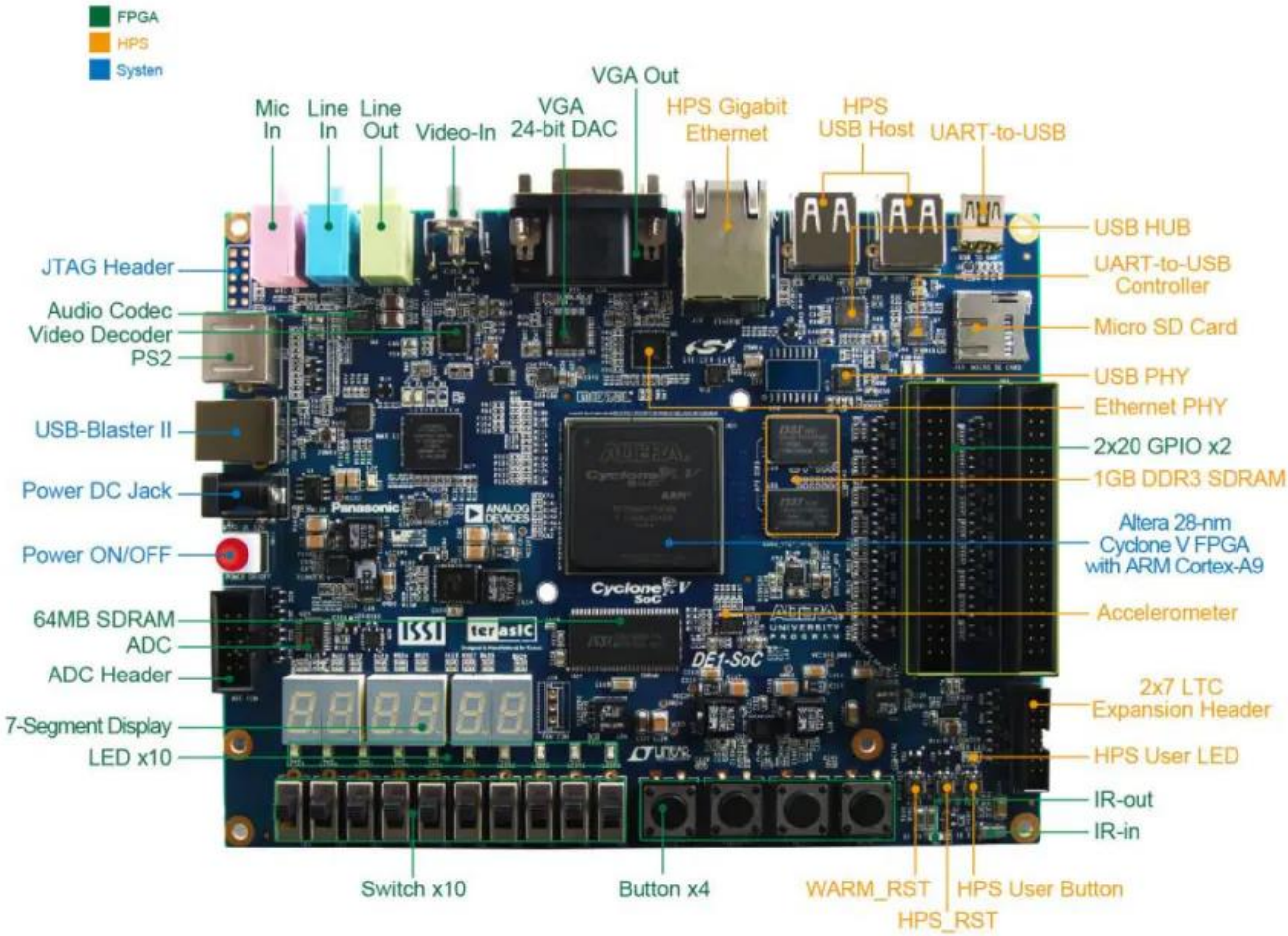
Hybrid Cryptography using DE1-SoC



Student ID: B820928
Wolfson School of Engineering
Loughborough University
21/05/2022

Hybrid Cryptography using DE1-SoC

Student ID: B820928
Wolfson School of Engineering
Loughborough University
21/05/2022

Summary

The hybrid cryptography solution was implemented on the SoC which encrypted data using the processor and sent that data to the FPGA for decryption. The FPGA circuit was described using VHDL and two solutions were developed: one optimised for performance and another optimised for balance (a compromise between speed and area). The performance-based solution uses the idea of connecting the output of one circuit to the input of the next one until the output has been computed. This method provided strong results, completing the entire algorithm in the fastest time possible. The balanced solution also provided a fast competition time while taking up less area than the performance-based solution.

The balanced solution could not further reduce the area usage unless block RAM(BRAM) was used. The use of BRAM is something that can be useful for implementing larger algorithms.

The Simon Cipher is integrated with two applications written in C. The applications outputted data to the FPGA for decryption. A function was written to send data to the FPGA and read from it, this function proved to be useful as it was re-used for the different VHDL implementations. The Simon Cipher proved to be a useful block Cipher for both hardware and software purposes.

Table of Contents

Introduction

The DE1-SoC is a system on chip that contains a dual core ARM Cortex A9 coupled with the Intel (Altera) Cyclone V FPGA. This embedded system is able run RTOS and embedded Linux. The aim of this hybrid cryptography algorithm is to use the Simon Cipher to perform encryption in the processor and decryption in the FPGA. Simon and Speck are two lightweight block ciphers which provide high performance on hardware and software. However, Simon Cipher, which is discussed in this report, is optimized specifically for hardware [1]. The report covers the hybrid cryptography algorithm in detail. There are two applications which are inputted to the Simon Cipher. The first application takes an input string from the user, packs that into 64-bit integers to be encrypted and unpacks the decrypted output from the FPGA. The other application is a user defined Fibonacci Sequence. The design of these applications in C is covered in the first part of the report.

The FPGA is used to decrypt the incoming encrypted data from the processor. This requires the use of lightweight HPS-FPGA bridge. The VHDL implementation of generating subkeys and decryption in FPGA are discussed with their limitations and justifications. All the VHDL implementations are simulated in Questasim. The design was simulated using the testbench provided by Dr Luciano Ost, moreover another testbench was created to simulate the HPS-FPGA bridge connection to emulate the processor communicating with the FPGA. There are two solutions developed for the Simon Cipher in VHDL, one was a balanced version which provides a comprise between area and performance. The other solution provides maximal performance and does not take area into account.

The report is split into two major parts: “Hybrid Cryptography System Development” and “Hybrid Cryptography System Validation”. The first part focuses mainly on the design of embedded software, VHDL implementation and the integration of the software with hardware. It also explains and justifies all the necessary design decisions. This part of the report has a specific focus on explanation and design.

The latter part of the reports provides the result of the application running on Linux terminal and the annotated waveforms produced by Questasim simulator. This part of the report is more focused on testing, validation, and comparison of FPGA resource utilization.

Hybrid Cryptography System Development

Application in Embedded Software

There were two applications designed for this project. The first application used a string packing algorithm which stored a string as an integer. The first application design will now be discussed.

The string packing algorithm uses the simple concept of integer cast to extracts chars from a string. A char is 8-bit long and the Simon Cipher required 2x 64-bit integers as input, this meant 8 chars can be stored inside one 64-bit integer. The length of the string determines the number of 64-bit integers required to store the string.

To extract the chars from the string, the subscript operator is used to access the chars in the string. The chars are then AND with 255(or 0xFF) to extract the 8-bit value. The number of required 64-bit integers were determined by dividing the length of string by 8, and if the result of the division is not a multiple of 8 then we need another 64-bit integer to store the remaining chars. For example, a string of length 20 would require 3x 64-bit integers. Dynamic memory allocation function “calloc()” is used to allocate the array of integers. This is useful as we are dealing with an embedded processor with low RAM, we cannot afford to allocate a pre-defined large array.

The algorithm is summarized in the following simple steps:

1- Determine the length of the array required and dynamically allocate it.

2- AND the chars with 0xFF in the string and assignment OR with the 64-bit integer. The next char in the string is AND with 0xFF then left shifted by 8. Once shifted the resulting value is OR with the 64-bit integer to pack it in.

3- A ‘for loop’ runs that continues this sequence until the null termination in the string is reached.

4- After packing all the chars in the string, a pointer to the resulting 64-bit integer array is passed into a function that input the 64-bit integers to the Simon algorithm, two at a time.

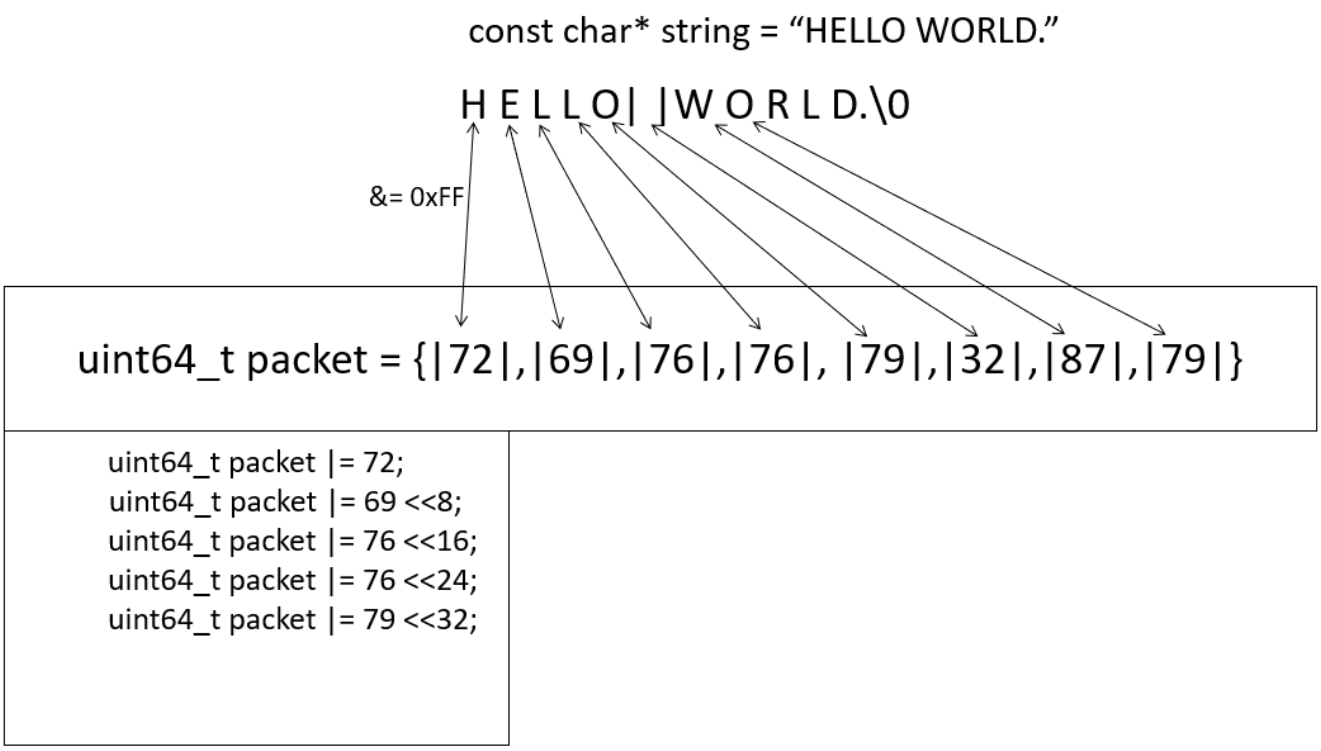


Figure 1.1. Realizing the casting of chars to int, the left shift and OR

```
void packer(const char* string, int length)
{
        int size = (length % 8) ? ((length / 8) + 1) : (length / 8); // size of array required

        uint64_t* packet = calloc(size, sizeof(*packet)); // dynamically allocate memory
        uint64_t chars = 0; // individual chars which is 8-bits

        for (size_t i = 0; i < size; ++i) // pack the string into integer array
        {
                for (size_t j = (i * 8); (j < 8 * (i + 1)) && (string[j] != '\0'); ++j)
                {
                        chars = (string[j] & 0xFF);// AND with 255 to extract 8-bit value
                        packet[j / 8] |= (chars << ((j - ((j / 8) * 8)) * 8)); // shift the bits left by 0-7
                }
        }
```

Figure 1.2. C code implementation of the char extraction

Care was taken to ensure the dynamically allocated memory was free at the end of the program.

The unpacking is done using the same thought process. Each of the 64-bit integers are AND with 0xFF then (0xFF << 8), (0xFF << 16) and so on. All the chars are then stored in a dynamically allocated char

array. The data is then shifted back to 8-bit then casted to char. The string is then printed to output the string unpacked from the 64-bit integer array.

```
void unpacker(int size, uint64_t* packet)
{
        char* unpacked = calloc(((size * 8) + 1), sizeof(*unpacked)); // dynamically allocate memory
        uint64_t extracter = 0xFFu;// used for ANDing data
        uint64_t chars = 0; // individual chars which is 8-bits
        size_t index = 0;// index to store unpacked data

        for (size_t i = 0; i < size; ++i) // for loop only runs as times as the size of the array
        {
                for (size_t j = 0; j < 8; ++j)// run 8 times since each char is 8-bit,64/8 =8
                {
                        chars = packet[i] & (extracter << (j * 8)); // extract the packed chars from the 64-bit integer
                        unpacked[index++] = (char)(chars >> (j * 8));// convert integer back to char
                }
        }
        printf("Unpacked data:\t\t\t'%s'\n\n", unpacked); // print the unpacked integer
        free(unpacked); // release dynamically allocated memory
}
```

Figure 1.3. The function to unpack data from a 64-bit integer array

The string is inputted by the user for this string packing application. This could be useful for sending messages which need to be encrypted. See Appendix A for the C code for storing an input string from the user.

The Fibonacci application sends two Fibonacci numbers to be encrypted using the Simon Algorithm. The limit of the Fibonacci numbers is specified by the user. Fibonacci numbers are stored in an array, the length of the array is also the limit of Fibonacci. This determines how many times the Simon Algorithm runs. For example, if 3 Fibonacci numbers are required to be encrypted, then the Algorithm would run twice as 2 integers are encrypted at a time, and the third Fibonacci number is encrypted in the second attempt and last integer is filled as completely zero.

```c
int limit = 0;

printf("Please enter limit of fibonacci sequence(must be greater than 2).\n");
scanf("%d", &limit);

uint64_t* fibonacci = calloc(limit, sizeof(*fibonacci));// allocate array

fibonacci[0] = 0;
fibonacci[1] = 1; // first 2 terms of fibonacci sequence

for (size_t i = 2; i < limit; ++i) // only generate until the limit
{
        fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];// generate fibonacci
}
```

Figure 1.4. Generating Fibonacci numbers to a limit specified by the user

Dynamic memory allocation is used to allocate the array. The memory is then free once the application ends.

Subkey Generation and Decryption in FPGA

There were two versions of the VHDL implementation of Simon Cipher. The first one was done using sequential process blocks that offers a nice balance between performance and area.

The balanced solution caters for both 128-bit and 192-bit key lengths. The subkey generation process block is placed inside the Simon top file due to it containing a type array for storing subkeys. The subkeys array stored the newly generated subkey and needed access to the previous ones therefore the conventional “in” and “out” port method did not work. Due to the time constraints of the project, the “inout” port could not be implemented or tested. The balanced solution design is described below:

The Simon algorithm example in C code, contained a “for loop” to generate the subkeys. This “for” loop was converted into a hardware “for” loop [2].

```
case seq_init is
    when 0=>
        continue <= '0'; -- do not start decrypting
        z <= x"fc2ce51207a635db"; -- assign value to z
        subkeys(0) <= key_64bit(2);
        subkeys(1) <= key_64bit(1);
        subkeys(2) <= key_64bit(0);
        seq_init <= 1;-- move to the next state

    when 1=>
        if l < 67 then --for i in 3 to 67 loop
            subkeys(l) <= (c xor (z and one) xor subkeys(l-3) xor ROR_64(subkeys(l-1),3)
            xor ROR_64(subkeys(l-1),4) );
            z<= shift_right(z,1);
            l <= l+1;-- increment variable each clock cycle
        else  -- for loop is completed
            subkeys(67) <=  (c xor subkeys(64) xor ROR_64(subkeys(66), 3) xor ROR_64(subkeys(66), 4) );
            seq_init <= 2;-- move to the next state when for loop is done
        end if; -- close if statement for l<67, hardware loop
    when 2=>
        subkeys(68) <= (c xor one xor subkeys(65) xor ROR_64(subkeys(67),3) xor ROR_64(subkeys(67),4) );
        continue <= '1';-- now start decrypting
    when others=> null; -- stop
end case;-- close case statements for seq_init
```

Figure 1.5. The snippet of the sequential process block which is used to generate subkeys

The continue signal is a used to prevent the decryption process block from running too early since the last generated subkey is required to decrypt data. Once the decryption is done, the continue signal is assigned a value of 1 to start decryption.

The decryption is performed using a similar sequential process block.

```
entity SIMON_CH_Decrypt is -- all these signals are required to validate the testbench provided
port
(
    clk             : in  std_logic;
    reset_n         : in  std_logic;
    data_valid      : in  std_logic; -- signal to indicate data has been received
    key_length      : in  std_logic_vector(1 downto 0); -- legnth of key 128-bit or 192-bit
    continue        : in  std_logic; -- signal to control the decryption process
    data_input      : in  t_data_in; -- input data to be decrypted
    encryption      : in  std_logic; -- signal to set to decrytin
    data_check      : in  std_logic;
    subkeys         : in  t_subkeys;

    data_ready      : out std_logic;
    x               : out unsigned(63 downto 0);-- output 64-bit decrypted data1
    y               : out unsigned(63 downto 0)-- output 64-bit decryoted data2
);
```

Figure 1.6. The port map of decryption module

Data_input is an array type which is used to store incoming as 32-bits. Intermediate signals are used to compute the final decrypted data. Decryption also follows a similar design as subkeys, the two states 1 and 2 and repeatedly performed until the variable ‘j’ has decreased to -1(Figure 1.8). Once the decryption has completed, the final value of the intermediate signals is assigned to the output x and y which represent both decrypted data.

```
elsif reset_n = '1' then

    if (data_valid = '0' and encryption = '0' and
    data_check = '1' and continue = '1') then
```

Figure 1.7. The signals required to start decryption

```
if (j >= 0) then -- for loop implementation

    case seq_dec1 is
        when 0=>
            int_x <= data_input(1);
            int_y <= (data_input(0) xor subkeys(j+1) )
            xor f(data_input(1));

            seq_dec1 <=1; -- move to the next state

        when 1=>
            int_x <= (int_x xor f(int_y) )
            xor subkeys(j);

            seq_dec1 <= 2;
        when 2=>
            int_y <= (int_y xor f(int_x) )
            xor subkeys(j-1);
            j := j-2; -- decrement for loop variable
            seq_dec1 <= 1; -- now go back to state 1

        when others=> null; -- do nothing

    end case;
else -- now the for loop has ended

    data_ready <= '1';-- data_ready is now 1, decryption done
    x <= int_x;
    y <= int_y;

end if; -- close if statement for "for loop" j >= 0
```

Figure 1.8. Process block of the decryption component 192-bit key length

The decryption only begins when the above if statement is true. When decryption has completed, another signal “data_ready” is also assigned a value of 1, this is useful for the HPS-FPGA bridge which will be explained in the next section. The signal “seq_dec1” is used for the state machine mechanism, this state is set to 0 when reset otherwise completes the decryption.

The performance-based solution did not use sequential process blocks. It used combinational process blocks and signal assignments to perform subkey generation and decryption in 0 clock cycles. The performance-based solution was split into 128-bit and 192-bit key versions, this was done to maximize performance. The concept of the performance-based solution were based on circuits being connected to another in a way where the output of circuit 1 is connected to the input of circuit 2 and so on.

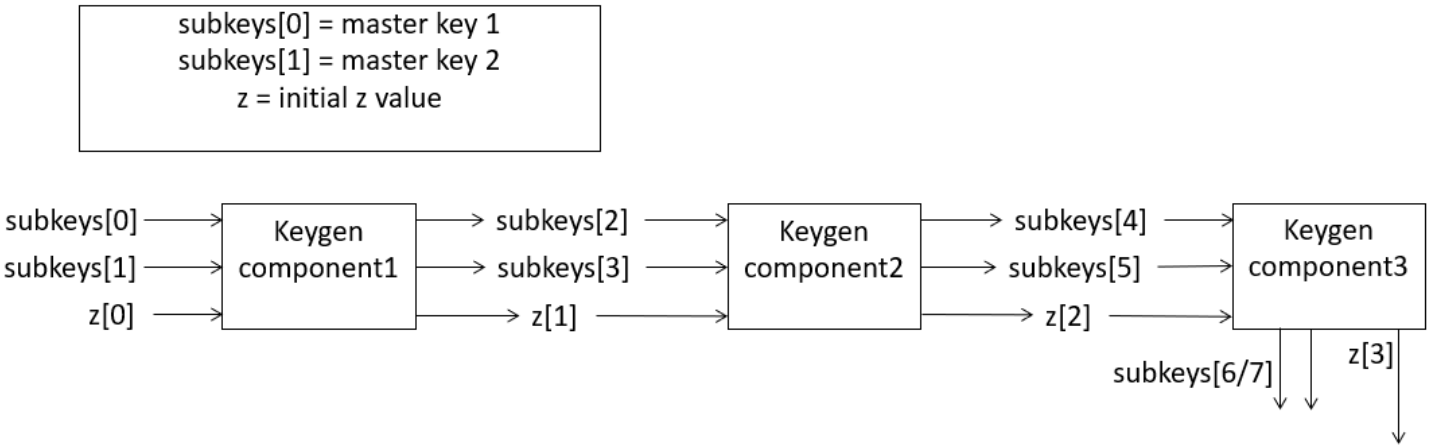


Figure 1.9. Thought process for a performance-based design

Figure 1.9 applies to 128-bit key length, but a similar design was also implemented for 192-bit key length with difference that it had three subkeys input instead of two.

```
-------------------------------------------------------Begin Key Initialisation-------
zs(0) <=  x"7369f885192c0ef5"; -- assign initial value for z

subkeys(0)  <= key_64bit(1);-- assign initial values
subkeys(1)  <= key_64bit(0);-- assign initial values

generator1 : for i in 0 to 31 generate

generate_keys : entity work.SIMON_keys_128 port map-- begin generator 1
(
    key_in_1  => subkeys(i*2),--  i=0
    key_in_2  => subkeys((i*2)+1),-- i*2+1
    z_in      => zs(i),
    key_out1  => subkeys((i*2)+2),
    key_out2  => subkeys((i*2)+3), -- max-index = 31*2 +3 = 65
    z_out     => zs(i+1)
);

end generate generator1;

subkeys(66) <= (c xor one xor subkeys(64)  -- generate remaining subkeys
                xor ROR_64(subkeys(65),3)
                xor ROR_64(subkeys(65),4));

subkeys(67) <= (c xor subkeys(65)
                xor ROR_64(subkeys(66),3)
                xor ROR_64(subkeys(66),4));

```

Figure 1.10. Generate statement and entity instantiation of subkeys generation entity(128-bit)

```
entity SIMON_keys_128 is -- all these signals are required to validate the testbench provided
port
(
    key_in_1     : in unsigned(63 downto 0); -- key input 1
    key_in_2     : in unsigned(63 downto 0);-- key input 2
    z_in         : in unsigned(63 downto 0);-- input z
    key_out1     : out unsigned(63 downto 0);-- key output1
    key_out2     : out unsigned(63 downto 0);-- key output1
    z_out        : out unsigned(63 downto 0)-- output shifted z value
);

end entity;

architecture rtl of SIMON_keys_128 is

signal int_z      : unsigned(63 downto 0);
signal int_subkey : unsigned(63 downto 0);

begin

    int_subkey <= (c xor (z_in and one) xor key_in_1
    xor ROR_64(key_in_2,3) xor ROR_64(key_in_2,4)); -- generate key

    key_out1 <= int_subkey; -- output the newly generated subkey

    int_z <= shift_right(z_in,1); -- shift the value of z and store in intermediate signal

    key_out2 <= (c xor (int_z and one) xor key_in_2
    xor ROR_64(int_subkey,3) xor ROR_64(int_subkey,4)); -- generate the next subkeys and output

    z_out <= shift_right(int_z,1); -- shift value for z


end architecture;
```

Figure 1.11. Subkey generation

The “int_subkey” is an intermediate signal which stores the newly generated subkey, then that intermediate signal is used to compute the next subkey. Two of the generated subkeys are outputted for the next entity. The constants ‘c’ and ‘one’ are defined in a VHDL, where ‘c’ is the constant for both subkey lengths and ‘one’ is just the 1 represented as 64-bit.

The same concept and design are used for 192-bit VHDL implementation of the Simon cipher.

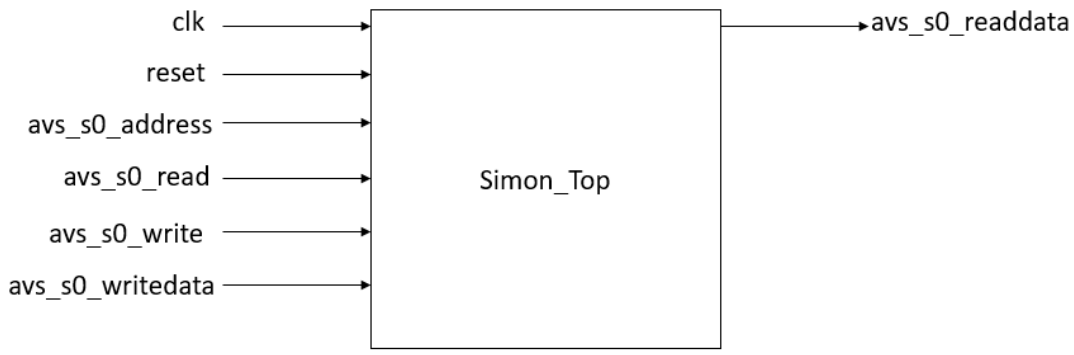
```
---------------------------------------------------------FINAL ENCRYPTED/DEC
x_dec_final <= dec_x(34);--decrypted data1
y_dec_final <= dec_y(34);--decrypted data2
-------------------------------------------------------------DECRYPTION--
dec_x(0) <= data_input(0);-- assign initial data
dec_y(0) <= data_input(1);

generator3 : for i in 0 to 33 generate -- begin generator 3

decrypt_data : entity work.SIMON_Decrypt_128 port map
(
    x_in          => dec_x(i),
    y_in          => dec_y(i),
    subkey_in1    => subkeys(67 - (i*2)),
    subkey_in2    => subkeys(67- ((i*2)+1)),--min_index = 67-(33*2+1)=0
    x_out         => dec_x(i+1),
    y_out         => dec_y(i+1)
);
end generate generator3;
-----------------------------------------------------END-------------------
```

```
entity SIMON_Decrypt_128 is -- all these signals are required to validate the test
port
(
    x_in        : in unsigned(63 downto 0);-- take x as input(data1)
    y_in        : in unsigned(63 downto 0);-- take y as input(data2)
    subkey_in1  : in unsigned(63 downto 0);-- take subkey1 as input
    subkey_in2  : in unsigned(63 downto 0);-- take subkey2 as input
    x_out       : out unsigned(63 downto 0);-- output the new x value
    y_out       : out unsigned(63 downto 0)-- output the new y value
);

end entity;

architecture rtl of SIMON_Decrypt_128 is

signal x_int : unsigned(63 downto 0); -- intermediate signal for storing x value

begin


    x_int <= (x_in xor f(y_in)) xor subkey_in1; -- compute x
    x_out <= x_int; -- assign output signal to x_int
    y_out <= (y_in xor f(x_int)) xor subkey_in2;-- compute y


end architecture;
```

Figure 1.12/1.13. The entity instantiation of decryption module

Please see Appendix B for the VHDL package that contained the function “f()”. This package is used in all the VHDL files to import the function and types required. An array of type unsigned (63 downto 0) was used to store the intermediate decryption signals. The decrypted data was contained in the final element of the array which was assigned to “x_dec_final” and “y_dec_final”. The final signal is then read by the processor from the FPGA (more on this later).

The decryption has a similar design concept with the output of one circuit connected to the input of another. The port map of the top-level file can be summarized as the block diagram shown below:



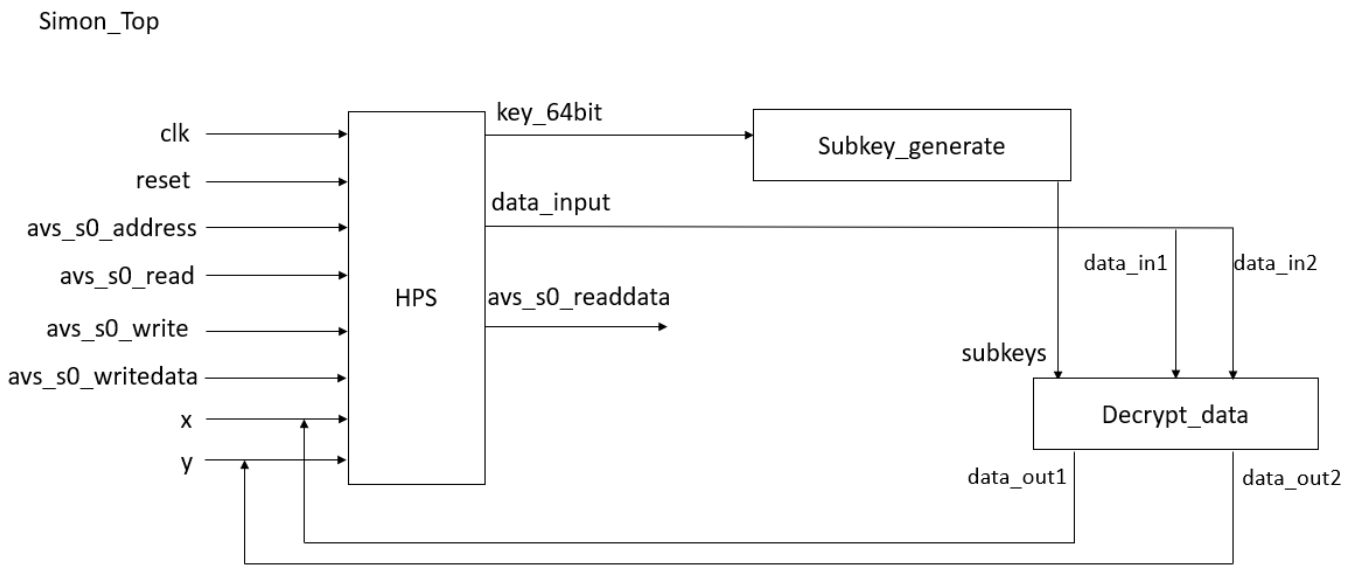


Figure 1.14/1.15. A simplified version of the top-level file

Lightweight HPS-FPGA bridge

The lightweight HPS-FPGA bridge was used to allow the HPS to communicate with the FPGA, this is a 32-bit bridge. This bridge allows the user to send data from the HPS to the FPGA. The input reset was used to change the value of “reset_n” in the design. The advantage of using the reset_n is that it allows the user to reset the system by sending data at that address and not use the HPS reset button.

```
  -- purpose: Respond to write operations from the Avalon bus
  -- type    : sequential with asychronous reset
  -- inputs : clk, reset, avs_s0_write, avs_s0_address
  -- outputs: none
write_proc : process (clk, reset) is
begin  -- process write_proc
  if reset = '1'  then                  -- reset active high
    reset_n <= '0';-- reset

  elsif rising_edge(clk) then  -- rising clock edge
    if avs_s0_write = '1' then
      case avs_s0_address is

        when b"0000" => reset_n <= avs_s0_writedata(0);----registor #0

        when b"0001" => key32_1  <= unsigned(avs_s0_writedata);-----registor #1
        when b"0010" => key32_2  <= unsigned(avs_s0_writedata);-----registor #2
        when b"0011" => key32_3  <= unsigned(avs_s0_writedata);-----registor #3
        when b"0100" => key32_4  <= unsigned(avs_s0_writedata);-----registor #4

        when b"0111" => data32_1 <= unsigned(avs_s0_writedata);-----registor #7
        when b"1000" => data32_2 <= unsigned(avs_s0_writedata);-----registor #8
        when b"1001" => data32_3 <= unsigned(avs_s0_writedata);-----registor #9
        when b"1010" => data32_4 <= unsigned(avs_s0_writedata);-----registor #10

        when others => reset_n <= '1'; -- do nothing
      end case;
    end if;
  end if;
end process write_proc;
```

Figure 1.16. Snippet of the write process block for HPS-FPGA bridge

In the above figure, reset_n is assigned the first element of avs_s0_writedata which is 32-bit std_logic_vector. This allows the user to reset the system if ‘0’ is written to register #0.

The 32-bit data and key values are then concatenated into 64-bit arrays for subkey generation and decryption. This is done using a combinational process block that fills the array to zero if the reset_n is assigned ‘0’ by the user. See Appendix C for the corresponding C code for the processor.

Encryption in Embedded Software

Both applications output integers which are then inputted to the Simon Cipher algorithm in C. The data from the applications is stored in an array of 64-bit integers. The algorithm is run in a “for loop” and the number of iterations is determined by the size of the array.

The Simon Algorithm encrypts and decrypts data two at time. If the size of the array is a multiple of 2 then the algorithm is run size/2 times. If the array size is not a multiple of 2 then the algorithm is run size/2 times, but the last remaining data value is encrypted and the second input to the Simon Algorithm is filled with zeroes. For example, if the size of the array is 3, then the for loop is run once encrypting the first two values, and the last value is encrypted with the other input to the Simon set to zero.

```c
    }
    else if (limit % 2)// if the limit is not a multiple of 2
    {
        for (size_t i = 0; i < limit; i += 2)
        {
            text[0] = fibonacci[i];
            text[1] = fibonacci[i + 1];

            SIMON_encrypt(&context, text, cipherText);// encrypt data
            encrypted[i] = cipherText[0];
            encrypted[i + 1] = cipherText[1]; // store the encrypted data in array
            FPGA_decrypt(key,cipherText,decryptedText);// decrypt data using the FPGA
            decrypted[i] = decryptedText[0];
            decrypted[i + 1] = decryptedText[1];// store the decrypted data in array

            if (i + 2 == limit)// reached the last remaining number
            {
                text[0] = fibonacci[i+2]; // assign last value of array
                text[1] = 0; // make it zero

                SIMON_encrypt(&context, text, cipherText);
                encrypted[i + 2] = cipherText[0];// store the last encrypted fibonacci number
                FPGA_decrypt(key,cipherText,decryptedText);// decrypt data using the FPGA
                decrypted[i + 2] = decryptedText[0];// store the last decrypted fibonacci number
                break; // end the for loop
            }
        }
    }
```

Figure 1.17. C code for encrypting data depending on the size of array

The “text[]” is the array which is inputted to the Simon Algorithm. Expanding on the previous example of size of the array being 3; the “i” variable starts as 0, the if statement checks if i+2 equals 2. The index 2 is third element in the array, in our example the if statement is true and it simply makes the text[0] to the last remaining data and text[1] to 0.

(The “FPGA_decrypt()” function is not discussed in the report but more information can be found in Appendix C).

Hybrid Cryptography System Validation

Validating Applications

The string application is validated by inputting a large string to check if the algorithm is able to successfully decrypt the data. The C code is run from the Linux terminal, which is accessed using ethernet cable and VNC viewer software.

```
File Edit View Search Terminal Help
root@de1soclinux:~/tutorial_files/Simon_Applications# gcc -std=gnu99 main.c -std=gnu99 SIMON.c -o simon_run
root@de1soclinux:~/tutorial_files/Simon_Applications# ./simon_run
Select application(enter either 1,2 or 3):
1 - Send secret message
2 - Fibonacci Number encryption
3 - Quit.
1
Enter your message(terminated by '*'):
This is for testing purposes, let's check if this string is decrypted properly.*
'This is for testing purposes, let's check if this string is decrypted properly.' is packed to give:
1- 2073692073696854
2- 7473657420726f66
3- 7072757020676e69
4- 656c202c7365736f
5- 6365686320732774
6- 696874206669206b
7- 676e697274732073
8- 7263656420736920
9- 7270206465747079
10- 2e796c707265706f
```

Figure 2.1. The string inputted from user being packed into integers

```
192-bit key Test---------------------------------------------
key:                        1716151413121110 0f0e0d0c0b0a0908 0706050403020100
text:                       2073692073696854 7473657420726f66
encrypted text:             70c14ed3ca0d241a e3aadff4da06f4da
decrypted text:             2073692073696854 7473657420726f66
Unpacked data:              'This is for test'

key:                        1716151413121110 0f0e0d0c0b0a0908 0706050403020100
text:                       7072757020676e69 656c202c7365736f
encrypted text:             0bcace28977bd13b a46faed84a491fbc
decrypted text:             7072757020676e69 656c202c7365736f
Unpacked data:              'ing purposes, le'

key:                        1716151413121110 0f0e0d0c0b0a0908 0706050403020100
text:                       6365686320732774 696874206669206b
encrypted text:             a3ec3a19c513a9bf 2771853cecd87f5a
decrypted text:             6365686320732774 696874206669206b
Unpacked data:              't's check if thi'

key:                        1716151413121110 0f0e0d0c0b0a0908 0706050403020100
text:                       676e697274732073 7263656420736920
encrypted text:             58fefd2c12ded2b0 f0f50fa57dc0d91a
decrypted text:             676e697274732073 7263656420736920
Unpacked data:              's string is decr'

key:                        1716151413121110 0f0e0d0c0b0a0908 0706050403020100
text:                       7270206465747079 2e796c707265706f
encrypted text:             181852be1b0cc3b1 f14e8f98c0e4f2d8
decrypted text:             7270206465747079 2e796c707265706f
Unpacked data:              'ypted properply.'

Unpacked data:              'This is for testing purposes, let's check if this string is decrypted properply.'

Select application(enter either 1,2 or 3):
1 - Send secret message
2 - Fibonacci Number encryption
3 - Quit.
```

Figure 2.2. The packed integers decrypted and unpacked in software

The Fibonacci sequence was validated in software by inputting it to the Simon Algorithm.

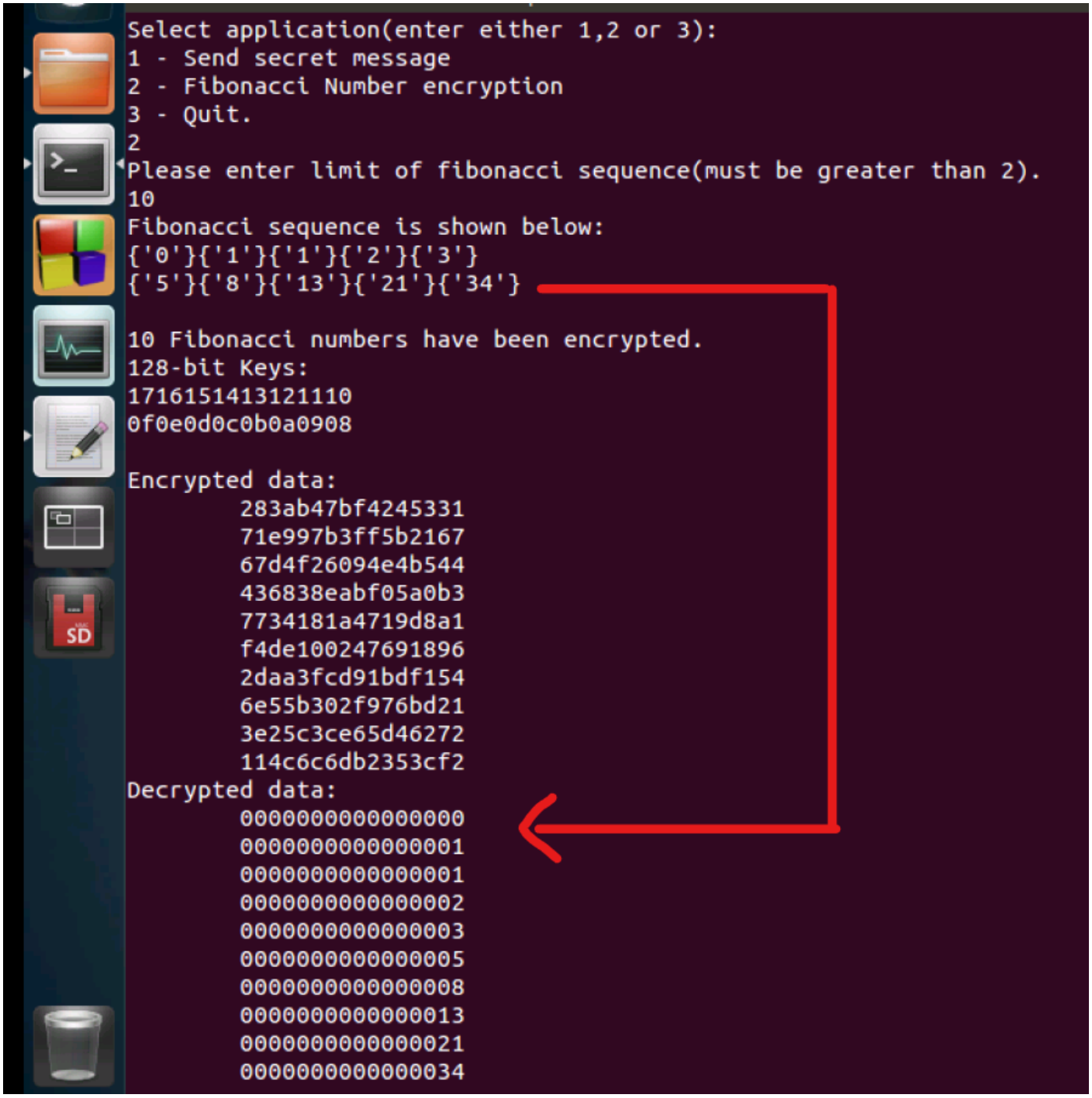


Figure 2.3. Fibonacci sequence being produced and encrypted/decrypted

Simulating the Balanced Solution

The VHDL implementation of Simon Cipher were validated using both the testbench provided and a testbench specifically for the HPS-FPGA bridge. The balanced solution can perform the algorithm using both key lengths therefore another test was added to the testbench for 192-bit key length. (Please zoom to about 150%-200% to see the waveforms clearly, higher quality screenshots can be found in the corresponding folder).

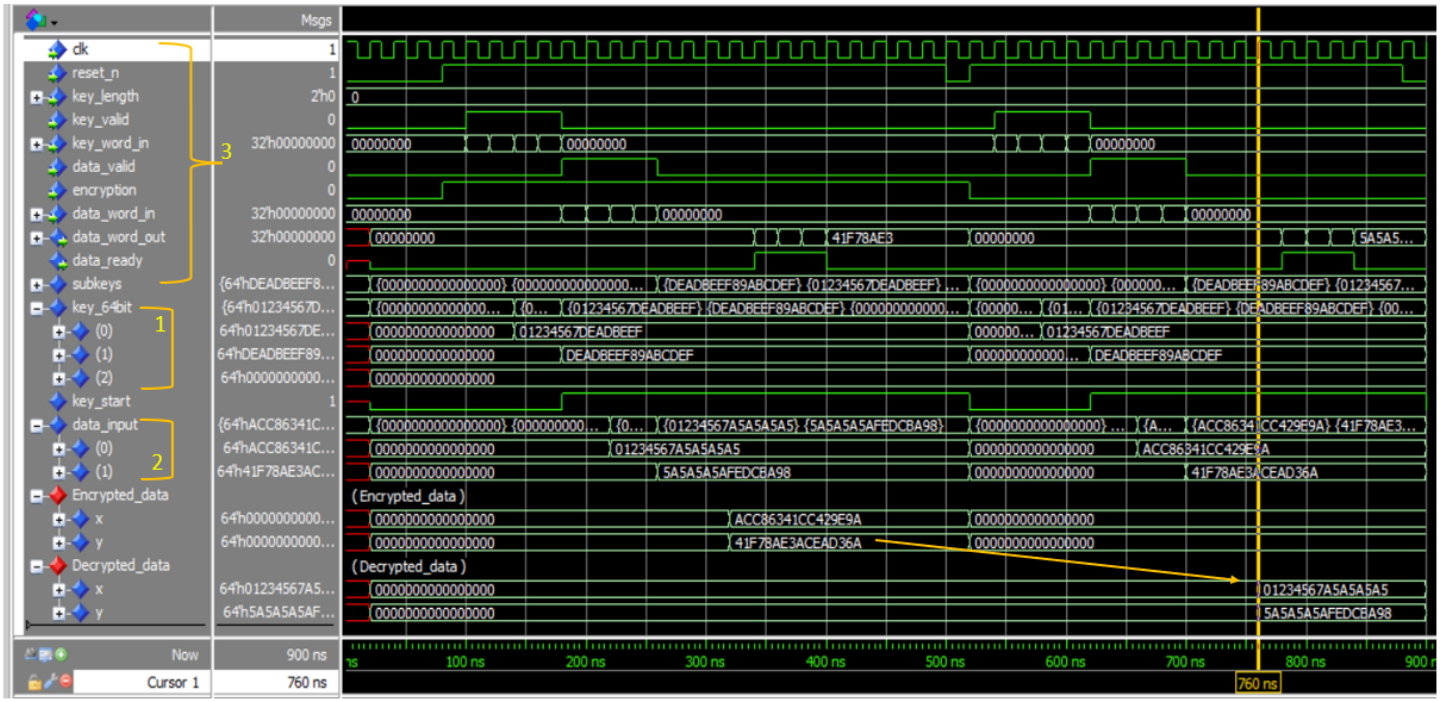


Figure 2.4. Simon Cipher simulated using the testbench provided (128-bit key length)

The pointed yellow arrow shows the encrypted data which is decrypted.
1- shows the 64-bit array used to store incoming 32-bit keys. Note that the element 2 is set to zero since the key length is 128-bit.
2- Shows incoming 32-bit data stored in a 64-bit array.
3-Are all the signals required to validate the testbench provided. The entire Simon Cipher for 128-bit is completed in 760 ns.

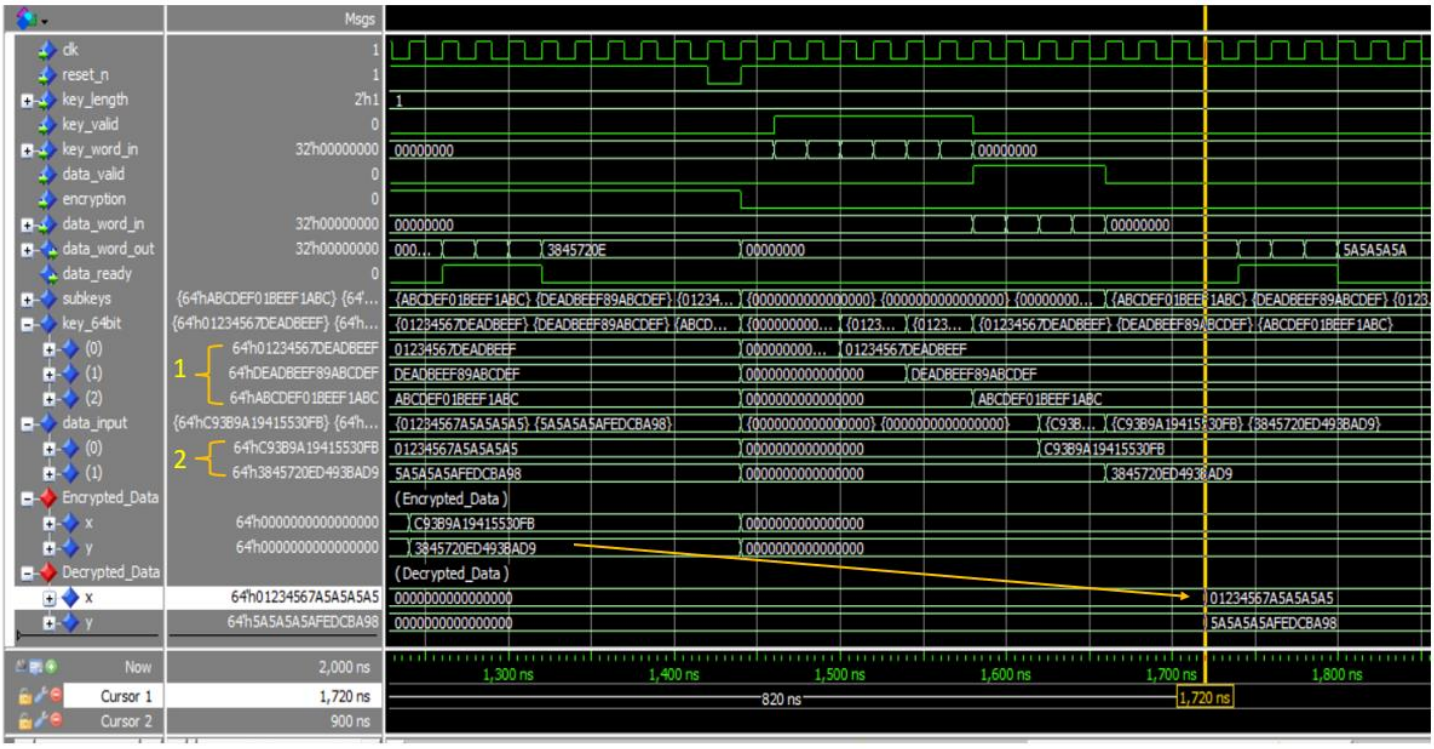


Figure 2.5. Simon Cipher simulated using the testbench provided (192-bit key length)

1- Shows the 192-bit keys stored in the array. Note that element 2 has been filled with a key this time.
2- This is the input data stored as 64-bit, same as Figure 2.4.
The pointed yellow arrow shows the encrypted data, which is decrypted at 1720ns, the testbench resets at 900ns after completing the 128-bit key length test. The total time for the Simon Cipher can be seen as 820ns, slightly longer than for 128-bit key length.

The following testbench was for validating the HPS-FPGA bridge and contains subkeys generation with decryption. The encryption module was removed for the top file and the number of variables were reduced since they caused Intel Quartus some problems, the variables were replaced by signals instead. Therefore, the balanced solution VHDL implementation was slightly altered to ensure synthesis.

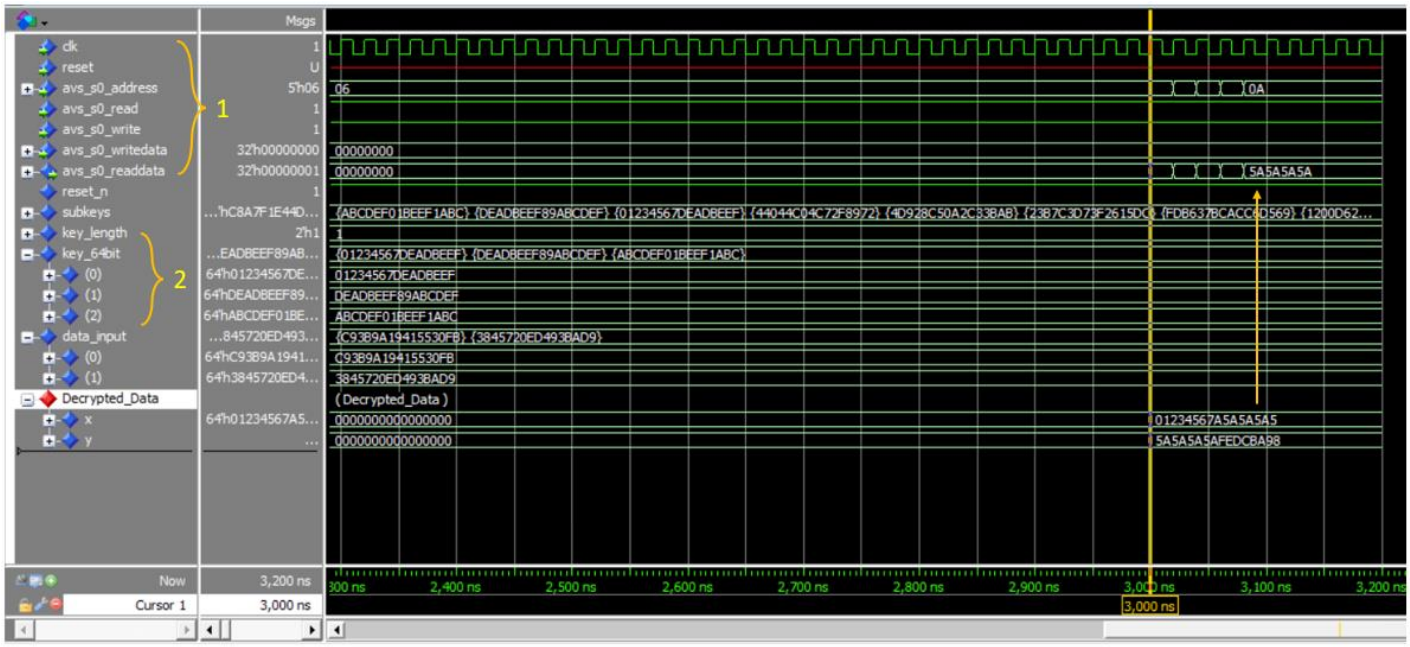


Figure 2.6. Simulation of the Hybrid balanced solution(192-bit).
1- All these are the signals required to communicate with the HPS.
2- The key_length signal is set to ‘1’ which indicates a 192-bit key length. key_64bit shows the 192-bit being stored.

The pointed yellow arrow emulates the last of the decrypted data transferred in 32-bit to the HPS. The total time to complete subkeys generation and decryption is much higher at 3000ns. Therefore, the signal data_ready goes high when the decryption has been completed. The testbench then tests for 128-bit key length, this is not shown since it has a very similar waveform to 192-bit but slightly faster.

Simulating the Performance based Solution

The following waveforms are shown for the 192-bit key length. The test for 128-bit key length is almost identical and does not convey any additional information to the reader therefore it is not included in this section.

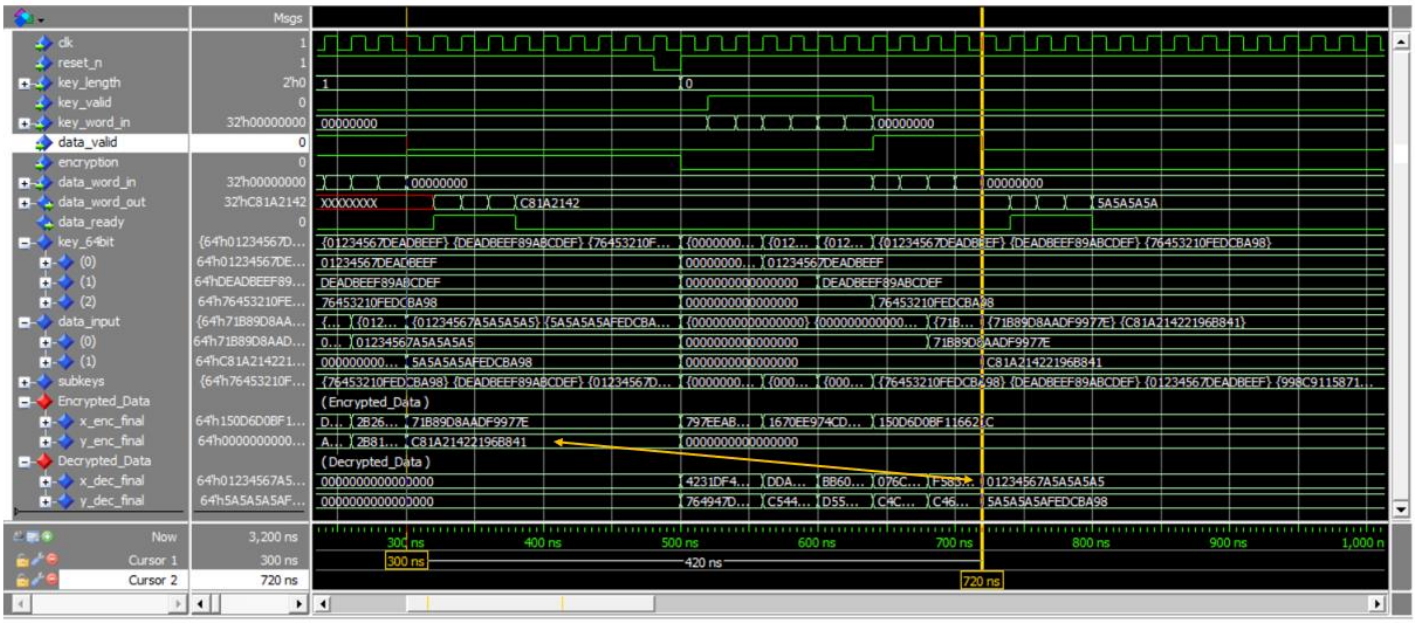


Figure 2.7. The doubly pointed yellow arrow shows the encrypted and decrypted data using the testbench provided

This performance-based method rapidly generates subkeys, encrypts and decrypts data as soon as the required data is received. The whole Simon Cipher is completed in 720ns, this is the fastest possible implementation for 192-bit key length. The figure below shows the generation of subkeys.

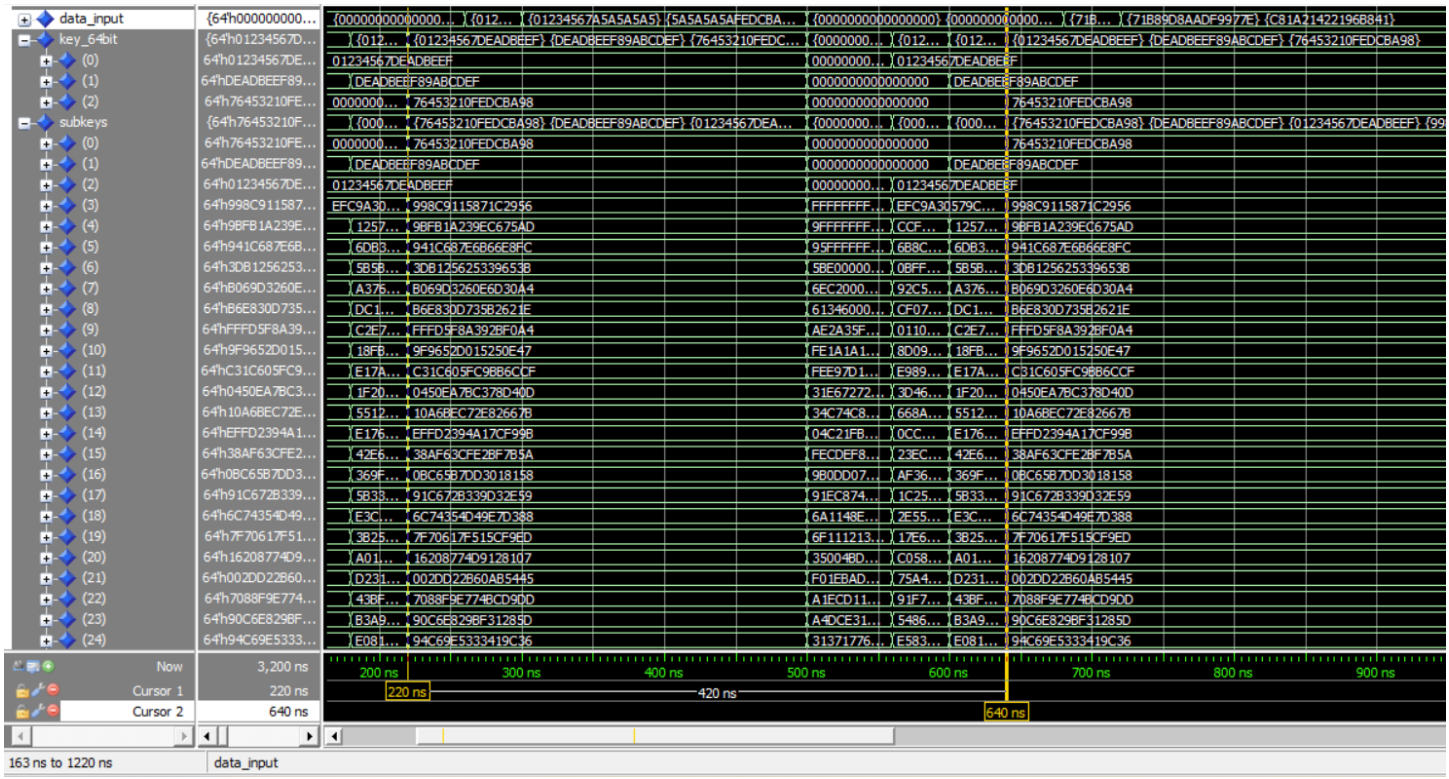


Figure 2.8. Rapid subkeys generation

Cursor 1 and 2 shows that as soon as element 2 of the key_64bit array is received, all the subkeys are generated.

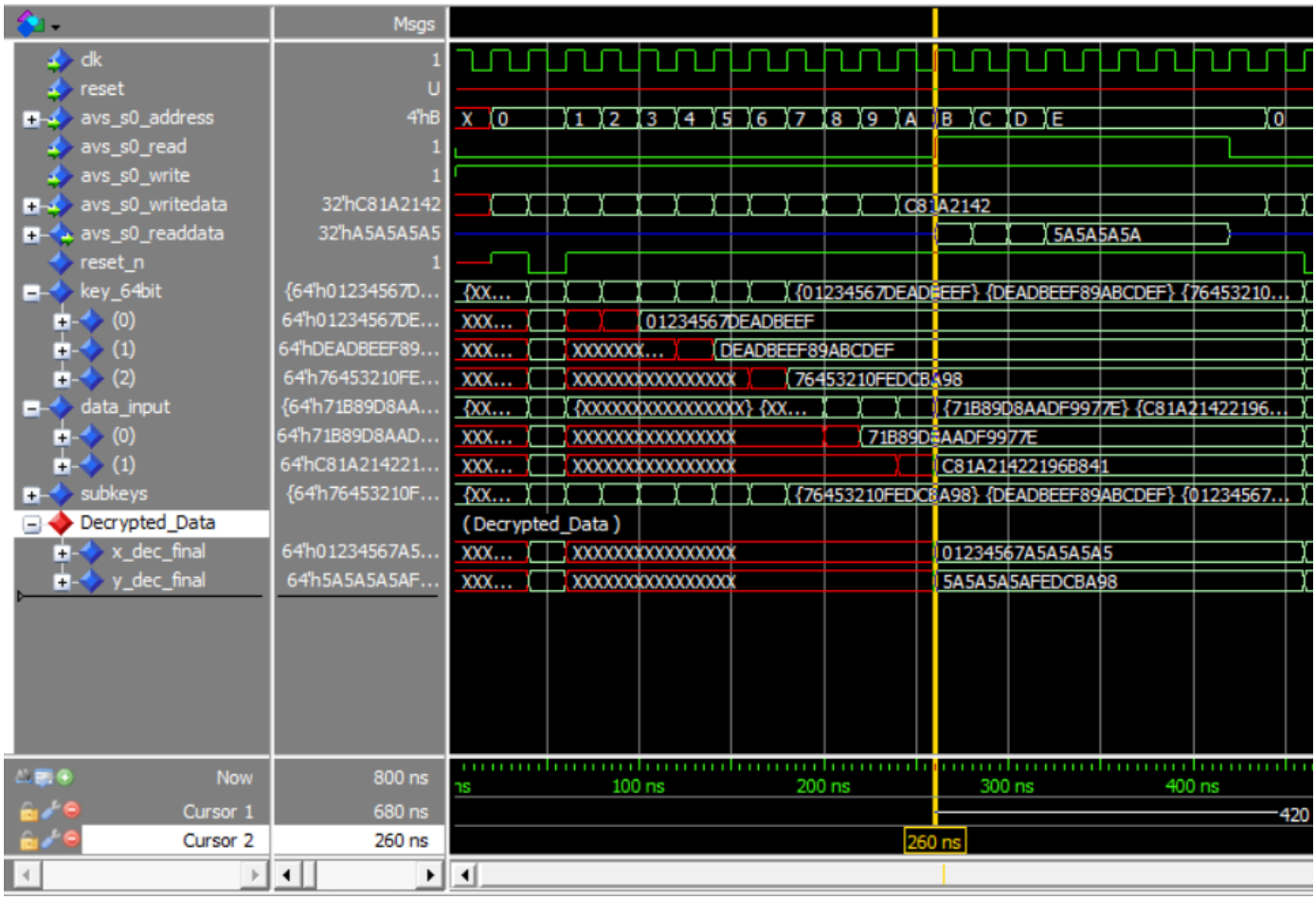


Figure 2.9. The waveform for the HPS-FPGA bridge integrated with Simon Cipher

The "XXX…" symbols stand out in the figure above. These simply appear due to how the testbench is setup and do not warrant any concern. In the testbench the "reset" signal for the HPS is not used, moreover the avs-s0-address is also not initialized and the "when others" statement inside the HPS write process is executed. This starts the signal concatenation. Since the first 32-bit values for key and data are uninitialized, they are combined to produce "X" symbol. The cursor at 260 ns shows that as soon as the data is received, the decrypted data is outputted. The entire Simon Cipher in Hardware is completed in 200ns (after subtracting the reset time) for 192-bit key length.

Comparison of the VHDL Solutions

The VHDL solutions are compared to analyze the area usage and speed in simulation.

Solution Type	Area		
	Adaptive Logic Modules	Registors	Block Memory Bits
Balanced Solution	10851	9970	526336
Performance(128-bit)	11631	3368	526336
Performance(192-bit)	11295	3485	526336
Solution Type	**Time(ns)**		
	Subkeys Generation	**Decryption**	**Total time**
Balanced Solution	1340	1400	3100
Performance(128-bit)	0	0	300
Performance(192-bit)	0	0	340

Figure 2.10. Area and speed comparison of the different VHDL solutions

The measurements are taken in Questasim simulator, starting from as soon as the process block starts executing. For example, in the case of decryption the time measured is when the subkeys are generated up until when the data is decrypted. The total time represents the time it takes to output the very last 32-bit decrypted data and takes all reset, reads and writes from the Avalon bus into account.

The Adaptive Logic Modules are described as LUT-based resources that can be divided between two adaptive LUTs (ALUTs). One ALM can implement any function with up to six inputs and certain seven-input functions. In addition to the ALUT-based resources, each ALM contains two programmable registers, two dedicated full adders, a carry chain, a shared arithmetic chain, and a register chain. The ALM can efficiently implement various arithmetic functions and shift registers with these dedicated resources [3].

It can also be important to consider the time it took for Intel Quartus Prime to synthesize these designs. The compilation settings were changed to “Area Aggressive”.

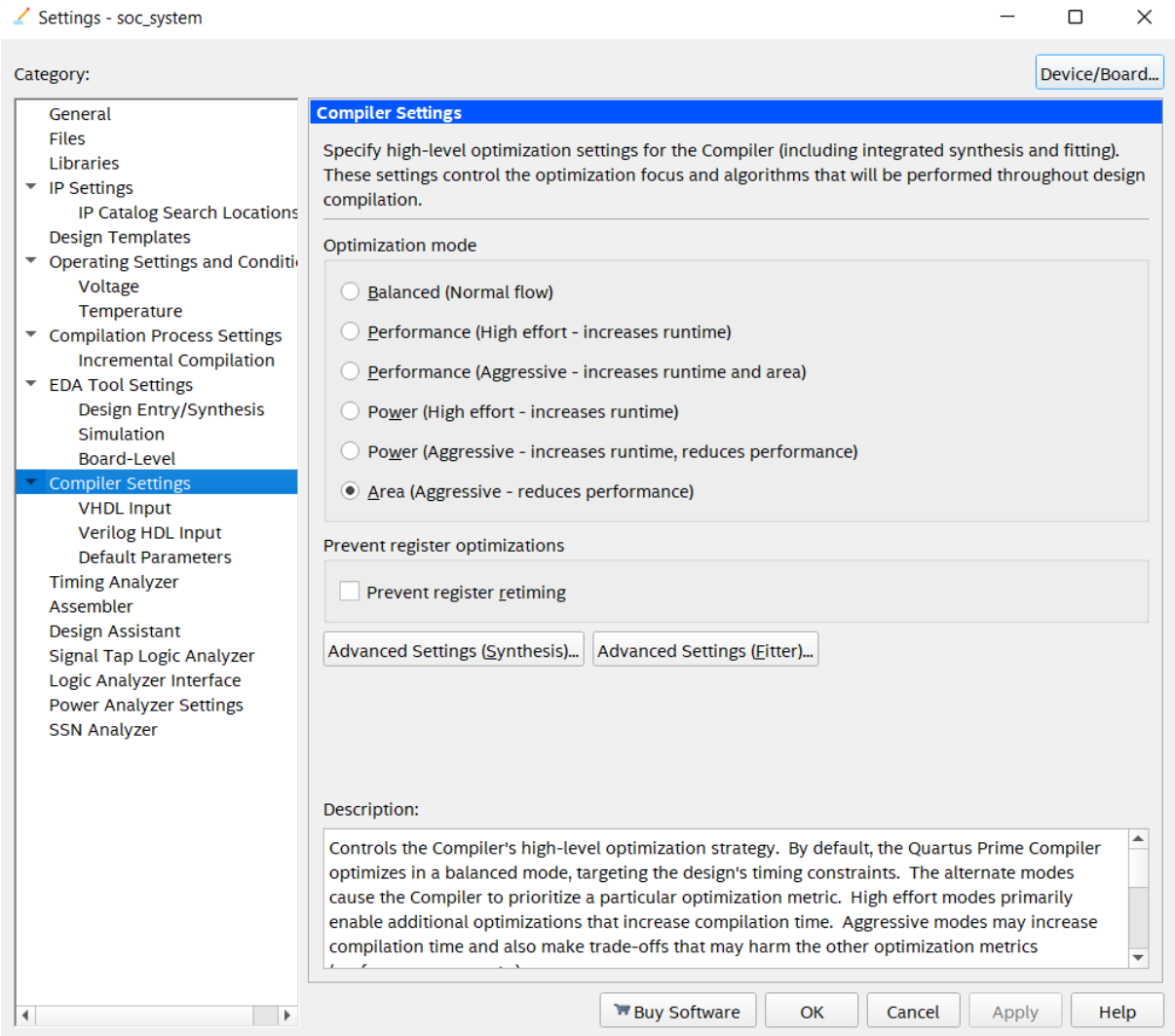


Figure 2.11. Quartus Prime Compiler settings

Table 2.1. Time to synthesize comparison.

Solution Type	Time to synthesize (Minutes: Seconds)
Balanced Solution	17:13
Performance(128-bit)	44:11
Performance(192-bit)	39:47

The synthesize time is of course dependent on the machine used. The results shown in table 2.1. were obtained from the specifications:

Processor: AMD Ryzen 7 5800H
RAM: 16 GB DDR4 3200Mhz
OS: Windows 11

The balanced solution does use less ALMs but then uses significantly uses more registers than the performance-based solutions. The time it takes to compile the hybrid solutions yielded disappointing results especially for the performance-based solution for 128-bit taking up to 44

minutes. This is clearly not desirable as the Simon Cipher is supposed to be a lightweight algorithm designed for hardware. A possible method to further reduce the area usage of the design would be to use dual port RAM. This would use the dedicated piece of silicon on the DE1-SoC to store the subkeys instead of storing them in a signal array. Due to the time constraints of the project, a RAM based solution could not be implemented.

Conclusion

The area usage of the balanced solution provided strong results with a lower ALM usage than the performance-based solutions. The speed difference between the solutions was large however even the balanced solution managed to complete the algorithm in a respectable time. What is fast and what is slow? This is dependent on the design requirements of the system. The applications performed as expected. The VHDL and C code could have been made more modular if BRAM was used, this would allow all the components to be split into different files which allows easy removal of components in the top-level file.

The Simon Cipher implementation on the FPGA was able to drastically outperform the embedded processor. Simon and Speck are two block ciphers that are optimized for hardware, it remains to be seen whether other Cipher algorithms are able to provide the same high performance in the FPGA as Simon/Speck. In the end, the power of FPGA was evident as it provided precise control over the number of instructions performed in clock cycles, making it a flexible device.

Appendix A: Storing a string from user input in C

To convert a string to integers for the first application, the following C implementation was used. It uses the realloc() function and returns the pointer to the first char of the string. The string chars are stored at a memory address until the specified '*' symbol has been reached [4].

```c
/*
    Function : inputString
    Purpose : This function is used to take a string from the user of an
    unknown length
    Arguments: pointer to FILE struct, size_t initial length of string
    Return values: pointer to the first element of string
*/
char* inputString(FILE* fp, size_t size) {
    //The size is extended by the input with the value of the provisional
    char* str;
    int ch;
    size_t len = 0;
    str = realloc(NULL, sizeof(*str) * size);//size is start size
    if (!str)return str;
    while (EOF != (ch = fgetc(fp)) && ch != '*') {
        str[len++] = ch;
        if (len == size) {
            str = realloc(str, sizeof(*str) * (size += 16));
            if (!str)return str;
        }
    }
    str[len++] = '\0';

    return realloc(str, sizeof(*str) * len);
}
```

Figure A1.0. C code for storing a string from the user of an unknown length

Appendix B: The VHDL package used

Since many of the VHDL components used the same functions and types, it made sense to declare all the required functions and types in a package. The main advantage of this is that components can be ported using the types. Moreover, using a package significantly reduces clutter and makes the VHDL files look neater [5].

```
-- This packages contains all the usefull array types
-- Also contains repeated constants and functions
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all; -- use library for unsigned

package SIMON_C_PACKET is

 -- one represented as 64-bit
constant one    : unsigned(63 downto 0):= B"0000000000000000000000000000000000000000000000000000000000000001";
constant c      : unsigned(63 downto 0):= x"fffffffffffffffc"; -- constant value for both keys length
type t_subkeys   is array(0 to 68) of unsigned(63 downto 0);-- store generated subkeys
type t_key_64bit is array(0 to 2)  of unsigned(63 downto 0);-- store keys as 64-bit
type t_data_in   is array(0 to 1)  of unsigned(63 downto 0);-- store data as 64-bit

function ROR_64(x : in unsigned(63 downto 0); n : in integer)-- Rotate RIGHT circular shift 32 bits
    return unsigned;

function ROL_64(x : in unsigned(63 downto 0); n : in integer)-- Rotate LEFT circular shift 32 bits
    return unsigned;

function f(x : in unsigned(63 downto 0)) -- helper function for emulating the "R2" function
    return unsigned;


end package SIMON_C_PACKET;

package body SIMON_C_PACKET is

  function ROR_64(x : in unsigned(63 downto 0); n : in integer)-- Rotate RIGHT circular shift 32 bits
    return unsigned is variable shifted : unsigned(63 downto 0);
begin
    shifted := ( shift_right(x,n) OR shift_left(x,(64-n)) );
    return unsigned(shifted);
end function;

function ROL_64(x : in unsigned(63 downto 0); n : in integer)-- Rotate LEFT circular shift 32 bits
    return unsigned is variable shifted : unsigned(63 downto 0);
begin
    shifted := ( shift_left(x,n) OR shift_right(x,(64-n)) );
    return unsigned(shifted);
end function;

function f(x : in unsigned(63 downto 0)) -- helper function for emulating the "R2" function
        return unsigned is variable rolled : unsigned(63 downto 0);
begin
        rolled := ( (ROL_64(x,1) and ROL_64(x,8)) xor ROL_64(x,2) );
        return unsigned(rolled);
end function;


end package body SIMON_C_PACKET;
```

Figure B1.0/B1.1. The required package and its body section

The package is used in the corresponding VHDL files as “use work.SIMON_PACKAGE.all”.

Appendix C: HPS-FPGA bridge access in Linux

The lightweight HPS-FPGA bridge is accessed using the following C code.

```c

    volatile uint32_t* pData;
    int fd = -1;
    void* LW_virtual;

    if ((fd = open_physical(fd)) == -1)
        return -1;
    if (!(LW_virtual = map_physical(fd, LW_BRIDGE_BASE, LW_BRIDGE_SPAN)))
        return -1;

    pData = (uint32_t*)(LW_virtual + COUNT_BASE); // use base address of 0x0

    *(pData + 0x0) = 0; // reset first
    *(pData + 0x0) = 1;// now start program, reset_n set to 1

//-------------------------------------------------------------SENDING KEY
    *(pData + 0x1)  = key32[0]; // key word in # 0
    *(pData + 0x2)  = key32[1]; // key word in, #1
    *(pData + 0x3)  = key32[2]; // #2
    *(pData + 0x4)  = key32[3];//#3
    *(pData + 0x5)  = key32[4];//#4
    *(pData + 0x6)  = key32[5];// #5
//-----------------------------------------------SENDING ENCRYPTED DATA
    *(pData + 0x7) = encrypt_in[0]; // data word in
    *(pData + 0x8) = encrypt_in[1]; // data word in
    *(pData + 0x9) = encrypt_in[2]; // data word in
    *(pData + 0xA) = encrypt_in[3]; // data word in
//-------------------------------------------------------------------

    decryptedText[0] = *(pData + 0xB);// store first set of 32-bit decrypted data
    decryptedText[0] |= ( ((uint64_t)(*(pData + 0xC))) << 32 );// shift the data then OR, to pack
    // inside the 64-bit integer
    decryptedText[1] = *(pData + 0xD); // store the third set of 32-bit of decrypted data
    decryptedText[1] |= ( ((uint64_t)(*(pData + 0xE))) << 32 );// shift the data then OR, to pack
    // inside the 64-bit integer

    unmap_physical(LW_virtual, LW_BRIDGE_SPAN);
    close_physical(fd);
```

The map_physical function is used to access physical memory [6]. Once this function is called it then calls the mmap kernel function to create a physical-to-virtual address. Data is then sent to the specified address in the VHDL implementation of “avs-s0-address”.

The figure above shows the main functionality of the “FPGA_decrypt()” function. This is how data and keys are sent and read from the FPGA.

References

1. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B. and Wingers, L. (2013). *The Simon and Speck Families of Lightweight Block Ciphers*. [online] National Security Agency. Available at: https://eprint.iacr.org/2013/404.pdf.

2. NANDLAND (n.d.). *For Loop - VHDL & Verilog Example*. [online] www.nandland.com. Available at: https://www.nandland.com/vhdl/examples/example-for-loop.html [Accessed 23 May 2022].

3. Intel ® Quartus ® Prime Standard Edition Handbook Volume 2 Design Implementation and Optimization. (2017). [online] Intel, pp.326–326. Available at: https://faculty-web.msoe.edu/johnsontimoj/Common/FILES/qts-qps-5v2_implementation_optimization_17.1.pdf [Accessed 23 May 2022].

4. klutt (2020). *Storing a String from the User of an Unknown Length*. [online] StackOverflow. Available at: https://stackoverflow.com/questions/16870485/how-can-i-read-an-input-string-of-unknown-length.

5. John (2020). *Using Procedures, Functions and Packages in VHDL*. [online] FPGA Tutorial. Available at: https://fpgatutorial.com/vhdl-function-procedure-package/.

6. Intel Corporation - FPGA University Program. (2019). *Using Linux* on DE-series Boards*. [online] Available at: https://ftp.intel.com/Public/Pub/fpgaup/pub/Teaching_Materials/current/Tutorials/Linux_On_DE_Series_Boards.pdf.