SMAesH: technical documentation

Masked Hardware AES-128 Encryption with HPC2

SIMPLE-Crypto

Contents

1 Overview

This document describes SIMPLE-Crypto's Masked AES in Hardware (SMAesH), implemented in the `aes_enc128_32bits_hpc2` hardware IP.

2 History

1.0.0 (2023-05-01) Initial release.

3 Features

The AES-128 HPC2 module is a masked hardware implementation of the AES-128 encryption algorithm as specified in [NIS01].

- The core implements the AES-128 encrypt function.
- The implementation is protected against side-channel attacks using the HPC2 masking scheme [CGLS21].
- The amount of shares $d \geq 2$ can be chosen at synthesis time.
- The randomness required for the masking scheme is internally generated using an embedded PRNG.
- The core is controlled through three simple valid-ready stream interfaces (input data/key, output data and PRNG seed.
- The core has an encryption latency of 105 clock cycles and a throughput of one 128-bit block of data per 105 clock cycles.
- There is no latency penalty for key change.
- The state of the core is automatically cleared when encryption finishes.

4 Core User Guide

A top-level view of the core is shown in Figure 7 and a detailed list of the ports is given in Table 1. The interface is composed of three independent interfaces: the input composed of the plaintext and the key (in red), the ciphertext output (in blue) and the PRNG seed (in green). The key (`in_shares_key`), plaintext (`in_shares_plaintext`) and ciphertext (`out_shares_ciphertext`) are all 128-bit masked values. The internal PRNG seed (`in_seed`) is 80-bit wide.

In this section we next detail the operation of the Synchronous Valid-Ready Stream (SVRS) protocol for the data interfaces, the operation of the `aes_enc128_32bits_hpc2` core, and the masked data encoding.

4.1 SVRS protocol

The Synchronous Valid-Ready Stream (SVRS) protocol operates between a sender and a receiver. The bus is composed of the two control signals `valid` and `ready`, as well as any number of `data` wires. The `valid` and `data` signals are outputs (resp. inputs) of the

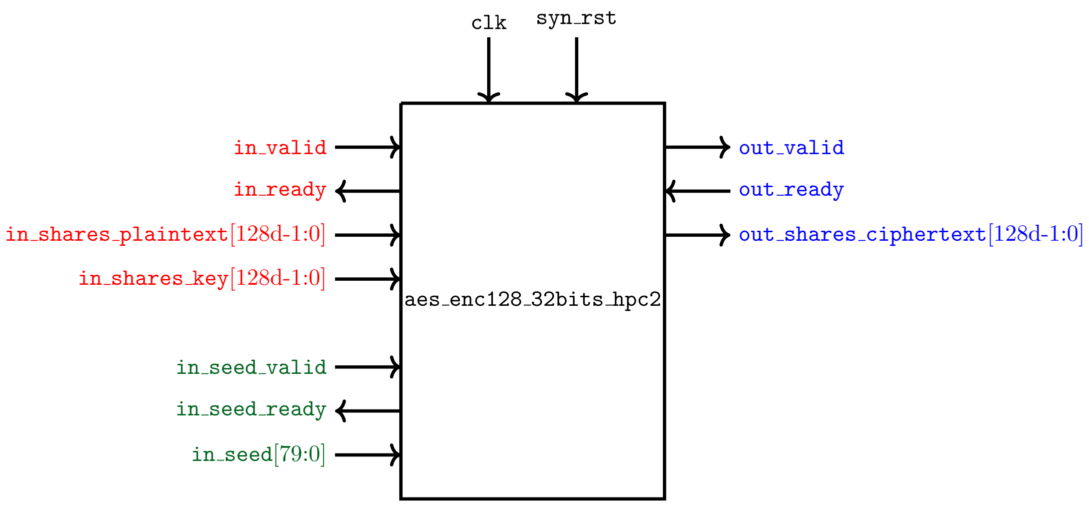

Figure 1: Top level view of module `aes_enc128_32bits_hpc2`.

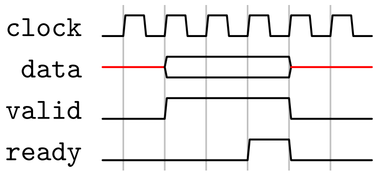

Figure 2: SVRS transaction (don't care (X) signals are represented with a flat red solid line).

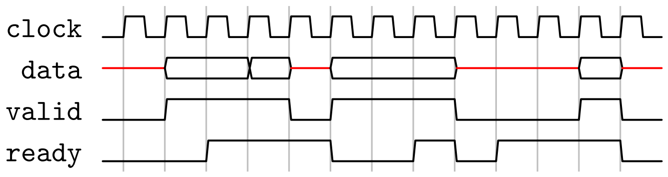

Figure 3: Multiple SVRS transactions.

Module Generics				
Parameter	Value Type			Description
d	integer			Amount of shares
`PRNG_MAX_UNROLL`	integer			Maximum unrolling for the embedded PRNG.
Module Ports				
Ports Name	Type	Direction	Width [bits]	Description
`clk`	clock	input	1	Clock (all the logic is synchronized on the positive edge).
`syn_rst`	control	input	1	Active high synchronous reset. Keep asserted for at least one cycle.
SVRS Input interface				
`in_shares_plaintext`	data	input	$128d$	Shared plaintext (SVRS `data` signal).
`in_shares_key`	data	input	$128d$	Shared key (SVRS `data` signal).
`in_valid`	control	input	1	SVRS `valid` signal.
`in_ready`	control	output	1	SVRS `ready` signal.
SVRS Seed interface				
`in_seed`	data	input	80	Fresh randomness used as a seed by the embedded PRNG (SVRS `data` signal).
`in_seed_valid`	control	input	1	SVRS `valid` signal.
`in_seed_ready`	control	output	1	SVRS `ready` signal.
SVRS Output interface				
`out_shares_ciphertext`	data	output	$128d$	Shared ciphertext (SVRS `data` signal).
`out_valid`	control	output	1	SVRS `valid` signal.
`out_ready`	control	input	1	SVRS `valid` signal.

Table 1: `aes_enc128_32bits_hpc2` port description.

sender (resp. receiver), while the `ready` signal is an input (resp. output) of the sender (resp. receiver).

The bus operates synchronously with an event source shared by the sender and the receiver (here, the positive edges of the clock). At each event, a transaction occurs if both `valid` and `ready` are asserted (i.e. set to logical 1). The transmitted data of the transaction is the value of the `data` signals at the event.

Once `valid` is asserted, it cannot be de-asserted (i.e., sticky signal), nor can the value of `data` be changed until a transaction occurs. To prevent deadlocks, a sender must not wait until the assertion of `ready` before asserting `valid`. To prevent combinational logic loops, the `valid` signal may not combinationally depend on the `ready` signal.

Examples of protocol use are given in Figures 2 and 3.

4.2 Core Usage

Encryption An encryption is started by executing a transaction on the `in` interface. The encryption is executed using the shared key and plaintext provided in the transaction, then the `out` interface becomes valid, with the shared ciphertext as data.

The core can only perform one execution at a time and will not start a new encryption before the ciphertext of the current encryption has been consumed from the `out` interface. Figure 4 illustrates the interface signal for two consecutive encryptions.

Security: The `out_shares_ciphertext` is gated to not expose any confidential value when `out_valid` is not asserted.

Initialization: After reset, the core will not start an encryption before it is reseeded.

Latency and throughput: The AES implementation has a latency of 105 clock cycles. To achieve the maximum throughput of one encrypted block per 105 cycles, there must be no back-pressure (i.e., `out_ready` must be high at the clock cycle where `cipher_valid` becomes asserted) and the input must be valid (`valid_in` asserted) at least one cycle before `cipher_valid` is asserted.

(Re-)seeding The `seed` interface is used to reseed the internal PRNG (this PRNG generates the internal masking randomness, see Section 5.5 for details). A reseed is executed by means of a transaction on the `seed` interface, as shown in Figure 5. During this transaction, the provided seed data `must` be uniform randomness (i.e. all the bits must be fresh, uniform and independent). After a reseed transaction, the reseeding procedure lasts for a few cycles (the duration depends on the core configuration, it is typically less than a dozen cycles).

Interactions with encryption.

- After a reset, the core does not start any encryption before being reseeded once.
- The core will not accept a reseed transaction while it is encrypting.
- The core will not start an encryption while it is reseeding.

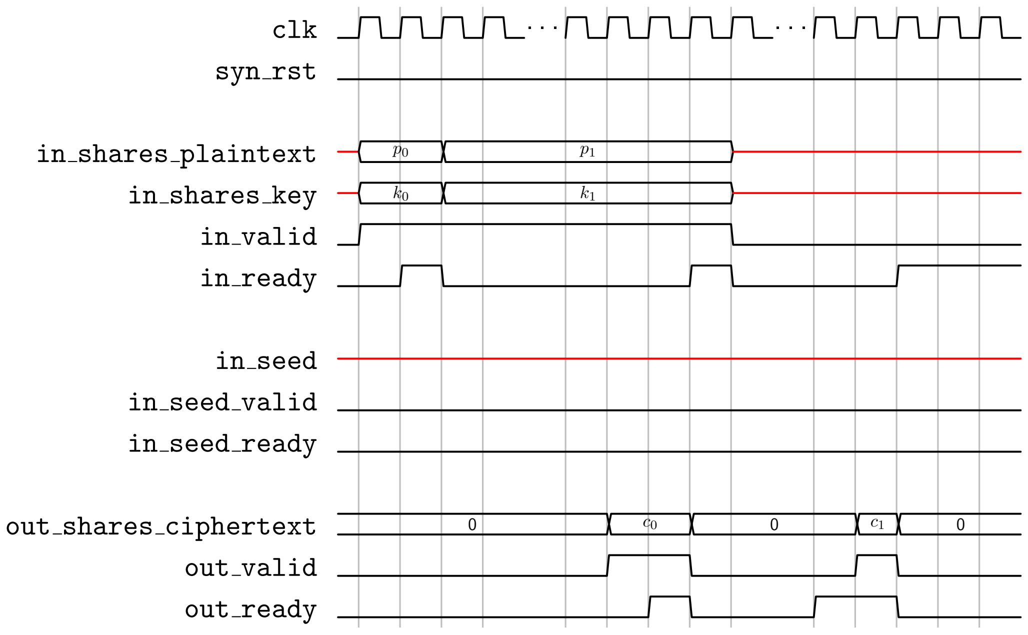

Figure 4: Exemplary interface view for two executions.

- Starting a new encryption takes precedence over starting a reseed, hence if reseeding if needed, no new valid input should be asserted before a reseed transaction happens.

4.3 Sharing encoding

The busses `in_shares_plaintext`, `in_shares_key` and `out_shares_ciphertext` contain respectively the shared representation of the plaintext, the key and the ciphertext.

A sharing (or shared representation) of a bit b is a tuple of d shares $\left(b^0, b^1, \dots, b^{d-1}\right)$ such that $\bigoplus_{m,0\leq m<d} b^m = b$. The sharing of a n-bit bus `data` $[n-1:0]$ where `data`$[i] = b_i$ is `shares_data` $[nd-1:0]$ where `shares_data` $[ni+j] = b_i^j$ and $\left(b_i^0, \dots, b_i^{d-1}\right)$ is a sharing of b_i. This representation is illustrated in Figure 6.

The key and the plaintext must be fed as uniform sharings (i.e. the sharing is selected uniformly at random among possible sharings that represent the correct value). The output ciphertext sharing is guaranteed to be uniform.

5 Core Architecture

The top-level architecture of `aes_enc128_32bits_hpc2` is depicted in Figure 7: its main components are the encryption unit `MSKaes_32bits_core` and the PRNG. Some additional logic is used to handle the encrypt/reseed interlocking, as well as units to shuffle the shares of the masked busses.

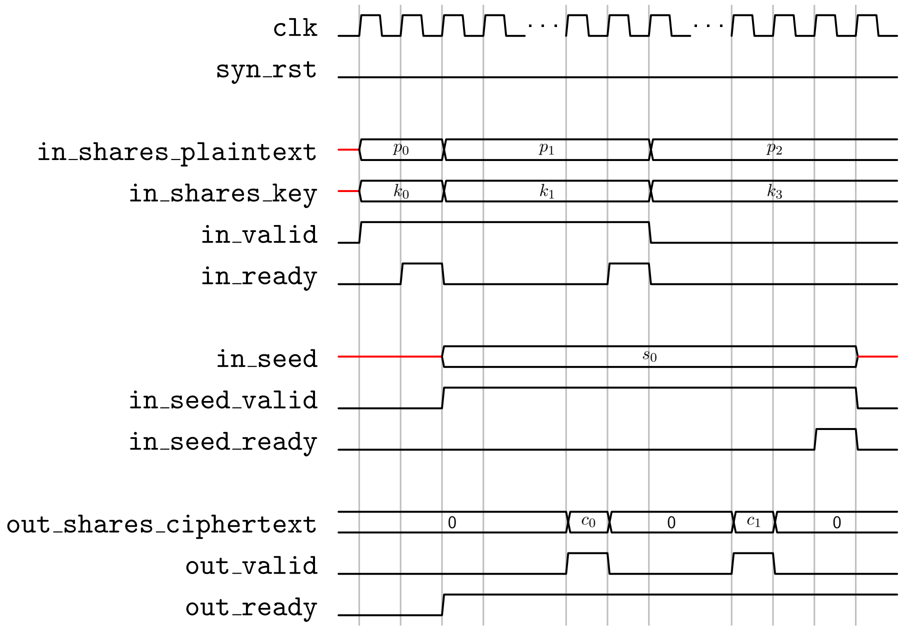

Figure 5: Exemplary reseeding procedure.

Core The module `MSKaes_32bits_core` implements a masked version of the AES encryption algorithm by serially processing 32-bits parts of the state. It runs a single AES execution at a time and the ciphertext produced (`sh_ciphertext`) has to be fetched before a new execution can start. The shared plaintext (`sh_plaintext`) and the shared key (`sh_key`) are fetched at the beginning of a new execution by performing a simple transaction at the input interface (with `valid_in` and `in_ready`). Similarly, the shared ciphertext (`sh_ciphertext`) is output from the core with a dedicated interface (with `cipher_valid` and `out_ready`). The signal `busy` is asserted when an execution is ongoing inside the core.

PRNG The module `prng_top` is generating the randomness required by the masking scheme. It is the producer on the randomness bus, while `MSKaes_32bits_core` is the receiver.

When not reseeding, it takes only a single cycle to generate the fresh randomness, therefore at the next cycle after a randomness transaction, new randomness is already available (i.e., `rnd` carries fresh randomness, and `out_valid` is asserted). During an encryption, `MSKaes_32bits_core` needs randomness at all clock cycles, hence it keeps `out_ready` asserted, and thanks to the high-throughput capability of the PRNG, a transaction happens on the randomness bus at every clock cycles (`out_valid` stays asserted).

This high throughput capability is actually relied upon by `MSKaes_32bits_core`: it needs randomness for security at every cycle during the encryption and cannot stall once encryption is started. The signal `out_valid` is de-asserted only when the PRNG has not been seeded after a reset, or while it is reseeding. To ensure that fresh randomness is al-

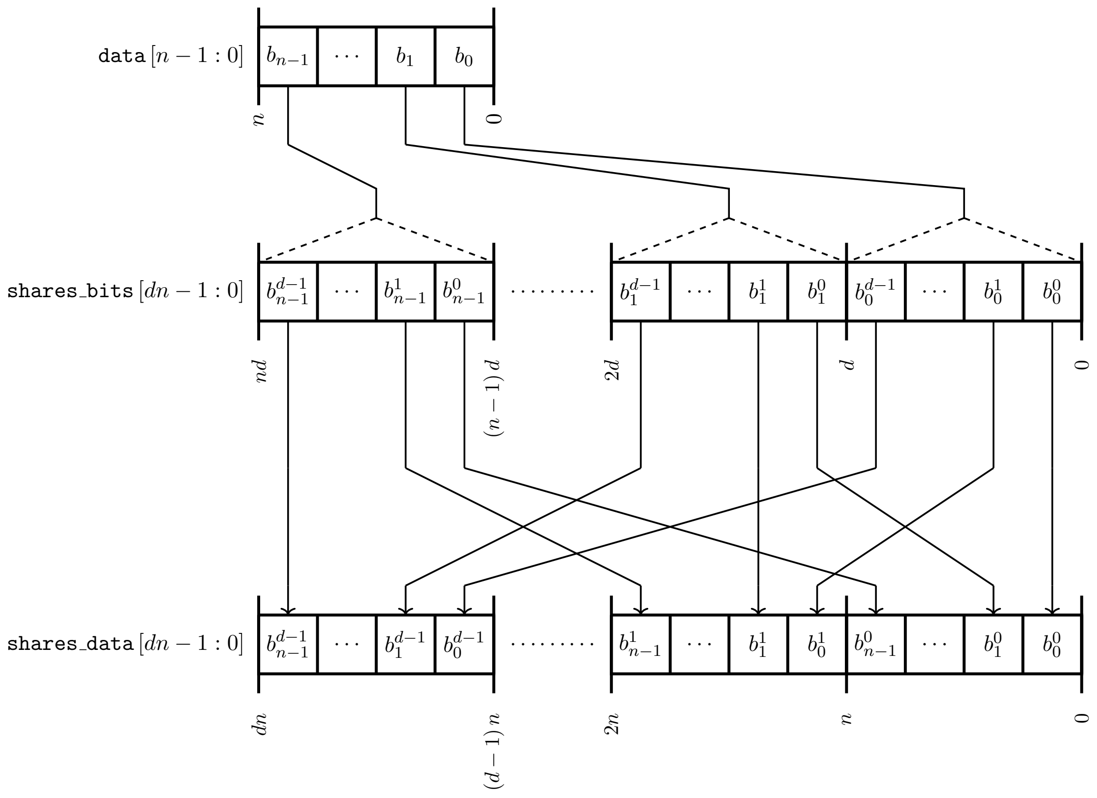

Figure 6: Encoding of a shared n-bit wide data with d shares.

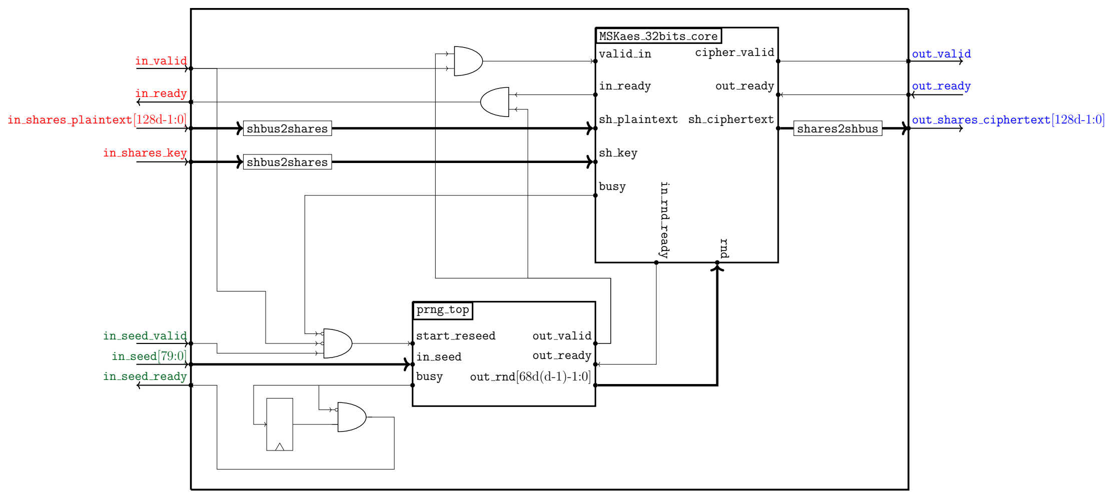

Figure 7: Global architecture of the module `aes_enc128_32bits_hpc2`.

ways available when encrypting, the interlocking logic prevents the `MSKaes_32bits_core` from starting an encryption if `out_valid` is de-asserted, while it prevents `prng_top` from starting a reseed when an encryption is ongoing. If no encryption is ongoing and `in_seed_valid` is asserted, then a reseed is initiated and a transaction on the `seed` bus occurs at the next cycle (this is to avoid a combinational dependency `in_seed_valid` $\rightarrow$ `in_seed_ready`, and is achieved by detecting a rising edge on the PRNG `busy` signal).

Share shuffling The modules `shares2shbus` and `shbus2shares` are simple wire shufflings that "transpose" the encoding of the shared data. More precisely, the encoding of a sharing inside `MSKaes_32bits_core` is `shares_data_inner` $[ni + j] = b_j^i$ unlike the more intuitive external representation `shares_data` $[ni + j] = b_i^j$ described in Section 4.3. This internal representation is more convenient for the implementation, as it makes it easier to describe the extraction of masked bits from a masked bus using Verilog operators.

5.1 Masked AES Core Architecture

The module `MSKaes_32bits_core` is almost identical to the 32-bit masked AES implementation presented in [MCS22]. As shown in Figure 8, the module is organized around two datapath blocks performing the operations dedicated to the round computation (denoted `MSKaes_32bits_state_datapath`) and the key scheduling (denoted `MSKaes_32bits_key_datapath`). The module `MSKSbox` is shared between the two datapath blocks and implements the `SubBytes` layer for 4 masked bytes. In particular, it is composed of 4 parallel instances of masked S-boxes that follow an optimized architecture based on the S-box representation presented in [BP12]. A single S-box is organized as a pipeline of 6 stages with 34 HPC2 AND gadgets each requiring $d(d-2)/2$ bits of

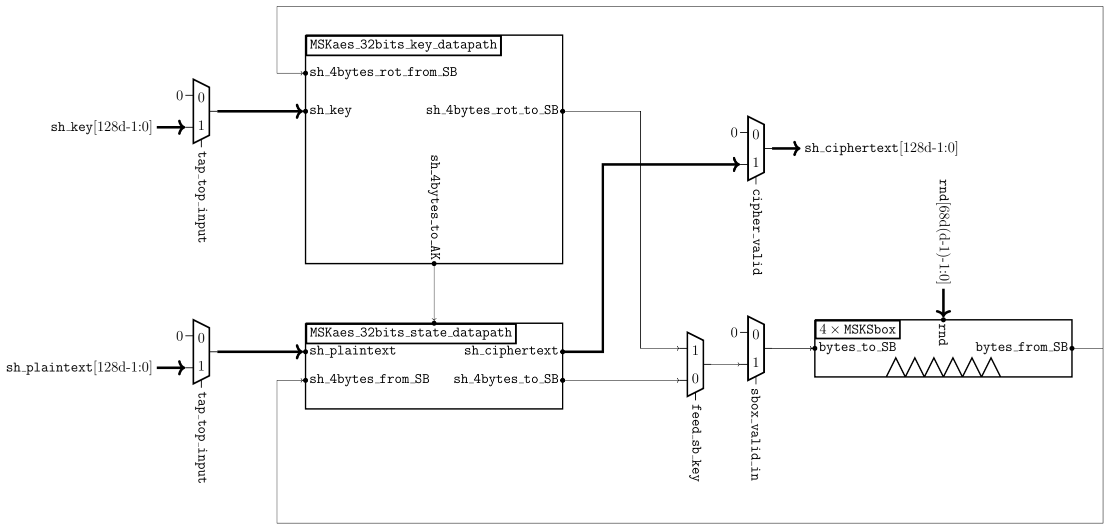

Figure 8: Datapath architecture of the module MSKaes_32bits_core. Wires not in bold are $32d$ bits wide (apart from muxes control signals).

randomness per execution. The bus rnd is used to provide the fresh randomness to the 4 S-boxes instances (randomness is not used anywhere else in MSKaes_32bits_core).

5.2 Architecture of the MSKaes_32bits_state_datapath module

Figure 9 shows the detailed architecture of the module MSKaes_32bits_state_datapath. It is organized as a shift register where each register unit holds a masked state byte (the numbers on the figure indicate the byte index in the unmasked state). The module operates on 32-bit parts of the state and is also implementing the logic that computes the AddRoundKey, ShiftRows and MixColumns layers. In particular, these are implemented in purely combinational logic. Addition gadgets (i.e., XORs) are used to perform the key addition with key bytes coming from the round key (denoted sh_4bytes_from_key). The module MC_unit computes the result of the MixColumns operation for a masked column (i.e., 4 masked bytes). The ShiftRows layer is free, being implemented as a specific routing at the input of the SubBytes layer. In particular, the ordering of the bytes routed to the S-boxes (denoted sh_4bytes_to_SB) is selected such that the rotations over the rows are applied. Dedicated MUXes (controlled by route_MC) are used in order to bypass the MixColumns logic block when executing the last round. Other MUXes (controlled by loop) are used during the last key addition in order to bypass the ShiftRows, SubBytes and MixColumns layers. When a new execution starts, the masked plaintext bytes are loaded in the register through the MUXes controlled by init. Then, the AddRoundKey and ShiftRows layers are executed by propagating the data across the pipeline to the S-boxes. The MixColumns operation is performed when the result of the SubBytes layer is coming back to the core by asserting the signal route_MC.

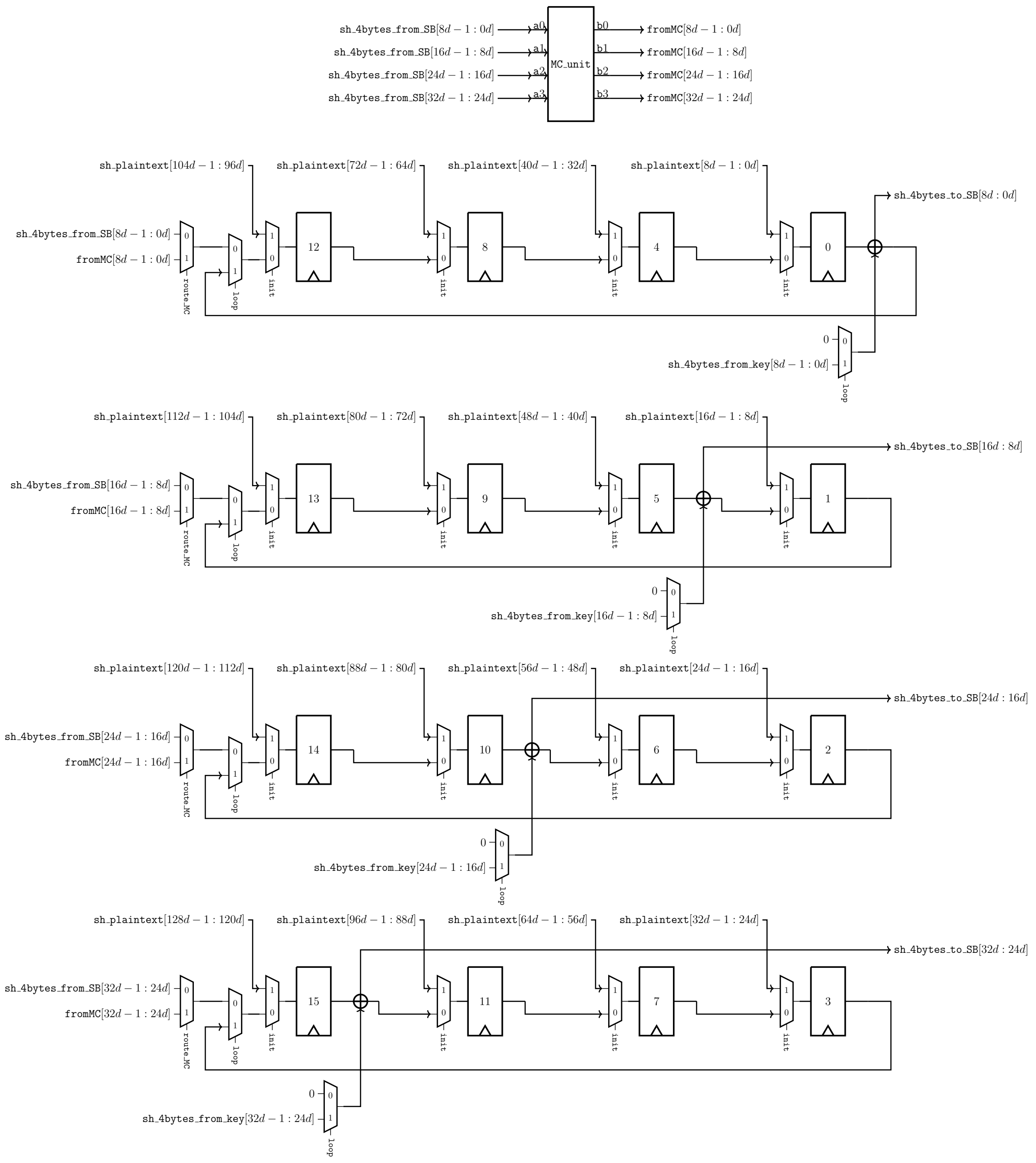
sh_4bytes_from_SB[8d − 1 : 0d]
sh_4bytes_from_SB[16d − 1 : 8d]
sh_4bytes_from_SB[24d − 1 : 16d]
sh_4bytes_from_SB[32d − 1 : 24d]
a0
a1
a2
a3
MC_unit
b0
b1
b2
b3
fromMC[8d − 1 : 0d]
fromMC[16d − 1 : 8d]
fromMC[24d − 1 : 16d]
fromMC[32d − 1 : 24d]
sh_plaintext[104d − 1 : 96d]
sh_plaintext[72d − 1 : 64d]
sh_plaintext[40d − 1 : 32d]
sh_plaintext[8d − 1 : 0d]
sh_4bytes_to_SB[8d : 0d]
route_MC
loop
init
12
8
4
0
sh_4bytes_from_key[8d − 1 : 0d]
sh_plaintext[112d − 1 : 104d]
sh_plaintext[80d − 1 : 72d]
sh_plaintext[48d − 1 : 40d]
sh_plaintext[16d − 1 : 8d]
sh_4bytes_to_SB[16d : 8d]
13
9
5
1
sh_4bytes_from_key[16d − 1 : 8d]
sh_plaintext[120d − 1 : 112d]
sh_plaintext[88d − 1 : 80d]
sh_plaintext[56d − 1 : 48d]
sh_plaintext[24d − 1 : 16d]
sh_4bytes_to_SB[24d : 16d]
14
10
6
2
sh_4bytes_from_key[24d − 1 : 16d]
sh_plaintext[128d − 1 : 120d]
sh_plaintext[96d − 1 : 88d]
sh_plaintext[64d − 1 : 56d]
sh_plaintext[32d − 1 : 24d]
sh_4bytes_to_SB[32d : 24d]
15
11
7
3
sh_4bytes_from_key[32d − 1 : 24d]

Figure 9: Global architecture of the `MSKaes_32bits_state_datapath`module. The value held by the DFF at index *i* is depicted by the signal `sh_reg_out[i]` in the HDL.

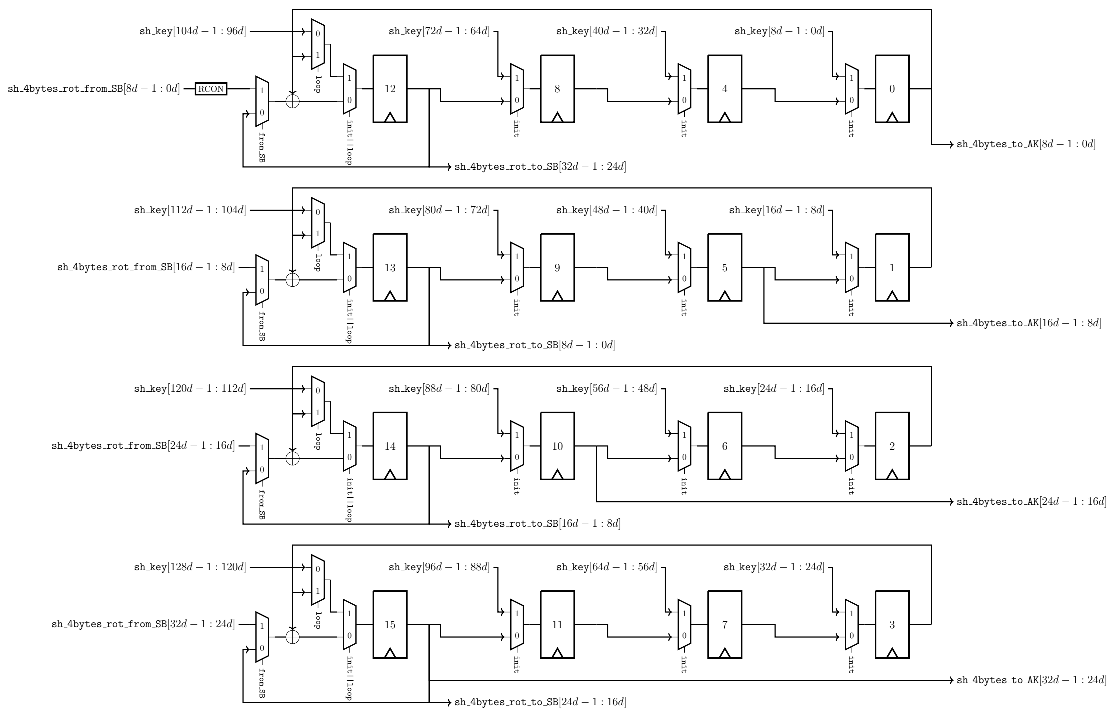
sh_key[104d − 1 : 96d]
sh_4bytes_rot_from_SB[8d − 1 : 0d]
RCON
loop
from_SB
init||loop
12
sh_key[72d − 1 : 64d]
8
sh_key[40d − 1 : 32d]
4
sh_key[8d − 1 : 0d]
0
init
sh_4bytes_to_AK[8d − 1 : 0d]
sh_4bytes_rot_to_SB[32d − 1 : 24d]
sh_key[112d − 1 : 104d]
sh_4bytes_rot_from_SB[16d − 1 : 8d]
13
sh_key[80d − 1 : 72d]
9
sh_key[48d − 1 : 40d]
5
sh_key[16d − 1 : 8d]
1
sh_4bytes_to_AK[16d − 1 : 8d]
sh_4bytes_rot_to_SB[8d − 1 : 0d]
sh_key[120d − 1 : 112d]
sh_4bytes_rot_from_SB[24d − 1 : 16d]
14
sh_key[88d − 1 : 80d]
10
sh_key[56d − 1 : 48d]
6
sh_key[24d − 1 : 16d]
2
sh_4bytes_to_AK[24d − 1 : 16d]
sh_4bytes_rot_to_SB[16d − 1 : 8d]
sh_key[128d − 1 : 120d]
sh_4bytes_rot_from_SB[32d − 1 : 24d]
15
sh_key[96d − 1 : 88d]
11
sh_key[64d − 1 : 56d]
7
sh_key[32d − 1 : 24d]
3
sh_4bytes_to_AK[32d − 1 : 24d]
sh_4bytes_rot_to_SB[24d − 1 : 16d]

Figure 10: Global architecture of the module `MSKaes_32bits_key_datapath`. The value held by the DFF at index *i* is depicted by the signal `sh_m_key[i]` in the HDL.

5.3 Architecture of the `MSKaes_32bits_key_datapath` module

The module `MSKaes_32bits_key_datapath` is shown in Figure 10. It is organized as a shift register where each register unit holds a masked byte of the key (the numbers on the Figure indicate the byte index in the unmasked key). The module is split in 4 independent parts, each taking care of the key scheduling operation on a single row. The sharing of the 128-bit key is routed from the input with the control signal `init`.

Concretely, the key scheduling starts by sending the last column of the key (i.e., byte indexes 12, 13, 14 and 15) to the S-boxes. The `RotWord` operation is performed by the routing that sends the key bytes to the S-boxes. Once computed, the result of the `SubBytes` layer is routed back to the core through the MUX controlled by the signal `from_SB`. At the same time, the round constant is applied and the first column (i.e., byte indexes 0,1,2 and 3) of the new key is computed by adding its value to the column coming back from the S-boxes. The remaining three columns (i.e., byte indexes [4,5,6,7], [8,9,10,11] and [12,13,14,15] are then updated sequentially by XORing each bytes with the value of the last byte updated in the same row. The signal `loop` is used to make the key shares loop across the key pipeline. This is required to keep the key material after the `AddRoundKey` operations while the `SubBytes` results of the key scheduling is still under computation.

```
%%% First key addition
for $0 \leq i < 16$ do
    $S_i^0 = P_i \oplus K_i$;
done

%%% Perform the rounds
for $0 \leq r < 9$ do
    % Operation for a single round
    $SR^r$ = ShiftRows($S^r$);
    $SB^r$ = SubBytes($SR^r$);
    $MC^r$ = MixColumns($SB^r$);
    $AK^r$ = AddRoundKey($MC^r$,$RK^r$);
    $S^{r+1} = AK^r$;
done

%%% Last round
$SR^9$ = ShiftRows($S^9$);
$SB^9$ = SubBytes($SR^9$);
$AK^9$ = AddRoundKey($SB^9$);
$C = AK^9$;
```

Figure 11: Pseudo-code of the AES encryption.

5.4 Internal operation

Let us first introduce notations for the intermediate states in the AES algorithm with pseudo-code in Figure 11 and Figure 12. Each variable denotes a state or subkey byte at a given step of the algorithm. In particular, the plaintext (resp. key, ciphertext) byte at index $0 \leq i < 16$ is denoted P_i (resp. K_i, C_i), and the value S_i^r (resp. RK_i^r) denotes the byte at index i of the state (resp. round key) starting the r-th round. When no index is given, the full 128-bit state is considered instead.

Using these notations, Figures 13, 14 and 15 describe the evolution of the AES states stored in the architecture over the computation of one round. Next, Figures 16, 17 and 18 depict the control signals that drive the datapath for the first round, middle rounds, and last round. In particular, for the first round (Figure 16), the data is fetched by the module when the signal `valid_in` is asserted if the core is not busy, there is no ciphertext stored in the core and randomness is available. At the next clock cycle, the internal FSM counters `cnt_round` and `cnt_fsm` are reset and the execution begins. The round function and the key scheduling algorithm are executed in parallel by interleaving the S-boxes usage appropriately. In particular, the first cycle of the execution is used to start the key scheduling algorithm by asserting `feed_sb_key` and `sbox_valid_in`. During this cycle, both the module `MSKaes_32bits_state_datapath` and `MSKaes_32bits_key_datapath` are disabled.

Then, the core enters into a nominal regime that computes a round in 9 cycles, as depicted in Figure 17. A typical round starts with 4 clock cycles during which data is read from the state registers, XORed with the subkey and fed to the S-boxes, which performs the `AddRoundKey`, `ShiftRows` and `SubBytes` layers for the full state (one column

```
%%% Key evolution for each round key
for $0 \leq r < 10$ do
    % Fetch value on which operate
    if $r == 0$ then
        $t^r = \mathtt{K}$;
    else
        $t^r = \mathtt{RK}^{r-1}$;
    end

    % Perform the last column rotation
    $[\mathtt{R}^r_0, \mathtt{R}^r_1, \mathtt{R}^r_2, \mathtt{R}^r_3] = [t^r_{13}, t^r_{14}, t^r_{15}, t^r_{12}]$;

    % Perform SubWord on the rotated column
    $[\mathtt{RSB}^r_0, \mathtt{RSB}^r_1, \mathtt{RSB}^r_2, \mathtt{RSB}^r_3] = [\mathtt{SubWord}(\mathtt{R}^r_0), \mathtt{SubWord}(\mathtt{R}^r_1), \mathtt{SubWord}(\mathtt{R}^r_2), \mathtt{SubWord}(\mathtt{R}^r_3)]$

    % Compute the first column of the next round key
    $\mathtt{RK}^r_0 = \mathtt{RSB}^r_0 \oplus t^r_0 \oplus \mathtt{RCON}^r$;
    $\mathtt{RK}^r_1 = \mathtt{RSB}^r_1 \oplus t^r_1$;
    $\mathtt{RK}^r_2 = \mathtt{RSB}^r_2 \oplus t^r_2$;
    $\mathtt{RK}^r_3 = \mathtt{RSB}^r_3 \oplus t^r_3$;

    % Generate the three remaining columns
    for $1 \leq i < 4$ do
        for $0 \leq j < 4$ do
            $\mathtt{RK}^r_{4i+j} = \mathtt{RK}^r_{4(i-1)+j} \oplus t^r_{4i+j}$;
        done
    done
done
```

Figure 12: Pseudo-code for the AES key evolution.

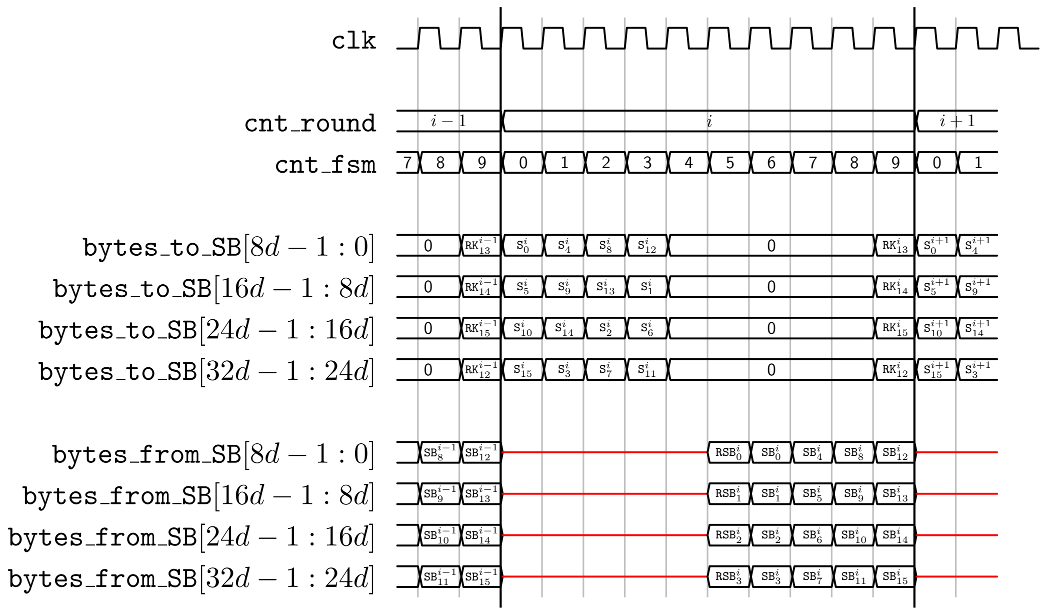
clk
cnt_round
cnt_fsm
bytes_to_SB[8d − 1 : 0]
bytes_to_SB[16d − 1 : 8d]
bytes_to_SB[24d − 1 : 16d]
bytes_to_SB[32d − 1 : 24d]
bytes_from_SB[8d − 1 : 0]
bytes_from_SB[16d − 1 : 8d]
bytes_from_SB[24d − 1 : 16d]
bytes_from_SB[32d − 1 : 24d]
i − 1
i
i + 1

Figure 13: Data going into / coming from the S-boxes during a round.

per cycle). During these cycles, `sbox_valid_in` is asserted and data (state and subkey) loops over the shift registers. At the fifth cycle of a round (i.e., when `cnt_fsm` $= 4$), the module `MSKaes_32bits_key_datapath` is disabled in order to wait one cycle for the S-box results. At the same cycle, the S-boxes output the column of the new subkey value, which is processed by enabling the module `MSKaes_32bits_key_datapath` and asserting `from_SB` for one cycle. Next, during the last 4 cycles of a round, the S-boxes output the 4 columns of the state, on which the `MixColumns` layer is directly applied, and the result is stored in the state registers. At the same time, the subkey update is finalized, such that a new subkey is ready at the last cycle of a round (i.e., `cnt_fsm` $= 9$). During this last cycle, the next key schedule round is started, and a new state round starts at the following cycle.

Finally, the last round is very similar to the regime mode except that the module `MC_unit`is bypassed. In particular, the signal `route_MC` is de-asserted and the shift registers are configured to make the data loop. No new key scheduling round is started during this last cycle. At the end of the last round, once the ciphertext has been fetched from the output, a new encryption starts immediately (if `valid_in` is asserted), or the state register is cleared by asserting the control signal `init`. This ensures that the core is completely clear of any key- or plaintext-dependent data.

5.5 Randomness Generation

The module `prng_top` is a PRNG generating `NRNDBITS` $= 4 \cdot 34 \cdot$ `MUL_HPC2_RND` $= 4 \cdot 34 \cdot [d(d-1)/2]$ pseudo-random bits per clock cycle. It is based on one or multiple instances of the Trivium stream cipher [CP08] from which the key stream is used as the PRNG

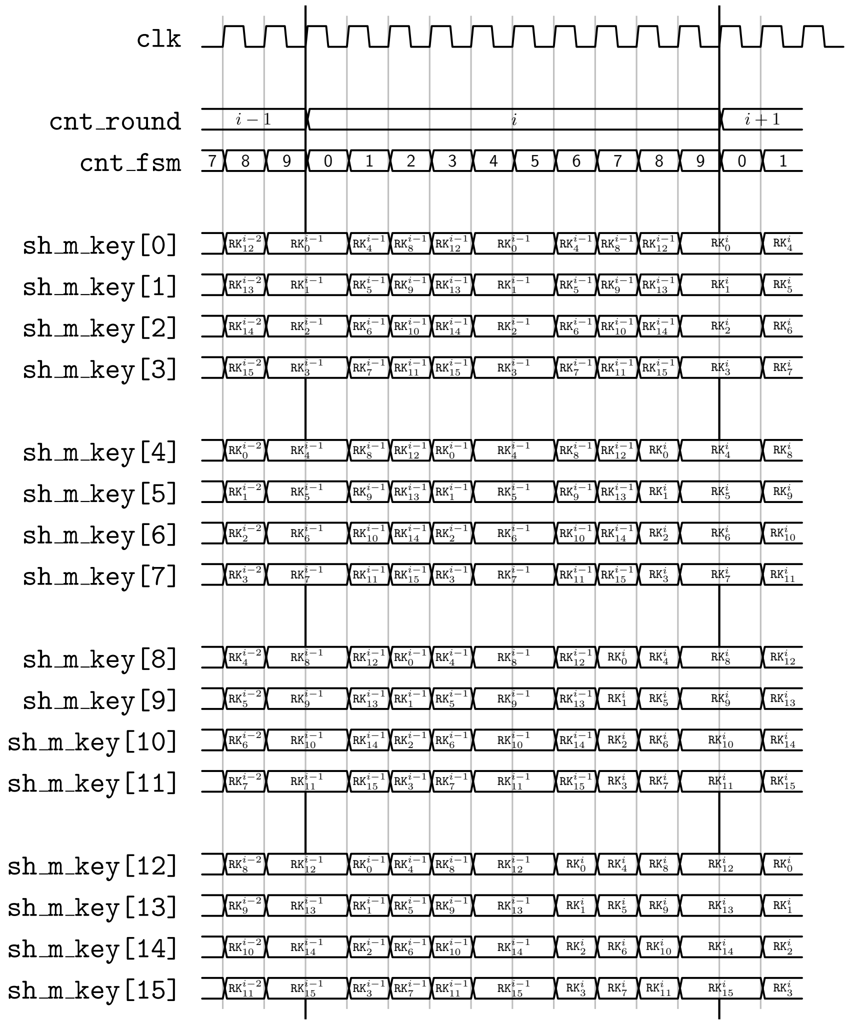

Figure 14: Data going into / coming from the key scheduling datapath during a round.

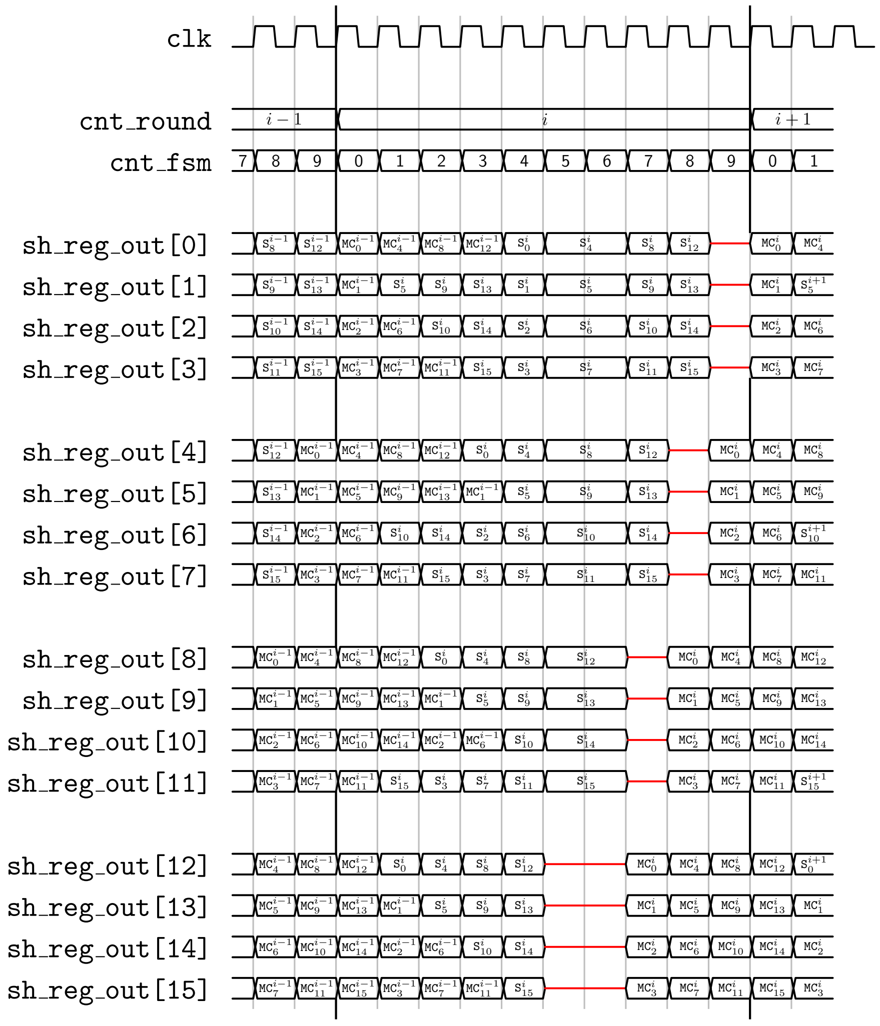
Figure 15: Data going into / coming from the round function datapath during a round.

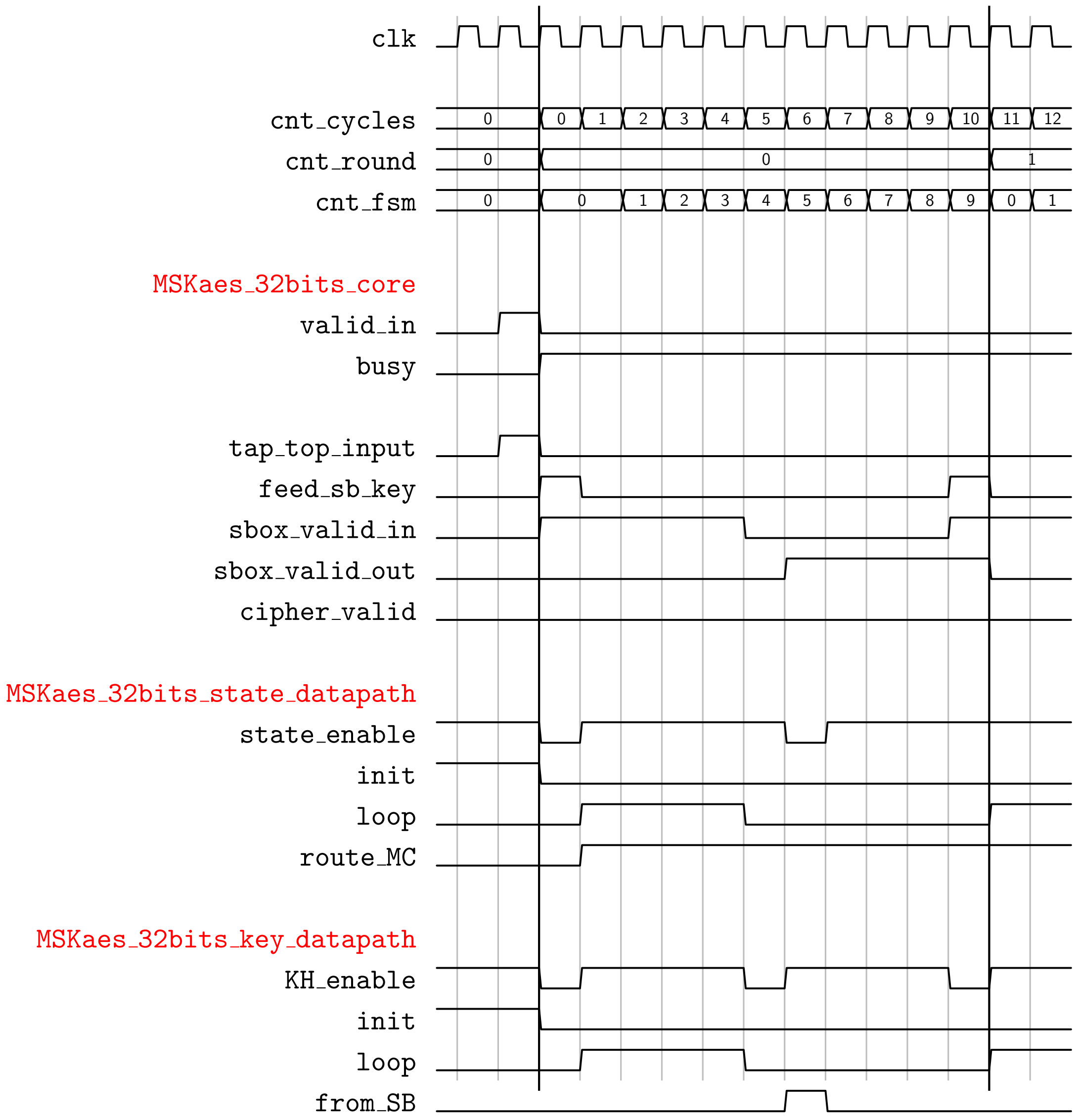

Figure 16: Data routing when a new execution starts.

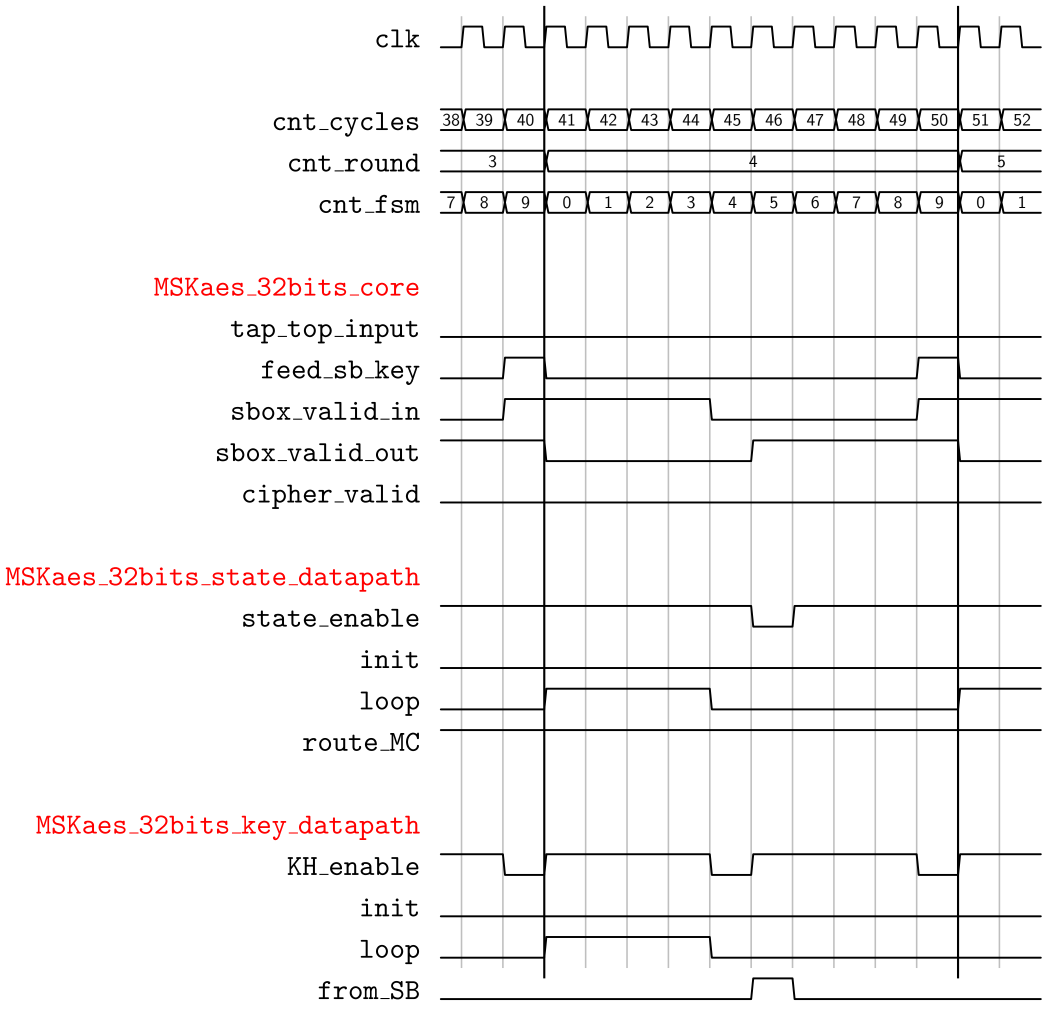

Figure 17: In regime data routing.

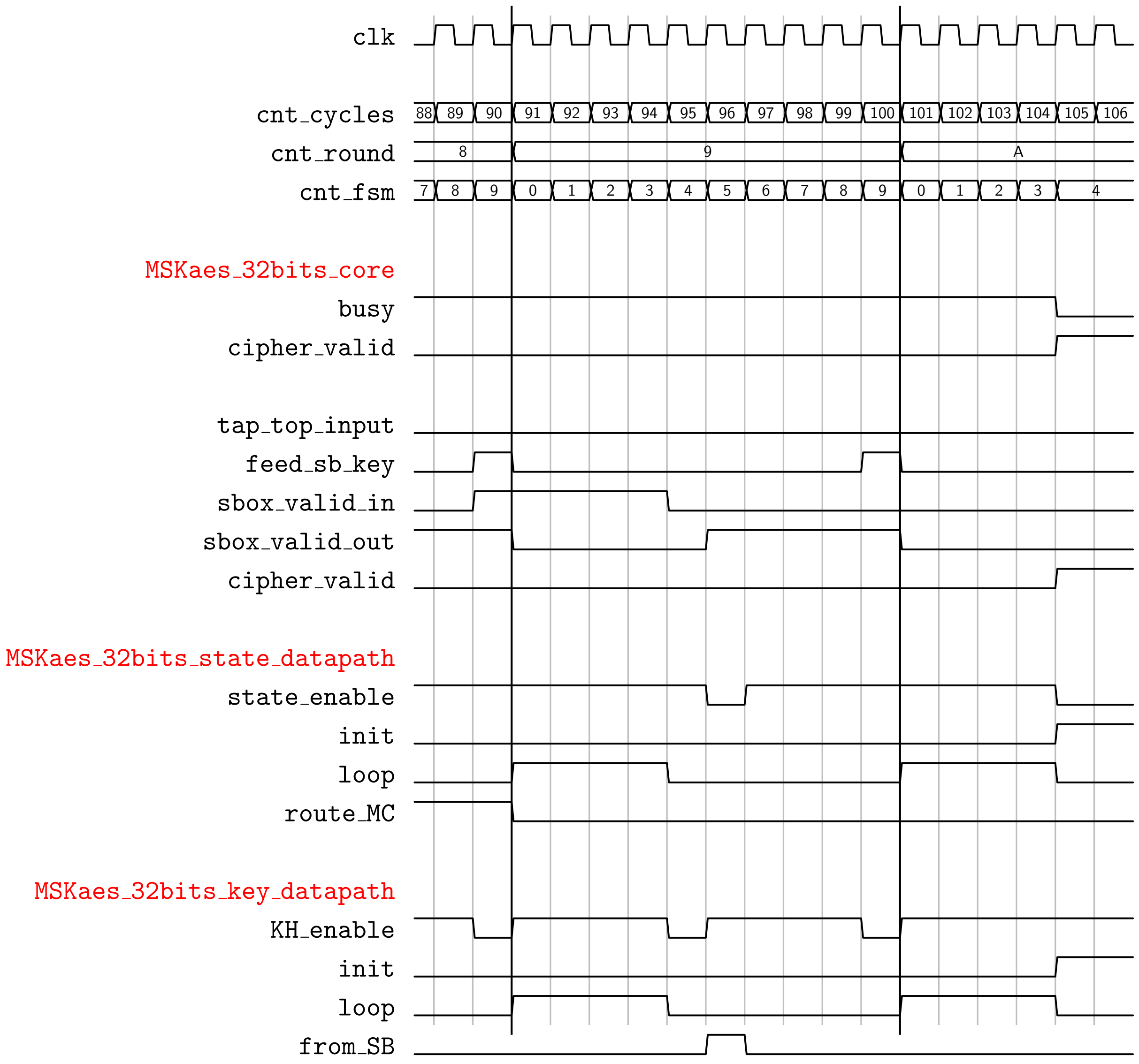

Figure 18: Data routing during last rounds.

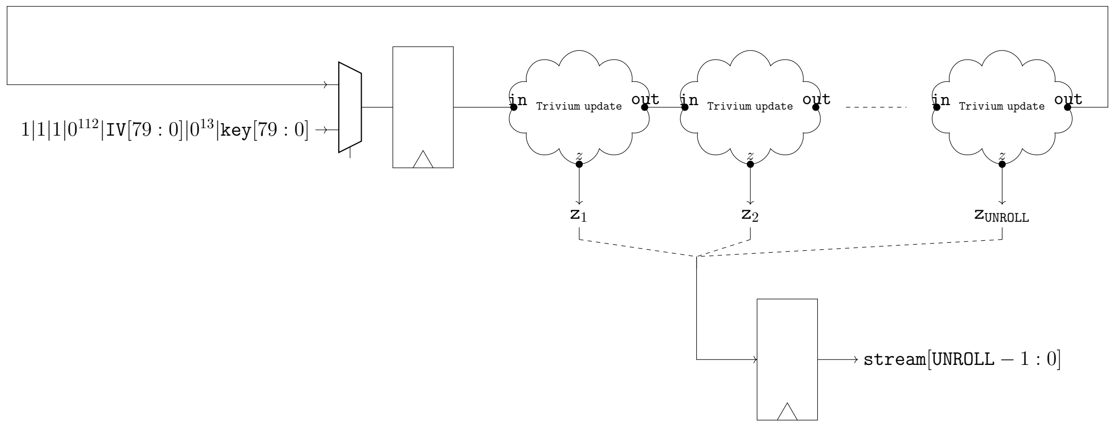

Figure 19: Datapath Architecture of a unrolled Trivium module

output. As shown in Figure 19, a Trivium instance is implemented using a 288-bit state register and `UNROLL` cascaded combinational layers that each implement one state update step and produce one keystream bit. Moreover, the state register is either taken from a reseed value (to initiate a reseed), or from the output of the final update step (during normal operation). At the output, the keystream is stored in a register to avoid the propagation of glitches that could reduce the security of the masked circuit.

The use of multiple Trivium instances allows us to adjust the area-latency trade-off: with more Trivium instances, `UNROLL` can be reduced, leading to a lower combinational logic depth. The top-level `PRNG_MAX_UNROLL` parameter is used for this purpose: the number of instances is $\texttt{NTRIVIUMS} = \lceil \texttt{NRNDBITS/PRNG\_MAX\_UNROLL} \rceil$, and $\texttt{UNROLL} = \lceil \texttt{NRNDBITS/NTRIVIUMS} \rceil$, which ensures that $\texttt{UNROLL} \leq \texttt{PRNG\_MAX\_UNROLL}$.

The reseeding follows the initialization of Trivium. Concretely, the state is first set to $\texttt{1}^3|\texttt{0}^{112}|\texttt{IV}|\texttt{0}^{13}|\texttt{KEY}$, where the `KEY` is set to the 80-bit externally provided seed (it is the same for all Trivium instances), while the `IV` is a constant, which is distinct for each Trivium instance. Then, the update function is applied at least $4 \cdot 288$ times, i.e., the PRNG is executed while feeding back its state for $4 \cdot 288/\texttt{UNROLL}$ cycles. During the reseed, the signal `busy` is asserted and `out_valid` is not. Once finished, the signal `out_valid` is asserted. After a reset, the core requires will not output valid data (i.e., `out_valid` will stay de-asserted) until the completion of a reseed.

6 Core Performances

Following the architecture described section 5.2, the latency is 105 cycles per execution. Table 2 contains performance metrics for two implementations on a Xilinx Series 7 FPGA: one with the optimisations disabled at every steps of the toolflow and one with the optimisation enabled.

Part	Shares (d)	Slices	LUTs	Regs	SRLs	Latency [cycle]	Freq. [MHz]	TP [Mbit/s]
xc7a100tftg256-2	2	1495	4664	3502	0	105	68.6	83.6
(opt. disabled[1])	3	3069	9960	7157	0	105	66.2	80.7
	4	4829	16 527	12 189	0	105	66.5	81.0
xc7a100tftg256-2	2	910	2950	3145	144	105	159	193
	3	1868	6071	6620	216	105	137.9	168.1
	4	3244	10 759	11 474	288	105	128.76	156.96

Table 2: Artix-7 FPGA synthesis results (`out_of_context`, post-implementation, `PRNG_MAX_UNROLL` = 128).

[1] Optimizations might impact the side-channel security of the implementation, see section 7.

7 Core Verification

Functionality In order to ensure the proper functionality of the AES core, the Known-Answer Tests of the NIST "Advanced Encryption Standard Algorithm Validation List" is verified with the provided testbench[1].

In particular, all the testvectors related to the encryption algorithm from the files `ECBGFSbox128.rsp`, `ECBKeySbox128.rsp`, `ECBVarKey128.rsp` and `ECBVarTxt128.rsp` are tested at the RTL level. The testbench follows a randomized regression testing strategy to assess the functionality of the module. In particular, the execution related to each testvector cases is started sequentially by performing a transaction at the input interface. To simulate the behavior that may happen due to the integration of the core in a more complex system, a random amount of clock cycles is waited before initializing a transaction (i.e., before asserting the `in_valid` signal). Similarly, in order to simulate (hard) back-pressure conditions that may occur in practice, the output interface is simulated with random assertion of the `out_ready` signal. Besides, in parallel to the behavioral known-answer tests, the reseeding procedure is tested by issuing reseed requests at regular interval. This is achieved by waiting a random amount of clock cycles before asserting the signal `in_seed_valid` and waiting until a transaction at the seed interface occurs.

Additionally, a practical implementation on an Artix7 FPGA (xc7a100tftg256-2) has been tested with random known-test vectors (i.e., random key, plaintext and seed).

Side-channel security This core has been formally verified for security in the glitch+transition robust probing model using the `fullVerif`tool [CGLS21, CS21][2]. The scripts for this verification are provided along with the implementation. The implementation has also been empirically evaluated on an FPGA (with synthesis optimizations disabled), the

[1]https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/block-ciphers
[2]`https://github.com/cassiersg/fullverif`

evaluation report is available at `https://simple-crypto.org/outputs`. Note that this evaluation is device-specific, and should be performed on every instantiation of this device.

8 Copyright

References

[BP12] Joan Boyar and René Peralta. A small depth-16 circuit for the AES s-box. In *SEC*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 287–298. Springer, 2012.

[CGLS21] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware private circuits: From trivial composition to full verification. *IEEE Trans. Computers*, 70(10):1677–1690, 2021.

[CP08] Christophe De Cannière and Bart Preneel. Trivium. In Matthew J. B. Robshaw and Olivier Billet, editors, *New Stream Cipher Designs - The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 244–266. Springer, 2008.

[CS21] Gaëtan Cassiers and François-Xavier Standaert. Provably secure hardware masking in the transition- and glitch-robust probing model: Better safe than sorry. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):136–158, 2021.

[MCS22] Charles Momin, Gaëtan Cassiers, and François-Xavier Standaert. Handcrafting: Improving automated masking in hardware with manual optimizations. In *COSADE*, volume 13211 of *Lecture Notes in Computer Science*, pages 257–275. Springer, 2022.

[NIS01] NIST. Advanced Encryption Standard (AES), 2001.