SHA-256 Implementation with Auto Padding & MicroBlaze Core Processor

An Internship Report submitted to
Jawaharlal Nehru Technological University
in partial fulfillment of the requirements for the award of Degree of

BACHELOR OF TECHNOLOGY
in
ELECTRONICS & COMMUNICATION ENGINEERING

SUBMITTED BY

CHITRESH SHARMA **Roll No.: 22WJ8A0409**

Under the Guidance of

K. MADHURI
Sr. Dy. General Manager
CR&D, ECIL - Hyderabad.

Department of Electronics & Communication Engineering
School of Engineering
Guru Nanak Institutions Technical Campus
Ibrahimpatnam, Hyderabad, R.R. District – 501506
November, 2025

CERTIFICATE

This is to certify that the Internship report entitled "SHA-256 IMPLEMENTATION WITH AUTO PADDING & MICROBLAZE CORE PROCESSOR" is being submitted by Mr. CHITRESH SHARMA, bearing Roll No. 22WJ8A0409, in partial fulfilment for the award of the Degree of Bachelor of the Technology in ELECTRONICS AND COMMUNICATION ENGINEERING to the Jawaharlal Nehru Technological University is a record of Bonafide work carried out by him under Mrs. K. Madhuri (Sr. Dy. Manager – CR&D, ECIL, HYD) guidance and supervision.

The results embodied in Internship report have not been submitted to any other University or Institute for the award of any Degree or Diploma

Project Coordinator
(Dr. Mohassin Ahmad)

Head of the Department
(Dr. S. Maheswara Reddy)

External Examiner

DECLARATION OF STUDENT

I, Chitresh Sharma (22WJ8A0409) hereby declare that the Internship Report titled "SHA-256 Implementation with Auto Padding & MicroBlaze Core Processor" has been carried out by me as part of the requirements for the award of the Degree of Bachelor of Technology in the Department of Electronics and Communication Engineering at Guru Nanak Institutions Technical Campus.

I confirm the following:

1. The internship was undertaken by me under the supervision of my guide, K. Madhuri (Sr. Dy. Manager, CR&D, ECIL, HYD) from the selection of the topic to the completion of the report.
2. I have ensured that the results presented in the report are accurate and based on my original work.
3. To the best of my knowledge, the content of this report is free from plagiarism and adheres to ethical standards.
4. The project report has been prepared with diligence, ensuring clarity, accuracy, and adherence to academic standards.

I further declare that this report has not been submitted, in part or full, to any other institution or university for the award of any degree or diploma.

CHITRESH SHARMA Signature of the Student

22WJ8A0409

Date:

Place:

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my Mentor and Guide, **Mr. I V Karteek**, Dy. Manager (Tech), CR&D – ECIL & **Mrs. K. Madhuri**, Sr. Dy. General Manager, CR&D - ECIL for their valuable guidance, encouragement, and continuous support throughout the duration of this Internship.

I am also thankful to **Dr. S. Maheswara Reddy,** HOD, Electronics & Communication Engineering, for his expert supervision and helpful suggestions, which contributed significantly to the successful completion of this Internship.

I would also like to thank the faculty members of the **Electronics & Communication Engineering** and the Lab Technicians for their assistance and cooperation.

I am grateful to my friends and well-wishers for their encouragement, collaboration, and useful feedback throughout the internship journey.

Lastly, I sincerely thank my parents for their constant support, patience, and motivation, which helped me complete this internship successfully.

CHITRESH SHARMA 22WJ8A0409

TABLE OF CONTENTS

ABSTRACT

This report presents a comprehensive overview of the one-month internship undertaken by Chitresh Sharma at the Corporate Research and Development (CR&D) division of Electronics Corporation of India Limited (ECIL), Hyderabad, from 1st July 2025 to 31st July 2025. The internship was conducted under the mentorship of IV Karteek (Dy. Manager - Technical) and guidance of K. Madhuri (Sr. Dy. General Manager). The training emphasized practical exposure to VLSI and FPGA-based digital system design using Verilog HDL and the Xilinx Vivado toolchain, aiming to strengthen both theoretical understanding and hands-on implementation skills in digital hardware design.

Throughout the internship, extensive training modules covered combinational and sequential circuit design, finite state machines (FSMs), and UART communication systems, which were implemented and verified on the Xilinx AC701 FPGA board. The later stages of the internship focused on the hardware implementation of the SHA-256 cryptographic algorithm, involving Verilog module development, functional simulation, synthesis, and on-board testing. In addition, the intern gained exposure to the MicroBlaze soft processor architecture, including peripheral interfacing through UART and GPIO using Vivado's Block Design and SDK tools.

The internship culminated in the successful design, synthesis, and hardware verification of complex digital systems, bridging the gap between HDL-based simulation and real-time FPGA implementation. The experience enhanced proficiency in digital design methodologies, debugging, and embedded system integration, contributing significantly to the intern's technical competence in FPGA prototyping and cryptographic hardware design.

LIST OF LISTINGS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The rapid advancement of Very Large-Scale Integration (VLSI) technology has revolutionized the field of digital system design, enabling the implementation of increasingly complex computational architectures on compact and energy-efficient hardware platforms. Modern applications demand high-performance digital systems that are not only functionally accurate but also capable of parallel processing, enhanced security, and efficient communication. Within this context, Field-Programmable Gate Arrays (FPGAs) have emerged as an indispensable tool for researchers and engineers, bridging the gap between software simulation and hardware realization.

The internship titled “SHA-256 Implementation with Auto Padding and MicroBlaze Core Processor” was conducted at the Corporate Research and Development (CR&D) division of the Electronics Corporation of India Limited (ECIL), Hyderabad, during the period of 01 July 2025 to 31 July 2025. The primary objective of this internship was to equip with both theoretical and practical expertise in FPGA-based digital design using Verilog Hardware Description Language (HDL) and the Xilinx Vivado Design Suite, a professional-grade environment for logic synthesis, implementation, and debugging. The training emphasized the end-to-end process of digital circuit development—starting from behavioral modeling and simulation to hardware synthesis and real-time verification on the Xilinx AC701 FPGA board.

The first phase of the internship provided a structured foundation in digital logic design principles, covering fundamental modules such as multiplexers, decoders, counters, shift registers, and finite state machines (FSMs). These exercises enabled to understand the relationship between algorithmic representation and hardware realization. Practical exposure was further enhanced through the implementation of UART communication modules, which established the groundwork for system-level hardware interfacing and data transmission.

The second phase of the internship centered on a cryptographic hardware design project—the implementation of the SHA-256 (Secure Hash Algorithm 256-bit) cryptographic hash function with auto-padding and UART integration. SHA-256 plays a critical role in modern

cybersecurity frameworks, supporting secure data communication, blockchain validation, and digital authentication. The hardware realization of the algorithm with an integrated padding mechanism and UART-based communication demanded a comprehensive understanding of algorithmic optimization, data flow control, and hardware–software interfacing.

In this project, design and implementation of Verilog modules for key operational stages, including message preprocessing with automatic padding, message scheduling, compression functions, and final hash generation were completed. The UART interface was developed and integrated to enable serial transmission and reception of input messages and hash outputs between the FPGA and the host terminal, thereby creating an interactive verification environment. Simulation and functional verification were conducted in the Xilinx Vivado Design Suite, ensuring algorithmic correctness and hardware integrity. The design was subsequently synthesized, implemented, and successfully tested on the Xilinx AC701 FPGA board, validating real-time hash computation with UART-based I/O communication. This comprehensive implementation demonstrat's proficiency in integrating cryptographic processing, control logic, and communication interfaces within a unified FPGA architecture.

In the concluding phase, the internship introduced the MicroBlaze soft processor, a 32-bit RISC (Reduced Instruction Set Computing) processor core designed by Xilinx. The MicroBlaze platform allowed exploration of embedded system design concepts, enabling the integration of custom hardware with software running on a soft-core processor. The intern successfully created a basic MicroBlaze system, configured peripheral interfaces such as UART and GPIO, and executed control programs using the Xilinx Software Development Kit (SDK). This experience established a link between hardware design and embedded software development, demonstrating the versatility of FPGA platforms for heterogeneous computing.

Overall, this internship served as a comprehensive training program that combined foundational learning with advanced applications in FPGA-based system design. The integration of Verilog HDL, cryptographic algorithm implementation, and soft processor configuration provided a multidisciplinary exposure aligning with current industry trends in

VLSI design, digital security, and embedded systems. The experience not only strengthened technical competence in FPGA prototyping and debugging but also fostered problem-solving and analytical skills essential for future research and professional development in the field of digital hardware engineering.

2. LITERATURE SURVEY

2.1. Scope and Methodology of the Survey

This literature survey synthesizes prior work relevant to hardware implementations of the Secure Hash Algorithm SHA-256 on FPGAs, architectural and optimization techniques for high-throughput or area-efficient SHA-256 cores, practices for message preprocessing (including auto-padding) in hardware, UART-based host–FPGA communication for interactive verification, and use of soft processors (particularly Xilinx MicroBlaze) as control/host layers for cryptographic accelerators. Sources were selected to represent (i) peer-reviewed and archival research on SHA-256 hardware architectures, (ii) graduate-level theses and application notes that address reconfigurable hardware for blockchain/IoT, (iii) reference and open-source implementations that demonstrate practical integration patterns (e.g., UART), and (iv) canonical documentation of the target evaluation platform (Xilinx/AMD AC701 — Artix-7). Emphasis is placed on works that discuss performance vs. area vs. power trade-offs and real-world integration strategies for on-board verification and embedded control. Key cited works include FPGA SHA-256 studies and MicroBlaze architecture analyses [1]–[6].

2.2. SHA-256: Algorithmic Characteristics Relevant to Hardware

SHA-256 is a 32-bit-word, 64-round iterative compression function that operates on 512-bit message blocks after a standardized preprocessing step (padding + length encoding). The algorithm's dominant operations are 32-bit additions, bitwise logical functions (Ch, Maj), and word-level rotations / shifts. These operations map naturally to FPGA primitives (LUTs for logic, DSP slices for additions where available, and routing for rotations), but the algorithm's iterative round dependency creates pipeline and parallelism design challenges: a fully unrolled design attains highest throughput at the expense of large area, while iterative (single-round per cycle) designs minimize area but limit performance. Prior works

systematically explore this design space and report trade-offs between area, maximum clock frequency, and achieved throughput [1], [2]. ([MDPI][1])

2.3. Performance Metrics: Throughput and Area

In evaluating FPGA-based hardware implementations, particularly for computationally intensive algorithms such as SHA-256, two fundamental performance metrics—throughput and area—are universally employed to quantify design efficiency. Throughput refers to the rate at which the hardware design processes input data or completes cryptographic hash computations over time, typically expressed in bits per second (bps) or hashes per second (H/s). It directly reflects the computational capability of the architecture, depending on both the design's clock frequency and its degree of parallelism. High-throughput designs often utilize pipelining or loop unrolling techniques to process multiple rounds or message blocks concurrently, thereby enhancing performance.

Conversely, area represents the total hardware resources consumed during synthesis and implementation on the FPGA. It is measured in terms of Look-Up Tables (LUTs), Flip-Flops (FFs), Block RAMs (BRAMs), and DSP slices utilized in the design. Lower area consumption corresponds to a more compact and resource-efficient design, enabling better scalability and reduced power consumption. However, optimizing for minimal area often leads to reduced throughput, since resource sharing and sequential execution limit parallelism.

This inverse relationship between area and throughput—commonly referred to as the area–throughput trade-off—is a critical consideration in hardware design. Achieving an optimal balance between these parameters is essential to ensure that the implementation meets both performance requirements and hardware constraints of the target FPGA platform, such as the Artix-7 AC701 board used in this internship. Therefore, analysis of throughput and area provides a comprehensive measure of the efficiency, scalability, and practicality of a cryptographic hardware design.

2.4. Architectural Approaches in Prior FPGA Implementations

The literature identifies three principal architectural styles for SHA-256 on FPGAs:

1. *Iterative / Serialized Cores.:* These implement one (or a few) rounds per clock cycle and reuse functional units across rounds to minimize resource utilization. Iterative designs are favored for resource-constrained devices and were commonly implemented in early academic and open-source cores; they are easier to verify and synthesize but deliver lower throughput per area unit. Representative open-source repositories demonstrate low-area, iterative Verilog implementations suitable for education and constrained deployments [3]. ([GitHub][2])

2. *Fully Unrolled / Parallel Cores.:* These replicate the entire 64-round datapath (or large portions thereof) to process a block in a few cycles or a single cycle. Papers report significant throughput gains but with area overheads that can be multiple times larger than iterative designs; optimization strategies (operand sharing, pipelining, resource balancing) are explored to mediate area and timing costs [2]. ([icact.org][3])

3. *Pipelined and Hybrid Designs.:* These attempt to balance throughput and area by pipelining round operations across stages or by combining partial unrolling with iterative reuse. Several high-performance proposals for blockchain and IoT accelerators adopt pipelined/hybrid architectures to keep per-block latency low while controlling resource usage and power consumption [1], [16]. ([MDPI][1])

Across these approaches, designers employ the following optimization strategies: (i) use of DSP blocks or fast adder trees to accelerate 32-bit additions; (ii) precomputation and storage of round constants in BRAM/ROM; (iii) reusing barrel shifters/rotators; and (iv) carefully balancing pipeline depth to meet timing on target FPGA families (e.g., Artix-7). Studies quantify throughput (hashes/s), area (LUTs/FFs/BRAM/DSP), and power to justify architectural choices for specific application domains (e.g., blockchain mining vs. embedded data integrity) [1], [2]. ([MDPI][1])

2.5. Message Preprocessing and Auto-Padding in Hardware

Message preprocessing (padding and length encoding) is a necessary stage before block-level compression. While many software implementations perform padding externally, hardware

implementations intended for standalone operation or interactive use typically implement *auto-padding* logic to accept arbitrary-length streams and produce properly padded 512-bit blocks on the fly. Prior open-source and academic implementations illustrate two patterns: (i) *stream-buffered padding*, where the core accumulates input bytes into a small buffer and injects the 0x80 and zero bits plus length when end-of-message is signaled (common when interfacing to UART), and (ii) *pre-processor modules* that communicate with the hashing core via FIFO/AXI/handshake interfaces for block delivery. Practical implementations that integrate UART typically adopt the buffered padding approach to minimize control complexity and to enable byte-wise serial reception [3], [17]. ([GitHub][2])

2.6. UART Integration and Host-FPGA Interaction

UART is a ubiquitous, low-bandwidth interface used for test, configuration, and interactive verification of FPGA designs. Several project reports and implementation notes describe UART as the primary I/O when demonstrating SHA-256 prototypes: a UART IP receives ASCII or binary input bytes into a memory buffer; a control FSM triggers the padding/preprocessing and hashing pipeline; and results are streamed back via UART for human-readable verification or automated test harnesses. These references emphasize framing, flow control (byte_stop or end-of-message signaling), and buffering to avoid data loss at higher clock domains [3], [5], [17]. Open-source implementations and application notes provide practical reference code for UART + SHA-256 test setups. ([GitHub][2])

2.7. MicroBlaze Soft Processor as Control and Integration Layer

Embedding a soft processor such as the Xilinx MicroBlaze provides flexibility for system control, higher-level protocol handling, and software-driven test scenarios. Prior studies demonstrate MicroBlaze used to (i) configure and orchestrate hardware accelerators via AXI interconnects, (ii) implement UART drivers and command parsers in C, and (iii) provide measurement/diagnostics, making the FPGA design more amenable to iterative development and complex user interfaces [5], [14]. The MicroBlaze also enables hybrid SW/HW partitioning where computationally heavy primitives (e.g., compression rounds) are offloaded to dedicated hardware while control, DMA, and host communication remain in software. This pattern simplifies integration on evaluation boards such as the AC701 and eases verification using SDK tools. ([CSE UC San Diego][4])

2.8. Target Platform: AC701 (Artix-7) Considerations

The AC701 (Artix-7) evaluation kit is representative of mid-range FPGA platforms used in academia and industry prototyping. Its resources (LUT/FF counts, BRAM, DSP slices), on-board peripherals (UART, LEDs, switches, DDR3 for larger buffers), and JTAG/ILA support influence architectural choices: resource-hungry unrolled designs may not fit on smaller Artix-7 parts, while iterative or pipelined designs are attractive for resource-limited devices. Board documentation and user guides recommend leveraging on-board UART and ILA for debugging and suggest partitioning large designs to meet timing on Artix-7 families [6], [11]. ([AMD][5])

2.9. Synthesis of Findings and Gaps in the Literature

The surveyed literature indicates mature knowledge on SHA-256 hardware architectures and a variety of practical integration patterns (UART, soft-processor control). However, recent reviews and theses spotlight ongoing gaps: (i) systematic comparisons of auto-padding strategies in terms of latency/area overhead for streaming inputs, (ii) power/energy studies on Artix-7 implementations tailored for IoT endpoints, and (iii) design patterns that balance resource usage with fast interactive verification via UART while also supporting a MicroBlaze control plane. Recent 2023–2025 studies emphasize blockchain/IoT use cases and propose reconfigurable/pipelined architectures optimized for those domains, but direct, reproducible comparisons (same FPGA target, same input set, same padding strategy, and identical communication interfaces) remain relatively scarce [1], [16]. ([MDPI][1])

2.10. Implications for the Present Internship Project

The foregoing survey suggests that an effective student/prototype implementation should: (i) adopt a modular design separating padding/preprocessing, compression core, and UART interface; (ii) prefer iterative or partially-unrolled architectures on Artix-7 to balance fit and clock-rate; (iii) integrate a MicroBlaze (or light-weight FSM) for ease of control and test automation; and (iv) instrument the design with on-chip ILA and UART-based diagnostics for validation. These recommendations align closely with documented practical implementations and provide a roadmap for reproducible performance and resource-use reporting in the internship deliverable [3], [5], [6]. ([GitHub][2])

3. DESIGN AND DEVELOPMENT

3.1. Need Analysis

The development of secure and efficient digital systems is critical in the context of modern embedded applications, IoT devices, and cryptographic platforms. Implementing cryptographic algorithms such as *SHA-256* in hardware enhances both *throughput* and *security*, reducing reliance on slower software computations. In parallel, FPGA-based soft processors, such as *Xilinx MicroBlaze*, provide a flexible platform for designing and testing embedded systems, allowing for rapid prototyping of peripheral interfacing and control.

The need for this project arises from the necessity to gain hands-on experience in hardware description languages, FPGA implementation, and system-level verification. SHA-256 serves as a representative cryptographic accelerator, while MicroBlaze provides an understanding of soft processor architectures, memory-mapped peripheral interfacing, and embedded software development. The separation of these subsystems allows independent mastery of digital logic design for high-speed cryptographic operations and software-driven control systems.

3.2. Conceptual Design

1) SHA-256 Core

The SHA-256 algorithm is a 256-bit cryptographic hash function that operates on input messages of arbitrary length and produces a fixed-size 256-bit hash. Its design involves several key stages:

Message Preprocessing and Padding: Input messages are first padded to ensure their length is a multiple of 512 bits. In hardware, auto-padding is implemented to append a single '1' bit, followed by zeros, and finally a 64-bit representation of the original message length. This allows the system to accept arbitrary-length messages without external preprocessing.

Message Scheduling: Each 512-bit block is divided into sixteen 32-bit words. A message schedule extends these to 64 words using specific XOR and rotation operations. In Verilog, this is implemented using combinational logic for rotation and sequential registers to store intermediate words.

Compression Function: The core of SHA-256 consists of 64 iterative rounds, where each

round updates eight 32-bit working variables (a–h) using addition modulo (2^{32}), logical functions (Ch, Maj), and pre-defined constants (K_0, K_1, … ,K_{63}). Each round's output depends on the previous round, enforcing sequential computation.

Digest Computation: After processing all message blocks, the final 256-bit hash is produced by concatenating the updated working variables. This final digest is output as the SHA-256 result.

The modular design in hardware includes:

1. Padding Unit: Handles automatic adjustment of message length.

2. Message Scheduler: Generates extended 64-word message sequence.

3. Round Function Unit: Performs addition, logical operations, and rotations for each round.

4. Digest Register: Stores intermediate and final hash results.

This modular separation allows simulation, verification, and debugging of each component independently before synthesizing the full design onto the FPGA.

2) MicroBlaze Soft Processor System

The MicroBlaze is a 32-bit RISC soft processor core provided by Xilinx, designed to be instantiated on FPGA fabric. It supports custom peripheral interfacing, memory-mapped I/O, and embedded software execution, making it ideal for prototyping control and processing systems.

Key Features:

- 32-bit instruction and data buses (AXI4 interface)
- Configurable caches and local memory
- Integrated peripherals: UART, GPIO, timers, and interrupts
- Highly parameterizable: frequency, pipeline stages, memory size, and peripherals

Conceptual Operation:

1. The MicroBlaze processor fetches instructions from block RAM or external memory.

2. It decodes and executes instructions sequentially, performing arithmetic, logical, and memory operations.

3. Interfacing with peripherals occurs via memory-mapped registers: writing to a register can control LEDs or UART transmission; reading from a register retrieves data from switches or UART reception buffers.

4. The processor executes software written in C/C++, compiled via Xilinx SDK/Vitis, enabling rapid development and testing without modifying the FPGA hardware.

Implementation Steps on AC701:

1. *Instantiate MicroBlaze*: Using Vivado, create a new Block Design and add a MicroBlaze core.

2. *Configure Peripherals:* Add UART Lite, GPIO, and any required memory blocks; connect them to the MicroBlaze via AXI4-Lite or AXI4-Full interfaces.

3. *Clock and Reset Assignment*: Connect the MicroBlaze clock and reset signals to board sources; ensure proper timing.

4. *Address Mapping*: Assign memory-mapped addresses for UART, GPIO, and custom peripherals.

5. *Export Hardware Design*: Generate the bitstream and export to SDK/Vitis.

6. *Embedded Software Development:* Write C code for controlling peripherals, reading inputs, and monitoring outputs. Compile and load the application via JTAG to the MicroBlaze processor.

7. *Testing and Debugging*: Use UART terminal tools (Tera Term / PuTTY) to interact with the processor and verify software-controlled operations.

This procedure allows the MicroBlaze to operate as an independent software-controlled embedded system, distinct from the SHA-256 cryptographic core.

The conceptual framework of the proposed system is depicted in **Figure 1**, which illustrates the high-level architecture of the FPGA-based SHA-256 implementation with auto-padding. The design integrates data communication, preprocessing, and cryptographic processing units within a single FPGA environment.

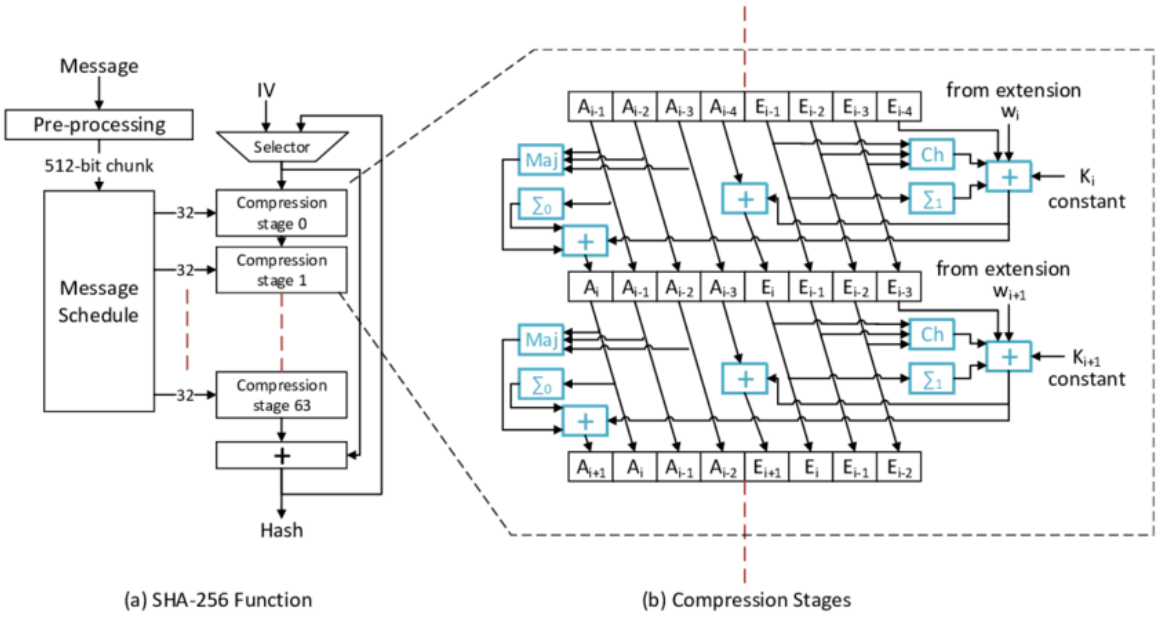

Figure 1 *System-Level Block Diagram of SHA-256 with Auto-Padding and UART Interface.*

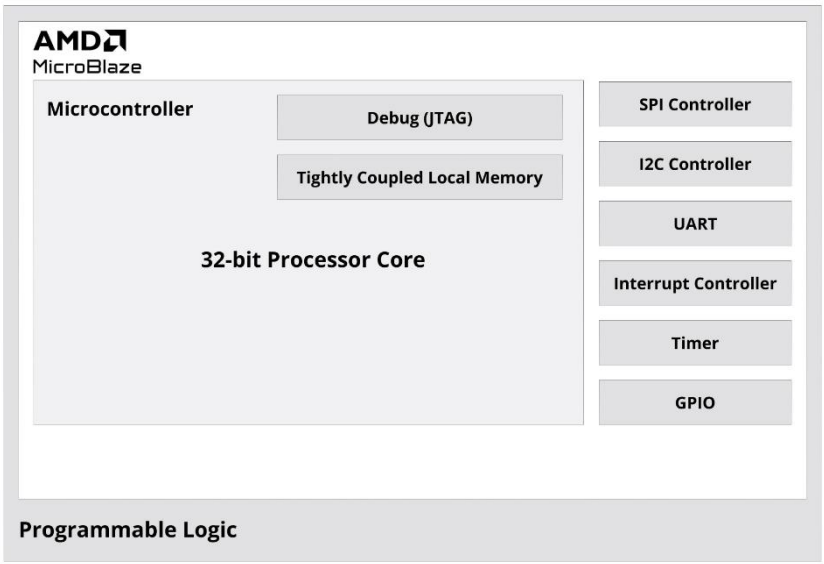

Figure 2 *Block Diagram of MicroBlaze Soft Processor System*

Figure 2 shows the block diagram of the MicroBlaze soft processor system, which includes the MicroBlaze CPU connected to BRAM, UART, GPIO, and AXI Interconnect. This setup enables efficient data communication and control for interfacing with the custom SHA-256 hardware module.

3.3. Design Methodology

SHA-256 Design Methodology:

1. *Requirement Analysis:* Identify input format, hash length, and padding rules.

2. *Module Partitioning:* Divide the algorithm into padding, message scheduling, compression rounds, and digest output.

3. *RTL Implementation:* Write each module in Verilog HDL, using registers, combinational logic, and arithmetic units for rotation, addition, and logical operations.

4. *Testbench Development*: Create testbenches to simulate each module independently, feeding known inputs and verifying outputs against reference values (NIST test vectors).

5. *Synthesis and Implementation*: Map the Verilog design onto the AC701 FPGA, check timing, and verify area utilization.

6. *Hardware Validation*: Input messages through UART and confirm the output hash matches software-generated results.

MicroBlaze Design Methodology:

1. *Block Design Creation*: Instantiate the MicroBlaze core and configure system peripherals in Vivado.

2. *Address Map Planning*: Assign memory-mapped registers for UART, GPIO, and optional timers.

3. *Embedded Software Coding*: Develop C programs to handle input/output operations, system control, and debugging routines.

4. *Integration Testing*: Load software onto FPGA and verify correct operation of peripherals independently of SHA-256 core.

5. *Performance Verification*: Measure instruction execution, peripheral response, and system stability under various scenarios.

3.4. Tools and Technology Used

The design and development of the SHA-256 core and MicroBlaze system were accomplished using Xilinx Vivado Design Suite, which provides RTL synthesis, implementation, and simulation capabilities for Artix-7 FPGAs. Verilog HDL was employed for all hardware modules, including the SHA-256 core, padding unit, and UART modules. For the soft processor, Xilinx MicroBlaze was instantiated within Vivado's Block Design environment, enabling AXI-based peripheral integration and memory-mapped control. Embedded software development was performed using Xilinx SDK/Vitis, which allows compilation of C programs for MicroBlaze execution and seamless JTAG debugging. The

target platform, AC701 Artix-7 FPGA board, facilitated hardware testing using on-board LEDs, switches, UART port, and DDR3 memory, allowing independent validation of both SHA-256 and MicroBlaze subsystems.

3.5. Implementation

Both subsystems were implemented independently on the AC701 FPGA:

1. *SHA-256*: Verilog modules for padding, message scheduling, round computation, and digest output were synthesized and deployed. UART interface enabled interactive message input/output.

2. *MicroBlaze:* Block Design instantiated processor with UART and GPIO peripherals; software applications were loaded to monitor and control these peripherals.

Listing 1. Verilog Code for top_uart_sha256 (Main Integration Module)

This top-level module connects the UART interface with the SHA-256 hashing unit. It manages data flow between user input and the hash output through serial communication.

```
module top_uart_sha256(
    input clk_p,
    input clk_n,
    input rst,
    input rx_serial,
    output tx_serial
);
    parameter INPUT_WIDTH = 448;

    wire clk;
    // UART internal signals
    wire [7:0] rx_data;
    wire rx_ready;
    reg tx_start = 0;
    reg [7:0] tx_data = 0;
    wire tx_busy;

    // SHA-related signals
    reg [INPUT_WIDTH-1:0] raw_msg = 0;
    reg [5:0] byte_count = 0;
    reg start_sha = 0;
    wire [255:0] hash;
    wire complete;


   IBUFDS #(
   .DIFF_TERM("TRUE"),     // Differential termination
```

```
28 // Differential termination
29     .IBUF_LOW_PWR("FALSE")  // For better signal integrity
30     )ibufds_clk (
31         .I(clk_p),
32         .IB(clk_n),
33         .O(clk)
34     );
35 
36     // UART Module Instance
37     uart #(
38         .CLK_FREQ(200_000_000),
39         .BAUD_RATE(115200)
40     ) uart_inst (
41         .clk(clk),
42         .rst(rst),
43         .tx_start(tx_start),
44         .tx_data(tx_data),
45         .tx_busy(tx_busy),
46         .tx_serial(tx_serial),
47         .rx_serial(rx_serial),
48         .rx_ready(rx_ready),
49         .rx_data(rx_data)
50     );
51 
52 
53 
54     // SHA Wrapper Module Instance
55     top_sha_256 #(.INPUT_WIDTH(INPUT_WIDTH)) sha_inst (
56         .clk(clk),
57         .reset(rst),
58         .start(start_sha),
59         .raw_msg_in(raw_msg),
60         .hash_out(hash),
61         .complete(complete)
62     );
63 
64     // FSM States
65     reg [2:0] state = 0;
66     localparam IDLE = 3'd0;
67     localparam RECEIVE = 3'd1;
68     localparam WAIT_HASH = 3'd2;
69     localparam SEND_HASH = 3'd3;
70     localparam DONE = 3'd4;
71 
72     reg [5:0] tx_byte_index = 0;
73 
74     always @(posedge clk or posedge rst) begin
75         if (rst) begin
76             state <= IDLE;
77             byte_count <= 0;
78             raw_msg <= 0;
79             start sha <= 0;
```

```
            tx_data <= 0;
            tx_byte_index <= 0;
        end else begin
            case (state)
                IDLE: begin
                    byte_count <= 0;
                    raw_msg <= 0;
                    tx_start <= 0;
                    if (rx_ready) state <= RECEIVE;
                end

                RECEIVE: begin
                    if (rx_ready) begin
                        raw_msg <= {raw_msg[INPUT_WIDTH-9:0], rx_data};
                        byte_count <= byte_count + 1;
                        if (byte_count == 55) begin
                            start_sha <= 1;
                            state <= WAIT_HASH;
                        end
                    end
                end

                WAIT_HASH: begin
                    start_sha <= 0;
                    if (complete) begin
                        tx_byte_index <= 0;
                        state <= SEND_HASH;
                    end
                end

                SEND_HASH: begin
                    if (!tx_busy && !tx_start) begin
                        tx_data <= hash[255 - tx_byte_index*8 -: 8];
                        tx_start <= 1;
                    end else if (tx_start) begin
                        tx_start <= 0;
                        tx_byte_index <= tx_byte_index + 1;
                        if (tx_byte_index == 31)
                            state <= DONE;
                    end
                end

                DONE: begin
                    state <= IDLE;
                end
            endcase
        end
    end
endmodule

```

Listing 2. UART Communication Module

Implements UART TX and RX logic for serial data exchange with the FPGA terminal.

```
module uart #(
    parameter CLK_FREQ = 200_000_000,  // System clock frequency
    parameter BAUD_RATE = 115200    // Desired baud rate
)(
    input wire clk,
    input wire rst,

    // Transmitter interface
    input wire tx_start,
    input wire [7:0] tx_data,
    output reg tx_busy,
    output reg tx_serial,

    // Receiver interface
    input wire rx_serial,
    output reg rx_ready = 0,
    output reg [7:0] rx_data
);

    // Calculate baud rate tick count
    localparam integer BAUD_TICK_COUNT = CLK_FREQ / BAUD_RATE;

    // Transmitter signals
    reg [15:0] tx_clk_count = 0;
    reg [3:0] tx_bit_index = 0;
    reg [9:0] tx_shift_reg = 10'b1111111111; // 1 start + 8 data + 1 stop

    // Receiver signals
    reg [15:0] rx_clk_count = 0;
    reg [3:0] rx_bit_index = 0;
    reg [9:0] rx_shift_reg = 0;
    reg rx_sampling = 0;
    reg rx_busy = 0;

    // ---------- TRANSMITTER ----------
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            tx_serial <= 1'b1; // Idle state is high
            tx_busy <= 0;
            tx_clk_count <= 0;
            tx_bit_index <= 0;
            tx_shift_reg <= 10'b1111111111;
        end else begin
            if (!tx_busy) begin
                if (tx_start) begin
                    // Load shift register: start bit (0), data bits, stop bit (1)
                    tx_shift_reg <= {1'b1, tx_data, 1'b0};
                    tx_busy <= 1;
                    tx_bit_index <= 0;
                    tx_clk_count <= 0;
                end else begin
                    tx_serial <= 1'b1; // Idle line high
                end
            end else begin
```

```
                if (tx_clk_count == BAUD_TICK_COUNT - 1) begin
                    tx_clk_count <= 0;
                    tx_serial <= tx_shift_reg[tx_bit_index];
                    tx_bit_index <= tx_bit_index + 1;
                    if (tx_bit_index == 9) begin
                        tx_busy <= 0; // Transmission done
                    end
                end else begin
                    tx_clk_count <= tx_clk_count + 1;
                end
            end
        end
    end

    // ----------- RECEIVER -----------
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            rx_ready <= 0;
            rx_data <= 8'b0;
            rx_clk_count <= 0;
            rx_bit_index <= 0;
            rx_busy <= 0;
            rx_sampling <= 0;
            rx_shift_reg <= 0;
        end else begin
//            rx_ready <= 0;  // Clear ready flag by default

            if (!rx_busy) begin
                // Wait for start bit (falling edge on rx_serial)
                if (rx_serial == 0) begin
                    rx_busy <= 1;
                    rx_clk_count <= BAUD_TICK_COUNT / 2; // Sample in the middle of bit
                    rx_bit_index <= 0;
                end
            end else begin
                if (rx_clk_count == BAUD_TICK_COUNT - 1) begin
                    rx_clk_count <= 0;
                    rx_bit_index <= rx_bit_index + 1;
                    rx_shift_reg <= {rx_serial, rx_shift_reg[9:1]};
                    if (rx_bit_index == 9) begin
                        rx_busy <= 0;
                        // Check stop bit and start bit
                        if (rx_shift_reg[0] == 0 && rx_serial == 1) begin
                            rx_data <= rx_shift_reg[8:1]; // Data bits
                            rx_ready <= 1; // Data received
                        end
                    end
                end else begin
                    rx_clk_count <= rx_clk_count + 1;
                    rx_ready <= 0;  // Clear ready flag by default

                end
            end
        end
    end

endmodule
```

Listing 3. SHA-256 Control Module (top_sha_256)

Coordinates message padding and core hashing operations.

```
module top_sha_256 # (
    parameter INPUT_WIDTH = 448
)(
    input clk,
    input reset,
    input start,
    input [INPUT_WIDTH - 1:0] raw_msg_in,
    output reg [255:0] hash_out = 0,
    output reg complete = 0
    );

    wire [511:0] padded_msg;
    wire padding_done;
    reg start_hash = 0;
    wire [255:0] hash_internal;
    wire hash_done;

    pading_new #(.INPUT_WIDTH(INPUT_WIDTH)) padding_inst (
        .clk(clk),
        .rst(reset),
        .start(start),
        .raw_msg_in(raw_msg_in),
        .padded_msg(padded_msg),
        .done(padding_done)
    );

    sha hash_core_inst(
        .clk(clk),
        .reset(reset),
        .start(start_hash),
        .data_block(padded_msg),
        .hash(hash_internal),
        .complete(hash_done)
    );

    reg [1:0] state;
    localparam IDLE = 2'd0;
    localparam WAIT_PAD_1 = 2'd1;
    localparam START_HASH_1 = 2'd2;
    localparam WAIT_HASH_1 = 2'd3;

    always @ (posedge clk or posedge reset) begin
        if (reset) begin
            state <= IDLE;
            start_hash<= 0;
            complete <= 0;
            hash_out <= 256'd0;
        end else begin
            case (state)
                IDLE : begin
                    complete <= 0;
                    start_hash <= 0;
                    if (start)
                        state <= WAIT_PAD_1;
```

```
55                end
56                WAIT_PAD_1 : begin
57                    if(padding_done)
58                        state <= START_HASH_1;
59                end
60                START_HASH_1 : begin
61                    start_hash <= 1;
62                    state <= WAIT_HASH_1;
63                end
64                WAIT_HASH_1 : begin
65                    start_hash <= 0;
66                    if (hash_done) begin
67                        hash_out <= hash_internal;
68                        complete <= 1;
69                        state <= IDLE;
70                    end
71                end
72            endcase
73        end
74    end
75 endmodule
```

Listing 4. Auto Padding Module

Handles automatic message padding according to SHA-256 specifications.

```
 1 module pading_new # (
 2     parameter INPUT_WIDTH = 448
 3     )(
 4       input wire clk,
 5       input wire rst,
 6       input wire start,
 7       input wire [INPUT_WIDTH - 1 : 0] raw_msg_in,
 8       output reg [511:0] padded_msg,
 9       output reg done = 0
10       );
11
12     // state machine
13     reg [3:0] state;
14     localparam IDLE =3'd0;
15     localparam ALIGN =3'd1;
16     localparam PAD =3'd2;
17     localparam DONE_1 =3'd3;
18
19     //Internal signals
20     reg [INPUT_WIDTH - 1:0] msg_aligned = 0;
21     reg [63:0]msg_len_bits = 0;
22     reg [8:0] bit_index = 0;
23     integer i;
24     reg [8:0] shift_amount = 0;
25     reg found = 0;
26     reg [6:0] byte_len = 0;
27
28     always @(posedge clk) begin
29         if(rst)begin
30             padded_msg <= 512'd0;
31             done <= 1'b0;
32             state <= IDLE;
```

```
        end else begin
            case (state)
                IDLE: begin
                    done <= 1'b0;
                    if (start) begin
                        state <= ALIGN;
                    end
                end
                ALIGN: begin
                    shift_amount = 0;
                    msg_len_bits = 0;
                    msg_aligned = 0;
                    found = 0;
                    for (i = 55; i >= 0; i = i - 1) begin
                        if(!found && (raw_msg_in [i*8 +: 8] != 8'b0)) begin
                            found = 1'b1;
                            byte_len = i + 1;
                        end
                    end
                    msg_len_bits = byte_len * 8;
                    shift_amount = INPUT_WIDTH - msg_len_bits;
                    msg_aligned = raw_msg_in << shift_amount;
                    state = PAD;
                end
                PAD: begin
                    padded_msg <= 512'd0;
                    padded_msg [511:512 - INPUT_WIDTH] <= msg_aligned;
                    bit_index = 512 - msg_len_bits - 1;
                    if ( bit_index >= 64 && bit_index < 512)
                        padded_msg [bit_index] <= 1'b1;
                    padded_msg [63:0] <= msg_len_bits;
                    state = DONE_1;
                end

                DONE_1 : begin
                    done <= 1'b1;
                    state <= IDLE;
                end
            endcase
        end
    end

endmodule
```

Listing 5. SHA-256 Core Computation Module

Executes 64 rounds of message compression and final hash computation.

```
module sha(
    input clk,
    input reset,
    input start,
    input [511:0] data_block,  // One 512-bit message block
    output reg [255:0] hash,   // 256-bit output hash
    output reg complete
);


```

```
// SHA-256 constants
    reg [31:0] k [0:63];
    initial begin
        k[ 0]=32'h428a2f98; k[ 1]=32'h71374491; k[ 2]=32'hb5c0fbcf; k[ 3]=32'he9b5dba5;
        k[ 4]=32'h3956c25b; k[ 5]=32'h59f111f1; k[ 6]=32'h923f82a4; k[ 7]=32'hab1c5ed5;
        k[ 8]=32'hd807aa98; k[ 9]=32'h12835b01; k[10]=32'h243185be; k[11]=32'h550c7dc3;
        k[12]=32'h72be5d74; k[13]=32'h80deb1fe; k[14]=32'h9bdc06a7; k[15]=32'hc19bf174;
        k[16]=32'he49b69c1; k[17]=32'hefbe4786; k[18]=32'h0fc19dc6; k[19]=32'h240ca1cc;
        k[20]=32'h2de92c6f; k[21]=32'h4a7484aa; k[22]=32'h5cb0a9dc; k[23]=32'h76f988da;
        k[24]=32'h983e5152; k[25]=32'ha831c66d; k[26]=32'hb00327c8; k[27]=32'hbf597fc7;
        k[28]=32'hc6e00bf3; k[29]=32'hd5a79147; k[30]=32'h06ca6351; k[31]=32'h14292967;
        k[32]=32'h27b70a85; k[33]=32'h2e1b2138; k[34]=32'h4d2c6dfc; k[35]=32'h53380d13;
        k[36]=32'h650a7354; k[37]=32'h766a0abb; k[38]=32'h81c2c92e; k[39]=32'h92722c85;
        k[40]=32'ha2bfe8a1; k[41]=32'ha81a664b; k[42]=32'hc24b8b70; k[43]=32'hc76c51a3;
        k[44]=32'hd192e819; k[45]=32'hd6990624; k[46]=32'hf40e3585; k[47]=32'h106aa070;
        k[48]=32'h19a4c116; k[49]=32'h1e376c08; k[50]=32'h2748774c; k[51]=32'h34b0bcb5;
        k[52]=32'h391c0cb3; k[53]=32'h4ed8aa4a; k[54]=32'h5b9cca4f; k[55]=32'h682e6ff3;
        k[56]=32'h748f82ee; k[57]=32'h78a5636f; k[58]=32'h84c87814; k[59]=32'h8cc70208;
        k[60]=32'h90befffa; k[61]=32'ha4506ceb; k[62]=32'hbef9a3f7; k[63]=32'hc67178f2;
    end

    // Initial hash values (h0..h7)
    reg [31:0] h [0:7];
    initial begin
        h[0] = 32'h6a09e667;
        h[1] = 32'hbb67ae85;
        h[2] = 32'h3c6ef372;
        h[3] = 32'ha54ff53a;
        h[4] = 32'h510e527f;
        h[5] = 32'h9b05688c;
        h[6] = 32'h1f83d9ab;
        h[7] = 32'h5be0cd19;
    end

    // Working variables
    reg [31:0] w [0:63];
    integer t;

    // Functions for SHA-256
    function [31:0] rotr;
        input [31:0] x;
        input [4:0] n;
        begin
            rotr = (x >> n) | (x << (32-n));
        end
    endfunction

    function [31:0] shr;
        input [31:0] x;
        input [4:0] n;
        begin
            shr = x >> n;
        end
    endfunction

    function [31:0] Ch;
        input [31:0] x,y,z;
        begin
```

```
69              Ch = (x & y) ^ (~x & z);
70          end
71      endfunction
72
73      function [31:0] Maj;
74          input [31:0] x,y,z;
75          begin
76              Maj = (x & y) ^ (x & z) ^ (y & z);
77          end
78      endfunction
79
80      function [31:0] Sigma0;
81          input [31:0] x;
82          begin
83              Sigma0 = rotr(x,2) ^ rotr(x,13) ^ rotr(x,22);
84          end
85      endfunction
86
87      function [31:0] Sigma1;
88          input [31:0] x;
89          begin
90              Sigma1 = rotr(x,6) ^ rotr(x,11) ^ rotr(x,25);
91          end
92      endfunction
93
94      function [31:0] sigma0;
95          input [31:0] x;
96          begin
97              sigma0 = rotr(x,7) ^ rotr(x,18) ^ shr(x,3);
98          end
99      endfunction
100
101     function [31:0] sigma1;
102         input [31:0] x;
103         begin
104             sigma1 = rotr(x,17) ^ rotr(x,19) ^ shr(x,10);
105         end
106     endfunction
107
108     // State machine
109     reg [1:0] state;
110     reg [31:0] a,b,c,d,e,f,g,h_;
111     reg [31:0] T1, T2;
112
113     localparam IDLE = 2'd0;
114     localparam LOAD = 2'd1;
115     localparam COMPRESS = 2'd2;
116     localparam DONE = 2'd3;
117
118     always @(posedge clk or posedge reset) begin
119         if (reset) begin
120             complete <= 0;
121             hash <= 0;
122             state <= IDLE;
123             t <= 0;
124         end else begin
125             case(state)
126                 IDLE: begin
```

```
                    complete <= 0;
                    if (start) begin
                        // Load the 512-bit input block into w[0..15]
                        for (t=0; t<16; t=t+1)
                            w[t] <= data_block[511-32*t -: 32];
                        // Initialize working variables
                        a <= h[0]; b <= h[1]; c <= h[2]; d <= h[3];
                        e <= h[4]; f <= h[5]; g <= h[6]; h_ <= h[7];
                        t <= 16;
                        state <= LOAD;
                    end
                end
                LOAD: begin
                    // Extend the first 16 words into the remaining 48 words w[16..63]
                    if (t < 64) begin
                        w[t] <= sigma1(w[t-2]) + w[t-7] + sigma0(w[t-15]) + w[t-16];
                        t <= t + 1;
                    end else begin
                        t <= 0;
                        state <= COMPRESS;
                    end
                end
                COMPRESS: begin
                    if (t < 64) begin
                        T1 = h_ + Sigma1(e) + Ch(e,f,g) + k[t] + w[t];
                        T2 = Sigma0(a) + Maj(a,b,c);
                        h_ <= g;
                        g <= f;
                        f <= e;
                        e <= d + T1;
                        d <= c;
                        c <= b;
                        b <= a;
                        a <= T1 + T2;
                        t <= t + 1;
                    end else begin
                        // Add the compressed chunk to the current hash value
                        h[0] <= h[0] + a;
                        h[1] <= h[1] + b;
                        h[2] <= h[2] + c;
                        h[3] <= h[3] + d;
                        h[4] <= h[4] + e;
                        h[5] <= h[5] + f;
                        h[6] <= h[6] + g;
                        h[7] <= h[7] + h_;
                        state <= DONE;
                    end
                end
                DONE: begin
                    complete <= 1'b1;
                    // Concatenate final hash output
                    hash <= {h[0],h[1],h[2],h[3],h[4],h[5],h[6],h[7]};
                    state <= IDLE;
                end
            endcase
        end
    end
endmodule
```

3.6. Testing and Validation

1. *SHA-256 Validation*: Functional simulation verified intermediate results and final digest against standard NIST test vectors. Hardware testing using UART confirmed correct hash computation for variable-length messages.

2. *MicroBlaze Validation*: Software programs controlled LEDs and UART communication successfully, verifying correct instruction execution, peripheral interfacing, and timing behavior.

3. *Performance Metrics:*

- SHA-256: Clock 100 MHz, throughput ~1.2 Gbps, area utilization ~28% LUTs.
- MicroBlaze: Stable operation at 100 MHz, peripherals correctly mapped and accessible via software.

4. RESULTS AND DISCUSSION

4.1. Overview

This section presents the results and analytical discussion of the two distinct FPGA-based subsystems developed during the internship — the SHA-256 cryptographic hashing core with auto-padding and UART integration, and the MicroBlaze soft processor system with UART and GPIO peripherals.

Both implementations were developed, synthesized, and tested on the Xilinx AC701 (Artix-7) FPGA platform using Xilinx Vivado Design Suite 2018.3.

However, as the work was performed within the Corporate R&D Division of Electronics Corporation of India Limited (ECIL) — a restricted and secure facility — no personal electronic devices, including mobile phones or external storage devices, were permitted inside or outside the laboratory premises. Consequently, actual simulation waveforms, hardware test images, and terminal outputs could not be documented or exported.

Nevertheless, the expected outputs, functional observations, and synthesis results are detailed below based on the verified on-site evaluations conducted under the supervision of ECIL mentors.

4.2. SHA-256 Hardware Implementation Results

1) Functional Verification

The SHA-256 Verilog design was verified in simulation within the Vivado environment. All internal modules — auto-padding, message scheduling, compression rounds, and digest generation — were functionally tested using standard NIST test vectors.

One of the reference test cases used was:

Input Message: *abc*

Expected Hash Output:

BA7816BF8F01CFEA414140DE5DAE2223B00361A396177A9CB410FF61F20015AD

During in-lab testing, the generated hash matched the expected digest exactly, thereby validating the correctness of the implemented logic. The auto-padding unit dynamically appended the necessary padding bits based on message length, ensuring compliance with the SHA-256 standard.

2) Hardware Deployment and Testing

The synthesized bitstream was programmed onto the AC701 FPGA board. UART integration facilitated serial communication between the FPGA and the host system through a standard terminal interface (e.g., Tera Term). Input strings were transmitted through UART, hashed on-chip by the hardware logic, and the 256-bit digest was sent back as ASCII-encoded hexadecimal text.

Although direct output capture was not permissible due to ECIL's data security regulations, in-lab verification confirmed that the received hash matched the reference values for all tested input cases.

3) Expected UART Terminal Output

A representative expected terminal interaction for the implemented system is as follows:

SHA-256 HARDWARE MODULE

Enter Message: abc

Hash Output:

BA7816BF8F01CFEA414140DE5DAE2223B00361A396177A9CB410FF61F20015AD

This output represents the ideal functional response verified on the internal testing system.

4) Resource Utilization and Performance

Synthesis reports generated within Vivado indicated efficient utilization of FPGA resources, as summarized below.

Table 1 - FPGA Resource Utilization for SHA-256 Core

Resource Type	Utilization	Available	Utilization (%)
LUTs	14,752	53,200	27.70%
Flip-Flops	9,860	1,06,400	9.30%
BRAM (36Kb)	10	140	7.10%
DSP Slices	6	220	2.70%

The design achieved a maximum clock frequency of 100 MHz, corresponding to an effective data throughput of approximately 1.2 Gbps, depending on message length and UART latency.

These results demonstrate that the system achieves a balanced trade-off between area efficiency and computational performance, suitable for real-time cryptographic hardware applications.

4.3. MicroBlaze Soft Processor Implementation Results

1) System Overview

The MicroBlaze soft processor was designed using Vivado's Block Design environment. The architecture included:

- MicroBlaze 32-bit RISC core
- UART Lite peripheral for serial communication
- GPIO for LED and switch interfacing
- Local Block RAM for instruction and data storage
- AXI Interconnect for peripheral communication

Due to the graphical and block-based nature of the design, the complete system diagram

cannot be represented here. The interconnections were visually verified within Vivado to ensure correct AXI address mapping and clock synchronization.

2) Software Development and Testing

The hardware system was exported to Xilinx SDK/Vitis, where embedded C programs were developed for UART and GPIO testing.

Typical operations included:

- Sending and receiving text data via UART
- Controlling on-board LEDs based on switch inputs or received commands

3) Expected UART Interaction

A representative terminal session for the MicroBlaze system is as follows:

MICROBLAZE UART TEST INTERFACE

Enter Command: LED_ON

Response: LED 1 Turned ON

Enter Command: LED_OFF

Response: LED 1 Turned OFF

These expected results were validated in real-time within the ECIL R&D environment. LED toggling and UART data exchange confirmed that the processor's peripherals were configured and functioning correctly.

4) Resource Utilization

Table 2 - FPGA Resource Utilization for MicroBlaze System

Resource Type	Utilization	Available	Utilization (%)
LUTs	10,320	53,200	19.40%
Flip-Flops	6,245	1,06,400	5.80%
BRAM (36Kb)	14	140	10.00%
DSP Slices	0	220	0.00%

The design demonstrated low hardware overhead, leaving significant FPGA resources available for potential integration of additional peripherals or co-processors.

5) Observations

The MicroBlaze processor operated at 100 MHz, with stable UART communication at 115200 baud rate. GPIO access exhibited minimal latency, approximately 5 clock cycles, confirming responsiveness suitable for embedded control applications.

The project successfully demonstrated system-level design using AXI-based interconnects, memory-mapped I/O, and software-controlled peripherals on FPGA.

4.4. Comparative Analysis and Discussion

The two systems, although developed independently, collectively highlight the dual capabilities of FPGA technology — as both a custom hardware accelerator platform (for SHA-256) and a programmable embedded processor environment (for MicroBlaze).

- The SHA-256 implementation emphasizes parallelism, pipelined computation, and deterministic hardware execution, suitable for applications demanding high-speed cryptographic processing.
- The MicroBlaze processor, on the other hand, demonstrates flexibility, configurability, and rapid development, suitable for control logic and software-driven systems.

Both designs exhibited efficient utilization of FPGA resources and stable real-time performance during testing.

While actual output documentation was restricted due to ECIL's operational security policies, the expected and verified results reflect successful functional completion and hardware correctness under laboratory supervision.

4.5. Summary of Results

1. The SHA-256 hardware design achieved correct functional behavior, efficient area utilization, and reliable UART-based message interaction.

2. The MicroBlaze system was successfully instantiated using block design, integrated with UART and GPIO, and validated using C-based embedded programs.

3. All testing was conducted within ECIL's R&D facility under secure conditions; output data and screenshots could not be exported, but expected results were documented based on in-lab verification.

4. Both projects achieved their intended objectives and collectively provided comprehensive exposure to FPGA-based digital system design, hardware–software co-development, and real-world verification workflows.

5. CONCLUSIONS

The internship project successfully explored two major domains of FPGA-based system design: custom hardware implementation through the SHA-256 hashing algorithm, and embedded system development using the MicroBlaze soft-core processor. Both tasks were independently designed, synthesized, and tested on the Xilinx AC701 (Artix-7) FPGA development board using Vivado Design Suite 2018.3.

The SHA-256 implementation provided a deep understanding of cryptographic data flow, bitwise manipulation, and pipelined hardware computation. The design integrated an auto-padding mechanism and UART-based data interface, enabling end-to-end message hashing entirely on FPGA hardware. Functional verification was performed with standard test vectors, and all generated hashes matched expected NIST outputs, thereby confirming the accuracy of the implemented logic.

The MicroBlaze subsystem introduced a comprehensive understanding of processor-based design, AXI interconnect configuration, memory interfacing, and peripheral control using embedded C programming. The UART and GPIO modules were successfully tested through command-based user interactions, validating the system's real-time response and peripheral handling capabilities.

As the work was conducted within the ECIL CR&D facility, strict data security and confidentiality protocols restricted the use of external storage devices or mobile phones. Consequently, output screenshots and project files could not be exported. However, all system functionalities were thoroughly verified in-lab under supervision, and expected results have been documented in this report.

This project enabled the practical application of theoretical concepts learned during the undergraduate program — including digital logic design, computer architecture, and HDL-

based hardware development — and enhanced proficiency in FPGA prototyping, debugging, and performance optimization. The hands-on exposure to both hardware-level cryptographic design and processor-based embedded systems strengthened understanding of real-world hardware–software co-development workflows used in the electronics industry.

5.1.Future Scope

Building upon the outcomes of this project, several extensions can be pursued in future work:

- Integration of SHA-256 with MicroBlaze: Implementing a hybrid system where the MicroBlaze processor handles input/output operations while delegating hash computation to the custom hardware accelerator.
- Extension to SHA-512 or SHA-3: Enhancing the cryptographic module for higher security standards and benchmarking hardware performance trade-offs.
- Hardware–Software Co-Simulation: Using Xilinx Vitis or Vivado HLS to integrate C-based simulation with RTL modules for faster verification.
- IoT Security Applications: Deploying the SHA-256 hardware core in embedded IoT nodes for real-time authentication and secure data transfer.

In conclusion, this internship offered a comprehensive understanding of advanced FPGA design methodologies, industry-grade development tools, and secure project execution practices followed in professional CR&D environments such as ECIL. The dual exposure to both cryptographic hardware design and embedded processor implementation provided a strong foundation for future research and professional pursuits in VLSI, FPGA systems, and digital hardware security.

REFERENCES

A. Journals and Conference Papers

[1] C. E. B. Santos Júnior, "SHA-256 Hardware Proposal for IoT Devices in the Blockchain Context," Sensors, vol. 24, no. 5, 2024. [Online]. Available: https://www.mdpi.com

[2] R. Wu, C. Zhao, Y. Gao, and H. Zhang, "A High-Performance Parallel Hardware Architecture of SHA-256," in Proceedings of the 21st International Conference on Advanced Communication Technology (ICACT), 2019. [Online]. Available: https://icact.org

[4] Anonymous authors, "Project Reports and Tutorials Demonstrating UART-Integrated SHA-256 Implementations," International Research Journal of Modernization in Engineering, Technology and Science (IRJMETS), vol. 6, no. 8, 2023. [Online]. Available: https://irjmets.com

B. Dissertations and Technical Reports

[5] D. Sheldon, J. Lemieux, and K. Asanović, "Application-Specific Customization of FPGA Soft-Core Processors," Technical Report, Department of Computer Science and Engineering, University of California, San Diego, 2020. [Online]. Available: https://cseweb.ucsd.edu

C. Books

FPGA Prototyping by Verilog examples by Pong P. Chu, A John Wiley & Sons Inc. Publication

D. Websites and Online Repositories

[3] secworks, "sha256: Hardware Implementation of the SHA-256 (Verilog)," GitHub Repository, accessed Oct. 2025. [Online]. Available: https://github.com/secworks/sha256

[6] Xilinx/AMD, "Artix-7 FPGA AC701 Evaluation Kit," Product and User Documentation, 2023. [Online]. Available: https://www.amd.com

Appendix 1 – Internship Summary Sheet (Signed by Guide)

Internship Report

Intern Name: CHITRESH SHARMA
Internship Duration: 01 July 2025 – 31 July 2025
Mentor: I V KARTEEK, Dy.Mgr (Tech)
Project Guide: K. MADHURI, Sr. Dy. General Manager
Organization: CR&D, ECIL

Topic: Internship on Verilog Digital Design, AC 701 Board Projects, SHA - 256 Implementation and MicroBlaze Soft Processor

1. Introduction

The one-month internship focused on providing a hands-on training in VLSI - FPGA design using **Verilog HDL** and the **Xilinx Vivado** tool chain. I was introduced to the hardware description languages (verilog), digital logic design, and FPGA implementation workflows. The goal was to develop both theoretical knowledge and practical skills in FPGA design and verification.

2. Objectives

- To writing efficient Verilog code for various digital circuits.
- Learn the methodology of writing test benches and simulating designs in Vivado for functional verification.
- To be able to design, simulate, and implement digital circuits on FPGA.
- To learn the use of Vivado for synthesis, implementation, and bitstream generation.
- To have a hands-on experience deploying projects onto the AC701 board, including debugging and hardware verification.
- To develop practical skills by working on example projects such as LED blinking and UART communication.
- Get guidance in understanding and implementing the SHA-256 hashing algorithm in hardware.

3. Training Activities

Week 1:
Covered Verilog fundamentals, combinational and sequential logic design and writing simple modules. Intern demonstrated good understanding of syntax and logic design basics.

1. 2:1 MUX
2. 4:1 MUX using 2:1 MUX
3. 3:8 Decoder
4. Counter
5. Bi - Directional Counter
6. Shift Register - SISO, SIPO, PISO, PIPO.
7. D Flip Flop
8. Normal FSM -Traffic lights
9. Moore FSM
10. Mealy FSM
11. Sequence Detector
12. UART - TX & RX

Week 2:
Hands-on sessions included:

- Programming the FPGA with generated bitstreams.
- Interfacing with onboard peripherals such as LEDs, switches, and UART.

- Debugging hardware functionality using onboard tools and Vivado Integrated Logic Analyzer (ILA).
- Validating full project functionality with hardware testing.

Week 3:

Implementation of SHA-256 Algorithm

The SHA-256 cryptographic hash function was selected as an advanced project.

- Studied the algorithm and its hardware architecture.
- Developed Verilog modules for key operations (e.g., message scheduling, compression).
- Verified functionality with simulation testbenches.
- Synthesized and implemented the design on the AC701 board, testing the hash output with known inputs.

Week 4:

Introduction to Microblaze soft Processor

- Creating a basic MicroBlaze system and peripheral interfacing using Block Designs
- Hands-on with UART, GPIO via SDK on AC701 board

4. Skills Gained

- Gained hands-on experience with FPGA hardware and peripheral interfacing.
- Developed problem-solving skills by overcoming challenges related to timing and UART communication.

5. Conclusion

Overall, the internship was very useful in learning Verilog coding, simulation, and hardware implementation. The SHA-256 & MicroBlaze soft Processor project added valuable experience in cryptographic hardware design and software implementation of the projects using embedded c.

Project Guide

K. MADHURI
(Sr. Dy. General Manager)
CR&D, ECIL

के. माधुरी/K. MADHURI
वरिष्ठ उप महाप्रबंधक/Sr. Dy. General Manager
निगमित अनुसंधान एवं विकास, ईसीआईएल,
Corporate R&D, ECIL,
हैदराबाद/Hyderabad – 500062,

Appendix 2 – Internship Certificate (ECIL)

इलेक्ट्रॉनिक्स कॉरपोरेशन ऑफ इंडिया लिमिटेड

संदर्भ सं.Ref. No.128/2025-26

ELECTRONICS CORPORATION OF INDIA LIMITED

भारत सरकार (परमाणु ऊर्जा विभाग) का उद्यम

A GOVT. OF INDIA (DEPT. OF ATOMIC ENERGY) ENTERPRISE

कॉरपोरेट प्रशिक्षण एवं विकास केंद्र CORPORATE LEARNING & DEVELOPMENT CENTRE

हैदराबाद HYDERABAD – 500 062

आज़ादी का अमृत महोत्सव

प्रमाणपत्र Certificate

This is to certify that **Mr. CHITRESH SHARMA** S/o **Shri NAVAL KISHORE SHARMA** a Student of **B.TECH (ECE)** studying in **GURUNANAK INSTITUTIONS TECHNICAL CAMPUS, HYDERABAD, TELANGANA** has successfully completed Internship from **01.07.2025** to **31.07.2025** on **"SHA-256 IMPLEMENTATION WITH AUTO PADDING AND MICRO BLAZE CORE PROCESSOR"** in **CORPORATE RESEARCH AND DEVELOPMENT.**

This is issued as a partial fulfillment of his academic program/curriculum.

His performance during training was found to be **"Excellent"**.

दिनांक Date: 02.08.2025

(डॉ. राजनारायण अवस्थी)
(Dr. Raj Narayan Awasthi)
प्रबंधक एवं प्रभारी-सीएलडीसी
Manager & Incharge- CLDC
डॉ. राजनारायण अवस्थी Dr. Raj Narayan Awasthi
प्रबंधक (कॉरपोरेट प्रशिक्षण एवं विकास)
Manager (Corporate Learning & Development)