8-2020

Improving Hardware Implementation of Cryptographic AES Algorithm and the Block Cipher Modes of Operation

Chu-Wen Cheng
The University of Texas Rio Grande Valley

IMPROVING HARDWARE IMPLEMENTATION OF CRYPTOGRAPHIC AES ALGORITHM AND THE BLOCK CIPHER MODES OF OPERATION

A Thesis

by

CHU-WEN CHENG

Submitted to the Graduate College of
The University of Texas Rio Grande Valley
In partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE IN ENGINEERING

August 2020

Major Subject: ELECTRICAL ENGINEERING

IMPROVING HARDWARE IMPLEMENTATION OF CRYPTOGRAPHIC AES ALGORITHM AND THE BLOCK CIPHER MODES OF OPERATION

A Thesis
by
Chu-Wen Cheng

COMMITTEE MEMBERS

Dr. Sanjeev Kumar
Chair of Committee

Dr. Weidong Kuang
Committee Member

Dr. Wenjie Dong
Committee Member

August 2020

ABSTRACT

Cheng, Chu-Wen, Improving Hardware Implementation Of Cryptographic AES Algorithm And The Block Cipher Modes Of Operation. Master of Science in Engineering (MSE), August 2020, 149 pp., 19 tables, 127 figures, 45 references.

With ever increasing Internet traffic, more business and financial transactions are being conducted online. This is even more so during these days of COVID-19 pandemic when traditional businesses such as traditional face to face educational systems have gone online requiring huge amount of data being exchanged over Internet. Increase in the volume of data sent over the Internet has also increased the security vulnerabilities such as challenging the confidentiality of data being sent over the Internet. Due to sheer volume, all data will need to be effectively encrypted. Due to increase in the volume of data, it is also important to have encryption/decryption functions to work at a higher speed to maintain the confidentiality of sensitive data.

In this thesis, our goal is to enhance the hardware speed of encryption process of the standard AES scheme and its four variants such as AES-128, AES-192, AES-256 and new AES-512 and implement such functions on an FPGA. We also consider the FPGA implementation of different modes of AES operation.

By employing parallelism and pipelining approach, we attempt to speed up various computational components of AES implementations using the Quartus II onto Intel's FPGA. This approach shows improvement in the response speed, data throughput and latency.

DEDICATION

The contribution and dedication of many people is hidden behind my small achievement. I want to dedicate this thesis to

My parents

For supporting and encouraging me to believe in myself.

My wife

For taking care of me and kids' need in every way and letting me concentrate on my study.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to all committee members, Professor Sanjeev Kumar, Professor Weidong Kuang, and Professor Wenjie Dong, for their support. Special thanks to Dr. Kumar, Chair of Committee, for his patience, advisement, motivation, and immense knowledge in my studies which aided in accomplishing this thesis. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my master study.

Besides my advisor, I would like to thank the rest of my thesis committee: Professor Weidong Kuang, and Professor Wenjie Dong, for their insightful comments and encouragement. I thank my fellow colleague in networking lab for the stimulating discussions, for the time we were working together on conference or journal paper and in our group meeting. Their friendships are what I hope to keep forever even after I graduate.

Most importantly, none of this could have happened without my family. My parents have been kind and supportive to me over the last several years and I truly appreciate my wonderful wife taking great care of my kids while I am in school. This dissertation stands as a testament to their unconditional love and encouragement.

The support for the research in this thesis is provided in part by the grant awarded to Dr. Sanjeev Kumar by the National Science Foundation (NSF), Houston Endowment Chair in Science, Math, and Technology Fellowship, and Lloyd Benston Jr. Endowment Fellowship.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER I
INTRODUCTION

With today's development and expansion of networks and internet-connected devices, information security is an issue of increasing concern. Cryptography plays a key role in information security by providing confidentiality when transmitting data. It prevents unauthorized access so that data is not disturbed. With the emergence and rapid growth of cloud computing, current encryption technologies are often threatened making it important to study the characteristics of existing algorithms to match this advancement. Confidentiality is one of the focuses in network security for digital communication systems, where large data blocks go through a cryptographic algorithm with a cipher key that increases the security and complexity of the output ciphertext. For the past several years, multiple security algorithms have been developed as standard to be utilized in the data encryption process, such as the Data Encryption Standard (DES), Triple Data Encryption Standard (3DES), and the current one, designated by the U.S. National Institute of Standards and Technology (NIST), the Advanced Encryption Standard (AES).

AES, also known as Rijndael algorithm, is a symmetric encryption algorithm that has a minimum input data block size of 128-bits which undergo a series of permutations, substitutions, and digital logic operations over several rounds. Encryption algorithms are always improving on ciphertext complexity, required hardware storage allocation, and execution time. Field Programmable Gate Arrays (FPGA's) are a hardware alternative for encryption algorithm

implementation because, although the logic units in it are fixed, the functions and interconnections between them are based on the user's design which allow for improvement in performance and speed. The research presented in this thesis focuses on improving performance by analyzing the AES algorithm for efficient implementations, on an Altera Cyclone IV FPGA using the Intel Quartus II software and Verilog Hardware Description Language.

1.1 Problem Statement

With the emergence of high-performance cloud computing and increase in traditional businesses such as shopping and education moving completely on Internet during these days of pandemic, it is important to keep confidential data safe. Due to huge volume of data being exchanged over Internet, it is even more important for the security encryption schemes to efficiently utilize hardware implementations to perform at much higher speeds to provide fast, efficient, and secure data transmissions. There have been several AES hardware implementations related work done in literature [8-44]. Many literatures have proposed Mix Column Computation and S-box hardware lookup table implementations however none of the prior work utilized implementation on newer Intel's Cyclone IV FPGA involving parallelism and pipelining together. In this thesis, we hypothesize that various AES components can be made faster by utilizing parallelism and pipelining in their computation via FPGA implementations thus improving the overall speed of AES encryption process.

Some of our published work showed processing speed improvement for components such as AES Mix Columns module [28] and S-box module [45] where parallelism and pipelining in the computation were utilized to improve overall AES performance of those functions. In this

thesis, we considered the FPGA implementation for the entire AES algorithm based on improvement on these components. This thesis also presents high speed, fully pipelined FPGA implementation of AES Encryption in two different operation modes. Furthermore, we also evaluated the overall performance for the four variants of AES such as AES-128, AES-192, AES-256, and AES-512. The work in this thesis aims to speed up overall AES encryption by reducing processing delays and optimize silicon area for such implementations. Comparisons are conducted on both a theoretical basis and through timing simulations on the Intel Quartus II software to reveal the implication of increased complexity on the hardware performance of AES. Our work in this thesis involves Writing Verilog Code for design and verification of digital circuit and Simulating the code on "Quartus II".

1.2 Symmetric Encryption Algorithm- AES Algorithm

1.2.1 Background

The Advanced Encryption Standard (AES), also known as Rijndael, is an electronic data encryption specification established by the National Institute of Standards and Technology (NIST) in 2001 [1]. AES is a subset of the Rijndael block cipher. Developed by two Belgian cryptographers Vincent Rijmen and Joan Daemen, they submitted a proposal to NIST during the AES selection process. Rijndael is a series of passwords and block size passwords.

In the United States, AES was announced by the National Institute of Standards and Technology on November 26, 2001 as the US FIPS PUB 197 (FIPS 197) and is now used worldwide. It replaces the Data Encryption Standard (DES) released in 1977. The algorithm

described by AES is a symmetric key algorithm, meaning that the same key is used to encrypt and decrypt data.

Figure.1 shows the overall structure of the AES encryption process. The cipher takes a plaintext block size of 128 bits, or 16 bytes. The key length can be 16, 24, 32, or 64 bytes (128, 192, 256, or 512 bits). The algorithm is referred to as AES-128, AES-192, AES-256, or AES-512, depending on the key length. The input to the encryption and decryption algorithms is a single 128-bit block.

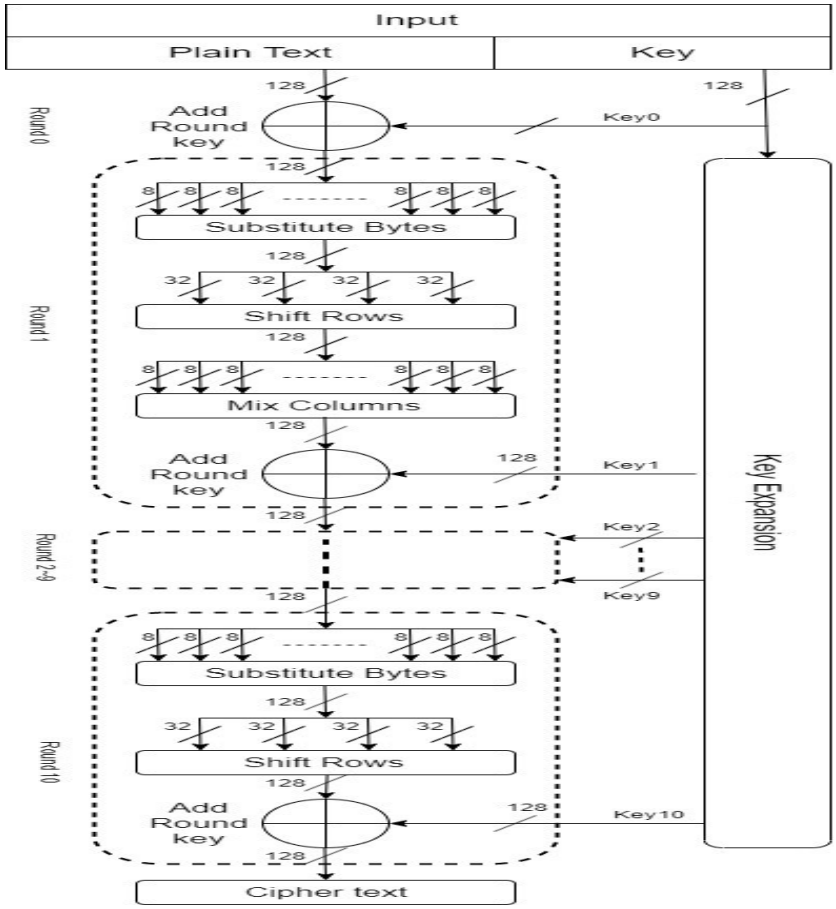

Figure 1: AES-128 Structure

The cipher consists of rounds, where the number of rounds depends on the key length: 10 rounds for a 16-byte key, 12 rounds for a 24-byte key, 14 rounds for a 32-byte key, 16 rounds for a 64-byte key (Table 1).

Table 1: AES Parameters

Key Size (words/bytes/bits)	4/16/128	6/24/192	8/32/256	16/64/256
Plaintext Block Size (words/bytes/bits)	4/16/128	4/16/128	4/16/128	4/16/128
Number of Rounds	10	12	14	16
Round Key Size (words/bytes/bits)	4/16/128	4/16/128	4/16/128	4/16/128
Expanded Key Size (words/bytes)	44/176	52/208	60/240	68/272

The overall data computation [1], [2] to obtain Rijndael cipher consists of

1. An initial "Add Round key' step to add obscurity
2. 9/11//13/15 rounds of 4 steps to adds confusion, diffusion, non-linearity
    - Substitute bytes: Uses an S-box to perform a byte-by-byte substitution of the block
    - Shift-Rows: A simple permutation

- Mix-Columns: A substitution that makes use of arithmetic over GF (2^8)
- Add-Round-Key: A simple bitwise XOR of the current block with a portion of the expanded key

3. A Final 10th/12th/14th/16th step of Substitute bytes, ShiftRows, and AddRoundKey to add obscurity

1.2.2 Substitute bytes

The forward substitute byte transformation, called SubBytes, is a simple table lookup (Figure 2). AES defines a matrix of byte values, called an S-box (Table 2), that contains a permutation of all possible 256 8-bit values. Each individual byte of State is mapped into a new byte in the following way: The leftmost 4 bits of the byte are used as a row value and the rightmost 4 bits are used as a column value. These row and column values serve as indexes into the S-box to select a unique 8-bit output value.

In this layer, each byte in the state will be substituted by values obtained from substitution boxes. This is done to achieve more security according to diffusion-confusion Shannon's principles for cryptographic algorithms design.

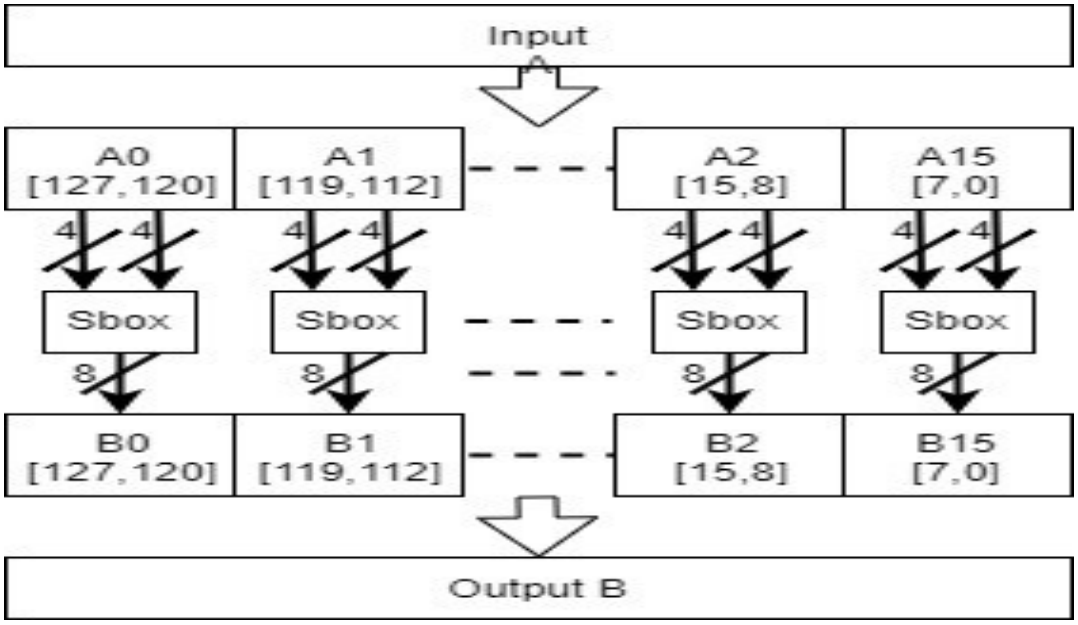

Figure2: Substitute bytes

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	B6	42	68	41	99	2D	0F	B0	54	BB	16

Table 2: S-Box

1.2.3 Shift Rows

The forward shift row transformation, called ShiftRows, is depicted in Figure 3. The first row of State is not altered. For the second row, a 1-byte circular left shift is performed. For the third row, a 2-byte circular left shift is performed. For the fourth row, a 3-byte circular left shift is performed.The following is an example of ShiftRows.

This layer is to provide diffusion for all the state. It contains two sub-layers to ensure the high-degree diffusion after transformation for many rounds.

.

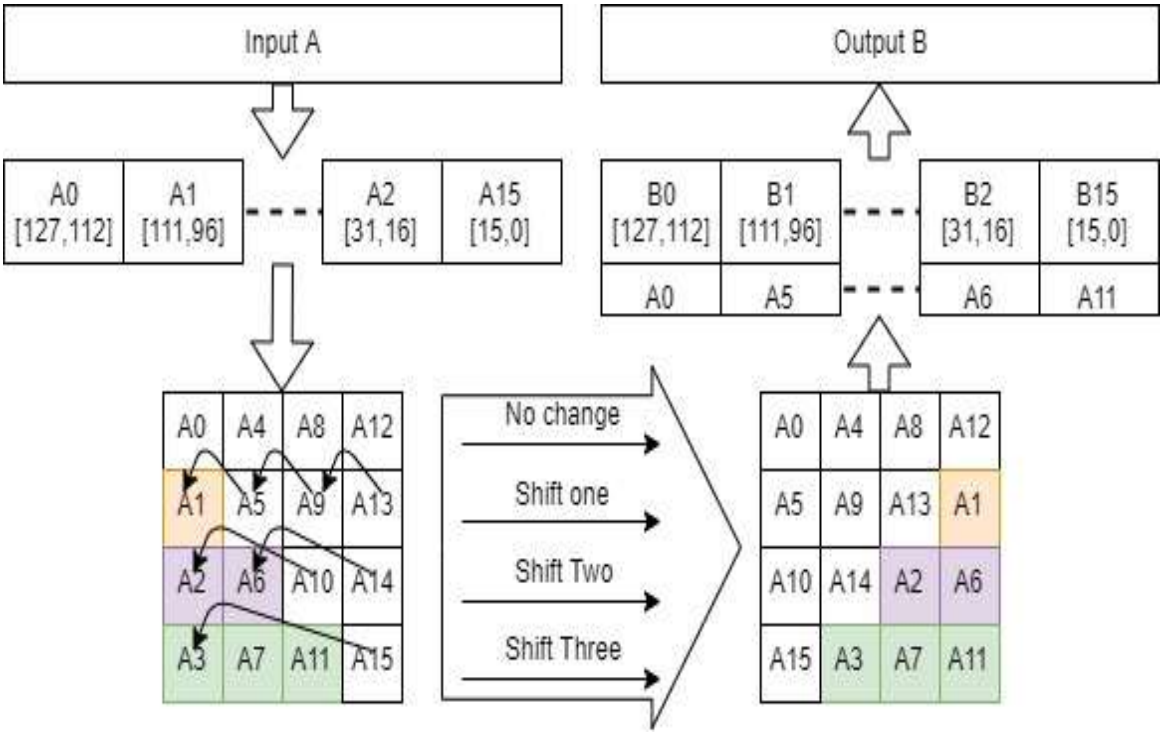

Figure3: Shift Rows

1.2.4 MixColumns

The forward mix column transformation, called MixColumns, operates on each column individually [3]. Each byte of a column is mapped into a new value that is a function of all four bytes in that column. The transformation can be defined by the following matrix multiplication on State (Figure 4).

This layer is to provide diffusion for all the state. It contains two sub-layers to ensure the high-degree diffusion after transformation for many rounds.

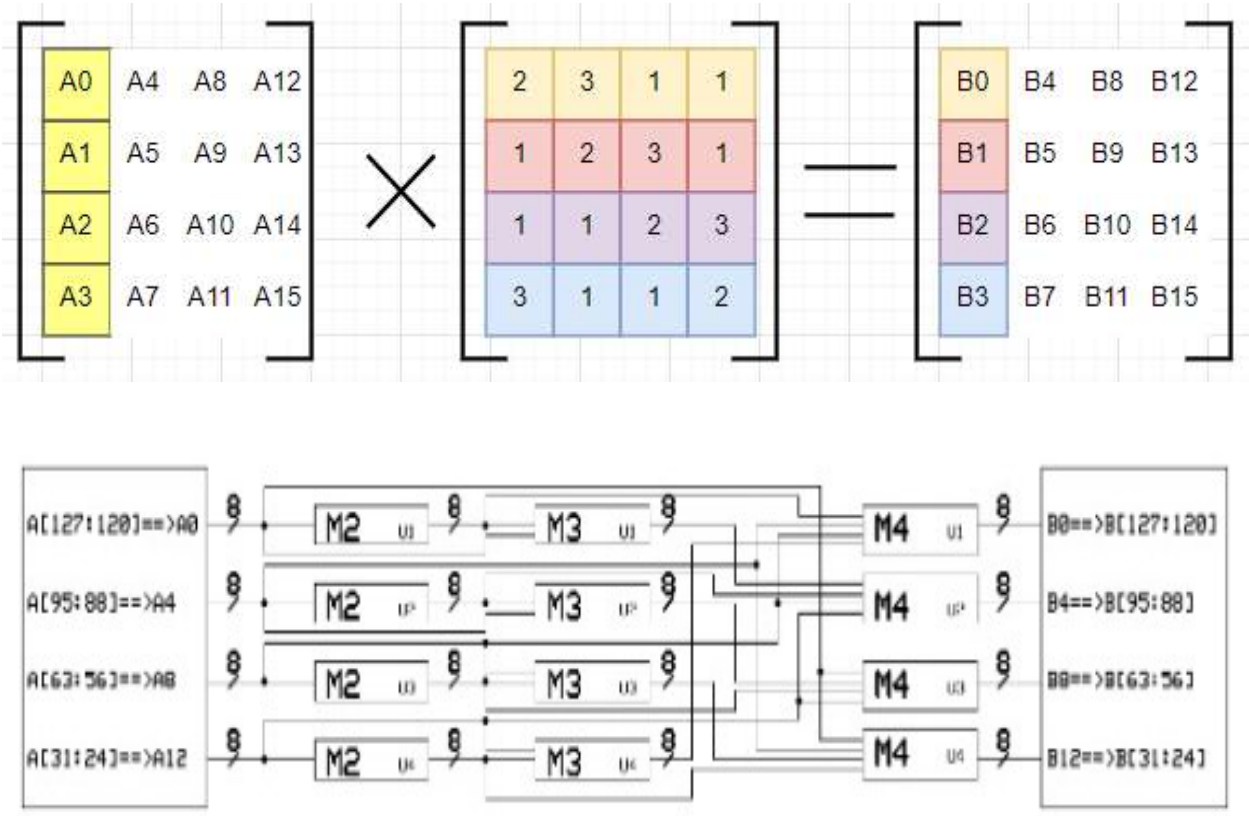

Figure4: MixColumns [28]

1.2.5 AddRoundKey

In the forward add round key transformation, called AddRoundKey, the 128 bits of State are bitwise XORed with the 128 bits of the round key. As shown in Figure 5, the operation is viewed as a columnwise operation between the 4 bytes of a State column and one word of the round key; it can also be viewed as a byte-level operation.

In this layer, the operation is to conduct XOR operation on round key (round key is obtained from the extension of secret key operation) and state. This layer is to establish the relationship between the key and the cipher-text more complicated and to satisfy the confusion principle.

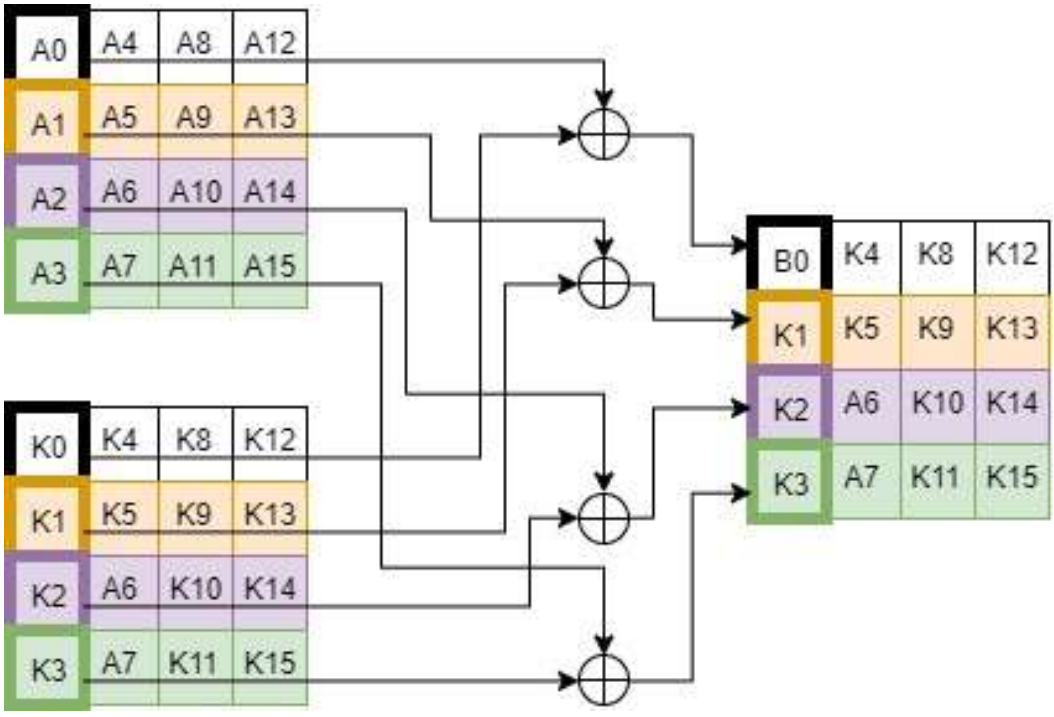

Figure 5: AddRoundKey

1.2.6 Key expansion of AES-128, 192, 256, and 512

The AES key extension algorithm takes a four-word (16-byte) key as input and produces a linear array of 44/52/60/68 words (176/208/240/272bytes). This is enough to provide a four-character round key for the initial AddRoundKey phase and each field, 10/12/14/16 rounds of password. The key is copied into the first four words of the extended key. The rest of the extended key is filled with four words at a time. Each added word depends on the previous word and the word is returned in four positions. In three of the four cases, a simple XOR was used. For words in the w array whose position is a multiple of 4/6/8/10, a more complex function is used. Figure 6,7,8,9 illustrate the generation of the extended key, using the symbol "⊕"to represent the complex function.

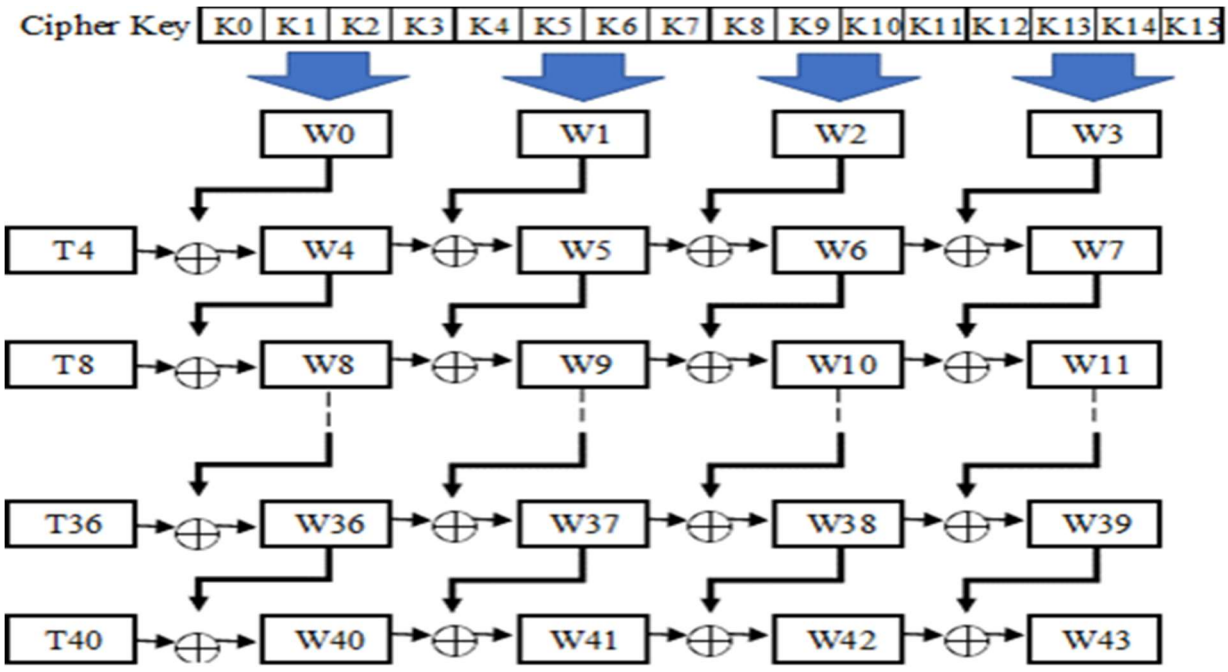

Figure 6: AES-128 Key Expansion

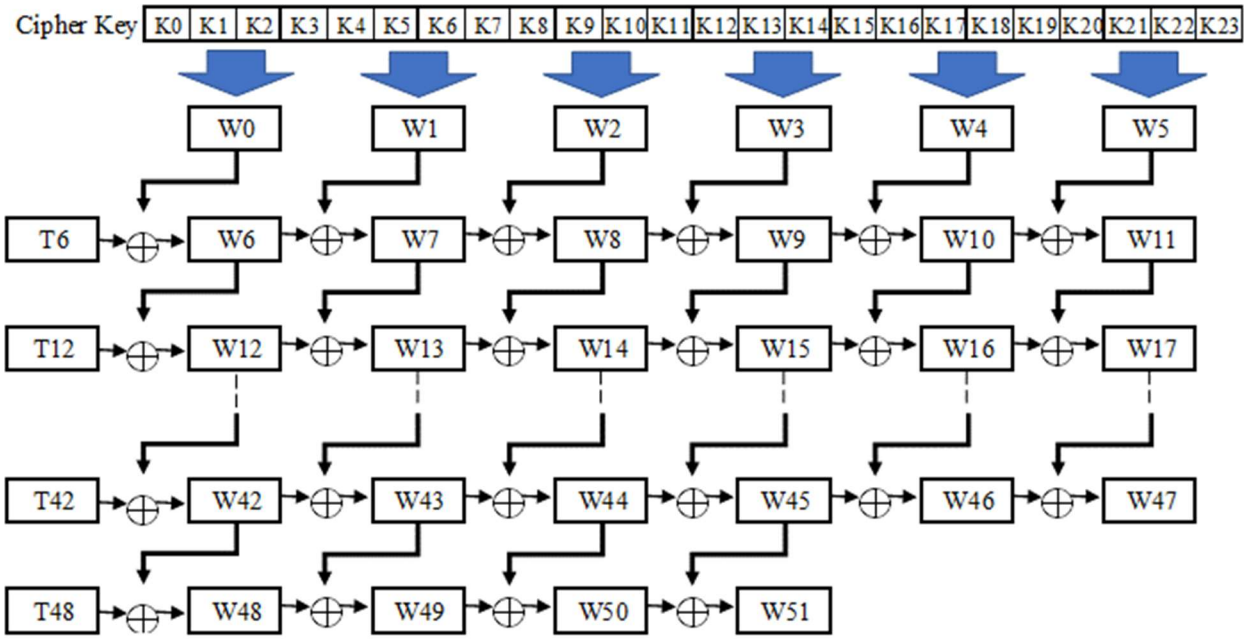

Figure7: AES-192 Key Expansion

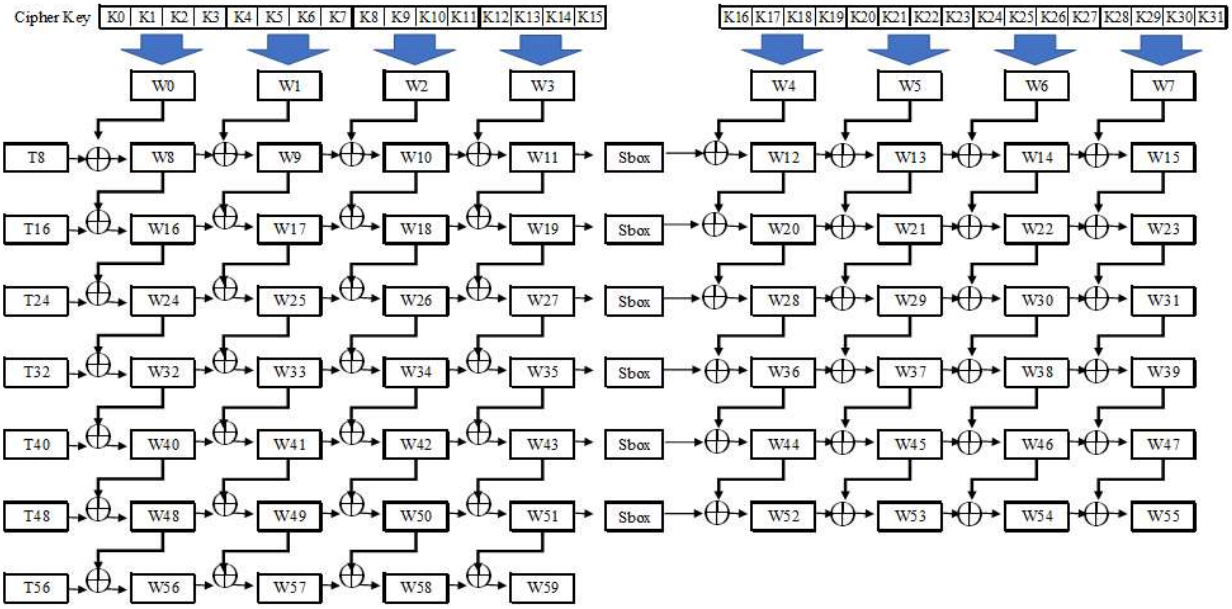

Figure 8: AES-256 Key Expansion

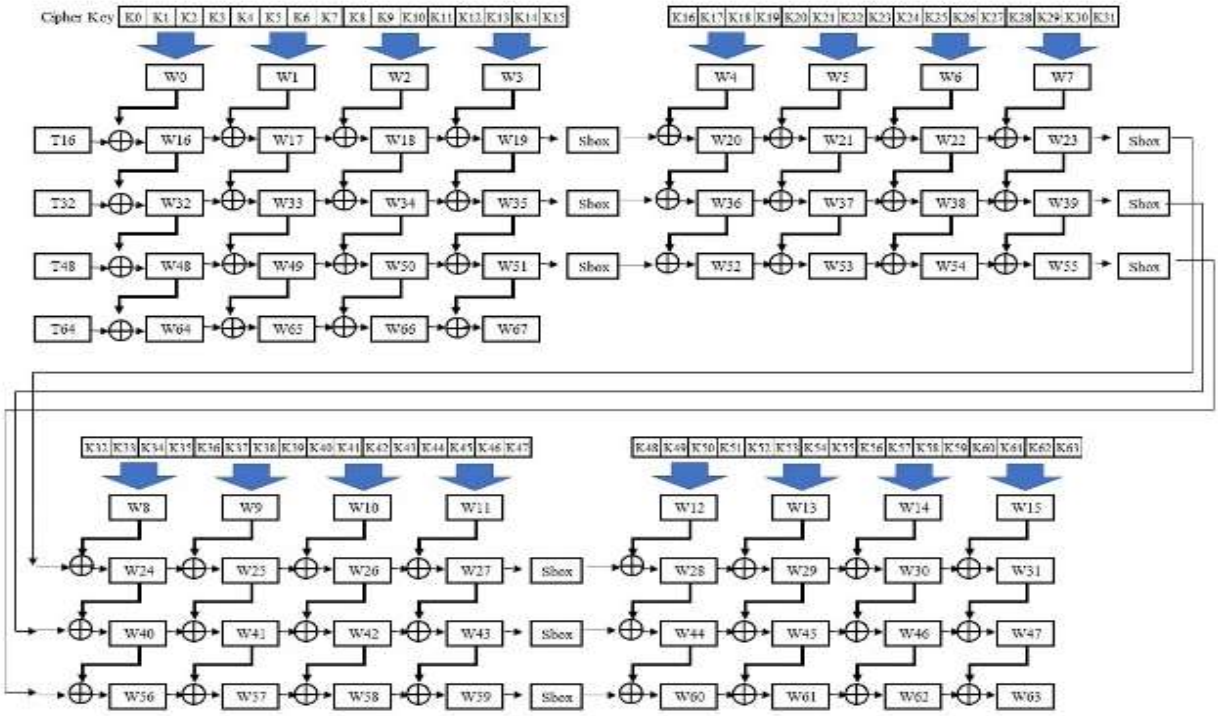

Figure 9: AES-512 Key Expansion

1.3 Block Cipher Modes of Operation

1.3.1 Background

A mode of operation is a technique for enhancing the effect of a cryptographic algorithm or adapting the algorithm for an application such as applying a block cipher to a sequence of data blocks or a data stream. It can be used with any symmetric block cipher algorithm such as DES, 3DES or AES. NIST originally defined four modes of operation, as part of FIPS 81, through which block ciphers can be applied to a variety of applications. However, with newer applications the NIST extended the list of federal recommended modes to five in Special Publication 800-38A.

1.3.2 Modes of Operation

In this section, we will discuss the different modes of operation of a block cipher. These are procedural rules for a generic block cipher. Interestingly, the different modes result in different properties being achieved which add to the security of the underlying block cipher. A block cipher processes the data blocks of fixed size. Usually, the size of a message is larger than the block size. Hence, the long message is divided into a series of sequential message blocks, and the cipher operates on these blocks one at a time.

1.3.2.1 Electronic Codebook (ECB)

This mode is a most straightforward way of processing a series of sequentially listed message blocks. The user takes the first block of plaintext and encrypts it with the key to produce the first block of ciphertext. He then takes the second block of plaintext and follows the same process with same key and so on so forth. The ECB mode is deterministic, that is, if plaintext block p1, p2…, pM are encrypted twice under the same key, the output ciphertext blocks will be the same.

In fact, for a given key technically we can create a codebook of ciphertexts for all possible plaintext blocks. Encryption would then entail only looking up for required plaintext and select the corresponding ciphertext. Thus, the operation is analogous to the assignment of code words in a codebook, and hence gets an official name − electronic codebook mode of operation (ECB). It is illustrated as follows –

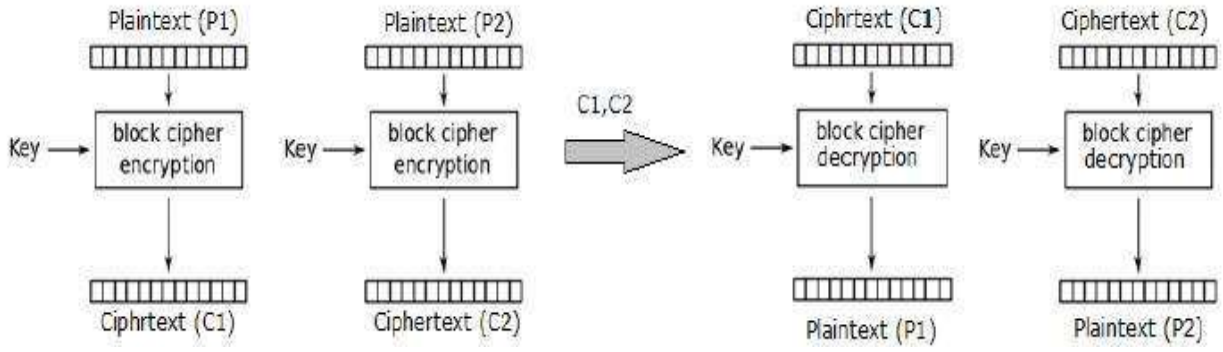

Figure 10: Electronic codebook mode of operation (ECB) [2]

In reality, any application data usually have partial information which can be guessed. For example, the range of salary can be guessed. A ciphertext from ECB can allow an attacker to guess the plaintext by trial-and-error if the plaintext message is within predictable.

For example, if a ciphertext from the ECB mode is known to encrypt a salary figure, then a small number of trials will allow an attacker to recover the figure. In general, we do not wish to use a deterministic cipher, and hence the ECB mode should not be used in most applications.

1.3.2.2 Cipher Block Chaining (CBC)

CBC mode of operation provides message dependence for generating ciphertext and makes the system non-deterministic. The operation of CBC mode is depicted in the following illustration. The steps are as follows −

- Load the n-bit Initialization Vector (IV) in the top register.
- XOR the n-bit plaintext block with data value in top register.
- Encrypt the result of XOR operation with underlying block cipher with key K.

- Feed ciphertext block into top register and continue the operation till all plaintext blocks are processed.
- For decryption, IV data is XORed with first ciphertext block decrypted. The first ciphertext block is also fed into to register replacing IV for decrypting next ciphertext block.

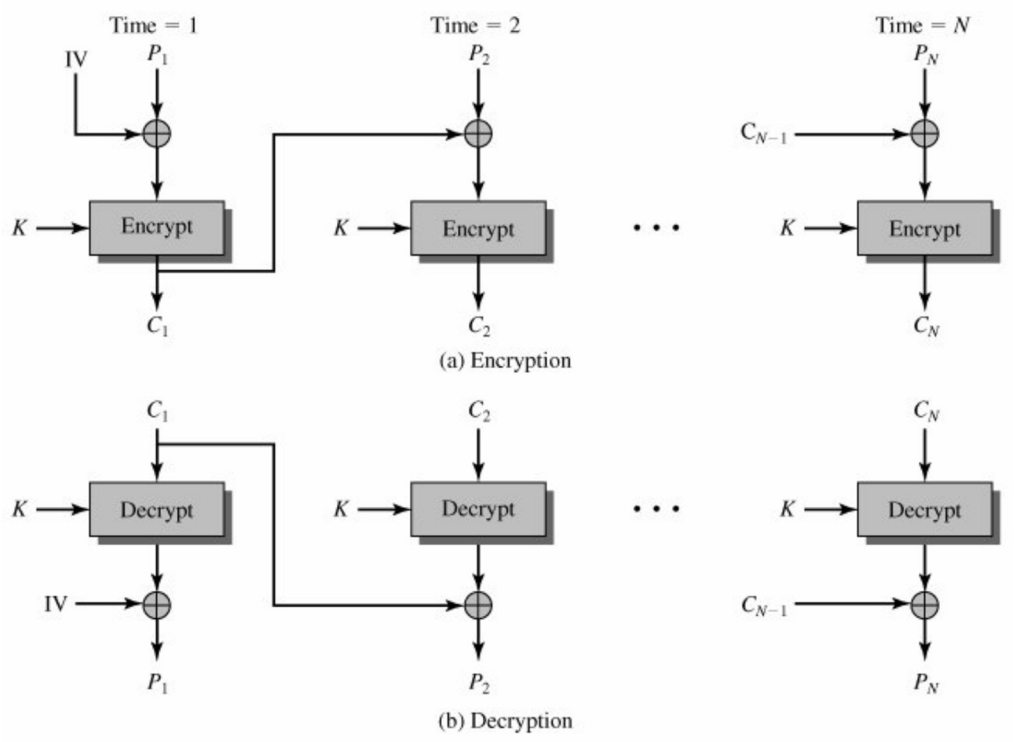

Figure 11: Cipher block chaining mode of operation (CBC) [2]

In CBC mode, the current plaintext block is added to the previous ciphertext block, and then the result is encrypted with the key. Decryption is thus the reverse process, which involves decrypting the current ciphertext and then adding the previous ciphertext block to the result.

Advantage of CBC over ECB is that changing IV results in different ciphertext for identical message. On the drawback side, the error in transmission gets propagated to few further blocks during decryption due to chaining effect. It is worth mentioning that CBC mode forms the basis for a well-known data origin authentication mechanism. Thus, it has an advantage for those applications that require both symmetric encryption and data origin authentication.

1.3.2.3 Cipher Feedback Mode (CFB)

In this mode, each ciphertext block gets 'fed back' into the encryption process in order to encrypt the next plaintext block. The operation of CFB mode is depicted in the following illustration. For example, in the present system, a message block has a size 's' bits where $1 < s < n$. The CFB mode requires an initialization vector (IV) as the initial random n-bit input block. The IV need not be secret. Steps of operation are −

- Load the IV in the top register.
- Encrypt the data value in top register with underlying block cipher with key K.
- Take only 's' number of most significant bits (left bits) of output of encryption process and XOR them with 's' bit plaintext message block to generate ciphertext block.
- Feed ciphertext block into top register by shifting already present data to the left and continue the operation till all plaintext blocks are processed.
- Essentially, the previous ciphertext block is encrypted with the key, and then the result is XORed to the current plaintext block.

- Similar steps are followed for decryption. Pre-decided IV is initially loaded at the start of decryption.

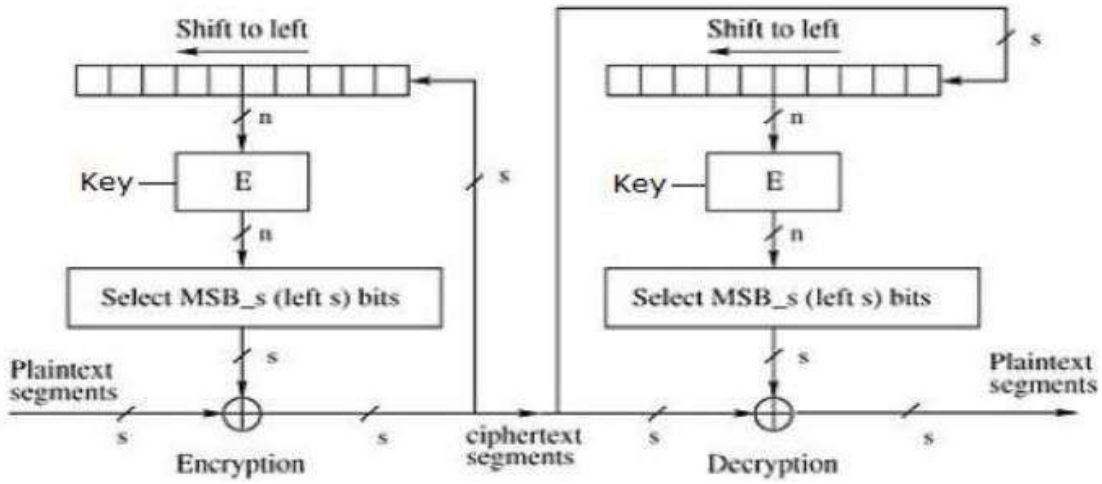

Figure 12: Cipher feedback mode of operation (CFB) [2]

CFB mode differs significantly from ECB mode, the ciphertext corresponding to a given plaintext block depends not just on that plaintext block and the key, but also on the previous ciphertext block. In other words, the ciphertext block is dependent of message. CFB has a very strange feature. In this mode, user decrypts the ciphertext using only the encryption process of the block cipher. The decryption algorithm of the underlying block cipher is never used. Apparently, CFB mode is converting a block cipher into a type of stream cipher. The encryption algorithm is used as a key-stream generator to produce keystream that is placed in the bottom register. This key stream is then XORed with the plaintext as in case of stream cipher. By converting a block cipher into a stream cipher, CFB mode provides some of the advantageous

properties of a stream cipher while retaining the advantageous properties of a block cipher. On the flip side, the error of transmission gets propagated due to changing of blocks.

1.3.2.4 Output Feedback (OFB) Mode

It involves feeding the successive output blocks from the underlying block cipher back to it. These feedback blocks provide string of bits to feed the encryption algorithm which act as the key-stream generator as in case of CFB mode. The key stream generated is XOR-ed with the plaintext blocks. The OFB mode requires an IV as the initial random n-bit input block. The IV need not be secret.

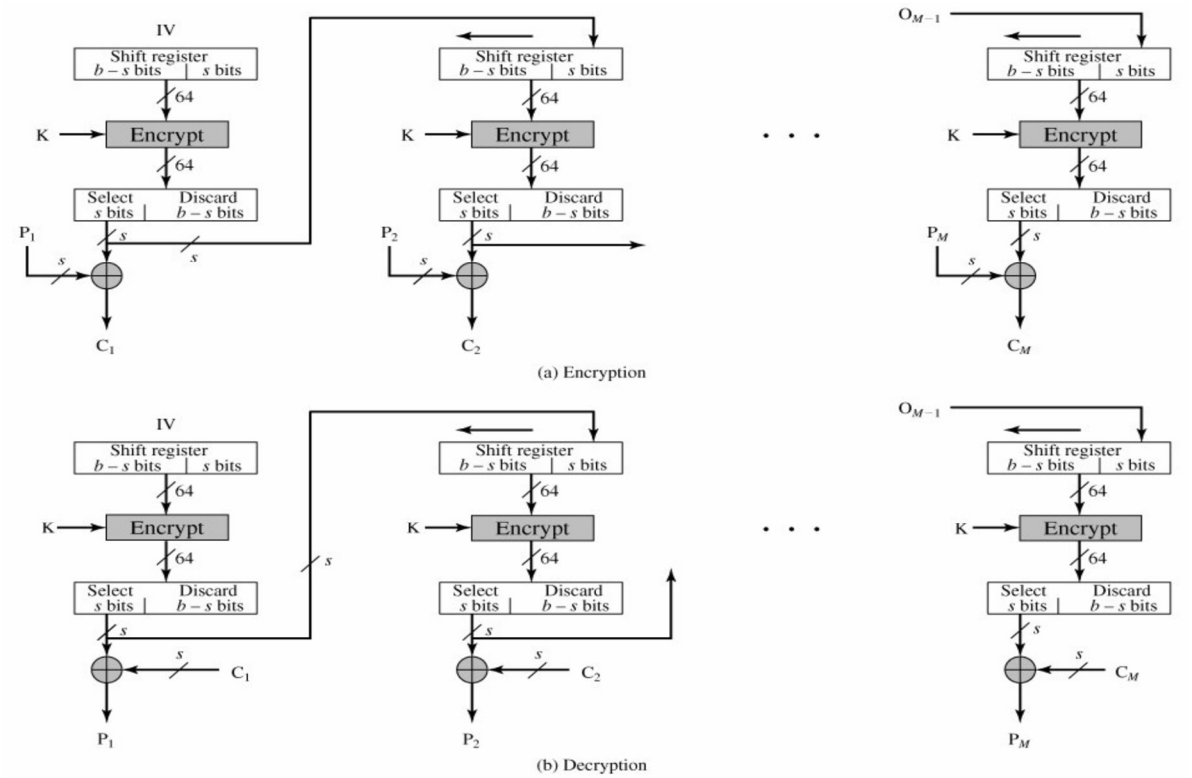

Figure 13: Output feedback mode of operation (OFB) [2]

1.3.2.5 Counter (CTR) Mode

It can be considered as a counter-based version of CFB mode without the feedback. In this mode, both the sender and receiver need to access to a reliable counter, which computes a new shared value each time a ciphertext block is exchanged. This shared counter is not necessarily a secret value, but challenge is that both sides must keep the counter synchronized. Both encryption and decryption in CTR mode are depicted in the following illustration. Steps in operation are −

- Load the initial counter value in the top register is the same for both the sender and the receiver. It plays the same role as the IV in CFB (and CBC) mode.
- Encrypt the contents of the counter with the key and place the result in the bottom register.
- Take the first plaintext block P1 and XOR this to the contents of the bottom register. The result of this is C1. Send C1 to the receiver and update the counter. The counter update replaces the ciphertext feedback in CFB mode.
- Continue in this manner until the last plaintext block has been encrypted.
- The decryption is the reverse process. The ciphertext block is XORed with the output of encrypted contents of counter value. After decryption of each ciphertext block counter is updated as in case of encryption.

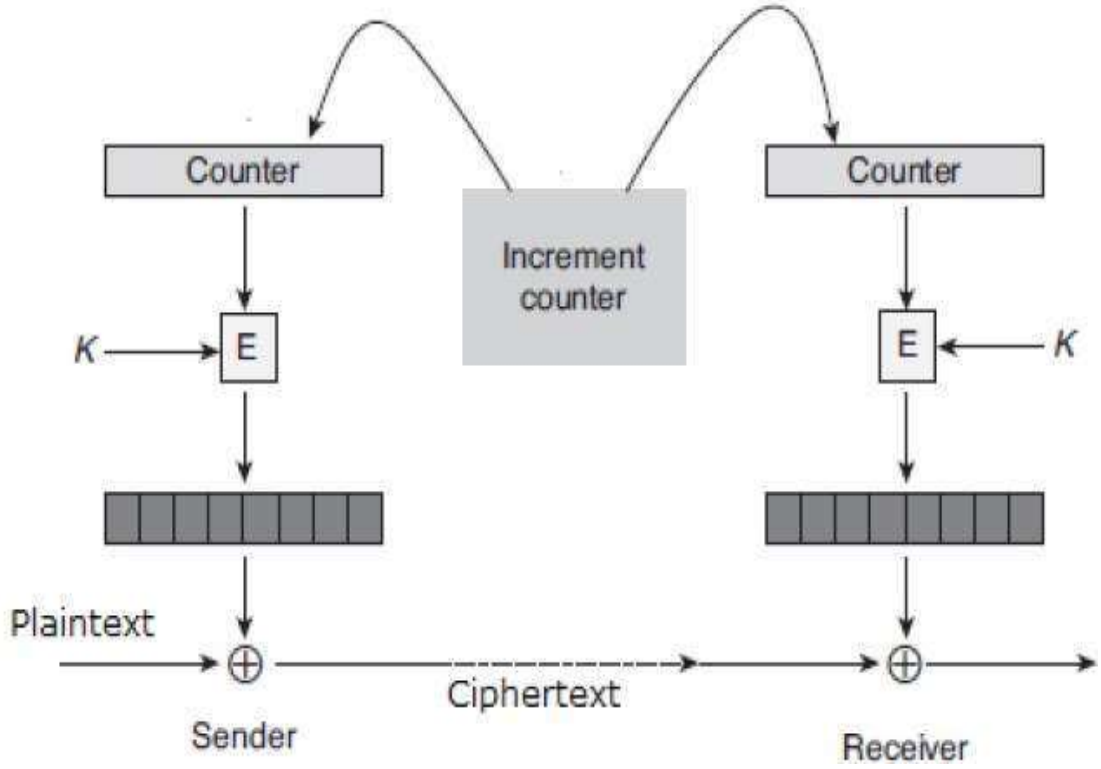

Figure 14: Counter mode of operation (CTR) [2]

It does not have message dependency and hence a ciphertext block does not depend on the previous plaintext blocks. Like CFB mode, CTR mode does not involve the decryption process of the block cipher. This is because the CTR mode is really using the block cipher to generate a keystream, which is encrypted using the XOR function. In other words, CTR mode also converts a block cipher to a stream cipher. The serious disadvantage of CTR mode is that it requires a synchronous counter at sender and receiver. Loss of synchronization leads to incorrect recovery of plaintext. However, CTR mode has almost all advantages of CFB mode. In addition, it does not propagate error of transmission at all.

1.4 Hardware and Software Design Assessment

We use the EDA tools available on the Altera website to evaluate our designs [4-7]. These tools, the Quartus II Web Edition and the Altera University Program Simulator, allow code to be built, compiled, synthesized, simulated, and finally programmed into DE2 hardware. In this work, we use Altera's Cyclone IV DE2-115 board EP4CE115F29 platform as shown in Figure 15. Cyclone IV technology was released in 2017. The model EP4CE115F29C7 has a density of 114,480 LE and it contains an internal 50 MHz clock. The development board is available on the Terasic website. The basic information about the computer we run the experiments is shown as Figure 16.

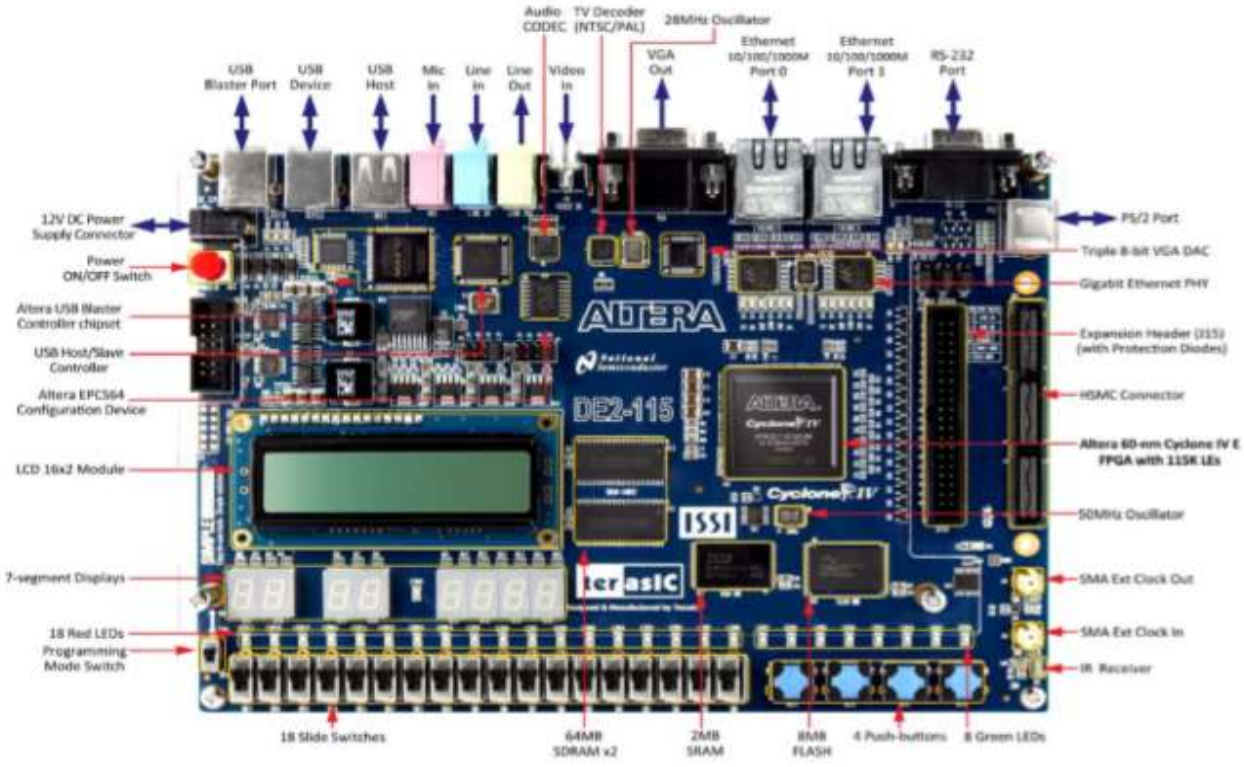

Figure 15: Altera Cyclone IV 4CE115 FPGA Device [7]

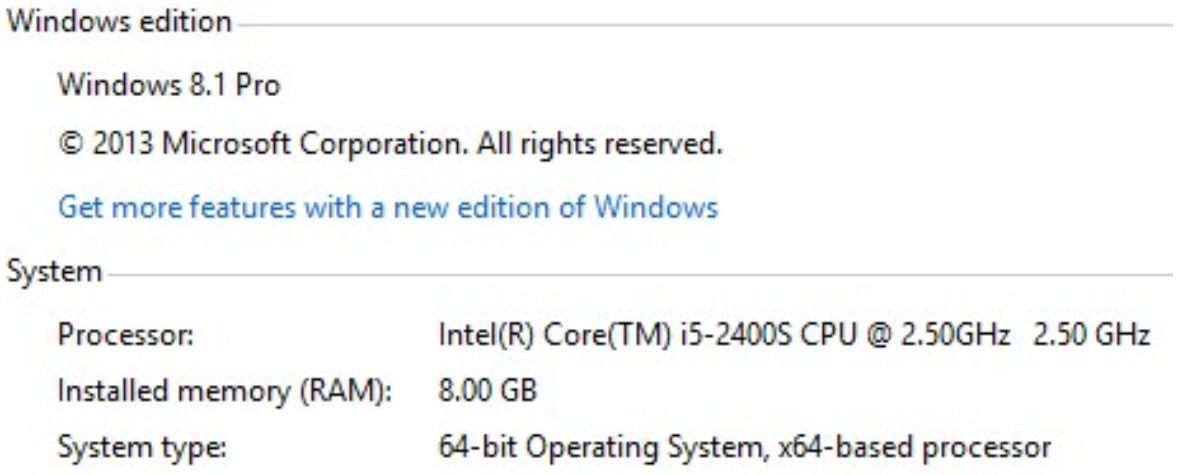

Figure 16: Basic information about the computer

1.5 Thesis Outline

This thesis is organized as follows. The first chapter discusses the introduction of cryptography, a brief overview of the symmetric encryption algorithm- AES, a brief overview of block cipher mode of operation, and problem statement. Chapter II discusses the Mix Column modules in detail and how to improve the computation by parallelism and pipelining. Chapter III focuses the S-box modules in detail how to improve the computation by parallelism and pipelining. Chapter IV discusses the performance comparison of AES variants which is the Key Expansion in detail. Chapter V gives the experimental results and simulation analysis of AES scheme and its four variants including AES-128, AES-192, AES-256 and new AES-512 with different modes of operation, on the FPGA platform. Finally, Chapter VI provides conclusion and the future work.

CHAPTER II

HARDWARE IMPLEMENTATION OF IMPROVED MIX COLUMN COMPUTATION OF CRYPTOGRAPHIC AES

With today's development and expansion of networks and internet-connected devices, information security is an issue of increasing concern. Confidentiality is one of the focuses in network security for digital communication systems, where large data blocks go through a cryptographic algorithm with a cipher key that increases the security and complexity of the output ciphertext. AES is a symmetric encryption algorithm that has a minimum input data block size of 128-bits which undergo a series of permutations, substitutions, and digital logic operations over several rounds. Encryption algorithms are always improving on ciphertext complexity, required hardware storage allocation, and execution time. Field Programmable Gate Arrays (FPGA's) are a hardware alternative for encryption algorithm implementation because, although the logic units in it are fixed, the functions and interconnections between them are based on the user's design which allow for improvement. The research presented focuses on the development and analysis of an efficient AES-128 Mix Columns algorithm implementation, utilized in the data block encryption rounds, on an Altera Cyclone IV FPGA using the Intel Quartus II software and Verilog Hardware Description Language.

2.1 Rijndael Mix Column Computation

Intensive computation of AES takes place in the Rijndael Mix Column segment. The Mix Column transformation operates on each column of the 4-byte by 4-byte matrix formed from the

input 128-bit data block. Each byte of the column is mapped into a new value that is a function of all four bytes in that column. The implementation of Mix Columns is based on the mathematical analysis in the Galois field, GF (2^8). For the AES algorithm this irreducible polynomial is:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \qquad \text{(Equation 2.1.1)}$$

The columns of the matrix are multiplied by modulo x^4 + 1 with a fixed polynomial c (x), given by:

$$c(x) = [03]x^3 + [01]x^2 + [01]x + [02] \qquad \text{(Equation 2.1.2)}$$

This polynomial is coprime to x^4 + 1 and therefore invertible. Only the multiplication module and the 32-bit XOR module of each processing unit (one column) are needed for the design because the elements of the multiplication and addition in the Galois field are commutative and associative. In Figure 17 shows a Mix Columns computation example. With this approach, the function of Mix Columns can be achieved.

For example:
1. {02.BF} {BF} . {02} = 1011 1111 << 1 =0111 1110 ⊕ 0001 1011 =0110 0101
2. {03.5D} {5D} . {03} = {0101 1101 . 02} ⊕ { 0101 1101} = 1011 1010 ⊕ 0101 1101 = 1110 0111
∴ r1 = {01.D4} + {02.BF} + {03.5D} + {01.30} = (1101 0100) ⊕ (0110 0101) ⊕ (1110 0111) ⊕ (0011 0000) = 0110 0110 = $(66)_{16}$

Figure17: Rijndael Mix Columns computation example

Many researchers have published comparative analysis and optimized the speed of hardware implementations on the performance of different modes of operation. The results, from [8-20], reveal that the scheme has low hardware resource consumption, high throughput and an excellent overall performance ratio. Here, we focus on the development and analysis of an efficient AES-128 Mix Columns algorithm implementation in two different approaches.

2.2 Design Methods and Discussion

Our first approach involved building the circuit modules around the traditional row-column multiplication method. This was done by creating codes for each row of the Rijndael mix columns matrix producing a total of four distinct sub-modules (shown in Figure 18). Each module would take 4 bytes (a column of the incoming data block matrix) and produce a single corresponding byte of the output matrix. This approach resulted in a circuit configuration that was simplistic in concept, however based on Figure 19, the circuit reveals only partial bit parallelization which could potentially deter the performance of the circuit because of misaligned clock cycles.

Our second approach focused on forcing a more parallel behavior into the circuit to align the input signals together potentially resulting in decreased delay. As seen in Figure 20, three distinct sub-modules were created that separate the Rijndael multiplication with factors 2 and 3 and adds an additional module ‘M4’ as a 4-input XOR function. Each of the 16 bytes of the input data block matrix will go through the same sub-modules, aligning them during their clock cycles, and producing the 16-byte output matrix at approximately the same cycle.

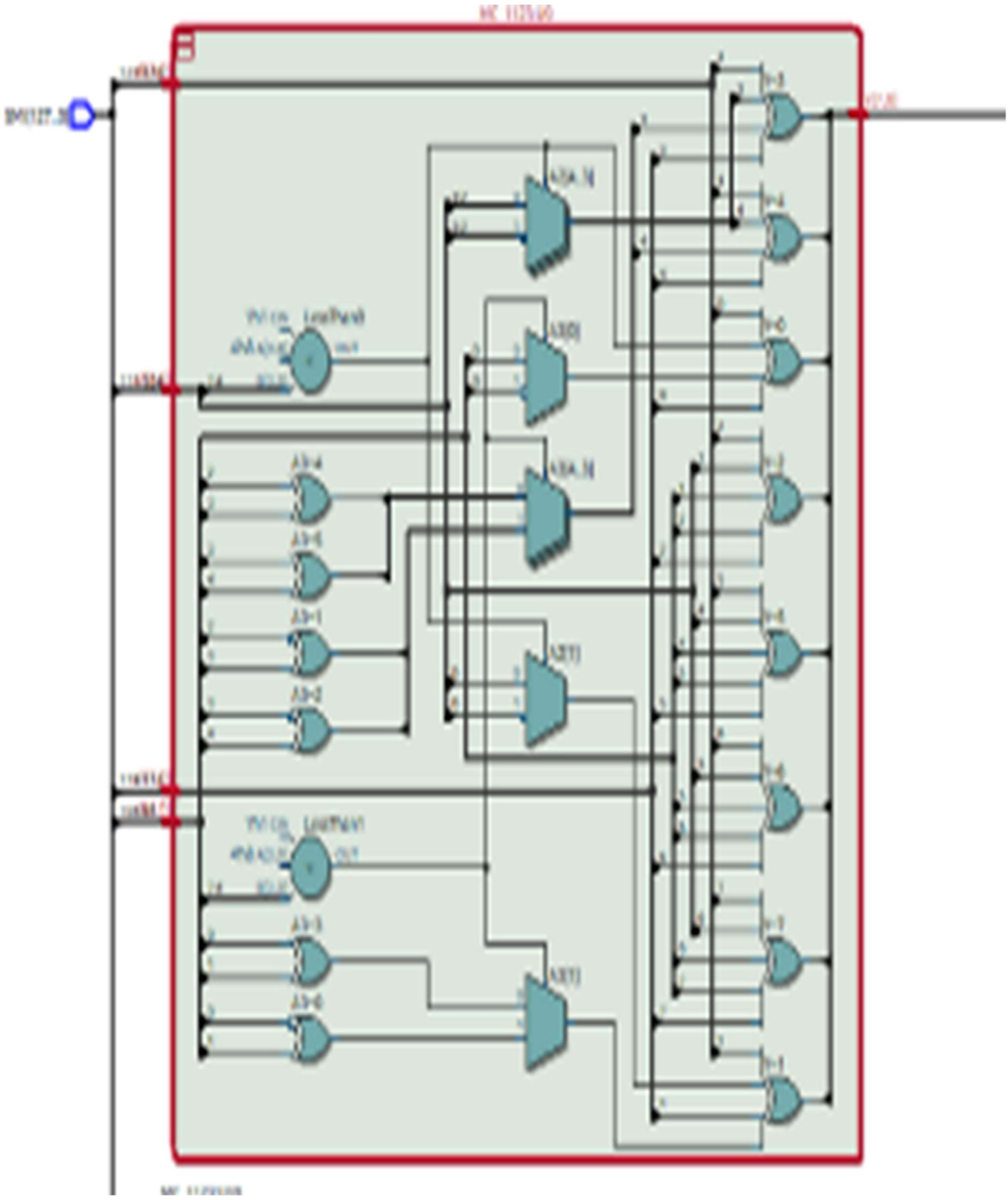

Figure 18: Internal schematic of each submodule

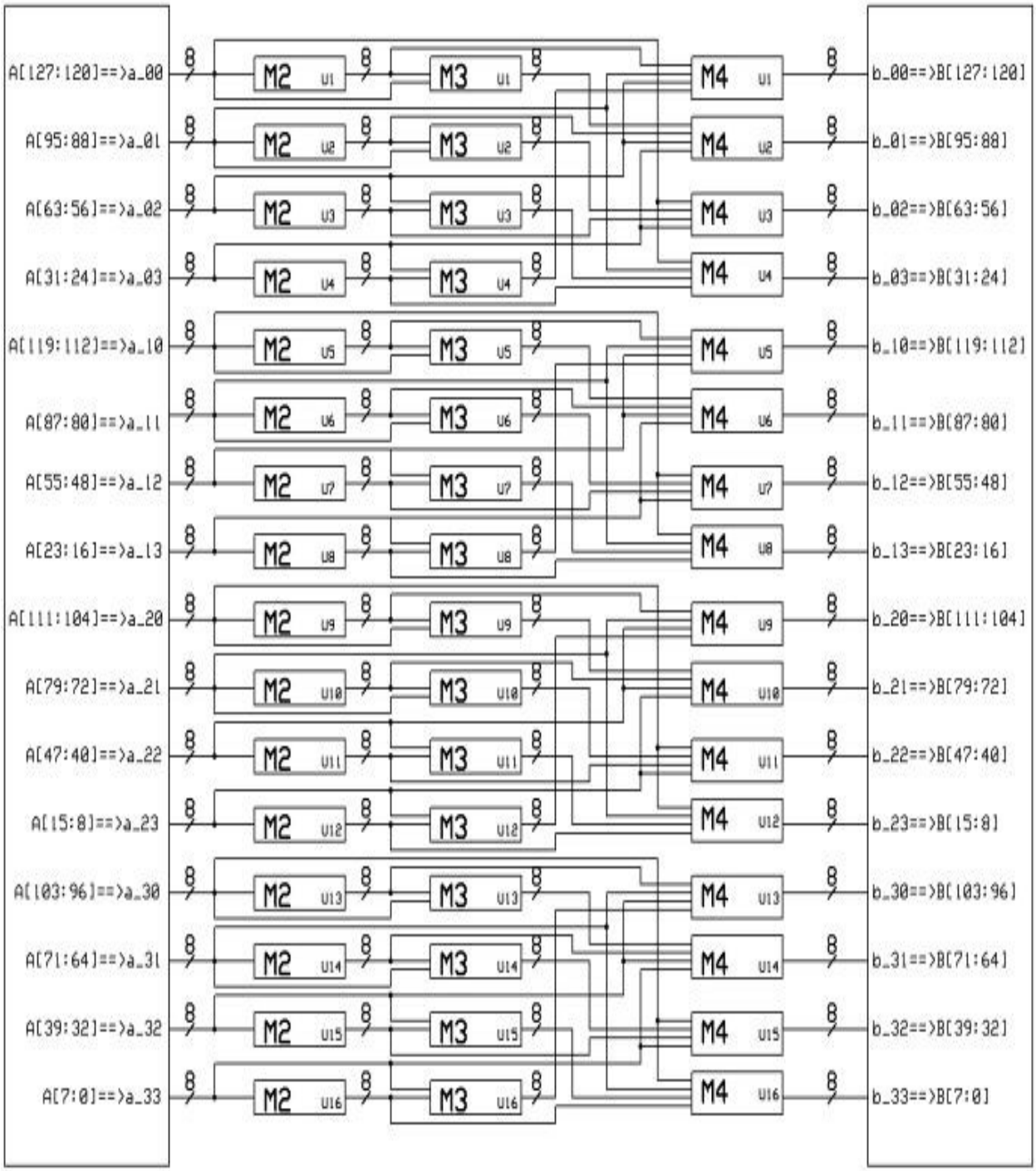

Figure 19: Module block diagram for parallelized configuration approach

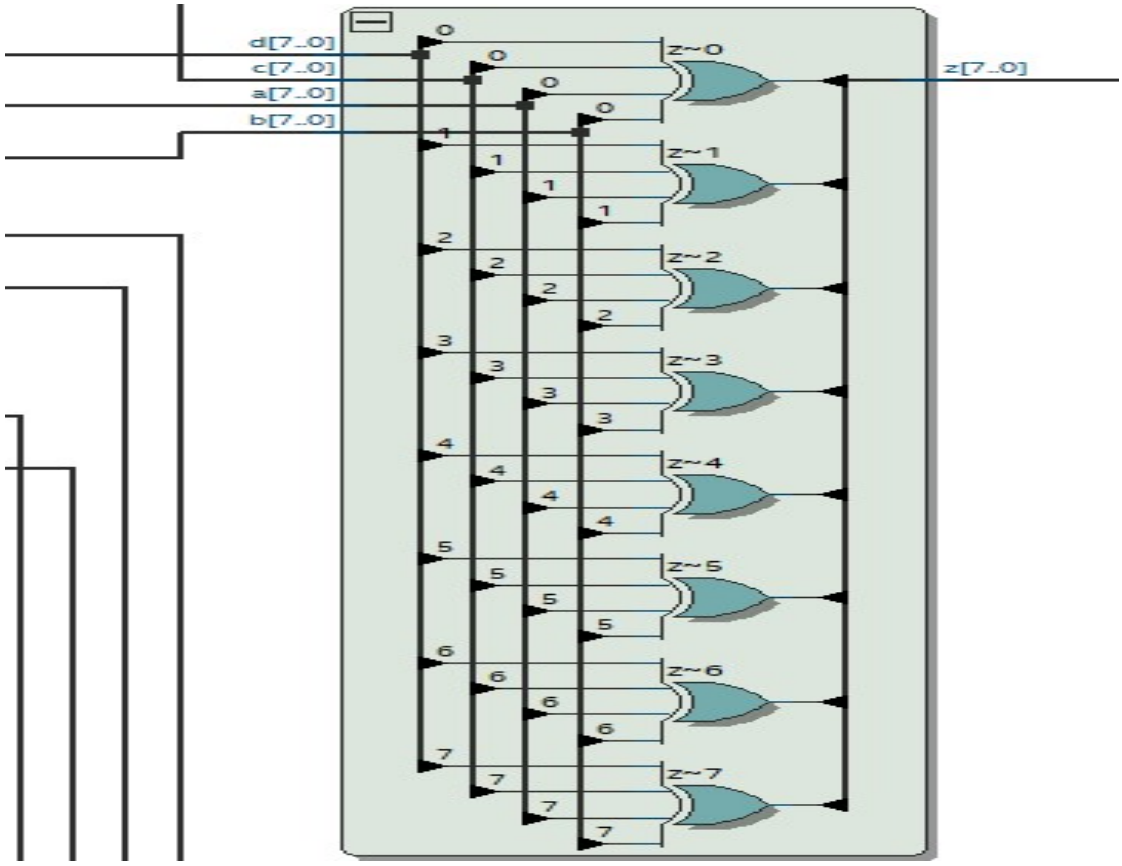

Figure 20: Internal schematic of M4 submodule

2.3 Result and Analysis

The following data presented is modeled after previous performance comparison attempts mentioned in [8-20]. According to the AES algorithm we mentioned in Section 2, we can distinguish Rijndael Mix Column into many operations or functions. In addition, we used the Quartus II software to perform the timing simulations for each operation or function in each step, as shown in Figure21~25. Finally, we developed the following tables to compare the performance.

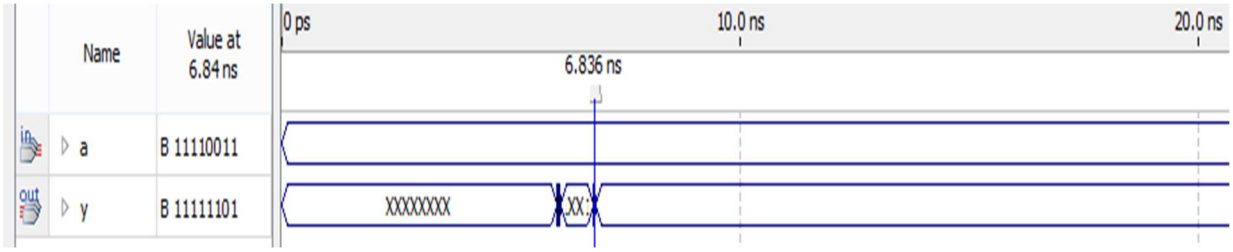

Figure 21: Delay of M2 submodule

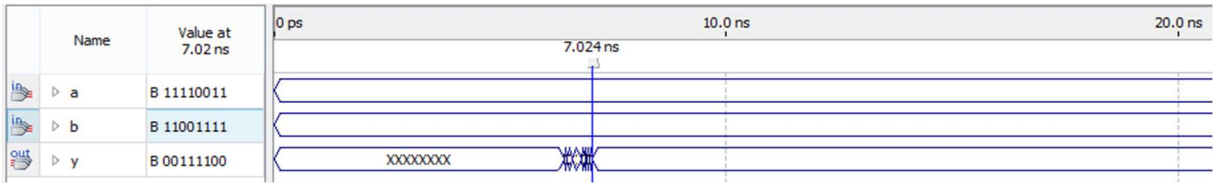

Figure 22: Delay of M3 submodule

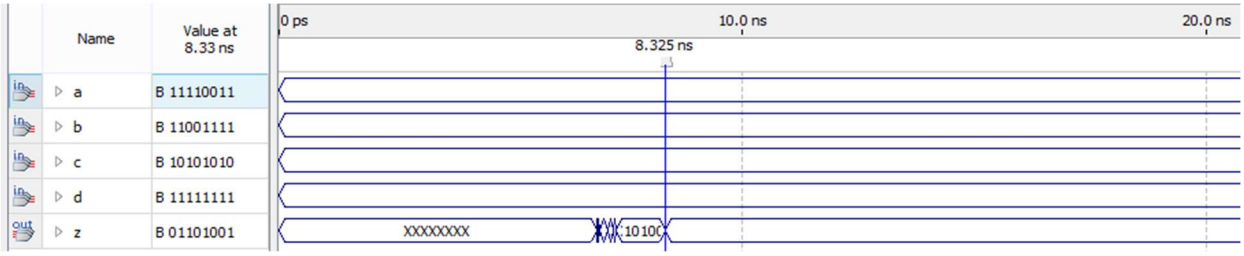

Figure 23: Delay of M4 submodule

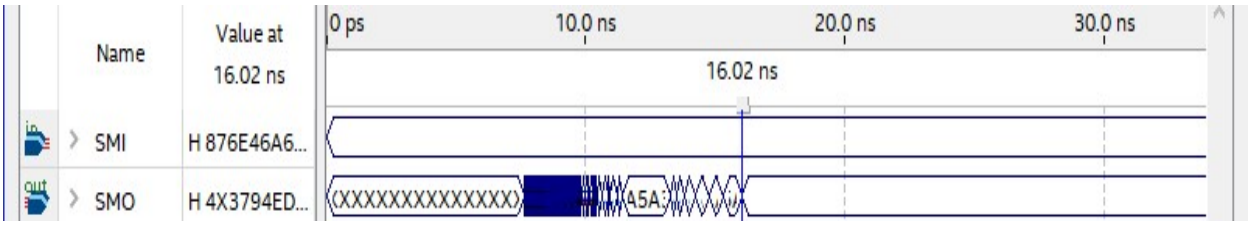

Figure 24: Timing simulation showing delay without parallelism

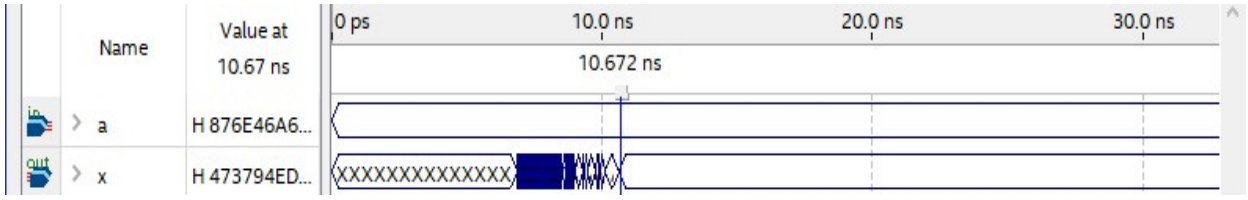

Figure 25: Timing simulation showing delay with parallelism

Function	Operation Type	Processing Time(ηs)		AES 128
Sub Bytes	Substitution (S-Box)	12		10
Shift Row	Assign	7.75		10
Mix Columns	M2 (Left Shift + XOR2)	6.84	10.51	9
	M3(XOR2)	7.02		9
	M4(XOR4)	8.83		9
Add Round Key	XOR2	7.02		11
Total Processing Time (ηs)				369.3

Table 3: Delay of Plaintext Encryption

The results in Table 3 for Mix Columns were obtained for each submodule operation. The final simultaion shows that the delay for each operation combines to give a reduced parallel result.

Prog.	Hardware Consumption		Delay (ηs)	Delay (ms)					
	TLE	PVM	128 bits	1Kbyte	1Mbyte	10Mbyte	100Mbyte	1Gbyte	10Gbyte
Without parallelism	224	5699MB	16.01	0.009	9.0	90.1	900.6	900563	9005625
With parallelism	192	5517MB	10.66	0.006	6.0	60.0	599.6	599625	5996250

Table 4: Time Delay and Memory results of both implementations

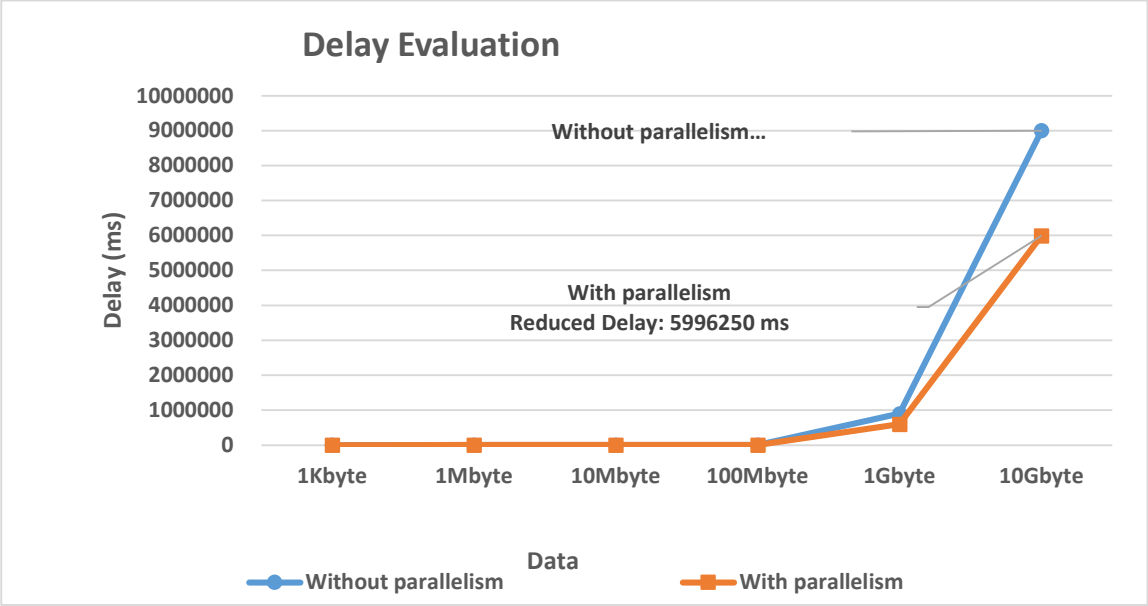

Figure 26: Time delay analysis of mixed columns

The results and calculations shown in Table 3 evaluate the Total Logic Elements (TLE) and the Peak Virtual Memory (PVM) of the non-parallelized program “Without parallelism” and

the parallelized one “With parallelism”. The graphical evaluation of the gathered data is depicted below.

Compared to the non-parallelized program, the parallelized version showed a significant decrease in needed logic elements and virtual memory for implementation. Based on the data in Table 4, at 10 GB of input data the parallelized implementation reduces the Rijndael Mix Columns operation delay by approximately 33.4%.

Based on encryption time for Mix Columns, we created a Delay Evaluation (Table 4) and made a plot to show the performance of the different approaches in terms of encryption time. We plotted for different file sizes shown in Figure 26 and observed that the 2nd approach, which parallelized the circuit signals more, had less time delay than the 1st approach. It was also noticed that the difference in memory allocation and size corresponded to the difference in delay.

2.4 Conclusion

In this Chapter, we conduct our study on the most popular encryption algorithm AES. We targeted one mode of operation, Cipher Block Chaining (CBC), in terms of encryption time and delay on the Mix columns section. The results in Table 4 reveals that using parallelism in signal processing results in reduced time delay, logic elements and virtual memory. In the following chapter, we will focus on other sections for parallelization and try to implement AES on the FPGA. Finally, we will be able to obtain optimized area and speed hardware implementations of AES based on the sub-pipelined architecture.

CHAPTER III

A FAST IMPLEMENTATION OF THE RIJNDAEL SUBSTITUTION BOX FOR CRYPTOGRAPHIC AES

Today's current standard in Cryptography is the Symmetric Advanced Encryption Standard (AES) as selected by the US National Institute of Standards and Technology (NIST). It is also known as Rijndael Encryption Algorithm. Compared to various components of the AES, Rijndael S-box (substitution box) is the only non-linear component of the cryptosystem and significantly affects the overall performance of the AES encryption scheme. In this chapter, we investigate various implementations for improving the hardware performance of the Rijndael S-box component of the AES algorithm in terms of delay and size on the Altera Cyclone IV FPGA (Field programmable gate arrays) using the Intel Quartus II software and Verilog Hardware Description Language (Verilog HDL).

The AES encryption algorithm accepts blocks of 128 or 192 or 256 bits and applies a series of substitutions and permutations [1-2]. A special substitution termed as "SubBytes Transformation" is also called Rijndael S-box, named after its designers. S-box is the main core structure of every block cipher system and controls the hardware complexity of Rijndael cipher elements due to its particular characteristics, a non-linear byte substitution and operating on each of the State bytes independently. The purpose of S-box is to produce confusion between the ciphertext and the secret key. There are 256 = 16 × 16 possible 8-bit numbers, and so the S-box can be represented as a 16 ×16 table mapping inputs to outputs.

S-box provides reversible conversion of plain text segments during the encryption process, while providing the opposite conversion during the decryption process. It is a single simple function that is applied to each byte over and over again during the encryption phase. Each of the 256 possible byte values is converted to another byte value by the transformation, which is a complete permutation as mentioned at Chapter 1. As a result, no two different byte values are changed to the same byte values.

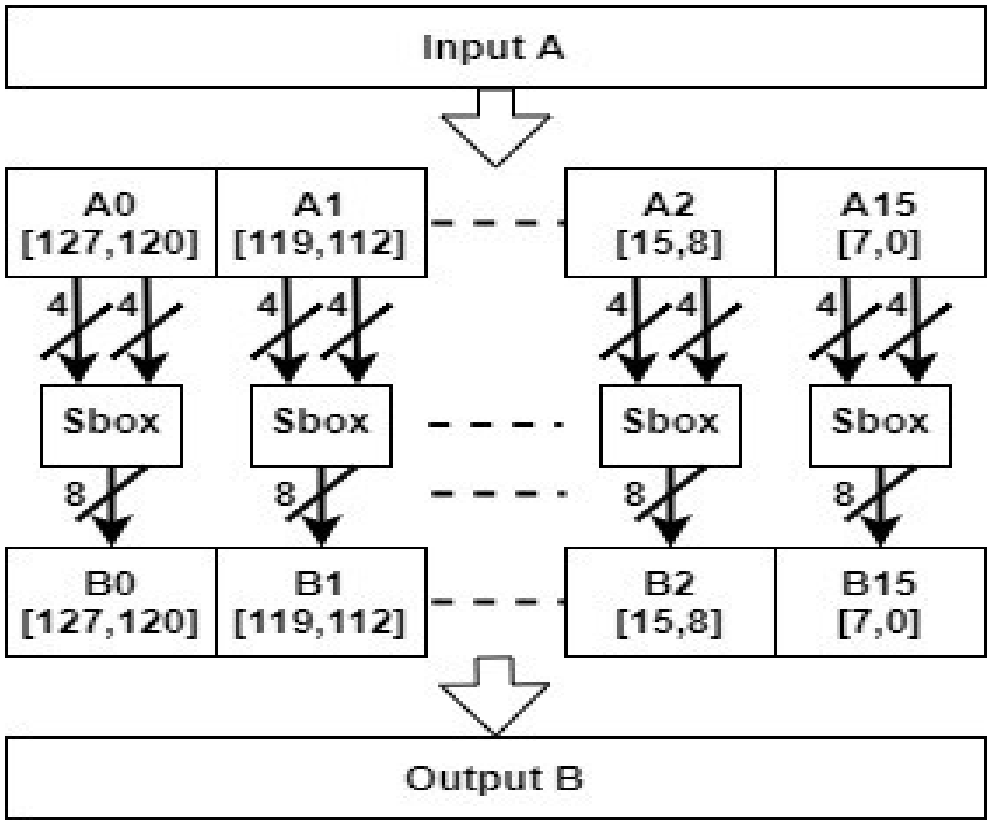

Figure 27: Substitute Bytes

The AES S-box is shown as Figure 27[1-2]. To find the output from the S-box table, the byte input is split into two 4-bit halves. The first half provides the row number and the second

half provides the column number for the Byte substitution. For example, an S-box transformation of 8'b11000011or 0xC3 can be found in a cell at the intersection of a row labeled C0 in hexadecimal and a column labeled 03 in hexadecimal. Therefore, 8'b11000011or 0xC3 becomes 8'b00101110 or 0x2E.

Several hardware implementations related work is available in literature [8-20]. Many literatures have proposed S-box hardware lookup table implementations [21-27]. To reduce LUT space requirements, the basic idea of Shannon's expansion theorem is applied. It helps achieve a logical design that has a greater number of levels with less implementation cost. This optimization technique reduces the complexity of the whole S-box which means it requires fewer arithmetic operations. As it simplifies table indexing, it simply consumes less power. Besides this optimization technique, including a smaller number of iterations will decrease the delay producing algebraic and matrix operations. In this chapter, we simulate different design techniques for the AES non-linear byte substitution in the Quartus-II simulator for a Cyclone IV FPGA platform. We verified the output performance of various implementations of the S-box in terms of delay and size.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 28: S-box table [2]

$$\begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} 0\,1\,0\,1\,0\,0\,1\,0 \\ 0\,0\,1\,0\,1\,0\,0\,1 \\ 1\,0\,0\,1\,0\,1\,0\,0 \\ 0\,1\,0\,0\,1\,0\,1\,0 \\ 0\,0\,1\,0\,0\,1\,0\,1 \\ 1\,0\,0\,1\,0\,0\,1\,0 \\ 0\,1\,0\,0\,1\,0\,0\,1 \\ 1\,0\,1\,0\,0\,1\,0\,0 \end{bmatrix} \times \left[\begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \right]$$

Figure 29: S-box Matrix Computation in GF (2^8)

3.1 Rijndael S-Box Computation

S-box treats the values as a polynomial in Galois Field in factor 2^8(GF (2^8)) [1-2, 21-27], irreducible polynomial uses $x^8 + x^4 + x^3 + x^1 + 1$. It's calculation basically involves two steps:

- The inverse multiplication: derived from multiplicative inverse over GF (2^8). "00" is mapped to itself.
- The affine transformation: applying the affine (on GF (2^8)) transformation.

Using Rijndael's finite field for affine transformations the following expresses it as an equation for an input vector signal "X".

$$GF(2^8) = \frac{X}{(x^8+x^4+x^2+x+1)} \qquad \text{(Equation 3.1.1)}$$

Equation 1.1 shows the affine transform in Galois Field (2^8) as a function, with the characteristic irreducible polynomial as the denominator. This polynomial is represented in hardware as the binary string "100011011" for Boolean addition operations in the algorithm. The matrix representation of this function for S-box is shown in Figure 28. The input signal "b[7:0]" represents the 8-bit multiplicative inverse vector which undergoes Boolean XOR and addition operations to obtain the S-box output vector "a[7:0]." Figure 29 has various equations displayed in Boolean logic for quick implementation that represent the vector computation that occurs within the Rijndael multiplication and are shown in sequence as equations below.

$$s = b \oplus (b \ll 1) \oplus (b \ll 2) \oplus (b \ll 3) \oplus (b \ll 4) \oplus 63_{16} \qquad \text{(Equation 3.1.2)}$$

$$s_i = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus c_i \quad \text{(Equation 3.1.3)}$$

$$s = \left(b\ x\ 31_d\right) mod\ 257_d \oplus 99_d \quad \text{(Equation 3.1.4)}$$

However, the Rijndael S-box is not economical because it is based on the LUTs and uses more resources during implementation. To achieve high throughput and low power consumption, many literatures have proposed S-box hardware lookup table implementations [21-27]. In this chapter, we focused on how to efficiently implement pipeline technology utilizing FPGA platform to make S-box fast and verify the performance of various implementations of S-box in terms of latency and size.

3.2 Design Methods and Discussion

This research proposes three unique new designs based on restricting the way the S-box is implemented. The baseline implementation is a 256-line Look-Up-Table (LUT) that is conducted using sequential logic. Using the LUT in the design logic can significantly impact the amount of logic elements (LE's) that get used up by the FPGA. This method of adopting the basic principles of Shannon's expansion theorem achieves a logical design, which a greater number of levels, reduces the space requirement of the LUT, and lowers the implementation cost. This optimization technique reduces the complexity of the S-box module which results in fewer arithmetic operations. As it simplifies table indexing, it additionally consumes less power. This design, along with the reduced number of iterations, significantly lowers the delay for the algebraic and matrix operations. The designs are shown in the figures below and depict how the

S-box is segmented to create smaller LUT's combined with multiplexer (MUX) logic for the correct output selection.

Baseline:

The baseline Verilog HDL module implements the Substitution Box as a single LUT. The Register Translation Language (RTL) shows the baseline LUT module in Figure 30. This module is a direct implementation of the standard Rijndael Substitution Box shown in Figure 2, where an 8-bit input vector corresponds to a specific 8-bit output vector. This module relies on sequential logic, which consumes both a significant amount of hardware Logic Elements (LE's) and processing time from input to output.

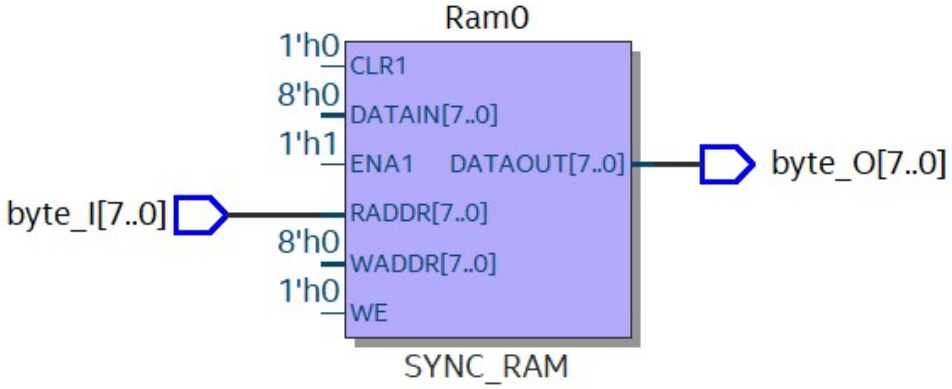

Figure 30: Baseline RTL

Design 1 (Row Column Parallelization of S-Box):

Design 1 aims to parallelize the Substitution Box by determining each corresponding 4-bit output in the same amount of clock cycles. This approach for the Substitution Box focuses on

separating the rows and columns of the 256-byte LUT (Figures 31-33). Each value in the row and column of the Substitution Box uses a nibble of the input byte-wise vector to determine the corresponding substitute value. This design utilizes 2 LUT's for each nibble (HIGH and LOW) of the input byte-wise vector and generates 16 possible output values that feed into a 16-to-1 Multiplexer (MUX) which uses the opposite nibble to select the correct 4-bit output. The module then concatenates the output of each MUX to create the correct corresponding output byte-wise vector.

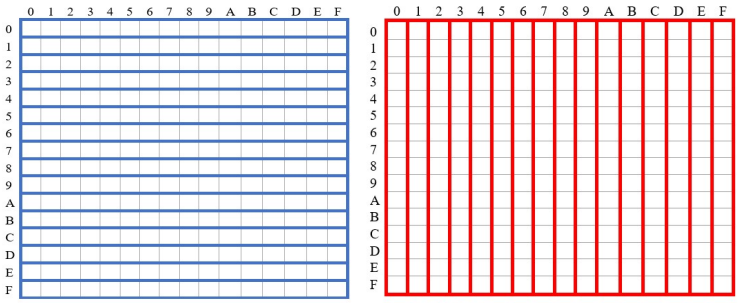

Figure 31: Design 1 S-box Segmentation

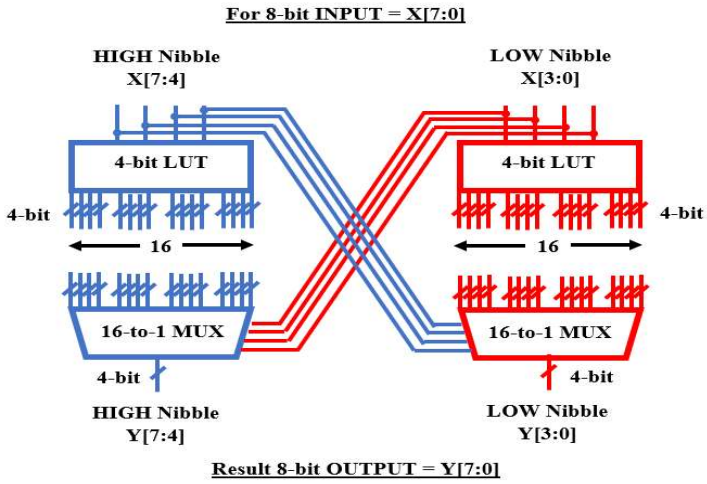

Figure 32: Design 1 Module Map

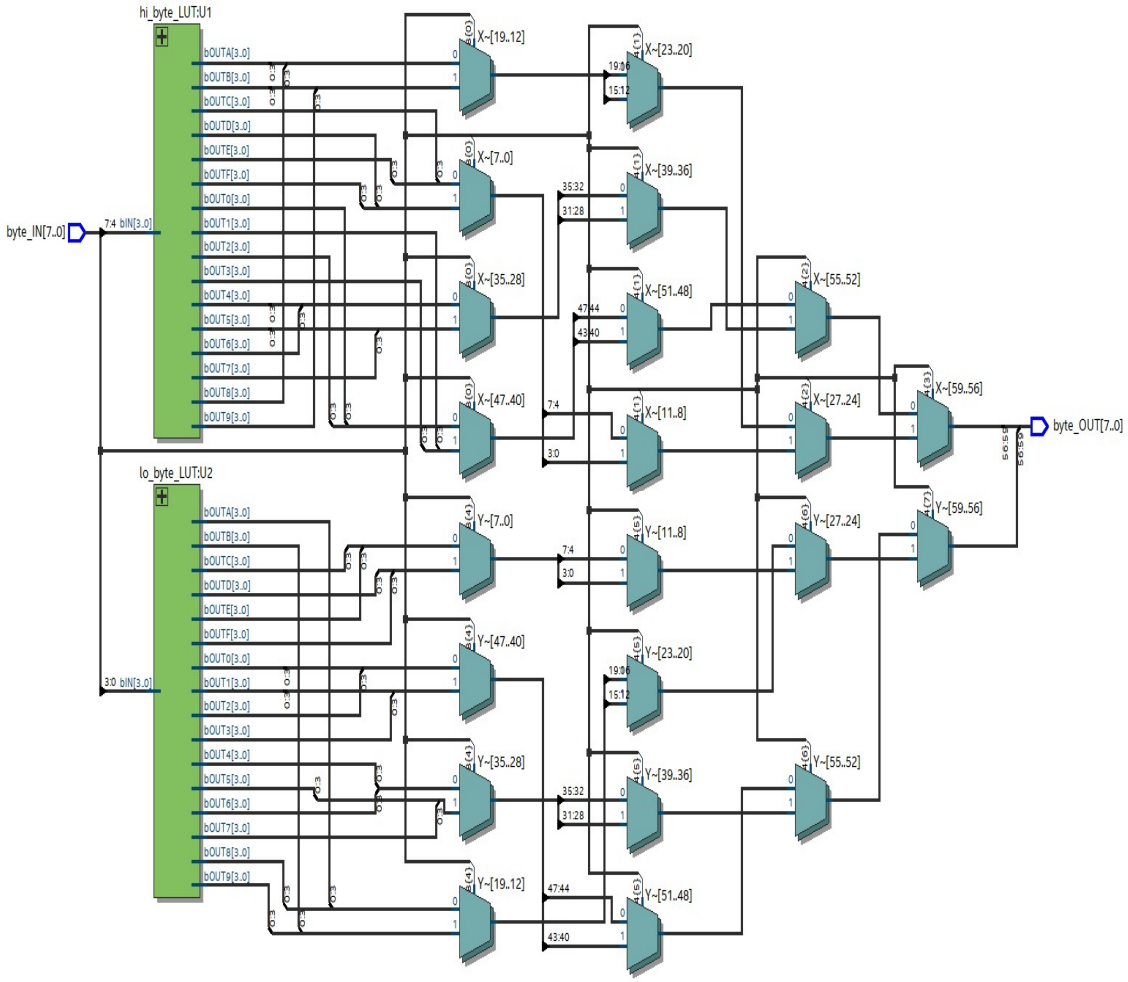

Figure 33: Design 1 RTL

Design 2 (Expanded RC Parallelization of S-Box):

Design 2, shown in Figures 34-36, aims to extend the technique of Design 1 by creating 4 LUT's instead of 2 for further parallelization. This design further segments the LUTs required to compute the corresponding substitute value. Through this LUT segmentation extension of the baseline Rijndael LUT S-Box, Design 2 attempts to produce the correct byte-wise output while increasing its LUT variable search, compared to Design 1, in the same clock cycle.

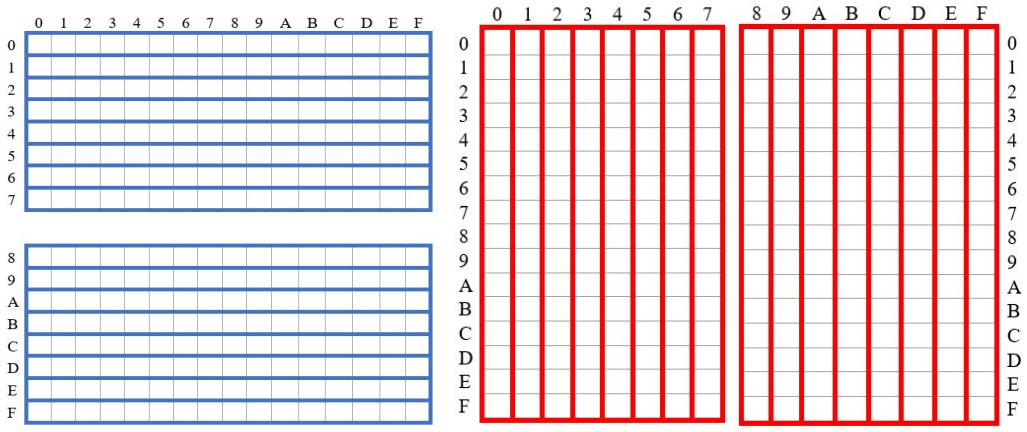

Figure 34: Design 2 S-box Segmentation

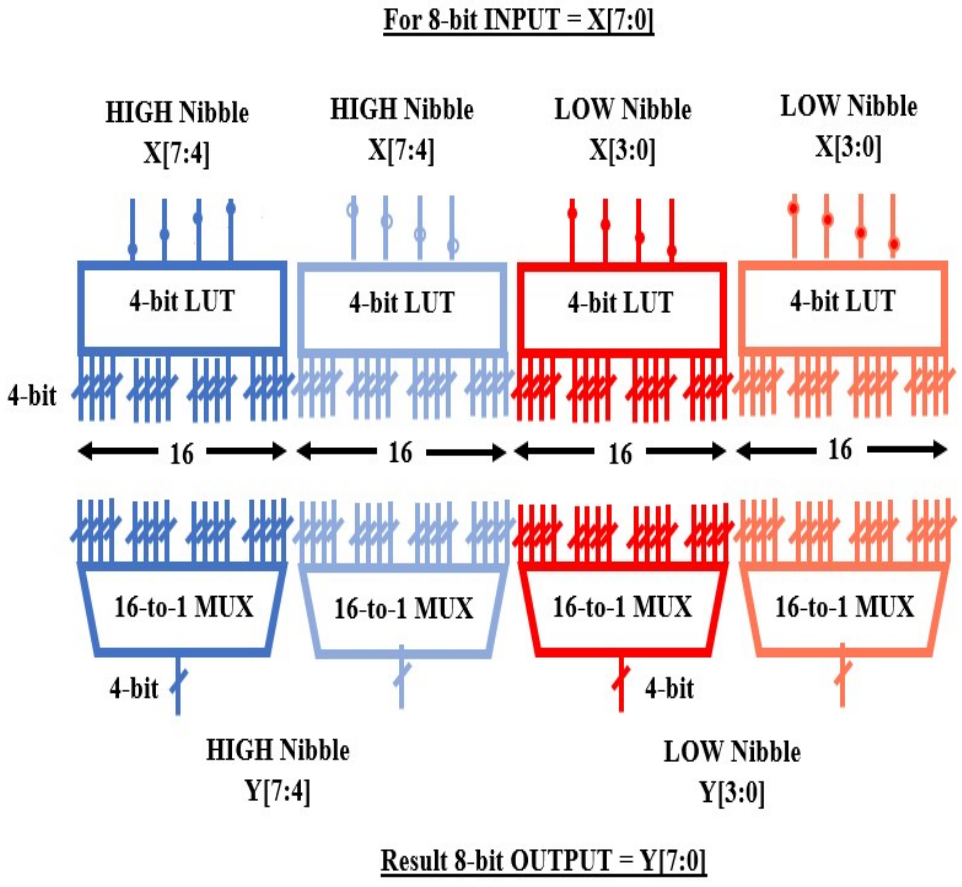

Figure 35: Design 2 Module Map

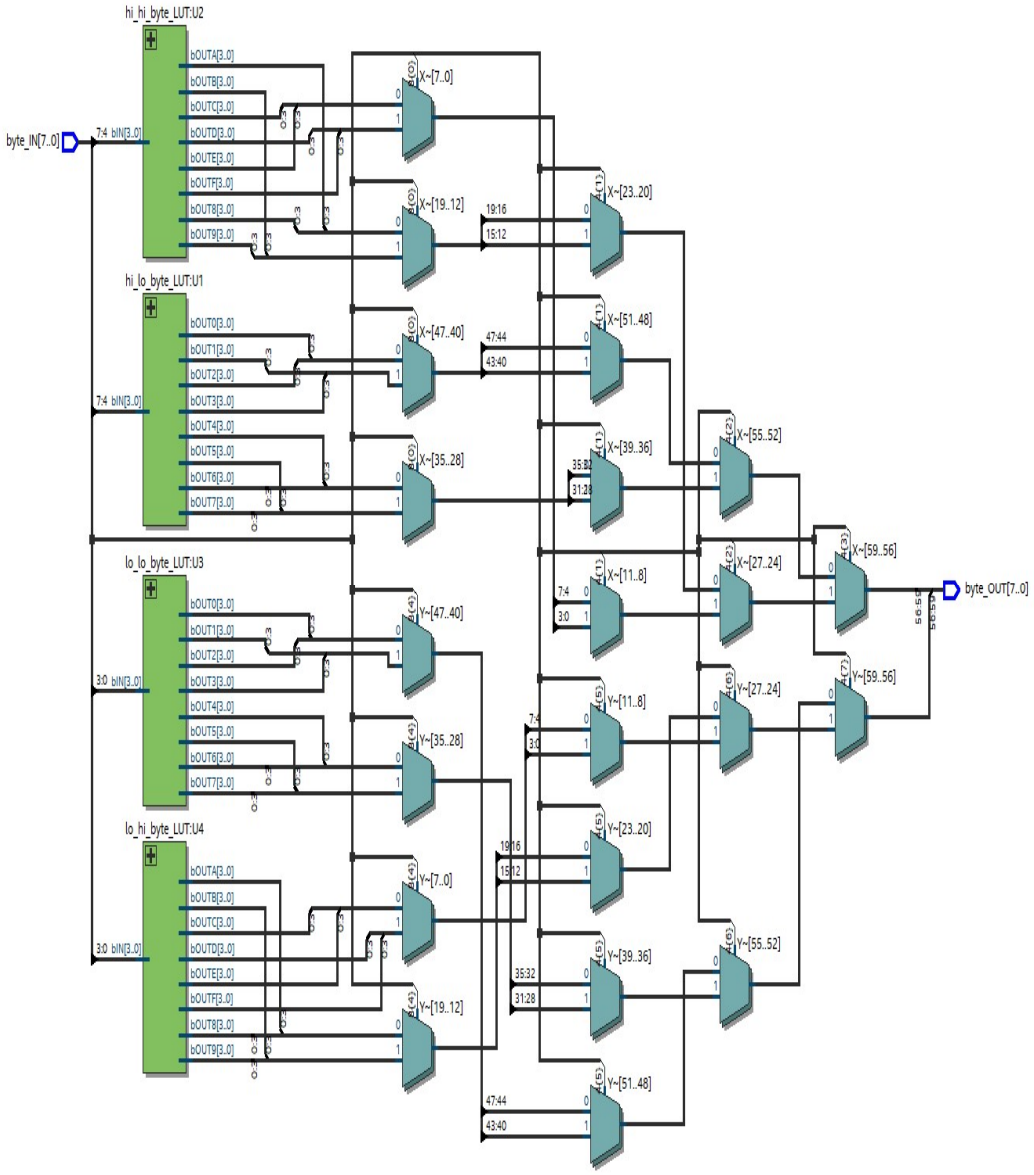

Figure 36: Design 2 RTL

Design 3 (Reduced LUT Parallization of S-Box): Design 3, shown in Figure 37-38, aims to parallelize the Rijndael S-Box by creating size reduced LUT's from the standard Substitution Box, shown in Figure 27. The standard Substitution Box is separated into 4 LUT's

one for each 8-byte by 8-byte quadrant. The byte-wise input vector goes through each of the 4 LUT's with only a single LUT generating the correct output vector while the others produce a logic zero. The output of all LUT's are then fed into a 4-to-1 byte-wise adder that combines all values to determine the output 8-bit substitution value.

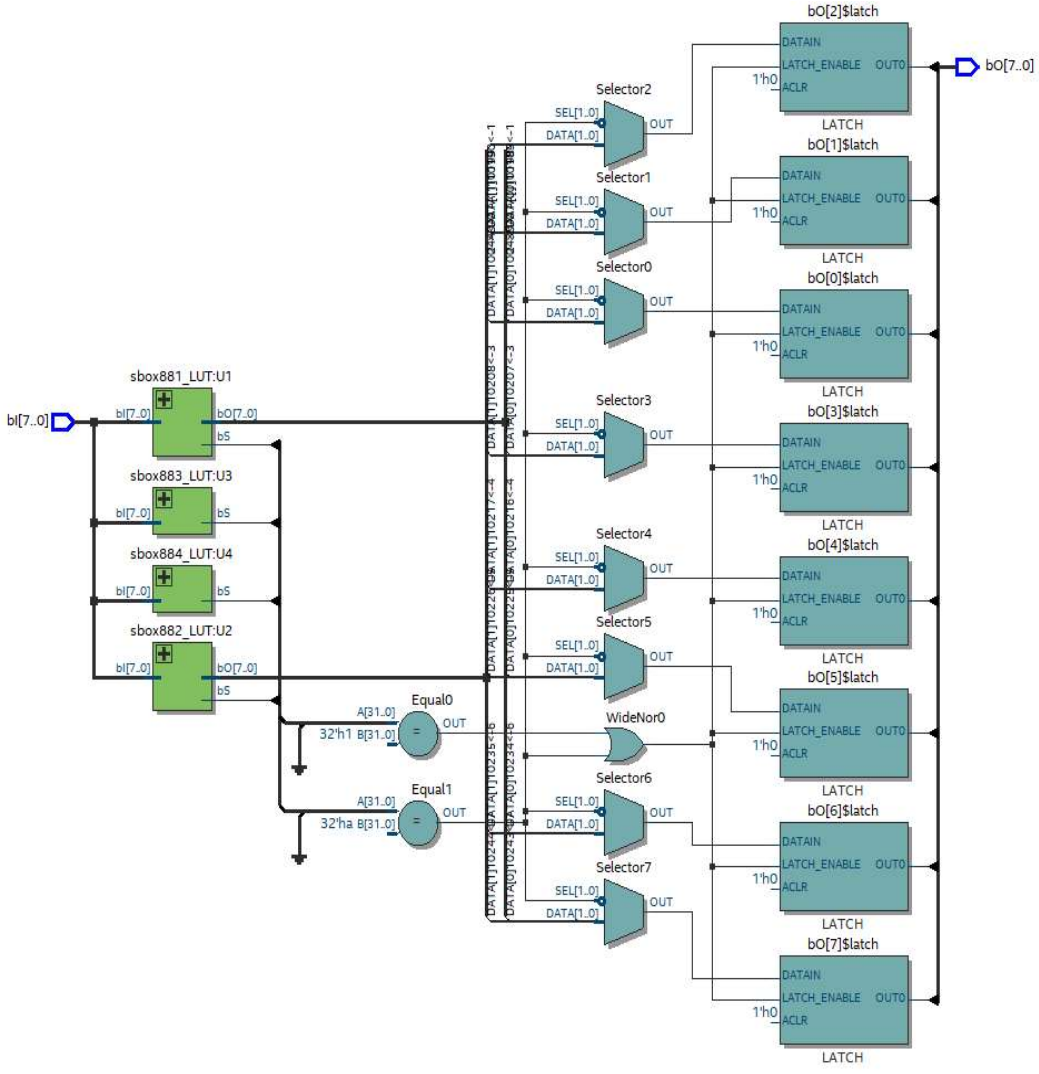

Figure 37: Design 3 RTL

Design 4 (Further Re. LUT Parallelization of S-Box): Design 4 aims to further reduce the LUT parallelization technique of Design 3 by creating smaller LUT's from the segmented quadrants. This approach extends the segmentation of the Rijndael S-Box by four times compared to Design 3. The aim of Design 3 and 4 is to parallelize the process of determining the correct output value by reducing the memory consumption of the sequential logic used to generate the LUT's. The use of an adder to determine the output increases the LE cost of the module but attempts to significantly reduce the processing delay.

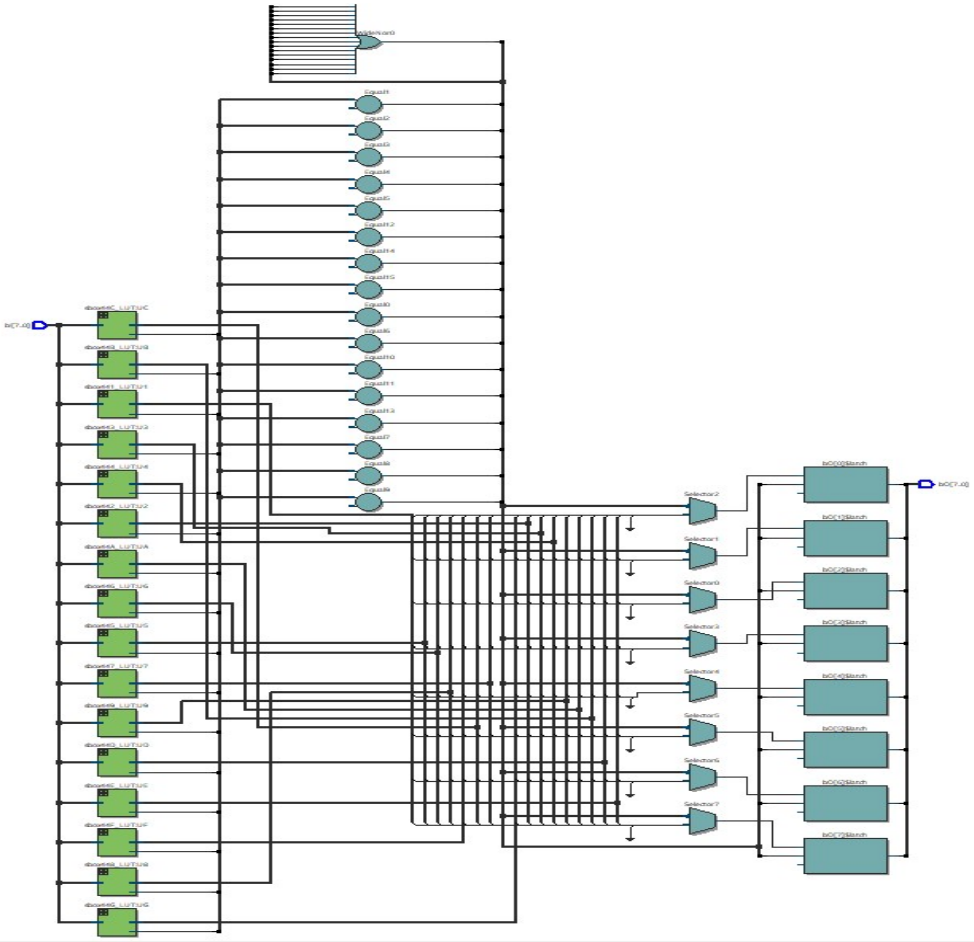

Figure 38: Design 4 RTL

Design 5 (Shannon's Expansion of S-Box): Design 5, shown in Figure 39, uses Shannon's expansion to reduce the processing time and cost consumption of the Substitution Box. This design uses the 8th and 7th MSB's (most significant bits) of the input byte-wise vector to determine the correct output substitute value through MUX logic. The input to the MUX logic includes four 6-byte x 6-byte LUT's that represent the possible outcomes that could be selected based on all variations of the 8th and 7th MSB such as "00", "01", "10", and "11". This approach, like the previous designs, aims to not only increase the searching process on the same clock cycle but reduce the LUT size for each search to achieve low LE cost and faster throughput.

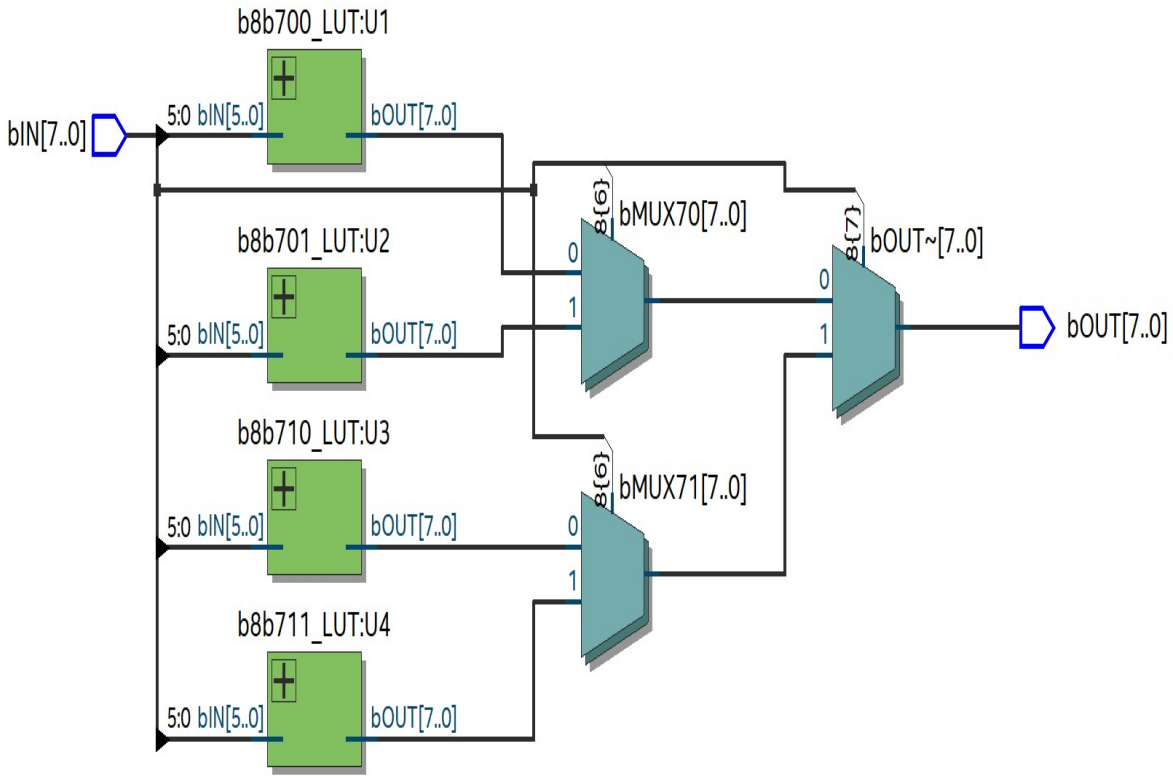

Figure 39: Design 5 RTL

3.3 Result and Analysis

The implementation of all Substitution Box designs was simulated on the Cyclone IV FPGA using NIST test vectors; the results are shown in Table 5. The baseline design is the initial direct implementation of the Rijndael Substitution Box from [1-2]. The baseline generated an average of 11.41 nanoseconds of delay while consuming 208 LE's. The results from the first implementation, displayed in Table 5, showed an increase in both average delay by 0.11 ns for design 1 and 0.52 ns for design 2. Design 3 showed the most efficient output compared to the baseline LUT; design 3 was able to generate the correct output 1.08 ns faster and with a 31.3% decrease in LE hardware consumption.

The Substitute Bytes module is repeated a total of 200 times, shown in Table 6, in the final implementation of AES-128. Compared to the other submodules that make up the AES-128 algorithm, only the byte-wide 2-input XOR (XOR2) module was instanced more than Substitute Bytes by a total of 120 times (Table 6). To improve the AES-128 algorithm in efficiency and throughput, it is crucial to target the most repetitive submodule since improvements in its implementation would be magnified. However, the XOR2 submodule is simply an expanded XOR gate that requires significantly less LE consumption than the Substitute Bytes module which is a 256-byte size LUT. The I/O delay and LE consumption for design 3 would be multiplied by the module count and would reduce the total delay of the AES encryption algorithm by approximately 200 ns compared to the baseline. The concept for design 3 and the other implementations, show the effectiveness of pipelining in processor operations. By segmenting the LUT into separate modules, each can be executed on the same clock cycle and reduce the overall delay of the process.

This was expected to bring a trade-off between the hardware consumption to reduce the delay however design 3 was able to improve on both areas.

Many literatures have proposed S-box hardware lookup table implementations [21-27]. The logic design using the basic principles of Shannon's expansion theorem is achieved. By comparing the suggested designs to other reported techniques, analysis of the results indicates that the proposed design 3 is capable of significantly outperforming the other four designs in terms of delay and area as measured by the simulation. From the simulations of the proposed designs, we observed that the throughput can be increased by reducing the delay of the critical path taken by the input byte-wise vector specifically in the LUT segment. By targeting one of the most utilized modules in the AES computation scheme, the efficiency of the overall implemented algorithm can be improved on. The achieved lower LE cost and faster throughput will multiply based on the number of instances of the submodule included in the final design of AES-128 and can lead to improvements in the extended iterations of AES such as AES-192, AES-256, and AES-512[30-40].

Table 5: Design Parameter Comparison

Design	Average Delay (ns)	Logic Elements	Virtual Pins
Baseline	11.41	208 (<1%)	16 (3%)
Design 1	11.52	208 (<1%)	16 (3%)
Design 2	12.33	294 (<1%)	16 (3%)
Design 3	10.73	65 (<1%)	16 (3%)
Design 4	18.41	352 (<1%)	16 (3%)
Design 5	11.36	208 (<1%)	16 (3%)

Table 6: Total Module Count for AES-128

AES-128 Top Level Module				
Sub module	**Instances**	**Peak Virtual Memory (MB)**	**Pins/529**	**Logic Elements /114,480**
Substitute Bytes	200	4788	16 (3%)	280 (<1%)
Mix Column	144	4770	16 (3%)	3 (<1%)
XOR2	320	4782	24 (5%)	8 (<1%)
XOR4	144	4770	40 (8%)	8 (<1%)
Increment Bytes	10	4789	70 (13%)	857 (<1%)
Subkey Round	10	4769	288 (54%)	128 (<1%)

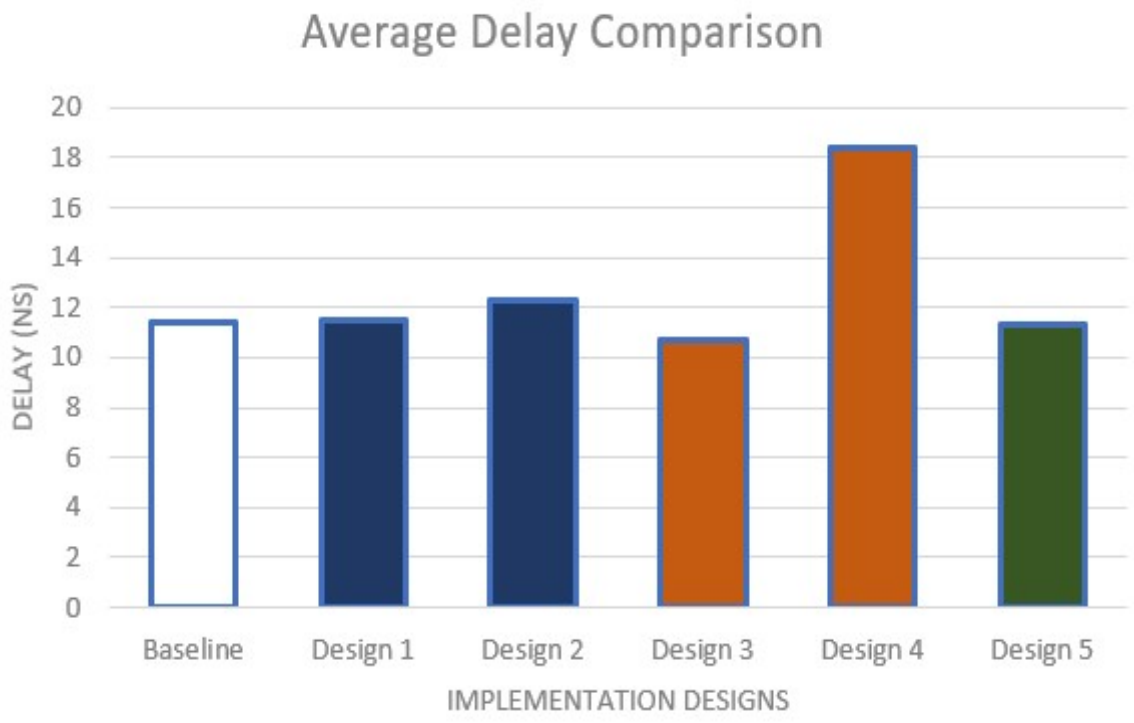

Figure 40: Average Delay (in nanoseconds) Comparison

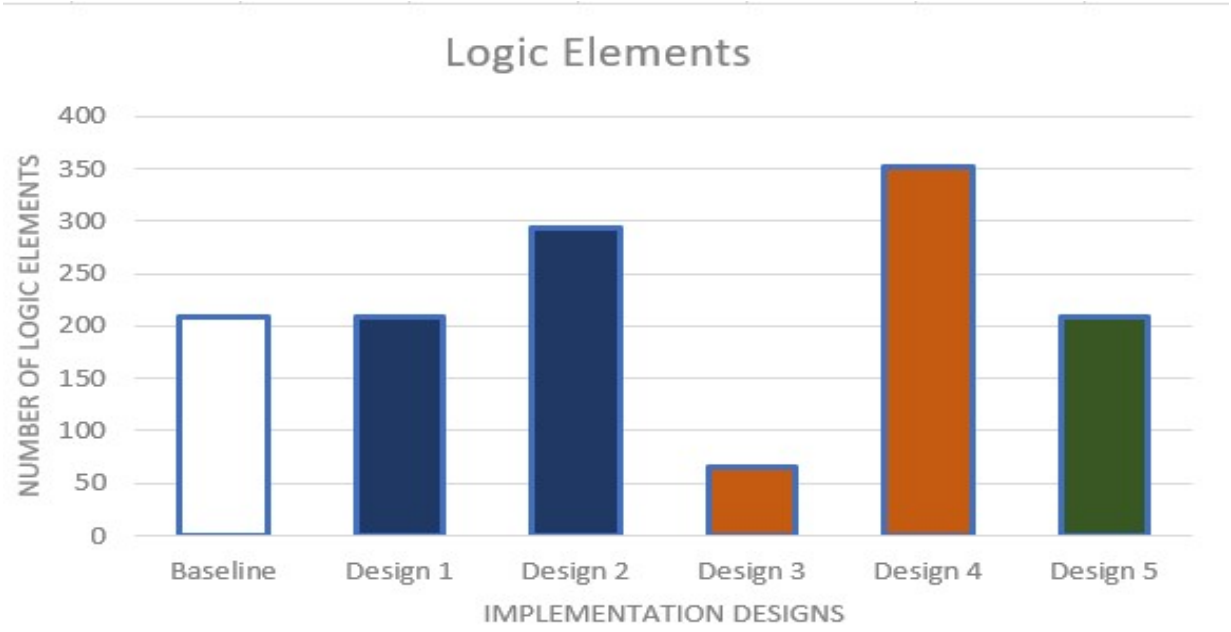

Figure 41: Comparison of Number of Logic Elements used

3.4 Conclusion

This chapter discusses the design and simulation of the AES non-linear byte substitution techniques[45]. By simulating the designs in the Quartus-II software on a Cyclone IV FPGA platform, we verified the output performance of various implementations of the S-box module with pipelining techniques in terms of delay and size as shown in Figure 40-41. From the simulations of the proposed designs, we observed that the throughput can be increased by reducing the delay of the critical path taken by the input byte-wise vector specifically in the LUT segment. The techniques proposed in our most optimized design 3 shows us how to achieve high throughput and low power consumption using the effectiveness of pipelining in processor operations.

CHAPTER IV

PERFORMANCE COMPARISON OF AES VARIANTS USING HARDWARE IMPLEMENTATIONS ON FPGA

AES has been extensively analyzed and is now widely used in modern-day technologies. It is a symmetric encryption algorithm that has a minimum input data block size of 128-bits that undergoes a series of permutations, substitutions, and digital logic operations over several rounds. This chapter evaluates AES-128, AES-192, AES-256, and AES-512 on various parameters and compares their hardware performance through I/O delays when implemented on the Cyclone IV Field Programmable Gate Array (FPGA). Through this comparison, the research presented provides a detailed scope at the complexity versus hardware consumption cost for all iterations of AES to conclude the most efficient implementation.

4.1 Rijndael Key Expansion Computation

The AES key expansion computation takes a four-word (16-byte) key as its input shown as Figure 42. and produces a linear array of 44/52/60/68 words (176/208/240/272 bytes) to generate enough exclusive keys to accommodate for the respective round size 10/12/14/16 of AES. The key is copied into the first four words of the extended key. The rest of the extended key is filled with four words at a time. Each added word depends on the previous word and the word is returned in four positions. In three of the four cases, a simple XOR was used. For words in the ‘w’ array whose position is a multiple of 4/6/8/16, a more complex function is used. Figure

43 illustrates the generation of the extended key, using the symbol "⊕" to represent the complex function.

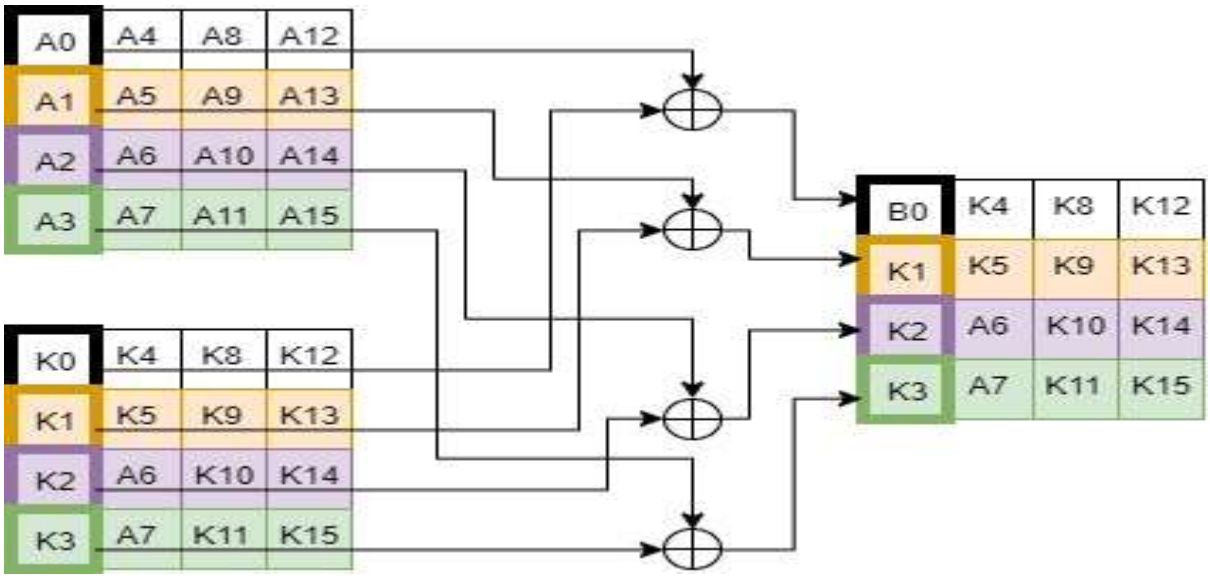

Figure 42: Add Round Key

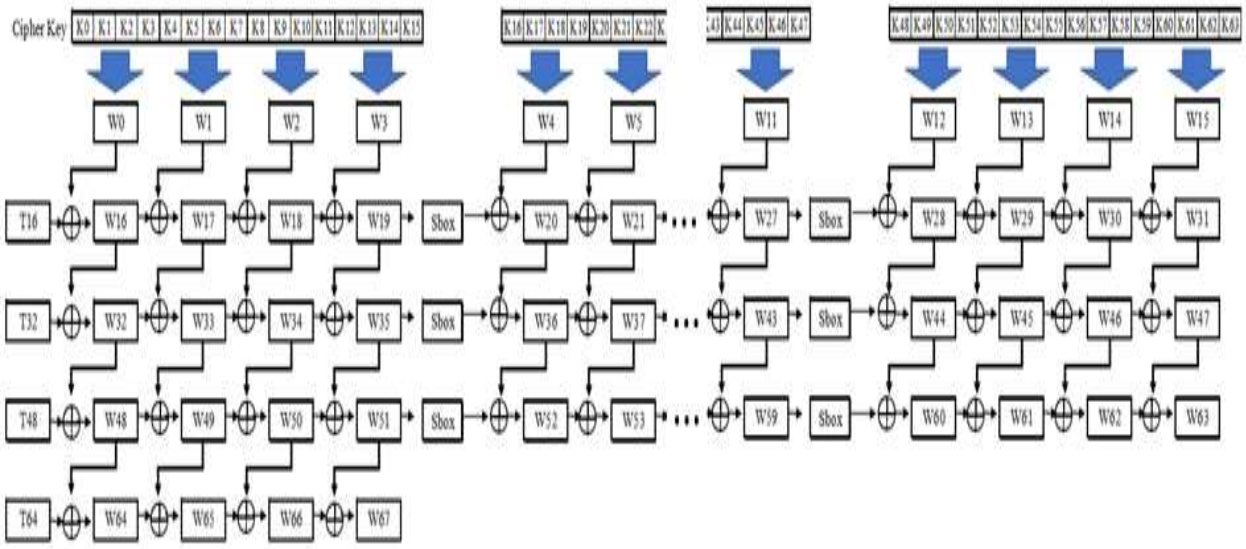

Figure 43: AES-512 Key Expansion

From the AES algorithm [1-2] we can find that the main difference between AES-128, AES-192 and AES-256 except for the different rounds is key expansion. As we can see from the Figure6~8, the Key Expansion routine for AES-256 is slightly different than for AES-128 and AES-192. Here, we chose AES-512 as an example to illustrate how to generate the key needed for each round base on the Key Expansion routine for AES-256.

The process of computing 512-bit round keys are depicted in the Figure 9 and algorithms are as follows:

- The elements of the original input key (512 bits) arranged in words and arranged from the most significant byte to the least significant byte key length are divided into four words, each word being 32 bits of equal size, thus Six words are formed in each line.
- All subkeys are stored in the key extension array, the elements are W[0], W[1],..., W[67] because there are 17 subkeys for maintaining 16 rounds and 4 iterations. The first subkey Key0 is obtained from the first word of the original input key from AES, and the Key0 is copied to the first four elements of the key array [W0, W1, W2, W3]. The Key1 is copied to the second four elements of the key array [W4, W5, W6, W7], the Key2 is copied to the first four elements of the key array [W8, W9, W10, W11], the Key3 is copied to the first four elements of the key array [W12, W13, W14, W15], and the remaining subkeys Obtained by the steps defined below.
- All other element of the array is computed as follows:

  If (i mod 16) =0 then $W_i = T_i \oplus W_{(i-16)}$

Here T_i= {SubWord (RotWord($W_{(i-1)}$))}$\oplus$ $Rcon_{(\frac{i}{16})}$

If (i mod 16) ≠0 and (i mod 4) =0 then W_i = {SubWord ($W_{(i-1)}$)} $\oplus$ $W_{(i-16)}$

If (i mod 16) ≠0 but (i mod 4) ≠0 then W_i = $W_{(i-1)}\oplus$ $W_{(i-16)}$

- After calculating all the elements of the word Matrix [W0, W1, ... W67], we compute 17 subkeys from K0 to K12 by taking the first four words, so that Key0 = [W0, W1, W2, W3], Key1 = [W4, W5, W6, W7], ... Key17 = [W64, W65, W66, W67]. We can also see that we only use up to Rcon4. (For AES-128, we use up to Rcon10. For AES-192, we use up to Rcon8. For AES-256, we use up to Rcon7.)

4.2 Design Methods and Discussion

This research will focus on extending the FPGA implementation to the entire algorithm; furthermore, we also evaluate the overall performance for the different variants of AES such as AES-128, AES-192, AES-256, and AES-512. The work in this thesis aims to speed up AES encryption overall and reduce processing delays. Comparisons are conducted on both a theoretical basis and through timing simulations on the Intel Quartus II software to reveal the implication of increased complexity on the hardware performance of AES in terms of Logic Elements (LE's) and transport delays.

4.3 Result and Analysis

According to the AES algorithm we mentioned in section 2, we can distinguish each step into several operations or functions. We used the Quartus II software to perform timing

simulations for each operation or function in every step, as shown in Figure 44~Figure 51. The following data presented is modeled after previous hardware performance comparison attempts mentioned in [3, 9-11,17-24].

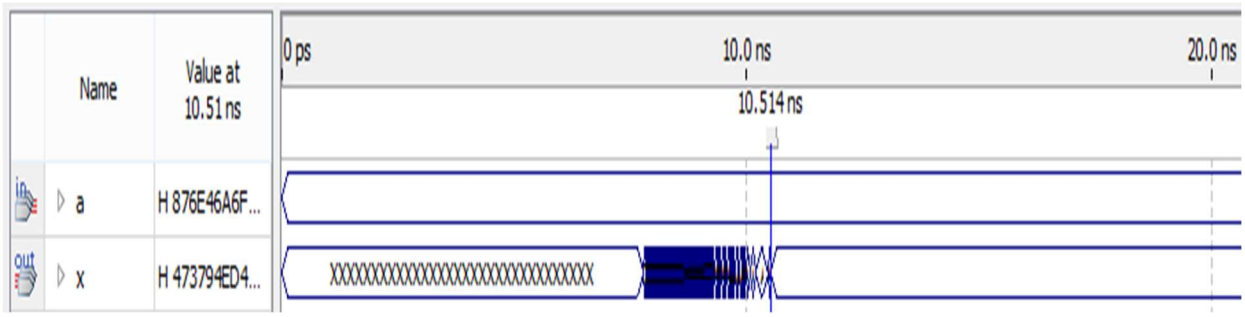

Figure 44: Delay of Mix Columns Module

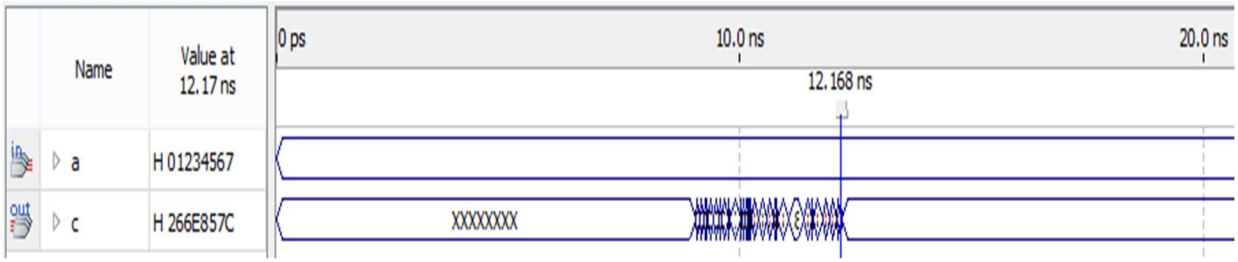

Figure 45: Delay of ShiftRows Module

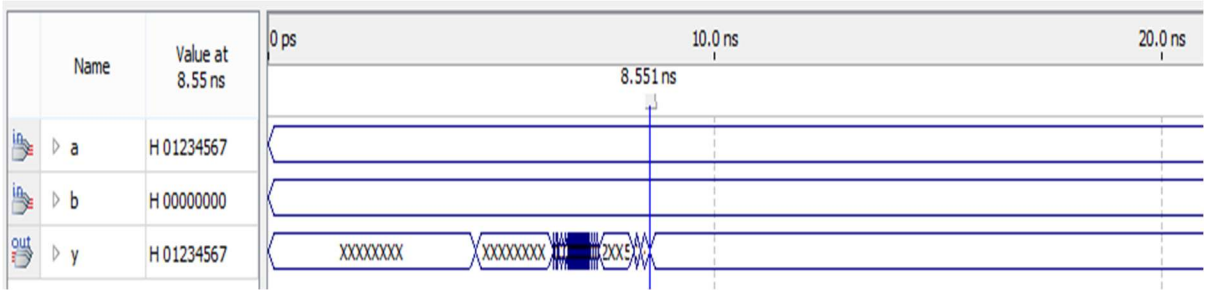

Figure 46: Delay of Key Module

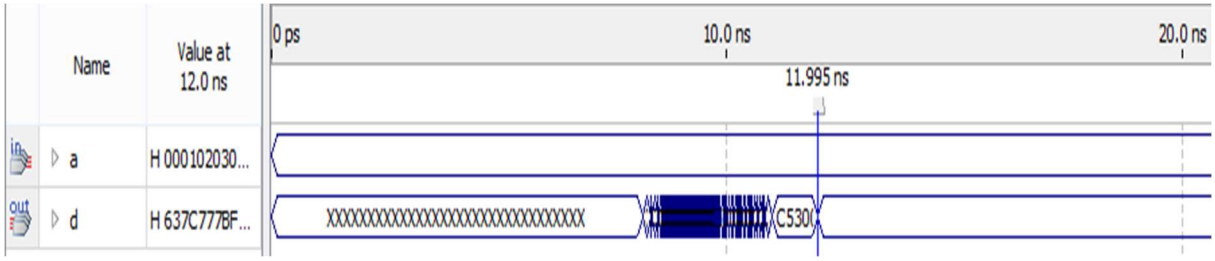

Figure 47: Delay of XOR32 Module

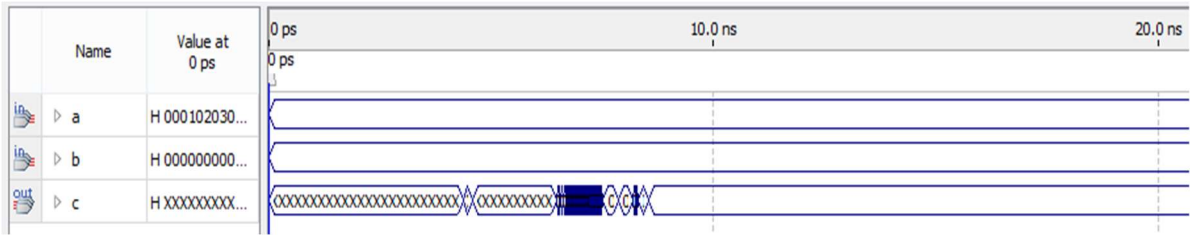

Figure 48: Delay of SBOX1 Module

Figure 49: Delay of Key Expansion Module

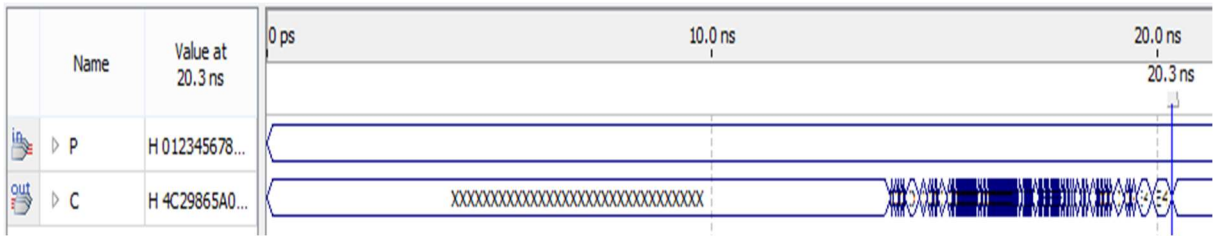

Figure 50: Delay of Round Module

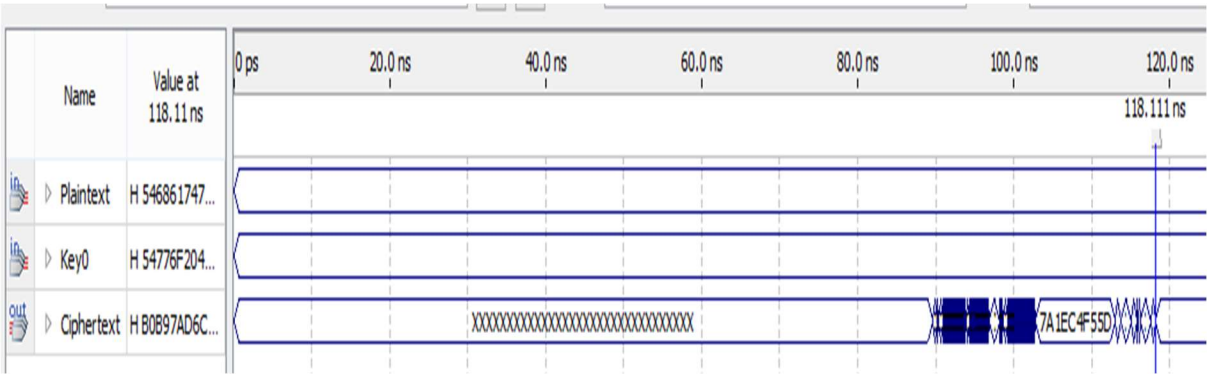

Figure 51: Delay of Encryption Module

The performance comparison is based on the timing simulations of the different key sizes for the AES variants. The calculation results are shown in the Table 7 and Table 8 and were conducted through scalar multiplication of the base processing time in nanoseconds for AES-128 and AES-192. The results show a linear trend where an increase in key size results in an increase in delay time for the plaintext encryption portion (Table 7); the overall encryption time is increased by approximately 14% from the 128-bit key to the 192-bit key, 29% to the 256-bit key, and 38% to the 512-bit key. In contrast, the results for the key expansion portion did not follow a similar linear trend; the overall key expansion time is increased by approximately 10% from the 128-bit key to the 192-bit key, and 19% to the 256-bit key as well as the 512-bit key. The data from the key expansion simulation revealed how the AES-256 and AES-512 variants shared a similar processing time despite AES-512 being computational stronger due to its extensive architecture.

Table 7: Delay of Plaintext Encryption

Function	Operation Type	Processing Time(ηs)		AES 128	AES 192	AES 256	AES 512
Sub Bytes	Substitution (S-Box)	12		10	12	14	16
Shift Row	Assign	7.75		10	12	14	16
Mix Columns	M2(Left Shift + XOR2)	6.84	10.51	9	11	13	15
	M3(XOR2)	7.02		9	11	13	15
	M4(XOR4)	8.83		9	11	13	15
Add Round Key	XOR2	7.02		11	13	15	17
Total Processing Time (ηs)				369.3	443.9	518.4	593.0
Processing Time Percent Increase Compared to AES-128 (%)					16.8	28.8	37.7

Table 8: Delay of Key Expansion

Function	Operation Type	Processing Time(ηs)		AES 128	AES 192	AES 256	AES 512
RotWord	Assign	6.31	12.17	10	12	14	16
SubWord	Substitution (S-Box)	12		10	12	14	16
XOR2(32bits)		8.51		50	54	59	56
Total Processing Time (ηs)				547.2	605.6	672	671
Processing Time Percent Increase Compared to AES-128 (%)					9.6	18.6	18.5

4.4 Conclusion

The results show how the encryption time varies with key size selection. It was not only found out from the AES algorithm but also from the actual simulation data, that when the key size increases, the encryption time for the four function modules of AES also increases. However, this increase is not proportional to the increase in key expansion processing time (Table 8). Despite having more rounds and increased complexity, AES-512 was found to be slightly computationally faster (or comparable) in processing time for key expansion than AES-256. When seeking to secure data confidentiality at the highest level, AES-512 offers this with only an 18% increase in processing time compared to the baseline variant AES-128 (which is not as secure as AES-512).

CHAPTER V

COMPARISON AND ANALYSIS

5.1 Avalanche Effect

Each of the encryption technique has its own strong and weak points. In order to apply an appropriate technique in a particular application we are required to know these strengths and weakness. Therefore, the analysis of these techniques is critically necessary. A desirable property of any encryption algorithm is that a small change in either the plaintext or the key should produce a significant change in the cipher text.

However, a change in one bit of the plaintext or one bit of the key should produce a change in many bits of the cipher texts. This property is known as Avalanche Effect. Avalanche Effect can be calculated by using above equation.

The performance of proposed algorithm is evaluated using Avalanche Effect due to one-bit variation in plaintext (before being mapped in various binary codes) keeping encryption key constant in a binary code [41].

Avalanche Effect is calculated for various combination of plaintext and encryption key by mapping them in various binary codes.

$$Avalanche\ Effect = \frac{Number\ of\ \boldsymbol{flipped}\ bits\ in\ ciphered\ text}{Number\ of\ bits\ in\ ciphered\ text}$$

The two characteristics of the symmetric encryption algorithm described in the system overview are confusing and spreading. The data in Table 9 and Table 10 are converted into a binary column as follows:

Table 9: Avalanche Effect of DES [2]

DES	
Ciphertext C1(binary)	1001 1000 1100 0110 0101 0000 0001 0111 1010 1001 1100 0001 1101 1101 0101 0010
Ciphertext C2(binary)	0110 1000 1011 1110 1010 0010 0000 0111 1001 1100 1101 0101 0010 0111 1111 1000

Table 10: Avalanche Effect of AES [2]

AES	
Ciphertext C1 (binary)	01010000 00000011 01110010 01011001 00100001 00110111 00110001 00011000 01010100 01111101 01001100 01101011 00010110 01001101 01001011 00110111
Ciphertext C2(binary)	01110001 01001011 00101101 00111011 00001110 01000100 01001100 00011111 01011000 00101111 00100001 00111001 01011110 01111000 01100100 01111000

We can observe that the confusion and diffusion of DES is not as good as AES. There are too many repetitions of the ciphertext of DES and the ciphertext after changing one bit. The changes in ciphertext 0 and 1 are relatively concentrated and not evenly dispersed. Therefore, it can be seen that the ciphertext complexity after the encryption algorithm is not enough, in terms of individual blocks. The avalanche effect data shows that the avalanche effect of DES is

relatively poor, while AES is closer to the theoretical value of 50%, and AES is the advanced encryption standard. Although DES encryption can be used, AES is definitely more secure than DES in terms of security. When the input (plaintext or key) to any cryptographic algorithm is changed slightly, then there must be significant change in the output. It is the most desirable property of any cryptographic algorithm is the avalanche effect. It was a term coined by Horst Feistel. It accounts for the randomization in the algorithm or can be thought of as a metric for diffusion & confusion. Normally, a change of about 50% is desirable as it makes the algorithm truly random.

5.2 Timing Simulation For AES-128, AES-192, AES, 256 and AES-512 in ECB Mode

Based on my previous works in chapter Ⅱ, Ⅲ, and Ⅳ, I can build the most efficiency modules of AES variants in different operation modes [42-44]. In this session, I discuss the performance of AES-128, AES-192, AES-256 and AES-512 in ECB mode. Via the Quartus II, we can find the device we are using, the processing time, and the total logic elements from the compilation report. We can also see the block diagram in register level. By running the timing simulation, we can find out the delay for each experiment.

5.2.1 Timing Simulation For AES-128

In order to observe the performance changes, we run the simulation and recorded the results of each round as below.

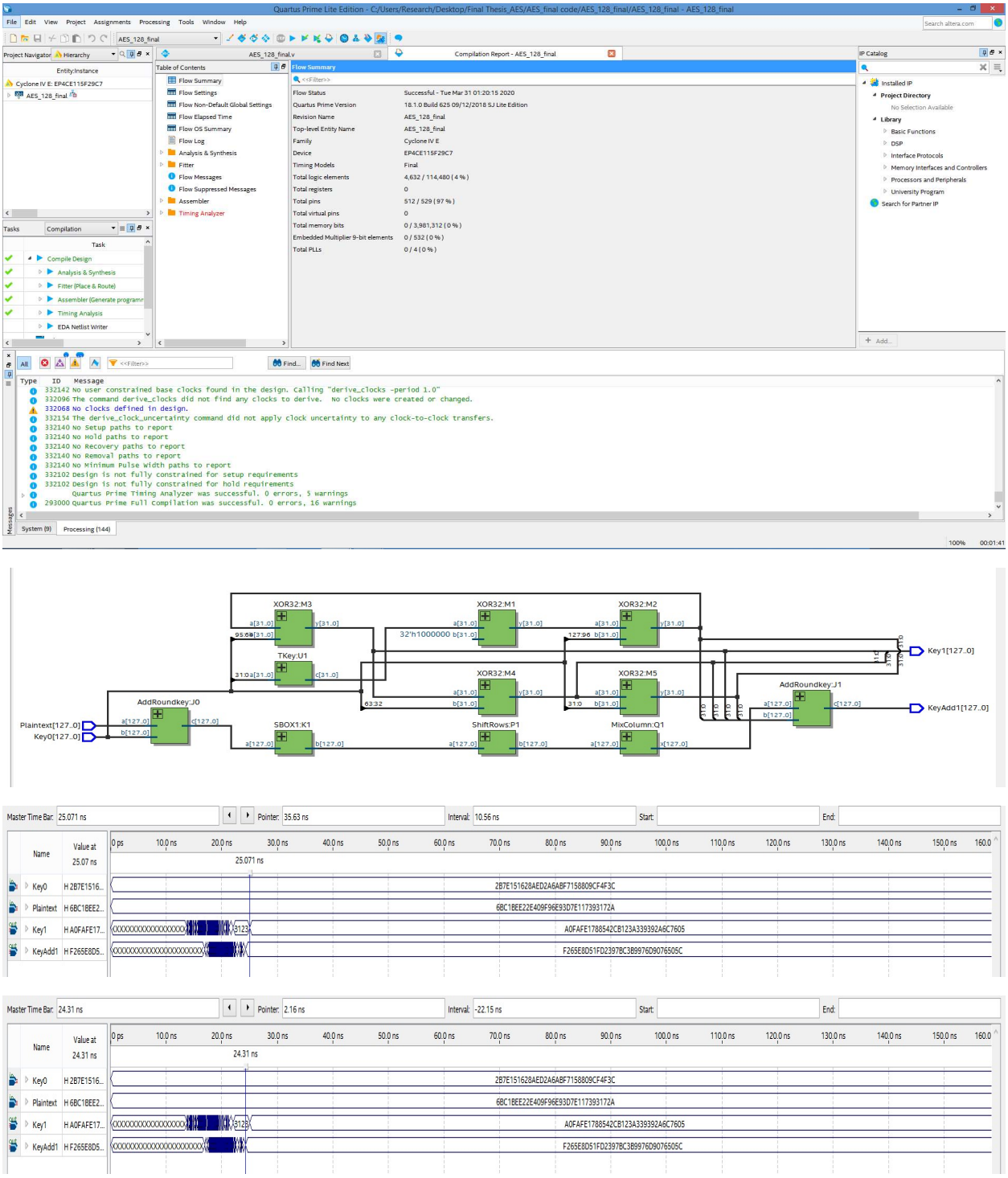

Figure 52: Timing Simulation For AES-128 after Round 1

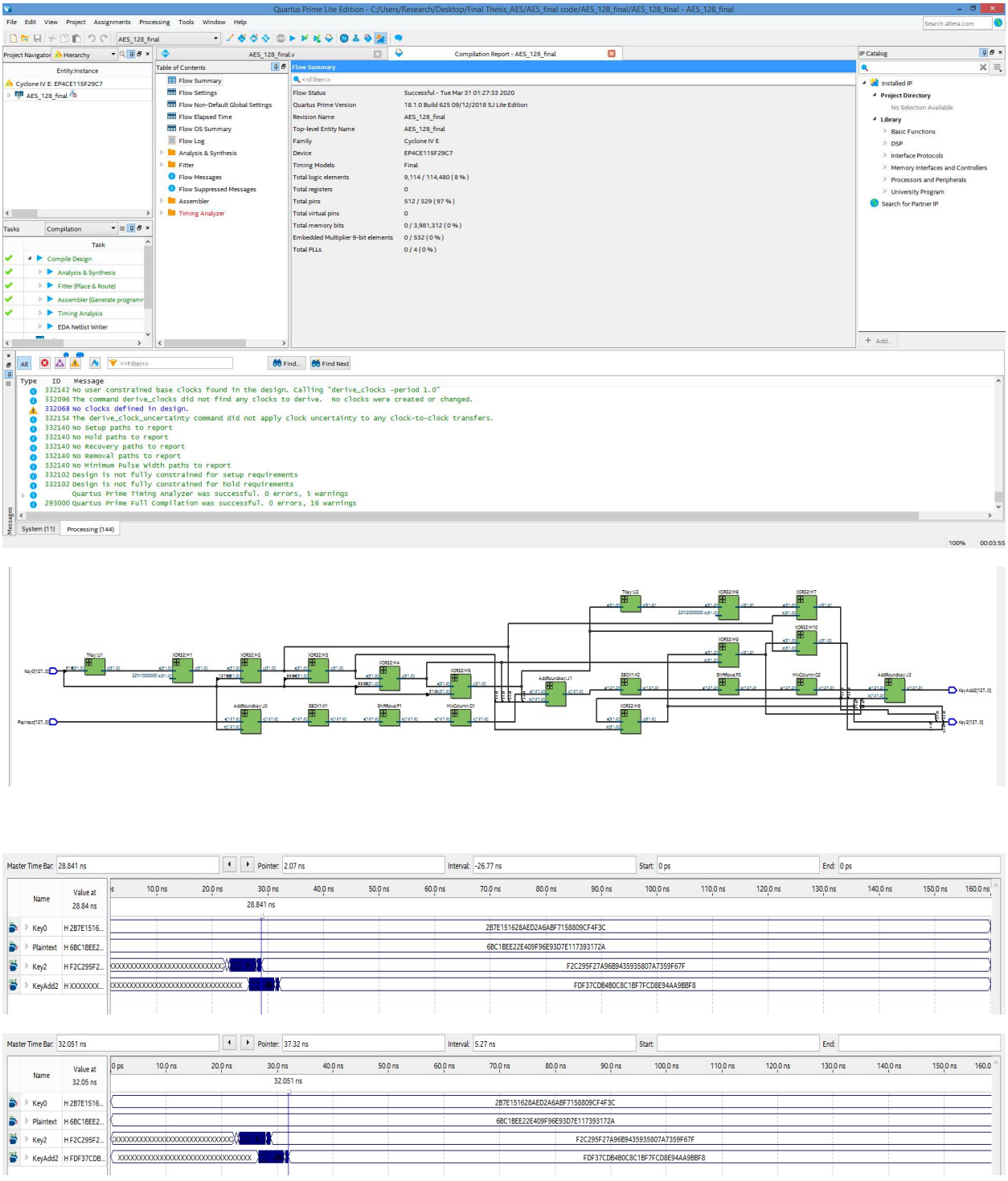

Figure 53: Timing Simulation For AES-128 after Round 2

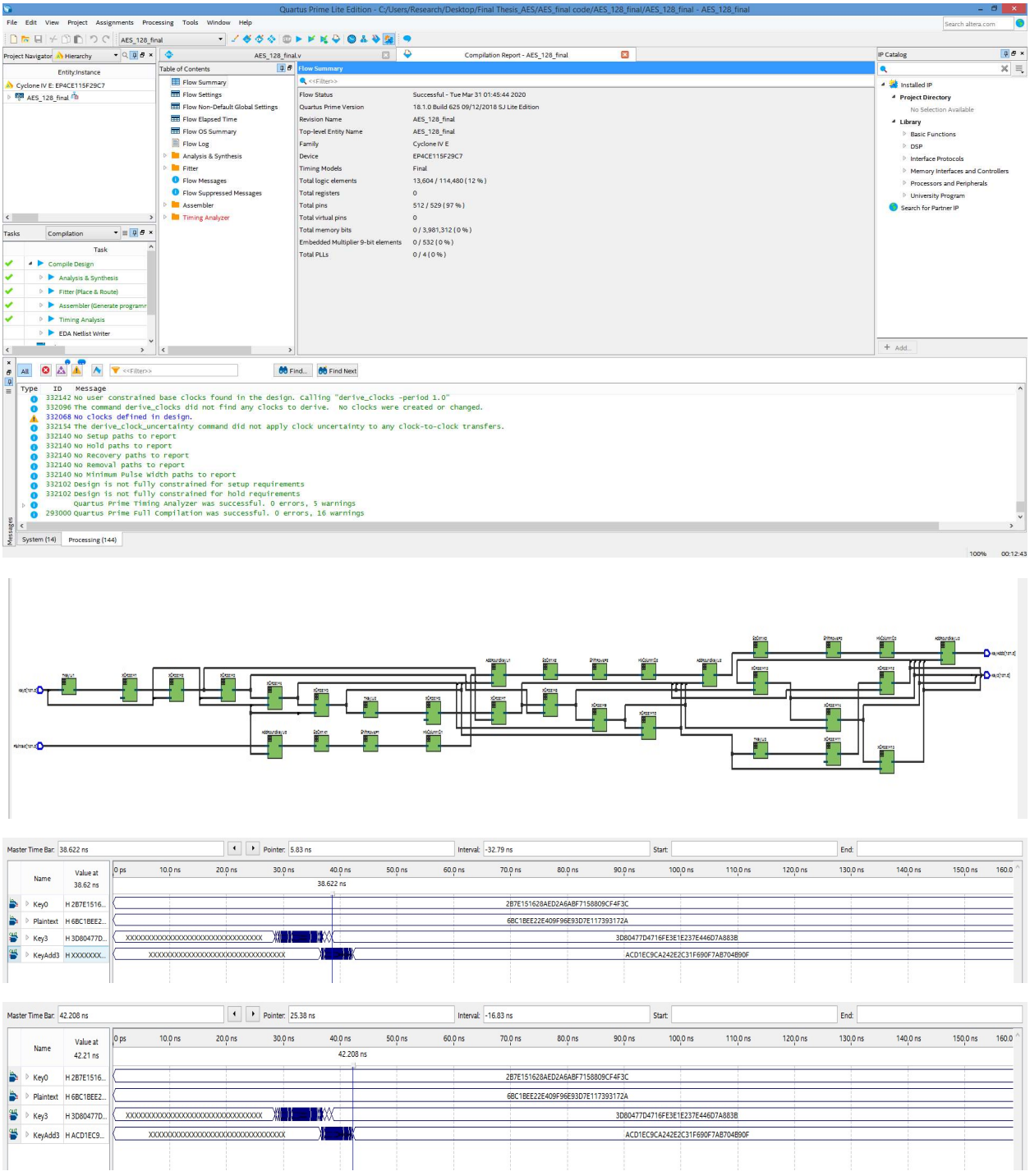

Figure 54: Timing Simulation For AES-128 after Round 3

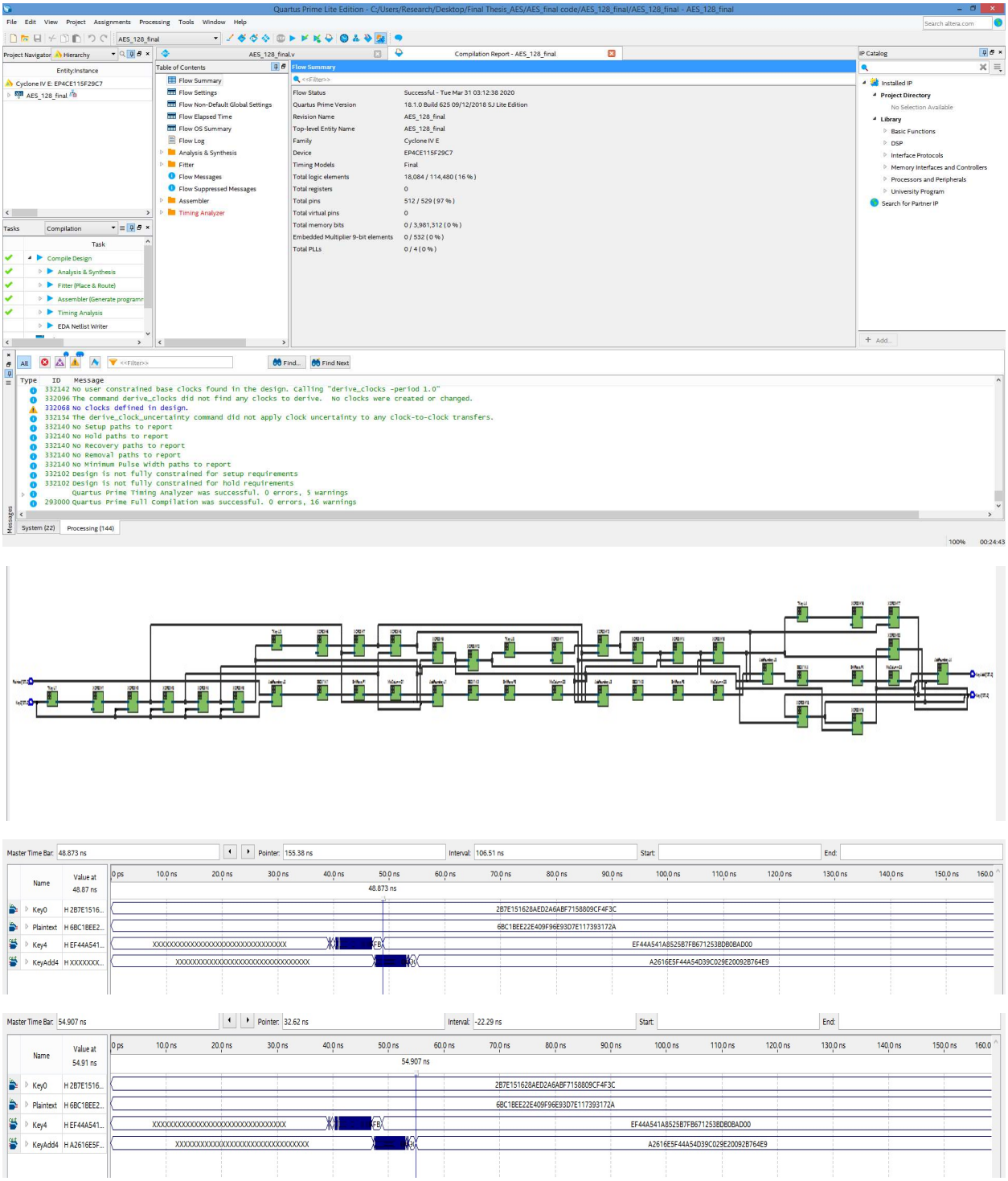

Figure 55: Timing Simulation For AES-128 after Round 4

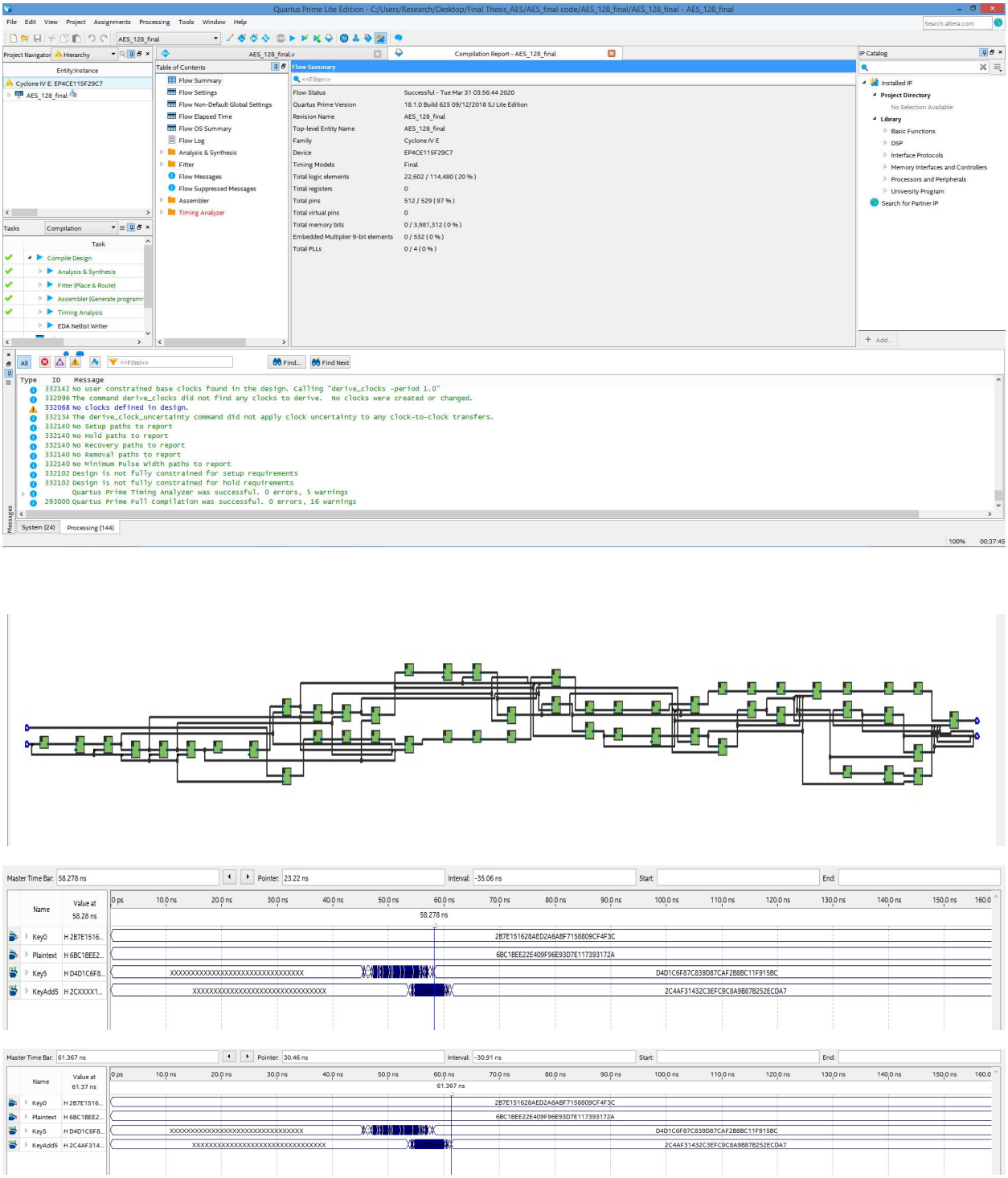

Figure 56: Timing Simulation For AES-128 after Round 5

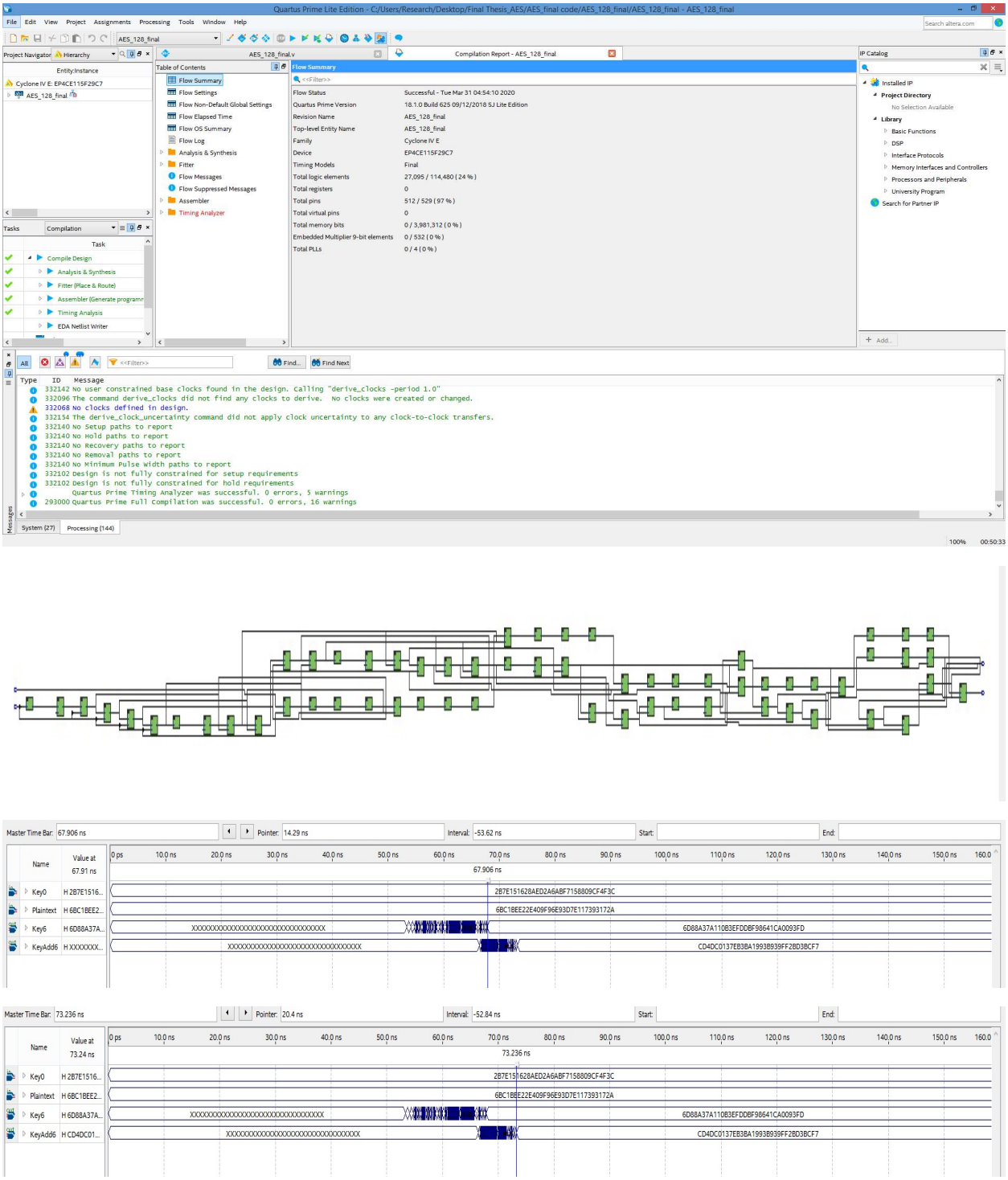

Figure 57: Timing Simulation For AES-128 after Round 6

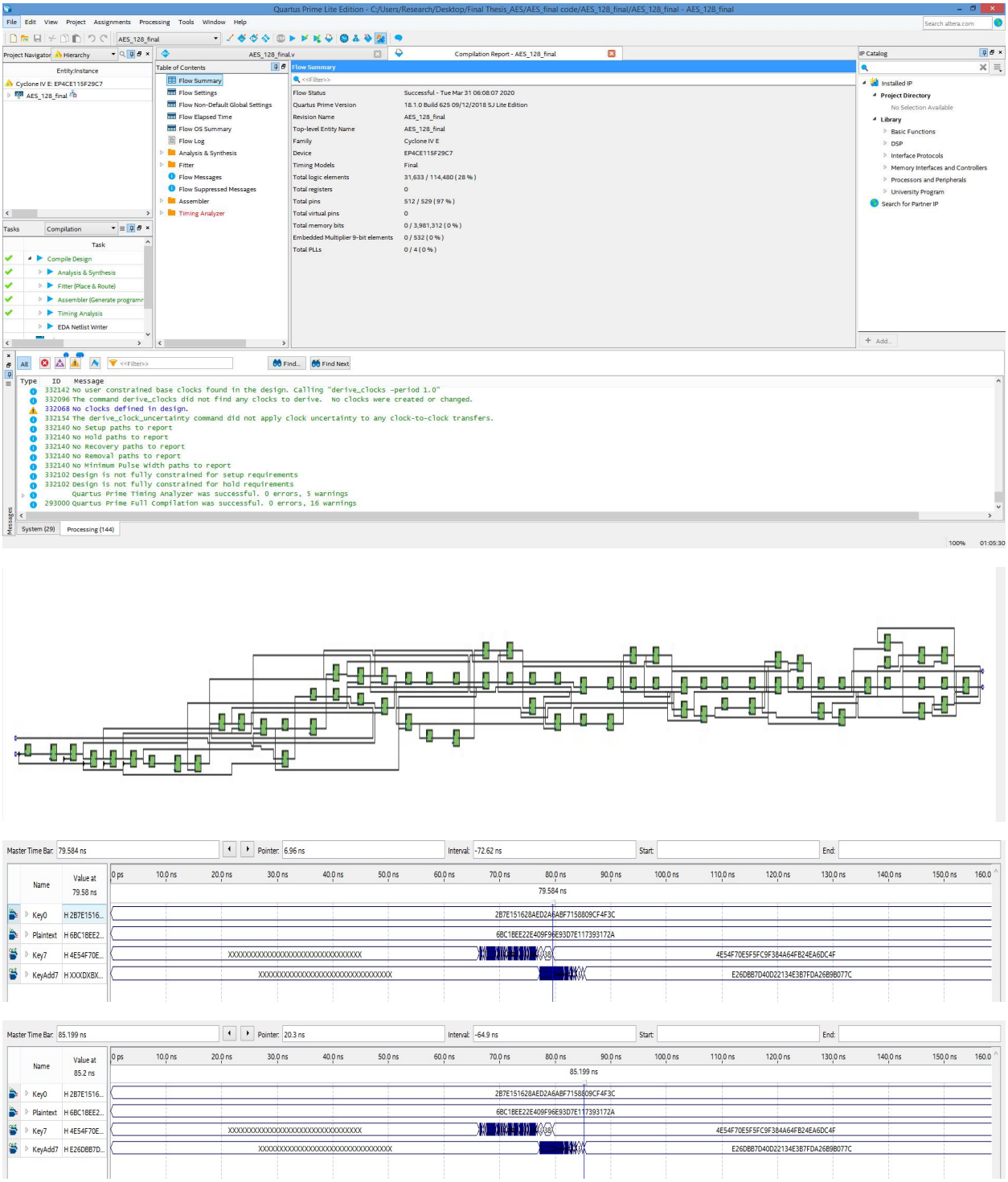

Figure 58: Timing Simulation For AES-128 after Round 7

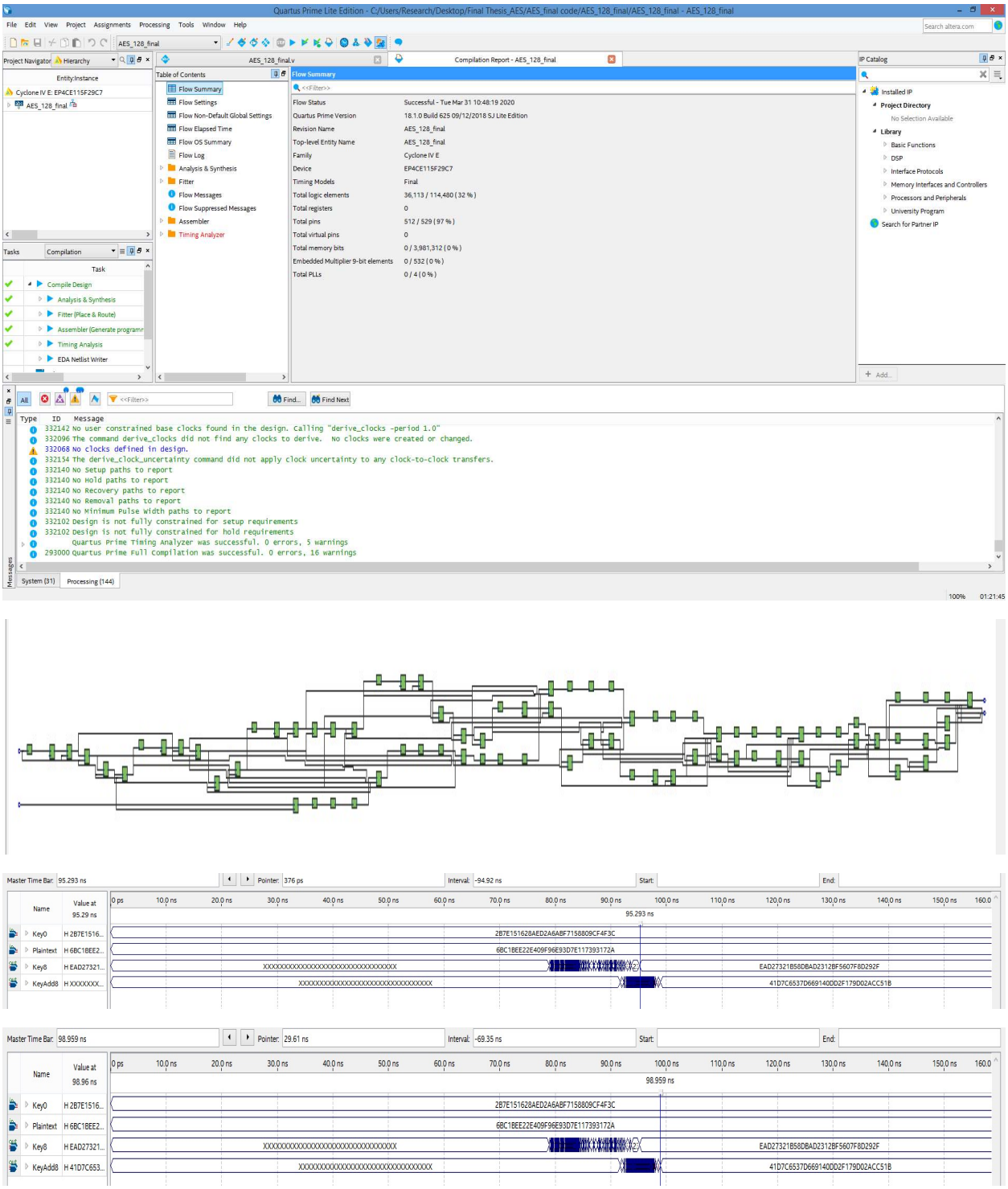

Figure 59: Timing Simulation For AES-128 after Round 8

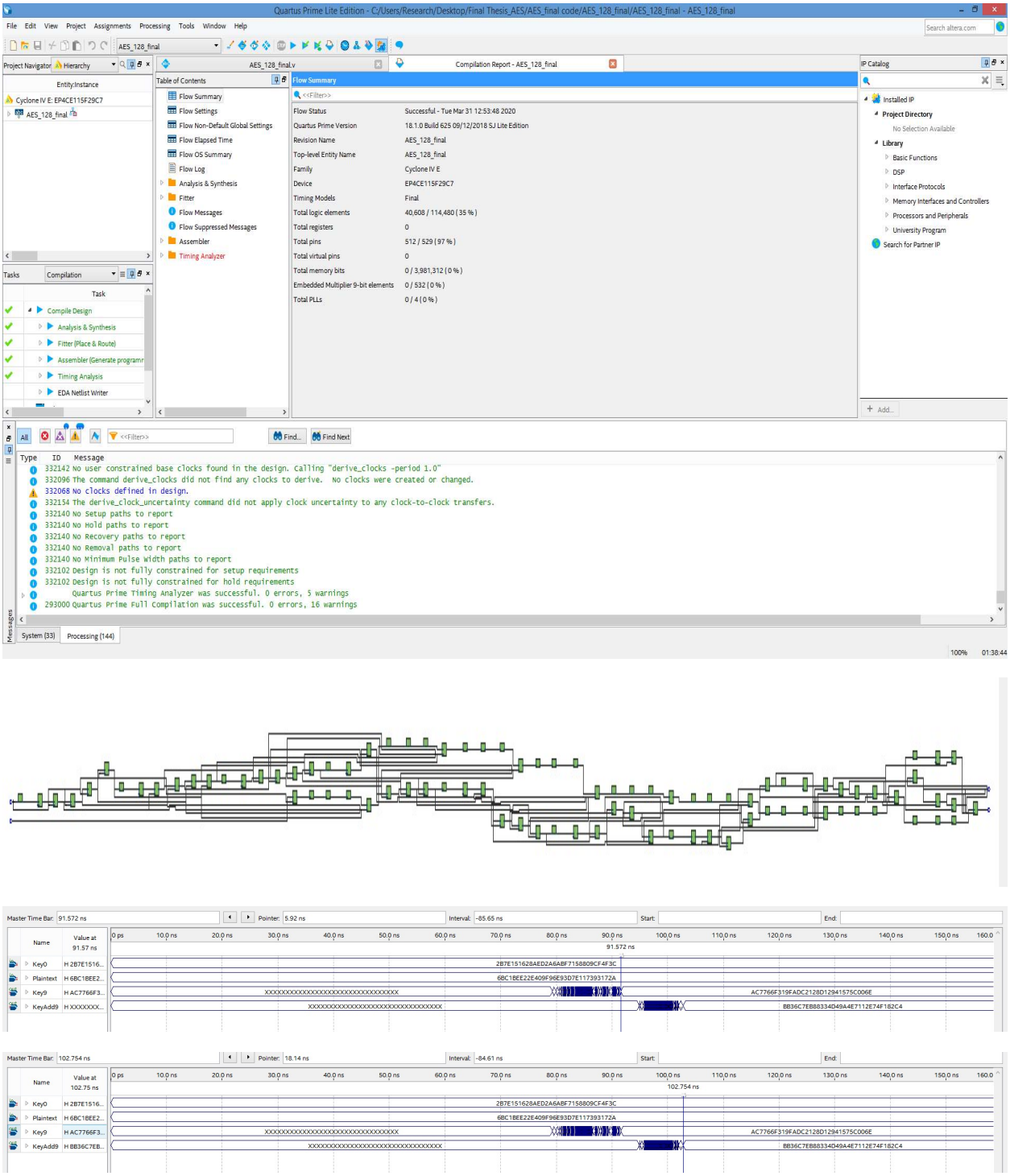

Figure 60: Timing Simulation For AES-128 after Round 9

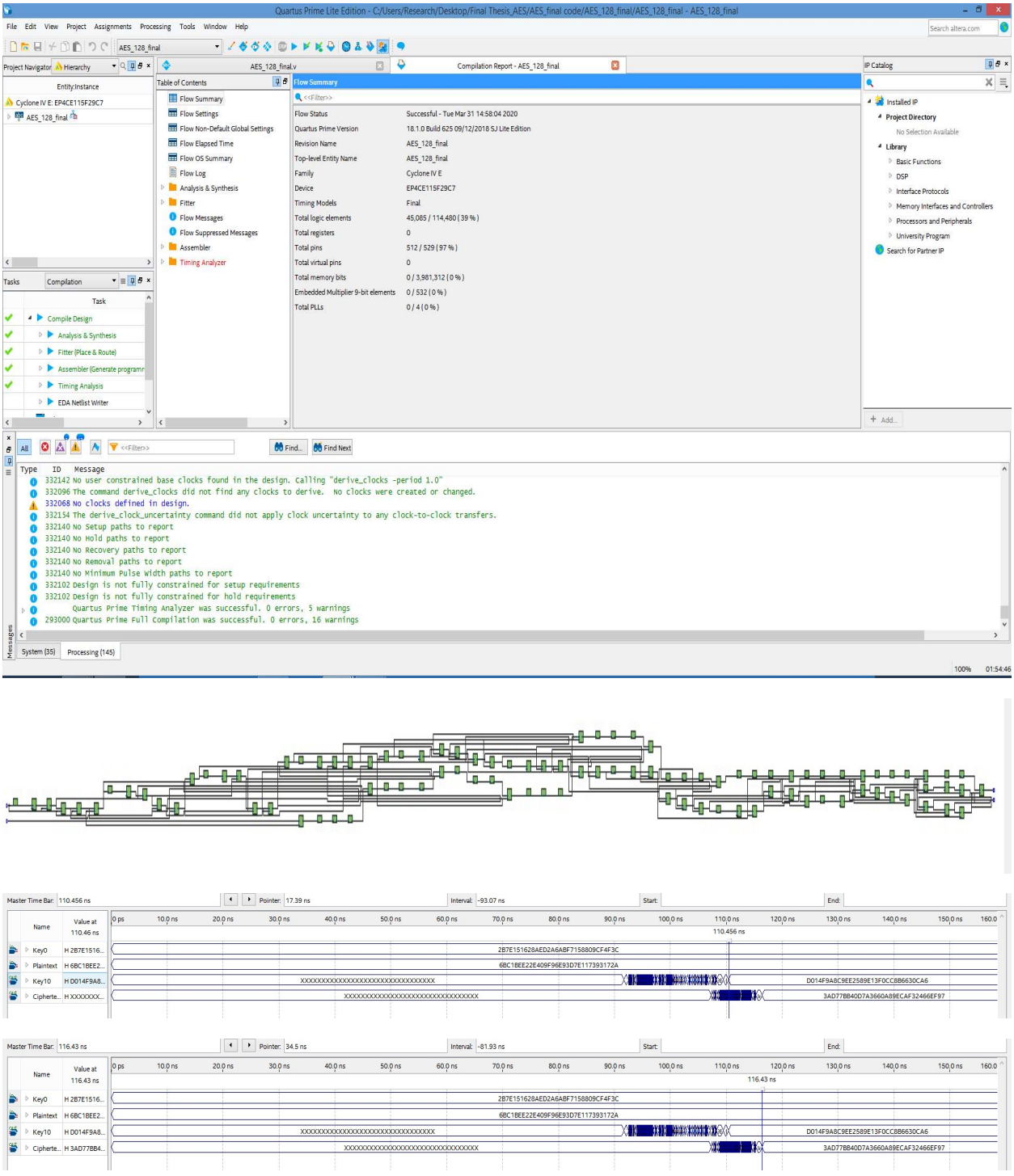

Figure 61: Timing Simulation For AES-128 after Round 10

Using the data collected from Figure 52-61, we built the Table 11 and created Figure 62 and Figure 63.

Table 11: Delay_AES_128_ECB

	Key (ns)	KeyAdd /Ciphertext (ns)	Logic elements (114480)	Processing time (H:M:S)
Round1	25.07	24.31	4632	1:41
Round2	28.84	32.05	9114	3:55
Round3	38.62	42.21	13604	12:43
Round4	48.87	54.91	18084	24:43
Round5	58.28	61.37	22602	37:45
Round6	67.91	73.24	27095	50:33
Round7	79.58	85.2	31633	1:05:30
Round8	95.29	98.96	36113	1:21:45
Round9	91.57	102.75	40608	1:38:44
Round10	110.46	116.43	45085	1:54:46

The results in Figure 62 show a linear trend where an increase in round number results in an increase in delay time for key expansion and the plaintext encryption portion. However, the processing time is not a linear trend as shown in Figure 63, because the processing time is based on the operating system.

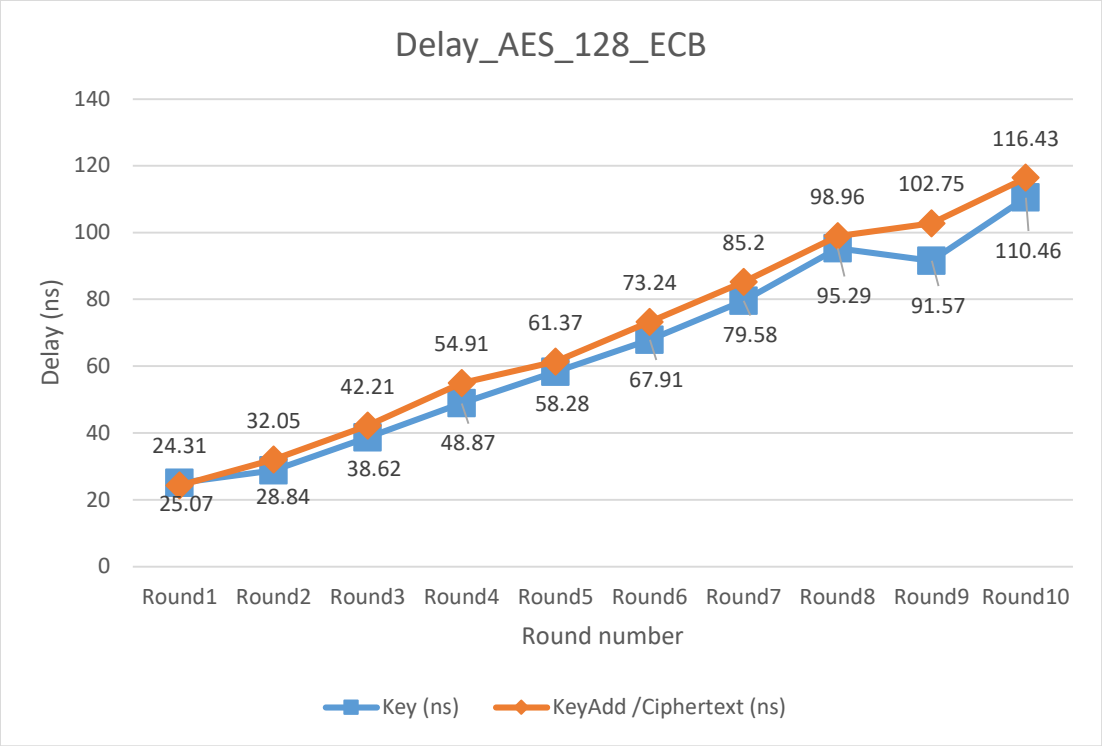

Figure 62: Delay_AES_128_ECB

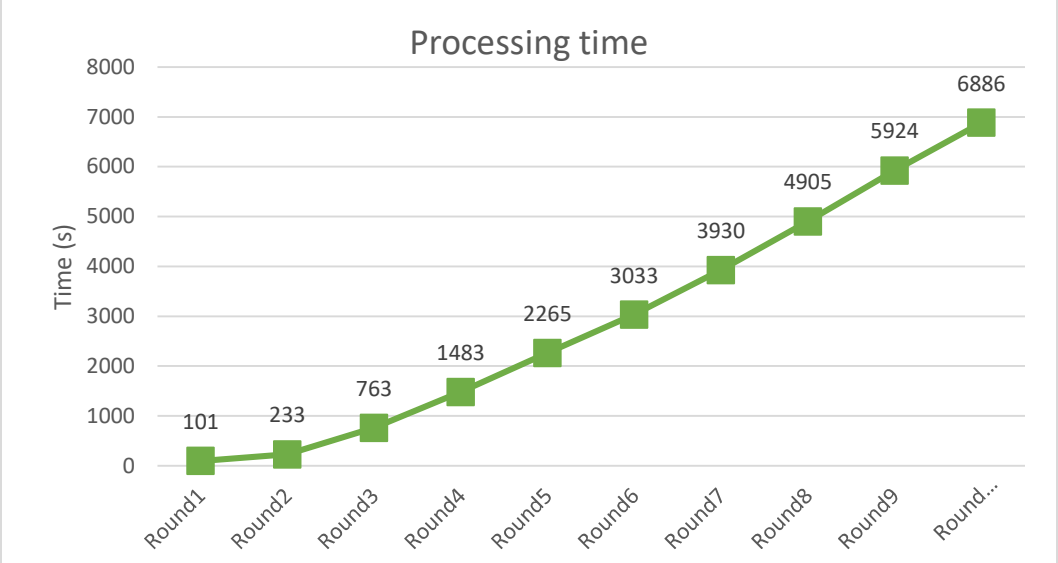

Figure 63: Processing time_ AES_128_ECB

5.2.2 Timing Simulation For AES-192

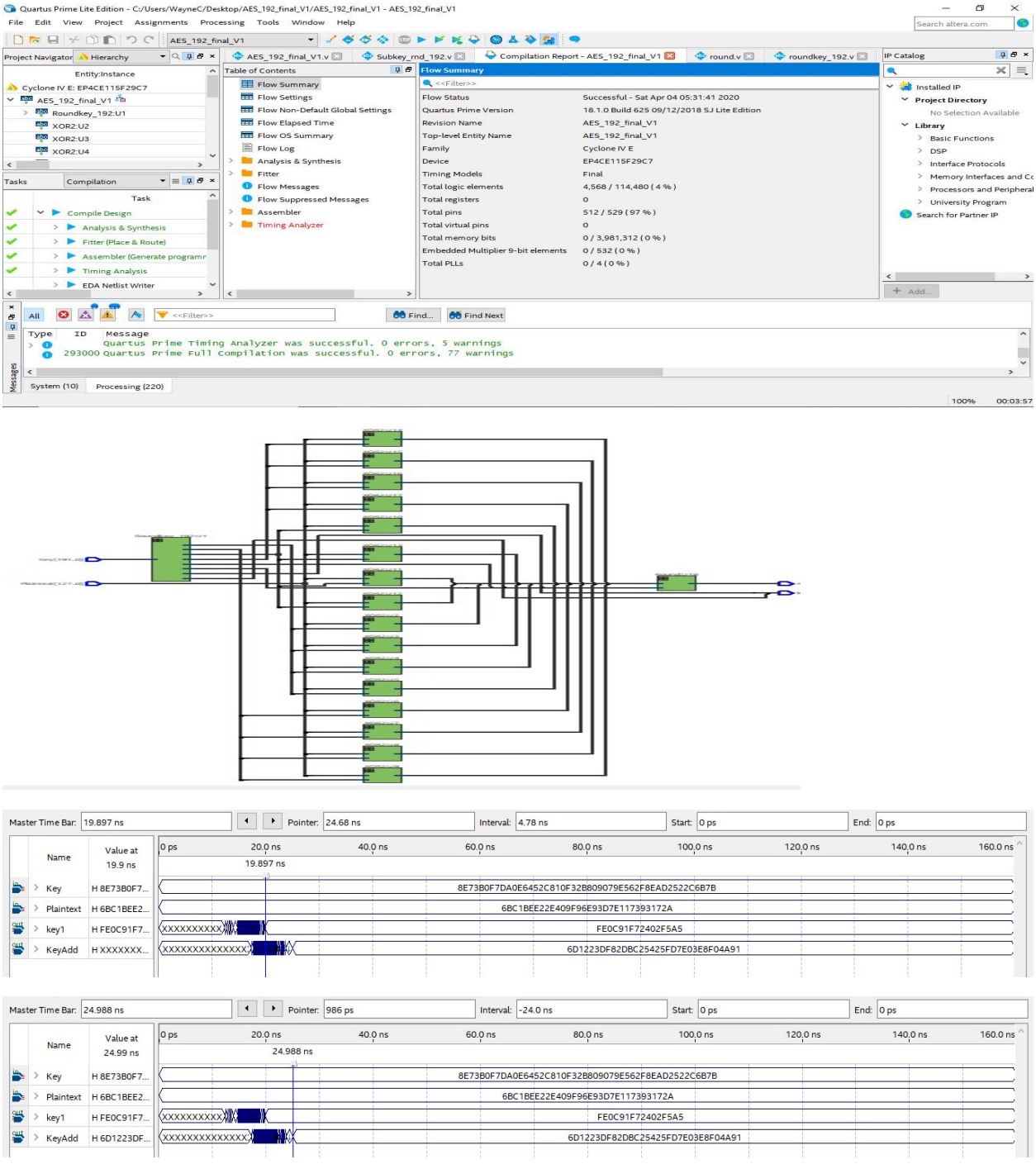

Figure 64: Timing Simulation For AES-192 after Round 1

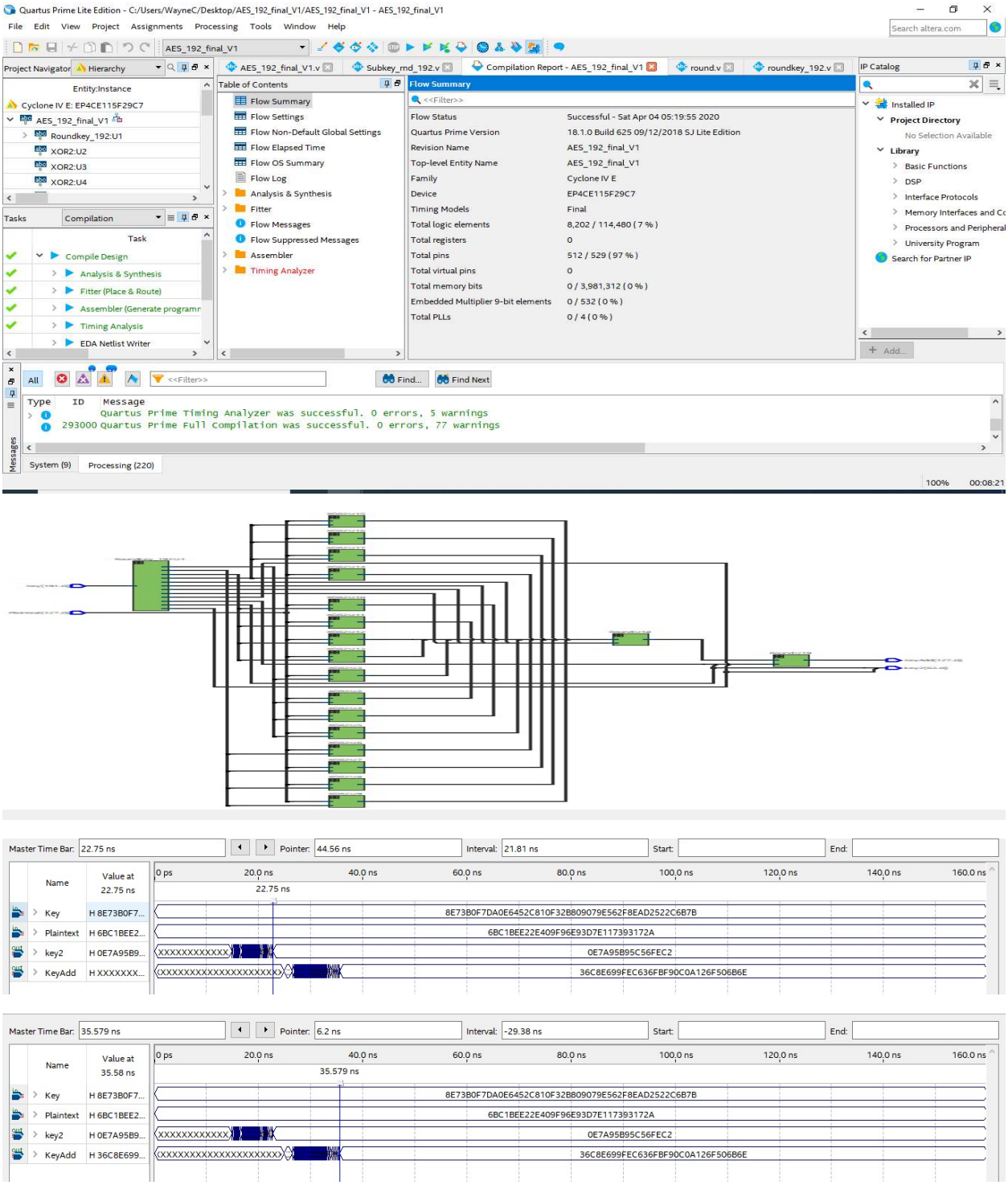

Figure 65: Timing Simulation For AES-192 after Round 2

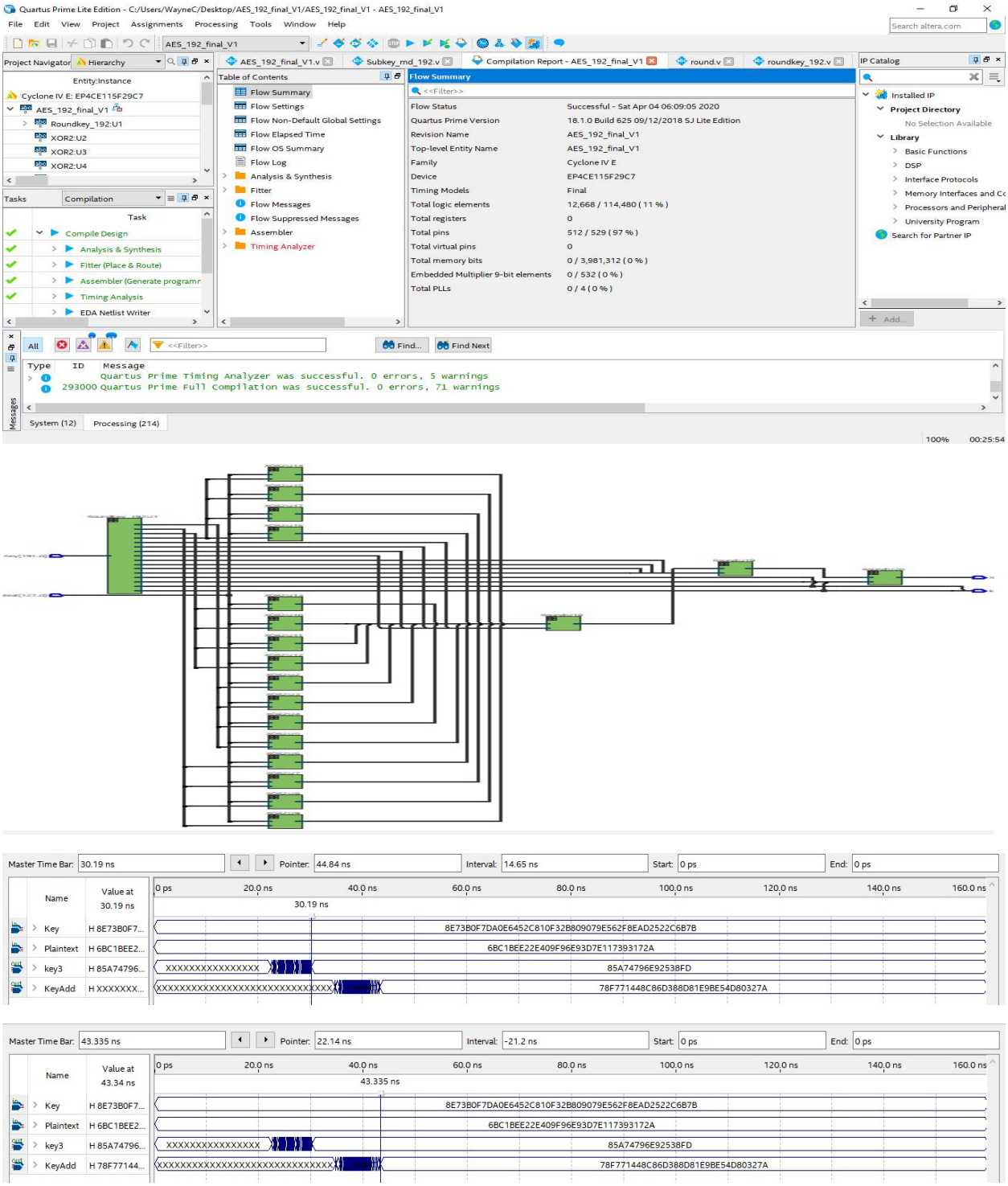

Figure 66: Timing Simulation For AES-192 after Round 3

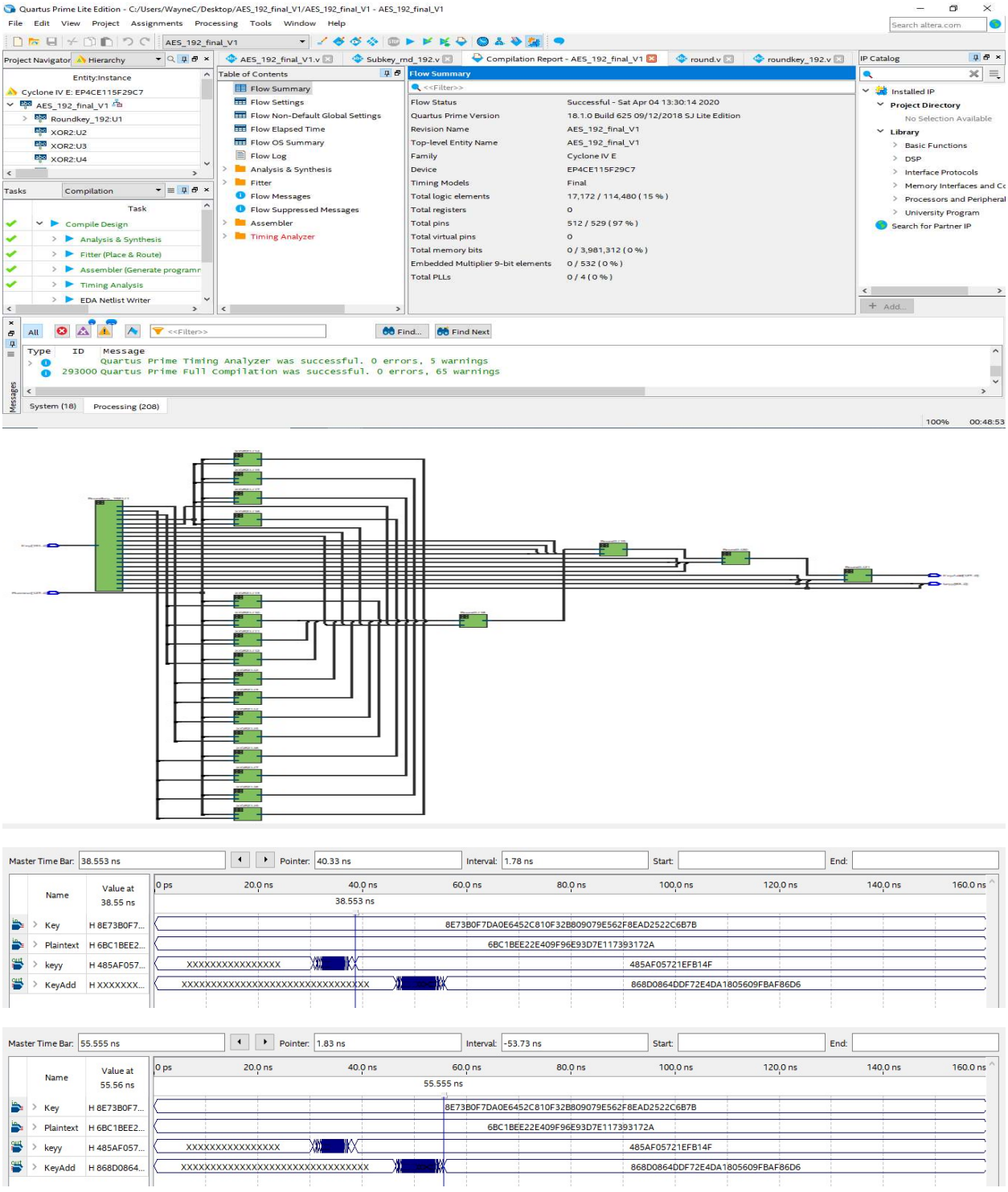

Figure 67: Timing Simulation For AES-192 after Round 4

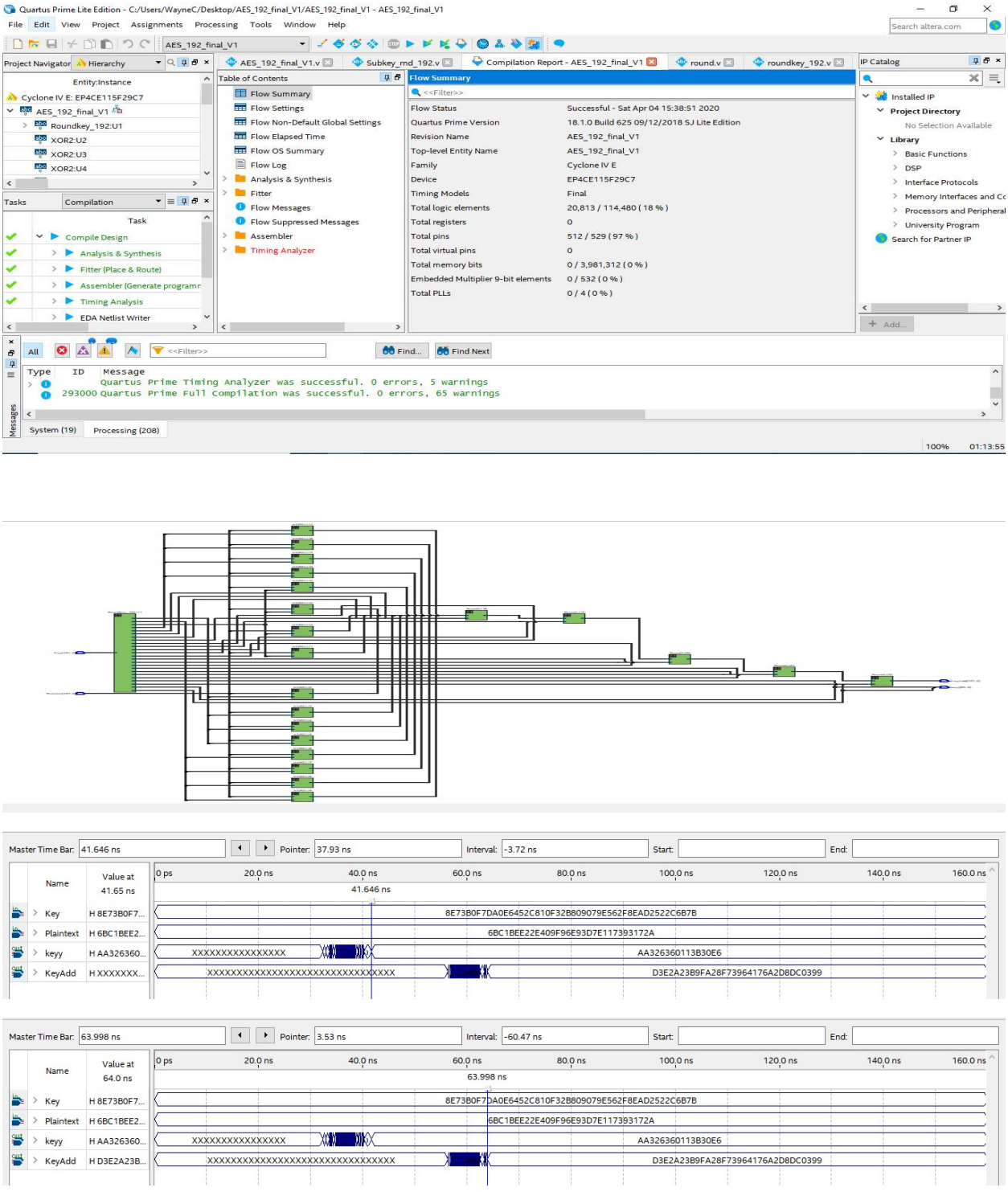

Figure 68: Timing Simulation For AES-192 after Round 5

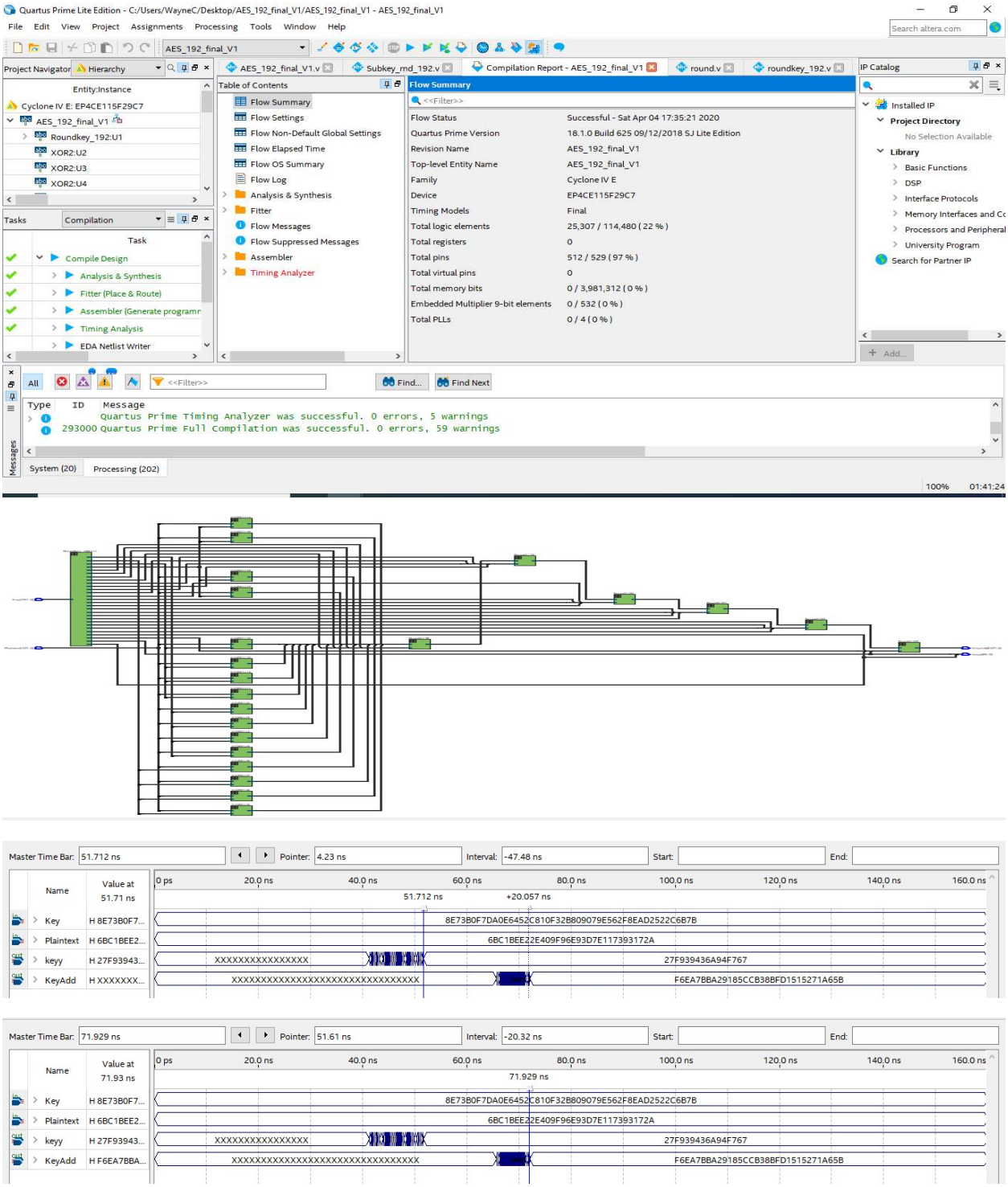

Figure 69: Timing Simulation For AES-192 after Round 6

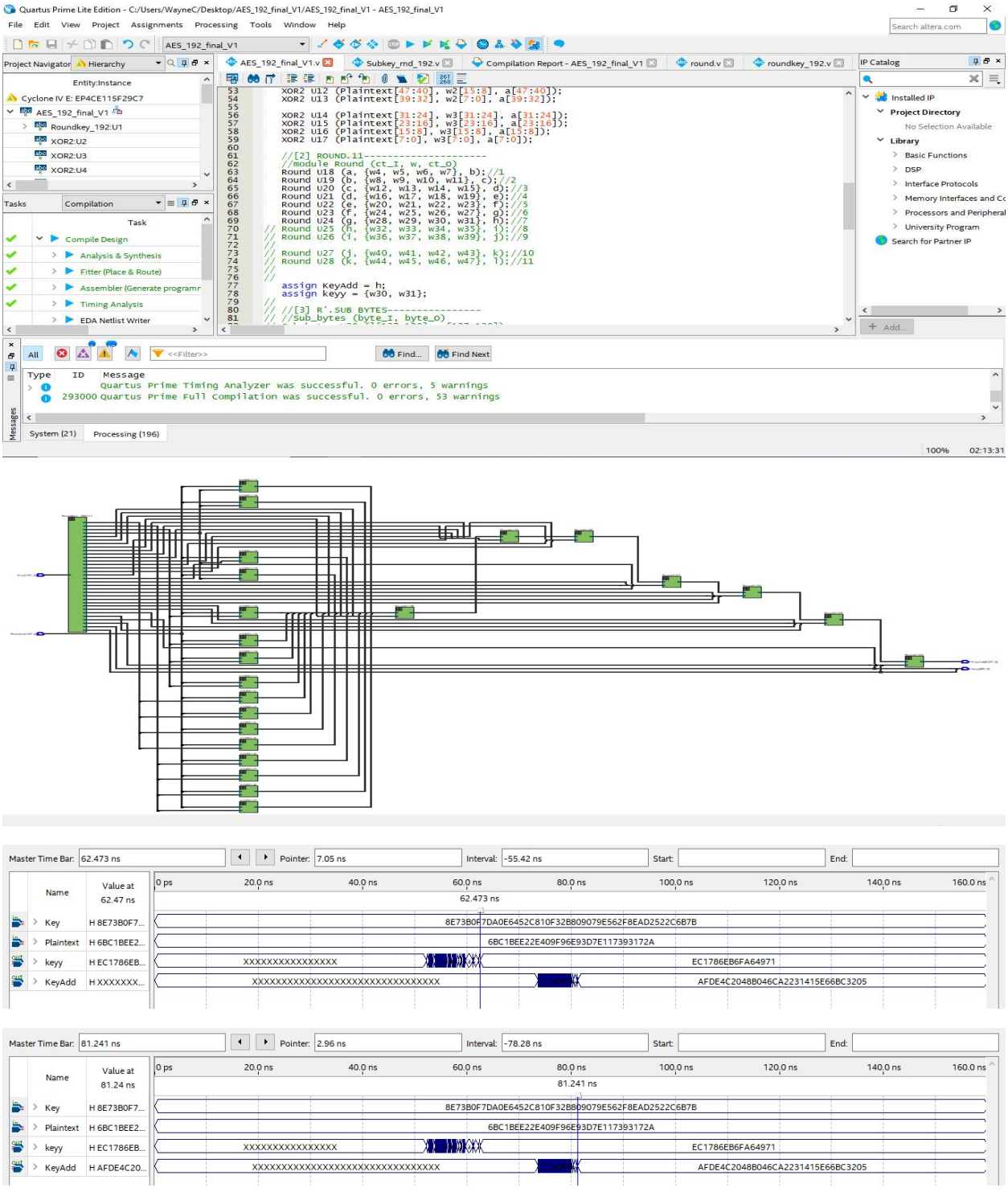

Figure 70: Timing Simulation For AES-192 after Round 7

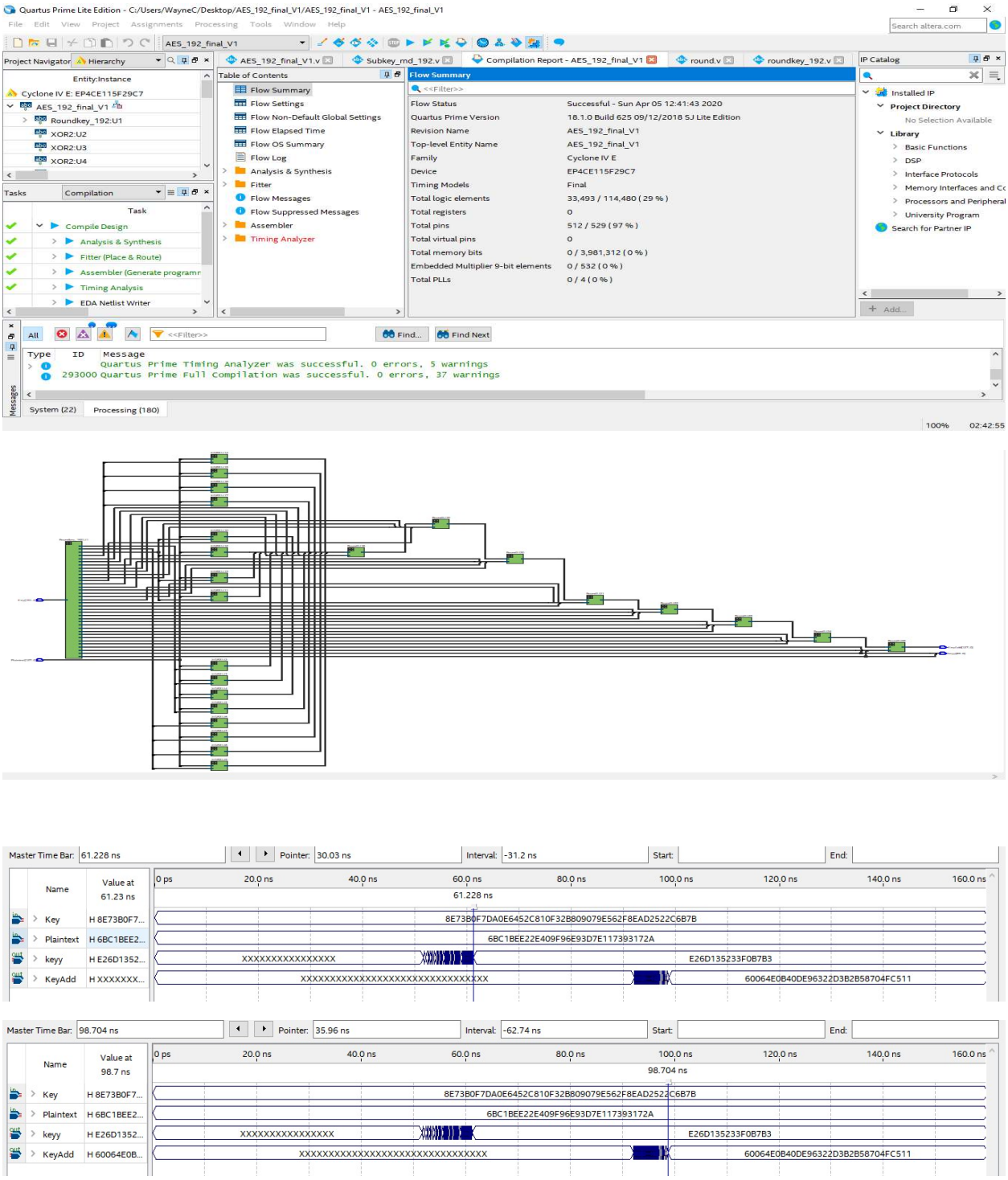

Figure 71: Timing Simulation For AES-192 after Round 8

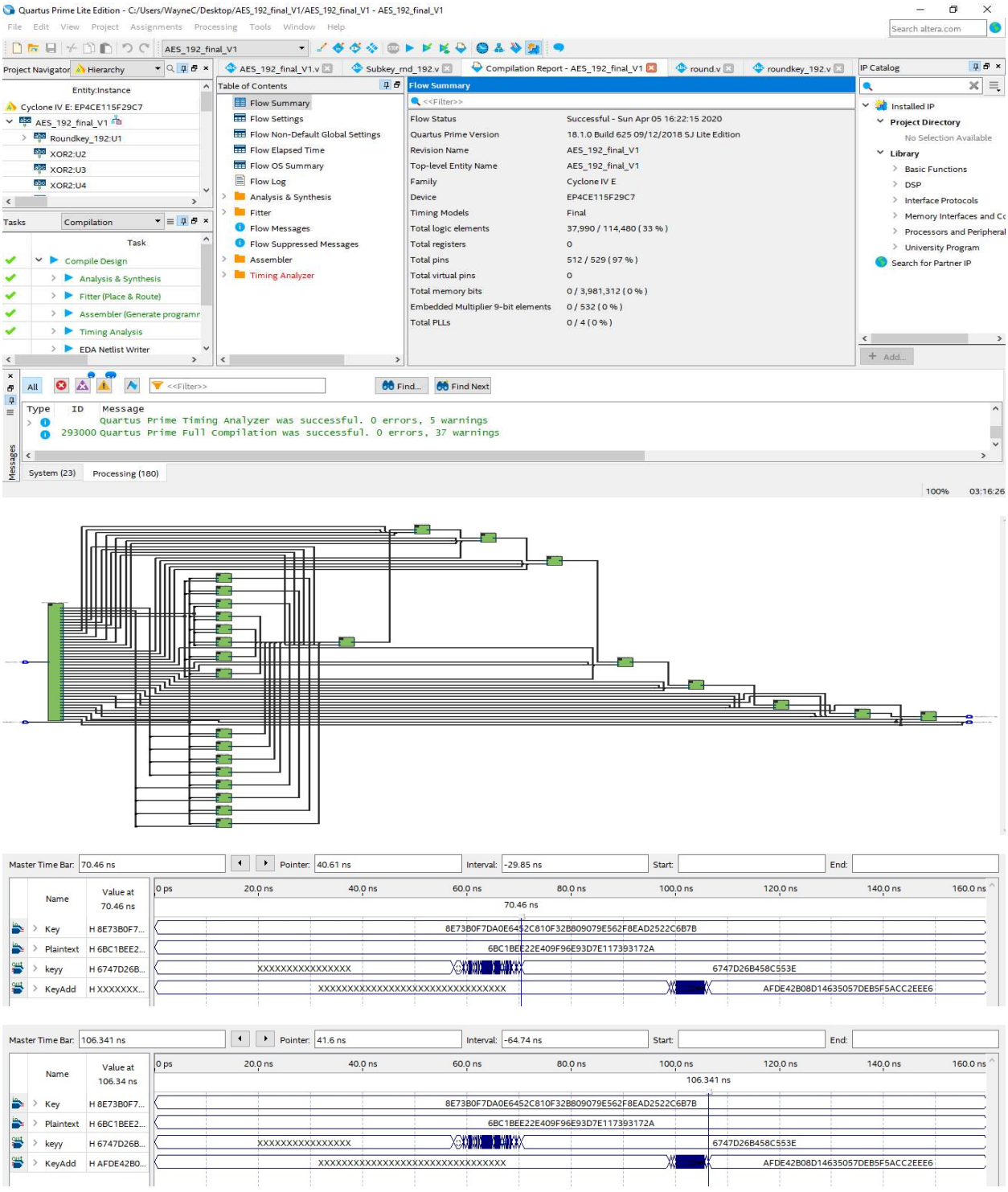

Figure 72: Timing Simulation For AES-192 after Round 9

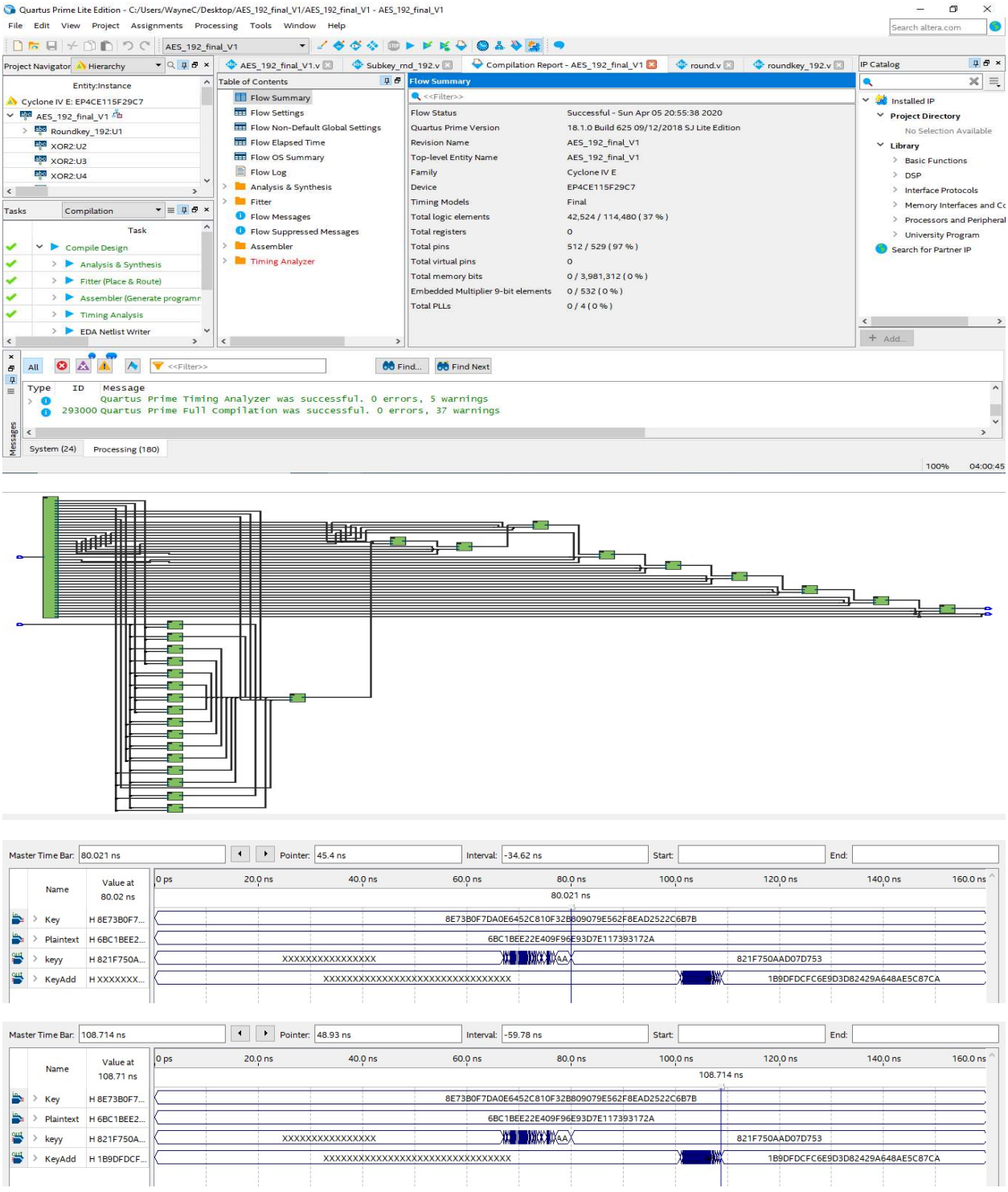

Figure 73: Timing Simulation For AES-192 after Round 10

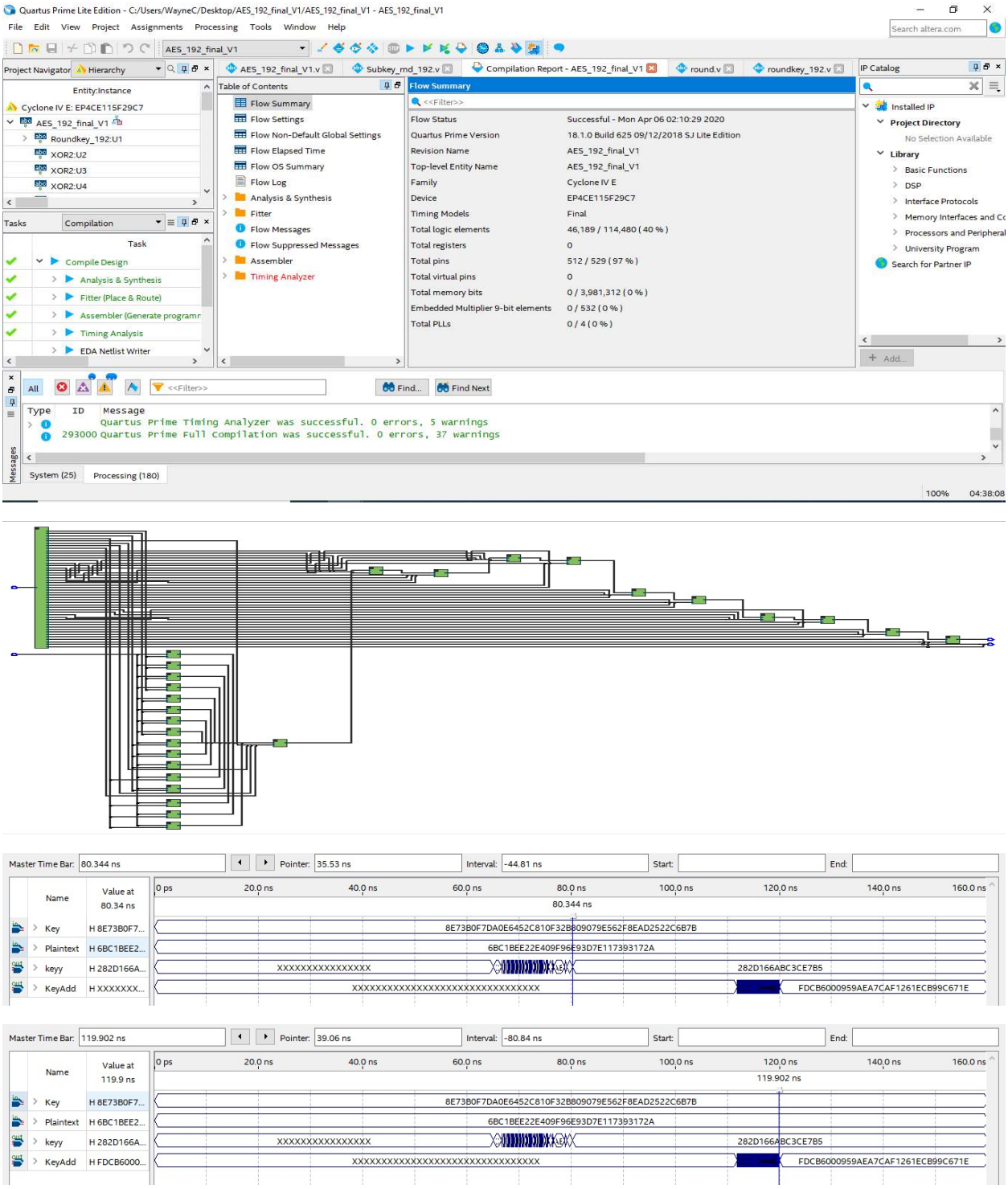

Figure 74: Timing Simulation For AES-192 after Round 11

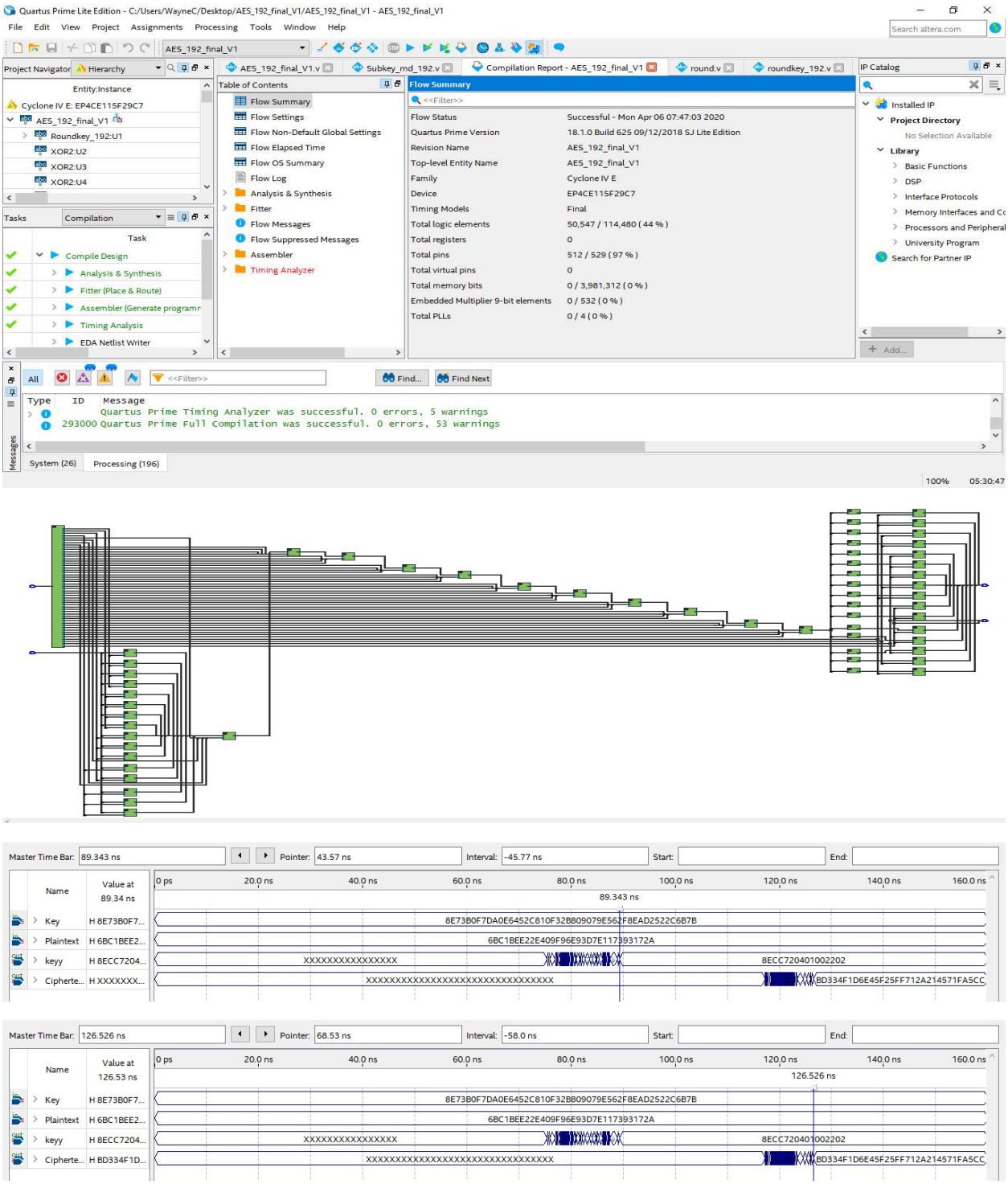

Figure 75: Timing Simulation For AES-192 after Round 12

Using the data collected from Figure 64-75, we built the Table 12 and created Figure 76 and Figure 77.

Table 12: Delay_AES_192_ECB

	Key	KeyAdd (Ciphertext)	Logic elements (114480)	Processing time
Round1	19.90	24.99	4568	3:57
Round2	22.75	35.58	8202	8:02
Round3	30.19	43.34	12668	25:54
Round4	38.55	55.56	17172	48:53
Round5	41.65	64.00	20813	1:13:55
Round6	51.71	71.93	25307	1:41:24
Round7	62.47	81.24	29400	2:13:31
Round8	61.23	98.70	33493	2:42:55
Round9	70.46	106.34	37990	3:16:26
Round10	80.02	108.71	42524	4:00:45
Round11	80.34	119.90	46189	4:38:08
Round12	89.34	126.53	50.547	5:30:47

The results in Figure 76 show a linear trend where an increase in round number results in an increase in delay time for key expansion and the plaintext encryption portion. However, the

processing time is not a linear trend as shown in Figure 77, because the processing time is based on the operation system.

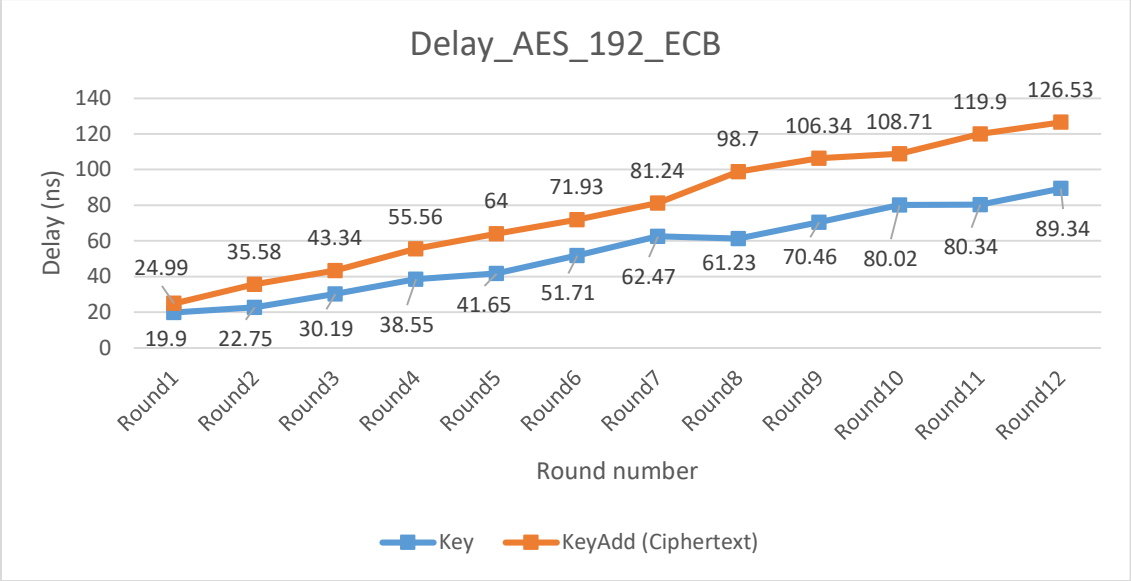

Figure 76: Delay_AES_192_ECB

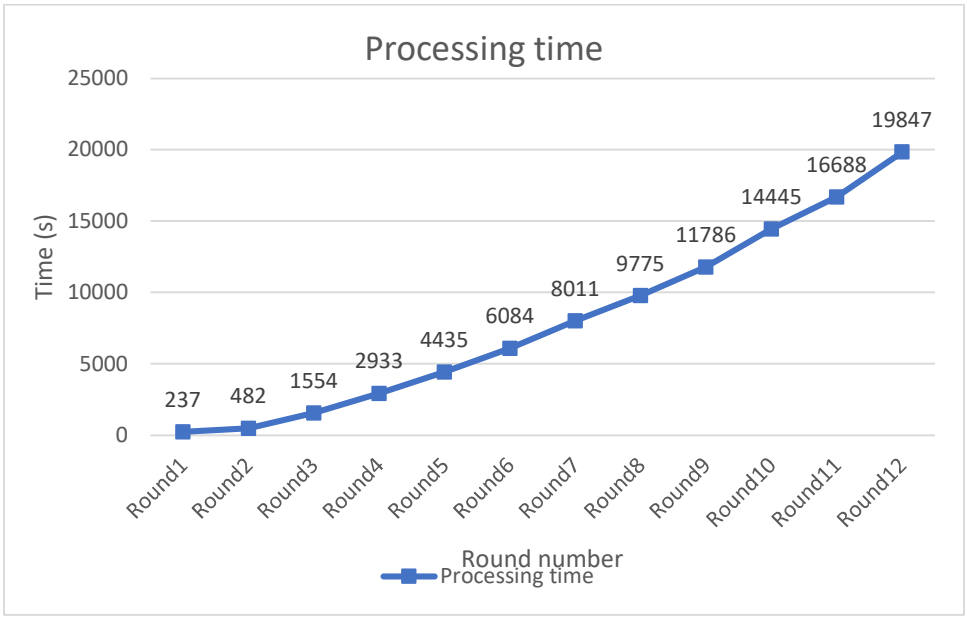

Figure 77: Processing time_ AES_192_ECB

5.2.3 Timing Simulation For AES-256

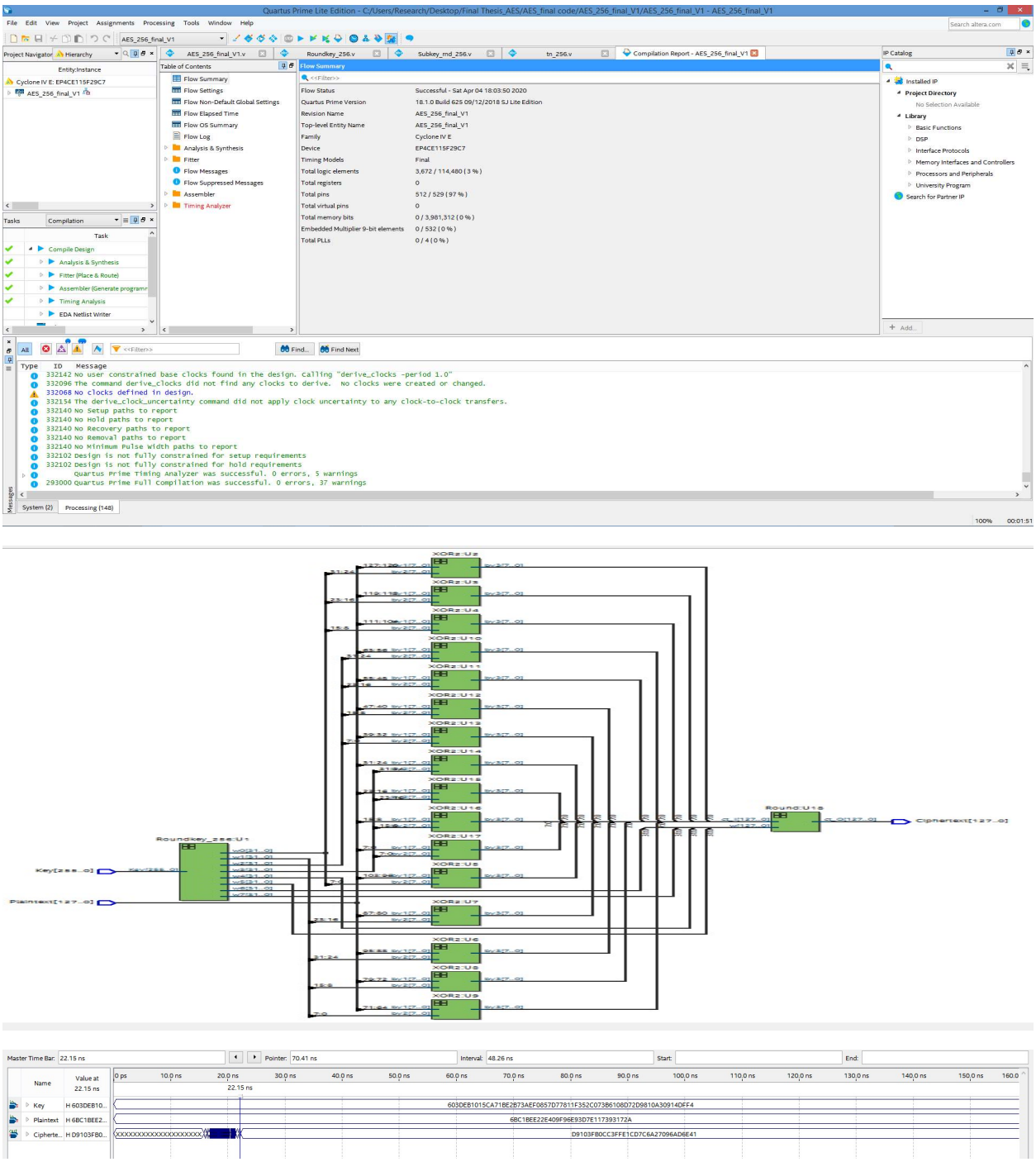

Figure 78: Timing Simulation For AES-256 after Round 1

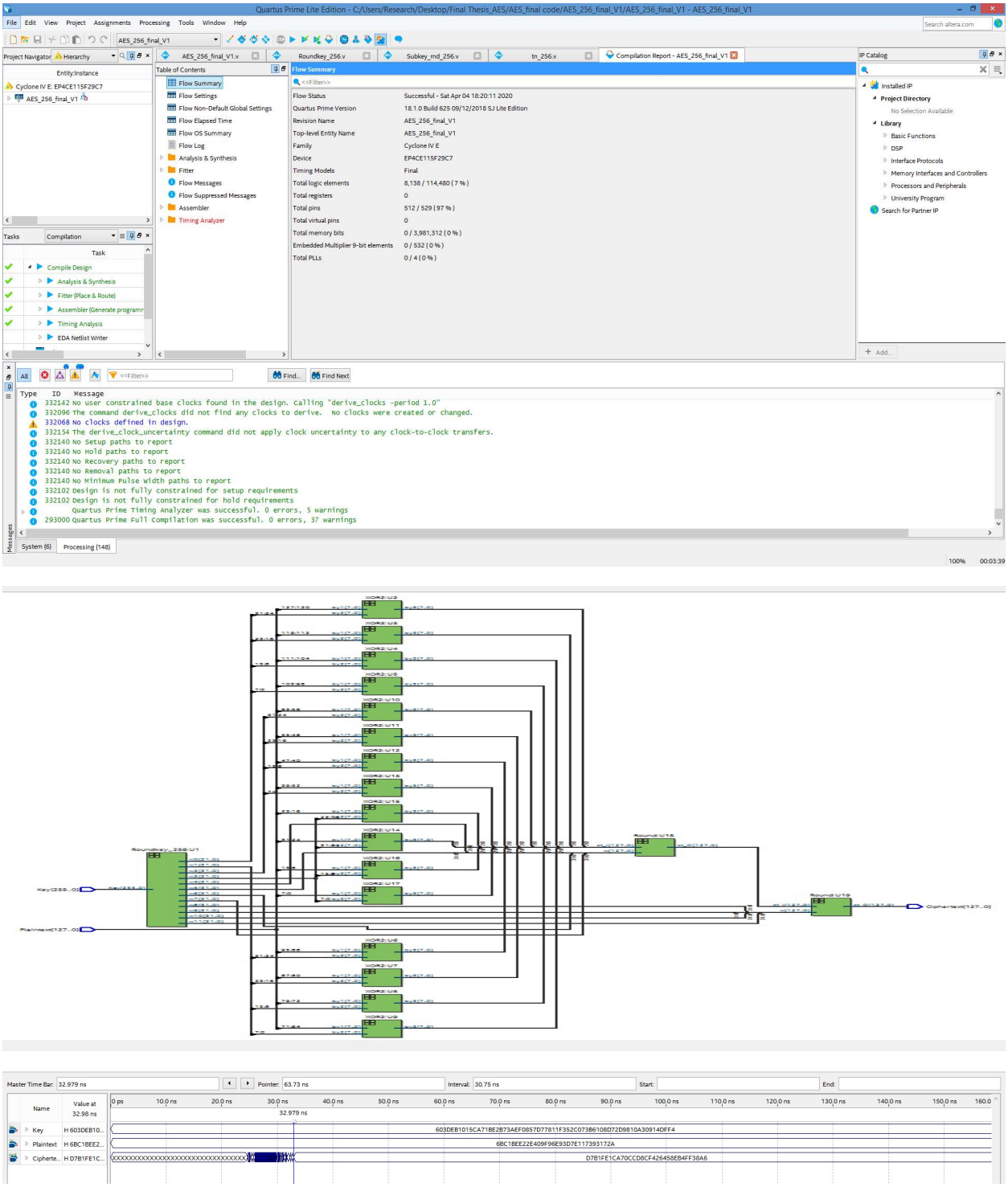

Figure 79: Timing Simulation For AES-256 after Round 2

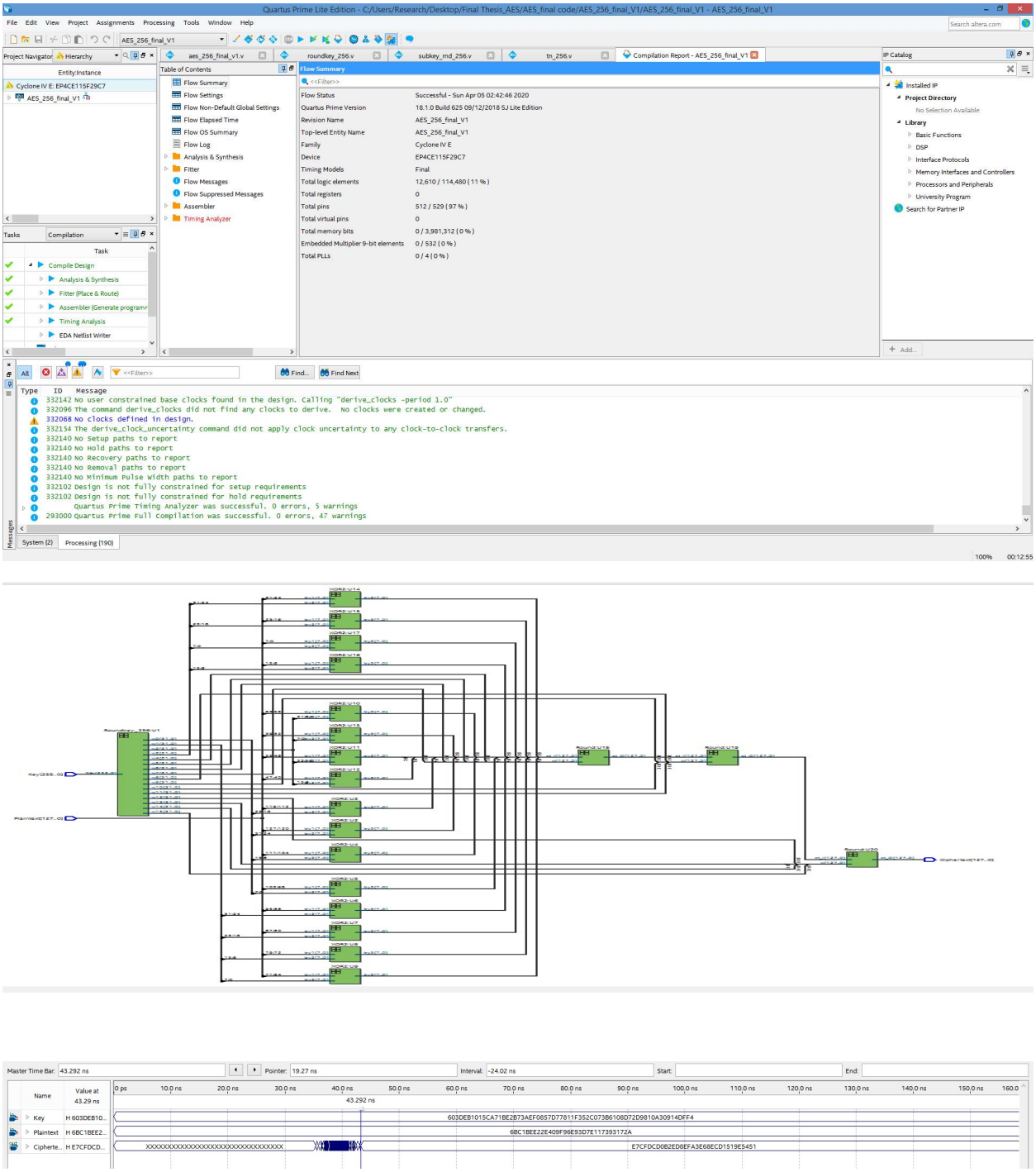

Figure 80: Timing Simulation For AES-256 after Round 3

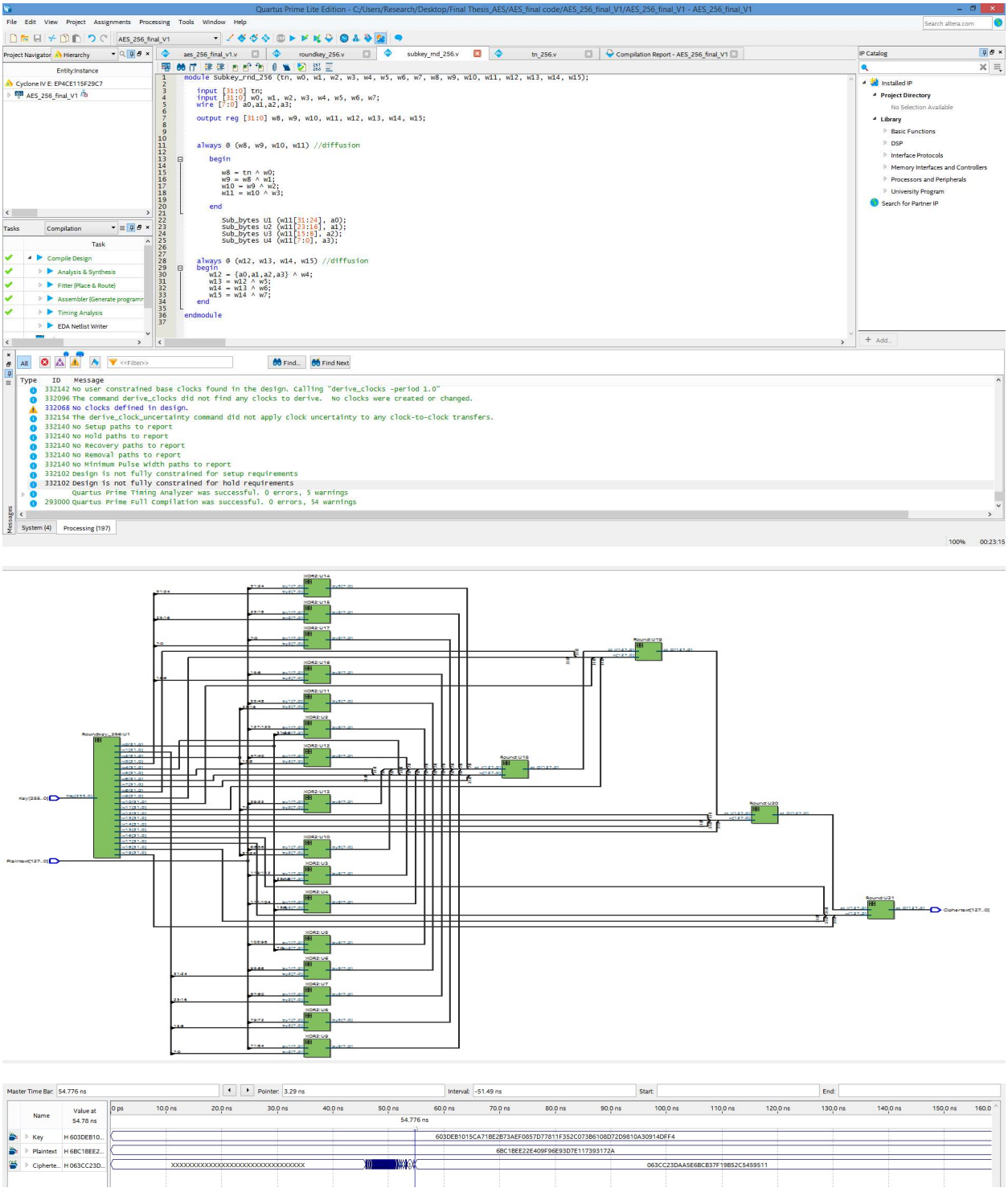

Figure 81: Timing Simulation For AES-256 after Round 4

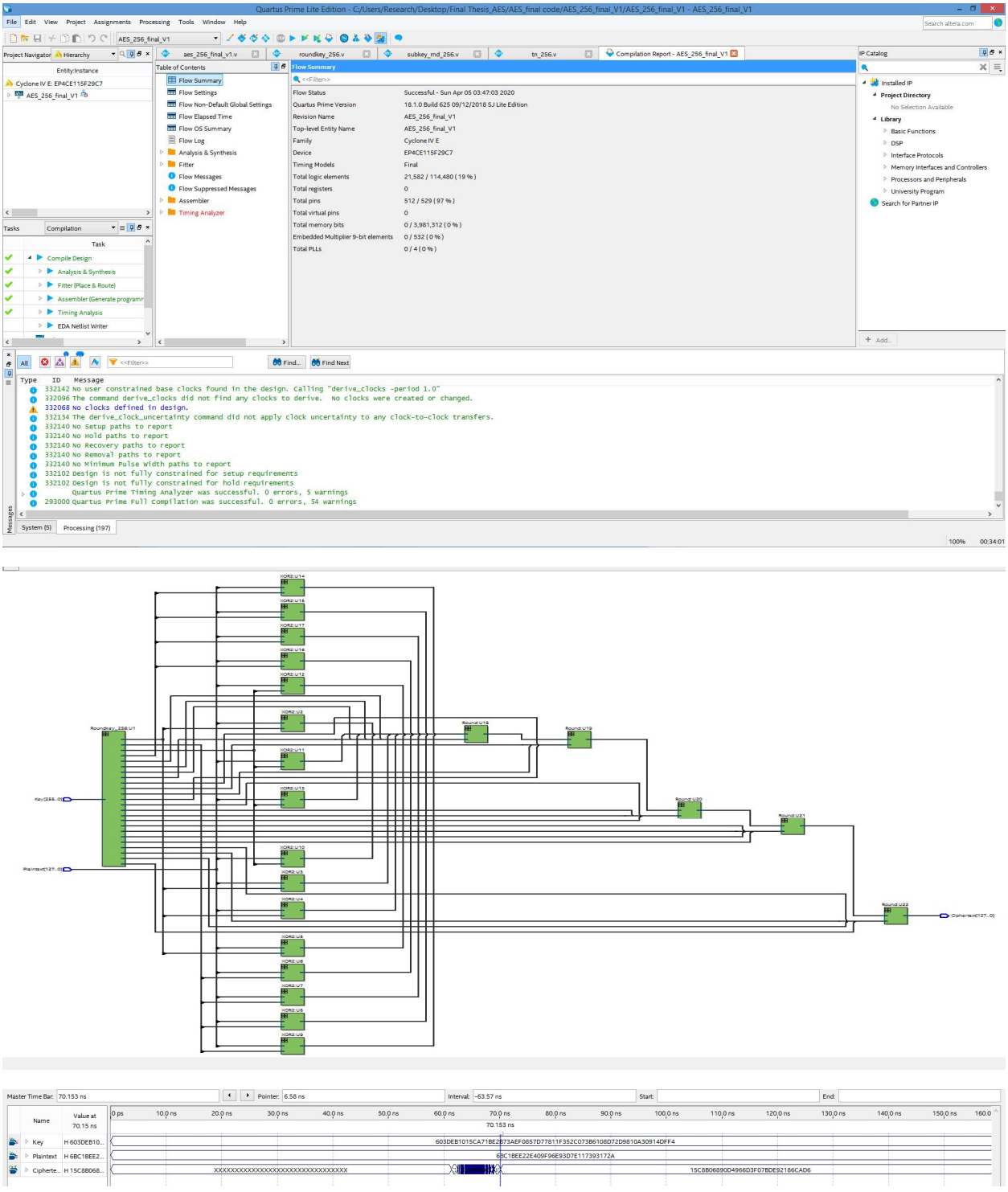

Figure 82: Timing Simulation For AES-256 after Round 5

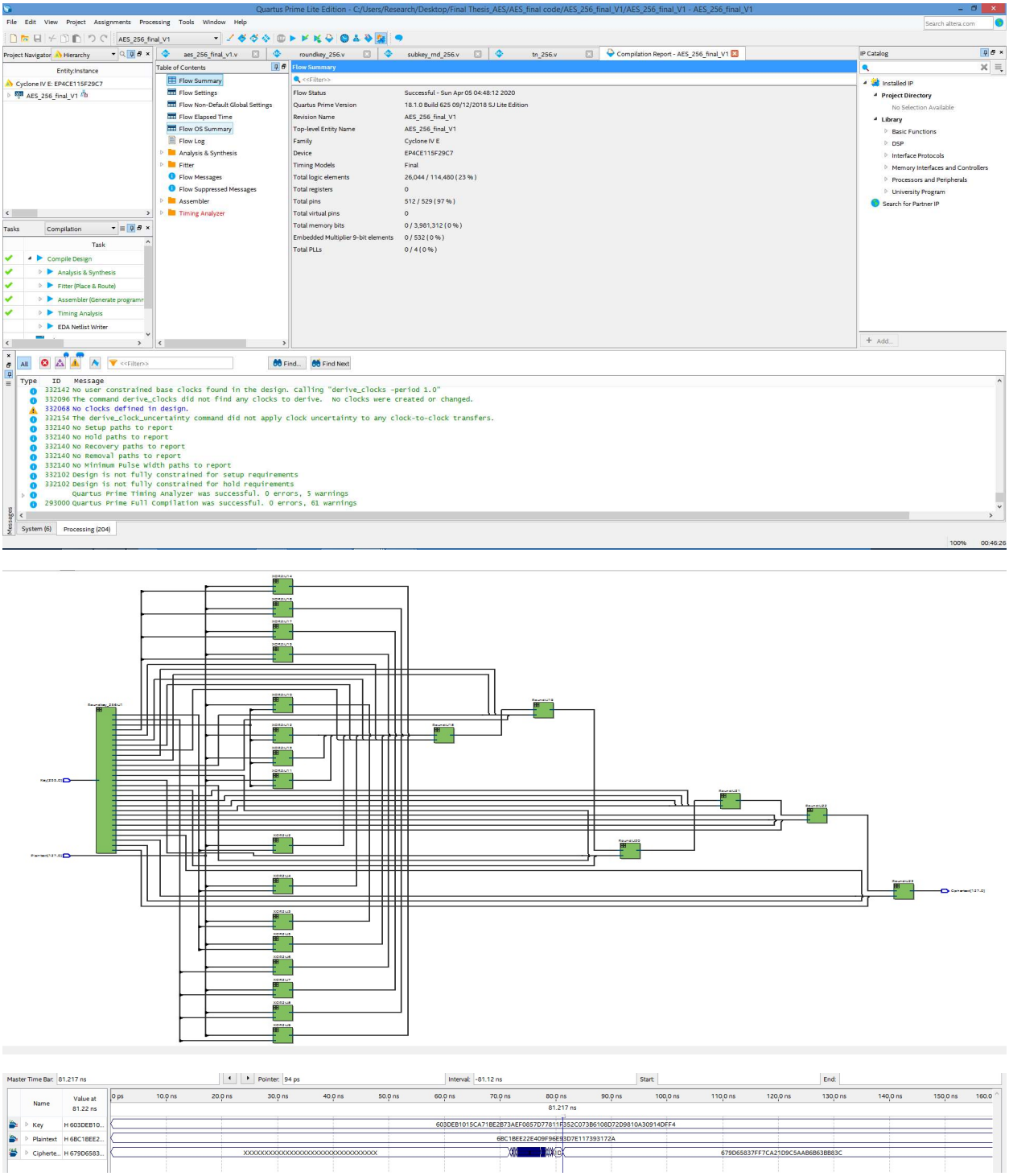

Figure 83: Timing Simulation For AES-256 after Round 6

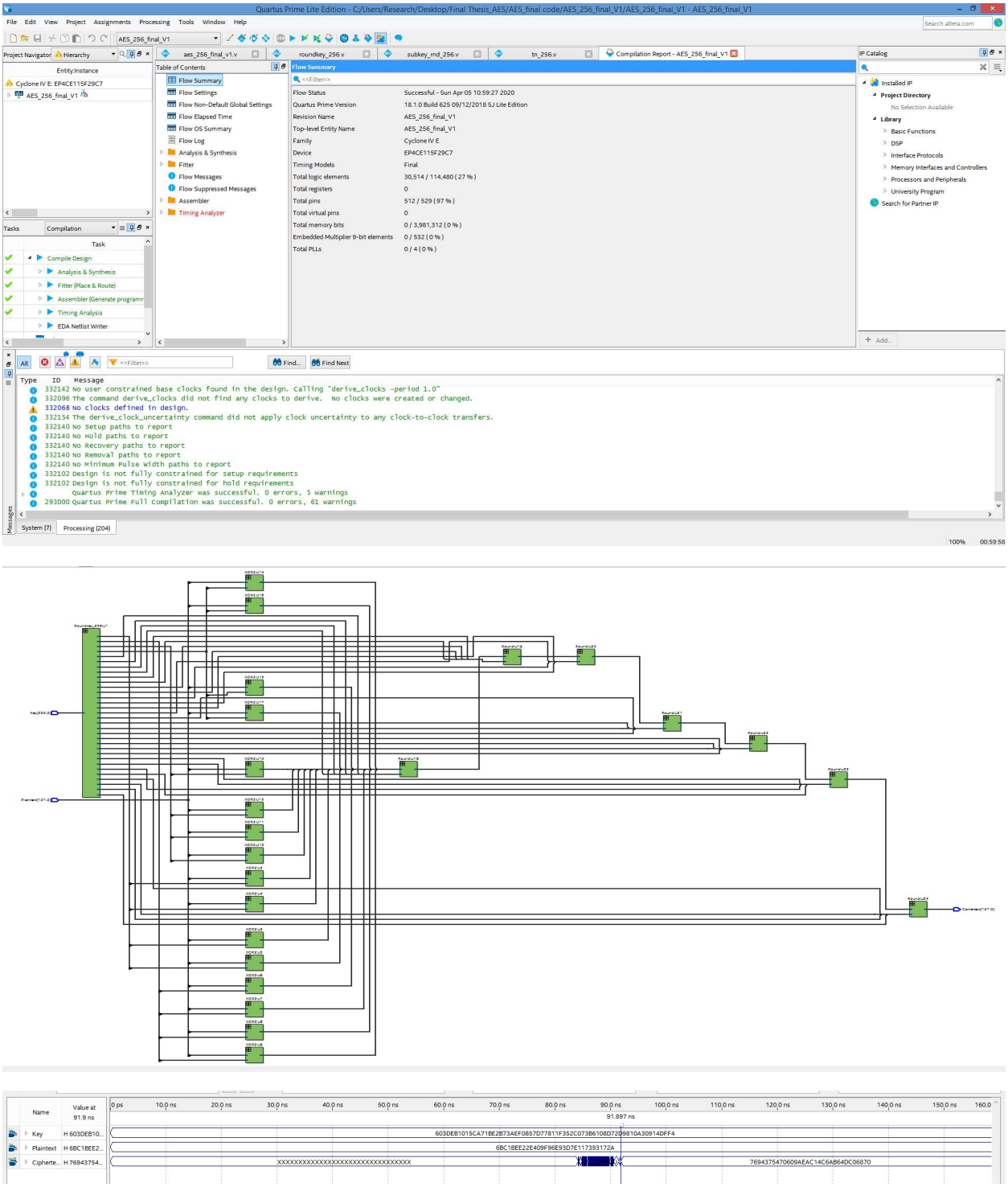

Figure 84: Timing Simulation For AES-256 after Round 7

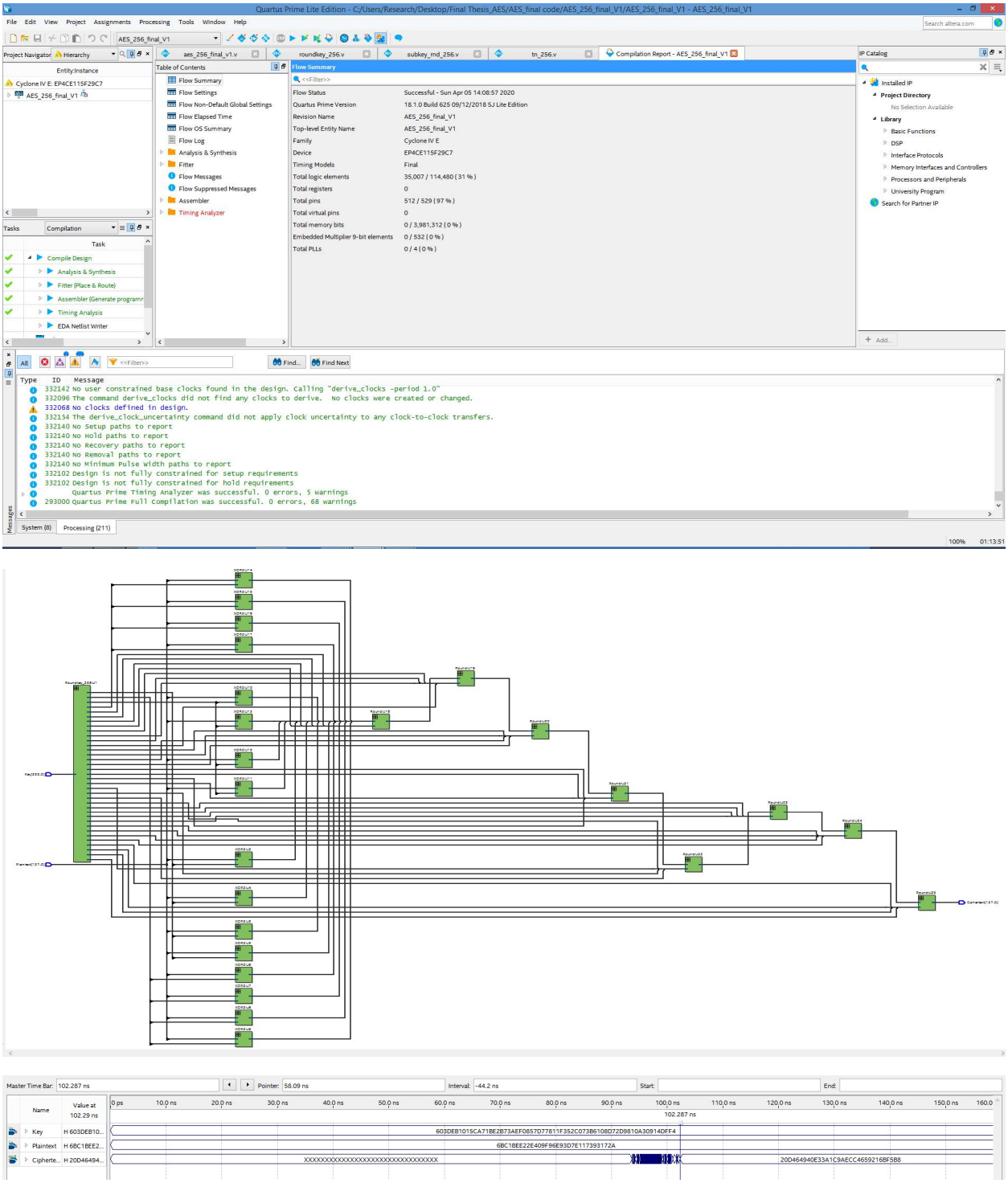

Figure 85: Timing Simulation For AES-256 after Round 8

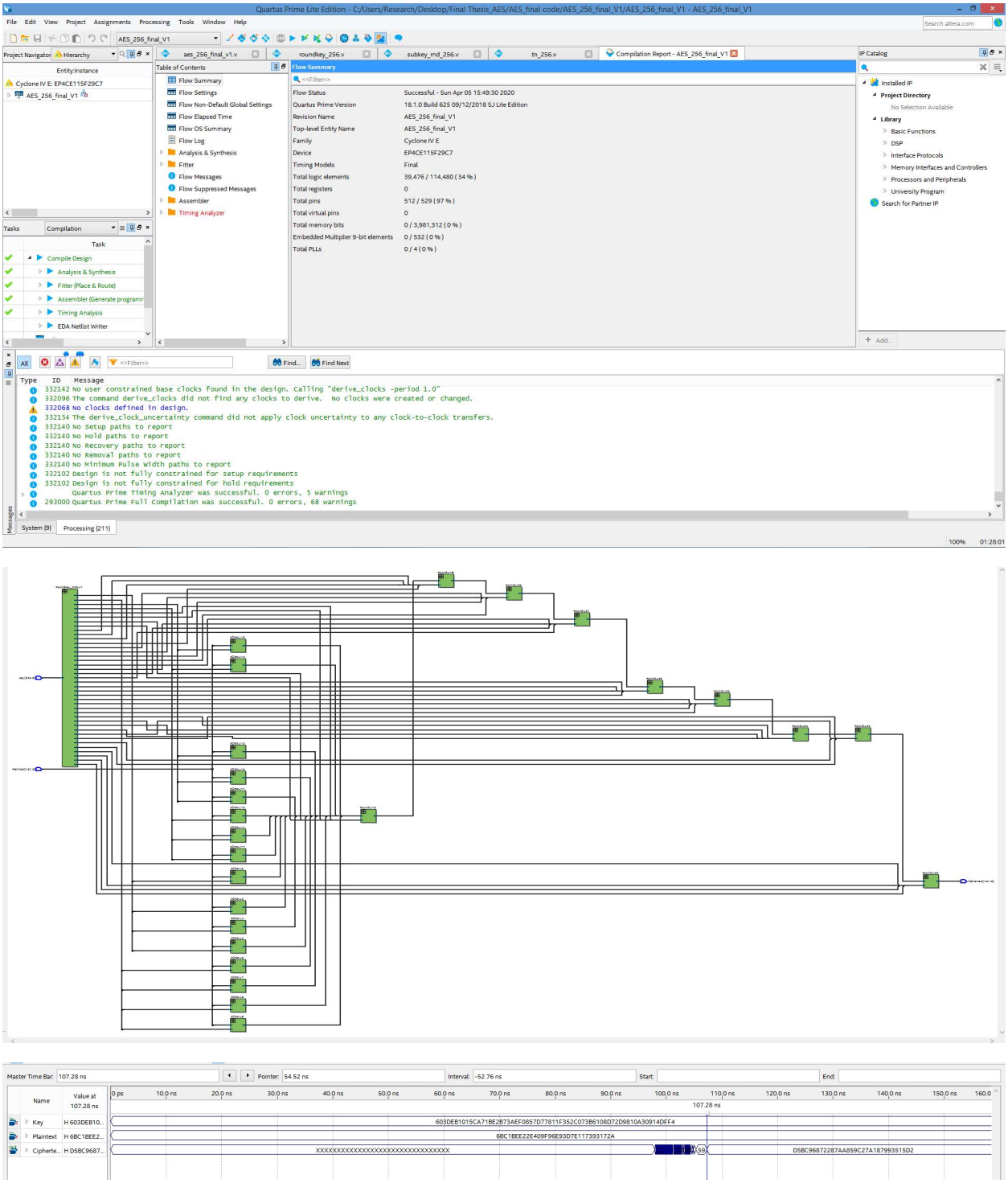

Figure 86: Timing Simulation For AES-256 after Round 9

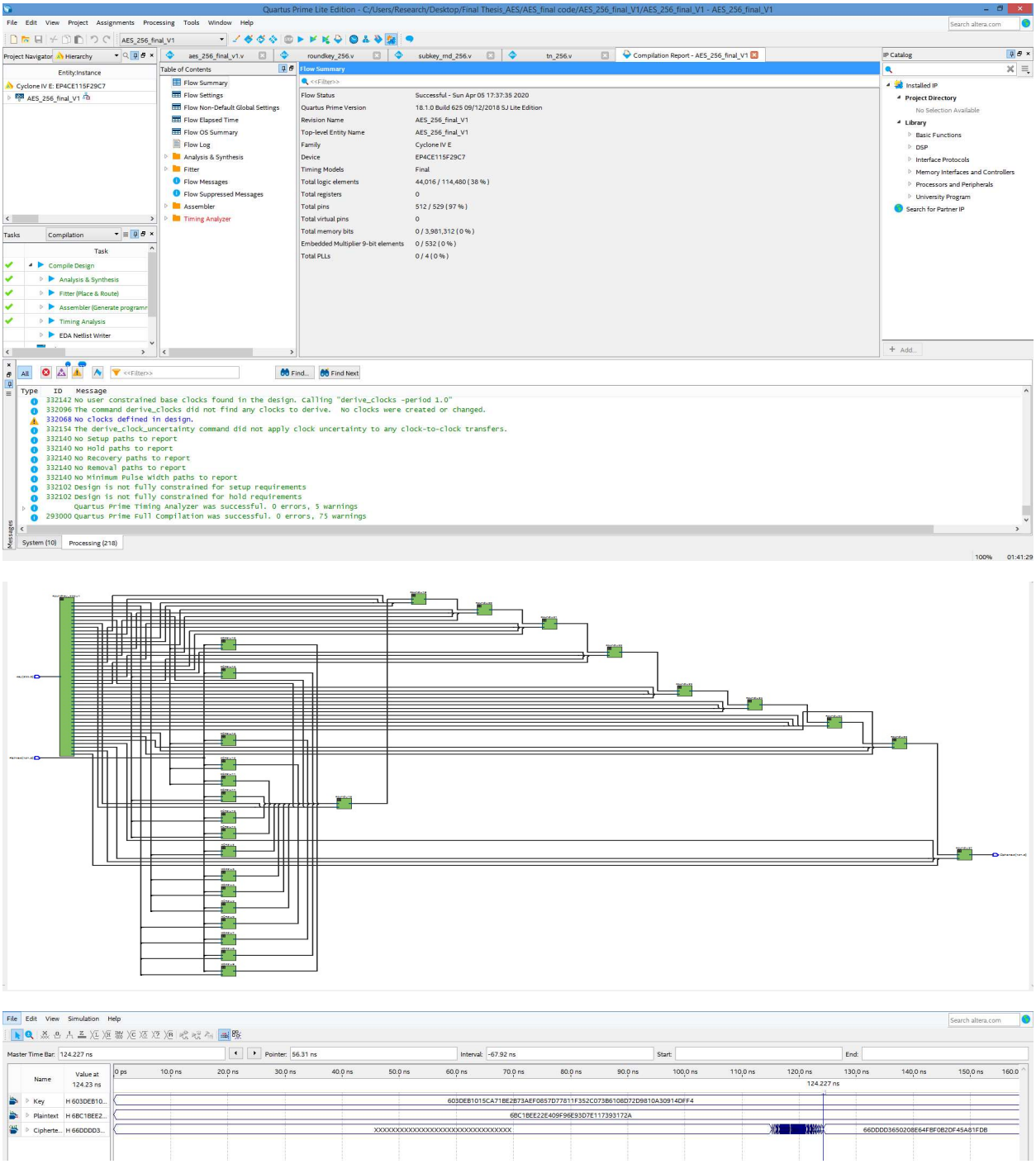

Figure 87: Timing Simulation For AES-256 after Round 10

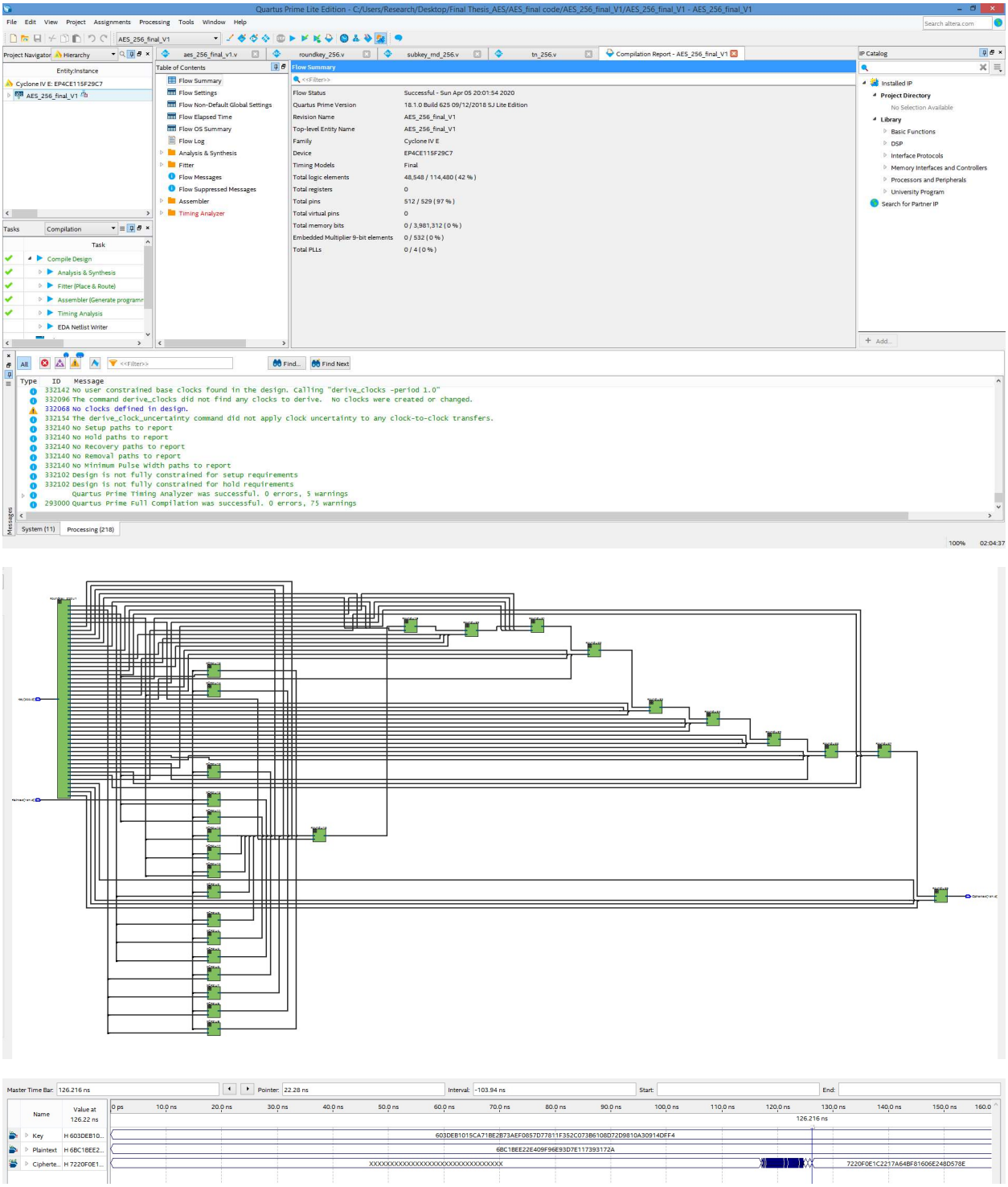

Figure 88: Timing Simulation For AES-256 after Round 11

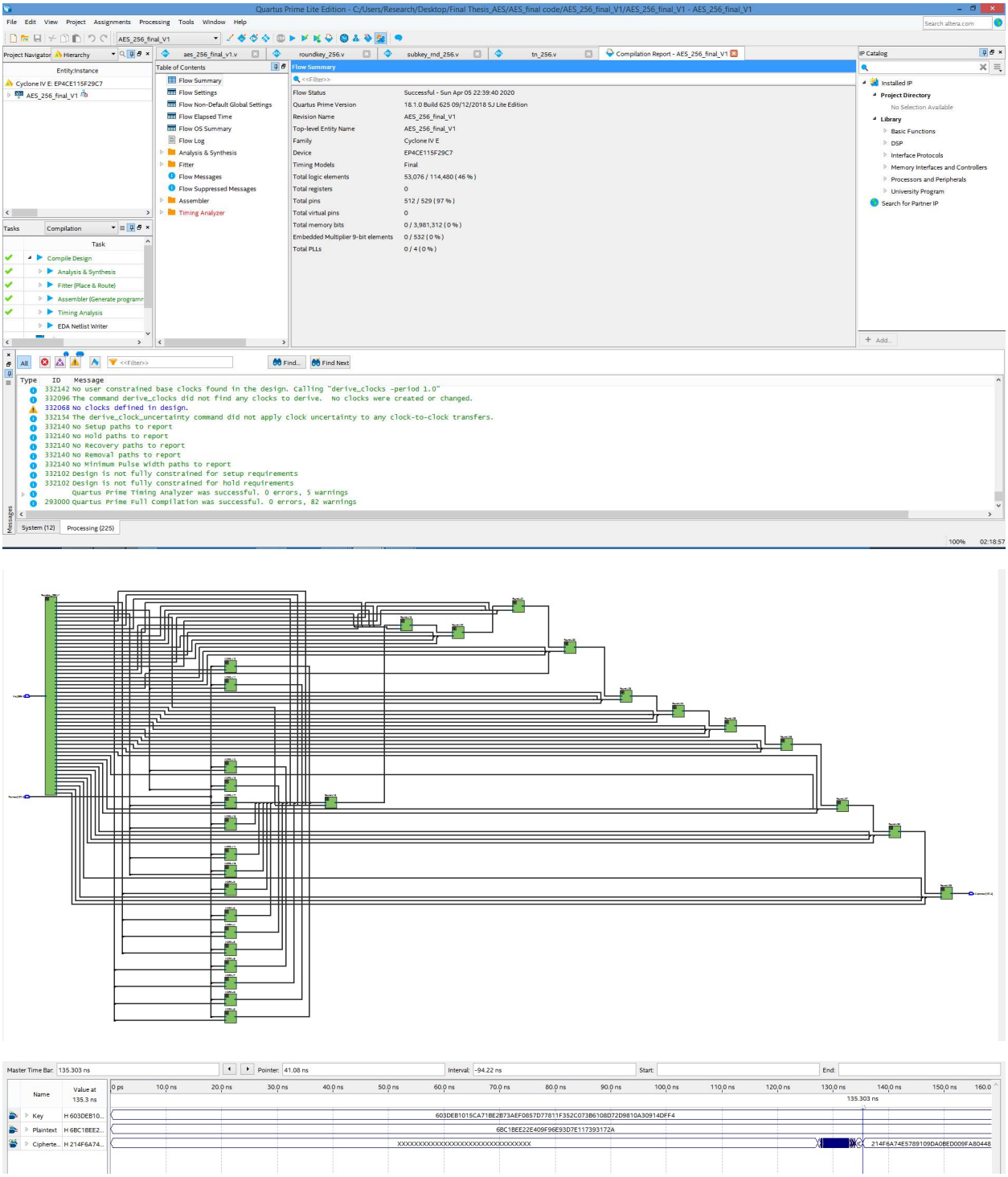

Figure 89: Timing Simulation For AES-256 after Round 12

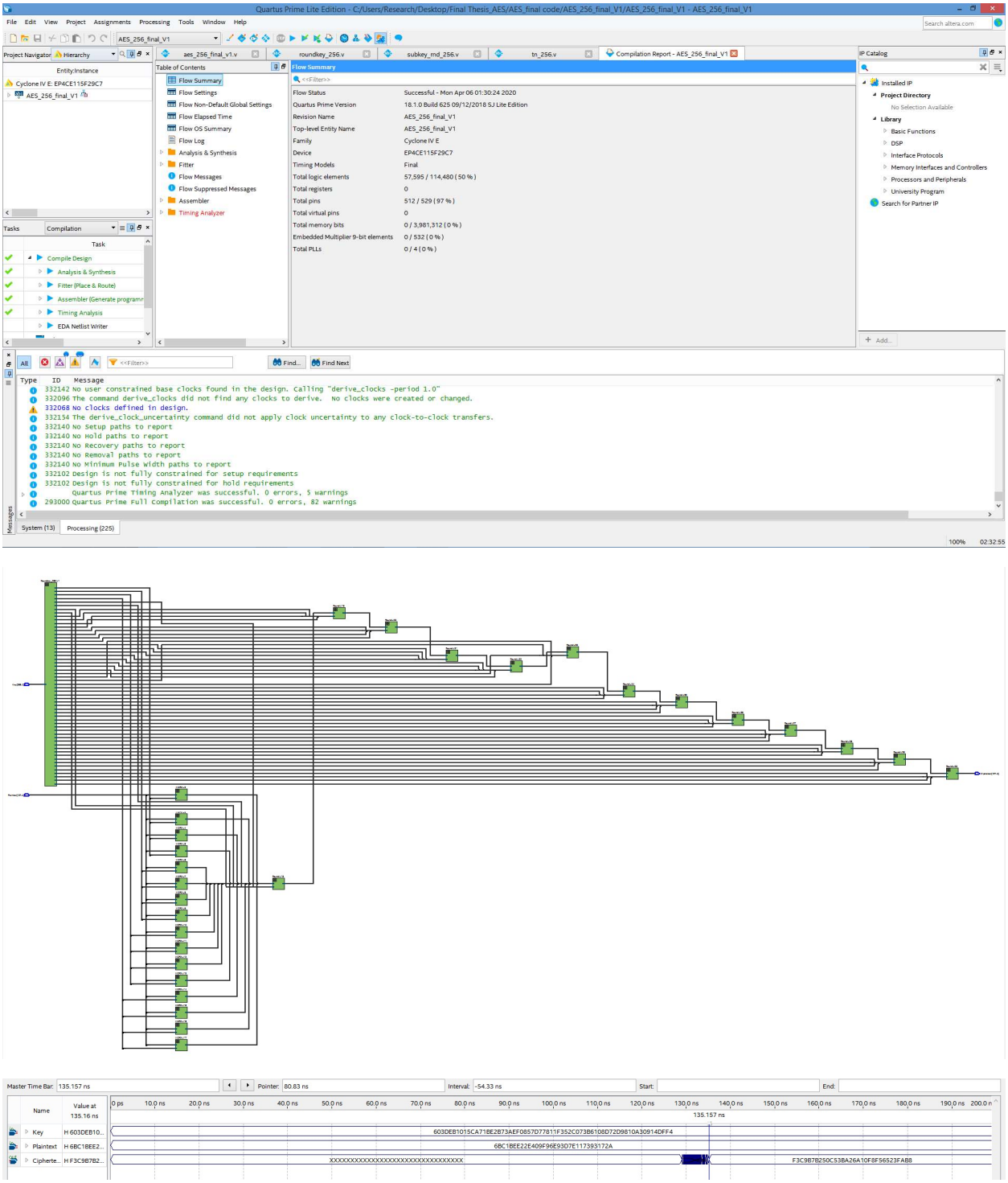

Figure 90: Timing Simulation For AES-256 after Round 13

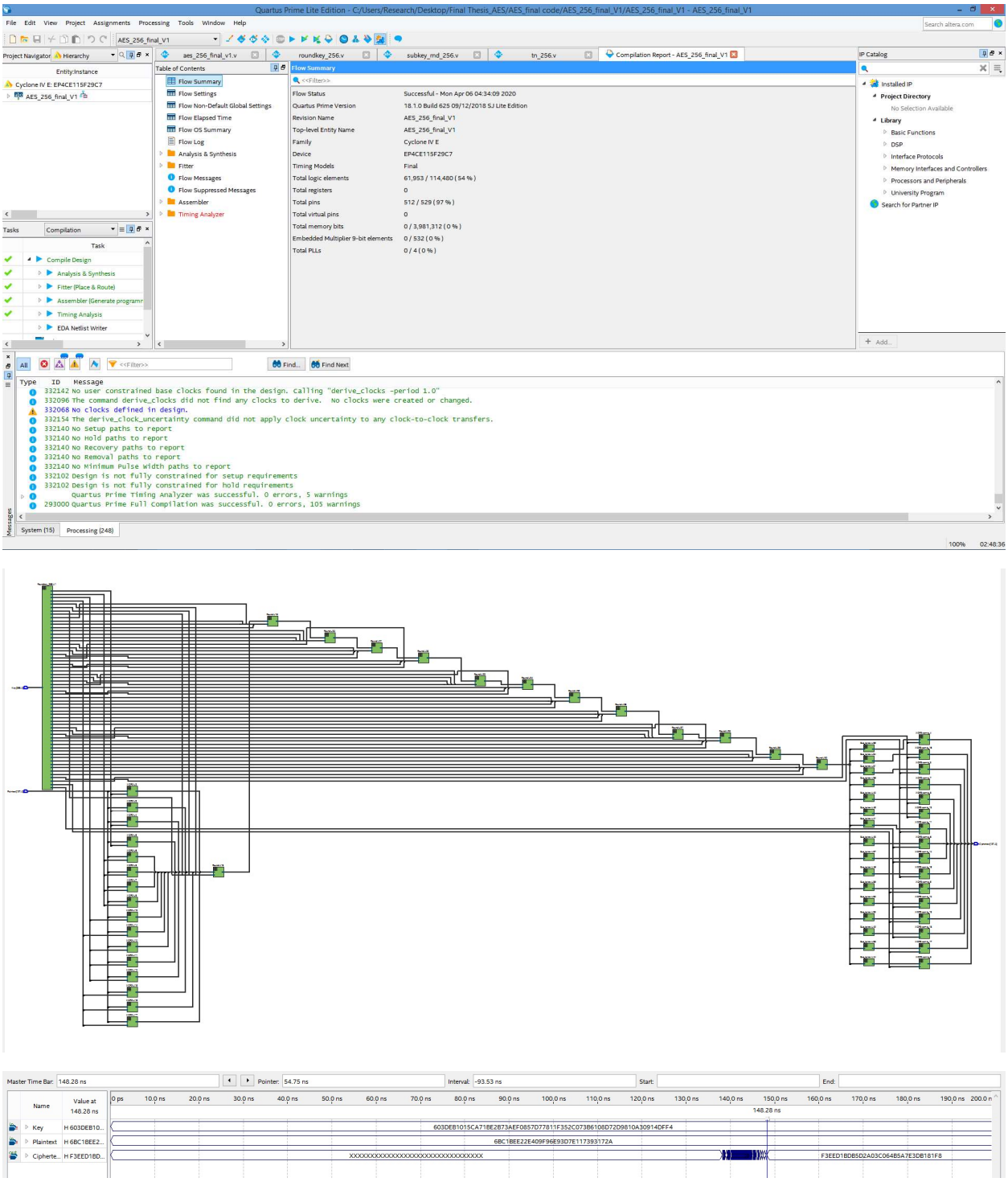

Figure 91: Timing Simulation For AES-256 after Round 14

Using the data collected from Figure 78-91, we built the Table 13 and created Figure 92 and Figure 93. The results in Figure 92 show a linear trend where an increase in round number results in an increase in delay time for the plaintext encryption portion. However, the processing time is not a linear trend as shown in Figure 93, because the processing time is based on the operation system.

Table 13: Delay_AES_256_ECB

	Key	KeyAdd (Ciphertext)	Logic elements	Processing time
Round1	*	22.15	3672	1:51
Round2	*	32.98	8138	3:39
Round3	*	43.29	12610	12:55
Round4	*	54.78	17096	23:15
Round5	*	70.15	21582	34:01
Round6	*	81.22	26044	46:26
Round7	*	91.90	30514	59:58
Round8	*	102.29	35007	1:13:51
Round9	*	107.28	39476	1:28:01
Round10	*	124.23	44016	1:41:29
Round11	*	126.22	48548	2:04:37
Round12	*	135.30	53076	2:18:57
Round13	*	135.16	57595	2:32:55
Round14	*	148.28	61953	2:48:36

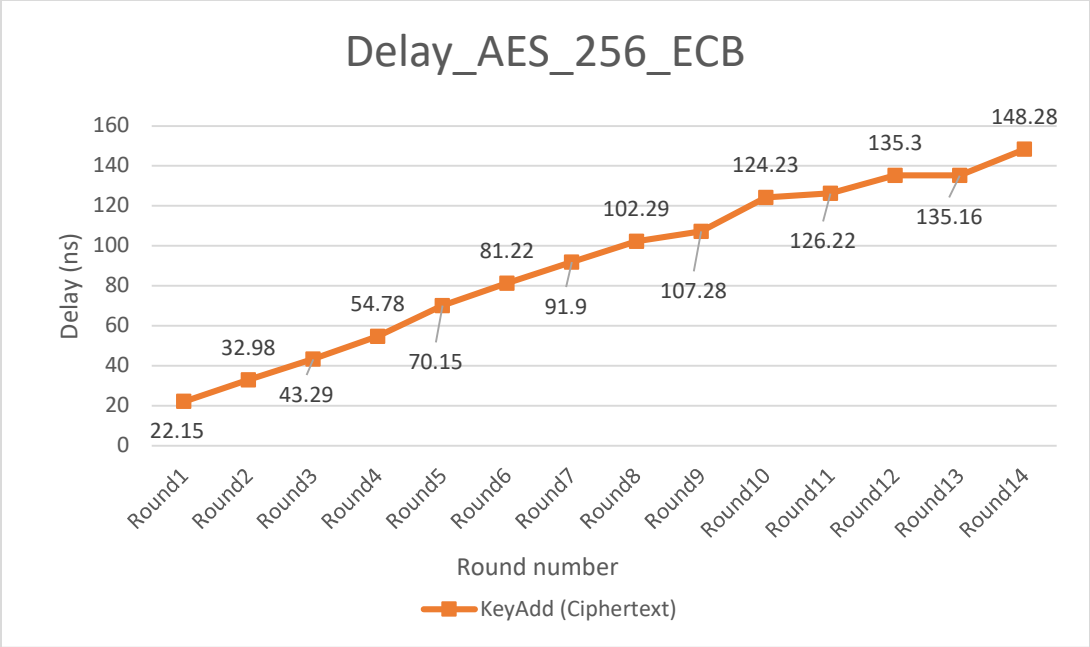

Figure 92: Delay_AES_256_ECB

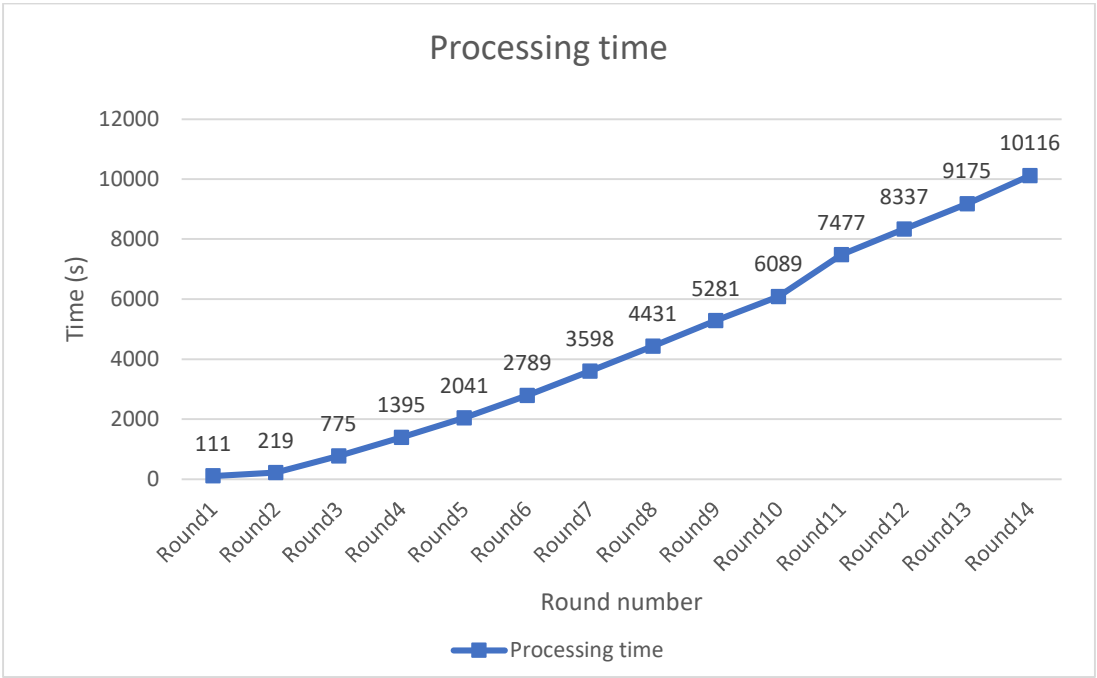

Figure 93: Processing time_ AES_256_ECB

5.2.4 Timing Simulation For AES-512

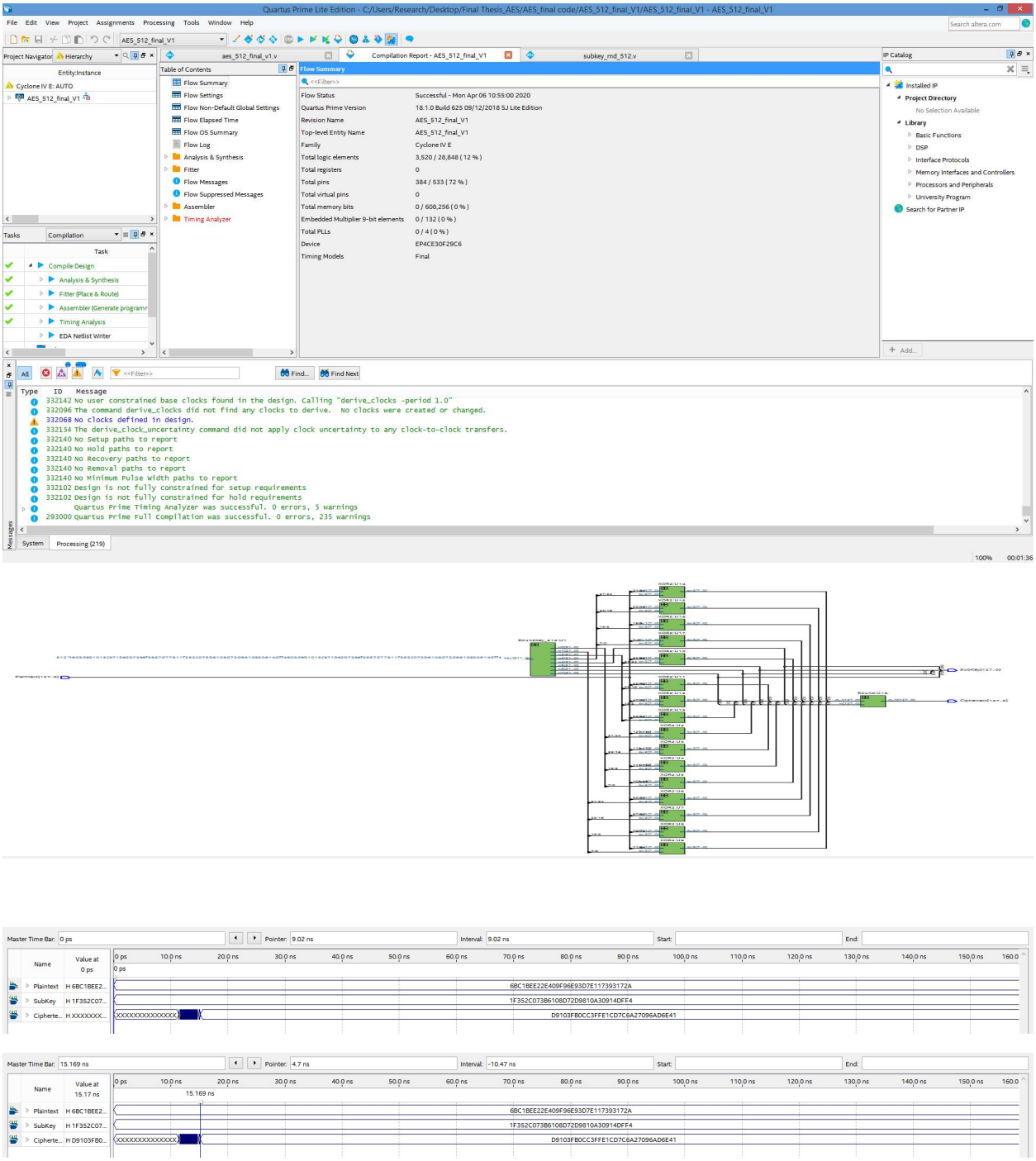

Figure 94: Timing Simulation For AES-512 after Round 1

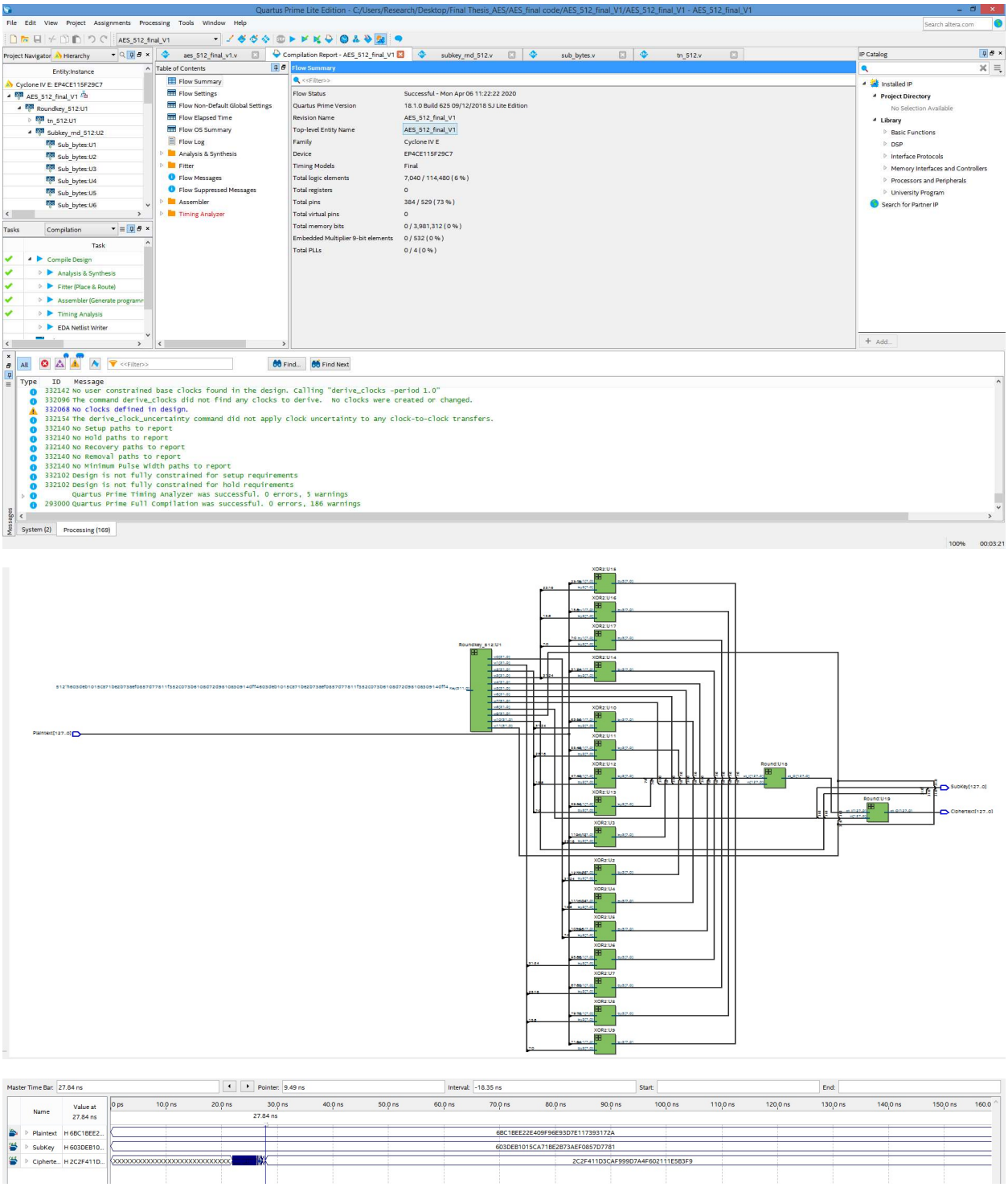

Figure 95: Timing Simulation For AES-512 after Round 2

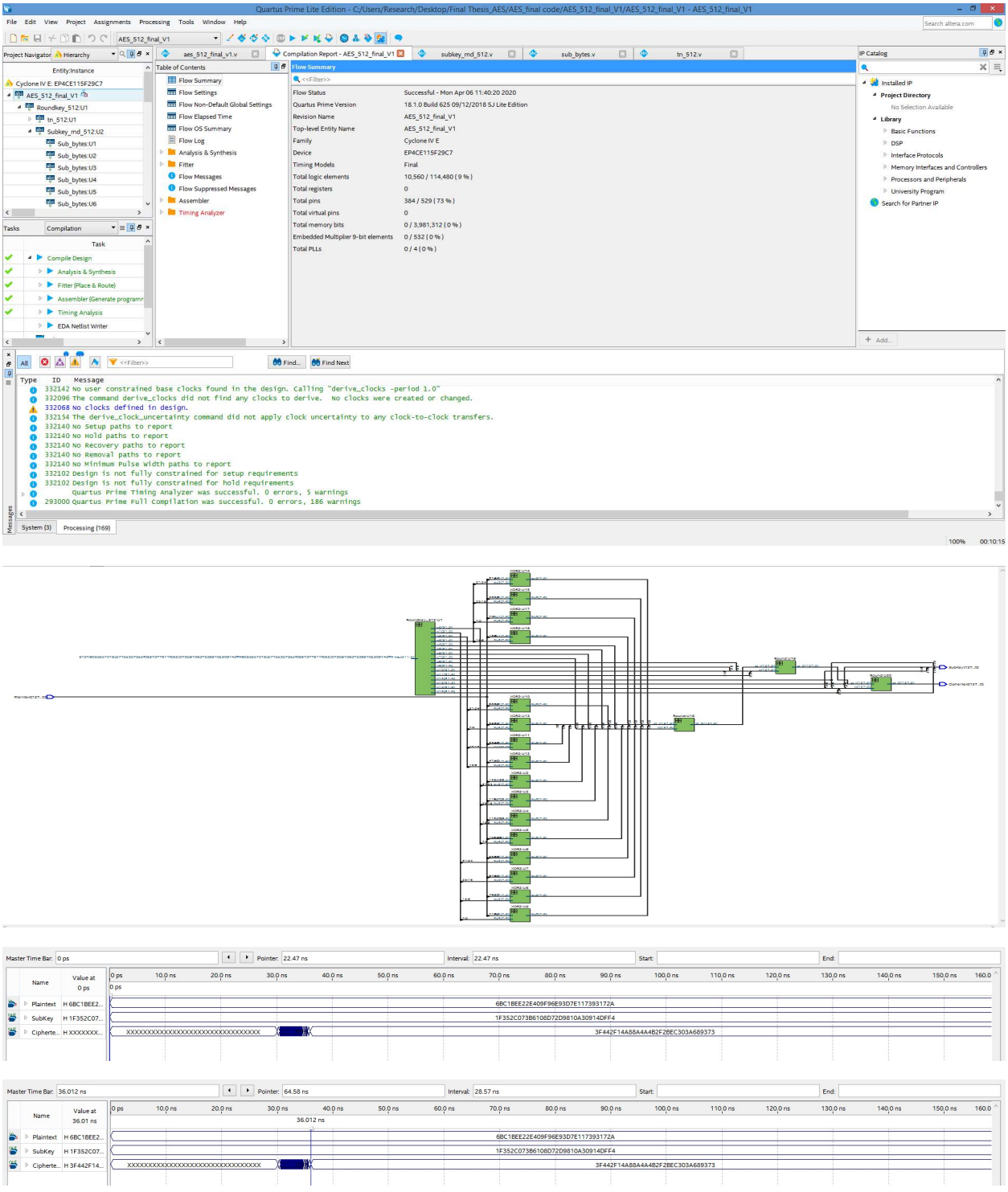

Figure 96: Timing Simulation For AES-512 after Round 3

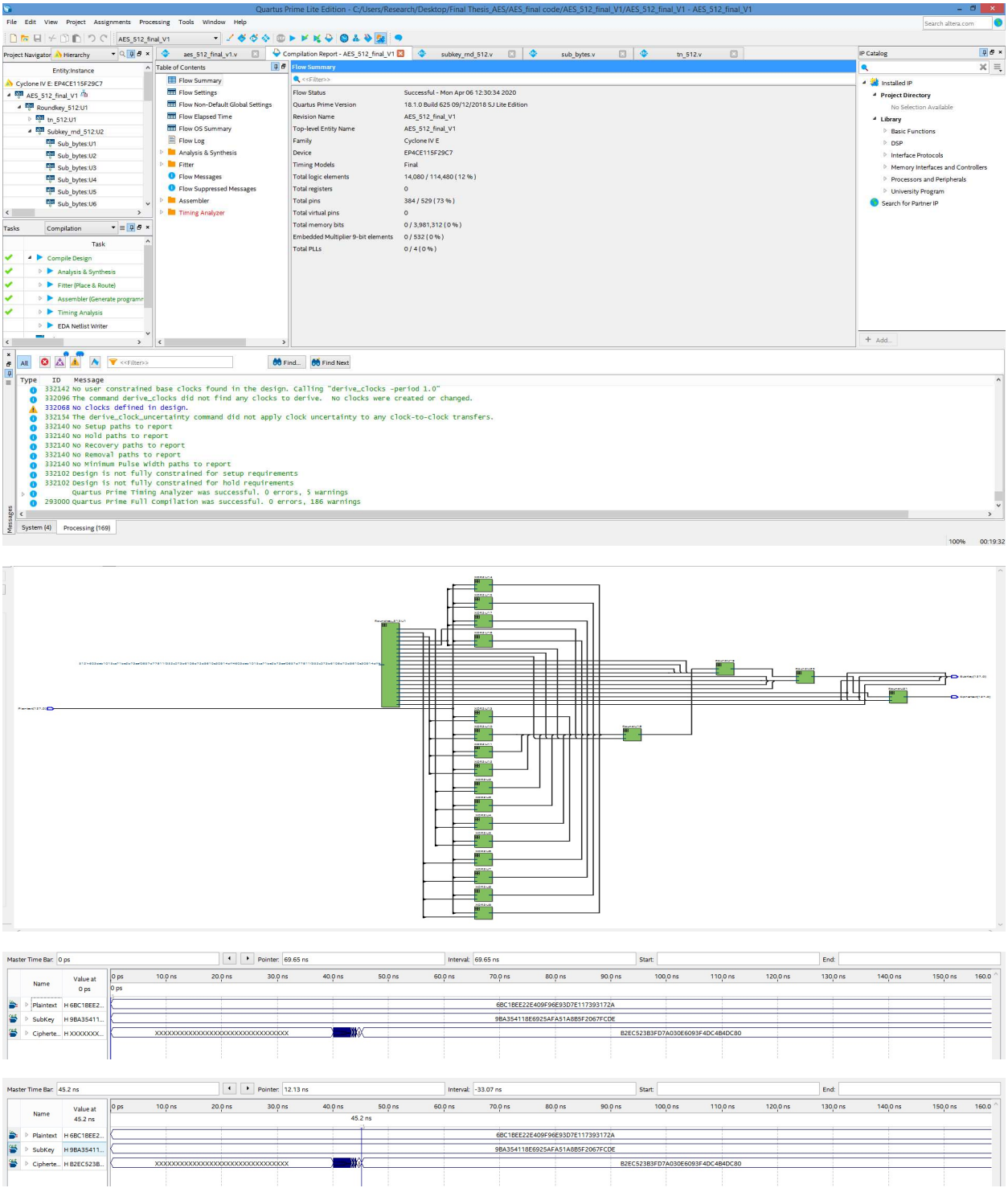

Figure 97: Timing Simulation For AES-512 after Round 4

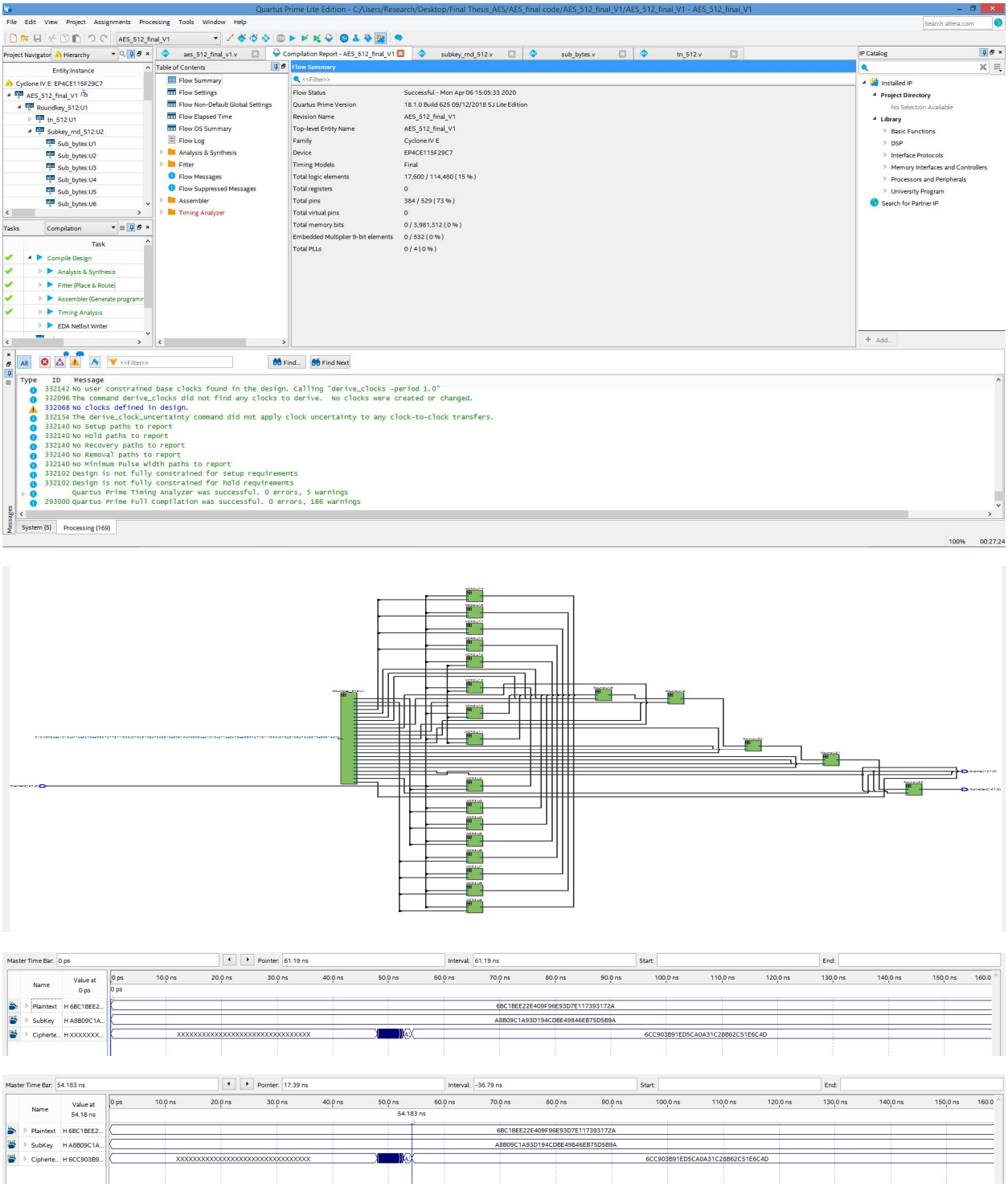

Figure 98: Timing Simulation For AES-512 after Round 5

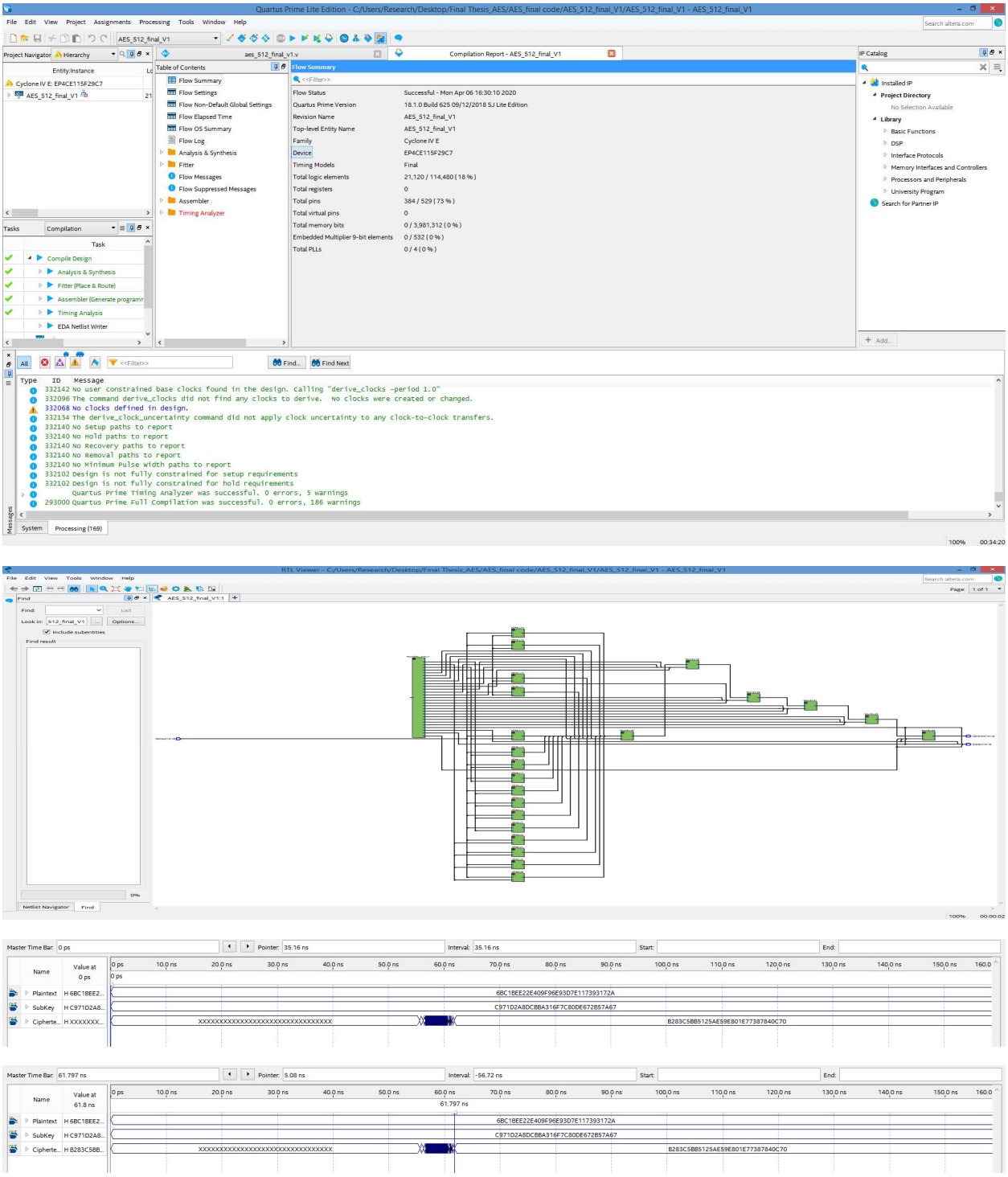

Figure 99: Timing Simulation For AES-512 after Round 6

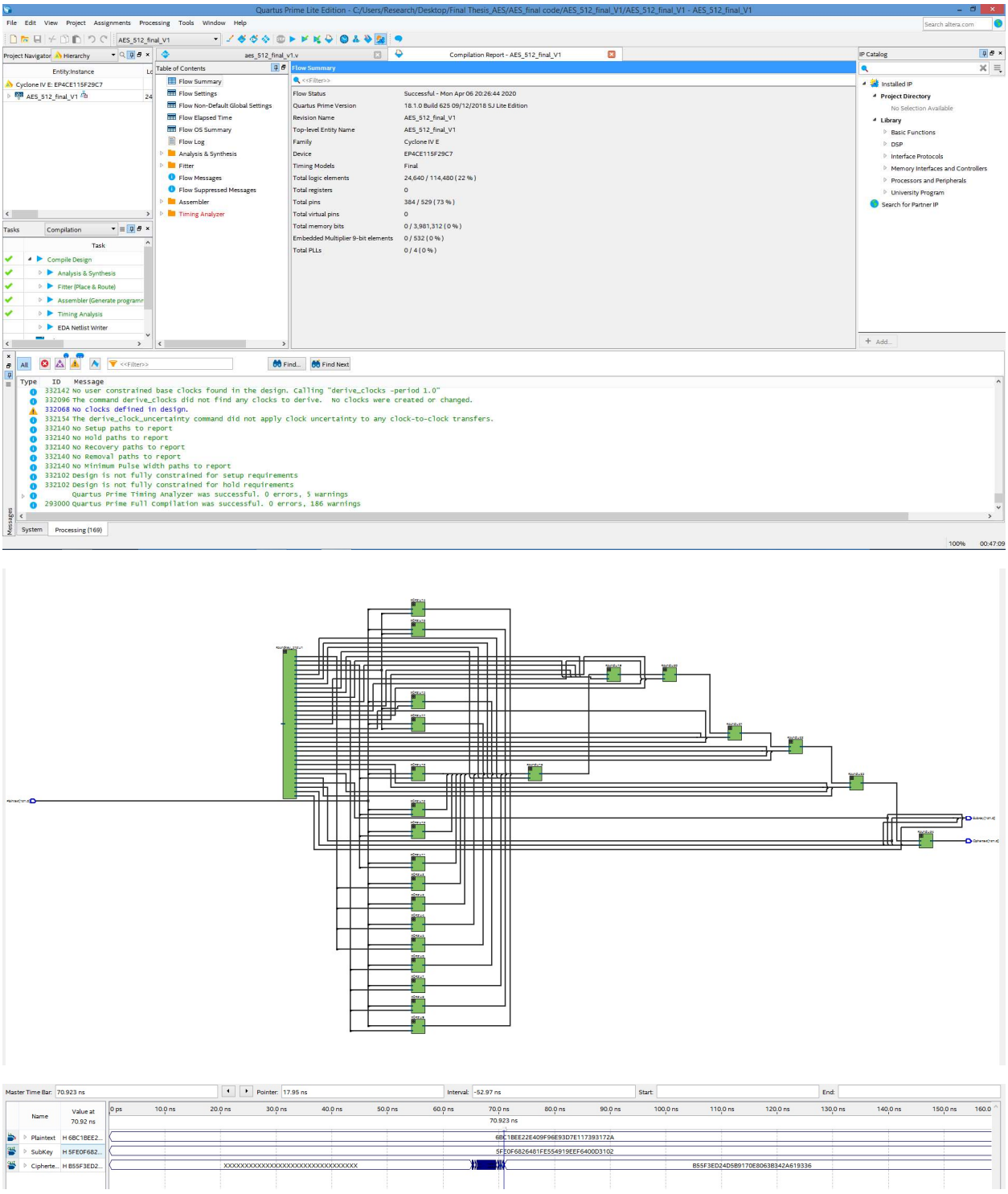

Figure 100: Timing Simulation For AES-512 after Round 7

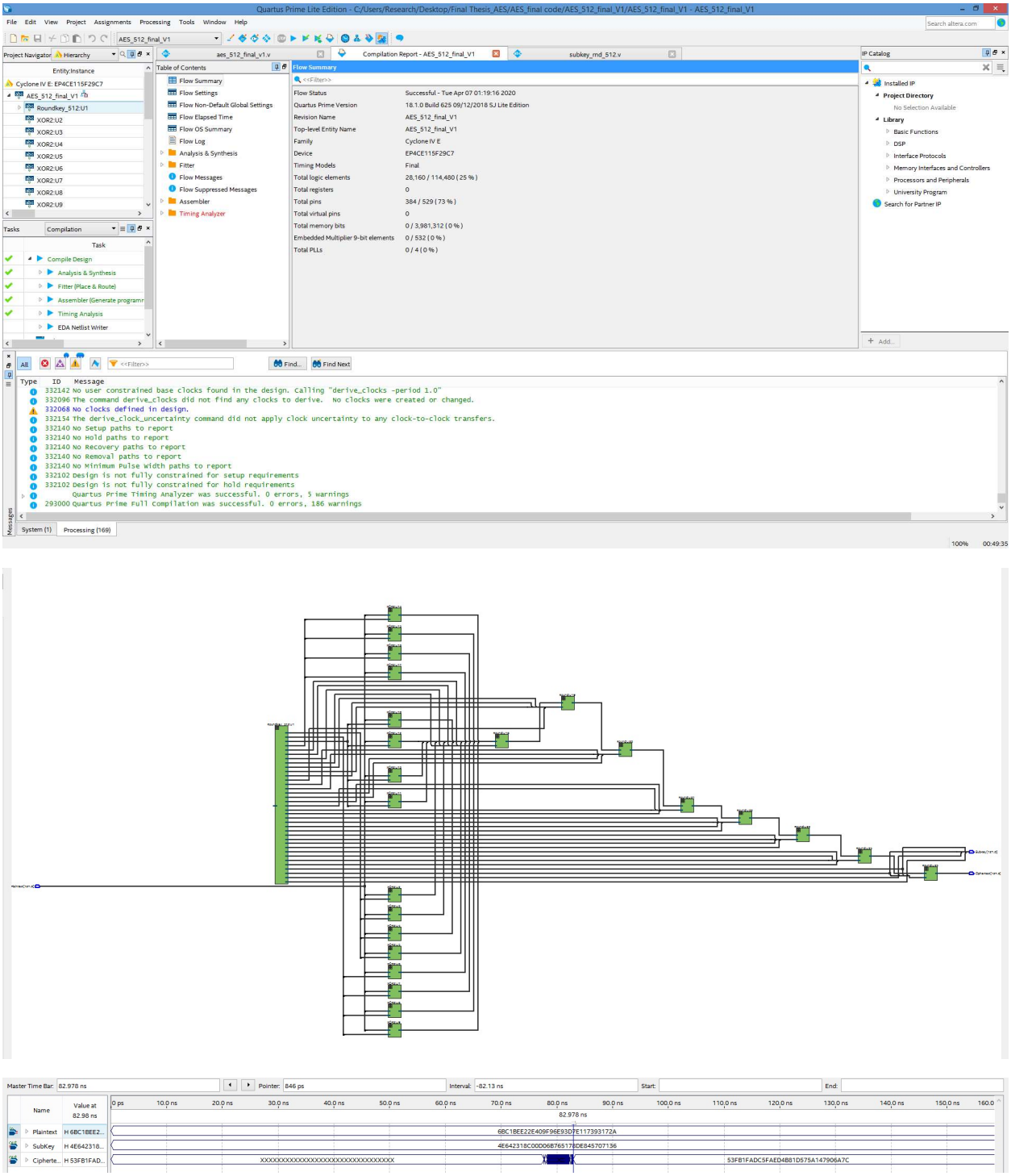

Figure 101: Timing Simulation For AES-512 after Round 8

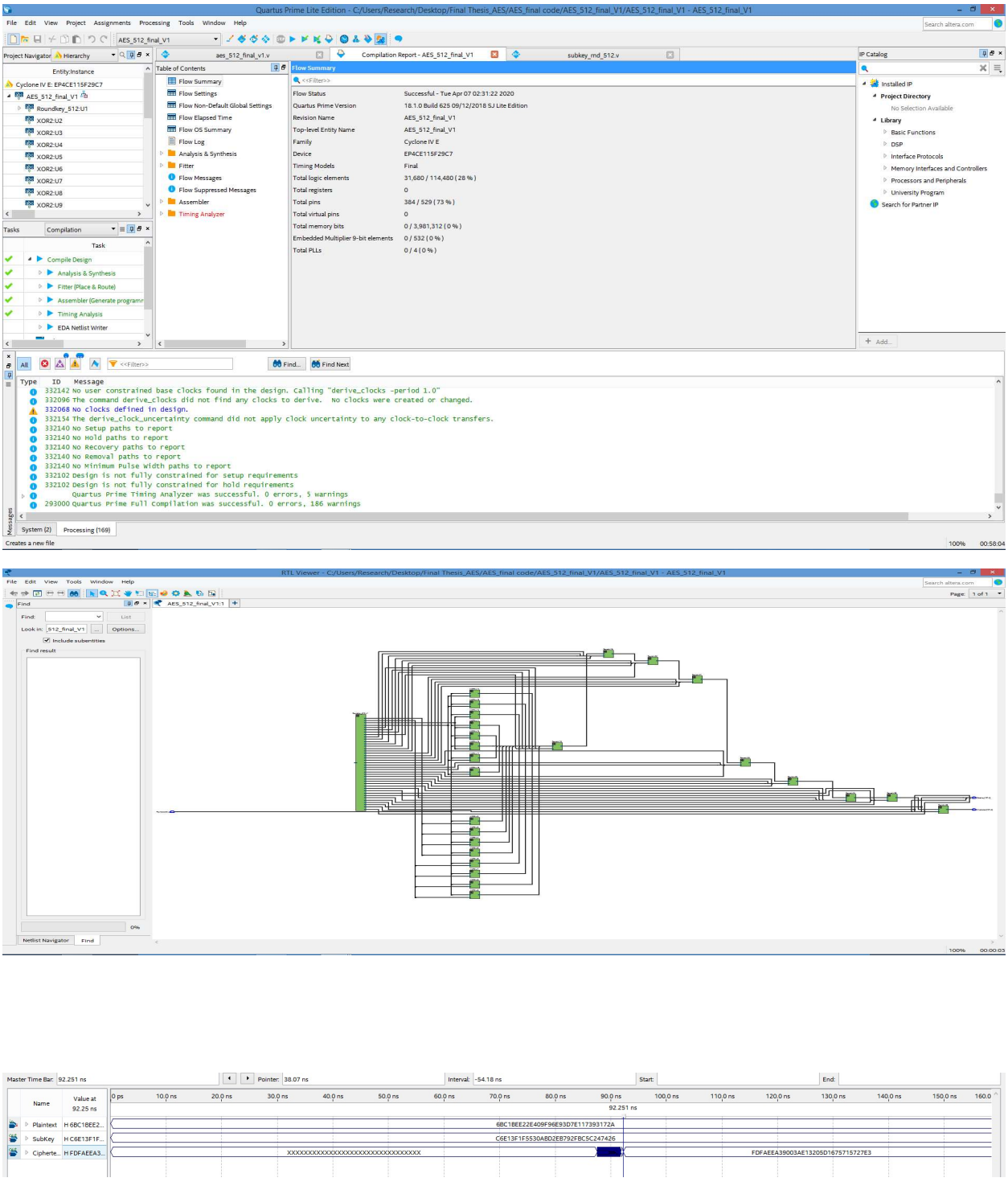

Figure 102: Timing Simulation For AES-512 after Round 9

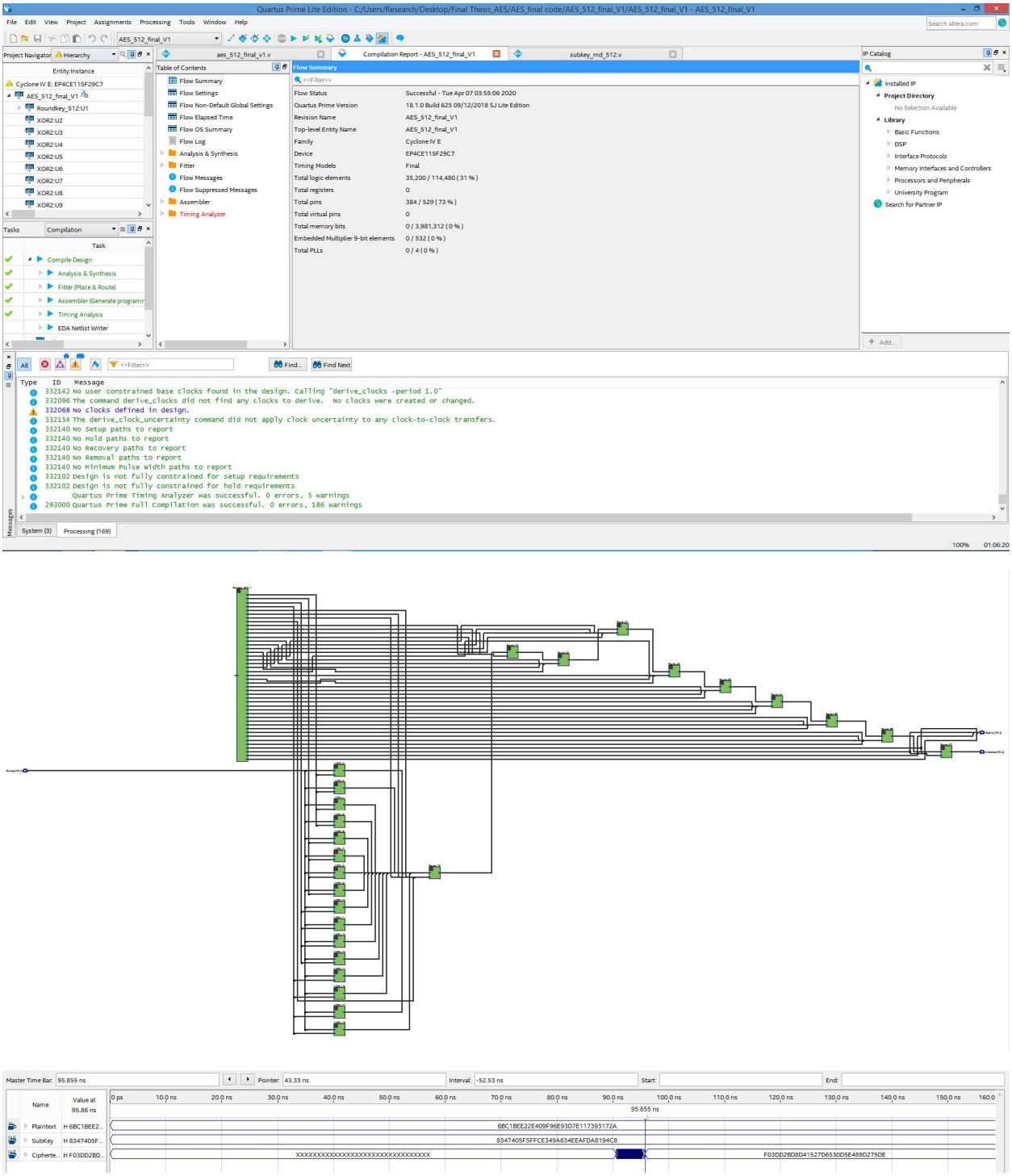

Figure 103: Timing Simulation For AES-512 after Round 10

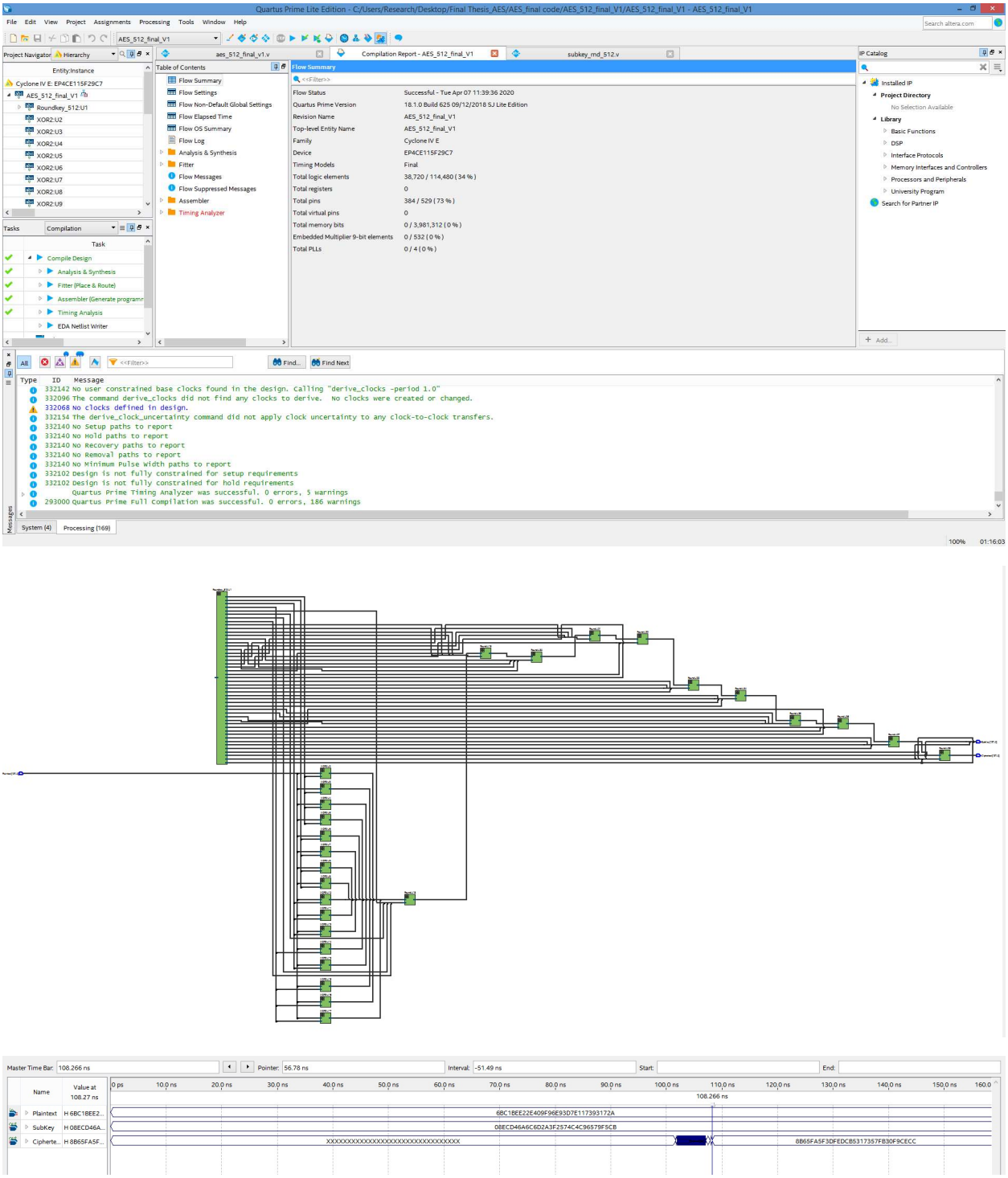

Figure 104: Timing Simulation For AES-512 after Round 11

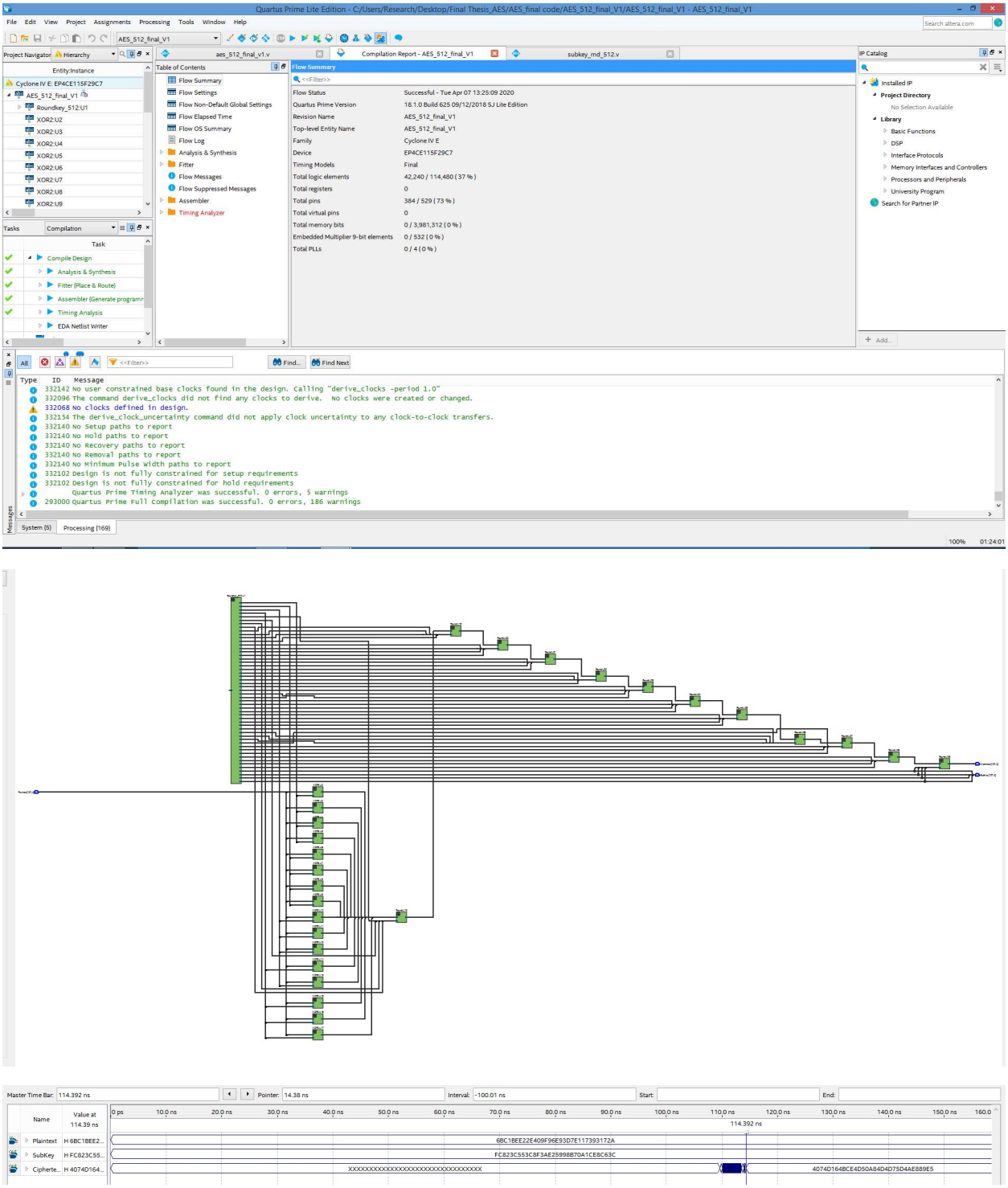

Figure 105: Timing Simulation For AES-512 after Round 12

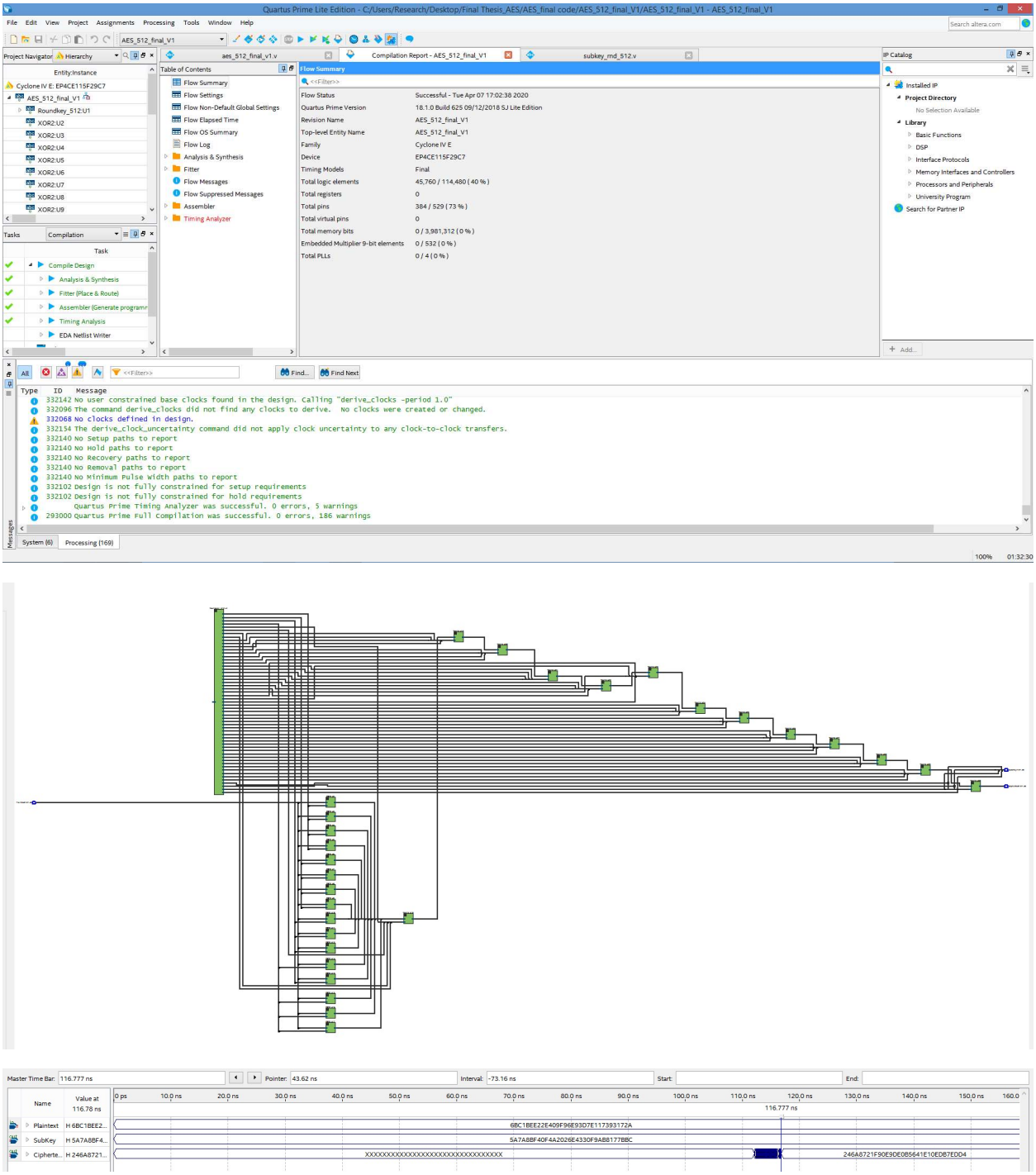

Figure 106: Timing Simulation For AES-512 after Round 13

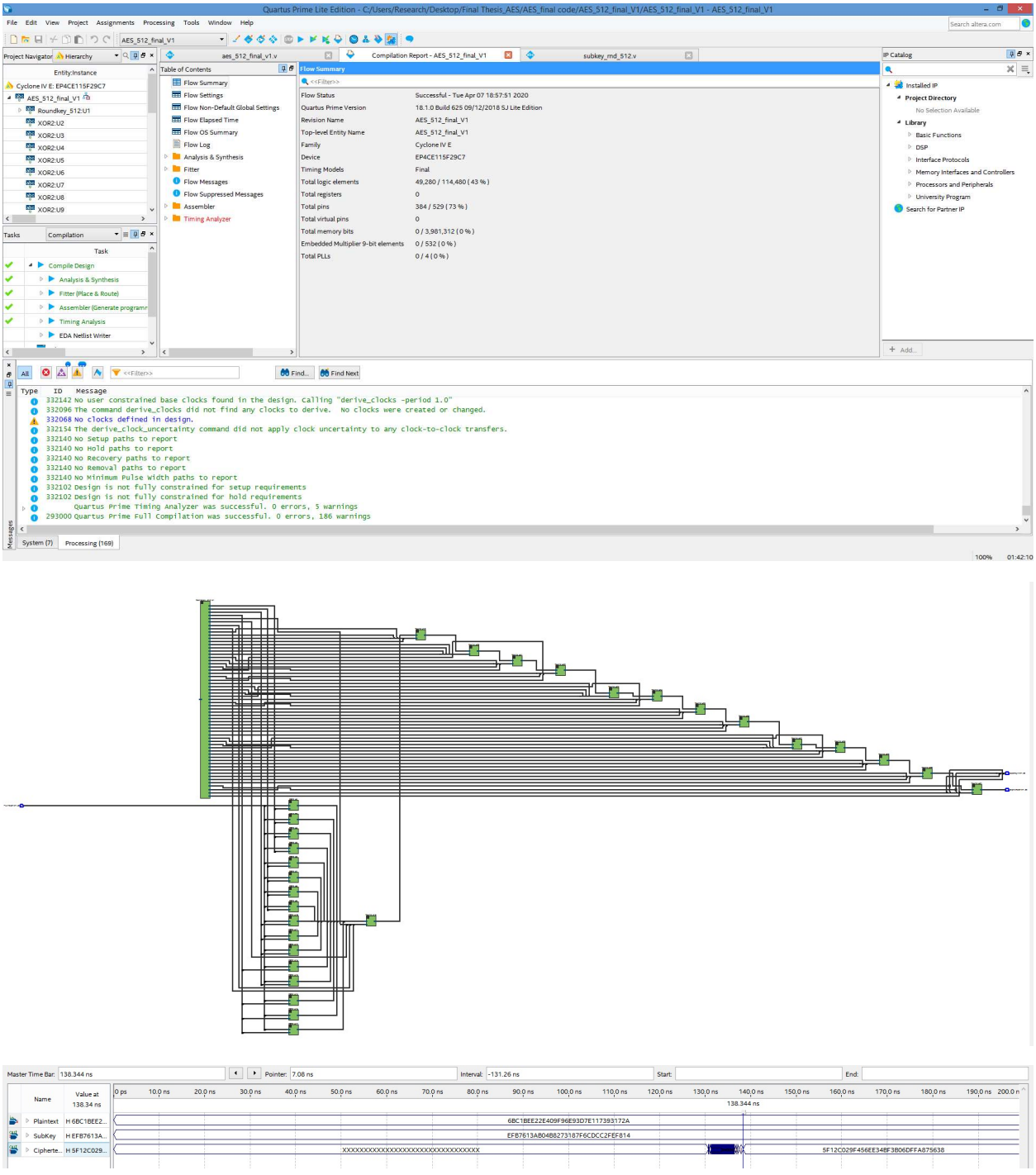

Figure 107: Timing Simulation For AES-512 after Round 14

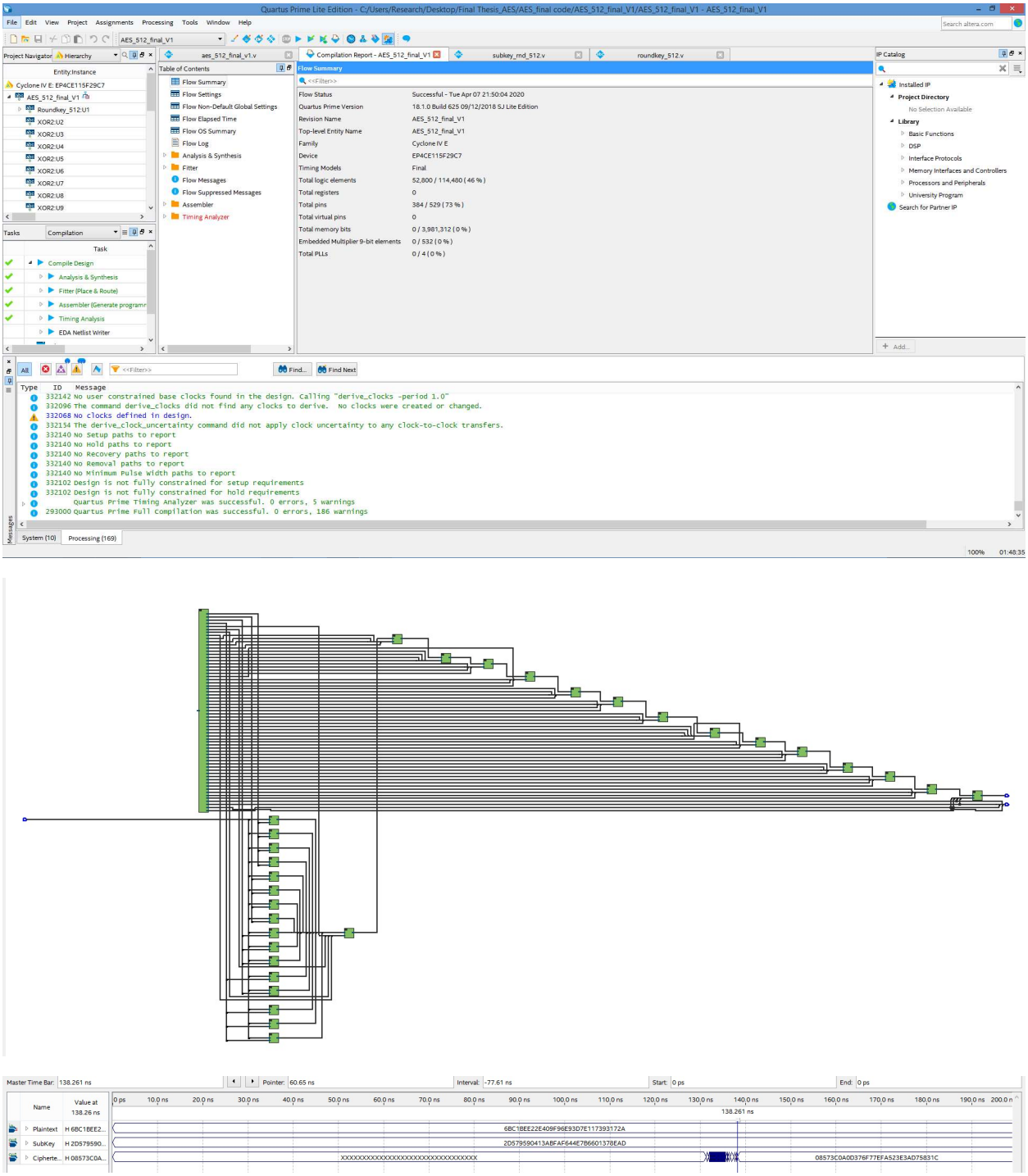

Figure 108: Timing Simulation For AES-512 after Round 15

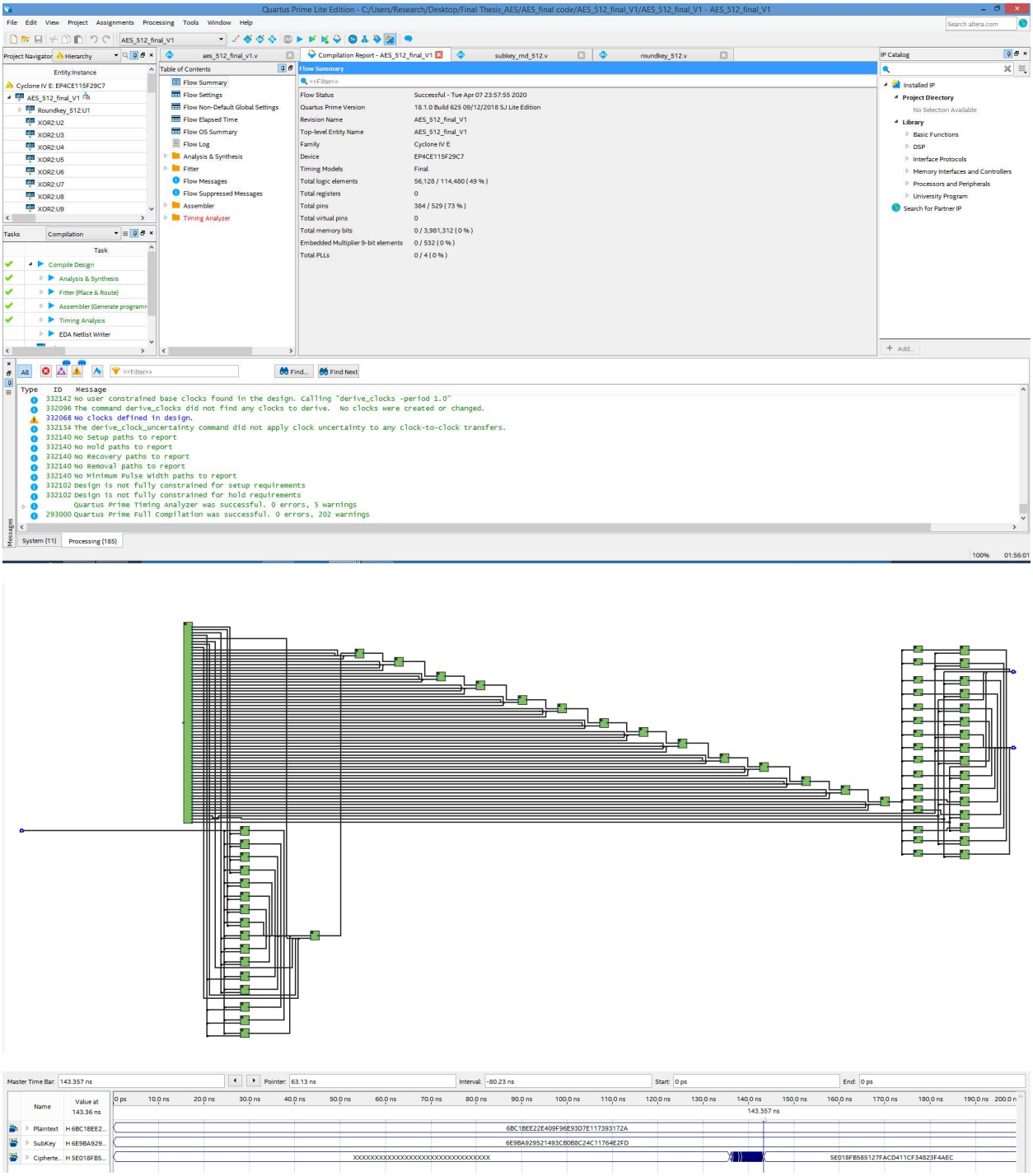

Figure 109: Timing Simulation For AES-512 after Round 16

Using the data collected from Figure 94-109, we built the Table 14 and created Figure 110 and Figure 111.

Table 14: Delay_AES_512_ECB

	Key	KeyAdd (Ciphertext)	Logic elements (114480)	Processing time
Round1	*	15.17	3520	1:36
Round2	*	27.84	7040	3:21
Round3	*	36.01	10560	10:15
Round4	*	45.20	14080	19:32
Round5	*	54.18	17600	27:24
Round6	*	61.80	21120	34:20
Round7	*	70.92	24640	47:09
Round8	*	82.98	28160	49:35
Round9	*	92.25	31680	58:04
Round10	*	95.86	35200	1:06:20
Round11	*	108.27	38720	1:16:03
Round12	*	114.39	42240	1:24:01
Round13	*	116.78	45760	1:32:30
Round14	*	138.34	49280	1:42:10
Round15	*	138.26	52800	1:48:35
Round16	*	143.36	56128	1:56:01

The results in Figure 110 show a linear trend where an increase in round number results in an increase in delay time for the plaintext encryption portion. However, the processing time is not a linear trend as shown in Figure 111, because the processing time is based on the operating system.

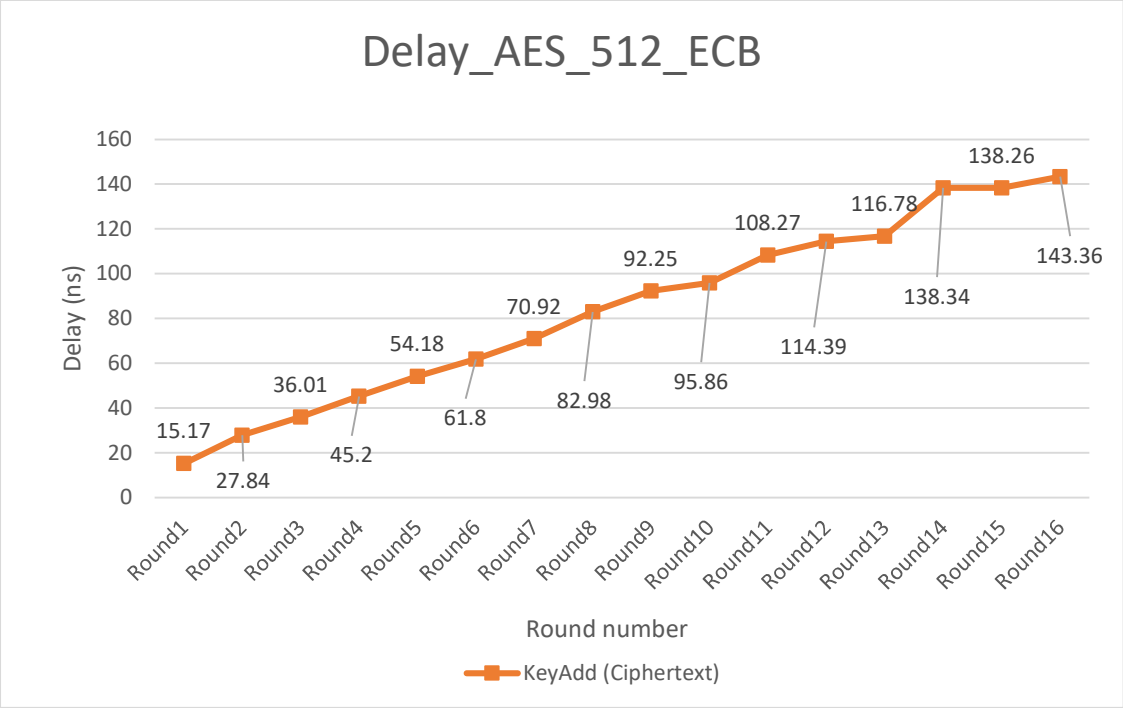

Figure 110: Delay_AES_512_ECB

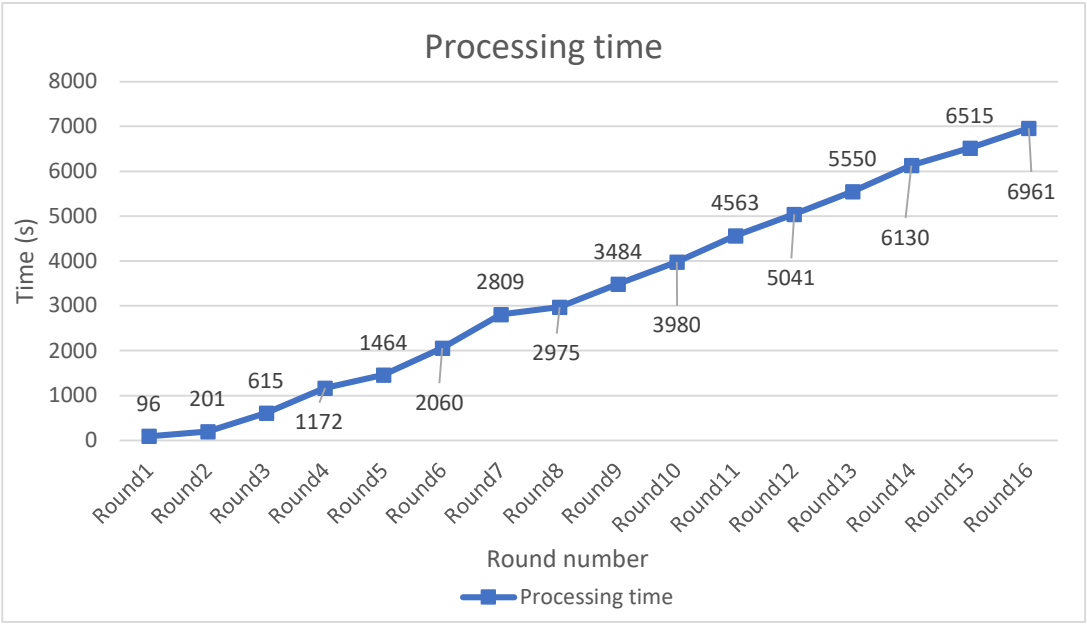

Figure 111: Processing time_ AES_512_ECB

5.3 Timing Simulation For AES-128, AES-192, AES, 256 and AES-512 in CBC Mode

To implement the AES algorithm in CBC mode, we simply used the module we designed for ECB mode twice and we only run the simulation for the final cyphertext of AES-128, AES-192, AES, 256 and AES-512 in CBC Mode. However, since there are limited logic elements and total pins, we had to put the key into the program code.

5.3.1 Timing Simulation of AES-128

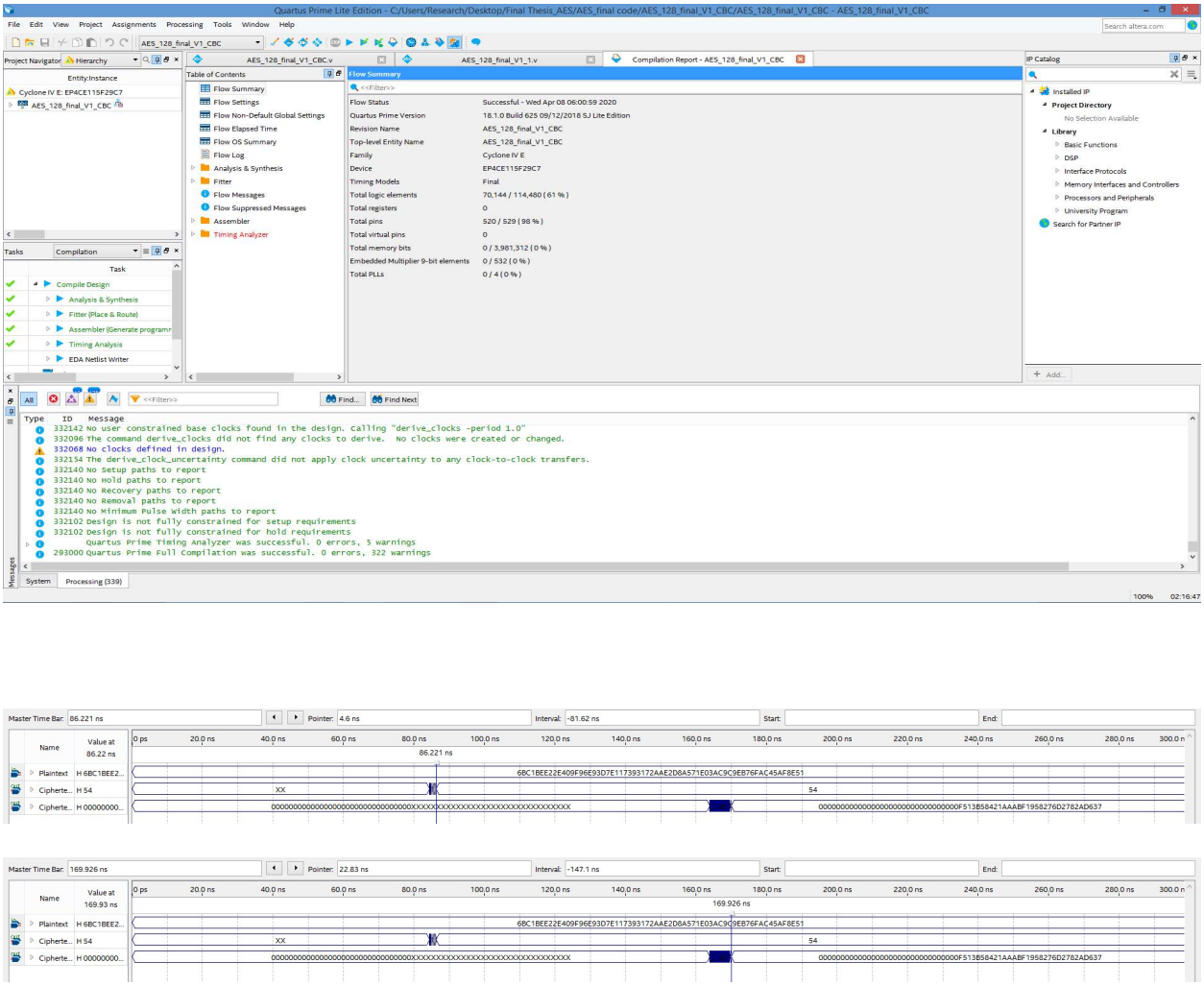

Figure 112: Timing Simulation For AES-128 in CBC Mode

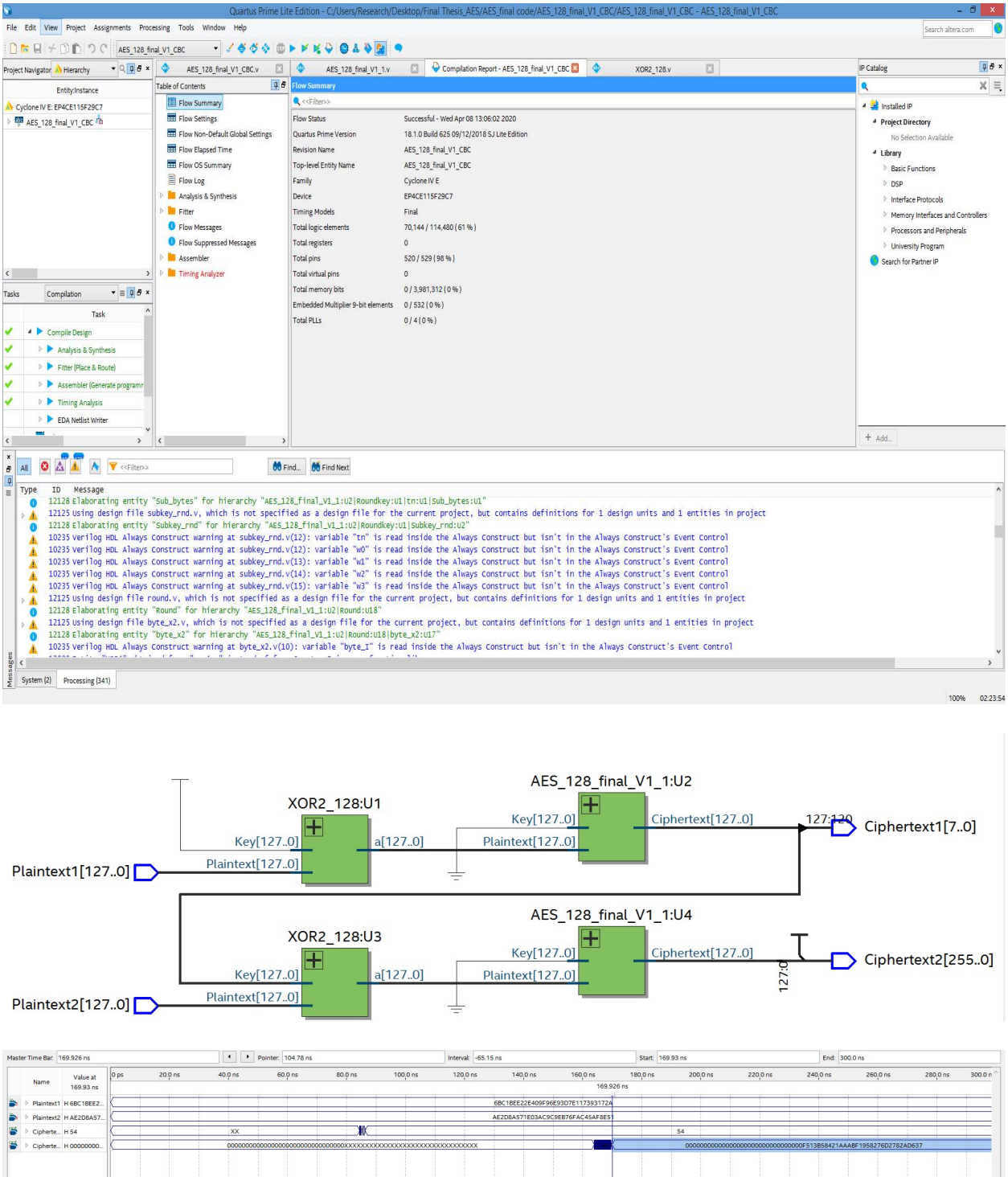

Figure 113: Timing Simulation For AES-128_1 in CBC Mode

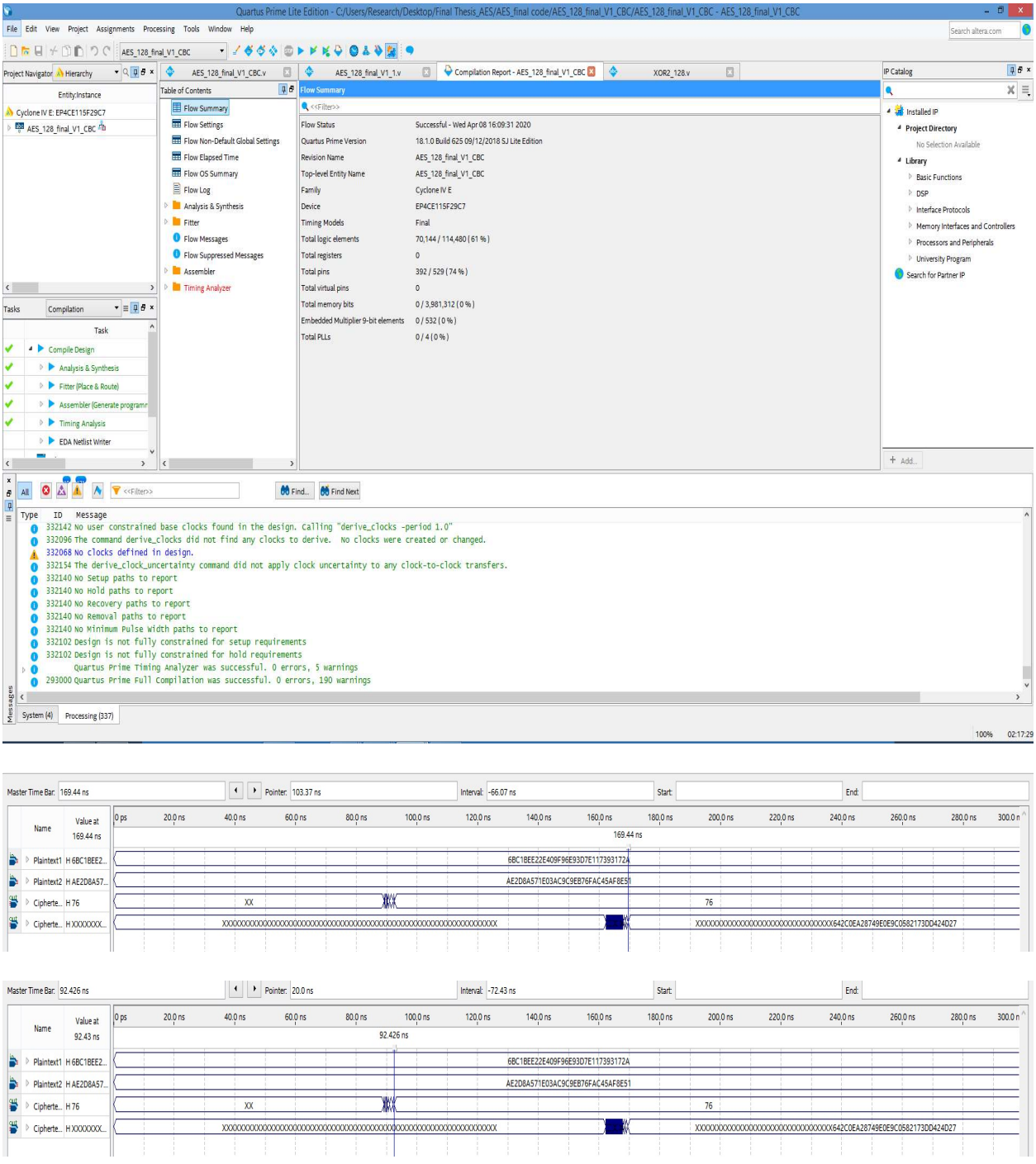

Figure 114: Timing Simulation For AES-128_2 in CBC Mode

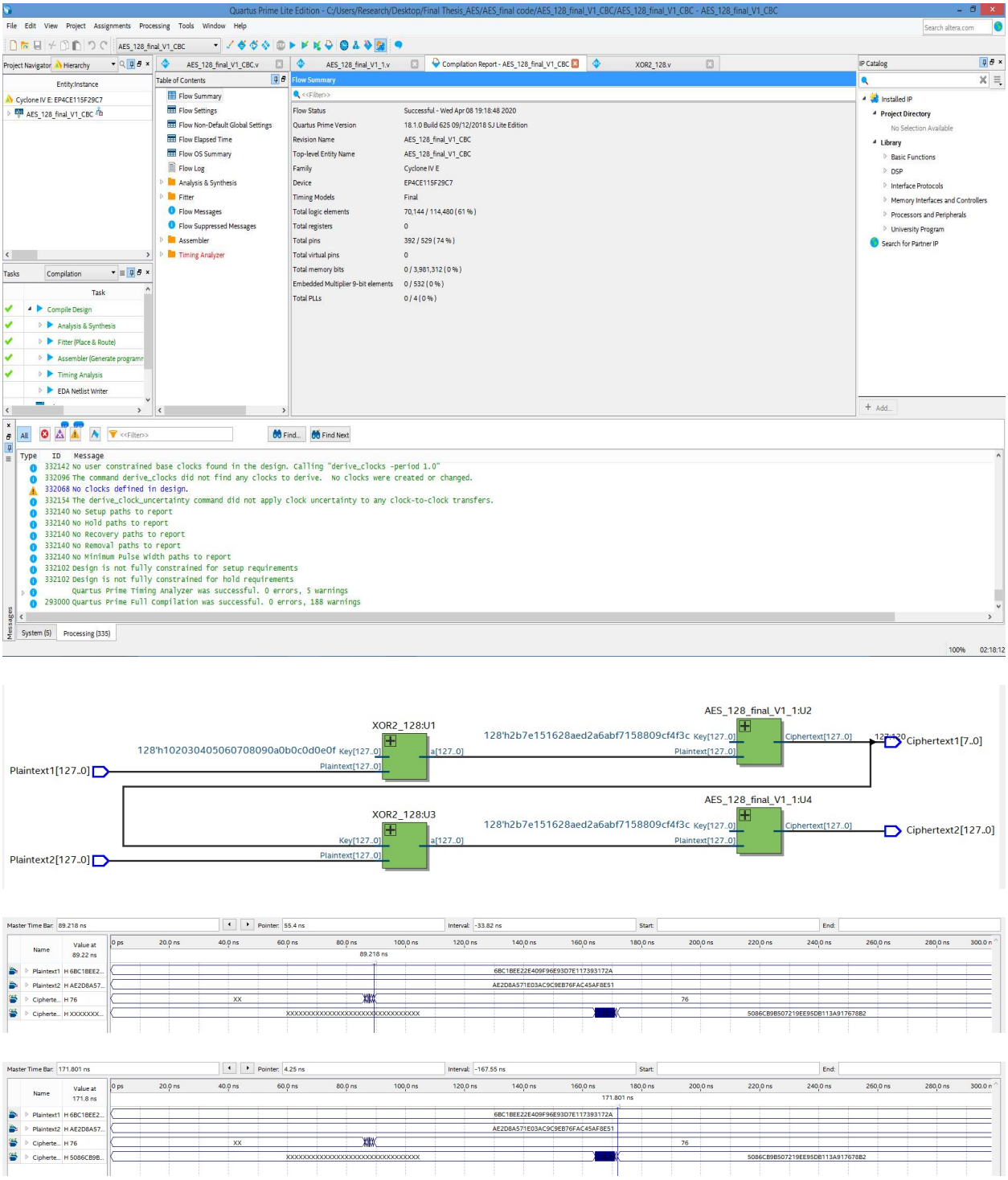

Figure 115: Timing Simulation For AES-128_3 in CBC Mode

Since there are limited logic elements and total pins, we had to put the key into the program code. Using the data collected from Figure 112-115, we were able to build the Table 15 and created Figure 116, 117 and 118. The results in Figure 116-118 show there are only minor differences for delay, total logic elements, and processing time between our design in CBC mode. However, the average delay of the first Ciphertext in CBC mode is only 88.52 ns which is 23.7 % less than the delay of the first Ciphertext in ECB mode. It is because of the design of the code that is we had to put the key into the program code. Using the same strategy putting the key into the program code, we run the timing simulation for AES-192, AES-256 and AES-512 in CBC Mode as shown in Figure 119-121.

Table 15: Delay_AES_128_CBC

	Ciphertext 1	Ciphertext 2	Logic elements	Processing time
AES_128	86.22	169.93	70144	2:16:47
AES_128_1	86.22	169.93	70144	2:23:54
AES_128_2	92.43	169.44	70144	2:17:29
AES_128_3	89.22	171.8	70144	2:18:12

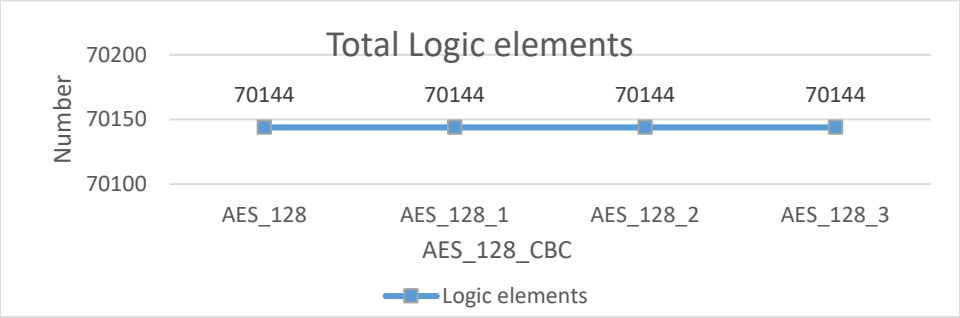

Figure 116: Total Logic elements_AES_128_CBC

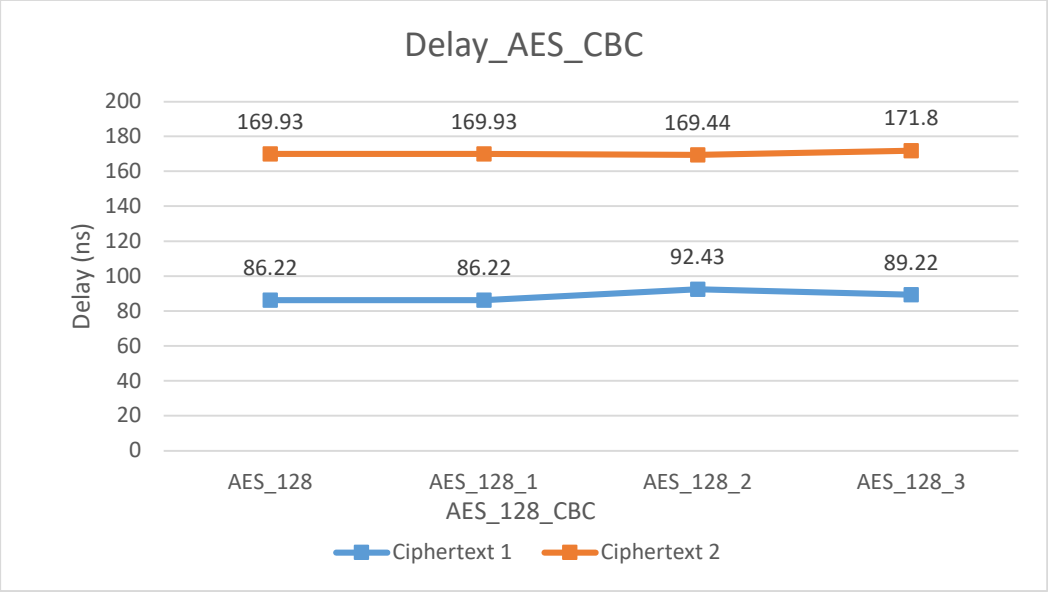

Figure 117: Delay_AES_128_CBC

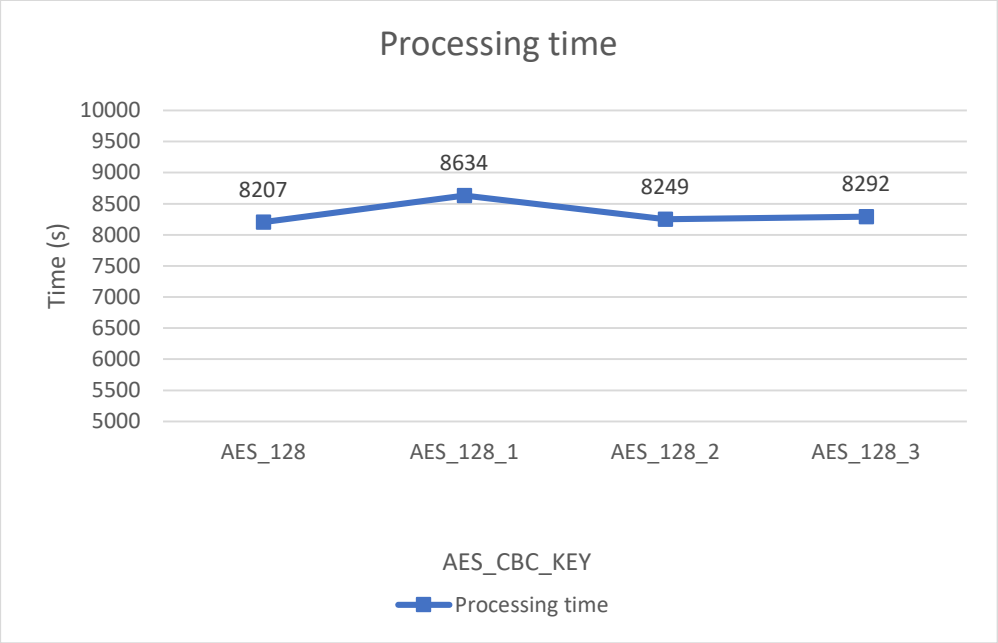

Figure 118: Processing time_AES_128_CBC

5.3.2 Timing Simulation of AES-192

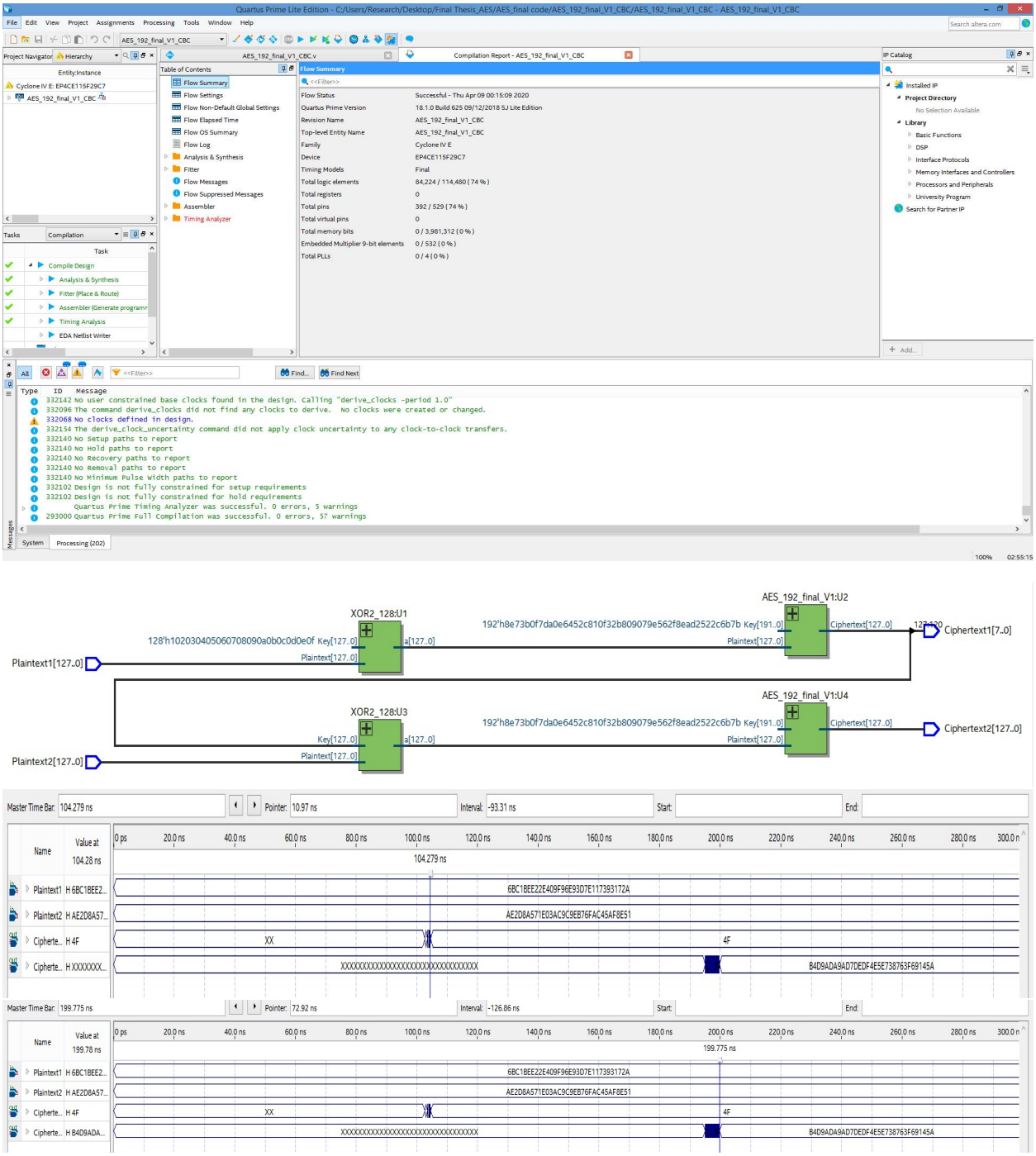

Figure 119: Timing Simulation For AES-192 in CBC Mode

5.3.3 Timing Simulation of AES-256

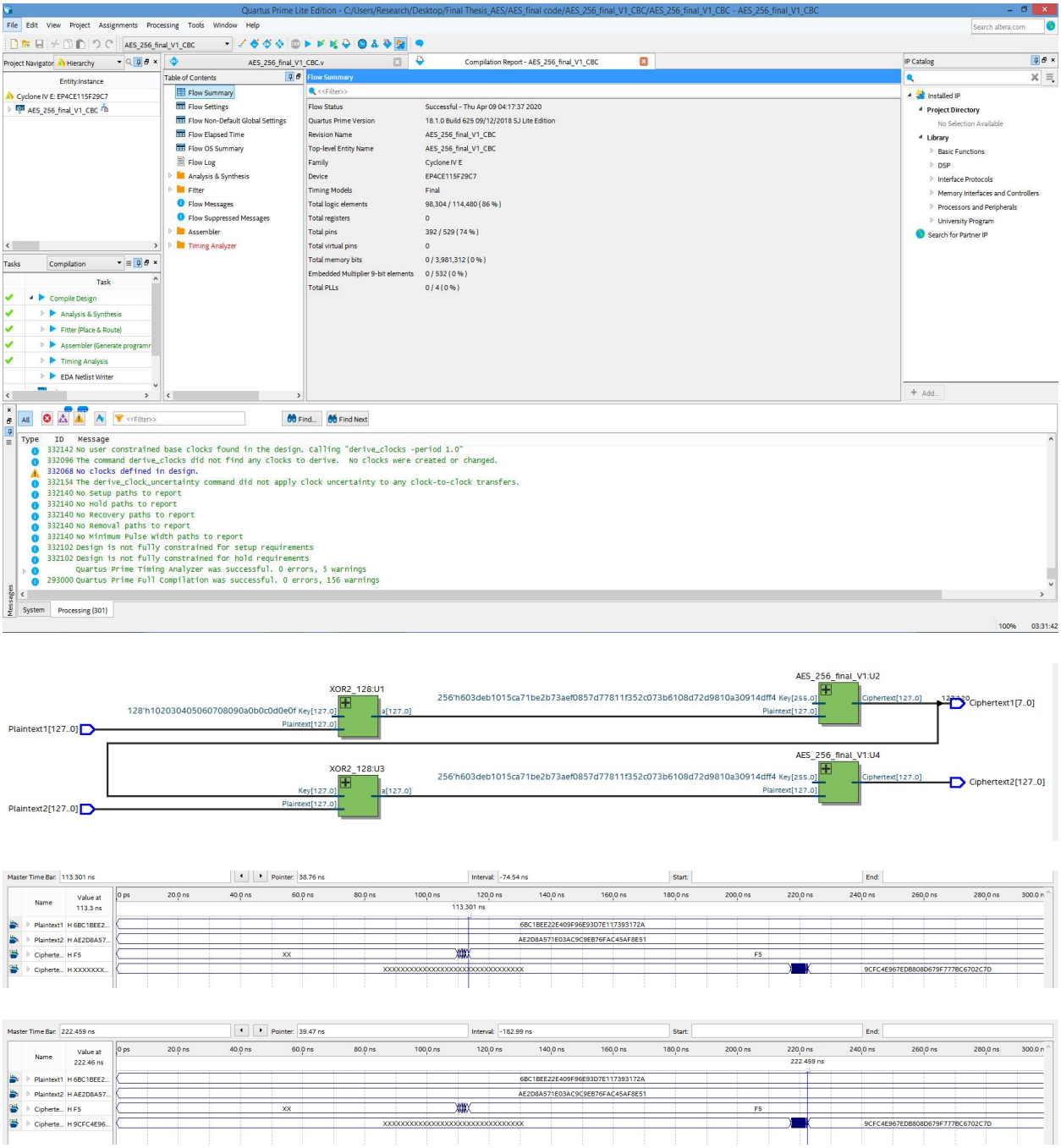

Figure 120: Timing Simulation For AES-256 in CBC Mode

5.3.4 Timing Simulation of AES-512

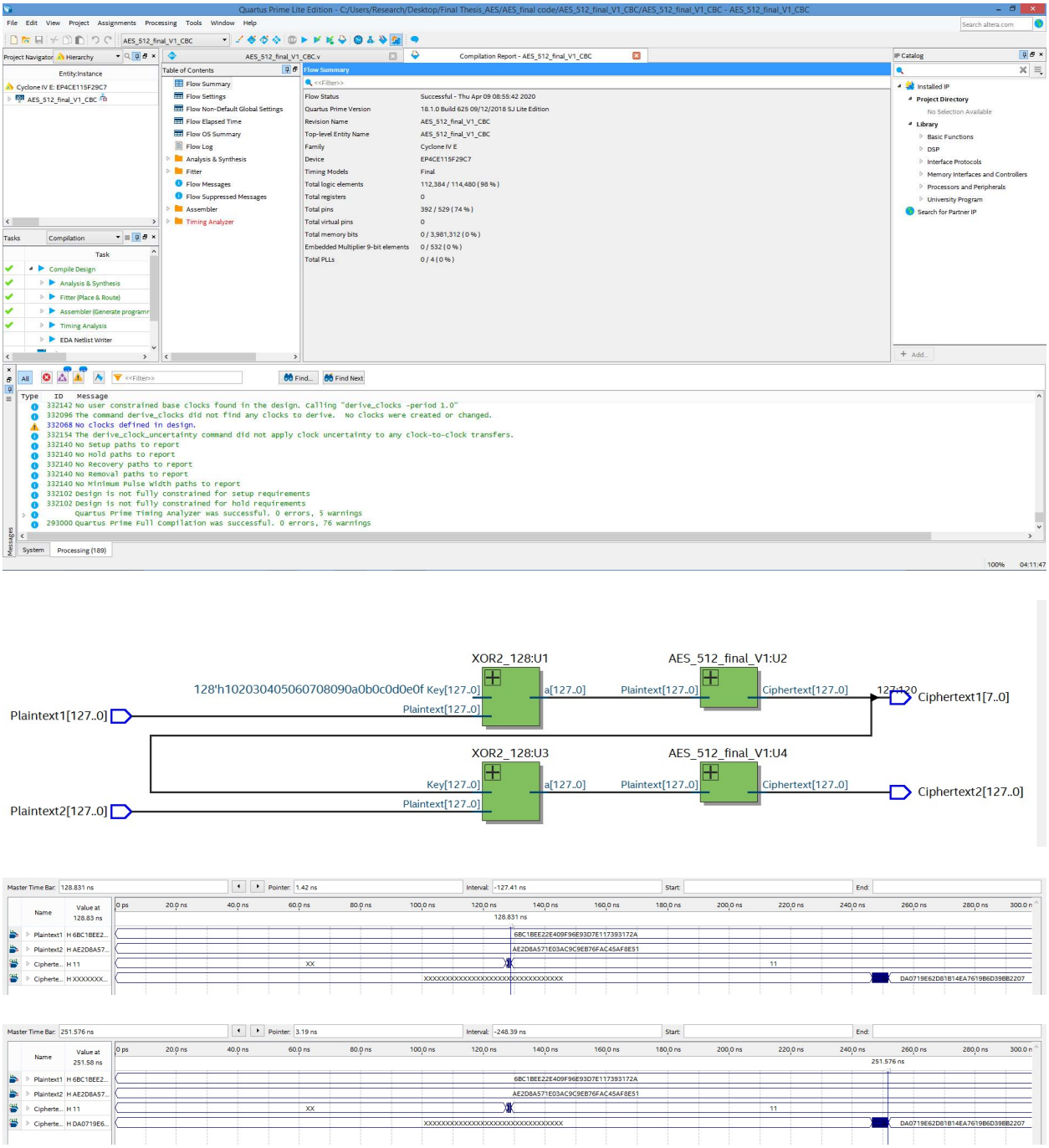

Figure 121: Timing Simulation For AES-512 in CBC Mode

5.4 Comparing of Timing Simulation of AES In ECB and CBC Mode

Table 16: Delay_AES_ECB_KEY

	AES-128	AES-192	AES-256	AES-512
Round1	24.3	25.0	22.2	15.2
Round2	32.1	35.6	33.0	27.8
Round3	42.2	43.3	43.3	36.0
Round4	54.9	55.6	54.8	45.2
Round5	61.4	64.0	70.2	54.2
Round6	73.2	71.9	81.2	61.8
Round7	85.2	81.2	91.9	70.9
Round8	99.0	98.7	102.3	83.0
Round9	102.8	106.3	107.3	92.3
Round10	116.4	108.7	124.2	95.9
Round11		119.9	126.2	108.3
Round12		126.5	135.3	114.4
Round13			135.2	116.8
Round14			148.3	138.3
Round15				138.3
Round16				143.4

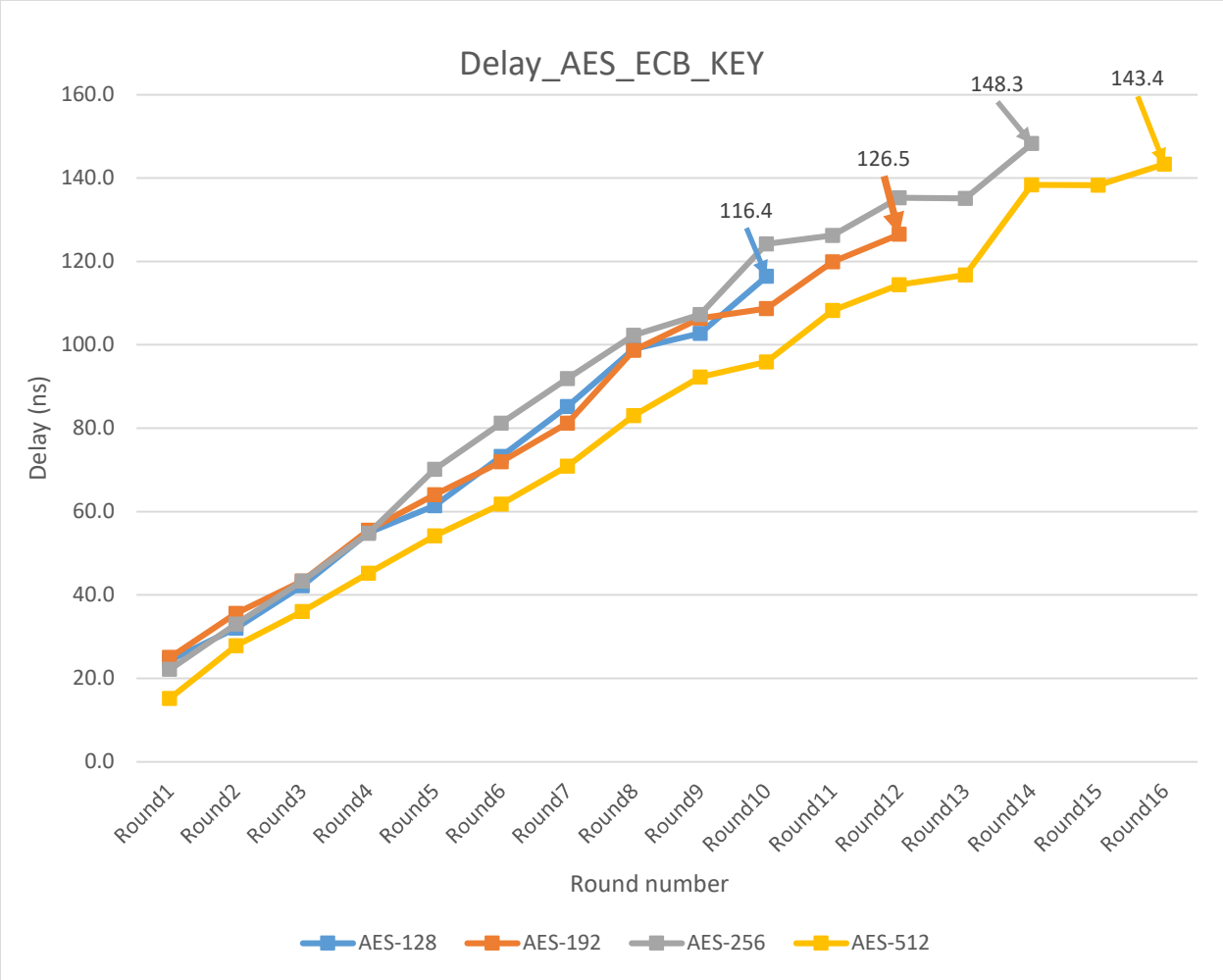

Figure 122: Delay_AES_ECB_KEY

From the Figure 122, we can observe that the delay is increasing as the key size becomes larger. However, the increase of delay for AES-256 and AES-512 are not absolutely related to AES-128 and ASE-256 because the basic method of key expansion is different. We can also observe that from the total logic elements for each type of AES shown in Table 17 or Figure 123.

Table 17: Total Logic Elements_AES_ECB_KEY

	AES-128	AES-192	AES-256	AES-512
Round1	4632	4568	3672	3520
Round2	9114	8202	8138	7040
Round3	13604	12668	12610	10560
Round4	18084	17172	17096	14080
Round5	22602	20813	21582	17600
Round6	27095	25307	26044	21120
Round7	31633	29400	30514	24640
Round8	36113	33493	35007	28160
Round9	40608	37990	39476	31680
Round10	45085	42524	44016	35200
Round11		46189	48548	38720
Round12		50547	53076	42240
Round13			57595	45760
Round14			61953	49280
Round15				52800
Round16				56128

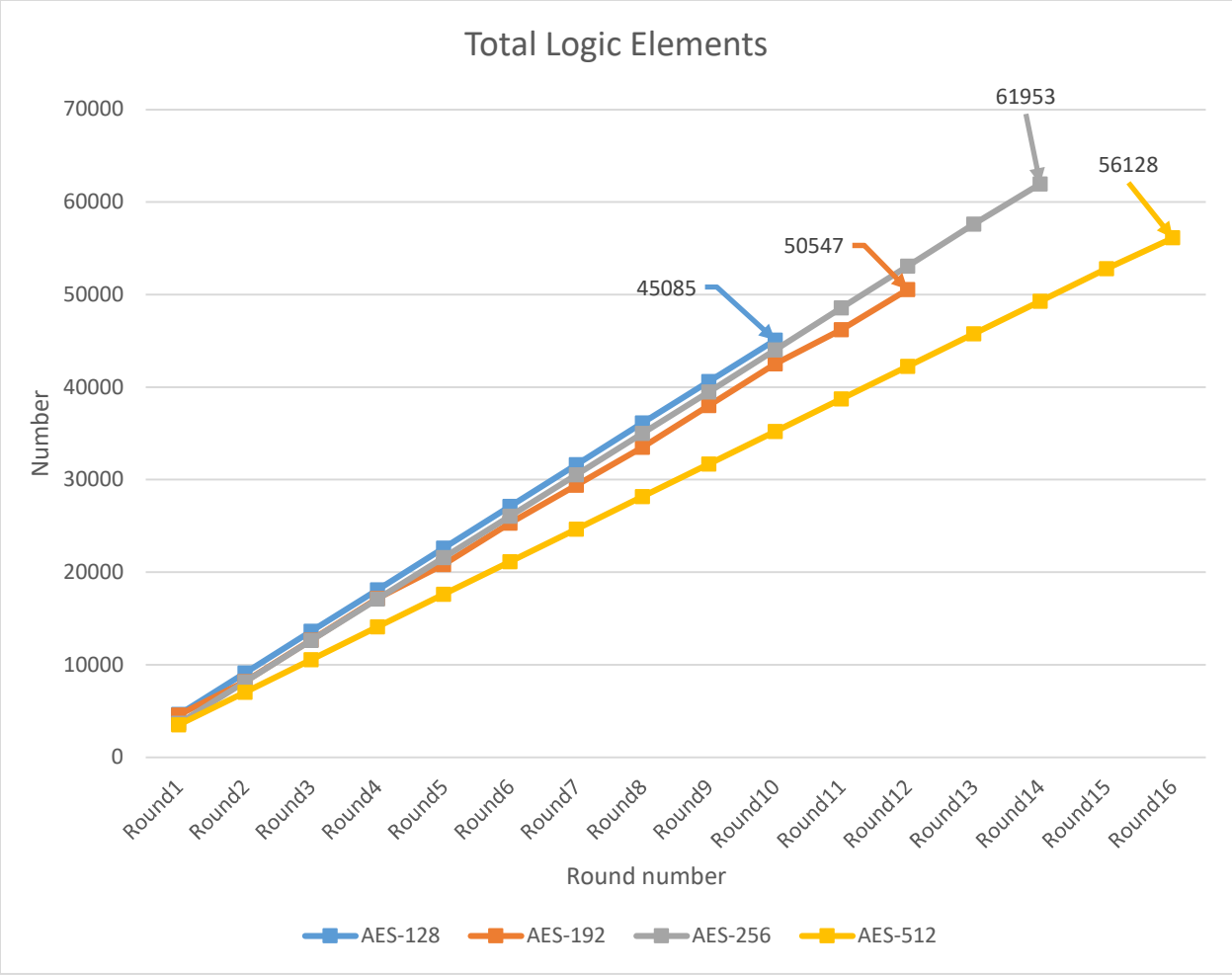

Figure 123: Total Logic Elements_AES_ECB_KEY

We can observe and assume that ASE-128 and AES-192 are using the same method of key expansion, and then when it comes to AES-256, it is using another Key Expansion routine which is slightly different than for AES-128 and AES-192. As we designed the AES-512 key based on the Key Expansion routine for AES-256, we can observe the pattern form the performance of delay and total logic elements.

Table 18: Processing time_AES_ECB_KEY

	AES-128	AES-192	AES-256	AES-512
Round1	101	237	111	96
Round2	233	482	219	201
Round3	763	1554	775	615
Round4	1483	2933	1395	1172
Round5	2265	4435	2041	1464
Round6	3033	6084	2789	2060
Round7	3930	8011	3598	2809
Round8	4905	9775	4431	2975
Round9	5924	11786	5281	3484
Round10	6886	14445	6089	3980
Round11		16688	7477	4563
Round12		19847	8337	5041
Round13			9175	5550
Round14			10116	6130
Round15				6515
Round16				6961

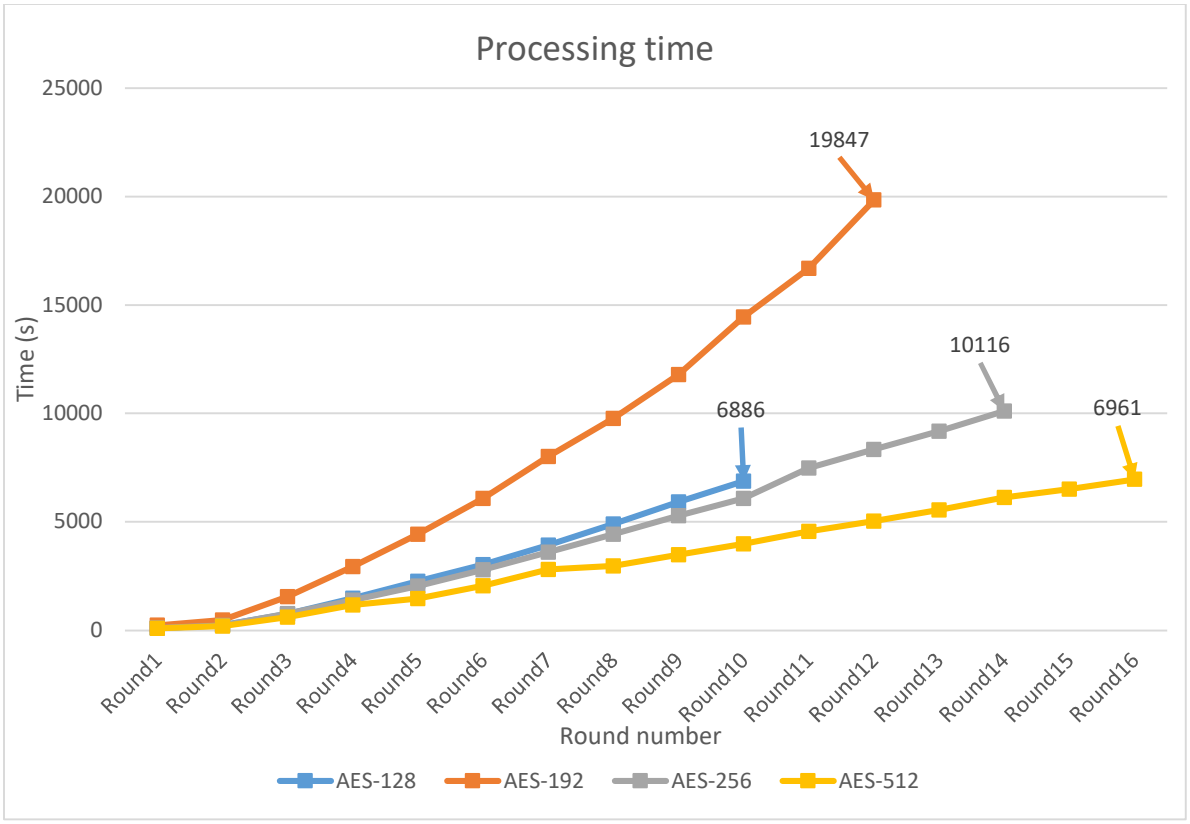

Figure 124: Processing time_AES_ECB_KEY

Based on the data we obtained from Table 16-18, we can observe the development of AES algorithm. To make AES-128 securer, AES-192 extended the key size so the delay increased as well as the processing time as shown in Table 18 and Figure 124 and the total logic elements which means the cost increased. Next, to make AES-256 securer than AES-128 and AES-192, not only the key size increased but also the key expansion routine changed. The additional benefit of that is reducing the processing time. Finally, when it comes to AES-512, AES-512 follows the same concept with AES-256 and uses the slightly different Key Expansion

than for AES-128 and AES-192. As a result, the processing time is even much less than all others.

To test the CBC mode in this device, we used 2 different block plaintexts (each block plaintext is 128 bits). Since there are limited logic elements and total pins, we had to put the key into the program code. The average delay of the first Ciphertext for AES-128 in CBC mode is only 88.52 ns which is 23.7 % less than the delay of the first Ciphertext for AES-128 in ECB mode. It is because of the design of the code that is we had to put the key into the program code. However, from Table 19, we can also observe the delay of Ciphertext 2 is approximately twice of Ciphertext 1 due to the block plaintexts are different, and so the delay of each Ciphertext will be slightly different.

Table 19: Performance_AES_CBC_KEY

	Ciphertext 1	Ciphertext 2	Logic elements	Processing time
AES_128	86.22	169.93	70144	2:16:47
AES_128_1	86.22	169.93	70144	2:23:54
AES_128_2	92.43	169.44	70144	2:17:29
AES_128_3	89.22	171.8	70144	2:18:12
AES_192	104.28	199.78	84224	2:55:15
AES_256	113.3	222.46	98304	3:31:42
AES_512	128.83	251.58	112384	4:11:47

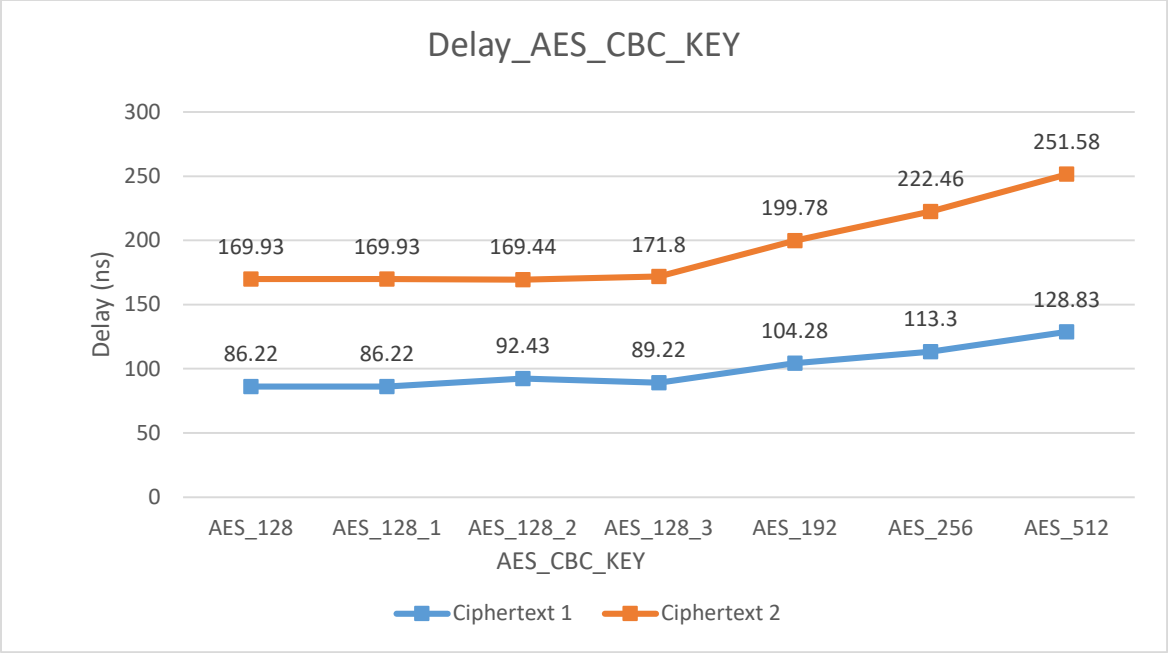

Figure 125: Delay_AES_CBC_KEY

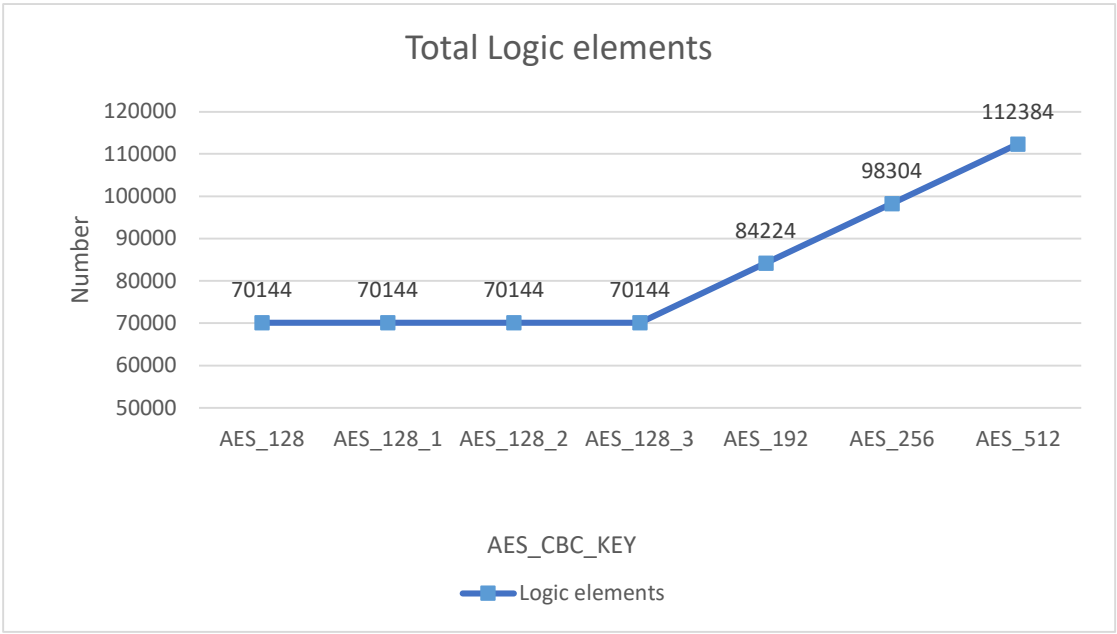

Figure 126: Total Logic Elements_AES_CBC_KEY

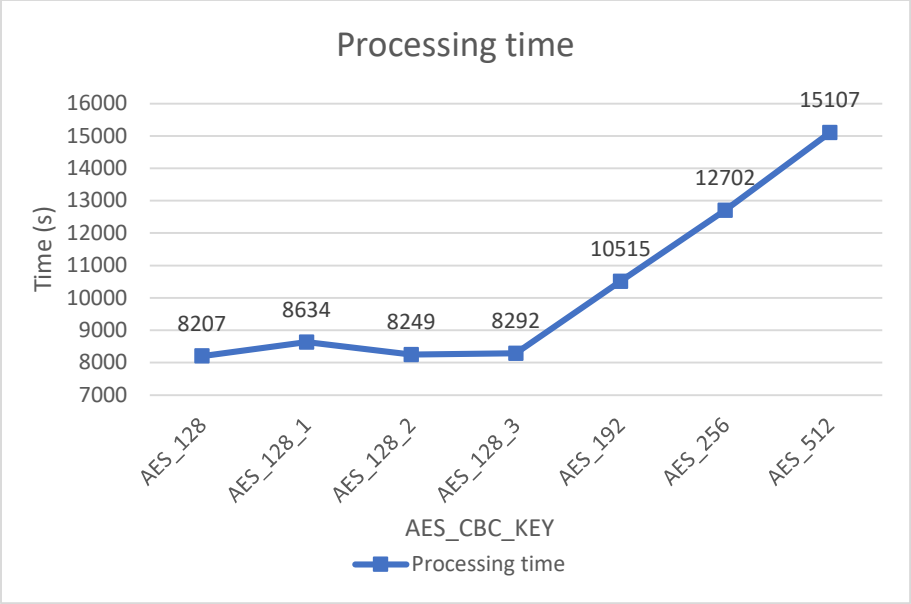

Figure 127: Total Processing time_AES_CBC_KEY

From Figure 125 -127, we can observe that the results show a linear trend where an increase in key size results in an increase in delay, number of logic element and processing time for the plaintext encryption portion.

CHAPTER VI
CONCLUSION AND FUTURE WORK

6.1 Conclusion

Cryptography plays a key role in information security by providing confidentiality when transmitting data. In network security for digital communication systems, where large data blocks go through a cryptographic algorithm with a cipher key that increases the security and complexity of the output ciphertext. With the emergence and rapid growth of cloud computing in the past several years, multiple security algorithms have been developed and utilized in the data encryption process, and the current one, designated by the U.S. National Institute of Standards and Technology (NIST), the Advanced Encryption Standard (AES).

Encryption algorithms are always improving on ciphertext complexity, required hardware storage allocation, and execution time. In our previous works, we presented an FPGA implementation of the AES Mix Columns module and S-box module where parallelism and pipelining in the computation was utilized. From the AES algorithm, we can find that the main difference between AES-128, AES-192 and AES-256 except for the different rounds is key expansion. The Key Expansion routine for AES-256 is slightly different than for AES-128 and AES-192. As the length of the Key changes, the key expansion method and numbers change as well. In the current standard, the length of the Key can only go up to 256 bits. Although, several AES-512 key expansion implementation related work is available in literature using 22 rounds.

According to the basic concept of the standard designed by National Institute of Standards and Technology (NIST) and many studies, we decided to choose 16 as the number of rounds.

This research focused on extending the FPGA implementation to the entire algorithm and will presents high speed, fully pipelined FPGA implementation of AES Encryption in different operation modes.; furthermore, we also evaluated the overall performance for the different variants of AES such as AES-128, AES-192, AES-256, and AES-512. The presented work attained speed up (i.e. high throughput No. of block processed per second) at the same time, silicon area optimization. Comparisons are conducted on both a theoretical basis and through timing simulations on the Intel Quartus II software to reveal the implication of increased complexity on the hardware performance of AES.

The results show how the encryption time varies with key size selection. It was not only found out from the AES algorithm but also from the actual simulation data, that when the key size increases, the encryption time for the four function modules of AES also increases. However, this increase is not proportional to the increase in key expansion processing time. Despite having more rounds and increased complexity, AES-512 was found to be slightly computationally faster in processing time for key expansion than AES-256. When seeking to secure data confidentiality at the highest level, AES-512 offers this with users only experiencing an 18% increase in nanosecond processing time compared to the baseline variant AES-128. As the information industry continues to introduce a demand for more secure transmissions, users wanting to implement stronger variants of AES can default to AES-512 as opposed to AES-256

with no detrimental results in the processing speed. In addition, the increase in key length to 512 makes the AES algorithm highly resistant to new attacks and has acceptable data encryption speed.

6.2 Future work

In the future, we can focus on the other modes which we were not able to implement, and we can focus on how to obtain optimized area and speed hardware implementations of AES based on the sub-pipelined architecture on the other FPGA. The main thing is to extend and to make an AES encrypted block which can be used as in advanced microprocessors or microcontrollers.

REFERENCES

[1] "Advanced Encryption Standard (AES)." Federal Information Processing Standards, US National Institute of Standards and Technology, 26 November 2001, https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[2] William Stallings. "Cryptography and Network Security 5th ed." ISBN-10: 0136097049

[3] Kit Choy Xintong. "Understanding AES Mix-Columns Transformation Calculation." http://www.angelfire.com/biz7/atleast/mix_columns.pdf

[4] Altera (2017) Quartus Prime Design Software, http://fpgasoftware.intel.com/17.0/?edition=lite

[5] John, L. (2016) "Altera Parts History. " University of California, Berkeley, http://www-inst.eecs.berkeley.edu/~cs294-59/fa10/resources/Altera-history/Alterahistory

[6] Altera (2017) Cyclone IV DE2-115 User Manual, Version 2.3, http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=502&PartNo=4

[7] Altera DE2 Board, Terasic, N.P. (2016) , https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=139&No=502

[8] Del Rosal, Edni, and Sanjeev Kumar. "A Fast FPGA Implementation for Triple DES Encryption Scheme." Circuits and Systems 8.09 (2017): 237

[9] M.R.M. Rizk, M. Morsy. "Optimized Area and Optimized Speed Hardware Implementations of AES on FPGA." 22 January 2008, https://ieeexplore.ieee.org/document/4437462

[10] Nalini C,Dr. Anandmohan P.V, Poomaiah D.V, and V.D.kulkami, "Compact Designs of SubBytes and MixColumn for AES", https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4809193

[11] Leventis, Paul, et al. "Cyclone/spl trade: a low-cost, high-performance FPGA." Proceedings of the IEEE 2003 Custom Integrated Circuits Conference, 2003.. IEEE, 2003. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1249357

[12] L. AIi, I. Aris, F. S. Hossain and N. Roy, "Design of an ultra high speed AES processo for next generation IT security," Computers and Eleetrieal Engineering, Vol.37 (6), pp.1160-1170, Nov. 2011.

[13] Raneesha K, Rema Vellody and R Nandakumar, "HARDWARE EFFICIENCY COMPARISON OF AES IMPLEMENTATIONS", 2012 International Conference on Communication Systems and Network Technologies

[14] Chih-Chung Lu and Shau-Yin Tseng, “Integrated Design of AES (Advanced Encryption Standard) Encrypter and Decrypter,” Proc. IEEE Int. Conf. on Application-Specific Systems, Architectures, and Processors, (ASAP’02), pp. 277-285, 2002

[15] Guang-liang Guo, Quan Qian, and Rui Zhang, "Different Implementations of AES Cryptographic Algorithm", 2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), and 2015 IEEE 12th International Conf on Embedded Software and Systems (ICESS)

[16] C. L. Duta, G. Michiu, S. Stoica and L. Gheorghe. “Accelerating Encryption Algorithms Using Parallelism”, 2013 19th International Conference on Control Systems and Computer Science (CSCS). IEEE Press, Bucharest, May 2013, pp.549-554, DOI: 10.1109/CSCS.2013.92.

[17] J. Fang. “Mix Column round transformation Optimization and Improvement in the AES Algoritm”, Control & Automation, Vol.25, No.21, pp. 49-50.

[18] Tanzilur Rahman, Shengyi Pan, Qi Zhang (2010) Design of a High Throughput 128-bt AES(Rijndael Block Cipher),Proceedings of the International MultiConference of Enigneers and Computer Scientists 2010 Vol II, March 17-19,2010.Hong Kong.

[19] Hossain FS, Ali L, Abedin Syed MA. (2011), "Design of a very low power and High Throughput AES Processor", IEEE conference on Computer and Information Technology (ICCIT), doi:10.1109/ICCITechn. 2011.6164810 pp. 339–343.

[20] Dilna.v, C. Babu “Area Optimized and high throughput AES algorithm based on permutation data scramble approach”, in International Conference on Electrical, Electronics and Optimization techniques (ICEEOT)- 2016.

[21] Rijmen, Vincent. "Efficient Implementation of the Rijndael S-box." Katholieke Universiteit Leuven, Dept. ESAT. Belgium (2000). http://luca-giuzzi.unibs.it/corsi/Support/papers-cryptography/rijndael-sbox.pdf

[22] Satoh, Akashi, et al. "A compact Rijndael hardware architecture with S-box optimization." International Conference on the Theory and Application of Cryptology and Information Security. Springer, Berlin, Heidelberg, 2001. https://link.springer.com/content/pdf/10.1007%2F3-540-45682-1_15.pdf.

[23] N. Mentens, L. Batinan, B. Preneeland, and I. Verbauwhede, “A systematic evaluation of compact hardware implementations for the Rijndael S-box,” in Proc. Topics in Cryptology - CT-RSA 2005, vol. 3376/2005. San Francisco, CA: Springer Berlin / Heidelberg, Feb. 2005, pp. 323–333.

[24] Canright, David. "A very compact S-box for AES." International Workshop on Cryptographic Hardware and Embedded Systems. Springer, Berlin, Heidelberg, 2005. https://www.iacr.org/archive/ches2005/032.pdf

[25] Wong Ming Ming and Dennis Wong Mou Ling, “A new lightweight and high performance AES S-box using modular design,” in IEEE International Conference on circuits and systems, pp. 65-70, 2013

[26] V. Rijmen, “Effcient implementation of the Rijndael S-box,” in World Wide Web - 171–184

[27] Hossain FS and Ali M.L (2015), "A Novel Byte-Substitution Architecture for the AES Cryptosystem",PLoS ONE 10(10): e0138457. doi:10.1371/journal. pone.0138457

[28] Aaron Barrera, Chu-Wen Cheng, Dr. Sanjeev Kumar. "Hardware Implementation of Improved Mix Column Computation of Cryptographic AES." June 2019, 2nd International Conference on Data Intelligence and Security.

[29] Carlos Cid, Sean Murphy and Matthew Robshaw, “Computational and AlgebraicAspects of the Advanced Encryption Standard”, Information Security Group, 2008

[30] Toa Bi Irie Guy-Cedric, Suchithra. R. "A Comparative Study on AES 128 BIT AND AES 256 BIT", 31 August 2018, http://www.isroset.org/pdf_paper_view.php?paper_id=782&5-IJSRCSE-01186.pdf.

[31] Diaa, S., E, Hatem M. A. K., & Mohiy M. H. "Evaluating the Performance of Symmetric Encryption Algorithms." May 2010, International Journal of Network Security. http://ijns.jalaxy.com.tw/contents/ijns-v10-n3/ijns-2010-v10-n3-p213-219.pdf.

[32] Sridevi Sathya Priya, S., Karthigaikumar, P. "FPGA implementation of High speed compact S-Box", https://acadpubl.eu/hub/2018-119-16/1/165.pdf.

[33] Vatchara Saicheur, Krerk Piromsopa. "An implementation of AES-128 and AES-512 on Apple mobile processor." 2017 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), https://ieeexplore.ieee.org/document/8096255.

[34] Abidalrahman Moh'd, Yaser Jararweh, Lo'ai Tawalbeh. "AES-512: 512-bit Advanced Encryption Standard algorithm design and evaluation." 2011 7th International Conference on Information Assurance and Security (IAS), https://ieeexplore.ieee.org/document/6122835.

[35] Awadhesh Kumar and R.R. Tewari, "Expansion of Round Key Generations in Advanced Encryption Standard for Secure Communication", International Journal of Computational Intelligence Research ISSN 0973-1873 Volume 13, Number 7 (2017), pp. 1679-1698

[36] Barker, E. & Roginsky, A. (2012). Recommendation for Cryptographic Key Generation (p. 26). USA: NIST Special Publication 800-133

[37] J. Takahashi and T. Fukunaga, “Differential Fault Analysis on AES with 192 and 256-bit Keys,” Cryptology ePrint Archive, Report 2010/023, 2010, http://eprint.iarc.org/

[38] P. Rogaway. “Evaluation of Some Blockcipher Modes of Operation”. Cryptography Research and Evaluation Committees (CRYPTREC), 2011, available at: http://www.cryptrec.go.jp/estimation/techrep_id2012_2.pdf.

[39] Christophe Clavier, Julien Francq, Antoine Wurcker, "Study of a Parity Check Based Fault-Detection Countermeasure for the AES Key Schedule", [Research Report] 2015/877, IACR Cryptology ePrint Archive. 2015. hal-02486939.

[40] Kazi Huma , Shete Chaitali, Vidhate Amruta, Deshmukh Sneha, "Implementation of AES using 512 bit key for secure communication", International Journal of Science, Engineering and Technology Research (IJSETR), Volume 4, Issue 5, May 2015

[41] "Study of avalanche effect in AES using binary codes." Advanced Communication Control and Computing Technologies (ICACCCT), 2012 IEEE International Conference on. IEEE, 2012

[42] Swati Paliwal, Ravindra Gupta, “A Review of Some Popular Encryption Techniques”, IJARCSSE Volume 3,2013.

[43] N. Dworkin. “Recommendation for Block Cipher Modes of Operation, Methods and Techniques”. NIST Special Publication 800-38A Edition 2001, available at: http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

[44] Naseema Bhanu, N.V. Chaitanya, "Aes Modes of Operation"', International Journal of Innovative Technology and Exploring Engineering (IJITEE) ISSN: 2278-3075, Volume-8 Issue-9, July 2019.

[45] Aaron Barrera, Chu-Wen Cheng, Dr. Sanjeev Kumar. " Improving performance of the Rijndael Substitution Box for Cryptographic AES", August 2020, 3rd International Conference on Data Intelligence and Security.

BIOGRAPHICAL SKETCH

Chu-Wen Cheng was born on September 23, 1975. He has completed his Bachelor of Science in Electrical Engineering from University of Texas Rio Grande Valley, TEXAS, USA in May 2018. He has completed his Master of Science in Electrical Engineering from University of Texas Rio Grande Valley, Texas, USA in August 2020. At UTRGV, he was listed on Dean's list and President's list 3 semesters in a row. He was nominated for Student Employee of the Year 2019 and selected as the outstanding graduate student in 2020. He has received 5 different scholarships and worked in several positions, such as Research Assistant in Network Research Lab and Teaching Assistant from May 2018 to August 2020.

Address:

1008 W Champion St. Apt # B

Edinburg, Texas, USA, 78539.

His Publications:

1) Aaron Barrera, Chu-Wen Cheng, Sanjeev Kumar (August 2020), “Improving performance of the Rijndael Substitution Box for Cryptographic AES”, IEEE-ICDIS
2) Aaron Barrera, Chu-Wen Cheng, Sanjeev Kumar (June 2019), “Improved Mix Column Computation of Cryptographic AES”, IEEE-ICDIS
3) Nazmul Islam, Rakesh Guduru, Chu-Wen Cheng (2018), “Improving the Micropump Velocity for Orthogonal Electrode Pattern”, ASME-IMECE