High-speed Instruction-set Coprocessor for Lattice-based Key Encapsulation Mechanism: Saber in Hardware

Sujoy Sinha Roy and Andrea Basso

University of Birmingham
Edgbaston B15 2TT, United Kingdom
s.sinharoy@cs.bham.ac.uk,a.basso@cs.bham.ac.uk

Abstract. In this paper, we present an instruction set coprocessor architecture for the module lattice-based post-quantum key encapsulation (KEM) scheme Saber. To achieve fast computation time, the architecture is a *full-hardware*, i.e., all the building blocks (including CCA transformations) are implemented in the hardware. Since polynomial multiplication plays a performance-critical role in the module and ideal lattice-based public-key cryptography, a parallel polynomial multiplier architecture is proposed that overcomes memory access bottlenecks and results in a highly parallel yet simple and easy-to-scale design. Besides optimizing polynomial multiplication, we make important design decisions and perform architectural optimizations to reduce overall cycle counts as well as improve resource utilization.
For the module dimension 3 (security comparable to AES-192), the coprocessor computes CCA key generation, encapsulation, and decapsulation in only 5,453, 6,618 and 8,034 cycles respectively. On a Xilinx UltraScale+ XCZU9EG-2FFVB1156 FPGA, the entire instruction set coprocessor architecture runs at 250 MHz clock frequency and consumes 23,708 LUTs, 9764 FFs, and 2 BRAM tiles (including 5124 LUTs and 3070 FFs for the Keccak core).

Keywords: Lattice-based Cryptography · Post-Quantum Cryptography · Hardware Implementation · Saber KEM · High-speed Instruction-set Architecture

1 Introduction

In October 2019, Google's 54-qubit quantum processor 'Sycamore' completed a task in 200 seconds, the equivalent of which can be computed in 10,000 years using a state-of-the-art supercomputer [Aea19]. To break our present-day public-key cryptographic primitives, namely the RSA and Elliptic Curve cryptosystems, Shor's algorithm [Sho97] needs a significantly more powerful quantum computer. However, several quantum computing scientists anticipate that powerful enough quantum computers to break these cryptosystems will be feasible in the next 15 to 20 years. Post-quantum cryptography is a branch of cryptography that focuses on designing quantum attack resistant public-key primitives and analyzing their securities. Existing post-quantum public-key cryptographic primitives have been built based on different problems that are presumed to be computationally infeasible for both present-day as well as quantum computers. In 2017 the National Institute of Standards and Technology (NIST) called for the standardization of post-quantum public-key algorithms. The majority of the candidate submissions use computationally infeasible lattice-problems. One such candidate scheme is Saber [DKRV19], which is a Chosen Ciphertext Attack (CCA) resistant module lattice-based key encapsulation mechanism (KEM). It is one of the nine lattice-based public-key encryption or encapsulation schemes that has proceeded to the second round of NIST's standardization project. Saber is based on the Module Learning With Rounding (MLWR) problem [BPR12] and it uses power-of-two moduli to achieve flexibility, simplicity, high security and efficiency [DKRV19].

It is well-known that in ideal or module lattice-based public-key cryptography, the performance of polynomial multiplication plays a big role in the overall performance of the cryptographic primitive. Number Theoretic Transform (NTT), which is a generalization of Fast Fourier Transform (FFT), has the asymptotically fastest time complexity $O(n \log n)$. However, the NTT requires the ciphertext modulus to be a prime. To achieve computational efficiency, several lattice-based schemes [ADPS16, BDK+18, ABB+19] use NTT-friendly parameter sets. Efficient hardware and software implementations of NTT-based polynomial multiplications [PG14, RVM+14a, ADPS16, BDK+18] have been reported in the literature. However, Saber uses power-of-two moduli, thus making it devoid of asymptotically fastest NTT-based polynomial multiplication. This non-typical parameter set in Saber makes its implementation an interesting as well as challenging research topic. Efficient software implementations of Saber have been reported in [DKRV19, KMRV18, Roy19, BMKV20]. However, the only published hardware implementation of Saber are [MTK+20, DFAG19] and both use HW/SW codesign. While HW/SW codesign has its benefits, such as flexibility and shorter design cycle, a full-hardware (i.e., including all building blocks) implementation of Saber can offer better latency and throughput. At the same time, implementing such an accelerator is a challenging research topic as it requires making careful design decisions after taking into account both algorithmic and architectural alternatives for the internal building blocks and their interaction in the protocol level.

Contributions

In this paper, we present an instruction-set coprocessor architecture for the module lattice-based post-quantum key encapsulation scheme Saber [DKRV19]. The architecture implements all the building blocks in the hardware thus making it one of the fastest implementations of Saber. In particular, we make the following contributions:

1. Since polynomial multiplication plays a central role in Saber, we analyze different algorithmic alternatives for implementing high-speed polynomial multiplication in hardware. By taking into account both computation and memory access overheads, we use a simple yet parallel and hardware-friendly polynomial multiplication algorithm targeting the parameter set of Saber.

2. We take advantage of the power-of-two moduli and small secret in Saber and implement a custom architecture for the polynomial multiplication algorithm. Additionally, we perform architectural optimizations to reduce both cycle, logic and register counts. The designed polynomial multiplier architecture is massively parallel and doesn't suffer from memory-access bottlenecks. With this multiplier, one polynomial multiplication operation requires only 256 cycles (excluding the overhead of operand loading). To compare with, the polynomial multiplier architecture by Roy et al. [RVM+14b] uses asymptotically fastest NTT-based polynomial multiplication and requires around 5,000 cycles to compute one polynomial multiplication.

3. The polynomial architecture is easy to scale to meet different performance-area trade-offs. We further show how to pipeline the polynomial multiplier architecture and achieve higher clock frequency with a negligible increase in the latency.

4. Several arithmetic operations in Saber use non-multiple of 8-bit operands, making their resource-shared and optimized hardware implementation challenging. We analyze these building blocks and perform optimizations to reduce both cycle and area counts.

5. The optimized building blocks are integrated to realize an instruction-set coprocessor architecture that computes all KEM operations, namely key generation, encapsulation and decapsulation in the hardware. Since several existing software implementations [KRSS19] of lattice-based KEMs reported that Keccak-based pseudo-random number generation takes the lion's share of the overall computation time, we used the high-performance Keccak core that was developed by the Keccak team [Tea19]. The unified architecture computes CCA-secure Saber key generation, encapsulation and decapsulation in only 5,453, 6,618 and 8,034 cycles respectively for the parameter set with security similar to AES-192.

6. Our design methodology is generic and hence can be followed to design instruction-set coprocessors for other lattice-based schemes. We will make the HDL source codes available to fellow researchers once the paper gets accepted.

2 Preliminaries

2.1 Notation

In this section, we introduce the notation used throughout the paper. Let p and q be two powers of 2, i.e. $p = 2^{\varepsilon_p}$ and $q = 2^{\varepsilon_q}$. We denote with $\mathbb{Z}_q$ the ring of integers modulo q. Define then the ring of polynomials $\mathcal{R}_p = \mathbb{Z}_p[x]/\langle x^N + 1\rangle$, for some integer N, and the corresponding $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^N + 1\rangle$. We write $a[i]$ to denote the ith coefficient of polynomial $a(x)$. A vector is represented in bold, such as $\mathbf{a}$. Let the operator $\lfloor \cdot \rceil$ denote rounding, i.e. $\lfloor a \rceil = \lfloor a + \frac{1}{2} \rfloor$. This can be extended to polynomials coefficient-wise.
Let β_μ denote a centered binomial distribution with even parameter μ. The distribution takes on values in the range $[-\mu/2, \mu/2]$ with probability

$$p(x) = \frac{\mu!}{(\mu/2 + x)!(\mu/2 - x)!} 2^{-\mu}.$$

We write $x \leftarrow \beta_\mu$ to denote x randomly sampled from a β_μ distribution. Given a set S, we write $x \leftarrow \mathcal{U}(S)$ for x uniformly randomly selected from S. In a straightforward way, these notations can be applied to a polynomial or a vector or a matrix.

2.2 Saber

Saber [DKRV19] is a IND-CCA secure Key Encapsulation Mechanism (KEM) that relies on the hardness of the module variant of the Learning With Rounding (Mod-LWR) problem [BPR12]. A Mod-LWR sample is given by

$$\left(\boldsymbol{a}, b = \left\lfloor \frac{p}{q}(\boldsymbol{a}^T \boldsymbol{s}) \right\rceil \right) \in R_q^{l \times 1} \times R_p \tag{1}$$

where $\boldsymbol{a} \leftarrow \mathcal{U}(R_q^{l\times 1})$, the secret $\boldsymbol{s} \leftarrow \beta_\mu(R_q^{l\times 1})$ is generated from a centred binomial distribution with parameter μ and is fixed, and the moduli $p < q$. The decisional variant of the problem asks to distinguish between Mod-LWR samples and uniformly random samples $\in R_q^{l\times 1} \times R_p$. This Mod-LWR problem is presumed to be computationally infeasible, both on classical and quantum computers.

Saber [DKRV19] uses the Mod-LWR problem with both p and q power-of-two to construct a Chosen Plaintext Attack (CPA) secure public-key encryption scheme. Following that, a CCA-secure Saber KEM is realized using a post-quantum variant of the Fujisaki-Okamoto transformation [HHK17]. In the following, we describe the algorithms used in CPA-secure 'Saber Public Key Encryption' (Alg. 1, 2, 3) and CCA-secure 'Saber Key Encapsulation' (Alg. 4, 5, 6). The function `gen` is a pseudorandom number generator based on SHAKE-128 [20115] and $\mathcal{G} : \{0,1\}^* \rightarrow \{0,1\}^{l\times n}$ and $\mathcal{H} : \{0,1\}^* \rightarrow \{0,1\}^n$ are hash functions SHA3-512 and SHA3-256 respectively, standardized in FIPS 202 [20115]. We refer to the original paper [DKRV19] for further information on the matter.

Algorithm 1 `Saber.PKE.KeyGen()` [DKRV19]

$seed_{\boldsymbol{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{A} = \texttt{gen}(\text{seed}_{\boldsymbol{A}}) \in R_q^{l\times l}$
$r = \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{s} = \beta_\mu(R_q^{l\times 1}; r)$
$\boldsymbol{b} = ((\boldsymbol{A}^T\boldsymbol{s} + \boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l\times 1}$
return $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}), sk := (\boldsymbol{s}))$

Algorithm 2 `Saber.PKE.Enc`$(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), m \in R_2; r)$ [DKRV19]

$\boldsymbol{A} = \texttt{gen}(\text{seed}_{\boldsymbol{A}}) \in R_q^{l\times l}$
if r *is not specified* **then**
  $r = \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{s'} = \beta_\mu(R_q^{l\times 1}; r)$
$\boldsymbol{b'} = ((\boldsymbol{A}\boldsymbol{s'} + \boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l\times 1}$
$v' = \boldsymbol{b}^T(\boldsymbol{s'} \bmod p) \in R_p$
$c_m = (v' + h_1 - 2^{\epsilon_p - 1}m \bmod p) \gg (\epsilon_p - \epsilon_T) \in R_T$
return $c := (c_m, \boldsymbol{b'})$

Algorithm 3 `Saber.PKE.Dec`$(sk = \boldsymbol{s}, c = (c_m, \boldsymbol{b'}))$ [DKRV19]

$v = \boldsymbol{b'}^T(\boldsymbol{s} \bmod p) \in R_p$
$m' = ((v - 2^{\epsilon_p - \epsilon_T}c_m + h_2) \bmod p) \gg (\epsilon_p - 1) \in R_2$
return m'

Algorithm 4 `Saber.KEM.KeyGen()` [DKRV19]

$(seed_{\boldsymbol{A}}, \boldsymbol{b}, \boldsymbol{s}) = $ `Saber.PKE.KeyGen()`
$pk = (seed_{\boldsymbol{A}}, \boldsymbol{b})$
$pkh = \mathcal{F}(pk)$
$z = \mathcal{U}(\{0,1\}^{256})$
return $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}), sk := (\boldsymbol{s}, z, pkh))$

Algorithm 5 `Saber.KEM.Encaps`$(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}))$ [DKRV19]

$m \leftarrow \mathcal{U}(\{0,1\}^{256})$
$(\hat{K}, r) = \mathcal{G}(\mathcal{F}(pk), m)$
$c = $ `Saber.PKE.Enc`$(pk, m; r)$
$K = \mathcal{H}(\hat{K}, c)$
return (c, K)

Algorithm 6 $\texttt{Saber.KEM.Decaps}(sk = (\boldsymbol{s}, z, pkh), pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), c)$ [DKRV19]

$m' = \texttt{Saber.PKE.Dec}(\boldsymbol{s}, c)$
$(\hat{K}', r') = \mathcal{G}(pkh, m')$
$c' = \texttt{Saber.PKE.Enc}(pk, m'; r')$
if $c = c'$ **then**
  return $K = \mathcal{H}(\hat{K}', c)$
else
  return $K = \mathcal{H}(z, c)$

Parameters Saber defines three sets of parameters which match NIST security levels 1, 3 and 5. They have been called LightSaber, Saber and FireSaber. All three levels use polynomial degree $N = 256$, and moduli $q = 2^{13}$ and $p = 2^{10}$. The three variants differ in the module dimension, the binomial distribution parameter and the message space. Namely, LightSaber uses module dimension 2, secrets sampled from $[-5, 5]$ and $t = 2^2$; Saber uses module dimension 3, secrets sampled from $[-4, 4]$ and $t = 2^3$; and FireSaber upgrades the parameters to module dimension 4, secrets sampled from $[-3, 3]$ and $t = 2^5$.

3 Design Decisions

In the previous section, we outlined the operations that are computed during key generation, encapsulation and decapsulation. These computations are composed of several elementary operations such as hashing, pseudo-random number generation, polynomial addition and multiplication, rounding, etc.

Since Saber uses power-of-two moduli p and q, all modulus reductions are free in hardware. Additionally, the rounding operation is cheap as it comprises only of additions, modulo reductions and finally bit selection. In the following subsections, we describe various design choices and the design decisions that we make while implementing Saber on hardware platforms. Our aim is to achieve both high speed and flexibility for the KEM operations.

3.1 High-level Architecture

There are two general methodologies to implement a computation-intensive cryptographic algorithm in hardware, namely HW/SW codesign, and full-HW design. While a HW/SW codesign strategy offers a shorter design cycle and higher flexibility, it may not result in the best performance. On the other hand, designing a full-HW architecture, i.e., with all the building blocks in the hardware, can offer significant speedup over a HW/SW codesign architecture. However, the HW-only design methodology demands significant implementation effort (hence a longer design cycle) and may result in diminished flexibility. In this paper, we target speed and hence we opt for a full-hardware implementation with all building blocks residing in the hardware. At the same time we try to make design decisions such that the hardware remains flexible to a great extent (e.g., can compute all of key generation, encapsulation and decapsulation for multiple Saber parameter-sets).

When a HW-only implementation is considered, a design option is to cascade different building blocks in the data-path, if required in multiple parallel instances, following the standard data-flow model. However, this approach results in large area and demands customized data-paths for different protocol-level operations namely, key-generation, encapsulation and decapsulation. Additionally, such an architecture becomes somewhat inflexible to different parameter-sets [GFS+12]. Hence, we do not follow this design methodology in this work.

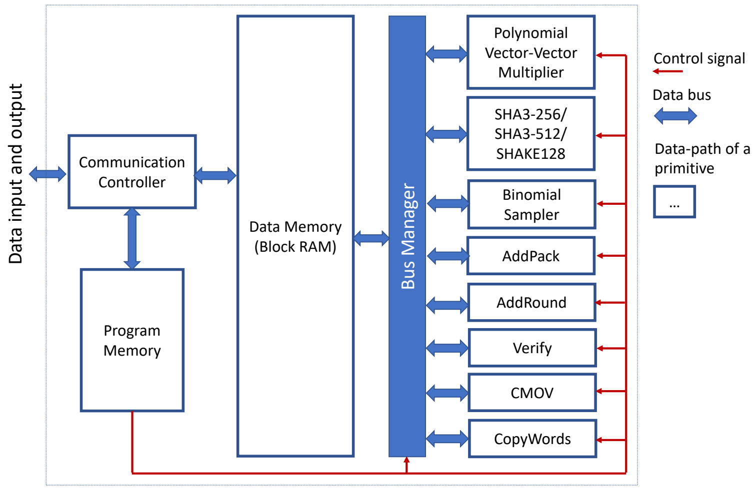


Figure 1: Instruction-set Hardware Architecture of Saber

To achieve programmability and flexibility, we realize an *instruction-set coprocessor architecture* for Saber. The advantages of this design strategy are: instruction-level flexibility and modularity, ease to add new instructions or modify them, and above all a unified architecture that can be used for multiple tasks. We analyzed the SW implementation of Saber [DKRV19] and identified the high-level instructions that are needed to support all the CCA-secure KEM routines, namely the key generation, encapsulation, and decapsulation. A high-level architecture diagram of the instruction-set coprocessor architecture (ISA) is shown in Fig. 1.

We would like to remark that, although in this work we implement the architecture targeting only Saber KEM (as a case study), the implementation strategy is quite generic in nature and hence can be followed to implement other lattice-based public-key schemes in the hardware. In the following sections, we describe the architectures for the building blocks.

3.2 SHA3-256/SHA3-512/SHAKE-128

As shown in Alg. 4, 5, and 6, Saber uses the hash functions SHA3-256 and SHA3-512 that were standardized in FIPS 202 [20115]. Moreover, to generate pseudorandom numbers, the extendable output function SHAKE-128, also standardized in FIPS 202 is used. Since, all of these functions use the Keccak sponge function [20115], we implement the block `SHA3-256/SHA3-512/SHAKE-128` in Fig. 1 as a wrapper around a *single* Keccak core.

In this paragraph, we justify why we use a single Keccak core in our implementation. Software benchmarking [KRSS19] of several lattice-based KEM schemes have reported that 50-70% of the overall computation time is spent in executing the Keccak function, thus making it the most performance-critical component. On software platforms with Single Instruction Multiple Data (SIMD) processors, such as Intel AVX2, the overhead pseudo-random number generation is reduced in Kyber KEM [BDK+18] (which is also based on module lattices) by using a vectorized implementation (factor 4) of Keccak. However, the Saber algorithm [DKRV19] calls the Keccak operations in a serial manner and thus a single call to a Saber KEM operation cannot leverage from a vectorized implementation of Keccak on software platforms with SIMD.

This serial execution of Keccak in the Saber algorithm does not cause concern as Keccak is very efficient [20115] on hardware platforms. In this work, we use the open-source high-speed implementation of the Keccak core that was designed by the Keccak

Team [Tea19]. This high-speed implementation of Keccak computes 'state-permutations' at a gap of only 28 cycles, thus generating 1,344 bits of pseudo-random string after every 28 cycles during the extraction-phase. Furthermore, we observed that one instance of the Keccak core consumes around 5K LUTs and 3K registers which are nearly 21% and 30% of the overall area in our implementation. The area consumption results indicate that instantiating multiple high-speed Keccak cores in the hardware would make the implementation area-expensive. Additionally, as the Keccak core is already very fast, the use of multiple such cores in parallel would help little in improving the speed. Due to these reasons, we instantiate only one high-speed Keccak core in the hardware. Furthermore, the serial use of the Keccak core makes our implementation simpler.

3.3 Data Memory

In the instruction-set architecture (Fig. 1), the building blocks read their operand-data from the data memory, and write their results back to the data memory. The data memory is of size 8KB such that all the parameter sets of Saber can be computed, and it is implemented using Block RAM tiles. An important design parameter is the word-size of the memory. We set the word-size to 64-bit as the high-speed Keccak core reads/writes data in 64-bit words. Additionally, when we consider integration of the instruction-set coprocessor architecture to a host computer (32-bit or 64-bit), the use of a 64-bit data-memory simplifies the data transfer protocol between the two sides. All the remaining compute blocks in Fig. 1 have been optimized to use 64-bit data read/write operations efficiently.

3.4 Binomial Sampling

A binomial sampler with parameter μ computes a sample from a μ-bit pseudo-random input string, say $r[\mu - 1 : 0]$, by subtracting the Hamming weight of the least-significant $\mu/2$ bits from the Hamming weight of the most-significant $\mu/2$ bits, i.e., by computing $\mathtt{HW}(r[\mu - 1 : \mu/2]) - \mathtt{HW}(r[\mu/2 - 1 : 0])$, where $\mathtt{HW}()$ stands for the Hamming weight.

In Saber, the secret coefficients are drawn from a centered binomial distribution with the parameter $\mu = 10$, 8, and 6 for LightSaber, Saber, and FireSaber respectively [DKRV19]. Hence, the secret coefficients are in $[-5, 5]$ for LightSaber, $[-4, 4]$ for Saber, and $[-3, 3]$ for FireSaber. As μ is small in all the variants of Saber, the sampler requires simple bit manipulations. In our architecture, the sampler is a combinational block that directly maps pseudo-random bits from an input buffer to a sample value.

For all the variants of Saber, a sample is represented as a 4-bit signed-magnitude number (pair of sign and an absolute value) in our implementation. Note that existing software implementations of Saber [DKRV19, KMRV18, Roy19] use the two's complement number system to represent the samples in the C data type `uint16_t`. The use of '4-bit signed-magnitude' representation simplifies the hardware architecture as we can store 16 such samples easily in a 64-bit word of the data memory. Additionally, in Sec. 3.5.1 we show that this representation simplifies the polynomial multiplier.

For Saber, since $\mu = 8$ divides the word-length of the data memory, two 64-bit pseudo-random words are read from the memory, then they are stored in a 128-bit buffer register, then 16 samples are generated in parallel and they are stored in an output buffer register of length 64-bit, and finally the output buffer is written to the data memory.

3.5 Polynomial Multiplication

In ideal and module lattice-based cryptosystems, the performance of polynomial multiplication plays a critical role. Since Saber uses power-of-two moduli $p = 2^{10}$ and $q = 2^{13}$, it is devoid of the asymptotically fastest Number Theoretic Transform (NTT)-based polynomial multiplication. Software implementations [DKSRV18] of Saber have used the

Toom-Cook polynomial multiplication algorithm [Knu97] which is a generic algorithm and is asymptotically the second best after the NTT-based polynomial multiplication. A recent TCHES-2020 paper [BMKV20] by Bermudo Mera, Karmakar, and Verbauwhede proposes arithmetic optimization techniques to speed up the Toom-Cook polynomial multiplication algorithm targeting software platforms. Hardware implementation of the Toom-Cook multiplication by Bermudo Mera, Turan, Karmakar, Roy, and Verbauwhede [MTK+20] describes the challenges in implementing the recursive function calls in the hardware and proposes efficient architectures.

In this work, we use the quadratic-complexity schoolbook polynomial multiplication algorithm and realize a simple, yet parallel and very fast polynomial multiplier architecture. Since the polynomials in Saber are only of degree 256, the asymptotic inferiority of the quadratic-complexity algorithm is outweighed by its simplicity and amiability to parallelization. The schoolbook polynomial multiplication algorithm for polynomials of degree N is described in Alg. 7.

Algorithm 7 Schoolbook polynomial multiplication.

Input: Two polynomials $a(x)$ and $b(x)$ in $\mathbb{Z}_q/\langle x^N+1\rangle$.
Output: The product $a(x)\cdot b(x)$ in $\mathbb{Z}_q/\langle x^N+1\rangle$.
1: $acc(x) \leftarrow 0$.
2: **for** $i=0$; $i<N$; $i=i+1$ **do**
3:     **for** $j=0$; $j<N$; $j=j+1$ **do**
4:         $acc[j] = acc[j] + b[j]\cdot a[i] \bmod \mathbb{Z}_q$
5:     **end for**
6:     $b = b\cdot x \bmod x^N+1$.
7: **end for**
8: **return** acc.

In line number 1, an accumulator which consists of N registers to contain the result of the polynomial multiplication is initialized to zero. Next, in line 4 inside the nested loops, the i-th coefficient of $a(x)$ is multiplied to the j-th coefficient of $b(x)$ and then the result of the multiplication is accumulated in the j-th register of the accumulator acc. This operation consists of an integer multiplication, followed by modular reduction and finally a modular addition. During a schoolbook multiplication, one polynomial needs to be rotated inside the outermost loop. In Alg. 7, $b(x)$ is rotated by multiplying it by x in $\mathcal{R}_q$.

Although the schoolbook polynomial multiplication algorithm looks rather simple, its efficient implementation on a hardware platform requires making wise design decisions as well as design-space exploration. In the remaining part of this section, we describe the optimizations that we perform, the implementation strategies that we follow, and their advantages (and a few drawbacks) over alternative design strategies.

3.5.1 Optimization of coefficient-wise modular multiplier

In the Saber protocol [DKSRV18], polynomial multiplications are computed between public polynomials in $\mathcal{R}_q$ or $\mathcal{R}_p$ and secret polynomials. For simplicity, we will denote the former by $a(x)$ and the latter by $s(x)$. As mentioned in section 2.2, the coefficients of the secret polynomial s are randomly generated from a binomial distribution and—depending on the version of Saber—they are contained in the interval $[-3,3]$, $[-4,4]$ or $[-5,5]$, hence small. Additionally, since both p and q are power-of-two in Saber, modular reduction by p or q are free.

We exploit 'short' secret-size and reduction-free modular multiplication to optimize the coefficient-wise multiplications in Alg. 7. A coefficient-wise multiplier is implemented using simple shift and add operations, as shown in Algorithm 8, instead of requiring a true

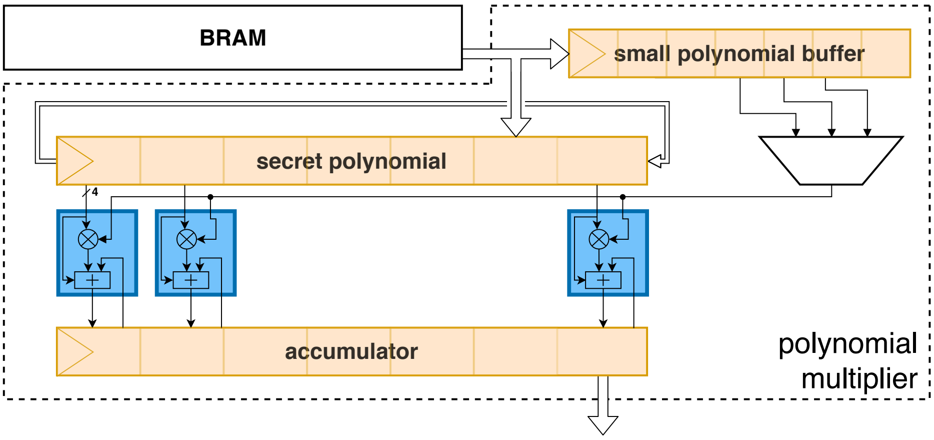


Figure 2: The polynomial multiplier architecture. Blue blocks denote processing units, orange blocks are registers and wide arrows represent 64-bit input/output to the multiplier.

integer multiplier. We compute up to times-five multiplication to fully support all variants of Saber. Implementations targeting exclusively the regular version of Saber or FireSaber can obtain slight gains in area consumption by avoiding unnecessary computations at this stage. Note that we represent the coefficients of s with a sign-magnitude system (Sec. 3.4) and perform multiplications only with their absolute values. The accumulator is then updated by adding or subtracting the results depending on the sign-bit of the coefficient of s. Furthermore, since the modulus q is a power of 2 and the coefficients of a are represented as 13-bit numbers, modulus reduction is implicit and requires no additional operation. In hardware, a bit-parallel combinatorial circuit is used to implement Alg. 8 and hence the multiplier is constant-time.

Algorithm 8 Coeffient-wise shift-and-add multiplier.

Input: a_i: 13-bit number, s_j: 3-bit number with $0 \leq s_j \leq 5$.
Output: $a_i \cdot s_j$ modulo $q = 2^{13}$.

$r_0 \leftarrow 0$,
$r_1 \leftarrow a_i$,
$r_2 \leftarrow a_i \ll 1$,
$r_3 \leftarrow a_i + (a_i \ll 1)$,
$r_4 \leftarrow a_i \ll 2$,
$r_5 \leftarrow a_i + (a_i \ll 2)$,
return r_k, where $k = s_j$.

3.5.2 Parallel polynomial multiplier architecture

Fig. 2 shows the polynomial multiplier architecture that implements a parallelized version of the schoolbook multiplication described in Algorithm 7. Since the coefficient-wise modular multiplication has a small area (Sec. 3.5.1), the schoolbook polynomial multiplier architecture instantiates several multiply-and-accumulate (MAC) units in parallel to compute line 4 of Alg. 7 in parallel. For example, by instantiating 256 MAC units in parallel, the innermost loop in Alg. 7 can be computed in one cycle, thus requiring only 256 cycles to compute one polynomial multiplication for $N = 256$.

The overhead of memory access during polynomial multiplication plays a critical role in lattice-based cryptography (e.g., [RVM[+]14a]) and could hinder or complicate logic-level parallel processing. For example, in NTT-based polynomial multiplication, the pattern of memory access changes with the iterations. Hence, special memory management technique is required to reduce the overhead of memory access [RVM+14a]. Additionally, the 'complicated' memory access pattern of NTT makes its parallel implementation rather challenging as special care must be taken to eliminate memory access conflicts [RJV+18, RTJ+19].

The schoolbook multiplication algorithm has a regular and simple data read/write pattern. To attain maximum parallelism in data read/write, and to avoid the above-mentioned memory-access bottlenecks, we store the entire secret polynomial $s(x)$ in a shift register (composed of flip-flops) (Fig. 2). It is well-known that all the bits of a register can be accessed simultaneously on a hardware platform. At the beginning of a polynomial multiplication, $s(x)$ is read from the data memory (block RAM) and then loaded into the shift register. That allows the architecture to access all the coefficients of $s(x)$ simultaneously.

As shown in Alg. 7, only one coefficient of the other polynomial $a(x)$ is required at a time to compute the scalar multiplication $s(x) \cdot a[i]$. Hence, it is not necessary to store the entire $a(x)$ polynomial in a register. The 'coefficient selector' block in Fig. 2 provides the required coefficient of $a(x)$ during the multiplication $s(x) \cdot a[i]$ by the parallel MAC cores. In the next subsection we describe, how the 'coefficient selector' block is designed for this purpose.

After the multiplication $s(x) \cdot a[i]$, $s(x)$ needs to be multiplied by x. This operation is a simple nega-cyclic left-shift operation that moves each coefficient from positions i to position $i+1$ and sends the 256th coefficient to the first position after a modular subtraction from zero. This nega-cyclic rotation happens because the reduction-polynomial is $x^{256}+1$. In our implementation, the binomial distributed coefficients of $s(x)$ are represented in the signed magnitude system. Hence, the sign of the 256th coefficient is just flipped instead of computing a true subtraction operation.

3.5.3 Data loading

In the previous subsection, we described a fast polynomial multiplier core for Saber. In practice, we can leverage from its speed if we can load the operands and also read the result of a polynomial multiplication in minimum cycle count. In this section we describe how we design a fast data exchange interface between the data-memory (block RAM in Fig. 1) and the polynomial multiplication core (Fig. 2).

The public polynomial $a(x)$ lives in the field $\mathcal{R}_\ell = \mathbb{Z}_\ell[x]/(x^n+1)$, where either $\ell = q = 2^{13}$ or $\ell = p = 2^{10}$. In the former case, the coefficients of $a(x)$ are 13-bits long and they are output from the SHAKE-128 block by expanding a seed. The output of the SHAKE-128 implementation that we use is a continuous stream of 64-bit words. Hence, an entire polynomial in $\mathcal{R}_q$ is stored in data-memory (block RAM) as a continuous string of length $256 \cdot 13 = 3328$ bits, divided into 64-bit words. Since the coefficient length (13-bit) clearly does not divide the block size, the information of a single coefficient may be split across different words.

On the other hand, coefficients of polynomials in $\mathcal{R}_p$ are 10-bit wide and are not generated by the SHAKE-128 block. To simplify the read/write of polynomials in $\mathcal{R}_p$, the coefficients are zero-padded up to 16-bit long, so that exactly four coefficients are contained in one data-memory word and no coefficient is split across different blocks. Our multiplier accommodates both situations while reusing most of its architecture, thus requiring only a few *ad hoc* modifications.

There are different possible approaches to solve the issue of coefficients being split over different blocks. The simplest approach involves a two-words long, i.e. 128-bit long, buffer. Whenever at least 64 bits are empty, a new word is written, while each cycle 13 bits are consumed at the end. This solution, the most software-like, however requires incoming data to be written at different indices (to ensure that coefficients are packed continuously). This approach can be problematic from a hardware-implementation point of view, as it requires a variable bit-shifter for each possible index, thus increasing the area consumption as well as the critical path delay.

Another possible solution that achieves lower area consumption relies on a long buffer, namely a 832-bit long buffer, since that is the least common multiplier of 13 and 64. After 13 cycles of loading, the buffer is filled with exactly 64 coefficients (each of size 13 bits), which can then be consumed. This approach avoids multiple writing indices, but requires a long buffer and a delay (13 cycles) to load 64 coefficients. When we consider a 256-coefficient polynomial, this data-load overhead is around 20% of the pure computation time.

We developed a solution that improves on the second strategy (i.e., use of a long buffer) and reduces both the buffer-size and the cycle overheads. We do not wait for the entire buffer to get filled; instead we start processing as soon as the first few coefficients (from the first word) are available in the buffer. This strategy requires a small multiplexer circuit. This multiplexer reads data from the positions where the first coefficient is on the first cycle, the second coefficient is on the second cycle, etc. More in details, after the first cycle, the first coefficient $a[0]$ is at the location $\texttt{buffer}[624 : 612]$, because $612 = \mathrm{len}(\texttt{buffer}) - 64$. After the second cycle, the second coefficient $a[1]$ is at the location $\texttt{buffer}[573 : 561]$ because the first block has been shifted and we have $561 = \mathrm{len}(\texttt{buffer}) - 2 \times 64 + 13$. More generally, the multiplexer reads the data for the ith coefficient, for $1 \leq i \leq 12$, starting at index $\mathrm{len}(\texttt{buffer}) - 64i + 13(i - 1)$. Fig. 4 shows the first three cycles of data loading and where the multiplexer receives the input from. Furthermore, since we are reading one coefficient per cycle while loading, we can thus shorten buffer as we do not need to store the coefficients that have already been used. Twelve coefficients are thus read during loading since there is a one-cycle delay between writing to the buffer and reading from it. Hence, our architecture uses a buffer that is 676-bit long, since $676 = 64 \times 13 - 12 \times 13$. This means that at the cost of a 13 to 1 multiplexer, our solution–compared to the longer buffer solution–requires almost 20% fewer registers for the buffer and adds a one-cycle delay, compared to 13.

The loading of 10-bit coefficients follows a similar but simplified pattern. Since each coefficient is zero-padded to 16 bits of length, we need to store only two blocks at a time. The loading phase consists of only two cycles. In the first cycle, the first block is loaded; in the second cycle, we read the first coefficient, shift the first block and load the second. Just before the buffer is emptied, we repeat the loading process. Hence we only require a 112-bit buffer. This is because two blocks require 128 bits of memory, but we consume one coefficient while loading.

Lastly, since the multiplier reads the coefficient values from the least-significant part of the buffer, it is possible to load the next 64-bit block of data in the most significant part of the buffer before the buffer is completely emptied out. In this way, multiplication can continue uninterrupted and thus, the overhead due to loading the polynomial $a(x)$ is only one cycle, the cycle needed to load the initial block into the buffer.

3.5.4 Alternative design decisions

Our multiplier loads the secret polynomial $s(x)$ into a register at the start and then progressively reads the coefficients of the polynomial $a(x)$. An alternative to this design decision will be to interchange the positions of $a(x)$ and $s(x)$, i.e., load $a(x)$ entirely into a register and then progressively read the coefficients of $s(x)$. The former design choice has several advantages over the latter, with some minor drawbacks.

Firstly, if the polynomial $a(x)$ were stored in a register, note that we would be doing operations that involve only one coefficient of $s(x)$ at a time. Considering that a potential attacker has control over the values of $a(x)$, such architecture would increase the chances

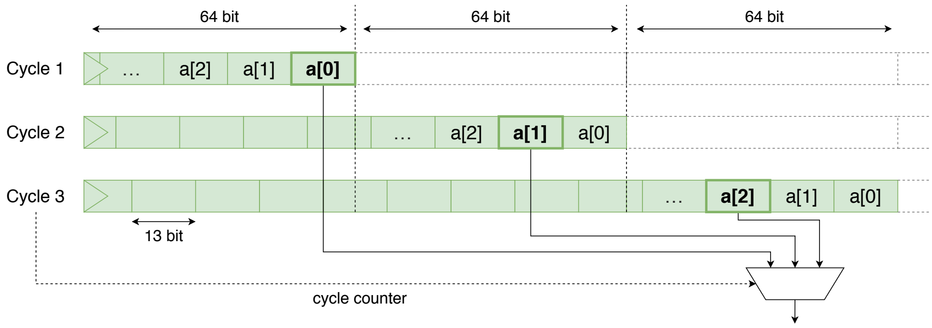


Figure 3: Buffer loading of polynomial data for the first three cycles. Each row represents the buffer at different cycles, and green indicates the polynomial data that has been loaded.

of mounting a successful simple side-channel attack. For instance, if $a(x)$ was set to be $a(x) = 1$, it could be possible to retrieve the secret $s(x)$ by retrieving the Hamming distance of the different states of the accumulator. By storing the secret into a register, any coefficient of $a(x)$ is simultaneously multiplied by *all* the coefficients of $s(x)$ in parallel, which makes the traces of such operations much noisier and thus making it harder for a side-channel attacker.

Secondly, the decision of storing the entire $s(x)$ in the register simplifies the overall architecture, as the register and the data exchange interface with the data-memory (block RAM) does not have to deal with different sizes of coefficients. Note that the coefficients of $s(x)$ are always 4-bits wide (a divisor of 64) and each load stores 16 coefficients into the buffer for $s(x)$. This architecture requires less overhead for data loading: loading $s(x)$ into the register takes 16 cycles only, whereas loading an entire $a(x)$ into the buffer would require 52 cycles.

Finally, our design optimizes the number of flip-flops and logic elements for the shift register. To store $s(x)$ we need only $4 \times 256 = 1{,}024$ flip-flops as opposed to $13 \times 256 = 3{,}328$ flip-flops in the other strategy.

This comes at the cost of a more complicated loading process, since the coefficients of $a(x)$ are stored over multiple RAM blocks, unlike the coefficients of $s(x)$. However, the loading techniques described in Section 3.5.3 ameliorate the problem and the advantages detailed so far greatly outweigh the drawbacks.

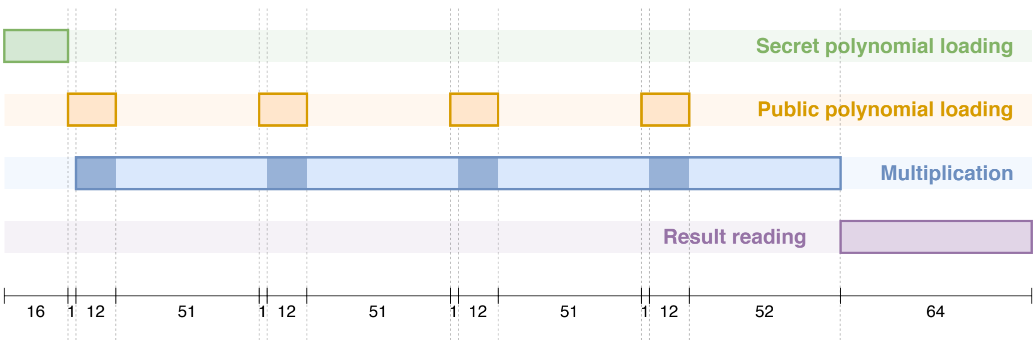


Figure 4: Timeline of polynomial multiplication when the public polynomial has 13 bit coefficients, from input loading to output reading. Darker blue areas denote when the multiplier reads coefficients from the loading data instead of the end of the buffer.

3.5.5 Pipelining the multiplier

It is possible to reduce the length of the critical path in the multiplier by pipelining the MAC units. A MAC unit receives a 13-bit coefficient of $a(x)$ and a 4-bit coefficient of $s(x)$. A pipelined implementation of the MAC computes at one cycle the product between the coefficient of $a(x)$ and the magnitude of the coefficient of $s(x)$ and buffers the result, together with the sign of the secret coefficient. The following cycle updates the accumulator by adding or subtracting the stored result, depending on the buffered sign. Figure 5b contains a representation of the pipelined architecture.

This design allows new inputs to be processed continuously. Thus, an entire polynomial multiplication now takes 257 cycles, which is virtually the same as the non-pipelined architecture (there is only a one-cycle overhead due to pipelining). These changes allow shortening of the critical path, but come at the cost of an additional 14-bit register per MAC unit, which implies an added 3384-bit register for the entire polynomial multiplier.

The same changes can also be applied to MAC units when they fit two multipliers, as described in the next subsection. In this case, the number of required registers is also doubled.

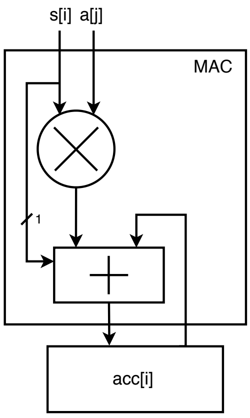
3.5.6 Scalability

The current polynomial multiplier architecture with 256 MACs achieves high performance with a moderate area consumption. However, such architecture can be extended to scale up or down to achieve different performance/area trade-offs. Reducing the area consumption can be achieved by decreasing the number of MACs used. For example, we can use 128 or 64 MACs and only multiply as many coefficients per cycle, which doubles or quadruples the number of cycles.

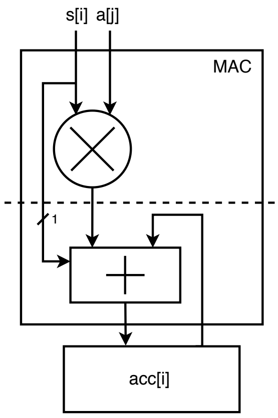
Increasing performance, on the other hand, requires somewhat more involved modifications. In order to reduce the multiplication cycle count to $256/d$, the multiplier must be changed to compute the multiplication of $s(x)$ with d coefficients of $a(x)$ in one cycle, i.e. compute $s(x) \cdot (a_i + a_{i+1}x + \ldots + a_{i+d-1}x^{d-1})$. Since the current architecture round-shifts the secret polynomial at each cycle (equivalent to multiplying it by x), the new architecture needs to cycle-shift the secret by d increments (equivalent to multiplying it by x^d) and the MAC units need to simulate the in-between shifts. In the regular architecture, if at one cycle we update the accumulator at position i with $s[i] \cdot a[j]$, the next cycle we shift s and use the next coefficient of a, so we increase the accumulator by $s[i-1] \cdot a[j+1]$. The following cycle will compute $s[i-2] \cdot a[j+2]$, the one after that $s[i-3] \cdot a[j+3]$ and so on. Thus, each MAC unit now needs to compute d such operations in one cycle. Namely, the MAC associated to position i in the accumulator needs to update the accumulator by $s[i] \cdot a[j] + s[i-1] \cdot a[j+1] + \ldots + s[i-(d-1)] \cdot a[j+d-1]$. This means that each MAC unit should receive in input $s[i], \ldots, s[i-(d-1)]$ and $a[j], \ldots, a[j+d-1]$ and be equipped with d multipliers (see Figure 5c for the MAC architecture when $d = 2$). Note that the indexing of the coefficients of $s(x)$ must be interpreted in a round way, i.e. if $j = 0$, then $s_[j-1]$ denotes the 256th coefficient with its sign flipped.

These changes have a positive impact on the registers required. Since we are now consuming d coefficients per cycle, the polynomial buffer length should be decreased. If $d = 2$, the buffer can be 520-bit long, since 24 coefficients can be read during loading and $520 = \text{lcm}(64, 13) - 24 \times 13$. This means we can reduce the buffers needed by 23%. More generally, the number of coefficients that can be consumed while loading is $12d$, thus the buffer should be $(\text{lcm}(64, 13) - 13 \cdot 12d)$-bit long.

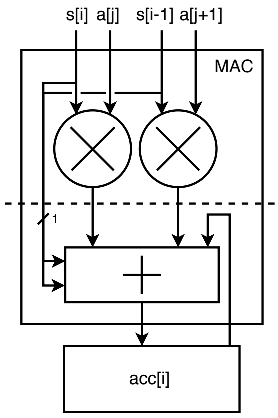
However, increasing the performance comes with an expensive cost in terms of area consumption. For $d = 2$, each MAC unit needs to be equipped with two multipliers and twice as many buffers, thus its area requirements are almost exactly doubled. More generally, we can achieve polynomial multiplication in $256/d$ cycles by multiplying d times



(a) The MAC architecture.



(b) The MAC architecture, pipelined with two stages.



(c) The pipelined MAC architecture when $d = 2$, i.e. it was modified to accomodate two multipliers.

Figure 5: Different architectures of MAC units.

the area consumption of each MAC unit.

4 Results

The instruction-set coprocessor architecture was described in mixed Verilog and VHDL and was compiled using Xilinx Vivado for the target platform Xilinx ZCU102 board that has an UltraScale+ XCZU9EG-2FFVB1156 FPGA. The implemented hardware architecture contains all the building blocks that are required to compute all KEM operations (key generation, encapsulation and decapsulation). During a KEM operation, the operand data is transferred to the coprocessor at once from a host processor, then all the computations are performed in the FPGA, and finally the result is read by the host processor.

4.1 Timing results

The hardware coprocessor runs at 250 MHz clock frequency. Such a high clock frequency on an FPGA demonstrates the impact of the design decisions that we made while implementing the instruction-set coprocessor architecture. All cycle counts are obtained from the hardware using a counter. The implementation is constant-time.

Table 1 shows the cycle counts for the individual low-level operations that are computed during the execution of Saber (module dimension 3). The polynomial multiplier here uses 256 MAC units in parallel, where each MAC fits one multiplier. Although a polynomial multiplication requires around 256 cycles, the KEM operations compute polynomial vector-vector and matrix-vector multiplications. Hence, the time spent on polynomial multiplications is 49%, 54%, and 56% of key generation, encapsulation, and decapsulation respectively. The total time spent on Keccak-based [PA11] functions, namely SHA3-256, SHA3-512, and SHAKE-128, is 33%, 31%, and 22% of key generation, encapsulation, and decapsulation respectively. The results show that, despite having a fast architecture, polynomial multiplication is the most time-consuming primitive, requiring more than half of the overall time.

Table 1: Total cycles spent in low-level operations for Saber (module dimension 3). Clock frequency is 250 MHz. The polynomial multiplier uses 256 MAC units in parallel, with each MAC equipped with one multiplier.

Instruction	**Cycle Count**		
	Keygen	Encapsulation	Decapsulation
SHA3-256	339	585	303
SHA3-512	0	62	62
SHAKE-128	1,461	1,403	1,403
Vector sampling	176	176	176
Polynomial multiplications	2,685	3,592	4,484
Remaining operations	792	800	1,782
Total	5,453	6,618	8,034

Table 2: Area results for the instruction-set coprocessor architecture for Saber.

Block	LUTs	Flip-flops	DSPs	BRAM Tiles
SHA/SHAKE	5,125	3,068	0	0
⌊ Keccak	4,978	2,964	0	0
Binomial sampler	949	412	0	0
Poly-vector multiplier (256 MACs)	17,493	5,113	0	0
⌊ Polynomial multiplier	17,466	5,100	0	0
Other blocks	1,512	2,157	0	0
Saber Coprocessor (256 MACs)	25,079	10,750	0	2
(% of overall FPGA)	9.5%	2.18%	0%	0.2%

4.2 Area consumption

The area results for our coprocessor architecture are shown in Table 2 along with a breakdown of the internal building blocks. The data-memory consists of 1,024 words of width 64-bit and it consumes 2 Block RAM tiles on the FPGA platform. The program-memory (Fig. 1) is a small memory and it is implemented using LUTs. Despite the high performance, our proposed architecture manages to achieve a moderate area consumption: only 9% of LUTs, 2% of flip-flops, 0% of DSP slices, and 0.2% of block RAMs on the target platform. The Keccak-based SHA3/SHAKE block occupies nearly 20 to 28% of the entire coprocessor.

4.3 Results for a higher-speed variant of the architecture

As the polynomial multiplier architecture is scalable, we implemented a variant of it with MAC units fitting two multipliers. With this higher-performing architecture, the cycle counts for polynomial multiplications nearly halves, thus balancing the time between Keccak-based functions and polynomial multiplications. The overall cycle count for Saber (module dimension 3) is 4,320, 5,231 and 6,461 for key generation, encapsulation, and decapsulation respectively. Thus, the cycle count is reduced by 21%, 21%, and 20% respectively. The increased speed comes with increased area consumption by 83% for LUTs and 74% for flip-flops (this is due both to the increased area consumption of the MAC units with two multipliers and of the pipelining).

Table 3: Comparisons with existing implementations of CCA-secure KEM schemes

Implementation	Platform	Time in μs (KeyGen./Encaps./ Decaps.)	Frequency (MHz)	Area (LUT/FF/DSP/BRAM) (or mm^2 for ASIC)
Kyber-768 [BUC19]	ASIC	1.5K/2.4K/2.6K	72	0.28 mm^2
NewHope-1024 [BUC19]	ASIC	1.3K/3.2K/3.6K	72	0.28 mm^2
FrodoKEM-976 [HOKG18]	Artix-7	45K/45K/47K	167	≈7.7K/3.5K/1/24
SIKEp503 [MLRB20]	Virtex-7 (HW/SW)	8.2K/13.9K/14.8K	142	21.2K/13.6K/162/38
Saber [MTK+20]	Artix-7 (HW/SW)	3.2K/4.1K/3.8K	125	7.4K/7.3K/28/2
Saber [DFAG19]	UltraScale+ (HW/SW)	-/60/65	322	≈12.5K/3.5K/256/4
Saber [this work]	UltraScale+	21.8/26.5/32.1	250	25K/10.7K/0/2

4.4 Comparisons with existing implementations

In Table 3 we compare our flexible coprocessor architecture with some of the recent hardware implementations of post-quantum KEM schemes. We remark that a fair comparison between the listed hardware implementations is not always possible since the implementations target different schemes, use different platforms and follow different design methodologies, and sometimes report simulation results. Nevertheless, our coprocessor has been tested in the hardware and the timing results in the table show that our architecture has a very fast computation time for the Saber KEM while consuming a modest area.

The fairest comparisons are with the existing implementations of Saber by Bermudo Mera et al. [MTK+20] and Dang et al. [DFAG19]. Both implementations follow HW/SW codesign to split the computation of a Saber operation among the hardware and software platforms. For example, [MTK+20] accelerates Saber by computing only the polynomial multiplications in the hardware. The high-speed implementation [DFAG19] implements matrix-vector multiplication and inner product computations, matrix and secret generation, and hashing in the hardware and additionally uses dedicated data-paths for key generation, encapsulation and decapsulation, thus lacking flexibility. On the other hand, our instruction-set coprocessor architecture is able to compute all protocol-operations. The results in Table 3 shows that our full-hardware architecture is faster than the two other HW/SW codesign implementations [MTK+20, DFAG19] of Saber.

Banerjee et al. [BUC19] implemented a unified architecture that can be used for multiple lattice-based schemes including Kyber [BDK+18], which is also a module lattice-based KEM scheme. Their design methodology aims at reducing power consumption. In TSMC 40nm technology, their cryptoprocessor occupies 0.28 mm^2 area. For Kyber-768 (module dimension 3), their architecture is around 100 times slower compared to our architecture.

We also compare our results with the hardware implementation of Frodo KEM by Howe et al. [HOKG18]. Their architecture uses dedicated data paths for the key generation, encapsulation and decapsulation operations. Since Frodo is based on the standard LWE problem, computationally expensive matrix-vector multiplications are computed several times, thus making Frodo significantly slower than other ring or module lattice-based schemes.

We also compare our results with non-lattice-based KEM schemes. The SIKE [FJP11] scheme relies on the computational hardness of the supersingular isogeny problem. Its most recent hardware implementation by Massolino et al. [MLRB20] targets high speed and even beats Frodo KEM. Our hardware implementation of Saber is around 500 to 600 times faster.

5 Conclusions

In this work, we proposed a flexible instruction-set coprocessor architecture for lattice-based public-key cryptography with Saber KEM as a case study. We showed how to design a fast yet simple and scalable polynomial multiplier using the schoolbook multiplication algorithm. We optimized the implementation targeting the parameter sets of Saber. For a security level similar to AES-192, the architecture achieves fast computation time and computes Saber key generation, encapsulation and decapsulation in 21.8, 26.5, and 32.1 μs respectively and consumes only 9% of LUTs and 2% of flip-flops on an UltraScale+ FPGA. The results show that the modular structure of Saber and the use of power-of-two moduli simplifies the architecture and results in better performance.

6 Acknowledgements

We thank Frederik Vercauteren for discussions during this work.

References

[20115] National Institute of Standards and Technology. 2015. SHA-3 standard: Permutation-Based Hash and Extendable-Output Functions. FIPS PUB 202, 2015.

[ABB+19] Erdem Alkim, Paulo S. L. M. Barreto, Nina Bindel, Juliane Kramer, Patrick Longa, and Jefferson E. Ricardini. The lattice-based digital signature scheme qtesla. Cryptology ePrint Archive, Report 2019/085, 2019. https://eprint.iacr.org/2019/085.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 327–343. USENIX Association, 2016.

[Aea19] Frank Arute et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[BDK+18] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehle. Crystals - kyber: A cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 353–367, 2018.

[BMKV20] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in toom-cook multiplication: an application to module-lattice based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):222–244, Mar. 2020.

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *EUROCRYPT 2012*, pages 719–737, 2012.

[BUC19] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):17–61, 2019.

[DFAG19] Viet B. Dang, Farnoud Farahmand, Michal Andrzejczak, and Kris Gaj. Implementing and benchmarking three lattice-based post-quantum cryptography algorithms using software/hardware codesign. In *International Conference on Field-Programmable Technology, FPT 2019, Tianjin, China, December 9-13, 2019*, pages 206–214. IEEE, 2019.

[DKRV19] Jan-Pieter DÁnvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber. Proposal to NIST PQC Standardization, Round2, 2019. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/round-2-submissions.

[DKSRV18] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. *Saber: Module-LWR Based Key Exchange, CPA-Secure Encryption and CCA-Secure KEM*, volume 10831, page 282–305. Springer International Publishing, 2018.

[FJP11] Luca De Feo, David Jao, and Jérôme Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. Cryptology ePrint Archive, Report 2011/506, 2011. https://eprint.iacr.org/2011/506.

[GFS+12] Norman Göttert, Thomas Feller, Michael Schneider, Johannes Buchmann, and Sorin Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, pages 512–529, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A Modular Analysis of the Fujisaki-Okamoto Transformation. Cryptology ePrint Archive, Report 2017/604, 2017. http://eprint.iacr.org/2017/604.

[HOKG18] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. Standard lattice-based key encapsulation on embedded devices. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):372–393, 2018.

[KMRV18] Angshuman Karmakar, Jose Maria Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on ARM cca-secure module lattice-based key encapsulation on ARM. *IACR Cryptology ePrint Archive*, 2018:682, 2018.

[Knu97] Donald Knuth. *The Art of Computer Programming, Volume 2. Third Edition.* Addison-Wesley, 1997.

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking nist pqc on arm cortex-m4. Second PQC Standardization Conference, 2019. https://eprint.iacr.org/2019/844.

[MLRB20] Pedro Maat C. Massolino, Patrick Longa, Joost Renes, and Lejla Batina. A compact and scalable hardware/software co-design of SIKE. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):245–271, 2020.

[MTK+20] Jose Maria Bermudo Mera, Furkan Turan, Angshuman Karmakar, Sujoy Sinha Roy, and Ingrid Verbauwhede. Compact domain-specific co-processor for accelerating module lattice-based key encapsulation mechanism. *IACR Cryptology ePrint Archive*, 2020:321, 2020.

[PA11] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche. The Keccak reference. Round 3 submission to NIST SHA-3, 2011.

[PG14] Thomas Pöppelmann and Tim Güneysu. Towards Practical Lattice-Based Public-Key Encryption on Reconfigurable Hardware. In *Selected Areas in Cryptography – SAC 2013*, Lecture Notes in Computer Science, pages 68–85. Springer Berlin Heidelberg, 2014.

[RJV+18] Sujoy Sinha Roy, Kimmo Järvinen, Jo Vliegen, Frederik Vercauteren, and Ingrid Verbauwhede. Hepcloud: An fpga-based multicore processor for FV somewhat homomorphic function evaluation. *IEEE Trans. Computers*, 67(11):1637–1650, 2018.

[Roy19] Sujoy Sinha Roy. Saberx4: High-throughput software implementation of saber key encapsulation mechanism. In *37th IEEE International Conference on Computer Design, ICCD 2019, Abu Dhabi, United Arab Emirates, November 17-20, 2019*, pages 321–324. IEEE, 2019.

[RTJ+19] Sujoy Sinha Roy, Furkan Turan, Kimmo Järvinen, Frederik Vercauteren, and Ingrid Verbauwhede. Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. In *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*, pages 387–398. IEEE, 2019.

[RVM+14a] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact ring-lwe cryptoprocessor. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 371–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[RVM+14b] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact ring-lwe cryptoprocessor. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 371–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[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, Oct 1997.

[Tea19] Keccak Team. Keccak in VHDL: High-speed core. https://keccak.team/hardware.html, Accessed on November 2019.