Hardware Implementation of SpoC-128
Compliant with George Mason University's LWC API

Ambati Sathvik[1], Tirunagari Rahul[1], Anubhab Baksi[2], and Vikramkumar Pudi[1]

[1] Indian Institute of Technology Tirupati, Andhra Pradesh, India
[2] Nanyang Technological University, Singapore

ee17b002@iittp.ac.in, ee17b030@iittp.ac.in, anubhab001@e.ntu.edu.sg, vikram@iittp.ac.in

Abstract. In this work, we present a hardware implementation of the lightweight Authenticated Encryption with Associated Data (AEAD) SpoC-128. Designed by AlTawy, Gong, He, Jha, Mandal, Nandi and Rohit; SpoC-128 was submitted to the Lightweight Cryptography (LWC) competition being organised by the National Institute of Standards and Technology (NIST) of the United States Department of Commerce. Our implementation follows the Application Programming Interface (API) specified by the cryptographic engineering research group in the George Mason University (GMU). The source codes are available over the public internet as an open-source project.

Keywords: lightweight cryptography · spoc-128 · hardware implementation · api compliance

1 Introduction

Lightweight cryptography is among the recent trends in the community. The objective of this is to design new primitives with low device footprint or find optimised device implementation of an existing primitive. To boost up research-and-development in this area, currently a competition named the "Lightweight Cryptography"[1] (LWC) is being organised by the National Institute of Standards and Technology (NIST). This competition is for the so-called Authenticated Encryption with Associated Data (AEAD) [3] schemes; which offer authenticity and privacy for a given plaintext, and authenticity for a given associated data.

To state more formally, given a message plaintext and an associated data; such a scheme encrypts the plaintext and generates a tag that takes both the plaintext and the associated data as inputs. The sender (Alice), after generating the ciphertext and the tag, sends over along with the associated data through an insecure channel (where the attacker, Eve, is active) to the recipient (Bob). Upon receiving, Bob recreates the plaintext and the tag. If both the tags match, then the received ciphertext and the associated data are considered not perturbed by Eve. Otherwise, Bob decides the received data are disturbed; in which case, he generates an invalid signal and follows the recovery procedure (typically discards the received data).

Given the fundamental aspect of the lightweight cryptography is to reduce the cost of device implementation, there is a body of research dedicated to find optimised implementation. Overall, this can be classified into software and hardware categories. Our target here is limited to hardware implementation only. Dedicated research works attempting to find optimised hardware implementation of lightweight ciphers have been reported earlier, for instance, one may refer to [2,5].

Contribution

While (at least) one hardware implementation (together with a benchmark) is almost an integral part of a lightweight cipher design, we observe some of the ciphers in LWC are missing any viable implementation. Here we pick the cipher SpoC-128, which is a primary recommendation (along with SpoC-64) in the SpoC family[2] [1]. There is a third-party open-source implementation of SpoC-64[3], which serves as the stepping stone in our implementation.

All the source codes, along with Vivado project files and stuffs, can be accessed in the public repository[4] as an open-source project. At the same time, the researchers from the cryptographic engineering Research group in the George Mason University (GMU); under the "ATHENa: Automated Tools for Hardware EvaluatioN"[5] project; have specified a standardised Application Programming Interface (API) in order to make hardware benchmark fair, comprehensive and homogeneous. Our implementation is compliant with this API. This is the first (and so far only) hardware implementation of SpoC-128, to the best of our knowledge.

[1] https://csrc.nist.gov/Projects/Lightweight-Cryptography.
[2] The cipher is eventually not selected in the final round of the LWC competition.
[3] https://github.com/vtsal/spoc_lwc.
[4] https://github.com/T-Rahul/SpoC_128 .
[5] Website: https://cryptography.gmu.edu/athena/index.php.

2 Description

SpoC-64 and SpoC-128 are the two member of the SpoC family [1], 64 and 128 are the rates at which information is fed and processed, both of them require 128 bit key and nonce. SpoC-128 is has a 256-bit state and generates 128-bit tag.

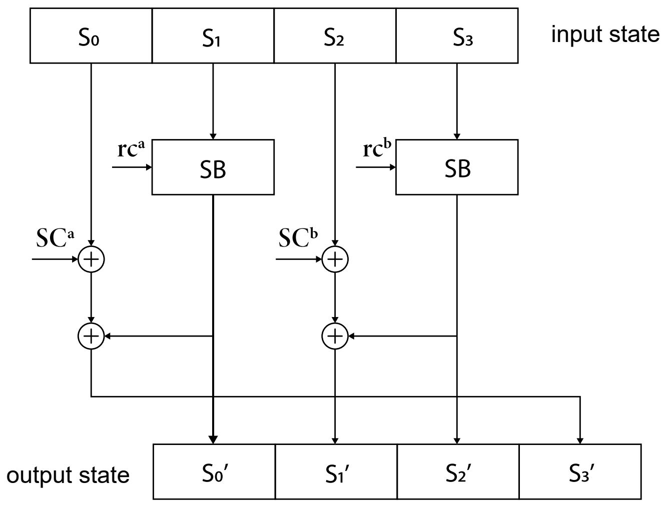

Fig. 1: sLiSCP-light (step) permutation

sLiSCP-light (step) permutation diffuses the information across the state as shown in Figure 1. The Simeck box (SB) is illustrated in Figure 2. The constants rc^a, rc^b, sc^a, sc^b are provided in [1], hence omitted here for the interest of brevity. The state is represented as $S_0||S_1||S_2||S_3$; where each S_i, $0 \leqslant i \leqslant 3$, is a 64-bit sub-state.

The state is initialised from 128-bit key (split as $K_0||K_1$) and nonce $N_0||N_1$, as $N_0||K_0||N_1||K_1$. As only 128-bit data can only be introduced into the state, associated data and plaintext are split into 128-bit blocks and the final block is padded (if necessary).

Control bits $(c_3||c_2||c_1||c_0)$ differentiate the complete and partial blocks, AD, plaintext and tag operations. The most significant bit c_3 is 1 during tag generation only. The bits c_2 and c_1 correspond to plaintext (also referred as message block) and associated data and are 1 during their processes respectively.

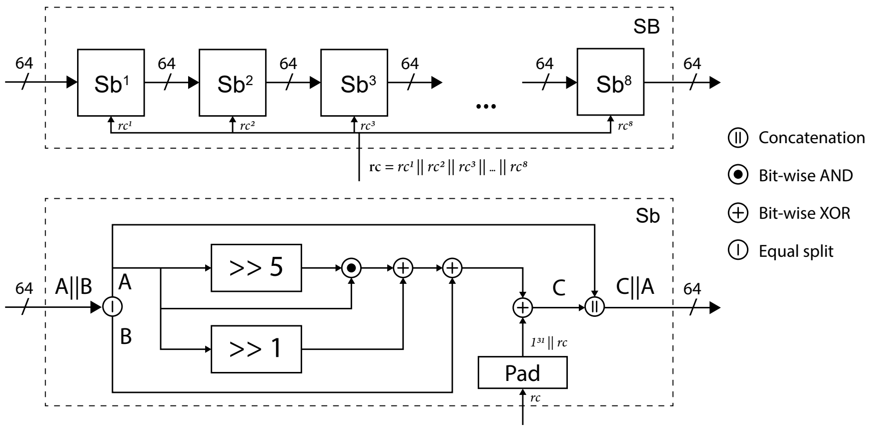

Fig. 2: Simeck box (SB) and Sub-Simeck box (Sb)

Processing associated data of length *adlen* starts with splitting AD into blocks of 128-bits (starting from MSB). If required, the last block is padded with $1||0^p$, $p = adlen \pmod{128} - 1$ and the further process is described in Figure 3.

Plaintext during encryption and ciphertext during decryption of length *pclen*, is padded (if required) and split into 128-bit blocks. During encryption, the input message block M^i is further split into 64-bit

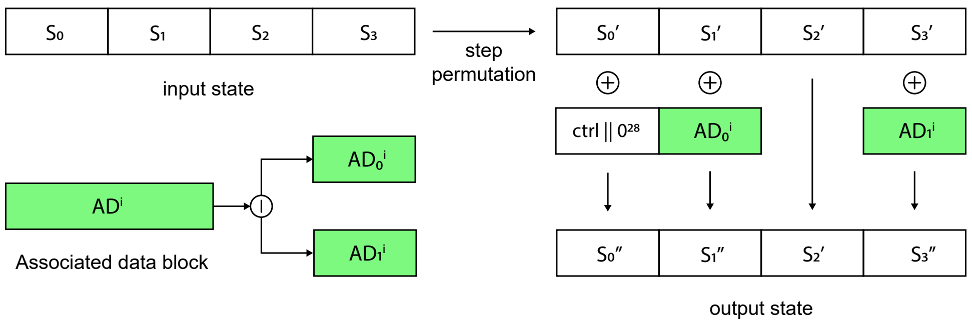

Fig. 3: Processing associated data

halves M_0^i, M_1^i and XORed with appropriate sub-states S_1' and S_2'. During the decryption process, the input ciphertext block CT^i is XORed with sub-states $S_0^i||S_2^i$ and the obtained plaintext block M^i is split and the successive steps are same as encryption.

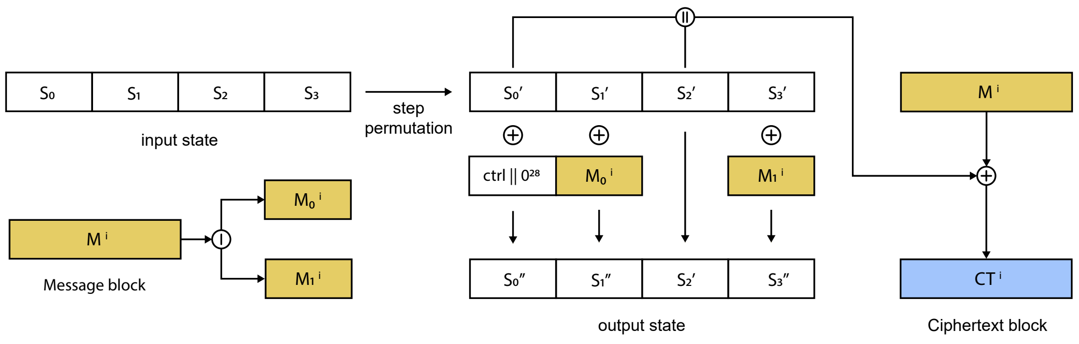

Fig. 4: Processing plaintext during encryption

The output state of plaintext process is XORed with $ctrl||0^{252}$ and step permutation is performed. The sub-states $S_1||S_3$ of the resulting state make up the 128-bit tag.

3 Hardware Configuration (LWC API by GMU)

As mentioned, the LWC API is a standardised interface specified by researchers (mostly) from GMU; which is used to implement authenticated ciphers efficiently with several constraints like permitted widths of input, output ports, single clock throughout the core[6].

- PreProcessor is the unit responsible for passing input blocks to the CryptoCore and records the remaining blocks, thereby specifying the type of input data (AD, PT/CT) and their nature (partial or not) through `bdi_type` signal.
- CryptoCore depends on the cipher, so, this unit performs padding, processes blocks of input data, provides CT and tag during encryption, PT and verification signal during decryption.
  - Controller mainly commands the stages of algorithm (reset, key stage, npub stage, initialisation, processing AD blocks, processing PT/CT blocks, pre-tag stage, finish tag) by instructing enable signals accordingly.
  - Datapath is solely dependent on cipher, this module directly collects `pdi` and `sdi` data from PreProcessor and yields the output ciphertext, tag and passes to PostProcessor.
- PostProcessor takes care of invalid bytes of output words, and functions with respect to message authentication signal.

The input data are categorised as secret data input (`sdi`) and public data input (`pdi`). The secret key enters through `sdi`, and nonce, AD, PT/CT, and tag are passed through `pdi` and block data input (`bdi`). General signals that make up the LWC core are explained below:

[6]API homepage: https://cryptography.gmu.edu/athena/index.php?id=LWC.

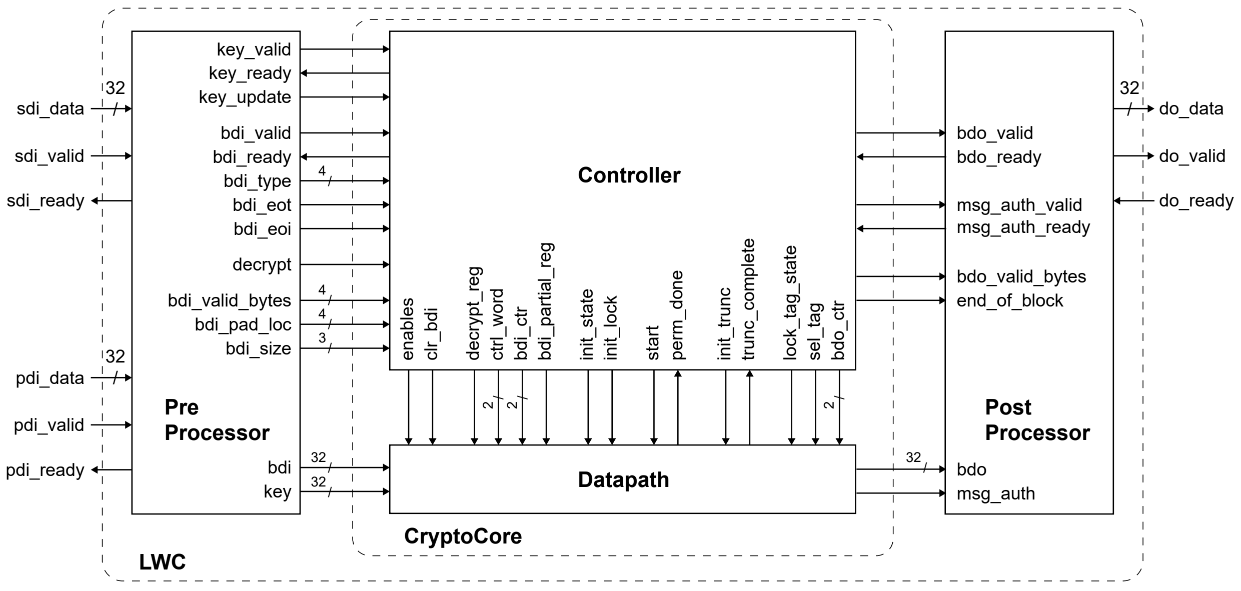

Fig. 5: Top level block diagram of LWC core

- `bdi`, `sdi` signals are of width 32 bits (4 bytes) carrying input data as mentioned above.
- `bdi_valid`, `sdi_valid` specify the instants when input is fed. `bdi_valid_bytes` of size, `bdi_size`/8 specifies the valid bytes of present input block. The signal is 1110, if 24 bits (first 3 bytes) of the current block (maximum of 4 bytes) are valid.
- `bdi_size` conveys the size of valid bytes in the current 32-bit block.
- `bdi_pad_loc` commands the padding by providing the starting location to pad. The size of this signal is same as that of `bdi_vali_bytes`. The signal is 0001 for the above example, so that the last byte is padded according to the cipher. For complete block inputs, `bdi_valid_bytes` is 1111 and `bdi_pad_loc` is 0000.
- `bdi_ready`, `sdi_ready` indicate that the module is prepared to take inputs.
- `bdi_type`, a 4-bit encoded representation identifies `bdi` data as nonce, AD, PT/CT, and tag.
- `bdi_eot` intimates the end of type that is being sent. This signal is reset by default, is set high to specify the last block of current input type. This signal is 1, only during the last blocks of nonce, AD and PT.
- `bdi_eoi` is set high to specify the last block of overall inputs. For example, if there exist no associated data and plaintext during encryption process, `bdi_eoi` is high at the last block of nonce itself, so that, the tag generation process is started.

Signals that are specific to SpoC-128 are explained below:

- `enables` is set of enable key, enable npub, enable bdi, enable state, and enable cumulative size signals which activates the corresponding activities.
- `decrypt`, `decrypt_reg` is low during the process of encryption and are high during decryption.
- As the key, npub sizes are of known sizes and are mandatory, these inputs can be automatically identified. So, `ctrl_word` (2 bits) indicate AD with 01 and PT/CT with 10, this representation simplifies the 4-bit control code used later, within the SpoC-128.
- `bdi_ctr` holds the number of 32-bit turns taken to process one 128-bit input block of AD/PT/CT. Similarly, `cum_size` holds the number of bytes, so, it is of log(128/8) bits.
- Initialisation stage is signalled by setting `init_state` and `init_lock` to 1.
- The step permutation of SpoC-128, consisting 18 rounds, is commenced by exciting `start` signal once. Datapath module, after the completion of 18 rounds, sets `perm_done` to high.
- The controller indicates truncation of partial blocks through `init_trunc`. Datapath replies the accomplishment of the task through `trunc_complete` signal.
- The tag process state is specified by setting `lock_tag_state` to 1, and `sel_tag` = 1 extracts tag through `bdo` port.

4 Result and Discussion

Our study resulted in a tangible output as we have developed a hardware code capable of implementing the discussed AE algorithms. The 1,088 test vectors (comprising several combinations of associated data and plaintext, for fixed 128-bit key and nonce) of SpoC-128 provided with the official C code[7] were passed successfully by our hardware description code. The properties of ZYNQ-7 ZC702 evaluation board for the proposed HDL code are indicated in Tables 1, 2 and 3.

Table 1: Cryptocore components (ZYNQ-7 FPGA)

Module	Component	Size	Quantity
Controller	Adders	2-input, 5-bit	1
		2-input, 2-bit	3
	Registers	1-bit	11
	MUXes	2-input, 1-bit	1020
Datapath	Adders	2-input, 5-bit	3
	XORs	2-input, 1-bit	261
	Registers	1-bit	391
	MUXes	2-input, 1-bit	3860
	LUTs	32×8	4

Table 2: Resource utilisation of SpoC-128 (ZYNQ-7 FPGA)

Component type	Available	Used	Utilised (%)
Slice LUTs	53200	2711	5.09
Flip-flops	106400	919	0.86
Bonded IOBs	200	105	52.50

Table 3: Primitives utilisation of SpoC-128 (ZYNQ-7 FPGA)

Ref Name	Used	Functional Category
LUT6	1412	LUT
FDRE	899	Flop & Latch
LUT5	669	LUT
LUT2	432	LUT
LUT3	349	LUT
LUT4	150	LUT
IBUF	69	IO
OBUF	36	IO
RAMD32	30	Distributed Memory
CARRY4	24	CarryLogic
FDSE	20	Flop & Latch
RAMS32	10	Distributed Memory
LUT1	8	LUT
BUFG	1	Clock

Various flowcharts for Controller can be found in Figures 6, 7, 8 and 9. The Controller starts in `RESET_ST` (reset state). The CryptoCore module allows only 32 bits for input and output. So, the key through `sdi` and then nonce through `bdi` are obtained in 4 cycles each, and the state gets loaded. Now, the Controller reaches `INIT_ST` as shown in flowchart of Figure 6.

Controller then proceeds to `PROC_ST` and rests in `STORE_PROC_ST` till the sLiSCP-light (step) permutation is performed as shown in Figure 7.

Each AD block is loaded in 4 cycles and then undergoes step permutation, and Controller finally reaches to `FINISH_PROC_ST`. After AD processing is finished, as `bdi_type_reg` is not 1 (NOT AD), it goes to `WRITE_PTCT_ST` to process plaintext or ciphertext. Figure 8 shows that the Controller directly reaches `PRE_TAG_ST` in case there is no PT/CT.If PT/CT is available, the complete blocks are processed and then

[7]https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/submissions-rnd2/spoc.zip

outputs the CT/PT block through bdo in 4 cycles. However, if the last PT/CT block is partial, after zero-padding and the step permutation, the produced CT/PT block is truncated with the help of cum_size signal and the obtained output is passed through bdo in appropriate number of cycles (for example, 72-bit block requires 3 cycles). The Controller then reaches PRE_TAG_ST.

In encryption mode, the Controller goes to FINISH_TAG_ST, then the tag gets generated and is passed through bdo in 4 cycles. In decryption mode, the tag is generated before the Controller reaches FINISH_TAG_ST and then the tag is verified. After this process, Controller again gets back to RESET_ST as in Figure 9.

5 Conclusion

We present a hardware implementation of the lightweight AEAD SpoC-128 which is compliant with the API designed by the researchers from GMU. Our work is the first implementation of the API complaint hardware (and it is not generated by high level synthesis) SpoC-128. It is featured in [4] unded the name "*SpoC_IIT*" with the design group name as "*VLSI Group, IIT Tirupati*". Our work will hopefully further help the community in making the hardware implementation/fair benchmark for lightweight ciphers in the coming future.

References

1. AlTawy, R., Gong, G., He, M., Jha, A., Mandal, K., Nandi, M., Rohit, R.: Spoc: An authenticated cipher. Submission to NIST LWC Standardization Process (Round 2) (2019), https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/spoc-spec-round2.pdf 1, 2
2. Baksi, A., Pudi, V., Mandal, S., Chattopadhyay, A.: Lightweight ASIC implementation of AEGIS-128. In: 2018 IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2018, Hong Kong, China, July 8-11, 2018. pp. 251–256. IEEE Computer Society (2018), https://doi.org/10.1109/ISVLSI.2018.00054 1
3. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. IACR Cryptology ePrint Archive 2000, 25 (2000), http://eprint.iacr.org/2000/025 1
4. Mohajerani, K., Haeussler, R., Nagpal, R., Farahmand, F., Abdulgadir, A., Kaps, J.P., Gaj, K.: FPGA benchmarking of round 2 candidates in the nist lightweight cryptography standardization process: Methodology, metrics, tools, and results. Tech. rep. (2020), https://eprint.iacr.org/2020/1207 6
5. Sönnerup, J., Hell, M., Sönnerup, M., Khattar, R.: Efficient hardware implementations of grain-128aead. In: Hao, F., Ruj, S., Gupta, S.S. (eds.) Progress in Cryptology - INDOCRYPT 2019 - 20th International Conference on Cryptology in India, Hyderabad, India, December 15-18, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11898, pp. 495–513. Springer (2019), https://doi.org/10.1007/978-3-030-35423-7_25 1

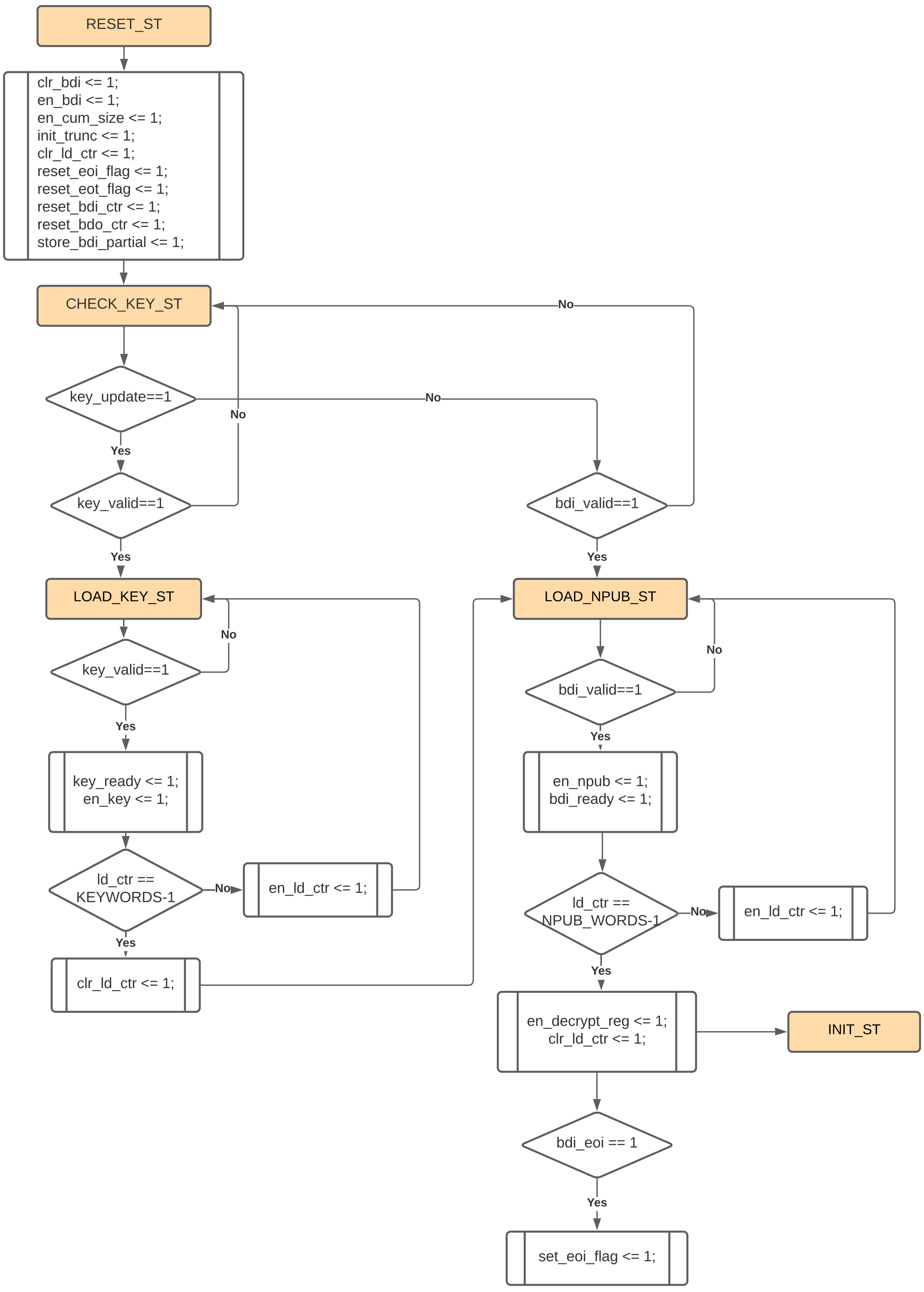

Fig. 6: Controller reaching initialisation from reset

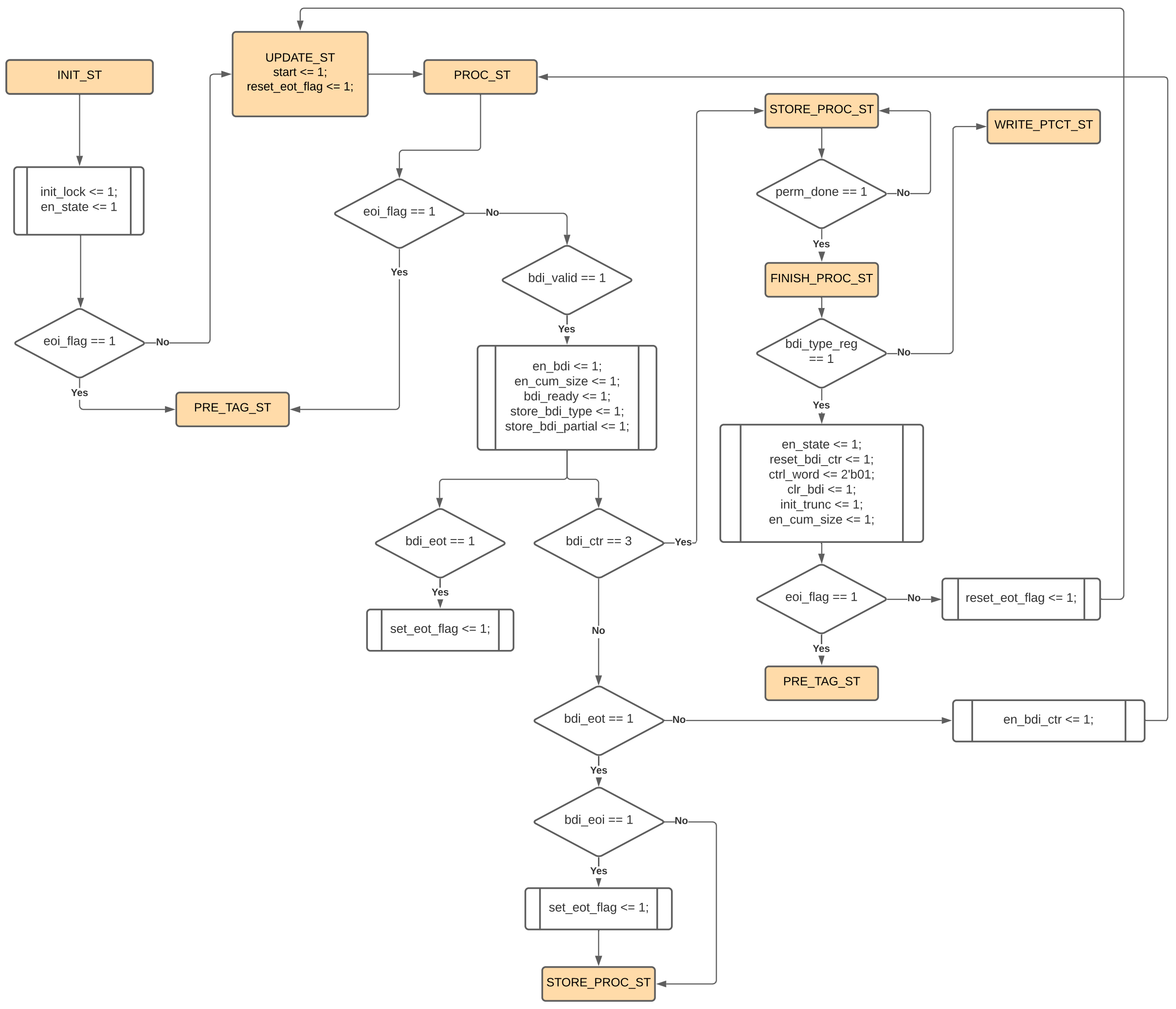

Fig. 7: Initialised state processing with AD

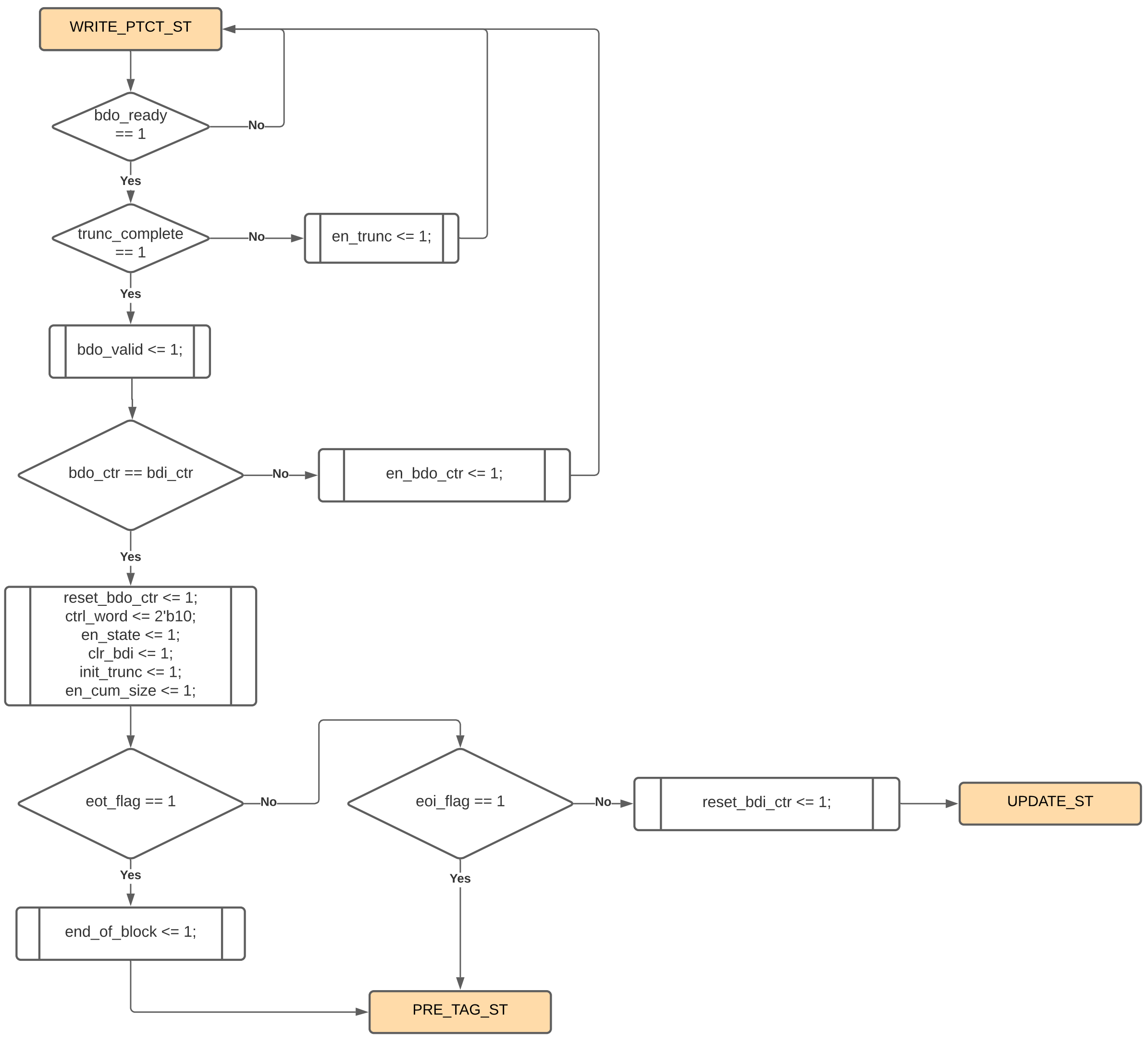

Fig. 8: Updated state processing with PT/CT

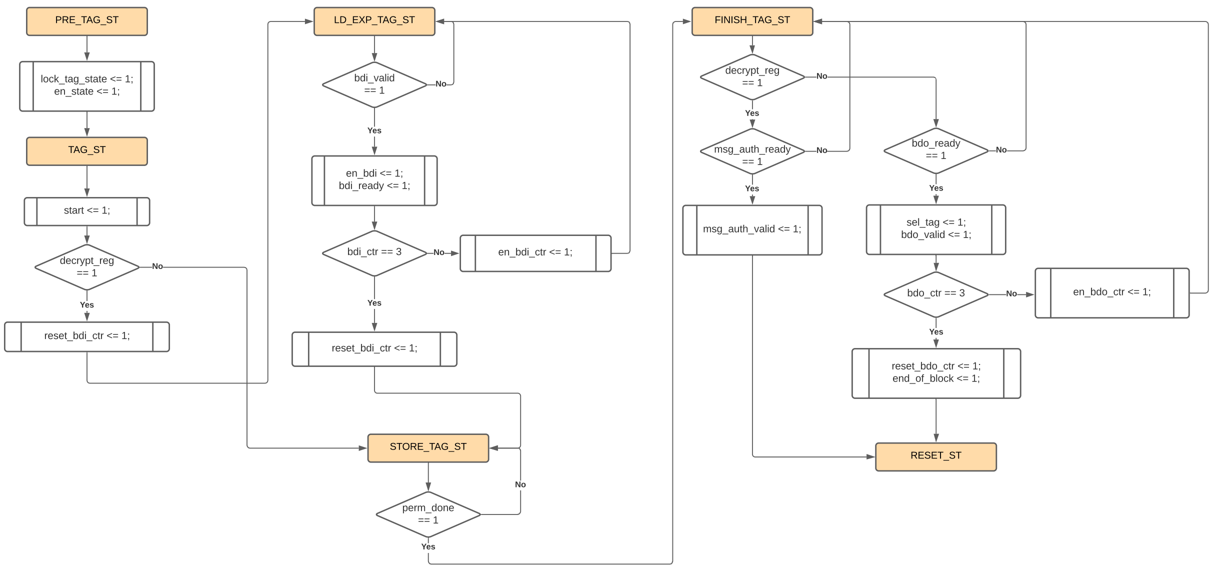

Fig. 9: Tag generation and Controller reaching reset state