High Level Synthesis in Implementing and Benchmarking Number Theoretic Transform in Lattice-based Post-Quantum Cryptography using Software/Hardware Codesign

No Author Given

No Institute Given

Abstract. Compared to traditional hardware development methodologies, High-Level Synthesis (HLS) offers a faster time-to-market and lower design cost at the expense of implementation efficiency. Although Software/Hardware Codesign has been used in many areas, its usability for benchmarking of candidates in cryptographic competitions has been largely unexplored. This paper provides a comparison of the HLS- and RTL-based design methodologies when applied to the hardware design of Number Theoretic Transform (NTT) – a core arithmetic function of lattice-based Post-Quantum Cryptography (PQC). As a next step, we apply Software/Hardware Codesign approach to the implementation of three PQC schemes based on NTT. Then, we integrate our HLS implementation into the Xilinx SDSoC environment. We demonstrate that an overhead of SDSoC compared to traditional Bare Metal approach is acceptable. This paper also shows that an HLS implementation obtained by modeling a block diagram is typically much better than an implementation obtained by using design space exploration. We conclude that the HLS/SDSoC and RTL/Bare Metal approaches generate comparable results.

1 Introduction

A threat of quantum computers triggered an effort aimed at designing a new class of cryptographic algorithms, collectively referred to as Post-Quantum Cryptography (PQC) [1]. These algorithms have two common features: a) there are no known attacks capable of breaking these cryptosystems, even assuming the availability of full-scale quantum computers, b) all PQC algorithms can be implemented using traditional computing platforms, based on standard semiconductor technology, such as microprocessors and FPGAs. In the standardization process currently run by the National Institute of Standards and Technology (NIST), 26 candidates remain in Round 2 and need to be evaluated from the point of view of their hardware efficiency [1]. These candidates represent 5 major families: lattice-based, code-based, hash-based, isogeny-based, and multivariate. A large number of candidates and high complexity of the majority of them make hardware benchmarking extremely challenging. In order to mitigate these difficulties, a new approach based on a) software/hardware codesign, and b) the

development of hardware accelerators using High-Level Synthesis (HLS) has been proposed [2].

Software/Hardware Codesign is widely used in academia and industry for many applications, but this approach has never been applied to benchmarking candidates in any past cryptographic competitions. We believe that it is the right time to apply this new methodology to benchmarking PQC, where hardware accelerators can be developed quickly using HLS. After developing HLS-ready C code, a designer is only one step away from creating SW/HW codesign. In the traditional RTL approach, a path from developing Hardware Description Language (HDL) code to running it on a target device is quite long, since the developer has to create an interface between CPU and FPGA. Additionally, various factors lead to changes in RTL design, which is very time-consuming.

On the other hands, the SDSoC framework uses abstract layers for developers to create SW/HW codesign, where most of the processes are automatically handled by the tool. Any changes in the HLS-ready C code are easy to incorporate thanks to the benefits of HLS. Therefore, using SDSoC with HLS minimizes the difficulties in implementing SW/HW codesign.

In 12 remaining KEM/Encryption and Signatures Round 2 lattice-based PQC candidates, 5 of them use Number Theoretic Transform (NTT) for polynomial multiplication. After software profiling, we decided to implement the NTT hardware accelerators for NewHope Kyber Round 2 (Kyber R2). We also showed that with the extended NTT hardware, Kyber Round 1 (Kyber R1) had similar performance to Kyber R2, but with smaller resource utilization.

Contribution. The paper demonstrates the following aspects of using the HLS/SDSoC combination versus the traditional RTL/Bare Metal approach

- The advantages of the HLS approach based on block diagrams vs. the HLS approach based on space exploration.
- The HLS vs. RTL implementation trade off.
- Overhead of SDSoC over the Bare Metal approach.
- Comparison of the total speed-up vs. software between HLS/SDSoC and RTL/Bare Metal.

2 Background

2.1 Number Theoretic Transform

Let n be a power of two, and q be a prime modulus. We define a ring $R_q[x] = Z_q[x]/(x^n + 1)$ as the ring of polynomials of degree $n - 1$ with coefficients in Z_q (a field of integers in the range $[0, q - 1]$ with addition and multiplication modulo q). Multiplications in $R_q[x]$ can be performed efficiently in software and hardware using the NTT, which has the complexity of $O(n \cdot log(n)$.

If $\psi^2 = \omega \bmod q$ exists, then it is recommended that the input polynomials should be multiplied by ψ^i before Forward NTT instead of supplementing them with n most significant terms equal to zero. As a result, the output of Inverse NTT must be multiplied by ψ^{-i}.

By using NTT, a multiplication in R_q can be computed as follows:

Algorithm 1 Iterative NTT

Require: $F(x) \in R_q[x]$; $ROM[i] = \omega^i$, $\omega^n = 1 \bmod q$
Ensure: $\overline{F}(x) = NTT(F)$
1: $F \leftarrow BitReverse(F)$
2: **for** $s = 0$ to $log_2(n) - 1$ by 1 **do**
3: $m \leftarrow 2 \ll s$
4: $\omega_m \leftarrow n/m$
5: $i \leftarrow 0$
6: **for** $j = 0$ to $m/2$ by 1 **do**
7: **for** $k = 0$ to n by m **do**
8: $u \leftarrow F[k + j]$
9: $t \leftarrow F[k + j + m/2] * ROM[i]$
10: $F[k + j] \leftarrow u + t$
11: $F[k + j + m/2] \leftarrow u - t$
12: $i \leftarrow i + \omega_m$

$$C = \mathbf{NTT}^{-1}(\overline{C}) = \mathbf{NTT}^{-1}(\overline{A} * \overline{B}) = \mathbf{NTT}^{-1}(\mathbf{NTT}(A) * \mathbf{NTT}(B))$$

where $\psi^2 = \omega$, and $A = (a_0, \psi a_1, \psi^2 a_2, \ldots, \psi^{n-1} a_{n-1})$,
$B = (b_0, \psi b_1, \psi^2 b_2, \ldots, \psi^{n-1} b_{n-1})$, $C = (c_0, \psi c_1, \psi^2 c_2, \ldots, \psi^{n-1} c_{n-1})$. a, b, c are polynomials in $R_q[x]$, with $q = 1 \bmod 2N$ [3].

The pseudo-code of the iterative version of Forward NTT is shown in Algorithm 1. The Inverse NTT is similar to Forward NTT but instead of multiplying with ω^i, we multiply with ω^{-1}.

In this paper, we divide the polynomial multiplication into two modes:

1. **NTT**: Forward (*NTT*) and Inverse NTT (*INTT*) transform
2. **MUL**: Coefficient-wise multiplication by ψ^i (*PSIS_MUL*), ψ^{-1} (*IPSIS_MUL*) and two polynomial (*COEF_MUL*).

Modular multiplication can be performed using two primary approaches: Montgomery multiplication (REDC) and the method introduced by Longa et al. in [4] (KRED).

3 Previous Work

The theory of NTT is summarized in [3]. Previous hardware implementations of NTT were reported in [5, 6]. The first hardware implementations of NTT targeted the Encryption/KEM PQC schemes such as Ring-LWE [7,8]. The most recent efforts aimed specifically at the efficient implementations of the Round 1 PQC candidate NewHope, qualified to the second round of the NIST PQC standardization process [9, 10].

The comprehensive survey of commercial and academic HLS tools was provided in [11]. Some of the reported benchmarks included AES, Blowfish, and SHA-1, but the reported results were significantly worse than those achievable using the traditional RTL-based design methodology. Efficient implementations

of block ciphers, hash functions, and authenticated ciphers using High-Level Synthesis were reported in [12, 13]. In the majority of cases, these implementations closely matched the performance of RTL implementations in terms of major metrics, such as throughput and throughput-to-area ratio. The first attempts at the use of HLS for benchmarking of PQC candidates were reported in [2].

4 Methodology

4.1 High Level Synthesis

High-Level Synthesis (HLS) enables modeling behavior of an algorithm in high-level language such as C/C++. Then, the tool creates the detailed RTL micro-architecture. In this paper, before writing HLS code, the block diagrams for the corresponding digital circuits have been developed in detail, using the same approach as that used in RTL design. The HLS-ready C code has been then written with the help of optimization directives, provided using *pragmas*, such as *pipeline, array_reshape, inline, dependence, resource*. The goal of these directives is to guide the synthesis tool.

Other approach, relying on reusing the existing software C code led to major inefficiencies. These inefficiencies could not be overcome by using HLS directives alone. Thus, the HLS code in this paper was developed from scratch, and its functionality verified using the reference code. The HLS-based development effort was still several times shorter than the RTL design, mostly due to the more efficient verification phase and the higher-level of abstraction.

4.2 Software/Hardware Codesign with SDSoC

The SDSoC environment provides a framework for developing and delivering hardware-accelerated embedded processor applications using standard programming languages such as C/C++. The *sds++* system compiler analyzes a program to determine the dataflow between software and hardware functions. Then, it generates an application-specific SoC supporting bare metal, Linux, or FreeRTOS. Additionally, the system compiler generates hardware IP, and the software control code automatically implements data transfers and synchronizes hardware accelerators and application software, therefore pipelining communication and computation.

As shown in Fig 1, initially, the reference code is profiled to find the most time-consuming operations. These operations are candidates to be implemented as hardware accelerators, which guarantees substantial overall speed up. The accelerator is implemented in Vivado HLx. The C-code is refactored or restructured to be synthesizeable in Vivado HLx. The logical verification at C level in Vivado HLx saves a lot of time compared to RTL design. In particular, debugging is possible at the C level, and a developer can track variables the same way as in traditional debugging. The tool directives help resolves dependency hazards, such as Write-After-Write and Write-After-Read. The simulation step

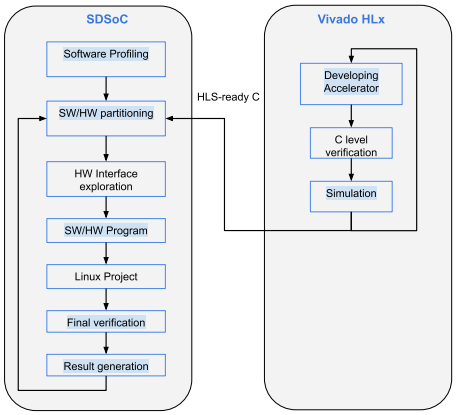


Fig. 1. The HW Accelerator Development using Vivado HLx and SDSoC; the highlighted text marks similarity between RTL/SDK and HLS/SDSoC

is for checking whether the generated HDL code has hardware operations run as expected (in parallel or sequential).

The HLS-ready C code is integrated with the reference code, replacing the reference C code most time-consuming functions by HLS-ready C code. The HLS-read code is designated as hardware functions, while the rest of the reference code is designated as software. When compiling to the SW/HW program, the SW code will invoke the HW accelerator via DMA transfer. The transfer is handled by *pragma data_mover*. The SDSoC compiler chooses the type of the data mover automatically by analyzing the code and physical memory layout. The developer can manually override the compiler default settings as follow:

- AXIFIFO: non-contiguous memory, less than 300 bytes
- AXIDMA_SIMPLE: contiguous memory less than 32MB
- AXIDMA_SG: either contiguous or non-contiguous memory, greater than 300 bytes
- FASTDMA: contiguous memory only.

Depending on the input to a hardware accelerator, the input can be sequential or random-access, fast or slow. Therefore, the protocol can be adjusted to AXIFIFO or AXIDMA_SG, AXIDMA_SIMPLE, or FASTDMA, accordingly, by changing the *data_mover* option, and then rebuilding the project.

After finding the suitable interface between software and hardware, the SW/HW program is compiled to a single executable binary, which runs on the target operating system, in our case Linux. The HDL code generated by HLS is translated to a bitstream, which is included in the Linux system image. After rebooting the operating system with the new Linux system image, the SW/HW program is able to run like traditional software. The transfer back and forth between SW and HW is hidden from developers.

At this point, the final functional verification, achieved by comparing intermediate results and final outputs generated by the SW-only program and the SW/HW program guarantees the correctness of the hardware accelerator and the DMA transfer. If both the SW-only program and the SW/HW program realize the same functionality, then the correct speed-up can be measured.

4.3 High Level Synthesis: Block Diagram versus Space Exploration

There are two approaches to implement hardware accelerators in HLS:

- SE/HLS: Implement HLS code to run as fast as possible by experimenting with design exploration; the final hardware architecture is unknown until the best result is achieved;
- BD/HLS: Develop block diagram, implement HLS code following this block diagram.

Both SE/HLS and BD/HLS approaches inherit the advantages of HLS: quicker verification and quicker development than in traditional RTL.

In the SE/HLS approach, a small portion of the total development time is spent on writing HLS code and verifying its functionality. The rest of the time is devoted to design space exploration using *pragma* directives. There are 24 *pragma* directives in HLS; different combinations will lead to different architectures. The impact of a particular *pragma* directive is heavily dependent on the code structure and the algorithm. Some directives may have no impact at all. Directive such as *unroll, pipeline, dataflow* may replicate same hardware resources multiple times, which increases the resources utilization significantly. Furthermore, the HLS synthesis process may take a lot of time depending on the number of hardware architectures it creates, e.g., partial unrolling vs. full unrolling of a big loop. The HLS resource usage reports are usually inaccurate, which makes comparison between HLS reports unreliable. The exact result can only be obtained after post-place and route in Vivado. Eventually, the HLS design by space exploration may create sub-optimal hardware architecture.

In BD/HLS approach, the large portion of the total development time is spent on developing a block diagram and implementing it in HLS-ready C. The rest of the time is spent on verification. Since the exact hardware architecture is known beforehand, space exploration is not required.

Structure of HLS code is the foundation for SE/HLS and BD/HLS code. With well-structured code, BD/HLS can outperform SE/HLS. For example, the

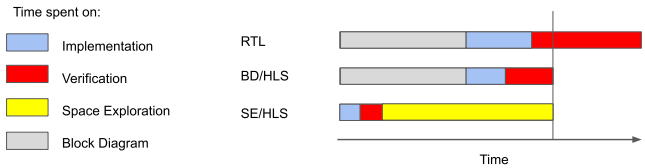


Fig. 2. BD/HLS versus SE/HLS development timeline

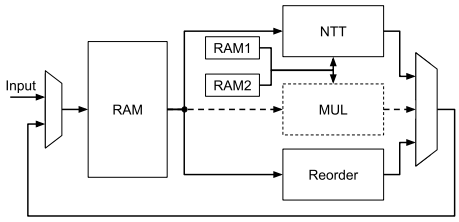


Fig. 3. Single NTT module top level design

reference C code will not perform as well as HLS-ready C code. Still, the HLS-ready C code will not run as well as having the code implemented from an optimized and verified block diagram.

In the end, both approaches generate hardware that has the same functionality, but performance can vary significantly.

5 Hardware Design

5.1 NTT Top Level Design

The top-level block diagram is shown in Fig. 3. There are 3 main components: *NTT*, *MUL* and *Reorder*. For NewHope and Kyber R1, the *NTT* component is responsible for the **NTT** and **MUL** modes. Due to the changes between Kyber R1 and Kyber R2, in Kyber R2, the *NTT* component is only responsible for **MUL** mode. Thus, the *NTT* component of Kyber R2 is simplified. The task of the Reorder component is to convert the coefficient indices to natural order after the **NTT** mode, as explained in Section 5.5. These components can be easily pipelined, as there is only one component running at a time.

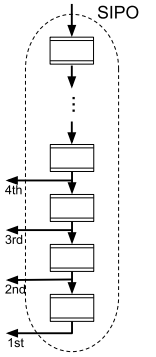
5.2 Memory write back scheme

SIPO (Serial In Parallel Out) is a simple Linear Shift Register (LSR), as shown in Fig 4, used to convert serial input to parallel output. In our design, we use SIPOs to store intermediate results generated by the circuit shown in Fig. 5. These results are written back to the RAM before being overwritten by new incoming results. A different number of registers is placed inside of each SIPO to make sure that only one SIPO can be full at a time. The benefit of the SIPO design is its role in preparing coefficients for the execution of the next NTT stages on the fly.

Algorithm 2 SIPO memory write back scheme

1: **if** $s = 0$ **then**
2: $RAM \leftarrow A_{1st}||C_{2nd}||B_{3rd}||D_{4th}$
3: **else**
4: **if** $SIPO_A$ is full **then**
5: $RAM \leftarrow A_{1st}||A_{2nd}||A_{3rd}||A_{4th}$;
6: **else if** $SIPO_B$ is full **then**
7: $RAM \leftarrow B_{1st}||B_{2nd}||B_{3rd}||B_{4th}$;
8: **else if** $SIPO_C$ is full **then**
9: $RAM \leftarrow C_{1st}||C_{2nd}||C_{3rd}||C_{4th}$;
10: **else if** $SIPO_D$ is full **then**
11: $RAM \leftarrow D_{1st}||D_{2nd}||D_{3rd}||D_{4th}$;

Fig. 4. SIPO Unit



5.3 Number Theoretic Transform

A block diagram of our NTT implementation is shown in Fig. 5. This diagram is based on the design from [6]. One of the improvements is support for both odd and even number of NTT layers.

When $log_2(n)$ is odd, the signal X is asserted at the *last* iteration to let coefficients A', B', C', D' come directly to the SIPO unit instead of going through the 2nd NTT layer. On the other hand, when $log_2(n)$ is even, the multiplexers with signal X can be stripped out.

Our NTT hardware architecture has 2x2 butterfly structure, which can process two layers of NTT with two butterfly units per layer. To implement this architecture in High Level Language, we actively avoid stalling, four coefficients are loaded in each clock cycle, without stalling, and placed into registers A, B, C, D. If KRED is selected as a modular reduction method, the square boxes m_1 and m_2 are KRED and KRED2x, respectively. If REDC Montgomery reduction is chosen, m_1 can be removed or substituted by LSR, and m_2 represe REDC.

When $S = 0$, the circuit operates in the **MUL** mode used to perform operations *PSIS_MUL*, *COEF_MUL*, and *IPSIS_MUL*. The coefficients in the lines B and D are multiplied by coefficients from RAM1, which are $(\psi^{4i+1}, \psi^{4i+3})$ or (r_{4i+1}, r_{4i+3}) or $(\psi^{-(4i+1)}, \psi^{-(4i+3)})$, depending on the performed operation. The obtained result are reduced by the function m_2 and stored in B_{save} and D_{save}, which go to SIPO_B and SIPO_D later on. After that, coefficients from lines A and C are switched to lines B and D, allowing them to be multiplied with coefficients from RAM2, reduced by m_2, and directed to SIPO_A and SIPO_C, respectively. When the following outputs of SIPOs: $A1st, B2nd, C3rd$ and $D4th$ available, they are concatenated and written back to RAM at the index where A1st was loaded from. After the computations are finished, the SIPOs iterate for a few cycles until they are emptied of coefficients.

When $S = 1$, the circuit operates in the **NTT** mode use to perform operations *INTT* and *NTT*. Four coefficients will go through the 2x2 butterfly structure,

and results are written to SIPOs, coefficients in lines B and D are multiplied with ω_n^i or ω_n^{-i} depend on whether the circuit compute NTT or $INTT$.

When $SIPO_A$ is full, four coefficients available at the outputs $A1st$, $A2nd$, $A3rd$, $A4th$ are concatenated, and stored back to the RAM at the index when $A1st$ is loaded. After one clock cycle, the same happens with results accumulated in $SIPO_C$, and then $SIPO_B$ and $SIPO_D$. After the NTT mode is complete, the circuit spends a few clock cycles for $SIPO_C, SIPO_B, SIPO_D$ until they are emptied of coefficients.

The red lines in Fig. 5 represent four likely critical paths in this design. Which of these paths becomes the critical path depends on routing delays, which are impossible to predict without running the actual synthesis, placing, and routing.

For the NewHope and Kyber R1, all five operations from Section 2.1 are supported by the circuit from Fig. 5. In the case of Kyber R2, *PSIS_MUL* and *IPSIS_MUL* do not apply. Additionally, in case of Kyber R2, only NTT and $INTT$ are performed in hardware. The *COEF_MUL* is performed by separate hardware explained in detail in Section 5.4. Therefore, the NTT mode of Kyber R2 can be simplified by stripping the dot line and removing multiplexers to save resources and improve maximum clock frequency.

The precomputed values of all constants are stored in the dual-port memories RAM1 and RAM2, of the size $2.5n$ and $3n$ memory locations, respectively. The number of bits stored at each memory location is equal to $log_2 q$. The memory map and formulas for the values of constants stored within each specific address range are shown in Fig. 6.

5.4 Kyber Round 2 Multiplication

In Kyber R2, basic multiplication is redefined, as shown in Algorithm 3. In the reference software implementation, instead of multiplying by z_{2i+1} each time in order to calculate r_i, bz_{2i+1} is defined as $b_{2i+1} \times z_{2i+1}$. These values are precomputed and stored in RAM1. As shown in the Fig 7, a_i, b_i, bz_i are coefficients coming from RAM, RAM1, and RAM2, respectively. m_2 is implemented as either KRED2x or REDC.

5.5 Shuffle and Reordering

The order of coefficients is changed in the **NTT** mode. Thus, after each NTT operation, one must shuffle and reorder its coefficients to avoid complexity in the

Algorithm 3 Kyber Round 2 Multiplication

Require: Polynomial of degree 2 $\overline{A}, \overline{B}$ in NTT domain
Ensure: $\overline{R} = r_{n-1}X^{n-1} + \cdots + r_1X^1 + r_0 = \overline{A} * \overline{B}$
1: $r_{2i+1}X + r_{2i} = \overline{A} * \overline{B} = (a_{2i+1}X + a_{2i}) * (b_{2i+1}X + b_{2i}) \mod X^2 - z_{2i+1}$
2: $r_{2i+1}X + r_{2i} = a_{2i+1}b_{2i+1}X^2 + (a_{2i+1}b_{2i} + a_{2i}b_{2i+1})X + a_{2i}b_{2i}$
3: $r_{2i+1}X + r_{2i} = (a_{2i+1}b_{2i} + a_{2i}b_{2i+1})X + a_{2i}b_{2i} + a_{2i+1}b_{2i+1}z_{2i+1}$

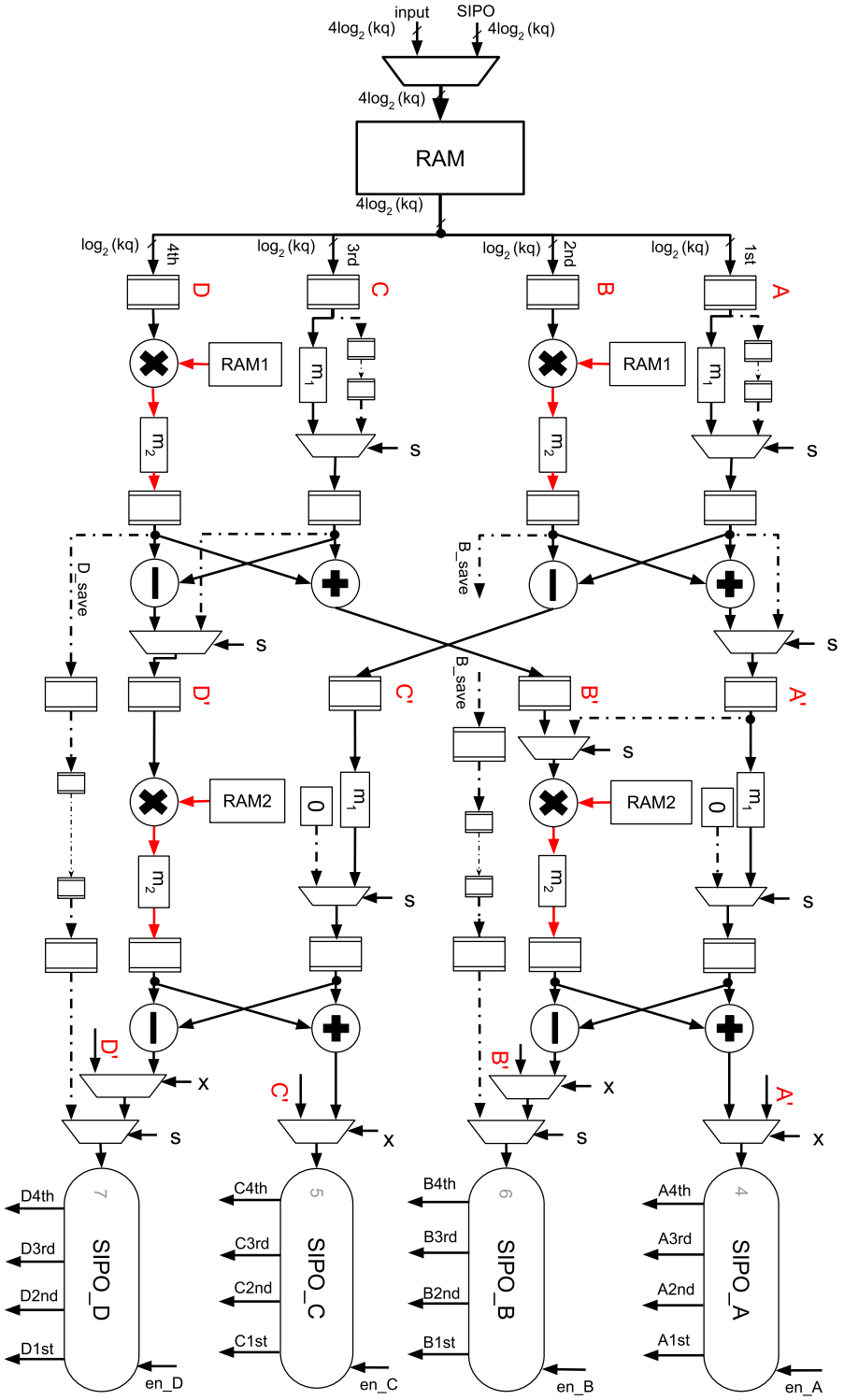


Fig. 5. Block diagram of the proposed hardware architecture to perform fast polynomial multiplication using NTT.

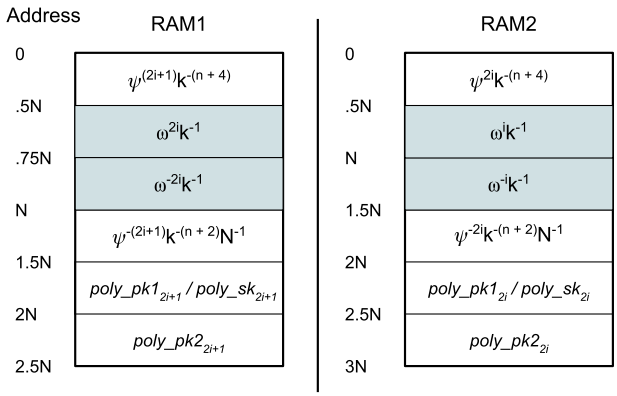


Fig. 6. The memory maps of RAM1 and RAM2, including formulas for values of constants stored in specific memory ranges. $n{=}log_2 n$, $\omega = \omega_n$, $i \in [0, 1, \ldots, n/2)$ for RAM1 and RAM2, except the gray area of RAM 1, where $i \in [0, 1, \ldots, n/4)$. For the KRED, the value of k is given in Table 1. If the REDC is used, k is assumed to be 1. *poly_pk* and *poly_sk* are NTT domain preloaded public and secret polynomials.

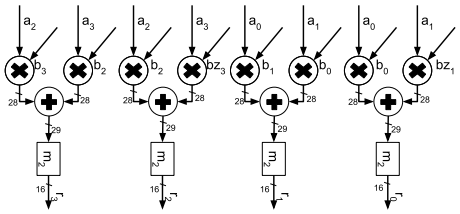


Fig. 7. Kyber Round 2 MUL component

MUL mode. Since the RAM unit stores four coefficients in each line, it forms a 4xn/4 matrix. After NTT or INTT, it becomes the transposed version of this matrix. By knowing exactly the structure of the coefficient indices, we apply the cycles-based Matrix Transpose proposed in [14]. The number of clock cycles for 128-, 256-, and 1024-point NTT is 64, 80, 318 clock cycles, respectively. In particular, for the 1024-point NTT, we use 318 clock cycles vs. 1024 clock cycles in [10].

6 Results

The target device is Zynq UltraScale+ MPSoC ZCU104, with CPU Cortex-A53 running at 1.2 GHz. Results of the HLS implementation of two alternative reduction methods, KRED and REDC, for the value of q corresponding to investigated candidates, are shown in Table 2. These results demonstrate that compared to REDC, the implementation of KRED uses less resources and is comparable in

Table 1. Selected NTT-based Round 2 PQC candidates investigated in this study. Major parameters of NTT: N and q. Parameters k and m of the Longa-Naehrig modular reduction, and $qinv$ used in the Montgomery Reduction.

Candidate	**Cat (#NTT)**	**N**	**q**	2^m	**k**	$\mathbf{k^2}$	**qinv**
NewHope	1,5 (1)	512/1024	12,289	2^{12}	3	2^3+1	$2^{13}+2^{12}-1$
Kyber R1	1,3,5 (2,3,4)	256	7,681	2^9	15	2^8-2^5+1	$2^{13}-2^9-1$
Kyber R2	1,3,5 (2,3,4)	256	3,329	2^8	13	$2^7+2^5+2^3+1$	2^9+2^8+1

Table 2. High Level Synthesis of KRED and REDC implementation, results obtained after Post-Place & Route

Candidate	**Modular Reduction**	**DSP**	**LUT**	**FF**	**Slice**	**Max. Freq**
NewHope	KRED	1	118	100	28	530
KyberR1		1	125	93	32	507
KyberR2		1	150	112	35	502
NewHope	REDC	1	370	357	85	515
KyberR1		1	387	333	69	512
KyberR2		1	391	382	91	476

term of performance. Therefore, in this paper, KRED is selected as a modular reduction method.

In Table 3, the comparison of the NTT implementations according to two approaches, BD/HLS and SE/HLS, is summarized. The BD/HLS approach uses **2x, 5x, 22x, 35x,** and **1.1x** less BRAMs, DSPs, LUTs, FFs and Clock Cycles, respectively. In [15], the authors experiment with multiple combinations of directives, applied to multiple loops. However, the design outcome is still not as efficient as in our BD/HLS design.

The maximum clock frequencies and the corresponding resource utilization, obtained after the synthesis and implementation, with the help of the tool for optimization of tool options, Minerva [16], are listed in Table 4. The number of BRAMs in HLS is higher than in RTL due to a higher abstraction level description of HLS. In particular, the tool duplicates RAM1 and RAM2 for each MUL component. Thus, the number of BRAMs for Kyber R2 is higher than in RTL. There are two pairs of RAM1 and RAM2 in a single HLS NTT module, instead of just one. With the exception of the number of FFs in Kyber R2, the number of LUTs, FFs, and Slices is consistently greater in HLS.

Table 3. Results for BD/HLS vs. results for SE/HLS for 1024-point NTT

Work	**BRAM 18K**	**DSP**	**FF**	**LUT**	**Cycles**	**Cycles Reduc.**
[15]	11.5	10	16,402	21,167	7,597	1.59
[15]	21.5	19	30,498	38,984	5,291	1.10
This work	**10**	**4**	**1,342**	**1,110**	**4,776**	**1.0**

Table 4. Resources Utilization HLS and RTL

Algorithm	#NTT	DSP	BRAM 36K	LUT	FF	Slice	Freq. (Mhz)
HLS							
NewHope 1	1	4	3	1,181	1,403	239	454
NewHope 5	1	4	5	1,110	1,342	219	455
Kyber R1-1	2	8	2	2,788	2,704	594	455
Kyber R1-3	3	12	3	4,205	4,058	879	455
Kyber R1-5	4	16	4	5,562	5,573	1,225	455
Kyber R2-1	2	24	7	2,325	2,346	430	455
Kyber R2-3	3	36	11	5,379	4,043	1,074	416
Kyber R2-5	4	48	14	7,111	5,457	1,374	416
RTL							
NewHope 1	1	4	3	1,040	940	190	476
NewHope 5	1	4	5	842	803	170	476
Kyber R1-1	2	8	2	2,185	2,625	411	500
Kyber R1-3	3	12	3	3,318	3,937	605	500
Kyber R1-5	4	16	4	4,363	5,237	795	500
Kyber R2-1	2	24	5	2,040	3,223	433	500
Kyber R2-3	3	36	7.5	3,054	5,098	637	500
Kyber R2-5	4	48	10	4,055	6,803	960	500

Table 5. Comparison of the transfer ratio, overhead between SDSoC and Bare Metal

Algorithm	Total Transfer Size (bytes)		Times	Total Transfer Time (μs)		Transfer Ratio	Transfer Overhead	
	In	Out		BM	SDSoC	SDSoC/ BM	BM	SDSoC
ENCAPSULATION								
NewHope 1	2,048	2,048	1	7.91	12.64	1.60	4.51%	7.01%
NewHope 5	4,096	4,096		11.90	19.50	1.64	3.67%	5.87%
Kyber R1-1	1,024	1,536		7.85	9.86	1.26	4.94%	6.12%
Kyber R1-3	1,536	2,048		8.05	11.71	1.46	3.58%	5.12%
Kyber R1-5	2,048	2,560		9.42	13.49	1.43	2.86%	4.04%
Kyber R2-1	1,024	1,536		7.85	9.86	1.26	7.77%	9.54%
Kyber R2-3	1,536	2,048		8.05	11.71	1.46	3.99%	5.69%
Kyber R2-5	2,048	2,560		9.42	13.49	1.43	3.12%	4.40%
DECAPSULATION								
NewHope 1	3,072	3,072	2	15.22	21.57	1.42	8.56%	11.69%
NewHope 5	6,144	6,144		19.81	32.13	1.62	5.93%	9.26%
Kyber R1-1	2,048	2,048		15.15	17.99	1.19	9.35%	10.89%
Kyber R1-3	3,072	2,560		15.90	20.76	1.31	7.02%	8.96%
Kyber R1-5	4,096	3,072		17.91	23.47	1.31	5.39%	6.94%
Kyber R2-1	2,048	2,048		15.15	17.99	1.19	11.36%	13.17%
Kyber R2-3	3,072	2,560		15.90	20.76	1.31	7.53%	9.58%
Kyber R2-5	4,096	3,072		17.91	23.47	1.31	5.78%	7.42%

Table 6. Speed up of Software/Hardware Codesign vs. Pure Software

Algorithm	Total SW (μs)	Total SW NTT (μs)	%SW NTT	Total SW/HW (μs)		Total Speed-up @Max Freq	
				BM	SDSoC	BM	SDSoC
ENCAPSULATION							
NewHope 1	360.3	199.8	55%	175.2	180.3	2.06	2.00
NewHope 5	737.0	438.1	59%	324.0	332.2	2.27	2.22
Kyber R1-1	389.2	240.9	62%	158.9	161.1	2.45	2.42
Kyber R1-3	582.3	368.3	63%	224.8	228.7	2.59	2.55
Kyber R1-5	826.9	509.4	62%	329.6	334.0	2.51	2.48
Kyber R2-1	328.5	237.8	72%	101.1	103.4	3.25	3.18
Kyber R2-3	533.9	343.0	64%	201.5	205.7	2.65	2.60
Kyber R2-5	785.2	495.4	63%	301.8	306.4	2.60	2.56
DECAPSULATION							
NewHope 1	427.5	273.5	64%	177.8	184.6	2.40	2.32
NewHope 5	895.7	598.0	67%	334.0	347.1	2.68	2.58
Kyber R1-1	483.2	340.8	71%	161.9	165.2	2.98	2.92
Kyber R1-3	710.4	504.2	71%	226.5	231.8	3.14	3.06
Kyber R1-5	992.1	682.4	69%	332.0	338.0	2.99	2.94
Kyber R2-1	429.5	315.5	73%	133.3	136.6	3.22	3.14
Kyber R2-3	667.8	476.8	71%	211.1	216.8	3.16	3.08
Kyber R2-5	950.8	662.9	70%	310.0	316.4	3.07	3.00

Traditional RTL SW/HW Codesign often uses Bare Metal (BM) to handle transfer between CPU and FPGA. The DMA in BM is often implemented manually. Contrary to that, SDSoC creates an abstraction layer of the interface handler. As a result, switching from software to hardware is very easy. To demonstrate the overhead of abstraction in using SDSoC, the best selected transfer interface in SDSoC is compared with Bare Metal in Table 5. Additionally, the **Transfer Overhead** column is the percentage of **Total Transfer Time** over the **Total SW/HW** in Table 6.

In Table 6, timing results are summarized. The **Total SW** is the software only execution time, the **Total SW NTT** column is the time spent on NTT operations in SW, **%SW NTT** is the percentage of the total execution time in software devoted to NTT, the **Total SW/HW** is the total time after offloading the critical function (NTT) to hardware. The **Total Speed-up @Max Freq** is the ratio between **Total SW** and **Total SW/HW**. This speed-up is roughly equal between the SDSoC and Bare Metal approaches.

7 Conclusions

Using HLS and SDSoC are two promising approaches to benchmarking SW/HW implementations of PQC. With the help of these approaches, the development time is substantially reduced, with the relatively small penalty in terms of the

total execution time, HW-SW transfer time, and the total speed-up vs. purely SW implementation. Overhead in terms of resource utilization is more substantial, especially in terms of the number of LUTs, FFs, and Slices. The BD/HLS approach, based on the use of block diagrams, was shown to be substantially more efficient than the approach, SE/HLS, based on applying various pragmas to existing code and letting the tool to infer the best possible architecture.

References

1. "Post-Quantum Cryptography Standardization," https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization, 2019.
2. F. Farahmand, V. B. Dang, D. T. Nguyen, and K. Gaj, "Evaluating the potential for hardware acceleration of four ntru-based key encapsulation mechanisms using software/hardware codesign," in *PQCrypto*, 2019, pp. 23–43.
3. E. C.-h. Chu and A. George, *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*, ser. Computational Mathematics Series.
4. P. Longa, M. Naehrig, P. Longa, and M. Naehrig, "Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography," in *Cryptology and Network Security - CANS 2016*, vol. 10052.
5. T. Pöppelmann and T. Güneysu, "Towards Efficient Arithmetic for Lattice-Based Cryptography on Reconfigurable Hardware," in *Progress in Cryptology – LATINCRYPT 2012*, Berlin, Heidelberg.
6. C. Du, G. Bai, and X. Wu, "High-Speed Polynomial Multiplier Architecture for Ring-LWE Based Public Key Cryptosystems," in *GLSVLSI'16*.
7. D. D. Chen *et al.*, "High-Speed Polynomial Multiplication Architecture for Ring-LWE and SHE Cryptosystems," *IEEE Trans. on Circuits and Systems*, vol. 62, no. 1, 2015.
8. C. P. Renteria-Mejia and J. Velasco-Medina, "High-Throughput Ring-LWE Cryptoprocessors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*.
9. T. Oder and T. Guneysu, "Implementing the NewHope-Simple Key Exchange on Low-Cost FPGAs," in *LATINCRYPT 2017*, Havana, Cuba, Sep. 2017.
10. P.-C. Kuo *et al.*, "High Performance Post-Quantum Key Exchange on FPGAs," Cryptology ePrint Archive 2017/690, Feb. 2018.
11. R. Nane *et al.*, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 2016.
12. E. Homsirikamol and K. Gaj, "Hardware Benchmarking of Cryptographic Algorithms Using High-Level Synthesis Tools: The SHA-3 Contest Case Study," in *Applied Reconfigurable Computing - ARC 2015*.
13. ——, "Toward a new HLS-based methodology for FPGA benchmarking of candidates in cryptographic competitions: The CAESAR contest case study," in *2017 International Conference on Field Programmable Technology, FPT 2017*. Melbourne, Australia: IEEE, Dec. 2017, pp. 120–127.
14. D. E. Knuth, *The Art of Computer Programming, Fundamental Algorithms*, 1997.
15. K. Kawamura, M. Yanagisawa, and N. Togawa, "A loop structure optimization targeting high-level synthesis of fast number theoretic transform," in *Int. Symposium on Quality Electronic Design, ISQED 2018*, vol. 2018-March, 2018, pp. 106–111.
16. F. Farahmand, A. Ferozpuri, W. Diehl, and K. Gaj, "Minerva: Automated hardware optimization tool," in *2017 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017*. Cancun: IEEE, Dec. 2017, pp. 1–8.