Accelerated RISC-V for SIKE

Rami Elkhatib
CEECS Department
Florida Atlantic University
Boca Raton, FL
relkhatib2015@fau.edu

Reza Azarderakhsh
CEECS Department
Florida Atlantic University
Boca Raton, FL
razarderakhsh@fau.edu

Mehran Mozaffari-Kermani
Dept. of CSE.
University of South Florida
Tampa, FL
mehran2@usf.edu

***Abstract*—Software implementations of cryptographic algorithms are slow but highly flexible and relatively easy to implement. On the other hand, hardware implementations are usually faster but provide little flexibility and require a lot of time to implement efficiently. In this paper, we develop a hybrid software-hardware implementation of the third round of Supersingular Isogeny Key Encapsulation (SIKE), a post-quantum cryptography algorithm candidate for NIST. We implement an isogeny field accelerator for the hardware and integrate it with a RISC-V processor which also acts as the main control unit for the field accelerator. The main advantage of this design is the high performance gain from the hardware implementation and the flexibility and fast development the software implementation provides. This is the first hybrid RISC-V and accelerator of SIKE. Furthermore, we provide one implementation for all NIST security levels of SIKE. Our design has the best area-time at NIST security levels 3 and 5 out of all hardware and hybrid designs provided in the literature.**

Keywords: isogeny-based cryptography, Montgomery multiplication, post-quantum cryptography, RISC-V, SIKE, software-hardware co-design.

I. Introduction

Public key cryptography, such as RSA and ECC, relies on hard mathematical problems to solve on classical computers. However, quantum computers have been discovered to be able to break such cryptosystems using Shor's algorithm [1]. To protect against quantum computers, NIST requested from experts in the industry to propose quantum safe algorithms through the Post-Quantum Cryptography (PQC) call for proposals [2]. The proposals have been heavily scrutinized through 3 rounds with the third round currently on going. There are currently 4 finalists and 5 alternative candidates for public-key encryption and key-establishment algorithms.

Supersingular Isogeny Key Encapsulation (SIKE) is one of the alternative candidates and the only candidate proposed that focuses on the difficulty of solving isogenies of elliptic curves. Traditional algorithms based on elliptic curves achieve pre-quantum security such as the implementations found in [3], [4], while supersingular elliptic curve arithmetic achieves post-quantum security. The main advantage of isogeny-based cryptography algorithms over other post-quantum cryptography algorithms is the relatively small key size. However, the smaller key size usually comes at the cost of higher computational time.

There are many pure hardware and pure software implementations of cryptography algorithms available in the literature. However, recently software-hardware co-design implementations have been proposed. In [5], Massolino *et al.* implement a software-hardware co-design of SIKE by using their custom-designed processor to control an isogeny-based hardware accelerator.

In the software side, processors that utilize the RISC-V architecture seem as an attractive choice because of the open standards. Furthermore, any improvements to SIKE can easily be ported to other RISC-based processors such as the ones found here [6], [7] for ARM Cortex M4. In [8], Banerjee *et al.* implement a software-hardware co-design for multiple Lattice-based cryptography algorithms. The authors utilize RISC-V processor to control their custom-designed Lattice-based crypto-accelerator. In [9], Banerjee *et al.* implement a software-hardware co-design of SIKE by using RISC-V processor to control an elliptic curve accelerator. Because the authors were aiming for area and utilized an elliptic curve accelerator instead of implementing an accelerator specifically targeting SIKE primes, the design is two orders of magnitude slower than state-of-the-art hardware-based SIKE implementations.

In this paper, we implement a RISC-V based software-hardware co-design of SIKE. On the hardware side, we implement an isogeny-based hardware accelerator which is controlled by a RISC-V processor. All security levels have been incorporated in one design and the design is aimed at maximizing area-time trade-off.

Our contributions:

- We improve area usage and operating frequency of Montgomery multiplication for SIKE prime.
- We implement a highly efficient software-hardware co-design of SIKE with all security levels supported in one design.
- This is the first implementation of SIKE in RISC-V with an isogeny-based accelerator aimed at maximizing area-time trade-off.
- We implement a dedicated instruction controller to efficiently process instructions in the accelerator.

The organization of the paper is as follows. In Section II, we give an overview of SIKE. In Section III, we propose our low

Table I. SIKE prime for each NIST security level [10]

Security Level	Prime Form
NIST level 1	$p_{434} = 2^{216}3^{137} - 1$
NIST level 2	$p_{503} = 2^{250}3^{159} - 1$
NIST level 3	$p_{610} = 2^{305}3^{192} - 1$
NIST level 5	$p_{751} = 2^{372}3^{239} - 1$

level efficient arithmetic operations. In Section IV, we propose our optimized architecture. In Section V, we implement our design on FPGA and present our results. In Section VI, we give our final thoughts and discuss future work.

II. Preliminaries

In this section, we provide a brief overview of SIKE protocol relevant to this paper. For a detailed overview of the SIKE protocol, we point the readers to [10]. SIKE supports a chosen-ciphertext attack (CCA) secure key encapsulation mechanism (KEM) and a chosen-plaintext attack (CPA) secure public key encryption (PKE) [10]. We will focus on KEM in this paper as it is the one implemented by most researchers in the literature and PKE can easily be implemented from KEM with some minor modification.

KEM allows two parties, usually named Alice and Bob for convenience, to securely exchange a shared secret. There are three main steps in KEM. In key generation, the secret key and public key are generated by Bob. Key generation is usually not important as it can be performed ahead of time. In key encapsulation, Alice retrieves Bob's public key and generates the cipher-text, which is sent to Bob, and the shared secret, which is stored locally. In key decapsulation, Bob uses his secret key and Alice's cipher-text to generate his shared secret, which is the same shared secret Alice generates.

There are two main independent operations in SIKE. The first operation is the hash-based operation. The only hashing operation used in SIKE is shake256, which is a NIST standardized hashing algorithm [11]. The main function in shake256 is the Keccak function [12] which is relatively cheap in comparison to the second operation.

The second operation, which is the main operation, is the isogeny-based operation. In SIKE, the isogeny operation works on elliptic curves defined over $\mathbb{F}_{p^2}$ [10]. The $\mathbb{F}_{p^2}$ can be further broken down to $\mathbb{F}_p$ operations. Here, p is a a constant prime number of special form $2^{e_A} \cdot 3^{e_B} - 1$. The size of the prime determines the security level of the protocol. Table I shows the prime used for different NIST security levels. In Section III, we are going to propose an efficient $\mathbb{F}_p$ architecture.

III. Proposed Efficient Low Level Arithmetic Operations.

In this section, we discuss low level arithmetic proposed for the isogeny accelerator. The isogeny accelerator requires two low-level units to perform all isogeny operations. These units are modular adder unit and the modular multiplier unit.

Table II. Steps for different operations performed by the modular addition unit

Operation	**Step 1**	**Step 2**	**Output**
mod add	$r_1 = a + b$	$r_2 = r_1 - 2p$	$a + b \mod 2p$
mod subtract	$r_1 = a - b$	$r_2 = r_1 + 2p$	$a - b \mod 2p$
mod correct	$r_1 = a + 0$	$r_2 = r_1 - p$	$a \mod p$

A. Modular Addition

The modular adder unit performs modular addition, modular subtraction and modular correction in two steps and each step utilizes the carry compact adder (CCA) proposed in [13], which is used for SIKE in [14], [15] and ECC over Curve25519 in [16]. The steps for each operation is shown in Table II. For modular addition and modular subtraction, the result is chosen from $r_1$and r_2 such that it is between 0 and $2p$ where p is the security level's prime. For modular correction, which is only performed at the end of the isogeny operation, the result is chosen from $r_1$and r_2 such that it is between 0 and p.

The unit is implemented for 752 bits which support the highest security level and all the lower security levels, and the design is modified to support a variable modulus as a variable modulus allows support for any security level. At the beginning of an isogeny operation, the modulus $2p$ is loaded once and all modular addition and modular subtraction operations can be performed. At the end of that isogeny operation, the modulus p is loaded and modular correction can be performed to correct the final result from $\mod (2p)$ to $\mod p$. Furthermore, since bits 1 to 215 (inclusive) are all 1s for any modulus chosen from any security level, the unit is designed with these bits of the modulus always set to 1. The main reason the two moduli p and $2p$ are loaded separately instead of deriving one from the other in hardware is to reduce the area for an operation that has minimal impact on performance. Another approach is to perform the correction from an external unit such as the RISC-V processor which will be used to control the isogeny accelerator.

In the architecture of the design, two fully pipelined adder/subtractor based on CCA are designed corresponding to each step of the unit. The optimal parameters for CCA can only be tested using trial and error when using an FPGA [13]. The optimal parameters H (for hierarchy level) and L (related to number of bits excluded from the hierarchy at each level) we obtained are $H = 1$ and $L = 5$ for the first CCA and $H = 1$ and $L = 15$ for the second CCA.

B. Modular Multiplication

In many cryptographic protocols such as RSA, DH, ECC and SIKE, modular multiplication is an expensive operation heavily studied by cryptographers. Many modular multipliers have been proposed for SIKE . Montgomery multiplication [17] is the most common method for modular multiplication used in SIKE. However, recently, new methods have been proposed such as the HFFM algorithm [18] which shows potential to be better than Montgomery multiplication. In this paper,

```
Algorithm 1: Proposed Montgomery Multiplication
  Input : $m = 2^{e_A} \cdot 3^{e_B} - 1 < 2^{K-2}$, $R = 2^K$, $w$, $s$,
          $K = w \cdot s$, $s_A = \lfloor 2^{e_A}/w \rfloor$, $a, b < 2m - 1$
  Output: MontMult$(a, b)$
1  $T \leftarrow 0$
2  for $i \leftarrow 0$ to $s - 1$ do
3      for $j \leftarrow 0$ to $s - 1$ do
4          $U[j] \leftarrow a[i] \cdot b[j]$                  } Mul
5      $(C, S) \leftarrow T[0] + U[0]$                       } First Acc
6      $q \leftarrow S$
7      for $j \leftarrow s_A$ to $s - 1$ do
8          $U[j] \leftarrow U[j] + q \cdot m[j]$             } Red
9      $U[s_A] \leftarrow U[s_A] + q$                         } First Red
10     for $j \leftarrow 0$ to $s - 1$ do
11         $(C, S) \leftarrow T[j] + U[j] + C$               } Acc
12         $T[j - 1] \leftarrow S$
13     $(C, S) \leftarrow C$                                 } Final Acc
14     $T[s - 1] \leftarrow S$
15 return $T$
```

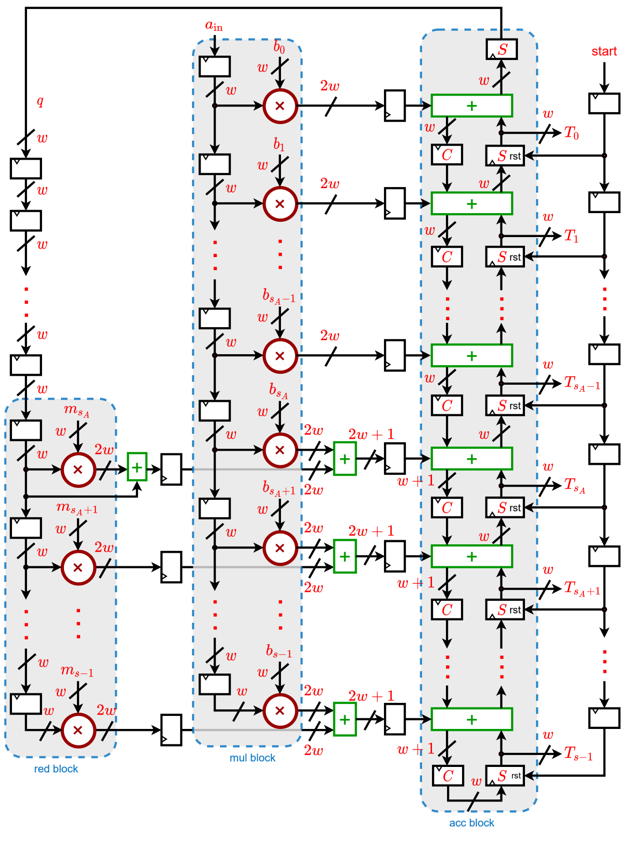
Figure 1. Architecture of multiplier core.

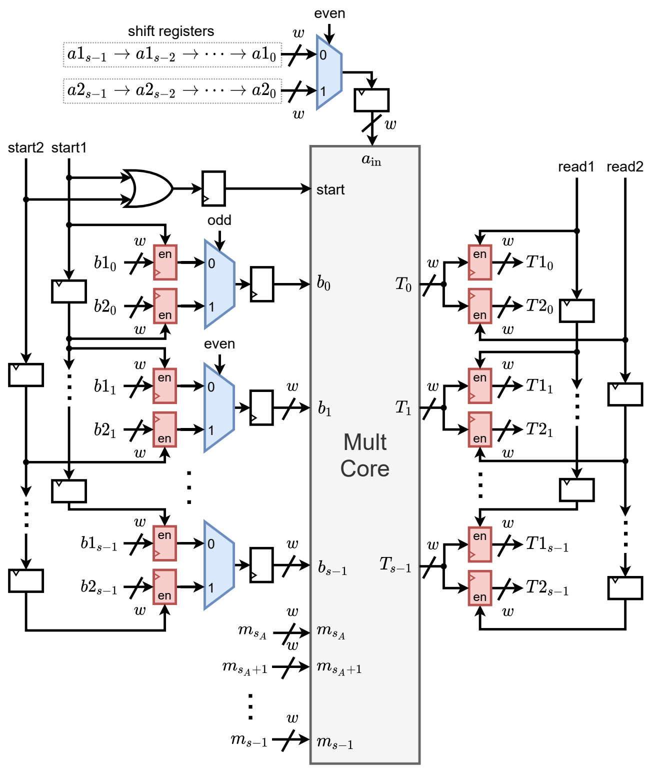
Figure 2. Dual multiplier IO for the multiplier core.

we improve the Montgomery multiplication proposed in [15] to consume less DSPs and operate at higher frequency while keeping the same number of interleave and multiplication cycles.

Algorithm 1 gives a breakdown of how the proposed architecture works while Fig. 1 shows the proposed architecture which is a systolic array architecture with each element working on w bits of the result. The architecture can be broken down into three major blocks; multiplication block, reduction block and accumulation block.

The multiplication block computes the product of the inputs a and b in s elements with b loaded in parallel and a loaded serially every two cycles. Each element in the multiplication block performs one iteration of Line 4 in Algorithm 1.

Similar to the multiplication block, the reduction block computes the product of a quotient q obtained from the accumulation block and the modulus m with m loaded in parallel and q loaded serially. Each element in the reduction block is added to its corresponding element in the multiplication block to perform one iteration of Line 8 in Algorithm 1. Since the first e_A bits of the modulus are 1, the first $s_A = \lfloor e_A/w \rfloor$ blocks are skipped and q is added to the first element of the reduction block as shown in Line 9 of Algorithm 1. Furthermore, q is delayed such that the first reduction element aligns with the s_A'th multiplication element when the addition is performed.

Finally, each element in the accumulation block performs a

Table III. DSP cost of Montgomery multiplier for $w = 17$

Reference	SIKEp434	SIKEp503	SIKEp610	SIKEp751
Elkhatib *et al.* [15]	65	75	90	113
This work	**40**	**46**	**55**	**69**

sum and propagates the least significant w bits S backwards and the remaining bits C forward along the systolic array. The sum is between S, C and the sum of the corresponding multiplication and reduction element. Since S is propagated backwards in the systolic array, it needs to be reset at the beginning of each multiplication. The first accumulation block, corresponding to Lines 5-6 of Algorithm 1, computes the quotient q for the reduction block and the first C for the accumulation block. The remaining elements perform Lines 11-12 of Algorithm 1 with the final C loaded to S of the same element similar to Lines 13-14. The first word of the final product is obtained from S of the second element after $2s$ cycles and the following words are obtained from the following elements in the following cycles until a total of $3s$ cycles are passed.

Unlike [15], the accumulation block is split from the multiplication block with each element of the multiplication and reduction blocks mapped to 1 DSP. Thus, the number of DSPs is reduced from $3s - s_A - 1$ to $2s - s_A$. Table III shows the number of DSPs reduced between our design and the design in [15] for $w = 17$. In addition, the adders not belonging to the accumulation block in Fig. 1 are mapped into the adder part of the DSP. Furthermore, the critical path is reduced to one DSP which improves the frequency from 294 MHz to 401 MHz for the device. The accumulation block is mapped into the fabric of the FPGA. To support all security levels, we set $s = 45$ and$s_A = 12$ with $w = 17$. Therefore, the architecture occupies a total of 78 DSPs.

If we breakdown the elements in the block to an even set and an odd set, the architecture operates on the partial result of only one set at any given cycle. To fully utilize all the elements, the unused elements at any given cycle can operate on a second pair of inputs effectively achieving two parallel multipliers. Fig. 2 shows how the inputs and outputs are connected to our Montgomery multiplier to support two parallel multipliers $T1$ =MontMult(a1,b1) and $T2$ =MontMult(a2,b2). Input a is loaded serially alternating between input a_1 and a_2 while input b is loaded in parallel with b_1 loaded to one set and b_2 loaded to the other set. To support interleaving, each word of input b is loaded as soon as the corresponding element requires it. The modulus m is loaded with the security level's prime prior to any multiplication. The outputs T_1 and T_2 are obtained from T of the multiplier as soon as they are available.

IV. Accelerated RISC-V

In this section, we discuss the RISC-V accelerator architecture for SIKE as shown in Fig. 3(a). The architecture is composed of a CPU which uses a 32-bit dedicated RAM for the software's instruction and data addresses. The CPU also allocates additional addresses for an APB interface by utilizing an APB bridge. The APB protocol is a royalty free protocol to connect low-bandwidth peripherals by ARM. In our setup, the CPU acts as the master and controls all peripherals which act as slaves. The peripherals used are GPIO and UART for IO operations and coprocessor for isogeny operations. The APB bridge uses an APB decoder to send the data to the correct peripheral. All the components excluding the coprocessor act as the main RISC-V chip and is based on Murax, which is a publicly available open source system on chip for VexRiscv CPU, an implementation of RISC-V CPU. The chosen CPU architecture implements rv32i instruction set which support basic integer operations excluding multiplication and division. The coprocessor, which is discussed in Section IV-A, is an isogeny accelerator with an APB-coprocessor bridge used to translate APB instructions to instructions the coprocessor can understand as shown in Fig. 3(b).

A. Coprocessor Architecture

At its core, the coprocessor is made up of an ALU unit and a RAM unit. The ALU, which is shown in Fig. 3(c), is made up of a modular adder as discussed in Section III-A and a dual Montgomery multiplier acting as two parallel Montgomery multipliers as discussed in section III-B. The two multipliers and the adder each have unique pair of operands. One operand of all three pairs reads data from one port of the RAM unit while the other operand reads data from the other port of the RAM unit. The adder output from the adder unit is written to one port of the RAM unit. The multiplier output is selected from one of the multipliers and written to the other port of the RAM unit. Since each multiplier starts at a different cycle, they never need to write data into the RAM in the same cycle.

The adder unit is continuously processing its operands with one bit used to select modular addition or modular subtraction. If no addition operation is required, the result is discarded and not written to the RAM unit. For the multiplication unit, three bits are used for each multiplier to load data from the RAM unit, start the multiplier, and start reading the result of the multiplier into the output register, respectively. Finally, one bit is used to select the output register to be written into the RAM unit.

The RAM unit is a 256×752 true dual-port RAM used to store all isogeny constants and computations. The RAM unit uses 9 bits to control each port with 8 bits to select the address and 1 bit to enable writing. To minimize the critical path, the data is available to be read from this RAM unit 2 cycles after the address is set.

An instruction controller, discussed in Section IV-B, is used to control the ALU and RAM units when the accelerator is performing the instructions received from the CPU. The instruction controller receives 26-bit instructions from an instruction memory which in turn receives it from the CPU. The instruction is composed of three 8-bit addresses, one for each operand's address and one for the destination address, and 2 bits for addition, subtraction, multiplication, or end opcode. The instruction memory is a 64×26 simple dual port RAM

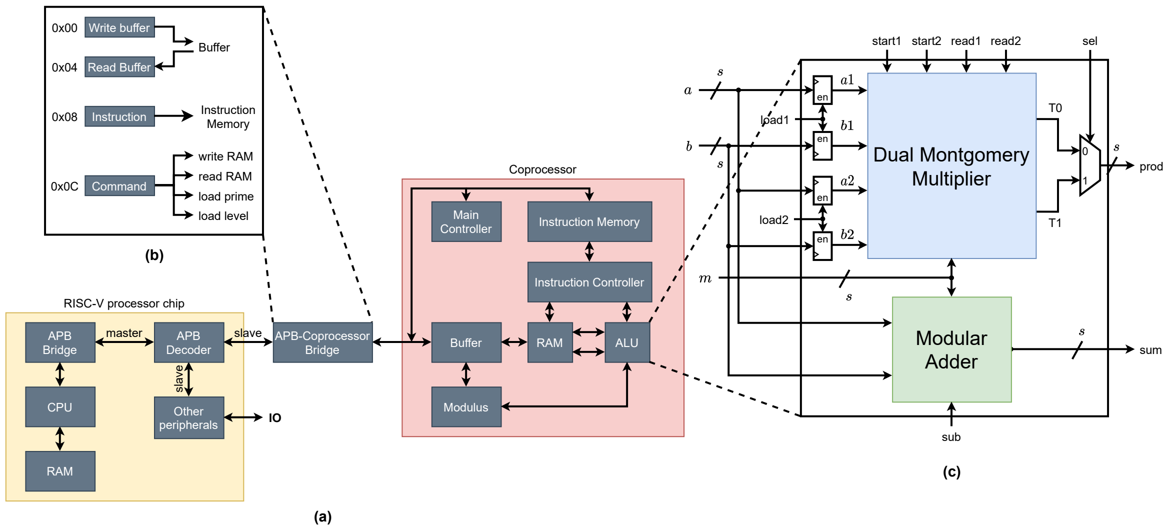

Figure 3. (a) RISC-V accelerator architecture. (b) Coprocessor-APB bridge. (c) ALU architecture.

used to buffer instructions to avoid any bottlenecks caused by the difference between the rate at which instructions are received from the CPU and the rate at which instructions are processed by the ALU unit. The instruction memory operates as a first-in first-out (FIFO) unit with a main controller controlling writing instructions into the memory while the instruction controller independently controlling reading instructions from the memory. The addresses in the instruction memory are divided in two 32-address sets. After one set is completely filled or an end instruction is received, the main controller instructs the instruction controller to begin processing the instructions in the set. When one set is completely filled while the other set is still being processed, the main controller instructs the CPU to halt until the set being processed is completed.

A 768-bit buffer is used to transfer data between the CPU and the coprocessor's RAM unit, as well as, to load a modulus into the modulus register. To transfer data from the CPU to the coprocessor, 32-bits are shifted into the buffer per transaction until all 768 bits are transferred with most significant bits set to 0 for unused bits. Once all bits are transferred, a command is issued to the main controller to load the data into a specific address of the RAM unit. To transfer data from the coprocessor back to the CPU, the reverse process is performed. A command is first issued into the main controller to load data from a specific address of the RAM unit into the buffer. Then, 32-bits are shifted from the buffer into the CPU per transaction until all required bits are transferred. In addition, the data in the buffer can be loaded into the modulus register by issuing a load modulus command to the main controller after the modulus is loaded into the buffer.

B. Instruction Controller

As stated in IV-A, the instruction controller (IC) is responsible for reading the instructions loaded into the instruction memory and controlling the RAM and ALU to perform the instruction. The IC operates in 2 fetch stages before decoding the instructions. On reset, the IC is in an inactive state. When a start signal from the main controller of the coprocessor is received, it goes into fetch 1 state where only fetch 1 is operating. In fetch 1 operation, the program counter, used to get the address in instruction memory, is incremented to fetch the next instruction while the current instruction in the instruction memory is being loaded into the first register. The IC then goes into fetch 2 state where both fetch 1 and fetch 2 are operating. In fetch 2 operation, the instruction from the first register is transferred into the second register. After fetch 2 state, the IC goes into an active state where, in addition to the previous two operations, the decode operation is running. In the decode operation, the instruction in the second register can begin to be processed by the IC.

The first step in the decode operation is for the IC to send the operand addresses to the RAM unit, the data of which is available after 2 cycles. In the case of an add or subtract opcode, the subtract bit is sent to the adder unit, and, 2 cycles later, the adder write port of the RAM unit is enabled to write the result. In case of a multiplication opcode, the IC loads the data into a register for the current multiplier which alternate after every multiplication instruction at the decode stage. Then, the IC instructs the multiplier to start the instruction with one multiplier waiting for an odd cycle and the other waiting for an even cycle. The IC waits using a stage 1 counter for the interleave cycles of the multiplier less 6 which account to 2 ram cycles, 1 load cycle, 1 start cycle, and 2 additional cycles. Once the cycles are completed, the first stage of the multiplier is done. A read signal is sent to the multiplier 6 cycles later to begin collecting the partial results. The IC then goes into a stage 2 counter which waits until one cycle before the multiplier finishes collecting all the partial results. After 1 cycle, the IC sends the select signal to the multiplier and enables the multiplier write port of the RAM unit. Stage 1 and stage 2 counters are variable counters with the number of cycles transmitted by the CPU before any computation is performed to support a variable number of digits.

The destination address of the instruction goes through a similar process to the opcode process before being transmitted into the RAM unit during a write cycle. First, the destination address gets loaded into new registers for 2 cycles corresponding to the RAM stages. The last register, in turn, gets loaded into two new registers over another 2 cycles corresponding to the adder stages, the last of which is transmitted to the write port's address of the RAM unit. In case of a multiplication, the destination address obtained at the end of the second RAM stage is loaded into one of two registers depending on which multiplier is loaded and this corresponds to the first multiplier stage. After the stage 1 counter of a multiplier is done, the destination address is transferred to a second register corresponding to the second multiplier stage. Finally, when the multiplier is ready to write the results back to the RAM unit, the destination address is selected from one of the two second multiplier stage registers. The set of all destination addresses excluding the destination address at the second add stage are additionally checked if they are active. We are going to call this set, henceforth, the running destination address set.

There are three cases that will cause the IC unit to temporarily halt the fetching and decoding operations but not operations that passed the decode stage. The first case is during a write operation since the RAM unit cannot perform read and write in the same cycle. In the write operation case, the IC stops all fetch and decode operations for one cycle.

The second case is during a memory lock state where one of the operands of an instruction is the output of a previous instruction that hasn't completed yet. When an instruction is in the decode stage, the operands (or source) addresses are checked against active addresses in the running destination address set. If either of the source addresses is found in the list, then the IC goes into a memory lock state. To unlock the memory lock state, the source address that caused the locked state needs to match the destination address written back to the RAM.

The third case is during multiplication lock state where both multipliers are unavailable which happens when both multipliers are processing data in the interleave stage. When an instruction at the decode stage is a multiplication instruction and it is ready to go through, the multiplier usage counter is incremented indicating a multiplier is locked. On the other hand, when the first stage counter for any multiplier is done, the multiplier usage counter is decremented indicating a multiplier is released. If the multiplier usage counter is greater than or equal to 2, which indicates both multipliers are unavailable, then any additional multiplication instructions at the decode stage will cause a multiplication lock state. Completing first stage counters for either multipliers unlocks the multiplication lock state.

During the active state of the IC unit, if an end instruction is passed at the decode stage, the IC unit will go into the inactive state if the other set of instructions in the instruction memory is not ready yet. Otherwise, the IC unit will go into fetch 1 state to begin the other set of instructions. If fetch 1 operation reaches the end of the set, the IC unit will go into two temporary states final 1 and final 2 to check if an end instruction is encountered in the last 2 instructions that the decode operation hasn't processed yet.

Table IV. Breakdown of Isogeny operations cost for SIKEp434

Function	#CC	Status
3 Pt Ladder	824	Feasible
Pt Quad	930	Optimized
Get 4-Iso	247	Optimized
Eval 4-Iso	619	Optimized
Pt Triple	927	Feasible
Get 3-Iso	375	Optimized
Eval 3-Iso	513	Optimized
Fp Inv	42,849	Feasible

C. Software

The software code is implemented based on Microsoft's software implementation of SIKE with some modification. First, all major isogeny functions, also found in Table IV, are broken into $\mathbb{F}_p$ instructions which can then be replaced with the coprocessor's equivalent instructions. A scheduler based on [19] is used to minimize the cycle count and ran for 1 hour per function. Table IV shows the cost of the isogeny functions for the lowest security level with a status indicating whether an optimal cycle count was found.

The keccak function, which is used for hashing, is implemented in assembly using [20]. It takes 23k cycles to complete one permutation which increases the cycle count by around 8% for the lowest security level and less for higher security levels. We consider this increase an acceptable increase in comparison to implementing the keccak function in hardware.

The software is compiled using RISC-V GNU compiler toolchain with rv32i architecture and level 3 optimization chosen. Each security level is compiled separately. The size of the program code including data storage is less than 48KB for all security levels. Therefore, the RAM unit of the RISC-V processor chip of our architecture is also set to 48 KB. The code of one security level is then loaded into the RAM unit and the SIKE code can then run inside the chip.

V. FPGA Implementation

In this section, we discuss FPGA implementation of the design described in section IV. The architecture is written in Verilog and SystemVerilog with the RISC-V processor chip generated in Verilog using SpinalHDL. The architecture is implemented in Xilinx Virtex-7 xc7vx690tffg1157-3 similar to the one used in most other SIKE implementations found in the literature. All results obtained are post-place and route.

Table V shows a detailed area result of the design. The total area of the chip is 4,611 slices (15,246 flip-flops and 11,212 LUTs), 78 DSPs and 34.5 BRAMs. The RISC-V processor

Table V. Detailed area result of RISC-V accelerator chip in Virtex-7 FPGA

Component	# FFs	# LUTs	# Slices	# DSPs	# BRAMs
RISC-V processor chip	1322	1321	555	0	13
Coprocessor	13922	9888	4087	78	21.5
- adder	2274	3082	1374	0	0
- mult	9378	2026	2730	78	0
Total	15,246	11,212	4,611	78	34.5

chip occupies 12% of the slice area and 38% of the BRAM area while the coprocessor occupies 88% of the slice area and 62% of the BRAM area. Most of the area of the coprocessor comes from the multiplier which occupies 66% of the slice area and uses all the DSPs required for the design. The remaining area of the coprocessor comes from the adder and occupies around 33% of the slice area.

Table VI shows the timing results of the design. The operating frequency of the design is 243.6 MHz with the adder being the bottleneck of the design. For security level 1, the key generation (keygen) takes 1.22 million clock cycles, the key encapsulation (keyencap) takes 2.26 million clock cycles, and the key decapsulation (keydecap) takes 2.41 million clock cycles. Since the keygen is usually pre-generated, it is not counted towards the total latency of the SIKE protocol. The total time, which is the clock cycles of the keyencap and keydecap divided by the frequency, is 19.2 ms. The total time is 25.1 ms for security level 2, 38.7 ms for security level 3, and 55.0 ms for security level 5.

Table VII compares our results with the results of other SIKE hardware and mixed software-hardware designs. We used 1 DSP = 1 Bram = 100 Slices for area-time trade-off as was used in [14]. Our design is more than $2\times$ slower than the fast hardware implementations used in [19], [15]. However, we use $3\times$ less multipliers and we support all security levels in one design. This means that at the highest security level, we use around $4\times$ less area in comparison to the state-of-the-art [15]. Our area-time trade-off is better than all other designs available in the literature at security level 3 and 5 with security level 5 almost having $2\times$ improvement in area-time trade-off. For security level 1 and 2, our design has a slightly worse area-time trade-off than state-of-the-art.

The authors in [5] implemented two software-hardware designs for SIKE with a custom-made CPU. Their fast implementation is worse than our designs in all parameters with our design having effectively $2\times$ better area-time trade-off. Their slow implementation occupies less area in comparison to our implementation. However, their performance is significantly lower and it has as significantly worse area-time trade-off.

Table VI. Timing results of RISC-V accelerator chip in Virtex-7 FPGA

Security	Freq.	#CC ($\times 10^6$)				Total time
Level	[MHz]	K	E	D	E+D	[ms]
1	243.6	1.22	2.26	2.41	4.68	19.2
2		1.64	2.95	3.17	6.11	25.1
3		2.39	4.66	4.77	9.43	38.7
5		3.72	6.46	6.95	13.41	55.0

Table VII. Comparison of area and timing results in Virtex-7 FPGA

Reference	Slices	DSPs	BRAMs	Time	AT ($\times 10^{-3}$)
SIKEp434					
Koziel *et al.* [14]	8,121	240	26.5	11.3	393
Elkhatib *et al.* [15]	5,527	195	32.0	8.8	248
Massolino *et al.* [5] (S)	3,415	57	21.0	50.4	565
Massolino *et al.* [5] (F)	7,408	162	38.0	24.3	666
This work	**4,611**	**78**	**34.5**	**19.2**	**305**
SIKEp503					
Koziel *et al.* [21]*	10,298	192	27.0	33.7	1,085
Koziel *et al.* [22]*	8,918	192	40.0	20.9	671
Koziel *et al.* [23]*	7,491	192	43.5	16.5	512
Koziel *et al.* [14]	8,907	264	33.5	14.1	545
Elkhatib *et al.* [15]	6,163	225	34.0	11.8	378
Massolino *et al.* [5] (S)	3,415	57	21.0	59.5	667
Massolino *et al.* [5] (F)	7,408	162	38.0	28.7	787
This work	**4,611**	**78**	**34.5**	**25.1**	**398**
SIKEp610					
Koziel *et al.* [14]	10,675	312	39.5	21.6	990
Elkhatib *et al.* [15]	7,461	270	38.5	19.1	732
Massolino *et al.* [5] (S)	3,415	57	21.0	107.2	1,202
Massolino *et al.* [5] (F)	7,408	162	38.0	51.8	1,420
This work	**4,611**	**78**	**34.5**	**38.7**	**614**
SIKEp751					
SIKE Team [24]	16,756	376	56.5	33.4	2,004
Koziel *et al.* [14]	15,834	512	43.5	27.8	1,984
Farzam *et al.* [19]**	15,336	512	45.0	24.1	1,712
Elkhatib *et al.* [15]	11,136	452	41.5	25.5	1,542
Massolino *et al.* [5] (S)	3,415	57	21.0	179.6	2,014
Massolino *et al.* [5] (F)	7,408	162	38.0	60.8	1,666
This work	**4,611**	**78**	**34.5**	**55.0**	**872**

$*$ SIDH

$**$ SIKE Round 1 Parameters

VI. Conclusion

In this paper, we implemented a software-hardware co-design for SIKE targeting all security levels in on design. We also improved the Montgomery multiplier architecture proposed in [15]. We presented FPGA implementations of our design and showed that we have a highly efficient area-time trade-off. Our design shows potential for software-hardware co-design to be competitive with pure hardware designs. Our future work will involve exploring a RISC-V coprocessor design with separate security levels and with more multipliers to have a more fair comparison with hardware-only designs.

VII. Acknowledgment

The authors would like to thank the reviewers for their comments. The authors would also like to thank Utsav Banerjee for his constructive comments. This work is supported in parts by DoD-N001741910031.

References

[1] P. W. Shor, "Algorithms for Quantum Computation: Discrete Logarithms and Factoring," in *35th Annual Symposium on Foundations of Computer Science (FOCS 1994)*, pp. 124–134, 1994.

[2] The National Institute of Standards and Technology (NIST), "Post-quantum cryptography standardization," 2017–2018. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization.

[3] M. Bisheh Niasar, R. Azarderakhsh, and M. M. Kermani, "Efficient hardware implementations for elliptic curve cryptography over curve448," in *Progress in Cryptology – INDOCRYPT 2020* (K. Bhargavan, E. Oswald, and M. Prabhakaran, eds.), (Cham), pp. 228–247, Springer International Publishing, 2020.

[4] M. Bisheh-Niasar, R. Azarderakhsh, and M. Mozaffari-Kermani, "Area-time efficient hardware architecture for signature based on ed448," *IEEE Transactions on Circuits and Systems II: Express Briefs*, pp. 1–1, 2021.

[5] P. M. C. Massolino, P. Longa, J. Renes, and L. Batina, "A compact and scalable hardware/software co-design of sike," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 245–271, 2020.

[6] H. Seo, M. Anastasova, A. Jalali, and R. Azarderakhsh, "Supersingular isogeny key encapsulation (sike)round 2 on arm cortex-m4," *IEEE Transactions on Computers*, pp. 1–1, 2020.

[7] M. Anastasova, R. Azarderakhsh, and M. M. Kermani, "Fast strategies for the implementation of sike round 3 on arm cortex-m4." Cryptology ePrint Archive, Report 2021/115, 2021. https://eprint.iacr.org/2021/115.

[8] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, pp. 17–61, Aug. 2019.

[9] U. Banerjee, S. Das, and A. P. Chandrakasan, "Accelerating post-quantum cryptography using an energy-efficient tls crypto-processor," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2020.

[10] R. Azarderakhsh, M. Campagna, C. Costello, L. De Feo, B. Hess, A. Jalali, D. Jao, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, G. Pereira, J. Renes, V. Soukharev, and D. Urbanik, "Supersingular Isogeny Key Encapsulation." Submission to the NIST Post-Quantum Standardization Project, 2020.

[11] M. Dworkin, "Sha-3 standard: Permutation-based hash and extendable-output functions," 2015-08-04 2015.

[12] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer, "Keccak Implementation Overview," May 2012.

[13] T. B. Preuxer, M. Zabel, and R. G. Spallek, "Accelerating Computations on FPGA Carry Chains by Operand Compaction," in *2011 IEEE 20th Symposium on Computer Arithmetic*, pp. 95–102, July 2011.

[14] B. Koziel, A. Ackie, R. El Khatib, R. Azarderakhsh, and M. M. Kermani, "Sike'd up: Fast hardware architectures for supersingular isogeny key encapsulation," *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–13, 2020.

[15] R. Elkhatib, R. Azarderakhsh, and M. Mozaffari-Kermani, "Highly optimized montgomery multiplier for sike primes on fpga," in *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, pp. 64–71, 2020.

[16] M. B. Niasar, R. El Khatib, R. Azarderakhsh, and M. Mozaffari-Kermani, "Fast, small, and area-time efficient architectures for key-exchange on curve25519," in *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, pp. 72–79, 2020.

[17] P. L. Montgomery, "Modular Multiplication without Trial Division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[18] W. Liu, Z. Ni, J. Ni, C. Rafferty, and M. O'Neill, "High performance modular multiplication for sidh," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 3118–3122, 2020.

[19] M.-H. Farzam, S. Bayat-Sarmadi, and H. Mosanaei-Boorani, "Implementation of supersingular isogeny-based diffie-hellman and key encapsulationusing an efficient scheduling," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2020.

[20] K. Stoffelen, "Efficient cryptography on the risc-v architecture." Cryptology ePrint Archive, Report 2019/794, 2019. https://eprint.iacr.org/2019/794.

[21] B. Koziel, R. Azarderakhsh, M. Mozaffari-Kermani, and D. Jao, "Post-Quantum Cryptography on FPGA Based on Isogenies on Elliptic Curves," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, pp. 86–99, Jan 2017.

[22] B. Koziel, R. Azarderakhsh, and M. Mozaffari-Kermani, "Fast Hardware Architectures for Supersingular Isogeny Diffie-Hellman Key Exchange on FPGA," in *Progress in Cryptology – INDOCRYPT 2016: 17th International Conference on Cryptology in India*, pp. 191–206, 2016.

[23] B. Koziel, R. Azarderakhsh, and M. Mozaffari-Kermani, "A High-Performance and Scalable Hardware Architecture for Isogeny-Based Cryptography," *IEEE Transactions on Computers*, vol. 67, pp. 1594–1609, Nov 2018.

[24] R. Azarderakhsh, M. Campagna, C. Costello, L. De Feo, B. Hess, A. Jalali, D. Jao, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, G. Pereira, J. Renes, V. Soukharev, and D. Urbanik, "Supersingular Isogeny Key Encapsulation." Submission to the NIST Post-Quantum Standardization Project, 2019.