Using MachXO3D ESB to Implement ECC-based Authentication

Reference Design

FPGA-RD-02065-1.0

September 2019

Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS and with all faults, and all risk associated with such information is entirely with Buyer. Buyer shall not rely on any data and performance specifications or parameters provided herein. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. No Lattice products should be used in conjunction with mission- or safety-critical or any other application in which the failure of Lattice's product could create a situation where personal injury, death, severe property or environmental damage may occur. The information provided in this document is proprietary to Lattice Semiconductor, and Lattice reserves the right to make any changes to the information in this document or to any products at any time without notice.

Contents

Figures

Tables

Acronyms in This Document

A list of acronyms used in this document.

Acronym	Definition
AES	Advanced Encryption Standard
DSA	Digital Signature Algorithm
ECC	Elliptic Curve Cipher
ECDSA	Elliptic Curve Digital Signature Algorithm
ESB	Embedded Security Block
EBR	Embedded Block RAM
HSP	High Speed Port
NIST	National Institute of Standards and Technology
OSC	Oscillator
SHA	Secure Hash Algorithm
TRNG	True Random Number Generator

1. Introduction

In cryptography, the Elliptic Curve Digital Signature Algorithm (ECDSA) offers a variant of the Digital Signature Algorithm (DSA) that uses Elliptic Curve Cryptography (ECC).

In ECC, the private key can be used to create a digital signature for any piece of data using a DSA. This typically involves taking a cryptographic hash of the data and operating on it mathematically using a private key. Anyone with the paired public key can check if this signature is created using the private key.

This ECC-based authentication demonstration shows the general flow as follows:

1. Generate a firmware image and encrypt/decrypt key using True Random Number Generator (TRNG) engine;
2. Generate the Public-Private ECC key pair using ECC Key Generation engine;
3. Encrypt the image using Advanced Encryption Standard (AES) engine;
4. Generate the digest of the encrypted image with SHA256 engine;
5. Generate the digital signature with ECDSA generator engine;
6. Verify the digital signature with ECDSA verification engine;
7. Decrypt the encrypted image using AES to recover the original image.

2. Demonstration Overview

This demonstration shows a flow of ECC-based authentication using the MachXO3D ESB engine. The flow includes TRNG, AES encryption/decryption, SHA256, ECC key generating, ECDSA generating, and ECDSA verification.

2.1. Block Diagram

Figure 2.1 shows the block diagram of the ECC-based authentication design flow.

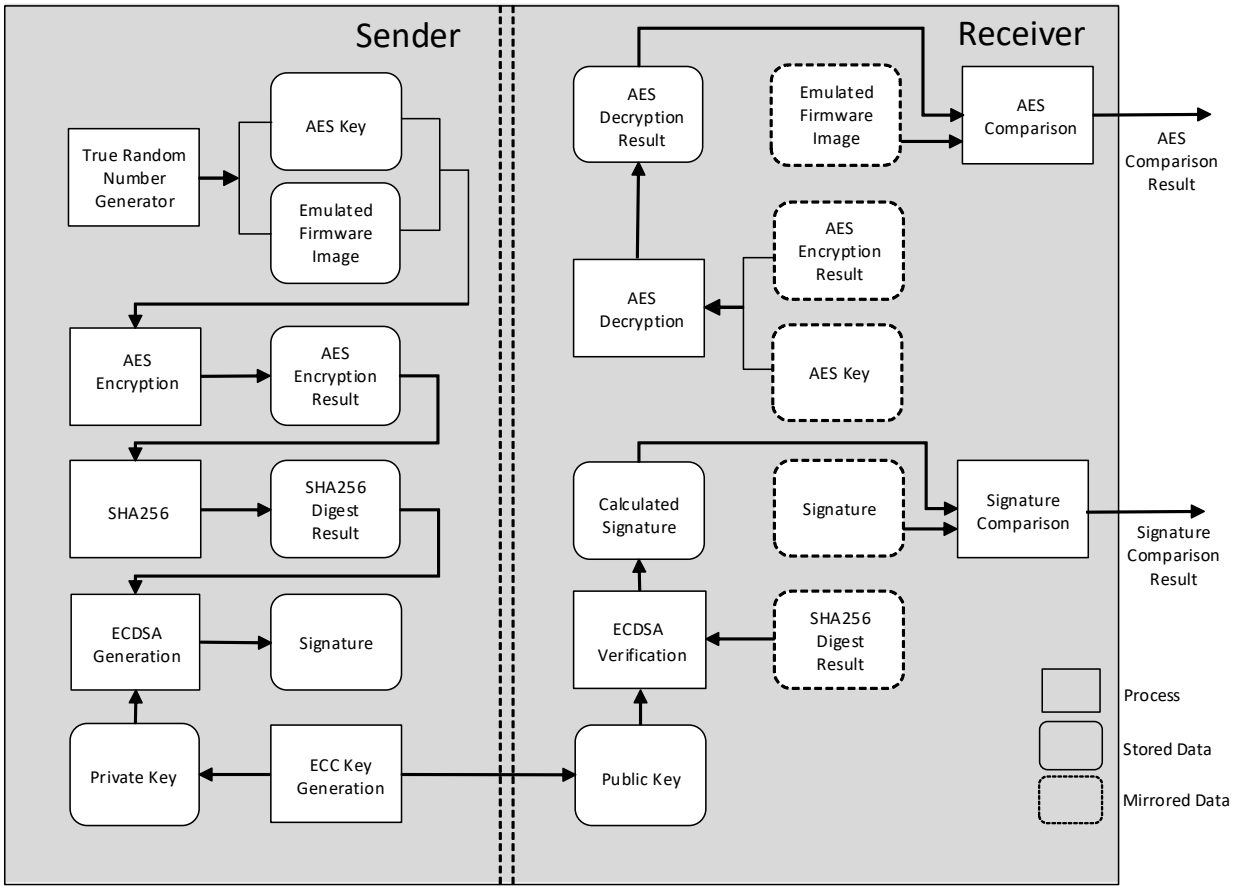

Figure 2.1. Top-Level Block Diagram

2.2. Overview

The ECC-based authentication demonstration consists of five major blocks:

- TRNG
  TRNG is used in most of the security functions to generate a random number. In this demonstration, we use TRNG to generate 1024-byte data as plain text, which is used as the firmware image to be signed and verified. We also use TRNG to generate the AES key.
- AES encryption/decryption
  AES is an encryption standard based on symmetric key algorithm, using the same key for encryption and decryption, issued by National Institute of Standards and Technology (NIST) in 2001. The firmware image generated by TRNG is encrypted by AES.

- SHA256
  SHA256 is one of the Secure Hash Signature Standards (SHS) provided by NIST and published in Federal Information Processing Standard Publications (FIPS PUBS). The purpose of this standard is to provide a condensed representation of electronic data or message. The 1024-byte AES encryption result is condensed to 256-bit message digest through the Embedded Security Block (ESB).
- ECC key generating
  The ESB is capable of generating an ECC-based public and private key pair. This function used in this demonstration is to generate the public and private keys for ECDSA generation and verification.
- ECDSA
  The ECDSA generation engine generates the digital signature for a given input data which are consist of the private key and the message digest. ECDSA verification engine are used to verify the digital signature using the public key.

3. Functional Description

In this ECC-based authentication demonstration, the following define compiler directive is used:

`define SIM_MODE

When in none SIM_MODE, this directive is not defined. All functions are implemented, including TRNG, AES encryption, AES decryption, SHA256, ECC key pair generation, ECDSA generation, and ECDSA verification. You can check the result by running the Lattice Reveal™ software.

When in SIM_MODE, this directive is defined. The TRNG and ECC key pair generation functions are not included to avoid the long simulation time. Instead, we use the predefined data for the keys and the firmware images.

Table 3.1 provides ECC-based authentication reference design directive usage.

Table 3.1. ECC-based Authentication Demonstration Define

Directive Define Name	Defined	Function
SIM_MODE	Yes	Simulation mode
	No	None simulation mode

4. Demonstration Design Description

As shown in Figure 4.1, there is only one input *rst_n* in this reference design. To trigger the start of the design, we need to provide one active low pulse to the *rst_n* input. There is no input clock pin. The on-chip oscillator (OSC) generates the system clock for the design.

The design automatically initiates a state machine to run through the necessary process for the ECC-based authentication. Refer to the Data Flow section for details. When the whole flow is completed, the signal *sig_r_compare_done* indicates the ECDSA verification is finished. The calculated signature is compared with the expected signature. You can check the compared result by signal *sig_r_compare_result*.

- Output HIGH means the signature is verified and confirmed to be genuine.
- Output LOW means the signature does not match the message. The message source should not be trusted.

The signal *aes_compare_done* indicates the AES comparison is completed. The decrypted message is compared with the original message. You can check the result by signal *aes_compare_result*.

- Output HIGH means the decrypted firmware data matches the original message.
- Output LOW means data mismatches.

With the compared results, we can verify the whole data flow functionality.

4.1. Detailed Input/Output of the Demonstration

Figure 4.1 is the I/O diagram of the ECC-based authentication demonstration.

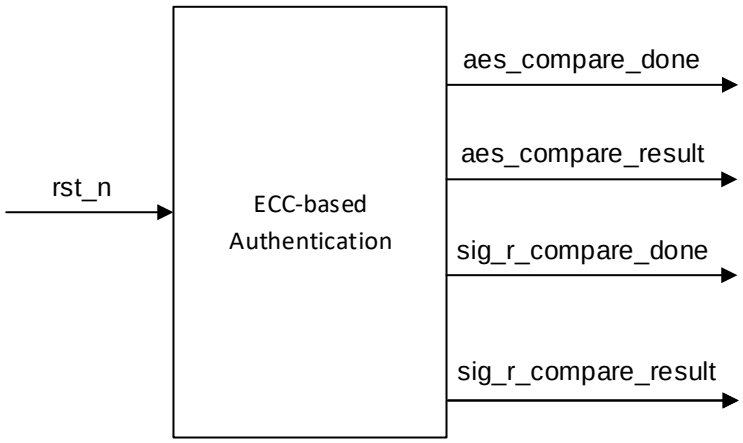

Figure 4.1. I/O Diagram of ECC-based Authentication Demonstration

4.2. Input/Output Ports Description of the Design

Table 4.1 lists input and output ports of the demonstration.

Table 4.1. Pin Descriptions

Signal	Width (bit)	I/O Type	Description
rst_n	1	Input	Global reset
aes_compare_done	1	Output	AES comparison is finished.
aes_compare_result	1	Output	AES comparison result 0: failed 1: passed
sig_r_compare_done	1	Output	Signature comparison is finished.
sig_r_compare_result	1	Output	Signature comparison result 0: failed 1: passed

4.3. Data Flow

Figure 4.2 shows the ECC-based authentication demonstration flow.

The demonstration begins with the firmware image generation. We use different ways to generate the data for simulation and hardware implementation. To save the simulation time, we use the Verilog system $random function to create the data for simulation. With four bytes per call, we repeat 256 times to get the 1024 kB image files. For hardware implementation, we use the TRNG function of the ESB. TRNG generates four bytes per call, and we repeat that 256 times. The raw data is stored in the EBR named *plain_txt*.

We generate the 256-bit AES key for the AES encryption and decryption process. For simulation, we use the fixed value 256'h5555_6666_3333_4444_1111_2222_9999_0000_7777_8888_5555_6666_3333_4444_1111_2222. For hardware implementation, we can use the TRNG to get the 256-bit key. The key is stored in the *aes_key_buff* register.

When the firmware image and AES key are ready, they are sent to the AES engine to perform the AES encryption. We use the HSP mode to provide better throughput performance. After the encryption, the result is stored in two EBRs: one named *aes_result*, which is 32-bit oriented for ease of AES decryption and plain text comparison; the other named aes_result2, which is 8-bit oriented for ease of SHA256 operation, since the SHA256 function is byte-oriented.

We use the ECC Key Generation function to generate the public-private key pair. They are used for the ECDSA process. The public key is stored at register *public_key_x/y*. The private key is stored at *private_key*.

The SHA256 engine takes the input data file from the EBR aes_result2. It generates the corresponding 256-bit digest file for the input data file. The digest is stored in the register *sha_res*.

The digest *sha_res* and private key *private_key* are sent to the ECDSA Generation function to get the signature. The signature is stored in the register *sig_r*.

With all the above steps, we get the encrypted files together with its signature. Next, we check that from the recipient's view.

To save the simulation and hardware run-time, we reuse the digest *sha_res* together with the *public_key_x/y* public key, and send them to the ECDSA verification function. We get the calculated signature. It is stored in register *c_sig_r*. We compare it with *sig_r*. If they match, it means the message is authenticated. Otherwise, we should discard the message as it can be compromised.

Finally, we send the encrypted data to the AES decryption engine, and run decryption. The result is stored in the EBR named *aes_de_result*. We compare it with the data stored in EBR *plain_txt* to verify the decryption is correct or not.

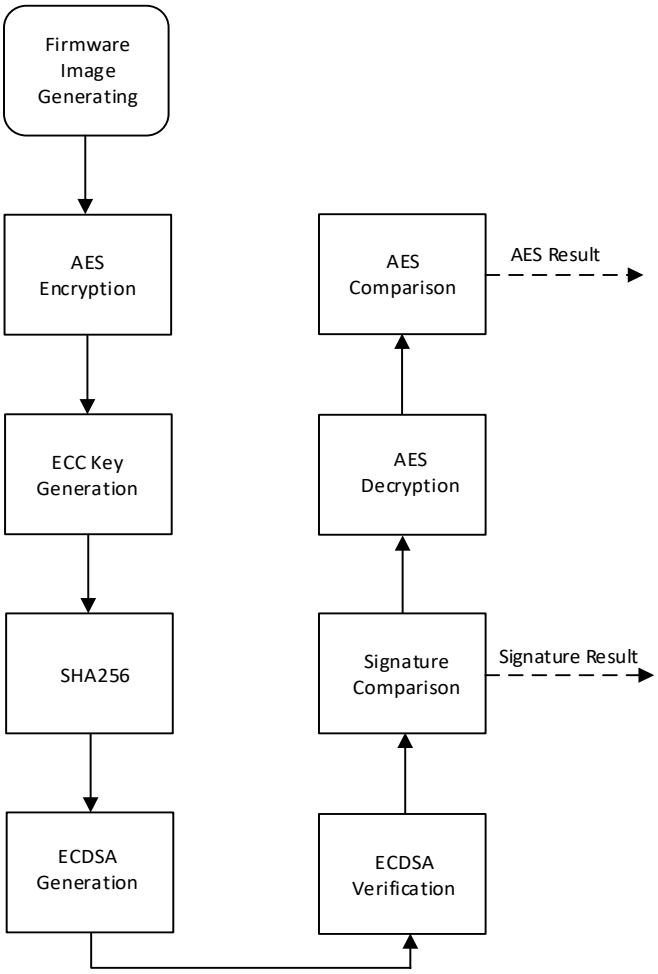

Figure 4.2. ECC-based Authentication Demonstration Flow

4.4. Data Storage of the ECC-based Authentication Demonstration

Table 4.2 lists the data storage of the demonstration.

Table 4.2. Data Storage Descriptions

Storage Type	Data Name	Purpose
Register	aes_key_buff	Store AES key
	public_key_x	Store public key X
	public_key_y	Store public key Y
	private_key	Store private key
	sig_r	Store signature generated by ECDSA
	c_sig_r	Store calculated signature during the ECDSA verification
	sha_res	Store SHA256 result
EBR	plain_txt	Store 1024-byte plain text
	aes_result	Store 1024-byte AES encryption result. Output data width is 32 bits, which is used for the AES decryption and plain text comparison.
	aes_result2	Store 1024-byte AES encryption result. Output data width is 8 bits, which is used as the SHA256 input.
	aes_de_result	Store 1024-byte AES decryption result

5. Verification Using Lattice Reveal

Due to the long simulation time for the ESB related functions, especially for the ECDSA functions, such as the ECC Key generation, ECDSA generation and verification, we use Lattice Reveal tool to check the overall functionality and performance. We provide the simulation testbench and simulation script in the demonstration design files. You can run the simulation to check implementation details.

As shown in Figure 5.1, we set up the trigger signals for the Reveal inserter. We capture the waveform when the AES decryption starts. We can observe all the final output results.

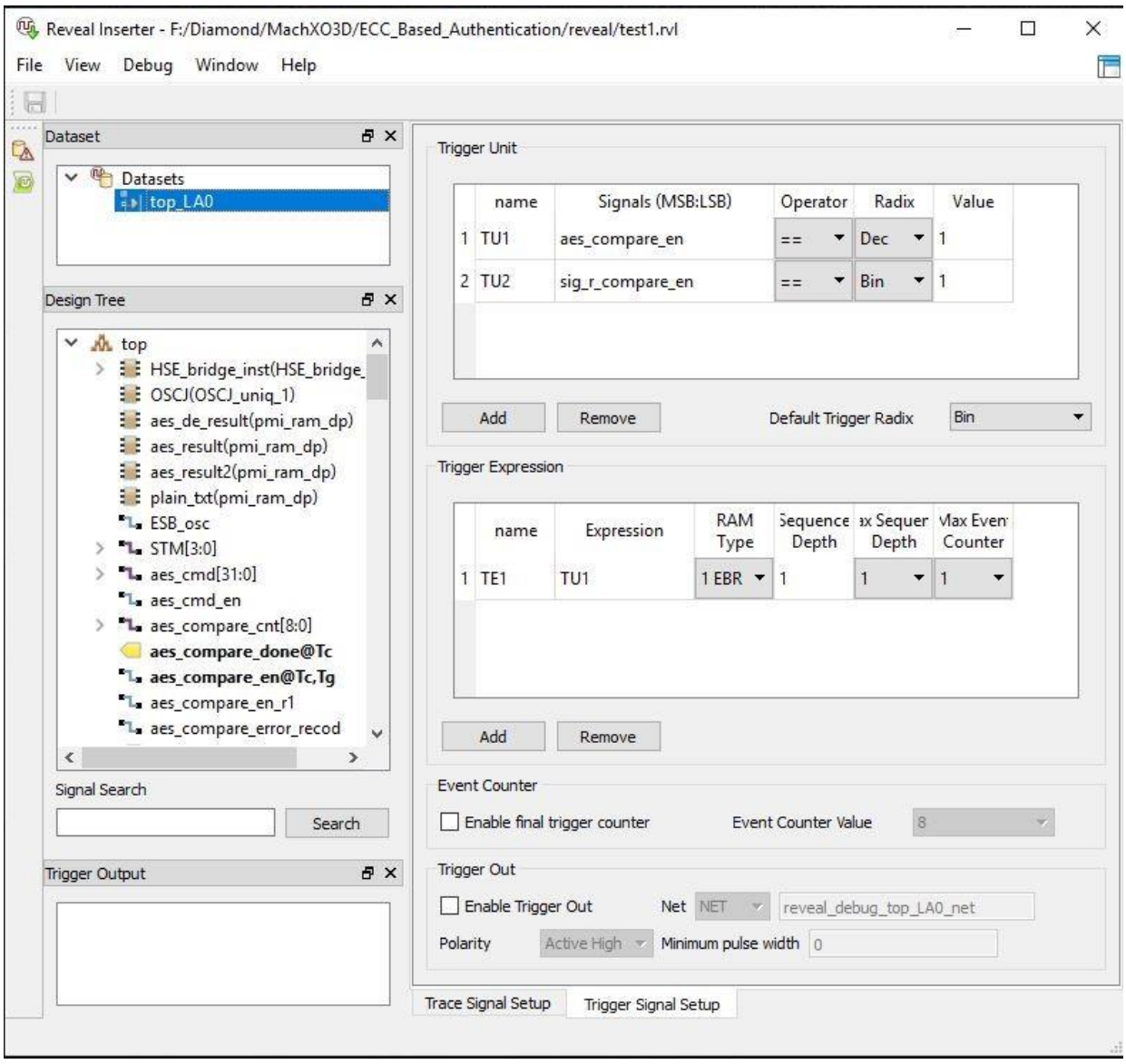

Figure 5.1. Trigger Setup in Reveal

As shown in Figure 5.2, we can observe all these signals in the Lattice Reveal Analyzer windows.

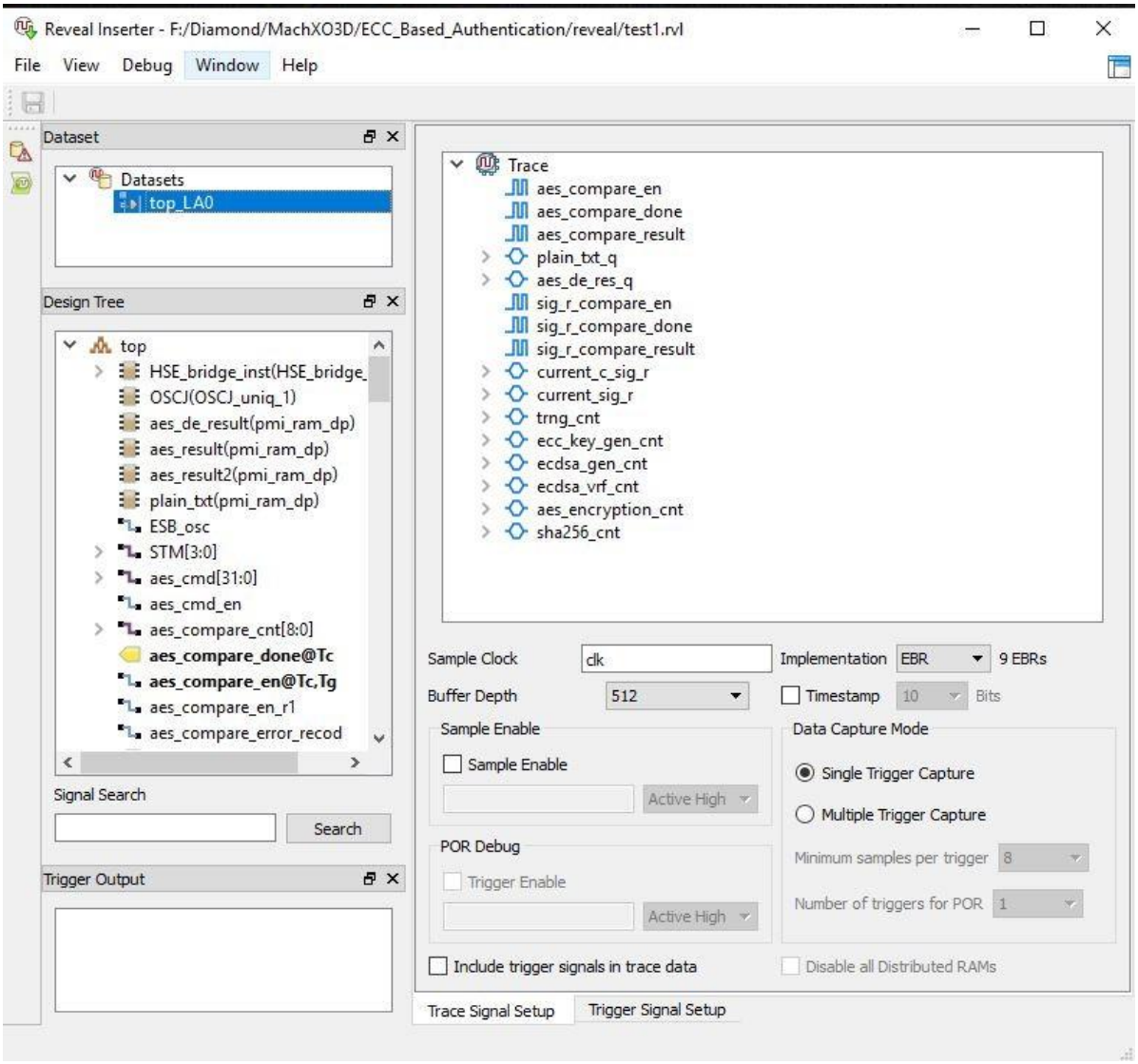

Figure 5.2. Signals in Reveal Analyzer

Figure 5.3 shows the AES comparison result. When *aes_compare_done* goes from 0 to 1, *aes_compare_result* is 1, which indicates the comparison passes with matched data of *plain_txt_q* and *aes_de_res_q*.

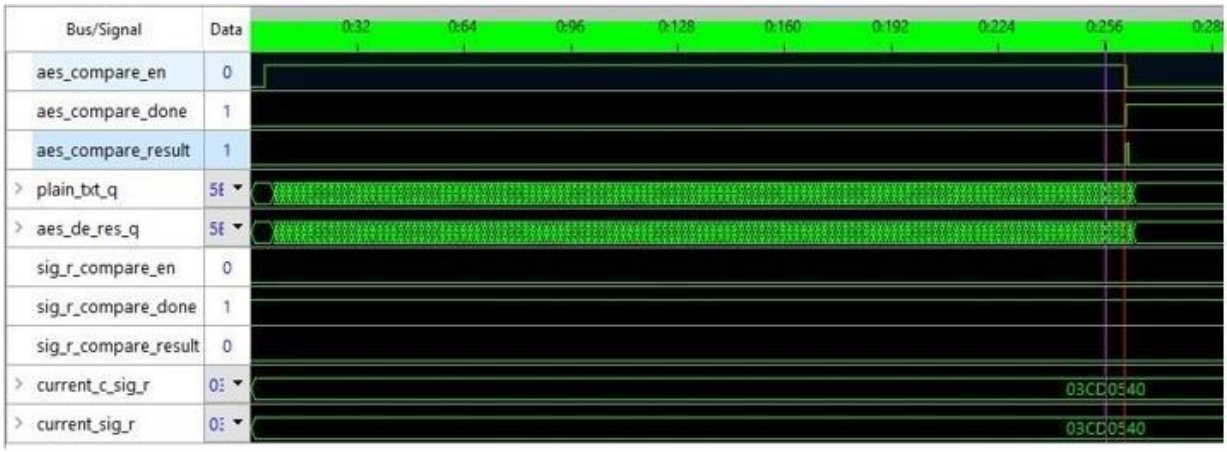

Figure 5.3. AES Comparison Result

Figure 5.4 shows the ECDSA signature verification result. When *sig_r_compare_done* goes from 0 to 1, *sig_r_compare_result* is 1, which indicated the comparison passes with matched data of *current_c_sig_r* and *current_sig_q*. The signal *sig_r_compare_en* is used as the trigger signal for Reveal.

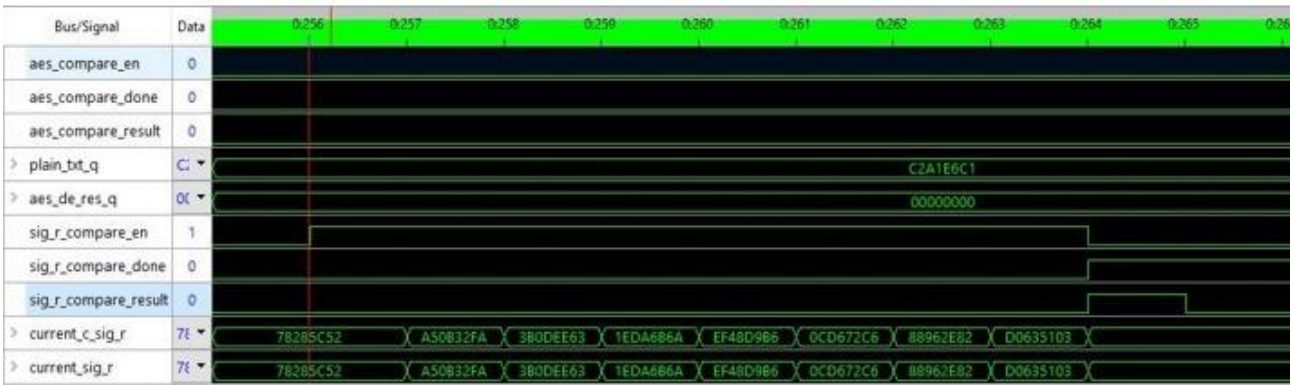

Figure 5.4. ECDSA Signature Verification

Figure 5.5 shows the performance for each of the ESB functions.

For True Random Number Generator, it uses 8597 cycles to generate 256-bit data per call. Similarly, around 3.4M cycles are needed for the ECC Key generation, 3.6M cycles for the ECC generation, 6.9M cycles for the ECC verification. For the AES and SHA function, they use pipelined structure. The total cycles depend on the input data file size. For the demonstration, the file size is 1024 kB. The AES encryption and decryption take 977 cycles. The SHA256 takes 1221 cycles. All the cycles here are based on the system clock with nominal frequency of 66 MHz.

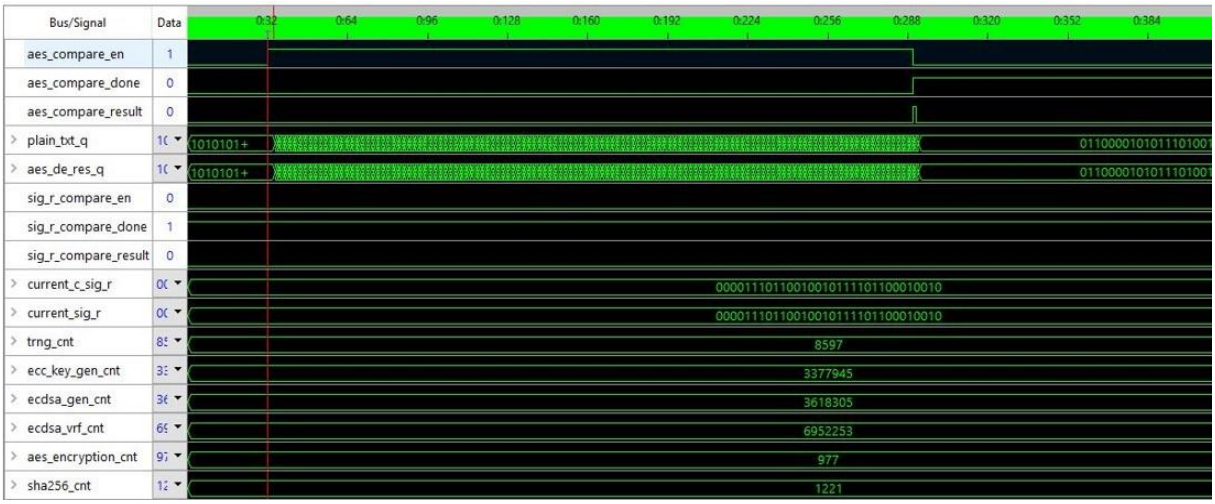

Figure 5.5. ECDSA Engine Performance

6. Implementation

This demonstration design is implemented in Verilog HDL using Lattice Diamond® software. The synthesis tool is set to Synplify Pro®. When using this design in a different device, density, speed, or grade, performance, and utilization may vary.

Table 6.1. Performance and Resource Utilization

Family	Language	Utilization	Operating Frequency	ESB Primitive	OSC Primitive	I/O
LCMXO3D-9400HC	Verilog HDL	3084 LUT4s	>50 MHz	Yes	Yes	5

References

MachXO3D Embedded Security Block (FPGA-TN-02091)

Technical Support Assistance

Submit a technical support case through www.latticesemi.com/techsupport.

Revision History

Revision 1.0, September 2019

Section	Change Summary
All	Production release.

www.latticesemi.com