DESIGN AND ANALYSIS Of SOC ARCHITECTURE FOR SECURED I0T DEVICES

A PROJECT REPORT

Submitted by

P.DEEPESH RAJ

R.SIVA KUMAR

Under the Guidance of

Dr. Lordwin Cecil Prabhakar M

in partial fulfillment for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

ELECTRONICS & COMMUNICATION ENGINEERING

Vel Tech
Rangarajan Dr. Sagunthala
R&D Institute of Science and Technology
(Deemed to be University Estd. u/s 3 of UGC Act, 1956)

MAY 2024

Vel Tech
Rangarajan Dr. Sagunthala
R&D Institute of Science and Technology
(Deemed to be University Estd. u/s 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this major project report entitled **"DESIGN AND ANALYSIS Of SOC ARCHITECTURE FOR SECURED IoT DEVICES"** is the bonafide work of **"P.Deepesh Raj (20UEEC0258) and R.Siva Kumar (20UEEC0290)"** who carried out the project work under my supervision.

SUPERVISOR	**HEAD OF THE DEPARTMENT**
Dr. Lordwin Cecil Prabhakar M	**Dr.A. Selwin Mich Priyadharson**
Associate Professor	Professor
Department of ECE	Department of ECE

Submitted for major project work viva-voce examination held on:______________________

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

We express our deepest gratitude to our respected Founder President and Chancellor **Col Prof. Dr. R. RANGARAJAN**, Foundres President **Dr. R. SAGUNTHALA RANGARAJAN**, Chairperson Managing Trustee and Vice President.

We are very thankful to our beloved Vice Chancellor Prof. **Dr. S. SALIVAHANAN**, for providing is with an environment to complete the work succesfully.

We are obligated to our beloved Registrar **Dr. E. KANNAN**, for providing immense support in all our endeavours. We are thankful to our esteemed Dean Academics **Dr. A. T. RAVICHANDRAN**, for providing a wonderful environment to complete our work successfully.

It is a great pleasure for us to acknowledge the assistance and contributions of our Head of the Department **Dr. A. SELWIN MICH PRIYADHARSON**, Professor for his useful suggestions, which helped us in completing the work in time and we thank him for being instrumental in the completion of final year with his encouragement and unwavering support during the entire course.

We are extremely thankful and pay our gratitude to our Dean SoEC **Dr. R. S. VALARMATHI**, for her valuable guidance and support on completion of this major project.

We are extremely thankful and pay our gratitude to our Project Coordinator **Dr. G. ALOY ANUJA MARY** for her valuable guidance and support on completion of this major project.

We are extremely thankful and pay our gratitude to our domain Major Project coordinator **Dr. M . SAHAYA SHEELA**, Assistant Professor for her valuable guidance and support on completing this major project report in a successful manner.

We are extremely thankful and pay our gratitude to our Supervisor **Dr. M. LORDWIN CECIL PRABHAKAR**,Associate Professor for his valuable guidance and support on completion of this major project.

P. DEEPESH RAJ

R. SIVA KUMAR

TABLE OF CONTENTS

ABSTRACT

Internet of Things (IoT) continues to expand, ensuring the security and integrity of data transmitted across IoT devices becomes paramount.With a specific emphasis on implementing the SHA-256 algorithm and incorporating six General Purpose Input/Output (GPIO) pins. In the realm of the Internet of Things (IoT), security is paramount, in order to address this imperative through the development of a System-on-Chip (SOC) architecture. The application of the SHA-256 algorithm is a strategic choice, known for its robust cryptographic properties, ensuring a high level of data security and integrity within IoT environments.The SOC architecture is meticulously crafted to strike a balance between security, performance, and power efficiency.The SHA-256 algorithm is seamlessly integrated, ensuring end-to-end encryption to safeguard the transmission of sensitive information.

The proposed architecture utilizes a Zynq UltraScale+ MPSoC board and integrates a SHA-256 encryption block to enhance security. The architecture incorporates GPIO for input, integrates a SHA-256 Verilog code for security purposes, and interfaces with a camera via I2C. The SHA-256 module is connected to SRAM, which communicates with an ALU and registers. The output is transmitted via UART for further analysis and processing.The results demonstrate that the proposed architecture not only provides robust security features but also exhibits exceptional performance and power efficiency, making it highly suitable for deployment in diverse IoT applications.A distinctive feature of the proposed architecture is the integration of six GPIO pins, enhancing the versatility and functionality of the IoT devices. These GPIO pins offer expanded capabilities, allowing for diverse applications and adaptability in various scenarios. It also involves a thorough analysis of the designed SOC architecture, assessing its effectiveness in fortifying the security posture of IoT devices. By combining the strength of the SHA-256 algorithm with the flexibility provided by the GPIO pins, in order to contribute to the advancement of secure IoT systems. The outcome is a comprehensive architectural solution that aligns cryptographic resilience with hardware versatility, addressing contemporary challenges in the evolving landscape of IoT security.In conclusion,it represents a novel SOC architecture specifically designed for securing IoT devices. By leveraging the SHA-256 algorithm and incorporating 6 GPIOs, the proposed architecture addresses critical security concerns and enables seamless connectivity, making it an ideal solution for ensuring the integrity and confidentiality of IoT device communication and data transmission.

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

IOT	-	Internet of Things
SHA	-	Secure Hash Algorithm
SOC	-	System on Chip
GPIO	-	General Purpose Input Output
MPSOC	-	Multiprocessor Systems-on-Chip
ALU	-	Arithmetic Logic Unit
UART	-	Universal Asynchronous Receiver / Transmitter
I2C	-	Inter-Integrated Circuit
SRAM	-	Static Random Access Memory
FPGA	-	Field Programmable Gate Array
APU	-	Application Processing Unit
RPU	-	Real Time Processing Unit
XDC	-	Xilinx Design Constraints File
RTL	-	Register Transfer Level

CHAPTER 1

INTRODUCTION

1.1 Design Of SOC Architecture

The SoC architecture that prioritizes security features to mitigate common IoT security threats, such as unauthorized access, data breaches, and malware attacks. This involves integrating robust encryption, authentication, and access control mechanisms into the architecture.The advantages of implementing security features at the hardware level to enhance the resilience of IoT devices against cyber attacks. Hardware-based security measures, such as secure boot, tamper detection, and cryptographic accelerators, will be considered to strengthen the overall security posture of the SoC architecture.An SoC architecture that balances security requirements with performance and resource constraints typical of IoT devices. This involves optimizing hardware components and algorithms to minimize power consumption, latency, and overhead while maximizing throughput and responsiveness.In order to ensure compatibility and interoperability with existing IoT ecosystems, protocols, and standards to facilitate seamless integration and interoperability with other devices, networks, and platforms.An SoC architecture that is scalable and adaptable to accommodate future enhancements, updates, and expansions in IoT applications and functionalities. This involves adopting modular and flexible design principles to facilitate easy upgrades and modifications.Mainly cost effectiveness as a critical factor in the design and implementation of the SoC architecture. Explore cost-efficient solutions and optimizations without compromising on security, reliability, or performance.To conduct a comprehensive analysis and evaluation of the proposed SoC architecture to assess its effectiveness in addressing IoT security challenges. This includes performance testing, security audits, simulation-based validation, and comparative analysis.

1.2 Overview Of IOT Security Challenges

The Internet of Things (IoT) has revolutionized the way devices communicate and interact with each other, offering unprecedented convenience and efficiency. However, the proliferation of IoT devices has also brought about a myriad of security challenges that need to be addressed to ensure the integrity, confidentiality, and availability of data and services.Many IoT devices are resource-constrained and lack robust security features, making them susceptible to various attacks such as malware, ransomware, and unauthorized access. These vulnerabilities can be exploited to compromise the device's functionality or gain unauthorized access to sensitive data.IoT devices often rely on wireless communication protocols such as Wi-Fi, Bluetooth, and Zigbee to transmit data. However, these protocols may have inherent security weaknesses, such as lack of encryption or authentication, making them vulnerable to eavesdropping, spoofing, and man-in-the-middle attacks.IoT devices collect vast amounts of data about users' behaviors, preferences, and surroundings. Ensuring the privacy of this data is crucial to maintaining user trust and compliance with privacy regulations. However, inadequate data encryption, storage, and access controls can result in data breaches and privacy violations.Authenticating and authorizing users and devices is essential for preventing unauthorized access to IoT systems and resources. Weak authentication mechanisms, such as default or hardcoded credentials, can be exploited by attackers to gain unauthorized access and control over IoT devices.IoT devices often run on embedded firmware or software that may contain security vulnerabilities. Ensuring timely and secure firmware/software updates is crucial for patching known vulnerabilities and protecting devices against emerging threats. However, the lack of standardized update mechanisms and device heterogeneity can make the update process challenging and error-prone.IoT devices deployed in uncontrolled environments, such as industrial facilities or smart homes, are vulnerable to physical tampering and theft. Securing physical access to IoT devices and implementing measures such as tamper detection and device hardening are essential for mitigating physical security risks.IoT devices often collect and transmit sensitive personal data, such as location information, health data, and behavioral patterns. Securing these devices helps prevent unauthorized access to personal data, reducing the risk of identity theft, fraud, and privacy violations.Unsecured IoT devices can serve as entry points for cyber attackers to infiltrate networks and launch malicious activities. By securing IoT devices, organizations can prevent unauthorized access, data breaches, and disruption of critical services. Many IoT devices are deployed in critical infrastructure and industrial systems, where downtime or disruption can have severe consequences. Securing IoT devices helps maintain business continuity by preventing cyber attacks, minimizing disruptions, and ensuring the reliability of essential services.IoT devices often contain proprietary information, firmware, and software that are valuable assets for organizations. Securing IoT devices helps protect intellectual property from theft, reverse engineering, and unauthorized tampering, safeguarding organizations' competitive advantage and innovation.

CHAPTER 2

LITERTAURE SURVEY

2.1 Overview

Bongsoo Lee et.al. [1] stated that the rapid advancement of the 4th industrial revolution, coupled with the robust capabilities of the 5th Generation (5G) era, has propelled substantial growth within the Internet of Things (IoT) industry. This surge is particularly evident in various sectors, including smart cars, smart homes, and smart healthcare. Concurrently, the escalating number of IoT devices underscores the heightened significance of ensuring their security. This study addresses this imperative by proposing a novel design methodology for a secure cryptographic system on a chip (SecSoC) tailored for application in the IoT industry. The research not only introduces the design but also provides comprehensive insights into the performance and security evaluations of the implemented chipset. Notably, the experimental results underscore the SecSoC's efficacy as a low-power, high-performance cryptographic chip, resilient against external threats. Distinguishing itself from conventional smart card integrated circuits, the proposed design integrates intrusion detection circuits to counteract external attacks. Furthermore, it incorporates a physical unclonable function for concealing secret data and cryptographic logic, ensuring the integrity and confidentiality of information. Remarkably, the SecSoC achieves a swift transfer rate of up to 110 Mbps, demonstrating exceptional efficiency while operating at a maximum frequency with a minimal power consumption of 95.8 mW. This research contributes to the burgeoning field of secure IoT devices, offering a promising solution to the escalating challenges associated with cybersecurity in the era of interconnected smart technologies.

Ayman Hroub and Muhammad E. S. Elrabaa [2] provided a method for the pervasive integration of Internet of Things (IoT) technology into diverse facets of daily life has ushered in an era of continual innovation, unveiling novel applications with each passing day. However, this widespread adoption has introduced a host of challenges, particularly in the realm of device and data security. A noteworthy issue arises from the shifting deployment of IoT devices to locations beyond the owner's premises, onto public or private property. This relocation exposes the devices to potential physical access by adversaries, enabling attacks that can subvert conventional protection methods. This paper addresses these challenges by proposing Secure System-on-Chip (SecSoC), an architecture designed to mitigate various attacks, including logical memory dump attacks, bus snooping attacks, and those targeting compromised operating systems. SecSoC employs two core mechanisms: firstly, it extends security features to the compute engine running user applications without altering the instruction set, and secondly, it integrates a Security Management Unit (SMU) providing hardware security primitives for encryption, hashing, random number generation, and secret storage. Notably, SecSoC ensures that no sensitive data exits the System-on-Chip (SoC) in plaintext. The implementation of SecSoC in Bluespec SystemVerilog is detailed, with forthcoming experimental results expected to elucidate the associated area, power, and cycle time overheads introduced by the security extensions. Furthermore, the paper outlines the evaluation of overall performance, measured through total execution time using established IoT benchmarks. This research contributes a robust solution to the evolving landscape of IoT security, addressing the specific challenges posed by the dynamic deployment scenarios of IoT devices.

Ahmad Showail et.al.[3] stated that as low-cost networked Internet of Things (IoT) devices become increasingly prevalent, embedded in various aspects of our lives, ranging from medical implants to smart cars and home automation systems, the extensive adoption of these devices has led to a proliferation of security, privacy, and trust issues unique to the IoT ecosystem. Traditional solutions prove inadequate in this dynamic landscape due to resource constraints, network dynamics, and evolving trust boundaries. In response to these challenges, this paper introduces a user-centric, cloud-based service aimed at empowering device owners with precise control over the type and extent of data shared through their IoT devices. Leveraging Intel Software Guard Extensions (SGX), the proposed scheme creates secure virtual clones or shadows of actual devices in the cloud, thereby minimizing the attack surface for IoT networks. The scalable cloud infrastructure facilitates the implementation of sophisticated policy enforcement and data scrubbing mechanisms on a per-application basis, affording users explicit control over data sharing. Notably, this approach places minimal burden on device vendors and users, as the service provider undertakes the bulk of the implementation. The empirical demonstration of the effectiveness of this approach involves the implementation of the service on SGX hardware, showcasing the deployment of advanced data cleansing policies on data generated by IoT devices. This research offers a promising avenue for addressing the unique security and privacy challenges in the evolving landscape of IoT, emphasizing user-centric control and cloud-based security mechanisms.

Kenneth Li-Minn Ang and Jasmine Kah Phooi Seng [4] provided a method for the burgeoning field of embedded intelligence (EI) research within the context of smart cities lacks a comprehensive survey to date. Addressing this gap, this article undertakes a thorough review of the state of the art in EI research for smart cities, encapsulating a broad spectrum of emerging and current paradigms. The review is organized into four key areas to offer a holistic understanding: firstly, it provides an overview and classifications of EI research, serving as a concise summary of the article's scope; secondly, it delves into the enabling technologies, encompassing EI platform technologies and EI and device analytics technologies; thirdly, the article explores diverse applications of EI, utilizing the identified technologies and techniques for the advancement of smart cities; finally, it discusses challenges and outlines insights, offering a roadmap for future research directions. By presenting representative studies and discussing key facets such as enabling technologies, applications, and challenges, this comprehensive survey not only fills a critical gap in the existing literature but also aims to provide valuable insights for researchers, guiding them towards the development of practical and deployable EI solutions for smart cities.

Eduardo B. Fernandez et.al.[5] stated that the security challenges inherent in Internet of Things (IoT) systems, characterized by diverse devices from various owners and manufacturers, demand effective solutions. This complexity is compounded by the integration of IoT applications with cloud and fog components within larger cyber-physical systems. Addressing these security issues requires a design capable of navigating the intricacies and heterogeneity of IoT ecosystems, and security patterns offer a promising approach due to their abstraction power. This literature review surveys and classifies existing IoT security patterns to assess their coverage and quality for potential inclusion in a comprehensive catalog. The analysis reveals that the current number of patterns is insufficient and often incomplete, necessitating the creation of a unified catalog. To address this gap, the authors propose new patterns and standardize existing ones. Furthermore, the review explores IoT development methodologies, discovering a lack of consideration for security or pattern usage. As a solution, the authors suggest adapting existing pattern-based methodologies for distributed systems, citing a variety of examples for reference. The article concludes by presenting potential research directions, emphasizing the critical need for a unified and comprehensive approach to IoT security patterns and development methodologies.

Ramesh Krishnamoorthy and Kalimuthu Krishnan [6] provided a method that the escalating growth of embedded devices has intensified the role of System-on-Chip (SoC) selection in shaping hardware security within embedded system design. SoC devices, comprising one or more CPUs and a variety of integrated peripherals, offer cost-effective options for system design. The choice of SoC becomes pivotal in determining the suitability for secure application development. This literature review conducts a design space analysis of symmetric key approaches, encompassing cryptographic algorithms such as Rivest Cipher (RC5), Advanced Encryption Standard (AES), Data Encryption Standard (DES), International Data Encryption Algorithm (IDEA), Elliptic Curve Cryptography (ECC), MX algorithm, and the Secure Hash Algorithm (SHA-256). Comparative evaluations are performed to identify the most suitable algorithm for on-chip security implementation. The study leverages Genetic Algorithm (GA) optimization to assess the fitness of SoC with enhanced crypto algorithms, considering power and area consumption requirements through FPGA-based implementations. The proposed work extends its validation by implementing security benchmarks on GA-derived hardware devices, evaluating execution time and performance. Algorithm accuracy is estimated using the confusion matrix method, achieving an impressive 89.66% accuracy. Additionally, the literature extensively discusses various challenges associated with transitioning from IoT system design prototypes to industrial applications, providing a comprehensive overview of the hurdles that must be overcome in this domain.

Abdulmalik Alwarafy et.al [7] proposed that the Internet of Things (IoT) has evolved into a transformative paradigm, integrating seamlessly into various aspects of our daily lives with millions of smart devices deployed in complex networks for communication, monitoring, and control of critical infrastructures. However, the exponential growth of IoT devices and the accompanying surge in data traffic at the network edge have strained the traditional centralized cloud computing paradigm due to bandwidth and resource constraints. In response, edge computing (EC) has emerged as an innovative strategy, positioning data processing and storage closer to end-users, thereby giving rise to the EC-assisted IoT paradigm. While this approach enhances Quality of Service (QoS), it introduces significant challenges in data security and privacy. This article conducts a comprehensive survey of security and privacy issues within the context of EC-assisted IoT. The review encompasses an overview of EC-assisted IoT, including definitions, applications, architecture, advantages, and challenges. Security and privacy in the context of EC-assisted IoT are defined, followed by an extensive discussion on major attack classifications and potential solutions, highlighting related research efforts. The literature is further categorized based on security services and objectives, providing a nuanced understanding of the landscape. The survey concludes by identifying open challenges and outlining future research directions for securing the EC-assisted IoT paradigm, offering valuable insights for researchers and practitioners grappling with the evolving complexities of this innovative approach.

Wazir Zada Khan et.al.[8] stated that The Social Internet of Things (SIoT) represents an extension of the Internet of Things (IoT), integrating with social networking concepts to form networks of interconnected smart objects, introducing a novel paradigm where human-to-thing (H2T) interactions complement the existing human-to-human (H2H) and thing-to-thing (T2T) interactions of IoT. This convergence results in the creation of smart "social objects" (SOs) that emulate human social behavior in daily life. These SOs possess social functionalities enabling them to discover and establish relationships with other objects, navigating the social network to seek relevant services and information. Despite the significant advancements in SIoT, the understanding of trust and trustworthiness within these emerging social communities is nascent. This article makes three key contributions: firstly, it elucidates the fundamentals of SIoT and trust concepts, differentiating between IoT and SIoT. Secondly, it categorizes and comprehensively reviews trust management solutions proposed in the literature for SIoT over the last six years, offering a comparative analysis of the state-of-the-art trust management schemes in terms of their processes. Lastly, the article identifies and discusses challenges and requirements in the evolving SIoT landscape, emphasizing the complexities associated with developing trust and evaluating trustworthiness among interacting social objects. This review provides a valuable synthesis of current research, paving the way for a deeper understanding of trust dynamics within the context of SIoT and guiding future research endeavors in this burgeoning field.

Manel Kammoun et.al.[9] stated that the SHA-2 stands out as a widely adopted hash function, renowned for ensuring information integrity and authenticity. Despite its efficiency, this paper highlights the associated drawbacks of heightened computational complexity and increased power consumption. To address these challenges, the study proposes a hardware acceleration solution, emphasizing the need for an optimal balance between speed and power consumption. Specifically, the research concentrates on the software/hardware (SW/HW) implementation of the SHA-256 hash function utilizing high-level synthesis (HLS) on Xilinx Zynq 7000-based FPGA. The synthesis results of the SW/HW solution showcase a notable gain, with up to 17% improvement in execution speed and an impressive 73% reduction in power consumption compared to the software-only case. Additionally, the hardware HLS architecture exhibits a remarkable 66% increase in throughput when compared to previous work, emphasizing the efficacy of the proposed approach in enhancing the performance and power efficiency of SHA-256 hash function implementations.

Nourah Almrezeq1 et.al.[10] proposed a method for The Internet of Things (IOT) is a revolution in thetechnology world, and this field is continuously evolving. It has made life easier for people by providing consumers with more efficient and effective resources in faster and more convenient ways.The Internet of Things is one of the most exciting fields for the future by 2030. 90% of the planet will be connected and all devices in homes and businesses around us will be connected to the Internet making it more vulnerable to violations of privacy and protection. Due to the complexity of its environment, security and privacy are the most critical issues relevant to IoT. Without the reliable security of the devices, they will lose their importance and efficiency. Moreover, the security violation will outweigh any of its benefits. In this paper, an overview of various layered IoT architectures, a review of common security attacks from the perspective of the layer, and the best techniques against these attacks are provided. Moreover, an enhanced layered IoT architecture is proposed, which will be protected against several security attacks.

Zandberg, Koen, et al.[11] stated that Real-world applicability is a key consideration in assessing the practical utility of secure firmware update mechanisms. Evaluation on commercially available constrained IoT devices offers valuable insights into the scalability, efficiency, and compatibility of prototype implementations across diverse hardware platforms. By benchmarking performance metrics such as update latency, resource utilization, and energy efficiency, researchers can ascertain the viability of these solutions in real world deployment scenarios.Critical to the effectiveness of any secure firmware update solution is its ability to uphold rigorous security standards while accommodating the resource limitations inherent in constrained IoT devices. Through rigorous evaluation, researchers have scrutinized the security properties of prototype implementations, assessing factors such as resilience to attacks, cryptographic robustness, and adherence to established standards.A comprehensive survey of open standards and open-source libraries reveals a diverse array of building blocks available for implementing secure firmware update mechanisms tailored to the specific constraints of IoT devices. These building blocks encompass protocols, cryptographic algorithms, and authentication mechanisms designed to ensure the integrity, confidentiality, and authenticity of firmware updates.

Ray, Partha Pratim [12]proposed that a central tenet of the surveyed architectures is their practical utility in addressing real-life challenges through the deployment of robust IoT solutions. Whether directly or indirectly, these architectures offer methodologies for tackling a myriad of issues across domains such as healthcare, transportation, agriculture, and smart cities. By leveraging IoT technologies, stakeholders can harness the power of data-driven insights to optimize processes, improve decision-making, and enhance overall efficiency.Despite the progress made in IoT architecture development, several research challenges persist, necessitating ongoing exploration and innovation. By identifying these challenges, the literature review serves to stimulate academic and industrial involvement in addressing key issues such as interoperability, scalability, security, and privacy. By confronting these challenges head-on, researchers can unlock the full potential of IoT architectures and pave the way for transformative advancements in diverse domains.Despite the progress made in IoT architecture development, several research challenges persist, necessitating ongoing exploration and innovation. By identifying these challenges, the literature review serves to stimulate academic and industrial involvement in addressing key issues such as interoperability, scalability, security, and privacy. By confronting these challenges head-on, researchers can unlock the full potential of IoT architectures and pave the way for transformative advancements in diverse domains.

Ngu, Anne H., et al [13] stated that a comprehensive survey of existing IoT middleware solutions provides insights into the diverse range of capabilities and functionalities offered by different platforms. These middleware solutions serve as the backbone of IoT ecosystems, offering support for device discovery, data aggregation, protocol translation, and application development. By analyzing the features and limitations of existing middleware solutions, researchers gain valuable insights into the evolving landscape of IoT middleware and the challenges associated with supporting heterogeneous IoT environments.Central to the development of effective IoT middleware are the essential ingredients of composition, adaptability, and security. Middleware solutions must be capable of dynamically composing and orchestrating services from diverse IoT devices, adapting to changing environmental conditions and user requirements, and ensuring robust security measures to protect sensitive data and mitigate potential threats.

Ortiz, Antonio M., et al. [14] proposed that The convergence of the Internet of Things (IoT) and social networks (SNs) represents a transformative cluster that facilitates the seamless connection between individuals and the ubiquitous computing universe. In this symbiotic framework, IoT devices gather and transmit data from the environment, while social networks serve as the conduit for human-to-device interactions, fostering a dynamic ecosystem of connectivity and collaboration. This literature review explores the emerging paradigm of the Social Internet of Things (SIoT), which extends beyond traditional IoT concepts to incorporate social-driven interactions and user centric perspectives. By providing a comprehensive overview of SIoT, this review aims to elucidate key perspectives and envision the realization of ubiquitous computing in the digital age.Building upon the insights gleaned from the literature review, this paper proposes a generic SIoT architecture that encapsulates the essential components and functionalities required to realize the vision of ubiquitous computing. The architecture emphasizes the seamless integration of IoT devices with social networks, enabling dynamic interactions and information exchange between users and smart objects. By providing a structured framework for SIoT development, the proposed architecture serves as a roadmap for future research and innovation in the field.

Xu, Teng. , et al.[15] stated that Computer-aided design (CAD) has been instrumental in driving design revolutions across various domains, continuously evolving to accommodate emerging technological trends and application demands. In the current landscape, one of the most significant emerging directions with the potential to reshape everyday life and engineering practices is the Internet of Things (IoT). IoT presents a vast and complex system fraught with conceptual and technical challenges, particularly in the realm of security. This literature review explores the intersection of CAD and IoT security, identifying key challenges, opportunities, and emerging research directions aimed at addressing the critical need for robust security mechanisms in IoT deployments.Recognizing the inherent vulnerabilities and limitations of software-based security mechanisms, the review explores the potential of hardware-based approaches to enhance IoT security. By leveraging hardware-level security features such as Physical Unclonable Functions (PUFs) and digital PUFs, researchers aim to bolster the resilience of IoT devices against a wide range of threats, including tampering, spoofing, and data manipulation. These hardware-based security solutions offer increased robustness and tamper resistance, thereby mitigating the risk of unauthorized access and ensuring the integrity of IoT ecosystems.To illustrate the practical application of hardware-based IoT security approaches, the review presents several case studies that advocate the use of stable PUFs and digital PPUFs in IoT security protocols. These case studies showcase the effectiveness of hardware-based security mechanisms in enhancing the trustworthiness and resilience of IoT systems, particularly in scenarios where traditional software-based approaches may fall short. By highlighting successful implementations and real-world use cases, the review underscores the potential of hardware-based IoT security techniques to address the evolving threat landscape and safeguard critical IoT deployments.

CHAPTER 3

METHODOLOGY

3.1 System Architecture

The proposed SoC architecture is designed to provide a secure framework for an IoT device. It incorporates various hardware components and interfaces to enable secure data processing and communication. The architecture prioritizes security features, leveraging hardware-based encryption and authentication mechanisms to protect sensitive data.It provides a secure framework for IoT devices, integrating essential hardware components and interfaces for data input, processing, and output. By incorporating features such as GPIO for input, SHA-256 for data integrity, I2C for camera interfacing, and UART for output, the architecture enables secure and efficient operation in IoT applications.

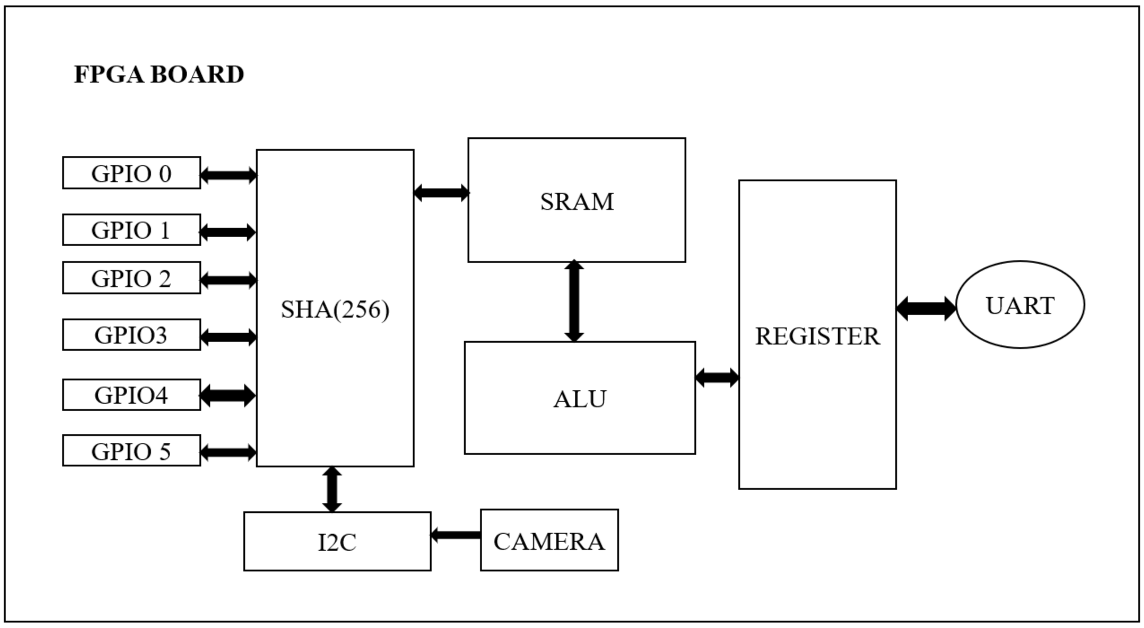


Figure 3.1: Block Diagram Of SOC

3.1.1 GPIO

General Purpose Input/Output (GPIO) pins are used to interface with external devices or sensors. In this architecture, GPIO 6 is designated as an input pin, allowing the device to receive external signals or commands.

3.1.2 Integration of SHA-256 Verilog Code

The Secure Hash Algorithm 256 (SHA-256) is a cryptographic hash function used for data integrity and authentication. The SHA-256 Verilog code is integrated into the SoC architecture to provide secure hashing capabilities. This block receives input data and computes the SHA-256 hash value, ensuring data integrity and security.

3.1.3 I2C Interface for Camera Input

The Inter-Integrated Circuit (I2C) interface is utilized to communicate with an external camera module. The camera captures images or video data, which is then transmitted to the SoC for processing and analysis. The I2C protocol facilitates communication between the camera and the SoC, allowing for seamless data transfer.

3.1.4 Connection Between SHA-256 and SRAM

The SHA-256 block is connected to Static Random Access Memory (SRAM), which serves as temporary storage for intermediate data during the hashing process. The SRAM provides fast and efficient access to memory, facilitating the computation of SHA-256 hash values.

3.1.5 ALU and Register Connections

The Arithmetic Logic Unit (ALU) performs arithmetic and logic operations on input data. It is connected to registers, which store temporary data and control signals. The ALU and register connections enable data manipulation and processing within the SoC architecture.

3.1.6 UART Output

Universal Asynchronous Receiver Transmitter(UART) is used for serial communication between the SoC and external devices or systems. In this architecture,the UART block facilitates the transmission of processed data or status information to external devices or host systems.It converts parallel data from the SoC into serial data streams for transmission over UART communication channels.

3.2 Design Methodology

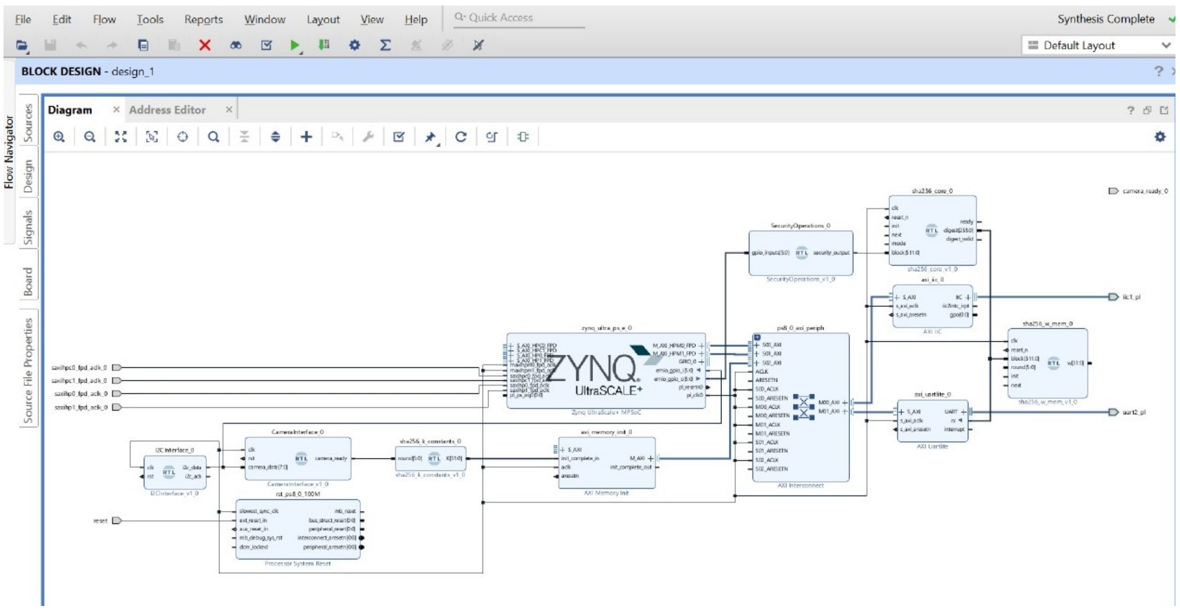


Figure 3.2: Structure of The Proposed Design

3.2.1 GPIO as Input

Determine the required input pins based on system requirements. Configure GPIO 6 as an input pin in the SoC configuration settings.By Utilizing the GPIO module available in the SoC architecture, ensuring compatibility and seamless integration.Implementing the input validation mechanisms to prevent buffer overflows, data injection attacks, and unauthorized access through GPIO pins.By Optimizing the GPIO configurations for minimal power consumption and latency, balancing performance with security requirements.

3.2.2 Integration of SHA-256 Verilog Code

Integrating the SHA-256 Verilog code into the SoC architecture, ensuring compatibility with the target hardware platform.Selecting a verified SHA-256 Verilog implementation with a proven track record of security and reliability.SHA-256 is a widely used cryptographic hash function known for its strong security properties and resistance to collision attacks.Implementing secure key management,randomization,and protection against side-channel attacks to enhance the security of the SHA-256 module.Optimizing the Verilog code for efficient resource utilization, parallel processing, and high throughput without compromising security.

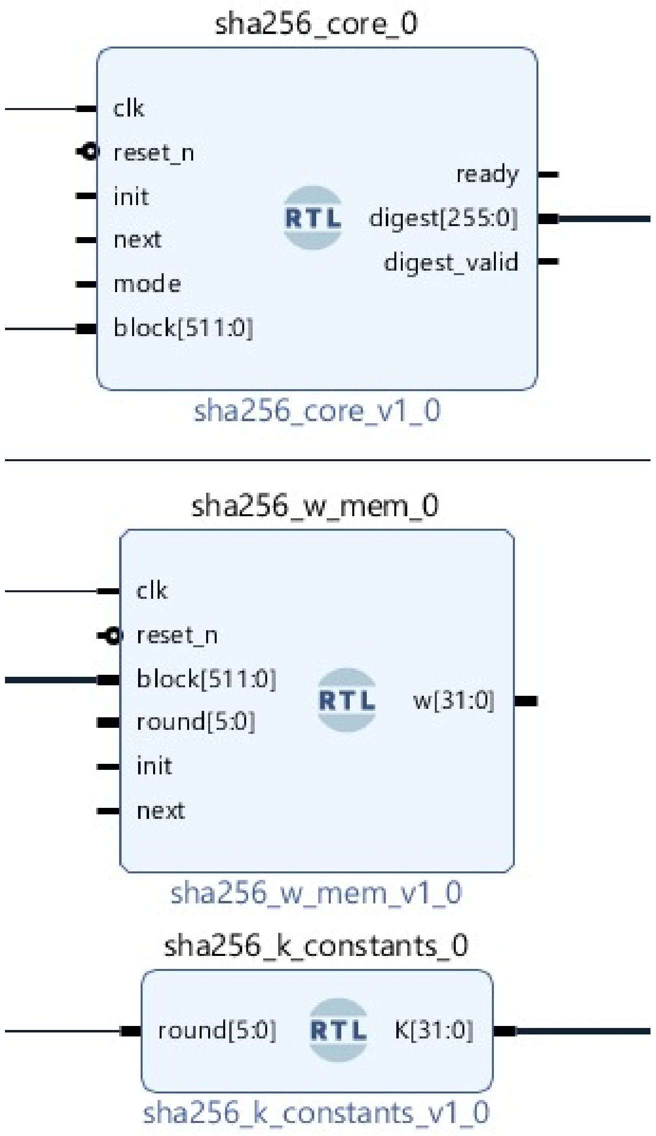


Figure 3.3: SHA-256

3.2.3 SHA-256 Verilog Code

The SHA-256 algorithm is a cryptographic hash function that takes an input message and produces a fixed-size output hash value.The Verilog code implementation of SHA-256 involves several key steps, including message padding, message scheduling, and hash computation.Below is a high-level overview of the implementation details:

- **Message Padding:**The input message is padded to ensure its length is a multiple of 512 bits, as required by the SHA-256 algorithm. Padding typically involves appending a single '1' bit followed by a series of '0' bits, with the last 64 bits representing the original message length.

- **Message Schedule:** The padded message is divided into 512-bit blocks, and each block is further divided into 16 32-bit words. These words are used to generate additional 64 words, known as the message schedule, through a series of bitwise operations and logical functions.

- **Initial Hash Values:** The SHA-256 algorithm uses a set of initial hash values, known as the H constants, which are combined with the message schedule during hash computation. These constants are derived from the fractional parts of the square roots of the first 8 prime numbers.

- **Hash Computation:** The main hash computation loop iterates over each 512-bit block of the padded message. Within each iteration, the current block is processed using a series of logical functions, including bitwise XOR, AND, OR, and NOT operations, along with modular addition and cyclic shifts.

- **Final Hash Value:** After processing all blocks, the final hash value is computed by concatenating the intermediate hash values obtained during each iteration. The resulting hash value represents the unique fingerprint of the input message and serves as its cryptographic hash.

3.2.4 I2C Interface for Camera Input

By Configuring the I2C interface to establish communication with the camera module, adhering to the I2C protocol specifications.Selecting an I2C controller compatible with the SoC architecture and the camera module, ensuring seamless data transfer. I2C is a widely used serial communication protocol suitable for connecting peripherals with low-speed data transfer requirements.By implementing the data encryption and authentication mechanisms to secure communication between the SoC and the camera module.To optimize I2C bus speed, clock frequency, and data transfer protocols for efficient communication and minimal latency.Each device connected to the I2C bus is assigned a unique 7-bit or 10-bit address. The camera module should have a pre-defined address, which is used by the SoC to initiate communication.The I2C bus operates at different clock speeds (standard mode, fast mode, and high-speed mode). The clock speed should be configured based on the camera module's specifications and the capabilities of the SoC.Before communicating with the camera module, the I2C interface on the SoC must be initialized. This involves configuring the relevant registers, enabling the I2C peripheral, and setting up the clock speed and addressing mode.By configuring the I2C interface and implementing the appropriate data transfer protocols and synchronization mechanisms, the SoC can effectively communicate with the camera module and exchange data for image capture, processing, and analysis in IoT applications.

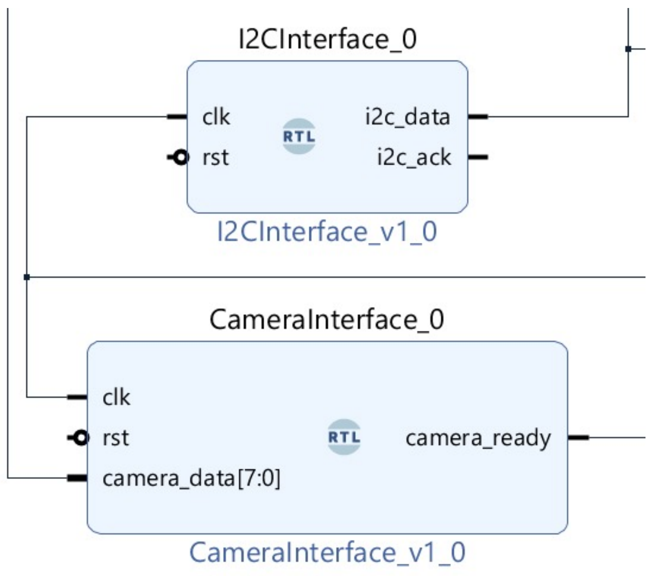


Figure 3.4: I2C Interface

3.2.5 Mechanism Of I2C

Start and Stop Conditions: Communication on the I2C bus begins with a start condition (S) and ends with a stop condition (P). These conditions indicate the beginning and end of a data transfer sequence, respectively.

- **Addressing:** The SoC initiates communication by sending the address of the camera module on the I2C bus. If the camera module acknowledges the address, data transfer can proceed. If not, the SoC can retry or abort the communication.

- **Data Transfer:** Data transfer on the I2C bus occurs in bytes, with each byte transmitted in 8 bits. The SoC can send or receive data from the camera module using read and write operations. These operations involve sending or receiving data bytes while monitoring the acknowledge (ACK) signal from the camera module.

- **Synchronization:**Proper synchronization between the SoC and the camera module is essential for reliable data transfer. The SoC must ensure that clock signals are generated at the correct frequency, and data is sampled and transmitted at the appropriate times.

- **Error Handling:** The I2C protocol includes mechanisms for error detection and recovery, such as NACK (not acknowledge) signals and arbitration. The SoC should implement error-handling routines to handle communication errors gracefully and ensure data integrity.

- **Transaction Sequencing:** Multiple transactions may be required to perform complex operations with the camera module, such as configuring settings or reading image data. The SoC should implement proper transaction sequencing to ensure that operations are performed in the correct order and without errors.

3.2.6 SHA-256 and SRAM

The establishment of the data path between the SHA-256 module and the SRAM is forthe temporary storage of intermediate hash values.An SRAM module with sufficient capacity and bandwidth to accommodate the data processing is required for the SHA-256 algorithm. SRAM offers fast read and write access times, making it suitable for storing temporary data during cryptographic operations.By implementing the memory protection mechanisms to prevent unauthorized access or tampering with sensitive data stored in SRAM.For the Optimized memory access patterns, cache utilization, and data prefetching to maximize throughput and minimize latency.

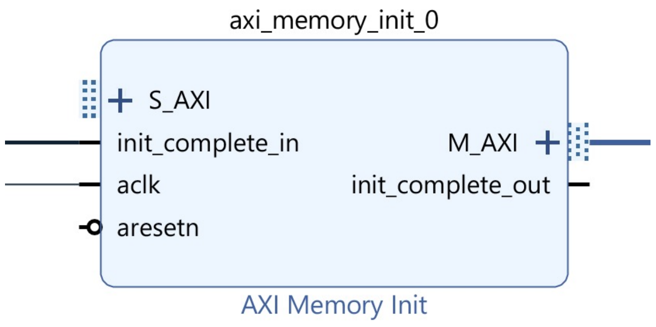


Figure 3.5: SRAM

3.2.7 ALU and Register

The Connection between the Arithmetic Logic Unit (ALU) with registers to facilitate data manipulation and processing within the SoC architecture.To Select an ALU design optimized for the target application's computational requirements, considering factors such as instruction set architecture and supported operations.The ALU performs essential arithmetic and logic operations required for cryptographic calculations, data processing, and system control.By implementing hardware-enforced access control mechanisms to prevent unauthorized access or modification of register contents.To optimize the ALU microarchitecture, instruction scheduling, and pipelining for the improvement of computational efficiency and reduce execution latency.

3.2.8 UART

The configuration of the UART interface to enable serial communication for the transmitting of processed data or status information to external devices.By selecting the UART controller which is compatible with the SoC architecture and the communication protocol requirements of the target devices.UART provides a simple and widely supported serial communication interface suitable for connecting to a variety of external devices, such as microcontrollers, sensors, or display modules.The implementation of the data encryption and integrity verification mechanisms to protect transmitted data from interception, tampering, or spoofing.Optimize UART baud rate, data frame format, and flow control settings for efficient data transmission and minimal overhead.

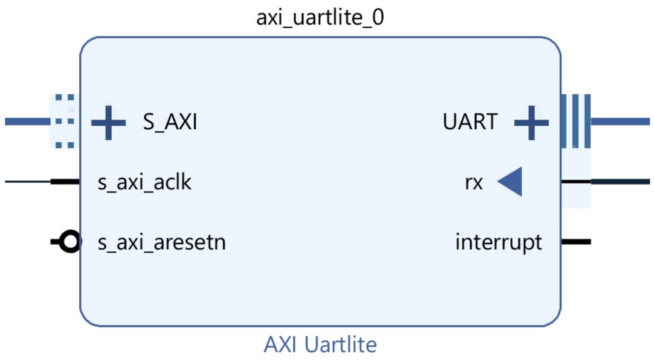


Figure 3.6: UART

3.3 Zynq UltraScale+ MPSoC

The Zynq UltraScale+ MPSoC (Multiprocessor System-on-Chip) is a highly integrated system-on-chip solution developed by Xilinx. It combines the flexibility and programmability of an FPGA (Field-Programmable Gate Array) with the processing power and capabilities of a multicore ARM Cortex-A53 application processor and ARM Cortex-R5 real-time processor.The Zynq UltraScale+ MPSoC offers a powerful and versatile platform for a wide range of embedded applications, combining the flexibility of FPGA technology with the processing capabilities of ARM-based processors. Its rich feature set, high-performance processing cores, and advanced security features make it well-suited for applications requiring real-time processing, high-speed connectivity, and secure operation. Here's a brief explanation of its key features:

- **FPGA Fabric:** The Zynq UltraScale+ MPSoC includes programmable FPGA fabric, allowing users to implement custom hardware accelerators, interfaces, and processing pipelines. This flexibility enables customization and optimization for a wide range of applications, including signal processing, image processing, and machine learning.

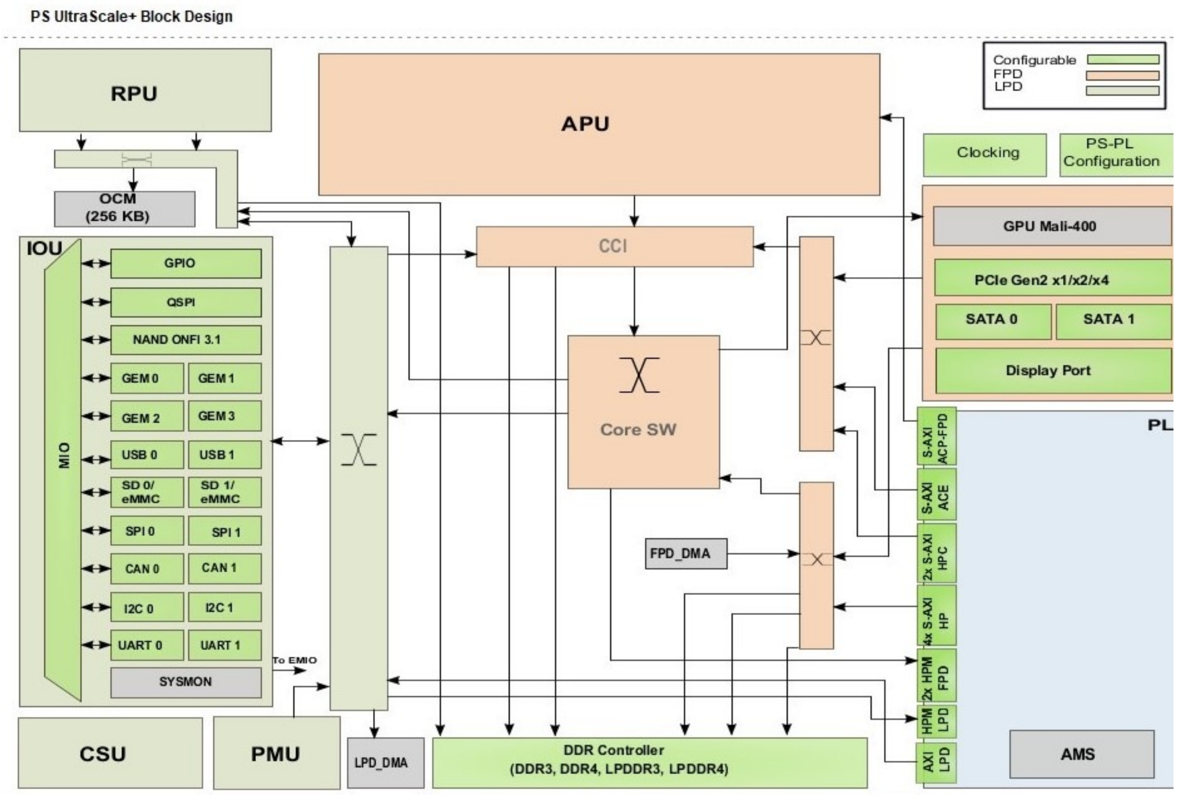


Figure 3.7: Architecture of SOC

- **Application Processing Unit (APU):** The MPSoC features multiple ARM Cortex-A53 application processor cores, which provide high-performance processing capabilities for running operating systems, applications, and middleware. These cores are well-suited for tasks such as user interface, networking, and general-purpose computing.

- **Real-Time Processing Unit (RPU):** In addition to the application processors, the MPSoC includes ARM Cortex-R5 real-time processor cores, optimized for deterministic and low-latency processing tasks. These cores are commonly used for control, monitoring, and safety-critical functions in embedded systems.

- **High-Speed Interfaces:** The MPSoC offers a variety of high-speed interfaces, including PCIe, Gigabit Ethernet, USB 3.0, and DisplayPort, enabling connectivity to external devices, networks, and peripherals. These interfaces support high-bandwidth data transfer and communication in embedded and networking applications.

- **Security Features:** Security is a key focus of the MPSoC, with features such as secure boot, hardware-based encryption, and secure key storage. These features help protect the integrity, confidentiality, and authenticity of data and software running on the device, making it suitable for secure communication, authentication, and cryptographic operations.

- **Advanced Memory Subsystem:** The MPSoC includes a high-performance memory subsystem, featuring DDR4 memory controllers, high-bandwidth on-chip memory (BRAM), and cache memory. This subsystem provides efficient data storage and access for both the programmable logic and processor subsystems, enabling high-speed data processing and manipulation.

- **System-Level Integration:** The MPSoC integrates multiple components and peripherals into a single chip, reducing system complexity, board footprint, and power consumption. This integration simplifies hardware design and enables faster time-to-market for embedded system developers.

3.4 SHA-256 Core

The hardware implementation of the SHA-256 cryptographic hash function is known as the Secure Hash Algorithm 256-bit (SHA-256 core). It is made to quickly calculate the input data's SHA-256 hash, offering strong security and cryptographic integrity.An entirely compliant implementation of the Message Digest Algorithm SHA-256 is the SHA-256 encryption IP core.

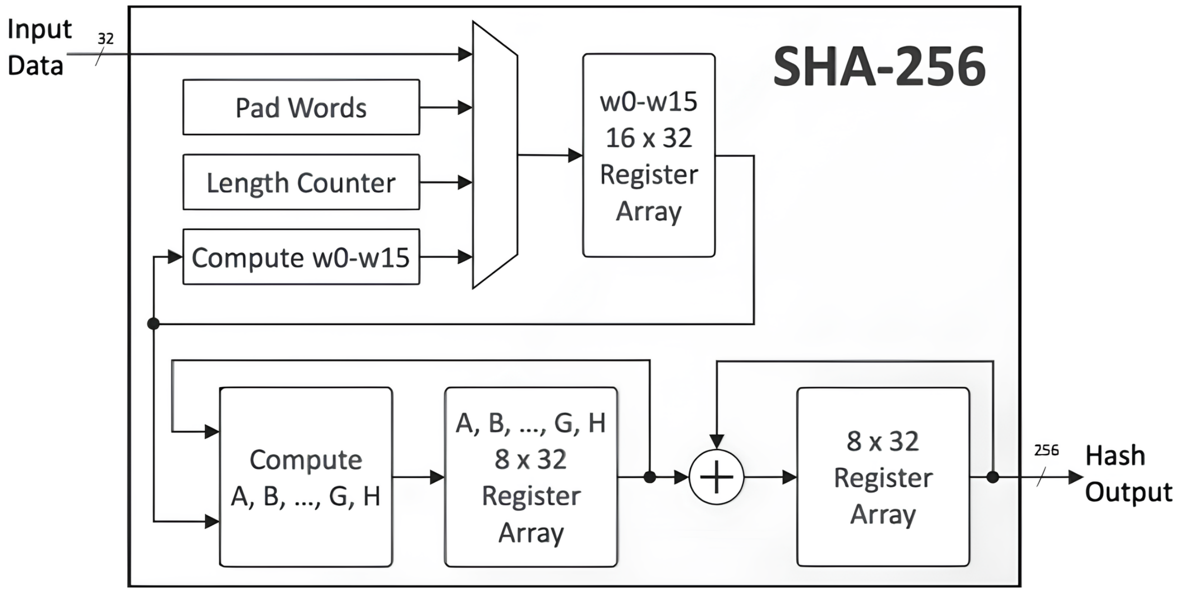


Figure 3.8: SHA-256 Core

- **Hash Function Operation:**The SHA-256 hash function, which accepts an input message of any length and outputs a fixed-size hash value (256 bits) is carried out by the core. Strong cryptographic security is ensured by the hash value's uniqueness to the input data and its inability to be reverse-engineered computationally.

- **Block Processing:** Data is processed by SHA-256 in fixed-size blocks (512 bits). The input message is divided into these blocks by the core, which then processes each block in turn to produce the final hash value. To make sure the input message is a multiple of the block size, padding can be added.
- **Message Expansion:** The core expands messages within each block to produce a set of constant values referred to as "message schedules" or "message words." These values come from the input block and are utilised by the SHA-256 compression function.
- **Compression Function:** The compression function updates the hash state by applying a sequence of bitwise operations, such as logical AND, XOR, and rotations, to the current block of data and hash value (or "state"). Up until the final hash value is calculated, this procedure is repeated for every block of input data, iteratively updating the hash state.
- **Initialization Vector (IV):** The Initialization Vector (IV) is a predefined initial value that the core uses to initialise the hash state. This number guarantees that the hash function begins each input message in the same state.
- **Finalisation:** The core applies finalisation steps to the hash state to generate the final hash value after processing each and every input block. To maintain cryptographic security, this may entail extra steps like padding and appending the input message's length.
- **Output:** The 256-bit hash value derived from the input message is the SHA-256 core's output. By acting as a distinct fingerprint for the input data, this hash value permits secure data authentication and cryptographic integrity verification.

3.5 SHA-256 Verilog Code Implementation

The SHA-256 hashing algorithm is implemented by the Verilog module. A 256-bit(32-byte) hash value, commonly shown as a hexadecimal integer, is generated using the cryptographic hash function SHA-256.

The inputs and outputs:

- **Inputs:** aclk Synchronous operation's clock input.
- **reset_n:** The reset signal is active-low.initial, subsequent, mode: Control signals for the next round, mode selection, and hash initialization.
- **block:** The data to be hash's input block (512 bits).

Final Products:

- **ready:** A signal that the module is prepared to take in data.
- **digest:** The hash value (256 bits) is output.

- **digest_valid:** A string that indicates the validity of the hash value.
- **SHA-224/256 Constants:** The SHA-224 and SHA-256 algorithms make use of constants. Control States: Constants that provide the module's control states (IDLE, ROUNDS, DONE).

Updates and Registers Variables:
Internal state variables and their update counterparts are defined as being stored in registers.The internal state is stored in a_reg to h_reg.The original hash values are stored in H0_reg to H7_reg.State transitions and control signals are handled by other registers.
Module Instances:

- **Sha256_k_constants:** This module is instantiated to produce round constants (k_data) according to the round that is being played.The sha256_mem module is instantiated to oversee the block of messages and produce the w_data for every round.

Logic of Control and Hashing:
The hash computation process is managed by the state machine (sha256_ctrl_reg) in this module.Using the existing control state as a basis, the always @ (*) block modifies the internal state.It manages the hash value's initialization, rounding calculations, and finalisation.
Round Constants:
The current round number (w_round) determines the round constants (k_data).
Results:
The output ports receive the validity signal and the final hash value from the module.
In summary,all of the code needed to execute SHA-256 hashing on input data is encapsulated in the given Verilog module. It manages the hash computing process by providing control signals and adhering to the standard algorithm. For crypto graphic applications, this module can be integrated into bigger digital systems, guaranteeing data security and integrity.
Verilog Code of SHA-256:

```verilog
default_nettype none
module sha256_core(
    input wire clk,
    input wire reset_n,
    input wire init,
    input wire next,
    input wire mode,
    input wire [511:0] block,
    output wire ready,
    output wire [255:0] digest,
    output wire digest_valid
);

    //----------------------------------------------------------------
    // Internal constant and parameter definitions.
    //----------------------------------------------------------------
    localparam [31:0] SHA224_H0_0 = 32'hc1059ed8;
    localparam [31:0] SHA224_H0_1 = 32'h367cd507;
    localparam [31:0] SHA224_H0_2 = 32'h3070dd17;
    localparam [31:0] SHA224_H0_3 = 32'hf70e5939;
    localparam [31:0] SHA224_H0_4 = 32'hffc00b31;
    localparam [31:0] SHA224_H0_5 = 32'h68581511;
    localparam [31:0] SHA224_H0_6 = 32'h64f98fa7;
    localparam [31:0] SHA224_H0_7 = 32'hbefa4fa4;

    localparam [31:0] SHA256_H0_0 = 32'h6a09e667;
    localparam [31:0] SHA256_H0_1 = 32'hbb67ae85;
    localparam [31:0] SHA256_H0_2 = 32'h3c6ef372;
    localparam [31:0] SHA256_H0_3 = 32'ha54ff53a;
    localparam [31:0] SHA256_H0_4 = 32'h510e527f;
    localparam [31:0] SHA256_H0_5 = 32'h9b05688c;
    localparam [31:0] SHA256_H0_6 = 32'h1f83d9ab;
    localparam [31:0] SHA256_H0_7 = 32'h5be0cd19;

    localparam [31:0] SHA256_ROUNDS = 63;

    localparam CTRL_IDLE = 2'b00;
    localparam CTRL_ROUNDS = 2'b01;
    localparam CTRL_DONE = 2'b10;

    //----------------------------------------------------------------
    // Registers including update variables and write enable.
    //----------------------------------------------------------------
    reg [31:0] a_reg, a_new;
    reg [31:0] b_reg, b_new;
    reg [31:0] c_reg, c_new;
    reg [31:0] d_reg, d_new;
    reg [31:0] e_reg, e_new;
    reg [31:0] f_reg, f_new;
    reg [31:0] g_reg, g_new;
    reg [31:0] h_reg, h_new;
    reg a_h_we;

    reg [31:0] H0_reg, H0_new;
    reg [31:0] H1_reg, H1_new;
    reg [31:0] H2_reg, H2_new;
    reg [31:0] H3_reg, H3_new;
    reg [31:0] H4_reg, H4_new;
    reg [31:0] H5_reg, H5_new;
    reg [31:0] H6_reg, H6_new;
    reg [31:0] H7_reg, H7_new;
    reg H_we;

    reg [5:0] t_ctr_reg, t_ctr_new;
    reg t_ctr_we, t_ctr_inc, t_ctr_rst;

    reg digest_valid_reg, digest_valid_new;
    reg digest_valid_we;

    reg [1:0] sha256_ctrl_reg, sha256_ctrl_new;
    reg sha256_ctrl_we;

    //----------------------------------------------------------------
    // Wires.
    //----------------------------------------------------------------
    reg digest_init, digest_update;
    reg state_init, state_update;
    reg first_block;
    reg ready_flag;
    reg [31:0] t1, t2;
    wire [31:0] k_data;
    reg w_init, w_next;
    reg [5:0] w_round;
    wire [31:0] w_data;

    //----------------------------------------------------------------
    // Module instantiations.
    //----------------------------------------------------------------
    sha256_k_constants k_constants_inst(
        .round(t_ctr_reg),
        .K(k_data)
    );

    sha256_w_mem w_mem_inst(
        .clk(clk),
        .reset_n(reset_n),
        .block(block),
        .round(t_ctr_reg),
        .init(w_init),
        .next(w_next),
        .w(w_data)
    );

    //----------------------------------------------------------------
    // Concurrent connectivity for ports etc.
    //----------------------------------------------------------------
    assign ready = ready_flag;
    assign digest = {H0_reg, H1_reg, H2_reg, H3_reg, H4_reg, H5_reg, H6_reg, H7_reg};
    assign digest_valid = digest_valid_reg;

    //----------------------------------------------------------------
    // reg_update
```

```verilog
    // Update functionality for all registers in the core.
    // All registers are positive edge triggered with asynchronous
    // active low reset. All registers have write enable.
    //----------------------------------------------------------------
    always @ (posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            // Reset values
            a_reg <= 32'h0;
            b_reg <= 32'h0;
            c_reg <= 32'h0;
            d_reg <= 32'h0;
            e_reg <= 32'h0;
            f_reg <= 32'h0;
            g_reg <= 32'h0;
            h_reg <= 32'h0;
            H0_reg <= 32'h0;
            H1_reg <= 32'h0;
            H2_reg <= 32'h0;
            H3_reg <= 32'h0;
            H4_reg <= 32'h0;
            H5_reg <= 32'h0;
            H6_reg <= 32'h0;
            H7_reg <= 32'h0;
            digest_valid_reg <= 1'b0;
            t_ctr_reg <= 6'h0;
            sha256_ctrl_reg <= CTRL_IDLE;
        end else begin
            if (a_h_we) begin
                a_reg <= a_new;
                b_reg <= b_new;
                c_reg <= c_new;
                d_reg <= d_new;
                e_reg <= e_new;
                f_reg <= f_new;
                g_reg <= g_new;
                h_reg <= h_new;
            end

            if (H_we) begin
                H0_reg <= H0_new;
                H1_reg <= H1_new;
                H2_reg <= H2_new;
                H3_reg <= H3_new;
                H4_reg <= H4_new;
                H5_reg <= H5_new;
                H6_reg <= H6_new;
                H7_reg <= H7_new;
            end

            if (t_ctr_we) begin
                t_ctr_reg <= t_ctr_new;
            end

            if (digest_valid_we) begin
                digest_valid_reg <= digest_valid_new;
            end

            if (sha256_ctrl_we) begin
                sha256_ctrl_reg <= sha256_ctrl_new;
            end
        end
    end

    //----------------------------------------------------------------
    // Internal hash calculations.
    //----------------------------------------------------------------
    always @ (*) begin
        case (sha256_ctrl_reg)
            CTRL_IDLE: begin
                a_new = H0_reg;
                b_new = H1_reg;
                c_new = H2_reg;
                d_new = H3_reg;
                e_new = H4_reg;
                f_new = H5_reg;
                g_new = H6_reg;
                h_new = H7_reg;
                a_h_we = 1'b1;
                H_we = 1'b0;
                t_ctr_new = 6'h0;
                t_ctr_we = 1'b1;
                digest_valid_new = 1'b0;
                digest_valid_we = 1'b1;
                sha256_ctrl_new = mode ? CTRL_DONE : CTRL_ROUNDS;
                sha256_ctrl_we = 1'b1;
                w_init = 1'b1;
                w_next = 1'b0;
                w_round = 6'h0;
            end
            CTRL_ROUNDS: begin
                if (w_next) begin
                    t_ctr_new = t_ctr_reg + 6'h1;
                    t_ctr_we = 1'b1;
                end
                w_round = t_ctr_reg;
                a_new = e_reg;
                b_new = f_reg;
                c_new = g_reg;
                d_new = h_reg;
                e_new = h_reg + t1;
                f_new = a_reg;
                g_new = b_reg;
                h_new = c_reg;
                a_h_we = 1'b1;
                H_we = 1'b0;
                digest_valid_new = 1'b0;
                digest_valid_we = 1'b1;
                sha256_ctrl_new = (t_ctr_reg == SHA256_ROUNDS) ? CTRL_DONE : CTRL_ROUNDS;
                sha256_ctrl_we = 1'b1;
                w_init = 1'b0;
                w_next = 1'b1;
            end
            CTRL_DONE: begin
                a_new = a_reg;
                b_new = b_reg;
                c_new = c_reg;
                d_new = d_reg;
                e_new = e_reg;
                f_new = f_reg;
                g_new = g_reg;
                h_new = g_reg;
                a_h_we = 1'b1;
                H_we = 1'b1;
                digest_valid_new = 1'b1;
                digest_valid_we = 1'b1;
                sha256_ctrl_new = CTRL_IDLE;
                sha256_ctrl_we = 1'b1;
                w_init = 1'b0;
                w_next = 1'b0;
            end
            default: begin
                a_new = a_reg;
                b_new = b_reg;
                c_new = c_reg;
                d_new = d_reg;
                e_new = e_reg;
                f_new = f_reg;
                g_new = g_reg;
                h_new = h_reg;
                a_h_we = 1'b1;
                H_we = 1'b0;
                t_ctr_new = t_ctr_reg;
                t_ctr_we = 1'b0;
```

```verilog
                digest_valid_new = digest_valid_reg;
                digest_valid_we = 1'b0;
                sha256_ctrl_new = sha256_ctrl_reg;
                sha256_ctrl_we = 1'b0;
                w_init = 1'b0;
                w_next = 1'b0;
            end
        endcase
    end

    always @ (*) begin
        case (w_round)
            0: k_data = 32'h428a2f98;
            1: k_data = 32'h71374491;
            2: k_data = 32'hb5c0fbcf;
            3: k_data = 32'he9b5dba5;
            4: k_data = 32'h3956c25b;
            5: k_data = 32'h59f111f1;
            6: k_data = 32'h923f82a4;
            7: k_data = 32'hab1c5ed5;
            8: k_data = 32'hd807aa98;
            9: k_data = 32'h12835b01;
            10: k_data = 32'h243185be;
            11: k_data = 32'h550c7dc3;
            12: k_data = 32'h72be5d74;
            13: k_data = 32'h80deb1fe;
            14: k_data = 32'h9bdc06a7;
            15: k_data = 32'hc19bf174;
            16: k_data = 32'he49b69c1;
            17: k_data = 32'hefbe4786;
            18: k_data = 32'h0fc19dc6;
            19: k_data = 32'h240ca1cc;
            20: k_data = 32'h2de92c6f;
            21: k_data = 32'h4a7484aa;
            22: k_data = 32'h5cb0a9dc;
            23: k_data = 32'h76f988da;
            24: k_data = 32'h983e5152;
            25: k_data = 32'ha831c66d;
            26: k_data = 32'hb00327c8;
            27: k_data = 32'hbf597fc7;
            28: k_data = 32'hc6e00bf3;
            29: k_data = 32'hd5a79147;
            30: k_data = 32'hd76aa478;
            31: k_data = 32'h98e7c984;
            32: k_data = 32'h11f12baf;
            33: k_data = 32'h17b7be43;
            34: k_data = 32'h1f83d9ab;
            35: k_data = 32'h5be0cd19;
            36: k_data = 32'h6a09e667;
            37: k_data = 32'hbb67ae85;
            38: k_data = 32'h3c6ef372;
            39: k_data = 32'ha54ff53a;
            40: k_data = 32'h510e527f;
            41: k_data = 32'h9b05688c;
            42: k_data = 32'h1f83d9ab;
            43: k_data = 32'h5be0cd19;
            44: k_data = 32'h6a09e667;
             .
             .
             .
            62: k_data = 32'h516b5d36;
            default: k_data = 32'h00000000;
        endcase
    end

    // Assigning outputs
    assign a = a_reg;
    assign b = b_reg;
    assign c = c_reg;
    assign d = d_reg;
    assign e = e_reg;
    assign f = f_reg;
    assign g = g_reg;
    assign h = h_reg;

    assign digest_valid = digest_valid_reg;

    // Hash outputs
    assign digest = {
        a_reg,
        b_reg,
        c_reg,
        d_reg,
        e_reg,
        f_reg,
        g_reg,
        h_reg
    };

endmodule
```

Listing 3.1: SHA-256 Verilog Code

3.5.1 IOT Security Challenges

While there are many advantages to the widespread use of Internet of Things (IoT) devices, there are also serious security risks.IoT devices frequently have limited resources; they don't have the RAM or processing power to support sophisticated security features.Because they are often used in a variety of unregulated and diverse situations, they are susceptible to a wide range of cyberthreats, including virus assaults, unauthorised access, and data breaches.Concerns over data protection and privacy are also raised by the possibility that IoT devices will gather and send sensitive data, such as private and confidential company information.

3.5.2 Importance of Encryption in Securing IoT Data

IoT data security is greatly enhanced by encryption, which prevents unauthorised parties from understanding the data. Since data is frequently transferred via unreliable networks in the context of the Internet of Things, encryption makes guarantee that the data is safe from eavesdropping and manipulation even in the event that it is intercepted. IoT devices may keep sensitive data private and secure by encrypting it both in transit and at rest. This reduces the possibility of unwanted access and data breaches.IoT deployment security is guaranteed to users and stakeholders through encryption, which also makes it easier to comply with data protection laws and standards.

3.5.3 SHA-256 Encryption Algorithm

A cryptographic hash algorithm called SHA-256 (Secure Hash Algorithm 256-bit) takes input data of any length and outputs a fixed-size hash result (256 bits).One of the most used hash functions, it is renowned for being resilient to collision attacks and strong.A sequence of bitwise operations and mathematical transformations are applied to the input data by SHA-256 in order to produce a unique and irreversible hash value.Data integrity and validity may be verified thanks to the produced hash value, which functions as a digital fingerprint of the input data.SHA-256 is frequently used for data integrity verification, digital signatures, and message authentication in the context of Internet of Things security. This provides a basic layer of security for IoT transactions and communications.An arbitrary length of input data is converted into a fixed-size hash value using the cryptographic hash algorithm SHA-256 (Secure Hash Algorithm 256-bit). We will go into the SHA-256 algorithm's complexities in this paper, examining its strengths and features as well as how it creates cryptographic hashes.The National Institute of Standards and Technology (NIST) in the United States and the National Security Agency (NSA) created and released the SHA-2 family of hash functions, which includes the SHA-256 algorithm. It generates a 256-bit hash value by operating on 512-bit data blocks.Multiple rounds of cryptographic operations, such as rotations, bitwise logical operations, and modular addition, make up the algorithm. Using a message schedule, it introduces confusion and diffusion features by permuting and mixing the input data in a non-linear way. This guarantees that little changes in the input cause noticeable changes in the output hash.

3.5.4 SHA-256 Generates Cryptographic Hashes

SHA-256 works by analysing 512-bit chunks of input data, or "message blocks." The input data is first padded by the method to make sure its length is a multiple of 512 bits.The padded message is then divided into blocks, each of which is processed repeatedly to get the final hash result. The following are the primary processes in creating a SHA-256 hash:

- **Message Padding:** To guarantee that the length of the input message is a multiple of 512 bits, padding is applied.
- **Parsing the Message:** The padded message is split up into 512-bit blocks.
- **Initialization:** The initial hash state, or collection of initial hash values, is defined.
- **Compression:** A compression function that combines the message block, a set of constant values, and the current hash value is applied to each message block.
- **Finalisation:** The last state of the compression algorithm is used to determine the final hash value after all message blocks have been processed.

3.5.5 Synthesis Process

Transforming the Register Transfer Level (RTL) description of the design into a gate-level netlist is the synthesis stage in the design flow of a digital circuit. Using registers, combinational logic, and data flow between them, RTL explains how a digital circuit behaves. Since it converts the high-level RTL description into a low-level representation appropriate for hardware implementation, synthesis is an essential stage in the design process.The RTL description, which is normally written in hardware description languages like Verilog or VHDL, is examined at the start of the synthesis process in order to determine the parts, data routes, and control signals.After that, the synthesis tool optimises the logic to reduce the number of gates and enhance the design's functionality. Techniques including resource sharing, technology mapping, and Boolean logic minimization are used in this optimisation.Following optimisation, the synthesis tool produces a gate-level netlist that uses flip-flops and logic gates (such as AND, OR, and NOT gates) to describe the design. The connection and functionality of the design are captured at the gate level by this netlist.The generated netlist is used as input for the design flow's further phases, which include technology mapping, location and route,and timing analysis.Mapping the behavioural constructs and logic primitives described in RTL to their appropriate gate-level representations during synthesis converts the RTL description into a gate-level netlist. Several crucial phases are involved in this transformation:
Parsing: To determine the modules, signals, and control structures specified in the design, the synthesis tool parses the RTL code.
Analysis: To ascertain the relationships and functioning of the design parts, such as registers, combinational logic, and control signals, the tool conducts static analysis.
Optimisation: To simplify the design and enhance performance, logic optimisation methods are used.

This might entail streamlining Boolean formulas, eliminating superfluous logic, and maximising resource use.
Mapping: Depending on the target technology library, the RTL constructions are mapped to the corresponding gate-level parts, such as flip-flops, multiplexers, and logic gates. Netlist Generation: Lastly, the design is represented in terms of coupled logic gates and flip-flops by the synthesised netlist. At the gate level, the structural and functional elements of the design are captured by this netlist.

3.5.6 Methods of Optimisation Used in Synthesis

Synthesis tools use a variety of optimisation strategies to fulfil design limitations and enhance the quality of the synthesised design.

- **Typical optimisation methods include the following:**
  Logic optimisation is the process of minimising the number of gates and logic levels in logic circuits in order to reduce their complexity. Area, power, and performance are optimised in the design using methods including retiming, logic restructuring, and Boolean logic optimisation.
- **Technology Mapping:** In the target technology library, technology mapping associates the RTL constructions with the appropriate gates. It takes time, area, and power restrictions into account while choosing the best gate-level components to implement the design.
- **Resource Sharing:** In order to minimise resource consumption, similar sub circuits or logic structures are found and combined using resource sharing techniques. This contributes to the synthesised design's increased efficiency and area reduction.
- **Timing Optimisation:** These methods make sure that the synthesised design satisfies the necessary timing requirements, such setup and hold durations .In order to minimise signal propagation delays and satisfy timing specifications,this entails modifying the positioning and routing of logic components.

3.5.7 Challenges and Considerations in Synthesis of Complex SoC Architectures

Because of the intricacy and size of the designs, synthesising intricate System-on-Chip (SoC) architectures presents a number of difficulties and problems. Among the principal difficulties are: Complex System-on-a-Chip (SoC) architectures are frequently hierarchical in structure, with several abstraction layers and hierarchical modules. Efficient hierarchical synthesis approaches are necessary for managing complexity and guaranteeing precise synthesis at various stages when synthesising such designs.

- **IP Integration:** Third-party components and Intellectual Property (IP) blocks are frequently included in SoC designs. IP reusability, interface compatibility, and design consistency are all at risk when integrating these IP blocks into the design and making sure they synthesise seamlessly and work with the rest of the design.

- **Timing Closure:** A crucial synthesis difficulty is achieving timing closure, or making sure the design satisfies the necessary temporal constraints. Timing closure gets more difficult to achieve as SoC designs get bigger and more complicated because of the greater design space, intricate interconnects, and timing dependencies.

- **Power Optimisation:** For battery-powered devices and energy-efficient applications in particular, power optimisation is a crucial factor in SoC synthesis. Clock gating, power gating, voltage scaling, and dynamic voltage and frequency scaling (DVFS) are some of the strategies needed to synthesise low-power systems that minimise power usage without sacrificing performance.

Advanced synthesis methods, optimisation strategies, and design approaches built to the unique specifications of intricate SoC architectures must all be used in tandem to address these issues and concerns.The synthesis process, which converts the RTL description into a gate-level netlist that is prepared for implementation, is essential to the electronic circuit design flow. Synthesis tools help designers develop effective, dependable, and high-quality designs for a variety of applications by utilising optimisation techniques and tackling issues that arise from complicated SoC architectures.

3.5.8 Output Analysis of Synthesis

Analysing the synthesised netlist to determine its properties, functionality, and resource use is a crucial step in the design process of digital circuits, known as synthesis output analysis. We will explore the complexities of synthesis output analysis in this part, including time analysis, resource utilisation analysis, possible bottleneck detection, and optimisation areas. We will also evaluate the properties of the synthesised netlist.

3.5.9 Evaluation of Synthesized Netlist Characteristics

The design's gate-level elements and connections are represented by the synthesised netlist. At the gate level, it captures the functional and structural elements of the design. The synthesised netlist is examined during synthesis output analysis in order to evaluate a number of attributes, such as:

- **Size and Complexity:** The scale and complexity of the design are reflected in the size and complexity of the synthesised netlist. Greater complexity and broad functionality may be indicated by larger netlists with a higher number of gates and flip-flops.Multi-level abstraction, modules, and submodules, as well as a hierarchical organisation, might be seen in the netlist. Debugging, modification, and understanding of the design are made easier by hierarchical organisation.

- **Connectivity:** To guarantee appropriate signal routing and connectivity, the relationships between the various components in the netlist are examined. Proper connection is necessary for functional accuracy and performance.

3.5.10 Timing Analysis to Ensure Design Meets Timing Constraints

To make sure the synthesised design satisfies the necessary timing requirements, such as setup and hold times, timing analysis is done. To achieve timing closure, this entails optimising the design and analysing the timing routes in the design to find any breaches.
Important facets of timing analysis consist of:
Identification of Critical pathways: The design's critical pathways that is, the paths with the longest propagation delays are located. These pathways are essential for fulfilling timing requirements since they establish the highest clock frequency that may be achieved.

- **Violating the Setup and Hold Time:** When data signals don't match the necessary timing specifications in relation to the clock signal, timing analysis finds setup and hold time violations. Design optimisations must be used to address violations in order to guarantee the proper operation of the design.

- **Clock Skew and Jitter:** Clock skew and jitter, which refer to variations in the arrival times of clock signals, are analyzed to ensure stable and reliable clock distribution. Minimizing clock skew and jitter helps in achieving consistent timing behavior across the design.

3.5.11 Resource Utilization Analysis

The use of hardware resources in the synthesised design, such as logic gates, flip-flops, memory components, and routing resources, is examined via resource utilisation analysis. This research sheds light on options for optimisation, efficiency, and resource allocation. Important facets of the examination of resource use comprise:

- **Logic Utilisation:** To guarantee effective use of the resources at hand and spot any resource bottlenecks, the use of logic resources—such as logic gates, multiplexers, and combinational logic blocks—is evaluated.

- **Flip-Flop Utilisation:** Flip-flops are used to store state information in sequential circuits. Their location and routing are optimised for better performance by analysing how they are used.

- **Memory Utilisation:** To guarantee sufficient storage capacity and optimise memory allocation for data processing and storage, the use of memory elements, such as registers and memory blocks, is investigated.

- **Routing Utilization:** Routing resources, including routing tracks and interconnects, are evaluated to ensure efficient signal routing and minimize routing congestion, which can degrade performance and increase power consumption.

3.5.12 Identification of Potential Bottlenecks and Areas for Optimization

Potential bottlenecks and places in the synthesised design that need to be optimised are found through synthesis output analysis. This entails checking the design for potential for optimisation as well as performance and resource restrictions. Important elements of identifying and optimising bottlenecks consist of:

- **Performance bottlenecks:** Potential bottlenecks include timing restrictions violations, setup and hold time violations, and critical pathways. Design optimisations are necessary to alleviate these bottlenecks in order to fulfil timing constraints and enhance performance.

- **Resource limitations:** Resource scalability and functionality of designs may be impacted by resource limitations, such as restricted logic or memory resources, which are identified via resource utilisation analysis. In order to reduce resource restrictions and increase resource utilisation, optimisation methods are used.

- **Opportunities for Optimisation:** Synthesis output analysis finds ways to optimise designs in order to save space and power, increase overall performance, and improve design efficiency. Examples of these ways include resource sharing, retiming, and logic restructuring.

To summarise, synthesis output analysis is an extensive assessment procedure that evaluates the properties, functionality, and resource efficiency of the synthesised netlist. Designers may pinpoint areas for optimisation and apply design changes to provide effective, dependable, and high-quality digital circuits for a variety of applications by conducting timing analysis, resource utilisation analysis, and bottleneck identification.

3.5.13 Implementation Process

The architectural design is converted into a synthesised netlist in Vivado so that it may be programmed for an FPGA. The stages involved in implementation are outlined below:

- **Coding:** Verilog HDL is used to convert the architectural design into RTL code. Every module is written in accordance with the design parameters, including the SHA-256 module, I2C interface, SRAM, ALU, and UART in addition to the GPIO 6 interface.

- **Design Entry:** After importing the RTL code into Vivado, it is added to the design hierarchy. Every module is created and linked in compliance with the architectural guidelines.

- **Synthesis:** To create a gate-level netlist, Vivado synthesises the RTL code. Logic optimisation techniques are used during synthesis to reduce the number of gates and boost efficiency.

- **Place and Route:** The FPGA device is used to place and route the synthesised netlist. In this step, physical limitations including power, time, and routing resources are taken into account.

- **Timing Analysis:** Timing analysis is done to make sure that the design works within predetermined clock frequencies and satisfies timing requirements.
- **Bitstream Generation:** A bitstream file containing configuration information for FPGA programming is created when the design passes timing analysis.
- **Verification and Debugging:** To guarantee proper functionality and fix any problems or faults, the design is checked and debugged.

One of the most important tasks in addressing the rising cybersecurity risks in IoT deployments is the design and study of SoC architecture for protected IoT devices. The suggested architecture guarantees data confidentiality, integrity, and authenticity by using SHA-256 encryption and creating secure communication routes. The Vivado implementation process makes the architectural concept a reality and offers a dependable and strong solution for IoT device security across a range of industries and applications.

3.5.14 XDC File

The Xilinx Design Constraints (XDC) file plays a crucial role in the implementation of the SoC architecture on the Zynq UltraScale+ MPSoC board. It contains constraints and directives that guide the Place and Route process, ensuring proper placement and connectivity of components. In this project, the XDC file specifies:

- **Pin Assignments:** Each GPIO port, I2C interface, SRAM, ALU, and UART interface is assigned to specific physical pins on the FPGA device. This ensures proper connectivity between the SoC components and the FPGA pins.
- **Timing Constraints:** Timing constraints such as clock frequency, setup time, and hold time are defined to meet the design's timing requirements. These constraints ensure that signals propagate correctly through the design and meet the required timing parameters.

3.5.15 Bit Stream Generation

The bitstream must be generated when synthesis and implementation are finished. The "Generate Bitstream" command or the graphical user interface (GUI) in Vivado are usually used for this. There are many crucial processes in the bitstream generating process:

- **Design Check:** Vivado conducts a number of design checks to make sure the design satisfies all requirements and is prepared for implementation before generating the bitstream.
- **Synthesis Output:** Vivado creates a bitstream file containing FPGA configuration data based on the synthesised netlist and implementation results.
- **Place and Route:** To complete the physical arrangement of logic units and signal routing on the FPGA, Vivado carries out place and route optimisations during bitstream creation.

- **Configuration File Generation:** The final step of bitstream generation is the creation of the bitstream file, which contains binary data representing the configuration settings for the FPGA. This file is used to program the FPGA and configure its internal resources accordingly.

3.5.16 Analysis and Verification in FPGA Design

Analysis and verification are an essential next phase in the FPGA design process, after bitstream generation. This stage attempts to guarantee that the implemented design performs as anticipated and satisfies the required specifications. To confirm the functionality, performance, and dependability of the design, analysis and verification comprise a variety of tasks including debugging, timing analysis, and simulation. We will explore the complexities of analysis and verification in FPGA design in this study, along with the methods, resources, and best practices used to guarantee the design's functional integrity.

3.5.17 Simulation-Based Verification

A key method for functional verification in FPGA design is simulation. It entails modelling the FPGA design and modelling how it will behave in various test situations. Through simulation-based verification, designers may confirm that the implementation is accurate, find defects, and check the design's functionality.

- **RTL Simulation:** Using programmes like ModelSim or Questa, RTL (Register Transfer Level) simulation is carried out early in the design phase. By confirming the behaviour of the design at the register transfer level, RTL simulation enables designers to confirm the accuracy of the design logic and the functioning of specific modules.

- **Gate-Level Simulation:** To confirm the behaviour of the design at the gate level, gate-level simulation is carried out following synthesis and implementation. Timing effects, delays, and interactions between various logic units and signals in the design are all taken into consideration via gate-level modelling.

- **Testbench Development:** For efficient simulation-based verification, it is essential to create thorough testbenches. In order to confirm that the design is accurate under various operating settings and corner cases, test benches produce stimuli for the design, apply input vectors, and track the output responses.

3.5.18 Timing Analysis

In high-speed and time-critical systems in particular, timing analysis is an additional crucial component of verification. Timing analysis evaluates the clock frequency, setup and hold durations, and signal propagation delays of the design in order to make sure that timing requirements are met and the system functions as intended.

- **Static Timing Analysis (STA):** STA analysis is done statically, without modelling the circuit's real operation, to examine the timing behaviour of the design. Timing anomalies, such as clock skew, setup and hold time violations, and signal integrity problems, are detected by STA, which also forecasts the worst-case time scenarios.

3.5.19 Validation and Debugging

A crucial component of verification is debugging, which finds and fixes problems and mistakes in the design's execution. Debugging is the process of identifying the core causes of problems and confirming the validity of the design by examining simulation results, waveform traces, and timing reports.

- **Waveform Debugging:** In order to comprehend how signals and design modules behave, simulated waveforms are analysed. To find anomalies, mistakes, or unexpected behaviours, designers examine timing relationships, waveform traces, and signal transitions.

- **Post-synthesis and Post-Implementation Debugging:** These two techniques are used to find problems unique to the synthesised or implemented design after the synthesis and implementation phases.To identify implementation-related issues, designers examine synthesis and implementation reports, such as resource utilisation summaries, place and route reports, and synthesis logs issues and optimize the design accordingly.

Model checking is a formal verification technique used to confirm that a design model satisfies given attributes or criteria by thoroughly exploring all potential states and transitions of the model. Model checking tools, such Synopsys Formality or Cadence JasperGold, examine the design model and confirm that its behaviour adheres to the required characteristics. Equivalency verification verifies that two design representations—such as synthesised netlist and RTL—have functional equivalency by comparing them. Tools for evaluating equivalency, such Cadence Conformal or Synopsys Formality, confirm that the synthesised implementation accurately reflects the original design intent as stated in the RTL description.In order to guarantee the functional integrity and dependability of FPGA designs, analysis and verification are essential. Debugging, timing analysis, formal verification, and simulation-based verification are some of the methods used to confirm that the design is right, performs as intended, and complies with the requirements. Through a combination of these methods, the design may be extensively verified, giving designers the confidence to deploy FPGA-based systems that adhere to strict quality and reliability criteria. In the end, the thorough analysis and verification procedure helps to successfully deploy FPGA designs in real-world applications by lowering design iterations and mitigating risks.

3.5.20 Implementation of Vivado Design into the Zynq Board

- **Setting Up the Zynq Development Environment:** Before proceeding with implementation, it is essential to set up the Zynq development environment. This involves configuring the necessary hardware, software, and toolchain components required for programming and debugging the Zynq board.

Key steps in setting up the development environment include: Installing the necessary drivers, software tools (e.g., Xilinx Vivado, Xilinx SDK), and development packages.Connecting the Zynq board to the host computer via USB, Ethernet, or other interfaces.Configuring the Zynq board for programming and debugging, including setting up the JTAG interface for communication with the FPGA and PS components.

- **Generating the Hardware Description File (HDF):** Before implementing the design on the Zynq board, it is necessary to generate the Hardware Description File (HDF) in Vivado. The HDF contains detailed information about the hardware configuration, including the FPGA fabric, PS components (e.g., processor cores, memory controllers), and peripheral interfaces (e.g., GPIO, UART, I2C). Generating the HDF involves exporting the hardware design from Vivado and packaging it into a format compatible with the Zynq development environment.

- **Integrating the Design with the Zynq PS:** The next step is to integrate the synthesized design with the Zynq PS components. This involves configuring the PS components, including the processor cores, memory controllers, and peripheral interfaces, to interact with the FPGA fabric and external devices.

Key tasks in this process include: Configuring the Zynq PS using the Vivado IP Integrator tool or Xilinx SDK. Configuring memory mappings, interrupt controllers, and peripheral interfaces to enable communication between the PS and FPGA fabric.Configuring clocking resources and reset signals to ensure proper synchronization and initialization of the FPGA design.

- **Configuring the FPGA Fabric:** Once the Zynq PS configuration is complete, the next step is to configure the FPGA fabric with the synthesized design. This involves programming the FPGA with the generated bitstream file, which contains the configuration settings for the FPGA fabric.

Key steps in this process include: Loading the bitstream onto the Zynq board using the Xilinx Vivado or SDK tools.Verifying the successful configuration of the FPGA fabric using status indicators or diagnostic tools provided by the Zynq development environment. Ensuring proper initialization and operation of the FPGA design within the context of the Zynq SoC environment.

- **Testing and Debugging:** Once the FPGA design has been implemented on the Zynq board, it is essential to perform testing and debugging to verify its functionality and performance. This involves running test cases, monitoring signals, and analyzing results to ensure that the design operates as expected.

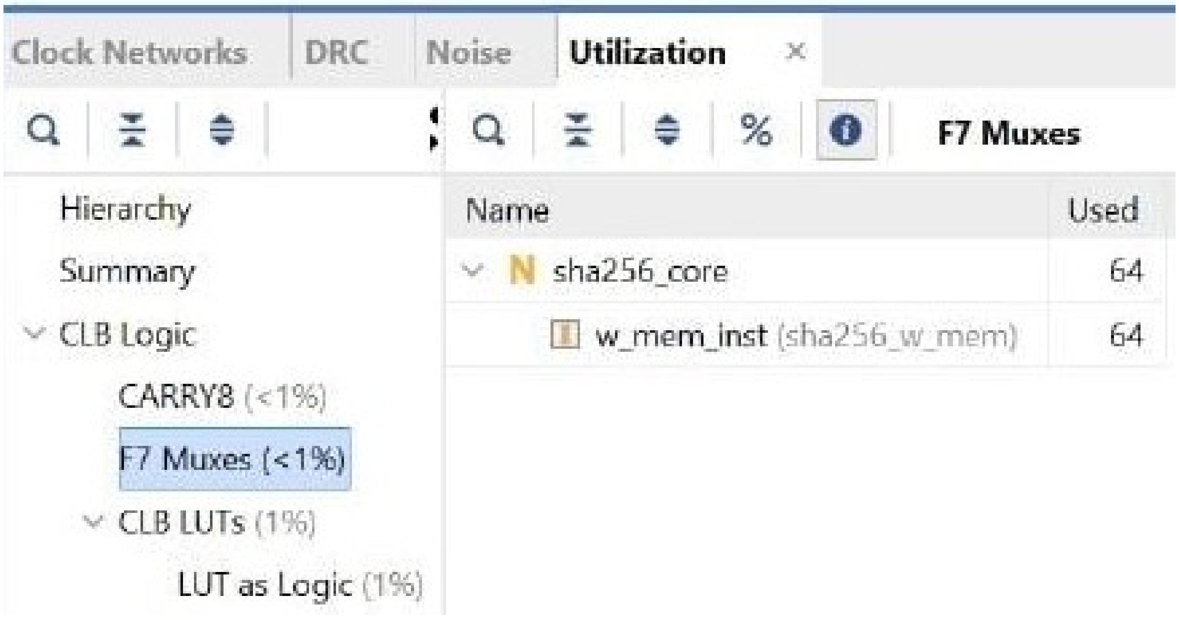
Key tasks in testing and debugging include: Writing testbench code or applications to validate the functionality of the FPGA design.Debugging any issues or errors encountered during testing, such as timing violations, interface mismatches, or functional errors.

CHAPTER 4

RESULTS AND ANALYSIS

4.1 Simulation Results

The designing and architecture involves planning and structuring the SOC, determining the arrangement and interaction of its components.Once this phase is completed, specific results or outcomes are obtained, possibly related to the finalized design, connectivity, or other design-specific aspects.Synthesis, in the context of chip design, refers to the process of transforming a high-level hardware description into a lower-level representation suitable for manufacturing.After the synthesis stage is completed, additional results are obtained, likely pertaining to the translated design, optimizations, or other characteristics relevant to the synthesized version of the SOC.



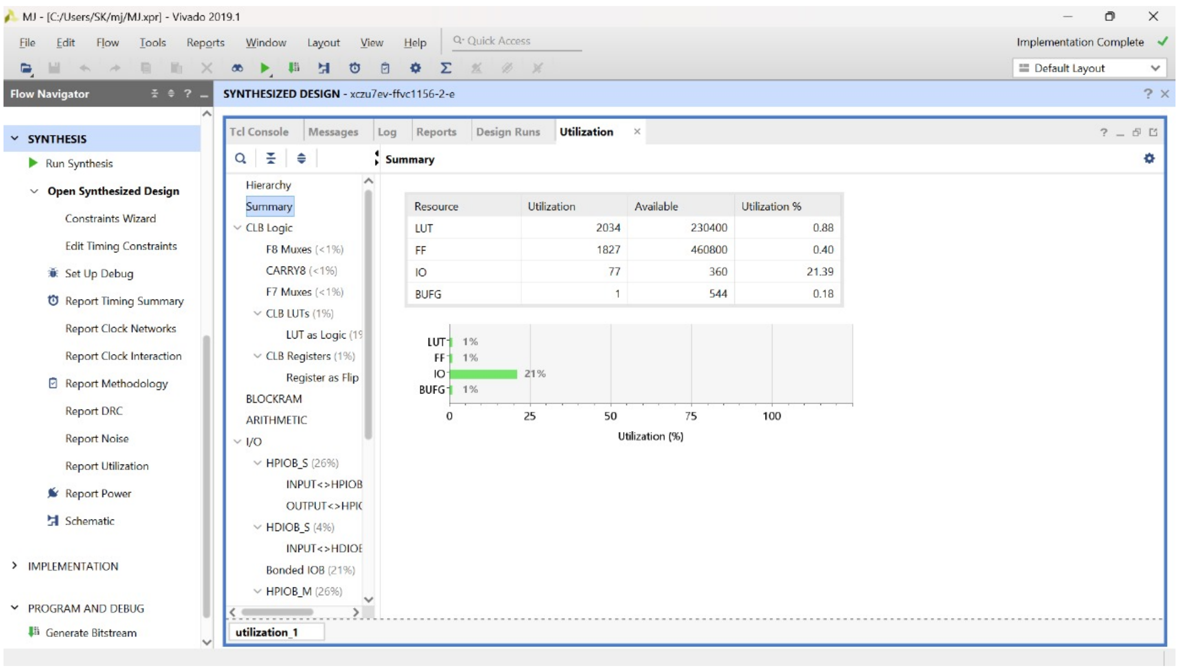
Clock Networks | DRC | Noise | Utilization

Hierarchy
Summary
CLB Logic
CARRY8 (<1%)
F7 Muxes (<1%)
CLB LUTs (1%)
LUT as Logic (1%)

F7 Muxes

Name	Used
sha256_core	64
w_mem_inst (sha256_w_mem)	64

Figure 4.1: Utilization of SHA-256



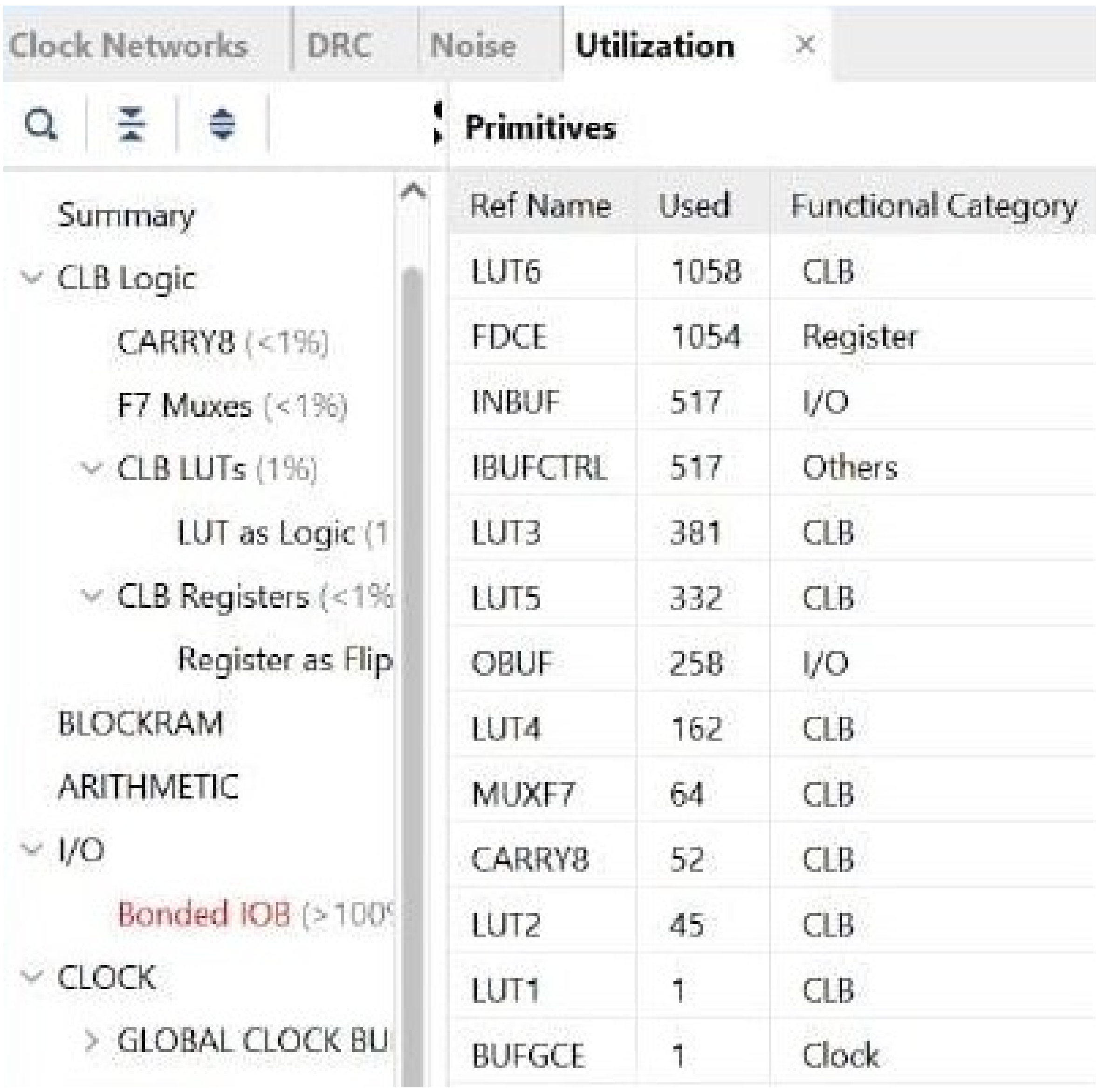
Resource	Utilization	Available	Utilization %
LUT	2034	230400	0.88
FF	1827	460800	0.40
IO	77	360	21.39
BUFG	1	544	0.18

Figure 4.2: Utilization of the Design [%]

- **Utilization Of SHA-256:** The implementation of SHA-256 and provides insights into resource allocation, including LUTs, FFs, IOs, and BUFGs. It details the utilization of these resources, as well as the total number available. Additionally, it outlines the percentage of resource utilization, offering a comprehensive understanding of the efficiency and capacity of the SHA-256 implementation.

4.2 Implementation Results

- **Utilization of Primitives:** The utilization of primitives, detailing the specific number employed and categorizing them based on their functional category.The various functional categories crucial to the architecture's operation. Among these categories are CLB, which stands for Configurable Logic Block. CLB architecture is noted for its broader functionality and reduced logic levels, attributes that contribute to heightened performance levels within the system.Another functional category is the register. Registers are assemblies of flip-flops designed to store binary data patterns. They play a pivotal role in temporarily holding information within the system, facilitating seamless data processing and manipulation.I/O, which stands for Input/Output. I/O components are essential for facilitating communication between the system and its external environment, enabling data exchange and interaction with peripheral devices.Clock functionality is also highlighted in the passage. Clocks serve as temporal references within the Vivado Design Suite, ensuring the reliable and synchronized transfer of data between registers. This synchronization is critical for maintaining data integrity and system stability during operation.The existence of other functional categories beyond those explicitly mentioned. These categories



Primitives

Ref Name	Used	Functional Category
LUT6	1058	CLB
FDCE	1054	Register
INBUF	517	I/O
IBUFCTRL	517	Others
LUT3	381	CLB
LUT5	332	CLB
OBUF	258	I/O
LUT4	162	CLB
MUXF7	64	CLB
CARRY8	52	CLB
LUT2	45	CLB
LUT1	1	CLB
BUFGCE	1	Clock

Figure 4.3: Primitives

likely encompass a range of auxiliary components and functionalities essential for the overall operation and performance optimization of the system.

- **Percentage of Utilization:** In FPGA (Field-Programmable Gate Array) design, the efficient utilization of resources is crucial for achieving optimal performance and functionality. Various resources, including Look-Up Tables (LUTs), Flip Flops (FFs), Input/Output pins (IOs), and Buffers (BUFGs), play essential roles in implementing the desired functionality of the design.The usage of various resources utilized in the context including of LUTs (LookUp Tables), FFs (flip flops), IOs (Input/Output), and BUFGs (Buffer).LUTs are fundamental building blocks in FPGA architecture, used to implement combinatorial logic functions. They are configurable memory units that store the truth table values of logic functions.LUTs enable the implemen-

Table 4.1: Utilization Hierarchy(SHA_256_Core)

S.NO	Ports	Utilization
1.	CLB LUTs(230400)	926
2.	CLB Registers(460800)	530
3.	CARRY 8(28800)	48
4.	CLB(28800)	135
5.	LUT as Logic(230400)	926
6.	Block RAM Tile(312)	530
7.	HPIOB_M(144)	263
8.	HPIOB_S(144)	120
9.	HDIOB_M(24)	120
10.	HDIOB_S(24)	4
11.	HPIOB_SNGL(24)	3
12.	HPIOBDIFFBUF(192)	16
13.	PLL(16)	1

tation of complex logic functions by storing precomputed outputs for all possible combinations of inputs. Efficient utilization of LUTs is essential for minimizing logic depth and maximizing design performance.IO pins facilitate communication between the FPGA device and its external environment, such as sensors, actuators, and other peripherals. They serve as interfaces for inputting external signals into the FPGA and outputting processed data or control signals to external devices. IOs play a critical role in system integration and interfacing with external components, influencing system performance and functionality.BUFGs are specialized buffers designed to distribute clock signals with low skew and jitter. They ensure consistent and reliable clock distribution across the FPGA device, essential for synchronous operation and timing closure. BUFGs are typically used in clock networks and high fanout scenarios to maintain signal integrity and minimize clock skew. Effective placement and utilization of BUFGs are crucial for meeting timing constraints and achieving reliable system performance.

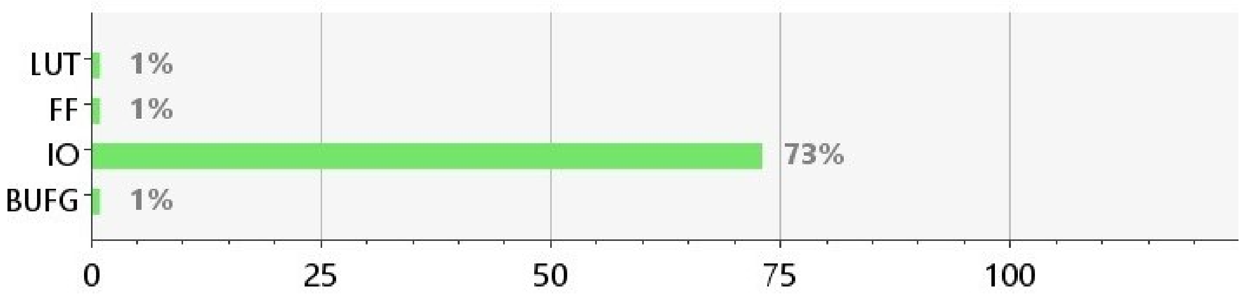


Figure 4.4: Utilization of Zynq Board [%]

Table 4.2: Utilization(%)

Resource	**Utilization**	**Available**	**Utilization**[%]
LUT	926	230400	0.40
FF	530	460800	0.12
IO	263	360	73.06
BUFG	1	544	0.18

- **ZYNQ-104:** The ZCU104 Evaluation Kit offers designers a head start in developing embedded vision applications, spanning areas like surveillance, Advanced Driver-Assisted Systems (ADAS), machine vision, Augmented Reality (AR), drones, and medical imaging.The ZU7EV device included boasts a quad-core ARM Cortex-A53 applications processor, a dual-core Cortex-R5 real-time processor, a Mali-400 MP2 graphics processing unit, a 4KP60 capable H.264/H.265 video codec, and 16nm FinFET+ programmable logic.The successful implementation of the design, which is then loaded onto the ZYNQ-104 hardware board, with the implementation completed successfully.The two primary parts of the Zynq-104 architecture are the Programmable Logic (PL) and the Processing System (PS). Input/output peripherals, memory controllers, ARM Cortex-A9 CPUs, and other system-level parts are all included in the PS. The FPGA fabric, which is part of the PL, can be programmed to create unique logic circuits and interfaces that are suited to the needs of particular applications.Numerous embedded system applications, such as industrial automation, automotive electronics, telecommunications, image processing, and signal processing, are appropriate for the Zynq-104 board. It is an excellent choice for tasks

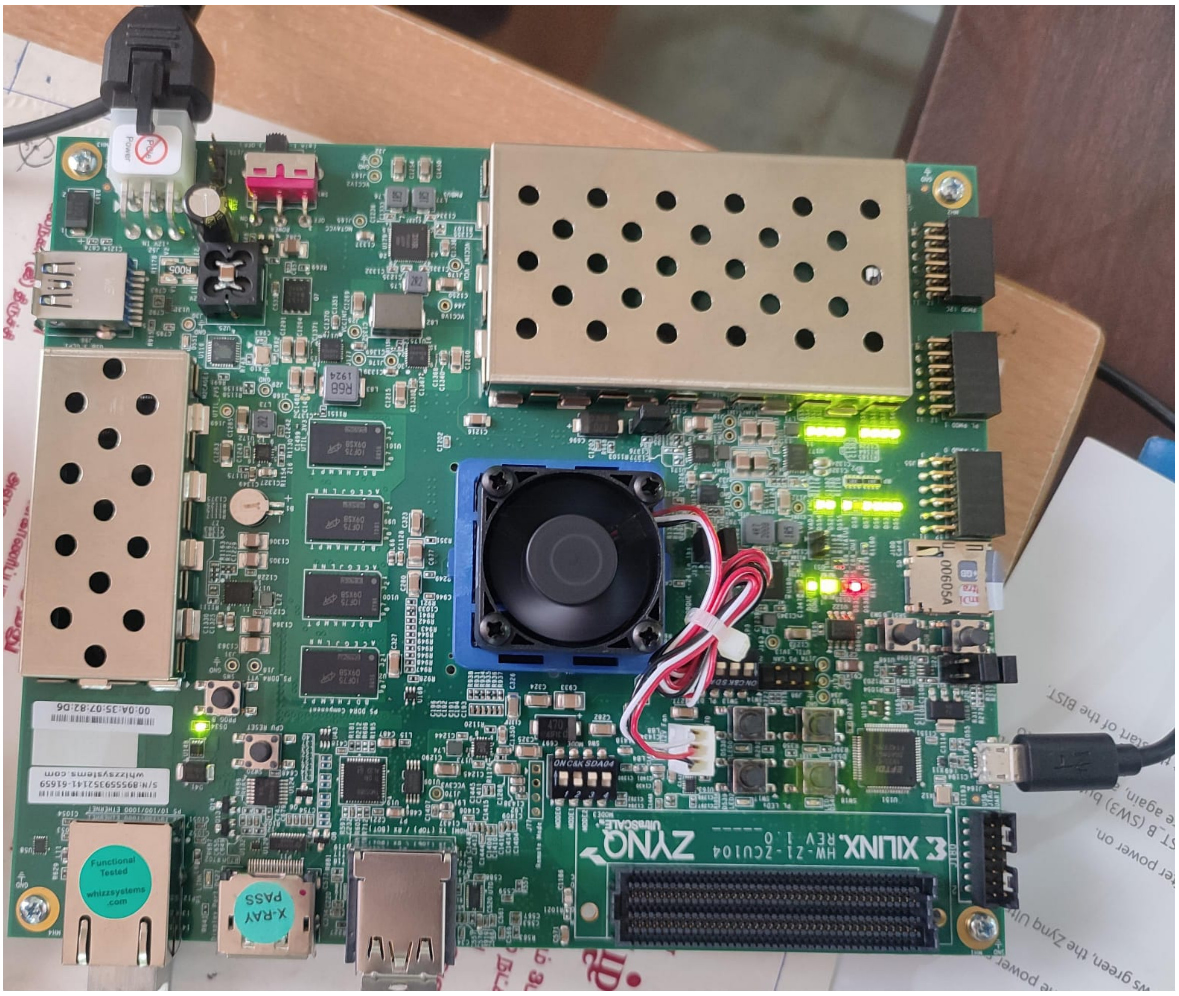


Figure 4.5: ZYNQ-104 Board

requiring a lot of computation and real-time processing because of its ability to combine ARM processors with FPGA fabric, which provides flexibility, performance, and scalability.

- **On-Chip Power:**On-chip power refers to the power consumption and management within an integrated circuit (IC) or semiconductor chip. It encompasses the electrical energy consumed by the various components and subsystems on the chip during operation, as well as the techniques employed to manage and optimize power consumption.On-chip power consumption arises from various sources, including logic gates, memory elements (such as flip-flops and registers), interconnects, input/output (I/O) buffers, and other functional units.Dynamic power consumption occurs when transistors switch states, resulting in energy dissipation due to charging and discharging of capacitances within the circuit.Static power consumption, also known as leakage power, occurs even when the circuit is idle, primarily due to subthreshold leakage currents in transistors and other leakage mechanisms.Additionally, power consumption may vary based on factors such as clock frequency, data activity, temperature, and process variations.Power management techniques are employed to mitigate excessive power consumption and optimize energy

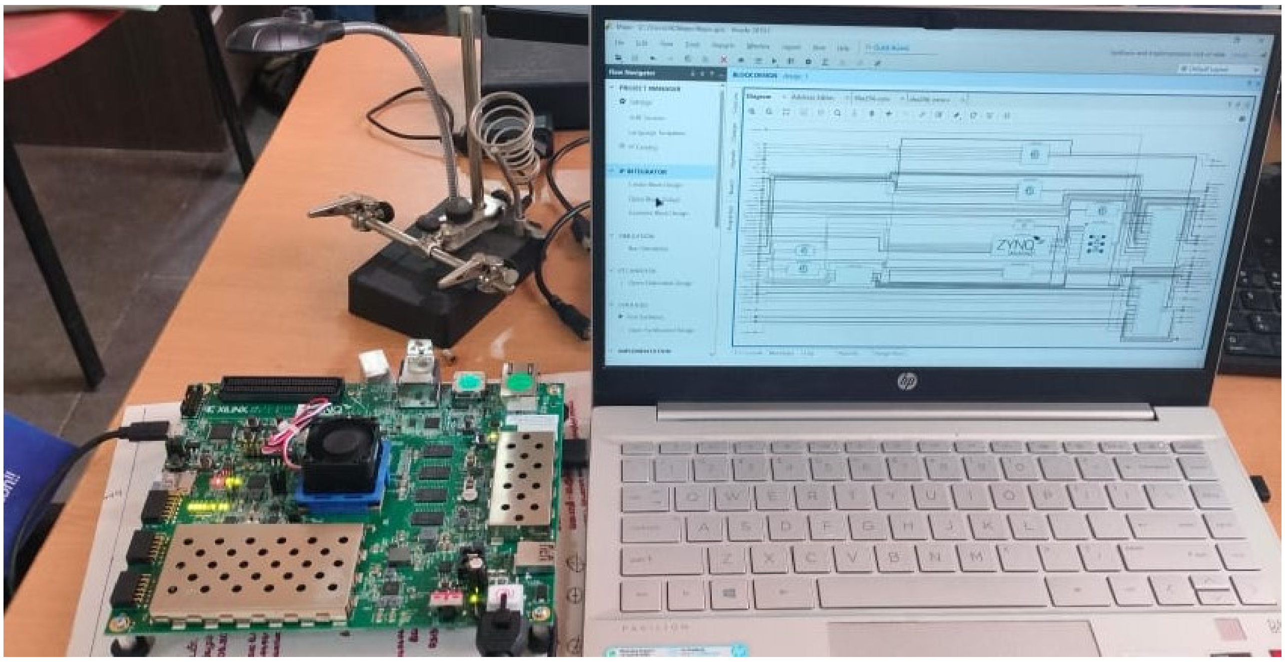
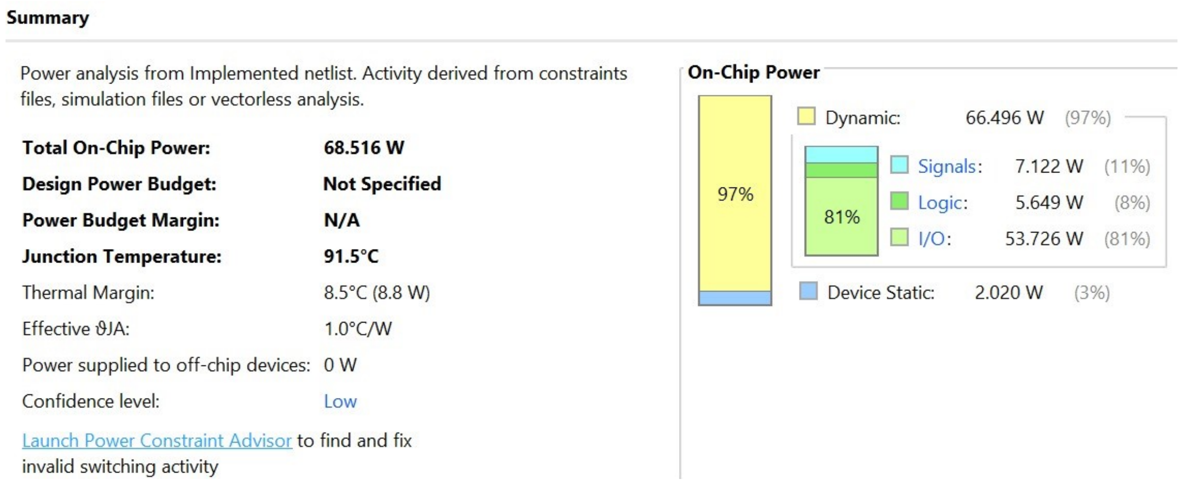


Figure 4.6: Implementation On ZYNQ-104

efficiency in on-chip designs.Dynamic Voltage and Frequency Scaling (DVFS) adjusts the operating voltage and clock frequency dynamically based on workload requirements, reducing power consumption during low activity periods.Clock gating selectively disables clock signals to idle or unused circuit blocks, reducing dynamic power consumption by preventing unnecessary switching activity.Power gating involves shutting down power to inactive circuit blocks or entire functional units to minimize static power consumption.Multi-Voltage and Adaptive Voltage Scaling (AVS) techniques adjust supply voltages dynamically based on workload and environmental conditions to optimize power efficiency.

- **Power Supply Analysis:**Designing an FPGA (Field-Programmable Gate Array) project, understanding the power supply source, voltage, and different power consumption metrics is crucial for optimizing the design's power efficiency.The external power source or power rails that supply electricity to the FPGA device are referred to as the power supply source.The usage of the constraints file (.xdc) or the GUI interface in Vivado to specify the voltage and source of power for the FPGA device. The voltage level, which is commonly expressed in volts (V), varies depending on the particular FPGA device being used. For instance, 1.0V or 1.2V is a typical voltage level for FPGA devices. Total power (A), which is commonly expressed in amperes (A) or milliamperes (mA), denotes the FPGA device's overall power consumption.Total power consumption in Vivado is calculated or measured based on a number of variables, including clock frequencies, routing congestion, and the logic utilisation of the design. The total power (A) is the result of adding the consumption of dynamic power (A) and static power (A).The power used by the FPGA device as a result of switching activity and signal transitions within the logic elements



Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**68.516 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**91.5°C**
Thermal Margin:	8.5°C (8.8 W)
Effective ϑJA:	1.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	66.496 W	(97%)
Signals:	7.122 W	(11%)
Logic:	5.649 W	(8%)
I/O:	53.726 W	(81%)
Device Static:	2.020 W	(3%)

Figure 4.7: On-Chip Power of Zynq Board

is referred to as dynamic power (A). The activity factor, clock frequencies, logic utilisation, and capacitance loading of the design are all taken into consideration during the dynamic power (A) estimation process.Higher clock frequencies, higher logic utilisation, and higher design activity all result in higher dynamic power (A) consumption.Leakage power, sometimes referred to as static power (A), is the amount of power used by the FPGA device even when it is idle or in a static state.Static power (A) estimation in Vivado takes into account various leakage mechanisms within the FPGA device as well as leakage currents in transistors.Temperature, voltage, transistor threshold voltages, and process technology are some of the variables that impact static power (A) consumption.

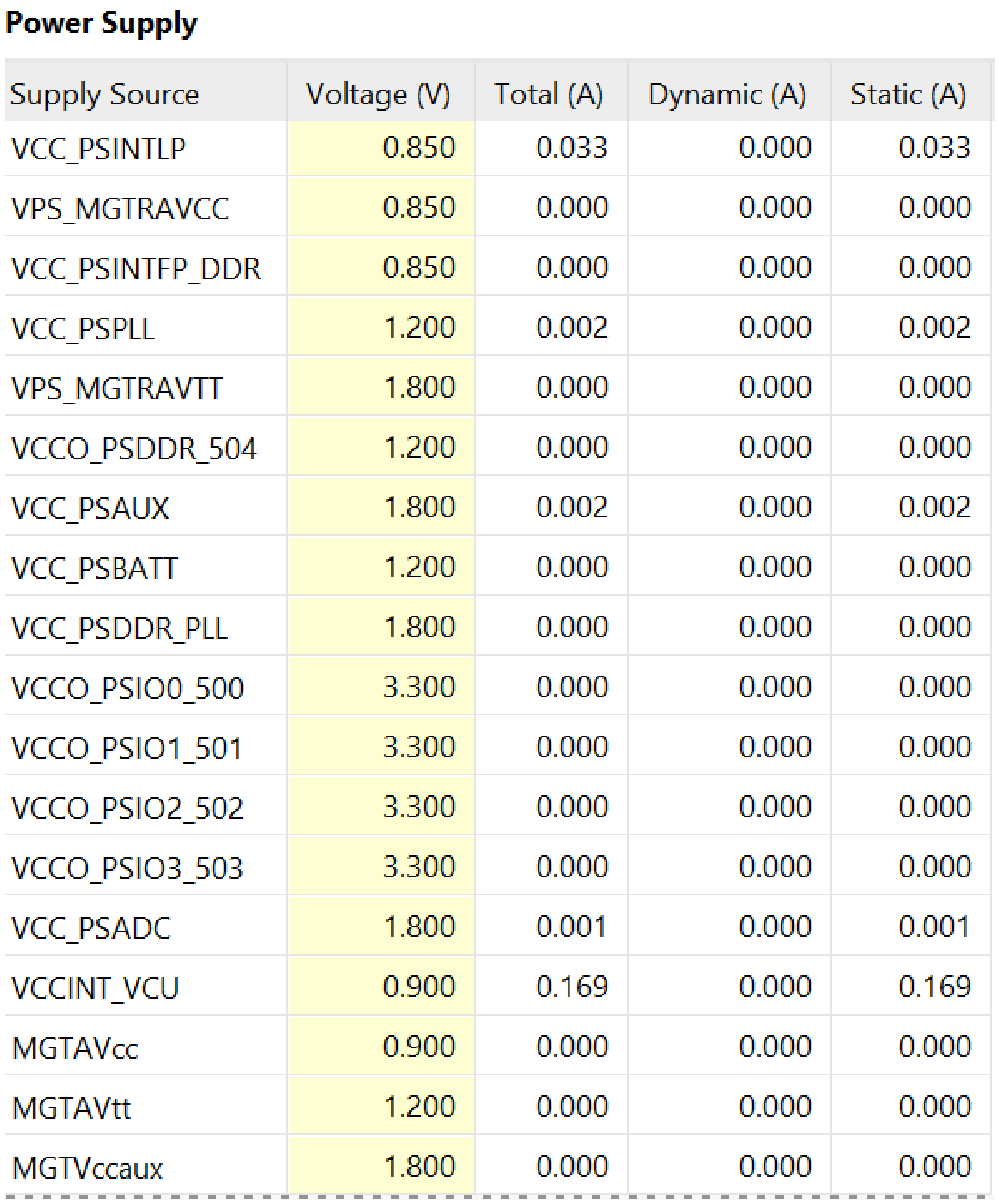
- **Utilization Of SHA-256:**Optimising design performance and resource usage on FPGA (Field-Programmable Gate Array) devices requires a thorough understanding of utilisation metrics, clocking, fanout, slice fanout, and signal rates when implementing cryptographic algorithms like SHA-256 (Secure Hash Algorithm 256-bit).The percentage of available resources (such as flip-flops, DSP slices, BRAM, and LUTs) that are used by the design is referred to as utilisation.Reports on resource utilisation, such as LUT, FF, DSP, and BRAM (block RAM) utilisation, offer comprehensive analyses of how resources are currently being used.Utilisation metrics analysis aids designers in determining possible bottlenecks or resource constraints as well as the effectiveness of their design implementation.The frequency at which signals travel through the FPGA design is referred to as the signal rate.Signal rates in cryptographic algorithms depend on clock frequencies, data dependencies, critical paths, and routing delays, among other things.Through the examination of the design's critical paths and timing constraints, timing analysis tools offer insights into signal rates. Clock frequencies, pipeline stages, and data dependencies can all be adjusted to optimise signal rates in order to meet timing requirements and meet performance goals.The number of logic elements that are driven by a single output signal is referred to as fanout.Fanout analysis aids in the identification of high fanout signals that could result in timing violations, signal skew, or routing congestion by designers.In order to guarantee dependable signal propagation and timing closure, fanout management effectively entails balancing signal loads, optimising routing resources, and minimising signal delays.

- **Utilization of Pins:**The assignment and use of input/output (I/O) pins on the FPGA device is referred to as pin utilisation.Constraint files (.xdc) are used by designers in Vivado to specify pin assignments to particular signals, interfaces, or peripherals.Vivado's pin utilisation reports offer information on the distribution of pins among various peripherals and interfaces, as well as the percentage of available I/O pins that are utilised. In order to guarantee correct connectivity, interface compatibility, and signal integrity in their FPGA designs, designers can benefit from analysing pin utilisation.The term "signal utilisation" (w) describes how internal signal wires are used in the FPGA design.The quantity and percentage of signal wires used to connect registers, logic components, and I/O interfaces are detailed in signal utilisation reports.Design professionals can evaluate routing congestion, signal fanout, and timing constraints by analysing signal utilisation.The term "data utilisation" (w) describes how data buses or paths are used in an FPGA design. Data buses that are used to transfer data between memory components, I/O interfaces, and functional units can be counted and their percentage is revealed by data utilisation reports.Data paths can be optimised, bottlenecks can be reduced, and effective data transfer rates can be achieved in designs with the aid of data utilisation analysis.The use of clock enable signals in the FPGA design is denoted by clock enable utilisation (w).Signals known as clock enables govern when particular clocked elements, like multiplexers, registers, and counters, are activated or deactivated. Clock enable utilisation reports give details on how many and what proportion of clock enable signals are used, as well as how they are distributed throughout the design. Clock enable utilisation analysis aids in the optimisation of clock gating, power reduction, and timing performance in designs. The term "logic utilisation" (w) describes how logic resources, like flip-flops and Look-Up Tables (LUTs), are used in FPGA designs. The quantity and percentage of LUTs and flip-flops used to implement sequential and combinatorial logic functions can be found in logic utilisation reports.

- **Comparision:**The table presents a comparison between the proposed model and Boyou Zhou's model, detailing parameters including average I/O utilization (W), SHA-256 utilization (W), logic utilization (W), data utilization (W), encryption time utilization (Sec), and power consumption utilization (W). This comparison enables us to discern the differences in parameters and their respective utilization across the models.The average input/output (I/O) resource consumption of the system, usually expressed in watts (W). I/O utilisation measures how well a system makes use of its input/output capabilities, including data transfer and communication with external devices.The typical amount of resources used for the SHA-256 algorithm, which is commonly expressed in watts (W). A cryptographic hash function called SHA-256 is used to measure how much power is used when SHA-256 operations are carried out.The system's average logical resource usage, expressed in watts (W). The power used by the system's components to perform logic operations like arithmetic and Boolean operations is measured by logic utilisation.The system's average consumption of its data processing power, which is commonly expressed in watts (W). The amount of power used by the system for data manipulation, storage, and retrieval is reflected in data utilisation.The average time in seconds (Sec) that the system takes to finish encryption operations. The analysis of encryption time utilisation sheds light on the effectiveness of encryption procedures and how they affect system performance.The total system's average power consumption expressed in watts (W). Power consumption utilisation is a measure of the system's total energy consumption across all of its parts and functions.

Table 4.3: Comparision Of Paramaters

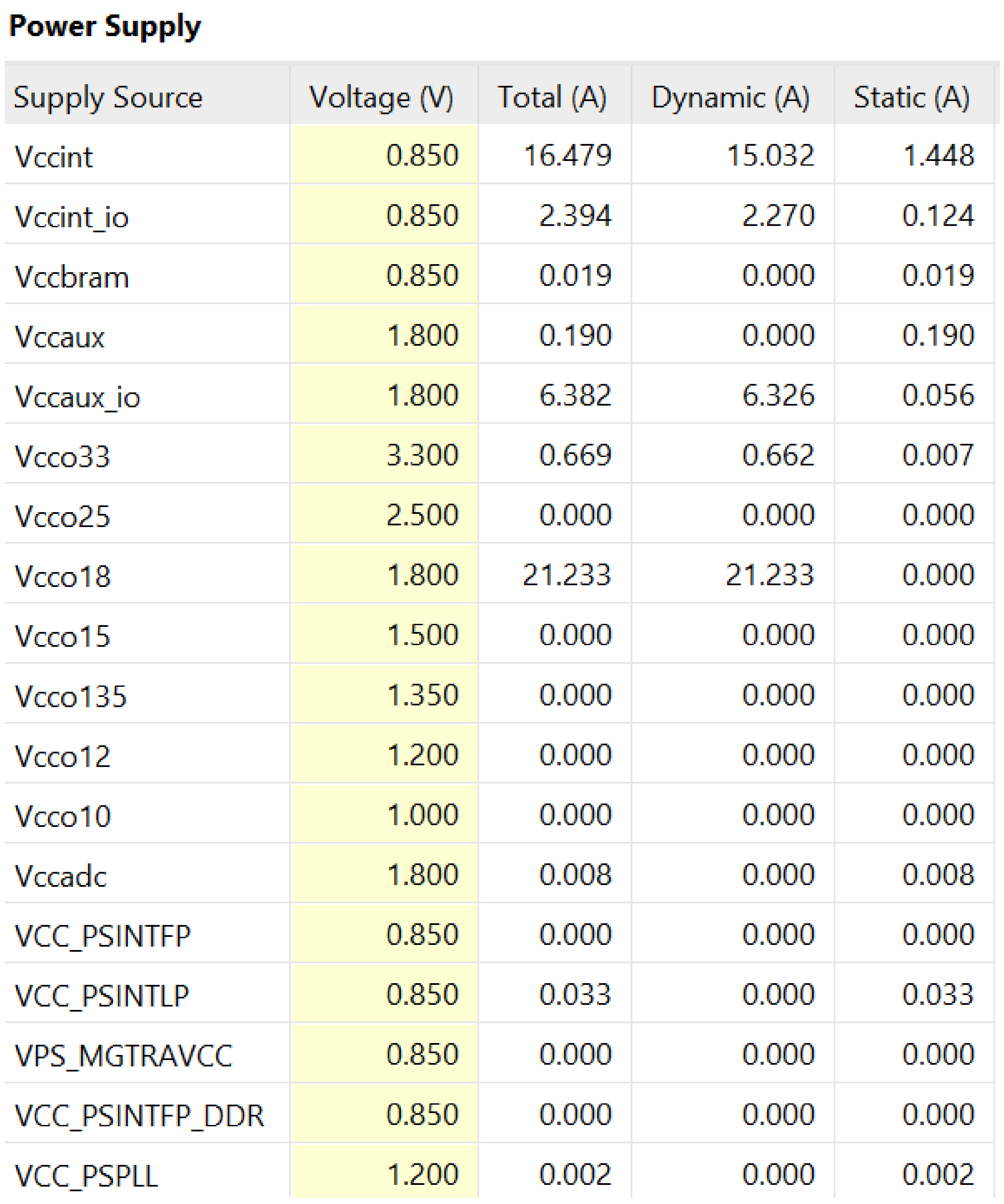
S.NO	Parameters	Boyou Zhou et.al,	Proposed Model
1.	Avg Utilization (I/0(W))	0.25W	0.148W
2.	Avg Utilization(SHA-256(W))	0.41W	0.16W
3.	Avg Utilization(Logic(W))	-	0.4W
4.	Avg Utilization(Data(W))	0.4W	0.33W
5.	Avg Encryption Time(Sec)	$10^{-1}sec$	$10^{-2}sec$
6.	Avg power consumption(W)	0.45W	1W



Power Supply

Supply Source	Voltage (V)	Total (A)	Dynamic (A)	Static (A)
VCC_PSINTLP	0.850	0.033	0.000	0.033
VPS_MGTRAVCC	0.850	0.000	0.000	0.000
VCC_PSINTFP_DDR	0.850	0.000	0.000	0.000
VCC_PSPLL	1.200	0.002	0.000	0.002
VPS_MGTRAVTT	1.800	0.000	0.000	0.000
VCCO_PSDDR_504	1.200	0.000	0.000	0.000
VCC_PSAUX	1.800	0.002	0.000	0.002
VCC_PSBATT	1.200	0.000	0.000	0.000
VCC_PSDDR_PLL	1.800	0.000	0.000	0.000
VCCO_PSIO0_500	3.300	0.000	0.000	0.000
VCCO_PSIO1_501	3.300	0.000	0.000	0.000
VCCO_PSIO2_502	3.300	0.000	0.000	0.000
VCCO_PSIO3_503	3.300	0.000	0.000	0.000
VCC_PSADC	1.800	0.001	0.000	0.001
VCCINT_VCU	0.900	0.169	0.000	0.169
MGTAVcc	0.900	0.000	0.000	0.000
MGTAVtt	1.200	0.000	0.000	0.000
MGTVccaux	1.800	0.000	0.000	0.000

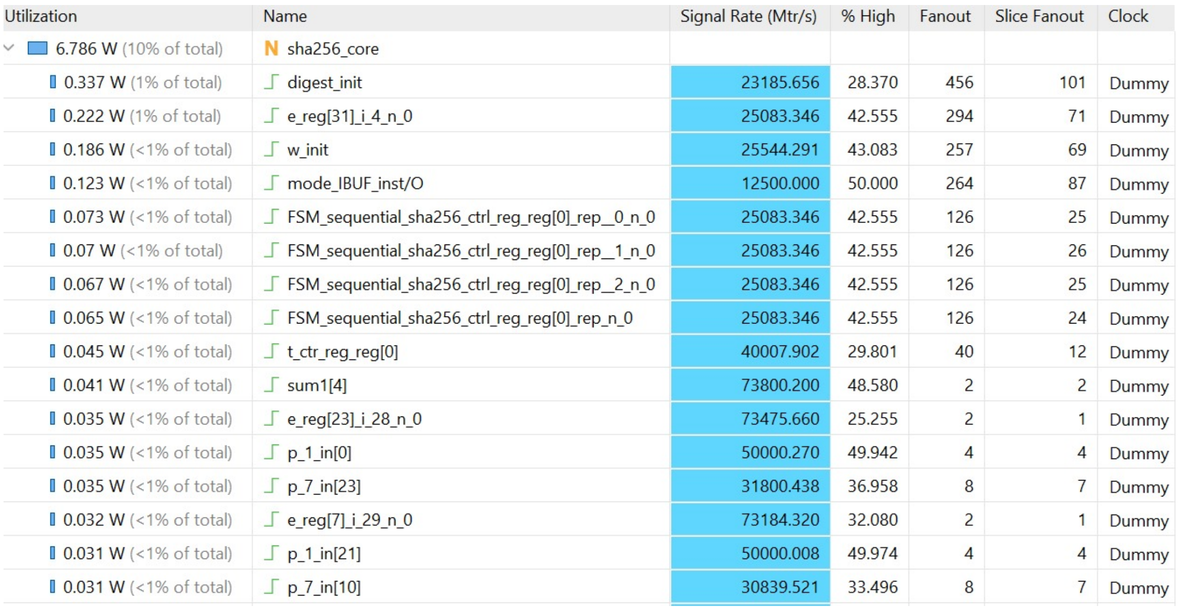
Figure 4.8: Power Supply(1)



Power Supply

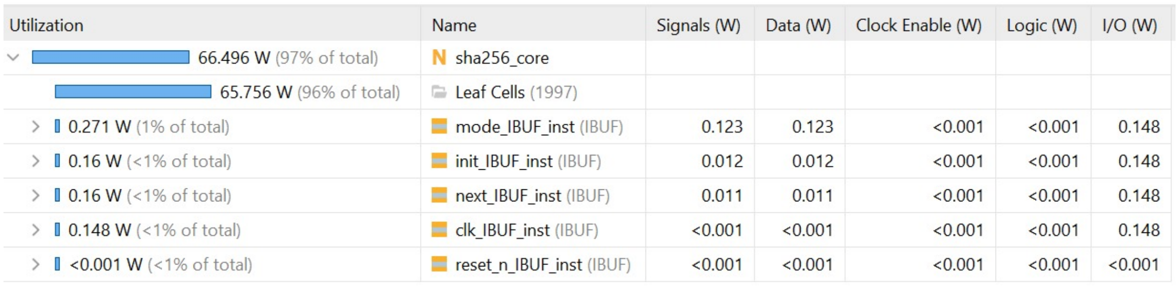
Supply Source	Voltage (V)	Total (A)	Dynamic (A)	Static (A)
Vccint	0.850	16.479	15.032	1.448
Vccint_io	0.850	2.394	2.270	0.124
Vccbram	0.850	0.019	0.000	0.019
Vccaux	1.800	0.190	0.000	0.190
Vccaux_io	1.800	6.382	6.326	0.056
Vcco33	3.300	0.669	0.662	0.007
Vcco25	2.500	0.000	0.000	0.000
Vcco18	1.800	21.233	21.233	0.000
Vcco15	1.500	0.000	0.000	0.000
Vcco135	1.350	0.000	0.000	0.000
Vcco12	1.200	0.000	0.000	0.000
Vcco10	1.000	0.000	0.000	0.000
Vccadc	1.800	0.008	0.000	0.008
VCC_PSINTFP	0.850	0.000	0.000	0.000
VCC_PSINTLP	0.850	0.033	0.000	0.033
VPS_MGTRAVCC	0.850	0.000	0.000	0.000
VCC_PSINTFP_DDR	0.850	0.000	0.000	0.000
VCC_PSPLL	1.200	0.002	0.000	0.002

Figure 4.9: Power Supply(2)



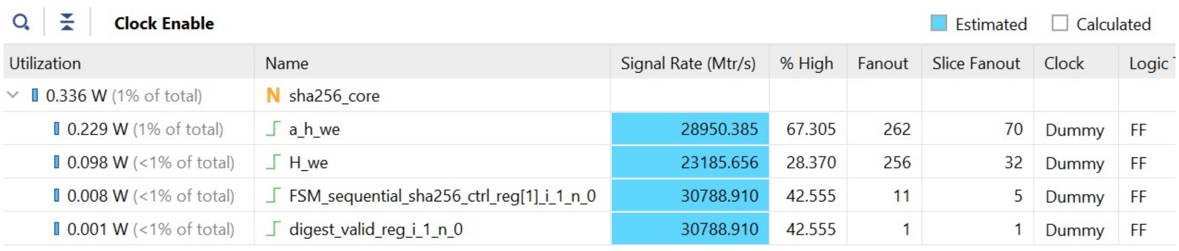
Utilization	Name	Signal Rate (Mtr/s)	% High	Fanout	Slice Fanout	Clock
6.786 W (10% of total)	sha256_core					
0.337 W (1% of total)	digest_init	23185.656	28.370	456	101	Dummy
0.222 W (1% of total)	e_reg[31]_i_4_n_0	25083.346	42.555	294	71	Dummy
0.186 W (<1% of total)	w_init	25544.291	43.083	257	69	Dummy
0.123 W (<1% of total)	mode_IBUF_inst/O	12500.000	50.000	264	87	Dummy
0.073 W (<1% of total)	FSM_sequential_sha256_ctrl_reg_reg[0]_rep__0_n_0	25083.346	42.555	126	25	Dummy
0.07 W (<1% of total)	FSM_sequential_sha256_ctrl_reg_reg[0]_rep__1_n_0	25083.346	42.555	126	26	Dummy
0.067 W (<1% of total)	FSM_sequential_sha256_ctrl_reg_reg[0]_rep__2_n_0	25083.346	42.555	126	25	Dummy
0.065 W (<1% of total)	FSM_sequential_sha256_ctrl_reg_reg[0]_rep_n_0	25083.346	42.555	126	24	Dummy
0.045 W (<1% of total)	t_ctr_reg_reg[0]	40007.902	29.801	40	12	Dummy
0.041 W (<1% of total)	sum1[4]	73800.200	48.580	2	2	Dummy
0.035 W (<1% of total)	e_reg[23]_i_28_n_0	73475.660	25.255	2	1	Dummy
0.035 W (<1% of total)	p_1_in[0]	50000.270	49.942	4	4	Dummy
0.035 W (<1% of total)	p_7_in[23]	31800.438	36.958	8	7	Dummy
0.032 W (<1% of total)	e_reg[7]_i_29_n_0	73184.320	32.080	2	1	Dummy
0.031 W (<1% of total)	p_1_in[21]	50000.008	49.974	4	4	Dummy
0.031 W (<1% of total)	p_7_in[10]	30839.521	33.496	8	7	Dummy

Figure 4.10: Utilization of SHA-256



Utilization	Name	Signals (W)	Data (W)	Clock Enable (W)	Logic (W)	I/O (W)
66.496 W (97% of total)	sha256_core					
65.756 W (96% of total)	Leaf Cells (1997)					
0.271 W (1% of total)	mode_IBUF_inst (IBUF)	0.123	0.123	<0.001	<0.001	0.148
0.16 W (<1% of total)	init_IBUF_inst (IBUF)	0.012	0.012	<0.001	<0.001	0.148
0.16 W (<1% of total)	next_IBUF_inst (IBUF)	0.011	0.011	<0.001	<0.001	0.148
0.148 W (<1% of total)	clk_IBUF_inst (IBUF)	<0.001	<0.001	<0.001	<0.001	0.148
<0.001 W (<1% of total)	reset_n_IBUF_inst (IBUF)	<0.001	<0.001	<0.001	<0.001	<0.001

Figure 4.11: Utilization Of Pins

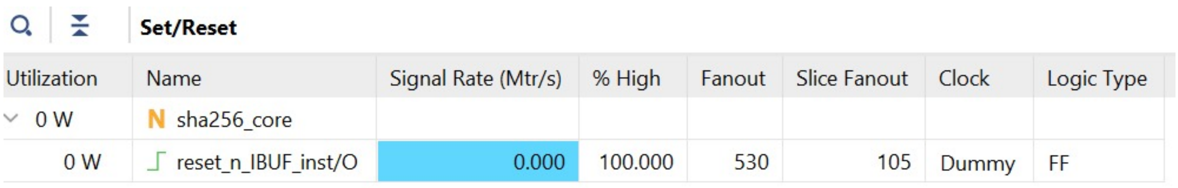


Clock Enable

🟦 Estimated ☐ Calculated

Utilization	Name	Signal Rate (Mtr/s)	% High	Fanout	Slice Fanout	Clock	Logic
0.336 W (1% of total)	sha256_core						
0.229 W (1% of total)	a_h_we	28950.385	67.305	262	70	Dummy	FF
0.098 W (<1% of total)	H_we	23185.656	28.370	256	32	Dummy	FF
0.008 W (<1% of total)	FSM_sequential_sha256_ctrl_reg[1]_i_1_n_0	30788.910	42.555	11	5	Dummy	FF
0.001 W (<1% of total)	digest_valid_reg_i_1_n_0	30788.910	42.555	1	1	Dummy	FF

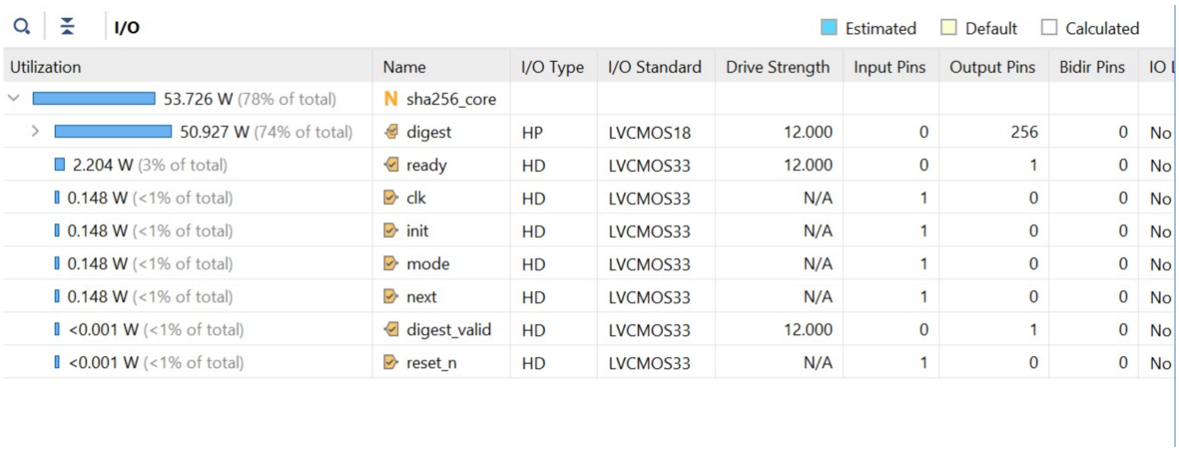
Figure 4.12: Utilization Of Clock



Set/Reset

Utilization	Name	Signal Rate (Mtr/s)	% High	Fanout	Slice Fanout	Clock	Logic Type
0 W	sha256_core						
0 W	reset_n_IBUF_inst/O	0.000	100.000	530	105	Dummy	FF

Figure 4.13: Set_Reset



Utilization	Name	I/O Type	I/O Standard	Drive Strength	Input Pins	Output Pins	Bidir Pins	IO
53.726 W (78% of total)	sha256_core							
50.927 W (74% of total)	digest	HP	LVCMOS18	12.000	0	256	0	No
2.204 W (3% of total)	ready	HD	LVCMOS33	12.000	0	1	0	No
0.148 W (<1% of total)	clk	HD	LVCMOS33	N/A	1	0	0	No
0.148 W (<1% of total)	init	HD	LVCMOS33	N/A	1	0	0	No
0.148 W (<1% of total)	mode	HD	LVCMOS33	N/A	1	0	0	No
0.148 W (<1% of total)	next	HD	LVCMOS33	N/A	1	0	0	No
<0.001 W (<1% of total)	digest_valid	HD	LVCMOS33	12.000	0	1	0	No
<0.001 W (<1% of total)	reset_n	HD	LVCMOS33	N/A	1	0	0	No

Figure 4.14: Utilization Of I_0

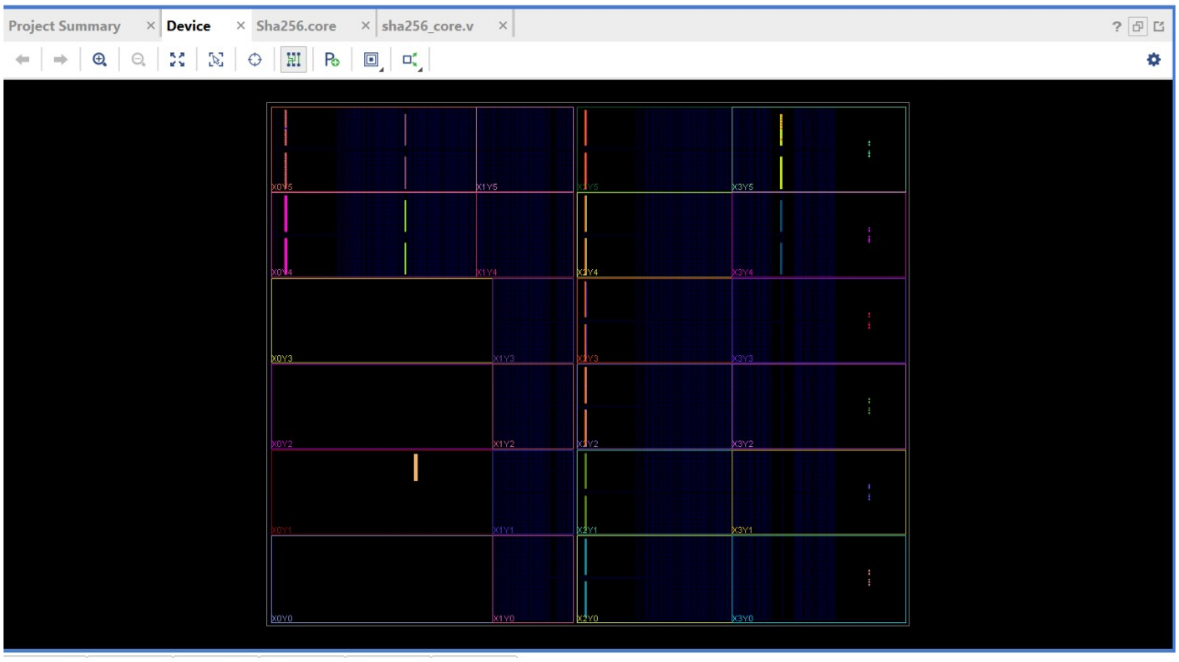


Figure 4.15: Netlist

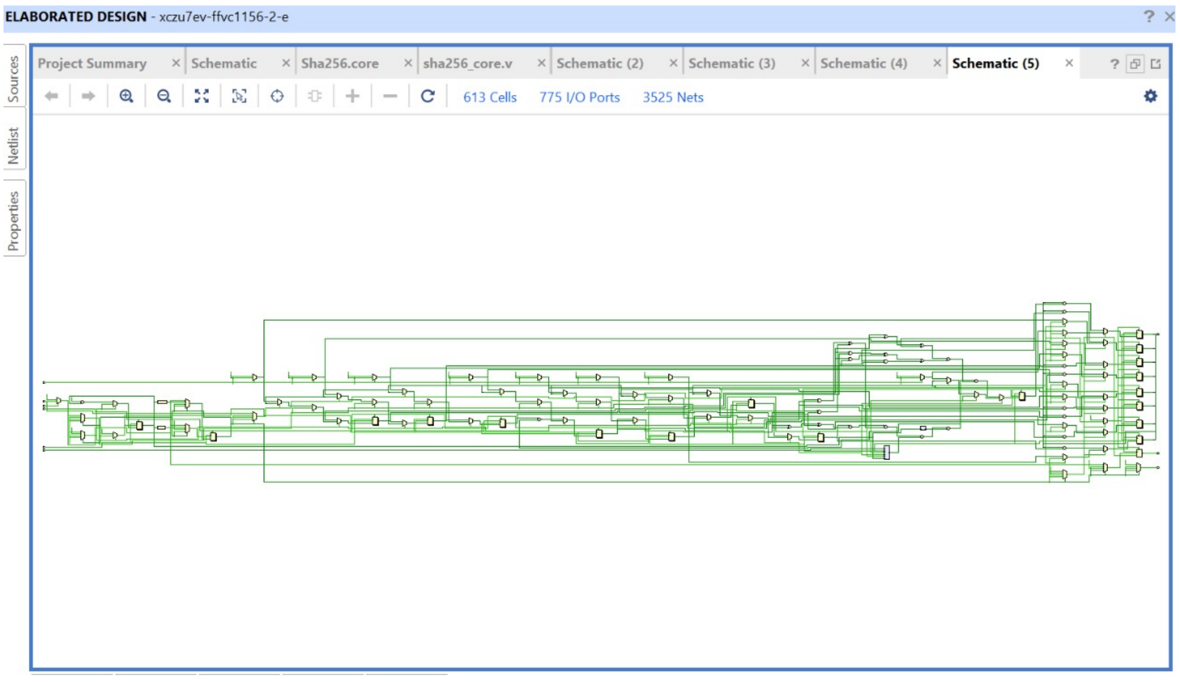
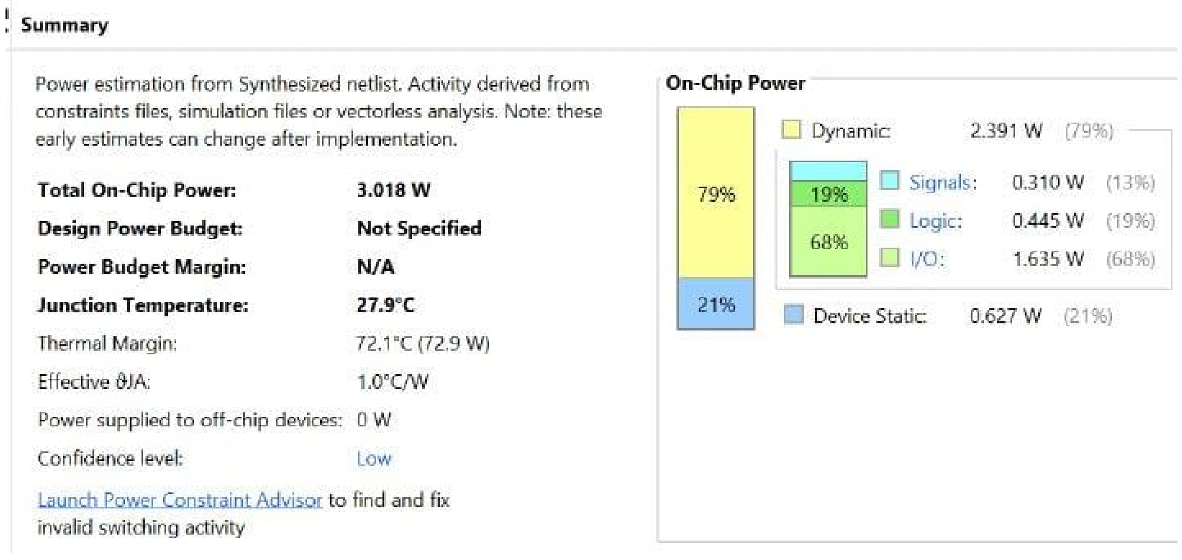


Figure 4.16: Eloborated Design



Summary

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**3.018 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**27.9°C**
Thermal Margin:	72.1°C (72.9 W)
Effective ϑJA:	1.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	2.391 W	(79%)
Signals:	0.310 W	(13%)
Logic:	0.445 W	(19%)
I/O:	1.635 W	(68%)
Device Static:	0.627 W	(21%)

Figure 4.17: On Chip Power Of Ideal Design

CHAPTER 5

CONCLUSION

The designed SoC architecture offers a comprehensive solution for building secured IoT devices, addressing the growing concerns surrounding data security and privacy in IoT deployments.By integrating cryptographic security features such as SHA-256 encryption, the device ensures data integrity, confidentiality, and authenticity, safeguarding sensitive information from unauthorized access or tampering. The inclusion of interfaces such as GPIO, I2C, and UART enhances the device's versatility and connectivity, allowing seamless integration with external sensors, actuators.The project demonstrates the feasibility and effectiveness of implementing secure IoT solutions using FPGA-based SoC architectures, offering a scalable and customizable platform for developing a wide range of IoT applications. Moving forward, the next steps involve testing the SoC architecture, verifying its functionality, and optimizing the design for performance, power efficiency, and scalability. Once the design is verified and optimized, the synthesis process will generate the bitstream required for programming the FPGA, enabling the deployment and testing of the secured IoT devices.In conclusion, the project lays the foundation for developing secure and reliable IoT devices, contributing to the advancement of IoT security and fostering trust and confidence in the deployment of IoT technologies across various domains.

REFERENCES

[1] Bongsoo Lee, Il-Gu Lee and Myungchul Kim, "Design and Implementation of Secure Cryptographic System on Chip for Internet of Things," *IEEE Access.,* vol. 10, pp. 8730-18742., Feb.2022.

[2] Hroub, Ayman, and Muhammad ES Elrabaa, "SecSoC: A Secure System on Chip Architecture for IoT Devices," *IEEE International Symposium on Hardware Oriented Security and Trust (HOST).,* pp.41-44, Jun.2022.

[3] Showail, Ahmad, Rashid Tahir, Muhammad Fareed Zaffar, Muhammad Haris Noor, and Mohammed Al-Khatib, "An internet of secure and private things:A service-oriented architecture.," *Computers & Security 120,* pp.102776., Sep.2022.

[4] Ang, Kenneth Li-Minn, and Jasmine Kah Phooi Seng, "Embedded intelligence: Platform technologies, device analytics, and smart city applications.," *IEEE Internet of Things Journal,* vol. 08, no. 17, pp. 13165-13182, Jun.2022.

[5] Fernandez, E.B., Washizaki, H., Yoshioka, N. and Okubo, T, "The design of secure IoT applications using patterns: State of the art and directions for research.," *Internet of Things.,*vol. 15, pp.100408,Sep 2021.

[6] Krishnamoorthy, R. and Krishnan, K, "Security Empowered System-on-Chip Selection for Internet of Things.," *Intelligent Automation & Soft Computing.,*vol.30,No.2, pp.404-418,Dec.2021.

[7] Alwarafy, A., Al-Thelaya, K.A., Abdallah, M., Schneider, J. and Hamdi, M., "A survey on security and privacy issues in edge-computing-assisted internet of things," *IEEE Internet of Things Journal.,* vol.8, no.06 , pp.4004-4022.,Aug.2020.

[8] Khan, Wazir Zada, Saqib Hakak, and Muhammad Khurram Khan, "Trust management in social internet of things: Architectures, recent advancements, and future challenges," *IEEE Internet of Things Journal.,* vol. 08, No.10, pp.7768-7788.,Nov.2020.

[9] Kammoun, M., Elleuchi, M., Abid, M. and BenSaleh, M.S, "FPGA-based implementation of the SHA-256 hash algorithm.," *IEEE international conference on design & test of integrated micro & nano-systems (DTS).,* pp.1-6, Jun. 2020.

[10] Almrezeq, N., Almadhoor, L., Alrasheed, T., Abd El-Aziz, A.A. and Nashwan, S, "Design a secure IoT architecture using smart wireless networks.," *International Journal of Communication Networks and Information Security.,*vol. 2, No.3, pp.401-410, Dec. 2020.

[11] Zandberg, K., Schleiser, K., Acosta, F., Tschofenig, H., "Secure firmware updates for constrained iot devices using open standards: A reality check.," *IEEE Access.,* No.7, pp.71907-71920, May. 2019.

[12] Ray, Partha Pratim., "A survey on Internet of Things architectures.," *Journal of King Saud University-Computer and Information Sciences.,*vol. 30, No.3, pp.291-319, Jul. 2018.

[13] Ngu AH, Gutierrez M, Metsis V, Nepal S, Sheng QZ, "IoT middleware: A survey on issues and enabling technologies.," *IEEE Internet of Things Journal.,*vol. 4, No.1, pp.1-20, oct. 2016.

[14] Ortiz AM, Hussein D, Park S, Han SN, Crespi N, "The cluster between internet of things and social networks: Review and research challenges.," *IEEE internet of things journal.,*vol. 1, No.3, pp.206-215, Apr. 2014.

[15] Xu T, Wendt JB, Potkonjak M., Abd El-Aziz, A.A. and Nashwan, S, "Security of IoT systems: Design challenges and opportunities.," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD).,* pp.417-423, Nov. 2014.

[16] Zhou, Boyou, Manuel Egele, and Ajay Joshi, "High-performance low-energy implementation of cryptographic algorithms on a programmable SoC for IoT devices.," *IEEE High Performance Extreme Computing Conference (HPEC).,* pp.1-6, Sep. 2017.