ULPGC
Universidad de Las Palmas de Gran Canaria

Instituto Universitario de Microelectrónica Aplicada

IUMA

Máster Universitario en Electrónica y Telecomunicación Aplicadas (META)

Trabajo Fin de Máster

Diseño de un sistema hardware para compresión de datos a una tasa de 8 Gbps basado en el estándar CCSDS 121.0-B-3

Autor: **Samuel Torres Fau**

Tutor(es): **Dr. Roberto Sarmiento Rodríguez**

Dr. Antonio José Sánchez Clemente

Fecha: **agosto - 2023**

IUMA

t +34 928 451 150
+34 928 451 086
f +34 928 451 083

e: iuma@iuma.ulpgc.es
w: www.iuma.ulpgc.es

Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

Versión: 1 – 20 julio 2020

ULPGC
Universidad de Las Palmas de Gran Canaria

Instituto Universitario de Microelectrónica Aplicada

IUMA

Máster Universitario en Electrónica y Telecomunicación Aplicadas (META)

Trabajo Fin de Máster

Diseño de un sistema hardware para compresión de datos a una tasa de 8 Gbps basado en el estándar CCSDS 121.0-B-3

HOJA DE FIRMAS

Alumno/a: Samuel Torres Fau Fdo.:

Tutor/a: Dr. Roberto Sarmiento Rodríguez Fdo.:

Tutor/a: Dr. Antonio José Sánchez Clemente Fdo.:

Fecha: agosto - 2023

IUMA
t +34 928 451 150
+34 928 451 086
f +34 928 451 083
e: iuma@iuma.ulpgc.es
w: www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

Versión: 1 – 20 julio 2020

ULPGC
Universidad de Las Palmas de Gran Canaria

Instituto Universitario de Microelectrónica Aplicada

IUMA

Máster Universitario en Electrónica y Telecomunicación Aplicadas (META)

Trabajo Fin de Máster

Diseño de un sistema hardware para compresión de datos a una tasa de 8 Gbps basado en el estándar CCSDS 121.0-B-3

HOJA DE EVALUACIÓN

Calificación: .....

Presidente: Fdo.:

Secretario: Fdo.:

Vocal: Fdo.:

Fecha: agosto - 2023

IUMA
t +34 928 451 086
+34 928 451 150
f +34 928 451 083
e: iuma@iuma.ulpgc.es
w: www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

Versión: 1 – 20 julio 2020

Abstract

Earth Observation (EO) satellites generate an increasing amount of data, which has led on-board data management systems to become a critical part of space missions. This is due to the constant improvement in sensors, which are now able to capture data at a higher resolution and at even faster rates, as well as the growing number of sensors included on-board satellites. As a result of this unceasing growth in the amount of data to be handled, data processing and compression systems have become mandatory in order to achieve a better and more efficient on-board storage and processing of the satellite information [1], as well as for optimizing the transmissions of the captured information.

The CCSDS 121.0-B-3 compression standard defines a universal lossless compressor specifically developed for space-borne systems. A unit-delay predictor is included for preprocessing the input samples. The entropy encoder works with blocks of samples, over which the coding option that provides the shortest codeword is applied, and thus the highest compression rate is achieved.

In this project, a high performance architecture based on the Consultative Committee for Space Data Systems (CCSDS) 121.0-B-3 data compression standard has been developed, implemented, verified and synthesized. Starting from the basis of the Hyperspectral Lossless Compressor for space applications (SHyLoC) 3.0 compressor, a highly parallelized architecture, in which an operation-based control allows the effective coordination of the different independent processing pipelines, has been designed and described in VHSIC Hardware Description Language (VHDL).

The design has been successfully verified through two different verification campaigns. A set of block-level testbenches was developed to specifically verify specific modules. Once the system was fully integrated, the most extensive verification phase started. A general testbench that checks for the correctness of the compressed bitstream by comparing it with its corresponding reference bitstream, previously generated, was developed.

Finally, the design has been synthesized and optimized, as some problems related to critical paths raised in the first synthesis runs. After introducing some changes in the pipeline, reasonable results were finally achieved, with an estimated throughput of 7.776 Gbps.

Resumen

Los satélites para observación terrestre generan una cantidad de datos cada vez mayor, lo que ha llevado a que los sistemas de procesamiento de datos se hayan convertido en una parte fundamental en las misiones espaciales. Esto es resultado de la constante mejora de los sensores, que provoca que estos sean capaces de tomar datos con mayores resoluciones y a una mayor velocidad, así como en el aumento del número de sensores que son incluidos en los propios satélites. A vista de estos crecimientos, los sistemas de procesamiento y compresión de datos se han convertido en partes cruciales para optimizar tanto el almacenamiento a bordo de la información como la propia transmisión de estos mismos datos.

El estándar CCSDS 121.0-B-3 define un compresor universal sin pérdidas desarrollado específicamente para sistemas espaciales. Este define un predictor *Unit-Delay* básico para el preprocesado de las muestras de entrada. El codificador entrópico trabaja con bloques de muestras, sobre cada uno de los cuales selecciona la opción de codificación más corta que, a su vez, proporciona la mayor tasa de compresión.

En este proyecto se ha desarrollado, verificado y sintetizado un compresor de datos de alto rendimiento que implementa el estándar CCSDS 121.0-B-3. Partiendo de la base del compresor SHyLoC 3.0, se ha planteado y descrito en VHDL una arquitectura altamente paralelizada, en la que un control basado en operaciones permite la coordinación de las diferentes líneas de procesamiento independientes.

El diseño ha sido verificado a través de dos fases bien diferenciadas. Primero, se desarrolló un conjunto de bancos de pruebas para verificar a nivel de bloque partes concretas y críticas del diseño. Tras la integración completa del sistema, se comenzó una fase de verificación más extensa y exhaustiva. En ésta, se desarrolló un banco de pruebas general que comprueba que los bitstream resultantes de las compresiones se corresponden con lo esperado, gracias a compararlos con sus bitstreams de referencia correspondientes, los cuales son generados de forma previa.

Por último, el diseño ha sido sintetizado y optimizado, dado que en las primeras pruebas aparecieron algunos problemas relacionados con caminos críticas. Tras introducir algunas modificaciones en el pipeline, se consiguió obtener resultados adecuados a lo esperado, pues estos indican que sistema es capaz de soportar flujos de procesamiento de alrededor de 7.776 Gbps.

Acknowledgements

First and foremost, I would like to express my deep gratitude to my supervisors, Antonio José Sánchez Clemente and Roberto Sarmiento Rodríguez, for their invaluable guidance, patience, and expertise. Their continuous support and constructive feedback have been crucial for the successful development of this project.

I am also indebted to the professors of the University of Las Palmas de Gran Canaria, especially to those who lectured me in the Master's Degree in Applied Electronics and Telecommunications, who have played a crucial role in my academic career. Their insightful lectures, enriching discussions and commitment to excellence have greatly contributed to both my academic and personal development.

I am grateful to my friends and colleagues, as their company, discussions, advice and shared experiences have encouraged me to move forward. I truly feel fortunate to have them by my side.

Last but certainly not least, I want to express my heartfelt appreciation to my family. Despite the physical separation, their unwavering love, encouragement, and belief in my abilities, have been the driving force behind my academic pursuits. I am forever grateful for their continuous support during this challenging period of my life.

Contents

Abstract i

Resumen iii

Acknowledgements v

List of Figures xi

List of Tables xiii

Abbreviations xv

1 Introduction 1
- 1.1 Outline . . . 1
- 1.2 On-Board Data Processing . . . 1
  - 1.2.1 Space Mission Compression . . . 2
- 1.3 Motivation . . . 3
- 1.4 Background . . . 4
  - 1.4.1 CCSDS 121.0-B-3 . . . 4
  - 1.4.2 SHyLoC IP . . . 5
- 1.5 Design Requirements and Objectives . . . 5
  - 1.5.1 Methodology . . . 7

2 Background 9
- 2.1 Outline . . . 9
- 2.2 CCSDS 121.0-B-3 . . . 10
  - 2.2.1 Overview of the standard . . . 10
  - 2.2.2 Unit-Delay Predictor . . . 11
  - 2.2.3 Block-Adaptive Encoder . . . 12
    - 2.2.3.1 Fundamental Sequence Option . . . 14
    - 2.2.3.2 K Split-Sample Options . . . 15

2.2.3.3 Second Extension Option . . . 15
2.2.3.4 Zero-Block Option . . . 16
2.2.3.5 No Compression Option . . . 17
2.2.4 Transmission Formats . . . 18
2.3 SHyLoC IP Core . . . 20
2.3.1 SHyLoC CCSDS 121.0-B-3 Compressor IP . . . 21
2.3.2 SHyLoC CCSDS 123.0-B-1 Compressor IP . . . 23

3 Architecture Design 25
3.1 Outline . . . 25
3.2 Overview of the implemented architecture . . . 26
3.2.1 Scalablility . . . 29
3.2.2 Design considerations . . . 30
3.2.2.1 Considerations about the parameter $'J'$ . . . 30
3.2.2.2 Considerations about the parameter $'Reference\ Sample\ Interval'$ . . . 31
3.3 Unit-Delay Predictor . . . 31
3.4 Post-Predictor Block Dispatcher . . . 32
3.5 Block-Adaptive Encoder . . . 34
3.5.1 Block Unbundler . . . 38
3.5.2 CDS Coding Option Selection and Length Calculation . . . 40
3.5.3 CDS Codification . . . 42
3.5.4 CDS Intermediate Reconstruction . . . 44
3.5.5 Zero-Block CDS Codification . . . 46
3.5.6 Header Insertion . . . 47
3.5.7 CDS Retirement . . . 48
3.5.7.1 FSM4 . . . 48
3.5.7.2 Additional pipelining . . . 49
3.5.7.3 CDS Reorder Unit . . . 52
3.5.7.4 CDS Retirement Unit . . . 53
3.5.7.5 Considerations on the output buffer size and the slice size 53
3.6 Data interfaces . . . 54
3.6.1 AXI4 Stream . . . 54
3.6.2 Input Interface . . . 56
3.6.3 Output Interface . . . 57
3.6.4 Modularity in the design interfaces . . . 58
3.7 Conclusion . . . 59

4 Design Verification and Synthesis 61
4.1 Outline . . . 61
4.2 VHDL Description . . . 61
4.3 Configuration Parameters . . . 63
4.4 Verification of the design . . . 64
4.4.1 Block-Level Verification . . . 64

4.4.1.1 Predictor Testbench 65
4.4.1.2 Post-Predictor Dispatcher Testbench 67
4.4.1.3 Block-Level Verification Results 68
4.4.2 System-Wide Verification 69
4.4.2.1 Original SHyLoC Testbench 69
4.4.2.2 Adaptation of the testbench 70
4.4.2.3 Verification flow 71
4.4.2.4 Testcases 71
4.4.2.5 System-Wide Verification Results 74
4.5 Synthesis results 76
4.6 Result analysis 78

5 Conclusions **81**

List of Figures

1.1 General overview of the CCSDS 121 compression algorithm [4] 4

2.1 General scheme of the CCSDS 121.0-B-3 standard [4] 10
2.2 Fundamental Sequence (FS) Codeword Generation 14
2.3 FS Coded Data Set 14
2.4 FS Coded Data Set with reference sample 15
2.5 K Split-Sample Coded Data Set 15
2.6 K Split-Sample Coded Data Set with reference sample 15
2.7 Second Extension Coded Data Set 16
2.8 Second Extension Coded Data Set with reference sample 16
2.9 Zero-Block FS Codeword Generation 17
2.10 Zero-Block Coded Data Set 17
2.11 Zero-Block Coded Data Set with reference sample 17
2.12 No Compression Coded Data Set 17
2.13 Packet Format with l Coded Data Set (CDS)es [4] 18
2.14 Packet Format with l CDSes [4] 18
2.15 SHyLoC Possible Configurations [13] 21
2.16 SHyLoC CCSDS 121.0-B-3 IP General Architecture [13] 22
2.17 SHyLoC CCSDS 121.0-B-3 Block-Adaptive Encoder Datapath [13] 22

3.1 Initial design of the parallelized architecture 26
3.2 Parallel architecture based on the SHyLoC CCSDS121-IP 27
3.3 General diagram of the extended compressor with 8 parallel processing lanes 30
3.4 Parallelized architecture of the CCSDS 121.0 Unit-Delay Predictor 32
3.5 Top view of the Parallel121 predictor module 33
3.6 Initial multiplexor-based design of the Post-Predictor Dispatcher 34
3.7 Final design of the Post-Predictor Dispatcher 35
3.8 General overview of the Parallel121 Block-Adaptive Encoder architecture . 37
3.9 General data flow in the parallelized design 39
3.10 Architectural Overview of the Block Unbundler Module 40
3.11 Architectural Focus in CDS Length Calculation Phase 41
3.12 Architectural Focus in CDS Codification Phase 43
3.13 New Finite State Machine 3 44
3.14 Overview diagram of the CDS Reconstruction phase 45
3.15 Overview diagram of the Zero-Block CDS Reconstruction phase 47

3.16 General representation of FSM4 functionality 50
3.17 Architectural comparison of the original and pipelined CDS retirement datapath 51
3.18 Architectural overview of the CDS Reorder Unit Component 52
3.19 AXI4 Stream handshake possibilities [17] 55
3.20 AXI4-Stream Input Interface Diagram 56
3.21 AXI4-Stream Output Interface Diagram 57
3.22 Data flow between the different clock domains 59

4.1 Overview of the Parallel121 Unit-Delay Predictor Testbench 65
4.2 Block diagram of the Parallel121 Post-Predictor Dispatcher Testbench . . . 68
4.3 Flow diagram of the verification 72

List of Tables

2.1 CDS Coding Option Identifiers 13
2.2 File Format Header fields 19

3.1 Operation Struct Fields 48

4.1 List of VHDL Sources 62
4.2 Compile time parameters 63
4.3 Runtime parameters 64
4.4 Unit-Delay Predictor Testbench Parameters 66
4.5 G1 Testcases 73
4.6 G2 Testcases (I) 74
4.7 G2 Testcases (II) 75
4.8 G3 Testcases 76
4.9 Final clock and resource utilization results 77
4.10 SHyLoC 121.0 resource utilization results 78
4.11 Result comparison summary 79

Abbreviations

AHB Advanced High-performance Bus

AMBA Advanced Microcontroller Bus Architecture

ASIC Application Specific Integrated Circuit

AXI Advanced eXtensible Interface

CCSDS Consultative Committee for Space Data Systems

CDS Coded Data Set

DSP Digital Signal Processing

DSI Diseño de Sistemas Integrados

DUT Design Under Test

EO Earth Observation

ESA European Space Agency

FF Flip-Flop

FIFO First-In First-Out

FPGA Field Programmable Gate Array

FS Fundamental Sequence

FSM Finite-State Machine

IC Integrated Circuit

IP Intellectual Property

LSB Less Significant Bit

LUT Look-Up Table

MSB Most Significant Bit

ROS Remainder of Segment

RTL Register Transfer Level

SHyLoC Hyperspectral Lossless Compressor for space applications

ULPGC University of Las Palmas de Gran Canaria

VHDL VHSIC Hardware Description Language

Chapter 1

Introduction

1.1 Outline

This chapter introduces some key aspects of the project. Firstly, the motivation beyond the new parallelized architecture will be discussed in Section 1.2, focusing onto the space mission compression algorithms. The CCSDS 121.0-B-3 standard is introduced in the Section 1.4.1. Next, a brief overview of the developed architecture is offered in Section 3.2, which is followed by the general objectives of the project, listed at Section 1.5. Lastly, both the work plan and the document structure are explained in Section 1.5.1.

1.2 On-Board Data Processing

The evolution of EO satellites has revolutionized our ability to gather data about our planet from space. These satellites generate an increasing amount of data, which has led on-board data management systems to become a critical part of space missions. This is due to the constant improvement in modern sensors. The number of integrated sensors is incessantly increasing, and these are able to capture data at higher resolutions and at even faster rates, pushing the boundaries of the processing systems that handle this information. Hyperspectral imaging serves as a prime example, where latest sensors generate massive data flows, often measured in gigabits per second, due to their exceptional resolution and their great dynamic ranges and operating frequencies.

As a result of this exponential growth in the amount of data to be handled, data processing and compression systems have become indispensable in order to achieve a better and more efficient on-board storage and download of the satellite information [2]. This project is devoted to the development of on-board electronic systems for compression of data gathered on satellites.

1.2.1 Space Mission Compression

When it comes to compression algorithms, three main types can be enumerated: Lossless, lossy and near-lossless compression algorithms. Lossless compression focuses on reducing the size of data by effectively addressing its redundancy, while allowing to completely recover the original data. In other words, no information is lost in the compression and decompression processes. On the other hand, lossy compression applies some transformations such as quantization to achieve even greater compression ratios, at the cost of losing some information. Near-Lossless compression is similar to lossy algorithms in that these algorithms do not allow to fully retrieve the original information from the compressed data. However, these algorithms include additional mechanisms that allow users to quantitatively control and limit the amount of information that is lost during the compression process.

Space-specific compression algorithms are necessary to address the unique challenges of data transmission and storage in space missions [3]. These algorithms optimize data size while preserving quality and integrity. These algorithms tend to leverage inherent redundancies and patterns in space data, such as high-resolution and multi-band imagery, to ultimately achieve greater compression ratios. Furthermore, space-specific algorithms consider onboard resource constraints, including computational power and storage capacity, enabling effective data processing within these limitations. Several compression standards have emerged to address the unique challenges of space-based applications.

In this regard, the CCSDS, an international standardization organization that develops and promotes standards for space data systems, is widely recognized as a leading authority in the field of space communications and data management. The CCSDS standards encompass various aspects of space missions, including data compression, data exchange, spacecraft commanding, telemetry, and timekeeping. These standards provide a framework for interoperability and compatibility among different space agencies and organizations, facilitating efficient and reliable communication, data processing, and storage in space

missions. The CCSDS plays a vital role in ensuring the seamless integration of space systems and promoting the advancement of space technology on a global scale.

The CCSDS has developed a suite of compression algorithms specifically oriented towards space missions, such as the CCSDS Lossless Universal Data Compression (CCSDS 121.0-B-3) [4], the CCSDS Image Data Compression (CCSDS 122.0-B-2) [5] and the CCSDS Lossless and Near-Lossless Multispectral and Hyperspectral Image Data Compression (CCSDS 123.0-B-2) [6] standards.

1.3 Motivation

As stated before, considering the sheer magnitude of the data that most recent satellites are able to gather, data compression becomes necessary to speed up transmissions between these satellites and ground stations, especially when taking into account the limited bandwidth of communication links. In addition, available hardware on-board satellites is often limited when it comes to computational performance, available area and power consumption [7]. To tackle this challenge, Field Programmable Gate Arrays (FPGAs) have emerged as a game-changing technology, enabling on-board computation of complex data intensive algorithms that would otherwise struggle to achieve real-time processing when executed solely with embedded software on microprocessors, such as the the LEON-based ones [8].

Some specific missions need to process even more massive amounts of data, i.e. satellites that incorporate hyperspectral cameras, while the available hardware resources are scarce. Space missions FPGAs do often incorporate multiple processing systems into the same chip, which leads to hardware constrained scenarios.

Space missions often require to process data at high throughput, while available resources are scarce. In those cases where high compression ratios are not required, 1D Compression (i.e. CCSDS 121) may be used instead of 3D Compression (i.e. 123) for image processing, ultimately leading to a much lower hardware occupancy. 1D compression rates (~ 2) are significantly lower than 3D lossless compression rates (~ 4) [9], but at the same time the overall complexity of the whole compression process tends to be lower as well. This fact facilitates the development of high-performance architectures. For these reasons, it has been decided that this project will be based on the CCSDS 121.0-B-3 standard.

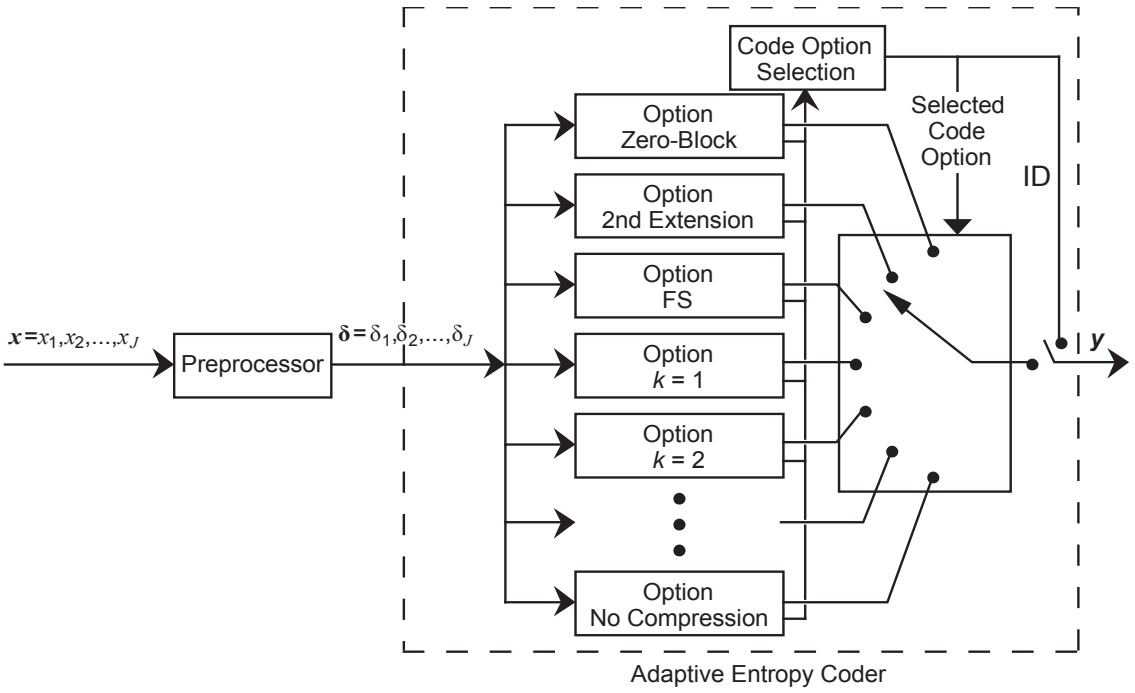

FIGURE 1.1: General overview of the CCSDS 121 compression algorithm [4]

1.4 Background

1.4.1 CCSDS 121.0-B-3

The CCSDS 121 standard specifies a universal data compressor based on the use of Rice codes. This means that any kind of data collected on the satellite from different sources, including scenes acquired by on-board hyperspectral imagers, can be compressed just with a single processing core. This project aims to implement the latest version of the standard, CCSDS 121.0-B-3 [4]. The compressor defined in CCSDS 121.0-B-3 consists of two well differentiated stages: a preprocessor and a block-adaptive encoder. An overview of the CCSDS 121 algorithm is shown in Fig. 2.16.

The preprocessor is in charge of transforming the raw input data (D-bit samples) into more advantageous data blocks, but without necessarily reducing their size. In a general manner, the main objective of the preprocessor is to reduce the overall entropy of consecutive samples, leading the system to achieve higher compression ratios. Concretely, the use of predictors as preprocessors implies that it is not the raw data that must be encoded, but the prediction errors (i.e., the prediction residuals). An optimized predictor will result in lower prediction errors, eventually achieving even higher compression ratios.

A noteworthy consideration is that, in order to narrow down the prediction errors and facilitate subsequent decompression, the predictor has to be periodically fed with reference samples. The CCSDS 121 standard defines a simple and reversible unit-delay predictor, which uses just the previous sample as an estimator of the current one.

Then, the block-adaptive encoder is responsible of computing the variable length codes, which are unique for each potential data block of J samples (user-defined parameter) coming from the preprocessing stage. This coding block features a number of different options, each of which is optimal for certain particular ranges of blocks. The encoder is able to detect for each block which of the encoding techniques, which are simultaneously computed, the optimal one, allowing to reach the highest possible compression rate, even in scenarios that require to compress highly heterogeneous data.

1.4.2 SHyLoC IP

SHyLoC is a set of lossless compression Intellectual Property (IP) Cores that were developed at the Diseño de Sistemas Integrados (DSI) Research Group. It includes both a CCSDS 121.0-B-3 Compressor IP and a CCSDS 123.0-B-1 Compressor IP. These IPs can be used in standalone or in tandem mode. These support wide ranges of configurations of its corresponding compression algorithm and have been included into the European Space Agency (ESA) portfolio [10]. The base architecture and base VHDL descriptions, which have been used for the development of this project, are the ones corresponding to the third version of the SHyLoC CCSDS 121 Compressor IP.

1.5 Design Requirements and Objectives

The main overall objective of this project consists in the development of a high-performance, parallel universal data compressor for space applications, which is CCSDS 121.0-B-3 compliant. This compressor is oriented to applications where large data streams are processed but in which hardware resources are limited.

The set of more specific objectives that were established during the initial phases of the project is the following:

1. Space Compression State-of-the-art Study.

- Modern compression standards and recent applications in the field of space compression data systems are studied to obtain the necessary background which is required to effectively develop this project. Specifically, SHyLoC IP Core has been thoroughly studied.

2. Design of a highly parallelized, scalable architecture.

   - Starting from the original SHyLoC CCSDS 121.0 Compressor, a new high-performance, parallel architecture is explored, analyzed and designed. This design shall be easily scalable, so future throughput-intensive application can utilize extensions of this architecture.

3. Description of the system in VHDL.

   - Once designed, the different components of the architecture are described in a Register Transfer Level (RTL) Language, specifically these are described in VHDL. This description allows the design to be synthesized and implemented in a wide range of applications and technologies.

4. Verification of the design.

   - Verification is a key phase in this project as it allows to demonstrate not only that the internal components behave as expected but also that the entire compressor system is standard compliant. The latter is checked by comparing the compressed bitstream with a golden bitstream, which is previously obtained by running a reference software.

5. Synthesis of the design.

   - The synthesis results allows the detection of additional problems in the design (i.e. unexpected critical paths in some stages of the pipeline) and help to estimate both the hardware resource utilization and the expected system clock frequency, among other relevant metrics.

Some specific requisites of the compressor design were also imposed:

1. The system has to compress datasets of varying size by following the CCSDS 121.0-B-3 standard specification.

2. The system must be able to process samples with a dynamic range of up to 16 bits.

3. The system receives groups of 4 samples, which are received through an input AXI4-Stream data interface.
4. The system shall be able to process input throughputs of up to 8 Gbps.
5. The system outputs a varying number of bytes per clock cycle through an AXI4-Stream interface.
6. The system offers a configuration interface which allows to configure both the reference sample insertion interval and the size of the input dataset, a subset of the CCSDS 121.0-B-3 configuration parameters, during runtime.
7. The system must be technology agnostic.

1.5.1 Methodology

These general objectives were accomplished in various phases of development: Architecture design, architecture description, architecture verification and architecture implementation.

The first of the phases, the design of the architecture helped to determine that applying a parallelization paradigm to achieve higher processing throughputs was the correct choice. During this phase, the base design was described: Processing lanes will take care of processing whole blocks of samples while the Zero-Blocks were to be processed independently. The operation-based control, that will be exhaustively explained in Chapter 3, was also designed at this point.

The description phase consisted in writing the VHDL source code files that described the different components that the final architecture includes. Some original SHyLoC CCSDS 121.0 Compressor IP modules have been used, although all of them had to be deeply modified to address the additional issues and requirements that the new parallelized scheme introduces. In addition to these, a wide set of new modules have been developed to implement the functionalities that the Parallel121 compressor adds, mainly related to the retirement of compressed data which comes from different parts of the design. This has been the longest phase, as it started during the second week of March and finished by the last week of May. In order to present the final architecture, the different components and subsystems that have been described will be explained in Chapter 3.

The verification phase was performed in two separate periods. Some components were verified independently with custom, entity-level testbenches, during the last weeks of

March. The second, most critical verification campaign took place in June. As it will be further explained in Chapter 4, this system-wide verification allowed us to demonstrate that the compressor worked as expected under a wide range of scenarios. This phase was the most complex one, as some bugs were well hidden into the sources and some modules were behaving in unexpected ways.

Once the Parallel121 architecture was exhaustively verified, the design was synthesised. The selected target board is the Xilinx KCU105 development board, which includes a Kintex UltraScale XCKU040-2FFVA1156E, an integrated circuit that is technologically equivalent to the space-grade XQRKU060 FPGA [11]. The initial results were disappointing, and led us to introduce some changes in the architecture to reach the objective performance. These changes and the final results will also be deeply explained in Chapter 4.

Some final thoughts and the conclusions are finally offered in Chapter 5.

Chapter 2

Background

2.1 Outline

This chapter introduces the CCSDS 121.0-B-3 Standard, which defines a universal lossless compression algorithm. After giving a brief overview of the standard in Section 2.2.1, the preprocessor that is included in the standard, a simple Unit-Delay predictor, will be more extensively introduced in Section 2.2.2. The different parts and considerations of the Block-Adaptive entropy encoder will be detailed in Section 2.2.3, paying special attention to the various coding options that are included. Lastly, the two bitstream packing options that the standard describes will be explained in Section 2.2.4.

SHyLoC IP Cores are a set of hardware designs that have been developed by the DSI Research Group of the University of Las Palmas de Gran Canaria (ULPGC). These two IP cores, the SHyLoC CCSDS 121.0-B-3 compliant Compressor IP and the SHyLoC CCSDS 123.0-B-1 compliant Compressor IP, will be shortly introduced in Section 2.3. A minimal recall of the CCSDS 121.0-B-3 standard will be offered in Subsection 2.3.1, followed by a more extensive explanation of the CCSDS 123.0-B-1 standard in Section 2.3.2.

2.2 CCSDS 121.0-B-3

2.2.1 Overview of the standard

The CCSDS 121.0-B-3 standard specifies a universal data compressor based on the use of Rice codes. This means that any kind of data collected on the satellite from different sources, including scenes acquired by on-board hyperspectral cameras, can be compressed just with a single processing core. This project aims to implement the latest version of the standard, CCSDS 121.0-B-3 [4].

The compressor defined in CCSDS 121.0-B-3 consists of two well differentiated stages (see Figure 2.1):

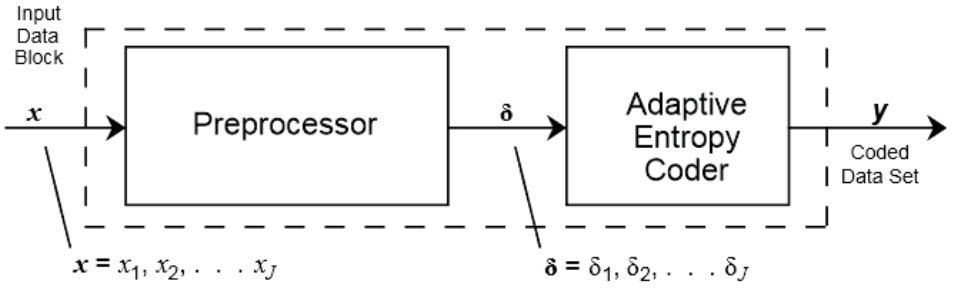

FIGURE 2.1: General scheme of the CCSDS 121.0-B-3 standard [4]

- Preprocess Stage. Unit-Delay Predictor.
- Encoding Stage. Block-Adaptive Encoder.

The preprocessor is in charge of transforming the raw input data (D-bit samples) into more advantageous data blocks, but without necessarily reducing their size. In a general manner, the main objective of the preprocessor is to reduce the overall entropy of consecutive samples, leading the system to achieve higher compression ratios. Concretely, the use of predictors as preprocessors implies that it is not the raw data what must be encoded, but the prediction errors (i.e., the prediction residuals). An optimized predictor will result in low prediction errors, eventually achieving even higher compression ratios. A noteworthy consideration is that, in order to harden the transmission of the coded bitstream, some types of predictor include reference samples into the final bitstream. If some error occurs during the transmission, the amount of information lost will be bounded by the next reference sample.

Some preprocessors are specifically designed to perform better with different types of data such as images, audio, video, and other forms of data. These preprocessors are often used as part of the overall data compression process to improve the efficiency and effectiveness of the compression algorithms, by applying specific transformations. The CCSDS 123.0 Predictor serves as a prime example, as it is specially design to predict samples from multispectral and hyperspectral imagery.

While it is true that CCSDS 121.0-B-3 defines a basic unit-delay predictor, the standard supports the integration of any other kind of preprocessor, or to not include any preprocessor at all.

2.2.2 Unit-Delay Predictor

The CCSDS 121 standard defines a simple, universal and reversible unit-delay predictor. A unit-delay predictor is a specific type of predictor used in data compression algorithms. It is designed to estimate the value of each data sample based on its previous value, assuming a constant or linear relationship between consecutive samples. The unit-delay predictor assumes a direct one-to-one relationship between consecutive samples, where the value of the current sample is estimated to be the same as the previous sample value. Therefore, it does not involve any complex modeling or parameter estimation.

A unit-delay predictor function can be described as it follows:

$$p(x_n) = x_{n-1}$$

where

$$n >= 0$$

The unit-delay predictor leverages the concept of temporal correlation or autocorrelation within the data. Many real-world signals and data sequences exhibit a certain level of continuity and predictability over time, where the current value can be approximated based on the immediate past values. By exploiting this correlation, the unit-delay predictor aims to compress the data by efficiently encoding the difference between the predicted value and the actual value. Some key points of these predictors are the following:

1. Computational Efficiency. A unit-delay predictor requires nearly none computational resources as it involves a simple assignment of the previous predicted value to the current sample. This simplicity translates into faster prediction times, making it ideal for real-time applications or time-constrained scenarios, and low hardware resources usage.

2. Memory Efficiency. Since the unit-delay predictor does not rely on storing and updating complex models or parameters, it minimizes the memory requirements. Just a single sample needs to be buffered in order to predict the following one.

3. Low Latency. The unit-delay predictor provides instantaneous prediction since it does not involve complex computations or model training. This low latency is advantageous in applications where quick responses or immediate decision-making is required, such as real-time control systems.

While it is true that unit-delay predictor's low complexity is beneficial in certain scenarios, it also has some limitations in others. These predictors are considered to be universally generalist, meaning it will not capture more complex patterns or heavier dependencies inherent to the data. In situations where the data exhibits non-linear relationships, significant variations, or other intricate structures, the unit-delay predictor performance may be limited, and more advanced prediction techniques, such as autoregressive models, adaptive predictors, or machine learning algorithms, tend to exhibit greater performances, but always at the expense of increased complexity and resource requirements.

A special situation takes place when a reference sample is received. These samples must be directly included in the output encoded bitstream, so these are forwarded directly to the encoder, meaning that no prediction is performed in this case.

2.2.3 Block-Adaptive Encoder

A block-adaptive encoder is a type of data compression encoder that operates on blocks or segments of the input data, where the size of the blocks can vary based on the characteristics of the data. It is designed to adapt to local variations in the data and optimize compression performance accordingly.

Before explaining the basics of the CCSDS Block-Adaptive Encoder, the basic set of parameters that influence it must be defined:

- **D**. Sample resolution, the size of each individual sample to be encoded.
- **J**. Number of samples per block. The allowed values are **8**, **16**, **32** and **64**.

The CCSDS 121.0-B-3 Block-Adaptive Encoder encodes the data prediction residuals (or the raw data in case that no predictor is being used) through Rice Coding, a hardware-efficient subset of the Golomb Codes that uses only powers of 2. Several different coding options, which can be observed in Figure 1.1, are applied in parallel to the same block of data, and the optimal codeword, which is the one that offers the shorter size, is selected as the winner and ultimately included in the encoded output. In order to identify which coding option has been applied to each of the J-Sample Blocks, a set of identifiers, one per option, is defined by the standard as shown in Table 2.1. The output data structure made of this identifier plus the J-samples encoded block is known as CDS.

Code Option	Resolution					
	Basic:	-	-	$n \leq 8$	$8 < n < 16$	$16 < n < 32$
	Restricted:	$n = 1, 2$	$n = 3, 4$	$4 < n < 8$	$8 < n < 16$	$16 < n < 32$
Zero-Block		00	000	0000	00000	000000
Second-Extension		01	001	0001	00001	000001
FS		N/A	01	001	0001	00001
k=1		N/A	10	010	0010	00010
k=2		N/A	N/A	011	0011	00011
k=3		N/A	N/A	100	0100	00100
k=4		N/A	N/A	101	0101	00101
k=5		N/A	N/A	110	0110	00110
k=6		N/A	N/A	N/A	0111	00111
k=7		N/A	N/A	N/A	1000	01000
k=8		N/A	N/A	N/A	1001	01001
k=9		N/A	N/A	N/A	1010	01010
k=10		N/A	N/A	N/A	1011	01011
k=11		N/A	N/A	N/A	1100	01100
k=12		N/A	N/A	N/A	1101	01101
k=13		N/A	N/A	N/A	1110	01110
k=14		N/A	N/A	N/A	N/A	01111
k=15		N/A	N/A	N/A	N/A	10000
...		...	...	...	...	...
k=29		N/A	N/A	N/A	N/A	11110
No-compression		1	11	111	1111	11111

Table 2.1: CDS Coding Option Identifiers

The property of being able to apply different codification techniques to each J-Sample Block of data allows the encoder to properly adapt to the characteristics of each part of the input data. More aggressive codification options, such as Second Extension or Zero-Block options, will be applied to low-entropy data, while for noisy or high-entropy data more conservative options will be applied, or even no codification at all, as explained in Subsection 2.2.3.5. Both the Zero-Block and the second extension options are designed to be specially efficient when coding low-entropy blocks (second extension) or zero-entropy blocks (Zero-Block) of predicted samples. Each of the included codification options are explained in the following subsections.

2.2.3.1 Fundamental Sequence Option

The FS option is considered to be the basic coding option of the standard as most of the other options are variants of it. FS consist in coding each predicted sample δ_i, where $\delta_i = m$, as m zeroes followed by a single 1 (See Figure 2.2). The J FS-coded samples are appended to the FS CDS Identifier, as shown in Figure 2.3, and finally included into the output bitstream. When a reference sample is included in the block, this is appended raw to the CDS identifier, and the other $J - 1$ encoded samples are concatenated at the end as shown in Figure 2.4.

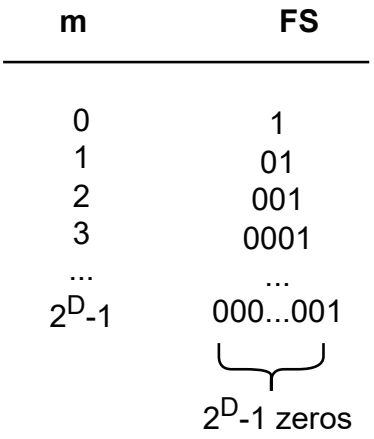

Figure 2.2: FS Codeword Generation

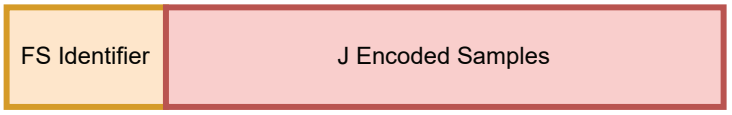

Figure 2.3: FS Coded Data Set

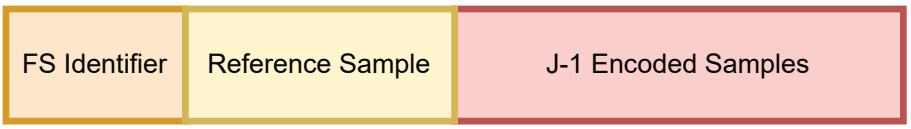

FIGURE 2.4: FS Coded Data Set with reference sample

2.2.3.2 K Split-Sample Options

Each of the K Split-Sample coding option lies in FS coding the $D - k$ Most Significant Bits (MSBs) of each sample, while leaving the k Less Significant Bits (LSBs) completely uncoded. The resulting length-varying CDS will be formed by the K-Split specific Identifier followed by the FS-encoded MSBs of each of the J samples, and all the uncoded LSBs at the end (See Figure 2.5). If a reference sample is included in the block, this is inserted following the identifier, followed by the Fundamental Sequences and the uncoded bits of the other $J - 1$ samples as shown in Figure 2.6

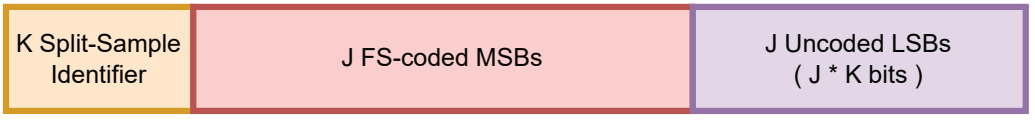

FIGURE 2.5: K Split-Sample Coded Data Set

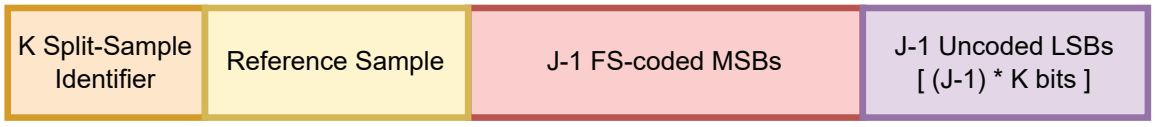

FIGURE 2.6: K Split-Sample Coded Data Set with reference sample

2.2.3.3 Second Extension Option

The second extension option is well suited for low-entropy blocks as the final size of the CDS will be smaller thanks to combining each pair of predicted samples.

The second extension options FS-codes each pair of consecutive predicted samples. Each of these pairs $(\delta_{2i-1}, \delta_{2i})$ is transformed by applying the following function:

$$\gamma_i = (\delta_{2i-1} + \delta_{2i}) * (\delta_{2i-1} + \delta_{2i} + 1)/2 + \delta_{2i}$$

The resulting J/2 γ symbols are FS-Coded and appended to the Second Extension Option Identifier as shown in Figure 2.7. When a reference sample is present in the block, a

special treatment must be applied to the data. The identifier and the reference sample are included at the beginning of the CDS as usual, followed by the $J/2$ γ symbols (See Figure 2.8). The difference is that in order to calculate the first γ symbol, δ_0 is taken as zero. This means that the first γ symbol will be obtained as it follows:

$$\gamma_0 = (\delta_0 + \delta_1) * (\delta_0 + \delta_1 + 1)/2 + \delta_1 = \delta_1 * (\delta_1 + 1)/2 + \delta_1$$

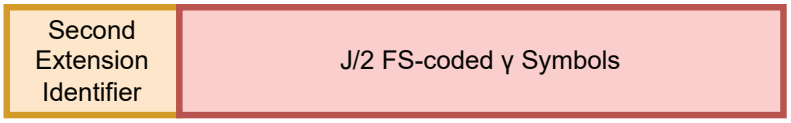

FIGURE 2.7: Second Extension Coded Data Set

FIGURE 2.8: Second Extension Coded Data Set with reference sample

2.2.3.4 Zero-Block Option

The Zero-Block option encodes up to 64 consecutive Zero-Blocks (blocks whose samples are 'empty', all zeros) into a single CDS. This option is always used when a single or multiple Zero-Blocks are detected, thanks to its inherent good compression capabilities.

The input predicted blocks are divided into segments of 64 blocks, with the exception of the last segment that may be smaller. Any sequence of consecutive Zero-Blocks that lies into the same segment will be encoded by FS-Coding the actual number of blocks that it is made of, as shown in Figure 2.9. However, if the end of segment is reached, and the number of consecutive block is greater than 4, a reserved codeword known as the Remainder of Segment (ROS) will be inserted instead (See Figure 2.9). This helps to achieve even greater compression ratios as completely homogeneous segments may be encoded in a single CDS.

The basic Zero-Block and the Zero-Block plus reference CDSes can be seen in Figures 2.10 and 2.11, respectively.

# of Consecutive Zero-Blocks	FS Codeword
1	1
2	01
3	001
4	0001
ROS	00001
5	000001
...	...
63	000...001 (63 zeros)

FIGURE 2.9: Zero-Block FS Codeword Generation

FIGURE 2.10: Zero-Block Coded Data Set

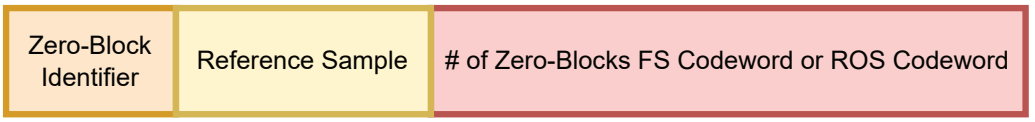

FIGURE 2.11: Zero-Block Coded Data Set with reference sample

2.2.3.5 No Compression Option

The no compression option consist in including the raw J-Samples in the CDS, headed by the no compression identifier as shown in Figure 2.12. As it might be expected, there is really no special consideration when a reference sample is to be included.

FIGURE 2.12: No Compression Coded Data Set

The no compression option is quite the opposite of the Zero-Block option, as it is well suited for highly entropic data blocks. In these cases, all the other options may produce CDSes with sizes larger than the actual uncompressed size, so including the option to just send the predicted samples uncompressed helps making the CCSDS 121.0-B-3 suitable for different scenarios.

2.2.4 Transmission Formats

The CCSDS 121.0-B-3 standard proposes two transmission formats for the encoded bitstream:

- Lossless Packet Format
- File Format

While both of these formats are designed to structure and organize the data to be transmitted or stored, they present some substantial differences.

The Lossless Packet Format organizes the output bitstream (the CDSes) into a single or many packets, each of these preceded by a header as shown in Figure 2.13. This header includes some control information relative to its own packet, such as synchronization markers, packet identifiers, packet lengths, and error detection codes. However, it generally doesn't contain extensive metadata about the entire file or additional information needed for decompression, so it is common to send this kind of information through side channels when using this format.

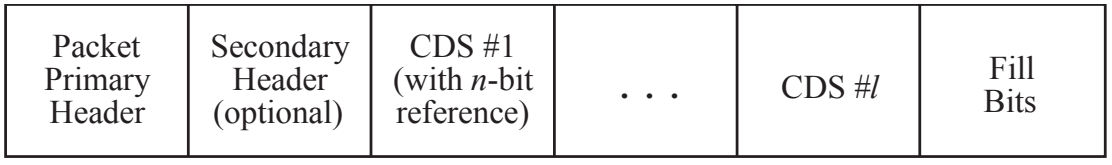

FIGURE 2.13: Packet Format with l CDSes [4]

The File Format is designed for the storage and transmission of large collections of compressed data. To be precise, a single file may contain up to 2^{48} compressed samples. It provides an alternative structure for organizing the contents of multiple packets into a single coherent file.

A file that follows the File Format specification is made of a single File Header and a single File Body (See Figure 2.14).

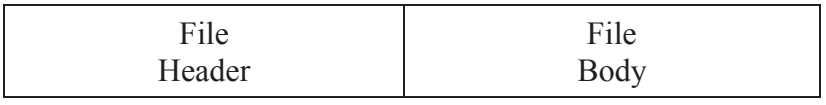

FIGURE 2.14: Packet Format with l CDSes [4]

The File Header structure, whose size is larger than the header of the other proposed format, includes all the neccessary information for the complete and correct decompression of the encoded samples embedded into the File Body. The complete list of the fields included in the header, as well as its corresponding widths and descriptions can be found in Table 2.2.

Field	**Width (bits)**	**Description**
Reserved	1	This field shall have value '0'.
Output Word Size (B	3	The value B-1 encoded as a 3-bit unsigned binary integer
Preprocessor Status	1	'0': Preprocessor absent '1': Preprocessor present
Predictor Type	3	'000': bypass predictor or preprocessor absent '001': unit delay predictor '111': application-specific predictor All other codes are reserved by CCSDS for future preprocessing options.
Mapper Type	2	'00': Prediction Error mapper or preprocessor absent '01': reserved '10': reserved '11': application-specific mapper
Data Sense	1	'0': two's complement '1': positive (mandatory if preprocessor is bypassed or preprocessor absent)
Reserved	8	This field shall have the value '00000000'.
Input Data Resolution	5	This field shall contain the value n-1 encoded as a 5-bit unsigned binary integer.
Reserved	1	This field shall have the value '0'.
Block Size	2	'00': J=8 '01': J=16 '10': J=32 '11': J=64
Restricted Code Option	1	'0': Basic set of code options are used; '1': Restricted set of code options are used.
Reference Sample Interval	12	This field shall contain a binary number equal to r–1, encoded as a 12-bit unsigned binary integer.
Reserved	8	This field shall have the value '00000000'.
Number of Samples (N)	48	This field shall be set to the total number of compressed input samples that are contained in the file, encoded as 48-bit unsigned integer, the binary representation of N-1.

Table 2.2: File Format Header fields

Although the larger size of this header initially suggests that the overhead is higher with this format than with the Packet Format, Packet Format includes all the necessary information about the coded bitstream, allowing to fully decompress it without the need of additional communications or assumptions. Another improvement is that only a single header is necessary per compression, while the Packet Format includes a header per packet, ultimately leading to a higher overhead, especially when compressing large amounts of data.

The File Body is built by concatenating the encoded CDSes that are being transmitted or stored in the file.

2.3 SHyLoC IP Core

SHyLoC [12] - [13] includes a sophisticated solution for implementing lossless data compression algorithms for space missions. This section provides an overview of the SHyLoC system, which comprises VHDL descriptions of two synthesizable IP cores. The latest version of these IP cores, namely SHyLoC 3.0 [14], adhere to the standards defined by the CCSDS 123.0-B-1 and CCSDS 121.0-B-3, offering efficient lossless data compression capabilities for both 1D and 3D data.

These IP cores can operate autonomously or in tandem. In the latter scenario, the CCSDS-123 Predictor IP functions as the preliminary processor, while the CCSDS-121 handles the entropy coding phase. These modes can be observed in Figure 2.15.

Independently from the selected operation mode, both IP Cores are mostly standard-compliant[1]. Both compressor cores can be configured at two different levels. One set of compile-time parameters is defined to statically define the capabilities of the system, while another set of run-time parameters allows different configurations to be applied individually to each compression run.

These IP cores have been developed in a technology-agnostic manner and can be mapped onto various Field Programmable Gate Array (FPGA) targets, including space-grade FPGAs. Both the SHyLoC CCSDS 121.0-B-3 Compressor Core and the SHyLoC CCSDS

[1]SHyLoC CCSDS 123.0-B-1 Compression IP supports all possible configuration apart from the custom weight initialization, the custom Sample-Adaptive Encoder Accumulator Table initialization and the Subframe Interleaving Depth parameter (Only 1 or Nz are supported). SHyLoC CCSDS 121.0-B-1 Compression IP supports all configurations

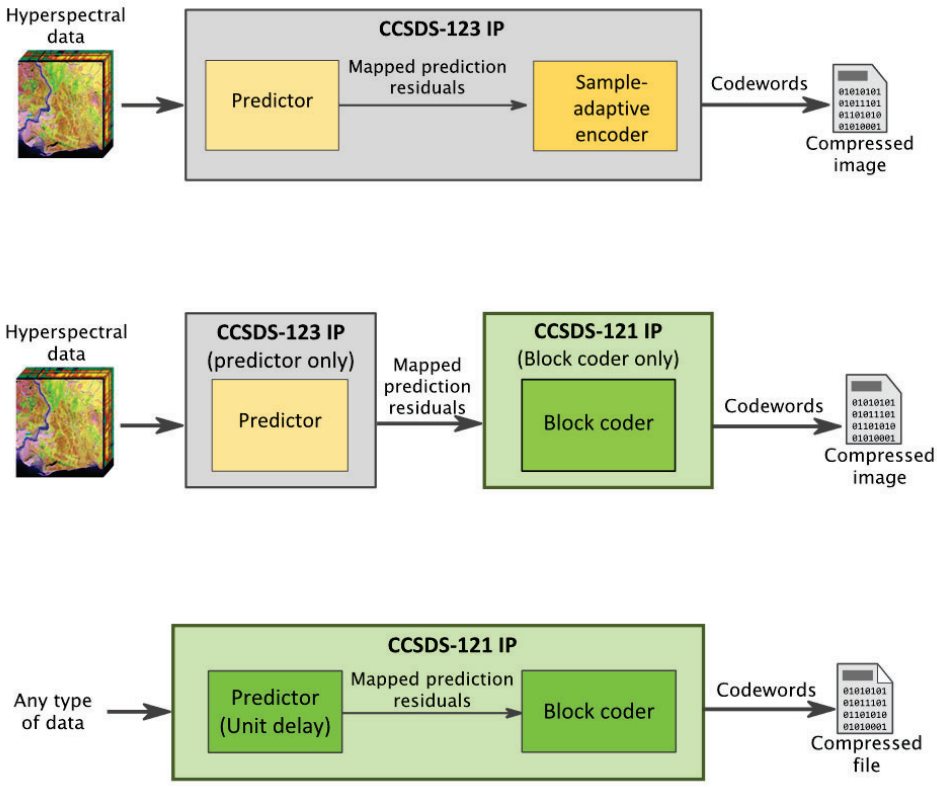

FIGURE 2.15: SHyLoC Possible Configurations [13]

123.0-B-1 Compressor Core process in a purely sequential manner the data throughputs that they receive through their interfaces.

2.3.1 SHyLoC CCSDS 121.0-B-3 Compressor IP

The CCSDS 121.0-B-3 is the third issue of the CCSDS 121.0 Lossless Data Compression Standard, which has been deeply in this chapter. It is composed of a Unit-Delay Predictor and a Block-Adaptive Entropy Coder as shown in Figure 2.16. Thanks to its inner flexibility and low complexity, the standard may be used wherever no extreme compression rates are need but other kinds of constrains exist.

A block diagram that represents the datapath of the SHyLoC CCSDS 121.0 Block-Adaptive Encoder is shown in Figure 2.17. The mapped residuals are received sequentially from the Unit-Delay Predictor and forwarded to 3 different components: The mapped FIFO, which stores the mapped residuals for the future coding of these, the snd_extension block, which computes both the length of the second extension option and the gamma values corresponding to each pair of predicted samples, and, lastly, to the lkcomp module, which

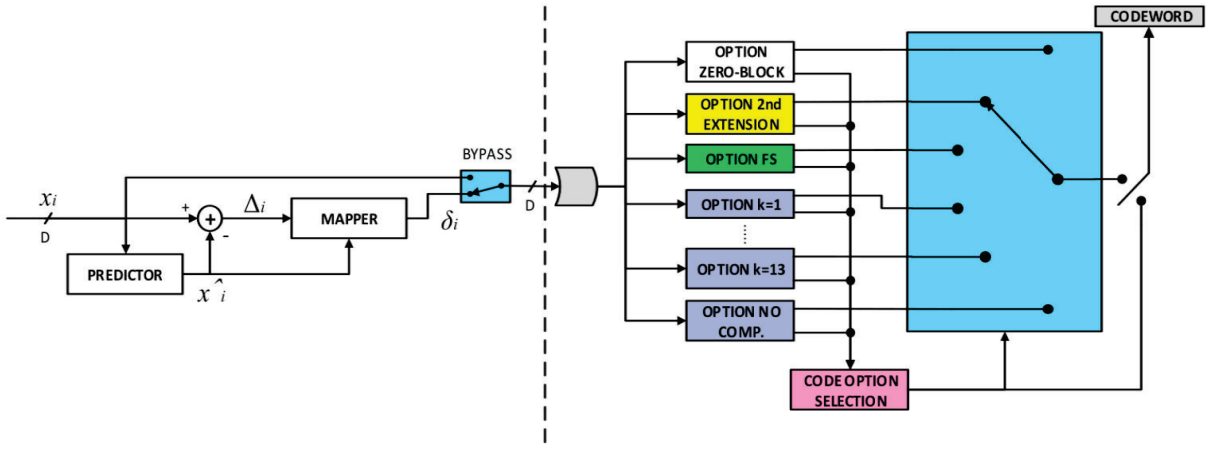

FIGURE 2.16: SHyLoC CCSDS 121.0-B-3 IP General Architecture [13]

computes the length of each coding option, except for second extension. optioncoder module selects the final winning option (the one that yields the shortest codeword) and transmits it to the fscoder module, which is responsible for building the codeword corresponding to each of the blocks of mapped residuals. These codewords are grouped and inserted into the final bitstream by using some different components (splitpacker, splits FIFO, fundamental sequence splitter, packing_final), which is finally sent out through the IP Interface. This internal pipelining allows to achieve processing rates of 1 sample per clock cycle

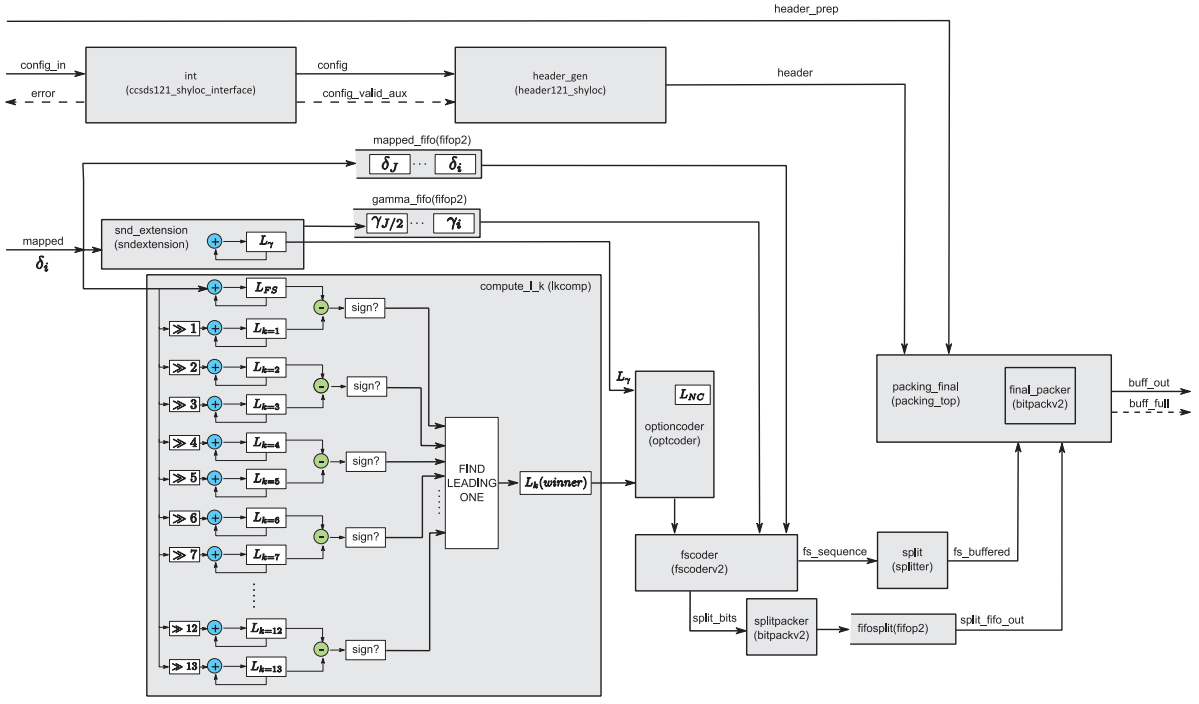

FIGURE 2.17: SHyLoC CCSDS 121.0-B-3 Block-Adaptive Encoder Datapath [13]

2.3.2 SHyLoC CCSDS 123.0-B-1 Compressor IP

The CCSDS 123.0-B-1 standard [15], which defines a lossless data compressor, is specifically designed for multispectral and hyperspectral images. It employs a predictive preprocessing stage to reduce correlation among input samples, resulting in efficient compression. In terms of compression ratio, experimental results demonstrate that the CCSDS 123 standard competes well with other state-of-the-art algorithms. It strikes a balance between coding performance and computational complexity, making it an optimal choice for hardware-constrained projects that still need to achieve great compression ratios [12].

Chapter 3

Architecture Design

3.1 Outline

During this chapter a thoroughful explanation of each part of the developed architecture will be given. A general overview of the design, along with insights of the key parts and design decisions that were made, is offered in Section 3.2. Some design constraints along with its respective justifications are included in Subsection 3.2.2, followed by some brief commentaries related to the expansion possibilities that the developed architecture offers, in Subsection 3.2.1. Firstly, the parallelized Unit-Delay Predictor will be shown in Section 3.3. The new component that has been exclusively developed to interconnect the parallelized predictor and encoder, the Post-Predictor Dispatcher, is architecturally described in Section 3.4. In Section 3.5, a walkthrough across the several components of the block-level entropic encoder will be provided. The data interfaces that have been developed for the design will be extensively explained in Section 3.6. A brief introduction to the AXI4-Stream protocol, which has been implemented in both interfaces, will also be offered in Section 3.6. Following these, the configuration parameters will be described in Section 4.3. Lastly, the conclusions obtained during the design and implementation of the design are given in Section 3.7.

3.2 Overview of the implemented architecture

As mentioned in previous chapters, this project is based on the SHyLoC CCSDS 121.0-B-3 IP Core [12]. As it has been more deeply explained in Chapter 2, this IP Core supports a wide range of configurations defined by the standard, and it can be configured during both compile and run time.

The SHyLoC CCSDS121-IP, though it is internally pipelined to achieve a maximum throughput of 1 sample per clock cycle as previously explained in Subsection 2.3.1, was not thought to simultaneously process more than 1 sample, which limits its final performance for highly constrained applications that require on-the-fly data compression. According to the SHyLoC documentation [16], maximum achievable data rates by the CCSDS121-IP are always lower than 3 Gbps (1.8 Gbps on a Xilinx Virtex-5 XC5VFX130T and 2.6 Gbps on the more novel Kintex UltraScale XCKU060 FPGA).

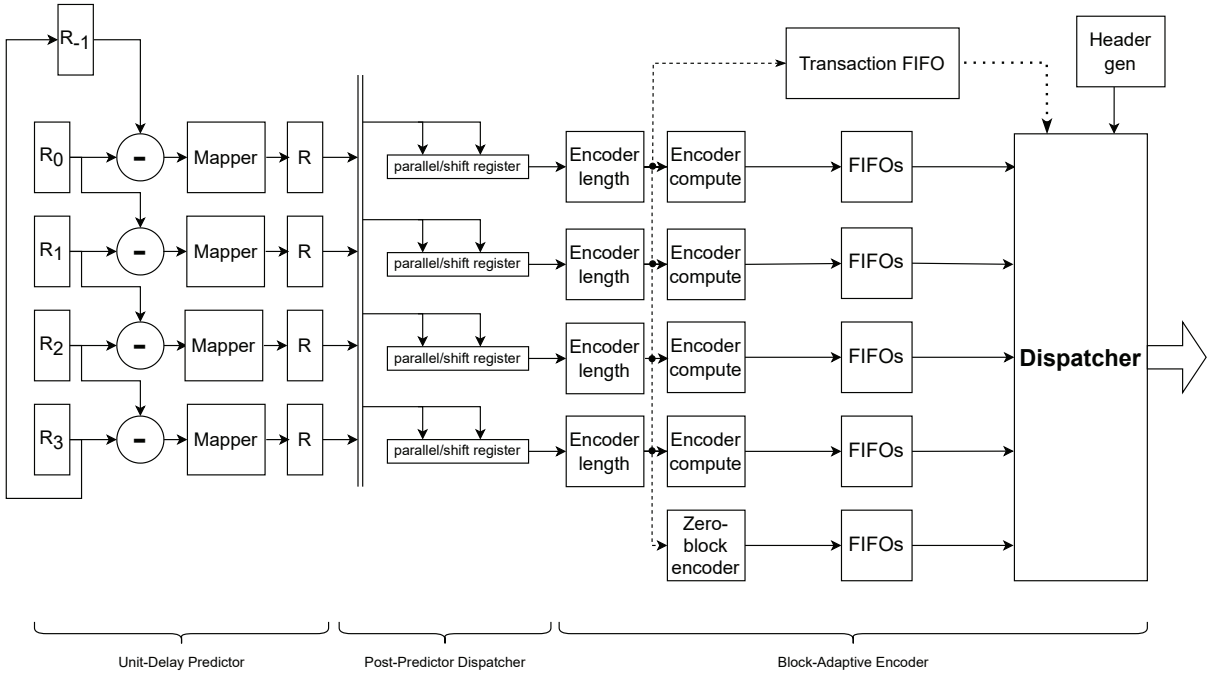

FIGURE 3.1: Initial design of the parallelized architecture

The first approach of the parallel design can be seen at Figure 3.1. The parallelized scheme consisting of 4 processing lanes is already defined, but the level of detail is quite limited. It shows clearly that 4 samples are received per clock cycle, and predicted blocks of samples are internally dispatched to any of the coder processing lanes. Once these have been coded, a dispatcher is in charge of writing the resulting CDSes into the output bitstream. The predictor scheme has not changed significantly from the initial architecture shown in Figure

3.1, thanks to the simplicity of the applied operations. In the other hand, the encoder architecture has been deeply studied and expanded, to address all the requirements and challenges of parallelizing what was originally a fully serial architecture. In Section 3.5, an extensive explanation of how all its parts have been designed, along with the general encoder block diagram, is given, in addition to how these interact with each other.

In order to be able to process bitstreams at higher data rates (e.g., targeting 8 Gbps), a highly parallelized architecture, in which all its inner components are replicated as shown in Fig. 3.2, which shows a general representation of the final parallel architecture. Once designed, this architecture was described in VHDL. This new architecture has been named after Parallel121.

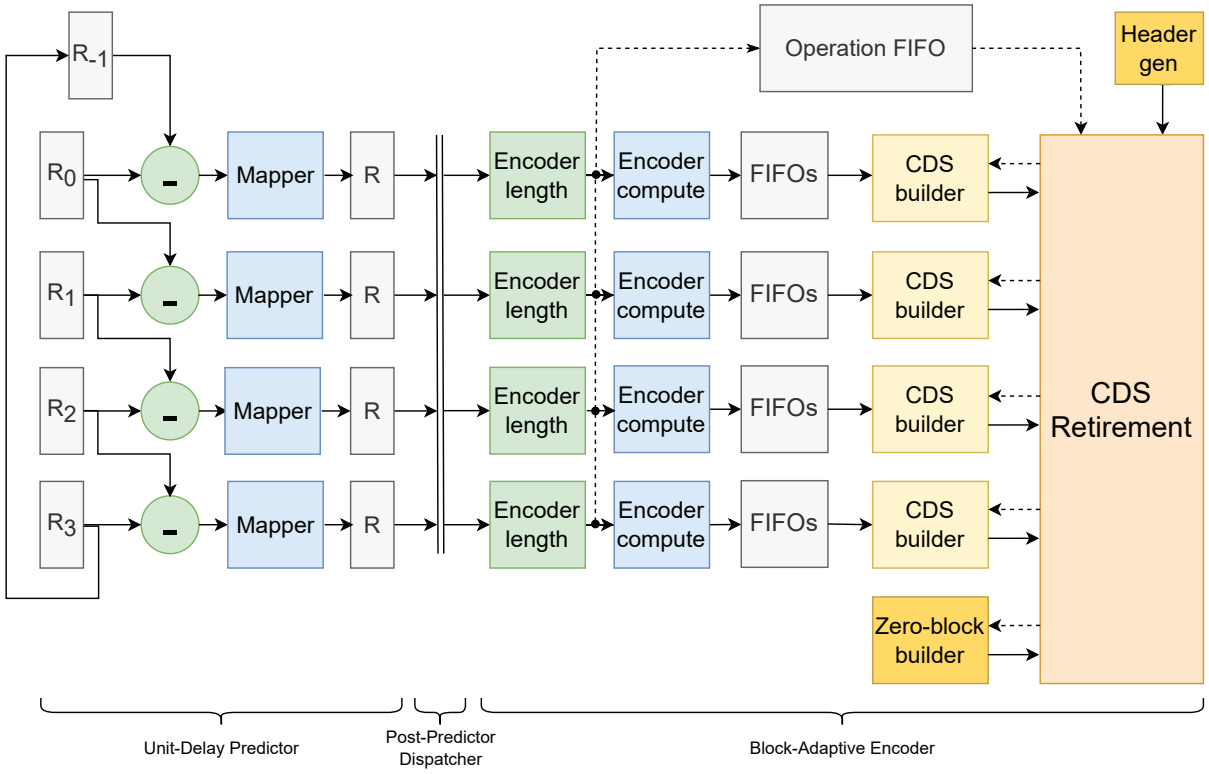

FIGURE 3.2: Parallel architecture based on the SHyLoC CCSDS121-IP

A major change is that the parallelized design implements an operation-based control, which radically differs from the flow-based SHyLoC control. In the latter, its components do not have much notion of what they are actually processing, as the purely serial architecture makes this not necessary at all. This information becomes necessary in the parallelized design, due to the possibility of some lanes processing data faster than others, thus writing the CDSes out of order in the output bitstream. This situation may take place when generating Zero-Block CDSes, as the timing of these differs from the rest of codification

options, which are processed on the main 4 processing lanes. The operation-based control allows the control unit to retire, that is, write the CDSes into the output bitstream, in a completely controlled order, thanks to the use of unique auto-incremental identifiers that are assigned to the blocks of data as these are received from the predictor.

It is important to remark that the design was not built from the ground. SHyLoC CCSDS 121.0-B-3 Compressor IP has been taken as the base architecture. This has been crucial for the project, as the different components of the original pipeline have already been exhaustively verified, easing the integration and verification of the final design. While it is true that part of the pipeline of the parallelized design has been designed by replicating the different components of the original one, other parts of the design have been implemented specifically to implement the new operation-based control scheme. The designed architecture receives 4 samples per clock cycle through an AXI4-Stream input interface, which are processed simultaneously by individual Unit-Delay predictors. Sample size is fixed at 16 bits, although this can be modified at compile time. Note that even though all 4 predictors work in parallel, data dependencies between consecutive samples are always satisfied. This is accomplished by adding inter-predictor connections as well as an additional register to store the latest sample of a group, needed to predict the first sample of those received in the next clock cycle.

Once the 4 samples have been predicted, their mapped prediction residuals are internally dispatched to one of the four encoder processing chains. This design fixes the J parameter to a value of 8, meaning that each encoding chain processes blocks of 8 preprocessed samples. The internal dispatcher is responsible for accumulating the preprocessed samples during two consecutive cycles.

The block-adaptive encoders, deeply explained in Section 2.2.3, work independently of each other, except for the zero-block encoding option, in which case an additional processing lane will be in charge of its coding. Although it may not be obvious at a first glance, zero-block computation differs drastically from any of the other encoding techniques as it introduces dependencies between an undetermined number of consecutive all-zeros blocks. This implies to break the autonomous scheme of the encoding parallel chains, thus complicating not only the control but also the datapath, as these may be injected at the output at any time from an independent block-adaptive encoder, as shown in Fig. 3.2.

Header insertion into the output bitstream has also been included in the parallelized scheme of the compressor. A header generation module, which is responsible for preparing the header with the necessary fields that describe the parameters that will be used for the

compression, acts like a completely independent lane. That is, the architecture conceptually includes 6 different processing lanes: The 4 basic encoding lanes, the special Zero-Block encoding lane and the header lane.

The configuration parameters of the design have been strictly limited to a predefined subset. As explained before, J parameter is fixed to 8, while the size of the input samples is defined at compile time. For the packing of the final coded blocks, only file format is supported. Both file format and packet format, which is the alternative packing format defined in the standard, are explained in 2.2.4.

The only runtime configurable parameters are the data input dimensions (N_x, N_y, N_z) and the reference sample interval, that is, the number of coded blocks of samples after which a raw sample is introduced into the output bitstream. To send these configurations to the IP an Advanced High-performance Bus (AHB) configuration interface is offered as a basic option, altough an AXI4-Lite to AHB bridge migth also be instantiated in case a more modern interface is required.

3.2.1 Scalablility

The architecture is fully scalable. New derived architectures created by replicating the number of inputted samples and the processing lanes (both the predictor and encoder processing lanes) of this design. These high performance architectures could be useful for applications that require to process extreme throughputs. While it is true that the parallel121 architecture has been completely designed to process blocks of 8 samples, its architecture has also been prepared to be easily upgraded to a more aggressive parallelization scheme. Anyway, it has to be highlighted that some parts, mainly the Finite-State Machine (FSM), would need to be slightly adjusted to properly fit the new requirements of the extended design.

As shown in Figure 3.3, the parallelization scheme and the operation-based control remains identical to the architecture developed in this project, simplifying the development of the new architecture by effectively reducing the time needed for the design, implementation and verification. Most of the modules of this project have been designed to be easily expandable, as these use a parameter that determines the number processing lanes, $LANES\_GEN$, whose default value is $J\_GEN \div 2$. Extreme performance architectures may be created by expanding this parallelization scheme.

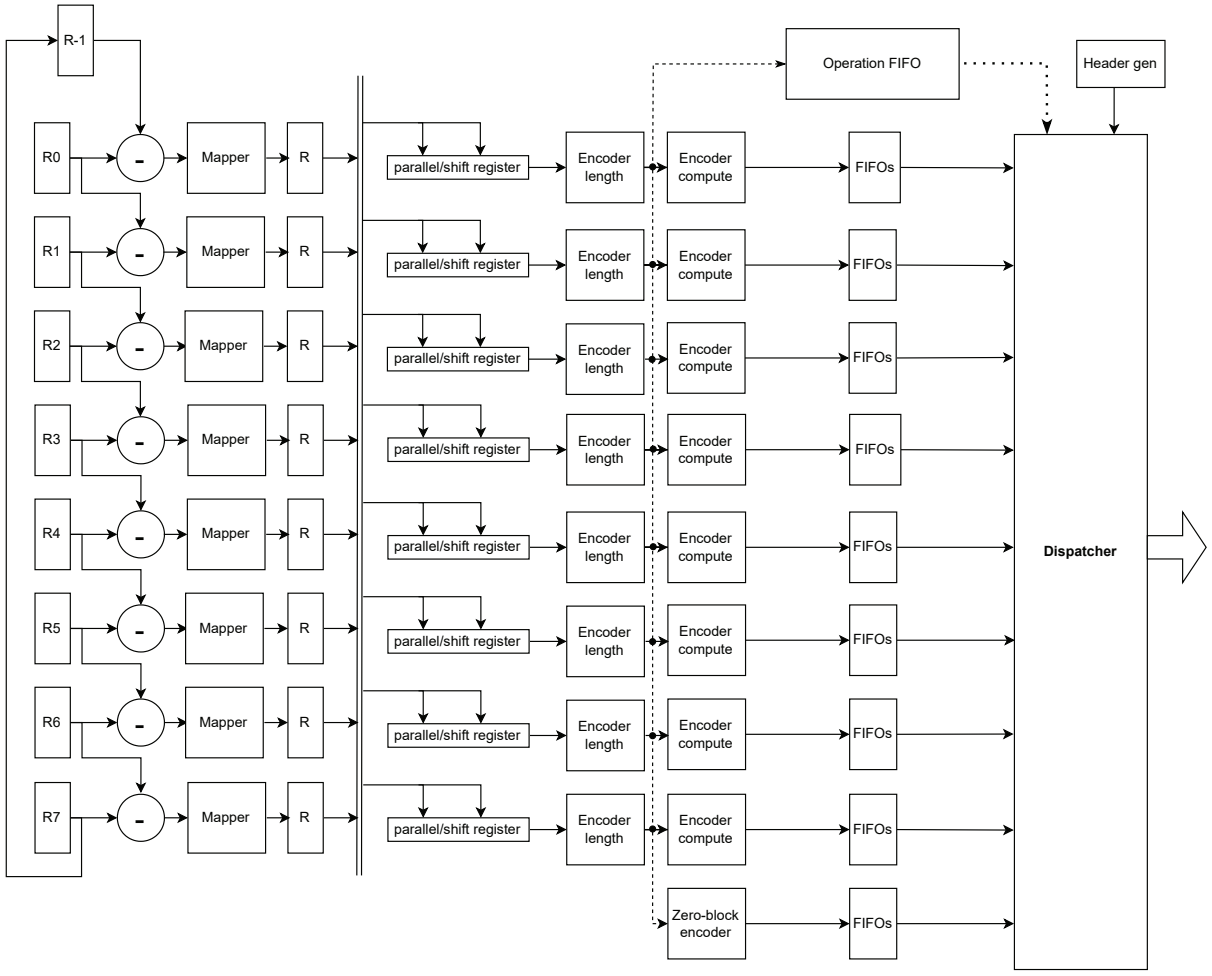

FIGURE 3.3: General diagram of the extended compressor with 8 parallel processing lanes

3.2.2 Design considerations

In this section, some comments about limitations of the design are provided.

3.2.2.1 Considerations about the parameter $'J'$

Initially, the parameter J is fixed to a value of 8. This was a decision taken from the start, as the original design already included the 4 processing lanes and processed a group of samples every two clock cycles. A greater J would require some additional adjustments in the communication between the different encoder FSMs as well as between the Post-Predictor dispatcher and the predictor and encoder.

3.2.2.2 Considerations about the parameter *'Reference Sample Interval'*

The reference sample insertion interval is limited by design to multiples of the segment size, that is, multiples of the 64. This design decision was made to effectively reduce the overall complexity of the control, as the generation of the Zero-Block through the usage of the operation mechanism would be way more complex.

By limiting this parameter, we manage to simplify the architecture in two different ways: First, the insertion of reference samples may occur exclusively on the first processing line, which translates in that the rest of lanes need less components and a simpler control logic. Second, we avoid the interruption of the generation of Zero-Block CDSes that may happen whenever a reference sample is inserted, leading to simplify the Zero-Block CDS generation process as less possibilities have to be considered.

3.3 Unit-Delay Predictor

As explained before, the design of the parallelized Unit-Delay Predictor has not varied much during the design of the architecture. The components of the SHyLoC 121.0 original predictor have been replicated as shown in Figure 3.4. Every single clock cycle, 4 samples (half of a block) are read from the input interface. Every sample is predicted using the previous one (see the shortcircuits in the datapath at Figure 3.4). For the prediction of the upper sample, the last sample from the previous prediction, that is, from the last clock cycle, is stored in a register. Once predicted, every prediction residual is mapped onto a positive, unsigned integer, which is then sent to the post predictor internal dispatcher.

Note that the periodical feeding of reference samples, whose importance has already been discussed in Chapter 2, required the implementation of an auxiliar bypass mechanism. When a reference sample is inserted, a multiplexor allows to send the raw reference sample to the post predictor dispatcher instead of its mapped residual. Note also that only the first lane, lane 0, includes this bypass mechanism, as the the reference samples can be exclusively received as the first sample of a block.

The predictor is composed of two well differentiated modules: **predictor_fsm** and **predictor_comp** as shown in Figure 3.5. The predictor FSM directly interacts with the input interface First-In First-Out (FIFO) by reading groups of samples. On the opposite side,

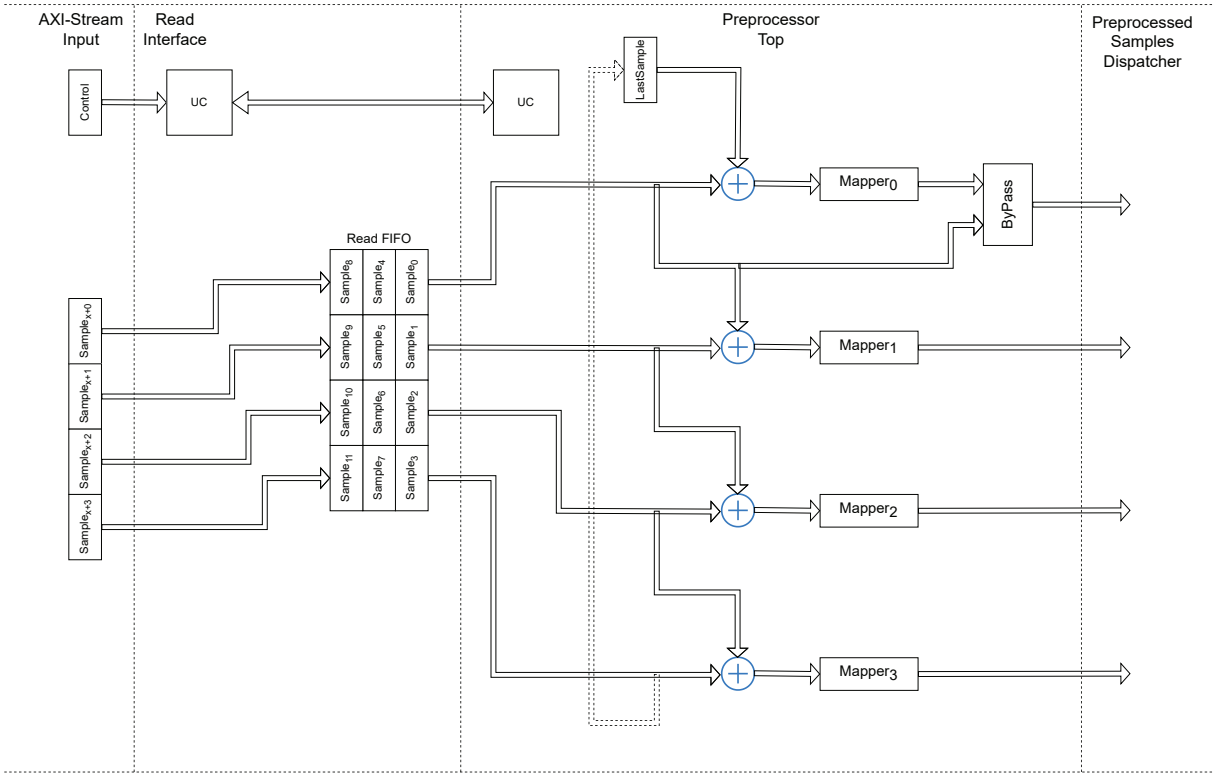

FIGURE 3.4: Parallelized architecture of the CCSDS 121.0 Unit-Delay Predictor

the preprocessed groups of samples (its mapped prediction residuals) are passed to the post-predictor dispatcher with a two-way ready/valid handshake. This mechanism has been introduced due to the possibility of occasional pipeline detentions at the encoder (previously it was a one-way valid signaling mechanism), whose cause will be discussed later. As shown in the diagram, a FSM submodule is responsible for the control of the module interfaces, and to indicate the components submodule, which is responsible of applying the transformations explained at Subsection 2.2.2, that a reference sample is included in the group of samples.

3.4 Post-Predictor Block Dispatcher

The Post-Predictor Block Dispatcher is a crucial module in this parallelized architecture. While the predictor processes 4 samples per clock cycle, each of the encoder processing lanes processes whole J-sample blocks in a serial manner. For this reason, an efficient interconnection module that stacks all the samples from a block, and immediately dispatches these to one of the encoder lanes is crucial to achieve the target throughput.

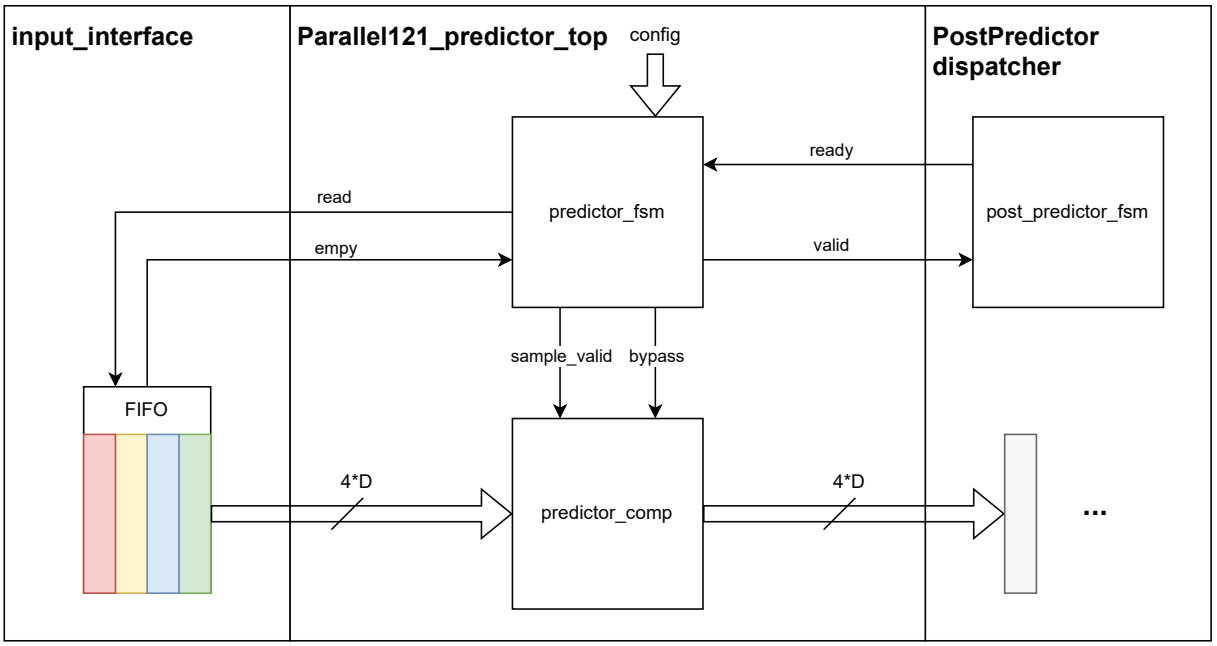

FIGURE 3.5: Top view of the Parallel121 predictor module

An initial 2-level multiplexed routing design was created (see Figure 3.6). While this design shows clearly what the target behaviour is, the usage of multiple multiplexors and individual write signals for each of the registers (one register per mapped residual), introduced an unnecessary complexity into the design.

For this reason, a second version of the dispatcher, which can be observed in detail at Figure 3.7, was designed. The multiplexors are completely removed, and there are 8 registers, each of which holds 4 mapped residuals. Two consecutive registers correspond to a single block, and both are connected to the same lane. The data received from the predictor is joined and driven to the input of each of these registers, and these are written in a circular manner.

Whenever two consecutive registers are written with new mapped residuals, the valid signal of the corresponding encoder lane is raised. The dispatcher then blocks waiting for the assertion of the ready signal (ideally it is always risen, meaning that the encoder lane can receive new blocks and there is no need to stop the previous pipeline). This ready-valid mechanism is identical to the one explained in section 3.6.1.

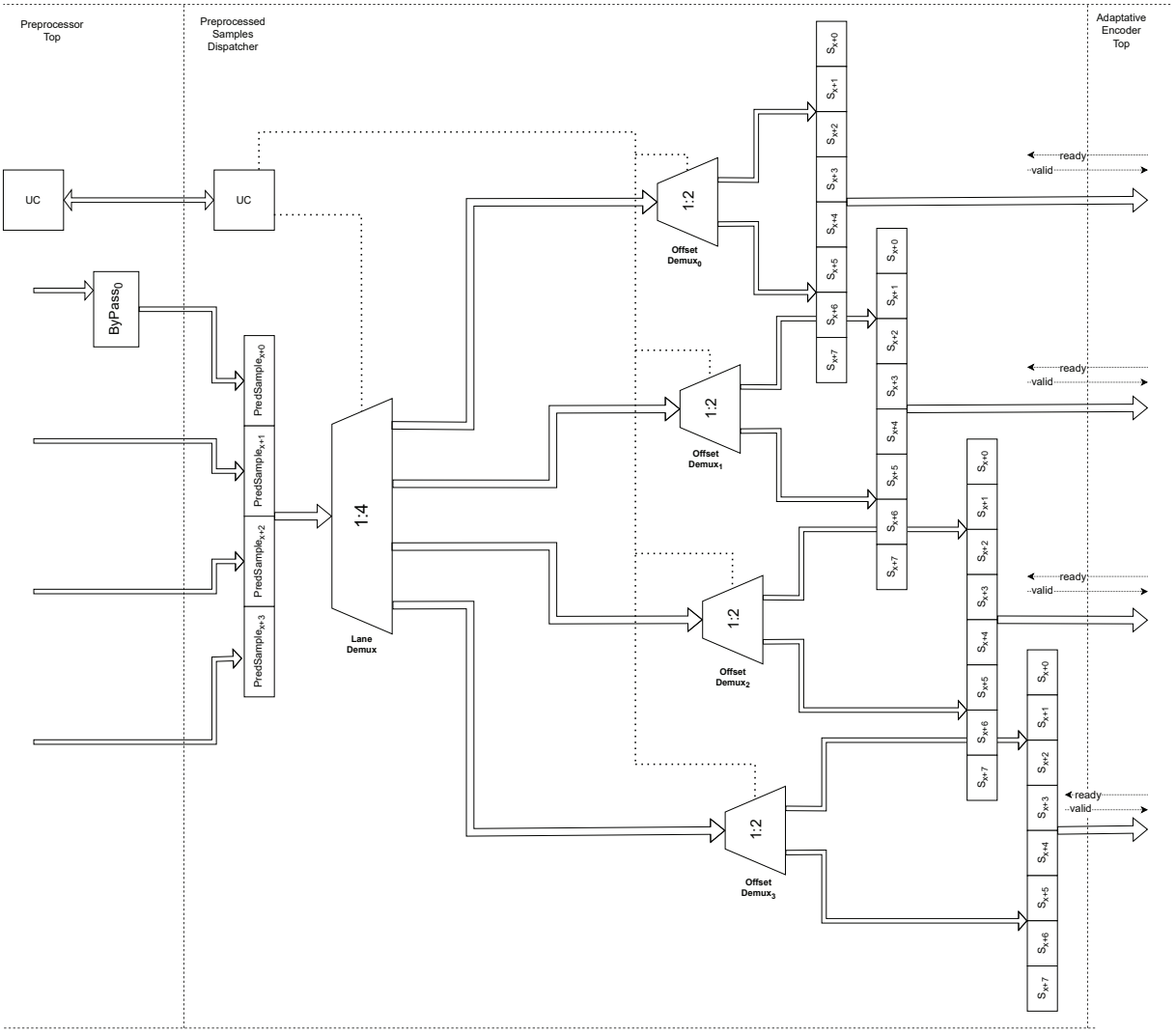

FIGURE 3.6: Initial multiplexor-based design of the Post-Predictor Dispatcher

3.5 Block-Adaptive Encoder

The Block-Adaptive Encoder receives blocks of preprocessed samples from the Post-Predictor Dispatcher, and is responsible for the encoding of these. Each of the coding lanes processes blocks of samples in a serial manner (similar to how blocks are processed in SHyLoC), but some major changes have been introduced to effectively parallelize the design.

The main change is that a new operation-driven control scheme has been introduced. Once the codification option to be applied to a block of samples is known (output of FSM1) an operation is generated, in addition to start generating the FS and the corresponding K-Splits of the block (in case that its codification option has K-Splits). An operation is

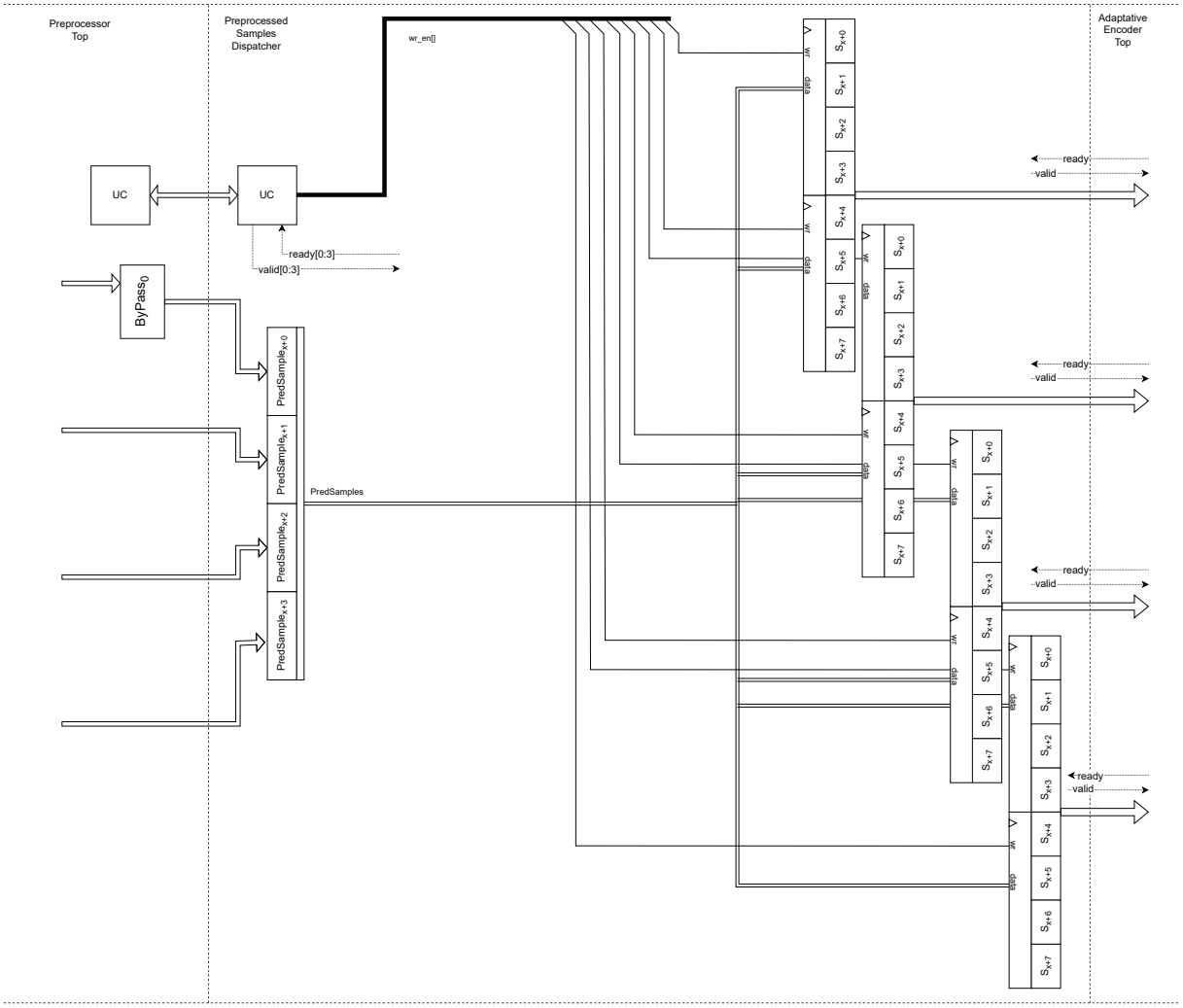

FIGURE 3.7: Final design of the Post-Predictor Dispatcher

a data structure that holds an unique auto-incremental identifier, a lane identifier that helps to know where this block is being processed, additional information related to the selected coding option and the final CDS size. These operations are inserted into the operations FIFO, which is later read by the FSM4, responsible for internally dispatching these operations to the internal CDS Builders (FSM3), that will build the CDS into an intermediate register. These intermediate registers are retired into the output buffer in an ordered manner, that is, the CDS order is identical to the one in which its corresponding samples had been previously received through the input interface.

The main reasons that encouraged the inclusion of this operation-driven control are the following:

- Retiring of the CDSes in order. The unique IDs help the FSM4 to recognize which CDSes must be written first into the output buffer. Some coding options may be faster (the intermediate CDS is built faster) than others. For example, a second extension coded CDS takes less cycles to be completely generated than other coding options. This can also happen with the Zero-Block coding option.

- Zero-Block CDS generation based on operations. As it will be further explained in subsection 3.5.5, the generation of the Zero-Block is one of the main challenges on a paralellized design. An centralized operation-based control mechanism helps easing the construction of these particular type of CDSes.

- Support to future expanded designs. An operation-based design helps to coordinate more advanced designs (i.e. duplication of the datapath processing lanes to easily support compression with J=16). With some minor improvements and changes, this control scheme must allow to control more complex, high-performance architectures.

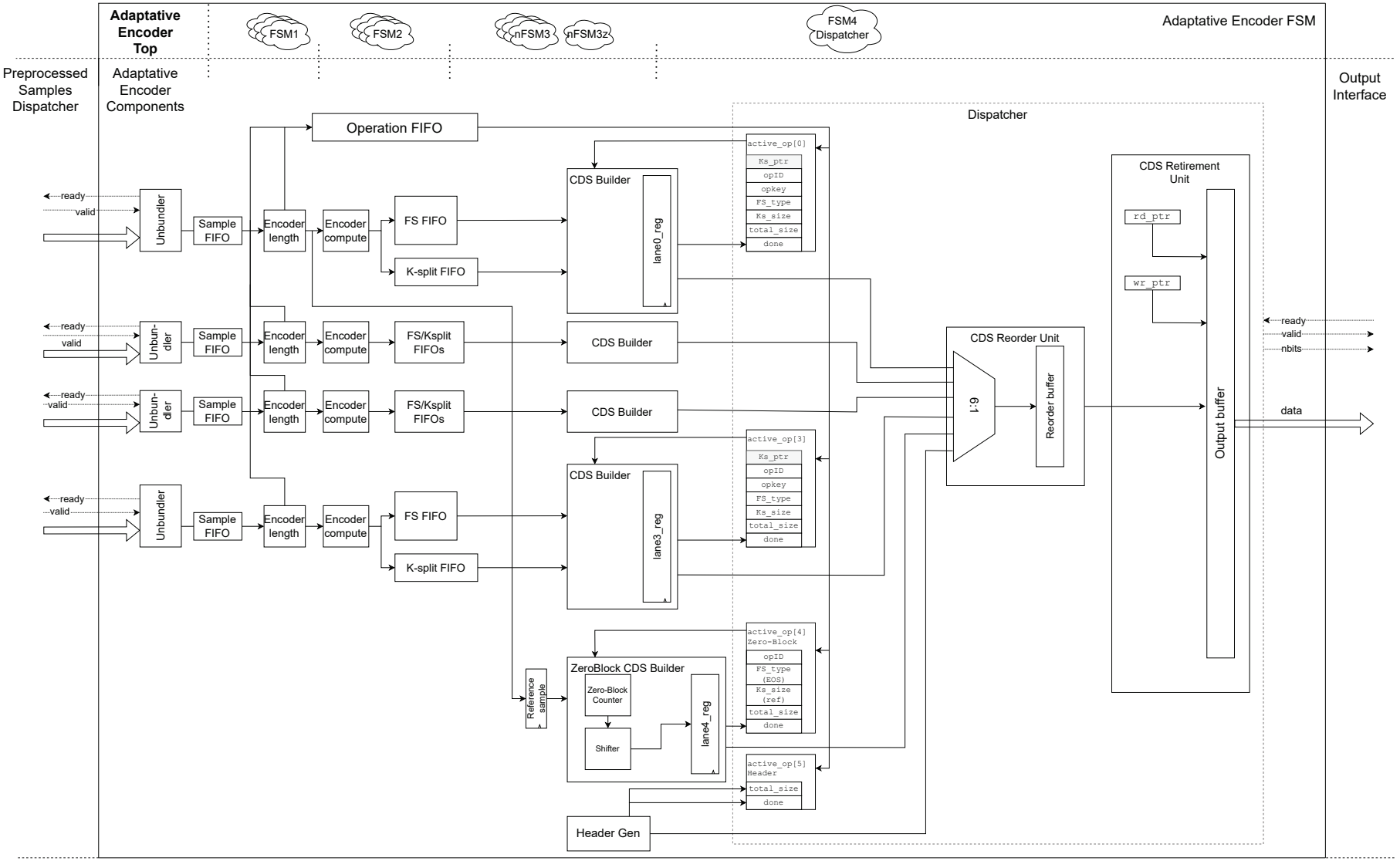

FIGURE 3.8: General overview of the Parallel121 Block-Adaptive Encoder architecture

Like the unit-delay predictor, the control logic and the datapath of the block-adaptive encoder are divided into two separate submodules ('encoder_fsm' and 'encoder_comp') as shown in Figure 3.8. All regular processing lanes work independently from each other, and both the Zero-Block CDS Insertion and the Header Insertion are processed in completely independent components, which are closely related to the operation based mechanism. The initial phases, CDS Length Calculation and Option Selection and CDS Computation, governed by FSM1s and FSM2s, are similar to the original SHyLoC processing phases. In the other hand, the next phases, managed by the nFSM3s, FSM4 and Reorder and Retirement Units, completely differ from the final packing that can be found in the original SHyLoC pipeline.

The general data flow of the encoder is shown in Figure 3.9. In the following subsections, all the modules that perform different processing operations over the preprocessed data will be further explained, starting with the Block Unbundler component (3.5.1) which is followed by the CDS Coding Option Selection and Length Calculation (3.5.2). Next, both the CDS Codification (3.5.3) and the CDS Intermediate Reconstruction (3.5.4 - 3.5.5) phases will be discussed, paying special attention to the differences between regular blocks and Zero-blocks. The rework of the header insertion mechanism will be briefly described (3.5.6). Lastly, the CDS Retirement (3.5.7) phase will be extensively explained, going through both the FSM4 and the final CDS Reorder and Retirement Units.

3.5.1 Block Unbundler

The block unbundler module is responsible for loading the blocks from the post-predictor dispatcher, and extracting the individual samples from these. The components that have been taken from the original SHyLoC 121.0 Compressor work in a sample-by-sample manner, so this module is essential to properly reutilize these.

The architecture of the block unbundler can be seen in Figure 3.10. Although it is optional, a recommended minimal FIFO stores the blocks received from the post-predictor dispatcher. These are loaded into a block register from which the individual samples are extracted sequentially by using a pointer and a multiplexor. If no FIFO is used, then the blocks are loaded directly into this register.

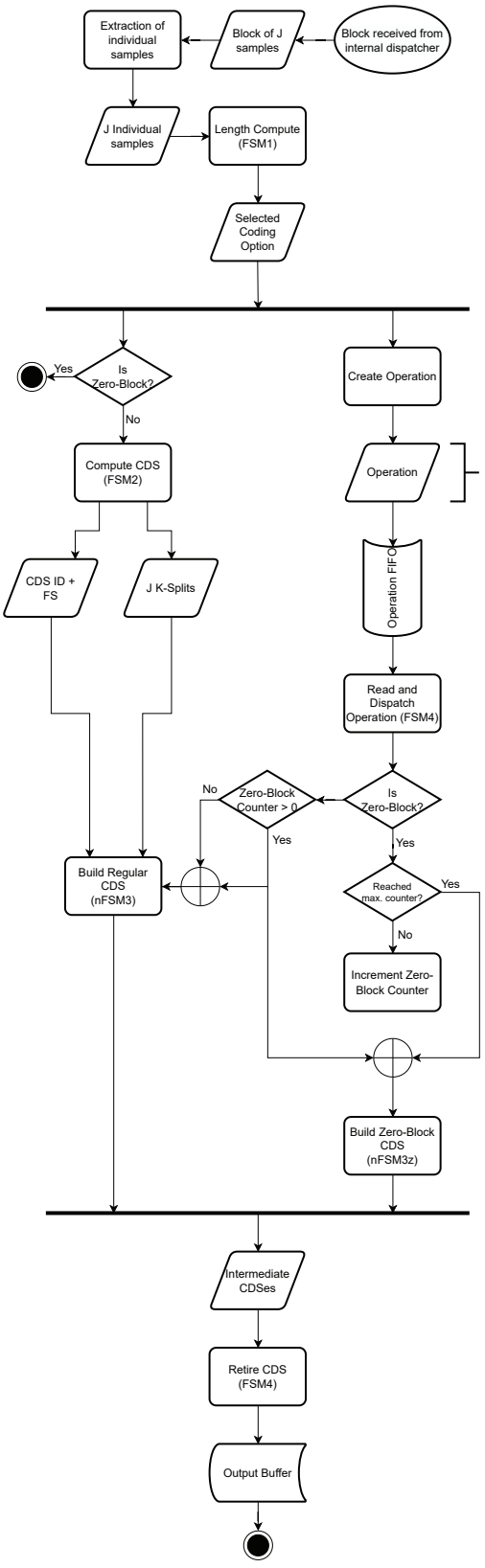

FIGURE 3.9: General data flow in the parallelized design

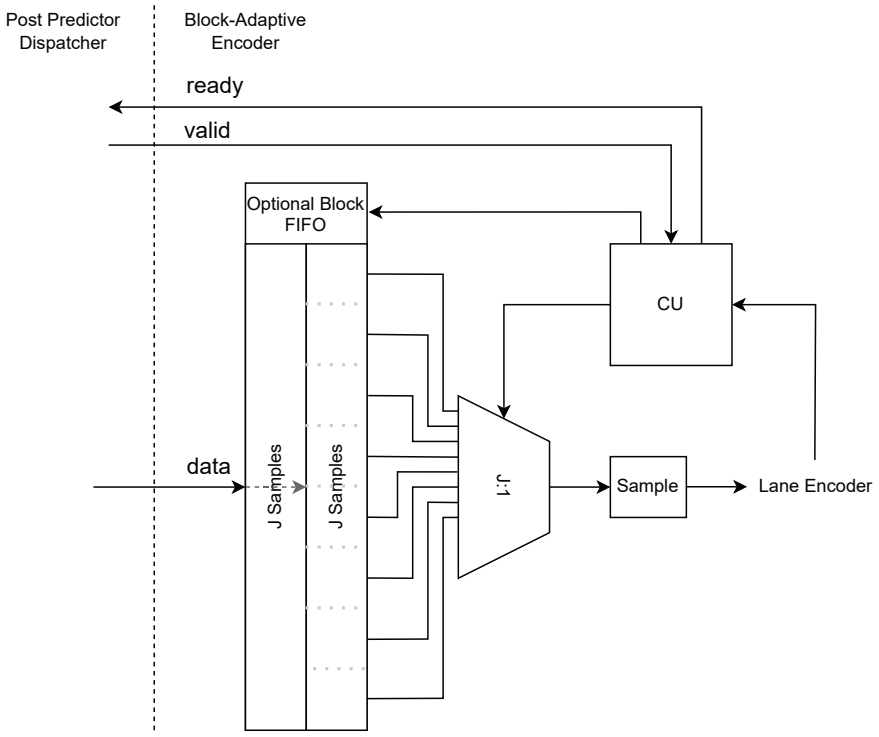

FIGURE 3.10: Architectural Overview of the Block Unbundler Module

3.5.2 CDS Coding Option Selection and Length Calculation

The CDS length calculation is the first step of the whole process of coding the blocks of samples, which are received in a serial manner, one per clock cycle, from the CDS Block Unbundler. For each of the received blocks of samples, the length corresponding to applying every single possible coding option to these is calculated. From all these coding options, the one that offers the smaller length is selected and passed to the next encoder components.

The FSM1 included in the 'encoder_fsm' is responsible of controlling all the submodules that appear in the diagram, coordinating them and assuring that the information is passed correctly from one to another.

Focusing into the components, once the mapped residuals are read from the 'sample FIFO', which acts like a small buffer for the individual samples that are received from the unbundler, these are distributed to three different components. First, these are stored into another FIFO known as the 'Mapped FIFO'. This FIFO will be read from the next phase, the encoder computation, to build the FS in accordance to the option obtained in the this phase. Secondly, these are inputted to the 'snd_extension' module, which calculates the length corresponding to the second extension coding option as well as the gamma symbols

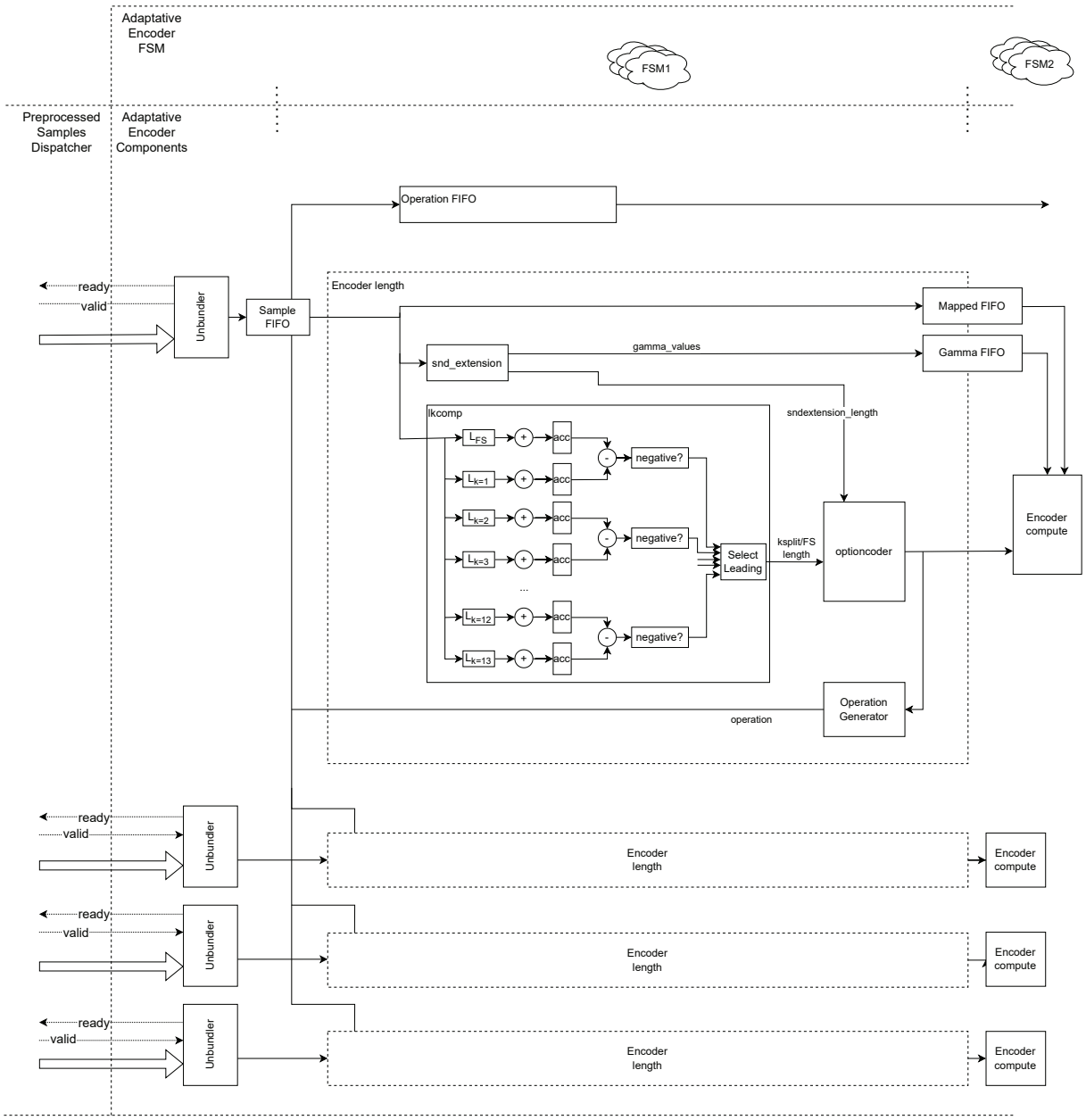

FIGURE 3.11: Architectural Focus in CDS Length Calculation Phase

as explained in subsection 2.2.3.3, which are also stored in another FIFO named after 'Gamma FIFO'. The samples are also passed to the 'lkcomp' submodule. This module calculates the length of all the K-Split possible options (including the pure FS coding option) and internally selects the one that yields the smaller length. This selection is performed by substracting consecutive coding options. The first negative substraction found will mark the optimal length, which will be passed to the 'optcoder' module.

The 'optcoder' module integrates information from all the other modules and obtains the final coding option. Internally, it calculates the No Compression Option length

($ID\ Length_{NC} + J * D$) and detects if the block is a Zero-Block, in which case this will always be the selected option. Both the selected option as well as its corresponding length are sent to the next coding phase, the CDS Codification, but also to the 'Operation Generator'. The latter creates the operation corresponding to the CDS whose coding option has just been selected, and introduces it into the 'Operation FIFO', a key part of the Operation based design. As it will be deeply explained in the next subsections, this FIFO is read at the final stage of the codification ultimately allowing to reconstruct the whole CDS and to insert it into the final output bitstream.

Every lane includes its own FSM1 and the components that have just been explained. These work independently. It might appear that the insertion of new operations into the Operation FIFO could experience some problems, due to multiple lanes wanting to insert operations in the same clock cycle, but there's no problem thanks to the 2 clock cycle processing offset between every consecutive processing lane. Anyways, it is true that the pipeline stop mechanism had to be designed properly to not alter this offset, thus forcing to stop the processing of all 4 lanes.

3.5.3 CDS Codification

The codification phase is responsible for calculating the corresponding fundamental sequence of the CDS being processed, as well as the spare K-bits splits. The main and only component of this phase is the 'fscoder', while the control signaling is performed entirely by the FSM2. See Figure 3.12 for a detailed architectural overview of the CDS Codification phase components.

Originally, the 'fscoder' from the original SHyLoC CCSDS 121.0 Compressor IP processed all the possible types of CDSes (No compression, Fundamental Sequence, K-Splits and Zero-Block). The functionality of the included 'fscoder' has been reduced to not process Zero-Block CDSes. Given the dependencies that consecutive Zero-Blocks holds (multiple blocks might be coded as a single CDS), this calculation has been extracted from the regular datapath, being completely done in an external module as it will be further explained in section 3.5.5. Every regular processing lane has its own 'fscoder' and FSM2, which work independently from the others.

Once the coding option has been selected and sent to the fscoder, both the mapped samples and the gamma symbols start being read from the corresponding FIFOs. The components

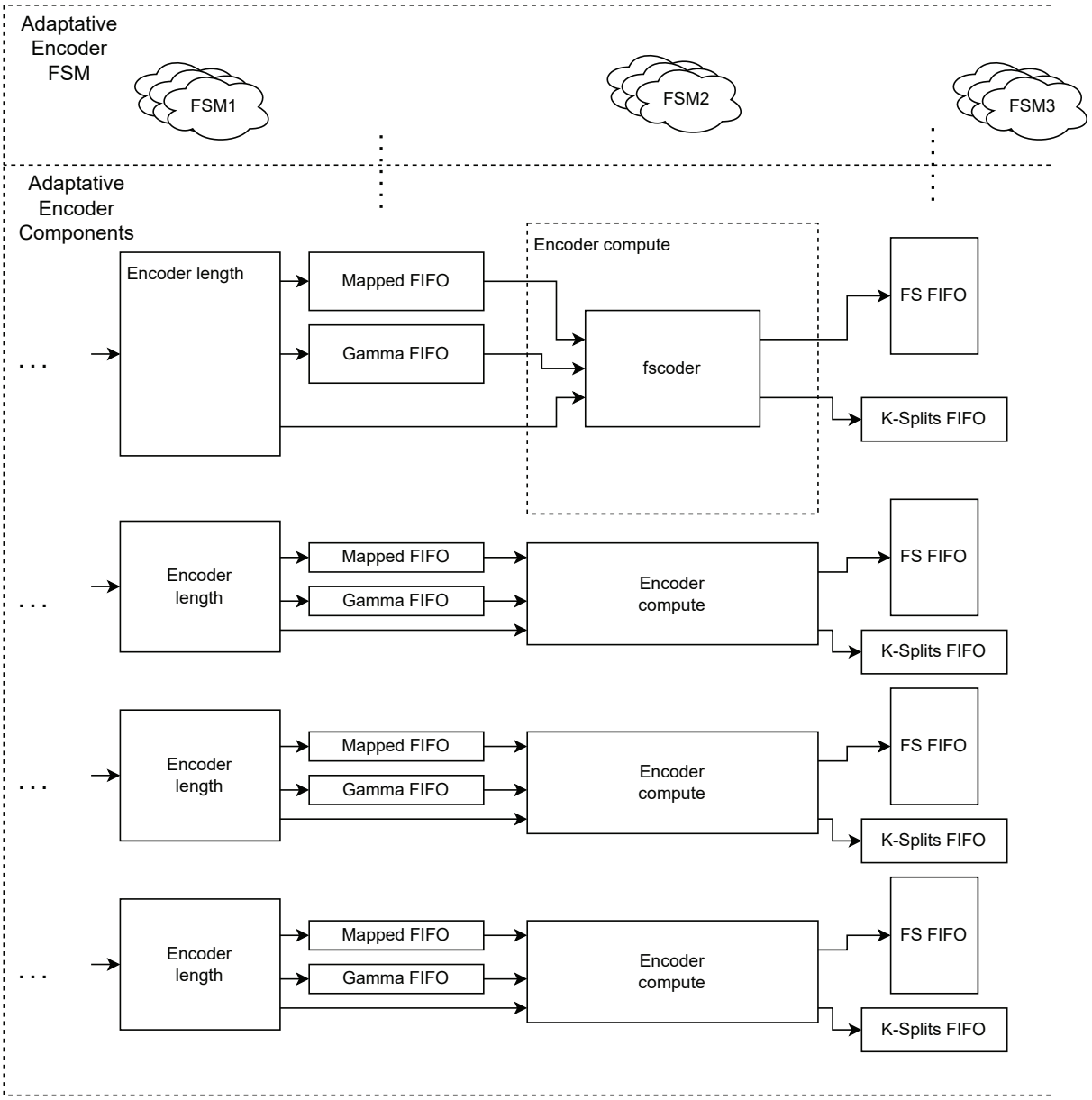

FIGURE 3.12: Architectural Focus in CDS Codification Phase

incrementally build the fundamental sequence (Except in case that the no compression is selected, where this is not necessary at all). If the coding option includes splits of any size, then these are outputted to the K-Split FIFO, which stores them for the next coding phase. If no compression option is selected, then the K-Split FIFO stores the entire samples instead. Once all J mapped samples (and the J/2 gamma symbols) are read and processed by the 'fscoder', the FS word (which also includes the CDS Codification ID corresponding to the applied coding option) is outputted and saved into the FS FIFO.

3.5.4 CDS Intermediate Reconstruction

The CDS Intermediate Reconstruction is the last phase of the CDS coding, as the next ones are related to the inclusion of these into the output bitstream. Following the same scheme as the previous phases, the functionality is again divided into the components submodule, in which the proper reconstruction is done, and the FSM submodule, which starts and controls this process.

The state machine that controls the reconstruction is known as the nFSM3 (new Finite State Machine 3). This name is given to differentiate it from the original SHyLoC CCSDS 121.0 Compressor IP FSM3, which was responsible for the packing of the previous information to build the final bitstream. The reason for removing the latter is that it didn't make sense to include it in the parallelized design, as it might cause some registers to be written from up to 5 different sources simultenously (the 4 regular processing lanes plus the ZeroBlock special lane). This was the main cause to design this new phase, in which the CDSes are completely reconstructed and written in order to the output bitstream, as shown in Figure 3.14.

The only component that is used in this phase is a completely new module known as the 'CDS Builder'. This module builds the whole CDS into an internal register. This is done by using the information included in the active operation struct that is received from the FSM4, which is responsible for reading the Operation FIFO and internally dispatching these operations to its corresponding FSM3s by loading the active operation struct registers.

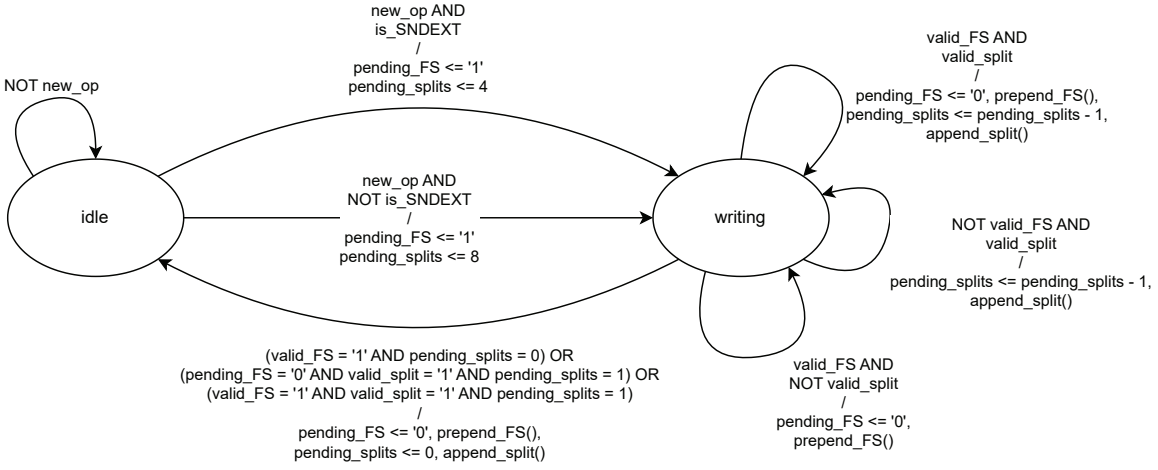

Figure 3.13: New Finite State Machine 3

Once a new active operation has been received (the 'new_op' signal is high), the nFSM3 starts reading both the FS FIFO (a single read) and the K-Split FIFO (J reads). This data is written by the 'CDS Builder' into its corresponding positions, finally leading to a full reconstruction of the CDS. Once it is fully built, the done flag of the lane is raised to point out FSM4 that it can be retired and written into the output bitstream. A general Mealy FSM that describes the functionality of this part can be found in Figure 3.13.

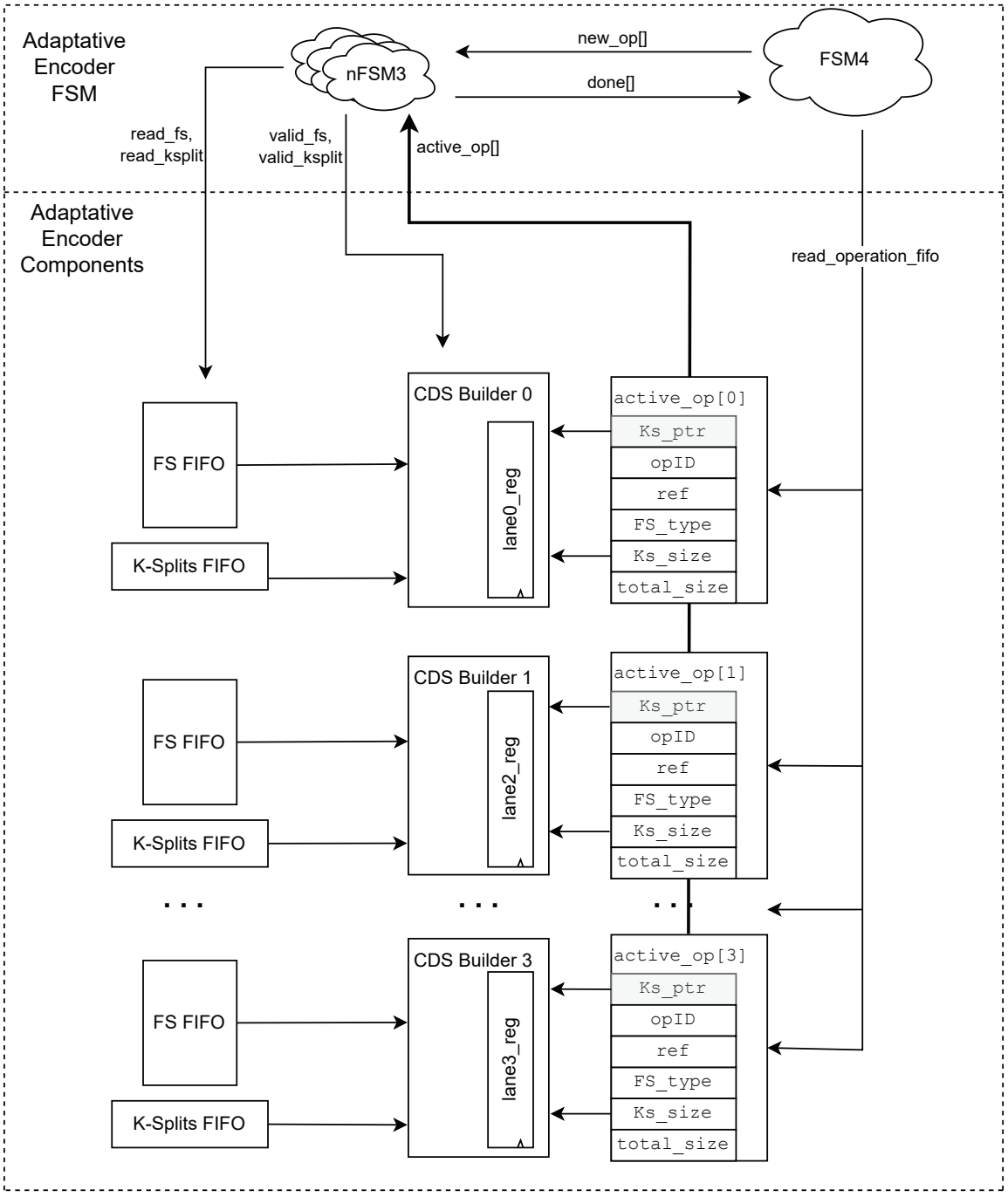

Figure 3.14: Overview diagram of the CDS Reconstruction phase

3.5.5 Zero-Block CDS Codification

As explained before, the Zero-Block CDS Codification can't be done in a fully distributed manner due to the data dependencies that may appear between consecutive Zero-Blocks. In this case, the centralized Operation-based control shows up as an adequate solution to ease this processing, along with a specialized nFSM3z (new FSM3 Zero-Block) and a 'Zero-Block Builder'. These parts of the design are referred to as the Zero-Block processing lane, but it must be remarked that these FSM and components, as well as its data flow are completely different to the regular processing lanes that have been explained in the previous sub-chapters.

Once a Zero-Block operation is taken from the operation FIFO, this is loaded into the Zero-Block lane active operation register, and the new_op signal of the Zero-Block processing lane is raised. Every clock cycle that the new_op signal is high, nFSM3z indicated the 'Zero-Block Builder' to increment its internal counter by 1. If the nFSM3z receives a new operation in which the EOS (End Of Segment) flag is set, or a regular, non-Zero-Block operation is internally dispatched by the FSM4, then the Zero-Block Intermediate CDS is built, and the done flag is raised once the latter is ready to be written into the output buffer.

The Zero-Block CDS might include reference samples like any of the other codification options. This requires an additional communication mechanism between the first processing lane, lane 0, and the special Zero-Block processing lane, as the latter is completely isolated from the regular flow of the mapped samples. The reference samples are always received in the first lane, and these are passed to the 'Zero-Block Builder', which stores them in case that it needs to include them into the Zero-Block CDS.

The component that performs the building of the Zero-Block is an independent submodule known as the 'Zero-Block Builder'. This module builds the Zero-Block fundamental sequence by right shifting a logic vector that holds a single logic one in its leftmost position. This specialized component is also able to also insert the reference samples as well as the special case EOS fundamental sequence codeword, thus covering all possible Zero-Block CDS generation cases. See Figure 3.15 for an architectural overview of the Zero-Block CDS Reconstruction phase.

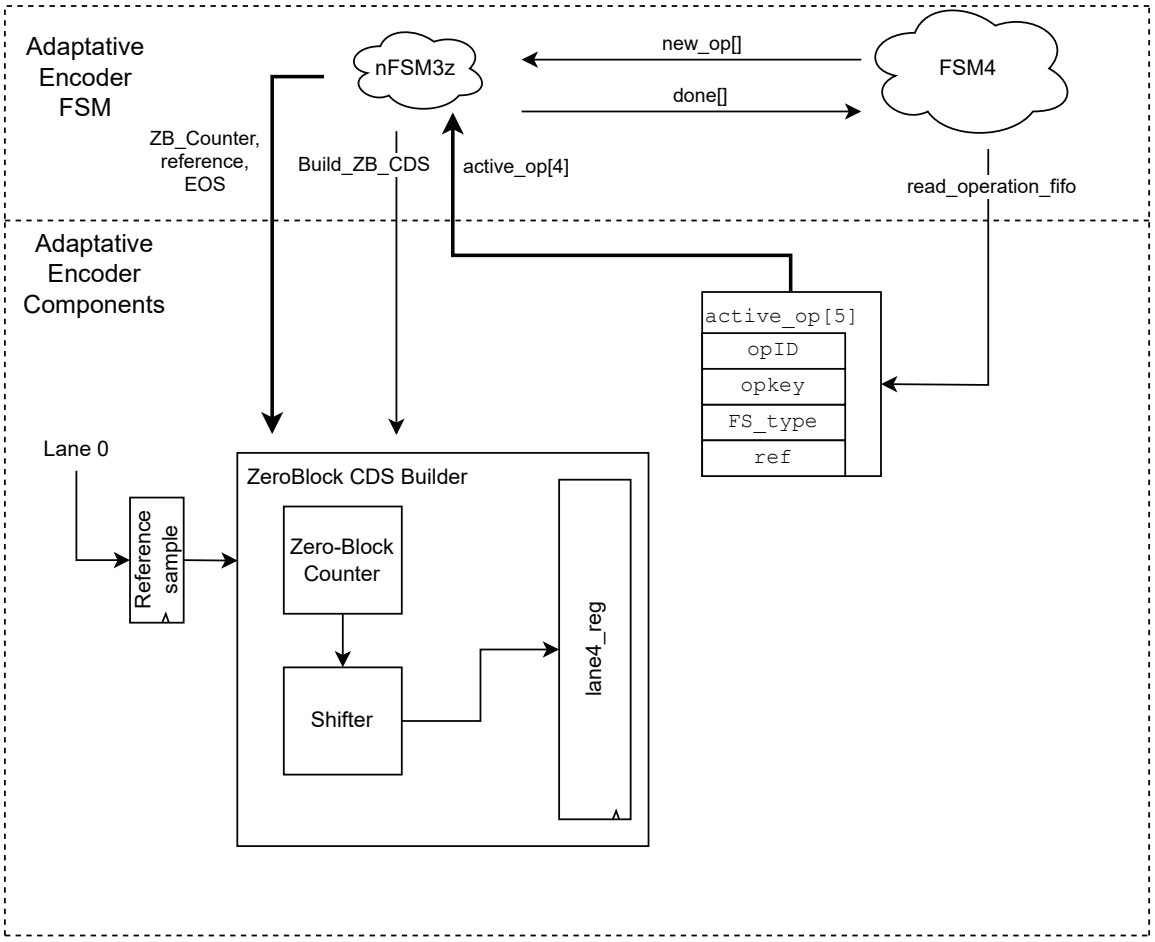

FIGURE 3.15: Overview diagram of the Zero-Block CDS Reconstruction phase

3.5.6 Header Insertion

The operation-driven control scheme eases as well the insertion of headers into the final bitstream, as these are considered to be additional information sources. This means that the header insertion can be considered as an independent 'processing' lane, like the 4 regular processing lanes and the Zero-Block independent processing lane.

A header generation module calculates the expected File Format header by reading the configuration values. Most of this information is included into the header, allowing to fully decompress the compressed bitstream without any additional side-communication channel, as it has already been explained in section 2.2.4.

The header bitstream is processed the same way as any other Reconstructed CDS. This mechanism makes possible to easily integrate different kinds of headers (IE the CCSDS

Field	Description
laneID	Lane Identifier (0-3 Regular, 4 Zero-Block, 5 Header)
opID	Unique Operation Identifier
FS-Type	Type of Fundamental Sequence (0 Regular, 1 Second Extension)
ref	Flag to indicate if a reference sample is included in the block
Ks_size	Size of K-splits (0 No K-Split, 1-13 K-split Size, D No compression)
total_size	Total size of the final CDS

Table 3.1: Operation Struct Fields

123.0 preprocessing header) by just modifying the header generation module and properly adjusting the header size.

3.5.7 CDS Retirement

The CDS Retirement is the last phase of the processing of the J-Sample Blocks. It consists in the building of the final coded bitstream, which is the result of the cooperation of two well differentiated parts of the architecture, the FSM4 (3.5.7.1) and the CDSReorder and Retirement Units (3.5.7.2).

3.5.7.1 FSM4

All the control, handshakes and retirement initialization is performed in a single, centralized finite state machine, known as the FSM4. This FSM holds 3 key functionalities:

- F1. Reading the operation FIFO
- F2. Internal dispatch of read operations to the active operation registers
- F3. Retirement of the already reconstructed CDS or header.

The first functionality, F1, is fairly simple; if the Operation FIFO is not empty, it reads its first operation. The Operation type is a custom struct type that holds all the necessary information for the processing of each block of samples (Each block has its own operation). The included fields and a brief description of them can be found at Table 3.1

The only fields that are used in all cases are both the 'laneID', which helps to indicate which lane is processing the corresponding block, and the 'opID', a unique autoincremental

identifier that helps to retire the operations in the correct order. Regular Operations (All coding options apart from Zero-Block) use all the struct fields, while the Zero-Block Operation use 'ref' field to indicate if a reference sample must be inserted and the 'FS-Type', used to indicate if the End Of Segment (EOS) has been reached with the operation. The header operation only needs to use the 'opID' and 'laneID' fields.

Every processing lane has its own active operation register. Once the FSM4 reads a new operation from the Operation FIFO, it checks if the lane that is processing that operation is already building an intermediate CDS. If it is the case, it waits for it to end, finally loading it into the active operation register, and raising the 'new operation' flag of that lane to point to the corresponding FSM3, which must start building a new intermediate CDS. This is the second functionality of the FSM, F2. Once the FSM3 ends building the CDS, it raises its done flag.

The last functionality of the FSM4, F3, is to select from the already built intermediate registers the next one that shall be retired, that is, written into the output buffer. To know which operation has to be retired next, an additional state signal known as the 'pending_CDS' is used. The 'opID' identifier that was explained before is crucial in this sense, as these are assigned in the same order that the blocks are received, thus assuring that the CDSes are retired in the expected order, without taking into account the parallelized processing of the blocks. The operation selected must have its corresponding 'done' and 'pending_CDS' flags raised, and its ID must be the lesser of all the active operation IDs that have not already been retired. Once selected, the CDS is sent to the CDS Reorder and Retirement Units, which are notified by activating its 'start' flag. The units effectively build the final bitstream, whose chunks are progressively sent through the output interface.

A general representation of how this phase interacts with the previous one, governed by the nFSM3s, and the following one, directed by the reorder and retire units, can be found in Figure 3.16.

3.5.7.2 Additional pipelining

Originally the selected CDS was directly forwarded to the submodule known as the CDS Retirement Unit. The CDS Retirement Unit holds the output buffer, and includes two independent FSMs that write and read it in a completely independent manner. The included output buffer is implemented as a circular buffer, in which the first-to-be-transmitted bytes are stored in the lesser, rightmost positions.

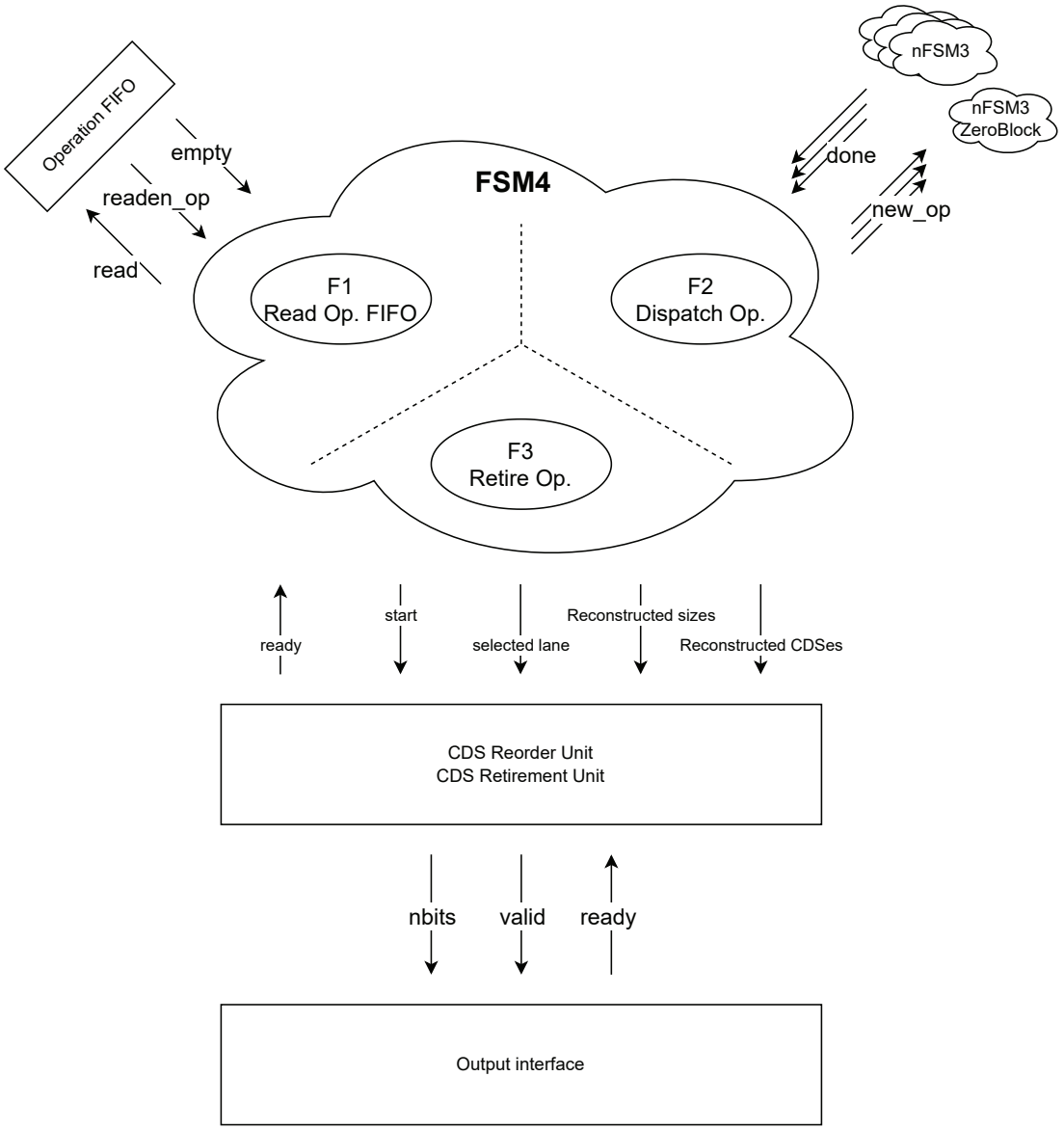

FIGURE 3.16: General representation of FSM4 functionality

Internally, the components work by adding the information in the leftmost bit positions, so a byte grouping and reordering was performed prior to the write into the output buffer. This required a finer control due to the fact that the sizes of the CDS can be not byte-aligned, which leads to the inclusion of a smaller auxiliar byte buffer. In addition to these, the intermediate CDSes might be larger than the output interface data width, so a splitting mechanism had also to be included. A complex packing logic that checked for many corner situations was introduced in addition to this splitting mechanism.

As it will be further explained in Chapter 4, all this processing was initially done in the same clock cycle in which the buffer was to be written. A critical path appeared at

this point, which severely limited the final operational frequency. To solve this issue, a pipelining strategy has been applied to the design by creating the CDS Reorder Unit, an intermediate component which is responsible of applying this intermediate modifications in the concatenated CDSes. This additional component helps to effectively remove the critical path, by introducing an intermediate buffer known as the reorder buffer. An overview comparison between the original and the pipelined datapath can be seen at Figure 3.17. An extensive explanation of each of these modules will be given in the following subsections.

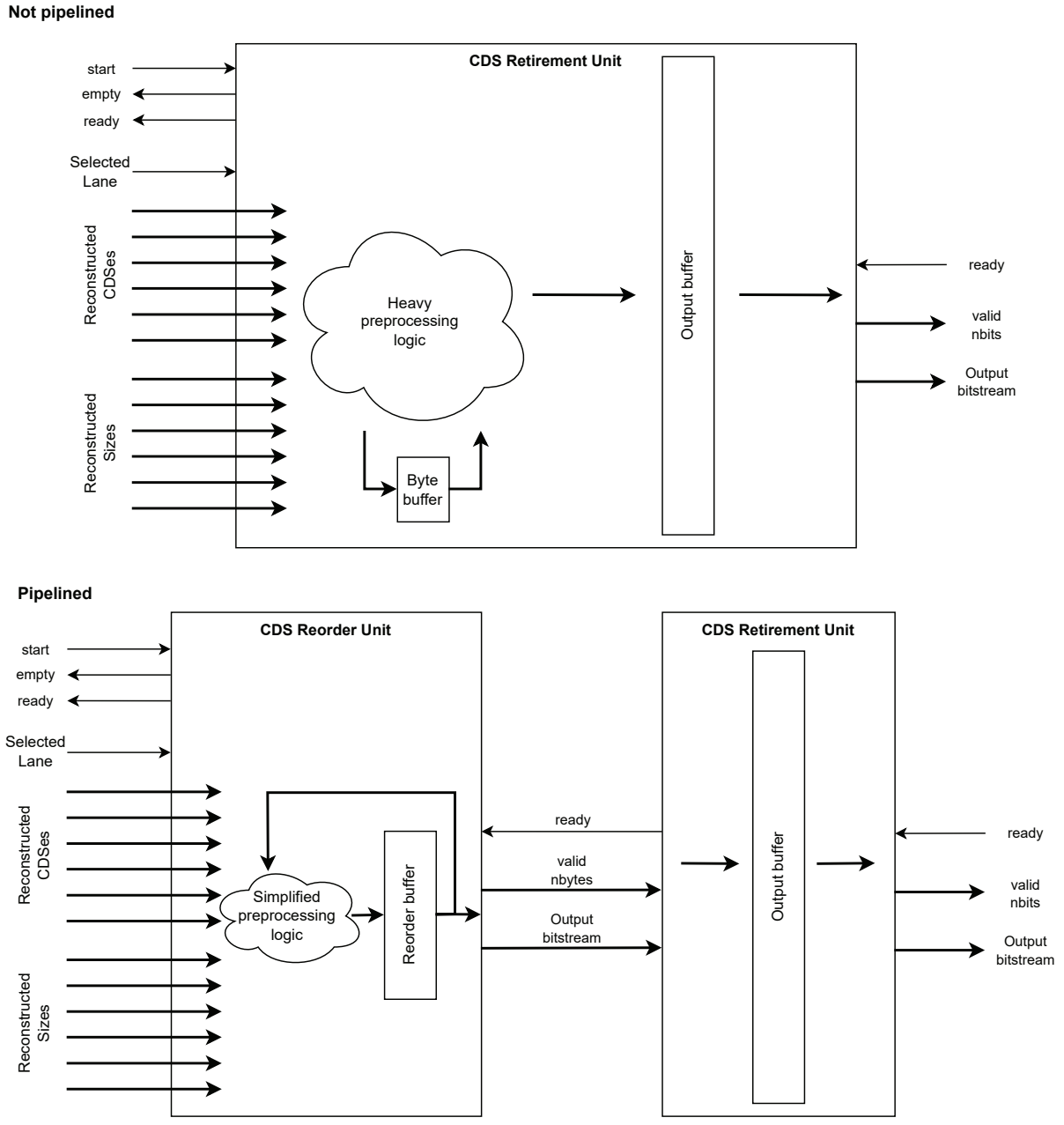

FIGURE 3.17: Architectural comparison of the original and pipelined CDS retirement datapath

3.5.7.3 CDS Reorder Unit

This CDS Reorder Unit receives the CDSes selected by the FSM4, and stores them in some internal registers known as the 'reorder buffer'. Whole bytes are extracted from this buffer and sent to the CDS Retirement Unit to be written into the output register.

The architecture of the 'CDS Reorder Unit' can be found at Figure 3.18. The key part of this module is the 'reorder buffer', a set of internal registers which are read and written simultaneously. When the buffer has enough space for storing at least one maximum-sized CDS, then the ready flag is raised to indicate the FSM4 that new information can be received. The FSM4 signals the Reorder Unit to write a new CDS into the reorder buffer by indicating which lane shall be read and raising the start flag.

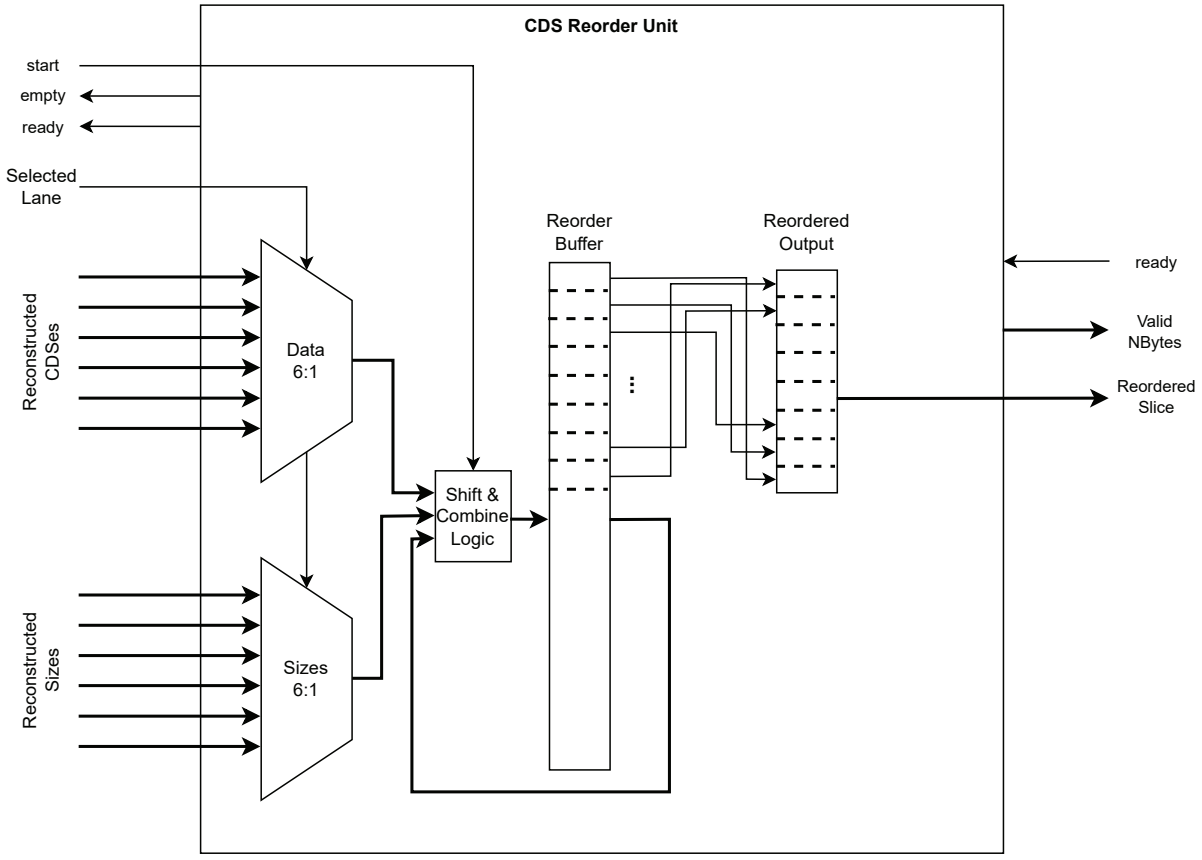

FIGURE 3.18: Architectural overview of the CDS Reorder Unit Component

The reorder buffer is read in slices of a compile-time configurable width, determined by the $SLICE\_SIZE$ parameter. Up to $SLICE\_SIZE/8$ whole bytes are extracted from the reorder buffer each clock cycle. These are swapped and loaded into the 'Reordered Output' register, and the corresponding Valid and NBytes signals are generated to indicate the Retirement Unit how many bytes are valid and therefore have to be written into the

output buffer. The reorder buffer will be read whenever the Retirement Unit is ready and there is at least one whole byte that has not been read yet. A special situation takes place at the end of the processing, as the complete coded bitstream may not be byte-aligned. In this situation, padding in the form of logical 0s are appended to the valid bits and the byte is finally sent to the Retirement Unit.

When bytes are read from the reorder buffer, these are effectively removed by left shifting its content. If a new CDS is to be written, its corresponding bits are appended to the valid ones. This write logic allows to greatly simplify the read of valid bytes. The reorder buffer is filled with valid bits from left to right. This means that the first bits to be produced are always written in the leftmost positions, so these can be easily byte-grouped and placed directly into the rightmost positions of the reordered output register. This byte-swapping is performed because of the fact that the encoder components treat the bitstreams as big endian, but the developed interface exclusively works in a LITTLE endian manner.

3.5.7.4 CDS Retirement Unit

The CDS Retirement Unit is responsible for managing the output buffer. Two independent processes write and read through this buffer, thanks to the usage of its own write and read pointers. Both processes traverse through the buffer in a circular way.

The write of the buffer is quite straightforward. The whole bytes that are received from the Reorder Unit are directly written into the output buffer. A finer control to avoid overflowing the buffer is implemented.

The reading of the buffer is done by extracting up to $W\_BUFFER/8$ bytes in each clock cycle. W_BUFFER is a compile-time configuration parameter that defines the width of the Advanced eXtensible Interface (AXI)-Stream output interface. These bytes are sent to the output interface, which implements a basic ready/valid handshake to receive new data, along with an additional nbits signal that helps to indicate which of the information is valid, easing the generation of the corresponding strobes.

3.5.7.5 Considerations on the output buffer size and the slice size

The size of the reorder buffer is chosen to allow simultaneous and continuous reads and writes through this buffer. Note that if the chosen SLICE_SIZE parameter is smaller than

the maximum CDS cases, the buffer may be completely filled under some scenarios, which could be a problem as the processing will eventually have to stop the pipeline, effectively reducing the final performance of the compressor.

Anyway, these are not common scenarios. In most of the testcases that have been simulated and successfully verified, which will be further explained in Chapter 4, the selected $SLICE\_SIZE$ was equal to the output buffer width, 128 bits, which is slightly less than the 132 maximum CDS size (No compression, D = 16). Under this configuration the pipeline was rarely stopped, as resulting CDS sizes tend to be less than this maximum size. For testing purposes, some simulations in which $SLICE\_SIZE$ parameter was reduced to 64 bits were performed. In these cases, pipeline was more frequently stopped.

3.6 Data interfaces

Two completely new data interfaces have been designed. Both the input interface and the output interface are AXI4-Stream compliant. While it is true that these two interface meet the specifications of the project, the architecture has been designed to be easily adapted to any other kind of interface. A brief explanation of the AXI4-Stream bus will be offered in the next subsection (3.6.1). Following this, a detailed overview of both the input (3.6.2) and output (3.6.3) interfaces will be given. Lastly, the modularity that these modules and its internal interfaces offer will be discussed (3.6.4).

3.6.1 AXI4 Stream

The AXI 4 Stream protocol is specifically designed to simplify the transportation of unrestricted unidirectional data. Within an AXI4-Stream bus, a specified number of bits, determined by the TDATA bus signal width, are transferred per clock cycle. The transfer process starts when the producer initiates the transmission by rising the TVALID signal. Whenever the transmitter rises the TVALID signal, it is pointing the receiver that the data in the TDATA bus is valid. The consumer, upon detecting the raised TVALID signal, can answer by rising the TREADY signal, indicating it has consumed the data sent by the transmitter. Once this handshake takes place, the producer can proceed to send new data. It is important to remark that there are no required dependency order between these signals: A receiver can rise first TREADY to indicate transmitter that it is prepared

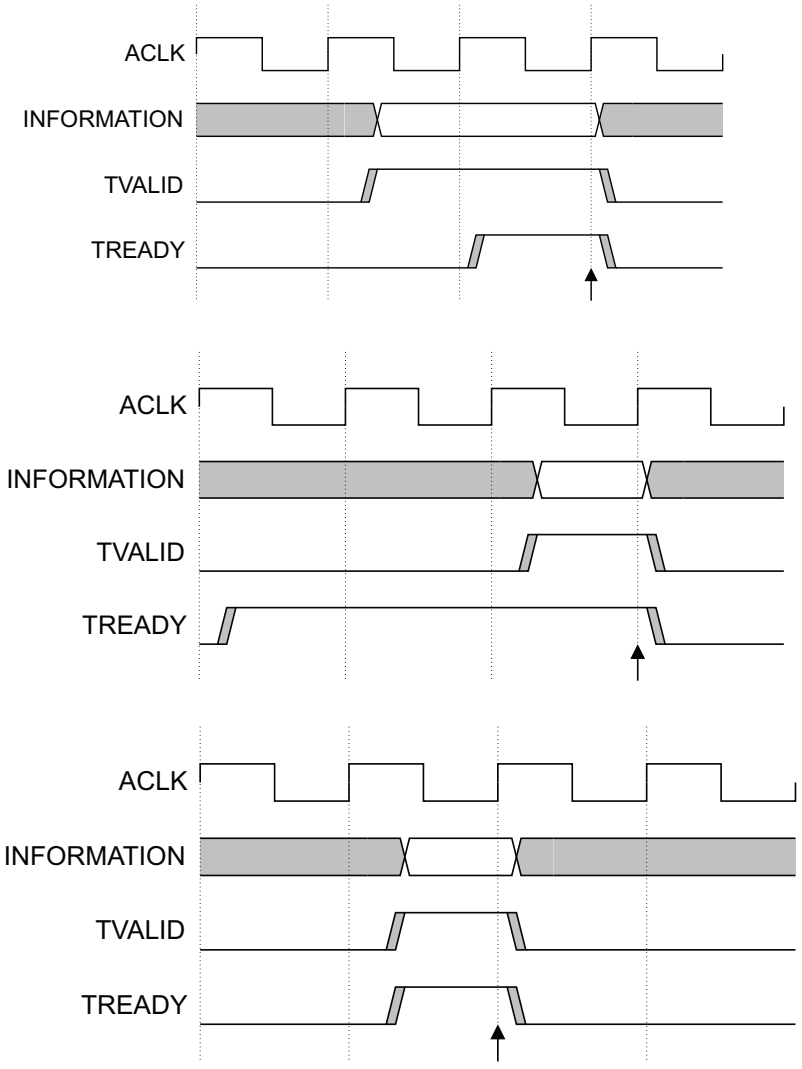

FIGURE 3.19: AXI4 Stream handshake possibilities [17]

to receive new data, while it is also valid to rise the TVALID first to point out new data is present in the bus, and these can even be raised simultaneously, in which case the handshake and data transmission takes place in a single clock cycle (see 3.19).

These 3 signals (TDATA, TVALID, TREADY) are mandatory for any AXI4-Stream bus, but many other signals are defined as optional and may help to offer greater control in some scenarios. TLAST is an optional signal that, once raised, indicates that the data in the bus is the last part of a packet, the last valid data from a consecutive stream (i.e. the last part from a compressed bitstream that has been divided in many individual transmissions). TSTRB optional signal helps transmitter to indicate which bytes in the TDATA signal are valid and which are not (it can be seen as a byte-level masking). TKEEP is also a byte-level qualifier that indicates if each of the bytes are part of the actual stream. Other typical

signal is TUSER, which can help as a side-channel to transmit additional information in parallel to the main TDATA signal (i.e. endianness of the data in the TDATA signal).

In addition to these data and control signals, AXI4-Stream also includes a global clock signal, ACLK, and a low-active asynchronous reset signal, ARESETn. This means that each of the buses have their own clock domain, that may differ from the internal IP Core clock domain.

3.6.2 Input Interface

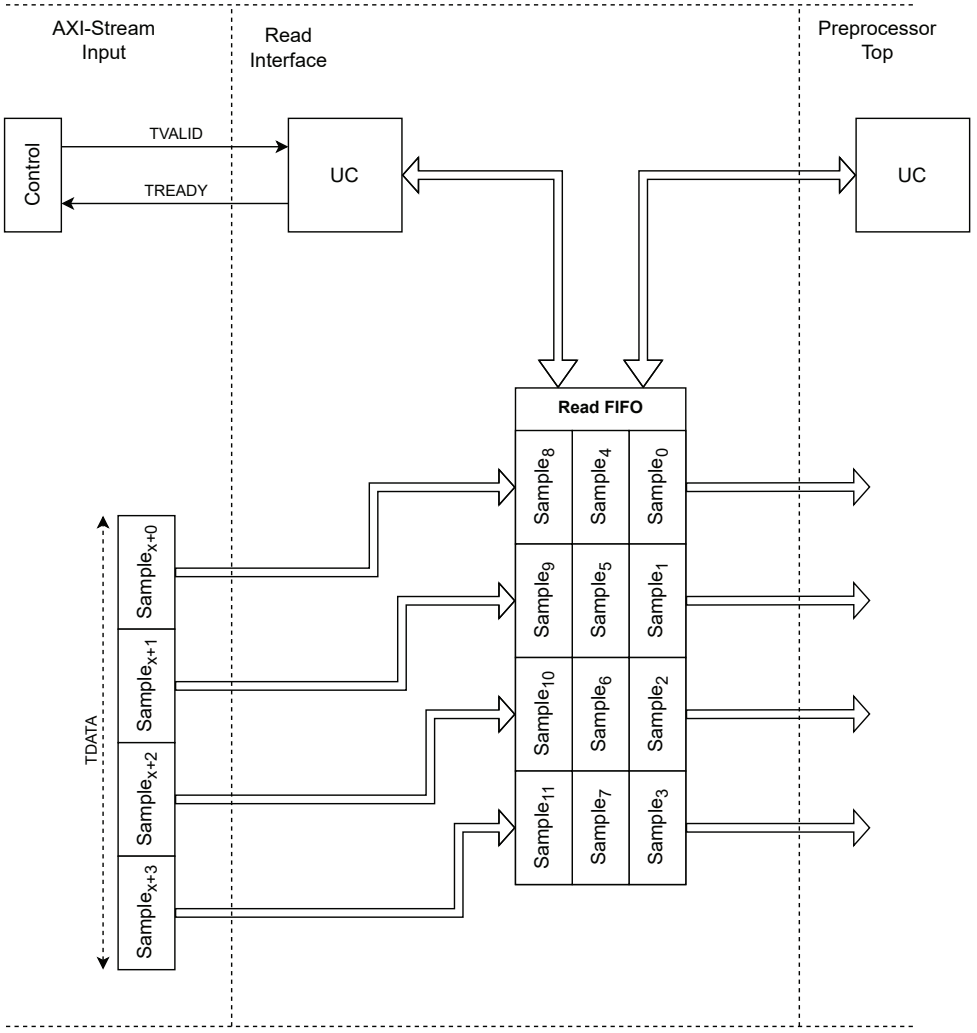

FIGURE 3.20: AXI4-Stream Input Interface Diagram

The designed input interface implements only the essential AXI4 Stream signals (TDATA, TVALID, TREADY). The architecture is designed to receive 4 samples each clock cycle (half a block), so the width of the TDATA signal is fixed to $4 \times D$ (four times the dynamic

range of the input samples, which by default is 16). Every time the valid signal is raised, 4 samples will be read from the bus (padding will be added automatically if this is requirement is not met) so no additional control signals are required.

The interface internally buffers the received data in a FIFO buffer. The predictor directly interacts with this FIFO, reading it on demand to predict groups of 4 samples as it has already been explained in section 3.3.

A diagram of the input interface can be seen in Figure 3.20.

3.6.3 Output Interface

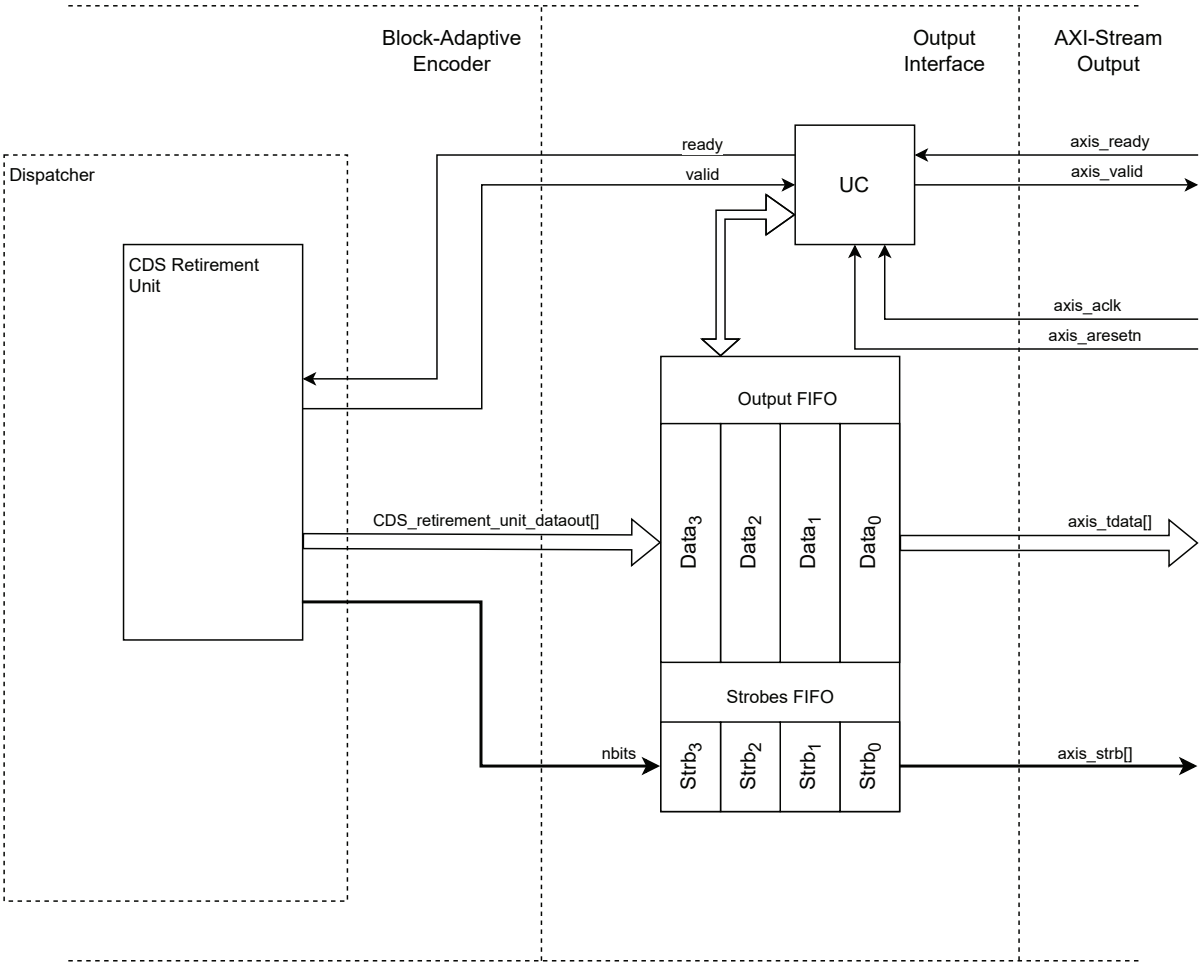

Figure 3.21: AXI4-Stream Output Interface Diagram

The output interface interacts with the block coder of the architecture. More specifically, a ready-valid handshake mechanism (similar to the specified in the AXI4-Stream [17] protocol) is implemented between the CDS Retirement Unit, which is responsible for

writing and reading the output buffer, and the output interface. Two data signals are defined bewteen these modules: A data signal, which holds the data to be transmitted, and a nbits signal, which helps indicating how many bits from the data signal are valid.

The control of this output interface is more complex than the one in the input interface, as the amount of data that can be received each clock cycle is not constant. Both the data and nbits values are buffered in internal FIFOs and transmitted to the outside. In this regard, the output interface implements the optional signals TSTRB and TKEEP to help indicating which bytes are holding valid values. TDATA is directly assigned from the data received from the encoder, while nbits is converted to TSTRB.

The data signal widths are determined by the user configurable parameter W_BUFFER, which is set by default at 128 bits as explained in the specifications of the design. TSTRB width will then be $W\_BUFFER \div 8$, meeting the AXI4-Stream protocol specification, and it is generated from the nbits buffered signals.

A diagram of the output interface can be seen in Figure 3.21.

3.6.4 Modularity in the design interfaces

Both the predictor and coder data interfaces have been designed to be easily adapted to any other protocol (IE an AXI4 Memory Mapped bus).

The input interface must offer a FIFO-like interface, while the output interface must implement the basic two-way ready-valid mechanism and a few status signals (such as empty and full). Both of these interfaces are common control schemes, so designing new interfaces and connecting these to the compressor should not be a problem.

Both interfaces implement small buffers. These FIFOs are important not only to buffer a small amount of samples, but also to synchronize clock domains as shown in Figure 3.22. The included FIFOs use independent clock and reset signals for its write and read ports, easing the synchronization between the AXI domains and the IP Core domain. This implicit Clock Domain Crossing (CDC) mechanism acts as a compatibility layer for many different types of buses, making the design considerably more versatile, thus facilitating its implementation in a larger number of cases.

Note that both data interfaces are completely optional. The design can be integrated into another top design by directly connecting it. Regarding the data input, just a FIFO

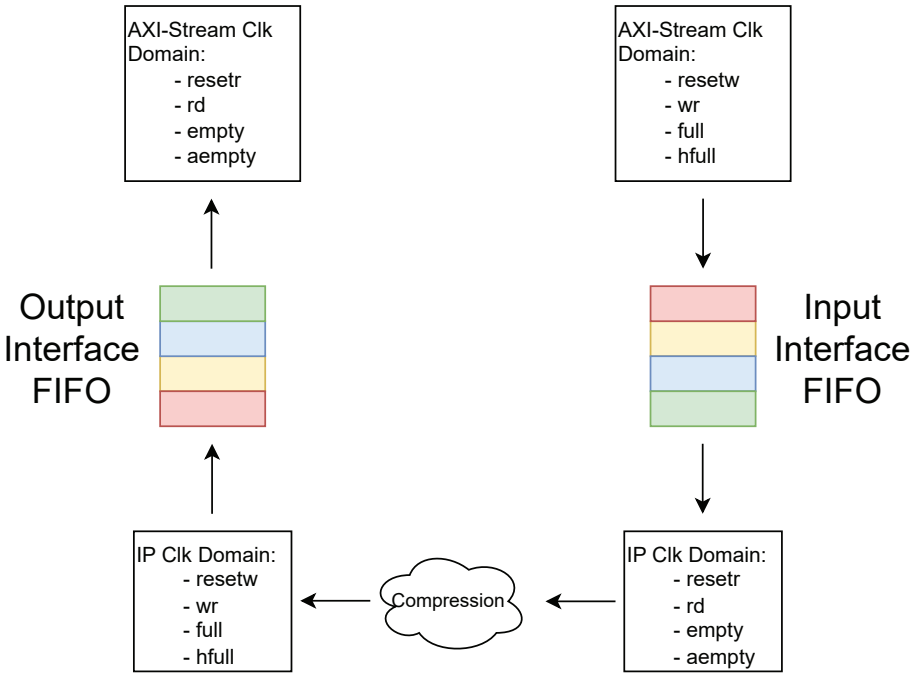

FIGURE 3.22: Data flow between the different clock domains

is needed, in which new data are loaded (Predictor takes care of reading it as needed) while for the output of data, implementing a simple handshake mechanism as explained before should be enough to receive the fragments of the final coded bitstream. By directly implementing this raw handshakes, the design can be easily integrated into more complex, custom IP Cores, rather than using it like a standalone accelerator.

3.7 Conclusion

A detailed explanation of each part of the design has been given in this chapter. A highly parallelized architecture has been designed to achieve significant processing throughput. The SHyLoC CCSDS 121.0-B-3 compressor has been used as the base architecture. Some of its original parts have been deeply modified to include 4 processing lanes, while others have been replaced by completely new modules implementing new required functionalities.

Two AXI4-Stream compliant interfaces have been built from scratch for reading and writing data. The input interface implements the basic set of signals to allow efficient and simple feeding of groups of 4 raw samples to the compressor IP. The output interface has been designed more carefully, as it is prepared to output a dynamic number of bytes per

clock cycle, requiring the implementation of additional AXI4-Stream signals such as the TSTRB.

The original unit-delay predictor has been modified to effectively predict 4 samples per clock cycle. The control scheme has been slightly adapted, but the major changes have been made in the data path, as the components have been replicated internally.

An entirely new interconnect component, the Post-Predictor Block Dispatcher, has been designed to provide a seamless interconnection between the predictor and the encoder. This component is essential in the architecture because each of the lanes contained in the encoder processes whole blocks of samples in a sequential manner, while the predictor produces blocks of predicted residuals every 2 clock cycles (half a block per cycle). The interconnect effectively accumulates the residuals and sends the blocks to the appropriate lanes.

The encoder is the part of the compressor that has undergone the most significant changes. The original SHyLoC data path has been replicated and deeply modified to implement a completely new operation-based control scheme. This scheme allows the different processing lanes of the encoder to be coordinated in a centralized manner, controlling both the reconstruction of the intermediate CDS and its incorporation into the final output bitstream. At the same time, the processing of zero blocks is implicitly included in the control, drastically reducing the dependencies that would otherwise occur between the processing lanes.

An additional Block Unbundler module has been designed to individually extract the samples of the blocks received from the Post-Predictor Dispatcher, since each of the processing lanes operates independently in a fully serial fashion. Both the CDS Length and Coding Option Calculation and the CDS Codification have been slightly modified and replicated to fit the new control scheme, but the real changes have been the inclusion of two new stages: The Intermediate CDS Reconstruction and the CDS Retirement. The first of them consists of rebuilding each of the CDSs into intermediate registers. The CDS Retirement is a complex phase that takes care not only of the internal dispatching of operations to the various processing lanes involved, but also of coordinating the writing of the fully reconstructed CDSes into the final bitstream. In this sense, both the CDS Reorder and Retirement units are critical components in preparing the output bitstream, whose chunks are progressively processed and outputted.

Chapter 4

Design Verification and Synthesis

4.1 Outline

This chapter covers both the verification phase, which was used to thoroughly demonstrate that the design works as intended, and the synthesis of the design. First, a list of the developed VHDL source files is offered in Section 4.2. The available configuration parameters, both the compile-time configurable and the run-time configurable, are listed in Section 4.3. Next, the verification phase explained in detail in Section 4.4, by analyzing the 2 verification campaigns that were performed: An initial block-level campaign that focused on specifically verifying some key components of the architecture and a longer, system-wide campaign that allowed to demonstrate that the compressor system generated the correct compressed bitstreams by running a set of different test cases. Finally, the synthesis results are analyzed in Section 4.5, and a brief analysis of these results is offered in Subsection 4.6.

4.2 VHDL Description

Once designed, the description of the whole system in VHDL was performed. A set of VHDL sources were generated to modularly describe the different parts of the compressor architecture. A list which contains all the resulting files, along with a minimal description of each of these can be found at Table 4.1.

Source file	**Description**
parallel121_constants.vhd	Constants of the architecture
parallel121_config_package.vhd	Axiliar package with configuration-related functionalities
parallel121_ahbs.vhd	AHB Configuration Bus Module
parallel121_shyloc_interface.vhd	Configuration interface module
parallel121_CDS_retirement_unit.vhd	CDS Retirement Unit module
parallel121_read_interface.vhd	AXI-Stream Input Interface
parallel121_predictor_comp.vhd	Components of the parallelized predictor
parallel121_predictor_fsm.vhd	State machines of the parallelized predictor
parallel121_predictor_top.vhd	Top wrapper of the parallelized predictor
parallel121_predictor_dispatcher.vhd	Post-Predictor Dispatcher, interconnection between parallelized predictor and encoder
parallel121_dualread_fifo.vhd	Special FIFO that allows to read up to 2 elements per clock cycle
optcoder.vhd	Selects the final winning coding option
lkcomp.vhd	Individually obtains the length of applying a specific coding option
parallel121_header.vhd	Module responsible for generating the expected header codeword
sndextension.vhd	Calculates the gamma values and obtains the length corresponding to applying SecondExtension option
fscoderv2.vhd	Component that calculates the FS of each block by applying the previously selected coding option
lkoptions.vhd	Obtains the winning obtion between all the FS and K-Split options
parallel121_clk_adapt.vhd	Module to manage Clock Domain Crossing
parallel121_CDS_builder.vhd	Module responsible for reconstructing intermediate regular CDSes
parallel121_ZeroBlock_builder.vhd	Module responsible for reconstructing intermediate ZeroBlock CDSes
parallel121_CDS_reorder_unit.vhd	CDS Reorder Unit Module
parallel121_operation_generator.vhd	Module responsible for generating the corresponding Operation given a block and additional information
parallel121_encoder_comp.vhd	Module that holds the processing components of the parallelized encoder
parallel121_encoder_fsm.vhd	Module that holds the control state machines of the parallelized encoder
parallel121_encoder_unbundler.vhd	Module responsible for extracting individual mapped residuals from whole blocks
parallel121_blockcoder_top.vhd	Top wrapper of the parallelized encoder
parallel121_output_interface.vhd	AXI-Stream Output Interface
parallel121_shyloc_top.vhd	Top wrapper of the parallelized compressor

Table 4.1: List of VHDL Sources

4.3 Configuration Parameters

The designed and described architecture defines a set of both compile-time and run-time configuration parameters. The compile-time parameters are statically defined and are defined in the Table 4.2. These parameters directly impact the behaviour of the components of the design and need to be carefully picked, as a inadequate selection of parameters might cause to experience additional delays during the compression process.

Parameter	**Description**	**Type, {range}, {Comment}**
EN_RUNCFG	Runtime configuration is enabled	Boolean
RESET_TYPE	Type of reset (0: Asynchronous, 1: Synchronous)	Boolean
J_GEN	Block size	Natural, 8
CODESET_GEN	Reduced codeset support	Boolean
REF_SAMPLE_GEN	Reference sample insertion interval in blocks	Natural, [64, REF_SAMPLE_GEN], multiplo of 64
W_BUFFER_GEN	Output buffer width	Natural, [8, 1024], multiplo of 8 (64, 128 tested)
DISABLE_HEADER_GEN	Disables the insertion of headers	Boolean
EDAC	Usage of EDAC FIFOs	Boolean
Ny_GEN	Y-dimension size of the input data	Natural, [0, $2^{48}-1$]
Nx_GEN	X-dimension size of the input data	Natural, [0, $2^{48}-1$]
Nz_GEN	Z-dimension size of the input data	Natural, [0, $2^{48}-1$]
D_GEN	Size of each of the input samples	Natural, [2,32]
ENDIANESS_GEN	Endianness of the input samples	String, ("le"|"be")
TECH	Technology used for internal FIFOs	Natural, [0,4]
HEADER_FORMAT	Type of bitstream packing (0: Packet Format, 1: File Format)	Boolean, 1, Packet not supported
SPLITTER_SLICE_SIZE	Maximum size of the internal slicing of the CDS	Natural, [32, OUTPUT_BUFFER_SIZE]
OUTPUT_BUFFER_SIZE	Size of the output buffer	Natural, [W_BUFFER_GEN, 1024]

Table 4.2: Compile time parameters

Regarding the run-time configuration parameters, these have been significantly restricted when compared to the original SHyLoC IP Core. The available configuration parameters can be found in the Table 4.3.

Parameter	Description	Type, {range}, {Comment}
N_y	Runtime Y-dimension size of the input data	Natural, [0, Ny_GEN]
N_x	Runtime X-dimension size of the input data	Natural, [0, Nx_GEN]
N_z	Runtime Z-dimension size of the input data	Natural, [0, Nz_GEN]
REF_SAMPLE	Runtime reference sample insertion interval in blocks	Natural, [64, REF_SAMPLE_GEN], multiple of 64

Table 4.3: Runtime parameters

4.4 Verification of the design

The verification methodology that has been followed to verify the proposed design is explained in this chapter. The verification of the design has been performed in two well differentiated stages. The first stage consisted in the development of a couple of block-level testbenches in which specifics components were verified, as explained in Section 4.4.1.

The second and last of the phases was a more extensive, system-wide verification process, which will be deeply explained in Section 4.4.2. It consisted in comparing the resulting compression bitstream with a golden reference that was generated by using an internally developed CCSDS 121.0-B-3 Compression Software.

As a result of the verification process, many architectural bugs were detected and successfully solved. This implied modifying some specific parts of the design and introducing some additional functionalities to match the expected behaviour.

4.4.1 Block-Level Verification

As explained in the outline, the initial verification stage was performed toward individual blocks. To be precise, block-level testbenches for both the predictor and the post predictor dispatcher were designed and implemented, allowing to demonstrate the correctness of these modules.

For each of these components, a specific testbench with custom drivers and monitors have been designed, as well as custom scoreboards that will check if the results are the expected ones. In the following subsections, the two block-level testbenches that have been

developed will be explained, both the predictor testbench (4.4.1.1) and the post-predictor dispatcher testbench (4.4.1.2).

4.4.1.1 Predictor Testbench

A custom testbench has been designed to verify the parallelized predictor of the new design. To check the correctness of the outputted mapped prediction residuals, both the original SHyLoC 121.0 Unit-Delay Predictor and the parallelized 121.0 Unit-Delay Predictor are instantiated within the testbench and fed with the same input data. The parallelized design is correct if the individual prediction residuals are identical to the ones provided by the original predictor.

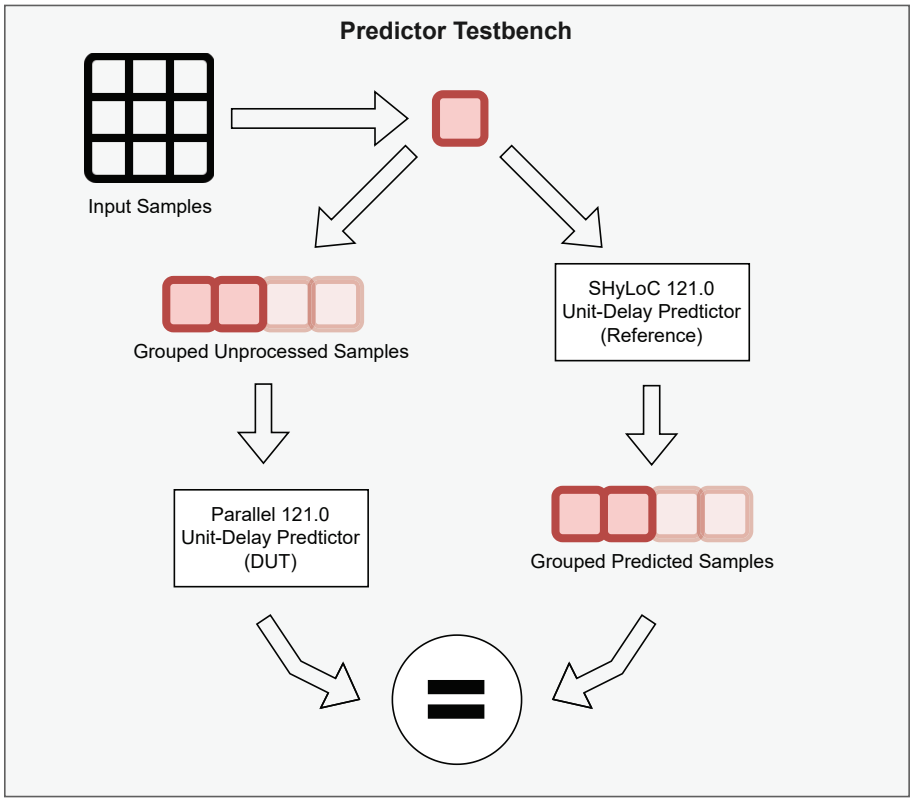

FIGURE 4.1: Overview of the Parallel121 Unit-Delay Predictor Testbench

A set of parameters were defined to determine how the test will be performed, these can be found at Table 4.4. A general overview diagram of the testbench architecture can be found at Figure 4.1. A pair of custom driver and monitors for each of the predictors were created.

For the generation of the input dataset, two options are offered: Use a file as input or randomly generate a set of data.

Parameter	Description
$SIZE_{NX}$, $SIZE_{NY}$, $SIZE_{NZ}$	3 dimensional sizes of the input data.
ITERATIONS	Number of iterations of the test
VERBOSE_LVL	Verbosity level
$SEED_1$, $SEED_2$	Pair of seeds used to initialize the random number generator.

Table 4.4: Unit-Delay Predictor Testbench Parameters

In the first case, the path of a file of that contains at least $SIZE_{NX} \times SIZE_{NY} \times SIZE_{NZ}$ elements of size D_GEN must be specified. Every element is read and included into the dataset, an array that holds $SIZE_{NX} \times SIZE_{NY} \times SIZE_{NZ}$ std_logic_vectors of width D_GEN, which is a compile time defined architectural parameter that indicates the dynamic range of each of the samples.

To generate random input datasets, the random number generator function $uniform$ from the standard VHDL package ieee.math_real is used. These datasets consist of an array type that holds $SIZE_{NX} \times SIZE_{NY} \times SIZE_{NZ}$ std_logic_vectors of width D_GEN.

Both drivers feed the predictors with the elements from the dataset that have previously been read or randomly generated, as already explained. The driver for the original predictor just performs the necessary handshake to drive each of the elements. The parallelized design driver also performs this handshake but, prior to this, it groups $J \div 2$ elements. This is necessary as the parallelized predictor needs to be fed with that exact amount of samples, as explained in subsection 3.6.2.

When it comes to the monitors the opposite situation takes place. The monitor of the parallelized predictor directly takes the groups of $J \div 2$ mapped residuals, while the other monitor, the one which interacts with the original predictor, reads individual mapped residuals and arranges them in groups of $J \div 2$ residuals.

The scoreboard works by bit-wise comparing the groups of mapped residuals. Scoreboard communicates with the driver/monitor processes with a simple start/done signaling scheme: Scoreboard raises start to indicate the driver that it can proceed to feed new samples to its corresponding predictor while monitor raises done to point the scoreboard out that new residuals are ready to be compared. If an error is found, the error counter is augmented. At the end of each iteration, a report that lists the number of tested samples and the number of errors found is shown.

Three levels of verbosity are defined. If VERBOSE_LVL is set to 0, just the final report will be printed. If VERBOSE_LVL is set to 1, then additional information of each of the found errors will be printed (This is the default verbosity level). If VERBOSE_LVL is set to 2 or higher, the additional information about each of the performed action in the test will be printed.

The predictor-specific testbench showed up as a useful and fast way to verify this component. The existence of a reference module, the original SHyLoC Unit-Delay Predictor, eased the design of the testbench. Some minor errors were detected and solved thanks to the testbench, and the whole verification process took less than 1 week.

4.4.1.2 Post-Predictor Dispatcher Testbench

Once the predictor was fully verified, the verification of the next critical component, the post-predictor dispatcher, began. As with the predictor, a custom testbench that enabled a block-level verification of this dispatcher was designed and implemented. The main difference that can be found is that in this case there is no reference software or design, so a direct comparison cannot be applied to check the correctness of the design. Due to this difference, and as it can be seen in Figure 4.2, no reference module is directly instantiated. The scoreboard incorporates a minor logic that internally models the expected output, and uses it to check if the received matches it.

The testbench configuration parameter generics are identical to those of the predictor testbench, which can be seen at Table 4.4. As for the generation of the dataset, only support for random generated dataset is offered in this case: $SIZE_{NX} \times SIZE_{NY} \times SIZE_{NZ}$ samples are randomly generated by an auxiliary function.

To drive the groups of $J \div 2$ samples (mapped residuals) to the dispatcher, a custom driver has been used. This driver arranges samples to builds groups of $J \div 2$ samples, which are stored in a FIFO. The Design Under Test (DUT) interface port is able to interact with FIFO Buffers, so it directly interacts with this FIFO by reading its contents under demand.

The scoreboard takes care of checking if the output is the expected one. The dispatcher must put the received samples in a descending order, and rise the valid flag once a complete set of J Samples is available for a lane. Once the valid signal of a lane is raised, the

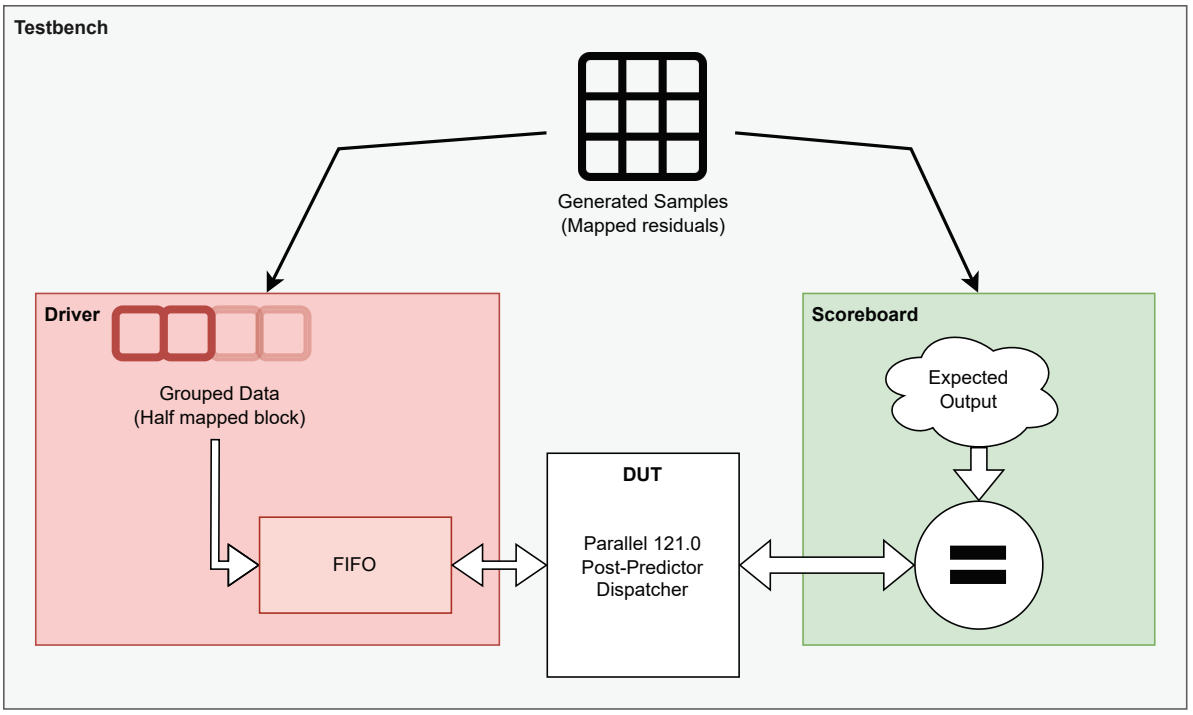

FIGURE 4.2: Block diagram of the Parallel121 Post-Predictor Dispatcher Testbench

scoreboard checks if the samples provided by the dispatcher are the expected ones by direct comparison.

The same 3 verbosity levels of the predictor testbench are implemented: Level 0 just prints a basic report after each test iteration while level 1 also includes additional information of the error founds. Level 2 continuously reports the performed actions by each of the testbench components.

The development of this testbench was significantly faster than the predictor testbench, as the latter was taken as a starting point. This allowed to just modify the drivers and scoreboard to check the dispatcher-specific functionality. Therefore, the whole verification process of the post-predictor block dispatcher module was completed within 3 days. No critical issues arose during the verification of the module.

4.4.1.3 Block-Level Verification Results

The block-level verification of the predictor and post-predictor dispatcher has allowed us to effectively demonstrate the correctness of both modules. This phase of the verification took place right before the design and implementation of the internal architecture of the

Block-Adaptive Encoder. As planned in the original Gantt diagram, block-level verification was to be performed along with the description of the rest of the design.

The explained modules were exhaustively verified, which translated during the last phases of the implementation into a fast and seamlessly integration of these with the rest of the system (Additional control, configuration, encoder). This verification methodology is based in a bottom-up approach that tends to ease the integration of the different components of the system at the expense of larger verification periods. In addition, not all components are suitable for this kind of verification methodology.

This is the case with the different submodules of the Block-Adaptive Entropy Coder. These submodules are designed to work in a completely collaborative, synchronized manner, so verifying the whole functionality of the compressor system rather than individually verifying each of these submodules showed up as the preferred choice.

4.4.2 System-Wide Verification

Once all the different components of the compressor system were integrated, the final, system-wide verification process started. This process took around 1 month to exhaustively verify the functionality of the design. The original testbench of the original SHyLoC has been adapted to fit the new requirements and interfaces of the Parallel121 Compressor.

4.4.2.1 Original SHyLoC Testbench

The original SHyLoC CCSDS 121.0 Compressor testbench is able to execute different user-defined testcases. Some testcases introduce special behaviours, but generally the testbench initially configures the compressor IP (if runtime configuration is enabled) and starts to read a raw input file, which will be compressed by the IP, while storing the outputted bitstream into another file. When the compression ends (Finished flag asserted), the testbench starts to compare the file that contains the compressed bitstream produced by the IP and another file that contains the reference bitstream. This reference bitstream is externally generated by using a reference CCSDS 121.0-B-3 Compression software, which was developed internally at the DSI Research Group.

Every clock cycle a sample from the raw file is driven to the IP, if this is ready to receive new samples, that is, its ready signal is high. Also, if the IP raises the IP valid signal,

which indicates that the data outputted is valid, this data is stored in the compressed bitstream file.

Once the Finished flag is raised by the IP, then the testbench stops driving new data into the IP and starts to compare the golden bitstream file with the one that holds the data extracted from the compressor IP. In addition to this comparison, that will determine if the compressed bitstream is correct and therefore if the system is compliant with the standard, some assertions that help checking the correct signaling of the IP are included.

4.4.2.2 Adaptation of the testbench

A new testbench was created by adapting the original SHyLoC testbench. Some significant changes had to be included in this testbench to address some of the changes introduced.

The main and most relevant change is related to the data interfaces, to both the input and output interfaces. Parallel121 data interfaces implement the Advanced Microcontroller Bus Architecture (AMBA) AXI4-Stream protocol, that has already been introduced in section 3.6.1. This protocol differs from the one offered in the original SHyLoC Compressor, which is based in raising a valid flag at the same time that data is introduced a cycle after the IP asserts its ready signal.

For inputting new samples, groups of 4 samples are driven to the input bus and the valid flag is raised. If the ready signal is also raised in the next rise clock edge, the new samples can be inputted, otherwise the testbench must wait for the ready to be asserted. The input interface does not implement additional signals such as strobes, so this adaptation has been done without any problems.

The output interface can output up to $W\_BUFFER/8$ bytes per clock cycle. This differs from the original interface in that the amount of outputted bytes is not constant, so a finer control in the reading of this bus must be implemented. Also, the bus is little endian, so byte swapping is performed in the testbench to enable a correct comparison with the reference bitstream. Some other minor modifications were performed. These changes are related to minor timing differences between the original and the parallelized design.

4.4.2.3 Verification flow

The complete verification flow is represented in Figure 4.3. When it comes to create a new test, the first decision to be made is the selection of which data will be compressed. This is a crucial choice as different datasets will be used to test different parts of the design (i.e. an almost uniform file will help to test the zero-block and other low-entropic coding options). Once the data to be compressed have been selected, the compression parameters shall be chosen. A list of available parameters as well as a brief explanation of each of these can be found at Tables 4.2 and 4.3. In addition to these, each testcase includes its name and the names of the files that contains both the raw dataset and the reference bitstream.

After defining the test parameters, both the reference software and the auxiliary Python scripts are executed. The first one is used to generate the reference bitstream, that will be used later to check if the compression process was successful. The Python scripts read the parameters from CSV files that contain them, and uses them to generate both the corresponding VHDL parameter packages and the TCL scripts, that are later executed to compile and optimize the design, and to finally simulate the testbench. The EDA tool that is used for compilation, optimization and simulation is Mentor QuestaSIM. If any error or mismatch shows up during the simulation, then the graphical simulator is launched to debug the simulation, which is done by checking the resulting waves, step-by-step execution, etc. Once the resulting bitstream matches the reference one that had been previously generated, the test has been successfully passed, and a new one can be executed by repeating the same steps.

4.4.2.4 Testcases

A set of testcases was designed to comprehensively test most of the possible scenarios that can take place during a data compression. Three groups of testcases can be defined:

- G1. Basic initial testcases. These tests are aimed at basic functionality checks.
- G2. Functionality-Specific testcases. These tests are aimed at testing concrete parts of the design, such as codification options, reconfiguration, etc.
- G3. Large testcases. Tests that work with large datasets to exhaustively verify the DUT in a general manner.

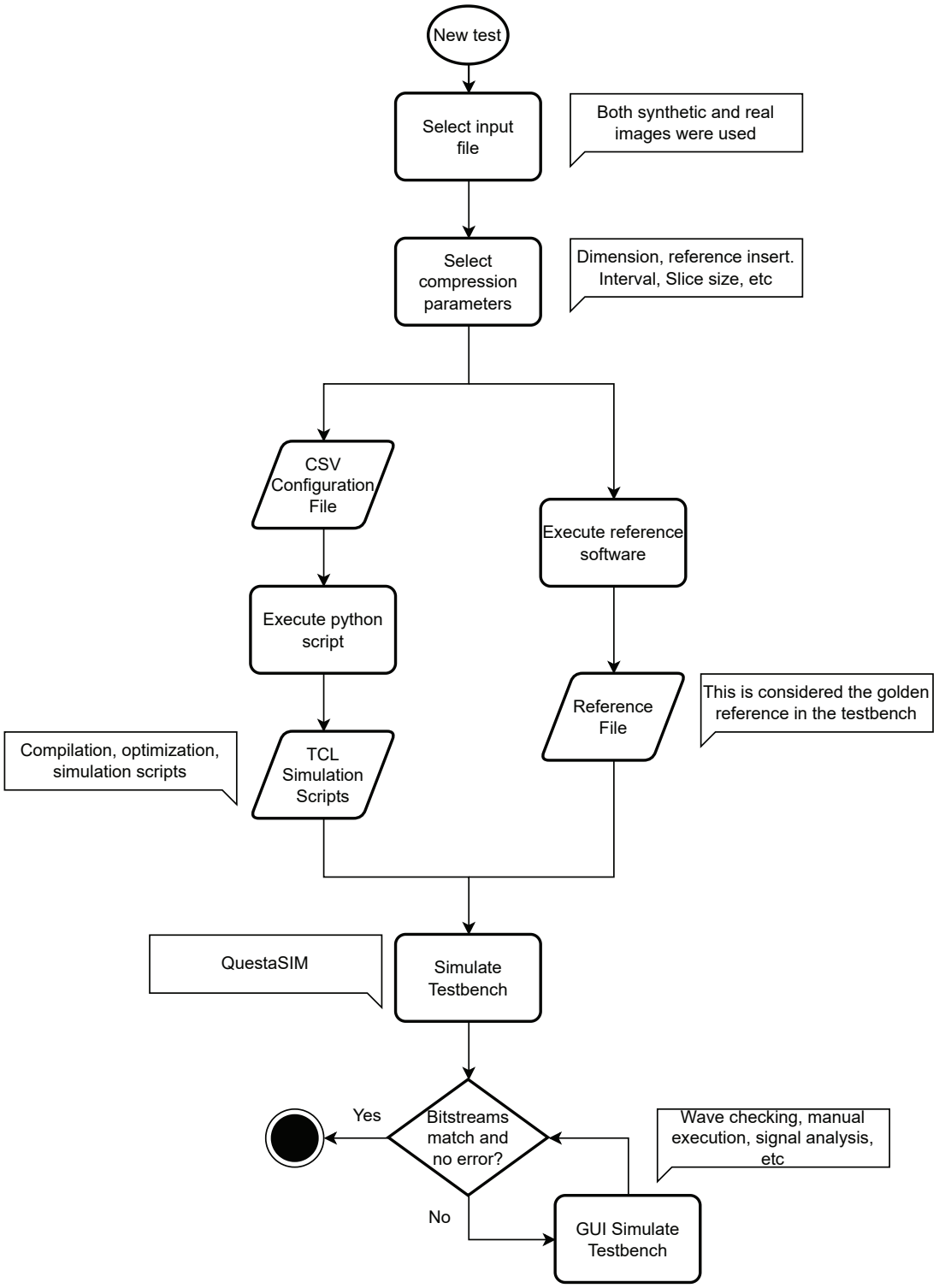

FIGURE 4.3: Flow diagram of the verification

As it can be expected, the G1 testcases were the initial tests that helped to debug the most obvious errors. These were fundamentally small images with a common set of parameters

as it can be shown in Table 4.5. These were fundamental to debug the compressor, helping to detect a wide variety of bugs and misconceptions. The images that these tests process are synthetic images that mainly utilize K-Split compression options.

TestId	**34b_Test**	**34c_Test**	**34c_w128_Test**
Image ID	13_121B2A16	13_121B2A16	13_121B2A16
Input Image	test_p256n16.dat	test_p1024n16.dat	test_p1024n16.dat
Ny	1	1	1
Nx	256	1024	1024
Nz	1	1	1
Set	01c_Set	01c_Set	01c_Set
EN_RUNCFG	0	0	0
RESET_TYPE	1	1	1
J_GEN	8	8	8
CODESET_GEN	0	0	0
REF_SAMPLE_GEN	4096	64	64
W_BUFFER_GEN	64	64	128
DISABLE_HEADER_GEN	0	0	0
EDAC	0	0	0
REF_SAMPLE	4096	64	64
output image	comp_34b.dat	comp_34c.dat	comp_34c.dat
Ny_GEN	1	1	1
Nx_GEN	256	1024	1024
Nz_GEN	1	1	1
D_GEN	16	16	16
ENDIANESS_GEN	le	le	le
TECH	0	0	0
HEADER_FORMAT	1	1	1
SPLITTER_SLICE_SIZE	64	64	128
OUTPUT_BUFFER_SIZE	256	256	512

Table 4.5: G1 Testcases

Once the initial testcases were executed succesfully, some functionality-specific tests were designed to verify more concrete parts of the design. These test can be found at Table 4.6. Tests 34s_mod_Test and 34s_Test were specially designed to process an image in which the second extension coding option had to be applied. In a similar way, 34z_Test was created to test different scenarios in which Zero-Block CDS had to be included into the coded bitstream. For the latter, a special file was manually designed, to test the Zero-Block insertion with different cases (Reference insertion, complete and incomplete Zero-Block Segments with and without EOS). 34u_Test processed an unaligned image (the number of blocks is not aligned with the number of processing lanes), which helped to detect some issues in the pipeline. Test 34b_rt_Test is a variant of 34b from the G1 testcases, that helped to test the runtime configuration of the compressor IP.

TestId	**34s_mod_Test**	**34s_Test**	**34z_Test**
Image ID	13_121B2A16	13_121B2A16	ZB_121B2A16
Input Image	test_ref2ndext _p336n16_mod.dat	test_ref2ndext _p336n16.dat	test_2048 _zb.dat
Ny	1	1	1
Nx	352	336	2048
Nz	1	1	1
Set	01c_Set	01c_Set	01c_Set
EN_RUNCFG	0	0	0
RESET_TYPE	1	1	1
J_GEN	8	8	8
CODESET_GEN	0	0	0
REF_SAMPLE_GEN	64	64	64
W_BUFFER_GEN	64	64	64
DISABLE_HEADER_GEN	0	0	0
EDAC	0	0	0
REF_SAMPLE	64	64	64
output image	comp_34s_mod.dat	comp_34s.dat	comp_ZB.dat
Ny_GEN	1	1	1
Nx_GEN	352	336	2048
Nz_GEN	1	1	1
D_GEN	16	16	16
ENDIANESS_GEN	le	le	le
TECH	0	0	0
HEADER_FORMAT	1	1	1
SPLITTER_SLICE_SIZE	128	128	128
OUTPUT_BUFFER_SIZE	256	256	256

Table 4.6: G2 Testcases (I)

Once the specific functionalities of the design were verified, a single stress test was performed, whose parameters can be found in Table 4.8. This test, named after 34m_Test processes a large image, mapped_06.bin. This is a more realistic test than the rest of testcases, as these compressed datasets of a limited size.

4.4.2.5 System-Wide Verification Results

The system-wide verification has been a crucial step in the development of the final compressor architecture. Many issues, from minor range outboundings to more dangerous architectural problems in the communication of the different modules, were successfully detected and fixed. Less obvious issues were addressed thanks to the special testcases from the group 2. The final test, 34m_Test, helped to demonstrate the correctness of the

TestId	**34u_Test**	**34b_rt_Test**
Image ID	13_121B2A16	13_121B2A16
Input Image	test_p264n16.dat	test_p256n16.dat
Ny	1	1
Nx	264	256
Nz	1	1
Set	01c_Set	01c_Set
EN_RUNCFG	1	1
RESET_TYPE	1	1
J_GEN	8	8
CODESET_GEN	0	0
REF_SAMPLE_GEN	4096	4096
W_BUFFER_GEN	128	64
DISABLE_HEADER_GEN	0	0
EDAC	0	0
REF_SAMPLE	4096	4096
output image	comp_34u.dat	comp_34b.dat
Ny_GEN	1	1
Nx_GEN	512	512
Nz_GEN	1	1
D_GEN	16	16
ENDIANESS_GEN	le	le
TECH	0	0
HEADER_FORMAT	1	1
SPLITTER_SLICE_SIZE	128	128
OUTPUT_BUFFER_SIZE	256	256

Table 4.7: G2 Testcases (II)

compressor, as the resulting bitstream included different kinds of CDS, and most of the situations take place during the compression. All the run tests passed correctly by the end of the verification phase.

Although this verification process has been exhaustive and has helped to prove that the compressor works as expected in most of the situations, it is important to remark that this cannot be considered as a fully verified design: More intensive approaches such as Formal Verification [18] and Random Constrained generated testing, such as the fuzz testing methodology [19], should be applied prior to implementing the design in final applications.

TestId	34m_Test
Image ID	13_121B2A16
Input Image	mapped_06.bin
Ny	16
Nx	16
Nz	224
Set	01c_Set
EN_RUNCFG	0
RESET_TYPE	1
J_GEN	8
CODESET_GEN	0
REF_SAMPLE_GEN	64
W_BUFFER_GEN	64
DISABLE_HEADER_GEN	0
EDAC	0
REF_SAMPLE	64
output image	comp_34m.dat
Ny_GEN	16
Nx_GEN	16
Nz_GEN	224
D_GEN	16
ENDIANESS_GEN	le
TECH	0
HEADER_FORMAT	1
SPLITTER_SLICE_SIZE	128
OUTPUT_BUFFER_SIZE	256

Table 4.8: G3 Testcases

4.5 Synthesis results

Once the design was successfully verified by passing the designed test cases, the design was synthesized. The initial results allowed to detect a critical issue in the datapath, a severe critical path that dramatically limited the operational frequency of the whole system. Additional pipelining was applied to effectively remove this critical path, and the system finally reached a reasonable performance. The software tool that has been used to synthesize the design is Synopsis™ Synplify Premier DP synthesizer (version 2021.9), while the selected target FPGA is the Xilinx Kintex UltraScale XCKU040-2FFVA1156E. However, it should be noted that the design is technology agnostic, meaning that any technology, mainly standard cell based Application Specific Integrated Circuits (ASICs) and FPGAs, can be the target of the synthesis stage.

After obtaining some discouraging initial results, additional pipelining was applying to the architecture datapath. As a result, the CDS Reorder Unit was completely designed, described and integrated into the system. After a couple of minor adjustments, all the testcases were once more successfully executed. After this, the updated architecture was synthesized, showing up great improvements over the initial results, as shown in Table 4.9.

Resource	**Result**	**Utilization (%)**
System clock	121.5 MHz	-
I/O Ports	378	-
I/O Register Bits	0	-
Block RAMs	0	0
LUTs	28329	11.68
Non I/O Register Bits	8774	1.80
Ultra RAMs	0	0
DSP48s	4	0.2

Table 4.9: Final clock and resource utilization results

The hardware resource utilization is reasonable (11% of the total Look-Up Tables (LUTs) and 1% of the available Flip-Flop (FF) registers). The achieved system frequency is also adequate: The estimated frequency is 121.5 MHz, which translates into a throughput of 7.776 Gbps.

We can directly compare our results against the original SHyLoC 121.0 Compressor results [16] obtained when synthesizing targeting the XQRKU060, which can be found at Table 4.10. The usage of registers is slightly higher than four times the number of used registers, mainly due to the inclusion of a more complex packing subsystem. The number of used LUTs is notably higher in our design. This might be caused by 2 factors: first, the coordination and bitstream packing logic of the parallelized design is notably more complex than the original design, thus leading to a higher resource consumption. Secondly, the parallelized and original design utilize the exact same number of Digital Signal Processing (DSPs) units, but the developed architecture includes 4 processing lanes, which translates in that some logic is being implemented by using LUTs instead of DSPs. Even with a lower system clock, which could be improved by progressively reducing the critical paths of the pipeline, the developed architecture is able to process data throughputs of up to 7.776 Gbps, which is significantly better than the 2.6 Gbps offered by the original SHyLoC CCSDS 121.0 Compressor.

These results could be further improved by analysing the rest of critical paths, but due to the limited remaining time, it has been decided to perform these optimizations in the short-future.

Resource	Result
System clock	160.2 MHz
I/O Ports	204
I/O Register Bits	-
Block RAMs	0
LUTs	3708
Non I/O Register Bits	1560
Ultra RAMs	-
DSP48s	4

TABLE 4.10: SHyLoC 121.0 resource utilization results

4.6 Result analysis

To get a reference of the quality of the proposed solution, it is compared with a solution using the CCSDS 123.0-B-1 compressor. SHyLoC 123.0-B-1 Compressor is specifically designed to compress 3D data, that is, hyperspectral images. Hyperspectral imaging sensors serve as a prime example for sensors that generate continuous throughputs of various Gbps, for which the developed architecture might be useful.

The SHyLoC CCSDS 123.0-B-1 compressor is a purely sequential design, processing one sample per clock cycle, as long as the selected processing order is BIP [16]. The standard allows to building high-performance implementations by integrating various individual CCSDS 123.0-B-1 compressors and dividing the input images into different segments that are processed in parallel. This direct parallelization scheme has some immediate consequences:

1. Compression rate reduction. CCSDS 123.0 standard is specifically designed to take advantage of the redundancies that tend to be present in hyperspectral data (i.e. spatial and spectral dependencies) and to statistically characterize the image by continuously adapting the prediction weights to the data being processed, ultimately allowing compression rates of up to 4 times the original input size. Dividing the input data into several segments not only means that some redundancies are not detected during the prediction, but also that the evolution of the prediction weights is worse, which ends up leading to obtain lower compression rates.

2. Increase hardware utilization. When replicating entire processors, the use of hardware resources is directly multiplied by the number of processors. In addition, some overhead is to be expected, since additional logic is required for the division of the input images, the general control for coordinating the individual compressors, etc.

3. Operational frequency. The expected operational frequency of the segmented processing should be equal or worse than the original sequential frequency, due to the new required coordination elements and a larger overall occupancy.

SHyLoC CCSDS 123.0-B-1 compressor does not follow this scheme. To get an approximate, optimistic idea, the synthesis results of the original SHyLoC 123 compressor [20], which yields a throughput of 2.43 Gbps when targeting the radiant tolerant XQRKU060 [11], are tripled to represent a parallel architecture consisting of 3 individual compressors, which would yield a throughput of about 7.29 Gbps, close to the throughput obtained for the Parallel121 compressor (7.776 Gbps). The target dimensions are the maximum offered by the AVIRIS sensor [21], 680x512x224 16-bit pixels.

A summary of the different results of the architectures can be found in the table 4.11. Some notable differences can be observed in the use of registers (non-I/O register bits) and memory: Parallel121 requires less registers, and no Block RAMs are used at all while the segmented 123 compressor would use at least 222 Block RAMs (36 Kb per Block RAM). Some subtle differences can also be observed in the use of DSP blocks, as Parallel121 uses only 4, while the segmented architecture would use about 48 (twelve times more). On the other hand, the usage of LUTs is higher in the developed architecture. As said before, this is likely related to the inclusion of a more complex scheme that leads to obtaining a unified, standard-compliant bitstream. In addition, more LUTs are used in the proposed solution for the implementation of logic which might be alternatively implemented by using more DSP blocks.

Resource	**SHyLoC 123.0-B-1**	**Estimated Parallel SHyLoC 123.0-B-1**	**Parallel121**
System clock	151.6 MHz	151.6 MHz	121.5 MHz
Throughput	2.425 Gbps	7.29 Gbps	7.776 Gbps
Block RAMs	74	222	0
LUTs	7667	23001	28329
Non I/O Register Bits	4035	12105	8774
DSP48s	16	48	4

Table 4.11: Result comparison summary

Chapter 5

Conclusions

As a result of this Master's degree final project, the Parallel121, a high performance CCSDS 121.0-B-3 Compressor IP, has been designed, described in VHDL, verified and synthesized. The paradigm that has been applied through all its components is the parallel computing. The SHyLoC CCSDS 121.0-B-3 IP core, which has been deeply studied, has been taken as the starting design of the project. Its two main components, the Unit-Delay Predictor and the Block-Adaptive Encoder, have been adapted to include 4 processing lanes, which help to effectively multiply the processing throughput that the architecture is able to handle. Two different verification methodologies have been applied to satisfactorily verify the design, and it has ultimately been synthesized for the Xilinx Kintex UltraScale XCKU040-2FFVA1156E FPGA.

The parallelization process of the Unit-Delay Predictor was quite straightforward, thanks to the minimal dependencies that exist between the processing lanes. These dependencies appear because this preprocessor is based on comparing each sample with the previous one, so just some basic shortcircuits between the lanes plus an additional sample register are needed to successfully solve these dependencies.

A completely new internal interconnection block, the post-predictor block dispatcher, has been designed to allow a seamless interconnection between the parallelized predictor and the parallelized encoder. This block is necessary because, although it is true that both parts have been parallelized, they do not follow the same parallelization scheme: The Unit-Delay Predictor block receives and predicts blocks of 8 samples every 2 clock cycles, i.e., all its processing lanes predict samples from the exact same block simultaneously. In the other hand, the encoder processing lanes work independently of each other processing

complete blocks in a fully serial manner. The interconnection block prepares the complete blocks of prediction residuals and sends them to the corresponding encoding lane, which will perform the coding of this block.

The parallelization of the encoder was a more complex and planned process, as each of its inner components process the information in different ways. Some original processing components of the SHyLoC IP Core have been replicated, while the control has been adapted to correctly manage each of them. Anyway, a new centralized, operation-based control has been implemented to enable a correct coordination of the processing lanes, allowing to progressively build the output bitstream. Regarding the dependencies between the processing lanes, one of the main challenges is related to the generation of the Zero-Block CDS, as each of these may correspond to multiple blocks of samples, up to 64. To remove these heavy dependencies from the pipeline, the generation of the Zero-Block CDSes has been extracted from the regular processing flow by integrating it into the new centralized control, leading to the introduction of what can be conceptually seen as an independent processing lane. Two independent processing units, the CDS Reorder Unit and the CDS Retirement Unit, have been created for the packing of the final bitstream. These two units include buffers to concatenate the information that is received from the processing lanes in an ordered manner.

Two new AXI4-Stream data interfaces have been designed to simplify the integration of the compressor IP into different, heterogeneous systems, as this standard is widely supported in all kinds of Integrated Circuits (ICs). The input interface includes the basic set of signals that the standard defines, and it is able to read 4 samples per clock cycle, as long as its ready flag is high. The output interface integrates a more complex logic, and additional signals that offer additional information about the data that is being outputted. This information is necessary because the number of bytes that are outputted can vary every clock cycle.

The verification of the design has been performed in two well separated phases. The first phase took place in parallel with the design of the parallelized encoder, and the followed approach was based in the development of a couple of specific block-level testbenches to verify both the parallelized predictor and the internal post-predictor dispatcher. This phase was remarkably useful, as the integration of these blocks into the complete design required minimal effort, and the time spent was scarce.

The second phase of the verification started once all the different components were described in VHDL, and integrated into the top module of the design. The approach of this phase

drastically differs from the previous one, as a system-wide testbench was developed to check that the resultant compressed bitstream of each test, the data outputted by the compressor IP, matches the expected one, the golden reference bitstream. The golden references of each performed test were previously generated by using an internally developed CCSDS 121.0-B-3 Compression software. The developed testbench was adapted from the original SHyLoC testbench. Different tests were designed and executed to help to demonstrate the correctness of the design at different levels: from basic compression runs to tests that specifically emphasized some specific parts of the architecture, to a final test that checked the compression of a large dataset. This phase was exhaustive and time-consuming, and helped to detect and resolve many bugs and unexpected behaviors.

Once exhaustively verified through the system-wide testing process, the design was synthesized for the selected board, a Xilinx KCU105 that includes a Kintex UltraScale XCKU040 FPGA. Thanks to the last changes that were performed over the pipeline, reasonable results were finally achieved. The estimated system clock frequency was defined at 121.5 MHz, while the resource utilization was kept reasonably low. The final throughput achieved is **7.776 Gbps**, which fulfills the objective of the project.

References

[1] B. Zhang, Y. Wu, B. Zhao, J. Chanussot, D. Hong, J. Yao, and L. Gao. Progress and Challenges in Intelligent Remote Sensing Satellite Systems. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 15:1814–1822,, 2022.

[2] B. Zhang, Y. Wu, B. Zhao, J. Chanussot, D. Hong, J. Yao, and L. Gao. Progress and Challenges in Intelligent Remote Sensing Satellite Systems. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 15:1814–1822,, 2022.

[3] M. Cabral, R. Trautner, R. Vitulli, and C. Monteleone. Efficient data compression for spacecraft including planetary probes. volume International Planetary Probe Workshop (IPPW-7), 2010. URL `http://spacewire.esa.int/edp-page/papers/ippw7_paper_358_Cabral_2010-25-05b.pdf`.

[4] Consultative Committee for Space Data Systems. *Lossless Data Compression, Recommended Standard CCSDS 121.0-B-3*. CCSDS, August 2020. Blue Book.

[5] Consultative Committee for Space Data Systems. *Image Data Compression, Recommended Standard CCSDS 122.0-B-2*. CCSDS, September 2017. Blue Book.

[6] Consultative Committee for Space Data Systems. *Low-Complexity Lossless and Near-Lossless Multispectral and Hyperspectral Image Compression, CCSDS 123.0-B-2*, volume 2. CCSDS, blue book edition, February 2019.

[7] JJ Hernández-Gómez, GA Yañez-Casas, Alejandro M Torres-Lara, C Couder-Castañeda, MG Orozco-del Castillo, JC Valdiviezo-Navarro, I Medina, A Solís-Santomé, D Vázquez-Álvarez, and PI Chávez-López. Conceptual low-cost on-board high performance computing in CubeSat nanosatellites for pattern recognition in Earth's remote sensing, 2019. URL `https://easychair.org/publications/open/Tdjm`.

[8] Jan Andersson, Magnus Hjorth, Fredrik Johansson, and Sandi Habinc. Leon processor devices for space missions: First 20 years of leon in space. In *2017 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pages 136–141, 2017. doi: 10.1109/SMC-IT.2017.31.

[9] Antonio Sánchez, Yubal Barrios, Roberto Sarmiento, David Hernández Expósito, and Antonio Sánchez Gómez. A lossless compression solution for SCIP and TuMag instruments aboard of SUNRISE III balloon-borne Solar Observatory. In *2022 37th Conference on Design of Circuits and Integrated Circuits (DCIS)*, pages 01–06, 2022. doi: 10.1109/DCIS55711.2022.9970132.

[10] ESA. SHyLoC IP core. `https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Microelectronics/SHyLoC_IP_Core`, 2018. Accessed: 2021-03-19.

[11] Xilinx AMD. Radiation Tolerant Kintex UltraScale XQRKU060 FPGA Data Sheet (DS882), 04 2022. URL `https://docs.xilinx.com/v/u/en-US/ds882-xqr-kintex-ultrascale`.

[12] L. Santos, A. Gomez, and R. Sarmiento. Implementation of CCSDS standards for lossless multispectral and hyperspectral satellite image compression. *IEEE Transactions on Aerospace and Electronic Systems*, pages 1–1, 2019. ISSN 0018-9251. doi: 10.1109/TAES.2019.2929971.

[13] Y. Barrios, A. J. Sánchez, L. Santos, and R. Sarmiento. SHyLoC 2.0: A Versatile Hardware Solution for On-Board Data and Hyperspectral Image Compression on Future Space Missions. *IEEE Access*, 8:54269–54287, 2020. doi: 10.1109/ACCESS.2020.2980767.

[14] Diego Ventura, Yubal Barrios, Antonio Sánchez, and Roberto Sarmiento. EDAC implementation on a data lossless compressor compliant with the CCSDS 121.0-B-3 standard. In *2021 XXXVI Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6, 2021. doi: 10.1109/DCIS53048.2021.9666189.

[15] Consultative Committee for Space Data Systems. *Lossless Multispectral and Hyperspectral Image Compression, Recommended Standard CCSDS 123.0-B-1.* CCSDS, May 2012. Blue Book.

[16] University of Las Palmas de Gran Canaria. *SHyLoC Product Datasheet*, Oct 2017. `https://amstel.estec.esa.int/tecedm/ipcores/SHyLoC_Datasheet_v1.0.pdf`.

[17] arm. *AMBA AXI-Stream Protocol Specification*, 2021. https://developer.arm.com/documentation/ihi0051/a/.

[18] Shiyu Liu, Dongfang Li, Wei Shen, Zhihao Wang, Guang Yang, and Xiaojing Song. Application Research of Formal Verification in Aerospace FPGA. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 797–805, 2021. doi: 10.1109/QRS-C55045.2021.00122.

[19] Weimin Fu, Orlando Arias, Yier Jin, and Xiaolong Guo. Fuzzing Hardware: Faith or Reality? : Invited Paper. In *2021 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pages 1–6, 2021. doi: 10.1109/NANOARCH53687.2021.9642252.

[20] Luis Alberto Aranda, Antonio Sánchez, Francisco Garcia-Herrero, Yubal Barrios, Roberto Sarmiento, and Juan Antonio Maestro. Reliability analysis of the shyloc ccsds123 ip core for lossless hyperspectral image compression using cots fpgas. *Electronics*, 9(10), 2020. ISSN 2079-9292. doi: 10.3390/electronics9101681. URL https://www.mdpi.com/2079-9292/9/10/1681.

[21] M. J. Ryan and J. F. Arnold. The lossless compression of AVIRIS images by vector quantization. *IEEE Transactions on Geoscience and Remote Sensing*, 35(3):546–550, 1997. doi: 10.1109/36.581964.