



NUST COLLEGE OF
ELECTRICAL AND MECHANICAL ENGINEERING



**CENTRAL BANKING DIGITAL CURRENCY
TRANSACTION PROCESSOR**

PROJECT REPORT

DE-42 (DC & SE)

Submitted by

NS FURQAN AHMAD
NS HAMZA BIN SAQIB
NS ALI NAWAB RANA
NS SANAULLAH AFZAL

BACHELORS

IN

COMPUTER ENGINEERING

YEAR

2024

PROJECT SUPERVISOR

DR. ALI HASSAN
DR. SHOAB AHMED KHAN

DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING
NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY,
ISLAMABAD, PAKISTAN

Certification

This is to certify that Furqan Ahmad(352076), Hamza Bin Saqib(342765), Ali Nawab Rana(340770) and Sanaullah Afzal(346463) have successfully completed the final project Central Banking Digital Currency Transaction Processor, at the National University of Science and Technology Islamabad, to fulfill the partial requirement of the degree Computer Engineering.



Signature of Project Supervisor
Dr. Ali Hassan
Designation

Sustainable Development Goals (SDGs)

SDG No	Description of SDG
SDG 9	Decent Work and Economic Growth



Sustainable Development Goals

Complex Engineering Problem

Range of Complex Problem Solving

	Attribute	Complex Problem	
1	Range of conflicting requirements	Involve wide-ranging or conflicting technical, engineering and other issues.	✓
2	Depth of analysis required	Have no obvious solution and require abstract thinking, originality in analysis to formulate suitable models.	✓
3	Depth of knowledge required	Requires research-based knowledge much of which is at, or informed by, the forefront of the professional discipline and which allows a fundamentals-based, first principles analytical approach.	✓
4	Familiarity of issues	Involve infrequently encountered issues	✓
5	Extent of applicable codes	Are outside problems encompassed by standards and codes of practice for professional engineering.	
6	Extent of stakeholder involvement and level of conflicting requirements	Involve diverse groups of stakeholders with widely varying needs.	✓
7	Consequences	Have significant consequences in a range of contexts.	✓
8	Interdependence	Are high level problems including many component parts or sub-problems	✓

Range of Complex Problem Activities

	Attribute	Complex Activities	
1	Range of resources	Involve the use of diverse resources (and for this purpose, resources include people, money, equipment, materials, information and technologies).	✓
2	Level of interaction	Require resolution of significant problems arising from interactions between wide ranging and conflicting technical, engineering or other issues.	✓
3	Innovation	Involve creative use of engineering principles and research-based knowledge in novel ways.	✓
4	Consequences to society and the environment	Have significant consequences in a range of contexts, characterized by difficulty of prediction and mitigation.	✓
5	Familiarity	Can extend beyond previous experiences by applying principles-based approaches.	✓

Dedicated to

the Prophet Muhammad (SAW), the mercy for all people. May his messages of love, justice, and dignity remain relevant in our society. It also honors the brave and resilient people of Gaza, who inspire many with their courage and persistence.

Acknowledgment

First of all, Alhamdulillah, that our FYP is finally made and all Thanks to Allah for giving us the strength and moral to keep pushing forward and helping us on each and every step of the way.

Secondly, we would like to offer heartily thanks our supervisors, Dr. Ali Hassan and Dr. Shoab Ahmed Khan, who helped us a lot, tremendously, on each and every single issue, who's help ad guidance became a source of strong determination for us. Thank You, sir's you played a great role in our lives, one that we can never forget.

And lastly, we would like to thank our parents and friends, without whose unimaginable support and constant motivation, we might not have been able to complete our Final year project. They played an unparalleled role throughout our journey and we are eternally thankful to them. Their constant support, motivated us to do more than we ever realized and they inspired new hope in us, when we found none in ourselves.

Abstract

A safe, effective, and scalable transaction processing system is now essential due to the growing number of countries implementing Central Bank Digital Currencies (CBDCs). Even with their robustness, the current financial systems have difficulties scalability, security, and transaction speed while adjusting to the digital currency paradigm. Existing blockchain technologies, like Hyperledger Fabric, provide decentralized, transparent, and secure answers to these problems. However, performance and efficiency are severely constrained by the substantial processing overhead associated with handling transactions and carrying out smart contracts. In order to overcome these obstacles, the final year project develops a CBDC transaction processor that uses a ZYBO board for increased processing power and integrates Hyperledger Fabric as the underlying blockchain technology. The permissioned, modular network architecture of Hyperledger Fabric is perfect for preserving the scalability, integrity, and secrecy needed for CBDC transactions. To increase overall efficiency, compute-intensive tasks like cryptographic procedures and the execution of smart contracts are offloaded from the main blockchain network and handled by the ZYBO board, a flexible and potent processing unit. The project's goal is to show how the ZYBO board and Hyperledger Fabric together can greatly increase transaction throughput and decrease latency. According to the system architecture, the ZYBO board functions as the Hyperledger Fabric network's central processing unit, effectively carrying out intricate tasks. To verify the efficacy of the system, performance measures including processing latency and transactions per second (TPS) rates are subjected to extensive testing and analysis. Comprehensive testing is also used to assess the security and dependability of the suggested solution against various cyberthreats and failure scenarios. This study not only contributes to the actual deployment of digital currencies but also explores the wider possibility of integrating blockchain technology with specialized hardware accelerators by bridging the gap between current financial systems and the future needs of CBDCs. The results offer insightful information for future advancements in blockchain technology and digital currency transaction platforms that will improve financial services.

Contents

Acknowledgment	v
Abstract	vi
List of Figures	x
List of Tables	xii
1 Introduction	1
1.1 Motivation	1
1.2 Problem Statement	3
1.3 Scope	4
1.4 Aims and Objectives	5
1.4.1 Aims	5
1.4.2 Objectives	6
1.5 Outcomes	7
1.5.1 Expected Outcomes	7
1.5.2 Potential Impact	8
1.6 Report Organization	8
2 Literature Review	10
2.1 Background	10
2.1.1 Central Banking Digital Currencies (CBDCs)	10
2.2 Related Projects	12
2.2.1 Project Hamilton	12
2.2.2 OpenCBDC	13
3 Methodology	17
3.1 Rationale	17
3.2 High Level Overview	18
3.2.1 Digital Wallets	18
3.2.2 ZyBo	21
3.2.3 DataFlow	23
4 Petalinux Software Development Kit	25
4.1 Why Petalinux?	25

4.1.1	Tailored for Xilinx Zynq SoCs	26
4.1.2	Comprehensive Development Environment	26
4.1.3	Linux-Based Environment	26
4.1.4	Balancing Control and Features	27
4.1.5	Real-Time Capabilities	27
4.1.6	Networking and Multi-Threading	28
4.2	Setting Up Petalinux	29
4.2.1	Prerequisites	29
4.2.2	Installation & Working Environment	29
4.3	Project Creation	30
4.3.1	Configuring a Hardware Platform for Linux	30
4.4	Build & Configuration	33
4.4.1	Kernel Configuration	35
4.4.2	Boot Configuration	36
4.4.3	Root Filesystem Configuration	38
4.4.4	Image Build	41
4.5	Packaging & Booting	42
4.5.1	SD Card Partitioning	43
4.5.2	Loading Files	43
5	IP Generation & Integration	45
5.1	SHA-256 IP	45
5.1.1	Creating SHA256 IP and Block Design	46
5.1.2	Writing Code in SDK and launching It	59
5.1.3	How to run the code	64
5.1.4	OUTPUT:	67
5.2	RSA IP	68
5.2.1	Introduction to AXI4-Lite (Advanced Extensible Interface)	72
5.2.2	Hardware Bottlenecks	73
5.2.3	Connections	77
5.2.4	How They Work Together	79
5.2.5	Summary	80
5.3	Simulation Results	80
5.4	Integration of IPs and Received Transaction	81
5.4.1	Steps for Integration	82
5.5	Project Presentation on Custom IP Core and Driver Abstraction	94
5.5.1	Introduction	94
5.5.2	Custom IP Core Implementation	94
5.5.3	Driver Functionality	94
5.5.4	Purpose of Abstraction	94
5.5.5	Driver Components	95
5.5.6	Driver Development Steps	95
5.5.7	Application Developer Perspective	95
5.5.8	Integration Steps	95
5.5.9	Example Workflow	95
5.5.10	Conclusion	96

6	Implementation of Hyperledger Fabric in CBDC Transaction Processing	97
6.1	Introduction	97
6.2	Setting up Hyperledger Fabric Environment	98
6.3	Installation of Hyperledger Fabric and Fabric Samples	99
6.3.1	Cloning fabric-samples Repository	99
6.3.2	Running the Installation Script	100
6.3.3	Confirming Installation	100
6.4	Explanation of Smart Contract	101
6.4.1	InitLedger	101
6.4.2	CreateAsset	101
6.4.3	ReadAsset	101
6.4.4	UpdateAsset	102
6.4.5	DeleteAsset	102
6.4.6	AssetExists	102
6.4.7	GetAllAssets	102
6.4.8	TransferAsset	102
6.5	Deploying Smart Contracts to Hyperledger Fabric Network	103
6.5.1	Chaincode Deployment Process	104
6.5.2	Endorsement Policy	105
6.6	Integration with FPGA-Based Transaction Processor	106
6.6.1	Data Transmission	107
6.6.2	Transmission Protocol	107
6.6.3	Data Processing	108
6.6.4	Transaction Execution	109
6.6.5	Performance Optimization	110
6.7	Conclusion	111
7	Conclusion and Future Work	113
7.1	Conclusion	113
7.2	Future Work	115
	Bibliography	118

List of Figures

1	Sustainable Development Goals	ii
2.1	Digital Currency Initiative	12
3.1	Digital Wallets	18
3.2	Transmitter Flowchart	20
3.3	ZyBo Development Platform	21
3.4	Zynq AP SoC Architecture	22
3.5	Transaction DataFlow	23
4.1	Sourcing Setup Script	30
4.2	Creating New Project	30
4.3	Zynq Block Automation	31
4.4	Peripherals Configuration	32
4.5	Petalinux Hardware Configuration	33
4.6	System & Image Packaging Configuration	34
4.7	Kernel Configuration	35
4.8	Linux/Arm 6.1.30 Kernel Configuration	36
4.9	U-Boot Configuration	37
4.10	U-Boot 2023.01 Configuration	37
4.11	Boot Media	38
4.12	RootFS Configuration	38
4.13	Filesystem Packages	40
4.14	Password Rules	40
4.15	Petalinux RootFS Settings	41
4.16	Petalinux Build	42
4.17	Petalinux Boot Package	43
5.1	Create and Package IP	46
5.2	Steps to create new IP	48
5.3	Adding the IPs directories to the project	50
5.4	Adding required registers	51
5.5	Replacing some <code>slv_reg</code>	52
5.6	Adding User Logic	53
5.7	Saving Changes made in IP	53
5.8	Creating Block Diagram Project	55
5.9	Adding IP repository	56

5.10 Adding IP Block	57
5.11 Adding Zynq Block	57
5.12 Final Block Diagram	58
5.13 Exporting Hardware	59
5.14 Launch SDK	60
5.15 Basic SDK Window	61
5.16 Creation of new Application	63
5.17 Code Snippet	64
5.18 Tera Term Serial Setting	65
5.19 Programming ZYBO Board	67
5.20 OUTPUT of SHA-256	67
5.21 RSA Signatures Explained	69
5.22 AXI4 Read and Write Transactions	73
5.23 Configurable Logic Block (CLB)	74
5.24 RSA Wrapper	76
5.25 RSA Semulation	80
5.26 RSA Result	81
5.27 Adding Both IPs	83
5.28 All Required Blocks Added	84
5.29 Disable Scatter Engine	84
5.30 Final Block Diagram	85
5.31 Exporting Hardware	88
5.32 Implementing Shared Memory	89
5.33 Launching SDK	90
5.34 Defining Shared Memory and DMA	91
5.35 Initializing DMA	92
5.36 Defining Transfer Function for DMA	92
5.37 Initializing Buffer Variables	92
5.38 Opening Shared Memory	93
5.39 Preparing Data for Input	93
5.40 Writing Data to Write Register	93
6.1 Various Tools and Technologies	99
6.2 Flowchart of smart Contract	103
6.3 Smart Contract Deployment	105
6.4 Transaction Endorsement Process	106
6.5 Data Transmission	107
6.6 Transmission Protocol	108
6.7 Data Processing	109
6.8 Transaction Execution	110
6.9 Performance Optimization	111

List of Tables

2.1 Comparison between Project Hamilton and OpenCBDC	15
4.1 Comparison of Bare Metal and PetaLinux	28
5.1 Comparison between RSA and AES	71

Chapter 1

Introduction

1.1 Motivation

CBDCs have become an essential topic to discuss in today's world as different states, as well as the international community as a whole, wake up to the concept of Digital Currencies and blockchain. While going for a cash less society with high online financial transactions, it is obvious that effective and secure transaction processing systems plays a crucial role.

Conventional methods of financial systems also have their several issues such as low speed, high cost and security issues of the cards. In this regard, there is an increased awareness of adopting approaches that utilise advanced systems to resolve emerging). In order to counter these problems, an increased focus has been placed on finding new ways of tackling the issues by using the new technologies.

The over-arching goal of this project is to design and implement a CBDC transaction processor developed with FPGA to increase processing velocity, and for the creation of a well-grounded private blockchain using Hyperledger Fabric. This is because FPGA provides unparalleled performance and versatility in handling a program, and is therefore desirable when processing large volumes of high-frequency transactions with minimal input lag. Moreover, Hyperledger Fabric has mechanism such as permissioned which helps

in the implementation of private blockchain that covers the issues of privacy, accuracy and scalability.

By integrating FPGA and Hyperledger Fabric, this project seeks to achieve several objectives:

1. **Enhanced Transaction Throughput:** With FPGA, parallel processing of transaction is possible; hence, transaction throughputs and time to process the transactions is much faster than the CPU-based system.
2. **Low Latency:** Thus, the real-time processing in FPGA implies very small transaction time and makes transactions fast and smooth.
3. **Improved Security:** In Hyperledger Fabric permissioned blockchain architecture allows only the verified entities to The computation power of FPGA is more hence transaction throughputs and time to process the transactions recorded in FPGA-based system are faster than that of the CPU-based system. assist in the gaining of access as well as the validation of the transactions leading to the improvement of the security of the system.
4. **Scalability:** The extensibility can also be considered a benefit due to the fact that Hyperledger Fabric is a modular solution, which facilitates the extension of the system that is able to handle a growing number of transactions as well as increasing users' count without experiencing a decline in performance rates.
5. **Privacy Preservation:** Hyperledger provides various capabilities or solutions on private channels and confidential transactions to achieve the private nature of the smart contracts and the participants' transactional informations.

This project focuses on designing a CBDC transaction processor utilizing FPGA and the Hyperledger Fabric architecture to enhance the knowledge regarding digital currency and blockchain applications for practical use, thereby enhancing the current operation of the financial system.

1.2 Problem Statement

Although CBDCs have a great prospect, there are some obstacles that should be solved relating to CBDC. The current TP systems are characterized by some inefficiencies and constraints that negatively impact CBDC's deployment and effectiveness. Some of the key challenges include: Some of the key challenges include:

1. **Slow Transaction Speeds:** In the context of CBDCs, CTSs are unable to support the expected, high-frequency transactions hence causing delays and congestion in the execution of payment settlements.
2. **High Transaction Costs:** Burdens which accrue from fulfilment of transactions such as fees and infrastructure costs may be exorbitant especially for small value transactions hence the challenge to financial inclusive and accessibility.
3. **Privacy and Security Concerns:** Typically, there is no issue of privacy when it comes to public blockchains since all the transaction information is recorded in the ledger that is publicly accessible by anybody with an internet connection, and central bank transactions require the highest possible levels of anonymity.
4. **Scalability Issues:** For instance, as the number of transactions and the number of users increases in the CBDC ecosystem, this may lead to a negative impact towards network traffic and hence, the efficiency of the CBDC will start to deteriorate.

Meeting these needs involves designing a strong transaction processing system that enables fast, cheap, reliable and elastic transaction processing for CBDCs. These challenges are going to be addressed with this project utilizing the implementation of the FPGA technology and Hyperledger Fabric to create an optimized CBDC transaction processor.

Thus, by focusing on the specified difficulties and offering unique resolutions for every issue, this project should offer valuable contributions to the field of digital currency and blockchain technology that will enable the global establishment of CBDCs.

1.3 Scope

The ambit of this project is to design and build efficient CBDC transaction processing system using FPGA and Hyperledger Fabric architecture. The main goal is to solve the main problems of CBDCs such as low speed of the transaction, high price of the transactions, the question of privacy and security, as well as the problems of the scalability.

The project will focus on the following key areas:

1. **Transaction Processing System Design:** CBDCs need to be supported by a strong transaction processing system that should be constructed for this purpose exclusively. Thus, this system should focus on high speed, low cost, reliability, and scalability of transaction processing.
2. **FPGA Integration:** Integrate FPGA technology into the transaction processing system to enhance its performance and efficiency. FPGA's inherent parallel processing capabilities will be leveraged to optimize transaction throughput and reduce latency.
3. **Hyperledger Fabric Implementation:** Adopt Hyperledger Fabric for the creation of CBDC transactional network since it is a permissioned blockchain. Hyperledger Fabric is a suitable blockchain platform to support UL Cyclone because of its modular design and endorsement of confidential transactions.
4. **Addressing Challenges:** Develop tailored solutions to address the specific challenges faced by CBDCs, including:
 - Overcoming slow transaction through better algorithms in transaction processing and incorporating FPGAs.
 - Cutting the costs of coordination by optimizing the process of fulfilling the transactions and decreasing infrastructure expenses.

- The obligations include the aspects of the privacy and security based on the means of cryptographic techniques and the means of confidential transactions.
 - Enabling the management of scale with a refined transaction-processing architecture to support the future growth of the CBDC ecosystem.
5. **Evaluation and Validation:** Assess the efficiency, stability, and security of the developed CBDC transaction processing system to a great extent. Carry out ample pilot testing to confirm whether the implemented system of solution solves the main difficulties of the undertaking.
 6. **Documentation and Dissemination:** This paper should outline the conceptualization, realization, and assessment of the CBDC transaction processing system. Knowledge outputs have to be shared through scientific journals, technical and research papers, as well as conferences as a means of providing advancements to the field of the digital currency and the blockchain technology.

In pursuing these aspects, the project has the objective of coming up with feasible solutions to the issues that have an impact on the implementation and use of CBDCs. Thus, the optimized CBDC transaction processing system evolving from this project should be quite beneficial in the development and uptake of CBDCs and blockchain.

1.4 Aims and Objectives

1.4.1 Aims

The goal of this project will be to develop an efficient system for processing Central Bank Digital Currency (CBDC) transactions that will proactively overcome some of the issues that are associated with CBDC that include slow transaction rates, high costs of the transactions, and privacy, as well as, security issues, and scalability. Therefore, with the use of FPGA and with the help of hyper ledger fabric, the project aims to establish a

strong CBDC transaction processing framework that can enable cheap, fast, reliable, and scalable CBDC transactions.

1.4.2 Objectives

To achieve the aim outlined above, the project will pursue the following objectives:

1. **Designing a Transaction Processing System:** Design a TP system that is specifically built for CBDCs; one that is fast, cheap, dependable, and can process a large number of transactions.
2. **Integrating FPGA Technology:** Introduce FPGA into the TPS as a solution to its performance that could particularly benefit from parallel processing.
3. **Implementing Hyperledger Fabric:** Hyperledger Fabric should be applied to create a privacy and security enhanced CBDC transaction network to mitigate privacy and security concerns.
4. **Addressing Key Challenges:** Find strategies for possible challenges on the low transaction speeds, high transaction costs, need for privacy and security and issues of scalability that are unique in CBDCs.
5. **Evaluating Performance:** Performance evaluation and validation of the developed CBDC transaction processing system must be carried out to check the level of compliance to best practices and to check its proficiency to deal with the outlined challenges.
6. **Documenting and Disseminating Findings:** Detail the design, implementation, evaluation and the findings of the project. Share information by producing articles, research papers, and papers on the results obtained as well as giving presentations to assist in the enhancement of digital currency and blockchain.

1.5 Outcomes

1.5.1 Expected Outcomes

The project aims to achieve the following outcomes:

1. **Optimized CBDC Transaction Processing System:** Develop an optimized transaction processing system for CBDCs that demonstrates improved transaction speeds, reduced transaction costs, enhanced privacy and security, and scalability to meet the demands of a growing CBDC ecosystem.
2. **Integration of FPGA Technology:** Successfully integrate FPGA technology into the transaction processing system, leveraging parallel processing capabilities to enhance performance and efficiency.
3. **Implementation of Hyperledger Fabric:** Implement Hyperledger Fabric to build a secure and privacy-enhanced CBDC transaction network, ensuring compliance with regulatory requirements and addressing privacy concerns.
4. **Addressing Key Challenges:** Develop tailored solutions to address key challenges faced by CBDCs, including slow transaction speeds, high transaction costs, privacy and security concerns, and scalability issues.
5. **Thorough Evaluation and Validation:** Conduct comprehensive performance evaluation and validation of the developed CBDC transaction processing system to ensure its effectiveness, reliability, and compliance with standards.
6. **Documentation and Dissemination of Findings:** Document the design, implementation, evaluation process, and findings of the project. Disseminate insights through academic publications, technical reports, and presentations to contribute to the advancement of digital currency and blockchain technology.

1.5.2 Potential Impact

Hypothesis testing research outcomes of this project are expected to bring influential change to the field of digital currency and blockchain technology. The optimized CBDC transaction processing system developed through this project has the potential to: The optimized CBDC transaction processing system developed through this project has the potential to:

- Enhance the ability of central banks to promote CBDCs by responding to major obstacles that limit their implementation and success.
- Improve financial inclusion and accessibility by reducing transaction costs and enhancing transaction speeds, particularly for small value transactions.
- Enhance efficiency in use of financial services by lowering the cost per transaction especially in instances where the value of the transaction is low, so as to promote financial access.
- Create the basis for further research and development of the topic of digital currency and the application of blockchain technology that will result in technological progress.

1.6 Report Organization

The organization of the thesis is as follows:

- **Introduction:** This chapter provides background information about the research project that this thesis belongs to, including its objectives, aims, rationale, and the identified problem. To achieve this, it defines the project's purpose, aims and objectives for readers to grasp quickly and easily. Moreover, it defines the expected advantages and possible consequences, and gives a short description of the organization of the report.

- **Literature Review:** This chapter for that reason gives a proper background of the subject under discussion, namely Central Banking Digital Currencies. To situate the current research, it identifies related works like Project Hamilton & OpenCBDC.
- **Methodology:** This chapter justifies the methods and approaches to be used in the study. They provide the general picture of the project, reveal what exactly is going to be developed and how all components named as Digital Wallets, ZyBo, DataFlow are interconnected and making a significant impact on the whole project.
- **Petalinux Software Development Kit:** This chapter focuses on the rationale of choosing Petalinux, and it considers Petalinux as a great development scaffold that supports Xilinx Zynq SoCs. It includes the processes of setup, project creation, and configuration that leads to build & packaging that are important aspects for system deployment.
- **IP Generation & Integration:** This chapter deals with the processes of generating and incorporating two Intellectual Property components; SHA-256 and RSA IPs. It defines the design module, coding, and testing, comparing simulation with actual results, as well as the procedure to incorporate the IPs into the system.
- **Implementation of Hyperledger Fabric in CBDC Transaction Processing:** This chapter provides the procedure of applying Hyperledger Fabric in the project. It describes how to deploy Hyperledger Fabric, outlines how smart contracts work and focuses on the deployment of such contracts. The chapter is also devoted to the introduction of the FPGA-based transaction processors and the possibilities of their performance enhancement.
- **Conclusion and Future Work:** The last chapter gives the conclusions of this research project and briefly describes the impact and significance of the project. It also describes probable avenues for future research and development to indicate the possible paths for future improvement and advancement of the area.

Chapter 2

Literature Review

2.1 Background

2.1.1 Central Banking Digital Currencies (CBDCs)

CBDCs can be defined as sovereign currency in a digital form and with central bank's backing. CBDCs are quite different from decentralized coins like Bitcoin and Ethereum since they are issued under central authority and present a new model of digital currency. Major key factors that have led to the development of CBDC's include, advanced under the headline financial inclusion, boosted through smaller scale payments, a substitute for the declining cash based fundamental, and as a reaction to private cryptocurrencies like stable coins [1].

CBDCs are generally categorized into two types:

- **Retail CBDCs:** The retail forms of CBDCs are intended for the population and act as digital cash equivalents. They can be stored in a digital wallet implemented by the central bank or other accredited agents. Retail CBDCs' goal is to increase the availability of financial services to the general public since a large number of people do not have bank accounts or limited access to them, offering a safe and effective

payment method.

- **Wholesale CBDCs:** Wholesale CBDCs are meant to be central bank money for payment settlement among institutions and are aimed at the wholesale sector. These seek to enhance the speed and security together with speed of transactions in the financial system. Thus, the use of wholesale CBDCs can contribute to faster and more secure cross-border payments than those that can be achieved with classical payment systems [2].

The implementation of CBDCs involves several critical considerations:

- **Privacy:** Balancing the need for privacy in individual transactions with regulatory requirements for transparency and anti-money laundering (AML) measures.
- **Security:** Ensuring the robustness of the digital currency system against cyber threats and fraud.
- **Scalability:** Developing a system capable of handling a high volume of transactions without compromising performance.
- **Interoperability:** Ensuring compatibility with existing financial systems and other CBDCs to facilitate seamless transactions.
- **Legal Framework:** Establishing a regulatory and legal framework to govern the issuance, distribution, and use of CBDCs [1].

2.2 Related Projects

2.2.1 Project Hamilton

Project Hamilton is a joint research and development project between the System's Boston Fed office and MIT's Digital Currency Initiative, which was started in the year 2020. The main goal of the project is to discuss and define the specifics of launching, designing, and implementing a CBDC that will be effective, reliable, and efficient.



Figure 2.1: Digital Currency Initiative

Key Contributions of Project Hamilton

- **Design Exploration:** Research published under Project Hamilton has looked into a number of architectures for a CBDC ranging from ledger-based to non-ledger-based. The emphasis is made on two-level structure of CBDC itself: the central bank directly distributes CBDCs to the intermediaries, including the commercial banks, for further distribution to the final users. This design is proposed to allow the existing banking facilities to still hold while the currency will be under central control.

- **Transaction Speed and Scalability:** Project Hamilton’s major accomplishment is the proof of concepts that show a capability of handling more than 1. A capability statement of approximately 7 million transactions per second coupled with a settlement finality that is less than a sub-second [3]. Such a high performance is provided by effective data structures and consensus which are free from the limitations of classical blockchain-related systems.
- **Privacy and Security:** Project Hamilton places a strong emphasis on transaction privacy while ensuring regulatory oversight. The project explores advanced cryptographic techniques such as zero-knowledge proofs and secure multi-party computation to achieve confidentiality and compliance [3].
- **Interoperability:** Project Hamilton defines standards and protocols that facilitate the integration of the CBDC system to other infrastructures in the payment system to complement other CBDCs across the world. Indeed, this interoperability is vital for the realization of international payments to facilitate a global digital currency environment [4].
- **Open Source Development:** Project Hamilton adopts an open-source approach to promote transparency, collaboration, and innovation. The OpenCBDC software, released under Project Hamilton, provides a foundational platform for further research and development in the field of digital currencies [3].

2.2.2 OpenCBDC

OpenCBDC is an open-source initiative derived from Project Hamilton’s research, aimed at providing a comprehensive framework for the development and deployment of CBDCs. OpenCBDC offers tools, libraries, and documentation to assist central banks and developers in creating secure and efficient digital currency systems.

Key Features of OpenCBDC

- **Modular Architecture:** OpenCBDC's modular design allows central banks to customize the system according to their specific needs and requirements. This flexibility supports the integration of various technologies and components, such as different consensus mechanisms, privacy-preserving techniques, and user interfaces.
- **High Performance:** Building on Project Hamilton's findings, OpenCBDC is designed to handle high transaction volumes with low latency. This ensures that the system can support national-scale payment infrastructures without compromising performance [5].
- **Security and Privacy:** OpenCBDC incorporates advanced cryptographic methods to ensure the security and privacy of transactions. Techniques such as zero-knowledge proofs, secure multiparty computation, and other privacy-enhancing technologies are employed to protect user data while enabling regulatory oversight.
- **Interoperability and Standards:** The platform supports interoperability with other payment systems and digital currencies. Adherence to international standards and best practices facilitates seamless integration and cross-border functionality, promoting a cohesive global financial ecosystem [5].
- **Developer and User Engagement:** OpenCBDC provides extensive documentation, tutorials, and support to engage both developers and end users. This community-driven approach encourages innovation and collaboration, fostering a vibrant ecosystem around CBDC development [5].
- **Regulatory Compliance:** The framework includes features to support regulatory compliance, such as transaction monitoring, reporting tools, and mechanisms for implementing monetary policy. These features ensure that the CBDC system adheres to legal and regulatory requirements while providing central banks with the necessary tools for oversight and control [5].

Feature	Project Hamilton	OpenCBDC
Initiative Origin	Federal Reserve Bank of Boston and MIT	MIT Digital Currency Initiative
Objective	Research and develop a hypothetical U.S. central bank digital currency (CBDC)	Provide an open-source platform for CBDC development
Primary Focus	Exploring technical and design aspects of a CBDC	Offering tools and frameworks for CBDC implementation
Core Components	High-throughput, resilience, and secure transaction processing	Modular architecture supporting various CBDC use cases
Scalability	Designed to handle high transaction volumes	Scalable design to adapt to different needs
Security	Emphasis on robust security mechanisms	Implements advanced security protocols
Transparency	Research results and code are made publicly available	Open-source platform ensuring transparency
Interoperability	Aims for compatibility with existing financial systems	Supports integration with various financial infrastructures
Privacy	Investigates privacy-preserving techniques	Includes options for configurable privacy features
Performance	High performance with minimal latency	Optimized for performance and efficiency
Collaborations	Collaboration between the Federal Reserve and MIT	Open for contributions from global developers and institutions
Development Stage	Ongoing research with published findings and prototypes	Continuous development with updates and community input
Use Case Flexibility	Focuses on specific U.S. CBDC requirements	Adaptable to multiple central bank digital currency needs globally
Documentation	Detailed research papers and technical reports	Comprehensive documentation and developer resources
Regulatory Considerations	Alignment with U.S. regulatory standards	Designed to accommodate various regulatory frameworks

Table 2.1: Comparison between Project Hamilton and OpenCBDC

Impact and Future Directions

OpenCBDC and Project Hamilton can be referred to as major milestones on the way toward the implementation of CBDCs. These actions reveal that despite central bank digital currencies being in their experimentation phase, these ICs help global central banks examine, discuss, and apply technical underpinnings by employing open-source procedures. The future work will be around the development of privacy-preserving protocols, the CBDC systems scalability, legal compliances and how multiple CBDC systems can coexists.

Chapter 3

Methodology

3.1 Rationale

This project involves a sophisticated integration of hardware and software components to create a Digital Currency Transaction Processor. The effectiveness of the system's performance is of absolute value since an implementation like this expects heavy network traffic as transaction requests are occurring in real-time and the system needs to handle them in parallel.

For this purpose, the hardware customization and flexibility of FPGAs come in handy, as we can optimize the hardware, potentially achieving very high efficiency. Because FPGAs can be configured to implement operations directly in hardware, they often exhibit lower latency compared to GPUs. This is crucial for real-time applications, such as this Transaction Processor, where immediate processing is required. FPGAs are also significantly more power efficient than GPUs, so they are a better option, specially because the system requires continuous operation.

3.2 High Level Overview

3.2.1 Digital Wallets

Digital wallets are one of the modern innovations in the sphere of financial applications and services that enable the users to store substantive funds, make transactions with those funds and track payment information on the commonly used today devices, such as, for instance, phones and tablets.

Clients do not interact directly with the hardware, but rather with the Digital Wallet platforms provided by the state bank or an accredited party. Users make their transaction requests in a similar manner as any other online banking applications, but the wallet transmits the transaction metadata to the FPGA board for further processing.



Digital Wallets share the transaction data with Zybo using JSON Remote Procedure Calls (RPC) which is a protocol encoded in JSON. It is commonly used in blockchain-related applications for communication between a wallet and a system. In a JSON-RPC request

or response, transaction information is usually structured in a JSON format and the data transfer can be validated with the JSON Web Tokens (JWT), an IETF standard (RFC 7519) which describes a format of transmitting data as a JSON object with protection against tampering. This information can be considered reliable because it is digital signed.

A normal JSON transmission is as follows:

```
{  
  "jsonrpc": "2.0",  
  "method": "sendTransaction",  
  "params": {  
    "from": "sender_address",  
    "to": "recipient_address",  
    "amount": 10.0,  
    "gas": 21000,  
    "nonce": 1  
  },  
  "id": 1  
}
```

In the context of a "sendTransaction" method, the parameters typically include:

- **from:** The address from which the funds are being sent (sender's address).
- **to:** The recipient's address to which the funds are being sent.
- **amount:** The amount of cryptocurrency or tokens to be sent in the transaction (in this case, 10.0 units).
- **gas:** The amount of gas units to be used for the transaction. Gas is a measure of computational effort required to execute operations on the blockchain.
- **nonce:** A unique number assigned to the transaction, preventing replay attacks.

JSON-RPC can be used with TCP as the transport protocol. JSON-RPC itself is a protocol-independent specification for remote procedure calls (RPC), and it can be used over various transport layers, including HTTP, WebSocket, and TCP.

The implementation of Digital Wallets is not within the scope of this project but for system testing, a c++ script will be responsible for emulating the behavior of a wallet that generates dummy transactions by randomizing the details between transactions. The transmitter establishes communication with the system using typical properly handled TCP socket connections, ensuring proper data transmission and reception.

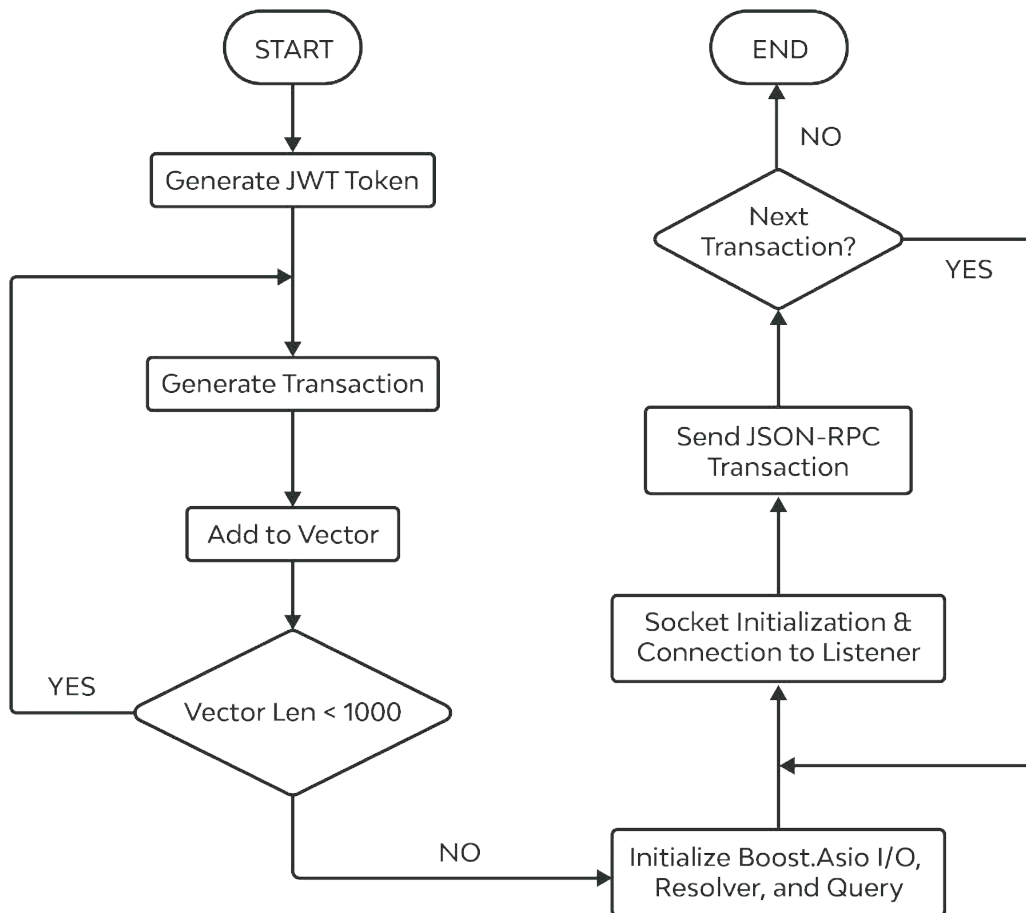


Figure 3.2: Transmitter Flowchart

3.2.2 ZyBo

This project's design and development is according to Digilent's Zybo Board. The ZYBO (ZYNq BOard) is an easy-to-use, feature-rich platform for digital circuit creation and embedded software entry-level that is based on the Z-7010, the smallest Zynq-7000 family member from Xilinx. The Xilinx All Programmable System-on-Chip (AP SoC) architecture, upon which the Z-7010 is built, tightly integrates Xilinx 7-series Field Programmable Gate Array (FPGA) logic with a dual-core ARM Cortex-A9 processor [6].

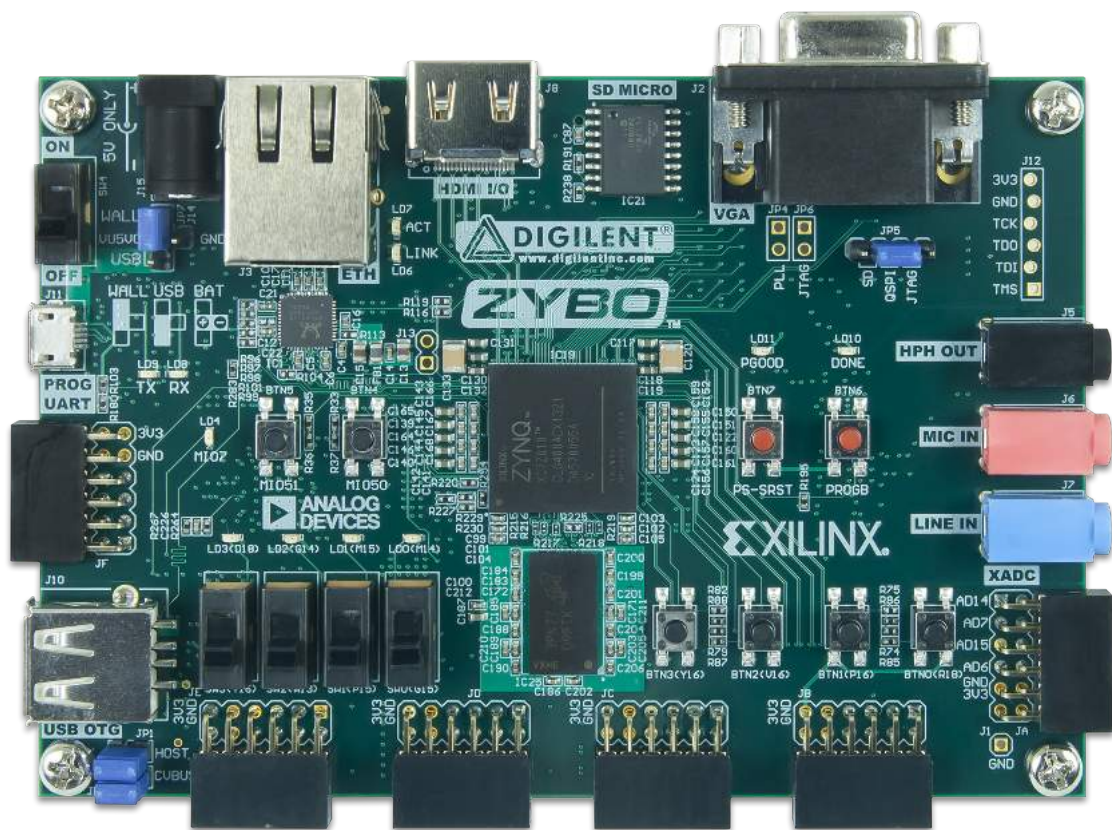


Figure 3.3: ZyBo Development Platform

In order to make the system able to communicate with external sources, Zybo offers High-bandwidth peripheral controllers, specifically a Gigabit Ethernet for network connection. For all intents and purposes, a Cat-6 Ethernet cable is sufficient and is supported by the RJ45 Connector on Zybo.

Incoming transactions will be handled at the ARM Processor, which will administer multi-threading to process the transactions in parallel. Every received transaction is validated and acknowledgment is sent back to the sender before the JSON is parsed and data is mapped onto the DDR3 shared memory. The Application Processing Unit (APU) is directly connected to the Multiport DRAM Controller while the the FPGA can access the memory via the Advanced Microcontroller Bus Architecture (AMBA) Interconnects [7].

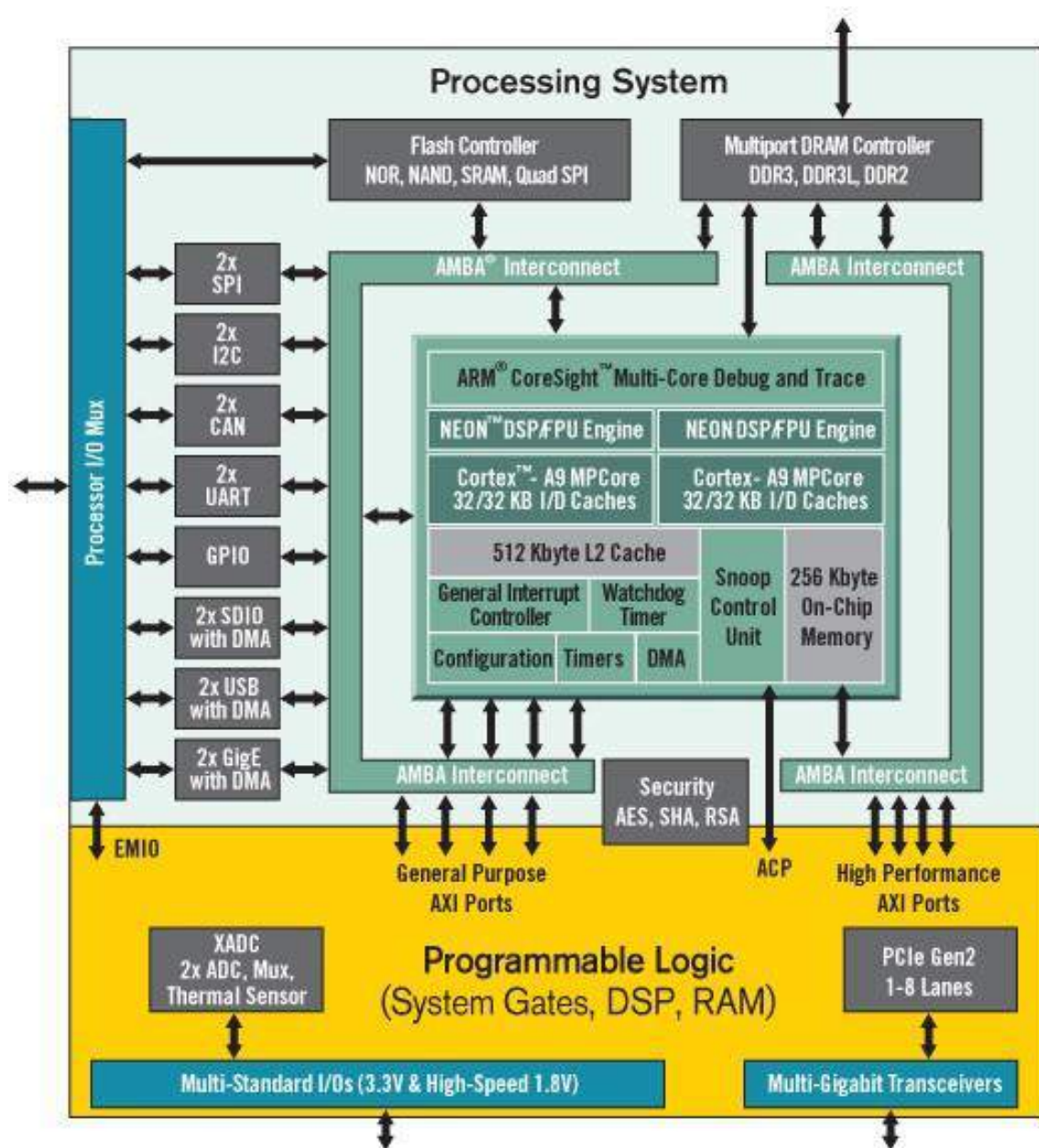


Figure 3.4: Zynq AP SoC Architecture

The High Performance Advanced eXtensible Interface (AXI) Ports are used by the IP Cores at FPGA to interact with ARM i.e. AXI Interfacing.

3.2.3 DataFlow

The system pre-allocates memory blocks for each transaction which are known to both the processor and the FPGA. Each thread stores the transaction data to their respective memory blocks. The key is to use a specific memory location as a flag or status register. The processor sets this flag after writing data, and the FPGA polls this flag to detect when data is ready. The processor sets this flag after writing data, and the FPGA polls this flag to detect when data is ready.

Each transaction is used to generate a 256-bit hash using SHA-256 and the transaction data itself is encrypted using RSA. The Hashing and Encryption procedures are carried out at FPGA by designated IP Peripherals. The processed data is written back at the transaction's corresponding memory range and the flag is set such that the thread can retrieve the hash & encrypted data.

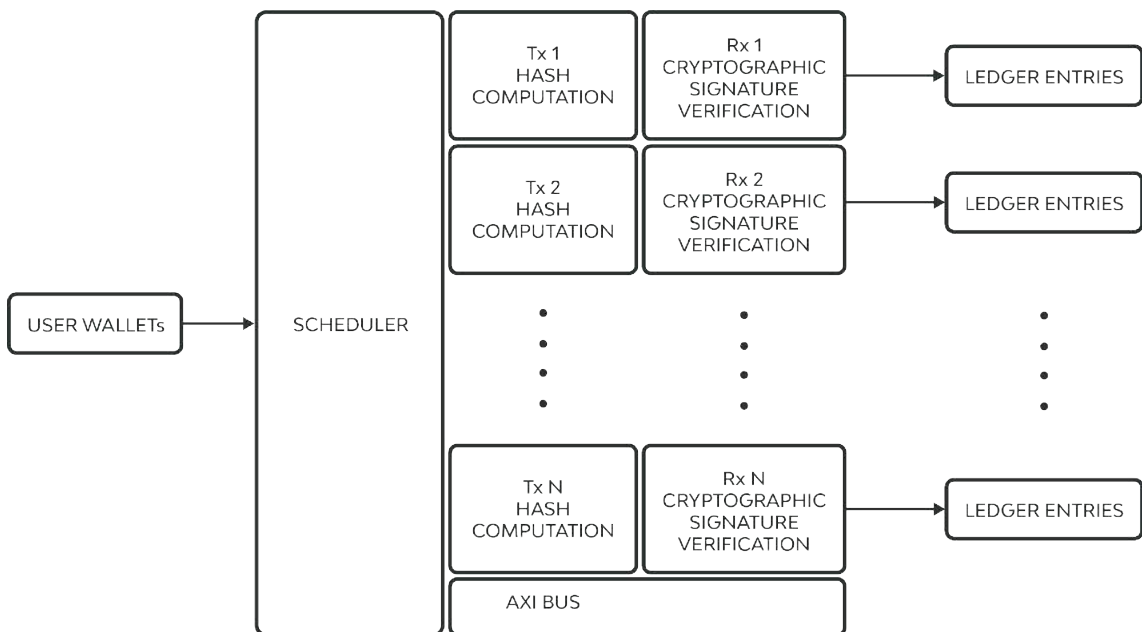


Figure 3.5: Transaction DataFlow

The motive behind the hash computation is for it to act as an identifier to its respective transaction once it is added to blockchain. Every transaction being encrypted adds an extra layer of security that makes the transaction record protected. The processor updates the local copy of the ledger and broadcasts it to the network for verification by the peers. Once verified, the HyperLedger Fabric ensures sync across the entire network and the wallet's database is updated.

Chapter 4

Petalinux Software Development Kit

4.1 Why Petalinux?

Using Linux on ARM processors combines the strengths of both worlds: the efficiency of ARM architecture together with power-saving characteristics and the stability, openness, and versatility of the Linux OS. Combined, they are very advantageous in numerous applications including embedded systems, Internet of Things, and servers. Therefore, making it perfect for this project as using ARM and FPGA in conjunction directs maximum efficiency and flexibility.

Choosing the right operating system for an embedded system is crucial to ensure optimal performance, efficiency, and functionality. Among various options, PetaLinux stands out as an excellent choice for embedded systems, particularly those based on Xilinx Zynq SoCs. PetaLinux is an embedded Linux Software Development Kit (SDK) targeting FPGA-based system-on-a-chip (SoC) designs or FPGA designs [8].

4.1.1 Tailored for Xilinx Zynq SoCs

Designed for Xilinx Hardware

- **Optimized Integration:** PetaLinux is particularly created for effortless usability with the Xilinx Zynq SoCs enjoying the best drivers, hardware assistance, and compatibility.
- **Hardware Acceleration:** Takes advantages of both the ARM processor and the FPGA fabric to support hardware accelerated computation for those high computational regions.

4.1.2 Comprehensive Development Environment

Integrated Development Tools

- **PetaLinux Tools:** Is a single-stage tool that contains all the necessary components for Linux Systems configuration, development, and integration onto Xilinx equipment.
- **Ease of Use:** Enables solving many tasks in the process of development with the help of special tools that do not require much time for learning and development.

4.1.3 Linux-Based Environment

Rich Feature Set

- **Networking Support:** Comes with strong networking feature that is vital in networking the embedded systems being developed.
- **Multi-threading Support:** Supports POSIX threads (pthread) and other threading libraries which are very much important for managing the transactions that involve multi-threading.

- **Extensive Libraries:** Availability of a large population of libraries and easily interface-able software packages.

Community and Support

- **Extensive Community Support:** The large number of developers and extensive documentation will make it easier in finding solutions and getting help.
- **Wide Range of Applications:** This tool of course works fine with a great number of the applications and tools which are available in the Linux environment.

4.1.4 Balancing Control and Features

Bare-Metal vs. Full OS - [4.1](#)

- **Balance Between Control and Features:** PetaLinux strikes a balance between the control offered by bare-metal environments and the rich features provided by full-fledged operating systems like standard Linux distributions.
- **Low-Level Hardware Interaction:** As an Operating System it also considers high-level design abstractions but it also offers direct hardware access when required.

4.1.5 Real-Time Capabilities

Real-Time Performance

- **PREEMPT-RT Patch:** Although not first-class supported by PetaLinux, it is possible to configure it to include the PREEMPT-RT patch which gives it real-time capabilities that are useful in applications that demand timely responses.
- **Xilinx Support for Real-Time:** There is very specific support from Xilinx as well as configuration in real time for the PetaLinux applications that improve the aspect of real time.

Aspect	Bare Metal	PetaLinux
Operating System	None	Linux-based OS
Performance	High, minimal overhead	Moderate, some overhead due to OS
Latency	Low, direct hardware access	Higher, due to OS layers
Complexity	High, requires managing all hardware aspects	Lower, OS abstracts many hardware details
Development Tools	Basic, low-level tools	Extensive, Linux toolchain and ecosystem
Scalability	Limited	High, supports multitasking and networking
Real-Time Capabilities	Excellent, direct control	Moderate, depends on real-time Linux patches
Memory Footprint	Small	Larger due to OS presence
Flexibility	Limited, fixed functionalities	High, easily add new features and services
Networking	Custom implementation required	Built-in support with Linux networking stack
Security	Custom implementation required	Built-in support with Linux security features
Use Cases	Real-time systems, low-level control	Complex systems, multitasking, networked applications

Table 4.1: Comparison of Bare Metal and PetaLinux

4.1.6 Networking and Multi-Threading

Handling Transactions

- **Efficient Transaction Handling:** Because of the multi-thread and networking capability the PetaLinux is ideal for the applications where number of transactions need to be handled in a most efficient way possible.
- **Minimized Overhead:** Through the restricted installation with only the necessary packages, PetaLinux only includes a light load, thus ensuring high performance.

4.2 Setting Up Petalinux

4.2.1 Prerequisites

It is advisable to install the PetaLinux tools in a manner that does not require root access. The standard development tools and libraries should be installed on your host Linux workstation for PetaLinux development.

PetaLinux tools require your host system `/bin/sh` to be 'bash'. If you use Ubuntu distribution and your `/bin/sh` is 'dash', then your system administrator should help you change your default system shell `/bin/sh` to 'bash' by using the command of `sudo dpkg-reconfigure dash`.

4.2.2 Installation & Working Environment

Once Petalinux Installer is downloaded from the Xilinx Downloads [\[9\]](#) section, run the installer and it should be installed into the current working directory by default. You can install by specifying installation path and while installing the tool, you can specify your preferred eSDK, since we are working on Zynq platform, we will opt to install the Arm eSDK into the PetaLinux tool.

```
./petalinux-v2023.2-10121855-installer.run --dir <INSTALL_-  
-DIR> --platform "arm"
```

The rest of the setup is done by sourcing the provided settings scripts. For Bash user shell:

```
source <PetaLinux-Path>/settings.sh
```

```
~$ source petalinux/2023.2/settings.sh
Petalinux environment set to '/home/hamza/petalinux/2023.2'
WARNING: This is not a supported OS
INFO: Checking free disk space
INFO: Checking installed tools
INFO: Checking installed development libraries
INFO: Checking network and other services
~$
```

Figure 4.1: Sourcing Setup Script

4.3 Project Creation

Create a new PetaLinux project named 'FYP' using the Zynq template and set up the project's environment. The temporary directory specified will be used during the creation and build processes to store temporary files. This can be useful for managing disk space and organizing project files.

```
petalinux-create -t project -n FYP --template zynq --tmpdir
/home/hamza/petalinux/temp
```

```
~$ petalinux-create -t project -n FYP1 --template zynq --tmpdir /home/hamza/petalinux/temp1
INFO: Create project: FYP1
INFO: Project TMPDIR is redirecting to /home/hamza/petalinux/temp1
INFO: New project successfully created in /home/hamza/FYP1
~$
```

Figure 4.2: Creating New Project

4.3.1 Configuring a Hardware Platform for Linux

To design your hardware platform, you can use the AMD Vivado tools. Whether the hardware platform has been designed and built from scratch or has been obtained as an off-the-shelf solution, there are only a few hardware IP and software platform configuration parameters that are required in order to make the hardware platform Linux capable.

- First of all, launch the Vivado and create a new RTL project for the ZYNQ-7 ZC702 Evaluation Board.

- In the 'Project Manager' window, click the 'Create Block Design' button.
- From the Add IP window add 'ZYNQ7 Processing System' as the IP.
- The following is the connection of FCLK_CLK0 output pin to M_AXI_GP0_ACLK input pin of 'ZYNQ7 Processing System' IP after the connection.
- To access this, go to the designer assistance and click on 'Run Block Automation' and it will appear as follows [10].

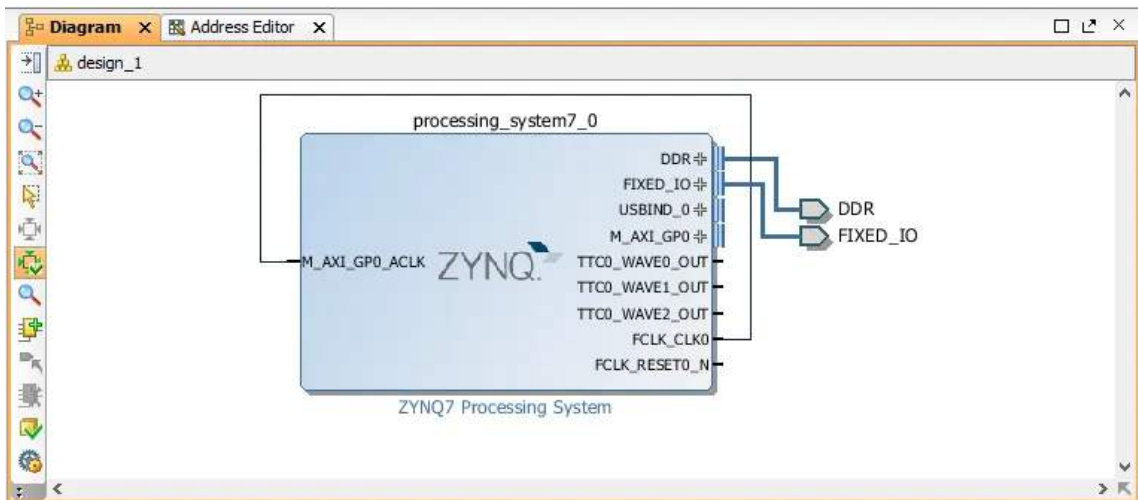


Figure 4.3: Zynq Block Automation

- Right click on ZYNQ PS IP and select the option Re-customize IP; then on the left side of the newly opened window selecting Peripheral I/O Pins tab.
- After confirming the configuration, we close the window by clicking on the OK button.
- On the block design file, right click the file as depicted in the figure below and choose 'Create a Wrapper'. Select "Let Vivado manage wrapper and auto-update"

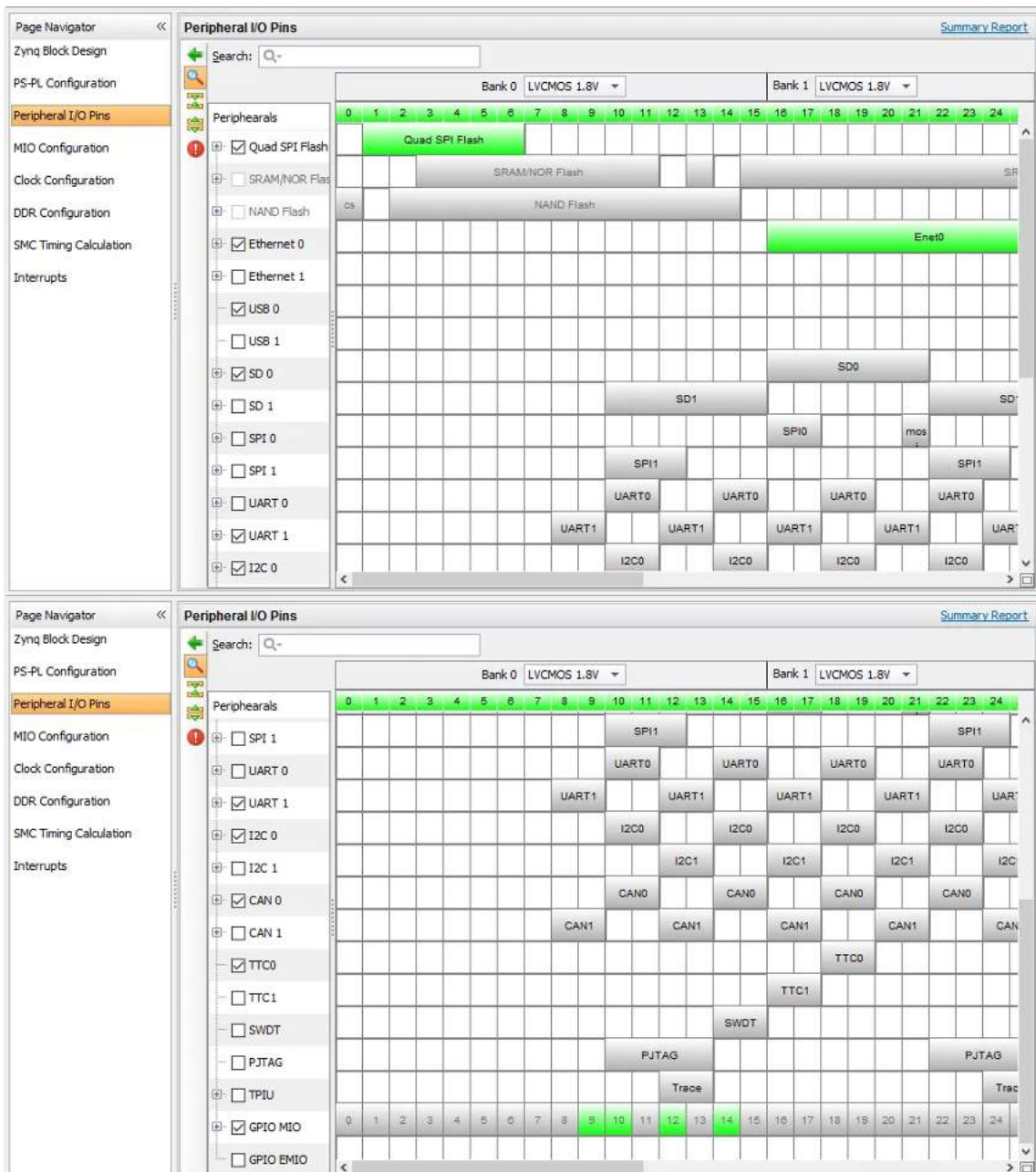


Figure 4.4: Peripherals Configuration

and click on “Ok”.

- Create a version of the design and produce the bit stream.
- After bit stream is generated, go to file and select export, then export Hardware, check the option of include bit stream then click on the ok button.

From the above step, hardware description file(.xsa) of the created design, will be saved

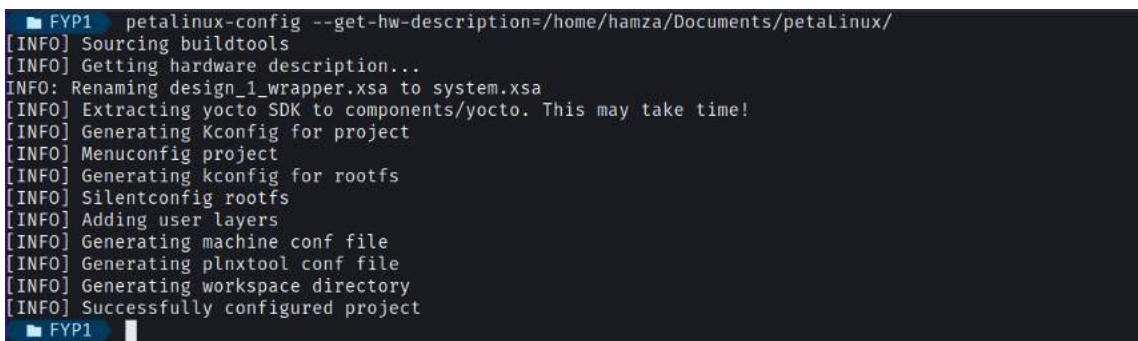
in the <project_name>.sdk folder in the project directory. Save this to the development linux distribution where petalinux is set up.

4.4 Build & Configuration

Import the hardware description (exported from Vivado in the previous section) into the PetaLinux project and use it to configure the project settings.

```
petalinux-config --get-hw description=/home/hamza/Documents  
/petalinux/
```

It pulls information on the hardware design elements like the processing system, peripherals, and interfaces and configures the PetaLinux project as needed.



```
FYP1 petalinux-config --get-hw-description=/home/hamza/Documents/petalinux/  
[INFO] Sourcing buildtools  
[INFO] Getting hardware description...  
INFO: Renaming design_1_wrapper.xsa to system.xsa  
[INFO] Extracting yocto SDK to components/yocto. This may take time!  
[INFO] Generating Kconfig for project  
[INFO] Menuconfig project  
[INFO] Generating kconfig for rootfs  
[INFO] Silentconfig rootfs  
[INFO] Adding user layers  
[INFO] Generating machine conf file  
[INFO] Generating plnxtool conf file  
[INFO] Generating workspace directory  
[INFO] Successfully configured project  
FYP1
```

Figure 4.5: Petalinux Hardware Configuration

The PetaLinux configuration menu provides a way to set various parameters for your project including kernel settings, root file system settings, and U-Boot settings. In the Image Packaging Configuration Section, uncheck the 'Copy final images to tftpboot' option.

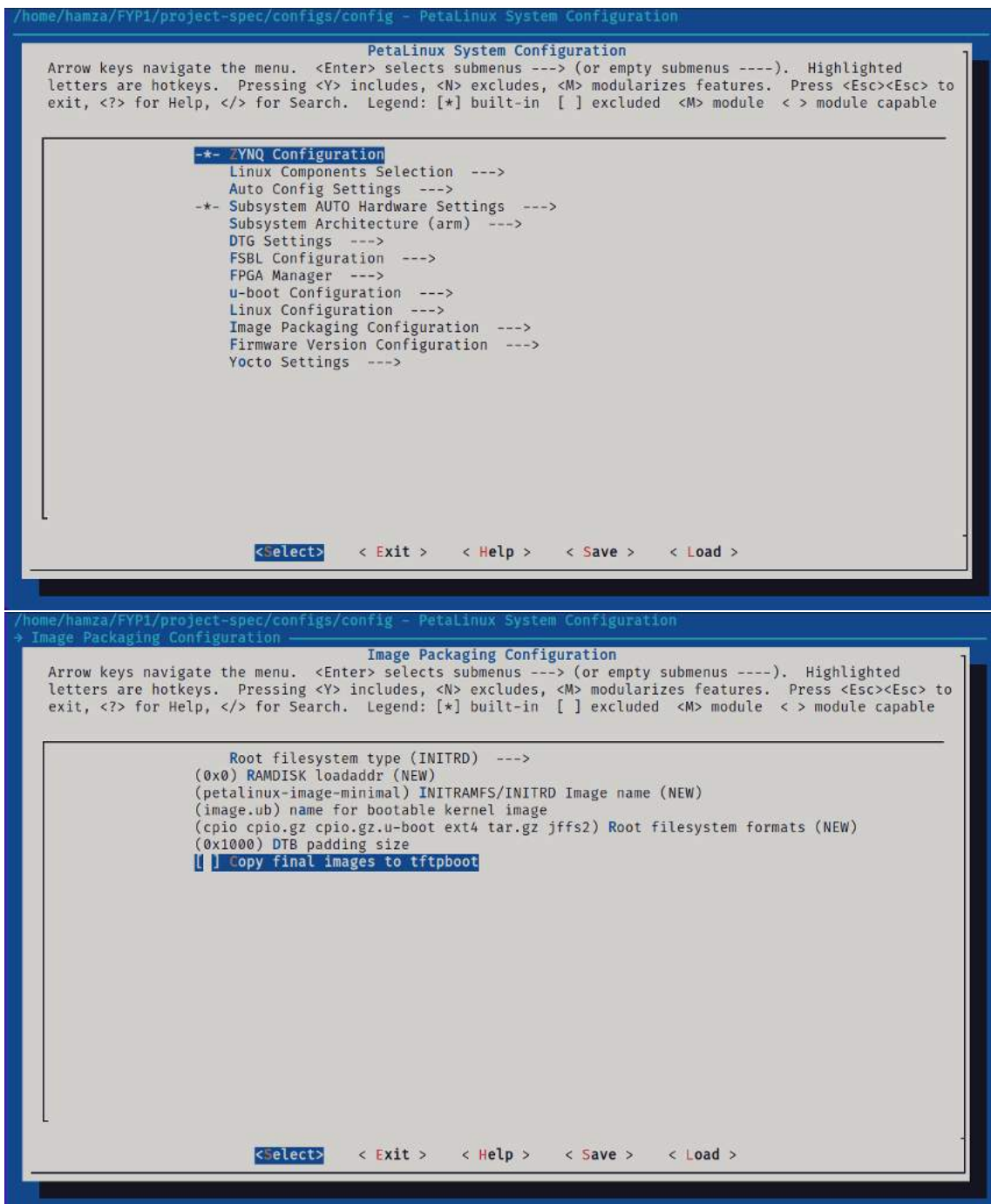


Figure 4.6: System & Image Packaging Configuration

4.4.1 Kernel Configuration

Running `petalinux-config -c kernel` presents the kernel configuration menu whereby you can adjust kernel parameter related to your PetaLinux project. This has possibilities connected with different kernel characteristic, motorists, file systems, and other kernel accommodations.

```
TransactionProcessor petalinux-config -c kernel
[INFO] Sourcing buildtools
[INFO] Silentconfig project
[INFO] Silentconfig rootfs
[INFO] Generating workspace directory
[INFO] Configuring: kernel
[INFO] bitbake virtual/kernel -c cleansstate
NOTE: Started PRServer with DBfile: /home/hamza/petalinux/2023.2/TransactionProcessor/build/cache/prserv.sqlite3, Address: 127.0.0.1:36937, PID: 5073
Loading cache: 100% [#####] Time: 0:00:09
Loaded 6330 entries from dependency cache.
Parsing recipes: 100% [#####] Time: 0:00:06
Parsing of 4400 .bb files complete (4391 cached, 9 parsed). 6339 targets, 392 skipped, 1 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% [#####] Time: 0:00:05
Sstate summary: Wanted 0 Local 0 Mirrors 0 Missed 0 Current 0 (0% match, 0% complete)
NOTE: No setscene tasks
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 3 tasks of which 0 didn't need to be rerun and all succeeded.
[INFO] bitbake virtual/kernel -c menuconfig
NOTE: Started PRServer with DBfile: /home/hamza/petalinux/2023.2/TransactionProcessor/build/cache/prserv.sqlite3, Address: 127.0.0.1:46275, PID: 6013
Loading cache: 100% [#####] Time: 0:00:03
Loaded 6330 entries from dependency cache.
Parsing recipes: 100% [#####] Time: 0:00:05
Parsing of 4400 .bb files complete (4391 cached, 9 parsed). 6339 targets, 392 skipped, 1 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% [#####] Time: 0:00:05
Sstate summary: Wanted 70 Local 0 Mirrors 69 Missed 1 Current 67 (98% match, 99% complete)
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 670 tasks of which 608 didn't need to be rerun and all succeeded.
[INFO] bitbake virtual/kernel -c diffconfig
NOTE: Started PRServer with DBfile: /home/hamza/petalinux/2023.2/TransactionProcessor/build/cache/prserv.sqlite3, Address: 127.0.0.1:39661, PID: 13056
Loading cache...done.
Loaded 6330 entries from dependency cache.
Parsing recipes...done.
Parsing of 4400 .bb files complete (4391 cached, 9 parsed). 6339 targets, 392 skipped, 1 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks...done.
Sstate summary: Wanted 35 Local 0 Mirrors 35 Missed 0 Current 33 (100% match, 100% complete)
NOTE: Executing Tasks
NOTE: Running task 310 of 310 (/home/hamza/petalinux/2023.2/TransactionProcessor/components/yocto/layers/meta-xilinx/meta-xilinx-core/recipes-kernel/linux/linux-6.1.30.bb:do_diffconfig)
NOTE: recipe linux-xlnx-6.1.30-xilinx-v2023.2+gitAUTOINC+a19da07cfs-r0: task do_diffconfig: Started
NOTE: recipe linux-xlnx-6.1.30-xilinx-v2023.2+gitAUTOINC+a19da07cfs-r0: task do_diffconfig: Succeeded
NOTE: Tasks Summary: Attempted 310 tasks of which 309 didn't need to be rerun and all succeeded.
[INFO] bitbake virtual/kernel -c cleansstate
NOTE: Started PRServer with DBfile: /home/hamza/petalinux/2023.2/TransactionProcessor/build/cache/prserv.sqlite3, Address: 127.0.0.1:43963, PID: 13955
Loading cache: 100% [#####] Time: 0:00:03
Loaded 6330 entries from dependency cache.
Parsing recipes: 100% [#####] Time: 0:00:06
Parsing of 4400 .bb files complete (4391 cached, 9 parsed). 6339 targets, 392 skipped, 1 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% [#####] Time: 0:00:05
Sstate summary: Wanted 0 Local 0 Mirrors 0 Missed 0 Current 0 (0% match, 0% complete)
NOTE: No setscene tasks
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 3 tasks of which 0 didn't need to be rerun and all succeeded.
[INFO] Successfully configured kernel
TransactionProcessor
```

Figure 4.7: Kernel Configuration

Here you can customize the kernel to include only the necessary features and drivers for your specific application and hardware, resulting in a smaller and more efficient kernel. One can also enable or disable kernel features to optimize performance for your embedded system.

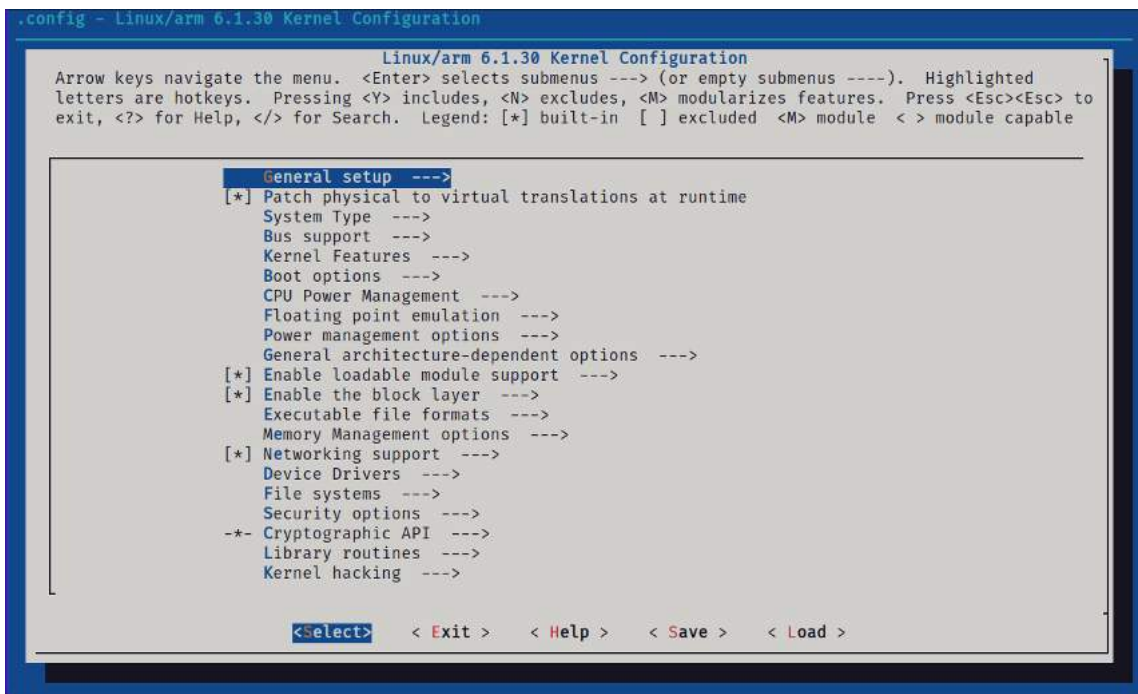


Figure 4.8: Linux/Arm 6.1.30 Kernel Configuration

4.4.2 Boot Configuration

Petalinux configuration command with the u-boot option is:

```
petalinux-config -c u-boot
```

This will launch the configuration menu specific to the bootloader. U-Boot largely takes up the role of turning on the hardware and as well as Linux kernel in the process of booting. The command opens a menu interface, where you can navigate through various configuration options for U-Boot.

```

TransactionProcessor petalinux-config -c u-boot
[INFO] Sourcing buildtools
[INFO] Silentconfig project
[INFO] Silentconfig rootfs
[INFO] Generating workspace directory
[INFO] Configuring: u-boot
[INFO] bitbake virtual/bootloader -c menuconfig
NOTE: Started PRServer with DBfile: /home/hamza/petalinux/2023.2/TransactionProcessor/build/cache/prserv.sqlite3, Address: 127.0.0.1:34731, PID: 15398
Loading cache: 100% [#####] Time: 0:00:03
Loaded 6330 entries from dependency cache.
Parsing recipes: 100% [#####] Time: 0:00:06
Parsing of 4400 .bb files complete (4391 cached, 9 parsed). 6339 targets, 392 skipped, 1 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% [#####] Time: 0:00:05
Sstate summary: Wanted 89 Local 0 Mirrors 87 Missed 2 Current 87 (97% match, 98% complete)
Removing 2 stale sstate objects for arch zynq_generic/2010: 100% [#####] Time: 0:00:00
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 788 tasks of which 772 didn't need to be rerun and all succeeded.
[INFO] bitbake virtual/bootloader -c diffconfig
NOTE: Started PRServer with DBfile: /home/hamza/petalinux/2023.2/TransactionProcessor/build/cache/prserv.sqlite3, Address: 127.0.0.1:35471, PID: 17378
Loading cache...done.
Loaded 6330 entries from dependency cache.
Parsing recipes...done.
Parsing of 4400 .bb files complete (4391 cached, 9 parsed). 6339 targets, 392 skipped, 1 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks...done.
Sstate summary: Wanted 0 Local 0 Mirrors 0 Missed 0 Current 0 (0% match, 0% complete)
NOTE: No setscene tasks
NOTE: Executing Tasks
NOTE: Running task 1 of 1 (/home/hamza/petalinux/2023.2/TransactionProcessor/components/yocto/layers/meta-xilinx/meta-xilinx-core/recipes-bsp/u-boot/u-boot-xlnx.bb::do_diffconfig)
NOTE: recipe u-boot-xlnx-1 v2023.01-xilinx-v2023.2+gitAUTOINC-0fc19cad5a-r0: task do_diffconfig: Started
NOTE: recipe u-boot-xlnx-1 v2023.01-xilinx-v2023.2+gitAUTOINC-0fc19cad5a-r0: task do_diffconfig: Succeeded
NOTE: Tasks Summary: Attempted 1 tasks of which 0 didn't need to be rerun and all succeeded.
[INFO] Successfully configured u-boot
TransactionProcessor

```

Figure 4.9: U-Boot Configuration

```

.config - U-Boot 2023.01 Configuration
U-Boot 2023.01 Configuration
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ---). Highlighted
letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to
exit, <?> for Help, </> for Search. Legend: [*] built-in [ ] excluded <M> module < > module capable

*** Compiler: arm-xilinx-linux-gnueabi-gcc (GCC) 12.2.0 ***
Architecture select (ARM architecture) --->
[ ] Skip the calls to certain low level initialization functions
[ ] Skip the calls to certain low level initialization functions
[ ] Skip the calls to certain low level initialization functions
[ ] Skip the call to lowlevel_init during early boot ONLY
[ ] Skip the call to lowlevel_init during early boot ONLY
ARM architecture --->
[ ] NXP ESBC (secure boot) functionality
*** Other functionality shared between NXP SoCs ***
General setup --->
API --->
Boot options --->
Console --->
Logging --->
Init options --->
Security support --->
Update support --->
Blob list --->
[*] Enable SPL
SPL configuration options --->
v(+)

<Select> < Exit > < Help > < Save > < Load >

```

Figure 4.10: U-Boot 2023.01 Configuration

There are a few configuration options related to boot media that need to be checked:

Boot Options → Boot Media → [*] Support for booting from QSPI flash

Boot Options → Boot Media → [*] Support for booting from SD/EMMC

Boot Options → Boot Media → [*] Support for booting from SD/EMMC & enable QSPI

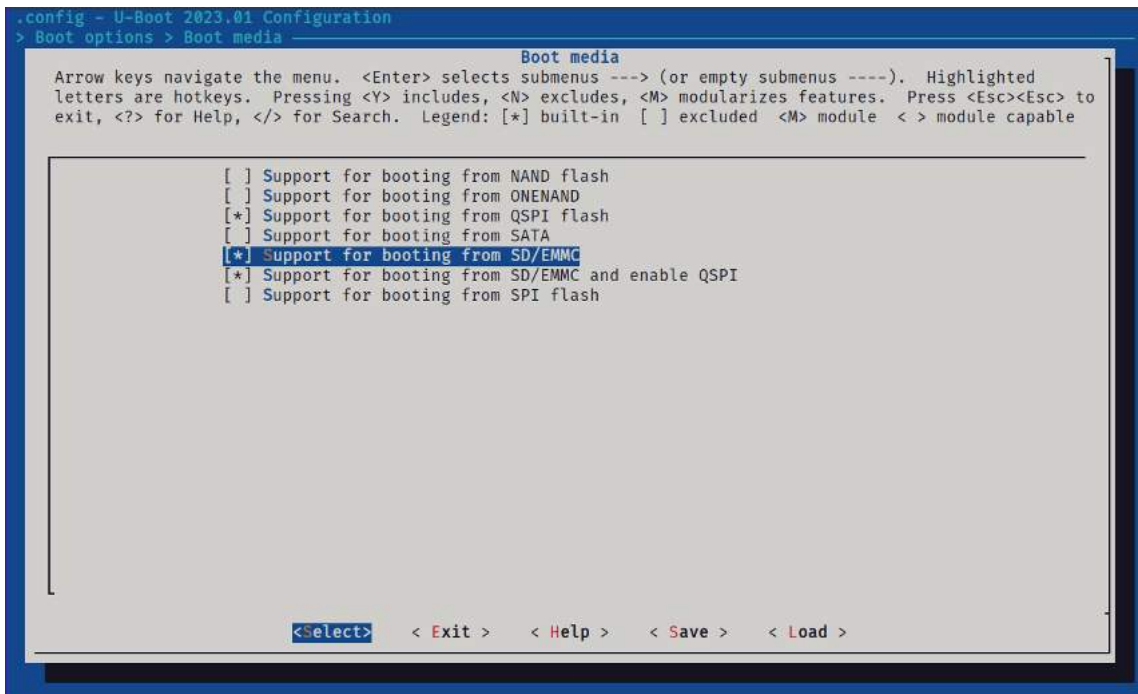


Figure 4.11: Boot Media

4.4.3 Root Filesystem Configuration

By running `petalinux-config -c rootfs`, you will be presented with root filesystem configuration options where you can modify one or many settings and/or include certain packages, libraries, and applications into the root file system of the embedded Linux system. You can customize the root filesystem to include only the necessary components for your application, resulting in a more efficient and smaller root filesystem as well as add development tools and debugging utilities that might be needed for development and troubleshooting. It is also possible to integrate custom applications and scripts into the root filesystem which will be helpful for IP integration later on.

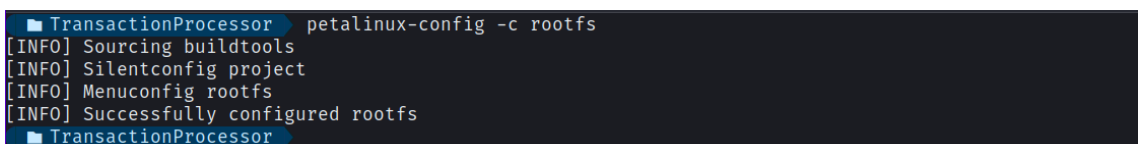


Figure 4.12: RootFS Configuration

There are a few configuration options related to Filesystem Packages, User Packages and RootFS Settings that need to be checked:

Filesystem Packages → admin → sudo → [*] sudo

Filesystem Packages → base → busybox → [*] busybox

Filesystem Packages → base → iproute2 → [*] iproute2

Filesystem Packages → base → opkg → [*] opkg

Filesystem Packages → base → shell → bash → [*] bash

Filesystem Packages → base → tar → [*] tar

Filesystem Packages → base → tzdata → [*] tzdata, tzdata-asia

Filesystem Packages → base → util-linux → [*] util-linux-umount, util-linux-mkfs, util-linux-mount, util-linux-fdisk

Filesystem Packages → base → xz → [*] xz

Filesystem Packages → console → network → curl → [*] curl

Filesystem Packages → console → network → ethtool → [*] ethtool

Filesystem Packages → console → network → wget → [*] wget

Filesystem Packages → console → tools → parted → [*] parted

Filesystem Packages → console → utils → git → [*] git

Filesystem Packages → console → utils → grep → [*] grep

Filesystem Packages → console → utils → gzip → [*] gzip

Filesystem Packages → console → utils → sysstat → [*] sysstat

Filesystem Packages → console → utils → unzip → [*] unzip

Filesystem Packages → console → utils → gzip → [*] gzip

Filesystem Packages → console → utils → vim → [*] vim

Filesystem Packages → console → utils → zip → [*] zip

Filesystem Packages → devel → make → [*] make

Filesystem Packages → devel → mpfr → [*] mpfr, mpfr-dev

Filesystem Packages → libs → libmpc → [*] libmpc, libmpc-dev

Filesystem Packages → misc → gcc-runtime → [*] libstdc++, libstdc++-dev

Filesystem Packages → misc → glibc → [*] glibc, glibc-dev

Filesystem Packages → misc → packagegroup-core-buildessential → [*] packagegroup-core-buildessential, packagegroup-core-buildessential-dev

Filesystem Packages → misc → python3 → [*] python3

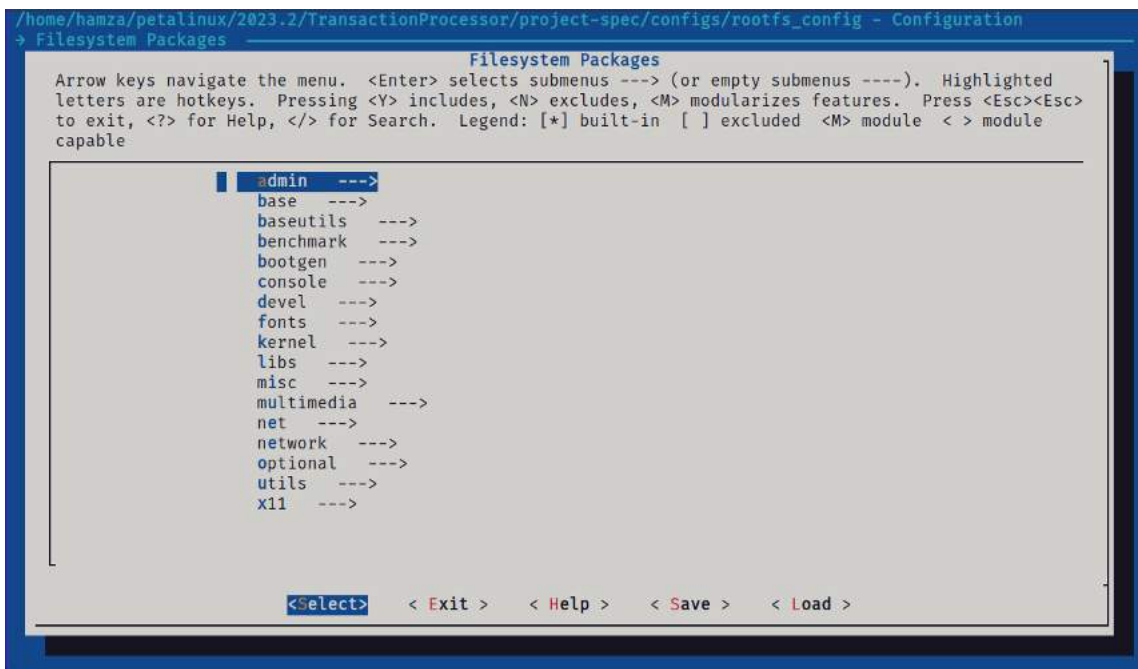


Figure 4.13: Filesystem Packages

Petalinux usually asks for a new password every time it is booted up. To prevent this behaviour, the passwd-expire field should be omitted [11].

PetaLinux RootFS Settings → (root:root;petalinux:<your-passwd>[:passwd-expire];)

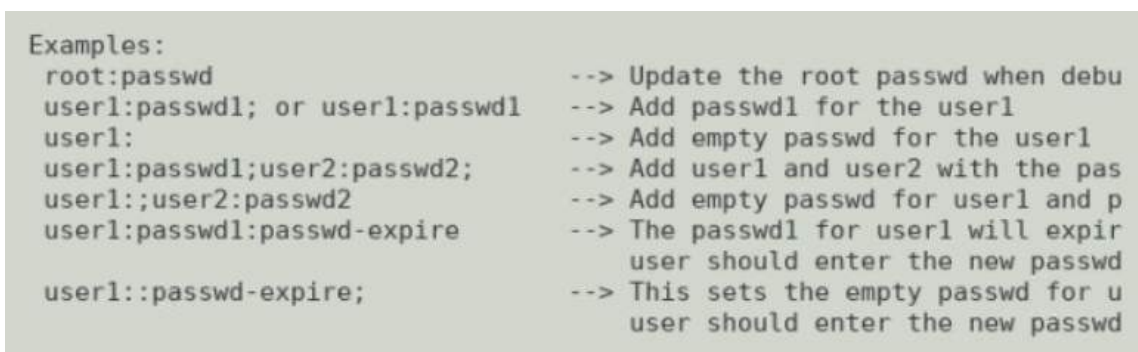


Figure 4.14: Password Rules

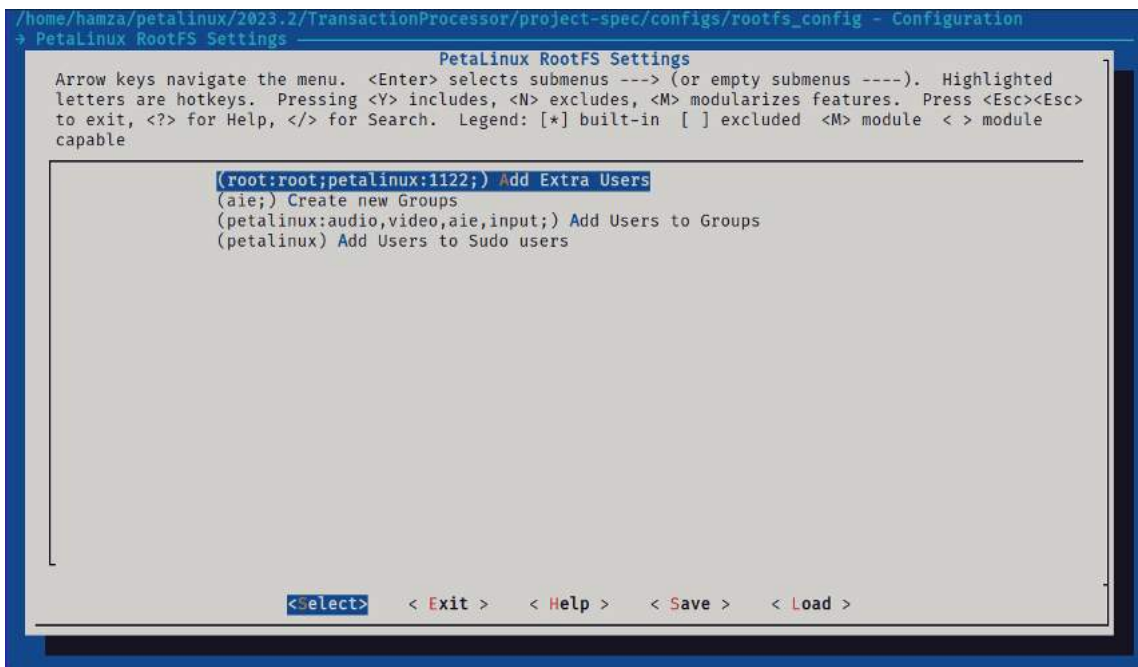


Figure 4.15: Petalinux RootFS Settings

4.4.4 Image Build

This step generates a device tree DTB file, a first stage boot loader (for Zynq 7000 devices), U-Boot, the Linux kernel, a root file system image, and the U-Boot boot script (boot.scr). Finally, it generates the necessary boot images.

```
petalinux-build
```

This command compiles all the configured components and generates the output files needed to boot and run the embedded Linux system on the target hardware. This process also includes applying any patches and integrating custom applications and drivers specified during configuration.

The full compilation log build. is saved in the build sub-folder of your PetaLinux project, To review the log information in the build sub-folder of your PetaLinux project, you can use any text editor. The final image, `<project-root>/images/linux/image.ub`, is a FIT image.

```

TransactionProcessor petalinux-build
[INFO] Sourcing buildtools
[INFO] Building project
[INFO] Silentconfig project
[INFO] Silentconfig rootfs
[INFO] Generating workspace directory
[INFO] bitbake petalinux-image-minimal
NOTE: Started PRServer with DBfile: /home/hanza/petalinux/2023.2/TransactionProcessor/build/cache/prserv.sqlite3, Address: 127.0.0.1:33881, PID: 19274
Loading cache: 100% |#####| Time: 0:00:03
Loaded 6330 entries from dependency cache.
Parsing recipes: 100% |#####| Time: 0:00:06
Parsing of 4400 .bb files complete (4391 cached, 9 parsed). 6339 targets, 392 skipped, 1 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |#####| Time: 0:00:18
Checking sstate mirror object availability: 100% |#####| Time: 0:02:06
Sstate Summary: Wanted 648 Local 15 Mirrors 588 Missed 45 Current 1457 (93% match, 97% complete)
WARNING: The busybox:do_fetch sig is computed to be 823772d40ea7b476ed771f0b124976c714bd25d931a1a3dcd20fee9003aebed, but the sig is locked to 4dcaff8cb14384
0f31476d3041aac2d22474848e638e2fe9ebaba in SIGGEN_LOCKEDSIGs_t-cortexa912hf-neon
The busybox:do_unpack sig is computed to be 996099871bb8c279b3e65a8934bea00971ab4ffd74a630e226e7041748c51648, but the sig is locked to 69213b70c4cb6f6d6c9020
9efbba40a4d88e6b7f85799e4fb31a in SIGGEN_LOCKEDSIGs_t-cortexa9c2hf-neon
The busybox:do_populate_ltc sig is computed to be 8532b4733169f8f0nc7ndb19c57c7047b9e8ed5673f6ff045ce03ce8f3259867, but the sig is locked to ce23a93f6a51b28a
9e4d7233ff7d0189e77a34c0b84f4fc0 in SIGGEN_LOCKEDSIGs_t-cortexa9c2hf-neon
The busybox:do_package sig is computed to be 82cc81e19902b36764709f45c0123775b9c3c0937730e2f37e2c385035becf2e, but the sig is locked to 57c0be2920b897b78d80c
c9b03a037b64d984f51808d7218c04 in SIGGEN_LOCKEDSIGs_t-cortexa912hf-neon
Removing 31 sstate objects for arch zynq_generic_7z010: 100% |#####| Time: 0:00:01
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 5604 tasks of which 5486 didn't need to be rerun and all succeeded.

Summary: There was 1 WARNING message.
INFO: copy to TFTP-boot directory is not enabled !!
[INFO] Successfully built project
TransactionProcessor

```

Figure 4.16: Petalinux Build

The kernel image (including RootFS, initramfs, if any,) is zImage for Zynq 7000 devices. The build images are located in the <project-root>/images/linux directory.

4.5 Packaging & Booting

Petalinux Packaging feature is invoked to create an image from the necessary boot components belonging to a PetaLinux project.

```

petalinux-package --boot --fsbl ./zynq\_fsbl. elf --fpga
./system.bit --u-boot

```

This command adds the First Stage Boot Loader FSBL, the FPGA bitstream image, and the U-Boot bootloader to the BOOT.BIN file. The BOOT.BIN file is used to boot the embedded system. This file is necessary to boot the embedded system, typically from an SD card or other bootable media.

```
linux petalinux-package --boot --fsbl ./zynq_fsbl.elf --fpga ./system.bit --u-boot --force
WARNING: Fpga Manager enabled, skipping bitstream to pack into BOOT.BIN
[INFO] Sourcing buildtools
INFO: File in BOOT BIN: "/home/hamza/petalinux/2023.2/TransactionProcessor/images/linux/zynq_fsbl.elf"
INFO: Generating zynq binary package BOOT.BIN...

***** Bootgen v2023.2
**** Build date : Sep  4 2023-15:57:12
** Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.
** Copyright 2022-2023 Advanced Micro Devices, Inc. All Rights Reserved.

[INFO] : Bootimage generated successfully
INFO: Binary is ready.
linux
```

Figure 4.17: Petalinux Boot Package

4.5.1 SD Card Partitioning

In order to prepare an SD card for use with the target, specific commands made available through an embedded Linux terminal, one can issue partition and format the card using the `fdisk` and `mkfs` facilities.

Open `fdisk` for the SD card: `sudo fdisk /dev/sdb`

Create first and second primary partitions by allocating the necessary sector sizes [\[12\]](#).

Format the first partition (FAT32): `sudo mkfs.vfat /dev/sdb1`

Format the second partition (EXT4): `sudo mkfs.ext4 /dev/sdb2`

4.5.2 Loading Files

Mount the FAT32 partition: `sudo mount /dev/sdb1 /mnt`

Copy boot files to the FAT32 partition:

```
sudo cp BOOT.BIN image.ub boot.scr /media/SD_CARD
```

Unmount the FAT32 partition: `sudo umount /mnt`

Mount the EXT4 partition: `sudo mount /dev/sdb2 /mnt`

Extract the root filesystem to the EXT4 partition:

```
sudo tar -xzvf /home/hamza/petalinux/2023.2/FYP/images/
```



```
linux/rootfs.tar.gz
```

Unmount the EXT4 partition: `sudo umount /mnt`

Now the SD Card is ready to be booted into the Zybo Board.

- Remove the microSD card from your computer and then place the microSD card in the connector J4 available on the ZYBO.
- Connect an external power supply and use JP7 to choose it.
- Insert one end of a jumper wire to the left most connector on the JP5 labeled “SD”.
- Turn the board on. The board will now proceed to load the data on the microSD card.

Chapter 5

IP Generation & Integration

Introduction

The Vivado IP integrator displays a design canvas to let you quickly create complex sub-system designs by integrating IP cores. It lets you create complex system designs by instantiating and interconnecting IP cores from the Vivado IP catalog. You will typically construct designs at the AXI-interface level for greater productivity. We have created the hashing and encryption algorithm in Vivado Design Suite. For hashing and encryption, we have taken the codes online, therefore, we will not discuss the code here. Now, we will create a custom IP for both of the codes.

5.1 SHA-256 IP

Let's start by looking at how we can create the Custom IP for the hashing (SHA-256) function. Let's consider that the SHA-256 Algorithm is already implemented. By following the below steps, we can create the IP.

5.1.1 Creating SHA256 IP and Block Design

After creating a SHA256 Algorithm, we want to use it for getting data from the outside world, and we can do this by creating a Custom IP of SHA256 using the "Create and Package IP" option. For that purpose, follow the below steps:

1. In Vivado, go to Tools and select "Create and Package new IP", as shows in Figure:5.1:

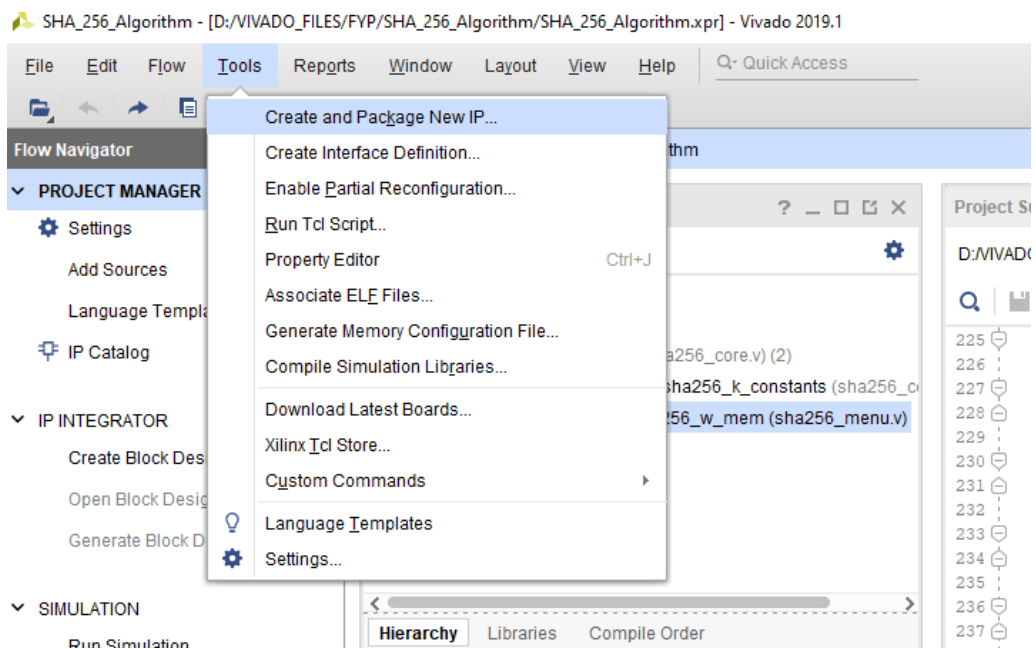


Figure 5.1: Create and Package IP

2. Hit Next → select "create a new AXI4 peripheral" → Give Name to your IP → Select Interface Type & Number of Registers to be used. In our case, the type is AXI Lite and registers = 20. And finally select "Edit IP" option and hit Finish. The steps are shown in Figure 5.2:

Create and Package New IP ×

Create Peripheral, Package IP or Package a Block Design

Please select one of the following tasks.

Packaging Options

- Package your current project
Use the project as the source for creating a new IP Definition.
- Package a block design from the current project
Choose a block design as the source for creating a new IP Definition.
Select a block design:
- Package a specified directory
Choose a directory as the source for creating a new IP Definition.

Create AXI4 Peripheral

- Create a new AXI4 peripheral
Create an AXI4 IP, driver, software test application, IP Integrator AXI4 VIP simulation and debug demonstration design.

Create and Package New IP ×

Peripheral Details

Specify name, version and description for the new peripheral

Name:

Version:

Display name:

Description:

IP location: ...

Overwrite existing

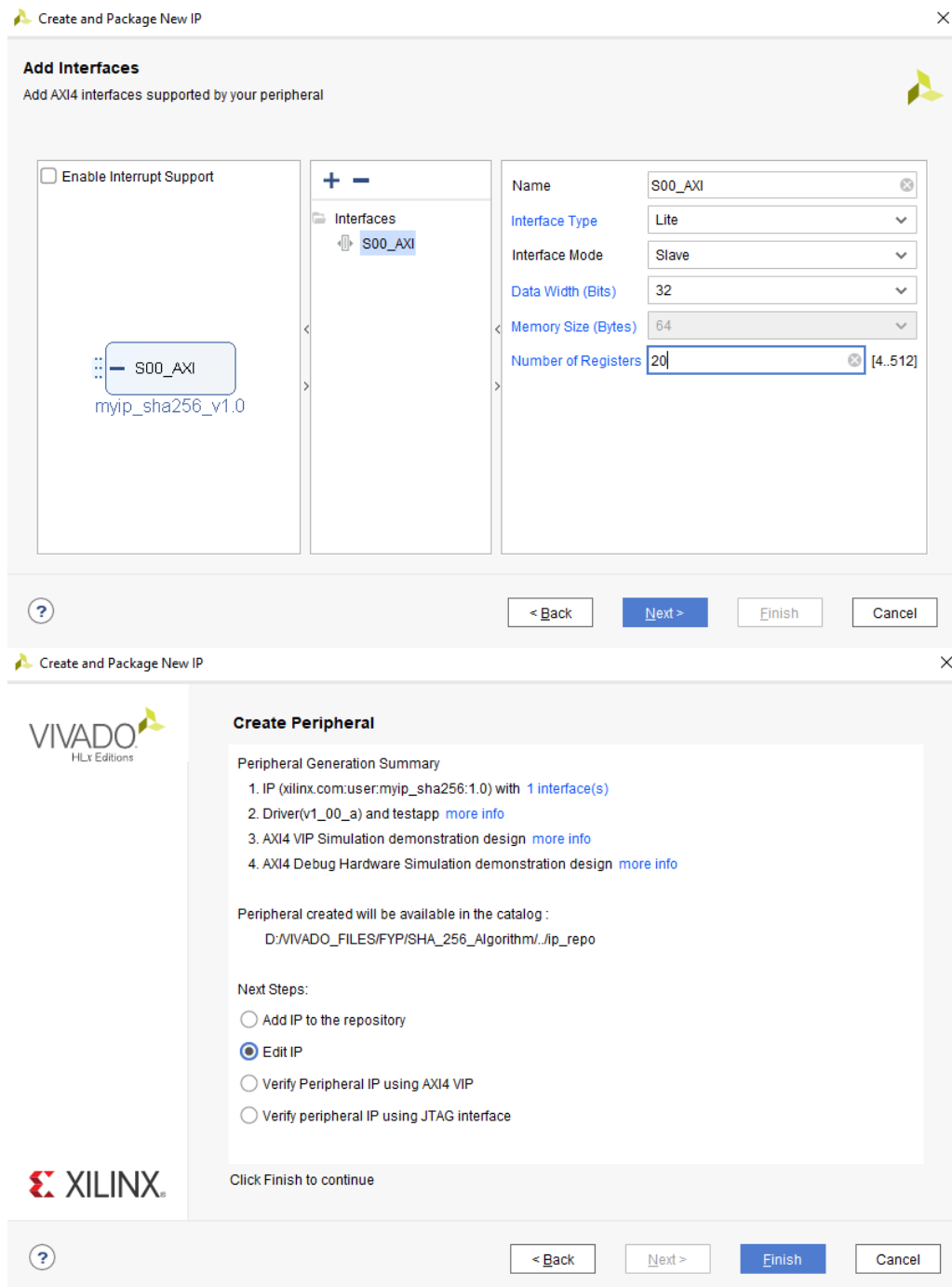
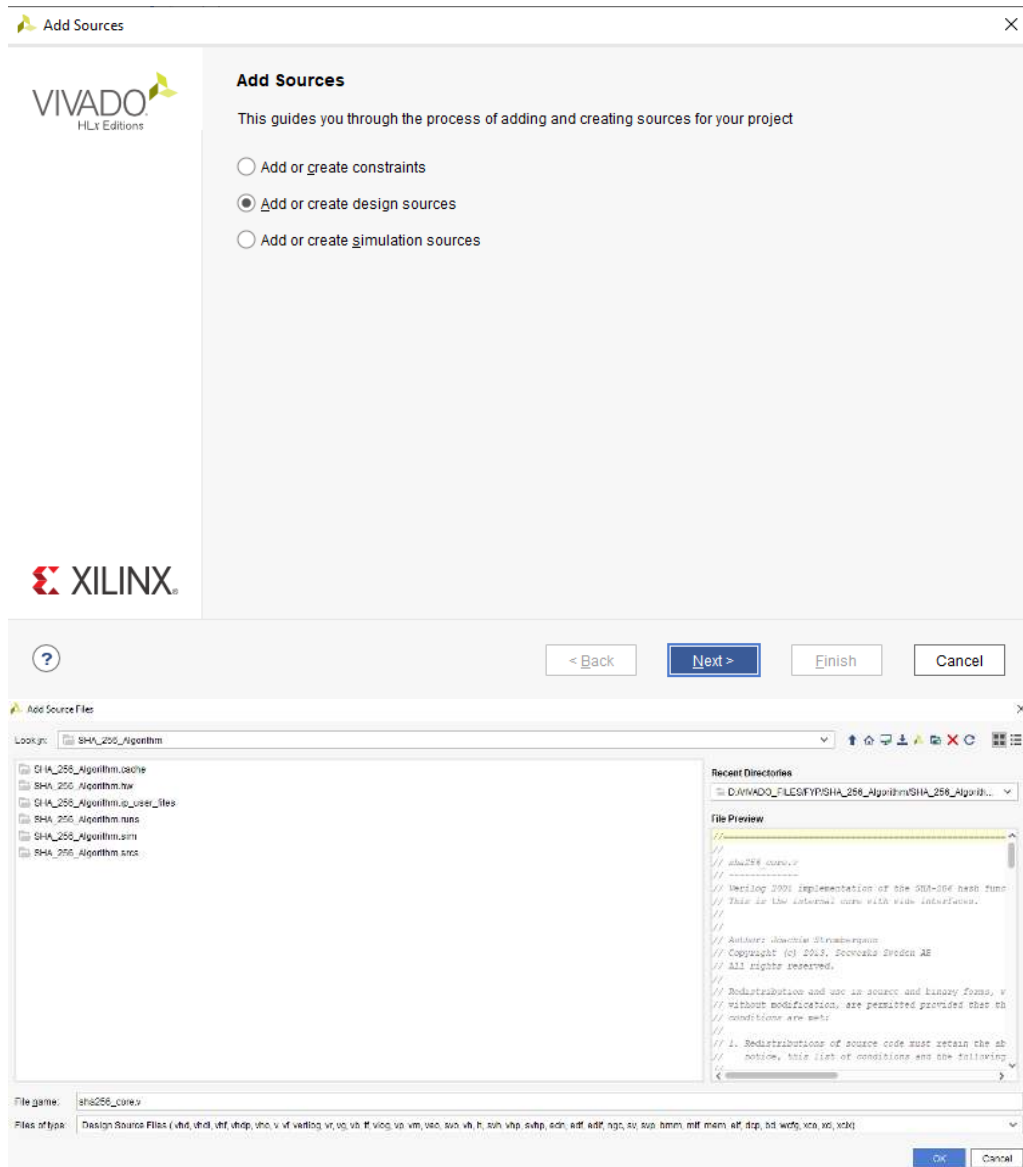


Figure 5.2: Steps to create new IP

3. A new window will open. Import the sha256_algorithm module that you created earlier by clicking on **Add sources** or pressing **Alt+A**. Select **Add Design Source** and click **Next**. Click on **Add Files** and select the project where you

have written the SHA256 algorithm code. Navigate to the `.srcs` folder, then to `sources`, then `new`, and select the files to be imported, as shown in Figure:5.3. Click **Finish**.



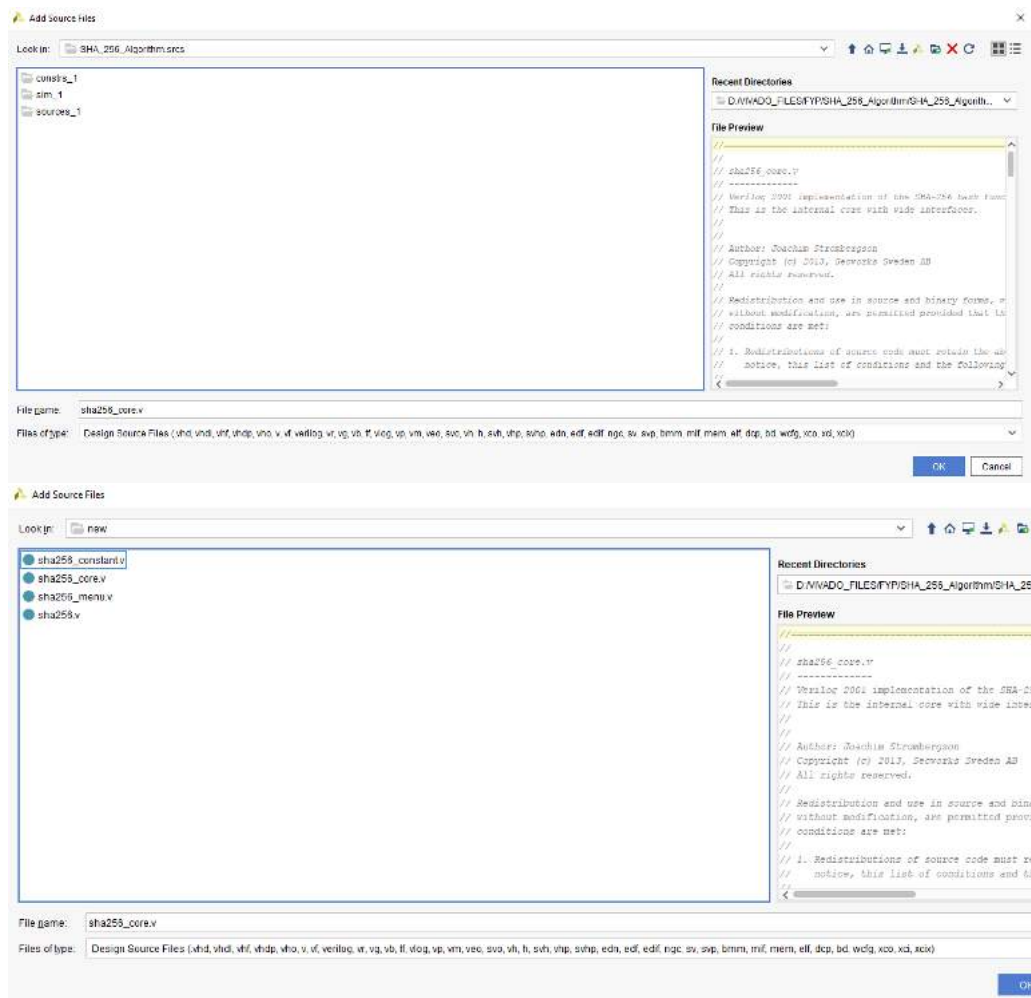


Figure 5.3: Adding the IPs directories to the project

4. Open the `myip_s00_axi` file and For outputs, add a few more registers as required by your Top Module below the "number of slave reg" at line 106. Remember to use `wire` as the type for all outputs, as shown in Figure 5.4.

```

reg [C_S_AXI_DATA_WIDTH-1:0]    reg_data_out;
integer byte_index;
reg  aw_en;

reg [2:0] controls;
wire [31:0] read_data;
wire error;
wire [255:0] OUTPUT;
wire [5:0] ready;
wire [2:0] init_controls;
wire [1:0] cs_we_check;
wire [7:0] address_check;
reg reset_n;
wire reset_check;
wire digest_ready;

```

Figure 5.4: Adding required registers

- Each register in the IP is 32 bits wide. They are denoted as `slv_reg0` to `slv_reg19` in the code, all pre-defined. To incorporate your custom registers, you need to add them in the "Address decoding for reading registers" section where the case statement is being used, typically at line 544 in your case. A glimpse of it is shown in Figure:5.5:


```

always @(*)
begin
    // Address decoding for reading registers
    case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
        5'h00 : reg_data_out <= slv_reg0;
        5'h01 : reg_data_out <= slv_reg1;
        5'h02 : reg_data_out <= slv_reg2;
        5'h03 : reg_data_out <= read_data;
        5'h04 : reg_data_out <= error;
        5'h05 : reg_data_out <= OUTPUT[255:224];
        5'h06 : reg_data_out <= OUTPUT[223:192];
        5'h07 : reg_data_out <= OUTPUT[191:160];
        5'h08 : reg_data_out <= OUTPUT[159:128];
        5'h09 : reg_data_out <= OUTPUT[127:96];
        5'h0A : reg_data_out <= OUTPUT[95:64];
        5'h0B : reg_data_out <= OUTPUT[63:32];
        5'h0C : reg_data_out <= OUTPUT[31:0];
        5'h0D : reg_data_out <= slv_reg13;
        5'h0E : reg_data_out <= ready;
        5'h0F : reg_data_out <= init_controls;
        5'h10 : reg_data_out <= cs_we_check;
        5'h11 : reg_data_out <= address_check;
        5'h12 : reg_data_out <= digest_ready;
        5'h13 : reg_data_out <= reset_check;
        default : reg_data_out <= 0;
    endcase
end

```

Figure 5.5: Replacing some `slv_reg`

6. Each case is called sequentially after every 4 bytes. To add input to `slv_reg0`, you must increment the base address by 4 (we will discuss this in detail later).
7. Lastly call the SHA256 algorithm top module below the "Add user logic here" section at the end of the module, as shown in Figure:5.6.

```

sha256 sha256_uut(
    .clk(S_AXI_ACLK),
    .reset_n(slv_reg0[2:2]),
    .cs(slv_reg0[0:0]),
    .we(slv_reg0[1:1]),
    .init(slv_reg0[3:3]),
    .next(slv_reg0[4:4]),
    .mode(slv_reg0[5:5]),
    .address(slv_reg1[7:0]),
    .write_data(slv_reg2),
    .read_data(read_data),
    .error(error),
    .OUTPUT(OUTPUT),
    .rounds(ready),
    .controls(init_controls),
    .cs_we_check(cs_we_check),
    .address_check(address_check),
    .reset_check(reset_check),
    .ready(digest_ready)
);

```

Figure 5.6: Adding User Logic

- Now save the file, do synthesis and click on "Edit Package IP" in the Project manager tab. A window will open up showing different things. Go to "Review and Package" and hit "Re-Package". The working is shown in Figure 5.7:

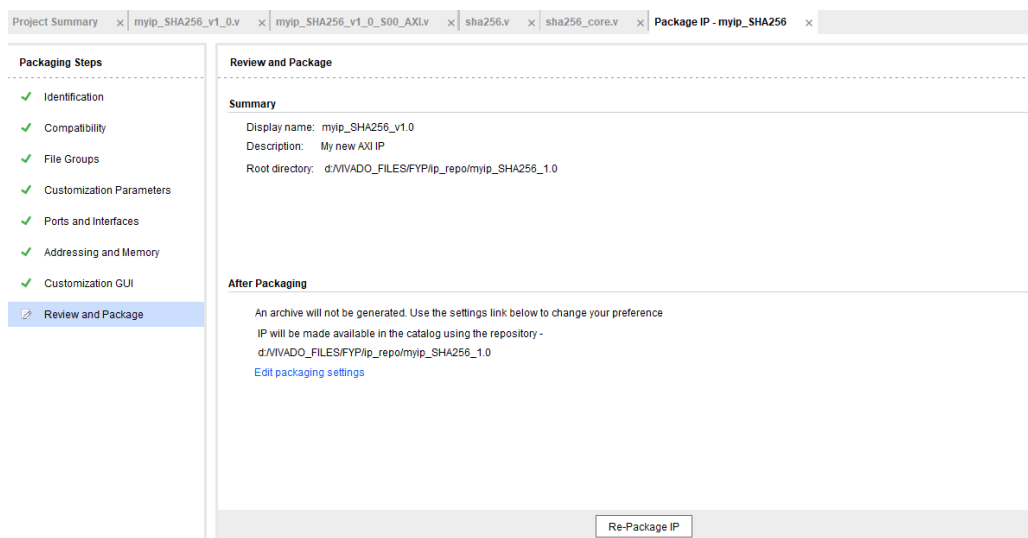
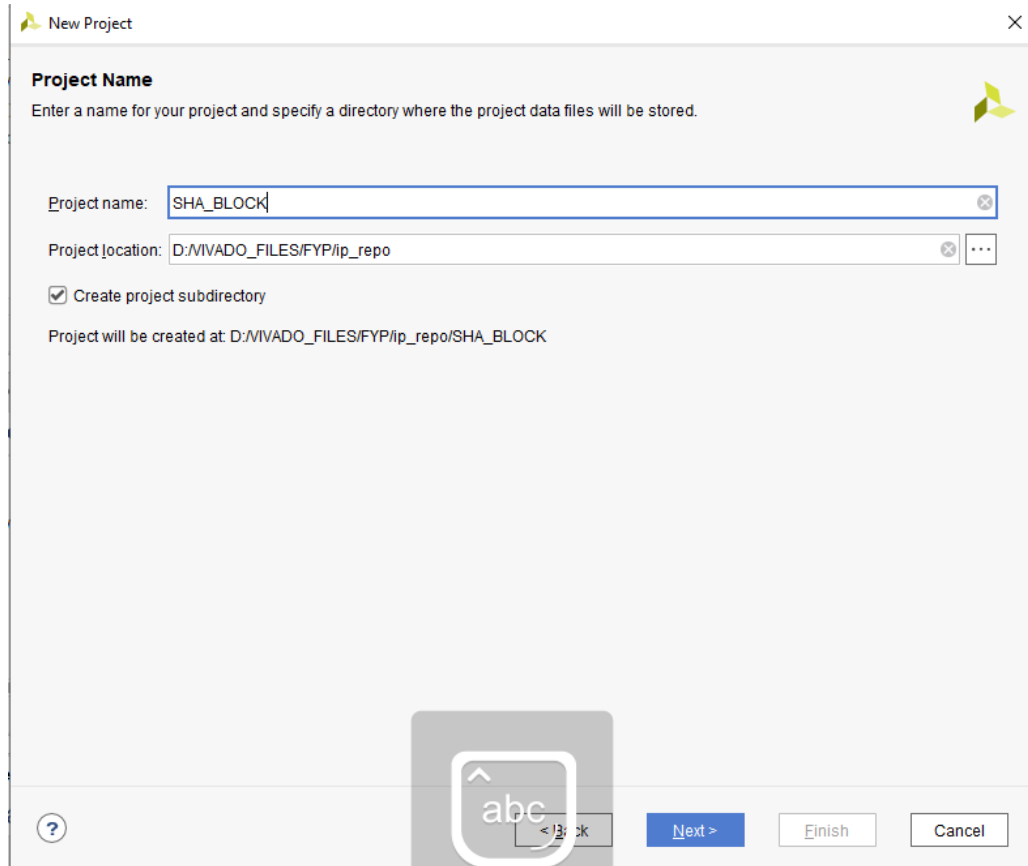


Figure 5.7: Saving Changes made in IP

9. Now create a new Project without adding any design files or constraints file. We will use it as a block diagram. Select your desired board and name for the project, as shown in Figure:5.8.



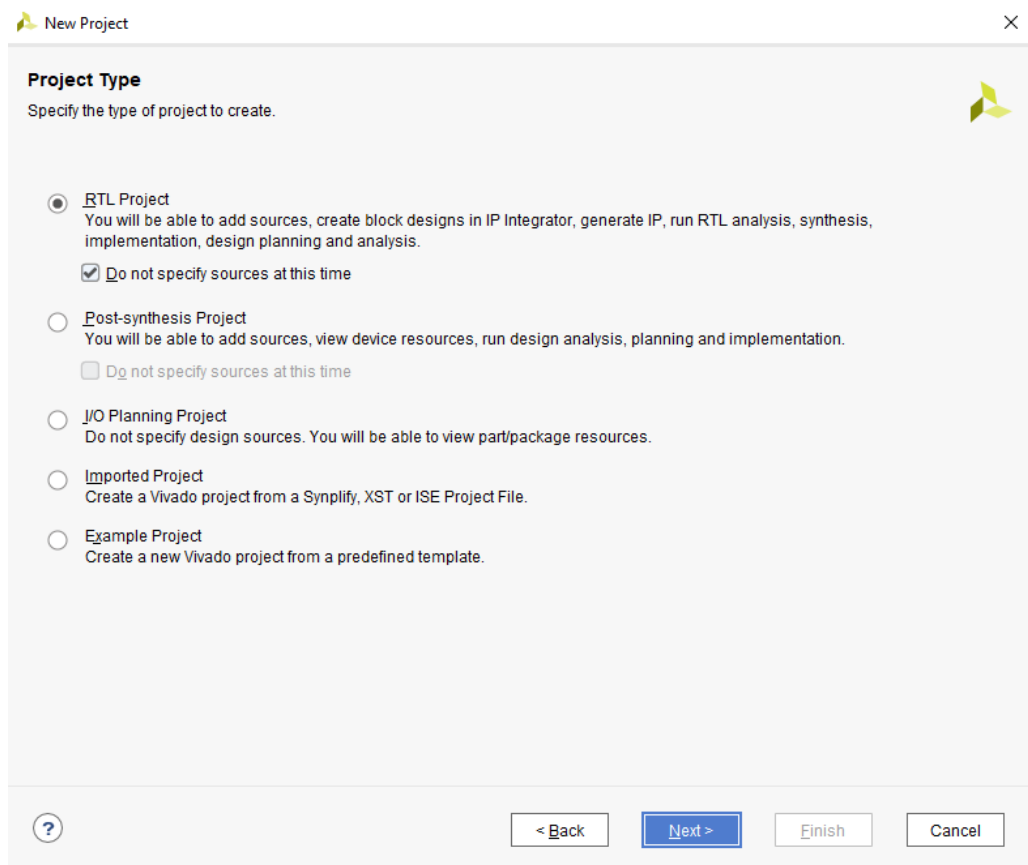


Figure 5.8: Creating Block Diagram Project

10. A new project will be created. Now create a new block diagram. Then go to Settings → IP → Repository and add the location of your Custom IP that you created earlier, shown in Figure:5.9.

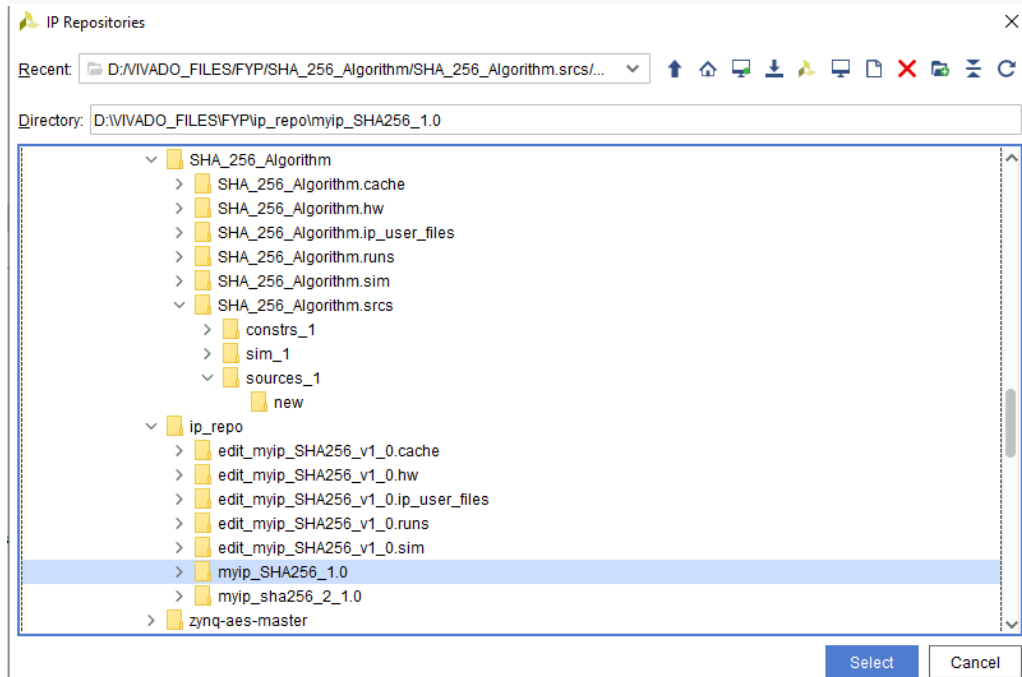
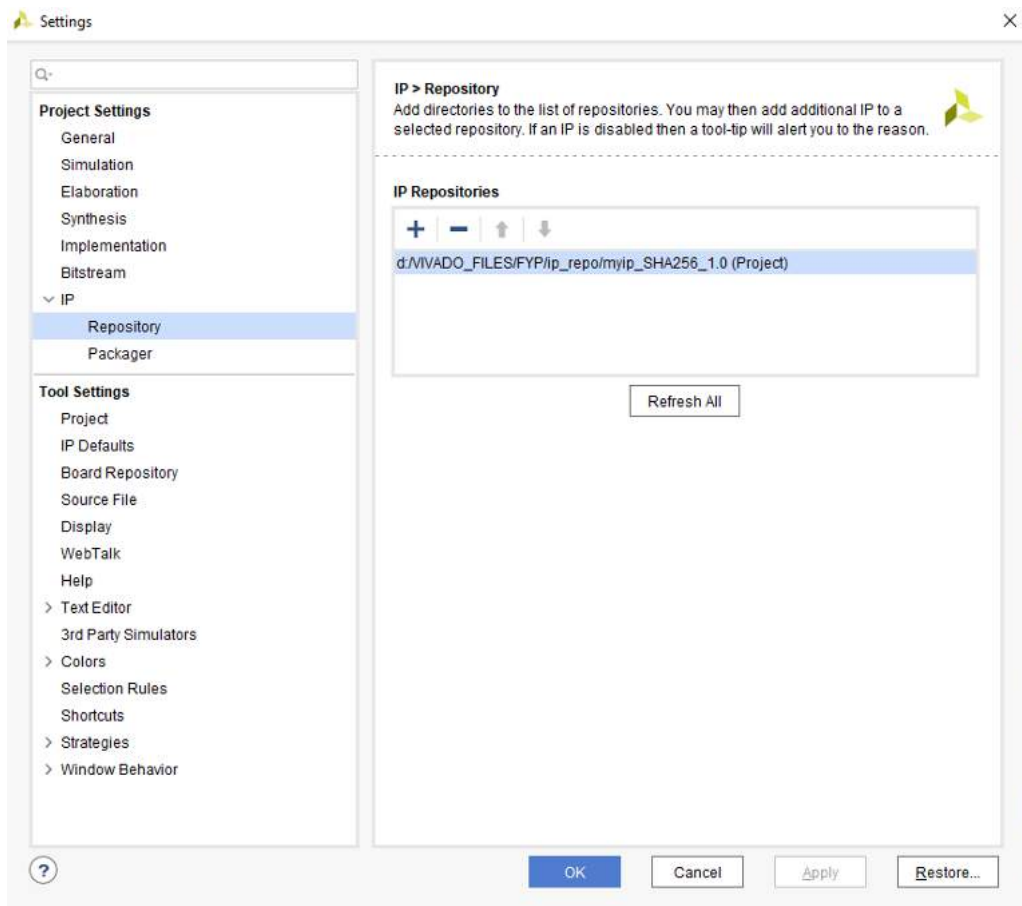


Figure 5.9: Adding IP repository

11. After adding the IP to your IP Repository, click on the Add Icon in the block design window and add the IP block that you have imported (e.g., myIP_sha256) and also add the ZYNQ Processing System. This adding of blocks is shown in Figure:5.10 and Figure:5.11

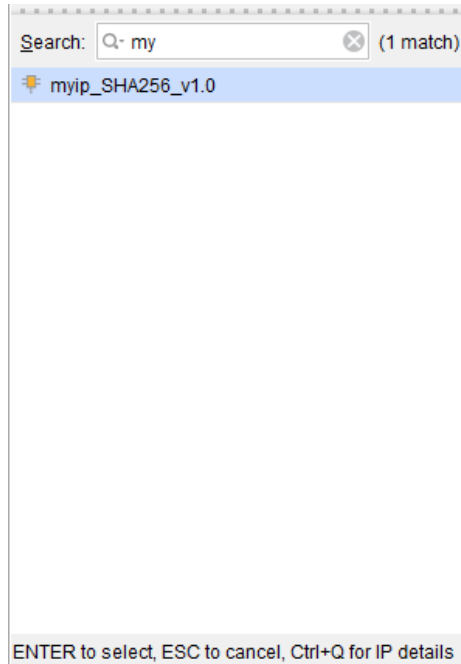


Figure 5.10: Adding IP Block

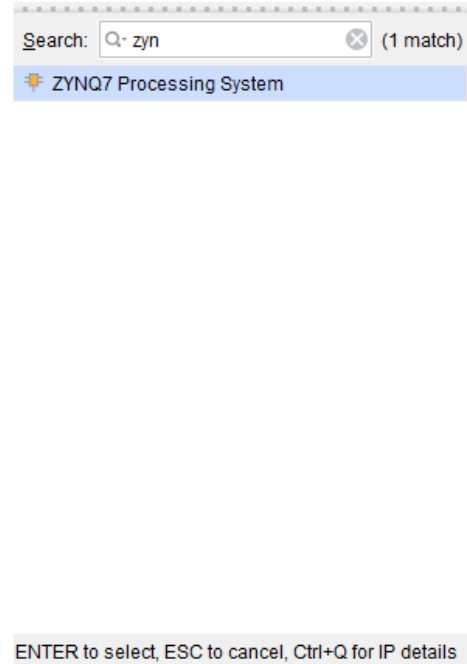


Figure 5.11: Adding Zynq Block

12. Now you will see two things at the top, "Run Block Automation" & "Run Connection Automation". First click on "Run Block Automation" and then "Run Connection Automation". After that, the final block diagram will look like the one below. The final Block Diagram is shown in Figure:5.12.

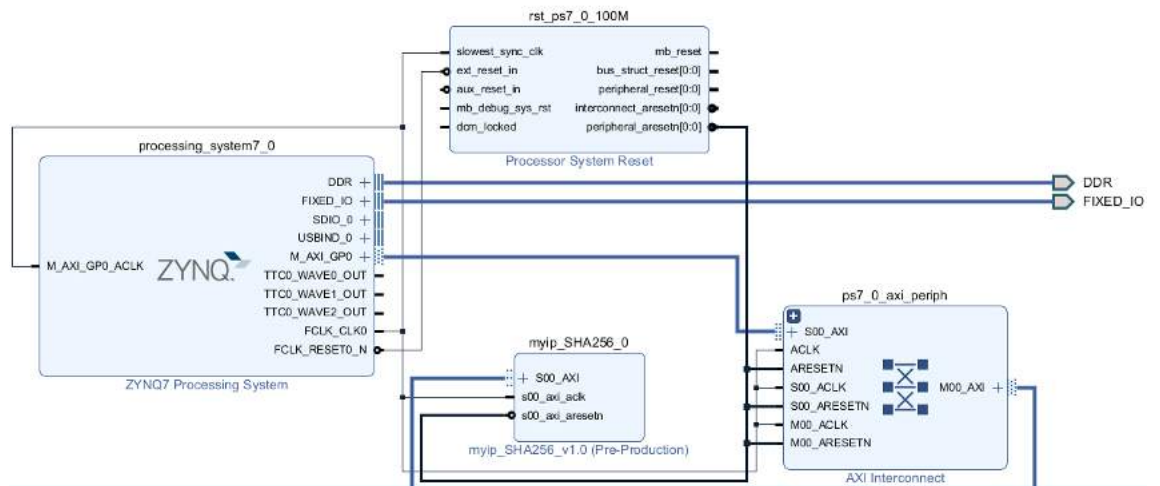


Figure 5.12: Final Block Diagram

- Now go to the Source tab. Right-click on block_1.bd and create a wrapper. After that, run synthesis, implementation, and generate bitstream. Finally, go to File → Import → Import Hardware, check the include bitstream box, and import, as shown in Figure:5.13.

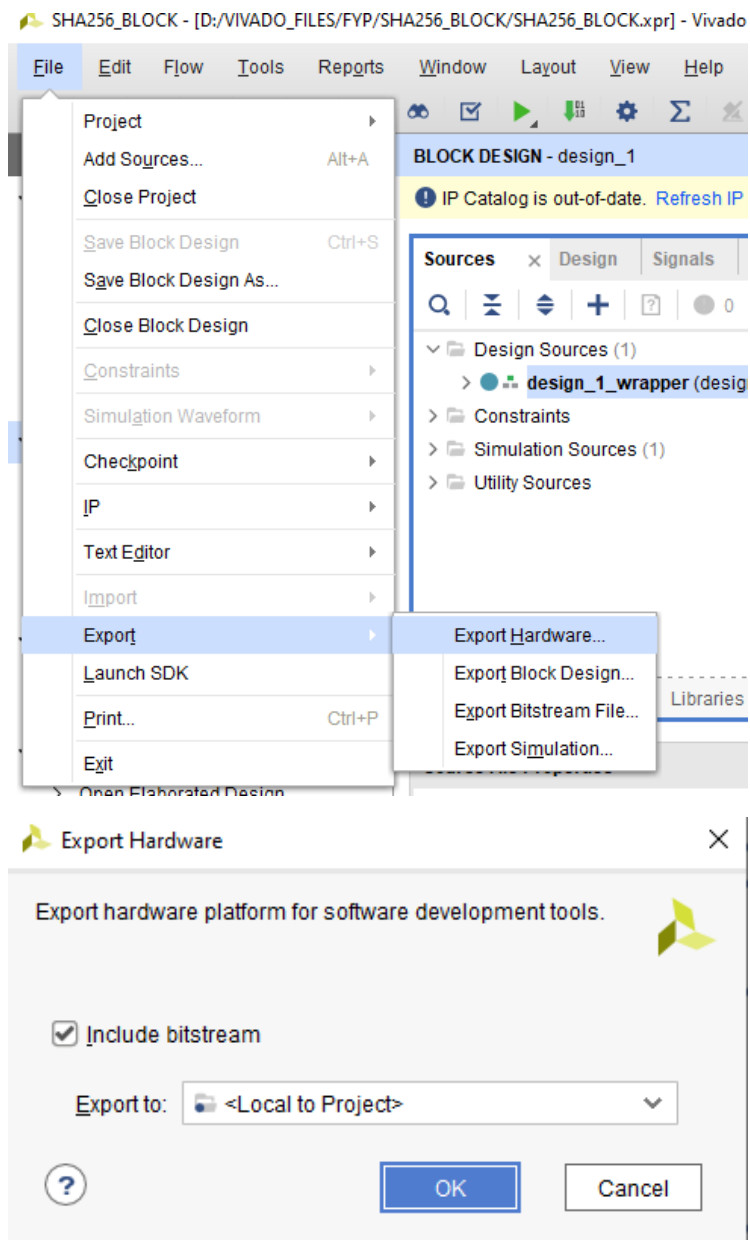


Figure 5.13: Exporting Hardware

5.1.2 Writing Code in SDK and launching It

In the field of Xilinx Vivado, an SDK (Software Development Kit) is an environment that can help in generating, as well as in testing and deploying software solutions in the Xilinx embedded systems. Vivado SDK is a component of the complete design tool called Vivado Design Suite that is designed for the FPGA and SoC development. The steps to create a SDK Code for our project is provided below.

1. After we have exported the Vivado Project of SH256 Block Diagram, we will now launch SDK. To do so, go to File → Launch SDK. This launching is shown in Figure:5.14.

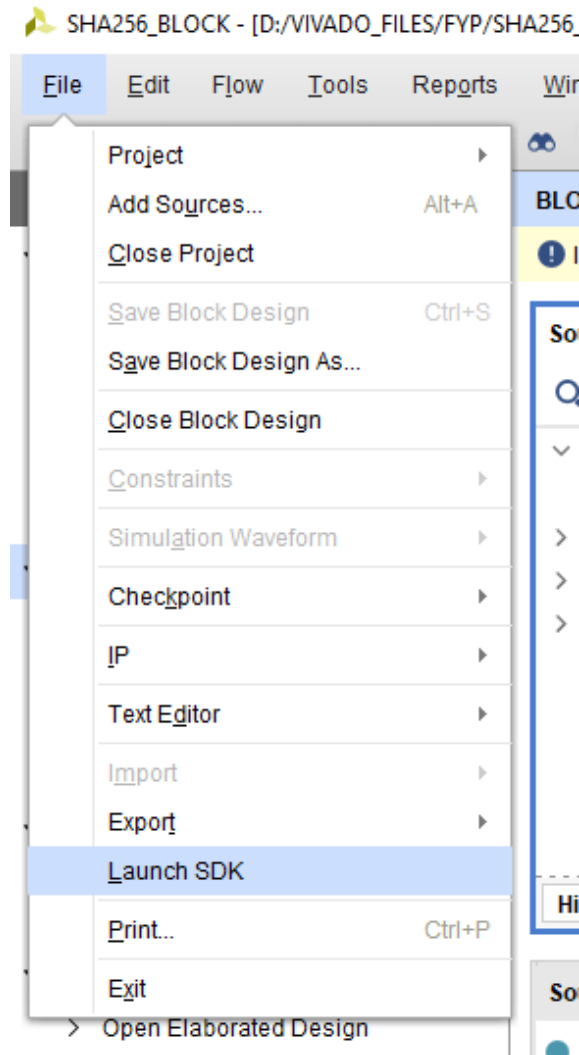


Figure 5.14: Launch SDK

2. A new window will pop up. Wait for some time till all the required dependencies gets imported according to our project. The window will look like Figure:5.15.

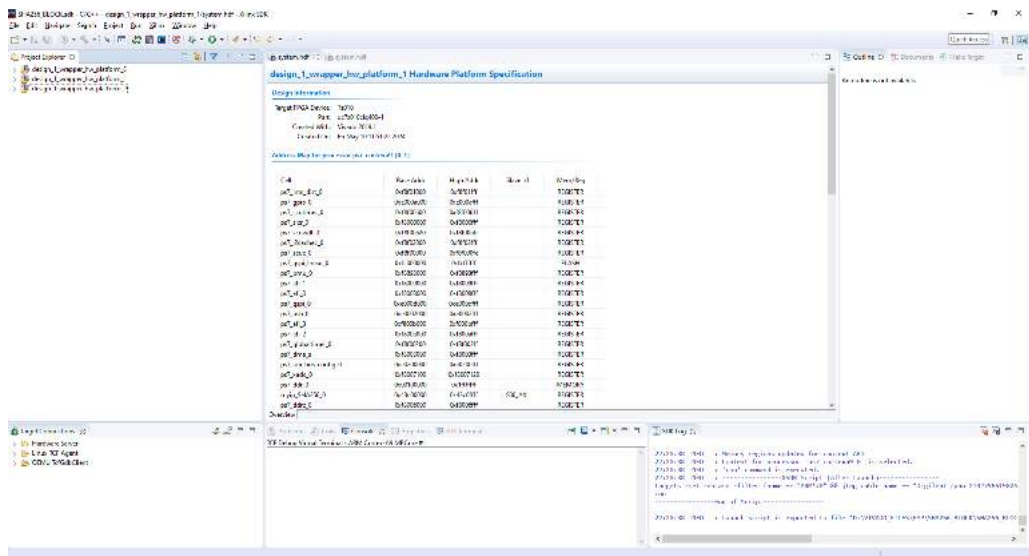


Figure 5.15: Basic SDK Window

3. After that, go to File → New → Application Project. A window about the new project will pop up. Provide a suitable name for it.
4. In OS Platform select standalone if you are working with only vivado code and select OS Platform as linux if you have booted linux on your FPGA and want to provide inputs from there.
5. In Hardware Platform select the design_wrapper that you will see on the left side of SDK Window. In Processor select the processor of your board as in my case it is cortex9. Let everything be default and click next. After that select Hello World and click finish. The creation of new project is shown in Figure:5.16

SHA256_BLOCK.sdk - C/C++ - design_1_wrapper_hw_platform_1/system.hdf - Xilinx SDK

File Edit Navigate Search Project Run Xilinx Window Help

New Alt+Shift+N >

- Open File...
- Open Projects from File System...
- Close Ctrl+W
- Close All Ctrl+Shift+W
- Save Ctrl+S
- Save As...
- Save All Ctrl+Shift+S
- Revert
- Move...
- Rename... F2
- Refresh F5
- Convert Line Delimiters To >
- Print... Ctrl+P
- Switch Workspace >
- Restart
- Import...
- Export...
- Properties Alt+Enter

1 system.hdf [design_1_wrapper_hw_pla...]

2 helloworld.c [SHA256_SW/src]

3 system.mss [SHA256_SW/bsp]

4 system.hdf [design_1_wrapper_hw_pla...]

Exit

Application Project

- SPM Project
- Board Support Package
- Project...
- Source Folder
- Folder
- Source File
- Header File
- File from Template
- Class Ctrl+N
- Other...

Cell	Base Addr	High Addr
ps7_intc_dist_0	0xf801000	0xf801fff
ps7_gpio_0	0xe000a000	0xe000afff
ps7_scutimer_0	0xf8006000	0xf8006fff
ps7_slcr_0	0xf8000000	0xf8000fff
ps7_scuwdt_0	0xf800620	0xf8006ff
ps7_l2cachec_0	0xf802000	0xf802fff
ps7_scuc_0	0xf800000	0xf8000fc
ps7_qspi_linear_0	0xc000000	0xc0fffff
ps7_pmu_0	0xf803000	0xf803fff
ps7_afi_1	0xf8009000	0xf8009fff
ps7_afi_0	0xf8008000	0xf8008fff
ps7_qspi_0	0xe00d000	0xe00dfff
ps7_usb_0	0xe0020000	0xe002fff
ps7_afi_3	0xf800b000	0xf800bfff
ps7_afi_2	0xf800a000	0xf800afff
ps7_globaltimer_0	0xf800200	0xf8002ff
ps7_dma_s	0xf8003000	0xf8003fff
ps7_iop_bus_config_0	0xe0200000	0xe0200fff

New Project

Application Project

Create a managed make application project.

Project name:

Use default location

Location:

Choose file system:

OS Platform:

Target Hardware

Hardware Platform:

Processor:

Target Software

Language: C C++

Compiler:

Hypervisor Guest:

Board Support Package: Create New Use existing

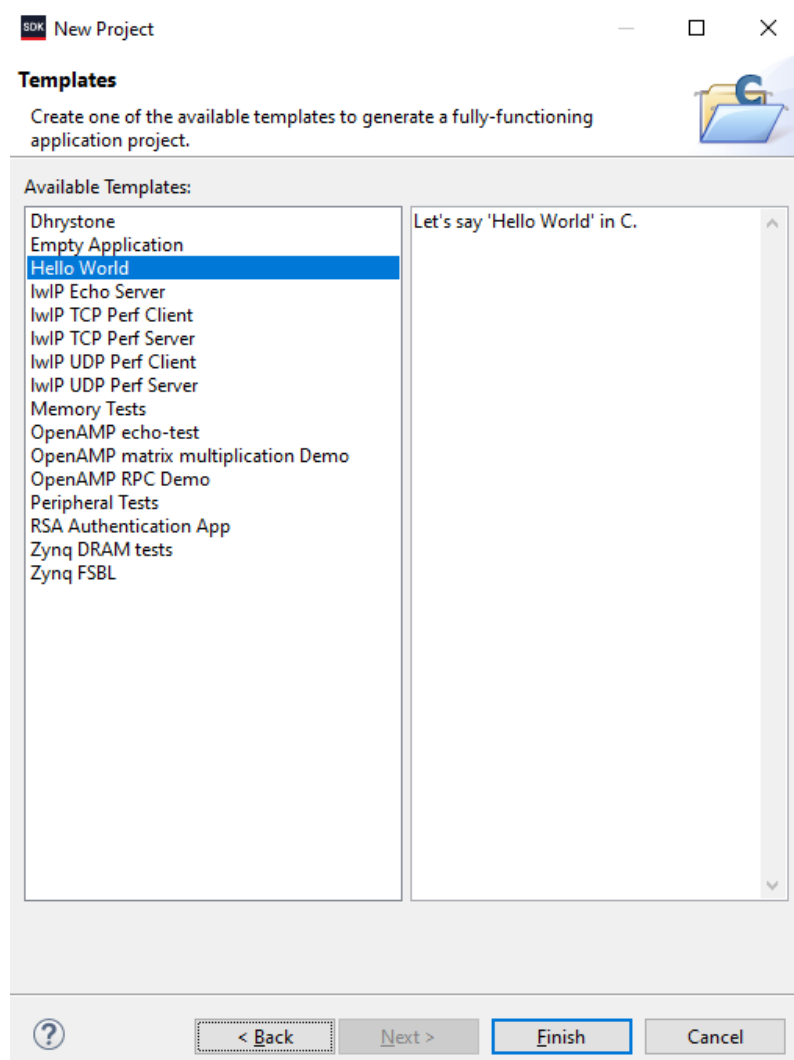


Figure 5.16: Creation of new Application

6. After that, open `HelloWorld.c` and write the code according to your IP. In your code, if you want to access your Custom IP, you should use it from `XPAR_MYIP_BASEADDR`. In your case, `MYIP_BASEADDR` will be different. In your IP, as each register carries the 32-bit size, therefore, the first register will be at offset 0. The next register will be at offset 4, the next at offset 8, and so on.
7. To provide input to the input registers of your IP, use `Xil_Out32(baseaddr + offset, value_to_store);`. To get the output from output registers of your IP, use `Xil_In32(baseaddr + offset)`. A glimpse of my code is provided

in Figure:5.17.

```
#include <stdio.h>
#include "xparameters.h" // Contains definitions for memory-mapped addresses
#include "xil_io.h"      // Contains functions for memory-mapped I/O access

// Define base address for your IP core
#define MY_IP_BASE_ADDRESS XPAR_MYIP_SHA256_0_S00_AXI_BASEADDR

// Define offsets for control and data registers
#define CS_WE_OFFSET      0x00
#define ADDRESS_OFFSET   0x04
#define WRITE_DATA_OFFSET 0x08
#define READ_DATA_OFFSET 0x0c
#define ERROR_OFFSET     0x10
#define OUTPUT_OFFSET    0x14 // Offset for the OUTPUT port
#define CONTROL_OFFSET   0x34
#define READY_OFFSET     0x38
#define INIT_OFFSET      0x3c
#define CS_CHECK_OFFSET  0x40
#define ADDRESS_CHECK_OFFSET 0x44
#define READY_DIGEST_OFFSET 0x48
#define RESET_CHECK_OFFSET 0x4c

// Function to write data to the specified address
void write_to_address(u32 offset, u32 data) {
    Xil_Out32(MY_IP_BASE_ADDRESS + offset, data);
}

// Function to read data from the specified address
u32 read_from_address(u32 offset) {
    return Xil_In32(MY_IP_BASE_ADDRESS + offset);
}

// Function to introduce a delay in nanoseconds (approximately)
void delay_ns(u32 ns) {
    for (volatile u32 i = 0; i < ns * 1000; ++i) {
        // This loop performs a delay of approximately 1 nanosecond
        // Adjust the loop count based on your processor's clock speed
        // to achieve the desired delay
    }
}

u8 read_address(u32 offset){
```

Figure 5.17: Code Snippet

5.1.3 How to run the code

In order to run the code, you have to install Tera Term. Open the app, connect the board to your laptop, and select the serial option in the app. After that, go to Setup → Serial Port → enter the speed = 115200. Then click New Open. The Tera Term windows are shown in Figure:5.18.

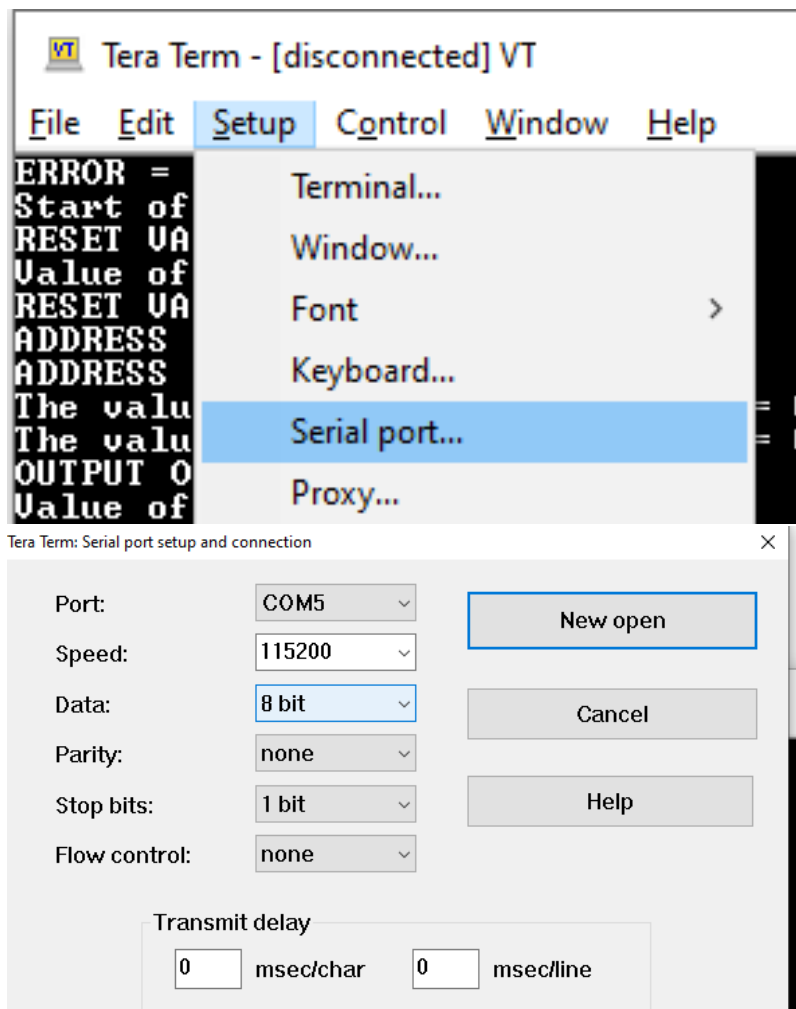
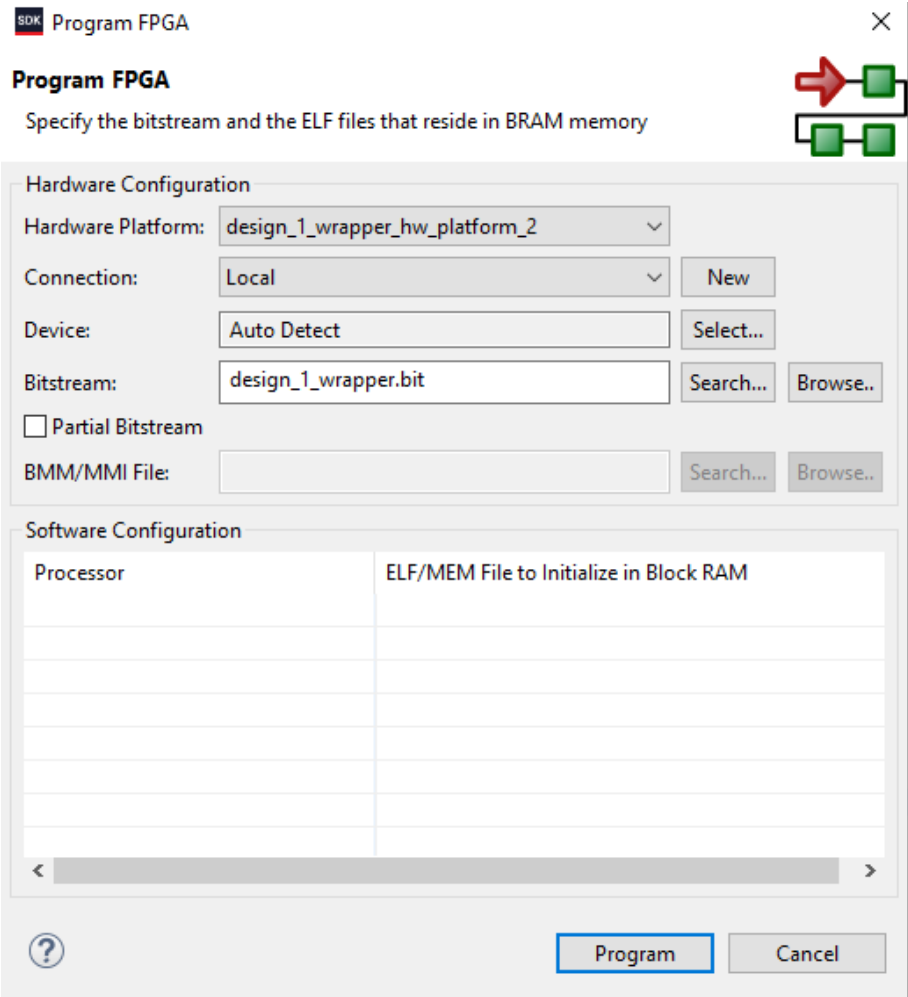


Figure 5.18: Tera Term Serial Setting

Now your board will get connected to your Tera Term terminal. Go back to your SDK app, and in the navigation bar, click on "Program FPGA". After that, right-click on your created application, select "Run As", and then choose "Launch on Hardware". Finally, your code will start running on your Tera Term terminal. The process is shown in Figure:5.19





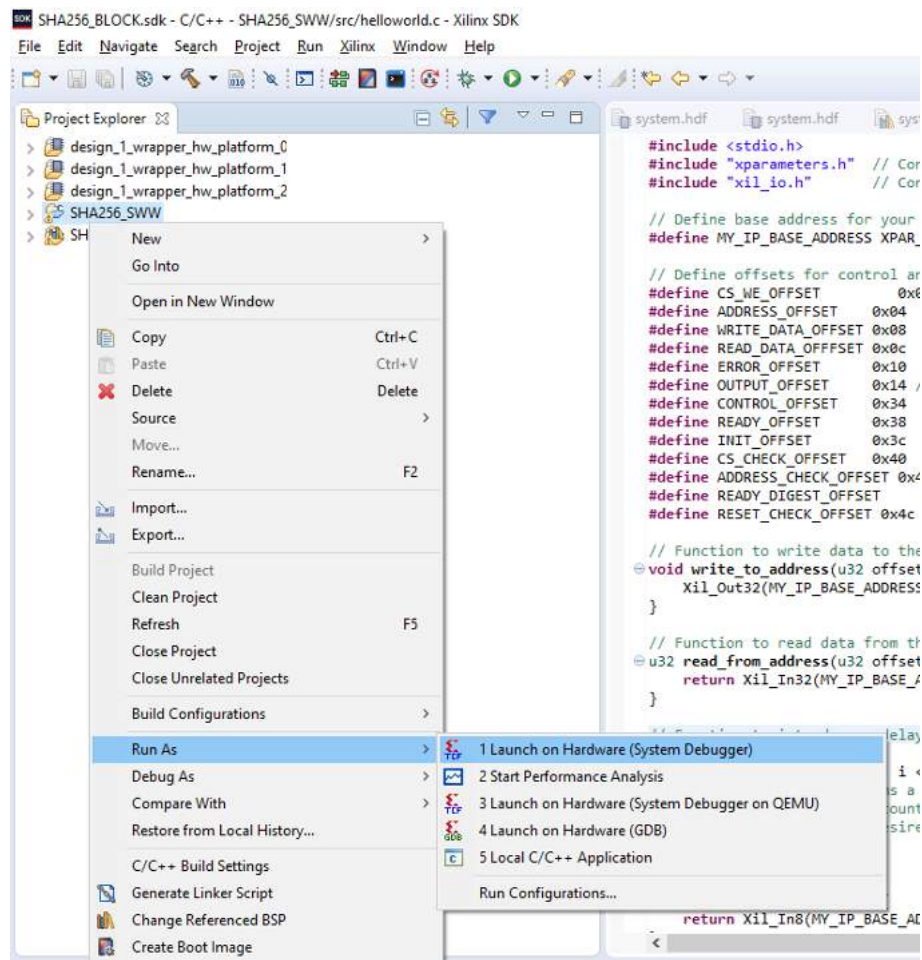


Figure 5.19: Programming ZYBO Board

5.1.4 OUTPUT:

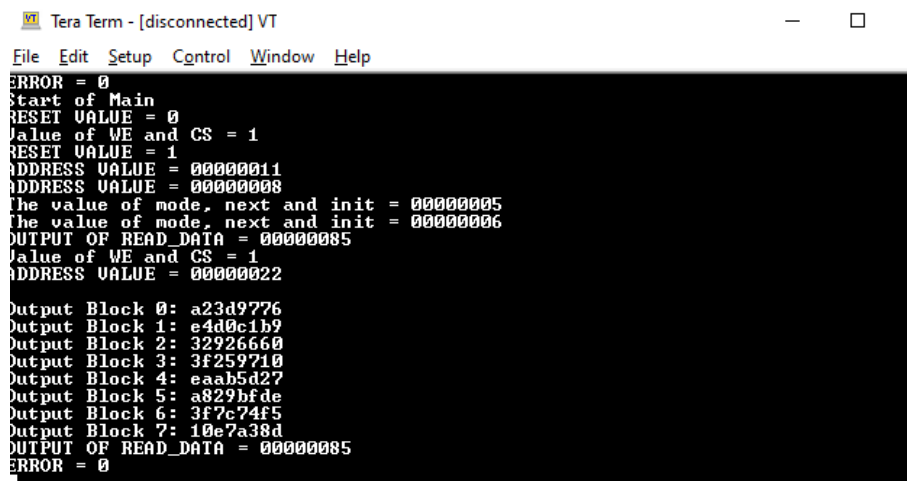


Figure 5.20: OUTPUT of SHA-256

5.2 RSA IP

Overview

[13] The RSA algorithm involves four steps: key generation, key distribution, encryption, and decryption. A basic principle behind RSA is the observation that it is practical to find three very large positive integers e , d , and n , such that for all integers m ($0 \leq m < n$), both m^e and m have the same remainder [14] when divided by n (they are congruent modulo [15] n):

$$(m^e)^d \equiv m \pmod{n}$$

However, when given only e and n , it is extremely difficult to find d . The integers n and e comprise the public key, d represents the private key, and m represents the message. The modular exponentiation [16] to e and d corresponds to encryption and decryption, respectively.

Digital Signatures

Suppose Alice uses Bob's public key to send him an encrypted message. In the message, she can claim to be Alice, but Bob has no way of verifying that the message was from Alice since anyone can use Bob's public key to send him encrypted messages. In order to verify the origin of a message, RSA can also be used to sign a message.

Suppose Alice wishes to send a signed message to Bob. She can use her own private key to do so. She produces a hash value of the message, raises it to the power of d (modulo n) (as she does when decrypting a message), and attaches it as a "signature" to the message. When Bob receives the signed message, he uses the same hash algorithm in conjunction with Alice's public key. He raises the signature to the power of e (modulo n) (as he does when encrypting a message), and compares the resulting hash value with the message's hash value. If the two agree, he knows that the author of the message was in possession

of Alice's private key and that the message has not been tampered with since being sent.

This works because of exponentiation rules: Thus the keys may be swapped without loss of generality, that is, a private key of a key pair may be used either to:

1. Decrypt a message only intended for the recipient, which may be encrypted by anyone having the public key (asymmetric encrypted transport).
2. Encrypt a message which may be decrypted by anyone, but which can only be encrypted by one person; this provides a digital signature.

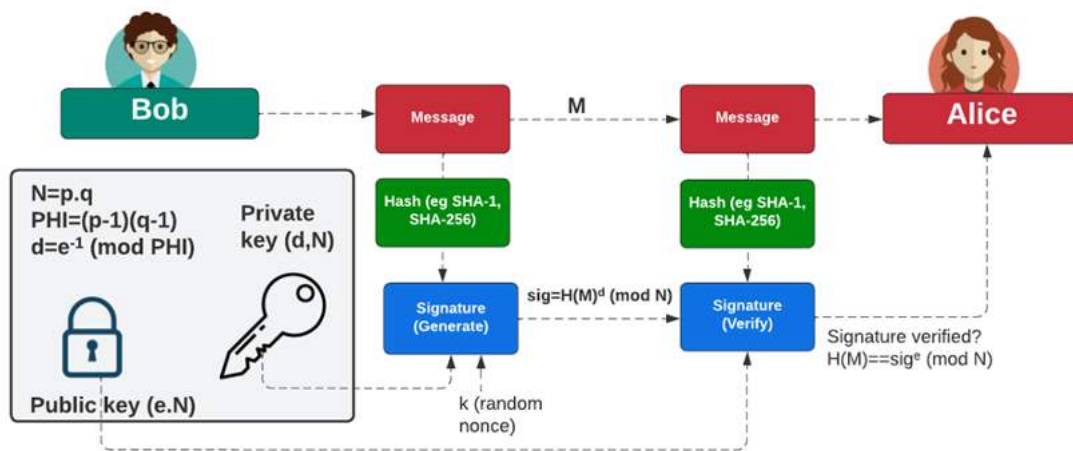


Figure 5.21: RSA Signatures Explained

RSA Encryption and Decryption Example

Step-by-Step Process

1. Choose Two Distinct Prime Numbers p and q :

$$p = 3$$

$$q = 11$$

2. Compute n :

$$n = p \times q = 3 \times 11 = 33$$

3. Compute Euler's Totient Function $\phi(n)$:

$$\phi(n) = (p - 1) \times (q - 1) = (3 - 1) \times (11 - 1) = 2 \times 10 = 20$$

4. Choose an Integer e :

$$1 < e < 20 \text{ and } \gcd(e, 20) = 1$$

Let's choose $e = 3$

5. Determine d :

$$d \times e \equiv 1 \pmod{\phi(n)}$$

This means $d \times 3 \equiv 1 \pmod{20}$. Finding the modular inverse of 3 modulo 20:

$$3d \equiv 1 \pmod{20}$$

Trying values, we find $d = 7$ because $3 \times 7 = 21 \equiv 1 \pmod{20}$

6. Public and Private Keys:

$$\text{Public key } (e, n) = (3, 33)$$

$$\text{Private key } (d, n) = (7, 33)$$

7. Encryption: Suppose we want to encrypt the message $m = 4$:

$$c = m^e \pmod n = 4^3 \pmod{33} = 64 \pmod{33} = 31$$

So, the ciphertext $c = 31$

8. Decryption: To decrypt $c = 31$ using the private key:

$$m = c^d \pmod n = 31^7 \pmod{33}$$

Computing $31^7 \pmod{33}$ step-by-step:

$$31^2 = 961 \equiv 4 \pmod{33}$$

$$31^4 = 4^2 = 16 \pmod{33}$$

$$31^6 = 31^4 \times 31^2 \equiv 16 \times 4 = 64 \equiv 31 \pmod{33}$$

$$31^7 = 31^6 \times 31 \equiv 31 \times 31 = 961 \equiv 4 \pmod{33}$$

Thus, the decrypted message $m = 4$

RSA vs. AES

Feature/Aspect	RSA (Rivest-Shamir-Adleman)	AES (Advanced Encryption Standard)
Type	Asymmetric encryption (public-key cryptography)	Symmetric encryption
Use Cases	Secure key exchange, digital signatures, encryption of small amounts of data	Encrypting large amounts of data, secure communication channels, data storage encryption
Strengths	<ul style="list-style-type: none"> - Provides secure key exchange - Supports digital signatures for authenticity and integrity 	<ul style="list-style-type: none"> - Very fast and efficient for large datasets - Strong security with 128, 192, or 256-bit keys
Weaknesses	<ul style="list-style-type: none"> - Slower and more computationally intensive - Not suitable for large data encryption directly 	<ul style="list-style-type: none"> - Key distribution challenge as both parties need the same secret key
Secure Communication	Used to securely exchange a symmetric key (e.g., AES key)	Used to encrypt the bulk of the data once the key is securely exchanged
Digital Signatures	Ideal for creating and verifying digital signatures	Not typically used for digital signatures
Data Encryption	Not efficient for encrypting large files or data streams	Preferred for encrypting large files or data streams due to speed and efficiency

Table 5.1: Comparison between RSA and AES

So for our case, RSA is preferable for secure key exchange and digital signatures. [17]

5.2.1 Introduction to AXI4-Lite (Advanced Extensible Interface)

Advanced eXtensible Interface 4 (AXI4) is a family of buses defined as part of the fourth generation of the ARM Advanced Microcontroller Bus Architecture (AMBA) standard. AXI was first introduced with the third generation of AMBA, as AXI3, in 1996.

AXI4 Protocols

The AMBA specification defines three AXI4 protocols:

- **AXI4:** A high-performance memory mapped data and address interface capable of burst access to memory mapped devices.
- **AXI4-Lite:** A subset of AXI, lacking burst access capability, and has a simpler interface than the full AXI4 interface.
- **AXI4-Stream:** A fast unidirectional protocol for transferring data from master to slave.

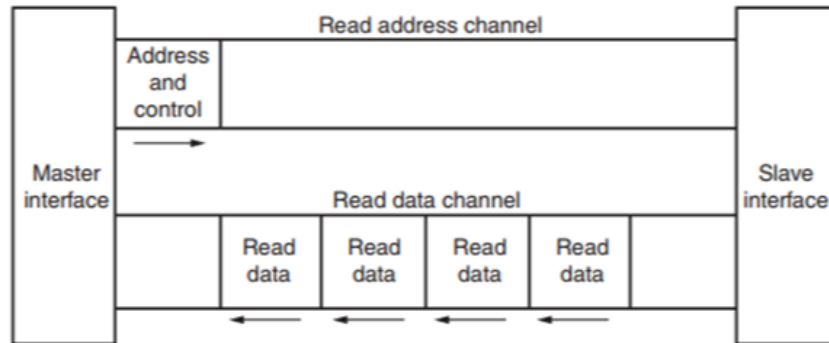
Use in Xilinx-Based Designs

Xilinx Vivado helps in the creation of custom IP with AXI4 interfaces. These can be connected to the Zynq's Processing System or to other devices. This document covers the operation of the AXI4-Lite interface, which is convenient for implementing memory mapped registers.

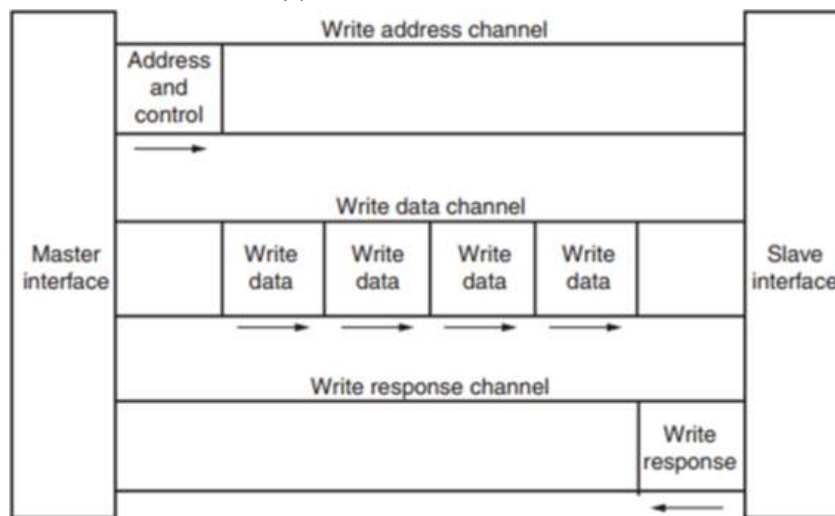
AXI4-Lite Interface Signals

The AXI4-Lite interface consists of five channels: Read Address, Read Data, Write Address, Write Data, and Write Response. An AXI4 read transaction using the Read Address and Data channels is shown in Figure 5.22a. Similarly, an AXI4 write transaction using

the Write Address, Data, and Response channels is shown in Figure 5.22b. Note that these figures depict burst transfers, which AXI4-Lite is incapable of.



(a) AXI4 Read Transaction



(b) AXI4 Write Transaction

Figure 5.22: AXI4 Read and Write Transactions

[18]

[19]

5.2.2 Hardware Bottlenecks

The number of LUTs on FPGA required is less. [20]

FPGA Description

A basic FPGA architecture (Figure 3.4) consists of thousands of fundamental elements called configurable logic blocks (CLBs) surrounded by a system of programmable interconnects, called a fabric, that routes signals between CLBs. Input/output (I/O) blocks interface between the FPGA and external devices. An individual CLB (Figure 5.23) is made up of several logic blocks. A lookup table (LUT) is a characteristic feature of an FPGA. An LUT stores a predefined list of logic outputs for any combination of inputs: LUTs with four to six input bits are widely used. Standard logic functions such as multiplexers (mux), full adders (FAs), and flip-flops are also common.

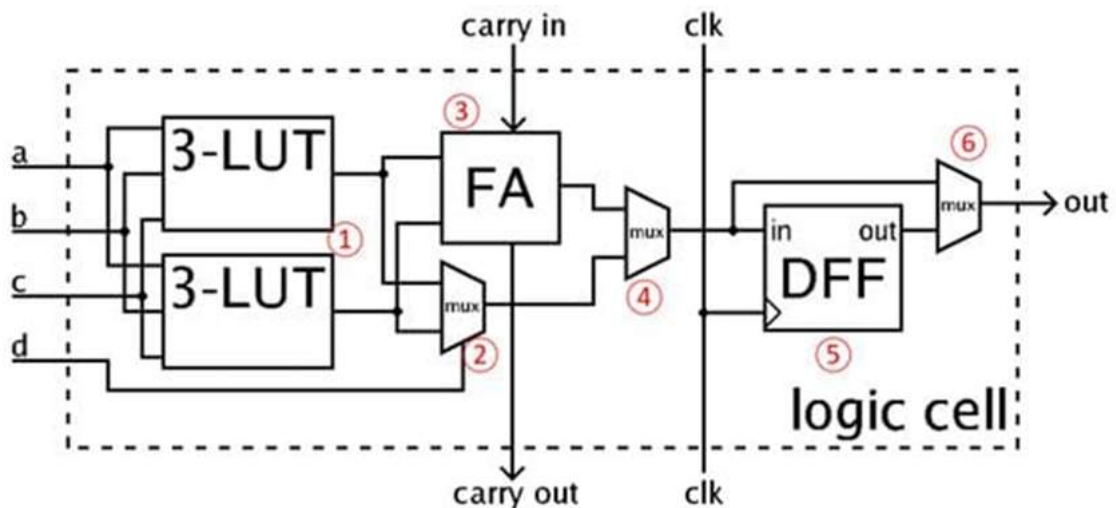


Figure 5.23: Configurable Logic Block (CLB)

[17] [21] [22] [23] [24] [25] [26] [27] [28] [29] [28] []

Error Encountered

Implementation Messages:

[Place 30-640] Place Check: This design requires more Slice LUTs cells than are available in the target device. **Explanation:** Our design requires more Look-Up Tables (LUTs) than what is available on the target FPGA. Specifically, it needs 25,617 Slice LUTs, but

only 17,600 are available. Our board actually has 4,400 logic slices, each with four 6-input LUTs and 8 flip-flops.

Solutions:

1. Optimize the Design: Review and optimize your HDL code to reduce the usage of LUTs.
2. Target a Larger Device: Select a larger FPGA device that provides more LUTs.
3. Change DRC Settings: If the design is close to the resource limit and might fit with minor adjustments, you can change the DRC settings to a warning:
4. Set the Tcl parameter: `set_param drc.disableLUTOverUtilError 1`

[Place 30-99] Placer failed with error: 'Implementation Feasibility check failed'. **Explanation:** The placer could not successfully place all instances of the design due to resource constraints or other implementation issues.

Solutions:

1. Increase Resources: Again, consider optimizing the design or targeting a larger FPGA with more resources.
2. Use a multiplexer to select the operands for shared units based on control signals.
3. DSP Blocks: Utilize dedicated DSP blocks for arithmetic operations like multiplication and MAC (multiply-accumulate) instead of implementing them in LUTs.

Example: Most FPGAs have DSP slices optimized for such operations, freeing up LUTs for other uses.

We opt for the first option by simplifying combinational logic expressions. We use the smallest bit width necessary for your signals and variables, i.e., 16-bit. Smaller bit widths reduce the number of LUTs required and the third approach is used somehow inherently for multiplication.

We can go for the second option, but this adds significant complexity by breaking down complex combinational logic into multiple stages and inserting registers to create a pipeline. This can reduce the logic depth and the number of LUTs.

RSA Design Wrapper

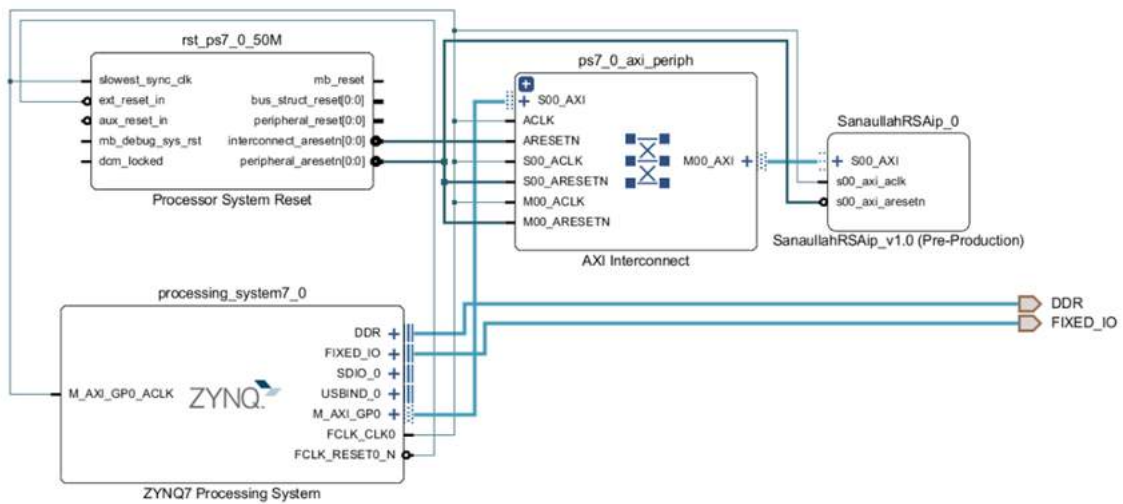


Figure 5.24: RSA Wrapper

This design wrapper in Xilinx Vivado shows the interconnections between different components in a Zynq-7000 SoC design. Let's break down the key components and their connections:

Components

1. Processor System Reset (`rst_ps7_0_50M`)
2. Processing System (`processing_system7_0`)
3. AXI Interconnect (`ps7_0_axi_periph`)
4. Custom IP Block (`SanaullahRSAip_0`)

5.2.3 Connections

Processor System Reset (`rst_ps7_0_50M`)

Inputs:

- `slowest_sync_clk`: Synchronization clock input.
- `ext_reset_in`: External reset input.
- `aux_reset_in`: Auxiliary reset input.
- `mb_debug_sys_rst`: Debug system reset.
- `dcm_locked`: Clock manager lock signal.

Outputs:

- `mb_reset`: MicroBlaze reset.
- `bus_struct_reset`: Bus structure reset.
- `peripheral_reset`: Peripheral reset.
- `interconnect_aresetn`: Interconnect reset (active low).
- `peripheral_aresetn`: Peripheral reset (active low).

These outputs are connected to various reset inputs of the processing system and AXI interconnect to ensure proper initialization and reset management.

Processing System (`processing_system7_0`)

Connections:

- `M_AXI_GP0_ACLK`: AXI General Purpose 0 clock.
- `DDR`: DDR memory interface.
- `FIXED_IO`: Fixed I/O connections for peripherals like GPIO.

- SDIO_0: Secure Digital Input Output for SD card interfaces.
- USBIND_0: USB interface.
- M_AXI_GP0: General Purpose AXI Master interface.

The processing system (PS) is the core of the Zynq-7000 SoC, integrating an ARM Cortex-A9 processor with various peripherals. The PS connects to the PL (Programmable Logic) via the AXI interconnect.

AXI Interconnect (ps7_0_axi_periph)

Inputs:

- S00_AXI: Slave AXI interface.
- ACLK: AXI clock.
- ARESETN: AXI reset (active low).

Outputs:

- M00_AXI: Master AXI interface.
- M00_ACLK: Master AXI clock.
- M00_ARESETN: Master AXI reset (active low).

The AXI interconnect module facilitates the connection between the processing system and the custom IP block. It routes AXI transactions from the PS to the custom IP and vice versa.

Custom IP Block (SanallahRSAip_0)

Connections:

- S00_AXI: Slave AXI interface.
- s00_axi_aclk: AXI clock.

- `s00_axi_aresetn`: AXI reset (active low).

This is a custom IP block named `SanallahRSAip_0`, likely implementing some cryptographic functions using the RSA algorithm. It interfaces with the AXI interconnect to communicate with the PS.

5.2.4 How They Work Together

1. **Reset and Clock Management:** The `rst_ps7_0_50M` block manages the reset signals for the entire design, ensuring all components are correctly initialized. The `slowest_sync_clk` input and various reset inputs are used to generate synchronized reset signals for different parts of the system.
2. **Processing System (PS):** The `processing_system7_0` block integrates the ARM Cortex-A9 processors and connects to external memory (DDR) and peripherals (FIXED_IO, SDIO, USB). It uses the `M_AXI_GP0` interface to communicate with the AXI interconnect in the programmable logic (PL) part of the SoC.
 - **AXI Interconnect:** The `ps7_0_axi_periph` block routes AXI transactions between the PS and the custom IP block. The `S00_AXI` interface connects to the PS, while the `M00_AXI` interface connects to the custom IP block (`SanallahRSAip_0`). It uses the `ACLK` and `ARESETN` signals for clocking and reset.
3. **Custom IP Block:** The `SanallahRSAip_0` block implements custom logic (e.g., RSA encryption/decryption) and interfaces with the AXI interconnect to communicate with the PS. It uses the `S00_AXI` interface for AXI transactions, driven by `s00_axi_aclk` and `s00_axi_aresetn` for clocking and reset.

5.2.5 Summary

This design wrapper illustrates a typical Zynq-7000 SoC setup where the processing system (ARM Cortex-A9) is connected to custom logic implemented in the programmable logic (PL) through an AXI interconnect. The `rst_ps7_0_50M` block ensures proper reset sequencing, the `processing_system7_0` block acts as the central processing unit, the `ps7_0_axi_periph` manages AXI transactions, and the `SanullahRSAip_0` block implements custom functionality interfaced via AXI.

5.3 Simulation Results

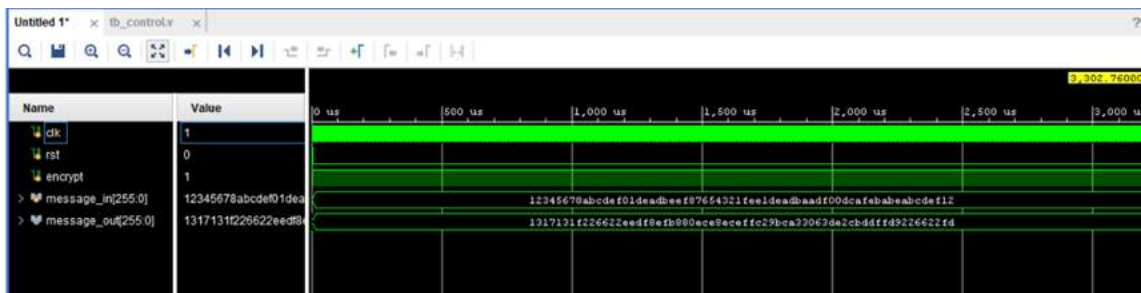


Figure 5.25: RSA Simulation

Bare Metal Results

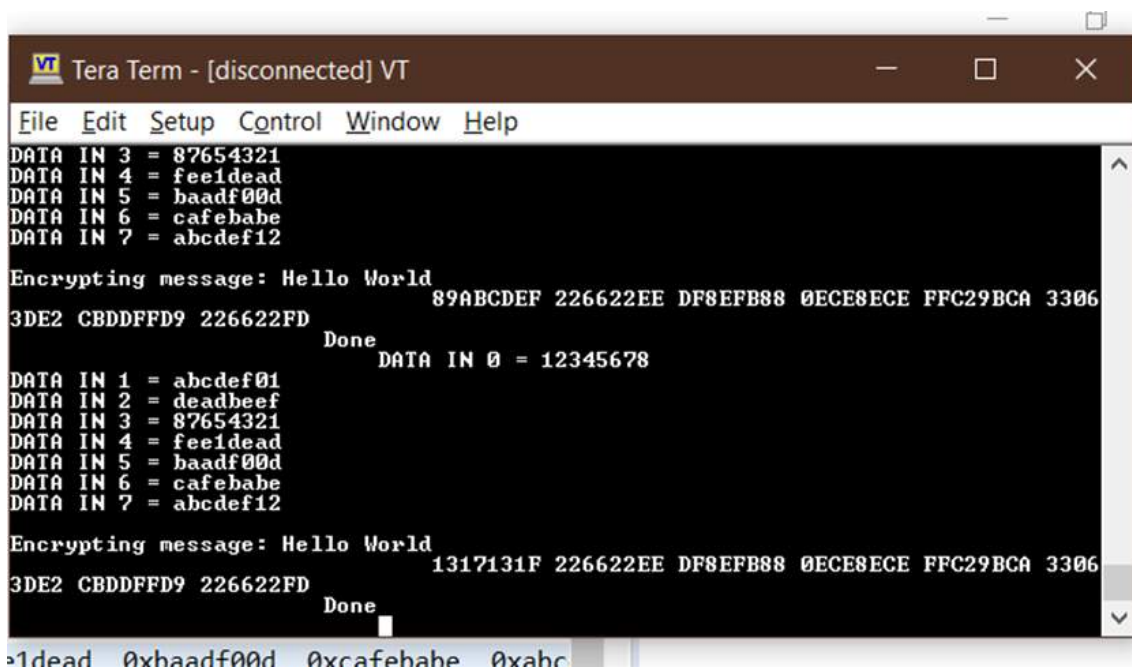


Figure 5.26: RSA Result

Input: 12345678 abcdef01 deadbeef 87654321 feelddead baadf00d
cafebabe abcdef12

Correct/desired output: 1317131f 226622ee df8efb88 0ece8ece
ffc29bca 33063de2 cbddffd9 226622fd

5.4 Integration of IPs and Received Transaction

As we have created the separate IPs for both encryption/decryption and hashing algorithms for our Project, now we want to integrate them together so that we can take the transaction received as an input for both of the IPs and then the IPs can process the algorithms and return the desired output to the Tera Term Terminal.

In this regard, first we have to modify the block diagram. Previously we had different block diagrams for both SHA256 and encryption modules. Then we will modify our C++ code that is used for receiving transactions and make it able to store the received transac-

tion to the memory address of the Zybo board. After that, we will make our SDK code fetch the transaction data saved in memory by the C++ code and use it for hashing and encryption.

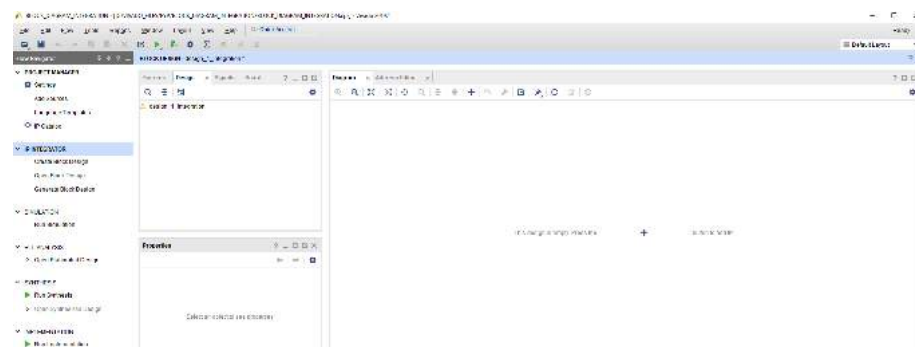
5.4.1 Steps for Integration

Modify Block Diagram

In order to modify the block diagram so that our custom IPs get connected to the transaction received using C++ code, we need to use the DMA block in our block diagram. This DMA block will provide a shared memory to both the C++ code running on Linux and the custom IPs code running on the SDK.

In this regard, the following steps are followed for the entire process:

1. Creating a new project. After the successful creation of the project, create a new block diagram. Then go to settings and include the IP repositories of both the hashing and encryption(Figure 5.27).



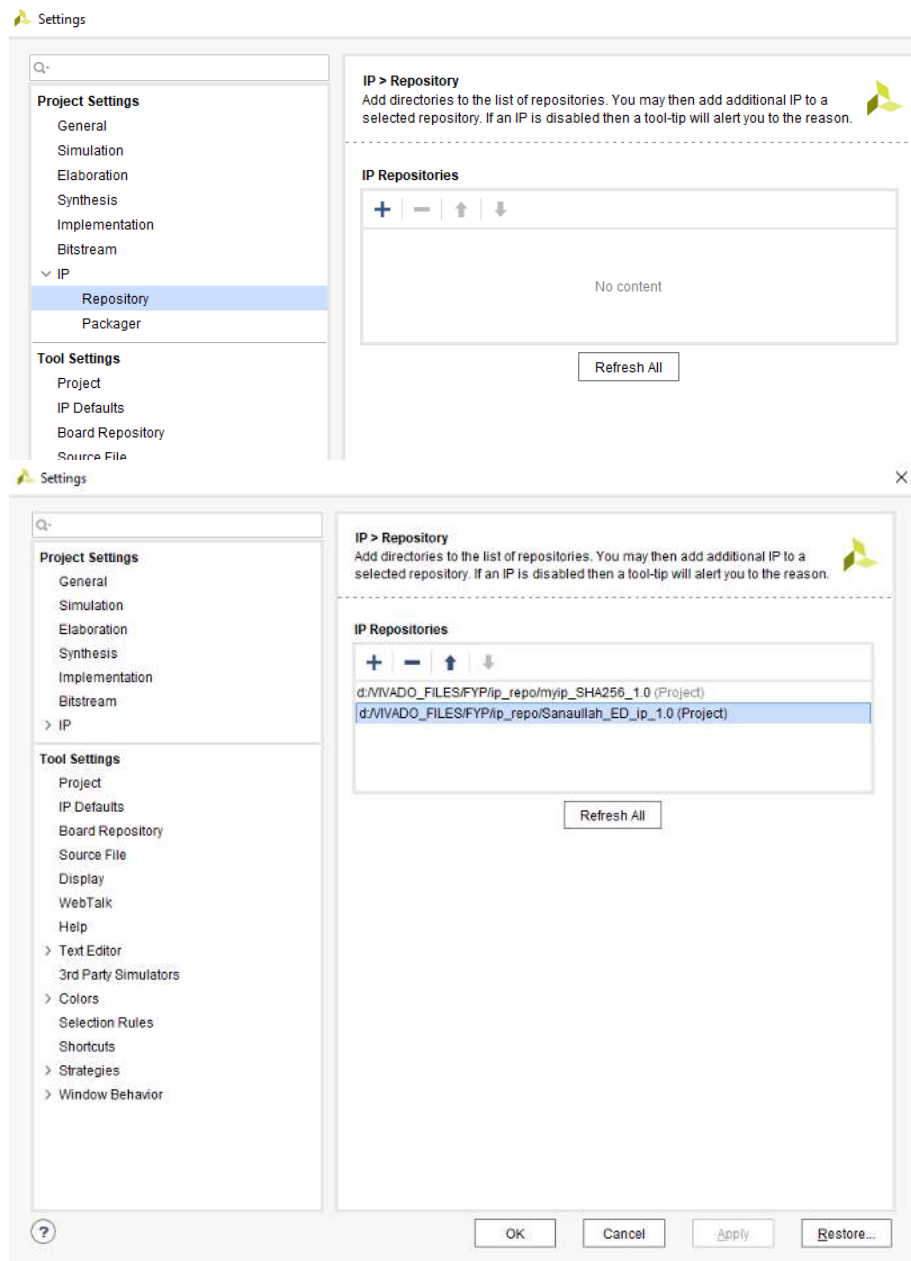


Figure 5.27: Adding Both IPs

2. After adding the IPs to the repository, we will use them to create a block diagram. For the block diagram, we will add a Zynq processor, the hashing IP, the encryption IP, and most importantly the AXI DMA block(Figure 5.28). The AXI DMA block is used for memory mapping.

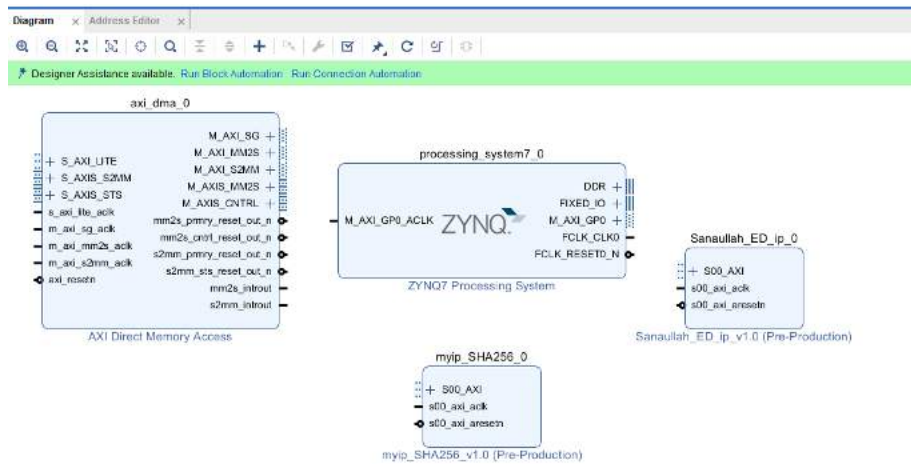


Figure 5.28: All Required Blocks Added

- Now double-click the `axi_dma` block and uncheck the "Enable Scatter Engine", as shown in Figure 5.29.

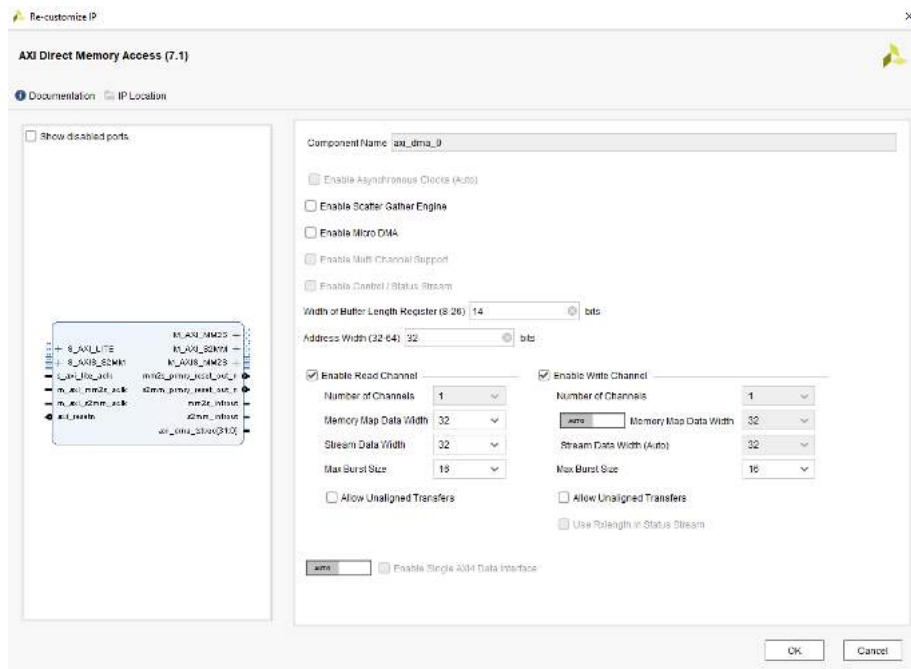


Figure 5.29: Disable Scatter Engine

- As you can see, there are two options given by Design Assistance. First, click on

"Run Block Automation" and then click on "Run Connection Automation"(Figure 5.30).

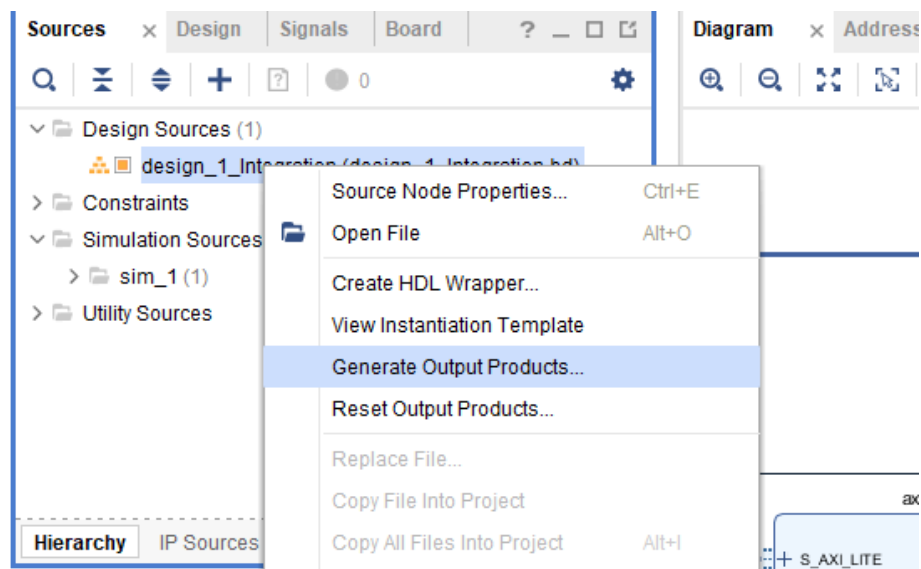
The image shows two dialog boxes from a design tool. The first, "Run Block Automation", has a left pane with "All Automation (1 out of 1 selected)" and "processing_system7_0" checked. The right pane shows a description: "This option sets the board preset on the Processing System. All current properties will be overwritten by the board preset. This action cannot be undone. Zynq7 block automation applies current board preset and generates external connections for FIXED_IO, Trigger and DDR interfaces." It also includes a "NOTE" about discarding IP configuration and an "Instance: /processing_system7_0". The "Options" section has "Make Interface External: FIXED_IO, DDR", "Apply Board Preset" checked, and "Cross Trigger In" and "Cross Trigger Out" set to "Disable".

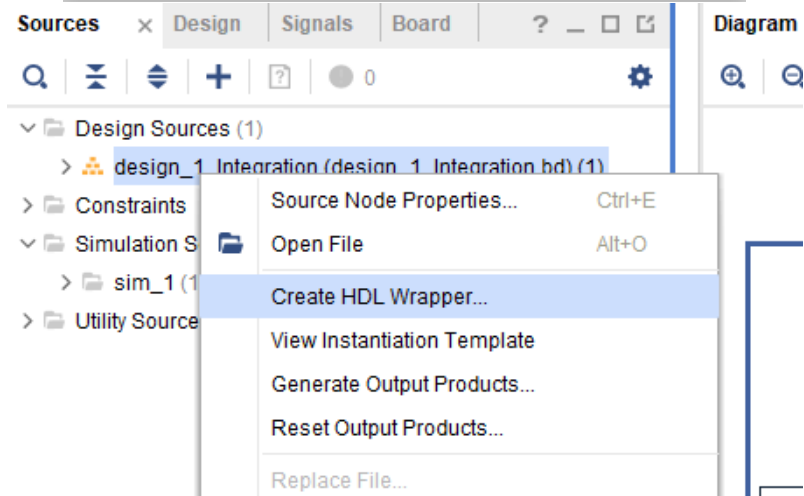
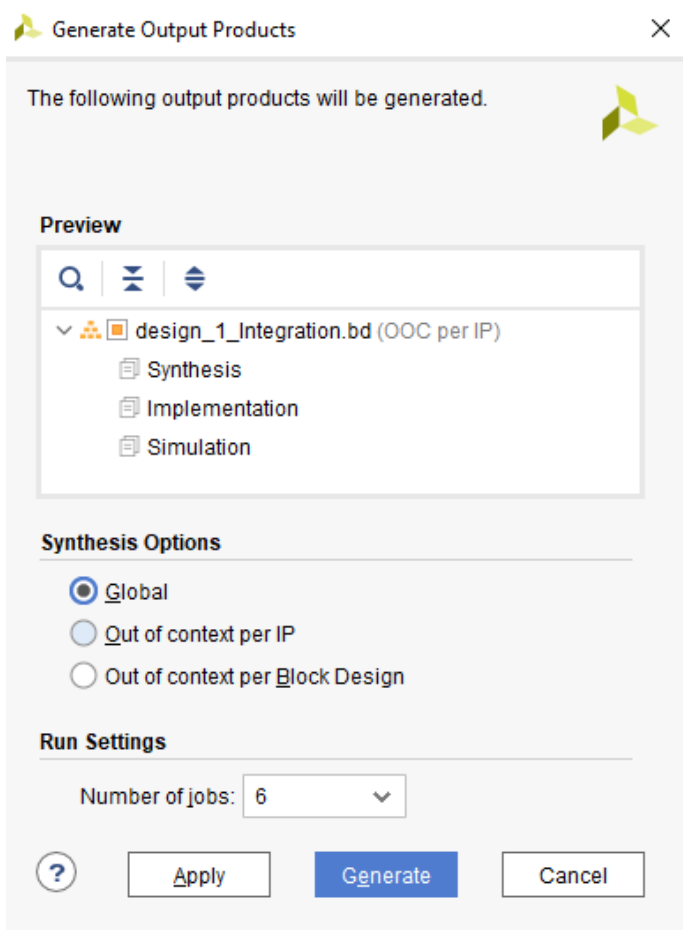
The second dialog, "Run Connection Automation", has a left pane with "All Automation (3 out of 3 selected)" and "axi_dma_0", "myip_SHA256_0", and "Sanaullah_ED_ip_0" checked. The right pane is empty with the text "Select an interface pin on the left panel to view its options".

Below the dialog boxes is a "Final Block Diagram" showing a central "ZYNQ Processing System" block. It is connected to several peripheral blocks: "axi_dma_0" (AXI Direct Memory Access), "myip_SHA256_0" (SHA256 IP), "Sanaullah_ED_ip_0" (Sanaullah ED IP), "ps7_0_axi_periph" (AXI Interconnect), and "ps7_0_axi_periph" (AXI Interconnect). The diagram shows the internal connections between these blocks and the ZYNQ core.

Figure 5.30: Final Block Diagram

5. The final block diagram is now created(Figure 5.30). First, validate the design by pressing **F6**, then go to the **Source** tab, right-click on `design_1`, and select **Generate Output Products**. Next, create a design wrapper by right-clicking on `design_1` again. Finally, run synthesis, implementation, and generate the bitstream. After that, export the hardware by including the bitstream(Figure 5.31).





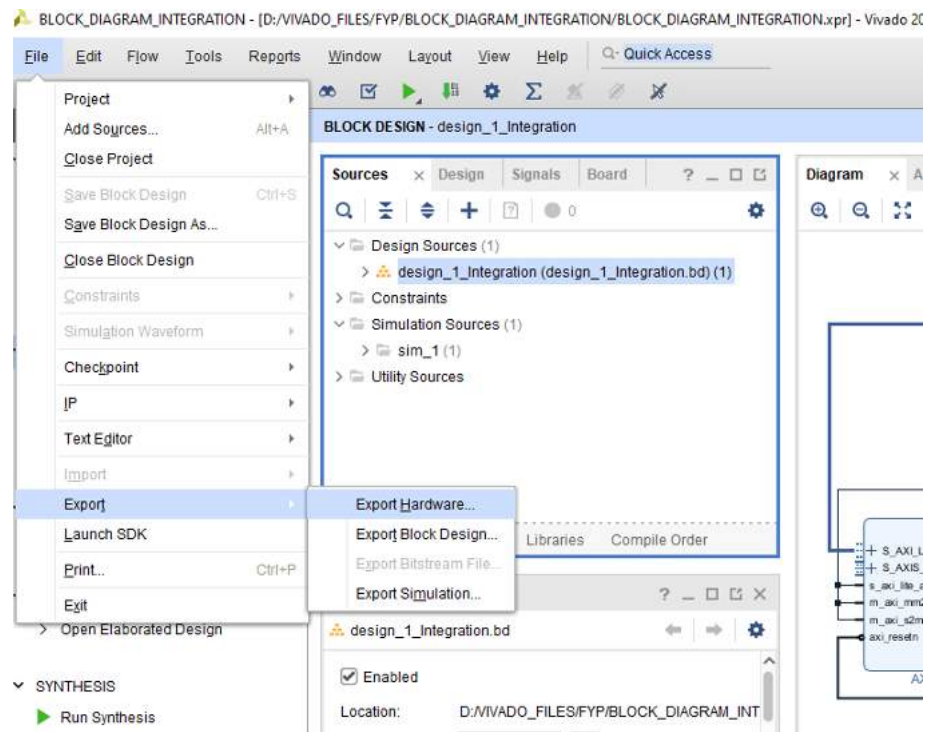


Figure 5.31: Exporting Hardware

Modify C++ Code

After creating the new block diagram and exporting the hardware, rebuild the Petalinux boot image using the same techniques as in Chapter 4. This time, we will modify the C++ code that we used for receiving the transaction. Remember that the directory where the memory addresses of the DMA are saved is `"/dev/mem"` on the SD card in which Linux is booted. We will use that directory to save the received transaction data so that the SDK can fetch the data from there to generate hash and encryption.

For this purpose, first, we will define the size of the shared memory and the memory directory. After that, open the shared memory in the code and map it to your project. This mapping will save the transaction data to a location in shared memory. The implementation is shown in Figure [5.32](#)

```

#define TRANSMITTER_IP "192.168.56.101"
#define PORT 8888
#define SHARED_MEMORY_SIZE 4096
#define SHARED_MEMORY_NAME "/json_shared_memory"

try {
    json transaction = json::parse(json_data);

    // Write to shared memory
    int fd = shm_open(SHARED_MEMORY_NAME, O_RDWR | O_CREAT, 0666);
    if (fd == -1) {
        std::cerr << "Failed to open shared memory\n";
        return;
    }

    if (ftruncate(fd, SHARED_MEMORY_SIZE) == -1) {
        std::cerr << "Failed to set shared memory size\n";
        close(fd);
        return;
    }

    void* shared_memory = mmap(NULL, SHARED_MEMORY_SIZE, PROT_READ | PROT_WRITE,
    if (shared_memory == MAP_FAILED) {
        std::cerr << "Failed to map shared memory\n";
        close(fd);
        return;
    }

    strncpy((char*)shared_memory, json_data.c_str(), SHARED_MEMORY_SIZE);

    // Clean up

```

Figure 5.32: Implementing Shared Memory

Now your C++ code is ready to save the transaction data to shared memory.

Modify SDK Code

Now that the C++ file is ready to share the transaction with the SDK, we will implement our SDK code.

1. Launch the SDK from our block diagram project so that it can automatically import

the required files for our hardware(Figure 5.33).

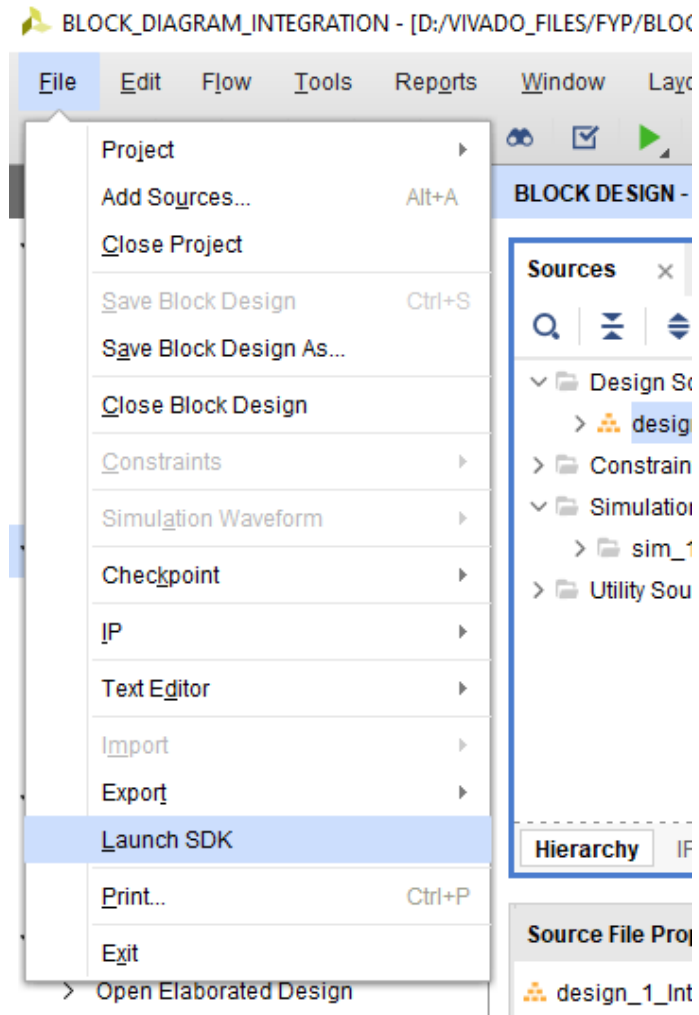


Figure 5.33: Launching SDK

2. Follow the same process as described in Chapter 5 to create a new application. After creating the new application, write our SDK code in it to use the input from the shared memory and implement hashing and encryption(Figure 5.34).
3. Define the memory addresses of AXI4-lite, shared memory base address, transmitting and receiving buffer base addresses, and maximum buffer length. Also include the `xaxidma.h` header file for dealing with DMA(Figure 5.34).

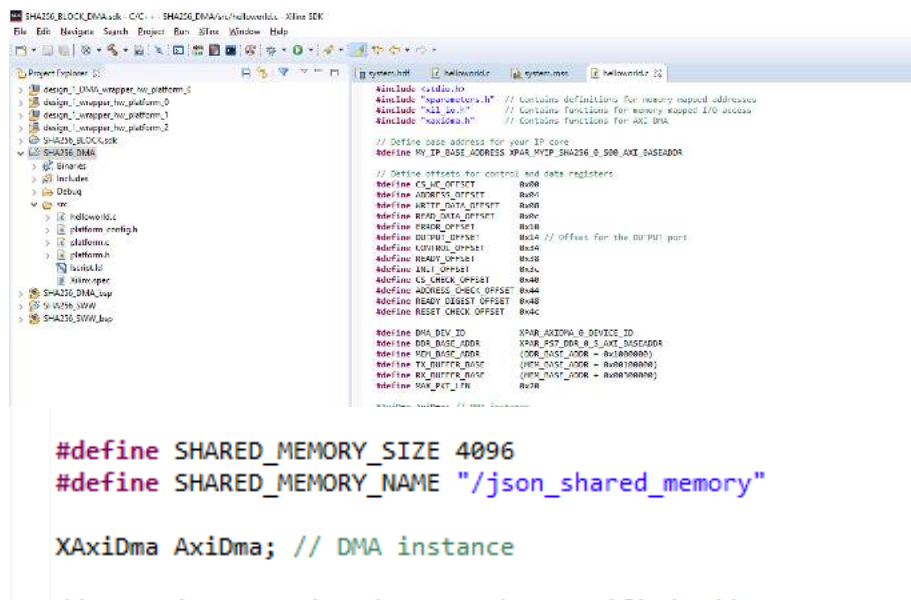


Figure 5.34: Defining Shared Memory and DMA

4. Create a DMA instance and a function that will initialize the DMA using XAxiDma_Config(Figure 5.35). After the successful initialization of DMA, create a function to transfer the data to and from using AXI(Figure 5.36). In the main function, create two buffers, one for transmitting and one for receiving(Figure 5.37). Open the shared memory and map it to your project(Figure 5.38). Parse the JSON data received using the map, and prepare the data to send to the hash and encryption function(Figure 5.39) and Writing data is shown in Figure 5.40. The remaining code will be similar to what we made in Chapter 5 for both hashing and encryption.


```

// Function to initialize the DMA
int init_dma() {
    XAxiDma_Config *CfgPtr;
    int Status;

    CfgPtr = XAxiDma_LookupConfig(DMA_DEV_ID);
    if (!CfgPtr) {
        printf("No configuration found for DMA\n");
        return XST_FAILURE;
    }

    Status = XAxiDma_CfgInitialize(&AxiDma, CfgPtr);
    if (Status != XST_SUCCESS) {
        printf("DMA Initialization failed\n");
        return XST_FAILURE;
    }

    if (XAxiDma_HasSg(&AxiDma)) {
        printf("DMA configured in SG mode\n");
        return XST_FAILURE;
    }

    return XST_SUCCESS;
}

```

Figure 5.35: Initializing DMA

```

// Function to transfer data using DMA
int dma_transfer(u32 *tx_buffer, u32 *rx_buffer, int length) {
    int Status;

    // Flush the buffers before the DMA transfer
    Xil_DCacheFlushRange((UINTPTR)tx_buffer, length * sizeof(u32));
    Xil_DCacheFlushRange((UINTPTR)rx_buffer, length * sizeof(u32));

    // Start the DMA transfer
    Status = XAxiDma_SimpleTransfer(&AxiDma, (UINTPTR)tx_buffer, length * sizeof(u32), XAXIDMA_DMA_TO_DEVICE);
    if (Status != XST_SUCCESS) {
        printf("DMA to Device transfer failed\n");
        return XST_FAILURE;
    }

    Status = XAxiDma_SimpleTransfer(&AxiDma, (UINTPTR)rx_buffer, length * sizeof(u32), XAXIDMA_DEVICE_TO_DMA);
    if (Status != XST_SUCCESS) {
        printf("DMA from Device transfer failed\n");
        return XST_FAILURE;
    }

    // Wait for the transfer to complete
    while (XAxiDma_Busy(&AxiDma, XAXIDMA_DMA_TO_DEVICE));
    while (XAxiDma_Busy(&AxiDma, XAXIDMA_DEVICE_TO_DMA));

    // Invalidate the cache to ensure data coherence
    Xil_DCacheInvalidateRange((UINTPTR)rx_buffer, length * sizeof(u32));

    return XST_SUCCESS;
}

```

Figure 5.36: Defining Transfer Function for DMA

```

int main() {
    int Status;
    u32 tx_buffer[MAX_PKT_LEN];
    u32 rx_buffer[MAX_PKT_LEN];
    int i;

    // Initialize the DMA
    Status = init_dma();
    if (Status != XST_SUCCESS) {
        return -1;
    }
}

```

Figure 5.37: Initializing Buffer Variables

```

// Open shared memory
int fd = shm_open(SHARED_MEMORY_NAME, O_RDONLY, 0666);
if (fd == -1) {
    printf("Failed to open shared memory\n");
    return -1;
}

void* shared_memory = mmap(NULL, SHARED_MEMORY_SIZE, PROT_READ, MAP_SHARED, fd, 0);
if (shared_memory == MAP_FAILED) {
    printf("Failed to map shared memory\n");
    close(fd);
    return -1;
}

```

Figure 5.38: Opening Shared Memory

```

// Read data from shared memory
char json_data[SHARED_MEMORY_SIZE];
strncpy(json_data, (char*)shared_memory, SHARED_MEMORY_SIZE);
printf("Received JSON Data: %s\n", json_data);

// Parse the JSON data (Assuming the data contains an array of integers)
// In a real scenario, you would adjust this based on the actual structure of your JSON data
json parsed_data = json::parse(json_data);
std::vector<u32> data = parsed_data.get<std::vector<u32>>();

// Clean up shared memory
munmap(shared_memory, SHARED_MEMORY_SIZE);
close(fd);

// Prepare data to send
for (i = 0; i < data.size() && i < MAX_PKT_LEN; ++i) {
    tx_buffer[i] = data[i];
}

```

Figure 5.39: Preparing Data for Input

```

// Assuming tx_buffer contains the data to write
write_to_address_u8(ADDRESS_OFFSET, 0x11); // Setting address = 8'h11
write_to_address(WRITE_DATA_OFFSET, tx_buffer[0]); // Assuming data[0] contains the data to write

```

Figure 5.40: Writing Data to Write Register

After all these steps, our IPs are integrated with the incoming transaction and are now ready to take the transactions as input for their modules.

5.5 Project Presentation on Custom IP Core and Driver Abstraction

5.5.1 Introduction

- **Objective:** To provide an abstraction layer for custom IP cores through drivers.
- **Purpose:** Simplify the usage of IP cores by abstracting register details and providing basic access functions.

5.5.2 Custom IP Core Implementation

- IP cores are implemented using registers inside the IP core.
- The driver provides basic access to the IP core for users.

5.5.3 Driver Functionality

- **Initialization:** A function to initialize the IP core.
- **Read Operation:** A function to read from the IP core.
- **Write Operation:** A function to write to the IP core.
- **Additional Functions:** Depending on the IP core type, there may be additional functions, e.g., for image processing.

5.5.4 Purpose of Abstraction

- **Ease of Use:** To simplify the programmer's task by providing an easy-to-use interface.
- **Hiding Complexity:** Abstracting away the details of registers, addresses, and user manuals.

5.5.5 Driver Components

- **Header File (.h):** Contains the hardware model and declarations of all functions.
- **Source File (.c):** Contains the definitions of all functions.

5.5.6 Driver Development Steps

1. **Create Header File:** Define the hardware structure and function declarations.
2. **Create Source File:** Implement the function definitions.

5.5.7 Application Developer Perspective

- **Simplified API:** The developer only needs to know the functions for reading and writing.
- **Abstraction:** All register details and base addresses are handled by the driver.

5.5.8 Integration Steps

1. **Delete Previous Driver Files:** Remove old .c and .h files from the project.
2. **Update Driver Files:** Add updated driver files to the IP repository.
3. **Update XML File:** Modify the .xml file to include the new driver files.
4. **Relaunch IP:** Reload the IP core in the design environment.

5.5.9 Example Workflow

1. **Wrapper:** Create the hardware wrapper.
2. **Launch SDK:** Open the Xilinx SDK.
3. **Standalone OS:** Choose a standalone OS for the project.

4. **Create Blank C Project:** Start with a blank C project.
5. **Develop Driver:** Write the header and source files for the driver.
6. **Testing:** Create a test file (e.g., test.c) to call driver functions without needing base addresses.

5.5.10 Conclusion

- **Simplification:** The abstraction provided by the driver makes the IP core easier to use.
- **Modularity:** Drivers encapsulate hardware details, promoting code reuse and maintainability.

Chapter 6

Implementation of Hyperledger Fabric in CBDC Transaction Processing

6.1 Introduction

This chapter is dedicated to exploring the aspect of implementation of Hyperledger Fabric for CBDCs in regard to the actual transactions' processing. Since Hyperledger Fabric is an open-source, permissioned distributed ledger technology that is designed specifically for enterprise applications, it can be recruited to build CBDC transaction processors that are scalable and secure enough to meet the users' expectation. The implications that follow include establishing the Hyperledger Fabric blockchain setup, loading the smart contracts, and real-time interfacing of this blockchain setup with the FPGA-based transaction processing architecture.

The setup of Hyperledger Fabric network is formed of nodes, identities, communication channels and access control or policies. This network is the total platform for the execution of the basic transaction for the CBDCs, besides facilitating their data integrity and consensus among the related companies and associations. Smart contracts are self-executing digital contracts that govern CBDC transactions and are deployed in the Hyper-

ledger Fabric network. These contracts incorporate the business relationships, compliance checks and balances and the processing of the CBDC relevant transactions as well as the record of such transactions hence enhancing the efficiency, automation, security, and verifiability of the transactions in the CBDC environment.

In this case, standing on the opportunity offered by Hyperledger Fabric's modularity, the network functions in unison with an FPGA-based transaction processing system. FPGAs provide much higher numerical computing precision and efficiency as opposed to the general purpose processors, therefore being very suitable for high traffic CBDC transactions. Interconnection of the two systems and secure means of communication and APIs facilitate the immediate analysis and verification of transactions.

During the implementation of the modern model, it is critical to respect the best practices of information protection and adhere to the requirements set out by legislation. The security of Hyperledger Fabric that stems from features like cryptographic mechanisms and access control is precisely tailored to protect the CBDC transaction processing system against various types of dangers and unauthorized persons.

Before the system can be distributed and used in live environment some tests are conducted among which are unit tests, integration tests, performance tests, and stress tests.

This implementation process lays down the foundation for secure and swift CBDC transaction processing using a convergence of distributed ledger technology, smart contracts, and High-Performance Computing that can make way for the digital currencies to be transacted in a secure, swift and reliable manner.

6.2 Setting up Hyperledger Fabric Environment

The first procedure for entering the process of using Hyperledger Fabric for CBDC transaction processing is creating a development environment. This entails downloading and installing dependent application like docker and Go language. To enhance development

using WSL with Visual Studio Code specifically for Hyperledger Fabric, the environment should be set up in this way. WSL makes the essential Linux kernel interface that enables direct application of a Linux distribution in Windows.

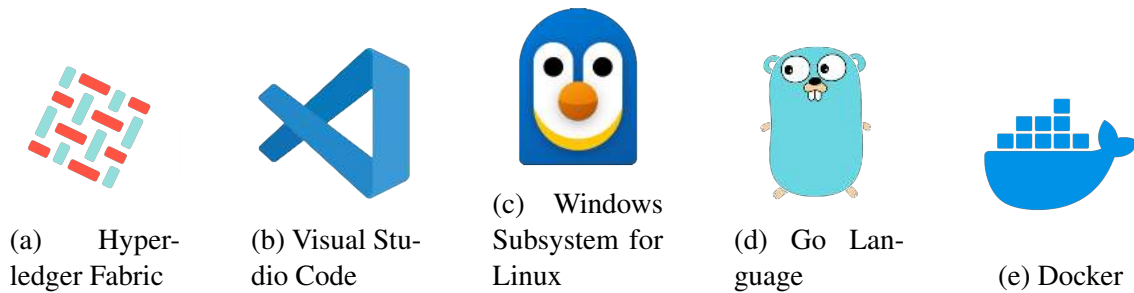


Figure 6.1: Various Tools and Technologies

6.3 Installation of Hyperledger Fabric and Fabric Samples

The setup of the development environment is followed by the installation of Hyperledger Fabric along with fabric-samples. Fabric-samples are examples of application and network usage to make the process of development and testing easier.

6.3.1 Cloning fabric-samples Repository

[30] To clone the fabric-samples repository, open your terminal and navigate to the directory where you want to store the repository. For example, if you are using WSL, navigate to your desired directory in the WSL terminal.

```
cd /mnt/c/Users/your_username/Documents/
```

Then, clone the fabric-samples repository using the following command:

```
git clone https://github.com/hyperledger/fabric-samples.git
```

This will download the fabric-samples repository into your current directory.

6.3.2 Running the Installation Script

Navigate to the `fabric-samples` directory that you just cloned:

```
cd fabric-samples/
```

In this directory, you'll find the installation script `install-fabric.sh`. This script automates the process of installing Hyperledger Fabric, Fabric binaries, and Docker images.

Make sure the script is executable by running:

```
chmod +x install-fabric.sh
```

Then, execute the script with the desired options. For example, to install Docker images and Fabric binaries, run:

```
./install-fabric.sh docker binary
```

This will download the latest versions of Docker images and Fabric binaries onto your system.

6.3.3 Confirming Installation

After the script completes execution, confirm that Hyperledger Fabric and `fabric-samples` are installed by checking the directories and versions.

Navigate to the `bin/` directory within `fabric-samples`:

```
cd bin/
```

Here, you should find the binaries such as `configtxgen`, `peer`, and `orderer`.

Additionally, ensure that the Docker images are downloaded by running:

```
docker images
```

You should see the Hyperledger Fabric Docker images listed.

With Hyperledger Fabric and fabric-samples successfully installed, you are now ready to set up a local Hyperledger Fabric network for development purposes.

6.4 Explanation of Smart Contract

Chaincodes in Hyperledger fabric define the request and response format for the CBDC transactions and are also used to code the smart contracts for CBDC. In this section, there is a brief description about the functions which have to be defined in the smart contract given below.

6.4.1 InitLedger

The `InitLedger` function initializes the ledger by adding a base set of assets. It is typically invoked when the smart contract is instantiated. In this function, assets with predefined IDs, owners, and amounts are added to the ledger.

6.4.2 CreateAsset

The `CreateAsset` function issues a new asset to the world state with the given details. It takes parameters such as ID, owner, and amount, and creates a new asset entry in the ledger if an asset with the same ID does not already exist.

6.4.3 ReadAsset

The `ReadAsset` function extracts the information of an asset that exists in the world state and belongs to a given ID. It accepts the ID of the asset as the parameter and brings out the details of the said asset; the details include Asset ID, owner, and amount.

6.4.4 UpdateAsset

The `UpdateAsset` procedure updates an existing asset with the required parameters in the world state with the given information. It receives ID, owner, and amount and records the given parameters in the particular asset's ledger in the ledger database.

6.4.5 DeleteAsset

The `DeleteAsset` function removes a specified asset from the world state according to its ID. This function receives the ID of the asset to be deleted and deletes the particular asset from the ledger.

6.4.6 AssetExists

The `AssetExists` function checks whether an asset with the given ID exists in the world state. It takes the ID of the asset as input and returns a boolean value indicating whether the asset exists in the ledger.

6.4.7 GetAllAssets

The `GetAllAssets` function retrieves all assets stored in the world state. It returns a list of all assets present in the ledger, including their IDs, owners, and amounts.

6.4.8 TransferAsset

function pinpoints whether there is any asset in the world state with the given ID or not. It expects the ID of an asset and yields a boolean value as to whether the information linked to it is present in the ledger data.

The `TransferAsset` updates the owner of the asset in the world state and triggers a notification to an external system about a change of the owner. It receives parameters like

ID and new owner and then it moves the ownership of the asset to the new owner, and also sends a notification to another system. The Flow diagram is shown in the Fig [6.2](#)

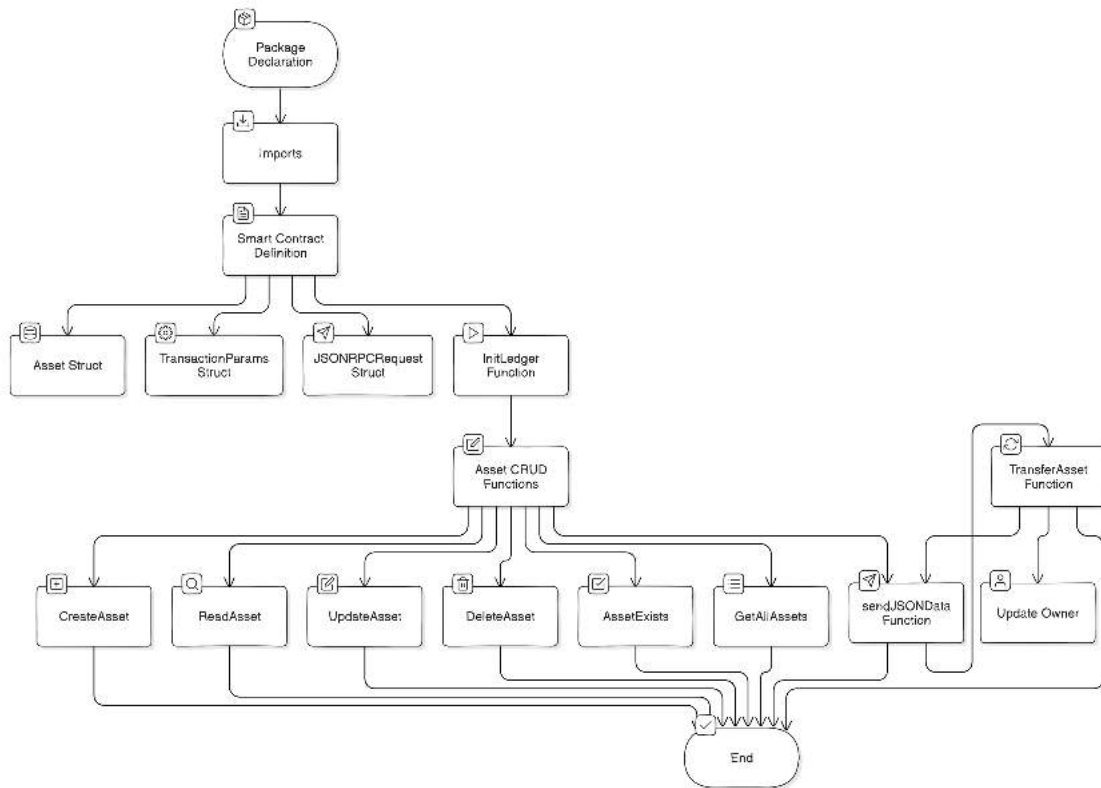


Figure 6.2: Flowchart of smart Contract

6.5 Deploying Smart Contracts to Hyperledger Fabric Network

Once these smart contracts have been developed, they are compiled and placed on the Hyperledger Fabric network. This involves deploying the chaincode on the peers and also on the endorsing organizations so as to approve transactions as per the laid down rules. The endorsement policy also means that the transaction is checked by the required number of participants before the transaction becomes permanent and is written to the ledger, which will increase security and reduce the occurrence of fraudulent transactions.

6.5.1 Chaincode Deployment Process

The deployment of smart contracts, referred to as chaincode in Hyperledger Fabric, follows a specific process:

1. **Packaging the Chaincode:** This is bundled into a deployment package which comprises the chaincode source code, any dependencies and metadata.
2. **Installing the Chaincode:** The deployment package is introduced to the peer nodes of the endorsing organizations. This step deploys the chaincode package to the filesystem of the peers so that it can be run.
3. **Approving the Chaincode:** In the second phase, the endorsing organizations of the channel review and sign off the chaincode for launching on the channel. This approval process ensures that all the organizations are informed of the chain code version and parameter that is to be deployed.
4. **Committing the Chaincode:** In this process, the required organizations run a set of transactions to approve the chaincode, after which the chaincode is committed to a channel. The commit process writes the chaincode in all the peers on the channel in order to execute the requested transactions and keep the ledger status shown in the Flow daigram [6.3](#).

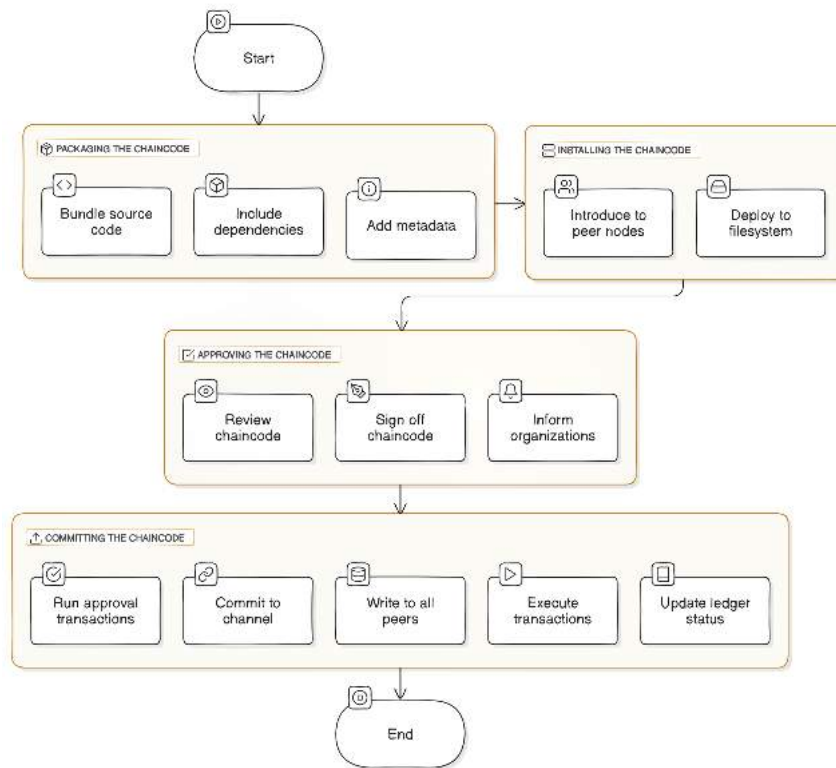


Figure 6.3: Smart Contract Deployment

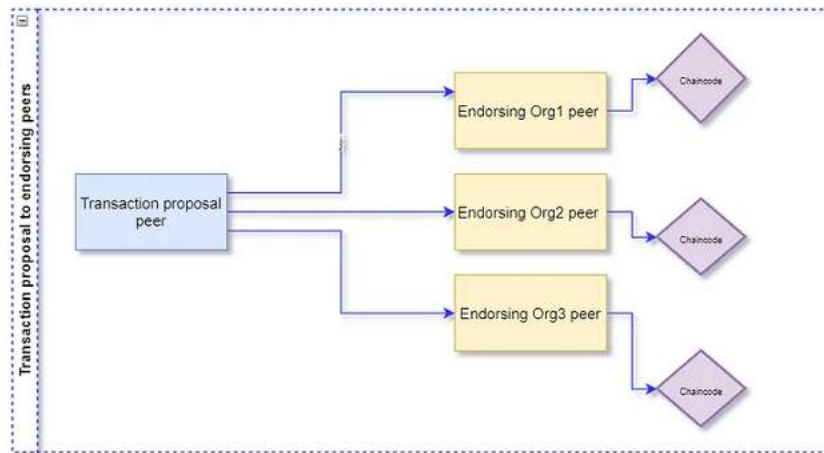
6.5.2 Endorsement Policy

[31] The endorsement policy that is applied in Hyperledger Fabric network is one of the most significant policies since it determines the organizations that need to endorse a particular transaction so that it can be written to the ledger. This one is set at the time of chaincode definition and helps to guarantee that the transactions are endorsed by the right participants making the system more secure.

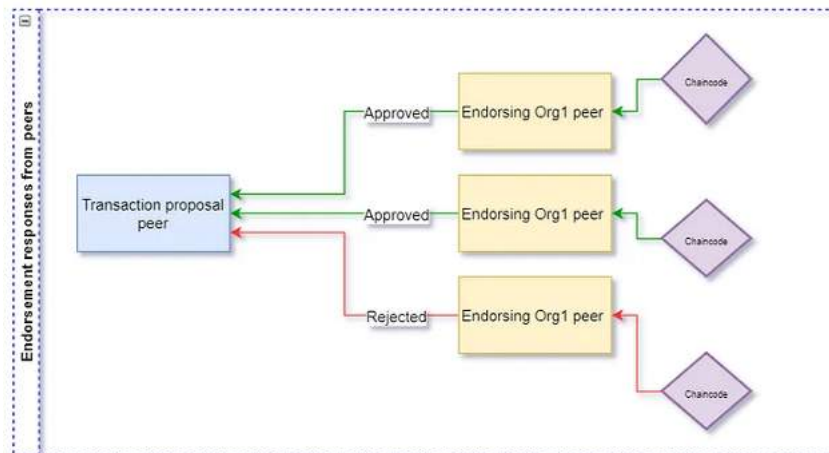
Endorsement can be controlled and managed depending on a number of factors including the number of organizations, role of the organizations, and the extent of trust among the organizations. For instance, in CBDC transaction processing system, the endorsement may be that a transaction must be first validated by the central bank and a selected set of the commercial bank before it can be committed.

In this way, Hyperledger Fabric applies the principles of the endorsement policy to guar-

antee that recordation transactions be done only by the organizations that should authorize it and thus do not include potential malicious transactions in the ledger. This mechanism enhances the credibility of the participating organizations in order to ensure secure transaction of CBDCs.



(a) Transaction Proposal



(b) Transaction Approval

Figure 6.4: Transaction Endorsement Process

6.6 Integration with FPGA-Based Transaction Processor

After adjustment of the Hyperledger Fabric network and smart contracts implementation the next action is the integration with FPGA-based TP. This integration ensures very efficient and fast processing of transactions in the ecosystem of CBDC.

6.6.1 Data Transmission

Activities that occur in the CBDC system create the right transactional data in JSON to represent the situation. For safe and correcting data transfer through the internet Necessary, Transaction Signature Protocol (TSP) is used. For record authenticity, integrity as well as non-repudiation TSP incorporates digital signatures to transaction data as in Fig

6.5. The digitally signed data is transmitted securely over the HTTPS.

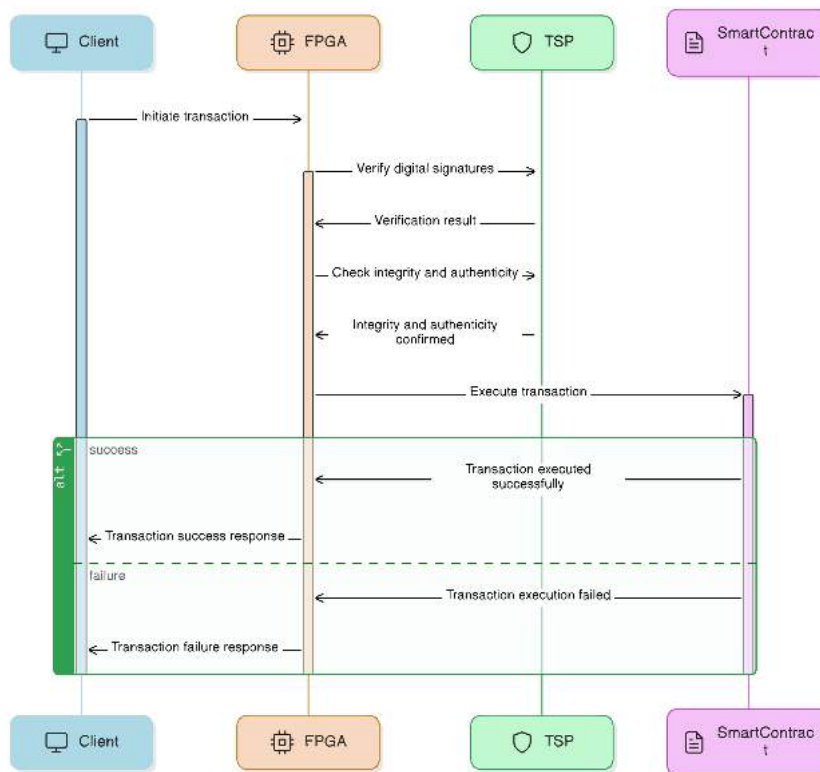


Figure 6.5: Data Transmission

6.6.2 Transmission Protocol

The integration employs the usage of HTTPS (Hypertext Transfer Protocol Secure) for shipment of transaction data on the internet. SSL or TLS involved in transmitting data across the HTTP ensures the data is encrypted to enhance the aspect of confidentiality and data integrity as shown in Fig 6.6. This means that the protocol ensures that Hyperledger Fabric network and the FPGA based transaction processor are secure.

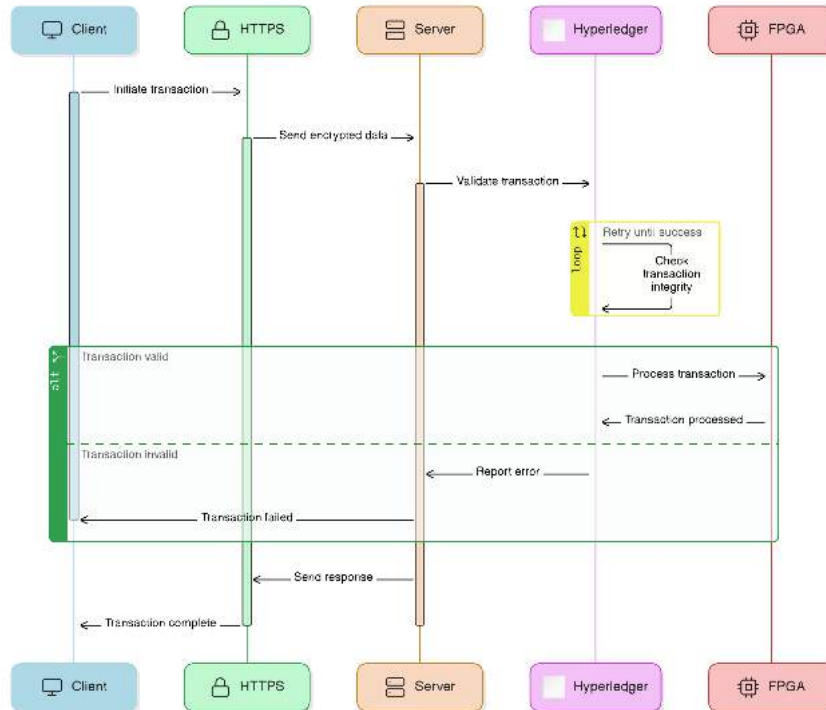


Figure 6.6: Transmission Protocol

6.6.3 Data Processing

When a transaction occurs in the Hyperledger Fabric network, the FPGA-based transaction processor compares the digital signatures with the help of a public key. TSP also cannot be altered and therefore acts as a means of checking the integrity and authenticity of the received data. After that, the transaction processor proceeds to execute the transactions in accordance with the wording of the smart contract that has been deployed by the participants of the transaction as illustrated in Fig 6.7.

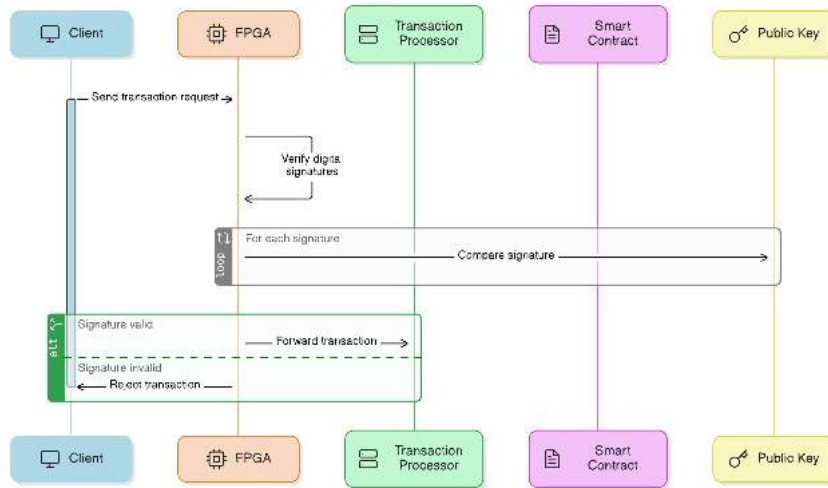


Figure 6.7: Data Processing

6.6.4 Transaction Execution

The proposed transaction processor employs an FPGA and can process transaction in real-time and with high speed attributable to the parallel processing nature of an FPGA. This ensures efficient and secure operations of transactions in CBDC to guarantee the requisite efficiency of the transactions involving the CBDC in the ecosystem as shown in Fig 6.8. However, one significant weakness of the implementation is that the server handles TSP for data authentication and HTTPS for transmission; thus, it provides sound security throughout transactions and high-performance transaction processing.

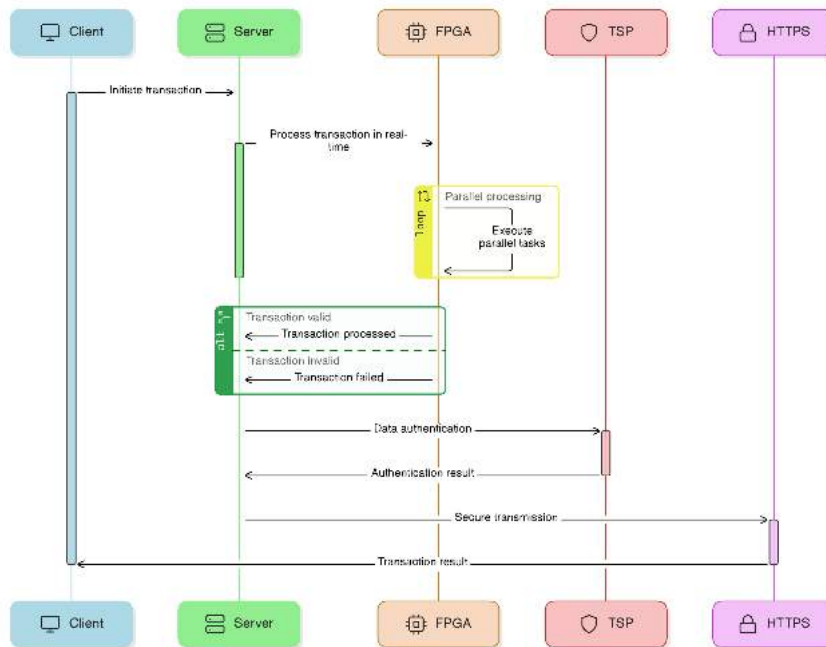


Figure 6.8: Transaction Execution

6.6.5 Performance Optimization

The main strategies of the architecture of the transaction processor which uses the FPGA and is aimed at the enhancement of the performance include the pipelining, parallel conducting and employment of dedicated processing channels as shown in the Fig 6.9. All these techniques help to increase the many transactions per second through put, decrease the time taken to perform the transaction and improve the overall system performance. Furthermore, there is other implemented hardware in FPGA including SHA-256 and RSA employed for cryptographic computation. Adding to it, to speed up and instill higher security in the processing of CBDC transactions these are the advanced hardware-accelerated cryptographic operations.

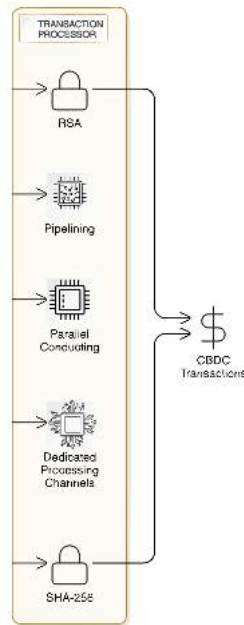


Figure 6.9: Performance Optimization

6.7 Conclusion

This chapter has discussed how Hyperledger fabric can be integrated with an FPGA-based transaction processor for the execution of CBDC transactions. Distribution included establishment of secure communication between the Hyperledger Fabric network and the FPGA-based processor, therefore the HTTP and the Transaction Signatory Protocol for data authentication, privacy and accuracy in transit.

We talked about the fact that, within the CBDC ecosystem, transactions are created in JSON, and transmitted with the help of HTTPS on the Internet. While the above-mentioned transactions are initiated, the other component of our architecture, the FPGA-based transaction processor, checks the digital signatures of the received transactions using public key cryptography and processes each of the transactions based on the rules defined in the smart contracts implemented in the said architecture.

Moreover, we explained assessing the performance optimization factors upon the FPGA-based transaction processor with the help of hardware acceleration techniques such as

pipelining, parallel processing, dedicated-processing units. Also, to support cryptographic computations required for CBDC operations, there is optimized hardware on the FPGA layer, including on SHA-256, as well as RSA for additional performance and security.

In general, combining Hyperledger Fabric with an FPGA-based transaction processor would provide the best solution for reliable and fast CBDC handling. Therefore, by the integration features like using secure transmission protocols and hard-wired acceleration, the integration enables more transaction throughput, decrease in the processing time of transactions as well as making the entire CBDC ecosystem operate seamlessly.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The concept and creation of CBDCs are a positive step towards the evolution of the current world's financial systems and improving the effectiveness, safety, and inclusion of digital payments. This project has therefore provided solutions to major challenges that have been seen with the implementation of CBDCs including slow transaction speeds, high costs, privacy and security concerns and scalability problems through use of FPGA technology and Hyperledger Fabric.

Design and development of a CBDC transaction processor using FPGA can offer a high transaction rate and low latency owing to the parallelism of FPGA. Hyperledger Fabric is a permissioned blockchain that provides better privacy, security and scalability than other types of blockchain to meet the needs of CBDCs.

Several goals were set, namely, to design a reliable transaction processing system, incorporate FPGA, use Hyperledger Fabric, and address specific issues. Thus, the overall assessment and validation of the performance of the system developed ensured its efficiency and accuracy. The deliverables of the project involve; efficient CBDC transaction processing systems, integration of FPGA technology and solutions to challenges affecting

CBDCs. Spreading the results about digital currency and blockchain technologies through academic papers and conferences helps the improvement of these systems.

The approach that was used in this project utilized the different features of the FPGAs and ARM processors in the Zynq-7000 SoC by designing the system in a way that it could efficiently manage a large number of transactions in a real time and low power mode. In the light of the presented architecture, digital wallets acted as a user interface, processing the transactions in JSON-RPC format and authenticated by JSON Web Tokens (JWT). The FPGA employed in this experiment was responsible for the important cryptographic operations including hashing using the SHA-256 algorithm as well as encryption using the RSA algorithm to improve the security of the transactions.

SHA-256 and RSA algorithms were used in this study with the aid of Vivado Design Suite in order to design the custom IP cores for SHA-256 and RSA algorithms on FPGA and the efficient and secure data processing was observed. These IPs were connected to the transaction reception system through AXI DMA which was of great benefit in the management of the memory access and processing.

Considering the Hyperledger Fabric, the integration of the FPGA-based transaction processor ensured secure and efficient handling of the CBDC transactions. This combined system employed secure transmission protocols and hardware acceleration to increase the number of transaction per second, reduce the time take to complete a transaction and hence making it suitable for the CBDC environment.

Thus, this project has suggested solutions and recommendations to the issues that affect the implementation of the CBDC and has positively influenced the advancement of the digital currency framework and future prospects. Based on the proposed model of incorporating FPGA into the Hyperledger Fabric in digital currency systems, the following can be predicted to be the benefits of the model.

7.2 Future Work

Based on the findings of this project, several recommendations for future research are presented to strengthen the CBDCs' implementation and effectiveness:

1. **Advanced Cryptographic Techniques:** Research and include better cryptographic algorithms and techniques to improve on security and efficiency. This entails the integration of post-quantum cryptography to prevent the system from being vulnerable to quantum computational threats.
2. **Optimization of FPGA Resource Utilization:** Propose techniques for minimizing the use of the FPGA resources especially the Look-Up Tables (LUTs) and increase the efficiency of the hardware implementation. This may include optimizing the HDL code and review the utilization of larger or more complex FPGA devices.
3. **Scalability Enhancements:** Propose and evaluate mechanisms for improving the system's capacity to handle more transactions and users. This could involve some of the following in Hyperledger Fabric and the interaction between FPGA modules and the blockchain network.
4. **Interoperability with Other Blockchain Platforms:** Examine possibilities of connecting the CBDC system with other blockchain-based systems to further enhance its usage and compatibility with the other financial systems. This may entail designing methods for inter and intra blockchain communication and also checking for compatibility with other block chain technologies.
5. **User Interface Improvements:** Improve the design of the digital wallets to make it easier to use and identify by consumers. This entails creating applications that are both for mobile and web platform with features such as easy and secure transaction processing, real time notifications and complete history of all transactions.
6. **Regulatory Compliance and Legal Framework:** Cooperate with other financial

regulators to make sure that the CBDC system is in line with current legislation and to support the further elaboration of legal frameworks for digital currencies. This may include the incorporation of compliance features and the submission of reports to the processing system of the transactions.

7. **Performance Benchmarking and Stress Testing:** Perform detailed performance analysis and stress testing under various conditions in order to determine the areas of the application that are most likely to cause performance problems. This entails checking how the system behaves when faced with a large number of transactions, slow network, and hardware breakdowns.
8. **Real-World Pilot Programs:** Pilot the implementation of the system with central banks and financial institutions in order to assess the real world effectiveness of the system. These pilots are useful in gathering information on the usage of the system, identifying the resilience of a system and the issues that may be encountered in an actual operation.
9. **Artificial Intelligence and Machine Learning Integration:** Explore how the current AI and ML technologies can be included in the practices of transaction monitoring, fraud detection, and the proactive analytics. These technologies can add on other features which can enhance the authentication process.

stability and effectiveness, thus guaranteeing the soundness and stability of the CBDC system.
10. **Energy Efficiency Improvements:** Emphasize the energy efficiency improvement of the FPGA-based transaction processor and the entire CBDC system. This involves improving the hardware design and evaluating the use of low-power FPGA to develop a better design.
11. **Enhanced Privacy Mechanisms:** Research and implement advanced privacy-preserving techniques such as zero-knowledge proofs and secure multi-party computation to

ensure user data and transaction details remain confidential.

12. **User Education and Training Programs:** Develop comprehensive educational resources and training programs to help users, financial institutions, and developers understand and effectively utilize the CBDC system. This includes creating tutorials, documentation, and support channels.

By addressing these future work areas, the project can continue to evolve and adapt to the changing needs of the digital economy, ensuring the successful deployment and adoption of CBDCs on a global scale.

Bibliography

- [1] Bank for International Settlements, “Central bank digital currencies: foundational principles and core features.” <https://www.bis.org/publ/othp33.htm>.
- [2] R. Auer and R. Böhme, “The technology of retail central bank digital currency,” *BIS Quarterly Review*, March 2020.
- [3] N. Narula, L. Swartz, J. Frizzo-Barker, S. Frank, and C. Calabia, “Expanding financial inclusion or deepening the divide.” <https://dci.mit.edu/cbdc-fi-1>.
- [4] Federal Reserve Bank of Boston, “Project hamilton phase 1: A high-performance payment processing system designed for central bank digital currencies.” <https://www.bostonfed.org/publications/one-time-pubs/project-hamilton-phase-1-executive-summary.aspx>.
- [5] MIT Digital Currency Initiative, “Openbdc project overview.” <https://dci.mit.edu/openbdc>.
- [6] Digilent, “Zybo reference manual.” <https://digilent.com/reference/programmable-logic/zybo/reference-manual>.
- [7] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, “The zynq book.” <http://www.zynqbook.com/>, 2014.
- [8] AMD, “Petalinux tools documentation.” <https://docs.amd.com/r/en-US/ug1144-petalinux-tools-reference-guide>, 2023.

- [9] Xilinx, “Petalinux downloads.” <https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools/2023-2.html>.
- [10] C. Rajapaksha, “Installing ubuntu on xilinx zynq-7000 ap soc using petalinux.” <https://medium.com/developments-and-implementations-on-zynq-7000-ap/install-ubuntu-16-04-lts-on-zynq-zc702-using-petalinux-2016-4-e1d>
- [11] edmund, “Why petalinux always ask to set new password on every power reset.” https://support.xilinx.com/s/question/0D54U00006xEYL0SAO/why-petalinux-always-ask-to-set-new-password-on-every-power-reset?language=en_US.
- [12] AMD, “Partitioning and formatting an sd card.” <https://docs.amd.com/r/en-US/ug1144-petalinux-tools-reference-guide/Partitioning-and-Formatting-an-SD-Card>.
- [13] W. contributors, “Rsa (cryptosystem),” 2024. [Online; accessed 30-May-2024].
- [14] W. contributors, “Euclidean division,” 2024. [Online; accessed 30-May-2024].
- [15] W. contributors, “Modular arithmetic,” 2024. [Online; accessed 30-May-2024].
- [16] W. contributors, “Modular exponentiation,” 2024. [Online; accessed 30-May-2024].
- [17] R. Digital, “A comprehensive guide to digital design,” 2024. [Online; accessed 30-May-2024].
- [18] A. Developer, “Axi protocol overview,” 2024. [Online; accessed 30-May-2024].
- [19] R. Education, “Understanding axi protocol - lecture series,” 2024. [Online; accessed 30-May-2024].

- [20] A. Electronics, “Fpga basics: Architecture, applications, and uses,” 2024. [Online; accessed 30-May-2024].
- [21] “Multiplexer.” <https://en.wikipedia.org/wiki/Multiplexer>. Accessed: 2024-05-29.
- [22] “Asic.” <https://en.wikipedia.org/wiki/ASIC>. Accessed: 2024-05-29.
- [23] “D latch.” https://en.wikipedia.org/wiki/D_latch. Accessed: 2024-05-29.
- [24] “Rom.” <https://en.wikipedia.org/wiki/ROM>. Accessed: 2024-05-29.
- [25] “Eprom.” <https://en.wikipedia.org/wiki/EPROM>. Accessed: 2024-05-29.
- [26] “Eeprom.” <https://en.wikipedia.org/wiki/EEPROM>. Accessed: 2024-05-29.
- [27] “Random-access memory.” https://en.wikipedia.org/wiki/Random-access_memory. Accessed: 2024-05-29.
- [28] “Boolean function.” https://en.wikipedia.org/wiki/Boolean_function. Accessed: 2024-05-29.
- [29] “Truth table.” https://en.wikipedia.org/wiki/Truth_table. Accessed: 2024-05-29.
- [30] The Hyperledger Fabric Contributors, “Hyperledger fabric documentation.” <https://hyperledger-fabric.readthedocs.io/>, Accessed: 2024. Accessed on May 28, 2024.
- [31] The Linux Foundation, “Hyperledger fabric.” <https://hyperledger-fabric.readthedocs.io/>