Project Report - PRESENT Block Cipher

Group 1

Team members:
Marius-Mihail Gurgu
Alexandre Janin
Tamil Selvi Pandiyan
Matteo Spallanzani
Wei-Heng Ke

Table of Contents

1. Overview

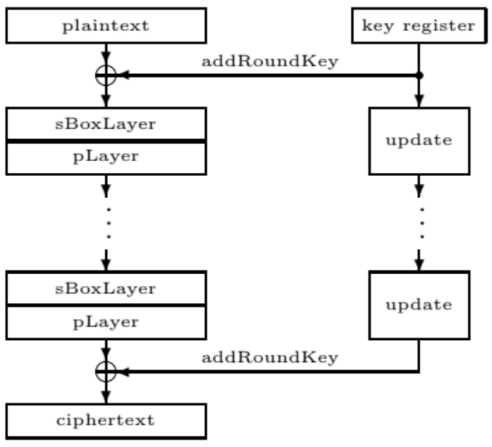
The block size is 64 bits and the key size can be 80 bit or 128 bit for a total of 31 rounds. The non-linear layer is based on a single 4-bit S-box which was designed with hardware optimizations in mind. PRESENT is intended to be used in situations where low-power consumption and high chip efficiency is desired.
In each of the 31 rounds, a XOR operation is applied between the round key (64 bits) and the 64-bit data block. This output enables the actual encryption of the processed 64-bit data block in 2 stages:

1) A non-linear substitution (sBoxLayer) with a substitution table that creates a 4-bit to 4-bit correspondence. The data block of 64 bits can be regarded as 16 words, each with a size of 4 bits.
2) A linear bitwise permutation (pLayer) with the help of the permutation table

If a **80-bit** user-supplied key is used, then the round key consists of the 64 leftmost bits in the 80-bit register of the encryption key.

Present here consists of 31 rounds. The block length is 64 bits and two key lengths of 80 and 128 bits are supported. However, this project implementation uses the version with 80-bit keys. This Version is more than adequate security for the low-security applications typically required in tag-based deployments[1].

Each of the 31 rounds consists of an xor operation to introduce a round key
Ki for 1 ≤ i ≤ 32, where K32 is used for post-whitening, a linear bitwise permutation and a non-linear substitution layer. The non-linear layer uses a single 4-bit S-box S which is applied 16 times in parallel in each round. The cipher is described in pseudo-code in Figure below [2], and each stage is now specified in turn.



Simplified flowchart of the PRESENT encryption [2]

addRoundKey

Given round key Ki = Ki63 . . . Ki0 for 1 ≤ i ≤ 32 and current
state b63 . . . b0, addRoundKey consists of the operation for 0 ≤ j ≤ 63,

$$bj \rightarrow bj \oplus \kappa ij$$

sBoxLayer & sBoxLayerDec

The variable “state” is received as input from the gen_round_keys module and converted to a Sbox code in hexadecimal notation as given by the following table. The S-box used in Present is a 4-bit to 4-bit S-box ‘S’. Therefore, the sbox code is converted to 64 bit output by appending the code 16 times.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[x]	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

For sBoxLayer the current state, sbox(state)63 . . . sbox(state)0 is considered as sixteen 4-bit words: layer15 . . . layer0 where layer(i) = state4∗i+3||state4∗i+2||state4∗i+1||state4∗i for 0 ≤ i ≤ 15 and the output nibble sBoxLayer_out[layer(i)] provides the updated state values in the obvious way.

The SBoxlayerDec is the decryption is just the reverse logic of the SBoxLayer encryption.
The received S[x] hexadecimal notation is converted to the original x. The value x becomes the output of this module.
The encrypted 64-bit code from sBoxLayer(out) is later stored in a register and then sent to the pLayer for further processing.

pLayer & pLayerDec

The module “pLayerDec” is the inverse module of “pLayer”. To enable it, the operation involved is the same as “pLayer”: the bit reallocation. The input is the output of the encryption module, and the reallocation of the bits is applied through the inverted matrix. To implement this, the code is the same as “pLayer”, but the position of the bits changed. The order of the bits is displayed below in the table below.

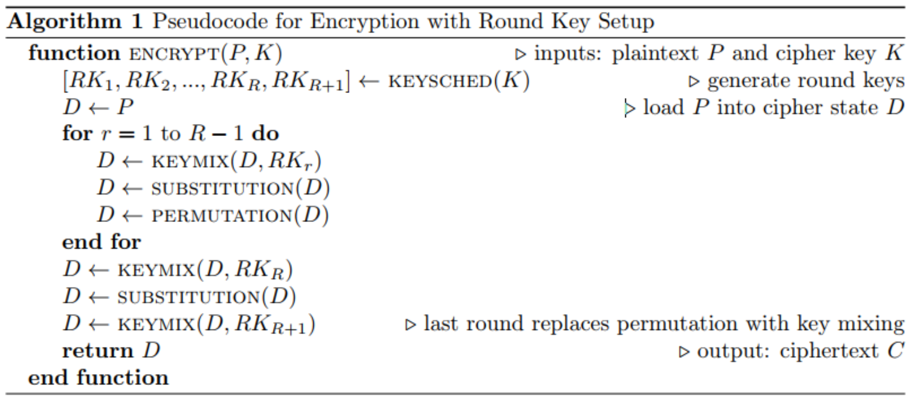
P	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60

P	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
i	1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61

P	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
i	2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62

P	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
i	3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63

Pseudocode Algorithm



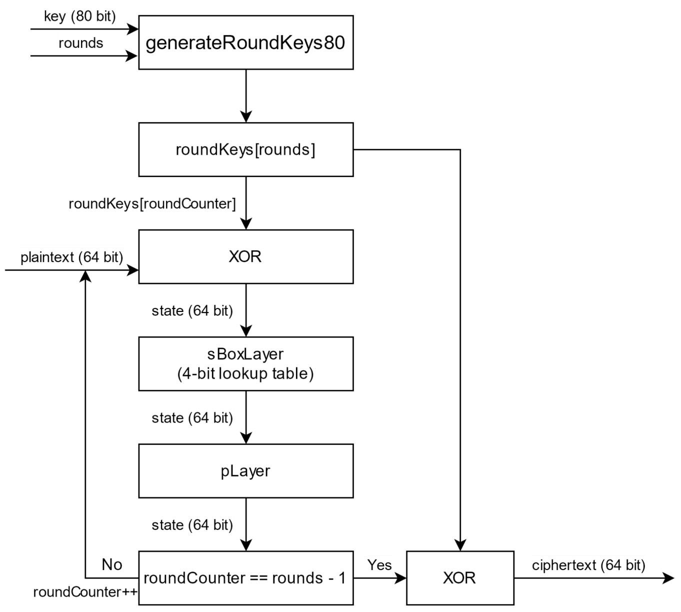
Algorithm 1 Pseudocode for Encryption with Round Key Setup

```
function ENCRYPT($P, K$)                          ▷ inputs: plaintext $P$ and cipher key $K$
    $[RK_1, RK_2, ..., RK_R, RK_{R+1}] \leftarrow$ KEYSCHED($K$)        ▷ generate round keys
    $D \leftarrow P$                                ▷ load $P$ into cipher state $D$
    for $r = 1$ to $R - 1$ do
        $D \leftarrow$ KEYMIX($D, RK_r$)
        $D \leftarrow$ SUBSTITUTION($D$)
        $D \leftarrow$ PERMUTATION($D$)
    end for
    $D \leftarrow$ KEYMIX($D, RK_R$)
    $D \leftarrow$ SUBSTITUTION($D$)
    $D \leftarrow$ KEYMIX($D, RK_{R+1}$)        ▷ last round replaces permutation with key mixing
    return $D$                                      ▷ output: ciphertext $C$
end function
```

(

Encryption algorithm [3]

High-level implementation and integrated Hardware Scheme

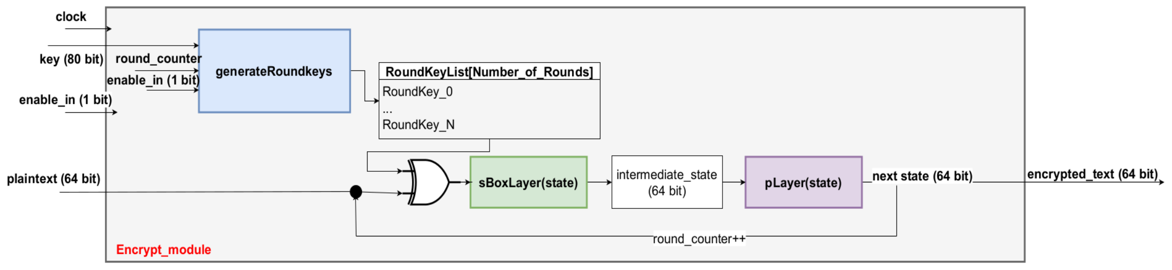


Python implementation flowchart

In order to test our implementation in Vivado, we generated 5 different test cases consisting of a plaintext and a user provided 80-bit key. After running the test cases in python, we compared the output with the Vivado implementation.

Encryption Scheme

We added a register called “intermediate_state” between sBoxLayer and pLayer to ensure the combinational modules have enough time per clock cycle to finish their processing.

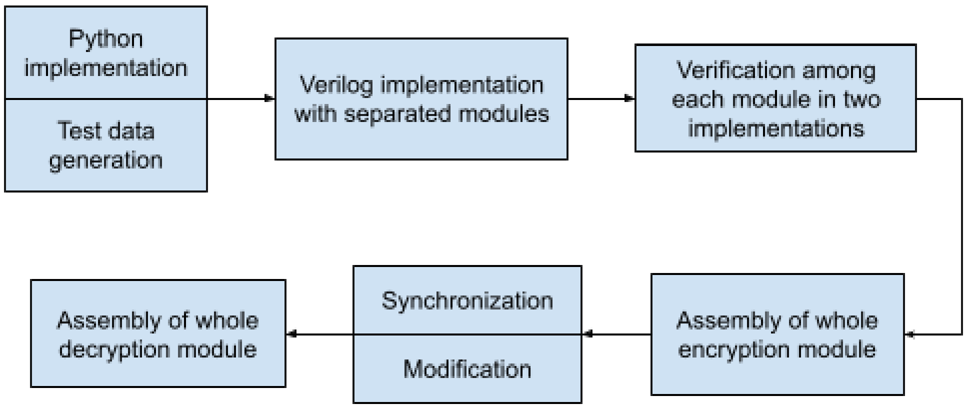


Encryption Hardware Scheme

The decryption scheme is very similar to the decryption one, the only difference being that pLayerDec and sBoxLayerDec modules use different look-up tables. Also, the XOR gate is followed by pLayerDec and then by sBoxLayerDec, basically the reverted order of encryption, in order to compute the deciphered text.

Process

In the stage of synchronization and modification, we took a bit more time because we lacked some intermediate phases like input ROM and output RAM. After assembling all missing parts, we managed to run the last implementation stage in Vivado.


Python implementation
Test data generation
Verilog implementation with separated modules
Verification among each module in two implementations
Assembly of whole encryption module
Synchronization
Modification
Assembly of whole decryption module

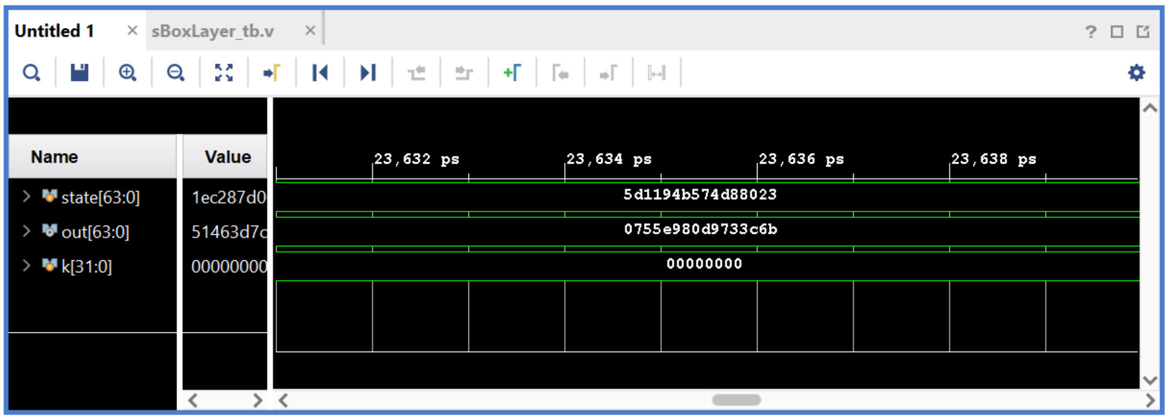
Development process of the project

Results from Vivado

The figures below are some examples of the simulation results done in Vivado in order to validate our results.

sBoxLayer & sBoxLayerDec Output

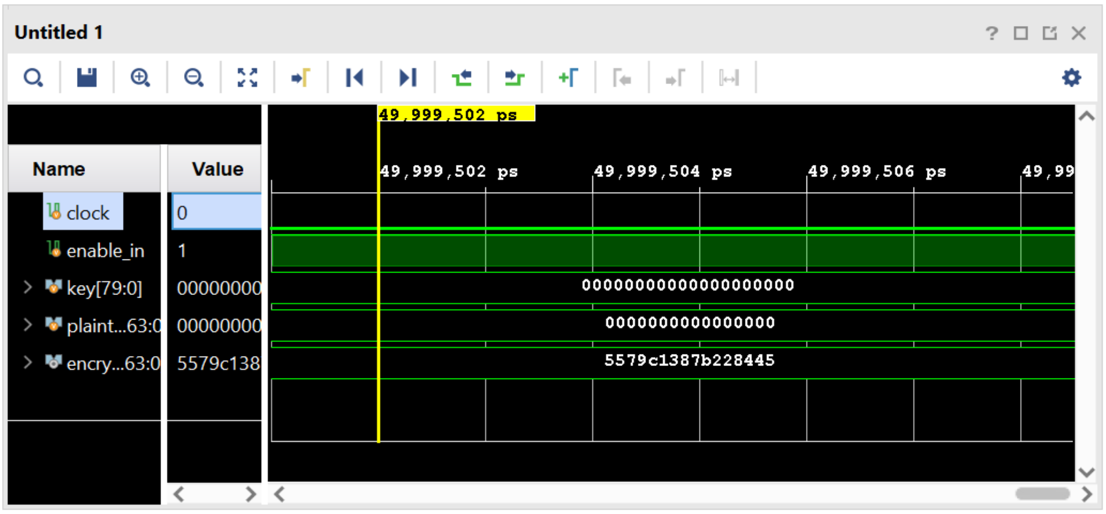
The figure below shows a snippet of the encrypted sBoxLayer results from vivado, where the input variable *state* (5d1194b574d88023) from gen_round_keys module is encrypted and written to the sBoxLayer output variable *out* (0755e980d9733c6b) using the hexadecimal notation.



Name	Value	
state[63:0]	1ec287d0	5d1194b574d88023
out[63:0]	51463d7c	0755e980d9733c6b
k[31:0]	00000000	00000000

Successful sBoxLayer result

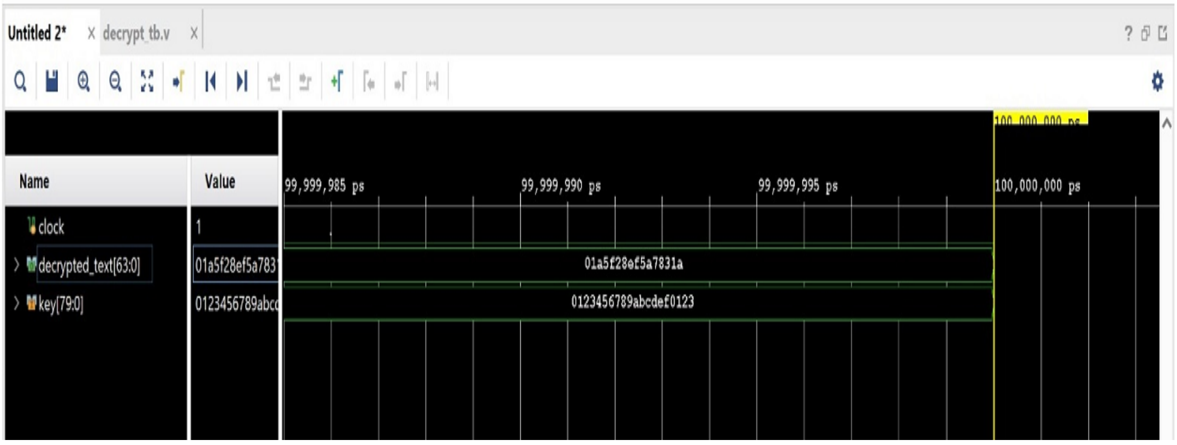
Encryption Output



```
clock          0
enable_in      1
key[79:0]      00000000    00000000000000000000
plaint...63:0  00000000    0000000000000000
encry...63:0   5579c138    5579c1387b228445
```

Fig. Successful encrypt example

Decryption Output



```
clock                1
decrypted_text[63:0] 01a5f28ef5a783   01a5f28ef5a7831a
key[79:0]            0123456789abcd   0123456789abcdef0123
```

Fig. Successful decrypt example

The results show that the plain texts are encrypted into a cipher text successfully by the module, which is met with the results from python. Similarly, the input for the description module which is similar to the output of the encrypt section is encoded into a plain text.

Encountered Challenges

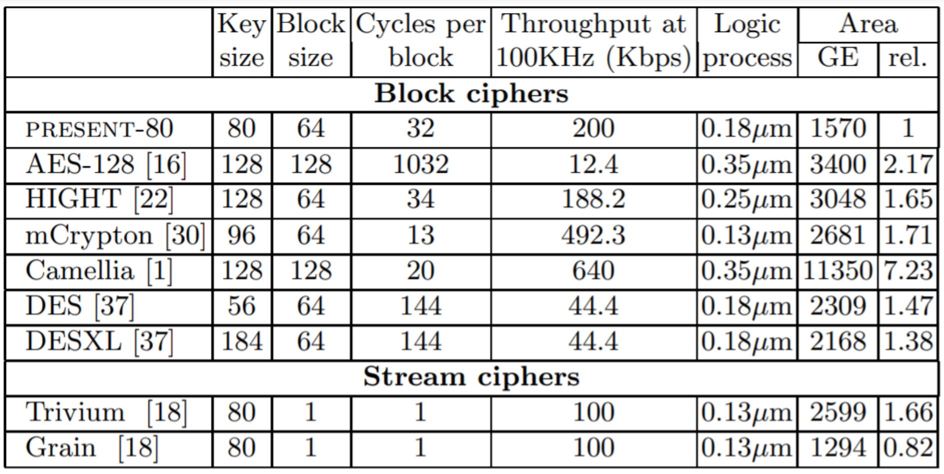
The main challenge we coped with was how to synchronize and integrate the different modules we designed. In the end, we created a design with a “top” module that consists of 2 memory blocks (for input and output) and the encryption (or decryption) module.
Other solved challenges were:

- Breaking points solution: a Vivado bug where setting up multiple breakpoints makes the simulation fail.
- Combinatorial loop solution (i.e. bad idea to simply translate python code into verilog and expect it to work :)); we managed to counter this problem by updating

the generateRoundKeys module to use a register that would **not** create a link between the output and the input of the module.
- Minor bugs: in the design phase of each module it was common to find and solve little implementation bugs that would cause logical errors and provide a wrong output.

Power consumption concerns

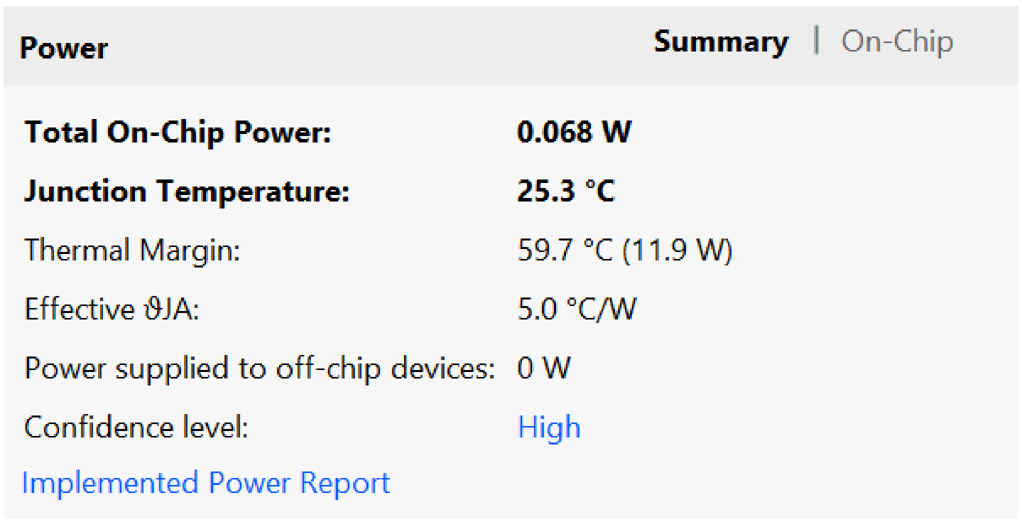
We first estimate the power after synthesis of various ciphers in comparison to our lightweight PRESENT-80 cipher implementation. The estimated table below shows that our 80 bit block cipher has a higher throughput of 200Kbps amongst others.



	Key size	Block size	Cycles per block	Throughput at 100KHz (Kbps)	Logic process	Area GE	Area rel.
Block ciphers							
PRESENT-80	80	64	32	200	0.18μm	1570	1
AES-128 [16]	128	128	1032	12.4	0.35μm	3400	2.17
HIGHT [22]	128	64	34	188.2	0.25μm	3048	1.65
mCrypton [30]	96	64	13	492.3	0.13μm	2681	1.71
Camellia [1]	128	128	20	640	0.35μm	11350	7.23
DES [37]	56	64	144	44.4	0.18μm	2309	1.47
DESXL [37]	184	64	144	44.4	0.18μm	2168	1.38
Stream ciphers							
Trivium [18]	80	1	1	100	0.13μm	2599	1.66
Grain [18]	80	1	1	100	0.13μm	1294	0.82

Key performance indicators of different cipher algorithms

After successfully completing the implementation stage, Vivado estimation of the power consumption was available. It was 2 orders of magnitude lower than the initial estimation (done based upon the synthesis stage) of more than 5 W of used power. This means that, in this way the device can work with 2 AA batteries for around 9 months with the total on-chip power of 0.068W.



Total On-Chip Power:	**0.068 W**
Junction Temperature:	**25.3 °C**
Thermal Margin:	59.7 °C (11.9 W)
Effective ϑJA:	5.0 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	High

Fig. Vivado implementation stage estimate of power usage

References

1) https://en.wikipedia.org/wiki/PRESENT
2) PRESENT: An Ultra-Lightweight Block Ciphe - A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann,M.J.B. Robshaw, Y. Seurin and C. Vikkelsoe (https://www.emsec.ruhr-uni-bochum.de/media/crypto/attachments/files/2010/04/present_ches2007.pdf)
3) A Tutorial on the Implementation of Block Ciphers: Software and Hardware Applications (https://eprint.iacr.org/2020/1545.pdf)