Journal of Cryptographic Engineering (2024) 14:1–18
https://doi.org/10.1007/s13389-023-00332-2

REGULAR PAPER

Uniform instruction set extensions for multiplications in contemporary and post-quantum cryptography

Felix Oberhansl[1] · Tim Fritzmann[3] · Thomas Pöppelmann[3] · Debapriya Basu Roy[4] · Georg Sigl[1,2]

Received: 24 February 2022 / Accepted: 22 July 2023 / Published online: 24 August 2023

Abstract
Hybrid key encapsulation is in the process of becoming the de-facto standard for integration of post-quantum cryptography (PQC). Supporting two cryptographic primitives is a challenging task for constrained embedded systems. Both contemporary cryptography based on elliptic curves or RSA and PQC based on lattices require costly multiplications. Recent works have shown how to implement lattice-based cryptography on big-integer coprocessors. We propose a novel hardware design that natively supports the multiplication of polynomials and big integers, integrate it into a RISC-V core, and extend the RISC-V ISA accordingly. We provide an implementation of Saber and X25519 to demonstrate that both lattice- and elliptic-curve-based cryptography benefits from our extension. Our implementation requires only intermediate logic overhead, while significantly outperforming optimized ARM Cortex M4 implementations, other hardware/software codesigns, and designs that rely on contemporary accelerators.

Keywords Post-quantum cryptography · Hybrid key encapsulation · Elliptic-curve cryptography · Lattice-based cryptography · RISC-V · Instruction set extensions

1 Introduction

Public-key cryptography is the foundation for secured communication over the Internet, but the security assumptions of contemporary cryptographic primitives like RSA and ECC are invalid once large-scale quantum computers are available. To protect communication from eavesdropping, NIST called for proposals of post-quantum cryptography (PQC) schemes. Approaches to integrate PQC into protocols like TLS and SSH feature almost exclusively hybrid key encapsulation mechanisms [1, 2], where a trusted contemporary standard (RSA or elliptic curve) and a post-quantum scheme are combined. This is recognized by institutions like the BSI [3] and NIST [4].

For constrained systems, the computationally expensive operations required for hybrid key encapsulation impose a serious challenge. Besides hashing, ECC/RSA and lattice-based cryptography require expensive arithmetic. For ECC and RSA, big-integer arithmetic is needed. Structured lattice-based schemes are built on polynomial arithmetic. In scenarios where a device has to support only big-integer arithmetic and where cryptographic operations must not exceed a certain time frame, hybrid encapsulation tightens the requirements in two different ways. For one, the same device must now

The majority of this work was done while the authors Tim Fritzmann, Debapriya Basu Roy were associated with Technical University Munich.

✉ Felix Oberhansl
felix.oberhansl@aisec.fraunhofer.de

Tim Fritzmann
tim.fritzmann@tum.de

Thomas Pöppelmann
thomas.poeppelmann@infineon.com

Debapriya Basu Roy
dbroy@cse.iitk.ac.in

Georg Sigl
sigl@tum.de

1 Hardware Security Department, Fraunhofer AISEC, Lichtenbergstr. 11, 85748 Garching near Munich, Germany

2 Chair of Security in Information Technology, Technical University of Munich, Theresienstr. 90, 80333 Munich, Germany

3 Connected Secure Systems, Infineon Technologies AG, Am Campeon 1-15, 85579 Neubiberg, Germany

4 Computer Science and Engineering, IIT Kanpur, Kanpur 208 016, India

support polynomial arithmetic, but manufacturers might not want to increase cost by adding a dedicated accelerator. Also, devices often have use-case-driven time frames that are available for cryptographic operations and should not be exceeded to avoid overall performance degradation. Now, two primitives must fit into a time frame that was previously allocated for one. In this work, we show a CPU hardware extension that uniformly supports big-integer and polynomial arithmetic and thus avoids the cost of integrating separate accelerators and performance degradation.

Hardware extensions can speed up computational bottlenecks, e.g., hashing and big-integer/polynomial arithmetic in cryptography. We propose to employ instruction set extensions, which—in a certain sense—are tightly coupled accelerators that do not require expensive data transfers to/from the CPU and can reuse its hardware resources. We integrate them into a RISC-V core. For evaluation, we choose Saber [5], since its polynomial multiplication is similar to that in NTRU [6] in the fact that modular coefficient reduction is easy. Therefore, supporting NTRU with our design is straightforward. We choose the widely deployed X25519 [7] as a contemporary standard.

Related Work

Previous efforts to support PQC and contemporary standards target contemporary accelerators developed for ECC or RSA [8–10]. The authors propose Kronecker-based algorithms to map polynomial to big-integer multiplication, which implies an overhead.

The authors of [11, 12] propose RISQ-V, a set of RISC-V instruction set extensions designed for lattice-based cryptography.

This is—to the best of our knowledge—the first work on new hardware architectures targeting both multiplications in contemporary cryptography and PQC. New architectures are required to achieve optimal cost and performance of hybrid key encapsulation on embedded systems.

Contribution

We use [11] as basis for our work by integrating our design into their *PQ* instruction set. Our contributions are:

- The development of the *XSMUL*, the first design that natively supports fast arithmetic for lattice-based and elliptic-curve cryptography. Our architecture is parameterizable, scalable, and can efficiently support multiplication algorithms beyond schoolbook multiplication like Karatsuba and Toom–Cook.
- An integration of the XSMUL into the RISC-V ISA. We provide results for a prototype FPGA implementation of the PULPino architecture, extended with our design.
- A novel, memory-optimized algorithm for polynomial multiplication in rings, which we refer to as *ring-splitting multiplication*.
- Optimized implementations of schoolbook, Karatsuba/Toom–Cook, and ring-splitting multiplication that are supported by the XSMUL. We use Saber [5] to benchmark our algorithms and review their applicability to other PQC algorithms.
- A new cycle count record for Saber [5] and the first implementation of X25519 [7] on a RISC-V platform with ISA extensions. All our implementations run in constant time.

Organization

Section 2 provides the necessary preliminaries for lattice-based and elliptic-curve cryptography and multiplication methods. In Sect. 3, we describe the extended schoolbook multiplier design, the rationale behind it, and optimizations for DSP slice architectures. The integration into a RISC-V core and the ISA extensions are presented in Sect. 4. In Sect. 5, we present a novel algorithm for polynomial multiplication. Sections 6 and 7 show the application to Saber and X25519. In Sect. 8, the results for our prototype FPGA implementation are shown and compared with state-of-the-art works. A final discussion in Sect. 9 concludes this work.

2 Background

2.1 Notation

In both lattice-based and elliptic-curve cryptography, integers are defined in a ring $\mathbb{Z}_q$, i.e., modulo an integer q. In lattice-based cryptography, polynomials are typically of degree n and elements of the ring $\mathcal{R} = \mathbb{Z}_q/\langle\phi(X)\rangle$, where n is an integer and $\phi(X)$ is a cyclotomic polynomial $\phi(X)$ (typically $\phi(X) = X^n + 1$).

2.2 Saber

Saber [5] is built upon polynomial arithmetic, where polynomials are defined in $\mathbb{Z}_q[X]/(X^{256}+1)$. All coefficient moduli q are powers of two with a maximum value of 2^{13}. To configure security levels, matrices and vectors of polynomials with size $l \times l$, l respectively, are used.

2.3 X25519

X25519 [7] is a Diffie–Hellmann key exchange that uses Bernstein's Curve25519. Two parties multiply a common base point with a secret scalar to establish a shared secret. A time-constant Montgomery ladder is used for scalar multiplication. The key operations are multiplications, additions, and subtractions in $\mathbb{F}_p$. If p is a (pseudo)-Mersenne prime, the modular reduction can exploit the fact that $a \equiv a_h \cdot k + a_l \pmod{(2^m - k)}$, where $a = a_h \cdot 2^m + a_l$. For X25519, $p = 2^{255} - 19$, and thus $a = a_h \cdot 2^{255} + a_l \equiv a_h \cdot 19 + a_l \pmod{p}$.

2.4 Multiplication methods in this work

In addition to the simple schoolbook method, polynomial and integer multiplication can be performed by different algorithms in different time complexities. For a more detailed overview, we recommend [10].

Kronecker
Kronecker substitution [13] maps $\mathbb{Z}[X] \rightarrow \mathbb{Z}$, s.t. the polynomial multiplication result can be recovered from the integer multiplication result. The main idea is to evaluate polynomials at a large enough value, multiply the resulting integers and read the coefficients from the integer product.

Number-Theoretic Transform (NTT)
The NTT [14] is a special case of the discrete Fourier transformation (DFT) in finite fields $\mathbb{Z}_q$. A polynomial $a(X)$ can be efficiently converted into its NTT-representation if the nth root of unity ω_n and the $2n$th root of unity are elements of $\mathbb{Z}_q$. In the NTT-representation, polynomial multiplication corresponds to coefficient-wise multiplication.

Nussbaumer
Nussbaumer made the observation that multiplications in $\mathbb{Z}[X]/(X^n + 1)$ can be simplified according to the mapping below [15]:

$$\begin{aligned} \Psi : \mathbb{Z}[X]/(X^n + 1) &\rightarrow (\mathbb{Z}[Y]/(Y^{n/t} + 1))[X]/(Y - X^t) \\ a = \sum_{i=0}^{n-1} a_i X^i &\rightarrow \Psi(a) = \sum_{i=0}^{t-1} \left(\sum_{j=0}^{n/t-1} a_{i+jt} Y^j \right) X^i \end{aligned} \tag{1}$$

The mapping splits the ring, i.e., the original polynomial is mapped to a sum of polynomials in the smaller ring $\mathbb{Z}[Y]/(Y^{n/t} + 1)$. Then the polynomial multiplication can be seen as a $2t$ cyclic convolution of $\Psi(a)$ and $\Psi(b)$, with $A_i, B_i = 0$ for $\forall i \geq t$. The coefficients of the polynomial $\left(\sum_{i=0}^{t-1} A_i X^i\right)$ are themselves polynomials in $\mathbb{Z}[Y]/(Y^{n/t} + 1)$.

Toom–Cook/Karatsuba
Toom–Cook-k-way multiplication [16] is a divide-and-conquer approach to schoolbook multiplication. Karatsuba [17] is a special case of Toom–Cook for $k = 2$. The underlying rationale is to split a large polynomial into a sum of $2k - 1$ polynomials of degree $\frac{n}{k}$, which are then multiplied. Afterward, the result of the original multiplication is reconstructed.

3 Extended schoolbook multiplier design

We introduce a uniform architecture for polynomial and integer multiplication to target the performance bottlenecks and integration issues discussed in Sect. 1. Our proposal is using an *Extended Schoolbook Multiplier*, hereafter referred to as XSMUL.

Rationale for using polynomial schoolbook multipliers
In hardware, NTT-unfriendly schemes typically employ schoolbook multiplication architectures, e.g., Saber [18, 19] and NTRU [20, 21]. Due to simple coefficient reduction, this approach can be implemented efficiently. Designs can be optimized for either high performance or low area by choosing the number of multiply & accumulate units that handle polynomial coefficient multiplication accordingly. State-of-the-art implementations [18–21] are tailored to the respective scheme, i.e., they are designed for the specific coefficient width and support only direct polynomial multiplication/convolution.

Our approach of supporting different polynomial multiplication algorithms in combination with integer multiplication demands a more general design. We want to extend the multiply & accumulate units for big-integer multiplication. Many big-integer multipliers rely on cascading smaller multiply & accumulate units. In [22], the authors propose an architecture of cascaded DSP slices for elliptic-curve cryptography. A distinct accumulator block handles carries and merges the partial schoolbook products. This specific approach makes it hard to support polynomial multiplication. In the following, we propose a new approach that supports both polynomial and integer multiplication.

Design motivation
The hardware primitive we present in the next subsection can be instantiated for arbitrary polynomial or integer sizes. For ECC and lattice-based cryptography, the integers needed for the former will always require less bit than the polynomials for the latter. For example, a schoolbook multiplier that can handle polynomials as needed for Saber would have to be built from 256 multiply & accumulate units with 13-bit each. We want to avoid this hardware overhead by choosing a smaller number that satisfies the requirement for ECC and allows integration into a small RISC-V core for embedded systems.

3.1 The XSMUL module

Figure 1 shows the basic structure of the proposed Extended Schoolbook Multiplier (XSMUL). The multiply & accumulate units support polynomial and big-integer multiplication. Integers are represented as polynomials by splitting them into chunks of w bit, i.e., $a = a_m 2^{mw} + \cdots + a_1 2^w + a_0$. Multiplying two chunks of w bits and adding w bits afterward yields a $2w + 1$-bit wide result. The lower half ($r_{k\text{low}}$) of the result is shifted into the subsequent multiply & accumulate unit to the right. The upper half ($r_{k\text{carry}}$) is fed back into the adder in the next cycle. For two integers of n bit, $N = \lceil n/w \rceil$

multiply & accumulate units are required, and it takes N steps until both input integers are absorbed (neglecting one pipeline cycle). Afterward, it takes additional N steps to shift out the last chunk and complete the integer multiplication. During this phase, only carry operations are performed. An additional accumulate block with carry logic could reduce these N cycles to a single cycle. However, it would need a connection to all N registers, which would also imply a deep critical logic path. Since we aim at keeping the design as compact as possible, we do not further explore this. Additionally, our design allows us to integrate modular reduction directly into the integer multiplication and map it directly into common DSP slices.

The pipeline registers between multiplier and adder are placed to keep the critical path short. The signs of adder inputs can be configured to integrate arithmetic beyond multiplication. In the following, arithmetic operations that were considered for the basic design are listed. The actual implementation depends on the parameter set (w, N). Parameters for w are chosen in accordance with DSP slice architectures (Sect. 3.2), and N is chosen in accordance with the RISC-V core (Sect. 4).

- **Polynomial multiplication** The basic schoolbook multiplication of polynomials $a(X)$, $b(X)$ with polynomial degrees $d_a, d_b < N$ is straightforward. Polynomial $a(X)$ is shifted through the leftmost register and every clock cycle a result coefficient r_k is produced in the rightmost register.
  Multiplication of a polynomial $a(X)$ with arbitrary d_a and $b(X)$ with $d_b < N$ is handled intuitively. After the first N coefficients $a_{N-1}X^{N-1} + \cdots + a_1X + a_0$ were shifted through register a, N result coefficients $r_{N-1}X^{N-1} + \cdots + r_1X + r_0$ were shifted out of the multiplier, and $N - 1$ intermediate result coefficients $r_{2N-2}X^{2N-2} + \cdots + r_NX^N$ reside in the XSMUL registers. The next chunk of $a(X)$ can be piped through register a to continue operation seamlessly.
  Multiplication of polynomials $a(X)$ and $b(X)$ with arbitrary degrees d_a, $d_b \geq N$ is divided into multiple multiplications of one polynomial with arbitrary degree and one polynomial part with $d_{\text{part}} < N$.
- **Convolution** Arithmetic in lattice-based cryptography is usually performed in the ring $\mathcal{R}_n = \mathbb{Z}_q/\langle\phi_n(X)\rangle$ where $\phi_n(X) = (X^n + 1)$ or $\phi_n(X) = (X^n - 1)$, i.e., the multiplication corresponds to a wrapped convolution. Instead of shifting the resulting coefficient r_k out of the architecture, it can be fed back into the leftmost unit for direct convolution support, similar to designs in [18, 20], and others.
- **Integer multiplication** Integer multiplication resembles the basic schoolbook approach with additional carry correction. In contrast to polynomial multiplication, the result only becomes valid once it passes the rightmost result register. For ease of implementation, the basic integer multiplication features operation modes to obtain only the lower and upper half of the result.
- **Integer multiplication with integrated Pseudo-Mersenne prime reduction** Elliptic-curve algorithms require arithmetic in $\mathbb{F}_p$. Some curves use a (pseudo)-Mersenne prime for p, e.g., for Curve25519, $p = 2^{255} - 19$. As described in Sect. 2, reduction modulo p can be calculated as $r \equiv r_h \cdot 19 + r_l \pmod{p}$. The proposed architecture can merge one reduction step into the integer multiplication with no cycle and only minimum hardware overhead. In the rightmost multiply & accumulate unit, r_h is continuously multiplied with 19 and added to the lower half, which is fed back into the adder. If two 256-bit integers are multiplied, this mode produces a 263-bit

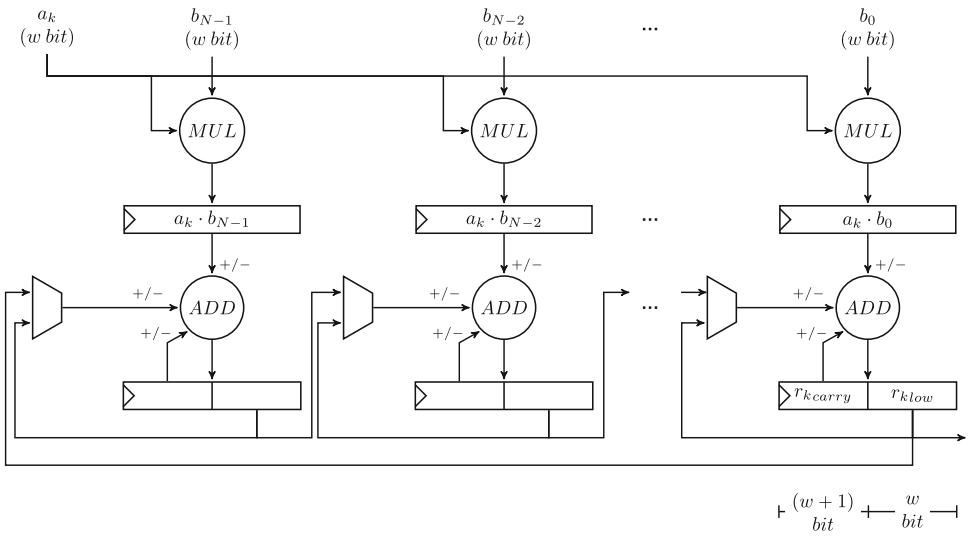

Fig. 1 Basic structure of the XSMUL module

result. For complete reduction modulo p, the procedure above must be repeated until $0 \leq r < p$.

- **Vector addition, vector multiplication & add** The proposed architecture supports addition in combination with scalar multiplication of polynomials, i.e., $(b_{1N-1}X^{N-1} + \cdots + b_{11}X + b_{10}) + l \cdot (b_{2N-1}X^{N-1} + \cdots + b_{21}X + b_{20})$.
- **Integer addition and subtraction** Integer chunks are added/subtracted in a vector fashion during which over-/underflows may occur. These are handled by propagating the result through the multiply & accumulate units as done during multiplication. Since adders are usually wider than multipliers, an additional parameter w_2 for addition/subtraction can be introduced (see next subsection).

3.2 DSP slice mapping

For our prototype, we target FPGAs due to their combination of performance and flexibility. DSP slices are essential building blocks in FPGAs. Mapping the proposed multiply & accumulate unit to DSP slice architectures leads to an efficient hardware implementation. For evaluation, we chose Xilinx UltraScale+ (DSP48E2 slices) and 7-series (DSP48E1 slices) FPGAs.

DSP48E2 slices are the primary target for this work. The complete multiply & accumulate unit can be mapped into one slice. The coefficient width was fixed to $w = 17$-bit, which can be supported with a slice-internal shift. Both registers of one multiply & accumulate unit are mapped to slice-internal registers. The DSP slice can be configured for all required arithmetic operations.

DSP48E1 slices do not provide all functionality for the multiply & accumulate unit, as the central adder lacks one input. The addition operation has to be divided into addition and carry computation, where the carry path is implemented in LUT logic. The output register is also mapped to LUT logic outside of the slice.

Addition and subtraction of integers are optimized uniformly for the 48-bit adder in DSP48E2 and DSP48E1 slices. We introduce the additional parameter $w_2 = 48$-bit. For addition and subtraction, integers are split into w_2-bit wide chunks instead of the smaller w-bit chunks that are used for multiplication. A separate 1-bit carry path is integrated for additions and subtractions.

To take full advantage of the DSP mapping, manual optimizations are required. A general SystemVerilog description of a multiply & accumulate unit implemented on an UltraScale+ FPGA yields the following utilization: 364 LUTs, 83 FFs, 1 DSP. By manually instantiating the DSP48E2 slice, the exact mapping can be achieved and the utilization reduces to 62 LUTs, 0 FFs, and 1 DSP.

While not evaluated in detail, we suspect that similar optimizations are possible for PolarFire FPGAs, which feature dedicated math-blocks with 18×18-bit multiply & accumulate units and Stratix FPGAs, which feature 18×19-bit multipliers. While not essential to this work, we included our DSP slice mapping as case study to demonstrate the value of architecture specific optimizations.

4 Integration into RISC-V Core and instruction set

This section discusses the integration of the XSMUL design into a RISC-V core and presents the corresponding ISA extension. Our target is the PULPino platform,[1] a single-core microcontroller system with CV32E40P core, developed for energy-efficient embedded systems. Its performance is comparable to the widely deployed ARM Cortex M4, but ARM compilers and ISA usually yield slightly better cycle counts.

We select $N = 16$ for the number of multiply & accumulate units. For polynomial multiplication in lattice-based schemes, a power of 2 should be selected, and Curve25519 requires 255-bit integer arithmetic, i.e., at least 15 multiply & accumulate units are required.

Register integration

Similar to [11], the 32×32-bit floating point register set and parts of the 32×32-bit general purpose register set are used. Data inputs and outputs of the XSMUL are directly mapped to registers (Fig. 2). Since $N = 16$, eight 32-bit registers are needed for one input block, e.g., $a_{N-1} \ldots a_0$. In total, 32 32-bit registers are needed, for the connection to the XSMUL, the entire floating point register. While computing, the module shifts input data through registers. The exact register coupling depends on the arithmetic mode, as different coefficient widths are used. It is important that register transfer operations do not become a bottleneck. For this purpose, the barrel-shift (dashed lines) and shadow load (dotted lines in Fig. 2) enable the transfer of multiple registers within a single cycle, e.g., from the general purpose register set to the floating point register set. Both the floating point and the general purpose register file must be extended with parallel access ports.

Pipeline integration

The XSMUL is coupled directly to CPU registers and requires parallel access; therefore, it is integrated into the Instruction Decoding (ID) stage (Fig. 3). The control interface in Fig. 2 is connected directly to the decoder. The RISQ-V [11] Keccak accelerator was also integrated into the ID stage. From Table 1, it can be seen that most of our instructions have a high latency of more than 10 CPU cycles.

[1] https://github.com/pulp-platform/pulpino.

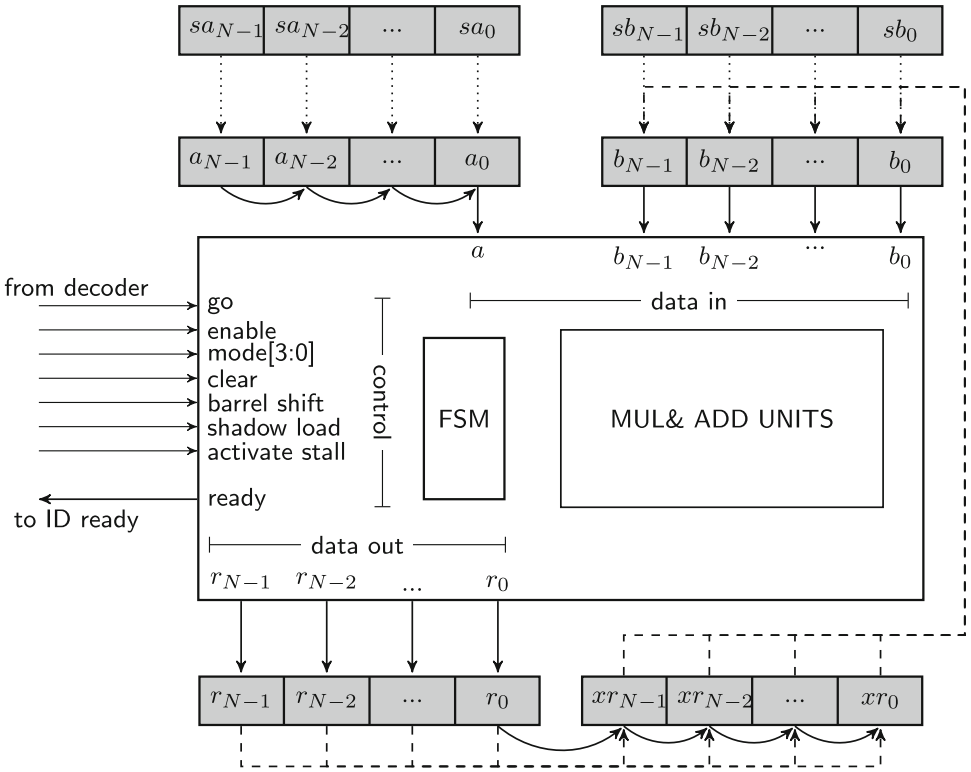

Fig. 2 CPU register coupling (blocks depicted in gray are CPU register halves) and control interface of the XSMUL in the CV32E40P

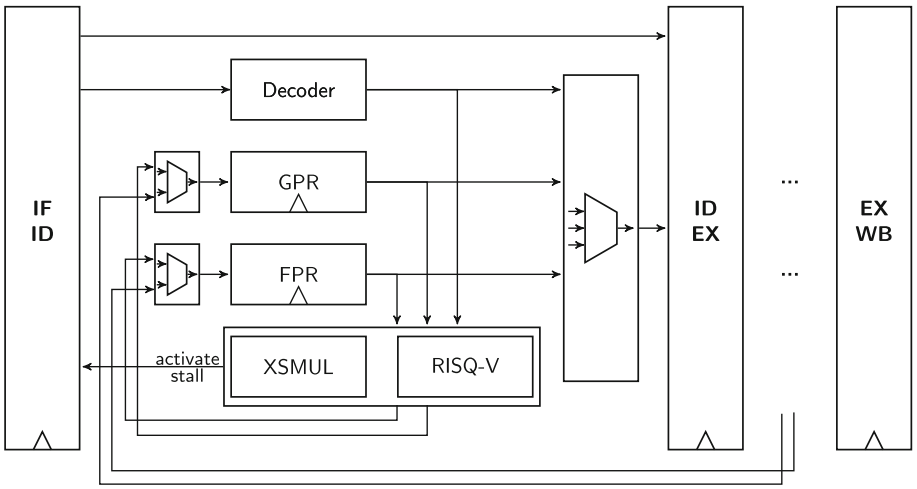

Fig. 3 Pipeline integration of the XSMUL into the Instruction Decoding (ID) stage of the CV32E40P

The pipeline stalls until one instruction is finished. As most of these instructions are immediately followed by another instruction with a high latency, this stalling does not harm the overall performance too much.

Extension of the RISC-V ISA

This work extends the *PQ* instruction set of [11] with nine different arithmetic modes and five configuration operations for the XSMUL. Only two assembly instructions are defined to save space for future extensions: The *pq.xsmul* and *pq.xsmul_config* instruction (Table 1). The actual operation is encoded into the *rs1* field of the instruction. The decoder is modified to translate the value for the control interface in Fig. 2. As discussed above, the registers are connected directly to the XSMUL; therefore, no further operands are necessary. This implies that assembly code must be carefully developed and before scheduling a *pq.xsmul* instruction, everything must be loaded into dedicated registers. We deem this approach as sufficient for our extension, as vector extensions of more advanced processors operate in a similar manner.

Applicability to other RISC-V cores

The PULPino with its four-stage pipeline is a rather minimalistic core. This has both advantages and disadvantages for our extension. The biggest advantage is that it makes the inte-

Table 1 XSMUL instructions in the PQ extension [11]

Instruction	*rs1*	Operation	Latency [CPU cycles]
pq.xsmul	0x0	Polynomial mul.	19
	0x1	Convolution	19
	0x2	Lower-half integer mul.	19
	0x3	Higher-half integer mul.	16
	0x4	$F_{2^{255}-19}$ integer mul.	35
	0x5	Vector add.	3
	0x6	Vector mul.& add.	3
	0x7	Integer add.	7
	0x8	Integer sub.	7
	0x9	Ring-reduction[a]	3
pq.xsmul_cfg	0x0	Clear	1
	0x1	Barrel-shift	1
	0x2	Shadow-load	1
	0x3	Stall	–

[a] The operation $\pmod{(Y - X^t)}$ (Sect. 2.4)

gration rather simple, as the XSMUL's operations can start directly after the instructions are decoded in the ID stage. On the other hand, this implies that our extension needs direct connections to the CPU's registers, which increases the number of ports needed for the register file. This has a negative impact on the critical path in the CPU. A more advanced core with a deeper pipeline could improve this bottleneck by stretching the register accesses over multiple pipeline stages. Those cores are often built on top of a 64-bit architecture and have a dedicated vector processing unit with dedicated vector registers. The XSMUL is the equivalent of such a unit for a constraint 32-bit core. Therefore, it could also be integrated directly into the vector processing unit of more advanced RISC-V cores. Instead of reusing CPU registers, the XSMUL could be connected to the vector register file, which features wider registers and is designed for parallel access.

5 Memory optimized ring-splitting multiplication

To multiply the 256-coeff. polynomials in Saber, we need efficient divide & conquer software implementations. We bring forward a novel, memory-optimized algorithm. Standard algorithms for polynomial multiplication do not exploit the fact that polynomials in lattice-based cryptography are defined in rings (e.g., $\mathbb{Z}_q[X]/(X^n+1)$). It is desirable to integrate the polynomial reduction modulo (X^n+1) directly into the multiplication to avoid that the result grows to double the size it needs to be. In [23], the authors achieve this, by combining Karatsuba multiplication with segmented reduction. We propose *ring-splitting multiplication*, a new algorithm dedicated to the introduced hardware accelerator. It is based on the idea of splitting the ring in Eq. 1, which inherently includes reduction and therefore reduces memory accesses. In comparison to [23], our algorithm is easier to vectorize in a hardware implementation.

Algorithm 1 Ring-Splitting Multiplication

Input: $a = a(X), b = b(X)$, **Output:** $r = r(X) = a(X) \cdot b(X) \pmod{(X^n+1)}$
Note: Polynomials are treated as zero-indexed arrays, t needs to be defined as described in Sect. 2

function RingSplittingMultiplication(a, b, r)
  $a \leftarrow \Psi(a)$
  $b \leftarrow \Psi(b)$
  for $j \leftarrow 0$ to $(t-1)$ **do**
    $i_r, i_a \leftarrow j$; $i_b \leftarrow 0$
    $R \leftarrow \forall 0$
    for $i \leftarrow 0$ to $(t-1)$ **do**
      $A \leftarrow a[(n/t)(i_a+1)-1 : (n/t)i_a]$
      $B \leftarrow b[(n/t)(i_b+1)-1 : (n/t)i_b]$
      $T \leftarrow A \cdot B \pmod{(Y^{n/t}+1)}$ ▷ pq.xsmul conv.
      if $(i_a + i_b) \geq t$ **then**
        $T \leftarrow T \pmod{(Y - X^t)}$ ▷ pq.xmsul ring-red.
      end if
      $R \leftarrow R + T$ ▷ pq.xsmul vector-add.
      $i_a \leftarrow i_a + 1$
      $i_b \leftarrow (i_b - 1) \pmod{t}$
    end for
    $r[(n/t)(i_r+1)-1 : (n/t)i_r] \leftarrow R$
  end for
  $r \leftarrow \Psi^{-1}(r)$
end function

In Nussbaumer's original algorithm, the mapping in Eq. 1 is used as the basis for a transformation, s.t., instead of t^2, t element-wise (nega)-cyclic convolutions are required. This was used in [24, 25] and adapted to map polynomials to integers with Kronecker substitution in [8, 10]. It cannot be

easily adapted for Saber as it requires a division by $2t$, for which no multiplicative inverse in $\mathbb{Z}_q$ exists. We propose to perform the t^2 (nega)-cyclic convolutions directly. This does not improve the time complexity of $\mathcal{O}(n^2)$ in any way. The underlying assumption is that a simple approach backed by efficient hardware-accelerated operations outperforms algorithms with better time complexity but complicated procedures. The procedure used for ring-splitting multiplication is listed in Algorithm 1. The structure introduced with Ψ allows to compute one coefficient polynomial completely, before writing it back to memory (Line 18).

6 Application to Saber

Saber's [5] main multiplication problem is a matrix–vector multiplication $A \cdot s$, where elements of both are polynomials in $\mathbb{Z}_q[X]/(X^n + 1)$. We start by looking at the problem of multiplying two polynomials and then use the matrix–vector structure for optimizations. The previously selected parameter $N = 16$ enables efficient arithmetic with polynomials of 16 coefficients.

6.1 Polynomial multiplication in Saber

In the following, we use our instruction set extensions to optimize different polynomial multiplication algorithms.

Schoolbook

Schoolbook multiplication forms the baseline for all multiplication algorithms. Multiplication of polynomial $a(X) \in \mathbb{Z}_q[X]/(X^n + 1)$ and 16 coefficients of polynomial $b(X)$ is straightforward. The procedure is repeated 16 times to extend the multiplication to that of two polynomials $\in \mathbb{Z}_q[X]/(X^n + 1)$. The reduction modulo $(X^n + 1)$ is enhanced with vector subtractions.

(Multilevel) Karatsuba

Karatsuba is applied as described in Sect. 2.4 with $k = 2$. The 1-level instance computes the polynomial multiplications of degree 127 directly with schoolbook multiplication, while the 2-level instance uses Karatsuba once more to decrease the degree of polynomial multiplications to 63. The polynomial additions/subtractions are accelerated with the vector mode.

Toom–Cook-4-way

Toom–Cook with $k = 4$ has been adapted for hardware [19, 26] and software [27]. Larger values for k are seldom found in the literature. The evaluation points $Y \in [\infty, 2, 1, -1, 1/2, -1/2, 0]$ are used. Evaluation corresponds to a multiplication with matrix E and interpolation to a multiplication with matrix I. Multiplying a polynomial with a scalar and accumulating it with others is directly supported by the XSMUL's multiplication & addition mode. For evaluation, this is straightforward, whereas the matrix I contains fractions, as it is derived from the inverse of an expanded and quadratic matrix E'. Fractions are computationally expensive. In [26], the authors propose a division-free Toom–Cook architecture by rewriting $I = \frac{1}{360} \cdot I_{\text{div-free}} = \frac{1}{8} \cdot \frac{1}{45} \cdot I_{\text{div-free}}$. All elements of $I_{\text{div-free}}$ are integers. The scaling by $1/360$ is split into two steps. The multiplication by $1/8$ is equivalent to shifting every coefficient 3-bit to the right; therefore, 3 extra bits for every coefficient are required. Since we use 16-bit anyway, this does not matter. The factor 45 has a multiplicative inverse for the Saber coefficient ring: $45^{-1} \equiv 4005 \pmod{2^{13}}$. An additional scaling step is required to multiply every coefficient with 4005 and shift it 3-bit to the right.

Ring-Splitting Multiplication

If we choose $t = 16$, we can directly use the algorithm for ring-splitting multiplication as described in Sect. 5. In this case, the nega-cyclic convolution of two polynomials of degree less than $n = 256$ is mapped to $t^2 = 256$ nega-cyclic convolutions of polynomials of degree less than $n/t = 16$. Since we chose $N = 16$, we can handle convolutions of such polynomials directly in hardware. The mapping Ψ is implemented, s.t. two 16-bit coefficients can be loaded with one 32-bit word.

Results

Table 2 shows the results for one polynomial multiplication in $\mathbb{Z}_q[X]/(X^n + 1)$. The ring-splitting multiplication outperforms all other approaches. It requires by far the lowest amount of stores. Of the 4352 loads and 512 stores, 256 loads and 384 stores are needed for the mapping Ψ. The 1-level Karatsuba instance performs slightly better than simple schoolbook multiplication as its straightforward approach of trading multiplications for additions can be implemented efficiently. Adding a second layer of Karatsuba or using Toom–Cook-4-way increases the linear overhead of pre- and postprocessing, s.t. performance deteriorates. All optimized approaches outperform the reference implementation by one order of magnitude. None of our implementations require branching on secret data and our hardware operations require a fixed number of clock cycles. Therefore, all our algorithms run in constant time.

6.2 Matrix–vector multiplication in Saber

Saber's public matrix A has the quadratic form $l \times l$ and the vector s is of length l. The result $A \cdot s$ is also a vector of length l. To compute one element of the resulting vector, l polynomial multiplications and additions are required. The additions are accelerated with the XSMUL's vector mode. Furthermore, we can exploit the matrix–vector struc-

Table 2 Results for one polynomial multiplication + reduction in Saber

Implementation	Latency (CPU cycles)	Nr. of loads	Nr. of stores
Ring-Splitting	11,190	4352	512
Karatsuba-1-Level	12,200	4220	2432
Schoolbook	12,530	4352	2432
Karatsuba-2-Level	14,310	4800	2816
Toom–Cook-4-way	17,216	5664	2144
Reference[a]	∼105,000	–	–

[a] Reference implementation of Saber from PQClean [28]. The multiplication algorithm relies on Toom–Cook-4-way, followed by Karatsuba multiplication. No instruction set extensions are used

ture for optimizations of polynomial multiplication with Karatsuba/Toom–Cook and ring-splitting.

Optimized ring-splitting

The optimizations for ring-splitting are straightforward. The mapping Ψ (i.e., the memory reordering) only needs to be applied once for each input and result polynomial, i.e., l^2+2l times. A naive application of the polynomial multiplication method in Algorithm 1 to the matrix–vector multiplication would lead to $3l^2$ applications of Ψ. The optimization requires additional memory for two polynomials in their reordered forms.

Time-memory trade-off in Toom–Cook multiplication

The authors of [27] present a time-memory trade-off for Toom–Cook multiplication. They establish evaluation and interpolation as linear transforms $\overline{TC}$, $\overline{TC}^{-1}$ and propose to apply $\overline{TC}$ only once for each input polynomial and $\overline{TC}^{-1}$ only once for each result polynomial. The memory trade-off is due to the more expensive storage of polynomials in their evaluated/interpolated forms. This lazy interpolation technique poses a significant advantage for Toom–Cook, in particular for higher dimensions l. However, a detailed performance analysis shows that for the maximum value $l = 4$, ring-splitting is still the most efficient multiplication method on our platform. It also requires less stack memory than matrix–vector optimized Toom–Cook.

6.3 Polynomial sampling in Saber

Besides polynomial multiplication, polynomial sampling is the largest performance bottleneck in lattice-based cryptography. The accelerators proposed in [11] include instruction set extensions for pseudo random number generation and binomial sampling to address this problem. As [11] was used as basis for this work, it is possible to use their extensions combined with the proposed XSMUL extension.

The Keccak accelerator from [11] computes a complete 1600-bit round permutation in a single CPU cycle. The cycle count of sampling a uniform random polynomial in Saber was improved by a factor of approximately 30.

In a single CPU cycle, the binomial sampling unit from [11] produces two 16-bit samples from four 16-bit uniform random samples with modular Hamming weight subtraction. The cycle count of sampling a Saber polynomial from a binomial distribution was improved by a factor of approximately 40.

7 Application to X25519

We propose an efficient implementation for the scalar multiplication in X25519 [7]. Arithmetic operations are defined in the finite field $\mathbb{F}_p$, where $p = 2^{255} - 19$, i.e., 255-bit integer arithmetic is required. The XSMUL architecture with $w = 17$-bit, $w_2 = 48$-bit (Sect. 3.2) and $N = 16$ (Sect. 4) enables direct 272-bit addition, subtraction, and multiplication. All integers $r \in \mathbb{F}_p$ are represented as 8-coefficient polynomials $r(X = 2) = r_0 + r_1 2^{32} + \cdots + r_7 2^{224}$, so one coefficient r_i is 32-bit wide.

Modular reduction

Pseudo-Mersenne prime reduction is used to reduce modulo p. Between operations on r, a weak reduction is performed, s.t. $r' < 2p$. We use the pseudo-Mersenne prime reduction with p to do this. Often $2p = 2^{256} - 38$ is used, as it allows to split $r = r_h \cdot 2^{256} + r_l$, but then $r' = 38 \cdot r_h + r_l$ has to be calculated more than twice in a row for $r' < 2p$ to hold. If p is used, two multiply-and-add steps suffice, s.t. $r' < 2p$ holds for multiplication results of two 256-bit integers. For our platform this lazy reduction approach has proven to be the most efficient implementation. Complete reduction is only carried out at the end of the ladder. One multiply-and-add step is directly integrated into the multiplication. The following arithmetic operations are required in X25519:

- **Addition/subtraction** Operands for addition/subtraction are split into 48-bit chunks. Subtraction $a-b$ is calculated as $2p - b + a$ to avoid underflows. Reduction to a value $< 2p$ is performed as described above.
- **Multiplication** The XSMUL supports integer multiplication with integrated pseudo-Mersenne prime reduction.

Table 3 Cycle count for arithmetic operations in $\mathbb{F}_{2^{255}-19}$ for an optimized implementation using instruction set extensions and a portable C implementation from libsodium [29]

Implementation	Inv.	Mul.	Mul. by word.	Add.	Sub.
Optimized	19,349	87	64	68	84
Reference [29]	343,334	1691	278	42	42

The operation in hardware yields a 263-bit integer. Repeating the reduction procedure leads to a result $< 2p$.

- **Multiplication by a 32 bit word** The Montgomery Ladder includes a multiplication with the constant 121665.
- **Inversion** The most expensive arithmetic operation is the calculation of an inverse field element $r^{-1} \equiv r^{p-2} \pmod{p}$. The standard procedure of 254 squarings and 11 multiplications is used to compute r^{p-2} [7]. Squaring can be chained together by barrel-shifting a reduced result $r^2 \pmod{p}$ into the XSMUL input registers and continue multiplying it with itself to get $r^4 \pmod{p}$ and so on.

Results

Table 3 shows the latencies for the respective arithmetic operations. The software implementation used for reference [29] relies on a different integer representation that allows carry-free additions and subtractions and does not require a direct reduction after addition/subtraction. Switching between representations is difficult, so XSMUL-accelerated multiplication cannot be combined with the addition and subtraction from [29]. Multiplications with XSMUL instruction set extensions are faster by multiple orders of magnitude. All latencies include fetching operands from memory at the beginning and writing them back at the end. In between ladder steps we do conditional constant-time swaps of complete field elements instead of just swapping pointers. Equivalent to our results for Saber our scalar multiplication runs in constant time.

Code size/ cycle count trade-off

A constant-time Montgomery ladder is used for scalar multiplication. If performance is prioritized over code size, loads and stores can be merged directly into the ladder code. This increases code size, but avoids unnecessary memory access between subsequent operations. For our architecture, this is feasible, since it includes enough registers to buffer values and can exploit barrel-shifting to enhance register transfer operations. Our optimized general-purpose implementation takes approximately 457,440 cycles and has a code size of 4,050 Byte. Approximately 600 Byte in code size can be traded for a speed-up of 55,000 cycles, if an implementation is strictly cycle count optimized.

8 Results and evaluation

We instantiate the PULPino with extensions for Saber and X25519 in a prototype FPGA implementation. The following subsections show resource utilization, latency, and code size results. State-of-the-art software and hardware/software codesign implementations are chosen for comparison. We discuss the applicability of our architecture to other primitives and hybrid key encapsulation.

8.1 FPGA resource utilization results

For evaluation of our architecture, we chose a Xilinx Zynq-7000[2] and UltraScale+ SoC.[3] Table 5 shows the resource utilization of the PULPino core with and without extensions. Default synthesis and implementation settings are used. The PULPino baseline does not include the FPU and the 32 floating-point registers. Our extended PULPino includes the Keccak, binomial sampling, and XSMUL extension and the 32 floating-point registers, but not the FPU itself.

On an UltraScale+ FPGA, our extensions cost 8,092 LUTs, 1,152 FFs, and 16 DSP slices. The overhead in registers can be mainly attributed to the 32 floating point registers; the 16 additional DSP slices are required for the XSMUL.

A direct comparison with the extended PULPino systems from [11] and [12] is not applicable, since the respective use cases differ. Instead of the XSMUL, the PULPino in [11] features a tightly coupled NTT accelerator for Kyber and NewHope, and modular multiply & accumulate extensions for Saber. The Keccak and binomial sampling accelerators are reused for our work. The authors of [12] use a loosely coupled NTT module that can support Kyber, NTRU, and Saber and similar Keccak and binomial sampling architectures. They show how masking increases the cost of accelerators, but the provided values are without masking. In general, [11] and [12] provide more flexibility in choosing a lattice-based PQC scheme, but do not support contemporary cryptography.

Due to a higher utilization, the routing gets more complex and the maximum possible frequency deteriorates a bit (33% and 45% of LUT logic are used for the UltraScale+ and Zynq-7000 SoC, respectively).

A deeper insight into the cost of individual extensions and their critical paths is provided in Table 4. It shows the utilization for a stand-alone implementation of the accelerators. This includes the cost for the connection to CPU resources, which is merged in the complete design, but not the cost for the registers themselves. Keccak and XSMUL require a

[2] ZedBoard with XC7Z020-CLG484 part: https://reference.digilentinc.com/programmable-logic/zedboard/start?redirect=1.

[3] Ultra96-V2 with UltraScale+ MPSoC ZU3EG A484: https://www.avnet.com/wps/portal/us/products/new-product-introductions/npi/aes-ultra96-v2/.

Table 4 Stand-alone utilization results for Keccak and binomial sampling unit from [11] and XSMUL extension

	Arch.	Utilization				f_{max} (MHz)
		LUTs	FFs	Slices[a]	DSPs	
Keccak [11]	US+	2,595	0	364	0	180
	Z-7000	2,599	0	705	0	85
Bino. Sampl. [11]	US+	130	0	21	0	210
	Z-7000	139	0	48	0	123
XSMUL	US+	3523	64	594	16	136
	Z-7000	4144	848	1244	16	45

[a] CLB on UltraScale+ architectures

Table 5 FPGA utilization results (placed and routed) for PULPino implementations with and without ISA extensions

	ISA ext.		Architecture	Utilization					f_{max} (MHz)
	PQC	ECC		LUTs	FFs	Slices[a]	DSPs	BRAMs	
PULPino	N	N	UltraScale+	15,103	9,909	3,149	6	32	83
			Zynq-7000	15,271	9,925	5,418	6	32	39
[11]	Y	N	Zynq-7000	23,947	10,847	–	21	32	–
[12]	Y	N	Artix-7	20,697	11,833	6,852	13	36.5	62.5
This work	Y	Y	UltraScale+	23,441	11,094	4386	22	32	76
			Zynq-7000	24,235	11,863	7658	22	32	34

This work includes the initial PULPino extended with the XSMUL and Keccak and binomial sampling accelerators proposed in [11]
[a] CLB on UltraScale+ architectures

significant amount of LUTs for round permutation and data path control logic, respectively. The XSMUL's register cost depends on the FPGA platform (Sect. 3), but does not change by much.

A logic path length analysis shows that paths inside the accelerators do not limit the clock frequency. The high difference of f_{max} for the XSMUL on different FPGA architectures is due to the advantageous DSP48E2 mapping, where all data paths inside a multiply & accumulate unit are mapped to short DSP slice internal paths.

It can be seen that the clock frequency is highly dependent on the underlying technology. Since our extensions do not limit the clock frequency, we decide to use cycle counts for the following evaluation of Saber and X25519.

8.2 Results for Saber

Table 6 shows the latencies of algorithmic steps in key enclosure with CCA secured Saber (NIST Level III). The work in [12] is also based on [11] (see above) and consumes fewer hardware resources. Our approach has the advantage of supporting contemporary schemes like X25519, while the NTT module in [12] allows implementers to switch directly between different lattice-based schemes.

RISQ-V [11], the basis of this work, includes a vectorized modular multiply & accumulate instruction to enhance the standard Toom–Cook implementation. It allows to execute two 16-bit computations of $a \cdot b + c \pmod q$ at the same time. This approach requires significantly fewer hardware resources but offers only a slight performance improvement for multiplications.

Compared to a software-only implementation on the PULPino, our architecture improves the performance by factor ten. The ARM Cortex M4 assembly optimized implementation in [30] uses an NTT-approach, outperforming the Toom–Cook approach from [27]. Our implementation is faster by a factor of three.

The ESP32 implementation of CPA secured Saber from [9] uses the accompanying RSA coprocessor for multiplications. Our extended PULPino implementation is three to four times faster than their work. We want to note that comparisons to fundamentally different platforms are not directly applicable, but still provide value in allowing the reader to judge the state-of-the-art.

The authors of [19] propose a loosely coupled Toom–Cook-4-way multiplier that supports our argument for tightly coupled accelerators. While their coprocessor only needs 8,176 cycles for multiplication, the data transfers to/from the CPU raise the latency to approximately 41,000 cycles. The advantage of the loosely coupled coprocessor is that multiplication and generation of polynomials can be interleaved. The loosely coupled accelerator can not reuse hardware resources in the CPU and has a similar hardware overhead to our work.

Table 7 includes a comparison with the standalone hardware implementations for UltraScale+ FPGA architecture in [31, 32] and the ASIC in [33]. We should note that a direct

Table 6 Cycle count for Key Encapsulation in CCA secured Saber (compiled with $-O3$, GCC PULPino RISC-V compiler 7.1.1 20170509)

	Type	Cycle counts (·1000)			Overhead			Frequency
		Key Gen.	Encaps.	Decaps.	LUTs	FFs	DSPs	
This work[a]	HW/SW	217	279	300	8337	1185	16	76 MHz (−8%)
ext. PULPino[a] [12]	HW/SW	229	308	347	2454	1917	7	56 MHz (−6%)
ext. PULPino[a] [11]	HW/SW	761	1000	1201	9050	1268	12	–
PULPino [28]	SW	2110	2737	2797	–	–	–	–
Cortex-M4 [30]	SW	658	864	835	–	–	–	–
ESP32 + Co. [9]	HW/SW	827	1070	243[b]	–	–	–	–
Z-7 SoC [19]	HW/SW	2180	2762	2560	7400	7331	28	125 MHz
US+ x1 [31]	HW	2.7	3.7	4.7	21,352	14,232	0	370 MHz
US+ x2 [31]	HW	1.8	2.2	2.8	32,099	21,037	0	345 MHz
US+ x4 [31]	HW	1.3	1.5	1.9	48,895	27,715	0	310 MHz
US+ [32]	HW	5.5	6.6	8.0	25,079	10,750	0	250 MHz
TSMC 40 nm [33]	HW	1.1	1.5	1.7	28,169	9504	85	160 MHz

[a] Utilization given as diff. to orig. PULPino
[b] Only CPA secured variant

Table 7 Code size for Saber CCA in byte (compiled with $-O3$, GCC PULPino RISC-V compiler 7.1.1 20170509)

	ISA	Code size
This work	RISC-V	10,364
PULPino + Ext. [12]	RISC-V	10,900
PULPino + Ext. [11]	RISC-V	11,802
PULPino Ref. [28]	RISC-V	17,912
ARM Cortex M4 [34]	ARM	9412

comparison with those design is not applicable. All these designs consume more hardware resources than the complete, extended PULPino core in this work without featuring a general purpose CPU that could implement a protocol on top of the cryptographic operations. However, the cycle count of the standalone hardware implementations is far lower than those of hardware/software co-designs (Table 8).

Table 7 shows code sizes for the respective implementations. All three optimized RISC-V implementations require a similar amount of memory to store their machine code. Compared to the reference implementation, complex operations are commenced with a few instructions, therefore less code is required. The ARM Cortex M4 implementation requires slightly less memory, due to the advantageous code size of the ARM ISA [42].

Our hardware extension does not change the maximum stack usage, which is at 16,644 bytes for Saber-CCA.

8.3 Results for X25519

Table 9 shows the latencies for the complete scalar multiplication and the required arithmetic operations in $\mathbb{F}_{2^{255}-19}$ on ARM architectures and the extended PULPino design. The below-average addition/subtraction performance was explained in Sect. 7. The XSMUL's multiplication performance is beyond that of the superior ARM A7, A8, and A15 processors, which feature vector extensions. Compared to the ARM Cortex M4, latency is lower by factor 1.5 if the core frequency is reduced or a flash pre-fetch engine is used, and factor two otherwise. The PULPino software-only reference implementation is outperformed by factor ten. In [41], the author optimizes his implementation to only use the RISC-V 32-bit basis instruction set and the multiply extension. The standalone hardware designs in [36–38] are more powerful in terms of latency, but require more FPGA resources. The hardware/software codesign in [35] shows that an extension for X25519 can be realized with a more compact overhead than our hybrid approach. However, it should be noted that their system-on-chip already includes a powerful general purpose CPU.

Compared to an unoptimized reference implementation, the usage of XSMUL instruction set extensions improves the code size approximately by factor three (Table 10). Without the cycle count/ code size trade-off discussed in Sect. 7, the code size of the proposed implementation is comparable to [40].

Similar to our Saber implementation, the stack usage is unchanged by our hardware extension and is at 788 bytes.

8.4 Design exploration

In the following, we describe the applicability of our extension to other cryptographic schemes. We start by evaluating other lattice-based proposals and consider two groups: (i)

Table 8 Cycle count for scalar multiplication in X25519

	Type	Scalar Mul.	Inv.	Mul.	Add.	Sub.
This work	HW/SW	411,810	19,350	87	68	84
Zynq SoC [35]	HW/SW	135,446	18,489	33	17	17
Zynq [36]	HW	13,639	2928	10	2	2
Zynq [37]	HW	10,465	2548	8	2	2
Zynq MC [38]	HW	34,052	1667	55	10	10
Zynq SC [38]	HW	79,400	14,630	55	10	10
ARM C. M4 [39]	SW	609,779–971,272[a]	42,590	153–237	52–95	65–124
ARM C. M4 [40]	SW	894,391	64,425	273	86	86
ARM C. A7 [40]	SW	825,914	62,648	290	52	52
ARM C. A15 [40]	SW	572,910	41,978	225	36	36
PULPino Ref. [29]	SW	4,103,653	343,334	1691	42	42
RISC-V Hifive1 [41]	SW	4,432,988	–	–	–	–

[a] Depends on the frequency and the usage of a flash pre-fetch-engine

Table 9 Resource utilization for hardware/software and hardware designs of X25519

	Type	Overhead			Frequency
		LUTs	FFs	DSPs	
This work[a]	HW/SW	8337	1185	16	76 MHz (−8%)
Zynq SoC [35]	HW/SW	2707	962	15	105 MHz
Zynq [36]	HW	21,107	26,483	260	115 MHz
Zynq [37]	HW	17,939	21,077	175	115 MHz
Zynq MC [38]	HW	43,675	34,009	220	100 MHz
Zynq SC [38]	HW	3592	2783	20	200 MHz

[a] Utilization given as diff. to orig. PULPino

Table 10 Code size for X25519 in byte (compiled with $-O3$, GCC PULPino RISC-V compiler 7.1.1 20170509)

	ISA	Code size
This work	RISC-V	4628
PULPino Ref [29]	RISC-V	14,768
ARM Cortex M4 [39]	ARM	3324
ARM Cortex M4 [40]	ARM	4152

schemes that do not inherently require the NTT and use a coefficient ring allowing easy reductions, and (ii) schemes that inherently require the NTT.

Different n and N_{slice} for non-NTT, coefficient-reduction-friendly polynomial multiplication

NTRU is similar to Saber in the fact that the main computational cost lies in polynomial multiplication and the coefficient reduction is virtually free. Polynomials in NTRU are defined in the ring $\mathbb{Z}_q[X]/\langle\Phi_n(X)\rangle$. The polynomial length is $n \in [509, 677, 701, 821]$, and coefficients are in the ring $\mathbb{Z}_q$, where q is either a power of two $\leq 2^{13}$ or 3 [6]. We evaluate the multiplication algorithms from Sects. 5 and 6 for other values n.

The other dimension of interest is the number of multiply & accumulate units N. We chose $N = 16$, but designers might prefer a lower or higher number, depending on the use case.

Our implementations are transformed into two-dimensional cost functions latency$(n)(N)$. Extrapolated values can be found in Table 11. We make a couple of interesting observations. Had we chosen $N \in [4, 8]$, Toom–Cook or Karatsuba would have been the better choice for the Saber implementation, even more so if matrix–vector optimizations are considered. If operations are not efficient enough in hardware, the advantageous time complexity of complicated algorithms is more important than their overhead. In NTRU, the prime values of n make the application of ring-splitting difficult. Coefficient reduction modulo 3 requires more effort than the bit masking for powers of 2 but can be integrated into our architecture. For the proposed design with $N_{\text{slice}} = 16$, 2-level Karatsuba should be chosen for NTRU if $n = 509$, and Toom–Cook-4-way for all other values of n.

Lattice-based cryptography with inherent NTT usage

Supporting Kyber [43] imposes two problems: (i) It inherently uses the NTT, i.e., it sends/receives data in NTT-representation, and (ii) coefficient reduction is a computationally expensive operation. The authors of [8] solve the first

Table 11 Extrapolated values from the two-dimensional cost function latency$(n)(N_{\text{slice}})$ for the XSMUL extension

	$N_{\text{slice}} = 4$				$N_{\text{slice}} = 8$			
	$n = 256$	$n = 512$	$n = 768$	$n = 1024$	$n = 256$	$n = 512$	$n = 768$	$n = 1024$
Ring-splitting	89,300	350,500	783,700	1,389,000	28,200	107,600	238,200	420,000
1L-Karatsuba	65,200	247,100	545,600	960,900	25,700	92,600	201,100	351,000
Schoolbook	80,300	316,200	707,700	1,254,900	29,500	114,200	254,200	449,600
2L-Karatsuba	57,500	202,000	434,000	753,600	25,300	81,200	168,200	286,300
Toom–Cook-4-way	58,700	185,200	379,800	642,400	27,700	79,200	154,900	254,800
	$N_{\text{slice}} = 16$				$N_{\text{slice}} = 32$			
	$n = 256$	$n = 512$	$n = 768$	$n = 1024$	$n = 256$	$n = 512$	$n = 768$	$n = 1024$
Ring-splitting	11,000	38,400	82,900	144,300	5300	16,800	34,600	58,700
1L-Karatsuba	12,200	40,800	85,800	147,400	7100	21,200	42,500	71,000
Schoolbook	12,500	47,000	103,500	182,000	6100	21,800	47,200	82,100
2L-Karatsuba	14,300	39,900	78,100	128,700	9600	23,900	43,700	68,900
Toom–Cook-4-way	17,200	41,000	75,700	120,100	10,900	25,500	44,400	67,500

problem by modifying Kyber, s.t. data is not sent in its NTT-representation. Solving the second problem would require significant changes to our design. Multiplication modulo a prime q is usually computed with Montgomery modular multiplication [44] to avoid computationally expensive divisions. Therefore, we would have to extend our design with Montgomery multipliers, which have a significant overhead compared to the multipliers currently used. It is unlikely that extending our design with Montgomery multiplication is reasonable compared to state-of-the-art NTT implementations. The same holds for the signature standards Falcon and Dilithium.

Other cryptographic primitives

Our design is capable of handling the matrix–vector multiplication of the unstructured lattice-based scheme Frodo [45]. Frodo's matrix coefficients fit into w-bit and the modulus q also allows virtually free reductions. Multiplications in NTRU Prime can be supported, but implementing the reduction algorithm on our platform requires further evaluation. Our architectural approach is applicable to other curves besides Curve25519. For Curves using Mersenne primes such as the NIST curve P-521, the adoption is straightforward. The XSMUL's width would have to be adjusted to support field elements of 521-bit and the cycle count of the integer multiplication with integrated reduction would accordingly double. For curves using Solinas primes like P-192, P-224, P-256 and P-384, modular reduction is more complicated. Therefore, the best approach would be to use the generalized integer multiplication and use the XSMUL's vectorization capabilities to integrate modular reduction suitable for those primes. For Solinas primes, a set of fast additions and subtractions can be used. Schemes like SIKE and RSA require integer arithmetic of more than 272-bit and typically use Montgomery's modular multiplication algorithm [44]. Only intermediate improvements can be expected, if the XSMUL is used.

8.5 Hybrid key encapsulation

We propose our architecture as basis for integration of hybrid key encapsulations into embedded systems. To get from our Saber and X25519 implementations to a secured handshake between two parties, hybrid handshakes need to be standardized. This is an active research topic. The topic of negotiation is reviewed in [46], the topic of combiners for multiple secrets in [2]. Robust combiners often require hash functions, which could potentially be supported by the Keccak extension we adapt from [11]. Particularly interesting for embedded systems is the KEMTLS handshake from [1]. The complete authenticated handshake is built solely with KEMs. Signatures are only used to prove the authenticity of a certificate, which often can be preinstalled in embedded applications. During the KEMTLS handshake, the client would have to perform one key generation, one decapsulation, and one encapsulation. In a hybrid handshake, each of those operations would be needed for the contemporary and the PQC standard. KEMTLS reduces the amount of necessary communication significantly, and does not require support for signatures on the embedded device. Both are desirable for the use case we consider for our architecture.

In recent works, contemporary accelerators have been used for PQC [8–10]. These efforts are crucial so that legacy hardware can stay protected in a post-quantum age. However, this does not mean that big-integer coprocessors are the best architecture for both polynomial and integer multiplication and consequently the best choice for new designs. The mapping of polynomials to integers implies an unavoidable

overhead. Our architecture handles both operations natively. A new design that considers both contemporary cryptography and PQC should be implemented, s.t. both are supported equally efficient. Our results show that this is the case for our design. A direct comparison with [8] and [9] would be unfair due to differences in the used platforms.

8.6 Side channel security

All our implementations run in constant time. The instruction set extensions we proposed need a constant cycle count and we integrated these instructions into algorithms such as our ring-splitting multiplication and a constant-time Montgomery ladder. We did not consider any countermeasures against power or electromagnetic based side-channel attacks. Therefore, our implementation is as vulnerable to these attacks as a generic unhardened software implementation of Saber and X25519 without instruction set extensions.

For lattice-based PQC, masking and shuffling are the most prevalent countermeasures against power and electromagnetic side channel attacks [12]. Masking, while widely used, seems to be vulnerable to horizontal template attacks, as the polynomial arithmetic is a long linear operation without mask refreshing for arithmetically masked polynomials. Shuffling could be more efficient, as it eliminates the temporal connection of operations. Applying both countermeasures to our extensions would not differ much from applying them to a generic software implementation.

For elliptic curve cryptography, scalar blinding can be used to avoid leakage of secret information over the power and electromagnetic side channel [47]. In this case, the applicability of our architecture depends on the chosen blinding factor.

9 Conclusion and future work

Our extended PULPino core supports efficient multiplication of polynomials and big integers. This enables high performance in a hybrid handshake with a contemporary and a PQC primitive. Our prototype implementation demonstrates this with Saber and X25519. Without our extension, the scalar multiplication in X25519 alone would take longer than optimized encapsulation/decapsulation with Saber and X25519. With 16 additional DSP slices and factor 1.5 more LUT logic, the overhead to achieve this on a FPGA platform is feasible for embedded devices. We show that the XSMUL supports a variety of multiplication algorithms. Our memory optimized ring-splitting multiplication is the best choice for Saber, whereas for NTRU, Toom–Cook-4-way would be the rational choice. The XSMUL itself is also scalable and can be modified to fit different use cases. Our platform is the first of its kind and presents an alternative to relying on loosely coupled, contemporary accelerators for polynomial multiplication.

For our work, we see three important areas left open for future efforts. First, it is important to harden our instruction set extensions against side-channel attacks. Second, it needs to be evaluated how we can port our design efficiently for ASIC implementations. Third, a uniform architecture for NTT and big-integer arithmetic would allow to combine all lattice-based schemes with contemporary primitives and provide an interesting comparison to our architecture.

Acknowledgements Funding was provided by the German Bundesministerium für Bildung und Forschung (Grant No. 16KIS1017K and 16KIS1018 Aquorypt).

Funding Open Access funding enabled and organized by Projekt DEAL.

Appendix A: Mapping of XSMUL slices to DSP slices

In Sect. 3.2 of our paper, we emphasized that our design maps well to DSP slices and manual optimizations are required to achieve an optimal utilization. In the following, this mapping to Xilinx DSP48E2 and DSP48E1 slices is shown. Detailed information on the configuration possibilities can be found in the respective Xilinx User Guides [48, 49].

A.1 DSP48E2 mapping

For DSP48E2 slices, the complete multiply& accumulate unit required for the XSMUL can be mapped into one slice. The used resources are marked in red (Fig. 4).

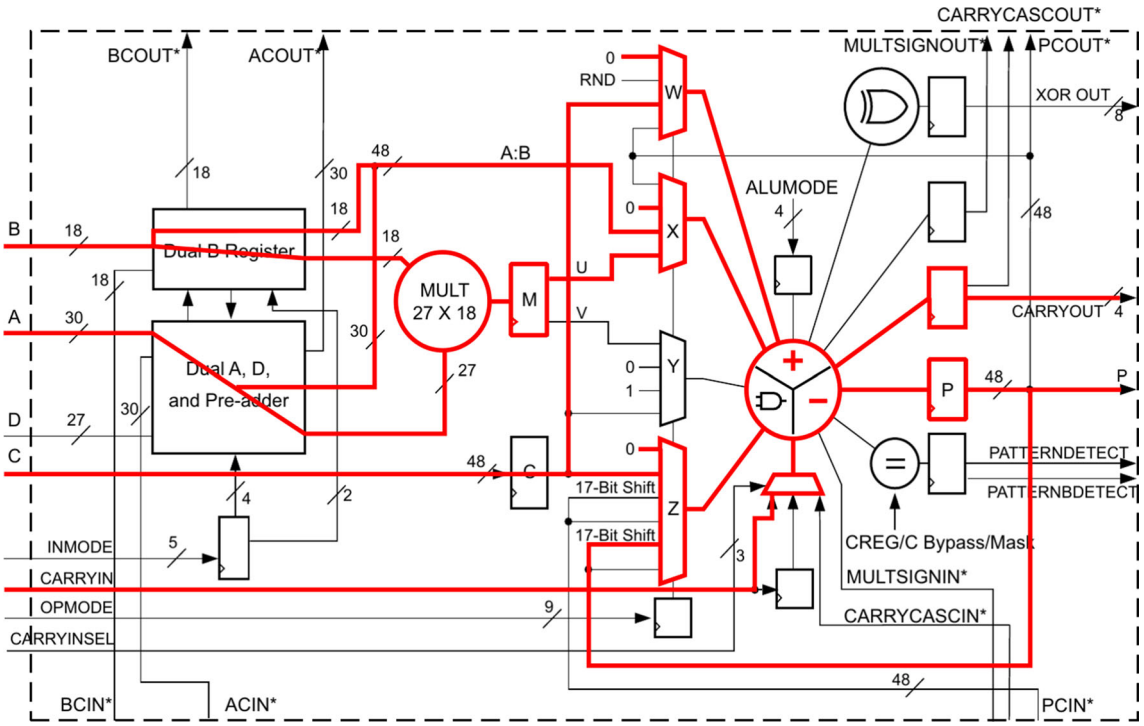

Fig. 4 Used resources in a DSP48E2 slice, adapted from [48]

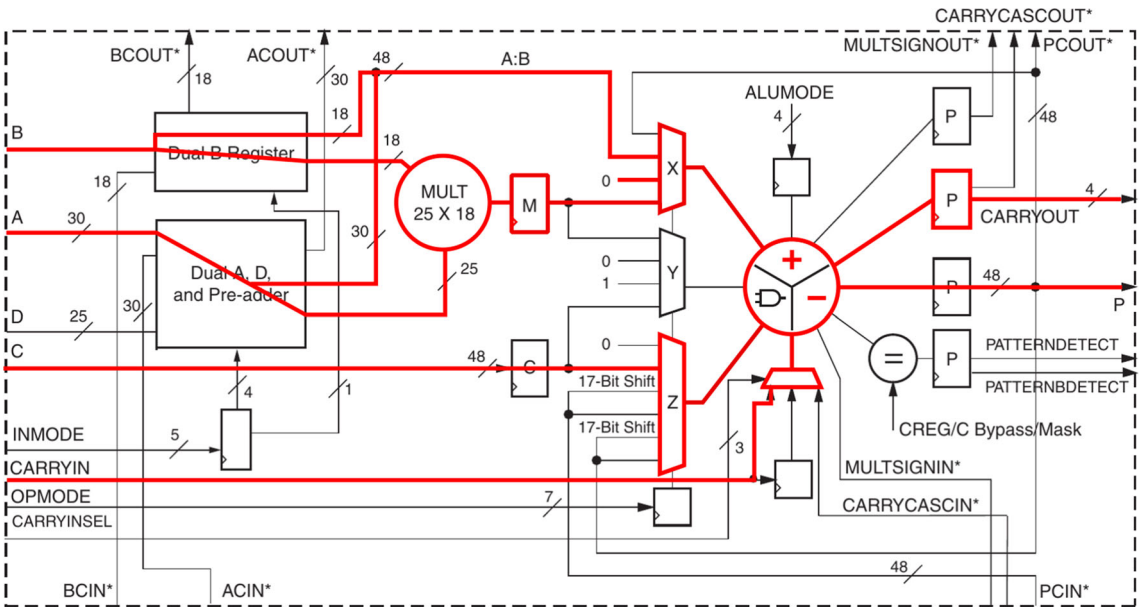

Fig. 5 Used resources in a DSP48E1 slice, adapted from [49]

A.2 DSP48E1 mapping

The multiplication carry path and output register are implemented outside of the DSP48E1 slice in LUT logic, as the central adder has one input less than the adder in DSP48E2 slices. The addition carry path and its register are mapped into the slice. The used resources are marked in red (Fig. 5).

Appendix B: Operation trade-off in (multi)-level Karatsuba and Toom–Cook-4-way

Table 12 shows the operations required for the schoolbook multiplication, 1/2-level Karatsuba, and Toom–Cook-4-way implementations that were proposed for Saber (256-coeff polynomials) in Section VI-A. Polynomial multiplications

Table 12 Operations for Schoolbook, 1/2-level Karatsuba and Toom–Cook-4-way multiplication in Saber

Algorithm	Polynomial multiplications	Polynomial additions/subtractions
Schoolbook	1× *256-coeff polynomials*	–
1-level Karatsuba	3× *128-coeff polynomials*	2× *256-coeff polynomials*
		4× *128-coeff polynomials*
2-level Karatsuba	9× *64-coeff polynomials*	2× *256-coeff polynomials*
		10× *128-coeff polynomials*
		12× *64-coeff polynomials*
Toom–Cook-4-way[a]	7× *64-coeff polynomials*	27× *128-coeff polynomials*
		21× *64-coeff polynomials*

[a] Requires an additional scaling step with bit-shifts and scalar multiplications

are traded for pre- and post-processing consisting of polynomial additions and scalar multiplication (for Toom–Cook-4-way).

References

1. Schwabe, P., Stebila, D., Wiggers, T.: Post-quantum TLS without handshake signatures. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS'20). Association for Computing Machinery, New York, NY, USA, pp. 1461–1480 (2020). https://doi.org/10.1145/3372297.3423350
2. Bindel, N., Brendel, J., Fischlin, M., Goncalves, B., Stebila, D.: Hybrid key encapsulation mechanisms and authenticated key exchange. In: Ding, J., Steinwandt, R. (eds.) Post-Quantum Cryptography, pp. 206–226. Springer, Cham (2019)
3. BSI: BSI-Technical Guideline: Cryptographic Mechanisms: Recommendations and Key Lengths. (Retrieved: 20.04.2021). https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf?__blob=publicationFile
4. Chen, L., Moody, D., Yi-Kai, L.: Post Quantum Cryptography—FAQ (Retrieved: 23.06.2021). https://csrc.nist.gov/Projects/post-quantum-cryptography/faqs
5. Basso, A., Bermudo Mera, J.M., D'Anvers, J.P.: SABER: Mod-LWR based KEM (Round 3 Submission). https://www.esat.kuleuven.be/cosic/pqcrypto/saber/files/saberspecround3.pdf
6. Chen, C., Danba, O., Hoffstein, J., et al.: NTRU—Algorithm Specifications and Supporting Documentation. https://ntru.org/f/ntru-20190330.pdf
7. Bernstein, D.J.: Curve25519: new Diffie–Hellman speed records. In: Public Key Cryptography—PKC 2006, pp. 207–228. Springer, Berlin (2006)
8. Albrecht, M.R., Hanser, C., Hoeller, A., Pöppelmann, T., Virdia, F., Wallner, A.: Implementing RLWE-based schemes using an RSA co-processor. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2019**(1), 169–208 (2018). https://doi.org/10.13154/tches.v2019.i1.169-208
9. Wang, B., Gu, X., Yang, Y.: Saber on ESP32. https://eprint.iacr.org/2019/1453. Cryptology ePrint Archive, Report 2019/1453
10. Bos, J.W., Renes, J., van Vredendaal, C.: Post-quantum Cryptography with Contemporary Co-Processors: Beyond Kronecker, Schönhage-Strassen & Nussbaumer. https://eprint.iacr.org/2020/1303. Cryptology ePrint Archive, Report 2020/1303
11. Fritzmann, T., Sigl, G., Sepúlveda, J.: RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. https://eprint.iacr.org/2020/446. Cryptology ePrint Archive, Report 2020/446
12. Fritzmann, T., Beirendonck, M.V., Roy, D.B., Karl, P., Schamberger, T., Verbauwhede, I, et al.: Masked Accelerators and Instruction Set Extensions for Post-quantum Cryptography. https://eprint.iacr.org/2021/479. Cryptology ePrint Archive, Report 2021/479
13. Kronecker, L.: Grundzüge einer arithmetischen Theorie der algebraischen Grössen. Journal für die reine und angewandte Mathematik **92**, 1–122 (2022)
14. Pollard, J.M.: The fast Fourier transform in a finite field. Math. Comput. **25**(114), 365–374 (2022)
15. Nussbaumer, H.: Fast polynomial transform algorithms for digital convolution. IEEE Trans. Acoust. Speech Signal Process. **28**(2), 205–215 (2022)
16. den Toom, A.L.: The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers
17. Karatsuba, A., Ofman, Y.P.: Multiplication of Many-Digital Numbers by Automatic Computers
18. Basso, A., Roy, S.S.: Optimized Polynomial Multiplier Architectures for Post-quantum KEM Saber. https://eprint.iacr.org/2020/1482. Cryptology ePrint Archive, Report 2020/1482
19. Maria Bermudo Mera, J., Turan, F., Karmakar, A., Sinha Roy, S., Verbauwhede, I.: Compact domain-specific co-processor for accelerating module lattice-based KEM. In: 2020 57th ACM/IEEE Design Automation Conference (DAC) pp. 1–6 (2020). https://doi.org/10.1109/DAC18072.2020.9218727
20. Liu, B., Wu, H.: Efficient architecture and implementation for NTRUEncrypt system. In: 2015 IEEE 58th International Midwest Symposium on Circuits and Systems (MWSCAS), pp. 1–4 (2015). https://doi.org/10.1109/MWSCAS.2015.7282143
21. Braun, K., Fritzmann, T., Maringer, G., Schamberger, T., Sepúlveda, J.: Secure and compact full NTRU hardware implementation. In: 2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), pp. 89–94 (2018). https://doi.org/10.1109/VLSI-SoC.2018.8645015
22. Güneysu, T., Paar, C.: Ultra high performance ECC over NIST primes on commercial FPGAs. In: Oswald, E., Rohatgi, P. (Eds.) Cryptographic Hardware and Embedded Systems—CHES 2008, pp. 62–78. Springer, Berlin (2008). https://iacr.org/archive/ches2008/51540064/51540064.pdf
23. Zhang, X., Parhi, K.K.: Reduced-complexity modular polynomial multiplication for R-LWE cryptosystems. In: ICASSP 2021—2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 7853–7857 (2021)
24. van der Lubbe, G.: A New Hope for Nussbaumer. https://www.cs.ru.nl/bachelors-theses/2016/Gerben_van_der_Lubbe___4389026___A_New_Hope_for_Nussbaumer.pdf
25. Lee, W., Akleylek, S., Wong, D.C., et al.: Parallel implementation of Nussbaumer algorithm and number theoretic transform on a GPU platform: application to qTESLA. J. Supercomput. **77**, 3289–3314 (2020)

26. Gu, Z., Li, S.: A division-free Toom–Cook multiplication-based montgomery modular multiplication. IEEE Trans. Circuits Syst. II Express Briefs **66**(8), 1401–1405 (2019). https://doi.org/10.1109/TCSII.2018.2886962
27. Bermudo Mera, J.M., Karmakar, A., Verbauwhede, I.: Time-memory trade-off in Toom–Cook multiplication: an application to module-lattice based cryptography. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(2), 222–244 (2020)
28. PQClean: PQClean. https://github.com/PQClean/PQClean
29. libsodium: libsodium. https://github.com/jedisct1/libsodium
30. Chung, C.M.M., Hwang, V., Kannwischer, M.J., Seiler, G., Shih, C.J., Yang, B.Y.: NTT multiplication for NTT-unfriendly rings: new speed records for Saber and NTRU on Cortex-M4 and AVX2. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(2), 159–188 (2021). https://doi.org/10.46586/tches.v2021.i2.159-188
31. Dang, V.B., Mohajerani, K., Gaj, K.: High-Speed Hardware Architectures and FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber. https://eprint.iacr.org/2021/1508. Cryptology ePrint Archive, Paper 2021/1508. https://eprint.iacr.org/2021/1508
32. Roy, S.S., Basso, A.: High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware. https://eprint.iacr.org/2020/434. Cryptology ePrint Archive, Paper 2020/434. https://eprint.iacr.org/2020/434
33. Zhu, Y., Zhu, M., Yang, B., Zhu, W., Deng, C., Chen, C., et al.: A High-performance Hardware Implementation of Saber Based on Karatsuba Algorithm. https://eprint.iacr.org/2020/1037. Cryptology ePrint Archive, Paper 2020/1037. https://eprint.iacr.org/2020/1037
34. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4
35. Turan, F., Verbauwhede, I.: Compact and flexible FPGA implementation of Ed25519 and X25519. ACM Trans. Embed. Comput. Syst. **18**(3), 66 (2019). https://doi.org/10.1145/3312742
36. Koppermann, P., De Santis, F., Heyszl, J., Sigl, G.: X25519 hardware implementation for low-latency applications. In: 2016 Euromicro Conference on Digital System Design (DSD) https://doi.org/10.1109/DSD.2016.65
37. Koppermann, P., De Santis, F., Heyszl, J., Sigl, G.: Low-latency X25519 hardware implementation: breaking the 100 microseconds barrier. Microprocess. Microsyst. **52**, 491–497 (2017). https://doi.org/10.1016/j.micpro.2017.07.001
38. Sasdrich, P., Güneysu, T.: Efficient Elliptic-Curve Cryptography Using Curve25519 on Reconfigurable Devices, pp. 25–36 (2014)
39. Haase, B., Labrique, B.: AuCPace: efficient verifier-based PAKE protocol tailored for the IIoT. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2019**, 1–48 (2019). https://doi.org/10.13154/tches.v2019.i2.1-48 https://doi.org/10.13154/tches.v2019.i2.1-48
40. Fujii, H., Aranha, D.F.: Curve25519 for the Cortex-M4 and beyond. In: Lange, T., Dunkelman, O. (Eds.) Progress in Cryptology—LATINCRYPT 2017, pp. 109–127. Springer, Cham (2019)
41. van den Berg, S.: RISC-V implementation of the NaCl-library. https://pure.tue.nl/ws/portalfiles/portal/169647601/Berg_S._ES_CSE.pdf
42. Perotti, M., Schiavone, P.D., Tagliavini, G., Rossi, D., Kurd, T., Hill, M., et al.: HW/SW Approaches For RISC-V Code Size Reduction
43. Bos, J., Ducas, L., Kiltz, E., et al.: CRYSTALS—Kyber: a CCA-secure module-lattice-based KEM. https://ia.cr/2017/634. Cryptology ePrint Archive, Report 2017/634
44. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**, 519–521 (1985)
45. Alkim, E., Bos, J.W., Ducas, L., Longa, P., Mironov, I., et al.: FrodoKEM Learning With Errors Key Encapsulation. https://frodokem.org/#spec
46. Crockett, E., Paquin, C., Stebila, D.: Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH. https://eprint.iacr.org/2019/858. Cryptology ePrint Archive, Report 2019/858
47. Fluhrer, S.: Scalar Blinding on Elliptic Curves based on Primes with Special Structure. https://eprint.iacr.org/2015/801. Cryptology ePrint Archive, Paper 2015/801. https://eprint.iacr.org/2015/801
48. Xilinx.: UltraScale Architecture DSP Slice—User Guide. (Retrieved: 20.04.2021). https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf
49. Xilinx.: 7 Series DSP48E1 Slice—User Guide. (Retrieved: 20.04.2021). https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf