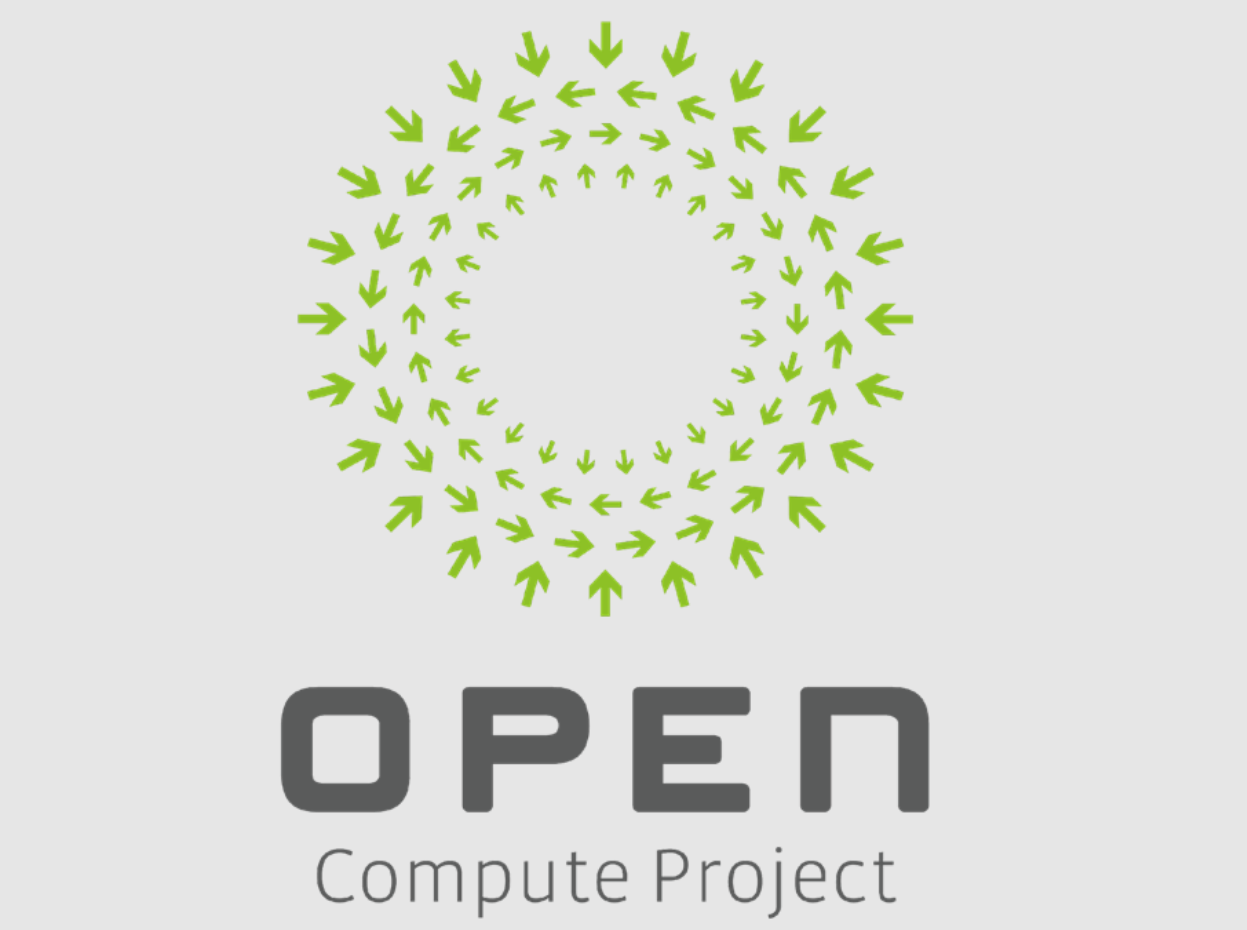



Caliptra Hardware Specification

V0.5

June 2022

CONTRIBUTORS:

Caliptra Consortium

June 2022

Revision Table

Date	Revision #	Description
6/24/2022	0.1	Initial draft

5

TODO List

Date	Author	Description
6/24/2022		AES can support AES GSM, AES CBC, and AES CTR
6/24/2022		SCA countermeasures for each component will be added.

Table of Contents

This document defines technical specifications for a Caliptra RTM[1] crypto subsystem used in Open Compute Project. This document, along with the [baseline specification] shall comprise product's technical specification.

1. Overview

This document provides definitions and requirements for a Caliptra crypto subsytem. The document then relates these definitions to existing technologies, enabling third device and platform vendors to better understand those technologies in trusted computing terms.

1.1. Acronyms and Abbreviations

For the purposes of this document, the following abbreviations apply:

Abbreviation	Description
AES	Advanced Encryption Standard
BMC	Baseboard Management Controller
CA	Certificate Authority
CDI	Composite Device Identifier
CPU	Central Processing Unit
CRL	Certificate Revocation List
CSR	Certificate Signing Request
CSP	Critical Security Parameter
DICE	Device Identifier Composition Engine
DME	Device Manufacturer Endorsement
DPA	Differential Power Analysis
DRBG	Deterministic Random Bit Generator
ECDSA	Elliptic Curve Digital Signature Algorithm

[1] *Caliptra. Spanish for "root cap" and describes the deepest part of the root*

FMC	FW First Mutable Code	
GPU	Graphics Processing Unit	
HMAC	Hash-based message authentication code	
IDevId	Initial Device Identifier	
iRoT	Internal RoT	
KAT	Known Answer Test	
KDF	Key Derivation Function	
LDevId	Locally Significant Device Identifier	
MCTP	Management Component Transport Protocol	
NIC	Network Interface Card	
NIST	National Institute of Standards and technology	
OCP	Open Compute Project	
OTP	One-time programmable	
PCR	Platform Configuration Register	
PKI	Public Key infrastructure	
PUF	Physically unclonable function	
RoT	Root of Trust	
RTI	RoT for Identity	
RTM	RoT for Measurement	
RTR	RoT for Reporting	
SCA	Side-Channel Analysis	
SHA	Secure Hash Algorithm	
SoC	System on Chip	
SPA	Simple Power Analysis	

SPDM	Security Protocol and Data Model	
SSD	Solid State Drive	
TCB	Trusted Computing Base	
TCI	TCB Component Identifier	
TCG	Trusted Computing Group	
TEE	Trusted Execution Environment	
TRNG	True Random Number Generator	
UECC	Uncorrectable Error Correction Code	

1.2. Requirements Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [BCP 14] [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.3. References

[1]	J. Strömbergson, "Secworks," [Online]. Available: https://github.com/secworks.
[2]	NIST, Federal Information Processing Standards Publication (FIPS PUB) 180-4 Secure Hash Standard (SHS).
[3]	OpenSSL. [Online]. Available: https://www.openssl.org/docs/man3.0/man3/SHA512.html.
[4]	N. W. Group, RFC 3394, Advanced Encryption Standard (AES) Key Wrap Algorithm, 2002.
[5]	NIST, Federal Information Processing Standards Publication (FIPS) 198-1, The Keyed-Hash Message Authentication Code, 2008.
[6]	N. W. Group, RFC 4868, Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec, 2007.
[7]	RFC 6979, Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA), 2013.
[8]	TCG, Hardware Requirements for a Device Identifier Composition Engine, 2018.
[9]	Coron, J.-S.: Resistance against differential power analysis for elliptic curve cryptosystems. In: Ko¸c, C¸ .K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 292–302.
[10]	Schindler, W., Wiemers, A.: Efficient side-channel attacks on scalar blinding on elliptic curves with special structure. In: NISTWorkshop on ECC Standards (2015)
[11]	National Institute of Standards and Technology, "Digital Signature Standard (DSS)", Federal Information Processing Standards Publication (FIPS PUB) 186-4, July 2013.
[12]	NIST SP 800-90A, Rev 1: "Recommendation for Random Number Generation Using Deterministic Random Bit Generators", 2012
	<TODO> PRM + RV spec

2. Caliptra Block Diagram

Caliptra top-level block diagram is shown in the figure below.

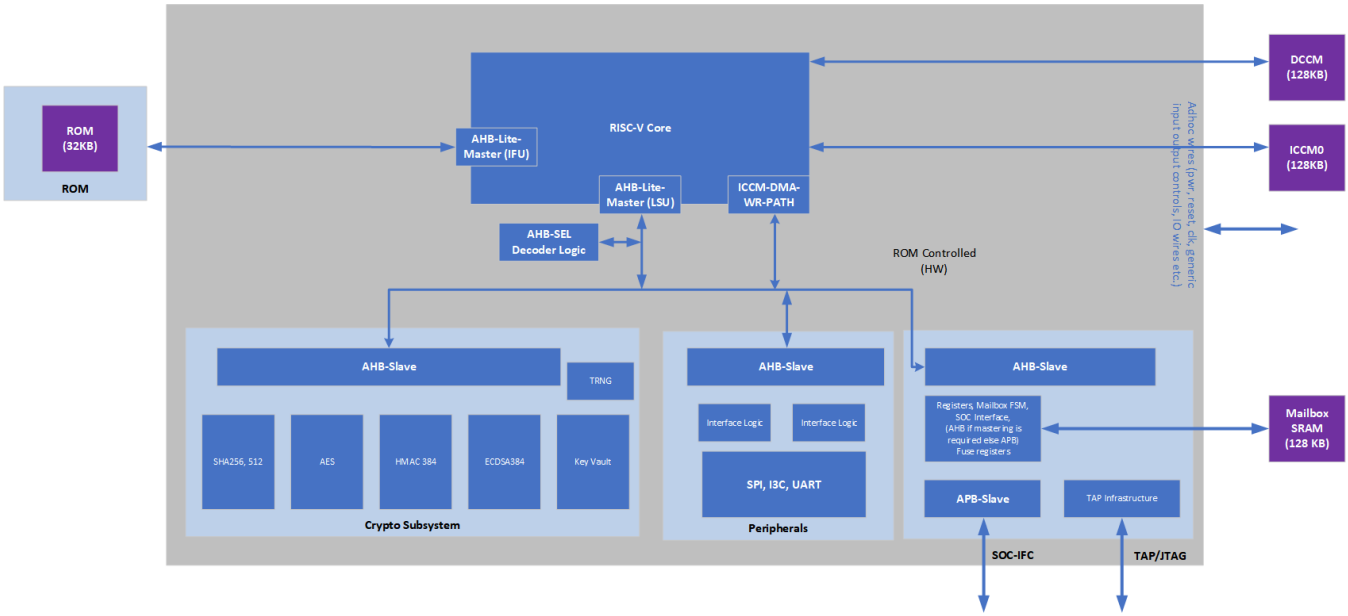


Figure 1: Caliptra Block Diagram

2.1. Boot Media Independent vs Boot Media Integrated

In boot media independent (used to be called passive profile), none of the IOs in the peripherals are active. This will be an integration time parameter passed to the HW which is exposed to ROM. Please see boot flows to see the difference in the HW/ROM behavior for passive profile vs active profile.
From SOC integration POV, peripheral IOs can be tied off appropriately for passive profile at SOC integration time.

2.2. Internal Blocks

2.2.1. BootFSM

The Boot FSM is responsible for detecting the SoC bringing Caliptra out of reset. Part of this flow involves signaling to the SoC that we are awake and ready for fuses. Once fuses have been populated and the SoC has indicated that they are done downloading fuses, we can wake up the rest of the IP by de-asserting the internal reset.

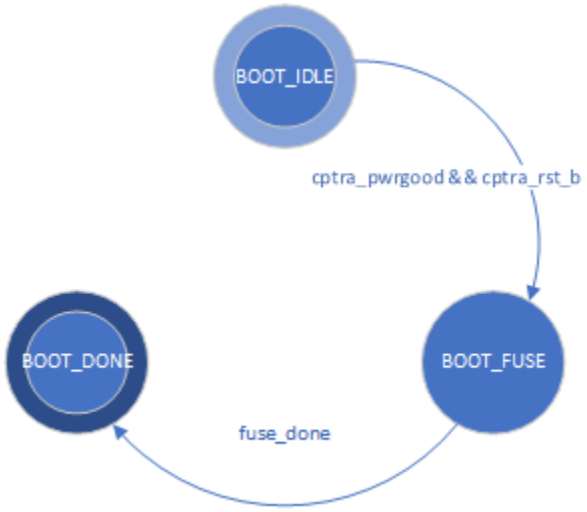


Figure 2: Mailbox Boot FSM State Diagram

The boot FSM first looks for the SoC to assert cptra_pwrgood and de-assert cptra_rst_b. In the BOOT_FUSE state, Caliptra will signal to the SoC that it is ready for fuses. Once the SoC is done writing fuses, it will set the fuse done register and the FSM will advance to BOOT_DONE. BOOT_DONE enables Caliptra reset de-assertion through a two flip-flop synchronizer.

2.2.2. FW Update Reset (Impactless FW Update)

Runtime FW updates write to fw_update_reset register to trigger the FW update reset. When this register is written, only the RISC-V core is reset using cptra_uc_fw_rst_b pin and all AHB slaves are still active. All registers within the slaves and ICCM/DCCM memories are intact after the reset. Since ICCM is locked during runtime, it must be unlocked once the RISC-V reset is asserted. Reset is deasserted synchronously after a programmable number of cycles (currently set to 5 clocks) and normal boot flow will update the ICCM with the new FW from the mailbox SRAM. Reset de-assertion is done through a two flip-flop synchronizer. The boot flow is modified as shown in state diagram below:

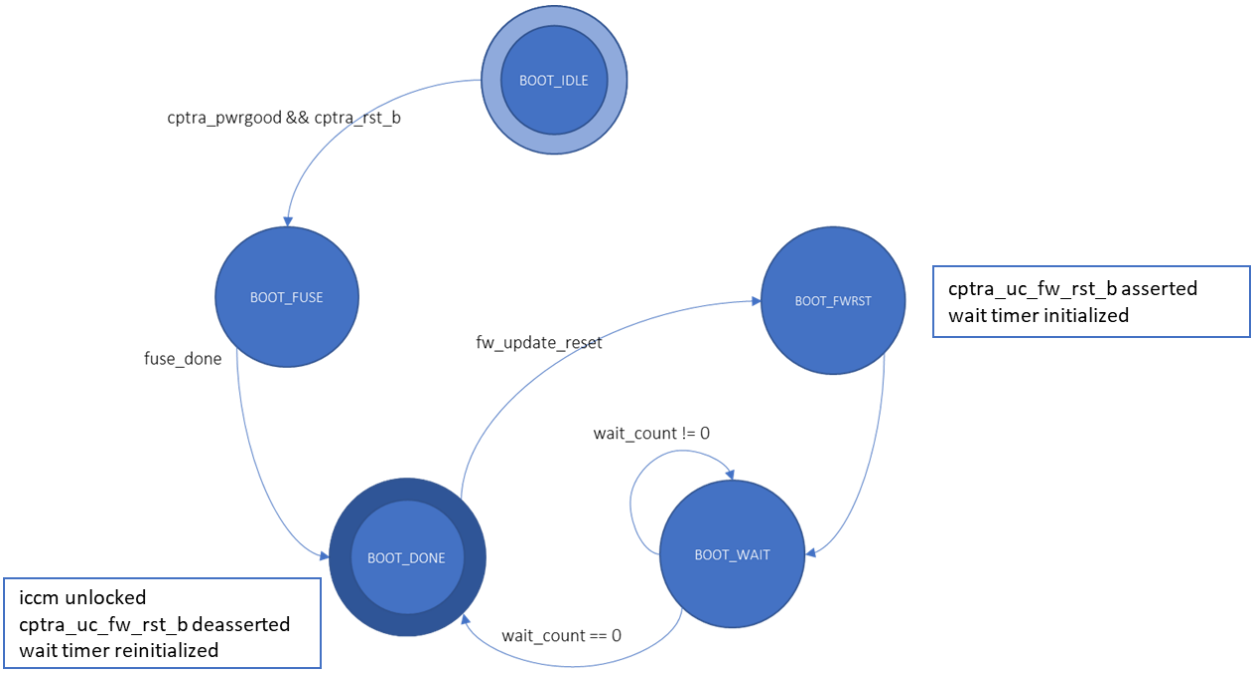


Figure 3: Mailbox Boot FSM State Diagram for FW update reset

Once Caliptra comes out of global reset and enters BOOT_DONE state, a write to the fw_update_reset register will trigger the FW update reset flow. In BOOT_FWRST state, only the reset to the VeeR core is asserted, ICCM is unlocked and the timer is initialized. Once the timer expires, the FSM advances from BOOT_WAIT to BOOT_DONE state where the reset is deasserted.

Control Register	Start Address	Desc
FW_UPDATE_RESET	0x30030418	Register to trigger FW update reset flow. Setting it to 1 will start the boot FSM. Field auto-clears to 0.
FW_UPDATE_RESET_WAIT_CYCLES	0x3003041C	Programmable wait time to keep the uC reset asserted.

1.1.3. RISC-V Core

The RISC-V core is VeeR EL2 from Chips Alliance. It is 32-bit CPU core that contains a 4-stage, scalar, in-order pipeline. The core supports RISC-V's integer(I), compressed instruction(C), multiplication and division (M), instruction-fetch fence, CSR, and subset of bit manipulation

instructions (Z) extensions. A link to the RISC-V VeeR EL2 Programmer's Reference Manual is provided in the Related Specifications section.

1.1.3.1. Configuration

The RISC-V core is highly configurable and has the following settings.

Parameter	Configuration
Interface	AHB-Lite
DCCM	128kB
ICCM	128kB
I-Cache	Disabled
Reset Vector	0x00000000
Fast Interrupt Redirect	Enabled
External Interrupts	31

Table 1: RISC-V Configuration

1.1.3.2. Embedded Memory Export

Internal RISC-V SRAM memory components are exported from the Caliptra subsystem to support adaptation to various fabrication processes. Refer to the Integration Specification for more detail.

1.1.3.3. Memory Map Address Regions

The 32-bit address region is subdivided into 16 fixed-sized, contiguous 256 MB regions. The following table describes the address mapping for each of the AHB devices that the RISC-V core interfaces with.

Subsystem	Address Size	Start Address	End Address
Crypto	512 KB	0x1000_0000	0x1007_FFFF
Peripherals	32 KB	0x2000_0000	0x2000_7FFF
SOC IFC	256 KB	0x3000_0000	0x3003_FFFF
RISC-V Core ICCM	128 KB	0x4000_0000	0x4001_FFFF
RISC-V Core DCCM	128 KB	0x5000_0000	0x5001_FFFF

RISC-V MM CSR (PIC)	256 MB	0x6000_0000	0x6FFF_FFFF

Table 2: Memory Map Address Regions by Subsystem

1.1.3.3.1. Crypto Subsystem

The table below shows the memory map address ranges for each of the IP blocks in the crypto subsystem.

IP/Peripheral	Slave #	Address Size	Start Address	End Address
Crypto Init	0	32KB	0x1000_0000	0x1000_7FFF
ECC Secp384	1	32KB	0x1000_8000	0x1000_FFFF
HMAC384	1	32KB	0x1001_0000	0x1001_7FFF
Key Vault	2	32KB	0x1001_8000	0x1001_FFFF
SHA512	3	32KB	0x1002_0000	0x1002_7FFF
SHA256	4	32KB	0x1002_8000	0x1002_FFFF

Table 3: Crypto Subsystem Memory Map

1.1.3.3.2. Peripherals Subsystem

The table below shows the memory map address ranges for each of the IP blocks in the peripherals' subsystem.

IP/Peripheral	Slave #	Address Size	Start Address	End Address
QSPI	5	4KB	0x2000_0000	0x2000_0FFF
UART	6	4KB	0x2000_1000	0x2000_1FFF
CSRNG	13	4KB	0x2000_2000	0x2000_2FFF
ENTROPY SRC	14	4KB	0x2000_3000	0x2000_3FFF
~~**I3C***~~	~~7~~	~~4KB~~	~~0x2000_2000~~	~~0x2000_2FFF~~

* I3C will not be included in Generation 1 of Caliptra

Table 4: Peripherals' Subsystem Memory Map

1.1.3.3.3. SOC Interface

The table below shows the memory map address ranges for each of the IP blocks in the SOC interface subsystem.

IP/Peripheral	Slave #	Address Size	Start Address	End Address
Mailbox	8	256KB	0x3000_0000	0x3003_FFFF

Table 5: SOC Interface Subsystem Memory Map

1.1.3.3.4. RISC-V Core Local Memories

The table below shows the memory map address ranges for each of the local memory blocks that interface with RISC-V core.

IP/Peripheral	Slave #	Address Size	Start Address	End Address
ICCM0 (via DMA)	9	128KB	0x4000_0000	0x4001_FFFF
DCCM	N/A	128KB	0x5000_0000	0x5001_FFFF

Table 6: RISC-V Local Memories Memory Map

1.1.3.4. Interrupts

The VeeR-EL2 processor supports multiple types of interrupts, including Non-maskable Interrupts (NMI), Software Interrupts, Timer Interrupts, External Interrupts, and Local Interrupts (events not specified by the RISC-V standard, such as auxiliary timers and correctable errors.
Caliptra uses NMI in conjunction with a watchdog timer to support fatal error recovery and system restart. The Watchdog feature is described in more detail later.

Software, Timer, and Local Interrupts are unimplemented in the 0p5 revision of Caliptra.
0p8 specification (or later) of Caliptra will document Timer interrupt usage for firmware optimization and task management.
<TODO> 0p8 review this

1.1.3.4.1. Non-Maskable Interrupts

<TODO> 0p8 describe a register bank that may be used to dynamically configure the NMI reset vector. (i.e., where the PC resets to).

1.1.3.4.2. External Interrupts

Caliptra uses the external interrupt feature to support event notification from all attached peripheral components in the subsystem. The RISC-V processor supports multiple priority levels (ranging from 1-15), which allows firmware to configure Interrupt priority per component.
Errors and Notifications are allocated as interrupt events for each component, with Error Interrupts assigned a higher priority and expected to be infrequent.
Notification Interrupts are used to alert the processor of normal operation activity, such as completion of requested operations or arrival of SoC requests through the shared interface.
Vector 0 is reserved by the RISC-V processor and may not be used, so vector assignment begins with Vector 1. Bit 0 of the interrupt port to the processor corresponds with Vector 1.

IP/Peripheral	Interrupt Vector	Interrupt Priority Example (Increasing, Max 15)
AES (Errors)	1	8
AES (Notifications)	2	7
ECC (Errors)	3	8
ECC (Notifications)	4	7
HMAC (Errors)	5	8
HMAC (Notifications)	6	7
KeyVault (Errors)	7	8
KeyVault (Notifications)	8	7
SHA512 (Errors)	9	8
SHA512 (Notifications)	10	7
SHA256 (Errors)	11	8
SHA256 (Notifications)	12	7
QSPI (Errors)	13	4
QSPI (Notifications)	14	3
UART (Errors)	15	4
UART (Notifications)	16	3
I3C (Errors)	17	4

I3C (Notifications)	18	3
Mailbox (Errors)	19	8
Mailbox (Notifications)	20	7

Table 7: RISC-V External Interrupt Vector Assignments

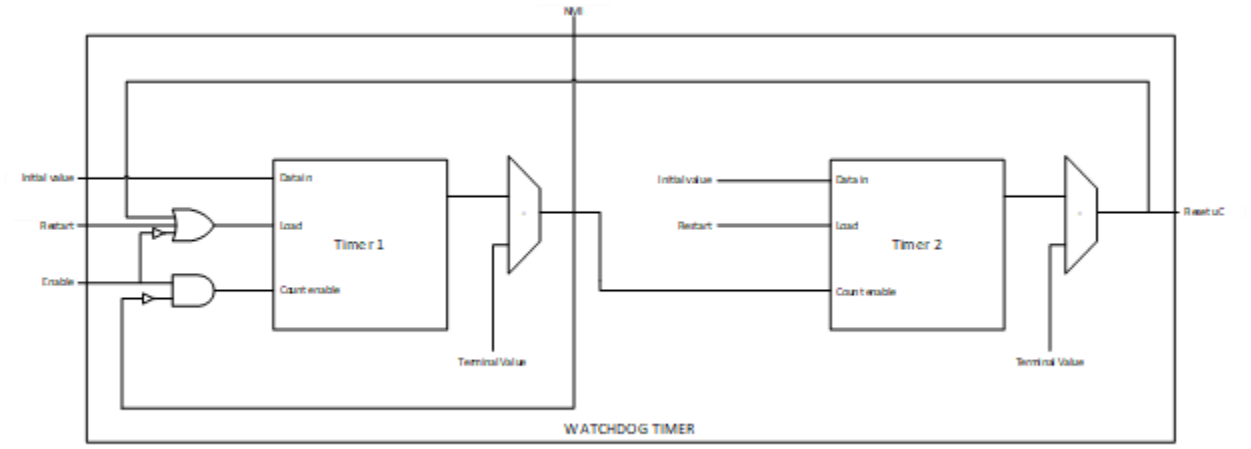
1.1.4. Watchdog Timer

The primary function of Caliptra Watchdog Timer (WDT) is to reset the microcontroller (Caliptra), in the event of a software malfunction, by resetting the device if it has not been cleared in software. It is a two-stage timer, independent of the RISCV core.

- #### 1.1.4.1 Operation

The WDT consists of two timers. When enabled, the WDT will increment Timer 1 until the counter rolls over or "times out". Typically, the timer is 'nudged' at regular intervals to prevent it from overflowing or rolling over. If Timer 1 has not timed out, Timer 2 is disabled and held at its initial value. However, when Timer 1 does roll over, it triggers a notification interrupt to the RISCV core. In parallel. Timer 2 is enabled. If the interrupt is processed before Timer 2 times out, the timers are reset and continue to operate normally. If Timer 2 times out, it asserts SOC fatal error and an NMI. The SOC fatal error is also captured in bit [0] of the CPTRA_HW_ERROR_FATAL register, which can be cleared by the SOC by writing a 1. A warm reset is required by the SOC to reset the timers when timer2 times out.

The WDT timers can be configured to operate independent of each other. When timer2 enable register is set, the default configuration of cascaded timers is disabled and both timers count independent of each other. In this case, a timeout on timer2 causes a notification interrupt to the RISCV core similar to timer1. Disabling timer2 will configure the timers back into the default cascaded mode.



- 1.1.4.2 Register Interface

Register	Start Address	Desc
CPTRA_WDT_TIMER1_EN	0x300300e0	Enable/Disable timer1. '1' written to this location enables the WDT timer1.
CPTRA_WDT_TIMER1_CTRL	0x300300e4	Software must write to this location periodically to reinitialize WDT Timer 1. This is required to keep the timer from timing out.
CPTRA_WDT_TIMER1_TIMEOUT_PERIOD [1:0]	0x300300e8	Specifies 64-bit timeout period for timer1. When timer1 reaches this value, a notification interrupt is triggered and timer2 is enabled. When the interrupt is cleared, both timer1 and timer2 are reset to 0 (provided timer2 has not timed out), timer2 is disabled and timer1 continues to operate normally. Set to a default value of FFs

CPTRA_WDT_TIMER2_EN	0x300300f0	Enable/Disable timer2. '1' written to this location enables the WDT timer2 and disables the default cascade configuration. This will cause both timers to operate independently
CPTRA_WDT_TIMER2_CTRL	0x300300f4	Software must write to this location periodically to reinitialize WDT Timer 2. This is required to keep the timer from timing out.
CPTRA_WDT_TIMER2_TIMEOUT_PERIOD [1:0]	0x300300f8	Specifies 64-bit timeout period for timer2. When timer2 reaches this value, a notification interrupt is triggered. When the interrupt is cleared, timer2 is reset to 0 and continues to operate normally. Set to default value of FFs
CPTRA_WDT_STATUS	0x30030100	This register is used to indicate the status of the WDT timers. [0]: Timer1 Watchdog timeout. A '1' written to this location indicates that Timer 1 has timed out and Timer 2 has been enabled. [1]: Timer2 Watchdog timeout. A '1' written to this location indicates that Timer 2 has timed out. This will assert an NMI to the microcontroller and an SOC fatal error.

- 1.1.4.3 Prescale settings

Assuming a clock source of 500 MHz, a timeout value of 32'hFFFF_FFFF will result in a timeout period of ~8.5 seconds. Two 32-bit registers are provided for each timer that can be programmed with timeout periods.

1.1.5. uC Interface

The Caliptra uC communicates with the mailbox through its internal AHB-Lite fabric.

1.1.5.1. AHB Lite Bus Interface

AHB-Lite is a subset of the full AHB specification. It is primarily used in single master systems. This interface connects VeeR EL2 Core (LSU master) to the slave devices as shown in the block diagram in Figure 1.
The interface can be customized to support variable address & data widths, and variable number of slave devices. Each slave device is assigned an address range within the 32-bit address memory map region. The interface includes address decoding logic to route data to the appropriate AHB slave device based on the address specified.
The integration parameters for AHB Lite Bus are as follows:

Parameter	Value
ADDRESS_WIDTH	32
DATA_WIDTH	64
NUM_OF_SLAVES	11

Table 8: AHB Lite Bus Integration Parameters

<TODO> Caleb add note about 32/64 implementation in crossbar

1.1.6. Crypto Subsystem

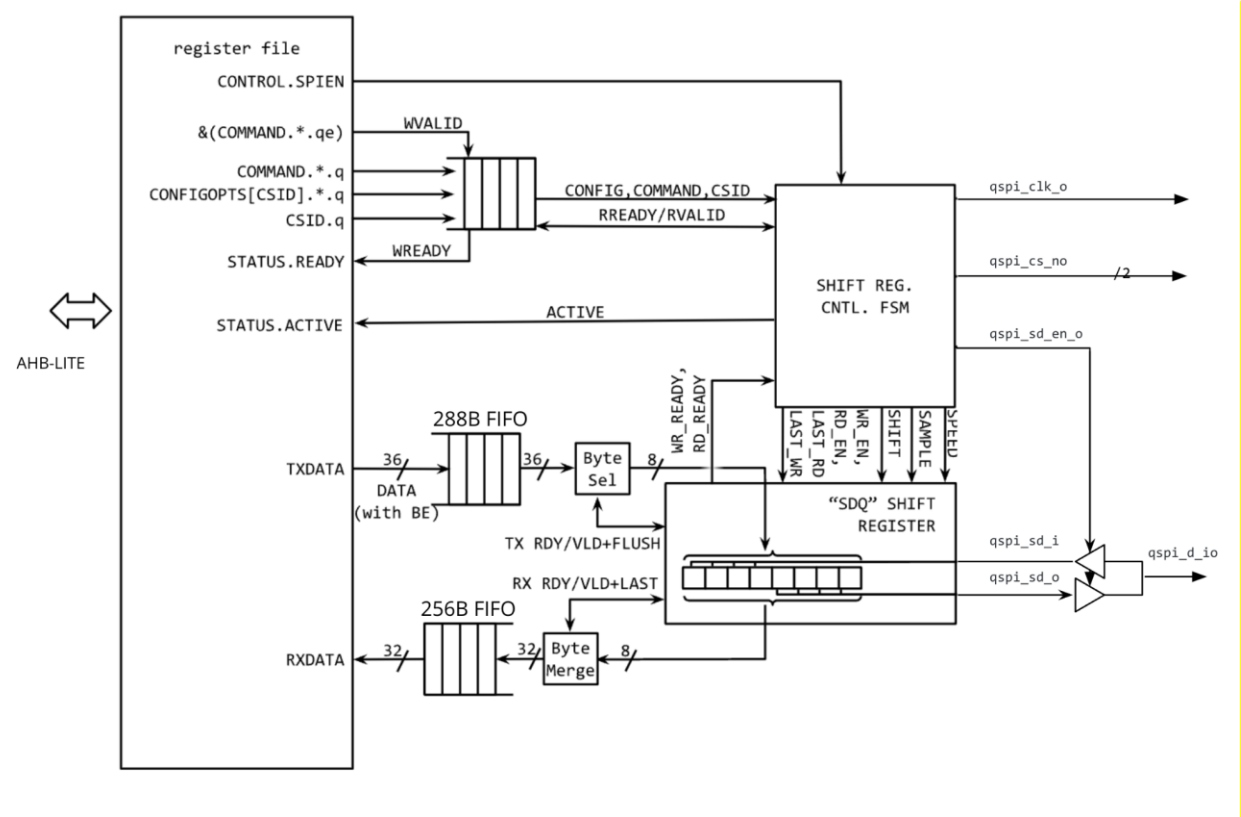
The details for the Crypto subsystem are included in a separate chapter

1.1.7. Peripherals Subsystem

1.1.7.1. QSPI Flash Controller

Caliptra implements a QSPI block that can communicate with 2 QSPI devices that is accessible to FW over the AHB-Lite Interface. This is a configuration that SOC opts-in by defining CALIPTRA_INTERNAL_QSPI.

The QSPI block is composed of the spi_host (doc) implementation. The core code (spi_host) is reused but the interface to the module is changed to AHB-Lite and the number of chip select lines supported is increased to 2. The design provides support for Standard SPI, Dual SPI, or Quad SPI commands.

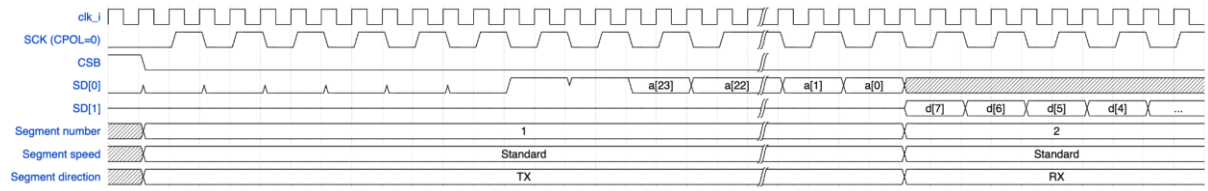


1.1.7.1.1. Operation

Transactions flow through the QSPI block starting with AHB-Lite writes to the TXDATA fifo. Commands are then written and processed by the control FSM, orchestrating transmissions from the TXDATA fifo and receiving data into the RXDATA fifo.

The structure of a command depends on the device and the command itself. In the case of a standard SPI device, the host ip will always transmit data on qspi_d_io[0] and always receive data from the target device on qspi_d_io[1]. In Dual or Quad modes, all data lines are bi-directional thus allowing full bandwidth in transferring data across 4 data lines.

A typical SPI command consists of different segments that are combined as shown below. Each segment can configure the length, speed, and direction. As an example, the following SPI read transaction consists of 2 segments:



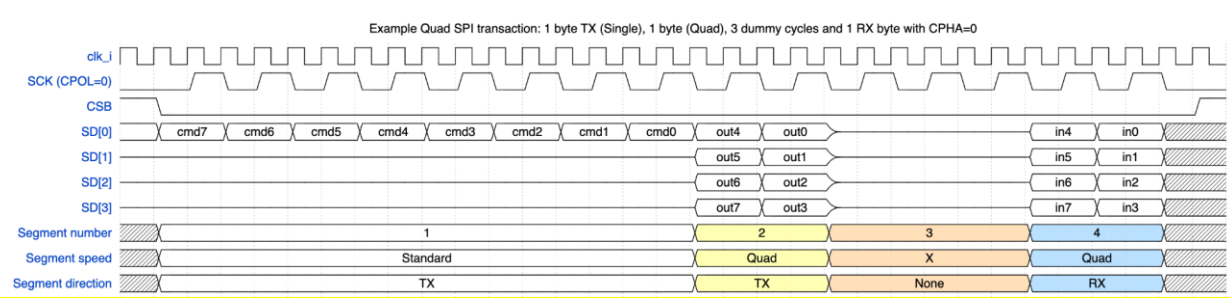
Standard SPI example: Flash Read command with 24-bit address, consisting of one TX and one RX segment

Segment #	Length (Bytes)	Speed	Direction	TXDATA FIFO	RXDATA FIFO
1	4	standard	TX qspi_d_io[0]	[0] 0x3 (ReadData) [1] Addr[23:16] [2] Addr[15:8] [3] Addr[7:0]	
2	1	standard	RX qspi_d_io[1]		[0] Data [7:0]

Here we can see that the ReadData (0x3) command was written to the TXDATA fifo, followed by the 3B address. This maps to a total of 4 bytes that will be transmitted out across qspi_d_io[0] in the first segment. The second segment consists of a read command that will receive 1 byte of data from the target device across qspi_d_io[1].

QSPI can be thought of consisting up to four command segments in which the host:

1. Transmits instructions or data at the standard rate
2. Transmits instructions address or data on 2 or 4 data lines
3. Holds the bus in a high-impedance state for some number of dummy cycles where neither side transmits
4. Receives information from the target device at the specified rate (derived from the original command)



Segment	Length	Speed	Direction	TXDATA FIFO	RXDATA FIFO

#	(Bytes)				
1	1	standard	TX qspi_d_io[3:0]	[0] 0x6B (ReadDataQuad)	
2	3*	quad	TX qspi_d_io[3:0]	[1] Addr[23:16] [2] Addr[15:8] [3] Addr[7:0]	
3	2	N/A	None (Dummy)		
4	1	quad	RX qspi_d_io[3:0]		[0] Data[7:0]

*Note: In the diagram above, the segment 2 doesn't show bytes 2+3 for brevity.

1.1.7.1.2. Configuration

The CONFIGOPTS multi-register has one entry per CSB line and holds clock configuration and timing settings which are specific to each peripheral. Once the CONFIGOPTS multi-register has been programmed for each SPI peripheral device, the values can be left unchanged.

The most common differences between target devices are the requirements for a specific SPI clock phase or polarity, CPOL and CPHA. These clock parameters can be set via the CONFIGOPTS.CPOL or CONFIGOPTS.CPHA register fields.

The SPI Clock rate depends on the peripheral clock and a 16b clock divider configured by CONFIGOPTS.CLKDIV. The following equation is used to configure the SPI Clock Period:

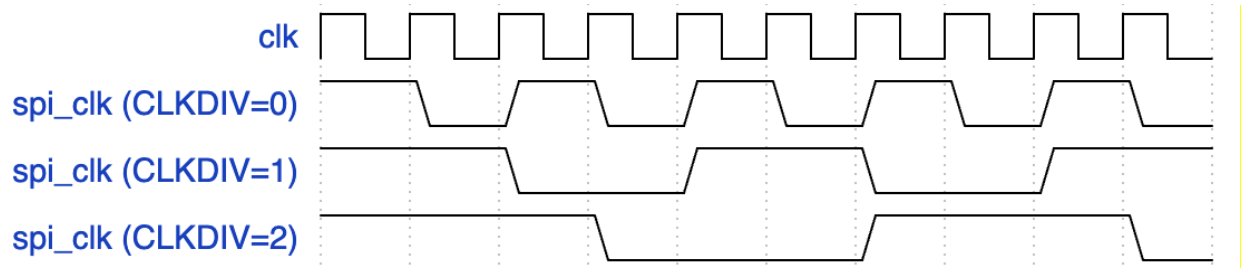
$$f_{SCK} = \frac{f_{clk}}{2*(CONFIGOPTS.CLKDIV + 1)}$$

By default, CLKDIV is set to 0, which means that the maximum frequency that can be achieved is at most half the frequency of the peripheral clock. (Fsck = Fclk/2)

We can re-arrange the equation to solve for the CLKDIV

$$CONFIGOPTS.CLKDIV = \frac{f_{clk}}{2*(f_{SCK})} - 1$$

Assuming a 400MHz target peripheral, and a SPI Clock Target of 100MHz:
CONFIGOPTS.CLKDIV = (400/(2*100)) -1 = 1



1.1.7.1.3. Signal Descriptions

The qspi block architecture inputs/outputs are described as follows:

Name	Input/Output	Description
clk_i	input	All signal timings are related to the rising edge of clk.
rst_ni	input	The reset signal is active LOW and resets the core.
cio_sck_o	output	SPI Clock
cio_sck_en_o	output	SPI Clock Enable
cio_csb_o[1:0]	output	Chip Select # (One hot, active low)
cio_csb_en_o[1:0]	output	Chip Select # Enable (One hot, active low)
cio_csb_sd_o[3:0]	output	SPI Data Output
cio_csb_sd_en_o	output	SPI Data Output Enable
cio_csb_sd_i[3:0]	input	SPI Data Input

1.1.7.1.4. Programming Guide

The operation of the SPI_HOST IP proceeds in seven general steps.

To initialize the IP:

1. Program the CONFIGOPTS multi-register with the appropriate timing and polarity settings for each csb line.
2. Set the desired interrupt parameters
3. Enable the IP

Then for each command:

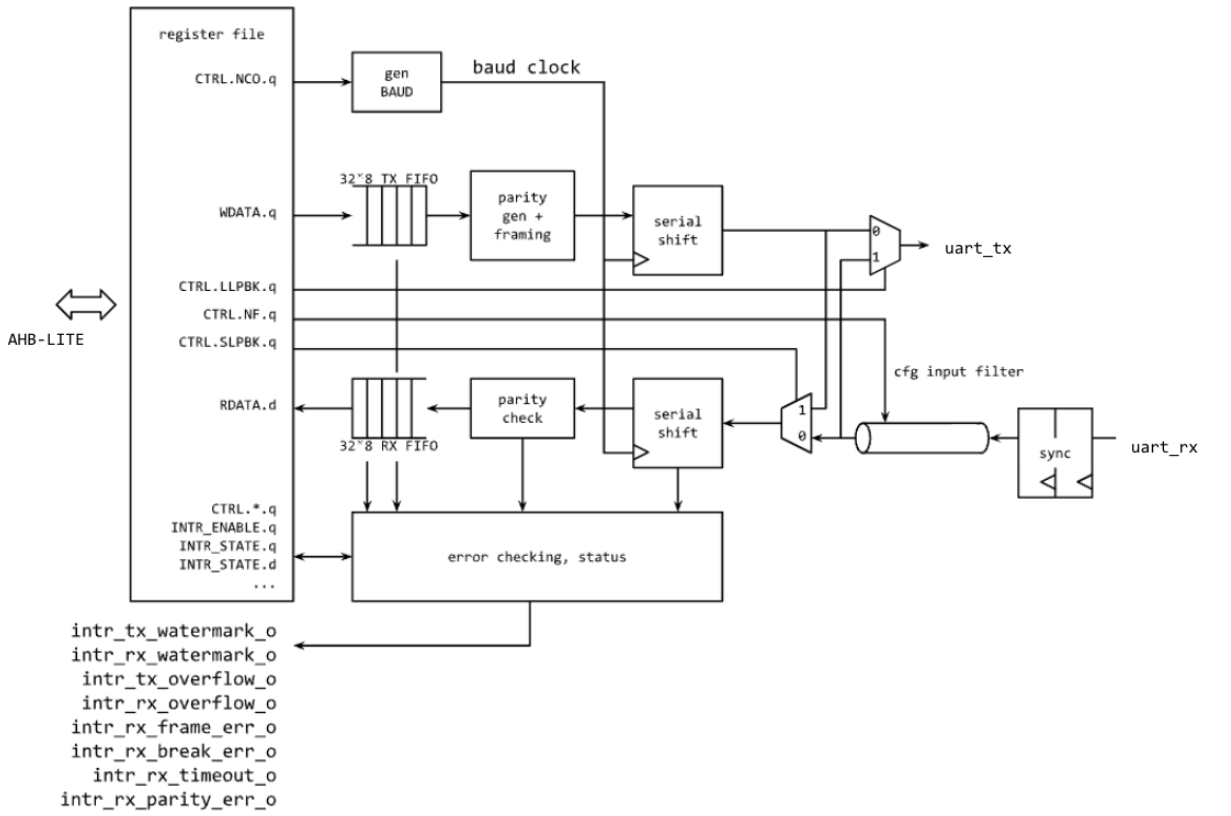
4. Load the data to be transmitted into the FIFO using the TXDATA memory window.
5. Specify the target device by programming the CSID
6. Specify the structure of the command by writing each segment into the COMMAND register
    - For multi-segment transactions, be sure to assert COMMAND.CSAAT for all but the last command segment
7. For transactions which expect to receive a reply, the data can then be read back from the RXDATA window.

Steps 4-7 are then repeated for each subsequent command.

1.1.7.2. UART

Caliptra implements a UART block that can communicate with a serial device that is accessible to FW over the AHB-Lite Interface. This is a configuration that the SOC opts-in by defining CALIPTRA_INTERNAL_UART.

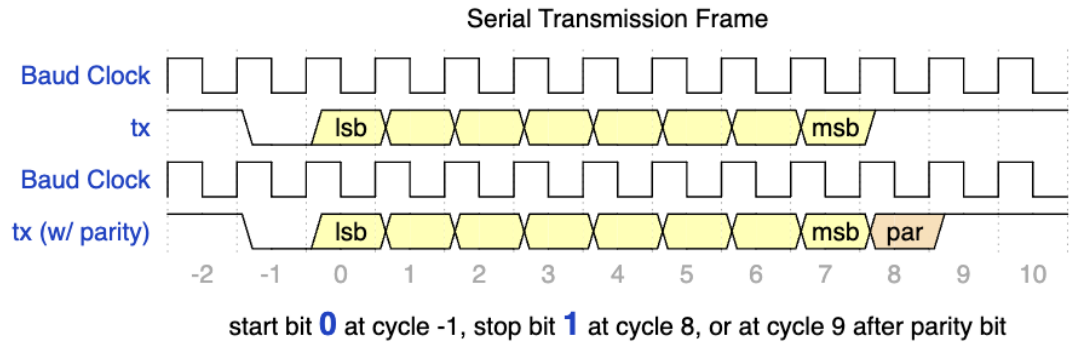
The UART block is composed of the uart (doc) implementation. The design provides support for a programmable baud rate.



1.1.7.2.1. Operation

Transactions flow through the UART block starting with a AHB-Lite write to WDATA, which triggers the transmit module to start a UART TX serial data transfer. The TX module dequeues the byte from the internal fifo and shifts it out bit by bit at the baud rate. Note that if TX is not enabled, the output will be set high and WDATA in the fifo will be queued up.

Here's a diagram of the transmit data on the serial lane, starting with the START bit, which is indicated by a high to low transition, followed by the 8 bits of data.



On the receive side, once the START bit has been detected, the data is sampled at the center of each data bit and stored into a fifo. A user can monitor the fifo status and read the data out of RDATA.

1.1.7.2.2. Configuration

The baud rate can be configured using the CTRL.NCO register field. This should be set using the equation below:

$$NCO = \frac{16 * 2^{\$bits(NCO)} x\ f_{baud}}{f_{pclk}}$$

If the desired baud rate is 115,200bps:
$bits(NCO) = 16
Fbaud = 115200
Fpclk = 100,000,000

$$NCO = \frac{2^{20} x\ 115200}{100000000} = 1208\ (0x4B7)$$

1.1.7.2.3. Signal Descriptions

The uart block architecture inputs/outputs are described as follows:

Name	Input/Output	Description
clk_i	input	All signal timings are related to the rising edge of clk.
rst_ni	input	The reset signal is active LOW and resets the core.
cio_rx_i	input	Serial Receive Bit
cio_tx_o	output	Serial Transmit Bit

1.1.7.3. I3C

TODO 0p8

1.1.8. SOC Mailbox

Please refer to the integration specification for mailbox protocol details. TODO: Fix this!

Control Register	Start Address	Desc
MBOX_LOCK	0x30020000	Mailbox lock register for mailbox access, reading 0 will set the lock
MBOX_USER	0x30020004	Stores the user that locked the mailbox
MBOX_CMD	0x30020008	Command requested for data in mailbox
MBOX_DLEN	0x3002000c	Data length for mailbox access
MBOX_DATAIN	0x30020010	Data in register, write the next data to mailbox
MBOX_DATAOUT	0x30020010	Data out register, read the next data from mailbox
MBOX_EXECUTE	0x30020018	Mailbox execute register indicates to receiver that the sender is done
MBOX_STATUS	0x3002001c	Status of the mailbox command CMD_BUSY - 2'b00 – Indicates the requested command is still in progress DATA_READY - 2'b01 – Indicates the return data is in the mailbox for requested command CMD_COMPLETE- 2'b10 – Indicates the successful completion of the requested command CMD_FAILURE- 2'b11 – Indicates the requested command failed
HW_ERROR_FATAL	0x30030000	Indicates fatal hardware error
HW_ERROR_NON_FATAL	0x30030004	Indicates non-fatal hardware error
FW_ERROR_FATAL	0x30030008	Indicates fatal firmware error
FW_ERROR_NON_FATAL	0x3003000c	Indicates non-fatal firmware error

HW_ERROR_ENC	0x30030010	Encoded error value for hardware errors
FW_ERROR_ENC	0x30030014	Encoded error value for firmware errors
BOOT_STATUS	0x30030018	Reports the boot status
FLOW_STATUS	0x3003001c	Reports the status of the firmware flows
GENERIC_INPUT_WIRES	0x30030024	Generic input wires connected to SoC interface
GENERIC_OUTPUT_WIRES	0x3003002c	Generic output wires connected to SoC interface
KEY_MANIFEST_PK_HASH	0x300302b0	
KEY_MANIFEST_PK_HASH_MASK	0x30030370	
KEY_MANIFEST_SVN	0x30030374	
BOOT_LOADER_SVN	0x30030384	
RUNTIME_SVN	0x30030388	
ANTI_ROLLBACK_DISABLE	0x3003038c	
IEEE_IDEVID_CERT_CHAIN	0x30030390	
FUSE_DONE	0x300303f0	

1.1.9. Security State

Caliptra uses the MSB of the Security State input to determine whether or not we are in “debug” mode.

When we are in debug mode:

Security State MSB is set to 0

Caliptra JTAG will be opened for uController and HW debug

Device secrets (UDS, FE, key vault, and obfuscation key) will be programmed to debug values.

If a transition to debug mode happens during ROM operation, any values computed from use of device secrets may not match expected values.

Transitions to debug mode will trigger a hardware clear of all device secrets, and an interrupt to FW to inform of the transition. FW is responsible for initiating another hardware clear of device secrets utilizing the clear secrets register, in case any derivations were in progress and stored after the transition was detected. FW may open the JTAG once all secrets have been cleared.

Debug mode values may be set by integrators in the Caliptra configuration files.

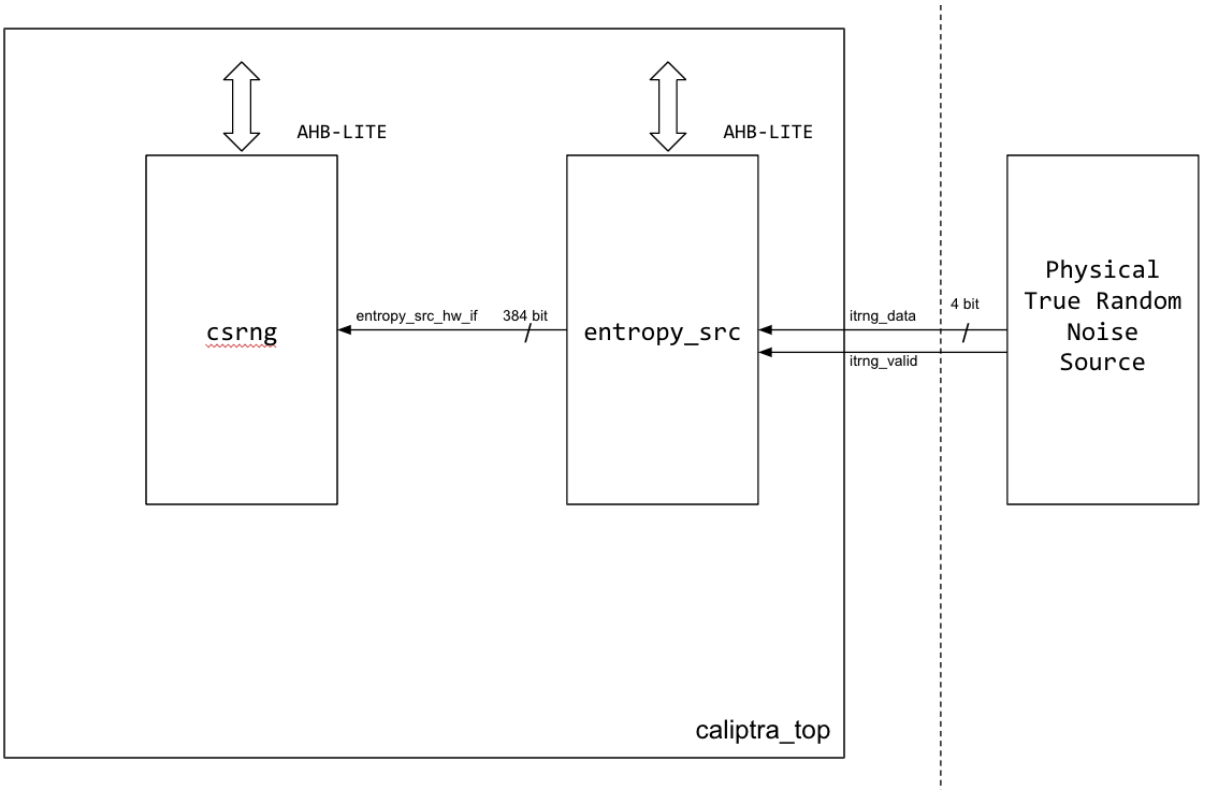
Name	Default value
Obfuscation Key Debug Value	All 0x1
UDS Debug Value	All 0x1
Field Entropy Debug Value	All 0x1
Key Vault Debug Value 0	All 0xA
Key Vault Debug Value 1	All 0x5

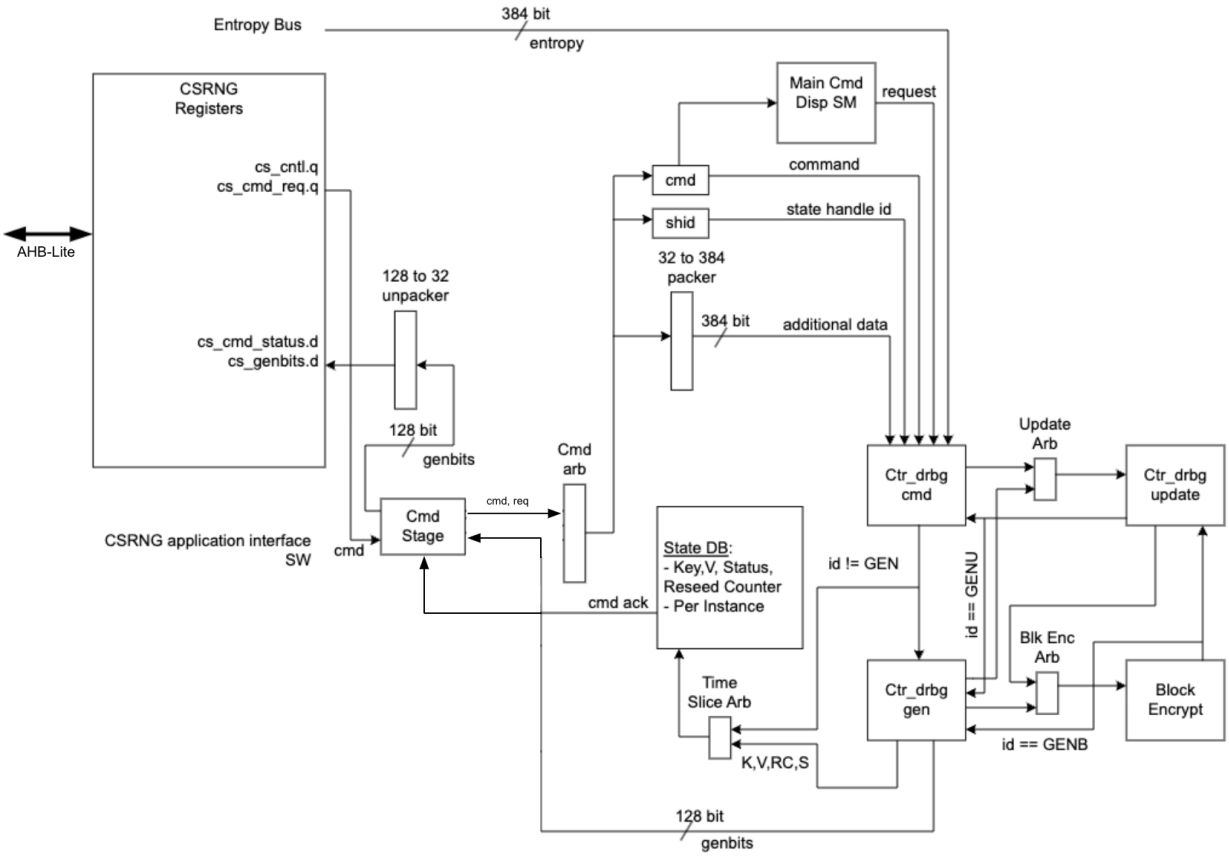
3.2. Integrated TRNG

Caliptra implements a TRNG block for local use models. TRNG is accessible to FW over AHB-Lite interface to read a random number/TRNG. This is a configuration that SOC opts-in by defining CALIPTRA_INTERNAL_TRNG.

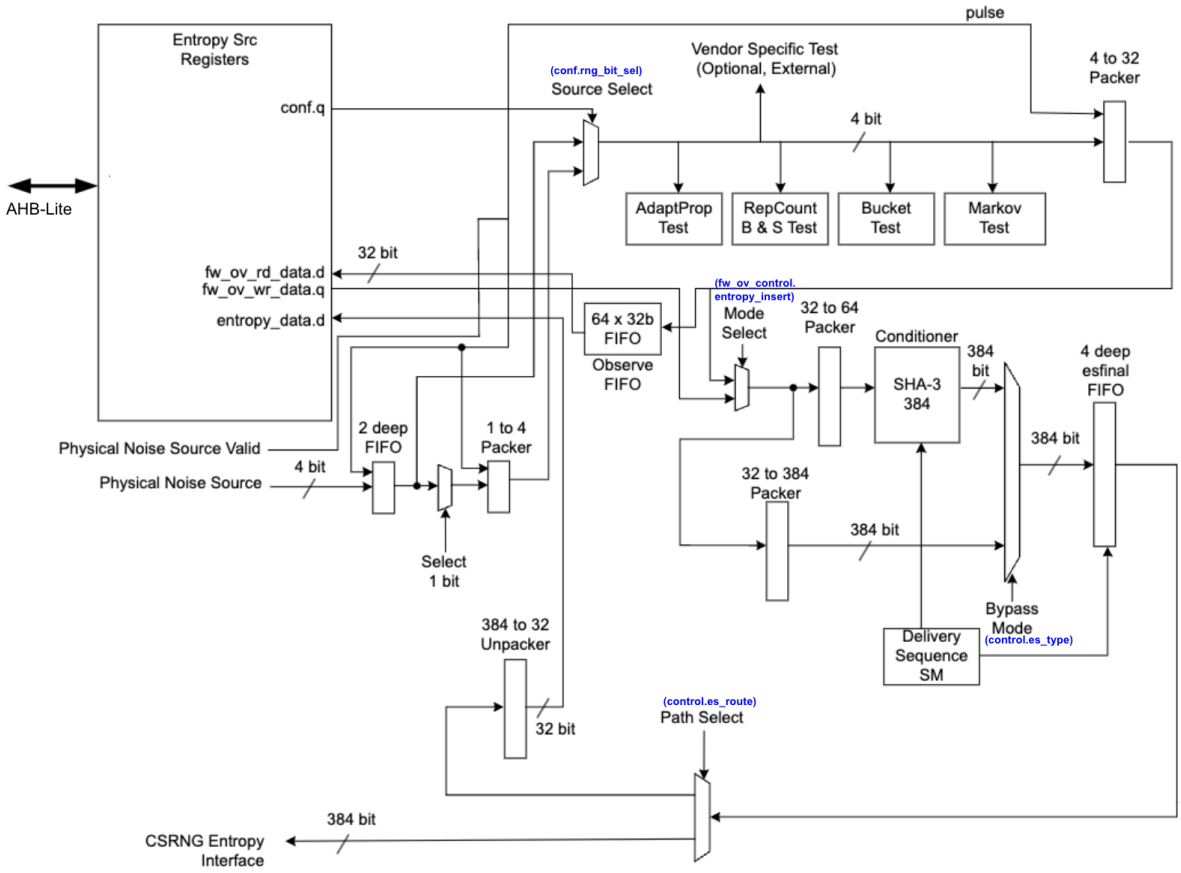
This TRNG block is a combination of entropy source (doc) and CSRNG (doc) implementations. The core code (entropy source & csrng) is reused from here but the interface to the module is changed to AHB-Lite. This design provides an interface to an external physical random noise generator (also referred to as a physical true random number generator). The PTRNG external source is a physical true random noise source. A noise source and its relation to an entropy source are defined by SP 800-90B.

The block is instantiated based on a design parameter chosen at integration time. This is to provide options for SOC to reuse an existing TRNG that they may already have to build an optimized SOC design. For the optimized scenarios, SOC needs to follow External-TRNG REQ HW API flow.


AHB-LITE
AHB-LITE
csrng
entropy_src_hw_if
384 bit
entropy_src
itrng_data
4 bit
itrng_valid
Physical True Random Noise Source
caliptra_top



CSRNG Block Diagram



Entropy Source Block Diagram

3.2.1. Operation

Requests for entropy bits start with command requests over the AHB-Lite interface to the csrng CMD_REQ register.

Command Request Header:

Bits	Name	Description
3:0	acmd	Application Command: Selects one of five operations to perform. The commands supported are instantiate, reseed, generate, update, and uninstantiate.
7:4	clen	Command Length: Number of 32-bit words that can optionally be appended to the command. A value of zero will only transfer the command header. A value of 4'hc will transfer the header plus an additional twelve 32-bit words of data.

11:8	flag0	Command Flag0: flag0 is associated with current command. Setting this field to True (4'h6) will enable flag0 to be enabled. Note that flag0 is used for the instantiate and reseed commands only, for all other commands its value is ignored.
24:12	glen	Generate Length: Only defined for the generate command, this field is the total number of cryptographic entropy blocks requested. Each unit represents 128 bits of entropy returned. A value of 8 would return a total of 1024 bits. The maximum size supported is 4096.

First an instantiate command is requested over the SW Application Interface to initialize an instance in the CSRNG module. Depending on the flag0 and clen fields in the command header, a request to the entropy_src module over the entropy interface will be sent to seed the csrng. This can take a few milliseconds if the seed entropy is not immediately available.

Example Instantiation:

acmd = 0x1 (Instantiate)
clen/flag0 = The seed behavior is determined by the table below
glen = Not used

flag0	clen	Description
F	0	Only entropy source seed is used.
F	1-12	Entropy source seed is xor'ed with provided additional data.
T	0	Seed of zero is used (no entropy source seed used).
T	1-12	Only provided additional data will be used as seed.

Next a generate command will be used to request generation of cryptographic entropy bits. The glen field defines how many 128b words are to be returned to the application interface. Once the generated bits are ready, they can be read out via the GENBITS register. Note that this register must be read out glen * 4 times for each request made.

Example Generate Command:

acmd = 0x3 (Generate)
clen = 0
flag0 = false (4'h9)
glen = 4 (4 *128 = 512b)

This will require 16 reads from GENBITS to read out all of the generated entropy.

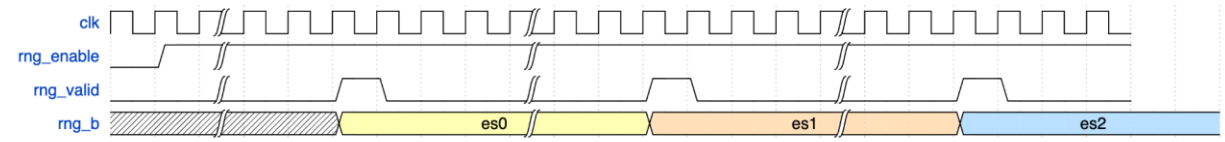
3.2.2. Configuration

The HW Application Interfaces are not supported. Only the SW Application Interface should be used for this design.

3.2.3. Physical True Random Noise Source Signal Descriptions

These are the top level signals defined in caliptra_top

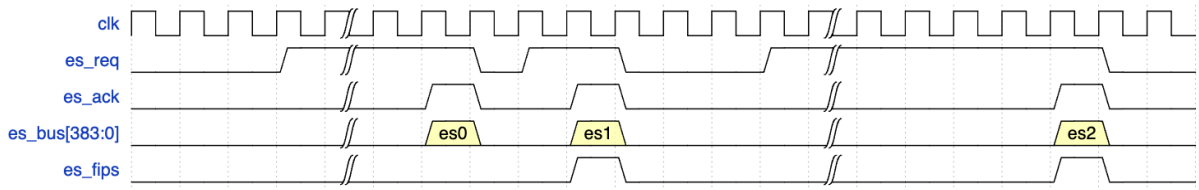
Name	Input/Output	Description
itrng_data	input	Physical True Random Noise Source Data
itrng_valid	input	Valid is asserted high for one cycle when data is valid. The expected valid output rate is about 50KHz.



3.2.4. Entropy Source Signal Descriptions

Name	Input/Output	Description
clk_i	input	All signal timings are related to the rising edge of clk.
rst_ni	input	The reset signal is active LOW and resets the core.
entropy_src_rng_req	output	Request from the entropy_src module to the physical true random noise source to start generating data. TBD if this is required.
entropy_src_rng_rsp	input	Contains both the Internal TRNG data / valid. Valid is asserted high for one cycle when data is valid.
entropy_src_hw_if_i	input	Downstream block request for entropy bits

entropy_src_hw_if_o	output	384 bits of entropy data. Valid when es_ack is asserted high.
cs_aes_halt_i	input	Response from csrng that all requests to AES block are halted.
cs_aes_halt_o	output	Request to csrng to halt requests to the AES block for power leveling purposes.



-

3.2.5. CSRNG Signal Descriptions

Name	Input/Output	Description
clk_i	input	All signal timings are related to the rising edge of clk.
rst_ni	input	The reset signal is active LOW and resets the core.
otp_en_csrng_sw_app_read_i	input	Enable firmware to access the ctr_drbg internal state and genbits through registers
lc_hw_debug_en_i	input	Lifecycle that will select which diversification value is used for xoring with he seed from entropy_src
entropy_src_hw_if_i	input	384 bits of entropy data. Valid when es_ack is asserted high.
entropy_src_hw_if_o	output	Downstream block request for entropy bits
cs_aes_halt_i	input	Request from entropy_src to halt requests to the AES block for power leveling purposes.
cs_aes_halt_o	output	Response to entropy_src that all requests to AES block are halted.

-

3.2.6. Programming Guide

The CSRNG may only be enabled if entropy_src is enabled. Once disabled, CSRNG may only be re-enabled after entropy_src has been disabled and re-enabled.

-

3.3. External-TRNG REQ HW API

For SOCs that choose to not instantiate Caliptra's embedded TRNG (as noted in the above section), Caliptra provides a TRNQ REQ HW API.

1. Caliptra asserts TRNG_REQ wire (this may be because Caliptra's internal HW or FW made the request for a TRNG)
2. SOC will write the TRNG architectural registers
3. SOC will write a done bit in the TRNG architectural registers
4. Caliptra desserts TRNG_REQ

Reason to have a separate interface (than using SOC mailbox) is to ensure that this request is not intercepted by any SOC FW agents [which communicate with SOC mailbox]. It is a requirement that this TRNG HW API is always handled by a SOC HW gasket logic (and not some SOC ROM/FW code) for FIPS compliance.

3.4. SOC-SHA accelerator HW API

Caliptra provides a SHA accelerator HW API for SOC and Caliptra Internal FW to use. It is atomic in nature that only one of them can use it at the same time.

SHA Accelerator registers:

Offset	Identifier	Name
0x000	LOCK	SHA Accelerator Lock
0x004	USER	SHA Accelerator User
0x008	MODE	SHA Accelerator Mode

0x00C	START_ADDRESS	SHA Accelerator Data Start Address
0x010	DLEN	SHA Accelerator Data Length
0x014	DATAIN	SHA Accelerator Data In
0x018	EXECUTE	SHA Accelerator Execute
0x01C	STATUS	SHA Accelerator Status
0x020	DIGEST[16]	SHA Accelerator Digest
0x060	CONTROL	SHA Accelerator Control

Theory of operation:

- A user of the HW API will first lock the accelerator by reading the LOCK register. A read 0 will set/grant the lock of the resource. A write of ‘1 will clear the lock
- USER register captures the user of the SHA accelerator - Michael Norris: are we taking PAUSER or its just that SOC vs Caliptra FW?
- MODE register is written to set the SHA execution mode.
    - SHA Supports both SHA384 and SHA512 modes of operation.
    - SHA supports **streaming** mode => SHA is computed on a stream of incoming data to DATAIN register. When the EXECUTE register is set, it tells the accelerator that streaming is completed and it can publish the result into DIGEST and this output is valid, when the VALID bit is set in the STATUS register.
    - SHA also supports **Mailbox** mode - SHA is computed on LENGTH (DLEN) bytes of data stored in the mailbox from START_ADDRESS when EXECUTE register is set. The result will be published in the DIGEST register and this output is valid, when the VALID bit is set in the STATUS register.
        - Please note that the default behavior assumes that data in mailbox is little endian; But when set to 0, data from mailbox will be swizzled from little to big endian at the byte level. When set to 1, data from the mailbox will be loaded into SHA as-is.
    - Please see the register definition for the encodings
- For Streaming Function, indicates that the initiator is done streaming. For the Mailbox SHA Function, indicates that the SHA can begin execution.

- SHA engine also provides a ‘zeroize’ function through its CONTROL register to clear any of the SHA internal state. This can be used for cases where the user wants to not have previous state present for debug/security reasons.

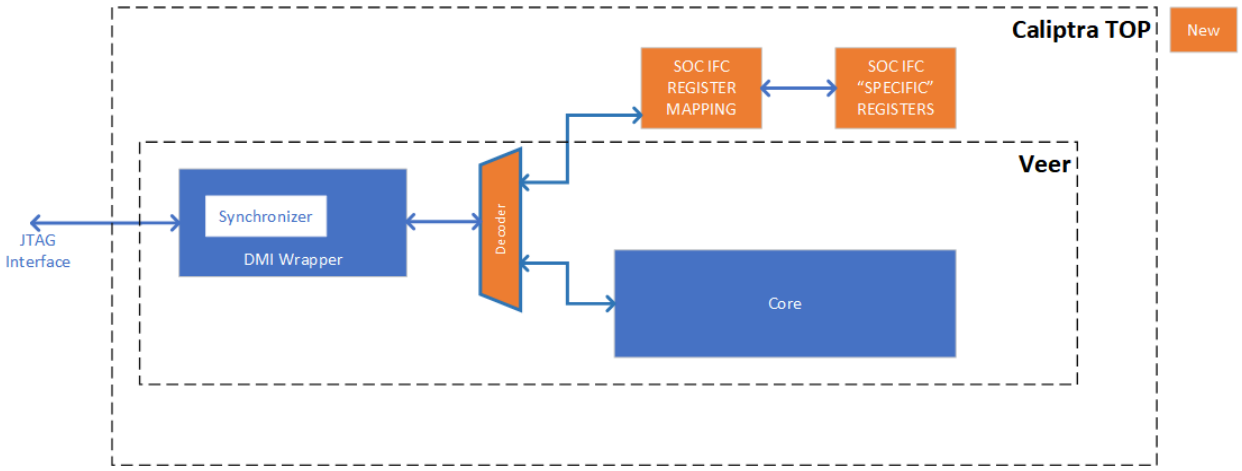
3.5. JTAG Implementation

Please see “Debug” section of the Caliptra ROT specification for debug flows.

Figure below shows the JTAG implementation within Caliptra boundary. The output of existing DMI wrapper is used to find the non-Core (Caliptra uncore) aperture to route the JTAG commands.

Caliptra’s JTAG/TAP should be implemented as a TAP EP. JTAG is open if the debug mode is set at the time of caliptra reset deassertion.

Note: If the debug security state switches to debug mode anytime, the security assets/keys are still flushed even though JTAG is not open.



●

4. Crypto High-Level Architecture

Special mention: Thanks to Emre Karabulut from NCSU for his contributions towards Side-Channel Analysis of the cryptos.

The architecture of Caliptra crypto subsystem includes the following components:

- Symmetric Cryptographic Primitives
  - De-Obfuscation engine
  - SHA512/384 (based on NIST FIPS 180-4 [2])
  - SHA256 (based on NIST FIPS 180-4 [2])
  - HMAC384 (based on NIST FIPS 198-1 [5] and RFC 4868 [6])
- Public-key Cryptography
  - NIST Secp384r1 Deterministic Digital Signature Algorithm (based on FIPS-186-4 [11] and RFC 6979 [7])
- Key Vault
  - Key Slots
  - Key Slot Management

The high-level architecture of Caliptra crypto subsystem is shown as follows:

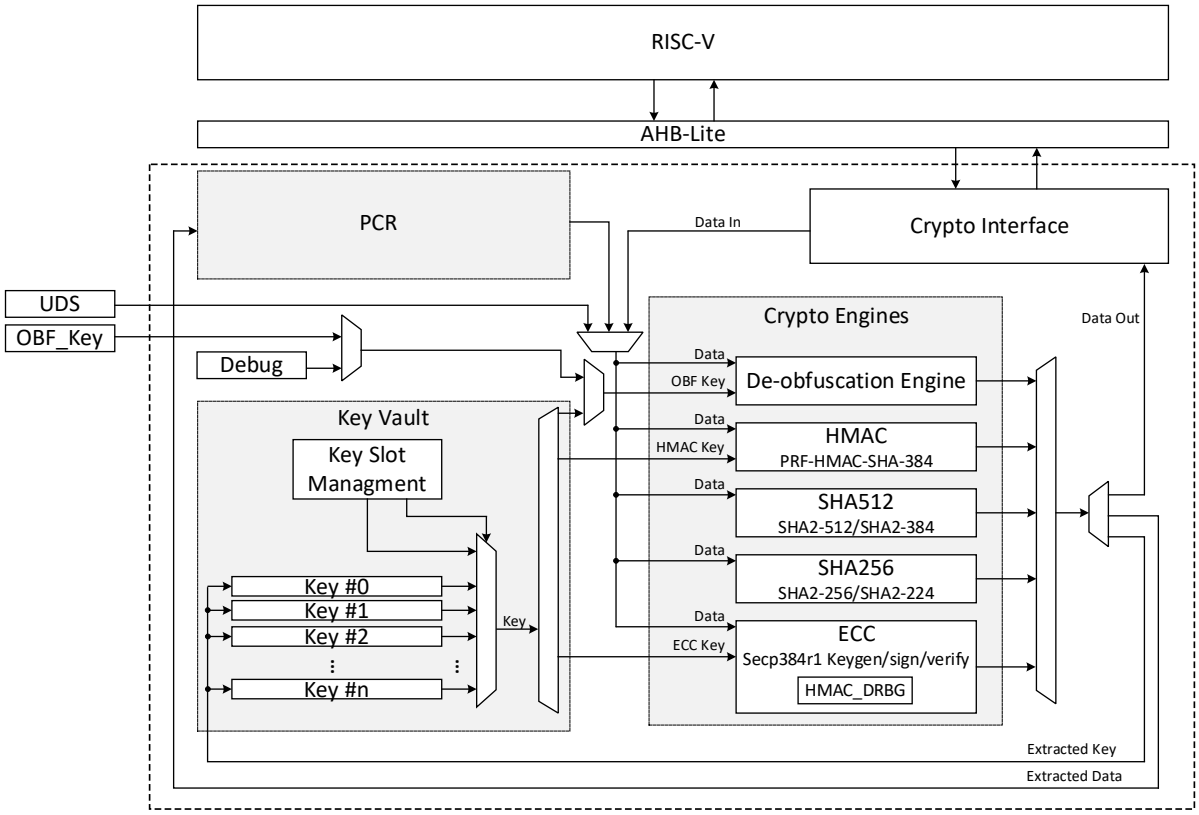


Figure 4- Caliptra crypto-subsystem

4.2. SHA512/SHA384

SHA-512 is a function of cryptographic hash algorithm SHA-2. The hardware implementation is based on Secworks/sha512 [1]. This implementation complies with the functionality in NIST FIPS 180-4 [2]. The implementation supports the SHA-512 variants SHA-512/224, SHA-512/256, SHA-384 and SHA-512.
SHA-512 algorithm is described as follows:

- The message is padded by the host and broken into 1024-bit chunks,
- For each chunk:
  - The message is fed to the sha512 core,
  - The core should be triggered by the host,
  - The sha512 core status is changed to ready after hash processing
- The result digest can be read after feeding all message chunks.

4.2.1. Operation

4.2.1.1. Padding

The message should be padded before feeding to the hash core. The input message is taken, and some padding bits are appended to it in order to get it to the desired length. The bits that are used for padding are simply '0' bits with a leading '1' (100000…000). The appended length of the message (before pre-processing), in bits, is a 128-bit big-endian integer.
The total size should be equal to 128 bits short of a multiple of 1024 since the goal is to have the formatted message size as a multiple of 1024 bits (N x 1024).

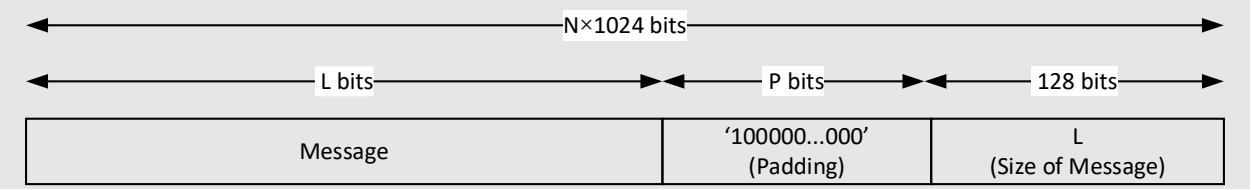


Figure 5- SHA512 input formatting

4.2.1.2. Hashing

The sha512 core performs 80 iterative operations to process the hash value of the given message. The algorithm works in a way where it processes each block of 1024 bits from the message using the result from the previous block. For the first block, the initial vectors (IV) are used for starting off the chain processing of each 1024-bit block.

4.2.2. FSM

The SHA512 architecture has the finite-state machine as follows:

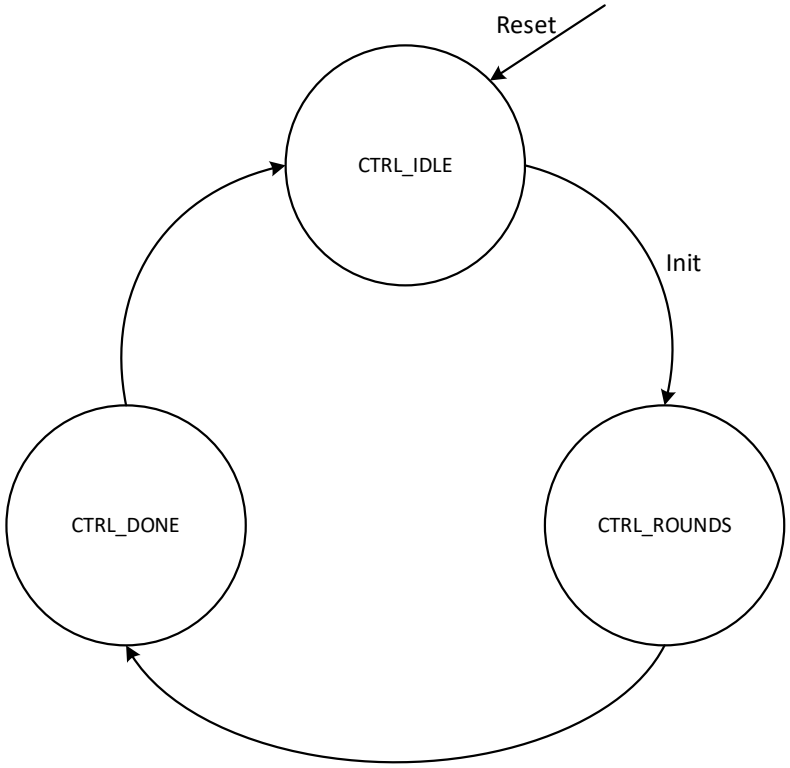


Figure 6- SHA512 FSM

4.2.3. Signal Descriptions

The SHA512 architecture inputs/outputs are described as follows:

Name	Input/Output	Description
clk	input	All signal timings are related to the rising edge of clk.
reset_n	input	The reset signal is active LOW and resets the core. This is the only active LOW signal.
init	input	The core is initialized and processes the first block of message.
next	input	The core processes the rest of message blocks using the result from the previous blocks.
mode[1:0]	input	Indicates the hash type of the function. This can be:

		- SHA512/224 - SHA512/256 - SHA384 - SHA512
zeroize	input	The core clears all internal registers to avoid any SCA information leakage.
block[1023:0]	input	The input padded block of message.
ready	output	When HIGH, the signal indicates the core is ready.
digest[511:0]	output	The hashed value of the given block.
digest_valid	output	When HIGH, the signal indicates is the result is ready.

Table 10- SHA512 signal descriptions

4.2.4. Address Map

The SHA512 address map is shown as follows:

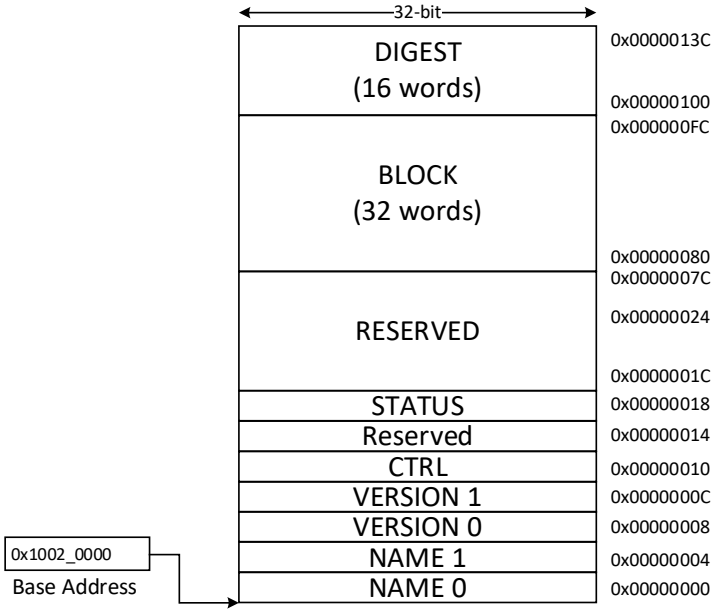


Figure 7- SHA512 Address Map

4.2.4.1. NAME

Read-only register consists of the name of component.

4.2.4.2. VERSION

Read-only register consists of the version of component.

4.2.4.3. CTRL

The control register consists of the following flags:

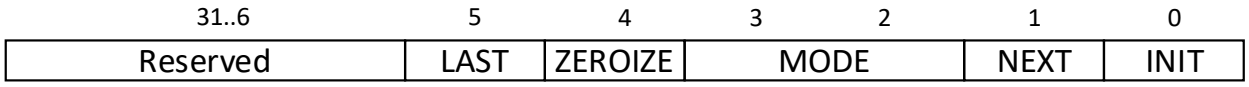


Figure 8- CTRL register

- INIT

Trigs the SHA512 core to start the processing for the first message block.

- NEXT

Trigs the SHA512 core to start the processing for the remining message block.

- MODE

Indicates the SHA512 core to set the type of hashing algorithm. This can be:

MODE	Hashing type
00	SHA512/224
01	SHA512/256
10	SHA384
11	SHA512

Table 11- SHA512 hashing mode

- ZEROIZE

Trigs the SHA512 core to clear all registers to avoid any information leakage.

- LAST

Indicates last iteration for hash extend function. The result of this INIT or NEXT cycle will be written back to PCR.

4.2.4.4. STATUS

The read-only status register consists of the following flags:

31..2	1	0
Reserved	VALID	READY

Figure 9- STATUS register

- READY

Indicates if the core is ready to process the block.

- VALID

Indicates if the hash value is computed and value stored in DIGEST register is valid.

4.2.5. Pseudocode

The following pseudocode demonstrates how SHA512 interface can be implemented.


```
Input:
    block[0:n-1]     //n blocks of 1024-bit message (32 32-bit words)
Output:
    hash_value       //512-bit (16 32-bit words)

//wait for SHA512 engine to be ready (STATUS flag should be 2'b01 or 2'b11)
read_data = 0;
while(read_data == 0){
   read_data = read(ADDR_STATUS);
};

for (i=0, i<n, i++){
    //feed 32 32-bit words as a block of message
    write(ADDR_BLOCK, block[i]);

    //trig the sha512 engine to perform hashing
    if(i==0)
        write(ADDR_CTRL, {28'h0, MODE, 0, INIT});
    else
        write(ADDR_CTRL, {28'h0, MODE, NEXT, 0});

    //wait for sha512 engine to be ready and valid (STATUS flag should be 2'b11)
    read_data = 0;
    while(read_data == 0){
        read_data = read(ADDR_STATUS);
    };
};

//read the outputs
hash_value = read(ADDR_DIGEST)
write(ADDR_CTRL, {27'h0, ZEROIZE, 2'b0, 0, 0});

return hash_value
```

Figure 10- SHA512 pseudocode

4.2.6. SCA Countermeasure

We do not propose any countermeasure to protect the hash functions. Inherently, hash functions do not work like other crypto engines. It targets integrity without requiring a secret key. Hence, the attacker can target only messages. Also, the attacker cannot build a CPA or DPA platform on the hash function because the same message ideally gives the same side-channel behavior.
If the attacker works on the multi-message mechanism, the attacker then needs to work with single trace attacks which are very unlikely in ASIC/FPGA implementations. Also, our hash implementation is a noisy platform. As a result, we do not propose any SCA implementation countermeasure on the hash functions.

4.2.7. Performance

The SHA512 core performance is reported considering two different architectures: (i) pure hardware architecture, and (ii) hardware/software architecture.

4.2.7.1. Pure Hardware Architecture

In this architecture, the SHA512 interface and controller are implemented in hardware. The performance specification of the SHA512 architecture is reported as follows:

Operation	Data bus	Cycle count [CCs]	Freq [MHz]	Time [us]	Throughput [op/s]
Data_In transmission	32-bit	33	400	0.08	-
Process		87		0.22	-
Data_Out transmission		16		0.04	-
Single block		136		0.34	2,941,176
Double block		224		0.56	1,785,714
1kB message		840		2.10	476,190
128kB message		17,632		44.08	22,686

4.2.7.2. Hardware/Software Architecture

In this architecture, the SHA512 interface and controller are implemented in RISC-V core. The performance specification of the SHA512 architecture is reported as follows:

Operation	Data bus	Cycle count [CCs]	Freq [MHz]	Time [us]	Throughput [op/s]
Data_In transmission	32-bit	990	400	2.48	-
Process		139		0.35	-
Data_Out transmission		387		0.97	-
Single block		1,516		3.79	263,852
Double block		2,506		6.27	159,617
1kB message		9,436		23.59	42,391

128kB message		1,015,276		2,538.19	394

Table 12-SHA512 Performance using RISC-V core and crypto hardware accelerator

4.2.7.3. Pure Software Architecture

In this architecture, the SHA512 algorithm is implemented fully in software. The implementation is derived from OpenSSL's SHA512 implementation [3]. The performance numbers for this architecture are as follows:

Data Size	Cycle Count
1 KB	147,002
4 KB	532,615
8 KB	1,046,727
12 KB	1,560,839
128 KB	16,470,055

Table 13-SHA512 performance over RISC-V core

4.3. SHA-256

SHA-256 is a function of cryptographic hash algorithm SHA-2. The hardware implementation is based on Secworks/sha256 [1]. The implementation supports the two variants SHA-256/224 and SHA-256.
SHA-256 algorithm is described as follows:

- The message is padded by the host and broken into 512-bit chunks,
- For each chunk:
  - The message is fed to the sha256 core,
  - The core should be triggered by the host,
  - The sha256 core status is changed to ready after hash processing
- The result digest can be read after feeding all message chunks.

4.3.1. Operation

4.3.1.1. Padding

The message should be padded before feeding to the hash core. The input message is taken, and some padding bits are appended to it in order to get it to the desired length. The bits that are used for padding are simply '0' bits with a leading '1' (100000…000). The appended length of the message (before pre-processing), in bits, is a 64-bit big-endian integer.
The total size should be equal to 64 bits, short of a multiple of 512 since the goal is to have the formatted message size as a multiple of 512 bits (N x 512).

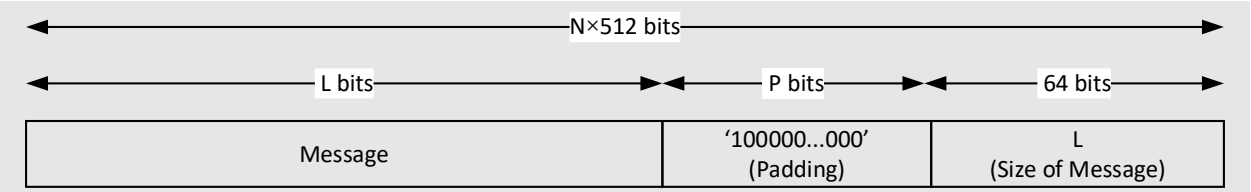


Figure 11- SHA256 input formatting

4.3.1.2. Hashing

The sha256 core performs 64 iterative operations to process the hash value of the given message. The algorithm works in a way where it processes each block of 512 bits from the message using the result from the previous block. For the first block, the initial vectors (IV) are used for starting off the chain processing of each 512-bit block.

4.3.2. FSM

The SHA256 architecture has the finite-state machine as follows:

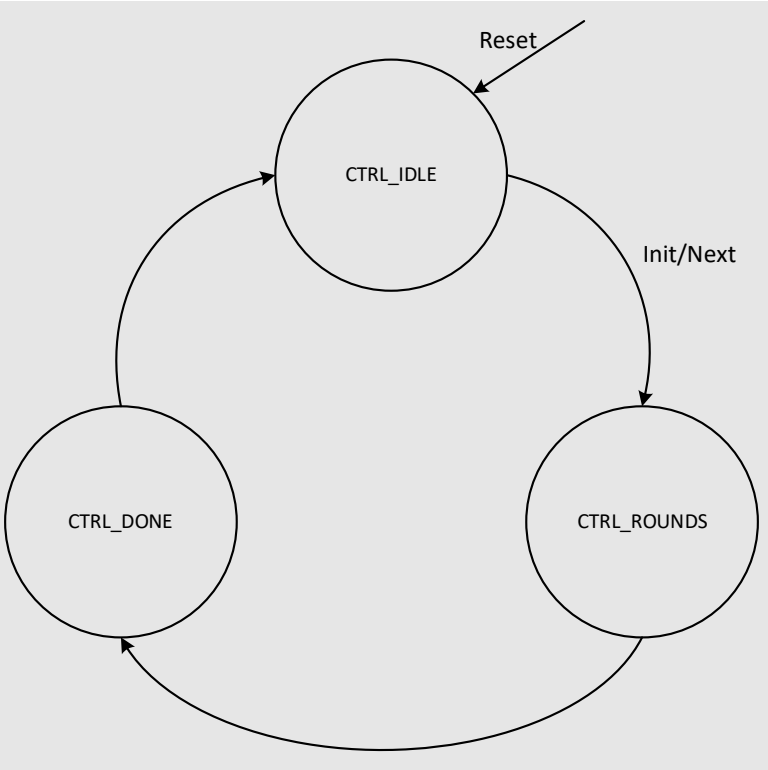


Figure 12- SHA256 FSM

4.3.3. Signal Descriptions

The SHA256 architecture inputs/outputs are described as follows:

Name	Input/Output	Description
clk	input	All signal timings are related to the rising edge of clk.
reset_n	input	The reset signal is active LOW and resets the core. This is the only active LOW signal.
init	input	The core is initialized and processes the first block of message.
next	input	The core processes the rest of message blocks using the result from the previous blocks.
mode	input	Indicates the hash type of the function. This can be: - SHA256/224 - SHA256
zeroize	input	The core clears all internal registers to avoid any SCA information leakage.
block[511:0]	input	The input padded block of message.
ready	output	When HIGH, the signal indicates the core is ready.
digest[255:0]	output	The hashed value of the given block.
digest_valid	output	When HIGH, the signal indicates is the result is ready.

4.3.4. Address Map

The SHA256 address map is shown as follows:

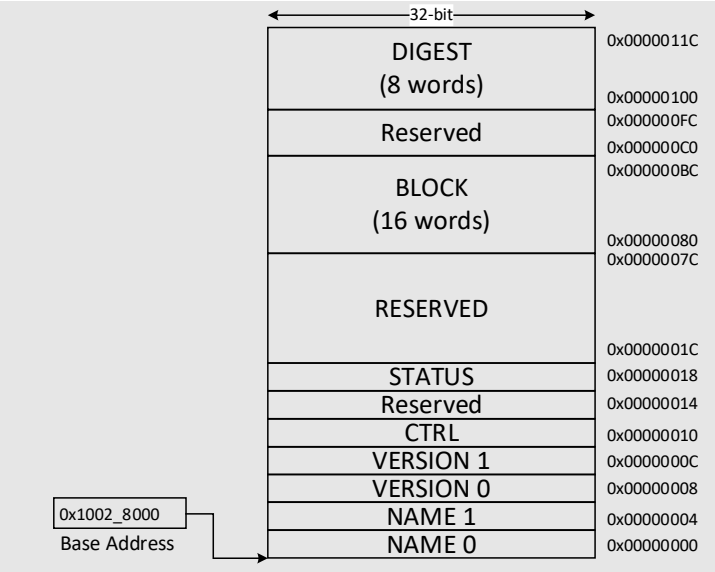


Figure 13- SHA256 Address Map

4.3.4.1. NAME

Read-only register consists of the name of component.

4.3.4.2. VERSION

Read-only register consists of the version of component.

4.3.4.3. CTRL

The control register consists of the following flags:

31..4	3	2	1	0
Reserved	ZEROIZE	MODE	NEXT	INIT

Figure 14- SHA256 CTRL register

- INIT

Trigs the SHA256 core to start the processing for the first message block.

- NEXT

Trigs the SHA256 core to start the processing for the remining message block.

- MODE

Indicates the SHA256 core to set the type of hashing algorithm. This can be:

MODE	Hashing type
0	SHA256/224
1	SHA256

- ZEROIZE

Trigs the SHA256 core to clear all registers to avoid any information leakage.

4.3.4.4. STATUS

The read-only status register consists of the following flags:

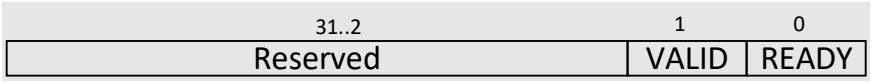


Figure 15- SHA256 STATUS register

- READY

Indicates if the core is ready to process the block.

- VALID

Indicates if the hash value is computed and value stored in DIGEST register is valid.

4.3.5. Pseudocode

The following pseudocode demonstrates how SHA256 interface can be implemented.


```
Input:
    block[0:n-1]     //n blocks of 512-bit message (16 32-bit words)
Output:
    hash_value       //256-bit (8 32-bit words)


//wait for SHA256 engine to be ready (STATUS flag should be 2'b01 or 2'b11)
read_data = 0;
while(read_data == 0){
   read_data = read(ADDR_STATUS);
};

for (i=0, i<n, i++){
    //feed 16 32-bit words as a block of message
    write(ADDR_BLOCK, block[i]);

    //trig the SHA256 engine to perform hashing
    if(i==0)
        write(ADDR_CTRL, {29'h0, MODE, 0, INIT});
    else
        write(ADDR_CTRL, {29'h0, MODE, NEXT, 0});

    //wait for SHA256 engine to be ready and valid (STATUS flag should be 2'b11)
    read_data = 0;
    while(read_data == 0){
        read_data = read(ADDR_STATUS);
    };
};

//read the outputs
hash_value = read(ADDR_DIGEST);
write(ADDR_CTRL, {28'h0, ZEROIZE, 0, 0, 0});


return hash_value
```

Figure 16- SHA256 pseudocode

4.3.6. SCA Countermeasure

Please see SCA countermeasure Section for the previous hash implementation.

4.3.7. Performance

The SHA256 core performance is reported considering two different architectures: (i) pure hardware architecture, and (ii) hardware/software architecture.

4.3.7.1. Pure Hardware Architecture

TODO 0p8

In this architecture, the SHA256 interface and controller are implemented in hardware. The performance specification of the SHA256 architecture is reported as follows:

Operation	Data bus	Cycle count [CCs]	Freq [MHz]	Time [us]	Throughput [op/s]
Data_In transmission	32-bit		400		-
Process					-
Data_Out transmission					-
Single block					
Double block					
1kB message					
128kB message					

4.3.7.2. Hardware/Software Architecture

TODO 0p8

In this architecture, the SHA256 interface and controller are implemented in RISC-V core. The performance specification of the SHA256 architecture is reported as follows:

Operation	Data bus	Cycle count [CCs]	Freq [MHz]	Time [us]	Throughput [op/s]
Data_In transmission	32-bit		400		-
Process					-
Data_Out transmission					-
Single block					
Double block					
1kB message					
128kB message					

4.4. HMAC384

Hash-based message authentication code (HMAC) is a cryptographic authentication technique that uses a hash function and a secret key. HMAC involves a cryptographic hash function and a secret cryptographic key. This implementation supports HMAC-SHA-384-192 as specified in NIST FIPS 198-1 [5]. The implementation is compatible with the HMAC-SHA-384-192 auth/integ function defined in RFC 4868 [6].
This version of HMAC implemented uses SHA-384 as the hash function, accepts a 384-bit key, and generates a 384-bit tag.
The implementation also supports PRF-HMAC-SHA-384. The PRF-HMAC-SHA-384 algorithm is identical to HMAC-SHA-384-192, except that variable-length keys are permitted, and the truncation step is NOT performed.

The HMAC algorithm is described as follows:

- The key is fed to HMAC core to be padded.
- The message is broken into 1024-bit chunks by the host,
- For each chunk:
  - The message is fed to the HMAC core,
  - The HMAC core should be triggered by the host,
  - The HMAC core status is changed to ready after hash processing
- The result digest can be read after feeding all message chunks.

4.4.1. Operation

4.4.1.1. Padding

The message should be padded before feeding to the HMAC core. Internally, the i_padded key will be concatenated with the message. The input message is taken, and some padding bits are appended to it in order to get it to the desired length. The bits that are used for padding are simply ‘0’ bits with a leading ‘1’ (100000…000).
The total size should be equal to 128 bits, short of a multiple of 1024 since the goal is to have the formatted message size as a multiple of 1024 bits (N x 1024).

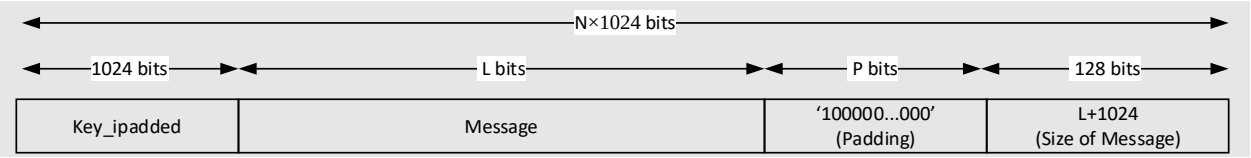


Figure 23- HMAC input formatting

Following figures show some examples of input formatting for different message lengths.

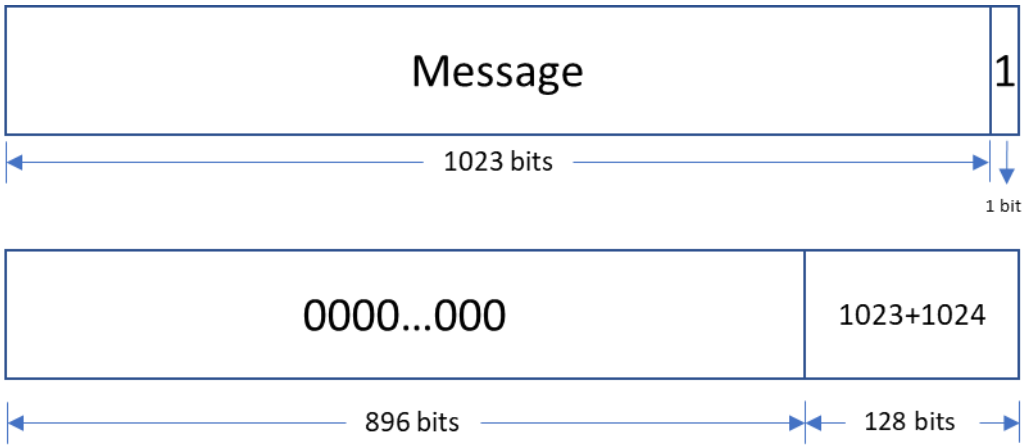


Figure – Message length of 1023 bits

When message is 1023 bits long, padding is given in the next block along with message size.

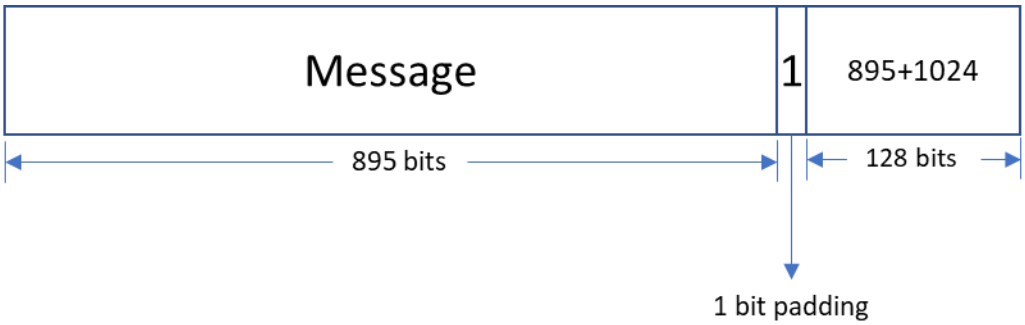


Figure – 1 bit padding

When message size is 895 bits, a padding of '1' is also considered valid followed by the message size.

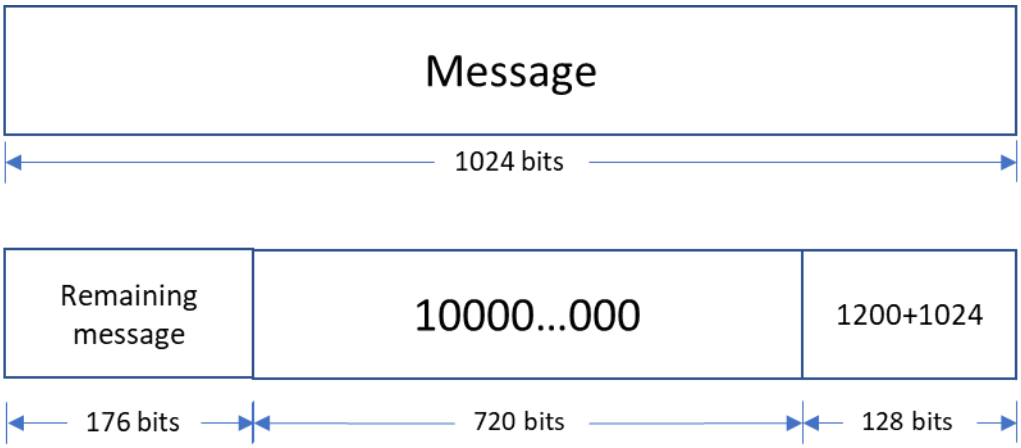


Figure – Multi block message

Messages of length > 1024 bits are broken down into N 1024-bit blocks. Last block will contain padding and size of message.

4.4.1.2. Hashing

The HMAC core performs sha2-384 function to process the hash value of the given message. The algorithm works in a way where it processes each block of 1024 bits from the message using the result from the previous block.

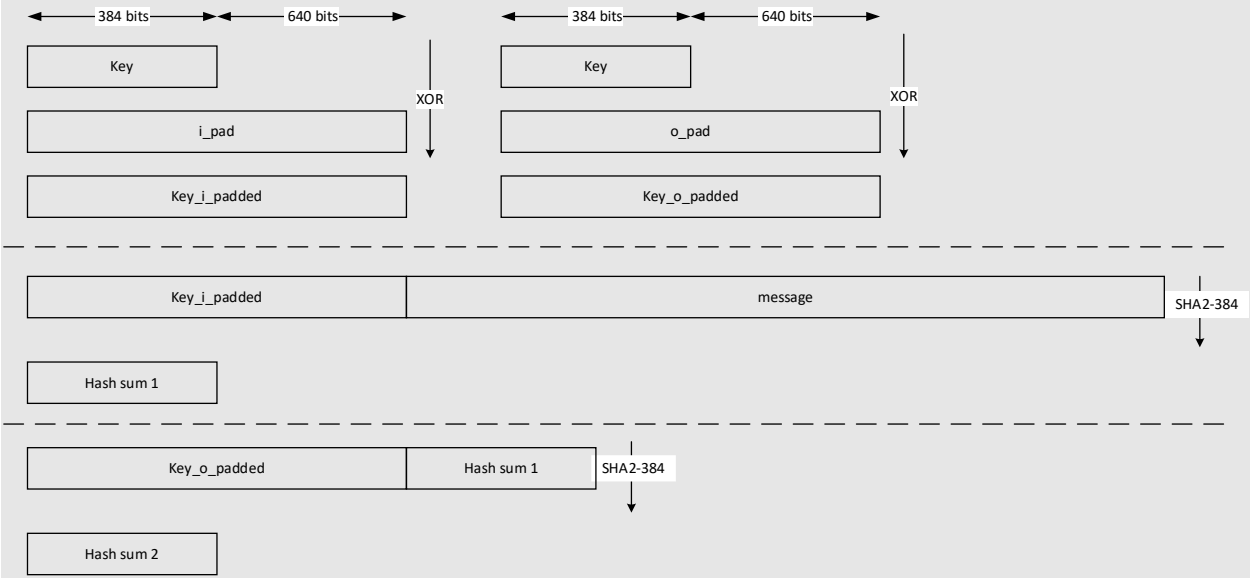


Figure 24- HMAC-SHA-384-192 data flow

4.4.2. FSM

The HMAC architecture has the finite-state machine as follows:

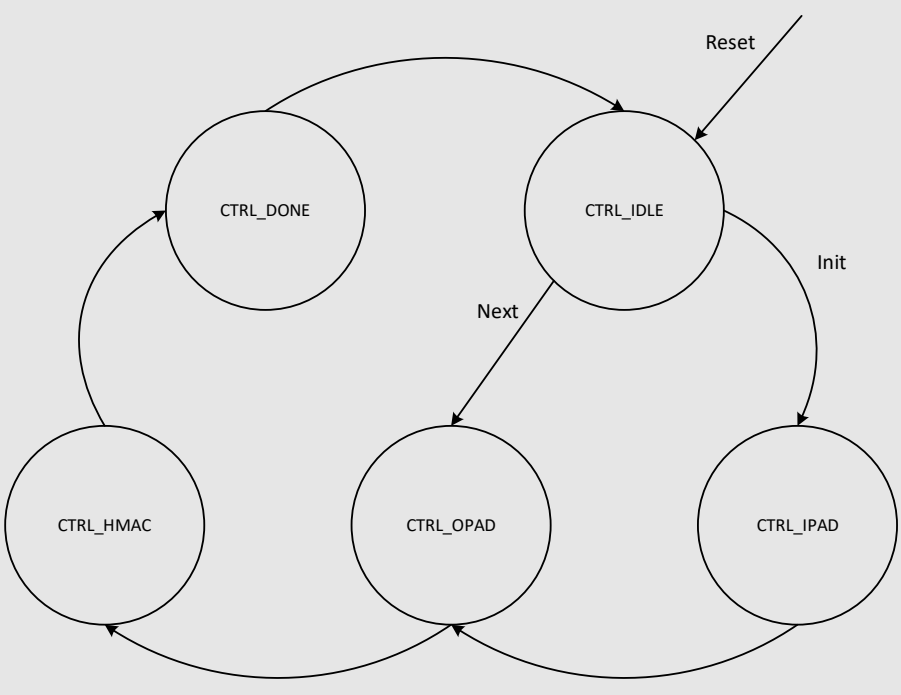


Figure 25- HMAC FSM

4.4.3. Signal Descriptions

The HMAC architecture inputs/outputs are described as follows:

Name	Input/Output	Description
clk	input	All signal timings are related to the rising edge of clk.
reset_n	input	The reset signal is active LOW and resets the core. This is the only active LOW signal.
init	input	The core is initialized and processes the key and the first block of message.
next	input	The core processes the rest of message blocks using the result from the previous blocks.
zeroize	input	The core clears all internal registers to avoid any SCA information leakage.
key[383:0]	input	The input key.
block[1023:0]	input	The input padded block of message.

LFSR_seed[159:0]	Input	The input to seed PRNG to enable masking countermeasure for SCA protection.
ready	output	When HIGH, the signal indicates the core is ready.
tag[383:0]	output	The hmac value of the given key/block. For PRF-HMAC-SHA-384, 384-bit tag is required, while for HMAC-SHA-384-192, the host is responsible to read 192 bits from MSB.
tag_valid	output	When HIGH, the signal indicates is the result is ready.

4.4.4. Address Map

The HMAC address map is shown as follows:

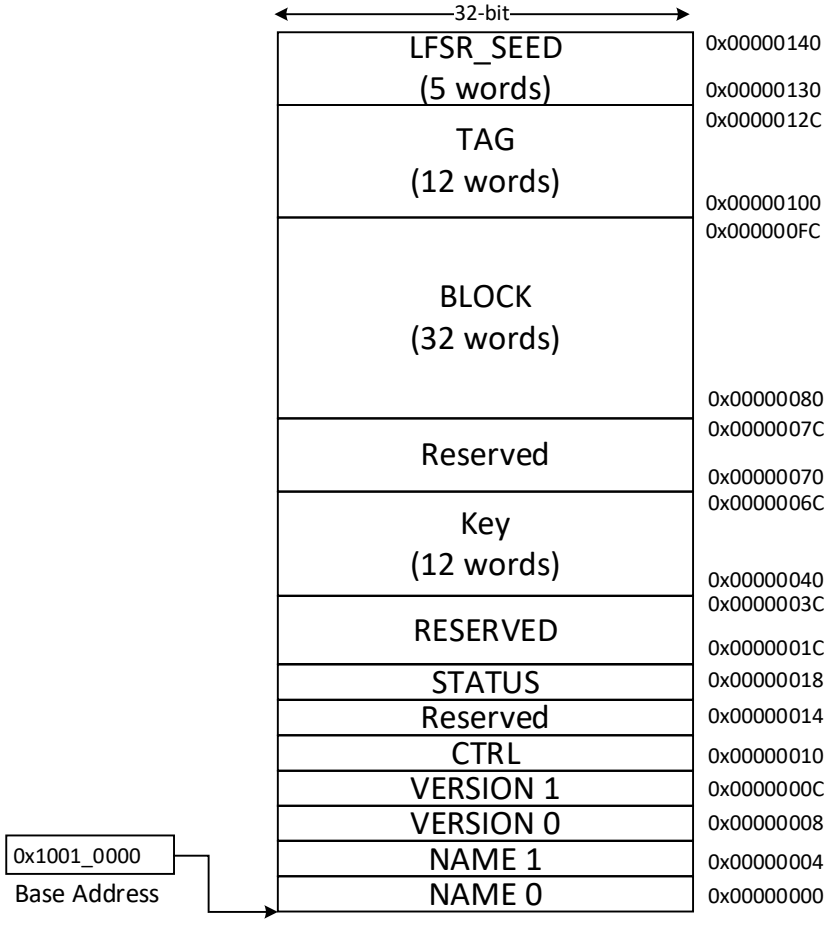


Figure 26- HMAC Address Map

4.4.4.1. NAME

Read-only register consists of the name of component.

4.4.4.2. VERSION

Read-only register consists of the version of component.

4.4.4.3. CTRL

The control register consists of the following flags:

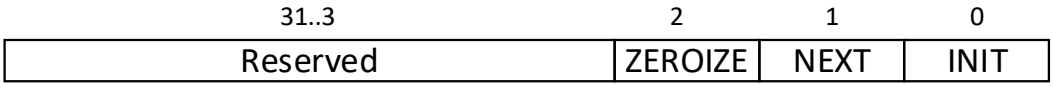


Figure 27- HMAC CTRL register

- INIT

Trigs the HMAC core to start the initialization and processing the key.

- NEXT

Trigs the HMAC core to start the processing for a message block.

- ZEROIZE

Trigs the SHA256 core to clear all registers to avoid any information leakage.

4.4.4.4. STATUS

The read-only status register consists of the following flags:

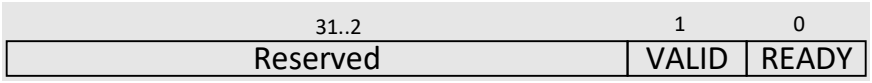


Figure 28- HMAC STATUS register

- READY

Indicates if the core is ready to process the block.

- VALID

Indicates if the hmac value is computed and value stored in TAG registers is valid.

4.4.5. Pseudocode

The following pseudocode demonstrates how HMAC interface can be implemented.

```
Input:
    key             //384-bit key (12 32-bit words)
    LFSR_SEED       //160-bit LSFR_seed (5 32-bit words)
    block[0:n-1]    //n blocks of 1024-bit message (32 32-bit words)
Output:
    tag             //384-bit (12 32-bit words)

//wait for HMAC engine to be ready (STATUS flag should be 2'b01 or 2'b11)
read_data = 0;
while(read_data == 0){
   read_data = read(ADDR_STATUS);
};

//config HMAC engine with the given key
write(ADDR_KEY, key);

//config LFSR with the given seed
write(ADDR_LFSR_SEED, LFSR_SEED);

for (i=0, i<n, i++){
    //feed 32 32-bit words as a block of message
    write(ADDR_BLOCK, block[i]);

    //trig HMAC engine to perform hashing
    if(i==0)
        write(ADDR_CTRL, {30'h0, 0,  INIT});
    else
        write(ADDR_CTRL, {30'h0, NEXT, 0});

    //wait for HMAC engine to be ready and valid (STATUS flag should be 2'b11)
    read_data = 0;
    while(read_data == 0){
        read_data = read(ADDR_STATUS);
    };
};

//read the outputs
tag = read(ADDR_TAG);
write(ADDR_CTRL, {2'h0, ZEROIZE, 0, 0});

Return tag;
```

Figure 29- HMAC pseudocode

4.4.6. SCA Countermeasure

In an attack model, an attacker can form hypotheses about the secret key value and compute the corresponding output values by using the Hamming Distance model as an appropriate leakage model. Particularly, having the knowledge of details of the implementation for open-source

architecture, this attack would be aided. The attacker can capture the power consumption traces to verify her hypotheses, by partitioning the acquisitions or using Pearson's correlation coefficient.

The making countermeasure is embedded into HMAC architecture to use random values to conceal intermediate variables in the implementation of the algorithm, thereby making the side-channel leakage independent of the secret intermediate variables.
The embedded countermeasures are based on "Differential Power Analysis of HMAC Based on SHA-2, and Countermeasures" by McEvoy et. al.. To provide the required random values for masking intermediate values, a lightweight 74-bit LFSR is implemented. Based on "Spin Me Right Round Rotational Symmetry for FPGA-specific AES" by Wegener et. al., LFSR is sufficient for masking statistical randomness.
Each round of SHA512 execution needs 6,432 random bits, while one HMAC operation needs at least 4 rounds of SHA512 operations. However, the proposed architecture requires only 160-bit LFSR seed and provides first-order DPA attack protection at the cost of 10% latency overhead with negligible hardware resource overhead.

4.4.7. Performance

The HMAC core performance is reported considering two different architectures: (i) pure hardware architecture, and (ii) hardware/software architecture.

4.4.7.1. Pure Hardware Architecture

In this architecture, the HMAC interface and controller are implemented in hardware. The performance specification of the HMAC architecture is reported as follows:

Operation	Data bus	Cycle count [CCs]	Freq [MHz]	Time [us]	Throughput [op/s]
Data_In transmission	32-bit	44	400	0.11	-
Process		254		0.635	-
Data_Out transmission		12		0.03	-
Single block		310		0.775	1,290,322
Double block		513		1.282	780,031
1kB message		1,731		4.327	231,107
128kB message		207,979		519.947	1,923

4.4.7.2. Hardware/Software Architecture

In this architecture, the HMAC interface and controller are implemented in RISC-V core. The performance specification of the HMAC architecture is reported as follows:

Operation	Data bus	Cycle count [CCs]	Freq [MHz]	Time [us]	Throughput [op/s]
Data_In transmission	32-bit	1389	400	3.473	-
Process		253		0.633	-
Data_Out transmission		290		0.725	-
Single block		1932		4.83	207,039
Double block		3166		7.915	136,342
1kB message		10,570		26.425	37,842
128kB message		1,264,314		3,160.785	316

4.5. HMAC_DRBG

Hash-based message authentication code (HMAC) deterministic random bit generator (DRBG) is a cryptographic random bit generator that uses a HMAC function. HMAC_DRBG involves a cryptographic HMAC function and a seed. This architecture is designed as specified in section 10.1.2. of NIST SP 800-90A [12]. For ECC signing operation, the implementation is compatible with section 3.1. of RFC 6979 [7].
This version of HMAC_DRBG implemented uses HMAC384 as the HMAC function, accepts a 384-bit seed, and generates a 384-bit random value.

The HMAC algorithm is described as follows:

- The seed is fed to HMAC_DRBG core by the host.
- For each 384-bit random value
    - The core should be triggered by the host,
    - The HMAC_DRBG core status is changed to ready after hmac processing
    - The result digest can be read.

4.5.1. Operation

HMAC_DRBG uses a loop of HMAC(K, V) to generate the random bits. In this algorithm, two constant values of K_init and V_init are used as follows:

```
1.   Set V_init = 0x01 0x01 0x01 ... 0x01  (V has 384-bit)
2.   Set K_init = 0x00 0x00 0x00 ... 0x00  (K has 384-bit)
3.   K_tmp = HMAC(K_init, V_init || 0x00 || entropy || nonce)
4.   V_tmp = HMAC(K_tmp,  V_init)
5.   K_new = HMAC(K_tmp,  V_tmp  || 0x01 || entropy || nonce)
6.   V_new = HMAC(K_new,  V_tmp)
7.   Set T = []
8.   T = T || HMAC(K_new, V_new)
9.   Return T if T is within the [1,q-1] range, otherwise:
10.  K_new = HMAC(K_new,  V_new || 0x00)
11.  V_new = HMAC(K_new,  V_new)
12.  Jump to 8
```

For ECC KEYGEN operation,HMAC_DRBG is used to generate privkey as follows:

- `Privkey = HMAC_DRBG(seed, nonce)`

For ECC SIGNING operation, HMAC_DRBG is used to generate k as follows:

- `K = HMAC_DRBG(privkey, hashed_msg)`

4.5.2. Signal Descriptions

The HMAC_DRBG architecture inputs/outputs are described as follows:

Name	Input/Output	Description
clk	input	All signal timings are related to the rising edge of clk.
reset_n	input	The reset signal is active LOW and resets the core. This is the only active LOW signal.
init	input	The core is initialized with the given seed and generates 384-bit random value.
next	input	The core generates a new 384-bit random value using the result from the previous run.
zeroize	input	The core clears all internal registers to avoid any SCA information leakage.
entropy [383:0]	input	The input entropy.
nonce [383:0]	input	The input nonce.
LFSR_seed [147 :0]	input	The input to seed PRNG to enable masking countermeasure for SCA protection.
ready	output	When HIGH, the signal indicates the core is ready.
drbg [383:0]	output	The hmac_drbg value of the given inputs.
valid	output	When HIGH, the signal indicates is the result is ready.

4.5.3. Address Map

The HMAC_DRBG in embedded into ECC architecture, since there is no address map to access it from FW.

4.5.4. SCA Countermeasure

Please see SCA section for the previous HMAC implementation.

4.6. ECC

The ECC unit includes the ECDSA (Elliptic Curve Digital Signature Algorithm) engine offering a variant of the cryptographically secure Digital Signature Algorithm (DSA) which uses elliptic curve (ECC). A digital signature is an authentication method used where a public key pair and a digital certificate are used as a signature to verify the identity of a recipient or sender of information.
The hardware implementation supports deterministic ECDSA, 384 Bits (Prime Field), also known as NIST-Secp384r1, described in RFC6979.
Secp384r1 parameters are as follows:

```
E : y^2 = x^3 +Ax + B
p  =  0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffeffffffff0000
        0000000000ffffffff
A  =  0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffeffffffff0000
        0000000000fffffffc
B  =  0xb3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5013875ac656398d8a2e
        d19d2a85c8edd3ec2aef
Gx =  0xaa87ca22be8b05378eb1c71ef320ad746e1d3b628ba79b9859f741e082542a385502f25dbf55
        296c3a545e3872760ab7
Gy =  0x3617de4a96262c6f5d9e98bf9292dc29f8f41dbd289a147ce9da3113b5f0b8c00a60b1ce1d7e
        819d7a431d7c90ea0e5f
G = (Gx, Gy)
n  =  0xffffffffffffffffffffffffffffffffffffffffffffffffc7634d81f4372ddf581a0db248b0
        a77aecec196accc52973
h = 0x1
```

4.6.1. Operation

The ECDSA consists of three operations.

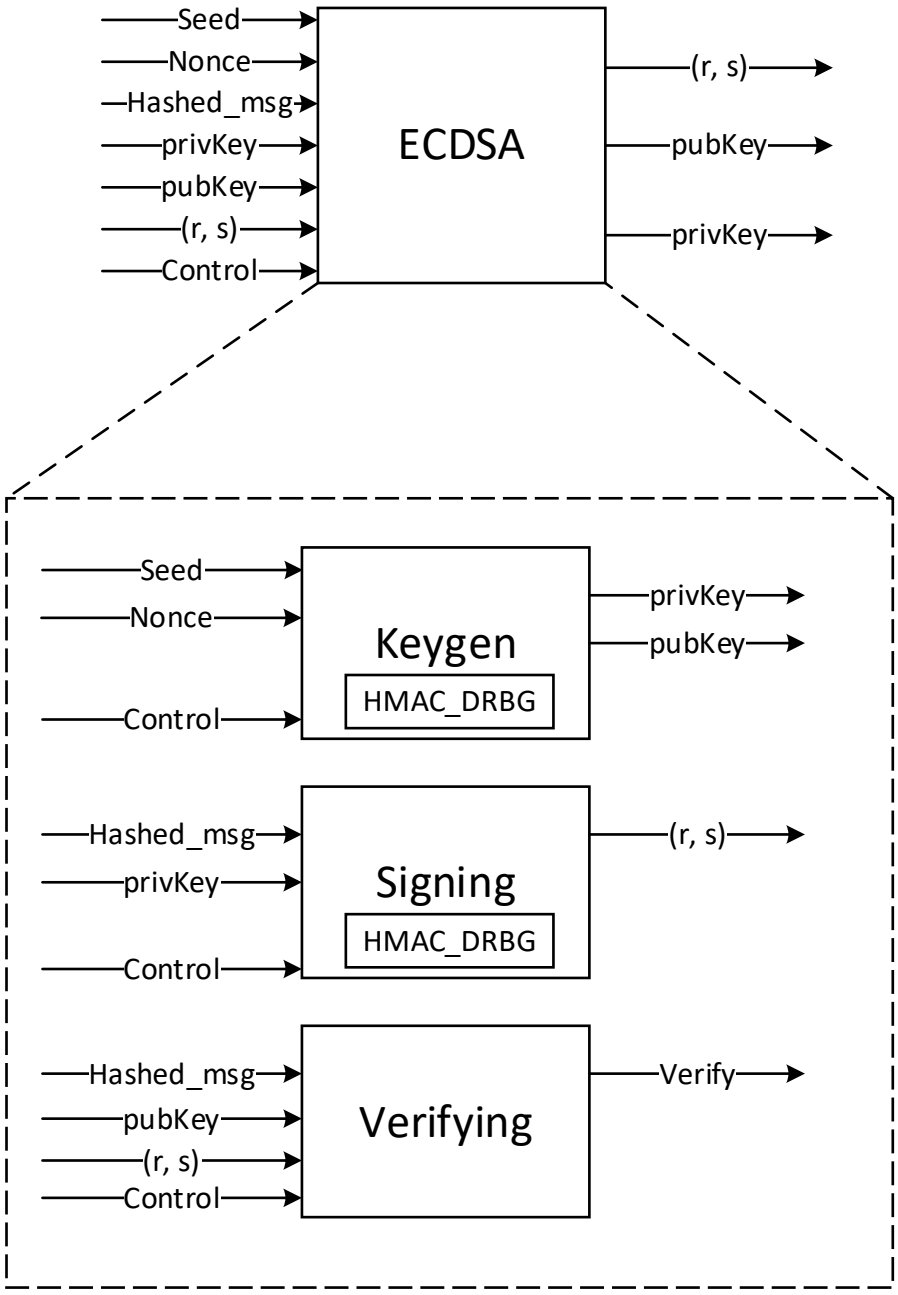


Figure -ECDSA Operations

4.6.1.1. Key Generation

In the deterministic key generation, the paired key of (privKey, pubKey) is generated by KeyGen(seed, nonce), taking a deterministic seed and nonce. Keygen algorithm is as follows:

- Compute privKey = HMAC_DRBG(seed, nonce) to generate a random integer in the interval [1, n-1] where n is the group order of Secp384 curve.

- Generate pubKey(x,y) as a point on ECC calculated by pubKey=privKey × G, while G is the generator point over the curve).

4.6.1.2. Signing

In signing algorithm, a signature (r, s) is generated by Sign(privKey, h), taking an privKey and hash of message m, h = hash(m) using a cryptographic hash function SHA-384. Signing algorithm includes:

- Generate a random number k in the range [1..n-1], while k = HMAC_DRBG(privKey, h)
- Calculate the random point R = k × G.
- Take r = Rx mod n, where Rx is x coordinate of R=(Rx, Ry).
- Calculate the signature proof: $s = k^{-1} \times (h + r \times privKey) \bmod n$
- Return the signature (r, s), each in the range [1..n-1].

4.6.1.3. Verifying

The signature (r, s) can be verified by Verify(pubKey ,h ,r, s) considering the public key pubKey and hash of message m, h=hash(m) using the same cryptographic hash function SHA-384. The output is r' value of verifying a signature. The ECDSA verify algorithm includes:

- Calculate $s1 = s^{-1} \bmod n$
- Compute $R' = (h \times s1) \times G + (r \times s1) \times pubKey$
- Take r' = R'x mod n, while R'x is x coordinate of R'=(R'x, R'y),
- Verify the signature by comparing whether r' == r

4.6.2. Architecture

ECC top-level architecture is shown in the following figure:

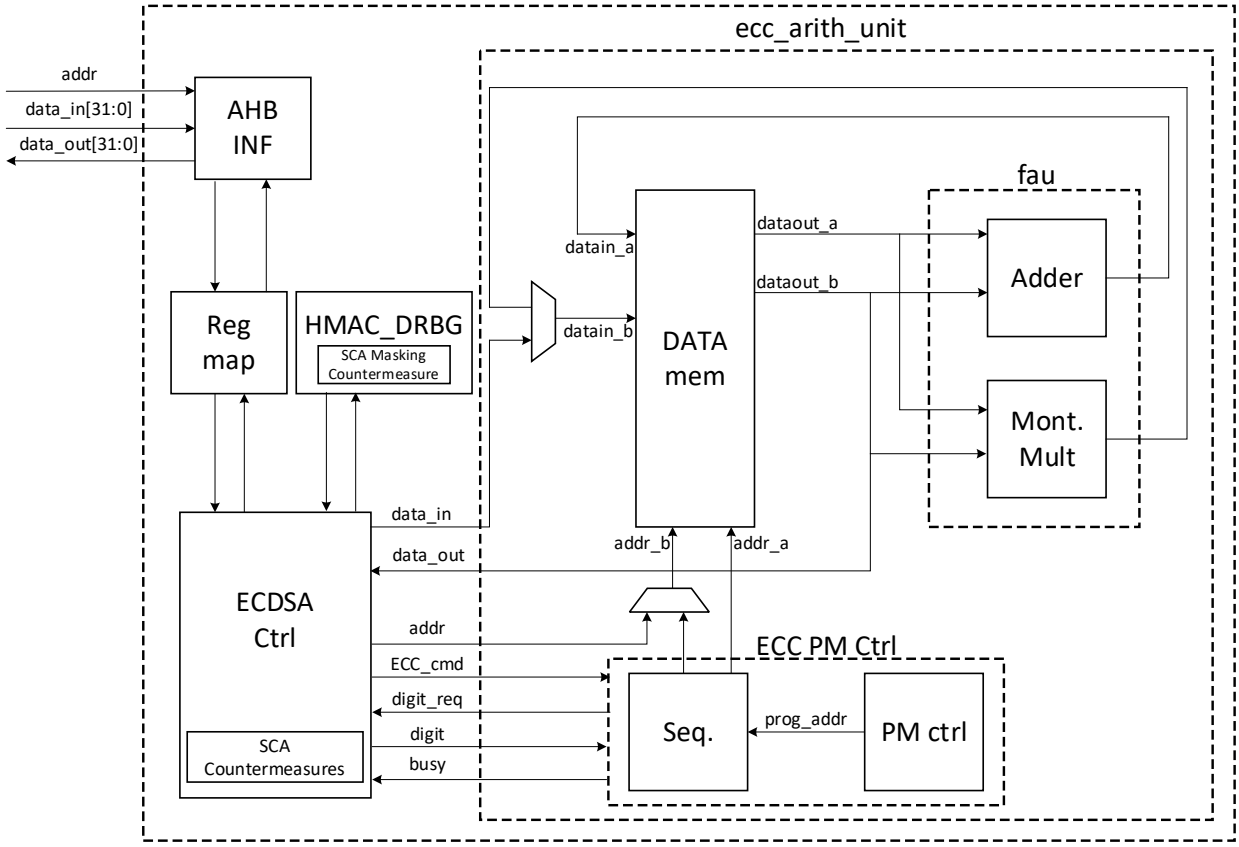


Figure -ECDSA Architecture

4.6.3. Signal Descriptions

The ECDSA architecture inputs/outputs are described as follows:

Name	Input/Output	Description
clk	input	All signal timings are related to the rising edge of clk.
reset_n	input	The reset signal is active LOW and resets the core. This is the only active LOW signal.
ctrl[1:0]	input	Indicates the AES type of the function. This can be: – 0b00: No Operation – 0b01: KeyGen – 0b10: Signing – 0b11: Verifying

zeroize	input	The core clears all internal registers to avoid any SCA information leakage.
seed [383:0]	input	The deterministic seed for HMAC_DRBG in Keygen operation.
nonce [383:0]	input	The deterministic nonce for HMAC_DRBG in Keygen operation.
privKey_in[383:0]	input	The input private key used in the signing operation.
pubKey_in[1:0][383:0]	input	The input public key(x,y) used in the verifying operation.
hashed_msg[383:0]	input	The hash of message using SHA384.
ready	output	When HIGH, the signal indicates the core is ready.
privKey_out[383:0]	output	The generated private key in keygen operation.
pubKey_out[1:0][383:0]	output	The generated public key(x,y) in keygen operation.
r[383:0]	output	The signature value of the given priveKey/message.
s[383:0]	output	The signature value of the given priveKey/message.
r'[383:0]	Output	The signature verification result.
valid	output	When HIGH, the signal indicates the result is ready.

4.6.4. Address Map

The ECDSA address map is shown as follows:

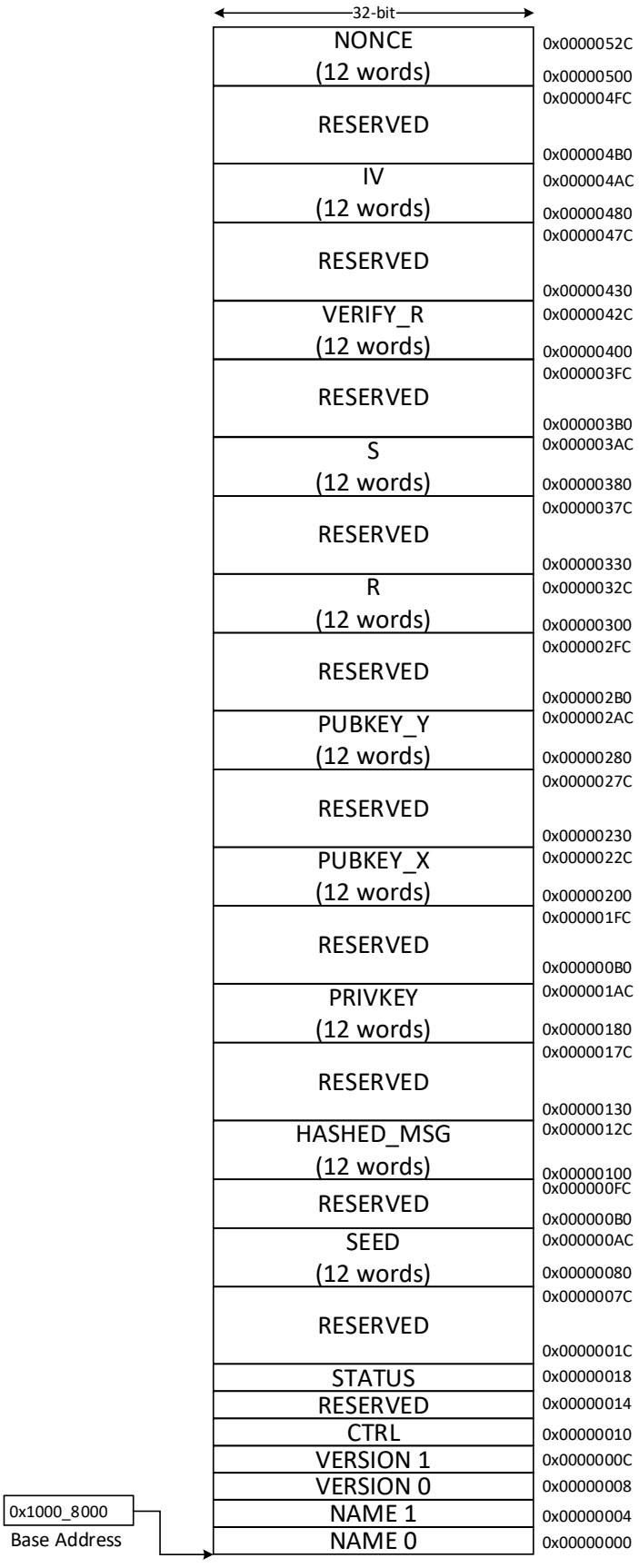

32-bit
NONCE
(12 words)
0x0000052C
0x00000500
RESERVED
0x000004FC
0x000004B0
IV
(12 words)
0x000004AC
0x00000480
RESERVED
0x0000047C
0x00000430
VERIFY_R
(12 words)
0x0000042C
0x00000400
RESERVED
0x000003FC
0x000003B0
S
(12 words)
0x000003AC
0x00000380
RESERVED
0x0000037C
0x00000330
R
(12 words)
0x0000032C
0x00000300
RESERVED
0x000002FC
0x000002B0
PUBKEY_Y
(12 words)
0x000002AC
0x00000280
RESERVED
0x0000027C
0x00000230
PUBKEY_X
(12 words)
0x0000022C
0x00000200
RESERVED
0x000001FC
0x000000B0
PRIVKEY
(12 words)
0x000001AC
0x00000180
RESERVED
0x0000017C
0x00000130
HASHED_MSG
(12 words)
0x0000012C
0x00000100
RESERVED
0x000000FC
0x000000B0
SEED
(12 words)
0x000000AC
0x00000080
RESERVED
0x0000007C
0x0000001C
STATUS
0x00000018
RESERVED
0x00000014
CTRL
0x00000010
VERSION 1
0x0000000C
VERSION 0
0x00000008
NAME 1
0x00000004
NAME 0
0x00000000
0x1000_8000
Base Address

Figure -ECDSA Address Map

4.6.4.1. NAME

Read-only register consists of the name of component.

4.6.4.2. VERSION

Read-only register consists of the version of component.

4.6.4.3. CTRL

The control register consists of the following flags:

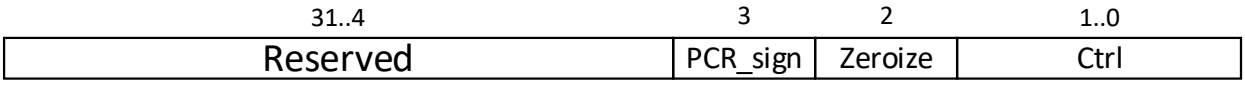


Figure - ECDSA CTRL register

- Ctrl = 0b00

No Operation.

- Ctrl = 0b01

Trigs the ECDSA core to start the initialization and perform keygen operation.

- Ctrl = 0b10

Trigs the ECDSA core to start the signing operation for a message block.

- Ctrl = 0b11

Trigs the ECDSA core to start verifying a signature for a message block.

- ZEROIZE

Trigs the ECC core to clear all registers to avoid any SCA information leakage.

- PCR_Sign

Trigs the ECC core to run PCR signing flow.

4.6.4.4. STATUS

The read-only status register consists of the following flags:

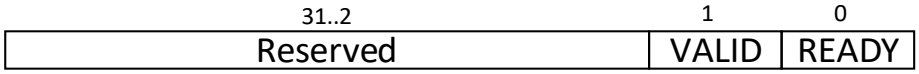


Figure - ECDSA STATUS register

- READY

Indicates if the core is ready to process the inputs.

- VALID

Indicates if the ECDSA process is computed and the output is valid.

4.6.5. Pseudocode

4.6.5.1. Keygen

```
Input:
    seed            //384-bit seed (12 32-bit words)
    nonce           //384-bit nonce (12 32-bit words)
Output:
    privKey         //384-bit (12 32-bit words)
    pubKey_x        //384-bit (12 32-bit words)
    pubKey_y        //384-bit (12 32-bit words)

//wait for the ECC core to be ready (STATUS flag should be 2'b01 or 2'b11)
read_data = 0;
while(read_data == 0){
   read_data = read(ADDR_STATUS);
};

//feed the required inputs
write(ADDR_SEED, seed);
write(ADDR_NONCE, nonce);
write(ADDR_IV, IV);

//trig the ECC for performing Keygen
write(ADDR_CTRL, {30'b0, 2'b01});  (STATUS flag will be changed to 2'b00)

//wait for the ECC core to be ready and valid (STATUS flag should be 2'b11)
read_data = 0;
while(read_data == 0){
   read_data = read(ADDR_STATUS);
};

//reading the outputs
privKey  = Read(ADDR_PRIVKEY);
pubKey_x = Read(ADDR_PUBKEY_X);
pubKey_y = Read(ADDR_PUBKEY_Y);

write(ADDR_CTRL, {29'b0, ZEROIZE, 2'b00}); //Clear all ECC registers

Return privKey, pubKey_x, pubKey_y;
```


4.6.5.2. Signing

```
Input:
    hashed_msg      //384-bit hash of message (12 32-bit words)
    privKey         //384-bit private key (12 32-bit words)
Output:
    r               //384-bit (12 32-bit words)
    s               //384-bit (12 32-bit words)

//wait for the ECC core to be ready (STATUS flag should be 2'b01 or 2'b11)
read_data = 0;
while(read_data == 0){
   read_data = read(ADDR_STATUS);
};

//feed the required inputs
write(ADDR_HASHED_MSG, hashed_msg);
write(ADDR_PRIVKEY, privKey);
write(ADDR_IV, IV);

//trig the ECC for performing Signing
write(ADDR_CTRL, {30'b0, 2'b10});  (STATUS flag will be changed to 2'b00)

//wait for the ECC core to be ready and valid (STATUS flag should be 2'b11)
read_data = 0;
while(read_data == 0){
   read_data = read(ADDR_STATUS);
};

//reading the outputs
r = read(ADDR_SIGN_R);
s = read(ADDR_SIGN_S);

write(ADDR_CTRL, {29'b0, ZEROIZE, 2'b00}); //Clear all ECC registers

Return r, s;
```


4.6.5.3. Verifying

```
Input:
    hashed_msg      //384-bit hash of message (12 32-bit words)
    pubKey_x        //384-bit (12 32-bit words)
    pubKey_y        //384-bit (12 32-bit words)
    r               //384-bit (12 32-bit words)
    s               //384-bit (12 32-bit words)
Output:
    Verify_flag     //a boolean to verify a valid signature


//wait for the ECC core to be ready (STATUS flag should be 2'b01 or 2'b11)
read_data = 0;
while(read_data == 0){
   read_data = read(ADDR_STATUS);
};

//feed the required inputs
write(ADDR_HASHED_MSG, hashed_msg);
write(ADDR_PUBKEY_X, pubKey_x);
write(ADDR_PUBKEY_Y, pubKey_y);
write(ADDR_SIGN_R, r);
write(ADDR_SIGN_S, s);

//trig the ECC for performing Verifying
write(ADDR_CTRL, {30'b0, 2'b11});  (STATUS flag will be changed to 2'b00)

//wait for the ECC core to be ready and valid (STATUS flag should be 2'b11)
read_data = 0;
while(read_data == 0){
   read_data = read(ADDR_STATUS);
};

//reading the outputs
r' = read(ADDR_VERIFY_R);

write(ADDR_CTRL, {29'b0, ZEROIZE, 2'b00}); //Clear all ECC registers

Return r'==r;
```

4.6.6. SCA Countermeasure

The described ECDSA has three main routines: KeyGen, Signing, and Verifying. Since Verifying routine requires operation with public values rather than a secret value, our side-channel analysis does not cover this routine. Our evaluation covers KeyGen, Signing routines where the secret values are processed.

KeyGen consists of HMAC DRBG and scalar multiplication, while Signing first requires a message hashing and then follows the same operations with KeyGen (HMAC DRBG and scalar multiplication). The last step of Signing is generating “S” as the proof of signature. Since HMAC DRBG and hash operations are evaluated separately in our document. This evaluation covers scalar multiplication and modular arithmetic operations.

4.6.6.1.1. Scalar Multiplication

To perform the scalar multiplication, the Montgomery ladder is implemented which is inherently resistant to timing and single power analysis (SPA) attacks.
Implementation of complete unified addition formula for the scalar multiplication avoids information leakage and enhances architecture from security/mathematical perspective.
To protect the architecture against horizontal power/EM and differential power analysis (DPA) attacks, two countermeasures are embedded to the design [9]. Since these countermeasures require random inputs, HMAC-DRBG is fed by “IV” to generate these random values.
Since HMAC-DRBG generates random value in a deterministic way, firmware MUST feed different IV to ECC engine for EACH keygen and signing operations.
The prior work shows that the countermeasure still shows enough resistance even though the attack somehow finds a stronger attack mechanism.

4.6.6.1.2. Base Point Randomization

This countermeasure is achieved using the randomized base point in projective coordinates. Hence, the base point G=(Gx, Gy) in affine coordinates is transformed and randomized to projective coordinates as (X, Y, Z) using a random value λ as follows:

$$X = G_x \times \lambda$$
$$Y = G_y \times \lambda$$
$$Z = \lambda$$

This approach does not have the performance/area overhead since the architecture is variable-base-point implemented.

4.6.6.1.3. Scalar Blinding

This countermeasure is achieved by randomizing the scalar k as follows:

$$k_{randomized} = k + rand \times n$$

Bases on [10], half of the bit size of n is required in order to prevent advanced DPA attacks. Therefore, $rand$ has 192 bits, and the blinded scalar $k_{randomized}$ has 576 bits. Hence, this countermeasure extends the Montgomery ladder iterations due to extended scalar,
This approach is achieved at the cost of 50% more latency on scalar multiplication and adding one lightweight block, including one 32*32 multiplier and an accumulator.

NOTE: the length of rand is configurable to have a trade-off between the required protection and performance.

4.6.6.2. Making countermeasures embedded into HMAC_DRBG

In the first step of Keygen operation, the privkey is generated using HMAC_DRBG by feeding two inputs, i.e., seed and nonce. To avoid SCA information leakage during this operation, we embed masking countermeasures into HMAC_DRBG architecture.
Each round of SHA512 execution needs 6,432 random bits, and one HMAC operation needs at least 4 rounds of SHA512 operations. Furthermore, each HMAC_DRBG round needs at least 5 rounds of HMAC operations. However, the proposed architecture uses a lightweight LFSR and provides first-order DPA attack protection with negligible latency and hardware resource overhead.

4.6.6.3. ECDSA Signing Nonce Leakage

Generating “S” as the proof of signature at the steps of the signing operation is leaking where the hashed message is signed with private key and ephemeral key as follows:

$$s = k^{-1} \times (h + r \times privKey) \, mod \, n$$

Since the given message is known or the signature part r is known. The attacker can perform a known-plaintext attack. The attacker can sign multiple messages with the same key, or the attacker can observe part of the signature that is generated with multiple messages but the same key.
The evaluation shows that the CPA attack can be performed with a small number of traces, respectively. Thus, an arithmetic masked design for these operations is implemented.

4.6.6.3.1. Masking Signature

This countermeasure is achieved by randomizing the privkey as follows:

$$s = \left[k^{-1} \times \left((h - d) + r \times (privKey - d)\right)\right] + [k^{-1} \times (d + r \times d)] \; mod \, n$$

Although computation of “S” seems the most vulnerable point in our scheme, the operation does not have a big contribution to overall latency. Hence, masking these operations have low overhead on the cost of the design.

3.2.1.1. TVLA results

Test vector leakage assessment (TVLA) provides a robust test using a t-test to evaluate the differences between sets of acquisitions to determine if one set of measurement can be distinguished from the other. This technique can detect different types of leakages, providing a clear indication of leakage or lack thereof.
In practice, observing t-value greater than a specific threshold (mainly 4.5) indicates the presence of leakage. However, in ECC due to its latency, around 5 million samples are required to be captured, and it leads to many false positives and the TVLA threshold can be considered a higher value than 4.5. Based on the following figure from “Side-Channel Analysis and Countermeasure

Design for Implementation of Curve448 on Cortex-M4" by Bisheh-Niasar et. al., the threshold can be considered equal to 7 in our case.

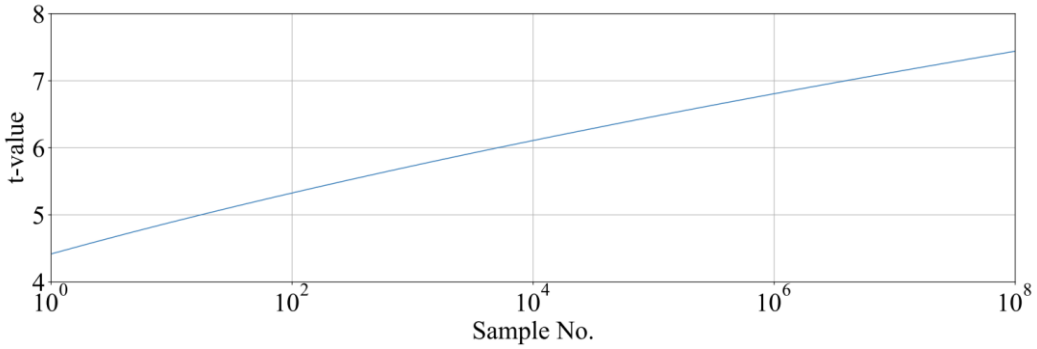


Figure - TVLA threshold as a function of the number of samples per trace.

3.2.1.1.1. Keygen TVLA

3.2.1.1.2. Signing TVLA

The TVLA results for performing privkey-dependent leakage detection using 20,000 traces is shown below. Based on this figure, there is no leakage in ECC signing by changing the privkey after 20k operations.

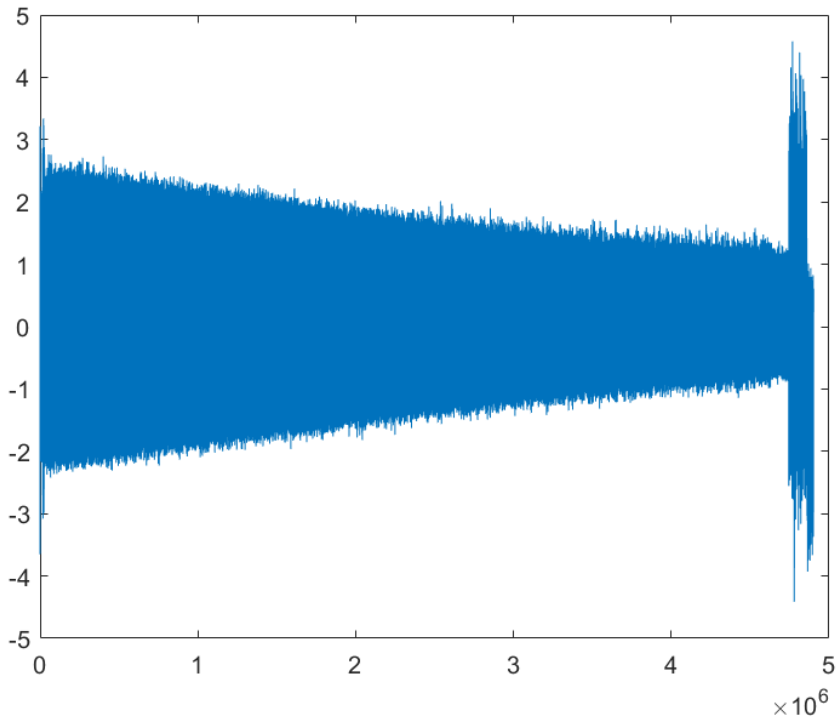


Figure – privkey-dependent leakage detection using TVLA for ECC signing after 20,000 traces.

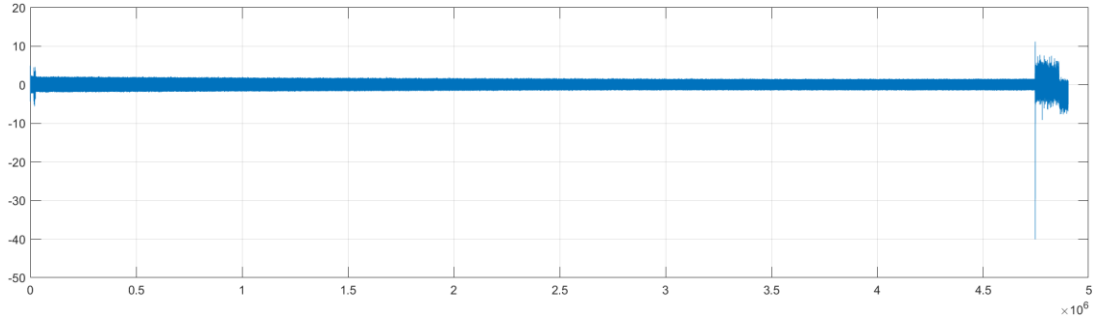


Figure – message-dependent leakage detection using TVLA for ECC signing after 64,000 traces

The point with t-value equal to -40 is mapped to the Montgomery conversion of the message that is a publicly know value (no secret is there). By ignoring those corresponding samples, there are some sparse samples with t-value greater than 7 as follows:

sample	duration	cycle	t-value	Operation
4,746,127	214	911,381.4	11.2	start of mont_conv(msg)
4,746,341		911,422.5	-40	end of mont_conv(msg)
4,757,797	1	913,622.0	7.4	inv_q
4,768,302	1	915,639.0	7.8	inv_q
4,779,610	1	917,810.1	-9.1	inv_q
4,788,120	1	919,444.0	7.6	inv_q
4,813,995	1	924,412.0	7.3	inv_q
4,822,693	1	926,082.1	7.5	inv_q
4,858,671	to the end	932,989.8	-7.6	Ended ECC signing

4.6.7. Performance

The ECC core performance is reported considering two different architectures: (i) pure hardware architecture, and (ii) hardware/software architecture.

4.6.7.1. Pure Hardware Architecture

In this architecture, the ECC interface and controller are implemented in hardware. The performance specification of the ECC architecture is reported as follows:

Operation	Data bus	Cycle count [CCs]	Freq [MHz]	Time [ms]	Throughput [op/s]
Keygen	32-bit	909,648	400	2.274	439
Signing		932,990		2.332	428
Verifying		1,223,938		3.060	326

3.3. Caliptra Vault

3.3.1. PCR Vault

- PCR Vault is a register file that stores measurements to be used by the uC.
- PCR entries are read only registers of 384 bits each.
- Control bits allow for entries to be cleared by FW which sets their values back to 0.
- A lock bit can be set by FW to prevent the entry from being cleared. The lock bit is sticky and will only reset on powergood cycle.

PCRV Register	Address Offset	Description
PCR Control[31:0]		32 Control registers, 32 bits each
PCR Entry[31:0][11:0][31:0]		32 PCR entries, 384 bits each

3.3.1.1. PCR Vault Functional Block

PCR Entries are hash extended using a hash extension function. The hash extension function will take the data currently in the PCR entry specified, concatenate data provided by the FW and perform a SHA384 function on that data, storing the result back into the same PCR entry.

3.3.1.2. PCR Hash Extend Function

FW provides the PCR entry to use as source and destination of the hash extend. HW will copy the PCR into the start of the SHA block and lock those dwords from FW access. FW will then provide the new data, and run the SHA function as usual. After init, the locked dwords will be unlocked.
FW must set a last cycle flag before running the last iteration of the SHA engine. This could be the first “init” cycle, or the Nth “next” cycle. This flag will allow HW to copy the final resulting hash output back to the source PCR.

3.3.1.3. PCR Signing

- PCR signing uses the key in key slot[7] - slot # 8 for PCR signing
- HW implements a HW function called GEN_PCR_HASH
    - HW will read all the PCR from all the PCR slots and hash extends it along with the NONCE that Caliptra FW provides

    - Michael Norris to write the pseudo-code
- HW also implements a HW function called SIGN_PCR which will take the PCR digest that was generated by the previous routine and signs it using the key in the key slot 7 following the same ECC sign flow defined in ECC crypto sub-section.
    - Please note that the above generated PCR DIGEST is used only once for signing by the HW. If a new PCR signing is required, GEN_PCR_HASH needs to be redone.

3.3.2. Key Vault

Key Vault is a register file that stores Keys to be used by the uC, but not observed by it. Each crypto function will have a control register and functional block designed to read from and write to the Key Vault.

KV Register	Description
Key Control[7:0]	8 Control registers, 32 bits each
Key Entry[7:0][15:0][31:0]	8 Key entries, 512 bits each No read or write access

3.3.2.1. Key Vault Functional Block

Keys and Measurements are stored in 512b register files. These have no read or write path from the uC. The entries are read through a passive read mux driven by each crypto, locked entries return zeros.
Entries in the KV must be cleared via control register, or by de-assertion of pwrgood.
Each entry has a control register that is writable by the uC.
The destination valid field is programmed by FW in the crypto block generating the key, and it is passed here at generation time. It cannot be modified after the key has been generated and stored in the KV.

KV Entry Ctrl Fields	Reset	Desc
Lock wr[0]	Cptra_rst_b	Setting the lock wr field will prevent the entry from being written by uC. Keys are always locked. Once set, lock cannot be reset until cptra_rst_b is de-asserted

Lock use[1]	Cptra_rst_b	Setting the lock use field will prevent the entry from being used in any crypto blocks. Once set, lock cannot be reset until cptra_rst_b is de-asserted
Clear[2]	Cptra_rst_b	If unlocked, setting the clear bit will cause KV to clear the associated entry. Clear bit is reset after entry is cleared
Copy[3]	Cptra_rst_b	**ENHANCEMENT:** Setting the copy bit will cause KV to copy the key to the entry written to Copy Dest field.
Copy Dest[7:4]	Cptra_rst_b	**ENHANCEMENT:** Destination entry for copy function
Dest_valid[12:8]	Cptra_rst_b	KV entry can be used with associated crypto if the appropriate index is set. [0] - HMAC KEY [1] - HMAC BLOCK [2] - SHA BLOCK [2] - ECC PRIVKEY [3] - ECC SEED
RSVD		Remaining fields are reserved for future use

3.3.2.2. Key Vault Crypto Functional Block

A generic block will be instantiated in each crypto block to enable access to KV.
Each input to a crypto engine can have a key vault read block associated with it. The KV read block takes in a keyvault read control register that will drive an FSM to copy an entry from the keyvault into the appropriate input register of the crypto engine.
Each output generated by a crypto engine can have it's result copied to a slot in the keyvault. The KV write block takes in a keyvault write control register that will drive an FSM to copy the result from the crypto engine into the appropriate keyvault entry. It also programs a control field for that entry to indicate where that entry can be used.

After programming key vault read control, FW needs to query the associated key vault read status to confirm that the requested key has been copied successfully. Once valid is set and error field reports success the key is ready to be used.
Similarly, after programming key vault write control and initiating the crypto function that will generate the key to be written, FW needs to query the associated key vault write status to confirm that the requested key was generated and written successfully.

KV Read Ctrl Reg	Description
read_en[0]	Indicates that the read data is to come from the key vault. Setting this bit to 1 initiates copying of data from the key vault.
read_entry[3:1]	Key Vault entry to retrieve the read data from for the engine
entry_data_size[9:5]	Size of the source data for SHA512 and HMAC384 Block only.This field is ignored for all other reads.Size is encoded as N-1 dwords.KV flow will pad the 1024 Block data and append the length for values 0-26.All 0 data and Length must be appended in the next Block for values 27-31. 5'd7 - 256b of data 5'd11 - 384b of data 5'd15 - 512b of data
rsvd[31:10]	Reserved field

KV Write Ctrl Reg	Description
write_en[0]	Indicates that the result is to be stored in the key vault. Setting this bit to 1 will copy the result to the keyvault when it is ready.
write_entry[3:1]	Key Vault entry to store the result
entry_is_pcr[4]	Entry selected is a PCR slot
hmac_key_dest_valid[5]	HMAC KEY is a valid destination
hmac_block_dest_valid[6]	HMAC BLOCK is a valid destination
sha_block_dest_valid[7]	SHA BLOCK is a valid destination
ecc_pkey_dest_valid[8]	ECC PKEY is a valid destination
ecc_seed_dest_valid[9]	ECC SEED is a valid destination
rsvd[31:10]	Reserved field

KV Status Reg	Description
ready[0]	Key vault control is idle and ready for a command
valid[1]	Requested flow is done

error[9:2]	SUCCESS - 0x0 - Key Vault flow was successful KV_READ_FAIL - 0x1 - Key Vault Read flow failed KV_WRITE_FAIL - 0x2 - Key Vault Write flow failed

3.3.3. De-obfuscation Engine

To protect software intellectual property from different attacks, particularly, for thwarting an array of supply chain threats, code obfuscation is employed. Hence, the de-obfuscation engine is implemented to decrypt the code.
Advanced Encryption Standard (AES) is used as a Deobfuscation function to encrypt/decrypt data [4]. The hardware implementation is based on Secworks/aes [1]. The implementation supports the two variants 128- and 256-bit keys with a block/chunk size of 128 bits.
The AES algorithm is described as follows:

- The key is fed to AES core to compute and initialize the round key
- The message is broken into 128-bit chunks by the host,
- For each chunk:
  - The message is fed to the AES core,
  - The AES core and its working mode (enc/dec) should be triggered by the host,
  - The AES core status is changed to ready after encryption/decryption processing
- The result digest can be read before processing the next message chunks.

3.3.3.1. Key Vault De-Obfuscation Block Operation

De-obfuscation Engine is used in conjunction with AES crypto to de-obfuscate the UDS and field entropy.
The obfuscation key is driven to the AES key and the data to be decrypted (either obfuscated UDS or obfuscated field entropy) are fed into the AES data.
An FSM is used to manually drive the AES engine and write the decrypted data back to the key vault.
FW is responsible for programming the DOE with the requested function (UDS or Field Entropy de-obfuscation), and the destination for the result.
After de-obfuscation is complete, we can clear out the UDS and Field Entropy values from any flops until cptra_pwrgood de-assertion.

DOE Register	Address	Desc
IV	0x10000000	128 bit IV for DOE flow Stored in Big-Endian representation
CTRL	0x10000010	Controls for DOE flows

STATUS	0x10000014	Valid indicates the command is done and results are stored in keyvault. Ready indicates the core is ready for another command.

DOE Ctrl Fields	Reset	Desc
COMMAND[1:0]	Cptra_rst_b	2'b00 Idle 2'b01 Run UDS flow 2'b10 Run FE flow 2'b11 Clear Obf Secrets
DEST[4:2]	Cptra_rst_b	Dest register for result of de-obfuscation flow. Field entropy will write into DEST and DEST+1 Key entry only, can't go to PCR

3.3.3.2. Key Vault De-obfuscation Flow

ROM will first load IV into DOE and write to DOE control register the Destination for the de-obfuscated result and set the appropriate bit to run UDS and/or Field Entropy flow.
DOE State Machine will take over and load the Caliptra obfuscation key into the key register.
Next, either the obfuscated UDS or FE is loaded into the block register 4 DWORDS at a time.
Results are written to the KV entry specified in DEST field of DOE control register.
State machine will reset the appropriate RUN bit when de-obfuscated key is written to KV. FW can poll this register to know when the flow is complete.

The clear obf secrets command will flush the obfuscation key, obfuscated UDS and Field Entropy from the internal flops. This should be done by ROM once both de-obfuscation flows have been completed.

3.3.4. Data Vault

Data Vault is a set of generic scratch pad registers with specific lock functionality and clearable on cold and warm resets.

- 48B scratchpad registers that are lockable but cleared on cold reset (10 registers)
- 48B scratchpad registers that are lockable but cleared on warm reset (10 registers)
- 4B scratchpad registers that are lockable but cleared on cold reset (8 registers)
- 4B scratchpad registers that are lockable but cleared on warm reset (10 registers)
- 4B scratchpad registers that are cleared on warm reset (8 registers)

3.4. Crypto Blocks Fatal/non-fatal Errors

Errors	Error Type	Desc
ECC_R_ZERO	HW_ERROR_NON_FATAL	Indicates non-fatal error in ECC signing if the computed signature R is equal to 0. FW should change the message or privkey to perform a valid signing.