# 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.

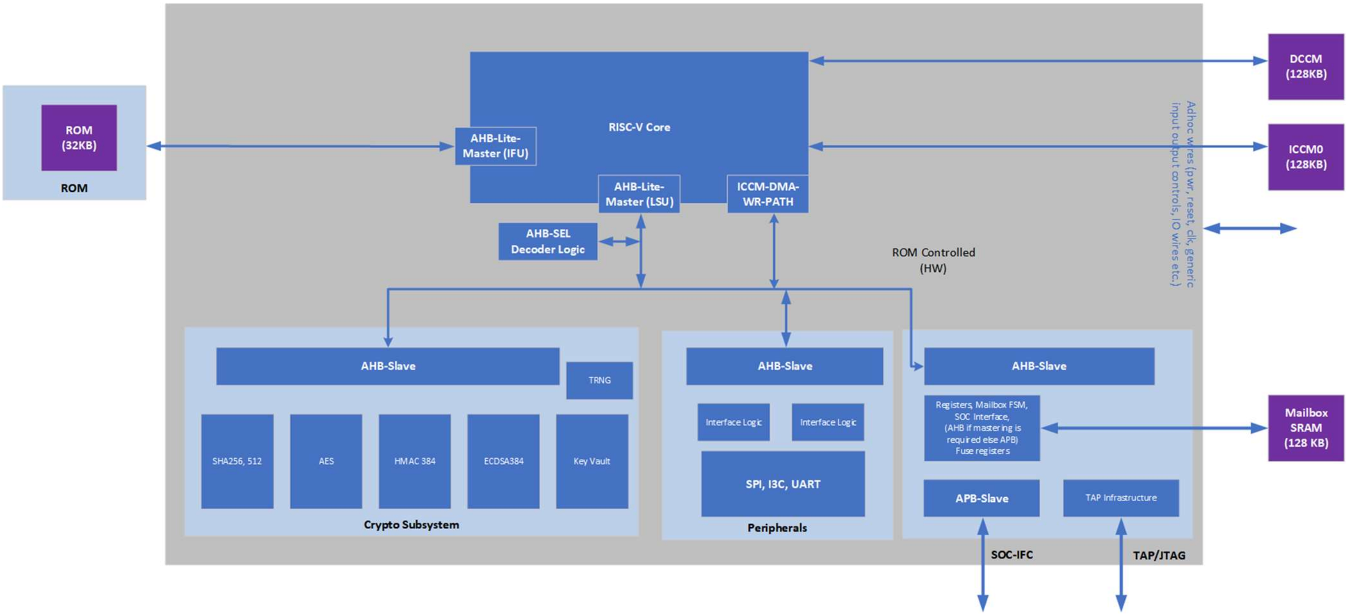


*Figure 1: Caliptra Block Diagram*

## 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

### 1.1.1. 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.

|  |  |  |
|---|---|---|

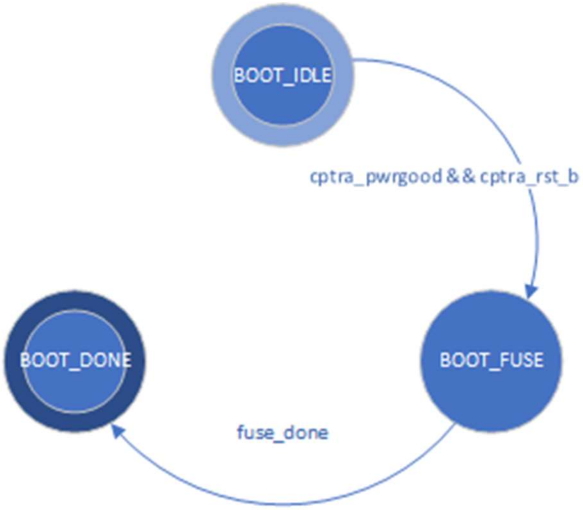


*Figure 2: Mailbox Boot FSM State Diagram*

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.

### 1. 1.1.2 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:

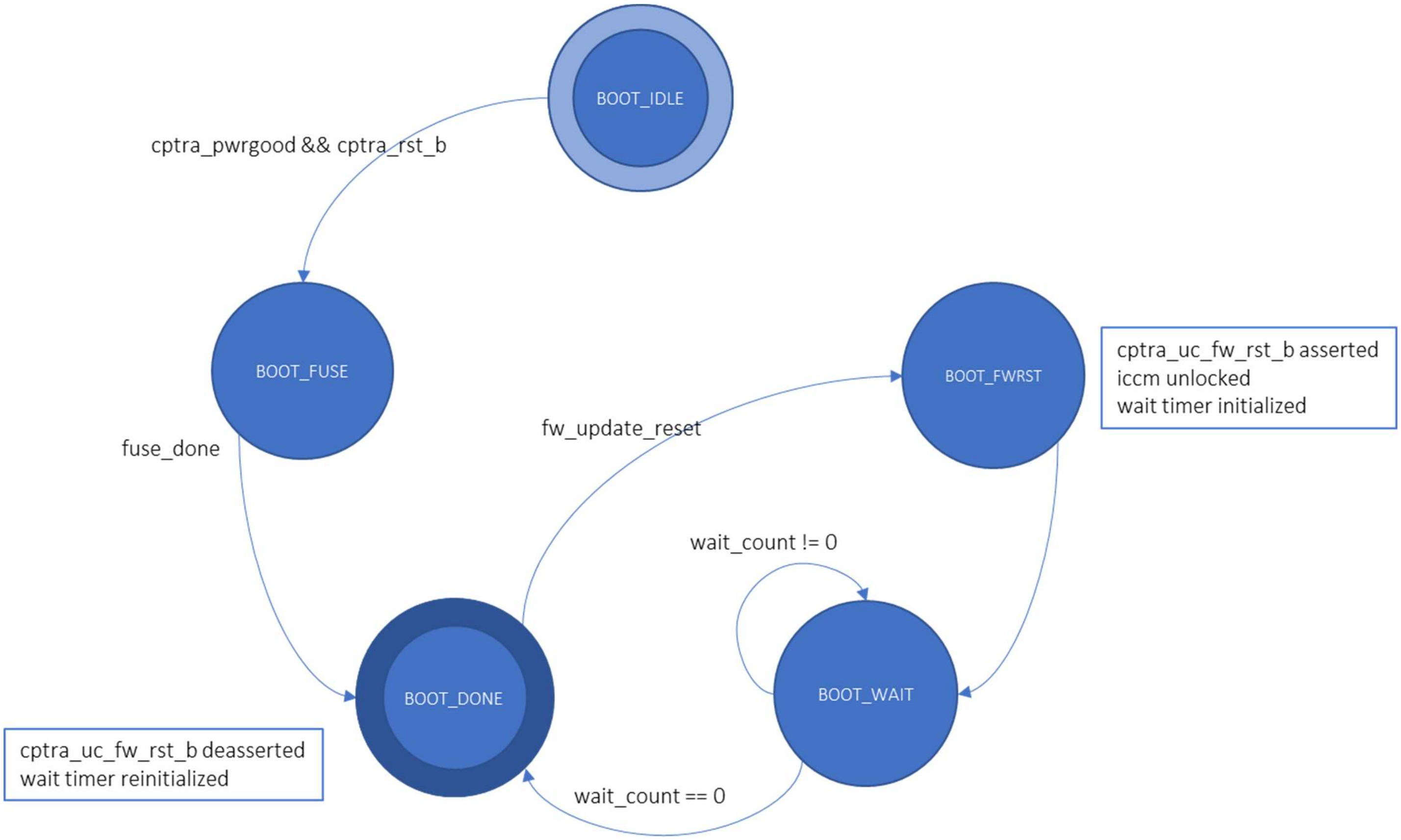


*Figure 3: Mailbox Boot FSM State Diagram for FW update reset*

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

Future iterations of Caliptra will include a Watchdog Timer. The RISC-V processor is required to periodically “pet” the Watchdog to reset the counter. In case of a deadlock scenario, the watchdog timer will overflow a configured threshold and assert a Non-Maskable Interrupt that immediately forces a system reset.
<TODO 0p8>

### 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

TODO 0p8

#### 1.1.7.2. UART

TODO 0p8

#### 1.1.7.3. I3C

TODO 0p8

### 1.1.8. SOC Mailbox

Please refer to the integration specification for mailbox protocol details.

| 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<br>CMD_BUSY - 2’b00 – Indicates the requested command is still in progress<br>DATA_READY - 2’b01 – Indicates the return data is in the mailbox for requested command<br>CMD_COMPLETE- 2’b10 – Indicates the successful completion of the requested command<br>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 |

# 2. Crypto High-Level Architecture

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:

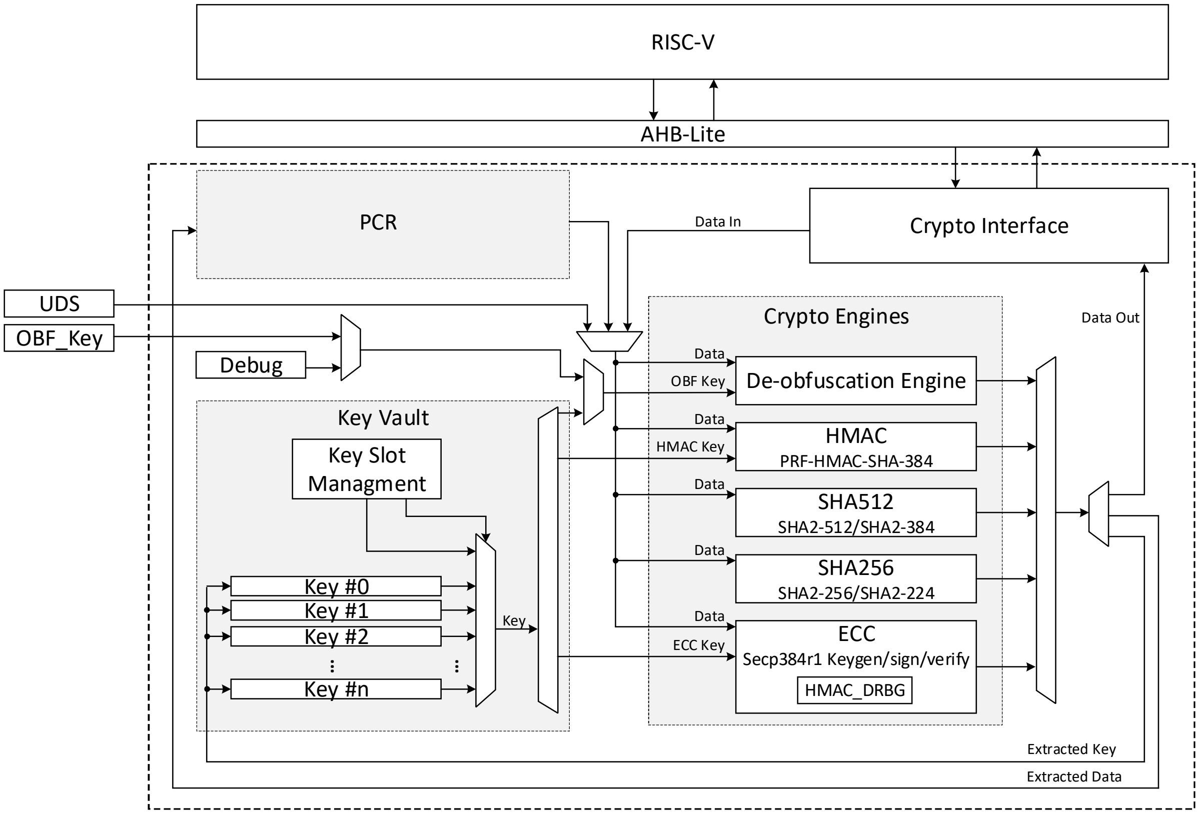


*Figure 4- Caliptra crypto-subsystem*

|  |  |  |
|---|---|---|

## 2.1. 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.

### 2.1.1. Operation

#### 2.1.1.1. 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).

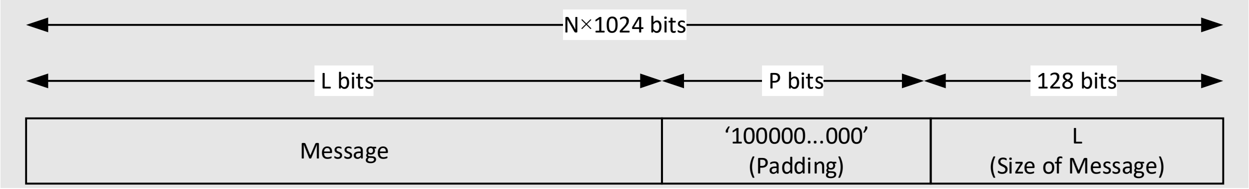


*Figure 5- SHA512 input formatting*

#### 2.1.1.2. 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.

|  |  |  |
|---|---|---|

### 2.1.2. FSM

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

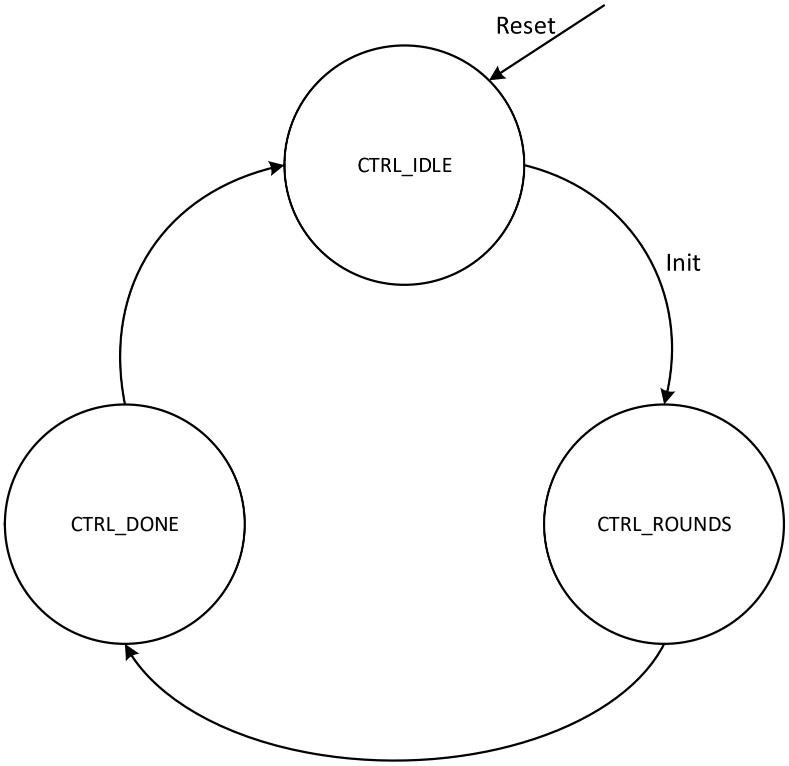


*Figure 6- SHA512 FSM*

### 2.1.3. 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<br>- SHA512/256<br>- SHA384<br>- SHA512 |
| 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*

### 2.1.4. Address Map

The SHA512 address map is shown as follows:

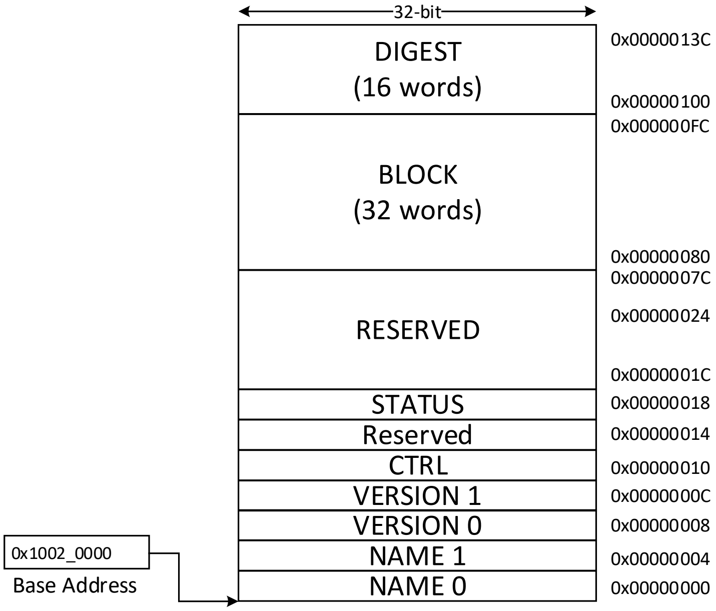


*Figure 7- SHA512 Address Map*

#### 2.1.4.1. NAME

Read-only register consists of the name of component.

#### 2.1.4.2. VERSION

Read-only register consists of the version of component.

| | | |
|---|---|---|

### 2.1.4.3. CTRL

The control register consists of the following flags:

| 31..4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Reserved | MODE | | NEXT | INIT |

*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

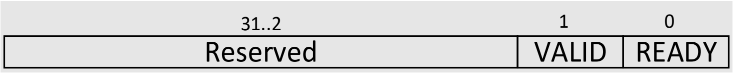
Indicates the SHA512 core to set the type of hashing algorithm. This can be:

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

*Table 11- SHA512 hashing mode*

### 2.1.4.4. 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.

| | | |
|---|---|---|
| | | |

### 2.1.5. 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)

return hash_value
```

*Figure 10- SHA512 pseudocode*

### 2.1.6. 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.

|  |  |  |
|---|---|---|

### 2.1.7. Performance

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

#### 2.1.7.1. 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 |

#### 2.1.7.2. 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*

#### 2.1.7.3. 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*

## 2.2. 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.

### 2.2.1. Operation

#### 2.2.1.1. 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 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).

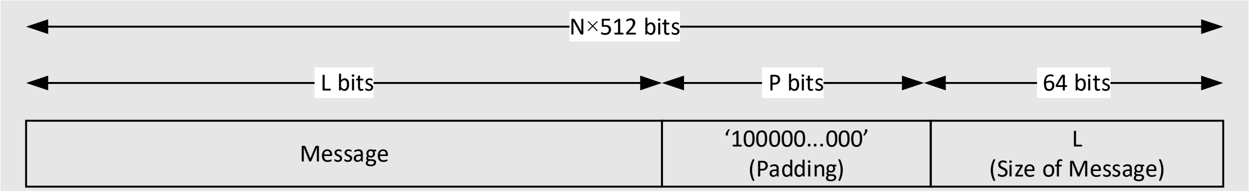


*Figure 11- SHA256 input formatting*

#### 2.2.1.2. 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.

### 2.2.2. FSM

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

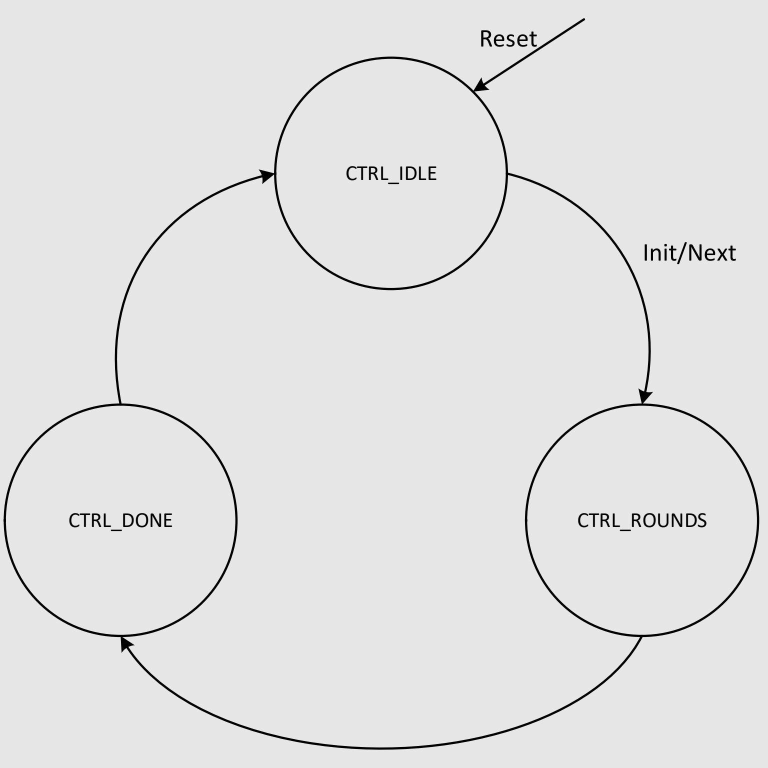


*Figure 12- SHA256 FSM*

### 2.2.3. 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:<br>- SHA256/224<br>- SHA256 |
| 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. |

### 2.2.4. Address Map

The SHA256 address map is shown as follows:

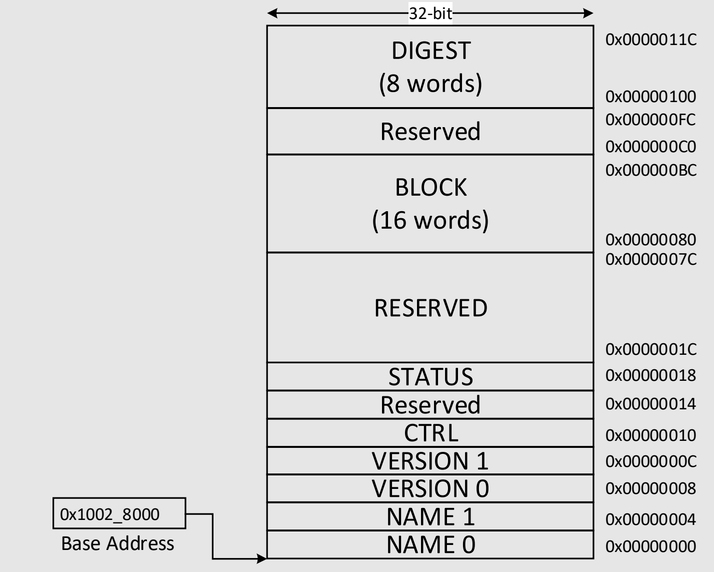


*Figure 13- SHA256 Address Map*

#### 2.2.4.1. NAME

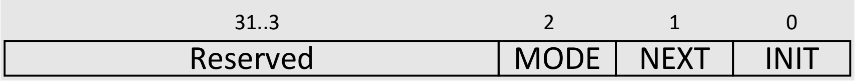
Read-only register consists of the name of component.

#### 2.2.4.2. VERSION

Read-only register consists of the version of component.

#### 2.2.4.3. CTRL

The control register consists of the following flags:



| 31..3 | 2 | 1 | 0 |
|---|---|---|---|
| Reserved | 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.

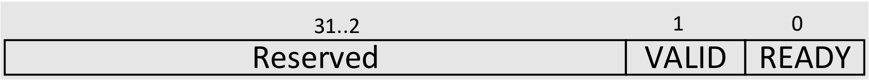
- MODE

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

| MODE | Hashing type |
|---|---|
| 0 | SHA256/224 |
| 1 | SHA256 |

#### 2.2.4.4. STATUS

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



| 31..2 | 1 | 0 |
|---|---|---|
| Reserved | VALID | READY |

*Figure 15- SHA256 STATUS register*

- READY

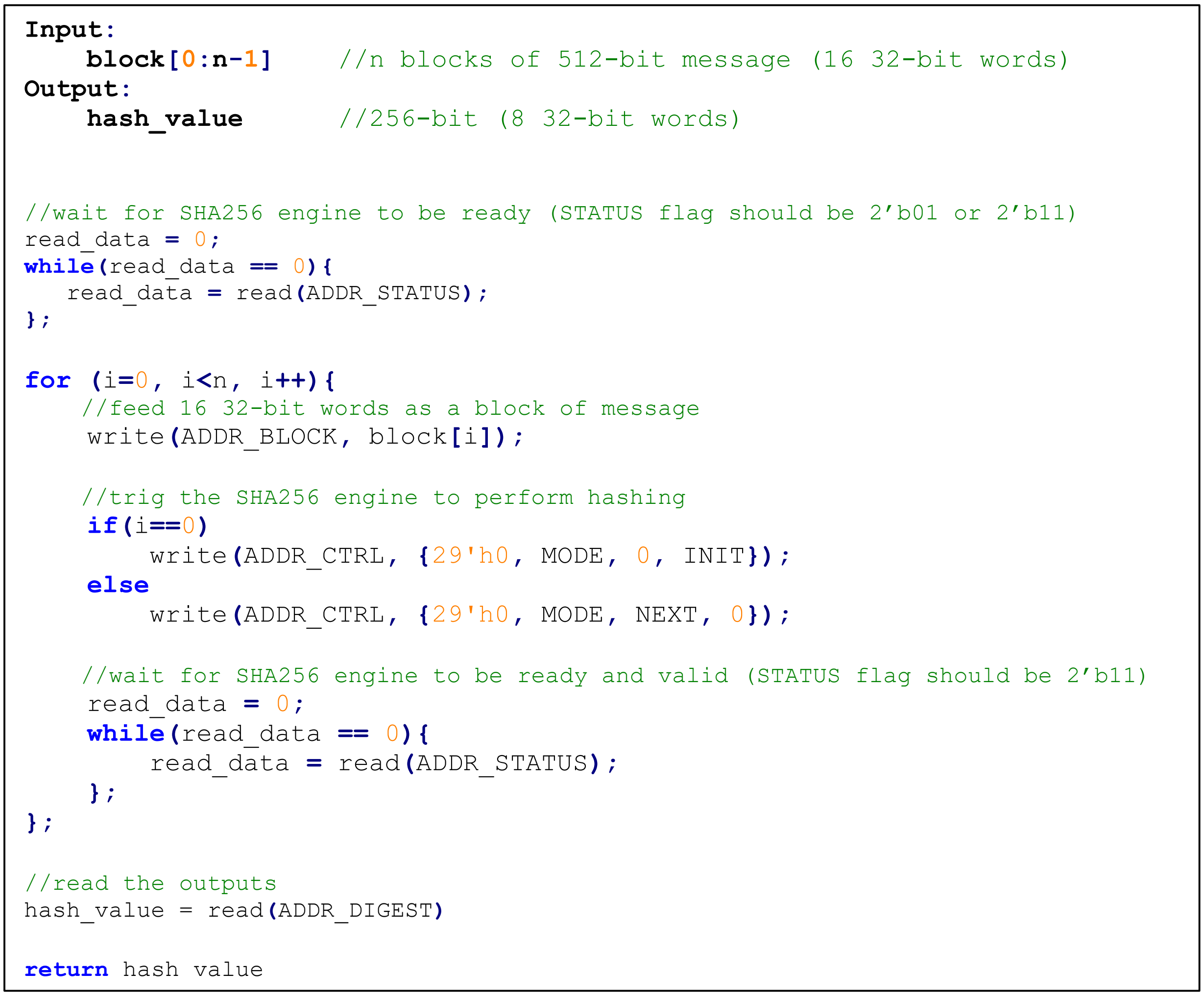
Indicates if the core is ready to process the block.

- VALID

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

### 2.2.5. 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)

return hash value
```

*Figure 16- SHA256 pseudocode*

### 2.2.6. SCA Countermeasure

Please see SCA countermeasure Section for the previous hash implementation.

### 2.2.7. Performance

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

#### 2.2.7.1. Pure Hardware Architecture

| | | |
|---|---|---|
| | | |

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 | | | | | |

### 2.2.7.2. 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 | | | | | |

| | | |
|---|---|---|
| | | |

| | | |
|---|---|---|
| | | |

## 2.3. 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.

### 2.3.1. Operation

#### 2.3.1.1. 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).

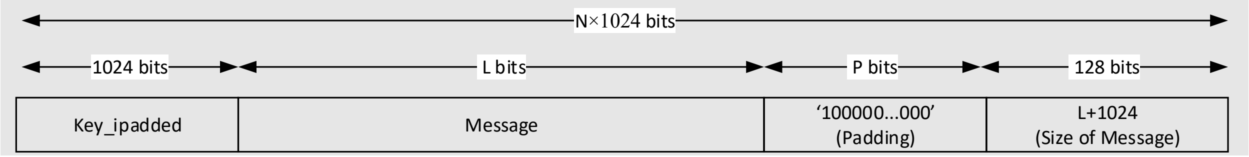


*Figure 23- HMAC input formatting*

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

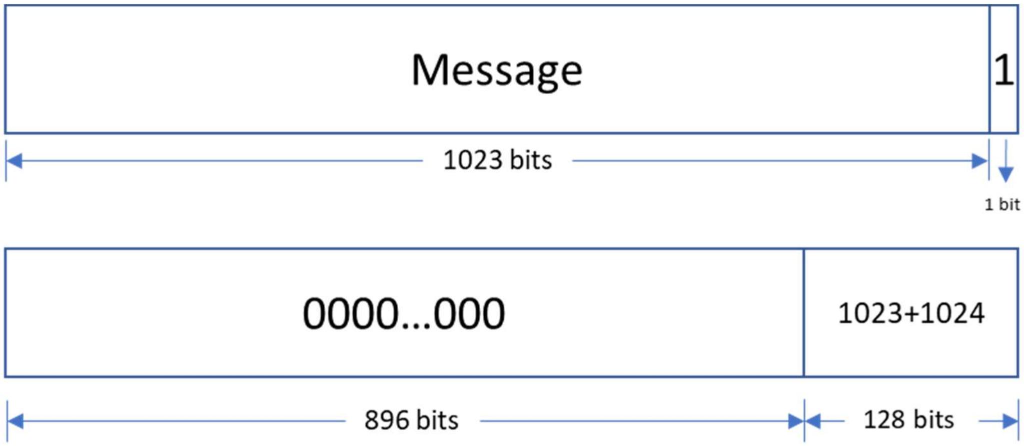


*Figure – Message length of 1023 bits*

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

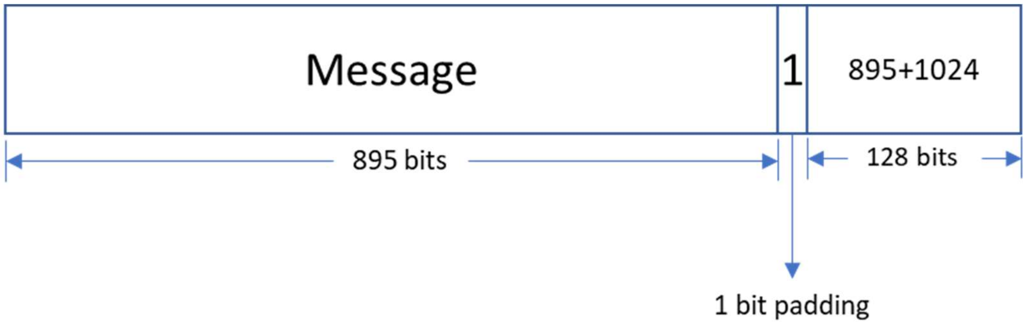


*Figure – 1 bit padding*

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

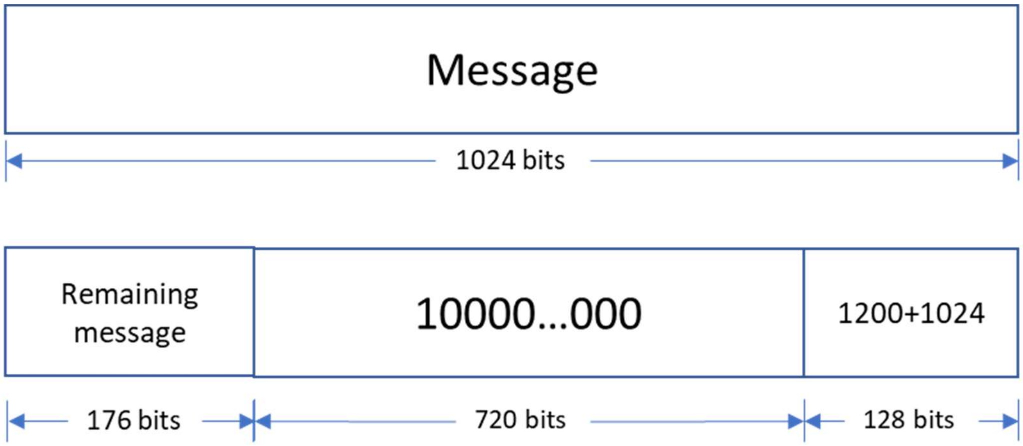


*Figure – Multi block message*

|  |  |  |
|---|---|---|

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

#### 2.3.1.2. 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.

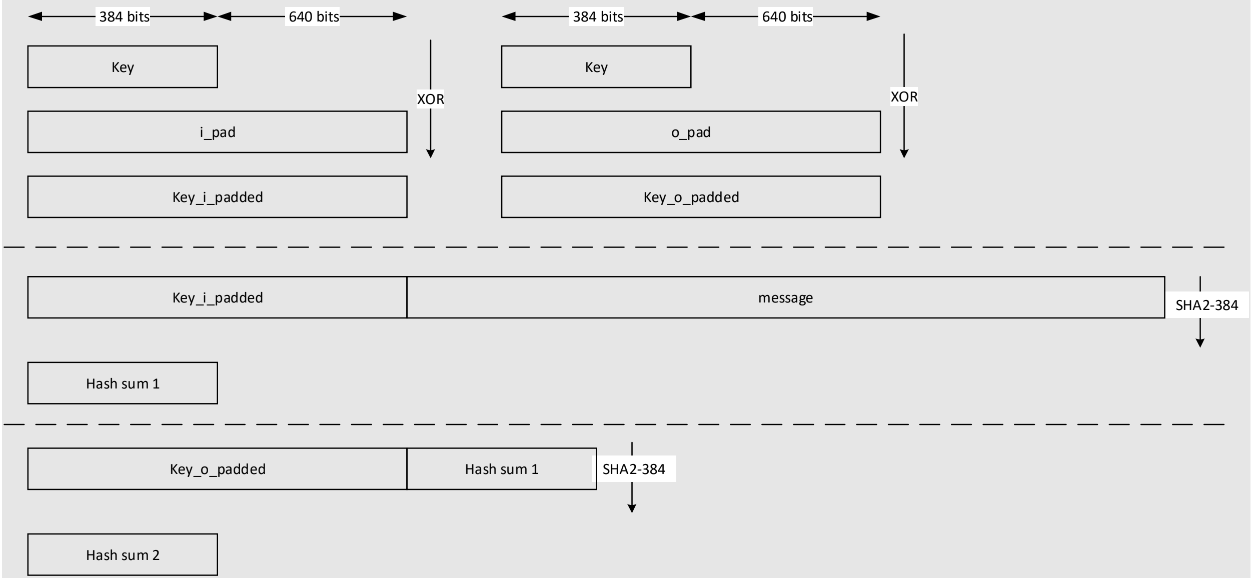


*Figure 24- HMAC-SHA-384-192 data flow*

### 2.3.2. FSM

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

|  |  |  |
|---|---|---|
|  |  |  |

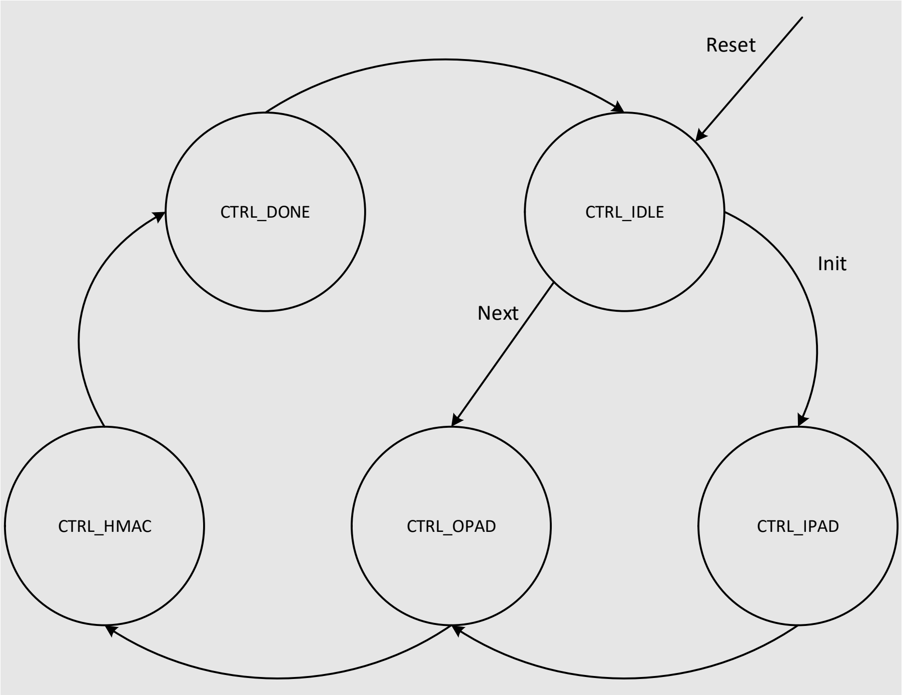


*Figure 25- HMAC FSM*

### 2.3.3. 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. |
| key[383:0] | input | The input key. |
| block[1023:0] | input | The input padded block of message. |
| 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. |

### 2.3.4. Address Map

The HMAC address map is shown as follows:

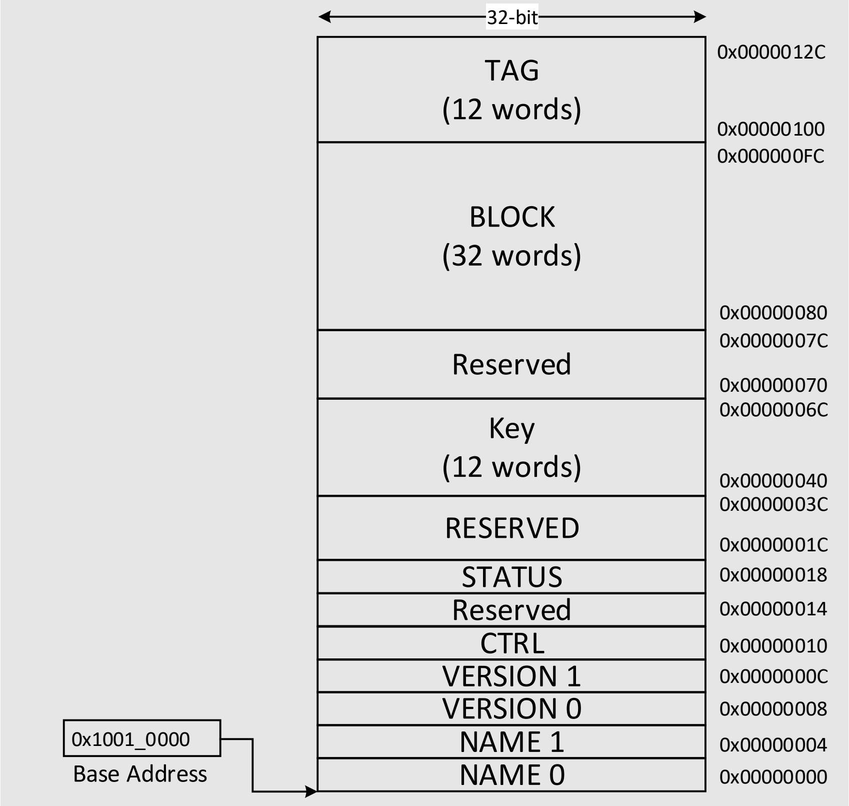


*Figure 26- HMAC Address Map*

#### 2.3.4.1. NAME

Read-only register consists of the name of component.

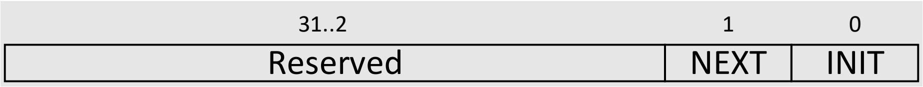
#### 2.3.4.2. VERSION

Read-only register consists of the version of component.

|  |  |  |
|---|---|---|

#### 2.3.4.3. CTRL

The control register consists of the following flags:



| 31..2 | 1 | 0 |
|---|---|---|
| Reserved | NEXT | INIT |

*Figure 27- HMAC CTRL register*

- INIT

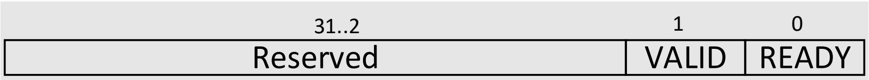
Trigs the HMAC core to start the initialization and processing the key.

- NEXT

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

#### 2.3.4.4. STATUS

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



| 31..2 | 1 | 0 |
|---|---|---|
| Reserved | VALID | READY |

*Figure 28- HMAC STATUS register*

- READY

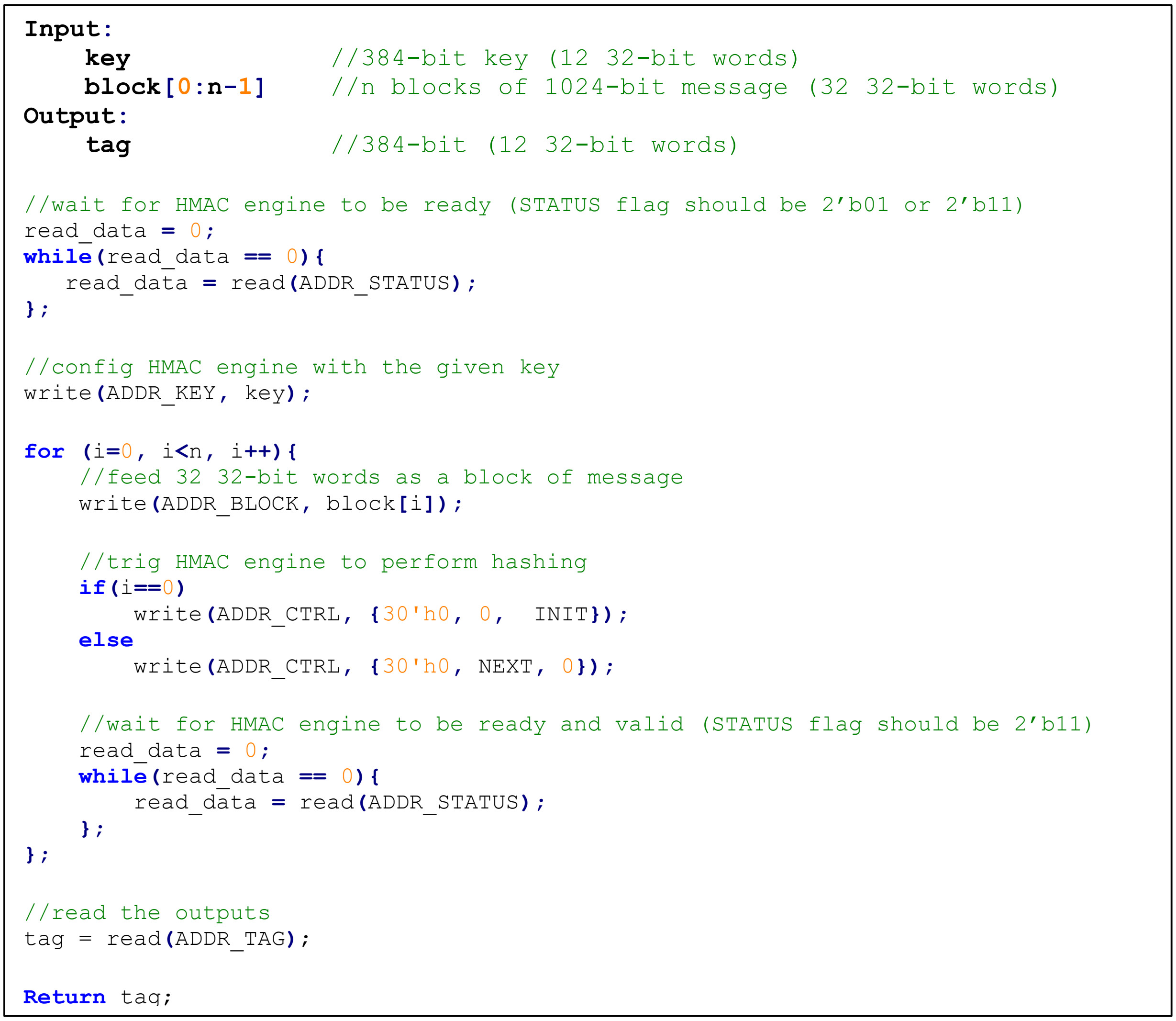
Indicates if the core is ready to process the block.

- VALID

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

### 2.3.5. Pseudocode

The following pseudocode demonstrates how HMAC interface can be implemented.



```
Input:
    key               //384-bit key (12 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);

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);

Return tag;
```

*Figure 29- HMAC pseudocode*

### 2.3.6. SCA Countermeasure

TODO post 0p5: Details will be added

### 2.3.7. Performance

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

#### 2.3.7.1. 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 |

### 2.3.7.2. 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 |

|  |  |  |
|---|---|---|

## 2.4. 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.

### 2.4.1. 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:
Mode 0:

```
2.   Set V_init = 0x01 0x01 0x01 ... 0x01  (V has 384-bit)
3.   Set K_init = 0x00 0x00 0x00 ... 0x00  (K has 384-bit)
4.   cnt = 0x00 (cnt has 8-bit)
5.   K_tmp = HMAC(K_init, V_init || cnt || seed)
6.   V_tmp = HMAC(K_tmp,  V_init)
7.   cnt++
8.   K_new = HMAC(K_tmp,  V_tmp  || cnt || seed)
9.   V_new = HMAC(K_new,  V_tmp)
10.  cnt++
11.  Return V_new
12.  If more random values are needed, set K_init = K_new, V_init = V_new,
     and jump to step 4, otherwise end.
```

For ECC signing operation, the following algorithm is performed to generate the nonce:
Mode 1:

```
13.     hashed_msg = SHA384(message)   (h1 has 384-bit)
14.     Set V_init = 0x01 0x01 0x01 ... 0x01  (V has 384-bit)
15.     Set K_init = 0x00 0x00 0x00 ... 0x00  (K has 384-bit)
16.     K_tmp = HMAC(K_init, V_init || 0x00 || privKey || hashed_msg)
17.     V_tmp = HMAC(K_tmp,  V_init)
```

```
18.      K_new = HMAC(K_tmp,  V_tmp  || 0x01 || privKey || hashed_msg)
19.      V_new = HMAC(K_new,  V_tmp)
20.      Set T = []
21.      T = T || HMAC(K_new, V_new)
22.      Return T if T is within the [1,q-1] range, otherwise:
23.      K_new = HMAC(K_new,  V_new || 0x00)
24.      V_new = HMAC(K_new,  V_tmp)
25.      Jump to 9
```

### 2.4.2. 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. |
| mode | input | Indicates the type of the function. This can be:<br>- 0 for general DRBG function<br>- 1 for DRBG used in ECC signing nonce |
| seed[383:0] | input | The input seed used in mode 0. |
| privkey[383:0] | input | The input privkey used in mode 1. |
| hashed_msg[383 :0] | input | The input hashed_msg used in mode 1. |
| ready | output | When HIGH, the signal indicates the core is ready. |
| nonce[383:0] | output | The hmac_drbg value of the given inputs. |
| valid | output | When HIGH, the signal indicates is the result is ready. |

### 2.4.3. Address Map

The HMAC_DRBG address map is shown as follows:

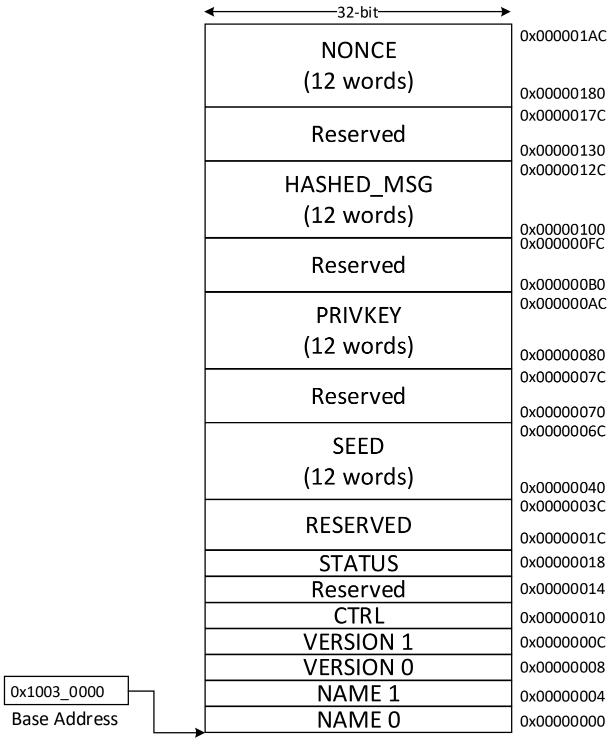


*Figure 13- HMAC_DRBG Address Map*

### 2.4.3.1. NAME

Read-only register consists of the name of component.

### 2.4.3.2. VERSION

Read-only register consists of the version of component.

### 2.4.3.3. CTRL

The control register consists of the following flags:

| 31..3 | 2 | 1 | 0 |
|---|---|---|---|
| Reserved | MODE | NEXT | INIT |

*Figure 14- HMAC_DRBG CTRL register*

- INIT

Trigs the HMAC_DRBG core to start the processing to generate the first 384-bit random value.

- NEXT

|  |  |  |
|---|---|---|

Trigs the HMAC_DRBG core to start the processing to generate another 384-bit random value seeded by the previous process.

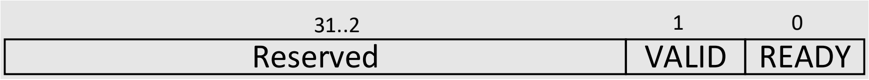
- MODE

Indicates the HMAC_DRBG core to set the type of DRBG algorithm. This can be:

| MODE | Hashing type |
|---|---|
| 0 | General HMAC_DRBG |
| 1 | HMAC_DRBG for ECC signing |

#### 2.4.3.4. STATUS

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



| 31..2 | 1 | 0 |
|---|---|---|
| Reserved | VALID | READY |

*Figure 15- HMAC_DRBG STATUS register*

- READY

Indicates if the core is ready to process the block.

- VALID

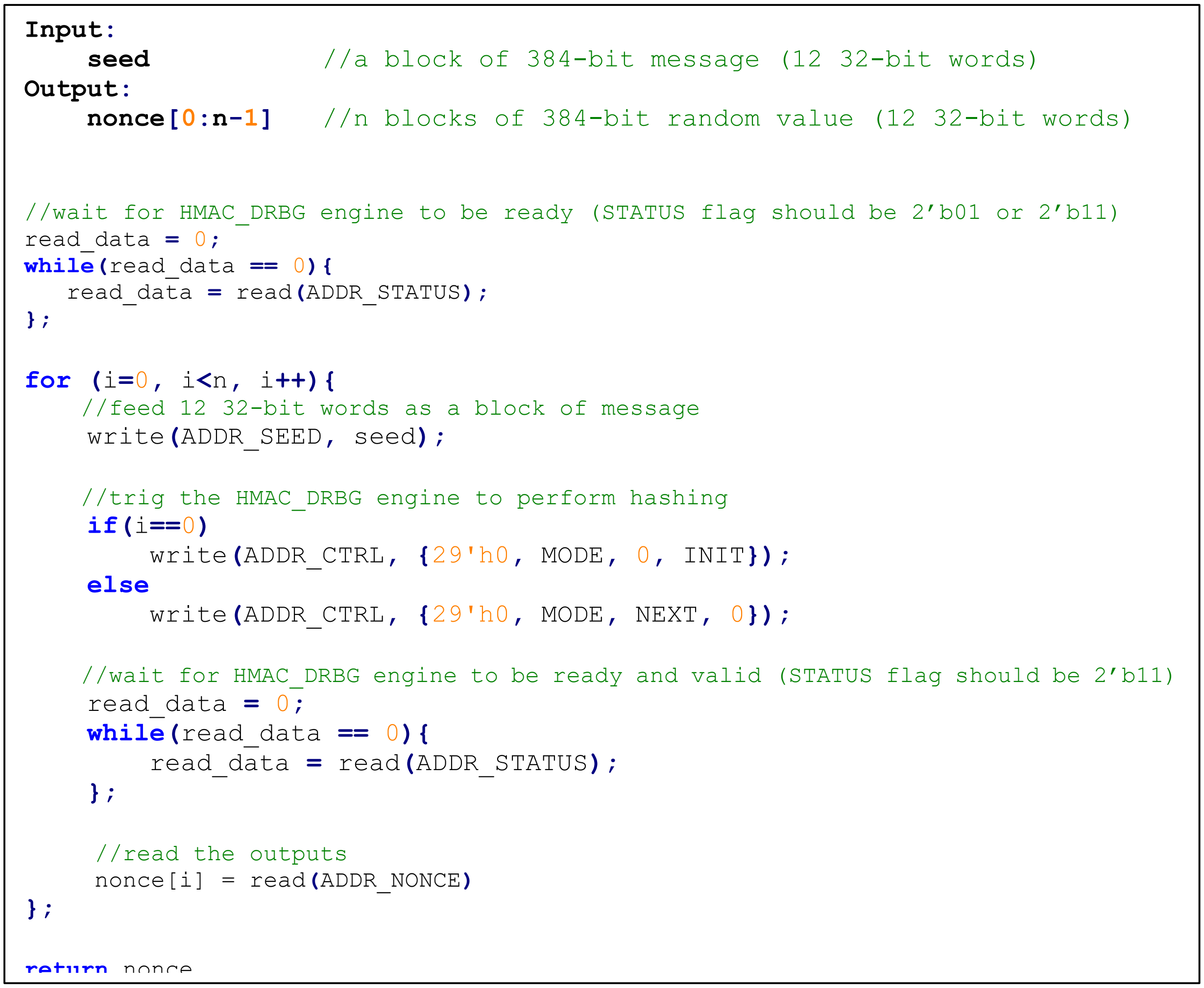
Indicates if the hash value is computed and value stored in NONCE register is valid.

### 2.4.4. Pseudocode

The following pseudocode demonstrates how HMAC_DRBG interface can be implemented.

|   |   |   |
|---|---|---|

### 2.4.4.1. Mode 0

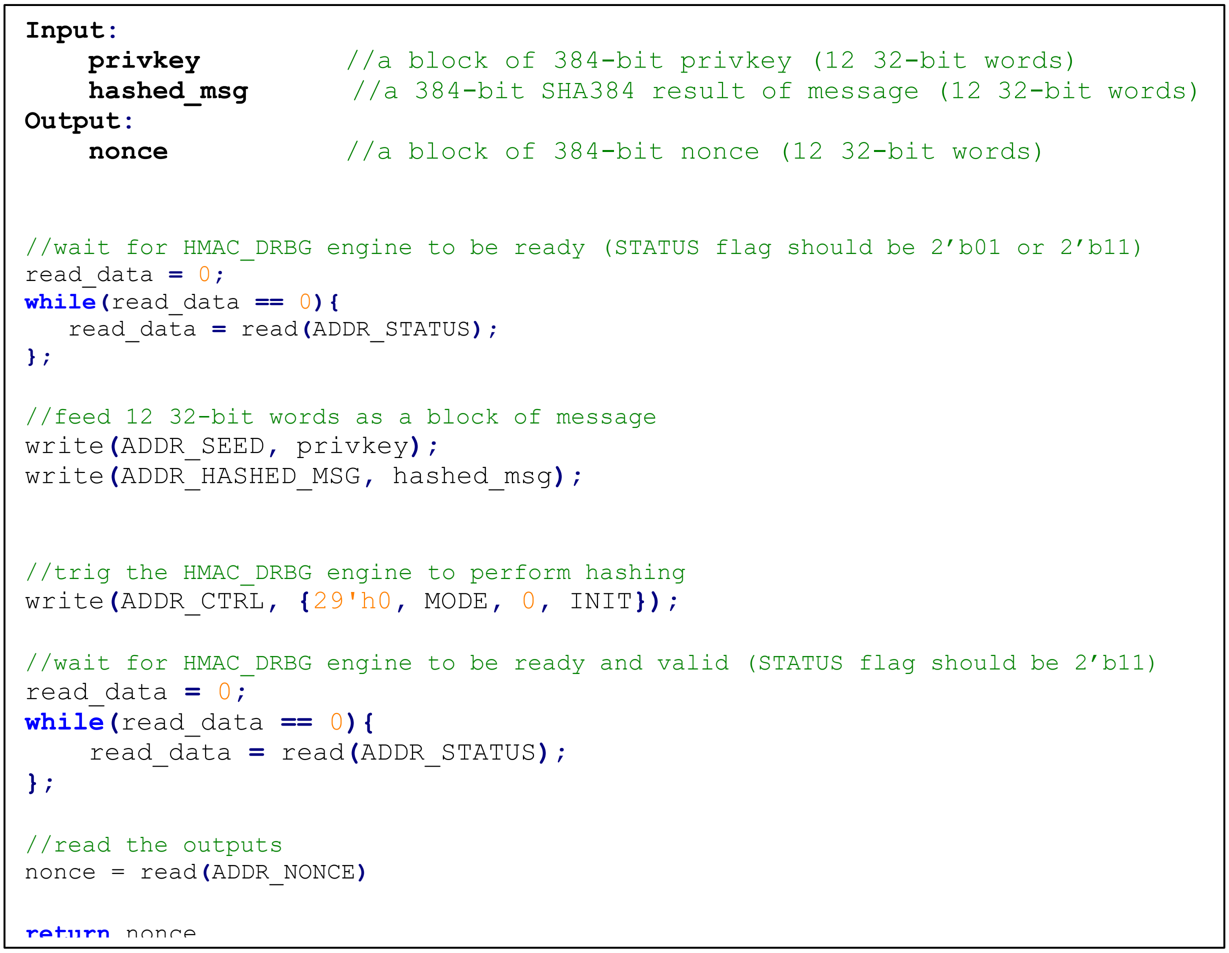


```
Input:
    seed            //a block of 384-bit message (12 32-bit words)
Output:
    nonce[0:n-1]    //n blocks of 384-bit random value (12 32-bit words)


//wait for HMAC_DRBG 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 12 32-bit words as a block of message
    write(ADDR_SEED, seed);

    //trig the HMAC_DRBG 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 HMAC_DRBG 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
     nonce[i] = read(ADDR_NONCE)
};

return nonce
```

*Figure 16- HMAC_DRBG pseudocode in Mode 0*

|  |  |  |
|---|---|---|

### 2.4.4.2. Mode 1



```
Input:
    privkey         //a block of 384-bit privkey (12 32-bit words)
    hashed_msg      //a 384-bit SHA384 result of message (12 32-bit words)
Output:
    nonce           //a block of 384-bit nonce (12 32-bit words)


//wait for HMAC_DRBG 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);
};

//feed 12 32-bit words as a block of message
write(ADDR_SEED, privkey);
write(ADDR_HASHED_MSG, hashed_msg);


//trig the HMAC_DRBG engine to perform hashing
write(ADDR_CTRL, {29'h0, MODE, 0, INIT});

//wait for HMAC_DRBG 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
nonce = read(ADDR_NONCE)

return nonce
```

*Figure 16- HMAC_DRBG pseudocode in Mode 1*

### 2.4.5. SCA Countermeasure

Please see SCA section for the previous HMAC implementation.

## 2.5. 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                                                     =
0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffeffffffff00
00 000000000000ffffffff
                      A                                                     =
0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffeffffffff00
00 000000000000fffffffc
                      B                                                     =
0xb3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5013875ac656398d8a
2e d19d2a85c8edd3ec2aef
                      Gx                                                    =
0xaa87ca22be8b05378eb1c71ef320ad746e1d3b628ba79b9859f741e082542a385502f25dbf
55 296c3a545e3872760ab7
                      Gy                                                    =
0x3617de4a96262c6f5d9e98bf9292dc29f8f41dbd289a147ce9da3113b5f0b8c00a60b1ce1d
7e 819d7a431d7c90ea0e5f
```

### 2.5.1. Operation

The ECDSA consists of three operations.

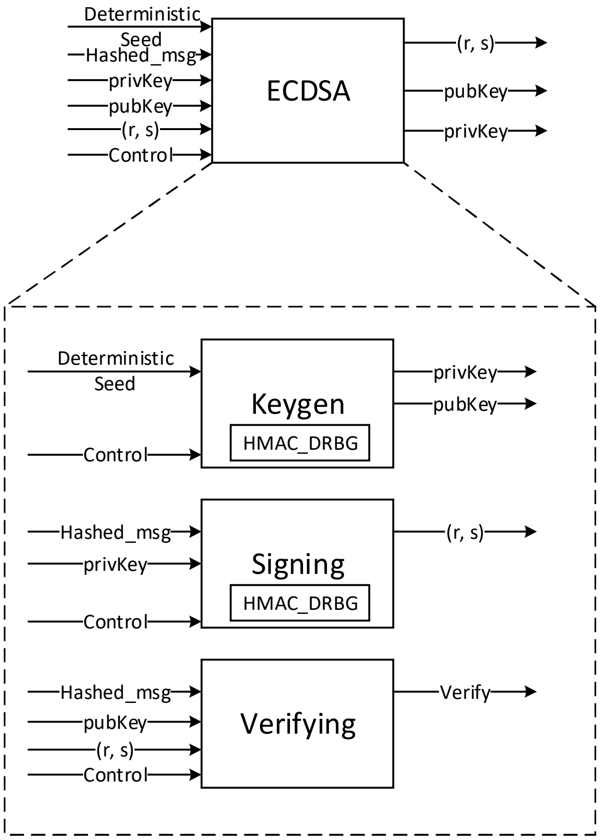


*Figure -ECDSA Operations*

#### 2.5.1.1. Key Generation

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

- Compute privKey = HMAC_DRBG(seed) 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).

#### 2.5.1.2. 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 = $k^{-1}$ × (h + r × privKey) mod n
- Return the signature (r, s), each in the range [1..n-1].

#### 2.5.1.3. 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 = $s^{-1}$ mod n
- Compute R' = (h × s1) × G + (r × s1) × 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

### 2.5.2. Architecture

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

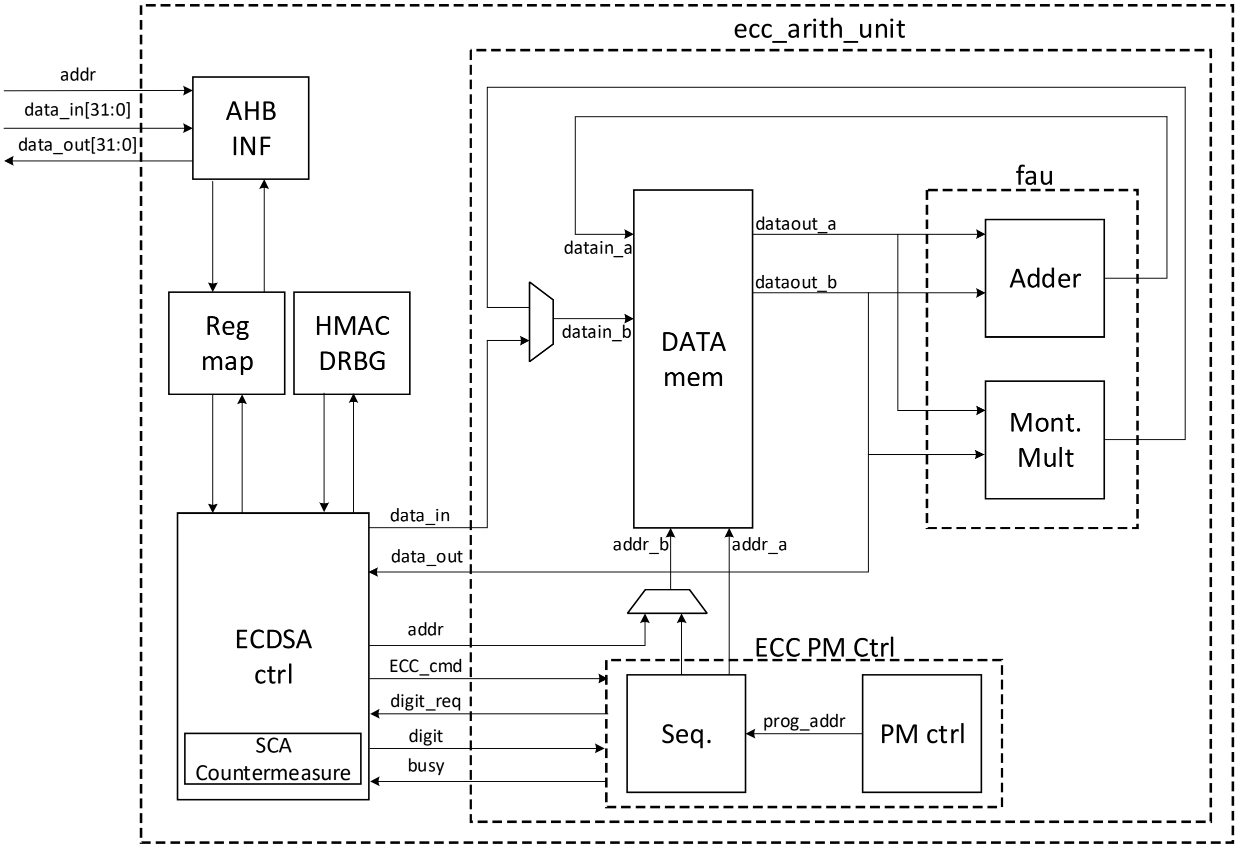


*Figure -ECDSA Architecture*

### 2.5.3. 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:<br>– 0b00: No Operation<br>– 0b01: KeyGen<br>– 0b10: Signing<br>– 0b11: Verifying |
| seed[383:0] | input | The deterministic seed 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. |

### 2.5.4. Address Map

The ECDSA address map is shown as follows:

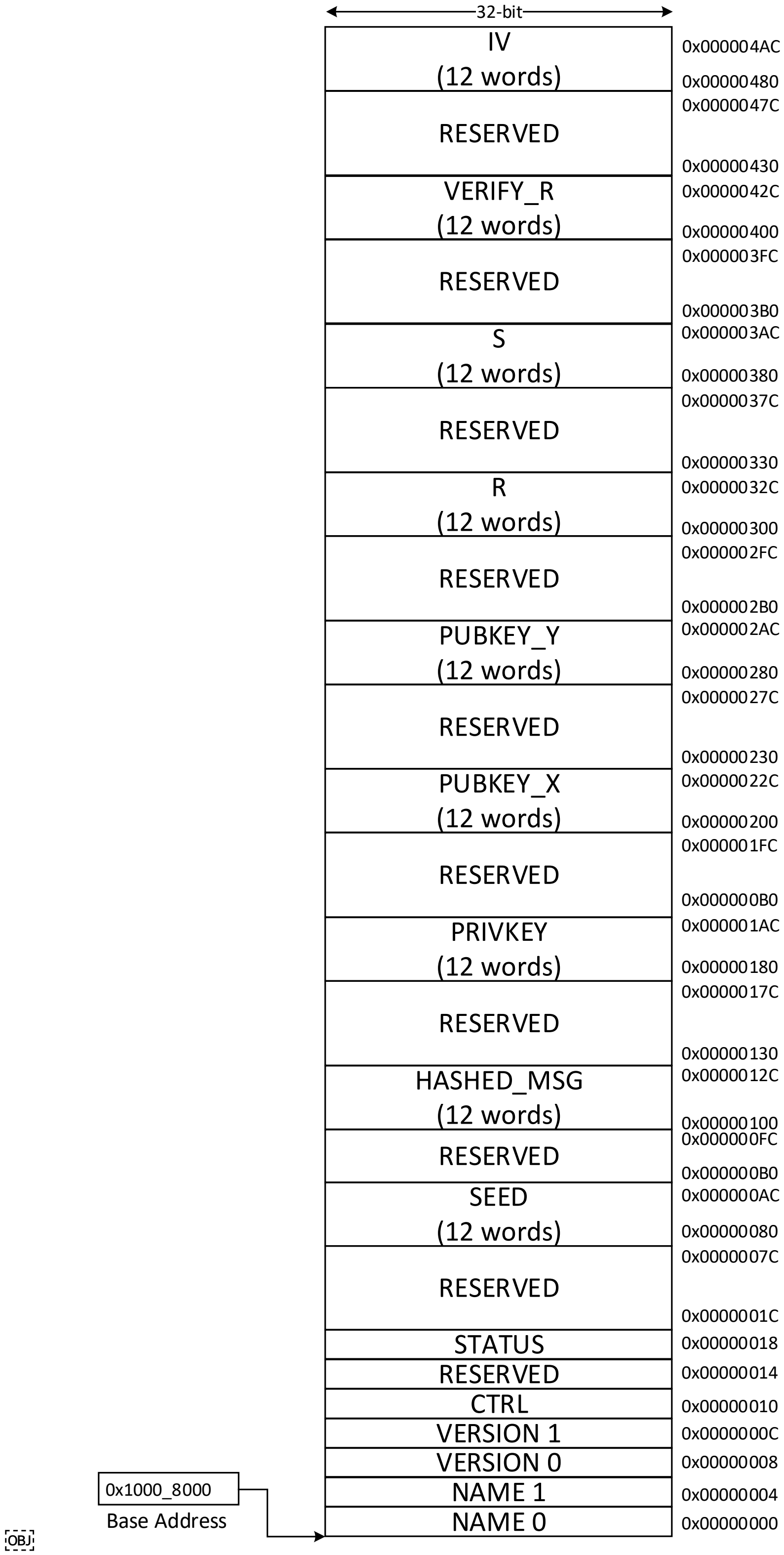


*Figure -ECDSA Address Map*

|  |  |  |
|---|---|---|

#### 2.5.4.1. NAME

Read-only register consists of the name of component.

#### 2.5.4.2. VERSION

Read-only register consists of the version of component.

#### 2.5.4.3. CTRL

The control register consists of the following flags:

| 31..3 | 1 | 0 |
|---|---|---|
| Reserved | Ctrl |  |

*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.

#### 2.5.4.4. STATUS

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

| 31..2 | 1 | 0 |
|---|---|---|
| Reserved | VALID | READY |

*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.

## 2.5.5. Pseudocode

### 2.5.5.1. Keygen

```
Input:
    seed             //384-bit seed (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_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);

Return privKey, pubKey_x, pubKey_y;
```

|   |   |   |
|---|---|---|

### 2.5.5.2. 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);

Return r, s;
```

|  |  |  |
|---|---|---|

### 2.5.5.3. 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);

Return r’==r;
```

## 2.5.6. 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.

### 2.5.6.1. 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.

#### 2.5.6.1.1. 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 $\lambda$ 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.

#### 2.5.6.1.2. 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.

| | | |
|---|---|---|
| | | |

#### 2.5.6.2. 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.

##### 2.5.6.2.1. 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.

### 2.5.7. Performance

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

#### 2.5.7.1. 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 |

|  |  |  |
|---|---|---|
|  |  |  |

## 2.6. Caliptra Vault

### 2.6.1. 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 |
|  |  |  |
|  |  |  |

#### 2.6.1.1. 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.

#### 2.6.1.2. 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.

#### 2.6.1.3. PCR Signing (TODO)

|  |  |  |
|---|---|---|
|  |  |  |

### 2.6.2. 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<br>No read or write access |
|  |  |
|  |  |

#### 2.6.2.1. 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[13:8]** | Cptra_rst_b | KV entry can be used with associated crypto if the appropriate index is set.<br>[0] - HMAC KEY<br>[1] - HMAC BLOCK<br>[2] - SHA BLOCK<br>[2] - ECC PRIVKEY<br>[3] - ECC SEED<br>[4] - ECC MSG |
|  |  |  |
| **RSVD** |  | Remaining fields are reserved for future use |

#### 2.6.2.2. 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.

| 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.<br><br>5'd7 - 256b of data |

| | |
|---|---|
| | 5'd11 - 384b of data<br>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 |
| **ecc_msg_dest_valid[10]** | ECC MSG is a valid destination |
| **rsvd[31:11]** | 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<br>KV_READ_FAIL - 0x1 - Key Vault Read flow failed<br>KV_WRITE_FAIL - 0x2 - Key Vault Write flow failed |

### 2.6.3. 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.

#### 2.6.3.1. 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<br>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.<br>Ready indicates the core is ready for another command. |
|  |  |  |

| | | |
|---|---|---|
| | | |

| DOE Ctrl Fields | Reset | Desc |
|---|---|---|
| **COMMAND[1:0]** | Cptra_rst_b | 2’b00 Idle<br>2’b01 Run UDS flow<br>2’b10 Run FE flow<br>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<br>Key entry only, can’t go to PCR |
| | | |
| | | |

#### 2.6.3.2. 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.

### 2.6.4. 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)

|  |  |  |
|---|---|---|

## 2.7. 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.

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 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 TRNG REQ HW API flow.

## 2.8. 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.

## 2.9. JTAG Implementation

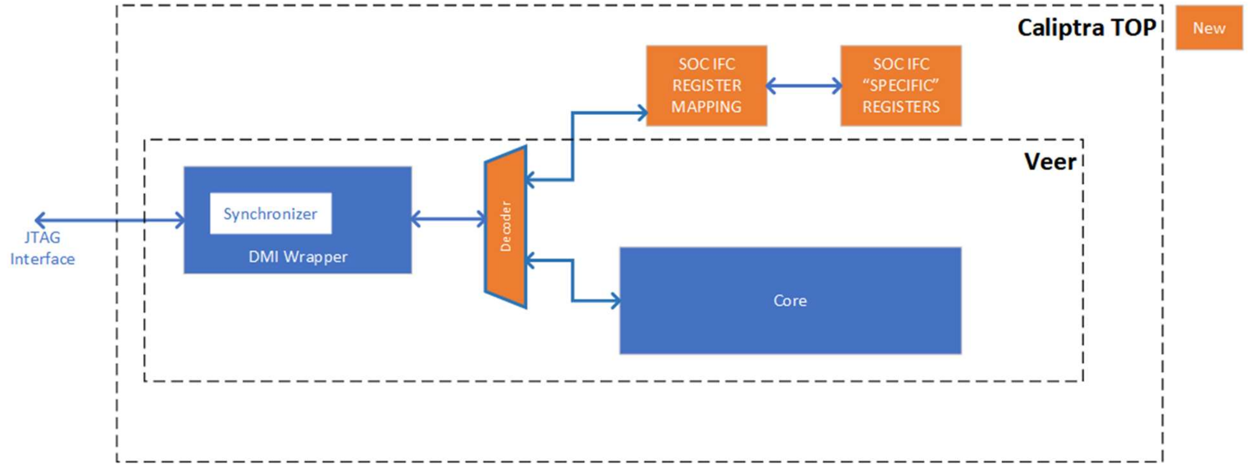
Please see "Debug" section of the Caliptra ROT specification for debug flows.

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.



## 2.10. 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. |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

|   |   |   |
|---|---|---|