XMSS and Embedded Systems

XMSS Hardware Accelerators for RISC-V

Wen Wang[1], Bernhard Jungk[2], Julian Wälde[3], Shuwen Deng[1], Naina Gupta[4], Jakub Szefer[1] and Ruben Niederhagen[3]

[1] Yale University, New Haven, CT, USA
{wen.wang.ww349,shuwen.deng,jakub.szefer}@yale.edu
[2] Independent Researcher
bernhard@projectstarfire.de
[3] Fraunhofer SIT, Darmstadt, Germany
ruben@polycephaly.org
[4] Fraunhofer Singapore, Singapore
naina.gupta@fraunhofer.sg

Abstract. We describe a hardware-software co-design for the hash-based post-quantum signature scheme XMSS on a RISC-V embedded processor. We provide software optimizations for the XMSS reference implementation for SHA-256 parameter sets and several hardware accelerators that allow to balance area consumption and performance based on individual needs. The version with the best time-area product for key generation gives a $47\times$ speedup in wall-clock time at $5.1\times$ larger resource requirements; the best speedup of $52\times$ is achieved at a higher resource cost. For signing, we achieve a maximum speedup of over $23\times$ and for verification of over $18\times$. We tested and measured the cycle counts of our implementation on Intel (Altera) and Xilinx FPGAs. The integration of our XMSS accelerators into an embedded RISC-V processor enables post-quantum secure signatures for a large variety of embedded applications.

Keywords: XMSS · hash-based signatures · post-quantum cryptography · hardware accelerator · FPGA · RISC-V

1 Introduction

Due to the continued computerization and automation of our society, more and more systems from consumer products and Internet-of-Things (IoT) devices to cars, high-speed trains and even nuclear power plants are controlled by embedded computers that often are connected to the Internet. Such devices can have a severe impact not only on our information security but increasingly also on our physical safety. Therefore, embedded devices must provide a high level of protection against cyber attacks — despite their typically restricted computing resources. If an attacker is able to disrupt the authenticity of transmitted data, he or she can undermine security of the system in many ways, e.g., malicious firmware can be loaded, or contents of a digital document can be changed without being detected. Authenticity of the data is commonly ensured using digital signature schemes, often based on the DSA and ECDSA algorithms [NIS13].

Commonly used cryptographic algorithms, however, are vulnerable to attacks using quantum computers: Shor's algorithm [Sho94, Sho99] is able to factor integers and compute discrete logarithms in polynomial time and Grover's algorithm [Gro96] provides a quadratic speedup for brute-force search. In light of recent advances in quantum-computer development and increased research interest in bringing practical quantum computers to life, a new field of post-quantum cryptography (PQC) has evolved [BBD09], which provides cryptographic algorithms that are believed to be secure against attacks using quantum computers. Among these PQC algorithms are a number of algorithms for signing (and verification) of data. This paper focuses on one of these algorithms, the eXtended Merkle Signature Scheme (XMSS), which has recently been standardized by the IETF [HBG+18].

XMSS is a stateful hash-based signature scheme proposed in 2011 by Buchmann, Dahmen and Hülsing [BDH11]. It is based on the Merkle signature scheme [Mer90] and proven to be a forward-secure post-quantum signature scheme with minimal security assumptions: Its security is solely based on the existence of a second pre-image resistant hash function family and a pseudorandom function (PRF) family. Both of these function families can be efficiently constructed even in the presence of large quantum computers [BDH11]. Therefore, XMSS is considered to be a practical post-quantum signature scheme. Due to its minimal security assumptions and its well understood security properties, XMSS is regarded as one of the most confidence-inspiring post-quantum secure signature schemes.

Embedded devices will need to use algorithms such as XMSS to make them future-proof and to ensure their security even in the light of practical quantum computers. One of the increasingly popular processor architectures for embedded devices is the RISC-V architecture. For example, RISC-V is used as a control processor in GPUs and storage devices [Hig18]. It is an open and free architecture that is proving to be a practical alternative to close-source designs. Consequently, this work uses a RISC-V-based system on chip (SoC) called Murax (see Section 2.3.1) as a representative for embedded system architectures and shows how to efficiently deploy the post-quantum secure signature scheme XMSS on an embedded device.

Hash-based signature schemes like XMSS have relatively high resource requirements. They need to perform thousands of hash-computations for key generation, signing and verification and need sufficient memory for their relatively large signatures. Therefore, running such post-quantum secure signature schemes efficiently on a resource-constrained embedded system is a difficult task. This work introduces a number of hardware accelerators that provide a good time-area trade-off for implementing XMSS on RISC-V.

1.1 Related Work

There is currently a small number of publications focusing on FPGA hardware implementations of hash-based signature schemes:

The classic Merkle signature scheme has been investigated and implemented for FPGAs, e.g., in [SH10, Sho10, SHM11]. These implementations, however, are no longer state-of-the-art: they provide none of the additional security features that have been developed for modern hash-based signature schemes like XMSS, LMS [MCF18], and the SPHINCS family [BHH+15].

The stateless hash-based signature scheme SPHINCS-256 [BHH+15] has been implemented in [ACZ18]. This signature scheme is closely related to XMSS and is a predecessor of the SPHINCS+ signature scheme[1], which is one of the submissions in NIST's PQC standardization process[2]. SPHINCS-256 requires the cryptographic primitives BLAKE-256, BLAKE-512, and ChaCha12. The authors provide efficient hardware implementations for these primitives and control logic to enable signing, key generation, and signature verification. They report timings of 1.53 ms for signing and 65 µs for verification, but no timings for key generation. Their source code is not freely available.

Our Contributions. We provide software optimizations and hardware accelerators for a software-hardware co-design of XMSS on a RISC-V-based embedded system by developing:

[1]`https://sphincs.org/`
[2]`https://csrc.nist.gov/Projects/Post-Quantum-Cryptography`

- SHA-256-specific software optimizations for the XMSS reference implementation and
- hardware accelerators with improved time-area trade-off for XMSS:
  - a general-purpose SHA-256 accelerator,
  - a SHA-256 accelerator with XMSS-specific optimizations,
  - a WOTS-chain accelerator, and
  - an XMSS-leaf generation accelerator.

We will release our software-hardware co-design under an open source license to enable academia and industry to fully exploit the benefits of our work.

1.2 Structure of the Paper

The following section provides some background information on the XMSS signature scheme, the hash function SHA-256, the RISC-V architecture, and our FPGA setup that we used for testing. Then, in Section 3, we introduce our software optimizations for the XMSS reference implementation. In Section 4, we describe the dedicated hardware modules we developed for accelerating XMSS computations. Details about the design of the interfaces between the RISC-V software and the hardware accelerators are also provided. A performance evaluation of our XMSS software-hardware co-design on a RISC-V Murax SoC is provided in Section 5.

2 Preliminaries

In this section, we give an introduction to the relevant aspects of the XMSS signature scheme, briefly recapitulate the functionalities of SHA-256, and give a brief introduction to the RISC-V ISA and the Murax SoC implementation of this ISA. Finally, we briefly describe the FPGA setup that we are using for our evaluation.

2.1 XMSS

The eXtended Merkle Signature Scheme (XMSS) [HBG+18] is a stateful signature scheme based on the Merkle signature scheme [Mer90]. Similar to the Merkle signature scheme, XMSS uses the Winternitz one-time signature scheme (WOTS or Winternitz-OTS) to sign individual messages [Mer90]. One private/public WOTS key pair is used to sign one single message (with the private secret key) and to verify the signature (with the corresponding public verification key). To be able to sign up to 2^h messages, XMSS uses 2^h pairs of WOTS secret and verification keys. To reduce the size of the public key, a Merkle hash tree and binary L-trees are used to reduce the authenticity of many WOTS verification keys to one XMSS public key. Since each WOTS key must only be used once, the signer needs to remember which WOTS keys already have been used. Hence, the scheme is stateful. Figure 1 shows the overall structure of XMSS.

The XMSS standard also defines multi-tree versions of XMSS (XMSS^MT) where the leaf nodes of a higher-level tree are used to sign the root of another tree. In this paper, we only consider single-tree XMSS. However, our results can be mapped to multi-tree XMSS in a straightforward way. For a detailed description of XMSS (and XMSS^MT) please refer to IETF RFC 8391 [HBG+18] and to [BDH11].

In the following we briefly introduce the XMSS address scheme, WOTS, the L-tree construction, and the procedure for constructing the Merkle tree. We also give an introduction to XMSS key generation, signing, and verification.

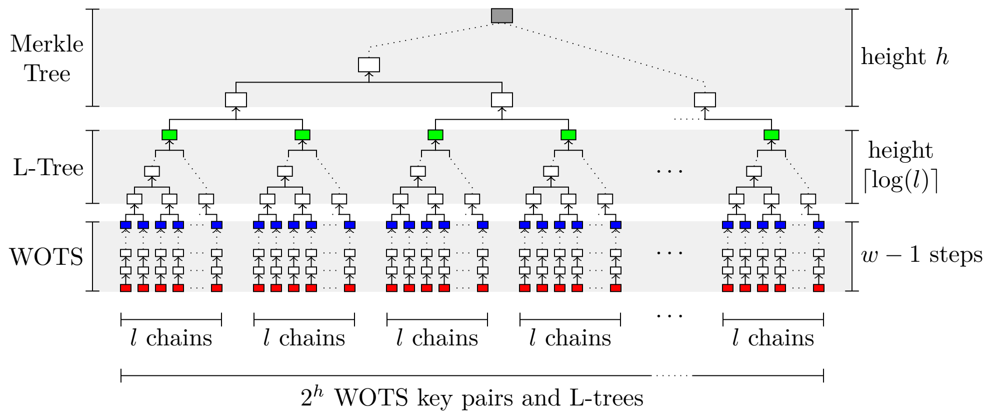
Merkle Tree
L-Tree
WOTS
height h
height $\lceil \log(l) \rceil$
$w-1$ steps
l chains
2^h WOTS key pairs and L-trees

Figure 1: XMSS tree with binary Merkle hash tree and L-tree plus WOTS instances as leafs. Red nodes are the WOTS private key and blue nodes are the WOTS public key values. Green nodes are the L-Tree roots and the gray node is the XMSS public key root.

2.1.1 Address Scheme

XMSS uses a hash function address scheme throughout the Merkle tree, L-tree, and WOTS computations to uniquely identify each individual step in the overall graph. These addresses are used to derive keys for keyed hash functions that are unique for each specific location in the graph. Each address is composed of eight 32 bit fields, with fields for, e.g., the level within a tree and the leaf index. In total, an XMSS address has a size of 256 bit. For more details about the hash function address scheme, please refer to IETF RFC 8391 [HBG+18, Sect. 2.5].

2.1.2 Winternitz OTS

The WOTS scheme was first mentioned in [Mer90]. For signing a message digest D of n-byte length, WOTS uses a cryptographically secure hash function with n-byte output strings to compute hash chains. The message digest is interpreted as binary representation of an integer d. First, d is split into $l_1 = \lceil 8n/\log_2(w) \rceil$ base-w words $d_i, 0 \leq i < l_1$ and a checksum $c = \sum_{i=0}^{l_1} w - 1 - d_i$ is computed for these base-w words (w is called the "Winternitz parameter"). The checksum c is split into $l_2 = \lfloor \log_2(l_1(w-1))/\log_2(w)) + 1 \rfloor$ base-w words $c_i, 0 \leq i < l_2$ as well. WOTS key generation, signing, and verification are performed as follows:

- To *create* a private/public WOTS key pair, Alice computes $l = l_1 + l_2$ secret strings $s_{0,0}, s_{0,1}, \ldots, s_{0,l-1}$, each of n-byte length (for example using a secret seed and a PRF). These l n-byte strings are the private WOTS key. Then, Alice uses a chaining function to compute l hash chains of length $w-1$, hashing each $s_{0,i}$ iteratively $w-1$ times. The resulting chain-head values $s_{w-1,i}, 0 \leq i < l$ of n-byte length are the public WOTS key and are published by Alice.
- To *sign* a message digest D split into l_1 base-w words together with l_2 base-w checksum values computed as described above, Alice (re-)computes the intermediate chain values $(s_{d_0,0}, s_{d_1,1}, \ldots, s_{d_{l_1-1},l_1-1}, s_{c_0,0}, s_{c_1,1}, \ldots, s_{c_{l_2-1},l_2-1})$ starting from her private key values. These $l = l_1 + l_2$ values are the signature.
- When Bob wants to *verify* the signature, he recomputes the remaining chain steps by applying, e.g., $w - 1 - d_0$ hash-function iterations to signature value $s_{d_0,0}$ and compares the resulting values with the corresponding public key values. If all chain-head values match the public WOTS key, the signature is valid.

XMSS uses a modified WOTS scheme, sometimes referred to as WOTS+ or as W-OTS+ [Hül13]; we use the term WOTS+ only when a explicit distinction from "original" WOTS is required for clarification. WOTS+ uses a function chain() as chaining function that is a bit more expensive than the simple hash-chain function described above. The function chain() uses a keyed pseudo-random function $\mathrm{prf}_k : \{0,1\}^{256} \mapsto \{0,1\}^{8n}$ and a keyed hash-function $\mathrm{f}_{k'} : \{0,1\}^{8n} \mapsto \{0,1\}^{8n}$. Within each chain step, the function chain() first computes a unique n-byte key k' and a unique n-byte mask using the $\mathrm{prf}_k()$ function. The input to $\mathrm{prf}_k()$ is the hash function address of the current step (including the chain step and a marker for the usage as key or as mask). The key k for $\mathrm{prf}_k()$ is a seed that is part of the XMSS public key. The mask is then XOR-ed with the n-byte output from the previous chain-function call (or the initial WOTS+ chain n byte input string) and the result is used as input for the hash-function f() under the key k', which gives the n-byte output of the chaining function chain() in the last iteration step.

The WOTS+ secret key consists of l (l is defined as described above for WOTS) pseudo-random strings of n-bytes in length. The XMSS specification does not demand a certain function to compute the WOTS+ private key. In the XMSS reference implementation, they are generated using the $\mathrm{prf}_k()$ function with the local address (including the chain index) as input and keyed with the XMSS secret key seed. Each WOTS+ secret key maps to one corresponding WOTS+ public key, which is computed by calling the chaining function chain() with $w - 1$ iteration steps. Signing and verification in WOTS+ work as described above for WOTS using the WOTS+ chaining function. The more complex structure of the chaining function of WOTS+ compared to WOTS is required for multi-target resistance and within the XMSS security proof.

2.1.3 L-tree

The leaf nodes of an XMSS tree are computed from the WOTS+ public keys by using an unbalanced binary tree of l leaf nodes (one leaf node for each WOTS+ public key value), hence called L-tree. The nodes on each level of the L-tree are computed by hashing together two nodes from the lower level. A tree hash function $\mathrm{hash}_{\mathrm{rand}} : \{0,1\}^{8n} \times \{0,1\}^{8n} \mapsto \{0,1\}^{8n}$ is used for this purpose.

The function $\mathrm{hash}_{\mathrm{rand}}()$ uses the keyed pseudo-random function $\mathrm{prf}_k()$ and a keyed hash-function $\mathrm{h}_{k''} : \{0,1\}^{16n} \mapsto \{0,1\}^{8n}$. First, an n-byte key k'' and two n-byte masks are computed using the $\mathrm{prf}_k()$ with the address (including the L-tree level and node index) as input and the same public seed as used for WOTS+ as key. The masks are then each XOR-ed to the two n-byte input strings representing the two lower-level nodes and the results are concatenated and used as input for the hash-function h() keyed with k'', which gives the n-byte output of the tree hash function $\mathrm{hash}_{\mathrm{rand}}()$.

To be able to handle the pairwise hashing at levels with an odd number of nodes, the last node on these levels is lifted to a higher level until another single node is available. The root of the L-tree gives one single hash-value, combining the l WOTS+ public keys into one WOTS+ verification key.

2.1.4 XMSS Merkle Tree

In order to obtain a small public key, the authenticity of many WOTS verification keys (i.e., L-tree root keys) is reduced to one XMSS public key using a binary Merkle tree. Similar to the L-tree construction described above, on each level of the binary tree, neighbouring nodes are pairwise hashed together using the $\mathrm{hash}_{\mathrm{rand}}()$ function to finally obtain one single root node that constitutes the XMSS public key root (see Figure 1).

2.1.5 XMSS Key Generation

XMSS key generation is quite expensive: In order to compute the XMSS public key, i.e., the root node of the Merkle tree, the entire XMSS tree needs to be computed. Depending on the height h of the tree, thousands to millions of hash-function calls need to be performed. XMSS key generation starts by generating 2^h leaf nodes of the Merkle tree. Each leaf node consists of an WOTS+ instance together with an L-Tree. For each WOTS+ instance, first l WOTS+ private keys are generated. These are then used to compute the l WOTS+ chains to obtain l WOTS+ public keys and then the L-trees on top of these. Once all 2^h L-tree root nodes have been computed, the Merkle tree is computed to obtain the XMSS public key.

The XMSS public key consists of the n-byte Merkle tree root node and of the n-byte public seed required by the verifier to compute masks and public hash-function keys using the $\mathrm{prf}_k()$ within the chain, L-tree, and Merkle tree computations. The XMSS standard does not define a format for the XMSS private key. In the XMSS reference implementation that accompanies the standard, an n-byte secret seed is used to generate the WOTS secrets using a pseudo random function (e.g., the $\mathrm{prf}_k()$).

2.1.6 XMSS Signature Generation

XMSS is a stateful signature scheme: Each WOTS+ private/public key pair must be used only once; otherwise, the scheme is not secure. In order to determine which WOTS+ key pair already has been used, an n-byte leaf index (the state) is stored with the private key. The index defines which WOTS+ key pair will be used for the next signature; after each signature generation, the index must be increased.

Similar to most signature schemes, for signing an arbitrary length message or document M, first a message digest of M is computed; details can be found in [HBG$^+$18, Sect. 4.1.9]. The digest M' is then signed using the selected WOTS+ instance. This results in l n-byte values corresponding to the base-w decomposition of M' including the corresponding check sum. Furthermore, in order to enable the verifier to recompute the XMSS public root key from a leaf node of the Merkle tree, the signer needs to provide the verification path in the Merkle tree, i.e., h n-byte nodes that are required for the pair-wise hashing in the binary Merkle tree, one node for each level in the Merkle tree.

Therefore, in the worst case, the signer needs to recompute the entire XMSS tree in order to select the required values for the verification path. There are several optimization strategies using time-memory trade-offs to speed up signature generation. For example, the signer can store all nodes of the Merkle tree up to level h' alongside the private key. Then, when signing, he only needs to compute a $(h - h')$-height sub-tree including the WOTS leafs and can reproduce the signature path for the remaining h' levels from the stored data. Other algorithms with different trade-offs exist; for example the BDS tree traversal algorithm targets at reducing the worst case runtime of signature generation by computing a certain amount of nodes in the Merkle tree at each signature computation and by storing them alongside the XMSS state [BDS08].

2.1.7 XMSS Signature Verification

Compared to key generation, XMSS signature verification is fairly cheap: An XMSS public key contains the Merkle root node and the public seed. An XMSS signature contains the WOTS leaf index, l WOTS-signature chain values, and the verification path consisting of h Merkle-tree pair values, one for each level in the tree. The verifier computes the message digest M' and then recomputes the WOTS verification key by completing the WOTS chains and computing the L-tree. He then uses the Merkle-tree pair values to compute the path through the Merkle tree and finally compares the Merkle tree root node he obtains with the root node of the sender's public key. If the values are equal, verification succeeds and the signature is sound; otherwise verification fails and the signature is rejected.

2.1.8 Parameter Set

RFC 8391 defines parameter sets for the hash functions SHA-2 and SHAKE targeting classical security levels of 256 bit and 512 bit in order to provide 128 bit and 256 bit of security respectively against attackers in possession of a quantum computer [HBG+18, Sect. 5]. Parameter sets with $n = 32$ provide a classical security level of 256 bit while those with $n = 64$ provide a classical security level of 512 bit.

For this work, we focus on the SHA-256 hash function, thus $n = 32$. In this case, the keyed hash functions $\mathrm{prf}_k : \{0,1\}^{256} \mapsto \{0,1\}^{256}$, $\mathrm{f}_{k'} : \{0,1\}^{256} \mapsto \{0,1\}^{256}$, and $\mathrm{h}_{k''} : \{0,1\}^{512} \mapsto \{0,1\}^{256}$, are implemented by computing the input to SHA-256 as concatenation of:

- a 256 bit hash-function specific domain-separator,
- the 256 bit hash-function key, and
- the 256 bit or 512 bit hash-function input.

For SHA-256, there are three different parameter sets provided in RFC 8391 [HBG+18, Sect. 5.3], all with $n = 32$ and $w = 16$ but with $h = 10$, $h = 16$, or $h = 20$. Since the tree height h has a significant impact on the cost of key generation, signing, and verification, one of the questions we want to answer in this work is what tree height is feasible for resource-restricted embedded systems.

2.2 SHA-256

The hash function SHA-256 [NIS12] computes a 256 bit hash value from a variable-length input. SHA-256 uses a 256 bit internal state that is updated with 512 bit blocks of the input. Therefore, SHA-256 defines a padding scheme in order to extend variable-length inputs to be a multiple of 512 bit. SHA-256 works as follows:

1. Initialize the internal state with a well-defined IV (see [NIS12, Sect. 4.2.2]).
2. Extend the ℓ-bit input message with a padding to make the length of the padded input a multiple of 512 bit:
    - append a single 1 bit to the input message, then
    - append $0 \leq k$ 0 bit such that $\ell + 1 + k + 64$ is minimized and is a multiple of 512, and finally
    - append ℓ as a 64 bit big-endian integer.
3. Iteratively apply a compression function to all 512 bit blocks of the padded input and the current internal state to obtain the next updated internal state.
4. Once all 512 bit blocks have been processed, output the current internal state as the hash value.

The compression function uses the current internal state and a 512 bit input block and outputs a new internal state. For SHA-256, the compression function is composed of 64 rounds.

2.3 RISC-V

The RISC-V instruction set architecture (ISA) is a free and open architecture, overseen by the RISC-V Foundation with more than 100 member organizations[3]. The RISC-V ISA

[3]`https://riscv.org/`

has been designed based on well-established reduced instruction set computing (RISC) principles. It has a modular structure: There are base-sets for 32-bit (RV32I and RV32E), 64-bit (RV64I), and 128-bit (RV128I) instructions. In addition, there are instruction set extensions, e.g., instructions for integer multiplication and division (M), single and double precision floating-point operations (F, D), and SIMD operations (P).

Due to its modular design, the RISC-V ISA is an increasingly popular architecture for embedded systems. It is used, e.g., as a control processor in GPUs and in storage devices [Hig18], for secure boot and as USB security dongle [Mer18], and for building trusted execution environments (TEE) with secure hardware enclaves [4]. Since the RISC-V ISA is an open standard, researchers and industry can easily extend and adopt it in their designs without IP constraints.

2.3.1 VexRiscv and Murax SoC

VexRiscv[5] is a 32-bit RISC-V CPU implementation written in SpinalHDL[6]. It supports the RV32IM instruction set and implements a 5-stage in-order pipeline. All complementary and optional components are implemented as plugins and therefore can easily be integrated and adapted into specific processor setups as needed. VexRiscv also provides memories and caches, IO peripherals, and buses, which can be chosen and combined as required.

The VexRiscv project provides a predefined processor setup called “Murax SoC” that aims at small resource usage. The Murax SoC integrates the VexRiscv CPU with a shared instruction and data memory, an Advanced Peripheral Bus (APB), a JTAG programming interface, a UART interface, and further optional peripherals. The Murax SoC has very low resource requirements (e.g., only 1380 ALMs on a Cyclone V FPGA) and can operate on its own without any further external components. Therefore, it is a good representative for an embedded system processor with low resources. The Murax SoC can be extended with new hardware accelerators using the APB bus. We used this feature for our XMSS accelerators. Murax SoC designs can be synthesized for FPGAs and with small modifications for ASICs as well.

2.4 Setup

We evaluated our design using a DE1-SoC evaluation board from Terasic as test-platform. This board has an Intel (formerly Altera) Cyclone V SoC 5CSEMA5F31C6 device with about 32,000 adaptive logic modules (ALMs) and about 500 KB of on-chip memory resources. (We do not use the DSP resources or the ARM Cortex-A9 CPU of the device.) We used Intel Quartus Software Version 16.1 (Standard Edition) for synthesis. On the DE1-SoC, we are running the Murax RISC-V SoC described in Section 2.3.1 with additional accelerators that will be described in Section 4. The DE1-SoC board is connected to a host computer by a USB-JTAG connection for programming the FPGA, a USB-serial connection for IO of the Murax SoC, and a second USB-JTAG connection for programming and debugging the software on the Murax SoC.

We configured the on-chip RAM size of the Murax SoC to 128 kB, which is sufficient for our all our experiments. We tested our implementations on the DE1-SoC board at its default clock frequency of 50 MHz; however, to achieve a fair comparison, our speedup reports presented in the following sections are based on the maximum frequency reported by the synthesis tools.

Our implementation is neither platform-specific nor dependent on a specific FPGA vendor; we also successfully tested our implementations with very similar results on an

[4] `https://keystone-enclave.org/`
[5] `https://github.com/SpinalHDL/VexRiscv/`
[6] `https://spinalhdl.github.io/SpinalDoc/`

Arty S7 development board from Digilent with a medium-size Xilinx Spartan 7 FPGA with part number XC7S50-1CSGA324C.

3 Software Implementation and Optimization

We used the official XMSS reference implementation[7] as software-basis for this work. The reference implementation is using OpenSSL for the hash functions. We applied minor modifications to the XMSS reference code to link against the mbed TLS library[8] instead, because mbed TLS generally is more suitable for resource-restricted embedded platforms like the Murax RISC-V platform and its SHA-256 implementation has less library-internal dependencies than that of OpenSSL, which simplifies stand-alone usage of SHA-256.

The tree-hash algorithm [HBG+18] used for computing the XMSS public key and the authentication path within the Merkle tree requires an exponential number of 2^h WOTS operations for computing tree leafs. However, key generation and signing is not memory intensive when the tree is computed with a depth-first strategy. The XMSS reference implementation provides two algorithms for signature generation. The first approach (implemented in file "xmss_core.c") straightforwardly re-computes all tree leaf nodes in order to compute the signature authentication path and therefore has essentially the same cost as key-generation. This approach does not require to store any further information. The second approach (implemented in file "xmss_core_fast.c") uses the BDS algorithm [BDS08] to make a trade-off between computational and memory complexity. It requires to additionally store a state along the private key. Both versions can be used with our hardware accelerators. Since the runtime of the basic signature algorithm is almost identical to key generation (also when using our hardware accelerators), we are using the fast BDS version of the signature algorithm for our performance reports.

To have a fair reference point for the comparison of a pure software implementation with our hardware accelerators, we implemented two software optimizations for the XMSS reference software implementation as described in the following sub-sections. These optimizations are also helpful on other processor architectures but only work for SHA-256 parameter sets, because they depend on the specific SHA-256 block size and padding scheme. We are going to provide our software optimizations to the maintainers of the XMSS reference implementation so they can integrate them if they wish to.

3.1 Fixed Input Length

Within the XMSS scheme, overall the most time is spent inside the hash-function calls. Therefore, the SHA-256 function is most promising for optimization efforts. The main interface to SHA-256 in mbed TLS has three functions, `mbedtls_sha256_init`, `mbedtls_sha256_update`, and `mbedtls_sha256_finish`. The "init"-function initializes the internal state of the SHA-256 implementation. The "update"-function allows to feed in message chunks of arbitrary size and updates the internal state accordingly. The "finish" function finally adds the padding and returns the message digest. Internally, these functions need to adapt arbitrary-length message chunks to the SHA-256 input block size of 512 bit: If the size of message-chunk input to `mbedtls_sha256_update` is not a multiple of 512 bit, the remaining data is buffered alongside the internal state and used either in the next "update" or in the final "finish" call.

The SHA-256 implementation of mbed TLS is intended to hash messages of an arbitrary length: When the "finish" function is called, the actual length of the entire message is computed as sum over the lengths of all individual message chunks and the padding is generated accordingly. However, within the XMSS scheme, the inputs of almost all

[7]`https://github.com/joostrijneveld/xmss-reference/`, commit 06281e057d9f5d22
[8]`https://tls.mbed.org/`

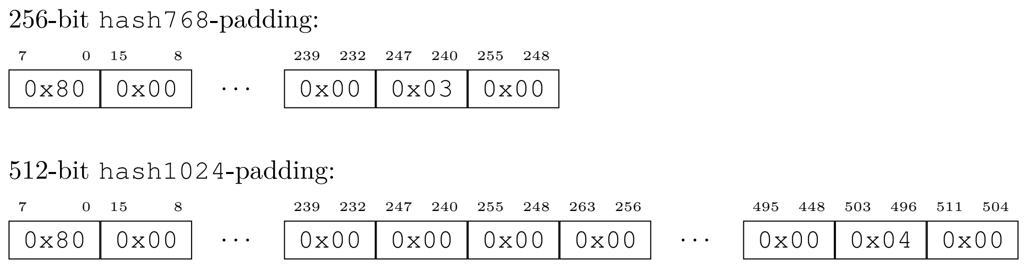

Figure 2: Fixed padding for `hash768` and `hash1024`.

SHA-256 calls have a well-known, fixed length: A general, arbitrary-length SHA-256 computation is only required when computing the actual hash digest of the input message, which is called only once for signing and once for verifying. For all the other SHA-256 calls, the length of the input data is either 768 bit or 1024 bit depending on where SHA-256 is called within the XMSS scheme: An input length of 768 bit is required within the PRF and within the WOTS-chain computation; an input length of 1024 bit is required within the Merkle tree and the L-trees to hash two nodes together. Therefore, we can eliminate the overhead for the padding computation of the SHA-256 function by "hardcoding" the two required message paddings, given that their length is known beforehand.

Implementation: We implemented two specialized SHA-256 functions: `hash768` targets messages with a fixed length of 768 bit and `hash1024` targets messages with fixed length of 1024 bit. Figure 2 shows the padding for `hash768` and `hash1024`. Since SHA-256 has a block size of 512 bit, two blocks are required to hash a message of length 768 bit. Therefore, we need to hardcode a 256 bit padding for `hash768` to fill up the second block to 512 bit. When a 768 bit message is fed to the `hash768` function, the 256 bit padding is appended to the message. Then, the new 1024 bit padded message is divided into two 512 bit blocks and the compression function is performed on each of them one by one. Once the compression function on the second message block has finished, the internal state is read out and returned as the output.

The SHA-256 standard always demands to append a padding even if the input length is a multiple of 512 bit. Therefore, for the `hash1024` function a 512 bit padding is hardcoded similarly to `hash768` and three calls to the compression function are performed.

Evaluation: Table 1 shows a comparison of the original XMSS reference implementation with an optimized version making use of the "fixed input lengths" optimization on the Murax SoC. The speedup for 768 bit inputs is about 1.07× and for 1024 bit inputs about 1.04×. The use of 768 bit inputs is more common during the XMSS computations. Therefore, we see an about 1.06× speedup for WOTS computations, key generation, signing, and verification. We observed a similar speedup in an Intel CPU.

3.2 Pre-Computation

Within XMSS, SHA-256 is used to implement four different keyed hash-functions, the function `F` for computing f() in the WOTS-chains, the function `H` for h() in the tree hashing, and the function `PRF` for computing the prf(), generating masks and hash-function keys. Furthermore, SHA-256 is used to compute the message digest that is signed using a WOTS private key. The domain separation between and the keying for these four functions are achieved by computing the input to SHA-256 as the concatenation of a 256 bit domain separator value (distinct for these four functions), the 256 bit hash key, and the hash-function input. Since SHA-256 operates on 512 bit blocks, one entire block is required for

	"original"	+ "fixed input length"		+ "pre-computation"		
	Cycles (A)	Cycle (B)	Speedup (AB)	Cycles (C)	Speedup (BC)	Speedup (AC)
`hash768`	11.5×10^3	10.7×10^3	1.07	5.87×10^3	1.83	1.95
`hash1024`	16.2×10^3	15.6×10^3	1.04	—	—	—
WOTS-chain	571×10^3	530×10^3	1.08	371×10^3	1.43	1.54
WOTS-leaf	42.2×10^6	39.8×10^6	1.06	27.7×10^6	1.44	1.53
key generation	43.3×10^9	40.8×10^9	1.06	28.3×10^9	1.44	1.53
signing	58.3×10^6	55.0×10^6	1.06	38.4×10^6	1.43	1.52
verification	26.7×10^6	25.2×10^6	1.06	17.4×10^6	1.45	1.54

Table 1: Cycle count and speedup of the "fixed input length" optimization and for both, the "fixed input length" and the "pre-computation" optimizations, on the Murax SoC (all rounded to 3 significant digits).

domain separation and keying of the respective hash function.

In case of the `PRF`, for all public-key operations when generating masks and hash-function keys for the WOTS chain, the L-tree and Merkle tree operations, the key to the `PRF` is the 256 bit XMSS public seed. Thus, both the 256 bit domain separator and the 256 bit hash-function key are the same for all these calls for a given XMSS key pair. These two parts fit exactly into one 512 bit SHA-256 block. Therefore, the internal SHA-256 state after processing the first 512 bit block is the same for all these calls to the `PRF` and based on this fact, we can save one SHA-256 compression function call per `PRF`-call by pre-computing and replaying this internal state. The internal state can either be computed once and stored together with the XMSS key or each time an XMSS operation (key generation, signing, verification) is performed. A similar optimization was also implemented in the reference implementation[9] of the hash-based signature scheme LMS [MCF18].

Implementation: At the first call to `PRF`, we store the SHA-256 context of mbed TLS for later usage after the first compression function computation. The state includes the internal state and further information such as the length of the already processed data.

When the `PRF` is called during XMSS operations, we first create a copy of the initially stored `PRF` SHA-256 context and then perform the following prf() operations based on this state copy, skipping the first input block. The cost for the compression function call on the first SHA-256 block within the `PRF` is therefore reduced to a simple and inexpensive memory-copy operation.

Evaluation: Performance measurements and speedup for our pre-computation optimization are shown in Table 1. For `hash768` used in the `PRF`, we achieve an 1.83× speedup over the "fixed input length" optimization (column "Speedup (BC)"), because only one SHA-256 block needs to be processed instead of two. Compared to the original non-optimized version, with both optimizations (including "fixed input length") enabled we achieve an almost 2× speedup (column "Speedup (AC)").

The function `F` for computing WOTS-chains requires two calls to the `PRF` (each on two SHA-256 blocks) for generating a key and a mask and one call to `hash768` (on two SHA-256 blocks). Without pre-computation, six calls to the SHA-256 compression function are required. With a pre-computed initial state for the `PRF`, only four calls to the SHA-256 compression function are required, saving one third of the compression function calls. This optimization leads to a 1.43× speedup for WOTS-chain computations (row

[9] `https://github.com/cisco/hash-sigs`

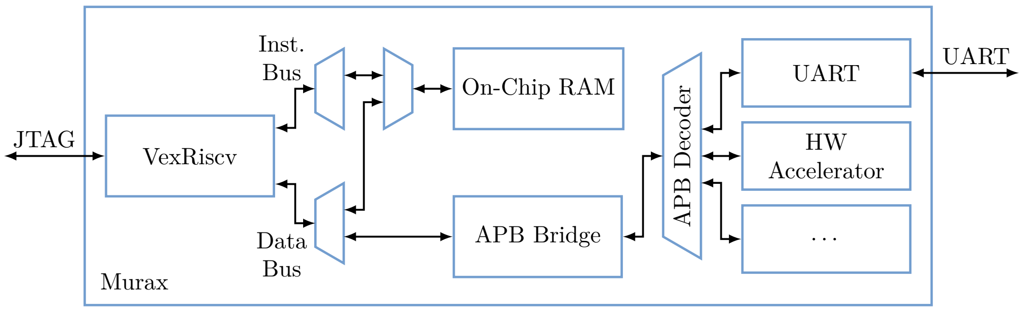

Figure 3: Schematic of the Murax SoC with hardware accelerators connected to the APB.

"WOTS-chain", column "Speedup (BC)"). The overall speedup including both optimizations "pre-computation" and "fixed input length" is 1.54×.

For L-tree computations, within the randomized tree hashing function `H`, there are three calls to the `PRF` (each on two SHA-256 blocks) for computing two masks and one hash-function key and one call to `hash1024` (on three SHA-256 blocks). Without pre-computation, nine calls to the SHA-256 compression function are required. With a pre-computed initial state for the `PRF`, only six calls to the SHA-256 compression function are required, again saving one third of the compression function calls. This optimization leads to a 1.44× speedup for the overall XMSS leaf computations (see Table 1, row "WOTS-leaf"). The speedup for both optimizations together including "fixed input length" is around 1.53×.

The expected speedup for Merkle tree computations is about the same as for the L-tree computations since the trees are constructed in a similar way. Table 1 shows that we achieve an overall speedup of more than 1.5× also for the complete XMSS operations, i.e., key generation, signing, and verification.

4 Hardware Acceleration

To further accelerate the XMSS computations, we developed several dedicated hardware modules together with software interfaces for the XMSS software. As shown in Figure 3, the Murax SoC uses an APB for connecting peripherals to the main CPU core. The peripheral can be accessed by the software running on the Murax SoC via control and data registers that are mapped into the address space. Therefore, the software interface can simply use read and write instructions to communicate with a hardware module. Due to the modularity of the VexRiscv implementation, dedicated hardware modules can be easily added to and removed from the APB before synthesis of the SoC (see Section 2.4).

We developed a general-purpose SHA-256 accelerator for accelerating the compression function of SHA-256 in hardware and the following XMSS-specific hardware accelerators: an XMSS-specific SHA-256 accelerator with fixed-length SHA-256 padding and an optional internal storage for pre-computation, a WOTS-chain accelerator for the WOTS computations, and an XMSS-leaf generation accelerator for WOTS and L-tree computations.

4.1 General-Purpose SHA-256 Accelerator

Since hash-function calls are the most frequent operation in the XMSS scheme, the first hardware module we developed is the `SHA256` module which is a general-purpose hash accelerator that accepts variable length inputs. The hardware module `SHA256` has a similar interface as the generic SHA-256 compression function in software: It receives a 512 bit data block as input and computes the compression function, updating an internal 256 bit state. This state can be read out as the 256 bit digest when the SHA-256 computation is finished. Padding is performed in software as before.

Implementation: We developed the module `SHA256` by implementing an iterative version of SHA-256. This provides a good trade-off between the throughput and the area consumption of the implementation [HRG11]. The iterative approach implements a single round of SHA-256 and a repeated usage of the same hardware. Therefore, the number of clock cycles to process one SHA-256 message block is 64.

The `SHA256` module is connected to the APB using a module called `Apb3SHA256` as bridge. The `Apb3SHA256` module connects on one side to the 32 bit data bus and the control signals of the APB and on the other side to the `SHA256` module. It provides one 32 bit control register and a 512 bit data register. The control register and the data register are mapped to the APB as 32 bit words using a multiplexer, selected by the APB address port on APB write. The 512 bit data registers are also directly connected to the 512 bit `data_in` port of the `SHA256` module. The `data_out` port of the `SHA256` module is directly multiplexed to the APB bus, selected by the APB address port on APB read. The `Apb3SHA256` module takes care of forwarding control signals written by software to the 32 bit control register to the `SHA256` module and the output signals of the `SHA256` module to the software-readable control register.

The software optimization of SHA-256 exploiting fixed input lengths of the SHA-256 function described in Section 3.1 can be mapped in a straightforward way to the `SHA256` module. The software prepares the SHA-256 input chunks with pre-defined paddings just as before and then transfers each chunk to the `SHA256` module for processing. Therefore, the speedup achieved for the pure software version can also be exploited for this hardware accelerator.

In order to support the "pre-computation" optimization (Section 3.2), we added an interface to the `SHA256` module that allows to set the internal state of the `SHA256` module from software. Reading the internal state is the same as reading the SHA-256 message digest after the last compression function computation.

We modified the function `mbedtls_sha256_init` from mbed TLS to replace the software implementation of the SHA-256 compression function with a call to our hardware accelerator as follows: The function first sets the `INIT` bit to high in the control register. When this bit is received as high by the `Apb3SHA256` module, it raises the `init_message` signal of the `SHA256` module, which resets the values of internal state registers to the SHA-256 initialization values. In order to set the internal state for the pre-computation optimization, the software writes a previously stored state to the data registers and then sets the control register bit `LOAD_IV` to high. Once the APB interface sees this bit as high, it sets the `init_iv` signal to high and the `SHA256` module sets the internal state to the 256 least significant bits of the input signal `data_in`. When the compression function is called in software, the 512 bit input message block is sent to the `SHA256` module via the APB bus in words of width 32 bit. Then, the `SHA256` computation is triggered by setting the `COMP` bit in the control register to high. When this bit is received as high, the SHA-256-to-APB interface toggles the `start` signal and the `SHA256` module begins the compression-function computation. Once the `SHA256` module is finished, it raises the `done` signal and the APB interface sets the `DONE` bit in the control register to high.

While the hardware is performing the hash computation, the software can go on doing useful stuff in parallel, e.g., transfer the next data block to the `SHA256` module. Once the software is ready to read the result, it polls the control register until the `DONE` bit is set high. The software then can read the 256 bit results via the APB in words of 32 bit.

Evaluation: Table 2 shows performance, resource requirements, and maximum frequency of the `SHA256` module. The module requires 64 cycles (one cycle per round) for computing

Design	Cycles	Area (ALM)	Reg.	Fmax (MHz)	Time (µs)	Time×Area (relative)	Speedup
			one 512 bit block				
`SHA256`	64	1180	1550	101	0.636	—	—
		`hash768` with pre-computation (one 512 bit block)					
`Murax`	4950	1380	1680	136	36.3	9.10	1.00
+ `SHA256`	253	2970	3990	99.7	2.54	1.00	14.3
		`hash768` without pre-computation (two 512 bit blocks)					
`Murax`	10,700	1380	1680	136	78.6	8.65	1.00
+ `SHA256`	576	2970	3990	99.7	5.78	1.00	13.6
		`hash1024` (three 512 bit blocks)					
`Murax`	15,600	1380	1680	136	114	10.3	1.00
+ `SHA256`	700	2970	3990	99.7	7.02	1.00	16.3

Table 2: Performance of the hardware module `SHA256` and comparisons of performing the SHA-256 compression function on different numbers of 512 bit blocks when called from the RISC-V software on a Murax SoC and on a Murax SoC with a `SHA256` accelerator. (all using the "fixed input length" optimization, i.e., no SHA-256 padding is computed).

the compression function on one 512 bit input block. It occupies about 1180 ALMs and uses about 1550 registers. The maximum frequency is reported by the synthesis tool as about 101MHz.

Table 2 also shows a comparison of computing one SHA-256 compression function call in software (design "`Murax`") with calling the hardware module from the software (design "`Murax` + `SHA256`"). Transferring data to the `SHA256` accelerator module and reading back the results contributes a significant overhead: The entire computation on a 512 bit input block (without SHA-256 padding computation) requires 253 cycles. This overhead is due to the simple bus structure of the Murax SoC; a more sophisticated bus (e.g., an AXI bus) may have a lower overhead — at a higher cost on resources. However, we achieve an almost 14.3× speedup over the software implementation of the SHA-256 compression function from the mbed TLS library which requires about 4950 cycles on the Murax SoC.

For one regular `hash768` function call, the SHA-256 compression function needs to be performed on two 512 bit blocks, while for one `hash1024` function call, three 512 bit blocks are needed. When the "pre-computation" optimization is enabled in the software, only one 512 bit block needs to be compressed in a `hash768` function call.

Table 6 shows the performance impact of the `SHA256` module on XMSS key generation, signing, and verification (designs "`Murax`" and "`Murax` + `SHA256`", including both "fixed input length" and "pre-computation" software optimizations). For these operations, the `SHA256` module accounts for an about 4.2× speedup for the key generation, signing and verification operations in the XMSS scheme.

To further accelerate the XMSS computations in an efficient way, in the following we first describe an XMSS-specific SHA-256 accelerator, which performs fixed-length SHA-256 padding and provides internal storage for one pre-computed state in hardware. Then, we describe how we use this XMSS-specific SHA-256 accelerator as building-block for larger hardware accelerators, an accelerator for WOTS-chain computations and an accelerator for XMSS-leaf generation including both WOTS-chain and L-tree computations.

4.2 XMSS-Specific SHA-256 Accelerator

In Section 3, we proposed two software optimizations for the XMSS scheme: "fixed input lengths" for accelerating SHA-256 computations on 768 bit and 1024 bit inputs and "pre-computation" for acceleration of the function prf(). For hardware acceleration, we introduced a general-purpose SHA-256 hardware module in Section 4.1, which replaces the SHA-256 compression function and thus naturally supports the "fixed input-lengths" optimization and the "pre-computation" optimization of the software implementation. However, the "pre-computation" optimization requires to transfer the pre-computed internal state from main memory to the `SHA256` module for each prf() computation. This data transfer introduces an overhead. To eliminate this communication overhead and as building block for the following hardware accelerator modules, we developed an XMSS-specific SHA-256 accelerator, the `SHA256XMSS` module. It has a similar functionality as the general `SHA256` module; however, the `SHA256XMSS` module supports both of the software optimizations: It only accepts complete input data blocks of size 768 bit or 1024 bit and adds the SHA-256 padding in hardware. In addition, it provides an internal 256 bit register for storing and replaying a pre-computed state.

Implementation: We used the `SHA256` module as basis for the implementation of the `SHA256XMSS` module. In order to handle larger input blocks, the `data_in` port of the `SHA256XMSS` module is 1024 bit wide. The `SHA256XMSS` module has an additional state machine to autonomously perform two or three compression-function iterations (depending on the input length). The state machine also takes care of appending the pre-computed SHA-256 padding to the input data before the last compression function computation. In addition to the `SHA256` module, the `SHA256XMSS` module has a `message_length` input signal that selects the required input length (low for 768 bit, high for 1024 bit), a `store_intermediate` input signal for requesting to store the result of the first compression-function iteration in the internal 256 bit register, and a `continue_intermediate` input signal for requesting to use the previously stored internal state instead of the first compression iteration. The pre-computation functionality can be enabled (marked as "`PRECOMP`" in the tables) or disabled at synthesis time in order to save hardware resources for a time-area trade-off.

The `SHA256XMSS` module is connected to the APB using the module `Apb3SHA256XMSS` as bridge, which works similarly to the `Apb3SHA256` module. It provides one 32 bit control register and a 1024 bit data register.

We replaced most of the SHA-256 function calls in the XMSS reference implementation with calls to the `SHA256XMSS` module. The software interface to `SHA256XMSS` is implemented in a function `sha256xmss`, which takes a `data_in` pointer to the input data block, a `message_length` flag, a `store_intermediate` flag, and a `continue_intermediate` flag as input and returns the 256 bit result in a `data_out` buffer.

When the function `sha256xmss` is called from the XMSS software, the 768 bit (256 bit if `PRECOMP` is used) or 1024 bit input data block (depending on the `message_length` flag) is sent to the `SHA256XMSS` module in 32 bit words via the APB. Then the software concatenates the inputs flags `message_length`, `store_intermediate`, `continue_intermediate`, and a high start bit to signal the start of the computation and writes them to the control register of the `Apb3SHA256XMSS` module, which dispatches them to the corresponding input ports of the `SHA256XMSS` module. On receiving a start signal, the `SHA256XMSS` module begins the hash computation. Once the hash computation is finished, the end of the computation is reported back to the software and the software reads the 256 bit result in 32 bit words via the APB.

Evaluation: Table 3 shows the performance, resource requirements, and maximum frequency of the `SHA256XMSS` module. When the pre-computation functionality is not

Design	Cycles	Area (ALM)	Reg.	Fmax (MHz)	Time (µs)	Time×Area (relative)	Speedup
			two 512 bit blocks				
`SHA256XMSS`	128	1410	2060	104	1.23	1.60	1.00
+ `PRECOMP`	64	1770	2320	95.3	0.672	1.00	1.83
			three 512 bit blocks				
`SHA256XMSS`	192	1410	2060	104	1.85	—	—
			`hash768`				
`Murax`	10,700	1380	1680	136	78.6	15.1	1.00
+ `SHA256XMSS`	512	3370	4920	103	4.97	1.76	15.8
+ `PRECOMP`	276	3540	5140	95.3	2.90	1.00	27.1
			`hash1024`				
`Murax`	15,600	1380	1680	136	114	9.73	1.00
+ `SHA256XMSS`	657	3370	4920	103	6.38	1.00	17.9

Table 3: Performance of hardware module `SHA256XMSS` and performance comparisons of SHA-256 computations for 768 bit and 1024 bit (functions `hash768` and `hash1024` when called from the RISC-V software on a Murax SoC and on a Murax SoC with a `SHA256XMSS` accelerator.

enabled, it requires 128 cycles and 192 cycles respectively (one cycle per round) for computing the hash digests for input messages of size 768 bit and 1024 bit. 1410 ALMs and 2060 registers are required according to the synthesis report; the synthesis tools report a maximum frequency of around 104 MHz. When the pre-computation functionality of the `SHA256XMSS` module is enabled, the cycle count for computing the hash digests for input messages of size 768 bit is halved, because only one 512 bit block needs to be compressed instead of two. However, storing the pre-computed state to achieve this speedup increases ALM and register requirements and causes a slight drop in the maximum frequency.

A comparison of the performance and resource requirements of the `hash768` and `hash1024` function calls for the plain `Murax` design with the "`Murax` + `SHA256XMSS`" design is also shown in Table 3. When the pre-computation functionality of the `SHA256XMSS` module is enabled, one `hash768` call within design "`Murax` + `SHA256XMSS` + `PRECOMP`" achieves a speedup of around 27.1× over the plain `Murax` design. However, the time-area product only improves by a factor of about 15.1×.

Table 6 shows the performance impact of the `SHA256XMSS` module on XMSS key generation, signing, and verification (`Murax` compared to "`Murax` + `SHA256XMSS` + `PRECOMP`"). For these operations, the `SHA256XMSS` module accounts for an about 5.4× speedup with pre-computation enabled.

4.3 WOTS-chain Accelerator

The `SHA256XMSS` module provides a significant speedup to the XMSS computations. However, since inputs and outputs need to be written to and read from the `SHA256XMSS` module frequently, the raw speedup of the SHA-256 accelerator cannot fully be exploited: It actually takes more time to send the inputs to and to read the results from the accelerator than the accelerator requires for the SHA-256 operations. This IO overhead can significantly be reduced by performing several SHA-256 operations consecutively in hardware. In this case, the hardware accelerator needs to be able to prepare some of the inputs by itself.

The WOTS chain computations are an ideal candidate for such an optimization, because the prf() computations performed in each chain step share a large amount of their inputs (only a few bytes are modified in the address fields for each prf() computation) and the f() computations use previous hash-function outputs. Therefore, we implemented the hardware module `Chain` as dedicated hardware accelerator for WOTS chain computations.

Implementation: One building block of the WOTS-chain hardware module `Chain` is the module `Step`, which implements the prf() and the keyed hash-function f() (see Section 2.1.2) in hardware. The `Step` module has an input port `key_in` for the 256 bit XMSS public seed, an input port `data_in` for the 256 bit data string and an input port `address_in` for the 256 bit address string. The hash result is returned over an 256 bit output port. Within the hardware module `Step`, two prf() computations and one f() computation are carried out one by one. These computations are realized by interfacing with a `SHA256XMSS` module described in Section 4.2. When module `Step` is triggered by a `start` signal, the input data (the address and the XMSS public key) for the first prf() computation is prepared and concatenated to the domain-separator of the prf(). Then the input data is delivered to the `SHA256XMSS` module.

When the `SHA256XMSS` module has finished the hash computations, the output is buffered in an 256 bit register `KEY` to be used later as the hash function key. At the same time, the second prf() computations begins (with a modified address as input) and its output is buffered in an 256 bit register `MASK` and the 256 bit input data of `Step` is XOR-ed to the value in register `MASK`. Then, the domain-separator for function f(), the value stored in register `KEY`, and the XOR-result are concatenated and sent to the `SHA256XMSS` module. The output of the `SHA256XMSS` module is returned as the result of the module `Step`.

The hardware chain module `Chain` repeatedly uses the module `Step`. The `Chain` module has two input ports `chain_start` and `chain_end`, which define the start and end step for the chain computation, e.g., 0 and $w - 1$ respectively for key generation. Each step in `Chain` uses a different address (chain step) and the output of the `Step` module from the previous step (or the value from port `data_in` of `Chain`) as input data. The result from the last step is returned via the output port of `Chain`.

The "pre-computation" optimization (see Section 3.2) can be enabled for the `Chain` module in the `SHA256XMSS` module before synthesis. In this case, the `store_intermediate` port of the `SHA256XMSS` module is set to high for the very first prf() computation of each WOTS chain computation. Therefore, the `SHA256XMSS` module stores the result of the first compression-function iteration in its internal 256 bit register. For all the following prf() computations, the input port `continue_intermediate` of the `SHA256XMSS` module is set to high. Therefore, the previously stored internal state is used instead of triggering the first compression iteration.

The `Chain` module is connected to the APB using a module called `Apb3Chain` as bridge, which works similarly to the APB bridge modules discussed before. It provides one 32 bit control register and three 256 bit data registers, which are directly connected to the 256 bit `data_in`, `key_in`, and `address_in` input ports of the `Chain` module. The output is directly read from the output port of the `Chain` module.

We integrated the call to the `Chain` module into the function `gen_chain` of the XMSS reference implementation, replacing the entire software implementation of the WOTS chain computation. The software interface is similar to the previously defined interfaces: The function `chain` has as arguments a `data` pointer to the input data string, a `key` pointer to the input key, and an `address` pointer to the address array for the inputs and a `data_out` pointer to the output buffer for the results. The `chain` function transfers the input data in 32 bit words via the APB bus to the `Chain` module, sends a start signal using the control register, and waits for the completion of the hardware module by polling the control register. Finally, it reads the 256 bit result in 32 bit words back via the APB and writes the 256 bit result into the `data_out` buffer.

Design	Cycles	Area (ALM)	Reg.	Fmax (MHz)	Time (µs)	Time×Area (relative)	Speedup
`Chain`	5880	2530	4120	96.0	61.2	1.41	1.00
+ `PRECOMP`	4030	2630	4380	91.9	43.8	1.00	1.40
`Murax`	530,000	1380	1680	136	3890	32.8	1.00
+ `Chain`	6830	4280	6730	87.5	78.1	1.31	49.8
+ `PRECOMP`	4980	4480	6900	93.5	53.3	1.00	73.0

Table 4: Performance of the hardware module `Chain` and comparisons of performing the `chain` function when called from the RISC-V software on a Murax SoC and on a Murax SoC with a `Chain` accelerator.

Evaluation: Table 4 shows performance, resource requirements, and maximum frequency of the `Chain` module. Enabling the "pre-computation" optimization ("+ `PRECOMP`") results in a 1.4× speedup for the chain computations in hardware.

A comparison between the pure software and the software/hardware performance of the function `chain` is also provided in Table 4. When `chain` is called in the design "`Murax` + `Chain` + `PRECOMP`", a speedup of around 73.0× is achieved compared to the pure software implementation using the `Murax` design. This speedup is much higher than the speedup we achieved by only accelerating single hash function calls (Table 3, "`Murax` + `SHA256XMSS` + `PRECOMP`"), because the communication overhead is significantly reduced.

Table 6 shows the performance impact of the `Chain` module on XMSS key generation, signing, and verification (`Murax` compared to "`Murax` + `Chain` + `PRECOMP`"). The acceleration of `Chain` module leads to a 5.3× speedup for key generation, a 5.8× speedup for signing and a 4× speedup for verification when the pre-computation functionality is enabled. To further accelerate the rest of the operations, i.e., the L-tree and Merkle tree operations, we can add a SHA256 accelerator to the Murax SoC, e.g., `SHA256` or `SHA256XMSS`. By adding a SHA accelerator, a much higher speedup can be achieved: For example, within the "`Murax` + `Chain` + `SHA256XMSS`" design, a 23.4× speedup is achieved for key generation with the pre-computation functionality enabled. For signing and verification, a 23.3× speedup and a 18.8× speedup is achieved respectively.

4.4 XMSS-leaf Generation Accelerator

When the `Chain` module is used to compute WOTS chains, the IO requirements are still quite high: For each of the l WOTS chains of one WOTS key generation, the 256 bit WOTS private key and a starting address need to be transferred to the `Chain` module and l WOTS chain public keys of 256 bit length need to be transferred back, although their inputs only differ in the address. To further reduce the IO requirements and thus increase the performance, we implemented a WOTS-leaf generation accelerator in hardware. The `Leaf` module requires a 256 bit address (leaf index), an 256 bit secret seed, and an 256 bit XMSS public seed as input. When the `Leaf` module finishes computation, an 256 bit leaf value (the L-tree root hash value) is returned as the output.

Implementation: The `Leaf` module uses two sub-modules: a `WOTS` module and an `L-tree` module. The `WOTS` module uses the `Chain` module described in the previous section to compute the WOTS chains and returns l 256 bit strings as the WOTS public key. Then, these l values are pairwise hashed together as described in Section 2.1.3 by the `L-tree` module. Finally, the output of the `L-tree` module (the root of the L-tree) is returned as the output of the `Leaf` module.

The `WOTS` module first computes the secret keys for each WOTS chain using a module

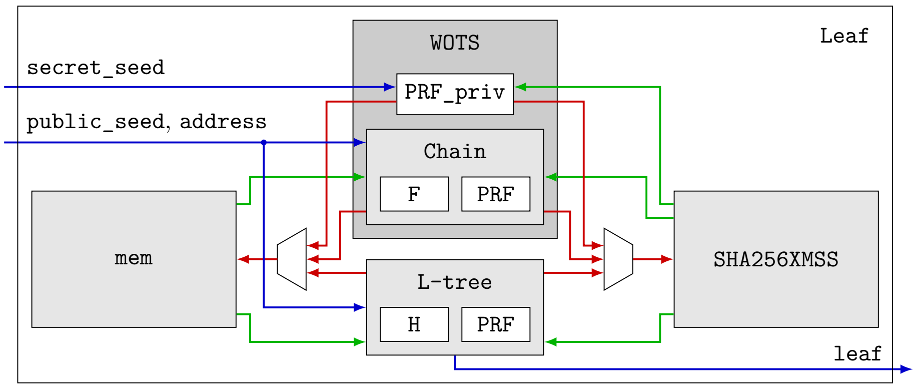

Figure 4: Diagram of the `Leaf` hardware module (control logic is not shown).

`PRF_priv` iteratively for l times. As opposed to the prf() computations during the WOTS chain, L-tree, and Merkle tree computations, the `PRF_priv` module takes a private, not a public seed as input. For each iteration, the corresponding address is computed and sent to the `PRF_priv` module as input as well. When the `PRF_priv` module has finished its computation, its output is written to a dual-port memory `mem`, which has depth l and width $8n$. Once the secret keys for the l WOTS chains have been computed and written to `mem`, the WOTS public key computation begins. This is done by using the `Chain` module described in Section 4.3. The l WOTS-chains are computed in sequence. First, a read request with the chain index as address is issued to `mem`, then the output of the memory is sent to the `data_in` port of the `Chain` module together with an address (with an individual chain index) and the XMSS public seed. The output of the `Chain` module is written back to `mem`, overwriting the previously stored secret chain key.

Once the WOTS public key computation is finished for all chains, the `L-tree` module begins its work. The building block of the `L-tree` module is a `RAND_HASH` module which implements the tree-hash function as described in Section 2.1.3. It takes in an 256 bit XMSS public seed, two 256 bit data strings, and a 256 bit address string as input and returns an 256 bit output. Within the hardware module `RAND_HASH`, three prf() and one h() computations are carried out in sequence using the modules `PRF` and `H`. The result generated by the first prf() computation is buffered as the 256 bit key while the results from the following prf() computations are buffered as the two 256 bit masks. The two 256 bit input data strings then get each XOR-ed with a mask and sent to the final h() computation together with the previously computed key and the address (tree layer, node index). The result of the h() computation is returned as the output of the module `RAND_HASH`.

The `L-tree` module constructs the nodes on the first level by first reading out two adjacent leaf nodes from the dual-port memory `mem` by issuing two simultaneous read requests to adjacent memory addresses. The memory outputs are sent to data input ports of the `RAND_HASH` module. Once `RAND_HASH` finishes computation, the result is written back to `mem` in order (starting from memory address 0). Since the L-tree is not a binary hash tree, it occasionally happens that there is a last node on one level that does not have a sibling node. This node is read out from `mem` and immediately written back to the next available memory address. This pattern of computation is repeated until the root of the L-tree is reached. This root is returned as the output of the `Leaf` module.

In order to minimize the resource usage of the `Leaf` module, all of the modules `PRF_priv`, `Chain`, and `RAND_HASH` are using one single `SHA256XMSS` module together. Figure 4 shows a diagram of the main building blocks of the `Leaf` module. The "pre-computation" optimization for the prf() computations again can be enabled for the `SHA256XMSS` module before synthesis.

Design	Cycles	Area (ALM)	Reg.	FMax (MHz)	Time (ms)	Time×Area (relative)	Speedup
`Leaf`	442×10^3	5130	8280	80.2	5.51	1.46	1.00
+ `PRECOMP`	296×10^3	5230	8540	71.8	4.12	1.00	1.34
`Murax`	27.7×10^6	1380	1680	136	203	17.6	1.00
+ `Leaf`	455×10^3	6830	10,900	78.4	5.80	1.43	35.0
+ `PRECOMP`	309×10^3	7010	11,100	79.9	3.87	1.00	52.5

Table 5: Performance of the hardware module `Leaf` and comparisons of performing the `leaf` function when called from the RISC-V software on a Murax SoC and on a Murax SoC with a `Leaf` accelerator.

The `Leaf` module is connected to the APB using a module called `Apb3Leaf` as bridge, which works similarly to the previously described APB bridge modules. It provides one 32 bit control register and three 256 bit data registers, which are directly connected to the 256 bit `secret_seed`, `public_seed`, and `address` ports of the `Leaf` module.

The `Leaf` module is called in the function `gen_leaf_wots` of the XMSS reference implementation. As interface to the `Leaf` module, we provide the software function `leaf`. This function has as arguments a `secret_seed` pointer to the secret key for `PRF_priv`, a `public_seed` pointer to the XMSS public seed, and a `address` pointer to the address array for the inputs and a pointer `data_out` for the result. The input values are sent to the `Leaf` module in 32 bit words via the APB bus and the start signal is sent via the control register. Then the software polls the control register until the computation is signaled as finished by the hardware module and reads back the result.

Evaluation: Table 5 shows performance, resource requirements, and maximum frequency of the `Leaf` module. Enabling the "pre-computation" optimization (design "`Leaf` + `PRECOMP`") gives a 1.3× speedup at the cost of only a small area overhead.

Comparisons of the software and hardware performance of one leaf computation are also provided in Table 5. The hardware implementation "`Murax` + `Leaf` + `PRECOMP`" achieves a 52.5× speedup over the pure software implementation using the plain `Murax` design.

Table 6 shows the performance impact of the `Leaf` module on XMSS key generation and signing (`Murax` and "`Murax` + `Leaf` + `PRECOMP`"). For the key-generation operation, the `Leaf` module accounts for an 47× speedup with `PRECOMP` enabled. The `Leaf` module is not useful for verification, since it computes a complete WOTS key generation, which is not required for verification. The BDS signing algorithm can make use of this accelerator; for signing the first 16 XMSS leaves, on average a 2.8× speedup is achieved. To further accelerate the rest of the hash-function-based operations, a `SHA256` or `SHA256XMSS` can be added to the Murax SoC in addition to the `Leaf` accelerator. In this case, a speedup of up to 52× is achieved with pre-computation optimization enabled for key generation and a speedup of over 16× for signing.

5 Performance Evaluation and Conclusion

Table 6 shows performance, resource requirements, and maximum frequency of different designs for the XMSS operations key generation, signing, and verification. Since the signature runtime of the BDS algorithm varies depending on the leaf index, we report the average timing for the first 16 signature leaves of the XMSS tree.

To accelerate the key generation, signing and verification operations in the XMSS scheme, the hardware accelerators we developed ("`SHA256`", "`SHA256XMSS`", "`Chain`" and

"`Leaf`") can be used separately or in a combined way for the acceleration, which leads to good speedups as we can see from Table 6. In general, from Table 6 we can see that the more computations we delegate to hardware accelerators, the more speedup we can achieve. However, at the same time, more overhead is introduced in the hardware resource usage, which is a trade-off users can choose depending on their needs. The best time-area product for key generation is achieved for "`Murax + Leaf`" with "`PRECOMP`" enabled.

The maximum frequency for the designs is heavily impacted by our hardware accelerators (which is accounted for in our speedup and time-area reports), dropping from 135 MHz down to as low as 71.6 MHz. If a high instruction throughput of the Murax SoC is required for an embedded application that is using our XMSS accelerators, a clock-frequency bridge between the APB and our accelerators might be necessary to enable independent clocks; however, this does not have an impact on the wall-clock speedup of our accelerators.

We measured a peak stack memory usage of 10.7 kB while the total memory usage is below 110 kB (including the binary code with stdlib and the stack; we do not use a heap). For a tree hight of $h = 10$, i.e., a maximum number of $2^h = 1024$ signatures per signature key, the time for XMSS key generation can be as short as only 4 s using our hardware accelerators. Even more signatures per signature key are conceivably possible using multi-tree XMSS. Signing takes only 21 ms and signature verification only 7 ms. This shows that using XMSS is very much feasible on a resource restricted embedded device like the RISC-V Murax SoC with the help of efficient dedicated hardware accelerators.

Acknowledgments. This work was supported in part by NSF grant 1716541. Part of the research was performed when the second author was affiliated at Fraunhofer Singapore.

References

[ACZ18] Dorian Amiet, Andreas Curiger, and Paul Zbinden. FPGA-based accelerator for post-quantum signature scheme SPHINCS-256. *IACR Transactions on Cryptographic Hardware and Embedded Systems – CHES 2018*, 2018(1):18–39, Feb. 2018. Open Access.

[BBD09] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors. *Post-Quantum Cryptography.* Springer, Heidelberg, 2009.

[BDH11] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS – a practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography – PQCrypto 2011*, volume 7071 of *LNCS*, pages 117–129. Springer, 2011. Second Version, IACR ePrint Report 2011/484.

[BDS08] Johannes Buchmann, Erik Dahmen, and Michael Schneider. Merkle tree traversal revisited. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography – PQCrypto 2008*, volume 5299 of *LNCS*, pages 63–78. Springer, 2008.

[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 Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 368–397. Springer, 2015.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Symposium on the Theory of Computing – STOC 1996*, pages 212–219. ACM, 1996.

Design	Cycles	Area (ALM)	BRAM (Blocks)	FMax (MHz)	Time	Time× Area (relative)	Speedup
				key generation			
`Murax`	28.3×10^9	1380	132	136	208 s	9.27	1.00
+ `SHA256`	4.87×10^9	2970	132	99.7	48.9 s	4.69	4.25
+ `SHA256XMSS`	5.58×10^9	3370	132	103	54.2 s	5.89	3.83
+ `PRECOMP`	3.65×10^9	3540	132	95.3	38.3 s	4.38	5.43
+ `Chain`	3.76×10^9	4280	132	87.5	43.0 s	5.94	4.83
+ `PRECOMP`	3.64×10^9	4480	132	93.5	38.9 s	5.63	5.34
+ `SHA256`	1.02×10^9	5910	132	90.9	11.2 s	2.14	18.5
+ `PRECOMP`	894×10^6	6100	132	92.9	9.63 s	1.89	21.6
+ `SHA256XMSS`	1.08×10^9	6360	132	91.3	11.8 s	2.42	17.6
+ `PRECOMP`	773×10^6	6650	132	86.8	8.90 s	1.91	23.4
+ `Leaf`	502×10^6	6830	145	78.4	6.41 s	1.41	32.5
+ `PRECOMP`	353×10^6	7010	145	79.9	4.42 s	1.00	47.0
+ `SHA256`	462×10^6	8380	145	71.6	6.45 s	1.75	32.2
+ `PRECOMP`	313×10^6	8560	145	77.9	4.01 s	1.11	51.8
+ `SHA256XMSS`	462×10^6	8770	145	76.7	6.03 s	1.71	34.5
+ `PRECOMP`	311×10^6	9150	145	77.9	3.99 s	1.18	52.1
				signing (average of the first 16 XMSS leaf signatures)			
`Murax`	64.8×10^6	1380	132	136	475 ms	5.03	1.00
+ `SHA256`	11.2×10^6	2970	132	99.7	112 ms	2.55	4.24
+ `SHA256XMSS`	12.8×10^6	3370	132	103	124 ms	3.20	3.83
+ `PRECOMP`	8.39×10^6	3540	132	95.3	88.0 ms	2.39	5.40
+ `Chain`	7.98×10^6	4280	132	87.5	91.2 ms	2.99	5.21
+ `PRECOMP`	7.69×10^6	4480	132	93.5	82.2 ms	2.82	5.78
+ `SHA256`	2.28×10^6	5910	132	90.9	25.1 ms	1.14	18.9
+ `PRECOMP`	1.99×10^6	6100	132	92.9	21.4 ms	1.00	22.2
+ `SHA256XMSS`	2.38×10^6	6360	132	91.3	26.1 ms	1.27	18.2
+ `PRECOMP`	1.77×10^6	6650	132	86.8	20.4 ms	1.04	23.3
+ `Leaf`	13.8×10^6	6830	145	78.4	176 ms	9.22	2.70
+ `PRECOMP`	13.6×10^6	7010	145	79.9	170 ms	9.09	2.80
+ `SHA256`	3.09×10^6	8380	145	71.6	43.2 ms	2.77	11.0
+ `PRECOMP`	2.82×10^6	8560	145	77.9	36.1 ms	2.37	13.2
+ `SHA256XMSS`	3.40×10^6	8770	145	76.7	44.4 ms	2.98	10.7
+ `PRECOMP`	2.28×10^6	9150	145	77.9	29.3 ms	2.05	16.2
				verification			
`Murax`	17.4×10^6	1380	132	136	127 ms	3.90	1.00
+ `SHA256`	2.99×10^6	2970	132	99.7	30.0 ms	1.98	4.25
+ `SHA256XMSS`	3.46×10^6	3370	132	103	33.7 ms	2.51	3.79
+ `PRECOMP`	2.24×10^6	3540	132	95.3	23.5 ms	1.85	5.41
+ `Chain`	3.05×10^6	4280	132	87.5	34.9 ms	3.31	3.65
+ `PRECOMP`	2.98×10^6	4480	132	93.5	31.9 ms	3.17	3.99
+ `SHA256`	764×10^3	5910	132	90.9	8.41 ms	1.10	15.2
+ `PRECOMP`	692×10^3	6100	132	92.9	7.44 ms	1.01	17.1
+ `SHA256XMSS`	856×10^3	6360	132	91.3	9.38 ms	1.32	13.6
+ `PRECOMP`	589×10^3	6650	132	86.8	6.78 ms	1.00	18.8

Table 6: Time and resource comparison for key generation, signing and verification on a Cyclone V (5CSEMA5F31C6) FPGA. "Time" is computed as quotient of "Cycles" and "FMax"; "Time×Area" is computed based on "Area" and "Time" relative to the time-area product of the respective most efficient design (gray rows); "Speedup" is computed based on "Time" relative to the respective `Murax` design.

[HBG$^+$18] Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. *RFC*, 8391:1–74, 2018.

[Hig18] Stacey Higginbotham. The rise of RISC - [opinion]. *IEEE Spectrum*, 55(8):18, Aug 2018.

[HRG11] Ekawat Homsirikamol, Marcin Rogawski, and Kris Gaj. Throughput vs. area trade-offs in high-speed architectures of five round 3 SHA-3 candidates implemented using Xilinx and Altera FPGAs. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *LNCS*, pages 491–506. Springer, 2011.

[Hül13] Andreas Hülsing. W-OTS+ – shorter signatures for hash-based signature schemes. In Amr Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *LNCS*, pages 173–188. Springer, 2013.

[MCF18] David McGrew, Michael Curcio, and Scott Fluhrer. Hash-based signatures. *cfrg*, draft-mcgrew-hash-sigs-1:1–60, 2018.

[Mer90] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO 1989*, volume 435 of *LNCS*, pages 218–238. Springer, 1990.

[Mer18] Rick Merritt. Microsoft and Google planning silicon-level security. *EE Times Asia*, Aug. 2018. URL: `https://www.eetasia.com/news/article/18082202-microsoft-and-google-planning-silicon-level-security`.

[NIS12] NIST. *FIPS PUB 180-4: Secure Hash Standard.* National Institute of Standards and Technology, 2012.

[NIS13] NIST. *FIPS PUB 186-4: Digital Signature Standard.* National Institute of Standards and Technology, 2013.

[SH10] Abdulhadi Shoufan and Nico Huber. A fast hash tree generator for Merkle signature scheme. In *International Symposium on Circuits and Systems – ISCAS 2010*, pages 3945–3948. IEEE, 2010.

[SHM11] Abdulhadi Shoufan, Nico Huber, and H. Gregor Molter. A novel cryptoprocessor architecture for chained Merkle signature scheme. *Microprocessors and Microsystems*, 35(1):34–47, 2011.

[Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Foundations of Computer Science – FOCS '94*, pages 124–134. IEEE, 1994.

[Sho99] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[Sho10] Abdulhadi Shoufan. An FPGA accelerator for hash tree generation in the Merkle signature scheme. In Phaophak Sirisuk, Fearghal Morgan, Tarek El-Ghazawi, and Hideharu Amano, editors, *Applied Reconfigurable Computing – ARC 2010*, volume 5992 of *LNCS*, pages 145–156. Springer, 2010.