Hardware, High Level Synthesis, and Software Implementations of the Autoclave Cipher and the Porta Table Cipher

Arnaf Aziz Torongo   Kamrul Hasan   S M Habibul Mursaleen Chowdhury   Md. Muzibur Rahman

***Abstract*—Throughout history, Cryptographic algorithms have been utilized to preserve message confidentiality and integrity. This paper presents the hardware and software implementation of two old stream ciphers, the Autoclave Cipher and the Porta Table Cipher. These ciphers were counted efficient in all cases in earlier days. In this paper, their different implementations are compared in terms of usage of FPGA resources, development time, and implementation cost. The hardware designs are developed by using the Very High Speed Integrated Circuit Hardware Description Language (VHDL). Xilinx Vivado Suite 2021.1 is used for simulation, synthesis, and optimization of VHDL code, and Xilinx Vitis Suite 2021.1 is utilized for these ciphers IP and software implementations. The advantage of Basys 3 Artix-7 FPGA Board was taken for the hardware implementations, and Zybo: Zynq-7000 ARM/FPGA SoC Development Board was employed for the software and IP implementations. The results illustrate the efficiency of the implementations and the ciphers.**

***Index Terms*—Autoclave, porta table, FPGA, VHDL, encryption, decryption.**

I. Introduction

Cryptography played an essential role throughout history to protect data from being hacked. One of the cryptographic systems is secret key cryptography which can be categorized into either *block* or *stream* ciphers. Stream cipher streams byte by byte data to convert plaintexts into ciphertexts [1, pp.267]. This paper discusses the implementations of two symmetric key encryption algorithm. A single key is used for encipherment and decipherment in Symmetric key encryption [1, pp.86]. Both the Autoclave Cipher and Porta Table Cipher has simple structure and can perform encryption and decryption quickly.

There are multiple ways to implement the cipher algorithms, i.e., hardware, software or High Level Synthesis (HLS). Field Programmable Gate Array (FPGA) utilized for hardware implementation, and VHDL is used for implementation on FPGA devices. Besides, Vitis HLS is used to design, synthesize and co-simulate the HLS implementations programmed in programming language C, generate IP block, and export RTL. Furthermore, Vitis IDE is used for software implementation and, likewise HLS, C is used to program the implementation.

This paper focused on the implementation methods of Autoclave Cipher and Porta Table Cipher to illustrate the performance and cost metrics as well as the efficiency of the effectiveness of each cipher.

There are six more sections in this paper, as follows. The methodology is covered in section II; Hardware and Software implementations are described in sections III and IV, respectively. Section V comprises the Vitis HLS implementation, section VI consists of implementation results, and finally section VII presents the conclusion of the implementations.

II. Methodology

A. Porta Table Cipher

An Italian Giovanni Baptista della Porta has invented the Porta Table Cipher. The uppercase letters on the porta table create an agreed-upon key, with the letters indicating the alphabets chosen in order. The alphabet is placed in two lines on the right of each pair of keys [2, pp.38].

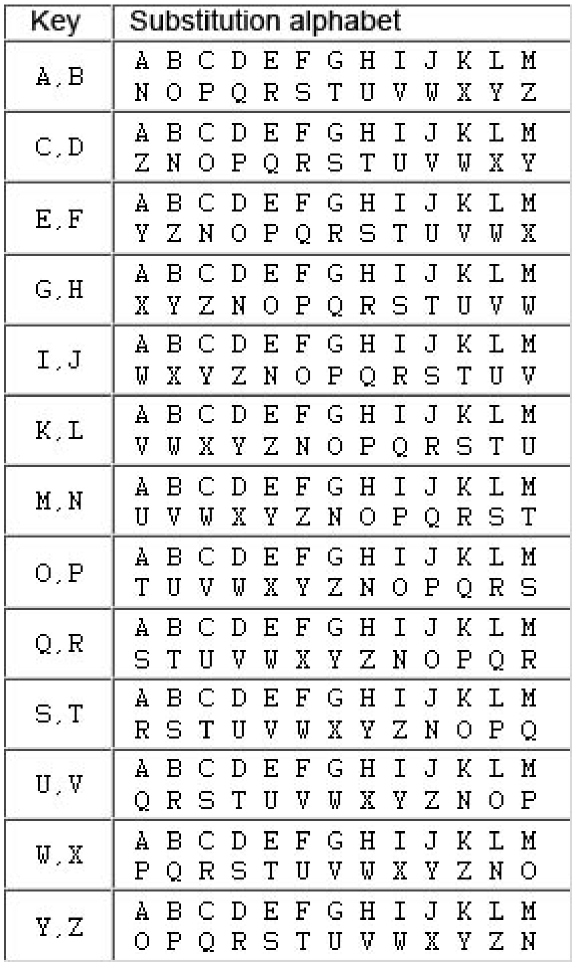
If the regular form of the message is *CHIEF LEAVES ON SUNDAY* and the key word is *MISTER* then the enciphered message which is called ciphertext, will be *WQZVQ QYWEVH JG JDJOSE*. The ciphertext can be deciphered in the same manner with the same key word to get back the regular form of the message called plaintext [2, pp. 38–41].

B. Autoclave Cipher

Autoclave Cipher is a symmetric poly-alphabetic substitution cipher. This is one of the Vigenère cipher variations, but Autoclave cipher uses a different key generation method. The Vigenère table uses one keyword to encrypt a plain text, whereas The Autoclave Cipher use one key letter [2, pp.42–43]. An initial key letter will encode the first letter of the plain text, and the first encrypted letter will be the next key letter for encoding the second letter of plain text and so on.

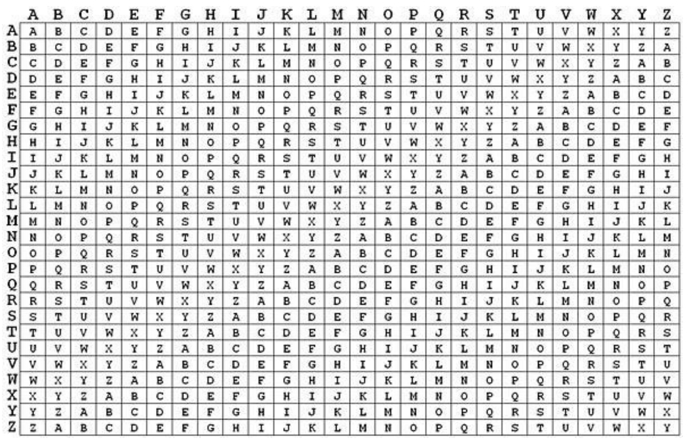
For instance, if the plain text *RETURN* needs to be encrypted and the initial key letter is *b* then the encoded ciphertext will be *SWPJAN*.

The decryption method is slightly different than the encryption method. Now again the initial key letter is same what have been used while encryption. In this case, again the initial key letter is *b* and now the ciphertext is *SWPJAN*. Similar to encryption the first letter *S* will be decrypted by initial key letter *b*. And then the next process is to decrypt the letter with its' previous given letter. For decrypting the first letter *S* the key letter will be initial key letter *b* and for *W* the key letter will be *S* and so on. Following this from the Vigenère table "Fig. 2", the plain text *RETURN* can be derived.



Key	Substitution alphabet
A,B	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
C,D	A B C D E F G H I J K L M Z N O P Q R S T U V W X Y
E,F	A B C D E F G H I J K L M Y Z N O P Q R S T U V W X
G,H	A B C D E F G H I J K L M X Y Z N O P Q R S T U V W
I,J	A B C D E F G H I J K L M W X Y Z N O P Q R S T U V
K,L	A B C D E F G H I J K L M V W X Y Z N O P Q R S T U
M,N	A B C D E F G H I J K L M U V W X Y Z N O P Q R S T
O,P	A B C D E F G H I J K L M T U V W X Y Z N O P Q R S
Q,R	A B C D E F G H I J K L M S T U V W X Y Z N O P Q R
S,T	A B C D E F G H I J K L M R S T U V W X Y Z N O P Q
U,V	A B C D E F G H I J K L M Q R S T U V W X Y Z N O P
W,X	A B C D E F G H I J K L M P Q R S T U V W X Y Z N O
Y,Z	A B C D E F G H I J K L M O P Q R S T U V W X Y Z N

Fig. 1. Porta Table



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Fig. 2. Vigenère table

III. Hardware Implementation

A. Porta Table Cipher

1) Design of FSMD Architecture and ASMD Chart: Several data units have been utilized to design Finite State Machine with Datapath(FSMD) architecture, i.e., UART, RAM, ROM, and counter. The FSMD is divided into control path (FSM) and data path. The data path shows the connection between data units, and the control path controls those data units. "Fig. 3" represents the FSMD of the Porta Table cipher. The input receives through UART_R and the output transmits through UART_T. A ROM has been introduced for storing the predefined key and a RAM has been used for storing the encrypted/decrypted message. To retrieve the key value from the ROM, counter A has been used. Also, another counter B has been used to generate the ram address to store the output of the cipher to the RAM. The input message goes through the cipher block followed by storing the cipher output into the RAM. UART_T then transmits the message stored in the RAM as the final encrypted/decrypted message.

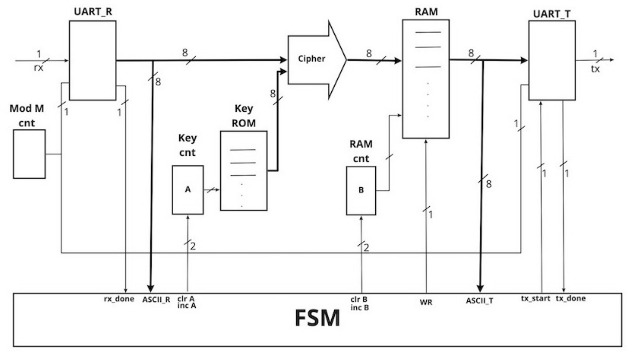


Fig. 3. Porta Table Cipher FSMD

In this Algorithmic State Machine with Datapath(ASMD) chart, four states have been proposed in the design. The first state **S0** is the reset state. In the second state **S1**, when the receiver completes receiving an input, the validation takes place comparing the ASCII code of the input. For valid ASCII, the write operation of RAM is enabled. To detect the end of the input, *Enter* keypress validation is placed afterward. If this condition fails, the program waits for the next input and so on. In contrast, if the condition succeeds, then the flow will go to the next state **S2**, and the transmission state will be started. In the next state **S3** the transmission will continue to take place until *tx_done* becomes true, and in such case, by incrementing counter B, the program will return to state **S2** to process the next input. This iteration continues till *ASCII_T = Enter* becomes true. Once *Enter* is pressed, the program state returns to the initial state **S0**.

2) Implementation: Hardware implementation was achieved as demonstrated in the FSMD table in "Fig. 3" and ASMD chart in "Fig. 4". Each of the key components is separated into its own file to construct a well-structured

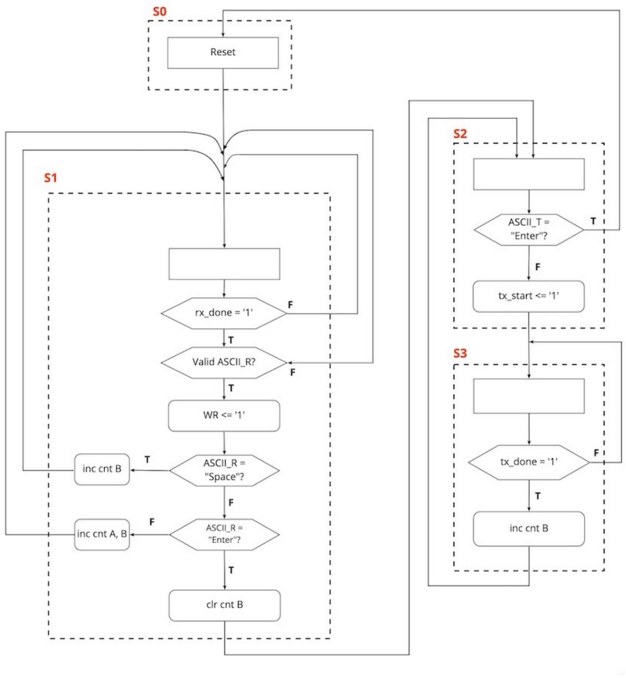


Fig. 4. Porta Table Cipher ASMD chart

VHDL design source. *fsm.vhd* is the FSM also known as the control path of the FSMD. The input-output control signals are handled from *fsm.vhd* component. *uart_rx.vhd* component receives serial data from the keyboard through the RS-232 system and converts it into parallel data for the receiving device. The number of input data is 8 and the sampling is configured 16 times of the baud rate clock to increase the maximum receiver tolerance to clock deviation [3, pp.164]. Another two vital components are *x_ram_async.vhd* and *x_rom_async.vhd* where the keys are stored in a rom component, and the ram component is utilized to store the output of the cipher before transmitting it as the final output. These two components are related to their corresponding counter components *cnt_ram.vhd* and *cnt_rom.vhd* respectively. The rom counter increment provides the address on the rom of the following key for the encryption/decryption operation. And the ram counter provides the ram address to write the next encrypted/decrypted output to the ram. The encryption and decryption takes place in the *cipher.vhd*. The 8-bit input and 8-bit key produces an 8-bit output. The top-level component is the *porta_top.vhd*, which holds the design together in a structured way.

A testbench file *porta_tb.vhd* has been constructed. It is used to verify the circuits' accuracy by inserting a predefined input into a stimuli process with a different clock period [3, pp.8]. The test vector generator block creates a testing input pattern using the constant as an input [3, pp.8]. In addition, as an indicated in the code, the value of each vector will last for 1ms. The ASCII hex code of the plain string ”chief” has been used successively in the test vector generator block to

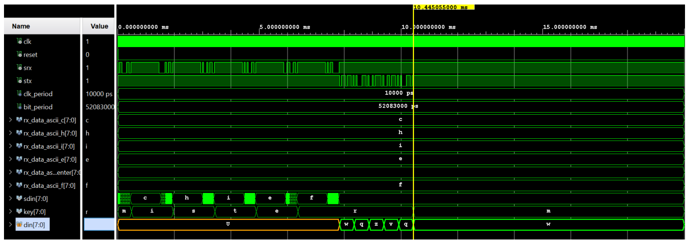


Fig. 5. Porta Table Cipher Testbench Waveform

construct the output waveform for encoding. “Fig. 5” shows the waveform of the behavioural simulation for the input defined in testbench. The input that has been defined in the test vector block is shown in the above waveform in the list of signals section. The signal *sdin* represents the plain text input signal towards the cipher block, while the signal *din* represents the ciphertext input to UART_T, which will eventually be the output. The signals *srx* and *stx* denote the receive and transmit operations, respectively. Because each test vector block has a 1ms life span, the clock value of the waveform can be used to visualize it. So, based on the wave shape, the cipher text for input ”chief” will be ”wqzvq” which proves that the design is valid.

```
library ieee;
use ieee.std_logic_1164.ALL;

entity porta_tb is
  -- Port ();
end porta_tb;

architecture arch of porta_tb is
constant clk_period : time := 10 ns;
constant bit_period : time := 52083 ns;

constant rx_data_ascii_c: std_logic_vector(7 downto
    0) := x"63"; -- receive c
constant rx_data_ascii_h: std_logic_vector(7 downto
    0) := x"68"; -- receive h
constant rx_data_ascii_i: std_logic_vector(7 downto
    0) := x"69"; -- receive i
constant rx_data_ascii_e: std_logic_vector(7 downto
    0) := x"65"; -- receive e
constant rx_data_ascii_f: std_logic_vector(7 downto
    0) := x"66"; -- receive f
constant rx_data_ascii_enter: std_logic_vector(7
    downto 0) := x"0D";

Component porta
Port ( reset , clk: in std_logic;
       rx:        in std_logic;
       tx:        out std_logic);
end Component;

signal clk, reset: std_logic;
signal srx, stx: std_logic;

begin
    uut: porta
    Port Map(clk => clk, reset => reset,
             rx => srx, tx => stx);
    clk_process: process
        begin
            clk <= '0';
            wait for clk_period/2;
            clk <= '1';
            wait for clk_period/2;
        end process;
    stim: process
        begin
        reset <= '1';
        wait for clk_period*2;
        reset <= '0';
        wait for clk_period*2;

            srx <= '0';
            wait for bit_period;
            for i in 0 to 7 loop
                srx <= rx_data_ascii_c(i);
                wait for bit_period;
            end loop;
            srx <= '1';
            wait for 1ms;

            srx <= '0';
            wait for bit_period;
            for i in 0 to 7 loop
                srx <= rx_data_ascii_h(i);
                wait for bit_period;
            end loop;
            srx <= '1';
            wait for 1ms;

            srx <= '0';
            wait for bit_period;
            for i in 0 to 7 loop
                srx <= rx_data_ascii_i(i);
                wait for bit_period;
            end loop;
            srx <= '1';
            wait for 1ms;

            srx <= '0';
            wait for bit_period;
            for i in 0 to 7 loop
                srx <= rx_data_ascii_e(i);
                wait for bit_period;
            end loop;
            srx <= '1';
            wait for 1ms;

            srx <= '0';
            wait for bit_period;
            for i in 0 to 7 loop
                srx <= rx_data_ascii_f(i);
                wait for bit_period;
            end loop;
            srx <= '1';
            wait for 1ms;

            srx <= '0';
            wait for bit_period;
            for i in 0 to 7 loop
                srx <= rx_data_ascii_enter(i);
                wait for bit_period;
            end loop;
            srx <= '1';
            wait;
        end process;
end arch;
```

Listing 1. Porta Table Testbench

B. Autoclave Cipher

1) Design of FSMD Architecture and ASMD Chart: Likewise Porta Table Cipher FSMD architecture, several resources has been utilized to construct the Autoclave Cipher FSMD architecture. However, some new data units have been introduced in this cipher that are not utilized in the Porta Table Cipher, i.e., Multiplexers, Registers. The initial key letter is stored in Register K, while the next key for encryption will be stored in register A. FSM and Cipher module has established a connection to determine whether the application is decryption or encryption.

MUX A has been used to control which key will go to the cipher unit during encryption by setting the mux control. In the initial state, the MUX A will deliver the output from Register K to the cipher as the initial key letter. Register A holds the value of cipher output in each state and when the MUX A sets to 1, the output of the register A becomes the key for the next encipher operation. The encryption and decryption mechanisms in the Autoclave cipher are distinct. As a result, a third register B is used for managing the key during decryption. The input value from UART_R goes towards the cipher as well as in the register B.

The output of Register K will be the initial key for the first letter decryption, whereas the value of register B from the UART_R input will be the key for subsequent decryption controlled by MUX B. The input value will be passed through the cipher block for both decryption and encryption, after which the cipher value will be saved in RAM and sent to the UART_T as output. Also, the counter A has been used to generate the ram address to store the output of the cipher to the RAM.

In this Algorithmic State Machine with Datapath(ASMD) chart, five states have been proposed in the design. The first state **S0** is the Reset. In this state, only Register K is loaded as the initial key will be needed for encrypt/decrypt the first input text. In the second state **S1**, when the receiver is done, then the validation of the input character will occur by validating the ASCII code. For a valid character, the write operation of RAM will be enabled. For every space input, the cipher operation will be skipped, and it will store directly to the RAM without any operation. To detect the end of the input, *Enter* keypress validation has been placed afterward. If this condition fails, the flow will go to the next state **S3**. State **S3** is almost identical to state **S2**, but the only difference is additionally set mux ctr A and set mux ctr B has been called in this state. The purpose of this additional state is to handle the dynamic key generation from the cipher operation. The **S2** state is responsible for processing only the initial key, while the **S3** state process the keys generated dynamically from the cipher output for the encryption mode and the ASCII input received from UART_R in the decryption mode. If the ASCII code is not *Enter* key, then it will wait to receive more input in state **S3**. If it is true, then the flow will go to the next state **S3**, and the transmission state will be started. In the next state **S4** the transmission will continue to happen until it is not done, and by incrementing the counter A, the value will be transmitted as output.

2) Implementation: Hardware implementation of Autoclave cipher was achieved as demonstrated in the FSMD table in “Fig. 6” and ASMD chart in “Fig. 7”. The *fsm.vhd* is

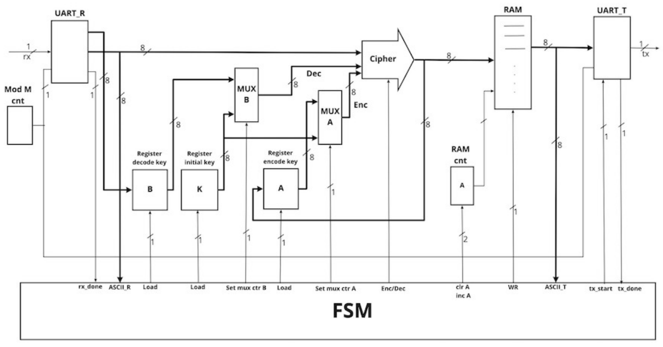


Fig. 6. Autoclave Cipher FSMD

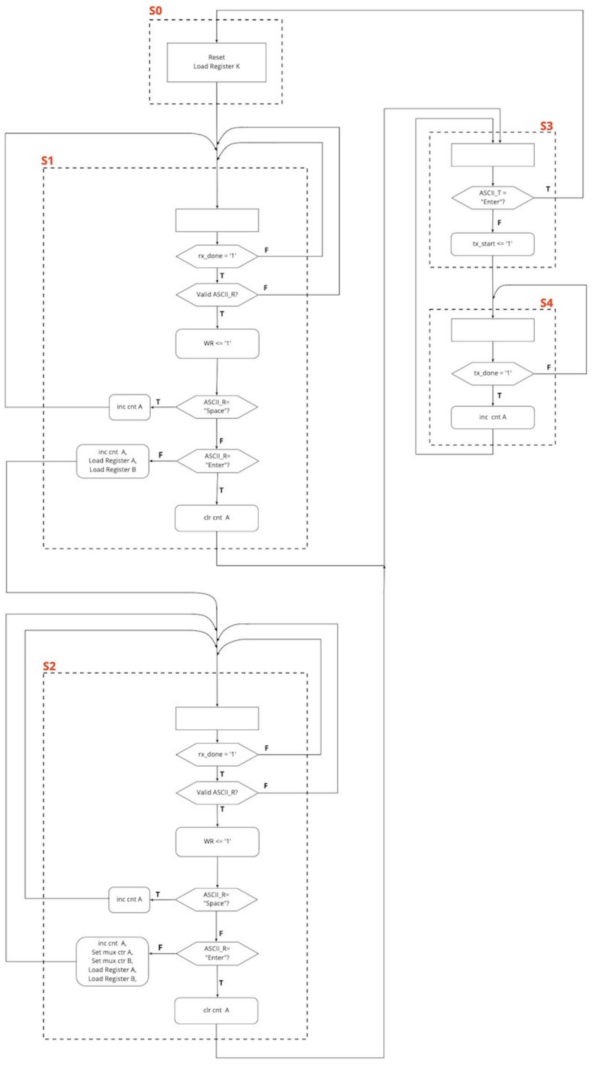


Fig. 7. Autoclave Cipher ASMD chart

the FSM as known as the control path of the FSMD. The data path controls are handled from *fsm.vhd*. The *uart_rx.vhd* component receives serial data from the keyboard through the RS-232 system and converts it into parallel data for the receiving device. Another vital component is the *ram.vhd*. The ram component has been utilized to store the output of the cipher before transmitting it to the system. The ram component is related to its' corresponding counter components *cnt_ram.vhd*. The counter is responsible for handling the read and write of values into ram. There are three registers in the FSMD; these registers are represented by *reg8bits.vhd*. These registers has been used to hold the initial key and the dynamic key generated at the time of encoding. The *cipher.vhd* component is where the encryption and decryption take place. During both encryption and decryption, the initial and dynamic keys to the cipher are handled by MUX, and this condition is defined in the top-level component. The top-level component is the *autoclave_top.vhd*, which holds the design together in a structured way. A testbench file *autoclave_tb.vhd* has been created to run the simulation. A *rom.vhd* and *cnt_rom.vhd* has also been included temporarily.

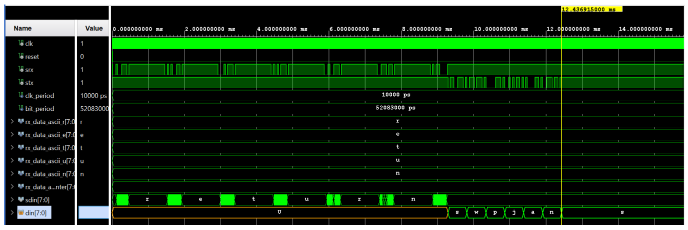


Fig. 8. Autoclave Cipher Testbench Encryption Waveform

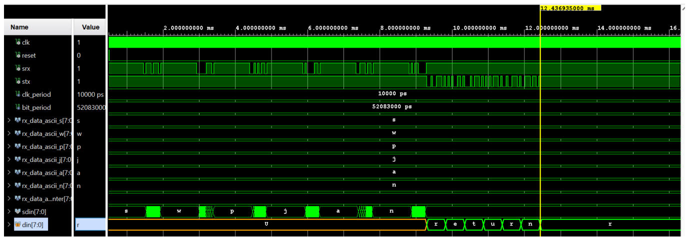


Fig. 9. Autoclave Cipher Testbench Decryption Waveform

The testbench file *autoclave_tb.vhd* to validate the correctness of the circuit by adding some defined input inside stimuli process with different clock period [3, pp.8]. The test vector generator block generates a testing input pattern with given constants as input [3, pp.8]. Also, the value of each vector will last for 1ms, as stated in the code. For encoding the ASCII hex code of the plaintext, ”return” has been used in the test vector generator block sequentially to generate the output waveform. ”Fig. 8” and ”Fig. 9” shows the waveform of the behavioural simulation for the input defined in testbench. In the above waveform in the list of signals section, the input has been defined in the test vector block. The signal *sdin* represents the input signal that enters into the cipher block as plain text, and the signal *din* is the input of UART_T as ciphertext, which will eventually be the output. The signal *srx* and *stx* represents the receive done and transmit done respectively. As each test vector block has 1ms of life time that can also be visualized by the clock value of the waveform. So from the waveform, it is evident that for input ”return” the ciphertext will be ”swpjan” which proves the design’s validation.

And for decryption, the ASCII hex code of the ciphertext ”swpjan” has been used in the test vector generator block sequentially to generate the output waveform. The signals in the waveform represent the same as above. And for the ciphertext ”swpjan” the design generates the output ”return” as expected.

IV. Software Implementation

1) Porta Table Cipher: The software implementation is completed using programming language C. The maximum text size of the plaintext and a hardcoded value for the cipher key and its length are defined in the software implementation. In the final software solution, the user can give the input as plain text to encrypt or an encrypted text to decrypt, and this program will return the encrypted/decrypted text generated with the help of the key. Through this process, multiple counter variables and encrypt/decrypt function are used to complete this cipher implementation. In this C program, both upper and lower case is handled for the cipher. If input text is uppercase [A-Z], 65 is subtracted to the input letter to fit it into 0-25 index for simplified calculation. If the input text is lowercase [a-z], 97 is subtracted. After handling the case sensitivity of the input the value of the keys also subtracted with 97 to make it lowercase. A equation for encryption and decryption of the Porta Table cipher has been used and developed for software implementation. The same formula works for both encryption and decryption as the encryption and decryption methods are the same for this cipher.

If the calculated position index of the message is less than 13 then,

$$cipher = 13 + (message - (k/2)) mod 13$$

other wise the equation will be

$$cipher = (message + (k/2)) mod 13$$

Here, message and key both are considered position index of respective character. e.g. a=0, b=1, z=25.

The main function is used to take user inputs and encrypt/decrypt the input based on user selection on the console. If user press ***e*** then it will encrypt and if user press ***d*** then the program will decrypt. If the user provides any invalid inputs, the program will detect and ask for a valid character to continue. After that, a single method is called with *inputText* and *convertedText* variable as arguments as the encryption and decryption method is same for this cipher as well. This method computes the encrypted or decrypted message and returns the output.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <ctype.h>

#define MAX_TEXT_SIZE 200
#define CIPHER_KEY "mistermistermistermister"
#define CIPHER_KEY_LEN 24

void encryptDecrypt(const char *inputText, char *
    outputText)
{
    char key[CIPHER_KEY_LEN] = CIPHER_KEY;
    int inputTextCounter, keyCounter = 0;
    int convertedText = 0;
    int messageToProcess = 0;

    for (inputTextCounter = 0; inputText[
    inputTextCounter] != '\0'; inputTextCounter++)
    {
        if (inputText[inputTextCounter] == ' ')
        {
            outputText[inputTextCounter] = ' ';
            continue;
        }
        if isupper (inputText[inputTextCounter])
        {
            messageToProcess = inputText[
    inputTextCounter] - 65;
        }
        else
        {
            messageToProcess = inputText[
    inputTextCounter] - 97;
        }
        int keyIndex = key[keyCounter] - 97;
        if (messageToProcess < 13)
        {
            int mod = 0;
            int difference = messageToProcess - (
    keyIndex / 2);
            convertedText = 13 + (difference + 13) %
     13;
        }
        else
        {
            convertedText = (messageToProcess + (
    keyIndex / 2)) % 13;
        }
        if isupper (inputText[inputTextCounter])
        {
            convertedText = convertedText + 'A';
        }
        else
        {
            convertedText = convertedText + 'a';
        }
        outputText[inputTextCounter] = (char)
    convertedText;
        keyCounter++;
        keyCounter = (keyCounter > CIPHER_KEY_LEN -
    1) ? 0 : keyCounter;
    }
    outputText[inputTextCounter] = '\0';
    return;
}
```

Listing 2. Porta Table Cipher Software Implementation

2) Autoclave Cipher: As done in Porta Table cipher, the software implementation is done in C programming for the Autoclave cipher as well. Firstly, the maximum text size is defined as 200. In the cipher solution, the user can provide the input to encrypt or decrypt and, the program will return the encrypted/decrypted text generated with the help of the key defined in the program. Multiple variables are used to keep track of input text characters, store ciphertext, calculate and store plain/cipher text character position index, and so on. *space* input from the user is handled during the process. Both upper and lower case inputs were taken care of for the cipher in this C program. A formula for encryption and decryption of the Autoclave cipher has been used and implemented for this software implementation purpose.

The encryption equation:

$$(Pi + Ki)mod26$$

Pi = Index of plaintext character at *i* position,
Ki = Index of key character at *i* position.

In the Autoclave cipher encryption, the ciphertext becomes the key for its' corresponding next plaintext. Hence, once encryption of plaintext is complete, the key is updated with the encrypted message. In this Autoclave cipher, encryption and decryption equations are different. In contrast to encryption in the decryption, the enchipered text becomes the key for decrypting after using the initial key once.

The decryption equation:

$$(Ci - Ki)mod26$$

Ci = Index of ciphertext character at *i* position,
Ki = Index of key character at *i* position.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <stdbool.h>

#define MAX_TEXT_SIZE 200

void encryptDecrypt(const bool isEncrypt, char *key,
     char *inputText, char *outputText)
{
    int inputTextIndex = 0;
    int processedText = 0;
    int messageToProcess = 0;
    int keyIndex = 0;

    for (inputTextIndex = 0; inputText[
    inputTextIndex] != '\0'; inputTextIndex++)
    {
        if (inputText[inputTextIndex] == ' ')
        {
            outputText[inputTextIndex] = ' ';
            continue;
        }

        if isupper (inputText[inputTextIndex])
        {
            messageToProcess = inputText[
    inputTextIndex] - 65;
        }
        else
        {
            messageToProcess = inputText[
    inputTextIndex] - 97;
        }

        if (key > 25)
        {
            keyIndex = key - 97;
        }
        else
        {
            keyIndex = key;
        }

        if (isEncrypt)
        {
            processedText = (messageToProcess +
    keyIndex) % 26;
            key = processedText;
        }
        else
        {
            processedText = (messageToProcess -
    keyIndex + 26) % 26;
            key = inputText[inputTextIndex];
        }
        if isupper (inputText[inputTextIndex])
        {
            processedText = processedText + 'A';
        }
        else
        {
            processedText = processedText + 'a';
        }
        outputText[inputTextIndex] = (char)
    processedText;
    }
    outputText[inputTextIndex] = '\0';
    return;
}
```

Listing 3. Autoclave Cipher Software Implementation

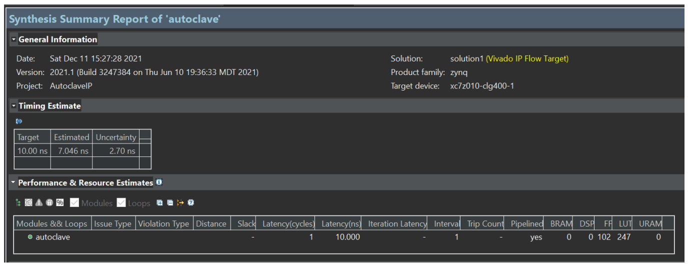
V. Vitis HLS Implementation

Vitis HLS makes it simple to write complicated FPGA-based algorithms with C/C++ code. It can easily share data with other IPs and supports sophisticated data formats and math algorithms [5]. HLS aims to extract concurrency from the input description and establish an FSMD architecture that is quicker and less expensive than merely running the input statement as a program on the hardware board [5]. A pipelined datapath and a cycle-by-cycle description of how data is routed through this datapath are included in the FSMD design. HLS produces RTL implementation, waveform, and information on performance, hardware costs, and other metrics to assist users better understand and enhance FSMD systems.

For the proper utilisation of Porta Table and Autoclave cipher, Vitis 2021.X, Viviado 2021.X, Vitis HLS 2021.X and Zybo Zynq-7000 development board were used. Whole process were divided into three inner section that explains further.

A. *High Level Synthesis in Vitis HLS*

A *Vitis HLS* project was formed with the *Zybo* board selected in the dialog box. After that, the projects' source and testbench files were built. During the implementation, functions with *pragma* were utilized in the source file to find the best solution for each cipher. Because the encryption and decryption of the Porta Table cipher are the same, only single method was utilized to encode and decode. However, since the encryption and decryption for the Autoclave cipher is different, multiple encode and decode methods were used. The top-level functions for *C Synthesis* and *C Simulation* were then defined to verify that the output looked as desired. A summary report has been prepared using the C Synthesis output, which revealed the implementation's performance and resources. (See Figure 10, 11).



Synthesis Summary Report of 'autoclave'

General Information

Date:	Sat Dec 11 15:27:28 2021	Solution:	solution1 (Vivado IP Flow Target)
Version:	2021.1 (Build 3247384 on Thu Jun 10 19:36:33 MDT 2021)	Product family:	zynq
Project:	AutoclaveIP	Target device:	xc7z010-clg400-1

Timing Estimate

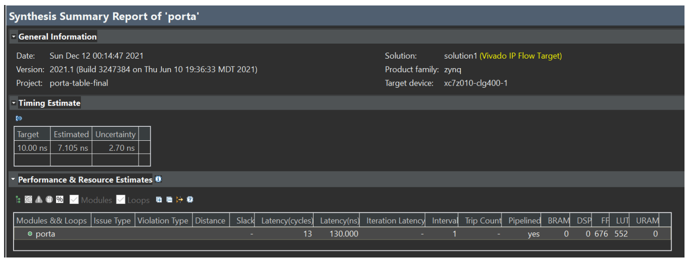
Target	Estimated	Uncertainty
10.00 ns	7.046 ns	2.70 ns

Performance & Resource Estimates

Modules Loops

Modules && Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
autoclave				-	1	10.000	-	1	-	yes	0	0	102	247	0

Fig. 10. Autoclave Cipher HLS Synthesis



Synthesis Summary Report of 'porta'

General Information

Date:	Sun Dec 12 00:14:47 2021	Solution:	solution1 (Vivado IP Flow Target)
Version:	2021.1 (Build 3247384 on Thu Jun 10 19:36:33 MDT 2021)	Product family:	zynq
Project:	porta-table-final	Target device:	xc7z010-clg400-1

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	7.105 ns	2.70 ns

Performance & Resource Estimates

Modules Loops

Modules && Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
porta				-	13	130.000	-	1	-	yes	0	0	676	552	0

Fig. 11. Porta Table Cipher HLS Synthesis

The code throughout this scenario has been optimized and has minimal latency. Afterwards, when the *RTL* file has been exported, then Cosimulation has been ran. Then it kept to *Verilog*, and set dump trace to all for the waveform (See Figure 12, 13) in Vivado.

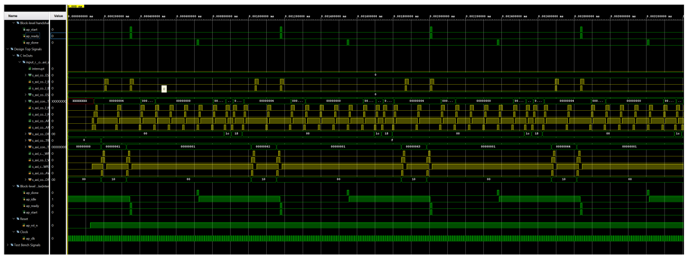


Fig. 12. Porta-Table Cipher HLS Waveform for Encode and Decode

B. Hardware Setup in Vivado

A Vivado project was constructed with a Zybo board for the hardware design before building a block design to add

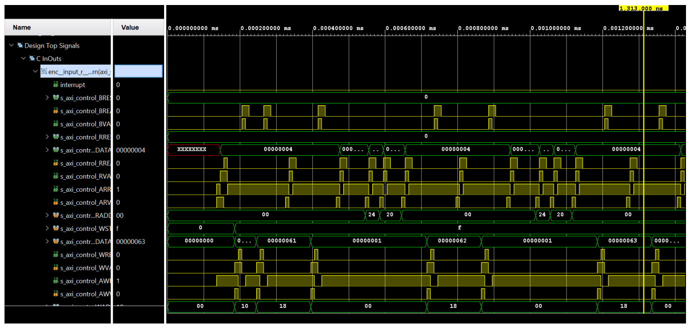


Fig. 13. Autoclave Cipher HLS Waveform

an IP for each IP implementation. In this procedure, the ZYNQ7 processing system has been used. The RTL file that is generated from the Vitis HLS and set in the Vivado repository under IP Catalog. The IP was integrated into the Block Design, resulting into a HLS IP Block Design that was later validated. A Bitstream file was generated and an HDL Wrapper was developed in vivado Hardware section. Hardware, including Bitstream files, were exported when they were generated as XSA.

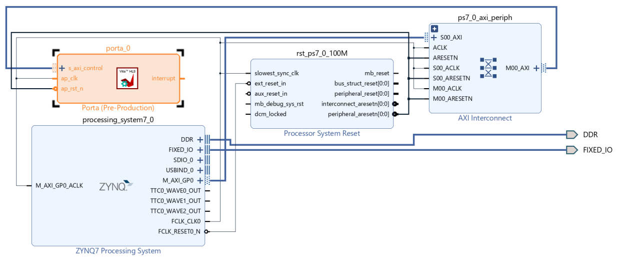


Fig. 14. Porta-Table Cipher HLS IP Block for Encode and Decode

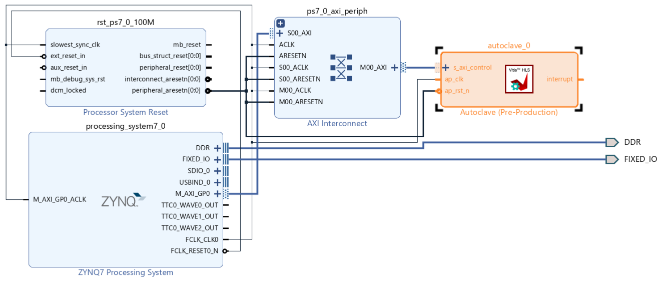


Fig. 15. Autoclave Cipher HLS IP Block for Encode and Decode

C. Application Setup in Vitis

The XSA file that has been developed in previous section and included in the Vitis IDE under Hardware Specification. Following the creation of the platform project, an application project was formed choosing the Zybo platform. Upon selecting *Hello World* template for the code, the application was built and run utilizing cipher codes. To interact with the program running on the Zybo board, a serial communication tool (screen, PuTTY) was deployed.

VI. Implementation Results

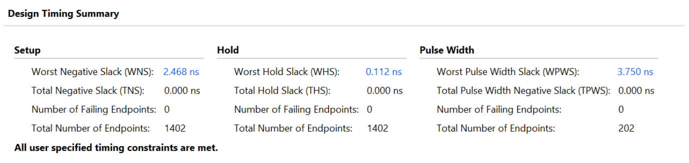
This paper is particularly focused on three different implementation technologies for each of the cipher methods. This is to present an overview of the implementation results and the comparison between these two cipher methods.

The synthesis results and FPGA resource utilization summary from the Hardware Implementation of the Autoclave Cipher and the Porta Table Cipher are presented in Table I. [1]

TABLE I
FPGA Resources Utilization Summary

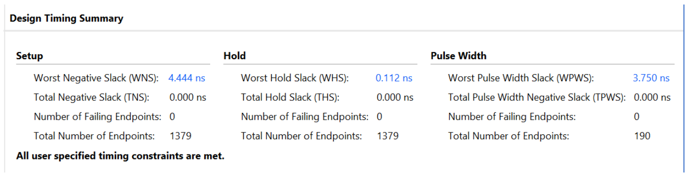
Resource	**Autoclave/ Utilization%**	**Porta Table/ Utilization%**	**Resource Available**
I/O	5 / 5%	4 / 4%	106
LUT	297 / 1%	299 / 1%	20800
FF	73 / 1%	61 / 1%	41600
LUTRAM	128 / 1%	128 / 1%	9600
Cells	15	13	-
Nets	186	152	-

The utilization percentage illustrates that both cipher methods consumed almost similar FPGA resources. However, it can also be seen that the Autoclave cipher used 5 I/O ports in contrast to 4 I/O ports in the Porta Table cipher. Apart from that, NETs usage also shows a significant difference. “Fig. 16” and “Fig. 17” show that both ciphers has met the timing constraints. Yet, the Autoclave cipher is marginally ahead of the Porta Table cipher in the design timing summary. Overall, the hardware implementations of both of the ciphers used somewhat similar resources.


```
Design Timing Summary

Setup                                      Hold                                          Pulse Width
  Worst Negative Slack (WNS):   2.468 ns     Worst Hold Slack (WHS):          0.112 ns     Worst Pulse Width Slack (WPWS):           3.750 ns
  Total Negative Slack (TNS):   0.000 ns     Total Hold Slack (THS):          0.000 ns     Total Pulse Width Negative Slack (TPWS):  0.000 ns
  Number of Failing Endpoints:  0            Number of Failing Endpoints:     0            Number of Failing Endpoints:              0
  Total Number of Endpoints:    1402         Total Number of Endpoints:       1402         Total Number of Endpoints:                202
All user specified timing constraints are met.
```

Fig. 16. Autoclave Cipher Timing Summary


```
Design Timing Summary

Setup                                      Hold                                          Pulse Width
  Worst Negative Slack (WNS):   4.444 ns     Worst Hold Slack (WHS):          0.112 ns     Worst Pulse Width Slack (WPWS):           3.750 ns
  Total Negative Slack (TNS):   0.000 ns     Total Hold Slack (THS):          0.000 ns     Total Pulse Width Negative Slack (TPWS):  0.000 ns
  Number of Failing Endpoints:  0            Number of Failing Endpoints:     0            Number of Failing Endpoints:              0
  Total Number of Endpoints:    1379         Total Number of Endpoints:       1379         Total Number of Endpoints:                190
All user specified timing constraints are met.
```

Fig. 17. Porta Table Cipher Timing Summary

The Vitis HLS implementation synthesis results are shown in Table II. The estimated timing and uncertainty are quite close in both of the ciphers. However, the latency is significantly higher in the Porta Table cipher compared to the Autoclave Cipher. Besides, Autoclave cipher used 102 flip-flops (FF) and 247 look-up tables (LUT), whereas the latter used 676 flip-flops and 552 look-up tables. This is a much bigger difference than what has been seen in the hardware implementations.

[1]LUT - Look up table, FF - Flip-flop

TABLE II
Vitis HLS Resources Utilization Summary

Resource	**Autoclave**	**Porta Table**
Target Timing	10ns	10ns
Estimated Timing	7.046ns	7.105ns
Uncertainty	2.7ns	2.7ns
Latency(Cycles)	1	13
Latency(ns)	10ns	130ns
Interval	1	1
FF	102	676
LUT	247	552

This discussion also presents the resources consumed by the hardware implementation of both ciphers consumed notably less than their corresponding HLS implementations. Usage of the resources impacts the development costs. However, development time also impacts the costs. Hardware implementations take much longer time and effort than the Vitis HLS or software implementations. The solutions can be programmed using C/C++ in the Vitis HLS or software implementation through Vitis IDE, and the process is straightforward. The extra work of manually handling the register level operations can be skipped, and designing the FSMD or ASMD is no longer required. Moreover, thousands of lines of VHDL codes can be compressed into a much smaller codebase [4]. Vitis HLS also supports exporting the RTL of the solution that can be programmed into hardware. The prototyping is also significantly faster in Vitis HLS and software implementation.

According to the metrics discussed, the Autoclave Cipher lagged behind the Porta Table cipher in hardware implementation. But, in Vitis HLS implementation Autoclave Cipher performed significantly better than the Porta Table Cipher. Apart from that, Autoclave Cipher uses a single key letter to encrypt the initial letter of the message. Then, that enciphered letter becomes that key letter for the following letter to be enciphered. The decryption also follows the same procedure, but the enciphered message letters become key for decryption. This makes the Autoclave Cipher considerably more secure than the Porta Table Cipher, which uses a keyword for encryption and decryption.

VII. Conclusion

The Autoclave Cipher and the Porta Table Cipher were implemented using the Basys 3 Artix-7 FPGA Board and Zybo: Zynq-7000 ARM/FPGA SoC Development Board for the hardware implementations and the software and IP implementations, respectively. An analysis of the implementation results and comparison has been discussed using the above-mentioned implementation technologies. For both ciphers, RTL implementations, graphical wave-forms, and reports on efficiency, development costs, and other metrics have been proposed to help users quickly identify and better manage the system. The FSMD design includes a multi-threaded controller and a cycle-by-cycle timing description for every state transition of how data is transmitted through the controller. The implementation discussion draws the conclusion predicating a superior, efficient, and cost-effective cipher.

References

[1] S. William, Cryptography and Network Security Principles and Practice, 7th Edition, Pearson Education, 2017.

[2] L. John, Codes and ciphers : secret writing through the ages, London ; New York : Abelard-Schuman, 1964.

[3] P. Chu Pong, FPGA PROTOTYPING BY VHDL EXAMPLES, London ; John Wiley & Sons, Inc,. Publication, 2008.

[4] Group 3, Hardware, High Level Synthesis, and Software Implementations of the Autoclave Cipher and the Porta Table Cipher, 2021, GitHub, https://github.com/iTorongo/CS4110

[5] ÓscarLucía, EricMonmasson† DenisNavarro, Luis A.Barragán, IsidroUrriza, José I.Artigas, Control of Power Electronic Converters and Systems, Volume 2, P2018, Pages 477-502.

Appendix

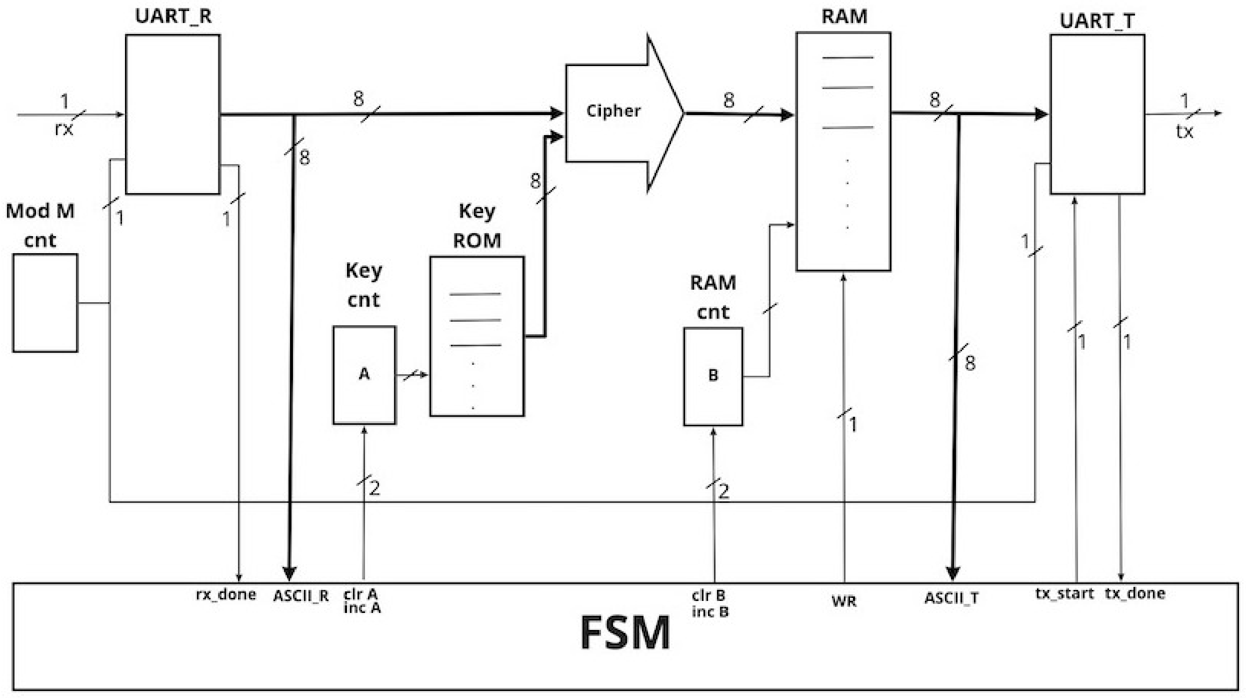


Fig. 18. Porta table cipher FSMD

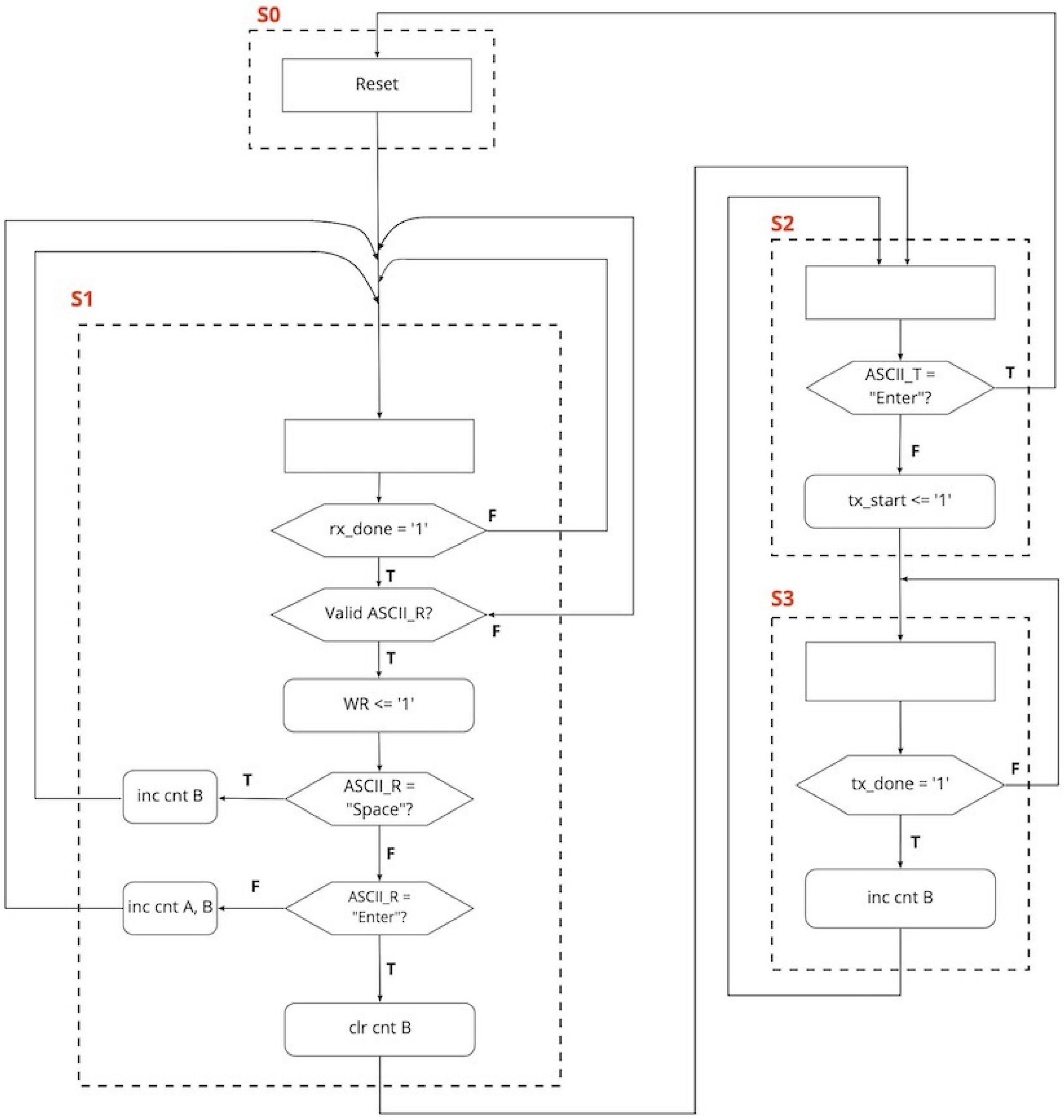


Fig. 19. Porta-table cipher ASMD chart

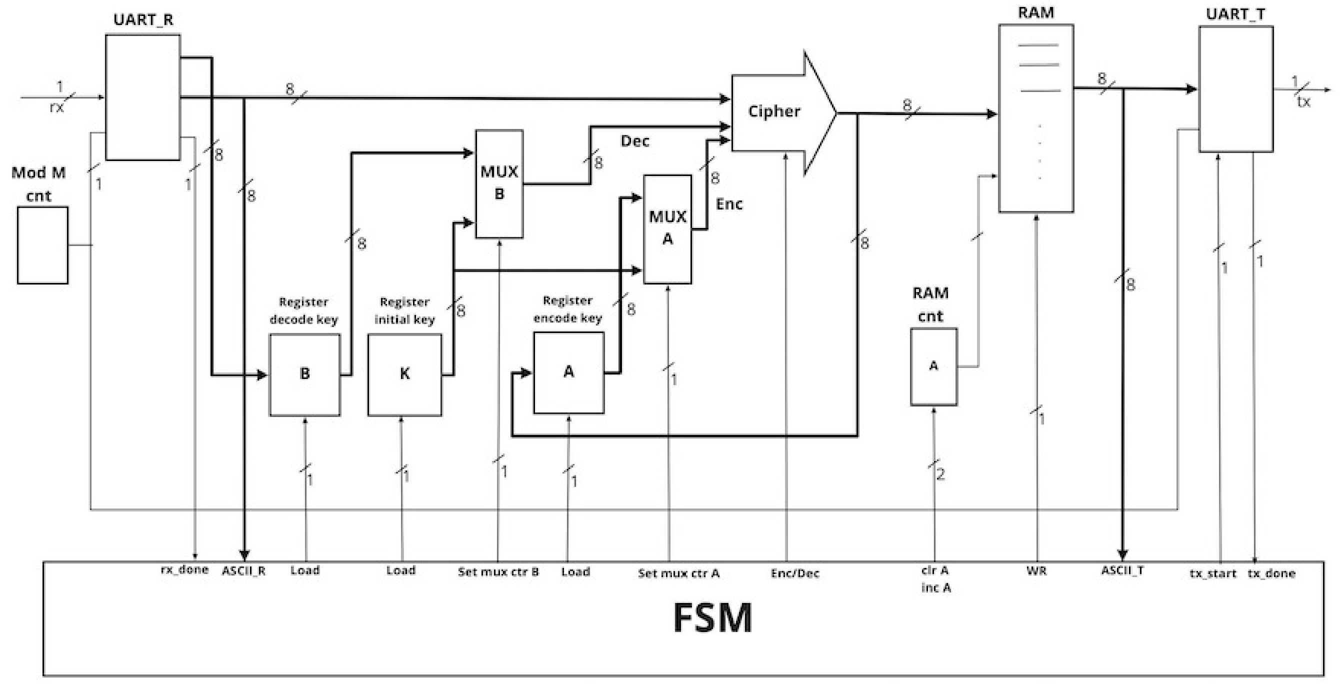


Fig. 20. Autoclave cipher FSMD

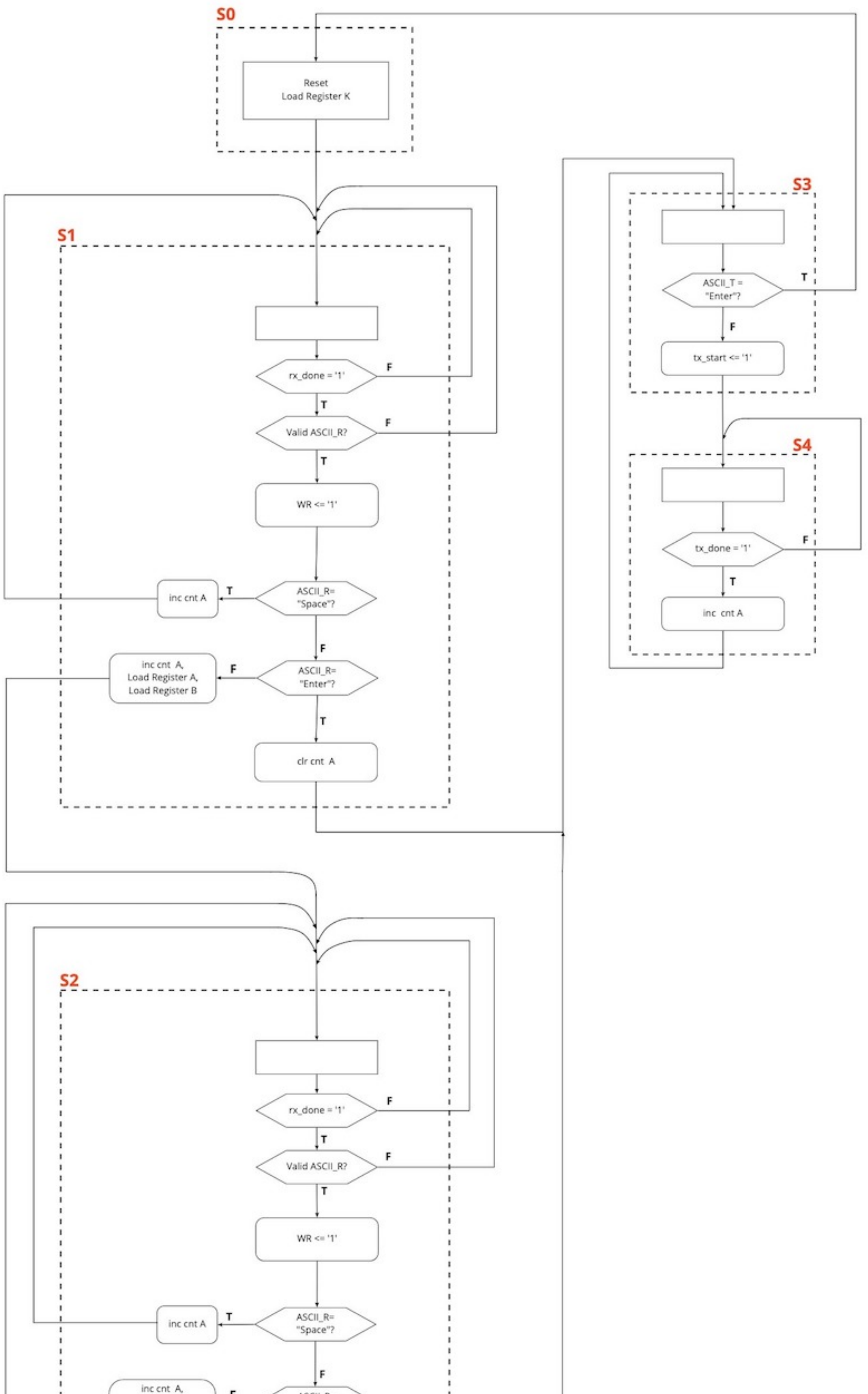
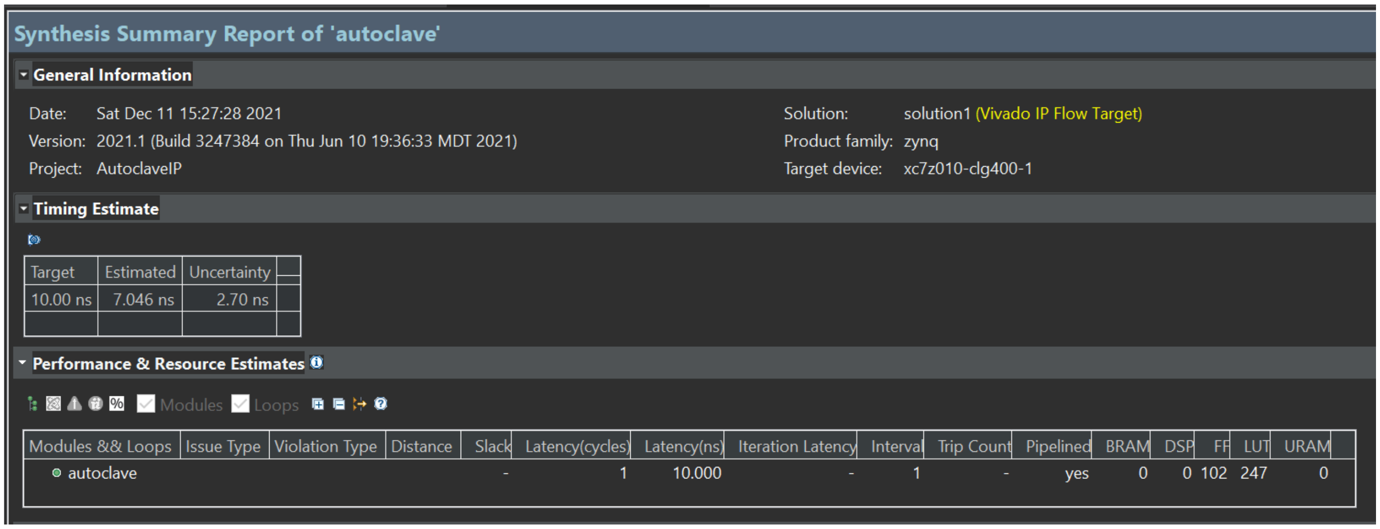


Fig. 21. Autoclave cipher ASMD chart


Synthesis Summary Report of 'autoclave'

General Information

Date: Sat Dec 11 15:27:28 2021
Version: 2021.1 (Build 3247384 on Thu Jun 10 19:36:33 MDT 2021)
Project: AutoclaveIP
Solution: solution1 (Vivado IP Flow Target)
Product family: zynq
Target device: xc7z010-clg400-1

Timing Estimate

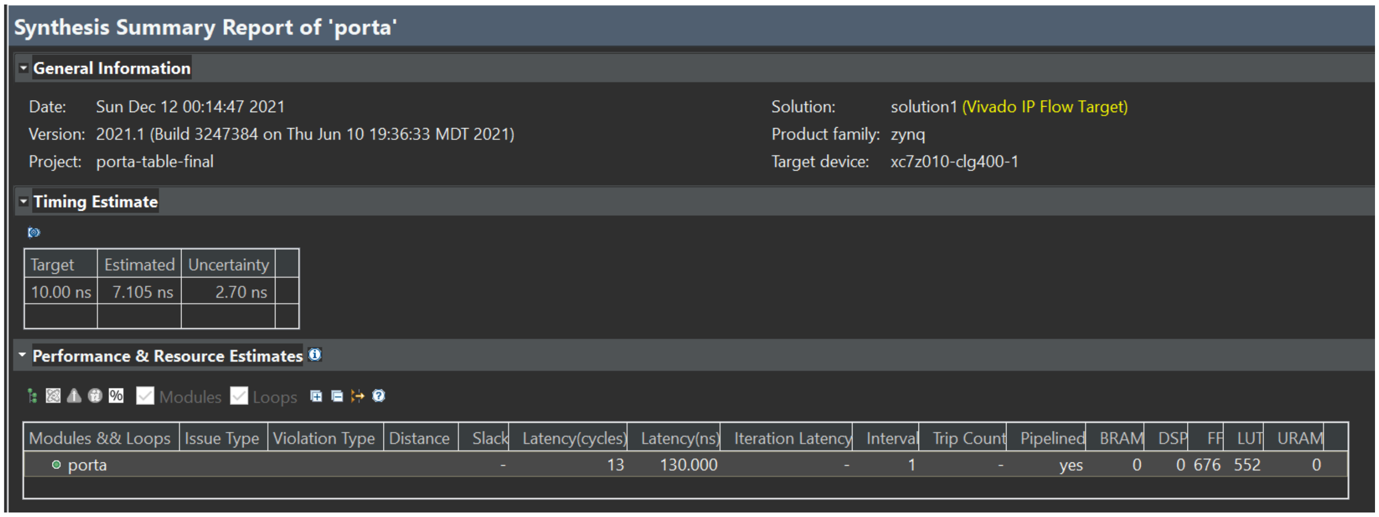
Target	Estimated	Uncertainty
10.00 ns	7.046 ns	2.70 ns

Performance & Resource Estimates

Modules Loops

Modules && Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
autoclave				-	1	10.000	-	1	-	yes	0	0	102	247	0

Fig. 22. Autoclave Cipher HLS Synthesis for Encode and Decode


Synthesis Summary Report of 'porta'

General Information

Date: Sun Dec 12 00:14:47 2021
Version: 2021.1 (Build 3247384 on Thu Jun 10 19:36:33 MDT 2021)
Project: porta-table-final
Solution: solution1 (Vivado IP Flow Target)
Product family: zynq
Target device: xc7z010-clg400-1

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	7.105 ns	2.70 ns

Performance & Resource Estimates

Modules Loops

Modules && Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
porta				-	13	130.000	-	1	-	yes	0	0	676	552	0

Fig. 23. Porta-table Cipher HLS Synthesis for Encode and Decode

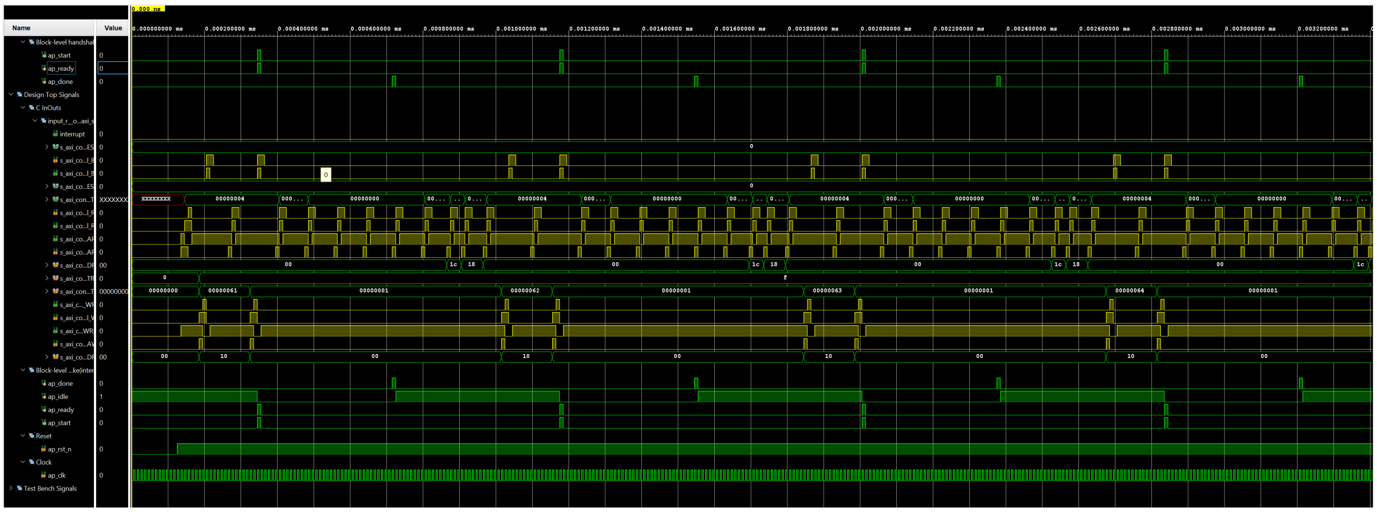
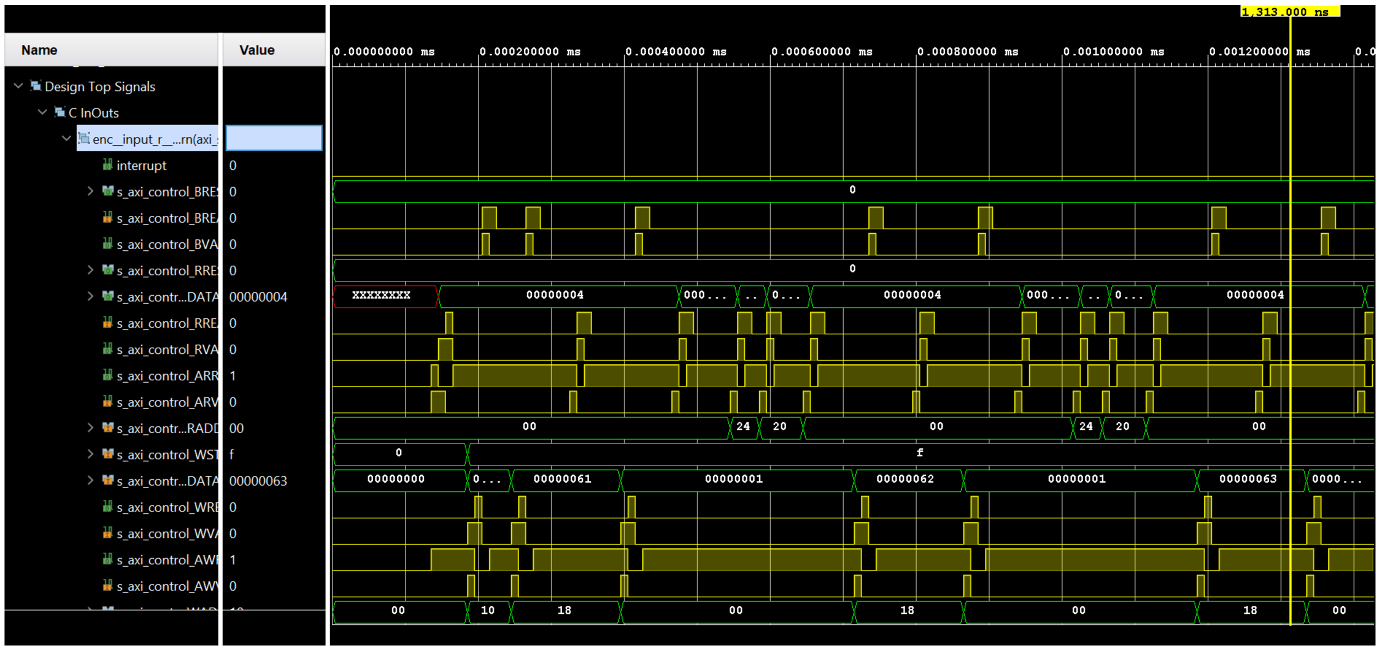


Fig. 24. Porta-table Cipher HLS Waveform for Encode and Decode



Name	Value
Design Top Signals	
C InOuts	
enc_input_r__...rn(axi_	
interrupt	0
s_axi_control_BRE	0
s_axi_control_BRE	0
s_axi_control_BVA	0
s_axi_control_RRE	0
s_axi_contr...DATA	00000004
s_axi_control_RRE	0
s_axi_control_RVA	0
s_axi_control_ARR	1
s_axi_control_ARV	0
s_axi_contr...RADD	00
s_axi_control_WST	f
s_axi_contr...DATA	00000063
s_axi_control_WRE	0
s_axi_control_WVA	0
s_axi_control_AWR	1
s_axi_control_AWV	0

Fig. 25. Autoclave Cipher HLS Waveform

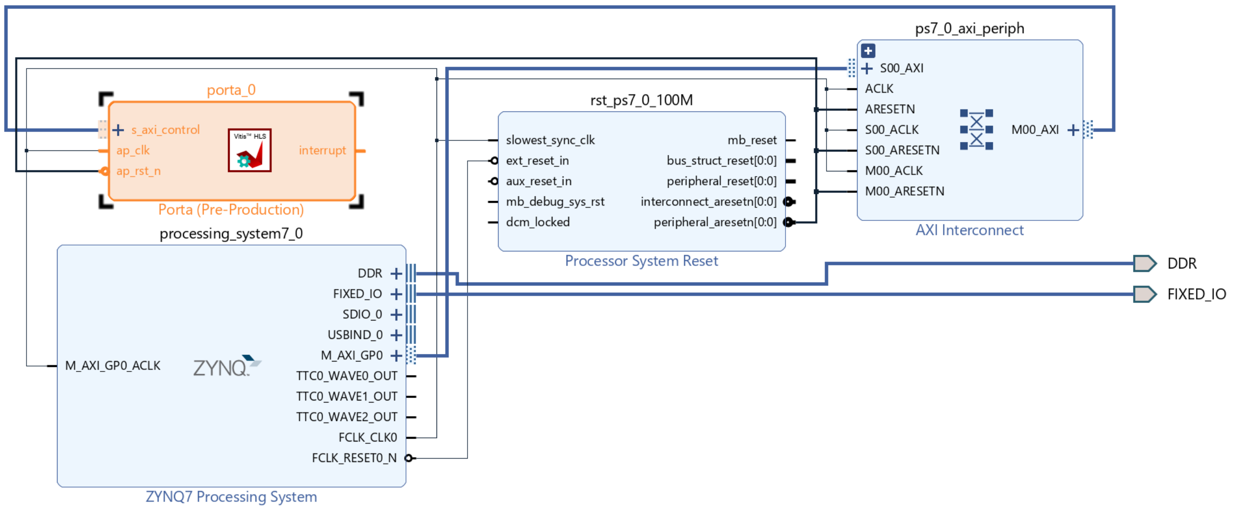


Fig. 26. Porta-table Cipher HLS IP Block for Encode and Decode

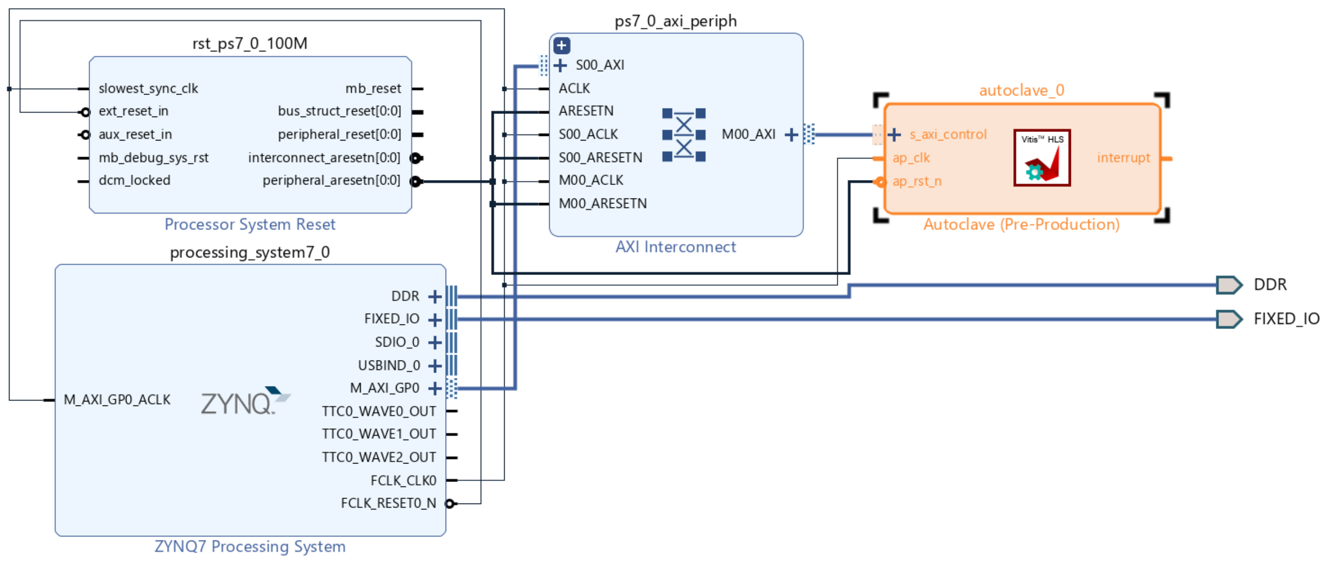
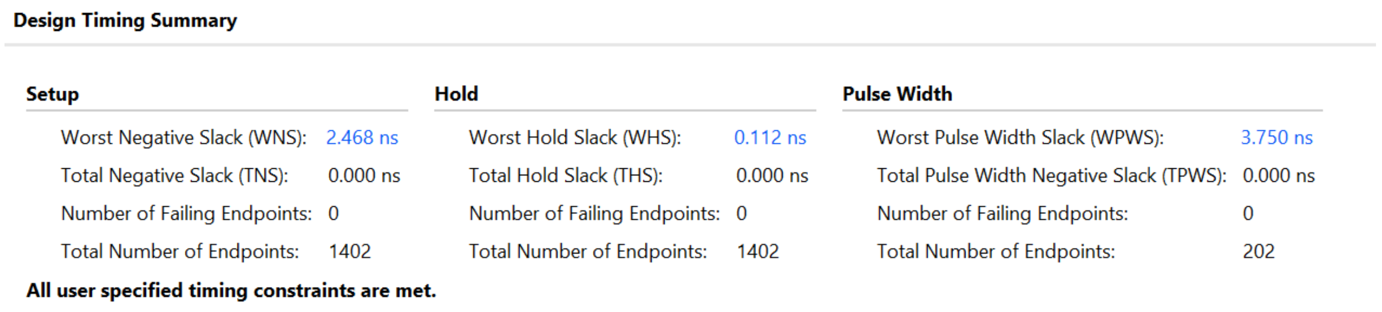


Fig. 27. Autoclave Cipher HLS IP Block for Encode and Decode

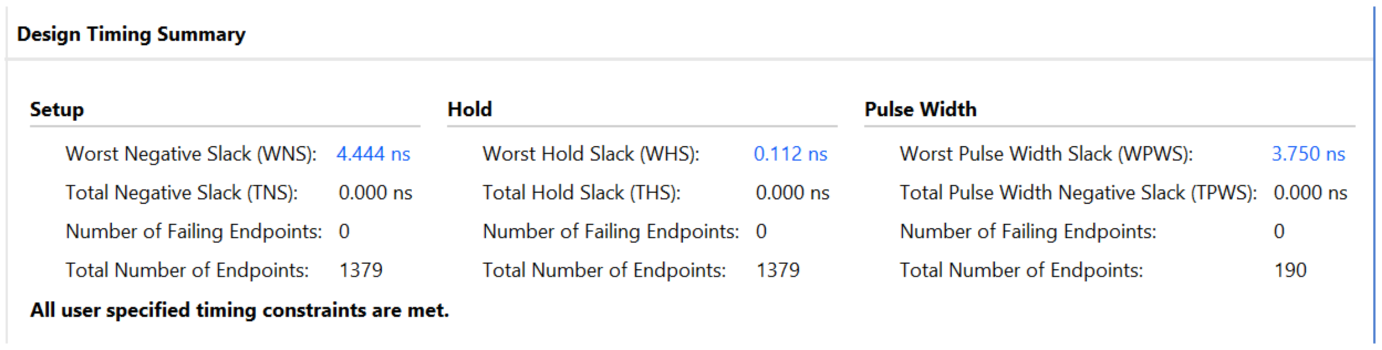


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	2.468 ns	Worst Hold Slack (WHS):	0.112 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1402	Total Number of Endpoints:	1402	Total Number of Endpoints:	202

All user specified timing constraints are met.

Fig. 28. Autoclave Cipher Timing Summary



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	4.444 ns	Worst Hold Slack (WHS):	0.112 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1379	Total Number of Endpoints:	1379	Total Number of Endpoints:	190

All user specified timing constraints are met.

Fig. 29. Porta Table Cipher Timing Summary