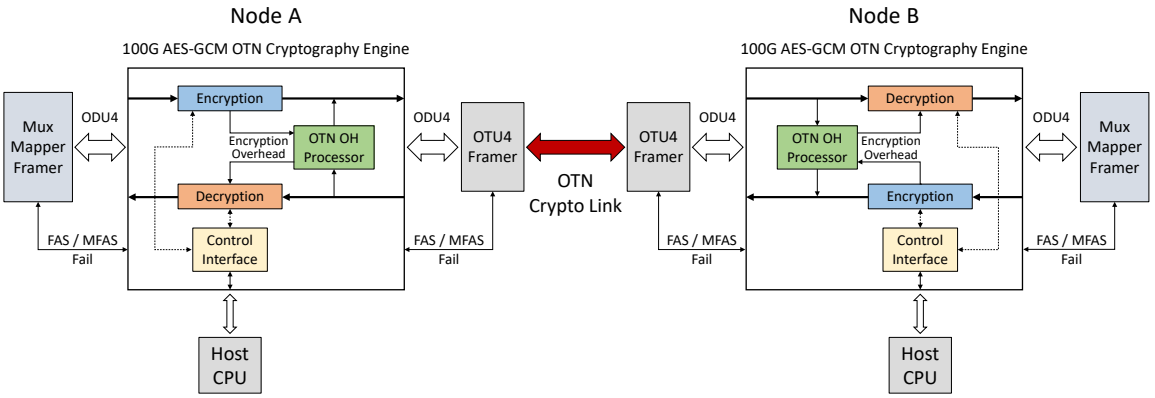


Figure 3.1 – 100G OTN cryptography high-level system architecture diagram.

Both inputs and outputs of the cryptography engine blocks handle a padded ODU frame (rows with 3840 columns) with MFAS and FAS signaling. The higher OTU layer is processed by other blocks (e.g. a Framer) in the OTN processor device. Encryption and decryption engines work just on the OPU area, and an OTN OH Processor inserts and extracts the cryptographic auxiliary information (TAG, AAD, and IV) using ODU reserved overhead fields. This provides the advantage of transparency in regenerations along the link or across multiple transport networks, where only the OTU overhead is processed.

With the use of padding, data corresponding to one line of the padded ODU frame will be transmitted in 48 clock cycles using a 640-bit bus. The last 128 bits of each line (or the least significant bits of the data bus in the 48th clock cycle) are padding bits (zeros) and must be ignored. Data processing is synchronized by the MFAS and FAS signaling.

3.2 Crypto Session, Crypto Block, and Crypto Packet

Before the establishment of an OTN secure communication link (OTN Crypto Link) between these two nodes, both sides need to communicate and exchange information to initiate Crypto Sessions, each one with a different cipher key. Once a crypto session is initiated, all OPU data are encrypted using the same key. In order to benefit from the inherent OTN multi-frame alignment feature, crypto sessions are further divided into a set of Crypto Blocks, all aligned with MFAS = 0.

Figure 3.2 shows the minimum transport unit for the encrypted data together with the encryption overhead (TAG, AAD, and IV), named Crypto Packet.

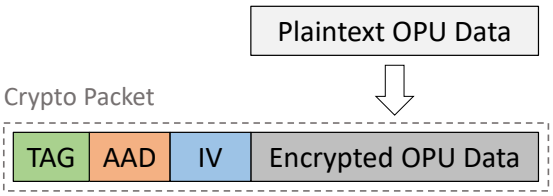


Figure 3.2 – 100G OTN crypto packet frame construction.

The minimum number of OPU payloads in the crypto packet is a function of the encryption overhead size and the available bandwidth in the ODU overhead fields. Since the IV and TAG are 96 and 128-bit sequences, at least 28 bytes are necessary. Assuming the use of the 2-byte and the 6-byte reserved fields of the ODU overhead, the available bandwidth is eight bytes per ODU frame. Then, with a minimum of four padded ODU frames (four OPU payloads in the crypto packet), a bandwidth of 32 bytes is available for the encryption overhead, providing a transport container for the cryptographic sideband data, including four bytes of AAD.

The crypto block, in this case, will have 64 crypto packets since it is aligned with an OTN multi-frame.

Table 3.1 summarizes these architectural conception definitions with additional remarks.

	Definition	**Remarks**
Crypto Session	Timeframe within which data are securely transmitted in an OTN Crypto Link using a single cipher key, which is generated by software on the transmitter side and shared with the receiver side using an asymmetric encryption protocol over a software communication link.	Minimum period dependent on the software layer response time and the software communication link bandwidth. It is a multiple of crypto blocks, which are aligned with OTN multi-frames (256 OPUs). Maximum period determined, in practice, by security policies and the number of unique IV sequences that can be generated within the session.
Crypto Block	Subdivision of a crypto session used to benefit from the OTN multi-frame alignment signal (MFAS).	Group of 64 crypto packets, each with four OPUs. Period = 299.008 µs (OTU4). Aligned with OTN MFAS = 0.
Crypto Packet	Transport unit for encrypted data together with encryption overhead (TAG, AAD, and IV), digitally wrapped into four ODU frames.	32 bytes for encryption overhead, transmitted in the eight reserved bytes available in each ODU overhead. 60960 bytes for encrypted data (four OPUs), transmitted in the OPU overhead and payload areas. Period = 4.672 µs (OTU4).

Table 3.1 – Definitions of crypto session, crypto block, and crypto packet.

3.3 HARDWARE AND SOFTWARE LAYERS

The hardware layer, made up of two processors (TX and RX), performs OPU data encryption, decryption, and authentication, as well as transmission and reception of crypto packets between two nodes and performance monitoring for statistics generation.

The software layer is mainly responsible for the hardware layer control (through a set of configuration registers), crypto session establishment and management, and cipher key management. An OTN in-band side channel called general communication channel (GCC), typically used to carry transmission management and signaling information, can be used as the software layer communication link.

3.3.1 HARDWARE LAYER CONTROL

The hardware layer is configured and commanded by the software layer. For example, cryptographic sessions can be restarted, or session periods can be changed at any time by software commands. However, for security reasons, a set of predefined hardware actions (called consequent actions) were also designed and implemented for faster response or for the case of software execution or communication failures.

3.3.2 ESTABLISHMENT AND MANAGEMENT OF CRYPTO SESSIONS

A crypto session is established through certificate exchange (with authentication) and sharing of a session key. During an active session, the software layer needs to share a cipher key for the next one. In this way, a new crypto session starts when the current one expires without traffic disturbances, using a hitless key change mechanism.

The common crypto session establishment procedure is as follows:

1. Nodes A and B communicate through a software communication link using an insecure channel;
2. Both nodes exchange certificates and verify the authenticity of each other;
3. Node A shares a 256-bit key to node B, using an asymmetric encryption protocol;
4. Nodes A and B use the shared session key for encryption and decryption of the OPU data in the crypto packets.

The session period, in terms of the number of crypto blocks, is defined by the software layer according to security policies and programmed into a hardware configuration register.

3.3.3 KEY MANAGEMENT

The software layer must generate the cryptographic symmetric key (session key) used for crypto packet encryption/decryption. It is then shared with the receiver side within the current crypto session period, so that it can be used in the next one. The generation of large ephemeral (short-term) session keys contributes to the security of the overall solution.

3.4 ENCRYPTION OVERHEAD FRAME

Figure 3.3 illustrates the correlation between the encryption overhead and the ciphered message of three consecutive crypto packets. The IV field of a crypto packet is related to the ciphered message of the next one, because at the receiver side it is needed in advance for OPU data decryption. In the same way, the TAG field of a crypto packet is related to the ciphered message of the previous one, since it will be calculated by the transmitter side after the entire message has been encrypted.

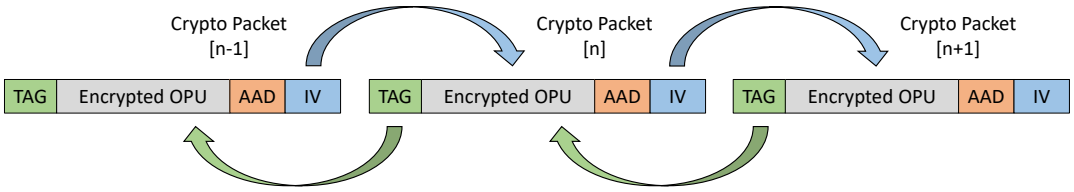


Figure 3.3 – Correlation of the encryption overhead and the encrypted message of consecutive crypto packets.

Table 3.2 shows the corresponding encryption overhead frame format and the field sizes adopted in this architecture, considering the available bandwidth of 32 bytes per crypto packet, with an indication of hardware (HW) and software (SW) layer management.

Field	Description	Size (byte)	Managed by
TAG	Message Authentication Code (MAC)	16	HW
AAD	Additional Authenticated Data	4	SW
IV	CSKS – Crypto Session Key Selection	1	SW/HW
	CSID – Crypto Session ID	4	SW
	CBID – Crypto Block ID	4	HW
	CPID – Crypto Packet ID	3	HW

Table 3.2 – Encryption overhead frame format and field sizes.

3.4.1 AUTHENTICATION TAG

The TAG field corresponds to the message authentication code (MAC) generated by the AES-GCM algorithm and is used by the receiver to verify the authenticity of the received message.

It is transmitted first in order to reduce data store-and-forward latency during the authentication process.

3.4.2 AAD

Additional authenticated data (AAD) comprises 4-bytes of plaintext per crypto packet.

It can be used by the optical transport network operator to transmit some supervision or management-related data, without the need for encryption, but ensuring authenticity.

3.4.3 IV

The initialization vector (IV) follows a deterministic construction with the concatenation of two fields, called the "fixed" field and the "invocation" field [100].

In this developed architecture, the fixed field is made up of a Crypto Session Key Selection (CSKS) control byte, together with a Crypto Session Identification (CSID) 4-byte number. The invocation field, which ensures the uniqueness requirement of a different IV per invocation of the authenticated encryption function [100] (in this case a unique IV per crypto packet), was implemented as integer counters corresponding to a Crypto Block Identification (CBID) 4-byte number and a Crypto Packet Identification (CPID) 3-byte number.

Since the CBID field is incremented at every multi-frame (64 crypto packets) of an active crypto session, its size defines, regardless of the CPID free-running counter value, the total number of $2^{32} \times 64$ unique IVs per crypto session. Considering the crypto packet period of 4.672 μs, this leads to a maximum session period of 14.86 days. Obviously, security policies will limit this period to a much shorter range.

The CSKS control byte (part of the IV fixed field) is used to select one of the crypto session keys (0 or 1) for decryption, as part of a hitless key change mechanism. In order to mitigate channel-induced errors in the transmission of this information, it is encoded in an 8-bit code word using a simple FEC strategy known as repetition code.

The receiver side then extracts this key selection information using majority decoding applied on this CSKS code word over 63 crypto packets, with the protection being accomplished by an 8 × 63 = 504 repetition code.

3.4.4 Encryption Overhead Transmission Format

Figure 3.4 shows the ODU overhead fields of an OTN frame.

In this developed architecture, the encryption overhead is transmitted in the eight reserved (RES) bytes of all the four ODU frames within a crypto packet.

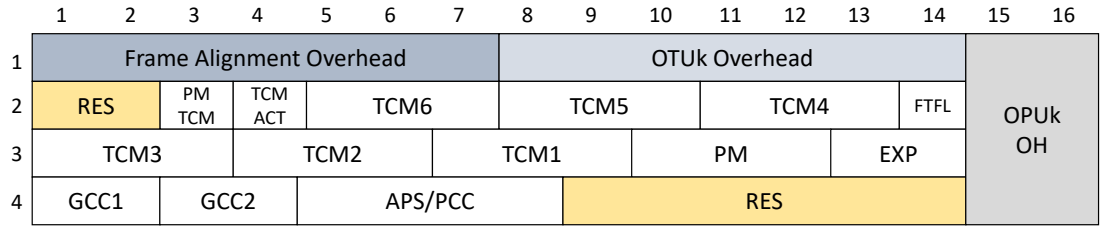
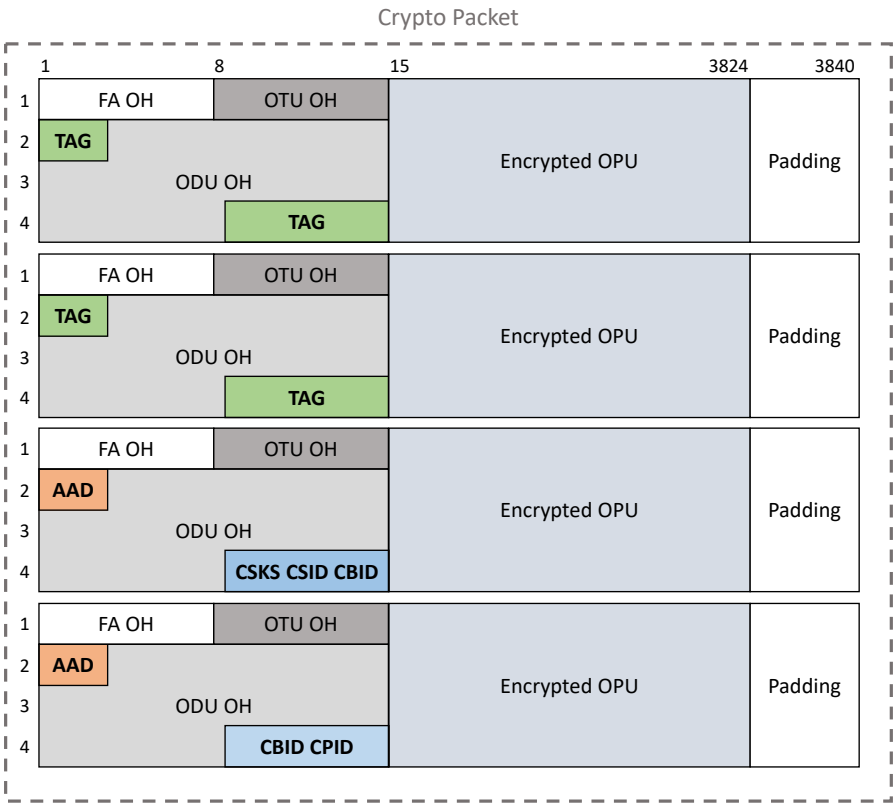


Figure 3.4 – ODU overhead fields of an OTN frame.

The encryption and decryption engines benefit from the availability of the OTN MFAS to delineate the boundaries of the ODU frames within the crypto packets, in order to insert and extract encryption overhead data.

Figure 3.5 shows a crypto packet made up of four padded ODU frames, as well as the distribution (location) of the encryption overhead data (16 bytes of TAG, 4 bytes of AAD, and 12 bytes of IV) in their relative positions within the RES fields of the ODU overhead (lines 2 and 4 of each frame).



MFAS [1:0]	RES (row, col)							
	(2, 1)	(2, 2)	(4, 9)	(4, 10)	(4, 11)	(4, 12)	(4, 13)	(4, 14)
0	TAG	TAG	TAG	TAG	TAG	TAG	TAG	TAG
1	TAG	TAG	TAG	TAG	TAG	TAG	TAG	TAG
2	AAD	AAD	CSKS	CSID	CSID	CSID	CSID	CBID
3	AAD	AAD	CBID	CBID	CBID	CPID	CPID	CPID

Figure 3.5 – Crypto packet made up of four padded ODU frames and distribution (location) of the encryption overhead bytes.

3.5 Error or Failure Recovery and Session Management

The hardware layer must be able to resynchronize from OTU link failures, such as those caused by signal degradation or loss of frame alignment, with minimal traffic disturbances when possible, as well as operate in a preconfigured way (through the consequent actions) in case of security-compromising failures.

Both software stacks running in the nodes A and B must also resynchronize is such circumstances. This can be accomplished by checking the hardware layer performance monitoring and status registers. Furthermore, they must also be able to recover in case of problems in the software layer itself, such as certificate or authentication failures.

3.5.1 HARDWARE RECOVERY AND CONSEQUENT ACTIONS

Hardware layer relies on the multi-frame alignment (and counter) signal to delineate crypto packets, blocks, and sessions. In the case of frame slips or any alignment errors, as well as client signal or OTN link failures, preconfigured consequent actions will be triggered.

Since the alarm events are also reported to the software layer by CPU interrupts, security policies may also determine other prevalent software-commanded actions.

Table 3.3 shows the programmable consequent actions for both TX and RX processors, which are only applicable for active crypto sessions.

Triggering Event	**TX Processor Consequent Actions**	**RX Processor Consequent Actions**
Session Expiration	1. None. 2. Assert *ipy_ssf_tx* and close the crypto session in the current direction. 3. Assert *ipy_ssf_tx* and close both crypto sessions in the current and opposite directions. 4. Reuse current key during a limited period determined by *tx_key_reuse_period* register. When the key reuse period expires, assert *ipy_ssf_tx* and close the crypto session in the current direction. 5. Reuse current key during a limited period determined by *tx_key_reuse_period* register. When the key reuse period expires, assert *ipy_ssf_tx* and close both crypto sessions in the current and opposite directions. 6. Close the crypto session in the current direction. 7. Close both crypto sessions in the current and opposite directions. 8. Reuse current key during a limited period determined by *tx_key_reuse_period* register. When the key reuse period expires, close the crypto session in the current direction. 9. Reuse current key during a limited period determined by *tx_key_reuse_period* register. When the key reuse period expires, close both crypto sessions in the current and opposite directions.	1. None. 2. Assert *ipy_ssf_rx* and close the crypto session in the current direction. 3. Assert *ipy_ssf_rx* and close both crypto sessions in the current and opposite directions. 4. Reuse current key during a limited period determined by *rx_key_reuse_period* register. When the key reuse period expires, assert *ipy_ssf_rx* and close the crypto session in the current direction. 5. Reuse current key during a limited period determined by *rx_key_reuse_period* register. When the key reuse period expires, assert *ipy_ssf_rx* and close both crypto sessions in the current and opposite directions. 6. Close the crypto session in the current direction. 7. Close both crypto sessions in the current and opposite directions. 8. Reuse current key during a limited period determined by *rx_key_reuse_period* register. When the key reuse period expires, close the crypto session in the current direction. 9. Reuse current key during a limited period determined by *rx_key_reuse_period* register. When the key reuse period expires, close both crypto sessions in the current and opposite directions.
***ipy_tsf_tx* input port assertion**	1. None. 2. Close the crypto session in the current direction. 3. Generate replacement signal and keep the crypto session active in the current direction.	

Triggering Event	TX Processor Consequent Actions	RX Processor Consequent Actions
ipy_tsf_rx **input port assertion**		1. None. 2. Close the crypto session in the current direction. 3. Close both crypto sessions in the current and opposite directions.
TAG Mismatch (TAG_FAIL)		1. None. 2. Close the crypto session in the current direction. 3. Close both crypto sessions in the current and opposite directions. 4. Assert *ipy_ssf_rx* and close the crypto session in the current direction. 5. Assert *ipy_ssf_rx* and keep the crypto session active in the current direction. 6. Assert *ipy_ssf_rx* and close both crypto sessions in the current and opposite directions.
Loss of Authentication (LOA)		1. None. 2. Close the crypto session in the current direction. 3. Close both crypto sessions in the current and opposite directions. 4. Assert *ipy_ssf_rx* and close the crypto session in the current direction. 5. Assert *ipy_ssf_rx* and keep the crypto session active in the current direction. 6. Assert *ipy_ssf_rx* and close both crypto sessions in the current and opposite directions.

Table 3.3 – Hardware layer programmable consequent actions.

3.5.2 SOFTWARE RECOVERY

Software protection mechanisms must be implemented to ensure recovery in different failure scenarios. In the case of certificate validation problems, for instance, the execution flow must return to an idle state, waiting for a new communication attempt to be established.

Supervisory routines (watchdog threads) can also be used to return to this same idle state in cases where there is loss of communication or failure to receive the expected asymmetric encryption protocol messages.

3.5.3 LOSS OF AUTHENTICATION AND TAG MISMATCH

The decryption engine verifies TAG matching at every crypto packet and asserts TAG_FAIL indication for every mismatch condition. When operating in authenticated encryption or authentication-only modes, it can output decrypted or authenticated data only after the TAG sequence is verified or as soon as data arrive, regardless of the matching condition.

In the first case, the store-and-forward delay will be equivalent to six ODU frames (hardware architecture pipelining not included). This time corresponds to the reception of a complete crypto packet (four ODU frames) followed by the first two frames of the next one (for the extraction of the transmitted TAG sequence, as represented in Figure 3.3).

Conversely, a cut-through mode can be configured to eliminate the authentication delay, and, in this case, data authenticity can still be signaled upon the reception of the first two ODU frames of the following crypto packet (from where the transmitted TAG sequence will be extracted).

For security reasons, an authentication blocking condition can also be configured to allow OPU data (overhead and payload) to be replaced by zeros in the crypto packets with TAG_FAIL indication.

In addition to the TAG matching verification, a loss of authentication (LOA) indication is evaluated by a sliding window mechanism (with programmable size and threshold). LOA can trigger consequent actions or can be used by the software layer, but does not directly control data output.

Both TAG_FAIL and LOA are asserted only if the crypto session is active. Otherwise, they are cleared.

3.5.4 SESSION-ABOUT-TO-EXPIRE INTERRUPT

Although the crypto session period is determined by the software layer, a CPU interrupt is generated by hardware to indicate when the current session is about to expire. This is done as a function of the crypto block count and a programmable register indicating the threshold above which an interrupt should be raised. This allows the software to tweak the interrupt

generation based on how long it will take for the transmitter and receiver to negotiate a new crypto session.

3.5.5 SESSION KEY REUSE

If the software layer is temporarily executing operations of higher priority and the crypto session period expires without a new key being provided, a configuration register will control whether the hardware layer will drop the crypto link (closing the crypto session) or reuse the current session encryption key.

In this case, a programmable counter limiting the number of reiterations ensures security, since the link is dropped when the limit is reached.

3.6 CONFIGURATION AND PERFORMANCE MONITORING PROCEDURES

This section describes basic software layer procedures for the establishment, maintenance, and monitoring of crypto sessions.

3.6.1 CRYPTO SESSION ESTABLISHMENT

Assuming that both TX and RX side crypto sessions are closed, the following procedures are expected for the software layer to establish a new one:

3.6.1.1 RX SIDE

1. Write the new key in the “key 0” register set;
2. Write ‘1’ in the *rx_session_state* register to request session establishment.

After step 2, the RX processor engine waits for a multi-frame start pulse and then keeps decoding the CSKS (Crypto Session Key Selection) FEC information until the selected key is ‘0’ at MFAS = 252. After that, it changes the key and establishes the crypto session in the beginning of the next crypto packet, at MFAS = 0.

3.6.1.2 TX SIDE

3. Write the new key in the “key 0” register set;
4. Write ‘1’ in the *tx_key_change* register to request a key change process (this bit is cleared by hardware);
5. Write ‘1’ in the *tx_session_state* register to request session establishment.

When crypto sessions are closed, the TX processor engine continuously sends a CSKS value of 0xff to the RX side, corresponding to the selection of "key 1".

After step 5, it waits for a multi-frame start pulse and then sends a new CSKS with value 0x00 to the RX side, corresponding to the selection of "key 0". This new key selection information will be decoded at MFAS = 252, and the crypto session will be established in the beginning of the next crypto packet, at MFAS = 0.

Before writing '1' in the *tx_key_change*, the software layer must also have provided a new CSID value in the *tx_iv_csid* register.

3.6.2 CRYPTO SESSION KEY CHANGE

Assuming that both TX and RX side crypto sessions are active, the following procedures are expected for the software layer:

3.6.2.1 TX SIDE

1. Read the *tx_session_status* register to verify the active key ('0' or '1');
2. Write the new key in the inactive key register set;
3. Write '1' in the *tx_key_change* register to request a key change process (this bit is cleared by hardware).

After step 3, the TX processor waits for a multi-frame start pulse and then sends a CSKS value corresponding to the inactive key ('1' or '0'). This information will be decoded by the RX processor at MFAS = 252, and a new session (with the new key) will start in the beginning of the next crypto packet, at MFAS = 0.

3.6.3 USE OF ADDITIONAL AUTHENTICATED DATA (AAD)

The transmitted AAD value is updated at every crypto packet, but it is only captured by hardware from the corresponding *tx_aad_buffer* register after '1' is written in the *tx_aad_capture* register (a self-clearing bit).

In this way, the available bandwidth for AAD transmission is limited not only by the virtual channel (4 bytes per crypto packet), but also by the software control interface response time and the application execution speed.

3.6.4 PERFORMANCE MONITORING COUNTERS

Both TX and RX processors provide a set of performance monitoring counters/registers.

Before reading operations, the software layer must write '1' in a special self-clearing register (*tx_cnt_latch* or *rx_cnt_latch*) dedicated to latch and holding all values.

For the TX processor, the latched values correspond to the following counters/registers:

- Crypto block ID;
- Crypto packet ID;
- Calculated TAG.

For the RX processor, the latched values correspond to the following counters/registers:

- Crypto session ID;
- Crypto block ID;
- Crypto packet ID;
- Calculated TAG;
- Received TAG;
- Crypto session counter;
- Crypto packet counter;
- LOA fail counter;
- TAG fail counter;
- CSKS (bit 0) counter;
- CSKS (bit 1) counter;
- CSKS (bit 0) histogram counter;
- CSKS (bit 1) histogram counter.

In the RX processor only, each counter also contains its own dedicated clearing register.

Chapter 4

100G AES-GCM CRYPTOGRAPHY ENGINE

This chapter describes the 100G AES-GCM Cryptography Engine, designed and implemented according to the conceived systemic architecture solution shown in the previous chapter. Its main features, characteristics, operation modes, and configuration procedures are presented, along with a detailed description of its hardware functional architecture and partitioning of the logic building blocks, which were later modeled and implemented in a hardware description language.

4.1 FEATURES AND CHARACTERISTICS

The following list presents the main technical and operational features of the developed 100G AES-GCM Cryptography Engine.

- Supports Galois/Counter Mode;
- 256-bit key AES (Rijndael) algorithm with optional key derivation function for enhanced security;
- 128-bit message authentication code (MAC);
- 96-bit initialization vector (IV);
- 640-bit bus data path interfaces;
- Receives and transmits padded ODU frames;
- Encryption and decryption affects just the OPU area (overhead and payload);
- Transport unit for encrypted data and encryption overhead (crypto packet) made up of four ODU frames;
- Encryption overhead (TAG, AAD, and IV) transmitted in the reserved (RES) fields of the ODU overhead with optional scrambling for enhanced security;
- Three operation modes: authenticated encryption, authentication-only, and encryption-only;
- Store-and-forward delay equivalent to six ODU frames when operating in authenticated encryption or authentication-only modes;

- Cut-through mode for delivering data with delayed TAG matching indication;
- Bypass and loopback functions;
- Programmable consequent actions in case of failure or security-compromising conditions;
- Provides several counters for performance monitoring;
- Configurable flow control mode ("push/pull") for both TX and RX data paths;
- Independent clock signals for TX and RX processors;
- Software configurable via a 16-bit register map interface with maskable interrupts.

4.2 OPERATION MODES

The 100G AES-GCM Cryptography Engine solution can be configured to operate in three different modes, as detailed in Table 4.1.

Operation Mode	Description/Characteristics
Authenticated Encryption	• OPU data are encrypted by the AES-GCM algorithm; • OPU data and AAD are authenticated in the RX processor side; • Authentication modes: o Store-and-forward: OPU data are delivered only after a TAG matching condition (with a delay equivalent to six ODU frames); o Cut-through: OPU data are delivered upon reception and the TAG matching condition is signaled later.
Authentication-Only	• OPU data are transmitted as plain text (no encryption); • OPU data and AAD are authenticated in the RX processor side; • Authentication modes: o Store-and-forward: OPU data are delivered only after a TAG matching condition (with a delay equivalent to six ODU frames); o Cut-through: OPU data are delivered upon reception and the TAG matching condition is signaled later.
Encryption-Only	• OPU data are encrypted by the AES-GCM algorithm;

Table 4.1 – 100G AES-GCM Cryptography Engine operation modes.

4.3 HARDWARE FUNCTIONAL ARCHITECTURE

Figure 4.1 shows the block diagram of the 100G AES-GCM Cryptography Engine hardware functional architecture, composed of several sub-blocks with specialized functions. Names in parentheses correspond to the top-level modules of the respective models described in the Verilog hardware description language [126].

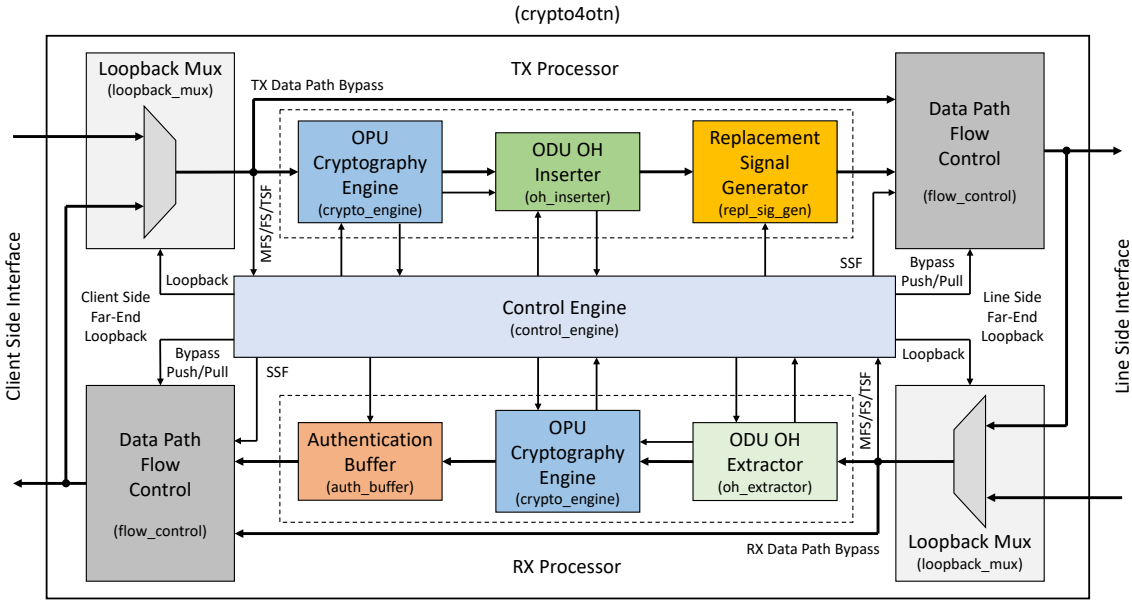


Figure 4.1 – 100G AES-GCM Cryptography Engine hardware functional architecture block diagram.

Data coming from the client side 640-bit interface are manipulated by the TX processor blocks (for encryption) and then output in the line side interface. In the opposite direction, decryption is performed by the RX processor blocks.

The Crypto4OTN block can be configured to operate in authenticated encryption, authentication-only, and encryption-only modes. The user can also define the authentication mode as store-and-forward, when OPU data are delivered only after a TAG matching condition (with a delay equivalent to six ODU frames) or cut-through, when OPU data are delivered upon reception and the TAG matching condition is signaled later.

Right at the input of both client and line side interfaces, a Loopback Mux sub-block (a data selector multiplexer switch) provides far-end loopback functionalities. Correspondingly, at the output of both interfaces a Data Path Flow Control sub-block handles flow control and bypass.

The OPU Cryptography Engine sub-block implements the core functionalities for data encryption, decryption, and authentication. It receives and transmits padded ODU frames and handles OPU drop/add synchronization. Depending on the operation mode, this sub-block outputs encrypted or clear data as well as a 128-bit authentication TAG. It is instantiated in both TX (for encryption) and RX (for decryption) processor lineups.

In the TX processor chain, the encryption overhead bytes (TAG, AAD, and IV) are inserted in the eight RES fields of the ODU overhead by the ODU OH Inserter sub-block. Since four consecutive frames are needed (making up a crypto packet), the multi-frame start (MFS) signal is used to synchronize the insertion location distribution.

In the end of the TX processor lineup, a Replacement Signal Generator sub-block generates an OTN AIS-like (alarm indication signal) or a user-defined pattern replacement signal to be used in case of authentication mismatch or any error or failure condition, according to register programmed consequent actions.

In the RX processor chain, the ODU OH Extractor sub-block extracts the encryption overhead bytes while the remaining input data are forwarded to the output port, except for the ODU overhead RES fields, which are filled with '0' in the output frame.

A second instance of the OPU Cryptography Engine then decrypts data, and, finally, an Authentication Buffer sub-block stores six ODU frames and forwards them after TAG matching verification when the authentication mode is configured for store-and-forward. Data must be buffered to wait for authentication because the TAG value generated in the transmitter (to be compared with the one calculated in the receiver) of a given crypto packet comes at the beginning of the next one, as previously shown in Figure 3.3.

The operation of the Crypto4OTN block is entirely orchestrated by the Control Engine sub-block. It was designed upon the use of two finite state machines to control and synchronize the TX and RX processor lineups, together with a Register and Reset Controller that provides CPU access to a set of configuration registers.

The Control Engine handles crypto session establishment and maintenance procedures, controlling the hitless key change mechanism, building up the IV sequence, and performing a series of other functions requested by the software layer through the configuration registers. All the resulting controlling actions are time-adjusted for each sub-block, based on the FS, MFS, and MFAS synchronization signals.

4.3.1 Nomenclature for Signals and Buses

The architecture design and implementation of the 100G AES-GCM Cryptography Engine were based on methodologies that include the use of a standardized nomenclature for name

assignments in the interfaces of the external signals of its logic blocks and sub-blocks, making them reusable between different projects.

This standardization defines a consistent input and output interface between blocks, as well as a common strategy for reset mechanisms, clock signals, and debug and test modes.

Table 4.2 shows the standardized interfaces used for the input and output signals, divided and organized in a color code scheme according to the functional characteristics of their group. A corresponding prefix is also defined to compose the signal names, thus allowing the quick identification of their respective interface.

Interface (Color)	Signal Name Prefix	Description/Characteristics
Forest-Green Line	*ipf_*	Send/Receive data to/from a FIFO memory.
Green Line	*ipg_*	Global signals, such as *clock* and *reset*.
Yellow Line	*ipy_*	Main data path.
Indigo Line	*ipi_*	Interrupt signals.
Sky-Blue Line	*ips_*	Register access signals.
Tan Line	*ipt_*	Test signals.

Table 4.2 – Standardized interfaces used in the 100G AES-GCM Cryptography Engine.

4.3.2 CLOCK AND RESET

Both TX and RX processors operate in different clock domains, and, therefore, two global logic clock signals must be provided for the Crypto4OTN block. The register access interface also has a dedicated clock signal.

Table 4.3 shows the used clock signals.

Clock Signal	Frequency/Duty-Cycle	Description/Characteristics
ipg_clk_sys_tx	180 MHz / 50%	TX processor *green line* interface. Clock signal used in the internal global logic.
ipg_clk_sys_rx	180 MHz / 50%	RX processor *green line* interface. Clock signal used in the internal global logic.
ipg_clk	70–90 MHz / 50%	*Green line* interface for register access.

Table 4.3 – Clock signals used in the 100G AES-GCM Cryptography Engine.

For reset signals there are two related inputs, controlled by the circuit in which the Crypto4OTN block is integrated — *ipg_hard_async_reset_b* and *ipg_soft_reset_async_b*. The first corresponds to a hardware reset, while the second can be generated through software controls. Both are asserted (at a low logic level) and de-asserted asynchronously with the global logic clock signals, producing the same effect on the internal logic.

These signals are internally combined within the Control Engine sub-block and synchronized with the clock domains of both TX and RX processors, generating two new reset signals, which can also be controlled through a configuration register. They are all presented in Table 4.4.

Reset Signal	Clock Domain	Sources
tx_reset_sync_b	*ipg_clk_sys_tx*	*ipg_hard_async_reset_b* *ipg_soft_reset_async_b* Register: *blk_rst[0]*
rx_reset_sync_b	*ipg_clk_sys_rx*	*ipg_hard_async_reset_b* *ipg_soft_reset_async_b* Register: *blk_rst[8]*

Table 4.4 – Reset signals used in the 100G AES-GCM Cryptography Engine.

The configuration register values are preserved when the Crypto4OTN block is reset by the *blk_rst* register.

4.3.3 CRYPTO4OTN BLOCK

As previously presented, the Crypto4OTN block corresponds to the top-level implementation of the 100G AES-GCM Cryptography Engine solution.

Input and output data are structured in what is called a padded ODU frame, as illustrated in Figure 4.2. In this way, data corresponding to one line of that frame are transmitted in 48 clock cycles over a 640-bit bus. The last 128 bits of each line (the least significant ones of the data bus at the 48th clock cycle) are "padding" bits (zeros) and should be ignored.

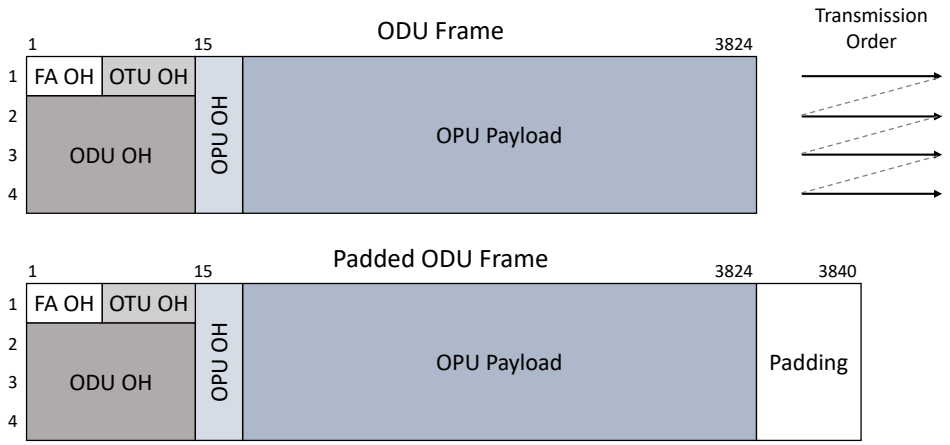


Figure 4.2 – Padded ODU frame handled by the Crypto4OTN block.

Figure 4.3 shows the representation of the Crypto4OTN block with its external input and output interfaces and the respective signals organized and separated between the TX and RX processors.

Table 4.5 brings the signal names with their interface directions (input/output) and corresponding descriptions.

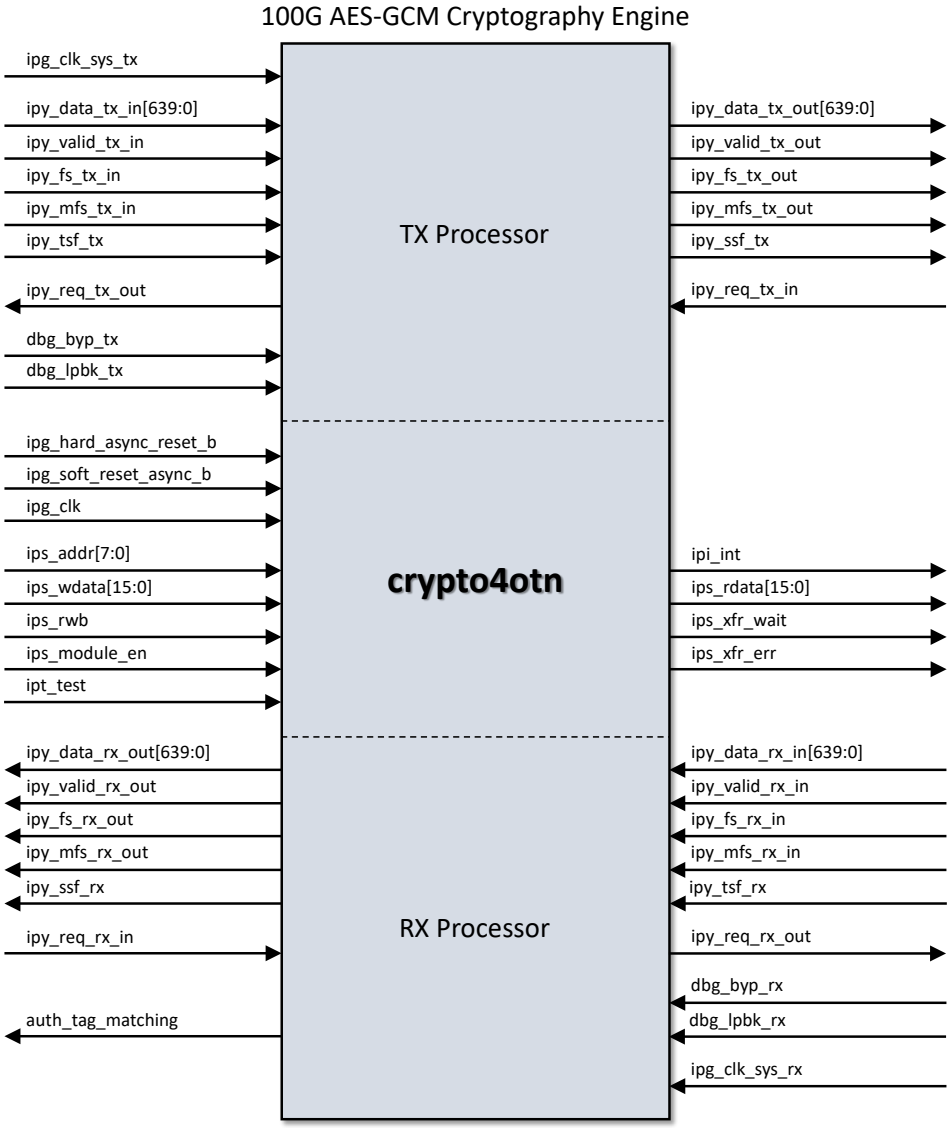


Figure 4.3 – Crypto4OTN block interface diagram.

Signal	I/O	Description
ipg_hard_async_reset_b	I	Green line asynchronous hard reset signal. This signal is asserted and de-asserted asynchronously. Active-low.
ipg_soft_reset_async_b	I	Green line interface soft reset signal. This signal is asserted and de-asserted asynchronously. Active-low.
ipg_clk	I	Green line interface clock at 90 MHz with duty cycle of 50%. Used for the register access interface.
ips_addr[7:0]	I	Sky-Blue line interface address bus. Used for register access.
ips_wdata[15:0]	I	Sky-Blue line interface write data bus. Used for register access.
ips_rwb	I	Sky-Blue line interface read/write selection. '0' – Data are written by the CPU. '1' – Data are read by the CPU.

Signal	I/O	Description
ips_module_en	I	Sky-Blue line interface module enable. Selects the crypto4otn block for register access. Active-high.
ipt_test	I	Test mode enable signal (for DFT purposes). Active-high.
ipg_clk_sys_tx	I	TX processor green line interface clock at 180 MHz with duty cycle of 50%. Used for internal global logic.
ipg_clk_sys_rx	I	RX processor green line interface clock at 180 MHz with duty cycle of 50%. Used for internal global logic.
ipy_data_tx_in[639:0]	I	Yellow line interface input data for the TX processor. Receives padded ODU frames coming from the client side with data to be encrypted.
ipy_valid_tx_in	I	Yellow line interface input data valid. Validates the padded ODU input data at the *ipy_data_tx_in* bus, as well as the *ipy_fs_tx_in*, *ipy_mfs_tx_in*, and *ipy_tsf_tx* signals. Active-high.
ipy_fs_tx_in	I	Yellow line interface input data frame start pulse. Indicates the start of the ODU input data frame at the *ipy_data_tx_in* bus. Active-high.
ipy_mfs_tx_in	I	Yellow line interface input data multi-frame start pulse. Indicates the start of the ODU input data multi-frame at the *ipy_data_tx_in* bus. Active-high.
ipy_data_rx_in[639:0]	I	Yellow line interface input data for the RX processor. Receives padded ODU frames coming from the line side with data to be decrypted.
ipy_valid_rx_in	I	Yellow line interface input data valid. Validates the padded ODU input data at the *ipy_data_rx_in* bus, as well as the *ipy_fs_rx_in*, *ipy_mfs_rx_in*, and *ipy_tsf_rx* signals. Active-high.
ipy_fs_rx_in	I	Yellow line interface input data frame start pulse. Indicates the start of the ODU input data frame at the *ipy_data_rx_in* bus. Active-high.
ipy_mfs_rx_in	I	Yellow line interface input data multi-frame start pulse. Indicates the start of the ODU input data multi-frame at the *ipy_data_rx_in* bus. Active-high.
ipy_tsf_tx	I	TX processor input client data fail signal indication. Active-high.
ipy_tsf_rx	I	RX processor input line data fail signal indication. Active-high.
ipy_req_tx_in	I	TX processor input data request for data flow control.
ipy_req_rx_in	I	RX processor input data request for data flow control.
dbg_byp_tx	I	Debug port for TX processor bypass control, with priority over the register command. ‘0’ – Normal operation. ‘1’ – Bypass mode.
dbg_byp_rx	I	Debug port for RX processor bypass control, with priority over the register command. ‘0’ – Normal operation. ‘1’ – Bypass mode.
dbg_lpbk_tx	I	Debug port for client side far-end loopback control, with priority over the register command. ‘0’ – Normal operation. ‘1’ – Client loopback mode.
dbg_lpbk_rx	I	Debug port for line side far-end loopback control, with priority over the register command. ‘0’ – Normal operation. ‘1’ – Line loopback mode.

Signal	I/O	Description
ipi_int	O	Indigo line interface interrupt request signal. Active-high.
ips_rdata[15:0]	O	Sky-Blue line interface read data bus. Used for register access.
ips_xfr_wait	O	Sky-Blue line interface transfer wait signal. When in logic low '0' it indicates that the access is complete. Active-high.
ips_xfr_err	O	Sky-Blue line interface transfer error signal. Active-high.
ipy_data_tx_out[639:0]	O	Yellow line interface output data for the TX processor. Transmits padded ODU frames going to the line side with encrypted data.
ipy_valid_tx_out	O	Yellow line interface output data valid. Validates the padded ODU output data at the *ipy_data_tx_out* bus, as well as the *ipy_fs_tx_out*, *ipy_mfs_tx_out*, and *ipy_ssf_tx* signals. Active-high.
ipy_fs_tx_out	O	Yellow line interface output data frame start pulse. Indicates the start of the ODU output data frame at the *ipy_data_tx_out* bus. Active-high.
ipy_mfs_tx_out	O	Yellow line interface output data multi-frame start pulse. Indicates the start of the ODU output data multi-frame at the *ipy_data_tx_out* bus. Active-high.
ipy_data_rx_out[639:0]	O	Yellow line interface output data for the RX processor. Transmits padded ODU frames going to the client side with decrypted data.
ipy_valid_rx_out	O	Yellow line interface output data valid. Validates the padded ODU output data at the *ipy_data_rx_out* bus, as well as the *ipy_fs_rx_out*, *ipy_mfs_rx_out*, and *ipy_ssf_rx* signals. Active-high.
ipy_fs_rx_out	O	Yellow line interface output data frame start pulse. Indicates the start of the ODU output data frame at the *ipy_data_rx_out* bus. Active-high.
ipy_mfs_rx_out	O	Yellow line interface output data multi-frame start pulse. Indicates the start of the ODU output data multi-frame at the *ipy_data_rx_out* bus. Active-high.
ipy_ssf_tx	O	TX processor output line data fail signal indication. Active-high.
ipy_ssf_rx	O	RX processor output client data fail signal indication. Active-high.
ipy_req_tx_out	O	TX processor output data request for data flow control. Active-high. In "pull" mode, it is asserted when necessary to request data. In "push" mode, it is continuously asserted.
ipy_req_rx_out	O	RX processor output data request for data flow control. Active-high. In "pull" mode, it is asserted when necessary to request data. In "push" mode, it is continuously asserted.
auth_tag_matching	O	RX processor authentication TAG matching indication. Updated at every crypto packet of an active crypto session, after TAG verification. Active-high. This output port is disabled in the encryption-only operation mode.

Table 4.5 – Crypto4OTN block interface signals.

Data processing is synchronized by the multi-frame start (MFS) and frame start (FS) pulses. The first is asserted one clock cycle before the start of a multi-frame (256 frames), and the second, one clock cycle before the start of an ODU frame.

Input and output data buses are sampled (captured) only when the corresponding validation (*valid*) signal is asserted. In some applications, the input *valid* signal is de-asserted during

few clock periods for data rate adaptation. In this case, this is reflected to the output, with a similar waveform being generated after the inherent processing latency.

Figure 4.4 shows the Crypto4OTN TX processor input data timing diagram.

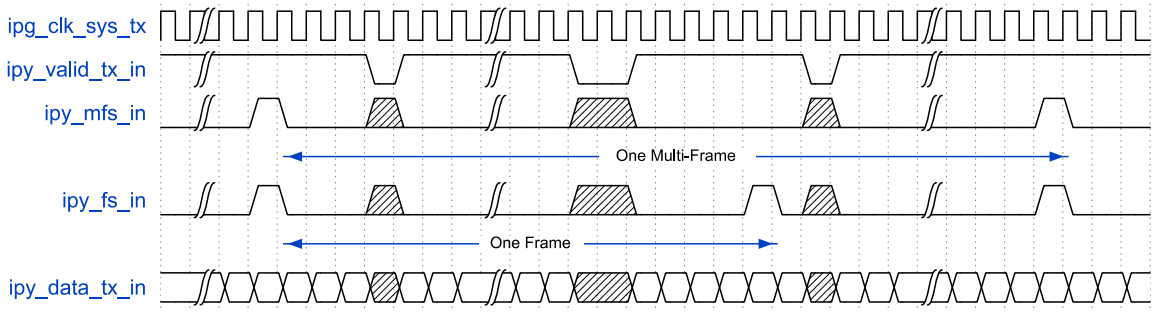


Figure 4.4 – Crypto4OTN TX processor input data timing diagram.

Multiplexer switches can be configured so that received data are completely bypassed directly to the respective output interfaces on both TX and RX processors (bypass), as well as looped back in the client and line side interfaces (far-end loopback).

The input fail indication signals (*ipy_tsf_tx* and *ipy_tsf_rx*) are registered to the respective output (*ipy_ssf_tx* and *ipy_ssf_rx*) with a delay equivalent to the Crypto4OTN total processing latency.

The output fail indication signals (*ipy_ssf_tx* and *ipy_ssf_rx*) may also be generated by the Control Engine sub-block in response to a consequent action triggered by predefined operating conditions.

An authentication TAG matching indication signal (*auth_tag_matching* output) is also generated (except in the encryption-only operation mode) and updated at every crypto packet of an active crypto session, after TAG verification by the RX processor.

4.3.4 LOOPBACK MUX SUB-BLOCK

The Loopback Mux sub-block corresponds to a data selector multiplexer switch used to provide far-end loopback functionalities.

Figure 4.5 shows its external input and output interface, and Table 4.6 brings the signal names, interface directions (input/output), descriptions, and the corresponding sources and destinations (from/to).

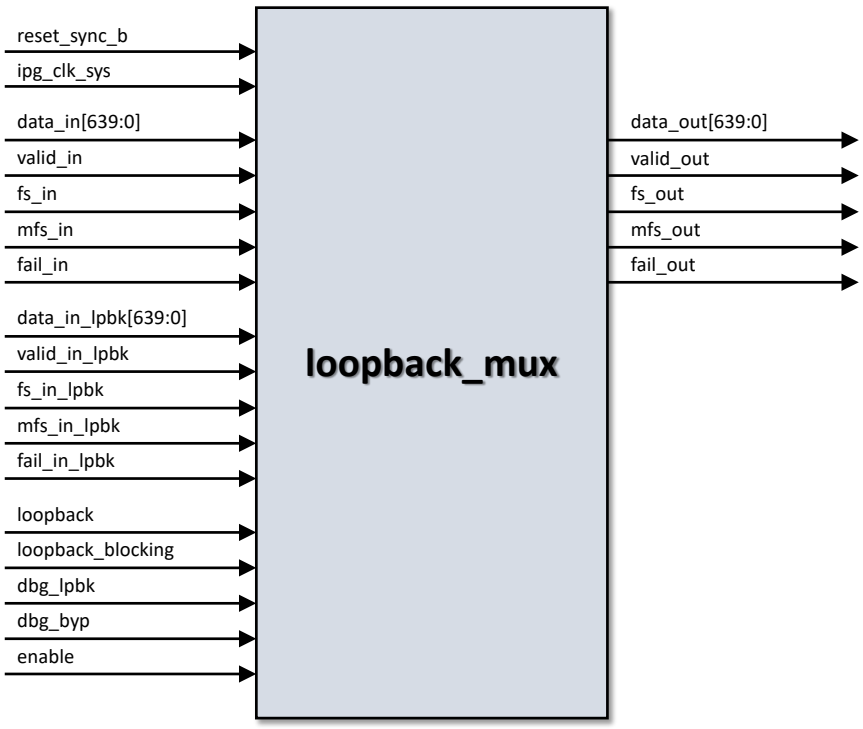


Figure 4.5 – Loopback Mux sub-block interface.

Signal	I/O	Description	From/To
reset_sync_b	I	Synchronous reset signal generated in the Control Engine sub-block. Active-low.	control_engine.tx_reset_sync_b (*) control_engine.rx_reset_sync_b (**)
ipg_clk_sys	I	Green line interface clock at 180 MHz with duty cycle of 50%. Used for internal global logic.	crypto4otn.ipg_clk_sys_tx (*) crypto4otn.ipg_clk_sys_rx (**)
data_in[639:0]	I	Padded ODU input data.	crypto4otn.ipy_data_tx_in (*) crypto4otn.ipy_data_rx_in (**)
valid_in	I	Input data valid. Validates the ODU input data at the *data_in* bus, as well as the *fs_in*, *mfs_in*, and *fail_in* signals. Active-high.	crypto4otn.ipy_valid_tx_in (*) crypto4otn.ipy_valid_rx_in (**)

Signal	I/O	Description	From/To
fs_in	I	Input data frame start pulse. Single clock period pulse that precedes the start of the ODU input data frame at the *data_in* bus. Active-high.	crypto4otn.ipy_fs_tx_in (*) crypto4otn.ipy_fs_rx_in (**)
mfs_in	I	Input data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU input data multi-frame at the *data_in* bus. Active-high.	crypto4otn.ipy_mfs_tx_in (*) crypto4otn.ipy_mfs_rx_in (**)
fail_in	I	Fail signal. Active-high.	crypto4otn.ipy_tsf_tx (*) crypto4otn.ipy_tsf_rx (**)
data_in_lpbk [639:0]	I	Padded ODU input data (loopback).	crypto4otn.ipy_data_rx_out (*) crypto4otn.ipy_data_tx_out (**)
valid_in_lpbk	I	Input data valid (loopback). Validates the ODU input data at the *data_in_lpbk* bus, as well as the *fs_in_lpbk*, *mfs_in_lpbk*, and *fail_in_lpbk* signals. Active-high.	crypto4otn.ipy_valid_rx_out (*) crypto4otn.ipy_valid_tx_out (**)
fs_in_lpbk	I	Input data frame start pulse (loopback). Single clock period pulse that precedes the start of the ODU input data frame at the *data_in_lpbk* bus. Active-high.	crypto4otn.ipy_fs_rx_out (*) crypto4otn.ipy_fs_tx_out (**)
mfs_in_lpbk	I	Input data multi-frame start pulse (loopback). Single clock period pulse that precedes the start of the ODU input data multi-frame at the *data_in_lpbk* bus. Active-high.	crypto4otn.ipy_mfs_rx_out (*) crypto4otn.ipy_mfs_tx_out (**)
fail_in_lpbk	I	Fail signal (loopback). Active-high.	crypto4otn.ipy_ssf_rx (*) crypto4otn.ipy_ssf_tx (**)
loopback	I	Loopback port selection control. '0' – Normal port selected for output. '1' – Loopback port selected for output.	control_engine.client_loopback (*) control_engine.line_loopback (**)
loopback_blocking	I	Loopback port selection blocking control, with priority over the *enable* signal, used here to inhibit the loopback port selection control. '0' – Loopback port selection enabled. '1' – Loopback port selection disabled (data loopback controlled just by the *dbg_lpbk* debug port).	crypto4otn.dbg_lpbk_rx (*) crypto4otn.dbg_lpbk_tx (**)

Signal	I/O	Description	From/To
dbg_lpbk	I	Debug port for loopback control, with priority over the *enable* signal. '0' – Normal port selected for output. '1' – Loopback port selected for output.	crypto4otn.dbg_lpbk_tx (*) crypto4otn.dbg_lpbk_rx (**)
dbg_byp	I	Debug port for bypass control, with priority over the *enable* signal, used here to inhibit the loopback port selection control. '0' – Loopback port selection enabled. '1' – Loopback port selection disabled (data loopback controlled just by the *dbg_lpbk* debug port).	crypto4otn.dbg_byp_tx (*) crypto4otn.dbg_byp_rx (**)
enable	I	Block enable control. '0' – Block disabled and all output ports de-asserted (at logic '0'). '1' – Block enabled.	control_engine.tx_enable (*) control_engine.rx_enable (**)
data_out[639:0]	O	Padded ODU output data.	crypto_engine(TX processor).data_in (*) flow_control(TX processor).data_in_byp (*) oh_extractor.data_in (**) flow_control(RX processor).data_in_byp (**)
valid_out	O	Output data valid. Validates the ODU output data at the *data_out* bus, as well as the *fs_out*, *mfs_out*, and *fail_out* signals. Active-high.	crypto_engine(TX processor).valid_in (*) flow_control(TX processor).valid_in_byp (*) control_engine.ipy_valid_tx_in (*) oh_extractor.valid_in (**) flow_control(RX processor).valid_in_byp (**) control_engine.ipy_valid_rx_in (**)
fs_out	O	Output data frame start pulse. Single clock period pulse that precedes the start of the ODU output data frame at the *data_out* bus. Active-high.	crypto_engine(TX processor).fs_in (*) flow_control(TX processor).fs_in_byp (*) control_engine.ipy_fs_tx_in (*) oh_extractor.fs_in (**) flow_control(RX processor).fs_in_byp (**) control_engine.ipy_fs_rx_in (**)
mfs_out	O	Output data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU output data multi-frame at the *data_out* bus. Active-high.	crypto_engine(TX processor).mfs_in (*) flow_control(TX processor).mfs_in_byp (*) control_engine.ipy_mfs_tx_in (*) oh_extractor.mfs_in (**) flow_control(RX processor).mfs_in_byp (**) control_engine.ipy_mfs_rx_in (**)

Signal	I/O	Description	From/To
fail_out	O	Fail signal. Active-high.	crypto_engine(TX processor).fail_in (*) flow_control(TX processor).fail_in_byp (*) control_engine.ipy_tsf_tx (*) oh_extractor.fail_in (**) flow_control(RX processor).fail_in_byp (**) control_engine.ipy_tsf_rx (**)
(*) When used in the TX processor (**) When used in the RX processor			

Table 4.6 – Loopback Mux sub-block interface signals.

The multiplexer switch has two input ports (normal and loopback) and one output port.

The *dbg_lpbk*, *dbg_byp*, and *loopback_blocking* input ports have priority over the *enable* control signal.

When the *dbg_lpbk* input is asserted, the loopback input port is selected and registered to the output. Otherwise, if either the *loopback_blocking* or the *dbg_byp* inputs are asserted, the normal input port is selected and registered to the output, ensuring the correct data port selection for the TX/RX processor bypass or opposite-side-loopback conditions.

When these ports are de-asserted, the loopback input controls the selection of the loopback or normal input ports when the block is enabled.

4.3.5 DATA PATH FLOW CONTROL SUB-BLOCK

This sub-block stores a small amount of data and incorporates a selector multiplexer switch to handle data path flow control and bypass functionalities.

Figure 4.6 shows its external input and output interface, and Table 4.7 brings the signal names, interface directions (input/output), descriptions, and the corresponding sources and destinations (from/to).

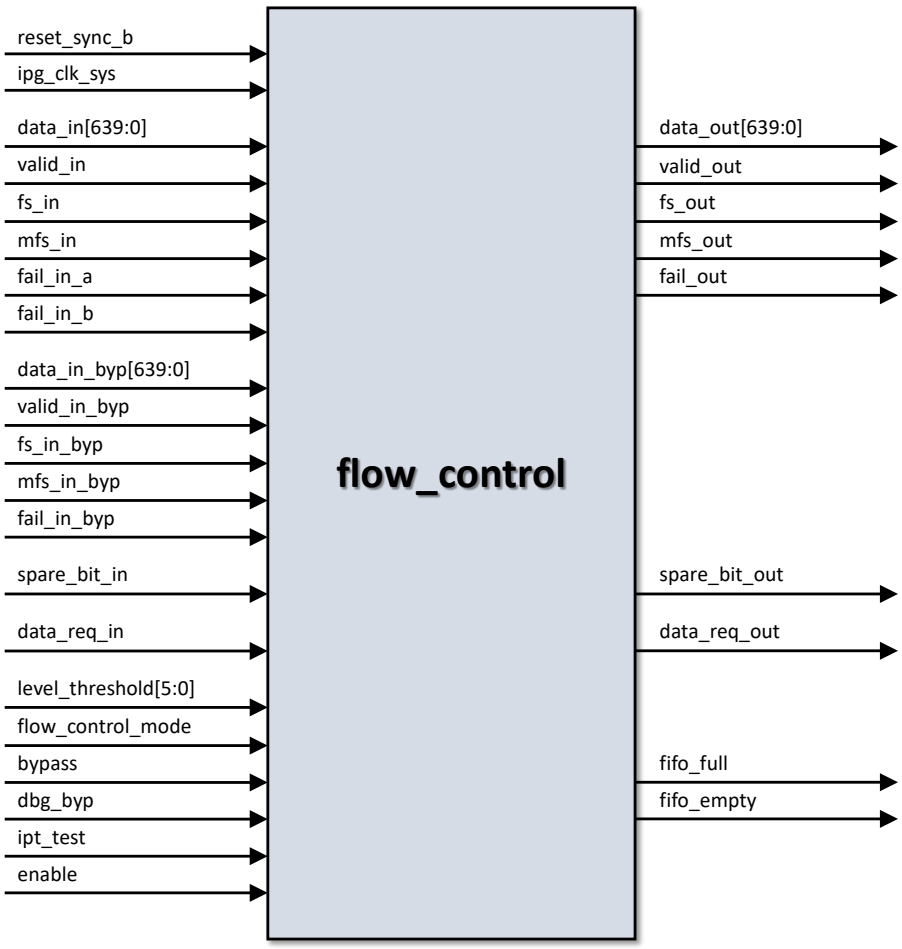


Figure 4.6 – Data Path Flow Control sub-block interface.

Signal	I/O	Description	From/To
reset_sync_b	I	Synchronous reset signal generated in the Control Engine sub-block. Active-low.	control_engine.tx_reset_sync_b (*) control_engine.rx_reset_sync_b (**)
ipg_clk_sys	I	Green line interface clock at 180 MHz with duty cycle of 50%. Used for internal global logic.	crypto4otn.ipg_clk_sys_tx (*) crypto4otn.ipg_clk_sys_rx (**)

Signal	I/O	Description	From/To
data_in[639:0]	I	Padded ODU input data.	repl_sig_gen.data_out (*) auth_buffer.data_out (**)
valid_in	I	Input data valid. Validates the ODU input data at the *ipy_data_in* bus, as well as the *fs_in*, *mfs_in*, *fail_in_a*, *fail_in_b*, and *spare_bit_in* signals. Active-high.	repl_sig_gen.valid_out (*) auth_buffer.valid_out (**)
fs_in	I	Input data frame start pulse. Single clock period pulse that precedes the start of the ODU input data frame at the *ipy_data_in* bus. Active-high.	repl_sig_gen.fs_out (*) auth_buffer.fs_out (**)
mfs_in	I	Input data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU input data multi-frame at the *ipy_data_in* bus. Active-high.	repl_sig_gen.mfs_out (*) auth_buffer.mfs_out (**)
fail_in_a	I	Fail signal. Active-high.	repl_sig_gen.fail_out (*) auth_buffer.fail_out (**)
fail_in_b	I	Fail signal. Active-high.	control_engine.ipy_ssf_tx (*) control_engine.ipy_ssf_rx (**)
data_in_byp[639:0]	I	Padded ODU input data (bypass).	loopback_mux(TX processor).data_out (*) loopback_mux(RX processor).data_out (**)
valid_in_byp	I	Input data valid (bypass). Validates the ODU input data at the *ipy_data_in_byp* bus, as well as the *fs_in_byp*, *mfs_in_byp*, and *fail_in_byp* signals. Active-high.	loopback_mux(TX processor).valid_out (*) loopback_mux(RX processor).valid_out (**)
fs_in_byp	I	Input data frame start pulse (bypass). Single clock period pulse that precedes the start of the ODU input data frame at the *ipy_data_in_byp* bus. Active-high.	loopback_mux(TX processor).fs_out (*) loopback_mux(RX processor).fs_out (**)
mfs_in_byp	I	Input data multi-frame start pulse (bypass). Single clock period pulse that precedes the start of the ODU input data multi-frame at the *ipy_data_in_byp* bus. Active-high.	loopback_mux(TX processor).mfs_out (*) loopback_mux(RX processor).mfs_out (**)
fail_in_byp	I	Fail signal (bypass). Active-high.	loopback_mux(TX processor).fail_out (*) loopback_mux(RX processor).fail_out (**)
spare_bit_in	I	Spare signal input.	Tied to '0' (*) auth_buffer.auth_tag_matching_retimed (**)
data_req_in	I	Data request. Active-high.	crypto4otn.ipy_req_tx_in (*) crypto4otn.ipy_req_rx_in (**)

Signal	I/O	Description	From/To
level_threshold[5:0]	I	FIFO level threshold value.	control_engine.tx_level_thr (*) control_engine.rx_level_thr (**)
flow_control_mode	I	Flow control mode selection. '0' – Push mode. '1' – Pull mode.	control_engine.tx_flow_ctrl_mode (*) control_engine.rx_flow_ctrl_mode (**)
bypass	I	Bypass port selection control. Active-high.	control_engine.tx_bypass (*) control_engine.rx_bypass (**)
dbg_byp	I	Debug port for bypass control, with priority over the *enable* signal. Active-high.	crypto4otn.dbg_byp_tx (*) crypto4otn.dbg_byp_rx (**)
ipt_test	I	Test mode enable signal (for DFT purposes). Active-high.	crypto4otn
enable	I	Block enable control. '0' – Block disabled and all output ports de-asserted (at logic '0'). '1' – Block enabled.	control_engine.tx_enable (*) control_engine.rx_enable (**)
data_out[639:0]	O	Padded ODU output data.	crypto4otn.ipy_data_tx_out (*) loopback_mux(RX processor).data_in_lpbk (*) crypto4otn.ipy_data_rx_out (**) loopback_mux(TX processor).data_in_lpbk (**)
valid_out	O	Output data valid. Validates the ODU output data at the *ipy_data_out* bus, as well as the *fs_out*, *mfs_out*, *fail_out*, and *spare_bit_out* signals. Active-high.	crypto4otn.ipy_valid_tx_out (*) loopback_mux(RX processor).valid_in_lpbk (*) crypto4otn.ipy_valid_rx_out (**) loopback_mux(TX processor).valid_in_lpbk (**)
fs_out	O	Output data frame start pulse. Single clock period pulse that precedes the start of the ODU output data frame at the *ipy_data_out* bus. Active-high.	crypto4otn.ipy_fs_tx_out (*) loopback_mux(RX processor).fs_in_lpbk (*) crypto4otn.ipy_fs_rx_out (**) loopback_mux(TX processor).fs_in_lpbk (**)
mfs_out	O	Output data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU output data multi-frame at the *ipy_data_out* bus. Active-high.	crypto4otn.ipy_mfs_tx_out (*) loopback_mux(RX processor).mfs_in_lpbk (*) crypto4otn.ipy_mfs_rx_out (**) loopback_mux(TX processor).mfs_in_lpbk (**)
fail_out	O	Fail signal. Active-high.	crypto4otn.ipy_ssf_tx (*) loopback_mux(RX processor).fail_in_lpbk (*) crypto4otn.ipy_ssf_rx (**) loopback_mux(TX processor).fail_in_lpbk (**)
spare_bit_out	O	Spare signal output.	Not connected (*) crypto4otn.auth_tag_matching (**)

Signal	I/O	Description	From/To
data_req_out	O	Data request. Active-high.	crypto4otn.ipy_req_tx_out (*) crypto4otn.ipy_req_rx_out (**)
fifo_full	O	FIFO full indication. Active-high.	control_engine.tx_flow_ctrl_fifo_full (*) control_engine.rx_flow_ctrl_fifo_full (**)
fifo_empty	O	FIFO empty indication. Active-high.	control_engine.tx_flow_ctrl_fifo_empty (*) control_engine.rx_flow_ctrl_fifo_empty (**)
(*) When used in the TX processor (**) When used in the RX processor			

Table 4.7 – Data Path Flow Control sub-block interface signals.

The Data Path Flow Control sub-block has two input ports (normal and bypass) and one output port.

Figure 4.7 shows its simplified functional architecture diagram.

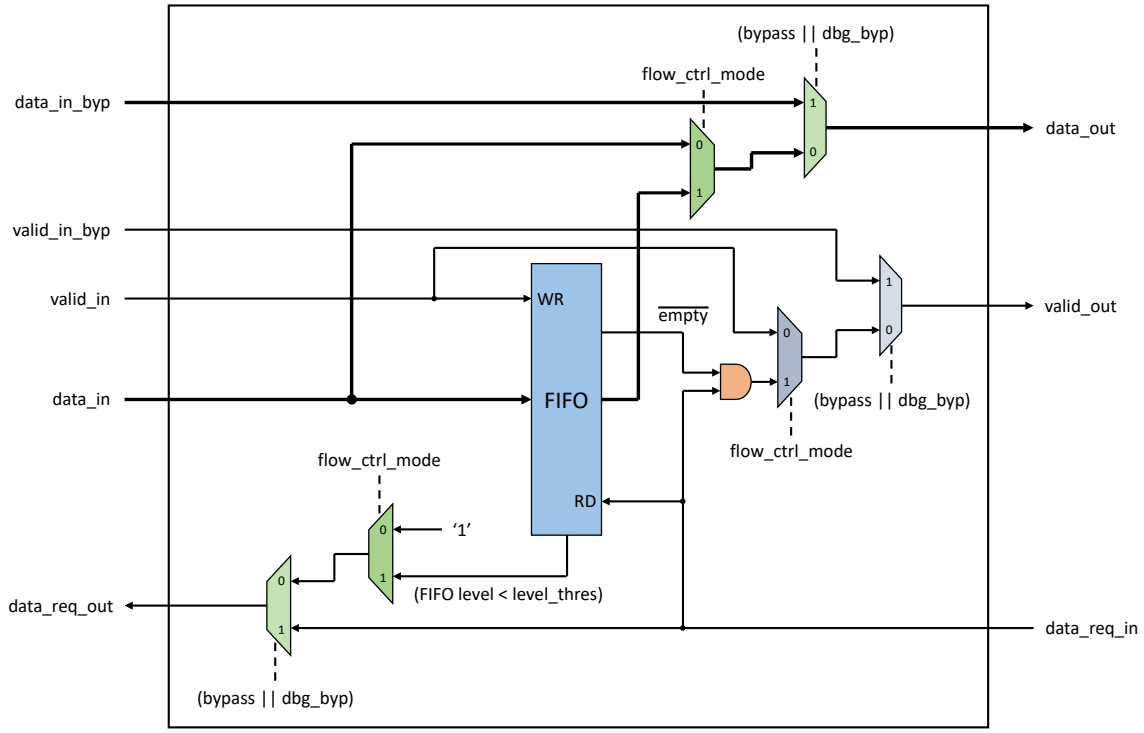


Figure 4.7 – Data Path Flow Control sub-block simplified functional architecture diagram.

When either the *bypass* or the *dbg_byp* inputs are asserted, the bypass input port is selected and registered to the output. Otherwise, the normal input port is used in the flow control pipeline. In bypass mode, the *data_req_in* input is also registered to the *data_req_out* output. The *dbg_byp* input has priority over the *enable* control.

When operating in "pull" mode (*flow_control_mode* input asserted), an internal FIFO stores data coming from the normal (not the bypass) input port. In this mode, the *data_req_out* is asserted (to request data from the preceding block) until the internal FIFO level reaches the value specified in the *level_threshold* input. The FIFO read enable is controlled by the *data_req_in* input.

Conversely, when operating in "push" mode (*flow_control_mode* input de-asserted), the *data_req_out* port is asserted and the input data are directly registered to the output port, being pushed by the preceding block.

The *level_threshold* value is set by the Crypto4OTN block control register (*blk_ctr*). The lower bound is four, but a default setting of 32 (four times the register value) is used to provide a margin for data stored in the preceding sub-blocks (pipeline flushing/filling), especially in the AES-GCM sub-block (instantiated in the OPU Cryptography Engine).

The data valid output signal (*valid_out*) logic level depends on the flow control operation mode. In "push" mode, it is a registered copy of *valid_in*, while in "pull" mode it is asserted when the FIFO is not empty and the data request input signal (*data_req_in*) is asserted.

Only in "pull" mode, after a reset event a pre-filling mechanism ensures that the FIFO is filled up to the *level_threshold* value before output is released. When the flow control mode is changed from "push" to "pull", this pre-filling mechanism is restarted.

Two spare signals (*spare_bit_input* and *spare_bit_output*) are also implemented to be used in the RX processor to synchronize the Crypto4OTN top block *auth_tag_matching* output signal. In bypass mode, the *spare_bit_output* signal is tied to logic level '0'.

The input fail indication signals (*fail_in_a* and *fail_in_b*) are combined by a logic OR operation, and the result is registered to the respective output (*fail_out*) with a delay equivalent to the Data Path Flow Control processing latency.

4.3.6 OPU CRYPTOGRAPHY ENGINE SUB-BLOCK

The OPU Cryptography Engine sub-block implements the core functionalities for OPU data encryption, decryption, and authentication. It receives and transmits padded ODU frames and handles OPU drop/add synchronization. Depending on the operation mode, this sub-block outputs encrypted or clear data, as well as a 128-bit authentication TAG. It is instantiated in both TX (for encryption) and RX (for decryption) processor lineups.

Figure 4.8 shows its external input and output interface, and Table 4.8 brings the signal names, interface directions (input/output), descriptions, and the corresponding sources and destinations (from/to).

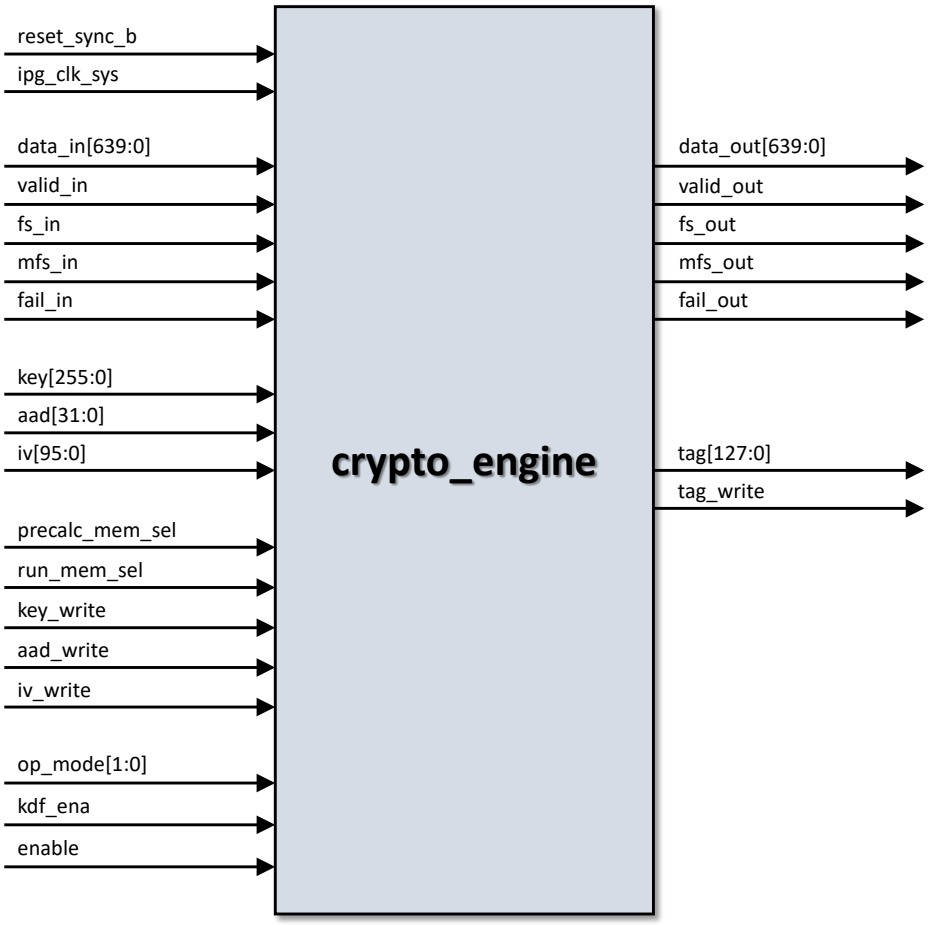


Figure 4.8 – OPU Cryptography Engine sub-block interface.

Signal	I/O	Description	From/To
reset_sync_b	I	Synchronous reset signal generated in the Control Engine sub-block. Active-low.	control_engine.tx_reset_sync_b (*) control_engine.rx_reset_sync_b (**)

Signal	I/O	Description	From/To
ipg_clk_sys	I	Green line interface clock at 180 MHz with duty cycle of 50%. Used for internal global logic.	crypto4otn.ipg_clk_sys_tx (*) crypto4otn.ipg_clk_sys_rx (**)
data_in[639:0]	I	Padded ODU input data.	loopback_mux(TX processor).data_out (*) oh_extractor.data_out (**)
valid_in	I	Input data valid. Validates the ODU input data at the *data_in* bus, as well as the *fs_in*, *mfs_in*, and *fail_in* signals. Active-high.	loopback_mux(TX processor).valid_out (*) oh_extractor.valid_out (**)
fs_in	I	Input data frame start pulse. Single clock period pulse that precedes the start of the ODU input data frame at the *data_in* bus. Active-high.	loopback_mux(TX processor).fs_out (*) oh_extractor.fs_out (**)
mfs_in	I	Input data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU input data multi-frame at the *data_in* bus. Active-high.	loopback_mux(TX processor).mfs_out (*) oh_extractor.mfs_out (**)
fail_in	I	Fail signal. Active-high.	loopback_mux(TX processor).fail_out (*) oh_extractor.fail_out (**)
key[255:0]	I	256-bit crypto session key.	control_engine.tx_key (*) control_engine.rx_key (**)
aad[31:0]	I	Additional authenticated data.	control_engine.tx_aad (*) oh_extractor.aad (**)
iv[95:0]	I	Initialization vector made up of a fixed field (CSKS and CSID) and an invocation field (CBID and CPID).	control_engine.tx_iv (*) oh_extractor.iv (**)
precalc_mem_sel	I	Pre-calculation memory selection.	control_engine.tx_precalc_mem_sel (*) control_engine.rx_precalc_mem_sel (**)
run_mem_sel	I	Running memory selection, from where pre-calculated data are read by the AES-GCM algorithm.	control_engine.tx_run_mem_sel (*) control_engine.rx_run_mem_sel (**)
key_write	I	Key write enable and pre-calculation trigger pulse.	control_engine.tx_key_write (*) control_engine.rx_key_write (**)
aad_write	I	AAD write enable and pre-calculation trigger pulse.	control_engine.tx_aad_iv_write (*) oh_extractor.aad_write (**)
iv_write	I	IV write enable and pre-calculation trigger pulse.	control_engine.tx_aad_iv_write (*) oh_extractor.iv_write (**)
op_mode[1:0]	I	Crypto session operation mode. ‘00’ – Authenticated encryption. ‘01’ – Authentication-only. ‘10’ – Encryption-only.	control_engine.tx_op_mode (*) control_engine.rx_op_mode (**)

Signal	I/O	Description	From/To
kdf_ena	I	Key derivation function enable control. Active-high. When enabled, key values are scrambled before being delivered to the AES-GCM sub-block.	control_engine.tx_kdf_ena (*) control_engine.rx_kdf_ena (**)
enable	I	Block enable control. '0' – Block disabled and all output ports de-asserted (at logic '0'). '1' – Block enabled.	control_engine.tx_enable (*) control_engine.rx_enable (**)
data_out[639:0]	O	Padded ODU output data.	oh_inserter.data_in (*) auth_buffer.data_in (**)
valid_out	O	Output data valid. Validates the ODU output data at the *data_out* bus, as well as the *fs_out*, *mfs_out*, and *fail_out* signals. Active-high.	oh_inserter.valid_in (*) auth_buffer.valid_in (**)
fs_out	O	Output data frame start pulse. Single clock period pulse that precedes the start of the ODU output data frame at the *data_out* bus. Active-high.	oh_inserter.fs_in (*) auth_buffer.fs_in (**)
mfs_out	O	Output data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU output data multi-frame at the *data_out* bus. Active-high.	oh_inserter.mfs_in (*) auth_buffer.mfs_in (**)
fail_out	O	Fail signal. Active-high.	oh_inserter.fail_in (*) auth_buffer.fail_in (**)
tag[127:0]	O	Authentication TAG generated for the current crypto packet.	control_engine.tx_calc_tag (*) oh_inserter.tag (*) control_engine.rx_calc_tag (**)
tag_write	O	Authentication TAG write enable pulse. TAG output bus value must be captured and stored on the rising edge of *tag_write*.	control_engine.tx_calc_tag_write (*) oh_inserter.tag_write (*) control_engine.rx_calc_tag_write (**)
(*) When used in the TX processor (**) When used in the RX processor			

Table 4.8 – OPU Cryptography Engine sub-block interface signals.

Figure 4.9 shows its high-level functional architecture diagram.

Incoming OPU data are dropped and sent to the AES-GCM sub-block for encryption/authentication. At the same time, the OTN frame overhead is buffered until data being output by the AES-GCM are ready to be added to the overall frame.

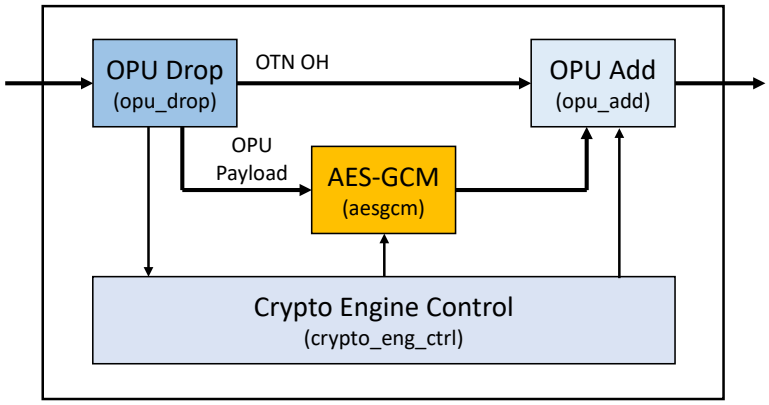


Figure 4.9 – OPU Cryptography Engine high-level functional architecture diagram.

Depending on the state of the *kdf_ena* input port, a key derivation function is used to scramble the key value before it is delivered to the AES-GCM sub-block. A simple byte transposition operation derives a new 256-bit sequence for the encryption and authentication algorithm, thus providing an additional security layer.

The input fail indication signal (*fail_in*) is registered to the respective output (*fail_out*) with a delay equivalent to the OPU Cryptography Engine processing latency.

4.3.6.1 OPU DROP SUB-BLOCK

The OPU Drop sub-block needs to handle data segregation in the 640-bit bus. For instance, right after the frame start pulse, the first bus cycle brings 14 bytes of OTN overhead (to be forwarded) and 66 bytes of OPU data (to be dropped). In the same way, the 48th bus cycle (the last one of a frame line) contains 64 bytes of OPU data (to be dropped) and 16 padding bytes (to be discarded). OPU data being dropped to the AES-GCM sub-block need to be regrouped in 640-bit words, and, therefore, if the dropped data bus is incomplete in a given clock cycle, more data must be appended in the next one.

Table 4.9 brings the signal names, interface directions (input/output), descriptions, and the corresponding sources and destinations (from/to).

Signal	I/O	Description	From/To
reset_sync_b	I	Synchronous reset signal generated in the Control Engine sub-block. Active-low.	crypto_engine
ipg_clk_sys	I	Green line interface clock at 180 MHz with duty cycle of 50%. Used for internal global logic.	crypto_engine
data_in[639:0]	I	Padded ODU input data.	crypto_engine

Signal	I/O	Description	From/To
valid_in	I	Input data valid. Validates the ODU input data at the *data_in* bus, as well as the *fs_in* and *mfs_in* signals. Active-high.	crypto_engine
fs_in	I	Input data frame start pulse. Single clock period pulse that precedes the start of the ODU input data frame at the *data_in* bus. Active-high.	crypto_engine
mfs_in	I	Input data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU input data multi-frame at the *data_in* bus. Active-high.	crypto_engine
enable	I	Block enable control. '0' – Block disabled. '1' – Block enabled.	crypto_engine
opu_data[639:0]	O	OPU output data.	aesgcm.data_in
opu_valid	O	Output data valid. Validates the OPU output data at the *opu_data* bus, as well as the *fs_out* and *mfs_out* signals. Active-high.	aesgcm.valid_in
fs_out	O	Output data frame start pulse. Single clock period pulse that precedes the start of the ODU output data frame. Active-high.	opu_add.fs_in
mfs_out	O	Output data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU output data multi-frame. Active-high.	opu_add.mfs_in
oh_data[111:0]	O	OTU/ODU overhead data output.	opu_add.oh_data_in
oh_valid	O	Overhead data valid. Validates the overhead data at the *oh_data* bus. Active-high.	opu_add.oh_valid_in

Table 4.9 – OPU Drop sub-block interface signals.

The internal logic for OPU data drop is synchronized by the incoming padded ODU multi-frame start (*mfs_in*) and frame start (*fs_in*) signal pulses. They are asserted one clock cycle before the start of each multi-frame and each frame, respectively. Incoming OPU data are dropped to the AES-GCM sub-block, while the OTN overhead is forwarded to the OPU Add sub-block.

Figure 4.10 illustrates how data are grouped along with the 48 bus cycles in each of the four lines of two consecutive padded ODU frames. The 14 bytes of OTN overhead in the beginning of each line are forwarded to the *oh_data* output. In the end of each line, 16 padding bytes are discarded. Each block in the figure represents a 640-bit input bus cycle, and numbers correspond to how many bytes are grouped to fill out a transmission bus that is dropped to the *opu_data* output.

Numbers enclosed by dashed lines represent bytes not used in that current cycle (remaining for the next one). It can be noticed that in the 48th cycle of the last line of the second consecutive frame (even frame) the dropped data bus (in the *opu_data* output) is complete, establishing a data grouping distribution periodicity of two (odd/even) frames or 384 clock cycles. In case of an OTN frame slip, the multi-frame start (*mfs_in*) signal ensures realignment.

The signal *opu_valid* is de-asserted (logic '0') during the bus cycles enclosed by the thick red dashed lines, since the 640-bit dropped data bus is not complete in these clock cycles. Considering the periodicity of two frames (384 cycles), de-assertion occurs at cycles 1, 144, and 241.

Both *opu_valid* and *oh_valid* are de-asserted when the enable input is de-asserted.

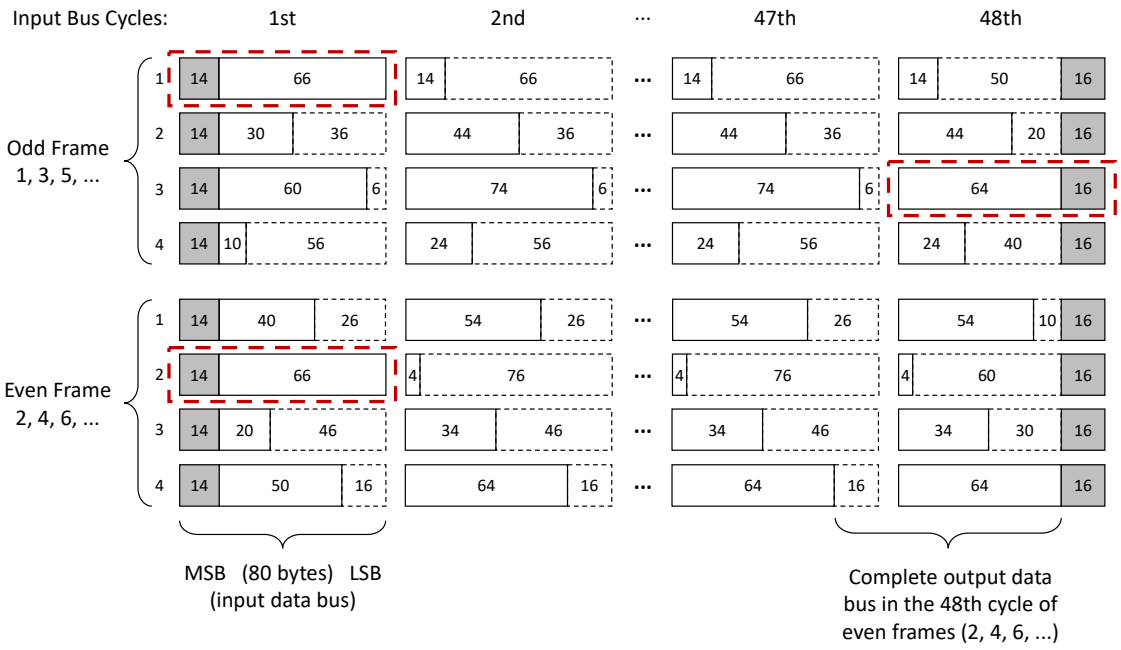


Figure 4.10 – OPU Drop data grouping in the 640-bit output bus.

Figure 4.11 and Figure 4.12 show the OPU Drop input and output data timing diagrams.

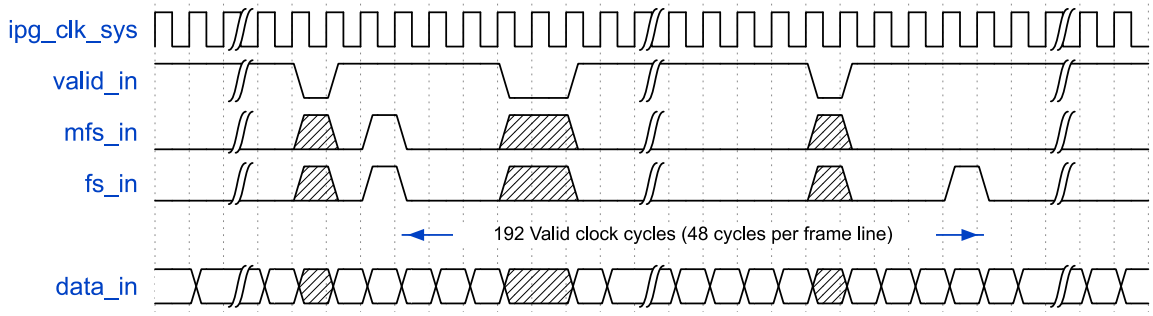


Figure 4.11 – OPU Drop input data timing diagram.

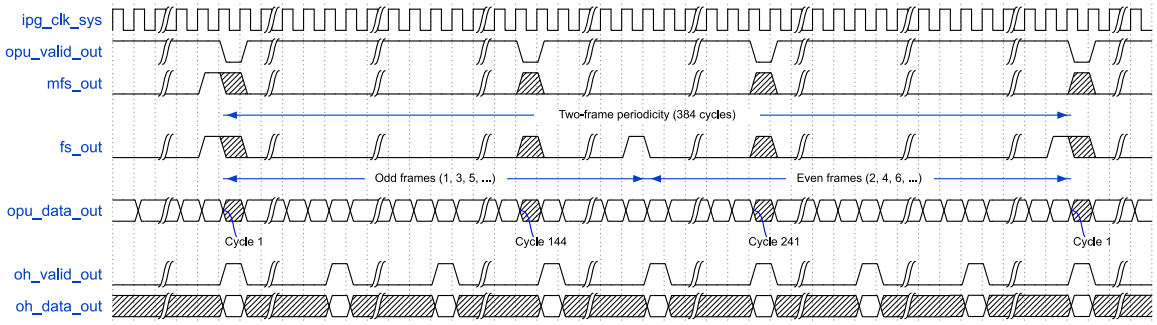


Figure 4.12 – OPU Drop output data timing diagram.

4.3.6.2 OPU ADD SUB-BLOCK

The OPU Add sub-block handles data aggregation.

In the beginning of each ODU frame line, 14 overhead bytes coming from the OPU Drop are buffered, while the remaining OPU bytes are processed by the AES-GCM sub-block. Then, a mechanism similar to the one used in the OPU Drop handles byte regrouping between consecutive input bus cycles of OPU data. In the end of each line, 16 zero padding bytes must be appended to complete the padded ODU frame structure.

Table 4.10 brings the signal names, interface directions (input/output), descriptions, and the corresponding sources and destinations (from/to).

Signal	I/O	Description	From/To
reset_sync_b	I	Synchronous reset signal generated in the Control Engine sub-block. Active-low.	crypto_engine
ipg_clk_sys	I	Green line interface clock at 180 MHz with duty cycle of 50%. Used for internal global logic.	crypto_engine
opu_data[639:0]	I	OPU input data.	aesgcm.data_out
opu_valid	I	OPU data valid. Validates the OPU input data at the *opu_data* bus, as well as the *fs_in* and *mfs_in* signals. Active-high.	aesgcm.valid_out

Signal	I/O	Description	From/To
fs_in	I	Input data frame start pulse. Single clock period pulse that precedes the start of an ODU frame. Active-high.	crypto_eng_ctrl.fs_retimed
mfs_in	I	Input data multi-frame start pulse. Single clock period pulse that precedes the start of an ODU multi-frame. Active-high.	crypto_eng_ctrl.mfs_retimed
oh_data[111:0]	I	OTU/ODU overhead data input.	opu_drop.oh_data_out
oh_valid	I	Overhead data valid. Validates the overhead data at the *oh_data* bus. Active-high.	opu_drop.oh_valid_out
enable	I	Block enable control. '0' – Block disabled. '1' – Block enabled.	crypto_engine
data_out[639:0]	O	Padded ODU output data.	crypto_engine
valid_out	O	Output data valid. Validates the ODU output data at the *data_out* bus, as well as the *fs_out* and *mfs_out* signals. Active-high.	crypto_engine
fs_out	O	Output data frame start pulse. Single clock period pulse that precedes the start of the ODU output data frame at the *data_out* bus. Active-high.	crypto_engine
mfs_out	O	Output data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU output data multi-frame at the *data_out* bus. Active-high.	crypto_engine

Table 4.10 – OPU Add sub-block interface signals.

The internal logic for OPU Add is also synchronized by the multi-frame start (*mfs_in*) and frame start (*fs_in*) signal pulses. These signals are retimed by the Crypto Engine Control to match the same processing delay of the AES-GCM.

Data grouping distribution has the same periodicity of 2 (odd/even) frames or 384 clock cycles. In case of an OTN frame slip, the multi-frame start (*mfs_in*) signal ensures realignment.

An additional OPU data storage of a couple of cycles is needed to compensate for the gaps in the *opu_valid* signal as data are not provided by the AES-GCM at cycles 1, 144, and 241.

Figure 4.13 and Figure 4.14 show the OPU Add input and output data timing diagrams.

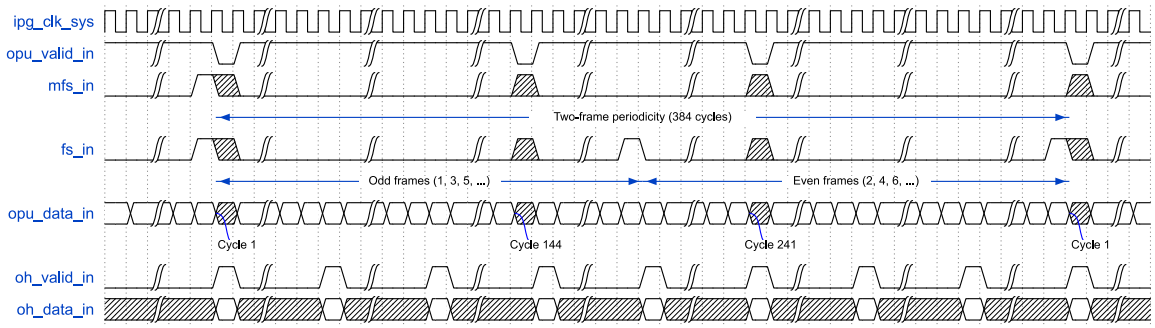


Figure 4.13 – OPU Add input data timing diagram.

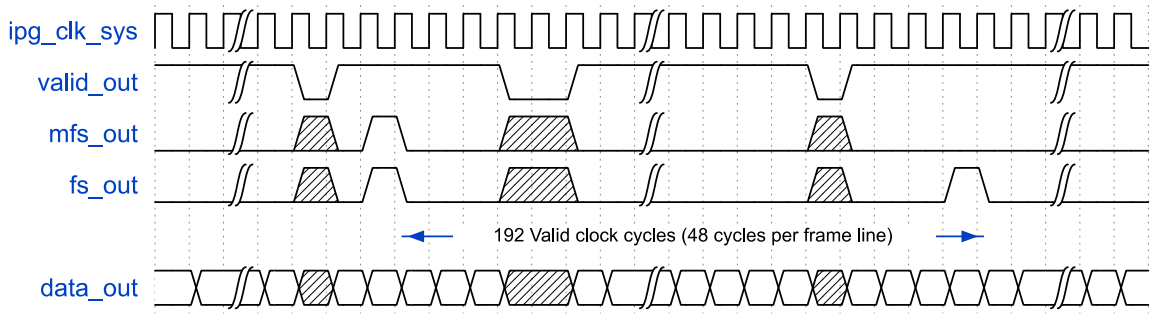


Figure 4.14 – OPU Add output data timing diagram.

4.3.6.3 AES-GCM SUB-BLOCK

The AES-GCM sub-block performs all the mathematical operations for encryption, decryption, and authentication of the OPU data. Every 640-bit word in the data path is sliced in five segments, which are processed by five instances of a 128-bit AES block cipher running in parallel, as previously shown in Figure 2.20. The block receives data to be encrypted or decrypted, a 256-bit cipher key, a 96-bit IV sequence, and the 32-bit AAD. It then outputs encrypted/decrypted data and a 128-bit authentication TAG.

Table 4.11 brings the signal names, interface directions (input/output), descriptions, and the corresponding sources and destinations (from/to).

Signal	I/O	Description	From/To
reset_sync_b	I	Synchronous reset signal generated in the Control Engine sub-block. Active-low.	crypto_engine
ipg_clk_sys	I	Green line interface clock at 180 MHz with duty cycle of 50%. Used for internal global logic.	crypto_engine
data_in[639:0]	I	Clear message input data.	opu_drop.opu_data

Signal	I/O	Description	From/To
data_valid_in	I	Input data valid. Validates the clear message input data at the *data_in* bus. Active-high.	opu_drop.opu_valid
key[255:0]	I	256-bit crypto session key.	crypto_engine
aad[31:0]	I	Additional authenticated data.	crypto_engine
iv[95:0]	I	Initialization vector made up of a fixed field (CSKS and CSID) and an invocation field (CBID and CPID).	crypto_engine
precalc_mem_sel	I	Pre-calculation memory selection.	crypto_eng_ctrl.precalc_mem_sel_retimed
run_mem_sel	I	Running memory selection, from where pre-calculated data are read by the AES-GCM algorithm.	crypto_eng_ctrl.run_mem_sel_retimed
precalc_key_pls	I	Key pre-calculation trigger. Key input bus value is captured and internal calculations are initiated on the rising edge of *precalc_iv_pls*.	crypto_eng_ctrl.key_write_retimed
precalc_aad_pls	I	AAD pre-calculation trigger. AAD input bus value is captured and internal calculations are initiated on the rising edge of *precalc_aad_pls*.	crypto_engine.aad_write
precalc_iv_pls	I	IV pre-calculation trigger. IV input bus value is captured and internal calculations are initiated on the rising edge of *precalc_iv_pls*.	crypto_engine.iv_write
msg_size_static[15:0]	I	Message size in multiples of 640 bits. It is represented by a 16-bit value.	Tied to a fixed value of 0x02fa, corresponding to a message size of 762 × 640 = 487,680 bits
data_direction	I	Specifies whether the AES-GCM sub-block is used in the TX or RX processor. ‘0’ – Used in the TX processor. ‘1’ – Used in the RX processor.	Tied to a fixed value
msg_sync_pls	I	Input clear message synchronization pulse. It precedes the start of a clear message block to synchronize internal message size counters.	crypto_eng_ctrl.crypto_packet_start
enable	I	Block enable control. ‘0’ – Block disabled. ‘1’ – Block enabled.	crypto_engine
op_mode[1:0]	I	Crypto session operation mode. ‘00’ – Authenticated encryption. ‘01’ – Authentication-only. ‘10’ – Encryption-only.	crypto_engine
data_out[639:0]	O	Encrypted output data.	opu_add.opu_data
data_valid_out	O	Output data valid. Validates the encrypted output data at the *data_out* bus. Active-high.	opu_add.opu_valid

Signal	I/O	Description	From/To
tag_out[127:0]	O	Authentication TAG generated for the current crypto packet.	crypto_engine.tag
tag_valid_out	O	Authentication TAG valid pulse. TAG output bus value must be captured and stored on the rising edge of *tag_valid_out*.	crypto_engine.tag_write
precalc_ready	O	Pre-calculation ready indication. Not used in this application.	Not connected

Table 4.11 – AES-GCM sub-block interface signals.

The AES-GCM *message_size* input port is tied to the fixed value 0x02fa, corresponding to a message size of 762 × 640 = 487,680 bits and making up four OPUs — the payload of one crypto packet.

When the OPU Cryptography Engine sub-block is instantiated in the TX processor lineup, the AES-GCM *data_direction* input port is tied to '0' to indicate that an encryption operation is expected. Conversely, in the RX processor it is tied to '1' to perform data decryption.

Due to the inherent latency of the necessary calculations, a dual-memory bank strategy was used to keep data processing at speed. They store cryptographic parameter values (key, AAD, and IV) to be used in two phases — pre-calculation and encryption/decryption. In this way, while one crypto packet is being processed using the parameters stored in the "running memory", the necessary pre-calculations for the next one are being concurrently executed using the ones stored in the "pre-calc memory". These memory banks are cyclically switched over at every crypto packet.

The AES-GCM is further internally divided into three sub-blocks — the first (*aesgcm_gctr*) performs the encryption and decryption processes, the second (*aesgcm_ghash*) calculates the authentication TAG, and the third (*aesgcm_ctrl*) controls and synchronizes data between the other two sub-blocks.

Table 4.12 shows the block data path latencies. The 16-cycle latency for the encryption/decryption process is a consequence of the 14-stage pipelined implementation of the 14-round AES algorithm, with two additional cycles of data registering in the block overall architecture.

Operation Mode	Data Output Latency	TAG Latency
Authenticated Encryption/Decryption	16	26
Authentication-Only	1	10
Encryption/Decryption-Only	16	-
Bypass	1	-

Table 4.12 – AES-GCM sub-block data path latencies.

The author was not directly involved in the RTL design of the AES-GCM sub-block, which was built based on two reference models written in C language, serving both as an architectural guideline and as golden models for functional verification purposes.

4.3.6.4 CRYPTO ENGINE CONTROL SUB-BLOCK

The Crypto Engine Control sub-block synchronizes the data flow at the input of the AES-GCM and OPU Add sub-blocks, in order to delineate the payload of crypto packets and perform ODU frame reconstruction.

Table 4.13 brings the signal names, interface directions (input/output), descriptions, and the corresponding sources and destinations (from/to).

Signal	I/O	Description	From/To
reset_sync_b	I	Synchronous reset signal generated in the Control Engine sub-block. Active-low.	crypto_engine
ipg_clk_sys	I	Green line interface clock at 180 MHz with duty cycle of 50%. Used for internal global logic.	crypto_engine
valid_in	I	Input data valid. Validates the *fs_in* and *mfs_in* signals. Active-high.	opu_drop.opu_valid
fs_in	I	Input data frame start pulse. Single clock period pulse that precedes the start of the ODU data frame. Active-high.	opu_drop.fs_out
mfs_in	I	Input data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU input data multi-frame. Active-high.	opu_drop.mfs_out
precalc_mem_sel	I	Pre-calculation memory selection.	crypto_engine
run_mem_sel	I	Running memory selection, from where pre-calculated data are read by the AES-GCM algorithm.	crypto_engine
key_write	I	Key write enable and pre-calculation trigger pulse.	crypto_engine

Signal	I/O	Description	From/To
op_mode[1:0]	I	Crypto session operation mode. '00' – Authenticated encryption. '01' – Authentication-only. '10' – Encryption-only.	crypto_engine
enable	I	Block enable control. '0' – Block disabled and all output ports de-asserted (at logic '0'). '1' – Block enabled.	crypto_engine
fs_retimed	O	Output data frame start pulse. Single clock period pulse that precedes the start of an ODU frame. Active-high. Corresponds to a delayed version of the *fs_in* input signal.	opu_add.fs_in
mfs_retimed	O	Output data multi-frame start pulse. Single clock period pulse that precedes the start of an ODU multi-frame. Active-high. Corresponds to a delayed version of the *mfs_in* input signal.	opu_add.mfs_in
crypto_packet_start	O	Crypto packet start synchronization pulse. Single clock period pulse that precedes the start of a crypto packet OPU data.	aesgcm.message_sync
precalc_mem_sel_retimed	O	Retimed version of the pre-calculation memory selection input bit, sampled at the rising edge of the *crypto_packet_start* synchronization pulse.	aesgcm.precalc_mem_sel
run_mem_sel_retimed	O	Retimed version of the running memory selection input bit, sampled at the rising edge of the *crypto_packet_start* synchronization pulse.	aesgcm.run_mem_sel
key_write_retimed	O	Retimed version of the key write enable and pre-calculation trigger pulse, sampled at the falling edge of the *crypto_packet_start* synchronization pulse.	aesgcm.precalc_key_pls

Table 4.13 – Crypto Engine Control sub-block interface signals.

Figure 4.15 shows how data flow synchronization is accomplished by asserting the *crypto_packet_start* output in the clock period that follows the detection of the frame start pulse at the *fs_in* input, in the beginning of each crypto packet. OPU data start right after this synchronization pulse, in the second clock cycle after the frame start pulse.

The *precalc_mem_sel_retimed* and *run_mem_sel_retimed* outputs are retimed versions of the *precalc_mem_sel* and *run_mem_sel* inputs, sampled at the rising edge of the *crypto_packet_start* synchronization pulse. In the same way, the *key_write_retimed* output is a retimed version of the *key_write* input, but sampled at the falling edge of the *crypto_packet_start* synchronization pulse.

The *fs_retimed* and *mfs_retimed* outputs are retimed versions of the *fs_in* and *mfs_in* inputs, with a delay equivalent to the AES-GCM sub-block processing latency (one clock cycle for authentication-only or 16 clock cycles for authenticated encryption and encryption-only operation modes).

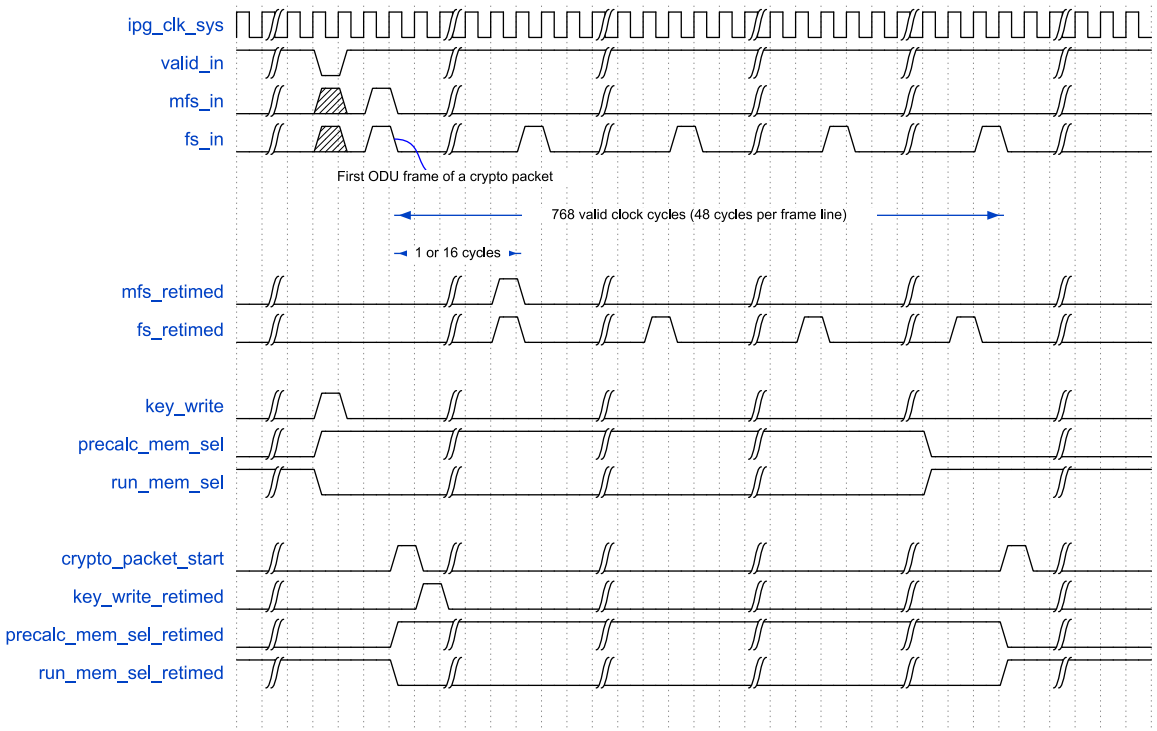


Figure 4.15 – Crypto Engine Control output data timing diagram.

4.3.7 ODU OH INSERTER SUB-BLOCK

This sub-block handles data insertion into the ODU overhead.

Encryption overhead data (16 bytes of TAG, 4 bytes of AAD, and 12 bytes of IV) are inserted into four consecutive ODU overheads, each with eight available RES fields.

Additionally, all these data can be scrambled right before insertion.

Figure 4.16 shows its external input and output interface, and Table 4.14 brings the signal names, interface directions (input/output), descriptions, and the corresponding sources and destinations (from/to).

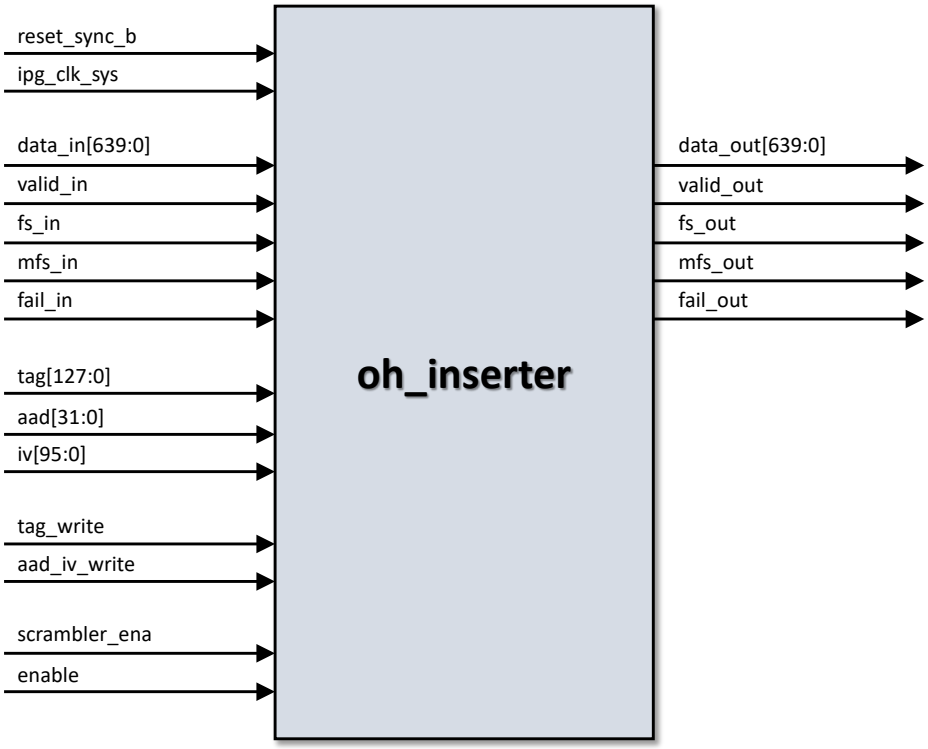


Figure 4.16 – ODU OH Inserter sub-block interface.

Signal	I/O	Description	From/To
reset_sync_b	I	Synchronous reset signal generated in the Control Engine sub-block. Active-low.	control_engine.tx_reset_sync_b
ipg_clk_sys	I	Green line interface clock at 180 MHz with duty cycle of 50%. Used for internal global logic.	crypto4otn.ipg_clk_sys_tx
data_in[639:0]	I	Padded ODU input data.	crypto_engine(TX processor).data_out

Signal	I/O	Description	From/To
valid_in	I	Input data valid. Validates the ODU input data at the *data_in* bus, as well as the *fs_in*, *mfs_in*, and *fail_in* signals. Active-high.	crypto_engine(TX processor).valid_out
fs_in	I	Input data frame start pulse. Single clock period pulse that precedes the start of the ODU input data frame at the *data_in* bus. Active-high.	crypto_engine(TX processor).fs_out
mfs_in	I	Input data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU input data multi-frame at the *data_in* bus. Active-high.	crypto_engine(TX processor).mfs_out
fail_in	I	Fail signal. Active-high.	crypto_engine(TX processor).fail_out
tag[127:0]	I	Authentication TAG to be inserted in the current crypto packet (calculated for the previous one).	crypto_engine(TX processor)
aad[31:0]	I	Additional authenticated data.	control_engine.tx_aad
iv[95:0]	I	Initialization vector made up of a fixed field (CSKS and CSID) and an invocation field (CBID and CPID).	control_engine.tx_iv
tag_write	I	TAG write enable pulse. TAG input bus value is captured and stored on the rising edge of *tag_write*.	crypto_engine(TX processor)
aad_iv_write	I	AAD and IV write enable pulse. AAD and IV input bus values are captured and stored on the rising edge of *aad_iv_write*.	control_engine.tx_aad_iv_write
scrambler_ena	I	Scrambler enable control. Active-high.	control_engine.tx_oh_scrambler_ena
enable	I	Block enable control. '0' – Block disabled and all output ports de-asserted (at logic '0'). '1' – Block enabled.	control_engine.tx_enable
data_out[639:0]	O	Padded ODU output data.	repl_sig_gen.data_in
valid_out	O	Output data valid. Validates the ODU output data at the *data_out* bus, as well as the *fs_out*, *mfs_out*, and *fail_out* signals. Active-high.	repl_sig_gen.valid_in
fs_out	O	Output data frame start pulse. Single clock period pulse that precedes the start of the ODU output data frame at the *data_out* bus. Active-high.	repl_sig_gen.fs_in
mfs_out	O	Output data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU output data multi-frame at the *data_out* bus. Active-high.	repl_sig_gen.mfs_in
fail_out	O	Fail signal. Active-high.	repl_sig_gen.fail_in

Table 4.14 – ODU OH Inserter sub-block interface signals.

The encryption overhead bytes (TAG, AAD, and IV) are inserted in the eight RES fields of the ODU overhead, with location (row, column) shown in Figure 3.5. Since four consecutive frames are needed, a clock cycle counter synchronized by the multi-frame start signal (*mfs_in*) is used to determine the insertion location distribution.

In the beginning of each frame line, the 14 most significant bytes of the 640-bit bus correspond to the OTN overhead fields. Data are replaced (inserted) during clock cycles number 49 and 145 in a given ODU frame.

Replacement data (encryption overhead) are captured from the respective input port and internally stored to be used at the time of insertion. All other incoming data are forwarded to the output port.

The input fail indication signal (*fail_in*) is registered to the respective output (*fail_out*) with a delay equivalent to the ODU OH Inserter processing latency.

4.3.7.1 SCRAMBLING FUNCTION

As an additional security feature, an optional scrambling of the encryption overhead data at the time of insertion was also implemented. Depending on the state of the *scrambler_ena* input port, all inserted data can be scrambled by the following mechanism.

A 16-stage Galois linear feedback shift register (LFSR) with feedback polynomial $x^{16} + x^{15} + x^{14} + x^{12} + x^{10} + x^8 + x^7 + x + 1$ can generate $2^{16} - 1$ (65535) distinct 16-bit words, with the most and the least significant bytes denoted by B1 and B0, respectively.

Since there are 49152 clock cycles in one crypto block (256 frames), this 16-stage LFSR can be used as a pseudo-random binary sequence (PRBS) generator for data scrambling. Shifting occurs at every clock cycle, only when *valid_in* and *scrambler_ena* input ports are active. For synchronization purposes, it is reinitialized at every multi-frame (after an *mfs_in* pulse) with the sequence 0x5555.

The LFSR was implemented with an internal feedback construction (modular or Galois) with x^{16} as the most significant bit, as depicted in Figure 4.17.

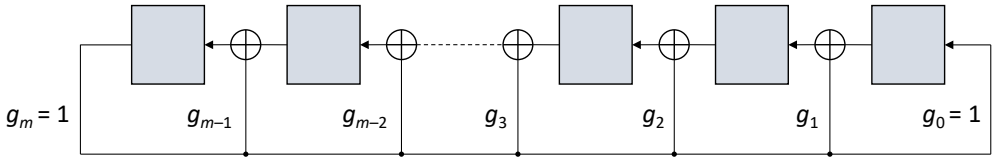


Figure 4.17 – Modular (Galois) LFSR with x^m as the most significant bit.

Overhead data (D) are XOR combined with a 16-bit PRBS output word (P), as detailed in Table 4.15, producing a scrambled sequence (S) at the time of insertion in the ODU overhead. Descrambling is accomplished later in the data extraction process by the use of the same synchronized LFSR, since if D ⊕ P = S, then S ⊕ P = D.

MFAS [1:0]	RES (row, col)							
	(2, 1)	(2, 2)	(4, 9)	(4, 10)	(4, 11)	(4, 12)	(4, 13)	(4, 14)
0	TAG ^ B1	TAG ^ B0	TAG ^ B1	TAG ^ B0	TAG ^ B1	TAG ^ B0	TAG ^ B1	TAG ^ B0
1	TAG ^ B1	TAG ^ B0	TAG ^ B1	TAG ^ B0	TAG ^ B1	TAG ^ B0	TAG ^ B1	TAG ^ B0
2	AAD ^ B1	AAD ^ B0	CSKS ^ B1	CSID ^ B0	CSID ^ B1	CSID ^ B0	CSID ^ B1	CBID ^ B0
3	AAD ^ B1	AAD ^ B0	CBID ^ B1	CBID ^ B0	CBID ^ B1	CPID ^ B0	CPID ^ B1	CPID ^ B0

Table 4.15 – Overhead data scrambling XOR operations.

4.3.8 ODU OH EXTRACTOR SUB-BLOCK

This sub-block handles data extraction from the ODU overhead.

Encryption overhead data (16 bytes of TAG, 4 bytes of AAD, and 12 bytes of IV) are extracted from four consecutive ODU overheads, each with eight available RES fields.

Additionally, all these data can be descrambled right after extraction.

Figure 4.18 shows its external input and output interface, and Table 4.16 brings the signal names, interface directions (input/output), descriptions, and the corresponding sources and destinations (from/to).

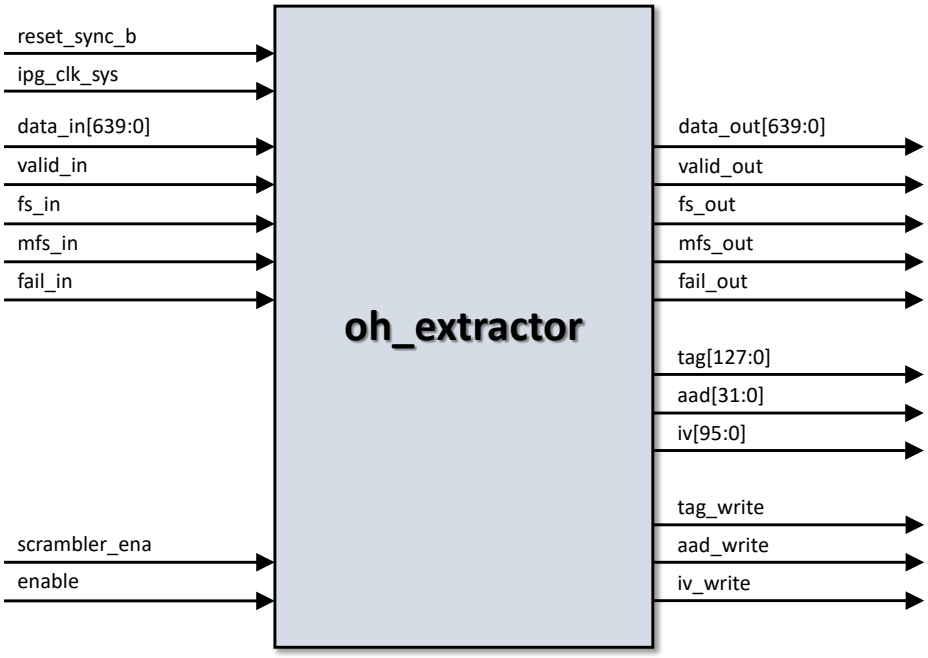


Figure 4.18 – ODU OH Extractor sub-block interface.

Signal	I/O	Description	From/To
reset_sync_b	I	Synchronous reset signal generated in the Control Engine sub-block. Active-low.	control_engine.rx_reset_sync_b
ipg_clk_sys	I	Green line interface clock at 180 MHz with duty cycle of 50%. Used for internal global logic.	crypto4otn.ipg_clk_sys_rx
data_in[639:0]	I	Padded ODU input data.	loopback_mux(RX processor).data_out
valid_in	I	Input data valid. Validates the ODU input data at the *data_in* bus, as well as the *fs_in*, *mfs_in*, and *fail_in* signals. Active-high.	loopback_mux(RX processor).valid_out

Signal	I/O	Description	From/To
fs_in	I	Input data frame start pulse. Single clock period pulse that precedes the start of the ODU input data frame at the *data_in* bus. Active-high.	loopback_mux(RX processor).fs_out
mfs_in	I	Input data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU input data multi-frame at the *data_in* bus. Active-high.	loopback_mux(RX processor).mfs_out
fail_in	I	Fail signal. Active-high.	loopback_mux(RX processor).fail_out
scrambler_ena	I	Scrambler enable control. Active-high.	control_engine.rx_oh_scrambler_ena
enable	I	Block enable control. '0' – Block disabled and all output ports de-asserted (at logic '0'). '1' – Block enabled.	control_engine.rx_enable
data_out[639:0]	O	Padded ODU output data.	crypto_engine(RX processor).data_in
valid_out	O	Output data valid. Validates the ODU output data at the *data_out* bus, as well as the *fs_out*, *mfs_out*, and *fail_out* signals. Active-high.	crypto_engine(RX processor).valid_in
fs_out	O	Output data frame start pulse. Single clock period pulse that precedes the start of the ODU output data frame at the *data_out* bus. Active-high.	crypto_engine(RX processor).fs_in
mfs_out	O	Output data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU output data multi-frame at the *data_out* bus. Active-high.	crypto_engine(RX processor).mfs_in
fail_out	O	Fail signal. Active-high.	crypto_engine(RX processor).fail_in
tag[127:0]	O	Authentication TAG corresponding to the previous crypto packet.	control_engine.rx_rcvd_tag
aad[31:0]	O	Additional authenticated data.	control_engine.rx_rcvd_aad crypto_engine(RX processor).aad
iv[95:0]	O	Initialization vector made up of a fixed field (CSKS and CSID) and an invocation field (CBID and CPID).	control_engine.rx_rcvd_iv crypto_engine(RX processor).iv
tag_write	O	TAG write enable pulse. TAG input bus value is captured and stored on the rising edge of *tag_write*.	control_engine.rx_rcvd_tag_write
aad_write	O	AAD write enable pulse. AAD input bus value is captured and stored on the rising edge of *aad_write*.	control_engine.rx_rcvd_aad_write
iv_write	O	IV write enable pulse. IV input bus value is captured and stored on the rising edge of *iv_write*.	control_engine.rx_rcvd_iv_write

Table 4.16 – ODU OH Extractor sub-block interface signals.

The encryption overhead bytes (TAG, AAD, and IV) are extracted from the eight RES fields of the ODU overhead, with location (row, column) shown in Figure 3.5. Since four consecutive frames are needed, a clock cycle counter synchronized by the multi-frame start signal (*mfs_in*) is used to determine the extraction positions.

In the beginning of each frame line, the 14 most significant bytes of the 640-bit bus correspond to the OTN overhead fields. Data are extracted during clock cycles number 49 and 145 in a given ODU frame.

Extracted data (encryption overhead) are internally stored and sent to the respective output port as soon as they are available. All other incoming data are forwarded to the output port, except for the ODU overhead RES fields, which are filled with '0' in the output frame.

The input fail indication signal (*fail_in*) is registered to the respective output (*fail_out*) with a delay equivalent to the ODU OH Extractor processing latency.

4.3.8.1 DESCRAMBLING FUNCTION

Depending on the state of the *scrambler_ena* input port, all extracted data can be descrambled by the same mechanism described in Section 4.3.7.1. Since the scrambling was performed by XOR operations combining data with PRBS words, descrambling is accomplished by repeating the same operations between scrambled data and the same corresponding PRBS words.

The same 16-stage Galois LFSR is used as a PRBS generator, with bit shifting occurring at every clock cycle, only when *valid_in* and *scrambler_ena* input ports are active. Again, for synchronization purposes, it is reinitialized at every multi-frame (after an *mfs_in* pulse) with the sequence 0x5555.

4.3.9 REPLACEMENT SIGNAL GENERATOR SUB-BLOCK

The Replacement Signal Generator sub-block generates an AIS-like (alarm indication signal) or user-defined pattern replacement signal in the output of the TX processor.

Figure 4.19 shows its external input and output interface, and Table 4.17 brings the signal names, interface directions (input/output), descriptions, and the corresponding sources and destinations (from/to).

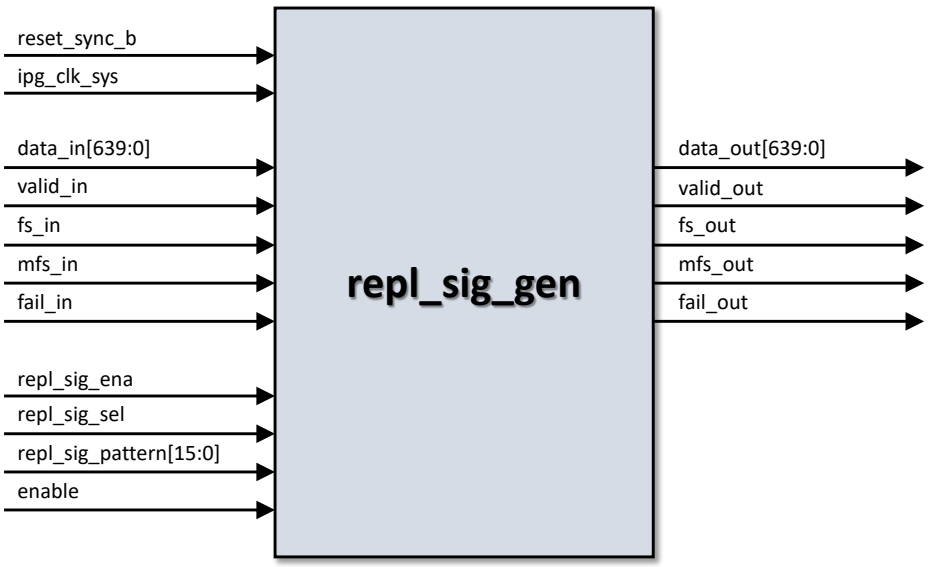


Figure 4.19 – Replacement Signal Generator sub-block interface.

Signal	I/O	Description	From/To
reset_sync_b	I	Synchronous reset signal generated in the Control Engine sub-block. Active-low.	control_engine.tx_reset_sync_b
ipg_clk_sys	I	Green line interface clock at 180 MHz with duty cycle of 50%. Used for internal global logic.	crypto4otn.ipg_clk_sys_tx
data_in[639:0]	I	Padded ODU input data.	oh_inserter.data_out
valid_in	I	Input data valid. Validates the ODU input data at the *data_in* bus, as well as the *fs_in*, *mfs_in*, and *fail_in* signals. Active-high.	oh_inserter.valid_out
fs_in	I	Input data frame start pulse. Single clock period pulse that precedes the start of the ODU input data frame at the *data_in* bus. Active-high.	oh_inserter.fs_out
mfs_in	I	Input data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU input data multi-frame at the *data_in* bus. Active-high.	oh_inserter.mfs_out
fail_in	I	Fail signal. Active-high.	oh_inserter.fail_out

Signal	I/O	Description	From/To
repl_sig_ena	I	Replacement signal enable control. Active-high.	control_engine
repl_sig_sel	I	Replacement signal selection. '0' – AIS-like pattern is generated as the replacement signal. '1' – User defined pattern is generated as the replacement signal.	control_engine
repl_sig_pattern[15:0]	I	Replacement signal pattern. 16-bit word repeatedly concatenated to make up the replacement signal.	control_engine
enable	I	Block enable control. '0' – Block disabled and all output ports de-asserted (at logic '0'). '1' – Block enabled.	control_engine.tx_enable
data_out[639:0]	O	Padded ODU output data.	flow_control(TX processor).data_in
valid_out	O	Output data valid. Validates the ODU output data at the *data_out* bus, as well as the *fs_out*, *mfs_out*, and *fail_out* signals. Active-high.	flow_control(TX processor).valid_in
fs_out	O	Output data frame start pulse. Single clock period pulse that precedes the start of the ODU output data frame at the *data_out* bus. Active-high.	flow_control(TX processor).fs_in
mfs_out	O	Output data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU output data multi-frame at the *data_out* bus. Active-high.	flow_control(TX processor).mfs_in
fail_out	O	Fail signal. Active-high.	flow_control(TX processor).fail_in_a

Table 4.17 – Replacement Signal Generator sub-block interface signals.

When *repl_sig_ena* signal is asserted by the Control Engine sub-block, output data are replaced by either an AIS-like or a user-defined pattern, depending on the state of *repl_sig_sel* input.

If AIS-like is selected (*repl_sig_sel* = '0'), almost all bits of the ODU frame are set to '1', except for the frame alignment overhead (FA OH), OTU overhead (OTU OH), ODU FTFL, and ODU overhead RES fields, as shown in Figure 4.20.

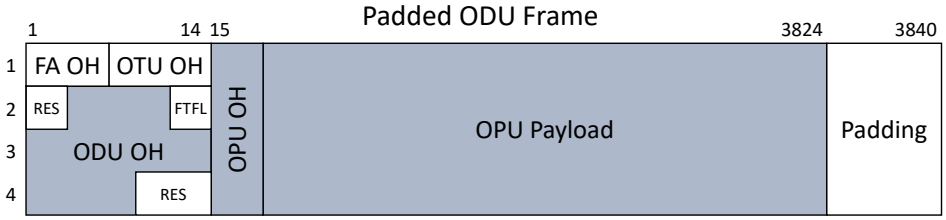


Figure 4.20 – ODU AIS-like — all '1s' pattern in the gray area.

If the user-defined pattern is selected instead (*repl_sig_sel* = '1'), then only the OPU overhead and payload areas of the padded ODU frames are replaced by new data generated by the concatenated repetition of a single 16-bit word read from the *repl_sig_pattern* input. The 16 padding bytes at the end of each line remain '0', as shown in Figure 4.21

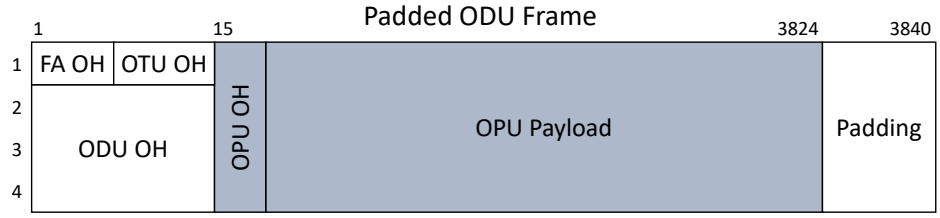


Figure 4.21 – ODU user-defined pattern in the gray area.

The data replacement action depends only on the *repl_sig_ena* control input, not being synchronized with any other signal.

The input fail indication signal (*fail_in*) is registered to the respective output (*fail_out*) with a delay equivalent to the Replacement Signal Generator processing latency.

4.3.10 AUTHENTICATION BUFFER SUB-BLOCK

When the Crypto4OTN block operates in authenticated encryption or authentication-only modes, with the authentication delay configured for the store-and-forward mode, this sub-block stores six ODU frames (corresponding to 1.5 crypto packets) and forwards them after TAG matching verification.

Figure 4.22 shows its external input and output interface, and Table 4.18 brings the signal names, interface directions (input/output), descriptions, and the corresponding sources and destinations (from/to).

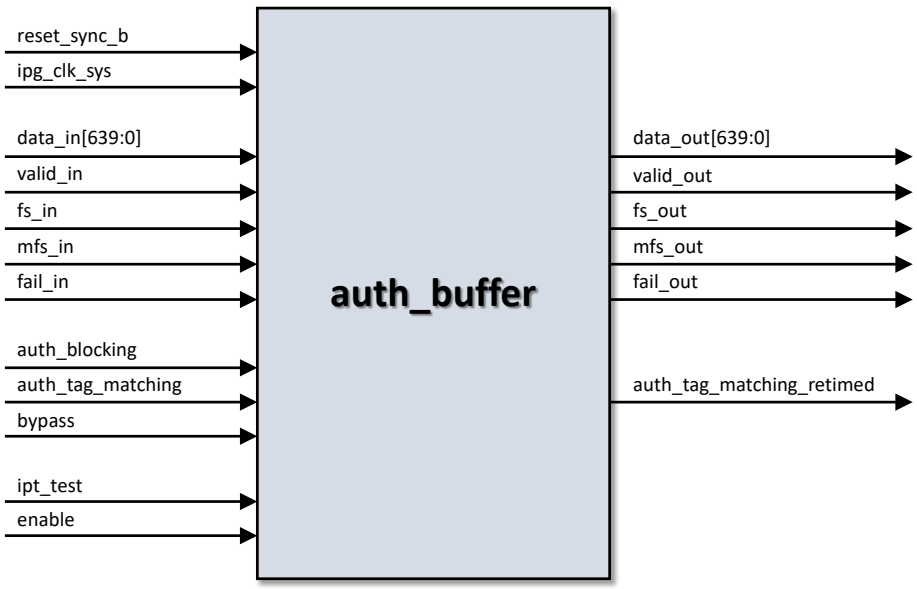


Figure 4.22 – Authentication Buffer sub-block interface.

Signal	I/O	Description	From/To
reset_sync_b	I	Synchronous reset signal generated in the Control Engine sub-block. Active-low.	control_engine.rx_reset_sync_b
ipg_clk_sys	I	Green line interface clock at 180 MHz with duty cycle of 50%. Used for internal global logic.	crypto4otn.ipg_clk_sys_rx
data_in[639:0]	I	Padded ODU input data.	crypto_engine(RX processor).data_out
valid_in	I	Input data valid. Validates the ODU input data at the *data_in* bus, as well as the *fs_in*, *mfs_in*, and *fail_in* signals. Active-high.	crypto_engine(RX processor).valid_out

Signal	I/O	Description	From/To
fs_in	I	Input data frame start pulse. Single clock period pulse that precedes the start of the ODU input data frame at the *data_in* bus. Active-high.	crypto_engine(RX processor).fs_out
mfs_in	I	Input data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU input data multi-frame at the *data_in* bus. Active-high.	crypto_engine(RX processor).mfs_out
fail_in	I	Fail signal. Active-high.	crypto_engine(RX processor).fail_out
auth_blocking	I	Authentication blocking control. Active-high. '0' – OPU data are forwarded regardless of the authentication TAG matching indication status. '1' – OPU data are replaced by zero in the case of a TAG mismatch.	control_engine
auth_tag_matching	I	Authentication TAG matching indication. Updated at every crypto packet, after TAG verification. Active-high.	control_engine
bypass	I	Bypass control. '0' – Normal operation. '1' – Bypass mode.	control_engine.auth_buffer_bypass
ipt_test	I	Test mode enable signal (for DFT purposes). Active-high.	crypto4otn
enable	I	Block enable control. '0' – Block disabled and all output ports de-asserted (at logic '0'). '1' – Block enabled.	control_engine.rx_enable
data_out[639:0]	O	Padded ODU output data.	flow_control(RX processor).data_in
valid_out	O	Output data valid. Validates the ODU output data at the *data_out* bus, as well as the *fs_out*, *mfs_out*, and *fail_out* signals. Active-high.	flow_control(RX processor).valid_in
fs_out	O	Output data frame start pulse. Single clock period pulse that precedes the start of the ODU output data frame at the *data_out* bus. Active-high.	flow_control(RX processor).fs_in
mfs_out	O	Output data multi-frame start pulse. Single clock period pulse that precedes the start of the ODU output data multi-frame at the *data_out* bus. Active-high.	flow_control(RX processor).mfs_in
fail_out	O	Fail signal. Active-high.	flow_control(RX processor).fail_in

Signal	I/O	Description	From/To
auth_tag_matching_retimed	O	Retimed version of the authentication TAG matching indication, synchronized with the output crypto packets.	flow_control(RX processor).spare_bit_in

Table 4.18 – Authentication Buffer sub-block interface signals.

Data must be buffered to wait for authentication since the TAG value generated in the transmitter (to be compared with the one calculated in the receiver) of a given crypto packet comes at the beginning of the next one, as described in Section 3.4 and shown in Figure 4.23.

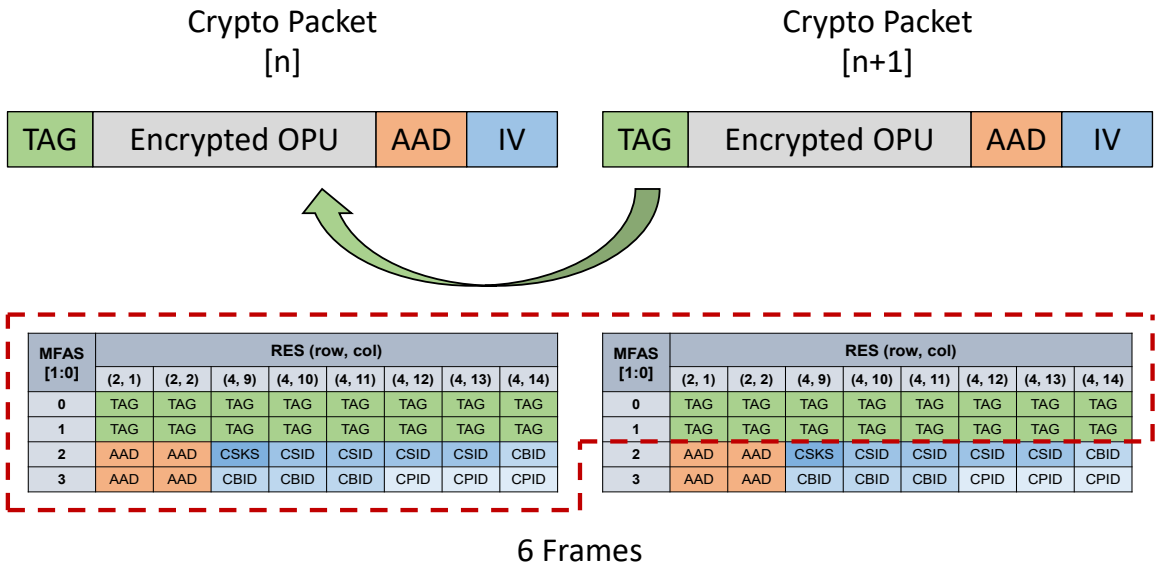


Figure 4.23 – Correlation of the authentication TAG and the encrypted OPU message of two consecutive crypto packets.

The 16-byte TAG data transmitted in the eight ODU overhead RES fields of the first two frames of a crypto packet are correlated to the encrypted OPU payload of the previous one. Consequently, six frames must be stored so that the TAG of the current crypto packet (calculated from the received encrypted OPU payload) can be compared to the TAG coming in the next one (previously calculated in the TX side) for matching verification before authenticated data are made available at the output.

When the *auth_blocking* input is asserted, the *auth_tag_matching* input controls whether the stored crypto packet (four oldest frames in the buffer) must be entirely forwarded to the output or have its OPU overheads and payloads replaced by '0'. Conversely, when

auth_blocking is '0', the stored crypto packet will be entirely forwarded to the output regardless of the TAG matching indication.

Input data are directly forwarded to the output (with no extra storing delay) when the *bypass* input is asserted. Output data behavior is unpredictable right after transitioning to/from the bypass mode.

The crypto packet boundary delineation is obtained from packet and frame counters, which are synchronized by the *mfs_in* signal.

The *auth_tag_matching_retimed* output is a retimed version of the *auth_tag_matching* input, synchronized with the output crypto packets.

The input fail indication signal (*fail_in*) is registered to the respective output (*fail_out*) with a delay equivalent to the Authentication Buffer processing latency.

4.3.11 CONTROL ENGINE SUB-BLOCK

This sub-block implements all the needed functionalities for control and synchronization of the 100G AES-GCM Cryptography Engine operation.

It is internally divided in two independent controllers (for the TX and RX processors) sharing a common Register and Reset Controller, as shown in Figure 4.24.

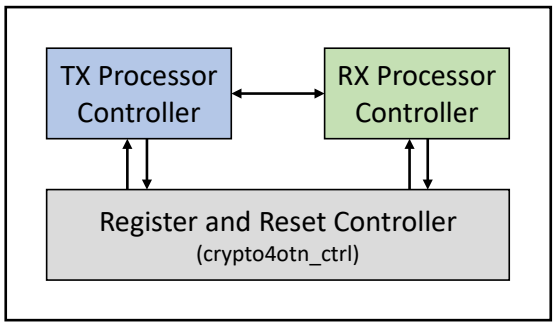


Figure 4.24 – Control Engine high-level architecture diagram.

Both internal controllers handle their own interrupt subset and consequent action procedures, interacting with each other when necessary. Since each controller belongs to a different clock domain, these interactions are implemented using proper clock domain crossing techniques.

These TX and RX processor controllers receive frame start (FS) and multi-frame start (MFS) synchronization pulses, which are used for crypto packet boundary delineation before generation of the controlling actions.

Figure 4.25 shows its external input and output interface, and Table 4.19 brings the signal names, interface directions (input/output), descriptions, and the corresponding sources and destinations (from/to).

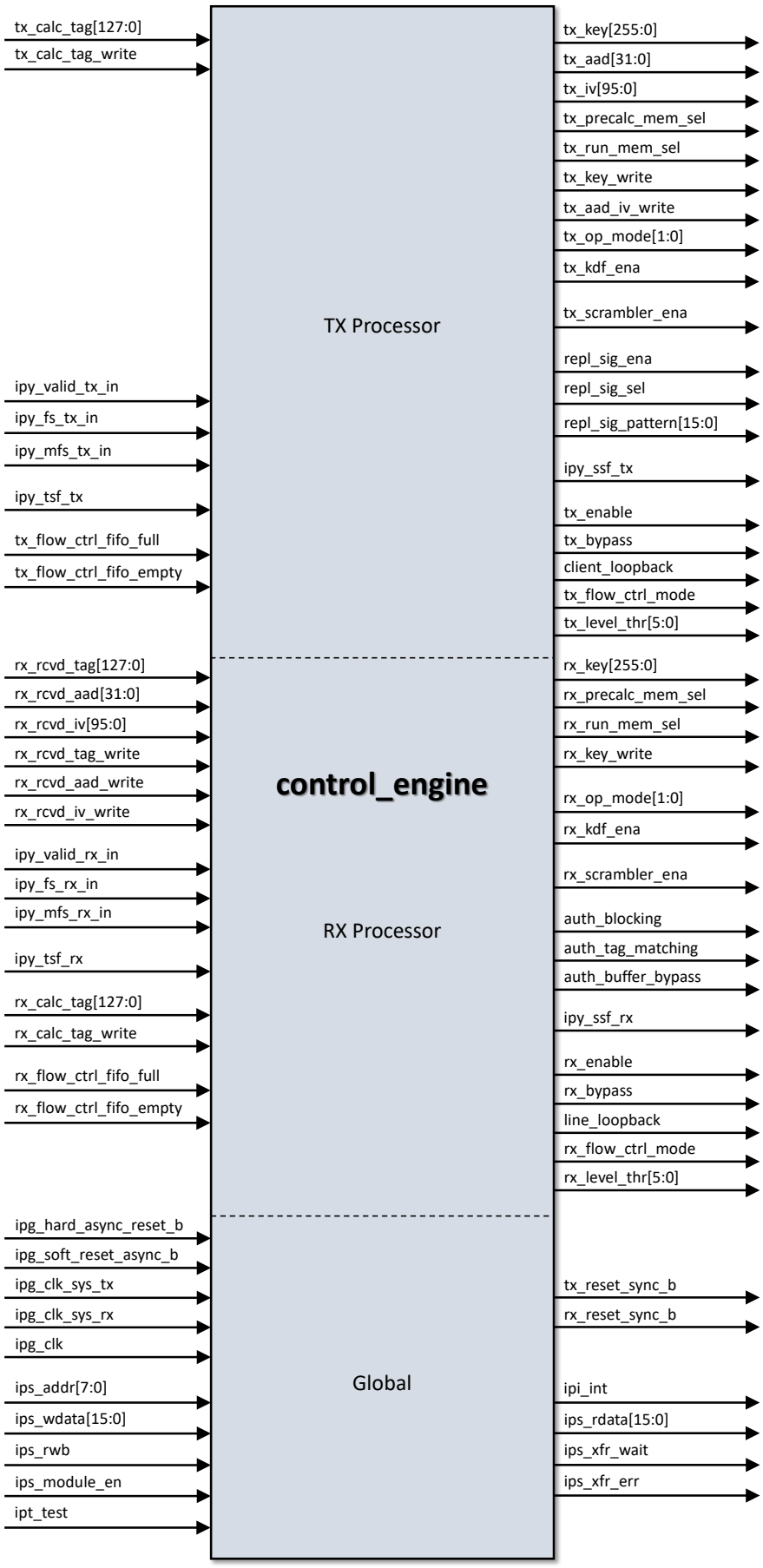


Figure 4.25 – Control Engine sub-block interface.

Signal	I/O	Description	From/To
ipg_hard_async_reset_b	I	Green line asynchronous hard reset signal. This signal is asserted and de-asserted asynchronously. Active-low.	crypto4otn
ipg_soft_reset_async_b	I	Green line interface soft reset signal. This signal is asserted and de-asserted asynchronously. Active-low.	crypto4otn
ipg_clk_sys_tx	I	TX processor green line interface clock at 180 MHz with duty cycle of 50%. Used for internal global logic.	crypto4otn
ipg_clk_sys_rx	I	RX processor Green line interface clock at 180 MHz with duty cycle of 50%. Used for internal global logic.	crypto4otn
ipg_clk	I	Green line interface clock at 90 MHz with duty cycle of 50%. Used for the register access interface.	crypto4otn
ips_addr[7:0]	I	Sky-Blue line interface address bus. Used for register access.	crypto4otn
ips_wdata[15:0]	I	Sky-Blue line interface write data bus. Used for register access.	crypto4otn
ips_rwb	I	Sky-Blue line interface read/write selection. '0' – Data are written by the CPU. '1' – Data are read by the CPU.	crypto4otn
ips_module_en	I	Sky-Blue line interface module enable. Selects the crypto4otn block for register access. Active-high.	crypto4otn
ipt_test	I	Test mode enable signal (for DFT purposes). Active-high.	crypto4otn
tx_calc_tag[127:0]	I	Authentication TAG generated for the current crypto packet.	crypto_engine.tag (TX processor)
tx_calc_tag_write	I	Authentication TAG write enable pulse. TAG input bus (*tx_calc_tag*) value is captured and stored on the rising edge of *tx_calc_tag_write*.	crypto_engine.tag_write (TX processor)
ipy_valid_tx_in	I	Yellow line interface input data valid. Validates the padded ODU input data at the Crypto4OTN *ipy_data_tx_in* bus. Active-high.	crypto4otn
ipy_fs_tx_in	I	Yellow line interface input data frame start pulse. Indicates the start of the ODU input data frame at the Crypto4OTN *ipy_data_tx_in* bus. Active-high.	crypto4otn
ipy_mfs_tx_in	I	Yellow line interface input data multi-frame start pulse. Indicates the start of the ODU input data multi-frame at the Crypto4OTN *ipy_data_tx_in* bus. Active-high.	crypto4otn
ipy_tsf_tx	I	TX processor input client data fail signal indication. Active-high.	crypto4otn

Signal	I/O	Description	From/To
tx_flow_ctrl_fifo_full	I	TX processor flow control FIFO full indication. Active-high.	flow_control.fifo_full (TX processor)
tx_flow_ctrl_fifo_empty	I	TX processor flow control FIFO empty indication. Active-high.	flow_control.fifo_empty (TX processor)
rx_rcvd_tag[127:0]	I	Received authentication TAG corresponding to the previous crypto packet.	oh_extractor.tag
rx_rcvd_aad[31:0]	I	Received additional authenticated data for the current crypto packet.	oh_extractor.aad
rx_rcvd_iv[95:0]	I	Received initialization vector made up of a fixed field (CSKS and CSID) and an invocation field (CBID and CPID), to be used for the next crypto packet.	oh_extractor.iv
rx_rcvd_tag_write	I	Authentication TAG write enable pulse. TAG input bus (*rx_rcvd_tag*) value is captured and stored on the rising edge of *rx_rcvd_tag_write*.	oh_extractor.tag_write
rx_rcvd_aad_write	I	AAD write enable pulse. AAD input bus (*rx_rcvd_aad*) value is captured and stored on the rising edge of *rx_rcvd_aad_write*.	oh_extractor.aad_write
rx_rcvd_iv_write	I	IV write enable pulse. IV input bus (*rx_rcvd_iv*) value is captured and stored on the rising edge of *rx_rcvd_iv_write*.	oh_extractor.iv_write
ipy_valid_rx_in	I	Yellow line interface input data valid. Validates the padded ODU input data at the Crypto4OTN *ipy_data_rx_in* bus. Active-high.	crypto4otn
ipy_fs_rx_in	I	Yellow line interface input data frame start pulse. Indicates the start of the ODU input data frame at the Crypto4OTN *ipy_data_rx_in* bus. Active-high.	crypto4otn
ipy_mfs_rx_in	I	Yellow line interface input data multi-frame start pulse. Indicates the start of the ODU input data multi-frame at the Crypto4OTN *ipy_data_rx_in* bus. Active-high.	crypto4otn
ipy_tsf_rx	I	RX processor input line data fail signal indication. Active-high.	crypto4otn
rx_calc_tag[127:0]	I	Authentication TAG generated for the current crypto packet.	crypto_engine.tag (RX processor)
rx_calc_tag_write	I	Authentication TAG write enable pulse. TAG input bus (*rx_calc_tag*) value is captured and stored on the rising edge of *rx_calc_tag_write*.	crypto_engine.tag_write (RX processor)
rx_flow_ctrl_fifo_full	I	RX processor flow control FIFO full indication. Active-high.	flow_control.fifo_full (RX processor)
rx_flow_ctrl_fifo_empty	I	RX processor flow control FIFO empty indication. Active-high.	flow_control.fifo_empty (RX processor)

Signal	I/O	Description	From/To
tx_reset_sync_b	O	TX processor synchronous reset signal generated in the *crypto4otn_ctrl* sub-block (Register and Reset Controller). Active-low.	TX processor sub-blocks
rx_reset_sync_b	O	RX processor synchronous reset signal generated in the *crypto4otn_ctrl* sub-block (Register and Reset Controller). Active-low.	RX processor sub-blocks
ipi_int	O	Indigo line interface interrupt request signal. Active-high.	crypto4otn
ips_rdata[15:0]	O	Sky-Blue line interface read data bus. Used for register access.	crypto4otn
ips_xfr_wait	O	Sky-Blue line interface transfer wait signal. When in logic low '0' it indicates that the access is complete. Active-high.	crypto4otn
ips_xfr_err	O	Sky-Blue line interface transfer error signal. Active-high.	crypto4otn
tx_key[255:0]	O	256-bit crypto session key for the current crypto packet.	crypto_engine.key (TX processor)
tx_aad[31:0]	O	Additional authenticated data for the current crypto packet.	crypto_engine.aad (TX processor) oh_inserter.aad
tx_iv[95:0]	O	Initialization vector made up of a fixed field (CSKS and CSID) and an invocation field (CBID and CPID) for the next crypto packet.	crypto_engine.iv (TX processor) oh_inserter.iv
tx_precalc_mem_sel	O	Pre-calculation memory selection.	crypto_engine.precalc_mem_sel (TX processor)
tx_run_mem_sel	O	Running memory selection, from where pre-calculated data are read by the AES-GCM algorithm.	crypto_engine.run_mem_sel (TX processor)
tx_key_write	O	Key write enable and pre-calculation trigger pulse.	crypto_engine.key_write (TX processor)
tx_aad_iv_write	O	AAD and IV write enable and pre-calculation trigger pulse.	crypto_engine.aad_write (TX processor) crypto_engine.iv_write (TX processor) oh_inserter.aad_iv_write
tx_op_mode[1:0]	O	Crypto session operation mode. '00' – Authenticated encryption. '01' – Authentication-only. '10' – Encryption-only.	crypto_engine.op_mode (TX processor)
tx_kdf_ena	O	Key derivation function enable control. Active-high. When enabled, key values are scrambled before being delivered to the AES-GCM sub-block.	crypto_engine.kdf_ena (TX processor)
tx_scrambler_ena	O	Overhead scrambler enable control. Active-high. When enabled, overhead data are scrambled before being inserted into ODU frames.	oh_inserter.scrambler_ena

Signal	I/O	Description	From/To
repl_sig_ena	O	Replacement signal enable control. Active-high.	repl_sig_gen
repl_sig_sel	O	Replacement signal selection. '0' – AIS-like pattern is generated as the replacement signal. '1' – User defined pattern is generated as the replacement signal.	repl_sig_gen
repl_sig_pattern[15:0]	O	Replacement signal pattern. 16-bit word to be repeatedly concatenated to make up the replacement signal.	repl_sig_gen
ipy_ssf_tx	O	TX processor output line data fail signal indication. Active-high.	crypto4otn
tx_enable	O	TX processor enable control. '0' – TX processor disabled. '1' – TX processor enabled.	TX processor sub-blocks
tx_bypass	O	TX processor bypass control. '0' – Normal operation. '1' – Bypass mode.	flow_control.bypass (TX processor)
client_loopback	O	Client side far-end loopback control. '0' – Normal operation. '1' – Client loopback mode.	loopback_mux.loopback (TX processor)
tx_flow_ctrl_mode	O	TX processor flow control operation mode. '0' – Push mode. '1' – Pull mode.	flow_control.flow_control_mode (TX processor)
tx_level_thr[5:0]	O	TX processor flow control FIFO level threshold.	flow_control.level_threshold (TX processor)
rx_key[255:0]	O	256-bit crypto session key for the current crypto packet.	crypto_engine.key (RX processor)
rx_precalc_mem_sel	O	Pre-calculation memory selection.	crypto_engine.precalc_mem_sel (RX processor)
rx_run_mem_sel	O	Running memory selection, from where pre-calculated data are read by the AES-GCM algorithm.	crypto_engine.run_mem_sel (RX processor)
rx_key_write	O	Key write enable and pre-calculation trigger pulse.	crypto_engine.key_write (RX processor)
rx_op_mode[1:0]	O	Crypto session operation mode. '00' – Authenticated encryption. '01' – Authentication-only. '10' – Encryption-only.	crypto_engine.op_mode (RX processor)
rx_kdf_ena	O	Key derivation function enable control. Active-high. When enabled, key values are scrambled before being delivered to the AES-GCM sub-block.	crypto_engine.kdf_ena (RX processor)

Signal	I/O	Description	From/To
rx_scrambler_ena	O	Overhead scrambler enable control. Active-high. When enabled, overhead data are scrambled after being extracted from ODU frames (equivalent to descrambling).	oh_extractor.scrambler_ena
auth_blocking	O	Authentication blocking control. Active-high. '0' – OPU data are forwarded regardless of the authentication TAG matching indication status. '1' – OPU data are replaced by zero in the case of a TAG mismatch.	auth_buffer
auth_tag_matching	O	Authentication TAG matching indication. Updated at every crypto packet, after TAG verification. Active-high.	auth_buffer
auth_buffer_bypass	O	Authentication Buffer bypass control. '0' – Normal operation. '1' – Bypass mode.	auth_buffer
ipy_ssf_rx	O	RX processor output client data fail signal indication. Active-high.	crypto4otn
rx_enable	O	RX processor enable control. '0' – RX processor disabled. '1' – RX processor enabled.	RX processor sub-blocks
rx_bypass	O	RX processor bypass control. '0' – Normal operation. '1' – Bypass mode.	flow_control.bypass (RX processor)
line_loopback	O	Line side far-end loopback control. '0' – Normal operation. '1' – Line loopback mode.	loopback_mux.loopback (RX processor)
rx_flow_ctrl_mode	O	RX processor flow control operation mode. '0' – Push mode. '1' – Pull mode.	flow_control.flow_control_mode (RX processor)
rx_level_thr[5:0]	O	RX processor flow control FIFO level threshold.	flow_control.level_threshold (RX processor)

Table 4.19 – Control Engine sub-block interface signals.

4.3.11.1 TX PROCESSOR CONTROLLER

The TX processor controller handles the crypto session establishment and maintenance procedures on the transmitter side, as commanded by the software layer.

Some of its main general requirements are listed in Table 4.20.

Requirement	Description
CPID counter	24-bit free running and wrap-around counter, restarted after a reset event. Incremented at every crypto packet, when MFAS [1:0] = 0. Used to form the invocation field of the initialization vector.
CBID counter	32-bit non-wrap-around counter, restarted at the beginning of a crypto session. It runs when the session is active and increments at every 64 crypto packets, right after an MFS pulse. Used to form the invocation field of the initialization vector and to verify the crypto session expiration.
(8, 1) repetition FEC encoder	Repetition FEC encoder for the key selection information. The CSKS 8-bit code word (CSKS = 8{~ACTIVE_KEY}) is generated after a KEY_CHANGE pulse. Final encoding is accomplished by the CSKS transmission over one complete crypto block (64 crypto packets). On the receiver side, decoding happens over 63 crypto packets, resulting in a final 504-bit repetition code.
MFS Watchdog Timer	Watchdog timer that monitors the presence of MFS pulses in the three operation modes: authenticated encryption, authentication-only, and encryption-only. It is restarted after a reset event. When it reaches a time-lapse greater than 10 seconds, the session state is forced to inactive.
IV and AAD Pre-Calculation Pulse Generators	Two concurrent one-cycle pulses generated in the middle of all crypto packets (at MFAS = 2, 6, 10, …, 254) corresponding to the *tx_iv_write* and *tx_aad_write* signals, which trigger the IV and AAD pre-calculation processes.

Table 4.20 – TX processor main general requirements.

Table 4.21 describes actions and procedures for a simplified finite state machine (FSM) valid for the three operation modes: authenticated encryption, authentication-only, and encryption-only. All controlling actions are time-adjusted for each sub-block, based on the FS, MFS, and MFAS synchronization signals.

State	Actions/Procedures
INIT	▪ SESSION_STATUS = '0', indicating that session is not established (insecure); ▪ Request enabling of the Replacement Signal Generator; ▪ KEY_SWAP = 0, indicating that key must be changed at MFAS = 252; ▪ ACTIVE_KEY = '1' (active key selection); ▪ *tx_precalc_mem*_sel = '0' (pre-calculation memory page selection); ▪ *tx_run_mem_sel* = '1' (running memory page selection); ▪ SESSION_START_REQ = '0';
SEND_IV_SYNC	▪ *tx_iv* = {0xff, 0, 0, 0} (send the next IV value {CSKS, CSID, CBID, CPID} with CSKS = 0xff to ODU OH Inserter and OPU Cryptography Engine sub-blocks); ▪ Wait for SESSION_START_REQ = '1', KEY_CHANGE_REQ = '1' and an MFS pulse;

KEY_CHANGE	▪ *tx_iv* = {0x00, 0, 0, 0} (send the next IV value{CSKS, CSID, CBID, CPID} with CSKS = 0x00 to ODU OH Inserter and OPU Cryptography Engine sub-blocks);
	▪ Wait for MFAS = 252 (last crypto packet of a crypto block) and then:
	  ▪ *tx_key* = *rgt_tx_key_0* (change the key value at the corresponding output port);
	  ▪ Pulse *tx_key_write*, triggering the key pre-calculation process;
	  ▪ Wait for MFAS = 253 (right before the middle of the last crypto packet);

SEND_IV	▪ CSID = *rgt_tx_iv_csid* (load CSID field from the corresponding register);
	▪ *tx_iv* = {0x00, CSID, 0, CPID} (send the next IV value {CSKS, CSID, CBID, CPID} to ODU OH Inserter and OPU Cryptography Engine sub-blocks);
	▪ CSKS_HOLD = 0x00;
	▪ If AAD_CAPTURE = '1', capture the AAD value from the corresponding register:
	  ▪ AAD = *rgt_tx_aad_buffer*;
	  ▪ AAD_CAPTURE = '0' (clear the AAD capture flag);
	▪ *tx_aad* = AAD (send the AAD value to ODU OH Inserter and OPU Cryptography Engine sub-blocks);
	▪ KEY_CHANGE_REQ = '0';
	▪ ACTIVE_KEY = '0' (active key selection);
	▪ PRECALC_SECOND_MEM = '1';
	▪ Wait for an MFS pulse;

ACTIVE_SESSION	▪ SESSION_STATUS = '1', indicating that session is established (secure);
	▪ Request disabling of the Replacement Signal Generator (it may keep enabled by a programmed consequent action or by the software layer);
	
	▪ If MFAS [1:0] = 0 (start of a crypto packet):
	  ▪ *tx_run_mem_sel* = MFAS[2] (running memory page selection);
	  ▪ *tx_precalc_mem_sel* = ~MFAS[2] (pre-calculation memory page selection);
	  ▪ If *crypto* session has just started, clear CBID;
	  ▪ If PRECALC_SECOND_MEM = '1':
	  ▪ Pulse *tx_key_write*, triggering the key pre-calculation process;
	  ▪ PRECALC_SECOND_MEM = '0';
	
	▪ If MFAS [7:0] = 0 (start of a crypto block):
	  ▪ If KEY_CHANGE_REQ = '1':
	  ▪ KEY_CHANGE_REQ = '0';
	  ▪ CSKS_HOLD = CSKS;
	  ▪ KEY_SWAP = 1;
	
	▪ If MFAS [7:0] = 252 (last crypto packet of a crypto block):
	  ▪ If KEY_SWAP = '1':
	  ▪ Change the key at the corresponding output port:
	  ▪ If ACTIVE_KEY = '1', *tx_key* = *rgt_tx_key_0*;
	  ▪ If ACTIVE_KEY = '0', *tx_key* = *rgt_tx_key_1*;
	  ▪ Pulse *tx_key_write*, triggering the key pre-calculation process;
	  ▪ ACTIVE_KEY = ~ACTIVE_KEY;
	  ▪ PRECALC_SECOND_MEM = '1';
	  ▪ CSID = *rgt_tx_iv_csid*;
	
	▪ If MFAS [1:0] = 2 (middle of a crypto packet):
	  ▪ If MFAS = 254 and KEY_SWAP = '1' (end of a crypto block and key change request):

State	Actions/Procedures
	▪ *tx_iv* = {CSKS_HOLD, CSID, 0, CPID} (send the next IV value to ODU OH Inserter and OPU Cryptography Engine sub-blocks); ▪ KEY_SWAP = '0'; ▪ Else if MFAS = 254 (just end of a crypto block): ▪ *tx_iv* = {CSKS_HOLD, CSID, CBID+1, CPID}; ▪ Else (just middle of a crypto packet): ▪ *tx_iv* = {CSKS_HOLD, CSID, CBID, CPID}; ▪ If AAD_CAPTURE = '1', capture the AAD value from the corresponding register: ▪ AAD = *rgt_tx_aad_buffer*; ▪ AAD_CAPTURE = '0'; ▪ *tx_aad* = AAD (send the AAD value to ODU OH Inserter and OPU Cryptography Engine sub-blocks);

Table 4.21 – TX processor session control state actions.

When the crypto session is not established, ACTIVE_KEY = 1, and the software layer must write the next (starting) key in the *tx_key_0* register.

The SESSION_START_REQ flag is asserted only by a '0' → '1' transition in the SESSION_STATE bit of the *tx_session_state* register, indicating a crypto session start request from the software layer.

The CSKS_HOLD register is updated in the beginning of a crypto block only if there is a pending key change request (KEY_CHANGE_REQ = 1). It is used to hold the value of CSKS and avoid disturbances in the transmitted IV, since the KEY_CHANGE bit is asynchronously controlled by the software layer.

Figure 4.26 depicts the TX processor controller FSM state diagram.

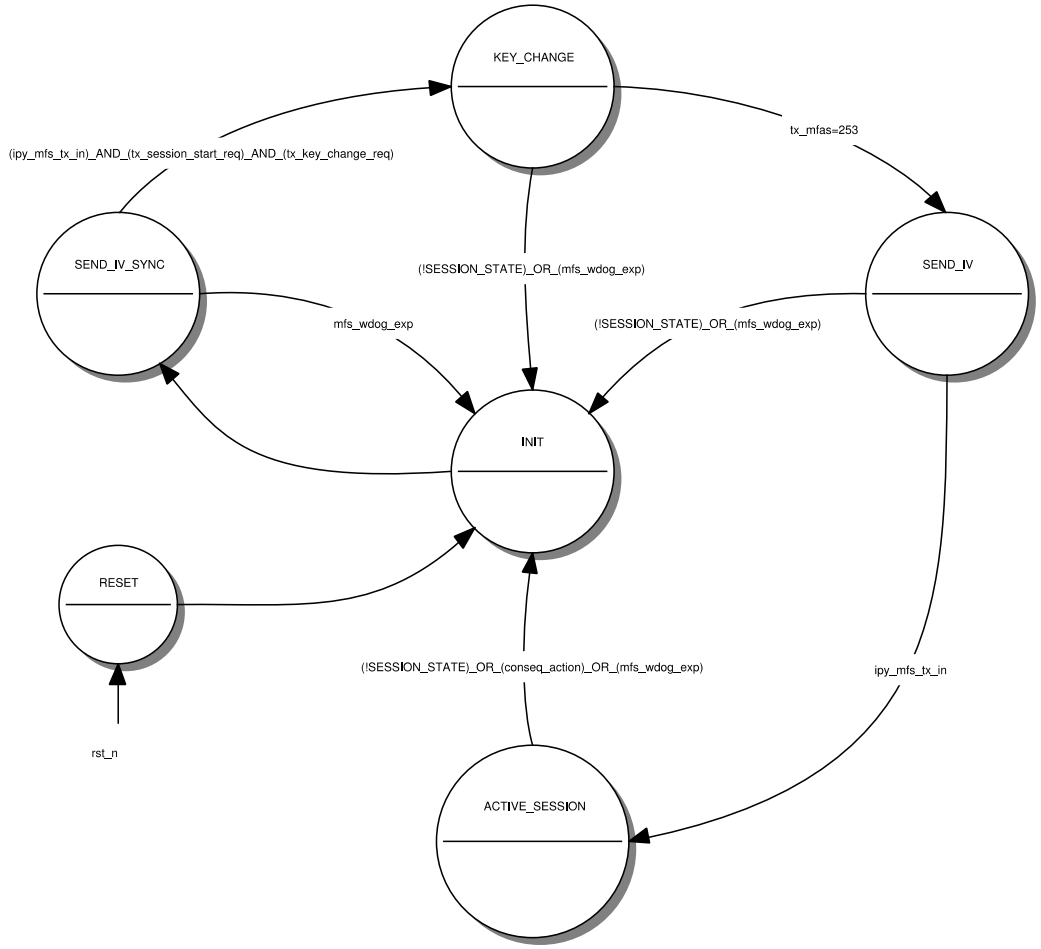


Figure 4.26 – TX processor controller finite state machine state diagram.

Figure 4.27 illustrates some of the TX control signals as a function of the MFAS values. The "key_write" control, enclosed by dotted lines, occurs only when there is a key change request (either in the first crypto session establishment or whenever commanded by the software layer).

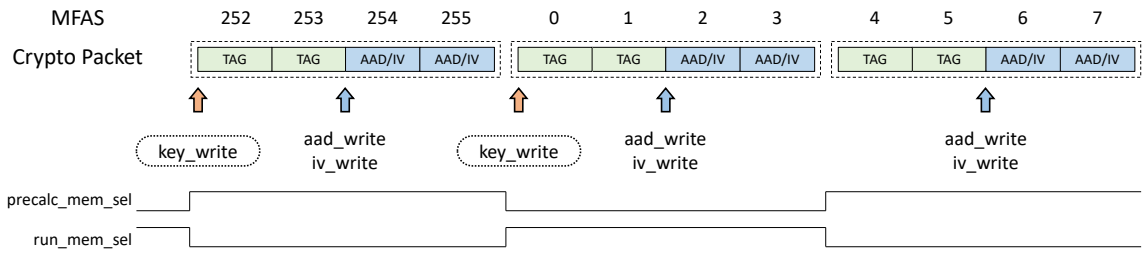


Figure 4.27 – Some TX control signals as a function of MFAS values.

When the crypto session is inactive, CSKS = 0xff is sent to the RX side. After SESSION_START_REQ = 1 and KEY_CHANGE_REQ = 1, the TX processor controller

waits for an MFS pulse and then CSKS = 0x00 is transmitted to indicate a key change process, which will be decoded by the RX processor controller at MFAS = 252.

At this time, the *tx_key_write* signal is pulsed to trigger a key pre-calculation process in the AES-GCM sub-block, and a flag (PRECALC_SECOND_MEM) is set to request this pre-calculation for the second memory right in the beginning of the next crypto block.

When the crypto session is active and a new key change process is requested, the TX processor controller waits for MFAS = 252 and repeats these same procedures.

In both inactive and active crypto sessions, the *tx_aad_iv_write* signal is pulsed in the middle of all crypto packets (at MFAS = 2, 6, …, 254) to transfer data to ODU OH Inserter and OPU Cryptography Engine sub-blocks and to trigger the AAD and IV pre-calculation processes.

4.3.11.2 RX PROCESSOR CONTROLLER

The RX processor controller handles the crypto session establishment and maintenance procedures on the receiver side, as commanded by the software layer.

Some of its main general requirements are listed in Table 4.22.

Requirement	Description
(504, 1) repetition FEC decoder	Repetition FEC decoder for the key selection information, received as a CSKS 8-bit code word and decoded by a majority gate over 63 crypto packets, resulting in a final 504-bit repetition code. Decoding process starts right after an MFS pulse and finishes at MFAS = 252, when decoded information is stored in DEC_KEY_SEL. If the number of bits '1' is greater than 252 (504/2), DEC_KEY_SEL = '1'. Otherwise, DEC_KEY_SEL = '0'.
MFS Watchdog Timer	Watchdog timer that monitors the presence of MFS pulses in the three operation modes: authenticated encryption, authentication-only, and encryption-only. It is restarted after a reset event. When it reaches a time-lapse greater than 10 seconds, session state is forced to inactive.
Maskable sliding window for TAG mismatch counting	64-bit shift register storing the TAG_FAIL status of 64 crypto packets. MSB corresponds to the current crypto packet and LSB, to the oldest one. Masked outputs are added to generate a 7-bit event count, updated at every crypto packet.

Table 4.22 – RX processor main general requirements.

Loss of authentication (LOA) indication is generated by TAG matching failure counting within a sliding window. A counter threshold is used for both assertion and de-assertion of the LOA indication and the window size can be configured by a masking register, as illustrated in Figure 4.28.

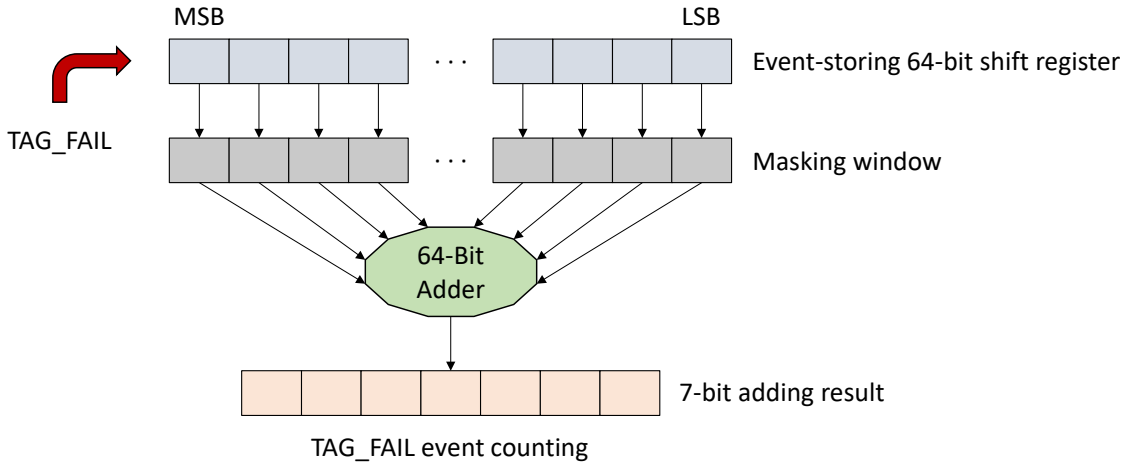


Figure 4.28 – Loss of authentication (LOA) 64-bit sliding window mechanism.

TAG_FAIL events are registered in a 64-bit shift register (starting at MSB), corresponding to a 64 crypto packet sliding window. By controlling a 64-bit mask, different window sizes and patterns can be created to limit and delimitate the event counting. If the configured threshold is reached and the session is active, LOA is asserted, otherwise it is de-asserted.

Both TAG_FAIL and LOA flags are cleared if the crypto session is closed (inactive).

Table 4.23 describes actions and procedures for a simplified finite state machine valid for the three operation modes: authenticated encryption, authentication-only, and encryption-only. All controlling actions are time-adjusted for each sub-block, based on the FS, MFS, and MFAS synchronization signals.

State	Actions/Procedures
INIT	▪ SESSION_STATUS = '0', indicating that session is not established (insecure); ▪ *auth_tag_matching* = '0'; ▪ Request assertion of the *ipy_ssf_rx* output port; ▪ ACTIVE_KEY = '1' (active key selection); ▪ *rx_precalc_mem_sel* = '0' (pre-calculation memory page selection); ▪ *rx_run_mem_sel* = '1' (running memory page selection);
KEY_DEC	▪ SESSION_START_REQ = '0'; ▪ Keep checking DEC_KEY_SEL value at MFAS = 252 until it is '0';
KEY_CHANGE	▪ *rx_key* = *rgt_rx_key_0* (change the key value at the corresponding output port); ▪ Pulse *rx_key_write*, triggering the key pre-calculation process;

State	Actions/Procedures
	▪ ACTIVE_KEY = '0'; ▪ PRECALC_SECOND_MEM = '1';
ACTIVE_SESSION	▪ SESSION_STATUS = '1', indicating that session is established (secure); ▪ Request de-assertion of the *ipy_ssf_rx* output port (it may keep asserted by a programmed consequent action or by the software layer); ▪ If MFAS [1:0] = 0 (start of a crypto packet): ▪ *rx_run_mem_sel* = MFAS[2] (running memory page selection); ▪ *rx_precalc_mem_sel* = ~MFAS[2] (pre-calculation memory page selection); ▪ If PRECALC_SECOND_MEM = '1': ▪ Pulse *rx_key_write*, triggering the key pre-calculation process; ▪ PRECALC_SECOND_MEM = '0'; ▪ If MFAS [7:0] = 252 (last crypto packet of a crypto block): ▪ Update key value in the corresponding output port: ▪ If DEC_KEY_SEL = '0': ▪ *rx_key* = *rgt_rx_key_0*; ▪ ACTIVE_KEY = '0 ▪ Else: ▪ *rx_key* = *rgt_rx_key_1*; ▪ ACTIVE_KEY = '1'; ▪ If the key has changed: ▪ Pulse *rx_key_write*, triggering the key pre-calculation process; ▪ If MFAS [1:0] = 3 (end of a crypto packet): ▪ If the key has changed: ▪ PRECALC_SECOND_MEM = '1';

Table 4.23 – RX processor session control state actions.

When the crypto session is not established, ACTIVE_KEY = 1, and the software layer must write the next (starting) key in *rx_key_0* register.

The SESSION_START_REQ flag is asserted only by a '0' → '1' transition in the SESSION_STATE bit of the *rx_session_state* register, indicating a crypto session start request from the software layer.

Figure 4.29 depicts the RX processor controller FSM state diagram.

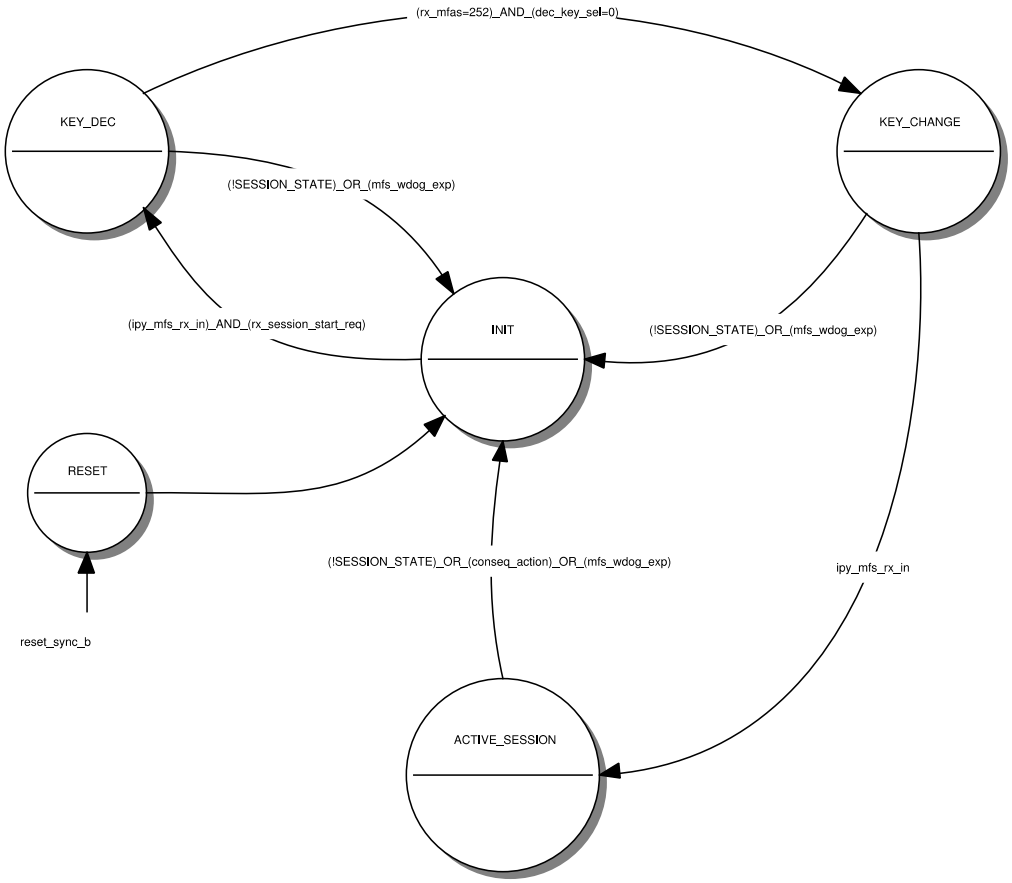


Figure 4.29 – RX processor controller finite state machine state diagram.

When the crypto session is inactive, after SESSION_START_REQ = 1 and an MFS pulse the RX processor controller keeps checking DEC_KEY_SEL at MFAS = 252 until it is '0'. Then, the *rx_key_write* signal is pulsed to trigger a key pre-calculation process in the AES-GCM sub-block, and a flag (PRECALC_SECOND_MEM) is set to request this pre-calculation for the second memory right in the beginning of the next crypto block.

AAD and IV write enable and pre-calculation functions are not commanded by the RX processor controller, but by the ODU OH Extractor sub-block instead.

When the crypto session is active, DEC_KEY_SEL is always evaluated at MFAS = 252. If a key change is detected, two key pre-calculation processes (for the first and second memories) are triggered at MFAS = 252 and 0, respectively.

4.3.11.3 REGISTER AND RESET CONTROLLER

The Register and Reset Controller sub-block, instantiated by the Control Engine, handles the interface for configuration register access, as well as the synchronization of the reset signal and the clock domain crossings.

The input reset signals, *ipg_hard_async_reset_b* and *ipg_soft_reset_async_b*, are synchronous with the *ipg_clk* clock signal and are used as a logic reset only on circuits belonging to this clock domain. Based on these signals, the Register and Reset Controller sub-block generates two new reset signals, *tx_reset_sync_b* and *rx_reset_sync_b*, synchronous with the Crypto4OTN internal global logic clock domains, *ipg_clk_sys_tx* and *ipg_clk_sys_rx*.

Additionally, it is also in charge of implementing the clock domain crossing of the signals read from the block internal logic. They are at the system clock domain (*ipg_clk_sys_tx* or *ipg_clk_sys_rx*) and are read by the IPS Bus at the *ipg_clk* domain.

This synchronization, performed by the Register and Reset Controller sub-block on all signals that cross different clock domains, is necessary to avoid the occurrence of metastability events [127]. A metastable state results from a violation of the setup and hold times of a flip-flop when a signal is sampled by the clock during its logic state transition. Its output may then show anomalies (glitches) or remain in an undetermined state for a certain time.

The probability of a flip-flop entering a metastable state and the time it takes to get out of it depends on the technology used in its manufacturing process, as well as on the operating conditions (e.g., voltage, temperature, and frequency). Typically, they return to a steady state after one or two clock periods [128].

A common way of mitigating metastability, implemented in the Register and Reset Controller sub-block, is the use of synchronization cells made up of two or more flip-flop stages, as illustrated in Figure 4.30 (a). The timing diagram shown in (b) depicts the clock (CLK), the asynchronous (S1), the metastable (S2), and the synchronized (S3) signals.

Through the flip-flop time parameters, as well as the frequencies involved, it is possible to statistically calculate the mean time between failures (MTBF) related to the occurrence of metastability events.

If necessary, other stages can be added to the synchronization cell for the cases where the metastable state resolution time lasts more than one clock cycle.

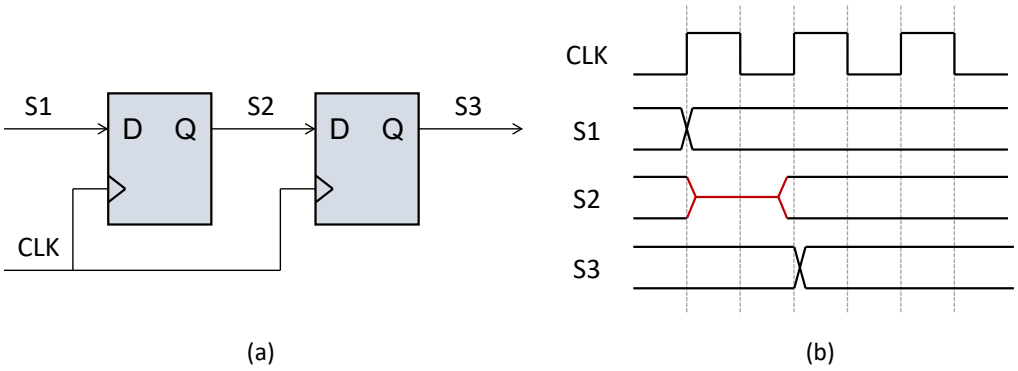


Figure 4.30 – Multistage synchronization cell (a) and corresponding timing diagram (b).

Chapter 5

DESIGN, VERIFICATION, AND SILICON PROTOTYPING

The previous chapter presented the hardware functional architecture of the 100G AES-GCM Cryptography Engine, developed upon a systemic architecture of a cryptographic solution conceived to allow for the establishment of secure communication links over OTN systems. This chapter describes its design, verification (through functional simulations), and silicon prototyping, integrated into a 100 Gbit/s OTN Processor built in the 40 nm technology. The results of laboratory tests with the developed ASSP prototypes are also presented.

5.1 CRYPTO4OTN BLOCK DESIGN AND TESTING

All the 100G AES-GCM Cryptography Engine functional logic blocks described in Section 4.3 were implemented within the Crypto4OTN top module, according to a digital integrated circuit design flow used by CPQD, described in Appendix B.

Throughout its conception and architectural design, the complete solution was partitioned into several sub-blocks and each one was later coded in a hardware description language (HDL), which provides syntactic and semantic support for modeling the temporal behavior and spatial structure of the hardware.

5.1.1 RTL DESIGN

From the architectural design, the Crypto4OTN top module and each of its sub-blocks were modeled in Verilog aiming at the subsequent logic synthesis, thus generating the respective RTL (register transfer level) designs.

Figure 5.1 shows a code excerpt corresponding to the Verilog modeling of a 16-stage Galois linear feedback shift register — the same used in the ODU OH Inserter and Extractor sub-blocks, as described in Section 4.3.7.1. The interested reader is referred to Appendix C for the entire RTL design code listing of the ODU OH Inserter sub-block.

```
1  always @(posedge ipg_clk_sys, negedge reset_sync_b)
2  begin : lfsr_seq_proc
3     if (!reset_sync_b) begin
4        lfsr   <= 16'd0;
5     end else begin
6        if (enable) begin
7           if (valid_in) begin
8              // 16-stage Galois LFSR
9              if (scrambler_ena) begin
10                if (mfs_in) begin
11                   lfsr <= 16'h5555;
12                end else begin
13                   for (i=15; i>0; i=i-1) begin
14                      if (fb_pol[i]) begin
15                         lfsr[i] <= lfsr[i-1] ^ lfsr[15];
16                      end else begin
17                         lfsr[i] <= lfsr[i-1];
18                      end
19                   end
20                   lfsr[0] <= lfsr[15];
21                end
22             end
23          end // if (valid_in)
24       end // if (enable)
25    end
26 end // lfsr_seq_proc
```

Figure 5.1 – 16-Stage Galois LFSR RTL design code excerpt.

Although the code conditional and loop statements (*if*, *else*, and *for*) look similar to their counterparts in other common programming languages, this is an RTL model that describes a hardware implementation. It means they are not translated into a set of executable machine instructions but, instead, they are interpreted by the logic synthesis tool and converted into a netlist of connected logic cells.

The *always* statement (line 1) creates a structured procedure which, in this case, is sensitive to the rising edge of the *ipg_clk_sys* clock signal, as well as to the falling edge of the *reset_sync_b* signal, thus creating a sequential logic circuit.

The *lfsr* signal in line 4 is declared at the beginning of the ODU OH Inserter Verilog module as a 16-bit register (16 flip-flops). The *if* statement in line 3 ensures the reset of these flip-flops when the *reset_sync_b* signal is asserted. Other *if* statements in lines 6, 7, 9, and 10 are synthesized as multiplexers controlled by the respective signals inside their conditional expressions (*enable*, *valid_in*, *scrambler_ena*, and *mfs_in*).

The linear feedback shift register is actually created by the *for* loop statement in line 13, with the flip-flop cascade and feedback connections being selected by a set of multiplexers generated by the repetition of the *if* statement in line 14 (inside the loop), which are controlled by the feedback polynomial coefficients stored at the *fb_pol* register.

Figure 5.5 shows the schematic view of this code, generated by the ModelSim® [129] simulation tool. Highlighted sections correspond to *if* multiplexers (1–4 and 6–10), LFSR XOR gates (5), and the shift register flip-flops (11).

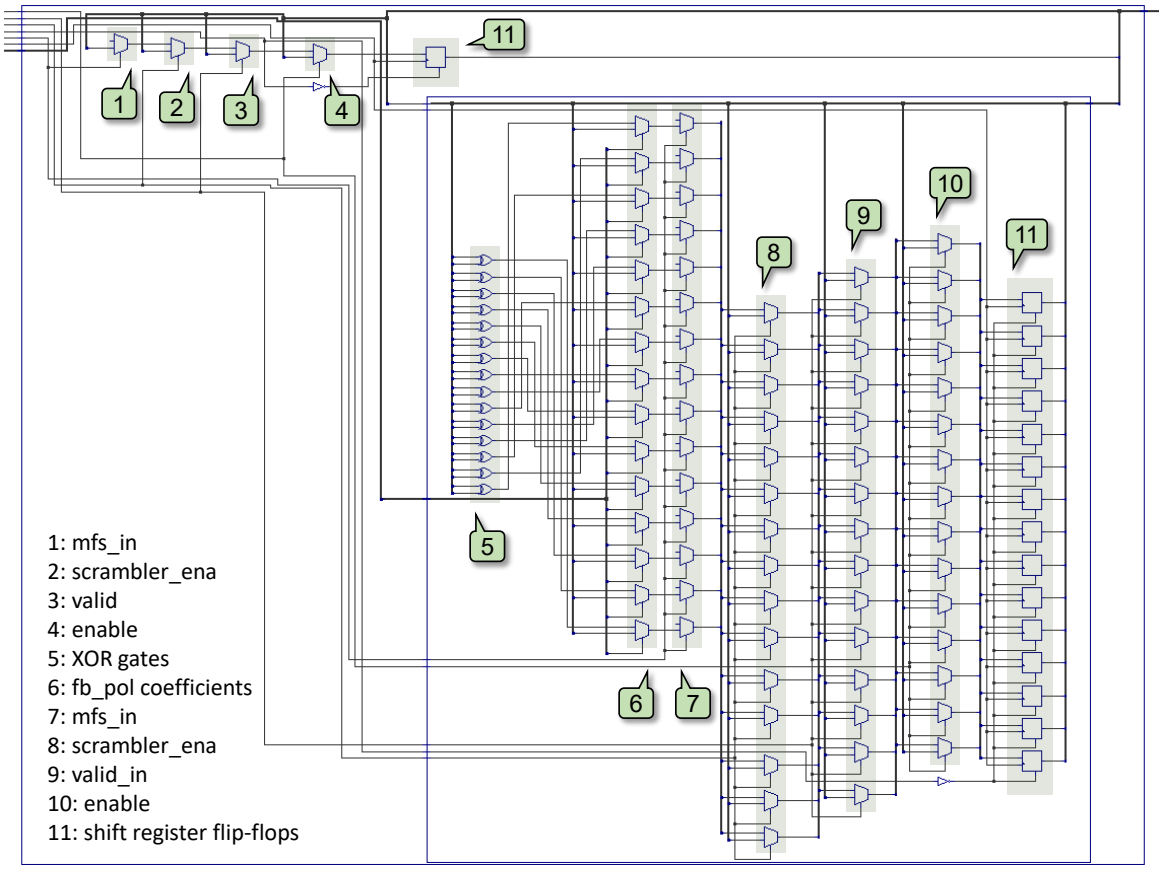


Figure 5.2 – 16-Stage Galois LFSR schematic view.

Further optimizations are later performed by the synthesis tool. For instance, the *fb_pol* coefficient multiplexers (6) are certainly replaced by hardwired logic connections.

Magnified views of the top-left corner multiplexers and some shift register flip-flops are shown in Figure 5.3 and Figure 5.4.

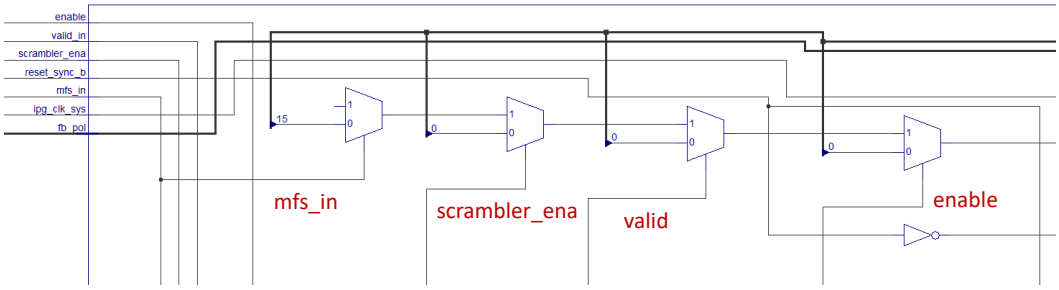


Figure 5.3 – Magnified view of some multiplexers from the 16-stage Galois LFSR schematic.

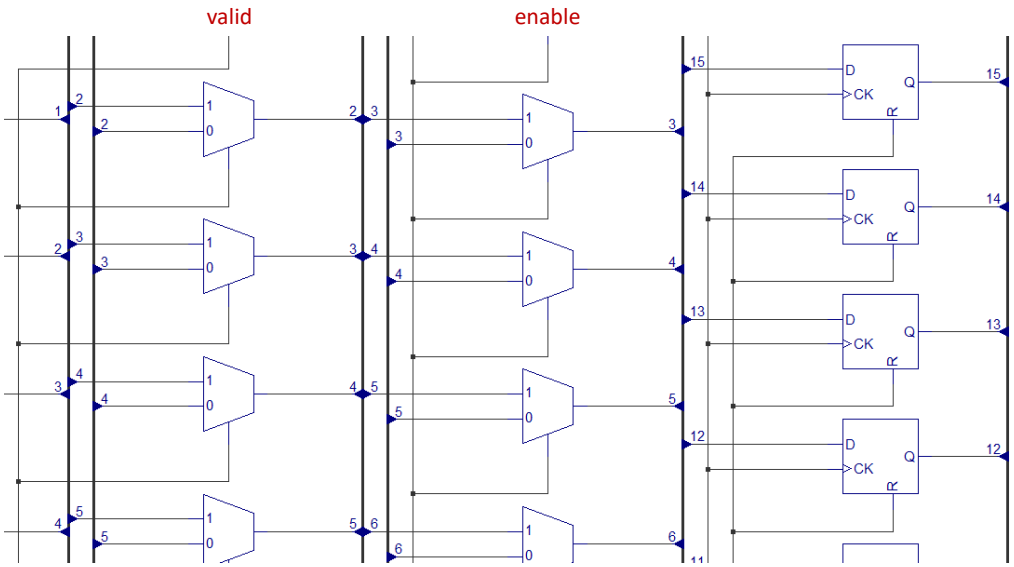


Figure 5.4 – Magnified view of some shift register flip-flops from the 16-stage Galois LFSR schematic.

There are commonly multiple ways of modeling a specific logic circuit using an HDL like Verilog. Figure 5.5 and Figure 5.6 show different code excerpts corresponding to a shift-register-based delay line — the same used in the Crypto Engine Control sub-block, as described in Section 4.3.6.4, to generate the retimed versions of the *mfs_in* and *fs_in* input signals. Both use parameters (SR_DW = 2 and DLY_CYCLES = 15), which were previously defined by a *localparam* directive.

The first one uses a *generate* statement (line 29) with a loop construct (line 30) to instantiate multiple copies of a flip-flop register (line 37) that samples data coming from the previous instance. This creates the cascade connection for the shift register.

The assignment statements in lines 19 and 20 create the connections between the *mfs_in* and *fs_in* input signals and the inputs of the two shift registers denoted by SR[0] and SR[1].

A multiplexer created by the *if* statement in line 9 and controlled by the *op_mode* input signal connects the *mfs_retimed* and *fs_retimed* output signals to the shift register output (lines 12 and 13) or directly to their corresponding input signals (lines 16 and 17), thus creating the necessary delays depending on the selected operation mode.

```
1   always @(posedge ipg_clk_sys, negedge reset_sync_b)
2   begin : fs_mfs_retimed_seq_proc
3      if (!reset_sync_b) begin
4         fs_retimed  <= 1'b0;
5         mfs_retimed <= 1'b0;
6         sr[0]       <= {SR_DW{1'b0}};
7      end else begin
8         if (enable) begin
9            if ((op_mode == 2'b00) || (op_mode == 2'b10)) begin
10           // If operation mode is "authenticated encryption"
11           // or "encryption-only".
12              fs_retimed  <= sr[DLY_CYCLES - 1][1];
13              mfs_retimed <= sr[DLY_CYCLES - 1][0];
14           end else begin
15           // If operation mode is "authentication-only".
16              fs_retimed  <= fs_in & valid_in;
17              mfs_retimed <= mfs_in & valid_in;
18           end
19           sr[0][1] <= fs_in & valid_in;
20           sr[0][0] <= mfs_in & valid_in;
21        end else begin
22           fs_retimed  <= 1'b0;
23           mfs_retimed <= 1'b0;
24        end // if (enable)
25     end
26  end // fs_mfs_retimed_seq_proc
27  // SR-based delay line
28  genvar i;
29  generate
30     for (i = 1; i < DLY_CYCLES; i = i + 1) begin : SR
31        always @(posedge ipg_clk_sys, negedge reset_sync_b)
32        begin : sr_proc
33           if (!reset_sync_b) begin
34              sr[i] <= {SR_DW{1'b0}};
35           end else begin
36              if (enable) begin
37                 sr[i] <= sr[i-1];
38              end // if (enable)
39           end
40        end // sr_proc
41     end
42  endgenerate
```

Figure 5.5 – Shift-register-based delay line modeled with generate statements.

In the second code, the shift registers are created by the use of the left shift operator (<<) in the assignment statements in lines 20 and 21.

```
1   always @(posedge ipg_clk_sys, negedge reset_sync_b)
2   begin : fs_mfs_retimed_seq_proc
3      if (!reset_sync_b) begin
4         fs_retimed      <= 1'b0;
5         mfs_retimed     <= 1'b0;
6         sr[0]           <= {DLY_CYCLES{1'b0}};
7         sr[1]           <= {DLY_CYCLES{1'b0}};
8      end else begin
9         if (enable) begin
10           if ((op_mode == 2'b00) || (op_mode == 2'b10)) begin
11           // If operation mode is "authenticated encryption"
12           // or "encryption-only".
13              fs_retimed  <= sr[1][DLY_CYCLES - 1];
14              mfs_retimed <= sr[0][DLY_CYCLES - 1];
15           end else begin
16           // If operation mode is "authentication-only".
17              fs_retimed  <= fs_in & valid_in;
18              mfs_retimed <= mfs_in & valid_in;
19           end
20           sr[1] <= sr[1] << 1;
21           sr[0] <= sr[0] << 1;
22           sr[1][0] <= fs_in & valid_in;
23           sr[0][0] <= mfs_in & valid_in;
24        end else begin
25           fs_retimed  <= 1'b0;
26           mfs_retimed <= 1'b0;
27        end // if (enable)
28     end
29  end // fs_mfs_retimed_seq_proc
```

Figure 5.6 – Shift-register-based delay line modeled with the left shift operator (<<).

5.1.2 Test Bench Design

Functional verification of the cryptographic solution during the RTL design phase was done with standard simulation approaches based on a regular test bench module, as depicted in Figure 5.7. The Cryto4OTN block was instantiated as the design under test (DUT) module, and appropriate stimuli were applied in the client side interface. The output data in the line side interface were fed back to the RX processor by a loopback connection.

Along with the reset and clock, all the necessary control signals were generated by the test bench driver in order to simulate the software layer commands and exercise the establishment and maintenance of crypto sessions. Padded ODU frames were generated with either fixed or random payloads, simulating OTN data produced by the associated blocks in an ASIC/ASSP processing chain.

All sub-blocks were verified and debugged by timing diagram analysis with waveform inspection over the entire design hierarchy. The expected DUT functionality was also

automatically verified by a test bench monitor, which checked both transmitted and received data.

In a similar approach, the AES-GCM sub-block was independently tested with the encryption and authentication algorithms being successfully verified against a golden model written in System Verilog, stimulated by test vectors from a validation suite standardized by NIST [130].

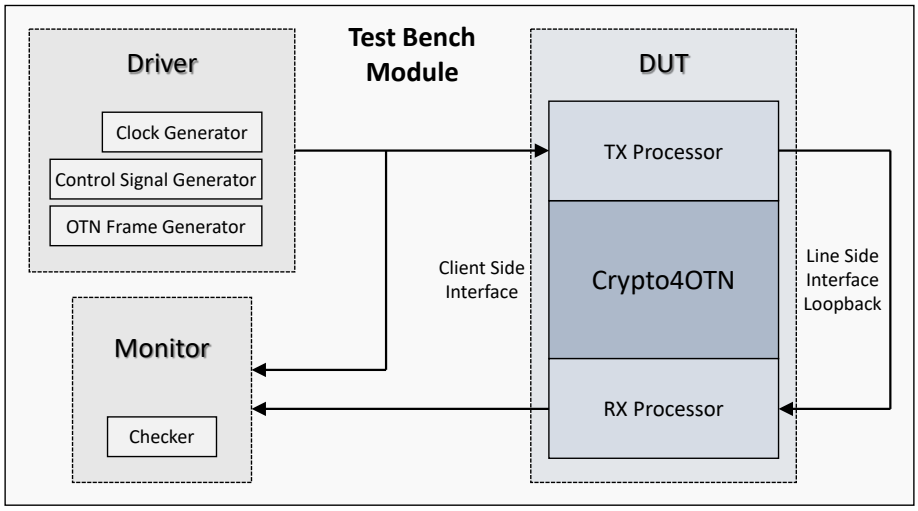


Figure 5.7 – Test bench module architecture used for functional verification of the Crypto4OTN block.

In order to facilitate waveform inspection during the debug phase, padded ODU frames were generated with fixed patterns as shown in Figure 5.8. In the beginning the first line, the OTN frame alignment signal (FAS) is followed by the multi-frame alignment signal (MFAS) counter (0x00 to 0xff).

Arbitrary hexadecimal values with different letters and numbers for each line were then added to the frame overhead and payload areas, allowing the quick location of each field during waveform inspection.

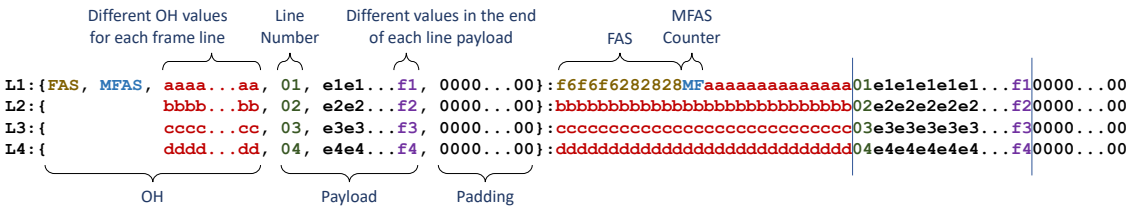


Figure 5.8 – Padded ODU frame structure with fixed patterns, as generated by the test bench driver.

5.2 FUNCTIONAL SIMULATIONS

The following sections bring some functional simulation results for the 100G AES-GCM Cryptography Engine solution, demonstrating the overall functionality of some sub-blocks and the complete integrated Crypto4OTN design. They are part of a long set of test cases used during the RTL design phase.

Simulations were done with the test bench module depicted in Figure 5.7, using padded ODU frames generated with fixed patterns as shown in Figure 5.8.

5.2.1 OPU CRYPTOGRAPHY ENGINE

As described in Section 4.3.6, this sub-block implements the core functionalities for OPU data encryption, decryption, and authentication.

In this simulation, which spans over three complete multi-frames, padded ODU frames generated by the test bench driver are fed into the client side interface of the Crypto4OTN block, pass through the Loopback Mux, and reach out to the OPU Cryptography Engine.

Verilog *force* statements in the test bench driver simulate and generate control signals at the required time instants, also configuring the sub-block with the following parameters:

```
key = {8'h31, 8'h30, 8'h29, 8'h28, 8'h27, 8'h26, 8'h25, 8'h24,
       8'h23, 8'h22, 8'h21, 8'h20, 8'h19, 8'h18, 8'h17, 8'h16,
       8'h15, 8'h14, 8'h13, 8'h12, 8'h11, 8'h10, 8'h09, 8'h08,
       8'h07, 8'h06, 8'h05, 8'h04, 8'h03, 8'h02, 8'h01, 8'h00};

aad = {8'h03, 8'h02, 8'h01, 8'h00};

iv  = {8'h11, 8'h10, 8'h09, 8'h08, 8'h07, 8'h06,
       8'h05, 8'h04, 8'h03, 8'h02, 8'h01, 8'h00};
```

Figure 5.9 shows simulation waveforms corresponding to the main signals of the external input and output interfaces of the OPU Cryptography Engine and its sub-blocks (OPU Drop, AES-GCM, OPU Add, and Crypto Engine Control).

The waveforms show the MFS and FS input synchronization pulses (1) and the fixed-pattern padded ODU frame lines (2–5). The corresponding output synchronization pulses (9) precede the start of the output frames with encrypted data (10), which make up the crypto packets. The TAG value (calculated for the previous crypto packet) is also output with the respective TAG write pulse (11).

Five control signals coming from the Control Engine sub-block are also shown: the pre-calculation and running memory selection (6), which switch over at every crypto packet, and the IV, AAD, and key write pulses (7–8).

The signal *opudrop_opu_valid* is de-asserted (logic '0') during the bus cycles 1, 144, and 241 (within the periodicity of 2 frames or 384 clock cycles), as described in Section 4.3.6.1 and illustrated in Figure 4.10. These gaps (12–14) happen due to the data segregation mechanism populating the 640-bit bus.

Finally, waveforms of the aggregated OH + encrypted OPU data (15) and some retimed control signals (16) are presented.

An additional view of the same signal waveforms is shown in Figure 5.10. Arrows indicate the bus cycles corresponding to the beginning and ending of the first line, as well as the beginning of the second, third, and fourth lines of a frame for both the clear input data and the encrypted output data, with the corresponding values presented in Figure 5.11.

Now the system clock signal (1) periods can be clearly seen, as well as the timing relationship between the padded ODU frame synchronization pulses (2, 3, 4, 13, and 14) along with the different sub-blocks.

The OPU data (5–8) and ODU overhead (9–12) dropped by the OPU Drop sub-block are distinguishable thanks to the fixed pattern structure used in the test bench driver frame generation.

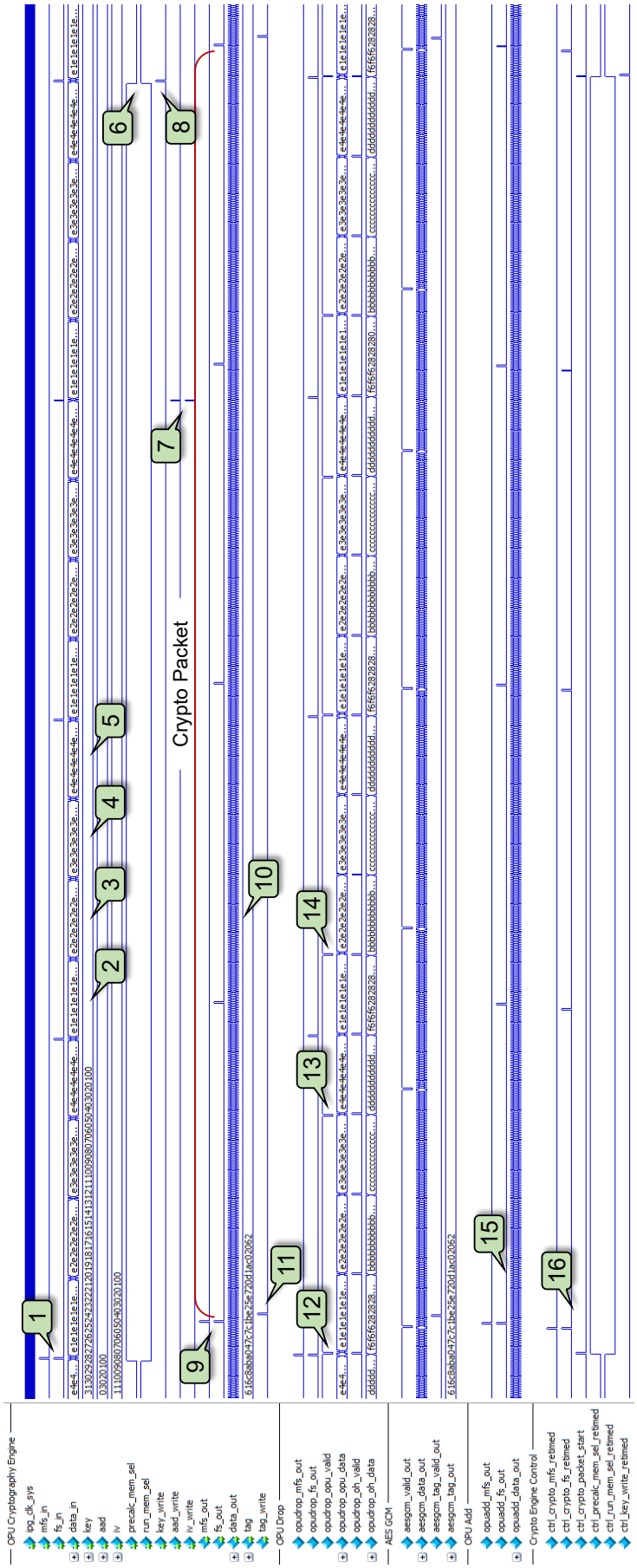
The *crypto_packet_start* (15) pulse is generated by the Crypto Engine Control sub-block after every four FS pulses, starting right after an MFS pulse that comes from the OPU Drop sub-block. Signals *precalc_mem_sel_retimed* and *run_mem_sel_retimed* are retimed versions (16) of the *precalc_mem_sel* and *run_mem_sel* inputs, synchronized with the rising edge of the *crypto_packet_start* pulse. These three signals are fed into the AES-GCM sub-block to delineate the boundaries of the data blocks to be encrypted or decrypted and to switch over the memory banks of pre-calculated and running cryptographic parameters.

Figure 5.12 brings a third view of the same signal waveforms with time delay information.

The input to output data processing delay (2) of 22 clock cycles correspond to the pipeline delays of 2 cycles in the OPU Drop (5), 16 cycles in the AES-GCM (data processing latency), 3 cycles in the OPU Add (8), and 1 cycle in the last output register (3). The Crypto Engine Control sub-block delays the data synchronization signals (11) for the OPU Add according to the AES-GCM data processing latency (10).

The *tag_write* output pulse is registered directly from the *aesgcm_tag_valid_out* signal with a delay of one clock cycle (6).

The retimed control signals (12) are generated right after an MFS pulse coming from the OPU Drop sub-block (7).



1: Input data synchronization signals
2: ODU frame clear data (1st line)
3: ODU frame clear data (2nd line)
4: ODU frame clear data (3rd line)
5: ODU frame clear data (4th line)
6: Pre-calculation and running memory selection
7: Input AAD and IV write pulses
8: Input key write pulse
9: Output data synchronization signals
10: ODU frame encrypted data
11: Output TAG write pulse
12: OPU Drop first data valid gap
13: OPU Drop second data valid gap
14: OPU Drop third data valid gap
15: Aggregated overhead + encrypted OPU data
16: Retimed control signals

Figure 5.9 – Simulation waveforms (I) for the OPU Cryptography Engine sub-block.

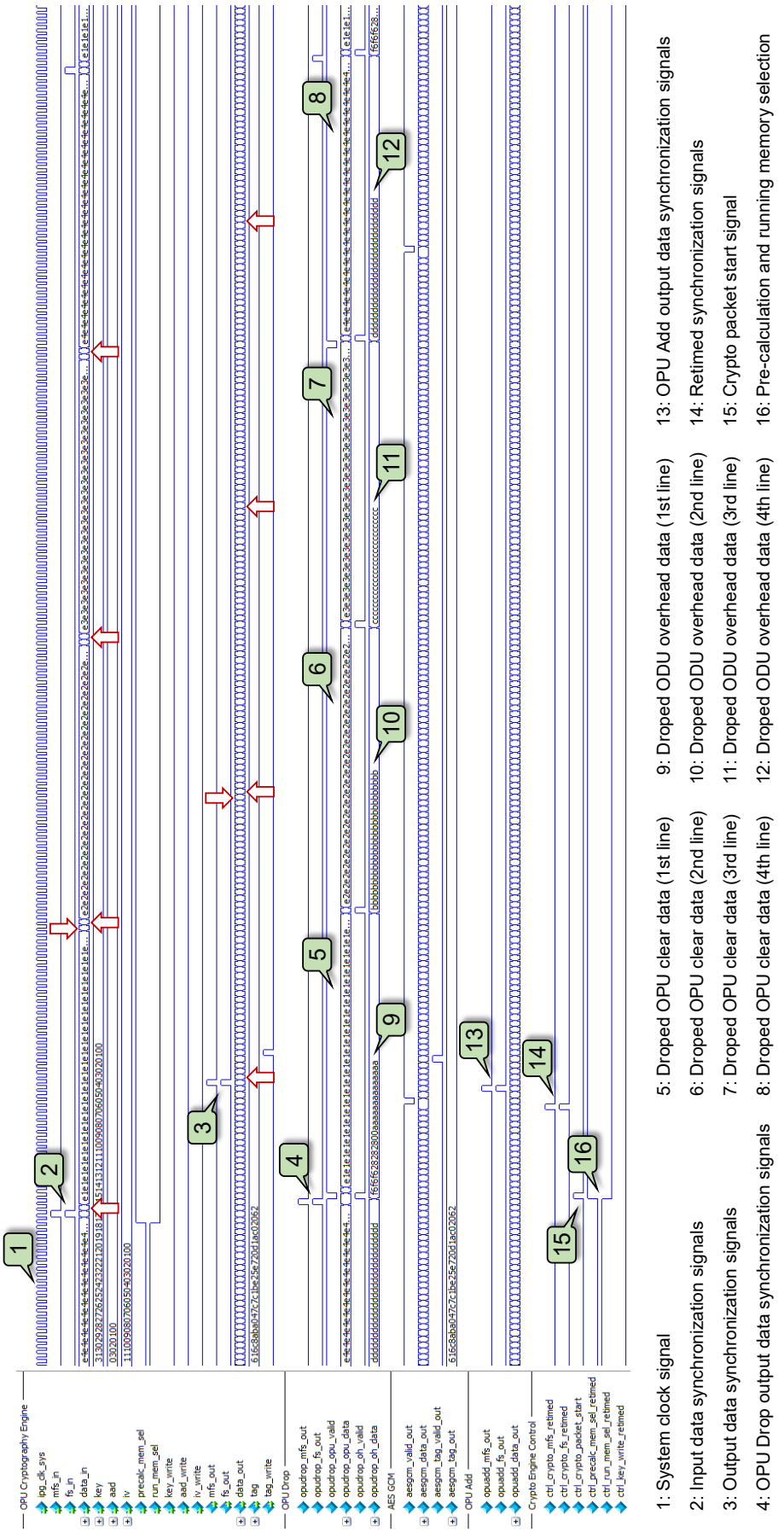


Figure 5.10 – Simulation waveforms (II) for the OPU Cryptography Engine sub-block.

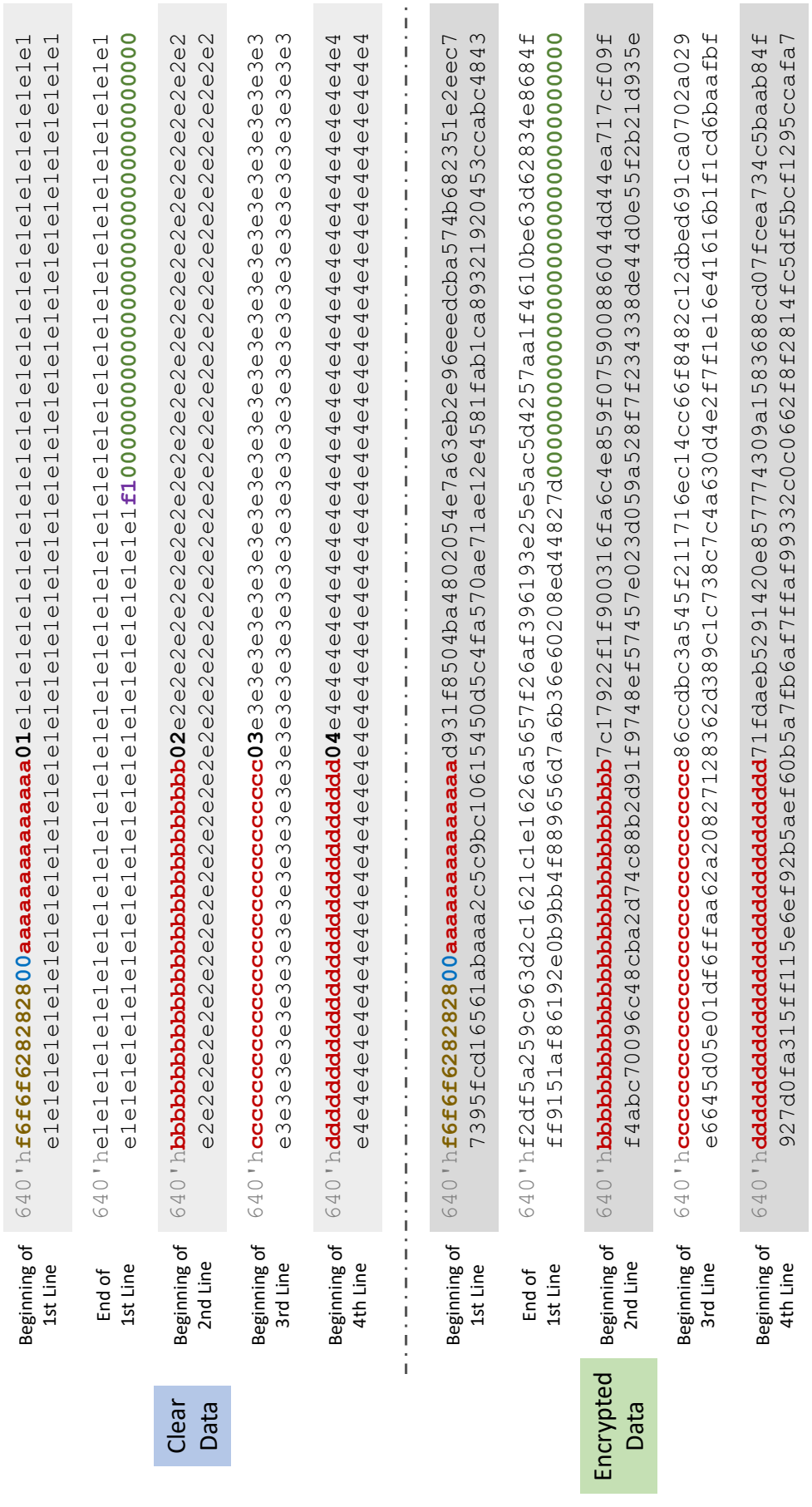


Figure 5.11 – Clear and encrypted data values corresponding to different lines of a frame.

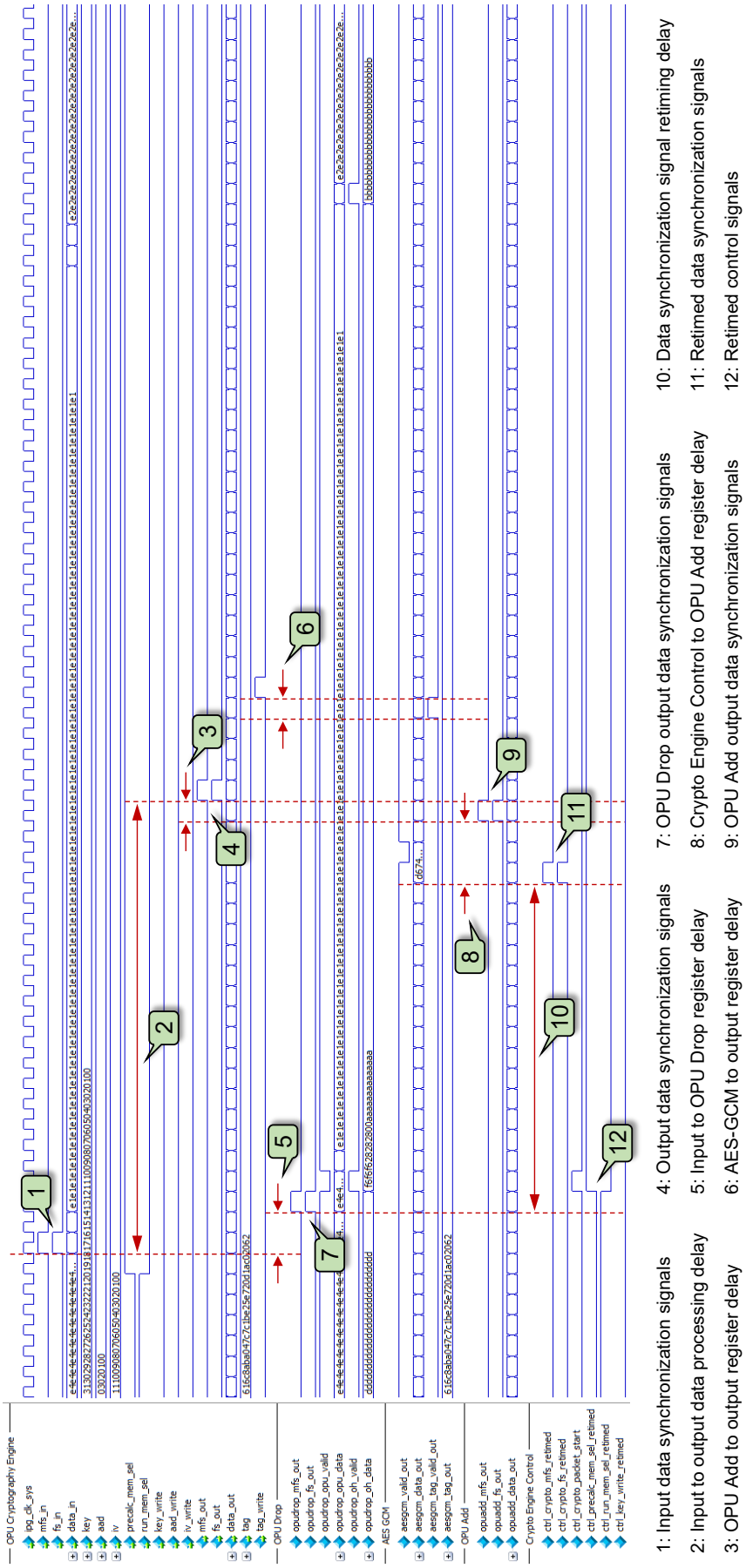


Figure 5.12 – Simulation waveforms (III) for the OPU Cryptography Engine sub-block.

5.2.2 ODU OH INSERTER AND EXTRACTOR

These sub-blocks handle data insertion and extraction into/from the ODU overhead, as described in Sections 4.3.7 and 4.3.8.

This simulation spans over four complete multi-frames, with padded ODU frames generated by the test bench driver being again fed into the client side interface of the Crypto4OTN block, passing through the Loopback Mux and the OPU Cryptography Engine, and reaching out to the ODU OH Inserter. Data then pass through the Replacement Signal Generator and Data Path Flow Control sub-blocks. The Crypto4OTN TX processor line side output interface is looped back into the RX line side input. Consequently, data pass through the Loopback Mux and, finally, reach out to the ODU OH Extractor.

To speed up the simulations, the AES-GCM sub-block (inside the OPU Cryptography Engine) was replaced by a delay-only model, which provides the same latency but does not modify the incoming data.

Verilog *force* statements in the test bench driver simulate and generate control signals at the required time instants, also configuring the sub-blocks with the following parameters:

```
tag = 128'ha1a2a3a4a5a6a7a8b1b2b3b4b5b6b7b8;
aad =  32'hc1c2d1d2;
iv  =  96'he1e2e3e4e5e6f1f2f3f4f5f6;
```

Figure 5.13 shows simulation waveforms corresponding to the main signals of the external input and output interfaces of the ODU OH Inserter and Extractor sub-blocks. The internal LFSR register signals are also presented.

The waveforms show the MFS and FS input (1, 4) and output (3, 5) synchronization pulses. TAG, AAD, and IV input bus values are captured and stored on the rising edge of the arbitrarily generated *tag_write* and *aad_iv_write* pulses (2), being later gradually inserted into the corresponding ODU overhead fields starting at the third frame of multi-frame 1. Overhead extraction is also demonstrated, with data being gradually recovered (6–13) starting from the third frame.

Overhead data write enable pulses (14–16) are generated by the ODU OH Extractor sub-block when the extracted data are available to be captured by the Control Engine and the RX processor OPU Cryptography Engine sub-blocks.

Simulated frame data showing overhead insertion within a crypto packet boundary are shown in Figure 5.14. The beginning of the four lines of frames number 8, 9, 10, and 11 of multi-frame 1 are presented with TAG, AAD, and IV values inserted in their relative positions within the RES fields of the ODU overhead, as previously described in Section 3.4.4 and Figure 3.5.

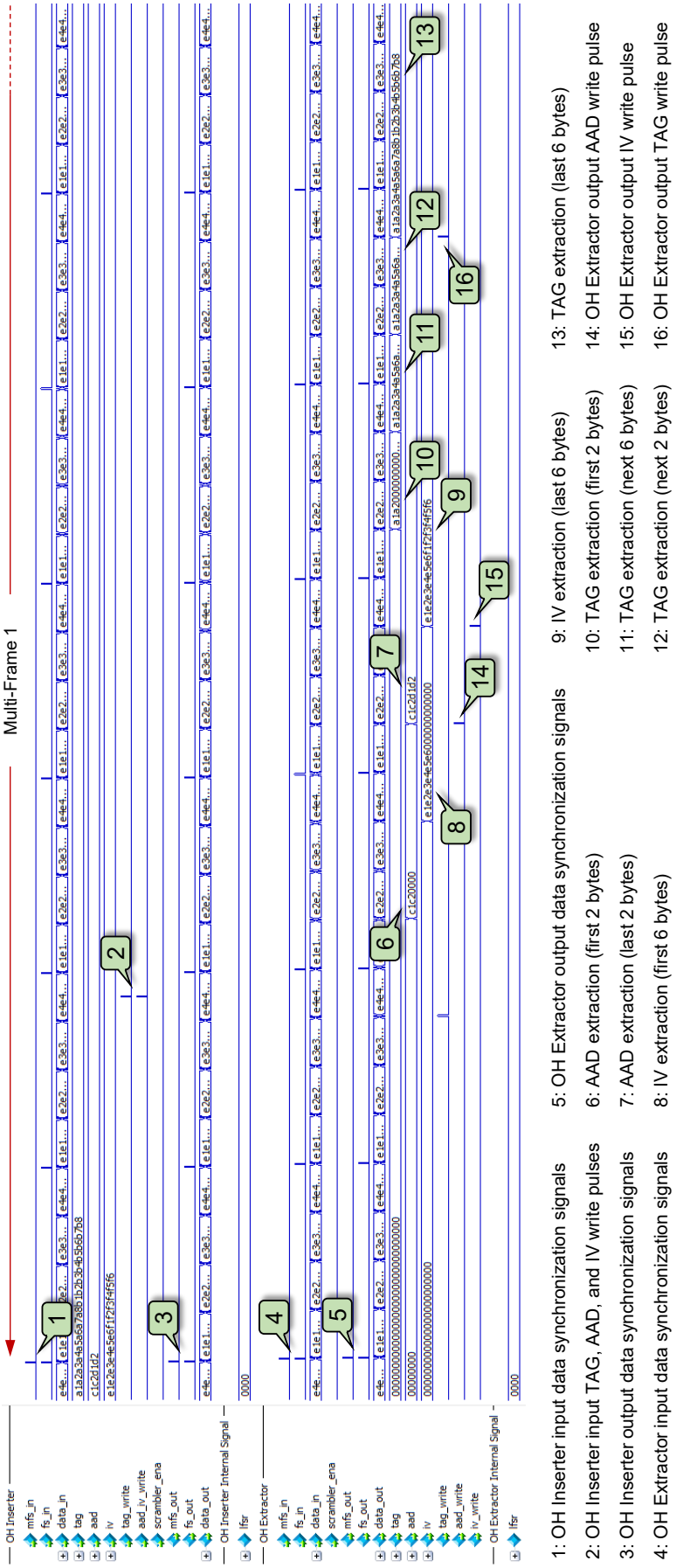
An additional view of the previous signal waveforms is shown in Figure 5.15. The ODU OH Inserter *scrambler_ena* control signal is asserted (2) in the end of multi-frame 1, causing the respective LFSR to resynchronize (4) right after the MFS input pulse (1). Inserted overhead data are then scrambled as described in Section 4.3.7.1 and Table 4.15. The corresponding ODU OH Extractor *scrambler_ena* control signal is asserted later (7), in the beginning of multi-frame 2, causing its LFSR to resynchronize (12) only after the next MFS input pulse (6).

In this way, extracted overhead data are correct (9) until the ODU OH Inserter scrambler is enabled. After that, they become scrambled (10) because the ODU OH Extractor LFSR is in a reset (zero) stuck state. Finally, they are correctly recovered (11) starting at multi-frame 3.

Figure 5.16 brings a third view of the same signal waveforms showing the expanded ODU OH Inserter 16-bit pseudo-random LFSR data (2) resynchronized right after the MFS input pulse (1). Extracted TAG data are then gradually scrambled (3–6) along with the first two frames of multi-frame 2, while the ODU OH Extractor LFSR remains in a reset state (7).

A detailed view of these signals at this simulation step is shown in Figure 5.17. The ODU OH Inserter *scrambler_ena* control signal is asserted (2) in the end of frame 254 of multi-frame 1. Its LFSR resynchronizes (3) right after the MFS input pulse (1). The corresponding ODU OH Extractor *scrambler_ena* control signal is asserted later (5), in the end of frame 2 of multi-frame 2. Correct extracted overhead data (6–8) then become scrambled (9–11) since the ODU OH Extractor LFSR remains in a reset stuck state (15).

Figure 5.18 brings a final view of these signal waveforms, showing the scrambled extracted overhead data (3) from multi-frame 2 being gradually correctly recovered (4–6) as the ODU OH Extractor LFSR resynchronizes (10) after the MFS input pulse (2) at the start of multi-frame 3.



1: OH Inserter input data synchronization signals
2: OH Inserter input TAG, AAD, and IV write pulses
3: OH Inserter output data synchronization signals
4: OH Extractor input data synchronization signals
5: OH Extractor output data synchronization signals
6: AAD extraction (first 2 bytes)
7: AAD extraction (last 2 bytes)
8: IV extraction (first 6 bytes)
9: IV extraction (last 6 bytes)
10: TAG extraction (first 2 bytes)
11: TAG extraction (next 6 bytes)
12: TAG extraction (next 2 bytes)
13: TAG extraction (last 6 bytes)
14: OH Extractor output AAD write pulse
15: OH Extractor output IV write pulse
16: OH Extractor output TAG write pulse

Figure 5.13 – Simulation waveforms (I) for the ODU OH Inserter and Extractor sub-blocks.

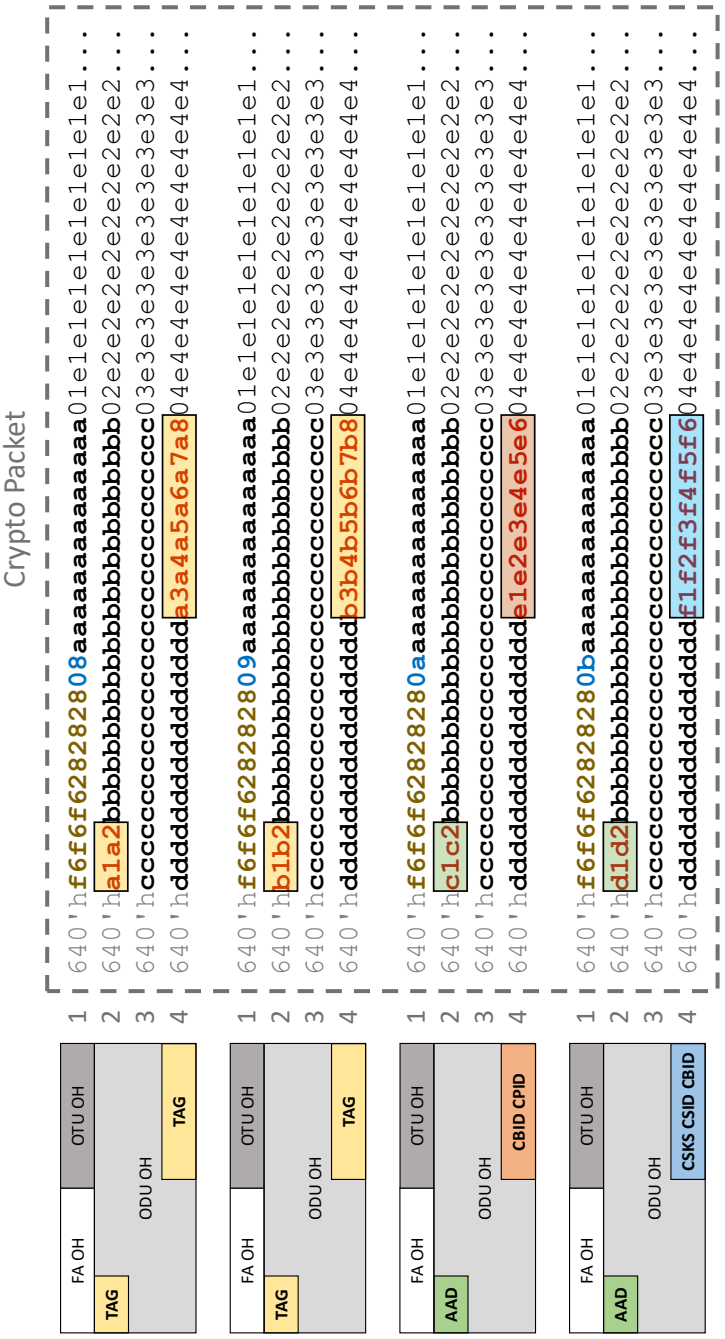
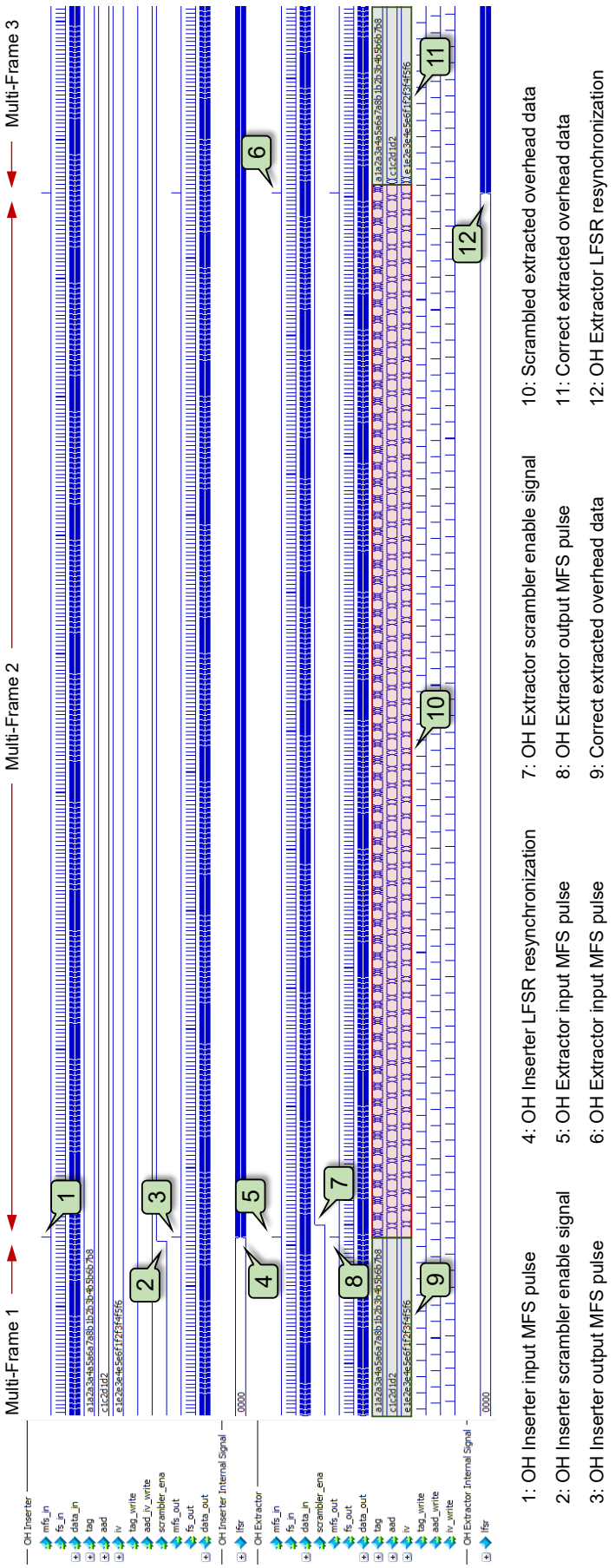


Figure 5.14 – Crypto packet simulated frame data showing overhead insertion.



1: OH Inserter input MFS pulse
2: OH Inserter scrambler enable signal
3: OH Inserter output MFS pulse
4: OH Inserter LFSR resynchronization
5: OH Extractor input MFS pulse
6: OH Extractor input MFS pulse
7: OH Extractor scrambler enable signal
8: OH Extractor output MFS pulse
9: Correct extracted overhead data
10: Scrambled extracted overhead data
11: Correct extracted overhead data
12: OH Extractor LFSR resynchronization

Figure 5.15 – Simulation waveforms (II) for the ODU OH Inserter and Extractor sub-blocks.

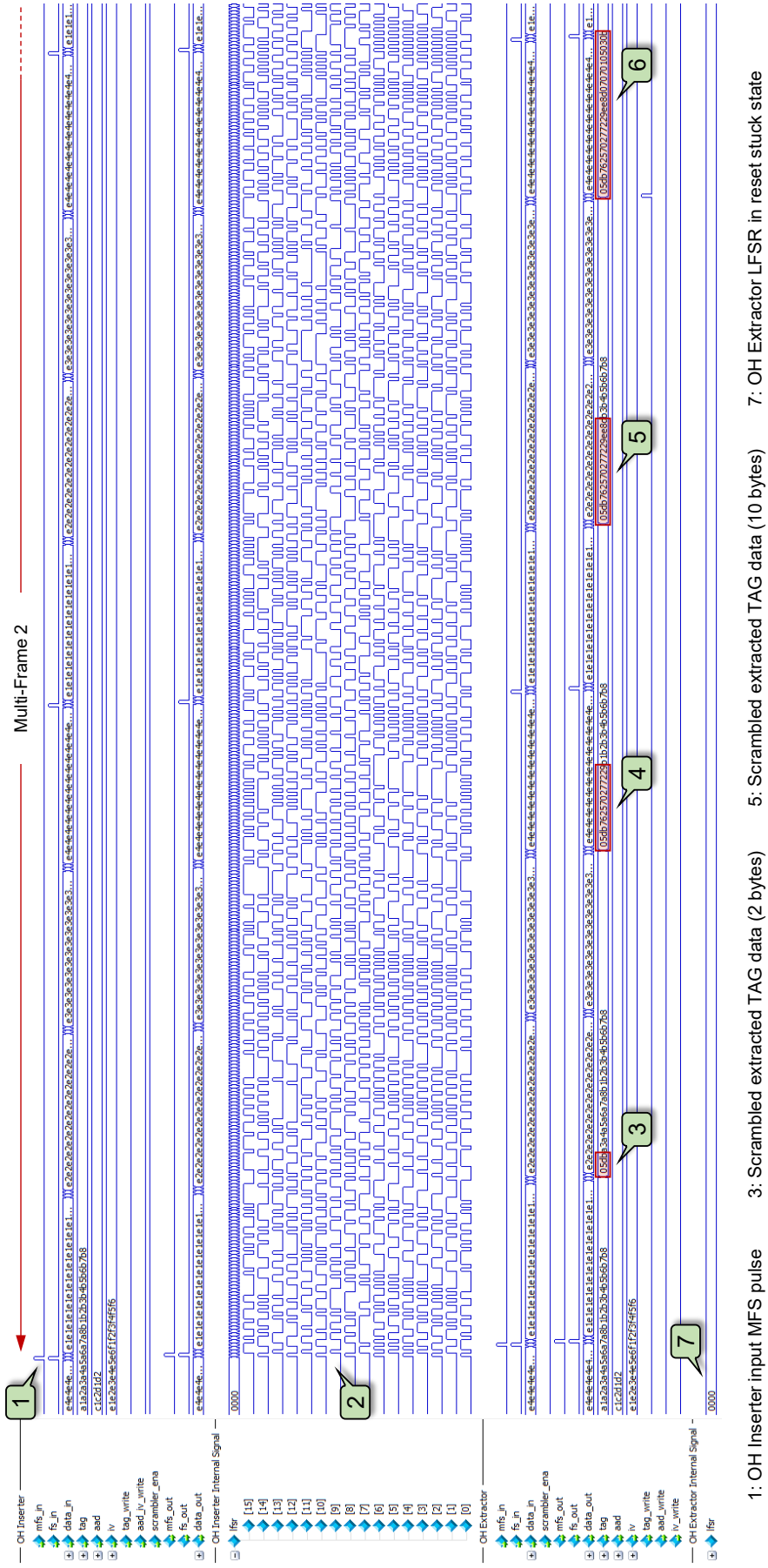


Figure 5.16 – Simulation waveforms (III) for the ODU OH Inserter and Extractor sub-blocks.

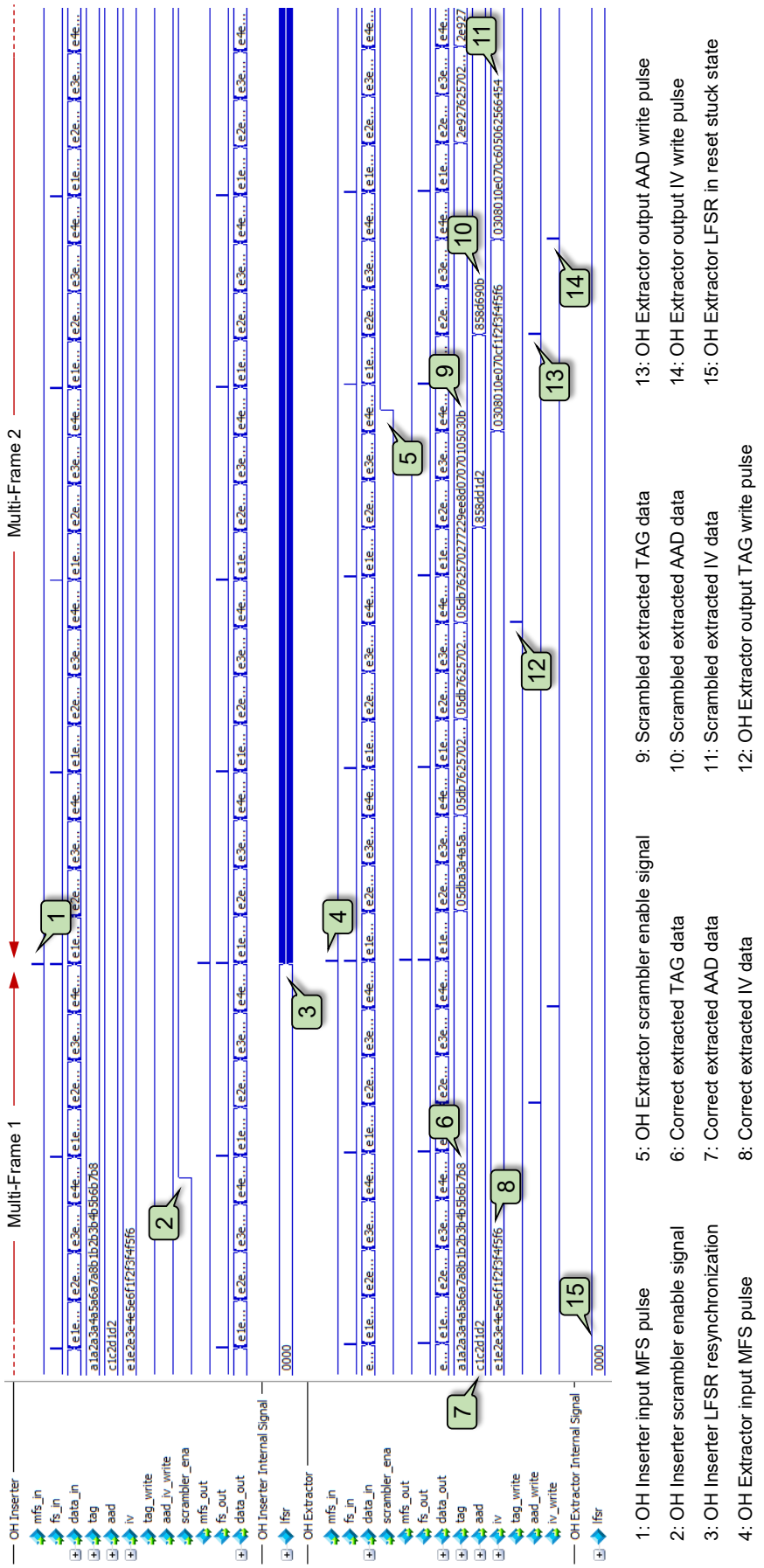


Figure 5.17 – Simulation waveforms (IV) for the ODU OH Inserter and Extractor sub-blocks.

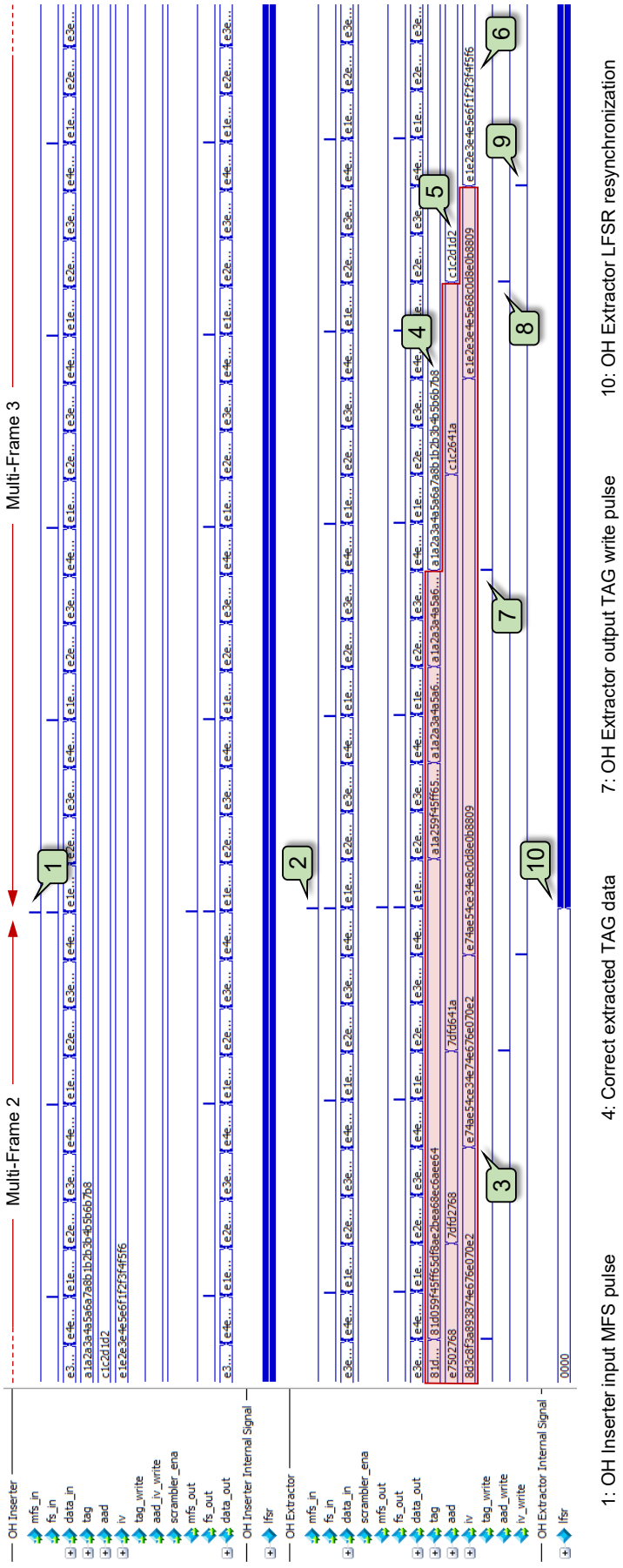


Figure 5.18 – Simulation waveforms (V) for the ODU OH Inserter and Extractor sub-blocks.

5.2.3 REPLACEMENT SIGNAL GENERATOR

The Replacement Signal Generator sub-block generates an AIS-like or user-defined pattern replacement signal in the output of the TX processor, as described in Section 4.3.9.

This simulation spans over two complete multi-frames, with padded ODU frames generated by the test bench driver being again fed into the client side interface of the Crypto4OTN block. After passing through the Loopback Mux, OPU Cryptography Engine, and ODU OH Inserter in the TX processor lineup, data reach out to the Replacement Signal Generator.

Verilog *force* statements in the test bench driver simulate and generate control signals at the required time instants.

Figure 5.19 shows simulation waveforms corresponding to the main signals of the external input and output interfaces of the Replacement Signal Generator sub-block.

The waveforms show the MFS and FS input and output synchronization pulses (1, 5). The *repl_sig_ena* control signal is asserted (2) right in the beginning of multi-frame 1, enabling the output signal replacement. The *repl_sig_sel* control signal is asserted (3) at the beginning of the following second frame, then selecting the user-defined pattern which changes to 0xcafe (4) also at the beginning of multi-frame 1.

The data output bus shows the original (6), AIS-like (7), and user-defined pattern (8) replaced data.

Simulated data are also presented in this same figure, showing the four lines of the output frame 255 (end of multi-frame 0) with the original data, frame 0 (start of multi-frame 1) with the AIS-like replaced data, and frame 1 with the user-defined pattern replaced data.

The RES fields of the ODU overhead are filled with zero because no overhead data were provided for the ODU OH Inserter sub-block in this simulation. The highlighted replaced data can be compared with Figure 4.20 and Figure 4.21.

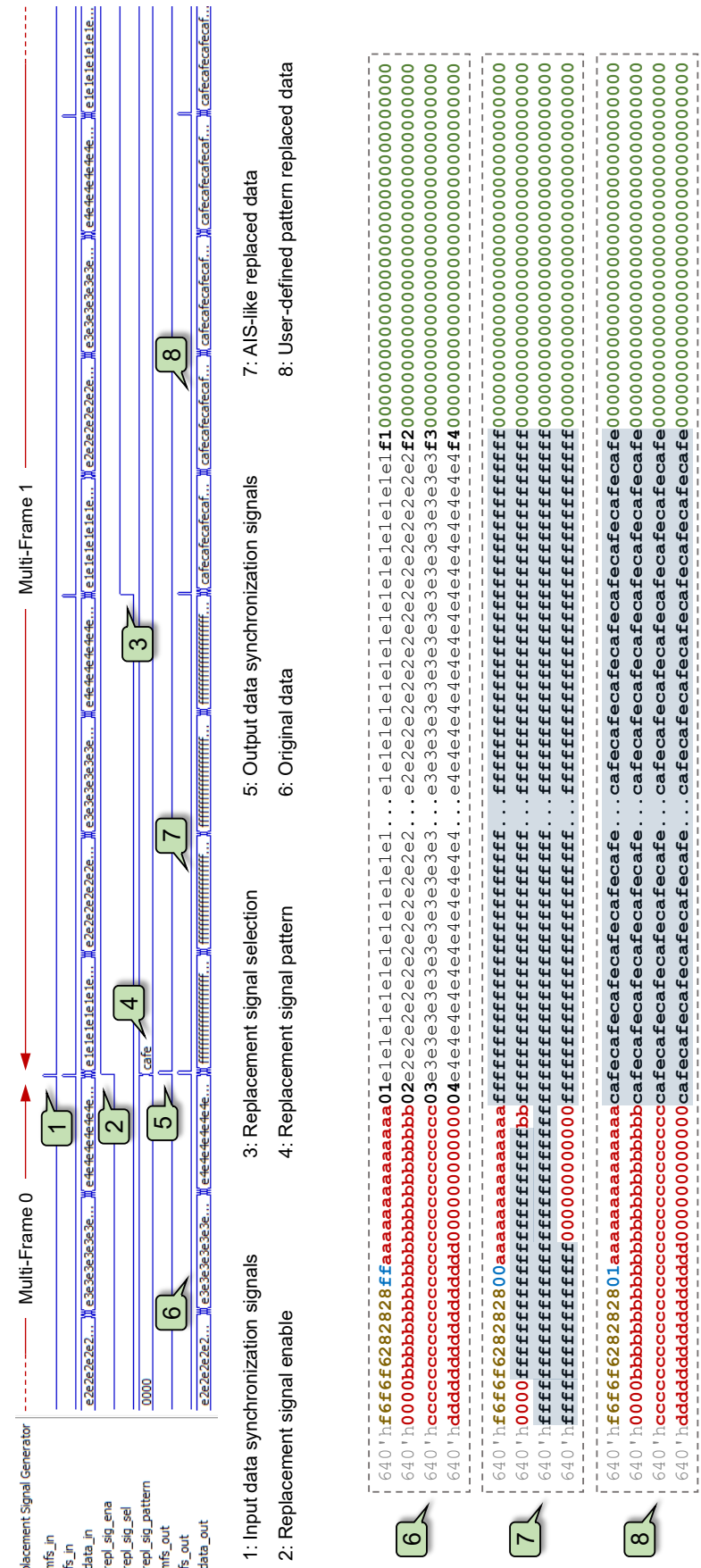


Figure 5.19 – Simulation waveforms and data for the Replacement Signal Generator sub-block.

5.2.4 CRYPTO4OTN

The following simulation results show the main functionalities of the complete integrated Crypto4OTN design. The overall simulation spans over five complete multi-frames, with padded ODU frames generated by the test bench driver being fed into the client side interface of the Crypto4OTN block and looped back in the line side.

Verilog *force* statements in the test bench driver simulate and generate control signals at the required time instants, also configuring the Control Engine sub-block with the following parameters:

```
tx_key_0     = 256'hbaba01aabb0000000000000...00000000000000000000000;
tx_key_1     = 256'hcafe02ccdd0000000000000...00000000000000000000000;
tx_iv_csid   =  32'h22aa22bb;
rx_key_0     = 256'hbaba01aabb0000000000000...00000000000000000000000;
rx_key_1     = 256'hcafe02ccdd0000000000000...00000000000000000000000;
```

Figure 5.20 shows simulation waveforms corresponding to the main signals of the external input and output interfaces of the Crypto4OTN block, on the client and line sides, as well as some internal signals of the Control Engine sub-block.

The waveforms show the MFS input and output synchronization pulses (1, 3, and 5), corresponding to the client and line side input data (2, 4) and client side output data (6). During the first multi-frame, the bit 0 of *rx_session_state* register is asserted (9), simulating a software layer request for crypto session establishment. The same is done some time later with the *tx_session_state* register (8), followed by a key change request (12). This crypto session establishment process is described in Section 3.6.1.

In multi-frame 2, both TX and RX session status bit (10, 11) indicate an active crypto session. Data tampering is also simulated, with authentication TAG mismatches being then indicated by the corresponding output signal (7).

Later, a second software layer TX key change request is also simulated (13) with both TX and RX keys being changed from ‘0’ (14, 15) to ‘1’ (16) in the last multi-frame.

A second view of the same waveforms is shown in Figure 5.21, with more details on the crypto session establishment process. Arrows over the *received_iv* data indicate the CSKS 8-bit code word.

The RX processor controller receives an IV sequence with CSKS = 0xff (4), transmitted by the TX processor for synchronization purposes. After the TX session start (6) and key change (12) requests, the TX processor controller waits for the next MFS pulse (start of multi-frame 1) and then transmits a new IV sequence with CSKS = 0x00. This new code word is received by the RX processor controller (5) and is decoded to '0' (indicating key 0) at MFAS = 252, as described in Table 4.22 of Section 4.3.11.2.

Crypto session is then established in both TX and RX processors after the next MFS pulse (start of multi-frame 2). Both *tx_session_status* and *rx_session_status* registers show 0x01 (15), indicating key '0' as the active one, as well as the session establishment.

The state transitions of the TX and RX processor controller finite state machines are indicated by *tx_ctrl_fsm_st* and *rx_ctrl_fsm_st* signals (8–11), according to the state diagrams shown in Figure 4.26 and Figure 4.29.

Figure 5.22 brings a detailed view of these waveforms, showing the data tampering simulation. Arrows indicate the locations in the 3rd, 5th, 8th, and 9th frames where the encrypted data looped back at the line side interface were tampered with by inserting (forcing) a 640-bit word in the second bus cycle of these frames.

A magnified view of the line side input data (*ipy_data_rx_in* signal) shows the FAS right after the FS pulse (6), with MFAS = 0x0a (7) indicating frame number 10 and the 640-bit tampered word (8) in the following cycle (0xfaca00000…00000).

A section of the client side output data (*ipy_data_rx_out* signal) is also presented in a magnified view, showing the tampered output data (10) in frames 8–11 replaced by zeros (authentication blocking enabled) and the corresponding authentication TAG matching indication (11).

Figure 5.23 shows the correspondence between the TX and RX calculated TAG mismatches and the authentication TAG matching indication, such as in (6) and (4).

The simulation results for data encryption are shown in Figure 5.24. The line side data are replaced by an AIS-like pattern (5) while the TX crypto session is not established. OPU encrypted data (4) are then available after a 26-cycle TX processing (pipeline) delay (6).

Simulated data are also presented in this same figure, showing the 640-bit clear (2) and encrypted (4) data from the beginning of the first frame of multi-frame 2. The OTN FAS (0xf6f6f6282828) can be clearly distinguished, followed by the MFAS counter starting at 0x00 right after the MFS pulse.

Figure 5.25 shows the client side output decrypted (clear) data (7) being available after a 6-frame delay (5) from the line side input encrypted data (4), corresponding to the store-and-forward delay created by the Authentication Buffer sub-block to wait for the evaluation of the TAG matching condition, as described in Section 4.3.10 and Figure 4.23.

Again, simulated data are also presented, showing the 640-bit clear (2), encrypted (4), and decrypted (7) data from the beginning of the first frame of multi-frame 2.

Figure 5.26 brings another view of these waveforms, with simulation results for the hitless key change process. During the active crypto sessions in multi-frame 2, a software layer TX key change request is simulated (9). The TX processor controller waits for the next MFS pulse (start of multi-frame 3) and then transmits a new IV sequence with CSKS = 0xff (encoding the key '1').

The new code word is decoded to '1' by the RX processor controller at MFAS = 252 in multi-frame 3, and both TX and RX keys are hitless changed (10) at the new crypto session starting in multi-frame 4. Registers *tx_session_status* and *rx_session_status* show 0x03 (11), indicating the active key '1' and the session establishment.

A magnified *received_iv* signal waveform (5) is also presented, showing the CSID (12), the CBID (13), the CSKS = 0x00 at CPID = 0xc0 or 192 (15), and the CSKS changing to 0xff (14) at CPID = 0xc1 or 193 (16).

Finally, Figure 5.27 evidences the hitless key change process by showing the TX and RX calculated TAG matching conditions (5) right in the end of multi-frame 3 and in the beginning of multi-frame 4, leading to non-disrupted decrypted (clear) data (4) at the client side output.

The previous CBID (7) is restarted to zero (8) after the key change process, corresponding to the beginning of a new crypto session.

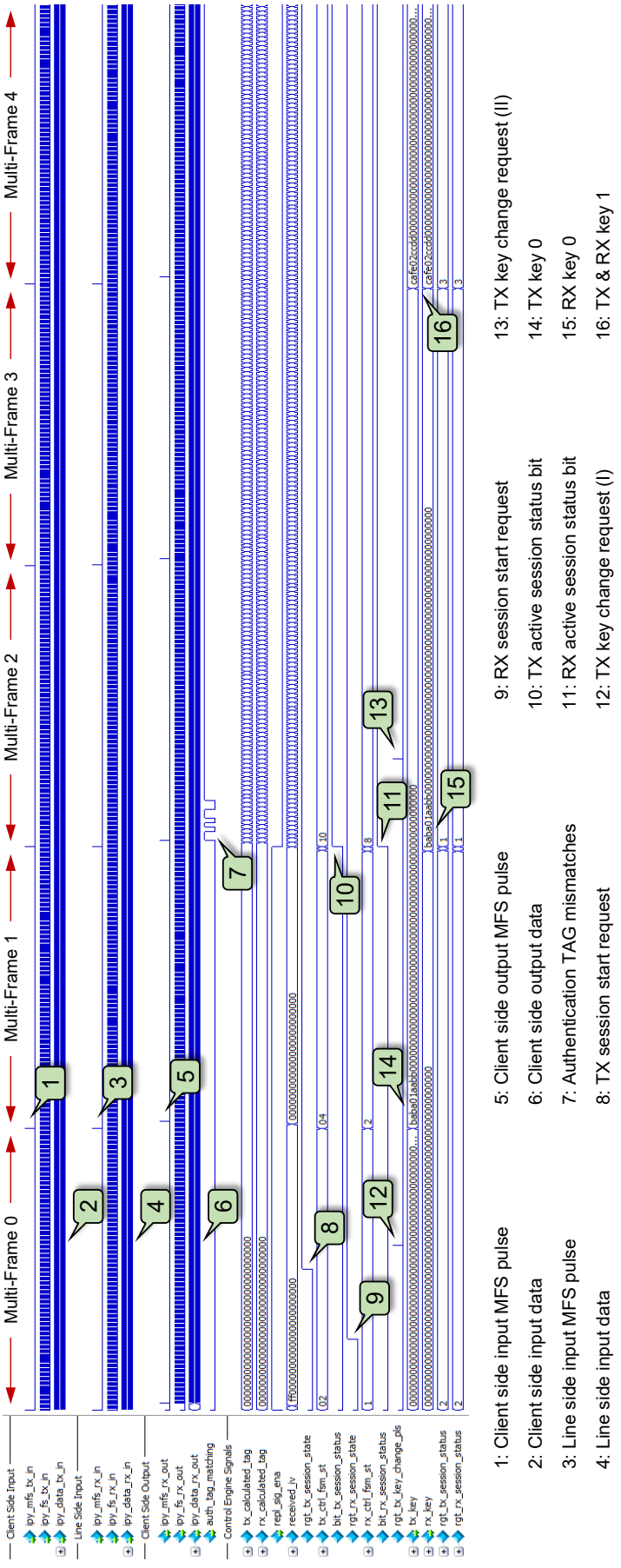


Figure 5.20 – Simulation waveforms (I) for the Crypto4OTN block.

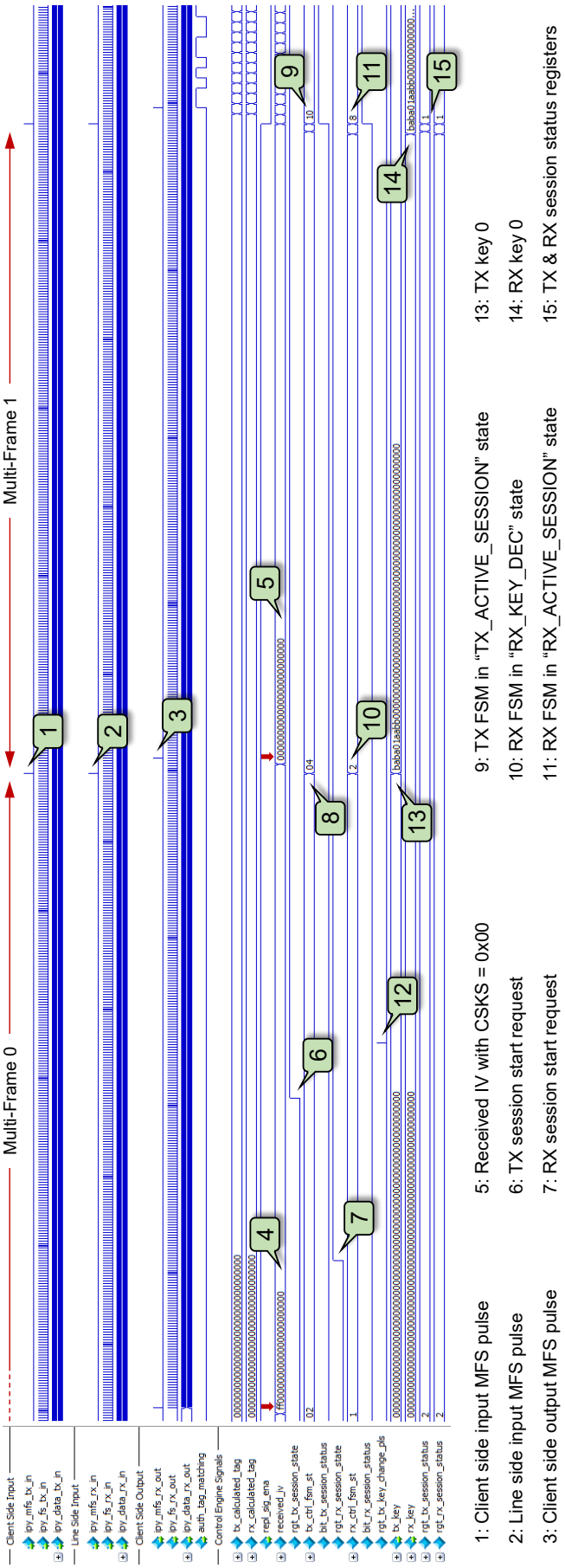


Figure 5.21 – Simulation waveforms (II) for the Crypto4OTN block.

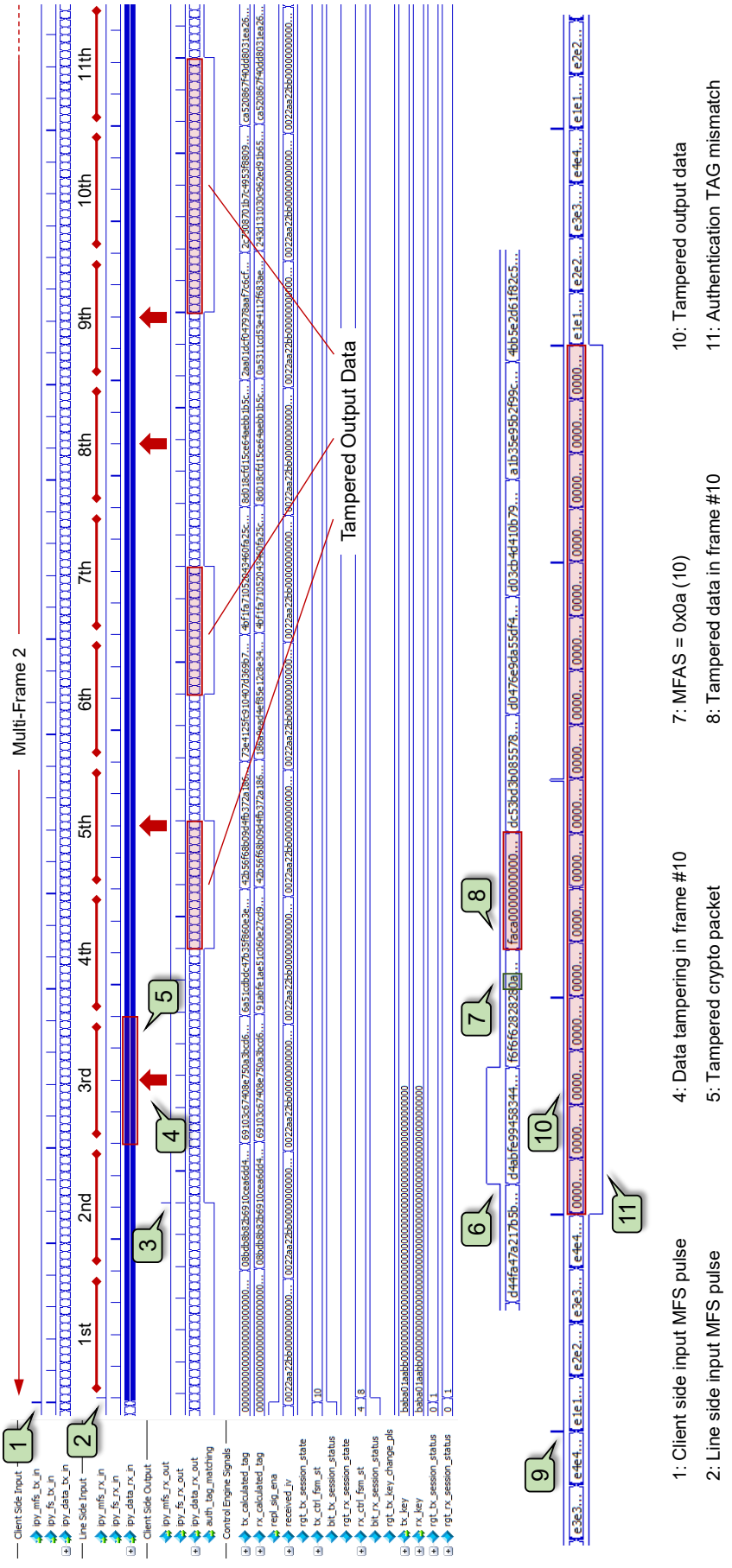


Figure 5.22 – Simulation waveforms (III) for the Crypto4OTN block.

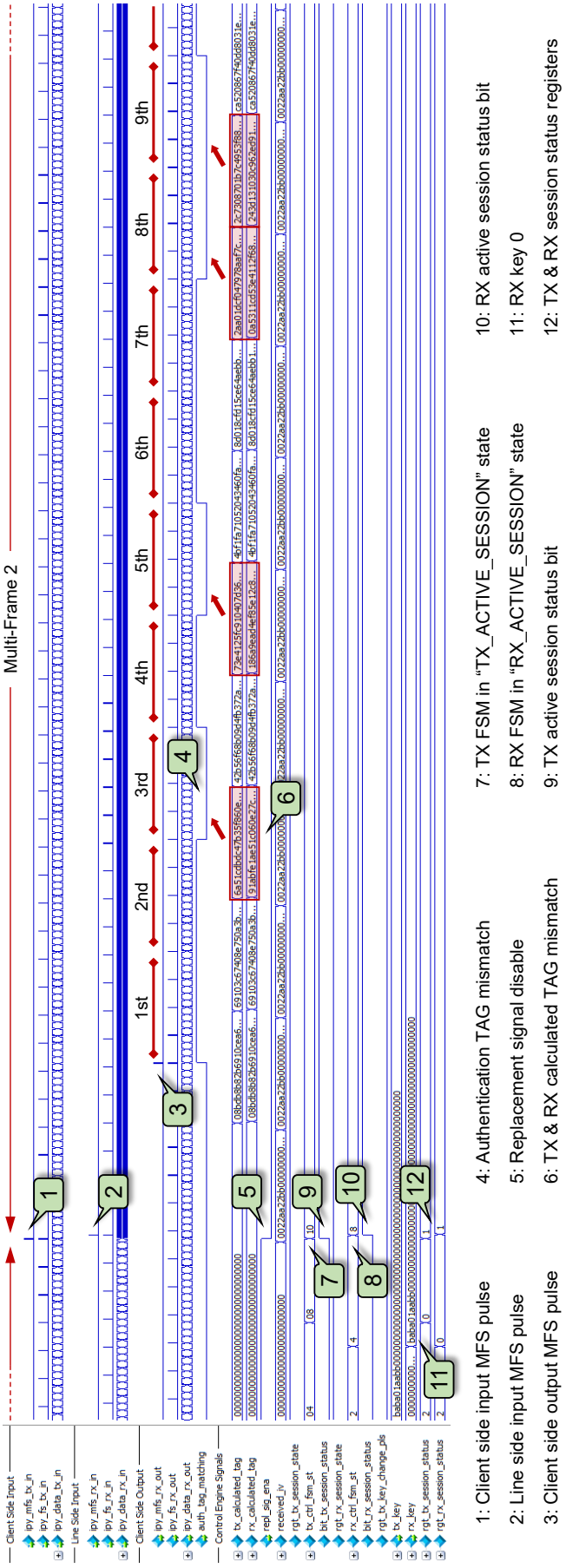


Figure 5.23 – Simulation waveforms (IV) for the Crypto4OTN block.

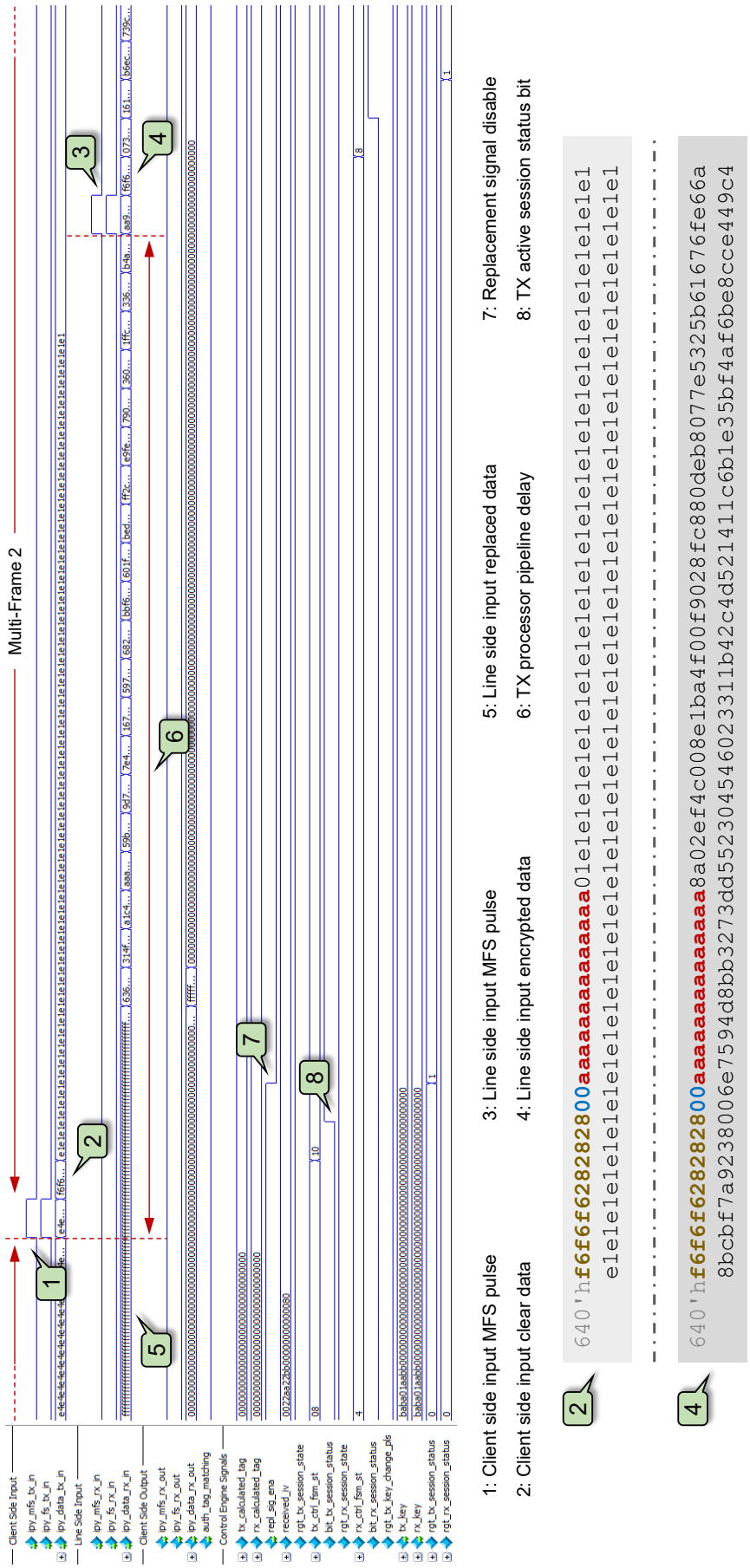


Figure 5.24 – Simulation waveforms (V) for the Crypto4OTN block.

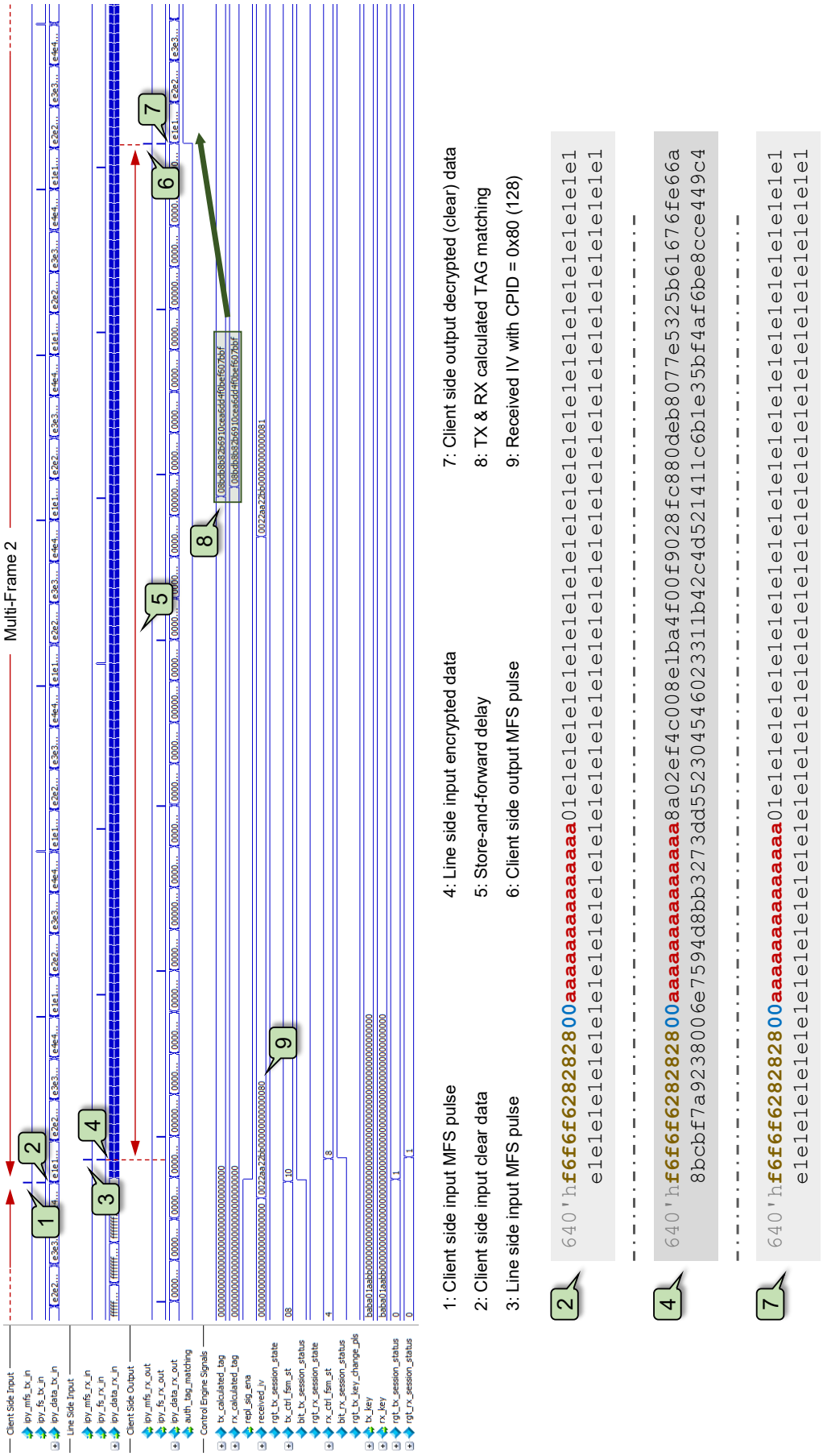


Figure 5.25 – Simulation waveforms (VI) for the Crypto4OTN block.

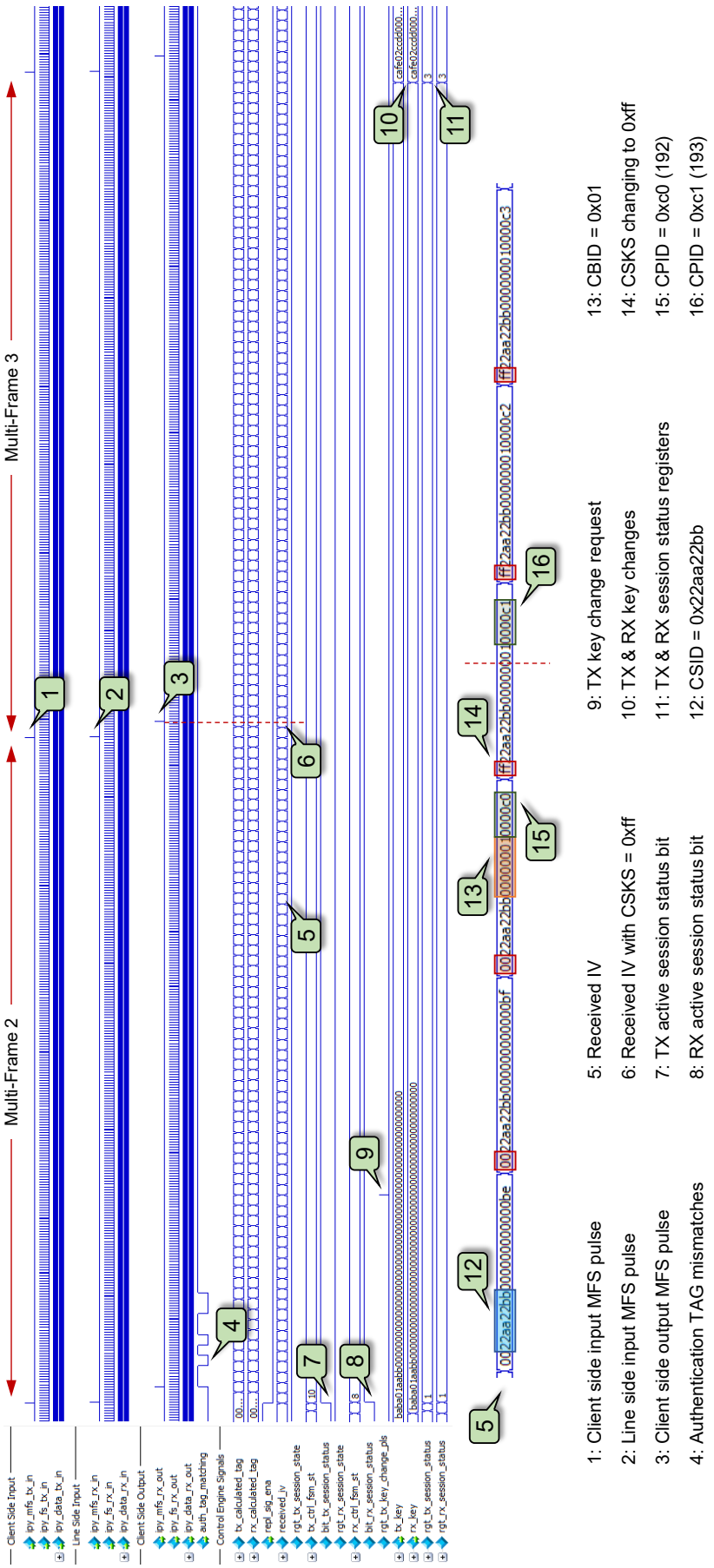


Figure 5.26 – Simulation waveforms (VII) for the Crypto4OTN block.

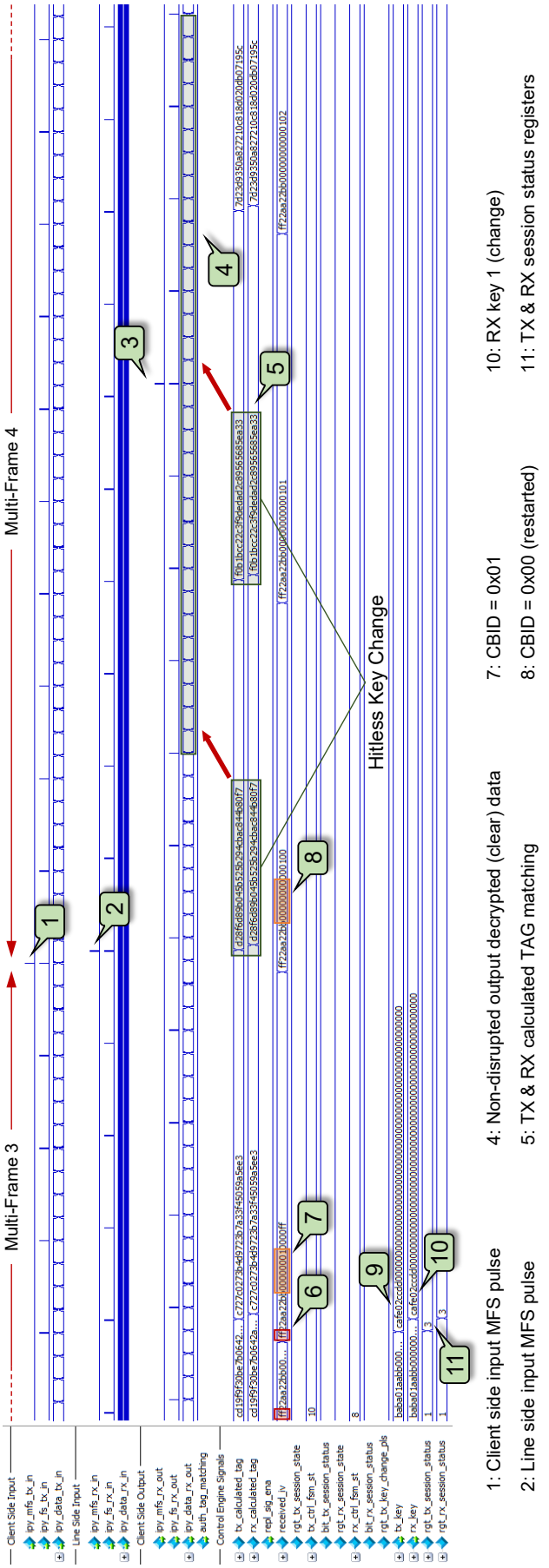


Figure 5.27 – Simulation waveforms (VIII) for the Crypto4OTN block.

5.3 100G OTN PROCESSOR

CPQD has leaded a challenging project aiming at the development of a 100 Gbit/s OTN processor ASSP device for the Brazilian telecom industry, featuring the 100G AES-GCM Cryptography Engine described in Chapter 4 to provide the establishment of secure communication links. In its first silicon release, the ASSP handles OTU4 signals only, making it suitable for regenerator applications or cryptography-enabled terminal transponders.

A hardware functional architecture block diagram is depicted in Figure 5.28, with the Crypto4OTN block (highlighted in the middle of the TX and RX processor lineups) performing encryption in one direction and decryption in the other.

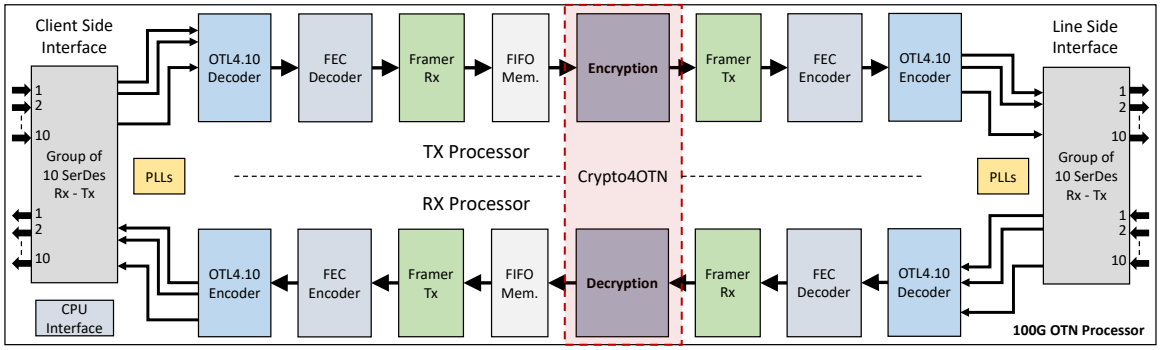


Figure 5.28 – CPQD's 100G OTN Processor ASSP hardware functional architecture block diagram.

In the TX processor client side interface, a 111.8 Gbit/s OTU4 signal is received in 10 lanes of 11.18 Gbit/s, each of which being handled by a SerDes (serializer/deserializer) device with a 64-bit input/output bus running at 174.70 MHz. Recovered data are first processed by an optical transport layer (OTL) IP block, featuring functions like lane deskew and reordering. OTN aligned frames follow then to the FEC decoder, where errors are detected and corrected using the redundancy information. In the sequence, the Framer block allows for overhead termination and monitoring, and a FIFO memory block provides clock domain separation.

At this point, data are ready to be forwarded to the line side, being processed by another Framer (for new overhead insertion), the FEC encoder (for redundancy calculation and insertion), and the OTL4.10 encoder (generating the multilane data stream). Alternatively, when establishing a secured OTN communication link, the Crypto4OTN engine performs its cryptography functions right in the middle of this processing chain.

The same functions are applied for the RX processor, with data being received in the line side and transmitted to the client side. In this design, the OTN scrambling function, which ensures a minimum number of transitions in the bit stream (breaking long sequences of consecutive digits) in order to facilitate synchronization and clock recovery, was embedded into the OTL4.10 block. A CPU interface block with a dedicated configuration register bank was also implemented, allowing an external CPU/MCU to manage and monitor all the internal functional blocks.

5.3.1 CHIP DESIGN AND MANUFACTURING

The 100G OTN Processor design flow, including the Crypto4OTN block, was based on Cadence (RTL simulation, verification, and physical design) and Synopsys (synthesis) tools. The complete ASSP design was later fully verified using the Universal Verification Methodology (UVM) [131, 132], including post-synthesis simulations and static timing analysis.

The ASSP manufacturing was done at TSMC (Taiwan Semiconductor Manufacturing Company) using the 40 nm general purpose technology in a multi-layer mask (MLM) program, a cost-effective strategy for chip prototyping or small volume production runs, as it reduces costs by combining many mask layers into a single reticle.

Chip conception, architecture, front-end design, and verification was done by CPQD, while the back-end (physical) design was built up with the partnership of other national and international institutions. The front-end RTL design was focused on the OTN, FEC [133], and the 100G AES-GCM Cryptography Engine [134] solutions, with complex mixed-signal blocks such as the SerDes and phase-locked loop (PLL) devices being licensed from silicon IP providers.

The 40 nm technology node was adopted as a good cost and performance tradeoff at the time of development and tape-out (2013–2017), considering the high data rates and logic circuit density, as well as the availability of the critical IP solutions — SerDes and PLLs. Typical core clock frequencies in this node are below 400 MHz, what demands parallel processing of the high-speed signals. Both 100 Gbit/s (111.8 Gbit/s) client and line side interfaces are made up of 10 lanes of 11.18 Gbit/s, which are internally parallelized by SerDes IP blocks, creating a 640-bit wide data path running below 180 MHz.

Table 5.1 shows the hierarchical area distribution of the Crypto4OTN design (top block and sub-blocks), as reported by the synthesis tool.

	Hierarchical Cell	Absolute Total Area (µm²)	Percent Total Area
Top Level Design	crypto4otn	2930205.2904	100.0
TX Processor Lineup	loopback_mux_client_loopback	3658.8888	0.1
	crypto_engine_tx	936325.2176	32.0
	oh_inserter	5959.1448	0.2
	repl_sig_gen	4066.1964	0.1
	flow_control_tx	73466.0843	2.5
RX Processor Lineup	loopback_mux_line_loopback	3663.1224	0.1
	oh_extractor	5580.5904	0.2
	crypto_engine_rx	936353.4416	32.0
	auth_buffer	829702.7142	28.3
	flow_control_rx	73470.6707	2.5
Controller	control_engine	57956.5735	2.0

Table 5.1 – Crypto4OTN block hierarchical area distribution.

Table 5.2 brings some of the chip and package main parameters, and Figure 5.29 shows pictures of the packaged device, with and without the heat spreader top cover.

Die size	7 × 10 mm
Gate count	≈ 8 M gates
Data path width	640 bits
Core frequency	180 MHz
Package type	FCBGA – Flip Chip Ball Grid Array
Package ball count	1369 balls
Package size	37.5 × 37.5 mm

Table 5.2 – 100G OTN Processor chip and package parameters.

Figure 5.29 – Packaged FCBGA 100G OTN Processor ASSP pictures.

5.4 CHIP PROTOTYPE TESTING

Several tests were planned for the chip prototypes in order to preliminarily validate the main IP blocks and the overall main functionalities. The CPU interface and the SerDes IP blocks constitute the main access ports for the 100G OTN Processor device, being then the first ones to be tested and verified.

Some mixed-signal and digital IP blocks had already been validated in a Test Chip [25] previously designed to allow for exercising the design flow tool set, procedures, and the manufacturing chain. These include the SerDes, PLLs, CPU interface, and OTL4.10 blocks.

For the 100G OTN Processor ASSP, the idea was to generate OTN traffic using a protocol analyzer equipment, inject data in the client side input interface, loop them back in the line side, and then return to the OTN protocol analyzer from the client side output. In this way, each individual block could be tested by configuring the bypass and loopback modes as needed in the other blocks of the TX and RX processor chains.

Figure 5.30 shows a setup diagram of the planned test environment, with a JDSU ONT-512 protocol analyzer being connected to a C form-factor pluggable (CFP) optical module in the evaluation board by a 20-lane multi-fiber push-on (MPO) cable.

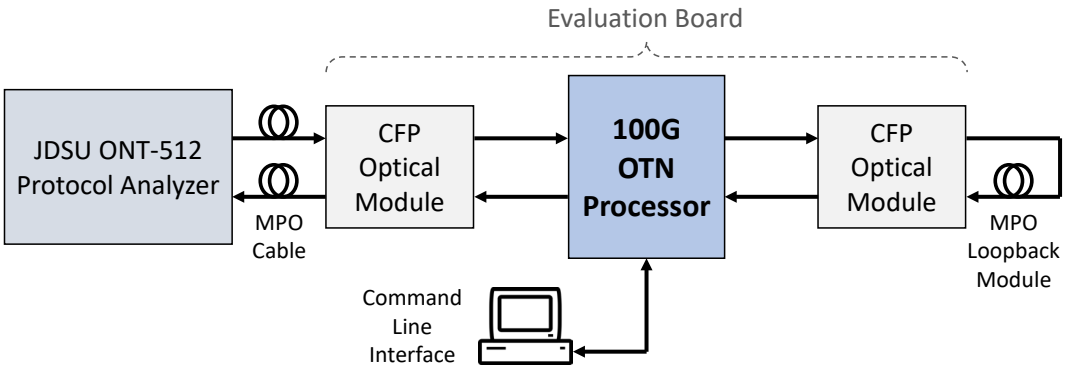


Figure 5.30 – 100G OTN Processor test environment setup diagram.

5.4.1 EVALUATION BOARD

Figure 5.31 shows a 3D model of the evaluation board (EVB) developed and manufactured to allow for testing and characterization of the 100G OTN Processor ASSP using an OTN protocol analyzer test equipment.

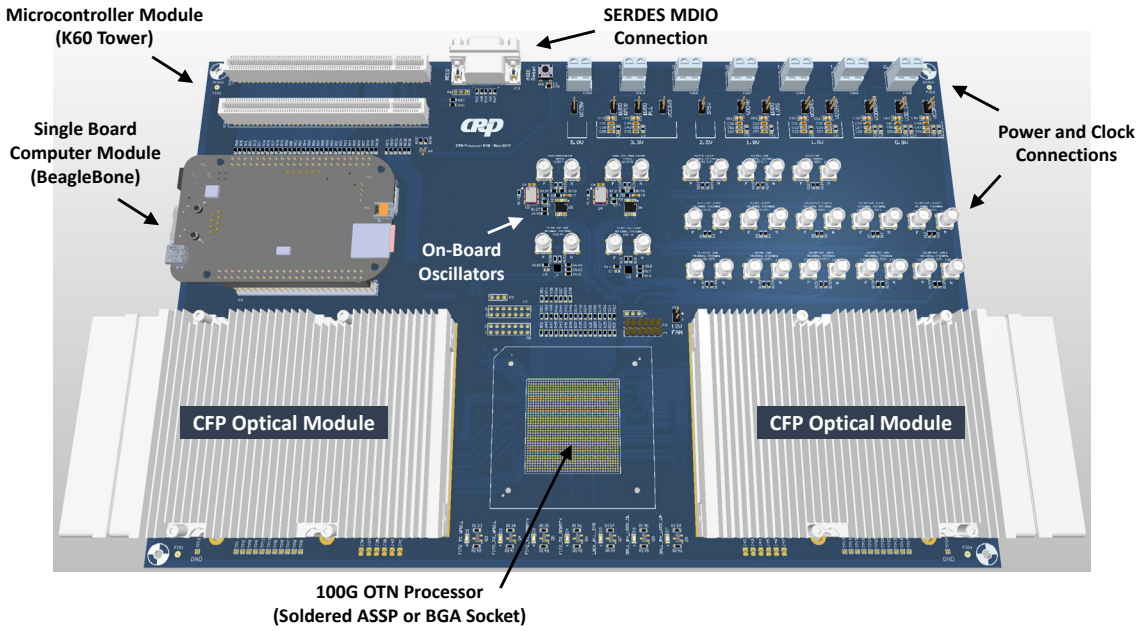


Figure 5.31 – 3D model of the evaluation board developed for the 100G OTN Processor.

The CPU module (used for accessing the 100G OTN Processor configuration registers) is external, not integrated into the EVB. There are two options: NXP tower system microcontrollers (like the K60 series) or the BeagleBone single board computer (SBC).

In order to simplify the design, power supply circuits were not incorporated. The necessary voltages are provided by external sources connected to a set of terminal blocks available in the EVB. Several SMA RF coaxial connections have been included for accessing the clock signals (input and output) of the 100G OTN Processor. On-board oscillators can also be used to generate the necessary clock signals. A DE-9 connector provides access to the management data input/output (MDIO) interface of each set of SerDes, allowing their configuration to be carried out during the testing phase.

Figure 5.32 shows a picture of the manufactured EVB with the 100G OTN Processor device soldered on the printed circuit board (PCB). Slots and connectors for 100 Gbit/s CFP optical modules can be seen in both sides of the ASSP, providing access to the client and line side interfaces.

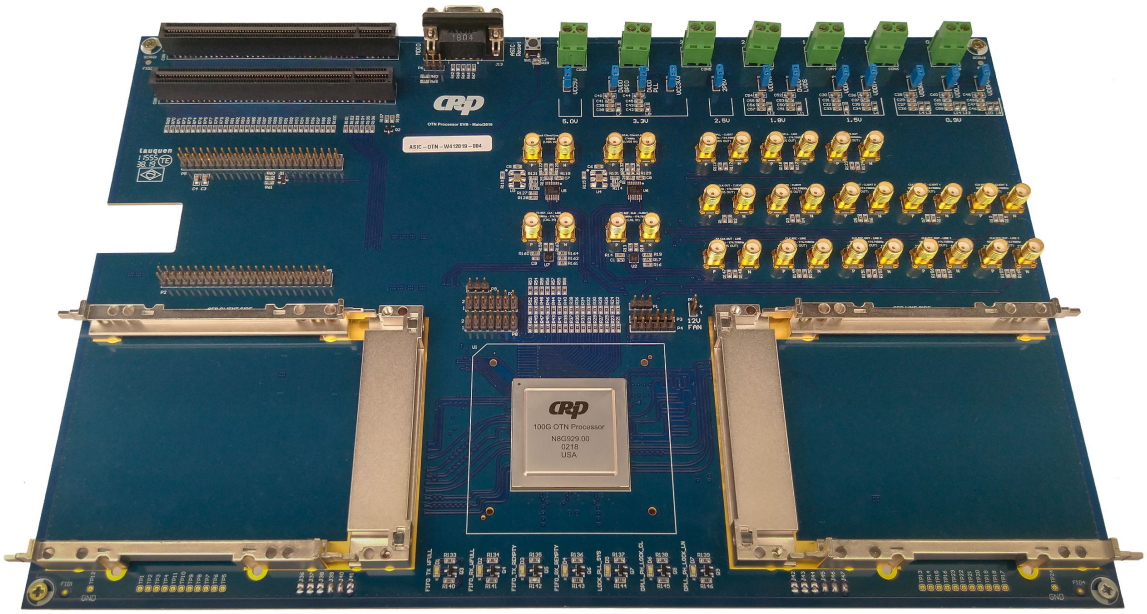


Figure 5.32 – Evaluation board designed and manufactured for testing the 100G OTN Processor ASSP.

The 100G OTN Processor device can also be mounted on a ball grid array (BGA) socket, as shown in the pictures of Figure 5.33. This allows for an easy device exchanging during the testing phase.

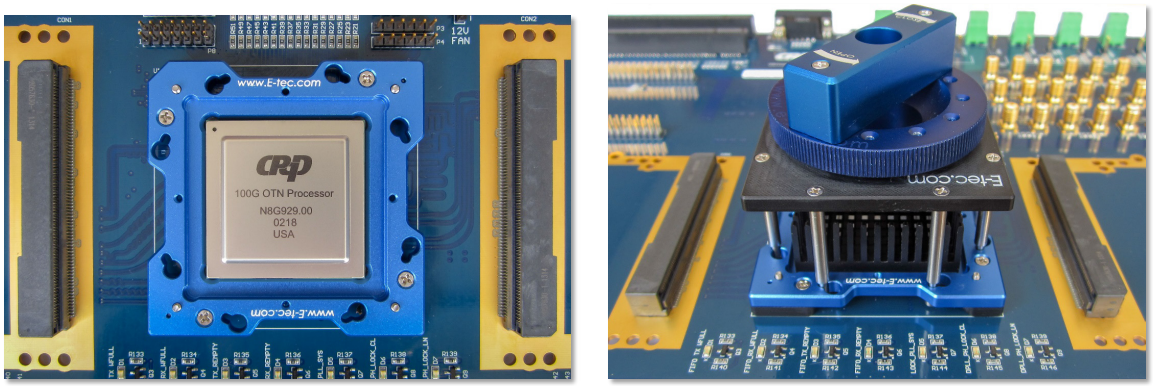


Figure 5.33 – BGA socket for the 100G OTN Processor assembly and testing.

A picture showing the completely mounted EVB, with a soldered 100G OTN Processor (under a heatsink), the CFP optical modules, the CPU module (BeagleBone Black), and a USB-MDIO conversion board is shown in Figure 5.34.

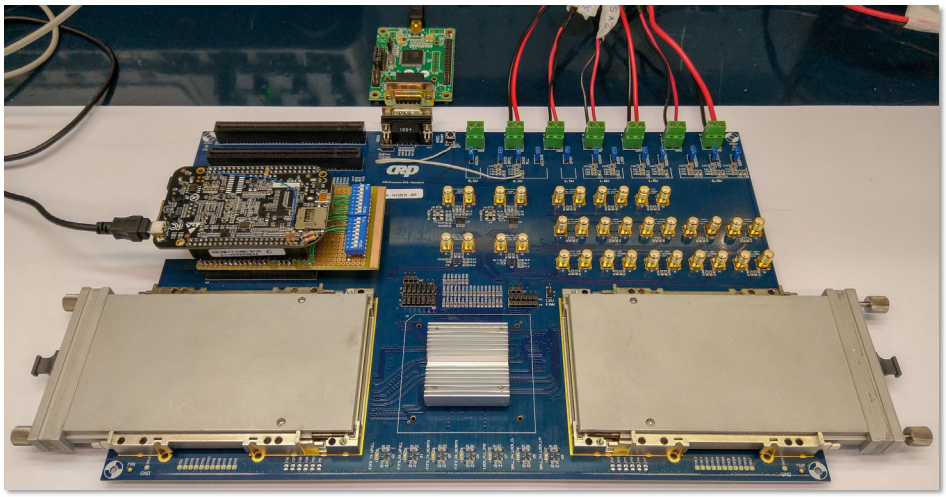


Figure 5.34 – 100G OTN Processor evaluation board with the CPU (SBC) and CFP modules installed.

Figure 5.35 shows the setup prepared for the EVB bring-up and the preliminary tests of the 100G OTN Processor. It is part of the overall arrangement depicted in Figure 5.30, with an additional digital oscilloscope for elementary signal measurements.

Figure 5.35 – Setup for EVB bring-up and preliminary tests of the 100G OTN Processor.

5.4.2 CPU Interface Testing

An application was developed and implemented in the selected CPU module (BeagleBone Black) based on two processes (one written in Python and the other, in C language) running under the embedded Linux operating system. The first provides a command-line interface (CLI) that enables read and write access to the various registers of the 100G OTN Processor, while the second controls the data bus according to the protocol of the ASSP

CPU interface sub-block. After bringing the EVB up and debugging the CLI code, preliminary register access tests were performed.

Most logic blocks (or, in some cases, groups of blocks) of the device have a test register dedicated for writing and reading operations, as well as another one programmed with the version of the RTL code, as shown in Table 5.3.

Block	Test Register Address	RTL Version Register Address	RTLVersion
sys_ctrl	0x0	0x1	0x100
pllwrp_sysb	0x100	0x101	0x100
crypto4otn	0x200	0x201	0x100
xco4 #1	0x4000	0x4001	0x100
gfec100 #1	0x5C80	0x5C81	0x100
otl410 #1	0x5D00	0x5D01	0x100
xco4 #2	0x8200	0x8201	0x100
gfec100 #2	0x9E80	0x9E81	0x100
otl410 #2	0x9F00	0x9F01	0x100
sd10wrp #1	0xA000	-	-
dpll_0 #1	0xC100	0xC101	0x100
dpll_1 #1	0xC140	0xC141	0x100
dpll_2 #1	0xC180	0xC181	0x100
sd10wrp #2	0xCE40	-	-
dpll_0 #2	0xEF40	0xEF41	0x100
dpll_1 #2	0xEF80	0xEF81	0x100
dpll_2 #2	0xEFC0	0xEFC1	0x100
cbr_framer	0xFFE0	0xFFE1	0x100

Table 5.3 – Test and RTL version registers for each block (or group of blocks) of the 100G OTN Processor.

Figure 5.36 shows the results of the CLI commands for (a) writing and (b) reading of a dataset (0xA001, 0xB002, ..., 0xBF18) to/from all test registers of Table 5.3. The correspondence in the observed values evidences the correct operation of the configuration register access interface. Figure 5.36 (c) shows the values read back from the RTL version registers (not available for the blocks *sd10wrp #1* and *sd10wrp #2*). Again, the correspondence in the observed values certifies the desired operation.

```
-> Writing Data: 0xA001 to Addr: 0x0
-> Writing Data: 0xB002 to Addr: 0x100
-> Writing Data: 0xC003 to Addr: 0x200
-> Writing Data: 0xD004 to Addr: 0x4000
-> Writing Data: 0xE005 to Addr: 0x5C80
-> Writing Data: 0xF006 to Addr: 0x5D00
-> Writing Data: 0xAA07 to Addr: 0x8200
-> Writing Data: 0xAB08 to Addr: 0x9E80
-> Writing Data: 0xAC09 to Addr: 0x9F00
-> Writing Data: 0xAD10 to Addr: 0xA000
-> Writing Data: 0xAE11 to Addr: 0xC100
-> Writing Data: 0xAF12 to Addr: 0xC140
-> Writing Data: 0xBA13 to Addr: 0xC180
-> Writing Data: 0xBB14 to Addr: 0xCE40
-> Writing Data: 0xBC15 to Addr: 0xEF40
-> Writing Data: 0xBD16 to Addr: 0xEF80
-> Writing Data: 0xBE17 to Addr: 0xEFC0
-> Writing Data: 0xBF18 to Addr: 0xFFE0
```

(a)

```
-> Data: b'0XA001' read from addr: 0x0
-> Data: b'0XB002' read from addr: 0x100
-> Data: b'0XC003' read from addr: 0x200
-> Data: b'0XD004' read from addr: 0x4000
-> Data: b'0XE005' read from addr: 0x5C80
-> Data: b'0XF006' read from addr: 0x5D00
-> Data: b'0XAA07' read from addr: 0x8200
-> Data: b'0XAB08' read from addr: 0x9E80
-> Data: b'0XAC09' read from addr: 0x9F00
-> Data: b'0XAD10' read from addr: 0xA000
-> Data: b'0XAE11' read from addr: 0xC100
-> Data: b'0XAF12' read from addr: 0xC140
-> Data: b'0XBA13' read from addr: 0xC180
-> Data: b'0XBB14' read from addr: 0xCE40
-> Data: b'0XBC15' read from addr: 0xEF40
-> Data: b'0XBD16' read from addr: 0xEF80
-> Data: b'0XBE17' read from addr: 0xEFC0
-> Data: b'0XBF18' read from addr: 0xFFE0
```

(b)

```
-> Data: b'0X100' read from addr: 0x1
-> Data: b'0X100' read from addr: 0x101
-> Data: b'0X100' read from addr: 0x201
-> Data: b'0X100' read from addr: 0x4001
-> Data: b'0X100' read from addr: 0x5C81
-> Data: b'0X100' read from addr: 0x5D01
-> Data: b'0X100' read from addr: 0x8201
-> Data: b'0X100' read from addr: 0x9E81
-> Data: b'0X100' read from addr: 0x9F01
-> Data: b'0X1' read from addr: 0xA001
-> Data: b'0X100' read from addr: 0xC101
-> Data: b'0X100' read from addr: 0xC141
-> Data: b'0X100' read from addr: 0xC181
-> Data: b'0X1' read from addr: 0xCE41
-> Data: b'0X100' read from addr: 0xEF41
-> Data: b'0X100' read from addr: 0xEF81
-> Data: b'0X100' read from addr: 0xEFC1
-> Data: b'0X100' read from addr: 0xFFE1
```

(c)

Figure 5.36 – Results of the CLI commands for the 100G OTN Processor test register data (a) writing and (b) reading, and (c) RTL version register reading.

5.4.3 SERDES TESTING

The SerDes bring-up and testing were carried out with a USB-MDIO converter module connected to the EVB. MDIO signals of each set of SerDes, which are made available in dedicated pins of the ASSP, were routed to a DE-9 connector mounted in the EVB, as shown on top of the picture in Figure 5.34.

The configuration of the SerDes parameters can also be done through some registers accessed by the CLI described in Section 5.4.2. However, the use of the MDIO interface brings the advantage of enabling the use of a configuration and testing software (with a graphical user interface), developed and supplied by the SerDes IP provider.

Preliminary tests with the SerDes blocks were done with the use of loopback connections in one of the main ASSP interfaces (client or line side), provided by an interconnection module similar to a CFP — a passive break-out module, as shown in Figure 5.37 — which enables access to all the TX and RX lanes of the CFP connector. A single-channel 11.18 Gbit/s differential electrical loopback connection is shown in Figure 5.38.

All channels were tested in this way with two EVBs populated with a soldered 100G OTN Processor. On a third board, using the BGA socket, the communication with the ASSP through the CLI was achieved, but there was no successful SerDes loopback test result.

One of those two EVBs with the soldered ASSP had problems in the client side interface, while the other, in the line side. Tests were done by a bit error rate tester (BERT) module integrated into the SerDes IP block.

Figure 5.37 – CFP passive break-out module.

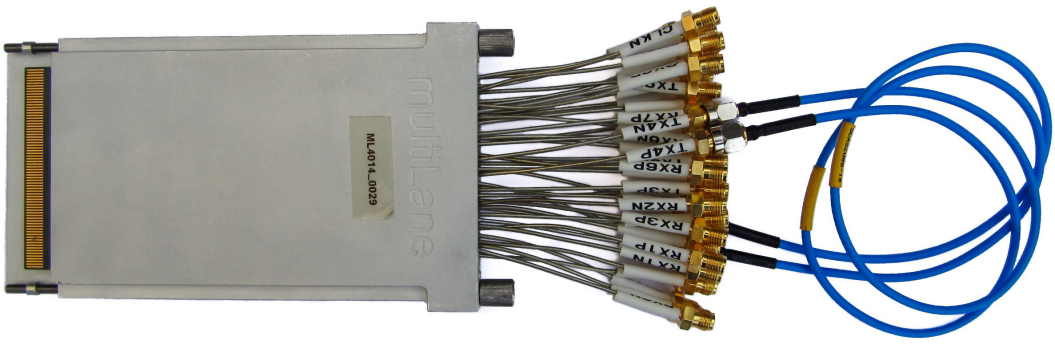


Figure 5.38 – CFP passive break-out module with a single-channel 11.18 Gbit/s differential electrical loopback connection.

Once the SerDes blocks were working properly, the tests proceeded with an attempt to connect the CFP optical modules, together with the use of an MPO loopback adapter, as shown in Figure 5.39.

Figure 5.39 – Multi-fiber push on loopback adapter.

The strategy was to try getting the same previous results obtained with the SerDes electrical loopbacks. Nevertheless, several unsuccessful attempts were carried out with the available EVBs and different CFP modules.

The SerDes circuit incorporates an eye diagram analyzer that allows observing the high-speed serial signal using the manufacturer's configuration software connected via the MDIO interface.

Figure 5.40 shows the results obtained with the electrical loopback (using the CFP passive break-out module). Bit transitions are clearly observed, creating the characteristic eye diagram shape. Figure 5.41, on the other hand, shows the corresponding signal without the expected transitions obtained from the tests with the CFP module and the MPO loopback adapter.

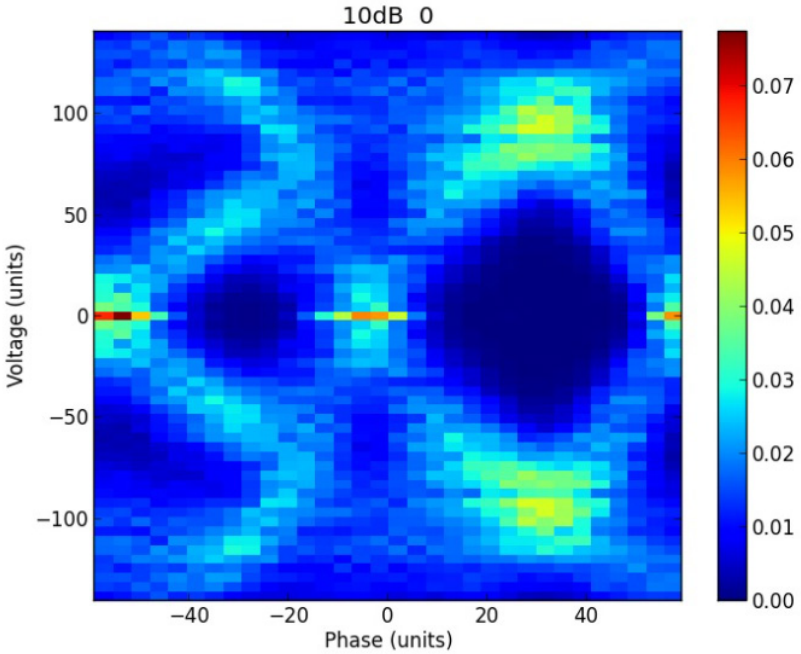


Figure 5.40 – Eye diagram of an 11.18 Gbit/s signal received by one of the SerDes showing bit transitions.

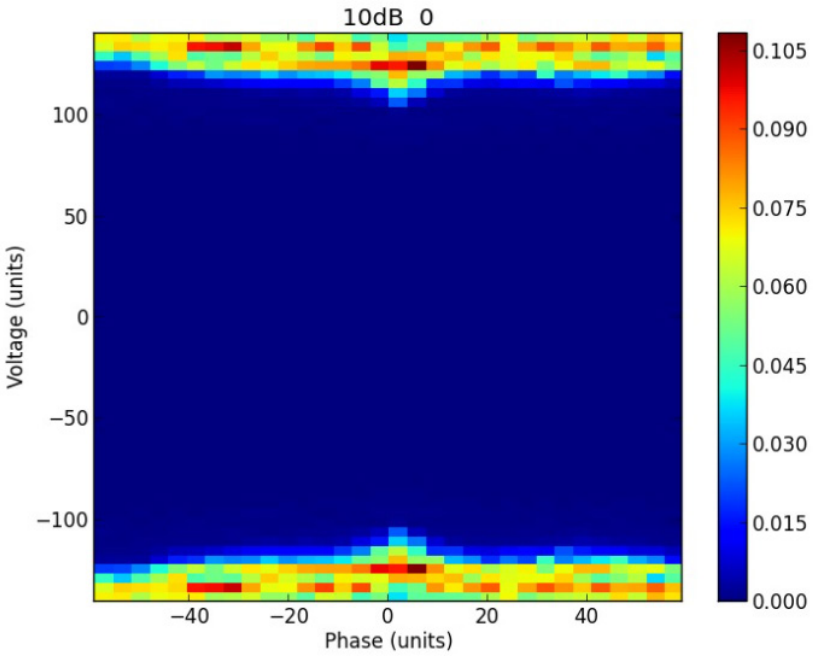


Figure 5.41 – Eye diagram of an 11.18 Gbit/s signal received by one of the SerDes without bit transitions.

The MPO loopback adapter was then substituted by an optical connection with a second CFP module installed in the OTN traffic protocol analyzer equipment (JDSU ONT-512). Figure 5.42 shows the main components of the test arrangement depicted in Figure 5.30.

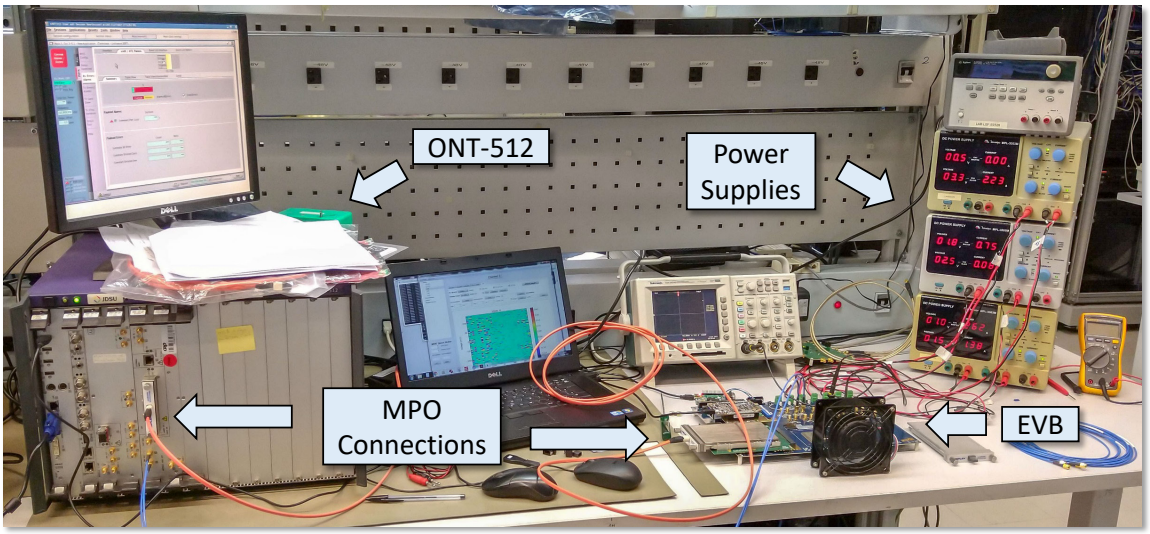


Figure 5.42 – 100G OTN Processor EVB test setup with a JDSU ONT-512 OTN traffic analyzer.

None of the additional tests was successful, and, after a thorough investigation work, a classical problem was finally detected in the EVB: the RX signals from the CFP, corresponding to the data outputs, were incorrectly connected to the TX outputs of the

100G OTN Processor. In the same way, the TX signals of the CFP, corresponding to the data inputs, were connected to the RX inputs of the ASSP.

The previous SerDes tests were successful because of the use of electrical connections (via RF cables) for the TX-RX loopback, creating an effective output-to-input connection in the ASSP despite the PCB routing inversion on the CFP connector.

5.4.4 IP Block Testing

The mistake in the EVB design, described in the last section, made it impossible to continue testing on this platform. Consequently, the ASSP IP blocks, including of course the 100G AES-GCM Cryptography Engine, could not be tested.

A small CFP Cross Adapter card was designed and manufactured in an attempt to solve the interconnection mistake between the CFP modules and the ASSP, as shown in Figure 5.43. A detailed view on the right-hand side shows how TX and RX lanes are crossed to correct the input/output connections of the high-speed interface.

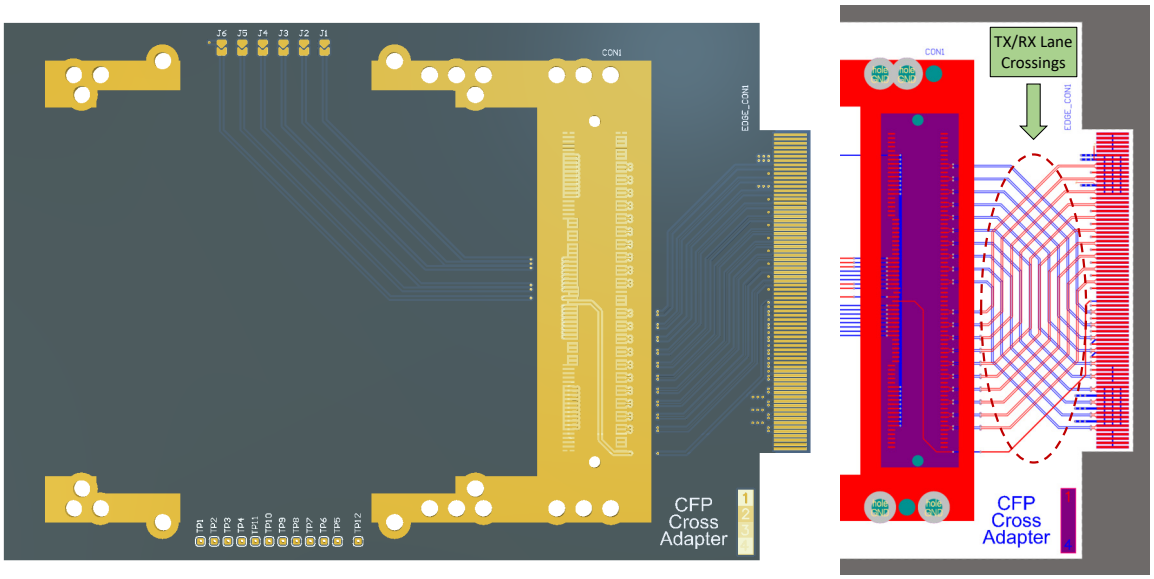


Figure 5.43 – CFP Cross Adapter card designed to correct the interconnection mistake between a CFP module and the ASSP.

The CFP module is assembled in this card, which is then inserted in the original ASSP EVB CFP slot, as shown in Figure 5.44.

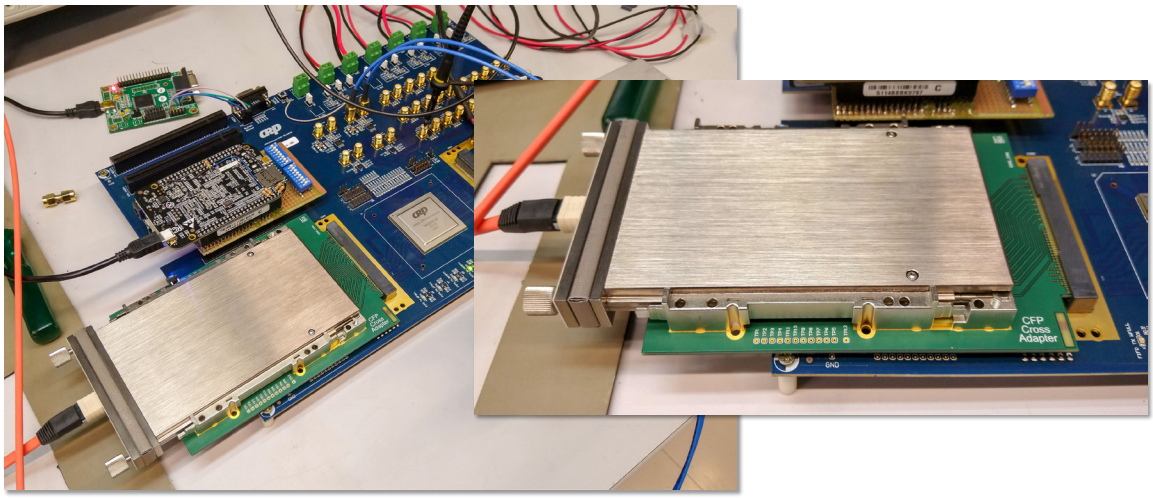


Figure 5.44 – CFP Cross Adapter card inserted on the ASSP EVB CFP slot.

New tests were then performed using the ONT-512 protocol analyzer. As depicted in Figure 5.45, a successful (nearly error-free) SerDes parallel bus loopback was achieved with 100 Gbit/s PRBS data.

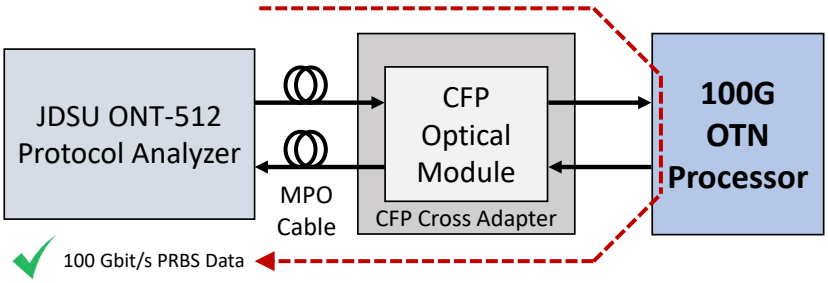


Figure 5.45 – Successful SerDes parallel bus loopback with 100 Gbit/s PRBS data.

When configuring the equipment to generate the OTN protocol, however, it changes the bit rate to 111.8 Gbit/s (OTU4), corresponding to 11.18 Gbit/s per lane. Although the SerDes worked properly at this data rate with the electrical loopback, as described in Section 5.4.3, this was not the case using the CFP module with the CFP Cross Adapter card. The ONT-512 protocol analyzer could not establish a data link in any lane.

A couple of reasons may explain this — frequency response problems in the small sections of high-speed PCB transmission lines on the CFP Cross Adapter card and/or performance issues with some sub-modules of the SerDes IP block.

In the first assumption, high-speed digital signal integrity may be degraded by the poor frequency response of the low-cost dielectric material (FR-4) used in the PCB stack-up

construction. Ideally, this should be compensated by the SerDes preemphasis (in the transmission path) and digital equalizer (in the reception path), which invert the frequency response of the PCB lossy channel [25].

The second hypothesis relates to the performance of the high-speed PLLs and clock recovery circuits within the SerDes IP block. During the Test Chip [25] evaluation and testing, the SerDes IP provider had suggested increasing the core supply voltage since the integrated PLL was apparently not responding very well at 11.18 Gbit/s data rates. This was also tried with the 100G OTN Processor with no meaningful results.

Despite the frustrating attempt to use the EVB for the ASSP IP block testing, this can still be done through a Debug Module also designed and implemented in the 100G OTN Processor, which allows for testing the complete TX and RX data paths using a standard JTAG test access port, together with a custom software that needs to be developed to generate, collect, and analyze data.

As depicted in Figure 5.46 a Debug Generator sub-block inserts software-generated data in the beginning of both TX and RX processor data paths. A Debug Monitor sub-block at the end of each chain collects data (with triggering functionalities), which are then sent to an external computer through the JTAG interface port.

The 100G OTN Processor remains yet to be fully tested and validated using this technique.

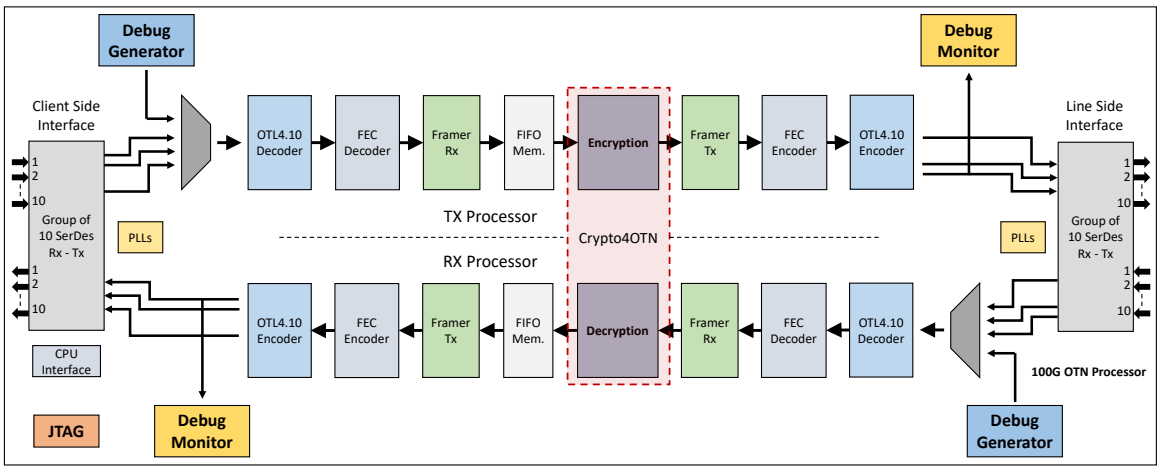


Figure 5.46 – Debug Module (generator and monitor) implemented in the 100G OTN Processor.

Chapter 6

CONCLUSIONS AND FINAL REMARKS

This work reported the conception, architectural design, RTL design, verification, and a 40 nm silicon prototyping of a 100 Gbit/s AES-GCM Cryptography Engine solution specifically developed to secure high-speed OTN communication links.

Systemic aspects were addressed, with emphasis on the use of a block cipher infrastructure properly adapted to enable data integrity and confidentiality over the inherently insecure OTN protocol. The logic architecture was described in a top-down approach, showing implementation details related to the generation and transport of cryptographic packets, encryption overhead construction, key change, and the necessary synchronization mechanisms.

Chip and ASSP manufacturing parameters of a 100 Gbit/s OTN processor device developed by CPQD and integrating the proposed cryptography solution were also presented, along with laboratory prototype evaluations.

Chapter 2 briefly described optical transport technologies, highlighting their frame structure and protection mechanisms. Security threats in optical networks were also overviewed, with a few considerations regarding software-defined and disaggregated optical networks.

Cryptographic systems were then presented after some relevant mathematical background, starting with the main concepts related to data encryption and authentication, briefing over their historical evolution, presenting the main definitions relating to encryption, authentication, and symmetric/asymmetric cryptosystems as well as some details of the AES algorithm, and stepping into quantum cryptography.

The final sections showed some of the cryptographic modes of operation, with emphasis on the most relevant in the scope of this work (Galois/Counter Mode), the quantum

computing threats endangering classical encryption algorithms, as well as their quantum-resistant counterparts of post-quantum cryptography.

Chapter 3 described the entire architecture of the cryptographic system developed for the establishment of secure links in OTN systems, with emphasis on the concepts of cryptographic session, block, and packet, as well as details about the role of the hardware and software layers, the generation and transport of the encryption overhead data (TAG, AAD, and IV) necessary for the link operation, the error or failure recovery mechanisms, and the cryptographic session management.

Chapter 4 presented the 100G AES-GCM Cryptography Engine, designed and implemented according to the conceived systemic architecture solution, with its features, characteristics, operation modes, configuration procedures, hardware functional architecture, partitioning of the logic building blocks, and implementation details.

The design, verification, and 40 nm silicon prototyping of the proposed solution was shown in Chapter 5. All the 100G AES-GCM Cryptography Engine functional logic blocks were implemented within the Crypto4OTN top module, and each one was coded in the Verilog hardware description language aiming at the subsequent logic synthesis phase.

Functional verification of the complete cryptographic solution during the RTL design phase was also described, with digital simulations based on a regular test bench module being explained and demonstrated by timing diagrams and signal waveforms, evidencing the overall functionality of some sub-blocks and the complete integrated Crypto4OTN design.

The last sections presented the 8M gate 40 nm 100 Gbit/s OTN processor ASSP developed by CPQD for the Brazilian telecom industry, integrating the 100G AES-GCM Cryptography Engine, as well as the laboratory test setups, procedures, and main results.

6.1 MAIN CONTRIBUTIONS

The author's main contributions were the systemic and functional architecture design of the cryptographic solution for optical transport networks, as well as the RTL design and verification (through functional digital simulations) of almost all the logic blocks that make up the 100G AES-GCM Cryptography Engine.

He also actively participated in the design of the 100G OTN Processor ASSP developed by CPQD for the Brazilian telecom industry, in the development of the command-line interface software for its evaluation board, and in the laboratory prototype tests.

6.2 REMARKS AND FUTURE WORK SUGGESTIONS

Being driven by the original requirements of the 100G OTN Processor, the development of the 100G AES-GCM Cryptography Engine was tailored to the specificities of the OTN protocol structure.

Recent contributions of the ITU-T Study Group 15 [135–139] have presented questions and proposals for a multi-vendor interoperable OTN Security (OTNsec) standard [140]. Alternative encryption procedure candidates are being considered, such as the OTN Encryption Procedure (OEP) and the Generic Encryption Procedure (GEP). The first one carries OTNsec specific overheads inside the OxU (OPU/ODU/OTU) overhead, while the second, in the OPU or FEC payload area.

It has also been proposed that OTNsec must not occupy all reserved overhead bits and bytes in the OPU, ODU, and OTU overhead areas, what implies, in the conception of the 100G AES-GCM Cryptography Engine, larger crypto packets in order to accommodate the size of the necessary TAG and IV fields and, consequently, a higher store-and-forward latency.

The OTN technology is now in the beyond 100G (B100G) era, extending data rates over 100 Gbit/s and introducing a new Flexible OTN (FlexO) [141] interface with its own secure transport architecture (FlexOsec) [142].

Future works could review the latest ITU-T SG15 contributions and develop cryptography solutions in accordance with these new multi-vendor interoperable OTNsec concepts.

The feasibility and possible advantages of transferring the asymmetric encryption protocol used to exchange keys in an OTN link, such as the Elliptic Curve Diffie–Hellman (ECDH), from the software to the hardware layer could also be investigated. In this case, cryptographic protocol messages could be transmitted through the general communication channel (GCC) bytes of the OTN overhead.

Without the OTN protocol restrictions, a Standalone AES-GCM Cryptography Module could also be developed. A terminal communication unit capable of establishing secure point-to-point Ethernet links, for instance, would provide confidentiality and integrity to transmitted data in untrusted network domains. It could operate either in low (100 Mbit/s) or high-speed (1/10/100 Gbit/s) data rates. Client data mapping procedures, as well as the encryption overhead container, would be the main research goals in this case.

Aiming at the current quantum-safe security trends, a hardware interface could be designed and implemented to integrate the 100G AES-GCM Cryptography Engine (or any other symmetric-key encryption module) with a quantum key distribution system, possibly with a standardized QKD interface such as the European OpenQKD project [83] and the work of the European Telecommunications Standards Institute (ETSI) [84].

Current high-end FPGA technology allows for a complete prototyping or even final deployment of the cryptography solution hardware layer, turning concepts, demonstrations, and products into reality without the risks and costly efforts of high-speed digital IC design and manufacturing.

REFERENCES

[1] S. Aleksic, "Towards Fifth-Generation (5G) Optical Transport Networks". *2015 17th International Conference on Transparent Optical Networks (ICTON)*, pp. 1–4, 2015. https://doi.org/10.1109/ICTON.2015.7193532.

[2] Xiang Liu, "Evolution of Fiber-Optic Transmission and Networking Toward the 5G Era". *iScience*, vol. 22, pp. 489–506, 2019. https://doi.org/10.1016/j.isci.2019.11.026.

[3] J. Pedro, N. Costa, and S. Pato, "Optical Transport Network Design Beyond 100 Gbaud [Invited]". *IEEE/OSA Journal of Optical Communications and Networking*, vol. 12, no. 2, pp. A123–A134, February 2020. https://doi.org/10.1364/JOCN.12.00A123.

[4] Zhou He and Fanjian Hu, "Research on Novel Modulation Format Apol-FSK for Optical Transport Network of 5G". *Optics Communications*, vol. 474, 126055, 2020. https://doi.org/10.1016/j.optcom.2020.126055.

[5] Gorshe, S., "A Tutorial on ITU-T G.709 Optical Transport Networks (OTN)". *PMC-Sierra*, white paper, document ID: PMC-2081250, issue 2, June 2011.

[6] M. Carroll, J. Roese, and T. Ohara, "The Operator's View of OTN Evolution". *IEEE Communications Magazine*, vol. 48, no. 9, pp. 46–52, September 2010. https://doi.org/10.1109/MCOM.2010.5560586.

[7] Paul Littlewood, Fady Masoud, and Malcolm Loro, "Optical Transport Networking". *Ciena Expert Series*, 2015. Available at: https://media.ciena.com/documents/Experts_Guide_to_OTN_ebook-Utilities-Edition.pdf. Accessed 21 April 2023.

[8] Andreas Schubert, "G.709 – The Optical Transport Network (OTN)". *Viavi*, white paper, 2021. Available at: https://www.viavisolutions.com/en-us/literature/g709-optical-transport-network-otn-white-papers-books-en.pdf. Accessed 21 April 2023.

[9] D. Cavendish, "Evolution of Optical Transport Technologies: from SONET/SDH to WDM". *IEEE Communications Magazine*, vol. 38, no. 6, pp. 164–172, June 2000. https://doi.org/10.1109/35.846090.

[10] "Interfaces for the Optical Transport Network (OTN)". *ITU-T recommendation G.709/Y.1331*, 2020.

[11] "Optical Transport Network Global Market Report". *The Business Research Company*, 2023. Available at: https://www.thebusinessresearchcompany.com/report/optical-transport-network-global-market-report. Accessed 21 April 2023.

[12] J. Justesen, K. J. Larsen, and L. A. Pedersen, “Error Correcting Coding for OTN”. *IEEE Communications Magazine*, vol. 48, no. 9, pp. 70–75, September 2010. https://doi.org/10.1109/MCOM.2010.5560589.

[13] S. V. Kartalopoulos, “A Primer on Cryptography in Communications”. *IEEE Communications Magazine*, vol. 44, no. 4, pp. 146–151, April 2006. https://doi.org/10.1109/MCOM.2006.1632662.

[14] S. V. Kartalopoulos, “Quantum Cryptography for Secure Optical Networks”. *IEEE International Conference on Communications*, Glasgow, UK, pp. 1311–1316, 2007. https://doi.org/10.1109/ICC.2007.221.

[15] K. Guan, J. Kakande, and J. Cho, “On Deploying Encryption Solutions to Provide Secure Transport-as-a-Service (TaaS) in Core and Metro Networks”. *42nd European Conference on Optical Communication (ECOC)*, pp. 1–3, 2016.

[16] Cho J. Y., “Securing Optical Networks by Modern Cryptographic Techniques”. *NordSec 2019*. *Lecture Notes in Computer Science*, vol. 11875. Springer, Cham, 2019. https://doi.org/10.1007/978-3-030-35055-0_8.

[17] Dilip Kumar Sharma, Ningthoujam Chidananda Singh, Daneshwari A Noola, Amala Nirmal Doss, and Janaki Sivakumar, “A Review on Various Cryptographic Techniques & Algorithms”. *Materials Today*, vol. 51, part 1, pp. 104–109, 2022. https://doi.org/10.1016/j.matpr.2021.04.583.

[18] Rachelle Miller, “The OSI Model: An Overview”. *SANS Institute*, white paper, 2001. Available at: https://www.sans.org/white-papers/543/. Accessed 21 April 2023.

[19] Xinmiao Zhang and K. K. Parhi, “High-Speed VLSI Architectures for the AES Algorithm”. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 9, pp. 957–967, September 2004. https://doi.org/10.1109/TVLSI.2004.832943.

[20] L. Henzen and W. Fichtner, “FPGA Parallel-Pipelined AES-GCM Core for 100G Ethernet Applications”. *Proceedings of ESSCIRC*, Seville, Spain, pp. 202–205, 2010. https://doi.org/10.1109/ESSCIRC.2010.5619894.

[21] M. Mozaffari-Kermani and A. Reyhani-Masoleh, “Efficient and High-Performance Parallel Hardware Architectures for the AES-GCM”. *IEEE Transactions on Computers*, vol. 61, no. 8, pp. 1165–1178, August 2012. https://doi.org/10.1109/TC.2011.125.

[22] A. P. Anusha Naidu and P. K. Joshi, “FPGA Implementation of Fully Pipelined Advanced Encryption Standard”. *2015 International Conference on Communications and Signal Processing (ICCSP)*, pp. 649–653, 2015. https://doi.org/10.1109/ICCSP.2015.7322568.

[23] B. Buhrow, K. Fritz, B. Gilbert, and E. Daniel, "A Highly Parallel AES-GCM Core for Authenticated Encryption of 400 Gb/s Network Protocols". *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Riviera Maya, Mexico, pp. 1–7, 2015. https://doi.org/10.1109/ReConFig.2015.7393321.

[24] David Smekal, Jan Hajny, and Zdenek Martinasek, "Comparative Analysis of Different Implementations of Encryption Algorithms on FPGA Network Cards". *IFAC-PapersOnLine*, vol. 51, no. 6, pp. 312–317, 2018. https://doi.org/10.1016/j.ifacol.2018.07.172.

[25] E. Mobilon, R. Bernardo, and L. R. Monte, "100 Gbit/s Optical Transport Network 40 nm Test Chip Design and Prototyping". *SBMO/IEEE MTT-S International Microwave and Optoelectronics Conference (IMOC)*, Águas de Lindóia, pp. 1–5, 2017. https://doi.org/10.1109/IMOC.2017.8121108.

[26] R. Bernardo, A. H. Salvador, E. Mobilon, L. R. Monte, S. Boisclair, and A. Warshawsky, "Design and FPGA Implementation of a 100 Gbit/s Optical Transport Network Processor". *23rd International Conference on Field Programmable Logic and Applications*, Porto, Portugal, pp. 1–4, 2013. https://doi.org/10.1109/FPL.2013.6645601.

[27] Anton A. Huurdeman. *The Worldwide History of Telecommunications*. John Wiley & Sons, ISBN 9780471205050, 2003. https://doi.org/10.1002/0471722243.

[28] D. Y. Al-Salameh, M. T. Fatehi, W. J. Gartner, S. Lumish, B. L. Nelson, and K. K. Raychaudhuri, "Optical Networking". *Bell Labs Technical Journal*, vol. 3, no. 1, pp. 39–61, January-March 1998. https://doi.org/10.1002/bltj.2092.

[29] "MPLS Technology White Paper". *New H3C Technologies*, 2020. Available at: https://www.h3c.com/en/Support/Resource_Center/EN/Home/Switches/00-Public/Trending/Technology_White_Papers/MPLS_Technology_WP-6W100/. Accessed 21 April 2023.

[30] William Stallings. *Cryptography and Network Security – Principles and Practice*. Seventh edition, Pearson, ISBN 9781292158587, 2017.

[31] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering – Design Principles and Practical Applications*. Wiley Publishing, Inc., ISBN 9780470474242, 2010.

[32] Shu Lin. *An Introduction to Error-Correcting Codes*. Prentice-Hall, USA, ISBN 9780134828107, 1970.

[33] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North-Holland Publishing Company, Netherlands, ISBN 9780444850096, 1977.

[34] "Forward Error Correction for Submarine Systems". *ITU-T recommendation G.975*, October 2000.

[35] "Switched OTN for Regional Carrier Networks". *Fujitsu*, white paper, 2013. Available at: https://www.fujitsu.com/ca/en/imagesgig5/OTN-for-Regional-Carriers.pdf. Accessed 21 April 2023.

[36] "The Role of OTN Switching in 100G & Beyond Transport Networks". *Coriant*, white paper, 2015. Available at: https://www.ofcconference.org/getattachment/90c0e6a4-08c1-45fb-a7f2-2957d444dc7d/The-Role-of-OTN-Switching-in-100G-Beyond-Transpo.aspx. Accessed 21 April 2023.

[37] M. L. Jones, "Mapping and Transport Standard for OTU4". *2010 Conference on Optical Fiber Communication (OFC/NFOEC)*, pp. 1–3, 2010. https://doi.org/10.1364/NFOEC.2010.NTuB2.

[38] "Forward Error Correction for High Bit-Rate DWDM Submarine Systems". *ITU-T recommendation G.975.1*, 2004.

[39] "OTU4 Long-Reach Interface". *ITU-T recommendation G.709.2/Y.1331.2*, 2018.

[40] Govind P. Agrawal. *Fiber-Optic Communication Systems*. Second edition, John Wiley & Sons, USA, ISBN 9780471175407, 1997.

[41] Govind P. Agrawal. *Nonlinear Fiber Optics*. Second edition, Academic Press, USA, ISBN 9780120451425, 1995.

[42] T. Huang, F. Qi, and F. Gao, "Failure Detection and Localization in OTN Based on Optical Power Analysis". *2010 Second International Conference on Communication Software and Networks*, pp. 46–51, 2010. https://doi.org/10.1109/ICCSN.2010.68.

[43] M. P. Fok, Z. Wang, Y. Deng, and P. R. Prucnal, "Optical Layer Security in Fiber-Optic Networks". *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 3, pp. 725–736, September 2011. https://doi.org/10.1109/TIFS.2011.2141990.

[44] N. Skorin-Kapov, M. Furdek, S. Zsigmond, and L. Wosinska, "Physical-Layer Security in Evolving Optical Networks". *IEEE Communications Magazine*, vol. 54, no. 8, pp. 110–117, August 2016. https://doi.org/10.1109/MCOM.2016.7537185.

[45] M. Zafar Iqbal, H. Fathallah, and N. Belhadj, "Optical Fiber Tapping: Methods and Precautions". *8th International Conference on High-Capacity Optical Networks and Emerging Technologies*, pp. 164–168, 2011. https://doi.org/10.1109/HONET.2011.6149809.

[46] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey". *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14-76, 2015. https://doi.org/10.1109/JPROC.2014.2371999.

[47] A. S. Thyagaturu, A. Mercian, M. P. McGarry, M. Reisslein, and W. Kellerer, "Software Defined Optical Networks (SDONs): A Comprehensive Survey". *IEEE Communications Surveys & Tutorials*, vol. 18, no. 4, pp. 2738-2786, 2016. https://doi.org/10.1109/COMST.2016.2586999.

[48] R. Alvizu et al., "Comprehensive Survey on T-SDN: Software-Defined Networking for Transport Networks". *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2232-2283, 2017. https://doi.org/10.1109/COMST.2017.2715220.

[49] Open and Disaggregated Transport Network (ODTN) project website. Available at: https://opennetworking.org/odtn/. Accessed 21 April 2023.

[50] I. Ahmad, S. Namal, M. Ylianttila, and A. Gurtov, "Security in Software Defined Networks: A Survey". *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2317–2346, 2015. https://doi.org/10.1109/COMST.2015.2474118.

[51] H. Davenport. *The Higher Arithmetic – An Introduction to The Theory of Numbers*. Eighth edition, Cambridge University Press, ISBN 9780521722360, 2008.

[52] Joseph H. Silverman. *A Friendly Introduction to Number Theory*. Fourth edition, Pearson, ISBN 9780321816191, 2012.

[53] Paar, Christof and Pelzl, Jan. *Understanding Cryptography – A Textbook for Students and Practitioners*. Springer, ISBN 9783642041006, 2010.

[54] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, ISBN 9780849385230, 1997.

[55] Boudot, F., Gaudry, P., Guillevic, A., Heninger, N., Thomé, E., Zimmermann, P., "Comparing the Difficulty of Factorization and Discrete Logarithm: A 240-Digit Experiment". *Advances in Cryptology – CRYPTO 2020. Lecture Notes in Computer Science*, vol 12171. Springer, Cham, 2020. https://doi.org/10.1007/978-3-030-56880-1_3.

[56] D. Kahn. *The Codebreakers – The Story of Secret Writing*. Macmillan, New York, ISBN 9780025604605, 1967.

[57] Singh, Simon. *The Code Book – The Secret History of Codes and Code-Breaking*. Fourth State, London, ISBN 9781857028799, 1999.

[58] Image of a "Scytale" device. Available at: https://commons.wikimedia.org/w/index.php?title=File:Skytale.png&oldid=4477453. Accessed 21 April 2023.

[59] Image representing the Caesar cipher. Available at: https://commons.wikimedia.org/wiki/File:Caesar_cipher_left_shift_of_3.svg. Accessed 21 April 2023.

[60] Picture of a metallic cipher disk. Available at: https://commons.wikimedia.org/wiki/File:CipherDisk2000.jpg. Accessed 21 April 2023.

[61] Image of the "Tabula Recta". Available at: https://commons.wikimedia.org/wiki/File:Vigen%C3%A8re_square_shading.svg. Accessed 21 April 2023.

[62] P. C. J. Hill, "Vigenère Through Shannon to Planck – A Short History of Electronic Cryptographic Systems". *IEEE History of Telecommunications Conference*, pp. 41–46, 2008. https://doi.org/10.1109/HISTELCON.2008.4668712.

[63] Dirk Rijmenants, Cipher Machines and Cryptology Website. Available at: https://www.ciphermachinesandcryptology.com/. Accessed 21 April 2023.

[64] Picture of the Enigma cryptography machine. Available at: https://commons.wikimedia.org/wiki/File:EnigmaMachine.jpg. Accessed 21 April 2023.

[65] 3D image of the Enigma cryptography machine rotors. Available at: https://commons.wikimedia.org/wiki/File:Enigma_rotor_set.png. Accessed 21 April 2023.

[66] Picture of the Enigma cryptography machine plug board. Available at: https://commons.wikimedia.org/wiki/File:Enigma-plugboard.jpg. Accessed 21 April 2023.

[67] Auguste Kerckhoffs, "La Cryptographie Militaire". *Journal des Sciences Militaires*, vol. 9, pp. 5–38, 1883.

[68] C. E. Shannon, "Communication Theory of Secrecy Systems". *The Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, October 1949. https://doi.org/10.1002/j.1538-7305.1949.tb00928.x.

[69] W. Diffie and M. Hellman, "New Directions in Cryptography". *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, November 1976. https://doi.org/10.1109/TIT.1976.1055638.

[70] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978. https://doi.org/10.1145/359340.359342.

[71] Bravyi S., Dial O., Gambetta J. M., Gil D., and Nazario Z., "The Future of Quantum Computing with Superconducting Qubits". *Journal of Applied Physics*. vol. 132, 160902, 2022. https://doi.org/10.1063/5.0082975.

[72] A. V. Sergienko (editor). *Quantum Communications and Cryptography*. CRC Press (Taylor & Francis), ISBN 9780849336843, 2006.

[73] Bennett, Charles H. et al., "Quantum Cryptography". Scientific American, vol. 267, no. 4, pp. 50–57, 1992. https://www.jstor.org/stable/24939253.

[74] Bennett, C. H., Bessette, F., Brassard, G., et al., "Experimental Quantum Cryptography". *Journal of. Cryptology*, vol. 5, pp. 3–28, 1992. https://doi.org/10.1007/BF00191318.

[75] Ralph C. Merkle, "Secure Communications Over Insecure Channels". *Communications of the ACM*, vol. 21, no. 4, pp.294–299, 1978. https://doi.org/10.1145/359460.359473.

[76] "Understanding Quantum Cryptography". *ID Quantique*, white paper, 2020. Available at: https://www.quantumcommshub.net/wp-content/uploads/2020/09/Understanding-Quantum-Cryptography_White-Paper.pdf. Accessed 21 April 2023.

[77] Bennett, C. H. and Brassard, G., "Quantum Cryptography: Public Key Distribution and Coin Tossing". *Proceedings of the International Conference on Computers, Systems & Signal Processing*, Bangalore, India, pp. 175-17, 1984. https://doi.org/10.48550/arXiv.2003.06557.

[78] Z. Yuan et al., "10-Mb/s Quantum Key Distribution". *Journal of Lightwave Technology*, vol. 36, no. 16, pp. 3427–3433, 2018. https://doi.org/10.1109/JLT.2018.2843136.

[79] MagiQ company website. Available at: https://www.magiqtech.com/. Accessed 21 April 2023.

[80] KETS company website. Available at: https://kets-quantum.com/. Accessed 21 April 2023.

[81] ID Quantique company website. Available at: https://www.idquantique.com/. Accessed 21 April 2023.

[82] Toshiba company website. Available at: https://www.global.toshiba/ww/products-solutions/security-ict/qkd.html. Accessed 21 April 2023.

[83] OpenQKD project website. Available at: https://openqkd.eu/. Accessed 21 April 2023.

[84] European Telecommunications Standards Institute (ETSI). Industry Specification Group on Quantum Key Distribution website. Available at: https://www.etsi.org/committee/qkd. Accessed 21 April 2023.

[85] "ID Quantique, SK Telecom & Nokia Secure Optical Transport System Using Quantum Key Distribution (QKD)". Press release, 2018. Available at: https://www.idquantique.com/idq-sk-telecom-nokia-secure-optical-transport-system-using-qkd/. Accessed 21 April 2023.

[86] "ADVA to play key role in OPENQKD project pioneering market-ready quantum-safe communications". Press release, 2020. Available at: https://www.adva.com/en/newsroom/press-releases/20200128-adva-to-play-key-role-in-openqkd-project. Accessed 21 April 2023.

[87] Joan Daemen and Vincent Rijmen, "AES Proposal: Rijndael". Available at: https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf. Accessed 21 April 2023.

[88] "Advanced Encryption Standard (AES)". *FIPS Publication 197*, National Institute of Standards and Technology (NIST), 2001.

[89] Joan Daemen and Vincent Rijmen. *The Design of Rijndael – The Advanced Encryption Standard (AES)*. Second edition, Springer, ISBN 9783662607688, 2020. https://doi.org/10.1007/978-3-662-60769-5.

[90] Image representing the AES algorithm Substitute Bytes transformation. Available at: https://commons.wikimedia.org/wiki/File:AES-SubBytes.svg. Accessed 21 April 2023.

[91] Image representing the AES algorithm Shift Rows transformation. Available at: https://commons.wikimedia.org/wiki/File:AES-ShiftRows.svg. Accessed 21 April 2023.

[92] Image representing the AES Algorithm Mix Columns transformation. Available at: https://commons.wikimedia.org/wiki/File:AES-MixColumns.svg. Accessed 21 April 2023.

[93] Image representing the AES algorithm Add Round Key transformation. Available at: https://commons.wikimedia.org/wiki/File:AES-AddRoundKey.svg. Accessed 21 April 2023.

[94] Morris Dworkin, "Recommendation for Block Cipher Modes of Operation: Methods and Techniques", *NIST Special Publication 800-38A*, 2001.

[95] Rolf Oppliger. *Cryptography 101 – From Theory to Practice*. Artech House, ISBN 9781630818463, 2021.

[96] S. Goldwasser and S. Micali, "Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information". *Proc. 14th Symposium on Theory of Computing*, pp. 365–377, 1982. https://doi.org/10.1145/800070.802212.

[97] Image of the Tux penguin, Linux mascot created in 1996 by Larry Ewing with GIMP. lewing@isc.tamu.edu. Available at: https://commons.wikimedia.org/wiki/File:Tux.jpg. Accessed 21 April 2023.

[98] Image derived from the Tux penguin, Linux mascot created in 1996 by Larry Ewing with GIMP, encrypted with the ECB mode of operation. lewing@isc.tamu.edu. Available at: https://commons.wikimedia.org/wiki/File:Tux_ecb.jpg. Accessed 21 April 2023.

[99] Image derived from the Tux penguin, Linux mascot created in 1996 by Larry Ewing with GIMP, encrypted with the CBC or CTR mode of operation. lewing@isc.tamu.edu. Available at: https://commons.wikimedia.org/wiki/File:Tux_secure.jpg. Accessed 21 April 2023.

[100] Morris Dworkin, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC". *NIST Special Publication 800-38D*, 2007.

[101] A. Satoh, "High-Speed Parallel Hardware Architecture for Galois Counter Mode". *IEEE International Symposium on Circuits and Systems*, New Orleans, LA, USA, pp. 1863–1866, 2007. https://doi.org/10.1109/ISCAS.2007.378278.

[102] E. Savaş and Ç. K. Koç, "Finite Field Arithmetic for Cryptography". *IEEE Circuits and Systems Magazine*, vol. 10, no. 2, pp. 40–56, 2010. https://doi.org/10.1109/MCAS.2010.936785.

[103] T. Chen, W. Huo, and Z. Liu, "Design and Efficient FPGA Implementation of Ghash Core for AES-GCM". *International Conference on Computational Intelligence and Software Engineering*, Wuhan, China, pp. 1–4, 2010. https://doi.org/10.1109/CISE.2010.5676905.

[104] McGrew D. A. and Viega J., "The Security and Performance of the Galois/Counter Mode (GCM) of Operation". *Progress in Cryptology – INDOCRYPT 2004*. *Lecture Notes in Computer Science*, vol. 3348, pp. 343–355. Springer, Berlin, Heidelberg, 2004. https://doi.org/10.1007/978-3-540-30556-9_27.

[105] McGrew David and Viega John, "The Security and Performance of the Galois/Counter Mode of Operation (Full Version)". *IACR Cryptology ePrint Archive*, report 2004/193. Available at: https://eprint.iacr.org/2004/193.pdf. Accessed 21 April 2023.

[106] Antoine Joux, "Authentication Failures in NIST Version of GCM". National Institute of Standards and Technology (NIST), 2006. Available at: https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/joux_comments.pdf. Accessed 21 April 2023.

[107] Niels Ferguson, "Authentication Weaknesses in GCM". Microsoft Corporation, 2005. Available at: https://csrc.nist.rip/CSRC/media/Projects/Block-Cipher-Techniques/documents/BCM/Comments/CWC-GCM/Ferguson2.pdf. Accessed 21 April 2023.

[108] Venkateswaran Kasirajan. *Fundamentals of Quantum Computing – Theory and Practice*. Springer, ISBN 9783030636883, 2021. https://doi.org/10.1007/978-3-030-63689-0.

[109] Daniel D. Stancil and Gregory T. Byrd. *Principles of Superconducting Quantum Computers*. John Wiley & Sons, Inc., ISBN 9781119750727, 2022.

[110] Thomas Wong. *Introduction to Classical and Quantum Computing*. Rooted Grove, ISBN 9798985593105, 2022.

[111] Arute, F. et al., “Quantum Supremacy Using a Programmable Superconducting Processor”. *Nature*, vol. 574, pp. 505–510, 2019. https://doi.org/10.1038/s41586-019-1666-5.

[112] Pan, F., Chen, K., and Zhang, P., “Solving the Sampling Problem of the Sycamore Quantum Circuits”. *Physical Review Letters*, vol. 129, no. 9, pp. 090502, 2022. https://doi.org/10.1103/PhysRevLett.129.090502.

[113] Zhong, H.-S. et al., “Quantum Computational Advantage Using Photons”. *Science*, vol. 370, no. 6523, pp. 1460–1463, 2020. https://doi.org/10.1126/science.abe8770.

[114] Madsen, L. S., Laudenbach, F., Askarani, M. F., et al., “Quantum Computational Advantage with a Programmable Photonic Processor”. *Nature*, vol. 606, pp. 75–81, 2022. https://doi.org/10.1038/s41586-022-04725-x.

[115] Ladd, T., Jelezko, F., Laflamme, R., et al., “Quantum computers”. *Nature*, vol. 464, pp. 45–53, 2010. https://doi.org/10.1038/nature08812.

[116] Sen, A. and Rezai, K., “Comparing Qubit Platforms in the Race to Feasible Quantum Computing”. *Journal of Student Research*, vol. 10, no. 4, 2021. https://doi.org/10.47611/jsrhs.v10i4.2236.

[117] P. W. Shor, “Algorithms for Quantum Computation: Discrete Logarithms and Factoring”. *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134, 1994. https://doi.org/10.1109/SFCS.1994.365700.

[118] Lov K. Grover, “A Fast Quantum Mechanical Algorithm for Database Search”. *Proceedings of the 28th ACM Symposium on Theory of Computing (STOC)*, pp. 212–219, 1996. https://doi.org/10.1145/237814.237866. Updated version available at https://doi.org/10.48550/arXiv.quant-ph/9605043.

[119] Michele Mosca and Marco Piani, “2021 Quantum Threat Timeline Report”. *Global Risk Institute*, 2022. Available at: https://globalriskinstitute.org/publications/2021-quantum-threat-timeline-report/. Accessed 21 April 2023.

[120] National Academies of Sciences, Engineering, and Medicine. *Quantum Computing – Progress and Prospects*. The National Academies Press, ISBN 9780309479691, 2019. https://doi.org/10.17226/25196.

[121] IBM 2022 Development Roadmap. Available at: https://www.ibm.com/quantum/roadmap. Accessed 21 April 2023.

[122] Bernstein, D., Lange, T., “Post-Quantum Cryptography”. *Nature*, vol. 549, pp. 188–194, 2017. https://doi.org/10.1038/nature23461.

[123] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen (editors). *Post-Quantum Cryptography*. Springer Berlin, Heidelberg, ISBN 9783540887010, 2009. https://doi.org/10.1007/978-3-540-88702-7.

[124] Joseph, D., Misoczki, R., Manzano, M., et al., "Transitioning Organizations to Post-Quantum Cryptography". *Nature*, vol. 605, pp. 237–243, 2022. https://doi.org/10.1038/s41586-022-04623-2.

[125] "Transitioning to a Quantum-Secure Economy". *World Economic Forum*, white paper, 2022. Available at: https://www.weforum.org/whitepapers/transitioning-to-a-quantum-secure-economy. Accessed 21 April 2023.

[126] "IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language". *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, February 2018. https://doi.org/10.1109/IEEESTD.2018.8299595.

[127] R. Ginosar, "Metastability and Synchronizers: A Tutorial". *IEEE Design & Test of Computers*, vol. 28, no. 5, pp. 23–35, September-October 2011. https://doi.org/10.1109/MDT.2011.113.

[128] Mohit Arora. *The Art of Hardware Architecture – Design Methods and Techniques for Digital Circuits*. Springer, ISBN 9781461403968, 2012. https://doi.org/10.1007/978-1-4614-0397-5.

[129] ModelSim® simulation tool. Available at: https://eda.sw.siemens.com/en-US/ic/modelsim/. Accessed 21 April 2023.

[130] Lawrence E. Bassham III, "The Advanced Encryption Standard Algorithm Validation Suite (AESAVS)". National Institute of Standards and Technology (NIST), 2002.

[131] "Universal Verification Methodology (UVM) 1.2 User's Guide". Accellera Systems Initiative, 2015. Available at: https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf. Accessed 21 April 2023.

[132] Pankaj S. Vitankar and A. K. Kureshi, "UVM Architecture for Verification". *International Journal of Electronics and Communication Engineering & Technology*, vol. 7, no. 3, pp. 29–37, 2016.

[133] A. Salvador, D. Carvalho, C. Nakandakare, E. Mobilon, J. C. de Oliveira, and D. S. Arantes, "100 Gbit/s FEC for OTN Protocol: Design Architecture and Implementation Results". *International Telecommunications Symposium (ITS)*, São Paulo, Brazil, pp. 1–5, 2014. https://doi.org/10.1109/ITS.2014.6947951.

[134] Eduardo Mobilon and Dalton Soares Arantes, "100 Gbit/s AES-GCM Cryptography Engine for Optical Transport Network Systems: Architecture, Design and 40 nm Silicon Prototyping". *Microelectronics Journal*, vol. 116, 105229, ISSN 0026-2692, 2021. https://doi.org/10.1016/j.mejo.2021.105229.

[135] "Considerations on OTNsec: GEP and OEP". *Contribution SG15-C.1683*, ITU-T Study Group 15, 2020.

[136] "OTNSec at the ODU Layer". *Contribution SG15-C-1841*, ITU-T Study Group 15, 2020.

[137] "Further Considerations on OTNsec Using L1GCM Based OTN Encryption Procedure". *Contribution SG15-C.1984*, ITU-T Study Group 15, 2020.

[138] "Overhead Usage for OxUsec". *Contribution SG15-C-2083*, ITU-T Study Group 15, 2020.

[139] "Further OTN Security Considerations". *Contribution SG15-C.2483*, ITU-T Study Group 15, 2021.

[140] "Optical Transport Network Security". *ITU-T Series G Supplement 76*, 2021.

[141] "Flexible OTN Short-Reach Interfaces". *ITU-T recommendation G.709.1/Y.1331.1*, 2018.

[142] "Flexible OTN Short-Reach Interfaces". *ITU-T recommendation G.709.1/Y.1331.1*, Amendment 2, 2020.

Appendix A

CONFIGURATION REGISTER MEMORY MAP

Table A.1 shows the 100G AES-GCM Cryptography Engine configuration register memory map. They can be accessed by the software layer through the CPU interface, according to their types: read only (R), write only (W), or read and write (RW).

Each register is further described in the subsequent sections.

#	Name	Address	Type	Description
1	ips_access_tst	0x00	RW	16-bit dummy register provided for writing and reading access tests.
2	rtl_version	0x01	R	Shows the implementation revision and version data of the Crypto4OTN block.
3	blk_rst	0x02	W	Used to reset the Crypto4OTN block.
4	blk_ctr	0x03	RW	Used to control the main functionalities of the Crypto4OTN block.
5	(RESERVED)	0x07 – 0x04	–	Reserved for future use.
6	tx_intr_status	0x08	R	TX processor status of interrupt events.
7	tx_intr_mask	0x09	RW	TX processor interrupt request mask.
8	tx_intr_hist	0x0a	RW	TX processor history of interrupt events.
9	rx_intr_status	0x0b	R	RX processor status of interrupt events.
10	rx_intr_mask	0x0c	RW	RX processor interrupt request mask.
11	rx_intr_hist	0x0d	RW	RX processor history of interrupt events.
12	tx_proc_ctrl	0x0e	RW	TX processor control.
13	tx_session_state	0x0f	RW	TX processor crypto session state control.
14	tx_session_status	0x10	R	TX processor crypto session status.
15	tx_session_period	0x12 – 0x11	RW	TX processor session period.
16	tx_session_threshold	0x14 – 0x13	RW	TX processor session period threshold.
17	tx_key_reuse_period	0x16 – 0x15	RW	TX processor key reuse period.
18	tx_conseq_act	0x17	RW	TX processor consequent actions.
19	tx_key_0	0x27 – 0x18	W	TX processor crypto session 256-bit key 0.

#	Name	Address	Type	Description
20	tx_key_1	0x37 – 0x28	W	TX processor crypto session 256-bit key 1.
21	tx_key_change	0x38	W	TX processor key change control.
22	tx_aad_buffer	0x3a – 0x39	RW	TX processor additional authenticated data buffer.
23	tx_aad_capture	0x3b	W	TX processor additional authenticated data buffer capture control.
24	tx_iv_csid	0x3d – 0x3c	RW	TX processor IV crypto session ID.
25	tx_iv_cbid	0x3f – 0x3e	R	TX processor IV crypto block ID.
26	tx_iv_cpid	0x41 – 0x40	R	TX processor IV crypto packet ID.
27	tx_calc_tag	0x49 – 0x42	R	TX processor calculated authentication TAG.
28	tx_repl_signal_pattern	0x4a	RW	TX processor replacement signal pattern.
29	tx_cnt_latch	0x4b	W	TX processor counter latch control.
30	rx_proc_ctrl	0x4c	RW	RX processor control.
31	rx_session_state	0x4d	RW	RX processor crypto session state control.
32	rx_session_status	0x4e	R	RX processor crypto session status.
33	rx_session_period	0x50 – 0x4f	RW	RX processor session period.
34	rx_session_threshold	0x52 – 0x51	RW	RX processor session period threshold.
35	rx_key_reuse_period	0x54 – 0x53	RW	RX processor key reuse period.
36	rx_conseq_act	0x55	RW	RX processor consequent actions.
37	rx_key_0	0x65 – 0x56	W	RX processor crypto session 256-bit key 0.
38	rx_key_1	0x75 – 0x66	W	RX processor crypto session 256-bit key 1.
39	rx_aad_buffer	0x77 – 0x76	R	RX processor additional authenticated data buffer.
40	rx_iv_csid	0x79 – 0x78	R	RX processor IV crypto session ID.
41	rx_iv_cbid	0x7b – 0x7a	R	RX processor IV crypto block ID.
42	rx_iv_cpid	0x7d – 0x7c	R	RX processor IV crypto packet ID.
43	rx_calc_tag	0x85 – 0x7e	R	RX processor calculated authentication TAG.
44	rx_rcvd_tag	0x8d – 0x86	R	RX processor received authentication TAG.
45	rx_cnt_latch	0x8e	W	RX processor counter latch control.
46	rx_cs_cnt_clear	0x8f	W	RX processor crypto session counter clear control.
47	rx_cs_cnt	0x91 – 0x90	R	RX processor crypto session counter.
48	rx_cp_cnt_clear	0x92	W	RX processor crypto packet counter clear control.
49	rx_cp_cnt	0x95 – 0x93	R	RX processor crypto packet counter.
50	rx_loa_window_mask	0x99 – 0x96	RW	RX processor loss of authentication window mask.
51	rx_loa_cnt_threshold	0x9a	RW	RX processor loss of authentication counter threshold.
52	rx_loa_fail_cnt_clear	0x9b	W	RX processor loss of authentication failure counter clear control.
53	rx_loa_fail_cnt	0x9d – 0x9c	R	RX processor loss of authentication failure counter.

#	Name	Address	Type	Description
54	rx_tag_fail_cnt_clear	0x9e	W	RX processor authentication TAG failure counter clear control.
55	rx_tag_fail_cnt	0xa0 – 0x9f	R	RX processor authentication TAG failure counter.
56	rx_csks_bit_cnt_clear	0xa1	W	RX processor CSKS bit counter clear control.
57	rx_csks_bit0_cnt	0xa3 – 0xa2	R	RX processor CSKS bit 0 counter.
58	rx_csks_bit1_cnt	0xa5 – 0xa4	R	RX processor CSKS bit 1 counter.
59	rx_csks_hist_cnt_clear	0xa6	W	RX processor CSKS histogram counter clear control.
60	rx_csks_bit0_hist_cnt	0xac – 0xa7	R	RX processor CSKS bit 0 histogram counter.
61	rx_csks_bit1_hist_cnt	0xb2 – 0xad	R	RX processor CSKS bit 1 histogram counter.
62	(RESERVED)	0xff – 0xb3	–	Reserved for future use.

Table A.1 – 100G AES-GCM Cryptography Engine configuration register map.

A.1.1 IPS_ACCESS_TST – IPS ACCESS TEST

Used to write and read 16-bit data into/from the block, in order to check the IPS data bus access. This is for testing purposes only and does not affect the block functionality.

Bit	Name	Description
15:0	IPS_TST	Data used to test the IPS access. Both hard and soft reset forces all zeros asynchronously by *ipg_hard_async_reset_b* and *ipg_soft_reset_sync_b*, respectively.

Table A.2 – Description of the register ips_access_tst.

A.1.2 RTL_VERSION – CRYPTO4OTN RTL VERSION

Stores version and revision information for the Crypto4OTN implementation.

The format is: 8 bits for major number and 8 bits for minor number. This register starts with 0x0100 and is updated only when a new version of the block is released for tape-out or as an IP.

Bit	Name	Description
15:8	RTL_VER	Stores a binary number that represents the version of the Crypto4OTN RTL implementation. This value is not affected by soft or hard reset.
7:0	RTL_REV	Stores a binary number that represents the revision of the Crypto4OTN RTL implementation. This value is not affected by soft or hard reset.

Table A.3 – Description of the register rtl_version.

A.1.3 BLK_RST – CRYPTO4OTN BLOCK RESET

Used to reset the Crypto4OTN block, without affecting configuration register values.

Bit	Name	Description
8	RX_RST	RX processor reset. It does not affect configuration register values. This bit is cleared by hardware. '0' – Normal operation. '1' – Reset.
0	TX_RST	TX processor reset. It does not affect configuration register values. This bit is cleared by hardware. '0' – Normal operation. '1' – Reset.

Table A.4 – Description of the register blk_rst.

A.1.4 BLK_CTR – BLOCK CONTROL

Controls the main functionalities of the Crypto4OTN block (TX and RX data paths).

Bit	Name	Description
15:12	RX_FLOW_CTRL_LEVELTHR	RX data path flow control FIFO level threshold setting. Used to trigger data request assertion in "pull" mode. Minimum value is RX_FLOW_CTRL_LEVELTHR = 2. FIFO level threshold value = 4 × RX_FLOW_CTR_LEVELTHR.
11	RX_FLOW_CTRL_MODE	RX data path flow control operation mode. '0' – Push mode. '1' – Pull mode.
10	LINE_LOOP	Line side far-end loopback mode control. '0' – Line side far-end loopback disabled. '1' – Line side far-end loopback enabled, without normal traffic disruption.
9	RX_BYP	RX data path bypass control. '0' – RX data path bypass disabled. '1' – RX data path bypass enabled.
8	RX_ENA	RX data path enable control. '0' – RX data path disabled and all output ports de-asserted (at logic '0'). '1' – RX data path enabled.
7:4	TX_FLOW_CTRL_LEVELTHR	TX data path flow control FIFO level threshold setting. Used to trigger data request assertion in "pull" mode. Minimum value is TX_FLOW_CTRL_LEVELTHR = 2. FIFO level threshold value = 4 × TX_FLOW_CTR_LEVELTHR.
3	TX_FLOW_CTRL_MODE	TX data path flow control operation mode. '0' – Push mode. '1' – Pull mode.
2	CLI_LOOP	Client side far-end loopback mode control. '0' – Client side far-end loopback disabled. '1' – Client side far-end loopback enabled, without normal traffic disruption.
1	TX_BYP	TX data path bypass control. '0' – TX data path bypass disabled. '1' – TX data path bypass enabled.
0	TX_ENA	TX data path enable control. '0' – TX data path disabled and all output ports de-asserted (at logic '0'). '1' – TX data path enabled.

Table A.5 – Description of the register blk_ctr.

A.1.5 RESERVED – RESERVED REGISTER

Bit	Name	Description
15:0	RES	Reserved bits.

Table A.6 – Description of the register (RESERVED).

A.1.6 TX_INTR_STATUS – TX PROCESSOR INTERRUPT STATUS

Status of the interrupt signal sources in the TX processor. All bits are active-high.

Bit	Name	Description
7	FLOW_FIFO_FULL	TX processor flow control FIFO full indication.
6	FLOW_FIFO_EMPTY	TX processor flow control FIFO empty indication.
5	MFS_WDOG	TX processor MFS watchdog expired.
4	IPY_SSF	*ipy_ssf_tx* port asserted.
3	IPY_TSF	*ipy_tsf_tx* port asserted.
2	SE_EXPD	TX processor crypto session expired (CBID counter > *tx_session_period*).
1	SE_AEXP	TX processor crypto session about to expire (CBID counter > *tx_session_threshold*).
0	SE_CSD	TX processor crypto session closed (SESSION_STATUS = 0).

Table A.7 – Description of the register tx_intr_status.

A.1.7 TX_INTR_MASK – TX PROCESSOR INTERRUPT MASK

Each bit masks the corresponding interrupt request in the *tx_intr_hist* register.

Asserted bits disable (mask) the corresponding interrupt requests.

Bit	Name	Description
7	FLOW_FIFO_FULL	TX processor flow control FIFO full indication.
6	FLOW_FIFO_EMPTY	TX processor flow control FIFO empty indication.
5	MFS_WDOG	TX processor MFS watchdog expired.
4	IPY_SSF	*ipy_ssf_tx* port asserted.
3	IPY_TSF	*ipy_tsf_tx* port asserted.
2	SE_EXPD	TX processor crypto session expired (CBID counter > *tx_session_period*).
1	SE_AEXP	TX processor crypto session about to expire (CBID counter > *tx_session_threshold*).
0	SE_CSD	TX processor crypto session closed (SESSION_STATUS = 0).

Table A.8 – Description of the register tx_intr_mask.

A.1.8 TX_INTR_HIST – TX PROCESSOR INTERRUPT REQUEST HISTORY

Stores interrupts requested by their individual sources (and masked by *tx_intr_mask* register). An asserted valid bit generates an interrupt request if its corresponding bit mask is de-asserted in the *tx_intr_mask* register.

Once a bit is active, it can only be deactivated by writing '1' into the corresponding bit or through a global reset. The *blk_rst* register does not deactivate the bits in this register.

Bit	Name	Description
7	FLOW_FIFO_FULL	TX processor flow control FIFO full indication.
6	FLOW_FIFO_EMPTY	TX processor flow control FIFO empty indication.
5	MFS_WDOG	TX processor MFS watchdog expired.
4	IPY_SSF	*ipy_ssf_tx* port asserted.
3	IPY_TSF	*ipy_tsf_tx* port asserted.
2	SE_EXPD	TX processor crypto session expired (CBID counter > *tx_session_period*).
1	SE_AEXP	TX processor crypto session about to expire (CBID counter ≥ *tx_session_threshold*).
0	SE_CSD	TX processor crypto session closed (SESSION_STATUS = 0).

Table A.9 – Description of the register tx_intr_hist.

A.1.9 RX_INTR_STATUS – RX PROCESSOR INTERRUPT STATUS

Status of the interrupt signal sources in the RX processor. All bits are active-high.

Bit	Name	Description
10	LOA	Loss of Authentication failure detected in the counting sliding window.
9	TAG_FAIL	Authentication failure detected in the received crypto packet.
8	SE_KEY_CGD	RX processor crypto session key changed.
7	FLOW_FIFO_FULL	RX processor flow control FIFO full indication.
6	FLOW_FIFO_EMPTY	RX processor flow control FIFO empty indication.
5	MFS_WDOG	RX processor MFS watchdog expired.
4	IPY_SSF	*ipy_ssf_rx* port asserted.
3	IPY_TSF	*ipy_tsf_rx* port asserted.
2	SE_EXPD	RX processor crypto session expired (*rx_iv_cbid* > *rx_session_period*).
1	SE_AEXP	RX processor crypto session about to expire (*rx_iv_cbid* > *rx_session_threshold*).
0	SE_CSD	RX processor crypto session closed (SESSION_STATUS = 0).

Table A.10 – Description of the register rx_intr_status.

A.1.10 RX_INTR_MASK – RX PROCESSOR INTERRUPT MASK

Each bit masks the corresponding interrupt request in the *rx_intr_hist* register.

Asserted bits disable (mask) the corresponding interrupt requests.

Bit	Name	Description
10	LOA	Loss of Authentication failure detected in the counting sliding window.
9	TAG_FAIL	Authentication failure detected in the received crypto packet.
8	SE_KEY_CGD	RX processor crypto session key changed.
7	FLOW_FIFO_FULL	RX processor flow control FIFO full indication.
6	FLOW_FIFO_EMPTY	RX processor flow control FIFO empty indication.
5	MFS_WDOG	RX processor MFS watchdog expired.

Bit	Name	Description
4	IPY_SSF	*ipy_ssf_rx* port asserted.
3	IPY_TSF	*ipy_tsf_rx* port asserted.
2	SE_EXPD	RX processor crypto session expired (*rx_iv_cbid* > *rx_session_period*).
1	SE_AEXP	RX processor crypto session about to expire (*rx_iv_cbid* > *rx_session_threshold*).
0	SE_CSD	RX processor crypto session closed (SESSION_STATUS = 0).

Table A.11 – Description of the register rx_intr_mask.

A.1.11 RX_INTR_HIST – RX PROCESSOR INTERRUPT REQUEST HISTORY

Stores interrupts requested by their individual sources (and masked by *rx_intr_mask* register). An asserted valid bit generates an interrupt request if its corresponding bit mask is de-asserted in the *rx_intr_mask* register.

Once a bit is active, it can only be deactivated by writing '1' into the corresponding bit or through a global reset. The *blk_rst* register does not deactivate the bits in this register.

Bit	Name	Description
10	LOA	Loss of Authentication failure detected in the counting sliding window.
9	TAG_FAIL	Authentication failure detected in the received crypto packet.
8	SE_KEY_CGD	RX processor crypto session key changed.
7	FLOW_FIFO_FULL	RX processor flow control FIFO full indication.
6	FLOW_FIFO_EMPTY	RX processor flow control FIFO empty indication.
5	MFS_WDOG	RX processor MFS watchdog expired.
4	IPY_SSF	*ipy_ssf_rx* port asserted.
3	IPY_TSF	*ipy_tsf_rx* port asserted.
2	SE_EXPD	RX processor crypto session expired (*rx_iv_cbid* > *rx_session_period*).
1	SE_AEXP	RX processor crypto session about to expire (*rx_iv_cbid* > *rx_session_threshold*).
0	SE_CSD	RX processor crypto session closed (SESSION_STATUS = 0).

Table A.12 – Description of the register rx_intr_hist.

A.1.12 TX_PROC_CTRL – TX PROCESSOR CONTROL

Controls the main functionalities of the TX processor.

Bit	Name	Description
15	FORCE_SSF	*ipy_ssf_tx* output port level control. '0' – The *ipy_ssf_tx* port is controlled by hardware. '1' – High level (asserted). The output is forced to '1'.
11	SCRAMBLER	ODU overhead scrambler enable control. Active-high. When enabled, encryption overhead bytes (TAG, AAD, and IV) are scrambled right before being inserted in the ODU overhead.
10	KDF	Key derivation function enable control. Active-high. When enabled, key values are scrambled before being used by the AES-GCM cryptography algorithm.

Bit	Name	Description
9:8	OP_MODE	Crypto session operation mode. ‘00’ – Authenticated encryption. ‘01’ – Authentication-only. ‘10’ – Encryption-only.
2	FORCE_REPLSIG	Replacement signal generation activation control. ‘0’ – Replacement signal generation is commanded by the TX processor controller. ‘1’ – Replacement signal generation is activated, outputting the data pattern selected by (REPL_SIG_SEL).
1	REPLSIG_ENA	Replacement signal generation enable control. ‘0’ – Replacement signal generation is disabled. ‘1’ – Replacement signal generation is enabled, outputting the data pattern selected by (REPLSIG_SEL) when requested by the TX processor controller (depending on the session state and consequent action).
0	REPLSIG_SEL	Replacement signal selection. ‘0’ – AIS-like pattern is generated as a replacement signal. ‘1’ – User defined pattern (*tx_repl_signal_pattern*) is generated as a replacement signal.

Table A.13 – Description of the register tx_proc_ctrl.

A.1.13 TX_SESSION_STATE – TX CRYPTO SESSION STATE CONTROL

Controls crypto session state in the TX processor.

Bit	Name	Description
0	SESSION_STATE	Crypto session state bit, controlled by software to change the state of the crypto session. 0’ – Force session state to “not established” (insecure). ‘1’ – Request session establishment.

Table A.14 – Description of the register tx_session_state.

A.1.14 TX_SESSION_STATUS – TX CRYPTO SESSION STATUS

Crypto session status in the TX processor.

Bit	Name	Description
1	ACTIVE_KEY	Active key indication. This bit indicates which key (0 or 1) is active and being used for encryption. ‘0’ – Key 0 is the active key. ‘1’ – Key 1 is the active key.
0	SESSION_STATUS	Crypto session status bit, controlled by hardware to indicate the status of the crypto session. ‘0’ – Session not established (insecure). ‘1’ – Session established (secure).

Table A.15 – Description of the register tx_session_status.

A.1.15 TX_SESSION_PERIOD – TX CRYPTO SESSION PERIOD

32-bit value that specifies the number of crypto blocks that make up a crypto session in the TX processor. Each crypto block is aligned with MFAS = 0, corresponding to 256 OTN frames and lasting for about 300 µs.

Bit	Name	Description
15:0	SESSION_PERIOD	32-bit value that specifies the number of crypto blocks that make up a crypto session.

Table A.16 – Description of the register tx_session_period.

A.1.16 TX_SESSION_THRESHOLD – TX SESSION PERIOD THRESHOLD

32-bit value that specifies the session period counter threshold above which the session-about-to-expire interrupt will be asserted in the TX processor.

Bit	Name	Description
15:0	SESSION_THRESHOLD	32-bit value that specifies the session period counter threshold.

Table A.17 – Description of the register tx_session_threshold.

A.1.17 TX_KEY_REUSE_PERIOD – TX KEY REUSE PERIOD

32-bit value that specifies the period (number of crypto blocks) for the reuse of the current key in the case of session expiration in the TX processor.

Bit	Name	Description
15:0	KEY_REUSE_PERIOD	32-bit value that specifies the period (number of crypto blocks) for the reuse of the current key in the case of session expiration.

Table A.18 – Description of the register tx_key_reuse_period.

A.1.18 TX_CONSEQ_ACT – TX CONSEQUENT ACTIONS

Defines consequent actions for the TX processor, which are applicable only for active crypto sessions (SESSION_STATUS = 1).

Bit	Name	Description
11:8	SESSION_EXP	Consequent action selection for the case of session expiration. '0000' – None. '0001' – Assert *ipy_ssf_tx* and close the crypto session in the current direction. '0010' – Assert *ipy_ssf_tx* and close both crypto sessions in the current and opposite directions. '0011' – Reuse current key during a limited period determined by *tx_key_reuse_period* register. When the key reuse period expires, assert *ipy_ssf_tx* and close the crypto session in the current direction. '0100' – Reuse current key during a limited period determined by *tx_key_reuse_period* register. When the key reuse period expires, assert *ipy_ssf_tx* and close both crypto sessions in the current and opposite directions. '0101' – Close the crypto session in the current direction. '0110' – Close both crypto sessions in the current and opposite directions. '0111' – Reuse current key during a limited period determined by *tx_key_reuse_period* register. When the key reuse period expires, close the crypto session in the current direction. '1000' – Reuse current key during a limited period determined by *tx_key_reuse_period* register. When the key reuse period expires, close both crypto sessions in the current and opposite directions.

Bit	Name	Description
1:0	IPY_TSF_TX	Consequent action selection for the case of *ipy_tsf_tx* signal detection. '00' – None. '01' – Close the crypto session in the current direction. '10' – Generate replacement signal and keep the crypto session active in the current direction.

Table A.19 – Description of the register tx_conseq_act.

A.1.19 TX_KEY_0 – TX CRYPTO SESSION KEY 0

Crypto session 256-bit key 0 used for encryption and authentication in the TX processor.

Bit	Name	Description
15:0	SESSION_KEY	Crypto session 256-bit key.

Table A.20 – Description of the register tx_key_0.

A.1.20 TX_KEY_1 – TX CRYPTO SESSION KEY 1

Crypto session 256-bit key 1 used for encryption and authentication in the TX processor.

Bit	Name	Description
15:0	SESSION_KEY	Crypto session 256-bit key.

Table A.21 – Description of the register tx_key_1.

A.1.21 TX_KEY_CHANGE – TX KEY CHANGE CONTROL

Controls crypto session key change in the TX processor.

Bit	Name	Description
0	KEY_CHANGE	Indicates that a new 256-bit key is ready in the selected register (key 0 or key 1) to be used for the next crypto session and triggers the key change process. A new CSID must also have been provided in the *tx_iv_csid* register before assertion of this bit. This bit is cleared by hardware. '0' – No session key change. '1' – Session key change request.

Table A.22 – Description of the register tx_key_change.

A.1.22 tx_aad_buffer – TX ADDITIONAL AUTHENTICATED DATA BUFFER

32-bit additional authenticated data provided by the software layer in the TX processor.

Bit	Name	Description
15:0	AAD	32-bit additional authenticated data.

Table A.23 – Description of the register tx_aad_buffer.

A.1.23 tx_aad_capture – TX AAD CAPTURE CONTROL

Controls AAD value capturing from the corresponding register buffer for transmission in the next crypto packet in the TX processor.

Bit	Name	Description
0	AAD_CAPTURE	Indicates that a new 32-bit AAD value is ready in the corresponding register buffer to be captured for transmission in the next crypto packet. Active-high. This bit is cleared by hardware.

Table A.24 – Description of the register tx_aad_capture.

A.1.24 TX_IV_CSID – TX IV CRYPTO SESSION ID

32-bit crypto session identification number provided by the software layer in the TX processor to form the fixed field of the initialization vector. This CSID value is captured by hardware only right before the start of a new crypto session.

Bit	Name	Description
15:0	CSID	32-bit crypto session identification number.

Table A.25 – Description of the register tx_iv_csid.

A.1.25 TX_IV_CBID – TX IV CRYPTO BLOCK ID

32-bit crypto block identification number generated by hardware in the TX processor to form the invocation field of the initialization vector.

Bit	Name	Description
15:0	CBID	32-bit crypto block identification number.

Table A.26 – Description of the register tx_iv_cbid.

A.1.26 TX_IV_CPID – TX IV CRYPTO PACKET ID

24-bit crypto packet identification number generated by hardware in the TX processor to form the invocation field of the initialization vector.

Bit	Name	Description
15:0	CPID	24-bit crypto packet identification number.

Table A.27 – Description of the register tx_iv_cpid.

A.1.27 TX_CALC_TAG – TX CALCULATED AUTHENTICATION TAG

128-bit authentication TAG generated by hardware in the TX processor.

Bit	Name	Description
15:0	TAG	128-bit authentication TAG.

Table A.28 – Description of the register tx_calc_tag.

A.1.28 TX_REPL_SIGNAL_PATTERN – TX REPLACEMENT SIGNAL PATTERN

16-bit user defined pattern to be used as a replacement signal in the TX processor, according to the configuration of the *tx_proc_ctrl* register.

Bit	Name	Description
15:0	REPL_SIG_PATTERN	16-bit user defined pattern to be repeatedly concatenated to make up a replacement signal in the TX processor, according to the configuration of the *tx_proc_ctrl* register.

Table A.29 – Description of the register tx_repl_signal_pattern.

A.1.29 TX_CNT_LATCH – TX COUNTER LATCH CONTROL

Controls the latching circuits of all TX processor counters, triggering the data sampling process before reading operations.

Bit	Name	Description
0	CNT_LATCH	Sample and hold command for all TX processor counters. Active-high. This bit is cleared by hardware.

Table A.30 – Description of the register tx_cnt_latch.

A.1.30 RX_PROC_CTRL – RX PROCESSOR CONTROL

Controls the main functionalities of the RX processor.

Bit	Name	Description
15	FORCE_SSF	*ipy_ssf_rx* output port level control. '0' – The *ipy_ssf_rx* port is controlled by hardware. '1' – High level (asserted). The output is forced to '1'.
14	SSF_SE_ENA	*ipy_ssf_rx* output port response to the RX processor crypto session state. '0' – The *ipy_ssf_rx* port response is disabled. '1' – The *ipy_ssf_rx* port response is enabled. *ipy_ssf_rx* is asserted when the crypto session is inactive.
11	SCRAMBLER	ODU overhead scrambler enable control. Active-high. When enabled, encryption overhead bytes (TAG, AAD, and IV) are scrambled right after being extracted from the ODU overhead. Since data have being previously scrambled in the TX processor side, repeating this operation here corresponds to a descrambling function.

Bit	Name	Description
10	KDF	Key derivation function enable control. Active-high. When enabled, key values are scrambled before being used by the AES-GCM cryptography algorithm.
9:8	OP_MODE	Crypto session operation mode. '00' – Authenticated encryption. '01' – Authentication-only. '10' – Encryption-only.
7	FORCE_AUTAG	*auth_tag_matching* output port level control. '0' – The *auth_tag_matching* port is controlled by hardware. '1' – High level (asserted). The output is forced to '1' by the software layer.
6	AUTAG_ENA	*auth_tag_matching* output port enable control. '0' – The *auth_tag_matching* port is disabled (forced to '0'). '1' – The *auth_tag_matching* port is enabled.
1	AUTH_BLOCK	Authentication blocking when operating in "store-and-forward" mode. '0' – OPU data are forwarded regardless of the authentication TAG matching indication status, which is updated at every crypto packet. '1' – OPU data (overhead and payload) are replaced by zero in the case of a TAG mismatch.
0	AUTH_DELAY	Authentication delay when operating in "Authenticated Encryption" or "Authentication-only" modes. '0' – Cut-through mode. OPU data are available to the client side with no extra authentication delay. TAG matching indication comes after a delay equivalent to 6 ODU frames (hardware architecture pipelining not included). '1' – Store-and-forward mode. OPU data are stored and delivered to the client side together with a TAG matching indication only after TAG verification, with a delay equivalent to 6 ODU frames (hardware architecture pipelining not included).

Table A.31 – Description of the register rx_proc_ctrl.

A.1.31 RX_SESSION_STATE – RX CRYPTO SESSION STATE CONTROL

Controls crypto session state in the RX processor.

Bit	Name	Description
0	SESSION_STATE	Crypto session state bit, controlled by software to change the state of the crypto session. '0' – Force session state to "not established" (insecure). '1' – Request session establishment.

Table A.32 – Description of the register rx_session_state.

A.1.32 RX_SESSION_STATUS – RX CRYPTO SESSION STATUS

Crypto session status in the RX processor.

Bit	Name	Description
1	ACTIVE_KEY	Active key indication. This bit indicates which key (0 or 1) is active and being used for decryption. '0' – Key 0 is the active key. '1' – Key 1 is the active key.
0	SESSION_STATUS	Crypto session status bit, controlled by hardware to indicate the status of the crypto session. '0' – Session not established (insecure). '1' – Session established (secure).

Table A.33 – Description of the register rx_session_status.

A.1.33 RX_SESSION_PERIOD – RX CRYPTO SESSION PERIOD

32-bit value that specifies the number of crypto blocks that make up a crypto session in the RX processor. Each crypto block is aligned with MFAS = 0, corresponding to 256 OTN frames and lasting for about 300 µs.

Bit	Name	Description
15:0	SESSION_PERIOD	32-bit value that specifies the number of crypto blocks that make up a crypto session.

Table A.34 – Description of the register rx_session_period.

A.1.34 RX_SESSION_THRESHOLD – RX SESSION PERIOD THRESHOLD

32-bit value that specifies the session period counter threshold above which the session-about-to-expire interrupt will be asserted in the RX processor.

Bit	Name	Description
15:0	SESSION_THRESHOLD	32-bit value that specifies the session period counter threshold.

Table A.35 – Description of the register rx_session_threshold.

A.1.35 RX_KEY_REUSE_PERIOD – RX KEY REUSE PERIOD

32-bit value that specifies the period (number of crypto blocks) for the reuse of the current key in the case of session expiration in the RX processor.

Bit	Name	Description
15:0	KEY_REUSE_PERIOD	32-bit value that specifies the period (number of crypto blocks) for the reuse of the current key in the case of session expiration.

Table A.36 – Description of the register rx_key_reuse_period.

A.1.36 RX_CONSEQ_ACT – RX CONSEQUENT ACTIONS

Defines consequent actions for the RX processor, which are applicable only for active crypto sessions (SESSION_STATUS = 1).

Bit	Name	Description
11:8	SESSION_EXP	Consequent action selection for the case of session expiration. '0000' – None. '0001' – Assert *ipy_ssf_rx* and close the crypto session in the current direction. '0010' – Assert *ipy_ssf_rx* and close both crypto sessions in the current and opposite directions. '0011' – Reuse current key during a limited period determined by *rx_key_reuse_period* register. When the key reuse period expires, assert *ipy_ssf_rx* and close the crypto session in the current direction. '0100' – Reuse current key during a limited period determined by *rx_key_reuse_period* register. When the key reuse period expires, assert *ipy_ssf_rx* and close both crypto sessions in the current and opposite directions. '0101' – Close the crypto session in the current direction. '0110' – Close both crypto sessions in the current and opposite directions. '0111' – Reuse current key during a limited period determined by *rx_key_reuse_period* register. When the key reuse period expires, close the crypto session in the current direction. '1000' – Reuse current key during a limited period determined by *rx_key_reuse_period* register. When the key reuse period expires, close both crypto sessions in the current and opposite directions.
7:5	LOA	Consequent action selection for the case of loss of authentication (LOA). '000' – None. '001' – Close the crypto session in the current direction. '010' – Close both crypto sessions in the current and opposite directions. '011' – Assert *ipy_ssf_rx* and close the crypto session in the current direction. '100' – Assert *ipy_ssf_rx* and keep the crypto session active in the current direction. '101' – Assert *ipy_ssf_rx* and close both crypto sessions in the current and opposite directions.
4:2	TAG_FAIL	Consequent action selection for the case of TAG mismatch (TAG_FAIL). '000' – None. '001' – Close the crypto session in the current direction. '010' – Close both crypto sessions in the current and opposite directions. '011' – Assert *ipy_ssf_rx* and close the crypto session in the current direction. '100' – Assert *ipy_ssf_rx* and keep the crypto session active in the current direction. '101' – Assert *ipy_ssf_rx* and close both crypto sessions in the current and opposite directions.
1:0	IPY_TSF_RX	Consequent action selection for the case of *ipy_tsf_rx* signal detection. '00' – None. '01' – Close the crypto session in the current direction. '10' – Close both crypto sessions in the current and opposite directions.

Table A.37 – Description of the register rx_conseq_act.

A.1.37 RX_KEY_0 – RX CRYPTO SESSION KEY 0

Crypto session 256-bit key 0 used for decryption and authentication in the RX processor.

Bit	Name	Description
15:0	SESSION_KEY	Crypto session 256-bit key.

Table A.38 – Description of the register rx_key_0.

A.1.38 RX_KEY_1 – RX CRYPTO SESSION KEY 1

Crypto session 256-bit key 1 used for decryption and authentication in the RX processor.

Bit	Name	Description
15:0	SESSION_KEY	Crypto session 256-bit key.

Table A.39 – Description of the register rx_key_1.

A.1.39 RX_AAD_BUFFER – RX ADDITIONAL AUTHENTICATED DATA BUFFER

32-bit additional authenticated data extracted from the encryption overhead of the received crypto packet.

Bit	Name	Description
15:0	AAD	32-bit additional authenticated data.

Table A.40 – Description of the register rx_aad_buffer.

A.1.40 RX_IV_CSID – RX IV CRYPTO SESSION ID

32-bit crypto session identification number extracted from the encryption overhead of the received crypto packet. Before reading operations, data must be sampled using the *rx_cnt_latch* control register.

Bit	Name	Description
15:0	CSID	32-bit crypto session identification number.

Table A.41 – Description of the register rx_iv_csid.

A.1.41 RX_IV_CBID – RX IV CRYPTO BLOCK ID

32-bit crypto block identification number extracted from the encryption overhead of the received crypto packet. Before reading operations, data must be sampled using the *rx_cnt_latch* control register.

Bit	Name	Description
15:0	CBID	32-bit crypto block identification number.

Table A.42 – Description of the register rx_iv_cbid.

A.1.42 RX_IV_CPID – RX IV CRYPTO PACKET ID

24-bit crypto packet identification number extracted from the encryption overhead of the received crypto packet. Before reading operations, data must be sampled using the *rx_cnt_latch* control register.

Bit	Name	Description
15:0	CPID	24-bit crypto packet identification number.

Table A.43 – Description of the register rx_iv_cpid.

A.1.43 RX_CALC_TAG – RX CALCULATED AUTHENTICATION TAG

128-bit authentication TAG generated by hardware in the RX processor. Before reading operations, data must be sampled using the *rx_cnt_latch* control register.

Bit	Name	Description
15:0	TAG	128-bit authentication TAG.

Table A.44 – Description of the register rx_calc_tag.

A.1.44 RX_RCVD_TAG – RX RECEIVED AUTHENTICATION TAG

128-bit authentication TAG extracted from the encryption overhead of the received crypto packet. Before reading operations, data must be sampled using the *rx_cnt_latch* control register.

Bit	Name	Description
15:0	TAG	128-bit authentication TAG.

Table A.45 – Description of the register rx_rcvd_tag.

A.1.45 RX_CNT_LATCH – RX COUNTER LATCH CONTROL

Controls the latching circuits of all RX processor counters, as well as *rx_iv_csid*, *rx_iv_cbid*, *rx_iv_cpid*, *rx_calc_tag*, and *rx_rcvd_tag* registers, triggering the data sampling process before reading operations.

Bit	Name	Description
0	CNT_LATCH	Sample and hold command for all RX processor counters. Active-high. This bit is cleared by hardware.

Table A.46 – Description of the register rx_cnt_latch.

A.1.46 RX_CS_CNT_CLEAR – RX CRYPTO SESSION COUNTER CLEAR

Crypto session counter clear control.

Bit	Name	Description
0	CLEAR	Counter clear control. This bit is cleared by hardware. '0' – Normal counter operation. '1' – Counter clear.

Table A.47 – Description of the register rx_cs_cnt_clear.

A.1.47 RX_CS_CNT – RX CRYPTO SESSION COUNTER

32-bit non-wrap-around crypto session counter. It indicates the number of crypto session establishment events, up to 2^{32}.

Bit	Name	Description
15:0	CS_CNT	32-bit non-wrap-around crypto session counter. It is incremented in the '0' → '1' transitions of SESSION_STATUS and after key changes.

Table A.48 – Description of the register rx_cs_cnt.

A.1.48 RX_CP_CNT_CLEAR – RX CRYPTO PACKET COUNTER CLEAR

Crypto packet counter clear control.

Bit	Name	Description
0	CLEAR	Counter clear control. This bit is cleared by hardware. '0' – Normal counter operation. '1' – Counter clear.

Table A.49 – Description of the register rx_cp_cnt_clear.

A.1.49 RX_CP_CNT – RX CRYPTO PACKET COUNTER

38-bit non-wrap-around crypto packet counter. It indicates the number of crypto packets within an active crypto session, up to $2^{32} \times 64 = 2^{38}$.

Bit	Name	Description
15:0	CP_CNT	38-bit non-wrap-around crypto packet counter. It is incremented when SESSION_STATUS = 1 and MFAS[1:0] = 0 and is cleared at the beginning of crypto sessions (after key changes).

Table A.50 – Description of the register rx_cp_cnt.

A.1.50 RX_LOA_WINDOW_MASK – RX LOSS OF AUTHENTICATION WINDOW MASK

64-bit value that masks the 64-bit sliding window inside which authentication failures are counted to assert loss of authentication (LOA) indication.

Bit	Name	Description
15:0	LOA_WINDOW_MASK	64-bit value masking the LOA counting sliding window. MSB corresponds to the current crypto packet TAG matching status (TAG_FAIL). Other bits correspond to previous crypto packets. LSB masks the 64th oldest crypto packet.

Table A.51 – Description of the register rx_loa_window_mask.

A.1.51 RX_LOA_CNT_THRESHOLD – RX LOSS OF AUTHENTICATION COUNTER THRESHOLD

7-bit value (0 to 64) that specifies the threshold above which the authentication failure counting asserts the loss of authentication (LOA) indication.

Bit	Name	Description
6:0	LOA_CNT_THRESHOLD	7-bit value (0 to 64) specifying the threshold for LOA assertion.

Table A.52 – Description of the register rx_loa_cnt_threshold.

A.1.52 RX_LOA_FAIL_CNT_CLEAR – RX LOSS OF AUTHENTICATION FAILURE COUNTER CLEAR

Loss of authentication failure counter clear control.

Bit	Name	Description
0	CLEAR	Counter clear control. This bit is cleared by hardware. '0' – Normal counter operation. '1' – Counter clear.

Table A.53 – Description of the register rx_loa_fail_cnt_clear.

A.1.53 RX_LOA_FAIL_CNT – RX LOSS OF AUTHENTICATION FAILURE COUNTER

32-bit non-wrap-around LOA failure event counter. It indicates the number of loss of authentication (LOA) failure events within an active crypto session, up to 2^{32}.

Bit	Name	Description
15:0	LOA_FAIL_CNT	32-bit non-wrap-around counter showing the number of loss of authentication (LOA) events. It is incremented at every '0' → '1' transition of LOA and is cleared at the beginning of crypto sessions (after key changes).

Table A.54 – Description of the register rx_loa_fail_cnt.

A.1.54 RX_TAG_FAIL_CNT_CLEAR – RX AUTHENTICATION TAG FAILURE COUNTER CLEAR

Authentication TAG failure counter clear control.

Bit	Name	Description
0	CLEAR	Counter clear control. This bit is cleared by hardware. '0' – Normal counter operation. '1' – Counter clear.

Table A.55 – Description of the register rx_tag_fail_cnt_clear.

A.1.55 RX_TAG_FAIL_CNT – RX AUTHENTICATION TAG FAILURE COUNTER

32-bit non-wrap-around TAG_FAIL event counter. It indicates the number of crypto packets with TAG mismatch failure (TAG_FAIL) within an active crypto session, up to 2^{32}.

Bit	Name	Description
15:0	TAG_FAIL_CNT	32-bit non-wrap-around counter showing the number of crypto packets with authentication failure. It is incremented at every TAG mismatch (TAG_FAIL) and is cleared at the beginning of crypto sessions (after key changes).

Table A.56 – Description of the register rx_tag_fail_cnt.

A.1.56 RX_CSKS_BIT_CNT_CLEAR – RX CSKS BIT COUNTER CLEAR

CSKS bit '0' and bit '1' counter clear control.

Bit	Name	Description
0	CLEAR	Counter clear control for both bit '0' and bit '1' counters. This bit is cleared by hardware. '0' – Normal counter operation. '1' – Counter clear.

Table A.57 – Description of the register rx_csks_bit_cnt_clear.

A.1.57 RX_CSKS_BIT0_CNT – RX CSKS BIT 0 COUNTER

32-bit wrap-around counter of total received bits '0' in the crypto session key selection FEC code word, up to 2^{32}. It is updated after every crypto block.

Bit	Name	Description
15:0	CSKS_BIT0_CNT	32-bit wrap-around counter of total received bits '0' in the crypto session key selection FEC code word. It is updated after every crypto block.

Table A.58 – Description of the register rx_csks_bit0_cnt.

A.1.58 RX_CSKS_BIT1_CNT – RX CSKS BIT 1 COUNTER

32-bit wrap-around counter of total received bits '1' in the crypto session key selection FEC code word, up to 2^{32}. It is updated after every crypto block.

Bit	Name	Description
15:0	CSKS_BIT1_CNT	32-bit wrap-around counter of total received bits '1' in the crypto session key selection FEC code word. It is updated after every crypto block.

Table A.59 – Description of the register rx_csks_bit1_cnt.

A.1.59 RX_CSKS_HIST_CNT_CLEAR – RX CSKS HISTOGRAM COUNTER CLEAR

CSKS bit '0' and bit '1' histogram counter clear control.

Bit	Name	Description
0	CLEAR	Counter clear control for both bit '0' and bit '1' counters. This bit is cleared by hardware. '0' – Normal counter operation. '1' – Counter clear.

Table A.60 – Description of the register rx_csks_hist_cnt_clear.

A.1.60 RX_CSKS_BIT0_HIST_CNT – RX CSKS BIT 0 HISTOGRAM COUNTER

96-bit non-wrap-around histogram counter of received bits '0' in the crypto session key selection FEC code word. It is updated after every crypto block.

Bit	Name	Description
15:0	CSKS_BIT0_HIST_CNT	96-bit non-wrap-around histogram counter of received bits '0' in the crypto session key selection FEC code word. It is subdivided into four 24-bit counters that are incremented after every crypto block, depending on the total bit count within the 504-bit CSKS FEC code word (in the last crypto block). [95:72] – 24-bit counter incremented if 378 < total bit '0' count < 505. [71:48] – 24-bit counter incremented if 252 < total bit '0' count < 377. [47:24] – 24-bit counter incremented if 126 < total bit '0' count < 251. [23:0] – 24-bit counter incremented if total bit '0' count < 125

Table A.61 – Description of the register rx_csks_bit0_hist_cnt.

A.1.61 RX_CSKS_BIT1_HIST_CNT – RX CSKS BIT 1 HISTOGRAM COUNTER

96-bit non-wrap-around histogram counter of received bits '1' in the crypto session key selection FEC code word. It is updated after every crypto block.

Bit	Name	Description
15:0	CSKS_BIT1_HIST_CNT	96-bit non-wrap-around histogram counter of received bits '1' in the crypto session key selection FEC code word. It is subdivided into four 24-bit counters that are incremented after every crypto block, depending on the total bit count within the 504-bit CSKS FEC code word (in the last crypto block). [95:72] – 24-bit counter incremented if 378 < total bit '1' count < 505. [71:48] – 24-bit counter incremented if 252 < total bit '1' count < 377. [47:24] – 24-bit counter incremented if 126 < total bit '1' count < 251. [23:0] – 24-bit counter incremented if total bit '1' count < 125

Table A.62 – Description of the register rx_csks_bit1_hist_cnt.

A.1.62 RESERVED – RESERVED REGISTER

Bit	Name	Description
15:0	RES	Reserved bits.

Table A.63 – Description of the register (RESERVED).

Appendix B

Digital Integrated Circuit Design Flow

Figure B.1 represents, in a simplified way, the main phases of a traditional digital integrated circuit design flow used by CPQD in the design and implementation of the 100G OTN Processor device.

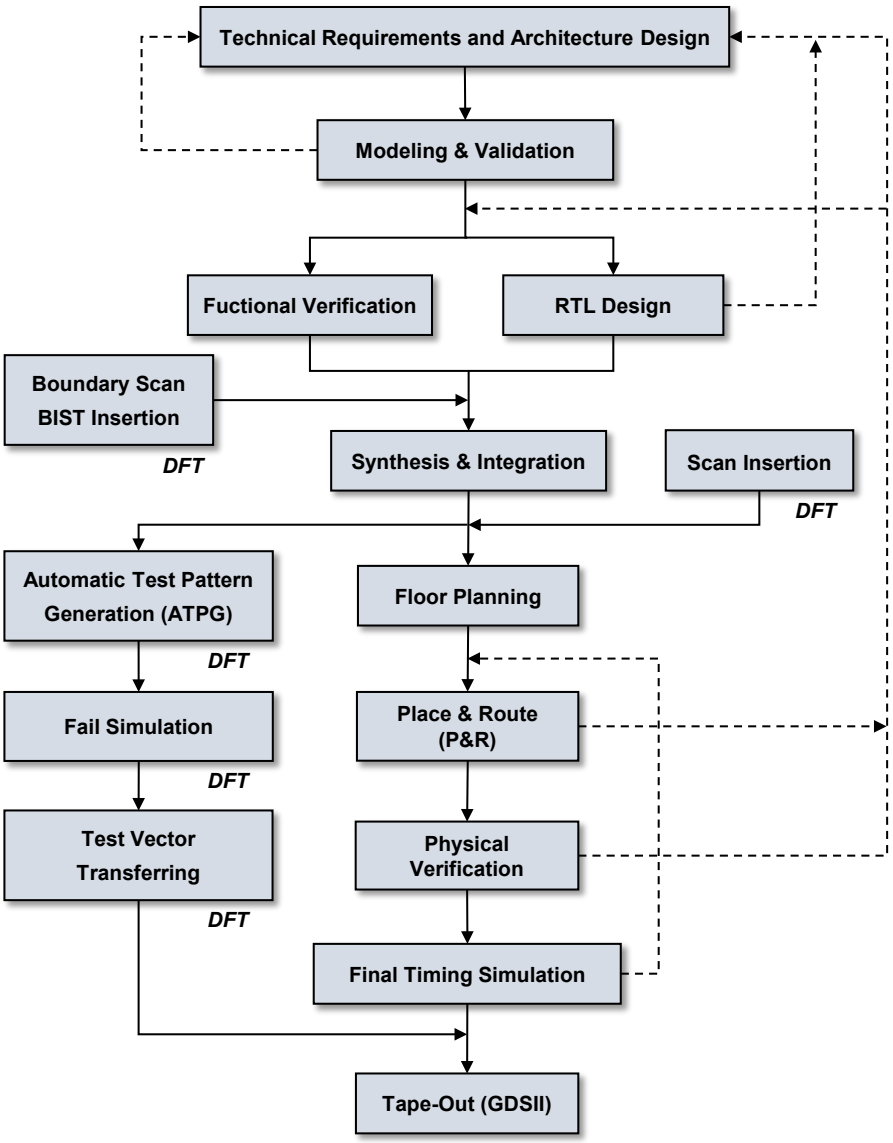


Figure B.1 – Digital integrated circuit design flow used by CPQD.

B.1 FRONT-END PHASE

B.1.1 TECHNICAL SPECIFICATIONS

The design flow starts with the technical specification, when all the characteristics of the integrated circuit are defined according to the proposed functional conception.

B.1.2 ARCHITECTURE DESIGN

It is one of the most important phases of the design flow and can be divided into functional and device architecture. The first deals with the hardware structures for implementing the desired functionalities, while the second, comprises the study and definition of physical parameters such as the number of pins, type of chip packaging, type of chip assembly on the chosen package, etc.

The functional architecture conception follows a top-down design methodology, with a macro-level definition of the desired functionalities, followed by a process of subdivision into smaller partitions up to the level where they can be implemented by a corresponding hardware structure. Thus, a high-level block diagram (top diagram) is obtained, with each block being expanded into new lower-level diagrams.

The device architecture design also comprises the definition of its manufacturing process technology (or technology node). Design for test (DFT) parameters are also studied in this phase, with specification results that will guide the coding stages for implementing test structures.

The documents generated in this phase provide the necessary inputs for the execution of the entire project.

B.1.3 MODELING AND VALIDATION

In this phase, behavioral models that represent the functionalities of each block of the chip are created using languages such as Verilog, System Verilog, C, C++, System C, and even dedicated tools such as MATLAB®.

They are designed to behave like the functional blocks of the integrated circuit, allowing for the validation of its functionality. Typically, they are also reused during the verification

phase, serving as a reference (golden models) for the comparison of results during tests carried out in a computational environment.

B.1.4 RTL DESIGN

From the documentation generated in the architecture design phase, which includes specifications for each functional block, the work of modeling the hardware structures necessary for implementing the desired functionalities can be started. The RTL design is then created using a specific language for hardware description.

Simple simulations for preliminary functional verification are also performed in this phase, where the behavior of the input and output signals of the logic block is evaluated.

Partial logic syntheses may be necessary, when RTL codes are converted into a physical hardware implementation in terms of logic cells (logic gates, flip-flops, etc.), allowing for the analysis of timing requirements such as propagation time, operating speed, etc.

B.1.5 FUNCTIONAL VERIFICATION

Starting from the same reference documentation used in the RTL design, test bench modules are coded for functional verification of each logic block developed for the device.

A typical test bench structure includes some mechanism for generating stimuli that are injected into the design or block under test (DUT) and a corresponding monitor or set of monitors that analyze the data produced by the DUT, comparing them to the expected results (according to the block specification, generated in the architecture design). The computer models generated in the modeling stage can also be used as a reference (golden models) for comparing the results.

The generated stimuli depend on the coverage desired for the verification process. Typically, random sequences are used to check for the occurrence of any unexpected result or behavior. The same structure applies to testing a set of interconnected blocks or even the complete chip design (top design test).

Special techniques are applied to achieve specific goals, such as code coverage, formal verification, and emulation. The first analyzes whether all sections of an RTL code are exercised in a test scenario. The second has an approach of formal proof of system

functionality represented by mathematical models. The last one corresponds to a prototyping technique of the entire chip (or some of its blocks) in hardware, usually on FPGA platforms, aiming at the execution of test cases in an accelerated way.

The verification phase consumes many computational resources, which requires powerful workstations or dedicated servers with high processing and storage capacity.

B.1.6 BOUNDARY SCAN

Usually, the development strategy considers the use of special design for test (DFT) techniques, through the insertion of scanning mechanisms (boundary scan) and self-test modules (built-in self-test – BIST), also described and modeled in RTL.

B.1.7 SYNTHESIS AND INTEGRATION

In the final step of the front-end stage, with all the functional blocks implemented in RTL and verified, all the IP cores are integrated into a top design to create the application conceived in the architecture design.

Then, the top synthesis process converts the RTL description of each functional block into a physical hardware implementation in terms of a netlist of logic cells (logic gates, flip-flops, etc.) available in a library provided by the foundry, generating files that feed the physical design phases in the back-end stage. These cells are interconnected according to the logic design of the block, and the synthesis process can be optimized to the area, power, and electrical length aspects of the chip (directly related to the signal propagation times).

A static timing analysis (STA) is performed as part of the optimization process, so that the resulting circuit meets the design frequency specification. Changes in the architecture of one or more functional blocks of the device may be required until all performance parameters are achieved.

Before the start of the physical implementation, scan chains are inserted to enable the manufacturing validation tests. This technique makes it possible to put the chip into a test mode, when the flip-flops are chained together in a shift register configuration, allowing for the insertion of a test pattern or vector that is shifted at each clock cycle until it is extracted and analyzed at the end of the chain.

B.2 BACK-END PHASE

B.2.1 PLACE AND ROUTE

This phase begins with a preliminary planning of the silicon layout (floor planning), defining the format, block partitions, power rings, connection points (pads), etc. Macro cells, such as memories and some specific IP cores, are also placed at this time.

Then, the placement process of the logic cells is carried out automatically within the areas defined in the preliminary layout floor planning.

Another important activity in this phase is the clock tree definition. Since the flip-flops of a synchronous design are scattered within a certain area of the silicon die, clock signals must be driven to the circuits so that all pulses reach their destination at the same time, regardless of different electrical lengths. This synchronization is obtained through a clock tree synthesis (CTS) process, performed automatically and characterized by the insertion of buffers to adjust the propagation delays along the different paths of the signal distribution network.

The interconnection of cells and macro blocks is then performed in the routing activity. It is done automatically in many design parts, but manually in others, as in the case of power connections and mixed-signal (analog and digital) blocks.

In the final stage, delay parameters are extracted (RC extraction) for use in an STA analysis that compares the physical design performance — in terms of operating speed, propagation times, etc. — with the design requirements.

Another important analysis is that of equivalence between the RTL design and the netlist of the logic gates in the physical design (gate netlist), since new components or structures may have been inserted at this stage (such as the clock tree buffers). This process is known as logic equivalence checking (LEC).

B.2.2 PHYSICAL VERIFICATION

A final verification phase is carried out directly on the silicon layout physical design, evaluating whether the behavior of the structures to be implemented (transistors, connections, etc.) corresponds to the expected functionality.

The layout versus schematic (LVS) analysis checks the equivalence between the silicon layout (transistors) and the functional design (represented by the schematic or netlist).

Additionally, manufacturing restrictions related to the process technology used, such as minimum spacing, thicknesses, etc., are also verified through the design rule checking (DRC) analysis.

B.2.3 Automatic Test Pattern Generation (ATPG)

According to the methodology and test strategy adopted for the integrated circuit design, a set of patterns or vectors is developed and generated at this stage to perform structural tests on the chip, through automatic test pattern generation (ATPG) tools.

B.2.4 Fail Simulation

In this phase, techniques such as fault grading and fault simulation are used to verify the coverage achieved by the proposed set of test vectors. In the first case, probabilistic techniques are used. It is faster but less accurate, so it is commonly used during the structural test development (ATPG) phase. The second case uses deterministic techniques, being much slower and, therefore, used to obtain additional coverage for fabrication.

B.2.5 Vector Test Transferring

The test vectors must be converted to a suitable format to be used by the automated test equipment (ATE), which will carry out the functional and structural validation tests of the chip under production.

B.2.6 Tape-Out

This is the final phase in which the integrated circuit manufacturing files are transferred to the foundry. The term tape-out dates back to the time when artwork designs for printed circuit boards were produced using black adhesive tape.

GDSII (Graphic Design System) is a standardized binary database file format used by the industry to exchange data corresponding to the integrated circuit artwork design.

Appendix C

ODU OH Inserter Sub-Block RTL Code Listing

The Verilog code listing below corresponds to the RTL design modeling of the ODU OH Inserter sub-block.

Input and output ports are defined right in the beginning, followed by the declaration of register variables (storage elements, not necessarily synthesizable) and wires (used for connecting different elements).

The *fb_pol* wire is then initialized with the feedback polynomial coefficients.

A concurrent sequential process named *input_register_seq_proc* (sensitive to *ipg_clk_sys* and *reset_sync_b*) registers (stores) the values of the TAG, AAD, and IV input signals.

Another concurrent sequential process named *cp_clk_cnt_seq_proc* handles a crypto packet wrap-around frame clock counter, which is incremented at every clock cycle when the *valid_in* signal is asserted and cleared right after an MFS pulse.

The overhead data insertion and scrambling functions are carried out by a combinational and a sequential process.

The first one, named *oh_variable_update_comb_proc*, updates the variable registers *res1* and *res2* with the TAG, AAD, or IV input data (previously stored in the registers *tag_buf*, *aad_buf*, and *iv_buf*). Data selection depends on the crypto packet frame clock counter, since the RES fields of the ODU frames within a crypto packet transport different encryption overhead parameters. When the *scrambler_ena* signal is enabled, 16-bit PRBS words are XOR combined with the updated data.

The second, named *oh_insertion_seq_proc*, effectively inserts the updated ODU overhead RES fields into the output frames.

The last coding block is a concurrent sequential process named *lfsr_seq_proc*, which creates a 16-bit LFSR that generates PRBS words at every clock cycle.

```
// +FHDR-------------------------------------------------------------------------
// Copyright (c) CPqD. All rights reserved
// ------------------------------------------------------------------------------
// FILE NAME: oh_inserter.v
// AUTHOR: Eduardo Mobilon
// ------------------------------------------------------------------------------
// PURPOSE: Overhead inserter module, responsible for the insertion of encryption
//          overhead data (TAG, AAD, and IV) into padded ODU overhead RES fields.
// ------------------------------------------------------------------------------
// KEYWORDS: Overhead, RES.
// ------------------------------------------------------------------------------
// REUSE ISSUES:
// Reset Strategy:
// Clock Domains:
// Critical Timing:
// Test Features:
// Asynchronous I/F:
// Scan Methodology:
// Synthesizable (y/n): y
// Other:
// ------------------------------------------------------------------------------
// RELEASE HISTORY:
// VERSION : DATE        : AUTHOR          : DESCRIPTION
//   1.0   : 07-MAY-2015 : Eduardo Mobilon : Initial version.
//   1.1   : 08-JUN-2015 : Eduardo Mobilon : Removal of some input signals.
//   1.2   : 06-AGO-2015 : Eduardo Mobilon : 'enable' port included.
//   1.3   : 16-OCT-2015 : Eduardo Mobilon : 'aad_write' and 'iv_write' combined
//                                           in just one 'aad_iv_write' port.
//   1.4   : 27-OCT-2015 : Eduardo Mobilon : LFSR implementation bug fixing.
//   1.5   : 05-NOV-2015 : Eduardo Mobilon : 'fail_in' and 'fail_out' ports included.
//   1.6   : 02-DEC-2015 : Eduardo Mobilon : Replacement of some conditional operators (?:)
//                                           because of code coverage problems.
// -FHDR-------------------------------------------------------------------------


module oh_inserter #(
   // Parameters
   parameter                  IPY_DW = 640      // Yellow line interface data width
) (
   // Inputs
   input wire                 reset_sync_b,     // Synchronous active-low hard reset
   input wire                 ipg_clk_sys,      // Green line interface clock at 180 MHz.
   input wire [IPY_DW-1:0]    data_in,          // Padded ODU data.
   input wire                 valid_in,         // Padded ODU data valid.
   input wire                 fs_in,            // Padded ODU frame start pulse.
   input wire                 mfs_in,           // Padded ODU multi-frame start pulse.
   input wire                 fail_in,          // Fail signal.
   input wire [127:0]         tag,              // TAG value.
   input wire [31:0]          aad,              // AAD value.
   input wire [95:0]          iv,               // IV value.
   input wire                 tag_write,        // TAG write pulse.
   input wire                 aad_iv_write,     // AAD and IV write pulse.
   input wire                 scrambler_ena,    // Scrambler enable control.
   input wire                 enable,           // Block enable control.

   // Outputs
   output reg [IPY_DW-1:0]    data_out,         // Padded ODU data.
   output reg                 valid_out,        // Padded ODU data valid.
   output reg                 fs_out,           // Padded ODU frame start pulse.
   output reg                 mfs_out,          // Padded ODU multi-frame start pulse.
   output reg                 fail_out          // Fail signal.

);


// +----------------------------------------------------------------------------
// | Variables
// +----------------------------------------------------------------------------
   reg  [15:0]    lfsr;                         // Linear feedback shift register.
   reg  [9:0]     cp_clk_cnt;                   // Crypto packet clock counter.

   reg  [127:0]   tag_buf;                      // TAG value buffer.
   reg  [31:0]    aad_buf;                      // AAD value buffer.
   reg  [95:0]    iv_buf;                       // IV value buffer.

   integer        i;                            // Index variable.

   wire [16:0]    fb_pol;                       // Feedback polynomial.

   wire [111:0]   oh_row2;                      // ODU Overhead row 2.
   wire [111:0]   oh_row4;                      // ODU Overhead row 4.
```

```
   wire [527:0]   pyld;                         // ODU payload.

   reg [15:0]     res1;                         // ODU OH reserved field 1 (2 bytes).
   reg [47:0]     res2;                         // ODU OH reserved field 2 (6 bytes).


// +----------------------------------------------------------------------------
// | Variable & signal initialization
// +----------------------------------------------------------------------------
   assign fb_pol = 17'b11101010110000011;       // x16 + x15 + x14 + x12 + x10 + x8 + x7 + x + 1


// +----------------------------------------------------------------------------
// | OH data input register
// +----------------------------------------------------------------------------
   always @(posedge ipg_clk_sys, negedge reset_sync_b)
   begin : input_register_seq_proc
      if (!reset_sync_b) begin
         tag_buf     <= 128'd0;
         aad_buf     <= 32'd0;
         iv_buf      <= 96'd0;
      end else begin
         if (enable) begin
            if (tag_write) begin
               tag_buf <= tag;
            end
            if (aad_iv_write) begin
               aad_buf <= aad;
               iv_buf <= iv;
            end
         end // if (enable)
      end
   end // input_register_seq_proc


// +----------------------------------------------------------------------------
// | Crypto packet frame clock counter
// +----------------------------------------------------------------------------
   always @(posedge ipg_clk_sys, negedge reset_sync_b)
   begin : cp_clk_cnt_seq_proc
      if (!reset_sync_b) begin
         cp_clk_cnt <= 10'd0;
      end else begin
         if (enable) begin
            if (valid_in) begin
               if (mfs_in) begin
                  cp_clk_cnt <= 10'd0;
               end else begin
                  if (cp_clk_cnt < 10'd767) begin
                     cp_clk_cnt <= cp_clk_cnt + 10'd1;
                  end else begin
                     cp_clk_cnt <= 10'd0;
                  end
               end
            end // if (valid_in)
         end // if (enable)
      end
   end // cp_clk_cnt_seq_proc


// +----------------------------------------------------------------------------
// | OH data insertion and scrambler
// +----------------------------------------------------------------------------
// Depending on the state of 'scrambler_ena' input port, OH data are scrambled by a simple XOR
// operation between data and repeated copies of the 16-bit word generated by a 16-bit LFSR.
// For synchronization purposes, scrambling occurs at the moment of OH field insertion.

   // Frame overhead fields
   assign oh_row2 = {res1, data_in[623:528]};   // 14-byte OH row 2.
   assign oh_row4 = {data_in[639:576], res2};   // 14-byte OH row 4.

   // Frame payload field
   assign pyld    = data_in[527:0];

   // OH field variable update
   always @*
   begin : oh_variable_update_comb_proc
      // Variable default values (preventing Latch synthesis)
      res1 = data_in[639:624];
      res2 = data_in[575:528];

      case (cp_clk_cnt)
      48:            // OH Line 2      --- Crypto packet frame 1 ---
         begin
            if (scrambler_ena) begin
               res1 = tag_buf[127:112] ^ lfsr;
            end else begin
               res1 = tag_buf[127:112];
```

```
            end
         end
      144:           // OH Line 4
         begin
            if (scrambler_ena) begin
               res2 = tag_buf[111:64] ^ {3{lfsr}};
            end else begin
               res2 = tag_buf[111:64];
            end
         end
      240:           // OH Line 2      --- Crypto packet frame 2 ---
         begin
            if (scrambler_ena) begin
               res1 = tag_buf[63:48] ^ lfsr;
            end else begin
               res1 = tag_buf[63:48];
            end
         end
      336:           // OH Line 4
         begin
            if (scrambler_ena) begin
               res2 = tag_buf[47:0] ^ {3{lfsr}};
            end else begin
               res2 = tag_buf[47:0];
            end
         end
      432:           // OH Line 2      --- Crypto packet frame 3 ---
         begin
            if (scrambler_ena) begin
               res1 = aad_buf[31:16] ^ lfsr;
            end else begin
               res1 = aad_buf[31:16];
            end
         end
      528:           // OH Line 4
         begin
            if (scrambler_ena) begin
               res2 = iv_buf[95:48] ^ {3{lfsr}};
            end else begin
               res2 = iv_buf[95:48];
            end
         end
      624:           // OH Line 2      --- Crypto packet frame 4 ---
         begin
            if (scrambler_ena) begin
               res1 = aad_buf[15:0] ^ lfsr;
            end else begin
               res1 = aad_buf[15:0];
            end
         end
      720:           // OH Line 4
         begin
            if (scrambler_ena) begin
               res2 = iv_buf[47:0] ^ {3{lfsr}};
            end else begin
               res2 = iv_buf[47:0];
            end
         end
      endcase
   end // oh_variable_update_comb_proc


   // OH field insertion
   always @(posedge ipg_clk_sys, negedge reset_sync_b)
   begin : oh_insertion_seq_proc
      if (!reset_sync_b) begin
         data_out  <= {IPY_DW{1'b0}};
         valid_out <= 1'b0;
         fs_out    <= 1'b0;
         mfs_out   <= 1'b0;
         fail_out  <= 1'b0;
      end else begin
         if (enable) begin
            if (valid_in) begin
               valid_out <= 1'b1;
               fs_out    <= fs_in;
               mfs_out   <= mfs_in;
               fail_out  <= fail_in;

               case (cp_clk_cnt)
               48:            // OH Line 2      --- Crypto packet frame 1 ---
                  begin
                     data_out <= {oh_row2, pyld};
                  end
               144:           // OH Line 4
                  begin
                     data_out <= {oh_row4, pyld};
                  end
```

```
                240:           // OH Line 2      --- Crypto packet frame 2 ---
                   begin
                      data_out <= {oh_row2, pyld};
                   end
                336:           // OH Line 4
                   begin
                      data_out <= {oh_row4, pyld};
                   end
                432:           // OH Line 2      --- Crypto packet frame 3 ---
                   begin
                      data_out <= {oh_row2, pyld};
                   end
                528:           // OH Line 4
                   begin
                      data_out <= {oh_row4, pyld};
                   end
                624:           // OH Line 2      --- Crypto packet frame 4 ---
                   begin
                      data_out <= {oh_row2, pyld};
                   end
                720:           // OH Line 4
                   begin
                      data_out <= {oh_row4, pyld};
                   end
                default:       // Remaining data (no OH insertion)
                   begin
                      data_out <= data_in;
                   end
             endcase
          end else begin
             valid_out <= 1'b0;
          end // if (valid_in)
       end else begin
          data_out  <= {IPY_DW{1'b0}};
          valid_out <= 1'b0;
          fs_out    <= 1'b0;
          mfs_out   <= 1'b0;
          fail_out  <= 1'b0;
       end // if (enable)
    end
  end // oh_insertion_seq_proc


// +-----------------------------------------------------------------------------
// | Linear feedback shift register
// +-----------------------------------------------------------------------------
// A linear feedback shift register (LFSR) of 16 stages with feedback polynomial
// x16 + x15 + x14 + x12 + x10 + x8 + x7 + x + 1 is used to generate
// 2^16-1 distinct 16-bit words, being reinitialized at every multi-frame
// (after an MFS pulse) with the sequence '0x5555'.
// Shifting occurs at every clock cycle, only when 'valid_in' input port is active.
// The LFSR must be implemented with internal feedback construction
// (modular or Galois type) with x16 as the MSB.
  always @(posedge ipg_clk_sys, negedge reset_sync_b)
  begin : lfsr_seq_proc
     if (!reset_sync_b) begin
        lfsr   <= 16'd0;
     end else begin
        if (enable) begin
           if (valid_in) begin
              // 16-stage Galois LFSR
              if (scrambler_ena) begin
                 if (mfs_in) begin
                    lfsr <= 16'h5555;
                 end else begin
                    for (i=15; i>0; i=i-1) begin   // x^n and x^0 coefficients are always included.
                       if (fb_pol[i]) begin
                          lfsr[i] <= lfsr[i-1] ^ lfsr[15];
                       end else begin
                          lfsr[i] <= lfsr[i-1];
                       end
                    end
                    lfsr[0] <= lfsr[15];
                 end
              end
           end // if (valid_in)
        end // if (enable)
     end
  end // lfsr_seq_proc

endmodule
```