Open-Source Design of a Cryptographic ASIC

A Serial Implementation of Ascon for Side Channel Analysis and Testing

Submitted by:
Trevor Drane

Advisor:
Patrick Schaumont

March 22, 2024

ABSTRACT

This project focuses on implementing a version of Ascon, an algorithm selected by NIST as part of the Lightweight Cryptography standard, into an application-specific integrated circuit (ASIC) design. The new standard aims to address the security needs of small wireless devices with limited power applications. An ASIC design benefits these applications by drastically improving efficiency for specific tasks compared to general-purpose processors. This implementation adapts a high-speed design into a serial format by adding a serial-to-parallel interface with several test modes and a simplified input/output to improve the design's compatibility with commonly used test platforms. As physical implementations of this standard are relatively untested, this project delivers an open-source ASIC implementation of Ascon offering a valuable tool for further evaluation and research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. Introduction

New hardware developments and trends have led to smaller and smaller wireless devices that rely on limited power supplies like batteries or electromagnetic coils, which create inherent system limitations. For these devices to maintain their benefits they use high-efficiency, low-power processors that have significant limitations to their processing speed. To allow this category of devices to maintain security during communication the National Institute of Standards and Technology (NIST) created a new category of algorithms called Lightweight Cryptography. This standard prioritizes an algorithm with lower processing requirements to reduce the load of encryption on devices with limited resources.

The algorithm selected by the NIST for standardization is called Ascon which was developed primarily by a team of researchers from Graz University of Technology in Austria. The algorithm was designed to have relatively high security, a low hardware footprint, and is particularly efficient at computing results for short messages [1].

Another way to increase speed while decreasing power consumption or improving efficiency is to use dedicated hardware to perform these tasks. An application-specific integrated circuit (ASIC) is designed to perform specific tasks while optimizing for higher speeds or better efficiency than would be possible using a software program running on a general-purpose processor. The logic of an ASIC is configured to perform a limited number of specific tasks which reduces overhead but also limits its use cases because it cannot be reconfigured after it is produced. ASICs, however, provide an excellent solution for encryption on the devices described above.

Due to its recent establishment as a new standard, physical implementations of the design are relatively untested. The goal of this MQP was to implement a version of Ascon as an ASIC that can be evaluated for its functionality and security. The design was tested with several scenarios and sets of data. After verification, it was processed into a layout where the specifications were analyzed, and the design was tested for timing and correctness. The source code is published as open-source to make the design available to the public for further research.

II. Background

A. Cryptography

Cryptography is the science of converting confidential information, such as a password or private message, to a form that is inaccessible to third parties. At its core, cryptography relies on algorithms, mathematical functions that manipulate data, to encrypt and decrypt messages [2].

1) Encryption

Encryption in its most basic form is the process of converting plain text into ciphertext using an encryption algorithm and a key. The plain text is the input data that the user considers confidential. To convert the plain text to ciphertext an algorithm will use a cryptographic key, usually in the form of a string of binary a set number of digits in length. In symmetric algorithms, the key is also considered private but must be shared with the second party to decrypt the message using the same algorithm. The length of the key typically determines the strength of the algorithm to brute force methods of attack in cases where all else is equal.

Most modern internet transactions utilize encryption in some way if built using best practices: email messages, web pages, and e-commerce transactions all use encryption. Modern personal computers can be configured to encrypt their hard drives and sensitive data stored on servers or in data centers is also encrypted [3]. Below in Table 1 is an example of encryption and decryption shown with hexadecimal representation. To replicate the encryption example in Table 1 more information would be required and it is provided as an example to improve understanding. This example uses a 128-bit key and a 64-bit word length for the plaintext and ciphertext.

Table 1: Encryption and Decryption Examples

	Key	**Plaintext**	**Ciphertext**
Encryption	00010203 04050607	00010203	7763F8BA
(Plaintext → Ciphertext)	08090A0B 0C0D0E0F	80000000	E8EC18D6

	Key	**Ciphertext**	**Plaintext**
Decryption	00010203 04050607	7763F8BA	00010203
(Ciphertext → Plaintext)	08090A0B 0C0D0E0F	E8EC18D6	80000000

2) Hashing

Hashing is another form of cryptographic operation that is similar to encryption because it involves the obfuscation of plaintext into a secure format. It is often based on a similar algorithm to its encryption counterpart and will use similar mathematical operations or logic to create the output digest. It differs from encryption in a few critical ways, it is not reversible and the digest will have a fixed length irrespective of the length of the plaintext. While an encrypted message can be decrypted to reveal the plaintext, a hash cannot be undone because the length of the plaintext is lost during the calculation. Most hashing algorithms do not depend on a key unlike encryption.

The fact that it is not reversible makes it useful for applications where even the recipient of the message would not be allowed to know the contents of the message. Specifically, it is

commonly used to store passwords in a way that would not reveal the user's passwords should the password database be compromised. The password can be hashed locally on the user's device during account creation and whenever login is attempted, meaning the actual password never leaves the device even during communication and for storage in a database. Hashing has other uses in situations like the verification of information integrity and the simplification of larger sets of data for identification [4]. Below in Table 2 is an example of hashing shown with hexadecimal representation and a 256-bit hash.

Table 2: Hashing Example

	Plaintext	**Hash**
Hashing	00010203 80000000	8013EAAA 1951580A 7BEF7D29 BAC32337 7E64F279 EA73E688 1B8AED69 855EF764

3) Lightweight Cryptography

As the size of electronics has diminished and their efficiency has improved, demand for battery-powered devices and other low-energy applications has increased. The amount of data produced, and the size of files, have grown in a similar fashion demanding higher throughput devices. The NIST's Lightweight Cryptography (LWC) standard aims to increase the efficiency of encryption in these types of applications. The NIST lists Internet of Things (IoT) devices, implanted medical devices, embedded stress sensors for roads and bridges, and keyless entry fobs as some of the targets for the implementation of this type of algorithm [5]. In high-throughput applications like servers, a lightweight algorithm's light computational load allows for higher-performance implementations [6].

In April 2018, the NIST announced a competition to find a new cryptographic standard for use by small electronic devices. The standard is called 'lightweight cryptography' because of its lighter computational, and therefore energy, requirements for devices utilizing these algorithms. The announcement described the increasing number of IoT devices and electronic devices that operate using limited resources. The commonly used cryptographic standards available at the time used more resources than were reasonable for those applications. The NIST identified the need to plan ahead with the standardization of cryptography for this class of devices. Because of the vast applications of small electronic devices, the NIST stated that it would attempt to pick a versatile algorithm with wide applications and variations [7].

B. ASCON

In February 2023, the NIST announced a winner to its Lightweight Cryptography standard competition, a family of algorithms called Ascon. The algorithm defeated 56 other submissions over several rounds of selection [5].

1) About Ascon

Ascon is a lightweight cryptographic algorithm specifically designed to use less resources in both hardware and software implementations. It has low power and energy requirements and can be very low area in hardware. Ascon makes use of two important concepts that work together to achieve a high level of security while optimizing hardware usage: a sponge and a permutation. Ascon reuses its permutation operation, the mapping of the algorithm's internal state back onto

itself, to allow for the same gates to be used repeatedly. This also allows for scalability in speed because these operations can be done in series to optimize for power consumption and core area, or in parallel to prioritize speed. The sponge structure, which allows the algorithm to 'absorb' a long message for use with hashing or authentication, can also be shared between operations and saves die area in an embedded design. Figure 1 depicts the sponge structure of Ascon as presented to the NIST competition and how it would be used to compute a hash digest where M is the plaintext, H is the hashed output, and the block p^a represents a rounds of permutations.

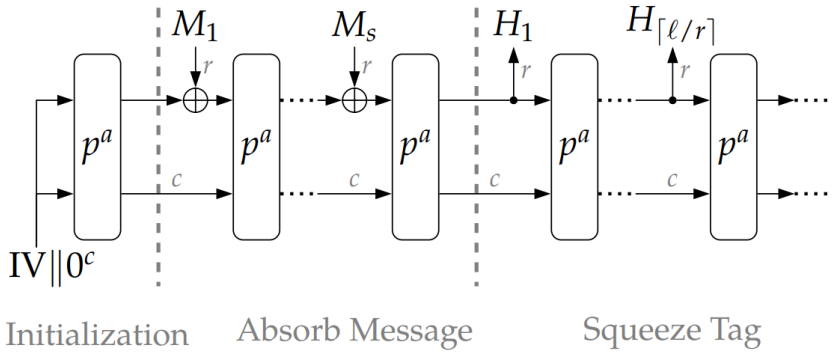

Figure 1: Ascon Sponge Structure [1]

Ascon also prioritized built-in side channel protection that is featured in the architecture of the algorithm and does not require special hardware. This project focuses on Ascon-128 and Ascon-Hash but there are several other variations of Ascon. Ascon-128 has twelve rounds of permutation processing for initialization and finalization and six rounds of intermediate permutation processing. In other words, it uses twelve rounds to absorb the key and nonce for initialization, and twelve rounds to produce or check the tag. It uses six rounds to process each block of the associated data and plaintext.[1].

The version of Ascon selected for hardware implementation uses Ascon-128 and Ascon-Hash. Ascon-128 is an encryption/decryption algorithm that uses authenticated encryption with associated data (AEAD). Authenticated encryption (AE) identifies encryption that generates a tag to go with the ciphertext and is required to decrypt the message. The algorithm requires the verification of the tag of 128 bits before the plaintext is returned to the user. This means that if the message is tampered with in any way or the message is attempted to be decrypted without the tag the plaintext will not be returned to the user. AEAD varies from AE by also requiring the input of associated data (AD) which must be the same between encryption and decryption for the message to be decrypted correctly. AD is typically a non-private piece of information not limited in length that accompanies the private portion of the message like a title or name. The AD adds additional dependencies and context to the operation and is included in the calculation of the tag which will prevent tampering even without encrypting the AD. Ascon also requires a key of 128 bits and a matching nonce of 128 bits to encrypt and decrypt. The nonce is a randomly generated number that is required for encryption and decryption but should not be used twice between two messages to ensure the same message is never encrypted the same way twice. Figure 2 below shows the order of information and the type of information required to perform encryption or decryption; the nonce is labeled as Npub. The data block size for AD, plaintext, and ciphertext is 64 bits [8].

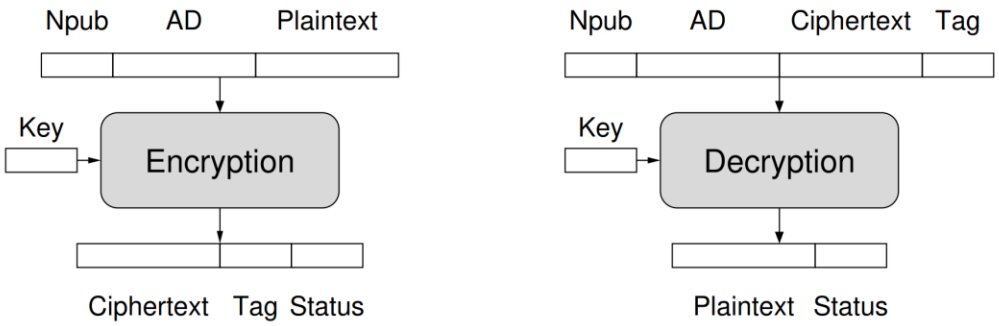

Figure 2: Diagram of Required IO Data Fields for AEAD [8]

Ascon-Hash is the matching hashing algorithm which also has a data block size of 64 bits but only requires the message as input and will output a hash of 256 bits. Twelve rounds of processing are used to produce the hash once indicated the plaintext is provided in its entirety [1]. An example of the required Ascon-Hash inputs and outputs is shown below.

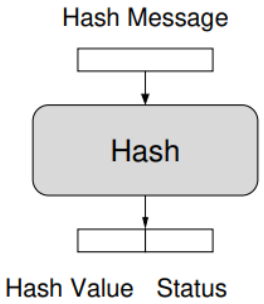

Figure 3: Diagram of Required IO Data Fields for Hashing [8]

2) SystemVerilog Hardware Implementation

This project uses a SystemVerilog implementation of Ascon written by Robert Primas using algorithm variants Ascon-128 and Ascon-Hash. The interface of the implementation is influenced by research by George Mason University's Cryptographic Engineering Research Group. The resulting interface uses 32-bit buses to transmit the data in and out, and provide a key. A four-bit bus is used to indicate the type of data being transmitted which can be nonce, AD, plaintext or ciphertext, tag, or hash which is labeled in order from one to five. The rest of the pins are described below in Figure 4 [9].

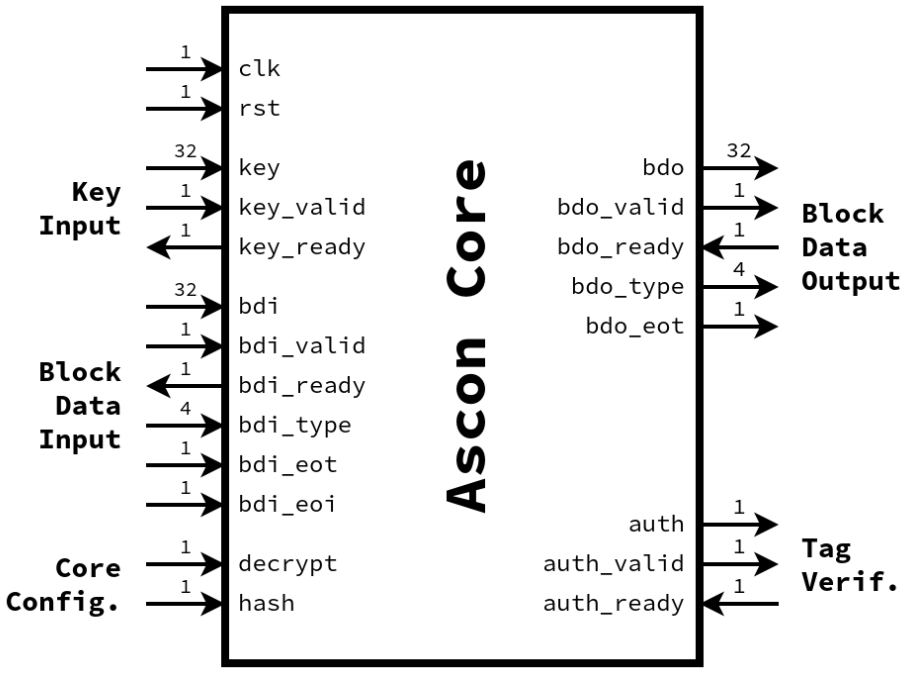

Figure 4: Interface Diagram of the System Verilog Ascon Implementation [9]

The design can perform encryption, decryption, and hashing and the version chosen for implementation performs one round of permutation processing per clock cycle. The design is scalable and other versions of the core are available that can process 2, 3, or 6 permutations per clock cycle which can be created by changing the UROL parameter in config_core.sv.

C. Open Source

The goal of this project is to develop open-source tools to allow for further development and testing of the Ascon algorithm. Open-source projects make their source code available to anyone interested in viewing it. They also allow anyone to modify it and contribute to the development of the project or share it themselves. These two concepts are what make open-source projects so great, they allow them to have a potentially infinite source of knowledge and perspectives contributing to their development. Ideally, this is great for security because it allows contributors to check each other's work and examine the source code for weaknesses. It allows users to add features as they wish, and maintain the code without limitations or having to rely on a profit-oriented organization for its support [10].

However, the open-source aspects of this project have some limitations. The Cadence tools used for the development of this project are not available to the public, meaning none of the scripts used in creating the layout can be published. There are however other open-source tools available, like OpenLane, that can be used to create a layout with similar results.

III. Developing a Serial Test Interface

A. Design Criteria

The biggest constraint for the final device is compatibility with the IO on the ChipWhisperer CW308 UFO main-board. The ChipWhisperer board does not have enough pins available for data transfer and perform the configuration necessary to operate the core. It has only ten pins available for target-specific usage and only 60 pins total including several for various voltages and ground. The existing implementation of Ascon described in SystemVerilog uses 32-bit buses to transmit most of its data and has a total of 121 data pins. This interface is great for high-speed use cases but is not great for testing applications where IO is limited. Figure 5 shows a block diagram of how the ChipWhisperer setup would be used with the interface.

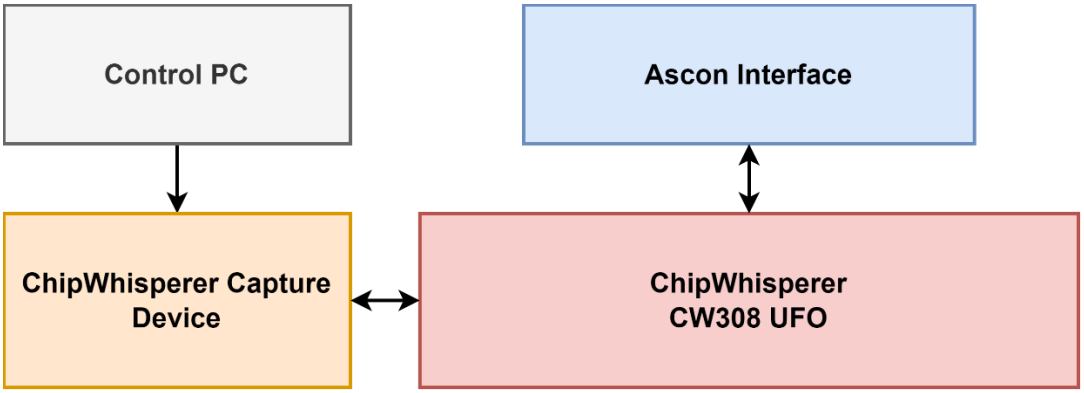

Figure 5: Block Diagram of a ChipWhisperer Setup

To overcome this discrepancy, a serial interface was developed in SystemVerilog that will act as the new top module. The interface only has nine total top-level data pins and will receive a serial stream of instructions and data before converting it to parallel data and control operations that can operate the core module at full speed. Its operation is configurable with several test functions allowing for encryption, decryption, and hashing with various message lengths and run times. It can operate for longer run times by performing the same operation repeatedly using the same data that was provided at the start. It can run encryption and hashing with message lengths that are longer than the input registers by repeating its input configuration to simulate a longer message. Figure 6 below shows a diagram of the hierarchy of the interface as well as its top-level pins and connectivity to the Acon core. Busses with more than one pin are represented by an arrow with an open head.

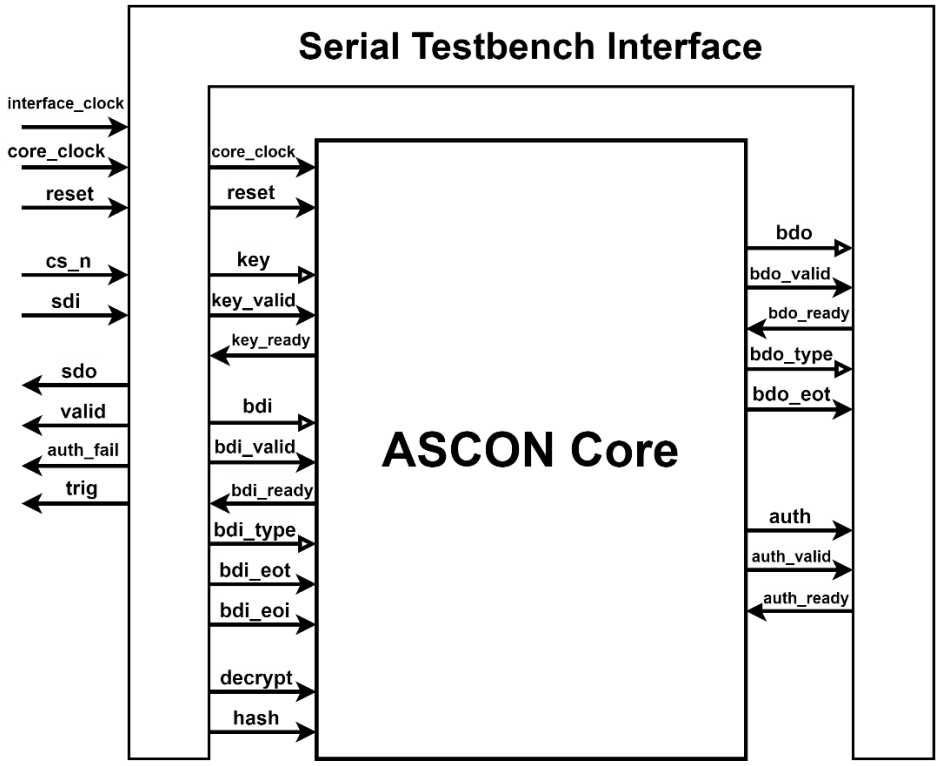

Figure 6: Diagram of the Serial Testbench Interface Implementation

Beyond overcoming IO limitations, the goal of this interface is to operate in a fashion where the speed of the core is not limited by the speed of the serial interface. To allow the user to test the limits of the core while still communicating with the interface at reasonable speeds the interface has two clock inputs that control different logic in the device. The device needs to be able to operate the serial input and output at a slower speed like 38.4 kHz while still running the core at up to 100 MHz or more.

A trigger pin was also added to the design, as shown in Figure 6, that provides a reference to the user about when the core starts a new permutation and can be used for testing.

B. IO Features

The interface module has nine top-level data pins; five of these pins are inputs and four are outputs. The inputs consist of two clocks: the interface clock (interface_clk) which configures the speed for all of the IO, and the core clock (core_clk) which controls the speed of the core module and its operation. There is also a reset pin (rst), a chip select pin (cs_n), and a pin for serial data in (sdi). The four outputs are the serial data out pin (sdo), an output valid pin (valid), an authentication failure pin (auth_fail), and a trigger pin (trig). A diagram of these pins is shown in Figure 6 and the extent of their use and functionality is described in section III.D.

C. Operation Instructions

This section will go over information important to the proper operation of the interface, including instructions and behavioral features.

1) Instruction Set

The interface is designed to work with a set of instructions and data headers that were created to communicate with the serial interface without the need for additional control pins like the pins used in the core module. Each instruction must be 32 bits long and will contain a four-bit operation code followed by either zeroes or a configuration number determined by the type of instruction. The instructions are listed below in Table 1. There are three main types of instructions that need to be used. Three core configuration instructions are used to communicate to the interface which cryptographic operation to configure the core to perform. They use hex codes 0x0 through 0x2 to configure either encryption, decryption, or hashing and do not require a configuration number. The second type of instruction is the data type header that must be provided to the core to indicate the type of data that will be provided by the user in the next 64 to 128 bits. They use hex codes 0x3 through 0x8 to identify the key, nonce, AD, plaintext, ciphertext, or tag and are followed by a number indicating the size of that data as either sixteen or eight bytes. The final type of instruction is for interface configuration and should be the last serial information sent to the core because it will also tell the interface to begin the test run. They use hex codes 0x9 through 0xB to indicate that the interface should perform single (SINGLE) or repeated (RPT) cryptographic operation, or simulate a longer plaintext message (VML). The RPT and VML interface instructions should also contain a configuration within the ranges listed in Table 3.

Table 3: Interface Instructions and Headers

Instruction (32 bits)	Operation Code	Config Range	Config Use
Core Encryption	0x0000	0x0000	NA
Core Decryption	0x1000	0x0000	NA
Core Hashing	0x2000	0x0000	NA
Key Header	0x3000	0x0010|0x0008	Length in Bytes
Nonce Header	0x4000	0x0010|0x0008	”
Associated Data Header	0x5000	0x0010|0x0008	”
Plaintext Header	0x6000	0x0010|0x0008	”
Ciphertext Header	0x7000	0x0010|0x0008	”
Tag Header	0x8000	0x0010|0x0008	”
Interface Single Operation	0x9000	0x0000	NA
Interface Repeated Operation	0xA000	0xFFFF-0x0001	OP*Config*2^{16}
Interface Variable Message Length	0xB000	0xFFFF-0x0000	OP*(Config+1)

2) Clocks

The interface has two clock inputs that must be driven for the device to operate. The interface clock operates the logic that controls the input and output of the serial data and should be set to the same speed as serial communication is expected. The core clock controls the logic that operates the core and is passed to the core module. The details about how fast the clocks can be driven vary by architecture but the specifics can be found in section V. The system is designed so the clocks do not have to be driven at the same frequency, allowing the full potential of the core to be tested with a faster clock speed while keeping the speed of the serial interface at reasonable speeds.

3) *Operation*

The serial input to the device is enabled by the selection of the chip select (cs_n) pin to its active low state. The test function will continue and data will be returned without the activation of the cs_n pin. The serial data in (sdi) pin operates at the frequency of the interface clock and is used to load the information required to configure the interface. The interface expects an instruction as the first piece of data provided to tell the interface what the next type of information is or to configure the interface. Instructions should be loaded most significant bit first. The order the data is provided to the interface does not matter to its operation however, all of the required types of data must be provided to operate correctly, and the interface instruction must be provided last because it is used to initiate the test run. The required types of information, regardless of the type of cryptographic operation intended, are a core instruction, a plaintext or ciphertext instruction and its data, and an interface instruction. Until the interface instruction is given any of the information that was already loaded can be overwritten. The reset pin is synchronous to the clock controlling each register and should be held high for at least one clock cycle of the slowest clock.

Output from the serial data out (sdo) pin is returned after the completion of the configured test operation. The valid pin will be enabled when the output is ready to indicate the return of valid data. Only one or two types of data will be returned per operation. The plaintext, ciphertext, or hash will always be returned first. Only the ciphertext return for encryption will be followed by the tag, The tag will be separated by one clock cycle where the valid pin is disabled. The trigger pin is used to indicate the first clock cycle in a round of permutations.

If the tag provided for decryption does not match the one calculated, the authentication fail pin is activated, this will also prevent the return of the data to the user. To recover from this state the reset pin must be used.

The best way to understand the specifics of the operation of the interface beyond this description would be to run it using the testbench in a simulation tool and observe the waveforms of the device.

D. *Major Components*

The source code for the hardware descriptive language used to design the following structures is available on GitHub and listed in Appendix A.

1) *SIPO*

The first substructure described in the interface is the serial input parallel output (SIPO) shift register on lines 55-86 of ascon_spi.sv. This section takes the serial input from the sdi pin on the package and converts it to 32-bit parallel data that can be stored in the data registers. The shift register is activated by the active low chip select pin cs_n. The structure also counts the number of shifts since the start of the interaction and provides a done flag to the interface FSM to signal the reading of the data. The SIPO conversion is the only system in the device that is controlled directly by the user.

2) *Finite State Machines*

The finite state machine (FSM) is the most complicated structure in the interface because it is required to interact with both clock domains and control the actions of nearly every other structure in the interface. The finite state machines are described from line 183 to 382 and their

enumerated types are defined at line 88. It is built from two finite state machines that are on two separate clocks with buffers in between to reduce the chances of metastability during the cases where they need to interact. An overview of the FSM structure is shown below in Figure 7.

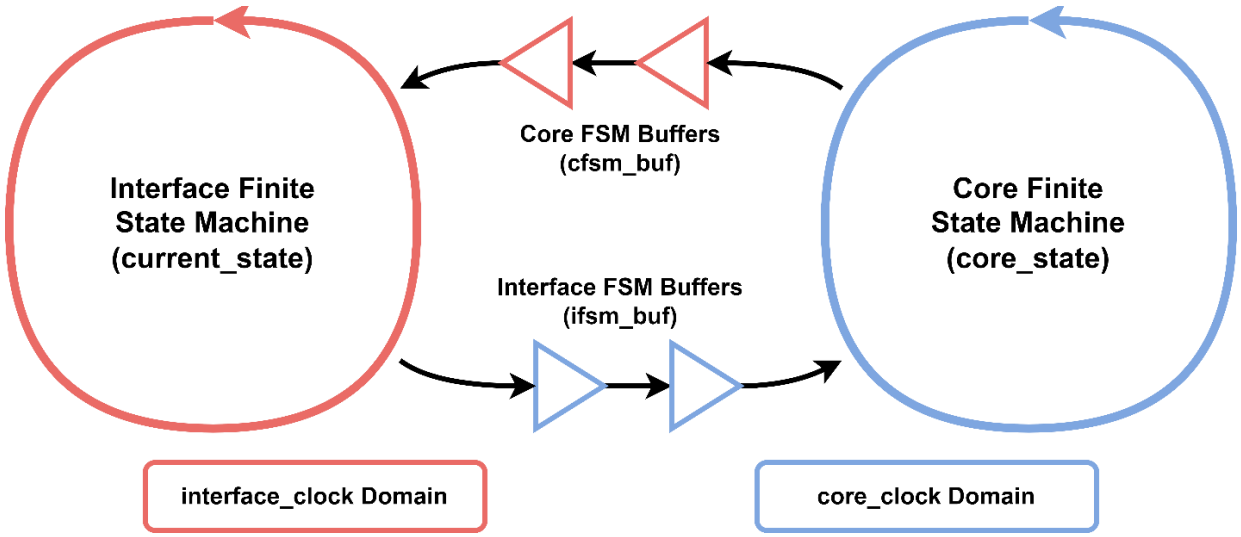

Figure 7: Top Level Finite State Machine Flow Diagram

The interface finite state machine is synchronized by the interface_clock and controls two main structures: the logic for storing the parallelized data in their appropriate data registers and the parallel in serial out (PISO) return order. The interface FSM is also used to enable the core FSM. The description of the interface FSM starts on line 197 of ascon_spi.sv.

The first information the interface FSM expects is an instruction defining the core mode, (encryption, decryption, or hashing) or a data type instruction that tells the interface how long the next block of data is and what type of data it is. If it's a data instruction the FSM moves to the next state where the parallelized data will be stored in the appropriate registers. Once data of the expected length is provided the interface will then expect another instruction. If all of the required data is already loaded the user can provide one of the three run mode instructions. Once a run instruction is provided the configuration is stored and the interface FSM will enable the core FSM.

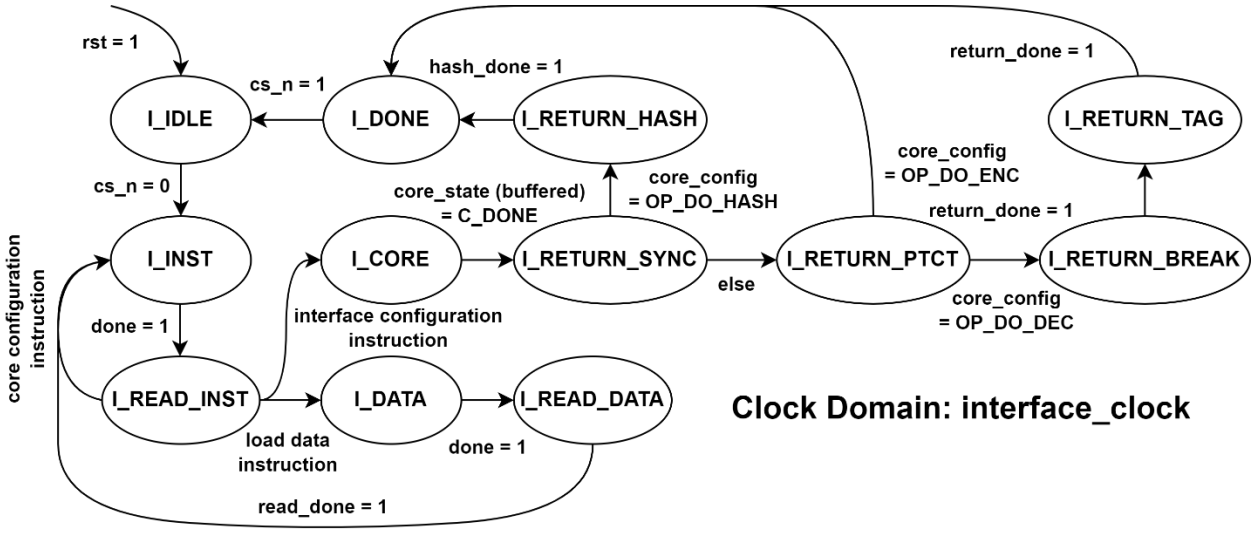

Figure 8: Interface Finite State Machine Flow Diagram

The two FSMs need to communicate with each other, however, because they run on separate clocks there is a chance for there to be timing violations between the clock domains. The buffers are inserted to reduce these chances and give the data two clock cycles to settle out of metastability. The core FSM is enabled once the interface FSM buffers show that the interface FSM is in the state I_CORE. The interface FSM will then wait for the Core FSM to be in the state C_DONE before returning the data using the PISO converter.

The core FSM description starts on line 277 of ascon_spi.sv and has significantly more logical checks than the interface FSM. It checks the core mode and the interface mode to determine how it is supposed to run with a total of nine different configurations. The three core modes are encryption, decryption, or hashing, each of which requires a different order of operations and slightly different behavior required to operate the core correctly. The three interface modes also each require different orders of operation to function correctly. The core FSM is mostly in charge of operating the high-speed logic that controls the inputs to the core module including the various flags, data types, and data buses shown in Figure 4. It is also responsible for controlling the logic that makes sense of the output signals from the core and preventing the return of information in the case of authentication failure.

Figure 9 below shows all of the possible paths of flow for the core FSM. However, some of the logic is too complicated to be summed up in the diagram. For example, C_RUN has a total of three, layered, logical checks and three possible next states. For a complete understanding of how the core FSM works see the SystemVerilog in Appendix A or ascon_spi.sv on lines 277 through 373.

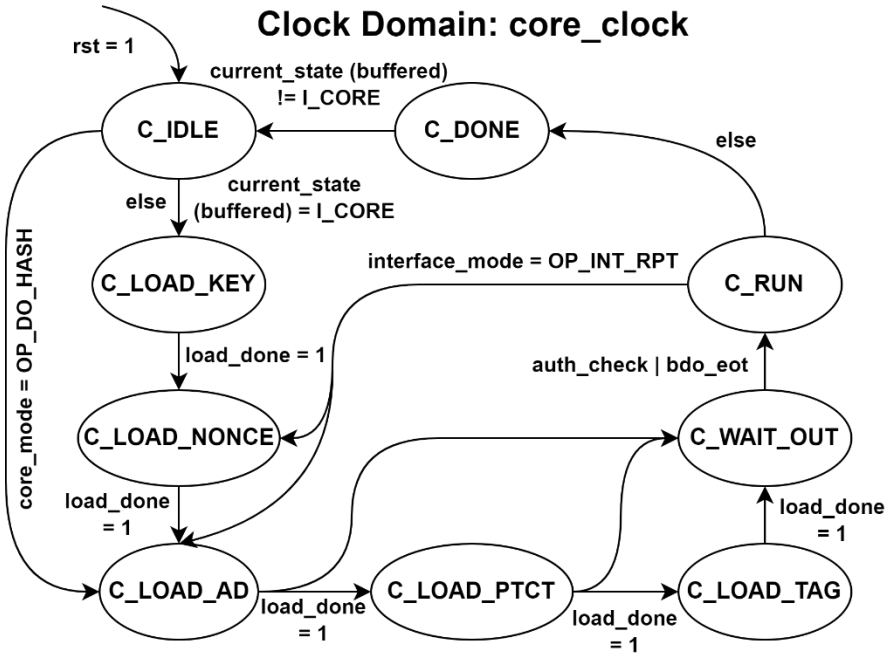

Figure 9: Core Finite State Machine Flow Diagram

3) Interface Test Mode Control

The finite state machines described above also make use of the interface test mode control logic defined on lines 384-420 of ascon_spi.sv. This structure is used to control two of the three test modes: variable message length (vml) and repeated operation (rpt). Each mode has its own logic which is a counter that controls a done flag that is thrown when the counter reaches a value greater than or equal to the number configured by the interface instruction. The way these modes work and information about how to configure the interface using instructions is described later in Section III.D. The flags created by this logic are mostly used by the core FSM and the core control logic.

4) Data Registers

The data registers are broken down into two parts: the input registers and the output registers. The input registers store the data loaded by the user for use in testing the core, this part of the device is controlled by the interface FSM and the interface clock. The output registers store the results of the test run with the core. This part is controlled by the core clock and the core FSM because it interacts with the core module which also uses the core clock. The input registers consist of one 4x32-bit register each for the key, nonce, AD, plaintext or ciphertext, and tag. The output registers are one 4x32-bit register for the plaintext or ciphertext, and one for the tag. There is a separate 8x32-bit register for hashing output.

The logic for loading the input registers is on lines 422-508 of ascon_spi.sv and in Appendix A. The logic uses a 32-bit header instruction to put the data in the appropriate register. The header instruction will tell the interface what type of data will be provided next and how long that data will be in bytes. This data must be provided in chunks of 32 bits.

The logic for loading the output registers is on lines 594-619 of ascon_spi.sv and has the much simpler task of reading the output of the Ascon core and storing it in the correct registers.

When the output is valid it reads the 32-bit output bus from the core to the register associated with the type of data indicated by the bdo_type lines. It will count each clock cycle that data is transferred and reset that count after the core indicates the end of the type (bdo_eot).

5) Core Control

The next significant component in the interface is the core control section, which is defined from line 512-591 of ascon_spi.sv. This structure controls all of the input lines on the Ascon core module making significant use of the core FSM current state. This means that this section of the interface must also use the core clock. The structure of an FSM works great for this kind of application because the type of output required by the core varies greatly from one type of operation to another but also follows a very predictable order that can be described well by using an FSM. The FSM can also be configured by input instructions to change the order of the FSM without changing the structure of the logic in the core control. In summary, the FSM and core control structures work together and improve the code's readability. The core control uses a case statement to define this structure where each case gets its own core FSM state that will define the input to the key or the bdi lines. It will read the information from the input data registers and pass it to the output that controls the core. This section can access the same registers as the input data logic even though it uses a different clock because it is only reading the data from those registers, and it is never enabled at the same time that the data in those registers would be changed.

6) PISO

The parallel in serial out (PISO) conversion logic is the part of the interface responsible for producing the output for the sdo pin. This logic is described from line 623 to 697 of ascon_spi.sv. The PISO conversion is controlled by the interface FSM and reads data that was stored in the data output registers. It works by reading the required data from the data registers to a shift register and then shifting that data through the register so that each bit is read from the thirty-first bit in the array to the output. This logic also controls the valid pin which indicates that the information provided by the sdo pin during that clock cycle should be recorded as part of the output data.

7) Encryption Core

The encryption core was designed by Robert Primas and is available on GitHub. The top of the core module is shown in Figure 4 and is instantiated in the ascon_spi.sv file as core. This is the module that contains the Ascon algorithm logic necessary to perform encryption and hashing. Only small changes were made to his code to ensure compatibility with Cadence tools and add the trigger pin that will be used for testing purposes without modifying the functionality of the core.

IV. Design Verification

A. Test Bench

The verification of the interface's functionality was confirmed using a custom testbench that was expanded over time as different aspects of the device were being tested. In its final form, the testbench can read a test vector file in hexadecimal format to the inputs of the interface and record its outputs to the command line. It will run until either the specified time (in ns) is reached or the output is finished, whichever happens first. It can run multiple test sets from the same file when they are separated by the keyword WAIT, where it will wait for the previous run to finish, reset the interface, and begin the next test. The results of the testbench are not checked automatically and must be verified after each run. The testbench implements two submodules, the Ascon interface as the device under test (dut) and a simple serial in parallel out (SIPO) module that is used to read the output from the interface into 32-bit words. The hierarchy of the testbench setup is shown in Figure 10 and the testbench SystemVerilog code is in Appendix B.

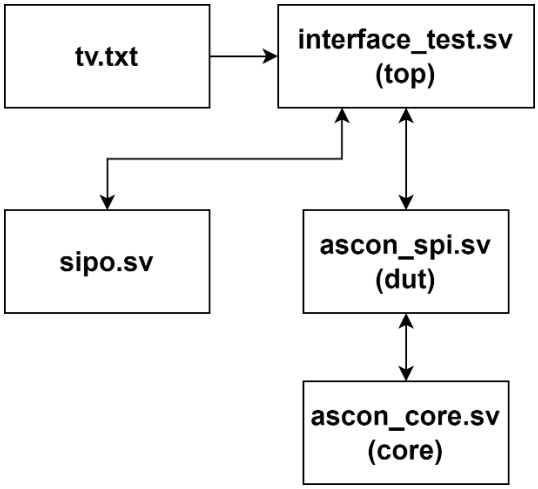

Figure 10: Interface Testing Hierarchy

The testbench does have some quirks, mainly that it outputs a waveform to the interface using the negative edge of the interface clock. The reasoning for this is discussed later in the complications section but does not affect the testing functionality. It also uses the system function $value$plusargs to allow the user to easily configure which test vector set they want to use without recompiling the testbench. An example of how to use the testbench makefile commands would be ‘make sim TVSET=1’. Files in the sim folder are compatible with tools like AMD Vivado and Cadence Xcelium. An example testbench compatible with Icarus Verilog and GTKWave will be made available in the ivsim directory to provide an accessible example waveform.

B. Test Vectors

There are four sets of test vectors provided with the interface, each set is designed to check different functionality of the interface to make sure each part is functioning properly. Each set tests each core mode while varying some other aspect of the test vector set. The first test vector set checks the functionality of encryption, decryption, and hashing, using the simplest

interface instruction for a single operation of each type. The second test vector set uses shorter message lengths of only 64 bits for encryption, decryption, and hashing. The third set tests the second type of interface instruction and testing mode for repeated messages. The length of each run can easily be changed by modifying the configuration number at the end of the instruction in the test vector file. The last test vector set checks the final test mode included in the interface the variable message length. This set shows the limitations of the variable message length operation because no matter how it is configured the decryption part of this message will always fail to pass authentication. The test vectors were hand assembled and adapted from the test vectors generated using a reference Python implementation of Ascon intended to create test vectors for the encryption core. Test vector set number one is included in Appendix C as an example.

V. Layout

A. Layout Process

Building a layout involves many steps of testing and synthesis throughout the process of converting the hardware descriptive language (HDL) programming into a map of standard cells and wires that make up an ASIC design. There are several layers of scripts and designs that all work to ensure a functional layout that meets the designer's intentions.

After the HDL design is verified using the setup described in Section IV the first step in creating the layout is synthesis. Synthesis converts the HDL code to a netlist using a library of standard cells from the architecture chosen to fabricate the design. A standard cell is the logical building block of an integrated circuit (IC) and each cell describes the silicon structure that makes up a standard logic gate. A netlist lists each required standard cell and wire used to describe how all of the cells are connected to reproduce the interface design. After a netlist is created, static timing analysis (STA) can be done on the design using a database of timing values that describes the standard cells. STA calculates the signal propagation times between standard cells to verify all signals arrive before the clock edge for a given clock speed. From there, the layout can be generated using the netlist which places each of the standard cells on a two-dimensional plane and draws their metal interconnects on the given number of layers. It also places a network of power distribution and connects the clock wires to each register.

After the layout is generated gate-level (GL) simulation steps can be taken that now take into account more physical properties of the design like the length of wires and the actual number of standard cells required for the design. The GL simulation step is done to confirm the functionality of the design after it has been converted to standard cells. GL simulation can use the same Verilog testbench and test vectors to make sure the operation is identical. GL STA should be done to confirm the addition of wires and physical distance between cells will not create any timing violations. GL power simulation will provide a more accurate estimate of the power consumption of the chip. These steps are necessary to check that the design is still functional after the synthesis and layout conversion steps.

B. Pads

Pads are additional cells required to create contact points for connecting bond wires to complete the chip package. Most steps from the process above need to be repeated with some minor modifications to add the pads to the final layout. The pads have a separate library and need a separate module declaration in a Verilog file to define their connections to the design.

The pads are laid out with input pads on the top and left side, output pads on the bottom and right side, and one positive and one ground pad on each side of the chip. The order of the IO pads is defined in the chip/chip.io file.

C. Final Layout

The final layout design produced using the scripts, the interface design, and Cadence tools is shown in Figure 11. Figure 11 shows a closeup of the die with the standard cells and metal interconnects and layers depicted with various colors. Each color represents a different metal layer or via between layers. There is a ring of metal for power distribution shown in brown and light blue that surrounds the chip and a layer of blue interconnects that cross the chip to deliver power to each standard cell.

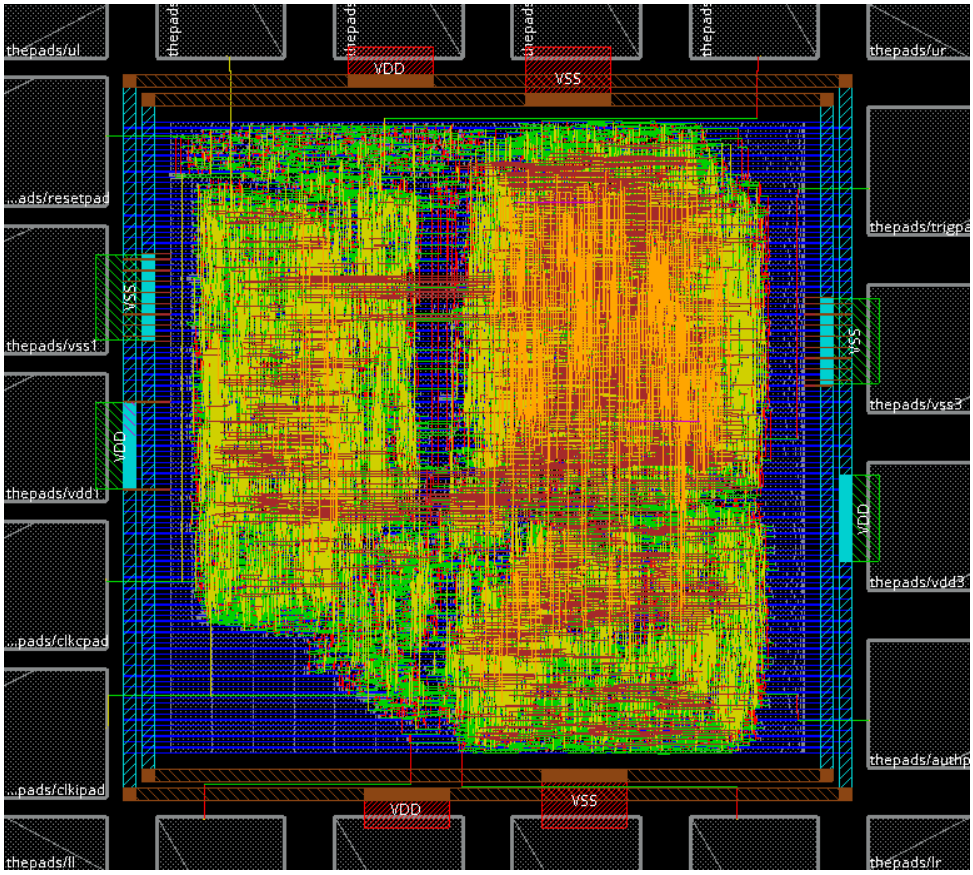

Figure 11: Closeup of the Final Layout

D. Design Specifications and Analysis

Cadence's layout design tool allows the user to highlight modules of a design and break down the number of cells and their area. The interface makes up more than half of the area and and number of cells in the whole device at 5,259 out of 9,276 cells. The serial interface top module is highlighted in red in Figure 12. The Ascon core is made up of a total of 4,017 standard cells cells with its submodule for calculating permutations using 1,678 of those cells. The core module is highlighted in pink and the permutation is highlighted in green. These numbers are summarized in Table 4.

Table 4: Module Specifications

Module Name	Cell Count	Active Area (µm)	Color
ascon_spi	5,259	16,805	Red
ascon_core	2,339	6,634	Pink
asconp	1,678	3,699	Green
Total	**9,276**	**27,138**	**N/A**

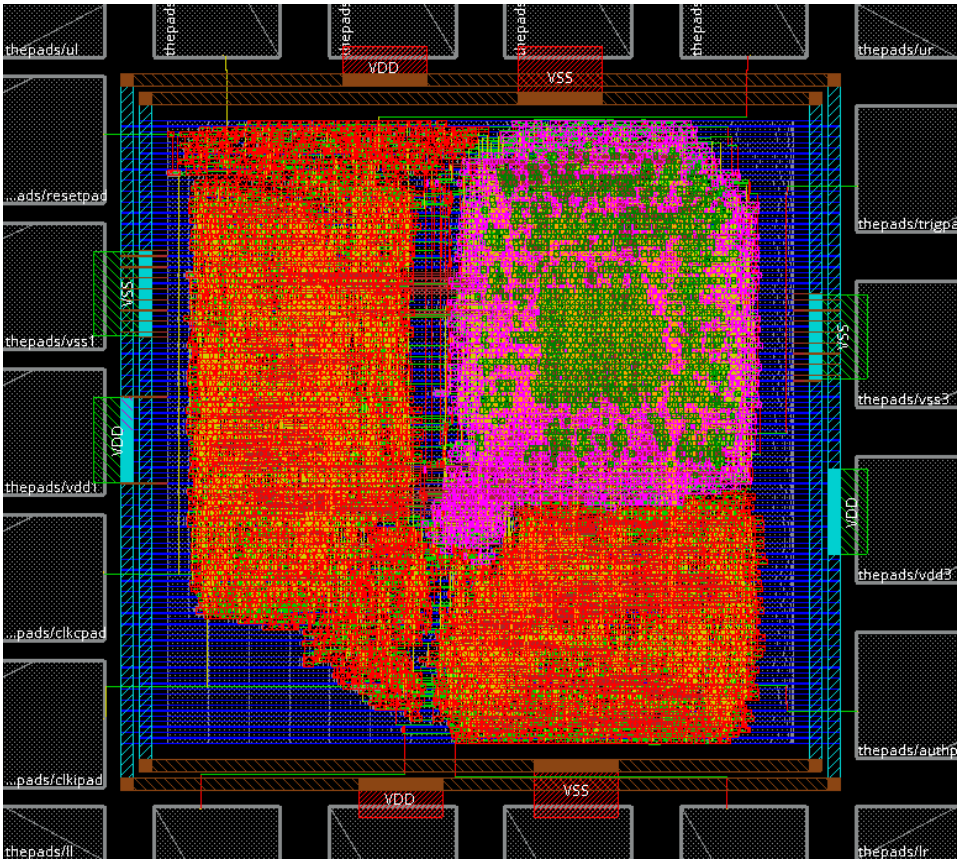

Figure 12: Layout of Final Design with Modules Highlighted

The following Table 5 lays out the results from RTL synthesis performed to create the first type of netlist. This step was performed with both 130 nm and 45 nm architectures using fast and slow operating conditions. The clock period was set at 10 ns for 45 nm and 20 ns for 130 nm which is equivalent to 100 MHz and 50 MHz respectively. These speeds were selected to not act as the limiting factor in the fast or slow design and not act as a variable. The active area represents the area taken up by those cells without taking into account the space needed for interconnects or wasted when creating a layout. There is minimal difference between the fast and slow operating conditions for each respective architecture where both 45 nm designs use about 25,000 μm^2 and the 130 nm designs use about 90,000 μm^2. All of the designs will use 1,875 registers to store information regardless of architecture while the 45 nm designs use about one thousand more standard cells than the 130 nm designs.

The next important statistic is the critical path, which is the path between registers and/or IO that has the longest signal propagation time. For both slow designs, the critical path is in the interface module, and for both fast designs, the critical path is in the core module. All critical paths are constrained by the core clock. The slack represents the time left after the arrival of a signal before the required arrival time which is mostly determined by the arrival of the next clock edge. The slack listed below is the slack on the critical path, a higher slack is better, but also means that the clock speed could be increased to operate the device faster. The final specification is the power draw calculated in milliwatts as an estimate of the power draw for the final device. Typically the power draw increases with faster clocks or conditions, and larger, less efficient architectures which is consistent with the results in Table 5.

Table 5: RTL Synthesis Design Specifications

RTL Synthesis	45 nm Slow	45 nm Fast	130 nm Slow	130 nm Fast
Clock Period (ns)	10	10	20	20
Clock Frequency (MHz)	100	100	50	50
Active Area (μm²)	25311	25156	90865	90231
Cell Count	9848	9966	8990	8958
Register Count	1875	1875	1875	1875
Critical Path	vml_counter[0] → vml_counter[31]	fsm[1] → auth_intern	interface_config[0] → bdi_eot	fsm[1] → state[4]
Critical Path Clock	core_clk	core_clk	core_clk	core_clk
Slack (ns)	6.270	9.123	11.453	16.011
Power (mW)	0.689	1.586	7.044	8.692

The results in Table 6 are calculated using the same process as the results in Table 5 except the clock speed has been increased to the limits of the architecture and timing specifications. The 45 nm architecture provided the fastest results operating up to 2 GHz under optimal conditions. When the clock speed becomes a limiting factor the critical path becomes more consistent between the designs occuring only on a path from the state register in the core module. Besides the increase in clock speed and slightly different critical paths is higher power consumption caused by an increased switching frequency.

Table 6: Maximum Clock Speed RTL Synthesis Design Specifications

RTL Synthesis	45 nm Slow	45 nm Fast	130 nm Slow	130 nm Fast
Clock Period (ns)	2.5	0.5	6	2
Clock Frequency (MHz)	400	2000	166.7	500
Active Area (μm²)	25461	25287	90629	90657
Cell Count	9974	9730	8723	8785
Register Count	1875	1875	1875	1875
Critical Path	fsm[1] → state[4]	fsm[2] → state[1]	state[3] → state[1]	state[3] → state[0]
Critical Path Clock	core_clk	core_clk	core_clk	core_clk
Slack (ns)	0.005	0.006	0.159	0.007
Power (mW)	2.752	33.431	24.189	87.875

Unfortunately, the Skywater 130 nm open-source architecture development library is not compatible with Cadence's layout tools, meaning that the 130 nm design cannot be developed past a netlist. As a result, the design for the final layout is limited to the 45 nm architecture. The results of the final layout design using the 45 nm architecture are listed below in Table 7. The table lists four designs using two different timing libraries, each timing library was used to create one layout at 10 ns clock period and one at the upper limits of the design. The biggest difference between the two sets of data is the slight decrease in speed resulting from the additional calculations required by the physical distance between standard cells. The layout designs are about half as fast as their netlist counterparts at maximum clock speed. The layouts also result in

multiple different area calculations that include different aspects of the device. The core area is the calculated area of the logic design excluding the pads but including the space in the layout surrounding the standard cells. The layout results in about a fifty percent increase in area and somewhere between 65 and 70 percent efficiency in area usage. The chip area represents the area taken up by the entire chip design including the pads. The pads increase the area of the design significantly up to 545,000 µm² but are necessary to implement the design as a physical chip. Other specifications of the design are not significantly altered aside from small decreases in the number of registers. The critical path remains similar to the netlist designs.

Table 7: Final Layout Design Specifications

Final Layout 45 nm	**Slow**	**Slow Max Clock**	**Fast**	**Fast Max Clock**
Clock Period (ns)	10	4	10	1
Clock Frequency (Mhz)	100	250	100	1000
Core Area (µm²)	38819	39434	37078	37270
Chip Area (µm²)	543442	545752	536879	537609
Cell Count	9289	9407	9289	9185
Register Count	1875	1875	1875	1875
Critical Path	fsm[0] → auth_intern	fsm[3] → auth_intern	fsm[0] → state[3]	fsm[1] → state[0]
Critical path clock	core_clk	core_clk	core_clk	core_clk
Slack (ns)	5.827	0.033	8.960	0.053

VI. Complications and Further Development

A. Simulation

Several complications came up during the simulation step that slowed the progress of development for the interface and led to the inability to complete the project in an architecture that could be fabricated before the end of the project. Most issues did not result in too much of a delay, however the setbacks added up over the length of the project.

1) Timing and System Functions

There were several weeks lost towards the end of B term that were spent trying to chase down an issue with the simulation of the core module using the provided testbench. This was particularly confusing because the core would simulate properly while using Icarus Verilog and was mostly functional while using an early implementation of the interface. It turns out that both Cadence's Xcelium and AMD's Vivado do not impose the same timing restrictions on system functions, like $fscanf, $fgets, or $sscanf, which are applied to assignments that are implemented synchronously. This meant that the testbench built by Robert Primas for verification of the Ascon core would not simulate properly with Xcelium or Vivado because it used $fscanf to read test vectors to the core module. This manifested as waveforms that looked like they should operate the core correctly but when digging a little further showed that the data assigned directly by $fscanf was changed a touch too early resulting in a mismatch between the control flags and the data bus leaving empty data registers in the core.

In a sense, this error in the way the system functions work was discovered during the simulation process while building the custom testbench for the interface (interface_test.sv). There were some inconsistencies discovered surrounding the testbench towards the beginning of the project where assignments would be read from the beginning of the clock cycle rather than the end. To get around this, assignments in the testbench were changed to occur on the negative edge of the clock cycle. This did not uncover the root of the problem but allowed the simulation to continue with only minor delays. The timing error did not show up between the interface (ascon_spi.sv) and the core module because all assignments to the core are done synchronously with non-blocking assignments.

There are some separate complications with system functions where IcarusVerilog has different requirements for their arguments. This means that the testbench and test vectors had to be redeveloped for IcarusVerilog to allow for different simulation options.

2) Gate Level vs. RTL Simulation

There are a few structures that were discovered during the testing phase that caused issues with gate-level simulation even though they passed RTL simulation without issues.

a) Complicated Assignments

Some complicated if statements caused issues when it came to gate-level simulation even though they passed RTL simulation. This revealed itself as metastability that propagated through most signals in the system over time. It was resolved by examining the waveforms of the gate-level simulation to find the source of the metastability. There were several cases where the issue was resolved by simplifying the condition of an if statement to a simple true or false statement and moving the complicated logic to an assign statement.

b) Case Statements

Ideally, the design should not make use of latches especially if the structure is intended to be a register. Some Verilog structures can cause registers to be interpreted as latches if the logic does not provide a case for every logical possibility. These can be solved by making sure that all case statements have a default case assignment for each value assigned in the always block. Other solutions include providing an initial value to each wire in the procedural block or making sure that if statements have their complementary else statement.

c) Reset Cases

Some interface and core registers were missing reset cases. This resulted in RTL simulation that was functional, however, this oversight revealed itself during gate-level simulation. Adding reset cases to these registers and verifying that the core is reset before beginning simulation ensures that the device will behave as expected.

B. SIMPLIFICATIONS

The process of designing the serial interface was a significant learning experience about the specifics of SystemVerilog. It allowed experimentation with what works and what does not through many different types of applications. As a result, the design is not optimal, there are significant simplifications that could be made to some of the structures that could reduce the area of the design and also improve the readability. There are almost certainly unnecessary registers that could be removed by simplifying logic and implementing a better system of combinational control signals. Almost the entire system of core loading logic could be simplified to using purely combinational logic eliminating at least four registers.

Other simplifications also involve the optimization of register usage, simplifying the size of registers to eliminate bits that are never used by an instruction, or allowing some registers that are never used at the same time to be shared by multiple applications. For example, the test mode counters could be modified into one structure rather than one for each mode without too many structural changes.

C. LAYOUT

(this 45nm cannot be fabricated)

The 45 nm architecture used for the layout design is not a physical architecture and cannot be fabricated. If a physical design is desired in the future it must be done using a different process like GlobalFoundaries 180 nm which requires signing a nondisclosure agreement. Another approach would be to use open-source tools to create a Skywater 130 nm design. Both options would be too time-consuming to adapt to this late in the project but provide options for the project to continue in the future.

D. ADDITIONAL FEATURES

Time constraints also lead to the inability to add certain features to the interface that would be nice to have for further testing. For example, the interface can only handle 128 bits of AD, plaintext, or ciphertext, the functionality could be modified to allow the user to load more information to the interface and return a longer output in the middle of the core operation. This would require a major overhaul of both the core and interface FSMs and would result in much more complicated operating instructions. The FSMs would be required to handshake with each

other more often increasing complexity and chances of metatstability. It would solve some of the limitations of the test modes but would limit the operating speed of the core to how fast serial information could be loaded.

VII. BIBLIOGRAPHY

[1] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, “Ascon v1.2. Submission to NIST.” Graz University of Technology, May 31, 2021. Accessed: Oct. 07, 2023. [Online]. Available: https://ascon.iaik.tugraz.at/files/asconv12-nist.pdf

[2] NIST, “Cryptography,” National Institute of Standards and Technology. Accessed: Feb. 19, 2024. [Online]. Available: https://www.nist.gov/cryptography

[3] K. Stine and Q. Dang, “Encryption Basics,” *J. AHIMA*, vol. 82, no. 5, pp. 44–47, May 2011.

[4] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, 1st ed. Boca Raton: CRC Press, 2018.

[5] “NIST Selects ‘Lightweight Cryptography’ Algorithms to Protect Small Devices,” NIST. Accessed: Sep. 05, 2023. [Online]. Available: https://www.nist.gov/news-events/news/2023/02/nist-selects-lightweight-cryptography-algorithms-protect-small-devices#:~:text=Lightweight%20cryptography%20is%20designed%20to,as%20for%20other%20miniature%20technologies.

[6] H. Gross, E. Wenger, C. Dobraunig, and C. Ehrenhofer, “Suit up! -- Made-to-Measure Hardware Implementations of ASCON,” in *2015 Euromicro Conference on Digital System Design*, Madeira, Portugal: IEEE, Aug. 2015, pp. 645–652. doi: 10.1109/DSD.2015.14.

[7] NIST, “NIST Issues First Call for ‘Lightweight Cryptography’ to Protect Small Electronics,” National Institute of Standards and Technology. Accessed: Sep. 19, 2023. [Online]. Available: https://www.nist.gov/news-events/news/2018/04/nist-issues-first-call-lightweight-cryptography-protect-small-electronics

[8] J.-P. Kaps, W. Diehl, M. Tempelmeier, E. Homsirikamol, and K. Gaj, “Hardware API for Lightweight Cryptography.” Jan. 31, 2022. Accessed: Sep. 10, 2023. [Online]. Available: https://cryptography.gmu.edu/athena/LWC/LWC_HW_API_v1_1.pdf

[9] R. Primas, “Verilog Hardware Design of Ascon v1.2,” GitHub. Accessed: Oct. 12, 2023. [Online]. Available: https://github.com/rprimas/ascon-verilog

[10] J. Mertic, *Open source projects - beyond code: a blueprint for scalable and sustainable open source projects*, 1st edition. Birmingham: Packt Publishing, 2023.

APPENDIX A: ASCON_SPI.SV

```
// Top spi control and testing interface for 32 bit ascon core intended
// for use with the ascon hardware design by Robert Primas
// available at https://github.com/rprimas/ascon-verilog
//
// Author: Trevor Drane
// Design Repository: https://github.com/twdrane/ascon-test-spi
// Designed as part of a WPI Major Qualifying Project

`timescale 1ns / 1ps

module ascon_spi (
    input logic interface_clk,
    input logic core_clk,
    input logic rst,
    input logic cs_n,
    input logic sdi,

    output logic valid,
    output logic sdo,
    output logic auth_fail,
    output logic trig
    );

    //parameters
    parameter logic TRUE = 1'b1;
    parameter logic FALSE = 1'b0;

    // core io
    logic [31:0]    key_w;
    logic           key_valid;
    logic           key_ready;
    logic [31:0]    bdi;
    logic           bdi_valid;
    logic           bdi_ready;
    logic [3:0]     bdi_type;
    logic           bdi_eot;
    logic           bdi_eoi;
    logic           decrypt;
    logic           hash;
    logic [31:0]    bdo;
    logic           bdo_valid;
    logic           bdo_ready;
    logic [3:0]     bdo_type;
    logic           bdo_eot;
    logic           auth;
    logic           auth_valid;
    logic           auth_ready;

    // core op wires
    logic           key_valid_next;
    logic           bdi_valid_next;
    logic           bdi_eot_next;
    logic           bdi_eoi_next;

    //////////////////
```

```
// sipo convert //
//////////////////

logic [31:0] d_parallel;
logic [31:0] shift_reg;
logic [31:0] sipo_next;
logic [4:0] n_shifts,shifts_next;
logic done;

always_ff @ (posedge interface_clk) begin : sipo1_seq
    if (rst) begin
        shift_reg <= 32'd0;
        n_shifts <= 5'd31;
    end
    else begin
        shift_reg <= sipo_next;
        n_shifts <= shifts_next;
    end
end

// only shift if cs_n is active
assign sipo_next = ~cs_n ? {shift_reg[30:0],sdi} : shift_reg;
assign shifts_next = ~cs_n ? n_shifts + 1 : n_shifts;
assign done = ~cs_n & (n_shifts == 5'd31);
// end sipo convert

// create sipo output
always_ff @ (posedge interface_clk) begin : sipo2_seq
    if (rst) d_parallel <= 32'b0;
    else d_parallel <= (done) ? shift_reg : d_parallel;
end

////////////////////////////
// FSM enumerated types //
////////////////////////////

// spi encryption interface state machine
typedef enum bit [4:0] {
    I_IDLE,
    I_INST,
    I_READ_INST,
    I_DATA,
    I_READ_DATA,
    I_CORE,
    I_RETURN_SYNC,
    I_RETURN_PTCT,
    I_RETURN_BREAK,
    I_RETURN_TAG,
    I_RETURN_HASH,
    I_DONE
} spi_fsm;

spi_fsm current_state;
spi_fsm next_state;

typedef enum bit [4:0] {
    C_IDLE,
```

```
        C_LOAD_KEY,
        C_LOAD_NONCE,
        C_LOAD_AD,
        C_LOAD_PTCT,
        C_LOAD_TAG,
        C_RUN,
        C_WAIT_OUT,
        C_DONE
    } core_fsm;

    core_fsm core_state;
    core_fsm next_core_state;

    logic rpt_done,vml_done;

    logic [3:0]  inst;
    logic [3:0]  next_data_type,data_type;
    logic        read_done;
    logic [2:0]  read_count,data_size,size_calc;

    // main data registers
    logic [31:0] interface_config; //configure interface settings and run
cycles
    logic [31:0] core_config; //encrypt/decrypt/hash
    logic [2:0]  key_size;
    logic [2:0]  nonce_size;
    logic [2:0]  ad_size;
    logic [2:0]  ptct_size;
    logic [2:0]  tag_size;
    // main data registers

    // main data wires
    logic [3:0] interface_mode;
    assign interface_mode = interface_config[31:28];
    logic [3:0] core_mode;
    assign core_mode = core_config[31:28];

    // load control values
    logic [2:0]  load_count,load_count_next;
    logic        load_done;

    // return data control registers
    logic [3:0] reg_count;
    logic [4:0] piso_count;

    assign inst = d_parallel[31:28];
    assign size_calc = (d_parallel[1:0] != 2'b0) ? d_parallel[5:2] :
d_parallel[5:2]-1; // calculate size with rounding up
    assign read_done = (current_state == I_READ_DATA) & (read_count ==
data_size); // defines finishing read and instruction count;

    logic return_done;
    logic hash_done;

    assign return_done = ((reg_count == 4'd3 & current_state == I_RETURN_TAG)
| (reg_count == ptct_size & current_state == I_RETURN_PTCT)) & (piso_count ==
5'd31);
```

```
    assign hash_done = (reg_count == 4'd7) & (piso_count == 5'd31);

    logic auth_check;

    assign auth_check = auth_valid && auth;

    // auth fail pin
    // set auth fail flag
    always_ff @(posedge core_clk) begin : auth_fail_seq
        if (rst == TRUE) begin
            auth_fail <= FALSE;
        end
        else begin
            if (auth_ready == TRUE)
                auth_fail <= (auth_valid) ? ~auth : auth_fail;
        end
    end

    ////////////////////////////////
    //    _______ _____ __   __    //
    //   |  _____/  ____|  \/  |   //
    //   | |__ | (____ | \  / |   //
    //   |  __| \____ \| |\/| |   //
    //   | |    _____) | |  | |   //
    //   |_|   |______/|_|  |_|   //
    //                            //
    ////////////////////////////////

    // clock transfer buffers
    spi_fsm ifsm_buf_1,ifsm_buf_2;
    core_fsm cfsm_buf_1,cfsm_buf_2;

    // next state logic
    always_comb begin : interface_state_comb
        case (current_state)
            I_IDLE: begin
                next_state = (cs_n === 1'b0) ? I_INST : I_IDLE;
            end
            I_INST: begin
                next_state = done === TRUE ? I_READ_INST : I_INST;
            end
            I_READ_INST: begin
                case (inst)
                    // configure core
                    OP_DO_ENC,OP_DO_DEC,OP_DO_HASH:
                    begin
                        next_state = I_INST;
                    end
                    // load data

OP_LD_KEY,OP_LD_NONCE,OP_LD_AD,OP_LD_PT,OP_LD_CT,OP_LD_TAG:
                    begin
                        next_state = I_DATA;
                    end
                    // configure interface
                    OP_INT_SINGLE,OP_INT_RPT,OP_INT_VML:
                    begin
```

```
                        next_state = I_CORE;
                    end
                    default: next_state = I_INST;
                endcase
            end
            I_DATA: begin
                next_state = done === TRUE ? I_READ_DATA : I_DATA;
            end
            I_READ_DATA: begin
                next_state = read_done === TRUE ? I_INST : I_DATA;
            end
            // pass control to core fsm
            I_CORE: begin
                next_state = cfsm_buf_2 == C_DONE ? I_RETURN_SYNC : I_CORE;
            end
            I_RETURN_SYNC: begin // extra clock cycle to assign sipo reg
                next_state = core_mode === OP_DO_HASH ? I_RETURN_HASH :
I_RETURN_PTCT;
            end
            I_RETURN_PTCT: begin
                if (core_mode == OP_DO_ENC)
                    next_state = return_done === TRUE ? I_RETURN_BREAK :
I_RETURN_PTCT;
                else
                    next_state = return_done === TRUE ? I_DONE :
I_RETURN_PTCT;
            end
            I_RETURN_BREAK: begin
                next_state = I_RETURN_TAG;
            end
            I_RETURN_TAG: begin
                next_state = return_done === TRUE ? I_DONE : I_RETURN_TAG;
            end
            I_RETURN_HASH: begin
                next_state = hash_done === TRUE ? I_DONE : I_RETURN_HASH;
            end
            I_DONE: begin
                // resets only after cs is deasserted
                next_state = cs_n === TRUE ? I_IDLE : I_DONE;
            end
            default: next_state = I_IDLE;
        endcase
    end

    // assign next state and reset case
    always_ff @ (posedge interface_clk) begin : interface_state_seq
        if (rst) begin
            current_state <= I_IDLE;
            cfsm_buf_1 <= C_IDLE;
            cfsm_buf_2 <= C_IDLE;
        end
        else begin
            current_state <= next_state;
            cfsm_buf_1 <= core_state;
            cfsm_buf_2 <= cfsm_buf_1;
        end
    end
```

```
    // end SPI FSM

    // core next state logic
    always_comb begin : core_state_comb
        next_core_state = C_IDLE;
        case (core_state)
            C_IDLE: begin
                if (ifsm_buf_2 == I_CORE) begin
                    // check for hashing
                    next_core_state = core_mode === OP_DO_HASH ? C_LOAD_AD :
C_LOAD_KEY;
                end
                else next_core_state = C_IDLE;
            end
            C_LOAD_KEY: begin
                next_core_state = load_done === TRUE ? C_LOAD_NONCE :
C_LOAD_KEY;
            end
            C_LOAD_NONCE: begin
                next_core_state = load_done === TRUE ? C_LOAD_AD :
C_LOAD_NONCE;
            end
            C_LOAD_AD: begin
                if (load_done) begin
                    // check core operation for hashing
                    if (core_mode == OP_DO_HASH) begin
                        // check op for vml
                        if (interface_mode == OP_INT_VML) begin
                            if (vml_done) begin
                                next_core_state = C_WAIT_OUT;
                            end else next_core_state = C_LOAD_AD;
                        end else next_core_state = C_WAIT_OUT;
                    end else next_core_state = C_LOAD_PTCT;
                end else next_core_state = C_LOAD_AD;
            end
            C_LOAD_PTCT: begin
                if (load_done) begin
                    if (interface_mode == OP_INT_VML) begin
                        if (vml_done) begin
                            case (core_mode)
                                OP_DO_ENC: next_core_state = C_WAIT_OUT;
                                OP_DO_DEC: next_core_state = C_LOAD_TAG;
                                default: next_core_state = C_WAIT_OUT;
                            endcase
                        end
                    end
                    else begin
                        // check core operation
                        case (core_mode)
                            OP_DO_ENC: next_core_state = C_WAIT_OUT;
                            OP_DO_DEC: next_core_state = C_LOAD_TAG;
                            default: next_core_state = C_WAIT_OUT;
                        endcase
                    end
                end
                else next_core_state = C_LOAD_PTCT;
            end
```

```
            C_LOAD_TAG: begin
                if (load_done) begin
                    // check interface mode
                    next_core_state = C_WAIT_OUT;
                end
                else next_core_state = C_LOAD_TAG;
            end
            C_RUN: begin
                if (interface_mode == OP_INT_RPT)begin
                    if (rpt_done) begin
                        next_core_state = C_DONE;
                    end
                    else begin
                        case (core_mode)
                            OP_DO_ENC,OP_DO_DEC: next_core_state =
C_LOAD_NONCE;
                            OP_DO_HASH: next_core_state = C_LOAD_AD;
                            default: next_core_state = C_LOAD_NONCE;
                        endcase
                    end
                end
                else next_core_state = C_DONE;
            end
            C_WAIT_OUT: begin // wait for the tag, auth, or hash message to
be calculated
                // check interface mode
                if (core_mode == OP_DO_DEC) begin
                    // check auth
                    next_core_state = auth_check === TRUE ? C_RUN :
C_WAIT_OUT;
                end else begin
                    // hashing and enc
                    // wait for end of bdo
                    next_core_state = bdo_eot === TRUE ? C_RUN : C_WAIT_OUT;
                end
            end
            C_DONE: begin
                next_core_state = (ifsm_buf_2 == I_CORE) ? C_DONE : C_IDLE;
            end
            default: next_core_state = C_IDLE;
        endcase
    end

    // core next state
    always_ff @ (posedge core_clk) begin : core_state_seq
        if (rst) begin
            core_state <= C_IDLE;
            ifsm_buf_1 <= I_IDLE;
            ifsm_buf_2 <= I_IDLE;
        end
        else begin
            core_state <= next_core_state;
            ifsm_buf_1 <= current_state;
            ifsm_buf_2 <= ifsm_buf_1;
        end
    end
    // end core fsm
```

```
    ////////////////////////////////
    // interface mode ctrl logic //
    ////////////////////////////////

    logic [15:0] interface_count;
    assign interface_count = interface_config[15:0];

    // OP_INT_RPT control
    logic [31:0] rpt_counter;
    assign rpt_done = (interface_mode == OP_INT_RPT) & (rpt_counter[31:16] >=
interface_count);

    always_ff @ (posedge core_clk) begin : rpt_ctl
        if (rst) begin
            rpt_counter <= 32'b0;
        end
        else begin
            if ((interface_mode == OP_INT_RPT) & (core_state == C_RUN)) begin
                rpt_counter <= rpt_counter + 1;
            end
        end
    end

    // OP_INT_VML control
    logic [31:0] vml_counter;
    assign vml_done = (interface_mode == OP_INT_VML) & (vml_counter >=
interface_count);

    always_ff @ (posedge core_clk) begin : vml_ctl
        if (rst) begin
            vml_counter <= 32'b0;
        end
        else begin
            if ((interface_mode == OP_INT_VML) & ((core_state == C_LOAD_AD &
core_mode == OP_DO_HASH) | (core_state == C_LOAD_PTCT)) & load_done) begin
                vml_counter <= vml_counter + 1;
            end
        end
    end
    // end interface mode ctrl logic

    ////////////////////////////////
    // memory and config logic //
    ////////////////////////////////

    // inputs
    // encrypt
    logic [31:0] key        [3:0];
    logic [31:0] nonce      [3:0];
    logic [31:0] a_data     [3:0];
    logic [31:0] ptct       [3:0];
    // decrypt
    logic [31:0] tag        [3:0];

    // outputs
    logic [31:0] ptct_out   [3:0];
```

```
    logic [31:0] tag_out    [3:0];

    logic [31:0] hash_out   [7:0];

    // data type logic
    always_comb begin : data_type_comb
        // determine data type from instruction
        if (current_state == I_READ_INST) begin
            case (inst)
                OP_LD_KEY: next_data_type = D_KEY;
                OP_LD_NONCE: next_data_type = D_NONCE;
                OP_LD_AD: next_data_type = D_AD;
                OP_LD_PT: next_data_type = D_PTCT;
                OP_LD_CT: next_data_type = D_PTCT;
                OP_LD_TAG: next_data_type = D_TAG;
                default: next_data_type = D_NULL;
            endcase
        end else next_data_type = data_type;
        // end read instructions
    end

    // syncronized assignments
    always_ff @ (posedge interface_clk) begin : memory_config
        // define resets
        if (rst) begin
            core_config <= 32'b0;
            interface_config <= 32'b0;
            data_size   <= 3'b0;
            data_type   <= 4'b0;
            read_count  <= 3'b0;
            key[0]      <= 32'b0; key[1] <= 32'b0; key[2] <= 32'b0; key[3] <=
32'b0;
            nonce[0]    <= 32'b0; nonce[1] <= 32'b0; nonce[2] <= 32'b0;
nonce[3] <= 32'b0;
            a_data[0]   <= 32'b0; a_data[1] <= 32'b0; a_data[2] <= 32'b0;
a_data[3] <= 32'b0;
            ptct[0]     <= 32'b0; ptct[1] <= 32'b0; ptct[2] <= 32'b0; ptct[3]
<= 32'b0;
            tag[0]      <= 32'b0; tag[1] <= 32'b0; tag[2] <= 32'b0; tag[3] <=
32'b0;
            key_size    <= 3'b0;
            nonce_size  <= 3'b0;
            ad_size     <= 3'b0;
            ptct_size   <= 3'b0;
            tag_size    <= 3'b0;
        end else begin
            // update instructions
            core_config <= ((inst == OP_DO_ENC | inst == OP_DO_DEC | inst ==
OP_DO_HASH) & current_state == I_READ_INST) ? d_parallel : core_config;
            interface_config <= ((inst == OP_INT_SINGLE | inst == OP_INT_RPT
| inst == OP_INT_VML) & current_state == I_READ_INST) ? d_parallel :
interface_config;
            data_size   <= (current_state == I_READ_INST) ? size_calc :
data_size;
            data_type   <= next_data_type;
            // end update instructions
```

```
            // update data size
            key_size    <= (data_type == D_KEY)     ? data_size : key_size;
            nonce_size  <= (data_type == D_NONCE)   ? data_size : nonce_size;
            ad_size     <= (data_type == D_AD)      ? data_size : ad_size;
            ptct_size   <= (data_type == D_PTCT)    ? data_size : ptct_size;
            tag_size    <= (data_type == D_TAG)     ? data_size : tag_size;
            // end update size
            // update data
            //count reads
            if (current_state == I_READ_DATA) begin
                read_count <= read_count + 1;
            end
            else if (current_state == I_DATA)
                read_count <= read_count;
            else read_count <= 3'b0;
            //save data to registers
            key[read_count]     <= (data_type == D_KEY & current_state ==
I_READ_DATA)  ? d_parallel : key[read_count];
            nonce[read_count]   <= (data_type == D_NONCE & current_state ==
I_READ_DATA)    ? d_parallel : nonce[read_count];
            a_data[read_count]  <= (data_type == D_AD & current_state ==
I_READ_DATA)       ? d_parallel : a_data[read_count];
            ptct[read_count]    <= (data_type == D_PTCT & current_state ==
I_READ_DATA)     ? d_parallel : ptct[read_count];
            tag[read_count]     <= (data_type == D_TAG & current_state ==
I_READ_DATA)  ? d_parallel : tag[read_count];
            // end update data
        end
    end
    // end memory and config


    //////////////////////
    // core load logic //
    //////////////////////

    // define load_done
    always_comb begin : load_done_comb
        case (core_state)
            C_LOAD_KEY:     load_done = key_ready & (load_count == key_size);
            C_LOAD_NONCE:   load_done = bdi_ready & (load_count ==
nonce_size);
            C_LOAD_AD:      load_done = bdi_ready & (load_count == ad_size);
            C_LOAD_PTCT:    load_done = bdi_ready & (load_count ==
ptct_size);
            C_LOAD_TAG:     load_done = bdi_ready & (load_count == tag_size);
            default: load_done = FALSE;
        endcase
    end

    always_comb begin : core_load_comb
        if (load_done) load_count_next = 3'b0;
        else if ((bdi_ready & bdi_valid) | (key_ready & key_valid))
load_count_next = load_count + 1;
        else load_count_next = load_count;
        key_w = 32'b0;
        key_valid_next = FALSE;
```

```
        bdi = 32'b0;
        bdi_valid_next = FALSE;
        bdi_type = D_NULL;
        bdi_eot_next = FALSE;
        bdi_eoi_next = FALSE;
        case (core_state)
            C_LOAD_KEY: begin
                key_w           = key[load_count];
                key_valid_next  = ~load_done;
            end
            C_LOAD_AD: begin
                bdi             = a_data[load_count];
                bdi_type        = D_AD;
                bdi_valid_next  = ~load_done;
                bdi_eot_next    = ((ad_size == load_count_next) & bdi_ready &
((core_mode != OP_DO_HASH) | (interface_mode != OP_INT_VML) | vml_done));
                bdi_eoi_next    = ((ad_size == load_count_next) & bdi_ready &
(core_mode == OP_DO_HASH) & ((interface_mode != OP_INT_VML) | vml_done));
            end
            C_LOAD_NONCE: begin
                bdi             = nonce[load_count];
                bdi_type        = D_NONCE;
                bdi_valid_next  = ~load_done;
                bdi_eot_next    = (nonce_size == load_count_next);
            end
            C_LOAD_PTCT: begin
                bdi             = ptct[load_count];
                bdi_type        = D_PTCT;
                bdi_valid_next  = ~load_done;
                bdi_eot_next    = ((ptct_size == load_count_next) & bdi_ready
& ((interface_mode != OP_INT_VML) | vml_done));
                bdi_eoi_next    = ((ptct_size == load_count_next) & bdi_ready
& ((interface_mode != OP_INT_VML) | vml_done));
            end
            C_LOAD_TAG: begin
                bdi             = tag[load_count];
                bdi_type        = D_TAG;
                bdi_valid_next  = ~load_done;
                bdi_eot_next    = (tag_size == load_count_next);
            end
            default: begin
                bdi             = 32'b0;
                bdi_type        = D_NULL;
                bdi_valid_next  = FALSE;
            end
        endcase
    end

    // sychronous state logic
    always_ff @(posedge core_clk) begin : core_load_seq
        if (rst == TRUE) begin
            load_count  <= 3'b0;
            key_valid   <= FALSE;
        end else begin
            load_count  <= load_count_next;
            key_valid   <= key_valid_next;
            bdi_valid   <= bdi_valid_next;
```

```
            bdi_eot     <= bdi_eot_next;
            bdi_eoi     <= bdi_eoi_next;
        end
    end
    //end core load logic


    //////////////////////
    // core read logic //
    //////////////////////

    logic [4:0] core_out_count;

    always_ff @(posedge core_clk) begin : core_read
        if (rst) begin
            core_out_count <= 5'b0;
            ptct_out[0] <= 32'b0; ptct_out[1] <= 32'b0; ptct_out[2] <= 32'b0;
ptct_out[3] <= 32'b0;
            tag_out[0]  <= 32'b0; tag_out[1]  <= 32'b0; tag_out[2]  <= 32'b0;
tag_out[3]  <= 32'b0;
            hash_out[0] <= 32'b0; hash_out[1] <= 32'b0; hash_out[2] <= 32'b0;
hash_out[3] <= 32'b0;
            hash_out[4] <= 32'b0; hash_out[5] <= 32'b0; hash_out[6] <= 32'b0;
hash_out[7] <= 32'b0;
        end else begin
            if (bdo_eot)
                core_out_count <= 5'b0;
            else if (bdo_valid && bdo_ready)
                core_out_count <= core_out_count + 1;
            else
                core_out_count <= core_out_count;
            ptct_out[load_count]     <= (bdo_type == D_PTCT & bdo_valid) ?
bdo : ptct_out[load_count];
            tag_out[core_out_count]  <= (bdo_type == D_TAG & bdo_valid) ? bdo
: tag_out[core_out_count];
            hash_out[core_out_count] <= (bdo_type == D_HASH & bdo_valid) ?
bdo : hash_out[core_out_count];
        end

    end
    // end core read logic


    ///////////////////
    // piso convert //
    ///////////////////

    logic [31:0] piso,piso_next;
    logic [3:0] reg_count_next;
    logic [4:0] piso_count_next;

    assign sdo = valid ? piso[31] : 1'b0;
    assign valid = (current_state == I_RETURN_PTCT | current_state ==
I_RETURN_TAG | current_state == I_RETURN_HASH);

    always_comb begin : piso_comb
        piso_next = piso;
```

```
        reg_count_next = reg_count;
        piso_count_next = piso_count;
        if (current_state == I_RETURN_SYNC | current_state == I_RETURN_BREAK)
begin
            case (next_state)
                I_RETURN_PTCT: piso_next = ptct_out[0];
                I_RETURN_TAG: piso_next = tag_out[0];
                I_RETURN_HASH: piso_next = hash_out[0];
                default: piso_next = 32'b0;
            endcase
        end
        else if (valid) begin
            if (piso_count == 5'd31) begin
                if (current_state == I_RETURN_HASH) begin
                    if (reg_count == 3'd7) begin
                        reg_count_next = 4'b0;
                        piso_next = hash_out[0];
                    end
                    else begin
                        reg_count_next = reg_count + 1;
                        piso_next = hash_out[reg_count+1];
                    end
                end
                else begin
                    if (return_done) begin
                        reg_count_next = 4'b0;
                    end
                    else begin
                        reg_count_next = reg_count + 4'b1;
                        case (next_state)
                            I_RETURN_PTCT: piso_next = ptct_out[reg_count+1];
                            I_RETURN_TAG: piso_next = tag_out[reg_count+1];
                            default: piso_next = 32'b0;
                        endcase
                    end
                end
                piso_count_next = 5'b00000;
            end
            // shift
            else begin
                piso_next = {piso[30:0],1'b0};
                piso_count_next = valid ? piso_count + 5'b1 : piso_count;
            end
        end
        else begin
            reg_count_next = 4'b0;
            piso_count_next = 5'b00000;
        end
    end

    always_ff @(posedge interface_clk) begin : piso_seq
        if (rst == TRUE) begin
            piso <= 32'b0;
            reg_count <= 4'b0;
            piso_count <= 5'b00000;
        end
        else begin
```

```
            piso <= piso_next;
            reg_count <= reg_count_next;
            piso_count <= piso_count_next;
        end
    end
    // end piso convert

    // define config flags
    assign hash = (core_mode == OP_DO_HASH);
    assign decrypt = (core_mode == OP_DO_DEC);

    // define read flags
    assign bdo_ready = (core_state == C_LOAD_PTCT | (core_state == C_WAIT_OUT
& (core_mode == OP_DO_ENC | core_mode == OP_DO_HASH)));
    assign auth_ready = (core_state == C_WAIT_OUT) & (core_mode ==
OP_DO_DEC);

    // instantiate the spi interface and convert for the ascon core module
    ascon_core core(
        .clk(core_clk),          //i
        .rst(rst),               //i
        .key(key_w),             //i 32
        .key_valid(key_valid),   //i
        .key_ready(key_ready),   //o
        .bdi(bdi),               //i 32
        .bdi_valid(bdi_valid),   //i
        .bdi_ready(bdi_ready),   //o
        .bdi_type(bdi_type),     //i 4
        .bdi_eot(bdi_eot),       //i
        .bdi_eoi(bdi_eoi),       //i
        .decrypt(decrypt),       //i
        .hash(hash),             //i
        .bdo(bdo),               //o 32
        .bdo_valid(bdo_valid),   //o
        .bdo_ready(bdo_ready),   //i
        .bdo_type(bdo_type),     //o 4
        .bdo_eot(bdo_eot),       //o
        .auth(auth),             //o
        .auth_valid(auth_valid),//o
        .auth_ready(auth_ready),//i
        .trig(trig)              //o
    );

endmodule // ascon_spi
```

APPENDIX B: INTERFACE_TEST.SV

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//////
// Company: WPI
// Engineer: Trevor Drane
//
// Create Date: 10/25/2023 02:43:11 PM
// Design Name: Ascon SPI
// Module Name: interface_test
// Project Name:
// Target Devices: Basys 3
// Tool Versions:
// Description: Simulation source for interface testing
//
// Dependencies: sipo.sv
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////


module interface_test(

    );
    logic interface_clk;
    logic core_clk;
    //logic clk;
    logic rst;
    logic cs_n;
    logic sdi;
    logic valid;
    logic sdo;
    logic auth_fail;
    logic [31:0] in;
    logic [31:0] out;
    logic trig;


    int tv_num;
    string tv_path;
    string TV_SET1 = "../tv/set1/tv.txt"; // long set
    string TV_SET2 = "../tv/set2/tv.txt"; // short set
    string TV_SET3 = "../tv/set3/tv.txt"; // rpt
    string TV_SET4 = "../tv/set4/tv.txt"; // vml
    string TV_SET5 = "../tv/set5/tv.txt"; //
    int arg_status = 0;
    //string CHECK_FILE = "../tv/set1/check_d.txt";
    int fvectors,fcheck;
    int SIM_T = 60000; // 60 ms for one round of hashing
    string line;
    string hdr,ignore;
    int cnt;
```

```
logic [31:0] expected;
logic verify;


always #10 interface_clk = ~interface_clk;
always #5 core_clk = ~core_clk;
//always #5 clk = ~clk;

initial begin
    verify = 1;
    $dumpfile("trace.vcd");
    $dumpvars(0, interface_test);
    //fcheck = $fopen (CHECK_FILE, "r");
    arg_status = $value$plusargs("TV_SET=%d",tv_num);
    if (arg_status) begin
        case (tv_num)
            1: tv_path = TV_SET1;
            2: tv_path = TV_SET2;
            3: begin tv_path = TV_SET3;
                SIM_T = 150_000_000;
            end
            4: tv_path = TV_SET4;
            5: tv_path = TV_SET5;
            default: tv_path = TV_SET1;
        endcase
        fvectors = $fopen (tv_path, "r");
    end
    else fvectors = $fopen (TV_SET1, "r");
    if (fvectors == 0) begin
       $display("Could not open test vector file");
       $finish;
    end
    rst = 1'b1;
    core_clk = 1'b0;
    interface_clk = 1'b0;
    cs_n = 1'b1;
    sdi = 1'b0;
    @(posedge interface_clk);
    @(negedge interface_clk) rst = 1'b0;
    @(negedge interface_clk) cs_n = 1'b0;
    while (!$feof(fvectors)) begin
        void'($fgets(line, fvectors));
        void'($sscanf(line, "%s", hdr));
        if (hdr == "WAIT") begin
            fork
                begin
                    wait (auth_fail);
                    $display("AUTH FAIL @ %0t", $time);
                    @(negedge interface_clk) rst = 1'b1;
                    @(negedge interface_clk) rst = 1'b0;
                end
                begin
                    wait (valid);
                    wait (~valid);
                    @(posedge interface_clk);
                    @(posedge interface_clk);
```

```
                    wait (~valid);
                end
            join_any
            @(negedge interface_clk) rst = 1'b1;
            @(negedge interface_clk) rst = 1'b0;
            $display("timestamp: %0t", $time);
            continue;
        end
        if (hdr == "STOP") continue;
        if (hdr == "#") void'($fgets(line, fvectors));
        void'($sscanf(line, "%s %h", hdr, in));
        //if (hdr == "STOP") $stop;
        cnt = 31;
        repeat(32) begin
            sdi = in[cnt];
            @(negedge interface_clk);
            cnt = cnt - 1;
        end
    end
    $fclose(fvectors);
end

always #SIM_T $stop;

sipo sipo (
    .clk(interface_clk),
    .cs(valid),
    .sdi(sdo),
    .pdo(out)
);

always @(out) begin
    //void'($fscanf(fcheck, "%h", expected));
    //verify = (expected==out) && verify;
    $display("%h", out);
    // if ($feof(fcheck)) begin
    //  if (verify == 1'b1)
    //      $display("Testbench Passed");
    //  else
    //      $display("Testbench Failed");
    // end
end

ascon_spi dut (
    .interface_clk(interface_clk),
    .core_clk(core_clk),
    //.clk(clk),
    .rst(rst),
    .cs_n(cs_n),
    .sdi(sdi),
    .valid(valid),
    .sdo(sdo),
    .auth_fail(auth_fail),
    .trig(trig)
);

endmodule
```

APPENDIX C: SAMPLE TEST VECTOR SET

```
# Load key
INS 30000010
DAT 9D79B1A3
DAT 7F31801C
DAT D11A6706
DAT FB40D6BD
# Specify authenticated encryption
INS 00000000
# Load nonce
INS 40000010
DAT 57526846
DAT 903BB13E
DAT DE562439
DAT E9C1B823
# Load associated data
INS 50000010
DAT 1AB3C589
DAT E3E64EC6
DAT 1F7EC67B
DAT F7017780
# Load plaintext
INS 61000010
DAT 4FCF816F
DAT B65763D3
DAT A38824BB
DAT 6AAC9780
# Interface instruction
INS 90000000
WAIT
# Load key
INS 30000010
DAT 9D79B1A3
DAT 7F31801C
DAT D11A6706
DAT FB40D6BD
# Specify authenticated decryption
INS 10000000
# Load nonce
INS 40000010
DAT 57526846
DAT 903BB13E
DAT DE562439
DAT E9C1B823
# Load associated data
INS 50000010
DAT 1AB3C589
DAT E3E64EC6
DAT 1F7EC67B
DAT F7017780
# Load ciphertext
INS 71000010
DAT 13E3AD30
DAT AB2075C8
DAT 44E2D23F
DAT 15A86348
```

```
# Load tag
INS 81000010
DAT 31639907
DAT F2D82FD7
DAT 58725E70
DAT 6CE3C1F7
# Interface instruction
INS 90000000
WAIT
# Specify hashing
INS 20000000
# Load message data
INS 51000010
DAT 1AB3C589
DAT E3E64EC6
DAT 1F7EC67B
DAT F7017780
# Interface instruction
INS 90000000
WAIT
STOP
```