# Dilithium Hardware Accelerator — Keccak/SHAKE Subsystem Documentation

Scope: This document explains the Keccak/SHAKE portion of our Dilithium hardware accelerator, covering the mathematics (Keccak-f[1600] sponge and SHAKE modes), the RTL architecture, the input→output dataflow/handshakes, and a concrete worked example using the included test wrappers. It is written so that a teammate can understand, simulate, and integrate the modules without reading the source first.

## Purpose (Why this module exists)

This subsystem is the cryptographic backbone of Dilithium’s signature generation and verification, supplying deterministic, secure randomness and binding hashes.

## 1) Mathematical Background (What the hardware computes)

### 1.1 Keccak-f[1600] state and sponge construction

* **State**: 1600 bits organized as a 5×5 grid of 64-bit lanes: A[x][y][z] with x,y∈{0..4} and z∈{0..63}.
* **Permutation**: Keccak-f[1600] applies 24 rounds, each composed of the step mappings θ (theta), ρ (rho), π (pi), χ (chi), ι (iota) in that order.
* **Sponge**: To build SHAKE, the sponge absorbs message blocks of size rate r into the state (XOR into the first r bits), applies the permutation, and then squeezes r-bit output blocks between permutations. Security depends on capacity c, where r + c = 1600.

### 1.2 SHAKE modes used by Dilithium

* **SHAKE128**: r = 1344, c = 256 (more output per permutation, lower capacity)
* **SHAKE256**: r = 1088, c = 512 (less output per permutation, higher capacity)
* Dilithium uses **SHAKE256** for hashing message + public key to µ, challenge generation, seed expansion. Our core supports both modes via shake\_mode (0 = SHAKE128, 1 = SHAKE256).

### 1.3 Padding (pad10\*1)

* Standard SHAKE uses multi-rate padding pad10\*1: append a domain-separation byte (e.g., 0x1F for SHAKE) and a final bit 1 at the end of the rate block.
* **Important**: The RTL presented here does not show an explicit padding engine in the buffer; therefore, either:
  1. the caller supplies already-padded blocks and asserts last\_block, or
  2. a small pre-pad module should be placed in front of keccak\_buffer to inject the standard pad bits into the final absorbed block.

We will assume caller-provided padding unless otherwise stated.

## 2) RTL Overview (What the files provide)

### 2.1 Packages and types — keccak\_globals.vhd

Defines shared constants and types: - Lane/state types: k\_lane (64 bits), k\_plane (5 lanes), k\_state (5 planes ⇒ 5×5×64). - SHAKE parameters: RATE\_SHAKE128 = 1344, RATE\_SHAKE256 = 1088, and capacities.

### 2.2 Top permutation & sponge control — keccak.vhd

**Ports** (top-level Keccak/SHAKE core): - **Inputs**: clk, rst\_n, start, din[63:0], din\_valid, last\_block, shake\_mode (0: SHAKE128, 1: SHAKE256) - **Outputs**: buffer\_full, ready, dout[63:0], dout\_valid

**What it does**: - Instantiates: - keccak\_round (one round of θ, ρ, π, χ, ι) - keccak\_round\_constants\_gen (RC for ι) - keccak\_buffer (word packing/unpacking, see below) - Maintains a state register reg\_data : k\_state. - **Absorb**: XORs up to rate\_bits from keccak\_buffer into reg\_data, then iterates 24 rounds. - **Squeeze**: After a permutation, hands the first 256 bits of the state to keccak\_buffer to be serialized as 64-bit words (dout + dout\_valid). - ready/buffer\_full/din\_valid implement a simple handshake with the buffer (see §3).

Note: Only the first 256 bits of the post-permutation state are fed to the buffer for output at a time. Additional output words require switching buffer mode and/or running another permutation as guided by the control FSM.

### 2.3 Round function — keccak\_round.vhd

Implements the 5 step mappings in order: 1) **θ**: column parity, adds parity to each bit in the column. 2) **ρ**: lane-wise rotations by fixed offsets. 3) **π**: lane permutation (re-indexing of (x, y)). 4) **χ**: non-linear step on rows: B[x,y] = A[x,y] XOR ((NOT A[x+1,y]) AND A[x+2,y]). 5) **ι**: XOR round constant RC into A[0,0].

The file wires theta\_in → theta\_out → rho\_in → rho\_out → pi\_in → pi\_out → chi\_in → chi\_out → iota\_in → iota\_out and exports round\_out.

### 2.4 Round constants — keccak\_round\_constants\_gen.vhd

Combinational case on round\_number[4:0] to produce 64-bit RC constants for ι.

### 2.5 Absorb/Squeeze buffer — keccak\_buffer.vhd

**Ports**: - Inputs: clk, rst\_n, start, din[63:0], din\_valid, last\_block, shake\_mode, dout\_buffer\_in[255:0], ready. - Outputs: din\_buffer\_full, din\_buffer\_out[1343:0], dout\_buffer\_out[63:0], dout\_buffer\_out\_valid.

**Behavior**: - Maintains a mode bit: 0 = input (absorbing), 1 = output (squeezing). - **Dynamic rate selection**: rate\_bits = 1344 (SHAKE128) or 1088 (SHAKE256). Also computes rate\_words = 21 or 17 respectively. **Absorbing**: - Accepts 64-bit words with din\_valid until the rate block is full or last\_block is asserted. - Packs words into a buffer\_data[rate\_bits-1:0] vector and asserts din\_buffer\_full when filled. - **Squeezing**: - Latches the 256-bit chunk from dout\_buffer\_in and serializes it into four 64-bit words (dout\_buffer\_out + dout\_buffer\_out\_valid). - After four words, returns to input mode; the core can trigger another permutation to continue squeezing.

### 2.6 BRAM-facing wrapper — keccak\_bram.vhd

A practical wrapper that demonstrates how to feed the core and collect output using a simple FSM: - Generics: ADDR\_WIDTH, DATA\_WIDTH=64 (64-bit BRAM words) - Ports: clk, reset, start, done, ram\_addr, ram\_din, ram\_dout, ram\_en, ram\_we - Internal handshake to the core: keccak\_start, keccak\_din, keccak\_din\_valid, keccak\_last\_block, keccak\_dout, keccak\_valid - States (high level): IDLE → ABSORB → WAIT\_VALID → READ → FINISH - ABSORB: pulses keccak\_start, drives two example 64-bit words, sets last\_block on the second - WAIT\_VALID / READ: collects up to 16 output words (example), writing them into BRAM

This file shows a minimal reference handshake for absorb/squeeze.

### 2.7 32↔64 adapter — keccak\_adapter.vhd

Adapter to 32-bit external memory while keeping a 64-bit BRAM/Keccak interface: - Splits each 64-bit word into two 32-bit transactions when reading/writing external RAM. - FSM S\_IDLE, S\_READ\_LOW, S\_READ\_HIGH, S\_FEED, S\_WAIT, S\_WRITE\_LOW, S\_WRITE\_HIGH, S\_FINISH: - Reads two 32-bit halves → packs into buf64 → feeds keccak\_bram - After squeezing, writes two 32-bit halves per 64-bit output word back to RAM

## 3) Input→Output Path & Handshakes

### 3.1 Absorb path (message → state)

1. **User/Wrapper** presents 64-bit words on din and asserts din\_valid.
2. keccak\_buffer (mode=input) packs incoming words into buffer\_data up to rate\_bits.
3. When a full rate block is ready (or on last\_block), keccak XORs buffer\_data into reg\_data (the 1600-bit state) and starts 24 rounds by driving keccak\_round with round\_constants\_gen.
4. After permutation finishes, permutation\_computed toggles and keccak may proceed to squeeze if output is requested.

**Key signals**: - din\_valid (from caller) enables packing; din\_buffer\_full back-pressures the caller if needed. - last\_block tells the buffer to commit a partial block (used with padding).

### 3.2 Squeeze path (state → output stream)

1. After a permutation, keccak exports the first 256 bits of reg\_data on dout\_buffer\_in[255:0] to keccak\_buffer.
2. keccak\_buffer (mode=output) serializes into four 64-bit words with dout\_buffer\_out\_valid.
3. keccak drives the public dout/dout\_valid from the buffer’s serialized output.
4. To continue squeezing beyond 256 bits, the core returns to absorb/squeeze control: either switch buffer back to input and run another permutation, or perform another output burst per the FSM.

**Key signals**: - ready indicates the permutation engine can accept new buffer data / switch modes. - dout\_valid pulses for each 64-bit output word.

Throughput intuition: In SHAKE256 mode (r=1088), each absorb/permutation can emit 4×64 = 256 bits without another permutation. To get more, run another 24-round permutation on the updated state and repeat.

## 4) Worked Example (walkthrough with keccak\_bram)

Here’s a concrete sequence using the example logic in keccak\_bram.vhd (simplified to illustrate signals):

1. **Reset**: rst\_n=0 clears reg\_data and counters. Release reset (rst\_n=1).
2. **Start Absorb**: Assert start=1 for one cycle. Wrapper sets keccak\_start=1.
3. **Feed two 64-bit words** (example from the file):
   * Cycle T1: din=x"0123456789ABCDEF", din\_valid=1, last\_block=0
   * Cycle T2: din=x"0000000000000000", din\_valid=1, last\_block=1 (final/padded block in a real design)
4. **Permutation**: Core switches to permutation; performs 24 rounds (θ, ρ, π, χ, ι). During this, ready=0.
5. **Squeeze 256 bits**: After rounds, keccak\_buffer outputs four 64-bit words:
   * On four consecutive cycles, observe dout\_valid=1 with dout[63:0] word0..word3.
6. **Collect more output** (optional): The wrapper in keccak\_bram loops to capture up to 16 words by alternating buffer/output bursts with permutations.
7. **Finish**: Wrapper deasserts start to return to IDLE.

This mirrors the provided FSMs: - keccak\_bram: IDLE → ABSORB → WAIT\_VALID → READ → FINISH - keccak\_buffer: toggles between input and output modes depending on whether the core is absorbing or squeezing.

## 5) Integration Notes for Dilithium

* **Mode**: Set shake\_mode='1' (SHAKE256) for Dilithium-critical flows.
* **Padding**: Ensure \*\*pad10\*1padding is applied by the caller on thefinal absorbed block\*\* (domain byte for SHAKE + terminating 1 bit). If this core will own padding, add a small pad unit in front of keccak\_buffer that:
  + ORs domain byte at the current absorb position
  + Sets the final bit in the last rate position, respecting r=1088 for SHAKE256
* **Endianness / lane packing**: 64-bit words are serialized LSB..MSB per the buffer and wired into state lanes in the keccak.vhd absorb logic. When integrating with software or other IP, verify byte order expectations at the boundary (test with known SHAKE256 test vectors).
* **Backpressure**: Obey din\_buffer\_full during absorb and sample dout\_valid during squeeze. The core will ignore bytes beyond the configured rate within a single block.
* **Throughput/latency**: Each 24-round permutation takes a fixed number of cycles determined by the round implementation (combinational vs. one-round-per-cycle). The provided keccak\_round is combinational across the step mappings inside a clocked wrapper, so expect ~1 round per cycle if fully pipelined, or one permutation per 24 cycles plus absorb/squeeze overhead if scheduled sequentially by the FSM.

## 6) Module-by-Module Interface Summary

### keccak.vhd

* **Inputs**: clk, rst\_n, start, din[63:0], din\_valid, last\_block, shake\_mode
* **Outputs**: buffer\_full, ready, dout[63:0], dout\_valid
* **Internals**: reg\_data : k\_state, round\_in/out, counter\_nr\_rounds[4:0], round\_constant\_signal[63:0], reg\_data\_vector[255:0] (for squeeze path)
* **Role**: Top-level; orchestrates absorb (XOR into state), 24-round permutation, and 256-bit squeeze bursts via keccak\_buffer.

### keccak\_buffer.vhd

* **Absorb**: Packs up to rate\_words of 64-bit input into din\_buffer\_out[rate\_bits-1:0], asserts din\_buffer\_full when ready.
* **Squeeze**: Latches dout\_buffer\_in[255:0] and emits 4×64-bit words with dout\_buffer\_out\_valid.
* **Control**: mode toggles between input and output. shake\_mode selects r.

### keccak\_round.vhd

* **Inputs**: round\_in : k\_state, round\_constant\_signal[63:0]
* **Output**: round\_out : k\_state
* **Role**: Implements θ, ρ, π, χ, ι. Rotation amounts and wiring are hard-coded per Keccak-f[1600].

### keccak\_round\_constants\_gen.vhd

* **Input**: round\_number[4:0]
* **Output**: round\_constant\_signal\_out[63:0]
* **Role**: Provides the RC values used by ι.

### keccak\_bram.vhd

* **Inputs**: clk, reset, start, ram\_dout
* **Outputs**: done, ram\_addr, ram\_din, ram\_en, ram\_we
* **Role**: Demonstrates a minimal controller that feeds two input words (example), sets last\_block, waits for output, and writes results to BRAM. Good template for integration.

### keccak\_adapter.vhd

* **Role**: Bridges a 32-bit external RAM to the 64-bit BRAM/core path using a simple FSM (two 32-bit reads per 64-bit word; two 32-bit writes per 64-bit output).

## 7) Typical Usage Pattern (Pseudo-code / Timeline)

reset();

start = 1 for 1 cycle;

// ABSORB one or more 64-bit words up to r=1088 bits (SHAKE256)

for each 64-bit word w in padded\_message\_block:

din = w;

din\_valid = 1;

last\_block = (w is last word of final padded block);

// Core takes over: XOR into state, run 24 rounds

wait until ready=1 and buffer indicates output mode;

// SQUEEZE 256 bits per permutation

repeat as needed:

for i in 0..3:

wait(dout\_valid==1);

collect dout;

if more output required:

trigger next permutation;

else

done;

* This sequence applies equally to SHAKE128 (r=1344) or SHAKE256 (r=1088).
* Padding (pad10\*1) must be handled by the caller unless a dedicated padding unit is added.
* last\_block is critical for correct finalization.

8) Appendix: Signal Quick Reference

**Core I/O (keccak.vhd)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Dir** | **Width** | **Meaning** |
| clk | In | 1 | System clock |
| rst\_n | In | 1 | Active low reset |
| Start | In | 1 | Initialize absorb/permutation |
| Din | In | 64 | Absorb data word |
| Din\_valid | In | 1 | Din is valid for cycle |
| Last\_block | In | 1 | Marks end of padded block |
| Shake\_mode | In | 1 | 0:shake128, 1:shake256 |
| Buffer\_full | Out | 1 | Buffer cannot accept new words |
| Ready | Out | 1 | Core ready to accept input or produce output |
| Dout | Out | 64 | Output word during squeeze |
| Dout\_valid | out | 1 | Dout valud this cycle |

**Buffer I/O (keccak\_buffer.vhd)**

|  |  |  |  |
| --- | --- | --- | --- |
| Signal | Dir | Width | Meaning |
| Din\_valid | In | 1 | Input word strobe |
| Din\_buffer\_full | Out | 1 | Signal for absorb |
| Din\_buffer\_out | Out | 1344/1088 | Packed input block ready for XOR into state |
| Dout\_buffer\_in | In | 256 | First 256 bits of state after permutation |
| Dout\_buffer\_out | Out | 64 | 64 bit output word |
| Dout\_buffer\_out\_valid | out | 1 | Strobe when dout\_buffer\_out is valid |