**Lab 1 - SIMON encryption/decryption**

Joseph Burke  
Idan G. Kanter  
Seth Rausch

ECE 459 - Secure & Trustworthy Computer Hardware Design

Spring 2017

Dr. Garrett S. Rose

January 28 2017

Instructor Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Abstract/Introduction**

The field of security in software and hardware was always an important subject. A few encryption algorithms were introduced including the older Data Encryption Standard (DES), and the newer Advanced Encryption Standard (AES). AES is a very strong non-Feistel encryption algorithm; however, it is also considered heavyweight; AES requires heavy usage of the system resources as memory, area, and power. Due to the rising need of a more lightweight encryption algorithm for FPGAs, ASICs, Internet of Things (IOT), embedded systems, and more, a team at the National Security Agency (NSA) introduced the SIMON block cipher algorithm which is optimized for hardware. SIMON is a Feistel encryption algorithm that uses less system resources than the AES, which makes it suitable for small systems that were introduced above.

The purpose of the lab is to learn how SIMON works and to implement some of its functions. After reading two papers about the implementation of SIMON, the round and the key expansion function are implemented. After the algorithm is completed, some test benches are run to simulate the encryption. Next, the program is synthesized, implemented, and is generated to a bit file for testing on a Nexys-4 FPGA board. Finally, the implementation is programmed to the board and the program encrypts two image files. In addition, the FPGA is used to test the decryption of the images since SIMON is a Feistel algorithm. During the whole process, interesting features, observations, and shortcomings of SIMON are recorded.

Simon is comprised of several round functions that do various different operations on the plaintext for encryption. These operations are comprised of bitwise XOR, bitwise AND, and left circular shift. When the plaintext is put into the round function the block data is split into 2 halves and at the end of each round the left and right halves of the data are swapped.

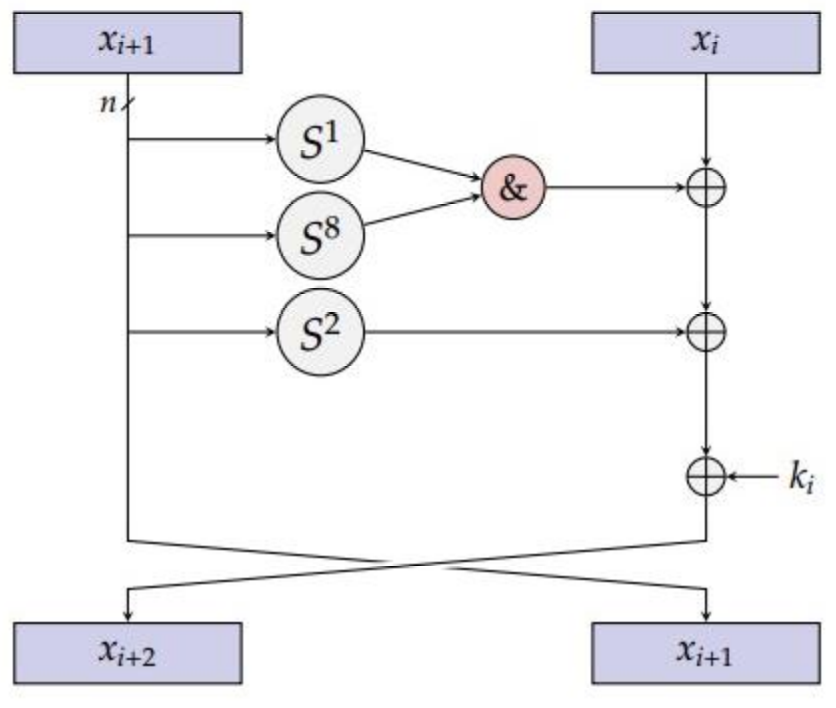


Figure. 1.

Figure 1 shows the SIMON round function which is used for encryption and its inverse for decryption. We define the SIMON round function used for encryption as:



and the SIMON round function used for decryption as:



Additionally, the SIMON key schedule allows us to use key expansion which generates all the round keys from the master keys. This is done in each round i, by combining the current cached previous n round keys with the constant c and a 1-bit round constant. Key expansion utilizes bitwise XOR (𝑥 ⊕ y) and right bitwise rotation (ROR). The SIMON four word key expansion is shown in figure 2. The key expansion function can be expressed as:



such that:



After each expansion the cached keys are rotated by the right bitwise rotation operation which means the the first element is discarded and the last is replaced with the generated key.

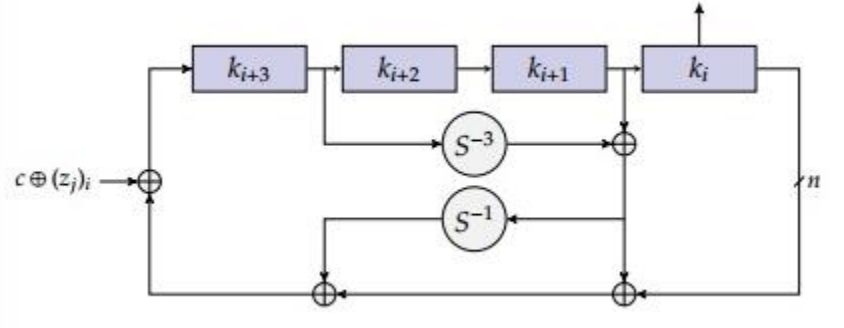


Figure 2.

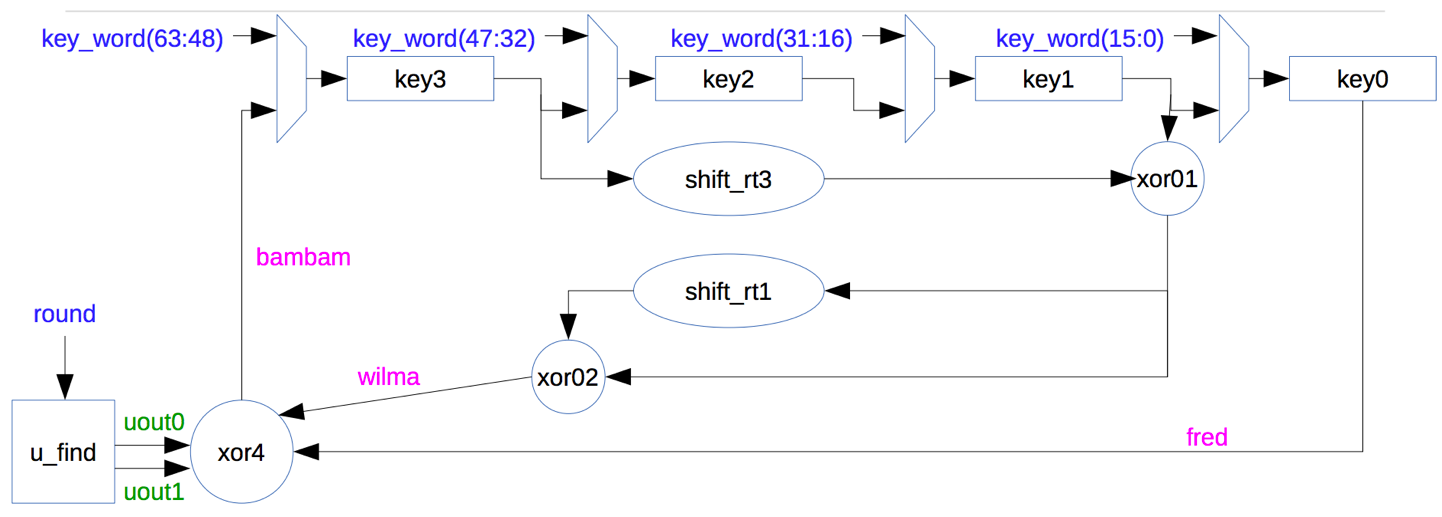
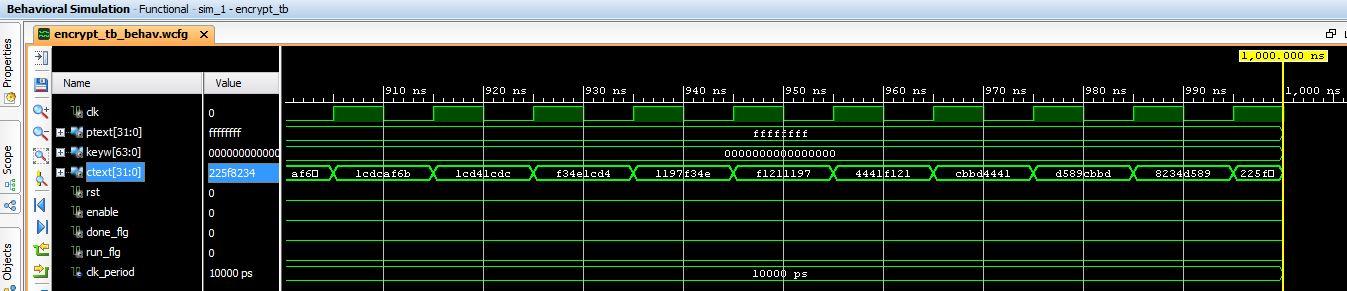
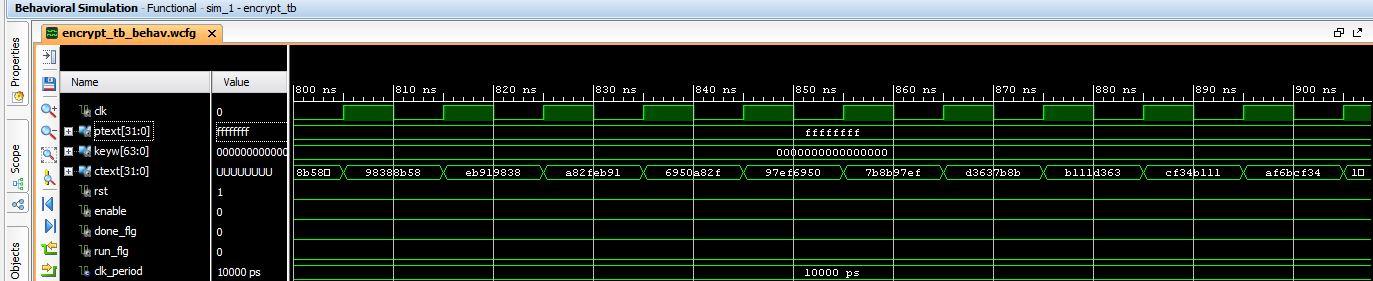
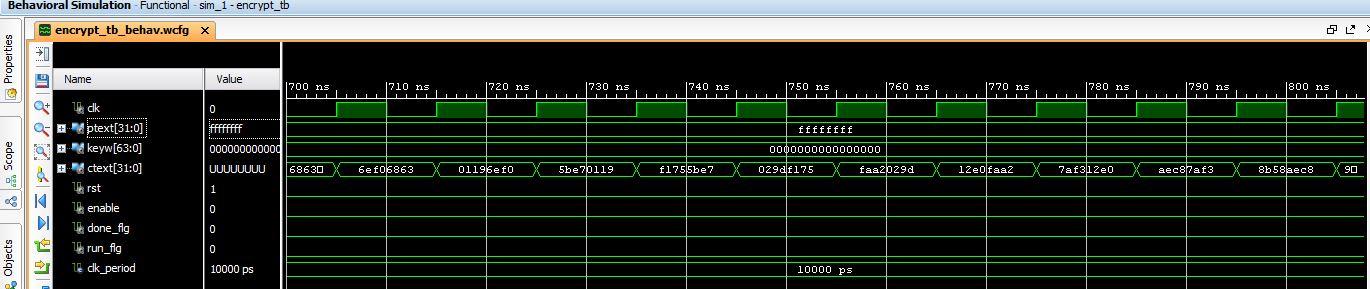
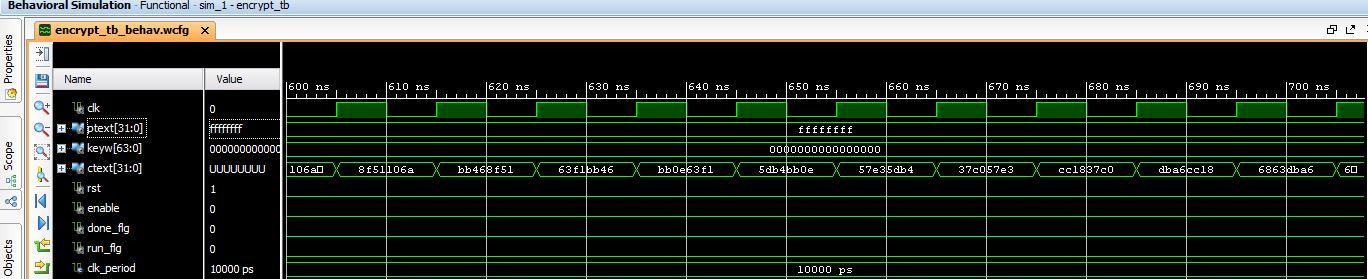
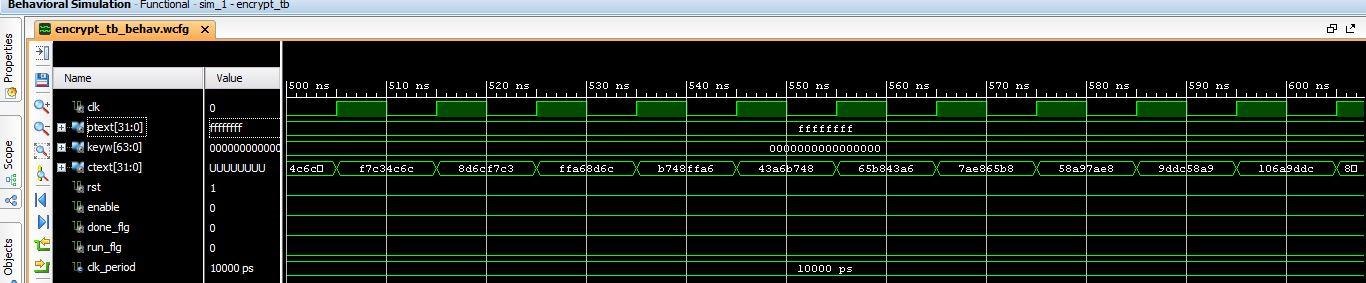
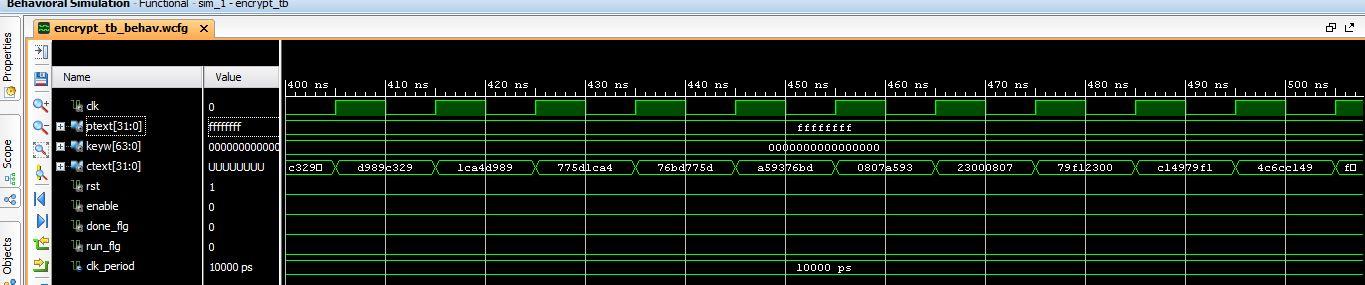
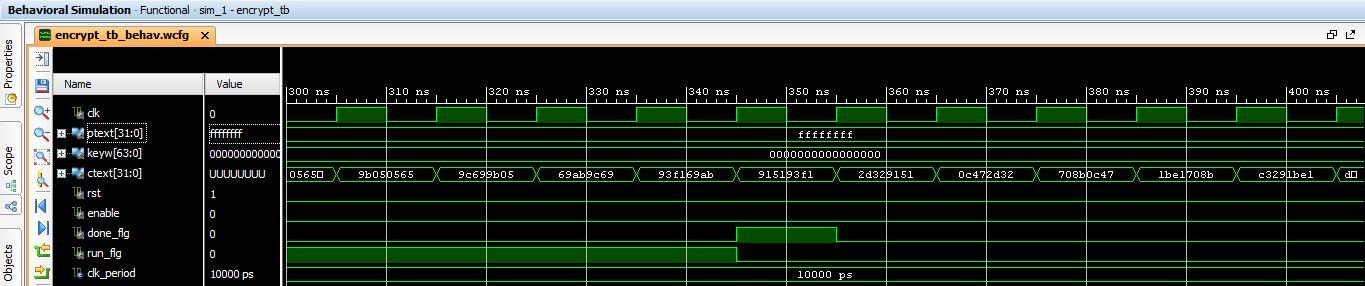
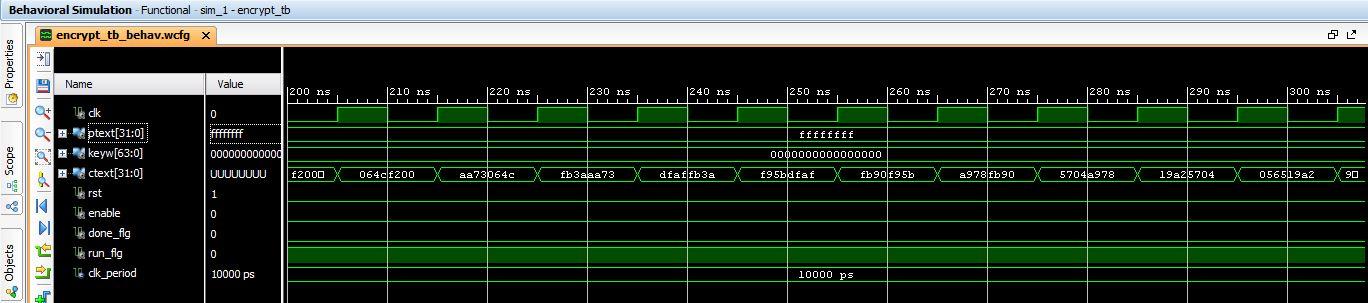
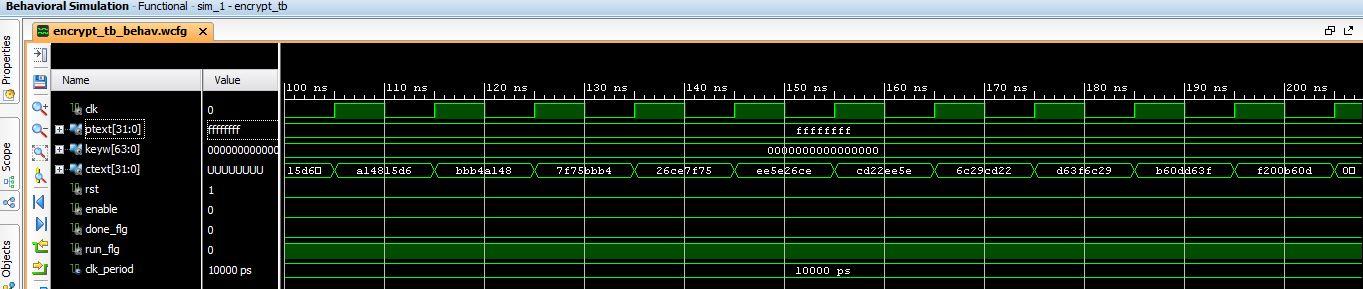
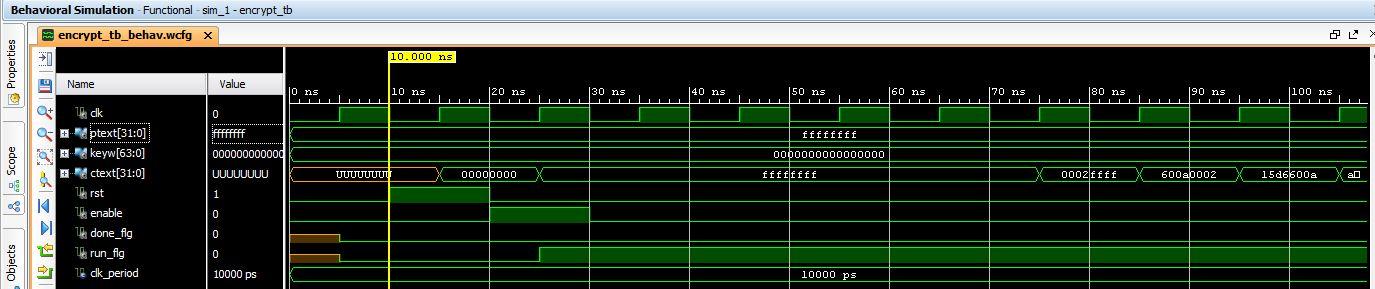


Figure 3.

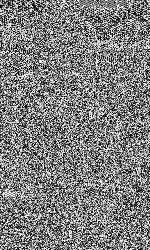
In figure 3 we see that it is very similar to figure 2. This is the key expansion with our vhdl components. Each arrow is a different signal in vhdl. Finally, the signals are port mapped to tell the fpga where the signals are traveling to and from each component.

**Results**

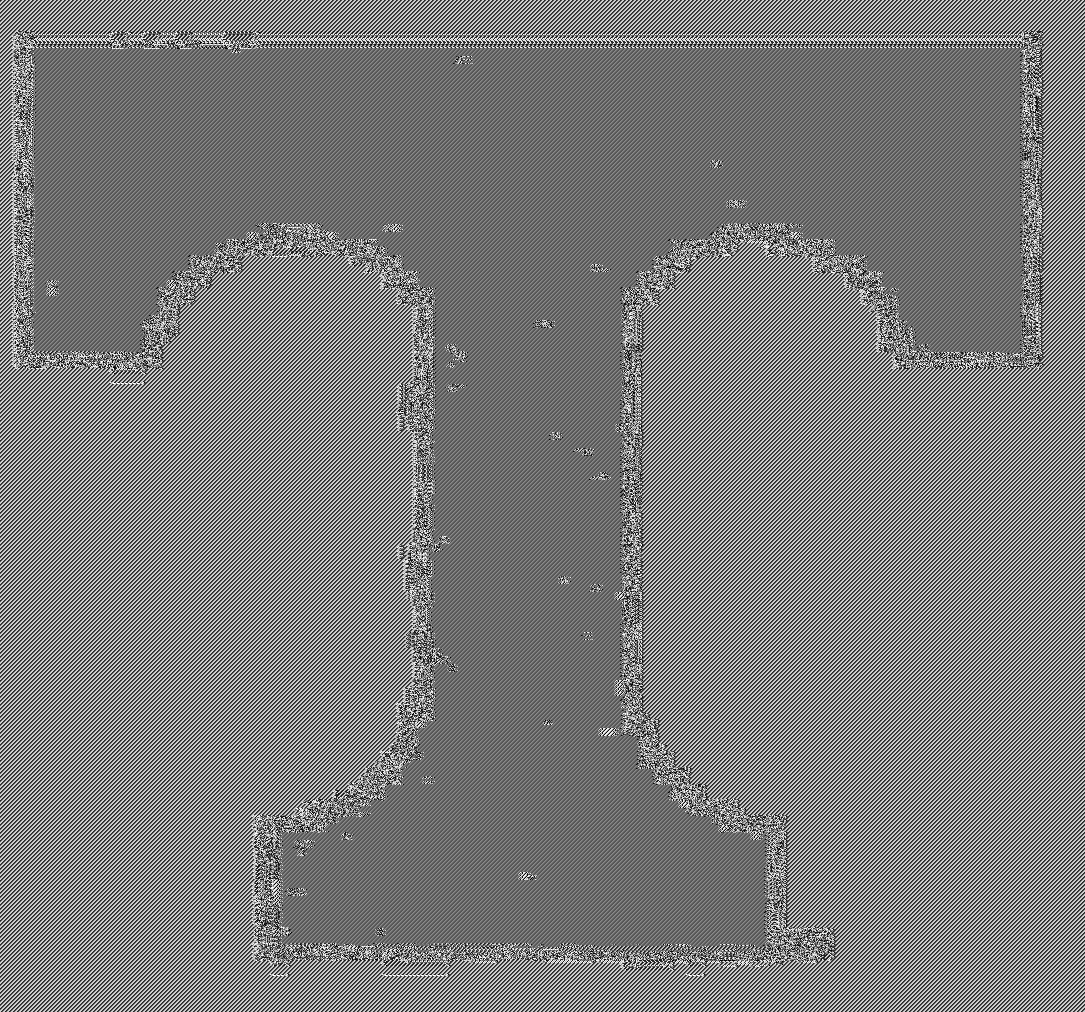
In the behavioral simulation we showed that the 64 bit passed plaintext was successfully encrypted by the key. As you can see the ctext line is the cipher text which is alternating at each round.



**Figure 1 - Payton\_encrypt**



**Figure 2 - PowetT\_encrypt**



**Observations**

On inspection of the encrypted PGM files depicted in figures 1 and 2 simon you can see that the SIMON cipher works well in some cases and not so well in others. In figure 2 it is evident that the image was not very well encrypted, as the pattern is still visible. This shows some of the limitations and place where it could be improved. We found that SIMON64/128 has many tradeoffs that come with the design as well as the issues we encountered during the implementation. The SIMON family of block ciphers has proved to be a great candidate for applications that require flexible, lightweight cryptography.

**Appendix**

**Round\_Cipher.vhd**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.all;

use IEEE.STD\_LOGIC\_ARITH.all;

use IEEE.STD\_LOGIC\_UNSIGNED.all;

entity round\_cipher is

port (

blockcipher : in std\_logic\_vector(31 downto 0);

key\_word : in std\_logic\_vector(15 downto 0);

cipher\_text : out std\_logic\_vector(31 downto 0)

);

end round\_cipher;

architecture your\_code of round\_cipher is

-- You can use these components if needed...

-- Feel free to develop your own logic as well...

component shift\_left1 is

port (

input1 : in std\_logic\_vector(15 downto 0);

output : out std\_logic\_vector(15 downto 0)

);

end component shift\_left1;

component shift\_left2 is

port (

input1 : in std\_logic\_vector(15 downto 0);

output : out std\_logic\_vector(15 downto 0)

);

end component shift\_left2;

component shift\_left8 is

port (

input1 : in std\_logic\_vector(15 downto 0);

output : out std\_logic\_vector(15 downto 0)

);

end component shift\_left8;

component xor16bit is

port (

input1 : in std\_logic\_vector(15 downto 0);

input2 : in std\_logic\_vector(15 downto 0);

output : out std\_logic\_vector(15 downto 0)

);

end component xor16bit;

component and16bit is

port (

A : in std\_logic\_vector(15 downto 0);

B : in std\_logic\_vector(15 downto 0);

Y : out std\_logic\_vector(15 downto 0)

);

end component and16bit;

component ctext\_reg is

port (

clk : in std\_logic;

reset : in std\_logic;

round : in std\_logic\_vector(7 downto 0);

input1\_16bits : in std\_logic\_vector(15 downto 0);

input2\_16bits : in std\_logic\_vector(15 downto 0);

flop32 : out std\_logic\_vector(31 downto 0);

text\_flag : out std\_logic

);

end component ctext\_reg;

-- Will need to declare intermediary signals

signal left\_shift\_1\_block : std\_logic\_vector (15 downto 0); -- upper block after left shift by 1 bit

signal left\_shift\_2\_block : std\_logic\_vector (15 downto 0); -- upper block after left shift by 2 bit

signal left\_shift\_8\_block : std\_logic\_vector (15 downto 0); -- upper block after left shift by 8 bit

signal and\_1\_8\_blocks : std\_logic\_vector (15 downto 0); -- AND of the 1 and 8 left shifts

signal and\_XOR\_lower : std\_logic\_vector (15 downto 0); -- XOR of and result with lower\_block

signal sl2\_XOR\_xor1 : std\_logic\_vector (15 downto 0); -- XOR of the left shift by 2 block with the upper xor

signal lower\_block : std\_logic\_vector (15 downto 0); -- lower block of blockcipher

signal upper\_block : std\_logic\_vector (15 downto 0); -- upper block of blockcipher

signal xor2\_XOR\_key : std\_logic\_vector (15 downto 0); -- xor of middle xor and the key

begin

sl1 : shift\_left1 port map (blockcipher(31 downto 16), left\_shift\_1\_block); -- upper block shift left by 1

sl8 : shift\_left8 port map (blockcipher(31 downto 16), left\_shift\_8\_block); -- upprr block shift left by 8

and\_1\_8 : and16bit port map (left\_shift\_1\_block, left\_shift\_8\_block, and\_1\_8\_blocks); -- AND the 1-bit left shift and the 8-bit left shift blocks

sl2 : shift\_left2 port map (blockcipher(31 downto 16), left\_shift\_2\_block); -- upper block shift left by 2

xor1 : xor16bit port map (and\_1\_8\_blocks, blockcipher(15 downto 0), and\_XOR\_lower); -- XOR the result of the AND operation with the lower block of the blockcipher

xor2 : xor16bit port map (left\_shift\_2\_block, and\_XOR\_lower, sl2\_XOR\_xor1); -- XOR the result of upper xor with the 2-bit left shift block

xor3 : xor16bit port map (sl2\_XOR\_xor1, key\_word, xor2\_XOR\_key); -- XOR the result of middle xor with the key

lower\_block <= blockcipher(31 downto 16); -- new lower block is the upper block

upper\_block <= xor2\_XOR\_key; -- new upper block is the result of bottom xor

cipher\_text <= upper\_block & lower\_block; -- attach the two blocks together to form the cipher text - output

end your\_code;

KEY\_EXPANSION.vhd

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.std\_logic\_signed.all;

entity key\_expansion is

port (

round : IN std\_logic\_vector(7 downto 0);

sel : IN std\_logic;

key\_word : IN std\_logic\_vector(63 downto 0);

clk : IN std\_logic;

reset : IN std\_logic;

key\_expansion : OUT std\_logic\_vector(15 downto 0)

);

end key\_expansion;

architecture your\_code of key\_expansion is

-- These components are available to use if desired...

-- Feel free to implement your own...

component reg16 is

port (

clk : in std\_logic;

reset : in std\_logic;

input\_16bits : in std\_logic\_vector(15 downto 0);

flop16 : out std\_logic\_vector(15 downto 0)

);

end component reg16;

component mux2to1 is

port (

SEL : in std\_logic;

A : in std\_logic\_vector (15 downto 0);

B : in std\_logic\_vector (15 downto 0);

X : out std\_logic\_vector (15 downto 0)

);

end component mux2to1;

component shift\_right1 is

port (

input1 : in std\_logic\_vector(15 downto 0);

output : out std\_logic\_vector(15 downto 0)

);

end component shift\_right1;

component shift\_right3 is

port (

input1 : in std\_logic\_vector(15 downto 0);

output : out std\_logic\_vector(15 downto 0)

);

end component shift\_right3;

component xor16bit is

port (

input1 : in std\_logic\_vector(15 downto 0);

input2 : in std\_logic\_vector(15 downto 0);

output : out std\_logic\_vector(15 downto 0)

);

end component xor16bit;

component u\_bit is

port (

round : in std\_logic\_vector(7 downto 0);

u\_out : out std\_logic\_vector(15 downto 0);

u\_int : out std\_logic\_vector(15 downto 0)

);

end component u\_bit;

component xor16bit\_triple is

port (

input1 : in std\_logic\_vector(15 downto 0);

input2 : in std\_logic\_vector(15 downto 0);

input3 : in std\_logic\_vector(15 downto 0);

input4 : in std\_logic\_vector(15 downto 0);

output : out std\_logic\_vector(15 downto 0)

);

end component xor16bit\_triple;

-- Will need to declare intermediary signals here

signal uout0 : std\_logic\_vector(15 downto 0); -- 'C'

signal uout1 : std\_logic\_vector(15 downto 0); -- 'Z'

signal XOR4\_out : std\_logic\_vector(15 downto 0); -- output of XOR 'C', 'Z', key0, XOR2\_out - "bambam"

signal mux1\_out : std\_logic\_vector(15 downto 0); -- output of left mux1

signal mux2\_out : std\_logic\_vector(15 downto 0); -- output of 2nd from the left mux2

signal mux3\_out : std\_logic\_vector(15 downto 0); -- output of 3rd from the left mux3

signal mux4\_out : std\_logic\_vector(15 downto 0); -- output of right mux4

signal key3 : std\_logic\_vector(15 downto 0); -- output of register key3

signal key2 : std\_logic\_vector(15 downto 0); -- output of register key2

signal key1 : std\_logic\_vector(15 downto 0); -- output of register key1

signal key0 : std\_logic\_vector(15 downto 0); -- output of register key0 - "fred"

signal sr3\_out : std\_logic\_vector(15 downto 0); -- output of the shift\_rt3

signal key1\_XOR\_sr3 : std\_logic\_vector(15 downto 0); -- output of xor01, input are key1 and sr3

signal sr1\_out : std\_logic\_vector(15 downto 0); -- output of the shift\_rt1

signal xor2\_out : std\_logic\_vector(15 downto 0); -- output of xor2\_out - "Wilma"

begin

u\_find : u\_bit port map (round, uout0, uout1); -- round -> 'C' and 'Z'

xor4 : xor16bit\_triple port map (uout0, uout1, key0, xor2\_out, XOR4\_out); -- xor 4 inputs -> input of mux1

mux1 : mux2to1 port map (sel, key\_word(63 downto 48), XOR4\_out, mux1\_out); -- mux xor4\_output and key -> key3

key3\_reg : reg16 port map (clk, reset, mux1\_out, key3); -- register 3

mux2 : mux2to1 port map (sel, key\_word(47 downto 32), key3, mux2\_out); -- mux key3 and key -> key2

key2\_reg : reg16 port map (clk, reset, mux2\_out, key2); -- register 2

sr3 : shift\_right3 port map (key3, sr3\_out); -- key3 shift right by 3 bits

mux3 : mux2to1 port map (sel, key\_word(31 downto 16), key2, mux3\_out); -- mux key2 and key -> key1

key1\_reg : reg16 port map (clk, reset, mux3\_out, key1); -- register 1

mux4 : mux2to1 port map (sel, key\_word(15 downto 0), key1, mux4\_out); -- mux key1 and key -> key0

key0\_reg : reg16 port map (clk, reset, mux4\_out, key0); -- register 0

xor1 : xor16bit port map (key1, sr3\_out, key1\_XOR\_sr3); -- xor key1 and shift\_rt3

sr1 : shift\_right1 port map (key1\_XOR\_sr3, sr1\_out); -- xor01 shift right by 1 bit

xor2 : xor16bit port map (sr1\_out, key1\_XOR\_sr3, xor2\_out); -- xor shift\_rt1 and xor01

key\_expansion <= key0; -- output

end your\_code;