Implementation of the Rainbow signature scheme on SoC FPGA

Tomáš Přeučil, Petr Socha, Martin Novotný
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9, 160 00 Praha, Czech Republic
{tomas.preucil | petr.socha | martin.novotny}@fit.cvut.cz

***Abstract*—Thanks to the research progress, quantum computers are slowly becoming a reality and some companies already have their working prototypes. While this is great news for some, it also means that some of the encryption algorithms used today will be rendered unsafe and obsolete. Due to this fact, NIST (US National Institute of Standards and Technology) has been running a standardization process for quantum-resistant key exchange algorithms and digital signatures. One of these is Rainbow—a signature scheme based on the fact that solving a set of random multivariate quadratic system is an NP-hard problem.**

This work aims to develop an AXI-connected accelerator for the Rainbow signature scheme, specifically the Ia variant. The accelerator is highly parameterizable, allowing to choose the data bus width, directly affecting the FPGA area used. It is also possible to swap components to use the design for other variants of Rainbow. This allows for a comprehensive experimental evaluation of our design.

The developed accelerator provides significant speedup compared to CPU-based computation. This paper includes detailed documentation of the design as well as performance and resource utilisation evaluation.

***Index Terms*—post-quantum cryptography, multivariate quadratic, hardware design, FPGA, system on a chip**

I. Introduction

The number of devices connected to the internet has been increasing rapidly in the past years, and so is the need for a secure communication. At this moment, most of these devices use symmetric encryption algorithms and/or asymmetric public key algorithms such as RSA (security of which relies on the difficulty of factoring the product of two large prime numbers) or Diffie-Hellman key exchange (where the security of the contemporary version relies on the discrete logarithm problem). These public-key cryptosystems are considered secure against the cryptanalysis that we can perform today.

However, quantum computers may be a threat for today's cryptography. Quantum computing uses quantum bits—qubits—for encoding information. These bits can exist in more than one state at the same time. Unlike classical computers, the power of quantum computer increases exponentially with each added qubit [1]. It is only a matter of time until a sufficiently large quantum computer is built.

This research has been supported by the grant VJ02010010 of the Ministry of the Interior of the Czech Republic, Tools for AI-enhanced Security Verification of Cryptographic Devices in the programme Impakt1 (2022-2025).

Example of such a threat is Shor's algorithm [2], a quantum computer algorithm for an efficient factoring of numbers. The algorithm allows a quantum computer to factor numbers a lot faster than if we would use conventional methods; in fact, the time complexity is just $O((logN)^3)$ where N is the number of bits. Compared to the classical computer, the complexity is, even for the General number sieve which is the best known algorithm for factoring large numbers, exponential [3] [4]. For factoring a number, Shor's algorithm needs approximately $O(3N)$ qubits where N is the number of bits of the factored number [5]. Therefore, when a sufficiently large quantum computer is manufactured, it will render most of the RSA-based schemes, as well as schemes relying on the discrete logarithm problem, insecure.

In 2020, IBM released a quantum computer with 65 qubits, and they plan to make computers with 1121 qubits available in 2023 [6]. Given that NIST recommends at least RSA-2048 (2048-bit key) and the memory complexity of Shor's algorithm, it is still not enough to crack today's encryption.

The Rainbow scheme is one of the third-round candidates in the NIST Post-Quantum Cryptography Standardization Process. The process aims to standardize a set of key encapsulation mechanisms and signature algorithms secure in the presence of a quantum computer. Anyone can comment on the submissions, and NIST evaluates both security and performance [7].

Rainbow [8] is a generalisation of the Unbalanced Oil and Vinegar (UOV) [9] construction. The UOV scheme intends to provide systems, that only have a small amount of RAM and processing power, with a public key authentification.

In this work, we implement the Rainbow scheme on an FPGA, with the result being an Intellectual Property (IP) block connected over the Advanced eXtensible Interface (AXI) and serving as an accelerator in hardware/software codesign in a system on a chip. Furthermore, we provide an experimental evaluation of the performance and the area requirements.

II. Background

A. Rainbow

The Rainbow [10] scheme is based on the Unbalanced Oil and Vinegar system, which means that it relies on the fact that solving m quadratic equations for n variables gets very

difficult [11]. To solve these, the Rainbow scheme (and the UOV scheme) uses a special structure of polynomials.

Solving systems of general quadratic equations is believed to be NP-hard [12]. However, thanks to their special structure, we are able to solve it while it is not feasible for the attacker. The signature is easy to verify (because verifying a solution of a system of equations can be done in constant time) [9].

In Rainbow, secret linear maps are used to hide the special structure which allows the signee to solve the system easily. The quadratic equations consist of n unknowns called oils and v unknowns called vinegars over a finite field **K** [9].

Rainbow takes the principle of UOV and applies it in layers, i.e., subsets of equations that are solved independently at a time, over a finite field **F**. Then, the Rainbow parameters are integers $0 < v_1 < ... < v_u < v_{u+1} = n$ where u is the layer number. A layer consists of two sets (indexed by i) of variables over a finite field:

$$V_i = \{1, ..., v_i\}(i = 1, ..., u) \tag{1}$$

which we call vinegars and

$$O_i = \{v_i + 1, ..., v_{i+1}\}(i = 1, ..., u) \tag{2}$$

which are called oils. The vinegar variables are supplied for each layer and we solve the equations for the oil variables in the respective layer. Therefore, we have a set of $m = n - v_1$ (number of variables minus the number of vinegars in the first layer—the vinegars for the first layer are populated by random data so we get an exact solution and not a subspace) equations.

The Rainbow private key therefore consists of three maps: The two linear maps and a central quadratic map. The central map F which consists of m polynomials (equations) the form of which is presented by Equation 3.

$$f^k(x_1, ..., x_n) := \sum_{i,j \in V_l} \alpha_{i,j}^{(k)} x_i x_j + \sum_{\substack{i \in V_l \\ j \in O_l}} \beta_{i,j}^{(k)} x_i x_j \tag{3}$$

In Equation 3, α and β are quadratic coefficients from the sets of vinegars and oils, respectively. The linear and constant coefficients are omitted, as they are not used in the proposed signature scheme nor in our implementation.

The two linear invertible maps T and S are used to hide the structure of F. The public key P is the composition of these three maps. All these maps are constructed over a finite field [13].

To get a signature $x = P^{-1}(y)$ we first need to apply the inverse of the affine map S, i.e. compute $z = S^{-1}(y)$, then solve the equation system $w = F^{-1}(z)$ and then apply the map T (compute $x = T^{-1}(w)$).

After applying the inverse of the affine map S, the signature is obtained by first generating random data which are used to populate the v_1 vinegar unknowns. Then, we populate these vinegars into part of the central map and check whether it is invertible. If it is not, the whole signing process is repeated with new random vinegars. Next, we compute the remaining variables in all the layers using Gaussian elimination. Thanks to the special structure of F (no $oil \times oil$ terms in the polynomials), it is guaranteed that we will get a set of linear equations. Lastly, we apply the second affine map T.

When we get only the public key P, it seems that it is a random quadratic system since the structure is hidden by the affine maps S and T [8].

Rainbow signature is much smaller than the previously described Oil-Vinegar variants. The signature is only slightly longer than the digest of the document [8]. These schemes are efficient when it comes to RAM and required processing power but quite inefficient when the public key size is considered. However, private key computations may be performed without the public key (which is larger) [9].

B. Parameters and security levels

In the NIST competition, the Rainbow offers three different sets of parameters for different levels of security [10]. The parameters, as well as the minimum equivalent security of a block cipher, are shown in Table I. The parameters are: the size of the Galois Field, the number of vinegars for the first layer, the number of oils for the first layer and the number of oils for the second layer. These parameters not only affect security but also the key size [10]. Besides these parameters, there is also the number of layers which is always two. If we would use only one layer, we get UOV.

TABLE I
PARAMETERS AND MINIMUM SECURITY EQUIVALENTS FOR DIFFERENT VARIANTS OF RAINBOW [10]

Level	Parameters ($\mathbf{F}, v_1, o_1, o_2$)	Equiv. to (bits)
I	$\mathbf{GF}(16), 36, 32, 32$	128
III	$\mathbf{GF}(256), 68, 36, 36$	192
V	$\mathbf{GF}(256), 92, 48, 48$	256

The Rainbow scheme implements three different security levels. Table II shows the key and signature sizes for different security levels.

TABLE II
SECURITY LEVELS AND KEY SIZES OF RAINBOW [14]

Level	Public key size (KiB)	Private key size (KiB)	Sig. size (bits)
I	157.8	101.2	528
III	861.4	611.3	1312
V	1885.4	1375.7	1632

There are two variants of Rainbow—classic and cyclic. This work focuses on the classic variant, namely the Ia classic (first line of Table I). However, the presented design can be altered for any classic variant if the target device is large enough.

C. Algorithm description

This section contains an analysis of the reference implementation. Anything related to random number generation and hashing is omitted as this work focuses solely on the Rainbow algorithm (therefore lines 2, and 9 in algorithm 1 are omitted). The input of the algorithm is a hashed and salted document (digest).

First (lines $1-6$ in algorithm 1), the vinegar variables of the first layer are filled by random data and are populated into the central map. This is done by using the same function that performs matrix-vector multiplication. Then, the part of the central map that belongs to the first layer, with regards to these vinegars, is inverted. If the inverse is found, the algorithm proceeds to the next stage.

After that (line 10 in algorithm 1), the remaining operations for the first layer are performed. These consist of a multiplication of the input digest by the S matrix and solving the equations for the first layer.

Next (line 11 in algorithm 1), the variables needed for the second layer are computed. In this phase, the function for matrix-vector multiplication is reused, and a function for evaluating triangular matrices is also needed.

Next (lines $12-13$ in algorithm 1), a set of 32 equations (with 32 unknowns) is evaluated. If the evaluation is successful, the algorithm proceeds to the last stage.

In the last stage (lines $15-17$ in algorithm 1), operations over the T map are performed and a signature is returned.

Input: document d, Rainbow private key $(InvS, c_S, \mathcal{F}, InvT, c_T)$, length ℓ of the salt
Output: signature $\sigma = (\mathbf{z}, r) \in \mathbb{F}^n \times \{0,1\}^\ell$ such that $\mathcal{P}(\mathbf{z}) = \mathcal{H}(\mathcal{H}(d)||r)$
1: **repeat**
2: $\quad y_1, \ldots, y_{v_1} \leftarrow_R \mathbb{F}$
3: $\quad \hat{f}^{(v_1+1)}, \ldots, \hat{f}^{(n)} \leftarrow f^{(v_1+1)}(y_1, \ldots, y_{v_1}), \ldots, f^{(n)}(y_1, \ldots, y_{v_1})$
4: $\quad (\hat{F}, c_F) \leftarrow \texttt{Aff}^{-1}(\hat{f}^{(v_1+1)}, \ldots, \hat{f}^{(n)})$
5: **until** IsInvertible$(\hat{F}) ==$ **TRUE**
6: $InvF = \hat{F}^{-1}$
7: **repeat**
8: $\quad r \leftarrow \{0,1\}^\ell$
9: $\quad \mathbf{h} \leftarrow \mathcal{H}(\mathcal{H}(d)||r)$
10: $\quad \mathbf{x} \leftarrow InvS \cdot (\mathbf{h} - c_S)$
11: $\quad (y_{v_1+1}, \ldots, y_{v_2}) \leftarrow InvF \cdot ((x_{v_1+1}, \ldots, x_{v_2}) - c_F)$
12: $\quad \hat{f}^{(v_2+1)}, \ldots, \hat{f}^{(n)} \leftarrow \hat{f}^{(v_2+1)}(y_{v_1+1}, \ldots, y_{v_2}), \ldots, \hat{f}^{(n)}(y_{v_1+1}, \ldots, y_{v_2})$
13: $\quad t, (y_{v_2+1}, \ldots, y_n) \leftarrow \text{Gauss}(\hat{f}^{(v_2+1)} = x_{v_2+1}, \ldots, \hat{f}^{(n)} = x_n)$
14: **until** $t ==$ **TRUE**
15: $\mathbf{z} = InvT \cdot (\mathbf{y} - c_T)$
16: $\sigma \leftarrow (\mathbf{z}, r)$
17: **return** σ

Algorithm 1: Rainbow sign algorithm [13]

D. Related research

There are several published papers that focus on hardware implementations of Rainbow. The first three presented implementations use different parameters to the version of Rainbow that was submitted to the NIST standardization process. The first paper [15] was published in 2008, and its authors designed an architecture that can sign a digest in 804 clock cycles at 105 MHz. It was followed by [16] in 2011, where the signature process takes one fourth of the clock cycles but only at 50 MHz. In these cases, the digest and the signatures are only 24 and 42 bytes long, respectively. Lastly, there is [17] from 2018 where the authors present a way to generate the signature in 242 clock cycles at 50 MHz with the digest and signature being 26 and 43 bytes, respectively.

The next paper [18] was published in 2018, and when run with the same parameters as [15] and [16], it produces a signature in only 148 clock cycles at 200 MHz. Furthermore, it can work over different finite fields as well as with different sets of parameters and, for the Ia classic variant, it produces a signature in 1980 clock cycles at 181 MHz.

In February 2022, Ward Beullens published a pre-print that presents a possibility of breaking the Ia classic variant of Rainbow signature scheme in 53 hours on average using an off the shelf laptop [19]. Therefore, the current parameters of Rainbow may not be sufficient anymore due to the new attack vectors.

III. Design

Based on the algorithm needs, we decided to design modules implementing following operations:

- Multiplying a vector by a scalar
- Merging the vinegars of layer one into the central map
- Matrix-vector multiplication
- Evaluating the triangular matrices
- Addition
- Memory copying
- Memory resetting
- Linear equation solver
- Main controller

These modules were developed in VHDL.

A. Block diagram

In this subsection, we describe the modules implementing the previously mentioned operations in more detail.

a) Scalar-vector multiplication: This module takes the following signals as input: a 4096-bit accumulator (A), a 4096-bit chunk of multiplied matrix (M) and a 4-bit $\mathbf{GF}(16)$ element (k). The operation itself is $A = A + M \times k$. Therefore, we multiply a vector of up to 1024 $\mathbf{GF}(16)$ elements by a single $\mathbf{GF}(16)$ scalar.

The multiplication itself is implemented as a lookup table. There is the correct number of individual multipliers to multiply all the $\mathbf{GF}(16)$ elements in one BUS_WIDTH long word within one clock cycle.

b) Matrix-vector multiplication: This module is called *gfmat_prod* in the implementation and handles both the matrix-vector multiplication as well as the merging of the vinegars into the central map. The input is an address of the matrix being multiplied and an address of a multiplicand vector. For the multiplication itself, the module for scalar-vector multiplication is used. The output is stored at a provided memory address which can be the same as the input one.

Merging the vinegars of layer one into the central map is also handled by this module.

c) Triangular matrix evaluation: This module is called *batch_quad_trimat_eval* in the implementation. Triangular matrix evaluation is done serially in two nested loops. This happens once for the first layer and twice for the second. The input is an address of the matrix, the result is stored in-place.

d) Addition: The module is called *gf16_add* in the implementation. This module adds two words that are BUS_WIDTH long (i.e., a vector of multiple elements) from the provided addresses and saves them at a destination address. In the case of $\mathbf{GF}(16)$ the addition is equivalent to a bitwise xor.

e) Memory copying: The *memcpy* module copies data from one memory address to another.

f) Memory resetting: The *reset_mem* module writes zeros over all the bits in the specified address range.

g) Linear equation solver: The most complicated module is, by far, the linear equation solver. The GSMITH [20] solver is used, whose structure is described in subsection III-B.

h) Main controller: A controller responsible for issuing all the control signals needed by the design.

All modules handle the loading of the needed data from the block RAM by themselves. The block diagram of the design is shown in Figure 1.

B. Linear equation solver

A linear equation solver is used twice during the signing process to solve a set of linear equations, and to calculate an inverse matrix.

One of the state-of-the-art approaches is the GSMITH [20] architecture. It is highly parallel and allows for quick computation. There are multiple variants proposed by the authors in the same paper but due to the data handling and speed, the original GSMITH seems best for this purpose.

We further modified the GSMITH solver to allow computation of the inverse of a matrix by elimination with an augmented identity matrix.

C. Communication

The industry standard for connecting the ARM processor and the FPGA sides is the AXI bus. There are three types of the AXI [21] bus—AXI, AXI lite and AXI streaming.

Both AXI and AXI lite use memory mapping. AXI lite allows for single-bit memory map transactions only, has very low overhead and is ideal for control signals. The full-fledged AXI allows memory map burst transactions. AXI lite is used for the control signals and the full fledged AXI is used for data transfers of the digest, the required random data for the vinegars for the first layer, and the signature.

The modules communicate with the controller using control signals and load the data needed for their computations directly from the memory and write back their results. They do not need to communicate directly between themselves. When data from both the key ROM and the all-purpose RAM are needed in one step, the data are loaded at the same time, saving on clock cycles.

D. Parametrisation

The whole accelerator is designed to be parameterisable. Different bus widths, number of variables (both oils and vinegars) for each layer, or the number of multipliers that run in parallel can be selected. The parameters are inspired by the reference implementation, and most of the paramters that can be changed there can also be changed in our design. This also includes the number of vinegars and oils, and the Galois field can be changed as well if the $\mathbf{GF}(16)$ multiplier is replaced by one that works over the respective field.

E. Parallelisation

The part that benefits from parallelisation the most is the matrix multiplication. The main multiplier implements $\frac{BUS\_WIDTH}{ELEMENT\_SIZE}$ individual $\mathbf{GF}(16)$ multipliers. Both parameters can be specified before the synthesis and directly affect the area used. In our case, the element size is four bits (for $\mathbf{GF}(16)$).

Every time a chunk of data is loaded from the memory, all of it can be multiplied in the next clock cycle. Adding is also parallelized in the same way as multiplication is—all the loaded bits are added in one clock cycle.

Latency masking is another way of speeding the design up by loading data from the memory while performing operations on other, already loaded, data. Every time some data must be loaded and processed, latency masking is implemented.

IV. Implementation

A. Platform and language choice

For the design, we chose the ZedBoard[1] which contains the Zynq-7000 series SoC (XC7Z020-CLG484). The SoC FPGA has 85000 logical cells available, as well as 560 KiB Block RAM. The private key is stored in the block RAM. There is also an integrated dual-core ARM Cortex A9 processor, which allows performance comparison.

The accelerator was developed in VHDL. For the ARM core, the reference implementation [13], which is written in C, was used and it was slightly modified to run on the ARM processor.

B. Top-level memory implementation and structure

The memory is implemented using BRAMs and distributed flip flops. The BRAM is used for two purposes, the first one being temporary storage. The individual modules load content from the temporary storage into registers for processing. The accelerator assumes that the whole block contains zeros when the accelerator starts receiving the data to be signed. The RAM is dual-port, therefore writing and reading can be performed simultaneously, which is used for masking the memory latency. Besides the temporary storage for the modules, there are three special blocks accessible over AXI in this memory for storing the digest, the vinegars for the first layer, and the output.

The second purpose of using BRAM is storing the key. The key storage structure is organized in the same fashion as in the reference implementation, with one exception, i.e., the key storage is modified that each part of the key always starts at an address divisible by 64. The length of memory block is set using the parameter BUS_WIDTH. Each block can be addressed and read within one memory cycle.

[1] https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/zedboard/zedboard-board-family

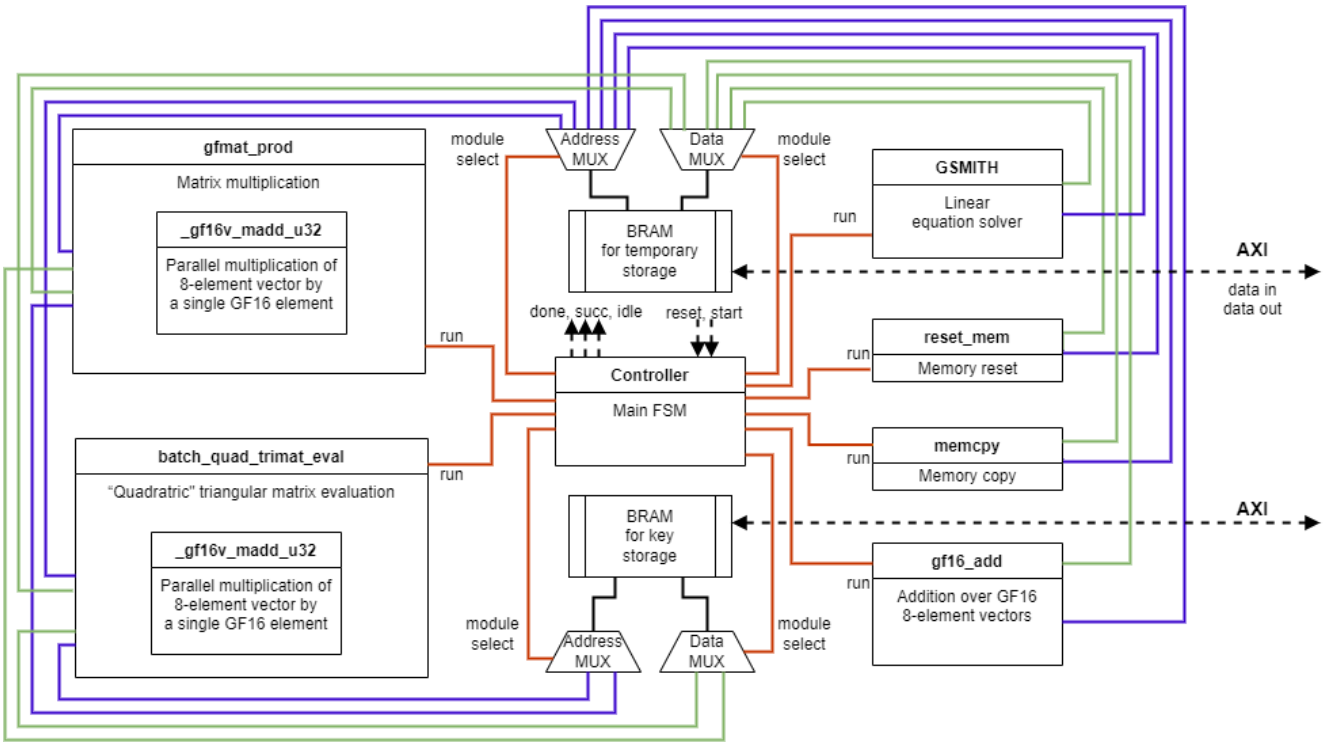

Fig. 1. Block diagram

C. GSMITH

GSMITH itself is a very effective and parallel structure without a need for an external controller. However, it is designed so that a new matrix row must be loaded in each clock cycle. To deal with this, we add an externally driven clock enable signal to its working registers to control the memory reads.

As mentioned in subsection III-B, we further modified the solver to provide an inverse of the matrix. This is done by the Gaussian elimination of the original matrix with an augmented identity matrix. Working registers of the solver are doubled and the Gaussian elimination is performed on the input matrix, while the same operations are performed on the augmented matrix in the next clock cycle. In the end, if the matrix is invertible, the result is stored in the registers that the original matrix was loaded into.

D. Other modules

The multiplication in $\mathbf{GF}(16)$ is implemented as a lookup table. This scalar multiplier is used in other blocks of the design. Addition in the finite field is implemented as a bitwise xor. The triangular matrix evaluation is implemented as a separate module because of memory latency. In this case, computation using GSMITH would be faster but thanks to the structure of the matrices it is possible to load and process them one column at a time. Therefore, once a column is loaded, processing can start directly and memory latency can be masked by loading the next column while the previous column is still being processed. All modules perform their own memory operations and are controlled by their own controllers, which provide flags to the main controller.

V. Evaluation

Several different metrics were evaluated for different values of the BUS_WIDTH parameter. These include the used FPGA area, power consumption, number of needed clock cycles and the maximum frequency. This section presents these tests in detail. All the presented data are based on post-implementation reports generated by Xilinx Vivado 2021.2.

Changing the parameters, such as the bus width, is done in a configuration file. The bus width directly affects the number of LUTs used, and the computation speed.

For a comparison, we have run the reference C implementation on the ARM core where one signature takes approximately 13.227 ms ($8,809,000$ clock cycles at 666 MHz).

TABLE III
Used area according to the bus width

Bus width (bits)	Number of LUTs	Number of FFs
8	40425	34962
16	35833	22533
32	39992	35190

A. Used area

The used area was measured in terms of number of LUTs and flip flops. Table III presents the used area for different bus widths. If the 32-bit version is taken as the baseline, it makes sense that the 16-bit version uses less resources as fewer

TABLE IV
NUMBER OF NEEDED CLOCK CYCLES AND MAXIMUM FREQUENCY ACCORDING TO THE BUS WIDTH

Bus width (bits)	Clock cycles (with reset)	Clock cycles (without reset)	Maximum frequency (MHz)	Total time (ms) (with reset)	Total time (ms) (without reset)
8	281000	119000	33.60	8.346	3.354
16	132000	67000	33.56	3.933	1.996
32	72000	39000	40.16	1.796	0.973

multipliers are generated to work in parallel. The amount of needed resources is higher for the 8-bit variant because of the LUTs necessary for address multiplexing and signal routing.

A significant part of the area is consumed by the linear equation solver, which is not much affected by the bus width since a lot of logic is needed for its highly parallel cell structure. This is also why the differences in the number of LUTs are not more significant.

B. Number of clock cycles, maximum frequency, and total time

The clock frequency was set to the maximum possible frequency reported by the estimation tool in Vivado.

The design assumes that the BRAM is reset to zero before the signing starts. Therefore, the memory needs to be reset as some modules accumulate data onto the same address. This memory reset is done prior to the signing and consumes a significant amount of time. This could be solved by, for example, holding dirty bits for each memory block. The available BRAM does not support resetting using any other method than sequentially writing zeros over the whole memory region, byte by byte.

Table IV presents the number of clock cycles for different bus widths, with and without the memory reset. The number of clock cycles scales almost linearly with the bus width (more cycles for lower bus widths). The more data can be transferred in one cycle, the faster the computation is.

Table IV also presents the measurements of the maximum frequency and accordingly calculated execution time. Based on our examination, the maximum frequency similar for the 8-bit and 16-bit variant but rises when the bus width increases.

C. Power consumption

Table V presents the estimated power consumption of the design as well as the total power consumption per successful signature generation (without reset). The power consumption corresponds to the amount of resources used. The total power used per one generated signature is reported in milliwattseconds.

TABLE V
POWER CONSUMPTION ACCORDING TO THE BUS WIDTH

Bus width (bits)	Power consumption (W)	Power used (mWs)
8	0.307	1.03
16	0.284	0.57
32	0.298	0.29

VI. COMPARISON WITH STATE OF THE ART

As presented in subsection II-D, we are aware of four hardware implementations of Rainbow. In this section, we present a comparison of our work with these implementations.

Table VI presents the existing hardware implementations as well as our results. The parameters (second column) are explained in subsection II-B. Most of the existing implementations use different parameters than those used in the NIST standardization process. Our work was also tested on different hardware and with parameters submitted into the third round of the NIST standardization process. The only implementation that uses such parameters (by A. Ferozpuri et al. [18]) uses parameters for the second round which uses a very different and highly parallel structure when compared to our work. Our work also includes the comparison between running the same algorithm both in software and hardware on the same chip.

VII. CONCLUSION

In this work, we developed a parameterisable accelerator for the Rainbow signature scheme and evaluated how different data bus width affect the synthesised design in terms of the area used, speed of computation and power consumption. We also presented a comparison with the state of the art.

Our design generates a signature based on the digest and provides the result over AXI. The data bus width can be adjusted from 8 to 32 bits which directly affects the computation time and the consumed FPGA area. The execution time scales linearly with the data bus width and based on our testing, we can achieve significant speedup. We provide detailed results that allow selection of the correct parameters based on the need of the user.

ACKNOWLEDGEMENTS

Tomáš Přeučil and Petr Socha are members of the student research team of the internal Czech Technical University (CTU) project No. SGS20/211/OHK3/3T/18.

REFERENCES

[1] J. Frankenfield, *Quantum Computing*, Investopedia, [cit. 2022-01-16]. [Online]. Available: https://www.investopedia.com/terms/q/quantum-computing.asp

[2] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, p. 14841509, oct 1997. [Online]. Available: https://doi.org/10.1137/S0097539795293172

[3] D. Beckman, A. N. Chari, S. Devabhaktuni, and J. Preskill, "Efficient networks for quantum factoring," *Phys. Rev. A*, vol. 54, pp. 1034–1063, Aug 1996. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.54.1034

[4] D. Bernstein and A. Lenstra, *A general number field sieve implementation*, 11 2006, vol. 1554, pp. 103–126.

TABLE VI
COMPARISON OF THE EXISTING IMPLEMENTATIONS

Work	Parameters	Platform	LUT	FF	Clock cycles	Max. f. (MHz)	Ex. time (μs)
S. Balasubramanian et al. [15]	$\mathbf{GF}(16), 17, 12, 12$	Virtex-4	63593	6106	804	67	12
S. Tang et al. [16]	$\mathbf{GF}(16), 17, 12, 12$	Stratix II	?	?	198	50	3.96
H. Yi et al. [17]	$\mathbf{GF}(16), 17, 13, 13$	TSMC-0.18 μm	30000 GE		242	50	4.9
A. Ferozpuri et al. [18]	$\mathbf{GF}(16), 32, 32, 32$	Kintex-7	27712	27679	1980	111	17.84
A. Ferozpuri et al. [18]	$\mathbf{GF}(16), 32, 32, 32$	Virtex-7	27556	27675	1980	181	10.93
Our work (8bit)	$\mathbf{GF}(16), 36, 32, 32$	Zynq-7000 Artix-7	40425	34962	119000	33.67	3354
Our work (16bit)	$\mathbf{GF}(16), 36, 32, 32$	Zynq-7000 Artix-7	35833	22533	67000	33.56	1996
Our work (32bit)	$\mathbf{GF}(16), 36, 32, 32$	Zynq-7000 Artix-7	39992	35190	39000	40.16	973
Reference implementation (SW)	$\mathbf{GF}(16), 36, 32, 32$	Zynq-7000 ARM	N/A	N/A	8809000	N/A	13227

[5] C. Zalka, "Fast versions of shor's quantum factoring algorithm," *arXiv: Quantum Physics*, 1998.

[6] J. Gambetta, *IBMs Roadmap For Scaling Quantum Technology*, IBM, [cit. 2021-05-13]. [Online]. Available: https://www.ibm.com/blogs/research/2020/09/ibm-quantum-roadmap/

[7] *Cyber Centres summary review of final candidates for NIST PostQuantum Cryptography standards*, Canadian Centre for Cyber Security, [cit. 2021-05-13].

[8] J. Ding and D. Schmidt, "Rainbow, a new multivariable polynomial signature scheme," vol. 3531, 06 2005, pp. 164–175.

[9] A. Kipnis, H. Hotzvim, J. Patarin, and L. Goubin, "Unbalanced oil and vinegar signature schemes," 01 2000.

[10] J. Ding, *Rainbow*, supporting documentation for the NIST contest submisstion. [cit. 2021-05-13]. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/round-2-submissions

[11] N. Courtois, L. Goubin, W. Meier, and J.-D. Tacier, "Solving underdefined systems of multivariate quadratic equations," in *Public Key Cryptography*, D. Naccache and P. Paillier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 211–227.

[12] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*, first edition ed. W. H. Freeman, 1979. [Online]. Available: http://www.amazon.com/Computers-Intractability-NP-Completeness-Mathematical-Sciences/dp/0716710455

[13] J. Ding, *Rainbow*, supporting documentation for the NIST contest submisstion. [cit. 2021-05-13]. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/round-2-submissions

[14] *Rainbow Signature One of the Three NIST Post-quantum Signature Finalists*, [cit. 2021-05-27]. [Online]. Available: https://www.pqcrainbow.org/

[15] S. Balasubramanian, A. Bogdanov, A. Rupp, J. Ding, and H. W. Carter, "Fast multivariate signature generation in hardware: The case of rainbow," in *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, April 2008, pp. 281–282.

[16] S. Tang, H. Yi, J. Ding, H. Chen, and G. Chen, "High-speed hardware implementation of rainbow signature on fpgas," vol. 2011, 11 2011, pp. 228–243.

[17] H. Yi and U. M. Khokhar, "Under quantum computer attack: Is rainbow a replacement of rsa and elliptic curves on hardware?" *Sec. and Commun. Netw.*, vol. 2018, jan 2018. [Online]. Available: https://doi.org/10.1155/2018/2369507

[18] A. Ferozpuri and K. Gaj, "High-speed fpga implementation of the nist round 1 rainbow signature scheme," in *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2018, pp. 1–8.

[19] W. Beullens, "Breaking rainbow takes a weekend on a laptop," Cryptology ePrint Archive, Report 2022/214, 2022, https://ia.cr/2022/214.

[20] A. Rupp, T. Eisenbarth, A. Bogdanov, and O. Grieb, "Hardware sle solvers: Efficient building blocks for cryptographic and cryptanalytic applications," *Integration*, vol. 44, no. 4, pp. 290–304, 2011, hardware Architectures for Algebra, Cryptology and Number Theory. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016792601000057X

[21] ARM, "Amba axi and ace protocol specification," online, 2003, 2004, 2010, 2011, http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf.