

FPGA ASCON Accelerator

Arul Kolla

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
arul@mit.edu

Tristan Kay

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
tristan@mit.edu

Abstract—In this paper we present a design on FPGA fabric for an hardware accelerator ASCON-128, a cryptographic protocol for authenticated encryption and decryption. Using a custom SPI communication scheme to interface with a Teensy 4.0, the controller is able to send approximately 8 MB/s to encrypt or decrypt on the FPGA.

I. INTRODUCTION

ASCON is a family of lightweight authenticated encryption and hashing schemes selected in the NIST Lightweight Cryptography standardization process. In this work, we design an FPGA accelerator for the ASCON-128 AEAD variant, with the goal of achieving high throughput while minimizing power usage.

At a high level, the accelerator is partitioned into two main groups of modules:

- the SPI communication module, which interfaces with an external controller, parses message types, performs basic error checking, and encapsulates payloads;
- the AEAD module, which implements the ASCON-128 authenticated encryption and decryption datapath, including a controller, nonce generator, ASCON permutation core, and submodules for initialization, associated-data processing, payload processing, and finalization.

We target the ASCON-128 parameters with a 128-bit key, 64-bit rate (block size), and two variants of the ASCON- p permutation: a 12-round version for initialization and finalization, and a reduced-round version for intermediate processing.

Fig. 1 shows the official ASCON encryption and decryption structure that our accelerator follows.

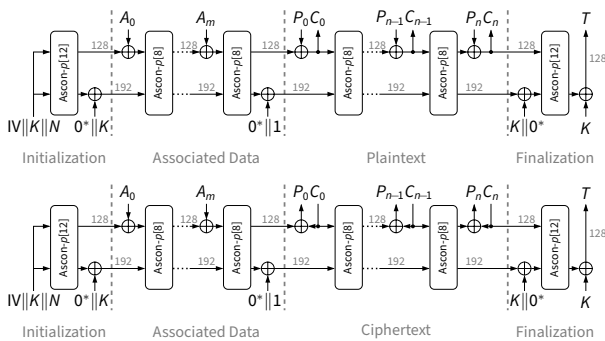


Fig. 1. Reference ASCON AEAD encryption (top) and decryption (bottom) schemes.

II. SYSTEM ARCHITECTURE AND BLOCK DIAGRAM

Fig. 2 shows the current top-level organization. The external device acts as the SPI controller, while the FPGA implements the SPI peripheral and the AEAD core. The SPI front-end decodes message types, manages checksums, and forwards the relevant payload and control signals to the AEAD module. The AEAD controller orchestrates the ASCON initialization, associated-data absorption, payload processing, and finalization, reusing a single permutation core.

The following sections describe the communication protocol and the internal AEAD datapath in more detail.

III. COMMUNICATION MODULE

A. SPI Physical Implementation

In our implementation, our FPGA will have a 8-bit wide bus for both COPI and CIPO, with a single line each for the CS and DCLK. This allows us to communicate 1 byte instead of 1 bit per DCLK cycle in each direction, increasing the overall communication throughput, which is the current bottleneck in throughput.

B. SPI Communication Protocol

The FPGA implements the peripheral side of a standard SPI interface. We distinguish data flowing from the device to the FPGA (D→F, over COPI) and from the FPGA to the device (F→D, over CIPO). In the baseline design, COPI and CIPO are not required to carry unrelated payloads simultaneously: F→D responses are generally coupled to the most recent D→F request, so the interface behaves like a request–response channel. If future use-cases demand full-duplex streaming, the design can be extended with deeper FIFOs and explicit flow control.

On the D→F path, the device sends typed message blocks that carry control information and payload data. The main D→F message types are:

- **New encryption block:** indicates the start of a new encryption message.
- **New decryption block:** indicates the start of a new decryption message and includes the nonce.
- **End encryption block:** indicates the end of the encryption.
- **End decryption block:** indicates the end of the decryption message, along with sending the tag.
- **Key block:** carries the 128-bit key.

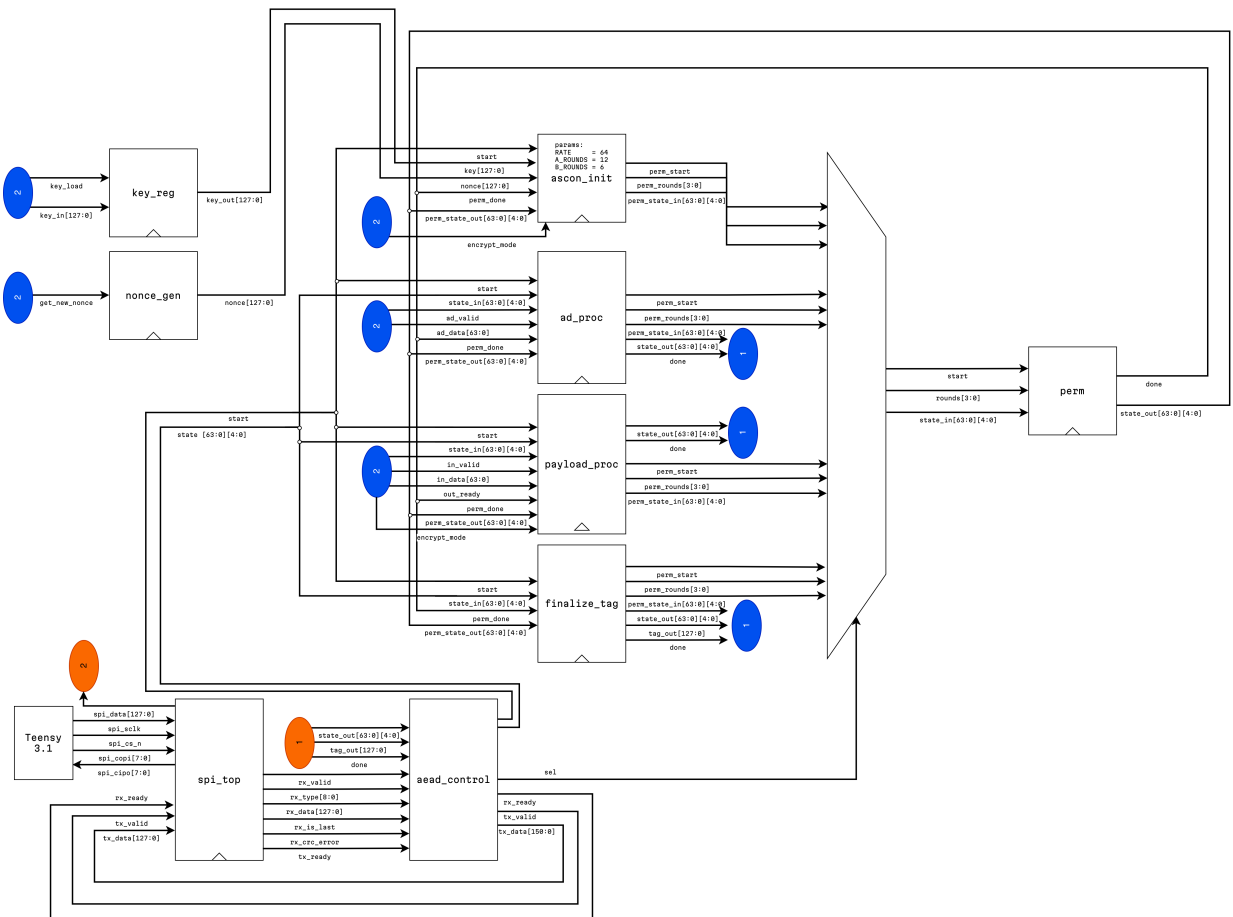


Fig. 2. Block diagram of the proposed system. “Portal 2” aggregates the signals shown at the SPI peripheral output; for clarity some internal signals are grouped into `spi_data[150:0]`.

- **Authenticated data (AD) blocks:** associated data in 64-bit chunks. Because no immediate response is required, two AD blocks may be packed into a single SPI transaction if desired.
 - **Ciphertext/plaintext blocks:** 64-bit payload blocks.
- On the F→D path, the FPGA responds with:
- **Nonce blocks:** the 128-bit nonce generated for encryption.
 - **Ciphertext/plaintext blocks:** the transformed payload.
 - **Tag block:** The 128-bit tag
 - **Verification block:** A single byte indicating whether the tag matched.
 - **Error block:** a dedicated message indicating checksum failure or a protocol violation (such as attempting to start a new message before the previous one has ended).

Table I summarizes the nominal payload sizes for each request–response pair, excluding type and checksum overhead.

The total number of bytes per SPI transaction is roughly constant across message types (around 16-17). This allows us to have relatively constant SPI message lengths. In a typical transaction, the controller first sends the D→F type field so that the peripheral can determine whether it needs to



Fig. 3. Communication Protocol CIPO and COPI

return data, then streams the payload and checksum. Then, the peripheral will send the F $\rightarrow$ D field, payload, and checksum back in turn. This is shown in Figure 3. Importantly, ciphertext/plaintext blocks are produced directly from the state via XOR rather than after a full permutation call, so the latency from state-read to SPI output is only a few clock cycles. Implementation-wise, the SPI module is a FSM that walks across the various stages indicated in Figure 3, and at the end, waits until CS comes back up.

The actual overhead of a block is 1 byte receiving header, 1 byte for receiving checksum, 1 byte for sending header, 1 byte for sending checksum, and a payload of up to 17 bytes, for a total of 21 bytes. If the FPGA is busy for one iteration, this could become 22 bytes, so we send SPI blocks in 22 byte increments.

D→F message	F→D message	D→F size (bytes)	F→D size (bytes)
New encryption block	Nonce block	0	16
New decryption message block	N/A	16	0
End message block (enc.)	Tag block	0	16
End message block (dec.)	Validation block	16	1
Key block	N/A	16	0
Authenticated data block	N/A	8/16	0
Ciphertext/plaintext block	Ciphertext/plaintext block	8	8

TABLE I
FPGA COMMUNICATION RESPONSE BY MESSAGE TYPE.

C. Checksum

We use an 8-bit checksum implemented as a CRC-8 via a linear feedback shift register (LFSR). A 32-bit checksum would significantly increase overhead for the relatively short messages we consider (on the order of 64 or 128 bits), while a 4-bit checksum would incur too high a probability of undetected errors. The specific generator polynomial is $x^8 + x^2 + x + 1$. Additionally, noting a LFSR does not have any conditional logic, we can collapse computation for 8 bits into a single round as several XORs without issue, as to allow for faster DCLK speeds.

D. SPI Controller

To implement the controller side to send and receive information from the FPGA, we used a Teensy 4.0, which in turn was communicating with our laptop with a USB protocol. A Teensy 4.0 can theoretically receive up to 480 Mbits/sec through USB Serial, but in practice, the communication is much less because of the laptop side of things. Thus we tried to optimize the communication from the laptop to the Teensy to some extent first (had the Teensy echo output from Python and tested what maximized speed, which included separate threads for reading and writing to Serial), and got a communication speed around 25-30 MB/s. Then, we added in the SPI communication to the loop. The initial plan was to try to optimize these reads and writes as well with MMIO instead of `DigitalWrite/DigitalRead`, but strangely, without this optimization, the Teensy was sporadically achieving very fast communication speeds. This will be discussed more in results. It should be noted that in the programming, the Teensy doesn't change its output depending on the response from the FPGA (since we figured such logic could drastically slow the Teensy down), and instead, sends the result of everything to the laptop for processing.

IV. AEAD CORE: AUTHENTICATED ENCRYPTION AND DECRYPTION

A. AEAD Controller

At a high level, the AEAD controller is a finite state machine (FSM) that sequences the ASCON-128 AEAD algorithm,

arbitrates access to the permutation core, maintains the 320-bit internal state $S[0..4]$, and interfaces to the SPI front-end.

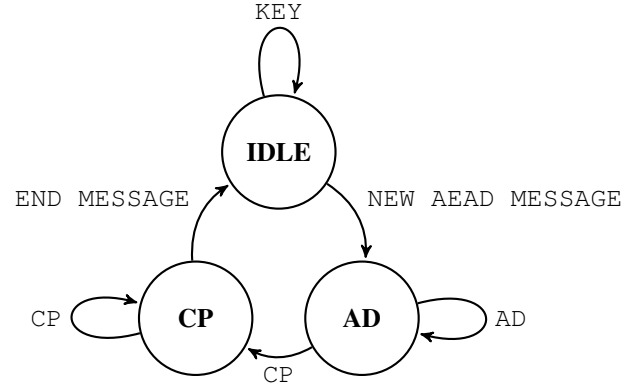


Fig. 4. FSM for the AEAD controller.

The controller has three main states, summarized below.
IDLE:

- **Key block:** Latch the 128-bit key into a key register and remain in **IDLE**.
- **New encryption block:** Request a fresh 128-bit nonce from the nonce generator, return this nonce to the device via SPI, and invoke the initialization module with $\{K, N\}$. Load the resulting state S and move to **AD**.
- **New decryption block:** Receive the 128-bit nonce from SPI, invoke initialization with $\{K, N\}$, load S , and move to **AD**.

AD:

- **Authenticated data blocks:** Buffer incoming associated data bytes into 128-bit chunks. For each full or final padded chunk, XOR the lower 64 bits into $S[0]$ and the upper 64 bits into $S[1]$, then run a reduced-round permutation (e.g., 6 or 8 rounds). While only AD blocks arrive, remain in **AD**.
- When the first ciphertext/plaintext block or an end-of-message block arrives, finalize the AD phase by flipping the domain-separation bit in $S[4]$ (XOR with $1 \ll 63$) and either move to **CP** or, if there is no payload, proceed directly to finalization.

CP:

- **Ciphertext/plaintext blocks:** Buffer incoming payload into 64-bit units. In encryption mode, XOR plaintext into $S[0]/S[1]$, output the corresponding block of S as ciphertext, and run a reduced-round permutation between full blocks. In decryption mode, XOR ciphertext with $S[0]/S[1]$ to recover plaintext, update $S[0]/S[1]$ with the ciphertext, and again invoke the reduced-round permutation between blocks.
- **End message block:** In encryption mode, call the finalization module to derive the 128-bit tag and send it over SPI. In decryption mode, receive the tag from the device, run the same finalization procedure, compare tags in constant time, and send a single-bit authentication result. Then return to **IDLE**.

Any block type that is illegal in the current state (for example, receiving ciphertext/plaintext while **IDLE**, or receiving a new AEAD message while in **CP**) causes the controller to raise an error and issue an error response on the F→D channel.

In theory, the FSM for the AEAD core is as in Figure 4; in practice, however, the RTL implementation refines these three conceptual states into a richer seven-state FSM that accounts for nonce generation, initialization, finalization, and error handling, and it decomposes each phase into a sequence of smaller *tasks* that share a single permutation core.

Concretely, the controller introduces separate **WAIT NONCE** and **INIT** states between **IDLE** and **AD**. In encryption mode, a **NEW_ENCRYPT** command moves the FSM from **IDLE** into **WAIT NONCE**, where it stalls until the on-chip nonce generator returns a fresh 128-bit value. Only then does the controller assert an **INIT** task, which runs the 12-round Ascon initialization permutation on $\{K, N\}$ and, on completion, transitions the FSM into **AD**. In decryption mode, the host supplies the nonce directly, so the FSM jumps from **IDLE** to **INIT** without visiting **WAIT NONCE**.

Similarly, the end of the message is split off into an explicit **FINAL** state. When the host issues **END_MSG**, the controller stops accepting further payload, injects the key for a second time, and runs the final 12-round permutation as a dedicated finalization task. In encryption mode this task produces the 128-bit tag; in decryption mode it recomputes the expected tag and outputs a one-bit success flag. Only after this finalization task completes does the FSM return to **IDLE**, with all message-local registers cleared.

Finally, illegal command sequences (for example, receiving payload while still in **IDLE**, or a second **NEW_ENCRYPT** while a message is active) are routed into a dedicated **ERROR** state. Entering **ERROR** forces the controller to discard the current internal state, clear all pending tasks, and emit a fixed 128-bit error pattern on the response channel; the key register is left untouched so that software can recover by reissuing a valid **NEW_ENCRYPT** or **NEW_DECRYPT** command.

B. Nonce Generator (*nonce_gen*)

The nonce generator provides 128-bit nonces on demand to the AEAD controller during encryption. To avoid complex key-dependent state, the nonce generator is implemented as a

128-bit LFSR configured with a primitive polynomial, cycling through $2^{128} - 1$ possible values. When a new encryption message is requested, the next LFSR state is taken as the nonce and returned both to the device (for inclusion with the ciphertext) and to the initialization module.

A reset or power cycle poses a potential risk of reusing nonces if the LFSR state is reinitialized to a fixed seed. Possible mitigations include:

- Sampling on-chip analog noise sources on startup (if available) to randomize the LFSR seed.
- Adding a message type that allows the device to explicitly set the starting nonce after reset.
- Relying on the fact that ASCON retains a degree of robustness in certain nonce-misuse scenarios, although this is not a substitute for correct protocol design.

C. ASCON Permutation Core (*perm*)

The permutation core implements the ASCON- p permutation over the 320-bit state $S = (S[0], \dots, S[4])$, where each $S[i]$ is 64 bits. The core accepts the current state and a requested number of rounds (12 for initialization/finalization, fewer for intermediate calls), then applies one round per clock cycle.

Each permutation round consists of:

- 1) **Add round constant:** XOR an 8-bit round-dependent constant into $S[2]$.
- 2) **Nonlinear substitution layer:** Apply the 5-bit S-box independently to each bit position across the five 64-bit words, as depicted in Fig. 5.
- 3) **Linear diffusion layer:** XOR each $S[i]$ with rotated versions of itself to spread differences across bit positions. For example, one of the words is updated as $S[0] \triangleq \text{ROTR}(S[0], 19) \oplus \text{ROTR}(S[0], 28)$.

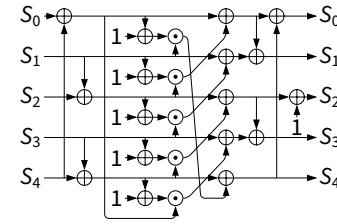


Fig. 5. ASCON S-box structure at a single bit position. Here $\odot$ denotes logical AND.

The core maintains an internal 320-bit register that latches the input state and iterates the specified number of rounds. When the final round completes, the core asserts *perm_done* and exposes the updated $S[0..4]$ to the caller. This one-round-per-cycle organization avoids deep combinational paths and simplifies timing closure, while still allowing bitwise operations within a round to be fully parallelized.

D. Initialization Module (*ascon_init*)

The initialization module constructs the initial state S from the ASCON parameters, key, and nonce. Conceptually, the state is initialized as $S \leftarrow (IV \parallel K \parallel N)$, where IV encodes the algorithm parameters (version, rate, round counts, and tag

We are not entirely clear why this is or why the communication speed is extremely sporadic—we however speculate that, examining the inputs of a logic analyzer, the slower clock speed allowed for more potential for an

accidental bit flip which crashes our designed system.

C. Power Usage

There were some power optimizations used in this design, in particular clock gating the `Ascon` module in `aead_control`, for a Total On-Chip Power of 0.130 W, with 0.058 W dynamic and 0.072 W static. However, we have not benchmarked the power to throughput efficiency of this accelerator to others that exist in the literature.

VI. CONCLUSION

We have presented the architecture of an FPGA-based accelerator for the ASCON-128 AEAD scheme. The design separates concerns between a structured SPI communication front-end and a reusable AEAD core centered around a round-based ASCON permutation unit. The controller FSM, nonce generator, initialization, associated-data, payload, and finalization modules collectively implement the full AEAD flow for both encryption and decryption while sharing a compact permutation core.

The primary bottleneck in the current design is the SPI communication stability rather than the permutation throughput. Future work includes refining the communication protocol for higher effective throughput, adding more robust nonce seeding for reset and power-loss resilience, implementing and measuring both BRAM and LUTRAM versions of the state storage, and exploring further optimizations for power and energy efficiency after synthesis and place-and-route on target devices.

REFERENCES

- [1] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schl  ffer, "Ascon v1.2: Lightweight authenticated encryption and hashing," IACR Cryptology ePrint Archive, Paper 2016/099, 2016. <https://eprint.iacr.org/2016/099>
- [2] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schl  ffer, "Ascon v1.2: Lightweight authenticated encryption and hashing," *Journal of Cryptology*, vol. 34, no. 3, art. 33, 2021.
- [3] M. S. Turan, K. A. McKay, D. Chang, J. Kang, and J. Kelsey, "Ascon-based lightweight cryptography standards for constrained devices," National Institute of Standards and Technology (NIST), Gaithersburg, MD, NIST Special Publication (SP) 800-232 (Initial Public Draft), Nov. 2024.