SPQCop: Side-channel protected Post-Quantum Cryptoprocessor

Arpan Jati[1], Naina Gupta[2,3], Somitra Kumar Sanadhya[4] and Anupam Chattopadhyay[2]

[1] IIIT, Delhi, arpanj@iiitd.ac.in
[2] NTU, Singapore, naina003@e.ntu.edu.sg, anupam@ntu.edu.sg
[3] Fraunhofer, Singapore, naina.gupta@fraunhofer.sg
[4] IIT, Ropar, somitra@iitrpr.ac.in

Abstract. The past few decades have seen significant progress in practically realizable quantum technologies. It is well known since the work of Peter Shor that large scale quantum computers will threaten the security of most of the currently used public key cryptographic algorithms. This has spurred the cryptography community to design algorithms which will remain safe even with the emergence of large scale quantum computing systems. An effort in this direction is the currently ongoing post-quantum cryptography (PQC) competition, which has led to the design and analysis of many concrete cryptographic constructions. Among these, Lattice based algorithms have emerged to be promising candidates. Therefore, we focus on the efficient implementation of Ring-LWE based quantum-safe key-exchange algorithms. Further, deployment of hardware implementing such algorithms in critical applications requires security against implementation attacks.
In this work, we design a side channel resistant post-quantum cryptoprocessor which supports `NewHope-NIST`, `NewHope-USENIX` and `HILA5` key-exchange schemes. The implemented cryptoprocessor is highly optimized with minimal overhead due to the countermeasures. It requires about 13,500 LUTs and 8,100 FFs. Due to a significantly pipelined architecture, an operating speed of 406 MHz could be achieved on the latest 16nm FPGAs; resulting in a key-exchange time of only 158μS, 157μS and 148μS for the above mentioned designs respectively. We also present detailed area and performance metrics for different modules required for all the designs. To the best of our knowledge, this work presents the first side-channel leakage resistant post quantum accelerator. Furthermore, this is also the fastest hardware implementation of `NewHope-NIST`.

Keywords: cryptography · post-quantum · key-exchange · cryptoprocessor · NewHope · NewHope-Simple · HILA5 · fault-resistance · SCA

1 Introduction

The idea of building computers based on quantum mechanics for solving computational problems originated in the works of Deutsch and Feynman [Deu98, Fey98]. However, significant progress in their physical realization took place only in the last decade because of breakthroughs in stability of entanglement and the development of new materials [BWM+16]. It has been estimated that in the coming years, practical quantum computers will exist which will be able to break most of the currently used cryptographic systems [Jus17].

In light of the above, tremendous work has happened in the field of post-quantum cryptography. There are many classes of hard problems which can protect cryptographic

constructions against quantum computers. These hard problems span a diverse range of fields such as hash functions [BDH11, BHH+15], coding theory [Mce78, Nie86], lattices [Pei14, GLP12], etc. Among these, lattice-based cryptography has emerged to be a promising candidate. This is due to the strong theoretical security guarantees obtained from constructions based on the hardness of some lattice problems. The significant theoretical work in [NS01, Reg04] has led to the development of several lattice based cryptographic constructions over the years.

In order to be prepared for the post-quantum era, National Institute of Standards and Technology (NIST) published a call for proposals in December 2016 [NIS]. The purpose of the proposal was to standardize some quantum-safe key-exchange, public-key encryption and signature schemes. As a response, 69 designs were submitted for public review out of which 26 designs are shortlisted for second round. In this work, we focus on implementing some key-exchange schemes based on the ring-learning-with-errors (RLWE) problem [LPR10]. The design `NewHope-USENIX` [ADPS15], its variants `NewHope-Simple` [ADPS16] and `NewHope-NIST` introduced by Alkim et al., and the design `HILA5` [Saa17] designed by Saarinen are the focus of this work. `HILA5` was originally submitted as an individual candidate in first round but was later merged with `Round2` [BBGM+17] (now known as `Round5` [BGML+18]). Further, a variant of `NewHope-Simple` referred as `NewHope-NIST` in this paper and `Round5` are two of the shortlisted candidates for second round to the NIST PQC competition.

1.1 Motivation for a flexible design

Designing a unified hardware architecture supporting multiple cryptographic algorithms has been a well established practice. Many such interesting works have been reported in literature [SKR+13, BHO04, RVM+13]. Designing a flexible cryptoprocessor has multiple advantages for both FPGA as well as ASIC implementations. For instance, such a design supporting multiple implementations has the major benefit of significantly reduced area usage compared to separate implementations. There is also an added benefit in the scenario when, either due to a recently discovered security vulnerability or due to efficiency considerations, there a need to switch the algorithm being used. The unified hardware allows this switch easily. Furthermore, different applications can have different requirements. Using a cryptoprocessor can provide multiple choices to choose from depending on the specific needs of the applications. It is cheaper as well as easily maintainable from the user perspective.

Applications using an FPGA with an integrated hard ARM core processor or a soft Microblaze core instance are interesting from the perspective of a cryptoprocessor since the combination can be used in a hybrid fashion. Some part of the operation can be performed in software and some part, which is computationally expensive (for instance the NTT operation used for the polynomial multiplication), can be executed in the specially designed hardware.

1.2 Motivation for side-channel resistance

Side-channel attacks are known in literature for quite some time. Implementation attacks utilize vulnerabilities in the implementation instead of the mathematical structure. There are several classes of such attacks, for example: faults injected into a system can cause incorrect results [BS97, BDL97], which may be used maliciously to recover secret keys or other associated information. A device may also leak sensitive information passively by the means of power consumption or electromagnetic emanations; which can be captured and used for attacks [KJJ+98, KJJ99, BCO04]. So, for a practical system protection against such attacks is mandatory.

1.3 Contributions

In this work, we present the design for an efficient and flexible Ring-LWE based cryptoprocessor designed with fault and side channel leakage resistance. In particular, we make the following contributions:

1. The cryptoprocessor is designed to support multiple key-exchange algorithms and can switch between algorithms on the fly. As already mentioned, many such attempts have been made for classical cryptographic algorithms. However, to the best of our knowledge, this is the first attempt towards such a combined architecture for post-quantum algorithms.
2. We implement the design and provide experimental results for two modern FPGAs. We also present the first hardware implementation for `HILA5` and the fastest implementation for `NewHope-NIST`.
3. The cryptoprocessor can also work both in standalone as well as hybrid (hardware/-software) environments providing acceleration for a wide range of use cases. As discussed in Section 5.4, the implementation results on a MiniZed board demonstrate significant performance gains over a purely software based implementation.
4. We have added many fault countermeasures to the overall design. The individual countermeasures are designed to have high performance and minimal overheads while providing good fault resistance. The utilized techniques are designed to compliment each other and further enhance the security. We also present in Section 3.7.1, new fast hashing based checksum construction with good differential characteristics; which can be used to detect maliciously injected faults with high probability.
5. Fault countermeasures using inverted/complimentary logic, and state counters were also implemented and designs for secured FSM's for fetch and execute operations are provided in Sections 3.7.3, 3.7.4 and 3.7.5.
6. Many optimization techniques for critical path shortening such as pipelining, register-retiming, logic-reorganization, resource sharing etc. were used for the processor and its individual modules, resulting in significant gains in performance and reduction in area while maintaining a good area-vs-performance trade-off.

In order to support the 512-bit variants of the `NewHope-NIST` proposal, some extra instructions may be added, with negligible increase in resources because of the configurable architecture. This is possible as many of the instruction modules are configurable including NTT.

2 Preliminaries

In this section, we briefly discuss the three post quantum key exchange protocols `NewHope-USENIX`, `NewHope-NIST`, and `HILA5`.

2.1 Notations

Throughout the paper, we use bold symbol to represent polynomial coefficients in time domain (for example $\mathbf{s}$). A hat over the top of a symbol is used to represent coefficients in the frequency domain (e.g. $\hat{\mathbf{s}}$). $\mathscr{R}$ defined as $\mathscr{R} = \mathbb{Z}[X]/(X^n + 1)$ denotes the ring of integer polynomials modulo $(X^n + 1)$, where n is a power of 2. $\mathscr{R}_q$ is the polynomial ring where the coefficients are modulo q. Let $\mathcal{X}$ be a probability distribution over $\mathscr{R}$, then $x \leftarrow_{\$} \mathcal{X}$ means that x has been sampled according to $\mathcal{X}$. Further, ψ_k^n is used to denote an

array of n elements where each element has been chosen independently at random from the centered binomial distribution with parameter k.

2.2 Protocol description

The goal of a key-exchange algorithm is to generate a shared secret key between two distrusting parties. The two parties, conventionally named Alice and Bob, first agree on a global system parameter $\mathbf{a} \in \mathscr{R}_q$. Following this, three algorithms are used for the actual key exchange:

1. *KeyGen(**a**)* : Alice chooses two polynomials ($\mathbf{s}$, $\mathbf{e}$) from the noise distribution ψ_{16}^n and computes $\mathbf{b} = \mathbf{as} + \mathbf{e}$. The public key pair $(\mathbf{a}, \mathbf{b})$ is sent to Bob and the secret key $\mathbf{s}$ is stored. The polynomial $\mathbf{e}$ is a random masking noise used to hide the secret key.

2. *Encaps(**a**, **b**)* : Bob samples ($\mathbf{s}'$, $\mathbf{e}'$) from ψ_{16}^n. Using these polynomials and the public key of Alice, Bob computes $\mathbf{u} = \mathbf{as}' + \mathbf{e}'$ and sends this to Alice. Bob also generates $\mathbf{v} = \mathbf{bs}' = (\mathbf{as} + \mathbf{e})\mathbf{s}' = \mathbf{ass}' + \mathbf{es}'$.

3. *Decaps(**u**)* : Alice can now compute $\mathbf{v}' = \mathbf{us} = (\mathbf{as}' + \mathbf{e}')\mathbf{s} = \mathbf{ass}' + \mathbf{e}'\mathbf{s}$

As the noise polynomials $\mathbf{e}$ and $\mathbf{e}'$ are small, the computed shared secrets of Alice and Bob (resp., $\mathbf{ass}' + \mathbf{e}'\mathbf{s}$, and $\mathbf{ass}' + \mathbf{es}'$) are approximately the same ($\approx \mathbf{ass}'$). Thus, both Alice and Bob have approximately equal shared secret. A problem arises when the value $\mathbf{ass}'$ is near the boundary between where the point rounds to 0 or to 1. Addition of different noise terms by Alice and Bob may result in different values after rounding. In order to solve this problem and extract exactly the same shared secret, error recovery/reconciliation mechanism is used. This mechanism is used to send some “hints" about the shared secret. Both the parties use these hints to extract secret bits, which are same with high probability.

In this section, we first present an overview of the operations required in a key exchange protocol using `NewHope-USENIX` as an example. We do not describe the detailed workings of `NewHope-NIST` and `HILA5`. Instead, later in this section, we explain the differences among these protocols.

In contrast to Peikert's KEM mechanism [Pei14] where '$\mathbf{a}$' is kept fixed, the `NewHope-USENIX` algorithm randomly generates '$\mathbf{a}$' using SHAKE-128 for every key exchange. Further, the LWE secret and error polynomials are now derived from ψ_k^n. In the protocol definition [ADPS15, Protocol 2], the elements in key exchange messages are polynomials in $\mathscr{R}_q$. The description of the protocol is shown in Figure 1.As can be seen in the protocol description, Bob adds more random noise (denoted as $\mathbf{e}''$) to its version of the shared secret in the Encapsulation algorithm. This noise is also sampled from the noise distribution ψ_{16}^n. As mentioned earlier, this extra random noise is required to ensure that both the parties extract same secret bits with high probability. The HelpRec function is used to generate the hints. Finally, Alice and Bob use the reconciliation function Rec with these hints to extract the secret bits from the already shared approximately-equal secret.

2.2.1 Algorithms

We are now ready to briefly describe various algorithms and their parameters used in the implemented protocols.

- **Generation of a**: A randomly generated *seed* is given as input to the function Parse. This function expands the seed to generate the secret consisting of required number of bits. The parse function for `NewHope-USENIX` and `NewHope-NIST` uses SHAKE-128 whereas `HILA5` uses SHAKE-256. Keccak permutation is used inside SHAKE-128 and SHAKE-256 for the expansion.

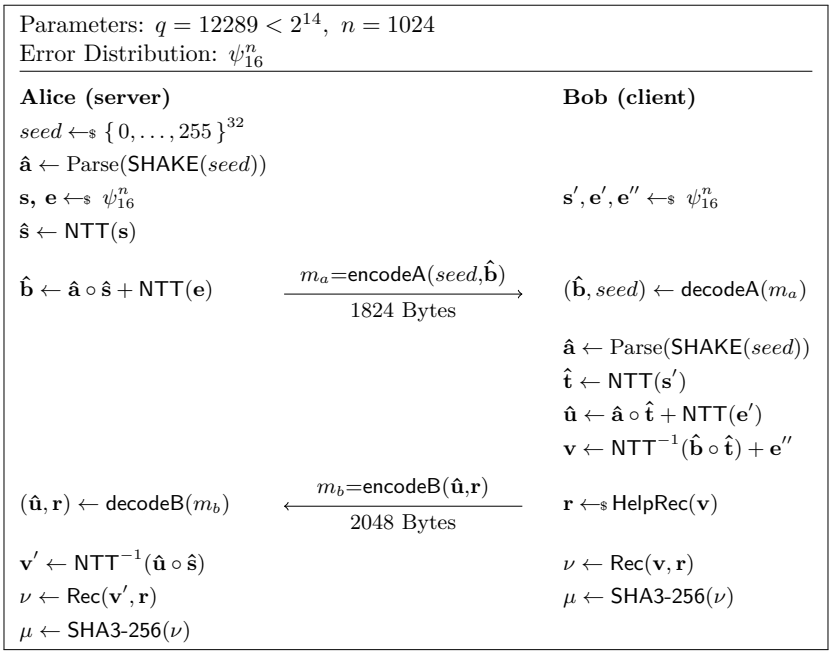


Figure 1: `NewHope-USENIX` Protocol Description

- **Number-Theoretic Transform** : Multiplication of two polynomials of degree n using school-book method (i.e. term by term multiplication) requires $O(n^2)$ operations. Number Theoretic Transform (NTT) introduced by Nussbaumer [Nus80] is an efficient method to perform this multiplication in $O(n \log n)$ time. Polynomial multiplication using NTT is performed as $\mathbf{c} = \mathsf{NTT}^{-1}(\mathsf{NTT}(\mathbf{a}) \circ \mathsf{NTT}(\mathbf{b}))$ where $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathscr{R}$.
  For a polynomial $\mathbf{g} = \sum_{i=0}^{n-1} \mathrm{g}_i X^i \in \mathscr{R}_q$, NTT is defined as follows:

$$\mathsf{NTT}(\mathbf{g}) = \hat{\mathbf{g}} = \sum_{i=0}^{n-1} \hat{\mathrm{g}}_i X^i, \text{ with, } \hat{\mathrm{g}}_i = \sum_{j=0}^{n-1} \gamma^j \mathrm{g}_j \omega^{ij}$$

  where, ω is a primitive nth root of unity and γ is given by $\gamma = \sqrt{\omega} \bmod q$. For `NewHope-USENIX` and `NewHope-NIST`, $\gamma = 7$ whereas for `HILA5` $\gamma = 1945$. The corresponding inverse NTT operation is defined as below with $\gamma^{-1} \bmod q = 8778$ for `NewHope-USENIX` and `NewHope-NIST` and $\gamma^{-1} \bmod q = 4050$ for `HILA5`. This operation also multiplies each coefficient with $n^{-1} \bmod q = 12277$.

$$\mathsf{NTT}^{-1}(\hat{\mathbf{g}}) = \mathbf{g} = \sum_{i=0}^{n-1} \mathrm{g}_i X^i, \text{ where, } \mathrm{g}_i = n^{-1}\gamma^{-i} \sum_{j=0}^{n-1} \hat{\mathrm{g}}_j \omega^{-ij}$$

- **Modular Reductions**: A single NTT operation for `NewHope-USENIX` requires 5120 *butterfly* operation [GS66], each consisting of one addition, one subtraction, one multiplication and two modular reductions. The large number of reductions required for a high degree polynomial results in a performance bottleneck. The Montgomery algorithm [Mon85] and the Short Barrett algorithm [Bar86] are well known techniques for fast-modular multiplication and fast-modular reduction. The *Montgomery reduction* algorithm in Listing 1 is used to reduce the result after multiplication,

whereas the *Barrett reduction* algorithm in Listing 2 is used to reduce the result after additions.

```
ushort mred(uint a) {
    uint u;
    u = (a * 12287);
    u &= ((1 << 18) - 1);
    a += u * 12289;
    return a >> 18;
}
```

Listing 1: Montgomery Reduction (R=2^{18}, q=12289)

```
ushort bred(ushort a) {
    uint u;
    u = ((uint) a * 5) >> 16;
    a -= u * 12289;
    return a;
}
```

Listing 2: Short Barrett Reduction (q = 12289)

It is known that the Montgomery algorithm requires the modulus to be an odd integer. In order to calculate the modulus with an even integer, we used the technique presented by Koc [Koç94]. This operation is needed for the SafeBits operation in `HILA5` where modulus by 12289/4 ≈ 3072 is required. The complete algorithm for modulus with 3072 is presented in Listing 5.

- **Noise Sampling**: A common way for sampling high-precision discrete Gaussian distribution is by using the Knuth-Yao algorithm [Knu76]. This is a considerably complicated algorithm to be implemented in hardware. Sampling at such a high precision may lead to side-channel attacks due to non-constant execution time. To avoid these drawbacks, all the three protocols discussed in this paper use a centered binomial distribution ψ_k^n. The protocols `NewHope-USENIX` and `HILA5` differ from `NewHope-NIST` in the distribution parameter k where the former uses k = 16 whereas later uses k = 8. The technique mentioned in [ADPS15] requires 32 random bits to generate a single coefficient. These random bits can be generated from any suitable uniformly distributed randomness source (in this case ChaCha20 with random seed).

- **Error Recovery Mechanism**: The algorithms presented in this paper differ in the approach used for sending additional information (hints) during key exchange. As mentioned earlier, these hints are required to extract the exact secret key instead of the approximate one. For example, in case of `NewHope-NIST`, a random 256-bits secret ν is chosen and encoded into four coefficients using NHSEncode. The corresponding NHSDecode function is used to extract the original secret bits from the encoded coefficients. The compression (NHSCompress) and decompression (NHSDecompress) functions of `NewHope-NIST` are used to switch the coefficient-wise modulus between modulus 12289 and modulus 8 [ADPS16, Section 2.2]. The `NewHope-USENIX` algorithm uses HelpRec function to compute the reconciliation information and Rec is used to extract the secret key using hints from the reconciliation information. Similarly, functions Safebits and XE5_Cod are used to generate the reconciliation information and the error correcting code in case of `HILA5`. The corresponding functions Select and XE5_Fix are used to compute the secret using the reconciliation information and to correct the error.

2.3 Resistance against Side-channel Attacks

2.3.1 Fault-Attacks

Resistance against fault attacks is a challenging yet important requirement for designing secure cryptographic systems. There are several types of fault injection techniques and countermeasures known in literature [BECN+06]. There are three categories of fault injection: non-invasive, semi-invasive and invasive. Non-invasive faults do not damage the

chip functionality and the package is not opened. The other two types are more powerful and partially or fully damage the device.

Most of the techniques work by actively modifying some memory/logic element or the other. Faults injected to the program counter can result in instruction skip, jumps to invalid locations etc, changes to FSM/state causes control flow modification, etc [MHER14, EFGT16]. Faults in data or instruction memory leads to incorrect results which then can be utilized to mount very powerful attacks against an otherwise secure implementation. However, by utilizing certain countermeasures like error-detection, protection of instruction-pointers and state registers by duplication, checksum or similar techniques, it is possible to make the fault attack significantly more difficult, especially for non-invasive attacks [BECN+06]. In this work, we focus on such countermeasures.

2.3.2 Power and Timing Attacks

Power attacks work by analyzing the power consumption during the execution of cryptographic operations. Simple Power Analysis (SPA) utilizes information from a single power trace during a cryptographic operation to reveal secret information. Whereas, Differential Power Analysis (DPA) utilizes the differences between two simultaneous invocations of a cryptographic algorithm to form a statistical model and then attempts to recover secrets by comparing it with power consumption traces. As the dynamic power consumption of a CMOS circuit is highly correlated to the operation being performed, these are very powerful techniques to extract sensitive data from hardware circuits.

Timing attack comes under another class of attacks which rely on the differences in execution time depending on differences in operands. To protect against timing-attacks, we have ensured that all the modules in this implementation are constant time. The constant-time execution strictly thwarts all attempts at timing attacks. One should note that the *rejection-sampling* as expected is not constant time.

3 Design

In this section, we discuss the main design decisions that led to the development of the overall proposed architecture. Most RLWE algorithms are complex in structure and utilize a large number of constructs, many of which typically have very long critical paths. These long paths lead to poor performance in hardware if directly implemented without consideration. Identifying and optimizing such paths, so that good performance to area trade-offs are achieved is one of the major challenges in any hardware implementation. In this work, the main goal is to have high performance with moderate area utilization. In order to achieve these goals, the following key strategies (among many others) can be used depending on the specific module:

- **Targeting high clock frequency:**
  - Pipelining
  - Balancing logic for critical path shortening
  - Adding registers whenever needed
- **Reducing the number of clock cycles:**
  - Algorithm level modifications
  - Pipelining
  - Efficient memory bandwidth utilization

However, a high performance implementation will utilize a lot of resources (area). Such a design may not be suitable for low-power or resource constrained applications. So, area optimization techniques such as resource sharing, mutual exclusion, usage of FPGA primitives, etc. can be used.

3.1 Design Rationale

3.1.1 Memory

The memory design in any cryptographic system needs careful consideration for good performance and protection against attacks. In this work, we use the following design principles for memory design.

Bus-Width Most processors typically use 8 or 32-bit as the data bus-width. The algorithms discussed in this paper require polynomial operations with coefficients which fit within 15-bits [ADPS15, Section 7.2]. Using 32-bit as bus-width would lead to inefficient usage of memory resources as most of the bits will remain unused. Also, it would require more resources like registers and arithmetic/DSP blocks. Using smaller 8-bit bus is also not preferable as it would require multiple clock cycles to fetch single operands, and would reduce the performance. So, using 16-bit data bus is a good option for our design. In order to allow for usage as a co-processor with extended memory range access, the internal address bus supports 16-bit addresses. As a result the data and instruction memory can be easily extended up-to 128 KiB.

Use of Dual-Port Memories Single port memories are most commonly used for memory access. These allow only a single read or write operation at a time. Unlike single port memories, dual port memories have two sets of address and data lines along with control ports, allowing two simultaneous independent read/write operations. Even though single port memory is most commonly used, such a memory is not preferable for Post-Quantum algorithms as they are typically complex and need a large number of read/write operations. For example, the operation like NTT implemented using single port memory will have drastically reduced performance compared to its dual port counterpart. It is also possible to have quad port memories, but they are not typically supported by the common FPGA platforms/tools and also suffer from issues like large area and reduced performance. Hence, we chose to use dual port memories wherever possible.

Caching / No-Caching Post-quantum algorithms requires large polynomials to be stored in registers. The load and store operations on these registers takes several hundred clock cycles. Having a cache memory to load such large registers is not beneficial and would significantly reduce performance. This is due to the significant latency in transferring data to and from the cache. As a result, we are not using cache memories in our design.

Instruction and Data Memory Among the two popular memory architectures Harvard and von Neumann, the Harvard architecture typically leads to better performance. This is because it has separate signal path for instructions and data. Such an architecture allows for simultaneous access to the instruction and data memory, wherein, the instruction memory can be changed, while a long-running instruction is executing and utilizing the data memory. This allows for easier use as a cryptographic accelerator. This is why, in our design instruction and data memories are kept separate.

Regarding memory sizes there is no defined upper or lower bound. The sizes are purely application dependent. In our design, the instruction memory is a 1 KiB (16-bit $\times$ 512) simple dual port RAM. This is enough for implementing all the supported protocols. The

data memory is a 16 KiB dual-port memory with (16-bit × 8192) and (32-bit × 4096) ports.

3.1.2 Register access across multiple instructions

In accumulator based instruction sets (like *Intel 8085*) the results of instructions are written to a few specific registers, while temporary registers are used to hold local variables. Applications in such processors require a lot of copying between the accumulator and temporary registers. This is acceptable as it only takes one clock cycle to copy the result from the accumulator to temporary registers. However, the polynomial registers used in post-quantum algorithms are quite large and there is a latency of several hundred clock cycles to copy data between registers. In such a design, limiting register access to separate instructions would hamper performance.

Further, to provide full flexibility in writing assembly instructions, the proposed instruction-set is designed in a way that all the registers (of the specified type) can be accessed by all instruction ports (input and output). For this purpose, a switch matrix is used to connect the large number of interface wires dynamically depending on the instruction. Also, separate enable signals are assigned to each instruction so that registers can be accessed by only one instruction at a time.

3.2 Architecture

Considering the above mentioned design considerations, the high level architecture of the proposed cryptoprocessor is shown in Figure 2. The Fetch, Decode and Execute units are the major components of the processor. Protecting these units against side channel attacks is challenging as any such measure typically leads to increase in area and reduction in performance. Later in this section, we discuss how a combination of multiple techniques leads to a significantly secure design with minimal overheads and high performance. A

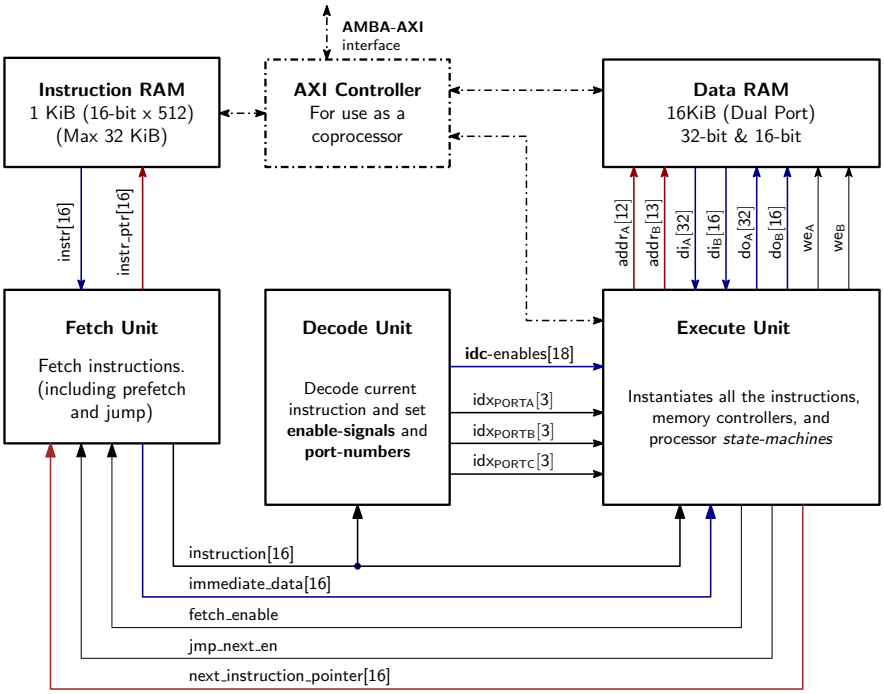


Figure 2: Overall Architecture of the Cryptoprocessor

LWE cryptoprocessor is quite similar to a general-purpose CPU in some aspects, but due to its specific nature certain features are quite different. One major difference is that the instructions are quite large in terms of the area as well as the number of clock-cycles required. Many of the instructions take a few thousand clock-cycles. As a result the fetch and decode units, which require a few clock-cycles are implemented without pipelining.

3.3 Fetch and Decode Units

The fetch unit reads the instructions from an external RAM based on the signals from the execute unit. As mentioned previously, some of the instructions also need an immediate address/data which is stored next to the instruction. The fetch unit handles this requirement transparently. It also accepts next instruction addresses and jumps to the given address when required. The decode unit decodes the instruction as per the instruction specification and generates separate enable signals (**i**nstruction **dec**oder enable - idc-enables) for each instruction. It also generates I/O port indexes to be used by the switch-matrix in the register units.

3.4 Execution Unit

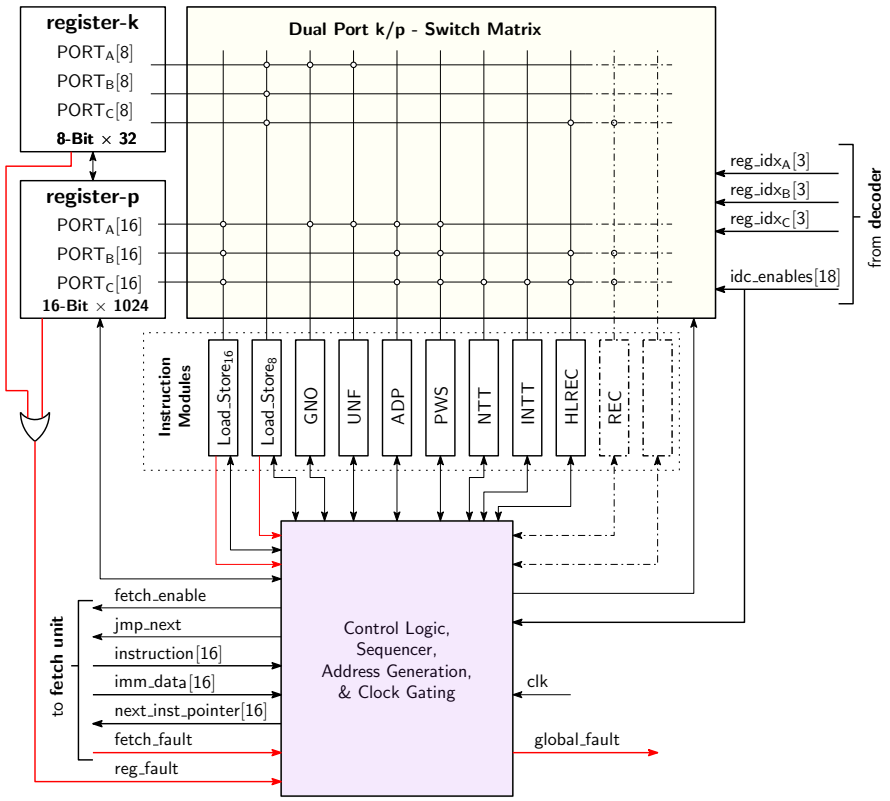


Figure 3: Execution Unit

The execution unit as shown in Figure 3 is the largest and the most important part of the processor. Because of the *subscalar* design, we decided to implement the control/logic unit within the execution unit. Also, all the modules for instructions are instantiated in the execution unit. Although, this design decision is straightforward; the implementation was challenging. This is because it was difficult to manage the large number of connections

between signals (415), registers (101), multiplexers (144) and the specific instruction modules (29). The major building blocks of an execution unit and their functions are described below.

3.4.1 Register Modules

As mentioned previously, registers can be accessed by instructions for read and write operations. As shown in Figure 4, a register module contains a set of registers which can be individually addressed and connected to any of the ports. The 3 ports can be used simultaneously and they work independent of each other. The only requirement is that a single register cannot be connected to more than one port. The functionality is achieved by using a simple dual-port bus matrix which allows any register to be selected and connected to any of the port. One of the major challenges while designing this unit was the large

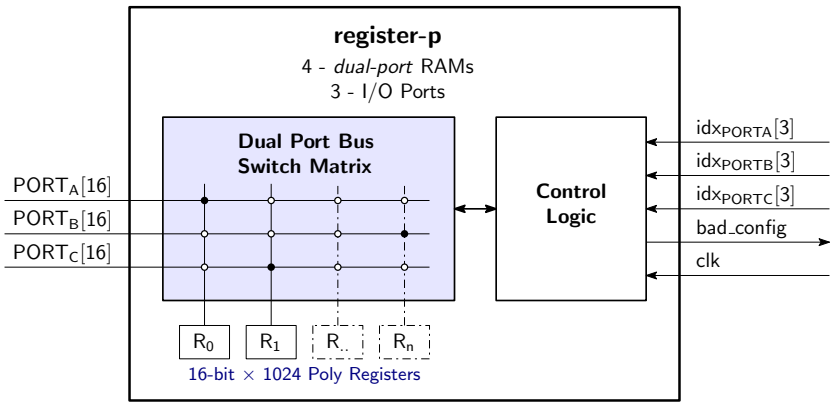


Figure 4: Register-P: 16-bit × 1024 Polynomial Registers. It is assumed that the polynomial coefficients would fit a 16-bit register.

number of connections in the switch-matrix. As every module needs to be connected to the Load-store units the multiplexer became very large. Leading to very long critical paths across multiple LUTs (used for MUXes). As the bus-matrix is an inherently combinatorial circuit, adding a large number of registers will cause reduced timing performance. As a result, we decided to allow for up-to 8 registers, but, implementing the least number of registers as possible is recommended (in our design we were able to implement all the algorithms using only 4 registers). If more registers are needed, pipelining may be needed to maintain good performance.

3.4.2 Load-Store Units

The load-store units and the associated instructions are one of the most used parts of the processor. This is because any data coming in and out of the processor has to pass through it. There are three load-store units, one each for the 16-bit poly, the 8-bit keys/associated data, and the large scratch-pad usage.

In order to maximize the throughput of the load-store unit, dual-port memories were used to transfer two chunks of data at a time. But, the load-store units use both 8-bit and 16-bit memory elements. Using asymmetric dual-port memories with a 32-bit port and a 16-bit port as external RAM, and generating requisite addresses; we were able to perform two transfers per-clock for both the 8-bit and 16-bit load store units resulting in optimal performance for this unit.

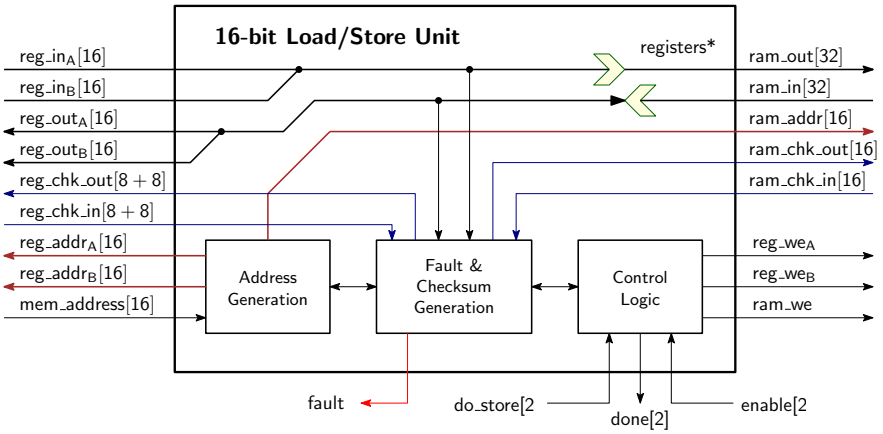


Figure 5: Load-Store Unit

3.5 Execution Sequence

Figure 6 shows the execution sequence of the processor. The processor starts by fetching an instruction from the instruction memory. The decoder then decodes the instruction and provides control signals and register port indexes. These control signals then enable

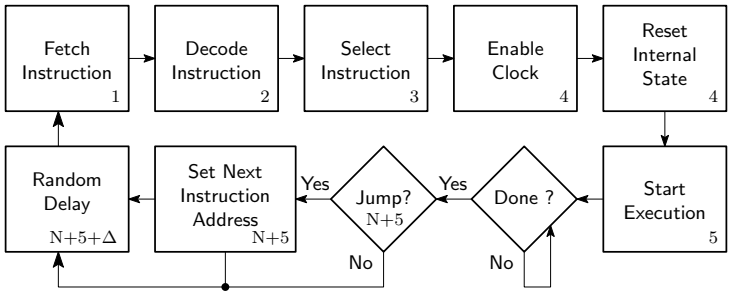


Figure 6: Execution Sequence

the specific instruction module instance and start the clock for it. Following this, the instruction module is reset (this is not needed for some instructions) and the execution starts. The execution unit then waits for the specific instruction to complete. When the instruction is complete, status is updated and jump requirements are evaluated and calculated. SFBT and SEL can set the error flag which causes the CJMP instruction to jump to the *immediate* address. If no jump is needed, the address is incremented. The execution unit then waits for the random delay delta (Δ) to prevent side-channel attacks. The new instruction is fetched only after the Δ clock cycles. This cycle repeats itself until a halt instruction (all zero) is encountered.

3.6 Instruction Set Design

Since this processor is designed to support multiple algorithms, the instructions are designed so that there can be a good overlap between designs. At the same time some specific instructions are needed to cater for algorithmic differences. The instruction set is designed mainly on the basis of instruction and register count, simultaneous register accesses and decoding speed. But, we did not try to optimize or encode large amount of information.

We have defined and implemented two types of registers as discussed below:

Table 1: Instruction set

Mnemonic	Instruction	Instruction Usage
	Load, Store and Jump Instructions	
LDP	*load* poly	1000 0001 | XXXX XPPP
LDK	*load seed or key*	1000 0010 | XXXX XKKK
STP	*store* poly	1000 0100 | XXXX XPPP
STK	*store seed or key*	1000 0101 | XXXX XKKK
CJMP	*jump when error flag*	1000 0111 | XXXX XXXX
EQP	*equality check poly*	0100 0001 | XX ppp ppp
EQK	*equality check key*	0100 0010 | XX kkk kkk
	Common Instructions	
UNF	*uniform*	0000 0010 | XX PPP kkk
GNO8	*getnoise* ψ_8	1000 1000 | XX PPP kkk
GNO16	*getnoise* ψ_16	1000 1001 | XX PPP kkk
ADP	*add polynomials*	0000 0011 | PP ppp ppp
PWS	*pointwise multiply*	0000 0100 | PP ppp ppp
BREV	*bit reverse poly*	0000 1001 | XX XXX PPP
NTT	*fwd NTT* ($\gamma = 7$)	0000 0101 | XX XXX PPP
INTT	*inv NTT* ($\gamma = 7$)	0000 0110 | XX XXX PPP
NTT2	*fwd NTT* ($\gamma = 1945$)	0000 0101 | XX XXX PPP
INTT2	*inv NTT* ($\gamma = 1945$)	0000 0110 | XX XXX PPP
HASHB	*sha3-begin*	0010 0000 | XX XXX XXX
HASHU	*sha3-update*	1010 0001 | XX XXX XXX
HASHF	*sha3-final*	0010 0010 | XX XXX kkk
TRNDP	true random *poly*	0010 0011 | XX XXX KKK
TRNDK	true random *key*	0010 0100 | XX XXX PPP
	NewHope Instructions	
HLREC	*helprec*	1000 0111 | PP ppp kkk
REC	*rec*	0000 1010 | KK ppp ppp
ENC	*encode*	0000 1011 | XX PPP kkk
DEC	*decode*	0000 1100 | XX KKK ppp
COMP	*compress*	0000 1101 | XX PPP ppp
DCMP	*decompress*	0000 1110 | XX PPP ppp
SUBP	*subtract polynomials*	0000 1111 | PP ppp ppp
	HILA5 Instructions	
SFBT	*safebits*	0001 0000 | PP PPP ppp
SEL	*select*	0001 0001 | PP ppp ppp
COD	*xe5-cod*	0001 0010 | XX KKK kkk
FIX	*xe5-fix*	0001 0011 | XX KKK kkk

1. Poly Registers (P) : These registers are used to store the 1024 polynomial coefficients. They are implemented using 16-bit wide dual-port RAM blocks. `Rp0 - Rp7` are used to represent poly registers.

2. Seed/Key Registers (K) : The initial seed required for key generation, secret keys, etc are stored in these registers. They are 8-bits wide, can be accessed using a dual-port bus and can store 32 bytes in a single register. These registers are represented using `Rk0 - Rk7`.

Table 1 lists the instructions supported by the processor. All the instructions are 16-bits long. Instructions such as *load, store, getnoise* and *helprec* require extra immediate address/ data fields (for example, the *helprec* instruction needs an initial seed). Thus, these instructions are followed by another field (consisting of 16-bits of content depending on the specific instruction).

As shown in Figure 7, this is indicated using the first bit of the opcode. PORTA,

PORTB and PORTC are used to indicate the register port indexes. The values of these ports are optional depending on the instruction being executed. For instance, for the point-wise polynomial multiply operation `'PWS Rp3, Rp2, Rp1'`, poly register `Rp3` is the output, while `Rp1` and `Rp2` are the inputs. PORTA and PORTB are used to set the register indexes for the inputs while PORTC indicates the register index for the output.

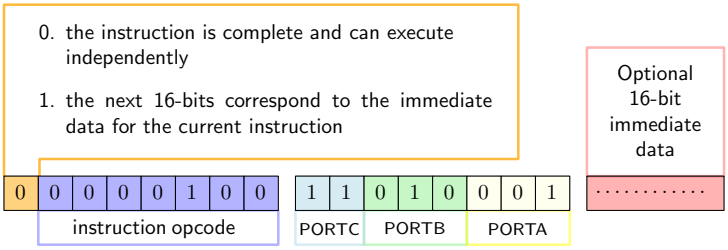


Figure 7: **Instruction format:** Using PWS point-wise polynomial multiplication.

3.7 Fault Protection

Over the years, several countermeasures have been proposed to protect against fault attacks targeted towards a multitude of designs and surfaces [BECN+06, SL80]. The following section discusses about the proposed checksum construction and the different countermeasures implemented to protect different attack targets.

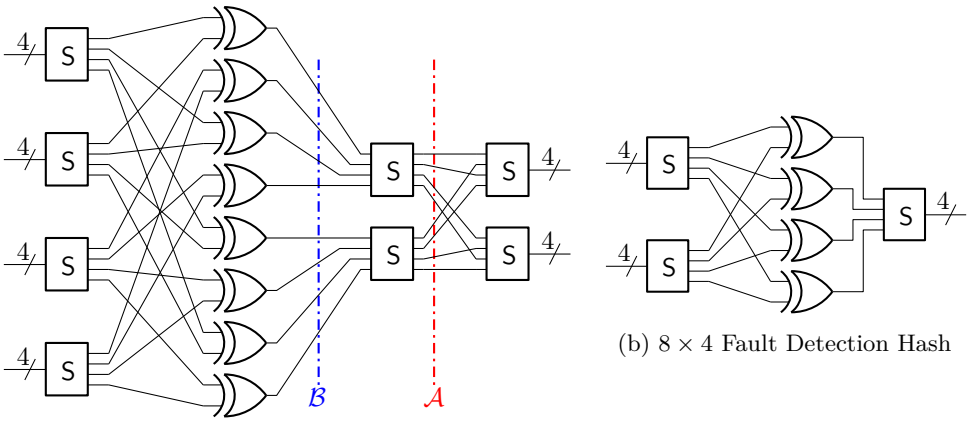
3.7.1 Fault Detection Hashes

Error detection schemes such as CRC (cyclic redundancy check) are used widely to verify the integrity of data stored on disk or transferred over networks. Similarly, parity checks by means of Hamming Codes are good for error detection. But, usage of such schemes is not justified as they are designed for correction of random errors and not maliciously injected faults. For example, if a fault is injected in the input, then the propagation of the error to the checksum is not guaranteed. Hence, we designed some low-latency checksum functions. These functions are used in multiple modules to ensure that random single or multi-bit faults lead to error conditions with high probability.

Figure 8 shows two constructions which have been specifically designed to detect faults with high probability. The basic idea behind the design is that whenever a single or multiple bits change on the input side, the output bits should change unpredictably, with high probability while ensuring uniformity. Such a design can be adapted from block cipher constructions with good differential properties. A good example can be AES, unfortunately, the AES-128 round function is complex and has high latency and area requirements. Hence, using a lightweight cipher is a better option. The designed hashes use the SPN (Substitution Permutation Network) structure, but with drastically reduced rounds and XOR for compression. The substitution layer uses the `PRESENT` SBOX because of its known good properties.

In order to design good functions for fault detection, we designed and analyzed several possible candidates. Some performed well and some were highly unsuitable. One of the techniques for analyzing a function was to see how *differential bias* propagates through it. We considered only the single and double bit *fault-models* for the candidates. Figure 9 shows the results for the function shown in Figure 8a. The *histograms* are calculated in two steps:

- Performing differential propagation analysis and computation of a bias distribution table for all the input bits. This results in a table containing 8 columns corresponding



(a) 16×8 Fault Detection Hash

(b) 8×4 Fault Detection Hash

Figure 8: **Fast Hash Functions for Fault Detection:** The current implementation primarily uses the 16×8 hash for instruction-pointer and memory protection. The 8×4 version can also be used for different applications, at the cost of reduced protection.

to the output bits, and 16 rows for the single bit inputs. We used 10^5 random inputs per-bit in the computation. The expected value should be 0.5 for all the entries.

- We subtract 0.5 from all the values in the table and get absolute values for all the entries. The *histogram* is then generated using all the values from this table.

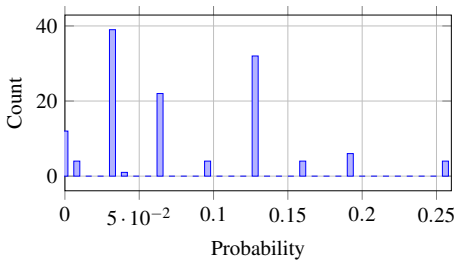
It is clear from Figure 9a and 9c that the *histograms* corresponding to the point $\mathcal{A}$ has many significant peak with high probabilities (close to 0.25), they also have many peaks throughout the probability axis. This means that for certain inputs there is some probability that the fault will not uniformly propagate. The Figures 9b and 9d corresponding to the complete function shows much better results. This limited analysis does not guarantee suitability for all applications, but it demonstrates good fault propagation results.

Both the hashes can be implemented within 4 levels of logic using 6-input LUTs. This allows the design to be fast enough for our purposes. Even stronger hashes can be constructed, but, the latency and long critical paths, make them harder to use without pipelining.

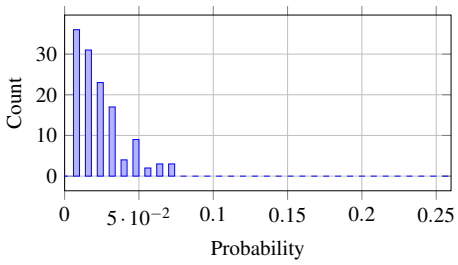
3.7.2 Protecting Critical Signals using Complimentary Duplicate Logic

Having multiple copies of the same circuit with equality checks protects an implementation from random fault as it is much harder to fault two signals simultaneously [BECN+06]. Such a countermeasure requires twice the area and memory resources. But, most Ring-LWE algorithms are quite complex and require large memories to store polynomials. Hence, addition of redundancy (especially in logic) should only be used sparingly. Therefore, only protecting critical signals like *state*, instruction, jump, etc. using duplicate logic is a good option. Although this will help prevent fault attacks, using inverted duplicate logic will add on to the overall fault resistance. Inverted logic is the implementation of some logic using '1' instead of '0' and vice versa. This makes insertion of faults further difficult as now two signals have to be faulted but with opposite polarity.

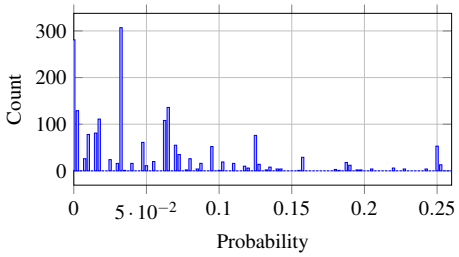
Differential Control Using the same technique, the signals *done* and *enable* can also be protected avoiding instruction skip (for example by setting *done* signals).



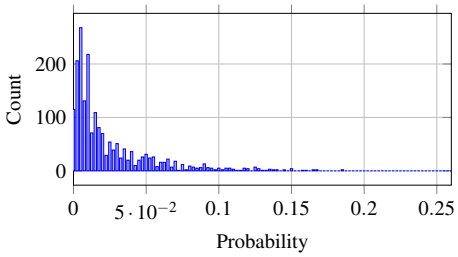
(a) Single bit at point A.



(b) Single bits at output.



(c) Double bit at point A.



(d) Double bits at the output.

Figure 9: Differential bias histograms for single and double bit random faults.

3.7.3 Protecting the instruction pointer

This is one of the most common targets for fault injection. The ability to skip/manipulate instructions (especially conditional jumps) allow attackers to thwart many software based protection schemes. To protect against such attacks we use two countermeasures:

- **Hashing:** We add an hash to the *instruction pointer* using the above mentioned 16×8 function. The hash is written every time the value is updated, and the hash value is verified at every clock cycle.
- **Inverted Logic:** To further protect the *instruction pointer*, we also implemented a duplicate pointer with inverted logic. Every time the pointer needs a increment, the value is inverted incremented and inverted again: $ptr_{n+1} <= \sim(\sim ptr_n + 1)$. The inverted logic also contains the hash function checks.

Both these circuits are evaluated at every clock cycle and they generate error signals whenever a fault state occurs. Figure 10 shows the detailed instruction pointer update function.

3.7.4 Protection against Control Flow (FSM state) Modification

Proper execution of internal state machines are critical to the overall operation of any processor. If the attacker can manipulate the current state register or next state calculations/registers for FSMs, security of the design would be compromised. Similar to instruction pointer manipulations, any software countermeasures can be rendered useless. We, use duplicate inverted logic state machines with concurrent matching on all critical FSMs in the cryptoprocessor. Figure 11 shows the detailed implementation.

3.7.5 Protection against Memory Faults

Any fault injected to the data memory leads to incorrect results. There are many results on a wide variety of cryptographic constructions, where even a single bit error leads to the

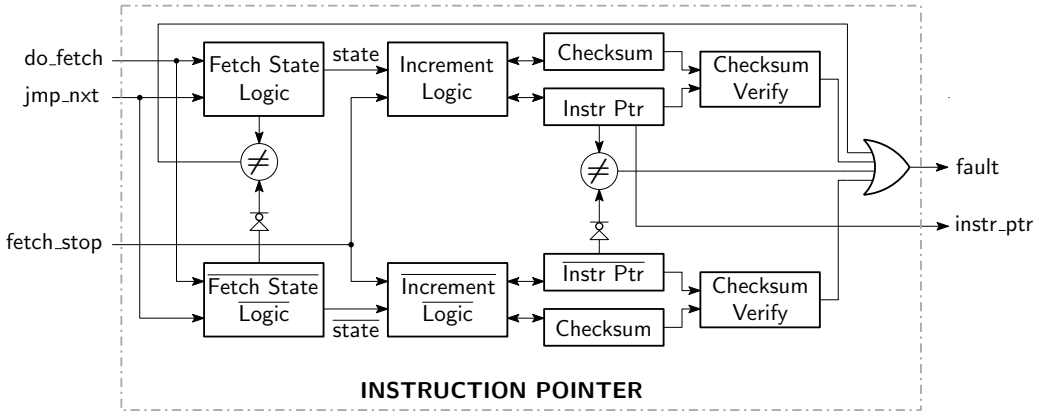


Figure 10: **Instruction Pointer Update:** Fault Countermeasures

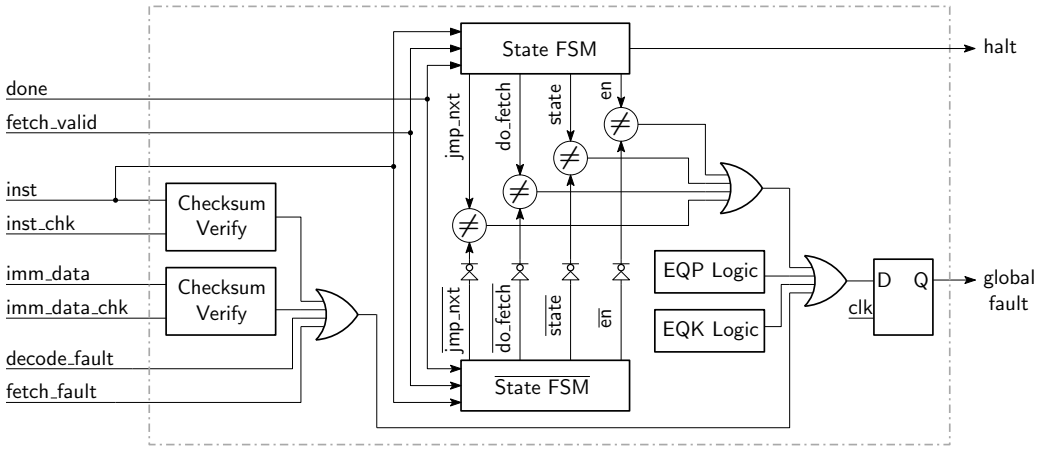


Figure 11: **Execution State FSM:** Fault Countermeasures

recovery of the entire state or secret keys. Similarly, faults in instruction memory can lead to arbitrary code execution. This can also be used to mount several attacks.

For instruction and data memory extra error detection data (result of a hash) is stored along-with the memory itself. Basically, with every 16-bits of data 8 extra bits of hash are stored. Any load operation from the memory verifies the data integrity before passing the data to other modules. Likewise, during write operations to the memory, the error detection data is calculated and stored. This ensures any access to the external memory is secure and fault-attack resistant.

3.7.6 Additional Protection: Instruction Cycle Count

As all the instructions in the implementation are constant-time and take the same number of clock cycles. Counters can be used to ensure that they indeed do take the correct number of clock cycles. This ensures once an instruction module starts execution, it completes fully and is not skipped in between. For example the NTT instruction requires 6206 clock cycles, any injected fault to the numerous control logic signals can cause it to end prematurely, possibly with minimal change to the operands, leading to attacks. Even though, one can use techniques like duplication and inverted logic for all the modules and all state machines, such a design would lead to a significant increase in area. By ensuring clock-cycle count correctness, we protect against this powerful attack vector with relatively cheap/minimal

hardware resources.

3.7.7 Fault Tolerance: Direct Memory Access Isolation

Although many of the instructions can be allowed to access the external memory, we chose to maintain an isolation. Only two instructions LDx and STx are permitted to load and store data to and from the external RAM. This is done in order to prevent fault attacks during the execution of a cryptographic operation. More specifically, whenever data is read the checksum is also calculated/validated, similarly, during every write operation, a new checksum is calculated and written to the memory with the data, to be used during verification later. This protects the data in memory from maliciously injected faults. This isolation also simplifies both the instruction set design and the overall architecture.

3.7.8 Fault Tolerance: Equality Instructions

The two instructions EQK and EQP have been added to help in the implementation of software based fault countermeasures. These instructions compare two register values and generate a reset signal whenever there is a mismatch. So, in software/assembly, the implementation can perform an operation twice (using different memory/register) and then compare the results to ensure correctness. For example, Listing 3 shows a code snippet to use the cryptoprocessor's equality instruction EQP for the safe execution of getnoise operator to generate the centred binomial distribution.

```
LDK Rk0, 0       // noise (from memory 0-31)
GN016 Rp0, Rk0, 0  // poly_gno(Rp0, noise, 0)
GN016 Rp1, Rk0, 0  // poly_gno(Rp1, noise, 0)
EQP Rp0, Rp1     // verify the two results
```

Listing 3: Example assembly listing for verifying the output of getnoise. Errors in EQP would reset the module.

3.8 True Random Number Generator

Designing a good true random number generator is challenging and requires several considerations. There are several techniques to generate true random numbers in FPGA, many techniques like *metastability*, *delay lines*, have been used for this purpose. In this work, we use, ring oscillator based TRNG's because they offer good performance on a wide variety of platforms, we have implemented, designed and tested a TRNG with 32 rings, using a design similar to the one presented in [WT09]. The secure element is followed by a shift register. All the modules share this common source of randomness whenever required.

3.9 Side Channel Protection

A large number of countermeasures such as masking [ITT01, KHL11], threshold implementation [NRR06] etc. have been developed over the years to protect against these attacks. But, most of the fully secure countermeasures are expensive or difficult to apply on certain schemes. As a Ring-LWE implementation contains several large modules there is a large attack surface for SPA/DPA, so adding specific countermeasures would be difficult and expensive. In a typical scenario for key-exchange, multiple traces of the same execution are not available, so DPA cannot be attempted easily. SPA or *Template Attack* are still possibilities. In this work, the following countermeasures are used to provide a good balance between performance and security:

- *Random Delays:* Delays added to the instruction scheduler can make the power analysis very difficult. This is especially true for FPGA implementations where the

Signal to Noise Ratio (SNR) is typically very low for good quality automated trace alignment like Dynamic Time Warping. This random delay (dummy clock cycles) is generated from the common/system Ring-Oscillator based TRNG. A 4-bit output (value 0-15) is used per instruction; this range can be adjusted during synthesis, as per requirements.

- *Address Randomization:* We implemented a randomization scheme for I/O operations. For example, during ADP, PWS, GNOx and many other instructions; the order of operation does not matter. Hence, we used a permutation for addresses randomization initially created at system start-up/reset using random data and the *knuth-shuffle* technique. The permutation is updated during the random wait cycles at the end of every instruction.

To further improve resistance against such an attack, the system clock can be dynamically varied using a circuit as demonstrated in [BVR+13]. One should note that such a module normally reduces the clock frequency and performance and should be carefully designed to prevent glitches.

4 Implementation & Optimizations

In this section, we present details regarding the hardware implementation and the corresponding design decisions.

We implemented a general lattice cryptography accelerator for Microblaze/ARM based soft/hard cores within modern FPGAs. For this, instruction modules were individually implemented and a controller state machine was added along with fetch and decode units. This allows for a flexible design and full support for multiple schemes. Further, we focus on achieving a good area-vs-performance trade-off.

4.1 A Fast and Configurable NTT Module

One of the most expensive operation in LWE/RLWE is the polynomial multiplication. Many of the current PQC schemes explicitly define and use NTT/NTT^{-1} (section 2.2.1) as a core operation for this purpose. As NTT requires multiplications and modular reductions to be performed at every step ($\frac{N}{2}logN$ butterflies), it may take a very large number of clock cycles depending on different implementations. In view of this, implementing a fast and efficient NTT module needs careful consideration. Roy et.al. in [RVM+13] present comprehensive and detailed implementation and optimization techniques for NTT. Further details regarding other NTT implementation can be found in [APS13, PG13]. The primary implementation aspects of our design are as follows:

- **Modular Multiplication and Reduction:**

  The NTT algorithm requires multiplication operations between the coefficients and the *twiddle* factors. The result of these multiplications need to be reduced so that they do not exceed the available 16-bits. One should also ensure that minimum number of reductions are performed as they tend to be expensive. Techniques like conditional subtraction for modular reduction, though faster, are not constant time. Thus, they cannot be pipelined effectively, apart from leaking side-channel information. Even though, the conditional subtraction can be used after additions, at high clock rates, three operations (with internal carry-chain) cannot be combined.

  Considering the above factors, the Montgomery and the Barrett reduction algorithms from [ADPS15] were used in this implementation. In order to further improve performance, these modules were pipelined.

- **Twiddle Factor Unit:** The coefficients needed for multiplication/scaling and the *butterfly* operations can be obtained either by pre-computation and stored in a RAM/ROM or they can be generated on-the-fly. Generating these coefficients instead of storing them saves area in ASIC implementations. However, it adds to the design complexity. Most moderate FPGA's have a good amount of BRAM (Block RAM) blocks and storing the *twiddle* factors is not a major issue and hence we take this approach. As we store the *twiddle* factors in RAM, it is possible to update them on the fly using a custom AXI-Controller to switch between different algorithms very easily. A large volume full-custom ASIC should preferably use a more complex design with on-the-fly generation with some pipeline stages.

```
for (i = 0; i < 10; i++)
{
    for (m = 0; m < d; m++)
    {
        t = 0;
        for (n = m; n < N - 1; n += 2 * d)
        {
            W = omega[t++];
            U = a[n]; V = a[n + d];
            a[n] = bred(U + V);
            a[n + d] = mred((W * (U + 3 * Q - V)));
        }
    }
    d = 2 * d;
}
```

Listing 4: NTT Algorithm, using the Gentleman-Sande butterfly.

- **NTT Butterfly Address Generation:** The NTT algorithm shown in Listing 4 has two inner loops, the m-loop and n-loop. It is clear that, for higher values of d, there will be a lot of switching between the m and n loops, even though the number of overall operations remains the same. For a pipelined implementation will lead to the introduction of a lot of bubbles whenever the inner loop ends. As a result, the pipeline will be stalled for several clock cycles. To alleviate this problem, the logic of two loops can be combined into one. The resulting module generates 5120 address pairs in the correct sequence, with no delays and no extra cycles (stalls). These addresses correspond to the $512 \times 10 = 5120$ *butterfly* operations. The control unit then uses these addresses to perform the *butterfly* operations in the correct order.

- **Control and Memory Address Generation:**

  This module generates control signals for all operations in the NTT module. As there are multiple pipeline stages, extra delay registers are required for the addresses and the enable signals. The control unit generates signals depending on the current operation.

Figure 12 shows the overall architecture of our NTT module. The NTT module can perform both forward and inverse transforms using the same hardware just by changing some control signals. The *twiddle* factor RAM can store constants for two different algorithms with different values of γ. This allows our NTT module to be very flexible as consecutive NTT invocations may be used for different PQC algorithms.

Forward Transform: The first step of the forward transform is the pointwise multiplication of the coefficients of the input parameters with the constants in the *twiddle* factor RAM ($g^x \times \mathcal{R}$). This is also known as *scaling*. The multiplication by $\mathcal{R} = 2^{18} \bmod q = 4075$

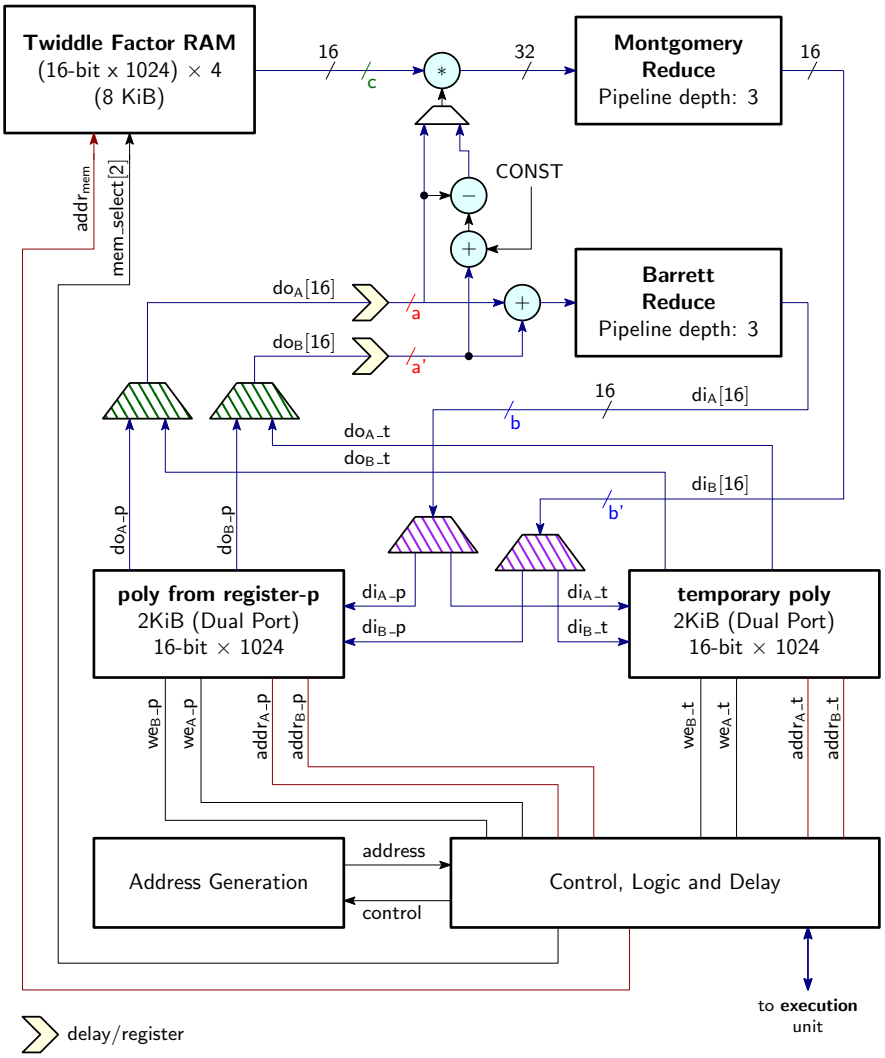


Figure 12: Pipelined NTT Module.

is required as the *Montgomery reduce* step expects the inputs in the Montgomery form. This pre-computation saves a multiplication and a reduction.

Elements from the poly register are read one by one from the port $\mathsf{do_A}$ marked as 'a' and multiplied to the *twiddle* factors marked as 'c'. The result is then reduced and is written back using the port $\mathsf{di_B}$. There are 3 register stages along the path. As a result, it takes $(1024 + 3 + \epsilon = 1027 + \epsilon)$ [1] clock cycles in total.

The second step is the actual NTT operation. This step is an iterative process with three levels of loops. For $N = 1024$, the outer loop iterates $\log_2(1024) = 10$ times. For this implementation, the NTT address generator outputs an address pair every clock cycle. The actual implementation can be done in many ways. The simplest way is to read data from addresses (for the current *butterfly*) process and store, this will take $5120 \times 2 = 10240$ clock cycles. The reason for this is that we cannot read and write at the same clock cycle at two different addresses using normal dual port BRAM memories. A better method, and the one that we use, is to utilize two memories and swap (read and write operations) between them for every layer of the NTT for every 512 *butterfly* operations. This also

[1] ϵ is the time it takes to load data, initialize and finalize an instruction. Typically, ϵ lies between 2 to 5.

ensures that the *butterfly* calculation and the *reduction units* are fully utilized. So initially, data from the operand register is read and the results are stored in a temporary poly register. Then after 512 *butterfly* operations, the direction is reversed and the data from the poly register is read and the results are stored to the operand register. This swapping continues till all the 10 levels of the NTT are complete.

The total number of clock cycles for the main part of NTT is $(512 + 5) \times 10 = 5170$. Five clock cycles are required for the pipelining and state machines. As a result, the total number of clock cycles for the NTT instruction (scaling + main NTT) is 6206 (5170 + 1024 + 12). The 12 cycles are needed for the deep pipelines, internal module resets, NTT state machine, and initialization steps. The extra cycles can possibly be reduced, but at negligible real benefits.

4.2 Modular Reductions

LWE/RLWE schemes need reductions of some form or the other. As mentioned in section 4.1 we work with values in the Montgomery domain and then perform Montgomery and Barrett reductions whenever needed. Initial implementation of both the reduction algorithms were running at 96 MHz and 175 MHz respectively. Since, these reduction were required in many places, they became the critical path for those modules resulting in an overall low performance. Hence, to overcome this performance bottleneck, we added several pipeline stages to reduce the overall critical path for both the modules. We required two versions of the reduction: one with 2 register stages and another with 3 stages. Basically, when the input values are used directly without significant (time-sensitive) operations, the input register stage can be removed. Figure 13 shows the 3-stage pipelined reduction circuit. The algorithm presented in Listing 5 is pipelined with 2 register stages.

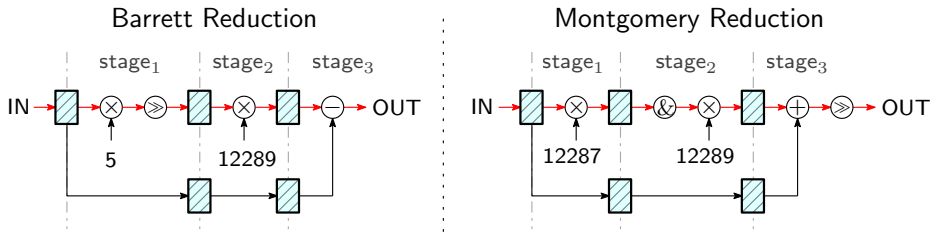


Figure 13: Pipelined Modular Reductions

It is known that the Montgomery algorithm does not support the modulus with an even integer. As calculating modulus using a division operation is very expensive in hardware, it is not preferable to use this technique. Hence, in order to calculate modulus by an even integer (for example $12289/4 \approx 3072$), we used the technique presented by Koc [Koç94]. This modulus is required for the SafeBits operation in HILA5. As one can see from the Listing 5, modulus by 3072 has been reduced to modulus by 1024 and modulus by 3. We know that modulus by 1024 is free in hardware, thus resulting in an efficient implementation in hardware. The same technique can easily be used for modulus by different even integers.

```
ushort mod_even(uint a) {
  uint x1, x2, x, y, u ;
    // a % 3 using Montgomery Reduction
    u = (a * 87381);
    u &= ((1 << 18) - 1);
    u = 3 * u;
    a = a + u;
    x1 = a >> 18;
```

```
    x2 = a % 1024;
    // Combining the two; mod (3 and 1024)
    // to get (a mod 3072)
    y = ((x2 - x1) * 683) % 1024;
    x = x1 + (3 * y);
    return (x);
}
```

Listing 5: Modulus by 3072 ($R = 2^{18}$)

4.3 Error Recovery Functions

Error recovery / correction is one of the most important steps in a LWE/RLWE algorithm. This step is critical to correct the introduced errors and obtain the desired intermediate values which are then used to compute the shared secret. Even though reconciliation operations are cheap in terms of clock cycles, they can lead to poor performance. For example, the f and g functions in `NewHope-USENIX` has many multiplication operations which if not pipelined will result in long critical paths. Hence, the reconciliation for `NewHope-USENIX`, `NewHope-NIST` and `HILA5` needed significant pipelining to achieve good performance.

As shown in Figure 14, 8 stage pipelines were required for both the f and g functions in the HelpRec and Rec functions of `NewHope-USENIX` respectively. The execution of

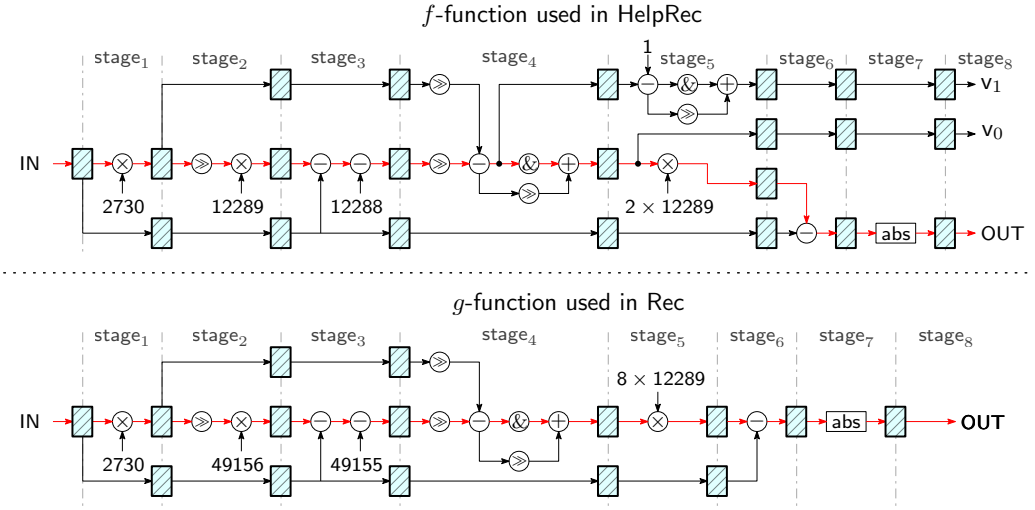


Figure 14: f and g internal functions used for reconciliation in `NewHope-USENIX`.

HelpRec is shown in Figure 15. The implementation is fully pipelined with a latency of 9 cycles. As a result it takes 1024+9+ϵ clock-cycles to complete. Similarly, the Rec instruction takes 1024+9+ϵ clock-cycles. As the Rec instruction generates a key bit every cycle, a byte of the key is written every 8 clock-cycles. Similarly, for `NewHope-NIST`, the error-correction functions NHSEncode and NHSDecode are pipelined with a latency of 2 cycles. For these modules, we used the dual port RAM to write the key bytes. This results in a total of 512+3+ϵ clock-cycles for both the modules. The functions NHSCompress and NHSDecompress are also pipelined with a latency of 10 cycles for the former and a latency of 3 cycles for the latter. In case of NHSDecompress, we used dual port RAM to write the *ciphertext*. This results in a total of 1024+8+ϵ clock-cycles for NHSDecompress and 512+3+ϵ clock-cycles for NHSCompress. Figure 16 presents the 3-stage pipelined architecture for `HILA5` Safebits function. As discussed in section 2.2.1, this module uses

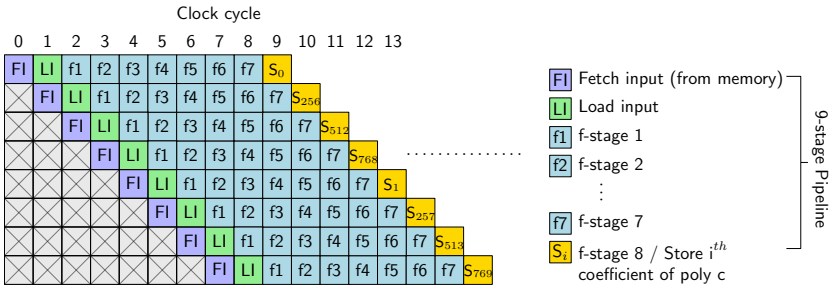


Figure 15: Pipelined HelpRec Function from `NewHope-USENIX`

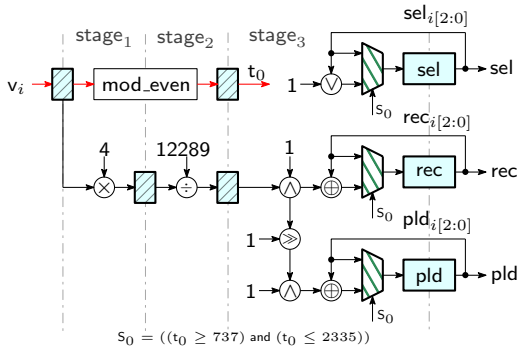


Figure 16: Pipelined HILA5 Safebits Function

the algorithm presented in Listing 5. It is also interesting to note that on every iteration, this function sets a bit of the three outputs at a time. Hence, we write the output to the memory after every 8 clock-cycles. Similar pipeline strategy has been used for the implementation of Select function used in `HILA5`. These functions are data dependent and may fail depending on the number of obtained payload and reconciliation bits. The maximum number of clock-cycles required for our implementation is $1024+5+\epsilon$. All the subcodewords in XE5_COD of `HILA5` are computed in parallel. Hence, this module is not pipelined. Same is the case with XE5_FIX of `HILA5`.

4.4 Pipelining ChaCha20

A simple implementation for a fast version of ChaCha20 would have a design with one round per clock-cycle. Unfortunately all the operations in the Quarter-Round operation are chained; leading to very long critical paths. On the Zynq-7000 platform, the maximum frequency was a mere 74 MHz, leading to a performance bottleneck. Hence, to overcome this issue, we implemented a using 2-stage pipeline for the ChaCha20 Quarter-Round as shown in the Figure 17. Using the 2-stage pipeline architecture, we were able to achieve a frequency of 204 MHz, resulting in performance gains of about 175%. The points I and II in the figure indicates possible locations for adding more pipeline stages to further increase the performance. But, this would results in addition of two more 512-bit register stages; a significant increase in area. So, in order to keep a balanced design, we ony used two pipeline stages for Chacha20.

4.5 Resource Sharing

Resource sharing is a common optimization technique in hardware implementations. The primary benefit is reduction in the area/number of FPGA resources being used. One side

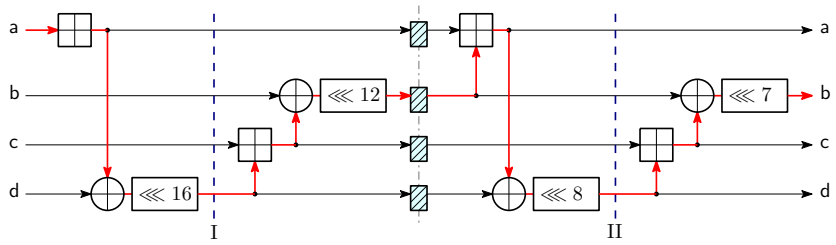


Figure 17: Pipelined ChaCha20 Quarter-Round.

effect is the reduction in performance as sharing is often implemented using multiplexers and de-multiplexers. This is not a major concern when the modules are not a part of the *timing critical path*. We used resource sharing at many places in our implementation. Most notably, the KECCAK module was shared between the UNF and the SHA-3 hash units. Similarly, the ChaCha20 module was shared between (GNO and HLREC). This led to an increase in the number of LUTs in the Execute unit, because of extra multiplexers. However, as the shared modules are large, there were significant savings overall. In total saving about 5000 LUTs and about 1000 *flip-flops*, which is almost 37% of the current area in LUTs.

4.6 Use as a coprocessor

The proposed processor can be used in a standalone mode; but using it as a *coprocessor* as shown in Figure 18 significantly improves flexibility and ease of use. For this purpose, a custom AXI interface controller was implemented. The interface exposes some control and status registers to the CPU core, it also provides direct *memory-mapped* access to the Instruction and Data memories of the cryptoprocessor, while handling fault signals and checksum computations transparently. The AXI Slave interface controller connected to a processor (Microblaze/ARM) can be used to update the instructions, load/set keys to the data memory, and control the *coprocessor* to significantly improve key-exchange performance over embedded software implementations. The comparison results from the tests are presented in Section 5.4.

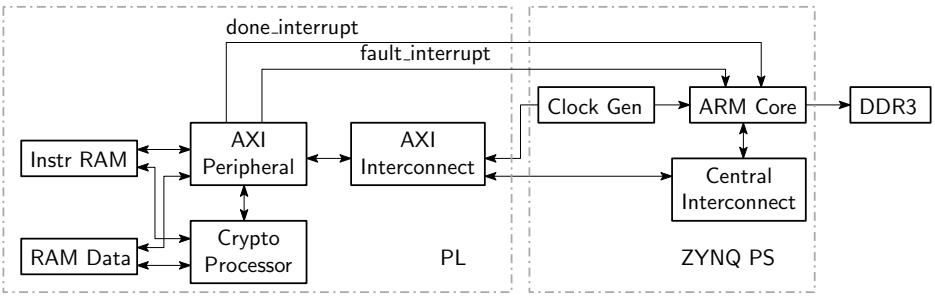


Figure 18: Usage as a cryptographic accelerator on the Xilinx Zynq Platform.

4.7 Clock Network & Power Consumption

The *fan-out* for the CLK network after initial synthesis was 8663. This meant that the clock signal had to reach 8663 points in the FPGA with minimal *skew*. Increased *skew* causes *setup* and *hold* time violations, leading to reduced maximum clock frequencies. Further, the large number dynamically switching clock signals causes increased power

consumption. To counteract both these problems, we implemented *clock-gating* in the design by manually adding BUFHCE (Regional Horizontal Clock Buffer, with enable) and BUFGCE (Global Clock Buffer) to all the instruction modules. In total 21 such modules were added, reducing the average *fan-out* to about 400. Clock was only enabled to the instruction modules one cycle before start of operation. These changes led to improved timing and reduction in power consumption.

4.8 Assembler

An assembler was written to assemble handwritten assembly routines for the processor to binary files. This is not strictly necessary, but it significantly helps in writing protocols and code for debugging; primarily because fiddling with bits tends to be very error prone.

```
void newhope_shareda(
    unsigned char *sharedkey,
    const poly *sk,
    const unsigned char *received)
{
    poly v, bp, c;

    decode_b(&bp, &c, received);
    poly_pointwise(&v, sk, &bp);
    poly_invtt(&v);
    rec(sharedkey, &v, &c);
}
```

Listing 6: C code

```
1000000100000000 <= LDP Rp0, 5120
0001010000000000
1000000100000001 <= LDP Rp1, 128
0000000010000000
0000010010001000 <= PWS Rp2, Rp1, Rp0
0000100100000010 <= BREV Rp2
0000011000000010 <= INTT Rp2
1000000100000000 <= LDP Rp0, 7168
0001110000000000
0000101011010000 <= REC Rk3, Rp2, Rp0
1000010100000011 <= STK Rk3, 96
0000000001100000
```

Listing 7: Equivalent binary codes

Listing 6 contains a partial and simplified[2] C implementation for the *shareda* operation in `NewHope-USENIX`. Listing 7 shows the assembled binary opcodes generated from the user written equivalent mnemonics and operands. RpX are 16-bit poly registers, while RkX are the 8-bit key/associated data registers. Full code for `NewHope-USENIX` can be found in Appendix A.

5 Implementation Results

This section presents performance evaluation and comparison with other implementations.

5.1 Evaluation Platform

We used Xilinx Vivado 2018.1 for synthesis, placement and routing. The results are presented for two FPGAs Xilinx Zynq-7000 XC7Z020CLG484-3 (28nm) and Zynq Ultrascale+ XCZU4EGSFVC784-3 (16nm). The latter is significantly faster and more efficient due to the modern architecture and the 16nm FinFET+ Programmable Logic, but, the former is much more common and we use it to compare this work with other similar designs. We present data for the highest (-3) speed grade; in our experiments, the more common (-2) speed grade has about 10% performance degradation for our design in case of both the FPGA's. The timing and utilization results are quite sensitive to the synthesis and implementation strategy as well as the timing constraints. We used PerfOptimized_high for synthesis and ExtraTimingOpt for the implementation strategy. This resulted in a slightly large, but better performing implementation.

[2]The SHA-3 at the end is not shown for clarity.

5.2 Results and Discussion

Table 2 shows implementation results for all the modules. It is clear from the results that the functions SHA3 and ChaCha20 used by the *uniform* and *binomial* sample generation modules, occupy most of the area and have the longest timing path (hence are the slowest). There are many possible low area implementations possible for these modules, but those designs have a comparatively poor performance due to their large latencies. As we are targeting high performance, using modules with low latency is important. In order to obtain optimal performance from the target FPGAs, we used the resources like DSPs and BRAMs whenever suitable. We tried to ensure that we don't overuse certain hard-blocks; and attempted to maintain a good LUT/FF to BRAM/DSP ratio.

In order to evaluate timing (maximum frequency) for the modules, we synthesized and implemented each of them separately as well. For two of the modules (ChaCha20 & Keccak) which had a large number of I/O ports, it was not possible to implement them separately. For them, we had to use `out_of_context` mode for the synthesis, which does not connect the module to the pads. This slightly improves the overall timing. This was not an important factor as the modules are anyways not connected to the pads in the final design.

It is interesting to note the wide variation in the maximum achievable frequency for each of the modules. This is quite significant for the XCZU4 FPGA. As the processor is *subscalar* with large instruction latencies, one possible future extension is to use multiple *PLLs* in the FPGA to generate multiple clocks and run different modules at different clock frequencies (this is a well known optimization). This can easily improve performance by 15-20 percent.

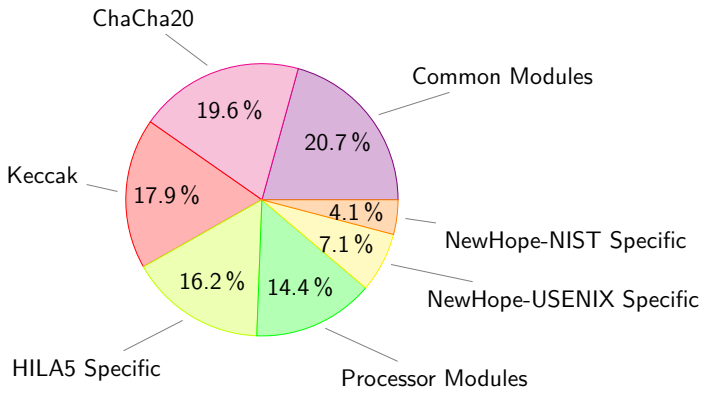


Figure 19: Breakup of the different components in terms of the number of LUTs required for `NewHope-USENIX`, `NewHope-NIST` and `HILA5`

As mentioned earlier, as well as shown in Figure 19, the area taken up by ChaCha20 and Keccak is about 37%, which is quite large. In order to reduce the overall area they can be replaced by lightweight hardware oriented primitives of similar strength.

Any hardware or software implementation can be optimized to a large extent. But once significant level of optimization has been achieved, further optimizations either require a lot of effort or further gains are minimal. In this work, we kept on optimizing individual modules in terms of timing (frequency improvements and reduction in latency), and in terms of area (by sharing and using alternative logic), till any further optimization would provide 5% or more benefits without adversely affecting the overall results. As a result, we have a design which is well optimized. Of course, there are certain modules which can be further optimized with small benefits but we do not attempt since the gains are not likely

Table 2: Implementation Results for all the modules used in the three implemented schemes. The shown area utilization values are for XC7Z020; the values for XCZU4 are not shown as they are very similar (within 2-6%). These results are for the unprotected implementation. The (SCA + FAULT) implementations add about 600 LUTs and 500 FFs without any effect on the timing performance.

Component	**Clock**	**Frequency (MHz)**		**Area Utilization**				
	Cycles	XC7Z020	XCZU4	LUTs	FFs	BRAM		DSP
Processor Modules	(latency)	28nm	16nm			36K	18K	
Fetch Unit	1	485	724	9	53	0	0	0
Decode Unit	1	592	850	33	40	0	0	0
Execute Unit (logic & control)	-	**190**	**406**	1449	14	0	0	0
- Load-Store (16-Bit)	512	378	850	41	18	0	0	0
- Load-Store (8-Bit)	16	520	850	21	13	0	0	0
- 8-bit Register Module	1	290	651	198	12	0	4	0
- 16-bit Register Module	1	271	595	420	12	0	4	0
Multipurpose RLWE Modules (Including `NewHope-USENIX`)								
- Keccak (SHA3)	-	238	408	2315	0	0	0	0
- ChaCha20 Stream	-	276	664	321	428	0	0	0
- ChaCha20	40	**204**	480	2534	1037	0	0	0
- Polynomial Add/Sub	512	275	793	230	214	0	0	0
- Polynomial Multiply	512	258	515	163	316	0	0	12
- Polynomial Bit-Reversal	1024	321	811	18	14	0	0	0
- Combined Pipelined NTT	6206	251	528	343	493	0	6	3
- GetNoise (w/o ChaCha20)	2872	287	599	106	379	0	0	0
- Sampler (w/o Keccak)	1837	238	**408**	1492	1956	0	0	0
`NewHope-USENIX` Modules								
- Helprec	1043	226	605	411	269	0	0	3
- Rec	1043	220	588	504	380	0	0	1
`NewHope-NIST` Modules								
- Encode	519	443	850	45	31	0	0	0
- Decode	515	277	677	182	108	0	0	0
- Compress	1036	195	591	225	67	0	0	1
- Decompress	518	286	850	81	67	0	0	0
`HILA5` Modules								
- safebits	952	**191**	**430**	244	170	0	0	2
- select	998	210	533	159	131	0	0	3
- xe5_cod	276	322	673	873	616	0	0	0
- xe5_fix	224	291	605	816	341	0	0	0
Total	-	-	-	13224	8272	0	16	24

to be significant.

5.3 Comparison

Table 3 shows a comparison of our design with other post-quantum schemes. As of today, only two directly comparable FPGA implementations by Kuo et al. [KLC+17] and by Oder et al. [OG17] exist. Even though both are FPGA implementations on similar hardware platforms, architectural differences and choices in design strategies make a fair comparison difficult. For completeness, we include other implementations of PQC schemes [KAKJ17, RVM+13, PG13, HMO+16] in Table 3. However, our implementation

Table 3: Performance Comparison

Implementation	Time			Area				Device
	Freq. (MHz)	Cycles ($\times 10^3$)	Time (μS)	LUT	FF	RAM Blocks	DSP	
Parameters: $n = 1024,\ q = 12289$								
NewHope-USENIX	**190**	30.6/33.5	160/175	13244	8272	18	24	XC7Z020
NewHope-NIST		30.6/34.0	160/178					(28nm)
HILA5		30.1/30.0	160/156					*(this work)*
NewHope-USENIX	**406**	30.6/33.5	75/82	13961	8149	18	25	XCZU4EG
NewHope-NIST		30.6/34.0	75/83					(16nm)
HILA5		30.5/29.8	75/73					*(this work)*
NewHope-USENIX	133	6.9+2.8	51.9+21.1	18756	9412	14/14	8/8	XC7Z020
[KLC+17]	/131	/10.3	/78.6	/20826	/9975			(28 nm)
NewHope-	125	171	988+473	5142	4452	4	2	XC7A35T
Simple [OG17]	/117	/179	/1434	/4498	/4635	4	2	(28 nm)
Parameters: $n = 511, 503$ bit primes								
SIDH [KAKJ17]	177	5967	33700	30031	24499	27	192	7VX690
SIDH [KAMK16]	181	3800	20900	26659	19882	40	192	7VX690
Parameters: $n = 512,\ q = 12289$								
RLWE [RVM+13]	278	13.3/5.8	47.9/21	1536	953	3	1	V6LX75T
RLWE [PG13]	251	13.8/8.8	54.9/35.4	5595	4760	14	1	V6LX75T
Parameters: $n = 125,\ q = 4096$								
LWE [HMO+16]	125	98.3/32.8	786/228	6152	4804	73	1	S6LX45T

cannot be directly compared to these works because of significant differences in design, parameters or mathematical structure.

5.3.1 Comparison with `NewHope-Simple` Implementation (NHS) [OG17]

The authors of [OG17] present an implementation targeted towards low-cost FPGA's. The said implementation is quite small (5K LUTs) but takes a large number of clock cycles ($\approx$ 170K) for server and client each. In comparison, our design requires 12.9K LUTs for three algorithms and needs $\approx 32K$ clock cycles. Comparison in terms of area is further made difficult as [OG17] uses Trivium instead of ChaCha20. Considering speed, our design runs about 65 MHz faster on the same FPGA architecture. This means that our design is overall 8.6 times faster for the complete NHS execution.

5.3.2 Comparison with `NewHope-USENIX` Implementation (NH) [KLC+17]

The authors of [KLC+17] focused primarily on performance. In order to do so, they used a 128-bit bus and replicated hardware modules to improve performance. For example, they use four instances of Uniform and Binomial sampler. Similarly, they use 8 instances of *multiplier*, *adder* and *reduction* modules. They also use 4 parallel *butterfly* operations in the NTT module. In contrast to this, we used a more balanced design, with replication only in the adder and multiplication units, while all other modules were not replicated. As we were targeting modularity and support for multiple algorithms, replication of more modules would require much larger bus sizes and significantly large area overheads.

5.4 Application and Performance as a Cryptographic Accelerator

We used the Avnet MiniZed board to verify the implementation on real a physical device. The targeted FPGA was a Xilinx XC7Z007S. It is one of the the smallest Xilnx FPGA with an embedded hard ARM core processor. The processor is a ARM Cortex A9 (ARMv7-A) running at 667 MHZ with the DDR3 RAM clocked at 533 MHz. The PL (programmable logic) clock was running at 125 MHz, mainly because the FPGA used on the board has the slowest *speed-grade* of -1. Table 4 shows the performance gains for some of the instructions. For most of the operations more than an order of magnitude gains are obtained. For example, the uniform noise sampling UNF is more than 90 times compared to the software implementation, similarly the NTT operation is about 23 times faster. As the NTT module supports externally provided *twiddle* factors and dynamic size options. It can be used to accelerate any other design which uses it. The value δ represents the extra time needed to copy the data to and from the memory to the processor; the value can vary widely depending on amount of data transferred, *bus-widths* used in the interconnect, and *burst-mode* support in the interface used and many other reasons. Having a fast, wide and low-latency connection to the CPU core is necessary if single instructions are required to be accelerated.

Table 4: Performance as a Hardware Accelerator (on the MiniZed Board). Measured on a *standalone* application using the hardware F_CLK/2 timer. The hardware accelerated execution time includes the two-way enable and done interrupt execution delays. For the software implementation we used the reference implementation.

Mnemonic	Instruction	Hardware (μs)	Software (μs)
UNF	*uniform*	14.93 + δ	1356.44
GNO	*getnoise*	23.27 + δ	784.56
ADP	*add polynomials*	4.33 + δ	108.76
PWS	*pointwise multiply*	4.54 + δ	209.87
NTT	*forward NTT*	49.7 + δ	1136.44
INTT	*inverse NTT*	49.7 + δ	1184.68
HLREC	*helprec*	9.64 + δ	258.42
REC	*rec*	8.52 + δ	229.47

6 Conclusions

In this work, we have presented a leakage resistant cryptoprocessor for `NewHope-USENIX`, `NewHope-NIST` and `HILA5`. To the best of our knowledge, this work presents the first side-channel protected post-quantum cryptoprocessor. This is also the fastest implementation of `NewHope-NIST`. Due to pipelining in most of the modules, this work achieves the highest clock frequencies among the results reported in literature. This leads to a highly compact, efficient and modular design. As this work follows the reference implementation of `NewHope-USENIX`, it is fully compatible with its software version.

References

[ADPS15] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - a new hope. *IACR Cryptology ePrint Archive*, 2015:1092, 2015.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Newhope without reconciliation. *IACR Cryptology ePrint Archive*, 2016:1157, 2016.

[APS13] Aydin Aysu, Cameron Patterson, and Patrick Schaumont. Low-cost and area-efficient FPGA implementations of lattice-based cryptography. In *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*, pages 81–86. IEEE, 2013.

[Bar86] Paul Barrett. Implementing the Rivest, Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 311–323. Springer, 1986.

[BBGM+17] Hayo Baan, Sauvik Bhattacharya, Oscar Garcia-Morchon, Ronald Rietman, Ludo Tolhuizen, Jose Luis Torre-Arce, and Zhenfei Zhang. Round2: Kem and pke based on glwr. *IACR Cryptology ePrint Archive*, 2017:1183, 2017.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis With a Leakage Model. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 16–29. Springer, 2004.

[BDH11] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS-a practical forward secure signature scheme based on minimal security assumptions. In *International Workshop on Post-Quantum Cryptography*, pages 117–129. Springer, 2011.

[BDL97] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997.

[BECN+06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[BGML+18] Sauvik Bhattacharya, Oscar Garcia-Morchon, Thijs Laarhoven, Ronald Rietman, Markku-Juhani O Saarinen, Ludo Tolhuizen, and Zhenfei Zhang. Round5: compact and fast post-quantum public-key encryption. *Submitted for publication*, 2018.

[BHH+15] Daniel J Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 368–397. Springer, 2015.

[BHO04] Rainer Buchty, Nevin Heintze, and Dino Oliva. Cryptonite–A programmable crypto processor architecture for high-bandwidth applications. In *International Conference on Architecture of Computing Systems*, pages 184–198. Springer, 2004.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Annual international cryptology conference*, pages 513–525. Springer, 1997.

[BVR+13] Ali Galip Bayrak, Nikola Velickovic, Francesco Regazzoni, David Novo, Philip Brisk, and Paolo Ienne. An eda-friendly protection scheme against side-channel attacks. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 410–415, 2013.

[BWM+16] Bela Bauer, Dave Wecker, Andrew J Millis, Matthew B Hastings, and Matthias Troyer. Hybrid quantum-classical approach to correlated materials. *Physical Review X*, 6(3):031045, 2016.

[Deu98] Deutsch. *The Fabric of Reality: The Science of Parallel Universes and Its Implications*. Penguin Books, 1998.

[EFGT16] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Loop-abort faults on lattice-based fiat-shamir and hash-and-sign signatures. In *International Conference on Selected Areas in Cryptography*, pages 140–158. Springer, 2016.

[Fey98] R. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6-7):467–488, 1998.

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 530–547. Springer, 2012.

[GS66] W Morven Gentleman and Gordon Sande. Fast Fourier Transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, pages 563–578. ACM, 1966.

[HMO+16] James Howe, Ciara Moore, Máire O'Neill, Francesco Regazzoni, Tim Güneysu, and Kevin Beeden. Lattice-based Encryption Over Standard Lattices in Hardware. In *Proceedings of the 53rd Annual Design Automation Conference*, page 162. ACM, 2016.

[ITT01] Kouichi Itoh, Masahiko Takenaka, and Naoya Torii. Dpa countermeasure based on the âĂIJmasking methodâĂİ. In *International Conference on Information Security and Cryptology*, pages 440–456. Springer, 2001.

[Jus17] Russ Juskalian. Practical quantum computers. MIT Technology Review, March/April 2017. https://www.technologyreview.com/s/603495/10-breakthrough-technologies-2017-practical-quantum-computers/.

[KAKJ17] Brian Koziel, Reza Azarderakhsh, Mehran Mozaffari Kermani, and David Jao. Post-quantum cryptography on FPGA based on isogenies on elliptic curves. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(1):86–99, 2017.

[KAMK16] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Fast Hardware Architectures for supersingular Isogeny Diffie-Hellman Key Exchange on FPGA. In *International Conference in Cryptology in India*, pages 191–206. Springer, 2016.

[KHL11] HeeSeok Kim, Seokhie Hong, and Jongin Lim. A fast and provably secure higher-order masking of aes s-box. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 95–107. Springer, 2011.

[KJJ+98] Paul Kocher, Joshua Jaffe, Benjamin Jun, et al. Introduction to differential power analysis and related attacks, 1998.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[KLC+17] Po-Chun Kuo, Wen-Ding Li, Yu-Wei Chen, Yuan-Che Hsu, Bo-Yuan Peng, Chen-Mou Cheng, and Bo-Yin Yang. High Performance Post-Quantum Key Exchange on FPGAs. *IACR Cryptology ePrint Archive*, 2017:690, 2017.

[Knu76] Donald Knuth. The complexity of nonuniform random number generation. *Algorithm and Complexity, New Directions and Results*, pages 357–428, 1976.

[Koç94] Ç Kaya Koç. Montgomery reduction with even modulus. *IEE Proceedings-Computers and Digital Techniques*, 141(5):314–316, 1994.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.

[Mce78] Robert J Mceliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, 44:114–116, 1978.

[MHER14] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4(3):145–156, 2014.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[Nie86] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Prob. Control and Inf. Theory*, 15(2), 1986.

[NIS] NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-final-dec-2016.pdf.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security*, pages 529–545. Springer, 2006.

[NS01] Phong Q Nguyen and Jacques Stern. The two faces of lattices in cryptology. In *Cryptography and lattices*, pages 146–180. Springer, 2001.

[Nus80] Henri Nussbaumer. Fast polynomial transform algorithms for digital convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(2):205–215, 1980.

[OG17] Tobias Oder and Tim Güneysu. Implementing the NewHope-Simple Key Exchange on Low-Cost FPGAs. *Progress in Cryptology–LATINCRYPT*, 2017, 2017.

[Pei14] Chris Peikert. Lattice cryptography for the internet. In *International Workshop on Post-Quantum Cryptography*. Springer, 2014.

[PG13] Thomas Pöppelmann and Tim Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In *International Conference on Selected Areas in Cryptography*, pages 68–85. Springer, 2013.

[Reg04] Oded Regev. Quantum computation and lattice problems. *SIAM Journal on Computing*, 33(3):738–760, 2004.

[RVM+13] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact Ring-LWE based cryptoprocessor. *Cryptographic Hardware and Embedded Systems–CHES 2014*, pages 1–1, 2013.

[Saa17] Markku-Juhani O Saarinen. Hila5: On reliability, reconciliation, and error correction for ring-lwe encryption. In *International Conference on Selected Areas in Cryptography*, pages 192–212. Springer, 2017.

[SKR+13] Khawar Shahzad, Ayesha Khalid, Zoltán Endre Rákossy, Goutam Paul, and Anupam Chattopadhyay. CoARX: a coprocessor for ARX-based cryptographic algorithms. In *Proceedings of the 50th Annual Design Automation Conference*, page 133. ACM, 2013.

[SL80] Richard M. Sedmak and Harris L. Liebergot. Fault tolerance of a general purpose computer implemented by very large scale integration. *IEEE Transactions on Computers*, (6):492–500, 1980.

[WT09] Knut Wold and Chik How Tan. Analysis and enhancement of random number generator in fpga based on oscillator rings. *International Journal of Reconfigurable Computing*, 2009:4, 2009.

Appendix A Assembly listing for the NewHope-USENIX Key Exchange

```
/*  NewHope RLWE Test
  ---------------------
    Memory Map [16 KB]
  ---------------------
[seed] [noiseseed]   [SK_A]
 0-31     32-63     128-2175

[SENA_A_POLY] [SEND_B_POLY]
    3072        5120-7167

[SEND_B_HREC] [GEN_KEY_B]
  7168-9215      64-95

[GEN_KEY_A]
  96-127
*/
// START of keygen

LDK Rk0, 0       // seed
LDK Rk1, 32      // noise

// gen_a(&a, seed);
UNF Rp0, Rk0

// poly_gno(sk, noise, 0);
GNO16 Rp3, Rk1, 0

// poly_ntt(sk);
NTT Rp3

// Store SK_A
STP Rp3, 128

// poly_gno(&e, noise, 1);
GNO16 Rp1, Rk1, 1

// poly_ntt(&e);
NTT Rp1

// poly_pws(&r, sk, &a);
PWS Rp2, Rp3, Rp0

// poly_add(&a, &e, &r);
ADP Rp0, Rp1, Rp2

// Store SEND_A_POLY
STP Rp0, 3072

// --- START of sharedb ---

LDK Rk0, 0
LDK Rk1, 64

// gen_a(&a, seed);
UNF Rp0, Rk0

// poly_gno(&sp, noise, 0);
GNO16 Rp1, Rk1, 0

// poly_ntt(&sp);
NTT Rp1

// poly_gno(&ep, noise, 1);
GNO16 Rp2, Rk1, 1

// poly_ntt(&ep);
NTT Rp2

// poly_pws(&a, &a, &sp);
PWS Rp3, Rp0, Rp1

// poly_add(&a, &a, &ep);
ADP Rp0, Rp3, Rp2

// Store SEND_B_POLY
STP Rp0, 5120

// Load SEND_A_POLY
LDP Rp2, 3072

// poly_pws(&ep, &ep, &sp);
PWS Rp3, Rp2, Rp1

// poly_bit_reverse(&ep),
BREV Rp3

// poly_invntt(&ep);
INTT Rp3

// poly_gno(&sp, noise, 2);
GNO16 Rp1, Rk1, 2

// poly_add(&ep, &ep, &sp);
ADP Rp2, Rp3, Rp1

// helprec(&sp, &ep, noise, 3);
HLREC Rp1, Rp2, Rk1, 3

// Store SEND_B_HREC
STP Rp1, 7168

// rec(sharedkey, &ep, &sp);
REC Rk2, Rp2, Rp1

// Store Shared Key B
STK Rk2, 64

// --- START of shareda ---

// Load SEND_B_POLY
LDP Rp0, 5120

// Load SK_A
LDP Rp1, 128

// poly_pws(&v, sk, &bp);
PWS Rp2, Rp1, Rp0

// poly_bit_reverse(&v)
BREV Rp2

// poly_invntt(&v);
INTT Rp2

// Load SEND_B_HREC
LDP Rp0, 7168

// rec(sharedkey, &v, &c);
REC Rk3, Rp2, Rp0

// Store Shared Key A
STK Rk3, 96
```

Figure 20: Assembly listing for the NewHope-USENIX Key Exchange (in-place)