Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Besjan Tomja

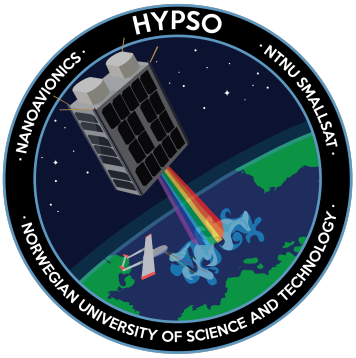
Enhancing Versatility and Efficiency of the CCSDS-123 Compression Algorithm through Dynamic Partial Reconfiguration for the HYPSO Mission

Master's thesis in Embedded Computing Systems
Supervisor: Milica Orlandic
Co-supervisor: Dordije Boskovic
July 2023

NTNU
Norwegian University of
Science and Technology

Besjan Tomja

Enhancing Versatility and Efficiency of the CCSDS-123 Compression Algorithm through Dynamic Partial Reconfiguration for the HYPSO Mission



Master's thesis in Embedded Computing Systems
Supervisor: Milica Orlandic
Co-supervisor: Dordije Boskovic
July 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Abstract

Hyper-Spectral Small Satellite for Ocean Observation (HYPSO) is a space mission at the Small-SatLab at NTNU in Trondheim with the mission to provide rapid and continuous capturing and monitoring of the Norwegian coast. The first mission was HYPSO-1 launched in 2022, which is a 6U CubeSat containing a Hyperspectral Imaging (HSI) Payload. HYPSO's mission includes a lot of high-complexity tasks concerning onboard data handling or processing that consist of machine learning and artificial applications. This is mainly supported by the use of Commercial off-the-shelf products (COTS) like FPGAs (Field Programmable Gate Array) which are high-density programmable logic devices that are very popular in space projects because of their reconfigurable characteristic. One of the challenges that HYPSO missions and space missions in general face is the possibility to change the hardware logic with the aim to update, remove or interchange the existing logic with a new one. This thesis adds this functionality to the HYPSO mission by providing a dynamic reconfiguration approach. Besides the reconfiguration functionality, this thesis provides another approach to increase the flexibility and efficiency of the on-board compression algorithm by allowing the change of the image dimensions dynamically, since based on the previous master thesis it was noted that the compression core cannot be used for varied hyperspectral images.

Preface

This work concludes the Master of Science (MSc) degree of the European Master in Embedded Computing Systems at the Department of Electronic Systems at the Norwegian University of Science and Technology (NTNU). I would like to express my sincere gratitude to my supervisor, Milica Orlandic, for her invaluable guidance and unwavering support throughout this research. I am also grateful to my co-supervisor, Dordije Boskovic, for his valuable contributions and collaborative approach.

I am truly thankful to the Erasmus Mundus in Embedded Computing Systems (EMECS) program for providing me with the unique opportunity to pursue my Master's degree at two renowned universities, Technische Universität Kaiserslautern (TUK) and Norwegian University of Science and Technology (NTNU). The multidimensional experience gained from studying at these institutions has broadened my perspective and enriched my academic journey.

I extend my sincere appreciation to the program coordinators, faculty members, and selection committee for their belief in me and this invaluable experience. The diverse curriculum and support from my EMECS classmates have fostered personal and academic growth. Lastly, heartfelt thanks to my family for their unwavering support and encouragement.

Table of Contents

List of Figures **viii**

List of Tables **x**

1 Introduction **1**

1.1 Motivation 1

1.2 Hyperspectral Imaging and Processing 1

1.3 HYPSO Mission 1

1.4 Dynamic Reconfiguration 2

1.5 Specialisation Project 2

1.6 Main contributions 2

1.7 Structure of the Thesis 3

2 Background **4**

2.1 CubeSats 4

2.2 Hyperspectral Data Representation 4

2.2.1 Component Orderings 5

2.3 Heterogeneous Platforms 6

2.3.1 Comparison of ASICs, FPGAs, CPUs, and GPUs 6

2.4 FPGA Architecture 7

2.4.1 The Logic Fabric 7

2.4.2 Zynq-7000 SoC 8

2.4.3 Zynq UltraScale+ MPSoC 9

2.4.4 Fabric Structure and Reconfiguration 9

2.5 FPGA Design Flow 10

2.5.1 Design Entry 11

2.5.2 Behavioral Simulation . . . 11
2.5.3 RTL (Register Transfer Level) Synthesis . . . 12
2.5.4 Netlist Generation . . . 12
2.5.5 Placement and Routing . . . 12
2.5.6 Bitstream Generation . . . 12
2.5.7 Configuration and Testing . . . 13
2.6 AXI4-Lite Interface . . . 13
2.6.1 Introduction to AXI4 Protocol . . . 13
2.6.2 AXI4-Lite Communication Flow . . . 13
2.6.3 Features and Advantages of AXI4-Lite . . . 15
2.7 Design Tools and Environments . . . 15
2.8 Embedded Operating System . . . 16
2.8.1 Types of Embedded Operating Systems . . . 16
2.8.2 Yocto Project . . . 17
2.8.3 PetaLinux . . . 17
2.8.4 Device Drivers . . . 18
2.8.5 Device Tree Overlay (DTO) . . . 18
2.8.6 Integration of Embedded Operating Systems with Dynamic Partial Reconfiguration . . . 19
2.9 Boot and Configuration . . . 19
2.9.1 Boot Image . . . 19
2.9.2 Boot Process Stages . . . 20
2.10 FPGA Manager . . . 21
2.10.1 Overview . . . 21
2.10.2 Functionalities . . . 23
2.10.3 FPGA Manager Execution Flow . . . 24
2.11 Design Constraints . . . 24
2.12 FPGA Cores and Primitives . . . 25
2.12.1 CubeDMA core . . . 25
2.12.2 Compression Algorithm . . . 27
2.12.3 Pipeline Implementation . . . 29
2.13 Summary . . . 29

3 Programmable Logic Reconfiguration **30**

3.1 Types of Configuration . . . 30
3.2 Partial Reconfiguration in FPGAs . . . 31

3.2.1 Terminology . . . 31
3.2.2 General Concept of Partial Configuration . . . 32
3.2.3 Partial Reconfiguration Methodologies . . . 32
3.2.4 Partial Reconfiguration Challenges . . . 33
3.3 Floorplanning . . . 33
3.4 PL Reconfiguration Paths . . . 34
3.4.1 JTAG . . . 34
3.4.2 PS PCAP . . . 35
3.4.3 ICAP . . . 35
3.5 Review of Proposed Custom PR Controllers . . . 36
3.5.1 AC_ICAP . . . 36
3.5.2 HSDPRC . . . 37
3.5.3 ZyCAP . . . 37
3.5.4 MiCAP and MiCAP-PRO . . . 37
3.5.5 Tiny ICAP Controller . . . 38
3.5.6 DyRACT . . . 38
3.5.7 VR-ZYCAP . . . 38
3.5.8 Comparative Analysis . . . 38
3.5.9 Emerging Trends and Future Directions . . . 39

4 Cude Dimension Controller Module 41
4.1 Introduction . . . 41
4.2 Motivation . . . 41
4.2.1 Relevance of AXI4-Lite in Hyperspectral Imaging Applications . . . 42
4.3 Implementation Details . . . 42
4.3.1 Custom AXI4-Lite IP . . . 42
4.3.2 RTL Code . . . 43
4.4 Device Driver Development . . . 44
4.4.1 Build and Install . . . 44
4.5 Testing . . . 46
4.6 Conclusion . . . 47

5 Dynamic Partial Reconfiguration using PCAP 48
5.1 Vivado Software Flow . . . 48
5.1.1 Synthesis . . . 48
5.1.2 RTL Project Flow in Vivado IDE . . . 50

5.1.3 Assembly of the design . . . 51
5.1.4 Floorplanning . . . 51
5.1.5 Implementation of the configurations . . . 52
5.1.6 Bitstream Generation . . . 53
5.2 PL Configuration through PS . . . 54
5.3 Summary . . . 57

6 Dynamic Partial Reconfiguration Testing 58
6.1 Introduction . . . 58
6.2 Experimental Setup . . . 58
6.2.1 ZedBoard System Architecture . . . 58
6.2.2 Additional Tools and Equipment . . . 59
6.2.3 Considerations for Testing Dynamic Partial Reconfiguration . . . 60
6.3 Testing Methodology . . . 60
6.4 Directory Structure . . . 60
6.5 Synthesis and Assembly . . . 61
6.5.1 Synthesis . . . 61
6.5.2 Assemble the Design . . . 62
6.6 Floorplanning . . . 63
6.7 Implementation of Configurations . . . 63
6.8 Bitstream Generation . . . 64
6.9 CubeDMA Verification . . . 64
6.9.1 CubeDMA Kernel Driver . . . 64
6.9.2 CubeDMA Reserved Area . . . 66
6.9.3 Verifying Script . . . 67
6.10 CCSDS-123 Verification . . . 67
6.11 Reconfiguration Using FPGA Manager . . . 68

7 Experimental Results 69
7.1 Resource Utilization . . . 69
7.2 Reconfiguration Speed . . . 70
7.3 Bitstream Upload Time Analysis . . . 71
7.3.1 Comparison and Implications . . . 72
7.4 Power Consumption . . . 72
7.5 Conclusion . . . 73

8 Conclusion **74**

8.1 Summary 74

8.2 Conclusion 74

8.3 Future Work 75

Appendix 83

A Partial Reconfiguration Synthesize TCL Script 83

B Reconfiguration using the devcfg library 84

C DPR Github Repository 85

List of Figures

2.1 1U(left) and 3U (right) CubeSat 4
2.2 Hyperspectral Image Cube 5
2.3 Illustration of the three different sample orderings of spectral images 5
2.4 The logic fabric of a Zynq device 7
2.5 Zynq-7000 SoC Architecture Overview 8
2.6 Zynq UltraScale+ MPSoC CG Architecture Overview 9
2.7 FPGA architecture of the ZedBoard 10
2.8 Simplified design flow of FPGA 11
2.9 AXI4 Read and Write Transaction 14
2.10 Simple form of Boot Image with FSBL and PMU Firmware 20
2.11 Detailed Boot Flow Example 21
2.12 System organization in case of full reconfiguration using FPGA Manager 22
2.13 Partial Reconfiguration using FPGA Manager 23
2.14 Overview of CubeDMA core 26
2.15 CCSDS-123 compressor Schematic 28
2.16 CCSDS-123 pipeline implementation 29

3.1 Illustration of FPGA configuration using full and partial bitstreams 30
3.2 Partial Reconfiguration Notations 31
3.3 Concept of configurations in Partial Reconfiguration 32
3.4 PL Configuration Paths 34
3.5 Xilinx ICAP Primitive 36
3.6 Comparison of Partial Reconfiguration Controllers 39

4.1 DimController - Custom AXI4-Lite IP 42
4.2 Overview of the automatic verification system 47

5.1 Graphical PR flow representation based on DFX 49
5.2 High-level structure of the design 50
5.3 Design including CCSDS-123, CubeDMA and DimController components 50
5.4 Structure of the full and partial bitstream files 53
5.5 High-level flow of PL configuration through PS 54

6.1 The layout of ZedBoard by Avnet 59
6.2 Selected area for black box ccsds123, instantiated as ji 63

7.1 Total On-Chip Power Consumption for Different Configurations 73

List of Tables

2.1 Comparison of CPU, FPGA, ASIC, and GPU 7

7.1 Resource Utilization Summary for Configuration 1 70
7.2 Resource Utilization Summary for Memory and DSP (Configuration 1) 70
7.3 Resource Utilization Summary for Configuration 2 71
7.4 Resource Utilization Summary for Memory and DSP (Configuration 2) 71
7.5 Bitstream sizes for different configurations 71

Acronyms

API	Application Programming Interface
APU	Application Processing Unit
ASICs	Application-Specific Integrated Circuits
ATF	ARM Trusted Firmware
AXI	Advanced eXtensible Interface
BIL	Band Interleaved by Line
BIP	Band Interleaved by Pixel
BLE	Basic Logic Element
BSQ	Band Sequential
CCSDS	Consultative Committee for Space Data Systems
CLI	Command Line Interface
CLP	Configurable Logic Blocks
CPUs	Central Processing Units
CSU	Configuration Security Unit
EEMI	External Embedded Memory Interface
DFA	Deterministic Finite Automaton
DFX	Device Feature Exchange
DMA	Direct Memory Access
DPR	Dynamic Partial Reconfiguration
DRC	Design Rule Check
DSP	Digital Signal Processing
DTO	Device Tree Overlay
FSBL	First Stage Boot Loader
FSM	Finite State Machine
FPGAs	Field-Programmable Gate Arrays
GPUs	Graphics Processing Units
HDL	Hardware Description Language
HSI	Hyperspectral Imaging
HLS	High Level Synthesis
ICAP	Internal Configuration Port
IOBs	I/O Blocks
IOP	I/O Peripherals
IP	Intellectual Property
OS	Operating System
LUT	Look Up Table
LUTs	Lookup Tables

MCOS Microcontroller Operating Systems
MCU Microcontroller
MM2S Memory Map to Stream
MPU Microprocessor
OCM On-Chip Memory
PCA Principal Component Analysis
PL Programmable Logic
PMU Platform Management Unit
POR Power-on Reset
PS Processing System
RAM Random Access Memory
RM Reconfigurable Module
RP Reconfigurable Partition
RTOS Real-Time Operating Systems
RR Reconfigurable Region
TF-A Trusted Firmware-A
SDK Software Development Kit
S2MM Stream Memory Map
GP General Purpose
HP High Performance
PLB Processor Local Bus

CHAPTER 1

Introduction

1.1 Motivation

In the last decades, CubeSats have been a trend in space missions, used exclusively in Earth Orbit but now they are also being used in interplanetary missions. CubeSats are small satellites, with the first design proposed in late 1990 by professors Jordi Puig-Suari of California Polytechnic State University and Bob Twiggs of Stanford University [1] with the intent of making space science accessible to education. Because of their low cost and their small physical size, nowadays they are being used in many space projects including academic but also commercial ones. CubeSat projects include a lot of high-complexity tasks concerning onboard data handling or processing that consist of machine learning and artificial applications. This is generally supported by the use of Commercial off-the-shelf products (COTS). FPGAs (Field Programmable Gate Arrays) that are high-density programmable logic devices are very popular in space projects due to their reconfiguration nature, allowing modifications based on user requirements. However, the configuration process poses some challenges such as time consumption, execution interruption, and temporary increase in power consumption. To mitigate these issues, dynamic partial reconfiguration of FPGAs can be used to enable selective modification of specific portions of the FPGA while the remaining system continues operation. By leveraging dynamic partial reconfiguration, the configuration time is minimized, execution continuity is maintained, and opportunities for power optimization are realized.

1.2 Hyperspectral Imaging and Processing

Hyperspectral Imaging is a powerful technique that allows capturing of high-resolution images of objects and scenes with hundred or thousand spectral bands. This provides detailed information about the composition and physical properties of materials, allowing for a detailed analysis of complex substances such as minerals, vegetation, and pollutants. This principle is achieved by the hyperspectral sensors, that by measuring light absorption and reflection at different wavelengths, create data cubes with spectral bands for each image pixel. Hyperspectral Imaging has found a great application in space exploration because it allows for geological and mineralogical mapping of planetary surfaces through high-resolution spectral data.

1.3 HYPSO Mission

Hyper-Spectral Small Satellite for Ocean Observation (HYPSO) is a space mission at the Small-SatLab at NTNU in Trondheim with the mission to provide rapid and continuous capturing and monitoring of the Norwegian coast. The first mission was HYPSO-1 launched in 2022, which is a 6U CubeSat containing a Hyperspectral Imaging (HSI) Payload. The mission objectives of

HYPSO-1 include gathering, monitoring, and analyzing ocean color data with a focus on detecting the presence of algae blooms, micro-organisms, and chemical substances. The following mission will be the launch of HYPSO-2 which will include a Software Defined Radio (SDR) with the goal of improving Arctic communication infrastructure in order to enable data retrieval from different remote sensor nodes.

1.4 Dynamic Reconfiguration

Dynamic reconfiguration refers to the ability to modify the hardware configuration of an FPGA device while it is in operation. This allows for the implementation of flexible and adaptive systems that can be reconfigured to meet changing requirements without interrupting operation. Dynamic reconfiguration enables the replacement or addition of modules, the modification of interconnects, and the adjustment of device parameters, all while the system is running [2].

Dynamic reconfiguration enables FPGA-based systems to adapt and respond to evolving requirements and changing operating conditions in real time. Unlike traditional reprogramming methods that require system interruption, dynamic reconfiguration allows for on-the-fly changes to the FPGA's functionality. This capability opens up new possibilities in various domains such as aerospace, telecommunications, software-defined radio, and adaptive computing.

This thesis focuses on exploring the potential of dynamic reconfiguration in FPGA-based systems. It aims to explore the profound impact of dynamic reconfiguration on performance, resource optimization, and run-time adaptability.

1.5 Specialisation Project

This thesis is a continuation of the specialization project with the topic "Onboard image processing pipeline and on-flight FPGA reconfiguration for hyperspectral remote sensing CubeSats". The project aimed to gain a comprehensive understanding of the HYPSO-1 systems, which include the onboard processing pipeline and the onboard processing unit. In the initial phase of the project, significant emphasis was placed on examining the onboard processing pipeline and validating the functionality of various implemented modules, such as classification, dimensionality reduction, smile detection, and keystone correction. Subsequently, the project delved into exploring diverse methodologies for achieving run-time reconfiguration of the FPGA. Based on knowledge and findings from the specialization project, the focus of this master thesis will be to perform dynamic partial reconfiguration of the FPGA based on the CCSDS-123 compression algorithm.

1.6 Main contributions

The project assignment for the master thesis encompasses two primary tasks, which are as follows:

- Conducting a thorough analysis of the CCSDS-123 compression algorithm and modifying its implementation to support the compression of images with various dimensions, thereby enhancing its versatility and applicability.
- Establishing a comprehensive data flow model that enables dynamic reconfiguration of the FPGA, specifically focusing on performing partial configurations based on the CCSDS-123 compression algorithm.

Firstly, during the HYPSO mission, it is necessary to perform the compression on different cube-size dimensions which until now was not possible. One approach to support this is to perform full reconfiguration of FPGA by changing the generic parameters X, Y, and Z, and then creating the boot files based on the changed hardware specification. A more efficient approach is to change the

cube size dimension by writing and reading the cube size dimension from registers in hardware implementation without needing to perform full reconfiguration. Additionally, the second and main part of this master thesis is to perform dynamic partial reconfiguration on-flight based on the CCSDS-123 compression algorithm. Partial reconfiguration offers the opportunity to interchange different modules in the FPGA without affecting the other functional modules in FPGA. In the case of the CCSDS-123 algorithm, the FPGA can be reconfigured to perform compression on different cube dimensions based on the partial bitstream that will be used.

1.7 Structure of the Thesis

This thesis is organized into seven chapters to provide a comprehensive exploration of the topic. The structure of the thesis is as follows:

- **Chapter 2**: Background
  This chapter establishes the necessary foundation by presenting background information on key aspects such as FPGAs, algorithms, reconfiguration concepts, and other relevant factors essential for the development, implementation, and testing of dynamic partial reconfiguration.
- **Chapter 3**: Programmable Logic Reconfiguration
  This chapter presents information regarding PL reconfiguration and also includes a literature review of the proposed reconfiguration controllers.
- **Chapter 4**: Cube Dimension Controller Module
  In this chapter, the implementation of a custom AXI4-Lite component for changing the Y-dimension of the HSI cube is presented, providing detailed insights into the implementation process.
- **Chapter 5**: Dynamic Partial Reconfiguration using PCAP
  This chapter outlines a step-by-step flow that can be followed to perform partial reconfiguration using the CCSDS-123 compression algorithm. It delves into the methodology and techniques employed during the process.
- **Chapter 6**: Dynamic Partial Reconfiguration Testing
  In this chapter, the testing flow conducted to verify the functionality of the presented approach to perform partial reconfiguration is described in detail.
- **Chapter 7**: Experimental Results
  This chapter evaluates the results obtained from the experiments and analysis conducted in the previous chapters. It offers a comprehensive evaluation of the performance and efficacy of the presented approach.
- **Chapter 8**: Conclusion
  This chapter provides a short summary of this master thesis, by also mentioning the conclusions based on the experimental results. Finally, it also includes recommendations for future work.

CHAPTER 2

Background

2.1 CubeSats

The concept of CubeSats was developed at the Department of Aeronautics and Astronautics at Stanford University in collaboration with Jordi Puig-Suari at the Aerospace Department at California State Polytechnic University in 1999 [3], with the aim to develop a design that can be built by students in short time with low cost. The physical standard of CubeSats is based on a 10 x 10 x 10 cm cube, with a mass of around 1.33 kg, referred to as one unit "1U" CubeSat. However, they come in several sizes such as 1.5U, 2U, 3U, 6U, or 12U [4]. In Figure 2.1 an example of 1U and 3U CubeSats are shown.

Figure 2.1: 1U(left) and 3U (right) CubeSat

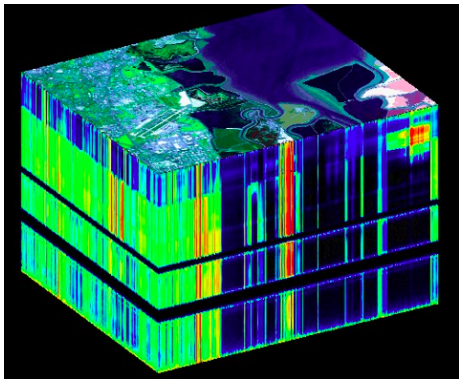
Source: [4]

CubeSats are used in a wide range of missions, such as Earth observation, technology demonstration, communication, and scientific missions. Besides these applications, one popular usage of CubeSats that is supported by their low cost is the studying of Earth's atmosphere and space weather monitoring.

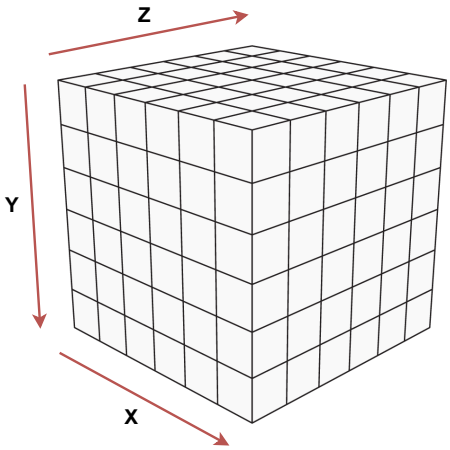
2.2 Hyperspectral Data Representation

A digital image is a rectangular array made up of a certain number of rows and columns, each containing a single pixel value associated with a specific coordinate in the image. In contrast, three-dimensional images, such as hyperspectral images (HSI), consist of a series of contiguous narrow wavelength intervals, forming a cube-like structure with dimensions defined by the number of rows, columns, and spectral bands [5]. Hyperspectral images have many more bands (features) than traditional RGB images, which only consist of red, green, and blue color channels. Figure

2.2b illustrates the structure of a hyperspectral image cube, where the X and Y axes represent the spatial dimensions of the image, while the Z-axis corresponds to the spectral dimension, and Figure 2.2a shows a hyperspectral data cube.



(a) A hyperspectral data cube. (Credit: NASA / JPL)



(b) HSI Cube dimensions

Figure 2.2: Hyperspectral Image Cube

Hyperspectral images (HSI) are characterized by two main features: a large number of spectral bands, up to 1000, and the fact that each pixel in the image is represented by a complete spectrum with associated spectral interpretation, transformation, and data analysis [5]. HSI images are known for their high spectral resolution and strong correlation among adjacent bands, which can provide valuable insights for various applications [6].

2.2.1 Component Orderings

The ordering of components in a three-dimensional HSI cube is important for streaming, storing, and processing purposes. Three are the most common component orderings: Band Interleaved by Pixel (BIP), Band Interleaved by Line (BIL), and Band Sequential (BSQ).

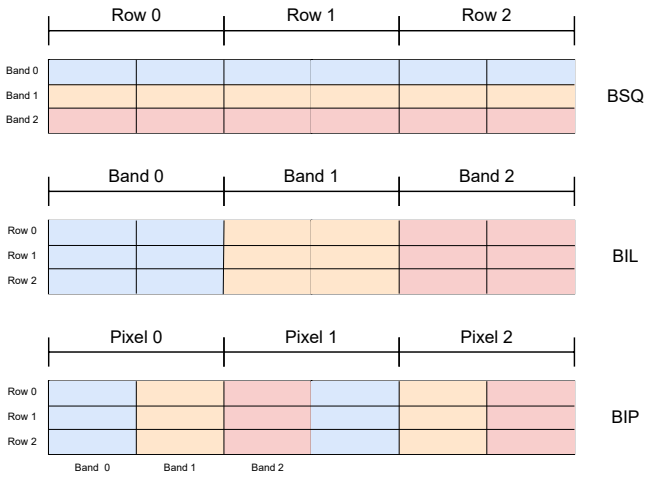


Figure 2.3: Illustration of the three different sample orderings of spectral images

Source: [7]

In the BSQ ordering scheme, components within a hyperspectral data cube are arranged in a manner where all components within each spectral band are grouped together. This arrangement

starts from the upper-left pixel and proceeds to the lower-right pixel. On the other hand, the BIL ordering scheme also organizes components separately for each band, but within each line. Specifically, the components of each pixel in the first line are ordered sequentially, followed by the components of the second line, and so on throughout the entire data cube. This pattern is replicated for subsequent lines within the cube. In contrast, the BIP ordering scheme ensures that all components of a pixel are contiguous. This means that the components of the upper-left pixel are immediately followed by the components of the pixel to its right, and this sequence continues until reaching the lower-right pixel [7]. Figure 2.3 visually illustrates these three orderings using a sample image with dimensions of 3 x 3 x 3, providing a clearer representation of the arrangement methods.

2.3 Heterogeneous Platforms

Heterogeneous platforms refer to computer systems that encompass diverse processors, memory architectures, and communication structures, thereby delivering potent computing capabilities across various application domains like machine learning, big data analytics, and scientific computing. The strength of these platforms lies in their utilization of disparate processors such as CPUs, GPUs, FPGAs, and ASICs interconnected through communication networks.

Application-Specific Integrated Circuits (ASICs) are custom-designed circuits, optimized for specific tasks or applications that are used in many fields of industries such as aerospace, medical, automotive, and customer electronics. ASICs are known for their high performance, low power consumption, and high level of integration. However, ASIC development can be time-consuming and costly, and ASICs are less suitable for applications that require flexibility or quick time-to-market.

On the other hand, Field-Programmable Gate Arrays (FPGAs) are versatile hardware devices that offer the capability to be reconfigured for specific applications. These devices include programmable logic blocks, configurable interconnects, and programmable I/Os, allowing them to be programmed using hardware description languages (HDL). FPGAs can be realized using diverse semiconductor technologies, such as SRAM-based or Flash-based approaches [8].

Graphics Processing Units (GPUs) are hardware components specifically designed for optimized graphics rendering and parallel processing having the capability to execute thousands of threads concurrently, making them highly efficient for parallel computing tasks. The embedded GPU configurations are diverse since these configurations can have different characteristics, including the size of caches, the number of shader cores, and the number of Arithmetic Logic Units (ALUs) per shader core[9].

2.3.1 Comparison of ASICs, FPGAs, CPUs, and GPUs

When it comes to hardware accelerators for various applications, it is important to compare the characteristics of ASICs, FPGAs, and GPUs. Table 2.1 presents a comparison based on various characteristics, reproduced from [10]. As presented, CPUs exhibit moderate flexibility during development and high flexibility after development, allowing for versatile programming and reprogramming capabilities. However, their parallelism and performance levels are relatively low compared to other architectures. On the other hand, FPGAs offer high flexibility during development and after development, making them suitable for a wide range of applications. They excel in parallelism and provide moderate performance. ASICs, specialized chips designed for specific applications, offer very high flexibility during development and low flexibility after development. They deliver high performance and low power consumption but incur high development and production setup costs. Lastly, GPUs provide low flexibility during development but high flexibility after development, enabling efficient parallel processing for tasks like graphics rendering and machine learning. They offer moderate performance and high power consumption. This comparison highlights the trade-offs between these architectures, helping guide the selection of the most suitable computing platform for specific application requirements.

Table 2.1: Comparison of CPU, FPGA, ASIC, and GPU

Characteristics	CPU	FPGA	ASIC	GPU
Flexibility during development	Medium	High	Very High	Low
Flexibility after development	High	High	Low	High
Parallelism	Low	High	High	Medium
Performance	Low	Medium	High	Medium
Power consumption	High	Medium	Low	High
Development cost	Low	Medium	High	Low
Production setup cost	None	None	High	None
Unit cost	Medium	High	Low	High
Time-to-market	None	Medium	High	Medium

Source: [10]

2.4 FPGA Architecture

2.4.1 The Logic Fabric

To fully understand the capabilities of Zynq devices, it is important to have a clear understanding of the various components that make up the programmable logic (PL) portion of the device. The Programmable Logic (PL) of a Zynq device is generally composed of general purpose logic FPGA logic fabric that consists of slices and Configurable logic Blocks (CLBs) which can be configured to perform various logic operations based on the design requirements. Furthermore, the PL also includes Input/Output Blocks (IOBs) that allow the device to interface with external devices or systems. The PL part of the Zynq device together with its components is illustrated in Figure 2.4.

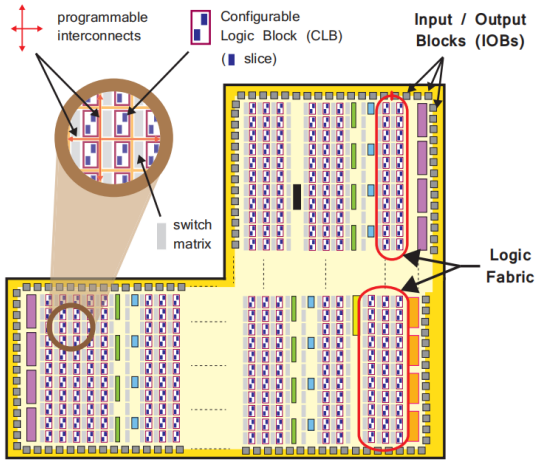


Figure 2.4: The logic fabric of a Zynq device

Source: [11]

The Configurable Logic Blocks (CLBs) are fundamental building blocks in an FPGA that contains a number of resources such as lookup tables (LUTs), and flip-flops and carry logic that can be configured to implement custom logic circuits. They are connected to other resources via programmable interconnects. A sub-unit of CLB is a slice that is used as the basic unit for implementing logic functions. Slices can contain resources like LUTs which are small memory components used to implement combinatorial and sequential logic circuits. They are used typically in combination with other logic elements such as flip-flops, to implement complex digital circuits. Flip-flops are circuit elements used to store a single bit of information that can be used to implement different logics, such as a function of up to six inputs, a small Read-Only Memory (ROM), a small Random Access Memory (RAM), or a shift register [11]. The Switch Matric is a collection of programmable switches that are used to connect various components of an FPGA device, responsible for

routing signals between different functional blocks. IOBs (Input/Output Blocks) are specialized components that are used to interface with external devices that provide flexible I/O capabilities. Carry logic are a chain of routes and multiplexers that are used to link slices in a vertical column. Apart from general logic fabric, FPGAs have two special purpose components: Blocks RAMs for accommodating high-density memory requirements and Digital Signal Processing (DSP) slices for high-speed arithmetic. Both of those resources are placed in columns within the logic array, near each other, since computation-intensive tasks and memory storage operations are often interconnected. Another important concept is the Clock Management Tile (CMT) which is a dedicated hardware block that provides clock generation and distribution, by managing clock signals within the FPGA design.

Depending on the organization, fine-grained and coarse-grained refer to the level of granularity or size of the programmable logic blocks. In fine-grained FPGAs, the logic elements are small and highly configurable such as LUTs, or CLBs. On the other hand, coarse-grained FPGAs have a larger pre-designed logic block, such as IP blocks or predefined functions. Coarse-grained FPGAs offer a higher level of abstraction and can simplify the design process by providing pre-designed blocks for common functions.

2.4.2 Zynq-7000 SoC

The Zynq-7000 is an integrated system-on-chip (SoC) platform created by Xilinx that combines a dual-core ARM Cortex-A9 processor with programmable logic (PL) in a single device. The architecture of the Zynq-7000 SoC, as depicted in Figure 2.5, comprises a single or dual-core ARM Cortex-A9-based processing system (PS) and a programmable logic (PL) section integrated into a single device [12]. The PL utilizes Xilinx's 28nm programmable logic technology, enabling the incorporation of tailored hardware accelerators. Interconnecting the PS and PL is the Advanced eXtensible Interface (AXI), a high-bandwidth interconnect facilitating efficient and rapid communication between the two components. The PS encompasses memory controllers, external interfaces, and system-level features essential for executing software applications. In contrast, the PL comprises programmable logic resources such as CLBs, DSP blocks, and BRAMs [12]. These resources can be configured to implement customized hardware accelerators. Additionally, the Zynq-7000 SoC includes various features that support system-level functionality, including Ethernet interfaces, a USB controller, and more.

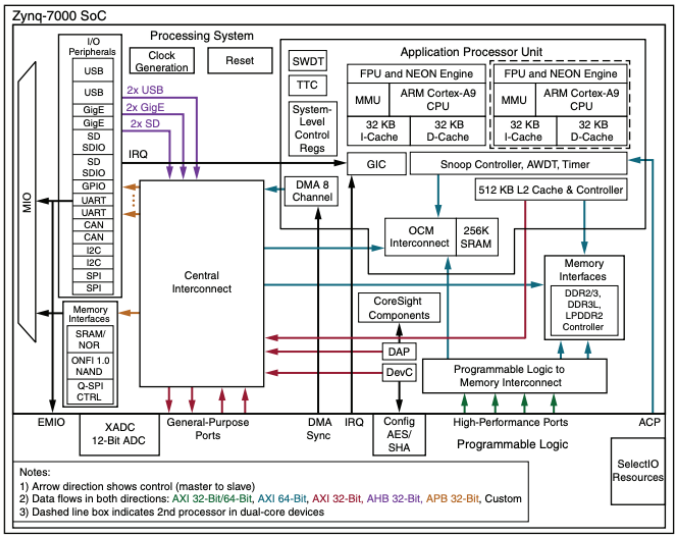


Figure 2.5: Zynq-7000 SoC Architecture Overview

Source: [12]

The Zynq-7000 SoC is used in different applications such as embedded systems, automotive, industrial automation, and networking. In embedded systems, the platform finds extensive application

in real-time control and signal-processing tasks. The automotive industry utilizes it for advanced driver assistance systems (ADAS) and engine control units (ECUs), while industrial automation benefits from its use in process control and monitoring. In the networking domain, the platform plays a crucial role in packet processing and network acceleration. Besides the many advantages that this platform provides, it has its limitations as well. One of the most important ones is the complexity of the system, which requires specialized expertise in hardware-software co-design. Also, it requires careful management of power consumption to avoid overheating.

2.4.3 Zynq UltraScale+ MPSoC

The Zynq UltraScale+ MPSoC, a product of Xilinx, represents an extensively integrated System on Chip (SoC) that combines a quad-core ARM Cortex-A53 processor with a programmable logic subsystem. This cutting-edge generation builds upon its predecessor, the Zynq-7000 SoC, to deliver enhanced capabilities. The MPSoC showcases advanced memory and storage subsystems, encompassing DDR4 memory controllers and High-Speed Serial I/O interfaces. Supporting both 32-bit and 64-bit operating systems, it finds utility in diverse applications such as embedded systems, networking, and automotive domains. In contrast to conventional SoC solutions, the Zynq UltraScale+ MPSoC demonstrates exceptional programmability, flexibility, high performance, and low power consumption. Additionally, it features secure boot, encrypted bitstream programming, and tamper detection to address security concerns.

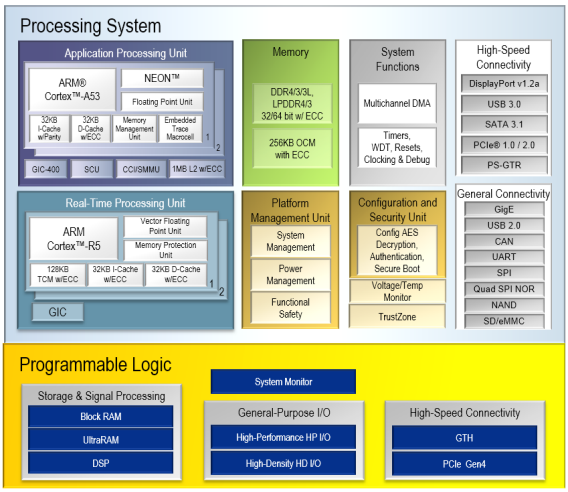


Figure 2.6: Zynq UltraScale+ MPSoC CG Architecture Overview

Source: Xilinx

2.4.4 Fabric Structure and Reconfiguration

This section will discuss the general architecture of an Xilinx FPGA device. FPGAs typically include CLBs, IOBs, and logic cell arrays (LCAs). The FPGA architecture is composed of columns of IOBs, CLBs, DSPs, CMTs, and BRAMs. As illustrated in Figure 2.7, FPGA horizontally is divided into two halves, the top half and the bottom half, represented by TOP(0) and Bottom(1), respectively. In each of those halves, there can be one or several horizontal clock rows (HCLK), based on the FPGA family and producer. For example, Figure 2.7, is illustrated the structure of the ZynQ SoC FPGA, and as it can be observed, each half includes four HCLKs. Furthermore, each of those HCLKs consists of a number of CLBs, BRAMs, DSPs, and IOBs depending on the specific device.

Each HCLK row represents a major row and is divided by regional clock resources into an upper and lower half. The height of an HCLK row corresponds to the CLB column (50 CLBs), and its width matches the device width. Each HCLK row contains different numbers of CLBs, BRAM,

DSPs, and I/Os based on the device width. A CLB column consists of 400 LUTs, with each CLB containing four LUTs per slice. The LUTs in the seven-series architecture are capable of implementing six-input boolean functions with a 64-bit initialization value, and they can also be used as dual five-input LUTs with shared inputs.

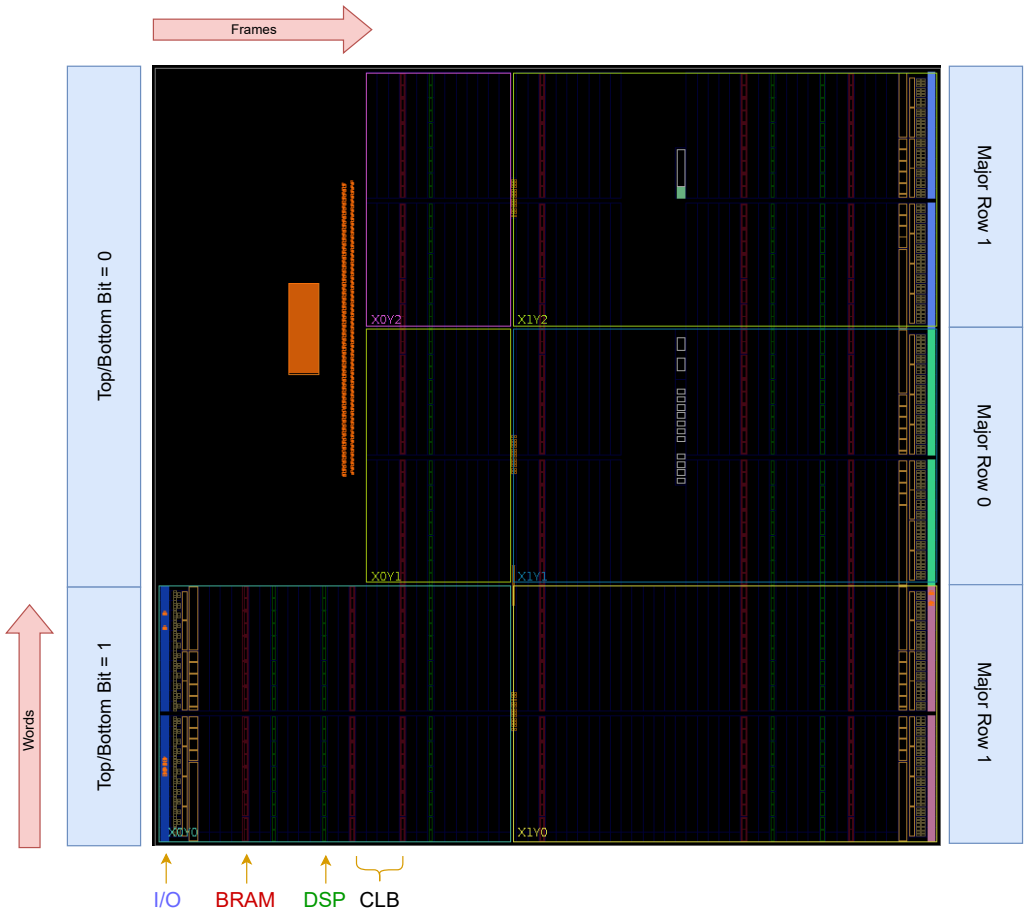


Figure 2.7: FPGA architecture of the ZedBoard

To dynamically reconfigure a system at run-time, the location of the logic resource needs to be known in both the XY coordinate system and the configuration memory space. The XY coordinates of the target slice, specified in a constraint file, can be mapped to the corresponding frame address in the configuration memory using the frame address register (FAR), which is a 32-bit value. The LUT location is defined as a pair of (X, Y) coordinates and a BEL value, where X represents the row, Y represents the column of a slice, and BEL specifies the LUT position within the slice (LUTA, LUTB, LUTC, or LUTD).

2.5 FPGA Design Flow

The FPGA design flow is the process of creating a digital circuit using an FPGA device. The flow consists of several stages, as shown in Figure 2.8. The first stage is the definition and specification of the design requirements. Following the design specification, the high-level design stage begins, employing hardware description languages (HDLs) like VHDL or Verilog. This stage involves representing the circuit's structure and functionality using block diagrams or modular representations. The high-level design allows designers to capture the overall architecture and functionality before diving into the implementation details.

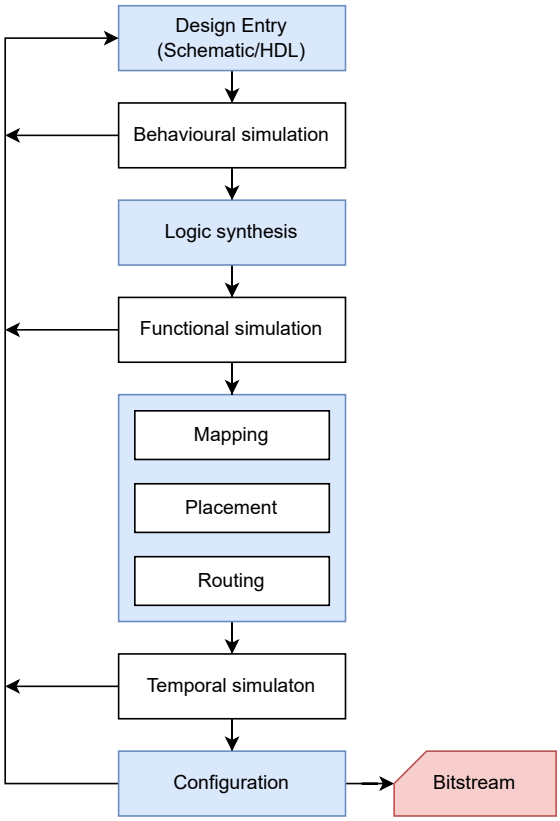


Figure 2.8: Simplified design flow of FPGA

Source: [13]

2.5.1 Design Entry

Design Entry is the stage of FPGA design flow where the design is described using hardware description languages (HDL) like VHDL or Verilog. Hardware description languages are used to specify the behavior and the structure of the design. In some cases, this code can be generated using high-level synthesis (HLS) tools. In order to enable the partial reconfiguration of the design, some changes are required at this stage. First, the reconfigurable modules need to be incorporated into the overall system design, by also identifying the boundaries and defining the necessary interfaces between them and static region. Secondly, other changes that can be done in this stage are related to the configuration path that will be used for partial reconfiguration. This includes the addition of the IP, representing the hardware controller, that facilitates this process.

2.5.2 Behavioral Simulation

Behavioral simulation is a crucial step in the design process as it allows for the testing and validation of the functionality of a circuit. This testing is performed using simulation tools, where the HDL code is simulated with input test vectors. The purpose of this simulation is to verify that the circuit behaves as intended and meets the desired requirements. By subjecting the HDL code to simulation, designers can observe the behavior of the circuit and identify any logical or functional errors that may be present. This process helps in detecting and resolving issues before moving forward with subsequent stages of the design flow. In terms of DPR (Dynamic Partial Reconfiguration), this stage plays an important role in creating additional reset scenarios that should be considered to validate the correct behavior and the interaction of the reconfigurable modules with the static region.

2.5.3 RTL (Register Transfer Level) Synthesis

RTL synthesis involves translating a high-level design description into a gate-level representation. The RTL code is analyzed and optimized by synthesis tools to generate a netlist consisting of gates and flip-flops that represent the circuit's behavior. This gate-level representation serves as a basis for further implementation steps. During the DPR, this stage will handle the synthesis of both the static and the reconfigurable partitions of the design.

2.5.4 Netlist Generation

After RTL synthesis, a netlist is generated, capturing the circuit's structure and connectivity information. The netlist represents the circuit in terms of gates, flip-flops, and interconnections. It provides a structural representation that serves as input for subsequent implementation steps, such as placement and routing. In the DPR, this stage is modified to generate separate netlists for the static and reconfigurable portions of the design. The netlist for the static portion represents the always-operating components, while the netlist for the reconfigurable portion represents the modules that can be dynamically loaded and swapped in and out during runtime.

2.5.5 Placement and Routing

Placement and routing involve mapping the circuit's netlist onto the physical resources of the FPGA device. Placement determines the optimal location for each logic element on the FPGA, while routing establishes the interconnections between these elements. Placement and routing tools consider various factors like performance, timing, and power consumption to optimize the circuit's layout and ensure efficient utilization of FPGA resources. This stage is also impacted during the partial reconfiguration since the placement and routing should consider the interaction between the static and reconfigurable modules. The placement must ensure that the static region is placed and routed efficiently, while also reserving sufficient space for the reconfigurable modules. However, a crucial role in the DPR plays floorplanning, which is the definition of where the reconfigurable modules will be placed. Before the placement and routing phase, some analysis regarding the placement of the static region can be performed, in order that the floorplanning of the reconfigurable modules to be defined considering the short interaction time between the two portions.

2.5.6 Bitstream Generation

A Bitstream is a binary file containing a sequence of bits that includes all the configuration information for a specific FPGA. It also contains commands that can control the functionality of the chip. There are two types of bitstreams: full and partial. Full bitstreams configure the entire FPGA, while partial bitstreams only configure specific sections of the device. The size of a partial bitstream is proportional to the size of the region that it is reconfiguring. This is because partial bitstreams have the same structure as full bitstreams, but are limited to specific address sets to program only part of the device.

A Bitstream file consists of three main parts: Header Packets, Configuration bits, and Trailer Packets. Header Packets are used for configuration preparation, Configuration bits are a sequence of data packets that contain configuration information, and Trailer Packets are used to clean up the configuration process. There are different formats for bitstreams, but they all contain these three parts. The most commonly used formats are .bin and .bit. The .bit format includes a text header (usually 129 bytes) that provides information about the bitstream name, user ID, or FPGA board. Besides the .bit format, the bitstream can be found in the .bin format that is used for configuring Xilinx FPGAs. However the .bin format, in contrast to the .bit format, does not include a text header.

This stage is the most affected one while the DPR is used. The bitstream generation stage is

modified to generate both the configuration bitstream for the static portion of the design and the partial bitstreams for the reconfigurable modules. The configuration bitstream includes the necessary information to initialize the static region of the FPGA. The partial bitstreams contain the configuration data for the reconfigurable modules and are dynamically loaded into the FPGA during run-time.

2.5.7 Configuration and Testing

In this stage, the generated bitstream file is loaded onto the FPGA device to configure the device so that it can operate according to the designed circuit. After the FPGA is programmed, the next step is to perform testing and validation in order to ensure the correctness and if it meets the specified requirements. In this step, test vectors or test benches are used in order to test functionality or performance. During partial reconfiguration, the configuration and testing stage is also modified to accommodate the dynamic loading and unloading of partial bitstreams. Instead of configuring the entire FPGA with a single bitstream, only the relevant sections of the FPGA are configured using partial bitstreams. The partial bitstreams containing the configuration data for the reconfigurable modules are loaded into the FPGA during run-time, allowing for dynamic changes to the circuit functionality.

2.6 AXI4-Lite Interface

The AXI4-Lite interface is a bus protocol, used in the field of digital design and system-on-chip (SoC) integration that provides a simple type of communication between the processor and peripheral devices within an SoC. As the name suggests, it is based on the Advanced eXtensible Interface (AXI) standard developed by ARM. This section will give a brief introduction to the protocol and communication flow, highlighting its features, advantages, and relevance in hyperspectral image applications.

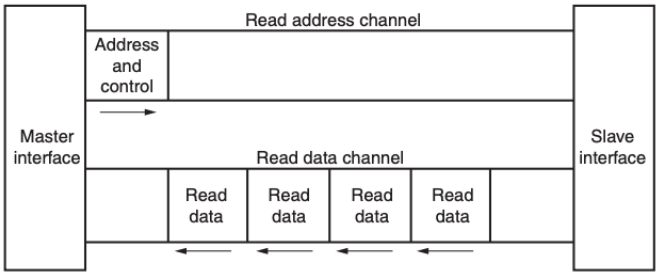
2.6.1 Introduction to AXI4 Protocol

AXI is a part of the ARM AMBA family of microcontroller buses. It was first introduced in 1996 and has undergone several versions, including AXI4, which was included in AMBA 4.0 released in 2010. The AXI4 interface includes three types: AXI4 for high-performance memory-mapped requirements, AXI4-Lite for simple low-throughput memory-mapped communication with control and status registers, and AXI4-Stream for high-speed streaming data [14]. AXI4 has become the go-to interface in Xilinx product offerings, and it brings a lot of benefits that boost productivity, flexibility, and availability. Firstly, by adopting the AXI interface, developers can focus on learning just one protocol for IP, saving time and effort. Secondly, AXI4 offers different versions tailored to specific applications. AXI4 provides high-performance memory-mapped interfaces, supporting lightning-fast data transfers in bursts. On the other hand, AXI4-Lite is a lightweight, user-friendly interface perfect for quick and simple memory-mapped communication. Another advantage is the wide availability of AXI, as it has become an industry-standard protocol [14].

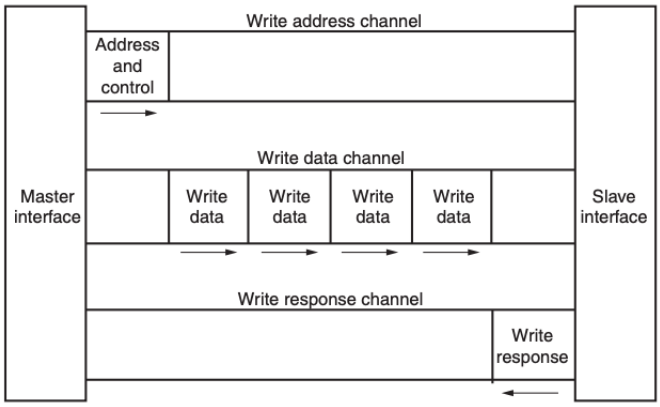
2.6.2 AXI4-Lite Communication Flow

The AXI4-Lite interface comprises five communication channels: Read Address, Read Data, Write Address, Write Data, and Write Response. Figure 2.9 illustrates the AXI4 read and write transactions. As illustrated, the data can move in both directions between the master and slave simultaneously. However, it should be noted that Figure 2.9 illustrated burst transfers, which is a characteristic of AXI4, but AXI4-Lite allows only one data transfer per transaction [14]. The AXI4 interface includes a VALID/READY handshake process across all five transaction channels to facilitate the transfer of address, data, and control information. This mechanism allows both

the master and slave components to regulate the flow rate of information between them. The master generates a VALID signal to indicate the availability of address, data, or control information, while the slave generates a READY signal to signify its readiness to receive the information. The handshake process is considered successful when both the VALID and READY signals are asserted concurrently during a rising clock edge. This ensures efficient and synchronized communication between the master and slave, enabling controlled information transfer.



(a) AXI4 Read Transaction



(b) AXI4 Write Transaction

Figure 2.9: AXI4 Read and Write Transaction

Source: [14]

In an AXI4-Lite read transaction, as shown in Figure 2.9a the master initiates the process by sending a valid address on the Read Address channel and asserting the ARVALID and RREADY signals. The slave responds by asserting the ARREADY signal, indicating its readiness to receive the address. Upon the next rising clock edge, the handshake occurs, and both the ARVALID and ARREADY signals are de-asserted, indicating successful address transmission. The slave then provides the requested data on the Read Data channel, asserting the RVALID signal to indicate its validity. Once the RREADY and RVALID signals are both asserted, the transaction is completed, and the RREADY and RVALID signals can be de-asserted.

Similarly, in an AXI4-Lite write transaction, illustrated in Figure 2.9b the master sends the address and data on the Write Address and Write Data channels while asserting the AWVALID and WVALID signals. It also asserts the BREADY signal, indicating its readiness to receive a response. The slave responds by asserting the AWREADY and WREADY signals, completing the handshakes, and acquiring the write address and data. The slave then provides a response on the Write Response channel, asserting the BVALID signal to indicate its validity. The transaction is completed on the next rising clock edge when both the ready and valid signals on the Write Response channel are high.

2.6.3 Features and Advantages of AXI4-Lite

The AXI4-Lite interface offers a simplified and streamlined version of the AXI4 interface, specifically designed for simple, low-throughput memory-mapped communication. The advantages of the AXI4 protocol interface include:

- Simplicity: It provides a simpler and more lightweight design and offers an interface for low-throughput applications.
- Resource Efficiency: It has a smaller logic footprint, making it suitable for resource-constrained designs.
- Reduced Complexity: It supports only single transactions, simplifying communication between components.
- Easier Integration: It can be easily integrated into SoC designs
- Lower Power Consumption: It consumes less power compared to AXI4, benefiting power-sensitive applications.
- Faster Development time: Due to its simplicity and availability of resources lead to quicker development times for low-throughput applications.

In general, AXI4-Lite offers a more streamlined and efficient approach for memory-mapped communication with low data transfer needs, when compared to AXI4. Its benefits lie in its simplicity, resource efficiency, compatibility, and reduced complexity, making it the preferred option for applications that require less data throughput.

2.7 Design Tools and Environments

This section provides an overview of the most commonly used tools and environments in FPGA design. These tools are integral to the FPGA design flow, providing support in different stages such as circuit creation, simulation, synthesis, optimization, and implementation

- **Xilinx Vivado** Vivado is a comprehensive software suite developed by Xilinx for designing, implementing, and optimizing digital designs for Xilinx FPGAs and SoCs. It offers features like design entry, simulation, synthesis, place-and-route, and bitstream generation which were described in the previous sections [15].
- **Vitis SDK**
  Vitis Software Development Kit (SDK) is another tool provided by Xilinx that is designed to facilitate the creation of applications that leverage the power of FPGA and SoC devices. Vitis SDK helps to design accelerated applications by integrating hardware accelerators with embedded software that executes on the CPU. This tool supports programming languages such as C, C++, or OpenCL making it very accessible to a wide group of developers. Furthermore, it includes a set of development tools such as compilers, debuggers, profilers, and libraries that help during the software development process. This tool is integrated with other Xilinx development tools and platforms, such as Vivado Suite, which helps to facilitate the development process [16].
- **Vivado HLS**
  Vivado HLS (High-Level Synthesis) simplifies FPGA design by converting high-level code (C, C++, or SystemC) into optimized RTL code suitable for FPGA implementation. It automates analysis, applies optimizations, and generates RTL code that can be further synthesized and implemented using Vivado. Vivado HLS reduces design time and effort by automating complex RTL design tasks. It enables designers to explore design options, optimize performance, and evaluate trade-offs between performance, area utilization, and power consumption [17].

- **Version Control System**
  A Version Control System (VCS) is essential for code management and collaboration. Git, a distributed VCS, is widely used for its features and integration with platforms like GitHub. Git tracks code changes, allowing offline work and collaboration among geographically dispersed teams. GitHub provides a centralized platform for code sharing, issue tracking, and collaboration, promoting open-source development and facilitating knowledge sharing in software development projects [11].

2.8 Embedded Operating System

An operating system is a crucial software component responsible for the management of the computer system's hardware [18]. In the context of embedded systems, the operating system plays a vital role in enabling these systems to perform their designated tasks within a larger framework. An embedded operating system is a specialized software system designed to run on embedded devices with limited computing resources such as a microcontroller, single-board computer, or custom-designed hardware. The main component of the operating system is the kernel, which is a program responsible for managing computer system resources and allocating them between different processes and users. The kernel operates at the privileged level of the processor so that the user code cannot access certain hardware resources. Overall, the kernel's functions include managing resources, controlling hardware access, and facilitating context switching between processes [18].

Today exist different types of operating systems, with many specific characteristics, however, some of the general characteristics that embedded operating systems have, based on [18], include:

- Subsystem of a device or machine: Embedded systems are components within more board systems, such as engine management systems or microprocessors, and can also exist as standalone devices like access points or routers.
- Dedicated application: Embedded systems are designed to perform specific tasks, though modern devices may support multiple applications.
- Small footprint: Embedded systems are typically designed with a minimal footprint, driven by the specific task they perform.
- Low power consumption: Embedded Systems in most cases are battery-powered and require energy-efficient operation.
- Interaction with the physical world: Embedded systems are involved in physical computing, sensing, and responding to the analog world.
- Single board computers (SBCs): Entire computer systems can be built on a single printed circuit board, known as SBCs, which are commonly used as platforms for embedded systems.

These characteristics collectively define the general aspects of embedded operating systems, however, it is important to note that there are many other characteristics specific to embedded systems that can vary based on their intended applications.

2.8.1 Types of Embedded Operating Systems

- Real-Time Operating Systems (RTOS)
  Real-Time Operating System is an operating system designed for the real time-applications where a timely response to events is critical. One of the distinguishing features of an RTOS is its ability to provide precise timing guarantees. There are two categories of RTOS: hard RTOS and soft RTOS, each offering different levels of assurance. A hard RTOS ensures that critical tasks consistently meet their deadlines, thereby minimizing the potential for severe consequences in the event of a missed deadline. On the other hand, a soft RTOS provides reliable responses for the majority of tasks, although occasional deadline misses may not have significant consequences [19].

- Linux-based Operating Systems
  Linux-based operating systems are operating systems built upon the Linux Kernel, which is an open-source operating system that follows Unix-like architecture created by Linux Torvalds. Those operating systems inherit the principles of Linux and can be run on various devices and platforms because they are open-source operating systems. Some of them are Ubuntu, Buildroot, Yocto Project, OpenEmbedded, etc.

- Microcontroller Operating Systems
  Microcontroller operating systems (MCOS) are specifically designed for microcontrollers, which are small and low-power integrated circuits that contain a processor core, memory, and peripheral interfaces on a single chip.

2.8.2 Yocto Project

The Yocto Project is a collaborative open-source initiative that offers a comprehensive set of templates, tools, and methodologies to facilitate the creation of customized Linux-based systems for embedded products, irrespective of the underlying hardware architecture [20]. It offers a collection of recipes, configuration values, and dependencies that enable the creation of a tailored Linux run-time image. Unlike desktop distributions, Yocto allows to choose packaging formats and init systems based on their specific requirements. It supports deb, rpm, ipk, or tar package formats by default and provides flexibility to add custom formats. Similarly, selecting between sysvinit or system init systems can be easily configured. A Yocto build initially constructs native utilities the build system requires to minimize dependencies on the host OS and ensure a known set of package versions. It then builds a cross-compilation environment before proceeding to build binaries for the target platform. The build process, managed by the bitbake [1] executive, involves multiple phases (fetch, compile, install, package) and necessitates the specification of run-time and build-time dependencies. Parallel execution of tasks is determined by host system resources and task interdependencies [22].

2.8.3 PetaLinux

PetaLinux is a Software Development Kit(SDK) specifically designed for embedded Linux systems deployed on FPGA-based system-on-chip designs. The PetaLinux tool includes essential components such as the Yocto Extensible SDK (eSDK), XSCT (Xilinx Software Command-Line Tool), toolchains, and Petalinux Command-Line Interface (CLI) tools that help developers to effectively build, customize, and validate embedded Linux systems [23]. PetaLinux is built on top of the Xilinx Yocto Project, so it leverages the power of Yocto build systems that enable efficient construction of Linux images with a high degree of customization. The PetaLinux tool incorporates a CLI that offers a range of commands for effective management and customization of embedded Linux systems. One of the commands, `petalinux-create` allows users to create new PetaLinux projects, while `petalinux-config` enables customization of specific projects by adjusting their configurations. The `petalinux-build` command facilitates the building process of either a specified component or the entire embedded Linux system. Additionally, the `petalinux-util` command provides supplementary services to support various PetaLinux workflows. In order to package the PetaLinux projects into deployable formats the `petalinux-package` command can be used, and the `petalinux-upgrade` command allows for workspace upgrades. The `petalinux-devtool` command employs the Yocto devtool for building, testing, and packaging software. Lastly, the `petalinux-boot` command is utilized to boot MicroBlaze™ CPU, Zynq® devices, Versal® ACAP, and Zynq® UltraScale+™ MPSoC with PetaLinux images through JTAG/QEMU.

[1] BitBake is a main component of Yocto Project that is used to build images [21]

2.8.4 Device Drivers

Device drivers are software programs that serve as an intermediate level between the operating system and the hardware devices [24]. The purpose of the device driver is to make sure that the operating system can interact with the hardware device when it is connected. In Linux, device drivers are represented by loadable modules that are loaded during the boot process and facilitate the communication between OS and the hardware.

Device drivers are responsible for managing the control and status registers (CSRs) of hardware controllers, which can differ between devices. By centralizing the code for managing hardware controllers, Linux avoids duplication in applications. Linux's device drivers are shared libraries that contain low-level hardware handling routines. They handle the specific operations on the devices they manage, abstracting the complexities of device handling from applications. Linux supports three main types of hardware devices: character devices, block devices, and network devices. Through the utilization of device drivers, Linux provides a unified and efficient approach to interacting with these hardware devices. In order to provide consistent and streamlined device management in Linux, the operating system is divided into two spaces: kernel and user space. In user space the processor and user applications run, and on the other hand, the kernel space is a privileged part that contains the kernel. This is why the device driver resides in kernel space so that they can provide the necessary interface for the user applications, which make system calls to the kernel, that then communicates with the respective device driver to perform operations on hardware.

2.8.5 Device Tree Overlay (DTO)

As discussed in the previous subsection the device tree is a data structure constructed as a hardware description of FPGA components by providing a hierarchical representation of hardware components. In some cases, during the reconfiguration process, it is necessary to change the device tree in order to include the new hardware components that are added or removed. Specifically, the device tree overlay is the mechanism that allows modification of the device tree in order to support dynamic reconfiguration. The device tree is represented by a textual format that consists of nodes and properties. The nodes include their name and their unit address as illustrated below:

```
label: node_name@node_address {
    /* properties */
};
```

The label is used during the compilation process, and it can be used as a reference in other device nodes. The node name serves as the identifier and the node address represents the base address of the device register. Properties in the device node are used to provide specific information and configuration parameters for the corresponding hardware component. As the name suggests the device tree is a tree structure, where the root node is the starting point of the tree marked by "/" and it contains all other nodes. The basic format of the device tree is as below:

```
 /dts-v1/;

/ {
      label: node_name@node_address {
          reg = <node_address>;
          property-1 = "";
          property-2 = <>;
          property-3 = 1;
      };

      // rest of the devices
};
```

~

The Device Tree Overlay is a method that allows the dynamic modification of the device tree in a live Linux kernel environment. It enables the insertion of device tree fragments, which can add or remove hardware components or modify their properties, without the need to rebuild the entire device tree. The main purpose of the DTO is to modify the kernel's live tree by modifying it in a way that reflects the required changes [25].

The device tree overlay is defined in a `.dts` file, which follows a specific format. In the device tree overlay, the _overlay_ node contains the fragments that will be added to the device tree. Each component is defined using the `fragment@X` syntax, where X is a unique identifier. The target/target-path property specifies the location in the device tree where the fragments should be added.

When using PetaLinux, there are some considerations to keep in mind. PetaLinux creates the device tree by combining the Board Support Package (BSP) and Vivado export files into a single `system.dts` file during the build process. Each hardware component has its own device tree, specified in `.dtsi` files, which are then integrated into the final `system.dtb` file by PetaLinux. Configuring the PL in PetaLinux involves modifying the appropriate `.dtsi` file for the target hardware component. The PetaLinux project should already be created and in use on the board before making these modifications.

2.8.6 Integration of Embedded Operating Systems with Dynamic Partial Reconfiguration

Another important aspect of the partial reconfiguration is the integration with an embedded operating system which includes considerations and challenges. Firstly, hardware abstraction is an important aspect since it establishes a unified interface between the embedded operating system and the reconfigurable hardware components. This abstraction can be used to facilitate high-level programming. This abstraction level can be ensured by implementing Application Programming Interfaces (APIs). Secondly, the device driver can be customized to facilitate the dynamic loading and unloading of IP (Intellectual Property) components, the reconfiguration process, and the communication between the reconfigurable modules and operating system. Another important aspect is the implementation of fault tolerance mechanisms and error handling strategies in order to detect and recover from errors that may occur during the reconfiguration process. In conclusion, the integration of embedded operating systems with dynamic partial reconfiguration enhances system flexibility, scalability, and adaptability by leveraging the capabilities of the operating system while utilizing reconfigurable hardware modules.

2.9 Boot and Configuration

FPGAs offer a compelling solution for high-performance computing in space applications, particularly in the realm of data processing. By employing reconfigurable FPGA devices that incorporate processors, a versatile hardware and software co-design architecture can be achieved. This design approach enables the updating of processing algorithms during missions, proving particularly advantageous for space endeavors demanding intensive onboard computations, such as the HYPSO mission for hyperspectral images.

2.9.1 Boot Image

Figure 2.10 shows a basic format for a boot image, consisting of several headers: Boot Header, headers required by the First Stage Boot Loader (FSBL), PMU Firmware Image, and FSBL image. The Boot Header is a plain text header that provides information about the FSBL location, length,

and system initialization details. The Partition Header is an array of structures that contains information for each partition, such as partition size, address, and load address in RAM.

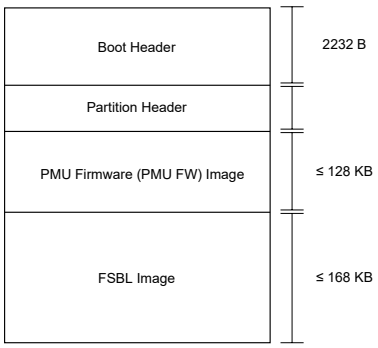


Figure 2.10: Simple form of Boot Image with FSBL and PMU Firmware

Source: [12]

The PMU Firmware, which contains a MicroBlaze processor, loads executable code from 32 KB ROM and 128 KB RAM into flat memory space and controls the power-up, reset, and resource monitoring within the system. The FSBL is responsible for configuring the FPGA with hardware bitstream and loading the Operating System (OS) Image Standalone (SA) Image or Second Stage Boot loader Image from the non-volatile memory. It handles the booting process, starting with PS block initialization, locating the boot image in storage media, encrypting or decrypting and moving the partition to its destination, and finally handing off execution. U-Boot acts as a secondary boot loader that loads the Linux and configures the rest of the peripherals in the processing system based on board configuration.

2.9.2 Boot Process Stages

While the boot process for the Zynq-7000 device follows a similar pattern to most Xilinx devices, in this discussion, this paragraph focuses on the boot process of the UltraScale+ devices, specifically the Zynq UltraScale+ MPSoC. The reason for this choice is to highlight the additional complexities and components involved in the boot process of the UltraScale+ architecture, providing a comprehensive understanding of its unique features. The Zynq UltraScale+ MPSoC uses two components, the Platform Management Unit (PMU) and the Configuration Security Unit (CSU), to manage the boot process. The PMU powers up and down peripherals and processors, generates and handles system reset signals, and power sequences the different domains in the platform. The CSU manages secure and non-secure system-level configurations.

To boot Linux on the MPSoC, three files are required: `BOOT.BIN`, `image.ub`, and `boot.src`. BOOT.BIN contains PMU firmware, First and Second Stage BootLoader, ARM Trusted firmware-A, and can optionally contain the bitstream file. Image.ub is U-Boot's Flattened Image Tree (FIT), containing the Linux Image and device tree. Finally, boot.src is the U-Boot Boot script. Trusted firmware-A (TF-A) is an open-source reference implementation of the ARM Trusted Firmware (ATF) specification. TF-A ensures a secure and trusted boot process for Arm-based devices by establishing a trusted execution environment, providing security services, and maintaining the integrity and confidentiality of system operations.

The boot process has three stages: pre-configuration, configuration, and post-configuration. During the pre-configuration stage, the PMU ROM code is executed after the power-on reset (POR) and initializes the system. During the configuration stage, the CSU ROM code executes and performs On-Chip Memory (OCM) initialization, determines boot mode, interprets the boot image header, and loads the FSBL from the boot image in the OCM for execution by the selected processor. The CSU also loads PMU FW from the boot image into PMU RAM, however, this step is optional. During the post-configuration stage, the CSU monitors tamper signals from the system and detect if the secure boot process is correct. The CSU also provides hardware services like file authentication and decryption, secure key management and storage, and PL configuration via the PCAP interface.

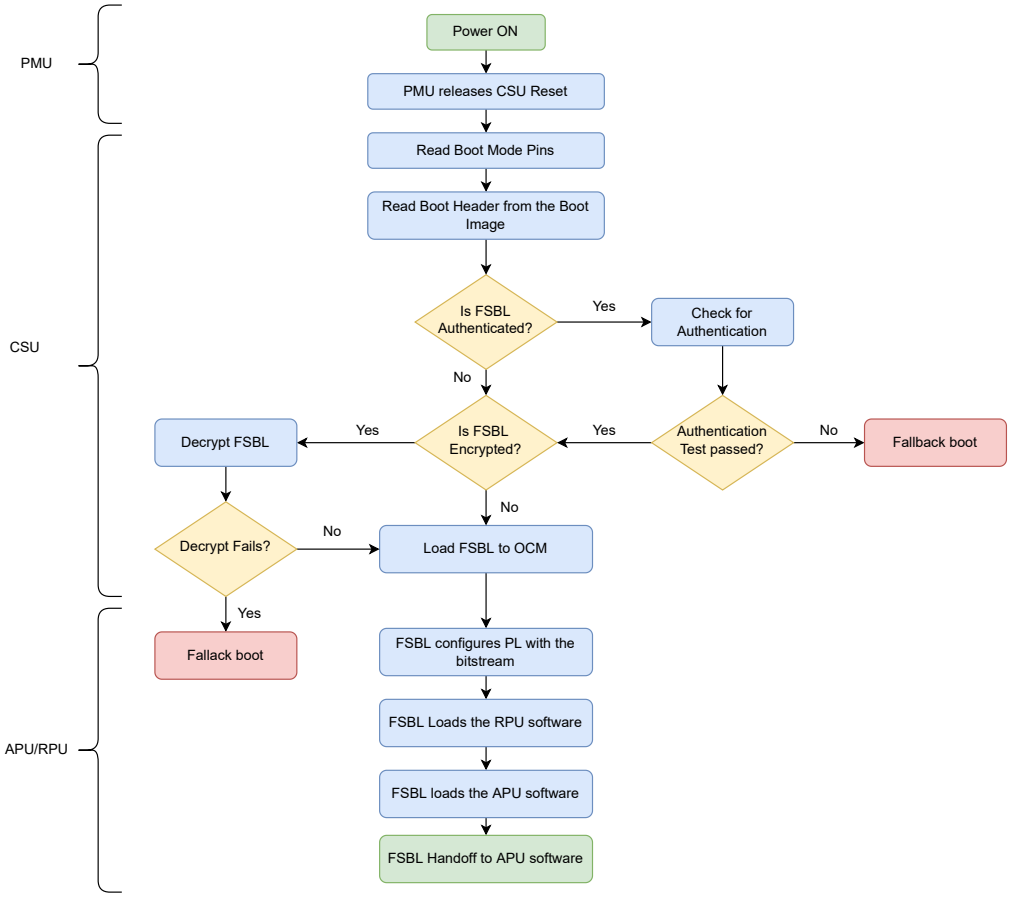


Figure 2.11: Detailed Boot Flow Example

Source: [26]

Finally, the FSBL handles other boot operations such as loading the PMU Firmware (if CSU Boot Room didn't load it previously), bitstream for the PL, and software executable for the APU and RPU (Raid Processing Unit) processors.

2.10 FPGA Manager

The FPGA manager is a tool that enables the loading and programming of bitstreams onto an FPGA within a Linux environment by providing an interface for users to interact with the FPGA. It includes a set of functions specifically designed for programming the FPGA with new configurations. One of the main features of FPGA Manager is the support for partial reconfiguration of FPGA. The FPGA Manager consists of three key entities: Managers, Bridges, and Regions. Managers handle FPGA programming, Bridges separate regions during reconfiguration, and Regions represent programmable parts of the FPGA. The FPGA Manager provides functionalities such as full and partial bitstream loading, encrypted and authenticated loading, and read-back of configuration registers and the bitstream itself [26]. This section explores the FPGA Manager in detail and its role in facilitating FPGA reconfiguration.

2.10.1 Overview

The FPGA Manager subsystem consists of several components that work together to enable FPGA programming. It follows a multilayered architecture where the upper layer provides a platform-agnostic API, hiding platform-specific details and the lower layer consists of an FPGA driver

implementing the operations. The interface between layers includes operations for loading bitstreams onto the FPGA. Flags are used by the FPGA Manager to specify properties like support for partial reconfiguration, external configuration, encryption, and compression [27, 28]. Figure 2.12 illustrates the functionality of FPGA Manager to perform full reconfiguration

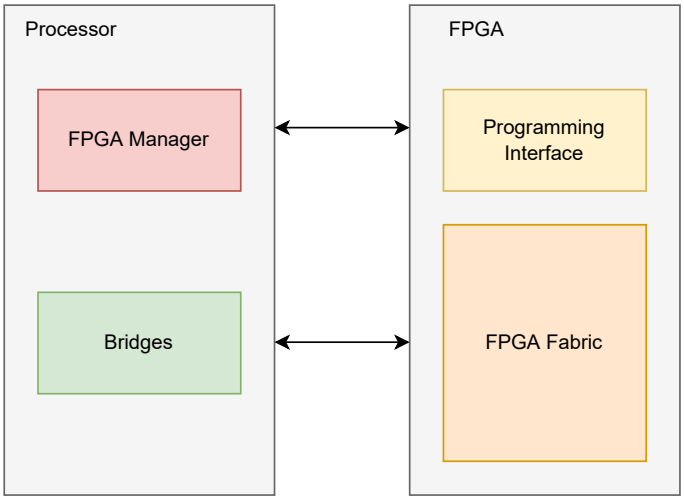


Figure 2.12: System organization in case of full reconfiguration using FPGA Manager

The key entities within the FPGA Manager include:

- **Managers**: These entities handle the programming process for the FPGA. They encapsulate the necessary logic and functionality to load different types of bitstreams onto the FPGA. The Managers interact with other components to facilitate the programming process effectively.
- **Bridges**: The FPGA Manager employs Bridges to isolate different regions within the FPGA during reconfiguration. Their purpose is to prevent irrelevant signals from propagating outside the intended region, ensuring safe and controlled reconfiguration.
- **Regions**: FPGA Regions represent specific portions of the FPGA that can be programmed independently. Depending on the reconfiguration requirements, the FPGA Manager can target either the entire FPGA logic or specific partial regions for programming. Regions are connected to the FPGA Manager and their associated Bridges.

Figure 2.13 provides a high-level overview of how these components communicate.

The FPGA is divided into two main sections: the PS and the PL. The PS contains the programming interface that communicates with the FPGA Manager, enabling the processor to control the reconfiguration process. On the other hand, the PL is divided into regions, where each region represents a distinct portion of the PL that can be independently reconfigured. As illustrated in Figure 2.13, these regions communicate with their respective bridge in the PL.

A bridge acts as an interface between a region and the rest of the system, providing the necessary communication channels and protocols for controlling the reconfiguration process. It ensures that the reconfiguring section is decoupled from the other logic, maintaining isolation. This component includes three main sub-components: configuration memory, configuration access port (CAP), and the configuration controller. The configuration memory stores the configuration bitstreams for the static region and the reconfiguration regions. The CAP is the interface that allows writing the configuration data into the memory, and finally, the configuration controllers manage the reconfiguration process by coordinating the loading of the configuration data and ensuring the correct execution of the reconfiguration steps.

During partial reconfiguration, the FPGA is reprogrammed using a base image to create the slots represented by the regions shown in Figure 2.13. Bridges are employed to decouple the reconfiguring

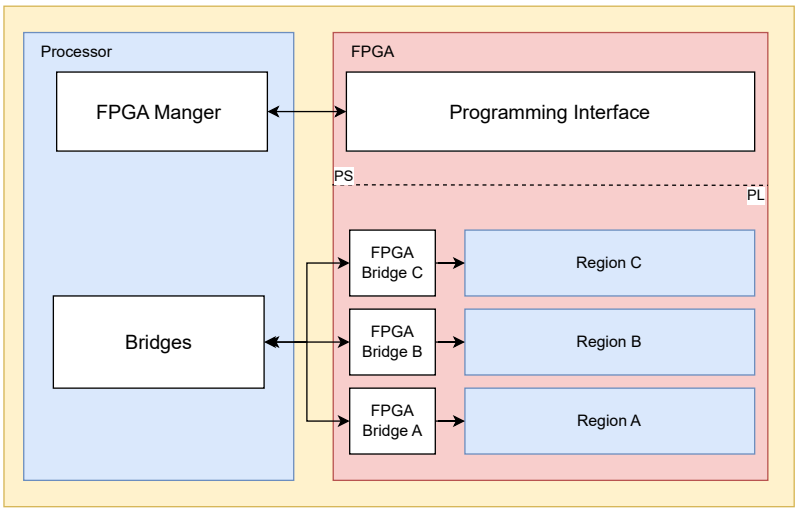


Figure 2.13: Partial Reconfiguration using FPGA Manager

Source: [29]

slot from the rest of the logic, ensuring proper isolation and avoiding unintended behavior. To facilitate communication between the bridges in the PL and the bridge component in the PS, all three bridges are connected to the main API. This connection allows the processor to access and control the PL regions and bridges throughout the reconfiguration process.

Overall, bridges play a crucial role in enabling communication and managing data flow between the regions in the PL and the rest of the system during partial reconfiguration. They provide the necessary interfaces and isolation to ensure seamless reconfiguration while maintaining the integrity of the overall FPGA design [29].

2.10.2 Functionalities

The FPGA Manager offers a range of functionalities to accommodate various programming scenarios. Some of the main features provided by the FPGA Manager include:

- Full bitstream loading: It supports loading complete bitstreams onto the FPGA and thus allowing the full reconfiguration.
- Partial bitstream loading: In scenarios where only specific regions of the FPGA need to be reconfigured, the FPGA Manager allows for the loading of partial bitstreams.
- Encrypted full/partial bitstream loading: It supports the loading of the encrypted bitstream, for both full or partial reconfiguration, in order to enhance security.
- Authenticated full/partial bitstream loading: It provides functionality for loading authenticated bitstreams, ensuring the integrity and authenticity of the programming data during the reconfiguration process.
- Read-back of configuration registers: Users can read back the configuration register values from the FPGA using the FPGA Manager. This feature helps to debug and to monitor the FPGA state after reconfiguration.
- Read-back of the bitstream: It enables users to read back the bitstream from the FPGA as a mechanism for verifying and analyzing the programmed configuration.

All those functionalities that FPGA Manager provides, make it a powerful tool for reconfiguring the FPGA within a Linux environment.

2.10.3 FPGA Manager Execution Flow

In order to successfully reconfigure the FPGA using the FPGA Manager, it is important to understand its execution flow, which involves the following steps:

1. **Write Initialization (.write_init)**: This function prepares the FPGA to receive the image data by allocating the required memory and initializing the necessary data structures where initially, a buffer of size equal to .initial_header_size bytes is provided. During this phase, the FPGA is prepared to receive the configuration data and in order to facilitate this process this buffer is allocated in memory. The buffer serves as a storage space where FPGA Manager can organize the data before transferring it to the FPGA. The allocation of the buffer and the initialization of the data structures, are necessary to ensure that the required memory resources are available and properly configured to receive the image data.
2. **Write Operation (.write or .write_seg)**: The .write function is responsible for writing the buffer that contains the entire FPGA image onto the FPGA. This function can be called multiple times if the image is too large to fit in a single buffer. Alternatively, the .write_seg function can be used, which takes a scatter list as input. A scatter list is a list of array pointers and lengths used by DMA to transfer several buffers in a single operation [27].
3. **Write Completion (.write_complete)**: After the image data has been written, the .write_complete function is called to put the FPGA into operational mode, allowing it to utilize the newly programmed configuration. Additionally, the .state function can be used to determine the current state of the FPGA.

During execution, the FPGA Manager performs several tasks. It includes the allocation of memory and utilization of the Embedded Energy Management Interface (EEMI) API to load the bitstream onto the FPGA. Subsequently, the FPGA Manager waits for a response from Trusted Firmware-A (TF-A). Upon receiving the response, the FPGA Manager passes it from the FPGA core layer to the application. To initialize the Programmable Control and other necessary hardware components, the FPGA Manager relies on the `xilfpga` library [26].

2.11 Design Constraints

Design constraints have an important role in the design and implementation of FPGA-based systems since they ensure that the systems meet the target performance, timing requirements, and power requirements but they can also define the pin requirements. Constraints are utilized to exert influence over the implementation tools of FPGA design, such as the synthesizer and place-and-route tools. Their purpose is to guide and shape the behavior and decisions made by these tools during the design process. This section will describe the main design constraints that are used during FPGA design.

- Timing Constraints
  Timing constraints are used to specify the timing characteristics of the design and are usually defined in terms of clock periods that are essential in design operation. Timing constraints in FPGA designs can be broadly classified into two categories: global and path-specific constraints. Global timing constraints encompass all paths within the logic design, while path-specific constraints are tailored to address specific paths [30]
- Power Constraints
  Power constraints are used to optimize the power consumption of the FPGA design. These

constraints encompass various aspects, including maximum power consumption limits, average power limits, or specific power profiles that the system must adhere to. By imposing these constraints, designers are guided in making architectural decisions, implementing algorithmic optimizations, and incorporating low-power design techniques throughout the design process.

- I/O Constraints
  I/O (Input/Output) constraints, referred also as pin assignments, define the characteristics and requirements of the system's interfaces with the external devices. They are used to assign a signal to a specific I/O pin but can be also used to specify parameters such as voltage levels, signals timing, data rate limits, etc. So they ensure the proper communication of the FPGA with the external devices. Pin assignment in FPGA design plays a crucial role in determining the location of resources within the device. When it comes to pin assignment there are several factors that should be taken into consideration. The tools generally aim to spread functionality across available resources to avoid routing congestion.

- Area Constraints and Floorplanning
  Area constraints have also an important role in the FPGA design since they provide guidance and control over the placement of the design elements. By defining the allowable region for placement, these constraints significantly reduce the implementation time of place-and-route tools. This process, known as floorplanning, involves strategically arranging multiple design blocks on the target FPGA architecture. By applying area constraints, the tools can avoid the time-consuming search for suitable locations for block elements. Early floorplanning can yield benefits if approached with a comprehensive understanding of the design and target architecture, along with ample design margin. Floorplanning not only facilitates the placement of specific design elements, such as block memories but also enables the optimization of data flow within the FPGA. By carefully considering the interfaces between design blocks and their intended implementation locations, interleaved logic from multiple blocks and distributed memory implementations can be effectively accommodated [30]. As it will be seen in the later sections, floorplanning plays a crucial role in partial reconfiguration flow.

2.12 FPGA Cores and Primitives

FPGA cores and primitives are fundamental building blocks used in the implementation of efficient and optimized algorithms for data processing in embedded systems and space applications. These cores and primitives provide essential functionality and enable the development of complex systems on FPGA platforms. This section focuses on two specific algorithms: CubeDMA and CCSDS-123 compression, which have been implemented by students from the HYPSO team as part of their master's thesis projects over the years, to facilitate the efficient processing and compression of HSI cube in the HYPSO mission.

2.12.1 CubeDMA core

The CubeDMA core is a specialized DMA (Direct Memory Access) core designed for use in hyperspectral imaging applications. It is capable of efficiently processing data in various streaming orders, including BSQ (Band Sequential), BIP (Band Interleaved by Pixel), and other formats commonly used in hyperspectral data processing. The CubeDMA core plays a crucial role in this process by facilitating high-speed data transfer between different components of the system, such as the image sensor, memory, and processing units. The CubeDMA algorithm implemented within the core optimizes data transfer and processing, resulting in improved overall system performance. By efficiently managing the movement of data between memory and processing units, the CubeDMA core minimizes latency and maximizes the utilization of system resources. The core's ability to handle different streaming orders, such as BSQ and BIP, allows it to adapt to the specific data formats commonly encountered in hyperspectral imaging applications. This flexibility ensures compatibility with different processing algorithms and data organization schemes, making the CubeDMA core a versatile and valuable component in HYPSO mission but also in hyperspectral image applications.

Direct Memory Access (DMA)

DMA is a hardware module that provides high-speed memory transfer between memory and peripherals, without involving the CPU by helping thus offloading data transfers from the CPU and improving the system performance since the CPU is able to perform other tasks while the DMA performs the data transfers. The DMA module is controlled by the DMA controller that sets up and manages the data transfers and can act as both a bus master and bus slave. In its role as a bus master, the DMA controller manages communication with the memory controller and performs arbitration for the bus. Meanwhile, when functioning as a slave, the DMA controller responds to requests from bus masters such as the processor to establish memory transfers. This subsection describes the CubeDMA, which is implemented by previous students of the HYPSO mission.

Overview of CubeDMA core

The implemented CubeDMA core consists of Memory Map to Stream (MM2S) and Stream Memory Map (S2MM) channels, that are configured using a common register interface. The MM2S channel reads data from memory and streams it into the accelerator. The S2MM channel, on the other hand, receives a data stream from the accelerator and writes the incoming stream of data sequentially in the memory [7].

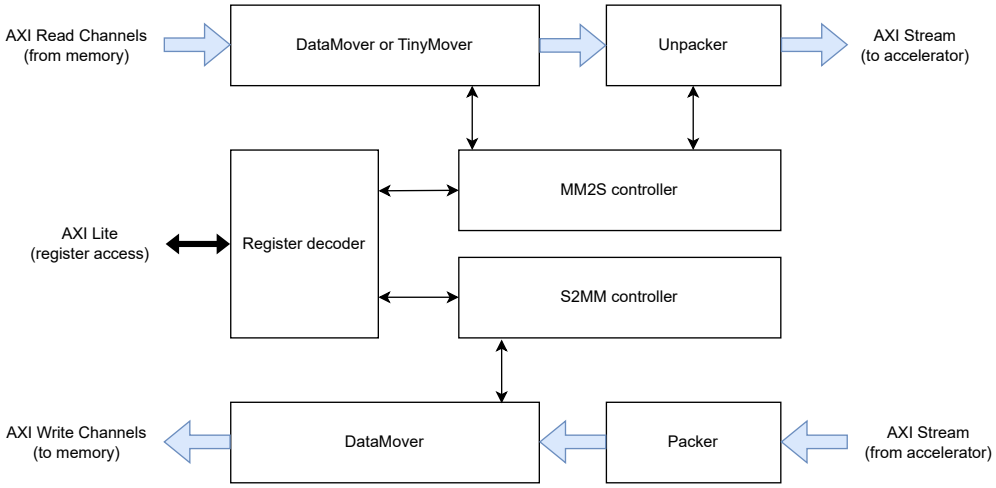


Figure 2.14: Overview of CubeDMA core

Source: [7]

Figure 2.14 provides an overview of the Cube DMA system. The CubeDMA can handle component sizes that are not multiples of bytes. In the MM2S channel, packed data from memory is unpacked into separate components and streamed to the accelerator. In the S2MM channel, components from the accelerator are packed into 64-bit words before being stored in memory.

The MM2S channel of the CubeDMA provides flexibility for both BIP-ordered and BSQ-ordered transfers, which can be selected during run-time through register configuration. In BSQ mode, components from different planes can be streamed in parallel, meaning that components from the same pixel but in different planes are output simultaneously in each transfer beat. Similarly, in BIP mode, multiple components from the same pixel can be transferred in parallel. The number of planes (for BSQ) or pixel components (for BIP) to transfer in parallel is determined by generic parameters when setting up the DMA core.

On the other hand, the S2MM channel of the CubeDMA writes incoming stream data sequentially to memory. It supports data words of varying sizes and collects the data into 64-bit packets that are stored in memory. The S2MM channel continues storing data until the TLAST signal, which is used to indicate the end of the transfer, of the incoming stream is asserted. The number of bytes received can subsequently be read from the register interface [31].

As illustrated in 2.14, the MM2S (Memory-Mapped to Stream) channel consists of three key components: the DataMover/TinyMover, unpacker, and controller. The DataMover/TinyMover is an advanced IP core provided by Xilinx that specializes in high-speed data transfers that with its support for interfaces like AXI and AXI4-Stream, it ensures optimal performance and minimal latency. In the CubeDMA module, the DataMover IP is utilized for BIP transfers, while the TinyMover is specifically designed for BSQ transfers, both of which have been modified and adapted for seamless integration with the CubeDMA module [32]. The Unpacker component plays a crucial role in aligning and correcting word sizes, ensuring that the transferred data is accurately processed. Lastly, the controller oversees and manages the operations of the DataMover/TinyMover and Unpacker, generating a control stream that is vital for the proper functioning of the CubeDMA module. Together, these components work harmoniously to enable efficient and reliable data transfer within the CubeDMA module.

The S2MM channel operates in a manner analogous to the MM2S channel but with reversed functionality. In this channel, data is received from a stream and undergoes packing to ensure proper word sizes before being forwarded to the Datamover. The DataMover then sequentially writes the packed data to memory using the AXI bus. Throughout this process, the controller assumes the responsibility of managing and coordinating the operations of the DataMover and Packer modules [32].

CubeDMA Operation

In summary, CubeDMA is used to perform direct memory accesses, offloading the CPU from those operations. The module consists of memory-mapped registers, such as control registers, status registers, base address and cube dimension, and row size registers. The control register enables the initiation of the DMA engine for starting data transfers. The status register indicates the completion of a transfer or any triggered errors. To ensure proper access to the DDR memory shared by the DMA and CPU, a base address register defines the starting point for DMA reads. The cube dimension register determines the number of addresses to be read, representing the size of the hyperspectral image in terms of width, height, and depth (X, Y, and Z axes) where the row size, obtained by multiplying X and Z, is crucial for the CubeDMA to operate correctly.

2.12.2 Compression Algorithm

Compression schemes can be divided into two major classes: lossless compression schemes and lossy compression schemes. The difference between the two is that data compression using lossless compression schemes can be recovered to the original data, but using lossy compression this cannot be done since it introduces some loss of information [33]. The compression algorithm implemented in the HYPSO mission is based on the CCSDS-123 algorithm. The CCSDS-123 is a compression algorithm standardized by the Consultative Committee for Space Data Systems (CCSDS) designed for lossless compression of hyperspectral images. The lossless source coding technique helps to remove redundancy from the input data source while preserving the data accuracy [34]. In this case during decoding, the original hyperspectral image can be reconstructed by restoring the removed redundancy. The following description of the algorithm is based on the CCSDS Recommended Standard given in [35].

CCSDS-123 Algorithm Overview

The high-level schematic of the CCSDS-123 algorithm is shown in Figure 2.15. The compressor takes as input, a three-dimensional image, represented as an array of integer sample values $s_{z,y,x}$, where x and y represent the spatial dimensions and z represents the spectral dimension. The structure of the compressed image is composed of the header that encodes the image and compression parameters, followed by a body that encodes the image samples. As illustrated in Figure 2.15 the compressor is composed of two main stages: prediction and encoding.

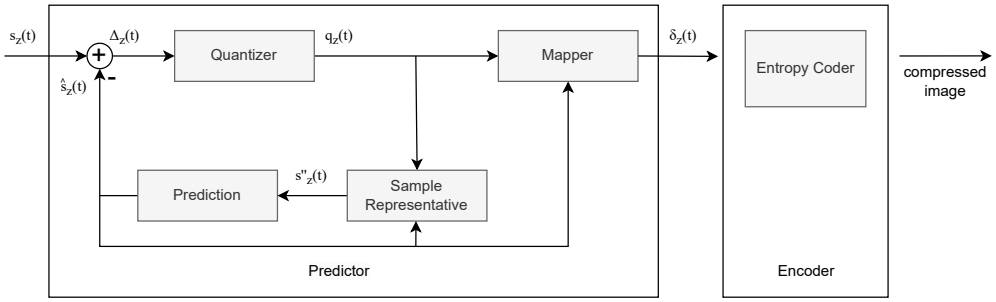


Figure 2.15: CCSDS-123 compressor Schematic

Source: [35]

During the prediction stage, the module calculates estimates for each component by analyzing preceding components that are located nearby in the cube and have similar spectral characteristics. The difference between the actual component values and these estimates which are called residuals is transformed into variable-length code words that will serve as input for the encoding stage.

In order to describe the CCSDS-123 algorithm is important to introduce the terminology and definitions that will be used in the upcoming sections.

- Samples
  Samples refer to individual components in an HSI cube that can be unsigned or signed integers with a bit resolution given by the parameter D and have lower and upper-value limits denoted by s_{min} and s_{max}, respectively.
- Cube Size
  The size of an HSI cube is denoted by the symbols N_X, N_Y, and N_Z, and samples within the cube can be addressed by their x, y, and z coordinates in the form $s_{z,y,x}$, or by the index $t = y \cdot N_X + x$.

Predictor

The prediction component in the CCSDS-123 algorithm is responsible for performing the prediction process based on the input hyperspectral data. Before describing the prediction process it is important to define what a sample is.

A sample is an individual component in an HSI cube and is denoted by s. For example $s_{z,y,x}$ referees to the sample in the respective cube dimensions denoted by x, y, and z. Also, in order to define the bit resolution of the sample a D parameter is used which takes values from 2 to 16. The stage of prediction calculates approximations of individual parts using a 3D space sample surrounding the analyzed part. The dissimilarity between the approximation and the factual value of the part, known as the prediction residual, is subsequently encoded using the entropy encoder. The bitstream initially encodes various parameters such as cube size, output word size, scanning order, sample type, dynamic range, sub-frame interleaving depth, and entropy encoder type.

Encoder

The entropy encoder in the CCSDS-123 algorithm is responsible for compressing the prediction error data generated by the predictor step. This is achieved by assigning shorter codes to frequently occurring prediction error values and longer codes to less frequently occurring values. The entropy encoder uses variable-length coding, which means that each prediction error value is assigned a variable-length code, where more probable values are assigned shorter codes and less probable values are assigned longer codes.

2.12.3 Pipeline Implementation

Figure 2.16 displays the structure of one pipeline in a multiple pipeline implementation of the CCSDS-123 algorithm. The diagram illustrates the flow of data through the design, with thick gray lines indicating the data path. The predictor stage consists of the first four blocks, while the encoding stage is represented by the last block. In the predictor stage, local differences are computed, multiplied by a weight vector, and passed to the predictor block for calculating predicted values. Mapped prediction residuals are then computed and sent to the encoder stage.

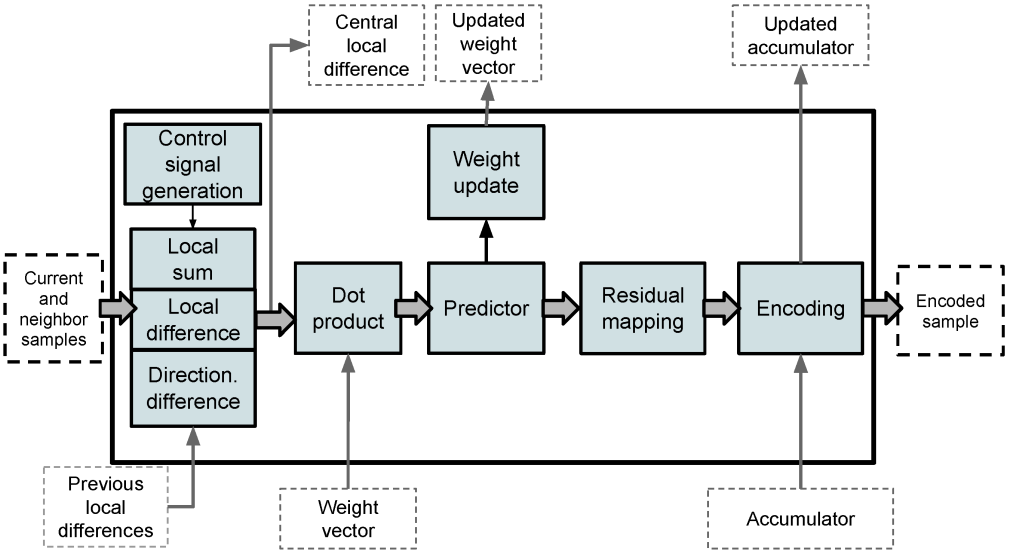


Figure 2.16: CCSDS-123 pipeline implementation

Source: [7]

In a multiple-pipelined implementation, multiple pipelines are used to process samples concurrently. The pixels enter the sample delay block where they are held until all the samples for a specific neighborhood have arrived, following a BIP ordering. Since the central local differences remain consistent across pipelines, they are stored in shared storage accessible by all pipelines. The same applies to weight and accumulator values. The packer component organizes the data from the pipelined implementation into specified-length words and outputs them. Furthermore, the design can be configured to process samples "on the fly," meaning that samples are streamed directly to the compressor without any input data control, which can be suitable when the data rate is predetermined or when synchronization is not feasible, for example when connected directly to a camera sensor.

2.13 Summary

This section provided background information that was necessary to understand and analyze before starting with the partial reconfiguration process. An important part of this chapter was focused on the FPGA architecture and structure in order to define in detail all the necessary concepts that play an important role in a successful reconfiguration. Furthermore, a description of the FPGA design flow was given, highlighting the changes that are necessary to be done in each phase. This chapter also provided an overview of FPGA cores and primitives, highlighting their importance in implementing optimized algorithms for data processing in embedded systems and for the HYPSO mission. It focuses on two specific implementations: the CubeDMA core and the CCSDS-123 compression algorithm. The CubeDMA core facilitates high-speed data transfer for HSI image cubes, while the CCSDS-123 algorithm is a lossless compression algorithm designed for hyperspectral images.

CHAPTER 3

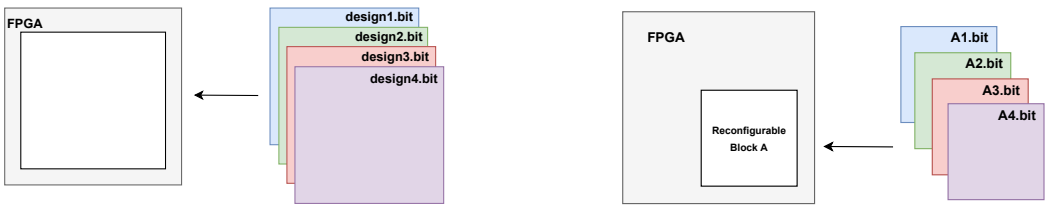
Programmable Logic Reconfiguration

As mentioned in the previous chapters, one of the main characteristics of FPGAs that makes them popular is their ability to reconfigure hardware at runtime. However, full reconfiguration of an FPGA can be time-consuming and resource-intensive, particularly for larger designs. Partial reconfiguration (PR) offers a solution to this problem by allowing specific portions of the FPGA to be reconfigured independently, while the rest of the system continues to operate. This section will describe how the PL reconfiguration is performed, by also making a literature review of proposed controllers.

3.1 Types of Configuration

There are three main types of FPGA configuration: static reconfiguration, static partial reconfiguration, and dynamic partial reconfiguration (DPR). Static reconfiguration involves loading the entire FPGA with a full bitstream, which requires stopping the execution of the FPGA. Static partial reconfiguration allows for a portion of the FPGA to be reconfigured while the rest of the system continues to run, but the execution is paused to load the partial bitstream. Dynamic partial reconfiguration, on the other hand, enables part of the FPGA to be reconfigured while the rest of the system continues to operate without interruption. This technique is useful for applications that require continuous operation and the ability to update or add functionality to the FPGA design without affecting the entire logic. Figure 3.1 illustrates the difference between using full and partial bitstreams for FPGA configuration.

FPGA configuration types are important considerations for designers when implementing hardware designs. The choice of configuration method depends on the specific requirements and constraints of the application. For example, static reconfiguration is suitable for applications that do not require continuous operation, while static partial reconfiguration and DPR are useful for applications that require partial reconfiguration or on-the-fly updates [2].



(a) Full configuration using full bitstreams

(b) Partial configuration using partial bitstreams

Figure 3.1: Illustration of FPGA configuration using full and partial bitstreams

3.2 Partial Reconfiguration in FPGAs

The motivation behind partial reconfiguration stems from several key factors, including area optimization, security and robustness, and shorter configuration time. One of the main benefits of PR is area optimization. By dynamically swapping out unused logic blocks, designers can reduce the size of the FPGA logic required to implement a given function, resulting in less power consumption and reduced cost. This is particularly useful when only a portion of the FPGA is being utilized at any given time, as it allows for more efficient use of resources.

Another key motivation for PR is security and robustness. With the ability to dynamically swap out logic blocks, designers can implement new techniques related to design security. For example, if a PL accelerator for encryption and decryption is not required at all times, its area can be configured with dummy logic to ensure security. Additionally, PR can help to improve the robustness of FPGA-based systems by allowing faulty blocks to be swapped out and replaced without the need for a full system reconfiguration.

Finally, PR offers a shorter configuration time compared to full reconfiguration. Since only a portion of the bitstream needs to be modified, the total reconfiguration time is shortened, which can be particularly beneficial for time-critical applications.

3.2.1 Terminology

In order to describe the concept of partial reconfiguration it is important to understand some terms that will be used throughout this section. The below terms are defined based on Xilinx Partial Reconfiguration User Gide in [36].

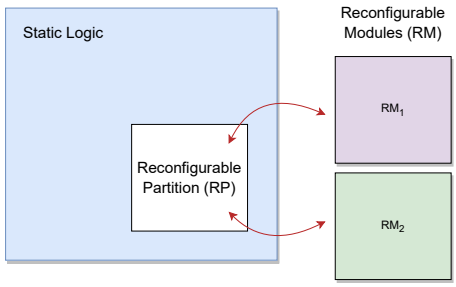


Figure 3.2: Partial Reconfiguration Notations

- Partitioning
  Partitioning is a way of dividing the design into logical sections, defined by the user at a hierarchical level, with the aim of reusing these sections.
- Reconfigurable Partition
  Reconfigurable Partition (RP) is a characteristic assigned to an instance, indicating that it can be reconfigured. It refers to the physical or logical division of an FPGA into separate regions or sections, as illustrated in Figure 3.2.
- Reconfigurable Module
  A Reconfigurable Module (RM) is a description or a netlist written in HDL, which is implemented when instantiated by an instance that has the Reconfigurable Partition attribute.
- Static Logic
  Static logic refers to logical components that are not included in a Reconfigurable Partition, that are always active, and cannot be partially reconfigured while the RPs are being reconfigured.

- Configuration
  Configuration in a PR project refers to a complete design that consists of one Reconfigurable Module for each Reconfigurable Partition. Multiple configurations can exist within the project, with each configuration generating both a full and a partial BIT file for every RM. The combination of static and each RM, forming a configuration, is illustrated in Figure 3.3.

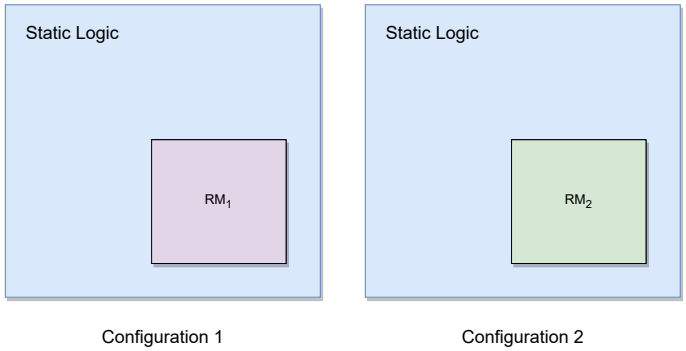


Figure 3.3: Concept of configurations in Partial Reconfiguration

3.2.2 General Concept of Partial Configuration

FPGAs are reconfigurable devices that offer the possibility to change their logic whenever it is possible. Generally, to change the logic of FPGA it is used a process called full reconfiguration which allows uploading a modified or updated design on the board. Even though the full configuration allows reconfiguring FPGAs, it is not the best suitable option possible. This process requires the pause of the current operation and it consumes time which is dependent on the size of the logic.

Partial Reconfiguration is the process of configuring part of FPGA logic blocks by using a partial bit file without affecting the other logic of FPGA. The unaffected part is referred to as the static part and the other part is called the reconfigurable logic. During the reconfiguration process, the static part continues to function without being affected, and only the logic in the reconfigurable logic is changed. Figure 3.1b shows a design with the partially reconfigurable region which is used interchangeably by four configurations.

In Figure 3.1b, the gray area represents the static logic and the white block notated as Reconfigurable Block A, represents the reconfigurable logic. FPGA partial reconfiguration can be performed in two different approaches, in regard to the states of FPGA logic while the reconfiguration is performed, static and dynamic. When the operation is performed in a static state, it means that FPGA logic is in the reset state. On the other hand, the dynamic approach means that reconfiguration is performed while logic is running in FPGA [36].

3.2.3 Partial Reconfiguration Methodologies

The two major methodologies for performing PR are module-based and difference-based. The module-based approach involves implementing reconfigurable modules separately and constraining their components to be placed at a given location. The complete bitstream is then built as the sum of all partial bitstreams, each corresponding to a specific reconfigurable module. In contrast, the difference-based approach involves implementing complete bitstreams separately, with fixed and reconfigurable parts with components constrained at the same location within all bitstreams. The difference between two complete bitstreams is then computed to obtain the partial bitstream needed to move from one configuration to the next context. Both methodologies have their own advantages and disadvantages, and the choice of methodology depends on the specific application requirements, the available hardware resources, and the designer's preference. In the difference-based approach, the number of frames to be written in case the frames are identical in both designs

is less compared to the module-based approach. In this thesis, the focus will be on module-based partial reconfiguration.

3.2.4 Partial Reconfiguration Challenges

One of the main challenges of partial reconfiguration flow is that it is more complex than the standard FPGA design flow. The complexity relies on the fact that designers must plan and create the partitions in the design. Designers can choose to have one or more partitions based on the modules that will be reconfigured and then decide the hardware resources for each of them. This makes the partitioning task more time-consuming and it is a very important step in the flow of dynamic partial reconfiguration since it has an impact on reconfiguration time but also on the area allocated. Choosing a large partition to be configured, increases the configuration time needed to interchange logic in the partition design. On the other hand, choosing to have more than one partition decreases the time needed to reconfigure it but as a consequence, it results in more area used. The process of dynamic partial reconfiguration mostly consists of automatic steps that can be performed using tcl commands. Besides the task of deciding the partition in the design, another manual step in the flow is floorplanning, which is discussed in the section 3.3.

3.3 Floorplanning

Floorplanning is the process of arranging and placing components in an integrated circuit (IC) design to meet specific performance, power, and area requirements. In most cases, floorplanning is a technique used to help a design meet timing. Floorplanning can help improve the setup slack (TNS, WNS) by reducing the average routed delay, but it should be considered that floorplanning can only improve the setup slack. The designer, using floorplanning, can physically place different components of the design in a way that optimizes the timing of the critical paths. The placement of elements like flip-flops, multiplexers, and combinational logic on FPGA can have a significant impact on the timing of the design because the delay of signals propagating through the routing resources can vary depending on their physical distance and topology [37]. This process can be performed through the high-level hierarchy layout or through detailed gate placement. In this context there are multiple approaches to perform floorplanning, each of them with its own advantages and disadvantages.

One approach is detailed gate-level floorplanning that consists of placing individual leaf cells in precise locations on the device, taking into account various design constraints and requirements. This process involves placing each individual logic gate and interconnecting them to form a complete design. One of the main advantages of this technique is the ability to achieve high performance by minimizing the interconnect delays between gates since the designer can choose to place gates that are connected near each other, thus reducing the length of interconnect wires and minimizing signal delay. However, this advantage can be a huge disadvantage sometimes, taking into consideration the level of control, since it can increase the design time and design complexity.

The second technique, hierarchical floorplanning, is dividing the design into smaller sub-blocks and then assigning those blocks to physical locations. This technique in contrast to gate-level floorplanning, allows the designer to work only on specific smaller blocks at a time, reducing thus design complexity and improving design time. One of the main advantages of this technique is the ability to reuse modules across multiple designs. Also, this dividing of the design into smaller sub-blocks simplifies the design verification, allowing each block to be verified independently.

Floorplanning plays a vital role in the partial reconfiguration flow since the placement of the module that will be reconfigured should be done taking into consideration timing and interconnect constraints. In order to achieve the best performance it is important to make an analysis of the proper area where Vivado places the static logic without restriction. This task, however, is not part of the floorplanning process, but it helps the designer to find a better placement for the module that will be reconfigured.

3.4 PL Reconfiguration Paths

The ability to reconfigure the programmable logic (PL) portion of a Field-Programmable Gate Array (FPGA) while the device is in operation is a powerful feature that enables a wide range of dynamic functionality. However, in order to effectively use this capability, it is important to understand the different paths through which the PL can be reconfigured. In this section are explored the various reconfiguration paths available for Xilinx FPGAs, with a focus on the different interfaces used to access the configuration memory of the device. Specifically, we will discuss the JTAG, PS PCAP, and ICAP interfaces, providing an overview of how each interface works and the benefits and limitations of each. This information will provide a foundation for understanding the different options available for configuring and reconfiguring the PL portion of a Xilinx FPGA, enabling more efficient and effective use of this powerful capability. The below subsection refers to the Xilinx technical reference manual document provided by Xilinx in [12]. The PL configuration paths are illustrated in Figure 3.4

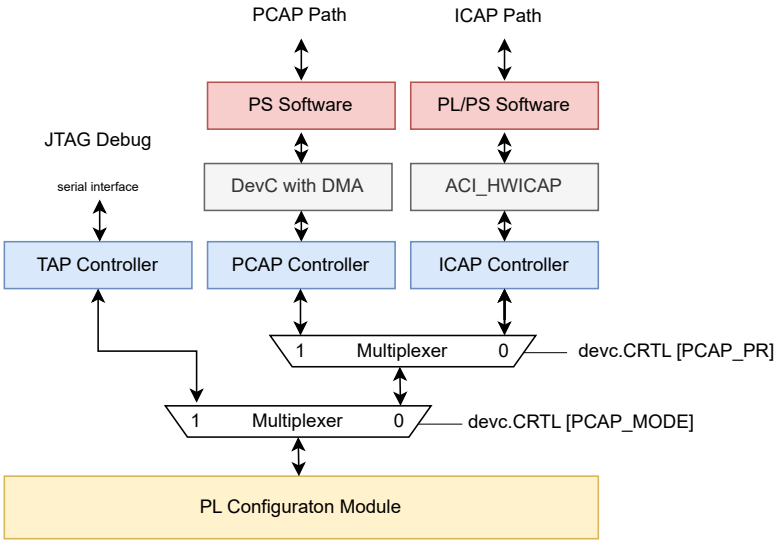


Figure 3.4: PL Configuration Paths

Source: [12]

3.4.1 JTAG

The JTAG interface is a widely-used boundary scan technique that enables easy hardware debugging, testing, and programming of programmable logic devices such as Xilinx FPGAs. JTAG can be used to fully or partially reconfigure Xilinx FPGAs, provided that support for Partial Reconfiguration (PR) is configured on the device. When using JTAG for PR, a specific sequence of JTAG instructions must be executed to program the PR region with the new bitstream data while keeping the rest of the device operational.

The main benefit of the JTAG interface for the partial configuration is that it is easy to integrate into the existing designs. Furthermore, besides the development, JTAG is widely used for in-circuit debugging and reconfiguration, and streamlining the development process. However, the serial interface of JTAG can lead to slower bitstream transfer rates, limiting its efficiency. Additionally, JTAG is a shared resource, meaning that it can only be used by one device at a time, causing delays in multi-device systems.

Despite its limitations, JTAG provides a reliable and established interface for reconfiguring Xilinx FPGAs, including Partial Reconfiguration. With proper device configuration and a well-designed JTAG sequence, designers can take advantage of this interface for efficient and effective FPGA reconfiguration. However, this configuration path is not used for partial reconfiguration but is is

used also to support the testing phase of this process.

3.4.2 PS PCAP

The PS PCAP interface is a fast connection that lets the main processor (PS) access the configuration memory of the FPGA. It allows us to change the FPGA's programming while it's running. This interface has its own special bus for transferring the configuration data quickly and with a lot of capacity. We can control the PS PCAP interface using software, which gives us flexibility and efficiency in managing the configuration process. When it comes to partial reconfiguration, we can use the PS PCAP interface to load specific parts of the FPGA's programming, enabling us to change only certain sections of the device while the rest continues to run. The configuration process in this case is carried out by software, so the need for the hardware IP components is not necessary. However, since during the reconfiguration process, the behavior of the reconfigurable partitions can be random, it is necessary to control the reset and clock signals for each RP. This task can be accomplished by leveraging the capabilities of the AXI GPIO (General Purpose Input/Output) interface. The AXI GPIO interface offers an input/output interface specifically designed for integration with the AXI (Advanced eXtensible Interface) interface. Within the ZYNQ block, this interface is an integral part of the DevC (Device Configuration) component, as illustrated in Figure 2.5.

The device configuration interface (DevC) serves as an intermediary between the configuration source, which can be an external memory device or a microcontroller, and the configuration logic of the FPGA. Its primary functions include configuring the programmable logic (PL), managing device security, and accessing the Xilinx Analog-to-Digital Converter (XADC) through three logic modules as outlined in the Xilinx Technical Reference Manual [12]. Moreover, the DevC encompasses a collection of control and status registers, as well as the capability to generate interrupts. These interrupts can originate from any of the three modules and play a role in determining the PL's state, monitoring the activity of the DMA controller in the PCAP bridge, and facilitating XADC operations.

The PCAP bridge is a vital component that facilitates access to the programmable logic (PL) configuration module and decryption unit by the FSBL/User code software. The configuration module plays a crucial role in processing the bitstream and subsequently loading the SRAM within the PL. On the other hand, the decryption unit serves the purpose of decrypting the encrypted bitstream and code files. It is important to note that the PL must be powered up for the PCAP bridge to function properly.

Regarding the data paths associated with the PCAP bridge, there are four commonly used paths. The first path involves the transmission of non-secure, unencrypted bitstreams. The second path deals with the handling of secure bitstreams and software boot images that are encrypted. The third path facilitates the readback of the PL bitstream from the PL itself. Lastly, the fourth path encompasses a loopback mechanism specifically designed for boot image transfers.

3.4.3 ICAP

The Internal Configuration Access Port (ICAP) is a unique feature of Xilinx FPGAs that allows for Partial Reconfiguration (PR) from within the device's own logic fabric. What makes ICAP particularly appealing is its flexibility in sourcing bitstream data. As long as the data is available to the FPGA logic that feeds the ICAP, it can be used for reconfiguration.

Additionally, the ICAP interface can be used to reconfigure an RP of the PL while the rest of the device continues to operate. This can result in significant time and resource savings, as only the portion of the device that needs to be reconfigured is taken offline. In order to use the ICAP interface for PR, a specific piece of FPGA logic must be dedicated to the task of loading the bitstream. This is because the loading process must remain intact throughout the reconfiguration process. However, with proper design and implementation, the benefits of using the ICAP interface can far outweigh any limitations.

Overall, the ICAP interface provides a powerful and flexible option for performing Partial Reconfiguration in Xilinx FPGAs. As shown in Figure 3.4, in the ICAP Path, the software communicates with a PL logic that is represented by AXI_HWICAP which is a primitive offered by Xilinx.

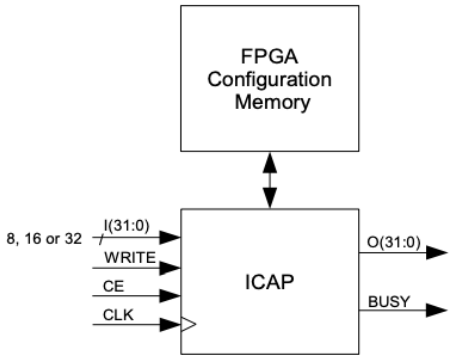


Figure 3.5: Xilinx ICAP Primitive

Source: [38]

The ICAP (Internal Configuration Access Port) primitive, illustrated in Figure 3.5, serves as an interface to the configuration memory of an FPGA [38]. This interface facilitates access to the FPGA's configuration section by incorporating separate input and output data ports dedicated to reading from and writing to the configuration memory. Key input signals include chip enable (CE) and write enable (WRITE), while the busy/ready (BUSY) output signal and clock (CLK) input are also part of the interface. The output signals play a crucial role in reading back the bitstream from the FPGA fabric, enabling access to runtime reconfiguration information.

It is important to consider the ICAP interface's operational specifications. It can be clocked with a maximum frequency of 100 MHz and features a 32-bit wide data interface. These attributes result in a maximum default reconfiguration speed of 400 MB/s, which is theoretically speed. However, the HWICAP IP that uses the ICAP configuration port, can reach a throughput of 19 MB/s because of the complex state machine and the communication overhead.

3.5 Review of Proposed Custom PR Controllers

The following section presents a literature review exploring various possibilities and approaches for dynamic partial reconfiguration. Those approaches are primarily focused on using ICAP or PCAP configuration paths. Building upon the background information provided in the previous chapter, the objective of this review is to give an analysis of the existing research and industry efforts related to partial reconfiguration techniques. This examination of the literature will give an insight into different methodologies, tools, and frameworks proposed in the field. Furthermore, the comparison of those approaches will be given based mostly on the reconfiguration throughput they provide. Finally, this chapter will identify emerging trends and trends in the research industry.

Throughout the years, several custom controllers have been proposed for performing and managing partial reconfiguration using the ICAP (Internal Configuration Access Port) IP. These controllers aim to provide efficient and flexible mechanisms for managing the reconfiguration process on FPGA platforms. This section will discuss some of the proposed approaches, which include both hardware implementations and software routines or drivers. The advantages and disadvantages of each approach will be highlighted based on the findings presented in the specific papers.

3.5.1 AC_ICAP

In the paper [39] a controller AC_ICAP is proposed, similar to the AXI_HWICAP provided by Xilinx. However, in contrast to the AXI_ICAP which requires performing some tasks as software

routines, the proposed controller is implemented completely in hardware. Some of the functionalities that support includes functions to read and write frames, modify LUTs, and load the partial bitstreams from the flash or BRAM memory. The controller supports run-time reconfiguration of LUTs, eliminating the need for precomputed partial bitstreams and facilitating fine-tuned hardware modifications based on dynamically generated values, a characteristic very important for the self-adapted systems. As mentioned in the paper this controller was first developed using a board equipped with Virtex-5 LX110T FPGA and then the same implementation flow was repeated for the Xilinx 7-series family. The controller then was verified for both FPGAs, managing to load bitstreams, read and write frames and also modify the LUTs. Another benefit of this controller is that it can be easily included in systems with embedded processors using the PLB (Processor Local Bus), FSL (Freescale Semiconductor Bus), and AXI (Advanced eXtensible Interface) links [39].

3.5.2 HSDPRC

HSDPRC (High-Speed dynamic partial reconfiguration controller) is a soft IP core presented in [40] that uses the LocalLink DMA and the Xilinx 32b-ICAP primitives. The core utilizes the LocalLink DMA and Xilinx 32b-ICAP primitives, with the hard DMA (HDMA) and ICAP connected via the Xilinx Virtex-5 hard crossbar primitive. Similarly, the Soft DMA (SDMA) and ICAP are linked through the Xilinx multiport Memory Controller (MPMC) Soft IP core. To enhance usability, the controller incorporates internal ICAP functions that enable users to perform tasks such as reading specific addresses in the FPGA and configuring internal read and write masks. In the following section, we will explore the approach employed in the development of the HSDPRC [40]. The controller was able to avoid the use of the PLB bus during the dynamic partial reconfiguration allowing thus the processor full access during the DPR process.

3.5.3 ZyCAP

In the paper [41] a custom controller called ZyCAP was proposed together with its corresponding driver was developed. The authors in this paper improve the configuration speed by connecting the AXI HwICAP to the hard DRAM controller to transfer bitstreams. ZyCAP consists of two interfaces: an AXI-Lite interface connected to the PS through a GP (General Purpose) port, and an AXI4 interface connected to an HP (High-Performance) port. Internally, ZyCAP employs a soft DMA controller, an ICAP manager, and the ICAP primitive. Through the AXI-Lite interface, the DMA controller is configured with the PR bitstream's starting address and size. Bitstreams are then efficiently transferred from external memory (DRAM) to the controller via the high-speed AXI4 interface on the HP port, supporting burst-capable operations. The ICAP manager converts the streaming data from the DMA controller into the appropriate format for the ICAP primitive. Once the complete bitstream has been transferred to the ICAP, ZyCAP triggers an interrupt. In order to support the run-time management of partial reconfiguration, together with the IP, in this paper a ZyCAP software driver is proposed, that includes functionality such as transferring bitstream to DRAM, memory management, bitstream caching, interrupt synchronization, etc. The driver internally manages information about partial bitstreams, including their names, sizes, and locations in DRAM. When a configuration command is received, the driver checks if the bitstream is cached in DRAM. If it is, the ZyCAP soft DMA controller is configured to trigger reconfiguration. If the bitstream is not cached, it is transferred from non-volatile memory to DRAM and a data structure is created. If all DRAM bitstream slots are full, the least recently used bitstream is replaced.

3.5.4 MiCAP and MiCAP-PRO

MiCAP is another custom controller proposed in [42] that is built with an optimized state machine, compared to the HWICAP primitive. The optimized state machine improves the reconfiguration speed of the Xilinx primitive. This controller is composed of two asynchronous FIFO buffers, the

state machine, and the ICAP Primitive. On the other hand, MiCAP-PRO is an improved version that increases the reconfiguration throughput. This is done by adding another DMA engine with an HP port.

3.5.5 Tiny ICAP Controller

This is a recently presented controller in [43] that combines different components like PicoBlaze, Dual Port RAM, Configuration RAM, ICAPE2, FRAME_ECC2, and Scan Monitor. What sets it apart is its smart control system, led by PicoBlaze, which efficiently manages the process of reconfiguring data from the dual port RAM to the configuration RAM. Compared to older controllers, this one stands out for its efficient use of resources. One of the reasons for its success is the smooth coordination between the dual port RAM and configuration RAM, ensuring the easy configuration of the partial bitstream. This controller is designed to perform various tasks without requiring too many resources. It can read information, detect and correct errors, map physical addresses to linear addresses, generate address maps for readback CRC frames, read specific frames using ICAP, store them in the RAM buffer, write information frames to the configuration memory, and even handle Single Event Upsets (SEUs).

3.5.6 DyRACT

DyRACT is a framework proposed in [44] that provides the loading of partial bitstreams over the PCIe interface. This framework encompasses both hardware and software components, enabling seamless communication between the host and accelerator while maximizing throughput. The DyRACT framework comprises two key components: the control logic and the user logic. The control logic is responsible for managing interfaces, controlling reconfiguration processes, and handling clock management to ensure seamless operation. On the other hand, the user logic focuses on implementing a customized hardware accelerator that brings specific functionality to the system. The control logic remains in the static region, while the user logic resides in a partially reconfigurable region (PRR), which allows for dynamic reconfiguration at runtime using the PCIe interface.

3.5.7 VR-ZYCAP

VR-ZYCAP is a controller presented in [45], that is a controller implemented in PL of Zynq SoC enabling flexible and fine-grained incremental reconfiguration of FPGA logic (LUTs and FFs) in run-time. Compared to the other proposed controllers, it reduces the processor dependency, and it has less resource utilization. It stands apart from existing controllers that focus on coarse-grained reconfiguration and achieves faster results by handling LUT and FF reconfiguration directly in hardware while preserving the static logic. Instead of relying on pre-generated bitstreams, VR-ZyCAP generates partial bitstreams dynamically during the run-time read-modify-write cycle, eliminating the need for time-consuming compilation using FPGA design tools. The controller initially uses the PCAP for configuration downloading and subsequently utilizes the ICAP configuration ports for reconfiguration, avoiding the repetitive loading of partial bitstreams from external memory.

3.5.8 Comparative Analysis

A comparison of the throughput achieved by each controller is given in Figure 3.6, however, it is essential to consider other factors for a comprehensive analysis. One crucial aspect is the frequency at which the controllers are operating. In the case of most controllers, they utilize a frequency of 100MHz, except for HSDPC which operates at 133MHz, and DyRACT at 250MHz. The frequency at which a controller operates can significantly impact its performance and throughput.

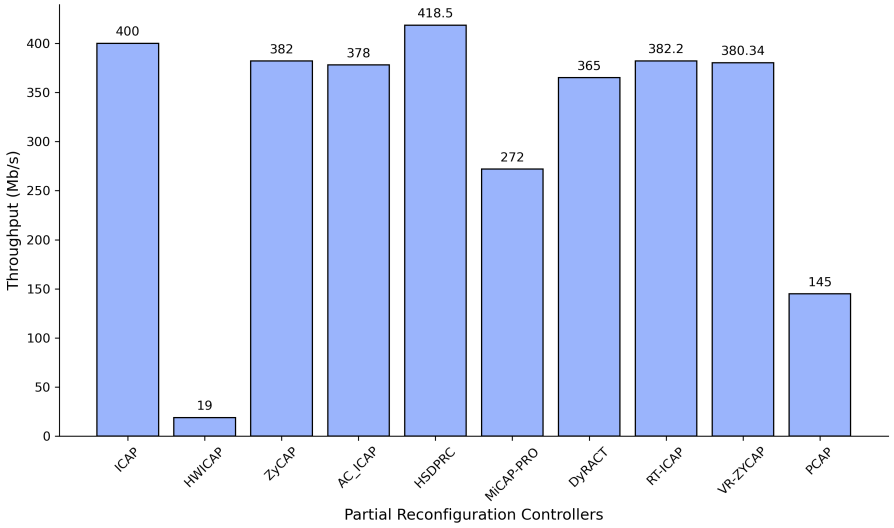


Figure 3.6: Comparison of Partial Reconfiguration Controllers

The figure illustrates various controllers and their corresponding throughput capabilities. The ICAP, with a theoretical throughput of 400 MB/s, is limited to the ZynQ board. It is important to note that this maximum throughput may not be achievable in practical scenarios. Comparatively, the HWICAP controller offers a relatively low throughput of 19 Mb/s when compared to other controllers. On the other hand, ZyCAP boasts a high throughput of 382 MB/s. Although it lacks explicit support for specific features, ZyCAP efficiently transfers bitstreams from external memory to the controller. AC_ICAP provides a throughput of 378 MB/s and supports fine-grain reconfiguration. Meanwhile, HSDPRC achieves a high throughput of 418.5 Mb/s by utilizing the LocalLink DMA and Xilinx 32b-ICAP primitives. This enables the processor to have full access during the reconfiguration process. DyRACT facilitates the loading of partial bitstreams over the PCIe interface, promoting seamless communication between the host and accelerator for maximizing throughput. RT-ICAP supports bitstream compression and utilizes SPM memory, although specific throughput information is not available in the provided table. The Tiny ICAP controller efficiently utilizes resources, enabling seamless coordination between the dual-port RAM and configuration RAM. It performs various tasks with minimal resource requirements and effectively handles Single Event Upsets (SEUs). Lastly, VR-ZYCAP allows fine-grained incremental reconfiguration of FPGA logic, reducing processor dependency. It dynamically generates partial bitstreams during runtime, eliminating the need for pre-generated bitstreams and time-consuming compilation. The PCAP configuration path was selected for performing partial reconfiguration in this master's thesis for two primary reasons: flexibility and resource usage. Firstly, PCAP takes advantage of the existing processor interface and infrastructure, simplifying the design process. Secondly, it avoids additional resource consumption as it doesn't necessitate dedicated hardware components.

3.5.9 Emerging Trends and Future Directions

Due to its benefits, the DPR is being applied to different applications, including space applications, software-defined radio, real-time accelerators, and security or machine learning applications. However, the flow to perform reconfiguration is sometimes complex and time-consuming, since it needs to be supported by hardware and software applications. As described in this section, many controllers are being developed in order to make this process easy, and also to achieve a high throughput. However, there is more to be done to the reconfiguration flow in general. First and foremost, one trend of partial reconfiguration can be to reduce the element that can be reconfigured, fine-grained reconfiguration. Fine-Grained reconfiguration allows the reconfiguration of smaller sub-modules or individual circuit elements. This provides higher flexibility since the resources to be configured are less, and as a consequence achieves less configuration time. Another future direction in the DPR

is the support of High-Level Design tools. The integration with high-level design tools and methodologies will provide a higher level of abstraction and will simplify the design process, enabling thus the designers to utilize all the benefits of the DPR.

CHAPTER 4

Cude Dimension Controller Module

4.1 Introduction

This chapter focuses on the Cube Dimension Controller Module, which is a custom AXI4-Lite interface dedicated to modifying the Y dimension of an HSI cube in the context of the HYPSO mission. By discussing the design considerations and presenting experimental results, this chapter provides a comprehensive guide to the module's functionality and highlights the importance of dynamic dimension changes in HSI analysis.

The Cube Dimension Controller Module is a critical component of the HYPSO mission, where the ability to manipulate dimensions is essential for practical hyperspectral imaging. This chapter explores the implementation of the custom AXI4-Lite interface, which enables seamless modification of the Y dimension. It delves into the design considerations involved in developing the module, emphasizing the advantages of the chosen interface for this specific task. By introducing the motivation and providing an overview, readers will gain a clear understanding of the objectives and significance of the custom AXI4-Lite interface in enabling flexible and efficient manipulation of the Y dimension of the HSI cube.

4.2 Motivation

The motivation behind the development of the Cube Dimension Controller Module stems from the inherent limitations posed by fixed dimensions in the hyperspectral image (HSI) cubes in the HYPSO mission. The ability to dynamically manipulate the cube's dimensions offers significant advantages in various HSI applications. In the HYPSO mission, the module plays a crucial role in efficiently adjusting the Y dimension of the HSI cube, where X and Y represent spatial coordinates and Z represents spectral information. This adjustment is necessary because the CCSDS-123 compression algorithm is unable to compress images of varying sizes. However, the satellite is capable of capturing images in multiples of a fixed size. As the satellite progresses along its orbit, it captures consecutive images with Y values, such as 200, followed by another 200, and so on.

Based on previous research conducted in master theses [7] and [46], it was discovered that the limitation arises from the algorithm's reliance on parallel processing and pipelining to achieve high-throughput rates. This necessitates a fixed hardware configuration that can efficiently process the input data in parallel. If the image size or band configuration changes, the hardware must be reconfigured to accommodate the new input data, resulting in a significant burden and reduced performance.

Moreover, dynamic dimension changes can facilitate the CCSDS-123 compression algorithm for efficient storage and transmission of HSI data. By adjusting the dimensions of the cube, it is possible to reduce its overall size while maintaining essential spectral information, enabling flexibility and avoiding the FPGA reconfiguration. This can lead to significant savings in storage space and

transmission bandwidth, which are crucial considerations for the HYPSO mission.

4.2.1 Relevance of AXI4-Lite in Hyperspectral Imaging Applications

Due to the advantages of the AXI4-Lite interface, like its compact design, memory-mapped communication capabilities, and compatibility make it an ideal choice for integrating and controlling various components within the limited resources of CubeSats platforms. In the case of the HYPSO mission, the AXI4-Lite interface was specifically chosen for its ability to efficiently facilitate the dynamic adjustment of the dimensions of the hyperspectral image cube. Using AXI4-Lite the cube dimension, specifically the Y dimension, can be easily modified ensuring efficient utilization of onboard resources and optimizing data capabilities. Moreover, this interface can be easily integrated with the current implemented design of the HYSPO mission.

4.3 Implementation Details

This section will discuss the implementation details related to the integration and utilization of the AXi4-Lite interface. Moreover, the section will cover signal connections, register mapping, and configuration considerations.

4.3.1 Custom AXI4-Lite IP

In this subsection, the custom AXI4 Lite IP was developed to control the Y dimension of the HSI cube. In Figure 4.1 it is given the custom IP, named DimController, that is implemented based on Xilinx resources.

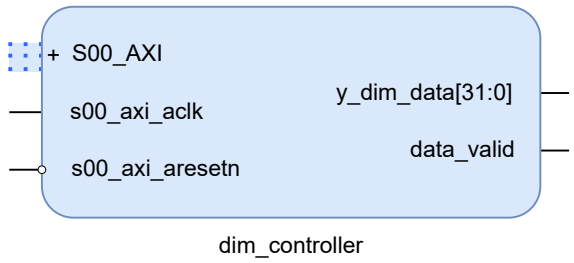


Figure 4.1: DimController - Custom AXI4-Lite IP

As illustrated the IP features several ports such as:

1. S00_AXI: This port represents the AXI4-Lite interface of the DimController. It serves as the communication channel between the component and a host processor. The S00_AXI port includes signals such as AWVALID, AWREADY, AWADDR, WVALID, WREADY, WDATA, BVALID, BREADY, and BRESP, enabling read and write transactions.
2. s00_axi_ackl: This signal is an acknowledgment signal from the DimController to the host processor. It indicates the successful completion of a read or write operation initiated by the processor. The processor waits for this signal before proceeding with subsequent transactions.
3. s00_axi_aresetn: This signal is the active-low asynchronous reset signal for the DimController. When this signal is asserted low, it resets the component to its initial state, ensuring a known starting point for proper functionality.
4. y_dim_data[31:0]: This port is an output signal that represents the Y dimension of the HSI cube controlled by the DimController. It is a 32-bit vector that carries the data related to the Y dimension of the cube.

5. data_valid: This signal is also an output of the DimController. When the data_valid signal is asserted the register y_dim_data register contains a valid y dimension.

4.3.2 RTL Code

The module was implemented in System Verilog based also on the resources and recommendations provided by Xilinx. In order to customize the AXI4-Lite IP the below RTL code modifications were done in order to add the functionality to change the Y dimension of the HSI cube:

Listing 4.1: Dimensionality Controller section code to write to Y dimension register

```
1  reg y_valid;
2  always @(posedge S_AXI_ACLK) begin
3
4  if (S_AXI_AWVALID && S_AXI_AWREADY && S_AXI_WVALID && axi_wready) begin
5    //A write operation has been completed, set the signal to 1
6    y_valid = 1'b1;
7  end else begin
8    // No write operation, keep the signal at 0
9     y_valid = 1'b0;
10  end
11 end
12
13 assign y_dim_data = slv_reg0;
14 assign data_valid = y_valid;
```

The code is contained within an always block that is triggered by the positive edge of the AXI4 Lite clock signal (S_AXI_ACLK). The purpose of this code is to determine the validity of data written in the y_dim_data register. By evaluating the states of various AXI4 Lite interface signals, such as write validation (S_AXI_AWVALID), write readiness (S_AXI_AWREADY), data validation (S_AXI_WVALID), and response validation axi_wready, the code determines whether a write operation has been successfully completed. If all the specified signals are asserted at the same time, indicating a completed write operation, the y_valid signal is set to high, signifying the validity of the data.

Conversely, if any of the signals are not asserted or the write operation is incomplete, the y_valid signal is set to low, indicating the data's invalidity or the absence of a write operation. This mechanism enables the host processor to determine the appropriateness of reading the HSI cube's dimension from the y_dim_data register, enhancing the overall control and synchronization between the IP and the processor. This part is to ensure the validity of the value stored in y_dim_data. This is because the CCSDS-123 algorithm is set up using the generic parameters given as input.

The last two assign lines indicate that the value stored in the slv_reg0 register is assigned to the y_dim_data output signal. The slv_reg0 is a register within the AXI4 Lite IP. The assign statement in the RTL code is used to assign the value of slv_reg0 to the y_dim_data output signal. This allows the value stored in slv_reg0 to be propagated to the output port and made available for the host processor to read.

In order to use the output data from this module, the CCSDS-123 module was changed in order to support the dynamic change of the Y dimension. The code snippet below represents a process that handles the reading of the y dimension, where NY is a generic value of a CCSDS123 compression IP, and y_valid is a signal coming from the dimensionality controller IP specifically the data_valid signal. Within the process, the y dimension (y_dim) is updated based on the rising edge of the CLK signal. The purpose of this process is to determine the value of the Y dimension based on the state of the valid signal, in order to be sure that a value has been written in the register. Then in the module, the value in the y_dim signal is used during the operation.

Listing 4.2: Changes in the CCSDS-123 top module

```
1 signal y_dim : integer := NY;
```

```
2
3 process (clk)
4 begin
5   if rising_edge(clk) and y_valid = '1' and unsigned(y_value) >= 0 then
6     y_dim <= to_integer(unsigned(y_value));
7   end if;
8 end process;
```

If the valid signal is asserted ('1'), indicating that new data is written in the y_dim_data register, the process saves the value read from the DimController IP into the y_dim signal. A simple check is added in the if condition to ensure that the value that will be written is not negative. Conversely, if the valid signal is de-asserted ('0'), indicating that the data is not valid, the value present in the y_dim signal is not changed.

By implementing this process, the code ensures that the value of the y dimension is determined based on the state of the enable signal. If the data is valid, the process updates the Y dimension with the value that needs to be changed. Otherwise, it sets the Y dimension to a default or generic value (NY). Now the calculations for the cube size in the architecture of the ccsds123 top file are done based on the value present in the y_dim register.

In order to support the dynamic change in the CCSDS-123 module, besides the top module, the pipeline and control modules are the only ones that were changed. The changes consist of adding another input port for the new Y dimension, and also the changes in the architecture part in order to do the calculations based on this new value.

4.4 Device Driver Development

As discussed in the background chapter, device drivers play a crucial role in facilitating communication between the operating system and hardware devices by providing an interface for applications to interact with the hardware components. This section provides a description of the device driver for the Cube Dimension Controller module. The device driver is implemented as a loadable kernel module that registers itself as a character device. It offers two primary input/output control commands: IOCTL_SET_DIMENSION and IOCTL_GET_DIMENSION. These commands enable user-space applications to set and retrieve the Y-dimension value, respectively.

The driver, which can be found in the appendix, consists of two main files: dim_controller.h and dim_controller.c. The header file defines the IOCTL commands, device file name, and other essential constants. On the other hand, the dim_controller.c source file implements the functionality of the device driver. It includes the IOCTL handling function, file operations structure, and initialization and cleanup functions. The input/output control functions interpret the commands received from the user application, perform the necessary operations, and communicate the results back to the application. Overall, the developed device driver provides a solid framework for controlling the Y-dimension of the HSI cube and serves as a crucial component in enabling efficient communication between the operating system and the Cube Dimension Controller module.

4.4.1 Build and Install

To create the module under the PetaLinux project, use the following command:

```
petalinux-create -t modules --name dimcontroller --enable
```

This command generates a module named *dimcontroller* within the PetaLinux project. It automatically creates a Makefile, a C-file, and a README file. Make the necessary modifications to these files to add the desired functionality explained in the section above.

Once the Linux is loaded onto the board, the loading and unloading of the Linux kernel module for this module can be done using the below commands:

```
modprobe dimcontroller.ko   #Loads the module into the kernel
modprobe -r dimcontroller.ko #Unloads the module into kernel
```

To interact with the *dimcontroller* module, develop a user application that communicates with the module through the */dev/dimcontroller* device file. In order to set and get the cube dimension the below two functions are used, which call the IOCTL commands;

Listing 4.3: Functions to set and get the Y cube dimension

```
int setDimension(int new_dimension) {
    int file_desc = open(DEVICE_FILE, O_RDWR);
    if (file_desc < 0) {
        perror("Failed to open device file");
        return -1;
    }
    if (ioctl(file_desc, IOCTL_SET_DIMENSION, &new_dimension)<0) {
        perror("Failed to set dimension");
        close(file_desc);
        return -1;
    }
    printf("Set dimension to %d\n", new_dimension);
    close(file_desc);
    return 0;
}
int getDimension(int *current_dimension) {
    int file_desc = open(DEVICE_FILE, O_RDWR);
    if (file_desc < 0) {
        perror("Failed to open device file");
        return -1;
    }
    if (ioctl(file_desc, IOCTL_GET_DIMENSION, current_dimension)<0) {
        perror("Failed to get dimension");
        close(file_desc);
        return -1;
    }
    close(file_desc);
    return 0;
}
```

This code opens the device file in read-write mode and returns a file descriptor (‘file_desc‘) for further operations. Once the device file is successfully opened, utilize the functionality of the module. For instance, to set the y-dimension value, use the following code:

Listing 4.4: Application to call the set and get functions

```
int main() {
    int new_dimension;
    int current_dimension;
    printf("Enter the new dimension: ");
    scanf("%d", &new_dimension);

    if (new_dimension <= 0) {
        printf("Invalid dimension.\n");
        return 1;
    }
    if (setDimension(new_dimension) != 0) {
        printf("Failed to set dimension\n");
        return 1;
    }
```

```
    if (getDimension(&current_dimension) != 0) {
        printf("Failed to get dimension\n");
        return 1;
    }
    printf("Current dimension: %d\n", current_dimension);
    return 0;
}
```

This code sends an IOCTL command ('IOCTL_SET_DIMENSION') along with the desired new dimension value to the *dimcontroller* module through the file descriptor. Furthermore, a device node was added to the device tree for this module in order to avoid kernel panic that can happen because of memory limitations.

4.5 Testing

This section will include the testing approach followed to test the Dimensionality Controller module. Firstly, in order to verify that the functionality of the module is correct the simulation and debugging in Vivado was used. After the simulation that initiate a write transaction, it was seen that both data and valid signals were set correctly. Secondly, the communication between the module and the CCSDS-123 module was tested in order to ensure that the desired functionality was achieved. In order to test the functionality of the module a simple software application was also used. This application which can be found in the project consists of a self-test function that checks if the module is operating within expected parameters and that all its functionalities are working as intended.

In check, if the changes in the CCSDS-123 module have not affected its functionality, the Emporda tools were used. Emporda is a software implementation of CCSDS Recommended Standard for multi-spectral and hyperspectral image coding, that offers two modes, compression and decompression. In the compression mode, Emporda takes an image as input and generates a compressed file. In decompression mode, Emporda expects a compressed file and recovers the original image. The software allows for the configuration of various parameters related to input images, compression/-decompression algorithms, and coding/decoding algorithms. Default parameter values, as well as specific predefined parameter files, are provided, with Emporda verifying parameter values and combinations to prevent errors.

The verification system utilizes simulation data from a specific hyperspectral image cube and performs comparisons between the compressed bitstream produced by the implemented design and the decompressed bitstream obtained through Emporda's software implementation. A visual representation of the automated verification system can be found in Figure 4.2. In the verification system, the new Y dimension and the valid signal were provided as inputs for testing purposes. Through randomized testing, it was observed that the resulting bitstreams from the compression and decompression processes were identical, affirming the successful functionality of the module.

In order to ensure accurate results, adjustments were made to the automated tool during this phase. It was discovered during testing that the system incorporated a comparison of residual bits between a byte string and an integer, which led to errors and resulted in differing bitstreams. These refinements aimed to enhance the reliability and consistency of the verification process. For the procedure of how to use the Emporda tool, the [7] defines all the steps that need to follow in order to test using the Emporda tool. Before starting testing using this tool, it is important to first run the simulation in Vivado, even though it will fail. This is done because Vivado will create the necessary files needed by the verification system.

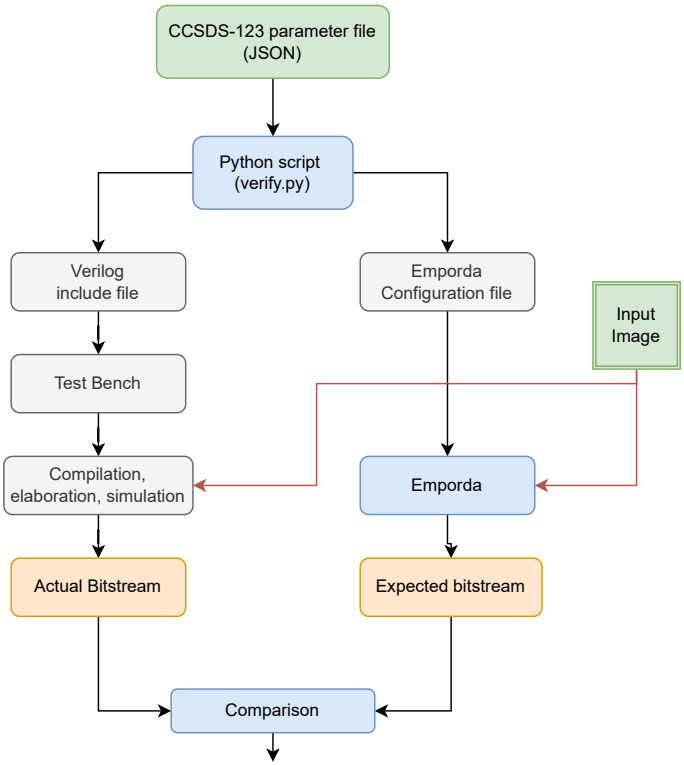


Figure 4.2: Overview of the automatic verification system

Source: [7]

4.6 Conclusion

In conclusion, the focus of this chapter was on the Cube Dimension Controller Module, which implements a custom AXI4-Lite interface for modifying the Y-dimension of an HSI cube in the HYPSO mission. The chapter has provided an introduction to the module, discussing its significance in enabling dynamic dimension changes in HSI analysis. The motivation behind the module's development was explained, highlighting the limitations of fixed dimensions in traditional HSI cubes and the advantages of dynamic dimension changes in terms of compression and resource utilization.

Furthermore, the chapter discussed the implementation details of the Cube Dimension Controller Module, including the development of a custom AXI4-Lite IP and the necessary modifications to the RTL code. The custom IP, named DimController, was described along with its ports and signals. The RTL code modifications were explained, focusing on the assignment of the Y-dimension data to the output signal and the determination of data validity based on the AXI4-Lite interface signals. Overall, this chapter provides a comprehensive guide to the Cube Dimension Controller Module and highlights the importance of the custom AXI4-Lite interface in enabling flexible and efficient manipulation of the Y-dimension of the HSI cube in the context of the HYPSO mission.

CHAPTER 5

Dynamic Partial Reconfiguration using PCAP

5.1 Vivado Software Flow

This subsection includes the flow that was followed in order to design a project that can be used to perform a partial reconfiguration. Part of the Vivado tool is the Dynamic Function eXchange (DFX) which allows for the reconfiguration of modules within a design. In the process of synthesizing and implementing a design, two crucial steps need to be undertaken. The first step involves generating a top-level wrapper that acts as an interface between the design and the overall system. This can be accomplished in the Vivado design environment by right-clicking on the top-level block diagram in the Sources window and selecting the `Create HDL Wrapper` option. The resulting wrapper file, named `design_wrapper.v`, is then included in the project and serves as the instantiation point for the top-level block diagram.

The second step entails generating the RTL and IP files required for synthesis. To perform this step in Vivado, you need to navigate to the Flow Navigator and select the "Generate Block Design" command located under the IP INTEGRATOR header. This action triggers a dialog box to appear, presenting you with several options: `Out of context per IP`, `Out of context per Block Design`, or `Global`. In the case of the partial configuration flow, it is crucial to choose the `Global` option since it will ensure that the generated RTL and IP files cover the entire design, including all components and connections. This comprehensive approach to synthesis allows for a thorough evaluation and optimization of the entire system. After this step, the creation of the necessary RTL and IP files can be initiated. This step is critical as it lays the foundation for subsequent stages in the design process, such as synthesis, implementation, and verification, enabling the successful realization of the DPR project [47]. The flow of the partial reconfiguration process is based on the DFX flow, provided by Xilinx that consists of different steps, as illustrated in Figure 5.1, each with a specific purpose.

5.1.1 Synthesis

A partial reconfigurable design requires a bottom-up design, which is a synthesis flow in which each module has its own synthesis project [47]. In order to achieve this, the first step is the convert the design into a design that consists of the static part and the Reconfigurable Partitions (RP). The top-level module should initiate both those components, but when it comes to Reconfigurable Partition, it should have a black box for each of them, without any logic. That means that static and reconfigurable designs should be synthesized separately. To synthesize the top-level module, a TCL script is used that specifies which parts of the design should undergo synthesis.

```
set run.topSynth 0
set run.rmSynth 1
set run.prImp 0
set run.prVerify 0
```

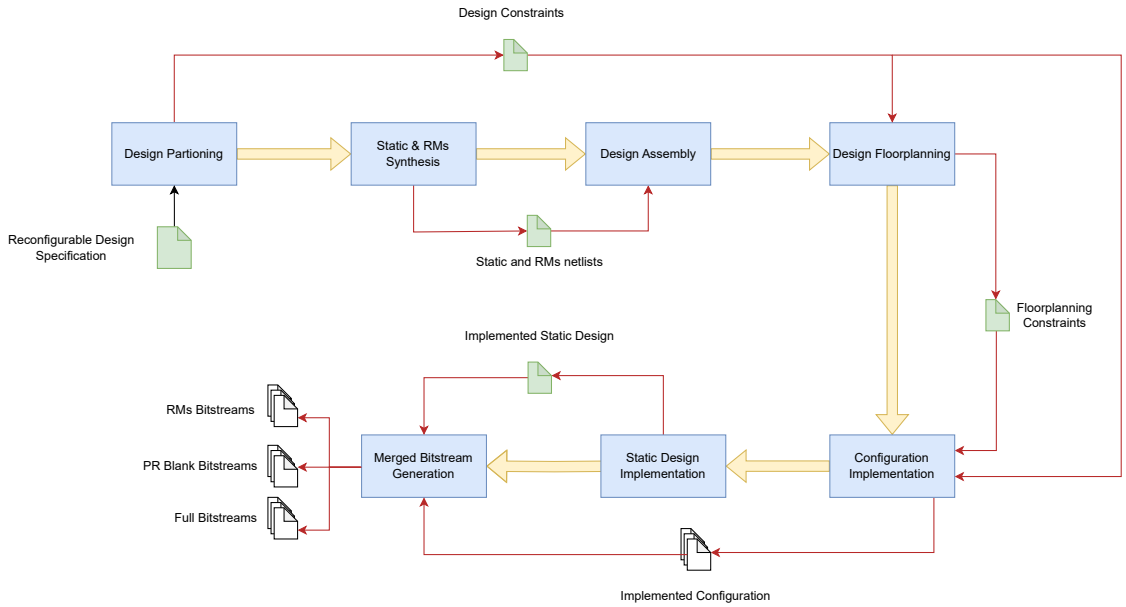


Figure 5.1: Graphical PR flow representation based on DFX

```
set run.writeBitstrim 0
```

In this case, the script is configured to synthesize only the reconfiguration modules, while avoiding the synthesis of the top module, which is synthesized separately in the Vivado IDE. By executing this script, the netlists for each of the RM are generated, each with its own configuration.

Global Synthesis

Global synthesis involves the synthesis of the entire design, including both the static and dynamically reconfigurable portions, into a single global design representation. The synthesis tool analyzes the design, performs optimizations such as logic folding, technology mapping, and resource sharing, and generates a gate-level netlist. Once the static portion of the design has been synthesized, the dynamically reconfigurable portions are integrated into the global design representation. This involves mapping the reconfigurable modules onto the FPGA device and establishing the necessary interconnections between the static and reconfigurable portions. The synthesis tool analyzes the reconfigurable modules, performs optimizations specific to partial reconfiguration, and generates a modified global netlist that includes both static and reconfigurable elements. The global synthesis also includes additional steps such as placement and routing. This ensures that the design meets the required timing constraints and minimizes the propagation delays between different elements.

The output of global synthesis is a complete design representation that can be used for programming the FPGA device. This representation includes the configuration bitstream for the static portion of the design and the partial bitstreams for the reconfigurable modules. The global synthesis process plays a crucial role in achieving efficient and optimized implementations of FPGA-based systems with dynamic reconfiguration capabilities.

Out-of-context Synthesis

In contrast to global synthesis, which synthesizes the entire design as a single entity, out-of-context synthesis allows for independent synthesis and optimization of specific modules or sub-designs. The purpose of out-of-context synthesis is to enable modular design and development where different modules can be synthesized and tested independently and thus offers several advantages, including better manageability, improved design reuse, and faster design iterations. The out-of-context

synthesis process focuses on optimizing the module's functionality, resource utilization, and timing characteristics without considering the interactions with other modules in the system. The out-of-context synthesis also facilitates design reuse. Once a module has been synthesized and verified, it can be reused in multiple designs or projects without the need for repeating the synthesis process. This promotes efficient development and accelerates time-to-market for FPGA-based systems.

5.1.2 RTL Project Flow in Vivado IDE

The DFX project flow inserts the key requirements into the existing Vivado project. Some of the key requirements include defining RP within the design hierarchy, population a set of RM for each RP, creating a set of top-level and module-level synthesis runs, and creating a set of related implementation runs. It should be mentioned that a project cannot be broken up or exported since Vivado will now be able to track dependencies between runs and sources.

- The first step is to create the top-level design
  So the top-level design will contain the block design and a wrapper for each of the reconfigurable regions. In this case, the CCSDS-123 component will be removed from the top-level design. Another top-level file, reconfig_top, was created in the design project that will instantiate the CCSDS-123 module which was converted into a black box[1]. As illustrated in Figure 5.2, the design is composed of the static logic and the reconfig_top module, which contains the reconfigurable logic that is represented by the ccsds123_top module.

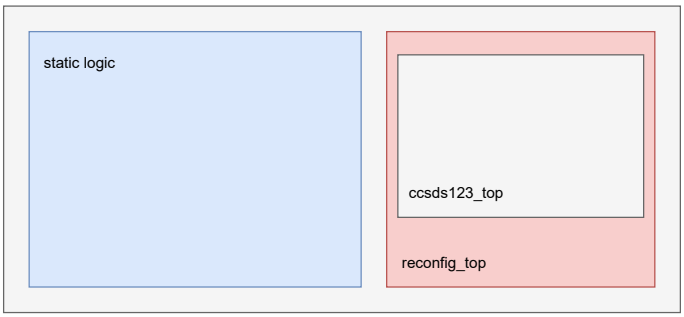


Figure 5.2: High-level structure of the design

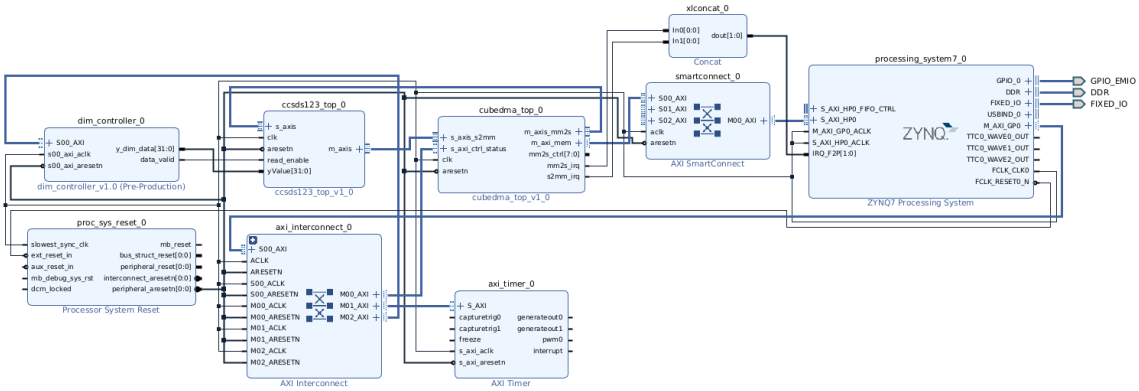


Figure 5.3: Design including CCSDS-123, CubeDMA and DimController components

- Creating the custom ccsds123 IP
  A custom IP named ccsds123 was developed, with the reconfig_top module serving as its top file. Subsequently, the IP was incorporated into the entire system diagram and connected to other modules using the required signals. This approach was adopted to establish a high-level

[1] A component is considered as a black box if it doesn't contain any implementation details but just the definition

design that facilitates the division of reconfiguration regions and the static portion. Moreover, since the state of the reconfiguration region (RR) is unknown during the reconfiguration process and can impact other FPGA logic, this high-level design aids the controller in isolating each reconfigurable partition (RP). The complete design that includes this IP as well is given in Figure 5.3

5.1.3 Assembly of the design

After the synthesis is completed, the complete netlists will that include the static logic and each RP logic, will be created. The netlists can be opened using the TCL command: `open_checkpoint <filename>`. When the static netlist is open, Vivado will issue a warning stating that it couldn't locate the logic for the previously defined black box. In this step, it is important to locate the path of the black box component since it will be required in subsequent steps. The black box component can be found in the netlist sources and is represented by a black square symbol.

Afterward, the netlist for each of the reconfigurable modules should be linked with the black box definition in the top-level module of the static part. In this case, the logic path of the module should be provided as input to the command, as shown below:

```
read_checkpoint -cell <blackBoxPath>/<componentName>
<pathToNetlist>/<filename>
```

This command connects the module logic with the previously defined black box component. After linking the components, the next step is to designate the Reconfigurable Partition (RP) as partially reconfigurable. This can be achieved by setting the `HD_Reconfigurable` property to 1.

Then the assembled design can be saved using the `write_checkpoint` command. This process should be performed for each of the Reconfigurable modules in the Reconfigurable Partition, by giving as input its own netlist file.

5.1.4 Floorplanning

During this step, the floorplanning process is performed to define the regions that will undergo partial reconfiguration. The floorplan, also known as Pblock, determines the allocation of physical resources for the Reconfigurable Modules. This step holds great importance in the partial reconfiguration flow as it ensures that the necessary restrictions and requirements are met. If the design includes flip-flops, it is crucial to consider a specific property in the floorplanning phase. In the properties section, the `RESET_AFTER_RECONFIG` checkbox should be enabled. This property is a beneficial feature that helps ensure the proper routing of the reset signal during the partial reconfiguration process.

By setting the `RESET_AFTER_RECONFIG` property, the tool automatically generates the required logic to synchronize the reset signal with the completion of the reconfiguration process. This proactive approach helps prevent any potential issues that may arise if the reset signal is not correctly synchronized with the reconfiguration process.

However, it is important that the floorplanning is performed for the configuration that has the biggest use of resources. So it is necessary first to define a configuration with the maximum use of resources and then the floorplanning can be used for other configurations as well. The floorplanning approach followed in this project prioritized implementing the configuration with the highest resource utilization. For 7-series devices, it is important to ensure that the Reconfiguration Partition (RP) is framed-aligned, meaning that the Pblock is aligned with the clock region boundaries when the `RESET_AFTER_RECONFIG` property is set. However, it should be noted that this requirement is not applicable to UltraScale devices.

During the floorplanning process, it should be considered that not all the elements can be configured so they should not be included in the Pblock. The elements that can be configured include

CLBs, BRAMs, DSPs, and routing resources. However, the elements like clocking resources, I/O resources, and architecture feature components cannot be configured. That means that the width and composition of the Pblock must not split interconnect columns. When determining the width of the RP, it is recommended to choose a width that optimizes the utilization of interconnect and closing resources. In 7-series devices, as explained in the background chapter, the routing resources are positioned adjacent to or back-to-back with each other. Therefore, a key requirement regarding the width is to ensure that the left and right edges of the Pblock are placed between two interconnect columns (INT-INT). This allows the tool to utilize all available resources for both the reconfigurable logic and the static logic during the place and route process. In cases where the specific locations of these columns are not known, the `SNAPPING_MODE` property can be utilized within the Pblock. This property automatically resizes the Pblock to avoid violations related to back-to-back placement. Another thing to consider is that the Pblocks should not overlap with each other in the design and also nesting of the RPs is not supported.

By following these considerations during floorplanning, the design can optimize resource utilization, ensure proper alignment, and avoid potential issues related to interconnect placement. The initial step is to determine the resources required by the CCSDS-123 IP and then add the 20% overhead that is needed for routing resources. This ensures that there is sufficient room for routing resources and helps avoid issues related to interconnect placement. In order to check if the floorplanning is performed correctly, the Partial Reconfiguration Design Rule Checks (DRC) must be run. This report will include all warnings and errors related to partial reconfiguration. Each of the warnings should be taken into consideration since they can affect the following steps in the partial configuration flow. After this step, the Pblock definitions and their associated properties must be saved on a .xdc file.

5.1.5 Implementation of the configurations

During the implementation stage of the partial reconfiguration flow, the synthesized netlist is mapped into the target FPGA device's resources. Since there can be multiple configurations in partial reconfiguration, implementation needs to be performed for each Reconfigurable Module (RM) separately. The implementation process includes loading the design constraints and performing the place and route. Loading design constraints is crucial in ensuring the success and reliability of the design during the implementation of each configuration. More specifically, the PS constrain file, which contains information about pin assignments, I/O voltage levels, and clock frequencies should be loaded for each of the configurations. This file can be generated by Vivado when the system design that contains the PS is synthesized. The below command is used to load the constraint:

```
read_xdc <filename.xdc>
```

After loading the design constraints, the constrained optimization, place, and routing processes are performed during implementation in the partial reconfiguration flow. The following commands can be used for these steps:

```
opt_design
place_design
route_design
```

After each step in the implementation process, it is crucial to preserve the generated placement and routing results to ensure their integrity and avoid unintentional modifications. This can be achieved by saving the implementation results as checkpoints. Checkpoints serve as snapshots of the design's implementation state at specific stages. They capture the placement and routing information, including the physical location of the design's components, interconnections, and other relevant data.

```
write_checkpoint <name.dcp>
```

Until this point, each of those steps should be performed for all RM variants. Furthermore, since the static portion of the configuration will be used for all subsequent configurations (for every RM or RPs), the static design must be isolated and the RM module must be removed. Also, the placement and routing are locked down in order to guarantee consistency for different RMs for each RP.

```
update_design -cell <path-to-blackbox-component> -black box
lock design -level routing
```

After the placement and routing process for all configurations is finished, it is important to perform a final verification check to ensure consistency between these configurations. This verification is carried out using the pr_verify command, which takes multiple routed checkpoints (DCPs) as inputs. The command then generates a log that highlights any differences found in the static implementation and Partition Pin placement between the configurations.

```
#To compare two configurations
pr_verify <config1_routed.dcp> <config2_routed.dcp>

#To compare more than two configurations
pr_verify -initial <config1.dcp -additional {config2.dcp confign.dcp}
```

5.1.6 Bitstream Generation

In the partial reconfiguration flow, the generation of bitstreams, instead of creating a single complete bitstream file for the entire design, the write_bitstream command is used to generate multiple bit files. For each design configuration, this command generates a full standard configuration file and additional partial bit files for each RM within that configuration. To ensure proper identification and organization of the generated bit files, Xilinx recommends providing the configuration name and RM names using the -file option when executing the write_bitstream command as below:

```
write_bitstream -file Config_1.bit
```

When performing partial reconfiguration using the FPGA Manager, it is necessary to create binary bitstreams. The binary bitstream format (.bin) is specifically designed to support the partial reconfiguration process and includes a header and other metadata required for successful reconfiguration.

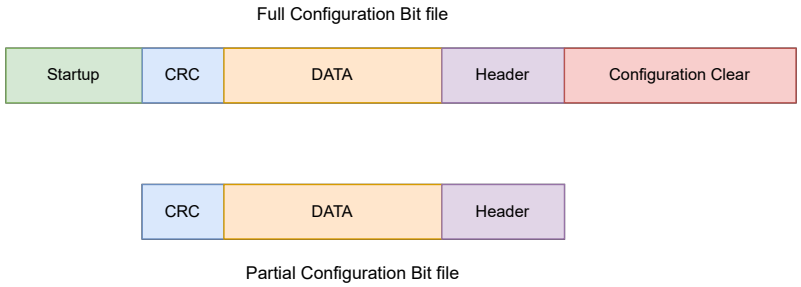


Figure 5.4: Structure of the full and partial bitstream files

Full and partial bitstream configuration bit files have different structures. As shown in Figure 5.4, the full configuration bit file includes the startup, CRC, data, header, and configuration clear. On the other hand, the partial bit files contain the CRC, configuration data, and header. The CRC (Cyclic Redundancy Check) is used as a mechanism for error detection and verification during the configuration process of FPGA.

Also, it should be mentioned, that this flow creates the black box partial bitstreams that don't contain any logic for the RPs. Those bitstreams can be used to remove the previous logic of RM and then substitute it with a new configuration bitstream.

5.2 PL Configuration through PS

The next and final step in dynamic partial reconfiguration is to reconfigure the PL through the PS. The high-level flow of how the partial reconfiguration flow is performed through software is given in Figure 5.5. A software application, that is executed in the CPU is used to give an instance as input to the configuration manager. The configuration manager then reads the specific bitstream file from SD Card, and then using one of the configuration paths, in our case the PCAP path, the bitstream is loaded in the FPGA configuration logic.

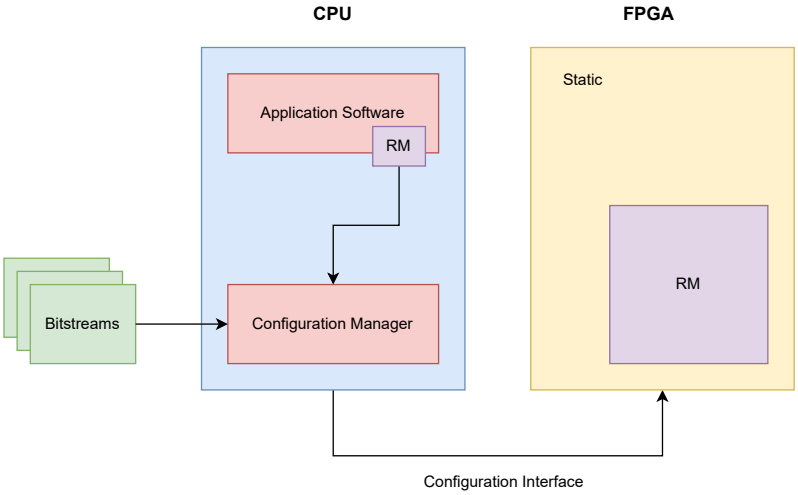


Figure 5.5: High-level flow of PL configuration through PS

To transfer the partial bitstream from the Programmable System (PS) to the Programmable Logic (PL) region of the FPGA, the Process Configuration Access Port (PCAP) interface can be utilized. The PCAP interface enables communication between the PS and the PL for configuration purposes. The general steps to configure the PL via the PCAP interface are as follows:

1. Enable the interface the select PCAP programming path. This step establishes the communication link between the PS and PL for configuration purposes.
2. Clear interrupts to ensure a clean configuration process. Clearing interrupts ensures that any pending signals are resent, avoiding potential interference during the configuration.
3. Initialize the PL by preparing it for the upcoming configuration. This involves setting up the necessary resources and ensuring a proper starting state.
4. Disable the internal DevC (Device Configuration) loopback function. This step ensures that the configuration proceeds smoothly without any interference from internal feedback. Disabling the loopback function ensures that the PL receives its configuration data through the intended method, with is via DevC DMA.
5. Transfer the new bitstream file to the PL using the DevC DMA.

To facilitate the testing of dynamic partial reconfiguration through the PCAP (Processor Configuration Access Port) interface, a simple application was developed. This application utilizes the devcfg driver, which acts as the interface between the processing system (PS) and programmable

logic (PL) of the FPGA. The devcfg driver enables the transfer of bitstream data from the PS to the PL, ensuring power synchronization and control during the configuration process.

It is important to note that the devcfg driver has been deprecated in the 2018.1 release of the Linux Kernel and is no longer included in the mainline tree. However, Xilinx now provides a higher-level abstraction called FPGA Manager, which offers a more advanced and recommended approach for dynamic FPGA reconfiguration. It is worth mentioning that some of the FPGA Manager APIs are built on the functionality provided by the devcfg driver.

Although FPGA Manager is the preferred method for dynamic partial reconfiguration, there are scenarios where the application based on the devcfg driver can still be useful. Firstly, in legacy systems that rely on older versions of Linux with support for the devcfg driver, the application can be utilized for dynamic partial reconfiguration without the need to migrate to newer Linux versions. Additionally, in customized environments where specific Linux versions can be maintained, the application can be deployed to enable dynamic partial reconfiguration.

This application begins by initializing the necessary components such as the interrupt controller, PCAP interface, and SD card. It then performs a full configuration of the FPGA using a specified bitstream file. After that, it proceeds with two partial reconfigurations using different bitstream files. The application also includes functions to handle interrupt callbacks and error handling. In the end, the cache is flushed, and the application returns. The application code can be found in the appendix.

FPGA Manager provides a framework for managing the partial reconfiguration of FPGAs, and it simplifies the process of reconfiguring specific regions of FPGAs. As explained in the background chapter, two are the main concepts in FPGA Manager, FPGA Region, and FPGA Bridge. The FPGA Region is defined during the design phase, which also specifies the boundaries and the resources needed by this region that will serve as the part that needs to be configured by the FPGA Manager tool. Another component concept is FPGA Bridge, which facilitates the communication between the reconfiguration regions and the static region, by acting as an interface for data and control signals. During the reconfiguration process, the FPGA Manager gets as input the specified bitstream and loads it into the FPGA region's configuration memory through the CAP. Furthermore, the reconfiguration controller initiates the processes by setting the control signals and commands, that enable the reconfiguration. Reconfiguration is in the region without affecting the static region,

To reconfigure the Programmable Logic (PL) using the FPGA Manager, the following steps need to be followed:

1. Create the `/lib/firmware` directory: The first step is to create the `/lib/firmware` directory, which will serve as the storage location for the bitstream file. The bitstream file contains the necessary configuration data specific to the desired design that will be loaded onto the FPGA.
2. Copy the desired bitstream file: The next step involves copying the desired bitstream file to the `/lib/firmware` directory. The bitstream file contains the configuration data that is required to program the FPGA with the specific design. By executing the below command, the bitstream file named partial_bitstream.bin is copied to the designated directory.
3. Set the necessary flags: The FPGA Manager needs to be informed that a partial reconfiguration of the PL logic is desired. This is achieved by setting the appropriate flags in the FPGA Manager.
4. Load the bitstream file using the FPGA Manager: The final step involves loading the bitstream file onto the FPGA using the FPGA Manager module.

   This command instructs the FPGA Manager to upload a specified bitstream file into the FPGA. The FPGA Manager reads the file from the `/lib/firmware` directory and programs the corresponding reconfiguration region, that is specified in the bitstream content.

For convenience and ease of replication, a configuration file for each RM can be created. Below is an example of such a configuration script:

Listing 5.1: Script to perform partial reconfiguration

```
1 echo 1 > /sys/class/fpga_manager/fpga0/flags
2 mkdir -p /lib/firmware
3 cp /mnt/vivado_designs/partial_wrapper.bit.bin /lib/firmware
4 echo partial_wrapper.bit.bin > /sys/class/fpga_manager/fpga0/firmware
```

The steps in the configuration script are as follows:

- Set the flags to indicate a partial programming of the PL logic to the FPGA Manager.
- Create the necessary firmware folder.
- Copy the bitstream file (partial_bitstream.bin) to the firmware folder.
- Provide the bitstream file (partial_bitstream.bin) to the FPGA Manager module.

By following these steps and utilizing the configuration file, the FPGA Manager can be easily configured to perform partial reprogramming of the PL logic, granting the flexibility to dynamically update and optimize your FPGA design.

As mentioned before, and as it can be seen in the script the bitstream needs to be in the .bin format. The below script is used to do this, but first, it requires creating the BIF (boot image format) files for each bitstream that will be used as input to the bootgen command.

Listing 5.2: Script convert bitstreams into binary format

```
1  #!/bin/bash
2
3  # Define the input bitstream file names
4  input_bitstream_1="blackboxpartial.bif"
5  input_bitstream_2="full1.bif"
6  input_bitstream_3="full2.bif"
7  input_bitstream_4="cube2partial.bif"
8  input_bitstream_5="cube1partial.bif"
9  input_bitstream_6="blackbox.bif"
10
11 # Define the output binary file names
12 output_file_1="blackboxpartial.bit"
13 output_file_2="full1.bit"
14 output_file_3="full2.bit"
15 output_file_4="cube2partial.bit"
16 output_file_5="cube1partial.bit"
17 output_file_6="blackbox.bit"
18
19 # Run the bootgen command to convert the bitstreams
20 bootgen -image "$input_bitstream_1" -arch zynq
21 -process_bitstream bin -w -o "$output_file_1"
22
23 bootgen -image "$input_bitstream_2" -arch zynq
24 -process_bitstream bin -w -o "$output_file_2"
25
26 bootgen -image "$input_bitstream_3" -arch zynq
27 -process_bitstream bin -w -o "$output_file_3"
28
29 bootgen -image "$input_bitstream_4" -arch zynq
30 -process_bitstream bin -w -o "$output_file_4"
31
32 bootgen -image "$input_bitstream_5" -arch zynq
33 -process_bitstream bin -w -o "$output_file_5"
34
35 bootgen -image "$input_bitstream_6" -arch zynq
36 -process_bitstream bin -w -o "$output_file_6"
```

```
37
38 echo "Conversion completed."
```

5.3 Summary

The chapter discusses the process of dynamic partial reconfiguration using the Partial Configuration Access Port (PCAP) in Vivado IDE. It provides a summary of the RTL project flow in Vivado, including the creation of a top-level design and the creation of a custom IP. The Vivado software flow is also explained, highlighting the steps involved in the synthesis, assembly, floorplanning, implementation, and bitstream generation.

The chapter then focuses on the process of PL configuration through the PS using the PCAP interface. It describes the high-level flow of PL configuration through the PS and explains the steps involved in transferring the partial bitstream from the PS to the PL using PCAP. The process of enabling the PCAP interface, clearing interrupts, initializing the PL, disabling internal loopback, and transferring the bitstream file to the PL using DevC DMA are discussed.

Furthermore, the chapter introduced the concept of binary bitstreams and their specific format with header and metadata, which are essential for transferring the partial configuration from the PS to the PL through the PCAP interface. Lastly, it highlighted the FPGA Manager as a higher-level abstraction provided by Xilinx, enabling easier management and communication between the PS and PL during the reconfiguration process. The FPGA Manager eliminates the need for custom applications and simplifies the configuration process by handling bitstream transfer through the PCAP interface.

CHAPTER 6

Dynamic Partial Reconfiguration Testing

6.1 Introduction

This chapter focuses on the testing of the dynamic partial reconfiguration of FPGA using the CCSDS-123 compression algorithm for the HYPSO mission. The objective is to evaluate the functionality, performance, and effectiveness of the implemented system. Through a systematic testing approach, it is validated the successful execution of the dynamic partial reconfiguration process and assess the impact of the CCSDS-123 compression algorithm. Furthermore are discussed the steps involved, including directory structure organization, synthesis, assembly, floorplanning, configuration implementation, bitstream generation, and software application development using PCAP.

6.2 Experimental Setup

This section will describe the hardware and software environment utilized for testing and experimentation. The FPGA platform is described together with additional tools or equipment used, and also specific considerations or requirements for testing dynamic partial reconfiguration with the CCSDS-123 compression algorithm are outlined.

6.2.1 ZedBoard System Architecture

ZedBoard is a low-cost development board by Avnet with a rich set of peripherals and interfaces. The main part of Zedboard is the Zynq-7000 SoC (XC7Z020-CLG484-1), with a dual-core ARM Cortex-A9 processor with Xilinx programmable logic fabric.

The Zynq-7000 SoC provides a processing system (PS) and a programmable logic (PL) section. The processing system consists of the dual-core Cortex-A9 processor, peripherals, and memory interfaces. The programmable logic section, on the other hand, offers a reconfigurable hardware fabric that can be customized to meet specific application requirements. It includes programmable logic cells, DSP slices, block RAM, and peripheral interfaces. It has a capacity of 53200 LUTs, 106400 flip-flops, 220 DSP48E1s, and 140 BRAMs [11]. Furthermore, the Zynq device interfaces to a 256Mbit flash memory and 512MB DDR3 memory. The layout of ZedBoard by Avnet is given in Figure 6.1.

ZedBoard's system architecture also includes a wide range of peripherals and interfaces to facilitate connectivity and interaction with external devices. It features HDMI and VGA ports for video output, USB ports for connecting peripherals, Ethernet for networking, audio input/output, and various other interfaces such as GPIO, I2C, SPI, and UART.

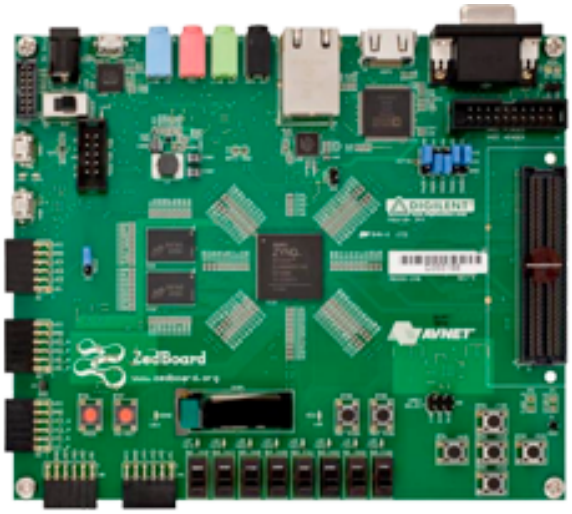


Figure 6.1: The layout of ZedBoard by Avnet

6.2.2 Additional Tools and Equipment

In addition to the ZedBoard, additional tools and equipment that facilitate the experimental setup are as below:

- Vivado Design Suite 2019.1 and 2022.2
  Vivado is used as a developing environment provided by Xilinx, a leading provider of Field-Programmable Gate Array (FPGA) and programmable logic solutions. Vivado provides a set of tools and features for FPGA design, implementation, and verification. Every year, Xilinx offers a new version of the Vivado design suite in order to improve and optimize its functionality. However, this is not very helpful, since if a design is developed using an older version, upgrading it to a newer version requires a careful examination of all changes and updates done to Vivado, in order to avoid any problems in the design functionality. The opu-system design was developed using version 2019.1 and that is why during testing this version was mostly used. However in order to ensure that the flow also works in the newer versions, the 2022.2 version was used as well.

- Vits SDK
  Vitis Software Development Kit (SDK) as described in the background chapter, is a development environment for designing and deploying software applications on Xilinx platforms. During testing, Vitis SDK was used mostly for verifying the functionality of modules after reconfiguration.

- Version Control tools (GitHub, Git)
  GitHub and Git are two other tools that have been used during this master thesis, and during testing specifically. Those tools help keep track of changes done in scripts and in the design through the testing procedure.

- Petalinux
  PetaLinux is a development environment and embedded Linux distribution provided by Xilinx specifically for designing and building custom Linux-based systems for Xilinx FPGAs and SoCs. This tool is used to create the boot files for running the Linux operating system on the board. Since Xilinx recommends using the same version as the Vivado Design Suite, PetaLinux 2019.1 is also used during testing. However, this tool requires a detailed examination and understanding of its functionalities, since the creation and building of the project will fail many times, without a clear explanation. So in order to have a smooth usage of this tool, it is recommended to stick to a specified flow and to keep track of any warnings or errors throughout the process.

6.2.3 Considerations for Testing Dynamic Partial Reconfiguration

Testing dynamic partial reconfiguration with the compression algorithm necessitates specific considerations to ensure accurate and reliable results. These considerations encompass various aspects of the testing process, including the configuration of the FPGA design, the validation of the compression algorithm's functionality, and the assessment of the reconfiguration infrastructure's performance. By addressing these factors, the effectiveness and robustness of the dynamic partial reconfiguration system can be thoroughly evaluated. Two main factors addressed are:

- Partial Reconfiguration Support
  To test dynamic partial reconfiguration with the CCSDS-123 compression algorithm, it is crucial to ensure that the FPGA design tools, such as Vivado, support the concept of dynamic reconfiguration. This includes the ability to generate the necessary partial bitstreams, handle the reconfiguration process, and provide a seamless interface between the reconfigurable regions and the rest of the design. It is essential to verify that the tools and infrastructure support the specific requirements of dynamic partial reconfiguration, allowing for the reconfiguration of specific regions while maintaining the functionality of the remaining FPGA fabric.
- Resource Utilisation
  The CCSDS-123 compression algorithm and the dynamic partial reconfiguration infrastructure both require FPGA resources to operate effectively. It is important to analyze and manage resource utilization during testing to ensure that the FPGA resources are appropriately allocated and utilized. This involves examining the resource usage of the CCSDS-123 compression algorithm and determining the additional resources required for the dynamic partial reconfiguration process. Careful resource planning and allocation are necessary to avoid conflicts or shortages that could impact the overall functionality and performance of the system. Thorough testing should include resource utilization analysis to ensure that the resources are allocated efficiently and that the system operates within the desired constraints.

By addressing these considerations and utilizing the appropriate tools and equipment, a robust experimental setup for testing dynamic partial reconfiguration with the CCSDS-123 compression algorithm is established. This setup enables the validation of the effectiveness and performance of the proposed approach and gathers meaningful results for analysis and evaluation.

6.3 Testing Methodology

The testing methodology employed in this thesis aimed to ensure a comprehensive evaluation and verification of the dynamic partial reconfiguration functionality. The following steps were incorporated into the testing process, each playing a crucial role in assessing the effectiveness and reliability of the implemented approach. A well-organized directory structure was established to manage the project files effectively. This structure facilitated the separation of source code, checkpoint files, synthesis scripts, bitstream files, and other relevant files. By maintaining a clear directory hierarchy, it was easier to track and manage the various components of the design, leading to efficient testing and debugging.

By incorporating these specific steps into the testing process, this methodology aimed to comprehensively evaluate and verify the dynamic partial reconfiguration functionality. Each step contributed to addressing different aspects of the design, from high-level synthesis to physical implementation, enabling a thorough assessment of the system's performance and effectiveness.

6.4 Directory Structure

To effectively manage the source files, netlists, bitstreams, and reports generated throughout the partial reconfiguration flow, a well-organized directory structure was established based on the

guidelines presented in [47]. The structure of the partial reconfigurable directory is outlined below:

- Bitstreams
  This folder serves as the repository for the bitstreams generated during the static and partial region configurations. It stores the compiled binary files that can be loaded onto the FPGA for reconfiguration.
- Implement
  The Implement folder acts as the target location for checkpoints and reports related to each design configuration. It also contains a separate folder for each configuration, ensuring a clear organization of results and facilitating easy access to specific configuration information and analysis.
- Sources
  This folder will contain the design source files that are used during the design. It consists of the HDL files and the constraints files for each of the reconfigurable regions and for each of the configurations.
  - hdl
    This subfolder contains the Hardware Description Language (HDL) files, such as Verilog or VHDL, required for the reconfigurable regions. Each configuration has its corresponding set of HDL files, ensuring a clean separation and management of design sources.
  - xdc
    The xdc subfolder contains the constraints files for each of the reconfigurable regions and their respective configurations. These files define the design constraints, such as pin assignments, clocking, and timing requirements, enabling accurate synthesis and implementation of the design.
- Synth
  The Synth folder stores the post-synthesis checkpoint files for all the modules within the design. These checkpoint files capture the state of the design after the synthesis stage and serve as an intermediate representation for subsequent implementation steps.
- TCL
  The TCL folder encompasses lower-level TCL scripts that are invoked by the top-level TCL scripts. These lower-level scripts handle specific tasks or subroutines required during the design process, contributing to the overall automation and manageability of the flow.
- design_complete.tcl
  This file serves as the master script that defines the design sources, parameters, and overall flow. It is responsible for orchestrating the execution of the entire design flow, from RTL to bitstream generation, ensuring a streamlined and consistent process.
- run_dfx.tcl
  In contrast to design_complete.tcl, run_dfx.tcl focuses solely on running the synthesis stage of the design flow. It encapsulates the necessary steps and settings to perform synthesis, generating the synthesized netlists as an intermediate output.

6.5 Synthesis and Assembly

6.5.1 Synthesis

The synthesis process in this project involves generating netlists for reconfigurable modules using the "run_dfx.tcl" script, which has been adapted from the guide provided in [47]. The script is responsible for creating the necessary files and saving them in the Synth folder.

The "run_dfx.tcl" script, adapted from [47], is used to initiate the synthesis process. Executing this script generates netlists for the reconfigurable modules, which are saved in the Synth folder.

In the project, a single reconfigurable partition named "ccsds123_top" is defined, which consists of three reconfigurable modules: cube_1, cube_2, and cube_3. Each module has specific dimensions:

- cube_1: X = 300, Y = 300, Z = 300
- cube_2: X = 250, Y = 250, Z = 250
- cube_3: X = 200, Y = 200, Z = 200

To ensure efficient synthesis for the embedded system, the "advanced_settings.tcl" script is used, which is customized to exclude sources related to the static part. Consequently, the section responsible for synthesizing the static part sources in the "run_dfx.tcl" script is commented out.

To initiate the synthesis process, the Vivado TCL Shell is opened using the following commands:

```
cd /tools/Xilinx/Vivado/2022.2/bin/
./vivado -mode tcl
```

Once the Vivado TCL Shell is open, the TCL script "run_dfx.tcl" is run, to start the synthesis and generate the output files in the Synth folder:

```
source run_dfx.tcl -notrace
```

The "-notrace" option suppresses the echoing of TCL commands, providing a cleaner output. By following these steps, the synthesis process will be initiated, and the netlists for the reconfigurable modules will be generated and saved in the designated Synth folder.

6.5.2 Assemble the Design

The assembly phase involves mapping the synthesized design onto the target FPGA platform, addressing the physical aspects of implementation such as logic element assignments and routing resources within the FPGA. Through assembly, the design's compatibility with the target device is validated, ensuring successful integration and functionality.

After generating the netlists for different configurations of the CCSDS-123 module, the next step is to connect these configurations, represented by separate netlists, with the black box in the embedded system design. This process requires loading the synthesized netlist checkpoints for each reconfigurable partition (RP) and defining the RP as partially reconfigured. Additionally, after each step, the assembled design state for the initial configuration is saved as a checkpoint.

To locate the path of the black box within the design, you need to open the netlist of the static part, which contains the black box component. By navigating through the netlist dialog box and following the hierarchy, you can identify the path to the black box module. For example:

```
System_i/ccsds123_ip_0/U0/ji
```

Once the path to the black box is identified, the design needs to be assembled. Next, the first synthesized checkpoint for the first reconfigurable module (RM) of the RP is loaded:

```
read_checkpoint -cell System_i/ccsds123_ip_0/U0/ji Synth/cube_1/
ccsds123_top_synth.dcp
```

After that, the RP is defined as partially reconfigurable using the below command:

```
set_property HD.RECONFIGURABLE 1 [get_cells System_i/ccsds123_ip_0/U0/ji]
```

This checkpoint for the first assembled state is saved.

```
write_checkpoint ./Checkpoint/top_cube_1.dcp
```

After execution of these steps, the design is assembled by connecting the synthesized netlist checkpoints for each reconfigurable module with the black box component. This process allows for the creation of an integrated design that combines the static and reconfigurable parts, enabling successful implementation and functionality on the target FPGA platform.

When the synthesis of the static part and the floorplanning for the RP exist, those steps can be run all that once. For this purpose, a bash script, `completeDPR.sh` was created in order to run those steps all at once.

6.6 Floorplanning

The floorplanning step involves defining the regions within the FPGA that will be used for partial reconfiguration. In this project, floorplanning was performed to determine the areas where partial reconfiguration would take place. Specifically, the Pblock (partial reconfiguration block) was defined in the XOYO and X1Y0 clock regions, as illustrated in Figure 6.2.

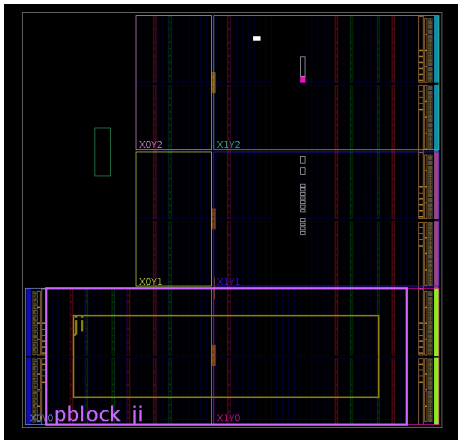


Figure 6.2: Selected area for black box ccsds123, instantiated as ji

The specific Pblock region was defined by considering various factors, including the warnings and errors given by the Design Rule Check (DRC) report. By analyzing the DRC report, any potential issues or conflicts related to the floorplanning could be identified and resolved. Additionally, the resource requirements of the reconfigurable modules (RMs) were taken into account to ensure sufficient space and resources within the defined Pblock. The floorplanning can be saved in the form of a constrain file (xdc) which defines the location and the specification of the floorplanning. Later, this file can be read in the project, which helps create an automated flow of creating the entire DPR project with all the necessary definitions.

6.7 Implementation of Configurations

In this step, the implementation of different configurations was performed through the place and route process. The design was processed using the opt_design, place_design, and route_design commands. These commands optimize the design, perform placement of the components, and establish the routing connections for each configuration. To ensure the reusability of the design with new reconfigurable modules (RMs), it was necessary to isolate the static design by removing the RM and locking all placement and routing. This was achieved using the following commands:

```
update_design -cell System_i/ccsds123_ip_0/U0/ji -black_box
lock_design -level routing
write_checkpoint -force Checkpoint/static_route_design.dcp
```

The update_design command removes the specified reconfigurable module (RM), in this case, "System_i/ccsds123_ip_0/U0/ji," from the design, treating it as a black box. The lock_design command locks all the components in their current placements and routes to preserve the static part of the design. Finally, the write_checkpoint command saves the checkpoint for the static-only design, named "static_route_design.dcp," which can be used for the other configurations.

6.8 Bitstream Generation

The final step in the testing methodology is the generation of the bitstream, which is used to program the FPGA with the desired configuration. The bitstream contains all the necessary information to configure the FPGA, including the partial reconfiguration regions and the associated bitstreams for each module. By generating the bitstream, the correct implementation of dynamic partial reconfiguration functionality on the FPGA can be verified. To generate the bitstream for the first configuration, the following was executed:

```
write_bitstream -file Bitstreams/Config_cube_1.bit
```

This command generates the full bitstream for the first configuration, which is saved as "Config_cube_1.bit" in the Bitstreams folder. The same procedure is performed for the second configuration as well. The generated bitstream is in the bit format. However, if the bitstream needs to be formatted as binary (BIN) and the interface is specified as SMAPx32, the following command can be used:

```
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -loadbit
"up 0x0 Bitstreams/Config_cube_1_pblock_ji_partial.bit" -file
Bitstreams/Config_cube_1_pblock_ji_partialu
```

This command formats the bitstream as binary (BIN) and specifies the interface as SMAPx32. It also disables the byte swap of the bitstream using the "-disablebitswap" option. The bitstream associated with the partial reconfiguration of the "ji" module within the "pblock" region for the first configuration is loaded and saved as "Config_cube_1_pblock_ji_partial.bin" in the Bitstreams folder. However, if the FPGA Manager interface is used, there is no need to manually perform the byte swap of the bitstream, as it is handled by the tool itself. In the board, the conversion of the bitstream to the binary file is done by the `bootgen` command.

6.9 CubeDMA Verification

This section will describe the steps performed to verify the CubeDMA module. It is important to note that the specifications done are related to the ZedBoard, that is the board used to test the partial reconfiguration. If the tests are performed on a different board, the specifications are not relevant, but they should be defined based on the board specification.

6.9.1 CubeDMA Kernel Driver

The CubeDMA kernel driver was designed in [32] and optimized in [46] to facilitate the management of the two character devices: one for sending data, the send channel, and the other for receiving data details. This section gives a high-level description of the CubeDMA kernel driver, and for a more technical explanation, the above-mentioned thesis should be consulted.

1. Initialization and Device Setup
   Firstly, when the driver is initialized it sets the two instances of the device structures, one for the send channel and another for the receive channel. Those two structures hold information about each device including the major number, device class, and the associated character. An important part of the initialization is the request of the memory regions for the send and receive channels to ensure that they have the necessary memory resources. The driver also initializes the character device structures and associates the relevant file operations with the devices. These file operations include functions for opening, reading from, writing to, and closing the devices, as well as memory mapping and I/O control operations. The driver registers a unique major number for each device, that serves as an identifier for the driver and its devices.

2. Memory Mapping and Cache Operations
   After the device is initialized, the driver maps the physical memory regions associated with the send and receive devices to the kernel space. In this case for the send channel, the 0x10000000 address is used, and for the receive channel the 0x18000000 address. This mapping allows the driver to directly access the device memory. Furthermore, to ensure data consistency and synchronization between levels of the memory hierarchy the drivers perform the cache operations. ache flushing operations ensure that any modifications made to the send and receive data buffers are propagated through the cache hierarchy. Cache invalidation operations make sure that subsequent data read from the receive buffer fetch the most up-to-date data from the physical memory.

3. Device Access and File Operations
   To enable user space applications to interact with the devices, the driver implements various file operations. These operations include reading from and writing to the devices, memory mapping the devices' physical memory to user space, and performing I/O control operations.

4. Cleanup and Resource Release
   During the cleanup process, the driver takes care of releasing all the resources that were allocated during initialization. This involves several important tasks, such as removing the devices themselves, unregistering the device class, freeing up the assigned major numbers, removing the character devices, and unmapping the memory regions. By performing these cleanup steps, the driver ensures that the system resources it utilized are properly released and can be utilized by other processes or drivers. This helps maintain a healthy and efficient system environment.

To integrate the custom kernel driver into the PetaLinux root file system, a module was created using the following command:

```
petalinux-create -t modules --name cubedma
```

This command generates a module recipe in the $< plnx - proj - root > /project - spec/meta - user/recipes - modules/cubedma$ directory. The next step involved editing the cubedma.c file to add the implementation of the custom CubeDMA kernel driver.

Once the boot files were generated and copied to the SD card for the target board, the module was inserted into the running kernel by executing one of the following commands:

```
insmod /lib/modules/<kernel version>/extra/cubedma.ko or modprobe cubedma
```

The insmod command loads the cubedma.ko module from the specified location ($/lib/modules/ < kernelversion > /extra/$) into the kernel, making it available for use. To remove the module from the kernel, the following command can be used:

```
rmmod /lib/modules/<kernel version>/extra/cubedma.ko or modprobe -r cubedma
```

The rmmod command unloads the cubedma.ko module from the kernel, freeing up system resources. By creating the module, adding the custom driver code, and inserting or removing the module using the appropriate commands, the custom kernel driver, in this case, the CubeDMA driver, can be seamlessly integrated into the PetaLinux system. This allows for the utilization of the driver's functionality in the target board's environment.

6.9.2 CubeDMA Reserved Area

Based on the insights gained from [46], it was observed that during the verification of the CubeDMA module, a kernel panic occurred during the booting process of the board. This unexpected kernel panic can be attributed to various factors such as memory conflicts or resource allocation issues. To mitigate the occurrence of such a situation, it became imperative to establish a reserved memory region within the RAM, ensuring that the specified range remains unutilized.

To accomplish this, two crucial steps were undertaken. Firstly, the amount of memory accessible to the Linux kernel was reduced, thereby creating a designated space for the reserved memory. By limiting the memory available to the kernel, the chances of any overlapping memory usage were minimized. This reduction in accessible memory was achieved through appropriate configuration adjustments.

The second step involved defining the reserved memory region in the device tree, thereby notifying the system about the dedicated memory area. The size of this reserved memory region can be determined based on the dimensions and requirements of the HSI cube, ensuring sufficient space for its operations. By explicitly specifying the reserved memory region in the device tree, potential conflicts or unintended usage of the designated memory range were avoided.

Below is given how the memory is reserved using the device tree.

Listing 6.1: Device Tree to reserve memory for CubeDMA and DimController modules

```
chosen{
    bootargs = "earlycon root=/dev/mmcblk0p2 rootfstype=ext4 rw
    rootwait clk_ignore_unused uio_pdrv_genirq.of_id=generic-uio";
    stdout-path = "serial0:115200n8";
};
reserved-memory{
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;

    reserved: buffer@0x10000000{
        reg = <0x10000000 0x10000000>;
        no-map;
    };
    reserved_dimcontroller: buffer@0x28000000{
        reg = <0x28000000 0x00200000>;
        no-map;
    };
};

cubedma: cubedma@43C00000{
    memory-region = <&reserved>;
    status = "okay";
};
dimcontroller: dimcontroller@43C10000{
    memory-region = <&reserved_dimcontroller>;
    status = "okay";
};
```

To incorporate the provided section into the system configuration of your Petalinux project, it is necessary to locate and modify the system-config.dtsi file. This file is typically found in the

$< petalinux-project-direcotry > /project-spec/meta-user/recipes-bsp/device-tree/files$ directory. By including this section in the system-config.dtsi file, it is ensured that the reserved memory region and the CubeDMA module configuration are incorporated into the device tree of the Petalinux project. This allows the system to recognize and utilize the reserved memory region and the CubeDMA module appropriately during system initialization and operation.

Another important part added to the device tree is the "chosen" section which is used to avoid kernel panic that can happen when the kernel is unable to mount the root filesystem specified during boot. The provided device tree section defines the "chosen" node, which sets global properties for system configuration. The "bootargs" property specifies boot arguments passed to the kernel during boot, controlling initial kernel configuration. The "stdout-path" property determines the output path for the system standard output, in this case directing it to the serial console. These properties configure boot behavior and console output settings for the system and it is used

6.9.3 Verifying Script

In order to ensure the accurate implementation of the partial reconfiguration using the CCSDS-123 compression algorithm, it is crucial to conduct a thorough verification of the CubeDMA module. This verification process involves the utilization of a verifying script that performs data movement operations within the memory, employing the DMA module to transfer data from one memory location to another. Subsequently, the script compares the contents of the two memory regions or locations, and if they are identical, it indicates a successful DMA transfer. Conversely, if the content of the two regions differs, it signifies an unsuccessful DMA transfer.

The script employed for verifying the CubeDMA module is based on a similar verifying script developed in previous works such as [32] and [46]. It can be accessed in the online repository of opu-systems. This script not only sets the CubeDMA control signal but also defines the addresses to be used, which should align with the addresses specified in the kernel driver mentioned in the preceding section. However, it is important to note that the script was initially tested on the ZedBoard platform. Consequently, the addresses were adjusted to accommodate the memory specifications and DDR size of the ZedBoard.

6.10 CCSDS-123 Verification

After ensuring that the CubeDMA module is working as expected, the next phase is to verify the CCSDS-123 module. The script defines the encoding configuration parameters and handles the compression. This step is important in order to check if the dynamic partial reconfiguration is performed as expected and has not affected the functionality of the design. The overall purpose of the script is to implement the image compression and encoding functionality, and after that, the dynamic partial reconfiguration is performed using different parameters.

Another approach to test the CCSDS-123 compression algorithm is by using the debugging cores in Vivado. In order to test it, the ILA core was used, by asserting the debugging property of the signals necessary to be debugged. Then a simple application was created in Vitis in order to initialize the PS. In order to start a transaction between the DMA and CCSDS-123 cores, two signals were added in the probe widow of ILA, respectively TVALID and TREADY. Then after running the c application in Vitis, the waveform is filled with all the values. Besides this, the internal signals were checked using the Set Up Debug tool in Vivado.

However, sometimes it was observed that the CubeDMA module sometimes was having unexpected behavior, and the S2MM stream channel was not able to complete, affecting thus also the behavior of the CCSDS-123 module. The reasons behind this can be the wrong data in the memory location or the input signals to this core are not properly isolated. Even though the FPGA Manager has the responsibility to decouple the RP from the static logic, this can also be ensured by using the DFX decouple IP, which is a hardware resource that controls the isolation of the RP from the hardware.

6.11 Reconfiguration Using FPGA Manager

As mentioned in the Implementation chapter, the tool that is used to reconfigure the FPGA is FPGA Manager. To use the FPGA Manager tool, the first step is to enable the tool in the configuration of the Petalinux project. This can be done using the Petalinux configuration GUI or using Petalinux commands. After that, the scripts with the content as explained in the background chapter were created. For example in order to reconfigure the PL using the first configuration of the RM the below script was used:

```
echo  1 > /sys/class/fpga_manager/fpga0/flags
mkdir -p /lib/firmware
cp /mnt/bitstreams/cube1partial.bit.bin /lib/firmware
echo cube1partial.bit.bin > /sys/class/fpga_manager/fpga0/firmware
```

For each bitstream file that was used to reconfigure the PL, a script was created. In order to see if the FPGA was configured as expected the dmesg command that shows the kernel logs in Linux can be used. After executing the script the below output is shown:
`fpga_manager fpga0: writing cube1partial.bit.bin to Xilinx Zynq FPGA Manager`

Even though FPGA Manager offers the readback of configuration registers for the reconfiguration process, it should be noted that Zynq devices do not support this capability, but this is supported in the UltraScale devices.

This chapter provided an overview of the testing methodology employed to evaluate dynamic partial reconfiguration for the CCSDS-123 compression algorithm. The experimental setup, including the hardware and software environment, was described. The importance of each testing step, such as synthesis, assembly, floorplanning, bitstream generation, and software application development, was emphasized. By following this methodology, the functionality, performance, and effectiveness of the implemented system can be thoroughly evaluated.

CHAPTER 7

Experimental Results

This section will include the experimental results taken from the tests conducted for the dynamic partial reconfiguration and the dimensionality reduction module in the ZedBoard. More specifically, the chapter will include the resource utilization, reconfiguration speed, and function density for the dynamic partial reconfiguration approach used during this master thesis. Based on those results, at the end of the chapter, an analysis of those results is conducted to identify the advantages and disadvantages of the test performed.

7.1 Resource Utilization

In embedded systems, resource utilization is a critical aspect of hardware design. It is important to strike a balance between minimizing resource usage and ensuring efficient functionality. The resource utilization for the CSDS-123 module was evaluated through testing on the ZedBoard platform using two different cube dimensions: 300x300x300 and 250x250x250. The resource utilization results for these configurations are presented in Table 7.1 and Table 7.3.

Table 7.1 shows the resource utilization summary for Configuration 1. The "Slice LUTs," "LUT as Logic," and "LUT as Memory" columns indicate the utilization of Look-Up Tables (LUTs) in different modes. The "Slice Registers" column represents the utilization of registers in the design. It can be observed that the utilization percentages for these resources range from 13.62% to 35.39%. The "F7 Muxes" and "F8 Muxes" columns show the utilization of multiplexers, with very low utilization percentages.

Similarly, Table 7.3 presents the resource utilization summary for Configuration 2. The utilization percentages for LUTs, registers, and multiplexers in this configuration range from 12.33% to 32.52%. Comparing the two configurations, it can be observed that Configuration 2 utilizes fewer resources in general, indicating better resource optimization.

In addition to the resource utilization summary, Tables 7.2 and 7.4 provide detailed information about the utilization of memory and DSP resources for Configuration 1 and Configuration 2, respectively. These tables show the utilization percentages for specific memory types (e.g., Block RAM Tile, RAMB36/FIFO) and DSP resources.

This approach allows the designer to have more control of the resources used by the reconfiguration partition. After each reconfiguration partition is synthesized, the reports regarding resource utilization are generated together with the necessary log files. So, based on those data and knowing the available resources on the board, the designer can make he necessary decisions for the image dimensions.

Resource Type	Used	Available	Utilization (%)
Slice LUTs	18,829	53,200	35.39
- LUTs as Logic	15,786	53,200	29.67
- LUTs as Memory	3,043	17,400	17.49
- LUTs as Distributed RAM	2,016	-	-
- LUTs as Shift Register	1,027	-	-
Slice Registers	14,493	106,400	13.62
- Registers as Flip Flop	14,493	106,400	13.62
- Registers as Latch	0	106,400	0.00
F7 Muxes	155	26,600	0.58
F8 Muxes	32	13,300	0.24

Table 7.1: Resource Utilization Summary for Configuration 1

3. Memory			
Resource Type	**Used**	**Available**	**Utilization (%)**
Block RAM Tile	49.5	140	35.36
RAMB36/FIFO*	44	140	31.43
RAMB18	11	280	3.93
4. DSP			
Resource Type	**Used**	**Available**	**Utilization (%)**
DSPs	28	220	12.73

Table 7.2: Resource Utilization Summary for Memory and DSP (Configuration 1)

7.2 Reconfiguration Speed

Besides resource utilization, another essential factor to consider for the DPR is the reconfiguration speed. The reconfiguration speed specifies how long the bitstream interchange can happen in FPGA, for different configurations. This section calculates the reconfiguration speed based on the provided data and analyses the size of the bitstream and how long it takes to perform the reconfiguration of FPGA.

The table 7.5 below shows the size of the different bitstreams used in the testing.

To calculate the reconfiguration speed, we need to measure the time it takes to perform the reconfiguration for each scenario. Let's denote the reconfiguration times as follows:

- T1: Time to reconfigure using the "Static + Partial Configuration logic", or the full bitstream.
- T2: Time to reconfigure using the partial bitstream.

Now, let's calculate the reconfiguration speeds for each scenario:

Reconfiguration speed for the full reconfiguration scenario:

$$Speed_1 = \frac{Size\ of\ bitstream}{T1} = \frac{4MB\ bytes}{180.61ms} = 22.14\ MB/s$$

Reconfiguration speed for the partial scenario:

$$Speed_2 = \frac{Size\ of\ bitstream}{T2} = \frac{2.6MB\ bytes}{82.3} = 31.59\ MB/s$$

By performing the reconfiguration and measuring the respective times, the actual values can be substituted into the above formulas to calculate the reconfiguration speeds for each scenario. Based on the experiment it was seen that the time taken to perform full reconfiguration was T1 = 180.61 ms and the time taken to perform partial reconfiguration was T2 = 82.3 ms.

Resource Type	Used	Available	Utilization (%)
Slice LUTs	17,302	53,200	32.52
LUTs as Logic	15,157	53,200	28.49
LUTs as Memory	2,145	17,400	12.33
Slice Registers	14,431	106,400	13.56
Registers as Flip Flop	14,431	106,400	13.56
F7 Muxes	151	26,600	0.57
F8 Muxes	32	13,300	0.24

Table 7.3: Resource Utilization Summary for Configuration 2

3. Memory			
Site Type	**Used**	**Available**	**Utilization (%)**
Block RAM Tile	37.5	140	26.79
RAMB36/FIFO*	32	140	22.86
RAMB18	11	280	3.93
4. DSP			
Site Type	**Used**	**Available**	**Utilization (%)**
DSPs	28	220	12.73

Table 7.4: Resource Utilization Summary for Memory and DSP (Configuration 2)

Bitstream	Size (MB)	Size (B)
Static + Configuration 1	4 MB	4,045,564 bytes
Static + Configuration 2	4 MB	4,045,564 bytes
Static + Black Box	4 MB	4,045,564 bytes
Configuration 1	2.6 MB	2,560,260 bytes
Configuration 2	2.6 MB	2,560,260 bytes

Table 7.5: Bitstream sizes for different configurations

7.3 Bitstream Upload Time Analysis

In the context of partial reconfiguration in an HYPSO mission, it is important to examine the differences between uploading the full bitstream and the partial bitstream. This section analyzes the implications of using partial reconfiguration and compares the upload times for both scenarios considering that the uplink speed of HYPSO-1 is limited to 150Kbps. To estimate the upload time for the full bitstream, we can utilize the formula:

$$\text{Upload Time (seconds)} = \frac{\text{Bitstream Size (bits)}}{\text{Uplink Speed (bits per second)}}$$

Considering an uplink speed of 150Kbps and a bitstream size of 4MB (4,000,000 bytes), the upload time for the full bitstream is calculated as follows:

$$\text{Upload Time (Full Bitstream)} = \frac{4,000,000 \times 8}{150,000} = 213.33 \text{ seconds}$$

Therefore, with an uplink speed of 150Kbps, it would take approximately 213.33 seconds or around 3 minutes and 33 seconds to upload the entire 4MB full bitstream. In contrast, when utilizing partial reconfiguration, the upload time can be significantly reduced by sending only the partial bitstream that contains the updated configuration information. The size of the partial bitstream is typically smaller than the full bitstream since it only includes the changes required for the reconfiguration.

Let's assume the size of the partial bitstream is 2MB (2,000,000 bytes). Using the same uplink speed of 150Kbps, the upload time for the partial bitstream can be calculated as follows:

$$\text{Upload Time (Partial Bitstream)} = \frac{2,000,000 \times 8}{150,000} = 106.67 \text{ seconds}$$

Therefore, with the same uplink speed, it would take approximately 106.67 seconds or around 1 minute and 47 seconds to upload the 2MB partial bitstream.

7.3.1 Comparison and Implications

The comparison between the full bitstream upload time and the partial bitstream upload time clearly highlights the advantage of using partial reconfiguration. By sending only the partial bitstream, the upload time can be reduced by approximately 50% compared to the full bitstream upload. This reduction in upload time has significant implications for HYPSO operations since it allows for faster reconfiguration and update of the FPGA configuration, enabling more agile and flexible in-orbit operations. Furthermore, the reduced upload time minimizes the impact on limited uplink resources, such as bandwidth and power, which are valuable and constrained in CubeSat space missions. Another option to further optimize the reconfiguration process is to store different bitstreams with various configurations in the SD card. By having pre-loaded bitstreams available, during the reconfiguration it can swiftly switch between configurations as needed, avoiding the need for bitstream uploads altogether. This approach eliminates the time and resources required for bitstream uploads, providing instant reconfiguration capabilities.

7.4 Power Consumption

In the context of CubeSats, power consumption is an important consideration alongside reconfiguration speed. Power consumption in reconfigurable computing systems comprises static and dynamic power. Static power accounts for the leakage current in the transistors, while dynamic power involves the switching of transistors. Theoretically, the power consumption should decrease during partial reconfiguration since the partial bitstream is up to 50% smaller, resulting in a shorter reconfiguration time. However, this reduction depends on additional factors such as the efficiency of the reconfiguration process and the power characteristics of the FPGA itself. Therefore, it is crucial to conduct a thorough analysis and measure power consumption during the partial reconfiguration process to accurately assess the potential power savings.

Firstly, theoretically, the reduction in power consumption can be achieved if the configuration throughput is high enough to reduce the energy overhead of the partial reconfiguration. Based on the above section, on configuration speed, it was seen that the reconfiguration time is a function of bitstream size and the reconfiguration throughput. So the necessary throughput in order to reduce the power consumption should be greater than the product of reconfiguration time and the size of the bitstream, taking also into consideration the power consumption of the CCSDS-123 while this module is not being used.

Secondly, since a main portion of the power consumption is static power, reducing this will have an immediate effect on the power consumption. So, in the cases when the CCSDS-123 algorithm is not being used, the FPGA can be configured using a blank bitstream, that removes the logic from the reconfiguration partition. In order to support this, a simple theoretical power estimation was done for three types of configurations respectively: configuration 1, configuration 2, and black box configuration as shown in Graph 7.1. To get those data was used the Power Estimator tool, provided by Vivado, that estimates the power estimation of the digital design.

As seen in the graph the total on-chip power is highly reduced when using the back box configuration. Furthermore, between the two configurations, the one with smaller cube dimensions has less power consumption.

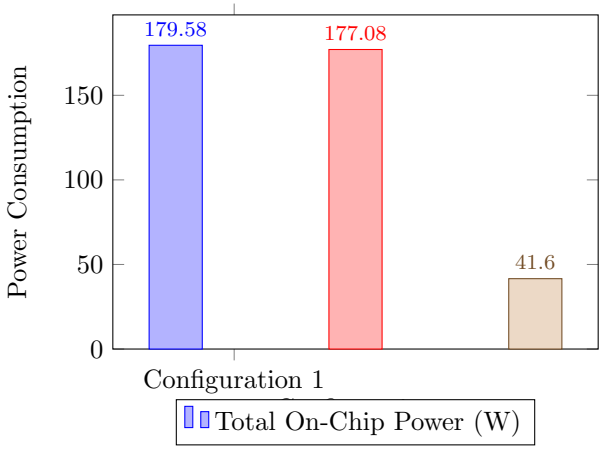


Figure 7.1: Total On-Chip Power Consumption for Different Configurations

7.5 Conclusion

In conclusion, this section presented the experimental results obtained from the tests conducted for the dynamic partial reconfiguration (DPR) and dimensionality reduction module in the ZedBoard. The resource utilization, reconfiguration speed, and function density were evaluated to assess the effectiveness and efficiency of the DPR approach.

The resource utilization analysis revealed the utilization percentages for various resources such as Slice LUTs, Slice Registers, F7 Muxes, and F8 Muxes. Two different configurations, Configuration 1 and Configuration 2, were compared, indicating that Configuration 2 utilized fewer resources, suggesting better resource optimization. Detailed information about the utilization of memory and DSP resources was also provided, highlighting the significance of these resources in the design.

Furthermore, the reconfiguration speed was measured for different scenarios, including full reconfiguration using the full bitstream and partial reconfiguration using the partial bitstream. The calculated reconfiguration speeds indicated that the partial reconfiguration scenario achieved a higher speed, demonstrating the advantages of dynamic partial reconfiguration in terms of faster reconfiguration times.

Overall, the experimental results presented in this chapter provide valuable insights into the performance and efficiency of the dynamic partial reconfiguration and dimensionality reduction module implemented on the ZedBoard. These results serve as a basis for further analysis and discussion in the subsequent chapters, enabling a comprehensive evaluation of the advantages and disadvantages of the tested approaches.

CHAPTER 8

Conclusion

8.1 Summary

The main focus of this thesis was the exploration of the dynamic reconfiguration of FPGA, with a specific emphasis on the applications in the context of hyperspectral remote sensing in HYPSO-1. The objective was to implement an approach for the on-flight FPGA reconfiguration, particularly utilizing the CCSDS-123 compression algorithm. The thesis also aimed to enhance the versatility and applicability of the algorithm by modifying its implementation to support the compression of different types of images. Additionally, the thesis aimed to develop a comprehensive dataflow model for dynamic reconfiguration, focusing on partial reconfiguration using the CCSDS-123 algorithm.

The thesis began with an introduction to the motivation behind CubeSats and their increasing utilization in space missions, as well as the significance of hyperspectral imaging and processing in various domains. The HYPSO mission, which serves as the basis for this thesis, was introduced, along with the concept of dynamic reconfiguration in FPGA-based systems. The thesis was built upon a previous specialization project that focused on the onboard image processing pipeline and on-flight FPGA reconfiguration for hyperspectral remote sensing CubeSats.

The main contributions of this thesis include an analysis of the CCSDS-123 compression algorithm and its modification to support various image types. A custom AXI4-Lite component was implemented to enable efficient modification of the hyperspectral imaging (HSI) cube's Y-dimension without full FPGA reconfiguration. Furthermore, a systematic partial reconfiguration flow for the CCSDS-123 algorithm was developed, allowing module interchangeability without affecting other modules. Testing and validation of the proposed approach demonstrated its functionality and effectiveness in the context of hyperspectral remote sensing CubeSats, particularly in the HYPSO mission.

8.2 Conclusion

Through the exploration and implementation of dynamic partial reconfiguration using the CCSDS-123 compression algorithm, this thesis has achieved to highlight the benefits of using partial reconfiguration flow in the CubeSat missions, specifically for the HYPSO mission. First and foremost, the modification of the CCSDS-123 compression algorithm has enhanced its versatility and applicability by enabling the compression of different types of images. This modification increases the flexibility and adaptability of the algorithm for various use cases. The implementation of a custom AXI4-Lite component for changing the Y-dimension of the hyperspectral imaging (HSI) cube has been successfully achieved. This component allows for efficient and flexible modification of the cube size dimension without the need for full reconfiguration of the FPGA. Secondly, the developed partial reconfiguration flow for the CCSDS-123 compression algorithm provides a systematic methodology for performing dynamic on-flight reconfiguration of the FPGA. Furthermore, it should be noted that the testing of this approach was done using a small HSI cube in order

to fulfill the floorplanning requirements in Zynq devices. Another approach will be to reduce the abstraction level, by introducing smaller RP inside the CCSDS-123 IP, allowing thus more control of the reconfiguration process. This flow enables the interchange of different modules in the FPGA without affecting the functionality of other modules. The testing and validation conducted in this thesis have demonstrated the functionality and effectiveness of the proposed approach. The results obtained from testing and validation indicate that dynamic partial reconfiguration is a viable solution for hyperspectral remote sensing CubeSats, offering advantages in terms of reconfiguration speed, resource utilization, and adaptability.

8.3 Future Work

While this thesis has made significant progress in the field of dynamic partial reconfiguration for space applications, there are several avenues for future work and further exploration. Some potential directions for future research include:

- **Optimization of the partial reconfiguration flow**
  Further optimization of the partial reconfiguration flow can be investigated to improve the efficiency and performance of the FPGA reconfiguration process. This could involve exploring different compression algorithms or techniques to reduce the size of the partial bitstream and enhance the reconfiguration speed.
- **Integration of additional functionalities**
  The developed framework for dynamic partial reconfiguration can be expanded to include other functionalities or algorithms relevant to space applications. This could involve integrating modules for image processing, feature extraction, or classification, enabling a more comprehensive onboard processing pipeline while keeping resource utilization and power consumption as low as possible.
- **Developing a custom controller based on ICAP**
  Investigation into the development of a custom controller utilizing the ICAP controller can be explored. This utilizes the benefits of the configuration from hardware and also offers more flexibility and customization options.
- **Exploration of other reconfigurable hardware technologies**
  While this thesis focused on FPGA-based systems, future work could explore the potential of other reconfigurable hardware technologies, such as ASIC-based reconfigurable systems or hybrid approaches. Comparisons and evaluations of different reconfigurable hardware technologies would contribute to a more comprehensive understanding of their strengths and limitations.
- **Propose a solution for a secure dynamic reconfiguration** Future work can focus on adding security features to dynamic reconfiguration. Developing techniques and protocols to ensure the integrity and confidentiality of partial bitstreams and secure the reconfiguration process would be valuable in space applications and in HYPSO mission specifically.
- **Optimized and efficient floorplanning techniques** Investigating floorplanning techniques specific to dynamic partial reconfiguration is an important step to meet specific timing and area requirements. This could include investigating partitioning approaches in order to support a more flexible floorplanning process for a number of RP.

Bibliography

[1] Stanford. *CubeSat: The little satellite that could*. June 2022. URL: https://engineering.stanford.edu/magazine/cubesat-little-satellite-could.

[2] Kizheppatt Vipin and Suhaib A. Fahmy. 'FPGA Dynamic and Partial Reconfiguration'. In: *ACM Computing Surveys* 51.4 (July 2018), pp. 1–39. DOI: 10.1145/3193827. URL: https://doi.org/10.1145/3193827.

[3] *Space Mission Engineering*. Microcosm Press, Jan. 2011.

[4] Nasa. *CubeSat 101 Basic Concepts and Processes for First-Time CubeSat Developers*. 2017.

[5] Hans F. Grahn and Paul Geladi, eds. *Techniques and Applications of Hyperspectral Image Analysis*. Wiley, Sept. 2007. DOI: 10.1002/9780470010884. URL: https://doi.org/10.1002/9780470010884.

[6] Liguo Wang and Chunhui Zhao. *Hyperspectral Image Processing*. Springer Berlin Heidelberg, 2016. DOI: 10.1007/978-3-662-47456-3. URL: https://doi.org/10.1007/978-3-662-47456-3.

[7] Joar Andreas Gjersund. 'Efficient Streaming and Compression of Hyperspectral Images'. 2018. URL: http://hdl.handle.net/11250/2566932.

[8] Biagio Peccerillo et al. 'A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives'. In: *Journal of Systems Architecture* 129 (Aug. 2022), p. 102561. DOI: 10.1016/j.sysarc.2022.102561. URL: https://doi.org/10.1016/j.sysarc.2022.102561.

[9] Joao Manuel Paiva Cardoso, Jose Gabriel de Figueired Coutinho and Pedro C Diniz. *Embedded computing for high performance*. Oxford, England: Morgan Kaufmann, June 2017.

[10] Vansteenkiste, Elias. 'New FPGA design tools and architectures'. eng. PhD thesis. Ghent University, 2016, XXIII, 237. ISBN: 9789085789604.

[11] Louise H Crockett et al. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, July 2014.

[12] *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*. 2022. URL: https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm/.

[13] Bekkay Hajji, Adel Mellit and Loubna Bouselham. *A Practical Guide for Simulation and FPGA Implementation of Digital Design*. Springer Singapore, 2022. DOI: 10.1007/978-981-19-0615-2. URL: https://doi.org/10.1007/978-981-19-0615-2.

[14] *Vivado Design Suite AXI Reference Guide*. 2015. URL: https://xilinx.eetrend.com/files-eetrend-xilinx/download/201706/11565-30612-ug1037-vivado-axi-reference-guide1.pdf.

[15] *Vivado Design Suite User Guide*. 2022. URL: https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_1/ug892-vivado-design-flows-overview.pdf.

[16] *Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400)*. 2022. URL: https://docs.xilinx.com/r/en-US/ug1400-vitis-embedded/Zynq-UltraScale-MPSoC-Boot-and-Configuration.

[17] *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*. 2021. URL: https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis.

[18] Alan Holt and Chi-Yu Huang. *Embedded Operating Systems*. Springer International Publishing, 2018. DOI: 10.1007/978-3-319-72977-0. URL: https://doi.org/10.1007/978-3-319-72977-0.

[19] Janez Puhan. *Operating systems, embedded systems and real-time systems*. en. 2015.

[20] *Yocto Project: Building a small, custom Linux for IoT devices*. 2018. URL: https://developer.ibm.com/tutorials/l-yocto-linux/.

[21] *Yocto Project Documentation*. Yocto Project. 2023. URL: https://docs.yoctoproject.org/index.html.

[22] Drew Moseley. *Why the Yocto Project for my IoT project?* 2017. URL: https://www.embedded.com/why-the-yocto-project-for-my-iot-project/.

[23] *PetaLinux Tools Documentation: Reference Guide*. 2022. URL: https://docs.xilinx.com/r/en-US/ug1144-petalinux-tools-reference-guide.

[24] Alessandro Rubini, Jonathan Corbet and Greg Kroah-Hartman. *Linux Device Drivers*. en. 3rd ed. Sebastopol, CA: O'Reilly Media, Feb. 2005.

[25] The Linux Kernel. *Linux and the Devicetree*. URL: https://docs.kernel.org/devicetree/usage-model.html.

[26] *Zynq UltraScale+ MPSoC Software Developer Guide (UG1137)*. 2022. URL: https://docs.xilinx.com/r/en-US/ug1137-zynq-ultrascale-mpsoc-swdev/Detailed-Boot-Flow.

[27] Jonathan Corbet, Alessandro Rubini and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005. ISBN: 0596005903.

[28] *FPGA Manager - Linux Kernel Documentation*. URL: https://www.kernel.org/doc/html/v4.19/driver-api/fpga/fpga-mgr.html.

[29] Alan Tull. *Reprogrammable Hardware under Linux*. ELC Dublin. 2015.

[30] R C Cofer, Rc Cofer and Benjamin F Harding. *Rapid system prototyping with FPGAs*. en. Embedded Technology. Newnes, May 2014.

[31] Joar Andreas Gjersund. 'A Reconfigurable Fault-Tolerant On- Board Processing System For The HYPSO CubeSat'. 2020. URL: https://hdl.handle.net/11250/2778120.

[32] Andreas Varntresk. 'Assembly and testing of baseline processing chain'. 2019. URL: http://hdl.handle.net/11250/2624678.

[33] S Cenk Sahinalp and Nasir M Rajpoot. 'Dictionary-Based Data Compression'. In: *Lossless Compression Handbook*. Elsevier, 2003, pp. 153–167.

[34] CCSDS: The consultative Committee for Space and Data Systems. *LOSSLESS DATA COMPRESSION*. Aug. 2020. URL: https://public.ccsds.org/Pubs/121x0b3.pdf.

[35] CCSDS: The consultative Committee for Space and Data Systems. *Low-complexity lossless and near-lossless multispectral image compression*. Dec. 2022. URL: https://public.ccsds.org/Pubs/120x2g2.pdf.

[36] Xilinx. *Partial Reconfiguration User Guide*. 2013. URL: https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx14_5/ug702.pdf.

[37] *Vivado Design Suite User Guide: Design Analysis and Closure Techniques (UG906)*. 2021. URL: https://docs.xilinx.com/r/2021.1-English/ug906-vivado-design-analysis.

[38] Simen Gimle Hansen, Dirk Koch and Jim Torresen. 'High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro'. In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, May 2011. DOI: 10.1109/ipdps.2011.139. URL: https://doi.org/10.1109/ipdps.2011.139.

[39] L.A. Cardona et al. 'Performance-Area Improvement by Partial Reconfiguration for an Aerospace Remote Sensing Application'. In: *2011 International Conference on Reconfigurable Computing and FPGAs*. 2011, pp. 497–500. DOI: 10.1109/ReConFig.2011.69.

[40] John C. Hoffman and Marios S. Pattichis. 'A High-Speed Dynamic Partial Reconfiguration Controller Using Direct Memory Access Through a Multiport Memory Controller and Overclocking with Active Feedback'. In: *International Journal of Reconfigurable Computing* 2011 (2011), pp. 1–10. DOI: 10.1155/2011/439072. URL: https://doi.org/10.1155/2011/439072.

[41] Kizheppatt Vipin and Suhaib A. Fahmy. 'ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq'. In: *IEEE Embedded Systems Letters* 6.3 (Sept. 2014), pp. 41–44. DOI: 10.1109/les.2014.2314390. URL: https://doi.org/10.1109/les.2014.2314390.

[42] Amit Kulkarni, Vipin Kizheppatt and Dirk Stroobandt. 'MiCAP: a custom reconfiguration controller for dynamic circuit specialization'. In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, Dec. 2015. DOI: 10.1109/reconfig.2015.7393327. URL: https://doi.org/10.1109/reconfig.2015.7393327.

[43] Wang Guohua et al. 'A tiny and multifunctional ICAP controller for dynamic partial reconfiguration system'. In: *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE, July 2017. DOI: 10.1109/ahs.2017.8046361. URL: https://doi.org/10.1109/ahs.2017.8046361.

[44] Kizheppatt Vipin and Suhaib A. Fahmy. 'DyRACT: A partial reconfiguration enabled accelerator and test platform'. In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Sept. 2014. DOI: 10.1109/fpl.2014.6927507. URL: https://doi.org/10.1109/fpl.2014.6927507.

[45] Bushra Sultana et al. 'VR-ZYCAP: A Versatile Resourse-Level ICAP Controller for ZYNQ SOC'. In: *Electronics* 10.8 (Apr. 2021), p. 899. DOI: 10.3390/electronics10080899. URL: https://doi.org/10.3390/electronics10080899.

[46] Christoffer Boothby. 'An implementation of a compression algorithm for hyperspectral images. A novelty of the CCSDS 123.0-B-2 standard'. 2020. URL: https://hdl.handle.net/11250/2778129.

[47] *Dynamic Function eXchange: Vivado Design Suite User Guide*. 2022. URL: https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0017-vivado-partial-reconfiguration-hub.html.

Tutorial

This tutorial will describe the steps to create a Pealinux project and also how to prepare the SD-Card for booting. The purpose of this tutorial is also to document the errors that were faced during the testing phase.

Petalinux

1. Installation

1. Installation Steps
   The installer for PetaLinux can be downloaded from the Xilinx download page. In this case, version 2022 was downloaded. PetaLinux installer can be installed in the desired directory by running the below command:

   ```
   ./petalinux-v<petalinux-version>-final-installer.run
   ```

2. BSP (Board Support Packages) Installation
   Xilinx together with PetaLinux provides BSP (Board Support Packages) in order to start working and customize the project, in our case for Zedboard. The packages include configuration files, pre-built and tested hardware, and software images. H

After the PetaLinux tool and BSP installation, the next step is the creation and specification of the hardware platform which is explained in the upcoming subsection.

2. Project Creation and Configuration

1. Project Customisation
   The project can be created using the below command:

   ```
   petalinux-create -t project -s <path-to-bsp>
   ```

   However, during testing in the Zedboard, the boot files were not created properly. For this reason, the method with the BSP was not used, but instead, the method using templates was utilized as below:

   ```
   petalinux-create --type project --template zynq --name zedboard
   ```

   Then the next states are the same for both methods.

2. Importing Hardware Configuration The example hardware configuration is imported into the created project. The first step is to change into the directory of created PetaLinux project and then import the hardware description by giving the path of the directory containing the .xsa file as follows:

   ```
   petalinux-config --get-hw-description <PATH-TO-XSA Directory>
   ```

   After running this command, the tool detects the changes in the imported hardware design and launches the system configuration menu. During this step, the FPGA Manager and the device tree overlay were enabled by using the system configuration menu. It is important to include bootgen package in the PetaLinux project. The packages in PetaLinux can be installed using the below command that opens the menu to select a specific package to be added to the PetaLinux project.

   $$petalinux - config - c\ rootfs$$

   Some other packages that are necessary to be added to the petalinux project in order to add the GCC and other development tools. Those packages are packagegroup-core-buildessential and packagegroup-core-buildessential-dev. Those packages can be enabled by going to FileSystemPackages -¿ misc -¿ packagegroup-core-buildessentail

3. Project Build

3. Build PetaLinux System Image
   The project is built by the following command:

   $$petalinux - build$$

   This command generates a DTB file, the first stage boot loader, U-Boot, Linux kernel, a root file system image, and generates the boot images.

4. Generating Packaging Boot Image The following command generates the boot image in .bin format:

   ```
   petalinux-package --boot --fsbl images/linux/zynq_fsbl.elf
   --fpga images/linux/system.bit --u-boot
   ```

   The –u-boot adds all required images to boot up to U-Boot into BOOT.BIN and –format

5. Notes

Some things to be considered while testing the dynamic partial reconfiguration for the ccsds123 module are:

First, it is very important to check the version with which the project was created, since Xilinx updates its products a lot throughout the years, it is important to check all the changes in its products.

One issue found, while using Petalinux 2019.1 was that this version of Petalinux is based on a 2.x version of Python, so it should also execute in the environment with the same Python version. This error was seen while trying to configure the project, and it was not able to complete the step. So in order to resolve this step, a tool named Anaconda was created with the correct version of Python needed by Petalinux. The environment can be activated using the below command:

```
conda activate petalinux
```

Another error faced during testing is related to the XPM library. When the script synthesis of the reconfigurable modules was run in Vivado 2019.1, the error was that the XPM library was not found. This error was removed by adding the below command in the synthesize.tcl script:

```
set_property XPM_LIBRARIES {XPM_CDC XPM_MEMORY} [current_project]
```

However, even though the library was found, the tool was not able to bind the i_fifo to the hdl files, and thus it was considered as black box.

Preparing SD-Card

After building the project and creating the images, the next step is to put those files on SD Card. In order to prepare the SD Card the following steps were performed:

- Un-mount SD Card
  First, the card should be un-mounted, and if it has been used before it is recommended to delete the existing partitions.
- Create partition
  Three partitions were created: BOOT, of type fat32, with the size 1GB, FAT (type fat32) with the size 2GB, and roofs, of type ext4, with the remaining space.
- Set Flags
  In the boot partition is required to set the boot and lba flags.
- Root File System Creation
  Then the rootfs.tar.gz should be extracted in rootfs partition on SD Card. For example:

  $$tar - xf\ rootfs.tar.gz\ \ - C\ /media/"name"/rootfs$$

  After this step, the SD Card is ready to be used for booting the hardware device.

Debugging using Vivado Logic Analyzer

This section will describe how to use the ILA IP provided by Xilinx to debug a specific IP, which in this case is the CubeDMA module. Integrated Logic Analyzer (ILA) is an IP core that can be used to monitor the internal signals of a design. There are two ways to use the ILA IP, the manual and automatic approaches. This section will describe the automatic approach.

- Mark the necessary interfaces or nets that need to be probed using the Debug option
  This can be done by right-clicking on the interface and selecting Debug from the context menu.
- Run connection Automation
  After the interfaces that need to be debugged are defined, the next step is to run the connection automation which will include in the block design the ILA IP.
- Open Hardware Manager
  After synthesis, implementation, and bitstream generation the hardware manager is open in order to see the ILA waveform. However since the PS is not configured, Vivado will not recognize the ILA component. So in order to reconfigure the PS, a simple Vitis application can be created for this purpose.

Another approach can be to use the Setup As Debug in the Synthesis Desing Mode when in the debug window the internal signal can be added also in order to see their values.

After setting the PS using the Vitis application, it is necessary to probe the necessary signals in the ILA, in order to initiate a transaction. For example, the ready and valid signals can be added in the probe section of the ILA.

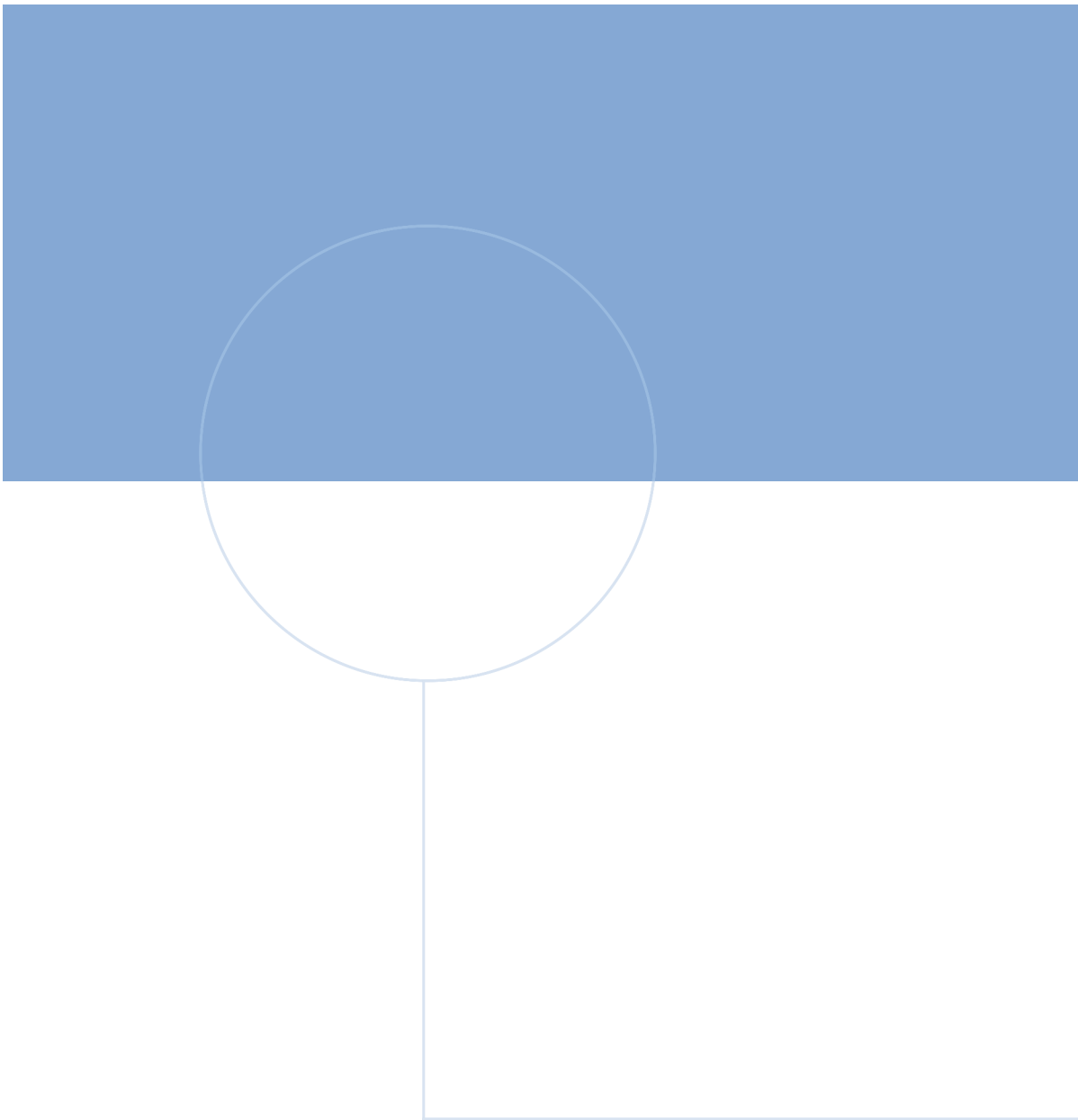
Appendix

A Partial Reconfiguration Synthesize TCL Script

B Reconfiguration using the devcfg library

C DPR Github Repository

The DPR project can be found in the opu-system repository of HYPSO mission: opu-system



NTNU

Norwegian University of
Science and Technology