5G Lattice ORAN Solution Stack 1.0

Reference Design

FPGA-RD-02257-1.0

June 2022

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, with all faults and associated risk the responsibility entirely of the 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
AHBL	Advanced High-performance Bus-Lite
APB	Advanced Peripheral Bus
AXI	Advanced eXtensible Interface
CPU	Central Processing Unit
DMA	Direct Memory Access
ECC	Elliptical Curve Cryptography
FIFO	First-In-First-Out
HMAC	Hash Message Authentication Code
IRQ	Interrupt Request
OSE	ORAN Security Enclave
PCIe	Peripheral Component Interconnect Express
PKC	Public Key Cryptography
RISC-V	Reduced Instruction Set Computer-V
RSA	Rivest–Shamir–Adleman
RTL	Register-Transfer Level
SHA384	Secure Hash Algorithm
SMBus	System Management Bus
UART	Universal Asynchronous Receiver-Transmitter

1. Introduction

This document provides the overall design flow of the 5G Lattice ORAN™ Solution Stack 1.0 reference design. In this project, the CertusPro™-NX is used with soft IPs to support fast packet encryption/decryption using either AES-CBC/GCM mode.

Support for packet authentication is provided by ECC384, HMAC384 or RSA 3K/4K over PCIe Interface. Keys and other required configuration are set up through SMBus as a part of sideband communication.

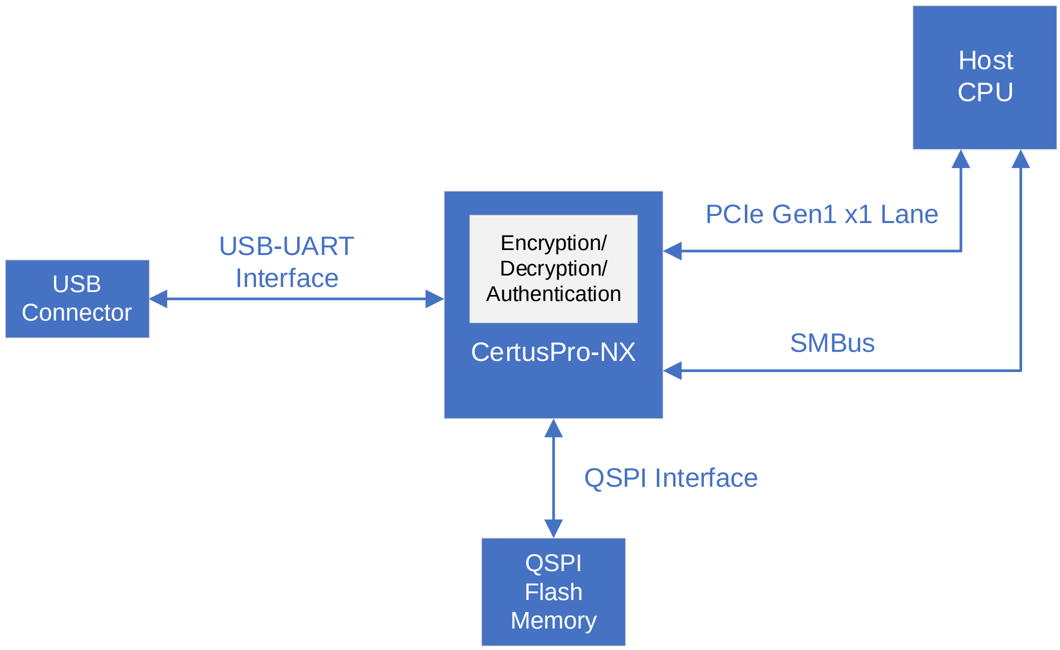


Figure 1.1. Block Diagram

Figure 1.1 shows the system block diagram.

CertusPro-NX is connected to the Host PC using PCIe x1 Endpoint IP. CertusPro-NX boots from the bitstreams stored at an external QSPI flash. It also connects to external components through SMBus. Furthermore, CertusPro-NX connects to an external USB through soft IP UART and an external converter UART to USB.

This document discusses the critical functions of various IP's and individual components. It show the integration with CertusPro-NX and provides detailed description of all crypto algorithm implementations in the FPGA.

The main functions of the 5G Lattice ORAN Solution Stack are:

- Packet authentication, encryption, and decryption between the Host CPU and CertusPro-NX over PCIe.
- Support of AES-256 CBC, AES-256 GCM, SHA384, HMAC384, RSA 3K/4K, ECDH crypto algorithms.
- Crypto-256 and Crypto-384 services to customers through software APIs.
- Support of SPDM protocol over MCTP.
- Support of secure out-of-band communication over SMBus.

2. Design Architecture

The design architecture consists of three subsystems:

- PCIe
- Application CPU
- Crypto

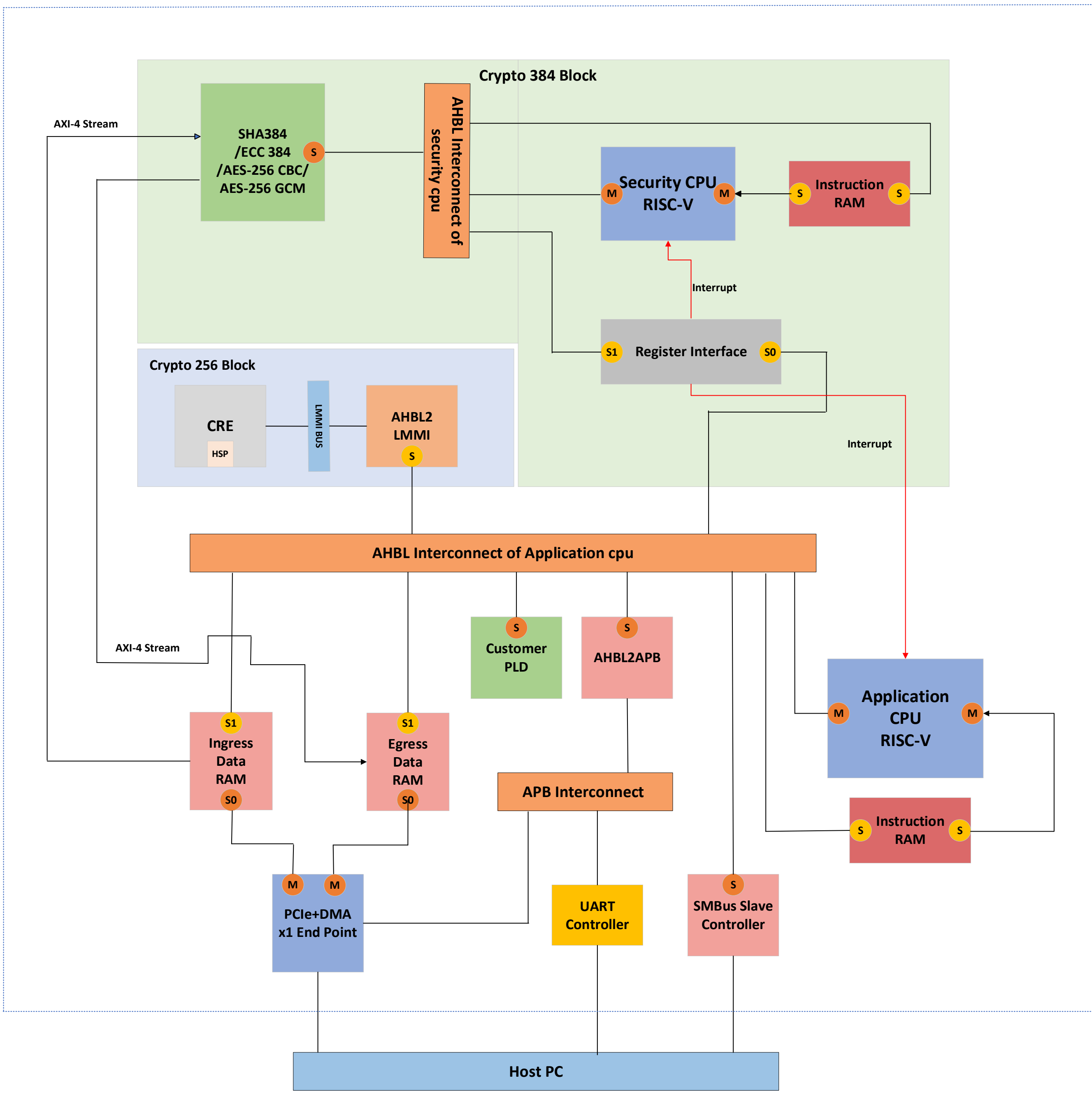


Figure 2.1. Top Level Architecture

Figure 2.1 shows the top level architecture of the 5G Lattice ORAN Solution Stack 1.0. The three main subsystems are described below.

2.1. PCIe Subsystem

The PCIe subsystem is built by the following IPs:

- PCIe Endpoint IP – This IP is configured with DMA and two AHBL Master interfaces and one APB interface.
- AHBL Interconnect of PCIe – It supports two AHBL Master and two AHBL Slave interfaces.
- Ingress RAM – It has two AHBL Slave Ports and one AXI stream Master Port. One of the ports, S0, is connected to the AHBL Interconnect of PCIe. The other port, S1, is connected to the AHBL Interconnect of Application.
- Egress RAM – It has two AHBL Slave Ports and one AXI stream Slave Port. One of the ports, S0, is connected to the AHBL Interconnect of PCIe. The other port, S1, is connected to the AHBL Interconnect of Application.

The Host PC initiates the transaction over PCIe Endpoint to CertusPro-NX. The PCIe Endpoint IP is configured with DMA and two AHBL Master and one AHBL slave, The DMA transfers the incoming packets to Ingress RAM from its AHBL Master Port to Slave Port S0 at Ingress RAM. Furthermore, the DMA transfers outgoing packets from Egress RAM to PCIe Endpoint and then to the Host CPU. The PCIe IP also has an APB interface, from which the Application CPU firmware can do some configurations and check its status. Both Ingress RAM and Egress RAM has a second AHBL Slave Port S1, from which the RISC-V CPU firmware can also control AXI4 Stream data transfer from the Ingress RAM to AES Encrypt/Decrypt IP block and from AES Encrypt/Decrypt IP block to Egress RAM. The AHBL port S1 can also be used for reading the data/printing the data on UART terminal in PCIe to UART flow and providing data in UART to PCIe Flow.

The PCIe subsystem is configured by the following modules:

- PCIe Endpoint IP
- AHBL Interconnect of PCIe
- Ingress RAM and Egress RAMs

2.1.1. PCIe Endpoint with DMA Enabled

The design is configured with two AHBL masters(one for reading and another for writing) and one APB slave interface for register configuration of PCIe IP, PCIe with DMA interface data flow block diagram as shown in Figure 2.2.

Brief description of each block is explained in the PCIe DMA section.

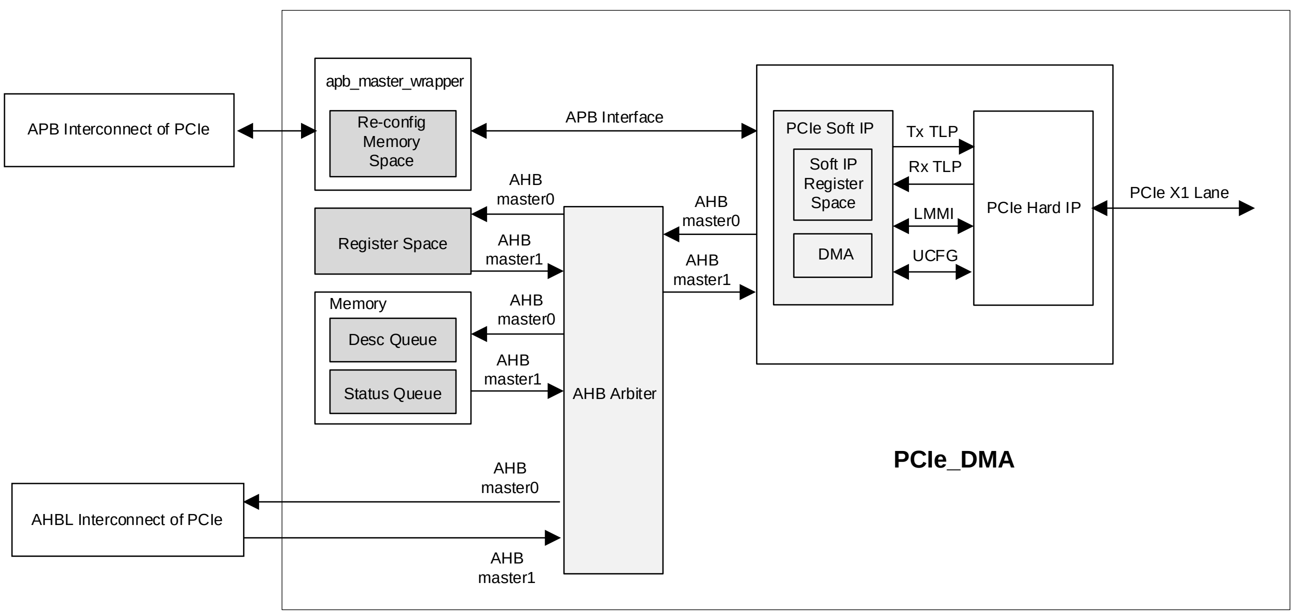


Figure 2.2. PCIe Endpoint with DMA Enabled

2.1.2. AHBL Interconnect of PCIe

The design supports two AHBL Master interfaces and two AHBL Slave interfaces to write and read from the Ingress RAM and the Egress RAM.

2.1.3. Ingress RAM

The design has two AHBL Slave Ports. One (S0) is connected to the AHBL interconnect of PCIe through which RX data is acquired from the PCIe module. The other (S1) is connected to the AHBL interconnect of the Application CPU through which the CPU can control/configure the ingress data RAM.

In addition, the AXI stream port is available for sending the incoming data to Crypto 384 block for encryption/decryption.

This has one True Dual Port RAM, two AHBL slaves and one AXI master. The base address for AHBL slaves and memory depth for True Dual Port RAM (for two ports) should be provided in a general manner. Data width for True Dual Port RAM are 64 width for Port A and 128 width for port B. The Port A of True Dual Port RAM is being accessed by S0_AHB_slave.The Port B of True Dual Port RAM is being accessed by AXI master and S1_AHB_slave.

S0_AHB Slave

This S0_AHB slave only operates on data.

Here, the slave stores data in True Dual Port RAM by mapping AHB S0 slave address with Port A of True Dual Port RAM based on AHB transactions.

Based upon ahb_write and ahb_trans, the slave writes data into True Dual Port RAM.

AXI_master

The three AXI signals, TDATA, TVALID, and TREADY are used for AXI transactions.

Here, the master provides data only when there is enough data entered from Port A of True Dual Port RAM and makes TVALID HIGH.

If TREADY is LOW, the master holds data and TVALID in same position.

If TREADY is HIGH, the transaction gets completed and master provides next data if it has enough data in True Dual Port RAM.

S1_AHB_slave

The S1_AHB_slave helps to transfer data and also used for control signals based on the address given. Data is being operated on addresses from (0x0000) - (0xEFFF).Here slave takes data from True Dual Port RAM by mapping AHB S1 slave address with Port B of True Dual Port RAM based on AHB transactions. This slave can also writes data in Port B of Tue dual port RAM.

Based upon ahb_write and ahb_trans, the slave writes and reads data in the True Dual Port RAM.

The controls are PRI on address from (0xF000).

Control Signals

(0xf008)

31:2	1:0
reserved	To indicate the AES flow direction and whether decryption or encryption is taking place. 0x1: for AES Decryption 0x2: for AES Encryption *Default*: 0x0
	Write only

(0xf00C)

31:1	0
reserved	To enable the AXI master *Default*: 0 (AXI master is disabled)
	Write only

(0xf01C)

31:1	0
reserved	To select which protocol type should be used for RAM Port B access.(whether AXI master or s1_AHB_slave) *Default*: 1 (AXI protocol)
	Write only

(0xf024)

31:1	0
reserved	To decide whether port b is for reading or writing *Default*: 0 (Reading)
	Write only

(0xf028)

31:0
Data size given by UART *Default*: 0x0
Write only

(0xf004)

31:0
READ ADDRESS of port B
Read only

(0xf010)

31:0
WRITE ADDRESS of port A
Read only

2.1.4. Egress RAM

The Egress RAM has two AHBL Slave Ports. One (S0) is connected to the AHBL interconnect of PCIe through which PCIe module reads the data from Egress RAM. The other (S1) is connected to the AHBL interconnect of the Application CPU through which the CPU controls/configures the Egress RAM.

In addition, the Egress RAM has an AXI stream port through which it receives the data from the Crypto 384 block after performing encryption/decryption. There is one True Dual Port RAM, two AHBL slaves, and one AXI slave. The base address for AHBL slaves and memory depth for True Dual Port RAM (for two ports) are provided. Data width for True Dual Port RAM is 64 for Port A and 128 for Port B. Port A of True Dual Port RAM is accessed by S0_AHBL_slave. Port B of is accessed by AXI master and S1_AHBL_slave.

S0_AHBL slave

The S0_AHBL slave only operates on data.

Here, the slave takes data from True Dual Port RAM by mapping AHB S0 slave address with Port A of True Dual Port RAM based on AHBL transactions.

Based upon ahb_write and ahb_trans, the slave takes data from True Dual Port RAM.

AXI_slave

The Three AXI signals, TDATA, TVALID, and TREADY are used for AXI transactions.

Here, slave takes data without any interrupt.

S1_AHBL_slave

The S1_AHBL_slave helps to transfer data and is also used for control signals based on the given address. Data is operated on addresses from (0x0000) to (0xEFFF). Here, slave stores data in True Dual Port RAM by mapping the AHB S1 slave address with Port B based on AHBL transactions. This slave can also read data from Port B.

Based upon ahbl_write and ahbl_trans, the slave writes and reads data in True Dual Port RAM.

The controls are given on address from (0xF000).

Control Signals

(0xf008)

31:2	1:0
reserved	To indicate the AES flow direction and whether decryption or encryption is taking place. 0x1: for AES Decryption 0x2: for AES Encryption *Default*: 0x0
	Write only

(0xf00C)

31:1	0
reserved	To Enable the AXI Slave *Default*: 0 (AXI slave is disabled)
	Write only

(0xf01C)

31:1	0
reserved	To select which protocol type should be used RAM Port B access. (whether AXI slave or s1_AHB_slave) *Default*: 1 (AXI protocol)
	Write only

(0xf020)

31:1	0
reserved	SHA data ready *Default*: 0 (data not ready)
	Write only

(0xf024)

31:1	0
reserved	To decide whether Port-B is for reading or writing. *Default*: 0 (Reading)
	Write only

(0xf028)

31:0
Data size given by UART *Default*: 0x0
Write only

(0xf004)

31:0
WRITE ADDRESS of port B
Read only

(0xf010)

31:0
READ ADDRESS of Port A
Read only

(0xf014)

31:0
DATA SIZE given by PCIe for AES
Read only

(0xf018)

31:0
Data ready signal which tells data is encrypted/decrypted *Default*: 0x0
Read only

(0xf044)

31:0
Performance counter for AES algorithm *Default*: 0x0
Read only

The two blocks, Ingress RAM and Egress RAM, are built from an existing IP. In Dual Port EBR memory, the writing port and the reading port need to be controlled using AHB Lite and AXI stream interfaces, as shown in Figure 2.3 and Figure 2.4.

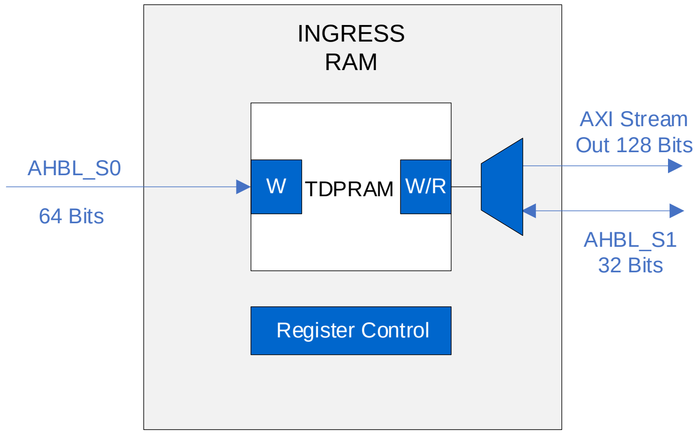


Figure 2.3. Ingress RAM with Mux Selection

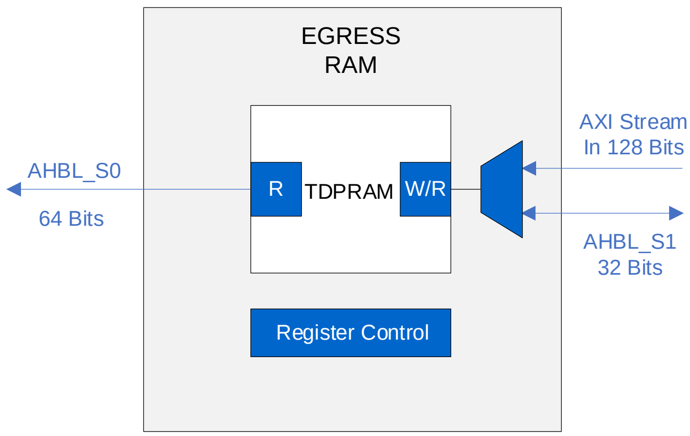


Figure 2.4. Egress RAM with Mux Selection

2.2. Application CPU Subsystem

The Application CPU is the main interface that controls the process flow to other submodules. It interacts with submodules over the AHBL interface. The Application CPU has two AHBL Master Ports. One is tasked to send/receive data from other interfaces, which are connected through AHBL interconnect. The other is connected to the Instruction RAM.

The Application CPU subsystem has the following sub-interfaces:

- Application CPU – It is a RISC-V processor with two AHBL Master Ports. One fetches instruction and the other fetches data.
- AHBL Interconnect of Application – It supports one AHBL Master Port and multiple Slave Ports. The Application CPU is the master. Data RAM, Register Interface module (Port S0), Ingress RAM (port S1), Egress RAM (port S1), AHBL2LMMI (connected to CRE/HSE), AHBL2APB, SMBus, and Slave Customer PLD are the slaves. The block AHBL2APB is connected to APB Interconnect, which connects to multiple slaves including SMBus Slave controller, UART controller.
- Instruction RAM – It has one AHBL Slave Port.
- Data RAM – It has one AHBL Slave Port.
- Customer PLD Logic

2.3. Crypto Subsystem

The design is divided into two sub systems one is Crypto-256 subsystem and others one is Crypto-384 subsystem.

2.3.1. Crypto-256 Subsystem

Crypto-256 Block has a CRE module and an AHB Lite to LMMI slave interface. The AHB Lite slave interface is connected to the AHB Lite Interconnect of Application. The Crypto-256 block also has a block of AHBL2LMMI to convert AHB Lite bus signals to LMMI signals since HSE uses LMMI interface. Crypto-256 block is used for generating the 256 bit Key Pair (Public and Private), generating 256 bit Public Key using Private Key for validation.

2.3.2. Crypto-384 Subsystem

- AHBL Interconnect of Security – It supports one AHBL Master Port and three AHBL Slave Ports. One Master Port is connected to the Security CPU and the Slave Ports are Crypto-Accelerators (SHA2-384/ECC-384/AES-CBC/AES-GCM), Register Interface (port S1), and Instruction RAM of the Security CPU.
- Instruction RAM – It has one AHBL Slave Port.
- Register Interface – It has two AHBL Slave Ports: one port S0 is connected to the AHBL Interconnect of the Application CPU, the other port S1 is connected to the AHBL Interconnect of the Security CPU.
- Crypto IPs – It includes accelerator for SHA2-384, PKC IP, and AES-CBC/AES-GCM. It has one AHBL Slave Port and two AXI4-Stream ports (one for stream in and one for stream out).
- Security RISC-V CPU – It includes one master AHBL interconnect of security interface, Instruction RAM register interface, crypto IPs.

2.3.3. Register Interface

The Register Interface block has two AHB Lite Slave Ports. One port is connected to the Security CPU AHB Lite Interconnect and the other one is Security RISC-V CPU base address 0x2C0000 is used. When accessing from the Security CPU through S1 Port of the Register Interface, the base address 0x2E0000 is used.

When the Application CPU requests any service from Crypto-384, it writes certain information to the Register Interface which then generates an interrupt to the Security CPU. The interrupt service routine at the Security CPU reads the information from the Register Interface and provides service such as SHA2-384, AES CBC and ECC384 and then clear the interrupt. Once the service is completed, the Security CPU writes to the interrupt set register at the Register Interface, which generates an interrupt to Application to inform that the request has been completed. The Application CPU can read the status register at the Register Interface and then send the next service request.

The design can be improved by allowing pipelined request for Crypto-384 service. Once the Security CPU finishes reading the data from the Register Interface, it can generate an interrupt to the Application CPU and to inform the Application CPU that the data has been read through a status register. The Application CPU can then send a new request while the current request is being serviced by Crypto-384.

Crypto services are provided through the Register Interface which consists of:

- Interrupt registers
- Control and status registers
- Scratch RAM (0x2D_0000 – 0x2D,3F00)

The Register Interface has interrupt registers, 8 Kbytes (8192 x 32) two-port (2RW) scratch memory and control/status registers. Its memory space mapping is shown in Figure 2.5.

Interrupt Registers are mapped to the top of the address space from 0x002C_0000 and 0x002C_0014. These registers are implemented in the FPGA fabric.

The Scratch Memory has 4096 bit depth, 32 bit width, and is implemented with EBRs with two RW ports. The memory address is from 0x002D_0000 to 0x000D_3FFF. The bottom of this memory space from 0x002D_3FEC to 0x000D_3FFC is used for control and status registers.

Other spaces such as the gap between (1) and (2) are unmapped.

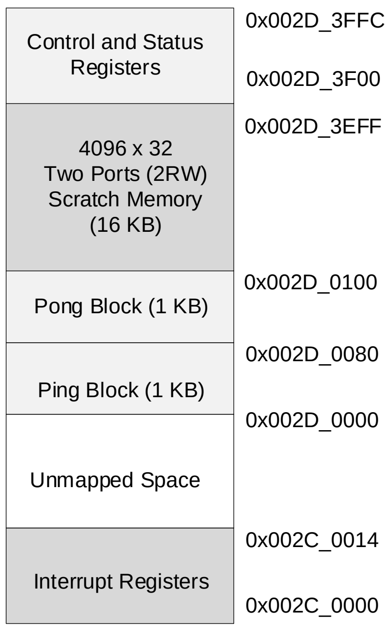


Figure 2.5. Register Interface Memory Mapping Space

Table 2.1. Interrupt Registers Definition in Register Interface

Name	Address Offset	Width (Bits)	Access from Application CPU	Access from Security CPU	Default	Description
int_status_app	0x2C_0000	1	RW1C	RO	0	Interrupt status register. 1'b0: no interrupt to the Application CPU. 1'b1: interrupt is set for the Application CPU. If int_enable_app is set, the interrupt is generated. The interrupt service routine at the Application CPU should write 1 to clear the interrupt after it is be serviced.
int_enable_app	0x2C_0004	1	RW	N/A	0	Interrupt enable register. 1'b0: interrupt to the Application CPU is disabled. 1'b1: interrupt to the Application CPU is enabled.
int_set_app	0x2E_0008	1	N/A	RW	0	Interrupt set register. 1'b0: no interrupt is set. 1'b1: causes int_status_app to be set to 1. Read from this register always return 0 per Lattice Hard IP Interface Standard (see ref[7]).
int_status_security	0x2E_000C	1	RO	RW1C	0	Interrupt status register. 1'b0: no interrupt to the Security CPU. 1'b1: interrupt is set for the Security CPU. If int_enable_security[0] is set, the interrupt is generated. The interrupt service routine at the Security CPU should write 1 to clear the interrupt after it is be serviced.
int_enable_security	0x2E_0010	1	N/A	RW	0	Interrupt enable register. 1'b0: interrupt to the Security CPU is disabled. 1'b1: interrupt to the Security CPU is enabled.
int_set_security	0x2C_0014	1	RW	N/A	0	Interrupt set register. 1'b0: no interrupt is set. 1'b1: causes int_status_security[0] to be set to 1. Read from this register always return 0 per Lattice Hard IP Interface Standard (see ref[7]).

The Control and Status registers are defined in Table 2.2. To save fabric resources, these Control and Status registers are mapped to the scratch memory (instantiating system memory with 2RW ports). It is better to map these registers near the end address of the scratch memory. If the scratch memory has 16 Kbytes with address offset from (0x0000 to 0x3FFF), the address offset should map backwards from the end of the space. In this way, the top space can be mapped to SHA and PKC with the same offset as their IP requires.

Table 2.2. Control and Status Registers

Register Name	Address Offset	Width (Bits)	Access from Application CPU	Access from Security CPU	Default	Description
Mode	0x0002D_3FFC	32	RW	R	0	CPU needs to write to this register. The Security CPU can only read it.
AES key length	0x0002D_3FF8	2	RW	R	0	Bit 0: 0 means 128-bit key 1 means 256-bit key
SHA source	0x0002D_3FF4	1	RW	R	0	Bit 0: 0 means SHA message source from Register File.
AES source	0x0002D_3FF0	1	RW	R	0	Bit 0: 0 means AES message source from PCIe
Status	0x0002D_3FEC	32	R	RW		Bit [31:16] Error Code (To be defined) Bit 0 0 means IDLE 1 means DONE
Version	0x0002D_3FE8	32	R	RW	0	Bit [31:16] version number of Crypto-384 IP Bit [15: 0] version number of Crypto CPU Firmware

Table 2.3. Mode Registers

Mode Register	Operating Mode
0x35	SHA384
0x36	HMAC-SHA384
0x3A	ECDH
0x3F	AES-256 CBC encryption
0x40	AES-256 CBC decryption
0x41	AES-256 GCM encryption
0x42	AES-256 GCM decryption
0x43	RSA 3K Authentication
0x44	RSA 4K Authentication
0x50	ECC 256 bit public key generation from a given private key*(using CRE I P)*
0x57	ECC 256 key pair generation (Qx, Qy, d) *(using CRE IP)*

The other memory spaces in the Register Interface are two-port scratch memory with starting address offset at 0x002D_0000. We can define the initial 32 entries as ping region (from 0x002D_0000 to 0x002D_007F) and the next 32 entries as pong region (from 0x002D_0080 to 0x002D_00FF).

To transfer data between the security firmware and the Application CPU firmware, the data regions below are defined.

Table 2.4. Scratch Memory Registers

Region	Offset	Size(Bytes)	Description
Ping buffer	0x2D,0000	128	Used to transfer SHA384 message and ECIES message
Pong buffer	0x2D,0080	128	
Output buffer	0x2D,0100	128	Buffer to output data
BUF1	0x2D,0180	48	Writing AES (CBC/GCM) key
BUF2	0x2D,01B0	48	Writing AES (CBC/GCM) initial vector
BUF3	0x2D,01E0	48	Writing AES (GCM) additional data
BUF4	0x2D,0210	48	Writing AES len(A)64 || len(C)64
BUF5	0x2D,0240	48	Reading GCM tag
Ping status	0x2D,0270	4	Bit 0: 1 – ping buffer ready APP CPU: when ready bit 0, write to buffer, then set 1 SEC CPU: when ready bit 1, read from buffer, then set 0 Always start from ping buffer
Pong status	0x2D,0274	4	Bit 0: 1 – pong buffer ready APP CPU: when ready bit 0, write to buffer, then set 1 SEC CPU: when ready bit 1, read from buffer, then set 0
Servo Status	0x2D,0278	4	Bit [1:0]: 0x0 – Servo Idle 0x1 – Servo in service 0x2 – Servo busy APP CPU: when servo idle, change state to servo in service, then issue interrupt to notify SEC CPU SEC CPU: when servo in service detected, set to servo busy then handle the request. After service done, change to servo idle then issue interrupt to notify APP CPU
Input buffer Size	0x2D,027C	4	Size of SHA input data in bytes
Output buffer Size	0x2D,0280	4	Size of data in output buffer
AES Mode (Read Only)	0x2D0290	4	1: AES CBC mode 2: AES GCM mode

2.4. Programming Model

At a high level, follow the steps below to program the registers:

1. From the Application CPU, write to the Register Interface control registers, data in scratch memory, and interrupt registers to generate an interrupt to the Security CPU.
2. The Security CPU interrupt service routine checks the Register Interface control registers (such as mode) and then reads the Register Interface scratch memory to ORAN Security Enclave IPs.
3. The Security CPU checks the Security Enclave registers for status and then copies the result back to the Register Interface scratch memory, and then writes to interrupt registers to generate an interrupt to the Application CPU. At the same time, the Security CPU should clear its interrupt.
4. The Application CPU checks the status registers at the Register Interface and reads the output back from the Register Interface scratch memory and then clears its interrupt.

3. Memory Map

The memory map of the three subsystems is defined in Table 3.1. Base Address remains same for every interface. Range needs to be changed according to memory requirement.

Table 3.1. Memory Map Details

Subsystem	Base Address	End Address	Range (Bytes)	Range (Bytes in Hex)	Size (Kbytes)	Block	Base Address	End Address
Application	00000000	0007FFFF	524288	80000	512	Application CPU Instruction RAM and data RAM	0	524287
	00080000	000803FF	1024	400	1	Application CPU PIC TIMER	524288	525311
	00080400	000BFFFF	261120	3FC00	255	RESERVED	525312	786431
	000C0000	000C1FFF	8192	2000	8	RESERVED	786432	794623
	000C2000	000C3FFF	8192	2000	8	UART	794624	802815
	000C4000	000C7FFF	16384	4000	16	SMBus Slave	802816	819199
	000C8000	000C9FFF	8192	2000	8	RESERVED	819200	827391
	000CA000	000CBFFF	8192	2000	8	PCIe EP	827392	835583
	000CC000	000CDFFF	8192	2000	8	RESERVED	835584	843775
	000CE000	000CEFFF	4096	1000	4	RESERVED	843776	847871
	000CF000	000CFFFF	4096	1000	4	RESERVED	847872	851967
	000D0000	000FFFFF	196608	30000	192	RESERVED	851968	1048575
	00100000	0013FFFF	262144	40000	256	HSE	1048576	1310719
	00140000	0017FFFF	262144	40000	256	Customer PLD Logic	1310720	1572863
PCIe	00180000	00180FFF	4096	1000	4	PCIe DMA control the Register Interface	1572864	1576959
	00181000	00182FFF	8192	2000	8	PCIe DMA descriptor and status queue	1576960	1585151
	00183000	0018FFFF	53428	D000	13	RESERVED	1585152	1638399
	00190000	001AFFFF	65536	10000	64	Ingress RAM Port S0: 0x0019_0000 Port S1: 0x001A_0000	1638400	1703935
	001B0000	001CFFFF	65536	10000	64	Egress RAM Port S0: 0x001B_0000 Port S1: 0x001C_0000	1703936	1769471

Subsystem	Base Address	End Address	Range (Bytes)	Range (Bytes in Hex)	Size (Kbytes)	Block	Base Address	End Address
Security	001D0000	001FFFF	65536	10000	64	RESERVED	1769472	1835007
	00000000	0001FFFF	131072	20000	128	Security CPU Instruction RAM/Data RAM	2097152	2162687
	00210000	0023FFFF	196608	30000	192	RESERVED	2162688	2359295
	00240000	0027FFFF	262144	40000	256	RESERVED	2359296	2621439
	00280000	002803FF	1024	400	1	Security CPU PIC/Timer	2621440	2622463
	00280400	002807FF	1024	400	1	Configuration Engine	2622464	2623487
	00280800	002BFFFF	260096	3F800	254	RESERVED	2623488	2883583
	002C0000	002FFFFF	262144	40000	256	Register Interface Port S0: 002C_0000 Port S1: 002E_0000	2883584	3145727
	00300000	0033FFFF	262144	40000	256	SHA2-384/ECC384/AES	3145728	3407871
	00340000	0034FFFF	65536	10000	64	RESERVED	3407872	3473407
	00350000	0035FFFF	65536	10000	64	RESERVED	3473408	3538943

4. IPs/RTL Blocks used in the Design

Some of the IPs used in this design, which are directly generated from Lattice Radiant software, are discussed in this section. Along with them, details about the ORAN Security Enclave (OSE) IP are also explained.

4.1. ORAN Security Enclave

Figure 4.1 shows the detailed architecture of the IP. It mainly involves three Crypto IP's namely Block Cipher IP, hash function IP and Public Key Cryptography (PKC) IP.

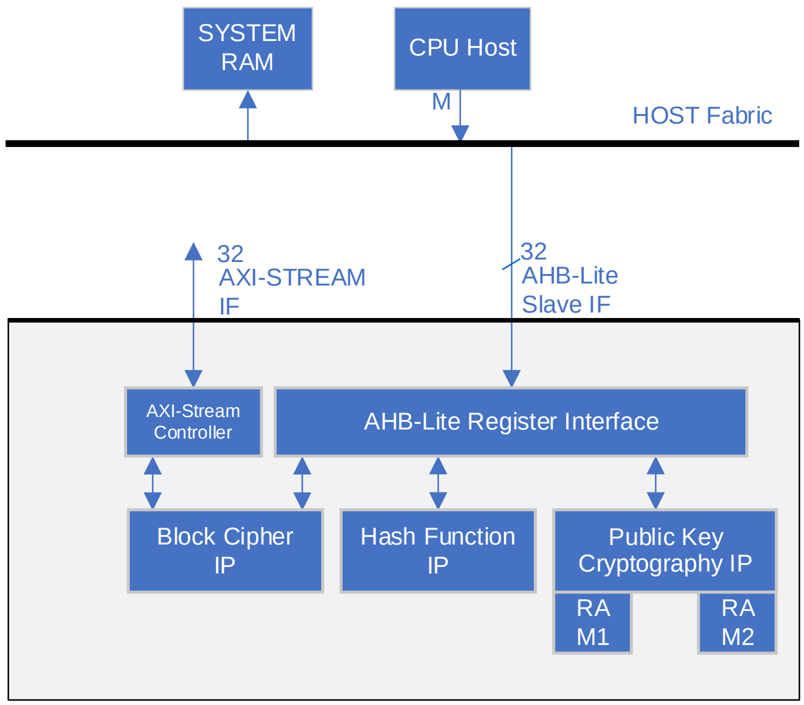


Figure 4.1. ORAN Security Enclave Detailed Architecture

Figure 4.1 shows that the AXI stream interface, AHB-Lite interface, and Clock reset interface are used in the OSE for inputs and outputs for the IP.

Table 4.1. OSE Top Level Signal Description

Ports	Direction	Width	Description
Clk	Input	1	Main clock signal, duty cycle 50:50
n_rst	Input	1	Asynchronous active - low reset signal
oran_security_enclave_irq_o	output	1	Interrupt signal raised by an IP done event
oran_security_enclave_ahblite_ahblite_slave_hsel_i	Input	1	hsel signal
oran_security_enclave_ahblite_ahblite_slave_hwrite_i	Input	1	hwrite Signal
oran_security_enclave_ahblite_ahblite_slave_hsize_i	Input	2	hsize signal
oran_security_enclave_ahblite_ahblite_slave_hburst_i	Input	3	hburst signal
oran_security_enclave_ahblite_ahblite_slave_hprot_i	Input	4	hprot signal
oran_security_enclave_ahblite_ahblite_slave_htrans_i	Input	2	htrans signal
oran_security_enclave_ahblite_ahblite_slave_hmastlock_i	Input	1	hmastlock signal
oran_security_enclave_ahblite_ahblite_slave_hready_i	Input	1	hready signal
oran_security_enclave_ahblite_ahblite_slave_haddr_i	Input	32	address signal

Ports	Direction	Width	Description
oran_security_enclave_ahblite_ahblite_slave_hwdata_i	Input	32	write data signal
oran_security_enclave_ahblite_ahblite_slave_hreadyout_o	Output	1	hreadyout signal
oran_security_enclave_ahblite_ahblite_slave_hresp_o	Output	1	hresp signal
oran_security_enclave_ahblite_ahblite_slave_hrdata_o	Output	32	read data signal
oran_security_enclave_axistream_slave_tdata_i	Input	128	AXI stream data input signal
oran_security_enclave_axistream_slave_tvalid_i	Input	1	AXI stream valid input signal
oran_security_enclave_axistream_slave_tready_o	Output	1	AXI stream readyout signal
oran_security_enclave_axistream_master_tdata_o	Output	128	AXI stream data out signal
oran_security_enclave_axistream_master_tvalid_o	Output	1	AXI stream valid out signal
oran_security_enclave_axistream_master_tready_i	Input	1	AXI stream ready in signal

4.1.1. AES CBC 256 IP

The Block-Cipher IP Core is a security enhanced hardware implementation of one or several block-cipher algorithms under a common and comprehensive interface. It also embeds one or several modes of operation with different key sizes on a 128 bits data path. The AES (Advanced Encryption Standard) performs encryption/decryption with 128/256 bits keying material, which is used in the Block-Cipher IP. The encryption and decryption in the Block-Cipher IP performs in two modes: CBC Mode (Cipher-Block Chaining) and GCM Mode (Galois Counter Mode). Based upon the requirements, the mode can be selected in the Block-Cipher IP.

Table 4.2. Block-Cipher IP (AES CBC-256) Register Description

Registers	Address Offset	Reset Value	Access	Description
VERSION	0x0000	0x00010005	RO	Version register. Provides a constant value relative to the project.
STATUS	0x0004	0x0	RO	Status register. Provides useful information regarding the IP. Bit[0]: Status 1’b1 : Ready for operation Bits[8:1] : Error code 0x01 : Unauthorized algorithm 0x02 : Unauthorized key size of algorithm 0x03 : Unauthorized cipher direction 0x04 : Unauthorized mode 0x05 : Unauthorized algorithm for mode 0x06 : Unauthorized direction for mode 0x07 : Unauthorized key size for algorithm and mode 0xff : Unknown setup error Bit [31:9] Reserved
CONTROL	0x0008	0x0	WO	Control register. This register is useful to start an AES operation and setup the random value to be used by the counter measure. It also drives the key selection signal as well as the key register lock signals. Bits[4:1] : Random data to be used by Block-Cipher IP Bit[0] : Control 1’b1 : Starts AES operation 1’b0 : Nothing

Registers	Address Offset	Reset Value	Access	Description
CONFIG	0x000C	0x0	WO	Configuration register. Used to set up the AES operation to be performed (encryption/decryption/mode). 31 → 27 Reserved (always read as 0b00000). 26 → 24 ALGO "0x00": AES "0x01": SM4 "0x02": TDEA "0x03": ARIA This field is always read as 0b000. 23 → 16 Reserved (always read as 0x00). 15 → 8 MODE "0x10": CBC_INIT "0x11": CBC_UPDATE 2 → 1 KSS "0x0": The Block Cipher IP is configured to use 128-bit keys. "0x1": The Block Cipher IP is configured to use 192-bit keys. "0x2": The Block Cipher IP is configured to use 256-bit keys. This field is always read as 0b00. 0 CD 0b0: encryption 0b1: decryption This field is always read as 0b0.
DIN	0x0010 to 0x001C	0x0	WO	Data input register. Contains the data to be processed by the Block Cipher IP. This register is wired to data_i. Bits[31 : 0] can be accessed at address 16 Bits[127 : 96] respectively at address 28
DOUT	0x0020 to 0x002C	0x0	RO	Data output register. Contains the data processed by the Block Cipher IP. This register is wired to data_o Bits[127 : 0]
KEY	0x0030 to 0x004C	0x0	WO	Key input registers. Contains the key. This register is wired to key_i[255: 0].
IV	0x0050 to 0x005C	0x0	WO	Initialization Vector (IV) input registers. Contains the Initialization Vector (IV). This register is wired to iv_i[127 : 0].

4.1.2. AES GCM 256 IP

The Block-Cipher IP Core is a security enhanced hardware implementation of one or several block-cipher algorithms under a common and comprehensive interface. It also embeds one or several modes of operation with different key sizes on a 128 bits data path. The AES performs encryption/decryption with 128/256 bits keying material, which is used in the Block-Cipher IP. The encryption and decryption in the Block-Cipher IP performs in two modes: CBC Mode (Cipher-Block Chaining) and GCM mode (Galois Counter Mode). Based upon the requirements, the mode can be selected in the Block-Cipher IP.

Table 4.3. Block-Cipher IP (AES GCM-256) Register Description

Registers	Address Offset	Reset Value	Access	Description
VERSION	0x0000	0x00020005	RO	Version register. Provides a constant value relative to the project.
STATUS	0x0004	0x0	RO	Status register
CONTROL	0x0008	0x0	WO	Control register. This register is useful to start an AES operation and setup the random value to be used by the counter measure. It also drives the key selection signal as well as the key register lock signals.
CONFIG	0x000C	0x0	WO	Configuration register. This register is useful to setup an AES operation to be performed (encryption/decryption/mode) 31 → 27 Reserved (always read as 0b00000). 26 → 24 ALGO "0x00": AES "0x01": SM4 "0x02": TDEA "0x03": ARIA This field is always read as 0b000. 23 → 16 optional block size in bits (CMAC_FINISH,CCM_UPDATE,GCM_update,GCM_FINISH,CCM_FINISH modes of operation only). 15 → 8 MODE "0x50": GCM_INIT "0x51": GCM_GHASH "0x52": GCM_UPDATE "0x53": GCM_FINISH_IV "0x54": GCM_FINISH 7->3 Reserved. 2 → 1 KSS "0x0": The Block Cipher IP is configured to use 128-bit keys. "0x1": The Block Cipher IP is configured to use 192-bit keys. "0x2": The Block Cipher IP is configured to use 256-bit keys. This field is always read as 0b00. 0 CD 0b0: encryption 0b1: decryption This field is always read as 0b0.
DIN	0x0010 to 0x001C	0x0	WO	Data input register. Contains the data to be processed by the Block Cipher IP. This register is wired to data_I Bits[127 : 0].
DOUT	0x0020 to 0x002C	0x0	RO	Data output register. Contains the data processed by the Block Cipher IP. This register is wired to data_o Bits[127 : 0].
KEY	0x0030 to 0x004C	0x0	WO	Key input registers. Contains the key. This register is wired to key_i[255 : 0].
IV	0x0050 to 0x005C	0x0	WO	Initialization Vector (IV) input registers. Contains the Initialization Vector (IV). This register is wired to iv_i[127 : 0].

4.1.3. Hash Function IP

Hash function is any function that can be used to map data of arbitrary size to fixed-size values. Based on that there are 5 classes of secure hash functions. SHA2-384 and SHA2-512 are the hash functions used. The values 384 and 512 represent the message digest size, which is fixed. The hash function takes the input and produces a hash value for the required message digest size.

Table 4.4. Hash Function IP Register Description

Registers	Address Offset	Default Value	Access	Description
VERSION	0x0000	0x00	RO	Version register. Provides information regarding the IP-Core version
STATUS	0x0004	0x00	RO	Status register. Provides information regarding the IP-Core. Bit[0] : Ready status 1'b1 : Ready 1'b0 : Not Ready
CONTROL	0x0008	0x00	RW	Control register. Starts an operation in the IP Core. Bit[0] : Start Operation. The operation begins when the bit is HIGH.
CONFIG	0x000C	0x00	RW	Operation, algorithm and mode selection port. x"04" : OP_HASH_INIT x"05" : OP_HASH_UPDATE x"06" : OP_HASH_FINISH others: Reserved
MSG_LEN_I(X)	0x0010 - 0x001c	0x0	RW	Message length input register for padding
DATA_I	0x0020	0x0	RW	Data Input register. Contains the data to be processed by the IP-Core.
DATA_O(X)	0x0034 - 0x0060	0x0	RO	Data Output register. Contains the data to be processed by the IP-Core.

4.1.4. Public Key Cryptography (PKC) IP

The PKC IP Core is a hardware implementation that supports and accelerates Public-Key Cryptography protocols of primitives standards. The PKC accelerator IP Core implements modular arithmetic over large numbers. This IP performs RSA primitives for the Digital Standard Signature and the RSA signature verification and ECC primitives for the Digital Standard Signature [ECDSA] and signature verification. Mathematically, the elliptic curve digital signature protocols are highly dependent of the ECSM (Elliptic Curve Scalar Multiplication) operation. The ECSM also relies on "lower-level" elliptic curve operations, namely the point doubling and point addition operations. Then, the RSA protocols depends on the modular exponentiation operation (called RSA primitives). Finally, the whole system depends on finite field arithmetic.

ECC module is composed of two ECDSA primitive operations: signature generation (private operation), signature verification (public operation). The physical counter measures are only used for the signature generation and the elliptic curve scalar multiplication in order to protect the private key.

Table 4.5. PKC IP Register Description

Registers	Address Offset	Default Value	Access	Description
MEMORY_1	0x0000	N.A	RW	Dedicated memory bank 1 for PKC accelerator
MEMORY_2	0x4000	N.A	RW	Dedicated memory bank 2 for PKC accelerator
OP0	0x6000	0x00	WO	Operand registers access for PKC accelerator.
OP1	0x6004	0x00	WO	Operand registers access for PKC accelerator.
OP2	0x6008	0x00	WO	Operand registers access for PKC accelerator.
OP3	0x600C	0x00	WO	Operand registers access for PKC accelerator.
OP4	0x6010	0x00	WO	Operand registers access for PKC accelerator.
OP5	0x6014	0x00	WO	Operand registers access for PKC accelerator.

Registers	Address Offset	Default Value	Access	Description
OP6	0x6018	0x00	WO	Operand registers access for PKC accelerator.
OP7	0x601C	0x00	WO	Operand registers access for PKC accelerator.
COUNT_LOAD_3	0x61DC	0x00	WO	Counter registers access for PKC accelerator.
COUNT_LOAD_2	0x61E0	0x00	WO	Counter registers access for PKC accelerator.
COUNT_LOAD_1	0x61E4	0x00	WO	Counter registers access for PKC accelerator.
NUM_WORDS_3	0x61E8	0x00	WO	Counter registers access for PKC accelerator.
NUM_WORDS_2	0x61EC	0x00	WO	Counter registers access for PKC accelerator.
NUM_WORDS_1	0x61F0	0x00	WO	Counter registers access for PKC accelerator.
STATUS	0x61F4	0x00	WO	PKC accelerator status register.
CONTROL	0x61F8	0x00	WO	PKC accelerator control register.
VERSION	0x61FC	0x00600505	RO	PKC accelerator version.

4.2. CRE Module IP

CRE stands for Cryptographic Engine. Figure 4.2 shows the block diagram of the CRE Module IP.

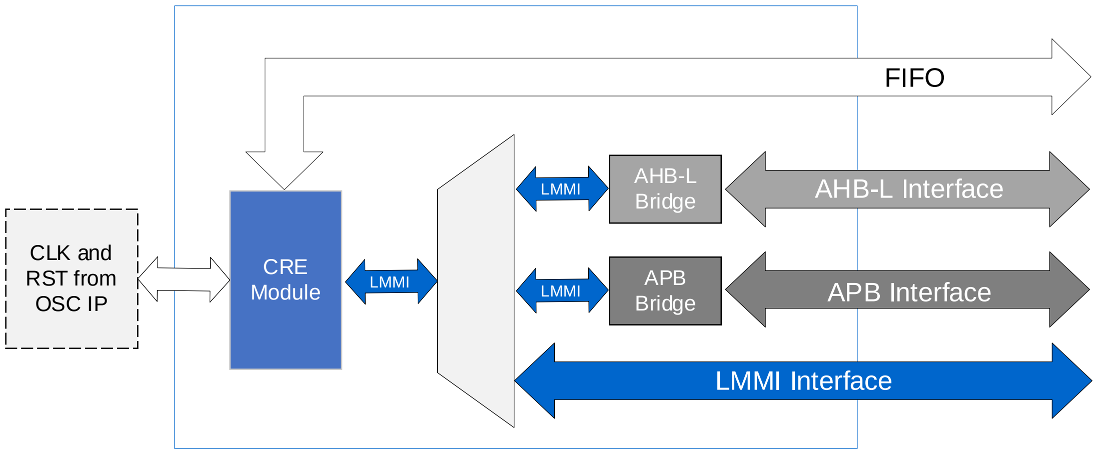


Figure 4.2. CRE Module IP Block Diagram

The FIFO I/O only applies to FIFO control pins; the data pins are shared with the LMMI. The FIFO interface also requires the information to be sent through the LMMI bus, hence the FIFO pins are only available if the LMMI interface is also selected.

The LMMI Interface is the native interface of the IP, and the most resource-efficient interface of the CRE Module. Using this interface, the user can directly use all the native IP features without using any fabric or additional control signals.

The LMMI + FIFO interface is similar to the native LMMI interface with the addition of a FIFO control port. The FIFO shares its input and output data connection with the LMMI's input and output connection, hence, the user must design additional circuitry to fully take advantage of this interface. The benefits of the FIFO data path are the increased throughput for AES/SHA transactions. In this configuration, the user can still utilize all the features of the IP while minimizing resource utilization.

Important: The LMMI write and read data ports are shared with the FIFO interface. Proper care must be taken when writing/sampling data to/from the IP using different clocks. This document assumes that the user has properly taken care of any possible clock crossing issues which could arise from the use of asynchronous clocks.

Table 4.6. CRE Module IP Signal Description

Signals	Direction	Width (Bits)	Description
Core IP Signals			
cfg_clk_i	INPUT	1	Configuration Clock Signal (from OSC IP)
cre_clk_i	INPUT	1	CRE Clock Signal (from OSC IP)
cre_rstn_i	INPUT	1	CRE Engine Reset Signal (Active Low)
LMMI Slave Interface			
lmmi_clk_i	INPUT	1	Clock Signal of the LMMI Interface
lmmi_resetn_i	INPUT	1	LMMI Reset Signal. Active Low, LMMI interface is in reset when asserted.
lmmi_request_i	INPUT	1	Active HIGH signal, indicates that the master wants initiate a transaction when asserted.
lmmi_wr_rdn_i	INPUT	1	Active HIGH signal, indicates a write transaction when the asserted.
lmmi_offset_i	INPUT	18	Offset address, the accessed location of the current active transaction.
lmmi_wdata_i	INPUT	32	Input data, the data to be written in the offset address. (This port is shared with the FIFO data input).

Signals	Direction	Width (Bits)	Description
lmmi_rdata_o	OUTPUT	32	Output data, the data result from the previous transaction. (This port is shared with the FIFO data output).
lmmi_rdata_valid_o	OUTPUT	1	Active HIGH, indicates that the data is valid when asserted.
lmmi_ready_o	OUTPUT	1	Active HIGH, indicates that the slave is ready to receive transactions when asserted.
FIFO Interface			
Async_fifo_clk_i	INPUT	1	Clock Signal of the FIFO Interface
Async_fifo_rst_i	INPUT	1	FIFO Reset Signal, Active HIGH, indicates that the FIFO interface is reset when asserted
Async_fifo_wr_en_i	INPUT	1	Active HIGH, indicates that an input data would be written to the FIFO if the FIFO is not full.
Async_fifo_rd_en_i	INPUT	1	Active HIGH, indicates that an output data would be generated from the FIFO if the FIFO is not empty.
Async_fifo_full_o	OUTPUT	1	Active HIGH, indicates that the FIFO is full.
Async_fifo_empty_0	OUTPUT	1	Active HIGH, indicates that the FIFO is empty.

Table 4.7. CRE Module IP Register Description

Name	LMMI [17:0]	Size	R/W	Description
RI_CTRL1	0x2 000C	4B	W/O	Instruction register, writing to this register defines the current function of the CRE Engine and automatically starts the Engine: 0x00: Clears previous instruction 0x02: True Random Generation 0x04: Generates ECC public keys from a private key 0x05: Starts SHA256 0x06: Starts HMAC-SHA256 0x07: Starts ECIES Encryption 0x08: Starts ECIES Decryption 0x09: Starts AES Engine 0x0C: Starts ECDSA Generation 0x0D: Starts ECDSA Verification 0x0E: Generates both ECC private and public keys from TRNG engine
RI_CTRL3	0x2 0014	4B	W/O	Sets the size of the message to be encrypted / decrypted (ECIES [1760B max] / HMAC-SHA [1980B max)
AES_SIZE	0x2 0018	4B	W/O	Sets the size of the key used in the encryption / decryption process (AES) 0x00: 128-bits (16B) 0x01: 256-bits (32B)
RO_GP0	0x2 0020	4B	R/O	Shows the current status of the CRE Engine: 0x0B0: Engine is ready to accept instructions 0x0B1: Engine is busy 0x0B2: Engine has completed performing instructions
DPA_CON	0x2 0030	4B	W	Writes the information controlling the Differential Power Analysis features of the IP. Bit[0] controls "Clock Randomization" Bit[1] controls "Random Noise Addition" Bit[3] controls the JML counter operation
DATA_SRC	0x2 003C	4B	W/O	Sets the data source for the SHA / AES engine: 0x00: Sets the AES engine data source to the bus 0x02: Sets the SHA engine data source to the bus 0x03: Sets the SHA engine data source to the FIFO 0x04: Sets the AES engine data source to the FIFO
AES_CON	0x2 2040	4B	W/O	AES control register, sets the current function of the AES engine to either encrypt or decrypt 0x00: Encryption 0x01: Decryption
AES_STAT	0x2 2044	4B	R/O	Shows the current status of the AES Engine: AES_STAT[0] = 0: AES is busy expanding the key AES_STAT[0] = 1: AES key expansion ready AES_STAT[1] = 0: AES is encrypting / decrypting AES_STAT[1] = 1: AES process finished
SHA_INIT	0x2 3070	4B	W/O	Initializes the SHA engine, must be written with 0x01 followed by 0x00

4.3. SMBus Controller

The System Management Bus (SMBus) is a two-wire interface through which simple system and power management devices can communicate with the rest of the system. The protocol is compatible with the I2C bus protocol and is often found in monitoring power conditions, temperature, and other sensors on a board. While SMBus is derived from I2C, there are several major differences existing between the specifications of the two buses. The device that initiates the transmission on the SMBus is commonly known as the Master, while the device being addressed is called the Slave.

SMBus protocols support many kinds of formats, such as SMBus write byte, SMBus write word, SMBus read byte, SMBus read word, SMBus write block, SMBus read block and so on. SMBus write byte and read byte message format is shown in Figure 4.3 and Figure 4.4.

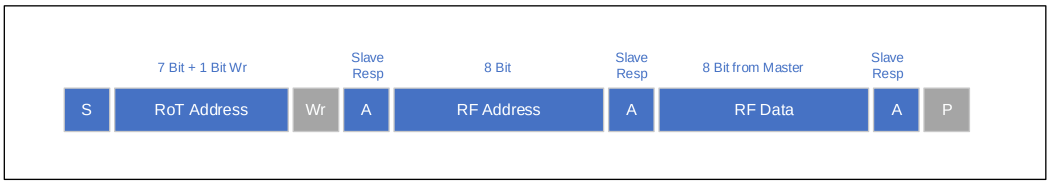


Figure 4.3. SMBus Mailbox Write Byte Message

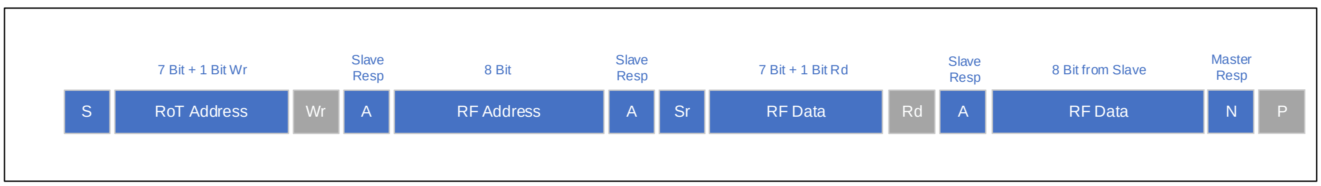


Figure 4.4. SMBus Mailbox Read Byte Message

The MCTP over SMBus/I2C transport binding defines how MCTP packets are delivered over a physical SMBus or I2C medium using SMBus transactions. All MCTP transactions are based on the SMBus Block Write bus protocol. The first 8 bytes make up the packet header. The first three fields—Destination Slave Address, Command Code, and Length—map directly to SMBus functional fields. The remaining header and payload fields map to SMBus Block Write "Data Byte" fields. The inclusion of the Source Slave Address in the header is specified by MCTP rather than SMBus. This is done to facilitate addressing required for establishing communications back to the message originator. The MCTP over SMBus packet format as shown in Figure 4.5.

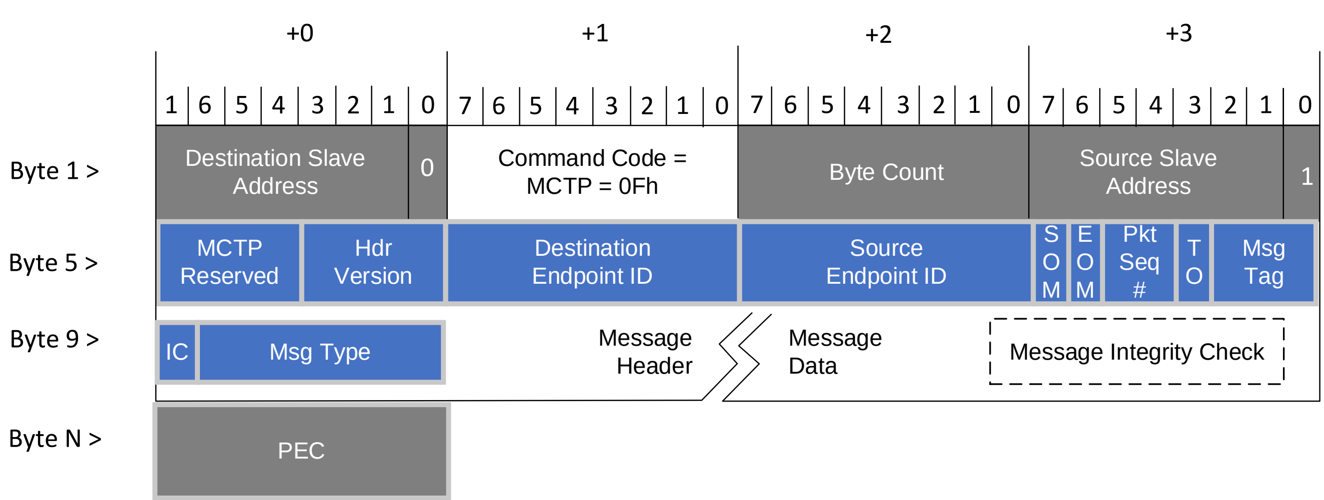


Figure 4.5. MCTP over SMBus Packet Format

Table 4.8. SMBus IP Interface Signal Description

Signal Name	Width	Direction	Description
Clock and Reset			
clk_i	1	input	system clock
rst_n_i	1	input	System reset. The reset assertion can be asynchronous but reset negation should be synchronous. When asserted, output ports and registers are forced to their reset values.
AHB-Lite Bus			
ahbl_hsel_slv_i	1	input	AHBL Select signal Indicates that the slave device is selected and a data transfer is required.
ahbl_haddr_slv_i	32	input	The system address bus.
ahbl_hburst_slv_i	3	input	3'b000: SINGLE burst 3'b001: INCR Incrementin gburst of undefined lengh (NOT supported) 3'b010: WRAP4 4-bit wrapping burst 3'b011: INCR4 4-bit incrementing burt 4'b100: WRAP8 8-bit wrapping burst 3'b101: INCR8 8-bit incrementing burst 8'b110: WRAP16 16-bit wrapping burst 3'b111: INCR16 16-bit incrementing burst
ahbl_hprot_slv_i	4	input	ahbl_hprot_slv_i [0] :1'b0 - opcode fetch; 1'b1 - data access ahbl_hprot_slv_i [1]: 1'b0 - user access; 1'b1 - privileged access ahbl_hprot_slv_i [2]: 1'b0 - non-bufferable, 1'b1 - bufferable ahbl_hprot_slv_i [3]: 1'b0 - non-cacheable; 1'b1 - cacheable
ahbl_hsize_slv_i	3	input	3'b000: 1 byte 3'b001: 2 bytes 3'b010: 4 bytes
ahbl_htrans_slv_i	2	input	Indicates the transfer type of the current transfer. This can be: 2’b00: IDLE 2’b01: BUSY 2’b10: NONSEQUENTIAL 2’b11: SEQUENTIAL
ahbl_hwdata_slv_i	32	input	The write data bus
ahbl_hwrite_slv_i	1	input	When HIGH, this signal indicates a write transfer and when LOW a read transfer.
ahbl_hready_slv_i	1	input	This signal should come from AHBL Interconnect. When set to 1, this indicates the previous transfer is complete.
ahbl_hrdata_slv_o	32	output	The read data bus
ahbl_hreadyout_slv_o	1	output	When HIGH, this signal indicates that a transfer has finished on the bus. This signal can be driven LOW to extend a transfer.
ahbl_hresp_slv_o	1	output	When LOW, this signal indicates that the transfer status is OKAY. When HIGH, it indicates that the transfer status is ERROR.

4.3.1. SMBus Functional Description

The SMBus interface is connected to the external bus through SDA/SCL signals. It connects to the Master Controller through P1 and to the Slave Controller through P0. Because Master Controller and Slave Controller share the same interface, a switch between these two controllers is required. The switch is implemented in the SMBus interface through the following method. If the Master Controller does not initiate transfer, P0 is routed to SMBus interface and P1 is switched of. Otherwise, P1 is routed to the SMBus interface, and P0 is switched off.

The Master Controller can initiate SMBus transfer to access other SMBus Slaves. The MCTP transfer is also controlled by this Master Controller logic. The Master Controller supports multi-master on one bus simultaneously.

The SMBus IP Core functional block diagram is shown in Figure 4.6.

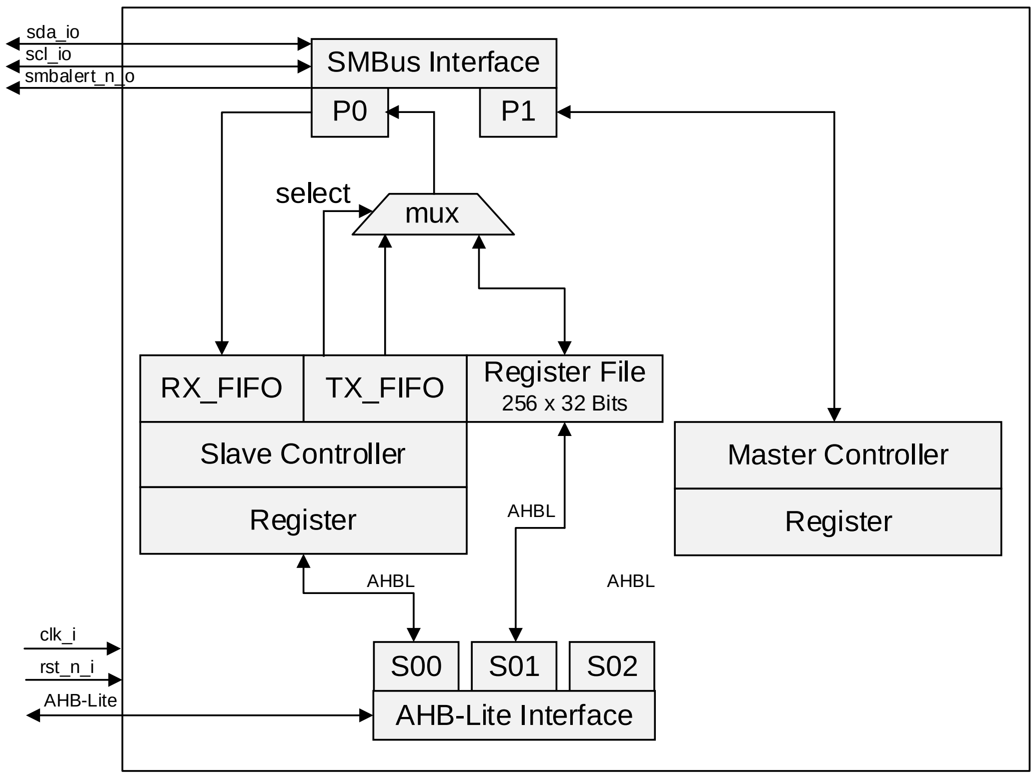


Figure 4.6. SMBus IP Core Functional Block Diagram

Table 4.9. SMBus Register Map Details

Offset	Register Name	Access	Reset	Description
0x0	RD_DATA_REG	RO	Not guaranteed	Read data register
0x4	WR_DATA_REG	WO	Not guaranteed	Write data register
0x8	SLVADR_L_REG	RW	[7] RSVD [6:0] I2C Slave Address[6:0]	Slave address lower register, same as I2C Slave address attribute
0xC	CONTROL_REG	RW	[7:5] RSVD [4:1] 0 [0] Addressing Mode in	Control register 7:5: RSVD 4: nack_data 3: nack_addr 2: reset 1: clk_stretch_en 0: addr_10bit_en
0x10	TGT_BYTE_CNT_REG	RW	8'h00	Target byte count register
0x14	INT_STATUS1_REG	RW1C	8'h00	Interrupt status first register 7: tr_cmp_int 6: stop_det_int 5: tx_fifo_full_int 4: tx_fifo_aempty_int 3: tx_fifo_empty_int 2: rx_fifo_full_int 1: rx_fifo_afull_int 0: rx_fifo_ready_int
0x18	INT_ENABLE1_REG	RW	8'h00	Interrupt enable register 7: tr_cmp_int 6: stop_det_int 5: tx_fifo_full_en 4: tx_fifo_aempty_en 3: tx_fifo_empty_en 2: rx_fifo_full_en 1: rx_fifo_afull_en 0: rx_fifo_ready_en
0x1c	INT_SET1_REG	WO	8'h00	Interrupt set first register 7: tr_cmp_set 6: stop_det_set 5: tx_fifo_full_set 4: tx_fifo_aempty_set 3: tx_fifo_empty_set 2: rx_fifo_full_set 1 :rx_fifo_afull_set 0: rx_fifo_ready_set
0x20	INT_STATUS2_REG	RW	8'h00	Interrupt status second register 7:3 reserved 2: external SMBus Master access slave default address(7'h61) 1: stop_err_int 0: start_err_int

Offset	Register Name	Access	Reset	Description
0x24	INT_ENABLE2_REG	RW	[7:2] RSVD [1:0] 2'b00	Interrupt enable second register 7: 2 reserved 1: stop_err_en 0: start_err_en
0x28	INT_SET2_REG	WO	[7:2] RSVD [1:0] 2'b00	Interrupt set second register 7: 2 reserved 1: stop_err_set 0: start_err_set
0x2c	FIFO_STATUS_REG	RO	[7:6] RSVD [5:0] 6'b011001	FIFO status register 7: 6 Reserved 5: tx_fifo_full 4: tx_fifo_aempty 3: tx_fifo_empty 2: rx_fifo_full 1: rx_fifo_afull 0: rx_fifo_empty
0x30	SMB_CONTROL_REG	RW	[7:1] RSVD [0] 1'b0	SMBus control and status register [7:1] RSVD [0] smb_alert: Transmits the alert interrupt to SMBus Master 1'b0 – No interrupt to Master 1'b1 – SMBus Slave sends alert interrupt to Master
0x34 to 0x3c	Reserved	RSVD	RSVD	Reserved Write access is ignored and 0 is returned on read access.

4.3.2. SMBus Program Flow

The SMBus mailbox IP is used as SMBus Master and SMBus Slave simultaneously. However, the SMBus Master function can also be disabled by unchecking Enable Master Function attribute box when configuring the IP in the Lattice Propel Builder software.

If both SMBus Master and SMBus Slave are enabled, when SMBus Master initiates a transfer, SMBus Slave logic halts and it cannot receive external master's messages. When SMBus Master logic halts, SMBus Slave logic wakes up and waits for external master's messages. The SMBus mailbox IP needs initialization for both SMBus Master and SMBus Slave controller logics before normal operation.

4.3.3. SMBus Slave Controller Initialization Flow

To perform initialization, load the appropriate registers of the Slave Controller namely:

- SLAVE_ADDRL_REG, SLAVE_ADDRH_REG – This step is optional. In most cases, the initial value set in I2C Slave Addresses attribute of the user interface does not need to be changed. Read access to the address by the external SMBus Master is routed to the Register File, while write access to the address is routed to the internal RX_FIFO.
- CONTROL_REG
- TGT_BYTE_CNT_REG – It is recommended to set this if the size of the data is known. Set this to 8'h00 if the number of bytes to transfer is not known, that is receiving unknown amount of data.
- INT_ENABLE1_REG – It is recommended to enable only the following interrupts when receiving commands from master.
  Transfer Complete Interrupt – If the size of data is known
  Receive FIFO Data Interrupt – If the size of data is unknown
- INT_ENABLE2_REG – It is recommended to enable both error interrupts

4.3.4. SMBus Master Initialization

Write the appropriate data to the prescale register based on the frequency of SCL through the AHB-Lite bus S02.The SCL frequency meets the equation: 5 × SCL frequency = clk_i / (PRERhi<<8 + PRERlo).

4.3.5. SMBus Slave Controller Operation Flow

This section describes the data transfer process in response to the read request of the external SMBus Master. It is assumed that the amount of data to send is known.

To perform data transfer in response to read request of SMBus Master:

1. Write data to WR_DATA_REG, amounting to <= FIFO Depth.
2. Enable only Transfer Complete Interrupt if transmit data is > FIFO Depth.
3. Enable TX FIFO Almost Empty interrupt if there are no other data to transfer. Otherwise, proceed to step 8.
4. Wait for TX FIFO Almost Empty Interrupt.

   If polling mode is desired, read INT_STATUS1_REG until tx_fifo_aempty_int asserts.

   If interrupt mode is desired, wait for the interrupt signal to assert.

   Read INT_STATUS1_REG and check that tx_fifo_aempt_int is asserted.

   Read INT_STATUS2_REG to make sure that no error occurred.

   Clear TX FIFO Almost Empty Interrupt. It is also acceptable to clear all interrupts.
5. Write data byte to WR_DATA_REG, amounting to less than or equal to (FIFO Depth - TX FIFO Almost Empty Setting).
6. If there are remaining data to transfer, go back to Step 3, otherwise, disable TX FIFO Almost Empty Interrupt.
7. Wait for Transfer Complete Interrupt.

   If polling mode is desired, read INT_STATUS1_REG until tr_cmp_int asserts.

   If interrupt mode is desired, wait for interrupt signal to assert.

   Read INT_STATUS1_REG and check if tr_cmp_int is asserted.

   Read INT_STATUS2_REG to make sure that no error occurred.
8. Clear all interrupts.

4.3.6. SMBus Master Controller Operation Flow

In the SMBus Master program flow, the Master Controller is used in polling mode. The polling mode is the same as the interrupt mode. However, the polling mode needs to poll the SR bit 0 instead of interrupted by int_o to check status. In the polling mode, set the CTR to 0x80.

4.3.7. Write Data to SMBus Slave

To write date to SMBus Slave:

1. Write 0x80 to the control register (CTR) to enable the SMBus Controller through the AHB-Lite bus. For enable interrupt, the write data is 0xC0.
2. Read the status register (SR) through the AHB-Lite bus until all bits of the status register is 0.
3. Write the SMBus Slave address and write bit to the transmit register (TXR) through the AHB-Lite bus.
4. Write 0x90 to the command register (CR) through the AHB-Lite bus to start the SMBus write operation.
5. When using polling mode, read the status register (SR) until bit 0 of the status register is set and check if other bits except bit 6 are 0s.

   When using interrupt mode, if host is interrupted by int_o signal, read the status register (SR) and check if other bits except bit 0 and bit 6 are 0s.

   Both modes need to write 0x1 to CR to clear bit 0 of SR. If other bits except bit 0 and bit 6 are not 0s, there is an error. Write 0x5 to CR to clear SR and go back to step 2.
6. Write the byte which is sent to the SMBus Slave to the transmit register (TXR) through the AHB-Lite bus.

7. Write 0x10 to the CR through the AHB-Lite bus to set SMBus write operation.
8. When using polling mode, read the status register (SR) until bit 0 of the status register is set and check if other bits except bit 6 are 0s.

   When using interrupt mode, if host is interrupted by int_o signal, read the status register (SR) and check if other bits except bit 0 and bit 6 are 0s.

   Both modes need to write 0x1 to CR to clear bit 0 of SR. If other bits except bit 0 and bit 6 are not 0s, there is an error. Write 0x5 to CR to clear SR and go back to step 2. If there is no error, another data needs to be written.

   Go back to step 6.
9. When all the bytes are sent, write 0x40 to the command register (CR) through the AHB-Lite bus to stop the SMBus write operation.
10. When using polling mode, read the status register (SR) until bit 0 of the status register is set and check if other bits except bit 6 are 0s. Bit6 is set when another master uses the bus at this time. Otherwise it also should be 0.

4.3.8. Read Data from SMBus Slave

To read data from SMBus Slave:

1. Write 0x80 to the control register (CTR) to enable the SMBus Controller through the AHB-Lite bus. If enable interrupt, the write data is 0xC0.
2. Read the status register (SR) through the AHB-Lite bus until all bits of the status register is 0s.
3. Write the SMBus Slave address and the read bit to the transmit register (TXR) through the AHB-Lite bus.
4. Write 0x90 to the command register (CR) through the AHB-Lite bus to start the SMBus read operation.
5. When using polling mode, read the status register (SR) until bit 0 is set and check if other bits except bit 6 are 0s.

   When using interrupt mode, if host is interrupted by int_o signal, read the status register (SR) and check if other bits except bit 0 and bit 6 are 0s.

   Both modes need to write 0x1 to CR to clear bit 0 of SR. If other bits except bit 0 and bit 6 are not 0s, there is an error. Write 0x5 to CR to clear SR and go back to step 2.
6. Write 0x20 to command register (CR) through the AHB-Lite bus to read data from the slave. If it is the last byte to read, write 0x28 to command register (CR) to NACK last byte.
7. When using polling mode, read the status register (SR) until bit 0 is set and check if other bits except bit 6 are 0s.

   When using interrupt mode, if host is interrupted by int_o signal, read the status register (SR) and check if other bits except bit 0 and bit 6 are 0s.

   Both modes need to write 0x1 to CR to clear bit 0 of SR. If other bits except bit 0 and bit 6 are not 0s, there is an error. Write 0x5 to CR to clear SR and go back to step 2.
8. Read data from the receive register (RXR) through the AHB-Lite bus. If there is no error and another data needs to be read, go back to step 6.
9. When the read operation is finished, write 0x40 to the command register (CR) through the AHB-Lite bus to stop the SMBus read operation.
10. When using polling mode, read the status register (SR) until bit 0 is set and check if other bits, except bit 6, are 0s. Bit 6 is set when other master use the bus at this time, otherwise it also should be 0.

    When using interrupt mode, if host is interrupted by int_o signal, read the status register (SR) and check if other bits except bit 0 and bit 6 are 0s.

    Both modes need to write 0x1 to CR to clear bit 0 of SR. If other bits except bit 0 and bit 6 are not 0s, there is an error. Write 0x5 to CR to clear SR and go back to step 9.

4.4. PCIe Subsystem IP

The PCIe subsystem is built by the PCIe Endpoint IP configured with DMA, two AHBL Master interfaces, and one APB interface. The IP also has the Ingress RAM and the Egress RAM.

At the top level of this IP, the following set of signals are present.

- PCIe interface signals and status signals
- APB slave interface for controlling the PCIe register interface
- Two AHBL Slave Ports (S1) for the Ingress RAM and Egress RAM data/configuration
- Two AXI Stream Ports (AXI master and AXI slave)

Table 4.10. PCIe IP Signal Description

Signal Name	Width	Direction	Description
rxp_i	1	input	Differential receive serial signal, RX+
rxn_i	1	input	Differential receive serial signal, RX-
txp_o	1	output	Differential transmit serial signal, TX+
txn_o	1	output	Differential transmit serial signal, TX-
clk_125	1	input	User clock 125 MHz
refclkp_i	1	input	Differential reference clock, CLK+ (100 MHz)
refclkn_i	1	input	Differential reference clock, CLK- (100 MHz)
perst_n_i	1	input	PCI Express fundamental reset active-low asynchronous assert, synchronous de-assert reset to the Link Layer, PHY, and Soft Logic blocks.
refret_i	1	input	1’b0
rext_i	1	input	1’b0
usr_rst_n	1	input	System reset. The reset assertion can be asynchronous but reset negation should be synchronous. When asserted, output ports and registers are forced to their reset values.
pll_lock	1	output	Pll_lock output along with reset
clk_sel,	1	output	1'b1; For END_POINT
pcie_sel,	1	output	1'b0; For END_POINT
pcie_sw1_pd	1	output	1’b0
pcie_sw2_pd	1	output	1’b0
linkup_done	1	output	PCIe link up
clock_flag	1	output	Clock flag reserved
dma_done_o	1	output	This signal indicates DMA completion
ahbl_s1_clk_i	1	input	Clock for AHB transactions
ahbl_s1_rstn_i	1	input	Reset for AHB transactions.
AHB-Lite Bus (Egress)			
ahbl_eg_s1_select_i	1	input	AHBL Select signal This signal indicates that the slave device is selected and a data transfer is required.
ahbl_eg_s1_address_i	32	input	The system address bus.
ahbl_eg_s1_burst_i	3	input	3'b000: SINGLE Single burst 3'b001: INCR Incrementing burst of undefined length (NOT supported) 3'b010: WRAP4 4-bit wrapping burst 3'b011: INCR4 4-bit incrementing burst 4'b100: WRAP8 8-bit wrapping burst 3'b101: INCR8 8-bit incrementing burst 8'b110: WRAP16 16-bit wrapping burst 3'b111: INCR16 16-bit incrementing burst

Signal Name	Width	Direction	Description
ahbl_eg_s1_prot_i	4	input	ahbl_hprot_slv_i [0] :1'b0 - opcode fetch; 1'b1 - data access ahbl_hprot_slv_i [1]: 1'b0 - user access; 1'b1 - privileged access ahbl_hprot_slv_i [2]: 1'b0 - non-bufferable, 1'b1 - bufferable ahbl_hprot_slv_i [3]: 1'b0 - non-cacheable; 1'b1 - cacheable
ahbl_eg_s1_size_i	3	input	3'b000: 1 byte 3'b001: 2 bytes 3'b010: 4 bytes
ahbl_eg_s1_mastlock_i	1	input	When HIGH, this signal indicates that the current transfer is part of a locked sequence. It has the same timing as the address and control signals.
ahbl_eg_s1_trans_i	2	input	This signal indicates the transfer type of the current transfer. This can be: 2'b00: IDLE 2'b01: BUSY 2'b10: NONSEQUENTIAL 2'b11: SEQUENTIAL
ahbl_eg_s1_wdata_i	32	input	The write data bus
ahbl_eg_s1_write_i	1	input	When HIGH, this signal indicates a write transfer and when LOW a read transfer.
ahbl_eg_s1_ready_o	1	output	This signal indicates whether slave is ready or not. When set to 1, this indicates slave is ready.
ahbl_eg_s1_rdata_o	32	output	The read data bus
ahbl_eg_s1_resp_o	1	output	When LOW, this signal indicates that the transfer status is OKAY. When HIGH, it indicates that the transfer status is ERROR.
AHB-Lite Bus (Ingress)			
ahbl_ing_s1_select_i	1	input	AHBL select signal This signal indicates that the slave device is selected and a data transfer is required.
ahbl_ing_s1_address_i	32	input	The system address bus.
ahbl_ing_s1_burst_i	3	input	3'b000: SINGLE Single burst 3'b001: INCR Incrementing burst of undefined length (NOT supported) 3'b010: WRAP4 4-bit wrapping burst 3'b011: INCR4 4-bit incrementing burst 4'b100: WRAP8 8-bit wrapping burst 3'b101: INCR8 8-bit incrementing burst 8'b110: WRAP16 16-bit wrapping burst 3'b111: INCR16 16-bit incrementing burst
ahbl_ing_s1_prot_i	4	input	ahbl_hprot_slv_i [0] :1'b0 - opcode fetch; 1'b1 - data access ahbl_hprot_slv_i [1]: 1'b0 - user access; 1'b1 - privileged access ahbl_hprot_slv_i [2]: 1'b0 - non-bufferable, 1'b1 - bufferable ahbl_hprot_slv_i [3]: 1'b0 - non-cacheable; 1'b1 - cacheable
ahbl_ing_s1_size_i	3	input	3'b000: 1 byte 3'b001: 2 bytes 3'b010: 4 bytes
ahbl_ing_s1_mastlock_i	1	input	When HIGH, this signal indicates that the current transfer is part of a locked sequence. It has the same timing as the address and control signals.

Signal Name	Width	Direction	Description
ahbl_ing_s1_trans_i	2	input	This signal indicates the transfer type of the current transfer. This can be: 2’b00: IDLE 2’b01: BUSY 2’b10: NONSEQUENTIAL 2’b11: SEQUENTIAL
ahbl_ing_s1_wdata_i	32	input	The write data bus
ahbl_ing_s1_write_i	1	input	When HIGH, this signal indicates a write transfer and when LOW a read transfer.
ahbl_ing_s1_ready_o	1	output	This signal indicates whether slave is ready or not. When set to 1, this indicates slave is ready.
ahbl_ing_s1_rdata_o	32	output	The read data bus
ahbl_ing_s1_resp_o	1	output	When LOW, this signal indicates that the transfer status is OKAY. When HIGH, it indicates that the transfer status is ERROR.
APB-Interface			
apb_s_clk_i	1	input	Clock for apb transactions
apb_s_rstn_i	1	input	Reset for apb transactions.
apb_s_sel_i	1	input	AHBL select signal Indicates that the slave device is selected and a data transfer is required.
apb_s_addr_i	32	input	The system address bus.
apb_s_enable_i	1	input	Enable This signal indicates the second and subsequent cycles of an APB transfer
apb_s_wdata_i	32	input	The write data bus
apb_s_write_i	1	input	When HIGH, this signal indicates a write transfer and when LOW a read transfer.
apb_s_ready_o	1	output	This signal indicates whether slave is ready or not. When set to 1, this indicates slave is ready.
apb_s_rdata_o	32	output	The read data bus
apb_s_slverr_o	1	output	When LOW, this signal indicates that the transfer status is OKAY. When HIGH, it indicates that the transfer status is ERROR.
AXI Interface			
axi_clk_i	1	input	Clock for AXI transactions.
axi_rstn_i	1	input	Reset for AXI transactions.
s_axis_tdata_i	128	input	Data bus.
s_axis_tvalid_i	1	input	When this signal is HIGH, the data is valid.
s_axis_tready_o	1	output	Slave is ready if this signal is HIGH.
m_axis_tdata_o	128	output	Data bus.
m_axis_tvalid_o	1	output	When this signal is HIGH, the data is valid.
m_axis_tready_i	1	input	Indicates slave is ready.

Table 4.11. Attribute Summary

Attribute	Values	Description
PCIe Device Type (not connected internally)	End port	PCIe acts as end port
SIM	0 1	For synthesis For simulation
DATA interface with ip (not connected internally)	TLP interface	PCIe uses TLP interface
TARGET LINK SPEEED (not connected internally)	2.5 G	PCIe with Gen 1
Main clock frequency (not connected internally)	125 MHz	PCIe with 125 MHz clock
FPGA_VERSION	32 bit value in hex	FPGA version should be given
ING_S0_BASE_ADDR	32 bit value in hex	Default value = 32'h00190000
ING_S1_BASE_ADDR	32 bit value in hex	Default value = 32'h001A0000
EG_S0_BASE_ADDR	32 bit value in hex	Default value = 32'h001B0000
EG_S1_BASE_ADDR	32 bit value in hex	Default value = 32'h001C0000

4.5. Reset Sync

The Reset Sync module is used to synchronize the external reset coming to the FPGA using the debounce logic. It has one parameter to select if the module is used in simulation or in synthesis.

Table 4.12. Reset Sync IP Signal Description

Signal Name	Width	Direction	Description
clk	1	input	Clock signal
pb_in_n	1	input	Signal to be debounced
pb_out	1	output	Debounced signal

Table 4.13. Attribute Summary

Attribute	Values	Description
SIM	0 1	For synthesis For simulation

4.6. OSC for CRE

The oscillator for CRE is used for generating the design clock (75 MHz) and the CRE clocks, which are connected to CRE IP.

Table 4.14. OSC for CRE IP Signal Description

Signal Name	Width	Direction	Description
hf_out_en_i	1	input	Enable port for hf_clock_out_o.
sedc_rst_n_i	1	input	Reset port for SEDC.
hf_clk_out_o	1	output	High frequency clock output, enabled by HFCLK Enable and controlled by HFCLK Divider
cre_clk_o	1	output	CRE block clock output, controlled by CRECLK Enable.
cfg_clk_o	1	output	Configuration clock output, enabled by SEDCLK Enable and controlled by SEDCLK Divider.

Table 4.15. Attribute Summary

Attribute	Values	Description
Fixed Frequency	75 MHz	Frequency is fixed at 75Mhz for this release

5. Detailed Description of Crypto Operations

Data flow direction can be from PCIe to UART or UART to PCIe.

5.1. AES-256 CBC Decryption (PCIe to UART)

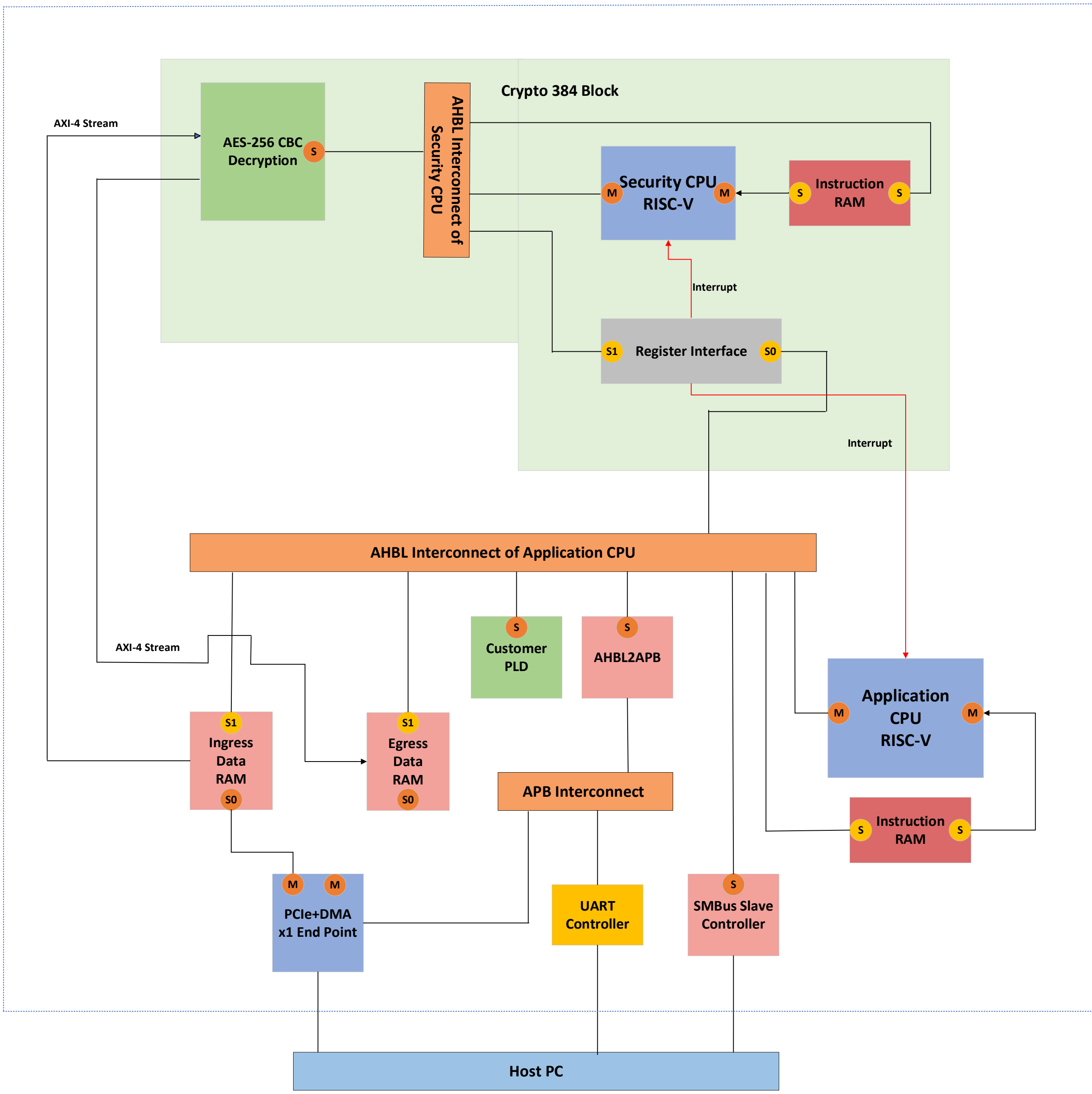


Figure 5.1. AES-256 CBC Decryption (PCIe to UART)

Input values to AES IP are initialization vector (128 bit), key (256 bit), and plain text (length=L). Output value is cipher text (length=L).

5.1.1. DMA Read

The following describes the AES-256 CBC Decryption (PCIe to UART) process:

1. Through SMBus, the Application CPU is directed to perform AES-256 CBC Decryption mode. 256 bit key, 128 bit initialization vector comes through SMBus to the Application CPU.
2. The Application CPU writes the mode register (0x002D_3FFC) with value 0x40 for AES-256 bit CBC Decryption and register address 0x002D_3FF8 with 0x1 for 256 bit key support.
3. The Application CPU writes the 256 bit key required for the operation in the BUF (starting address 0x2D_0180) and 128 bit initialization vector into BUF (starting address 0x2D_01B0).
4. The Application CPU instructs the Host PC to transfer data required for encryption through PCIe and then raises interrupt for the Security CPU (writes '1' to address 0x2C_0014).
5. The Security CPU performs checks for the interrupt and starts reading mode register to get the operation that needs to be performed.

The Host CPU should check register continuously at (@0x00180030). If this is 1, the CPU starts the DMA process.

The DMA_READ process is performed to send data into FPGA.

The following describes the DMA_READ (PC to FPGA) process:

1. Descriptor count is obtained as input from the user.
2. Descriptor data (requestor_id, source address, destination address, descriptor length) is written to the system memory starting from (@0x00181000).
3. DATA_size is obtained as input from the user (bytes in hex). This DATA_size should not be more than (descriptor_count × 512 bytes). If the user enters more DATA_size than (descriptor_count × 512 bytes), the CPU considers only (descriptor_count × 512 bytes) data.

   Note: Here, a maximum of 60 kB (120 descriptors) can be written using the DMA Read operation at once. If more data is required, the DMA Read operation should be performed again.
4. The Host CPU should write descriptor count in register_space at (@0x00180008).
5. (DATA_size divided by 16) is written in register_space at (@0x00180038) by the Host CPU.

   Note: Steps 2, 3, and 4 does not matter.
6. After the completion of steps 2, 3, 4, 5 and 6, using Host CPU, write 0x02 to the register_space address (@0x0018000C) to start DMA_READ.
7. PCIe Endpoint (by itself) starts DMA operation and serves the data.
8. After the data is obtained by the PCIe and sent to application (Ingress RAM), the Host CPU reads at (@0x00180000) if it comes as 0xC0 (indicates completion of DMA_READ).
9. The process continues or an error is printed. The tenth point should be implemented in the same way as DMA status is implemented in previous design.
10. AES operation takes place with help of the internal CPU (security and application) and data is written in EGRESS_RAM.

5.1.2. Application CPU Process

The following describes the process on the Application CPU side:

1. Before starting the process, wait for PCIe to linkup by checking continuously the register (@0x000CA004) for value of 0x1 through the Application CPU.
2. When PCIe provides linkup, the Application CPU writes x01 at (@0x000CA000).
3. Write 0x1 into the register @ 0x1AF008 to indicate the decryption operation.
4. The Application CPU waits until servo_status (@0x002D0278) equals to SERVO_IDLE (0x00) and sets servo_status to SERVO_IN_SERVICE (0x01).
5. The Application CPU writes 0x1 to register @0x002C0004 to enable its interrupt.
6. The Application CPU writes key (@0x002D0180), Initial Vector (@0x002D01B0), and mode of encryption (@0x002D3FFC) to the Register Interface.

   Note: These information comes from the SMBus.
7. The Application CPU sets interrupt for the Security CPU by writing 1 to INT_SET_SEC (@0x002C0014).
8. The Application CPU waits for interrupt by reading INT_STATUS_APP (@0x002C000).
9. If INT_STATUS_APP is 1, the Application CPU instructs the Ingress RAM port to send data to AESIP. Here, Ingress RAM waits if the data is received from PCIe. If the data is received from PCIe, then until this step happens, the data is not sent to AES Write 0x1AF00C address with 0x1 value.
10. The Application CPU clears its interrupt by writing 1 to INT_STATUS_APP (@0x002C000).
11. The Application CPU waits for interrupt by reading INT_STATUS_APP (@0x002C000).
12. If INT_STATUS_APP is 1, the Application CPU starts Egress RAM port to take the output decrypted data into it.
13. Write 0x1Cf01C to 0x1 for changing the Port B of RAM from AHB to AXI.
14. Write 0x1Cf024 to 0x1 for making Port B of RAM to be used for writing.
15. Write 0x1CF00C to 0x1 for taking data into Egress RAM.
16. As the Security CPU configured the IP, it continues to operate the data and provide output data.
17. ORAN Security Enclave AES IP decrypts the data and sends output to Egress RAM port.
18. Read 0x1CF014 to get the DMA Read size, that is, ingress of ‘16 bytes (no .of 128 bit blocks)’ sent from the PCIe to FPGA for Decryption in single iteration.
19. Read 0x1CF018 until it gets the value 0x1. This one is to understand that the complete decrypted data is stored in the Egress RAM and ready to read out from UART.
20. Now Disable the Ingress RAM for sending data to AES.
21. Write 0x1AF00C address with 0x0 value.
22. Set Interrupt to the Security CPU.
23. To read from Egress RAM through UART, Port B of RAM must be changed to AHBL reading.
24. Write 0x1Cf01C to 0x0 for changing the Port B of RAM from AXI to AHBL.
25. Write 0x1Cf024 to 0x0 for making Port B of RAM to be used for reading
26. Starting from 0x1C0000, read out the data from UART until the complete encrypted data comes out.

5.1.3. Security CPU Process

The following describes the process on the Security CPU side:

1. In the Security CPU side, write int_enable_security (@0x002E0010) to '0x01' and then read for INT_STATUS_SECURITY (@0x002E00C) to become 1.
2. If INT_STATUS_SECURITY is 1, the Security CPU reads servo status (@0x002F0278) mode of encryption (@0x002F3FFC).
3. The Security CPU clears its interrupt by writing 1 to INT_STATUS_SECURITY (@0x002E00C).
4. The Security CPU reads for mode (@0x2F3FFC) and servo_status (@0x2F0278).
5. If servo_status reads SERVO_IN_SERVICE (0x01) the Security CPU writes servo_status to SERVO_BUSY (0x02).
6. Based on mode of encryption, the Security CPU reads key (@0x002F0180) and initial vector (@0x002F01B0) from the Register Interface.
7. The Security CPU writes key (@0x00300030), initial vector (@0x00300050) to OSE.
8. The Security CPU configures the OSE with AES_256_DECRYPT by writing 0x00801004 into the register CONFIG (@ 0x0000000c).
9. The Security CPU sets interrupt for the Application CPU by writing 1 to INT_SET_APP (@0x002E0008).
10. The Security CPU waits for AES Interrupt by reading from AES_INT_STATUS_REG (@0x00300004).
11. If there is interrupt for AES, the Security CPU configures the OSE for the second time .
12. Write 0x00801104 into register CONFIG (@ 0x0000000c).
13. The Security CPU also clear AES interrupt by writing 1 to AES_INT_STATUS_CLR (@0x00300060) and it writes 0.
14. The Security CPU sets interrupt for the Application CPU by writing 1 to INT_SET_APP (@0x002E0008).
15. Decryption continues with the data going from the Ingress RAM to AES through AXI stream.
16. The decrypted data goes to the Egress RAM through AXI stream.
17. The Security CPU waits for the interrupt set by the Application CPU. Once the interrupt is received, it clears the Security CPU interrupt as well as the AES interrupt. (Registers are previously provided.)

5.2. AES-256 CBC Encryption (UART to PCIe)

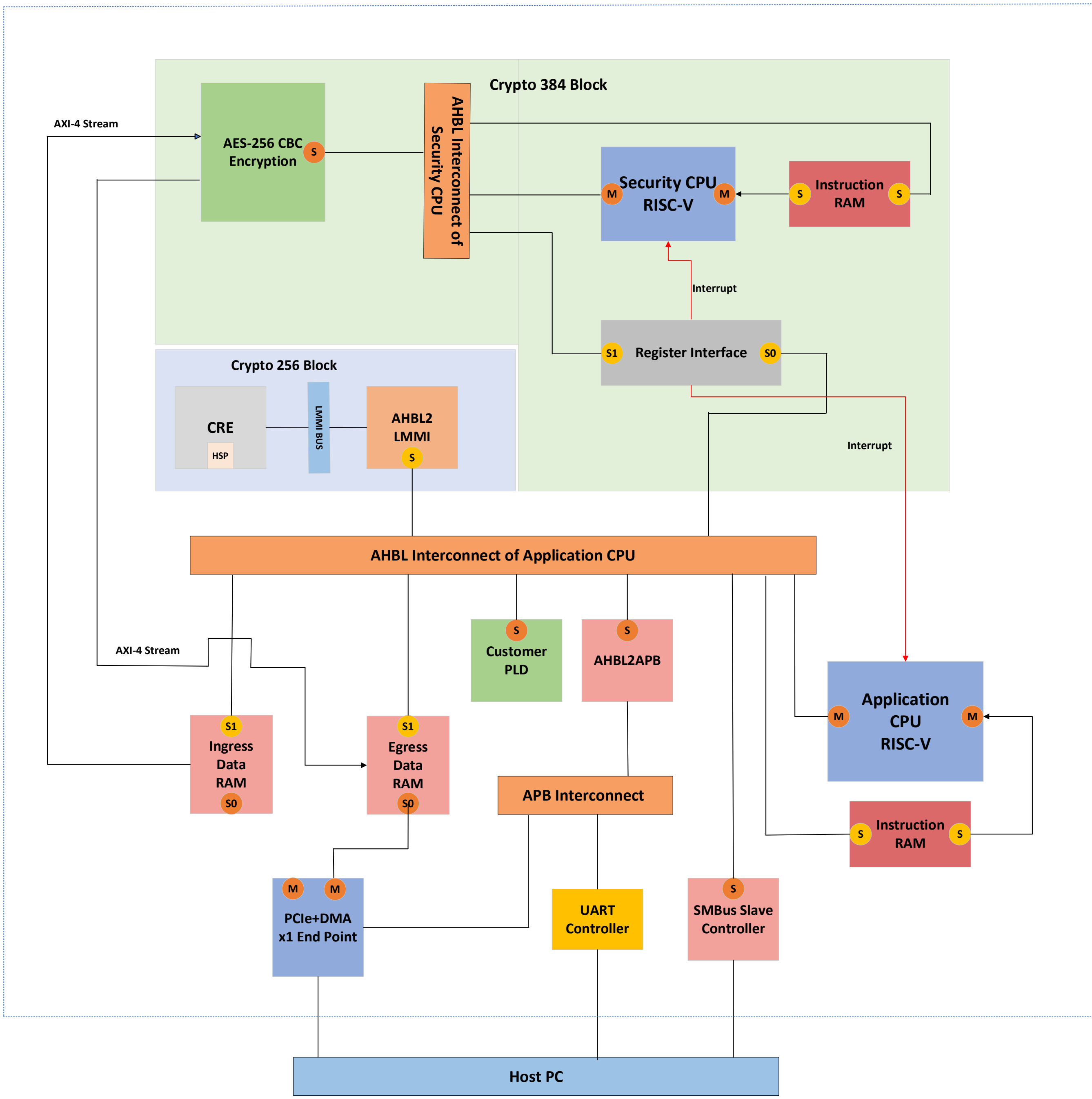


Figure 5.2. AES-256 CBC Encryption (UART to PCIe)

The input plain text is written to the Ingress RAM starting from the base address 0x1A0000.Once written, Port B is switched to AXI stream reading and the AES is performed similar to the AES-256 CBC Decryption (PCIe to UART) section. Once the complete data is stored into Egress RAM, PCIe DMA write operation is performed to get the encrypted data out.

The Host CPU should check register continuously at (@0x00180030). If this is 1, the CPU should start DMA process.

The DMA_WRITE process is performed to acquire data from the FPGA.

5.2.1. DMA Write

The following describes the DMA_WRITE (FPGA to PC) process:

1. The Host CPU reads from register_space address (0x00180034) until it becomes 1. This indicates that the AES processed data is entered into Egress RAM.
2. The Data size is read from register_space address (@0x00180050). The size represents no .of "128 bit" blocks of data.
3. This (Data size ×16) gives the total size in bytes. Based on this descriptor count is calculated and descriptors are formed.
4. Descriptor count = ((Data size ×16)/512) (this should be rounded to upper value.)
5. Descriptors data (requestor_id, source address, destination address, descriptor length) should be written to system memory starting from (@0x00181000).

   Note: Here, a maximum of 60 kB (120 descriptors) can be written using the DMA Read operation at once. If more data is required, the DMA Write operation should be performed again.
6. The Host CPU writes descriptor count in register_space at (@0x00180008).
7. Write 0x01 to the register_space address (@0x0018000C) to start DMA_WRITE.
8. PCIe Endpoint (by itself) starts DMA operation and serves the data.
9. After the data is obtained by application (Egress RAM), the Host CPU reads at (@0x00180000) if it comes as 0x42 (indicates completion of DMA_WRITE).
10. The process continues or an error is printed. The tenth point should be implemented in the same way as they have implemented DMA status in previous design.
11. AES operation takes place with the help of the internal CPUs (Security and Application) and data is written in EGRESS_RAM.

5.2.2. Application CPU Process

The following describes the process on the Application CPU side:

1. Before starting the process, wait for PCIe to linkup by checking continuously the register (@0x000CA004) for value of 0x1 through the Application CPU.
2. When PCIe provides linkup, the Application CPU writes x01 at (@0x000CA000).
3. Write 0x2 into the register @ 0x1AF008 to indicate the encryption operation.
4. DATA size is taken from UART in the form of 'no of bytes'.
5. (DATA size/16) is written into Ingress RAM 0x1Af028.
6. (DATA size/16) is written into Egress RAM 0x1Cf028.
7. The input plain text Is written to the Ingress RAM starting from the base address 0x1A0000 based on data size. This data should be taken from UART.
8. The Application CPU waits until servo_status (@0x002D0278) equals to SERVO_IDLE (0x00) and sets servo_status to SERVO_IN_SERVICE (0x01).
9. The Application CPU writes key (@0x002D0180), Initial Vector (@0x002D01B0) and mode of encryption (@0x002D3FFC) to the Register Interface.
10. The Application CPU writes 0x1 to register @0x002C0004 to enable its interrupt.

    Note: These information comes from the SMBus or key should be generated from CRE IP (procedure is given in Table 21 with Base Address 0x00100000 for CRE IP).
11. The Application CPU sets interrupt for the Security CPU by writing 1 to INT_SET_SEC (@0x002C0014).
12. The Application CPU waits for interrupt by reading INT_STATUS_APP (@0x002C000).
13. If INT_STATUS_APP is 1, the Application CPU instructs the Ingress RAM port to send data to AESIP, that is, start the Ingress RAM.
14. Write 0x1Af01C to 0x1 for changing the Port B of RAM from AHB to AXI.
15. Write 0x1Af024 to 0x0 for making Port B of RAM to be used for reading.

16. Write 0x1AF00C to 0x1 for taking data from the Ingress RAM.
17. The Application CPU clears its interrupt by writing 1 to INT_STATUS_APP (@0x002C000)
18. The Application CPU waits for interrupt by reading INT_STATUS_APP (@0x002C000).
19. If INT_STATUS_APP is 1, the Application CPU starts Egress RAM port to take the output encrypted data into it.
20. Write 0x1Cf01C to 0x1 for changing the Port B of RAM from AHB to AXI.
21. Write 0x1Cf024 to 0x1 for making Port B of RAM to be used for writing.
22. Write 0x1CF00C to 0x1 for taking data into Egress RAM.
23. While the Security CPU has configured the AES IP in update mode, it continues to encrypt the data and provide the encrypted output data.
24. ORAN Security Enclave AES IP decrypts the data and sends output to Egress RAM port.
25. Read 0x1CF018 until it gets the value 0x1. This one is to understand that the complete decrypted data is stored in the Egress RAM and ready to read out from UART.
26. Disable the Ingress RAM for sending data to AES.
27. Write 0x1AF00C address with 0x0 value.
28. Set Interrupt to the Security CPU.

5.2.3. Security CPU

The following describes the process on the Security CPU side:

1. In the Security CPU side, write int_enable_security (@0x002E0010) to ‘0x01’ and then read for INT_STATUS_SECURITY (@0x002E00C) to become 1.
2. If INT_STATUS_SECURITY is 1, the Security CPU reads servo status (@0x002F0278) mode of encryption (@0x002F3FFC).
3. The Security CPU clears its interrupt by writing 1 to INT_STATUS_SECURITY (@0x002E00C).
4. The Security CPU reads for mode (@0x2F3FFC) and servo_status (@0x2F0278).
5. If servo_status reads SERVO_IN_SERVICE (0x01), the Security CPU writes servo_status to SERVO_BUSY (0x02).
6. Then based on mode of encryption, the Security CPU reads key (@0x002F0180) and initial vector (@0x002F01B0) from the Register Interface.
7. The Security CPU writes key (@0x00300030), initial vector (@0x00300050) to OSE.
8. The Security CPU configures the OSE with AES_256_DECRYPT by writing 0x00801004 into register CONFIG (@ 0x0000000c).
9. The Security CPU sets interrupt for Application CPU by writing 1 to INT_SET_APP (@0x002E0008).
10. The Security CPU waits for AES Interrupt by reading from AES_INT_STATUS_REG (@0x00300004).
11. If there is interrupt for AES, the Security CPU configure the OSE for the second time by writing 0x00801104 into register CONFIG (@ 0x0000000c).
12. The Security CPU also clear AES interrupt by writing 1 to AES_INT_STATUS_CLR (@0x00300060) and it writes 0 .
13. The Security CPU sets interrupt for the Application CPU by writing 1 to INT_SET_APP (@0x002E0008).
14. Decryption continues with the data going from the Ingress RAM to AES through AXI stream and the decrypted data goes to the Egress RAM through AXI stream.
15. The Security CPU waits for the interrupt set by the Application CPU. Once interrupt is received, it clears the Security CPU interrupt as well as the AES interrupt. (Registers are already given previously.)

5.3. AES-256 GCM Decryption (PCIe to UART)

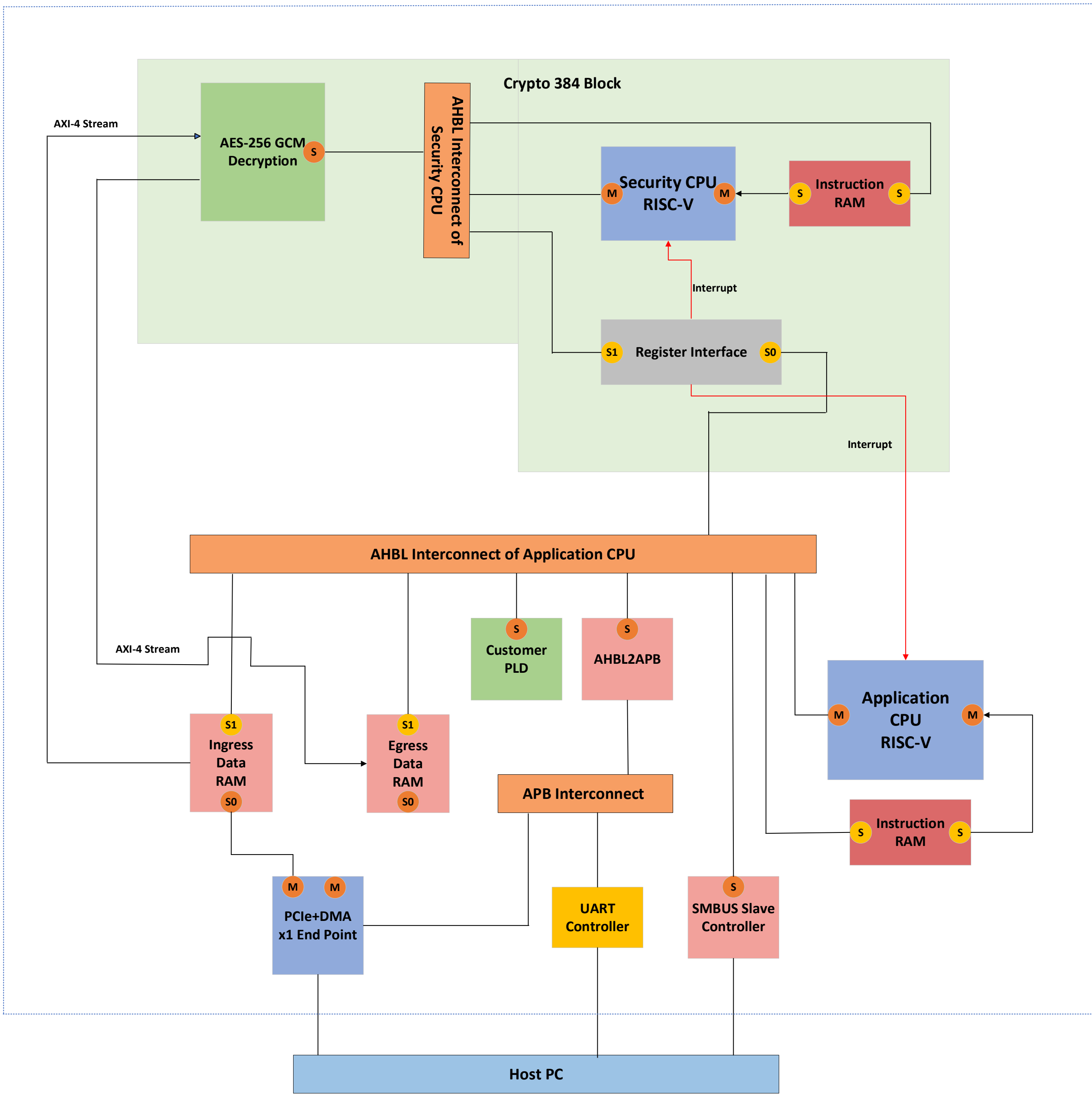


Figure 5.3. AES-256 GCM Decryption (PCIe to UART)

The Host CPU should check register continuously at (@0x00180030). If this is 1, the CPU should start DMA process. The DMA_READ process is performed to send data to the FPGA.

5.3.1. DMA Read

The following describes the DMA_READ (PC to FPGA) process:

1. Descriptor count should be taken as input from user.
2. Descriptor data (requestor_id,source address, destination address,descriptor length) should be written to system memory starting from (@0x00181000).
3. DATA_size should be taken as input from user (bytes in hex). This DATA_size should not be more than (descriptor_count × 512 bytes). If the user enters more DATA_size than (descriptor_count × 512bytes), the CPU considers only (descriptor_count × 512 bytes) data.
4. Host CPU should write descriptor count in register_space at (@0x00180008).

   Note: Here, a maximum of 60 kB (120 descriptors) can be written using the DMA Read operation at once. If more data is required, the DMA Read operation should be performed again.
5. (DATA_size divided by 16) is written in register_space at (@0x00180038) by Host CPU.
6. If cipher text length is not a multiple of 128 bit then we have to write size of cipher text in bits in the register (@0x00180070).
7. Steps 2, 3, and 4 sequence does not matter.
8. After completion of steps 2, 3, 4, 5, and 6 then by Host CPU write 0x02 to the register_space address (@0x0018000C) to start DMA_READ.
9. After this PCIe Endpoint (by itself) starts DMA operation and serve the data.
10. After data is taken by PCIe and sent to application (Ingress RAM), Host CPU should read at (@0x00180000) if it comes as 0xC0 (indicates completion of DMA_READ).
11. The process continues or an error is printed. The tenth point should be implemented in the same way as they have implemented DMA status in previous design.
12. Here, AES operation takes place with help of internal CPUs (Security and Application) and data is written in EGRESS_RAM.

5.3.2. Application CPU Process

The following describes the process on the Application CPU side:

1. Before starting the process, wait for PCIe to linkup by checking continuously the register (@0x000CA004) for value of 0x1 through the Application CPU.
2. When PCIe gives linkup, the Application CPU writes x01 at (@0x000CA000).
3. Write 0x1 into the register @ 0x1AF008 to indicate the decryption operation.
4. The Application CPU waits until servo_status (@0x002D0278) equals to SERVO_IDLE (0x00) and sets servo_status to SERVO_IN_SERVICE (0x01).
5. The Application CPU writes key (@0x002D0180), Initial Vector (@0x002D01B0)(96 BITS) and mode of encryption (@0x002D3FFC) to the Register Interface.
6. The Application CPU writes AADITIONAL_DATA (@0x002D01E0) to the Register Interface.
7. The Application CPU writes 0x1 to register @0x002C0004 to enable its interrupt.
8. The Application CPU sets interrupt for the Security CPU by writing 1 to INT_SET_SEC (@0x002C0014).
9. The Application CPU waits for interrupt by reading INT_STATUS_APP (@0x002C000).
10. If INT_STATUS_APP is 1, the Application CPU tells Ingress RAM port to send data to OSE.
11. Here, Ingress RAM keeps on waiting if the data is not received from PCIe. If the data is received from PCIe already, then until this step happens, data is not sent to AES Write 0x1AF00C address with 0x1 value.
12. Write 0x1Af01C to 0x1 for changing the Port B of Ingress RAM from AHB to AXI.
13. Write 0x1Af024 to 0x0 for making Port B of Ingress RAM to be used for reading.
14. The Application CPU clears its interrupt by writing 1 to INT_STATUS_APP (@0x002C000).

15. The Application CPU waits for interrupt by reading INT_STATUS_APP (@0x002C000).
16. If INT_STATUS_APP is 1, the Application CPU start Egress RAM port to take the output data.
17. Write 0x1CF00C address with 0x1 value.
18. Write 0x1Cf01C to 0x1 for changing the Port B of Egress RAM from AHB to AXI.
19. Write 0x1Cf024 to 0x1 for making Port B of Egress RAM to be used for writing.
20. As the Security CPU has configured the IP, it continues to operate the data and give output data.
21. OSE encrypts the data and send output to Egress RAM port.
22. Read 0x1CF014 to get the DMA DATA size, that is, ingress of '16 bytes (no .of 128 bit blocks)' sent from the PCIe to FPGA for Decryption in single iteration.
23. The Application CPU writes length (AADITIONAL_DATA)||length(CIPHER_TEST) (@0x002D0210) to the Register Interface.

    length (AADITIONAL_DATA) =128 BITS (FIXED FOR NOW)

    length (CIPHER_TEST) (@0x002D0210) = (DMA DATA size)*16*8.

    Note: If cipher text length is not a multiple of 128 bit then we have to read the register (@0x000CA034) and pass as length(CIPHER_TEST)
24. Read 0x1CF018 until it gets the value 0x1.This one is to understand that the complete decrypted data is stored in the Egress RAM.
25. Disable the Ingress RAM from sending data to AES.
26. Write 0x1AF00C address with 0x0 value.
27. The Application CPU sets interrupt for the Security CPU by writing 1 to INT_SET_SEC(@0x002C0014).
28. The Application CPU waits for interrupt by reading INT_STATUS_APP (@0x002C000).
29. If INT_STATUS_APP is 1, the Application CPU reads READ_TAG from the Register Interface (@0x002D0240).
30. The Application CPU sends READ_TAG to UART.
31. For reading from Egress RAM through UART, Port B of RAM must be changed to AHBL reading.
32. Write 0x1Cf01C to 0x0 for changing the Port B of RAM from AXI to AHBL.
33. Write 0x1Cf024 to 0x0 for making Port B of RAM to be used for reading
34. Starting from 0x1C0000, read out the data from UART until complete encrypted data came out.
35. Write 0x1CF00C address with 0x0 value to stop stream of Egress RAM.

5.3.3. Security CPU Process

The following describes the process on the Security CPU side:

1. In the Security CPU side write int_enable_security (@0x002E0010) to '0x01' and then read for INT_STATUS_SECURITY (@0x002E00C) to become 1.
2. If INT_STATUS_SECURITY is 1, the Security CPU reads servo status (@0x002F0278) mode of encryption (@0x002F3FFC).
3. The Security CPU clears its interrupt by writing 1 to INT_STATUS_SECURITY (@0x002E00C).
4. The Security CPU reads for mode (@0x2F3FFC) and servo_status (@0x2F0278).
5. If servo_status reads SERVO_IN_SERVICE (0x01), the Security CPU writes servo_status to SERVO_BUSY (0x02).
6. Based on the mode of encryption, the Security CPU reads key (@0x002F0180) and initial vector (@0x002F01B0) from the Register Interface.
7. The Security CPU writes key (@0x00300030), initial vector (@0x00300050) to OSE.
8. The Security CPU configures the OSE with AES_256_GCM_DECRYPT by writing 0x00805005 into register CONFIG (@ 0x0000000C).
9. The Security CPU gives start pulse to the AES GCM IP by writing 0x1 to register @0x00000008 with base address 0X00300000.

10. The Security CPU waits for AES Interrupt rising edge on irq_o.
11. If there is interrupt for AES, the Security CPU configures the OSE for the second time with GCM_GHASH By writing 0x00805105 into register CONFIG (@ 0x0000000c).
12. The Security CPU reads ADDITIONAL_DATA (@0x002F01E0) from the Register Interface.
13. The Security CPU writes ADDITIONAL_DATA (@0x00300010) to OSE.

    Note: AES IP is kept under AES GCM_GHASH mode until all the AAD data are sent.
14. The Security CPU gives start pulse to the AES GCM IP by writing 0x1 to register @0x00000008 with base address 0X00300000.
15. The Security CPU waits for AES Interrupt rising edge on irq_o.
16. The Security CPU configures the AES GCM IP with AES_256_GCM_DECRYPT_UPDATE mode by writing 0x00805205 into CONFIG register (@ 0x0030000c).
17. The Security CPU sets interrupt for the Application CPU by writing 1 to INT_SET_APP (@0x002E0008).
18. The Security CPU waits for AES Interrupt rising edge on irq_o.
19. The Security CPU sets interrupt for the Application CPU by writing 1 to INT_SET_APP (@0x002E0008).
20. The Security CPU waits for interrupt by reading INT_STATUS_SEC (@0x002E00C).
21. If INT_STATUS_SEC is 1, the Security CPU configures the AES GCM IP in GCM_GHASH mode by writing 0x00805205 into CONFIG register (@ 0x0030000c).
22. The Security CPU reads len (AADITIONAL_DATA)||len(CIPHER_TEST) (@0x002F0210) from the Register Interface.
23. The Security CPU writes len (AADITIONAL_DATA)||len(CIPHER_TEST) (@0x00300010) to OSE.
24. The Security CPU gives start pulse to the AES GCM IP by writing 0x1 to register @0x00000008 with base address 0X00300000.
25. The Security CPU waits for AES Interrupt rising edge on irq_o.
26. If there is interrupt for AES, the Security CPU configures the OSE with GCM_FINISH.
27. The Security CPU gives start pulse to the AES GCM IP by writing 0x1 to register @0x00000008 with base address 0X00300000.
28. The Security CPU waits for AES Interrupt rising edge on irq_o.
29. If there is interrupt for AES, the Security CPU reads READ_TAG (@0x00300020) from OSE.
30. The Security CPU writes READ_TAG (@0x002F0240) to the Register Interface.
31. The Security CPU sets interrupt for Application CPU by writing 1 to INT_SET_APP (@0x002E0008).

5.4. AES-256 GCM Encryption (UART to PCIe)

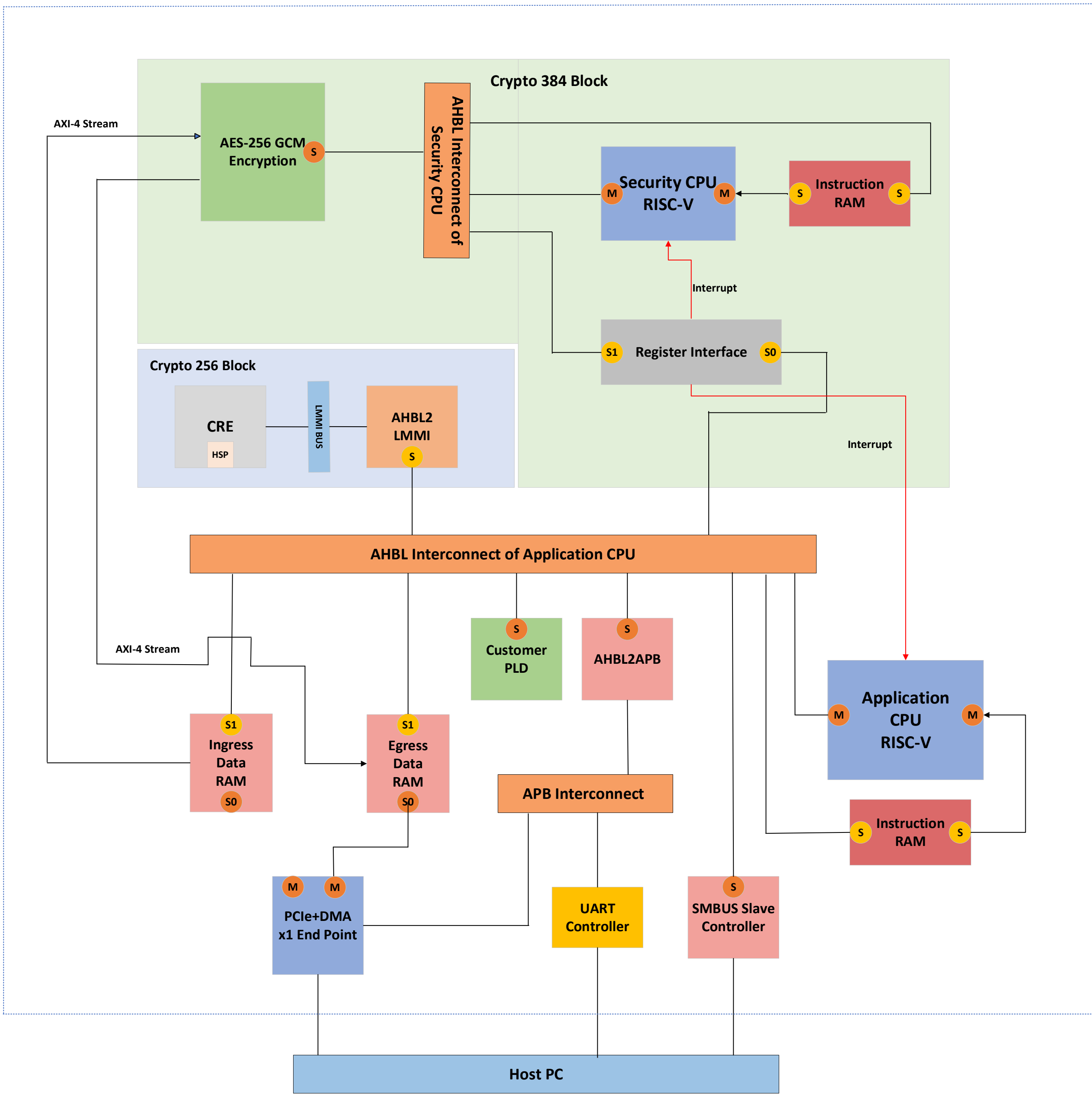


Figure 5.4. AES-256 GCM Encryption (UART to PCIe)

The Host CPU should check register continuously at (@0x00180030). If this is 1, the CPU should start DMA process. The DMA_WRITE process is performed to obtain data from the FPGA.

5.4.1. DMA Write

The following describes the DMA_WRITE (FPGA to PC) process:

1. The Host CPU reads from the register_space address (0x00180034) until it becomes 1. This indicates that the AES processed data is entered into the Egress RAM.
2. Data size is read from register_space address (@0x00180050). The size represents the number of 128 bit blocks of data.
3. This (Data size *16) gives the total size in bytes. Based on this, the descriptor count is calculated and descriptors are formed.
4. Descriptor count = ((Data size *16)/512). This is rounded to the upper value.
5. Descriptor data (requestor_id, source address, destination address, descriptor length) is written to the system memory starting from (@0x00181000).
6. The Host CPU writes the descriptor count in register_space at (@0x00180008).

   Note: Here, a maximum of 60 kB (120 descriptors) can be written using the DMA Read operation at once. If more data is required, the DMA Write operation should be performed again.
7. Write 0x01 to the register_space address (@0x0018000C) to start DMA_WRITE.
8. PCIe Endpoint (by itself) starts the DMA operation and serves the data.
9. After obtaining data from the application (Egress RAM), the Host CPU reads at (@0x00180000) if it comes as 0x42. This indicates completion of DMA_WRITE.
10. The process continues or an error is printed. The tenth point should be implemented in the same way as they have implemented DMA status in previous design.
11. AES operation takes place with help of internal CPUs (Security and Application) and data is written in EGRESS_RAM.

5.4.2. Application CPU Process

The following describes the process on the Application CPU side:

1. Before starting the process, wait for PCIe to linkup by checking continuously the register (@0x000CA004) for value of 0x1 through the Application CPU.
2. When PCIe provides linkup, the Application CPU writes x01 at (@0x000CA000).
3. Write 0x2 into the register @ 0x1AF008 to indicate the Encryption operation.
4. DATA size is taken from UART in the form of 'no of bytes'.

   Note: Maximum DATA size currently supported in this case is up to 60 kB.
5. (DATA size/16) should be written into Ingress RAM 0x1Af028.
6. (DATA size/16) should be written into Egress RAM 0x1Cf028.

   Note: These values should be rounded to upper value.
7. The input plain text is written to the Ingress RAM starting from the base address 0x1A0000 based upon data size. This data should be taken from UART.
8. The Application CPU waits until servo_status (@0x002D0278) equals to SERVO_IDLE (0x00) and sets servo_status to SERVO_IN_SERVICE (0x01).
9. The Application CPU writes key (@0x002D0180), Initial Vector (@0x002D01B0), and mode of encryption (@0x002D3FFC) to the Register Interface.
10. The Application CPU writes AADITIONAL_DATA (@0x002D01E0), len (AADITIONAL_DATA)||len(CIPHER_TEST) (@0x002D0210) to the Register Interface.

    Note: Additional data length and cipher text length should be in bits.
11. The Application CPU writes 0x1 to register @0x002C0004 to enable its interrupt.
12. The Application CPU sets interrupt for the Security CPU by writing 1 to INT_SET_SEC (@0x002C0014).
13. The Application CPU waits for interrupt by reading INT_STATUS_APP (@0x002C000).
14. If INT_STATUS_APP is 1, the Application CPU instructs the Ingress RAM port to send data to OSE.

15. Write 0x1AF00C address with 0x1 value.
16. Write 0x1Af01C to 0x1 for changing the Port B of Ingress RAM from AHB to AXI.
17. Write 0x1Af024 to 0x0 for making Port B of Ingress RAM to be used for reading.
18. The Application CPU clears its interrupt by writing 1 to INT_STATUS_APP (@0x002C000)
19. The Application CPU waits for interrupt by reading INT_STATUS_APP (@0x002C000).
20. If INT_STATUS_APP is 1, the Application CPU starts Egress RAM port to take the output data.
21. Write 0x1CF00C address with 0x1 value.
22. Write 0x1Cf01C to 0x1 for changing the Port B of Egress RAM from AHB to AXI.
23. Write 0x1Cf024 to 0x1 for making Port B of Egress RAM to be used for writing.
24. As the Security CPU has configured the AES IP in update mode, it continues to encrypt the data and provide the encrypted output data.
25. OSE encrypts the data and sends output to Egress RAM port.
26. The Application CPU writes length (AADITIONAL_DATA)||length(CIPHER_TEST) (@0x002D0210) to the Register Interface.

    length (AADITIONAL_DATA) =128 BITS(FIXED FOR NOW)

    length (CIPHER_TEST) (@0x002D0210) = (DATA size)*8.
27. Read 0x1CF018 until it gets the value 0x1. The completely decrypted data is stored in the Egress RAM.
28. If the condition is satisfied, the Application CPU sets interrupt for the Security CPU by writing 1 to INT_SET_SEC (@0x002C0014).
29. Disable the Ingress RAM from sending data to AES.
30. Write 0x1AF00C address with 0x0 value.
31. The Application CPU waits for interrupt by reading INT_STATUS_APP (@0x002C000).
32. If INT_STATUS_APP is 1, the Application CPU reads READ_TAG from the Register Interface (@0x002D0240).
33. The Application CPU sends READ_TAG to UART.
34. Write 0x1CF00C address with 0x0 value to stop AXI stream of Egress RAM.

5.4.3. Security CPU Process

The following describes the process on the Security CPU side:

1. In the Security CPU side write int_enable_security (@0x002E0010) to '0x01' and then read for INT_STATUS_SECURITY (@0x002E00C) to become 1.
2. If INT_STATUS_SECURITY is 1, the Security CPU reads servo status (@0x002F0278) mode of encryption (@0x002F3FFC).
3. The Security CPU clears its interrupt by writing 1 to INT_STATUS_SECURITY (@0x002E00C).
4. The Security CPU reads for mode (@0x2F3FFC) and servo_status (@0x2F0278).
5. If servo_status reads SERVO_IN_SERVICE (0x01) the Security CPU writes servo_status to SERVO_BUSY (0x02).
6. Based on mode of encryption, the Security CPU reads key (@0x002F0180) and initial vector (@0x002F01B0) from the Register Interface.
7. The Security CPU writes key (@0x00300030), initial vector (@0x00300050) to OSE.
8. The Security CPU configures the OSE with AES_256_GCM_ENCRYPT by writing 0x00805004 into register CONFIG (@ 0x0000000C).
9. The Security CPU gives start by writing 0x1 @0x00000008.
10. The Security CPU waits for AES Interrupt by reading from AES_INT_STATUS_REG (@0x00300004).
11. If there is interrupt for AES, the Security CPU configures the OSE for the second time with GCM_GHASH.
12. Write 0x00805104 into register CONFIG (@ 0x0000000c).
13. The Security CPU reads ADDITIONAL_DATA (@0x002F01E0) from the Register Interface.

14. The Security CPU writes ADDITIONAL_DATA (@0x00300010) to OSE.
15. The Security CPU gives start by writing 0x1 @0x00000008.
16. The Security CPU waits for AES Interrupt by reading from AES_INT_STATUS_REG (@0x00300004).

    Note: AES IP is kept under AES GCM_GHASH mode until all the AAD data is sent.
17. If there is interrupt for AES, Security CPU also clears AES interrupt by writing 1 to AES_INT_STATUS_CLR (@0x00300060) and it writes 0.
18. The Security CPU configures the OSE with AES_256_GCM_ENCRYPT_UPDATE by writing 0x00805204 into register CONFIG (@ 0x0000000c).
19. The Security CPU sets interrupt for the Application CPU by writing 1 to INT_SET_APP (@0x002E0008).
20. The Security CPU waits for AES Interrupt by reading from AES_INT_STATUS_REG (@0x00300004).
21. If there is interrupt for AES, the Security CPU sets interrupt for Application CPU by writing 1 to INT_SET_APP (@0x002E0008).
22. The Security CPU waits for interrupt by reading INT_STATUS_SEC (@0x002E00C).
23. If INT_STATUS_SEC is 1, the Security CPU configures the OSE with GCM_GHASH by writing 0x00805204 into register CONFIG (@ 0x0000000c).
24. The CPU reads len (AADITIONAL_DATA)||len(CIPHER_TEST) (@0x002F0210) from the Register Interface.
25. The Security CPU writes len (AADITIONAL_DATA)||len(CIPHER_TEST) (@0x00300010) to OSE.
26. The Security CPU gives start by writing 0x1 @0x00000008.
27. The Security CPU waits for AES Interrupt by reading from AES_INT_STATUS_REG (@0x00300004).
28. If there is interrupt for AES, the Security CPU configure the OSE with GCM_FINISH.
29. The Security CPU gives start by writing 0x1 @0x00000008.
30. The Security CPU waits for AES Interrupt by reading from AES_INT_STATUS_REG (@0x00300004).
31. If there is interrupt for AES, the Security CPU reads READ_TAG (@0x00300020) from OSE.
32. The Security CPU writes READ_TAG (@0x002F01E0) to the Register Interface.
33. The Security CPU sets interrupt for the Application CPU by writing 1 to INT_SET_APP (@0x002E0008).
34. The Security CPU also clear AES interrupt by writing 1 to AES_INT_STATUS_CLR (@0x00300060) and it writes 0.
35. The Security CPU sets interrupt for the Application CPU by writing 1 to INT_SET_APP (@0x002E0008).

5.5. SHA384 Authentication (PCIe to UART)

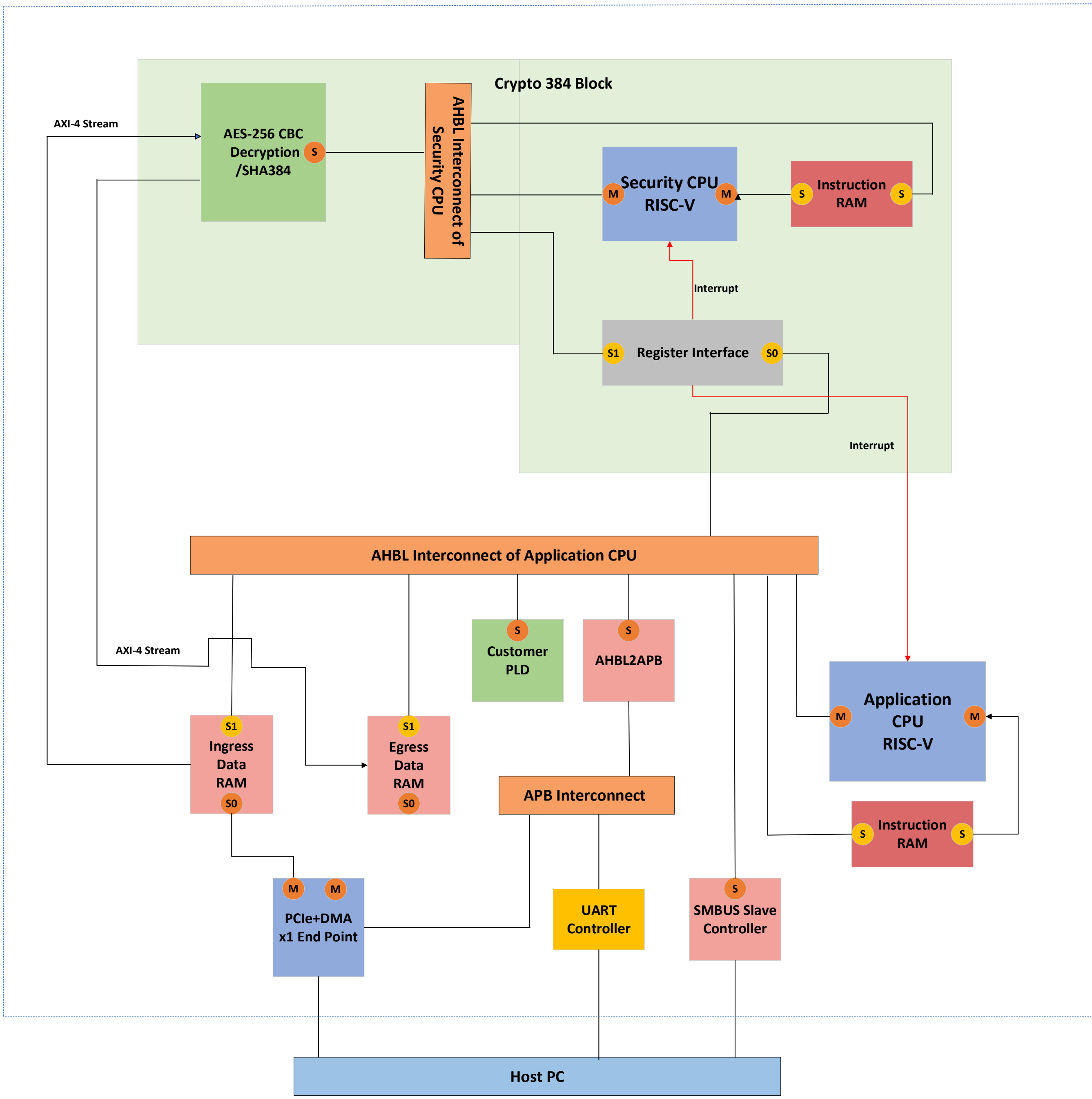


Figure 5.5. SHA384 Authentication (PCIe to UART)

SHA384 Authentication can be done with the help of the Application CPU, Security CPU, PCIe Endpoint, Ingress RAM, Egress RAM, Register Interface and SHA2 IP.

For this SHA Authentication, plain text is initially encrypted using AES-256-CBC mode in the Host PC and the encrypted data is sent over the PCIe using DMA Read to the Ingress RAM. From here, data goes to AES CBC Decryption with the key and IV. After decryption, the data is written to Egress RAM.

Up to here, the flow is the same as the AES-256 CBC Decryption (PCIe to UART). After that, on this decrypted data SHA384 needs to be performed using the Application CPU, Security CPU, and Register Interface in the following way.

5.5.1. PCIe DMA Read

Similar to AES-256 CBC Decryption (PCIe to UART). In addition to that, SHA Data length(in bits) needs to be written into 0x180028 register along with AES size register(0x180038) which is in terms of 128 bit blocks.

5.5.2. Application CPU Process

The following describes the process on the Application CPU side:

1. Follow steps 1 to 17 in the AES-256 CBC Decryption (PCIe to UART) section. To check whether the data is ready and Egress Port B must be switched to AHBL reading. It is same as Application CPU's steps 1 to 17 in the AES-256 CBC Decryption (PCIe to UART) section.
2. Write 0x1Cf01C to 0x0 for changing the Port B of Egress RAM from AXI to AHBL.
3. Write 0x1Cf024 to 0x0 for making Port B of Egress RAM to be used for reading
4. Application CPU writes to the mode register (address= 0x2D3FFC) of the Register Interface with the value 0x35 for SHA384 message digest Authentication.
5. Application CPU writes to the SHA SOURCE register (address = 0x2D3FF4) with the value 0x00.
6. Application CPU writes to the int_enable_app (0x2C004) with 0x01.
7. Read 0xCA00C to know the SHA Data Length (in bits).
8. Write SHA Data Length to register 0x2D027C in the Register Interface so that the Security CPU also knows the size of Plain Text data to be hashed.
9. The Application CPU writes 0x0 in ping and pong buffer ready registers.
10. The Application CPU reads the ping buffer ready from (0x2D0270) to know whether it is in LOW.
11. The Application CPU reads 1K block of data from Egress RAM address (0x1C0000) which is incremented by four offsets and sends it to ping data of the Register Interface starting from (0x2D0000).
12. The Application CPU sets ping status (0x2D0270) to 1.
13. The Application CPU reads the pong buffer ready from (0x2D0274) to know whether it is in LOW.
14. The Application CPU reads 1K block of data from Egress RAM (0x1C0080) which is incremented by four offsets and sends it to pong data of the Register Interface (0x2D0080).
15. The Application CPU sets pong status (0x2D0274) to 1.
16. The Application CPU sets interrupt for the Security CPU by writing 1 to INT_SET_SECURITY (@0x002C0014).
17. If the data is more than 2 kB (2048 bits), then it follows the steps for next ping and pong data and change the Egress RAM data address accordingly.
18. Here check the ping ready bit until 0 and write ping block of data. Same Ready Check for the pong block data
19. After the final data is sent, wait for the Application CPU interrupt status, and once it comes, read the data out from OUTPUT BUFFER(starting address @0x2D0100) and print through UART.

5.5.3. Security CPU Side Process

The following describes the process on the Security CPU side:

1. In the Security CPU side write int_enable_security (0x2E0010) to '0x01' and then read for int_status_security (0x2E000C) to become 1.
2. Follow steps 1 to 17 in the AES-256 CBC Decryption (PCIe to UART) section.
3. Then if the value of int_status_register becomes 1, it clear the interrupt and then read for mode (@0x2F3FFC) , SHA source (@0x2F3FF4) and servo status (@0x2F0278), SHA input message length(0x2F027C).
4. If mode registers reads value 0x35 then configure the value 0x304 (HASH_INITIAL) to the register 0x0031000C (Configuration register).
5. The Security CPU reads the ping ready buffer (@0x002F0270) until it reaches the value 0x01.
6. If ping ready buffer is HIGH then the data from ping block (@0x2F0000) of the Register Interface to SHA IP (@0x310020).
7. The Security CPU writes control register (@0x00310008) to 0x1.
8. It sets ping ready buffer (@0x002F0270) to 0x00.
9. The Security CPU waits for the SHA interrupt from sha_int_status_register (@0x310004) .
10. If the data input is more than 1Kbits, configure the value 0x305 (SHA384 update) to the Configuration register(0x31000C).
11. The Security CPU reads the pong ready buffer (@0x002F0274) until it reaches the value 0x01.
12. If pong ready buffer is HIGH then the data from pong block (@0x2F0080) of the Register Interface to SHA IP (@0x310020).
13. The Security CPU writes control register (@0x00310008) to 0x1.
14. It sets pong ready buffer (@0x002F0274) to 0x0.
15. The Security CPU waits for the SHA interrupt from sha_int_status_register (@0x310004) .
16. If more data is there, check for ping and pong Ready bits and if they are 1, read data from them and send to the SHA IP
17. For last block of SHA data, HASH FINISH (0x306) has to be written to the configuration register of Hash IP
18. Write the message length in SHA IP register (starting from 0x310010) and the last block of input data to 0x310020 register in Hash IP
19. Give the start command to SHA IP (writes control register (@0x00310008) to 0x1)
20. After complete transaction of data to the SHA IP, the Security CPU waits for the interrupt from the SHA IP, reads the data from the SHA IP of address (0x310134 to 0x310160) and sends it to output buffer (0X2F0100) of the Register Interface.
21. The Security CPU sets the interrupt for the Application CPU int_set_application (0x2E0008) to 0x01.

5.6. SHA384 Message Digest Generation (UART to PCIe)

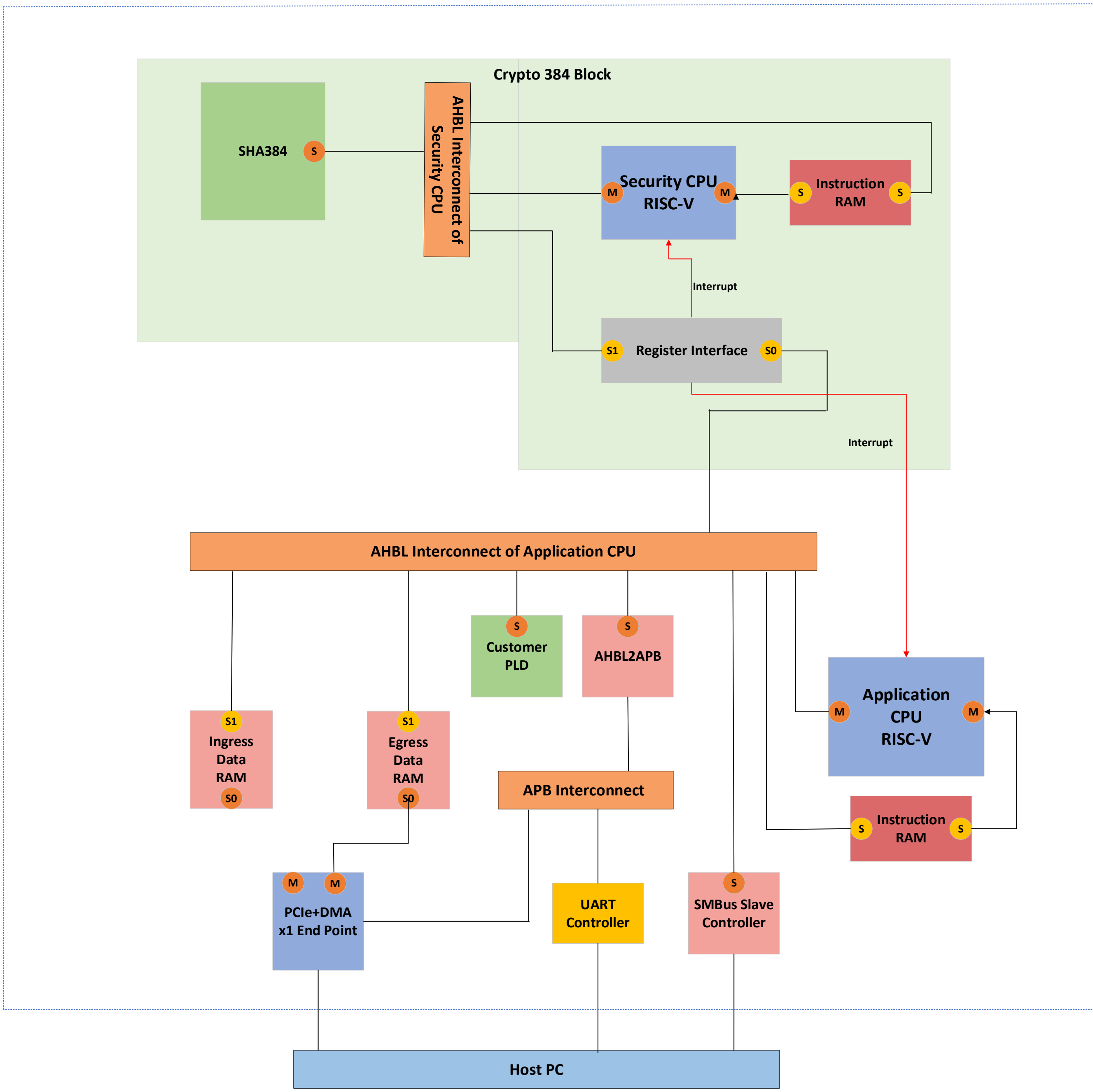


Figure 5.6. SHA384 Message Digest Generation (UART to PCIe)

The input plain text is written to the Ingress RAM starting from the base address 0x1A0000. Once completed, Port B is switched to AHBL reading and the SHA is performed similar to the SHA384 Authentication (PCIe to UART) section.

Once complete data is stored in the Egress RAM, PCIe DMA write operation is started to get the Message digest data out using only one descriptor similar to the AES-256 CBC Encryption (UART to PCIe) section based on 0x180034.

5.6.1. Application CPU Process

The following describes the process on the Application CPU side:

1. Before starting the process, wait for PCIe to linkup by checking continuously the register (@0x000CA004) for value of 0x1 through the Application CPU.
2. When PCIe provides linkup, the Application CPU writes x01 at (@0x000CA000).
3. DATA size is taken from UART in the form of 'no of bits'.
4. The input plain text is written to the Ingress RAM starting from the base address 0x1A0000 based upon data size. This data should be taken from UART.
5. Write 0x1Af01C to 0x0 for changing the Port B of Ingress RAM from AXI to AHBL.
6. Write 0x1Af024 to 0x1 for making Port B of Ingress RAM to be used for writing.
7. Write UART plain text data into Ingress RAM based upon the data size.
8. Write 0x1AF024 to 0x0 for making Port B of Ingress RAM to be used for reading.
9. Write 0x1AF00C to 0x1 for start Ingress RAM read.
10. The Application CPU waits until servo_status (@0x002D0278) equals to SERVO_IDLE (0x00) and sets servo_status to SERVO_IN_SERVICE(0x01) and write 0x35 to mode register 0x002D3FFC,data size is written into 0x2D0270.
11. The Application CPU sets interrupt for the Security CPU by writing 1 to INT_SET_SEC (@0x002C0014).
12. The Application CPU waits for interrupt by reading INT_STATUS_APP (@0x002C000).
13. If INT_STATUS_APP is 1 then clears its interrupt by writing 1 to INT_STATUS_APP (@0x002C000).
14. The Application CPU starts read data from the Ingress RAM starting address 0x1A0000.
15. The Application CPU writes 0x0 to ping and pong buffer ready registers.
16. The Application CPU writes 0x0 in ping and pong buffer ready registers.
17. The ping buffer ready is read from (0x2D0270) to know whether it is in LOW.
18. The Application CPU reads 1K block of data from the Ingress RAM address (0x1A0000), which is incremented by four offsets and sends it to ping data of the Register Interface starting from (0x2D0000).
19. The Application CPU sets ping status (0x2D0270) to 1.
20. It reads the pong buffer ready from (0x2D0274) to know whether it is in LOW.
21. It reads 1K block of data from the Ingress RAM (0x1A0080), which is incremented by four offsets and sends it to the pong data of the Register Interface (0x2D0080).
22. The Application CPU sets pong status (0x2D0274) to 1.
23. The Application CPU sets interrupt for the Security CPU by writing 1 to INT_SET_SECURITY (@0x002C0014).
24. If the data is more than 2 kB (2048 bits), it follows the steps for the next ping and pong data and changes the Egress RAM data address accordingly.
25. Check the ping ready bit until 0 and write ping block of data. Same Ready Check for the pong block data
26. After the final data is sent, wait for the Application CPU interrupt status. Once it arrives, read the data from the OUTPUT BUFFER (starting address @0x2D0100) and write to Egress RAM (starting with 0x1C0000).
27. For the Host PC to initiate DMA write transaction, write 0x1 followed by 0x0 to 0x1CF020.

5.6.2. Security CPU Process

The following describes the process on the Security CPU side:

1. In the Security CPU side, write int_enable_security (0x2E0010) to '0x01' and then read for int_status_security (0x2E000C) to become 1.
2. If the value of int_status_register becomes 1, it clears the interrupt and then reads for mode (@0x2F3FFC), SHA source (@0x2F3FF4), servo status (@0x2F0278), and SHA input data length(0x2F027C).
3. If mode registers read value 0x35, configure the value 0x304 (HASH_INITIAL) to the register 0x0031000C (Configuration register).
4. The Security CPU reads the ping buffer ready (@0x002F0270) until it reaches the value 0x01.
5. If the ping buffer ready value is 0x1, the data from the ping block (starting address @0x2F0000) of the Register Interface is read by the Security CPU and given to SHA IP (@0x310020).
6. The Security CPU writes control register (@0x00310008) to 0x1.
7. It sets ping ready buffer (@0x002F0270) to 0x00.
8. The Security CPU waits for the SHA interrupt from sha_int_status_register (@0x310004).
9. If the data input is more than 1 Kbit, configure the value 0x305 (SHA384 update) to the configuration register (0x31000C).
10. The Security CPU reads the pong ready buffer (@0x002F0274) until it reaches the value 0x01.
11. If pong ready buffer is HIGH, then the data from pong block (@0x2F0080) of the Register Interface to SHA IP (@0x310020).
12. The Security CPU writes control register (@0x00310008) to 0x1.
13. It sets pong ready buffer (@0x002F0274) to 0x0.
14. The Security CPU waits for the SHA interrupt from sha_int_status_register (@0x310004).
15. If more data is there, check for ping and pong ready bits. If they are 1, read the data and send it to the SHA IP.
16. For last block of SHA data, HASH FINISH (0x306) has to be written to the configuration register of Hash IP.
17. Write the message length in SHA IP register (starting from 0x310010) and the last block of input data to 0x310020 register in Hash IP.
18. Then give the start command to SHA IP (writes control register (@0x00310008) to 0x1).
19. After complete transaction of data to the SHA IP the Security CPU waits for the interrupt from the SHA IP, reads the data from the SHA IP of address (0x310134 to 0x310160) and sends it to output buffer (0X2F0100) of the Register Interface.
20. The Security CPU sets the interrupt for the Application CPU int_set_application (0x2E0008) to 0x01.

5.7. SHA384 Authentication (PCIe to UART using GCM Decryption)

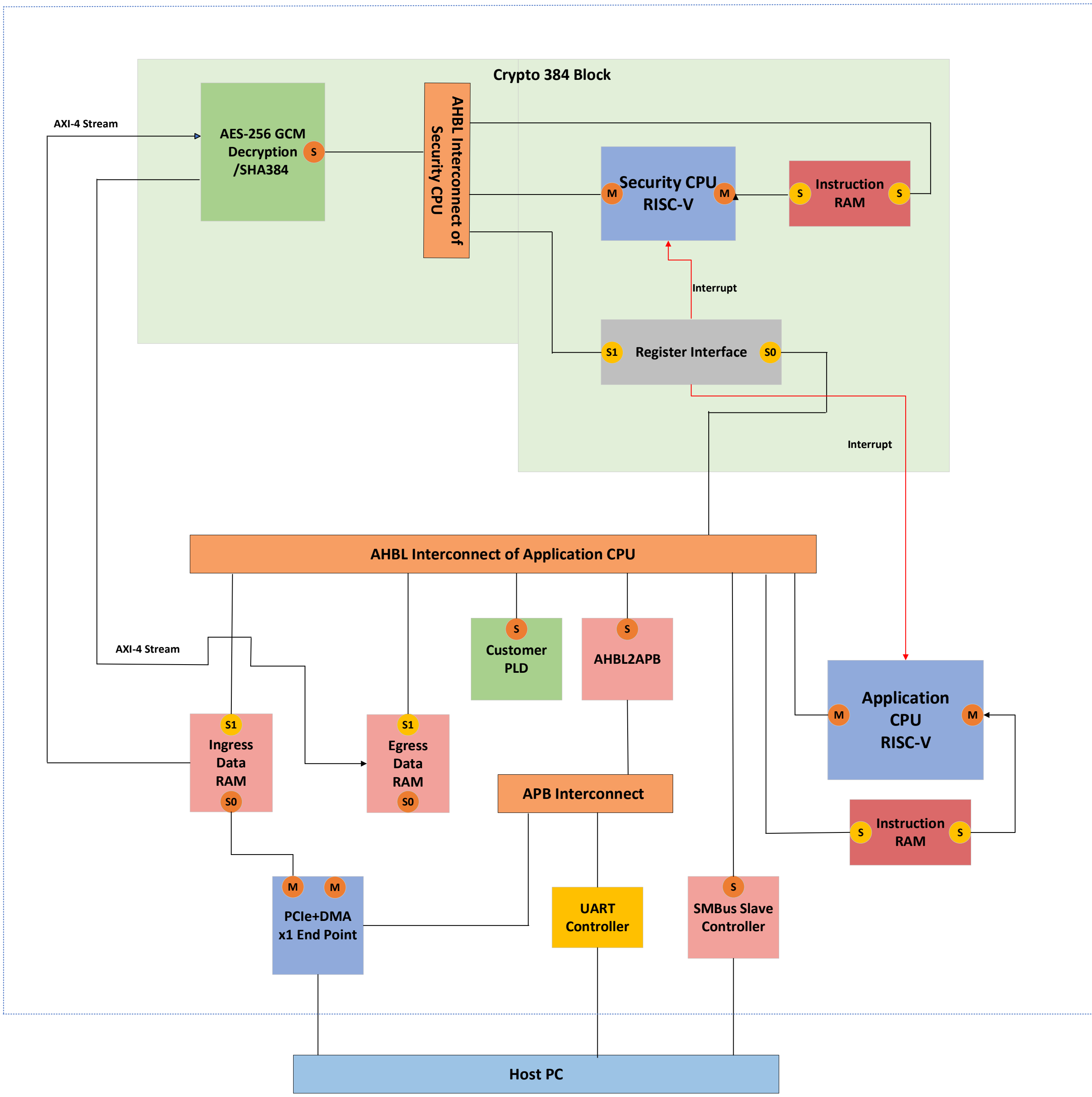


Figure 5.7. SHA384 Authentication (PCIe to UART using GCM Decryption)

SHA384 Authentication can be done with the help of the Application CPU, Security CPU, PCIe Endpoint, Ingress RAM, Egress RAM, Register Interface, and SHA2 IP.

For this SHA Authentication, plain text is initially encrypted using AES-256-GCM mode in the Host PC and the encrypted data is sent over the PCIe using DMA Read to the Ingress RAM. From here, data goes to AES GCM Decryption with the key and IV. After decryption, the data is written to Egress RAM. Up to here, the flow is the same as the AES Decryption explained in the AES-256 GCM Decryption (PCIe to UART) section.

On the decrypted data, SHA384 needs to be performed using the Application CPU, Security CPU, and Register Interface in the following way.

5.7.1. PCIe DMA Read

The PCIe DMA Read is similar to AES-256 GCM Decryption (PCIe to UART). In addition, SHA Data length (in bits) needs to be written into 0x180028 register along with AES size register(0x180038), which is in terms of 128 bit blocks.

5.7.2. Application CPU Process

The following describes the process on the Application CPU side:

1. Before starting the process, wait for PCIe to linkup by checking continuously the register (@0x000CA004) for value of 0x1 through the Application CPU.
2. When PCIe provides linkup, the Application CPU writes x01 at (@0x000CA000).
3. The Application CPU waits until servo_status (@0x002D0278) equals to SERVO_IDLE (0x00). It sets servo_status to SERVO_IN_SERVICE(0x01) and configures MODE_REG (0x002D3FFC) to 0x42 (AES-256GCM_DEC_MODE).
4. The Application CPU writes key (@0x002D0180), Initial Vector (@0x002D01B0), and mode of encryption (@0x002D3FFC) to the Register Interface.
5. The Application CPU configures interrupt enable register (0x002C0004) by writing 0x01.
6. The Application CPU sets interrupt to the Security CPU by writing 1 to INT_SET_SEC (@0x002C0014).
7. The Application CPU waits for interrupt by reading INT_STATUS_APP (@0x002C000).
8. If INT_STATUS_APP is 1, the Application CPU tells the Ingress RAM port to send data to OSE.
9. The Ingress RAM waits for the data from PCIe in the DMA Read operation.

   Note: Even If the data is received from PCIe before step 8, the data is not sent to AES until step 8 is completed.
10. Write 0x1AF00C address with 0x1 value to start the Ingress RAM.
11. Write 0x1Cf01C to 0x1 for changing the Port B of the Egress RAM from AHB to AXI.
12. Write 0x1Cf024 to 0x1 for making Port B of the Egress RAM to be used for writing.
13. Write 0x1CF00C to 0x1 for start egress RAM.
14. To check the DMA size, read the Egress RAM register (0x1CF014).
15. Poll the data ready register (0x1CF018) of the Egress RAM until the valid signal becomes 1.
16. Write 0x001AF00C to 0x0 for stopping AXI stream in the Ingress RAM for more than one iteration.
17. Write 0x001CF00C to 0x0 for stopping AXI stream in the Egress RAM for more than one iteration.
18. Write 0x1Cf01C to 0x0 for changing the Port B of the Egress RAM from AXI to AHBL.
19. Write 0x1Cf024 to 0x0 for making Port B of the Egress RAM to be used for reading.
20. The Application CPU writes to the mode register (address= 0x2D3FFC) of the Register Interface with the value 0x35 for SHA384 message digest Authentication.
21. The Application CPU writes to the SHA SOURCE register (address = 0x2D3FF4) with the value 0x00.
22. The Application CPU Reads 0xCA00C to know the SHA Data Length(in bits).
23. Write SHA Data Length to register 0x2D027C in the Register Interface so that the Security CPU also knows the size of plain text data to be hashed
24. The Application CPU initially writes 0x0 in the ping and pong buffer ready registers.
25. The Application CPU reads the ping buffer ready from (0x2D0270) to know whether it is in LOW.
26. The Application CPU reads 1K block of data from the Egress RAM address (0x1C0000), which is incremented by four offsets and sends it to ping data of the Register Interface starting from (0x2D0000).

27. The Application CPU sets ping status (0x2D0270) to 1.
28. The Application CPU reads the pong buffer ready from (0x2D0274) to know whether it is in LOW.
29. The Application CPU reads 1K block of data from Egress RAM (0x1C0080), which is incremented by four offsets and sends it to pong data of the Register Interface (0x2D0080).
30. The Application CPU sets pong status (0x2D0274) to 1.
31. The Application CPU sets interrupt for the Security CPU by writing 1 to INT_SET_SECURITY (@0x002C0014).
32. If the data is more than 2 kB (2048 bits), then it follows the steps for next ping and pong data and changes the Egress RAM data address accordingly.
33. Check the ping ready bit until 0 and write ping block of data. Same Ready Check for the pong block data.
34. After the final data is sent, wait for the Application CPU interrupt status, and once it comes, read the data out form OUTPUT BUFFER (starting address @0x2D0100) and print through the UART.

5.7.3. Security CPU Process

The following describes the process on the Security CPU side:

1. In the Security CPU side, write int_enable_security (0x2E0010) to '0x01' and then read for int_status_security (0x2E000C) to become 1.
2. Follow steps 1 to 10 in the AES-256 GCM Decryption (PCIe to UART) section.
3. The Security CPU waits for ORAN Security Enclave interrupt rising edge on irq_o and configures the AES_CONFIG_REG (0X0030000C) with AES_GCM_DEC_UPDATE (0x00805205).
4. The Security CPU sets interrupt to the Application CPU by writing 0x01 into the register int_set_app (0x2E0008).
5. The Security CPU waits for ORAN Security Enclave interrupt rising edge on irq_o.
6. The Security CPU sets interrupt to the Application CPU by writing 0x01 into the register int_set_app (0x2E0008).
7. Check the Security CPU interrupt status. If the value of sec_int_status_register becomes 1.

   Clear the interrupt and read for mode (@0x2F3FFC), SHA source (@0x2F3FF4), servo status (@0x2F0278), and SHA input message length (0x2F027C).
8. If mode register reads value 0x35, configure the value 0x304 (HASH_INITIAL) to the register 0x0031000C (configuration register).
9. The Security CPU reads the ping ready buffer (@0x002F0270) until it reaches the value 0x01.
10. If the ping ready buffer is HIGH, the data from ping block (@0x2F0000) of the Register Interface to SHA IP (@0x310020).
11. The Security CPU writes control register (@0x00310008) to 0x1.
12. The Security CPU sets ping ready buffer (@0x002F0270) to 0x00.
13. The Security CPU waits for OSE interrupt rising edge on irq_o .
14. If the data input is more than 1 kB, write value 0x305 (SHA384 update mode) to the configuration register (0x31000C).
15. The Security CPU reads the pong ready buffer (@0x002F0274) until it reaches the value 0x01.
16. If pong ready buffer is HIGH, the data from pong block (@0x2F0080) of the Register Interface to SHA IP (@0x310020).
17. The Security CPU writes control register (@0x00310008) to 0x1.
18. The Security CPU sets pong ready buffer (@0x002F0274) to 0x0.
19. The Security CPU waits for the ORAN Security Enclave interrupt rising edge on irq_o.
20. If more data is available, check for ping and pong Ready bits and if they are 1.

    Read data from the Register Interface ping pong index and send to SHA IP.
21. For the last block of SHA data, HASH FINISH (0x306) is written to the configuration register of Hash IP.

22. Write the message length in SHA IP register (starting from 0x310010) and the last block of input data to 0x310020 register in Hash IP.
23. Execute the start command to SHA IP (writes control register (@0x00310008) to 0x1).
24. After the complete transaction of data to the SHA IP, the Security CPU waits for the OSE interrupt rising edge on irq_o, reads the data from the SHA IP of address (0x310134 to 0x310160), and sends it to the output buffer (0X2F0100) of the Register Interface.
25. The Security CPU sets the interrupt for the Application CPU int_set_application (0x2E0008) to 0x01.

5.8. HMAC 384 Authentication (PCIe to UART)

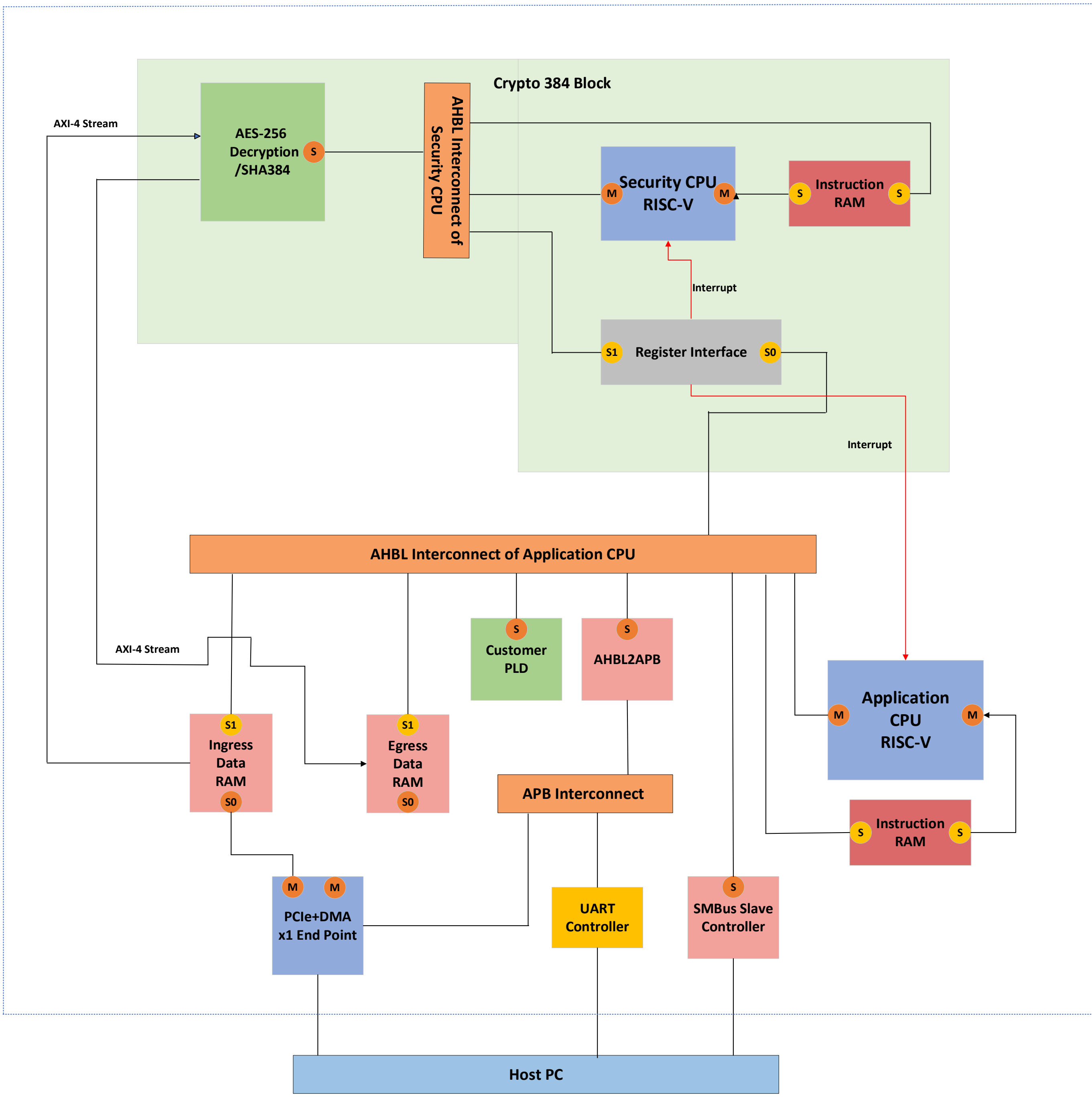


Figure 5.8. HMAC 384 Authentication (PCIe to UART)

HMAC 384 Authentication (PCIe to UART) is similar to the SHA384 Authentication (PCIe to UART) section. However, the key has to be managed and the second hash has to be performed in the Security CPU for the appended data of XOR (key, OPAD) and the first hash message digest.

5.9. HMAC 384 Message Digest Generation (UART to PCIe)

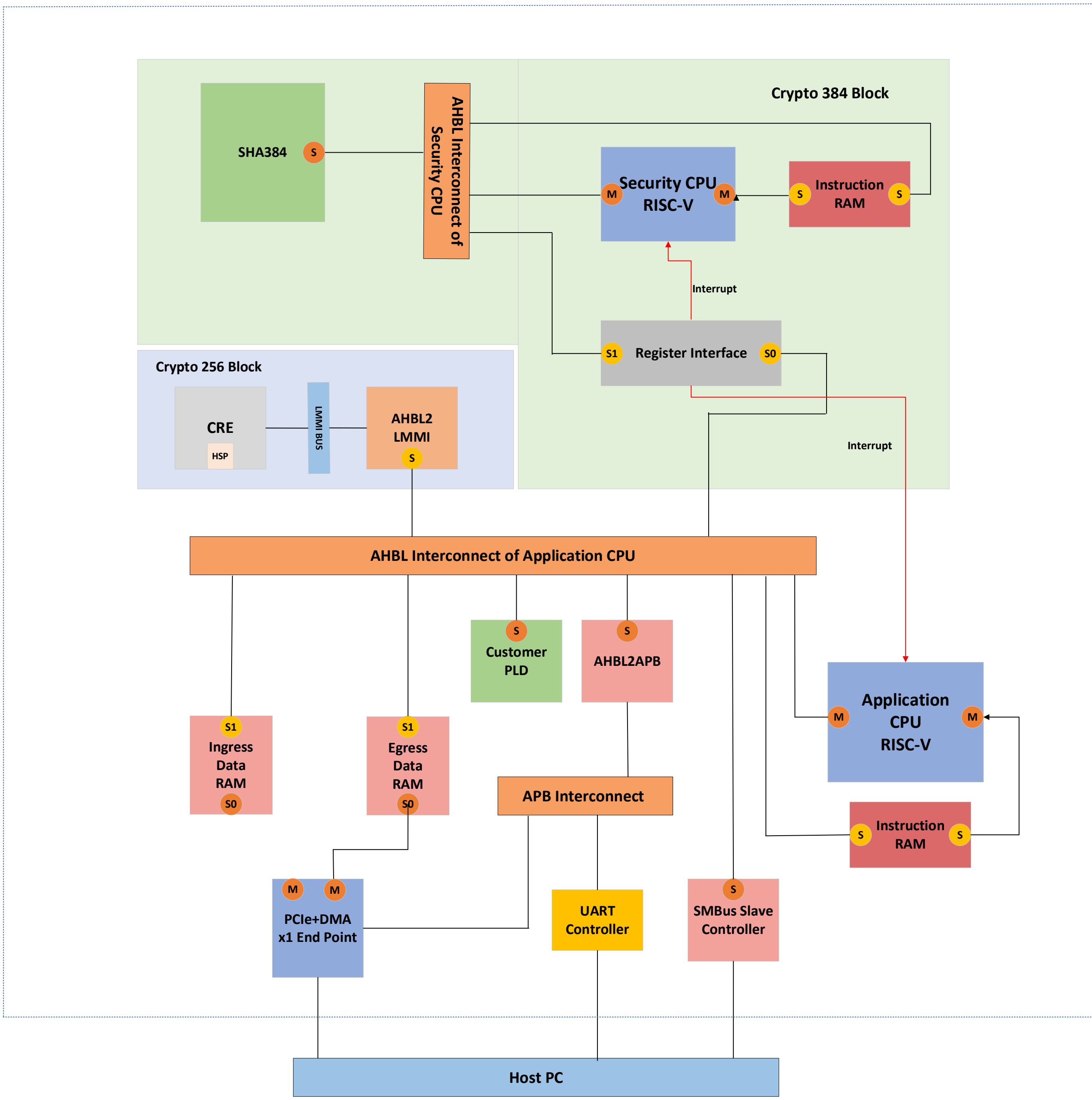


Figure 5.9. HMAC 384 Message Digest Generation (UART to PCIe)

HMAC 384 Message Digest Generation (UART to PCIe) is a combination of the AES-256 GCM Decryption (PCIe to UART) and AES-256 GCM Encryption (UART to PCIe) sections.

5.10. ECC 256 Bit Key Pair Generation (using CRE IP)

ECC 256 bit key pair generation can be performed with the help of the Application CPU, Security CPU, and CRE module.

The Application CPU waits until "servo status" (address = 0x2D0278) equals to "idle". It then and sets servo status to "servo in service" and perform the following steps:

1. The Application CPU writes to the mode register (address= 0x2D3FFC) of the Register Interface with the value 0x3E for ECC key pair generation operation.
2. The Application CPU generates interrupt to the Security CPU by writing into int_set_security (address= 0x2C0014) with value 1'b1.
3. The Security CPU reads the int_status_security (address= 0x2C000C) and starts performing the service requested by the Application CPU as per the mode register data.
4. The Security CPU performs the procedure shown in Table 5.1 to generate the ECC key pair. Base address to access HSE/CRE module is 0x00100000

Table 5.1. ECC Private + Public Key Generation Procedure

Transaction	LMMI	Data	Description
Read	0x2 0020	4B	Poll if IP is Ready. [*RO_GP0 == 0xB0*]
Write	0x2 000C	4B	[*RI_CTRL1 ← 0x0E*] Starts the ECC key generation process.
Read	0x2 0020	4B	Poll if transaction is done. [*RO_GP0 == 0xB2*]
Read	0x1 F840	32B	Public Key X
Read	0x1 F860	32B	Public Key Y
Read	0x1 F880	32B	Private Key
Write	0x2 000C	4B	[*RI_CTRL1 ← 0x00*] Clears the previous transaction, and sets the IP ready for the next.

5. Once keys are generated, Public Key X, Public Key Y, and Private Key are read to BUF1, BUF2, and BUF3 of the scratch memory. Refer to Table 2.4 for register details of BUF1, BUF2, and BUF3.
6. The Security CPU generates the interrupt to the Application CPU that the operation is completed. The Application CPU can obtain the generated keys from the BUF1, BUF2, and BUF3 respectively. Base address to access HSE/CRE module is 0x00100000.

Table 5.2. ECC Public Key (from Private Key) Generation Procedure

Transaction	LMMI	Data	Description
Read	0x2 0020	4B	Poll if IP is Ready. [*RO_GP0 == 0xB0*]
Write	0x1 F800	32B	Private Key
Write	0x2 000C	4B	[*RI_CTRL1 ← 0x04*] Starts the Public Key generation process.
Read	0x2 0020	4B	Poll if transaction is done. [*RO_GP0 == 0xB2*]
Read	0x1 FC00	32B	Public Key X
Read	0x1 FC80	32B	Public Key Y
Write	0x2 000C	4B	[*RI_CTRL1 ← 0x00*] Clears the previous

Once the keys are generated, read the Public Key X, Public Key Y to BUF2, BUF3 of the scratch memory.

Once done, the Security CPU generates the interrupt to the Application CPU that the operation is completed. The Application CPU can obtain the generated public keys from the BUF2 and BUF3 respectively.

5.11. RSA Encryption/Decryption

RSA encryption and decryption is managed by the firmware APIs. All the arithmetic operations required for the RSA algorithm is handled by the PKC IP.

5.12. AES Throughput Calculation

The FPGA runs one counter internally for calculating AES throughput.

This counter starts when the start pulse of the Egress RAM is given by the Application CPU.

Note: The Egress RAM stores decrypted data in the PCIe to UART flow and encrypted data in the UART to PCIe flow.

The operation stops when the complete data is encrypted/decrypted (according to the length provided by the user). The counter is placed in the register (0x1CF044).

The Application CPU reads register 0x1CF044 to get the counter value. After running the AES encryption/decryption, this counter value can be multiplied with 13.333 (using 75 MHz) to obtain the total time in nanoseconds for the given amount of data to be processed. For AES CBC/GCM mode, the calculated value for up to 60 kB is 1024018.42 ns. Based on this calculation, throughput is around 468.7 Mbps.

6. PCIe DMA

6.1. Overview

The term Bus Master, used in the context of PCI Express, indicates the ability of a PCIe port to initiate PCIe transactions, typically Memory Read and Write transactions. The most common application for Bus Mastering Endpoints is for DMA. DMA is a technique used for efficient transfer of data to and from Host CPU system memory. DMA implementations have many advantages over standard programmed input/output (PIO) data transfers. PIO data transfers are executed directly by the CPU and are typically limited to one (or in some cases two) DWORDs at a time. For large data transfers, DMA implementations result in higher data throughput because the DMA hardware engine is not limited to one or two DWORD transfers. In addition, the DMA engine offloads the CPU from directly transferring the data, resulting in better overall system performance through lower CPU utilization. There are two basic types of DMA hardware implementations found in systems using PCI Express: System DMA implementation and Bus Master DMA (BMD) implementation. System DMA implementations typically consist of a shared DMA engine that resides in a central location on the bus and can be used by any device that resides on the bus. System DMA implementations are not commonly found anymore and very few root complexes and operating systems support their use. A BMD implementation is by far the most common type of DMA found in systems based on PCI Express. BMD implementations reside within the Endpoint device and are called Bus Masters because they initiate the movement of data to (Memory Writes) and from (Memory Reads) system memory. Figure 35 shows a typical system architecture that includes a root complex, PCI Express switch device, and an integrated Endpoint block for PCI Express. A DMA transfer either transfers data from an integrated Endpoint block for PCI Express buffer into system memory or from system memory into the integrated Endpoint block for PCI Express buffer. Instead of the CPU having to initiate the transactions needed to move the data, the BMD relieves the processor and allows other processing activities to occur while the data is moved. The DMA request is always initiated by the integrated Endpoint block for PCI Express after receiving instructions and buffer location information from the application driver.

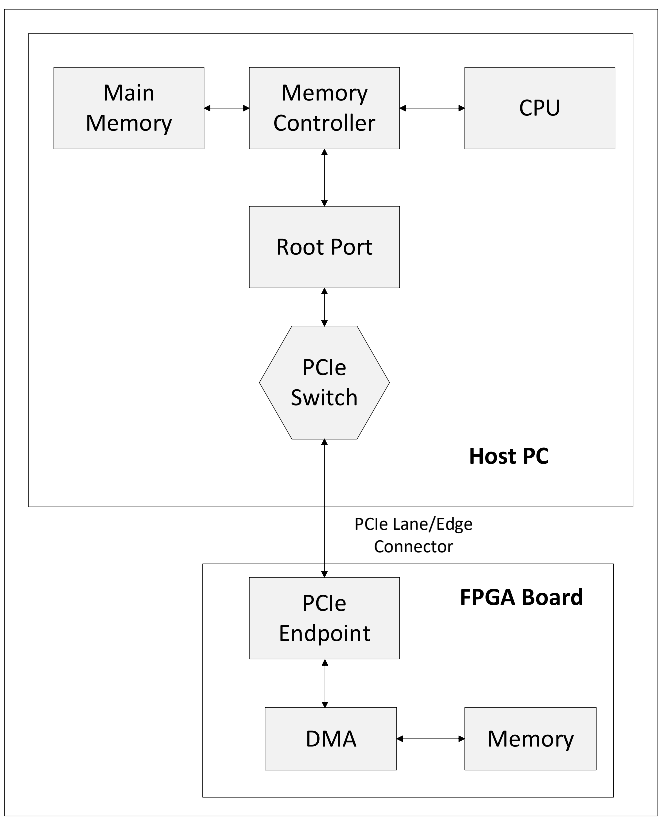


Figure 6.1. Top Level Block Diagram

In addition to the data-throughput advantages of DMA versus PIO transactions for large data transfers, many other variables can affect data throughput in PCI Express systems, e.g., link width and speed, receive buffer sizing, return credit latency, end-to-end latency, and congestion within switches and root complexes. For these reasons, the use of PCI Express for high data-throughput applications requires a BMD engine.

6.2. Components of DMA Design

A typical design for PCI Express includes the following main components:

- Hardware HDL
- Design Driver
- Design Software Application

The hardware design refers to the Verilog or VHDL application residing on the Lattice FPGA. In this case, it is the bus master DMA design or BMD. This design contains control engines for the receive and transmit data path along with various registers and memory interfaces to store and retrieve data. The driver design is normally written in C and is the link between the higher-level software application and the hardware application. The driver contains various routines that are called by the software application and are used to communicate with the hardware via the PCI Express link. The driver resides in the kernel memory on the system. The software application is most apparent to the user and can be written in any programming language. It can be as simple as a small C program or as complex as a GUI-based application. The user interfaces with the software application, which invokes routines in the driver to perform the necessary data movements. The software keeps on checking whether the data movement is completed or not. Once completed, the driver can invoke routines in the software application to inform the user that the request is completed.

6.3. FPGA Design

Figure 6.2 shows the top-level architecture of FPGA design.

DMA support is an option provided by the Lattice soft IP to enable more efficient data transfer when endpoint is acting as initiator or master. To transfer data through DMA, the Core requires source address, destination address, and transfer control, that is, length and direction of transfer. This information is collectively called descriptor.

To store the descriptor, two queues are implemented in a local memory. These are the descriptor queue and the status queue. When data transfer is completed or aborted, a status, which contains done flag, error flag, length of transfer and data address offset, is written into the status queue.

The description of each block of FPGA design architecture is given below.

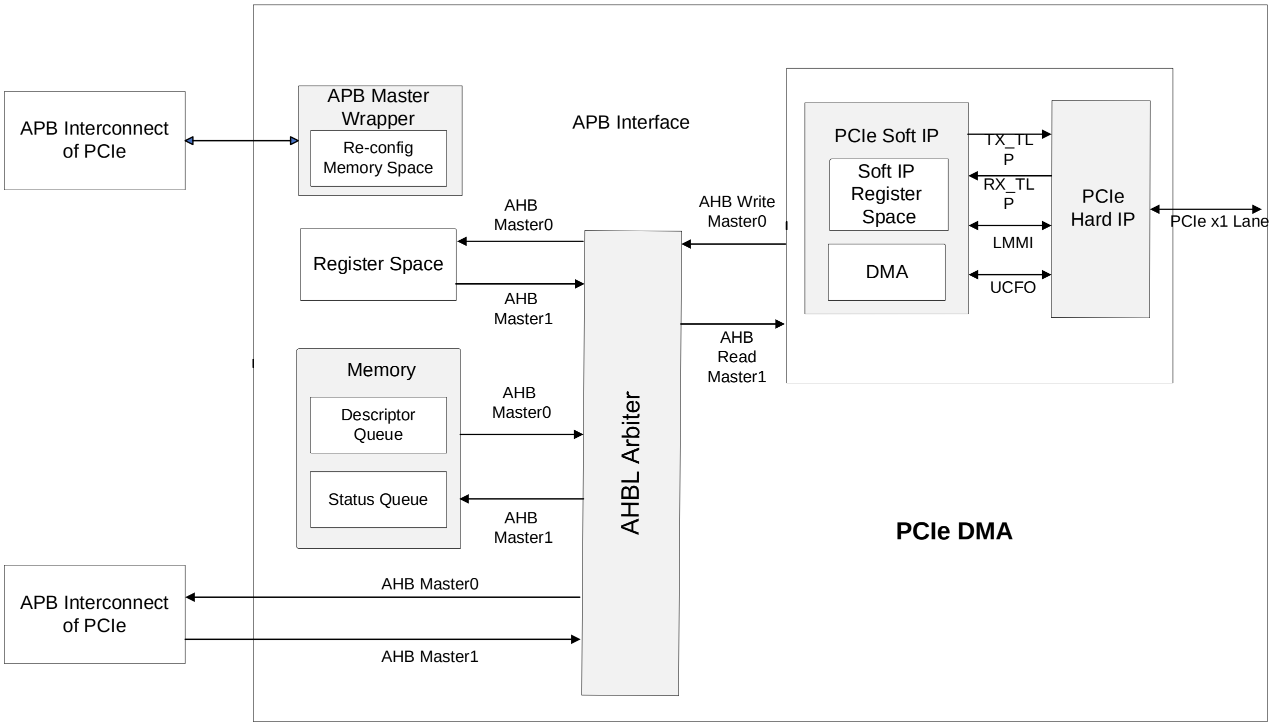


Figure 6.2. Top Level Architecture of PCIe Design

- AHB Arbiter
  This block selects the three blocks: APB master, system memory, and FIFO wrapper depending on the address received in the TLP. See the memory segregation for address range of the different blocks. The user can select the address range by modifying the parameter in *AHB_arbiter.v* file. The AHB master0 port is used for receiving (RX TLP) and AHB master1 port is being used for transmitting (TX TLP).
- APB Master
  The APB Slave Port is available to access the registers of soft IP or hard IP. To access these registers through software/driver, the APB master is needed. This block is used to make the APB master interface. Initial reconfiguration of soft IP and hard IP is done through the APB master. A configuration space is implemented in the design to store all the configuration values required for the PCIe IP.

  BASE ADDRESS: 0x00CA0000 for accessing from the Application CPU.

6.3.1.1. Register Address (0x00)

31:1	0
reserved	To start DMA *Default*: 0
Write	

6.3.1.2. Register Address (0x04)

31:1	0
reserved	To check PCIe linkup
Read	

6.3.1.3. Register Address (0x08)

31:1	0
reserved	To check DMA complete
Read	

6.3.1.4. Register Address (0x0C)

31:0
SHA DATA LENGTH
Read

6.3.1.5. Register Address (0x10)

31:0
FIRMWARE VERSION
Write

6.3.1.6. Register Address (0x14)

31:1	0
reserved	AES DATA READY IN EGRESS RAM
Read	

6.3.1.7. Register Address (0x18)

31:0
PCIe VERSION
Read

6.3.1.8. Register Address (0x1C)

31:0
FPGA VERSION
Read

6.3.1.9. Register Address (0x20)

31: 0
Application CPU can give signal to PCIe(used for handshaking) *Default* : 0
Write Read

6.3.1.10. Register Address (0x30)

31: 0
PCIe can give signal to the Application CPU
Read

6.3.1.11. Register Address (0x34)

31: 0
GCM CIPHER TEST LENGTH IN BITS
Read

- Desc Queue
  This DMA implementation is based on the descriptors. To store the descriptors a queue is implemented. The pointer of the descriptor queue is to be updated by the driver/software after/before writing the descriptors in the descriptors queue. Descriptors are fetched by the DMA soft IP to serve the descriptors. The corresponding read pointer is updated by the DMA Core.
- Status Queue
  After one descriptor is served by the DMA Engine, its status is stored. To report the status of a transfer, a status queue is implemented. The status of each descriptor or transfer is stored in this queue.
- Register Space
  A register space is implemented in the design to configure the DMA or to get the status of transfer and throughput.

6.3.2. Descriptor Field Format

Table 6.1 shows the descriptor entry format.

Table 6.1. Descriptor Entry Format

DW	DW name	Field name	Bit offset	Size	Description
0	desc_ctrl	length	0	13	Size of data transfer in bytes. (4096 bytes maximum)
		direction	13	1	Direction of transfer. 0 – AHB-Lite to PCIe 1 – PCIe to AHB-Lite
			14	10	reserved
		desc_id	24	8	Optional descriptor ID. If the parameter EN_DESC_ID == "Enable" the Core adds this information in the Status entry.
1	desc_src	addr_offset	0	32	source address/ offset
2	desc_dst	addr_offset	0	32	destination address/offset
3	desc_hdr	requester_id	0	16	Requester ID to be used in TLP Header requester_id [7:0] – bus number[7:0] requester_id [10:8] – function number[2:0] requester_id [15:11] – device number[4:0]
		traffic_class	16	3	Traffic Class to be used in TLP Header
		use_requestor_id	19	1	When set, it indicates that the requester_id field is valid and should be used in TLP header. Otherwise, the Core uses the captured configuration ID of function 0 as the default requester ID.
			20	12	Reserved

6.3.3. Status Field Format

Table 6.2 shows the status field format.

Table 6.2. Status Entry format

DW	DW Name	Field Name	Bit Offset	Size	Description
1	stat_flag	done	0	1	If this bit is asserted, it indicates that the transfer has been completed
		with_error	1	1	If this bit is asserted, it Indicates an error occurred during transfer.
		aborted	2	1	If this bit is asserted, it indicates the transfer was terminated before it completes the specified length.
		direction	3	1	Direction of transfer. 0 – AHB-Lite to PCIe 1 – PCIe to AHB-Lite
			4	4	reserved
		desc_id	8	8	Optional descriptor ID. Available if the parameter EN_DESC_ID == "Enable"
		length	16	13	Size of data transfer in bytes. (4096 bytes maximum)
			29	3	Reserved
2	stat_buff	addr	0	32	Data address. This is the local memory address where the data is stored (direction==1) or fetched (direction==0).

6.3.4. How to Trigger the DMA Operation.

It is assumed that the user is aware of the descriptor and status queue. Else, the user can go through Memory Map section.

To trigger/start the DMA operation:

1. Write the descriptors starting from address 0x1000. Note that one descriptor needs four DW (32 bit) space. Check the descriptor field format. If the first descriptor is written at 0x1000, the next descriptor should be written at 0x1010 address.
2. Write the number of descriptors at 0x8.
3. Write 0x1 at address 0x1 to start the DMA operation.

6.3.5. Register Space: BASE ADDRESS -- 0x00180000

6.3.5.1. Register Address (0x0) (*Default*: 0x60)

31:6	7	6	5	4	3	2	1	0
Reserved	DMA Read operation done	Reserved	Reserved	Reserved	DMA aborted in one iteration	Error in one iteration	DMA write Operation done	Reserved
Read Only								
	DMA read operation is completed				DMA iteration is aborted	DMA iteration is completed with error	DMA write iteration is completed	

6.3.5.2. Register Address (0x4)

31:0
Throughput Counter value
Read Only
Multiply this counter value by 13.33 to get the total time (in ns) of one iteration

6.3.5.3. Register Address (0x8)

31:8 Reserved	7:0
Reserved	Number of descriptors written in one iteration; valid values are between 1-255
Read Only	Write Only

6.3.5.4. Register Address (0xC)

31:2	1	0
Reserved	Start DMA read operation	Start DMA write operation
Write Only		
	Write 1 to start the DMA read operation	Write 1 to start the DMA write operation

6.3.5.5. Register Address (0x10)

31:0
reserved
Read Only

6.3.5.6. Register Address (0x14)

31:0
FPGA Version register; upper 8 bit [31:24] is indicating the date in decimal, next 8 bit [23:16] is indicating month in decimal, next 8 bit [15:8] is indicating the hour in 24 hour format, next 8 bit [7:0] is indicating the minute;
Read Only

6.3.5.7. Register Address (0x18)

31:0
reserved
Read Only

6.3.5.8. Register Address (0x1C)

31:0
reserved
Read Write

6.3.5.9. Register Address (0x20)

31:0
DMA write size in DW, this indicates how much DWs we have to write in DMA write operation
Read Write

6.3.5.10. Register Address (0x24)

31:0
DMA read size in DW, this indicates how much DWs we have to read in DMA read operation
Read Write

6.3.5.11. Register Address (0x28)

31: 0
SHA Data Length(in bits)
Write

6.3.5.12. Register Address (0x30)

31:1	0
reserved	Start given by the Application CPU
Read	

6.3.5.13. Register Address (0x34)

31:1	0
reserved	Data ready given by Egress RAM
Read	

6.3.5.14. Register Address (0x38)

31: 0
DMA DATA SIZE
Write

6.3.5.15. Register Address (0x3C)

31: 0
Ingress write port address
Read

6.3.5.16. Register Address (0x40)

31: 0
Ingress read port address
Read

6.3.5.17. Register Address (0x44)

31: 0
Egress write port address
Read

6.3.5.18. Register Address (0x48)

31: 0
Egress read port address
Read

6.3.5.19. Register Address (0x50)

31: 0
DMA SIZE GIVEN BY UART
Read

6.3.5.20. Register Address (0x54)

31: 0
FPGA VERSION
Read

6.3.5.21. Register Address (0x58)

31: 0
FIRMWARE VERSION
Read

6.3.5.22. Register Address (0x5C)

31: 0
PCIe VERSION
Read

6.3.5.23. Register Address (0x60)

31: 0
Reserved for PCIe
Write and Read

6.3.5.24. Register Address (0x64)

31: 0
Application CPU can give signal
Read

6.3.5.25. Register Address (0x70)

31: 0
GCM CIPHER TEST LENGTH IN BITS
Write

7. MCTP over SMBus

7.1. SMBus

The System Management Bus (abbreviated to SMBus or SMB) is a single-ended simple two-wire bus for the purpose of lightweight communication. Most commonly it is found in computer motherboards for communication with the power source for ON/OFF instructions. It is derived from I²C for communication.

7.2. MCTP

The Management Component Transport Protocol (MCTP) supports the PMCI goals by defining a media-independent transport protocol that enables communications between different intelligent hardware components that make up a platform management subsystem that provides monitoring and control functions inside a managed system.

MCTP can be implemented over many physical media, here we are using SMBus. The MCTP over SMBus/I 276 2C transport binding defines how MCTP packets are delivered over a physical SMBus or I 277 2C medium using SMBus transactions. This includes how physical addresses are used. Following is the diagram of MCTP over SMBus.

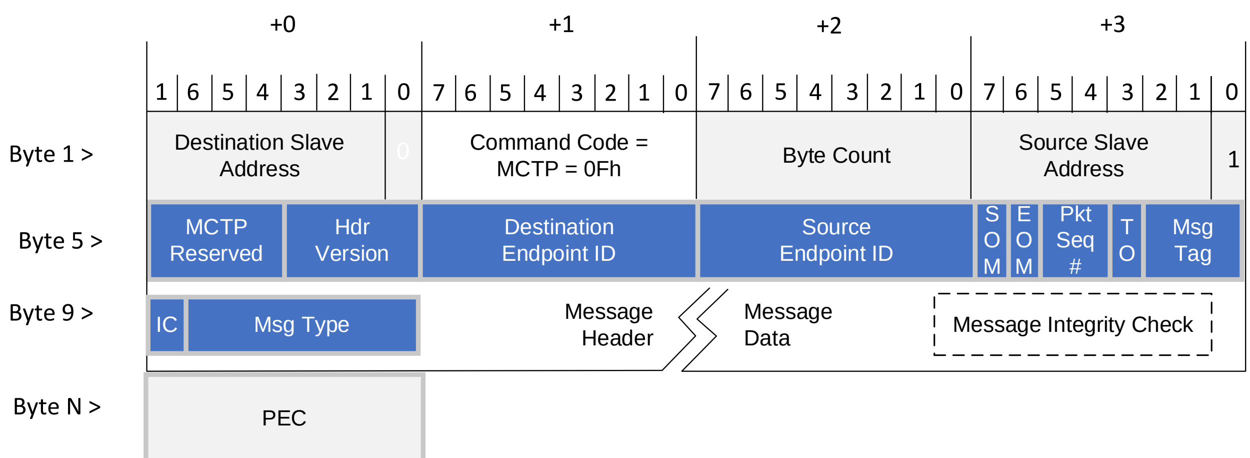


Figure 7.1. MCTP over SMBus

7.3. SPDM

The SPDM message exchanges are defined in a generic fashion that allows the messages to be communicated across different physical mediums and over different transport protocols.

The specification-defined message exchanges enable Requesters to:

- Discover and negotiate the security capabilities of a Responder.
- Authenticate the identity of a Responder.
- Retrieve the measurements of a Responder.
- Securely establish cryptographic session keys to construct a secure communication channel for the transmission or reception of application data.

There are different kind of messages for specific purpose, following is the description of messages used here:-

- Get-Version - Requester sends this message to know the version of SPDM it is supporting.
- Get Capabilities - Requester sends its capabilities (specifications it supports) to responder and in response it gets responder’s capabilities.
- Negotiate Algorithm - Through this algorithm requester and responder negotiates and agree over an algorithm which requester wants to perform.

- Challenge message - This message is used in authentication, where requester verifies signature sent by responder in challenge response message.
- Vendor Defined Message: This message is used when requester wants to some specific information to responder and it can be define by users for different purposes.

7.4. Algorithm Selection

As discussed in the above section, algorithm is selected through the negotiate algorithm message.

If both requesters have property to store the version and capability message response, only the negotiate algorithm massage is exchanged for changing the algorithm. In case the request is not able to store the messages response, then for changing algorithm again, get-version, get-capabilities, and negotiate algorithm is sent to the responder.

In the negotiate algorithm message, there are four types of algorithm selection tables. The following are the structure tables.

Table 7.1. Algorithm Selection Structure Tables

Offset	Field	Size (Bytes	Value
0	AlgType	1	0x2=DHE
1	AlgCount	1	Bit [7:4]=2

Offset	Field	Size (Bytes	Value
2	AlgSupported	2	Bit mask listing Requester-supported SPDM enumerated Diffie-Hellman Ephemeral (DHE) groups. Values in parenthesis specify the size of the corresponding public values associated with each group. Byte 0 Bit 0 ffdhe2048 (D=256) Byte 0 Bit 1 ffdhe3072 (D=384) Byte 0 Bit 2 ffdhe4096 (D=512) Byte 0 Bit 3 secp256r1 (D=64, C=32) Byte 0 Bit 4 secp348r1 (D=96, C=48) Byte 0 Bit 5 secp521r1 (D=132, C=66) All other values are reserved.
4	AlgExternal	4*ExtAlgCount2	List of Requester-supported extended DHE groups. The Extended algorithm field format table described the format of this field.

Offset	Field	Size (Bytes	Value
0	AlgType	1	0x3=AEAD
1	AlgCount	1	Bit [7:4]=2. Bit [3:0]=Number of Requester supported extended AEAD algorithms (=ExtAlgCount3).
2	AlgSupported	2	Bit mask listing Requester-supported SPDM enumerated AEAD algorithms. Byte 0 Bit 0. AES-128-GCM. 128-bit key; 96-bit IV (Initialization Vector); tag size is specified by the transport layer. Byte 0 Bit 1. AES-256-GCM. 128-bit key; 96-bit IV; tag size is specified by the transport layer. Byte 0 Bit 2. CHACHA20_POLY1305. 256-bit key; 96-bit IV; 128-bit tag. All other values are reserved.
4	AlgExternal	4*ExtAlgCount3	List of Requester-supported extended AEAD algorithms. The Extended algorithm field format table described the format of this field.

Offset	Field	Size (Bytes	Value
0	AlgType	1	0x4=ReqBaseAsymAlg
1	AlgCount	1	Bit [7:4]=2. Bit [3:0]=Number of Requester supported extended asymmetric key signature algorithms (=ExtAlgCount4).
2	AlgSupported	2	Bit mask listing Requester-supported SPDM enumerated asymmetric key signature algorithms for the purpose of signature generation. Byte 0 Bit 0. TPM_ALG_RSASSA_2048 Byte 0 Bit 1. TPM_ALG_RSAPSS_2048 Byte 0 Bit 2. TPM_ALG_RSASSA_3072 Byte 0 Bit 3. TPM_ALG_RSAPSS_3072 Byte 0 Bit 4. TPM_ALG_ECDSA_ECC_NIST_P256 Byte 0 Bit 5. TPM_ALG_RSASSA_4096 Byte 0 Bit 6. TPM_ALG_RSAPSS_4096 Byte 0 Bit 7. TPM_ALG_ECDSA_ECC_NIST_P384 Byte 0 Bit 0. TPM_ALG_ECDSA_ECC_NIST_P521 All other values are reserved.
4	AlgExternal	4*ExtAlgCount4	List of Requester-supported extended AEAD algorithms. The Extended algorithm field format table described the format of this field.

Offset	Field	Size (Bytes	Value
0	AlgType	1	0x5=KeySchedule
1	AlgCount	1	Bit [7:4]=2. Bit [3:0]=Number of Requester supported extended key schedule algorithms (=ExtAlgCount5).

Offset	Field	Size (Bytes	Value
2	AlgSupported	2	Bit mask listing Requester-supported SPDM enumerated key schedule algorithms. Byte 0 Bit 0. SPDM Key Schedule. All other values are reserved.
4	AlgExternal	4*ExtAlgCount5	List of Requester-supported key schedule algorithms. The Extended algorithm field format table describes the format of this field.

7.5. AES CBC/GCM Algorithm

The AES Encryption algorithm (also known as the Rijndael algorithm) is a symmetric block cipher algorithm with a block/chunk size of 128 bits. It converts these individual blocks using keys of 128, 192, and 256 bits. Once it encrypts these blocks, it joins them together to form the cipher text. Here we are using AES-256, which means it requires 32 bytes (256 bits) key.

It is based on a substitution-permutation network, also known as an SP network. It consists of a series of linked operations, including replacing inputs with specific outputs (substitutions) and others involving bit shuffling (permutations).

AES-CBC (Cipher Blocker Chaining) is an advanced form of block cipher encryption. With CBC mode encryption, each cipher text block is dependent on all plain text blocks processed up to that point. This adds an extra level of complexity to the encrypted data.

AES-GCM (Galois Counter Mode) is a mode of operation for symmetric key cryptographic block ciphers. GCM is ideal for protecting packets of data because it has low latency and a minimum operation overhead.

7.6. SHA Algorithm

Sha384 is a function of cryptographic algorithm Sha-2, evolution of Sha-1. It is the same encryption than Sha512, except that the output is truncated at 384 bits. There are also differences in the initialization process. It takes any number of data with its hash value of 384 bits. SHA384 is mainly used in message authentication.

7.7. HMAC Algorithm

HMAC algorithm stands for Hashed- or Hash-based Message Authentication Code. It is a result of work done on developing a MAC derived from cryptographic hash functions. HMAC is a great resistance towards crypto analysis attacks as it uses the hashing concept twice. HMAC consists of twin benefits of hashing and MAC and thus is more secure than any other authentication code.

HMAC takes data and key as input to produce the hash value of 384 bits.

7.8. ECDH Algorithm

ECDH: Elliptic Curve Diffie Hellman (ECDH) is an Elliptic Curve variant of the standard Diffie Hellman algorithm. See Elliptic Curve Cryptography for an overview of the basic concepts behind Elliptic Curve algorithms. ECDH is used for key agreement.

Elliptical curve used here is secp256r1, which produces two key pairs of 32Bytes. At both ends, the same arithmetic operation is done to calculate the shared secret.

PKC IP is used for performing the calculation part of this algorithm.

For example:

Alice private key = ***a***

Bob private key = ***b***

G parameter = ***G***

Alice public key which needs to be shared to bob = ***G*a*** *=* ***Pa***

*Bob public key which needs to be shared to Alice = G*b =* ***Pb***

Now Shared secret = ***Pa *b*** *(At Bob's end) =* ***Pb * a*** *(At Alice's end) =* ***G*a*b***

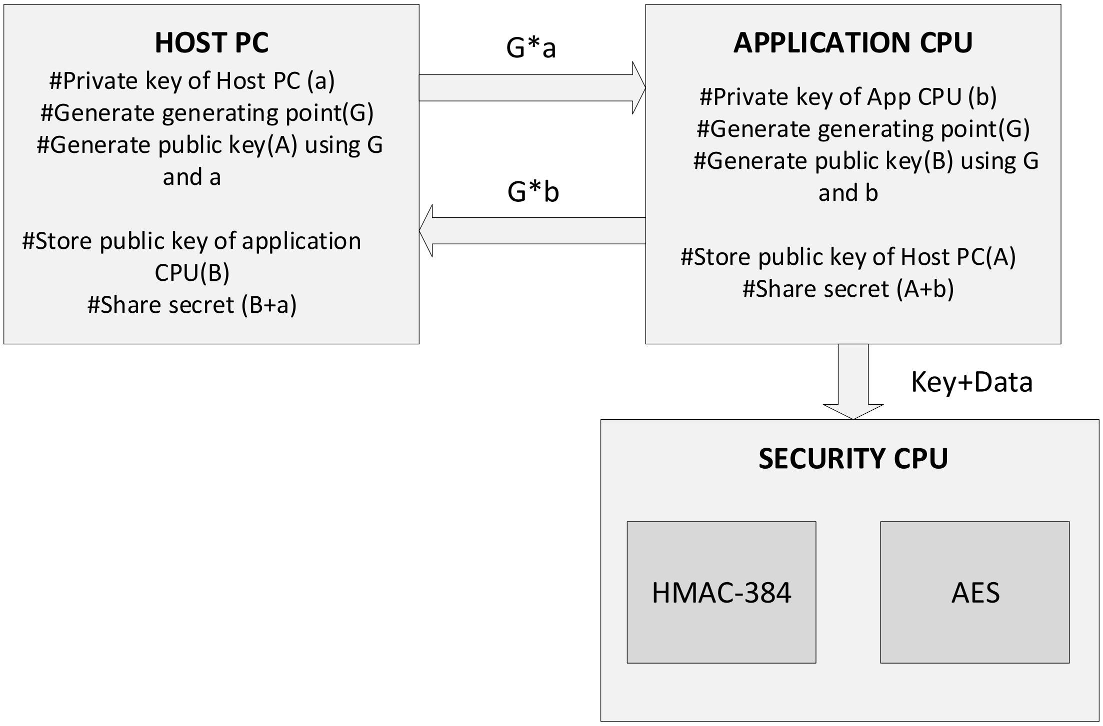


Figure 7.2. Flow of ECDH (Host PC and FPGA)

7.9. RSA Algorithm

RSA algorithm is asymmetric cryptography algorithm. Asymmetric means that it works on two different keys, such as public key and private key. PKC IP is used for performing the calculation part of this algorithm.

7.9.1. RSA Signature:

To sign a message msg with the private key exponent d:

1. Calculate the message hash: h = hash (msg)
2. Encrypt h to calculate the signature: $s = hd(\text{mod}\, n)$
3. The hash h should be in the range [0...n). The obtained signature s is an integer in the range [0...n).

7.9.2. RSA Verify Signature

To verifying a signature s for the message msg with the public key exponent e:

1. Calculate the message hash: *h = hash(msg)*
2. Decrypt the signature*: h'=se(modn)*
3. Compare h with h' to find whether the signature is valid or not

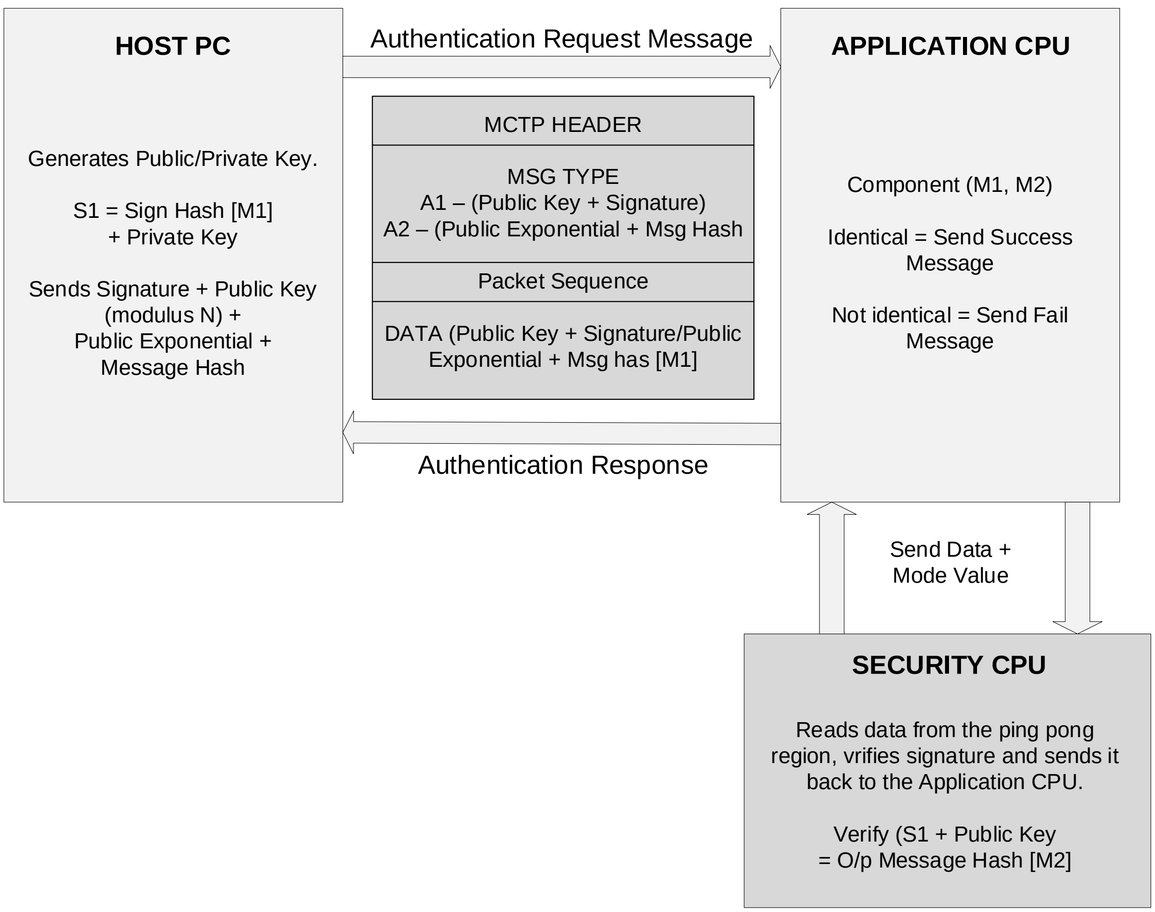


Figure 7.3. RSA Sign and Verify Flow

8. User Flow

The host computer connects to the CertusPro-NX board through the SMBus. The host computer then becomes the SMBus Master device and CertusPro-NX board becomes the SMBus Slave device.

The msg transferred between the host computer and the board is in MCTP format. The MCTP msg contains the SPDM Msg. SPDM version, capabilities, and algorithm are preformed between Requester and Responder.

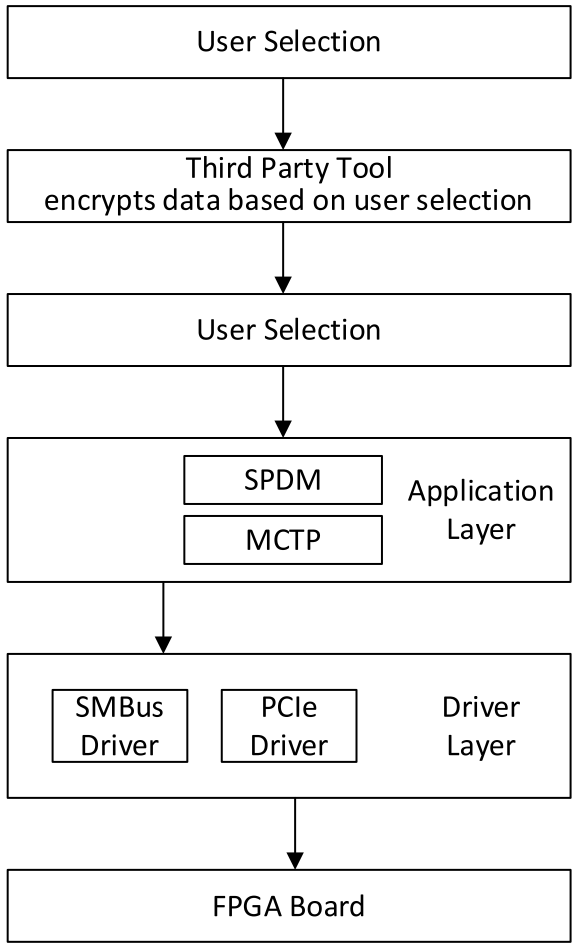


Figure 8.1. User Flow Diagram

8.1. Driver Initialization

The PCIe and the SMBus drivers are initialized and inserted into the kernel.

Supported Boards

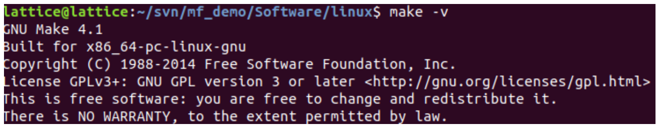
- CertusPro-NX

Supported OS

- Distributor ID: Ubuntu.
- Description: Ubuntu 16.04.3/18.04.3 & above LTS, Kernel version 4 and above.
- Release: 18.04.
- OS Type: 64bit.
- Codename: Bionic

Required Packages

To check whether packages are installed or not, run the commands shown in the figures below. For example, `make -v` to check the availability of make packages.



```
lattice@lattice:~/svn/mf_demo/Software/linux$ make -v
GNU Make 4.1
Built for x86_64-pc-linux-gnu
Copyright (C) 1988-2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

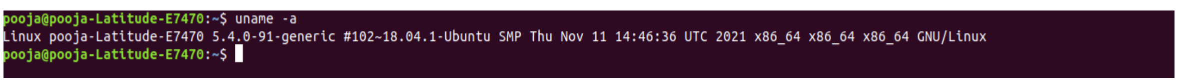
Figure 8.2. Make

```
lattice@lattice:~/svn/mf_demo/Software/linux$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 7.5.0-3ubuntu1~18.04' --with-bugurl=file:///usr/share/doc/gcc-7/README.Bugs --enable-languages=c,ada,c++,
go,brig,d,fortran,objc,obj-c++ --prefix=/usr --with-gcc-major-version-only --program-suffix=-7 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-
id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-bootstrap --enable-clocale=gnu --enable-libstdcxx-deb
ug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-libmpx --enable-plugin --enable-default-pie
--with-system-zlib --with-target-system-zlib --enable-objc-gc=auto --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,m
x32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-lin
ux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)
```

Figure 8.3. GCC

```
lattice@lattice:~/svn/mf_demo/Software/linux$ g++ -v
Using built-in specs.
COLLECT_GCC=g++
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 7.5.0-3ubuntu1~18.04' --with-bugurl=file:///usr/share/doc/gcc-7/README.Bugs --enable-languages=c,ada,c++,
go,brig,d,fortran,objc,obj-c++ --prefix=/usr --with-gcc-major-version-only --program-suffix=-7 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-
id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-bootstrap --enable-clocale=gnu --enable-libstdcxx-deb
ug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-libmpx --enable-plugin --enable-default-pie
--with-system-zlib --with-target-system-zlib --enable-objc-gc=auto --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,m
x32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-lin
ux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)
```

Figure 8.4. G++



```
pooja@pooja-Latitude-E7470:~$ uname -a
Linux pooja-Latitude-E7470 5.4.0-91-generic #102~18.04.1-Ubuntu SMP Thu Nov 11 14:46:36 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
pooja@pooja-Latitude-E7470:~$
```

Figure 8.5. Kernel Version

Pre-Requisites

Follow Package.sh to install required packages.

8.2. SMBus Driver

- SMBus_init – This function is called when the SMBus module is inserted into the kernel through the `insmod` command. It creates the SMBus type Adaptor on the I2C line, with its functionality, class, and adaptor name. It also creates the i2c client given i2c line. Add the device driver for the client.
- SMBus Probe – This function is called when the driver detects the device.
- SMBus Exit – This function is called when the device driver is removed from kernel.
- Cmd to insert/remove the driver into the kernel:
  For inserting the driver – `Sudo modprobe i2c_i801`
  For removing the driver – `Sudo rmmod i2c_i801`

8.3. PCIe Driver

- Pcie_register_driver – This function is called when the module is inserted into the kernel through the `insmod` command. It registers the Lattice PCIe driver into the kernel.
- Pcie_probe – This function reads the device vendor, device ID, number of bars, and IRQ. It also enables the PCIe Master bus and registers the driver for the device.
- CreateCharDevice – This function allocates the memory to the driver, major and minor number for device driver and mapping of file operations for read, write, ioctl, open and release.
- Pcie_unregister_driver – This function is called when the driver is removed from the kernel by `rmmod`.
- Cmd to insert/remove the driver into/from the kernel – Go to the driver directory, and run the commands below.
  For inserting the driver: `Sudo insmod lattice_main.ko`
  For removing the driver: `Sudo rmmod lattice_main.ko`

8.3.1. Core API supported in PCIe Driver

- Read (Addr, Data) – This function reads the register values.
- Write (Addr, Value) – This function writes to the register values.
- getDriverVerString (Value) – This function reads the register driver version string.
- ReadFPGAReg (Addr, Data) – This function reads the FPGA registers value.
- WriteFPGAReg (Addr, Value) – This function writes to the FPGA registers.
- PCIeConfigRead (Addr, Data) – This function reads the PCIe Configuration register values.
- PCIeConfigWrite (Addr, Value) – This function writes to the PCIe Configuration register values.
- Read (uint32_t addr, uint8_t *val, size_t le) – Read function is used to read bulk data through IOCTL. addr is Read start location, val data buffer read and len data length to be read.
- write (uint32_t addr, uint8_t *val, size_t le) – Write function is used to write bulk data through IOCTL. addr is Write start location, val data buffer write and len data length to be written.
- getPCIConfigRegs (uint8_t *pCfg) – This function returns the 256 bytes of the device's PCI configuration space registers. These registers must be present on any PCI/PCIexpress device.
- pCfg user location to store 256 bytes – The function returns true if read byte is successful and false if the driver reports an error.
- getPciDriverInfo (PCIResourceInfo_t **pInfo) – This function returns the extra device driver information structure. This includes the DMA memory buffer info, PCI bus/dev/func address. Pinfo is user's pointer that indicates the internal driver structure.
- getPciDriverDMAInfo (const DMAResourceInfo_t **pDMAInfo) – This function returns the extra device driver information structure in c++ supported format. This includes the DMA memory buffer info, PCI bus/dev/func address.

8.4. Functions Used

- Third party Tool (Openssl) is used for data encryption on PC side.
- Abstraction Layer APIs
- Spdm_client_init – This API forms the send or receive of the SPDM message for version, capabilities, and negotiation and sends packet to mctp layer.
- Mctp_responsder_init – This API receives the SPDM message and forms the MCTP packet.
- mctpSendMessage – This API sends the SPDM over MCTP packet over the SMBus.
- mctp_responsder_init – This API receives the SMBus packet, which is MCTP format, extracts the SPDM , and sends to SPDM.

8.5. Flow Description

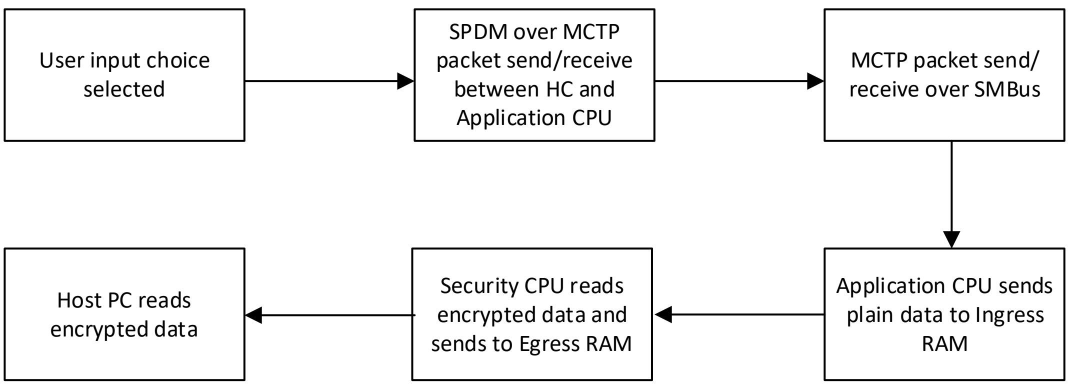


Figure 8.6. UART to PC

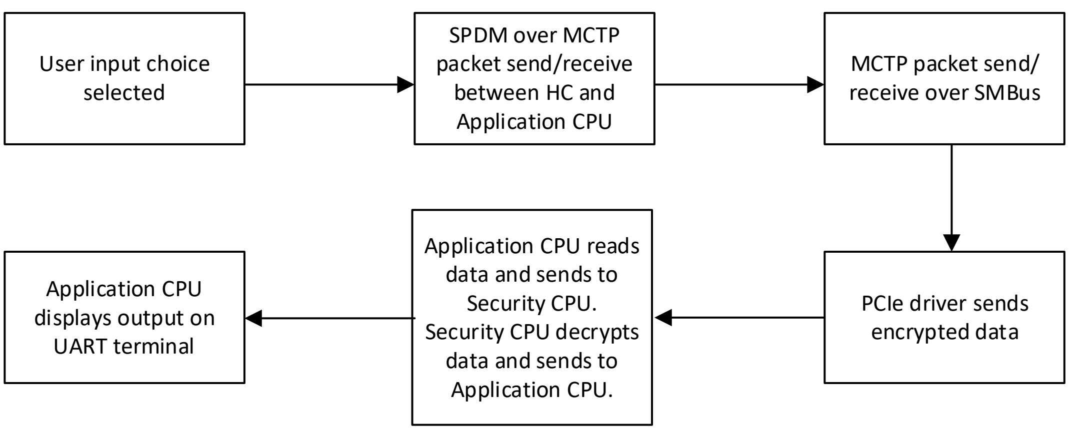


Figure 8.7. PC to UART

8.6. User Selection for Algorithm

The user can select the algorithm to be used.

```
Hello USER, select the Algorithm from below:

1. SHA 384 PC to UART
2. SHA 384 UART to PC
3. HMAC 384 PC to UART
4. HMAC 384 UART to PC
5. AES-GCM 256 ENCRYPTION (UART to PC)
6. AES-GCM 256 DECRYPTION (PC to UART)
7. Exit
```

Figure 8.8. Select Algorithm

In the PC to UART flow, the cipher text is sent over PCIe using DMA Read. After decryption, plain text data is sent over the UART terminal.

In the UART to PC flow, plain text is given from UART terminal. After encryption, the Cipher text data is sent over PCIe using DMA Write to the host computer.

8.6.1. Directory Structure

```
app    clean.sh            Input_data  libspdm     package.txt       preRequisites  RSA     userinput.sh
build  clientDetected.sh   libmctp     OutPut_data PCIe_Source_Code  README.md      run.sh
```

Figure 8.9. Directory

- App – Contains middleware API that takes user input and sends request to the SPDM module.
- Build – Contains SPDM binary and library protocols.
- libmctp – Contains MCTP code that sends packet to the SMBus.
- Libspdm – Contains the SPDM source code.
- PCIe_Source_code – Contains PCIe driver code and API that calls the PCIe driver.
- preRequistes – Contains packages to be installed on the Linux host computer.
- RSA – Contains *script.sh* that generates file with signature and SHA for authentication.

8.7. Application CPU Subsystem

The Application CPU is the main interface that communicates with the Security CPU through different interfaces that are listed below. The Application CPU manages the firmware level work, that is, it has the BSPs of all interfaces.

When the Application CPU requests any service from Crypto-384, it writes certain information to the Register Interface which then generates an interrupt to the Security CPU. The interrupt service routine at the Security CPU reads the information from the Register Interface and provides service such as SHA2-384 and then clear the interrupt. Once the service is completed, the Security CPU writes to the interrupt set register at the Register Interface, which generates an interrupt to Application to inform that the request has been completed. The Application CPU can read the status register at the Register Interface and then send the next service request.

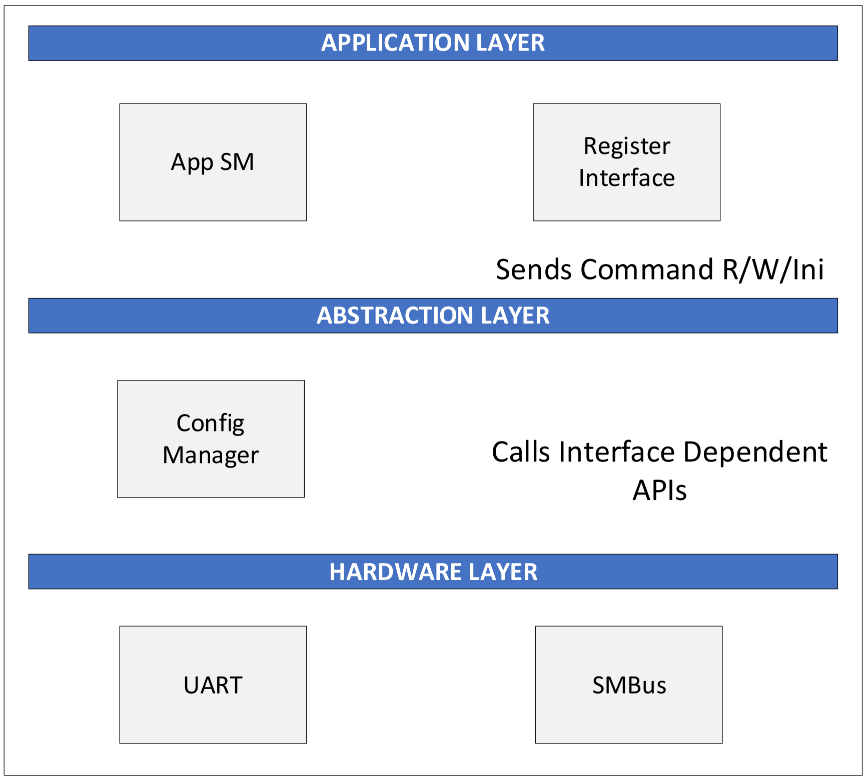


Figure 8.10. Application CPU Software Module

The Application CPU has the following interfaces:

- UART
- SMBus Slave

The following sections provide the details on the UART and SMBus protocols.

8.7.1. UART

UART stands for Universal Asynchronous Receiver/Transmitter. It is not a communication protocol like SPI and I2C, but a physical circuit in a micro-controller, or a stand-alone IC.

The main purpose of the UART is to transmit and receive serial data. In UART communication, two UART communicate directly with each other. Only two wires are needed to transmit data between two UART. Data flows from the TX pin of the transmitting UART to the RX pin of the receiving UART.

Data is transmitted to the data bus by adding one start bit, one parity bit, and one stop bit to make it a packet. In the receiving part to retrieve data, start, parity, and stop bit are removed.

And in receiving part to retrieve data start, parity and stop bit is removed.

8.7.1.1. UART Frame Packet

- Start/Stop bit
- Data bits
- Parity bit

8.7.1.2. UART Functions

- Initialization of UART – Initializes the UART.
- Receiving Data – Receives data by removing the start, parity and stop bits.
- Transmitting Data – Sends data in a packet format by adding start, parity and stop bits.
- Setting baud rate – Sets the baud rate for UART as in case we need to change the baud rate this function is called.
- Configure UART – Configures the UART.

8.7.1.3. Functions for UART to PCIe Flow

UART to PCIe flow is used in encryption. In this flow, data packet is created with first byte as 0xcc, second byte as 0x77, and then next two bytes are used for packet size. This packet is sent over UART to PCIe. Following are the crypto algorithm flow with UART to PCIe.

8.7.2. SMBus

The SMBus Interface is only used for message exchange and communication between the Host PC and the Application CPU. Algorithm selection and other processes are performed through the SMBus.

8.7.2.1. Functions in SMBus

- int smbus_mailbox_write_data_register (unsigned char wbyte) – This function writes one-byte wbyte to WR_DATA_REG. It pushes one byte to Transmit FIFO. If the Transmit FIFO is full, this API returns an error. During SMBus read transaction, this data is shown from the Transit FIFO.
- int smbus_mailbox_read_data_register (unsigned char *rbyte) – This function reads one byte from the Receive FIFO. After a data is received from SMBus during write transaction, the received data is pushed to Receive FIFO. Reading from RD_DATA_REG shows a word from Receive FIFO. If the Receive FIFO is empty, this function returns an error.
- void smbus_mailbox_write_slave_address_register (unsigned short slv_id) – Sets up SMBus Slave ID for the IP block.
- void smbus_mailbox_read_slave_address_register (unsigned short *slv_id) – Reads SMBus Slave ID for the IP block.
- void smbus_mailbox_set_control_register (unsigned char wbyte) – Sets up control register
- void smbus_mailbox_read_control_register (unsigned char *rbyte) – Reads control register.
- void smbus_mailbox_read_interrupt_status1_register (unsigned char *rdata) – Reads interrupt status register INT_STATUS1_REG.
- void smbus_mailbox_write_rf (unsigned char waddr, unsigned char wbyte) – Writes the data to Register File inside SMBus Mailbox IP.
- void smbus_mailbox_read_rf (unsigned char waddr, unsigned char *rbyte) – Reads the data from the Register File inside SMBus Mailbox IP.

8.7.2.2. Register Address Set

- SMBUS_MAILBOX_BASE_ADDR
- SMBUS_MAILBOX_RF_OFFSET 0x2000
- SMBUS_MAILBOX_RF_ADDR
- (SMBUS_MAILBOX_BASE_ADDR + SMBUS_MAILBOX_RF_OFFSET)
- SMBUS_MAILBOX_RD_DATA_REG (SMBUS_MAILBOX_BASE_ADDR + 0x00)
- SMBUS_MAILBOX_WR_DATA_REG (SMBUS_MAILBOX_BASE_ADDR + 0x00)
- SMBUS_MAILBOX_SLVADR_L_REG (SMBUS_MAILBOX_BASE_ADDR + 0x04)
- SMBUS_MAILBOX_SLVADR_H_REG (SMBUS_MAILBOX_BASE_ADDR + 0x08)
- SMBUS_MAILBOX_CONTROL_REG (SMBUS_MAILBOX_BASE_ADDR + 0x0C)
- SMBUS_MAILBOX_TGT_BYTE_CNT_REG (SMBUS_MAILBOX_BASE_ADDR + 0x10)
- SMBUS_MAILBOX_INT_STATUS1_REG (SMBUS_MAILBOX_BASE_ADDR + 0x14)
- SMBUS_MAILBOX_INT_ENABLE1_REG (SMBUS_MAILBOX_BASE_ADDR + 0x18)
- SMBUS_MAILBOX_INT_SET1_REG (SMBUS_MAILBOX_BASE_ADDR + 0x1C)
- SMBUS_MAILBOX_INT_STATUS2_REG (SMBUS_MAILBOX_BASE_ADDR + 0x20)
- SMBUS_MAILBOX_INT_ENABLE2_REG (SMBUS_MAILBOX_BASE_ADDR + 0x24)
- SMBUS_MAILBOX_INT_SET2_REG (SMBUS_MAILBOX_BASE_ADDR + 0x28)
- SMBUS_MAILBOX_FIFO_STATUS_REG (SMBUS_MAILBOX_BASE_ADDR + 0x2C)

8.8. Code Flow

8.8.1. Application CPU Main Flow

The Application CPU performs message exchange with the Host PC and gives the mode of operation value to the Security CPU to perform a particular task. The Application CPU previously performs authentication and verifies the Host PC, which is followed by key exchange through ECDH algorithm. After the exchange of keys, other cryptographic algorithms are performed. For better understanding authentication and key exchange flows are there.

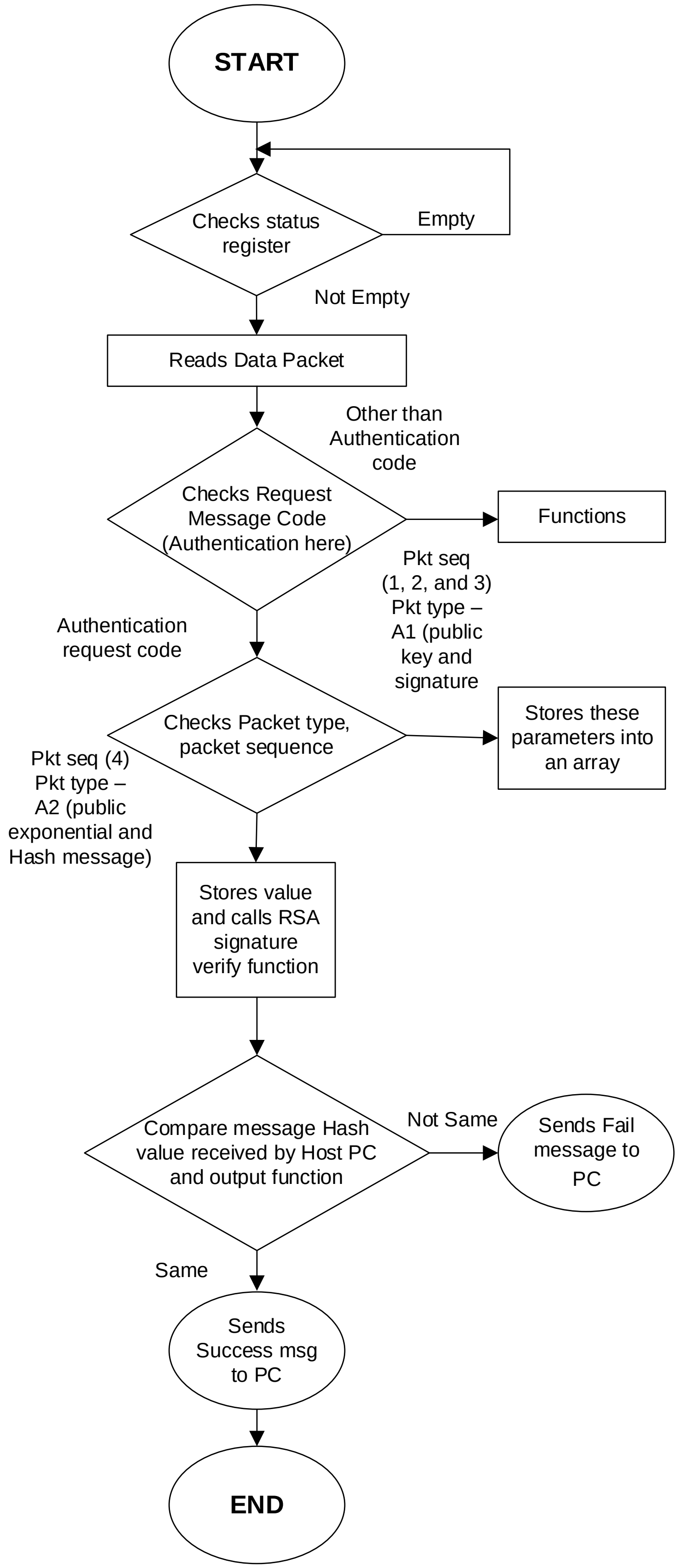


Figure 8.11. Authentication Flow

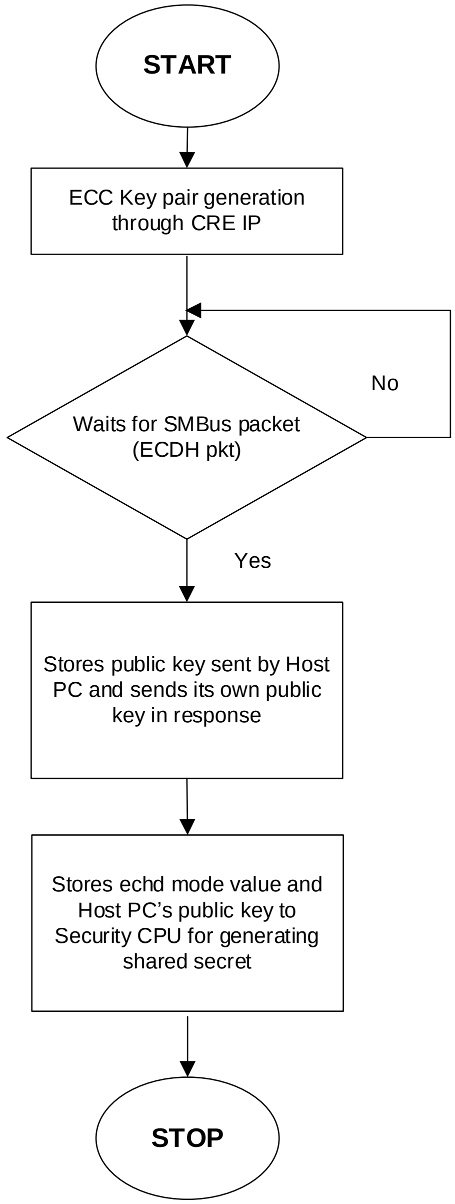


Figure 8.12. ECDH Flow

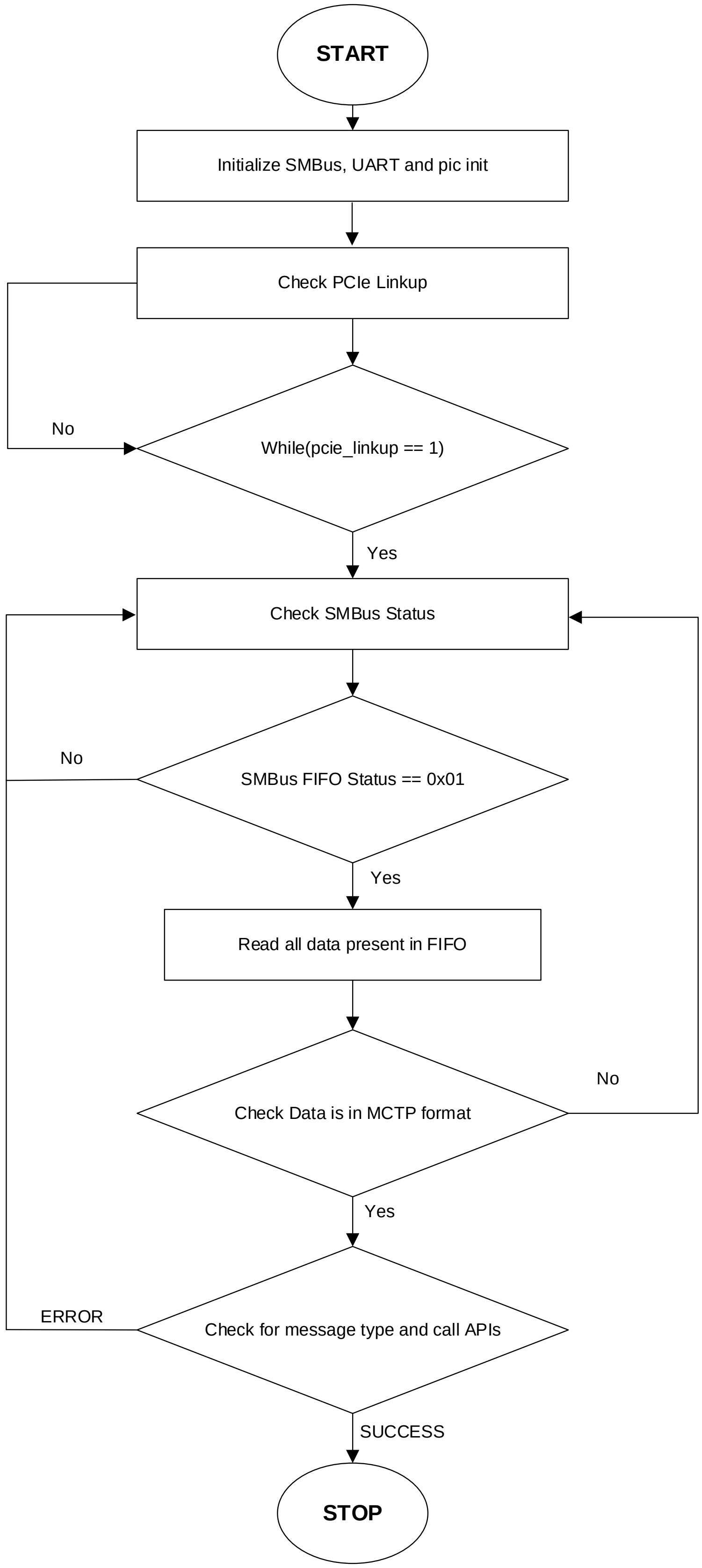


Figure 8.13. Main Code Flow

8.8.2. Algorithm APIs

There are two main flows for algorithms, one is for UART to PCIe and the other is for PCIe to UART. In PCIe to UART, data is sent by the Host PC to the Ingress RAM. The Application CPU reads and sends the data to the Security CPU. The Application CPU previously sends mode value (that decides the algorithm execution to perform) to the Security CPU, and sets interrupt for it. Until then it waits for the Application CPU interrupt which is set by the Security CPU once it has done its operation. The Application CPU then reads the output data and displays it on the UART terminal. Decryption is performed in PCIe to UART flow.

Following are the functions which are PCIe to UART:

- int aes_method_with_pcie (unsigned char *aes_key_arr, unsigned char *aes_iv_vector_arr, unsigned char mode_aes, int aes_sha_mode) – This API takes key, IV vector and mode value as input from the SMBus and sends it to the Security CPU for AES-CBC Decryption mode. In this API, the Application CPU writes key and IV vector, mode of AES, and data to register for the Security CPU. It then enables the interrupt for the Security CPU so that it can start processing. It waits for the Application CPU interrupt which comes from the Security CPU. After successfully completing AES operation, the Application CPU reads the data and displays it on UART terminal.
- unsigned int sha_with_pcie (unsigned char mode_sha, unsigned int sha_length) – This API is performs SHA384 operation. It takes mode and sha_length as input and gives the hash value of data as output. It initially writes SHA mode value and SHA length to the Security CPU. It calculates block size according to SHA length. SHA operation and interrupt enabling depends upon the SHA length as there are different conditions for length. One is length being equal to 128 bytes and another is length being less than 128 bytes. Data is sent through ping pong registers. After these operations, the API waits for the Application CPU interrupt to read the hash value of data as output.
- int sha_with_pcie_new (unsigned char mode_sha, unsigned int sha_length, unsigned char *aes_key_arr, unsigned char *aes_iv_vector_arr) – This API takes input for AES decryption and sends the output for SHA384 operation. The output is displayed on the UART terminal. This API mainly calls AES and SHA for operation and is used in PCIe to UART flow.
- unsigned int hmac_with_pcie (unsigned char *key, unsigned char mode_hmac, unsigned int sha_length) – This API is used to calculate hash value of data through HMAC Algorithm. It requires key as input. This is the same as the SHA API in terms of calculating block numbers from SHA length and performing pin pong operations. However, the Application CPU writes SHA length mode and key to the Security CPU.
- int aes_gcm (unsigned char *key, unsigned char *iv_vector, unsigned char *add_arr, unsigned int cipher_size) – This API works for AES GCM mode. It takes input key, IV vector, additional data, data length, and addition data length from the SMBus to the Security CPU. It then enables the interrupt for the Security CPU so that it can start its processing. Later, it waits for the Application CPU interrupt, which comes from the Security CPU. After successful completion of AES operation, the Application CPU reads the data and displays it on the UART terminal.

The following are the functions for UART to PCIe:-

For sending data from UART to PCIe, the user needs to create a packet on the UART terminal. Figure 8.14 shows the packet structure.

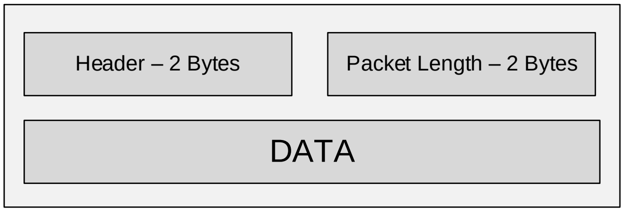


Figure 8.14. UART to PCIe

The header is = 0xcc and 0x77

- int aes_enc_uart_to_pcie (unsigned char mode_aes, unsigned char *aes_key_arr, unsigned char *aes_iv_vector_arr) – This API works for AES encryption. Input data is obtained from the UART and sent to the Security CPU. The output data is read once the Security CPU completes encryption and is sent to the Host PC through PCIe.
- int sha_uart_to_pcie (unsigned char mode_sha, unsigned int sha_length) – In this API, data is obtained from the UART. Data, key, IV vector and mode (encryption in this case) are sent to the AES API. The output of THE AES encryption data is sent for hash calculation through SHA384. The output hash value is then sent to the Host PC.
- int aes_gcm_uart_to_pcie (unsigned char *key, unsigned char *iv_vector, unsigned char *add_arr, unsigned int cipher_length) – This is for AES-GCM encryption calculation. The flow is the same as AES UART to PCIe. However, the key and IV vector additional data, data length, and additional data length are sent to the Security CPU.

8.9. Security CPU Main Flow

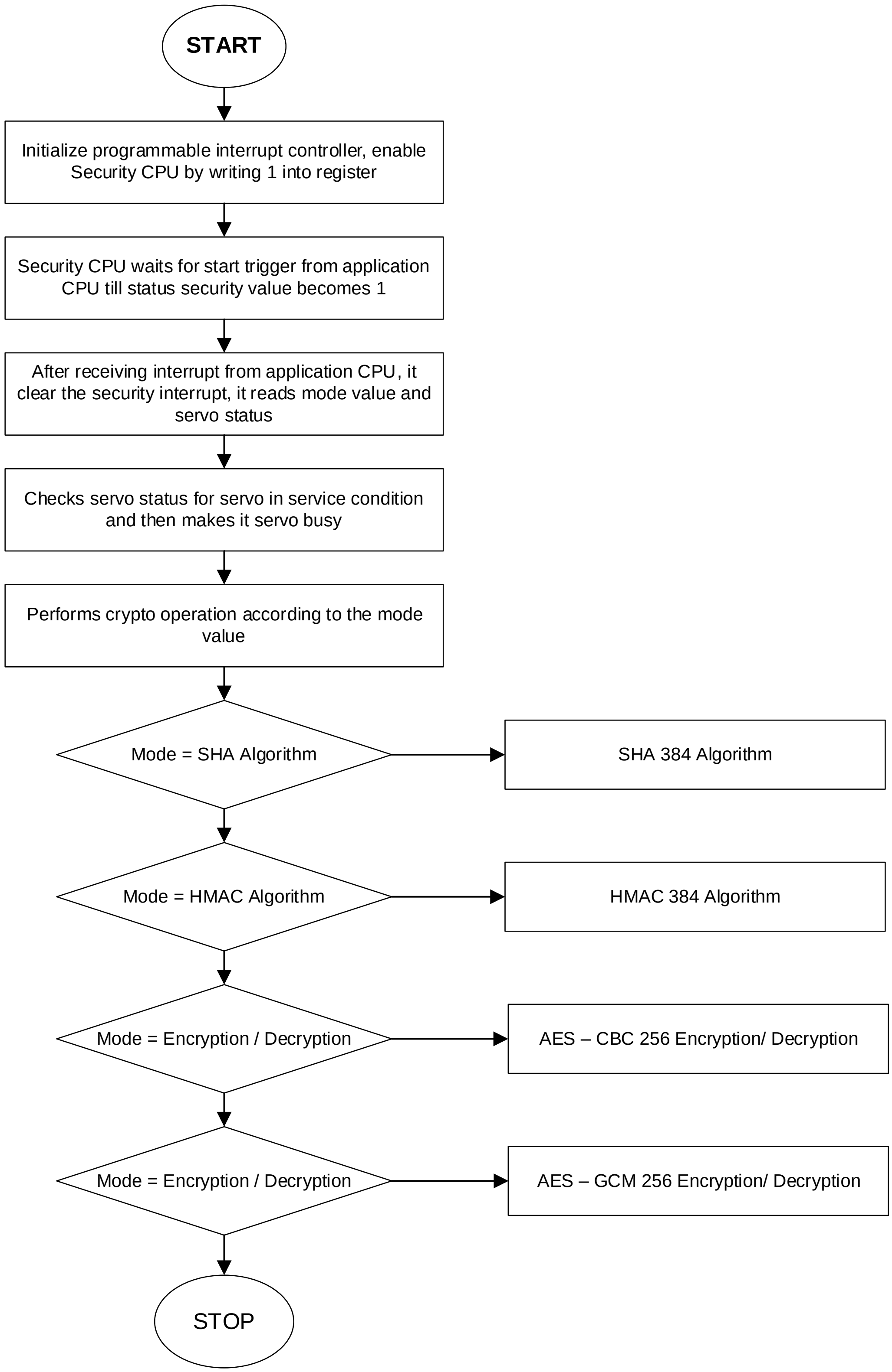


Figure 8.15. Security CPU Main Flow

8.10. Security CPU Algorithm APIs

In the Security CPU, there are common main flows for UART to PCIe and for PCIe to UART. In the case of PCIe to UART and UART to PCIe, data is read from the Register Interface by the Security CPU. The Application CPU sends the data through the Register Interface to the Security CPU. The Security CPU waits for interrupt to start the process and also reads the mode value and servo status sent by the Application CPU through the Register Interface to run the whole system for the selected algorithm based on the mode value.

Servo status has three operations:

- Idle (nothing is happening between application and the Security CPU).
- Service (the Application CPU starts sending data and interrupt to the Security CPU)
- Busy (the Security CPU is processing all the data sent by the Application CPU and sent it to the IP).

After completion of any algorithm, the Security CPU sends output data to the Application CPU through the Register Iinterface. The Application CPU reads output data and display it on the UART terminal.

The following are the functions used for PCIe to UART and UART to PCIe:

- unsigned int aes_enc_dec (unsigned int mode_value) – This API is for AES CBC 256 encryption/decryption. It reads key length, key, IV vector and based upon the mode value it, runs the encryption and decryption process through IP.
- unsigned int gcm_enc_dec (unsigned int mode_value) – This API is for AES GCM 256 encryption/decryption. It reads key length, key, IV vector, Additional data and Additional data length. Then, based upon the mode value, it runs the encryption and decryption process through IP.
- unsigned int sha384_sec() – This API is used for SHA384 Algorithm, it reads data length from SMBus, input data from the Register Interface. It uses HASH INITIAL to tell IP to process the data and configures HASH UPDATE to tell IP to process if data is greater than 128 bytes. HASH FINISH only configures for last block of input data and also sends data length to IP. After complete processing of data, IP generates Digest value of 384 bits and then sends it to the Application CPU through the Register Interface.
- unsigned int hmac_sec() – This API is used for HMAC SHA384 Algorithm, it reads data length from SMBus, input key and input data from the Register Interface. HMAC algorithm used IPAD (0x36) and OPAD (0x5C) value for XORing the input key before sending to IP. It uses HASH INITIAL to tell IP to process the data and configures HASH UPDATE to tell IP to process if data is greater than 128 bytes. HASH FINISH only configures for last block of input data and also sends data length to the IP. After the data is completely processed, the IP generates Digest value of 384 bits and then sends it to the Application CPU through the Register Interface.
- unsigned int sha_ping_block (unsigned int sha_input_length) – This API is used to read the data sent by the Application CPU through the Register Interface. The sha_input_length indicates the number of bytes to be read from the ping memory and sends it to the IP. To indicate that the entire data is read from this location, 0 is written at the end
- unsigned int sha_pong_block (unsigned int sha_input_length) – This API is used to read the data sent by the Application CPU through the Register Interface. The sha_input_length indicates the number of bytes to be read from the pong memory and sends it to the IP. To indicate that the entire data is read from this location, 0 is written at the end.
- unsigned int Hash_finish (unsigned int sha_input_length, unsigned char pingpong) – This API is used to perform operation on the last block of input data to complete the process. The ping-pong variable keeps track of the memory from which the data is read, either from the ping or the pong memory. The sha_input_length indicates the number of bytes to be read from the specific memory and sends it to the IP.
- unsigned int rsa_key_verify() – This API is used to read the public key and signature of 3072 bits from the ping and pong buffer through the Register Interface, in which the Application CPU writes these data. The RSA verification is performed using CRE IP. Output data is sent to the Application CPU through the Register Interface.
- unsigned int rsa_ping_block (unsigned int sha_input_length) – This API is used to read the data sent by the Application CPU through the Register Interface. The sha_input_length indicates the number of bytes to be read from the ping memory and sends it to the IP. To indicate that the entire data is read from this location, 0 is written at the end.
- unsigned int rsa_pong_block (unsigned int sha_input_length) – This API is used to read the data sent by the Application CPU through the Register Interface. The sha_input_length indicates the number of bytes to be read from the pong memory and sends it to the IP. To indicate that the entire data is read from this location, 0 is written at the end.
- unsigned char ecdh_algo() – This API is used to exchange shared secret between the Application CPU and the Security CPU. The Security CPU reads the public key x, public key y, and private key from the buffer and send the values to the

IP. After the IP completes the operation, the Security CPU sends the generated shared secret to the Application CPU through the Register Interface.

Note: For more details, refer to the Detailed Description of Crypto Operations section for all APIs.

Appendix A. Resource Utilization

Table A.1. Resource Utilization

Subsystems	Blocks	LUTs	PFU Registers	EBRs	DSPs
	Reset_Sync	79	35	0	0
Crypto-256	ahbl2lmmi	288	199	0	0
	CRE	0	0	0	0
	Sub-total (Crypto-256)	288	199	0	0
Crypto-384	AHBL Interconnect (1 masters, 3 slaves)	143	7	0	0
	Registers Interface(16 kB)	198	136	8	0
	Security RISC-V	2399	1008	2	0
	System (Instruction + Data) Memory (128 kB)	88	35	2 LRAMs	0
	PKC	5672	1314	6	16
	SHA2-384	4006	2882	0	0
	AES CBC	*9741*	2017	*40*	0
	AES-GCM	*9722*	2924	*24*	0
	ORAN Security Enclave top wrapper (interconnect + top wrapper)	*88*	36	*0*	0
	Sub-total (ORAN Security Enclave without CBC module)	19488	7156	30	16
	Sub-total (Crypto-384)	22316	8342	40B , 2L	16
PCIe	Ingress RAM (64 kB)	259	328	32	0
	Egress RAM (64 kB)	625	819	32	0
	PCIe with DMA (2 AHBL masters, 1 APB)	20383	11619	40	0
	Sub-total	21267	12766	104	0
Application	AHBL Interconnect of Application (1 master, 7 slaves)	297	11	0	0
	AHBL TO APB Interconnect	206	254	0	0
	APB Interconnect (1 master, 2 slaves)	54	3	0	0
	Application CPU (RISC-V)	2710	1449	2	0
	System (Instruction + Data) Memory (128 kB)	119	35	2 LRAMS	0
	UART	254	146	0	0
	SMBus Slave	1271	701	2	0
	Sub-total	4911	2599	4B,2L	0
Total Used		*48861*	*23941*	*148B,4L*	*16*
Total Available Resources		79872	79872	208 B,7 L	156

Technical Support Assistance

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

Revision History

Revision 1.0, June 2022

Section	Change Summary
All	Initial release.