ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

DEPARTMENT OF
INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Spring Semester 2025

Implementation of a SHA-256 hardware accelerator for the Croc SoC

VLSI II Project

Nikola Tesic & Eric Meier
ntesic@student.ethz.ch, meieeric@student.ethz.ch

July 2025

Advisors: Frank K. Gürkaynak
Enrico Zelioli
Philippe Sauter

Professor: Prof. Luca Benini

Abstract

The increasing volume of sensitive data transmission and the growing demand for heterogeneous System on Chips have necessitated the design of a cryptographic hardware accelerator for the Secure Hash Algorithm 256 (SHA-256) for Croc. To achieve this, a SHA-256 hardware accelerator was designed and subsequently integrated into the Croc System-on-Chip [1]. This resulted in a functional SHA-256 accelerator, enabling Croc to execute hashing operations in 2.79 μs, 53x faster than with a software implementation. The required core area only increased by 23%.

Acknowledgments

We would like to express our gratitude to the assistants of the VLSI II course for their patience, guidance and encouragement. Their assistance with any issues we encountered was crucial for the completion of the VLSI II exercises and this project.

We would also like to thank our parents and friends for their emotional support and understanding throughout the project.

Contents

Chapter 1

Introduction

With the advancement of open-source methodologies in the hardware domain, it has become feasible to design complete microchips without relying on proprietary tools or Intellectual Properties (IPs)[2]. This development enables a significantly broader community, including researchers and universities, to access these technologies. Consequently, microchip design has become increasingly democratized[3].

Furthermore, the capability to develop microchips utilizing open-source tools and IPs facilitates enhanced customization and optimized performance through the freedom to modify every component of the design. This technological evolution has catalyzed the emergence of platforms such as Parallel Ultra Low Power (PULP), which focus on creating comprehensive open-source hardware and software components.

One of their most recent projects is Croc, a System-on-Chip (SoC) developed within the framework of the Very-Large-Scale Integration 2 (VLSI 2) course. This initiative allows students to contribute additional functionalities or enhance existing chip components.

Among the various technological possibilities, the increasing volume of sensitive data transmission and the growing demand for heterogeneous SoCs have necessitated the design of a cryptographic hardware accelerator for the Secure Hash Algorithm 256 (SHA-256). SHA-256 constitutes an irreversible cryptographic function, a necessary feature for secure communication protocols. The implementation described in this work incorporates deliberate limitations that were established to maintain manageable complexity within the scope of this work. These constraints, together with the operational principles of the SHA-256 algorithm, will be explained in Chapter 2.

Through the integration of this new functionality into the SoC, the system demonstrates potential utility in domains such as telecommunications, blockchain technology, and any context where ensuring signal integrity during transmission is paramount.

The main goals of this project are:

- **The design of a SHA-256 hardware accelerator**
- **Integration of the accelerator into the Croc SoC**
- **Performance evaluation of the accelerator**

Chapter 2

Background

This chapter presents the SHA-256 algorithm [4] and all necessary knowledge to understand its functionality.

2.1 Algorithm

Security Hash Algorithm 256 (SHA-256) is an algorithm that belongs to the SHA-2 family and ensures signal integrity in communications. This type of algorithm has the following properties:

- Irreversible: Given the output, it is not possible to retrieve the original input.
- The output has a fixed length, independently from the input.
- A small change in the input produces a completely different output.

In our case, the length of the algorithm's output, called the digest, is fixed at 256 bits. All algorithms belonging to the SHA-2 family share the same characteristics, differing only in the length of their output.

Figure 2.1 illustrates communication with SHA-256 . The sender and receiver agree on a SHA implementation. The sender transmits a message of arbitrary length, along with its digest, which serves as a fingerprint or digital identity of the message.

The receiver, upon receiving both the message and the digest, independently computes the digest of the received message. If the calculated digest matches the received one, the receiver can conclude that the message has not been altered during transmission. In Figure 2.1 the digests do not match because the message has been tampered with.

The main purpose of SHA-256 is therefore to ensure the integrity of the message, not its confidentiality.

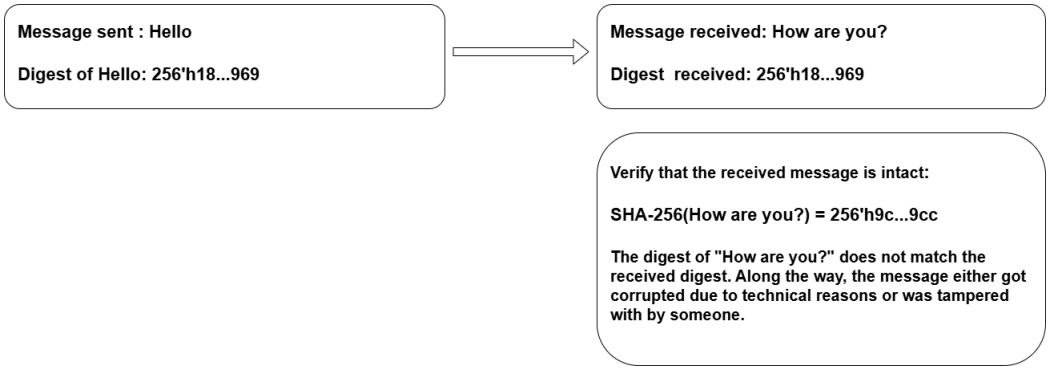

Figure 2.1: The image shows communication with SHA-356. The message "Hello" is sent along with its digest. The digest remains identical and the message "How are you?" is received. The received digest does not match the computed digest of the message, thus it can be concluded that the message was corrupted or altered.

2.1.1 Operating principle of SHA-256

The operational mechanism of the SHA-256 algorithm can be systematically categorized into the following fundamental processes: Input preparation, word expansion, hash computation, chunk transition, and iterative processing.

Input preparation:

The standard SHA-256 algorithm accommodates inputs of arbitrary length through padding mechanisms to achieve multiples of 512 bits. However, the hardware accelerator presented in this implementation operates exclusively on fixed-length inputs of 512 bits. It is assumed that the CPU core manages any padding operations when the input size deviates from the specified length, as this step lacks computational intensity and does not justify hardware acceleration.

Word Expansion:

Each chunk consists of 16 words, with each word being a 32-bit segment, extracted from the chunk. The initial 32 bits form the first word.

The initial chunk undergoes expansion to contain 64 words. The additional words are computed from existing ones according to the following function, with the index of the first word is being zero.

$$w[n] = w[n-16] + \sigma_0(w[n-15]) + w[n-7] + \sigma_1(w[n-2]) \quad \text{where } 15 < n < 64 \quad (2.1)$$

where σ_0 and σ_1 are defined as:

$$\sigma_0(w) = (w \ggg 7) \oplus (w \ggg 18) \oplus (w \gg 3) \tag{2.2}$$
$$\sigma_1(w) = (w \ggg 17) \oplus (w \ggg 19) \oplus (w \gg 10) \tag{2.3}$$

A fundamental characteristic of this implementation is that each computed value is dependent on preceding values, an aspect that appears throughout subsequent algorithmic steps.

Hashing:

Proceeding with the expanded chunk, the subsequent phase involves computation of the hash value. The hash values represent the initial 32 bits of the fractional components of the square roots of the first eight prime numbers, utilized at algorithm initialization. The implementation follows this structure:

$$h_7 = h_6 \tag{2.4}$$
$$h_6 = h_5 \tag{2.5}$$
$$h_5 = h_4 \tag{2.6}$$
$$h_4 = h_3 + Temp_1(n) \tag{2.7}$$
$$h_3 = h_2 \tag{2.8}$$
$$h_2 = h_1 \tag{2.9}$$
$$h_1 = h_0 \tag{2.10}$$
$$h_0 = Temp_1(n) + Temp_2(n) \tag{2.11}$$

The equations clearly identify the variables that most significantly influence hash value evolution. These two variables are defined as follows:

$$Temp_1(n) = h_7 + \Sigma_1 + Choice + k[n] + w[n] \tag{2.12}$$
$$Temp_2(n) = \Sigma_0 + Majority \qquad \text{where } n \in [0:63] \tag{2.13}$$

Only $k[n]$ represents a constant variable, determined by prime number derivatives. For example, $k[0]$ corresponds to the fractional component of the cube root of 2 (the first prime number), while $k[n]$ represents the fractional component of the cube root of the (n+1)th prime number. The remaining variables are computed as follows:

$$Choice = (h_4 \wedge h_5) \oplus ((\neg h_4) \wedge h_7) \tag{2.14}$$
$$Majority = (h_0 \wedge h_1) \oplus (h_0 \wedge h_2) \oplus (h_1 \wedge h_2) \tag{2.15}$$
$$\Sigma_1 = (h_4 \ggg 6) \oplus (h_4 \ggg 11) \oplus (h_4 \ggg 25) \tag{2.16}$$
$$\Sigma_0 = (h_0 \ggg 2) \oplus (h_0 \ggg 13) \oplus (h_0 \ggg 22) \tag{2.17}$$

Due to this continuous dependency on preceding values, even a single-bit modification results in a fundamentally different computational outcome. The implementation described above is iterated across the entire chunk. Upon completion of the final iteration (constrained by 64 words and 64 constants k), the initial hash value is summed with the final computed value.

Chunk computation:

The complete implementation described thus far applies exclusively to the initial chunk. This is sufficient if the input can be accommodated within a single chunk, otherwise, iterative continuation becomes necessary. The subsequent chunk is processed identically the previous chunk, with only the hash value computation being different. For the second chunk, the default hash values are no longer used, with the final value from the preceding chunk serving as the new default. Therefore the final iteration value is summed with the result from the previous chunk. After applying this process to every chunk, the final 256-bit hash value is obtained.

2.2 Croc

Croc is an SoC developed by the PULP team for educational purposes. It is designed to give students attending the VLSI II course at ETH the opportunity to work on a real practical project, thereby developing their skills in hardware design.

In this project a cryptographic hardware accelerator based on SHA-256 was developed. This approach ensures data integrity verification for all transmitted data while offloading computational burden from the main processor and achieving faster execution times compared to software-only implementations. For this reason, parts of the SoC that are relevant to this project is the interface with the USER domain, and the user domain itself, which starts at the address 0x2000_0000 Figure 2.3 shows a representation of the system.

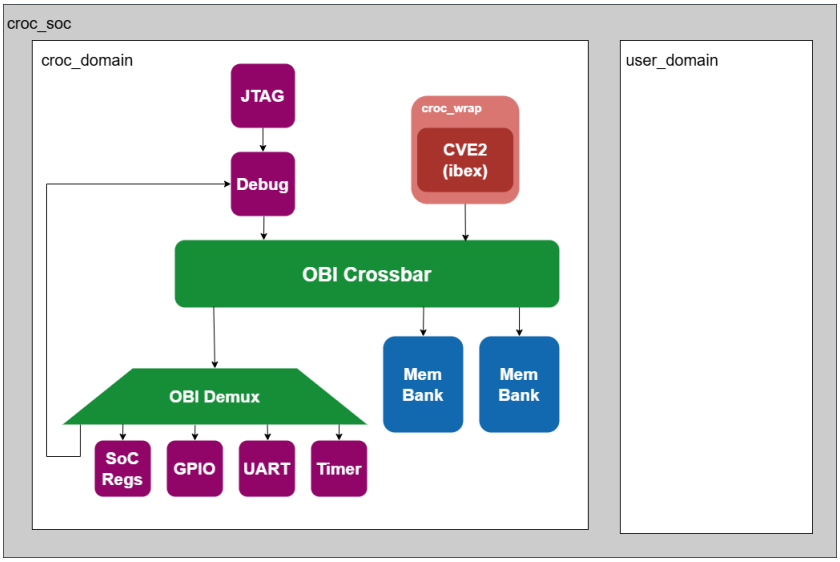

Figure 2.2: Hardware design of Croc

Chapter 3

Implementation for the VLSI II Project

Adding the SHA-256 cryptographic hardware accelerator in the USER domain of Croc involves two steps. The first consists of writing the RTL implementation of the SHA-256 code. The second is adding the structure needed to respect Croc's handshaking. The entire implementation is grouped into six steps: saving the input, hashing, iteration repetition, sum, chunk change, and sending the output to the testbench.

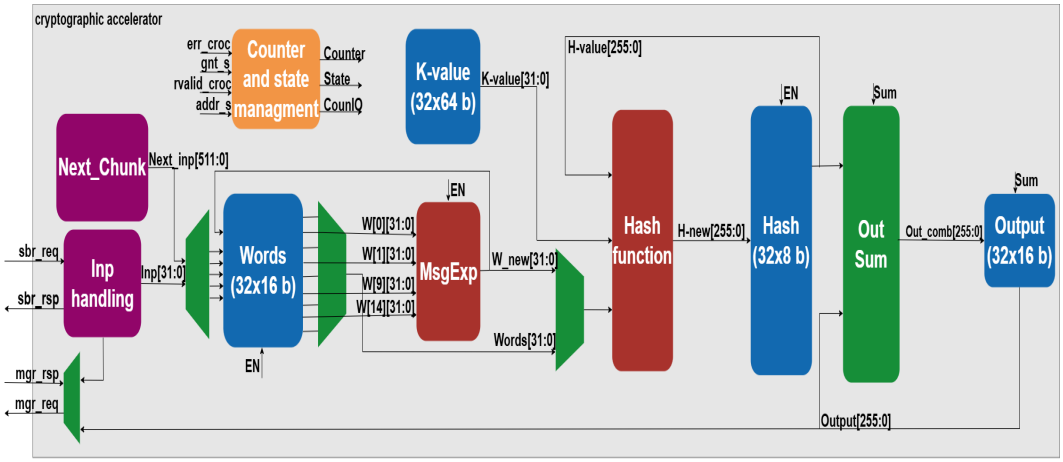

Figure 3.1: Hardware design of cryptographic SHA-256 HW accelerator

Input Saving:

The first objective consists of dividing the 512-bit input into 16 words and storing them all in memory.

As shown in Figure 3.1, a demultiplexer with two inputs is utilized. The current FSM state decides, which of the two input chunks to use. If it is in the Idle state, the 512 bit long input chunk is selected. If it is in the Next_chunk state, the bits contained in the

other chunk are selected.

Hashing:

A Carry Save Adder (CSA) is used in the accelerator to efficiently compute the sum of three binary numbers. It returns two outputs: the partial sum and the carry, which when added together produce the final sum of the three inputs.

Additionally, to avoid increasing the memory, a shift register has been implemented. Every time the enable signal (EN) is active, the values in the registers are shifted downward: $w[n] = w[n+1]$, while $w[15] = w\_new$ receives the newly calculated value.

The Hash Function module is described by Pseudocode 1.

Iterating:

In this section, the focus is on the control signals. The EN signal is responsible for controlling and coordinates the flow and is determined through the Finite State Machine. Two counters are present: one for input and output and another for the hash. As soon as the state becomes *Hashing*, the counter begins to increment. At the 64th iteration, EN returns to zero while the Sum signal is activated. This signal activates the Out Sum and Output memory components. This activation lasts exactly one cycle. At the same moment when the sum is performed, the state change from *Hashing* to *Next_Chunk* occurs. Since the state change requires one cycle, the counter advances and goes from 64 to 65. It remains in the *Next_Chunk* for exactly one cycle before returning to the *Hashing* state. Upon return, the counter is reset to zero and the entire flow that was just described is repeated.

To keep track of whether we are working on the first chunk or the second chunk, a one-bit memory is added that stores a signal called second_iteration.

Chunk Change:

When the state equals *Next_Chunk*, the active chunk must be replaced. The multiplexer, placed in front of the Words memory, allows another signal to write to memory. This new chunk is generated by a hardware component that always produces the same value, given that the input length is fixed.

The Output memory also updates the Hash memory. To accomplish this, the demultiplexer placed in front of the input of the previous hash-values is activated, as illustrated in Figure 3.1.

Summation:

The sum occurs, as previously indicated, at specific moments. The most relevant component is the demultiplexer with three inputs and one output. One of three sources, the hash memory, the result of the summation, or it is the memory itself can write to memory.

Pseudocode 1: Hash Function Module

1: **procedure** HASHFUNCTION($Words[31:0]$, $K\_value[31:0]$, $H\_values[31:0]$)
2: *// Variable declarations*
3: $S0xD, MajxD, CHxD, S1xD \in 2^{32}$
4: $CSAC1, CSAS1, CSAC2, CSAS2, CSAC3, CSAS3 \in 2^{32}$
5: $CSAC4, CSAS4, CSAC5, CSAS5, CSAC6, CSAS6 \in 2^{32}$
6: *// Hash operations*
7: $S0 \leftarrow (a \ggg 2) \oplus (a \ggg 13) \oplus (a \ggg 22)$
8: $Maj \leftarrow (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$
9: $Ch \leftarrow (e \wedge f) \oplus ((\neg e) \wedge g)$
10: $S1 \leftarrow (e \ggg 6) \oplus (e \ggg 11) \oplus (e \ggg 25)$
11: *// CSA operations*
12: $(CSAC1, CSAS1) \leftarrow CSA(Words, K_{values}, H_{values}[7])$
13: $(CSAC2, CSAS2) \leftarrow CSA(CSAC1, CSAS1, S1)$
14: $(CSAC3, CSAS3) \leftarrow CSA(CSAC1, CSAS1, CH)$
15: $(CSAC4, CSAS4) \leftarrow CSA(CSAC2, CSAS2, H_{values}[3])$
16: *// First assignment*
17: $H_{values}[4] \leftarrow CSAC4xD + CSAS4xD$
18: *// Additional CSA operations*
19: $(CSAC5, CSAS5) \leftarrow CSA(CSAC3, CSAS3, S0)$
20: $(CSAC6, CSAS6) \leftarrow CSA(CSAC4, CSAS4, Maj)$
21: *// Second assignment*
22: $H_{values}[0] \leftarrow CSAC6xD + CSAS6xD$
23: *// Hash values rotation*
24: $H_{values}[7] \leftarrow H_{values}[6]$
25: $H_{values}[6] \leftarrow H_{values}[5]$
26: $H_{values}[5] \leftarrow H_{values}[4]$
27: $H_{values}[3] \leftarrow H_{values}[2]$
28: $H_{values}[2] \leftarrow H_{values}[1]$
29: $H_{values}[1] \leftarrow H_{values}[0]$
30: **return** H_{new}
31: **end procedure**

The reason why the memory must write to itself is to guarantee stable synthesis by always providing a defined behavior at every cycle, avoiding undefined states in simulation and maintaining consistent RAM initialization.

Sending the output:

The final step consists of sending the calculation to the testbench (TB). When the counter reaches 64 and second_iteration is one, the sum is calulated and the state becomes *Output*. In this specific case, it is sufficient to compact the data into a single 256-bit signal and send it directly to the TB.

The TB receives the signal and compares it with the signal calculated by the golden model, ensuring that they are equal and confirming via the console output.

Testbench and golden model:

The TB serves to verify whether the implemented design is correct. It sends the input to the implementation, waits to receive the computed value, and subsequently compares it with the value calculated by the golden model.

3.1 Integration in the Croc SoC

3.1.1 Handshaking

Handshaking is a negotiation process that establishes a connection between two paties and ensures that both are ready and agree on a communication protocol.

Within the SoC, handshaking occurs with two types of signals, descibed in Table 3.1 and Table 3.2: One used by the hardware when acting as a manager, and the other when acting as a subordinate.

Registers:

The communication occurs through registers in the main memory. Inputs and outputs are saved to or loaded from the SRAM with registers in the user space used for communication between the two hardware components. Four registers are used with an address range from 0x2000_000C to 0x2000_00018, with each register containing 32 bits. The first address stores the SRAM address fot the input data, while the second specifies the output address. The third and fourth registers serve for the Core and accelerator to signal operation completion respectively.

The first three registers are reset by the accelerator when it is ready to receive new signals, while the fourth is handled by the Core.

Signals implementations:

For this section, Table 3.1 and Table 3.2 will be used to explain the individual signals used during communication.

Name	Bits	Description
req	1	Signals the availability of valid address phase signals.
addr	32	Specifies the address for the transaction.
we	1	Write enable signal (high for writes, low for reads).
be	4	Byte enable signal for specifying active bytes.
wdata	32	Write data (valid only for write transactions).
aid	5/1	Identifier for the address phase transaction.
a_optional	1	Optional signal for additional address phase information.

Table 3.1: Manager signals

Name	Bits	Description
gnt	1	Indicates readiness to accept an address transfer.
rvalid	1	Indicates the validity of response phase signals.
rdata	32	Read data (valid only for read transactions).
rid	5/1	Identifier for the response phase transaction.
err	1	Indicates an error during the transaction.
r_optional	1	Optional signal for additional response phase information.

Table 3.2: Subordinate signals

To send a request, the procedure is as follows: When the master is ready to send the signal, it sets req=1 and waits for the subordinate to respond with gnt=1 and rid==aid. The master hardware waits another cycle to receive the desired response with rvalid. Subsequently, depending on whether other signals need to be sent or not, it puts req to low.

The hardware needs to set req=0 after receiving rvalid=1. To avoid complications and to prevent data loss, req needs to be set to 1 for exactly 1 cycle, followed by gnt set to 1 for one cycle if the hardware is ready. When the transition is completed and req has returned to 0 it is also necessary to verify in the next cycle if Croc is available.

For the remaining bits, it is sufficient to consult the table and adapt them based on the operation to be performed, whether reading or writing data. The length of *aid* and of *rid*, varies with the accesses type (instruction fetch, data, atomic transactions, debug, etc.). The err bit in the subordinate activates when, during the request phase if the request is not supported by the module or *aid* is not correct.

It is important to keep in mind that the r_optional and a_optional signals must be implemented, but can be tied to zero. This is because if left undefined it could cause problems with the OBI.

3.1.2 DRC & LVF

During DRC validation a total of 320 errors were encountered, belonging to three types:

- 49 errors on the metal 2 layer (*M2.d*)
- 11 antenna errors on metal 3 and 5 (*Ant.b_Metal3* and *Ant.b_Metal5*)
- 260 *Lu.d* and *Lu.d1* errors.

The 49 *M2.d* errors were fixed manually, the others were not fixed and are further discussed in section 4.2.

Chapter 4

Results

In this chapter, the achieved design will be discussed. Croc 1.1 with and without the proposed accelerator implementation will also be compared, evaluating its advantages and disadvantages.

4.1 Names Stored in User Domain

The first names of the authors, Nikola & Eric, were permanently saved at the start of the user domain at the adderess 0x2000_0000. The string "Nikola&Eric" was saved as an ASCII zero-terminated string. The raw hex value of this string is 0x4E696B6F6C61264572696300 and can be read via UART. The storage and readout were confirmed to work by the authors.

4.2 DRC Errors

The 49 encountered M2.d DRC violations were more numerous than what the VLSI II exercises indicated. Despite this, all 49 errors were fixed manually.

The Ant.c Antenna violations were caused by OpenRoad not parsing the IHP rules properly [5]. Since the fix for this issue has not been merged and upstreamed by the OpenRoad team, the errors were left as is.

The 260 Lu.d and Lu.d1 errors are described as *"Max. Extension of NWell tie Activ tie beyond Cont"*. These errors, within and around the bonding pads, were probably caused by the wrong GDS file being used for the pads [6]. This error also caused the Contact layer to be empty and thus for the LVS to not match. A fix for this was not provided by time of writing and the errors remain.

4.3 Comparison

Figure 4.1 and Figure 4.2 illustrate the difference between CROC 1.1 with and without the SHA-256 accelerator.

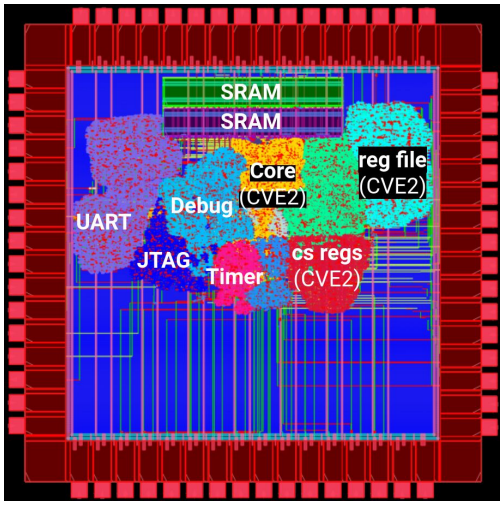

Figure 4.1: Baseline Croc 1.1

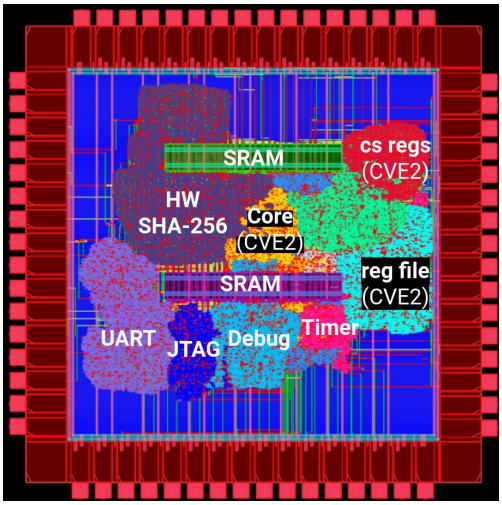

Figure 4.2: Croc 1.1 with the accelerator

Metric	Croc 1.1 Base	Croc 1.1 w. Acelerator	Improvement
Frequency (MHz)	77.64	77.82	1.002x faster
Cycle (cycles)	11683	217	53.84x faster
Time (μs)	150.47	2.788	53.97x faster
Area (kGE)	95.96	124.541	1.23x larger

Table 4.1: Performance comparison between the Croc SoC with and without the accelerator

Before proceeding with the analysis of the results, it is necessary to make three important clarifications.

The first concerns the measurement methodology: The code for running the SHA-256 directly on the CROC core is not fully optimized. Three different SHA-256 software implementations were adapted to work with the CROC Soc, further optimized and then tested. The fastest code used 11683 cycles to run the testbench. Further optimization, utilizing compiler optimization techniques, algorithmic optimizations specific to the RISC-V architecture, and fully exploiting the processor pipeline, could significantly reduce the number of required cycles. The current calculated improvements therefore overestimate the performance of the hardware accelerator.

The second clarification regards the backend design flow. During this phase, some signals did not meet timing constraints. After identifying the problematic signals and

analyzing their paths, the floorplanning was optimized by separating the two SRAM banks as seen in Figure 4.2.

The final clarification concerns the frequency measurement methodology. To measure the obtained value, a parasitic estimate is used, which is derived from average values on the routing layer and the routing distance. No coupling or other complex effects are considered in this approach. This simplified method is used instead of a parasitic extraction engine (PEX), which is significantly more complex and requires a separate calibration file downloaded from the openroad-flow-scripts repository. This methodology was chosen based on the PULP group's experience with Basilisk [7], where they found that this simplified parasitic estimation approach approximates the final frequency value with approximately 10% error.

Analyzing Table 4.1, the required active area has increased by a factor of 1.3. It is important to emphasize that Gate Equivalent (GE) represents the ratio between the active area and the reference unit area, in this case of NAND2x1. Base CROC 1.1 showed a core utilization of 34.6% while the CROC 1.1 with the accelerator showed 44.9%.

The results demonstrate that the proposed implementation provides an improvement in terms of cycles and execution time of around 53x, while requiring an increase in active area. It it is believed that this trade-off remains satisfactory even with a more optimized code, provided it exceeds a 5x improvement in terms of time and execution cycles.

4.4 Possible Improvements

The implementation presents room for improvement, particularly regarding MGE reduction. One possible improvement concerns the reduction of memory usage. The implementation requires approximately 3220 bits in total, of which 2048 are dedicated exclusively to storing K-values. An alternative strategy could consist of implementing the cube root mathematical function in hardware, allowing storage of only the first 64 prime numbers. However, it is not guaranteed that this implementation would result in a reduction of the overall active area.

Furthermore, comparing the energy used by the software and hardware implementation for a fixed amount of would be interesting as well.

Chapter 5

Conclusion and Future Work

The implementation of the SHA-256 hardware accelerator demonstrates a good trade-off between performance and area. The main results are:

- A functional SHA-256 accelerator has been designed in SystemVerilog
- The optimistic performance performance improvement of 53.8x better than the pure software implementation.
- The design has been synthesized, placed and routed in the IHP130nm technology using open source tools, achieving an operating frequency of 78MHz

With an additional 28.581 kGE in the total chip area (23% increase), the design reduces the execution time for SHA-256 by up to 53.8x. Thanks to this implementation, the Croc SoC will be able to execute SHA-256 operations in significantly reduced time when necessary, while maintaining the ability to handle other operations in parallel.

Bibliography

[1] P. Sauter, T. Benz, P. Scheffler, H. Pochert, L. Wüthrich, M. Povišer, B. Muheim, F. K. Gürkaynak, and L. Benini, "Croc: An end-to-end open-source extensible risc-v mcu platform to democratize silicon," 2025. [Online]. Available: https://arxiv.org/abs/2502.05090

[2] P. Scheffler, P. Sauter, T. Benz, F. K. Gürkaynak, and L. Benini, "Basilisk: An end-to-end open-source linux-capable risc-v soc in 130nm cmos," 2024. [Online]. Available: https://arxiv.org/abs/2406.15107

[3] Tiny Tapeout Team, "Tiny tapeout: From idea to chip design in minutes," https://tinytapeout.com/, 2024, accessed: 2025-06-13. [Online]. Available: https://tinytapeout.com/

[4] D. Martin, "Sha-256 algorithm explanation," 2022, accessed: 2025-06-13. [Online]. Available: https://github.com/dmarman/sha256algorithm

[5] V. I. Team, "Vlsi ii moodleoverflow question: Ant.c drc rule violation." [Online]. Available: https://moodle-app2.let.ethz.ch/mod/moodleoverflow/discussion.php?d=10447

[6] ——, "Vlsi ii moodleoverflow question: Lu.d drc errors." [Online]. Available: https://moodle-app2.let.ethz.ch/mod/moodleoverflow/discussion.php?d=10462

[7] P. Sauter, T. Benz, P. Scheffler, M. Povišer, F. K. Gürkaynak, and L. Benini, "Basilisk: A 34 mm2 end-to-end open-source 64-bit linux-capable risc-v soc in 130nm bicmos," 2025. [Online]. Available: https://arxiv.org/abs/2505.10060

[8] H. Kaeslin, *Top-Down Digital VLSI Design: From Architectures to Gate-Level Circuits and FPGAs*. Burlington, MA: Morgan Kaufmann, 2014, volume 1: Front-End Design.

[9] F. K. Gürkaynak, K. Gaj, B. Muheim, E. Homsirikamol, C. Keller, M. Rogawski, H. Kaeslin, and J.-P. Kaps, "Hardware evaluation of SHA-3 3rd round candidates," https://iis-people.ee.ethz.ch/~sha3/, 2016, accessed: 2025-06-14.