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.

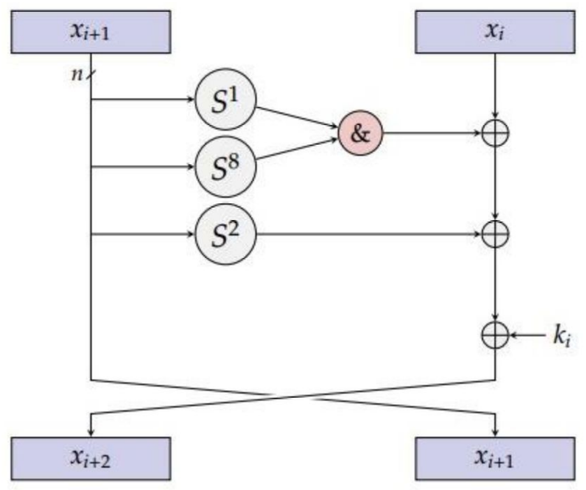


Figure. 1.

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:

$$R(l, r, k) = ((S^1(l) \,\&\, S^8(l)) \oplus S^2(l) \oplus r \oplus k, l)$$

and the SIMON round function used for decryption as:

$$R^{-1}(l, r, k) = (r, (S^1(r) \,\&\, S^8(r)) \oplus S^2(r) \oplus l \oplus k)$$

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 ($x \oplus$ 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:

$$K_i(k, c, z_j) = F(k_{i+3}, k_{i+1}) \oplus S^{-1}(F(k_{i+3}, k_{i+1})) \oplus k_i \oplus c \oplus (z_j)_i$$

such that:

$$F(x, y) = S^{-3}(x) \oplus y$$

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.

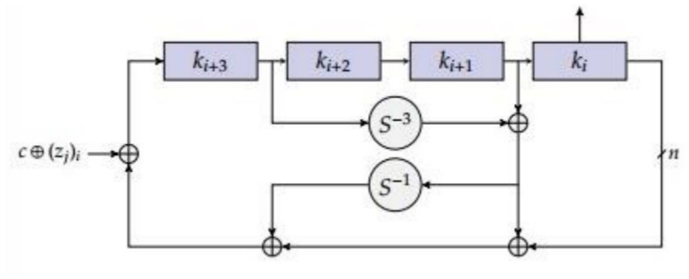


Figure 2.

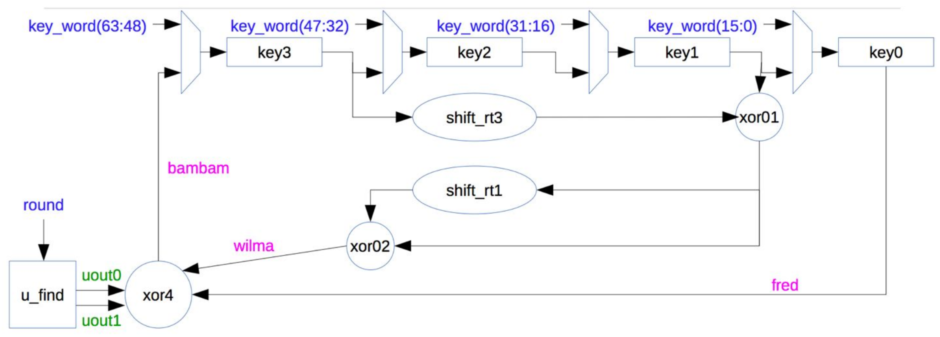
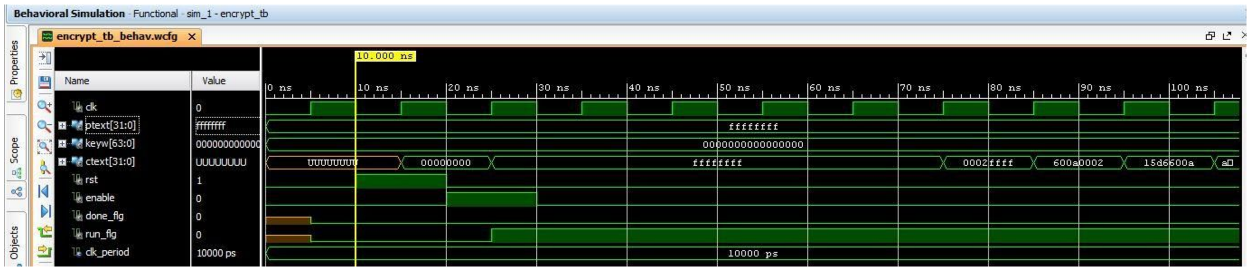


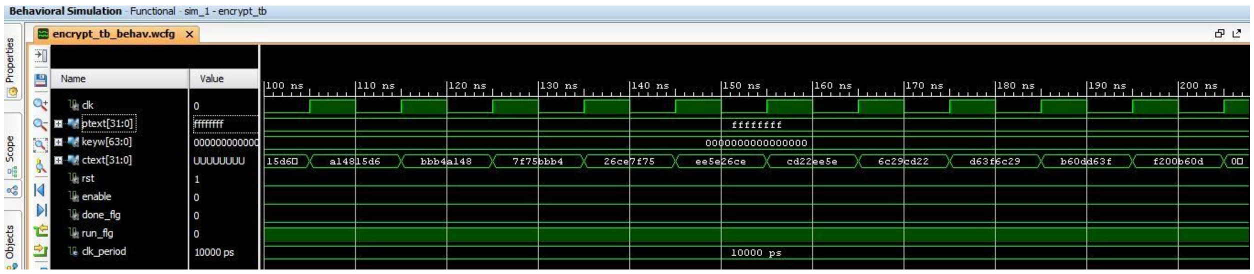
Figure 3.

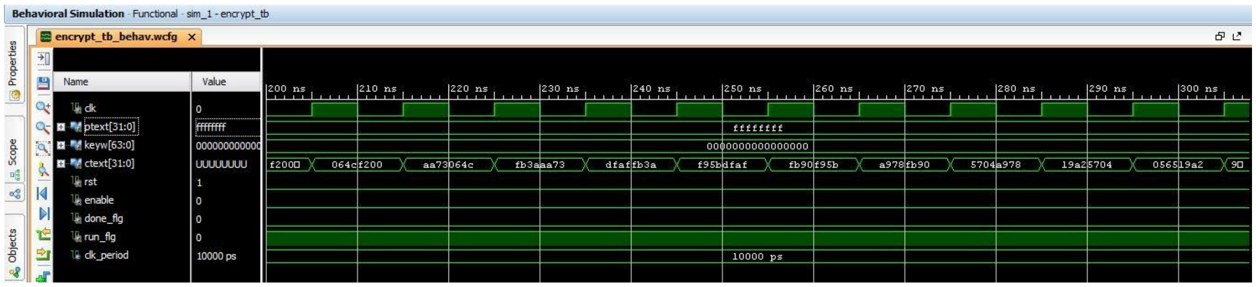
In figure 3 we see that it is very similar to figure 2. This is the key expansion with our vhdl components. Each arrow is a different signal in vhdl. Finally, the signals are port mapped to tell the fpga where the signals are traveling to and from each component.

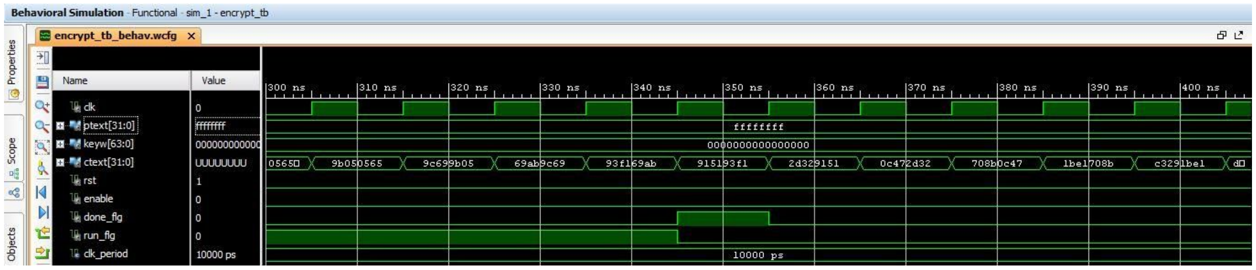
Results

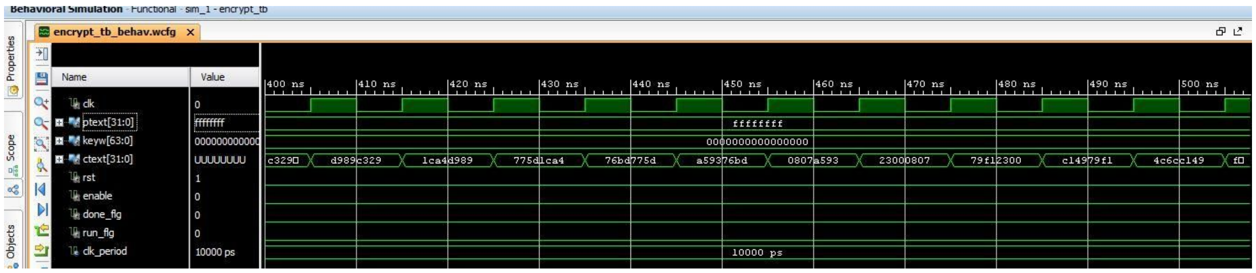
In the behavioral simulation we showed that the 64 bit passed plaintext was successfully encrypted by the key. As you can see the ctext line is the cipher text which is alternating at each round.

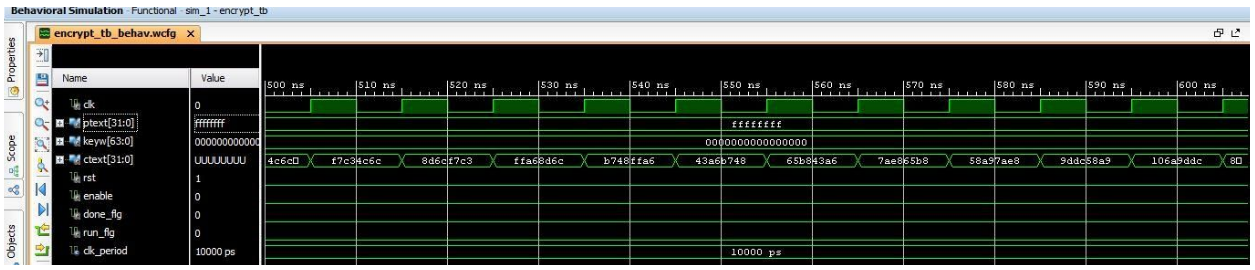

Behavioral Simulation - Functional - sim_1 - encrypt_tb
encrypt_tb_behav.wcfg
10.000 ns
Name
Value
clk 0
ptext[31:0] ffffffff
keyw[63:0] 000000000000
ctext[31:0] UUUUUUUU
rst 1
enable 0
done_flg 0
run_flg 0
clk_period 10000 ps
ffffffff
0000000000000000
UUUUUUUU
00000000
ffffffff
0002ffff
600a0002
15d6600a
10000 ps


Behavioral Simulation - Functional - sim_1 - encrypt_tb
encrypt_tb_behav.wcfg
clk 0
ptext[31:0] ffffffff
keyw[63:0] 000000000000
ctext[31:0] UUUUUUUU
rst 1
enable 0
done_flg 0
run_flg 0
clk_period 10000 ps
a14815d6
bbb4a148
7f75bbb4
26ce7f75
ee5e26ce
cd22ee5e
6c29cd22
d63f6c29
b60dd63f
f200b60d
10000 ps

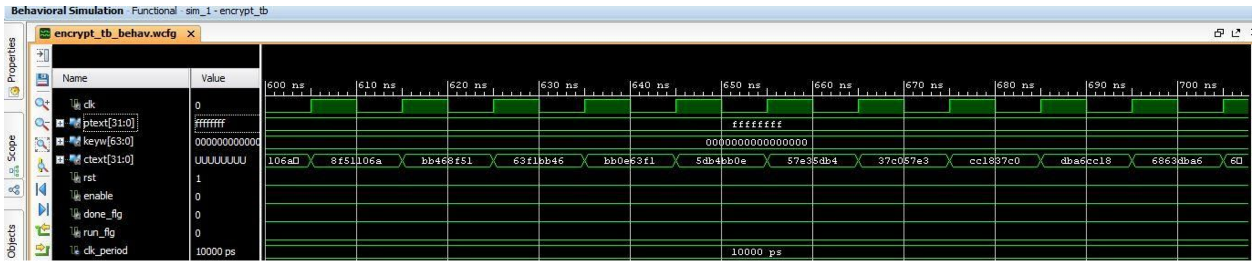

Behavioral Simulation - Functional - sim_1 - encrypt_tb
encrypt_tb_behav.wcfg
clk 0
ptext[31:0] ffffffff
keyw[63:0] 000000000000
ctext[31:0] UUUUUUUU
rst 1
enable 0
done_flg 0
run_flg 0
clk_period 10000 ps
064cf200
aa73064c
fb3aaa73
dfaffb3a
f95bdfaf
fb90f95b
a978fb90
5704a978
19a25704
056519a2
10000 ps


Behavioral Simulation - Functional - sim_1 - encrypt_tb
encrypt_tb_behav.wcfg
clk 0
ptext[31:0] ffffffff
keyw[63:0] 000000000000
ctext[31:0] UUUUUUUU
rst 1
enable 0
done_flg 0
run_flg 0
clk_period 10000 ps
9b050565
9c699b05
69ab9c69
93f169ab
915193f1
2d329151
0c472d32
708b0c47
1be1708b
c3291be1
10000 ps

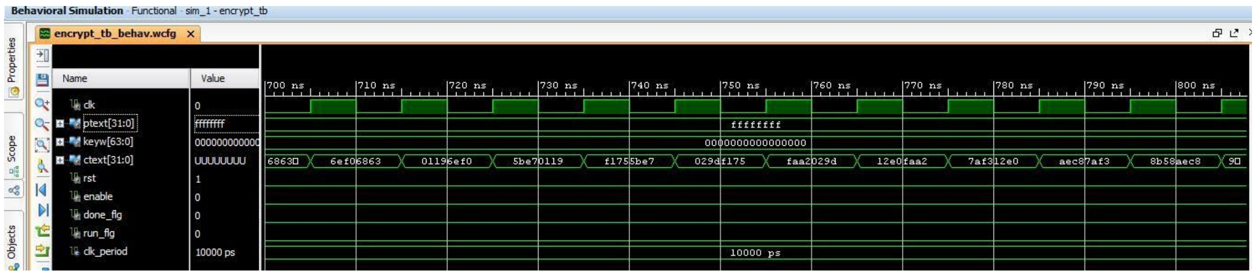

Behavioral Simulation - Functional - sim_1 - encrypt_tb
encrypt_tb_behav.wcfg
clk 0
ptext[31:0] ffffffff
keyw[63:0] 000000000000
ctext[31:0] UUUUUUUU
rst 1
enable 0
done_flg 0
run_flg 0
clk_period 10000 ps
d989c329
1ca4d989
775d1ca4
76bd775d
a59376bd
0807a593
23000807
79f12300
c14979f1
4c6cc149
10000 ps



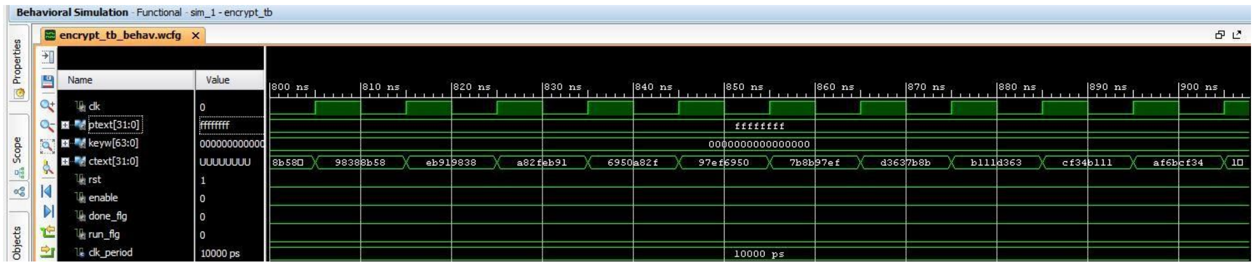
```
Behavioral Simulation - Functional - sim_1 - encrypt_tb
encrypt_tb_behav.wcfg

Name          Value
clk           0
ptext[31:0]   ffffffff
keyw[63:0]    000000000000
ctext[31:0]   UUUUUUUU
rst           1
enable        0
done_flg      0
run_flg       0
clk_period    10000 ps

500 ns  510 ns  520 ns  530 ns  540 ns  550 ns  560 ns  570 ns  580 ns  590 ns  600 ns
ptext: ffffffff
keyw:  0000000000000000
ctext: 4c6c□ f7c34c6c 8d6cf7c3 ffa68d6c b748ffa6 43a6b748 65b843a6 7ae865b8 58a97ae8 9ddc58a9 106a9ddc 8□
clk_period: 10000 ps
```



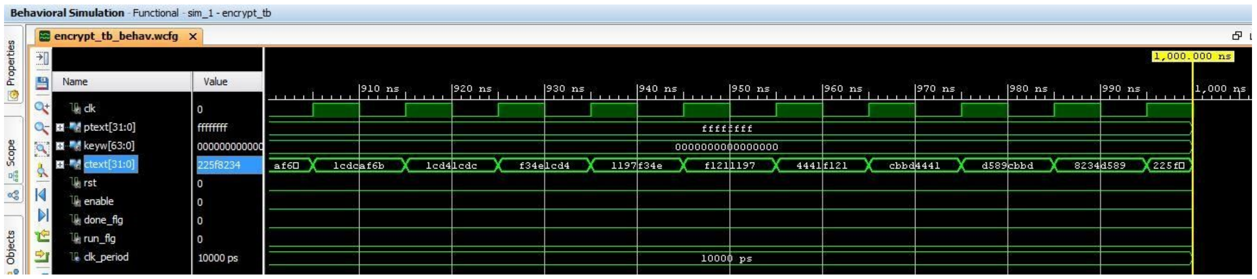
```
Behavioral Simulation - Functional - sim_1 - encrypt_tb
encrypt_tb_behav.wcfg

Name          Value
clk           0
ptext[31:0]   ffffffff
keyw[63:0]    000000000000
ctext[31:0]   UUUUUUUU
rst           1
enable        0
done_flg      0
run_flg       0
clk_period    10000 ps

600 ns  610 ns  620 ns  630 ns  640 ns  650 ns  660 ns  670 ns  680 ns  690 ns  700 ns
ptext: ffffffff
keyw:  0000000000000000
ctext: 106a□ 8f51106a bb468f51 63f1bb46 bb0e63f1 5db4bb0e 57e35db4 37c057e3 cc1837c0 dba6cc18 6863dba6 6□
clk_period: 10000 ps
```



```
Behavioral Simulation - Functional - sim_1 - encrypt_tb
encrypt_tb_behav.wcfg

Name          Value
clk           0
ptext[31:0]   ffffffff
keyw[63:0]    000000000000
ctext[31:0]   UUUUUUUU
rst           1
enable        0
done_flg      0
run_flg       0
clk_period    10000 ps

700 ns  710 ns  720 ns  730 ns  740 ns  750 ns  760 ns  770 ns  780 ns  790 ns  800 ns
ptext: ffffffff
keyw:  0000000000000000
ctext: 6863□ 6ef06863 01196ef0 5be70119 f1755be7 029df175 faa2029d 12e0faa2 7af312e0 aec87af3 8b58aec8 9□
clk_period: 10000 ps
```



```
Behavioral Simulation - Functional - sim_1 - encrypt_tb
encrypt_tb_behav.wcfg

Name          Value
clk           0
ptext[31:0]   ffffffff
keyw[63:0]    000000000000
ctext[31:0]   UUUUUUUU
rst           1
enable        0
done_flg      0
run_flg       0
clk_period    10000 ps

800 ns  810 ns  820 ns  830 ns  840 ns  850 ns  860 ns  870 ns  880 ns  890 ns  900 ns
ptext: ffffffff
keyw:  0000000000000000
ctext: 8b58□ 98388b58 eb919838 a82feb91 6950a82f 97ef6950 7b8b97ef d3637b8b b111d363 cf34b111 af6bcf34 1□
clk_period: 10000 ps
```



```
Behavioral Simulation - Functional - sim_1 - encrypt_tb
encrypt_tb_behav.wcfg

Name          Value
clk           0
ptext[31:0]   ffffffff
keyw[63:0]    000000000000
ctext[31:0]   225f8234
rst           0
enable        0
done_flg      0
run_flg       0
clk_period    10000 ps

1,000.000 ns
910 ns  920 ns  930 ns  940 ns  950 ns  960 ns  970 ns  980 ns  990 ns  1,000 ns
ptext: ffffffff
keyw:  0000000000000000
ctext: af6□ 1cdcaf6b 1cd41cdc f34e1cd4 1197f34e f1211197 4441f121 cbbd4441 d589cbbd 8234d589 225f□
clk_period: 10000 ps
```

Figure 1 - Payton_encrypt

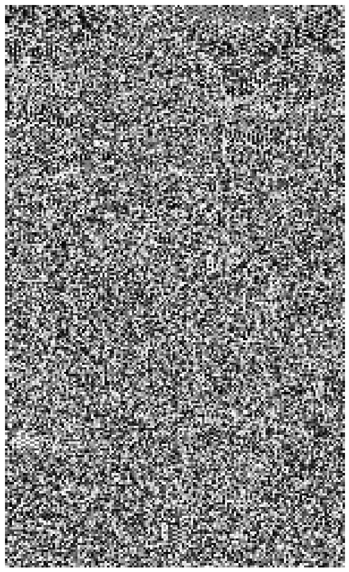
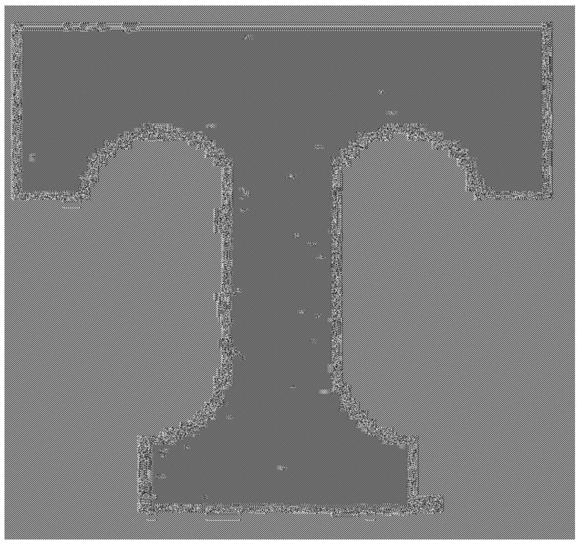


Figure 2 - PowetT_encrypt



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;
```