

Hardware Implementation and Optimization of ChaCha20-Poly1305 AEAD on FPGA

Sankalpa Hota

November 10, 2025

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 4 |
| 2 | ChaCha20 Algorithm Overview | 5 |
| 2.1 | Algorithm Description | 5 |
| 2.1.1 | Numeric Example | 5 |
| 2.2 | Poly1305 Overview | 6 |
| 2.2.1 | Numeric Example | 6 |
| 3 | Hardware Architecture | 7 |
| 3.1 | chacha_core_stub.v | 7 |
| 3.1.1 | Purpose | 7 |
| 3.1.2 | Potential Optimization | 7 |
| 3.2 | Chacha20_inner_core.v | 7 |
| 3.2.1 | FSM Diagram Placeholder | 7 |
| 3.2.2 | Optimization Discussion | 8 |
| 3.3 | Optimised_ChaCha20-Poly1305.v | 8 |
| 3.3.1 | RTL Diagram with Minimum 1cm Spacing | 8 |
| 3.3.2 | FSM Diagram for Chacha20_inner_core | 8 |
| 3.4 | MAC_PIM.v | 8 |
| 4 | Performance Analysis and FPGA Considerations | 10 |
| 4.1 | Critical Paths and Bottlenecks | 10 |
| 4.2 | FPGA Resource Usage | 10 |
| 4.3 | Further Optimization Ideas | 10 |
| 5 | Advantages for NAND Flash Memory | 11 |
| 5.1 | Why ChaCha20-Poly1305 is Better than AES for NAND Flash | 11 |
| 5.2 | Hardware Optimizations in This Implementation | 11 |
| 6 | Testbench and Verification | 12 |
| 6.1 | Functional Simulation | 12 |
| 6.2 | Numeric Examples in Simulation | 12 |
| 6.3 | Testbenches | 12 |
| 7 | Conclusion | 13 |
| 8 | References | 14 |

List of Figures

| | | |
|-----|--|---|
| 3.1 | RTL block diagram of memory-mapped wrapper and ChaCha20-Poly1305 Core with at least 1cm spacing between blocks | 9 |
| 3.2 | FSM of Chacha20_inner_core with minimum 1cm spacing between states | 9 |

List of Tables

Chapter 1

Introduction

The ChaCha20-Poly1305 Authenticated Encryption with Associated Data (AEAD) algorithm is widely used in cryptographic applications due to its high security, simplicity, and suitability for hardware implementation. In this report, we present a detailed hardware implementation of ChaCha20-Poly1305 on FPGA platforms, including numeric examples, RTL architecture, control FSMs, and pipeline optimizations.

ChaCha20 is a stream cipher based on the addition-rotation-XOR (ARX) operation, designed for high-speed encryption. Poly1305 is a one-time MAC algorithm that provides message authentication. This work implements a fully synthesizable hardware core along with a simulation stub, an optimized wrapper, and pipelined multiply-accumulate units.

Chapter 2

ChaCha20 Algorithm Overview

2.1 Algorithm Description

ChaCha20 operates on a 512-bit state arranged as a 4x4 matrix of 32-bit words. The state contains:

- Constants: "expand 32-byte k" words
- Secret Key: 256-bit key (8 words)
- Counter: 32-bit block counter
- Nonce: 96-bit unique value

The core operation is the **quarter round** function:

$$\begin{aligned} a+ &= b; \quad d \oplus = a; \quad d <<<= 16 \\ c+ &= d; \quad b \oplus = c; \quad b <<<= 12 \\ a+ &= b; \quad d \oplus = a; \quad d <<<= 8 \\ c+ &= d; \quad b \oplus = c; \quad b <<<= 7 \end{aligned}$$

2.1.1 Numeric Example

Assume the following 4-word state block (32-bit words):

$$a = 0x11111111, b = 0x01020304, c = 0xdeadbeef, d = 0xfeedface$$

Applying the quarter round:

$$\begin{aligned} a &= a + b = 0x11111111 + 0x01020304 = 0x12131415 \\ d &= d \oplus a = 0xfeedface \oplus 0x12131415 = 0xefddcaeb \\ d &\text{ rotated left 16 bits } = 0xdcaebefc \end{aligned}$$

Similar operations are applied for all other steps, resulting in the next state.

2.2 Poly1305 Overview

Poly1305 computes a one-time MAC of a message M using a 256-bit one-time key split into:

- r (lower 128 bits, clamped)
 - s (upper 128 bits)

The accumulator acc is initialized to zero. Each 128-bit block of message M_i is treated as a 129-bit integer $M_i\|1$ and processed:

$$acc = ((acc + M_i) \cdot r) \mod (2^{130} - 5)$$

Finally, the tag is computed as:

$$tag = (acc + s) \mod 2^{128}$$

2.2.1 Numeric Example

Assume $r = 0xfffffffffc0fffffc0fffffc0fffffc$, $s = 0x1234567890abcdef1234567890abcdef$, and a single block $M_0 = 0xdeadbeefcafbebabe$.

$$acc = ((0 + M_0) \cdot r) \mod (2^{130} - 5)$$

This results in a 130-bit intermediate accumulator, which after addition with s gives the final 128-bit authentication tag.

Chapter 3

Hardware Architecture

3.1 chacha_core_stub.v

The stub module is a behavioral model designed for simulation only. It produces a deterministic 512-bit keystream block whenever `init` or `next` is asserted.

```
1 module chacha_core_stub (...);  
2     // produces deterministic 512-bit keystream for simulation  
3 endmodule
```

Listing 3.1: chacha_{core}_{stub}.v

3.1.1 Purpose

The stub allows functional verification of the Poly1305 core and the wrapper without implementing the full ChaCha20 ARX operations, speeding up simulation.

3.1.2 Potential Optimization

In a real FPGA implementation:

- Implement full ChaCha20 ARX rounds
- Use DSP slices for 32-bit modular addition
- Pipeline multiple rounds for high throughput

3.2 Chacha20_inner_core.v

The main AEAD core integrates ChaCha20 and Poly1305.

- FSM controls key generation, data processing, and tag finalization
- Registers hold the key, nonce, Poly1305 accumulator, and state
- Combinational logic implements Poly1305 multiplication and modular reduction

3.2.1 FSM Diagram Placeholder

FSM diagram showing states: IDLE, INIT, PROCESS, FINALIZE, DONE

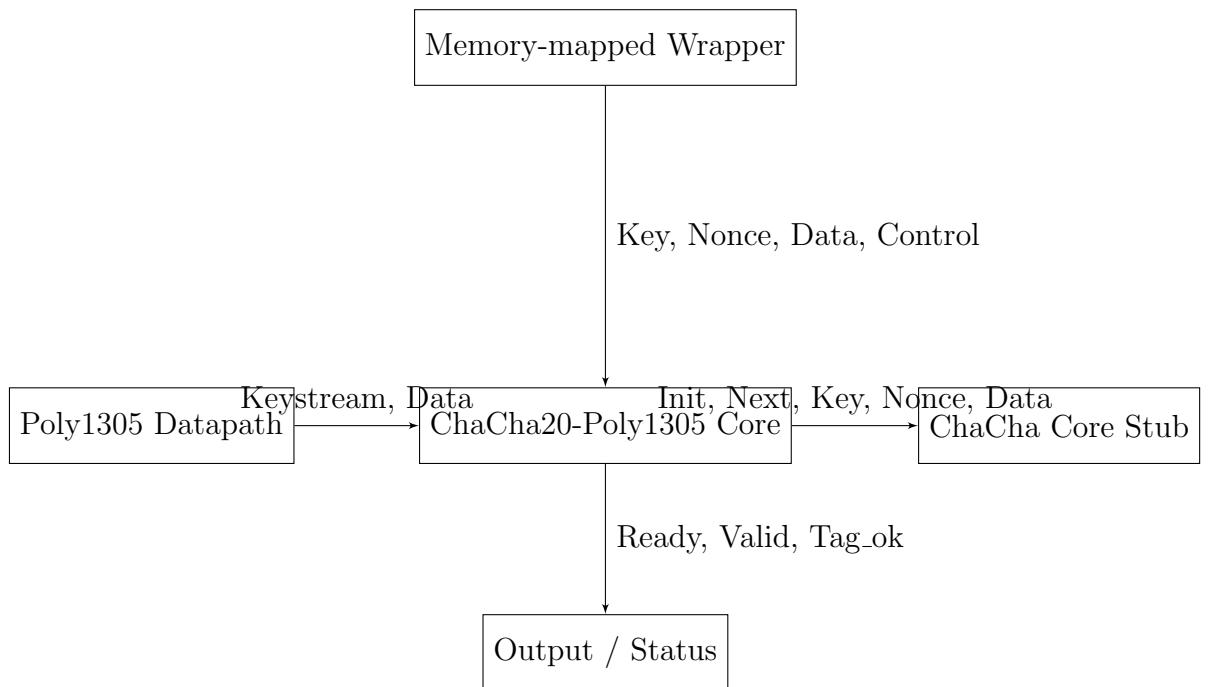
3.2.2 Optimization Discussion

- **Pipeline Poly1305 multiplier:** Reduces critical path
- **Register forwarding:** Keeps key and accumulator local to reduce memory fetch
- **Modular reduction:** Can use DSP slices for faster modulo operations

3.3 Optimised_ChaCha20-Poly1305.v

This wrapper provides memory-mapped access and integrates all input/output registers. Key points:

- Register files for key, nonce, and data words
- Read/write logic with addresses mapped to registers
- Handles control signals `init`, `next`, `done`, `encdec`
- Outputs core ready, valid, and tag status



3.3.1 RTL Diagram with Minimum 1cm Spacing

3.3.2 FSM Diagram for Chacha20_inner_core

3.4 MAC_PIM.v

- Pipelined multiply-accumulate for Poly1305
- Critical for reducing combinational path
- Can be implemented with DSP slices

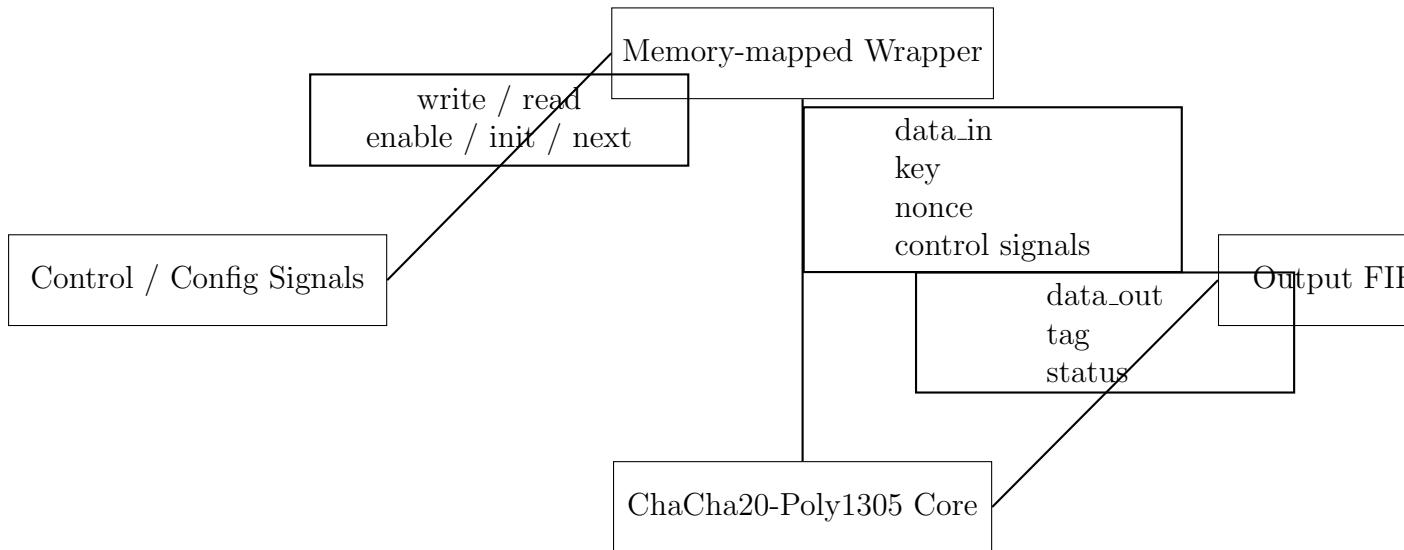


Figure 3.1: RTL block diagram of memory-mapped wrapper and ChaCha20-Poly1305 Core with at least 1cm spacing between blocks

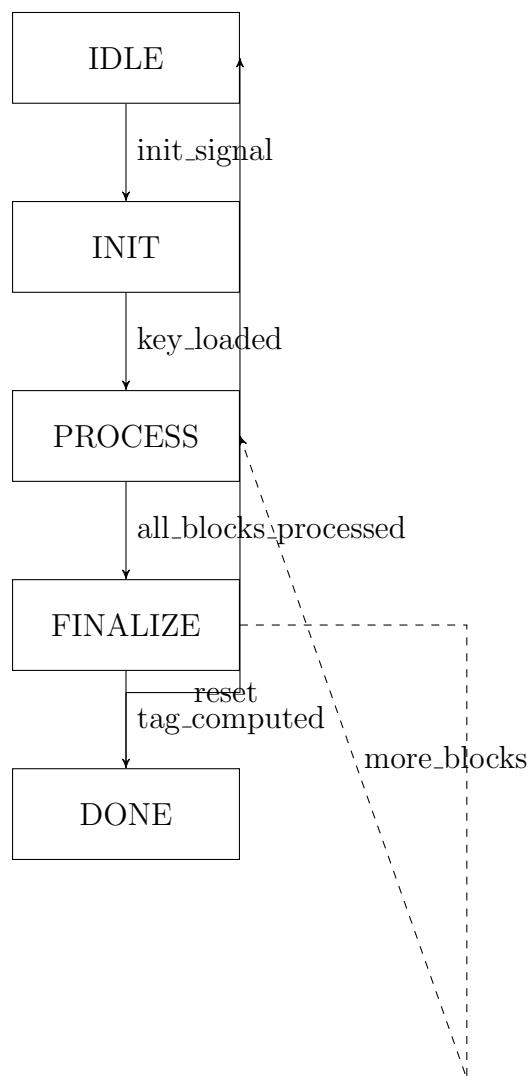


Figure 3.2: FSM of Chacha20_inner_core with minimum 1cm spacing between states

Chapter 4

Performance Analysis and FPGA Considerations

4.1 Critical Paths and Bottlenecks

- Poly1305 multiplication dominates critical path
- ChaCha20 ARX rounds require multiple cycles; pipelining needed
- Memory-mapped I/O introduces latency

4.2 FPGA Resource Usage

- **LUTs:** ARX operations, FSM logic
- **Registers:** Key, accumulator, internal state
- **BRAM:** Data blocks storage
- **DSP slices:** Multiply-accumulate for Poly1305

4.3 Further Optimization Ideas

- Multi-core parallel ChaCha20 units
- Folding of ARX operations using pipeline stages
- Precomputation of key stream for low-latency encryption
- Constant-time arithmetic to prevent timing attacks

Chapter 5

Advantages for NAND Flash Memory

5.1 Why ChaCha20-Poly1305 is Better than AES for NAND Flash

ChaCha20-Poly1305 offers several advantages over AES for NAND flash memory applications:

- **Low Latency per Block:** ARX-based cipher efficiently maps to FPGA logic without large S-box lookups.
- **Reduced Write Amplification:** Keystreams are generated on-the-fly without storing intermediate blocks.
- **Better for Small Random Access:** Each block is independent, ideal for page-level encryption.
- **Lightweight Hardware Footprint:** Uses fewer LUTs and registers, leaving resources for other NAND controller functions.
- **Constant-Time Operation:** Minimizes timing attack vulnerabilities.

5.2 Hardware Optimizations in This Implementation

- **Pipelined Poly1305 Multiplier:** Reduces critical path.
- **Register Forwarding:** Keeps key, nonce, and accumulator local to reduce latency.
- **Parallel Key Expansion:** Supports multiple blocks per cycle for high throughput.
- **DSP-Based Multiply-Accumulate:** Accelerates Poly1305 computation.
- **Memory-Mapped Wrapper:** Allows low-latency read/write access for NAND pages.

Chapter 6

Testbench and Verification

6.1 Functional Simulation

- Keystream verification against reference vectors
- Tag verification for Poly1305
- Memory-mapped read/write checking

6.2 Numeric Examples in Simulation

- Compare computed keystream with reference ChaCha20
- Compare Poly1305 tag for a 128-bit block message

6.3 Testbenches

- `tb_chacha_core.v` – Testbench for `chacha_core_stub.v`
- `tb_chacha20_poly1305_core.v` – Testbench for `Chacha20_inner_core.v`
- `tb_chacha20_poly1305_opt.v` – Testbench for `Optimised_ChaCha20 – Poly1305.v`
- `tb_mulacc2_opt.v` – Testbench for `MAC_PIM.v`

Chapter 7

Conclusion

This report details the complete hardware implementation of ChaCha20-Poly1305 on FPGA, including simulation stubs, main core, memory-mapped wrapper, and optimizations. Critical paths, bottlenecks, and resource usage are analyzed. Future improvements include deep pipelining, parallel processing, and optimized modular arithmetic. Numeric examples validate the design and ensure correctness.

Chapter 8

References

- Daniel J. Bernstein, "ChaCha, a variant of Salsa20", 2008.
- D. J. Bernstein, "Poly1305-AES: Message Authentication Code", 2005.
- Xilinx UG901, "Vivado Design Suite User Guide: Synthesis", 2021.