A Configurable Crystals-Kyber Hardware Implementation with Side-Channel Protection

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

[1] NTU, Singapore
[2] IIIT-Delhi, India
[3] IIT, Jodhpur, India
{arpan,anupam}@ntu.edu.sg
{naina003}@e.ntu.edu.sg
{somitra}@iitj.ac.in

Abstract. In this work, we present a configurable and side channel resistant implementation of the post-quantum key-exchange algorithm `Crystals-Kyber`. The implemented design can be configured for different performance and area requirements leading to different trade-offs for different applications. A low area implementation can be achieved in 5269 LUTs and 2422 FFs, whereas a high performance implementation required 7151 LUTs and 3730 FFs. Due to a deeply pipelined architecture, a high operating speed of more than 250 MHz could be achieved on 28nm Xilinx FPGAs. The side channel resistance is implemented using a carefully chosen set of techniques resulting in a low overhead of less than 5%. To the best of our knowledge, this work presents the first side-channel attack protected configurable accelerator for `Crystals-Kyber`. Furthermore, one of the configuration choices results in the smallest hardware implementation of `Crystals-Kyber` known in literature.

Keywords: cryptography · post-quantum · key-exchange · cryptoprocessor · Kyber · fault-resistance · SCA

1 Introduction

The idea of building computers based on the principles of quantum mechanics for solving computational problems originated in the works of Deutsch and Feynman [12,14]. 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 [4]. This is already reflected through the demonstrations by Quantum computers, achieving insurmountable computational advantage over classical ones [2]. It has been conjectured that in the coming years, practical quantum computers will exist which will be able to break most of the currently used cryptographic systems [19].

In light of the above, there is a serious effort across the world to adopt new cryptographic primitives to resist an adversary, having access to large-scale quantum computer. These works are collectively driven by a standardization process for Post-Quantum Cryptography (PQC), by National Institute of Standards and Technology (NIST). NIST published a call for proposals in December 2016 [25]. 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 were shortlisted for second round. For the third round, 7 candidates were chosen as finalists and 8 candidates were considered as alternate.

PQC primitives originate from hard mathematical problems that cannot be solved by a Quantum computer in polynomial time. These hard problems span a diverse range of fields such as hash functions [10,5], coding theory [21,23], lattices [26,15], etc. Among these, lattice-based cryptography has emerged to be a promising candidate for PQC with majority of the finalists belonging to the class of structured-lattice problem. 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 [22,29] has led to the development of several lattice based cryptographic constructions over the years. Consequently, there is already a rich body of literature studying various implementations [1,32,11,17] and side-channel attacks [28,27,13,16,9] on lattice-based PQC schemes. However, considering various implementation alternatives, there needs to be a comprehensive analysis on design trade-offs, configurability and side-channel resistant PQC implementations. This is exactly what we address in this work. More specifically, we focus on implementing the module learning-with-errors (MLWE) algorithm `CRYSTALS-Kyber`[8], which is one of the finalists for NIST PQC public-key encryption. However, some of our design modules are generic and can be easily ported across other lattice-based PQC primitives.

1.1 Contributions

In this work, we present the design for a configurable Module-LWE based hardware implementation with fault and side channel leakage resistance. The contributions of this paper are as follows.

(i) We present detailed implementation results for `CRYSTALS-Kyber-1024` for both CPA and CCA secure variants. We also provide results for all the parameters of `CRYSTALS-Kyber` for CCA secure version.
(ii) We present and compare results for two implementations while utilizing two different SHA3 implementations one leading to a high-performance design and other one leading to a low-area design.
(iii) Using extensive resource sharing (arithmetic operations as well as control signals, FSMs) and a smaller SHA3 Core, we are able to achieve a reduction of about 70% and 50% in terms of LUTs and FFs than state-of-the-art hardware implementation. Thus, to the best of our knowledge, this paper presents smallest hardware implementation of `Crystals-Kyber` requiring only 5269 LUTs and 2422 FFs.
(iv) We have added multiple 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 4.1, new fast hashing based checksum construction with good differential characteristics; which can be used to detect maliciously injected faults with high probability.

(v) Fault countermeasures using inverted/complementary logic and state counters are also implemented and designs for secured FSM's for fetch and execute operations are provided in Section 4.1.
(vi) The implementation is highly configurable in terms of side-channel countermeasures. The specific countermeasures can be enabled depending on the application.

2 Preliminaries

In this section, we briefly discuss the post quantum key exchange protocol `CRYSTALS-Kyber`. For more detailed description, the interested reader is referred to [8].

2.1 Notations

Throughout the paper, we use bold lower-case letters to represent vectors with coefficients in time domain (e.g. $\mathbf{e}$) and lower-case italicized letters to denote polynomials (e). A hat over the top of a symbol is used to represent coefficients in the frequency domain ($\hat{\mathbf{e}}$). Whereas bold upper-case letters are used to represent Matrices (e.g. $\mathbf{A}$). $\mathscr{R}$ 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 with coefficients modulo q. Let $\mathscr{X}$ be a probability distribution over $\mathscr{R}$, then $x \leftarrow\$ \mathscr{X}$ means that x has been sampled according to $\mathscr{X}$. 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.

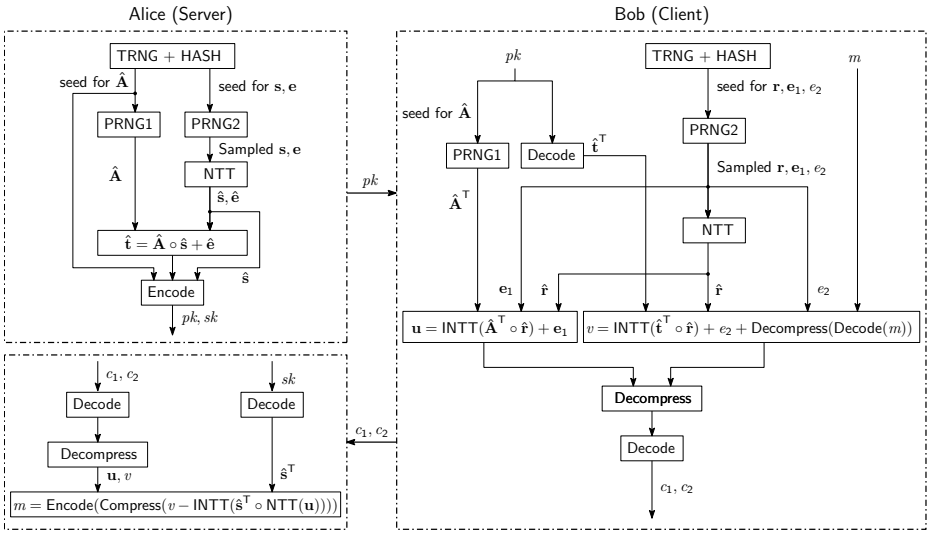


Fig. 1: `CRYSTALS-Kyber` Protocol Description

2.2 Protocol Description

The goal of a key-encapsulation mechanism (KEM) is to generate a shared secret key between two distrusting parties. The following three algorithms are used to carry out the actual KEM procedure:

1. *KeyGen()* : Using randomly generated seeds, Alice first creates a Matrix $\mathbf{A}$ sampled from a uniformly random distributed and two vectors ($\mathbf{s}$, $\mathbf{e}$) sampled from a centered binomial distribution. She then computes $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$. The seed used to sample $\mathbf{A}$ and the computed vector $\mathbf{t}$ encoded as public key (*pk*) is sent to Bob for use in Encapsulation. Further, vector $\mathbf{s}$ encoded as secret key (*sk*) is stored for Decapsulation process later. The vector $\mathbf{e}$ is a random masking noise used to hide the secret key.
2. *Encaps(pk)* : Bob samples vectors ($\mathbf{r}$, $\mathbf{e}_1$) and polynomial e_2 from the centered binomial distribution. Using these two vectors and the public key of Alice, Bob computes vector $\mathbf{u} = \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$. Bob also generates $v = (\mathbf{t}^\mathsf{T}\mathbf{r}) + e_2 + \mathsf{Decompress}(\mathsf{Decode}(m))$ using Alice's public key and the message m. Bob then computes the ciphertext c using $\mathbf{u}$ and v and sends this to Alice.
3. *Decaps(sk,c)* : Alice can now generate the message m using the ciphertext c and the stored secret key *sk*.

The overall description of the `CRYSTALS-Kyber` protocol is shown in Figure 1.

3 Architecture and Design Rationale

We implemented `CRYSTALS-Kyber` by breaking down the overall major steps (*keygen*, *encaps* and *decaps*) into smaller instructions. Similar to a general-purpose CPU our design also has fetch, decode and execute units as major components. But, certain features are quite different to accelerate a post quantum algorithm. One such major difference is that the instructions are quite large in terms of the area as well as the number of clock-cycles required. Some 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. We will discuss about the different components of the high level architecture shown in Fig. 2 in the following sections.

3.1 Fetch and Decode Units

The fetch unit reads the instructions from an external RAM based on the signals from the execute unit. Some of the instructions may 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 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.

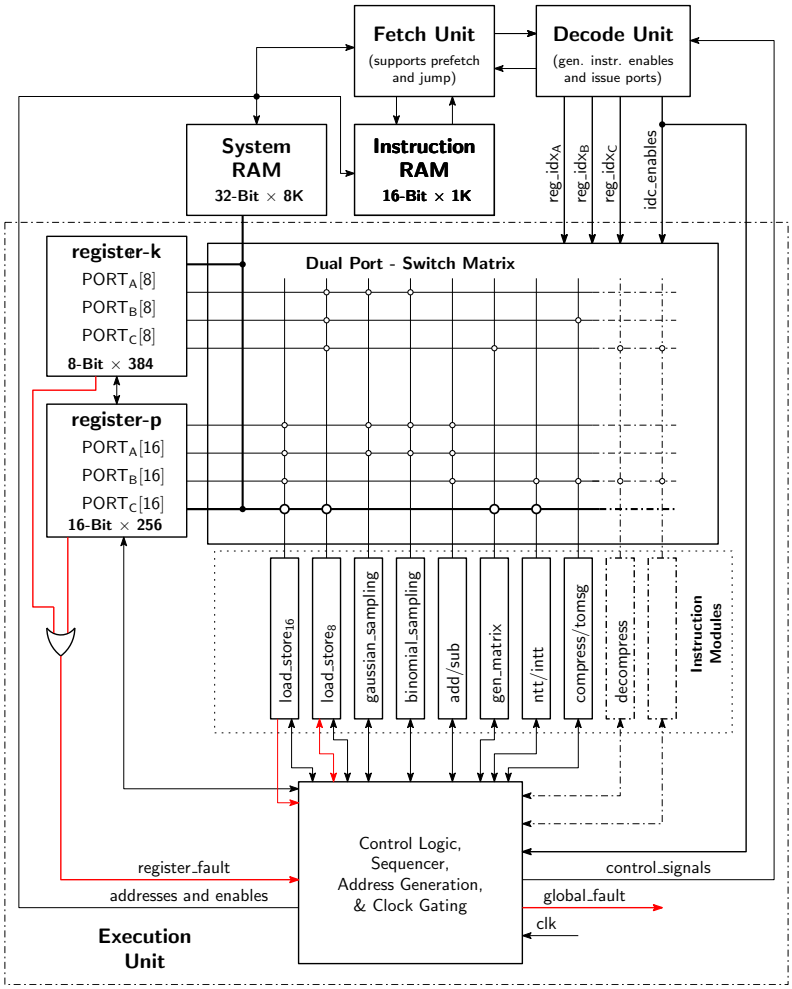


Fig. 2: Overall Architecture of the Proposed Design

3.2 Instruction and Data RAM

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. The instruction memory is a 2 KiB (8-bit $\times$ 1024) simple dual port RAM and the data memory is a 32 KiB (32-bit $\times$ 8192) dual-port memory.

3.3 Execution Unit

The execution unit is the largest and the most important part of the processor. The control/logic is instantiated within the execution unit. Also, all the modules for instructions are instantiated in the execution unit itself. The major building blocks for the execution unit and their functions are described below. The execution sequence is shown in A.

Register Modules Registers can be accessed by instructions for both read and write operations. As shown in Figure 3, 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 functionality is achieved by using a dual-port *bus-matrix* which allows any register to be selected and connected to any of the ports. One of the major challenges while designing this unit

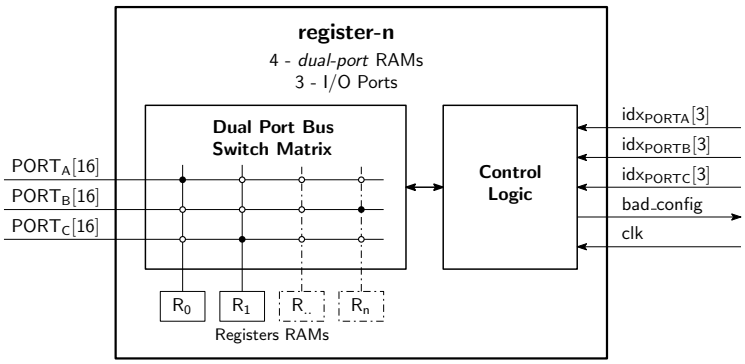


Fig. 3: Implementation for the different register modules.

was the large number of connections in the switch-matrix. As every module needs to be connected to the Load-Store units, the multiplexer becomes 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.

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 two load-store units, one each for the 16-bit poly and the 8-bit keys/encoded/compressed data.

We chose to create a 32-bit memory using two 16-bit dual-port memories with different write-enable lines. This allows us to read/write a maximum of 64-bits of data

per clock cycle. Using such a memory layout, 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. Further, the specific nature of Kyber NTT allows us to access two consecutive coefficients simultaneously, using a 32-bit dual-port memory allows for optimal utilization of the NTT pipeline.

3.4 NTT Module

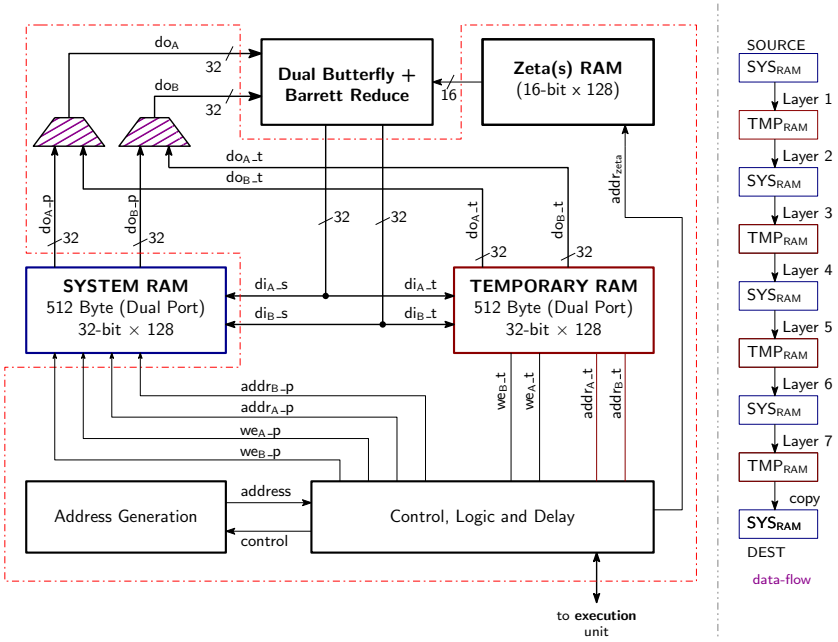


Fig. 4: Pipelined NTT Module.

Figure 4 shows the overall architecture of the Kyber NTT module. The NTT module can perform both forward and inverse transforms using the same hardware just by changing some control signals. The core part of the NTT module highlighted in the figure contains logic for control, address generation, a temporary RAM to store intermediate results and a ROM to store roots of unity (zetas). Given the NTT specification for `Kyber`, it is clear that two butterfly operations can be executed in parallel. So, by utilizing a 32-bit dual port memory we can access the inputs for two operations per clock cycle. In order to store the results we use a similar temporary RAM. This allows us to fully process a NTT layer (outer loop) containing 256 coefficients in 64 clock cycles. The layers can be executed continuously, and after the end of each layer we can switch direction, i.e. reading data from temporary RAM and storing the results in the global RAM. This memory architecture, along-with fully pipelined data-paths, allow us to

complete the NTT operation in (64+δ)*7+ε clock cycles. As we have seven layers, the final result remains in the temporary RAM, a further 64+ε cycles are utilized to write back this to the required location in system RAM. A final reduction is applied when the data is written back. The pipeline delay δ is 7 in this implementation. A dual butterfly is implemented with reduction circuits connected with multiplexers and working with a latency of 6 clock cycles. The address generation circuit generates addresses for all the butterfly operations, it also generates addresses for the Zeta(s) ROM.

```
k = 1;
for (len = 128; len >= 2; len >>= 1) { // layers [1-7]
       for (start = 0; start < 256; start = j + len) {
              zeta = zetas[k++];
              for (j = start; j < start + len; ++j) {
                     t = fqmul(zeta, r[j + len]);
                     r[j + len] = r[j] - t;
                     r[j] = r[j] + t;
              }
       }
}
```

Listing 1.1: NTT used in Kyber

3.5 Multiplications and Reductions

In a typical lattice-based based scheme, two form of reductions - Montgomery and Barrett are required. Both the reductions are very resource intensive units in a hardware design as they require three multiplication operations. Interestingly, a Barrett reduction can be performed using shift and addition operations as well[1]. Further, Montgomery reductions can be avoided altogether. Thus, in our design, we used a similar approach by Xing and Li [32] for Barrett Reduction. In order to minimize overall resource utilization

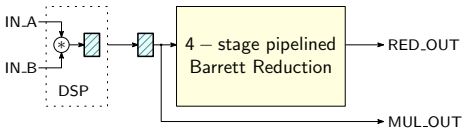


Fig. 5: Pipelined Multiply and Reduction Circuit

(especially DSPs), we implemented a single module (multiply and reduction) to provide three different modes - multiply two coefficients, multiply two coefficients and then perform barrett reduction or just perform a barrett reduction. All the three modes are used depending on the module. For e.g., multiplication in decompress can be achieved using mode 1, multiplication followed by a reduction in NTT butterfly can be achieved using mode 2, etc. Further, we added several pipeline stages to reduce the overall critical path. Figure 5 shows the overall pipelined multiply and reduction circuit.

3.6 Hash Functions

An implementation of Kyber requires several types of hash operations (for example SHA3-512, SHA3-256, SHAKE256, etc.), each of them is based around either SHA3 [7,6] or AES. We have implemented an outer wrapper which can perform all the required hash operations, while utilizing a common SHA3 core. In this implementation we use two versions of SHA3 core with very different area and performance characteristics. One is a high performance round based implementation while the other is a low area co-processor based implementation.

3.7 Compress and Polytomsg

From a hardware pespective, apart from some bit-select operations poly_compress, polyvec_compress and poly_tomsg modules require following two main operations:

1. Conditionally subtract q represented as a - q if a $>=$ q, else a
2. t = (((polynomial_coefficient $<<$ shift_val) + KYBER_Q / 2) / KYBER_Q) & shift_val

As mentioned, the two operations that require hardware resources are common in all the three modules. We combined the logic and FSMs for all the three modules into a single unit saving about 34% LUTs and 54.5% FFs compared to all the modules being implemented as a separate unit. Similar strategy is used wherever possible for other modules as well.

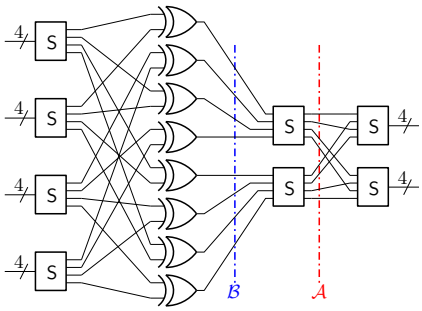
4 Fault and Side Channel Protection

Protecting the design 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.

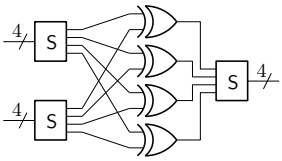
4.1 Fault Protection

Over the years, several countermeasures have been proposed to protect against fault attacks targeted towards a multitude of designs and surfaces [3,30]. The following section discusses about the proposed checksum construction and the different countermeasures implemented to protect different attack targets.

Fault Detection Hashes Error detection techniques 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 methods 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, the propagation of error to the checksum is not guaranteed. Hence, we designed some low-latency checksum functions. These functions are used in multiple



(a) 16×8 Fault Detection Hash



(b) 8×4 Fault Detection Hash

Fig. 6: **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.

modules to ensure that random single or multi-bit faults leads to error conditions with high probability.

Figure 6 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.

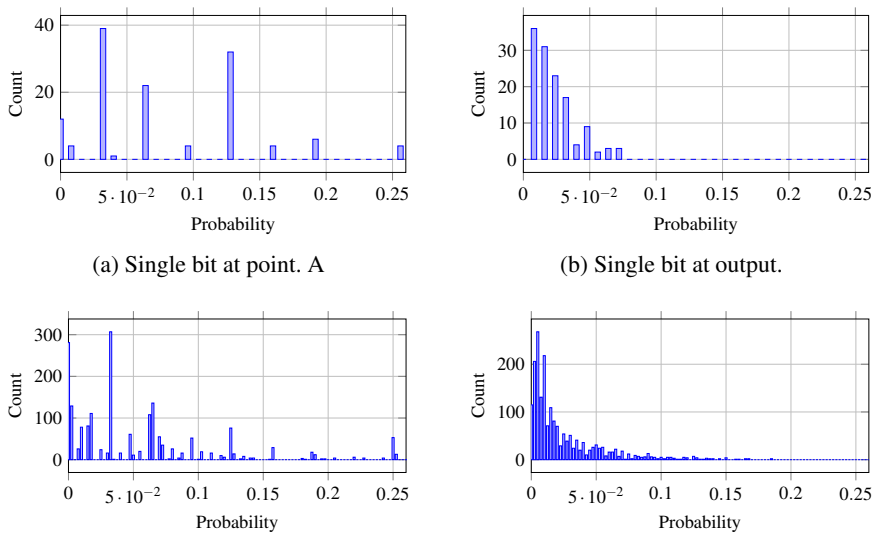
In order to identify good functions for fault detection, we designed and analyzed several possible candidates. Some performed well and some were highly unsuitable. One of the techniques for analyzing a function was to see how *differential bias* propagates through it. We considered only the single and double bit *fault-models* for the candidates. Figure 7 shows the results for the function shown in Figure 6a. 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 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 7a and 7c that the *histograms* corresponding to the point $\mathscr{A}$ 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 7b and 7d 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.



(a) Single bit at point. A

(b) Single bit at output.

(c) Double bits at point A.

(d) Double bits at the output.

Fig. 7: Differential Bias Histograms for Single and Double Bit Random Faults.

Protecting Critical Signals using Complementary 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 [3]. Such a countermeasure requires twice the area and memory resources. But, most 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 more 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).

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: $ptr_{n+1} <= \tilde{} (\tilde{} ptr_n + 1)$. 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 8 shows the instruction pointer update function.

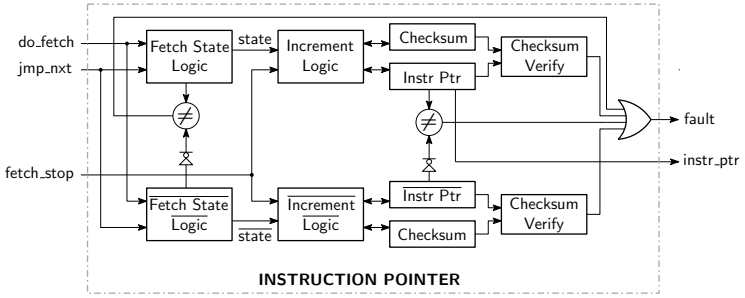


Fig. 8: **Instruction Pointer Update:** Fault Countermeasures

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 9 shows the corresponding implementation.

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 recovery of the entire state or secret keys.

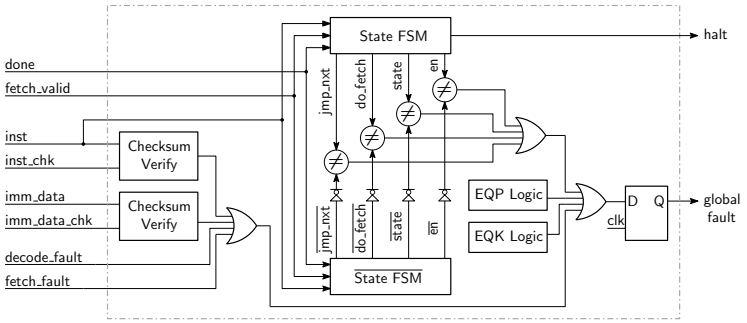


Fig. 9: **Execution State FSM:** Fault Countermeasures

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.

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 570 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.

Fault Tolerance: Equality Instructions The two instructions EQ_8 and EQ_16 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 1.2 shows a code snippet to use the equality instruction EQ_16 for the safe execution of getnoise operator to generate the centred binomial distribution.

```
LOAD8_32 Rk0, 0          // noise (from memory 0-31)
LOAD8_32 Rk1, 0          // noise (from memory 0-31)
```

```
GET_NOISE Rp0, Rk0, 0  // poly_gno(Rp0, noise, 0)
GET_NOISE Rp1, Rk1, 0  // poly_gno(Rp1, noise, 0)
EQ_16 Rp0, Rp1      // verify the two results
JP_M // jump to error when mismatch
. . .
```

Listing 1.2: Example assembly listing for verifying the output of getnoise. Errors in EQ_16 would reset the module.

Fault Tolerance: Direct Memory Access Isolation Although many of the instructions can be allowed to access the external memory, we chose to maintain isolation, even at the cost of reduction in performance. Only four instructions LOAD_x, STORE_x, NTT and GEN_MAT 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 simplifies both the instruction set design and the overall architecture.

4.2 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 [31]. The secure element is followed by a shift register. All the modules share this common source of randomness whenever required.

4.3 Side Channel Protection

Many countermeasures such as *masking*[18,20], *threshold implementation* [24] etc. have been developed over the years to protect against side-channel attacks. But, most of the secure countermeasures are expensive or difficult to apply on certain schemes. As a MLWE implementation contains several large modules there is a large attack surface for SPA/DPA, so adding specific countermeasures would be difficult and expensive. In this work, the following lightweight 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 techniques 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 `ADDx`, `SUBx`, `GET_NOISEx` 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 *fisher-yates-shuffle* technique. The permutation is updated during the random wait cycles at the end of every instruction.

Instruction Randomization: It is well known that processors can execute instructions in an *out-of-order* fashion. In fact, most most modern processors use this for performance optimization. In order to make SCA significantly much more difficult, we introduce and implement an instruction called `INSTRND`. This instruction takes a parameter N. Upon execution this instruction gives the processor a hint that the next N instructions can be executed in any order. The processor then randomly generates the order and executes accordingly. This causes the side channel traces to be out of alignment by a large margin, making differential analysis very difficult. This instruction was specifically implemented to allow us to improve security greatly without having to resort to expensive instruction buffers, queues and complex processing. Appendix B shows example usage for the instruction.

5 Implementation Results

This section presents performance evaluation and comparison with other implementations. We used Xilinx Vivado 2020.1 for synthesis, placement and routing. The results are presented for two FPGAs Xilinx Artix-7 XC7A35T-2 (28nm). The timing and utilization results are quite sensitive to the synthesis and implementation strategy as well as the timing constraints. We used default strategies for both synthesis and implementation.

5.1 Results

Table 1 shows implementation results for all the individual modules. These modules are executed by the execution unit depending on the instruction. As mentioned previously, we have combined multiple modules into one unit in order to optimize resource usage. For such modules, we have provided combined area (for e.g. Poly Basemul + Poly Reduce), but the clock cycles may differ depending on the operation. They are mentioned in a separate row in the table. Further, the logic for sampling is included in the wrapper module only. For instance, Generate Matrix module is responsible for invoking the shared SHA3 core and performing uniform sampling on the output to generate the matrix coefficients. Similarly, Get Noise module performs the centered binomial distribution on the output of SHA3 core to generate polynomial coefficients. Further, we have provided separate results for the two different versions of SHA3 core in the table. As can be seen

Table 1: Implementation results for individual modules.

Component	Clock	Frequency	Area Utilization			
	Cycles	XC7A35T	LUTs	FFs	BRAM	DSP
Processor Modules	(latency)	(MHz)			18K	
Fetch Unit	1	349	106	123	0	0
Decode Unit	1	467	44	45	0	0
Execute Unit (logic & control)	-	**258**				
- Load-Store (8-Bit)	16 - 192	394	59	54	0	0
- Load-Store (16-Bit)	128	336	37	19	0	0
- 8-bit regs	1	265	232	12	2	0
- 16-bit regs	1	258	360	12	2	0
- Generate Matrix	4623	270	246	133	0	0
- Get Noise	194	329	96	29	0	0
- Combined SHA3/SHAKE	98	333	74	29	0	0
- Poly Add + Sub	131	302	155	110	0	0
- Poly Basemul	773	298	248	197	0	0
+ Poly Reduce	137	-	-	-	-	-
- Polyvec compress	237	292	343	257	0	0
+ Poly Compress	141	-	-	-	-	-
+ Poly ToMsg	140	-	-	-	-	-
- Poly Decompress	132	327	232	93	0	0
+ Polyvec Decompress	197	-	-	-	-	-
- Poly FromMsg	129	313	89	80	0	0
+ Poly FromBytes	260	-	-	-	-	-
- Poly ToBytes	260	319	110	78	0	0
- multiply and reduction	6	260	262	334	0	2
- 16-bit Equality	-	306	65	95	0	0
- 8-bit Equality	-	363	33	48	0	0
- NTT/INTT	570	312	230	175	1	0
- SHA3 Co-processor Based	-	256	1497	665	1	0
- SHA3 Round Based	-	261	3380	1979	0	0

from the table, SHA3 requires a large majority of the resources even the low-area core compared to the other modules.

The 16-bit register unit, shown in Fig. 3 requires a lot of multiplexers to switch the four BRAM based memories among all the three ports dynamically. As a result, it has the longest critical path. Apart from this, the SHA3 and the reduce modules also have similar path delays. So, the overall design is limited to run at around 250 MHz.

Table 2 shows the comparison between clock cycles required for different operations (Generate Matrix, Get Noise, etc.) using the two variants of SHA3. It is interesting to note the significant difference in number of clock cycles between the two variants. Most

of the clock cycles are required for the SHA3 Permutation to finish its execution. In order to generate the initial random bytes (64 bytes) using SHA3-512, almost 54× more clock cycles will be required in case of a low-area implementation. Even though, clock cycles for a SHA3 modules is quite high, the overall time required to perform the key-exchange is in the order of a few milliseconds. But, as the implementation is quite low-area it can be useful for many low-end devices with limited resources.

Table 2: Clock-cycles comparison for the two variants of SHA3 used.

Processor Modules	Co-processor Based	Round Based
- Generate Matrix	252336	4623
- Get Noise	5399	194
- Combined SHA3/SHAKE	5304	98

Table 3: Clock-cycles comparison for the three variants of Kyber. The clock-cycles and time are reported for different variants as Keygen/Encaps/Decaps.

Kyber	**SHA3 Round Based**		**SHA3 Co-processor Based**	
Parameter	Clock Cycles ($\times 10^3$)	Time(μS)	Clock Cycles ($\times 10^3$)	Time(μS)
2 / Kyber-256	13.8/17.2/22.1	53.6/66.7/85.6	323/378/342	1294/1514/1369
3 / Kyber-512	26.6/30.9/37.8	103/119/146	666/733/688	2667/2932/2753
4 / Kyber-1024	43.8/48.8/57.2	170/189/224	1148/1236/1172	4593/4944/4691

5.2 Performance and Comparison

Table 3 presents the clock cycles and time required for different parameters of CRYSTALS-KYBER for both round based and co-processor based implementation. As can be seen from the table, for n = 4, our high performance implementation takes 170μs, 189μs and 221μs for *keygen*, *encaps* and *decaps* operations respectively. Whereas, our low area implementation requires 4.59ms, 4.94ms and 4.69ms respectively. Similarly, when n = 3, round based implementation requires 103μs, 119μs and 146μs for *keygen*, *encaps* and *decaps* operations. Whereas, co-processor based implementation requires 2.66ms, 2.93ms and 2.75ms respectively. Table 4 shows the performance comparison of our design with state-of-the-art implementations of CRYSTALS-KYBER. In terms of area, the high performance implementation requires, 7151 LUTs and 3730 FFs. Whereas by using a smaller SHA3 core the area reduces to 5269 LUTs and 2422 FFs, resulting in a reduction of 26.3% and 35% respectively.

Comparison with Compact Kyber Implementation (Xing & Li) [32] The authors of [32] present a compact implementation targeted towards low-cost FPGA's. The

Table 4: Performance and Comparison for Kyber-1024. SHA3-RB corresponds to implementation using Round based SHA3 Core, whereas SHA3-CB denotes Co-processor based SHA3 Core. The clock-cycles and time are reported for different operations as Keygen/Encaps/Decaps.

Kyber	**Time**			**Area**			
Variant	Freq. (MHz)	Cycles ($\times 10^3$)	Time (μS)	LUT	FF	RAM	DSP
This Work							
CPA-SHA3-RB	258	43.4/47.7/9.4	168/185/36.39	7151	3730	5.5	2
CCA-SHA3-RB	258	43.8/48.8/57.2	170/189/224	7151	3730	5.5	2
CPA-SHA3-CB	250	1081/1085/9.4	4324/4341/37.6	5269	2422	6.5	2
CCA-SHA3-CB	250	1148/1236/1172	4593/4944/4691	5269	2422	6.5	2
Related Works							
CCA (Xing & Li [32])	161	9.4/11.3/13.9	58.2/67.9/86.2	7412	4644	3	2
CCA (Dang et. al. [11])	210	-/5.7/7.4	-/27.4/35.2	12183	12441	15	8
CCA (Huang et. al. [17])	192	-/107.1/135.6	-/558/706	132918	172489	202	548
CCA (Alkim et. al. [1])	59	2203/2619/2429	37339/44390/41169	1842	1634	32	18

implementation is well optimized for both area and performance and has well-designed pipelines and instruction sequences. The authors implemented Kyber core operations in a highly interleaved manner, allowing them to execute multiple operations in parallel. This optimization strategy resulted in a low overall execution time of 67.9 μs for a client. The high-performance SHA3 takes only 26 clock cycles, thus, enabling the execution of other operations (such as NTT which requires 512 clock cycles) in parallel. This was utilzed in [32] to make the design compact. Using a low-area SHA3 would result in a similar performance as our low area implementation. This is because one invocation of a low-area SHA3 permutation requires about 5K clock cycles. Hence, in this case, the operations in the pipeline would have to wait longer as the hash latency is too long.

Compared to this, our design allows for a significantly more flexible execution of instructions, while arriving at the same result. This flexibility allows us to randomize the execution sequence in many places (as discussed in section 4.3) and hence achieving improved side-channel protection.

Comparison with Parallel Kyber Implementation (Dang et. al.) [11] The authors of [11] report multiple results on multiple platforms. For this comparison, we consider the results on similar FPGA platforms. The authors primarily focused on performance, by utilizing multiple invocations of core modules, leading to much improved performance at the cost of significantly increased area. Because of this, the implementation is better suited to high end devices. In comparison to our high-performance implementation, their design requires 5032 more LUTs whereas comparing with our low-area implementation, it requires 6914 more LUTs.

Comparison with Kyber Implementation (Huang et. al.) [17] The authors in this work present Kyber results for a pure hardware based implementation of Kyber. The authors used BRAMs to link multiple modules, thus requiring about 202 BRAMs. Their implementation requires a very large amount of area (18×) and clock cycles (2.19×) compared to our high-performance design.

Comparison with RISCV-based Kyber Implementation (Alkim et. al.) [1] The authors in [1] present instruction extensions for RISCV to implement Kyber. In their implementation, the SHA3 was software based and was the limiting factor for performance. As a result, if we compare the *keygen* operation, their design is 219× and 8.13× slower than our high-performance and low-area implementations respectively.

6 Conclusion

In this work, we presented a configurable hardware implementation of Crystals-Kyber. We present and compare results for two different implementations utilizing different versions of SHA3 core. To the best of our knowledge, we report the smallest hardware implementation of Kyber requiring only 5269 LUTs and 2422 FFs. Thus, making it suitable for multiple low-end devices. Compared to the currently known most compact hardware implementation, our smallest design resulted in a reduction of about 70% and 50% LUTs and FFs. Both our designs can run at an operating frequency of about 250 MHz. Further, we added multiple side-channel countermeasures consuming less than 5% of the total area.

References

1. Alkim, E., Evkan, H., Lahr, N., Niederhagen, R., Petri, R.: Isa extensions for finite field arithmetic. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 219–242 (2020)
2. Arute, F., Arya, K., Babbush, R., et al: Quantum supremacy using a programmable superconducting processor. Nature pp. 505–510 (2019)
3. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Proceedings of the IEEE **94**(2), 370–382 (2006)
4. Bauer, B., Wecker, D., Millis, A.J., Hastings, M.B., Troyer, M.: Hybrid quantum-classical approach to correlated materials. Physical Review X **6**(3), 031045 (2016)
5. Bernstein, D.J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., Wilcox-O'Hearn, Z.: SPHINCS: practical stateless hash-based signatures. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 368–397. Springer (2015)
6. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak hardware implementation, https://keccak.team/hardware.html
7. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 313–314. Springer (2013)
8. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-kyber: a cca-secure module-lattice-based kem. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 353–367. IEEE (2018)

9. Bruinderink, L.G., Pessl, P.: Differential fault attacks on deterministic lattice signatures. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 21–43 (2018)
10. Buchmann, J., Dahmen, E., Hülsing, A.: XMSS-a practical forward secure signature scheme based on minimal security assumptions. In: International Workshop on Post-Quantum Cryptography. pp. 117–129. Springer (2011)
11. Dang, V.B., Farahmand, F., Andrzejczak, M., Mohajerani, K., Nguyen, D.T., Gaj, K.: Implementation and benchmarking of round 2 candidates in the nist post-quantum cryptography standardization process using hardware and software/hardware co-design approaches. IACR Cryptol. ePrint Arch. **2020**, 795 (2020)
12. Deutsch: The Fabric of Reality: The Science of Parallel Universes and Its Implications. Penguin Books (1998)
13. Espitau, T., Fouque, P.A., Gerard, B., Tibouchi, M.: Loop-abort faults on lattice-based signature schemes and key exchange protocols. IEEE Transactions on Computers **67**(11), 1535–1549 (2018)
14. Feynman, R.: Simulating physics with computers. International Journal of Theoretical Physics **21**(6-7), 467–488 (1998)
15. Güneysu, T., Lyubashevsky, V., Pöppelmann, T.: Practical lattice-based cryptography: A signature scheme for embedded systems. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 530–547. Springer (2012)
16. Howe, J., Khalid, A., Martinoli, M., Regazzoni, F., Oswald, E.: Fault attack countermeasures for error samplers in lattice-based cryptography. In: 2019 IEEE International Symposium on Circuits and Systems (ISCAS). pp. 1–5. IEEE (2019)
17. Huang, Y., Huang, M., Lei, Z., Wu, J.: A pure hardware implementation of crystals-kyber pqc algorithm through resource reuse. IEICE Electronics Express pp. 17–20200234 (2020)
18. Itoh, K., Takenaka, M., Torii, N.: Dpa countermeasure based on the "masking method". In: International Conference on Information Security and Cryptology. pp. 440–456. Springer (2001)
19. Juskalian, R.: Practical quantum computers. MIT Technology Review (March/April 2017), https://www.technologyreview.com/s/603495/10-breakthrough-technologies-2017-practical-quantum-computers/
20. Kim, H., Hong, S., Lim, J.: A fast and provably secure higher-order masking of aes s-box. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 95–107. Springer (2011)
21. Mceliece, R.J.: A Public-Key Cryptosystem Based On Algebraic Coding Theory. Deep Space Network Progress Report **44**, 114–116 (1978)
22. Nguyen, P.Q., Stern, J.: The two faces of lattices in cryptology. In: Cryptography and lattices, pp. 146–180. Springer (2001)
23. Niederreiter, H.: Knapsack-type cryptosystems and algebraic coding theory. Prob. Control and Inf. Theory **15**(2) (1986)
24. Nikova, S., Rechberger, C., Rijmen, V.: Threshold implementations against side-channel attacks and glitches. In: Ning, P., Qing, S., Li, N. (eds.) Information and Communications Security. pp. 529–545. Springer (2006)
25. 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
26. Peikert, C.: Lattice cryptography for the internet. In: International Workshop on Post-Quantum Cryptography. Springer (2014)
27. Pessl, P., Prokop, L.: Fault attacks on cca-secure lattice kems. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 37–60 (2021)

28. Primas, R., Pessl, P., Mangard, S.: Single-trace side-channel attacks on masked lattice-based encryption. In: International Conference on Cryptographic Hardware and Embedded Systems. pp. 513–533. Springer (2017)
29. Regev, O.: Quantum computation and lattice problems. SIAM Journal on Computing **33**(3), 738–760 (2004)
30. Sedmak, R.M., Liebergot, H.L.: Fault tolerance of a general purpose computer implemented by very large scale integration. IEEE Transactions on Computers (6), 492–500 (1980)
31. Wold, K., Tan, C.H.: Analysis and enhancement of random number generator in fpga based on oscillator rings. International Journal of Reconfigurable Computing **2009**, 4 (2009)
32. Xing, Y., Li, S.: A compact hardware implementation of CCA-secure key exchange mechanism CRYSTALS-KYBER on FPGA. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 328–356 (2021)

Appendix A Execution Sequence

Figure 10 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

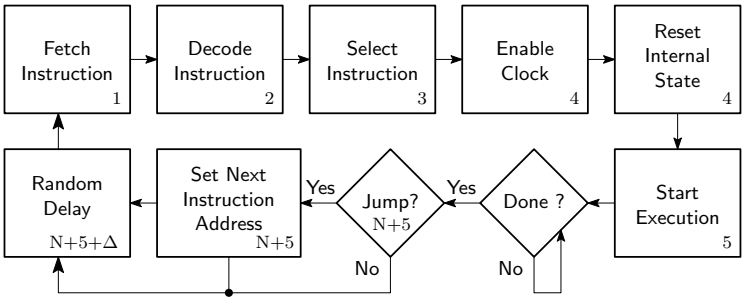


Fig. 10: Execution Sequence

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. Any error/fault will set the error flag which causes the execution to stop. If no error is encountered, the address is incremented. The execution unit then optionally waits for a random delay delta (Δ) time to prevent side-channel attacks. The new instruction is fetched only after the Δ clock cycles. This process repeats itself until a halt instruction (all zero) is encountered.

Appendix B Assembly listing for the `Crystals-Kyber` Keygen operation

The following code listing shows the instructions for `Kyber-1024` Keygen operation.

```
/*
< Memory 32 KiB >
< Organisation >
[   0 - 1024]-keys
[1034 - 3072]-skpv
[3072 - 5120]-pkpv
[5120 - 7168]-matrix
[8192 - 9760]-public keys
[10240-11776]-secret keys
*/

// START KYBER KEYGEN

LOAD8_32 Rk0, 0
SHA3 Rk1, Rk0
STORE8_64 Rk1, 128

INSTRND 2
GEN_MAT Rk1, 0
LOAD8_32 Rk2, 160

INSTRND 4
GET_NOISE Rp0, Rk2, 0
GET_NOISE Rp1, Rk2, 1
GET_NOISE Rp2, Rk2, 2
GET_NOISE Rp3, Rk2, 3

STORE16_256 Rp0, 1024
STORE16_256 Rp1, 1536

INSTRND 4
STORE16_256 Rp2, 2048
STORE16_256 Rp3, 2560
NTT 1024
NTT 1536

NTT 2048
NTT 2560

// ------------ 0
STORE16_1024 Rv0, 5120
LOAD16_256 Rp2, 1024
REDUCE Rp3, Rp2
STORE16_256 Rp3, 1024
LOAD16_256 Rp2, 5120
BASEMUL Rp0, Rp2, Rp3
LOAD16_256 Rp2, 1536
REDUCE Rp3, Rp2
STORE16_256 Rp3, 1536
LOAD16_256 Rp2, 5632
BASEMUL Rp1, Rp2, Rp3
ADD Rp3, Rp0, Rp1
LOAD16_256 Rp2, 2048
REDUCE Rp1, Rp2
STORE16_256 Rp1, 2048
LOAD16_256 Rp2, 6144
BASEMUL Rp0, Rp2, Rp1
ADD Rp1, Rp3, Rp0
LOAD16_256 Rp2, 2560
REDUCE Rp0, Rp2
STORE16_256 Rp0, 2560
LOAD16_256 Rp2, 6656
BASEMUL Rp3, Rp2, Rp0
ADD Rp2, Rp1, Rp3
REDUCE Rp0, Rp2
GET_NOISE Rp1, Rk2, 4
STORE16_256 Rp1, 7168
NTT 7168
LOAD16_256 Rp1, 7168
ADD Rp2, Rp0, Rp1
REDUCE Rp0, Rp2
STORE16_256 Rp0, 3072
// ------------ 1
STORE16_1024 Rv1, 5120
LOAD16_256 Rp3, 1024
LOAD16_256 Rp2, 5120
BASEMUL Rp0, Rp2, Rp3
LOAD16_256 Rp3, 1536
LOAD16_256 Rp2, 5632
BASEMUL Rp1, Rp2, Rp3
ADD Rp3, Rp0, Rp1
LOAD16_256 Rp1, 2048
LOAD16_256 Rp2, 6144
BASEMUL Rp0, Rp2, Rp1
ADD Rp1, Rp3, Rp0
LOAD16_256 Rp0, 2560
LOAD16_256 Rp2, 6656
BASEMUL Rp3, Rp2, Rp0
ADD Rp2, Rp1, Rp3
REDUCE Rp0, Rp2
GET_NOISE Rp1, Rk2, 5
STORE16_256 Rp1, 7168
NTT 7168
LOAD16_256 Rp1, 7168
ADD Rp2, Rp0, Rp1
REDUCE Rp0, Rp2
STORE16_256 Rp0, 3584
// ------------ 2
STORE16_1024 Rv2, 5120
LOAD16_256 Rp3, 1024
LOAD16_256 Rp2, 5120
BASEMUL Rp0, Rp2, Rp3
LOAD16_256 Rp3, 1536
LOAD16_256 Rp2, 5632
BASEMUL Rp1, Rp2, Rp3
ADD Rp3, Rp0, Rp1
LOAD16_256 Rp1, 2048
LOAD16_256 Rp2, 6144
BASEMUL Rp0, Rp2, Rp1
ADD Rp1, Rp3, Rp0
LOAD16_256 Rp0, 2560
LOAD16_256 Rp2, 6656
BASEMUL Rp3, Rp2, Rp0
ADD Rp2, Rp1, Rp3
REDUCE Rp0, Rp2
GET_NOISE Rp1, Rk2, 6
STORE16_256 Rp1, 7168
NTT 7168
LOAD16_256 Rp1, 7168
ADD Rp2, Rp0, Rp1
REDUCE Rp0, Rp2
STORE16_256 Rp0, 4096
// ------------ 3
STORE16_1024 Rv3, 5120
LOAD16_256 Rp3, 1024
LOAD16_256 Rp2, 5120
BASEMUL Rp0, Rp2, Rp3
LOAD16_256 Rp3, 1536
LOAD16_256 Rp2, 5632
BASEMUL Rp1, Rp2, Rp3
ADD Rp3, Rp0, Rp1
LOAD16_256 Rp1, 2048
```

```
LOAD16_256 Rp2, 6144
BASEMUL Rp0, Rp2, Rp1
ADD Rp1, Rp3, Rp0
LOAD16_256 Rp0, 2560
LOAD16_256 Rp2, 6656
BASEMUL Rp3, Rp2, Rp0
ADD Rp2, Rp1, Rp3
REDUCE Rp0, Rp2
GET_NOISE Rp1, Rk2, 7
STORE16_256 Rp1, 7168
NTT 7168
LOAD16_256 Rp1, 7168
ADD Rp2, Rp0, Rp1
REDUCE Rp3, Rp2
STORE16_256 Rp3, 4608
```

```
// PUBLIC KEY WRITEBACK
LOAD16_256 Rp0, 3072
LOAD16_256 Rp1, 3584
LOAD16_256 Rp2, 4096

TO_BYTES Rk3, Rp0
STORE8_384 Rk3, 8192
TO_BYTES Rk3, Rp1
STORE8_384 Rk3, 8576
TO_BYTES Rk3, Rp2
STORE8_384 Rk3, 8960
TO_BYTES Rk3, Rp3
STORE8_384 Rk3, 9344
STORE8_32 Rk1, 9728

// SECRET KEY WRITEBACK
```

```
LOAD16_256 Rp0, 1024
LOAD16_256 Rp1, 1536
LOAD16_256 Rp2, 2048
LOAD16_256 Rp3, 2560
TO_BYTES Rk3, Rp0
STORE8_384 Rk3, 10240
TO_BYTES Rk3, Rp1
STORE8_384 Rk3, 10624
TO_BYTES Rk3, Rp2
STORE8_384 Rk3, 11008
TO_BYTES Rk3, Rp3
STORE8_384 Rk3, 11392

HALT

// END-OF-KEYGEN
```