Department of Electrical and Information Technology
Faculty of Engineering, LTH, Lund University

Master's Thesis

Keccak Acceleration for Post-Quantum Cryptography: A DMA-Enhanced RISC-V Architecture

Teng Lin
te3387li-s@student.lu.se

Supervisor: Qian Guo
Examiner: Thomas Johansson

2026

Abstract

Post-quantum cryptography increases the computational and data-movement demands placed on embedded processors. Many standardized and candidate schemes rely heavily on Keccak, SHA-3, and SHAKE operations, where the 1600-bit Keccak state is repeatedly updated and may become a bottleneck when accelerated through software-hardware interaction. This thesis presents a direct memory access (DMA)-enhanced Keccak acceleration architecture for a RISC-V-based system, aiming to reduce Keccak offloading overhead while preserving a non-intrusive memory-mapped programming model.

The proposed design provides two complementary acceleration paths. The first is a raw Keccak-f[1600] DMA backend that offloads permutation-level state transfer and computation. The second is a one-shot SHAKE256 sponge DMA backend used for selected HQC operations, raising the acceleration boundary from a single permutation to a complete sponge transaction. Both accelerators are integrated into the X-HEEP system and evaluated on an FPGA platform using software-only, raw DMA, and hybrid configurations. Correctness is verified through key-encapsulation shared-secret checks and digital-signature verification across HQC, ML-KEM, ML-DSA, and SPHINCS+-SHAKE workloads.

Experimental results show that the raw DMA backend provides stable Keccak-layer acceleration of approximately 28× across the evaluated workloads. The corresponding end-to-end improvement varies with the Keccak fraction of each algorithm: HQC benefits only modestly, ML-KEM and ML-DSA obtain moderate speedups, and SPHINCS+-SHAKE benefits the most because it is dominated by repeated Keccak invocations. Hardware counters further indicate that the Keccak datapath itself is fast, while DMA transaction and software-interface overhead remain visible.

These results show that memory-mapped DMA provides reusable and low-intrusion Keccak acceleration for RISC-V-based post-quantum cryptographic workloads, with portability and fallback capability traded against higher invocation overhead than tightly coupled custom-instruction or register-based

co-processor interfaces.

Keywords: Post-quantum cryptography; Keccak; RISC-V; X-HEEP; Direct Memory Access; Hardware acceleration; FPGA.

Popular Science Summary

Today's online security relies on mathematical problems that ordinary computers struggle to solve. A large quantum computer could break several of these systems, which is why researchers are preparing post-quantum cryptography: new methods intended to remain secure in the quantum era.

Security alone is not enough. These algorithms must also run efficiently on real hardware, including the small processors used in connected devices, vehicles, sensors and industrial systems.

This project focused on Keccak, the technology behind SHA-3 and SHAKE. Keccak can be imagined as a digital blender: it takes data, mixes it in a carefully controlled way, and produces output that is extremely hard to reverse or predict. Many post-quantum algorithms use this blender again and again, making it an attractive target for hardware acceleration.

The challenge is that Keccak works on a large internal state of 1600 bits. Even if the hardware mixer is fast, time can still be lost moving data between the processor, memory and accelerator. This project therefore studied a DMA-enhanced Keccak accelerator for a RISC-V-based system. DMA, or direct memory access, gives the accelerator something like its own delivery service: it can fetch data from memory, process it, and write the result back without asking the processor to move every piece by hand.

The design used two acceleration paths. One speeds up the basic Keccak operation so it can be reused by several post-quantum algorithms. The other accelerates a complete SHAKE256 operation for selected HQC tasks, moving a larger piece of work into hardware. The accelerator was tested in an FPGA prototype based on the X-HEEP RISC-V platform with workloads including HQC, ML-KEM, ML-DSA and SPHINCS+-SHAKE.

The key result was that the Keccak part became about 28 times faster with the raw DMA accelerator. Full applications did not all speed up by the same amount: SPHINCS+-SHAKE benefited strongly because it uses Keccak heavily, while HQC improved only modestly because much of its time is spent elsewhere.

Acknowledgments

During my master's thesis, I have received support from several people. First, I would like to thank my supervisor, Qian Guo, for the guidance, feedback, and encouragement provided throughout this project. I am also grateful to my parents and friends for their support and for helping me stay motivated during the work.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Quantum computing promises new computational capabilities, but it also challenges the assumptions behind current public-key cryptography. RSA, Diffie–Hellman, and Elliptic Curve Cryptography rely on integer-factorization or discrete-logarithm problems that large-scale quantum computers could solve efficiently [1]. This risk has accelerated the move toward post-quantum cryptography (PQC). The goal is to deploy schemes that remain secure against classical and quantum adversaries.

Efficient implementation is a practical part of that migration. Standardized and selected schemes, including ML-KEM, ML-DSA, SLH-DSA/SPHINCS+-SHAKE, and HQC, rely heavily on hash and permutation primitives. Keccak therefore has a direct effect on the throughput of many PQC implementations [2, 4, 5, 6, 7]. Its regular round structure is well suited to hardware, yet each invocation manipulates a 1600-bit state. On embedded processors, repeatedly moving that state through software and memory interfaces can become a bottleneck even when the permutation logic itself is fast.

1.1 Project Background

Recent research shows that tightly coupled accelerators can reduce the cost of cryptographic kernels by placing custom functionality close to a RISC-V processor pipeline [8, 9]. This integration style lowers invocation overhead compared with loosely coupled coprocessors and matches primitives that are called repeatedly inside PQC software stacks. HQC and other code-based or hash-intensive schemes make this pressure visible: when Keccak remains entirely in software, repeated permutation calls can dominate execution time.

1.2 Problem Statement and Motivation

Acceleration does not remove every source of cost. Many interfaces still depend on the CPU to move the 1600-bit Keccak state through register transfers and memory operations. The processor then spends cycles on loads, stores, and synchronization rather than permutation work. Throughput falls, and communication becomes difficult to overlap with computation. In PQC workloads with frequent Keccak calls, state-transfer overhead can consume a large part of the gain provided by the accelerator datapath.

This thesis studies a memory-mapped, DMA-coupled Keccak accelerator as a way to reduce that transfer cost. The CPU configures a transaction, while the accelerator IP moves state data through its OBI master access to memory and invokes the Keccak datapath. The design keeps a conventional memory-mapped programming model and avoids custom instruction integration. It also adds controller logic, synchronization cost, resource overhead, and software orchestration, so the benefit must be measured at system level.

1.3 Thesis Objectives

This thesis develops a DMA-enhanced memory-mapped Keccak accelerator for a RISC-V platform and evaluates it in an X-HEEP-based FPGA prototype. The work focuses on throughput, hardware cost, and the programming model of a DMA-enabled external accelerator IP. It then positions the result relative to register-based and tightly coupled approaches. The evaluation links application-level timing, Keccak-layer timing, hardware DMA counters, correctness checks, resource utilization, and timing closure, rather than treating performance as a single number.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. The next chapter reviews the necessary background on post-quantum cryptography, Keccak, HQC, and RISC-V-based hardware accelleration. The following chapters present the proposed DMA-enhanced architecture, the design and integration methodology, and the implementation details of the accelerator and its system interface. Finally, the thesis reports the experimental results, analyzes the performance-resource trade-offs, and summarizes the main conclusions together with possible future directions.

Chapter 2
Background

2.1 Post-Quantum Cryptography

The security of widely deployed public-key cryptosystems is fundamentally challenged by quantum computing. In particular, Shor's algorithm shows that integer factorization and discrete logarithm problems can be solved efficiently on sufficiently powerful quantum computers, which directly threatens RSA, Diffie–Hellman, and Elliptic Curve Cryptography [1]. As a consequence, cryptographic systems that currently protect communication channels, authentication mechanisms, and digital signatures must eventually be replaced or augmented by algorithms that remain secure in the quantum era.

To address this transition, the National Institute of Standards and Technology (NIST) initiated a long-running standardization effort for post-quantum cryptography (PQC). This effort seeks not only to identify mathematically sound candidates, but also to promote schemes that are practical for software and hardware deployment at scale. The resulting PQC landscape includes lattice-based, hash-based, code-based, and other families of constructions, including standardized ML-KEM, ML-DSA, and SLH-DSA as well as HQC, which has been selected for standardization [4, 5, 6, 7]. From a system-design perspective, standardization has therefore created a new requirement: platforms must support cryptographic workloads whose computational and communication demands differ significantly from those of classical public-key algorithms.

This implementation challenge is especially visible on embedded and resource-constrained systems. PQC algorithms typically manipulate larger states, longer keys, and more intensive polynomial or hash-based computations than classical schemes. Even when an algorithm is secure and standardized, its real-world usability depends on whether it can be executed with acceptable throughput, latency, energy consumption, and silicon cost. For this reason, efficient hardware support has become an important research direction in the broader migration toward PQC.

2.2 Keccak, SHA-3, and SHAKE

Keccak is the permutation-based cryptographic primitive that underlies the SHA-3 family of hash functions and the SHAKE extendable-output functions standardized by NIST in FIPS 202 [3]. Unlike classical Merkle–Damgård hash constructions, Keccak is built around a sponge construction, in which data is first absorbed into an internal state and output is later squeezed from that same state. This structure is especially relevant from an architectural perspective because a single computational core can support fixed-length hashing, variable-length output generation, and several auxiliary cryptographic functions through the same underlying permutation.

In the sponge construction, the internal state has a fixed width b, which for the standardized Keccak-f[1600] instance is 1600 bits. The state is logically divided into two parts: the rate r, which interfaces with input and output data, and the capacity c, which determines the security margin. During absorption, message blocks are XORed into the rate portion of the state, after which the permutation is applied. During squeezing, output blocks are read from the rate portion, with additional permutation calls performed when more output is required. This absorb–permute–squeeze workflow is important for hardware design because it makes the permutation the dominant computational kernel while also imposing repeated state-movement requirements between memory, the processor, and the accelerator.

The core transformation used in SHA-3 and SHAKE is the Keccak-f[1600] permutation specified by FIPS 202 [3]. It operates on a 1600-bit state arranged as 5×5 lanes, where each lane is 64 bits wide. The permutation consists of 24 rounds, and each round applies the same sequence of five step mappings: θ, ρ, π, χ, and ι. At a high level, these steps combine linear mixing, structured rotations, data reordering, nonlinear substitution, and round-constant injection. This round structure is regular and deterministic, which makes it naturally amenable to hardware implementation. At the same time, the state is large enough that storing, updating, and transferring it efficiently becomes a nontrivial system concern.

From the viewpoint of processor acceleration, several characteristics of Keccak are particularly important. First, the algorithm relies heavily on bitwise Boolean operations, fixed rotations, and structured permutations, all of which map efficiently to specialized datapaths. Second, the round schedule is static, which simplifies control logic and favors pipelined or iterative hardware implementations. Third, the 1600-bit state is substantially larger than the native word width of an embedded processor, meaning that software implementations often require many instructions to marshal data in and out of the computational core. Consequently, even when the mathematical algorithm is conceptually simple, the practical implementation cost can be

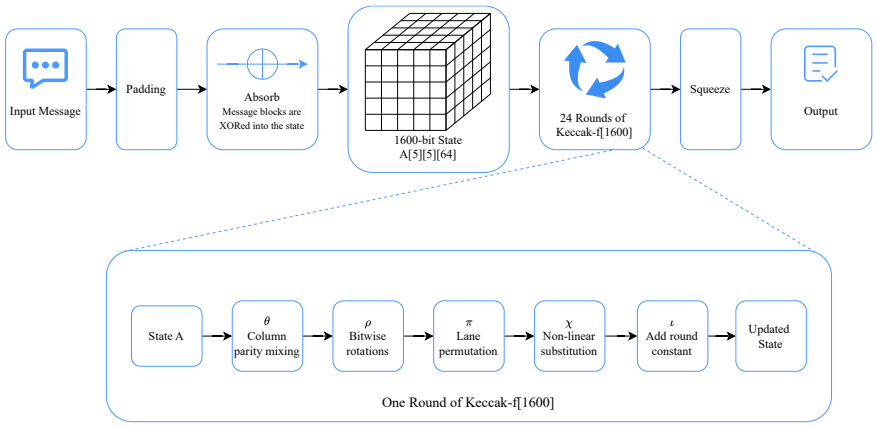

Figure 2.1: Schematic view of the Keccak processing flow. Input data is absorbed into the rate portion of a 1600-bit state, processed by repeated Keccak-f[1600] permutations, and then squeezed to produce SHA-3 or SHAKE outputs. From an architectural perspective, both the regular round structure and the repeated movement of the large internal state are important.

dominated by state handling rather than by arithmetic complexity alone.

This observation is especially relevant in post-quantum cryptography. PQC schemes frequently use SHA-3 or SHAKE for seed expansion, pseudorandom generation, rejection sampling support, domain separation, and challenge derivation. As a result, Keccak is often invoked repeatedly as an internal service primitive rather than as a one-time hash function. In such workloads, performance is determined not only by the latency of a single Keccak-f[1600] round sequence, but also by how efficiently the system can feed data into the permutation and retrieve results. Therefore, for architecture design, Keccak should be viewed as both a computation problem and a data-movement problem. This dual nature is precisely why it is a suitable target for hardware acceleration, and why communication support mechanisms such as DMA become central to end-to-end performance.

2.3 RISC-V Extensibility

RISC-V provides an open Instruction Set Architecture (ISA) with a modular design philosophy, making it particularly attractive for research and domain-specific processor customization. Unlike closed commercial ISAs, RISC-V allows architects to explore extensions at both the instruction and microarchitectural levels without proprietary restrictions. This flexibility

has encouraged its adoption in academic prototypes, industrial accelerators, and embedded platforms where workload-specific optimization is essential.

One of the most important advantages of RISC-V in this context is its support for custom instructions and tightly integrated accelerator interfaces. Frequently used operations can be exposed directly to software through ISA extensions, while more complex functions can be attached as coprocessor-like units that interact closely with the processor pipeline. This design space spans lightweight instruction-level acceleration and more substantial tightly coupled hardware blocks, allowing system designers to balance programmability, performance, and area according to application needs.

For cryptographic workloads, this extensibility is particularly valuable. Cryptographic kernels often consist of repeated arithmetic, permutation, and bit-manipulation operations with well-defined dataflow patterns, making them suitable candidates for specialization. In the case of PQC, where implementation pressure is high and algorithmic kernels are frequently reused, RISC-V offers a practical platform for exploring how custom hardware support can reduce overhead while preserving software control and flexibility.

In this thesis, the practical system implementation is built on the X-HEEP platform, an open-source and extendible RISC-V platform for embedded applications [10]. More specifically, X-HEEP is used as the base system into which a custom Keccak acceleration IP is integrated. On top of this hardware platform, the corresponding PQC software is developed so that the target algorithms can invoke and evaluate the accelerator in a realistic execution flow. The complete design is then deployed on FPGA for functional validation and performance-oriented testing, making X-HEEP the concrete hardware-software framework used to verify the proposed accelerator in the actual project. At the same time, it should be noted that X-HEEP is primarily an ASIC-oriented MCU platform, and the use of FPGA as the validation target introduces additional implementation constraints. As a result, the prototype used in this work is not expected to achieve very high operating frequencies during validation, and the measured frequency should therefore be interpreted as a limitation of the prototyping environment rather than of the architectural concept itself.

2.4 Keccak Hardware Acceleration

A software-only implementation of Keccak on a general-purpose embedded processor offers maximum flexibility, but it also exposes the full cost of repeated permutation execution to the CPU. Every round operation, state update, and memory movement must be orchestrated in software, which can

lead to high cycle counts and limited throughput. This baseline is important because it highlights the gap between algorithmic suitability for hardware and actual software performance on constrained platforms.

One way to narrow this gap is through instruction-level acceleration, where carefully selected operations are implemented as custom instructions. This approach preserves a close software-hardware interaction model and can reduce the cost of frequently repeated low-level transformations. A more aggressive alternative is the use of a dedicated accelerator or coprocessor that performs a larger portion of the Keccak computation in hardware. Such designs can provide substantial computational speedup by exploiting parallelism, specialized datapaths, and optimized internal state handling [8, 9].

When Keccak is executed purely in software, all round operations, state updates, and memory accesses are handled by the CPU, which offers flexibility but also leads to high instruction counts and limited throughput on embedded processors. Hardware acceleration addresses this limitation by moving part or all of the permutation into specialized logic, thereby reducing the computational burden on the core and improving execution efficiency. However, once the Keccak computation itself is accelerated, the bottleneck can shift from arithmetic processing to data transfer: the large 1600-bit state must still be moved between the CPU, memory, and the hardware accelerator. This observation motivates the next step in the design space, namely the use of DMA support to reduce CPU-driven communication overhead and improve end-to-end system performance.

2.5 DMA for Accelerator-Centric Systems

In many accelerator-centric systems, data movement is still driven explicitly by the CPU. The processor must issue load and store operations, marshal buffers, and synchronize the accelerator with the memory hierarchy. When the computational kernel is invoked frequently and the data footprint is large, this control burden can become a major bottleneck.

Direct Memory Access (DMA) addresses this issue by allowing data transfers between memory and hardware units to proceed with reduced CPU involvement. Instead of using the core to perform each transfer step explicitly, a DMA-capable subsystem can autonomously read from and write to memory according to a configured transfer pattern. This reduces instruction pressure on the CPU, frees execution resources for other tasks, and can create opportunities to overlap communication with computation.

For Keccak- and PQC-oriented workloads, DMA is especially attractive because the same primitive is invoked repeatedly on relatively large and

structured states. In the case of Keccak-f[1600], the accelerator must repeatedly consume and produce a 1600-bit state, which corresponds to 25 lanes of 64 bits each. If this transfer is handled purely by the CPU, the processor must issue a long sequence of load and store operations to move the state from memory into the accelerator interface and then write the updated state back after the permutation completes. This not only increases instruction count, but also occupies the pipeline with communication work that does not contribute directly to cryptographic computation.

DMA changes this interaction model by allowing the state transfer to be described once and then executed autonomously by dedicated hardware. Instead of manually moving all 25 words through the core, the CPU can configure the source address, destination address, transfer length, and synchronization conditions, after which the DMA engine fetches the Keccak state from memory and delivers it to the accelerator with minimal processor intervention. Once the permutation is complete, the updated state can be written back to memory through the same mechanism. In this way, the CPU is largely removed from the critical path of state transport.

From a system perspective, this solves several problems at once. First, it reduces CPU-driven communication overhead and frees processor cycles for control tasks or parallel software execution. Second, it better matches the structured nature of Keccak-f[1600], whose fixed-size state and repetitive invocation pattern are well suited to regular block transfers. Third, it creates the possibility of overlapping communication with computation, so that state movement and permutation execution are no longer serialized in the same way as in a software-managed flow. Offloading these transfers can therefore improve overall throughput even when the computational accelerator itself is already efficient. Consequently, DMA should be viewed not merely as a convenience feature, but as a key architectural mechanism for closing the gap between local computational speedup and end-to-end system performance. This observation motivates the DMA-enhanced memory-mapped accelerator architecture investigated in this thesis.

2.6 Related Work

Recent work on post-quantum cryptographic hardware has mainly evolved along three directions. A first line of research studies protocol-oriented HQC implementations on FPGA and related hardware platforms, with the goal of reducing the latency and area cost of key generation, encapsulation, and decapsulation. Representative examples include early full-HQC hardware realizations and later resource-optimized architectures [11, 12, 13]. Together, these studies show that HQC can be implemented efficiently in hardware, but

they also reveal that the main performance bottlenecks are concentrated in a small number of computational kernels, especially polynomial multiplication, encoding, and decoding-related stages. In this sense, the literature has progressively moved from full-function protocol implementation toward more specialized acceleration strategies.

A second line of work focuses on accelerating the arithmetic kernels that dominate the execution of code-based and hash-intensive post-quantum schemes. In this context, tightly coupled accelerators have emerged as an effective compromise between pure software flexibility and fully external hardware offload. Recent RISC-V-based accelerators demonstrate that Keccak-oriented support can significantly reduce execution time by moving repeated permutation operations into dedicated hardware while preserving a close coupling with the processor pipeline [8, 9]. At the same time, related work shows that the same trend extends beyond hashing into arithmetic-heavy kernels, where algorithm-specific accelerators are integrated at system level to target dense polynomial operations [14]. These studies establish that computation itself can be accelerated effectively, and they provide a strong foundation for using Keccak and related kernels as hardware acceleration targets in PQC systems.

A third line of work concerns system integration. Open and extensible RISC-V platforms make it possible to evaluate accelerator concepts in realistic embedded environments rather than in isolated kernel benchmarks alone. This system-level perspective is important because the performance of an accelerator depends not only on the speed of its internal datapath, but also on how it interacts with the processor, the memory system, and the surrounding software stack. In the present thesis, this role is fulfilled by the X-HEEP-based implementation platform discussed in Section 2.3 [10]. The broader direction is also consistent with recent work on RISC-V coprocessors and hybrid accelerator frameworks for post-quantum workloads, where the key design challenge is no longer the datapath alone, but the efficiency of the full hardware-software stack [15, 16].

Compared with the existing literature, the contribution of this thesis is centered less on proposing another standalone Keccak engine and more on addressing the communication bottleneck that remains after computation is accelerated. Prior work has clearly shown the value of tightly coupled acceleration, but the repeated movement of the 1600-bit Keccak state can still limit end-to-end performance when transfers are managed explicitly by the CPU. The main distinction of the present work is therefore the study of a DMA-enhanced memory-mapped accelerator architecture, implemented and validated in an X-HEEP-based environment, with the objective of reducing data-movement overhead rather than accelerating the permutation datapath alone.

Chapter 3
System Architecture

3.1 Overview of the Proposed Architecture

This work extends the X-HEEP RISC-V microcontroller with two memory-mapped Keccak acceleration backends. The architectural goal is to preserve the high-level software structure of the target post-quantum algorithms while moving the most frequently executed Keccak operations to dedicated hardware. The RISC-V core therefore remains responsible for algorithm control, memory allocation, random sampling, key generation, encapsulation, decapsulation, signing, verification, and UART-based profiling, whereas the accelerators are invoked only when execution reaches a Keccak-intensive kernel.

At system level, the architecture combines the RISC-V core, the on-chip memory subsystem, the system interconnect, a memory-mapped control and status space, a raw Keccak DMA accelerator, and a Keccak sponge DMA accelerator. Both accelerators are integrated as external peripherals. Their control registers are accessed by the CPU through normal MMIO transactions, while their data paths access memory through independent OBI master ports. As a result, control and bulk data movement are separated cleanly: the CPU configures the transaction, but the accelerator itself performs the memory transfers needed by the Keccak computation.

To avoid terminology ambiguity, the implemented architecture is referred to in the remainder of this thesis as a memory-mapped DMA-coupled accelerator architecture. The terms tightly coupled, register-based, and CV-X-IF-style accelerator are used only when discussing alternative integration styles or prior work. In this implementation, the Keccak IPs are not inserted into the processor pipeline and are not invoked through custom instructions; they are external X-HEEP peripheral IPs with MMIO control registers and OBI master ports for SRAM access.

A simplified block-level view of the system is shown in Figure 3.1.

The raw Keccak DMA backend accelerates the Keccak-f[1600] permutation at the software permutation-call boundary. Because this boundary is

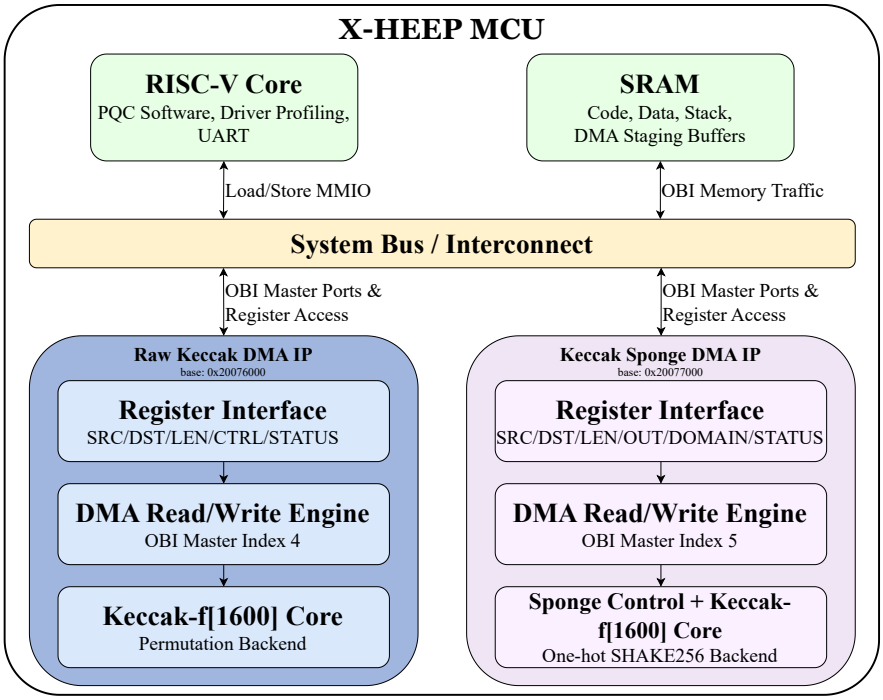

Figure 3.1: Block-level organization of the Keccak acceleration IPs. The raw Keccak DMA backend exposes a permutation-level interface, while the sponge DMA backend adds SHAKE256 absorb, padding, and squeeze control around the shared Keccak-f[1600] permutation core.

shared by SHAKE128, SHAKE256, SHA3, and several domain-separated sponge constructions, the raw backend provides a generic acceleration mechanism that can be reused across multiple PQC schemes. The sponge DMA backend operates at a higher abstraction level. It implements a complete one-shot SHAKE256 operation and is currently used for the HQC G/K `shake256_512_ds()` path. This creates a layered acceleration strategy: the raw backend emphasizes compatibility, while the sponge backend explores the extra performance available when a full XOF operation can be offloaded as one hardware transaction.

3.2 Baseline RISC-V System

The baseline system is the unaccelerated X-HEEP software implementation. In this mode, the complete post-quantum cryptographic workload executes on the RISC-V core. All high-level SHAKE and SHA3 logic remains unchanged,

Algorithm 1 Software-only baseline procedure

1: Seed the deterministic random-number generator
2: Run the target benchmark stage
3: Execute SHAKE/SHA3 fully in software
4: Run CPU Keccak permutations as required
5: Check outputs and report profiling data

including message absorption, padding, domain separation, squeezing, and incremental state management. The low-level permutation `KeccakF1600_StatePermute()` is also executed entirely in software.

The baseline workflow is summarized in Algorithm 1.

This software-only configuration has two purposes in the thesis. First, it provides the correctness reference for all accelerated implementations. Since the hardware-assisted builds preserve the same algorithm-level control flow and only replace selected Keccak operations, their outputs can be compared directly against the software baseline under the same deterministic random seed. Second, it provides the performance baseline used to quantify both the benefit and the overhead of DMA-based acceleration.

The baseline also defines the meaning of the software-visible Keccak counters used in the profiling output. When DMA is disabled, Keccak cycle counts correspond directly to CPU execution time spent inside the software Keccak path. When DMA is enabled, the same software-visible region includes driver overhead, memory staging, MMIO configuration, polling, and hardware execution. This distinction is essential when comparing software-visible Keccak cycles with accelerator-side cycle counters.

3.3 Keccak Accelerator Integration

The architecture integrates two Keccak hardware backends because Keccak appears at different abstraction levels inside the target algorithms. Some code paths benefit primarily from a generic permutation accelerator, while others contain fixed SHAKE invocations whose semantics are stable enough to justify a more specialized hardware implementation.

The raw Keccak DMA engine is the permutation-level backend. Its software interface replaces `KeccakF1600_StatePermute()`, and its input and output are 1600-bit Keccak states represented as 200-byte memory buffers. The surrounding SHAKE or SHA3 software remains responsible for rate handling, message absorption, domain separation, padding, and output squeezing. This makes the backend intentionally algorithm-independent: it accelerates the common permutation kernel without embedding high-level

sponge semantics in the RTL.

The sponge DMA engine is the sponge-level backend. Instead of exposing a single permutation call, it accepts an input buffer, input length, output buffer, output length, and domain byte, and then performs a full one-shot SHAKE256 operation in hardware. In the current implementation, this backend is used specifically for the HQC G/K `shake256_512_ds()` path. This path is a suitable target because its behavior is fixed: a domain-separated SHAKE256 invocation produces a 512-bit output. If the accelerator succeeds, software returns the output directly. If it fails, software records the fallback event and executes the reference software path instead.

The integration is therefore asymmetric by design. The raw backend is broad and reusable, while the sponge backend is narrower but potentially more powerful because it removes repeated software-hardware interactions around a complete XOF operation. This distinction is important for later evaluation, since it allows the results to separate permutation-level speedup from sponge-level offload benefits.

3.4 DMA-Enhanced Data Transfer Architecture

Both accelerators rely on DMA-style memory access in order to reduce CPU involvement in data movement. Instead of moving the Keccak state or message payload through a narrow peripheral register interface, the CPU writes only the configuration values needed to describe the transaction. Once started, the accelerator fetches input data from memory through its own OBI master port, performs the Keccak computation, and writes the output back to memory.

For the raw Keccak DMA backend, the data flow has four stages: aligned input buffering, DMA read, Keccak-f[1600] execution, and DMA write-back. Software materializes the state in memory, while the accelerator fetches, permutes, and returns the state for later software processing. The CPU-side procedure is compact: stage the 1600-bit state, write `SRC_ADDR`, `DST_ADDR`, and `DATA_LEN`, start the transfer, poll for completion, and restore the output state. Profiling registers such as `LAST_OP_CYCLES` and `LAST_CORE_CYCLES` are read after successful completion. This workflow reduces direct CPU participation in state transport, but it still leaves software responsible for the surrounding sponge logic.

For the sponge DMA backend, the data flow covers a complete one-shot SHAKE256 operation. The accelerator reads the input message, performs absorption, domain separation, padding, internal Keccak-f[1600] permutations, and squeezing, and then writes the XOF output back to memory. The CPU-side procedure remains register-oriented: prepare aligned buffers, write

Table 3.1: Implemented address map of the Keccak accelerators.

Module	Address expression	Runtime address	Function
Raw Keccak DMA	`EXT_PERIPHERAL + 0x06000`	`0x20076000`	Keccak-f[1600] permutation DMA
Keccak Sponge DMA	`EXT_PERIPHERAL + 0x07000`	`0x20077000`	One-shot SHAKE256 sponge DMA

`SRC_ADDR`, `DST_ADDR`, `DATA_LEN`, `OUT_LEN`, and `DOMAIN`, start the accelerator, poll `STATUS`, and consume the output buffer. Compared with raw DMA, this workflow removes more software interaction around absorb, padding, and squeeze when the operation semantics match the hardware implementation.

The DMA-enhanced architecture therefore reduces CPU involvement in data transport, but it does not eliminate all software overhead. The CPU still performs buffer staging, register programming, status polling, fallback handling, and algorithm-level control. As a result, hardware core cycles are expected to be lower than software-visible Keccak cycles, because the latter measure the full cost observed by software around each accelerated transaction.

3.5 Memory Map and Control Registers

Both accelerators are placed in the X-HEEP external peripheral region and are accessed through fixed memory-mapped base addresses. The raw Keccak DMA is mapped at `EXT_PERIPHERAL + 0x06000`, corresponding to runtime address `0x20076000`. The sponge DMA is mapped at `EXT_PERIPHERAL + 0x07000`, corresponding to runtime address `0x20077000`. This separation allows the two backends to be independently configured while sharing a common system-level programming model. Table 3.1 summarizes the implemented address map.

The raw DMA interface includes registers for source address, destination address, transfer length, control, and status. In addition, it exposes profiling-oriented registers such as the cycle count of the previous DMA transaction, the active Keccak core cycle count, and the accumulated number of completed operations. The corresponding raw Keccak DMA register map is shown in Table 3.2.

The sponge DMA follows the same general structure, but extends it with output length and domain separation registers, as well as a register that reports the number of internal Keccak permutations performed during the

Table 3.2: Raw Keccak DMA register map.

Offset	Register	Access	Description
0x00	SRC_ADDR	RW	Source address of the 200-byte input state
0x04	DST_ADDR	RW	Destination address of the output state
0x08	DATA_LEN	RW	Transfer length; the driver uses the Keccak state size
0x0c	CTRL	RW	Start control bit
0x10	STATUS	RO	Done, busy, and error status
0x14	LAST_OP_CYCLES	RO	Cycle count of the previous DMA transaction
0x18	LAST_CORE_CYCLES	RO	Cycles during which the Keccak core was active
0x1c	OP_COUNT	RO	Number of completed accelerator operations

previous transaction. Its register map is summarized in Table 3.3.

The software drivers require input and output buffers to be 4-byte aligned. After a successful call, the profiling layer reads the accelerator-side cycle counters and accumulates them at application level. This makes the register map important not only for control, but also for measurement: UART logs can report software-visible timing together with hardware-side timing, operation counts, and fallback counts.

3.6 Execution Flow

The application selects its Keccak backend at compile time. The key design principle is that the high-level cryptographic benchmark structure is preserved across all configurations, so that software, raw DMA, and hybrid execution can be compared directly. Each build therefore follows the same algorithm-level sequence for key generation, encapsulation, decapsulation, signing, or verification; only the implementation of selected Keccak operations changes.

Software Mode. In software mode, both accelerators are disabled and all Keccak computation is performed on the CPU. The benchmark procedure is summarized in Algorithm 2.

The corresponding software flow remains entirely on the CPU: the application invokes the PQC algorithm, the algorithm invokes the SHAKE/SHA3

Table 3.3: Keccak sponge DMA register map.

Offset	Register	Access	Description
0x00	SRC_ADDR	RW	Source address of the input message
0x04	DST_ADDR	RW	Destination address of the output buffer
0x08	DATA_LEN	RW	Input message length in bytes
0x0c	OUT_LEN	RW	Requested output length in bytes
0x10	DOMAIN	RW	Domain separation byte
0x14	CTRL	RW	Start control bit
0x18	STATUS	RO	Done, busy, and error status
0x1c	LAST_OP_CYCLES	RO	Cycle count of the previous sponge transaction
0x20	LAST_CORE_CYCLES	RO	Accumulated Keccak core active cycles
0x24	OP_COUNT	RO	Number of completed sponge operations
0x28	LAST_CORE_PERMS	RO	Number of internal Keccak permutations

Algorithm 2 Software-mode benchmark procedure

1: **for** each benchmark iteration **do**
2: Seed the deterministic random-number generator
3: Run the selected PQC stage
4: Use the software SHAKE/SHA3 and Keccak path
5: Check outputs and report profiling data
6: **end for**

Algorithm 3 Raw DMA invocation procedure

```
1: procedure RAW DMA PERMUTATION(state)
2:     Stage the aligned state buffer
3:     Program source, destination, and length registers
4:     Start DMA and poll STATUS
5:     if the DMA transaction succeeds then
6:         Restore the state and collect counters
7:     else
8:         Count fallback and run the CPU permutation
9:     end if
10: end procedure
```

routines, and those routines invoke the software Keccak permutation whenever needed.

Raw DMA Mode. In raw DMA mode, the high-level SHAKE and SHA3 routines remain unchanged, but calls to `KeccakF1600_StatePermute()` are redirected to the raw Keccak DMA driver. If the driver reports failure, the software permutation is used as fallback. The invocation procedure is summarized in Algorithm 3.

The invocation flow is therefore mixed: the application and SHAKE/SHA3 control logic remain in software, while the permutation call is redirected to the raw DMA backend.

Hybrid Mode. Hybrid mode is used for HQC. It enables both hardware backends. The HQC G/K `shake256_512_ds()` function is offloaded to the sponge DMA engine, while all other Keccak permutations continue to follow the raw DMA flow. The sponge-offload procedure is summarized in Algorithm 4.

All remaining Keccak permutation calls still use the raw backend when enabled; otherwise they fall back to the software permutation. The hybrid workflow is therefore split across two hardware paths: the HQC G/K flow is redirected to the sponge DMA engine, whereas the remaining permutation calls continue to use the raw DMA backend.

This execution model allows the evaluation to distinguish three effects clearly: the cost of the complete cryptographic algorithm, the benefit of permutation-level acceleration, and the additional benefit of replacing a complete SHAKE256 operation with a sponge-level hardware transaction. The UART output therefore reports both correctness and multiple layers of performance data, including full application cycles, software-visible Keccak

Algorithm 4 Hybrid HQC offload procedure

```
1: procedure HYBRID HQC SHAKE256(input, output, domain)
2:     if the domain is G or K and sponge DMA is enabled then
3:         Program sponge DMA and start SHAKE256
4:         Poll STATUS
5:         if the DMA transaction succeeds then
6:             Return the hardware output
7:         else
8:             Count fallback and run software SHAKE256
9:         end if
10:    else
11:        Run the normal software SHAKE path
12:    end if
13: end procedure
```

cycles, DMA transaction cycles, Keccak core active cycles, operation counts, and fallback counts.

Chapter 4

Hardware Implementation

4.1 Keccak-f[1600] Datapath

The hardware accelerator is centered on a Keccak-f[1600] permutation datapath. The datapath follows the standard Keccak state organization: a 1600-bit state is arranged as a 5×5 matrix of 64-bit lanes. This representation is important for hardware implementation because each round transformation operates naturally on columns, rows, lanes, or fixed lane positions. Instead of treating the state as a flat 1600-bit vector throughout the design, the internal datapath exposes the state as a two-dimensional lane array. This makes the hardware structure closely match the mathematical description of Keccak-f.

At block level, the implementation contains two Keccak-based IP variants. They share the same permutation-core design style but differ in the amount of sponge-level control implemented in hardware, as shown in Figure 4.1.

This organization clarifies the division of responsibility used throughout the chapter. The Keccak core implements the cryptographic permutation. The DMA controller moves data between SRAM and the local state buffer. The register interface provides software control and observability. The sponge-specific controller adds SHAKE256 semantics on top of the same permutation primitive.

The `keccak_data` module implements the sequential permutation datapath. It contains the internal 1600-bit state register, the round counter, the round-constant selection logic, and one instance of the combinational round function. The design uses an iterative architecture: one Keccak round is computed per active core cycle, and the result is written back into the state register before the next round begins. This choice avoids a fully unrolled 24-round datapath and therefore reduces area, register pressure, and routing complexity.

The datapath interface is intentionally compact, as summarized in Table 4.1.

When `start` is asserted, the permutation engine begins a new Keccak-

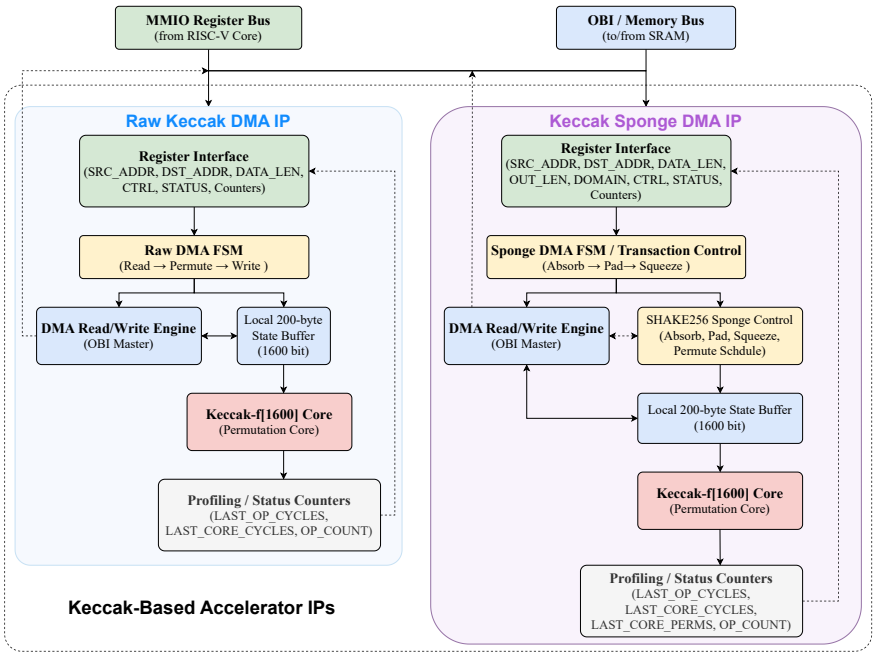

Figure 4.1: Block-level organization of the Keccak acceleration IPs. The raw Keccak DMA backend exposes a permutation-level interface, while the sponge DMA backend adds SHAKE256 absorb, padding, and squeeze control around the shared Keccak-f[1600] permutation core.

f[1600] operation. Each invocation is treated as a complete permutation of the 1600-bit value presented on `Din`. Internally, the local round state is cleared at the start of the operation and `Din` is used as the initial state to be permuted; equivalently, the first round operates on the supplied state rather than on a state retained from a previous transaction. The round counter then advances through the 24 Keccak rounds. While the permutation is active, `ready` is deasserted; when the final round completes, `ready` returns high and the output state is available on `Dout`.

This single permutation datapath is used by both accelerator variants, but the controllers prepare `Din` with different semantics. In raw permutation mode, the DMA controller reads the complete 200-byte software Keccak state from memory and supplies it as the full permutation input, so the hardware computes $S \leftarrow \mathrm{KeccakF1600}(S)$ for the state passed by software. In sponge mode, the sponge controller maintains a local 1600-bit sponge state buffer, XORs message bytes into the rate portion during absorption, applies the SHAKE domain and padding bytes, and then supplies the prepared full

Table 4.1: Datapath interface of the iterative Keccak-f[1600] permutation core.

Signal	Role
`start`	Begins a new permutation operation
`Din[1599:0]`	Provides the complete 1600-bit state for the current permutation invocation
`ready`	Indicates that the core is idle or that the permutation has completed
`Dout[1599:0]`	Exposes the current output state

state to the same permutation core. Thus the XOR-based absorb semantics belong to the sponge controller, not to the raw permutation interface.

The resulting design separates the cryptographic primitive from the data movement and control logic. The permutation core is responsible only for Keccak-f[1600]. The raw DMA controller and sponge DMA controller determine when to load data, when to start the core, when to store output, and how to expose status to software.

4.2 Round Function Implementation

The `keccak_round` module implements one complete Keccak-f[1600] round as combinational logic. It applies the five standard transformations in sequence: Theta, Rho, Pi, Chi, and Iota. Each transformation is represented explicitly in hardware, and the intermediate round states are connected combinationally. The round output is then registered by the surrounding permutation datapath.

This structure has two consequences. First, the logic for one round is evaluated in a single cycle, so the critical path is determined by the combined delay of the five transformations. Second, the state register is updated only once per round, which keeps the sequential structure simple and makes the total permutation latency proportional to the number of rounds.

The round constant used by Iota is generated by the `keccak_round_constants` module. It maps the current round index to the corresponding 64-bit Keccak-f[1600] round constant. Since the constants are fixed, a combinational lookup table is sufficient.

Throughout this section, the state bit notation $A_{y,x,i}$ is used to make the implemented row and column structure explicit. The index y denotes the row, x denotes the column or lane position, and i denotes the bit position inside a 64-bit lane. Relative to the conventional Keccak notation $A[x, y, z]$, the notation used here is $A_{y,x,i} \equiv A[x, y, i]$. Unless otherwise stated, all x

and y indices are evaluated modulo 5, and all bit indices i are evaluated modulo 64. Thus expressions such as $x - 1$, $x + 2$, and $i - 1$ wrap around their respective domains. This convention is independent of the sponge-level parameters b, r, and c introduced in Chapter 2.

4.2.1 Theta Step

Theta is the first diffusion step in each round. It computes the parity of each of the five columns and then mixes neighboring column parities into every lane. Architecturally, this step is implemented as an XOR network over the five lanes in each column, followed by additional XORs into all state lanes.

For a column index `x` and bit index `i`, the column parity can be described as:

$$C_{x,i} = A_{0,x,i} \oplus A_{1,x,i} \oplus A_{2,x,i} \oplus A_{3,x,i} \oplus A_{4,x,i}. \tag{4.1}$$

Each state bit is then updated using the parity of the previous column and a rotated parity from the next column:

$$A'_{y,x,i} = A_{y,x,i} \oplus C_{x-1,i} \oplus C_{x+1,i-1}. \tag{4.2}$$

In hardware, Theta is mainly an XOR-dominated network. It has a meaningful impact on timing because each output bit depends on multiple lane bits through column parity logic. However, it is fully parallel across rows, columns, and lane bits.

4.2.2 Rho and Pi Steps

Rho and Pi are implemented as fixed combinational wiring transformations. Rho rotates each 64-bit lane by a fixed offset defined by the Keccak specification in FIPS 202 [3]. Since the offsets are constants, the hardware does not require variable shifters or runtime barrel shifters. Each rotated output bit is connected to a statically determined input bit.

Pi then permutes the lane positions in the 5×5 state. It moves lanes between coordinates according to the Keccak mapping. Like Rho, this transformation is implemented as a fixed wiring permutation. It does not require arithmetic units during operation; the coordinate mapping is resolved structurally by synthesis.

From a resource perspective, Rho and Pi consume relatively little logic compared with Theta and Chi. Their main implementation cost is routing, because they move bits across the state matrix.

4.2.3 Chi Step

Chi is the nonlinear step of the Keccak round. It operates independently on each row of the state. Each output bit depends on the current lane bit, the next lane bit, and the lane after that:

$$A'_{y,x,i} = A_{y,x,i} \oplus (\neg A_{y,x+1,i} \wedge A_{y,x+2,i}) \,. \tag{4.3}$$

This step maps directly to inversion, AND, and XOR logic. Because Chi is evaluated for every bit of every lane, it provides a large amount of bit-level parallelism. It is also one of the major contributors to the combinational logic depth of a round.

4.2.4 Iota Step

Iota injects the round constant into lane `(0,0)`. All other lanes pass through unchanged. This step breaks the symmetry of the round function and ensures that each round is distinct. Written at bit level, the update for lane `(0,0)` is:

$$A'_{0,0,i} = A_{0,0,i} \oplus \mathrm{RC}_{\mathrm{round},i}. \tag{4.4}$$

The hardware implementation is small compared with the other steps. It consists of a 64-bit XOR into one lane and a round-constant selection network.

4.2.5 State Packing and Endianness

The DMA engines move Keccak state data through a 32-bit memory interface, while the Keccak permutation is naturally described as 25 64-bit lanes. The implementation bridges these two representations by loading the 200-byte state as 50 consecutive 32-bit words. Two adjacent 32-bit words correspond to one 64-bit Keccak lane, and the resulting 25 lanes form the 5×5 internal state matrix.

The packing order is kept consistent with the software Keccak representation used by the reference implementation. This is essential because the raw DMA backend is invoked at the `KeccakF1600_StatePermute()` boundary: software stages its lane array into memory, hardware loads that 200-byte value as the complete permutation input state, and software then consumes the returned state without changing the algorithm-level sponge logic. No previous accelerator state is combined with the raw input state. In other words, the hardware does not define a new external state format; it implements the same state interpretation expected by the software driver.

Table 4.2: Raw Keccak DMA controller state sequence.

State	Action	Next condition
IDLE	Wait for start and validate length	Valid: Read; invalid: Error
READ	Read input state through DMA	All read responses received
CORESTART	Pulse Keccak core start	Next cycle
COREWAITBUSY	Wait for core ready to deassert	`core_ready`=0
COREWAITDONE	Wait for core ready to assert again	`core_ready`=1
WRITE	Write output state through DMA	All write responses received
DONE	Latch counters and done status	Return to Idle
ERROR	Latch error status	Return to Idle

The system uses word-level DMA transfers and byte-level masks for partial final words, but Keccak permutation calls normally operate on the full 200-byte state. For sponge DMA, the same 50-word state buffer is used internally, while only the first 34 words correspond to the SHAKE256 rate portion. Input bytes are XORed into this rate portion, and output bytes are read back from it during squeeze.

Correct state packing is verified functionally rather than assumed from the RTL structure alone. The raw DMA output is compared against the software `KeccakF1600_StatePermute()` result under deterministic test inputs, and the sponge DMA SHAKE256 output is compared against the software SHAKE256 path for the HQC G/K use case. These tests ensure that the memory-word ordering, lane packing, and software-visible state semantics are aligned.

4.3 Control Finite State Machine

The accelerator design uses finite state machines to coordinate memory access, permutation execution, completion reporting, timeout detection, and profiling counters. There are two control layers: the raw permutation controller and the sponge controller.

The raw permutation controller is implemented by the `keccak` module. It accepts a source address, destination address, data length, and start command. Its role is to read one Keccak state from memory, run the permutation core, and write the resulting state back to memory.

The raw controller follows the detailed state sequence summarized in Table 4.2.

Table 4.3: Main FSM state groups used by the raw Keccak DMA controller.

State group	Architectural purpose
Idle/configuration	Wait for software start and validate the requested length
Read transfer	Fetch input words from memory through the OBI master port
Core start/wait	Start the permutation and wait for the core to complete
Write transfer	Store output words back to memory
Done/error	Update status, interrupt, and performance counters

The main state groups are summarized in Table 4.3.

The controller tracks granted requests, returned responses, and outstanding transactions. This avoids assuming that memory responses occur in the same cycle as requests. It also includes timeout counters for both memory transactions and core execution. If the expected grant, response, or core transition does not occur within the timeout window, the controller raises the error flag.

The raw DMA transaction can be summarized as the timing-oriented sequence in Table 4.4.

This sequence makes the implementation robust against non-zero memory latency. The controller does not require OBI grant and read response to occur in the same cycle; instead, it treats request acceptance and data return as separate events.

The sponge controller is more complex because it implements a complete SHAKE256 operation rather than one permutation. It still uses the same basic control principles: read from memory, update local state, start the Keccak core, wait for completion, and write results back to memory. However, it repeats this sequence across absorb and squeeze phases.

The key difference between the two FSMs is the level at which they terminate. The raw FSM terminates after one read-compute-write sequence because its unit of work is one Keccak state permutation. The sponge FSM terminates only after all input bytes have been absorbed, the final padded block has been permuted, and all requested output bytes have been written. Therefore, the sponge FSM must maintain additional loop variables: input byte offset, output byte offset, bytes remaining, output bytes remaining, current chunk length, final-block state, and internal permutation count.

The high-level difference can be summarized as:

In this high-level flow, the final padded block is always permuted before the squeeze phase begins. If the message length is not a multiple of the

Table 4.4: Timing-oriented sequence of a raw DMA permutation transaction.

Phase	Action	Implication
Start	CPU writes `CTRL.START`	Internal start pulse latches source, destination, and length
Read request	DMA issues OBI read requests	Requests may be granted independently of response timing
Read response	Returned words are stored in the local state buffer	Response count and outstanding count are tracked separately
Core active	Keccak core performs the iterative permutation	Controller waits for busy and then completion
Write request	DMA writes the updated state back to SRAM	Byte enables handle a partial final word if needed
Completion	Done/status and counters are latched	Software can read status and profiling registers

Algorithm 5 Raw-DMA transaction flow.

1: Read the Keccak state
2: Apply one Keccak-f[1600] permutation
3: Write the updated Keccak state
4: Complete the transaction

Algorithm 6 Sponge-DMA transaction flow.

```
for each full-rate message block do
    Read the 136-byte message block
    XOR the block into the rate portion of the local state
    Apply Keccak-f[1600]
end for
Read and XOR the final partial message block, if present
Apply the final domain and padding bytes
Apply Keccak-f[1600] to the final padded block
for each squeeze chunk do
    Write the output block
    if more output is required then
        Apply Keccak-f[1600]
    end if
end for
Complete the transaction
```

136-byte SHAKE256 rate, this final block contains the remaining message bytes followed by the domain and padding bytes. If the message length is an exact multiple of the rate, the final block is an additional empty rate block containing only the domain and padding bytes.

Both controllers generate hardware-visible profiling data. `LAST_OP_CYCLES` measures the transaction duration from accepted start until completion or error. `LAST_CORE_CYCLES` measures the cycles during which the Keccak core is active. The sponge controller additionally reports `LAST_CORE_PERMS`, the number of internal Keccak permutations used by the most recent sponge transaction.

4.4 Keccak Sponge DMA Architecture

The sponge DMA backend is a higher-level accelerator than the raw Keccak DMA backend. While the raw backend accelerates only one Keccak-f[1600] permutation, the sponge backend implements a complete one-shot SHAKE256 transaction. It is therefore responsible not only for invoking the permutation core, but also for managing SHAKE256 rate blocks, message absorption, domain separation, padding, output squeezing, and repeated internal permutations.

Table 4.5: Comparison of the raw Keccak DMA and sponge DMA hardware boundaries.

Backend	Hardware boundary	Software still handles
Raw Keccak DMA	One Keccak-f[1600] permutation	SHAKE/SHA3 absorb, padding, squeeze, state management
Sponge DMA	One complete SHAKE256 operation	Buffer preparation, register configuration, status polling

4.4.1 Architectural Role

The sponge DMA IP is designed for operations whose full hash/XOF semantics are known at the hardware boundary. In the current system, it is used for HQC G/K `shake256_512_ds()`. This function has a fixed semantic form: it consumes a domain-separated input and produces a SHAKE256 output. Because the operation can be submitted as one transaction, the CPU does not need to repeatedly enter the incremental SHAKE software path for this case.

The sponge DMA backend therefore occupies a different abstraction level from the raw DMA backend, as summarized in Table 4.5.

This distinction is important. The sponge DMA engine can reduce software overhead when the whole SHAKE256 operation matches the supported one-shot interface. However, it is not a universal replacement for all incremental SHAKE or SHA3 calls unless the hardware interface is extended to support those modes.

4.4.2 Top-Level Structure

The sponge DMA IP consists of four architectural blocks, as shown in Figure 4.2.

The register interface receives the source address, destination address, input length, output length, domain byte, and start command. The DMA read/write engine moves input and output data between SRAM and the local sponge state buffer. The sponge control logic implements SHAKE256 absorb, padding, and squeeze scheduling. The Keccak-f[1600] core performs the permutation whenever a full absorb block is processed or additional squeeze output is required.

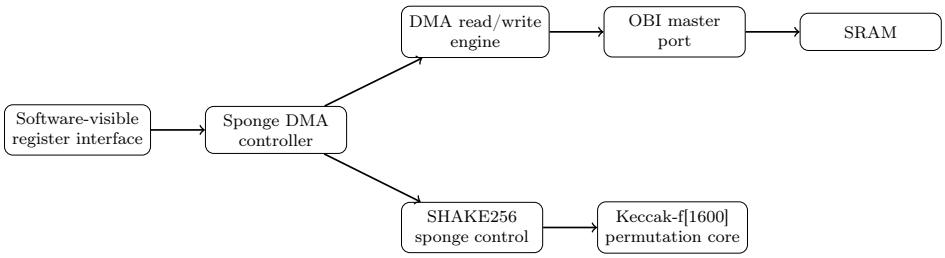

Figure 4.2: Top-level dataflow of the sponge DMA backend.

Table 4.6: SHAKE256 rate and state organization used by the sponge DMA backend.

Quantity	Value	Meaning
Keccak state size	1600 bits / 200 bytes	Full permutation state
SHAKE256 capacity	512 bits / 64 bytes	Security capacity
SHAKE256 rate	1088 bits / 136 bytes	Bytes absorbed or squeezed per block
Internal word width	32 bits	DMA transfer granularity
Rate words	34 words	136-byte rate portion
State words	50 words	200-byte full state

The top-level sponge module fixes the internal mode to SHAKE256 sponge mode. Although the internal controller contains a raw-permutation mode constant, the integrated sponge IP is used as a one-shot SHAKE256 backend in the current design.

4.4.3 SHAKE256 Rate and State Organization

SHAKE256 uses a 1600-bit Keccak state with a 512-bit capacity. The resulting rate is 1088 bits, or 136 bytes. The sponge DMA controller therefore processes input and output in chunks of up to 136 bytes.

The internal state buffer is organized as 50 32-bit words, corresponding to the 200-byte Keccak state. The rate portion occupies the first 34 words, because 136 bytes / 4 bytes per word = 34 words. During absorb, input words are XORed into this rate portion of the state. During squeeze, output words are written from the rate portion to the destination buffer.

The rate-based organization is summarized in Table 4.6.

This word-level organization matches the OBI memory interface, which transfers 32-bit data words. It also allows the controller to use byte enables for partial final output words.

4.4.4 Variable-Length Input and Output Handling

The current hardware supports variable-length messages and variable-length SHAKE256 output by decomposing each transaction into bounded chunks. The software provides the total input length through `DATA_LEN` and the requested output length through `OUT_LEN`. The hardware then derives the number of 32-bit memory words required for each chunk, the number of valid bytes in the final word, and the byte-enable mask used for the final write.

For the raw permutation backend, the transfer length is bounded by the Keccak state size. The controller clamps each raw chunk to at most 200 bytes, corresponding to the full 1600-bit state. In the normal software flow, the raw driver submits a 200-byte state for each `KeccakF1600_StatePermute()` call. Nevertheless, the controller still contains generic length-handling logic: it converts byte counts to word counts, masks unused bytes in a partial final read word, and generates byte enables for a partial final write word.

For the sponge backend, the variable-length mechanism is rate-based rather than state-size-based. The controller processes the input stream in chunks of at most 136 bytes, because SHAKE256 absorbs one rate block per permutation. The total message can therefore have any byte length:

The output side follows the same chunking principle. `OUT_LEN` is split into one or more chunks of at most 136 bytes. If the output fits in one rate block, the controller writes only that block and terminates. If more output is required, the controller writes one rate block, invokes Keccak-f[1600] again, and continues writing the next block. The final output chunk may be shorter than 136 bytes; in that case, the controller uses byte enables so that only the requested bytes are written to memory.

The conversion from byte length to word count is performed by rounding up to the next 32-bit word:

$$\text{word\_count} = \left\lceil \frac{\text{byte\_count}}{4} \right\rceil . \tag{4.5}$$

The number of valid bytes in the final word determines the final read mask or write byte enable, as shown in Table 4.7.

This mechanism allows the accelerator to support arbitrary byte lengths without requiring software to pad the input message manually or clean up extra output bytes after the DMA transaction.

4.4.5 Absorb Phase

The absorb phase reads the input message from SRAM and XORs it into the local Keccak state. The controller first determines the next absorb chunk

Algorithm 7 Sponge-DMA input handling for variable-length messages.

if DATA_LEN $= 0$ **then**
    Skip input reads
    Apply the domain byte at rate offset zero
    Apply the final `0x80` padding byte
    Run one absorb permutation
else if $0 <$ DATA_LEN < 136 **then**
    Read one partial input block
    Mask invalid bytes in the last input word
    Apply the domain byte after the message
    Apply the final `0x80` padding byte
    Run one absorb permutation
else if DATA_LEN $= 136n$ **then**
    Read and permute n full rate blocks
    Create an additional empty padded final block
    Run one final absorb permutation
else
    Treat DATA_LEN $= 136n + r$, where $0 < r < 136$
    Read and permute n full rate blocks
    Read one partial final block
    Apply domain separation and padding
    Run the final absorb permutation
end if

Table 4.7: Read masks and write byte enables for partial final words.

Tail bytes	Read mask	Write byte enable
0	`0xffffffff`	`1111`
1	`0x000000ff`	`0001`
2	`0x0000ffff`	`0011`
3	`0x00ffffff`	`0111`

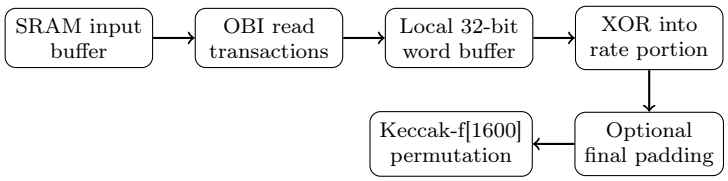

Figure 4.3: Dataflow of the absorb phase.

size. If at least 136 input bytes remain, it reads one full rate block. If fewer than 136 bytes remain, it reads the final partial block and then proceeds to padding.

The absorb dataflow is shown in Figure 4.3.

Input data is accumulated rather than overwritten. This is consistent with sponge absorption, where each message block is XORed into the current state before applying the permutation. For a full rate block, the controller starts the Keccak core immediately after the read completes. For a final partial block, the controller first applies the SHAKE domain byte and final padding before starting the core.

4.4.6 Domain Separation and Padding

The sponge DMA IP exposes an 8-bit `DOMAIN` register. During final-block processing, this domain byte is XORed into the byte immediately following the input message. The controller also XORs `0x80` into the final byte of the SHAKE256 rate portion. This implements the domain separation and multi-rate padding structure used by SHAKE-style functions in FIPS 202 [3].

Conceptually, let L_{msg} denote the total input message length in bytes, and let r_{off} denote the byte offset immediately after the last message byte inside the final rate block. For SHAKE256, the rate is 136 bytes and $r_{\text{off}} = L_{\text{msg}} \bmod 136$. The notation $State_{\text{rate}}[j]$ is byte-indexed within the current 136-byte rate portion, not within the global input stream. If L_{msg} is an exact multiple of the rate, padding is applied to an additional empty final block and $r_{\text{off}} = 0$ refers to that new empty block rather than to the preceding full-rate message block. The final rate block is transformed as follows:

$$\begin{aligned} State_{\text{rate}}[r_{\text{off}}] &\leftarrow State_{\text{rate}}[r_{\text{off}}] \oplus \text{DOMAIN}, \\ State_{\text{rate}}[\text{rate} - 1] &\leftarrow State_{\text{rate}}[\text{rate} - 1] \oplus \texttt{0x80}. \end{aligned} \tag{4.6}$$

The implementation performs this operation at word granularity while selecting the correct byte lane inside the word. This allows the input length to be any byte length, as long as the software provides aligned base addresses

Algorithm 8 Repeated absorb sequence for multi-block messages.

1: **while** input bytes remain **do**
2: Read one rate block
3: XOR the block into the local Keccak state
4: Start the Keccak-f[1600] permutation
5: Wait for completion
6: Advance the input pointer
7: **end while**

for the DMA buffers. The final `0x80` is applied to the last byte of the 136-byte rate block.

This padding design is the reason the sponge DMA backend can replace the HQC G/K SHAKE256 path directly: the software only needs to provide the correct domain byte and lengths, while the hardware completes the SHAKE256 block formatting internally.

4.4.7 Core Invocation During Absorb

After each absorbed block is ready, the controller pulses the Keccak core start signal. It then waits for the core to leave the ready state and later return to ready. This two-step observation prevents the controller from mistaking the idle-ready state before start for completion of the new permutation.

Each core start increments the internal permutation counter. The core-cycle counter remains active until the controller has observed both the busy transition and the return to ready. This counter structure is used to expose meaningful hardware-side cycle measurements through `LAST_CORE_CYCLES` and `LAST_CORE_PERMS`.

For a message longer than one rate block, the absorb phase repeats:

The final absorb permutation is followed by the squeeze phase.

4.4.8 Squeeze Phase

After the final absorb permutation completes, the sponge DMA controller writes output bytes from the rate portion of the state to the destination buffer. If the requested output length is less than or equal to 136 bytes, only one write phase is required. If more output is requested, the controller writes one rate block, starts another Keccak-f[1600] permutation, and then writes the next block.

The squeeze dataflow is shown in Figure 4.4.

For multi-block output, the repeated sequence is:

The controller supports partial final output words through byte enables.

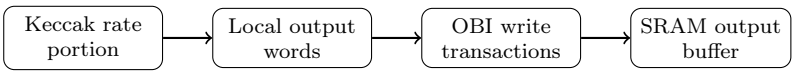

Figure 4.4: Dataflow of the squeeze phase.

Algorithm 9 Squeeze sequence for multi-block output.

```
1: repeat
2:     Write up to 136 output bytes
3:     if output bytes remain then
4:         Start the Keccak-f[1600] permutation
5:         Wait for completion
6:     end if
7: until no output bytes remain
```

This allows arbitrary output lengths without requiring the CPU to post-process the final word.

4.4.9 Sponge FSM Transition Logic

The sponge controller can be understood as a deterministic sequence of absorb, finalization, and squeeze phases. Its state transitions are driven by three quantities: the number of input bytes remaining, the number of output bytes remaining, and whether the current absorb block is the final block.

At the beginning of a transaction, the controller latches the source address, destination address, input length, output length, and domain byte. It also clears the byte offsets, local word buffer, operation cycle counter, core cycle counter, and internal permutation counter. The first decision is whether any input bytes remain:

During the input read phase, the controller issues OBI read requests until all words of the current input chunk have been requested and returned. Returned words are XORed into the local state buffer. If the chunk is a full 136-byte rate block, the controller starts the Keccak core immediately. If

Algorithm 10 Initial sponge-FSM branch based on input length.

```
1: if DATA_LEN ≠ 0 then
2:     Prepare an absorb chunk
3:     Enter the input-read state
4: else
5:     Prepare an empty final block
6:     Enter the padding state
7: end if
```

Algorithm 11 Absorb-side transition behavior of the sponge FSM.

```
Prepare the next absorb chunk
Read the chunk from SRAM
if chunk_bytes = 136 then
    Start the absorb permutation
else
    Apply final padding
    Start the absorb permutation
end if
Wait for the absorb permutation to complete
if final_block = true then
    Enter squeeze preparation
else
    Advance the input offset
    Prepare the next absorb chunk
end if
```

the chunk is shorter than the rate, the controller transitions to the padding state before starting the core.

The absorb-side transition behavior is:

A subtle case occurs when the input length is exactly a multiple of the SHAKE256 rate. In this case, all message bytes are processed as full blocks, but SHAKE padding still requires an additional final block. The controller handles this by advancing through all full input chunks and then entering the empty-pad path when no input bytes remain. This ensures that the domain byte and final `0x80` are applied to a fresh rate block rather than being incorrectly inserted into the previous full message block.

The squeeze-side FSM is controlled by `OUT_LEN`. Before each output write, the controller prepares a chunk of at most 136 bytes from the current state. It then issues OBI write transactions until all words in that chunk have been accepted and acknowledged. After the write phase, the controller checks whether more output bytes remain:

This structure means that the first squeeze block is produced directly from the state after the final absorb permutation. Additional squeeze blocks require one extra Keccak-f[1600] permutation per rate block. The internal `LAST_CORE_PERMS` counter therefore reflects both absorb permutations and squeeze permutations.

The FSM also separates the core wait logic into two phases. After asserting the core start pulse, the controller first waits until `ready` becomes low. Only after observing this busy phase does it accept a later high value of `ready` as completion. This avoids a false completion caused by the core

Algorithm 12 Squeeze-side transition behavior of the sponge FSM.

```
Prepare the next squeeze chunk
Write the chunk to SRAM
if output bytes remain then
    Start the squeeze permutation
    Wait for completion
    Prepare the next squeeze chunk
else
    Complete the transaction
end if
```

Table 4.8: Software-visible sponge DMA counters and their meanings.

Counter	Meaning in sponge mode
`LAST_OP_CYCLES`	Total hardware transaction cycles, including reads, padding, permutations, and writes
`LAST_CORE_CYCLES`	Sum of cycles during which the Keccak core was active
`LAST_CORE_PERMS`	Number of Keccak-f[1600] permutations used by the transaction
`OP_COUNT`	Number of completed or errored sponge transactions

being idle and ready before it has accepted the new operation.

4.4.10 Sponge DMA Completion and Observability

A sponge transaction completes when all requested output bytes have been written to memory. At this point, the controller sets the done status, raises the interrupt output, updates the operation counter, and latches the profiling counters.

The most important observable quantities are summarized in Table 4.8.

These counters allow the software benchmark to distinguish the cost of the full hardware transaction from the cost of the Keccak core itself. This is especially useful for HQC hybrid evaluation, where one sponge DMA call may internally contain many Keccak permutations while appearing as one high-level SHAKE256 call to software.

4.5 DMA Controller Interface

Both accelerator IPs use a minimal OBI-like master interface for memory access. The interface includes request, grant, write enable, address, write data, byte enable, read valid, and read data signals. The wrapper modules

Table 4.9: External OBI master-port assignment for the Keccak DMA engines.

IP	OBI master index	Function
Raw Keccak DMA	4	Reads and writes 200-byte Keccak states
Keccak Sponge DMA	5	Reads message buffers and writes SHAKE output buffers

adapt these internal signals to the X-HEEP OBI request and response records.

The raw and sponge DMA engines follow the same basic memory protocol. For reads, the controller issues an OBI request with write enable deasserted. When the request is granted, the internal grant counter is incremented. When read data returns, the response counter is incremented and the data word is stored in the local state buffer. The controller also tracks outstanding transactions to detect invalid or unexpected responses.

For writes, the controller issues OBI requests with write enable asserted. The write address is generated from the destination base address, the current byte offset, and the word index. The byte-enable signal is normally `1111`, but it is reduced for a partial final word. This is used by both raw DMA and sponge DMA when the transfer length is not a multiple of four bytes.

At the X-HEEP system level, the two DMA engines use the independent external master ports listed in Table 4.9.

This design avoids CPU-mediated word transfers. The CPU configures registers and polls status, while the DMA engines perform memory traffic directly.

4.6 Register Behavior and Start/Status Semantics

The detailed memory map and register list are described in Chapter 3. In this chapter, the important implementation point is the behavior of the control, status, and profiling registers. The raw and sponge IPs share the X-HEEP bus infrastructure but use independent register interfaces. Each register interface converts software-visible MMIO writes into internal control signals and exposes hardware state back to software.

The most important control behavior is the handling of `CTRL.START`. The software-visible start bit is converted into a one-cycle internal start pulse. This edge-detected pulse latches the current configuration registers and begins a new transaction. The pulse-based implementation prevents the accelerator from retriggering continuously if the software-visible control bit

Table 4.10: Hardware behavior of the main control, status, and profiling fields.

Register field	Hardware behavior
STATUS.BUSY	Asserted while a DMA or sponge transaction is active
STATUS.DONE	Latched after successful completion of the transaction
STATUS.ERROR	Latched when invalid configuration, OBI timeout, unexpected response, or core timeout is detected
LAST_OP_CYCLES	Latched at transaction completion or error
LAST_CORE_CYCLES	Latched with the measured Keccak core active cycles
LAST_CORE_PERMS	Latched by sponge DMA to report internal permutation count
OP_COUNT	Incremented when a transaction reaches completion or error handling

remains high for more than one cycle.

The status behavior is summarized in Table 4.10.

When an error is detected, the controller stops progressing the normal read, compute, or write sequence, records the error status, raises the interrupt output, deasserts the active operation state, and returns to an idle or terminal state according to the FSM transition. The software driver can then observe STATUS.ERROR and fall back to the software Keccak implementation. This policy is deliberately conservative: a failed hardware transaction is not silently treated as successful, and the software-visible fallback counters can report that the operation did not complete purely in hardware.

Performance counters are latched only at transaction boundaries. This avoids software observing partially updated values while an operation is still active. For raw DMA, the counters describe one permutation transaction. For sponge DMA, the counters describe the complete one-shot SHAKE256 transaction, including all internal absorb and squeeze permutations.

4.7 Timing and Resource Considerations

The primary timing path is inside the combinational Keccak round. Theta contributes a wide XOR network, Chi contributes nonlinear AND/XOR logic, and Rho/Pi contribute long-distance routing across the state matrix. Because one full round is evaluated per active core cycle, the maximum clock frequency depends strongly on the round-function critical path.

The iterative architecture is a deliberate area-performance trade-off. A fully unrolled 24-round design could reduce permutation latency, but it would significantly increase area and routing complexity. The current

implementation instantiates one round function and reuses it over multiple cycles. This is more suitable for integration into a compact X-HEEP-based FPGA system where the accelerator must coexist with the RISC-V core, SRAM banks, bus fabric, UART, debug logic, and other peripherals.

The theoretical datapath latency of one Keccak-f[1600] permutation is 24 round cycles, because the core evaluates one round per active cycle. The measured hardware core counter may include the controller's start and ready-observation boundary cycles, depending on the exact counter boundary used in the implementation. Therefore, the evaluation should interpret `LAST_CORE_CYCLES` as the measured hardware-active interval rather than as a pure mathematical round count. The corresponding permutation count is still exposed separately by the sponge DMA through `LAST_CORE_PERMS`.

The raw DMA backend has lower control complexity because it only performs one read-compute-write sequence per permutation request. The sponge DMA backend has higher control complexity because it manages SHAKE256 absorb and squeeze loops. However, the sponge backend can reduce software overhead when a complete SHAKE256 operation is offloaded as one transaction.

The hardware counters provide a practical way to separate these effects:

$$\begin{aligned} C_{\mathrm{SW}} &= C_{\mathrm{driver}} + C_{\mathrm{MMIO}} + C_{\mathrm{polling}} + C_{\mathrm{DMA}} + C_{\mathrm{bookkeeping}}, \\ C_{\mathrm{op}} &= C_{\mathrm{DMA\ reads}} + C_{\mathrm{padding/control}} + C_{\mathrm{core}} + C_{\mathrm{DMA\ writes}}, \\ C_{\mathrm{core}} &= \text{cycles spent inside Keccak-f[1600] permutations.} \end{aligned} \tag{4.7}$$

In normal operation, the following relationship is expected:

$$C_{\mathrm{SW}} \geq C_{\mathrm{op}} \geq C_{\mathrm{core}}, \qquad C_{\mathrm{core}} \approx (24 + \delta_{\mathrm{core}}) N_{\mathrm{perm}}, \qquad C_{\mathrm{core}} \geq 24 N_{\mathrm{perm}}. \tag{4.8}$$

The final lower bound follows from the iterative Keccak-f[1600] datapath used in this implementation: each permutation consists of 24 rounds, and the core evaluates one round per active cycle. The term δ_{core} represents implementation-dependent counter-boundary overhead, such as the cycles needed to observe the core leaving the ready state and returning to ready after the round sequence. In practice, the measured C_{core} may therefore be slightly larger than $24N_{\mathrm{perm}}$, but it should remain close to this round-count baseline.

For raw DMA, one software permutation call normally corresponds to one hardware permutation. For sponge DMA, one software SHAKE256 call may correspond to multiple internal permutations, depending on the input length and output length. This is why the sponge DMA exposes both core cycles and permutation count.

Table 4.11: Implementation resource utilization of the raw DMA and sponge DMA IPs.

Resource	Raw DMA IP	Sponge DMA IP	Notes
LUTs	7230	11025	Includes controller, datapath, and register logic
FFs	3781	4005	Includes state buffer, FSM registers, and counters
BRAM	0	0	Expected only if synthesis maps buffers to memory blocks
DSP	0	0	Keccak logic is not expected to require DSP blocks

From a resource perspective, the most expensive shared component is the Keccak-f[1600] round datapath. The DMA engines add address generation, counters, local word buffers, and OBI control logic. The sponge backend further adds rate-block scheduling, padding logic, squeeze control, and internal permutation counting. These additions are moderate compared with duplicating or unrolling the complete Keccak round datapath, and they provide a useful architectural comparison between permutation-level and sponge-level acceleration.

The post-synthesis resource results are summarized in Table 4.11. These results provide the implementation baseline used in the evaluation chapter and show the additional hardware cost of moving from permutation-level raw DMA support to a complete sponge-level DMA interface.

The sponge DMA IP therefore increases LUT utilization compared with the raw DMA IP, mainly because it contains additional absorb, padding, squeeze, and internal permutation-control logic. The FF increase is comparatively small, and neither design requires BRAM or DSP resources.

4.8 Functional Verification

The implementation is verified at multiple levels to ensure that hardware acceleration preserves the software-visible cryptographic semantics. The first level checks the Keccak-f[1600] permutation itself. Deterministic input states are processed by the hardware permutation core and compared with the software reference implementation. This validates the round function,

round constants, state update sequence, and state packing.

The second level verifies the raw DMA backend. The software prepares a Keccak state, invokes the raw DMA accelerator through the register interface, and compares the returned 200-byte state against the result of `KeccakF1600_StatePermute()` executed in software. This verifies the DMA read path, local state buffer, permutation invocation, DMA write path, and driver-visible completion behavior.

The third level verifies the sponge DMA backend. The hardware one-shot SHAKE256 result is compared against the software SHAKE256 implementation for the HQC G/K domain-separated test cases. This validates absorb chunking, domain separation, padding, internal permutation scheduling, squeeze output, and variable-length output handling.

Finally, full application-level tests compare the final cryptographic results. KEM applications check shared-secret equality after key generation, encapsulation, and decapsulation. Signature applications check that generated signatures verify successfully. These tests are run with deterministic seeds so that software, raw DMA, and hybrid configurations can be compared under identical inputs. UART logs additionally report hardware calls, fallbacks, status values, and performance counters, making it possible to detect both functional failures and unintended fallback execution.

Chapter 5

Software Interface and Programming Model

This chapter describes the software interface used to access the Keccak-based accelerator IPs from the X-HEEP RISC-V software stack. The programming model is designed to expose hardware acceleration without changing the reference post-quantum cryptographic APIs. Applications still call the original KEM, signature, SHAKE, SHA3, and Keccak routines. Hardware acceleration is inserted below these APIs through driver calls and Keccak-specific hooks.

A central design goal is that acceleration remains optional. If hardware execution succeeds, the software records the hardware call and uses the accelerator output. If hardware execution fails, the software falls back to the reference software path and preserves cryptographic correctness. Therefore, the software interface must report not only whether a test passes, but also which backend actually produced the result.

5.1 Software Stack Overview

The software stack is organized into four layers, as shown in Figure 5.1. The benchmark does not directly manipulate accelerator FSMs or DMA signals. Instead, it invokes the normal cryptographic API, while the Keccak integration layer selects the software, raw DMA, or sponge DMA backend.

The benchmark layer is responsible for experiment control: it prints the selected backend, probes enabled hardware blocks, runs the cryptographic workload, checks correctness, and reports timing through UART. The algorithm integration layer is responsible for preserving the original cryptographic API while redirecting selected Keccak operations to hardware. The driver layer provides memory-mapped access to the accelerators. The hardware interface layer consists of the raw Keccak DMA IP and the Keccak sponge DMA IP described in the previous chapters.

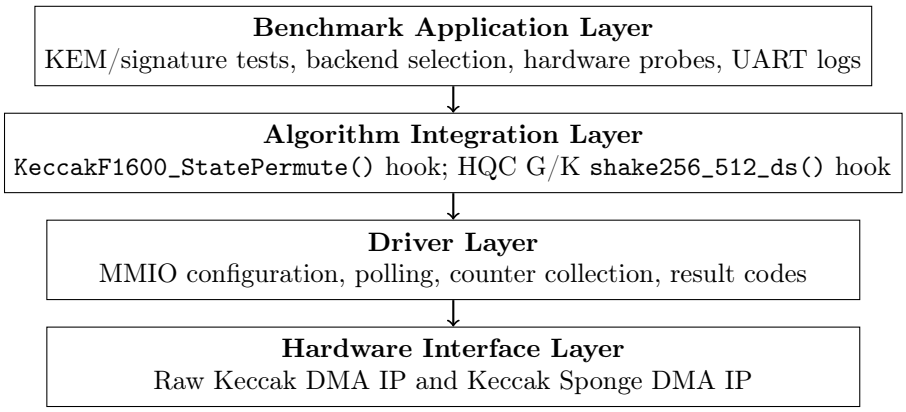

Figure 5.1: Software stack for Keccak DMA acceleration.

Table 5.1: Driver handles used by the software interface.

Driver object	Hardware target	Software role
`keccak_dma_t`	Raw Keccak DMA IP	Permutation-level transaction handle
`keccak_sponge_dma_t`	Keccak sponge DMA IP	One-shot SHAKE256 transaction handle

5.2 Driver Abstraction and Result Codes

The driver layer abstracts register-level accelerator control. It does not implement SHAKE, SHA3, KEM, or signature logic. Its responsibilities are limited to validating a transaction, writing MMIO registers, starting the accelerator, polling completion, reading status and counters, and returning a result code.

Two driver handles are used, as summarized in Table 5.1.

Each handle stores an `mmio_region_t` base address. Initialization converts the fixed accelerator base address into a driver object:

```
keccak_dma_init(handle, KECCAK_DMA_START_ADDRESS);
keccak_sponge_dma_init(handle,
                       KECCAK_SPONGE_DMA_START_ADDRESS);
```

The raw driver separates transaction start from completion polling. `keccak_dma_start()` the source address, destination address, transfer length, and start bit. `keccak_dma_wait()` polls the status register. The convenience function `keccak_dma_hash_block()` combines these two steps for the normal 200-byte Keccak-state transaction.

The sponge driver exposes `keccak_sponge_dma_shake256()` as a one-

Table 5.2: Driver result-code classes.

Result class	Meaning
Success	Hardware transaction completed and `STATUS.DONE` was observed
Timeout	Software polling limit expired before completion
Hardware error	Accelerator asserted `STATUS.ERROR`
Bad length	Driver rejected an invalid length before starting hardware
Bad address	Driver rejected an unaligned source or destination address

shot function because the current sponge IP performs a complete SHAKE256 transaction. The caller provides source address, input length, destination address, output length, domain byte, and timeout.

The driver result codes distinguish the main failure modes listed in Table 5.2.

This result-code interface allows the algorithm integration layer to apply a deterministic fallback policy without embedding low-level register behavior into the cryptographic code.

5.3 DMA Transaction Configuration

A DMA transaction is configured by placing input and output buffers in SRAM and passing their addresses to the accelerator through MMIO registers. The CPU does not transfer each data word through the register interface. Instead, after configuration, the accelerator reads and writes memory through its OBI master port.

Both accelerators require 4-byte aligned source and destination addresses. This matches the 32-bit memory access granularity of the OBI interface and simplifies byte-enable handling for partial final words.

5.3.1 Raw Keccak State Packing

The raw Keccak DMA backend operates on one 200-byte Keccak-f[1600] state. Software represents the state as 25 64-bit lanes, while the DMA engine transfers 32-bit words. The integration layer therefore packs each 64-bit lane into two 32-bit words before invoking the accelerator and reconstructs the lanes after completion.

The raw transaction configuration is summarized in Table 5.4.

The zero-to-one write sequence on `CTRL` is intentional. The hardware generates an internal one-cycle start pulse from the rising edge of `CTRL.START`.

Table 5.3: Raw Keccak state packing for DMA transfers.

Lane index	DMA word index	Meaning
i	$2i$	Lower 32 bits of `state[i]`
i	$2i + 1$	Upper 32 bits of `state[i]`

Table 5.4: Raw Keccak DMA transaction configuration.

Register	Value	Purpose
`SRC_ADDR`	aligned input buffer	200-byte input Keccak state
`DST_ADDR`	aligned output buffer	200-byte output Keccak state
`DATA_LEN`	`200`	Full Keccak-f[1600] state size in bytes
`CTRL`	`0`	Clear the software-visible start bit
`CTRL`	`1`	Generate a new rising-edge start event

Writing zero before one ensures that repeated software calls always create a new start event.

The packing order is kept consistent with the software Keccak reference implementation. This is essential because the raw DMA backend is inserted at the `KeccakF1600_StatePermute()` boundary; it must behave as a hardware implementation of the same state transformation, not as a new data format.

5.3.2 Sponge SHAKE256 Configuration

The sponge DMA backend is configured as a complete one-shot SHAKE256 transaction. Software provides the input message address, input length, output address, requested output length, and domain separation byte, as summarized in Table 5.5.

In HQC hybrid mode, `DOMAIN` corresponds to the G or K domain, and `OUT_LEN` is 64 bytes for SHAKE256-512 output. The sponge driver rejects zero-length output. Zero-length input is compatible with the hardware model if the address is aligned, although the HQC G/K path normally provides a non-empty input.

Table 5.5: Sponge DMA SHAKE256 transaction configuration.

Register	Value	Purpose
`SRC_ADDR`	aligned input buffer	Input message source address
`DST_ADDR`	aligned output buffer	SHAKE256 output destination address
`DATA_LEN`	input length in bytes	Number of input bytes to absorb
`OUT_LEN`	requested output length	Number of output bytes to squeeze
`DOMAIN`	domain byte	Domain separation value applied during padding
`CTRL`	`0`	Clear the software-visible start bit
`CTRL`	`1`	Generate a new rising-edge start event

5.4 Accelerator Invocation Paths

The accelerators are invoked through Keccak integration hooks, not directly from the benchmark. This keeps the cryptographic call graph close to the software reference implementation and localizes hardware-specific behavior.

5.4.1 Raw Permutation Hook

The raw DMA backend is invoked from `KeccakF1600_StatePermute()`. Algorithm 13 summarizes the software behavior.

This hook is generic because SHAKE128, SHAKE256, SHA3, and several post-quantum schemes all eventually call Keccak-f[1600]. For this reason, the same raw DMA backend is used by HQC, ML-KEM, ML-DSA, and SPHINCS+-SHAKE builds.

5.4.2 HQC G/K Sponge Hook

The sponge DMA backend is invoked from HQC `shake256_512_ds()` only when the domain byte corresponds to the G or K function. Algorithm 14 summarizes the hybrid path.

Algorithm 13 Raw DMA invocation at the Keccak permutation boundary

```
Read the software cycle counter
if raw DMA is enabled and not suppressed then
    Initialize the DMA handle if needed
    Pack 25 64-bit lanes into 50 32-bit DMA words
    Start a 200-byte raw DMA transaction
    Poll STATUS until done, error, or timeout
    if the DMA transaction succeeds then
        Unpack the output words back into the Keccak state
        Accumulate hardware operation and core-cycle counters
        Increment the raw DMA call counter
        return
    else
        Increment the raw DMA fallback counter
        Disable later raw DMA attempts
    end if
end if
Execute the software Keccak-f[1600] permutation
```

Algorithm 14 Sponge DMA invocation for HQC G/K SHAKE256

```
if sponge DMA is enabled and the domain is G or K then
    Validate input alignment and output handling
    Start a one-shot SHAKE256 sponge DMA transaction
    Poll STATUS until done, error, or timeout
    if the sponge DMA transaction succeeds then
        Use the 64-byte hardware output
        Accumulate sponge operation, core-cycle, and permutation counters
        return
    else
        Increment the sponge fallback counter
        Temporarily suppress raw DMA
        Execute the software SHAKE256 fallback
        Restore the raw DMA suppression state
        return
    end if
end if
Execute the software SHAKE256 path
```

The temporary raw-DMA suppression during sponge fallback is deliberate. It ensures that a failed sponge operation is recovered by pure software SHAKE256 rather than by a software SHAKE path internally accelerated by raw permutation DMA. Consequently, the sponge call and fallback counters

Table 5.6: Counter interpretation across the two DMA backends.

Counter	Raw DMA interpretation	Sponge DMA interpretation
`LAST_OP_CYCLES`	Full read-permute-write transaction	Full one-shot SHAKE256 transaction
`LAST_CORE_CYCLES`	Core-active interval for one permutation	Accumulated core-active interval over all internal permutations
`OP_COUNT`	Completed or errored raw DMA transactions	Completed or errored sponge DMA transactions
`LAST_CORE_PERMS`	Not implemented in raw DMA	Number of internal Keccak permutations

have a clear interpretation.

5.5 Completion, Polling, and Counter Collection

The current software model uses polling to detect accelerator completion. Although the hardware exposes interrupt outputs, polling is preferable for the present bare-metal benchmark because it is deterministic, simple to integrate, and easy to include in cycle-based profiling.

Both drivers use the same status interpretation, summarized in Algorithm 15.

Algorithm 15 Polling-based accelerator completion handling

```
1: while timeout remains do
2:     status ← read STATUS
3:     if STATUS.ERROR is set then
4:         return hardware error
5:     end if
6:     if STATUS.DONE is set then
7:         return success
8:     end if
9: end while
10: return timeout
```

The same counter name does not always represent the same abstraction level across the two backends. In raw DMA, `LAST_OP_CYCLES` describes one read-permute-write transaction. In sponge DMA, `LAST_OP_CYCLES` describes a complete SHAKE256 transaction, including absorb, padding, internal permutations, and squeeze.

Table 5.7: HQC backend-selection macros.

Macro	Meaning
`HQC_USE_KECCAK_DMA`	Enables raw Keccak DMA at the permutation boundary
`HQC_USE_KECCAK_SPONGE_DMA`	Enables sponge DMA for HQC G/K SHAKE256

Table 5.8: HQC application modes used for backend comparison.

Application mode	Raw DMA	Sponge DMA	Behavior
`HQC*_software`	disabled	disabled	Entire Keccak stack runs on the CPU
`HQC*`	enabled	disabled	Raw DMA accelerates visible Keccak permutations
`HQC*_hybrid_dma`	enabled	enabled	HQC G/K uses sponge DMA; remaining permutations use raw DMA

The software also measures CPU-visible time with the RISC-V `mcycle` counter. These measurements include driver calls, MMIO writes, polling, packing and unpacking, fallback checks, and result handling. Therefore, software-visible Keccak cycles and hardware core cycles measure different layers of the execution stack.

5.6 Backend Selection and Build Configuration

Backend selection is performed at compile time using application-local macros. These macros are compiled into separate binaries; they are not runtime switches. This makes each binary correspond to a specific experiment configuration.

For HQC applications, two macros define the backend, as shown in Table 5.7.

The resulting HQC modes are summarized in Table 5.8.

For PQClean-based ML-KEM, ML-DSA, and SPHINCS+-SHAKE applications, raw DMA is controlled by `PQCLEAN_USE_KECCAK_DMA`. These applications do not use the current HQC sponge DMA backend because

Table 5.9: Buffer and alignment requirements for the DMA backends.

Requirement	Raw DMA	Sponge DMA
Source alignment	4-byte aligned	4-byte aligned
Destination alignment	4-byte aligned	4-byte aligned
Data movement	200-byte Keccak state	Variable-length input and output
Staging strategy	Static 50-word input/output buffers	Direct input; staged output if needed
Typical HQC output	200-byte state	64-byte G/K SHAKE256 output

their SHAKE/SHA3 usage includes general incremental and one-shot patterns beyond the fixed one-shot SHAKE256 interface used by HQC G/K.

Profiling is also configured at compile time. HQC uses `HQC_PROFILE_STAGE` to select all stages or a single KEM stage. Deterministic seeding keeps inputs comparable across backend modes.

5.7 Buffer Management and Alignment

The programming model requires explicit buffer ownership and alignment. The accelerators access SRAM directly, so buffers must remain valid and must not be modified by software while a DMA transaction is active. In the current polling-based benchmark, this is naturally satisfied because the CPU waits for completion before reusing the buffers.

The main memory requirements are summarized in Table 5.9.

Static storage is used for reusable DMA staging buffers and for large cryptographic objects where necessary. This reduces stack pressure and is important for larger parameter sets and signature schemes.

The buffer policy also affects fallback behavior. A bad raw DMA address or length prevents the hardware transaction from starting and forces software permutation. A bad sponge input alignment forces software SHAKE fallback. An unaligned sponge output can be handled by an aligned temporary buffer followed by a software copy, because copying the output does not change SHAKE semantics.

5.8 Profiling and Correctness Reporting

The profiling system separates software-visible metrics from hardware-side DMA metrics. This avoids conflating full application cost, driver overhead,

Table 5.10: Software-visible profiling and correctness metrics.

Metric	Meaning
Stage cycles	Full keygen, encapsulation, decapsulation, signing, or verification cost
Keccak cycles	CPU-visible time around Keccak calls
Keccak permutations	Number of visible `KeccakF1600_StatePermute()` calls
Sponge software-visible cycles	CPU-observed time around sponge DMA calls
Correctness result	Shared-secret match or signature verification result

Table 5.11: Hardware-side DMA profiling metrics.

Metric	Meaning
Raw DMA calls/fallbacks	Successful raw offloads and failed raw attempts
Raw `hw_op` cycles	Hardware raw DMA transaction cycles
Raw `hw_core` cycles	Hardware Keccak-core active cycles
Sponge DMA calls/fallbacks	Successful sponge offloads and failed sponge attempts
Sponge `hw_op` cycles	Complete hardware sponge transaction cycles
Sponge `hw_core` cycles	Accumulated Keccak-core active cycles in sponge DMA
Sponge `hw_perms`	Internal Keccak permutation count in sponge DMA
`last_ret` and `status`	Last sponge driver return code and hardware status word

and accelerator core activity.

Software-visible metrics are summarized in Table 5.10.

Hardware-side metrics are summarized in Table 5.11.

Correctness is reported independently from these counters. For KEM workloads, the benchmark compares encapsulated and decapsulated shared secrets. For signature workloads, it verifies the generated signature. A `PASS` result means the cryptographic output is correct; it does not prove that hardware was used. Hardware usage must be inferred from the call and fallback counters.

5.9 Fallback and Error Semantics

Fallback is treated as part of the programming model rather than as an exceptional debug-only path. The design follows a correctness-first policy:

Table 5.12: Fallback and error semantics.

Failure point	Driver/backend response
Bad length	Return `BadLen`; do not start hardware
Bad address alignment	Return `BadAddr`; use software fallback when possible
Hardware `STATUS.ERROR`	Return hardware error and fall back
Software timeout	Return timeout and fall back
Raw DMA transaction failure	Increment raw fallback counter and disable later raw attempts
Sponge DMA transaction failure	Increment sponge fallback counter and run software SHAKE256
Probe failure	Report `FAIL` so the log identifies unavailable hardware

hardware failure must not change the cryptographic result. It may only affect performance and backend attribution.

The main failure cases and responses are summarized in Table 5.12.

Raw DMA uses a persistent disable policy after failure. Once a raw transaction fails, later permutation calls use software directly. Sponge DMA uses a stricter semantic policy in HQC hybrid mode: if the dedicated sponge transaction fails, the fallback suppresses raw DMA temporarily and computes the G/K SHAKE256 operation in pure software. This ensures that a reported sponge DMA call corresponds only to successful execution on the dedicated sponge IP.

Deterministic seeding strengthens this error model. Since software, raw DMA, and hybrid binaries can run the same input sequence, differences in correctness or timing can be attributed to backend behavior rather than input variation.

5.10 Summary

The software model preserves the reference cryptographic control flow and introduces hardware acceleration only through well-defined Keccak interfaces. Raw DMA accelerates the common permutation boundary, making it broadly applicable across Keccak-based algorithms. Sponge DMA accelerates a complete SHAKE256 operation when the operation semantics match the hardware interface, as in the HQC G/K path.

This programming model provides three experimental modes: a software baseline, a generic raw-DMA acceleration mode, and an HQC hybrid mode. The same correctness checks are used for all modes, while profiling counters

reveal which backend actually executed each Keccak operation. This separation between correctness, software-visible timing, and hardware-internal timing is essential for evaluating the accelerator architecture rigorously.

Chapter 6

Experimental Setup and Evaluation

This chapter evaluates the implemented Keccak acceleration architecture on the FPGA-based X-HEEP platform. The evaluation has three goals. First, it verifies that the raw Keccak DMA and Keccak sponge DMA backends preserve the correctness of the original post-quantum cryptographic implementations. Second, it quantifies the effect of acceleration at different abstraction levels: the full cryptographic operation, the software-visible Keccak layer, and the hardware DMA transaction counters. Third, it discusses the cost of the hardware integration in terms of FPGA resource utilization and timing closure.

The results in this chapter are taken from UART logs captured from the Nexys 4 board. Each benchmark prints the selected backend, hardware probe status, per-stage cycle counts, DMA call counters, fallback counters, hardware-side profiling counters, and final correctness status.

6.1 Evaluation Platform

The experimental platform is an X-HEEP-based RISC-V microcontroller implemented on a Digilent Nexys 4 FPGA board. The FPGA device reported by Vivado is `xc7a100tcsg324-1`. The design was built with Vivado 2024.1 on Ubuntu 22.04.5. The generated system uses the X-HEEP on-chip memory configuration with seven SRAM banks and includes both Keccak accelerator IPs, as summarized in Table 6.1.

The raw Keccak DMA IP and the sponge DMA IP are always present in the evaluated bitstream. Different software binaries select which backend is used at compile time. Therefore, software-only, raw DMA, and hybrid results are collected on the same hardware platform, while differing only in the compiled backend configuration.

For enabled accelerators, the firmware performs a startup probe before the benchmark begins. A probe failure would make the corresponding backend unavailable and would be visible in the UART output. In the complete hardware-accelerated runs used for this evaluation, the raw DMA

Table 6.1: Evaluation platform configuration.

Component	Configuration used in the evaluation
FPGA board	Nexys 4
FPGA device	`xc7a100tcsg324-1`
RISC-V system	X-HEEP / CORE-V mini MCU configuration
Software linker mode	`LINKER=on_chip`
On-chip SRAM configuration	7 memory banks
Generated system clock used by the MCU domain	40 MHz
Raw Keccak DMA base address	`0x20076000`
Keccak sponge DMA base address	`0x20077000`
Raw Keccak DMA OBI master	External master index 4
Keccak sponge DMA OBI master	External master index 5
External crossbar master count	`XHEEP_EXT_XBAR_NMASTER = 6`
Result interface	UART log output

probe passed whenever raw DMA was enabled, and the sponge DMA probe passed whenever the hybrid HQC backend was enabled.

6.2 Host-PC and FPGA Board Communication

The evaluation setup uses the host PC for three distinct tasks: FPGA configuration, software loading/control, and UART log capture. These tasks are separated from the measured cryptographic execution. The PC does not participate in the Keccak computation itself; after the RISC-V program starts, the CPU and the accelerator IPs execute inside the FPGA, while the PC only observes UART output and, when needed, controls the processor through the debug interface.

The bitstream is first downloaded to the FPGA through the JTAG path. This configures the X-HEEP system, including the RISC-V core, SRAM subsystem, peripheral interconnect, raw Keccak DMA IP, and Keccak sponge DMA IP. After configuration, the benchmark ELF is loaded into on-chip memory through the RISC-V debug path. In the implemented workflow, OpenOCD exposes the debug connection and GDB loads the selected application binary, resets or resumes the core, and starts execution.

UART is used as the measurement reporting channel. Each benchmark prints its backend configuration, accelerator probe result, per-stage cycle counts, profiling summaries, correctness result, and final success message.

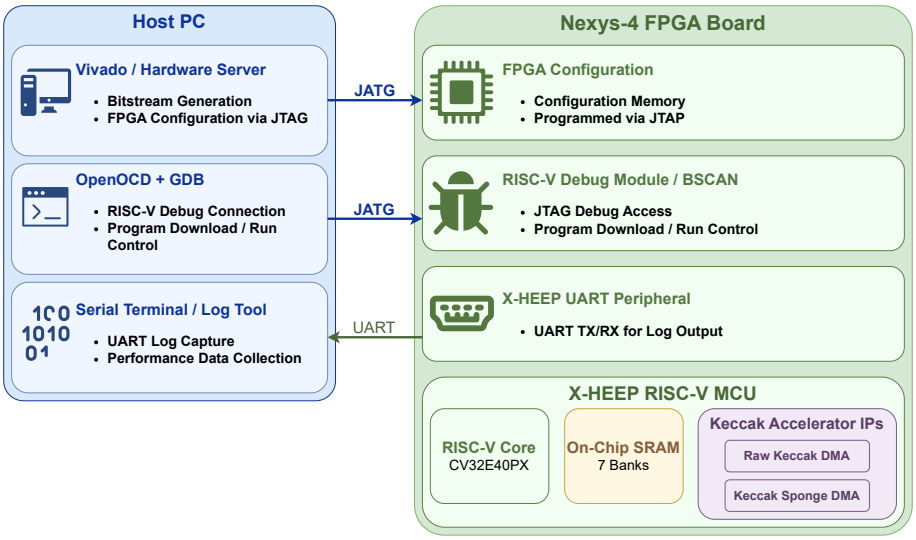

Figure 6.1: Host-PC and Nexys 4 communication paths used during evaluation.

The host PC records this UART stream through the USB serial interface exposed as a COM port. The UART transfer itself is not included inside the measured KEM or signature stage cycle counters, except for the explicit print statements placed before and after stages by the benchmark firmware. The reported cryptographic cycle counts are read from the RISC-V cycle counter around the measured software regions.

The accelerator datapath is therefore fully local to the FPGA. During a raw DMA transaction, the RISC-V core writes memory-mapped control registers, the raw Keccak DMA IP reads and writes SRAM through its OBI master port, and the CPU polls completion. During a sponge DMA transaction, the same host-independent pattern is used, but the sponge IP performs the SHAKE256 absorb, padding, permutation scheduling, and squeeze sequence internally. The host PC only receives the resulting textual report after the firmware prints it through UART.

6.3 Test Workloads

The benchmark suite covers three groups of post-quantum cryptographic workloads. HQC is used to evaluate the raw permutation backend and the HQC-specific hybrid backend; its standardization status is based on the NIST fourth-round selection report [7]. ML-KEM, ML-DSA, and SPHINCS+-SHAKE are used to evaluate the raw Keccak permutation backend on

Table 6.2: Benchmark workloads used for evaluation.

Workload family	Parameter sets	Operation measured	Backends evaluated
HQC	HQC-128, HQC-192, HQC-256	`keypair -> encaps -> decaps`	Software, raw DMA, hybrid DMA
ML-KEM	ML-KEM-512, ML-KEM-768, ML-KEM-1024	`keypair -> encaps -> decaps`	Software, raw DMA
ML-DSA	ML-DSA-44, ML-DSA-65, ML-DSA-87	`keypair -> sign -> verify`	Software, raw DMA
SPHINCS+-SHAKE	`shake-128f-simple`, `shake-192f-simple`	`keypair -> sign -> verify`	Software, raw DMA

additional Keccak-heavy PQClean workloads, corresponding respectively to FIPS 203, FIPS 204, and the SPHINCS+-derived SLH-DSA standard in FIPS 205 [4, 5, 6].

All complete workloads in the UART logs reported `PASS`. For KEM workloads, this means that the shared secret produced by encapsulation matches the shared secret recovered by decapsulation. For signature workloads, this means that the generated signature verifies against the fixed test message and public key. All reported hardware-accelerated runs also reported zero raw DMA fallbacks; hybrid HQC reported zero sponge DMA fallbacks.

6.4 Baseline Software Implementation

The software baseline uses the clean reference-style C implementations integrated into X-HEEP. In the baseline configuration, the Keccak-f[1600] permutation is executed entirely by the RISC-V core through the software `KeccakF1600_StatePermute()` routine. Higher-level SHAKE, SHA3, KEM, and signature control flow is unchanged.

This baseline is important for two reasons. First, it provides a correctness reference: accelerated runs must produce the same cryptographic result as the software-only path under the same deterministic random seed. Second, it provides the cycle baseline for speedup calculations. Since only the Keccak backend is changed, the difference between software and accelerated runs isolates the impact of moving Keccak permutation work to the accelerator, while preserving the cost of the surrounding algorithmic code.

The raw DMA backend intercepts the `KeccakF1600_StatePermute()` boundary. Each software permutation call is converted into a 200-byte DMA transaction: the current Keccak state is packed into an aligned memory buffer, the accelerator reads the state through its OBI master port, performs one Keccak-f[1600] permutation, and writes the updated state back to memory.

The HQC hybrid backend uses two hardware paths simultaneously. HQC G/K `shake256_512_ds()` calls are redirected to the one-shot Keccak sponge DMA backend, while all other Keccak permutation calls continue to use the raw Keccak DMA backend. This is why the hybrid HQC logs show fewer software-visible Keccak permutations: the permutations required by the

Table 6.3: Performance and profiling metrics reported by the benchmark firmware.

Metric	Meaning
Operation cycles	Total cycles for the complete KEM or signature workload
Keccak cycles	Software-visible time spent in the instrumented Keccak layer
Keccak permutations	Number of software-visible `KeccakF1600_StatePermute()` calls
Raw DMA calls/fallbacks	Number of successful raw DMA attempts and software fallbacks
Sponge DMA calls/fallbacks	Number of successful sponge DMA attempts and software fallbacks
Raw `hw_op` cycles	Hardware-side raw DMA transaction cycles
Raw `hw_core` cycles	Hardware-side raw Keccak core-active cycles
Sponge `hw_op` cycles	Hardware-side one-shot SHAKE256 transaction cycles
Sponge `hw_core` cycles	Hardware-side Keccak core-active cycles inside sponge execution
Sponge `hw_perms`	Number of Keccak permutations executed inside the sponge IP

G/K SHAKE256 operations are performed inside the sponge DMA IP and are instead reported through the sponge hardware counter.

6.5 Performance Metrics

The evaluation uses cycle counts rather than wall-clock time as the primary metric. This avoids ambiguity from UART output latency and host-side logging. With the 40 MHz MCU clock used in this implementation, one cycle corresponds to 25 ns; however, the tables retain cycle counts because they are independent of any later clock-frequency changes. The metrics reported by the benchmark firmware are summarized in Table 6.3.

Let $C_{\text{app}}^{\text{sw}}$ and $C_{\text{app}}^{\text{acc}}$ denote the software-only and accelerated end-to-end application operation cycle counts, respectively. Similarly, $C_{\text{kec}}^{\text{sw}}$ and $C_{\text{kec}}^{\text{acc}}$ denote the corresponding software-visible Keccak-layer cycle counts. The

two speedup metrics are defined as:

$$S_{\text{op}} = \frac{C_{\text{app}}^{\text{sw}}}{C_{\text{app}}^{\text{acc}}}, \qquad S_{\text{kec}} = \frac{C_{\text{kec}}^{\text{sw}}}{C_{\text{kec}}^{\text{acc}}}. \tag{6.1}$$

Here, S_{op} denotes the end-to-end speedup visible to the complete cryptographic workload, while S_{kec} denotes the Keccak-layer speedup measured at the instrumented Keccak boundary. Since S_{op} includes all non-Keccak computation and software orchestration, whereas S_{kec} only measures the accelerated Keccak region, the two values can differ substantially. In workloads where Keccak accounts for only a small fraction of the total runtime, a large S_{kec} may therefore produce only a modest S_{op}.

Hardware-side counters should not be interpreted as replacements for the software-visible quantities used in S_{op} and S_{kec}. `hw_op` and `hw_core` are measured inside the accelerator and exclude CPU-side driver work such as packing, register writes, polling, result unpacking, and UART printing. They are used to characterize accelerator latency and DMA overhead, while C_{app} and C_{kec} are used to evaluate application-level and Keccak-layer performance.

6.6 Cycle Count and Latency Analysis

The performance results are separated by workload family. Within each family, one table reports complete cryptographic operation cycles and end-to-end speedup, while a second table reports Keccak-layer profiling and backend attribution. This organization keeps the application-level and Keccak-level measurements separate while making the workload-dependent behavior easier to compare.

6.6.1 HQC

HQC was evaluated in three modes: software, raw DMA, and hybrid DMA. The hybrid backend uses sponge DMA only for the HQC G/K `shake256_512_ds()` path and raw DMA for the remaining Keccak permutations.

The raw DMA backend reduces the software-visible Keccak cost by approximately 27.8× for HQC. The end-to-end improvement is much smaller because the measured HQC implementations are dominated by non-Keccak arithmetic, sampling, coding, and memory operations. For hybrid HQC, the Keccak-layer speedup in Table 6.5 refers to the software-visible Keccak instrumentation point; it does not mean that the sponge-internal permutations disappear. The visible permutation reduction is explained by the G/K SHAKE256 permutations moving inside the sponge DMA IP.

Table 6.4: HQC end-to-end performance.

Workload	Backend	Operation cycles	End-to-end speedup
HQC-128	Software	438,693,969	1.000×
HQC-128	Raw DMA	434,142,077	1.010×
HQC-128	Hybrid DMA	434,093,661	1.011×
HQC-192	Software	720,362,807	1.000×
HQC-192	Raw DMA	711,511,130	1.012×
HQC-192	Hybrid DMA	710,892,525	1.013×
HQC-256	Software	1,512,010,258	1.000×
HQC-256	Raw DMA	1,498,224,204	1.009×
HQC-256	Hybrid DMA	1,494,452,191	1.012×

Table 6.5: HQC Keccak-layer profiling.

Workload	Backend	Keccak cycles	Keccak speedup	Visible perms	Raw calls/fallbacks	Sponge calls/fallbacks
HQC-128	Software	4,732,588	1.0×	173	0/0	0/0
HQC-128	Raw DMA	170,409	27.8×	173	173/0	0/0
HQC-128	Hybrid DMA	71,982	65.7×	73	73/0	4/0
HQC-192	Software	9,164,260	1.0×	335	0/0	0/0
HQC-192	Raw DMA	329,979	27.8×	335	335/0	0/0
HQC-192	Hybrid DMA	131,142	69.9×	133	133/0	4/0
HQC-256	Software	14,361,900	1.0×	525	0/0	0/0
HQC-256	Raw DMA	517,129	27.8×	525	525/0	0/0
HQC-256	Hybrid DMA	200,162	71.8×	203	203/0	4/0

Table 6.6: ML-KEM end-to-end performance.

Workload	Backend	Operation cycles	End-to-end speedup
ML-KEM-512	Software	4,115,781	1.000×
ML-KEM-512	Raw DMA	2,047,423	2.010×
ML-KEM-768	Software	6,734,119	1.000×
ML-KEM-768	Raw DMA	3,291,640	2.046×
ML-KEM-1024	Software	10,350,696	1.000×
ML-KEM-1024	Raw DMA	4,817,830	2.148×

Table 6.7: ML-KEM Keccak-layer profiling.

Workload	Backend	Keccak cycles	Keccak speedup	Visible perms	Raw calls/fallbacks
ML-KEM-512	Software	2,151,091	1.0×	79	0/0
ML-KEM-512	Raw DMA	77,109	27.9×	79	79/0
ML-KEM-768	Software	3,583,636	1.0×	131	0/0
ML-KEM-768	Raw DMA	127,861	28.0×	131	131/0
ML-KEM-1024	Software	5,717,404	1.0×	209	0/0
ML-KEM-1024	Raw DMA	203,989	28.0×	209	209/0

6.6.2 ML-KEM

ML-KEM was evaluated in software and raw DMA modes. Unlike HQC, ML-KEM spends a larger fraction of its execution time in Keccak, so the same raw permutation backend has a stronger effect on end-to-end cycles.

The ML-KEM results show a stable Keccak-layer speedup close to 28×. The end-to-end speedup grows from 2.010× for ML-KEM-512 to 2.148× for ML-KEM-1024 because the larger parameter set performs more Keccak work and exposes more of the accelerated primitive.

6.6.3 Signature Workloads

The signature workloads include ML-DSA and SPHINCS+-SHAKE. These benchmarks use the same raw Keccak DMA backend, but their total speedups differ because their non-Keccak work and number of Keccak permutations differ substantially.

The signature workloads show the clearest end-to-end benefit when the algorithm repeatedly invokes Keccak. SPHINCS+-SHAKE is especially favorable because Keccak dominates the software baseline. All complete entries in the evaluation passed their KEM shared-secret check or signature verification check, and all reported accelerated runs completed without raw DMA fallback.

Table 6.8: Signature workload end-to-end performance.

Workload	Backend	Operation cycles	End-to-end speedup
ML-DSA-44	Software	29,242,636	1.000×
ML-DSA-44	Raw DMA	16,780,199	1.743×
ML-DSA-65	Software	28,509,085	1.000×
ML-DSA-65	Raw DMA	13,406,170	2.127×
ML-DSA-87	Software	56,779,364	1.000×
ML-DSA-87	Raw DMA	27,479,292	2.066×
SPHINCS+-shake-128f	Software	4,195,172,192	1.000×
SPHINCS+-shake-128f	Raw DMA	1,107,528,662	3.788×
SPHINCS+-shake-192f	Software	6,790,219,139	1.000×
SPHINCS+-shake-192f	Raw DMA	1,856,392,849	3.658×

[a]SPHINCS+-128f and SPHINCS+-192f denote the SHAKE simple variants.

Table 6.9: Signature workload Keccak-layer profiling.

Workload	Backend	Keccak cycles	Speedup	Visible perms	Raw calls/fallbacks
ML-DSA-44	Software	13,015,462	1.0×	478	0/0
ML-DSA-44	Raw DMA	466,533	27.9×	478	478/0
ML-DSA-65	Software	15,792,820	1.0×	580	0/0
ML-DSA-65	Raw DMA	566,085	27.9×	580	580/0
ML-DSA-87	Software	30,118,956	1.0×	1,101	0/0
ML-DSA-87	Raw DMA	1,074,581	28.0×	1,101	1,101/0
SPHINCS+-128f[a]	Software	3,183,260,703	1.0×	116,907	0/0
SPHINCS+-128f	Raw DMA	114,101,249	27.9×	116,907	116,907/0
SPHINCS+-192f[a]	Software	5,086,894,551	1.0×	186,819	0/0
SPHINCS+-192f	Raw DMA	182,335,361	27.9×	186,819	186,819/0

[a]SPHINCS+-128f and SPHINCS+-192f denote the SHAKE simple variants.

6.7 DMA Transfer Overhead Analysis

The hardware counters provide a lower-level view of the accelerator behavior. For the raw Keccak DMA backend, one operation corresponds to one 200-byte Keccak state permutation. Let $C_{\text{op}}^{\text{raw}}$ denote the raw DMA hardware operation cycles per permutation, and let $C_{\text{core}}^{\text{raw}}$ denote the raw Keccak core-active cycles per permutation. Across the reported raw DMA workloads, the non-core component is estimated as

$$\begin{aligned} C_{\text{op}}^{\text{raw}} &\approx 280 \text{ cycles}, \\ C_{\text{core}}^{\text{raw}} &\approx 26 \text{ cycles}, \\ C_{\text{noncore}}^{\text{raw}} &= C_{\text{op}}^{\text{raw}} - C_{\text{core}}^{\text{raw}} \\ &\approx 280 - 26 \\ &= 254 \text{ cycles}. \end{aligned} \tag{6.2}$$

The 26-cycle core-active count is consistent with the measurement model in Section 4.7. A Keccak-f[1600] permutation requires 24 round cycles in the iterative datapath, while the observed counter adds a small boundary term:

$$C_{\text{core}}/N_{\text{perm}} \approx 24 + \delta_{\text{core}} \approx 24 + 2 = 26 \text{ cycles}. \tag{6.3}$$

Here, δ_{core} is not an additional cryptographic round; it is the start/ready observation and counter-boundary overhead included by the hardware counter. The approximately 254 raw non-core cycles are not Keccak computation. They represent the accelerator-side cost of the transaction around the permutation: DMA read requests, OBI grant/response latency, state buffering, controller sequencing, DMA write requests, write-response completion, and status/counter latching. This distinction is important because it shows why a raw permutation accelerator can still be limited by per-transaction overhead even when the permutation datapath itself is fast.

Table 6.10 summarizes representative per-operation hardware counter values. For raw DMA, the operation unit is one 200-byte Keccak state permutation. For sponge DMA, the operation unit is one HQC G/K SHAKE256 sponge transaction.

The sponge DMA shows the same core-active cost per internal permutation, which confirms that the sponge IP uses the same iterative Keccak core timing model. Its operation-level latency is much larger because one sponge operation performs a complete SHAKE256 transaction rather than a single permutation. The measured `hw_op` value includes input absorption, domain separation and padding, repeated permutation scheduling, output squeezing, memory reads, memory writes, and transaction completion.

The software-visible Keccak cycles are larger than `hw_op` cycles because they include the cost of CPU-side integration. In the raw DMA path, the

Table 6.10: Representative hardware-side counter values.

Backend/workload	Ops	Perms	hw_op/op	hw_core/op	hw_core/perm
Raw DMA, HQC-128	173	173	280.0	26.0	26.0
Raw DMA, HQC-192	335	335	280.0	26.0	26.0
Raw DMA, HQC-256	525	525	280.0	26.0	26.0
Sponge DMA, HQC-128 hybrid	4	100	2,692.0	650.0	26.0
Sponge DMA, HQC-192 hybrid	4	202	5,392.5	1,313.0	26.0
Sponge DMA, HQC-256 hybrid	4	322	8,605.5	2,093.0	26.0

Note: All counter values are in cycles. *Ops* denotes hardware transactions, and *Perms* denotes internal Keccak-f[1600] permutations.

Table 6.11: Full-system placed FPGA resource utilization.

Resource	Used	Available	Utilization
Slice LUTs	32,367	63,400	51.05%
Slice registers	20,951	126,800	16.52%
Block RAM tiles	56	135	41.48%
DSP blocks	5	240	2.08%
Bonded I/O	55	210	26.19%
BUFGCTRL	7	32	21.88%

CPU must pack the 25 software lanes into a 50-word aligned state buffer, write MMIO registers, poll completion, read hardware counters, and unpack the result. Therefore, the software-visible Keccak cycles are expected to be greater than or equal to the raw `hw_op` cycles. The hardware counters are best interpreted as accelerator latency, not as full software call latency.

6.8 Resource Utilization

Table 6.11 reports the placed utilization of the full X-HEEP system bitstream used for the board tests. The numbers include the RISC-V subsystem, SRAM banks, peripheral logic, BSCAN/JTAG support, raw Keccak DMA IP, and Keccak sponge DMA IP. This table therefore represents the implemented FPGA system rather than only the accelerator IPs.

To isolate the approximate area cost of the accelerator blocks, both Keccak IPs were also synthesized out-of-context using dedicated OOC harnesses. The OOC tops expose flat register-bus and OBI-style ports and instantiate the same wrapper-level IP used in the X-HEEP system. These results are useful for comparing the raw permutation backend and the sponge backend, but they are synthesized OOC estimates rather than placed-and-routed system-level utilization. They do not include the RISC-V core, SRAM banks,

Table 6.12: Out-of-context synthesized resource utilization of the Keccak accelerator IPs.

OOC design	Slice LUTs	Slice registers	BRAM tiles	DSP blocks	LUT util.	Register util.
keccak_dma_ooc_top	7,262	3,797	0	0	11.45%	2.99%
keccak_sponge_dma_ooc_top	10,012	3,997	0	0	15.79%	3.15%

Table 6.13: Out-of-context timing summary for the Keccak accelerator IPs.

OOC design	Clock constraint	WNS	TNS	Failing endpoints
keccak_dma_ooc_top	10.000 ns	3.194 ns	0.000 ns	0
keccak_sponge_dma_ooc_top	10.000 ns	3.079 ns	0.000 ns	0

system interconnect, UART, JTAG/BSCAN infrastructure, or board-level I/O buffering.

The OOC results show that both accelerator IPs are implemented entirely with LUT and flip-flop resources; neither IP infers BRAMs or DSP blocks. This matches the design structure described in Chapter 4: the 1600-bit Keccak state buffer, DMA control state, register interface, counters, and round datapath are implemented with registers and combinational logic. The sponge DMA IP uses approximately 2,750 more LUTs than the raw DMA IP, while adding only 200 registers. The LUT increase is mainly attributable to the higher-level SHAKE256 control logic: absorb scheduling, padding, squeeze sequencing, output-length handling, and the additional domain/output-length register path.

The OOC timing reports were generated with a 10 ns clock constraint. Both IPs meet this 100 MHz OOC constraint at synthesis timing analysis.

The full-system post-route physical optimization timing report shows that the final implementation also meets the user-specified timing constraints, as summarized in Table 6.14.

Table 6.14: Full-system post-route timing summary.

Timing metric	Value
Worst negative slack (WNS)	0.024 ns
Total negative slack (TNS)	0.000 ns
Setup failing endpoints	0
MCU generated clock period	25.000 ns
MCU generated clock frequency	40 MHz

The resource results show that the design fits within the Artix-7 100T

device with both Keccak acceleration backends enabled. LUT utilization is the most significant logic cost, while DSP utilization remains low because the Keccak datapath is dominated by bitwise operations rather than arithmetic multipliers. Block RAM utilization is primarily influenced by the configured X-HEEP SRAM banks rather than the Keccak datapath itself.

Post-implementation timing analysis showed that the full-system critical path was located in the CV32E40PX decode logic rather than in either Keccak accelerator. Therefore, the 40 MHz clock used for board evaluation reflects the timing closure point of the complete X-HEEP SoC, not the standalone timing limit of the Keccak DMA IPs.

6.9 Comparison with State-of-the-Art

This section gives a concise comparison with representative RISC-V post-quantum cryptography accelerators. The comparison is intended to position the proposed architecture in the design space rather than to provide a strict ranking, because the designs differ in processor coupling, accelerated function scope, workload, baseline, FPGA platform, frequency, and software stack.

Table 6.15: Representative speedup comparison with prior RISC-V PQC accelerators.

Work	Workload	Baseline	Reported speedup	Interface note
[17]	Kyber/ML-KEM KEM flow	RISC-V software path	7.68×–9.62×	Tightly coupled accelerator
[18]	Kyber KEM and Dilithium signing	CV32E40P baseline	9.89×–12.57×	SIMD processor extension
[19]	Kyber/ML-KEM KEM flow	PULPissimo software path	2.37×–2.52×	Memory-mapped Keccak accelerator
[9]	Keccak-oriented Kyber support	Software Keccak path	2.39×–3.97×	CV-X-IF co-processor
Raw DMA	HQC, ML-KEM, ML-DSA, SPHINCS+ SHAKE	Same X-HEEP software binary family	1.74×–3.79×	Memory-mapped DMA IP
Sponge DMA	HQC G/K SHAKE256 path	Same X-HEEP HQC software path	Workload-specific offload	Memory-mapped DMA IP

Note: Speedups are taken from the cited papers or from the measurements in this thesis. The values are not normalized across processor frequency, FPGA device, implementation flow, software stack, security level, or measurement boundary.

Table 6.16: Platform and measurement-boundary context for the speedup comparison.

Work	Platform/frequency	Reporting boundary
[17]	As reported	End-to-end KEM, multi-kernel
[18]	Zynq-7000, reported at 125–150 MHz	End-to-end KEM/sign, multi-kernel
[19]	Artix-7 class FPGA, frequency as reported	End-to-end KEM, Keccak-only
[9]	As reported	Keccak/co-processor boundary
Raw DMA	Nexys 4, 40 MHz SoC clock	End-to-end workload, Keccak-only backend
Sponge DMA	Nexys 4, 40 MHz SoC clock	One-shot SHAKE256 transaction within HQC hybrid

Note: This context table explains why this comparison should be read as a scope and interface comparison, not as a direct performance ranking.

Table 6.17: Representative resource-utilization comparison context with prior RISC-V PQC accelerators.

Work	Platform/frequency	Reporting boundary	LUTs	FFs	BRAM	DSP
[17]	As reported	Standalone Keccak functional unit	3,847	0	0	0
[18]	Zynq-7000, 125–150 MHz	Increment over baseline processor	+19,325	+6,717	+4	+32
[19]	Artix-7 class FPGA	Full-system increase	+7,128	+3,798	n.r.	n.r.
[9]	As reported	Keccak co-processor wrapper	6,591	3,372	n.r.	n.r.
Raw DMA	Nexys 4 target, 10 ns OOC constraint	OOC accelerator IP	7,262	3,797	0	0
Sponge DMA	Nexys 4 target, 10 ns OOC constraint	OOC accelerator IP	10,012	3,997	0	0

Note: Resource values are taken from the cited papers or from the out-of-context synthesis results in this thesis. OOC denotes out-of-context implementation results for the accelerator IP only. n.r. denotes resources not reported by the cited work. Entries prefixed with "+" are incremental resources over the corresponding baseline system. These rows use different reporting boundaries and should not be interpreted as area-normalized rankings.

The comparison shows that acceleration scope dominates the interpretation of the reported speedups. Multi-kernel processors accelerate lattice-specific operations such as NTT, modular arithmetic, matrix operations, and sampling in addition to Keccak, so their reported application-level speedups describe a broader optimization boundary than the Keccak-only backends in this thesis. The proposed raw DMA backend should therefore be compared primarily as a low-intrusion, memory-mapped point in the Keccak-acceleration design space rather than as a full PQC processor replacement.

Among Keccak-only designs, the main architectural difference is the

interface boundary. A tightly coupled co-processor can reduce invocation overhead by using custom instructions and local state transfer. The proposed raw DMA backend instead preserves the base RISC-V ISA and uses memory-mapped SRAM-backed transactions. This lowers processor intrusion and toolchain dependence, but each permutation includes DMA transfer, MMIO setup, polling, and state packing/unpacking overhead.

6.10 Discussion

The evaluation first shows that the raw DMA backend provides stable Keccak-layer acceleration. Across the ML-KEM, ML-DSA, and SPHINCS+-SHAKE experiments, replacing the software Keccak permutation with the raw DMA accelerator gives a local Keccak speedup close to 28×. This confirms that the hardware permutation datapath behaves consistently across different PQC software stacks and that the observed gains are not limited to a single benchmark.

The end-to-end speedup is smaller because it depends on the fraction of each workload that is originally spent in Keccak. HQC shows only limited improvement because a large part of its runtime remains in code-based operations outside the accelerated Keccak boundary. ML-KEM and ML-DSA expose a larger Keccak component and therefore obtain moderate application-level speedups. SPHINCS+-SHAKE benefits the most, since its execution is dominated by repeated SHAKE/Keccak calls and therefore maps more directly to the accelerated primitive.

The remaining bottleneck is the DMA interface overhead rather than the Keccak datapath itself. Hardware counters show that the core-active part of a raw Keccak-f[1600] permutation is approximately 26 cycles, whereas a full raw DMA transaction takes approximately 280 hardware cycles. The software-visible cost is higher still, because each accelerator invocation also includes state packing, MMIO register setup, completion polling, hardware-counter handling, and state unpacking. These costs explain why a large local Keccak speedup does not translate linearly into the same end-to-end speedup.

Compared with tightly coupled CV-X-IF or register-based accelerator interfaces, the proposed MMIO/DMA design therefore trades some invocation efficiency for lower integration intrusiveness. Custom-instruction and register-coupled designs can reduce interface overhead by moving operands through processor-side extension paths and local accelerator state, but they require processor integration, custom interface support, and often toolchain or microarchitectural changes. In contrast, the proposed backend keeps the base RISC-V ISA unchanged, exposes a conventional memory-mapped pro-

gramming model, reuses the same DMA-oriented mechanism across different PQC workloads, and can fall back to the original software implementation when the accelerator is unavailable. This makes the design less tightly optimized than a dedicated CV-X-IF co-processor, but more portable and reusable as a system-level acceleration block.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis presented, implemented, and evaluated a DMA-enhanced Keccak acceleration architecture for an X-HEEP-based RISC-V system. The design was motivated by the observation that Keccak is both a computational bottleneck and a data-movement bottleneck in post-quantum cryptographic software. While the Keccak-f[1600] permutation maps naturally to hardware, a practical accelerator must also handle the repeated movement of the 1600-bit state between software, memory, and the accelerator interface. The proposed architecture therefore integrates Keccak as memory-mapped accelerator IPs with DMA-style access to on-chip SRAM, allowing the CPU to configure transactions while the accelerator performs bulk state transfers through OBI master ports.

The implementation contains two complementary hardware backends. The raw Keccak DMA backend accelerates the `KeccakF1600_StatePermute()` boundary and is therefore reusable across SHAKE, SHA3, and several Keccak-based PQC schemes. The Keccak sponge DMA backend implements a more specialized one-shot SHAKE256 transaction and is currently used for the HQC G/K `shake256_512_ds()` path. Together, these two backends demonstrate two levels of Keccak acceleration: a broad permutation-level backend that preserves compatibility with existing software, and a narrower sponge-level backend that reduces software interaction when the complete SHAKE operation has stable semantics.

The accelerators were integrated into the X-HEEP system, exposed through memory-mapped control and status registers, and evaluated on the Nexys 4 FPGA platform. The software stack preserves the original cryptographic APIs and uses compile-time backend selection to compare software-only, raw DMA, and hybrid DMA configurations. Correctness was verified through KEM shared-secret checks and signature verification for HQC, ML-KEM, ML-DSA, and SPHINCS+-SHAKE workloads. In the reported hardware-accelerated runs, the raw DMA and sponge DMA probes

passed, correctness checks passed, and no raw or sponge fallbacks were observed.

The measurements show that the raw DMA backend provides a consistent Keccak-layer speedup of approximately 28× across the tested workloads. The hardware counters further show that one raw Keccak transaction takes approximately 280 hardware cycles, while the Keccak core itself is active for approximately 26 cycles. This confirms that the permutation datapath is efficient and that the remaining cost is dominated by DMA transaction overhead, controller sequencing, memory response latency, and software-visible orchestration. The end-to-end improvement depends strongly on the fraction of each workload spent in Keccak. HQC shows only modest total speedup because much of its runtime remains outside the accelerated Keccak boundary. ML-KEM and ML-DSA benefit more directly, while SPHINCS+-SHAKE shows the largest application-level improvement among the tested workloads because it is dominated by repeated Keccak invocations.

The state-of-the-art comparison places the result in context rather than establishing a strict performance ranking. Full lattice-oriented tightly coupled processors such as RISQ-V and recent SIMD RISC-V designs report speedups over a broader optimization boundary because they accelerate several algorithmic bottlenecks, not only Keccak. In contrast, the proposed design is closer in scope to Keccak-only accelerators such as the CF'23 memory-mapped Keccak accelerator and the ISCAS 2025 CV-X-IF Keccak co-processor. Compared with processor-coupled approaches, the proposed DMA-based design preserves the base RISC-V ISA and avoids processor pipeline modification, but pays a measurable per-transaction overhead. The thesis therefore demonstrates a different point in the design space: a low-intrusion, reusable, observable Keccak acceleration block that can be integrated into an existing X-HEEP system and applied across several Keccak-based PQC workloads.

7.2 Limitations

The main limitation of the proposed architecture is that it accelerates only selected Keccak-related boundaries rather than the full cryptographic algorithms. The raw DMA backend replaces `KeccakF1600_StatePermute()`, but it does not accelerate NTT, polynomial arithmetic, sampling, encoding, decoding, or other non-Keccak kernels. Consequently, the end-to-end speedup is bounded by the Keccak fraction of each workload. This is especially visible in HQC, where large portions of the total execution time remain in non-Keccak code. The design should therefore be understood as a Keccak-focused accelerator rather than a complete PQC processor.

A second limitation is the transaction overhead of the raw DMA interface. The Keccak core-active time is approximately 26 cycles per permutation, but a complete raw hardware transaction takes approximately 280 cycles. Software-visible execution is higher still because each offload requires state packing, aligned staging buffers, MMIO configuration, polling, counter reads, and state reconstruction. This overhead is the cost of preserving an external memory-mapped programming model without custom ISA support. It makes the design portable and non-intrusive, but less invocation-efficient than tightly coupled register-based or CV-X-IF-style co-processors.

The current hybrid scheme is also incomplete. In this implementation, the hybrid backend is specific to HQC: only the HQC G/K `shake256_512_ds()` path is redirected to the sponge DMA, while other Keccak calls either use raw DMA or software fallback. This validates the idea of combining sponge-level and permutation-level acceleration, but it is not yet a general hybrid scheme. A more complete design should introduce a general hybrid backend that can decide, across algorithms, whether a call should use raw permutation DMA, one-shot sponge DMA, incremental sponge hardware, or software. Such a backend would be necessary to extend sponge-level acceleration beyond HQC and to apply it systematically to ML-KEM, ML-DSA, SPHINCS+-SHAKE, and other Keccak-based workloads.

The sponge DMA itself is currently limited to one-shot SHAKE256 semantics. It does not yet support SHAKE128, SHA3 fixed-output hashes, cSHAKE-like customization, or a general incremental absorb/squeeze interface. As a result, it cannot replace the general software sponge layer used by all algorithms. The raw DMA backend remains the only broadly reusable backend in the present implementation.

The evaluation also has practical limitations. The prototype was validated on an FPGA implementation of X-HEEP rather than on an ASIC, and the measured system frequency is affected by full-SoC timing closure rather than by the Keccak IPs alone. The resource numbers include both full-system placed utilization and out-of-context accelerator estimates, but they are not equivalent to a final ASIC area or energy measurement. The thesis also does not evaluate side-channel resistance, fault-injection resistance, or energy consumption. These aspects are important for real cryptographic deployment and would require additional countermeasures and measurement infrastructure.

7.3 Future Work

The most direct extension is to develop a general hybrid Keccak backend. Instead of hard-coding the sponge DMA path only for HQC G/K, the

software and driver layers should expose a common backend-selection policy that can choose between software Keccak, raw DMA, sponge DMA, and future incremental sponge hardware. Such a policy could use operation type, input length, output length, domain separator, alignment, and expected permutation count to select the most appropriate backend. This would make the hybrid approach systematic rather than algorithm-specific.

A second direction is to generalize the sponge DMA accelerator. The current one-shot SHAKE256 engine could be extended into a more complete SHAKE/SHA3 accelerator supporting SHAKE128, SHAKE256, SHA3-256, SHA3-512, domain customization, and incremental absorb/squeeze operation. This would move more of the sponge construction into hardware and reduce repeated boundary crossings between software and the raw permutation accelerator. A general sponge engine would be especially useful for ML-KEM, ML-DSA, and SPHINCS+-SHAKE, where the software currently performs the absorb and squeeze phases around raw permutation calls.

The raw DMA interface could also be optimized to reduce per-transaction overhead. Possible improvements include batching multiple permutations, retaining state in the accelerator across consecutive calls, adding command queues, using interrupt-driven completion for longer operations, or providing a streaming interface for sponge-like workloads. These changes would reduce the cost of MMIO setup, polling, and repeated memory round trips while preserving the external accelerator model.

Another future direction is to explore mixed integration styles. A CV-X-IF or custom-instruction interface could be used for very small, frequent Keccak operations, while the DMA interface could remain available for larger memory-resident transactions. Such a combined architecture would allow the system to trade off invocation overhead and integration intrusiveness dynamically. It would also provide a more direct comparison between register-coupled and DMA-coupled Keccak acceleration within the same X-HEEP-based platform.

Finally, future work should broaden the evaluation methodology. ASIC synthesis would provide more meaningful area, timing, and power estimates than FPGA validation alone. Energy-per-operation measurements would clarify whether the cycle reductions translate into practical efficiency gains. Security-oriented evaluation should consider side-channel leakage, fault attacks, and constant-time behavior of both the hardware and driver paths. Extending the benchmark suite to additional PQC schemes and larger input distributions would further test the generality of the proposed Keccak acceleration model.

References

[1] D. Ott, C. Peikert, et al., "Identifying Research Challenges in Post Quantum Cryptography Migration and Cryptographic Agility," *CoRR*, vol. abs/1909.07353, 2019.

[2] G. Alagic, D. Apon, et al., "Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process," NIST Internal Report 8413, National Institute of Standards and Technology, July 2022, doi: 10.6028/NIST.IR.8413.

[3] National Institute of Standards and Technology, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," Federal Information Processing Standards Publication 202, Aug. 2015, doi: 10.6028/NIST.FIPS.202.

[4] National Institute of Standards and Technology, "Module-Lattice-Based Key-Encapsulation Mechanism Standard," Federal Information Processing Standards Publication 203, Aug. 2024, doi: 10.6028/NIST.FIPS.203.

[5] National Institute of Standards and Technology, "Module-Lattice-Based Digital Signature Standard," Federal Information Processing Standards Publication 204, Aug. 2024, doi: 10.6028/NIST.FIPS.204.

[6] National Institute of Standards and Technology, "Stateless Hash-Based Digital Signature Standard," Federal Information Processing Standards Publication 205, Aug. 2024, doi: 10.6028/NIST.FIPS.205.

[7] G. Alagic, M. Bros, P. Ciadoux, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, H. Silberg, D. Smith-Tone, and N. Waller, "Status Report on the Fourth Round of the NIST Post-Quantum Cryptography Standardization Process," NIST Internal Report 8545, National Institute of Standards and Technology, Mar. 2025, doi: 10.6028/NIST.IR.8545.

[8] A. Dolmeta, S. Di Matteo, E. Valea, M. Carmona, A. Loiseau, M. Martina, and G. Masera, "TYRCA: A RISC-V Tightly-Coupled Accelerator for Code-Based Cryptography," in *2025 Design, Automation & Test in Europe Conference (DATE)*, Lyon, France, 2025, pp. 1–7.

[9] A. Dolmeta, V. Piscopo, M. Mirigaldi, M. Martina, and G. Masera, "RISC-V Based Keccak Co-Processor for NIST Post-Quantum Cryptography Standards," in *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2025, pp. 1–5, doi: 10.1109/ISCAS56072.2025.11043433.

[10] S. Machetti, P. D. Schiavone, G. Ansaloni, M. Peón-Quirós, and D. Atienza, "X-HEEP: An Open-Source, Configurable and Extendible RISC-V Platform for TinyAI Applications," in *2025 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2025, doi: 10.1109/ISVLSI65124.2025.11130281.

[11] S. Deshpande, C. Xu, M. Nawan, K. Nawaz, and J. Szefer, "Fast and Efficient Hardware Implementation of HQC," in *Selected Areas in Cryptography – SAC 2023*, Lecture Notes in Computer Science, vol. 14201, pp. 297–321, Springer, 2024, doi: 10.1007/978-3-031-53368-6_15.

[12] F. Antognazza, A. Barenghi, and G. Pelosi, "An Efficient and Unified RTL Accelerator Design for HQC-128, HQC-192, and HQC-256," *IEEE Transactions on Computers*, vol. 74, no. 7, pp. 2306–2320, 2025, doi: 10.1109/TC.2025.3558044.

[13] Y. Tu, P. He, C.-H. Chang, and J. Xie, "LTE: Lightweight and Time-Efficient Hardware Encoder for Post-Quantum Scheme HQC," *IEEE Computer Architecture Letters*, pp. 1–4, 2024, doi: 10.1109/LCA.2024.3435495.

[14] A. Ras, A. Loiseau, M. Carmona, S. Pontié, G. Renault, B. Smith, and E. Valea, "Optimizing HQC using Frobenius Additive FFT on a RISC-V-based System-on-Chip," presented at the *6th PQC Standardization Conference*, Gaithersburg, MD, USA, Sept. 25, 2025.

[15] J. Lee, W. Kim, S. Kim, and J.-H. Kim, "Post-Quantum Cryptography Coprocessor for RISC-V CPU Core," in *ICEIC 2022*, Jeju, Republic of Korea, 2022, doi: 10.1109/ICEIC54506.2022.9748834.

[16] A. Dolmeta, M. Martina, and G. Masera, "ATHOS: A Hybrid Accelerator for PQC CRYSTALS-Algorithms Exploiting New CV-X-IF Interface," *IEEE Access*, vol. 12, pp. 182340–182352, 2024, doi: 10.1109/ACCESS.2024.3511340.

[17] T. Fritzmann, G. Sigl, and J. Sepúlveda, "RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 4, pp. 239–280, 2020.

[18] Z. Ye, R. Song, H. Zhang, D. Chen, R. C.-C. Cheung, and K. Huang, "A Highly-Efficient Lattice-Based Post-Quantum Cryptography Processor for IoT Applications," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, no. 2, pp. 130–153, 2024.

[19] A. Dolmeta, M. Mirigaldi, M. Martina, and G. Masera, "Implementation and Integration of Keccak Accelerator on RISC-V for CRYSTALS-Kyber," in *Proceedings of the 20th ACM International Conference on Computing Frontiers (CF '23)*, pp. 381–382, Association for Computing Machinery, New York, NY, USA, 2023.