POLITECNICO DI TORINO

Department of Electronics and Telecommunications

Master's Degree in Electronic Engineering

Design of an agile cryptographic accelerator for space communications



Supervisor:
Prof. Guido Masera
Prof. Maurizio Martina
Ph.D. Alessandra Dolmeta
Enrico Manfredi

Candidate:
Gabriele De Benedetto

Academic Year 2026

Abstract

The increasing cybersecurity requirements of satellite missions require efficient mechanisms for confidentiality, integrity, and authentication despite the stringent area and power constraints of space-platforms. This work presents the design, implementation and functional verification of a hardware accelerator for CCSDS-recommended AES-GCM, HMAC-SHA-384, and ECDSA over the NIST P-384 curve, integrated into a NOEL-V based SoC. The primary focus is on maximizing hardware resource reuse across the supported cryptographic primitives, to provide a crypto-agile and scalable accelerator core while limiting area overhead. It communicates with the processor with a combination of APB and AHB buses, to separate control and configuration operations from high-bandwidth data transfers. RTL verification and FPGA implementation results demonstrate the functionality and implementation efficiency of the proposed architecture, achieving a promising area reduction compared to the integration of equivalent standalone accelerator modules. Furthermore, the module supports a maximum data transfer rate of more than 1 Gbps for AES-GCM and HMAC-SHA-384 at approximately 100 MHz on a Kintex-7 FPGA, exceeding the throughput requirements of the targeted telemetry, telecommand, and user links. Finally, these results demonstrate that hardware resource sharing can provide an efficient cryptographic subsystem for next-generation satellite platforms under stringent area and power constraints.

Index Terms: Cryptography, resource agility, resource sharing, space communications, hardware accelerator, CCSDS, AES, AES-GCM, CMAC, SHA-2, HMAC, ECDSA, NOEL-V, FPGA.

Contents

List of Tables **VII**

List of Figures **X**

Acronyms **XI**

1 Introduction **1**
1.1 CCSDS algorithm selection 2

2 Background on cryptography algorithms **5**
2.1 AES (Advanced Encryption Standard) 5
2.1.1 GCM (Galois Counter Mode) 9
2.2 CMAC (Cipher-based Message Authentication Code) 13
2.3 SHA-2 (Secure Hash Algorithm) 16
2.4 HMAC (Hash-based Message Authentication Code) 19
2.5 ECDSA (Elliptic Curve Digital Signature Algorithm) 21

3 RTL design **31**
3.1 AES Agile Core 31
3.1.1 AES-256 32
3.1.2 GCM 35
3.1.3 CMAC 38
3.2 SHA Agile Core 41
3.2.1 SHA-384 core 41
3.2.2 HMAC 45
3.2.3 ECDSA P-384 48
3.3 Crypto core and final resource sharing 53
3.3.1 Padder sharing 55
3.3.2 Ports sharing 56
3.4 NOEL-V integration 58
3.4.1 APB/AHB buses 58
3.4.2 Interface Registers 59
3.4.3 DMA 60
3.4.4 Cacheable simulation memory 61
3.4.5 NOEL-V source-tree modifications 62
3.5 Custom API (Application Programming Interface) 62
3.5.1 Register map from the software side 62

3.5.2 Operation sequence 63
3.5.3 Buffer allocation, alignment, and cache coherence 63
3.5.4 Operand encoding 64

4 RTL simulation, FPGA synthesis and results 65
4.1 General testbenches and synthesis 65
4.1.1 Functional testbenches 65
4.1.2 Synthesis 66
4.1.3 Performance benchmarks 66
4.2 Software tests 67
4.2.1 Test file 67
4.2.2 Verification mode result handling 68
4.3 Results 68
4.3.1 Standalone cores versus the state of the art 68
4.3.2 Unified accelerator versus the state of the art 70
4.3.3 Unified accelerator versus `mods_wrapper` 71

5 Conclusions 73
5.1 Future work 73

List of Tables

3.1 Mapping of top-level shared I/O ports to internal submodules. 57
3.2 CTRL register field layout. 59

4.1 Resource utilisation and maximum clock frequency of the standalone cores, the two intermediate shared cores, and the fully integrated accelerator. 66
4.2 Measured performance of all modules. Cycle counts are obtained by RTL simulation which excludes fixed setup cost. Throughput is scaled to the post-route $F_{\max}$ of 125 MHz for the final core or to the obtained $F_{\max}$ for the standalone module. 67
4.3 Comparison with the reference implementations. Throughput is the steady-state rate; setup latency is excluded in all rows. Figures marked † are derived from the cited cycle count and clock frequency, not reported directly by the authors. 68
4.4 Where the sharing occurs. Panel (a) decomposes the total saving by hierarchy level, referenced to the arithmetic sum of the standalone cores. Panel (b) reports the cost of adding the symmetric and hash services on top of an ECDSA-P384 engine, whose modular-arithmetic datapath shares nothing with the other primitives and dominates absolute area. 71

5.1 Post place-and-route comparison of the unified accelerator against the equivalent non-shared union of the four standalone cores (`mods_wrapper`). 73

List of Figures

1.1 Primitive core reuse in the accelerator: a single AES-256 core serves CTR/GCM and CMAC, while a single SHA-384 core serves HMAC and ECDSA. 4

2.1 SubBytes illustration [10] 6
2.2 ShiftRows illustration [10] 7
2.3 MixColumns illustration [10] 8
2.4 AddRoundKey illustration [10] 8
2.5 Illustration of the GCTR function [11] 10
2.6 Illustration of the GHASH function [11] 11
2.7 Illustration of the authenticated encryption function [11] 12
2.8 Illustration of the authenticated decryption function [11] 13
2.9 Illustration of the MAC generation algorithm in CMAC [12] 15
2.10 Illustration of the MAC verification algorithm in CMAC [12] 16
2.11 Illustration of the compression algorithm in SHA-2 [14] 19
2.12 Illustration of the MAC generation algorithm in HMAC [15] 21

3.1 Control unit FSM of the AES-256 core (`cu_aes`). 35
3.2 AES core datapath. The CU is omitted for simplicity, every dashed signal is fed/taken from the CU. Floating signals are CU dedicated, inserted for completeness 35
3.3 Control unit FSM of the AES agile core, for GCM and CMAC flows (`cu_aes_agile`). 38
3.4 AES agile core datapath. The CU is omitted for simplicity, every dashed signal is fed/taken from the CU. Signals with uppercase names are inputs and outputs of the module. Floating signals are CU dedicated, inserted for completeness. 40
3.5 Control unit FSM of the SHA-384 core (`cu_sha`). 44
3.6 SHA core datapath. The CU is omitted for simplicity, every dashed signal is fed/taken from the CU. Floating signals are CU dedicated, inserted for completeness. 45
3.7 Control unit FSM of the HMAC wrapper (`cu_hmac`). 46
3.8 Micro sequencer FSM of the ECDSA controller (`cu_ecdsa`). 50
3.9 Macro-state FSM of the ECDSA controller (`cu_ecdsa`). 51

3.10 SHA agile core datapath. The CU is omitted for simplicity, every dashed signal is fed/taken from the HMAC CU and dotted signals fed/taken from ECDSA CU. Signals with uppercase names are inputs and outputs of the module. Floating signals are CU dedicated, inserted for completeness. The SHA-384 core interface is slightly different from that in Figure 3.6 due to adaptations for the new hardware. 53
3.11 Final agile crypto core datapath. Notice that the colored sub-blocks in gray rectangles are intrinsically connected to each other. The detailed routing is omitted for the sake of readability, knowing that the main engine architectures are illustrated in Figures 3.4 and 3.10. 57
3.12 Crypto core integration scheme in NOEL-V SoC. 64

Acronyms

AAD	Additional Authenticated Data
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BEA	Binary Euclidean Algorithm
BRAM	Block Random-Access Memory
CBC	Cipher Block Chaining
CCSDS	Consultative Committee for Space Data Systems
CMAC	Cipher-based Message Authentication Code
CSA	Carry-Save Adder
CTR	Counter
CU	Control Unit
DEMUX	Demultiplexer
DMA	Direct Memory Access
DRBG	Deterministic Random Bit Generator
DSA	Digital Signature Algorithm
DSP	Digital Signal Processing
DSS	Digital Signature Standard

DUT Design Under Test

ECDLP Elliptic Curve Discrete Logarithm Problem

ECDSA Elliptic Curve Digital Signature Algorithm

FF Flip-Flop

FIFO First-In, First-Out

FinFET Fin Field-Effect Transistor

FIPS Federal Information Processing Standard

FPGA Field Programmable Gate Array

FSM Finite-State Machine

GCM Galois/Counter Mode

GEO Geosynchronous Equatorial Orbit

GF Galois Field

GMAC Galois Message Authentication Code

HMAC Keyed-Hash Message Authentication Code

ICB Initial Counter Block

IP Internet Protocol

IV Initialisation Vector

KAT Known Answer Test

kGE kilo Gate Equivalents

LSB Least Significant Bit

LUT Look-Up Table

MAC Message Authentication Code

MUX Multiplexer

NIST National Institute of Standards and Technology

PE Processing Element

PKCS Public-Key Cryptography Standards

PnP Plug&Play

RAM Random-Access Memory

RIPEMD RACE Integrity Primitives Evaluation Message Digest

ROM Read-Only Memory

RSA Rivest–Shamir–Adleman

RTL Register-Transfer Level

SHA Secure Hash Algorithm

SoC System-On-Chip

SP Special Publication

VHDL VHSIC Hardware Description Language

VLIW Very Long Instruction Word

WNS Worst Negative Slack

XOR Exclusive OR

1 Introduction

Geosynchronous (GEO) satellite links act as the primary means of delivering high-speed Internet Protocol (IP) backhaul to remote critical infrastructure, including utility, military, and commercial networks. However, extensive field scans performed by Zhang *et al.* [1] reveal that 50% of these space communications rely on cleartext transmissions without link-layer or network-layer encryption. The assumption that space flight systems are immune to interception is obsolete, as the high availability of consumer-grade equipment such as standard motorized dishes and television tuners makes eavesdropping and targeted attacks viable. Because these communications are globally broadcast and contain proprietary information, unencrypted cellular backhaul and critical telemetry, the failure to protect transmitted data fundamentally compromises global user privacy and satellite mission success. Mitigating these vulnerabilities requires robust data-protection protocols, which are standardized by the Consultative Committee for Space Data Systems (CCSDS) [2], covering plaintext encryption, user authentication, authenticated encryption and digital signatures.

Enforcing these protocols in flight, however, is a hardware problem more than a specification problem. Modern satellite links push telemetry, telecommand and user traffic in the hundreds of Mbps to Gbps range, while the onboard platforms that must secure them are bounded by tight area, power and radiation budgets and cannot afford the latency of a software-only cryptographic implementation. Furthermore, the CCSDS recommendations do not converge on a single algorithm: a compliant node has to be able to encrypt, authenticate and sign, which means that several primitives must coexist on the same hardware. This combination of throughput, low area and power, and support for multiple services is what makes a crypto-agile hardware accelerator the natural target for onboard security in next generation satellite platforms.

The hardware cryptography literature, however, addresses only fragments of this problem. A first line of work optimizes individual primitives: Lin and Chuang [3] present a high-throughput AES-GCM architecture, Juliato and Gebotys [4] propose a compact HMAC-SHA-2 processor, and Kieu-Do-Nguyen *et al.* [5] implement an area-efficient ECDSA/EdDSA core. These designs are efficient in isolation, but a system that must provide all the required CCSDS algorithms is forced to instantiate one accelerator per algorithm and to replicate the surrounding integration logic (buses, DMA, register files, control), inflating area and power well beyond what the algorithms themselves cost. A second, more recent line of work introduces intra-family resource sharing: Nannipieri *et al.* [6] and the CIPHERX accelerator by Janamanchi *et al.* [7] share hardware across the block-cipher modes of a single symmetric algorithm, while SHARMONY [8] composes two hash primitives from

the SHA-2 and SHA-3 families in a single datapath. These works demonstrate that sharing can be effective within a family, but, to the best of the author's knowledge, no prior work covers a symmetric primitive, a hash primitive and a public-key signature scheme in a single hardware datapath, which is exactly the workload a CCSDS-compliant satellite node has to support.

This thesis addresses that gap. It presents the design, RTL implementation and functional verification of a crypto-agile hardware accelerator that unifies AES-256 CTR/GCM, AES-256 CMAC, HMAC-SHA-384 and ECDSA over the NIST P-384 curve into a single shared datapath, integrated into a NOEL-V based SoC through an APB/AHB interface and a DMA channel. The design maximizes hardware reuse across primitives: a single AES-256 core is shared between the CTR/GCM and CMAC modes, and a single SHA-384 core is shared between HMAC and the message-digest stage of ECDSA. As a result, the five CCSDS-recommended services are supported at a fraction of the area cost of the equivalent non-shared union of standalone cores, while sustaining throughputs above 1 Gbps for AES-GCM and HMAC-SHA-384 at roughly 100 MHz on a Kintex-7 FPGA. The CCSDS Informational Report [9] itself motivates this direction, observing that cipher-based MACs conserve onboard resources and that elliptic-curve signatures cut power, memory and bandwidth in constrained environments. The accelerator described here follows that line of reasoning across all supported primitives.

1.1 CCSDS algorithm selection

The suggested CCSDS cryptographic algorithms for space communications are the following [2]:

- **Symmetric Encryption (Confidentiality):**
  - **Algorithm:** Advanced Encryption Standard (AES).
  - **Key Length:** 256-bit keys for future implementations. 128-bit keys permitted for legacy/existing systems.
  - **Mode of Operation:** Counter (CTR) Mode.
- **Authenticated Encryption (AEAD):**
  - **Algorithm & Mode:** AES in Galois/Counter Mode (AES-GCM).
  - **Tag Length:** 128-bit Message Authentication Code (MAC tag t).
- **Message Authentication and Integrity:**
  - **Hash-Based Authentication:** Keyed-Hash Message Authentication Code (HMAC) using SHA-256 as the default hash function (with SHA-224, SHA-384, SHA-512, or RIPEMD-160 also permitted).
  - **Cipher-Based Authentication:** Cipher-based Message Authentication Code (CMAC) based on AES with 256-bit keys for future missions (128/192/256-bit for existing systems).

  - **Downgraded AEAD Authentication:** Galois Message Authentication Code (GMAC) when repurposing an AES-GCM engine for authentication-only operations.

- **Digital Signatures (Asymmetric Authentication):**

  - **Primary Algorithm:** RSA Digital Signature Algorithm (PKCS #1 v2.1), requiring 4096-bit keys for future implementations (2048-bit allowed for existing systems).
  - **Approved Alternatives:** Elliptic Curve Digital Signature Algorithm (ECDSA) and Digital Signature Algorithm (DSA) compliant with the Digital Signature Standard (DSS).

In this work, AES was selected as a primitive because it is the only encryption standard for all CCSDS missions and ground systems, and a 256-bit key was adopted because the standard requires it for future missions, with 128-bit keys retained only for legacy implementations. Counter mode was chosen because it is the mode explicitly prescribed by the standard. Since the standard also discourages unauthenticated encryption and requires Galois/Counter Mode whenever confidentiality must be combined with data integrity and origin authentication (with a 128-bit tag), the CTR datapath was extended with a GF(2^{128}) multiplier to support GCM. CMAC was then added at marginal cost: the standard designates it as the cipher-based MAC of choice, again with a 256-bit AES key for future implementations, and because CMAC is built on the same block cipher already instantiated for CTR and GCM, it required only the subkey generation and final-block logic rather than a separate primitive. This is the trade-off the Informational Report [9] highlights, noting that a cipher-based MAC conserves onboard resources and simplifies flight-system certification by requiring only a single algorithm for both security services. For digital signatures, the standard recommends RSA with a 4096-bit key for future implementations but explicitly permits other DSS-specified algorithms, including ECDSA. The Informational Report security-strength comparison shows that a P-384 curve delivers 192 bits of security, exceeding the 128 bits provided by RSA-4096, while the smaller key size yields savings in power, memory, bandwidth, and computation, an argument the report itself makes for constrained environments. ECDSA P-384 was therefore preferred to RSA-4096. Finally, since P-384 signatures require a SHA-384 message digest, the same hash engine was reused to implement HMAC. Although the standard names SHA-256 as the normal HMAC variant, it explicitly allows SHA-384 upon prior agreement between communicating entities, so no additional hash datapath was needed. The result is an agile accelerator in which two primitives, AES-256 and SHA-384, cover five services: encryption, authenticated encryption, cipher-based authentication, hash-based authentication, and digital signature, while remaining fully within the CCSDS recommendations.

Ultimately, the selected algorithms are:

- AES-256 CTR/GCM.
- AES-256 CMAC.

- SHA-384 HMAC.
- ECDSA P-384.

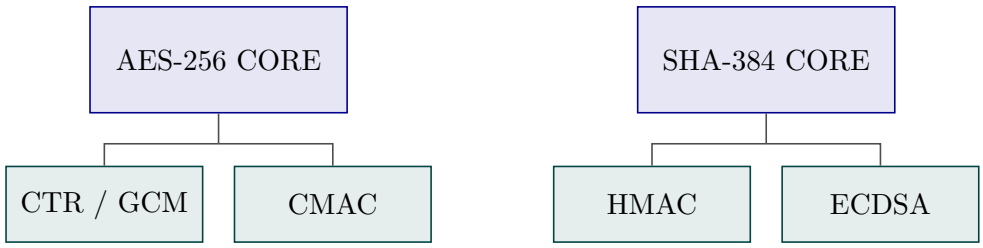


Figure 1.1: Primitive core reuse in the accelerator: a single AES-256 core serves CTR/GCM and CMAC, while a single SHA-384 core serves HMAC and ECDSA.

2 Background on cryptography algorithms

In this chapter, a theoretical background on the chosen cryptographic algorithms is provided.

2.1 AES (Advanced Encryption Standard)

The Advanced Encryption Standard, specified in FIPS PUB 197 [10], is a symmetric block cipher that processes data blocks of 128 bits under a Cipher Key of length 128, 192, or 256 bits. The algorithm is structured as an iterated round function applied to an internal array of bytes called the *State*, and its behaviour is fully determined by three parameters: the block length in 32-bit words, Nb, fixed at 4, the key length in 32-bit words, Nk, taking the values 4, 6, or 8, and the number of rounds Nr, which is 10, 12, or 14 respectively.

Internal State

The internal operations of the AES algorithm are performed on a two-dimensional array of bytes called the State, consisting of four rows and Nb columns. At the start of the Cipher, the 16 bytes of the input block $in_0, in_1, \ldots, in_{15}$ are copied into the State according to the rule

$$s[r, c] = in[r + 4c], \qquad 0 \leq r < 4,\ 0 \leq c < Nb, \tag{2.1}$$

so that the four bytes of each input word populate one column of the State from top to bottom. All round transformations act on this array and at the end of the Cipher the State is copied to the output block using the inverse mapping

$$out[r + 4c] = s[r, c]. \tag{2.2}$$

Each column can be seen as a 32-bit word, allowing the State to be viewed either as a $4 \times Nb$ byte matrix or as an array of Nb words.

Mathematical Preliminaries

All byte-level arithmetic in AES is performed in the finite field $\mathrm{GF}(2^8)$. A byte $\{b_7, b_6, \ldots, b_0\}$ is interpreted as the polynomial

$$b_7x^7 + b_6x^6 + \cdots + b_1x + b_0 \in \mathrm{GF}(2)[x], \tag{2.3}$$

and field operations are defined modulo the irreducible polynomial

$$m(x) = x^8 + x^4 + x^3 + x + 1. \tag{2.4}$$

Addition of two field elements reduces to the bitwise XOR of the corresponding bytes. Multiplication corresponds to polynomial multiplication modulo $m(x)$. Multiplication by the field element $\{02\}$, denoted `xtime`, is implemented as a one-bit left shift followed by a conditional XOR with `0x1B` whenever the most significant bit of the operand is set, and multiplication by higher constants such as $\{03\}$ is obtained by combining `xtime` with additional XORs.

The Cipher

At the start of the Cipher, the input is copied into the State and an initial AddRoundKey is applied, in which the first Round Key is XORed into the State. There follow $Nr - 1$ identical rounds, each consisting of the sequence

$$\textsc{SubBytes} \rightarrow \textsc{ShiftRows} \rightarrow \textsc{MixColumns} \rightarrow \textsc{AddRoundKey}, \tag{2.5}$$

together with a final round from which MixColumns is omitted. The State is then copied to the output block, producing the ciphertext.

SubBytes

SubBytes is a nonlinear byte substitution that operates on each byte of the State independently. The substitution is defined by a fixed table, the S-box, constructed as the composition of two transformations: the multiplicative inverse in GF(2^8), with the convention that $\{00\}$ is mapped to itself, followed by an affine transformation over GF(2). The affine transformation is defined by

$$b'_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i, \tag{2.6}$$

for $0 \leq i < 8$, where b_i denotes the i-th bit of the byte and c_i the i-th bit of the constant byte $\{63\}$. The multiplicative inversion provides the nonlinearity required for resistance to differential and linear cryptanalysis, while the affine layer disrupts the low algebraic degree of the inverse map.

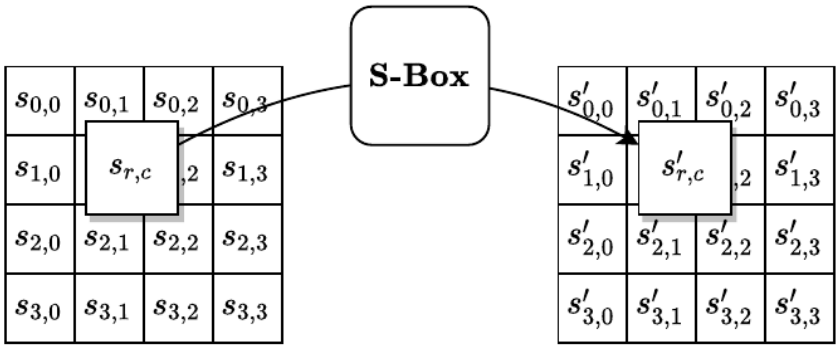


Figure 2.1: SubBytes illustration [10]

ShiftRows

ShiftRows cyclically shifts the last three rows of the State by different offsets. Row 0 is left unchanged, while row r is rotated to the left by r byte positions, for $r = 1, 2, 3$. Since the transformation acts only on the byte positions within each row, it consumes no combinational logic in a hardware implementation, contributing only to interconnect.

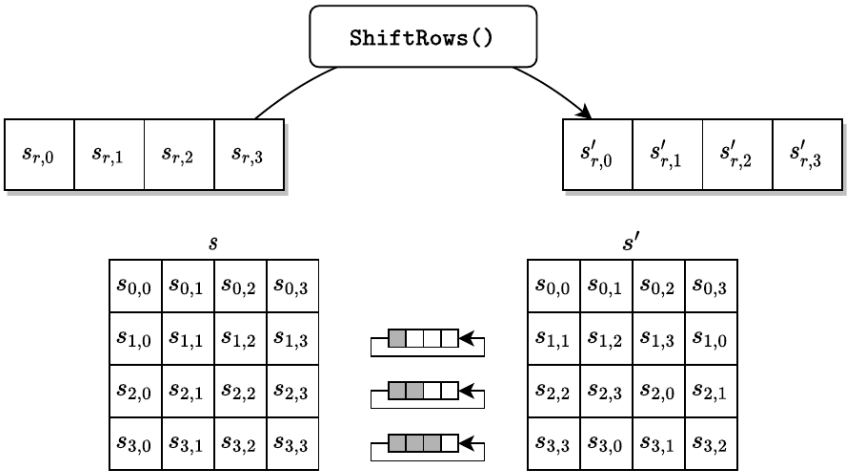


Figure 2.2: ShiftRows illustration [10]

MixColumns

MixColumns operates on the State column by column, treating each column as a four-term polynomial over GF(2^8) and multiplying it, modulo $x^4 + 1$, by the fixed polynomial

$$a(x) = \{03\}\, x^3 + \{01\}\, x^2 + \{01\}\, x + \{02\}. \tag{2.7}$$

Equivalently, each column of the State is replaced by the product of a fixed 4×4 matrix and the original column vector, where matrix–vector multiplication is carried out in GF(2^8). The transformation ensures that any change in a single input byte propagates to all four bytes of the corresponding output column.

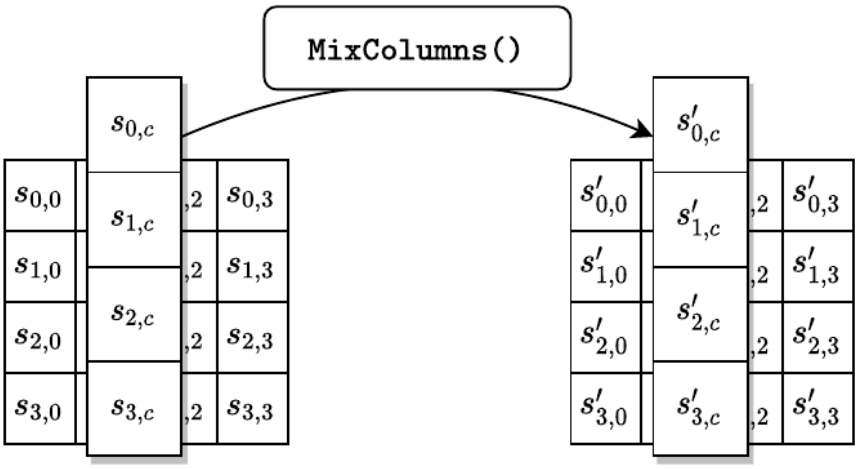


Figure 2.3: MixColumns illustration [10]

AddRoundKey

AddRoundKey adds a Round Key to the State by a bitwise XOR. Each Round Key consists of Nb words drawn in sequence from the expanded key schedule, so that the Round Key applied in round r comprises the words $w[Nb \cdot r]$ through $w[Nb \cdot r + Nb - 1]$. The initial AddRoundKey preceding the first full round applies the words $w[0]$ through $w[Nb - 1]$. AddRoundKey is the only transformation in which key material influences the State, since SubBytes, ShiftRows, and MixColumns are all fixed and independent of the key.

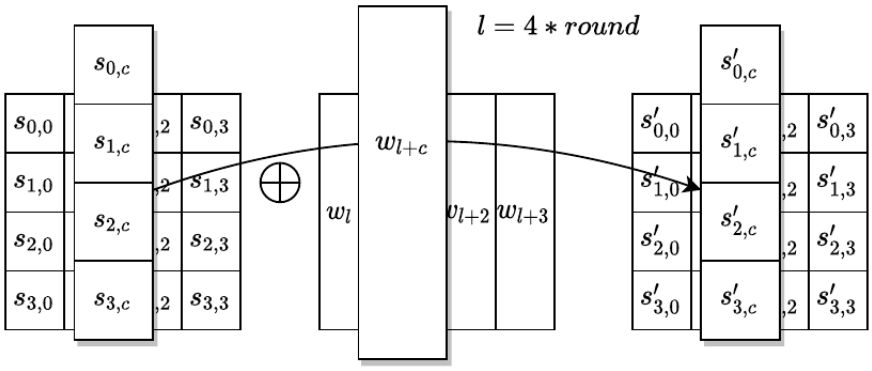


Figure 2.4: AddRoundKey illustration [10]

Key Expansion

The Cipher Key is expanded into a linear array of 32-bit words

$$w[0], w[1], \ldots, w[Nb(Nr + 1) - 1], \tag{2.8}$$

from which the Round Keys are drawn as consecutive groups of Nb words. The first Nk words of the expanded key are the Cipher Key itself. Each subsequent word $w[i]$ is generated by XORing the previous word $w[i - 1]$ with the word Nk positions earlier, $w[i - Nk]$. For those indices at which i is a multiple of Nk, however, the

previous word is first transformed before the XOR: it is cyclically rotated by one byte using the function RotWord, its bytes are then substituted through the S-box using SubWord, and the result is XORed with a round constant $Rcon[i/Nk]$. The round constants are defined as

$$Rcon[j] = (\{02\}^{j-1},\ \{00\},\ \{00\},\ \{00\}), \tag{2.9}$$

where the leading byte is the corresponding power of the field element $\{02\}$ in $\mathrm{GF}(2^8)$.

For $Nk = 8$, corresponding to AES-256, an additional application of SubWord is inserted at positions where $i - 4$ is a multiple of Nk. This supplementary step, present only for the largest key length, injects further nonlinearity into the schedule.

2.1.1 GCM (Galois Counter Mode)

The Galois/Counter Mode of Operation, specified in NIST Special Publication 800-38D [11], is an Authenticated Encryption with Associated Data (AEAD) mode built on top of an approved 128-bit block cipher. GCM provides two services in a single pass over the input: confidentiality of the plaintext P, obtained by a variant of Counter mode called GCTR, and integrity and origin authentication of both the plaintext and a separate string of additional authenticated data (AAD), denoted A, obtained by a universal hash function GHASH that operates in the binary field $\mathrm{GF}(2^{128})$. The mode is parallelisable, pipelineable, and requires the underlying block cipher only in the forward direction.

Inputs, Outputs, and Prerequisites

For a fixed key K and a fixed tag length $t \in \{128, 120, 112, 104, 96\}$, the authenticated encryption function of GCM, GCM-AE$_K$, accepts an initialisation vector IV whose length lies between 1 and $2^{64} - 1$ bits, a plaintext P of length at most $2^{39} - 256$ bits, and additional authenticated data A of length at most $2^{64} - 1$ bits. It returns a ciphertext C of the same bit length as P and an authentication tag T of t bits. The function never emits padding into the ciphertext, so that $|C| = |P|$ exactly.

Mathematical Components

The field $\mathrm{GF}(2^{128})$. GHASH operates on 128-bit blocks interpreted as elements of the finite field $\mathrm{GF}(2^{128})$. A block $x_0 x_1 \dots x_{127}$ is associated with the polynomial

$$x_0 + x_1 u + x_2 u^2 + \cdots + x_{127} u^{127}, \tag{2.10}$$

following the little-endian convention adopted in [11], and all arithmetic is carried out modulo the irreducible polynomial

$$f(u) = 1 + u + u^2 + u^7 + u^{128}. \tag{2.11}$$

Addition in the field reduces to a bitwise XOR of the two 128-bit operands. Multiplication corresponds to polynomial multiplication modulo $f(u)$. A bit-serial description is given by NIST as Algorithm 1 of [11] and uses the fixed reduction constant $R = 11100001 \,\|\, 0^{120}$.

The incrementing function. The counter mechanism of GCM makes use of a function inc_{32} that treats the rightmost 32 bits of a block as an unsigned integer and increments them modulo 2^{32}, leaving the remaining 96 bits unchanged. For any 128-bit block $X = A \,\|\, B$ with $|A| = 96$ and $|B| = 32$,

$$\mathrm{inc}_{32}(X) = A \,\|\, \Big[(\mathrm{int}(B) + 1) \bmod 2^{32}\Big]_{32}, \tag{2.12}$$

where $[n]_{32}$ denotes the 32-bit binary representation of the integer n.

The GCTR Function

The confidentiality primitive of GCM is the function GCTR_K, a variant of Counter mode. Given an initial counter block ICB and an input string X of arbitrary length, the function partitions X into $n = \lceil |X|/128 \rceil$ blocks $X_1, X_2, \ldots, X_{n-1}, X_n^*$, where the final block X_n^* may be shorter than 128 bits. A sequence of counter blocks

$$CB_1 = ICB, \qquad CB_i = \mathrm{inc}_{32}(CB_{i-1}) \text{ for } i \geq 2, \tag{2.13}$$

is generated, each is encrypted with the underlying block cipher, and the resulting keystream is XORed into the corresponding block of the input:

$$Y_i = X_i \oplus \mathrm{CIPH}_K(CB_i), \quad 1 \leq i \leq n-1, \tag{2.14}$$

$$Y_n^* = X_n^* \oplus \mathrm{MSB}_{|X_n^*|}\Big(\mathrm{CIPH}_K(CB_n)\Big). \tag{2.15}$$

The output is the concatenation $\mathrm{GCTR}_K(ICB, X) = Y_1 \,\|\, Y_2 \,\|\, \ldots \,\|\, Y_n^*$. No padding is introduced, since the final keystream block is truncated to the length of X_n^*.

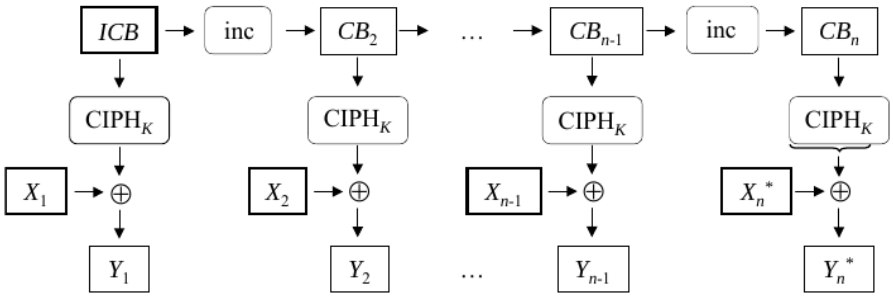


Figure 2.5: Illustration of the GCTR function [11]

The GHASH Function

The authentication primitive of GCM is the keyed hash function GHASH_H with a 128-bit *hash subkey* H. Given an input X whose length is a positive multiple of 128 bits, X is partitioned into m blocks $X_1, X_2, \ldots, X_m$, and the function evaluates the recurrence

$$Y_0 = 0^{128}, \qquad Y_i = (Y_{i-1} \oplus X_i) \bullet H, \quad 1 \leq i \leq m, \tag{2.16}$$

returning Y_m. Expanding the recurrence yields the equivalent closed form

$$\mathrm{GHASH}_H(X) = X_1 \bullet H^m \oplus X_2 \bullet H^{m-1} \oplus \cdots \oplus X_m \bullet H, \tag{2.17}$$

which exposes the parallelism that hardware implementations exploit by precomputing powers of H and combining several partial products per cycle.

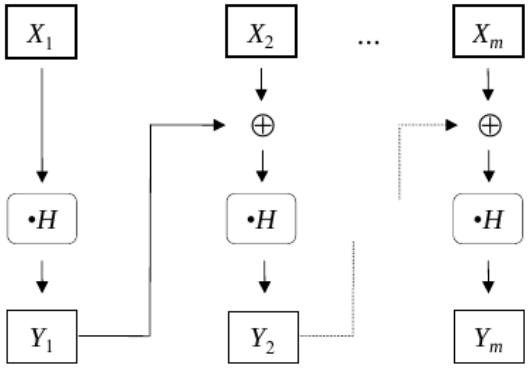


Figure 2.6: Illustration of the GHASH function [11]

Authenticated Encryption

Algorithm 4 of [11] specifies the authenticated encryption function $\mathrm{GCM\text{-}AE}_K(IV, P, A) = (C, T)$ through the following steps.

1. Derive the hash subkey by encrypting the zero block:

$$H = \mathrm{CIPH}_K(0^{128}). \tag{2.18}$$

2. Construct the pre-counter block J_0 from the IV. If $|IV| = 96$, then

$$J_0 = IV \,\|\, 0^{31} \,\|\, 1; \tag{2.19}$$

   otherwise, letting $s = 128\,\lceil |IV|/128 \rceil - |IV|$,

$$J_0 = \mathrm{GHASH}_H\Big(IV \,\|\, 0^{s+64} \,\|\, [|IV|]_{64}\Big). \tag{2.20}$$

3. Encrypt the plaintext by invoking GCTR with the counter initialised to $\mathrm{inc}_{32}(J_0)$:

$$C = \mathrm{GCTR}_K\Big(\mathrm{inc}_{32}(J_0),\, P\Big). \tag{2.21}$$

4. Compute the numbers of zero bits required to align A and C to the block boundary:

$$v = 128\,\lceil |A|/128 \rceil - |A|, \quad u = 128\,\lceil |C|/128 \rceil - |C|. \tag{2.22}$$

5. Assemble the input to GHASH by concatenating the padded AAD, the padded ciphertext, and the 64-bit length encodings, and evaluate

$$S = \mathrm{GHASH}_H\Big(A \,\|\, 0^v \,\|\, C \,\|\, 0^u \,\|\, [|A|]_{64} \,\|\, [|C|]_{64}\Big). \tag{2.23}$$

6. Produce the authentication tag by encrypting S under the pre-counter block J_0 and truncating the result:

$$T = \mathrm{MSB}_t\Big(\mathrm{GCTR}_K(J_0, S)\Big). \tag{2.24}$$

7. Return (C, T).

Two structural observations follow directly from Equations (2.18),(2.24): both the hash subkey H and the tag mask $\mathrm{CIPH}_K(J_0)$ are produced by the same forward AES invocation used in GCTR, so a single encryption datapath works for all four cipher calls; no inverse cipher is required. Moreover, the sequence of counter blocks used to encrypt P begins at $\mathrm{inc}_{32}(J_0)$ rather than at J_0, which guarantees that the block used to mask the tag can never coincide with a block used to mask the plaintext.

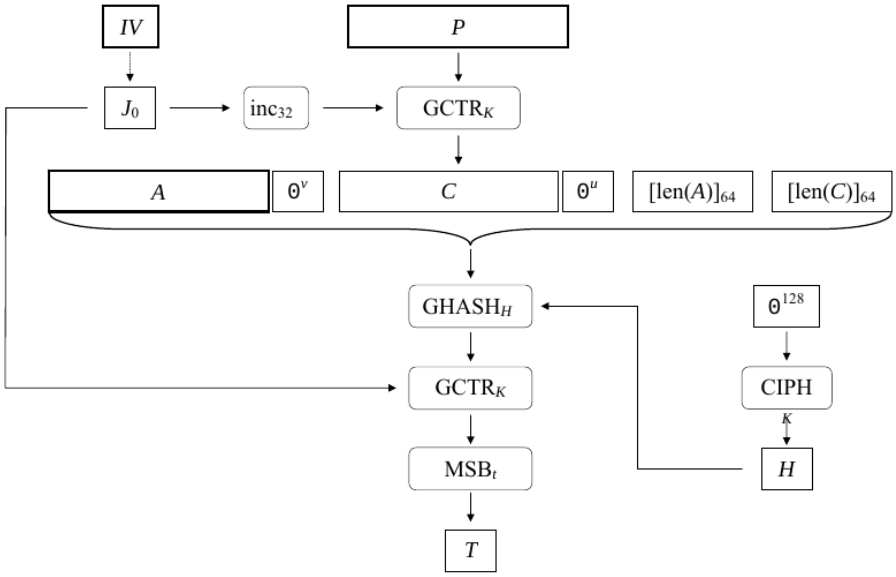


Figure 2.7: Illustration of the authenticated encryption function [11]

Authenticated Decryption

The authenticated decryption function $\text{GCM-AD}_K(IV, C, A, T)$, specified as Algorithm 5 of [11], follows the same construction: the hash subkey H and the pre-counter block J_0 are derived exactly as in Equations (2.18),(2.20), the plaintext is recovered as $P = \mathrm{GCTR}_K(\mathrm{inc}_{32}(J_0), C)$, and a candidate tag T' is computed from A and C following Equations (2.23),(2.24). The function returns P if $T' = T$ and the distinguished symbol FAIL otherwise. Since GCTR is symmetric in encryption and decryption and GHASH operates on the ciphertext in both directions, no additional primitive is required.

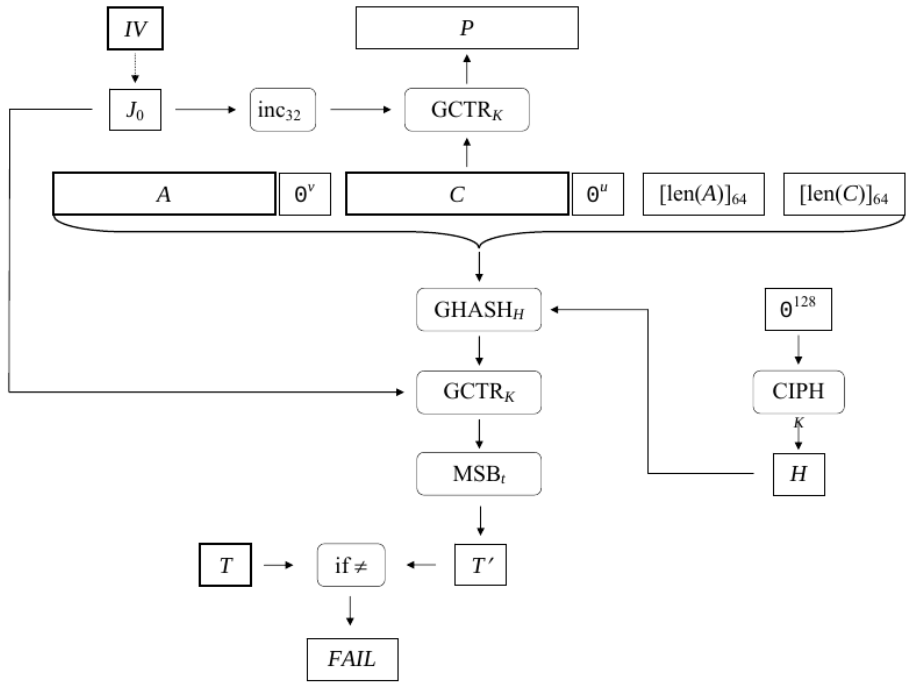


Figure 2.8: Illustration of the authenticated decryption function [11]

Uniqueness Requirement on Initialisation Vectors

The security guarantees of GCM are conditional on the uniqueness of the initialisation vector: for any fixed key K, no two invocations of GCM-AE$_K$ shall use the same IV. A repeated IV under the same key permits recovery of the hash subkey H and, consequently, forgery of authentication tags on arbitrary messages [11]. The 96-bit IV length is distinguished by the standard because it allows the pre-counter block to be formed by direct concatenation, as in Equation (2.19), without invoking GHASH.

2.2 CMAC (Cipher-based Message Authentication Code)

The Cipher-based Message Authentication Code, specified in NIST Special Publication 800-38B [12], is a mode of operation of a symmetric-key block cipher that produces a fixed-length authentication tag on a message of arbitrary bit length.

Inputs, Outputs, and Prerequisites

For a fixed key K and a fixed tag length $Tlen$, the MAC generation function of CMAC accepts a message M of arbitrary bit length $Mlen \geq 0$ and returns a MAC T of $Tlen$ bits, with $Tlen$ constrained to lie between 1 and the block size b of the cipher.

The Subkey Constant R_b

The subkey generation process uses a fixed constant R_b determined by the block size b. The bit string R_b is a representation of the first irreducible binary polynomial of degree b with the minimum possible number of nonzero terms. This polynomial is written as

$$u^b + c_{b-1}u^{b-1} + \cdots + c_1 u + c_0, \qquad c_i \in \{0, 1\}, \tag{2.25}$$

then $R_b = c_{b-1}c_{b-2}\ldots c_1 c_0$. For $b = 128$, the standard specifies

$$R_{128} = 0^{120} \,\|\, 10000111, \tag{2.26}$$

corresponding to the polynomial $u^{128} + u^7 + u^2 + u + 1$.

Subkey Generation

Two secret subkeys K_1 and K_2, each of length b bits, are derived from the block cipher key by encrypting the zero block and applying a conditional left shift.

1. Compute
$$L = \mathrm{CIPH}_K(0^b). \tag{2.27}$$

2. Derive the first subkey as
$$K_1 = \begin{cases} L \ll 1, & \text{if } \mathrm{MSB}_1(L) = 0, \\ (L \ll 1) \oplus R_b, & \text{if } \mathrm{MSB}_1(L) = 1, \end{cases} \tag{2.28}$$
where $X \ll 1$ denotes the bitwise left shift.

3. Derive the second subkey from K_1 by the same rule:
$$K_2 = \begin{cases} K_1 \ll 1, & \text{if } \mathrm{MSB}_1(K_1) = 0, \\ (K_1 \ll 1) \oplus R_b, & \text{if } \mathrm{MSB}_1(K_1) = 1. \end{cases} \tag{2.29}$$

The operations in Equations (2.28),(2.29) correspond to multiplication of L by the field elements u and u^2 respectively, within the binary field $\mathrm{GF}(2^b)$ whose reduction polynomial is represented by R_b. Both subkeys, as well as the intermediate value L, are required by the standard to remain secret and shall not be exposed to any other function of the implementation. Since K_1 and K_2 depend only on K, they may be precomputed once per key and stored alongside the key material, or generated at the start of each session by a single additional AES invocation followed by two conditional shift-XOR steps.

MAC Generation

Given the subkeys K_1 and K_2, the MAC generation function $\mathrm{CMAC}(K, M, \mathit{Tlen})$ is defined by the following steps.

1. Determine the number of blocks in the partition:

$$n = \begin{cases} 1, & \text{if } Mlen = 0, \\ \lceil Mlen/b \rceil, & \text{if } Mlen > 0. \end{cases} \tag{2.30}$$

2. Partition the message as

$$M = M_1 \,\|\, M_2 \,\|\, \ldots \,\|\, M_{n-1} \,\|\, M_n^*, \tag{2.31}$$

where $M_1, \ldots, M_{n-1}$ are complete b-bit blocks and M_n^* has bit length between 1 and b; if $Mlen = 0$, then M_n^* is the empty string.

3. Mask the final block, distinguishing two cases:

$$M_n = \begin{cases} K_1 \oplus M_n^*, & \text{if } |M_n^*| = b, \\ K_2 \oplus \left(M_n^* \,\|\, 1 \,\|\, 0^j\right), & \text{if } |M_n^*| < b, \end{cases} \tag{2.32}$$

with $j = nb - Mlen - 1$. In the first case the final block is already complete and is masked with the first subkey. In the second case the incomplete final block is padded with a single 1 bit followed by the minimum number of zero bits required to reach a full block length, and the padded block is masked with the second subkey.

4. Apply Cipher Block Chaining with the zero block as the initialisation vector:

$$C_0 = 0^b, \qquad C_i = \mathrm{CIPH}_K(C_{i-1} \oplus M_i), \quad 1 \le i \le n. \tag{2.33}$$

5. Return the MAC by truncating the final ciphertext block:

$$T = \mathrm{MSB}_{Tlen}(C_n). \tag{2.34}$$

The two-case masking of the final block in Equation (2.32) is what distinguishes CMAC from plain CBC-MAC. Because K_1 and K_2 are distinct with overwhelming probability, an attacker cannot append data to a legitimate message and obtain the MAC of the extended message from that of the original, which is the length-extension attack to which CBC-MAC is susceptible when the message length is not encoded in the input.

The formatting and chaining steps may be interleaved, so that each block is XORed with the previous ciphertext and encrypted as soon as it becomes available.

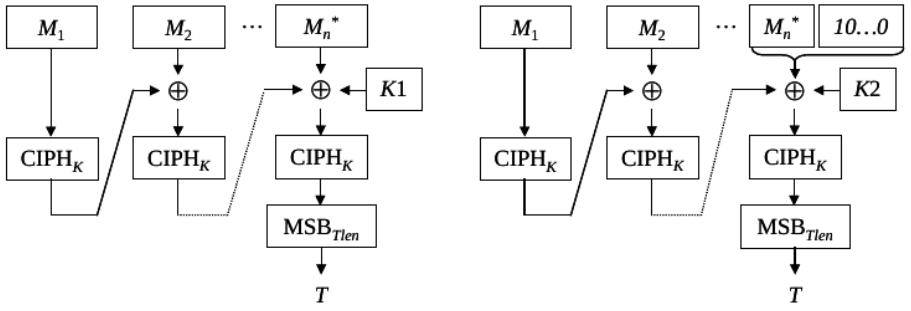


Figure 2.9: Illustration of the MAC generation algorithm in CMAC [12]

MAC Verification

The MAC verification function accepts a message M and a tag T' of length $Tlen$, recomputes the MAC T by applying the generation function to M under the same key K, and returns VALID if $T' = T$ and INVALID otherwise. No inverse cipher operation is required, since verification, like generation, uses CIPH_K exclusively in the forward direction.

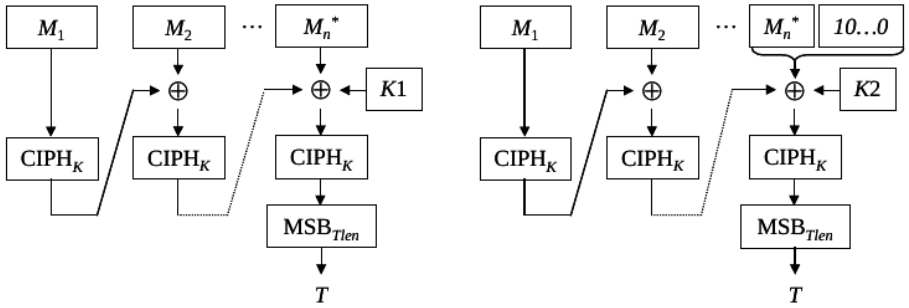


Figure 2.10: Illustration of the MAC verification algorithm in CMAC [12]

2.3 SHA-2 (Secure Hash Algorithm)

The Secure Hash Standard, specified in FIPS PUB 180-4 [13], defines a family of iterative, one-way hash functions collectively referred to as SHA-2, comprising the variants SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256. All variants share the same structure: the input message is padded and parsed into fixed-length blocks, a compression function is applied iteratively to each block starting from a variant-specific initialisation vector, and the final chaining value is truncated to the required digest length. The variants are partitioned into two groups by word size: SHA-224 and SHA-256 operate on 32-bit words with a 512-bit block, while SHA-384, SHA-512, and their truncated derivatives operate on 64-bit words with a 1024-bit block. Because SHA-384 is defined as SHA-512 with a different initial hash value and a truncated output, the description that follows covers the 64-bit family and specialises to SHA-384 at the end.

Word Operations and Notation

All arithmetic within the 64-bit family is performed on 64-bit words, with modular addition denoted by $+$ and taken modulo 2^{64}. The bitwise Boolean operators $\wedge$, $\vee$, $\oplus$, and $\neg$ act component-wise on words. The rotation by n positions to the right and the logical right shift by n positions are denoted respectively by

$$\text{ROTR}^n(x) = (x \gg n) \vee (x \ll (64 - n)), \qquad \text{SHR}^n(x) = x \gg n, \tag{2.35}$$

for $0 < n < 64$. The concatenation of two bit strings is written $X \,\|\, Y$.

Functions and Constants

The compression function of the 64-bit SHA-2 family employs six fixed logical functions of three 64-bit words. Two of these are the standard bitwise choice and

majority functions:

$$\mathrm{Ch}(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z), \tag{2.36}$$

$$\mathrm{Maj}(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z). \tag{2.37}$$

The remaining four are rotation-based diffusion functions of a single word: the two *capital sigma* functions

$$\Sigma_0^{\{512\}}(x) = \mathrm{ROTR}^{28}(x) \oplus \mathrm{ROTR}^{34}(x) \oplus \mathrm{ROTR}^{39}(x), \tag{2.38}$$

$$\Sigma_1^{\{512\}}(x) = \mathrm{ROTR}^{14}(x) \oplus \mathrm{ROTR}^{18}(x) \oplus \mathrm{ROTR}^{41}(x), \tag{2.39}$$

which act on the working variables during the round update, and the two *lowercase sigma* functions

$$\sigma_0^{\{512\}}(x) = \mathrm{ROTR}^{1}(x) \oplus \mathrm{ROTR}^{8}(x) \oplus \mathrm{SHR}^{7}(x), \tag{2.40}$$

$$\sigma_1^{\{512\}}(x) = \mathrm{ROTR}^{19}(x) \oplus \mathrm{ROTR}^{61}(x) \oplus \mathrm{SHR}^{6}(x), \tag{2.41}$$

which drive the expansion of the message schedule.

The compression function further employs a sequence of eighty 64-bit round constants $K_0^{\{512\}}, K_1^{\{512\}}, \ldots, K_{79}^{\{512\}}$, defined by the standard as the first 64 bits of the fractional parts of the cube roots of the first eighty prime numbers [13]. The complete numerical values are tabulated in Section 4.2.3 of FIPS 180-4 and are treated as fixed inputs to the implementation.

Preprocessing

A message M of bit length ℓ, with $0 \leq \ell < 2^{128}$, is first padded so that its total length becomes a multiple of the block size. Padding consists of appending a single 1 bit, followed by k zero bits, followed by the 128-bit binary representation of ℓ, where k is the smallest non-negative integer satisfying

$$\ell + 1 + k \equiv 896 \pmod{1024}. \tag{2.42}$$

The padded message is then parsed into N blocks $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$, each of 1024 bits, and every block $M^{(i)}$ is regarded as a sequence of sixteen 64-bit words $M_0^{(i)}, M_1^{(i)}, \ldots, M_{15}^{(i)}$, in big-endian order.

The initial hash value $H^{(0)}$ consists of eight 64-bit words $H_0^{(0)}, H_1^{(0)}, \ldots, H_7^{(0)}$, whose specific values depend on the chosen variant.

Hash Computation

For each message block $M^{(i)}$, with $i = 1, 2, \ldots, N$, the compression function is evaluated in three phases: expansion of the message schedule, iteration of the round function on the working variables, and update of the chaining value.

Message schedule. A sequence of eighty 64-bit schedule words $W_0, W_1, \ldots, W_{79}$ is generated from the current block by

$$W_t = \begin{cases} M_t^{(i)}, & 0 \le t \le 15, \\ \sigma_1^{\{512\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{512\}}(W_{t-15}) + W_{t-16}, & 16 \le t \le 79. \end{cases} \tag{2.43}$$

The first sixteen words are the message-block words themselves. The remaining 64 are produced by the linear recurrence in Equation (2.43), which mixes earlier schedule words through the σ_0 and σ_1 functions and modular addition.

Working variables. Eight 64-bit working variables a, b, c, d, e, f, g, h are initialised from the $(i-1)$-th intermediate hash value:

$$(a, b, c, d, e, f, g, h) \leftarrow \left(H_0^{(i-1)}, H_1^{(i-1)}, H_2^{(i-1)}, H_3^{(i-1)}, H_4^{(i-1)}, H_5^{(i-1)}, H_6^{(i-1)}, H_7^{(i-1)}\right). \tag{2.44}$$

Round function. For $t = 0, 1, \ldots, 79$, two temporary 64-bit words are formed as

$$T_1 = h + \Sigma_1^{\{512\}}(e) + \mathrm{Ch}(e, f, g) + K_t^{\{512\}} + W_t, \tag{2.45}$$

$$T_2 = \Sigma_0^{\{512\}}(a) + \mathrm{Maj}(a, b, c), \tag{2.46}$$

and the working variables are shifted down by one position, with T_1 injected into the state through the update

$$(a, b, c, d, e, f, g, h) \leftarrow (T_1 + T_2,\ a,\ b,\ c,\ d + T_1,\ e,\ f,\ g). \tag{2.47}$$

Equations (2.45),(2.47) implement one of the eighty rounds. The message dependent value enters exclusively through W_t, while the constant $K_t^{\{512\}}$ ensures that no two rounds apply the same transformation.

Chaining update. After the 80^{th} round has been executed, the i-th intermediate hash value is obtained by adding the final working variables into the previous chaining values, modulo 2^{64}:

$$H_k^{(i)} = a_k^{\text{final}} + H_k^{(i-1)}, \qquad k = 0, 1, \ldots, 7, \tag{2.48}$$

where $(a_0^{\text{final}}, \ldots, a_7^{\text{final}})$ denotes the tuple (a, b, c, d, e, f, g, h) after Equation (2.47) has been applied for the last time. This addition is what renders the compression function one-way: without it, the entire round sequence would be a bijection on the working variables and hence invertible from any known output.

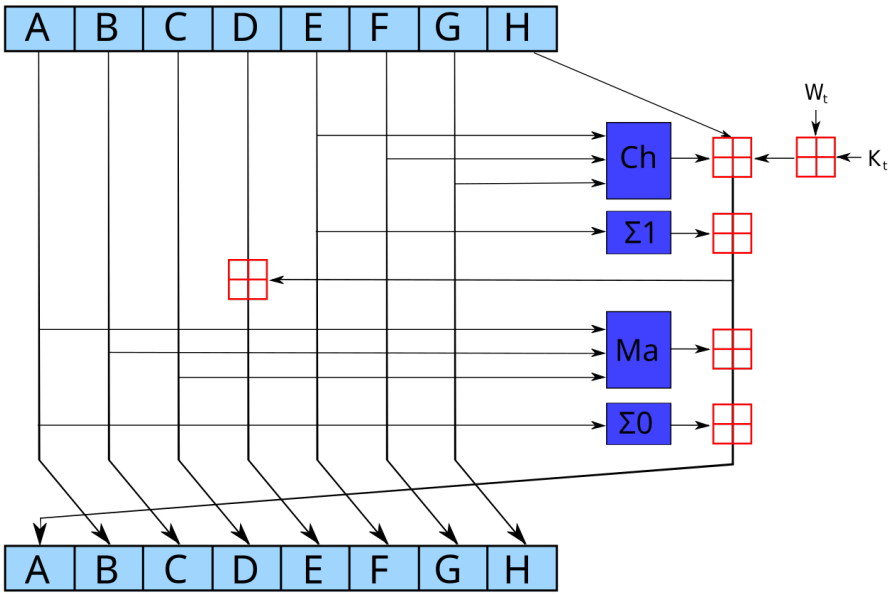


Figure 2.11: Illustration of the compression algorithm in SHA-2 [14]

SHA-512 and SHA-384 Digests

After all N message blocks have been processed, the final chaining value $H^{(N)}$ is a sequence of eight 64-bit words, from which the digest is formed. For SHA-512, the digest is the full concatenation

$$\text{SHA-512}(M) = H_0^{(N)} \parallel H_1^{(N)} \parallel \ldots \parallel H_7^{(N)}, \tag{2.49}$$

yielding a 512-bit output. For SHA-384, the algorithm is identical except for the choice of $H^{(0)}$ noted above and the truncation of the output to its six leftmost words:

$$\text{SHA-384}(M) = H_0^{(N)} \parallel H_1^{(N)} \parallel H_2^{(N)} \parallel H_3^{(N)} \parallel H_4^{(N)} \parallel H_5^{(N)}, \tag{2.50}$$

yielding a 384-bit output. The remaining two words $H_6^{(N)}, H_7^{(N)}$ are discarded. Their presence in the internal state during processing, however, is what allows SHA-384 to keep a similar collision resistance of the full 512-bit compression function while emitting a shorter digest.

2.4 HMAC (Hash-based Message Authentication Code)

The Keyed-Hash Message Authentication Code, specified in NIST Special Publication 800-224 [15], is a message authentication algorithm constructed on top of a cryptographic hash function.

Inputs, Outputs, and Prerequisites

Let H be an approved cryptographic hash function with output length ℓ bits and block length b bits, where b is a multiple of eight and satisfies $b \geq \ell$. NIST SP 800-224 [15] lists as approved for HMAC the SHA-2 variants (SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256) and the SHA-3 variants (SHA3-224, SHA3-256, SHA3-384, SHA3-512). For the SHA-384 instantiation adopted here, $\ell = 384$ and $b = 1024$.

The tag generation function accepts a secret key K and a message M of arbitrary bit length and returns a tag T of ℓ bits. The key length shall be at least 128 bits, and NIST SP 800-224 recommends against the use of keys longer than the block size b because any additional key entropy beyond b bits is discarded by the key processing step. When truncation is used, the output tag has bit length $\lambda \leq \ell$.

The Inner and Outer Pads

HMAC employs two fixed b-bit constants that separate the two hash invocations. The *inner pad* ipad is the concatenation of $b/8$ copies of the byte `0x36`, and the *outer pad* opad is the concatenation of $b/8$ copies of the byte `0x5C`:

$$\text{ipad} = \underbrace{\texttt{0x36} \,\|\, \texttt{0x36} \,\|\, \ldots \,\|\, \texttt{0x36}}_{b/8 \text{ bytes}}, \qquad \text{opad} = \underbrace{\texttt{0x5C} \,\|\, \texttt{0x5C} \,\|\, \ldots \,\|\, \texttt{0x5C}}_{b/8 \text{ bytes}}. \tag{2.51}$$

The two constants differ in every byte position, so that $\text{ipad} \oplus \text{opad} = (01101010)^{b/8}$ is a fixed non-zero string with Hamming weight $4b/8$: this separation is what allows the security proof of HMAC to reduce forgery attacks.

Key Processing

The secret key K is first transformed into a b-bit *prepared key* K_0 by one of three procedures, depending on the relationship between $\text{len}(K)$ and b:

$$K_0 = \begin{cases} K, & \text{if } \text{len}(K) = b, \\ K \,\|\, 0^{\,b-\text{len}(K)}, & \text{if } \text{len}(K) < b, \\ H(K) \,\|\, 0^{\,b-\ell}, & \text{if } \text{len}(K) > b. \end{cases} \tag{2.52}$$

A key shorter than the block size is right-padded with zeros. A key longer than the block size is first hashed and the resulting ℓ-bit digest is right-padded with zeros. A key exactly b bits long is not modified. In all three cases the prepared key has exactly b bits and can therefore be XORed with the pads defined by Equation (2.51).

The requirement that long keys be pre-hashed introduces a security subtlety noted in NIST SP 800-224: the effective key entropy of a long-key HMAC is bounded by ℓ rather than by $\text{len}(K)$, since the adversary can attempt key recovery either on K directly or on the intermediate digest $H(K)$. This is the technical reason for the recommendation that HMAC keys not exceed the block size.

Tag Generation

Given the prepared key K_0, the HMAC tag is computed by the two nested hash invocations

$$T = \mathrm{HMAC}(K, M) = H\Big((K_0 \oplus \mathrm{opad}) \,\|\, H\big((K_0 \oplus \mathrm{ipad}) \,\|\, M\big)\Big). \tag{2.53}$$

The construction proceeds in two stages. In the first, the *inner hash*

$$T_{\mathrm{inner}} = H\big((K_0 \oplus \mathrm{ipad}) \,\|\, M\big) \tag{2.54}$$

is computed over the concatenation of the inner-padded key with the message, producing an ℓ-bit intermediate digest. In the second, the *outer hash*

$$T = H\big((K_0 \oplus \mathrm{opad}) \,\|\, T_{\mathrm{inner}}\big) \tag{2.55}$$

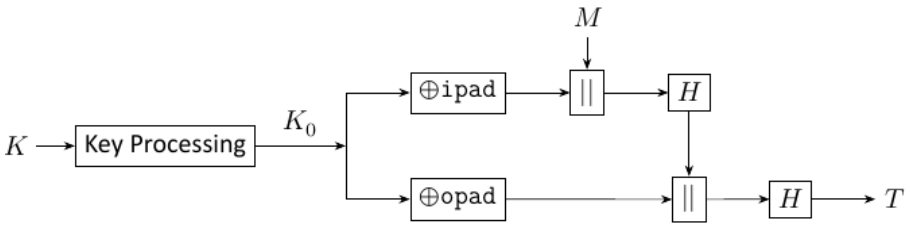


Figure 2.12: Illustration of the MAC generation algorithm in HMAC [15]

2.5 ECDSA (Elliptic Curve Digital Signature Algorithm)

The Elliptic Curve Digital Signature Algorithm (ECDSA) is the elliptic curve analogue of the Digital Signature Algorithm (DSA). Standardized in FIPS PUB 186-5 [16], ECDSA provides data origin authentication, data integrity, and signatory non-repudiation. The security of ECDSA relies directly on the intractability of the Elliptic Curve Discrete Logarithm Problem (ECDLP) over finite fields. Given an elliptic curve group and a point $Q = [d]G$, computing the scalar private key d from the public point Q and generator G is computationally infeasible for large group orders. Hence ECDSA offers security comparable to classical schemes such as RSA but with much shorter key lengths, thus decreasing computational overhead and hardware storage.

Mathematical Foundation and NIST P-384 Domain Parameters

An elliptic curve over a prime Galois field $\mathbb{F}_p$ is defined using the short Weierstrass equation:

$$E(\mathbb{F}_p) : y^2 \equiv x^3 + ax + b \pmod{p}, \tag{2.56}$$

where the field elements $a, b \in \mathbb{F}_p$ satisfy the non-singularity condition $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$. The elliptic curve domain parameters $D = (p, a, b, G, n, h)$ uniquely specify the cryptographic group:

- **Prime Modulus (p).** The order of the underlying finite field $\mathbb{F}_p$. For NIST P-384 (also designated as `secp384r1`), p is a 384-bit generalised Mersenne prime:

$$p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1. \tag{2.57}$$

  This algebraic structure enables highly optimised modular reduction using word-level shift-and-add operations without general division.

- **Fast reduction modulo p.** Because p is a generalised Mersenne prime, the modular reduction of a 768-bit product $c = \sum_{i=0}^{23} c_i\, 2^{32i}$ into $\mathbb{F}_p$ can be carried out without division. Writing the twelve high-order contributions $c_{12}, c_{13}, \ldots, c_{23}$ in terms of powers of 2 modulo p and rearranging yields the identity given by NIST SP 800-186, in which the reduced representative is expressed as a signed sum of nine copies of the low-order limbs of c, aligned at 32-bit word boundaries. The reduction therefore reduces to a small number of 384-bit additions and subtractions, followed by a final conditional subtraction of p to bring the result into $[0, p)$. This structure is what makes P-384 arithmetic tractable in hardware without a general-purpose modular divider.

- **Curve Coefficients (a, b).** The curve parameter a is fixed to

$$a \equiv -3 \pmod{p}, \tag{2.58}$$

  which admits the factorisation $3x_1^2 + a = 3\,(x_1 - 1)(x_1 + 1)$ in the numerator of the doubling slope, saving one field multiplication per point doubling in appropriate coordinate systems. The constant b is a pseudo-random 384-bit integer:

$$b = \texttt{0xb3312fa7 e23ee7e4 988e056b e3f82d19 181d9c6e fe814112} \\ \texttt{0314088f 5013875a c656398d 8a2ed19d 2a85c8ed d3ec2aef}. \tag{2.59}$$

- **Base Point (G).** A fixed base generator point $G = (x_G, y_G) \in E(\mathbb{F}_p)$ generating a prime-order subgroup, with affine coordinates specified by FIPS 186-5 as

$$x_G = \texttt{0xaa87ca22 be8b0537 8eb1c71e f320ad74 6e1d3b62 8ba79b98} \\ \texttt{59f741e0 82542a38 5502f25d bf55296c 3a545e38 72760ab7}, \tag{2.60}$$

$$y_G = \texttt{0x3617de4a 96262c6f 5d9e98bf 9292dc29 f8f41dbd 289a147c} \\ \texttt{e9da3113 b5f0b8c0 0a60b1ce 1d7e819d 7a431d7c 90ea0e5f}. \tag{2.61}$$

- **Subgroup Order (n).** The prime order of the subgroup generated by G, such that $[n]G = \mathcal{O}$, where $\mathcal{O}$ denotes the point at infinity:

$$n = \texttt{0xffffffff ffffffff ffffffff ffffffff ffffffff ffffffff} \\ \texttt{c7634d81 f4372ddf 581a0db2 48b0a77a ecec196a ccc52973}. \tag{2.62}$$

- **Cofactor (h).** The cofactor $h = \#E(\mathbb{F}_p)/n = 1$, ensuring that all valid non-infinity points on the curve belong strictly to the cryptographic group generated by G.

Group Operations on $E(\mathbb{F}_p)$

The elliptic curve $E(\mathbb{F}_p)$ together with the point at infinity $\mathcal{O}$ forms an abelian group. For two affine points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ on $E(\mathbb{F}_p)$ with $P_1 \neq \pm P_2$, the sum $P_3 = P_1 + P_2 = (x_3, y_3)$ is given by

$$\lambda = (y_2 - y_1)(x_2 - x_1)^{-1} \bmod p, \qquad x_3 = \lambda^2 - x_1 - x_2, \qquad y_3 = \lambda(x_1 - x_3) - y_1. \tag{2.63}$$

The doubling of a point $P_1 = (x_1, y_1)$ with $y_1 \neq 0$ yields $2P_1 = (x_3, y_3)$ with

$$\lambda = (3x_1^2 + a)(2y_1)^{-1} \bmod p, \qquad x_3 = \lambda^2 - 2x_1, \qquad y_3 = \lambda(x_1 - x_3) - y_1. \tag{2.64}$$

Equations (2.63),(2.64) each require one modular inversion in $\mathbb{F}_p$, an operation whose cost dominates that of multiplication. Practical implementations represent points in Jacobian projective coordinates (X, Y, Z), where the affine coordinates are recovered as $(x, y) = (X/Z^2,\ Y/Z^3)$: addition and doubling become inversion-free at the cost of additional multiplications and squarings, and a single inversion is needed for a conversion back to affine form.

Scalar multiplication $[k]P$, the operation that dominates both signature generation and verification, is then evaluated by iterating point doublings and additions according to the binary expansion of the scalar.

Modular Multiplication

Point addition and doubling on $E(\mathbb{F}_p)$, and hence scalar multiplication $[k]G$, reduce entirely to a sequence of additions, subtractions, multiplications, and inversions in the prime field $\mathbb{F}_p$. Of these, multiplication and inversion dominate both the operation count and the arithmetic cost. The design of an efficient $\mathbb{F}_p$-multiplication algorithm is therefore the first step toward an efficient ECDSA implementation.

The algorithm adopted in this work computes the product $c = a \cdot b \bmod p$ by the classical right-to-left binary method, described in [17, Algorithm 6] and specialised here to the multiplicative structure of $\mathbb{F}_p$. Writing the multiplier in binary form as

$$b = \sum_{i=0}^{n-1} b_i \, 2^i, \qquad b_i \in \{0, 1\}, \tag{2.65}$$

with $n = \lceil \log_2 p \rceil$, the product $a \cdot b$ can be expressed as

$$a \cdot b = \sum_{i=0}^{n-1} b_i \cdot \left(2^i \, a\right), \tag{2.66}$$

so that the full product is obtained by accumulating those multiples $2^i \, a$ for which the corresponding bit of b is set. Reducing each intermediate quantity modulo p as soon as it exceeds the modulus, rather than at the end, keeps every operand within

$[0, p-1]$ and confines every field operation to the same fixed n-bit width. This yields the recurrence

$$c_0 = 0, \qquad a_0 = a, \qquad c_{i+1} = c_i + b_i \cdot a_i \bmod p, \qquad a_{i+1} = 2\,a_i \bmod p, \tag{2.67}$$

whose terminal value c_n equals $a \cdot b \bmod p$. Algorithm 1 realises this recurrence explicitly.

Algorithm 1: Right-to-left binary modular multiplication [17]

Input: $a, b \in [0, p-1]$, prime p of bit length n
Output: $c = a \cdot b \bmod p$

```
c ← 0
for i = 0 to n − 1 do
    if b_0 = 1 then
        c ← (c + a) mod p
    end
    a ← (2 a) mod p
    b ← b ≫ 1
end
return c
```

The algorithm processes one bit of b per iteration. The partial-product accumulator c is conditionally updated by the current value of a whenever the inspected bit is set, and a is then doubled so that its value at iteration i equals $2^i\,a \bmod p$, in accordance with the invariant $a_i \equiv 2^i\,a \pmod p$. After n iterations, every non-zero bit of b has contributed its associated multiple to c, and the recurrence terminates with $c = a \cdot b \bmod p$.

Two properties of the algorithm justify its adoption. First, both non-trivial operations, the conditional accumulation $c \leftarrow c + a \bmod p$ and the doubling $a \leftarrow 2\,a \bmod p$ come from a single primitive, modular addition in $\mathbb{F}_p$. The doubling of a is nothing other than the modular addition of a with itself. Second, the algorithm requires only $O(n)$ modular additions and no field-wide multiplier: the asymptotic complexity is linear in the operand length rather than quadratic. These two features together, single primitive and linear complexity, are what make the right-to-left binary method a natural choice for $\mathbb{F}_p$-multiplication in a compact implementation, at the cost of higher latency than digit-serial or fully parallel alternatives.

Modular Inversion

Modular inversion is required at two distinct levels of the ECDSA hierarchy: inversion in $\mathbb{F}_p$ for the conversion of projective point representations back to affine coordinates, and inversion in $\mathbb{F}_n$ for the computation of k^{-1} during signature generation and s^{-1} during verification. Both share the same computational structure and, in the present accelerator, the same algorithm. The Binary Euclidean Algorithm (BEA), described in [17, Algorithm 3] in its adaptation to modular arithmetic, is adopted here because it operates using only right shifts and modular additions and subtractions, and can therefore be built entirely on top of the same modular add/subtract primitive introduced in Section 2.5.

Given a prime q and an operand $a \in [1, q-1]$, the algorithm maintains four intermediate variables u, v, x, y satisfying the loop invariants $u \equiv a \cdot x \pmod{q}$ and $v \equiv a \cdot y \pmod{q}$, and reduces u and v toward 1 by repeated even-halving and by mutual subtraction. When either variable reaches 1, the associated multiplier holds $a^{-1} \bmod q$. Algorithm 2 gives the schedule.

Algorithm 2: Binary Euclidean algorithm for modular inversion [17]

Input: Prime q and $a \in [1, q-1]$
Output: $a^{-1} \bmod q$

$u \leftarrow a$, $v \leftarrow q$, $x \leftarrow 1$, $y \leftarrow 0$
while $u \neq 1$ ***and*** $v \neq 1$ **do**
  while $u \bmod 2 = 0$ **do**
    $u \leftarrow u \gg 1$
    if $x \bmod 2 = 0$ **then**
      $x \leftarrow x \gg 1$
    else
      $x \leftarrow (x + q) \gg 1$
    end
  end
  while $v \bmod 2 = 0$ **do**
    $v \leftarrow v \gg 1$
    if $y \bmod 2 = 0$ **then**
      $y \leftarrow y \gg 1$
    else
      $y \leftarrow (y + q) \gg 1$
    end
  end
  if $u \geq v$ **then**
    $u \leftarrow (u - v) \bmod q$
    $x \leftarrow (x - y) \bmod q$
  else
    $v \leftarrow (v - u) \bmod q$
    $y \leftarrow (y - x) \bmod q$
  end
end
if $u = 1$ **then**
  return x
else
  return y
end

The algorithm terminates in at most $2\lceil \log_2 q \rceil$ iterations of the outer loop, so a modular inversion in $\mathbb{F}_p$ or $\mathbb{F}_n$ costs roughly $O(n)$ modular add/subtract operations with an $O(n)$ constant factor for the inner even-halving loops. Inversion is more expensive than multiplication, which motivates the use of Jacobian projective coordinates for the intermediate point operations of scalar multiplication so that only a single field inversion is required per scalar multiplication rather than one per point

operation. The single modular inversion in $\mathbb{F}_n$ required by the signature verification, for the computation of $w = s^{-1}$, is likewise a small constant contribution to the overall verification cost.

Scalar Multiplication

The scalar multiplication operation $[k]P$, which maps a scalar $k \in \mathbb{N}$ and a point $P \in E(\mathbb{F}_p)$ to the point obtained by adding P to itself k times, is the dominant cost of every ECDSA operation. Signature generation invokes it once, on the base point G, to compute the ephemeral point $R = [k]G$. Verification invokes it twice, on both G and the public key Q, to compute the two contributions $[u_1]G$ and $[u_2]Q$ whose sum yields the verification point. In every case k is a 384-bit integer, and the operation reduces to a sequence of point doublings and additions selected by the binary expansion of k.

The algorithm adopted in this work is the classical left-to-right binary method, which scans the bits of the scalar from the most significant to the least significant and performs, per bit, one unconditional point doubling followed by one conditional point addition. This is the additive-group analogue of square-and-multiply exponentiation and belongs to the same double-and-add family as the right-to-left binary method of [17, Algorithm 6]. The two variants differ in the direction in which the bits of k are scanned and in which of the two operands, accumulator or base point, is doubled at each step. Algorithm 3 states the version implemented here, including the standard skip-leading-zeros optimisation.

Algorithm 3: Left-to-right binary scalar multiplication with skip-leading-zeros

Input: Scalar $k = (k_{n-1}, \ldots, k_0)_2$ with $k_{n-1} \neq 0$, point $P \in E(\mathbb{F}_p)$
Output: $Q = [k]P$

Skip leading zeros of k and let j be the index of its most significant non-zero bit
$Q \leftarrow P$
for $i = j - 1$ **to** 0 **do**
  $Q \leftarrow 2\,Q$
  if $k_i = 1$ **then**
    $Q \leftarrow Q + P$
  end
end
return Q

The algorithm initialises the accumulator Q with the base point P at the position of the most significant non-zero bit of k, rather than starting from $\mathcal{O}$ and executing a redundant sequence of doublings on the point at infinity. From that position, each subsequent bit contributes one point doubling, and each set bit contributes one additional point addition. The total operation count is therefore j point doublings and, on average, $j/2$ point additions, where j is the position of the most significant non-zero bit of k. For a uniformly random $k \in [1, n-1]$, this evaluates to approximately 384 doublings and 192 additions per scalar multiplication.

Algorithmic observations

Two mode-dependent observations about the doubling and addition formulas used inside the loop are worth stating explicitly. First, because the base point P remains in affine form throughout the scalar multiplication while the accumulator Q is maintained in Jacobian projective coordinates, the addition step is a mixed affine–Jacobian addition rather than a full Jacobian–Jacobian addition. The mixed formula is cheaper by several field multiplications per invocation. Second, the doubling step exploits the $a \equiv -3 \pmod{p}$ property of NIST P-384 noted in Equation (2.58), which saves one field multiplication per doubling in Jacobian coordinates. The two optimisations together motivate the choice of coordinate system and are what make the implementation of Algorithm 3 on P-384 competitive in a compact accelerator.

Algorithm 3 is not constant-time: the number of point additions depends on the Hamming weight of k, and the position of the first non-zero bit is directly observable in the number of iterations skipped before the accumulator is initialised. Constant-time alternatives such as the Montgomery ladder [17, Algorithm 8] and Joye's Double-and-Add method [17, Algorithm 9] exists and would eliminate these dependencies at the cost of one point addition per bit regardless of the value of the bit. Their adoption is standard in implementations exposed to physical side-channel attacks such as power analysis, electromagnetic emanation measurement, or fault injection, all of which require the attacker to be in physical proximity to the device or to have direct control over its supply voltage and clock. In the deployment context of this work, an on-board cryptographic accelerator embedded in a spacecraft platform, no such proximity is available. Despite that, the non-constant runtime of the algorithm could be exploited by listening to the communication. Algorithm 3 is therefore not exploitable for physical side-channel attacks, but only for timing type of attacks. In a future version of the accelerator, a constant-time algorithm shall be implemented.

Signature verification imposes a different trade-off. The scalars u_1 and u_2 are functions of the public signature (r, s) and the message digest e, all of which are public, so no constant-time argument is required and the concerns of the preceding paragraph do not apply. The two contributions $[u_1]G$ and $[u_2]Q$ are computed by two independent invocations of Algorithm 3, each converted to affine coordinates, and combined by a single point addition to yield the verification point $[u_1]G + [u_2]Q$.

Key Pair Generation

To generate an asymmetric key pair for the domain parameters D, the following steps are needed:

1. **Private Key Selection.** Obtain a cryptographically secure random integer d in the open interval
$$d \in [1, n-1]. \tag{2.68}$$
To eliminate modular bias, FIPS 186-5 specifies using an approved Deterministic Random Bit Generator (DRBG) requesting extra random bits (e.g., $384 + 64 = 448$ bits) prior to reducing modulo $(n-1)$. The scalar d represents the private signing key and must remain confidential.

2. **Public Key Derivation.** Compute the public key point $Q = (x_Q, y_Q) \in E(\mathbb{F}_p)$ via scalar multiplication of the base point G:

$$Q = [d]G. \tag{2.69}$$

The point Q is validated by ensuring that $Q \neq \mathcal{O}$, that $x_Q, y_Q \in \mathbb{F}_p$, and that (x_Q, y_Q) satisfies the curve equation (2.56), that is, $y_Q^2 \equiv x_Q^3 + ax_Q + b \pmod{p}$. The public key is then distributed to verifiers directly or via digital certificates.

Digital Signature Generation

To sign a message M using domain parameters D and private key d, the signatory performs the following sequence:

1. **Message Hashing.** Compute the cryptographic hash of M using SHA-384 and derive the integer representative e:

$$H = \text{SHA-384}(M). \tag{2.70}$$

The digest H, a 384-bit string, is converted to the integer e used by the signing equation by taking the leftmost $\min(\text{hashlen}, \lceil \log_2 n \rceil)$ bits of H and interpreting them as a big-endian integer. For P-384 with SHA-384 both quantities are 384 bits, so no truncation occurs and e is the direct integer encoding of H.

2. **Ephemeral Key Generation.** Generate a per-message secret scalar k uniformly at random in the interval

$$k \in [1, n-1]. \tag{2.71}$$

The nonce k must be unique for every signing operation, reuse or leakage of k across two signatures permits recovery of the private key d.

3. **Ephemeral Point Calculation.** Compute the elliptic curve point $R = (x_R, y_R)$ using scalar multiplication:

$$R = [k]G. \tag{2.72}$$

4. **First Signature Component (r).** Extract the affine x-coordinate $x_R \in \mathbb{F}_p$ and reduce it modulo the subgroup order n:

$$r = x_R \bmod n. \tag{2.73}$$

If $r = 0$, the scalar multiplication yielded an x-coordinate that is a multiple of n; the algorithm must discard k and generate a new nonce.

5. **Second Signature Component (s).** Compute the modular multiplicative inverse $k^{-1} \bmod n$ using the modular inversion of Subsection 9 and evaluate

$$s = k^{-1} \cdot (e + r \cdot d) \bmod n. \tag{2.74}$$

If $s = 0$, the algorithm must discard k and restart the process.

6. **Output.** The signature for message M is the integer tuple

$$(r, s), \qquad r, s \in [1, n-1]. \tag{2.75}$$

Digital Signature Verification

To verify the signature (r, s) for message M using the signatory's public key Q and domain parameters D, the following steps are needed:

1. **Domain and Range Verification.** Verify that r and s are integers satisfying the range
$$1 \leq r < n \quad \text{and} \quad 1 \leq s < n. \tag{2.76}$$
If either bound is violated, the signature is rejected immediately.

2. **Public Key Integrity Check.** Verify that $Q \neq \mathcal{O}$, that Q lies on the curve, that is, $y_Q^2 \equiv x_Q^3 + ax_Q + b \pmod{p}$, and that $[n]Q = \mathcal{O}$.

3. **Message Hashing.** Compute the message digest independently using SHA-384 and convert it to the integer e as in Equation (2.70).

4. **Modular Inversion.** Compute the inverse of the signature component s modulo n:
$$w = s^{-1} \bmod n. \tag{2.77}$$

5. **Intermediate Multipliers.** Compute the two scalar multipliers
$$u_1 = (e \cdot w) \bmod n, \tag{2.78}$$
$$u_2 = (r \cdot w) \bmod n. \tag{2.79}$$

6. **Curve Point Reconstruction.** Compute the verification point $P = (x_P, y_P)$ as the sum of the two contributions $[u_1]G$ and $[u_2]Q$, each obtained by an independent invocation of Algorithm 3:
$$P = [u_1]G + [u_2]Q. \tag{2.80}$$
If $P = \mathcal{O}$, the signature is rejected.

7. **Signature Validation.** Extract the affine x-coordinate $x_P \in \mathbb{F}_p$ and reduce it modulo n:
$$v = x_P \bmod n. \tag{2.81}$$
The verifier accepts the signature as valid if and only if
$$v = r. \tag{2.82}$$

3 RTL design

In this chapter each implemented service-dedicated module is described. Section 3.1 details the AES agile core, in which one AES-256 datapath serves CTR, GCM and CMAC. Subsection 3.1.1 describes the primitive AES-256 module, where CMAC (described in Subsection 3.1.3) and GCM (described in Subsection 3.1.2) are built around. The same structure is used for the SHA agile core in Section 3.2, where the SHA-384 core and the built-around modules for HMAC and ECDSA, are described in Subsections 3.2.1,3.2.2 and 3.2.3 respectively. Section 3.3 presents the final top-level and the resources shared between the two agile cores (the unified padder, the register-file wrapper, and the top-level XOR). Finally, Section 3.4 covers the wrapper that maps the core onto the NOEL-V SoC via APB, AHB and DMA.

3.1 AES Agile Core

The AES-256 engine that will be presented in the next subsection is the core of the wrapping module `aes_agile_core` that turns it into a three-service accelerator capable of serving AES-256 in Counter mode (CTR), AES-256 in Galois/Counter Mode (GCM) for authenticated encryption, and AES-256 CMAC for cipher-based authentication. The wrapper adds five independent subsystems around the shared block cipher: an initialisation-vector counter block `iv_cnt` that maintains the pre-counter block J_0 and produces each successive counter Y_i through the `inc32` function of Equation 2.12, used by both CTR and GCM. A GHASH module built around a pipelined Karatsuba multiplier over GF(2^{128}) and a combinational reduction against the GCM polynomial of Equation 2.10 which realises the keyed hash of Equation 2.18 and, together with a hash-subkey register `h_reg` storing $H = \mathrm{CIPH}_K(0^{128})$, produces the authentication tag prescribed by Equation 2.24. A CMAC subkey generator that derives K_1 and K_2 from a single AES invocation on the zero block through the two shift-XOR paths of Equations 2.28,2.29, and a shared three-input 128-bit XOR that merges the previous ciphertext C_{i-1}, the padded message block M_i, and the selected subkey or GCM counter mask, realising both the CBC-MAC chaining of Equation 2.33 and the counter-mode masking of Equation 2.14 in a single physical XOR tree. A common preprocessor that aligns AAD, plaintext, ciphertext and CMAC message blocks to the 128-bit block requirement, dispatched to the mode-specific padding rule by a single `cmac` signal. Only one service is active at any given time, but the three datapaths co-exist physically because they share the AES-256 core, the preprocessor, the counter block generator and the three-input XOR. The rest of Section 3.1 details how each service uses these shared resources.

3.1.1 AES-256

The design adopted in this work for the AES core is a compact, iterative, single-round datapath: one full AES round is instantiated in hardware and executed once per clock cycle, with the intermediate State recirculated through a single 128-bit register until the fourteen rounds of AES-256 have been computed. The architecture is inspired by the compact core of Liu et al. [18], from which the on-the-fly key expansion described later in this section is directly adapted, and by the composite-field S-box of Reyhani-Masoleh et al. [19], from which the ROM-free byte substitution circuit is taken.

Choice of an iterative, non-pipelined datapath

Two architectural alternatives were considered before settling on the iterative one-round-per-cycle architecture: a fully unrolled datapath, in which the fourteen rounds are instantiated and produce one ciphertext per cycle at the cost of fourteen copies of the round logic, and a pipelined datapath, in which one round is instantiated per pipeline stage and independent input blocks are interleaved to reach a throughput of one block per cycle after the pipeline has filled, in the style of CIPHERX by Janamanchi et al. [7]. Both alternatives were rejected for the following explained reasons.

The first argument is dictated by the operating mode of CMAC. As shown in Equation (2.33), CMAC evaluates $C_i = \mathrm{CIPH}_K(C_{i-1} \oplus M_i)$, in which the input of the i-th AES invocation depends on the output of the $(i-1)$-th block. This is the chaining pattern of CBC-MAC and it does not allow data-level parallelism between successive blocks of the same message. A pipelined AES engine processing the same CMAC message would therefore leave every pipeline stage but one idle and an unrolled engine would produce one ciphertext per cycle whose latency is wasted, since the next input cannot be presented until the current output has fed back through the XOR. Only CTR and therefore GCM admit block-level parallelism, but supporting them at higher throughput than CMAC would require dedicating pipeline resources that CMAC cannot use, incompatible with the resource sharing objective of the accelerator.

The second argument is area. An unrolled AES-256 datapath replicates SubBytes, ShiftRows, MixColumns and AddRoundKey fourteen times, and a pipelined one, while single-instance in the round logic, still inserts a full 128-bit pipeline register after each stage and requires a key schedule that produces one Round Key per pipeline stage in parallel rather than one per cycle. The iterative one-round-per-cycle architecture instantiates each transformation exactly once and adds a single 128-bit register (the round register R), and its key expansion unit produces one Round Key per cycle from a 256-bit state, as described in the paragraph below. This is the smallest of the three variants and it is probably the best trade-off given between performance and area.

Datapath organisation

The complete datapath is shown in Figure 3.2. The 128-bit round register R stores the current State. Its output is fed in parallel into sixteen instances of the composite-field S-box, whose outputs are wired into the ShiftRows permutation and then into the column-parallel MixColumns network. The MixColumns output, the ShiftRows output, and the external plaintext `data_in` are the three inputs of a 3-to-1 multiplexer whose selection is driven by the control unit. The multiplexer output is XORed with the current Round Key inside AddRoundKey, and the result is stored in R on the next rising clock edge whenever the register enable is asserted. The ciphertext is taken from R and driven into `data_out`.

The three multiplexer inputs correspond exactly to the three phases of the AES cipher described in Section 2.1. Feeding `data_in` through the multiplexer, together with the first Round Key applied to AddRoundKey, realises the initial AddRoundKey that precedes the first full round. Feeding the MixColumns output through the multiplexer realises the ordinary round sequence SubBytes $\rightarrow$ ShiftRows $\rightarrow$ MixColumns $\rightarrow$ AddRoundKey of Equation (2.5). Feeding the ShiftRows output directly into the multiplexer, bypassing MixColumns, realises the final round, in which MixColumns is intentionally omitted as prescribed by FIPS 197 [10]. A single physical AddRoundKey therefore implements all fifteen key additions of an AES-256 encryption, and no inverse-direction hardware is instantiated, since all modes of the accelerator invoke the cipher exclusively in the forward direction.

Combinational transformations

ShiftRows is implemented as pure interconnect: the module contains no logic gates, only a rewiring of the sixteen State bytes according to the byte-position permutation of Section 2.1. AddRoundKey is a 128-bit bitwise XOR, described column by column as four 32-bit XOR instances. MixColumns is implemented one column at a time, following the matrix form of Equation (2.7): for each byte of the input column, the module computes `xtime` as a one-bit left shift followed by a conditional XOR with `0x1B` when the most significant bit of the byte is set, and combines these doubled bytes with the original bytes through the four fixed XOR patterns that reproduce the {02}/{03}/{01}/{01} coefficients of the MixColumns polynomial. None of these three transformations contain state, but they are purely combinational and are traversed in a single clock cycle together with SubBytes and AddRoundKey.

Composite-field S-box

The sixteen S-box instances in the datapath, together with the four instances in the key expansion unit, are implemented as purely combinational composite-field circuits rather than as a lookup table, following the construction of Reyhani-Masoleh et al. [19]. This architecture was selected for the purpose of radiation hardening: a look-up table would be implemented as a ROM, but in a space environment a bit flip in this memory caused by a small radiation particle would interrupt the correct functioning of the circuit. The full combinational implementation erases this risk. The idea is to exploit the tower-field isomorphism $\mathrm{GF}(2^8) \cong \mathrm{GF}\big((2^4)^2\big)$: the input

byte is first mapped through a linear input transformation into its representation in the tower field, the multiplicative inverse required by Section 2.1 is computed there using GF(2^4) subfield arithmetic (an exponentiation stage followed by a 4-bit subfield inverter and two multipliers by the recovered subfield inverse), and the result is passed through an output transformation that composes the inverse isomorphism with the affine map of Equation (2.6). The circuit is entirely gate-level and is also smaller than the ROM-based alternative once replicated twenty times across datapath and key expansion.

On the fly key expansion

The full expanded key of AES-256 comprises sixty 32-bit words, or 1920 bits. Storing it in a dedicated memory would represent a non-negligible area cost. The key expansion unit therefore generates the Round Keys on the fly, retaining only $N_k = 8$ words of internal state (a 256-bit register file $w_reg[0..7]$) and producing one 128-bit Round Key per clock cycle directly consumed by AddRoundKey. The unit follows the architecture proposed by Liu et al. [18] for the $N_k = 8$ case: rather than instantiating a full column of eight S-boxes to update the eight-word state in a single cycle, the update is split across two consecutive clock cycles distinguished by a single-bit `phase` flag and only four S-boxes are shared between the two cycles. When `phase` $= 0$, the unit outputs the upper four words of w_reg as the current Round Key and in the same cycle, updates the same four words according to the standard schedule $w[0] = \text{SubWord}(\text{RotWord}(w_reg[7])) \oplus \text{Rcon} \oplus w_reg[0]$ and $w[i] = w[i-1] \oplus w_reg[i]$ for $i = 1, 2, 3$. When `phase` $= 1$, the unit outputs the lower four words as the Round Key and updates them through the auxiliary schedule $w[4] = \text{SubWord}(w[3]) \oplus w_reg[4]$ and $w[i] = w[i-1] \oplus w_reg[i]$ for $i = 5, 6, 7$, which includes the additional application of SubWord prescribed for $N_k = 8$ at positions where $i - 4$ is a multiple of N_k. The four physical S-boxes are reused in both phases: in `phase` $= 0$ they compute $\text{SubWord}(\text{RotWord}(w_reg[7]))$, and in `phase` $= 1$ they compute $\text{SubWord}(w[3])$, driven from the same four bytes on their inputs through a multiplexer.

The round constants $\text{Rcon}[j] = \{02\}^{j-1}$ of Equation (2.9) are precomputed and stored in a seven-word table addressed by the three most significant bits of the round counter, `rcon_sel = nround_next[3:1]`. Because a new Rcon value is required only when a new pair of Round Keys begins, and because two consecutive rounds share the same Rcon (one from `phase` $= 0$, one from `phase` $= 1$), the counter advances the address every two rounds. No dedicated Rcon generator is required.

Control unit

The control unit is a five-state Mealy finite-state machine {`IDLE`, `INIT`, `ROUND_START`, `ROUND`, `COMP_DONE`} that sequences the datapath and the key expansion unit through the phases of Section 2.1. When `start` is asserted, `INIT` pulses `ld_key` to load the external 256-bit key into the key expansion register file and initialises the phase flag to zero. `ROUND_START` selects `data_in` through the input multiplexer and captures the initial AddRoundKey result into R. `ROUND` iterates the round transformation fourteen times, incrementing the four-bit round counter `nround_reg` on each cycle

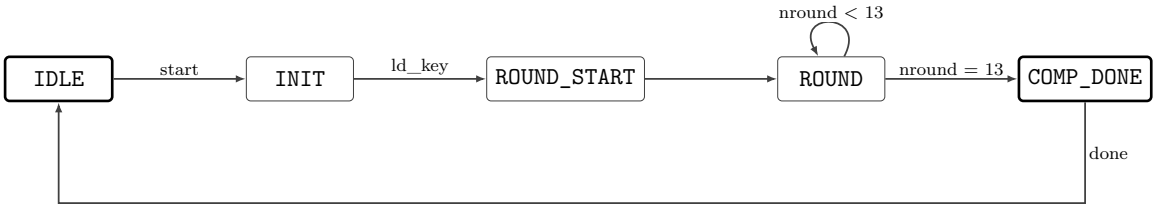


Figure 3.1: Control unit FSM of the AES-256 core (`cu_aes`).

and forcing the multiplexer to the ShiftRows path when `nround_reg` reaches 13, so that MixColumns is bypassed for the final round. `COMP_DONE` raises the `done` flag for one cycle before returning to `IDLE`. The complete encryption of one 128-bit block therefore requires seventeen clock cycles between `start` and `done`: one for key loading, one for the initial AddRoundKey, fourteen for the fourteen rounds, and one to signal completion.

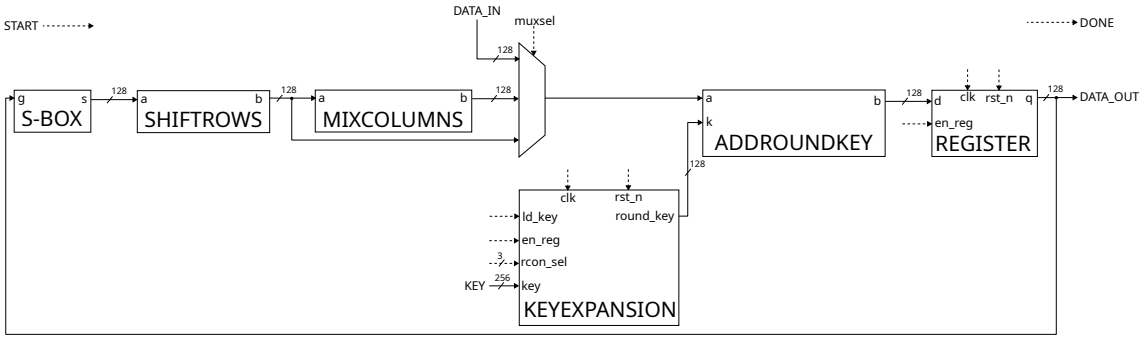


Figure 3.2: AES core datapath. The CU is omitted for simplicity, every dashed signal is fed/taken from the CU. Floating signals are CU dedicated, inserted for completeness

3.1.2 GCM

The GCM module reuses the iterative AES-256 core of Section 3.1.1 and surrounds it with four hardware blocks that together realise the authenticated encryption function of Equations (2.14),(2.24): a counter-block generator `iv_cnt` that maintains the pre-counter block J_0 and its increments, a `preprocessor` that aligns AAD, plaintext and ciphertext to the 128-bit block requirements of Section 2.1.1 a `ghash` module that implements the keyed hash of Equation (2.17) through a pipelined Karatsuba multiplier over GF(2^{128}), a three-input XOR that shares the counter-mode masking XOR with the CMAC chaining XOR of Section 3.1.3. The circuit follows the reference architecture of McGrew and Viega [20], in which a single forward AES engine, a counter, and a GF(2^{128}) multiplier are the only cryptographic primitives required for both confidentiality and authentication. CCSDS requires the tag length to be $t = 128$ bits [2]. Consistently with the specification of Section 2.1.1, the implementation supports only the 96-bit initialisation vector length, which allows the pre-counter

block to be formed by direct concatenation as in Equation (2.19).

Counter-block generator

The `iv_cnt` module stores a 96-bit `iv_reg` and a 32-bit counter `cnt`, and produces the current counter block Y_i as the 128-bit concatenation `y_vec = {iv_reg, cnt}`. When the CU asserts `iv_ld`, `iv_reg` is loaded from the external initialisation vector and `cnt` is initialised to 1. The value $Y_0 = J_0$ is produced at the first cycle after loading. Each subsequent counter block is obtained by asserting `cnt_up`, which implements the increment function inc_{32} of Equation (2.12), since only the low 32 bits are affected and no reduction modulo 2^{32} is required beyond the natural wrap-around of the 32-bit adder. The module also supports a counter reset `cnt_clear_n` and a completion signal `msg_done`.

Preprocessor

The `preprocessor` block computes the number of AES invocations required by the current operation and produces the correctly padded AAD and ciphertext blocks that will feed the GHASH input, in accordance with Equation (2.23). The number of full 128-bit blocks in a message of len bits is $\lceil len/128 \rceil$. When the message length is not a multiple of the block size, the final AAD block is right-zero-padded to 128 bits and the final ciphertext block is zero-padded on its unused byte positions, matching the padding lengths v and u of Equation (2.22). The two zero-padding paths are combinational and share the same shift and mask logic. A single mode signal (`cmac`) redirects the padding rule when the CMAC service is selected, as described in Section 3.1.3.

GHASH module

The `ghash` module implements the Equation (2.17) in a single multiplier organisation: a 128-bit accumulator is XORed with the current input block (Y_0 during initialisation, an AAD block, a ciphertext block, or the final length block $[|A|]_{64} \,\|\, [|C|]_{64}$, selected by `muxsel_aad`, `muxsel_y0` and `enc_done`) and multiplied by the hash subkey H stored in the register `h_reg`. Because the GCM specification interprets bytes in the little endian bit order of Section 2.1.1, but a hardware Karatsuba multiplier operates in the big endian bit order due to polynomial arithmetic, both operands are bit-reversed before entering the multiplier and the product is bit-reversed on the way out. The final tag is formed at the state `TAG_DONE` of the CU, when the last GHASH output is XORed with the counter mask Y_0 stored in `y0_reg`, realising the last step of Equation (2.24).

Karatsuba multiplier and $\mathrm{GF}(2^{128})$ reduction

The GHASH multiplier is inspired by the design of Lin and Chuang [3], whose distinguishing feature is the use of a recursive Karatsuba decomposition to improve the $O(n^2)$ multiplication cost of a common $\mathrm{GF}(2^{128})$ multiplier to an $O(n^{\log_2 3})$ AND/XOR complexity. The `karatsuba_block` module, parameterised on the operand width N, splits each input in two halves of $N/2$ bits, calls itself three times to obtain the low,

high, and mixed sub-products, and combines them through the three shifted XORs of the classical Karatsuba identity, in which subtraction reduces to XOR because the mathematical field is GF(2). The recursion terminates at $N = 2$ with a 2×2 multiplier of three gates. Registers are inserted at three points along the tree, at $N \in \{16, 64, 128\}$, turning the multiplier into a three-stage pipeline. The 255-bit product is then reduced modulo the GCM polynomial $f(u) = 1+u+u^2+u^7+u^{128}$ of Equation (2.11) by the combinational `gf_reduce` block: each bit at position $i \geq 128$ is XORed into positions $i-128$, $i-127$, $i-126$ and $i-121$ of the result, which is the direct application of the identity $u^{128} \equiv 1 + u + u^2 + u^7 \pmod{f(u)}$. The reduction does not require inversion and completes in a single clock cycle without lookup tables.

Control Unit

The `cu_aes_agile` control unit sequences the algorithm by means of different states. On the assertion of `start`, `IV` loads the initialisation vector and, if a new key has to be loaded, the next state becomes `HASH_KEY` to derive the hash subkey $H = \mathrm{CIPH}_K(0^{128})$ of Equation (2.18), saving it into `h_reg`. If additional associated data is present, the FSM iterates `AAD_INIT` and `AAD_COMP` to consume one padded AAD block per Karatsuba invocation, otherwise it proceeds directly to `Y_COMP` where Y_0 is computed and stored in `y0_reg`. During `ENCRYPT` state, there are different ongoing operations: AES invocation on the current counter block Y_i, a XOR of the resulting keystream with the current plaintext block through the shared 128-bit XOR (giving C_i), and a GHASH iteration on C_i. The same state sets the decryption path with the necessary input routing, since GCM is symmetric under encryption and decryption as explained at the end of Section 2.1.1. When the message length has been computed, `ENCRYPT_DONE` appends the length block to GHASH and `TAG_COMP`/`TAG_DONE` produce the final tag by XORing the last GHASH output with Y_0. On decryption the same state compares the recomputed tag with the received one and sets `verify_ok` accordingly.

IV repetition check

In the current revision of the core, verifying IV uniqueness under the same key (in compliance with NIST SP 800-38D [11]) is left to the caller. This design choice was made due to the fact that the caller knows when the key changes and can therefore set easily another nonce. However, this is a simple comparison operation and therefore could be included in newer versions of the core.

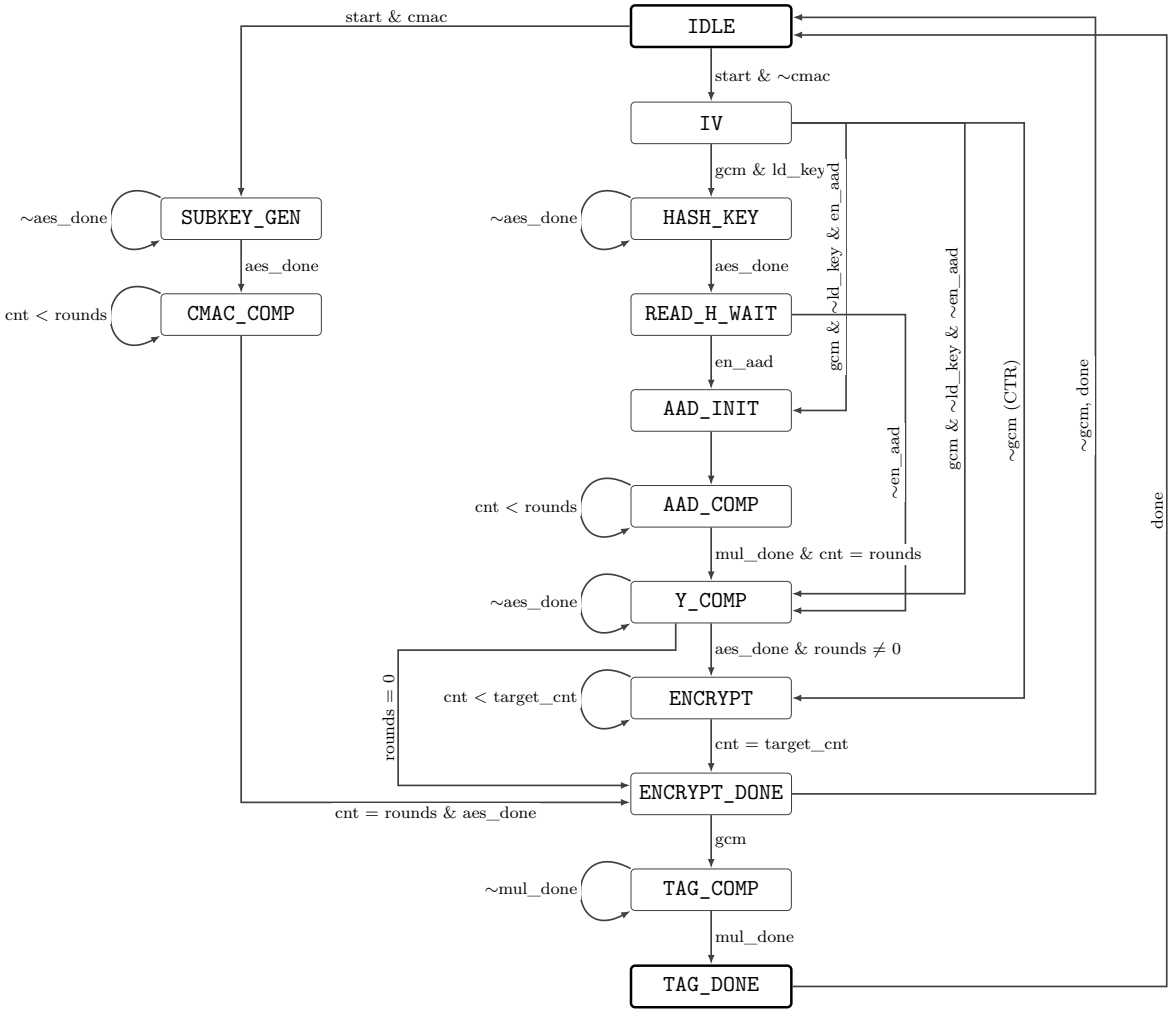


Figure 3.3: Control unit FSM of the AES agile core, for GCM and CMAC flows (`cu_aes_agile`).

3.1.3 CMAC

The CMAC module shares the 128-bit AES-256 datapath already instantiated for CTR and GCM, requiring only a small amount of additional logic for subkey derivation and final block masking. In this work $b = 128$ is used, since the block cipher is AES-256, and $Tlen = 128$ is adopted so that no truncation of the CBC output is performed. The architecture of the CMAC datapath follows the block schematic and structure idea of the CIPHERX unified accelerator [7], in which a single AES core, a subkey generator, a final-block masking XOR and a preprocessor are instantiated. The essential difference with the paper is that CIPHERX implements a fully pipelined AES core, which is not the case of this work, for the reasons discussed

in Section 3.1.1. As a consequence, no parallelism is exploited between successive CMAC blocks, since the CBC-MAC chaining of Equation (2.33) would prevent any pipelined AES from being fully exploited if the utilizer of this accelerator needs only authentication. CIPHERX authors exploit the pipeline by encrypting the message while generating MAC, which is not mandatory due to the fact that user would need only authentication or encryption.

Subkey generation

The `subkeys_gen` module implements Equations (2.28),(2.29). The initial value $L = \mathrm{CIPH}_K(0^{128})$ is obtained by setting the AES input to zero (`muxsel_aes_in` = 00) and triggering a single AES invocation from the CU state `SUBKEY_GEN`. When `aes_done` becomes *1*, the AES output is fed into two shift-XOR paths that produce K_1 and K_2: each path performs a one-bit left shift and conditionally XORs the result with the constant $R_{128} =$ `0x87` (the low byte of the CMAC reduction polynomial of Equation (2.26)) whenever the most significant bit of the operand is set. A one-bit selector `k_sel` computed by the preprocessor, chooses between K_1 and K_2 whether the final message block is complete or incomplete, and the selected subkey is stored into an internal 128-bit register on the same cycle that stores L. The two subkeys K_1 and K_2 and the intermediate value L are never exposed outside this module, in line with the confidentiality requirement of Section 2.2.

Preprocessor

When the CMAC mode is selected, the `preprocessor` block performs the two-case masking of the final block of Equation (2.32). If the last block is a full 128-bit block, the preprocessor keeps it unchanged. If the last block is shorter, the preprocessor pads it with a single 1 bit followed by the minimum number of zero bits required to reach 128 bits. The preprocessor selects the blocks padding by comparison with the `msg_len` 7 LSBs. The same block is reused from the GCM architecture of Section 3.1.2: the two operating modes are selected by a single mode signal, so no CMAC-specific instance of the preprocessor is required.

Shared XOR

The three-input XOR of `aes_agile_core`, whose inputs are multiplexed by `muxsel_xor_in1`, `muxsel_xor_in2` and `muxsel_xor_in3`, is the single point at which the CMAC chaining and the GCM counter-mode masking converge. During normal CMAC blocks, the three multiplexers select the previous ciphertext C_{i-1} (held in `aes_out_reg`), the padded message block M_i from the preprocessor, and zero. The XOR then produces $C_{i-1} \oplus M_i$, which is fed into the AES core to compute $C_i = \mathrm{CIPH}_K(C_{i-1} \oplus M_i)$ in agreement with Equation (2.33). During the final block computation, the third input is driven with the selected subkey K_1 or K_2, giving $C_{n-1} \oplus M_n^* \oplus K_{1\,\mathrm{or}\,2}$ in a single cycle, according to Equation (2.32) without a dedicated hardware block. The first CMAC block is handled by forcing the previous ciphertext input to zero, so that the initial condition $C_0 = 0^b$ of Equation (2.33) is satisfied.

Control Unit

The CMAC schedule is controlled by three dedicated states of `cu_aes_agile`: `SUBKEY_GEN` which computes L and saves the selected subkey, `CMAC_COMP` which iterates over the n message blocks, setting to *1* `aes_start` once per block and updating `aes_out_reg` on each `aes_done`, and `ENCRYPT_DONE` which raises `done` and on verification mode, drives `en_tag_chk` so that the received tag on `data_in` is compared with the recomputed one and the outcome captured into a dedicated register (with the codewords `AUTH_PASS` and `AUTH_FAIL`). Because CMAC and CBC-MAC verification share the same generation function as noted at the end of Section 2.2, only a single verification datapath is instantiated and the tag comparison reuses the same XOR tree followed by an OR reduction. The total cycle count is approximately $17\,n + 17$ clock cycles per message, where n is the number of 128-bit blocks and each AES invocation costs the seventeen cycles of Section 3.1.1: one leading AES invocation for the subkey generation, n for the CBC iteration, and a single final cycle for the tag output and the authentication flag.

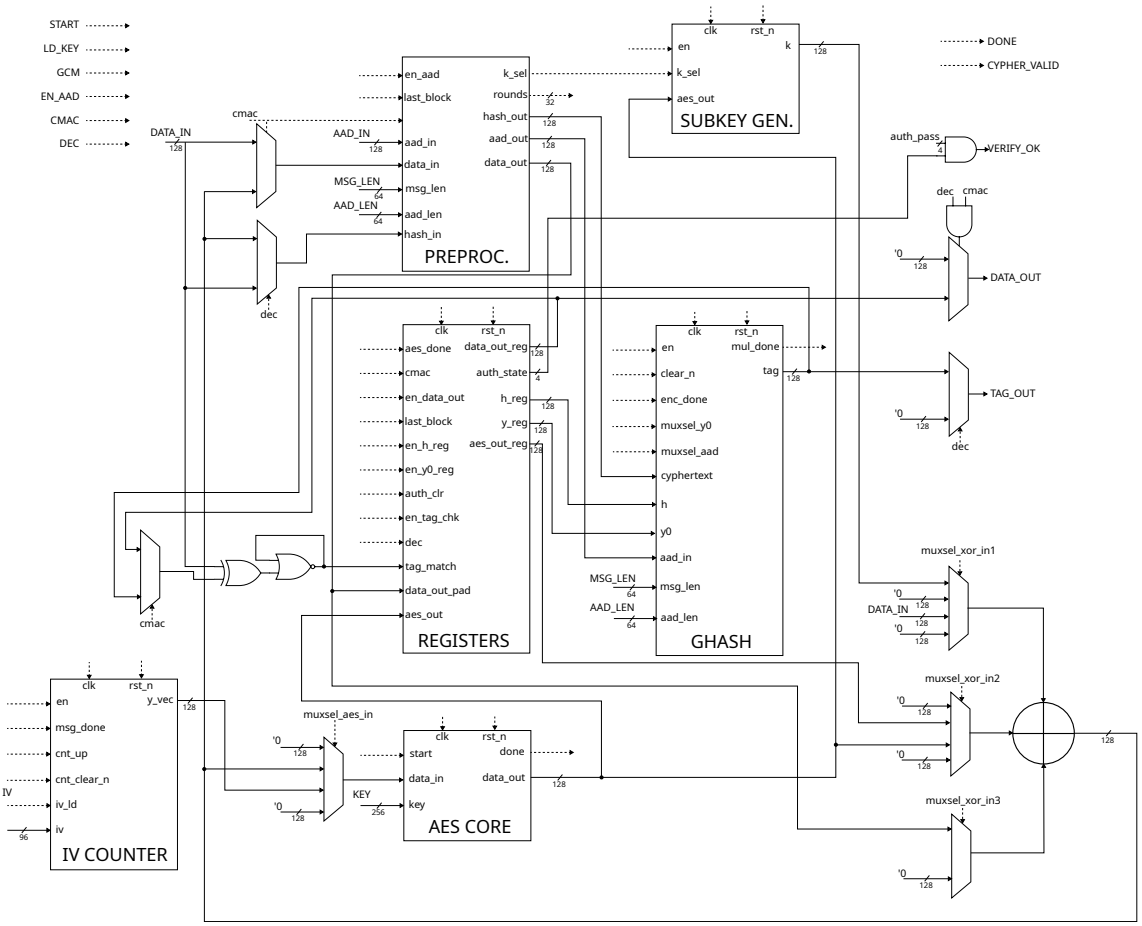


Figure 3.4: AES agile core datapath. The CU is omitted for simplicity, every dashed signal is fed/taken from the CU. Signals with uppercase names are inputs and outputs of the module. Floating signals are CU dedicated, inserted for completeness.

3.2 SHA Agile Core

The SHA-384 engine that will be presented in the next section, is the core of the wrapping module `sha_agile_core` that turns it into a two-service accelerator capable of serving both HMAC-SHA-384 tag generation and ECDSA over the NIST P-384 curve. The wrapper adds three independent subsystems around the shared hash primitive: a small HMAC controller (`cu_hmac`) with a key buffer and a XOR that implements the key/pad combination of Equation (2.53). A full ECDSA FSM (`cu_ecdsa`) with a micro-code ROM, a shared 32-entry 384-bit register file, four parallel modular processing elements, and one modular inverter that together evaluate $\mathbb{F}_p$ and $\mathbb{F}_n$ operations of Section 2.5. A set of shared muxes on the hash input that allow either the HMAC controller or the ECDSA sequencer to drive the SHA-384 core. Only one service is active at any given time, but the two datapaths co-exist physically because they share the register file, the padder, and the SHA-384 engine itself. The rest of Section 3.2 details how each service uses these shared resources.

3.2.1 SHA-384 core

The SHA-384 primitive of the accelerator is implemented as a compact engine that processes one 64-bit word of the message per clock cycle rather than instantiating a 1024-bit round in hardware. The datapath is composed of two arithmetic units, the `scheduler` which realises the message-schedule recurrence of Equation (2.43) and the `compressor`, which realises the round function of Equations (2.45), (2.46),(2.47) and the chaining update of Equation (2.48). There are also other modules in the datapath, the `padder` that applies the FIPS 180-4 padding rule of Equation (2.42) on the fly, two ROMs storing the 80 round constants $K_t^{\{512\}}$ and the eight SHA-384 initial hash words $H_i^{(0)}$. The whole module follows the architecture proposed by Juliato and Gebotys [4] for a HMAC-SHA-2 accelerator, where the collapsing of every multi-operand round addition into a CSA tree, and the reuse of a single carry-propagate adder for both the round update and the digest update, are implemented by modifying the original scheme. The choice of SHA-384 is motivated in Section 3.2: the same engine serves both the HMAC-SHA-384 tag generation and the SHA-384 message digest required by ECDSA over the NIST P-384 curve.

Word-serial datapath

The working state of the round function is stored not in eight individual 64-bit registers a, b, c, d, e, f, g, h but in two four-deep 64-bit shift-register FIFOs, `fifo_ad` and `fifo_eh`. The newest FIFO position exposes a (respectively e) and the oldest position exposes d (respectively h), so that the shift-down direction of the FIFO reproduces the one-position rotation
$(a, b, c, d, e, f, g, h) \leftarrow (T_1 + T_2,\ a,\ b,\ c,\ d + T_1,\ e,\ f,\ g)$ explained in Equation (2.47) without any multiplexer. This is the same construction used by Juliato and Gebotys [4] and it eliminates the entire crossbar between the eight working variables. The message schedule is stored in a sixteen-deep 64-bit FIFO `fifo_w` inside the scheduler, so that the four positions accessed by the operands of Equation (2.43) (W_{t-16}, W_{t-15},

W_{t-7}, and W_{t-2}) become the fixed FIFO outputs `fifo_out[0]`, `[1]`, `[9]` and `[14]` respectively. The schedule extension therefore requires no memory and relative address generator. Together, the three FIFOs replace what would otherwise be a $16 \times 64 + 8 \times 64 = 1536$-bit register file with a set of shift registers mappable to distributed RAM on the target FPGA.

Message schedule

The `scheduler` module implements the recurrence $W_t = \sigma_1^{\{512\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{512\}}(W_{t-15}) + W_{t-16}$ of Equation (2.43). The four contributors are formed through a logic network from the appropriate outputs of `fifo_w`, and the 64-bit modular addition of four operands is decomposed into two cascaded 3:2 carry-save adders (`csa0` and `csa1`) followed by a single 64-bit carry-propagate adder that produces `sum_out` $= W_t$. Rather than emitting only the final sum, the scheduler exports the reduced (sum, carry) pair (`w_sum_out`, `w_carry_out`) of the last CSA stage, so that the following compressor stage may absorb the carry vector into its CSA tree without the intermediate carry propagation. This carry-save composition shortens the critical path across the two modules. During the first sixteen cycles of a new chunk, when the message-block words $M_t^{(i)}$ are loaded directly from the padder, the multiplexer selected by `mux_sel_exp` bypasses the CSA network: the message word enters the FIFO through `fifo_in` and is simultaneously exported on `w_sum_out` with the carry vector forced to zero, so that the compressor sees the correct W_t from the very first round without any special case hardware.

Compressor and adder sharing

The `compressor` module implements the round function of Equations (2.45), (2.46), (2.47). The temporary T_1 of Equation (2.45) is the modular sum $h + \Sigma_1(e) + \mathrm{Ch}(e, f, g) + K_t + W_t$, which the design collapses into a tree of five 3:2 CSA cells (`csa0`–`csa2`, `csa_t1`) that computes the (sum, carry) pair supplied by the scheduler. The output of the T_1 tree is not resolved by a carry-propagate adder, but it is left as a (sum, carry) pair and forwarded, together with d or with $T_2 = \Sigma_0(a) + \mathrm{Maj}(a, b, c)$, into two 3:2 CSA cells (`csa_e`, `csa_a`, `csa_a0`) that produce the two (sum, carry) pairs whose computations yields $e_{\mathrm{new}} = d + T_1$ and $a_{\mathrm{new}} = T_1 + T_2$. Only these last two operations require a full 64-bit carry-propagate adder and one of the two, the `a_new` adder, is shared with the digest phase of the FSM: a multiplexer at its inputs selects either (`csa_a_sum`, `csa_a_carry`$\ll 1$) when the compressor is active, or (`fifo_h_out[0]`, h) when the digest phase is active. In the second mode, the adder implements the chaining update $H_k^{(i)} = H_k^{(i-1)} + a_k^{\mathrm{final}}$ of Equation (2.48) using the same physical 64-bit adder used per round for a_{new}, so no dedicated digest adder is instantiated. This approach is a direct implementation of resource sharing, while in the original paper architecture [4], the authors used two additional adders for the digest computation, which occupy more area.

The rotating FIFO organisation of the working state has one additional consequence that is exploited during the digest phase. When the multiplexer at the `fifo_eh` input is switched from e_{new} to d (as it is under `en_digest`), each digest

cycle pushes into e the value that just left d. Over the eight digest cycles required to update all eight chaining values, the working state (a, b, c, d, e, f, g, h) is repopulated with $(H_0^{(i)},\ H_1^{(i)},\ H_2^{(i)},\ H_3^{(i)},\ H_4^{(i)},\ H_5^{(i)},\ H_6^{(i)},\ H_7^{(i)})$ in the order required by the next chunk. So, the FSM can transition directly from `DIGEST` to `LD_DATA` without any explicit reload of the working state, and multi-chunk messages do not incur any chunk initialization overhead beyond the eight cycles of the digest.

Sigma functions

The four rotation based diffusion functions of the SHA-384 round are implemented by a single parameterised module, `sha384_sigma`, whose parameter `sel` selects one of four hardwired rotate/shift patterns realising respectively, $\sigma_0^{\{512\}}$, $\sigma_1^{\{512\}}$, $\Sigma_0^{\{512\}}$ and $\Sigma_1^{\{512\}}$ of Equations (2.40),(2.41),(2.38),(2.39). Each instance evaluates the three input XOR of two right rotations and either a third right rotation (for the capital sigmas) or a logical right shift (for the lowercase sigmas), and consumes no logic beyond the 64-bit XOR tree because rotations and shifts are pure interconnect at bit level. Two instances are instantiated in the scheduler (σ_0, σ_1 on the FIFO taps at positions 1 and 14) and two in the compressor ($\Sigma_0(a), \Sigma_1(e)$), each attached to its own dedicated port so that the four evaluations run in parallel within a single round cycle.

Padder

The `padder` module applies the padding rule of Equation (2.42) on the fly, one 64-bit word per cycle, driven by a four-bit `word_cnt` that indexes the sixteen words of the current 1024-bit chunk. At the beginning of a hash the module captures the total message length *msg_len* in `msg_len_reg`, and at each chunk subtracts 1024 to keep the residual bit count that remains to be consumed. From this residual the module derives two positional values that determine the padding for the current chunk: the target word index `word_target` = `msg_len_reg[9:6]`, and the target bit within that word `bit_target` = `msg_len_reg[5:0]`. Two right shifted bit mask constants, `pad_bit_64` = `0x8000...0` $\gg$ `bit_target` and `mask_64` $=\sim$ (`0xFFFF...F` $\gg$ `bit_target`), are then combined with the input word to inject the single `1` bit at the correct position and force every subsequent bit of the same word to zero. The remainder of the chunk is either zero-padded (if the residual length fits together with the 128-bit length encoding into the same 1024-bit block, that is if `msg_len_reg` < 896) or forwarded as a vector of zeros to a subsequent all-padding chunk in which only word positions 14 and 15 carry the two halves of the message length. The two conditions are handled by a small internal state made of the `one_appended`, `pad_en` and `one_append_en` flip-flops, which are set at the start of each new chunk and used by the word-position multiplexer. The flag `last_chunk` is exposed to the control unit so that the FSM knows when the digest of the current chunk will be the final digest.

Constant ROMs

The eighty 64-bit round constants $K_t^{\{512\}}$ defined in FIPS 180-4 [13] are stored in a small combinational ROM `k_rom` indexed by a 7-bit address that increments once per round cycle whenever the compressor is active. A similar 8-entry ROM `h_rom` stores the eight initial hash words $H_i^{(0)}$ specific to the SHA-384 variant, addressed in descending order so that the eight-cycle `INIT` phase streams them into `fifo_ad`, `fifo_eh`, and `fifo_h` in the correct positional order for the convention of `fifo_ad_out[3]`$= a$ and `fifo_eh_out[0]`$= h$. Both ROMs are read-only and implemented as distributed LUT-based memory on the target FPGA because their small size does not warrant the use of block RAM.

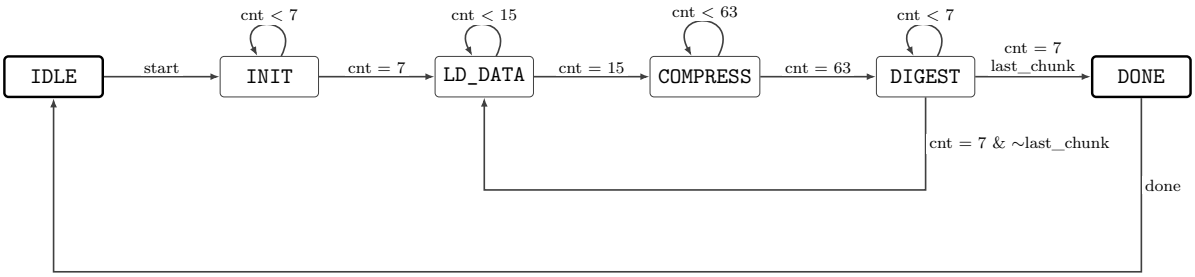


Figure 3.5: Control unit FSM of the SHA-384 core (`cu_sha`).

Control Unit

The finite-state machine `cu_sha` sequences the whole computation through six states, {`IDLE`, `INIT`, `LD_DATA`, `COMPRESS`, `DIGEST`, `DONE`}, driven by a six-bit cycle counter `cnt_cycles`. On `start` the FSM latches the message length into the padder and enters `INIT`, which streams the eight initial hash words from `h_rom` into the three FIFOs over eight cycles. The FSM then enters `LD_DATA`, in which the first sixteen padded message words of the current chunk are pushed into `fifo_w` while the compressor executes the first sixteen rounds ($t = 0, \ldots, 15$) using $W_t = M_t^{(i)}$ from the scheduler bypass path. `COMPRESS` then executes the remaining 64 rounds ($t = 16, \ldots, 79$), during which W_t is generated by the scheduler and, if the chunk is not the last, the padder concurrently prefetches the next chunk. The eight-cycle `DIGEST` phase applies the chaining update of Equation (2.48) through the shared `a_new` adder and, as noted above, repopulates the working state for the next chunk in the same phase. If the chunk was the last, the FSM transitions to `DONE` and asserts `rd_hash` throughout the digest so that the eight 64-bit words of the final chaining value are output on `data_out` at one word per cycle (the caller retains only the leading six for the 384-bit truncation of Equation (2.50)). One 1024-bit chunk therefore consumes $16 + 64 + 8 = 88$ clock cycles of active hashing after the eight-cycle `INIT`, and a single-chunk short message completes in approximately one hundred cycles from `start` to `done`.

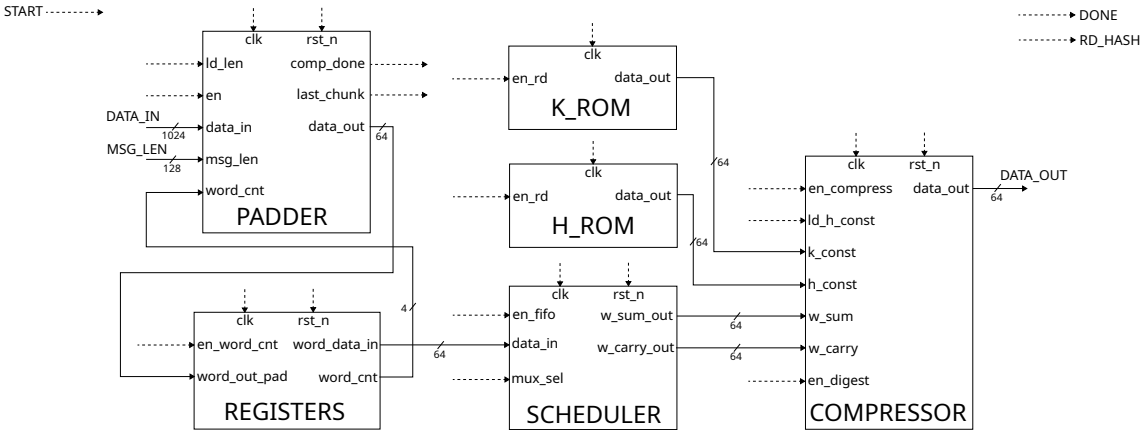


Figure 3.6: SHA core datapath. The CU is omitted for simplicity, every dashed signal is fed/taken from the CU. Floating signals are CU dedicated, inserted for completeness.

3.2.2 HMAC

The specification of Equations (2.52),(2.53) makes clear that HMAC requires no primitive other than the hash compression function and a small amount of block-wise XOR logic. The implementation follows the architecture of Juliato and Gebotys [4], in which the two hash invocations of Equation (2.53) are sequenced through a single SHA core by rewinding its chaining value between the inner and the outer invocation, and the two 1024-bit constants ipad and opad are computed into a single XOR whose second operand is switched by a MUX.

Key processing

The prepared key K_0 of Equation (2.52) is stored in a buffer `hmac_k0_mem` of sixteen 64-bit words, inside the shared `register_file` module of Section 3.2.3 so that no dedicated BRAM primitive is used. On the assertion of `start_hmac` with a new key, the FSM transitions to the state `KEYCOMP` and drives the padder in its dedicated *key-comparison* mode (see padder in Section 3.2.1), which sets three internal flags that distinguish the three cases of Equation (2.52): $\text{len}(K) < b$, $\text{len}(K) = b$, and $\text{len}(K) > b$. In the first two cases the FSM enters `KEYSAVE` and stores the sixteen 64-bit words of the padded key into `hmac_k0_mem` in sixteen cycles. In the third case the FSM enters `KEYHASH`, which calls the SHA engine on the long key and, on `rd_hash`, transitions to `KEYSAVEHASH`, in which the six 64-bit words of the ℓ-bit digest are written into `hmac_k0_mem` through a multiplexer input driven by the SHA output. In all three cases the buffer content at the end of the process is the 1024-bit prepared key K_0, in the order required by the ipad/opad XOR.

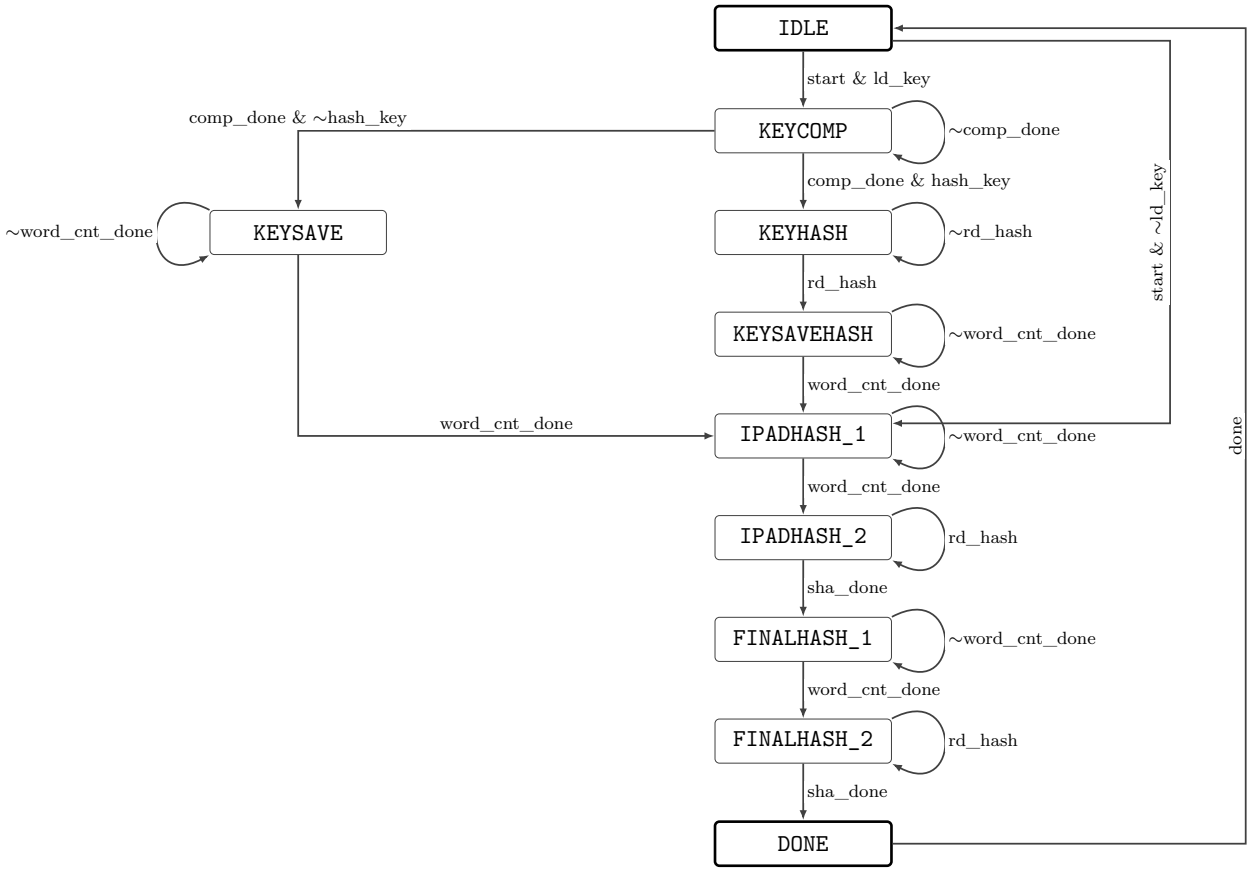


Figure 3.7: Control unit FSM of the HMAC wrapper (`cu_hmac`).

XOR structure

The XOR that combines K_0 with ipad or with opad is a single 64-bit XOR whose first operand is the current word read from `hmac_k0_mem` and the second operand is one of the two constants selected by `xor_sel`. Because ipad and opad are the repeated bytes `0x36` and `0x5C` respectively (see Equation (2.51)), both are 64-bit literals in the RTL. The XOR output feeds the SHA input through the multiplexer `sha_in_sel`, which also selects the current message word (from `shared_word_in`) for message blocks and the eight-word inner digest (from the buffer `hmac_fifo_mem` inside the same shared register file) for the outer hash. Another multiplexer called `data_len_sel`, switches the length input of the padder so that the SHA engine sees a total input length of $b + \text{Mlen}$ for the inner hash and $b + \ell$ for the outer hash, which is what Equations (2.54),(2.55) describe. No external padder configuration is required, since it is picked up directly from the XOR output when computed.

Control Unit

The HMAC controller sequences the two invocations of Equation (2.53) through the states `IPADHASH_1`/`IPADHASH_2` and `FINALHASH_1`/`FINALHASH_2`. In `IPADHASH_1` the sixteen 64-bit words of $K_0 \oplus$ ipad are fed into the SHA input, followed in `IPADHASH_2` by the message M and its padding. When the padder raises `rd_hash`, the six 64-bit words of the inner digest T_{inner} are captured into `hmac_fifo_mem` in a write pattern that follows the reverse order emission of the SHA engine described at the end of Section 3.2.1. `FINALHASH_1` then streams $K_0 \oplus$ opad (the same K_0 read from `hmac_k0_mem`, XORed with the second pad through the shared XOR), and `FINALHASH_2` streams T_{inner} from `hmac_fifo_mem` into the SHA input, so that the final digest is $T = H\Big((K_0 \oplus \text{opad}) \parallel T_{\text{inner}}\Big)$ in agreement with Equation (2.55). Because both hashes use the same padder, both padding lengths follow the rule of Equation (2.42) without any HMAC specific logic. The adjustment $\text{Mlen} \rightarrow \text{Mlen} + b$ is a compile time addition performed by `data_len_sel`. The six words of the outer digest are shifted into a 384-bit register `full_mac` through the signal `rd_mac`, and the resulting tag is outputted on `mac_out`.

K_0 reuse across operations

The computed key buffer `hmac_k0_mem` is not cleared between successive HMAC operations, so a valid K_0 produced by an earlier call remains available until a new one is written. `cu_hmac` exploits this by exposing a one-bit input `ld_key` that, when is zero at the start of a new operation, bypasses the entire key computing chain (`KEYCOMP`, `KEYSAVE`, `KEYHASH`, `KEYSAVEHASH`) and transitions the FSM directly to `IPADHASH_1`. In this fast path the ipad/opad XOR reads K_0 from `hmac_k0_mem` as if it had just been written, and the resulting HMAC is bit-identical to one produced by the full sequence. The saving is not negligible: in the short key case (`len(K)` $\leq b$), a full key setup consumes sixteen cycles of streaming into the buffer. In the long key case (`len(K)` $> b$), it additionally consumes one full SHA-384 call on the raw key. Because HMAC is typically applied to a stream of messages authenticated under a single session key, in the common case the key is prepared once at session start and every subsequent authentication in the session takes the fast path, reducing the per-message overhead of HMAC to the two hash invocations of Equation (2.53) without any preparatory work.

Verification

When the mode signal `dec` is asserted the module operates as a tag verifier: the received tag is presented on `rx_mac`, the recomputed tag is compared bitwise against it, and the match signal `mac_match` drives a two code authentication register `auth_state`. The register is cleared to `AUTH_FAIL` whenever a new HMAC operation starts, so no residual state from a previous verification can interfere during a new call.

3.2.3 ECDSA P-384

The specification of ECDSA over P-384 decomposes the algorithm into three classes of operations: a single SHA-384 invocation on the message, a sequence of finite-field operations in $\mathbb{F}_p$ that implement point addition, doubling and scalar multiplication, and a small number of $\mathbb{F}_n$ operations that produce the signature components r and s. As already stated, the SHA-384 instantiated for HMAC is reused for ECDSA. The elliptic-curve and modular inversion logic is unique to this service and is described in this section, together with the coordinate representation and the scalar multiplication algorithm adopted for the implementation. The ECDSA module described below follows the structure presented in the work of Kieu-Do-Nguyen et al. [5], where the architecture is organised as a small VLIW-style datapath around a shared register file, in which four parallel modular Processing Elements (`PE`) and one inverter (`PE_inv`) are sequenced by a micro code FSM `cu_ecdsa` that reads its program from a ROM.

Processing element

Each of the four `PE` instances implements the three mode dependent operations (modular addition, modular subtraction, and modular multiplication). The datapath of the PE is a hierarchical 385-bit carry select/carry look ahead adder that evaluates two candidate results in parallel on every cycle, $x + y$ and $x + y - p$, and selects between them from the sign of the extended precision sum, so that the reduction modulo p is absorbed into the addition without a separate comparison-and-subtract stage. This is the mechanism to which Kieu-Do-Nguyen *et al.* [5] refer when they note that "the addition/subtraction module naturally performs the modular reduction in each operation; there is no need for an additional modular reduction module": the reduction is not a distinct step but a property of the datapath that is exploited on every cycle of the multiplication. An input CSA network before the two adders includes a third operand into the sum when needed, so that the multi operand additions of the modular multiplication accumulation and of the modular inverter halving path both resolve into a single carry propagation. The subtraction operation is implemented as an addition with the two's complement form of the subtrahend, and the ADD, SUB, and multiplication mode share the same physical adder through a small set of one hot input multiplexers.

Modular multiplication is implemented by the same right-to-left binary Algorithm 1. The multiplicand and the multiplier are stored in two internal registers `a_reg` and `b_reg` and each iteration of the loop is executed in two cycles of the shared modular add/subtract unit: in the first cycle (`MULT_ACCUM`), the running product register is conditionally updated by $\mathrm{SP} \leftarrow (\mathrm{SP} + a) \bmod p$ whenever the least significant bit of b is set. In the second cycle (`MULT_DOUBLE`), the multiplicand register is updated by $a \leftarrow (a + a) \bmod p$ and the multiplier register is right shifted by one position. After $n = 384$ iterations the running product holds $a \cdot b \bmod p$ and the total cycle count is at most $2n = 768$ cycles. The elliptic curve doubling formula for NIST P-384 is expressed with the reduced coefficient $a \equiv -3 \pmod{p}$ noted in Equation (2.58), so no dedicated squaring hardware is required and squarings are implemented by loading the same operand into both `a_reg` and `b_reg`.

Modular inverter

The dedicated `PE_inv` module implements the Binary Euclidean Algorithm 2 using the same add/subtract datapath as the four `PE` instances. Four 384-bit registers u, v, x, y hold the loop invariants $u \equiv a \cdot x \pmod{q}$ and $v \equiv a \cdot y \pmod{q}$. The FSM alternates between an even halving phase, which right shifts one of u or v and either right shifts the corresponding multiplier or adds the modulus before right shifting it, and a subtraction phase, which subtracts the smaller of u, v from the larger and updates the associated multiplier by the corresponding modular subtraction. The same carry select/carry look ahead datapath used inside the four PEs is instantiated also in the inverter and reused for every halving-and-modular-add step of the algorithm. Two invocations are needed per scalar multiplication in the general case (one at the end of the Jacobian $\rightarrow$ affine conversion in $\mathbb{F}_p$ and one on the signature component s in $\mathbb{F}_n$) and the modulus is switched between p and n under `cu_ecdsa` control by the signal `modulo_pe`.

Register file

Every operand of the ECDSA computation is held in the shared `register_file`: a 32-entry 384-bit array implemented in distributed RAM, with a single write port and nine independent read ports (two per `PE`, and one for the inverter), so that all four PEs and the inverter can fetch operands from arbitrary addresses in the same cycle. Register 15 is hardwired to zero so that the constant 0 is available as a source operand without occupying a physical storage location. The write port is shared through a writeback multiplexer `rf_wb_sel` that selects among the four PE results and the inverter result. When several PEs finish in the same cycle their results are drained through the shared port over a small number of consecutive cycles, driven by the `ST_WRITEBACK` state of the sequencer. Six of the 32 entries (indices 13, 14, 19, 20, 21, 22) are also exposed as dedicated shadow registers that route the affine coordinates x_R, y_R, the intermediate scalars u_1, u_2, the signature components r, s, and the reconstructed x-coordinate x_P of the verification directly to the control unit, so that mode transitions can be taken without going through the read ports. The same distributed RAM primitive that holds the ECDSA operand file also hosts the HMAC prepared-key buffer `hmac_k0_mem` and the inner-digest buffer `hmac_fifo_mem`, which is the register file cited in the XOR paragraph of 3.2.2.

Micro-op ROM

The entire elliptic-curve program is stored in a small ROM (`ecdsa_rom`) as a sequence of micro-instructions defined by the `uop_t` type in `ecdsa_pkg`. Each micro instruction packages, in a single word, up to four parallel PE operations (each with its own opcode $\in \{\mathrm{ADD}, \mathrm{SUB}, \mathrm{MUL}, \mathrm{NOP}\}$ and its own source/destination register addresses) and, optionally, one inverter operation with its source and destination. A one-bit `mod_n` field selects the modulus between p and n for every instruction. Control flow opcodes (`CTRL_CALL`, `CTRL_RET`, `CTRL_LOOP`, `CTRL_NOP`) are encoded in the same word by asserting the `is_ctrl` bit, so that the same fetch/decode pipeline handles both computation and program flow. The ROM is organised into seven routines, each

occupying a contiguous 16 address slot: the scalar multiplication driver at `0x00`–`0x0A`, the point doubling formula at `0x10`–`0x1A`, the mixed affine point-addition formula at `0x20`–`0x2A`, the Jacobian to affine conversion at `0x30`–`0x3A`, the signature math at `0x40`–`0x4A`, the verification math at `0x50`–`0x5A`, and the verification check at `0x60`–`0x6A`. This ROM organisation is what enables the four way PE parallelism: the point doubling routine, for example, issues up to four independent squarings on its first instruction (register-file addresses $R_5 \leftarrow R_0^2$, $R_6 \leftarrow R_1^2$, $R_7 \leftarrow R_2^2$, $R_8 \leftarrow R_1 \cdot R_2$) that all four PEs execute concurrently, and only when all four report `pe_done` does the control unit proceed to the next micro instruction.

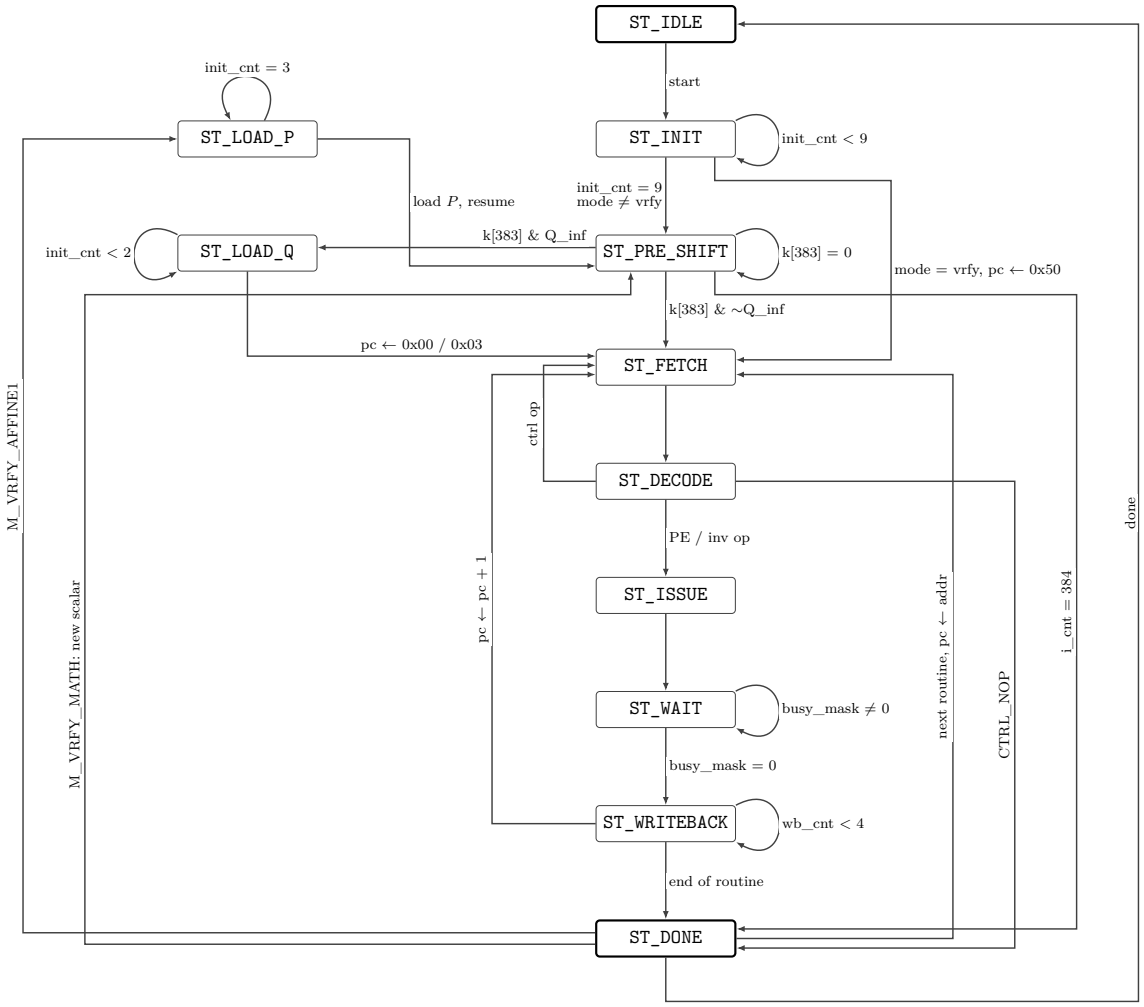


Figure 3.8: Micro sequencer FSM of the ECDSA controller (`cu_ecdsa`).

Control unit

The controller `cu_ecdsa` is organised as two nested FSMs. The inner one, `state`, implements the classical fetch-decode-issue-wait-writeback pipeline over the micro-op ROM: on every micro instruction it fetches the word at the current program counter `pc`, decodes the four PE opcodes and the optional inverter opcode, asserts the appropriate `pe_start[i]` and `inv_start` pulses, waits until the corresponding bits of a five-bit `busy_mask` are cleared by the `pe_done`/`inv_done` signals, and

then executes the writeback through the register file write port. The outer state machine, `macro_state`, tracks which stage of the ECDSA algorithm is currently in progress (`M_SIGN_PMULT`, `M_SIGN_AFFINE`, `M_SIGN_MATH` for signing; `M_VRFY_MATH`, `M_VRFY_PMULT1`, `M_VRFY_AFFINE1`, `M_VRFY_PMULT2`, `M_VRFY_MADD`, `M_VRFY_AFFINE2`, `M_VRFY_CHECK` for verification) and on each transition, sets `pc` to the entry address of the next routine. The scalar multiplication routines call the point doubling routine unconditionally and the mixed affine point addition routine only when the leading bit of the scalar shift register `k_reg` is set, which implements the left-to-right binary scalar multiplication with skip-leading-zeros optimisation of Algorithm 3. Because the whole inner loop schedule is described by a small ROM rather than by distributed hardwired state, adding a new curve or a new scalar recoding technique would require only a change to the ROM contents, not to the datapath itself.

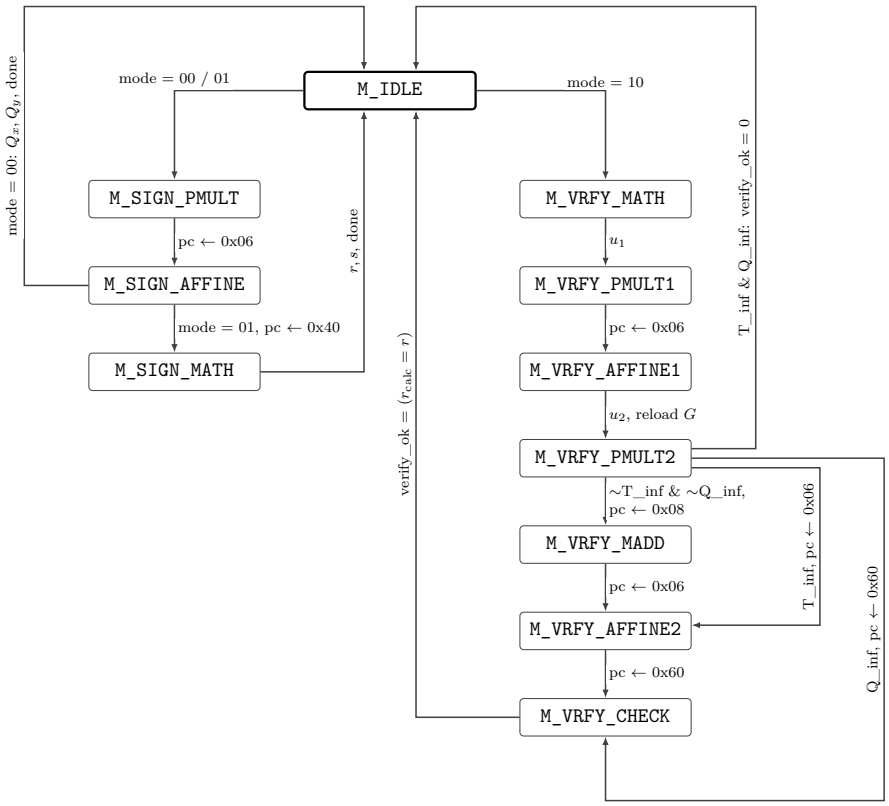


Figure 3.9: Macro-state FSM of the ECDSA controller (`cu_ecdsa`).

Operation modes

The mode signal `mode` selects between the three services supported by the ECDSA path: key generation (`mode` = 00, in which the sequencer computes $Q = [d]G$ and

exposes Q_x, Q_y), signature generation (`mode` = 01, which additionally executes the signature math routine and exposes r, s), and signature verification (`mode` = 10, which executes the verification math, two scalar multiplications, the mixed Jacobian addition, and the final check). The two signature components (r, s) are exposed on `r_out`, `s_out` at the end of the signing path, and the boolean outcome of the verification is exposed on `verify_ok`, which is the same output pin used by the HMAC verification path and is multiplexed at the top level of `sha_agile_core` by the selection signal `hmac`.

Validation Checks Delegated to the Caller

The ECDSA core implements the signature generation, verification, and key-derivation arithmetic specified by FIPS 186-5 [16], but does not enforce, in the current revision, the following validation checks that the standard prescribes for the algorithm. These checks are cheap in software and are assumed to be performed by the firmware layer that drives the accelerator. The core signals no error condition if they are omitted, and its outputs in that case are mathematically correct but not compliant with the standard.

- **Signature-component non-zero condition at signing.** FIPS 186-5 requires that if the computed r or s equals zero, the nonce k is discarded and the signing operation restarted with a fresh nonce. The signing datapath, corresponding to the ROM sequence `0x40`–`0x45` of the micro-code described in Subsection 3.2.3, computes both components unconditionally and returns them to the caller. The caller is expected to inspect (r, s) and re-issue the signing command with a new k in the rare event that either is zero.

- **Range check on incoming r and s at verification.** The verifier is required to reject the signature immediately if either component falls outside $[1, n-1]$. The core accepts whatever 384-bit integers are presented on its `r_in` and `s_in` ports and evaluates the verification equation on them. The check is left to the firmware layer.

- **Public-key validation.** The standard requires the verifier to confirm that the public key Q is not the point at infinity, that its coordinates satisfy the curve equation $y_Q^2 \equiv x_Q^3 + ax_Q + b \pmod{p}$, and that $[n]Q = \mathcal{O}$. Of these three, the first is trivially testable, the second requires a small number of field multiplications and additions, and the third amounts to a full scalar multiplication of the same order as the verification itself. None is currently implemented in the core. The on-curve check and the identity check are traditionally performed once, at the moment a public key is installed on the platform, rather than on each verification. The caller is assumed to perform this validation at key-provisioning time and to store only validated keys after.

The three checks above are simple to add in a future revision of the core. Their current delegation to software reflects an engineering decision to keep the accelerator focused on the arithmetic operations that dominate the latency of a signature, while

leaving to firmware the tests whose cost is either negligible or amortisable across many signatures.

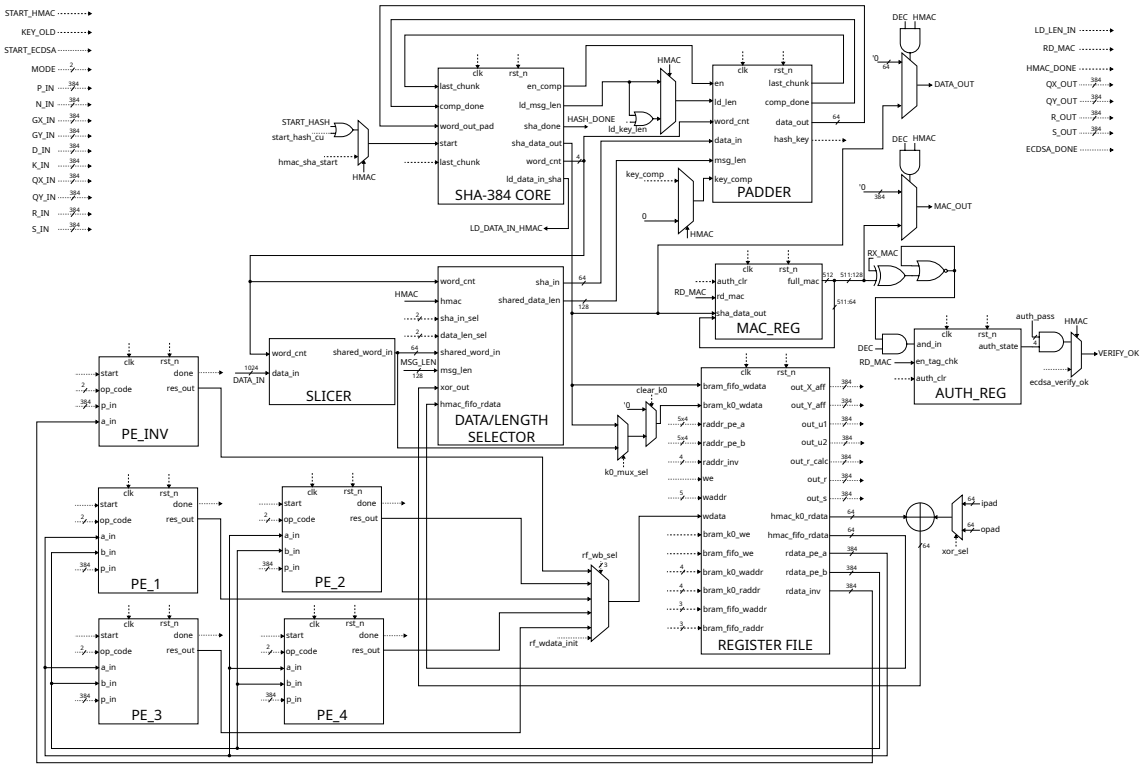


Figure 3.10: SHA agile core datapath. The CU is omitted for simplicity, every dashed signal is fed/taken from the HMAC CU and dotted signals fed/taken from ECDSA CU. Signals with uppercase names are inputs and outputs of the module. Floating signals are CU dedicated, inserted for completeness. The SHA-384 core interface is slightly different from that in Figure 3.6 due to adaptations for the new hardware.

3.3 Crypto core and final resource sharing

The two agile cores of Sections 3.1 and 3.2 are composed at the top level in a wrapper module `crypto_agile_core`, which exposes the entire accelerator through a single set of ports. Because the services are used one at a time (the protocol never invokes AES-GCM, CMAC, HMAC and ECDSA concurrently), the two agile cores are never simultaneously active, and the top level integration exploits this to merge four physical resources that are otherwise duplicated: the ipad/opad and CBC/CTR XOR gates, the storage arrays (ECDSA operand file, HMAC prepared key, HMAC inner digest, GCM H and Y_0 masks), the SHA and AES/CMAC padders, and the input/output pin lists. The result is a datapath in which every service reuses, wherever compatible, the same hardware of the others. The algorithm is selected by a mode selection signal decoded from a single configuration register `opcode_cfg`. The paragraphs below describe the main sharing strategies for the final core.

XOR sharing

Both agile cores contain a XOR computation in their datapath. In `aes_agile_core` (Section 3.1), a three input 128-bit XOR merges up to three of the following five contributions: a CMAC final block subkey K_1 or K_2, a plaintext block for GCM, an accumulated ciphertext C_{i-1} for CMAC chaining, an AES output block for GCM keystream masking and a padded message word from the preprocessor. In `sha_agile_core` (Section 3.2), a two-input 64-bit XOR merges the word of K_0 with the ipad or opad constant of Equations (2.54),(2.55). The two XOR gates are functionally distinct (128-bit vs 64-bit, three inputs vs two), but the wider one subsumes the narrower one and the two are never used in the same cycle. At the top level of `crypto_agile_core`, a single physical three-input 128-bit XOR, `shared_xor_out` is therefore instantiated. Its inputs are driven by a mode dependent MUX that selects between the AES three input configuration and the HMAC two input configuration:

$$\texttt{shared_xor_in1..3} = \begin{cases} \{K_0,\ \text{ipad/opad},\ 0\} & \text{if } \texttt{hmac_md} = 1, \\ \{\texttt{aes_xor_in1..3}\} & \text{otherwise.} \end{cases}$$

On the output side the same 128-bit result is exposed to the AES datapath and to the HMAC datapath on its lower 64 bits, so a single physical XOR implements the CMAC chaining, the GCM counter-mode masking, the CMAC final-block subkey injection and the HMAC key/pad combination.

Register File sharing

The two agile cores contain five separate storage elements:

- ECDSA operand file (32 entries of 384-bit words, with nine read ports serving the four PEs and one inverter, replicated as distributed RAM in Section 3.2.3);
- HMAC processed key buffer K_0 (sixteen 64-bit words, one read, one write port);
- HMAC inner digest buffer T_{inner} (eight 64-bit words, one read + one write port);
- GCM hash subkey register H (a single 128-bit word, one write port);
- GCM tag mask register Y_0 (a single 128-bit word, one write port).

All five are instantiated inside a single module `register_file` at the top of `crypto_agile_core`, so that the top-level datapath sees them as one wrapper with one write interface and a small number of dedicated read ports. In addition, six of the ECDSA operand-file entries (registers 13, 14, 19, 20, 21, 22, and the constant 15) are mirrored on dedicated shadow output ports that expose the affine coordinates $x_R,\ y_R$, the intermediate scalars $u_1,\ u_2$, the signature components $r,\ s$, and the reconstructed x-coordinate x_P of the verification path directly to `cu_ecdsa` without

going through the shared read ports.

Grouping the five storage elements inside the same wrapper module should not be confused with physically sharing the memory array between the three services, which would in fact be counter productive for FPGA synthesis. Two reasons prevent a beneficial outcome of merging the memories.

The first mismatch is in the number of read ports. The ECDSA Processing Elements and the inverter require nine simultaneous reads per cycle to keep the four-way PE parallelism of Section 3.2.3 busy, which is why the ECDSA operand file is instantiated nine times as separate distributed-RAM copies with a shared write port. In contrast, both HMAC buffers only expose one 64-bit word to the SHA input per cycle and the two GCM masks are read once at the end of the operation. Merging the HMAC and GCM storage into the ECDSA operand file would therefore replicate the HMAC and GCM entries nine times as well, incurring an area overhead on storage that needs only a single read port. Also, the writeback path of the merged file would have to arbitrate between the ECDSA writeback CU, the streaming HMAC key writer, and the GCM AES output, which currently reside on three independent write ports. The resulting arbitration multiplexer would slow the ECDSA writeback path without any benefit and add area comparable to the added registers.

The second mismatch is in the access pattern. ECDSA operands are accessed by random 5-bit addresses generated by the micro-op ROM, HMAC key words are streamed sequentially in and out of `hmac_k0_mem` at the SHA word counter rate and the GCM H and Y_0 registers behave as ordinary named flip flops. A common address generator that could serve all three would either be over engineered for the streaming access patterns or under engineered for the random ones. The conclusion is to keep each storage element sized to its own width, its own port count, and its own address-generation requirement, and to gather them inside the same wrapper module only for readability and top-level signal routing. The five arrays coexist inside `register_file` but remain physically independent, and the resource sharing benefit at this level is limited to reusing the same distributed RAM primitive across the three memories rather than to merging their address spaces.
This theoretical explanation is confirmed by comparing two FPGA syntheses, one with shared registers and one without. The result is more area occupation from the first solution.

3.3.1 Padder sharing

Both agile cores contain a padder that transforms an incoming bit-length message into a sequence of aligned blocks, but the padding rule differs between the four services: SHA-384 appends a single `1` bit followed by the minimum number of zero bits required to reach 896 (mod 1024) and finishes with the 128-bit length (Equation (2.42)), HMAC additionally pads or hashes the key to the block size (Equation (2.52)), GCM zero-pads the AAD and the ciphertext to 128-bit (Equation (2.22)), CMAC applies the final block rules of Equation (2.32). Despite these four different rules, all of them

reduce to the same operation: detect the word position at which the message ends, produce a right shifted mask that keeps the used bits of that word and clears the rest, and either OR in a single `1` bit (SHA and CMAC with incomplete last block) or leave it clear (GCM and CMAC with complete last block). All subsequent words up to the block boundary are zero.

So, the top level module instantiates a single `unified_padder` that implements this algorithm and dispatches the four service specific rules by two mode signals (`sha_mode` and `cmac`) and by three internal flags (`pad_en`, `one_append_en`, `one_appended`) that encode which of the four end-of-message cases applies. The shared comparators for `is_lt_1024`, `is_eq_1024` and `is_lt_896` are evaluated once and reused across services. The shared mask constants `mask_64` and `pad_bit_64` are computed once from the six low-order bits of the residual length and reused. Only the final output multiplexer is split by mode, since SHA emits 64-bit words while GCM and CMAC emit 128-bit blocks. The two output paths select the same masked word core and differ only in their word width. The result is a single padder that serves the SHA-384/HMAC padding of the hash agile core, the AAD and ciphertext alignment of GCM, the last-block masking of CMAC, and the key-padding step of HMAC, without duplicating either the length arithmetic or the mask generation across services.

3.3.2 Ports sharing

The four algorithms expose different inputs and outputs (a plaintext block for AES, a message stream for HMAC, a scalar and a curve point for ECDSA, ...) and instantiating one input port per logical operand would produce an interface too wide to be practical on the AHB/APB fabric described in Section 3.4. `crypto_agile_core` therefore exposes a small set of physically shared 384-bit and 128-bit input ports whose roles are reinterpreted for each mode. The four data ports are `data_primary_in`, `key_primary_in`, `length_in`, and `param_secondary_in`. Moreover, there are the five ECDSA-dedicated ports (`p_in`, `n_in`, `Gx_in`, `Gy_in`, `Qy_in`) exposed separately because the maximum port sharing has been exploited. The mapping between physical ports and logical operands is a set of combinational aliases, implemented as zero-extension or bit-slicing:

- `data_primary_in`: 128-bit AES plaintext block on the low half (padded with zeros on the upper half), ECDSA private key d or verification component r on the full 384 bits.

- `key_primary_in`: 256-bit AES key on the low half (padded on the upper half), ECDSA k or signature component s on the full 384 bits.

- `length_in`: 64-bit AES message length on the low half, 64-bit AAD length on the upper half, 128-bit SHA/HMAC message length on the full width.

- `param_secondary_in`: 96-bit GCM initialisation vector on the low quarter, ECDSA public key Q_x on the full 384 bits, HMAC received tag rx_mac on the top 384 bits.

Shared Port	AES / GCM / CMAC	SHA / HMAC	ECDSA
`data_primary_in`	`data_in` [127:0]	`data_in` [127:0]	`d_in`, `r_in` [383:0]
`key_primary_in`	`key_aes` [255:0]	—	`k_in`, `s_in` [383:0]
`length_in`	`aes_msg_len`, `aad_len` [63:0]	`msg_len` [127:0]	—
`param_secondary_in`	`iv` [95:0], `rx_mac` [383:0]	`rx_mac` [383:0]	`Qx_in` [383:0]
`data_primary_out`	`aes_data_out_int` [127:0]	`hmac_mac` [383:0]	`Qx_out_int`, `r_out_int` [383:0]
`data_secondary_out`	`tag_out_reg` [127:0]	0	`Qy_out_int`, `s_out_int` [383:0]

Table 3.1: Mapping of top-level shared I/O ports to internal submodules.

The mode selector `opcode_cfg` is a 32-bit configuration register in which each bit encodes a service or a service parameter: `opcode_cfg[0]` selects AES, `[1]` GCM, `[2]` CMAC, `[3]` SHA, `[4]` HMAC, `[5]` ECDSA, `[6]` a key-load pulse, `[7]` an AAD-enable, `[9:8]` the ECDSA submode (keygen / sign / verify), and `[10]` a decrypt/verify flag. The two output ports `data_primary_out` and `data_secondary_out` are similarly multiplexed by the same mode signals: they carry the AES ciphertext and GCM tag in AES modes, the CMAC/HMAC tag in the authentication modes, and the pair (r, s) or (Q_x, Q_y) in the ECDSA modes. The scalar status output `done` is also multiplexed among the four algorithms' completion signals (`aes_done_out`, `sha_done`, `hmac_done`, `ecdsa_done`), so that the NOEL-V-based SoC of Section 3.4 sees a single handshake protocol regardless of the service actually running. A single flip flop `auth_state`, backs the shared `verify_ok` output for the CMAC, GCM, HMAC, and ECDSA verification paths. The flip flop is cleared to `AUTH_FAIL` whenever a new operation starts, so no residual verification result can survive from one operation to the next.

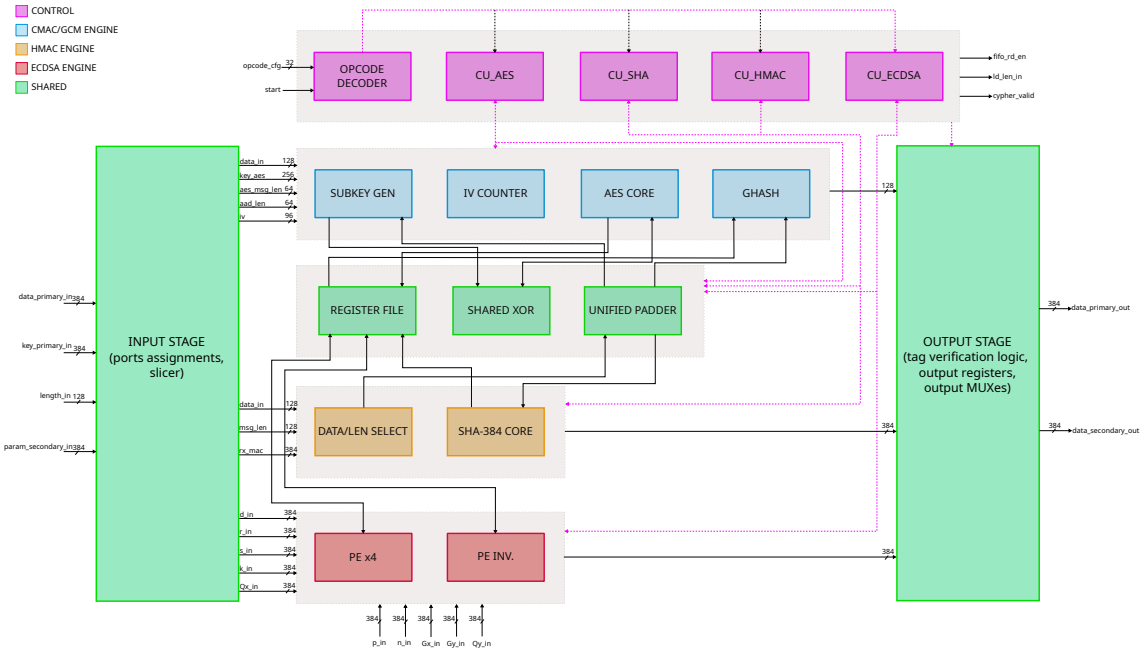


Figure 3.11: Final agile crypto core datapath. Notice that the colored sub-blocks in gray rectangles are intrinsically connected to each other. The detailed routing is omitted for the sake of readability, knowing that the main engine architectures are illustrated in Figures 3.4 and 3.10.

3.4 NOEL-V integration

The `crypto_agile_core` described in Section 3.3 exposes the already described interface that is not directly connectable to the narrow AMBA busses of the NOEL-V-based SoC. Its integration therefore requires a wrapper that exposes a memory mapped register file to the processor for control and status, a DMA engine that moves data between main memory and the accelerator and to be presentable as a Plug&Play (PnP) device like the others of the GRLIB library. The wrapper is organised as a small VHDL entity `crypto_core_top` that exposes the AMBA PnP interface and adapts the `grlib.amba` signal types to the flat signal lists expected by SystemVerilog, and a SystemVerilog module `bus_to_core` that instantiates inside it the three main blocks of the accelerator: the memory mapped register file, the AHB master DMA engine and the crypto core itself. Control/status and data are physically separated onto the APB and AHB busses of the SoC as described in Section 3.4.1, so that the processor can configure and check the accelerator status through low bandwidth register accesses while high bandwidth data transfers are exploited for maximum speed.

3.4.1 APB/AHB buses

The GRLIB `grlib.amba` package provides two AMBA busses to which every peripheral in a NOEL-V system attaches: an AHB bus for high bandwidth burst transfers (128-bit data on the treated configuration) and an APB bus for low bandwidth configuration accesses (32-bit wide). The accelerator uses both, but for different purposes. Control, status and pointers registers are exposed on the APB slave port of `crypto_core_top`, so that ordinary `sw`/`lw` instructions on the RISC-V core can configure the accelerator without occupying the AHB for few bits transfers. The register file appears at the base address `paddr` with a size determined by `pmask` (the generic values used in the target integration are 16′h019 for the device identifier and 16′hFFF for the mask, mapping the entire register file into a single 4 KB APB slot). The wide data transfers are instead performed by an AHB *master* instantiated inside `crypto_core_top`, connected to the shared AHB interface at the PnP index `hindex`. The DMA autonomously arbitrates for the bus through `hbusreq`/`hgrant` and moves 128-bit beats between the accelerator and the AHB memory that stores the plaintext, ciphertext, keys, ECDSA parameters and tags.

The PnP identity of the entity is set through two configuration structs generated from the constants of the `crypto_agile_config_pkg` package. The APB slave record `pconfig` declares the vendor (`VENDOR_CRYPTO_AGILE = VENDOR_CONTRIB`), the device identifier (`DEVICE_CRYPTO_AGILE = 16#019#`) and a revision, together with the memory bar produced by `apb_iobar` from `paddr` and `pmask`. The AHB master record `hconfig` declares the same vendor and device identifier. Both records are exposed on the respective bus output records of `crypto_core_top`.

3.4.2 Interface Registers

The interface register file is generated with an `hjson` file by means of the `regtool` tool of the lowRISC OpenTitan project, whose textual description of the register map is compiled into two SystemVerilog files: a package `crypto_agile_reg_pkg.sv` that defines the typed `reg2hw` and `hw2reg` structures visible to the hardware, and a top-level module `crypto_agile_reg_top.sv` that implements the registers interface.

Fifteen registers are visible to the processor. The single-bit `START` register at offset `0x00` is the launch trigger: a rising edge on its `q` output, detected by an edge detector in `bus_to_core`, produces a one-cycle `start_pulse` that enables the DMA. The 32-bit `CTRL` register at offset `0x04` groups all mode selection bits into a single word: six exclusive service selectors (`aes_md`, `gcm_md`, `cmac_md`, `sha_md`, `hmac_md`, `ecdsa_md`), the `ld_key` and `en_aad` flags, a two-bit ECDSA sub mode (`mode_ecdsa`), and a decrypt flag (`decrypt`). These fields are concatenated combinationally into the `flat_opcode` signal that drives the `opcode_cfg` input of `crypto_agile_core`, connecting the software programming model and the hardware mode selection logic of Section 3.3.2. The `STATUS` register at offset `0x08` is read only and updated from the hardware side through the `hw2reg` bus. It carries the four status bits `fifo_rd_en`, `ld_len_in`, `done`, and `verify_ok`, latched by `bus_to_core` until either a new start pulse or an explicit clear from the software resets the `done` flag. The remaining twelve registers, from `data_primary_in_ptr` at offset `0x0C` through `data_secondary_out_ptr` at offset `0x38`, are 32-bit pointer registers whose contents are accessed only by the DMA engine of Section 3.4.3. Each pointer identifies the base address in main memory of one of the input or output buffers that the DMA will move during operation. This architecture keeps the processor's programming model simple: the driver populates the pointer registers with the addresses of the relevant buffers, configures the mode through `CTRL`, and writes `START`, while the DMA and the accelerator carry the operation to completion without further software intervention.

Bit(s)	Field	Meaning
0	`aes_md`	Select AES-CTR service
1	`gcm_md`	Select AES-GCM service
2	`cmac_md`	Select AES-CMAC service
3	`sha_md`	Select SHA-384 service
4	`hmac_md`	Select HMAC-SHA-384 service
5	`ecdsa_md`	Select ECDSA-P384 service
6	`ld_key`	Key-load pulse
7	`en_aad`	Enable AAD stage (GCM)
9:8	`mode_ecdsa`	00 keygen / 01 sign / 10 verify
10	`decrypt`	Decrypt / verify flag

Table 3.2: CTRL register field layout.

3.4.3 DMA

The DMA ahb_dma_master is an AHB master module instantiated inside bus_to_core that moves every operand and result between the AHB memory and the accelerator. It exposes to the AHB a single 128-bit master port with the standard hbusreq/hgrant handshake and to the accelerator side an operands interface: 384-bit core_key_in, core_param_in, core_p_in, core_n_in, core_Gx_in, core_Gy_in, core_Qy_in registers, a 128-bit core_length_in bus, a streaming core_data_in port fed from an internal input FIFO. The engine is organised around two independent structures: a job scheduler that iterates an enumerated type job_e and a pair of 16-deep 128-bit FIFOs that decouple the input and output paths from the AHB bursts, so that the accelerator can consume and produce blocks at its own rate without stalling the bus.

Job scheduler

When the pulse start_dma goes up, the scheduler enters the setup state with job = JOB_KEY and, at each successful completion, advances to the next job in the sequence produced by the function next_cfg_job. For the symmetric services the sequence is KEY→KEYLEN→MSGLEN →PARAM→DATA. For ECDSA the five additional curve parameter jobs P, N, GX, GY, QY are inserted between PARAM and DATA. A pointer whose value is zero is interpreted as "not applicable" and the corresponding job is skipped, so the same hardware handles the different operands between modes without a mode-specific FSM branch. Every configuration job is a 384-bit or 128-bit burst (HBURST_INCR of one to four 128-bit beats) whose incoming data is routed by a DEMUX into the corresponding core input register. On completion the scheduler goes back to the setup state to launch the next job.

Streaming FIFOs

Once all configuration operands have been fetched, the scheduler enters the streaming state S_RUN, in which the DMA and the accelerator run concurrently. The input FIFO is filled from RAM in 128-bit wide HBURST_INCR4 bursts of four beats whenever its fill level falls below the REFILL_LEVEL threshold and the message pointer has not yet reached the end of the input buffer. The core reads one 128-bit block from the FIFO whenever it asserts fifo_rd_en. To eliminate the risk of crypto core stalling on the first block, the scheduler does not raise core_start until the FIFO has been prefilled up to the higher PREFILL_LEVEL threshold. From that point on, the FIFO depth of sixteen entries is large enough to absorb the latency of an AHB burst. On the output side, the FIFO is filled from the accelerator whenever core_cypher_valid is asserted and drained toward memory either at a low DRAIN_LEVEL or, if the accelerator is producing blocks faster than the AHB can accept them, at the higher URGENT_LEVEL. Underrun and overflow diagnostic flags dma_underrun and dma_overflow are latched on the first violation of these two invariants and exposed on the DMA output for analysis purposes.

Result writeback and completion

When the accelerator raises the `done` flag, the scheduler waits for the output FIFO to drain for a short `drain_wait` delay to allow the last few blocks to propagate through the FIFO logic. Then, it issues either the pair of writeback jobs `JOB_OUT_PRI`/`JOB_OUT_SEC` for services that produce two 384-bit outputs (the ECDSA signature (r, s) or the AES ciphertext and the GCM authentication tag) or only the secondary job when the primary output was already streamed through the output FIFO. A 3-bit counter `busy_tail` keeps `dma_busy` asserted for a few cycles after the return to `S_IDLE`, so that the top-level `done` bit visible in the `STATUS` register is asserted only after all running AHB beats have been completed.

AHB protocol details

Every burst is a 128-bit-wide transfer (`hsize = HSIZE_128`), which matches both the AHB data width of the target NOEL-V configuration and the block size of the symmetric primitives (a 128-bit AES state, a 128-bit GCM operand). The pipelined AHB address/data protocol is followed exactly: the address phase asserts `HTRANS_NONSEQ` on the first beat of each burst and `HTRANS_SEQ` on the subsequent ones, and the data phase is delayed by one `hready` cycle with respect to the address phase, with a small internal state (`a_onbus`, `d_beat`, `d_valid`) tracking the two phases independently. Loss of grant is handled by asserting the `resume_nonseq` flag, so that the first beat after reacquiring the bus is marked as `HTRANS_NONSEQ`. Non-cacheable data is declared by driving `hprot = "0011"` at the wrapper level, which is the standard GRLIB code for privileged data accesses. The data has to be non-cacheable because of cache coherence: when DMA accesses some cacheable addresses in RAM, expected data might not be present because it has not yet been flushed from the cache.

3.4.4 Cacheable simulation memory

The default simulation memory of GRLIB, `ahbram_sim`, always advertises its address space as cacheable in its Plug&Play memory bar, which is incompatible with the DMA driven flow of the accelerator: if the RISC-V core's data cache holds an old copy of a buffer that the DMA is about to consume, the accelerator will operate on incorrect data. The simulation flow used in this work modifies `ahbram_sim` into a new entity `ahbram_sim_cacheable`, like in the section *Memory model* of Master's thesis [21], that exposes an additional `cacheable` generic parameter and propagates its value into the third bit of the `ahb_membar` call that constructs the PnP memory bar. When the entity is instantiated for the DMA buffer with `cacheable = 0`, the memory bar marks the region as non-cacheable, so the processor's data cache does not access it. The entity is a full copy of `ahbram_sim` and is wrapped in a `pragma translate_off` block so that it does not enter the synthesis flow. In the simulation-only branch of `noelvmp.vhd`, a 64 kB instance is placed at the AHB memory region `16#E00#000` and dedicated as the DMA buffer.

3.4.5 NOEL-V source-tree modifications

Integrating the accelerator into the NOEL-V template requires a minimal set of changes to the standard GRLIB source files, none of which alters the behaviour of the processor. All modifications are localised in the file `noelvcore.vhd`, which describes the top-level SoC generator, and to the platform wrapper `noelvmp.vhd`, which describes the target platform.

The wrapper `noelvcore.vhd` is extended with a new AHB slave signal pair, `cryptomem_ahbsi` and `cryptomem_ahbso`. These signals are routed as an ordinary AHB slave slot inside the module at the index `CRYPTOMEM_HSINDEX`, following the same pattern used for the on-chip ROM. At the bottom of the same file, the accelerator itself is instantiated as

```
u_crypto_core:  entity contrib.crypto_core_top
```

with the generics `pindex`, `paddr`, `pmask`, `hindex` and `pirq` taken from constants added to `cfgmap.vhd` (`CRYPTO_PINDEX`, `CRYPTO_PADDR = 16#001000#`, `CRYPTO_PMASK = 16#FFF#`, `CRYPTO_HMINDEX`, `CRYPTO_PIRQ`), so that the register file appears in the low 4 kB region of the APB slot allocated to the peripheral. A small guard in the APB generation loop (`un_apb :  if i /= CRYPTO_PINDEX generate`) prevents the default GRLIB APB template from overwriting the slot. In the wrapper `noelvmp.vhd`, the signals wiring for `cryptomem_ahbsi/cryptomem_ahbso` is added and, in the simulation branch, the `ahbram_sim_cacheable` is instantiated. Because the Plug&Play mechanism marks the accelerator at synthesis time through the two configuration records of `crypto_core_top`, no further changes are required: the software driver discovers the accelerator by scanning the PnP table for the `VENDOR_CONTRIB` / device `16#019#` pair and retrieves the register file base address directly from the corresponding APB memory bar.

3.5 Custom API (Application Programming Interface)

The accelerator described in Sections 3.3 and 3.4 is controlled by the RISC-V core through a memory mapped programming interface (an API) that consists of the fifteen APB registers of Section 3.4.2 plus a set of memory buffers held in the AHB region introduced in Section 3.4.4. From the software point of view, the accelerator is a launch-and-forget peripheral: the driver sets the register file with a mode word and pointers, pulses a start bit and waits for a completion flag. All wide-operand transfers, all handshakes with the crypto core and all bus arbitration are handled autonomously by the DMA. This section describes the software side of the interface.

3.5.1 Register map from the software side

The APB register file appears at the base address `0xFF409000` chosen at SoC generation time (see Section 3.4.5) and the fifteen 32-bit registers of Section 3.4.2 are exposed to C code by a set of `volatile uint32_t` macros in the `crypto_config.h`

header. Three of these addresses are used for control: REG_START at offset 0x00 to launch a new operation, REG_CTRL at offset 0x04 to select the service and its modifiers, REG_STATUS at offset 0x08 to observe completion and verification outcome. The twelve remaining registers, from REG_DATA_IN_PTR at 0x0C through REG_DATA_OUT_SEC_PTR at 0x38, are pointer registers that carry the AHB addresses of the operand buffers that the DMA will fetch and store. The selection bits of REG_CTRL are encoded as one-hot service selectors by setting them bit by bit, matching the composition of opcode_cfg performed inside bus_to_core and described in Section 3.4.2. The completion and verification flags in REG_STATUS occupy dedicated bit positions (STATUS_DONE_BIT and STATUS_VERIFY_OK_BIT) so that the driver can poll each independently with a bitwise AND.

3.5.2 Operation sequence

Programming the accelerator from software reduces to four steps, implemented in the reference driver by the helpers init_crypto_pointers and run_crypto_polling. The first step, executed once at driver initialisation time, populates the twelve pointer registers with the addresses of the fixed operand buffers that the DMA will use for the whole session: the software does not update these pointers between individual operations, so a sequence of encryptions or hashes on the same buffers reuses the setup at no additional bus cost. The second step, executed per operation, writes the mode word into REG_CTRL. The third step, per operation, pulses REG_START through the sequence 0→1→0, producing the rising edge that the edge detector in bus_to_core converts into the one-cycle start_pulse of Section 3.4.2. The fourth step is a wait loop that polls REG_STATUS until the STATUS_DONE_BIT rises, which means that the results are available in the output buffers pointed to REG_DATA_OUT_PRI_PTR and REG_DATA_OUT_SEC_PTR. The polling loop consists of a single nop between reads, which is cheap enough to be acceptable in the current single-core RISC-V configuration.

3.5.3 Buffer allocation, alignment, and cache coherence

Every buffer visible to the DMA is declared with the C attribute __attribute__((aligned(16))) and the volatile qualifier. The sixteen-byte alignment matches the 128-bit AHB data width used by the DMA and, in particular, aligns the streaming buffers on the four-beat HBURST_INCR4 boundary discussed in Section 3.4.3. The volatile attribute prevents the compiler from caching buffer contents and reading junk values after the accelerator has updated them. On top of these two attributes, the two output buffers are placed at the fixed AHB addresses 0xE0000000 (DMA_OUT_PRI) and 0xE0001000 (DMA_OUT_SEC) inside the non-cacheable region in ahbram_sim_cacheable instance of Section 3.4.4. Because that region is marked as non-cacheable in the P&P memory bar, the processor's data cache never allocates a line for it and no explicit cache flush instruction is required between the software write of the input buffers and the DMA read that follows, or between the DMA write of the output buffers and the software read. A single fence rw, rw memory barrier before the REG_START pulse is set to guarantee that

all memory operations have been concluded before performing a new accelerator call.

3.5.4 Operand encoding

Wide operands (128-bit keys, 256-bit AES keys, 384-bit ECDSA parameters, 128-bit length fields) are defined in memory as arrays of 32-bit words in little endian order: the least significant 32-bit slice of the operand occupies the lowest address and each subsequent slice occupies an increasing address. This is the little endian layout of a C `uint32_t` array on the RISC-V target and matches the beat order in which the DMA fetches 128-bit AHB beats out of memory. For example, the prime modulus p of Equation (2.57) is exposed as the array

```
dma_p = {0xFFFFFFFF, 0x00000000, 0x00000000, 0xFFFFFFFF,
0xFFFFFFFE, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF},
```

which can be verified against $p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$ by inspecting the positions at which the two slices `0x00000000` and `0xFFFFFFFE` appear in the sequence. The base-point coordinates x_G, y_G and the subgroup order follow the same layout, so the driver never performs any endianness conversion between the FIPS 186-5 hexadecimal representation of the domain parameters and the memory seen by the accelerator. Length fields (message length, key length, AAD length) are laid out as 128-bit little endian integers in four consecutive 32-bit words, which is the width of the internal `msg_len_reg` of the unified padder of Section 3.3.1.

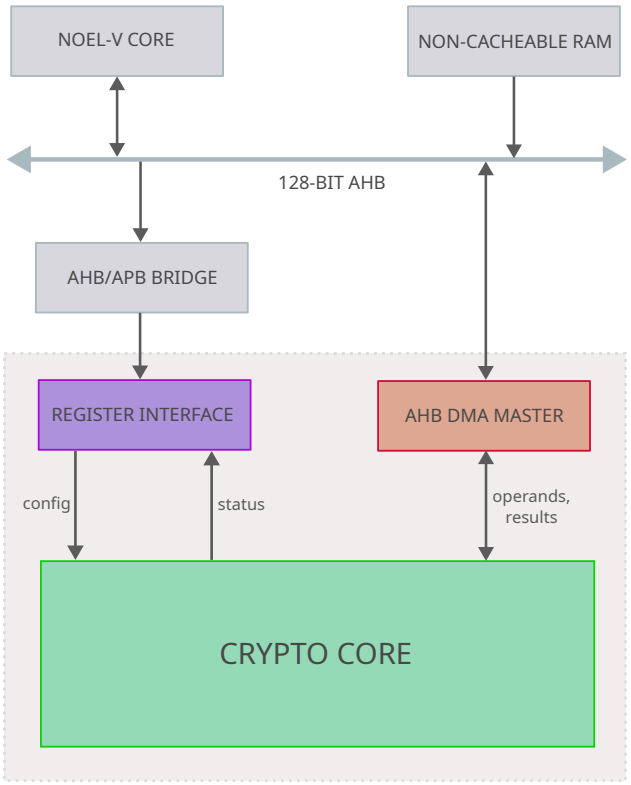


Figure 3.12: Crypto core integration scheme in NOEL-V SoC.

4 RTL simulation, FPGA synthesis and results

In this chapter functional testbenches and benchmarks are presented, then the obtained results are analyzed in three steps: Section 4.3.1 places the four standalone cores of Table 4.1 against the single-primitive references of Table 4.3, Section 4.3.1 places the agile accelerator against the same references, Section 4.3.3 reports the internal comparison against the non-shared core `mods_wrapper` of Tables 5.1 and 4.4, which isolates the area contribution of the sharing strategy of each module.

4.1 General testbenches and synthesis

The RTL of this work is verified through two families of SystemVerilog testbenches that share the same overall shape but pursue different goals: a set of *functional* testbenches that check bit-accurate correctness of every service against Known-Answer-Test (KAT) vectors, and a set of *performance* benchmarks that measure sustained throughput and per-operation latency once the RTL is known to be correct. The RTL designs have been simulated with **Verilator 5.032**.

4.1.1 Functional testbenches

Each module of the analyzed accelerator, from the primitive cores to the final wrapper, has a standalone testbench, which has been used extensively for testing the correct functioning. The testbenches of the two agile cores and the final one, follow the same template: they instantiate the DUT, drive its input ports through the same combinational composition performed by `bus_to_core` in the integrated system, and emulate the DMA streaming path with a testbench-side 128-bit-wide array drained on `fifo_rd_en` pulses and filled by two helper tasks (`push_fifo` and `reset_fifo`) that operate with wr/rd pointers. Each service is exercised by a dedicated test task (for example `test_gcm_enc`, `test_cmac`, `test_hmac`, `test_ecdsa`, ...) that configures the mode bits, loads a KAT vector, launches the operation, and delegates the outcome comparison to a `check_result` task. Verification modes are tested with both positive and negative (bit-flipped) vectors, so that a mis-triggered `verify_ok` in either direction is caught by the same code. Each mismatch increments a global error counter, and the top-level `initial` block prints a pass/fail summary at the end. The reference vectors are taken directly from the relevant NIST publications and from open-source crypto projects online.

4.1.2 Synthesis

The accelerator has been synthesized on a **Kintex-7 xc7k410tfbg676-2** chip, with software **Vivado 2019.2**. A `.tcl` script iterates over all modules, synthesizing, optimizing, and placing-and-routing the designs. For each module, up to 8 place-and-route trials are made to find the correct F_{max}. The script starts every time from an RTL checkpoint made after the design optimization. A passing trial tightens the period by $0.7 \cdot WNS$, a failing one loosens it by $1.1 \cdot |WNS|$, and once both a passing and a failing period are known, bisection takes over between them. The algorithm stops when the two bounds converge within 50 ps. Then, area and timing report files are saved, and a summary of all modules results is printed in a `.csv` file. All circuits are synthetised with Vivado option `-mode out_of_context` and `-flatten_hierarchy none` or `rebuilt` for better modules cross-boundary optimization. Table 4.1 shows the synthesis results, including the core `mods_wrapper` that unifies the cores without the sharing strategies discussed in Section 3.3.

Design	LUT	FF	Slices	CARRY4	BRAM	F_{max} (MHz)
Standalone cores						
`cmac_core`	3,239	684	922	25	0	285.25
`gcm_core`	11,972	2,777	2,605	54	0	156.14
`hmac_core`	5,305	3,138	1,514	166	0	157.70
`ecdsa_core`	31,945	17,546	11,064	2,200	2	123.00
Arithmetic sum	52,461	24,145	16,105	2,445	2	—
Shared cores						
`aes_agile_core`	13,115	3,098	2,856	54	0	177.43
`sha_agile_core`	33,507	18,071	11,505	2,257	2	122.73
Top level						
`mods_wrapper`	51,759	23,752	15,391	2,445	4	122.14
`crypto_agile_core`	45,947	21,308	13,714	2,263	2	125.03

Table 4.1: Resource utilisation and maximum clock frequency of the standalone cores, the two intermediate shared cores, and the fully integrated accelerator.

4.1.3 Performance benchmarks

Like the testbenches, each module has an associated benchmark, measuring the throughput after each operation. The structure is minimal: the clock period is fixed at compile time to the value obtained from the FPGA timing-analysis reports of Section 4.1.2 so that the reported figures correspond to the actual maximum operating frequency of the silicon rather than to an arbitrary simulation clock. Because the goal is timing and not correctness, the input data itself is irrelevant: the AES benchmark drives a simple 128-bit counter whose value is fed back into the streaming input on every `fifo_rd_en` pulse, and the HMAC and ECDSA benchmarks reuse fixed 1024-bit and 384-bit constants. The measurement task `run_benchmark` samples `$realtime` at the assertion of `start` and again at the assertion of `done`, divides the

processed payload by the elapsed time, and prints the result in Gbps or in operations per second. Throughput results are shown in Table 4.2.

			Throughput	
Service	**Block**	**Cycles**	**Unified (125 MHz)**	**Standalone**
AES-256-CTR	128 b	18	888.9 Mbps	1,110.3 Mbps
AES-256-GCM (encrypt)	128 b	18	888.9 Mbps	1,110.3 Mbps
AES-256-GCM (auth. only)	128 b	5	3,200.0 Mbps	3,997.2 Mbps
AES-256-CMAC	128 b	18	888.9 Mbps	2,028.4 Mbps
HMAC-SHA-384	1024 b	88	1,454.5 Mbps	1,835.1 Mbps
ECDSA-P384 key gen.	—	2,557,069	20.46 ms (48.9 ops/s)	20.79 ms (48.1 ops/s)
ECDSA-P384 sign	—	2,601,529	20.81 ms (48.1 ops/s)	21.15 ms (47.3 ops/s)
ECDSA-P384 verify	—	5,174,994	41.40 ms (24.2 ops/s)	42.07 ms (23.8 ops/s)

Table 4.2: Measured performance of all modules. Cycle counts are obtained by RTL simulation which excludes fixed setup cost. Throughput is scaled to the post-route $F_{\max}$ of 125 MHz for the final core or to the obtained $F_{\max}$ for the standalone module.

4.2 Software tests

The final accelerator integrated in NOEL-V SoC described in Section 3.4, has been tested with a simple `.c` testing program. The SoC testbench has been launched with **Questasim 2021.2_1**.

4.2.1 Test file

The `crypto_test.c` file is structured as a self-checking KAT bench that uses a table of test vectors declared in `crypto_vectors.h`. Each vector is a `crypto_vector_t` record that carries the opcode word, the input operands (`key`, `param`, `data`, `msg_len`, `key_len`), the expected primary and secondary outputs (`exp_pri`, `exp_sec`) with their offsets and lengths, and a reference name. The driver iterates over the table with `run_vector`, which loads the input buffers via `load_buf`, pre-loads the two output buffers with a debug purpose pattern `0xDEADBEEF` to detect DMA transactions that failed to occur, issues the `fence rw,rw` barrier, launches the operation, and compares the resulting `DMA_OUT_PRI` and `DMA_OUT_SEC` buffers against the expected outputs using `check_buf`. On a mismatch, the byte-level differences are emitted through the simulation trace device (`report_subtest`, `fail`) mapped at `0x80000000`, so that the failure is visible in the Questasim transcript without depending on any specific driver output channel. On completion of all vectors without error, the driver reports the special success marker `0xC0DE` and then writes to `SIM_END_TEST` at `0x80000014` to terminate the simulation.

4.2.2 Verification mode result handling

When the current operation is a verification (`OP_DECODE` bit set), the driver interprets the `STATUS_VERIFY_OK_BIT` in `REG_STATUS` as a boolean outcome that must be checked against the expected result of the test. The flag is driven from the top level `verify_ok` output of `crypto_agile_core`, which is the shared output port for all verification services. Because verification is a security-relevant operation, the flag is latched by `bus_to_core` until the next start pulse clears it, so a missed poll on `REG_STATUS` cannot leak an `AUTH_PASS` outcome from one operation into the next.

4.3 Results

4.3.1 Standalone cores versus the state of the art

Work	Platform	Area	F_{MAX} (MHz)	Throughput
ECDSA P-384				
Kieu-Do-Nguyen *et al.*	Virtex-7	10,259 slices	112.2	47 sign/s 23 vfy/s
This work `ecdsa_core`	Kintex-7	11,064 slices	123.0	47.3 sign/s 23.8 vfy/s
SHA-2 / HMAC				
Juliato & Gebotys	Virtex-II	6,247 slices	56.21	355.3 Mbps
This work `hmac_core`	Kintex-7	1,514 slices	157.7	1,835.1 Mbps
AES and AES-based modes				
Lin & Chuang (GCM)	Virtex-5	5,944 slices 109 BRAM	223.3	57,170 Mbps
Nannipieri *et al.*	7 nm TSMC	131.45 kGE	2550.0	23,310 Mbps
Nannipieri *et al.*	ZCU106	15,694 LUT 9,925 FF	200.0	1,828.6 Mbps†
This work `aes_agile_core`	Kintex-7	13,115 LUT 3,098 FF	177.43	1,261.7 Mbps
Unified accelerator				
This work `crypto_agile_core`	Kintex-7	13,714 slices	125.03	888.9 Mbps (AES) 3,200.0 Mbps (GHASH) 1,454.5 Mbps (HMAC) 48.1 sign/s, 24.2 vfy/s

Table 4.3: Comparison with the reference implementations. Throughput is the steady-state rate; setup latency is excluded in all rows. Figures marked † are derived from the cited cycle count and clock frequency, not reported directly by the authors.

Before entering the numerical comparison it is worth noting that the references in Table 4.3 span four generations of Xilinx FPGA silicon, from Virtex-II (2001) to Zynq UltraScale+ (2016), plus one ASIC point. The design in this work targets Kintex-7, a 28 nm 7-series device. Direct area and frequency comparisons across

generations must be read with care: newer families provide six-input LUTs, denser distributed-RAM primitives, wider carry chains and higher achievable clock rates, all of which move the numbers before any architectural change is made. The following paragraphs identify, for each reference, whether the FPGA gap works for or against the obtained results.

The ECDSA comparison is essentially technology-fair. Kieu-Do-Nguyen *et al.* [5] target a Virtex-7 device, which sits in the same 7-series generation as the Kintex-7 target and offers comparable slice and DSP primitives, so differences between them depend mostly on the microarchitecture rather than on the silicon. The implementation follows their architecture almost exactly and the measured results reflect that lineage: `ecdsa_core` occupies 11,064 slices against their 10,259, a 7.9 % increase, sustains 123.0 MHz against 112.2 MHz, and closes 47.3 signatures per second and 23.8 verifications per second against their 47 and 23. Neither design infers any DSP block.

The HMAC-SHA-384 comparison against Juliato & Gebotys [4] is the least technology-fair in the table and the one that most requires disclosure. Their design targets Virtex-II, a device family from the early 2000s that did not feature the six-input LUT, the CARRY4 primitive and the distributed-RAM macros used for wide memories such as the HMAC padding buffer. Every one of these features reduces the slice count required for the same design on a modern part, and a substantial fraction of the fourfold slice reduction reported here (1,514 against 6,247 slices) is attributable to the technology change rather than to architectural improvement. The frequency gap of 157.7 MHz against 56.2 MHz is likewise partly a consequence of the newer process. What the comparison does establish, once the technology contribution is acknowledged, is that the compression datapath of this work is at least as compact as a straightforward Virtex-II implementation would be if remapped to Kintex-7, and that the throughput of 1,834.9 Mbps against 355.3 Mbps is also credible.

The three AES reference points measure different things on different platforms. Lin & Chuang [3] target Virtex-5, two generations older than Kintex-7. Both families use six-input LUTs, so the essential difference lies in the slice organisation and the LUT-to-flip-flop ratio rather than in the primitives themselves. The technology gap is much smaller than the Virtex-II case discussed above. What makes their result architecturally distinct is the fully pipelined AES with pre-expanded round keys stored in 109 block RAMs, delivering one block per cycle at 223.3 MHz. The BRAM count is itself a substantial hidden area cost that this design avoids and the architecture converts LUT-based key expansion into memory lookups. The two designs occupy different points in the area–throughput trade-off space and are not directly comparable in either dimension. Nannipieri *et al.* [6] report 131.45 kGE at 2.55 GHz on 7 nm TSMC standard cells, which is orthogonal to any FPGA figure and included in the table only for completeness. Their ZCU106 prototype is the fair reference: the ZCU106 board carries a Zynq UltraScale+ MPSoC, a 16 nm FinFET device one full generation newer than the 28 nm Kintex-7 target used here, which would be expected to deliver both a higher clock rate and denser logic packing for the same RTL. The fact that `aes_agile_core` occupies 13,115 LUTs against their 15,694, a 16 % reduction, at 177.4 MHz against 200.0 MHz, only 11 % lower, on the older silicon is a stronger result than the raw numbers suggest, and their throughput

advantage of 1,828.6 Mbps against 1,261.7 Mbps is entirely accounted for by the architectural cycle count. Inverting their throughput yields exactly 14.0 cycles per AES-256 block ($2.55 \cdot 10^9 \cdot 128/23.31 \cdot 10^9$), one round per cycle. The treated agile core spends 18 cycles per block because the finite-state machine that arbitrates GCM, CMAC and CTR modes introduces four cycles of set-up and drain per block.

4.3.2 Unified accelerator versus the state of the art

Direct comparisons for the unified accelerator do not exist in the literature. SHARMONY [8] unifies two hash primitives from the same family, Nannipieri *et al.* [6] unify block-cipher modes of a single symmetric algorithm, and no prior work covers a symmetric primitive, a hash primitive and a public-key signature scheme in a single datapath. The F_{max} attainable by `crypto_agile_core`, 125.0 MHz, is set by the ECDSA modular arithmetic datapath and applies to every service under the single clock domain constraint. This immediately caps the achievable throughput per service below the corresponding standalone figure: 888.9 Mbps for AES-CTR, AES-CMAC and AES-GCM payload throughput, 3,200.8 Mbps for the GHASH multiplier alone, 1,454.5 Mbps for HMAC-SHA-384, and 48.1 signatures per second and 24.2 verifications per second for ECDSA-P384. Despite the throughput reduction, no service degrades in cycles per block. The unified frequency accounts for the entire performance gap. SHARMONY reports a whole design area reduction of 40 % in LUTs and 55 % in flip-flops against a set of external combined designs from OpenTitan, Caliptra, SLotH and SPHINCSLET, all implemented on 7-series Xilinx parts. This work reports 11.2 % and 10.3 % against `mods_wrapper`. The two designs therefore share the target family, so the difference in the headline percentage is not due to technology: both sets of numbers reflect the same generation of silicon and the same primitive library. The comparison instead turns on the choice of baseline and on the scope of what is unified. Their baseline is the arithmetic combination of third-party IPs that carry input buffers, parallel-load state interfaces and IP-integration overhead that a freshly written compact core does not need to carry. Their own controlled ablations against their own design, reported in the same paper, land in single digits: shared control logic saves 1.9 % of LUTs, moving round constants from LUT-ROM to BRAM saves a further 3.0 %, and unified padding saves 32.6 % locally, corresponding to roughly 3.5 % of the whole design. These are the numbers directly comparable with this work, because they compare a shared design against a non-shared version of the same design. Measured on the same basis, the sharing here delivers 11.2 % in LUTs and 10.3 % in flip-flops, larger than any single ablation SHARMONY reports. The scope of unification is the second field on which the two designs differ. SHARMONY unifies two hash primitives from the same family, whose state sizes coincide almost exactly (24×64 bits for the SHA-2 message schedule and working variables versus 25×64 bits for the Keccak state), and this coincidence lets a single 25×64-bit register bank serve both, with the two round datapaths sharing the same next-state multiplexers. Their large flip-flop saving derives almost entirely from this storage overlap. No such coincidence exists across AES, SHA-2 and ECDSA-P384: the 128-bit AES state, the 1024-bit SHA-2 padding buffer, the 384-bit ECDSA modular arithmetic register file and their attached multiplexers do not share

anything, so the sharing here is done by removing duplicated primitives (one AES core for CMAC and GCM, one SHA-384 core for HMAC and ECDSA) rather than by overlapping storage. The upper bound on whole-design LUT saving is therefore set by the size of what can be halved, which in this design amounts to approximately 6,500 LUTs out of 52,461, i.e. 12.4 % before any implementation loss. Table 4.4 shows that 12.4 % is reached against the arithmetic sum and 11.2 % against `mods_wrapper`, meaning that the sharing captures essentially all of the available reduction. In this sense the two designs achieve comparable efficiency on the shareable portion of their respective architectures.
The lower panel of Table 4.4 makes the same point in a form that removes the ECDSA modular arithmetic datapath from the denominator entirely. Excluding it, the sharing saves 29.3 % of the LUTs and 39.4 % of the flip-flops. The flip-flop figure in particular sits close to the 41 % by which SHARMONY reduces flip-flops against OpenTitan for state sharing alone, which supports the interpretation above: on the portion of the design that admits sharing, the two works are of comparable effectiveness.

4.3.3 Unified accelerator versus `mods_wrapper`

	LUT	**FF**	**Slices**
(a) Sharing by hierarchy level, vs. arithmetic sum			
AES agile: `cmac+gcm`	−2,096	−363	−671
area saving	13.8 %	10.5 %	19.0 %
SHA agile: `hmac+ecdsa`	−3,743	−2,613	−1,073
area saving	10.0 %	12.6 %	8.5 %
Top level: `aes_agile+sha_agile`	−675	+139	−647
area saving	1.4 %	−0.7 %	4.5 %
Total vs. arithmetic sum	**−6,514**	**−2,837**	**−2,391**
area saving	**12.4 %**	**11.7 %**	**14.8 %**
Total vs. `mods_wrapper`	**−5,812**	**−2,444**	**−1,677**
area saving	**11.2 %**	**10.3 %**	**10.9 %**
(b) Incremental cost, excluding the ECDSA datapath			
`mods_wrapper` − `ecdsa_core`	19,814	6,206	4,327
`crypto_agile` − `ecdsa_core`	14,002	3,762	2,650
Area saving	**29.3 %**	**39.4 %**	**38.8 %**

Table 4.4: Where the sharing occurs. Panel (a) decomposes the total saving by hierarchy level, referenced to the arithmetic sum of the standalone cores. Panel (b) reports the cost of adding the symmetric and hash services on top of an ECDSA-P384 engine, whose modular-arithmetic datapath shares nothing with the other primitives and dominates absolute area.

Table 4.4 reports the internal comparison of this work. `mods_wrapper` integrates the four standalone cores of Table 4.1 behind the same top-level interface as

`crypto_agile_core`. This is the sharing measurement in the strictest sense: any difference between the two rows is attributable to the sharing strategy alone, because the four algorithms and their cycle counts are the same.

The unified accelerator saves 11.2 % of the LUTs, 10.3 % of the flip-flops, 10.9 % of the occupied slices, and halves the block RAM count from four to two. It also reaches a slightly higher F_{max} than the wrapper (125.0 MHz against 122.1 MHz, an increase of 2.4 %), which is a small but genuine result: this is probably due to the input MUX for the wrapper, which extends the critical path of the ECDSA datapath.

Table 4.4 decomposes the total saving by hierarchy level. Most of the reduction occurs within the two intermediate agile cores: merging `cmac_core` and `gcm_core` into `aes_agile_core` saves 2,096 LUTs and 671 slices, while merging `hmac_core` and `ecdsa_core` into `sha_agile_core` saves 3,743 LUTs and 1,073 slices. In both cases the saving is the removal of a duplicated primitive: one AES core is shared by CMAC and GCM, and one SHA-384 core is shared by HMAC and ECDSA. Bringing the two subsystems together at the top level yields a further 675 LUTs, roughly 1.4 % of the sum, plus a small increase of 139 flip-flops that reflects the cost of the extended mode multiplexers required to route four families of operands through one datapath. The result is that cross-family sharing pays substantially where primitives share a datapath element (the AES core, the SHA-384 core) and pays only marginally otherwise, which is the same conclusion reached by SHARMONY [8]: their headline gain comes from two hash families whose state sizes coincide, and they explicitly keep the round datapaths separate.

The lower panel of Table 4.4 reports the incremental cost of the design, excluding the ECDSA modular arithmetic datapath. That datapath dominates absolute area and shares nothing with the other primitives, so it bounds the whole-design percentage by construction. Excluding it, adding AES-GCM, AES-CMAC and HMAC-SHA-384 on top of an ECDSA-P384 engine costs 29.3 % fewer LUTs, 39.4 % fewer flip-flops and 38.8 % fewer slices than instantiating the equivalent standalone cores. This is the measurement of the sharing strategy that is invariant to the choice of ECDSA microarchitecture, and it is arguably the strongest single area result of the design.

The throughput comparison in Table 4.2 closes the picture. Every service in the unified core runs at exactly the same cycles per block as its standalone counterpart. Sharing therefore introduces no architectural performance cost. The unified core delivers the same services at reduced area and matched functionality, at a single frequency lower than the standalone AES and HMAC cores could individually achieve but higher than the non-shared `mods_wrapper` manages under the same conditions.

As a final note, it is worth mentioning that the above comparisons were made without taking into account the peripherals like DMA and register interface described in Section 3.4. Therefore, the internal non-agile to agile comparison would be more profitable because, in the case of four different standalone cores, the peripherals should be repeated four times in hardware. On the other hand, the comparison is not straightforward to make because the DMA and register interface would be different for each standalone core, and each of them would require a dedicated RTL and resources analysis.

5 Conclusions

Metric	`mods_wrapper`	`crypto_agile_core`	Saving
Slice LUTs	51,759	45,947	11.2 %
Slice registers	23,752	21,308	10.3 %
Occupied slices	15,391	13,714	10.9 %
CARRY4	2,445	2,263	7.4 %
BRAM36	4	2	50.0 %
$F_{\max}$ (MHz)	122.14	125.03	+2.4 %

Table 5.1: Post place-and-route comparison of the unified accelerator against the equivalent non-shared union of the four standalone cores (`mods_wrapper`).

This work exploits the concept of resource agility to realize a CSSDS-compliant cryptographic accelerator for space environments. The results show that area savings are consistent for algorithms that share the same primitive (like CMAC and GCM, or HMAC and ECDSA), but less effective between totally different algorithms. The main area savings contributions are given by sharing AES and SHA primitive cores, with **13.8%/10.5%** and **10.0%/12.6%** LUTs/FFs savings respectively, compared to standalone cores. Additionally, integrating all modules into a single core reduced interface complexity and improved final routing, obtaining another small contribution of **1.4%/-0.7%** in LUTs/FFs reduction. Overall, compared to standalone modules instantiated in a single wrapper, the LUTs/FFs saved in the final agile core are **11.2%/10.3%** and a frequency gain of **2.4%**. If the ECDSA service is excluded considering the fact that it occupies most of the total area and shares only the SHA core, the area savings are **29.3%/39.4%** in LUTs/FFs. Further optimizations are unlikely to be feasible, since almost all the resource sharing possibilities are exploited, as explained in Section 3.3.

5.1 Future work

At the conclusion of this work, it is fair to acknowledge that some improvements of the realized architecture can be made. To start, the AES core could be slightly modified to allow also the computation of 128-bit keys for legacy implementations. The same argument is valid for the implementation of SHA-512 in the SHA-384 core, due to the fact that the two primitives share the same hardware architecture. The previously cited SHA core could be exposed on the API for computing only the hash,

without going through HMAC or ECDSA. Also, as noticed in Section 3.1.2, the `IV` nonce parameter of GCM should not be repeated under the same key, but this verification is left to the accelerator caller. The current revision of the ECDSA core leaves three FIPS 186-5 [16] validation checks to the caller: the signature-component non-zero condition at signing, the range check on incoming signature components at verification, and the public-key validation at verification. A future revision would move these checks into the core, so that the accelerator refuses to emit a non-compliant signature and rejects wrong verification inputs without involvement of the firmware layer. The signing-side check is trivial (a 384-bit comparison against zero on each component). The range check at verification is a pair of 384-bit comparisons against the loaded n. The public-key validation would require a curve-equation evaluator, reusing the existing PE array for the required field multiplications and additions, and would run once at key loading time rather than per verification. Moreover, a constant time algorithm for scalar multiplication could be added for better security. In the end, radiation hardening strategies could be implemented to make the accelerator appropriate for deployment in space environments. For this category of improvements, direct access to mathematical constants ROMs used for the algorithms could be added, to allow the caller the refreshing of the constants in case of random bit-flips in the memories.

Bibliography

[1] Wenyi Morty Zhang et al. "Don't Look Up: There Are Sensitive Internal Links in the Clear on GEO Satellites". In: *Proceedings of the 32nd ACM SIGSAC Conference on Computer and Communications Security (CCS '25)*. Taipei, Taiwan: ACM, 2025. DOI: 10.1145/3719027.3765198. URL: https://doi.org/10.1145/3719027.3765198.

[2] Consultative Committee for Space Data Systems. *CCSDS Cryptographic Algorithms*. Recommended Standard (Blue Book), Issue 2 CCSDS 352.0-B-2. Washington, DC, USA: The Consultative Committee for Space Data Systems, 2019. URL: https://ccsds.org/publications/bluebooks/entry/3128/.

[3] Ming-Bo Lin and Jen-Hua Chuang. "The Design of a High-Throughput Hardware Architecture for the AES-GCM Algorithm". In: *IEEE Transactions on Consumer Electronics* 70.1 (2024), pp. 425–432. DOI: 10.1109/TCE.2023.3332872.

[4] Marcio Juliato and Catherine Gebotys. *FPGA Implementation of an HMAC Processor Based on the SHA-2 Family of Hash Functions*. Technical Report. Waterloo, ON, Canada: University of Waterloo, Department of Electrical and Computer Engineering, 2011.

[5] Binh Kieu-Do-Nguyen et al. "Low-Cost Area-Efficient FPGA-Based Multi-Functional ECDSA/EdDSA". In: *Cryptography* 6.2 (2022), p. 25. DOI: 10.3390/cryptography6020025. URL: https://doi.org/10.3390/cryptography6020025.

[6] Pietro Nannipieri et al. "VLSI Design of Advanced-Features AES Cryptoprocessor in the Framework of the European Processor Initiative". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 30 (Dec. 2021), pp. 177–186. DOI: 10.1109/TVLSI.2021.3129107.

[7] Sreyas Janamanchi et al. "CIPHERX: A Unified High-Efficiency AES Encryption and CMAC Authentication Accelerator for Edge Applications". In: *Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. IEEE, 2025.

[8] Liga Anwar et al. *SHARMONY: Composing SHA-2 and SHA-3 Hardware for Crypto-Agile PQC*. Cryptology ePrint Archive, Paper 2026/1572. 2026. URL: https://eprint.iacr.org/2026/1572.

[9] Consultative Committee for Space Data Systems. *CCSDS Cryptographic Algorithms*. Informational Report (Green Book), Issue 2 CCSDS 350.9-G-2. Washington, DC, USA: The Consultative Committee for Space Data Systems, 2023. URL: https://ccsds.org/publications/greenbooks/entry/3141/.

[10] National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication FIPS 197. Gaithersburg, MD, USA: National Institute of Standards and Technology, 2023. DOI: 10.6028/NIST.FIPS.197-upd1. URL: https://doi.org/10.6028/NIST.FIPS.197-upd1.

[11] National Institute of Standards and Technology. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST Special Publication SP 800-38D. Gaithersburg, MD, USA: National Institute of Standards and Technology, 2007. DOI: 10.6028/NIST.SP.800-38D. URL: https://doi.org/10.6028/NIST.SP.800-38D.

[12] National Institute of Standards and Technology. *Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*. NIST Special Publication SP 800-38B. Gaithersburg, MD, USA: National Institute of Standards and Technology, 2016. DOI: 10.6028/NIST.SP.800-38B. URL: https://doi.org/10.6028/NIST.SP.800-38B.

[13] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*. Federal Information Processing Standards Publication FIPS PUB 180-4. Gaithersburg, MD, USA: National Institute of Standards and Technology, 2015. DOI: 10.6028/NIST.FIPS.180-4. URL: http://dx.doi.org/10.6028/NIST.FIPS.180-4.

[14] Wikipedia contributors. *Secure Hash Algorithm*. Wikipedia. 2025. URL: https://it.wikipedia.org/wiki/Secure_Hash_Algorithm.

[15] National Institute of Standards and Technology. *Keyed-Hash Message Authentication Code (HMAC)*. NIST Special Publication, Initial Public Draft SP 800-224 ipd. Gaithersburg, MD, USA: National Institute of Standards and Technology, 2024. DOI: 10.6028/NIST.SP.800-224.ipd. URL: https://doi.org/10.6028/NIST.SP.800-224.ipd.

[16] National Institute of Standards and Technology. *Digital Signature Standard (DSS)*. Federal Information Processing Standards Publication FIPS 186-5. Gaithersburg, MD, USA: National Institute of Standards and Technology, 2023. DOI: 10.6028/NIST.FIPS.186-5. URL: https://doi.org/10.6028/NIST.FIPS.186-5.

[17] Gerwin Gsenger. "Speeding Up Elliptic Curve Cryptography by Optimizing Scalar Multiplication in Software Implementations". Master's thesis. Graz, Austria: Graz University of Technology, Institute for Applied Information Processing and Communications (IAIK), 2014. URL: https://diglib.tugraz.at/speeding-up-elliptic-curve-cryptography-by-optimizing-scalar-multiplication-in-software-implementations-2014.

[18] Po-Chun Liu, Hsie-Chia Chang, and Chen-Yi Lee. "A 1.69 Gb/s Area-Efficient AES Crypto Core with Compact On-the-Fly Key Expansion Unit". In: *Proceedings of the 35th European Solid-State Circuits Conference (ESSCIRC)*. Athens, Greece: IEEE, 2009, pp. 404–407. DOI: 10.1109/ESSCIRC.2009.5326020.

[19] Arash Reyhani-Masoleh, Mostafa Taha, and Doaa Ashmawy. "Smashing the Implementation Records of AES S-box". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.2 (2018), pp. 298–336. DOI: 10.13154/tches.v2018.i2.298-336. URL: https://tches.iacr.org/index.php/TCHES/article/view/884.

[20] David A. McGrew and John Viega. *The Galois/Counter Mode of Operation (GCM)*. Submission to the NIST Modes of Operation Process. National Institute of Standards and Technology, 2005.

[21] Enrico Manfredi. "Tightly-coupled VS Loosely-coupled accelerators for data compression in space applications: a NOEL-V case study". Master's thesis. Torino, Italy: Politecnico di Torino, 2025. URL: https://webthesis.biblio.polito.it/35394/?template=default.

Acknowledgments

Thanks to my supervisors Enrico Manfredi, Alessandra Dolmeta, Guido Masera and Maurizio Martina for the big help in this work, from the project design choices to the writing of the thesis.