FPGA-based Accelerator for Post-Quantum Signature Scheme SPHINCS-256

Dorian Amiet[1], Andreas Curiger[2] and Paul Zbinden[1]

[1] IMES Institut für Mikroelektronik und Embedded Systems
HSR Hochschule für Technik, 8640 Rapperswil, Switzerland
damiet@hsr.ch, pzbinden@hsr.ch, https://imes.hsr.ch
[2] Securosys SA, 8005 Zürich, Switzerland
curiger@securosys.ch, https://www.securosys.ch/

Abstract. In recent years, a substantial amount of research has been conducted and progress made in the area of quantum computers. Small functional prototypes have already been reported. If they scale as expected, they will eventually be able to break current public-key cryptosystems. The goal of post-quantum cryptography is to develop cryptographic systems that are secure against attacks originating from both quantum and classical computers. Frequently referred post-quantum signature schemes are based on the security of hash functions. A promising candidate in this group is SPHINCS-256. This paper presents the first FPGA-based hardware accelerator for SPHINCS-256. It can be implemented on an entry-level FPGA, occupying roughly 19,000 LUTs, 38,000 FFs and 36 BRAMs. On a Kintex-7 Xilinx FPGA, signing takes 1.53 milliseconds, and verification needs only 65 microseconds. Area and throughput of the accelerator are in a range that outperform today's widely used RSA signature scheme. The performance can even keep up with ECDSA accelerators. Hence, SPHINCS-256 is a hot candidate to replace RSA and ECDSA in a post-quantum world.

Keywords: FPGA architecture · digital signature · post-quantum cryptography · SPHINCS-256 · computer science

1 Introduction

At the time of writing, it is still not clear whether large-scale quantum computers will ever be feasible. Recent initial successes in the field suggest that they might become available within the next decade or so [Mos15]. Today's signature algorithms in use (RSA [RSA78] and ECDSA [Nat09]) would be broken by quantum computers running Shor's algorithm [Sho97]. It will therefore be essential that all digital signing systems based on RSA or ECDSA be replaced by a system which will resist a quantum computer attack before large scale quantum computers become available. Several approaches which enable quantum computer safe signing can be found in the literature. Most of these so-called post-quantum signature schemes can be assigned to one of the following four groups [BL17]:

1. Lattice-based signature schemes: They are often favored as replacement candidates, because operations are usually faster compared to ECDSA and RSA. In addition, key and signature sizes are only moderately larger. However, their security level is not at all clear, especially with respect to quantum computers.

2. Multivariate quadratic signature schemes: They exhibit a similar behavior as lattice based schemes. Computation is fast, signature and key sizes are short, but for security analysis they are even more complicated than lattice-based schemes.

3. Code-based signature schemes: They provide better prerequisites for a security analysis than both of the schemes above, but involved key sizes are high (in the order of megabytes), which makes their implementation in some cases infeasible.

4. Hash-based signature schemes: They seem to be the most promising candidates as key sizes and processing speed are reasonable and the signature schemes appear relatively simple. They are better suited for a security analysis, are less susceptible to implementation errors, and generally receive better user confidence. However, most of them are state-based. This means that a private key is attached to a limited number of states, and each state can be used only once to generate just one signature. A signer hence needs to trace, which states have already been used, because using a state twice would break the signature scheme.

Recently, Bernstein at al. presented a stateless hash-based signature scheme called SPHINCS [BHH+15]. The stateless classification is guided by two main ideas: First, a private key includes a high number of states. During signing, a state is chosen at random, such that picking one state more than once is very unlikely. Second, a mechanism is in place ensuring that even if a state is picked more than once, the signature scheme will still be safe. For the signature scheme, the SPHINCS authors suggest a parameter set where most data words are aligned to 256 bits. A detailed security analysis of this SPHINCS-256 scheme exhibits a security level of 256 bits on classical and 128 bits on quantum computers. The price for being stateless and providing 256 bits of security is being paid with a longer signature size of 41 kB and an increased computation time for signing. In addition to a reference software implementation, the SPHINCS authors provide an optimized implementation generating “hundreds of signatures per second” on a quad-core vector processor.

Hundreds of signatures per second may be an acceptable throughput for a software-based signing system. Even when classical signature schemes such as RSA and ECDSA are in use, software-based signing systems usually attain throughputs within this order of magnitude. To reach higher performance it is common practice to use hardware accelerators or specialized co-processors respectively, whenever a high-performance signing system is built.

This paper presents a hardware accelerator for SPHINCS-256. To the authors’ knowledge, this is the first publication of a SPHINCS hardware architecture. In the next section, the idea of hash-based signature schemes is summarized and the SPHINCS-256 scheme is recalled. In sections 3 and 4, the architecture is described in detail and its performance is analyzed. To evaluate the performance, a comparison with other signature schemes is made in section 5. Throughout the whole paper, the following notations are used:

$\boxplus$	unsigned addition modulo 2^{32}
$\oplus$	bitwise XOR
$\lll k$	rotation by k bits to the left on a 32-bit data word
$\leftarrow$	value assignment
$a\|\|b$	concatenation, a contains the less significant bits

2 Hash-Based Signatures

Using cryptographic hash functions (digest = f(seed)) for signing is not at all a new idea. As early as 1979, Lamport [Lam79] introduced a signature scheme which is today referred as one-time signature (OTS). To sign one bit, the private key consists of a pair of random data seeds of size n. Function f is evaluated twice, both private seeds are used as input once. Both resulting outputs, two digests of size n, form the public key. Depending on the bit value which is signed (0 or 1), the signature consists of the first or second random data seed. To sign a message digest of size n, this procedure is expanded to n bits. Hence, both keys expand to $2n^2$ bits, and the signature expands to n^2 bits.

The security of this approach is based on the hardness of inverting the hash function f. Many hash functions are reported in the literature and most of them have not been broken yet by any (quantum-) attack. Together with the fact that hash functions are well understood and widely analyzed, Lamport's OTS approach could have been a hot candidate for post-quantum signing. However, Lamport's OTS turns out to be unpractical, because a key pair can only be used once.

2.1 Merkle Tree

To address this issue, Merkle published a method which allows many signatures related to a single public key [Mer89]. 2^H OTS key pairs are merged in a balanced binary tree of depth H. An example with $H = 2$ is shown in Figure 1. The OTS private seeds are represented as X_i, and the corresponding OTS public keys as Y_i. The leaves $N_{i,0}$ are calculated as $f(Y_i)$. Tree nodes are the digest of its concatenated child nodes (a pair of neighbor leaves or inner nodes). The root node $N_{0,H}$ is the public key which can be used for 2^H signatures. A signature itself contains a leaf index $0 \leq i < 2^H$, the OTS signature σ_i including its OTS public key Y_i and the authentication path. The authentication path includes all nodes in the tree which a verifier needs in order to validate the OTS public key Y_i (a verifier calculates the root on his own with Y_i and the authentication path). To verify the signature, σ_i is first verified using Y_i. Then leaf $N_{i,0}$ is generated using Y_i. Using Y_i and the authentication path, the verifier will then calculate the root. If the resulting digest is identical to the signer's public key, the signature is valid.

In the example from Figure 1 with $H = 2$, a signature for $i = 2$ consists of σ_2 (the signature part from the OTS scheme), Y_2 (the OTS public key), and $N_{3,0}$ and $N_{0,1}$ (the authentication path).

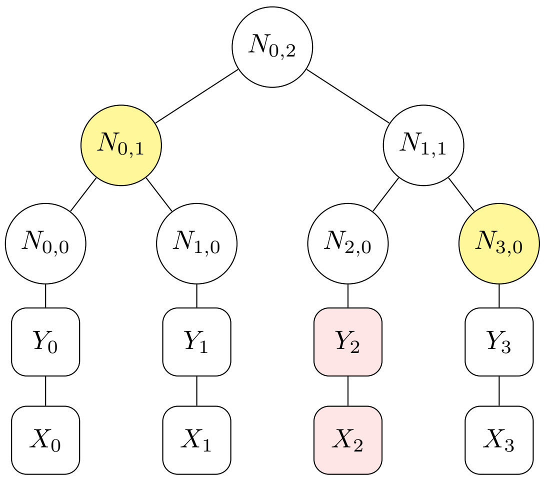

Figure 1: This Merkle tree compresses four OTS public keys into $N_{0,2}$. A signature for $i = 2$ consists of σ_2, Y_2, $N_{3,0}$, and $N_{0,1}$.

Several other improvements have been made since Lamport's OTS. These include OTS compression (WOTS) [Hül13], the use of pseudo-random numbers (PRN) for private seeds and tree chaining [BGD+06], and a few-time signature (HORS) [BS02]. The evolution and details of all these improvements are well described in [BDS09].

2.2 SPHINCS-256

All previously-mentioned OTS improvements have been combined in the SPHINCS-256 scheme, which makes it quite comprehensive. Signing starts with selecting a leaf address between 0 and $2^{60} - 1$. Based on this address, one of 2^{60} possible few-time signature (HORS) key pairs is selected. The HORS public key is compressed within a tree structure of 256 bits. Relating to the tree structure, this part of the signature has been dubbed HORST. The selected HORST key pair is used to sign the message digest. The leaf address is then used to pick the corresponding WOTS key pair. Its private key is used to sign the HORST public key. The WOTS public key is compressed to 256 bits within an unbalanced tree (L_tree). Together with compressed neighbor WOTS public keys as leaves, a tree of height 5 is calculated. Its root is treated the same way as the HORST public key before, i.e., it is signed with another WOTS key pair. The only difference is that the address shortens from 60 bits down to 55. This procedure is repeated until the leaf address reaches 0 bits. In other words, regardless of which HORST key pair is selected at the beginning, all authentication paths end up in the same final WOTS tree. The root of this last tree will finally be the public key of the SPHINCS scheme. This structure is shown in Figure 2. A more accurate description can be found in the original SPHINCS paper [BHH+15].

In the view of someone implementing a SPHINCS accelerator, the algorithms in use are the most interesting parts. As PRN source, the hash functions BLAKE-256 and BLAKE-512 [AHMP10] are being used. PRN words are expanded by the stream cipher ChaCha12 [Ber08], which will be rewritten in Algorithms 1 to 3. For tree calculations and WOTS operations, the same permutation as in ChaCha12 (Algorithms 1 and 2) will be used. This function will be referred to as π_{ChaCha}. The previously-described SPHINCS-256 signing will be supplied in Algorithm 6, which calls the tree generation functions from Algorithm 4 and the authentication path generation Algorithm 5.

2.3 Possible Improvements to SPHINCS

Aumasson and Endignoux suggest several techniques to reduce the signature size [AE17] (from 41 down to 20-30 kbytes). A modification to HORS would allow to remove two of the cascaded WOTS trees. Moreover, an algorithm called Octopus would avoid redundancies in the HORS-tree authentication paths. Another three WOTS signatures could be removed by significantly increasing the top Merkle tree in size (height 20 instead of 5). The latter is payed for, however, with processing time, whenever the top tree is processed. Aumasson and Endignoux argue that this would not be a penalty. Because the identical tree is used during every signing operation, the tree data could be cashed (stored) instead of recalculated every time. However, such an FPGA implementation would need 444 BRAMs (in a Xilinx FPGA) to hold the proposed two Mbytes of data. This would result in 12 times more BRAM resources compared to the SPHINCS-256 implementation introduced in this work.

Further room for improvement would lie in the choice of the hash functions. In our opinion, the selection of BLAKE-256 and BLAKE-512 remains insufficiently defended in the SPHINCS paper, whereas ChaCha12 and π_{ChaCha} are well reasoned with their performance. Independent of the selection, it would be favorable for hardware acceleration to use a single hash function for all parts within the algorithm. A good choice to strive high performance might be Gimli [BKL+17], a recently proposed hash function which exhibits high performance on different platforms. If security analysis and user confidence is more important than performance, SHA-3 might be a good choice as well. A nice analysis on SPHINCS performance depending on the choice of the hash function is given in [Köl17].

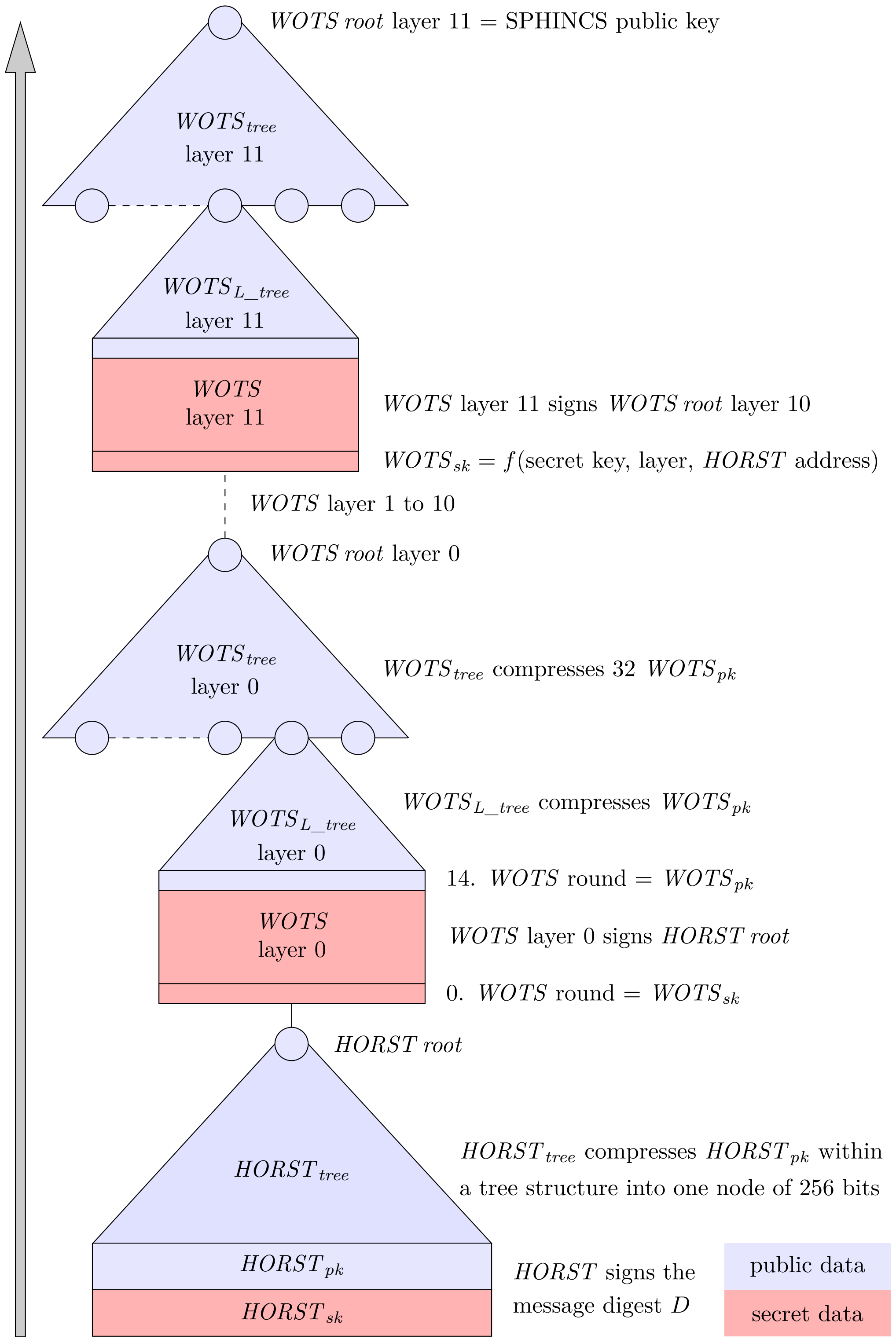
$WOTS$ $root$ layer 11 = SPHINCS public key
$WOTS_{tree}$ layer 11
$WOTS_{L\_tree}$ layer 11
$WOTS$ layer 11
$WOTS$ layer 11 signs $WOTS$ $root$ layer 10
$WOTS_{sk} = f$(secret key, layer, $HORST$ address)
$WOTS$ layer 1 to 10
$WOTS$ $root$ layer 0
$WOTS_{tree}$ layer 0
$WOTS_{tree}$ compresses 32 $WOTS_{pk}$
$WOTS_{L\_tree}$ layer 0
$WOTS_{L\_tree}$ compresses $WOTS_{pk}$
14. $WOTS$ round = $WOTS_{pk}$
$WOTS$ layer 0
$WOTS$ layer 0 signs $HORST$ $root$
0. $WOTS$ round = $WOTS_{sk}$
$HORST$ $root$
$HORST_{tree}$
$HORST_{tree}$ compresses $HORST_{pk}$ within a tree structure into one node of 256 bits
$HORST_{pk}$
$HORST_{sk}$
$HORST$ signs the message digest D
public data
secret data

Figure 2: Tree structures are the main concept in the SPHINCS-256 scheme. Twelve *WOTS* signatures and several trees compress 2^{60} different *HORST* public keys into the top root node.

Algorithm 1 ChaCha quarterround

```
Input: $a, b, c, d$: four 32-bit unsigned integers
Output: $a, b, c, d$: four 32-bit unsigned integers
1: function QUARTERROUND($a, b, c, d$)
2:     $a \leftarrow a \boxplus b$
3:     $d \leftarrow (d \oplus a) \lll 16$
4:     $c \leftarrow c \boxplus d$
5:     $b \leftarrow (b \oplus c) \lll 12$
6:     $a \leftarrow a \boxplus b$
7:     $d \leftarrow (d \oplus a) \lll 8$
8:     $c \leftarrow c \boxplus d$
9:     $b \leftarrow (b \oplus c) \lll 7$
10:    return $a, b, c, d$
```

Algorithm 2 ChaCha permutation π_{ChaCha}

```
Input: $X = x[0, ...15]$: 512-bit vector, cut in sixteen 32-bit integers
Output: $X = x[0, ...15]$: 512-bit vector, permuted input
1: function $\pi_{ChaCha}(X)$
2:     for $i \leftarrow 0$ to 5 do                                  ▷ 12 rounds
3:         QUARTERROUND($x[0], x[4], x[8], x[12]$)
4:         QUARTERROUND($x[1], x[5], x[9], x[13]$)
5:         QUARTERROUND($x[2], x[6], x[10], x[14]$)
6:         QUARTERROUND($x[3], x[7], x[11], x[15]$)      ▷ lines 3 - 6: horizontal round
7:         QUARTERROUND($x[0], x[5], x[10], x[15]$)
8:         QUARTERROUND($x[1], x[6], x[11], x[12]$)
9:         QUARTERROUND($x[2], x[7], x[8], x[13]$)
10:        QUARTERROUND($x[3], x[4], x[9], x[14]$)       ▷ lines 7 - 10: diagonal round
11:    return $X$
```

Algorithm 3 The ChaCha12 stream cipher

```
Input: $M = m[0...15]$: 512-bit vector, cut in sixteen 32-bit integers:
    $m[0...3] = $ 128-bit nonce (constant)
    $m[4...11] = $ 256-bit key
    $m[12...13]$: 64-bit counter value (with iv0...iv1)
    $m[14...15]$ iv2...iv3
    $l$: unsigned integer, number of iterations, $0 < l < 2^{64}$
Output: $M$: Same as input, but counter is incremented
    $V = v_{0...l-1}[0...15]$: $l$ pseudo-random data words, each 512 bits wide
1: function CHACHA12($M, l$)
2:     for $i \leftarrow 0$ to $l - 1$ do
3:         $v_i \leftarrow \pi_{ChaCha}(M)$
4:         for $j \leftarrow 0$ to 15 do
5:             $v_i[j] \leftarrow v_i[j] \boxplus m[j]$
6:         $m[12...13] \leftarrow m[12...13] + 1$                    ▷ 64-bit counter
7:     return $V, M$
```

Algorithm 4 Tree generation

```
Input: $h$: depth of tree
    $leaves$: $\leq 2^h$ data words, tree level $0 = N[i, 0]$
    $Q_{0...2h-1}$: level masks
    $C_{256}$: 256-bit constant, ASCII representation of "expand 32-byte to 64-byte state!"
Output: $tree$: all tree nodes $N[i, j]$ including $root$ node $N[0, h]$
 1: function TREE($leaves, Q, h$)
 2:     $N[0...2^h - 1, 0] \leftarrow leaves$
 3:     for $j \leftarrow 1$ to $h$ do
 4:         for $i \leftarrow 0$ to $2^{h-j} - 1$ do
 5:             if exist($N[2i, j - 1]$) then
 6:                 if exist($N[2i + 1, j - 1]$) then              ▷ both child nodes exists
 7:                     $(tmp_0||tmp_1) \leftarrow \pi_{ChaCha}(N[2i, j - 1] \oplus Q_{2(j-1)}||C_{256})$
 8:                     $tmp_0 \leftarrow tmp_0 \oplus N[2i + 1, j - 1] \oplus Q_{2(j-1)+1}$
 9:                     $N[i, j] \leftarrow \pi_{ChaCha}(tmp_0||tmp_1)$
10:                 else                                            ▷ only left child node exists
11:                     $N[i, j] \leftarrow N[2i, j - 1]$
12:             else                                                ▷ no child nodes exist
13:                 $N[i, j] \leftarrow$ not exist
14:     return $tree \leftarrow N[0...2^h - 1, 0...h]$
```

Algorithm 5 Get authentication path from given tree and node index

```
Input: $h$: depth of tree
    $tree$: tree including all nodes $N[i, j]$ $(0 \geq j \geq h,\ 0 \geq i \geq 2^{h-j} - 1)$
    $idx$: Node index
Output: $auth_{0...h-1}$, authentication path for node $N[idx, 0]$
 1: function AUTH($tree, h, idx$)
 2:     for $j \leftarrow 0$ to $h - 1$ do
 3:         if $idx\ mod\ 2 == 0$ then          ▷ $idx$ even, current node is on left side
 4:             $auth_j \leftarrow N[idx + 1, j]$          ▷ take sibling on right side
 5:         else                                 ▷ $idx$ odd, current node is on right side
 6:             $auth_j \leftarrow N[idx - 1, j]$          ▷ take sibling on left side
 7:         $idx \leftarrow idx/2$
 8:     return $auth_{0...h-1}$
```

Algorithm 6 The SPHINCS-256 signing algorithm

```
Input:
    $SK_1$: 256-bit random seed 1, part of private key
    $SK_2$: 256-bit random seed 2, part of private key
    $Q_{0...31}$: 32 random mask words, each 256-bit wide, part of both private and public key
    $M$: message to sign (arbitrary length)
    $root$: 256-bit top node of tree chain, part of public key
    $C_{128}, C_{256}$: 128-and 256-bit constant value, respectively
Output:
    $\sum_{0...1281}$: signature: size($\sum_0$) = 64 bits, size($\sum_{1...1281}$) = 1280·256 bits
    $root$: 256-bit data word (public key consists of $Q_{0...31}$ and $root$)
function SIGN($SK, M$)
    $leaf||i_{tree}||nonce_4||R_2||nonce_{128} \leftarrow$ BLAKE-512($SK_2||M$)
    $\sum_0 \leftarrow (leaf||i_{tree}||0_4)$                    ▷ add (5+55)-bit leaf address to signature
    $\sum_1 \leftarrow R_2$                                      ▷ add $R_2$ to signature
    $D_0||D_1||...||D_{31} \leftarrow$BLAKE-512($R_2||M||root||Q_{0...31}$)      ▷ 32 indexes of 16 bits
    $layer \leftarrow 12$                                        ▷ 4-bit tree layer, part of leaf address
    $\mathcal{S} \leftarrow$BLAKE-256($SK_1||layer||i_{tree}||leaf$)        ▷ seed for HORST private key
    $HORST_{sk}[0...2^{16}-1] \leftarrow$ CHACHA12($C_{128}||\mathcal{S}||0_{128}$, $2^{15}$)     ▷ $2^{16}$ private seeds
    for $i \leftarrow 0$ to $2^{16}-1$ do                        ▷ $2^{16}$ HORST elements
        $HORST_{pk}[i] \leftarrow \pi_{ChaCha}(HORST_{sk}[i]||C_{256})$
    $HORST_{tree} \leftarrow$ TREE($HORST_{pk}, Q_{0...31}, 16$)
    $\sum_{2..66} \leftarrow HORST_{tree}[0...63, 10]$           ▷ all nodes of HORST tree layer 10
    for $j \leftarrow 0$ to 31 do
        $\sum_{67+11j} \leftarrow HORST_{sk}[D_j]$
        $\sum_{67+11j...76+11j} \leftarrow$ AUTH($HORST_{tree}, 10, D_j$)
    $root \leftarrow HORST_{tree}[0, 16]$                        ▷ Tree root node
    for $layer \leftarrow 0$ to 11 do                            ▷ 12 WOTS layers
        $\mathcal{S} \leftarrow$BLAKE-256($SK_1||layer||i_{tree}||leaf$)    ▷ WOTS private key seeds
        $WOTS_\sigma[0...66] \leftarrow$CHACHA12($C_{128}||\mathcal{S}||0_{128}$, 34)     ▷ 67 private seeds
        $d_0||d_1||...||d_{63} \leftarrow root$      ▷ cut $root$ into 64 unsigned integer, each 4 bits wide
        $d_{64}||d_{65}||d_{66} \leftarrow 960 - d_0 - d_1 - ... - d_{63}$        ▷ $960 = 15 \cdot 64$
        for $i \leftarrow 0$ to 66 do                            ▷ 67 WOTS elements
            for $j \leftarrow 0$ to $d_i - 1$ do                 ▷ skip if $d_i == 0$
                $WOTS_\sigma[i] \leftarrow \pi_{ChaCha}(Q_j \oplus WOTS_\sigma[i]||C_{256})$
        $\sum_{418+72 \cdot layer...484+72 \cdot layer} \leftarrow WOTS_\sigma[0...66]$
        for $k \leftarrow 0$ to 31 do                            ▷ 32 WOTS key pairs per tree
            $\mathcal{S} \leftarrow$BLAKE-256($SK_1||layer||i_{tree}||k$)
            $WOTS_{pk}[0...66] \leftarrow$CHACHA12($C_{128}||\mathcal{S}||0_{128}$, 34)     ▷ 67 private seeds
            for $i \leftarrow 0$ to 66 do                        ▷ 67 WOTS elements
                for $j \leftarrow 0$ to 14 do
                    $WOTS_{pk}[i] \leftarrow \pi_{ChaCha}(Q_j \oplus WOTS_{pk}[i]||C_{256})$
            $WOTS_{L\_tree} \leftarrow$ TREE($WOTS_{pk}[0...66], Q_{0...13}, 7$)
            $L\_root[k] \leftarrow WOTS_{L\_tree}[0, 7]$
        $WOTS_{tree} \leftarrow$ TREE($L\_root[0...31], Q_{14...23}, 5$)
        $\sum_{485+72 \cdot layer...489+72 \cdot layer} \leftarrow$ AUTH($WOTS_{tree}, 5, leaf$)
        $root \leftarrow WOTS_{tree}[0, 5]$
        $leaf||i_{tree} \leftarrow i_{tree}||0_5$
    return $\sum$, $root$
```

3 Architectural Considerations

Based on the SPHINCS-256 scheme, a possible hardware architecture will be designed. In this section, some important design decisions which affect the whole implementation will be explained. The first and most important decision is, however, the overall design target: we will focus on fast signing with moderate use of FPGA resources. The throughput should be at least as high as the optimized software implementation, and the accelerator should fit in a reasonably sized FPGA.

3.1 Hash Evaluations per Signature

The most costly parts of the signing process include hash computations of BLAKE-512, BLAKE-256, ChaCha12, and π_{ChaCha}. How many times each function is evaluated is summarized in Table 1.

Table 1: Hash evaluations per SPHINCS-256 signature: Overhead is the WOTS signature of the previous tree root without authentication path calculations.

Function	Signing					Verification
Part	Start	HORST	12·WOTS	Overhead	**Total**	Total
BLAKE-256	0	1	384	12	**397**	0
ChaCha12	0	32,768	13,056	408	**46,232**	0
π_{ChaCha}	0	193,410	437,352	≈9,000	**≈640,000**	≈9,000
BLAKE-512	2	0	0	0	**2**	1

By far, the most frequently called hash function is π_{ChaCha}, followed by ChaCha12. Both operations consist basically of 12 ChaCha rounds, which is actually a π_{ChaCha} permutation. ChaCha12 needs only an extra 64-bit counter and an unsigned addition at the end. However, to reach the goal of fast signing in the SPHINCS accelerator, π_{ChaCha} calculation will have to be either very fast or heavily parallelized.

Compared to π_{ChaCha}, both BLAKE functions are called infrequently. BLAKE-512 is needed twice for signing and once for verification, each at the beginning. These functions can be called by the main processor before the SPHINCS accelerator is used, or even while the accelerator is processing a previous signature. Therefore, BLAKE-512 calculations will not be part of the SPHINCS co-processor. This will also alleviate communications between the accelerator and the main processor, because the accelerator input data length may be kept constant. BLAKE-256 calls are more critical. Their inputs and outputs must be kept private. In addition, BLAKE-256 is called more often than BLAKE-512. However, a small iterative BLAKE-256 instantiation will suffice for the SPHINCS accelerator.

3.2 One Intermediate Result per Clock Cycle

As explained before, fast or heavily parallelized π_{ChaCha} calculations are mandatory for a high-performance SPHINCS-256 implementation. Both techniques are covered by a fully unrolled loop pipeline. Such a block is efficient in hardware and generates one result per clock cycle. However, a highly pipelined block is only meaningful if its pipeline can be filled. This requirement will define additional design parameters. Firstly, ChaCha12 inputs and outputs are 512 bits wide. While the full 512 bits output will be used and, therefore, must be buffered in registers or RAM blocks, only 256-bit data words are needed at the input. The remaining 256 input bits are constant or a counter value. This fixes data paths that read from block RAMs to 256 bits and some write paths even to 512 bits. Secondly, many π_{ChaCha} calculations must be data independent to enable pipeline filling.

3.2.1 Data Dependencies

To find out how intermediate calculations depend on each other, let us consider an overgrown design were an arbitrary number of operations can be processed in parallel. Algorithm 6 might be executed as follows:

1. Signing starts with the BLAKE operation from line 2 $(leaf||i_{tree}||nonce(4)||R_2||nonce(128)) \leftarrow$ BLAKE-512$(SK_2||M)$.

2. When the leaf address is available, all other BLAKE evaluations can start in parallel. These are evaluations of the message digest D and all BLAKE-256 computations.

3. HORST: All private seeds can be calculated in parallel by using 2^{15} instances of ChaCha12 with initialization vectors 0 to $2^{15}-1$.
   WOTS: All private seeds of the 12 trees are evaluated using additional 13,056 instances of ChaCha12.

4. HORST: When private seeds are available, all public seeds will be calculated using 2^{16} π_{ChaCha} permutations in parallel.
   WOTS: 15 serial iterations of π_{ChaCha} on every one of the 26,112 private seeds are needed to generate the public key.

5. HORST: Two iterative π_{ChaCha} evaluations per tree level have to be applied. After 16 levels or 32 π_{ChaCha} iterations, the HORST root is calculated.
   WOTS: Two nested trees have to be calculated to get the WOTS root. The L-tree calculation with depth 7 is followed by the WOTS tree with depth 5. The parallelized delay is $24 \cdot t(\pi_{ChaCha})$.

6. Assuming that M is small and hence digest D is available in the meantime, all computations are finished. Based on D and the intermediate tree roots, parts of the generated data are copied to a signature array.

In conclusion, the minimal SPHINCS-256 signing delay due to true data dependencies is

$$t_{min} = t(\text{BLAKE-512}) + t(\text{BLAKE-256}) + t(\text{ChaCha12}) + 39 \cdot t(\pi_{ChaCha}). \qquad (1)$$

In contrast to the required 700,000 π_{ChaCha} operations, data dependencies are very low. Hence, the utilization rate of a deep π_{ChaCha} pipeline can theoretically exceed 99%.

3.3 Memory

An upper-bound approximation of memory usage can be obtained assuming that every intermediate result needs to be stored. The HORST tree inclusive private and public seeds needs 6 MB of storage. Each WOTS hypertree needs another 1.6 MB. This results in 25 MB of total data. While this is not a big deal for a desktop computer and may even fit in a processor cache memory, it means a big burden for a compact FPGA implementation. Our target device, a 7-series Xilinx FPGA, contains block RAMs (BRAMs) with 4.5 KB storage each. Therefore, saving everything would occupy thousands of BRAMs.

A practical lower bound is somewhere around the sum of key and signature size. Hülsing et al. demonstrated in [HRS15] that even less memory is required to generate a SPHINCS-256 signature. However, the 41 KB signature will fit into 10 BRAMs. If 12 BRAMs are being used, 256-bit wide data accesses can be implemented efficiently. A 256-bit wide memory consisting of 12 BRAMs contains 1,536 entries. This will be enough to store the signature, all masks, both key strings, the message digest D, and the leaf address. Therefore, a RAM block of this size will be perfect for data input and output. Apart from this I/O RAM, a cache RAM to store intermediate values will be needed. To enable appropriate parallelization, it should store a full WOTS tree ($32 \cdot 67$ L_tree leaves at 256 bits). Using 24 block RAMs, a memory with 512-bit write and 256-bit read access stores 3,072 256-bit data words. This will be sufficient to store 1/64 of the HORST tree.

3.4 Top-Level Architecture

Based on the previous considerations, the top-level architecture is defined. The main blocks are:

- *Control unit*: To generate internal instructions for signing and verification
- *BLAKE*: An iterative, low-speed block that evaluates BLAKE-256
- *ChaCha12*: An efficient pipeline calculating ChaCha12 and π_{ChaCha}
- *I/O RAM*: Keys and the signature storage, accessible from outside
- *Cache RAM*: 256-bit aligned intermediate results storage

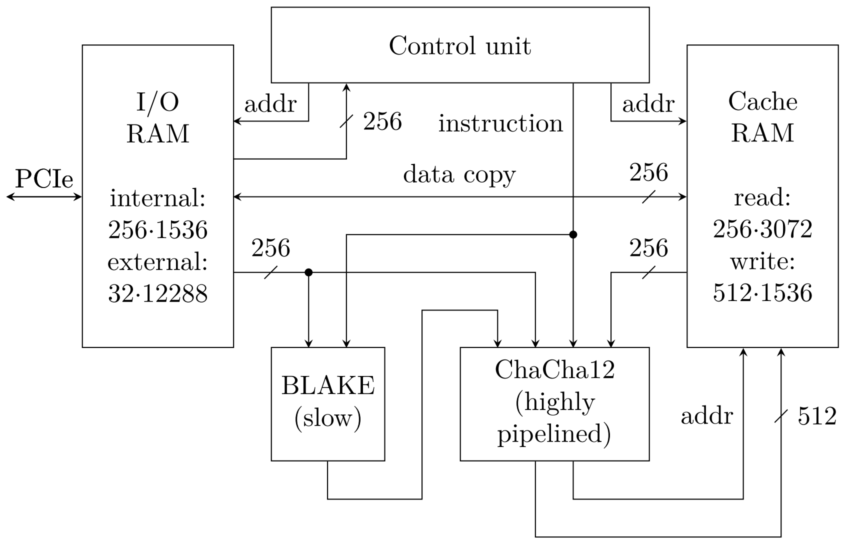

Figure 3: SPHINCS-256 co-processor architecture.

4 Implementation Details

In this section, the SPHINCS-256 co-processor is described in more detail. This includes a description of the control unit, the main processing π_{ChaCha} pipeline, implementation considerations about the BLAKE-256 block, performance results, and finally a side-channel analysis.

4.1 Control Unit

The control unit is the most intricate block in the design. Both signing and verification algorithms are stored in its state machine. In a black-box view, the control unit generates internal instructions based on message digest and leaf address. The used instruction set is summarized in Table 2. An instruction consists of the following parts:

1. Instruction code: A one-hot encoded automaton, indicating the instruction code

2. Valid flag: active-low if the FSM is idle or waiting for an intermediate value
3. The cache-RAM read address
4. The cache-RAM write address
5. The I/O RAM read or write address (depending on the instruction code)
6. The intermediate leaf address that is part of the BLAKE-256 data input

Table 2: Internal instruction set.

Instruction Name	RAM Addr Cache rd	RAM Addr Cache wr	I/O r/w	Input Data Read from (RAM, Register, Const C)	Size [bit]	Logic Operation	Result, Data Output Store at (RAM, Register)	Size [bit]
ChaCha12	-	✓	-	C_{128}||blake||count	512	ChaCha12	@wr, @wr+1	512
π_{ChaCha}	✓	✓	-	@rd||C_{256}	512	π_{ChaCha}	@wr	256
$\oplus\pi_{ChaCha}$	✓	✓	✓	(@rd$\oplus$@I/O)||C_{256}	512	π_{ChaCha}	@wr	256
$\oplus\pi_{ChaCha}1$	✓	✓	-	(@rd$\oplus$@I/O)||C_{256}	512	π_{ChaCha}	tmp (register)	512
$\oplus\pi_{ChaCha}2$	✓	✓	✓	tmp$\oplus$@rd$\oplus$@I/O	512	π_{ChaCha}	@wr	256
BLAKE-256	-	-	✓	leaf address||@I/O	320	BLAKE-256	blake (register)	256
Move to I/O	✓	-	✓	@rd	256	-	@I/O	256
Move to Cache	-	✓	✓	@I/O	256	-	@rd	256
Move to Ctrl	-	-	✓	@I/O	256	-	control unit	256
Is Equal	✓	-	✓	@rd, @I/O	512	==	control unit	1

Internally, the control unit is basically a nested finite-state machine (FSM) as shown in Figure 4. A master FSM controls several slave FSMs that generate the respective instructions. A slave FSM, for example, includes the tree-generation Algorithm 4. When the corresponding slave FSM is done, an appropriate control signal will inform the master FSM which in turn will start the next slave FSM.

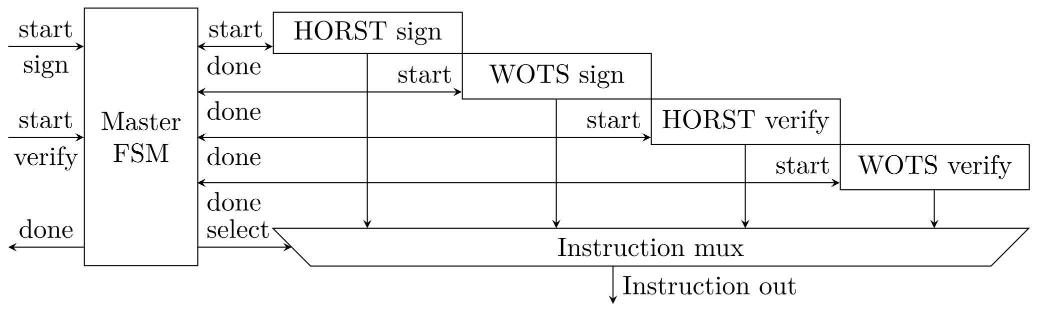

Figure 4: A Master FSM controls several slave FSM to reduce the number of states in one FSM.

The nested FSM architecture has some disadvantages though. It tends to take up more silicon area than a direct single FSM, and the sequence of states is less clear in the VHDL code. However, as fewer states are contained in an individual FSM, path lengths are shorter, which permit higher clock speeds.

Another design could have chosen to store all the instructions in several block RAMs. However, since instructions depend on D (the leaf address) and intermediate results (root nodes), a sizable logic block to calculate RAM addresses would have been required anyway. Therefore, great area savings would not have been anticipated.

4.2 ChaCha12 Pipeline

Before we dive into the description of the deep π_{ChaCha} pipeline, this section will cover the wrapper around it. The ChaCha12 unit is involved in six different internal instructions. A wrapper around the π_{ChaCha} part handles the instruction type. The wrapper input signals are: the instruction code, a 256-bit data word from cache RAM, a 256-bit data word (mask) from I/O RAM, the output of BLAKE, and the RAM write address for the result. Whenever a valid ChaCha12 instruction is received, the BLAKE output and the actual ChaCha12 counter value are routed to the π_{ChaCha} pipeline. In addition, the counter is incremented to be ready for the next ChaCha12 instruction. A valid BLAKE-256 instruction resets the counter such that a new ChaCha12 stream will restart at zero. Besides input routing, both input data and write address are forwarded to a shift register. The input data reaches the end of the shift register simultaneously with the result from the π_{ChaCha} permutation. Following the ChaCha12 algorithm, inputs and outputs are added: word by word, 32-bit-aligned and without carry propagation between words. After the final addition, all 512 data bits and the write address are sent to cache RAM.

If one of the $\oplus$ requesting π_{ChaCha} instructions is received, inputs from cache RAM and I/O RAM are immediately xor-ed. The result is expanded with the constant (C_{256} = the ASCII representation of “expand 32-byte to 64-byte state!”) and forwarded to the π_{ChaCha} pipeline. After execution, the lower 256 bits are written to cache RAM. One exception to this procedure occurs in tree calculations when instructions $\oplus\pi_{ChaCha}1$ and $\oplus\pi_{ChaCha}2$ are executed. The control unit ensures that these instructions are always issued side by side. In other words, whenever the $\oplus\pi_{ChaCha}1$ instruction is issued, a $\oplus\pi_{ChaCha}2$ follows during the next clock cycle. The $\oplus\pi_{ChaCha}1$ is forwarded directly to the pipeline. Data associated with $\oplus\pi_{ChaCha}2$ will be forwarded to the shift register. When $\oplus\pi_{ChaCha}1$ has finished, the result is not forwarded to RAM, but directly xor-ed with the $\oplus\pi_{ChaCha}2$ data and then again processed in the π_{ChaCha} pipeline. After this second loop, the resulting lower 256 bits will be sent to cache RAM.

As already mentioned in section 3, the π_{ChaCha} pipeline is fully unrolled such that one result per clock can be evaluated. Regarding the π_{ChaCha} Algorithm 2, lines 3 - 6 and 7 - 10 can be calculated in parallel, which will reduce processing delay. Note that parallel instantiation does not influence area usage in this case, because in a fully unrolled pipeline, a serial approach would also need four quarterround blocks per ChaCha round. Besides enabling high throughput, full unrolling has the advantage that there is no need for a multiplexer in the π_{ChaCha} pipeline. Hence, all connections are hard-wired between quarterround blocks. This fact is illustrated in Figure 5. Since there is no logic between the quarterround blocks, a well-designed pipeline in the quarterround block leads to a well-structured design in the whole π_{ChaCha} pipeline.

It is worth analyzing the quarterround Algorithm 1 in more detail. On lines 2 - 5 and 6 - 9, the same operations are executed. The only difference is the number of bits in the left-rotation operation. However, to design the pipeline, it is sufficient to consider lines 2 - 5 in the first step. In this short sequence, all variables (a, b, c, d) are assigned once. It seems straightforward to place one pipeline stage at the end of this sequence (between line 5 and 6) to get the following efficient structure: pipeline stage $\rightarrow$ small logic $\rightarrow$ pipeline stage. Unfortunately, lines 3 - 5 each depend on data from the previous line. Hence, the logical data path for b in lines 2 - 5 is

$$\boxplus \rightarrow \oplus \rightarrow \lll 16 \rightarrow \boxplus \rightarrow \oplus \rightarrow \lll 12. \tag{2}$$

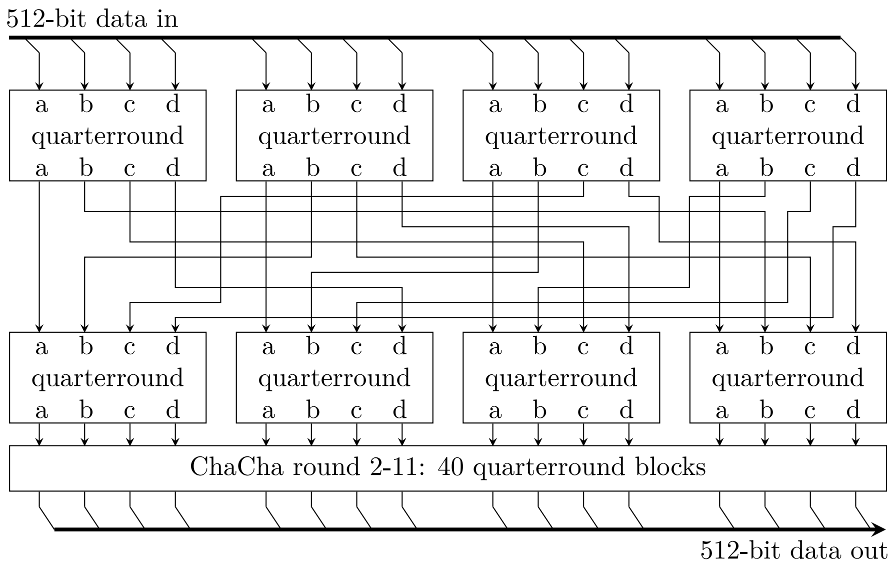

Figure 5: The ChaCha12 pipeline instantiates the quarterround block 48 times.

As rotations by n bits ($\lll$) are constant they come for free in hardware: Rotations are realized in the FPGA by inter-slice routing, which has to be done anyway. The remaining two operations ($\boxplus$ and $\oplus$) need logic primitives such as flip-flops (FFs), look-up tables (LUTs), and carry chains. Translated to Xilinx 7-series FPGA logic, the critical path of equation 2 leads to

$$\begin{gathered}\text{FF out} \rightsquigarrow \text{2-input LUT} \to \text{carry-chain (8x)} \rightsquigarrow \text{3-input LUT}\\ \to \text{carry-chain (4x)} \rightsquigarrow \text{2-input LUT} \to \text{FF in,}\end{gathered} \tag{3}$$

where $\to$ represents short connections within a slice and $\rightsquigarrow$ long connections between slices. This structure has high inherent delay, mainly because of the three inter-slice nets ($\rightsquigarrow$). Note that although carry chains include transfer through different slices, these nets are short.

It is easy to cut the path from equation 2 in almost half by adding an additional pipeline stage in the middle (between lines 3 and 4). The logic path shortens to

$$\boxplus \to \oplus \to \lll 16, \tag{4}$$

leaving the remaining critical path in the FPGA to

$$\text{FF out} \rightsquigarrow \text{2-input LUT} \to \text{carry-chain (8x)} \rightsquigarrow \text{2-input LUT} \to \text{FF in.} \tag{5}$$

Although this path is already quite short, it still has two connections between slices. By rearranging the stages, one of the inter-slice connections can be eliminated. Instead of having registers after lines 3, 5, 7 and 9, respectively, it is more efficient to place them after lines 2, 4, 6 and 8, respectively. The logic path for lines 3 and 4 results in

$$\oplus \to \lll 16 \to \boxplus. \tag{6}$$

This way, the rotation has again to be taken into account, because it is done in the routing ($\rightsquigarrow$), which should be eliminated. Due to associative properties of $\oplus$ and $\lll$, the operations can be exchanged

$$(d \oplus a) \lll 16 \equiv (d \lll 16) \oplus (a \lll 16). \tag{7}$$

The critical path finally gets down to

$$\text{FF out} \rightsquigarrow \text{3-input LUT} \to \text{carry-chain (8x)} \to \text{FF in.} \tag{8}$$

A quarterround block is composed of three constructions as described above and some additional input and output logic (in lines 2 and 9). Since several blocks are instantiated in series without any logic blocks in-between, lines 2 and 9 form together the same construct. The quarterround block diagram is shown in Figure 6 and includes some information about area usage.

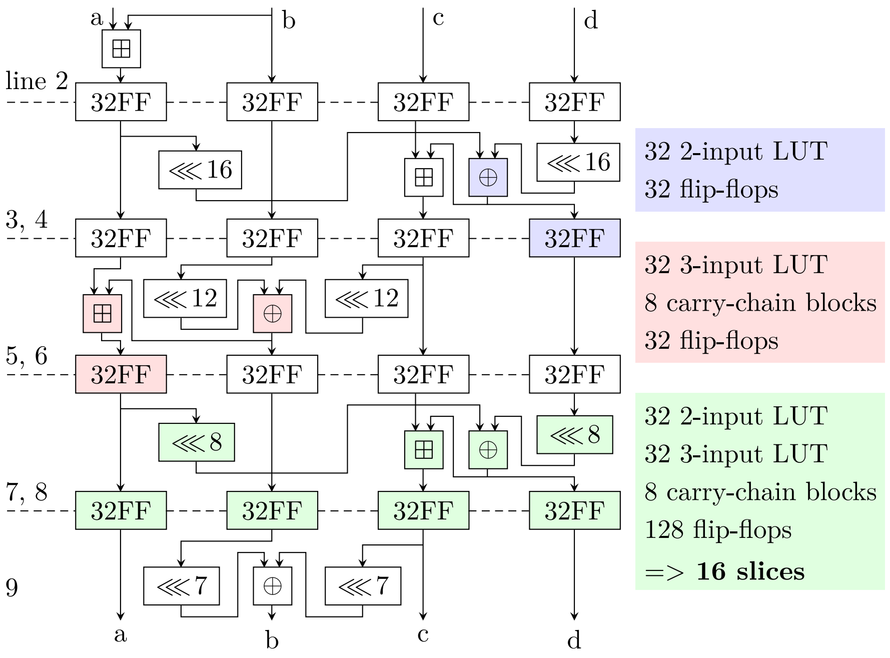

Figure 6: Quarterround, Algorithm 1

4.3 BLAKE

The BLAKE-256 function is called 397 times during signing, but is not used in verifying. In our implementation, the first call to BLAKE-256 stalls the whole SPHINCS accelerator. All other BLAKE-256 evaluations are executed in parallel to ChaCha computations. That is why its processing delay is negligible. The design goal for this unit is therefore minimizing area usage. Although the BLAKE-256 evaluation delay is not critical, the unit is designed to run at high clock speed. This allows the usage of the same clock as in the rest of the SPHINCS core, which gives the tools more freedom in the placing process (all flip-flops within a slice must be driven by the same clock).

A structural advantage of BLAKE-256 is that the input size is always 40 bytes (32 + 8 bytes for private seed and leaf address). Therefore, padding is constant and integrated in the initialization vector. In addition, the size is small enough, such that the compression function is called just once. The internal BLAKE unit structure consists of a small pipeline, some distributed RAM, and a control unit. The latter consists of a LUT ROM, which stores a small program and a program counter. It generates all control signals (RAM addresses, enable signals and multiplexer selects). Since all data paths are 32-bit aligned in the BLAKE unit and 256-bit aligned in the rest of the SPHINCS accelerator, an extra input multiplexer and a parallel accessible output shift register are needed.

All variables are filled into four distributed RAMs, each of which is 32 bits wide and stores four data words. These RAMs are directly connected to the arithmetic pipeline. Except for some extra round based constant look-up, the logic operations within the arithmetic pipeline have a similar shape as the ChaCha quarterround. Therefore, the same pipelining ideas as in the quarterround block can be applied.

4.4 Critical-Path Shortening

When we started with an initial working architecture of the SPHINCS accelerator, its performance already matched our goals (which we stated at the beginning of section 3). Implemented on a Kintex-7 device, the initial architecture occupied roughly 20,000 LUTs, signing took of 4.2 milliseconds, and the maximum clock speed was 175 MHz.

The signing latency was mainly caused by the 640,000 π_{ChaCha} evaluations. Hence, to reduce signing latency further, the ChaCha performance had to be increased. This could have been reached by using more parallelization, utilizing the π_{ChaCha} pipeline better, or speeding up the operating clock. If the π_{ChaCha} pipeline had been instantiated twice, also all other blocks would have had to be increased. Instancing the whole SPHINCS core several times might have ended up in a more efficient way to multiply the overall throughput than using multiple π_{ChaCha} pipeline instances within one core. The second strategy, utilizing the π_{ChaCha} pipeline better, did not exhibit much potential, because the pipeline was already filled quite well (see section 4.5). The third option, using a higher clock frequency, promised the biggest performance improvement.

Because a fully unrolled π_{ChaCha} pipeline produces one result per clock cycle, its throughput is linear to its operating clock frequency. To increase both clock frequency and throughput, the critical path has to be minimized. In Xilinx FPGAs, path delays are dominated by the number of cascaded logic elements (LUTs, carry chains, muxes etc.) and net delays. The particular path with the highest delay is defined as the critical path, the inverse of which defines the maximum clock frequency. To shorten this path, several steps have to be stepped through:

1. After placing and routing of the whole design by the Xilinx tools, all path delays are calculated and several slow paths are reported.
2. A detailed analysis of the critical path identifies why the path is slow (many cascaded logic elements or high net delays).
3. The path may be shortened by making changes to the RTL code (VHDL or Verilog).
    (a) If a path is slow because of cascaded logic, it usually depends on many other signals. Such a path may be shortened by adding pipeline stages (see section 4.2 for an example). If the cascade is provoked by a poor logic construct, simply rewriting the RTL code (e.g. replacing nested if-then-else by a case statement) may already shorten the path.
    (b) If a path is exhibiting a high net delay because of a FF output driving many endpoints (referred to as high fan-out), replication may be a solution. This means that the FF is "copied" by forwarding its input signal to several other FFs such that they all hold the same value at any time. A single FF then has to drive only a fraction of what the original FF had to drive.
    (c) If a path exhibit high net delays because of a far traversal within the FPGA, a solution may be inferring an additional FF (i.e., adding a pipeline stage). If placed somewhere in the middle, its net delay is almost cut in half.
4. Whenever an additional pipeline stage is inserted, the logic delay in clock-cycle count increases. This may require far-reaching modifications to the whole design to keep its correct execution.

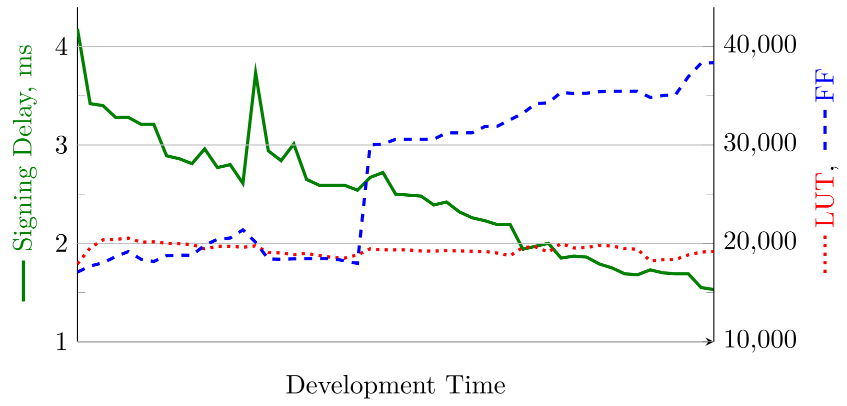

Figure 7: Signing latency has greatly been reduced without using significantly more LUTs.

By iteratively shortening all long paths, the critical path delay in the SPHINCS core decreased continuously. Consequently, the clock frequency could be increased, which simultaneously decreased the overall signing delay. As shown in Figure 7, the price for path shortening was mainly paid by higher FF usage. The moment to quit path shortening is defined by the FPGA structure. In 7-series Xilinx FPGAs, two FFs per LUT are placed in silicon. Path shortening was stopped when this rate was reached. Further path shortening would have become inefficient. A higher FF-to-LUT rate would have occupied many slices by FFs only. Because FFs are much smaller than LUTs this would have just been a waste of silicon area within the FPGA.

4.5 Performance Results

SPHINCS signing has been measured to take 804,500 clock cycles. Relating to some 687,000 π_{ChaCha} and ChaCha12 evaluations, the π_{ChaCha} pipeline capacity utilization is at 85%. The remaining 15% empty pipeline slots are mainly caused by data dependencies whenever tree top levels are evaluated. As stated in section 3.2.1, a higher pipeline usage would theoretically be possible, but it would be inefficient. If, e.g., calculations from the next tree starts before the present tree calculations are finished, the π_{ChaCha} pipeline would be utilized better, but the control unit would increase disproportionately in complexity and area usage.

Signature verification takes less than 35,000 clock cycles. The π_{ChaCha} pipeline has a usage of roughly 4% only. This is due to two reasons: First, much more data dependencies have to be taken into account. Second, since verifying is already 20 times faster than signing, less parallelization effort has been spent in order to keep the control unit smaller and simpler.

The SPHINCS accelerator was synthesized and implemented by Vivado 2017.2 on different target FPGAs. The smallest 7-series device the SPHINCS accelerator would fit in, is the $30 Artix XC7A35T-1. On this device, the clock runs at 280 MHz resulting in 2.9 ms signing time. On the high-end side of available FPGAs, the SPHINCS co-processor runs at 770 MHz on a Virtex UltraScale+ resulting in 1.04 ms signing time.

Note that all performance results presented in this paper are related to the Kintex-7 device XC7K325T-2. On this FPGA, the clock runs at 525 MHz and signing takes 1.53 ms. In Table 3, the hardware utilization of the SPHINCS core is shown, broken down to units. Over two thirds of occupied LUTs and FFs are utilized by the π_{ChaCha} pipeline. During the design phase, a balanced utilization has been striven for: 9.4% LUT, 9.4% FF, and 8.1% BRAM are being used in this device. This enables compact placing and therefore the utilization of high clock speeds, even if an FPGA is filled with multiple instances of the SPHINCS cores. In addition, three (corresponding to 0.4%) of the DSP blocks are assigned to the control unit to calculate RAM addresses.

Table 3: Area utilization of the SPHINCS core.[1]

Module	LUTs	FFs	BRAM	Slices
Control Unit	1,946	2,827	0	788
I/O RAM	1,138	1,546	12	747
Cache RAM	777	3,020	24	1,160
BLAKE-256	870	1,483	0	415
ChaCha12	13,305	27,060	0	4,807
SPHINCS Core	**19,067**	**38,132**	**36**	**7,306**

4.6 Side-Channel Analysis

Work on measuring side channels and attacking the accelerator is still in progress and detailed results cannot be presented at this stage. However, some theoretical considerations suggest that two parts in the signing algorithm tend to be susceptible to side-channel attacks. The first conspicuous event happens in Algorithm 6 in lines 10 to 13, when the HORST tree calculation is executed. Due to memory limitations, tree generation is split into 64 parts. Whenever a D_j indicates a leaf in the current part, the private HORST seed as well as the authentication path are copied to I/O RAM. Due to timing delays originating from the copy process, an attacker may collect information about $D_{0\ldots31}$. However, an attacker would find public information only here (i.e., the message digest).

A second obvious leak is found in lines 21 and 22. The for-loop "for $j \leftarrow 0$ to $d_i - 1$" is implemented in a loop that counts from 0 to 14. The following π_{ChaCha} operation is only executed if $j < d_i$ by using an enable signal. That way, the loop is executed in constant time, but a simple power analysis attack will leak information about $d_{0\ldots66}$. Since all tree roots have to be in public domain anyway, this information does not help an attacker either.

In the perspective of an attacker, the most interesting values are BLAKE-256 input SK_1, BLAKE-256 output $\mathcal{S}$, and all ChaCha12 outputs. Both involved functions BLAKE-256 and ChaCha-12 are processed in constant time. Further research will show whether these values can be extracted over some sort of side channels.

5 Comparison

To the authors' knowledge, this is the first reported hardware-based SPHINCS-256 implementation. Therefore, direct comparisons between our implementation and other published SPHINCS-256 hardware architectures are not possible. However, three other kinds of comparisons are presented in this section: In 5.1, we show that our accelerator signs an order of magnitude faster than the software-based implementation published in the original SPHINCS paper [BHH+15]. In 5.2, a comparison between our arithmetic blocks and hash implementations published in [ABO+14] show the high efficiency of our π_{ChaCha} pipeline. Finally, a comparison with other hardware-accelerated signature schemes shows in 5.3 that our SPHINCS co-processor outperforms today's widely used RSA signature scheme.

[1]Note that the numbers are post-implemented hierarchical utilization results from the Xilinx tool. They may be wrong because some logic parts that belong to a module may have been moved to another during optimization. In addition, slices can be shared by different modules (the sum from all modules is therefore bigger than the total number of used slices).

5.1 SPHINCS-256 Accelerator vs. Software Implementation

The original SPHINCS paper [BHH+15] presents performance results from an optimized vector processor implementation. The signing delay is reported as 51,636,372 cycles at a clock speed of 3.5 GHz. This corresponds to 14.75 ms delay. Our FPGA implementation features 1.53 ms signing delay, which is almost ten times faster. The acceleration can be explained as follows: The software-based implementation calculates eight π_{ChaCha} permutations in 420 clock cycles. On average, one π_{ChaCha} result is produced every 15 ns. The FPGA implementation produces one result per clock cycle, which is 1.9 ns. This gets already a speed-up factor of eight. A second effect is that the FPGA implementation does not have any overhead for pointer and index updates, because all such things are handled in the control unit and computed in parallel to ChaCha calculations. Signature verification with 414 μs vs. 65 μs delay is 6.3 times faster in the FPGA implementation. This comparison shows that the SPHINCS-256 scheme fits very well into hardware and especially FPGAs.

5.2 Hash Functions

The π_{ChaCha} pipeline represents the main core in the SPHINCS accelerator. The ChaCha12 unit takes 13,305 LUTs and 27,060 FFs, or roughly 4,800 slices. The throughput is 512 bits $\cdot$ 525 MHz = 269 Gbits/s. Divided by the 4,800 slices occupied, a performance indicator of 56 Mbits/(s·slice) results. The ChaCha architecture reported by At et al in [ABO+14] reaches a throughput of 422 Mbits/s at an area usage of 49 slices and 2 BRAMs. If BRAMs are ignored, the throughput per slice equals to 8.61 Mbits/(s·slice). Compared that way, the ChaCha pipeline in the SPHINCS accelerator is 6.5 times more efficient. Most of this comes from positive effects of pipeline unrolling. One may argue that this comparison is not fair, because the storage management is not included in the SPHINCS accelerator ChaCha12 unit. This is true, but even when the full SPHINCS co-processor, including control unit and BLAKE unit, is considered, an efficiency factor of 4.3 is still remaining. Putting it into Mbits/(s·BRAM usage), the SPHINCS accelerator performs 35 times better.

Regarding the BLAKE-256 unit, our implementation occupies 415 slices and runs with 525 MHz on the Kintex-7 device. At et al. report an architecture occupying 50 slices, 2 BRAMs and running at with 349 MHz on a Virtex-6 device. At's BLAKE-256 architecture needs 8.3 times less slices than our BLAKE-256 unit. Part of this discrepancy can be explained by not using BRAM (data and instructions are stored in LUTRAM) in our BLAKE-256 implementation. The SPHINCS co-processor already needs a notable amount of BRAMs, therefore BRAMs are not used in arithmetic units. Another notable amount of hardware is used to fit the 32-bit aligned BLAKE-256 unit into the 256-bit aligned SPHINCS accelerator.

5.3 SPHINCS-256 vs. Other Signing Schemes

A comparison to other signing algorithms is summarized in Table 4. This classifies rater the SPHINCS scheme than the actual hardware architecture. FPGA implementations can be built smaller at the price of more delay, or faster at the price of increased area usage, respectively. A reasonable figure of merit is the so-called area-time (AT) product. As a starting point, the SPHINCS accelerator is compared to the widely used RSA signature scheme. Even though the implementation of [SA14] is smaller than our SPHINCS core, the RSA signature scheme is clearly less efficient because of its signing delay. A signature scheme more frequently used in recent years is ECDSA. On a similar security level (only on classical computers), the ECDSA co-processor published in [ACZ16] and our SPHINCS accelerator are in the same performance range. Besides post-quantum security, SPHINCS-

256 has the advantage that verifying is only a fraction of the signing cost, while verification in ECDSA is at least as slow as signing.

Table 4: Our SPHINCS accelerator is more efficient than today's widely used RSA.

Ref	Scheme	Security		FPGA	Area	f	t	AT
		Classic	PQ		LUT/FF/DSP/BRAM	MHz	ms	s·LUT
this	SPHINCS-256	256	128	K7	19,067/38,132/3/36	525	1.53	29.4
[ACZ16]	ECDSA-256	128	0	V7	6,816/4,442/20/0	225	1.49	10.2
[ACZ16]	ECDSA-521	256	0	V7	8,273/7,689/64/0	161	5.02	41.5
[SA14]	RSA-2048	112	0	V7	3,558 slices/54/0	399	5.68	≈60
[PDG14]	BLISS-IV	192	?	S6	6,438/6,198/5/7	135	0.35	2.25
[BHH+15]	SPHINCS-256	256	128	Haswell CPU E3-1275 (1 core)		3500	14.7	-

The complex post-quantum secure, lattice-based BLISS signature scheme is an order of magnitude more efficient than our SPHINCS core. In [PDG14], even faster versions of BLISS (with lower security levels) are reported. However, using SPHINCS has the advantage that its security is fundamentally based on the difficulty of breaking the underlying hash function, which is well analyzed and well understood while the security of BLISS is not that clear (see section 1).

6 Future Work

Among the highest priorities for future works are side-channel analyses. This will go along with the implementation of possible countermeasures. Also, if the software-based BLAKE-512 calculation looks as if it will become a limiting factor in the whole signing system, a BLAKE-512 unit will have to be added to the SPHINCS accelerator.

7 Conclusion

We showed that the SPHINCS signature scheme can be extensively parallelized. All logic operations in the scheme are based on bit rotation, xor, and unsigned addition. Therefore, the algorithm fits very well into an FPGA. A comparison to a software-based implementation supports this statement. To the authors' knowledge, this paper presents the first hardware architecture that accelerates SPHINCS-256 signing, key generation and signature verification. Thousands of signatures per second can be generated on an FPGA if our SPHINCS accelerator is instantiated several times. Signing delay is around 1.53 ms, and verification takes just 65 μs. Compared to the optimized vector-processor implementation, the presented SPHINCS accelerator signs an order of magnitude faster. A comparison to FPGA-based implementations of the classical signature schemes RSA and ECDSA shows that the SPHINCS accelerator performance is highly competitive.

Acknowledgments

We thank the anonymous reviewers for their accurate reviews and their valuable comments. This work was supported by the Commission for Technology and Innovation CTI (also known as Innosuisse). CTI is the federal agency responsible for encouraging science-based innovation in Switzerland.

References

[ABO$^+$14] Nuray At, Jean-Luc Beuchat, Eiji Okamoto, Ismail San, and Teppei Yamazaki. Compact Hardware Implementations of ChaCha, BLAKE, Threefish, and Skein on FPGA. *IEEE Trans. on Circuits and Systems*, 61-I(2):485–498, 2014.

[ACZ16] Dorian Amiet, Andreas Curiger, and Paul Zbinden. Flexible FPGA-Based Architectures for Curve Point Multiplication over GF(p). In *2016 Euromicro Conference on Digital System Design, DSD 2016*, pages 107–114. IEEE Computer Society, 2016.

[AE17] Jean-Philippe Aumasson and Guillaume Endignoux. Improving Stateless Hash-Based Signatures. Cryptology ePrint Archive, Report 2017/933, 2017.

[AHMP10] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C W Phan. SHA-3 proposal BLAKE. SHA3 competition, 2010. http://www.131002.net/blake/blake.pdf.

[BDS09] Johannes Buchmann, Erik Dahmen, and Michael Szydlo. Hash-based Digital Signature Schemes. In Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors, *Post-Quantum Cryptography*, pages 35–93. Springer Berlin Heidelberg, 2009.

[Ber08] Daniel J. Bernstein. ChaCha, a variant of Salsa20, 2008. http://cr.yp.to/chacha/chacha-20080120.pdf.

[BGD$^+$06] Johannes A. Buchmann, Luis Carlos Coronado García, Erik Dahmen, Martin Döring, and Elena Klintsevich. CMSS - An Improved Merkle Signature Scheme. In *Progress in Cryptology - INDOCRYPT 2006*, volume 4329 of *LNCS*, pages 349–363. Springer, 2006.

[BHH$^+$15] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: Practical Stateless Hash-Based Signatures. In *Advances in Cryptology - EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 368–397. Springer, 2015.

[BKL$^+$17] Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. Gimli: a cross-platform permutation. Cryptology ePrint Archive, Report 2017/630, 2017.

[BL17] Daniel J. Bernstein and Tanja Lange. Post-quantum cryptography - dealing with the fallout of physics success. Cryptology ePrint Archive, Report 2017/314, 2017.

[BS02] Lynn Margaret Batten and Jennifer Seberry. Better than BiBa: Short One-Time Signatures with Fast Signing and Verifying. In *Information Security and Privacy, ACISP 2002*, volume 2384 of *LNCS*, pages 144–153. Springer, 2002.

[HRS15] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. ARMed SPHINCS – Computing a 41KB signature in 16KB of RAM. Cryptology ePrint Archive, Report 2015/1042, 2015.

[Hül13] Andreas Hülsing. W-OTS+ - Shorter Signatures for Hash-Based Signature Schemes. In *Progress in Cryptology - AFRICACRYPT 2013*, volume 7918 of *LNCS*, pages 173–188. Springer, 2013.

[Köl17] Stefan Kölbl. Putting Wings on SPHINCS. Cryptology ePrint Archive, Report 2017/898, 2017.

[Lam79] Leslie Lamport. Constructing Digital Signatures from a One Way Function. SRI International, CSL-98, 1979.

[Mer89] Ralph C. Merkle. A Certified Digital Signature. In *CRYPTO '89 Proceedings on Advances in Cryptology*, pages 218–238. Springer-Verlag New York, Inc., 1989.

[Mos15] Michele Mosca. Cybersecurity in an era with quantum computers: will we be ready? Cryptology ePrint Archive, Report 2015/1075, 2015.

[Nat09] National Institute of Standards and Technology. Digital Signature Standard (DSS). FIPS-PUB 186-4, 2009.

[PDG14] Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. Enhanced lattice-based signatures on reconfigurable hardware. In *Cryptographic Hardware and Embedded Systems - CHES 2014*, volume 8731 of *LNCS*, pages 353–370. Springer, 2014.

[RSA78] R L Rivest, A Shamir, and L Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[SA14] Ismail San and Nuray At. Improving the computational efficiency of modular operations for embedded systems. *Journal of Systems Architecture - Embedded Systems Design*, 60(5):440–451, 2014.

[Sho97] Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.