Session 1: The iterative architecture

Antonio de la Piedra

Institute for Computing and Information Sciences – Digital Security
Radboud University Nijmegen

February 18, 2015

Session 1

- The goal of this session is to show the basic VHDL constructions in a real implementation of a block cipher (NOEKEON)
- Introduce the iterative architecture
- The implementation that is used in this session can be download from `https://github.com/adelapie/noekeon/archive/master.zip`

Overview

1. Design parameters
2. The iterative architecture
3. The NOEKEON block cipher
   - The γ function
   - The θ function
   - The Π_1 and Π_2 functions
4. An iterative architecture for NOEKEON

How this is related to your projects

1. Start by implementing the block cipher you have selected in a high-level programming language e.g. C, Python, etc.
2. Read and understand the basic constructions of VHDL (e.g. *Circuit Design with VHDL, Volnei A. Pedroni* is a good book or another from the library you can find).
3. Study other implementations, e.g. NOEKEON and XTEA are two easy block ciphers:
   - `https://github.com/adelapie/noekeon/archive/master.zip`
   - `http://opencores.org/project,xteacore`
4. Implement the basic functions of your block cipher in VHDL and create testbenches that prove their correctness.
5. Implement a round and create a testbench for it
6. Depending on your project:
   - Adapt the round to an iterative architecture
   - Replicate the round for pipelining accordingly

Important

- Follow the instructions available at `https://rucryptoengineering.wordpress.com/fpga-assignments/` for installing Xilinx ISE and ModelSim.
- The theory behind this chapter is based on
  - *Cryptographic Engineering, Koc, Cetin Kaya (Ed.) 2009*
  - *Joan Daemen, Michael Peeters, Gilles Van Assche, Vincent Rijmen, The NOEKEON Block Cipher, Nessie Proposal*

Starting

- Implement the basic functions of your block cipher in VHDL and create testbenches that prove their correctness.
  - KLEIN (SP network): AddRoundKey, SubNibbles, RotateNibbles, MixNibbles
  - TWINE (Feistel network): Round function (4-bit XOR, S-BOX), KeySchedule
  - SIMON (Feistel network): Round function (shifts, XOR), Key expansion
  - RECTANGLE (SP network): AddRoundkey, SubColumn, ShiftRow, Key Schedule
  - LED (SP network): addRoundKey, AddConstants, SubCells, ShiftRows, MixColumnsSerial, KeySchedule

Part I: Design parameters

Minimum clock delay

- The maximum clock frequency (f_{CLK}) of a synchronous sequential circuit is limited by:
  - Propagation delays of FFs and gates
  - Setup time and hold time in FFs
- The minimum clock delay is its inverse $T_{CLK} = \frac{1}{f_{CLK}}$

Throughput

- Number of bits encrypted/decrypted in a certain time unit (e.g. seconds **s**).
- Generally the same in both operations.
- Typical units:
  - 1 Mbit/s $= 10^6$ bits/s
  - 1 Gbit/s $= 10^9$ bits/s

Latency

- Time needed for encrypting/decrypting a block of plaintext/ciphertext.
- The typical unit is nanoseconds (**ns**).

Both parameters are interrelated through the following formula:

Throughput

$$Throughput = \frac{blocksize \cdot \#blocks\ \ processed\ \ simultaneously}{latency}$$

Latency

- Latency and minimum clock delay are interrelated according to the number of rounds of the block cipher.
- For a certain round circuit implementation with minimum clock delay T_{CLK}, Latency is computed as:

Throughput

$$Latency = rounds \cdot T_{CLK}$$

Example 1

Given a simple hardware implementation of the AES-128 block cipher with the following parameters:

- 1 block processed at a time
- Minimum clock delay (T_{CLK}) of one round: 0.0025 ns.
- AES-128:
  - Blocks of 128 bits
  - 10 rounds

Extract:

- Maximum frequency f_{CLK}
- Latency
- Throughput

Example 1

- Maximum frequency f_{CLK}
  - $f_{CLK} = \frac{1}{0.0025x10^{-9}} = 400$ GHz
- Latency
  - $Latency = rounds \cdot T_{CLK}$
  - $Latency = 10 \cdot 0.0025$ ns $= 0.025$ ns
- Throughput
  - $Throughput = \frac{blocksize \cdot \#blocks \quad processed \quad simultaneously}{latency}$
  - $Throughput = \frac{128 \cdot 1}{0.025x10^{-9}} = 5,120$ Gbits/s

Part II: The iterative architecture

Architectures

- **Iterative**
- Loop unrolling
- Pipelining
  - partial outer-round
  - full outer-round
  - partial inter-round
  - full inter-round
  - mixed outer-round inter-round

Iterative

- One round of the block cipher is implemented as combinational logic.
- Elements:
  - 1x Register
  - 1x Multiplexer
- Behaviour:
  - First cycle of clock: the plaintext goes through the mux and is stored in the register.
  - Subsequent cycles: one round is evaluated, the result is fed back through the mux and stored in the register.

Iterative

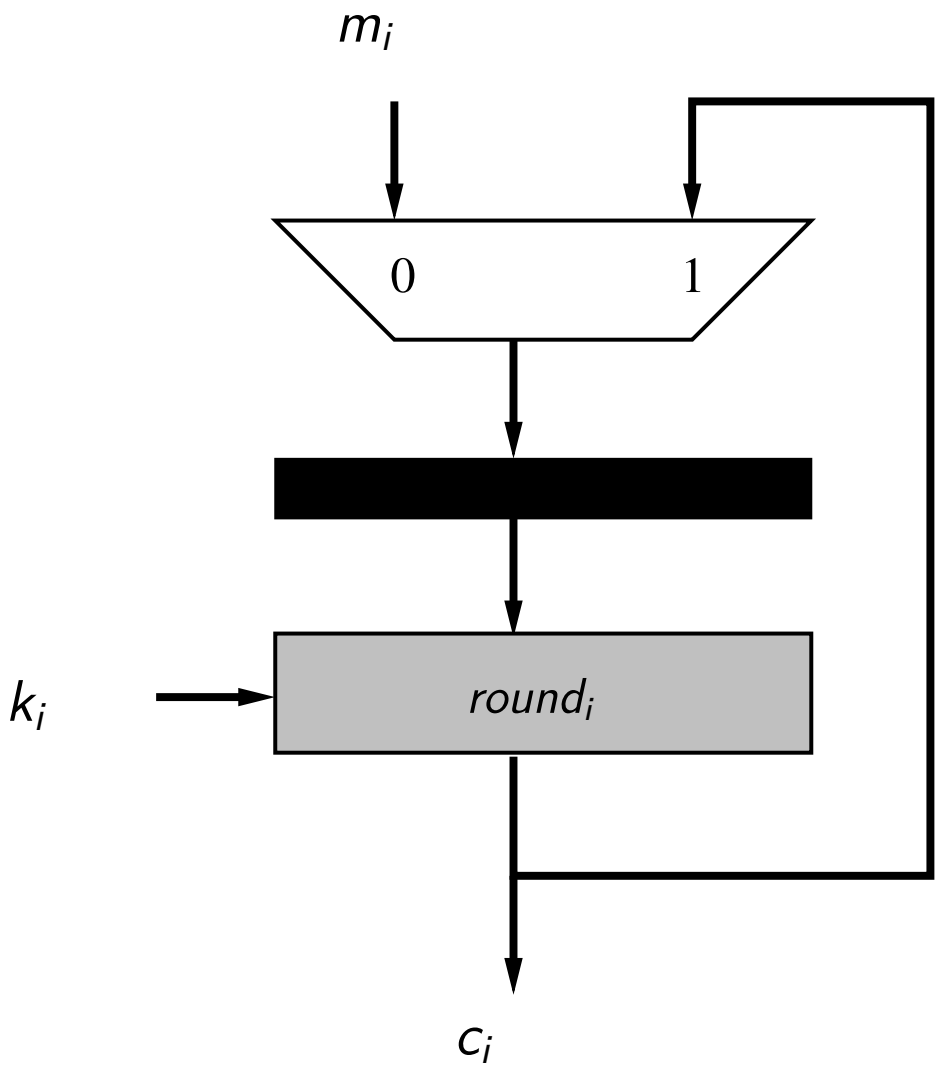


Figure : Iterative architecture of a block cipher

Iterative

- Useful for low-cost and low-power applications (area)
- Only one block of data is encrypted at the same time
- The number of clock cycles needed for encrypting a data block is equivalent to the number of rounds of the block cipher

Iterative

Throughput

$$Throughput = \frac{blocksize}{rounds \cdot T_{CLK}}$$

Latency

$$Latency = rounds \cdot T_{CLK}$$

Where T_{CLK} is the minimum clock period of the circuit and $\frac{1}{T_{CLK}}$ its maximum frequency.

Part III: The NOEKEON block cipher

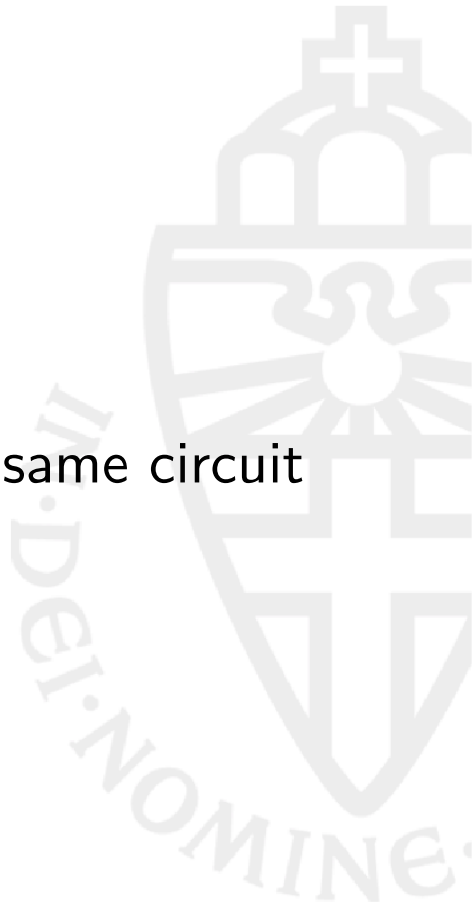
The NOEKEON block cipher

- *So simple that it can be memorized by an average person!*[1]
- 128-bit blocks and keys
- 16 rounds
- The round function is based on two functions (θ and γ) and bitwise shift operations.
- Based on 32-bit logic operations.

[1] Nessie Workshop 2001, On Noekeon, no!. Joan Daemen, Michael Peeters, Gilles Van Assche and Vincent Rijmen NESSIE (New European Schemes for Signatures, Integrity and Encryption)

Main functions of NOEKEON

- Simple and fast
- Supports CBC, CFB, OFB and counter mode
- Direct mode: key schedule is not applied
- The inverse operation can be performed with the same circuit

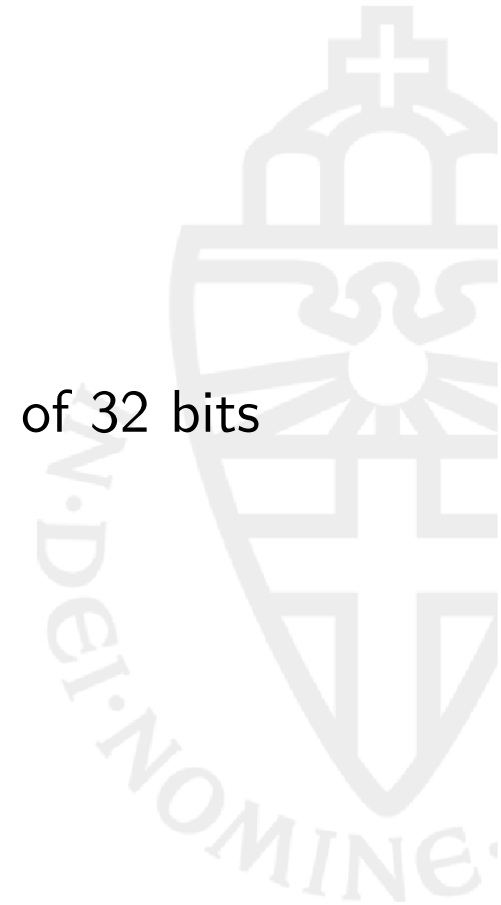


Main functions of NOEKEON

- θ is a linear operation based on the round key.
- γ adds non-linearity to the state.
- Π_1, Π_2: permutations.

The internal state

- a of 128 bits.
- a is divided into 4 blocks of 32 bits
  - $a = (a_0, a_1, a_2, a_3)$.
- The key k is also 128 bits, divided in four blocks of 32 bits
  - $k = (k_0, k_1, k_2, k_3)$.



Part IV: An iterative architecture for NOEKEON

Example 2: Designing the top block of a hardware implementation of NOEKEON

- Which signals do you think are important?
  - Data signals?
  - Control signals?
  - Clock signals?
- Which ports and which length would you use?
- Which direction?

Example 2: Designing the top block of a hardware implementation of NOEKEON

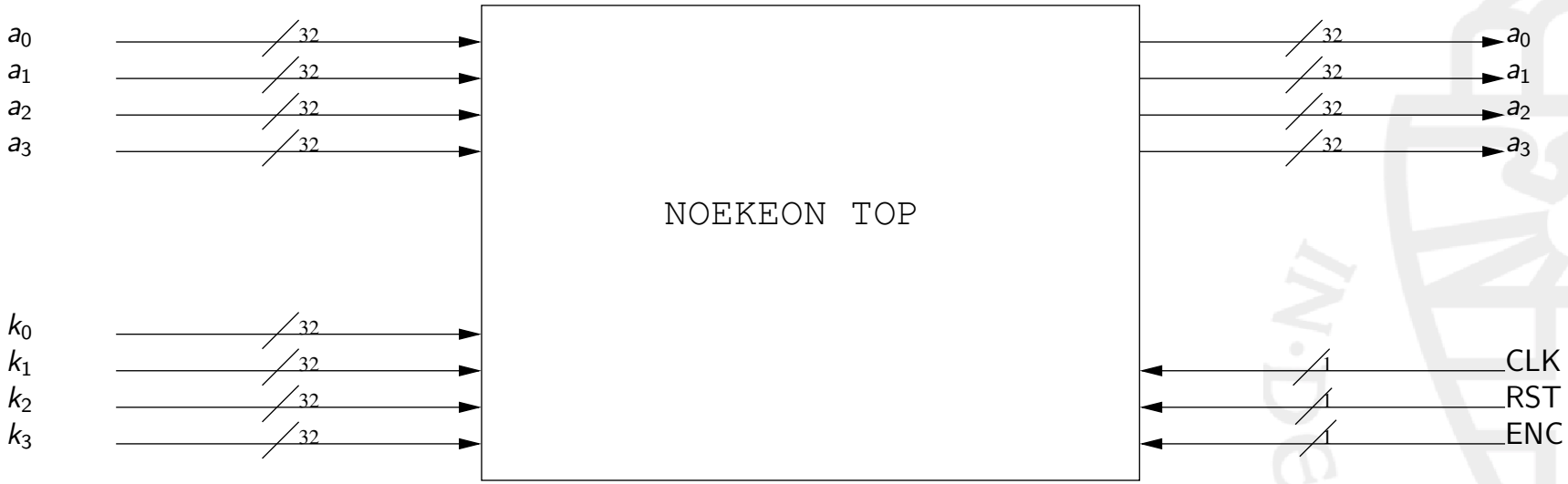


Figure : Top block of NOEKEON

Example 2: Designing the top block of a hardware implementation of NOEKEON

```
entity noekeon is
        port(       clk      : in std_logic;
                    rst      : in std_logic;
                    enc      : in std_logic; -- (enc, 0) / (dec, 1)
                    a_0_in   : in std_logic_vector(31 downto 0);
                    a_1_in   : in std_logic_vector(31 downto 0);
                    a_2_in   : in std_logic_vector(31 downto 0);
                    a_3_in   : in std_logic_vector(31 downto 0);
                    k_0_in   : in std_logic_vector(31 downto 0);
                    k_1_in   : in std_logic_vector(31 downto 0);
                    k_2_in   : in std_logic_vector(31 downto 0);
                    k_3_in   : in std_logic_vector(31 downto 0);
                    a_0_out  : out std_logic_vector(31 downto 0);
                    a_1_out  : out std_logic_vector(31 downto 0);
                    a_2_out  : out std_logic_vector(31 downto 0);
                    a_3_out  : out std_logic_vector(31 downto 0));
end noekeon;
```

A round in NOEKEON

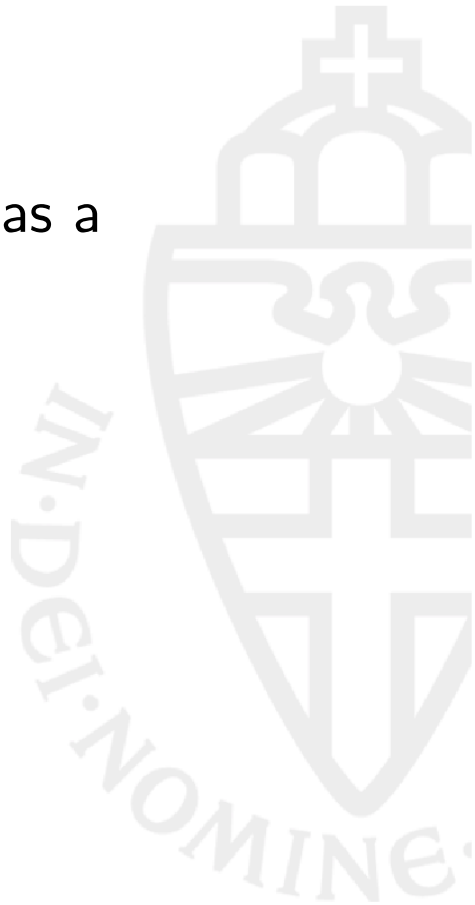
Given:

- State a of 128 bits as $a = (a_0, a_1, a_2, a_3)$.
- Key k, divided in four blocks of 32 bits as $k = (k_0, k_1, k_2, k_3)$
- Two constants per round c_1, c_2

```
Round(k, a, c1, c2, enc) {
   if (enc)
      a ^= c1;

   theta(k, a);

   if (!enc)
      a ^= c2;

   pi1(a);
   gamma(a);
   pi2(a);
}
```

Methodology

- We follow a bottom-up approach
- We start by implementing each required function as a combinational circuit:
  - The γ function
  - The θ function
  - The Π_1 and Π_2 functions
  - The constant geneneration function



- Adds non-linearity to the state
- Involution property
- It can be implemented as S-BOX in ROM

Algorithm 1 $\gamma(a)$.

Require: The state a of 128 bits divided in blocks of 32 bits a_i where $a = (a_0, a_1, a_2, a_3)$.

Ensure: The state $a(a_0, a_1, a_2, a_3)$ of 128-bit length after the non-linear transformation.

1: $t_0 \leftarrow \neg a_3 \wedge \neg a_2$
2: $t_0 \leftarrow \neg a_3 \wedge \neg a_2$
3: $a_1 \leftarrow a_1 \oplus t_0$
4: $t_1 \leftarrow a_2 \wedge a_1$
5: $a_0 \leftarrow a_0 \oplus t_1$
6: $swap(a_0, a_3)$
7: $t_2 \leftarrow a_2 \oplus a_0$
8: $t_3 \leftarrow a_1 \oplus a_3$
9: $a_2 \leftarrow t_2 \oplus t_3$
10: **return** (a_0, a_1, a_2, a_3)

Hardware implementation of the γ function

```
#define GAMMA(a0, a1, a2, a3) \
        a1 ^= ~ (a3 | a2); \
        a0 ^= a2 & a1; \
        t = a3; a3 = a0; a0 = t; \
        a2 ^= a0 ^ a1 ^ a3; \
        a1 ^= ~ (a3 | a2); \
        a0 ^= a2 & a1
```

An example implementation of γ using a C macro.

Hardware implementation of the γ function

The interface for gamma will be:

```
entity gamma is
        port(clk : in std_logic;
                   a_0_in : in std_logic_vector(31 downto 0);
                   a_1_in : in std_logic_vector(31 downto 0);
                   a_2_in : in std_logic_vector(31 downto 0);
                   a_3_in : in std_logic_vector(31 downto 0);

                   a_0_out : out std_logic_vector(31 downto 0);
                   a_1_out : out std_logic_vector(31 downto 0);
                   a_2_out : out std_logic_vector(31 downto 0);
                   a_3_out : out std_logic_vector(31 downto 0));
end gamma;
```

Hardware implementation of the γ function

- There are six operations
- The computation of a_1 is $a_1 = a_1 \oplus not(a_3||a_2)$, we expand this to $a_1 = a_1 \oplus not\ (a_3)\ \&not\ (a_2)$ by De Morgan's Laws
- We create a set of signals that will serve as wires of the full function

```
signal a_0_tmp_s : std_logic_vector(31 downto 0);
signal a_1_tmp_s : std_logic_vector(31 downto 0);
signal a_2_tmp_s : std_logic_vector(31 downto 0);
signal a_3_tmp_s : std_logic_vector(31 downto 0);
signal a_1_1_tmp_s : std_logic_vector(31 downto 0);
signal a_0_1_tmp_s : std_logic_vector(31 downto 0);
signal a_0_2_tmp_s : std_logic_vector(31 downto 0);
```

Hardware implementation of the γ function

```
-- a[1] ^= ~a[3]&~a[2];
        a_1_tmp_s <= a_1_in xor (not(a_3_in) and not(a_2_in));
-- a[2]& a[1];
        a_0_tmp_s <= a_0_in xor (a_2_in and a_1_tmp_s);

-- swapping (a[0], a[3])
        a_0_1_tmp_s <= a_3_in;
        a_3_tmp_s <=  a_0_tmp_s;

-- a[2] ^= a[0]^a[1]^a[3];
        a_2_tmp_s <= a_0_1_tmp_s xor a_1_tmp_s xor a_2_in xor a_3_tmp_s;
-- a[1] ^= ~a[3]&~a[2];
        a_1_1_tmp_s <= a_1_tmp_s xor (not(a_3_tmp_s) and not(a_2_tmp_s));
-- a[0] ^= a[2]& a[1];
        a_0_2_tmp_s <= a_0_1_tmp_s xor (a_2_tmp_s and a_1_1_tmp_s);

        a_3_out <= a_3_tmp_s;
        a_2_out <= a_2_tmp_s;
        a_1_out <= a_1_1_tmp_s;
        a_0_out <= a_0_2_tmp_s;
```

- θ is a linear operation based on the round key
- Involution property when $k = 0$
- During decryption the inverse of θ is computed with $k' = \theta(0, k)$.

Algorithm 2 $\theta(k, a)$.

Require: The state a of 128 bits divided in blocks of 32 bits a_i where $a = (a_0, a_1, a_2, a_3)$ and the round key $k = (k_0, k_1, k_2, k_3)$ of 128 bits.

Ensure: The state $a(a_0, a_1, a_2, a_3)$ of 128-bit length.

1: $t_0 \leftarrow a_0 \oplus a_2$
2: $t_0 \leftarrow t_0 \oplus t_0 \ggg 8$
3: $t_0 \leftarrow t_0 \oplus t_0 \lll 8$
4: $a_1 \leftarrow a_1 \oplus t_0$
5: $a_3 \leftarrow a_3 \oplus t_0$
6: $a_0 \leftarrow a_0 \oplus k_0$
7: $a_1 \leftarrow a_1 \oplus k_1$
8: $a_2 \leftarrow a_2 \oplus k_2$
9: $a_3 \leftarrow a_3 \oplus k_3$

Algorithm 3 $\theta(k, a)$.

Require: The state a of 128 bits divided in blocks of 32 bits a_i where $a = (a_0, a_1, a_2, a_3)$ and the round key $k = (k_0, k_1, k_2, k_3)$ of 128 bits.

Ensure: The state $a(a_0, a_1, a_2, a_3)$ of 128-bit length.

1: $t_1 \leftarrow a_1 \oplus a_3$
2: $t_1 \leftarrow t_1 \oplus t_1 \ggg 8$
3: $t_1 \leftarrow t_1 \oplus t_1 \lll 8$
4: $a_0 \leftarrow a_0 \oplus t_1$
5: $a_2 \leftarrow a_2 \oplus t_1$
6: **return** (a_0, a_1, a_2, a_3)

Hardware implementation of the θ function

```
#define THETA(a0, a1, a2, a3, k0, k1, k2, k3) \
        t  = a0 ^ a2; \
        t ^= ROL32(t, 8) ^ ROR32(t, 8); \
        a1 ^= t; \
        a3 ^= t; \
        a0 ^= k0; a1 ^= k1; a2 ^= k2; a3 ^= k3; \
        t = a1 ^ a3; \
        t ^= ROL32(t, 8) ^ ROR32(t, 8); \
        a0 ^= t; \
        a2 ^= t
```

An example implementation of θ using a C macro.

Hardware implementation of the θ function

The top block of θ can be defined as:

```
entity theta is
        port(
              a_0_in : in std_logic_vector(31 downto 0);
              a_1_in : in std_logic_vector(31 downto 0);
              a_2_in : in std_logic_vector(31 downto 0);
              a_3_in : in std_logic_vector(31 downto 0);

              k_0_in : in std_logic_vector(31 downto 0);
              k_1_in : in std_logic_vector(31 downto 0);
              k_2_in : in std_logic_vector(31 downto 0);
              k_3_in : in std_logic_vector(31 downto 0);

              a_0_out : out std_logic_vector(31 downto 0);
              a_1_out : out std_logic_vector(31 downto 0);
              a_2_out : out std_logic_vector(31 downto 0);
              a_3_out : out std_logic_vector(31 downto 0));
end theta;
```

Hardware implementation of the θ function

As before, we will design a set of signals that will serve as wires:

```
signal a_1_0_s : std_logic_vector(31 downto 0);
signal a_3_0_s : std_logic_vector(31 downto 0);

signal tmp_0_s : std_logic_vector(31 downto 0);
signal tmp_1_s : std_logic_vector(31 downto 0);
signal tmp_2_s : std_logic_vector(31 downto 0);
signal tmp_3_s : std_logic_vector(31 downto 0);

signal m_1_tmp_0_s : std_logic_vector(31 downto 0);
signal m_1_tmp_1_s : std_logic_vector(31 downto 0);
signal m_1_tmp_2_s : std_logic_vector(31 downto 0);
signal m_1_tmp_3_s : std_logic_vector(31 downto 0);

signal m_2_tmp_0_s : std_logic_vector(31 downto 0);
signal m_2_tmp_1_s : std_logic_vector(31 downto 0);
signal m_2_tmp_2_s : std_logic_vector(31 downto 0);
signal m_2_tmp_3_s : std_logic_vector(31 downto 0);
```

Hardware implementation of the θ function

```
-- temp = a[0]^a[2];
m_1_tmp_0_s <= a_0_in xor a_2_in;
-- temp>>>8
m_1_tmp_1_s <= m_1_tmp_0_s(23 downto 0) & m_1_tmp_0_s(31 downto 24);
-- temp<<<8;
m_1_tmp_2_s <= m_1_tmp_0_s(7 downto 0) & m_1_tmp_0_s(31 downto 8);
 -- temp ^= temp>>>8 ^ temp<<<8;
m_1_tmp_3_s <= m_1_tmp_0_s xor m_1_tmp_1_s xor m_1_tmp_2_s;

-- a[1] ^= temp
a_1_0_s <= a_1_in xor m_1_tmp_3_s;
-- a[3] ^= temp
a_3_0_s <= a_3_in xor m_1_tmp_3_s;

-- a xor k

tmp_0_s <= a_0_in xor k_0_in;
tmp_1_s <= a_1_0_s xor k_1_in;
tmp_2_s <= a_2_in xor k_2_in;
tmp_3_s <= a_3_0_s xor k_3_in;
```

Hardware implementation of the θ function

```
-- temp = a[1]^a[3]
m_2_tmp_0_s <= tmp_1_s xor tmp_3_s;
-- temp>>>8
m_2_tmp_1_s <= m_2_tmp_0_s(23 downto 0) & m_2_tmp_0_s(31 downto 24);
-- temp<<<8;
m_2_tmp_2_s <= m_2_tmp_0_s(7 downto 0) & m_2_tmp_0_s(31 downto 8);
-- temp ^= temp>>>8 ^ temp<<<8;
m_2_tmp_3_s <= m_2_tmp_0_s xor m_2_tmp_1_s xor m_2_tmp_2_s;

-- a[0] ^= temp;
a_0_out <= tmp_0_s xor m_2_tmp_3_s;
-- a[2] ^= temp;
a_2_out <= tmp_2_s xor m_2_tmp_3_s;

a_1_out <= tmp_1_s;
a_3_out <= tmp_3_s;
```

Shift operations

The function Π_2 is the inverse of Π_1 so they can be used during encryption and decryption:

- $\Pi_1 : a[1] <<<= 1; a[2] <<<= 5; a[3] <<<= 2;$
- $\Pi_2 : a[1] >>>= 1; a[2] >>>= 5; a[3] >>>= 2;$

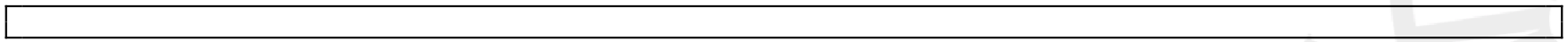
Hardware implementation of the Π_1 function

- $\Pi_1 : a[1] <<<= 1; a[2] <<<= 5; a[3] <<<= 2;$

```
a_1_out <= a_1_in(30 downto 0) & a_1_in(31);
a_2_out <= a_2_in(26 downto 0) & a_2_in(31 downto 27);
a_3_out <= a_3_in(29 downto 0) & a_3_in(31 downto 30);
```

Example 3: Hardware implementation of the Π_2 function

- $\Pi_2 : a[1] >>>= 1; a[2] >>>= 5; a[3] >>>= 2;$



Example 3: Hardware implementation of the Π_2 function

- $\Pi_2 : a[1] >>>= 1; a[2] >>>= 5; a[3] >>>= 2;$

```
a_1_out <= a_1_in(0) & a_1_in(31 downto 1);
a_2_out <= a_2_in(4 downto 0) & a_2_in(31 downto 5);
a_3_out <= a_3_in(1 downto 0) & a_3_in(31 downto 2);
```

We must note how easy is to perform cycle shifts by rearranging the positions of the array.

Round constants

Similar to AES, where RC_0 is equal to 0x80.

```
if (RC[i] & 0x80 == 0) {

RC[i + 1] = RC[i] << 1 ^ 0x1b;

} else {

RC[i + 1] = RC[i] << 1;

}
```

Round constants

They can be generate in reverse order (e.g. during the decryption) as:

```
if (RC[i] & 0x01 == 0) {

RC[i - 1] = RC[i] >> 1 ^ 0x8d;

} else {

RC[i - 1] = RC[i] >> 1;

}
```

Finally, instead of generating them on the fly, they can also be stored in a ROM and read sequentially during encryption and decryption (in inverse order).

Generation of the round constants

- We want on the fly generation.
- One constant generated per cycle (16 rounds, 16 cycles for processing one block).
- The generation of the constants is performed in parallel with the round computation.
- There are two ways of generating the constants
  - encryption
  - decryption

```
entity rc_gen is
port(clk : in std_logic;
             rst : in std_logic;
             enc : in std_logic; -- 0 (enc), 1 (dec)
             rc_out : out std_logic_vector(7 downto 0));
end rc_gen;
```

Generation of the round constants

```
        signal rc_s : std_logic_vector(7 downto 0);
begin

        pr_gen: process(clk, rst, enc)
        begin
                if rising_edge(clk) then
                        if rst = '1' then
                                if enc = '0' then
                                        rc_s <= X"80";
                                else
                                        rc_s <= X"D4";
                                end if;
                        else
                                if enc = '0' then
                                        if ((rc_s and X"80") = X"00") then
                                                rc_s <= rc_s(6 downto 0) &
'0';
                                        else
                                                rc_s <= (rc_s(6 downto 0) &
'0') xor X"1B";
                                        end if;
                                else
                                        if ((rc_s and X"01") = X"00") then
                                                rc_s <= '0' & rc_s(7 downto
1);
                                        else
                                                rc_s <= ('0' & rc_s(7 downto
1)) xor X"8D";
                                        end if;
                                end if;
                        end if;
                end if;
        end process;
```

Composition of the round block

Till now, we have created combinational modes of operation inside a round. Now, we will create the round as a combinational circuit including every operation following the order that appears in the specification of the block cipher:

```
Round(k, a, c1, c2, enc) {
  if (enc)
    a ^= c1;

  theta(k, a);

  if (!enc)
    a ^= c2;

  pi1(a);
  gamma(a);
  pi2(a);
}
```

Round block

For this scheme, it is clear that we need a circuit module with the following inputs:

```
entity round_f is
        port(clk      : in std_logic;
             enc : in std_logic;
                  rc_in    : in std_logic_vector(31 downto 0);
                  a_0_in   : in std_logic_vector(31 downto 0);
                  a_1_in   : in std_logic_vector(31 downto 0);
                  a_2_in   : in std_logic_vector(31 downto 0);
                  a_3_in   : in std_logic_vector(31 downto 0);
                  k_0_in   : in std_logic_vector(31 downto 0);
                  k_1_in   : in std_logic_vector(31 downto 0);
                  k_2_in   : in std_logic_vector(31 downto 0);
                  k_3_in   : in std_logic_vector(31 downto 0);
                  a_0_out  : out std_logic_vector(31 downto 0);
                  a_1_out  : out std_logic_vector(31 downto 0);
                  a_2_out  : out std_logic_vector(31 downto 0);
                  a_3_out  : out std_logic_vector(31 downto 0));
end round_f;
```

Round block

Finally, we import every component as before e.g.

```
component pi_1 is
port(a_1_in     : in std_logic_vector(31 downto 0);
          a_2_in        : in std_logic_vector(31 downto 0);
          a_3_in        : in std_logic_vector(31 downto 0);
          a_1_out       : out std_logic_vector(31 downto 0);
          a_2_out       : out std_logic_vector(31 downto 0);
          a_3_out       : out std_logic_vector(31 downto 0));
end component;
```

We must note that the same round function is repeated during the NOEKEON transformation (encryption) and its decryption, therefore, it will be reused.

The full architecture

- We are creating a loop architecture.
- That means that we need: a register that will store the last ciphertext of the round and a mux that will choose between the plaintext and the last ciphertext of the round.
- At the same time, are generating the round constants (all 1 cycle).

The full architecture

We mount first the register that will store the plaintext and the round ciphertext round by round, note that this is activated by a multiplexer according to the reset signal, that is, when rst is high we can store our plaintext there and if not, the input will be the output of the round block:

```
pr_noe: process(clk, rst, enc)
begin
        if rising_edge(clk) then
                if rst = '1' then
                        a_0_in_s <= a_0_in;
                        a_1_in_s <= a_1_in;
                        a_2_in_s <= a_2_in;
                        a_3_in_s <= a_3_in;
                else
                        a_0_in_s <= a_0_out_s;
                        a_1_in_s <= a_1_out_s;
                        a_2_in_s <= a_2_out_s;
                        a_3_in_s <= a_3_out_s;
                end if;
        end if;
end process;
```

The full architecture

```
RC_GEN_0 : rc_gen port map (clk, rst, enc, rc_s);
rc_ext_s <= X"000000" & rc_s;

ROUND_F_0 : round_f port map (clk, enc, rc_ext_s,
                              a_0_in_s, a_1_in_s, a_2_in_s, a_3_in_s,
                              k_0_mux_s, k_1_mux_s, k_2_mux_s, k_3_mux_s,
                              a_0_out_s, a_1_out_s, a_2_out_s, a_3_out_s);
```

What we did not see and you can study from the code:

- How decryption is implemented.
- How k' is computed.
- How do we select between k and k' in decryption and encryption.

Q/A

Questions?