SPHINCSLET: An Area-Efficient Accelerator for the Full SPHINCS+ Digital Signature Algorithm

SANJAY DESHPANDE*, Yale University, USA
YONGSEOK LEE*, Seoul National University, South Korea
CANSU KARAKUZU, Seoul National University, South Korea and University of Potsdam, Germany
JAKUB SZEFER†, Yale University, USA and Northwestern University, USA
YUNHEUNG PAEK†, Seoul National University, South Korea

This work presents SPHINCSLET, the first fully standard-compliant and area-efficient hardware implementation of the SLH-DSA algorithm, formerly known as SPHINCS+, a post-quantum digital signature scheme. SPHINCSLET is designed to be parameterizable across different security levels and hash functions, offering a balanced trade-off between area efficiency and performance. Existing hardware implementations either feature a large area footprint to achieve fast signing and verification or adopt a coprocessor-based approach that significantly slows down these operations. SPHINCSLET addresses this gap by delivering a 4.7× reduction in area compared to high-speed designs while achieving a 2.5× to 5× improvement in signing time over the most efficient coprocessor-based designs for a SHAKE256-based SPHINCS+ implementation. The SHAKE256-based SPHINCS+ FPGA implementation targeting the AMD Artix-7 requires fewer than 10.8K LUTs for any security level of SLH-DSA. Furthermore, the SHA-2-based SPHINCS+ implementation achieves a 2× to 4× speedup in signature generation across various security levels compared to existing SLH-DSA hardware, all while maintaining a compact area footprint of 6K to 15K LUTs. This makes it the fastest SHA-2-based SLH-DSA implementation to date. With an optimized balance of area and performance, SPHINCSLET can assist resource-constrained devices in transitioning to post-quantum cryptography.

CCS Concepts: • **Security and privacy → Hardware-based security protocols**.

Additional Key Words and Phrases: Post-Quantum Cryptography, PQC, Digital Signatures, SPHINCS+, SLH-DSA

ACM Reference Format:
Sanjay Deshpande, Yongseok Lee, Cansu Karakuzu, Jakub Szefer, and Yunheung Paek. 2025. SPHINCSLET: An Area-Efficient Accelerator for the Full SPHINCS+ Digital Signature Algorithm. *J. ACM* , (2025), 18 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The advent of practical quantum computers can pose a serious threat to the current public-key cryptographic standards we use [12]. As a result, there is an essential need to find alternative means

*Both authors contributed equally to this research.
†Co-Corresponding author: Jakub Szefer, Yunheung Paek.

Authors' addresses: Sanjay Deshpande, sanjay.deshpande@yale.edu, Yale University, New Haven, USA; Yongseok Lee, yslee@sor.snu.ac.kr, Seoul National University, Seoul, South Korea; Cansu Karakuzu, krcansu@sor.snu.ac.kr, Seoul National University, Seoul, South Korea and University of Potsdam, Potsdam, Germany; Jakub Szefer, jakub.szefer@northwestern.edu, Yale University, New Haven, USA and Northwestern University, Evanston, USA; Yunheung Paek, ypaek@snu.ac.kr, Seoul National University, Seoul, South Korea.

ACM 0004-5411/2025/-ART
https://doi.org/XXXXXXX.XXXXXXX

of securing our sensitive information, and this is where post-quantum cryptography (PQC) has emerged as a promising alternative [14, 16, 18]. In this context, the National Institute of Standards and Technology (NIST) has started a standardization effort for PQC algorithms [10] in 2016. The process involved the evaluation of different Key Encapsulation Mechanism (KEM) schemes and Digital Signature schemes.

In 2022, after the third round of evaluation, one KEM scheme and three digital signature schemes were selected for standardization [1]. And among these three digital signature schemes, SPHINCS+ is one of the selected candidates. In August 2024, NIST formally published the final standard (FIPS 205) for the SPHINCS+, the Stateless Hash-Based Digital Signature Algorithm (called SLH-DSA in the standard) [15]. Additionally, a new first round started in July 2023 for additional digital signature candidates.

As the name SLH-DSA suggests, SPHINCS+ is a stateless hash-based signature. The stateless classification of SPHINCS+ is sub-divided into the following mechanism:

(1) The private key used for signing includes several states, and during the signing operation, any state could be chosen at random. Ergo, choosing a specific state over the other states is highly unlikely.
(2) However, even if the choice of states is repeated, the overall signature scheme is still secure.

As recommended in the standard, SLH-DSA comes in three different security levels (128, 192, and 256), and each security level is sub-divided into two variants: a small signature ('s') variant for smaller signature size and a fast signature ('f') variant where the signature generation is relatively faster. The heart of all the variants lies in the underlying hash function used. Each variant of SLH-DSA could be constructed using either the SHA-2 or the SHAKE256 hash functions.

Among other criteria, hardware evaluation of the PQC candidates was also an essential criterion in the NIST's PQC standardization effort. Therefore, various efforts have been made to develop efficient implementations of SPHINCS+. These works can be categorized mainly into two types: hardware-software (HW-SW) codesign and full hardware (HW) design.

Multiple works in the literature present implementations that are based on a HW-SW codesign such as RISC-V interfaced with a hardware hash accelerator [13, 17, 21]. These implementations showcase the modularity of the designs, where users can adapt to different hardware hash modules as per their performance requirements. These works also highlight that the area and memory utilization of different security levels remain the same. This is because the processor controls the operational flow of the algorithm.

In addition to this, there are other works that are finite state machine (FSM) based full HW designs dedicated to accelerating SPHINCS+ [2, 3, 7] signing. While these designs offer compile-time parameterizability to switch between different security levels, the FSM is optimized to work with a specific hash function (e.g., SHAKE256), making it hard to switch to different hash configurations that SPHINCS+ specification [5] proposes.

Our work integrates the most effective design approaches from existing literature to achieve an optimized SPHINCS+ hardware implementation, which we refer to as SPHINCSLET. The proposed design is a low-area, FSM-based, fully hardware architecture that adheres to the FIPS 205 standard[1] for the stateless hash-based digital signature algorithm (SLH-DSA) [15] and is compile-time parameterizable across different security levels and hash functions. A key aspect of this design is the `HASH_TILE` approach, where the finite state machine (FSM) interfaces with a modular `HASH_TILE` block. The `HASH_TILE` is a versatile hash module capable of supporting all hash modes required by SLH-DSA. Moreover, the underlying hash function can be easily swapped, allowing flexibility in

[1]Our hardware design is compliant with the FIPS 205 version released in Aug 2024.

algorithm selection. The presented evaluation includes implementations for SHAKE256 and SHA-2, as recommended by the standard.

While the fully hardware-based designs in [3] achieve high performance, they come at the cost of substantial area overhead. Conversely, the hardware-software co-design approach in [17] reduces area utilization but results in significantly slower performance. The proposed design strikes a balance between these two approaches, offering an optimized trade-off between area and speed. The SHAKE256-based implementation presented in this work is considerably more area-efficient than existing full hardware designs while maintaining high performance. Furthermore, it achieves a 2.5× to 5× speedup in signature generation and a 2.3× to 3.5× speedup in signature verification across various parameter sets compared to [17], with only a marginal increase in area. Additionally, this work presents the fastest SHA-2-based SPHINCS+ hardware implementation to date.

2 CONTRIBUTION

Following is the list of contributions to this work:

- We present the fully standard-compliant, lightweight, SLH-DSA hardware implementation that uses SHAKE256 and SHA-2 (SHA256 and SHA512) as the hash functions for its construction. The design is parameterizable at compile time between different security levels and small and fast variants.
- We present a novel `HASH_TILE` module that is constructed using two hash modules to accomplish different hashing requirements of SLH-DSA. Our `HASH_TILE` module is parameterizable in terms of different hash functions (SHA256, SHA512, and SHAKE256).
- We present a high-speed full-width implementation of SHAKE256 module.
- We present a detailed evaluation of the related work and compare our work with other outdated SPHINCS+ hardware implementations.
- All the presented results could be regenerated using our code. Our code will be available with the open-source license at https://github.com/caslab-code/pqc-hw-sphincslet.

3 BACKGROUND

SLH-DSA utilizes several signature schemes as building blocks: Winternitz One-Time Signature Plus (WOTS$^+$), Forest of Random Subsets (FORS), and the eXtended Merkle Signature Scheme (XMSS). This section provides a brief introduction to these elements and explains the parameters of SLH-DSA for a better understanding of the SLH-DSA signature scheme.

3.1 SLH-DSA Parameter Sets

Table 1 shows the parameter sets of SLH-DSA in the standard, with each parameter defined as follows: n is the security parameter, length of the private key, public key, or signature element in bytes, h is the height of the SLH-DSA hypertree, d is the number of layers in the SLH-DSA hypertree, $logt$ is the number of leaves per hash tree in FORS, k is the number of hash trees in FORS, w is the Winternitz parameter and len is the number of secret values in the WOTS$^+$ private key. Each parameter set can be instantiated using either SHAKE256 or SHA-2 (SHA256 and SHA512) as the underlying hash function.

3.2 Winternitz One-Time Signature Plus

WOTS$^+$ is a one-time signature scheme used in SLH-DSA. Generating the WOTS$^+$ public key involves creating len secret key values, determined by n and w. As shown on line 15 of Algorithm 1, the **PRF** hash function generates each secret value by taking **PK**.seed, **ADRS**, and **SK**.seed as inputs. The public key is obtained through the iterative application of the hash function **F** to the

Table 1. SLH-DSA parameter sets.

Parameter Set	n	h	d	$logt$	k	w	len	Sec. Level
SLH-DSA-128s	16	63	7	12	14	16	35	1
SLH-DSA-128f	16	66	22	6	33	16	35	1
SLH-DSA-192s	24	63	7	14	17	16	51	3
SLH-DSA-192f	24	66	22	8	33	16	51	3
SLH-DSA-256s	32	64	8	14	22	16	67	5
SLH-DSA-256f	32	68	17	9	35	16	67	5

Algorithm 1: *wots_sign*, Signature Generation of WOTS$^+$

Input: Input Message M, secret seed **SK**.seed, public seed **PK**.seed, address **ADRS**)
Output: WOTS$^+$ signature sig

1 $csum = 0$
2 $msg = base\_2^b(M, lg_w, len_1)$
3 **for** i **from** 0 **to** $len_1 - 1$ **do**
4 $csum = csum + w - 1 - msg[i]$
5 **end for**
6
7 $csum = csum \ll ((8 - ((len_2 \cdot lg_w) mod 8)) mod 8)$
8 $msg = mgs \parallel base\_2^b\left(toByte\left(csum, \left\lceil \frac{len_2 \cdot lg_w}{8} \right\rceil\right), lg_w, len_2\right)$
9
10 $skADRS$ = **ADRS**
11 $skADRS.setTypeAndClear(WOTS\_PRF)$
12 $skADRS.setKeyPairAddress(ADRS.getKeyPairAddress())$
13 **for** i **from** 0 **to** $len_1 - 1$ **do**
14 $skADRS.setChainAddress(i)$
15 sk = **PRF**(**PK**.*seed*, **SK**.*seed*, $skADRS$) // Compute **PRF** hash function
16 **ADRS**.$setChainAddress(i)$
17 $sig = chain(sk, 0, msg,$ **PK**.*seed*, $skADRS)$ // Compute *chain* operation of WOTS$^+$
18 **end for**
19 **return** sig

Algorithm 2: *chain*, Chain operation of WOTS$^+$

Input: Input string X, start index i, number of steps s, public seed **PK**.seed, address **ADRS**)
Output: Value of **F** iterated s times on X

1 **if** $(i + s) \geq w$ **then**
2 **return** $NULL$
3 **end if**
4
5 $tmp = X$
6
7 **for** j **from** i **to** $i + s - 1$ **do**
8 **ADRS**.$setHashAddress(j)$
9 tmp = **F**(**PK**.*seed*, **ADRS**, tmp) // Compute **F** in chain operation of WOTS$^+$
10 **end for**
11 **return** tmp

secret keys for $w - 1$ iterations, known as the *chain* function, as shown on line 17 of Algorithm 1 and Algorithm 2. When it sets $i = 0$, the iteration number of the hash function **F** sets to $s = w - 1$. After computing *len* public values, they are compressed into a single n-byte value using a tweakable hash function. WOTS$^+$ is utilized as *wots_sign* with the XMSS Merkle tree as shown in Algorithm 3. The *wots_sign* function combines the hash function **PRF** and the *chain* function at each chain i in the range of $0 < i < len - 1$.

Algorithm 3: *ht_sign*, Signature Generation of XMSS with WOTS$^+$

Input: n-byte message M, secret seed **SK**.seed, index idx, public seed **PK**.seed, address **ADRS**)
Output: Signature SIG_{XMSS}

```
for j from 0 to h' − 1 do
    k = ⌊idx/2^j⌋ ⊕ 1
    AUTH[j] = xmss_node(SK.seed, k, j, PK.seed, ADRS) // Compute XMSS Merkle trees
end for

ADRS.setTypeAndClear(WOTS_HASH)
ADRS.setKeyPairAddress(idx)
sig = wots_sign(M, SK.seed, PK.seed, ADRS) // Compute WOTS+
SIG_XMSS = sig||AUTH
return SIG_XMSS
```

3.3 eXtended Merkle Signature Scheme (XMSS)

The eXtended Merkle Signature Scheme (XMSS) is one of the key building blocks of the SLH-DSA. Algorithm 3 shows the signature generation of XMSS with use of WOTS$^+$, this algorithm represents the *ht_sign* operation, which is used on line 21 of Algorithm 4 (the top-level SLH-DSA signature generation algorithm). The *ht_sign* includes the signature generation of XMSS and the computation of the public key from the signature. XMSS utilizes a Merkle tree structure that stores the WOTS$^+$ public keys as leaf nodes. This tree is formed by hashing child nodes to generate parent nodes, ending at the root node. A leaf node is selected for WOTS$^+$ signature generation. To reduce the chance of choosing the same leaf node repeatedly, SLH-DSA employs d layers of trees, referred to as the hypertree. In the bottom layer, the WOTS$^+$ leaf node signs the FORS public key, while in upper layers, the leaf node signs the lower tree's root node for the XMSS.

3.4 Forest of Random Subsets (FORS)

FORS is referenced on line 14 and 17 of Algorithm 4. In line 14 of Algorithm 4, FORS is utilized at the lowest layer and comprises k Merkle trees, with leaf nodes generated through the hashing of FORS secret keys. In line 17 of Algorithm 4, the k tree root nodes are compressed into a single n-byte valued node, forming the FORS public key.

3.5 Complete SLH-DSA

From SLH-DSA, we implement signature generation and signature verification, the algorithms for which are specified in FIPS 205 [15]. These algorithms build on the prior SPHINCS+ [5] submission. We expect key generation to be less frequent, and it can be implemented on standard CPUs, thus we do not provide custom hardware design for it. Key generation mainly computes random bit generator and XMSS tree to generate an SLH-DSA key pair (PK, SK). During the signature generation and verification, SLH-DSA scheme uses six different functions, namely $\mathbf{PRF}_{\text{msg}}$, $\mathbf{H}_{\text{msg}}$, **PRF**, $\mathbf{T}_\ell$, **PRF**, and **F** from [15, Section 4]. Each of these functions utilizes the designated hash function (SHAKE256 or SHA256 or SHA512) for the SLH-DSA instantiations.

3.5.1 Signature Generation. As shown in Algorithm 4, the complete signature generation procedure combines the WOTS$^+$, XMSS and FORS schemes (from §3.2, §3.3, §3.4) for quantum resilience. FORS is used to sign the message digest, creating a hypertree structure for the FORS public key to produce an SLH-DSA signature. Extracted bits from the message digest are utilized for signing with the FORS key, selecting an XMSS tree, and choosing a WOTS$^+$ key alongside its corresponding FORS key within that specific XMSS tree.

Algorithm 4: *slh_sign*, Signature Generation of SLH-DSA

Input: Message M, private key SK=(**SK**.seed, **SK**.prf, **PK**.seed, **PK**.root)
Output: Signature SIG

ADRS = $toByte(0, 32)$ // Initialize state
opt = **PK**.$seed$
R = $\mathbf{PRF}_{msg}(\mathbf{SK}.prf, opt, M)$
$SIG = SIG \parallel R$

$digest$ = $\mathbf{H}_{msg}(R, \mathbf{PK}.seed, \mathbf{PK}.root, M)$ // Compute message digest and index
tmp_md = first floor $((ka+7)/8)$ bytes of digest
tmp_idx_{tree} = next floor $((h - h/d + 7)/8)$ bytes of digest
tmp_idx_{leaf} = next floor $((h/d + 7)/8)$ bytes of digest
md = first ka bits of tmp_md
idx_{tree} = first $h - h/d$ bits of tmp_idx_{tree}
idx_{leaf} = first h/d bits of tmp_idx_{leaf}

$SIG_{FORS} = FORS\_sign(md, \mathbf{SK}.seed, \mathbf{PK}.seed, \mathbf{ADRS})$ // Compute FORS Merkle trees
$SIG = SIG \parallel SIG\_FORS$

$PK_{FORS} = fors\_pkFromSig(sig\_fors, md, \mathbf{PK}.seed, \mathbf{ADRS})$ // Compute Hash from FORS Merkle tree roots

ADRS.$setType(TREE)$
$SIG_{HT} = ht\_sign(PK_{FORS}, \mathbf{SK}.seed, \mathbf{PK}.seed, idx_{tree}, idx_{leaf})$ // Compute WOTS$^+$ and XMSS Merkle trees
$SIG = SIG||SIG_{HT}$

return SIG

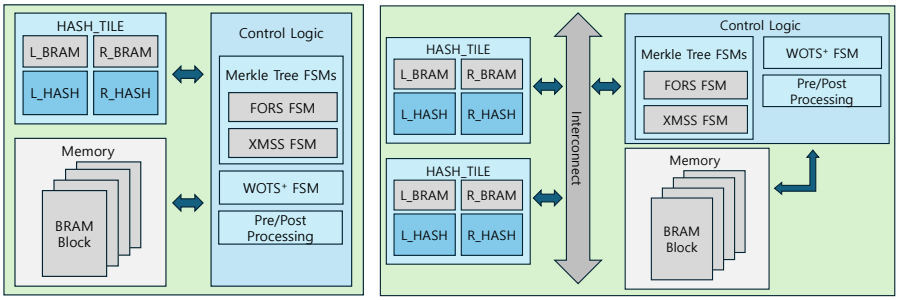
3.5.2 Signature Verification. Signature verification involves computing the message digest and reconstructing trees up to the hypertree root. It uses the same hash functions and similar operations for signature generation. For example, in WOTS$^+$, the signature verification uses the same *chain* operation of Algorithm 2 with $s = w - 1 - msg$. Hence, we do not discuss details of signature verification in this paper, and it can be obtained in [15].

4 RELATED WORK

Amiet et al. [2] introduced an FPGA-based accelerator for SPHINCS-256 [6], which represents the earlier version of SPHINCS+ with different internal operations and hash functions. Their SPHINCS-256 architecture comprises a control unit, a BLAKE-256 hash module, a highly pipelined ChaCha12 hash module. While they presented the design with fast signing and verification, it resulted in significant resource utilization, e.g., 19K LUTs, 38K FFs, and 36 BRAMs.

Following the proposal of SPHINCS+ [5] for the NIST PQC standardization project, Amiet et al. [3] presented an updated SPHINCS+ hardware implementation. They utilized a fully unrolled KECCAK pipeline, the core element of the SHAKE256 module, and targeted fast signing. Nonetheless, their implementation of SPHINCS+ utilized even greater resources, peaking at 51K LUTs, 74K FFs, and 22.5 BRAMs. They introduced a fault attack by inducing supply-voltage glitches in the SPHINCS+ FPGA implementation. To mitigate this vulnerability, they recommended duplicating the hardware. However, adopting this countermeasure would result in a doubling of the resource usage.

In order to achieve a more area-efficient SPHINCS+ core, Berthet et al. [7] proposed a SPHINCS+ implementation that utilizes the SHA256 hash function, which occupies small resources with 6K LUTs, 5K FFs, and 0.5 BRAM. Their design prioritized low-area usage but resulted in slower signing and verification compared to the previous hardware implementation of [3], lagging behind by approximately a factor of 100. As mentioned earlier in this paper in §3.5, FIPS 205 mandated changes with respect to the SPHINCS+ specification, replacing SHA256 with SHA512 hash function for $\mathbf{H}_{\text{msg}}$, $\mathbf{PRF}_{\text{msg}}$, $\mathbf{H}$, and $\mathbf{T}_{\ell}$ functions. This would mean incorporating an additional SHA512 core alongside



(a) SPHINCSLET module with one HASH_TILE

(b) SPHINCSLET module with two HASH_TILEs

Fig. 1. Overview diagram of our hardware architecture.

the existing SHA256 core, leading to an increase in the overall area. Therefore, the SPHINCS+-SHA256-256f-simple and SPHINCS+-SHA256-256s-simple implementations of SPHINCS+ in [7] are no longer compliant with the FIPS 205 specification.

Another area-efficient SPHINCS+ implementation was given by Wagner et. al in [21]. The implementation is targeted at a RISC-V-based secure boot system. They focused on SHA-2-based SPHINCS+ with a 32-bit RISC-V processor, which shows signature verification faster than software. [13] presented signature generation and verification of SHAKE256-based SPHINCS+ with a 32-bit RISC-V processor. They occupy 24K LUTs, 12K FFs in AMD Ultrascale+, which shows lower clock cycles than software. [17] implemented SHAKE256-based and SHA-2-based SLH-DSA with a 32-bit RISC-V processor with the hardware hash modules. They accelerated design by implementing the hardware hash modules and focused on firmware control to optimize the padding format for additional improvement. The RISC-V processor communicates with the hardware hash modules occupying 14K LUTs, 9K FFs, 32 BRAMs (for an integrated design combining both SHA-2 and SHAKE256 modules), and showing lower clock cycles than previous works.

In this paper, we introduce a specification-compliant, area-efficient SLH-DSA implementation using SHAKE256, SHA256, and SHA512. Given the highly parallelizable nature of SPHINCS+ operations, we chose to employ two low-area hash modules to construct a `HASH_TILE` that handles all the hashing requirements of the SLH-DSA described in §5.1, significantly boosting the speed of our implementation. Notably, our work also incorporates all the parameter sets and variants of SLH-DSA and implements the modifications outlined in the FIPS 205 standard.

5 SPHINCSLET - HARDWARE DESIGN

Figure 1 illustrates the design overview of our `SPHINCSLET` implementation. Our `HASH_TILE` module is utilized to support the hash operations in SLH-DSA, accompanied by Finite State Machine (FSM), which is also referred as the Control Logic in Figure 1. The FSM includes the WOTS$^+$, Merkle tree, and pre/post processing sub-modules and manages the `HASH_TILE` to perform the SLH-DSA operations. Input and output data for the `HASH_TILE` is effectively managed by storing odd and even indexed nodes in two internal memories, optimizing the signature generation and verification. Based on the security level and the underlying hash function used in the SLH-DSA design, we use two different types of hardware design architectures for our SPHINCSLET design shown in Figure 1. For all SHAKE-256 based SPHINCSLET and {128s, 128f} variants of SHA-2 based SPHINCSLET we use Figure 1a and for {192s, 192f, 256s, 256f} variants of SHA-2 based SPHINCSLET we use Figure 1b.

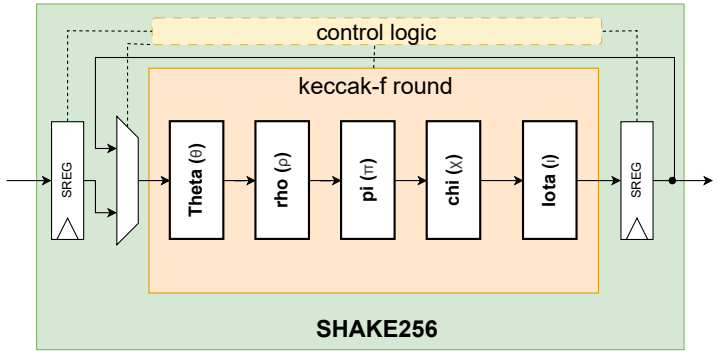


Fig. 2. Hardware block design of SHAKE256 module.

5.1 HASH_TILE

In this section, we present the details of our HASH_TILE hardware module. The hardware block design of the HASH_TILE is shown in Figure 3. The HASH_TILE performs the hash function and pseudorandom function operations described in the [15, Section 4] in an efficient processor-based form. The HASH_TILE is parameterized to work with either SHAKE256, or SHA256, or SHA512 hash functions to cater the needs of SHAKE256-based SPHINCS+ and SHA-2-based SPHINCS+ designs respectively. We build our HASH_TILE using two hash (SHAKE256 or SHA256 or SHA512) modules, L_HASH and R_HASH as shown in the Figure 3.

5.1.1 SHAKE256-based HASH_TILE. To construct the SHAKE256-based HASH_TILE, we design our own SHAKE256 hardware module.

SHAKE256. SHAKE256 is an extendable-output function (XOF) based on the Keccak sponge construction [8]. SHAKE256 allows generating output of any desired length unlike fixed-length hash functions (e.g., SHA3-256, SHA3-512). It uses the Keccak-f permutation function in its construction. Keccak-f is a sponge function that operates on a state of 1600 bits. The core operation of Keccak is the Keccak-f[b] permutation, which consists of multiple rounds of transformation. The number of rounds is determined by $12 + 2 \times log_2(w)$ (where w is the lane size). In the SHAKE256 setting of the keccak, $w = 64$. Therefore, the number of rounds is 24. Each round consists of five steps: θ - diffusion, ρ - bit rotation, π - lane permutation, χ - non-linearity, and ι - Xoring a round constant.

Figure 2 shows the hardware design of our SHAKE256 module. In our hardware implementation of keccak-f round function, { θ, ρ, π, χ, and ι } are implemented using combinatorial logic. Our keccak-f round function implementation is a full-width round-based iterative architecture. Hence, after each round, the output is registered and fed back into the round function. Although our keccak-f round function implementation is full width, we restrict the SHAKE256 to a 64-bit input and output interface. This is because all the data fed into the SHAKE256 module is loaded from a BRAM. Hence, having a wider data width would require larger block RAM utilization.

5.1.2 SHA-2-based HASH_TILE. As specified in §3.5, the SHA-2-based HASH_TILE is constructed either with SHA256 hash function or SHA512 hash functions depending on the security level of SLH-DSA. Both SHA256 and SHA512 belong to SHA-2 family [9].

- SHA256 generates a 256-bit (32-byte) hash value by processing 512-bit input blocks. It uses Merkle-Damgård structure with Davies-Meyer compression in its construction. It utilizes 64 rounds of bitwise operations, modular additions, and logical functions.

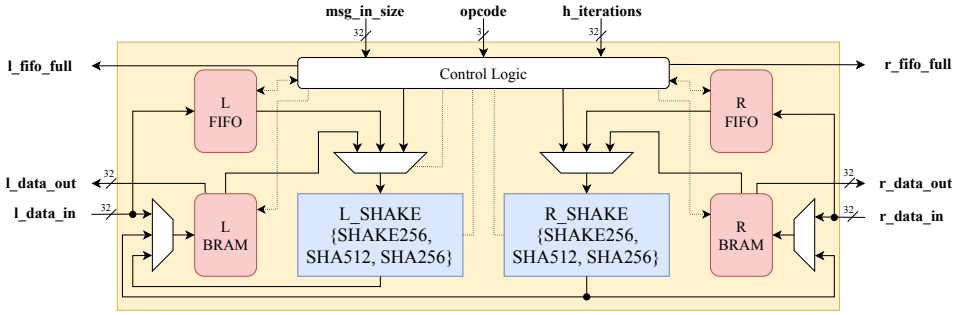


Fig. 3. Hardware block design of HASH_TILE module.

Table 2. Comparison of time and area of the hash modules targeting the AMD Artix 7 xc7a200t-3 FPGA. * Frequency is constrained to 100 MHz, F_{max} not reported.

Hash Variant	Area				Time	
	LUT	DSP	FF	BRAM	Cycles	F_{max} (MHz)
SHAKE256 [our]	4,383	0	2,708	0	24	250
SHAKE256 [17]	5,443	0	2,445	0	24	100*
SHAKE256 [11]	4,797	0	1,845	0	74	166
SHA256 [20]	1,758	0	1,052	0	66	113
SHA512 [19]	3,175	0	2,101	0	82	110

- The construction of SHA512 is also similar, but it operates on 1024-bit input blocks and produces a 512-bit (64-byte) hash. It uses 80 rounds and a larger word size (64-bit vs. 32-bit in SHA-256).

In security level 1, as shown in Figure 1a, we use a single SHA256-based HASH_TILE, which is constructed using two SHA256 hash functions. In security level 3 or 5, as shown in Figure 1b, we require two HASH_TILEs: one based on SHA256 and one based on SHA512. Each HASH_TILE is constructed using its respective two hash functions. It is noted as SHA256+SHA512-based HASH_TILE in Table 3. We utilize the SHA256 module referenced in [20] and the SHA512 module referenced in [19]. These implementations are employed in constructing our HASH_TILE due to their area efficiency, making them suitable for our low area designs. Their construction is a full-width, round-based implementation that requires external control logic to inform the modules about incoming block size, the last block, and padding. We engineer a wrapper around the SHA256 and SHA512 modules with a 64-bit interface that can manage all necessary control logic. Furthermore, it has the ability to load inputs sequentially by retrieving them from L_BRAM and R_BRAM, as illustrated in Figure 3.

Table 2 illustrates the time and area performance of various hash modules utilized in our design. First, when we assess our SHAKE256 implementation against other efficient open-source implementations in the literature, it is evident that our design outperforms them both in terms of area and time efficiency. Second, when we compare SHA256 versus SHA512, the area is doubled due to the state size and operating word size being increased from SHA256 to SHA512.

Both SHAKE256-based HASH_TILE and SHA-2-based HASH_TILE perform a set of predefined hashing and pseudorandom function operations described in the [15, Section 4] in a processor-based form. Since the HASH_TILE follows a processor-like architecture, we enable all required operations through a set of opcodes. In the following list, we provide the opcodes used in both SHAKE256-based and SHA-2-based HASH_TILE design.

(1) **T_L**: It represents the $\mathbf{T}_\ell$(**PK**.seed, **ADRS**, M_ℓ) from [15, Section 4] in our hardware design. It is a hash operation that maps an $n + 32 + n \times k$-byte input or an $n + 32 + n \times len$-byte input to an n-byte output. The specific input size depends on the chosen operation, which includes WOTS$^+$ public key compression and FORS root compression.
(2) **PRF**: It represents the **PRF**(**PK**.seed, **SK**.seed, **ADRS**) from [15, Section 4] in our hardware design. It is a pseudorandom function that generates the secret values in WOTS$^+$ and FORS private keys. It take $2n + 32$-byte input and generates n-bytes of pseudorandomness.
(3) **H**: It represents the **H**(**PK**.seed, **ADRS**, M_2) from [15, Section 4] in our hardware design. It is a special case of $\mathbf{T}_\ell$, which takes a $3n + 32$-byte input and generates hash of n-bytes. This function is used in Merkle tree generation in both FORS and XMSS.
(4) **H_MSG**: It represents the $\mathbf{H}_{msg}$(R, **PK**.seed, **PK**.root, M) from [15, Section 4] in our hardware design. It is used for computing the message digest of the message to be signed. It takes ($3n$ + Message Size) bytes as input and generates $(k \times log(t) + 7)/8 + (h - h/d + 7)/8 + (h/d + 7)/8$-byte output.
(5) **F**: It represents the **F**(**PK**.seed, **ADRS**, M_1) from [15, Section 4] in our hardware design. It is a hash function that takes an $2n + 32$-byte input and produces an n-byte output. It is used in iterating over WOTS$^+$ chains and generating the FORS leaves.
(6) **PRF_MSG**: It represents the $\mathbf{PRF}_{msg}$(**SK**.prf, $OptRand$, M) from [15, Section 4] in our hardware design. A pseudorandom function that generates the randomizer for the randomized hashing of the message to be signed. It takes (n + 32+ Message Size)-byte input and generates n-bytes of randomness.
(7) **F_WOTS_PLUS_h**: This is a special case which we design in our HASH_TILE using the **F** case. While computing the WOTS$^+$ public key and WOTS$^+$ signature, the **F** operation needs to be iterated. To minimize the input write and output read cycles, required iteration number is taken as input and the final result is computed by iterating **F** for the necessary number of times. For each iteration, the **ADRS** value from the **F** operation needs to be updated, and this is handled internally by the control logic shown in Figure 3.
(8) **H_TREE**: This is a special extension of **H** which we design in our HASH_TILE, which computes one level of a Merkle tree. It computes the hash of two child nodes using the L_HASH and R_HASH in parallel, followed by hash for the parent node using the L_HASH and returns the hash for the parent node as a result. We compute the tree for two reasons: 1. to minimize the cycles. 2. to reduce the complexity of the main control logic of our design.

We note from the aforementioned list of opcodes, all operations except **H_TREE** can be performed in sets of two because we have L_HASH and R_HASH. We also note that depending on the selected opcode, the HASH_TILE operates in two modes, namely, streaming mode and regular mode. The regular mode is used for the opcode {**T_L**, **PRF**, **H**, **F**, **F_WOTS_PLUS_h**, **H_TREE**} where the input and output sizes are fixed. In the regular mode, the data inputs and outputs move through the BRAMs. The opcode {**H_MSG**, **PRF_MSG**} uses the streaming mode because one of the inputs required in these modes is the message input. The message is not of fixed length; hence, the HASH_TILE allows a variable-length message to be streamed through the FIFO. The output still moves through the BRAMs. The variable length messages add complexity of maintaining the order of all required inputs in opcode {**H_MSG**, **PRF_MSG**}, and this is managed by the Pre/Post processing block, which is part of the Control logic (shown in Figure 1).

The reason we choose to store and retrieve the data through the BRAMs is that in most of the modes presented in Table 3, the output hash is fed as input in the next iteration of the hash computation. Consequently, to save the data load out and load in cycles, we write the hash to the appropriate memory location. For a given parameter set (shown in Table 1), and specific HASH_TILE

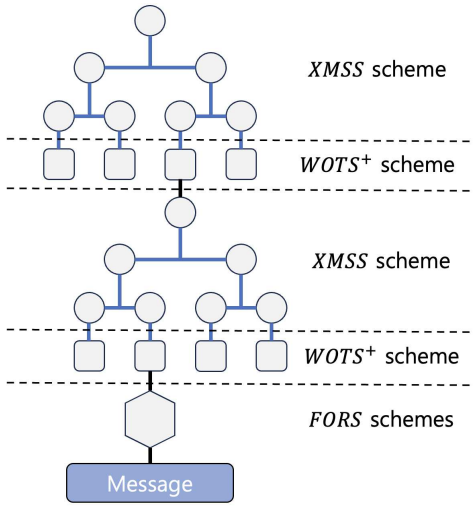


Fig. 4. Dataflow diagram of SLH-DSA.

opcode the number of blocks read from or written to the memory and the address locations remain constant. Therefore the design remains constant-time, avoiding any chance of leakages due to non-constant time behavior. The HASH_TILE also has optional output ports that are coming out of the L_HASH and R_HASH directly without passing through BRAMs. These ports are not shown in the Figure 3 because we do not use them in our SPHINCSLET design described in §5.

Our HASH_TILE module has modular structure, which allows us to easily swap between SHAKE256, SHA256, and SHA512 hash functions. We highlight that this parameterizability facilitates us to have a single same top-level control logic (described in §5.2) for different parameter sets and different underlying hash functions. As discussed in §5, the HASH_TILE is integral to building a purely hardware-based SPHINCSLET module. Its BRAM-based interface enables the HASH_TILE to function effectively as a co-processor as well.

5.2 FSM (Control Logic) Design

Algorithm 4 shows the algorithm for SLH-DSA signature generation, its operational flow could be visualized with Figure 4, and our hardware block design in shown in Figure 1. The SLH-DSA signing construction mainly consists of three main schemes: FORS, XMSS, and WOTS$^+$ (described in §3). In our hardware design, the flow of the operations is handled by the control logic to effectively handle our HASH_TILE design. Firstly, the Pre/Post Processing FSM loads the inputs message (M), public key seed (**PF**.$seed$) and PRF key (**SK**.prf) in to the HASH_TILE and initiates the **PRF** mode from the HASH_TILE to generate the pseudorandom bits (R). A copy of the R remains in the internal memory inside the HASH_TILE. Along with that, the Pre/Post Processing FSM loads **PF**.$seed$, and top root node (**PF**.$root$) in the HASH_TILE and **H_MSG** mode is set to generate the message digest.

Following that, the FORS FSM is initiated. The FORS FSM takes the inputs message digest, secret seed (**SK**.$seed$), **PF**.$seed$ and context information in the form of an address (**ADRS**) and generates a FORS public key using multiple FORS trees as described in §3.4. The root node of the FORS tree (which is the FORS public key) is then loaded into the WOTS$^+$ scheme. The WOTS$^+$ scheme uses a **SK**.$seed$, **PK**.$seed$, and **ADRS** to generate a secret key ($sk_0, sk_1, ..., sk_{len-1}$). Then, the WOTS$^+$ secret key undergoes '$(w-1)$ $iterations$' of **F** mode of hashing to generate a WOTS$^+$ public key (as shown in Figure 5). In WOTS$^+$ computation, we generate the leaf nodes of XMSS, as shown in

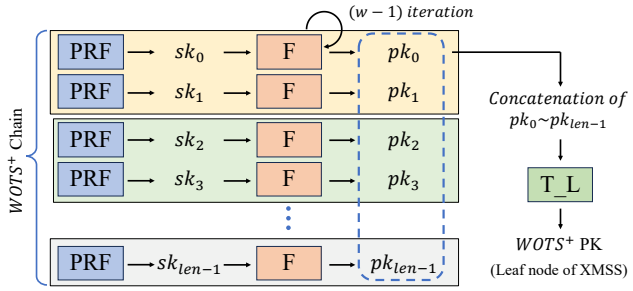


Fig. 5. Dataflow diagram of WOTS$^+$ PK generation. $(sk_0, sk_1..., sk_{len-1})$ and $(pk_0, pk_1..., pk_{len-1})$ refers to WOTS$^+$ secret and public keys respectively.

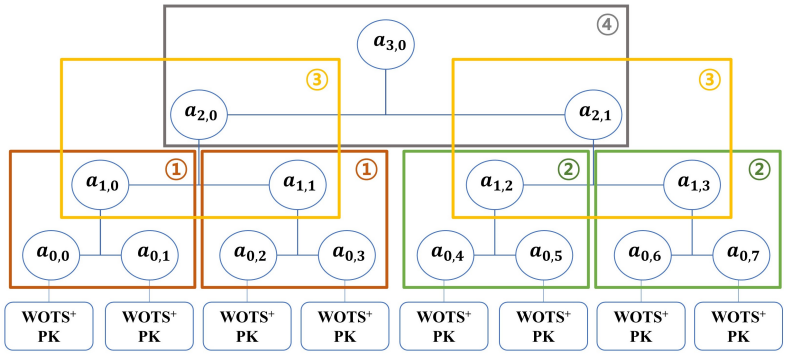


Fig. 6. Dataflow diagram of Merkle tree, circled numbers represent order of the data being processed.

Figure 5, we employ a chain-parallel approach using HASH_TILE. The FSM design in [3] focused on chain operation in WOTS$^+$ uses one-by-one approach to achieve the high throughput with one big pipelined hash module, while our FSM design uses the HASH_TILE, which contains two small hash modules, and utilizes the parallelism of the WOTS$^+$ chain. Once all operations for each chain are complete, we obtain a public key consisting of *len* blocks, each of *n* bytes (given in Table 1), which is compressed to form a single leaf node using opcode **T_L**. In our hardware design, all the WOTS$^+$ related operations are handled by the WOTS$^+$ FSM (shown in Figure 1). The WOTS$^+$ public key is then loaded into the XMSS scheme. Similar to the FORS, the XMSS scheme also uses Merkle trees to generate the signature. However, the parameter sizes for the Merkle trees in FORS and XMSS differ. The WOTS$^+$ and XMSS schemes are repeated iteratively for d layers (given in Table 1) for signature generation or signature verification computations.

The computational flow of signature generation and verification of SLH-DSA is very similar as specified in [15]. Hence, we design our SPHINCSLET hardware design control logic (shown in Figure 1) in a way that it can handle both operations. The key differences between the signature generation and signature verification are as follows: 1) In signature generation, for the XMSS and FORS computation, we need to compute all the nodes of the Merkle tree to generate the root node. Whereas in signature verification, we only need to compute one hash per layer in a Merkle tree, based on the provided neighboring node information. 2) In the signature generation process, we perform the WOTS$^+$ signing operation with m or w iterations of hash computations at each chain operation using **F_WOTS_PLUS_h** (described in §5.1). Whereas in signature verification, WOTS$^+$ signature is verified by iterating over the chain operation for only $(w - m - 1)$ times, which is also handled by the **F_WOTS _PLUS_h**.

Merkle Trees. As described in §3.2 and §3.4, FORS and XMSS schemes include computing Merkle trees, which have similar dataflow. Notable differences between that schemes include the method of leaf node generation and the size of the Merkle trees. Nonetheless, the underlying operation remains the same (i.e., hash computation on each node) Therefore, in our control logic, we designed a parameterizable Merkle tree FSM that takes the tree height and the number of leaf nodes as inputs, making our Merkle tree FSM adaptable for both FORS and XMSS operations. Additionally, this also contributes to the reduction in the area utilization. To support the leaf node generation for XMSS and FORS, our WOTS$^+$ processor uses different hash types described in the §5.1 (**PRF**, **F_WOTS_PLUS_h**, and **T_L** for XMSS and **PRF** and **F** for FORS computation). As mentioned at the beginning of this §5.2, the Merkle tree is parameterized to work for both XMSS and FORS operations since they share the same structure. Figure 6 shows an example dataflow of the Merkle tree generation for XMSS. The dataflow for FORS (shown in Figure 4) is the same except the inputs of the leaf nodes are part of FORS secret keys, instead of WOTS+ public key.

In the Merkle tree operations, two child nodes are taken as an input for one hash operation. As HASH_TILE can operate two hashes in parallel, a total of four child nodes are received as input. The child nodes are stacked to the internal BRAM by checking the index of the parent node's **ADRS**. The result is stored back in the internal BRAM to prepare for the subsequent operations i.e., the hash computation of the next node in the Merkle tree. This minimizes input loading and output retrieval cycles.

To efficiently utilize the HASH_TILE design along with internal BRAM, Merkle tree computations are performed in units of four nodes. The process begins with the computation of leaf nodes using WOTS+ public key generation (illustrated in Figure 5), starting from the leftmost node. Once four leaf nodes are computed, an **H** hash operation (denoted as operation 1 in Figure 6) is performed to generate two parent nodes at the upper level of the Merkle tree. Repeating this process once more results in four nodes at the upper level, which are then processed using another H hash operation (operation 3 in Figure 6). If additional leaf nodes remain, the computation returns to the lower level to process the next set of four nodes. In cases where no leaf nodes remain (as seen in Figure 6), a final H hash operation (operation 4) is performed at the upper level. This approach enables efficient BRAM utilization by limiting storage to only four nodes per level. The sequence of computations is outlined in Figure 6, demonstrating an optimized traversal strategy. Additionally, this method ensures that both the L_SHAKE and R_SHAKE components of the HASH_TILE (illustrated in Figure 3) are consistently utilized in parallel, maximizing computational efficiency.

6 EVALUATION

In this section, we evaluate different hardware modules described in §5 and provide a detailed comparison with the existing work. For the evaluation and comparison with the related work, we use the following metrics: for the area, we use look-up-tables (LUTs), Flipflops (FF), and Block RAM (BRAM), and for timing, we use clock cycles, maximum clock frequency (F_{max}), and time. For our work, the area results are extracted from post-implementation area reports of the AMD Vivado synthesis tool, and for timing, the clock cycles are computed using the simulations, and F_{max} is computed from post-implementation timing reports. We verify the functional correctness of our implementation with the reference implementation [4].

6.1 Evaluation of the HASH_TILE

Table 3 shows the performance results of our HASH_TILE module considering the parameter sizes for SLH-DSA-128s parameter set (values given in Table 1). The area results shown for SHA256+SHA512 based HASH_TILE sum of two individual SHA256 and SHA512 based HASH_TILE.

Table 3. Area utilization of the SHAKE-based and SHA-2-based HASH_TILEs and time taken for performing each opcode operation by the HASH_TILE targeting the AMD Artix 7 xc7a200t-3 FPGA. Please note that except for opcode **H_TREE**, for all other opcodes the time shown is to perform a set of two operations.

HASH_TILE variant	Area			F_{max}	t (μs)							
	LUT	FF	BRAM	(MHz)	**T_L**	**PRF**	**H**	**H_MSG**	**F**	**PRF_MSG**	**F_WOTS_PLUS_h**	**H_TREE**
Parameter Set = SLH-DSA 128s												
SHAKE256	9,962	6,038	4.0	150	1.01	0.32	0.32	0.32	0.32	0.32	3.19	0.64
SHA256	4,264	4,123	4.0	110	6.22	0.82	0.82	0.82	0.82	0.82	5.53	1.64
Parameter Set = SLH-DSA 192s												
SHA256+SHA512	12,466	11,881	8.0	110	8.59	0.84	1.13	1.13	1.44	1.13	10.47	2.25

Table 4. Area utilization of the FSMs and internal memory in the SHAKE-based and SHA-2-based designs targeting the AMD Artix 7 xc7a200t-3 FPGA.

Par.	Area			F_{max}
	LUT	FF	BRAM	(MHz)
FSMs and Internal Memory in SHAKE256-based design				
128f	1,602	1,421	4	150
128s	1,638	1,470	4	150
192f	1,702	1,642	4	150
192s	2,038	1,695	4	150
256f	1,754	1,732	4	150
256s	2,096	1,758	4	150
FSMs and Internal Memory in SHA256-based design				
128f	1,743	1,930	4	110
128s	1,859	1,908	4	110
FSMs and Internal Memory in SHA256+SHA512-based design				
192f	1,896	3,156	4	100
192s	2,011	3,195	4	100
256f	1,866	3,267	4	100
256s	1,923	3,290	4	100

Since this configuration is only used in {192s, 192f, 256s, 256f}, we present an example considering the SHA-2 based SLH-DSA 192s configuration.

6.2 Evaluation of FSM Design

We note that, as shown in Table 4, FSM hardware design remains similar for both SHAKE256-based design and SHA-2-based design in various parameter sets of SLH-DSA. The only change lies in the input format of the **ADRS**. In SHAKE256-based design, the **ADRS** size is 32 bytes, which includes layer address, type, padding, and index. As shown in Table 4, the utilization of internal BRAMs remains consistent across all parameter sets of SLH-DSA. This is because the maximum BRAM usage is dictated by the height of the Merkle tree in FORS, which is the tallest Merkle tree in SLH-DSA, and is given by $\log t$ (refer to Table 1). Consequently, even for the parameter set with the highest Merkle tree, the total BRAM utilization does not exceed 4 BRAM units. Additionally, the s variant of the parameter sets features a taller Merkle tree compared to the f variant, leading to differences in structural depth while maintaining the same BRAM constraints.

6.3 Evaluation of SPHINCSLET

Our SHAKE256-based and SHA-2-based SPHINCSLET hardware design results are shown in Table 5 and Table 6 respectively. These results are post-implementation results targeted to AMD Artix 7 (xc7a200t-3) FPGA and are generated using AMD Vivado 2024.2. Additionally, for SHA-2-based SPHINCSLET design, we also tabulate results targeting AMD Ultrascale+ (xczu3eg-2) FPGA in Table 6 for fair comparison with the related work [7].

Table 5. Comparison of our SHAKE256-based SPHINCSLET hardware implementation with other SPHINCS+ hardware implementations that use SHAKE256 as the underlying hash function.

Par.	Area				CC_{sign}	t_{sign}	CC_{verify}	t_{verify}
	LUT	FF	BRAM	DSP	(cycles.)	(ms)	(cycles.)	(ms)
SLH-DSA Our Work, Artix 7 (`xc7a200t-3`), F_{max} = 150 MHz								
128f	10,197	7,357	8	0	2,895,583	19.30	286,064	1.91
128s	10,255	7,410	8	0	54,450,583	363.00	95,744	0.64
192f	10,333	7,590	8	0	4,665,879	31.11	414,368	2.76
192s	10,644	7,643	8	0	94,740,920	631.61	137,816	0.92
256f	10,416	7,672	8	0	9,265,012	61.77	421,856	2.81
256s	10,802	7,698	8	0	83,552,886	557.02	204,104	1.36
Full Hardware Design SPHINCS$^+$ [3], Artix 7 (`xc7a100t-3`), F_{max} = 250 & 500 MHz								
128f	47,991	72,505	11.5	1	-	1.01	-	0.16
128s	48,231	72,514	11.5	0	-	12.40	-	0.07
192f	48,398	73,476	17	1	-	1.17	-	0.19
192s	48,725	72,514	17	0	-	21.40	-	0.10
256f	51,009	74,539	22.5	1	-	2.52	-	0.21
256s	51,130	74,576	22.5	1	-	19.30	-	0.14
RISC-V-based SLH-DSA [17], Artix 7 (`xc7a100t-3`), F_{max} = 100 MHz								
128f	8,605	3,745	32	0	4,903,978	49.04	440,636	4.41
128s	8,605	3,745	32	0	102,346,701	1,023.47	179,603	1.80
192f	8,605	3,745	32	0	10,596,236	105.96	711,431	7.11
192s	8,605	3,745	32	0	263,100,826	2,631.01	289,825	2.90
256f	8,605	3,745	32	0	23,660,226	236.60	857,059	8.57
256s	8,605	3,745	32	0	296,265,468	2,962.65	469,973	4.70
RISC-V-based SPHINCS$^+$ [13], Ultrascale+ (`xczu9eg-2`), F_{max} = 150 MHz								
128f	22,488	10,601	N/A	N/A	42,598,000	283.99	2,458,000	16.39
128s	22,488	10,601	N/A	N/A	837,742,000	5,584.95	852,000	5.68
192f	22,488	10,601	N/A	N/A	72,134,000	480.89	3,725,000	24.83
192s	22,488	10,601	N/A	N/A	1,538,599,000	10,257.33	1,301,000	8.67
256f	22,488	10,601	N/A	N/A	179,501,000	1,196.67	4,804,000	32.03
256s	22,488	10,601	N/A	N/A	1,666,663,000	11,111.09	2,369,000	15.79

Par. = Parameter Set

6.3.1 Performance Comparison with Full Hardware Design. In Table 5, we present the results for all possible variants of our SHAKE256-based SPHINCSLET designs, comparing them to the most recent and relevant full hardware implementation of the SPHINCS+ signature scheme at various security levels [3], which also utilizes SHAKE256 as the hash function in its construction. While the design in [3] takes less time to compute both signature generation (t_{sign}) and signature verification (t_{verify}) than our hardware design, we note that all of our designs take significantly less area. We acknowledge that the SHAKE256 design presented in [3] is a full-width implementation that is heavily pipelined, providing an inherent advantage with several independent hash computations in SPHINCS+. However, this advantage costs 47K-51K LUTs and 72K-74K flip-flops, making the design unsuitable for lightweight applications. Hence, to achieve a lower area design in our approach, we avoid pipelining in our SHAKE256 (rather we opt for round-based iterative architecture). Our `SPHINCSLET` hardware design utilizes under 10.8K LUTs, 7.7K flip-flops, and 8 BRAMs, which on average results in 4.7× lower LUT consumption, 9.7× lower flip-flops, and lesser BRAMs utilized when compared to [3].

Additionally, in Table 6, we tabulate the results for all potential variants of our SHA-2-based SPHINCSLET designs compared to the most recent non-standard compliant SHA-2-based SPHINCS+ across various security levels [7]. As described in §3.5, the impact of standard compliance is more pronounced in SHA-2-based SPHINCS designs because the standard suggests utilizing both SHA256 and SHA512 in parameter sets {192f, 192s, 256f, and 256s}, while for parameter sets {128f and 128s}, only SHA256 is employed. As noted in §5.1, we implement the design from Figure 1a for single `HASH_TILE` configurations such as parameter sets {128f and 128s}. We also implement the design from Figure 1b for dual `HASH_TILE`s, applicable to parameter sets {192f, 192s, 256f, and 256s}. In addition to presenting synthesized results for AMD Artix 7 (`xc7a200t`), we include results for our

Table 6. Performance comparison of SLH-DSA and SPHINCS+ hardware implementations that use hash functions other than SHAKE256. (Note that all designs have 0 DSP.)

Par.	Hash (SHA-2)	Security Level	Area LUT	Area FF	Area BRAM	F_{max} (MHz)	CC_{sign} (Cycles.)	t_{sign} (ms)	CC_{verify} (Cycles.)	t_{verify} (ms)
SLH-DSA Our Work, Artix 7 (xc7a200t-3)										
128f	SHA256	1	6,787	6,083	8	110	5,030,577	45.73	536,769	4.88
128s	SHA256	1	6,767	6,054	8	110	93,893,056	853.57	179,919	1.64
192f	SHA256 & SHA512	3	15,436	15,144	12	100	9,005,754	90.06	1,357,674	13.58
192s	SHA256 & SHA512	3	15,615	15,183	12	100	174,503,502	1,745.04	447,901	4.48
256f	SHA256 & SHA512	5	15,513	15,206	12	100	17,341,452	173.41	1,399,533	14.00
256s	SHA256 & SHA512	5	15,618	15,225	12	100	159,138,446	1,591.38	673,948	6.74
SLH-DSA Our Work, Ultrascale+ (xczu3eg-2)										
128f	SHA256	1	7,345	6,062	8	250	5,030,577	20.12	536,769	2.15
128s	SHA256	1	7,476	6,120	8	250	93,893,056	375.57	179,919	0.72
192f	SHA256 & SHA512	3	16,900	15,140	12	230	9,005,754	39.16	1,357,674	5.90
192s	SHA256 & SHA512	3	17,061	15,178	12	230	174,503,502	758.71	447,901	1.95
256f	SHA256 & SHA512	5	16,762	15,190	12	210	17,341,452	82.58	1,399,533	6.66
256s	SHA256 & SHA512	5	16,879	15,217	12	210	159,138,446	757.80	673,948	3.21
Full Hardware Design SPHINCS$^+$ [7], Ultrascale+ (xczu3eg)										
128f	SHA256	1	6,127	4,933	0.5	156	-	64.34	-	2.51
128s	SHA256	1	6,072	4,733	0.5	154	-	985.16	-	1.12
256f	SHA256	5	8,591	6,339	0.5	152	-	199.20	-	4.55
256s	SHA256	5	8,612	6,366	0.5	149	-	1,734.91	-	2.46
RISC-V-based SLH-DSA [17], Artix 7 (xc7a200t-2)										
128f	SHA256	1	5,486	3,675	32	100	9,127,150	91.27	691,186	6.91
128s	SHA256	1	5,486	3,675	32	100	190,085,952	1,900.86	268,445	2.68
192f	SHA256 & SHA512	3	8,965	6,823	32	100	23,726,217	237.26	1,290,921	12.91
192s	SHA256 & SHA512	3	8,965	6,823	32	100	626,858,593	6,268.59	641,048	6.41
256f	SHA256 & SHA512	5	8,965	6,823	32	100	50,240,516	502.41	1,419,466	14.19
256s	SHA256 & SHA512	5	8,965	6,823	32	100	696,201,400	6,962.01	894,078	8.94
RISC-V-based SPHINCS$^+$ [21], Ultrascale+ (xczu3eg)										
256s	SHA256	5	4,870	3,350	N/A	100	-	-	4,950,000	49.50
256s	SHA256	5	5,850	4,050	N/A	100	-	-	3,260,000	32.60

Par. = Parameter Set

design targeting AMD Ultrascale+ (xczu3eg) FPGA to ensure a fair comparison with the design introduced in [7]. From Table 6, we observe that our SHA-2-based SPHINCSLET design is 2×-3× faster compared to the existing non-standard compliant SPHINCS+ design in [7], with a smaller area for parameter sets {128f and 128s}. In terms of memory resources, the design in [7] exhibits a compact size of the BRAM, which transfers results to the host after each operation. This transfer back to the host incurs significant time overhead. Conversely, our design utilizes more BRAM than [7], storing all data on the FPGA and eliminating the time overhead associated with data transfers. For parameter sets {192f, 192s, 256f, and 256s}, the area increase is inherent due to the inclusion of both SHA256 and SHA512 HASH_TILE shown in Figure 1b.

6.3.2 Performance Comparison with HW-SW Codesign. In Table 5, we compare our design with two RISC-V-based designs that use SHAKE as a hash function targeting the AMD Artix 7 and Ultrascale+. Out of these two RISC-V-based works, our hardware design significantly outperforms [13] in both area and timing aspects. [17] is a more recent and optimized design that is fully standard compliant, similar to ours; therefore, we evaluate our work against [17]. The Keccak round implementation from [17] also presents a full-width implementation (taking one round per clock cycle), which is similar to our SHAKE256 module described in §5.1. We note that our overall SLH-DSA hardware implementation has a slightly larger area footprint than the implementation presented in [17], this is due to usage of two SHAKE256 modules inside the HASH_TILE. However, our BRAM utilization is significantly smaller. The design proposed in [17] relies on BRAM to store all intermediate data and program files due to its use of a RISC-V processor. In contrast, our design requires significantly less BRAM storage by minimizing the amount of intermediate data retained during computations. For instance, in Merkle tree operations, only the data for four nodes per level is necessary for

computation. As a result, we store only these four leaf nodes, reducing overall intermediate data storage and, consequently, lowering BRAM utilization. In terms of timing comparison, our design is 2.5× to 5.3× faster for signature generation and 2.3× to 3.4× faster for signature verification across various parameter sets compared to [17]. One reason for this speed-up is that our design utilizes two SHAKE256 modules in parallel, maximizing all possible parallel hash computations. Additionally, as stated in [17], the RISC-V module connects to the Keccak round accelerator via a 32-bit interconnect. It also incurs some penalty cycles from the RISC-V core for instruction fetch, both of which contribute to the overall timing.

In Table 6, we compare our design against the SHA2-based SPHINCS+ designs. Even in this case, our design uses more area footprint than that reported in [17, 21]. This is due to the use of two hash modules within the `HASH_TILE` module. In case of high security level parameter sets i.e., {192s, 192f, 256s, 256f}, we use two `HASH_TILE` one consisting two SHA256 modules and the other consisting of two SHA512 modules. Our BRAM utilization is lower compared to [17], and [21] did not report the number of BRAM used, as they employ the register interface to communicate between RISC-V and the hash hardware module. Our SHA-2 based `SPHINCSLET` design significantly outperforms all other existing modules in terms of both signature generation and signature verification times, making it the fastest SHA-2 based SPHINCS+ design.

7 CONCLUSION

In this work, we introduced SPHINCSLET, a fully standard-compliant and area-efficient hardware implementation of the SLH-DSA (formerly SPHINCS+) post-quantum digital signature scheme. Our design strikes an optimal balance between hardware area and performance, addressing the limitations of existing implementations that either sacrifice area efficiency for speed or rely on coprocessor-based architectures with significant performance overhead. Through our SHAKE256-based implementation, we achieved a 4.7× reduction in area compared to high-speed designs while maintaining a 2.5× to 5× faster signing time than the most efficient coprocessor-based alternatives. Additionally, our SHA-2-based implementation demonstrated a 2× to 4× speedup in signature generation while maintaining a compact area footprint of 6K to 15K LUTs, making it the fastest SHA-2-based SLH-DSA hardware implementation to date. By requiring fewer than 10.8K LUTs on an AMD Artix-7 FPGA for a SHAKE256-based SPHINCS+, our SPHINCSLET design proves to be a highly practical solution for resource-constrained devices, ensuring an efficient transition to post-quantum cryptography.

ACKNOWLEDGMENTS

This work was partially supported by the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2025, the Inter-University Semiconductor Research Center (ISRC), the Institute of Information & communications Technology Planning & Evaluation (IITP) under the artificial intelligence semiconductor support program to nurture the best talents (IITP-2023-RS-2023-00256081) grant funded by the Korea government(MSIT), the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2023-00277326), the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.RS-2023-00277060, Development of open edge AI SoC hardware and software platform, 0.01) and Korea Evaluation Institute of Industrial Technology(KEIT) grant funded by the Korea government(MOTIE) (No.RS-2023-00277060, Development of open edge AI SoC hardware and software platform, 0.01). The EDA tool was supported by the IC Design Education Center(IDEC), Korea. This work was also supported through a grant from TII and grant 2332406 from US National Science Foundation.

REFERENCES

[1] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, et al. 2022. Status report on the third round of the NIST post-quantum cryptography standardization process. US Department of Commerce, NIST (2022).

[2] Dorian Amiet, Andreas Curiger, and Paul Zbinden. 2018. FPGA-based accelerator for post-quantum signature scheme SPHINCS-256. IACR Transactions on Cryptographic Hardware and Embedded Systems (2018), 18–39.

[3] Dorian Amiet, Lukas Leuenberger, Andreas Curiger, and Paul Zbinden. 2020. Fpga-based sphincs+ implementations: Mind the glitch. In 2020 23rd Euromicro Conference on Digital System Design (DSD). IEEE, 229–237.

[4] Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, and Peter Schwabe. [n. d.]. SPHINCS+-software. https://github.com/sphincs/sphincsplus

[5] Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, and Peter Schwabe. 2017. SPHINCS+ - Submission to the NIST post-quantum cryptography project. https://sphincs.org

[6] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zookor Wilcox-O'Hearn. 2015. SPHINCS: Practical Stateless Hash-Based Signatures. In Advances in Cryptology – EUROCRYPT 2015. Springer Berlin Heidelberg, Berlin, Heidelberg, 368–397.

[7] Quentin Berthet, Andres Upegui, Laurent Gantel, Alexandre Duc, and Giulia Traverso. 2021. An area-efficient SPHINCS+ post-quantum signature coprocessor. In 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, 180–187.

[8] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2013. Keccak. In Annual international conference on the theory and applications of cryptographic techniques. Springer, 313–314.

[9] Ricardo Chaves, Georgi Kuzmanov, Leonel Sousa, and Stamatis Vassiliadis. 2006. Improving SHA-2 hardware implementations. In Cryptographic Hardware and Embedded Systems-CHES 2006: 8th International Workshop, Yokohama, Japan, October 10-13, 2006. Proceedings 8. Springer, 298–310.

[10] Information Technology Laboratory Computer Security Division. 2017. Post-quantum cryptography standardization - post-quantum cryptography: CSRC. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization

[11] Sanjay Deshpande, Chuanqi Xu, Mamuri Nawan, Kashif Nawaz, and Jakub Szefer. 2023. Fast and Efficient Hardware Implementation of HQC. In Proceedings of the Selected Areas in Cryptography (SAC).

[12] Craig Gidney and Martin Ekerå. 2021. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. Quantum 5 (April 2021), 433. https://doi.org/10.22331/q-2021-04-15-433

[13] Patrick Karl, Jonas Schupp, and Georg Sigl. 2024. The Impact of Hash Primitives and Communication Overhead for Hardware-Accelerated SPHINCS+. In International Workshop on Constructive Side-Channel Analysis and Secure Design. Springer, 221–239.

[14] Manoj Kumar and Pratap Pattnaik. 2020. Post quantum cryptography (PQC)-An overview. In 2020 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 1–9.

[15] National Institute of Standards and Technology. 2023. Stateless Hash-Based Digital Signature Standard. Technical Report Federal Information Processing Standards Publication (FIPS) NIST FIPS 205 ipd. U.S. Department of Commerce, Washington, D.C. https://doi.org/10.6028/NIST.FIPS.205

[16] Fazal Raheman. 2022. The future of cybersecurity in the age of quantum computers. Future Internet 14, 11 (2022), 335.

[17] Markku-Juhani O. Saarinen. 2024. Accelerating SLH-DSA by Two Orders of Magnitude with a Single Hash Unit. In Advances in Cryptology – CRYPTO 2024, Leonid Reyzin and Douglas Stebila (Eds.). Springer Nature Switzerland, Cham, 276–304.

[18] Naoyuki Shinohara and Shiho Moriai. 2019. Trends in Post-Quantum Cryptography: Cryptosystems for the Quantum Computing Era. the magazine of New Breeze, PP (2019), 9–11.

[19] Joachim Strömbergson. 2021. SHA512. https://github.com/secworks/sha512.

[20] Joachim Strömbergson. 2023. SHA256. https://github.com/secworks/sha256.

[21] Alexander Wagner, Felix Oberhansl, and Marc Schink. 2022. To be, or not to be stateful: post-quantum secure boot using hash-based signatures. In Proceedings of the 2022 Workshop on Attacks and Solutions in Hardware Security. 85–94.