The CRNS Framework and its Application to Programmable and Reconfigurable Cryptography

SAMUEL ANTÃO and LEONEL SOUSA, INESC-ID / Instituto Superior Técnico - TU-Lisbon

This article proposes the Computing with the Residue Number System (CRNS) framework, which aims at the design automation of accelerators for Modular Arithmetic (MA). The framework provides a comprehensive set of tools ranging from a programming language and respective compiler to back-ends targeting parallel computation platforms such as Graphical Processing Units (GPUs) and reconfigurable hardware. Given an input algorithm described with a high-level programming language, the CRNS can be used to obtain in a few seconds the corresponding optimized Parallel Thread Execution (PTX) program ready to be run on GPUs or the Hardware Description Language (HDL) specification of a fully functional accelerator suitable for reconfigurable hardware and embedded systems. The resulting framework's implementations benefit from the Residue Number System (RNS) arithmetic's parallelization properties in a fully automated way. Designers do not need to be familiar with the mathematical details concerning the employed arithmetic, namely the RNS representation. In order to thoroughly describe and evaluate the proposed framework, experimental results obtained for the supported back-ends (GPU and HDL) are presented targeting the implementation of the modular exponentiation used in the Rivest-Shamir-Adleman (RSA) algorithm and Elliptic Curve (EC) point multiplication. Results suggest competitive latency and throughput with minimum design effort and overcoming all the development issues that arise in the specification and verification of dedicated solutions.

Categories and Subject Descriptors: C.1.4 [Computer Systems Organization]: Processor Architectures—Parallel Architectures; G.2.3 [Mathematics of Computing]: Discrete Mathematics—Applications

General Terms: Algorithms

Additional Key Words and Phrases: Residue Number System (RNS), modular arithmetic, cryptography, general-purpose graphical processing units (GPGPU), framework

ACM Reference Format:

Antão, S. and Sousa, L. 2013. The CRNS framework and its application to programmable and reconfigurable cryptography. *ACM Trans. Architect. Code Optim.* 9, 4, Article 33 (January 2013), 25 pages.

DOI = 10.1145/2400682.2400692 <http://doi.acm.org/10.1145/2400682.2400692>

1. INTRODUCTION

Functionalities based on RNS and MA can be found in a variety of applications in Digital Signal Processing (DSP) [Mohan 2002] including cryptography [Guillermin 2010; Antão et al. 2009]. Accelerators for these applications may completely rely in MA or in a variety of blocks for which only part of them correspond to modular operations.

This work was supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under the project PEst-OE/EEI/LA0021/2011. This work was accomplished in the scope of the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC).

Authors' addresses: S. Antão (corresponding author), L. Sousa, Department of Electrical and Computer Engineering, Instituto Superior Técnico, Universidade Técnica de Lisboa, 1049-001 Lisboa, Portugal, and Instituto de Engenharia de Sistemas e Computadores Investigação e Desenvolvimento em Lisboa (INESC-ID), 1000-029 Lisboa, Portugal; email: samuel.antao@inesc-id.pt.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/01-ART33 \$15.00

DOI 10.1145/2400682.2400692 <http://doi.acm.org/10.1145/2400682.2400692>

Hence, for efficiency's sake, the designer should not only be aware of the best design practices but also of the mathematical details concerning MA. It turns out that the development time and effort for these accelerators is closely related to the MA building blocks. Note that systems supporting MA can be further used to compute generic integer arithmetic modulo 2^n , with n the datapath width of the computing platforms.

In spite of the flexibility of software solutions based on general-purpose processors, these solutions may not be attractive in what concerns latency and cost/resources. In order to take full advantage of software implementations, designers must take into account the properties of the underlying hardware platform, namely the datapath size, the operations natively/efficiently supported, the parallelism capabilities, etc. An alternative to the implementations based on software is the development of embedded systems to accelerate the most demanding operations, including the ones related to MA. Still, these embedded systems are usually supported on dedicated hardware implementations that constrain the flexibility and potential reutilization of hardware blocks previously designed. Reconfiguration is often the solution to overcome some of these problems.

The CRNS proposed in this article aims at aiding the development of flexible, yet efficient, MA accelerators. By targeting highly parallel architectures, this framework embeds the capabilities of the RNS, which supports an alternative data representation that splits the processing up into independent channels, allowing simultaneous execution in several processing elements. In other words, with the RNS it is possible to obtain an efficient parallel execution even for algorithms teeming with data dependencies. Moreover, the RNS concept has been successfully tested for low-latency computation in highly parallel architectures such as GPUs, while targeting hundreds-of-bits wide data for cryptographic purposes [Antão et al. 2011; Szerwinski and Güneysu 2008]. CRNS also includes a programming language and the respective compiler with different back-ends, each targeting different parallel programming platforms. Currently, the framework deploys efficient implementations using the PTX programming language for Nvidia GPUs and a target-independent HDL specification of a fully programmable and reconfigurable architecture suitable for Field Programmable Gate Arrays (FPGAs) along with the microcode required to program it. The framework's main building block is the LLVM Compiler Infrastructure [LLVM 2012], which is a tool that is supported by a community of compiler developers and used across a variety of tools comprising different languages and devices. Therefore, the usage of LLVM in the CRNS design flow is expected to ease the future interaction with other tools and ease the development of more and more complex targets for the CRNS back-end. The current version of the framework is thoroughly tested recurring to cryptographic applications. The main contributions of the work herein presented are the following.

- To the best of the authors' knowledge, no effort for developing a tool to aid in the hardware design for MA with RNS has been made in the past. In the related state-of-the-art, approaches to design accelerators for MA based on RNS were proposed targeting GPUs [Antão et al. 2011; Szerwinski and Güneysu 2008] and reconfigurable hardware [Schinianakis et al. 2009; Nozaki et al. 2001]. However, these approaches correspond to dedicated implementations that can only support a particular application.
- The development of CRNS comprises methods that enable the implementation of MA-based algorithms taking advantage of the RNS capabilities in a completely transparent way: all precomputed information, including modular/RNS-related constants and data's dynamic ranges, is automatically extracted from the provided algorithm and handled by the framework's tools. It allows the fast generation of efficient implementations that are less error prone than manually crafted ones.

Fig. 1. Hierarchy for computing the target algorithm based on the RNS representation. The operands used in the description of an MA algorithm. MA algorithms are converted to the RNS representation so that the SMA is computed in parallel by the several channels, each one computing CMA. Thereafter, the results are converted back to the original representation used in the SMA. The challenge in the implementation of the RNS modular operations is to find efficient ways of computing operations such as addition, multiplication, and modular reduction described in SMA with CMA.

- The CRNS innovates the implementation of RNS-related operations, namely the implementation of RNS forward and reverse conversions, in GPUs. Additionally, it enhances a scalable architecture tailored for reconfigurable devices also with the ability to compute generic algorithms based on MA.
- In spite of the complexity associated with RNS arithmetic, the interfaces of the implementations obtained with CRNS are simple, consisting of First-In First-Out (FIFO) buffers for a hardware implementation, and a comprehensive set of C functions for the GPU-based implementations.
- The CRNS provides in a single package the ability to target a variety of platforms ranging from GPUs to reconfigurable hardware with potential applications in embedded systems.

The article is organized as follows: Section 2 presents the details of the proposed framework broken down in its main components as well as the foundations of the RNS, more detailed in Appendix A. The details on how the framework's back-end targets the different computing platforms are presented in Section 3. Section 4 illustrates how the CRNS can be used and provides experimental results targeting modular exponentiation (the main operation of applications such as the RSA cryptographic algorithm) and EC point multiplication. Finally, Section 5 draws the main conclusions of the developed work.

2. FRAMEWORK COMPONENTS

The main goal of the framework is to allow the designer to benefit from the RNS properties, namely the parallelism at the arithmetic level, without even noticing that RNS is underlying the system. By defining a basis $B_i = \{m_{1,i}, \dots, m_{h_i,i}\}$ of pairwise coprime elements and an associated dynamic range $M_i = \prod_{e=1}^{h_i} m_{e,i}$, an integer $X < M$ has a corresponding RNS representation $x_{1,i} = X \bmod m_{1,i}, \dots, x_{h_i,i} = X \bmod m_{h_i,i}$. The main advantage of the RNS representation is the possibility to perform in parallel the same operations one would do with integers, $Z = X \odot Y$ (where $\odot$ is either an addition/subtraction or a multiplication and with $Z < M_i$), as $z_{1,i} = x_{1,i} \odot y_{1,i} \bmod m_{1,i}, \dots, z_{h_i,i} = x_{h_i,i} \odot y_{h_i,i} \bmod m_{h_i,i}$. An operation applied over the RNS representation $\bmod m_{e,i}$ is said to be performed on the RNS channel defined by $m_{e,i}$. Figure 1 and Example 2.1 illustrate the correspondence between an operand X of the System Modular Arithmetic (SMA), and the residues $x_{e,i}$ which are used within the Channel Modular Arithmetic (CMA) (see Appendix A for more details).

Fig. 2. Proposed framework's components. (i) *Programming Language*, to allow the designer to describe the target algorithm (program); (ii) *Back-End List*, to provide the implementations of the input algorithm for each of the supported targets; and (iii) *Compiler* to link the two previous components.

Example 2.1. Consider that one wants to compute the SMA operations $Z_+ = X + Y$ and $Z_\times = X \times Y$ for $X = 14$ and $Y = 23$ using the CMA. The expected results for these operations are $Z_+ = 27$ and $Z_\times = 322$. In order to obtain the CMA equivalent, one can define a basis B_1 of $h_1 = 3$ elements $B_i = \{7, 8, 9\}$. One can easily verify that all elements $m_{i,1}$ are pairwise coprime and that the dynamic range of B_i is $M_i = 7 \times 8 \times 9 = 504$. Since $M_i > Z_+$ and $M_i > Z_\times$ one knows that a mapping of SMA to CMA is possible to be obtained. One has only to obtain the residues $x_{i,1} = X \bmod m_{i,1}$ and $y_{i,1} = Y \bmod m_{i,1}$: $\{x_{1,1}, x_{2,1}, x_{3,1}\} = \{0, 6, 5\}$ and $\{y_{1,1}, y_{2,1}, y_{3,1}\} = \{2, 7, 5\}$. To compute the operations in CMA one can now compute $z_{+,i,1} = x_{i,1} + y_{i,1} \bmod m_{i,1}$ and $z_{\times,i,1} = x_{i,1} \times y_{i,1} \bmod m_{i,1}$. Therefore the result is $\{z_{+,1,1}, z_{+,2,1}, z_{+,3,1}\} = \{2, 5, 1\}$ and $\{z_{\times,1,1}, z_{\times,2,1}, z_{\times,3,1}\} = \{0, 2, 7\}$. The correctness of the mapping can be finally confirmed by checking the obtained residues against the values $Z_+ \bmod m_{i,1}$ and $Z_\times \bmod m_{i,1}$. With these operations an SMA operation was accomplished with the CMA in 3 parallel flows.

In the proposed framework an algorithm in SMA is mapped to CMA. The computing organization, precomputation of constants, channel definitions, and all optimizations are automatically handled by the tools. The proposed framework organization is depicted in Figure 2. The following subsections detail the properties and characteristics of the programming language and compiler and how the RNS arithmetic is accommodated. Given that the Back-End List is strongly related with the device's properties, it is addressed separately in Section 3.

2.1. Programming Language

The first framework component that the designer has to deal with is the Programming Language (PL). The PL is similar to the standard C programming language with some extensions for MA, but also some restrictions. In order to support a more clear understanding of the language, Figure 3 presents the sample code of a modular exponentiation using the framework's programming language. The main differences between this language and the C language are the following: (i) only the 'int' and 'intx' identifiers are accepted where x is the size in bits of the variable associated to it; (ii) a header should be provided as '(list_out) = (list_in)' where list_out and list_in are a list of comma separated outputs and inputs, respectively - the list_in has associated a intx identifier specifying their bit-width, which is essential to estimate the required dynamic range at compile time (iii) no input-dependent conditionals are supported since all conditionals have to be evaluated at compile time; and (iv) the bitwise logic operations cannot depend on the inputs since these operations are not supported by the RNS arithmetic in the back-end, thus should be computed at compile time. Note that, concerning the identifier intx, the bit-width information is strictly

01. (outputA) = (int1024 inputA)	14. while(M)
02. {	15. {
03. /*modulo:*/	16. int M2 = M & D;
04. int N = 0xbf4325a19 ... c92e820f7;	17. R = (R * R) % N;
05. /*exponent:*/	18. R = (R * R) % N;
06. int D = 0x922ad9c67 ... c1c856434;	19. if(M2)
07. /*mask:*/	20. R = (R * inputA) % N;
08. int M = 0x400000000 ... 000000000;	21. M = M >> 1;
09.	22. }
10. /*compute the exponentiation*/	23. }
11. int I = inputA % N;	24. }
12.	25. }
13. int R = I;	26. outputA = R;
	27. }

Fig. 3. Modular exponentiation described using the proposed framework supported language. In this program $\text{outputA} = \text{inputA}^D \bmod N$.

required to estimate the algorithm's dynamic range similarly to other approaches used in high-level synthesis tools [Mahlke et al. 2001]. A representation with unlimited (regardless of the computational resources available to the compiler) precision is supported in hexadecimal (0x), decimal, octal (0), or binary (0b) format. The operations addition, subtraction, multiplication, modulo, and, or, xor, right and left shift, and complement are supported using the same symbols as C. The operation's precedence can be controlled by using the '(' and ')' accordingly. The language supports loops by using the statement 'while(cond.)' and conditions by using the statement 'if(cond.)' and 'else'. The creation of contexts can be done by delimiting them with '{' and '}' being the declaration of variables in these contexts only available for that particular context.

2.2. Compiler

The core of the compiler is the LLVM Compiler Infrastructure [LLVM 2012]. The utilization of LLVM, which is maintained by a large community of developers, has several advantages, mainly:

- by using LLVM, the tool's components become a part of a comprehensive and widely adopted infrastructure, which enhances its scalability and leverages future developments;
- LLVM offers the functionality to run the program through a Just-In-Time (JIT) compilation step. As discussed ahead, this functionality is of extreme relevance to the CRNS framework given that the compilation stage of an RNS-based application requires the accurate estimation of the required dynamic range, which can only be done at runtime.

The implementation flow to use LLVM consists of: (i) translating the input program to the LLVM Internal Representation (IR), which is a target-independent language, (ii) applying several optimization and analysis passes to the LLVM IR, and (iii) extracting the required information so that the back-end can produce the final implementation.

The compiler is the framework's tool that interfaces with the designers' target algorithm, hence it performs the lexical evaluation and parsing of the program described with the framework's programming language (see Figure 2). Besides the LLVM as the main building block, the framework also makes use of other software tools and libraries, namely the Flex [2012] and Bison [GNU - Bison 2012] are used to load the LLVM IR of the target algorithm to the framework's implementation. After the program is loaded without any parse error, the tool advances to the arithmetic-related stage where extensive usage of the GNU Multiple Precision Arithmetic Library (GMP) [GNU - GMP 2012] is made to support multiprecision operations, namely the ones with dynamic

precision. Because such multiprecision operands are not natively supported by LLVM, most of the LLVM IR consists of instantiations of functions that handle pointers to multiprecision operands for the GMP library to deal with.

Along with the input program, the compiler must be informed about the desired datapath width (k) of the RNS channels. Small values of k may make it impossible for the tool to find the RNS basis and to verify some mathematical conditions, namely in what concerns the computation of the conversion offsets in the reduction (see Appendix A.3). Large values of k may result, in the limit, in only one channel that would not allow exploiting the RNS parallelization capabilities. In any case, each time a value of k that results in any of the aforementioned problems is inserted, the tool will provide the designer with the required feedback.

Due to the RNS arithmetic properties, the algorithm programmed by the user must comply with the two main limitations of the CRNS.

- No data-dependent conditionals allowed.* The CRNS supported programs' conditions must be computed at compile time. The reason for this arises from the estimation of the dynamic range of the program at compile time. Without knowing the conditions, the dynamic range estimation would have to be worst-case-driven which could result in much larger dynamic ranges than actually required. Given that some of the RNS operations' complexity grows quadratically (see Appendix A.3 for an example) with the number of channels (which are proportional to the dynamic range), overestimating these values would greatly compromise the performance. Also, the magnitude comparison is a complex operation in RNS which can be efficiently computed for given fixed moduli-sets but is not compatible with the generic algorithms that CRNS is meant to support based on flexible and scalable moduli-sets. Note that the purpose of this limitation is not to ease the parallelism extraction by CRNS. Data dependencies and data-independent control dependencies are still allowed.
- Outputs must not depend on bitwise operations.* The bitwise operations are only supported to define the conditions in the control instructions. The computation of bitwise (nonlinear) operations with RNS is still an open research topic.

If any of the aforementioned limitations conflicts with the user program the correspondent feedback is given. In the following paragraphs, the remaining main passes and analysis applied to the LLVM IR in order to obtain the final RNS-based program are detailed. Only the main mathematical steps are addressed. Appendix A provides a detailed insight about all mathematical details concerning the RNS operations.

LLVM built-in passes. While loading the LLVM IR, the framework benefits from some built-in optimizations such as *constant merging*, *operand reassociation*, and *instruction combining*. Given that most instructions are set as function calls, these optimization steps do not automatically apply. Therefore, the function calls performed in the program were inserted in the list of instructions to which these optimizations should apply by updating the LLVM sources accordingly.

Obtain the RNS domains. The modulo operations supported by the framework are computed by a method that is often referred to as Basis Extension (BE). This method requires the data to be converted from its Original Domain (OD) to a different domain called Montgomery Domain (MD) (see Appendix B for details). For each value N_d used in operations modulo N_d , there is an MD. Given that operations in the different domains require different constants to be precomputed, all the domains must be identified at compile time. Therefore, the list of values N_d has to be extracted from the program in order to precompute and initialize the data to be used during the computation.

Identify the RNS Worklist. During the evaluation of the program, not all the instructions should be considered. Only the instructions that are dependencies of the outputs need to be considered. For example, all the instructions that are used to define conditions are ignored. This pass creates a worklist of all instructions that need to be considered for the RNS computations.

Merge Multiplications with Basis Extensions. The establishment of a dynamic range which accommodates the operands in the program is mandatory to obtain an efficient RNS implementation. Maintaining this dynamic range as low as possible results in several advantages both in terms of resources and performance. Therefore, this pass performs optimizations towards the reduction of the program's dynamic range such as the detection of the operations immediately followed by a reduction (employing a BE), since the required dynamic range of a combination of any operation and a reduction is potentially lower than the sequence of the two separated operations. In particular, for the modular multiplication $R = AB \bmod N$ of two numbers $A, B < N$, the multiplication itself results in a maximum value of $N^2 - 1$, while the multiplication followed by the reduction is bounded by $R < 3N$; see Appendix A.3. For this particular example, the number of bits required to represent the dynamic range of the multiplication is almost twice the number of bits required when the multiplication is followed by the reduction. Therefore, this pass merges the two instructions into a single one.

Define Montgomery Domains and Conversions. In the target algorithm, the designer may be interested in using MA modulo $N_1, N_2, \dots, N_d$. For the architecture to support the reduction operation for these different moduli, it must also support the conversion of the data to/from the d different MDs, given that there is one MD for each modulus N_i in the SMA (see Appendix B for details). Each time an operand is involved in a reduction, it is previously converted to the corresponding MD. Another case that requires conversion from/to MDs occurs when data in the two different domains (MD and OD) is used in the same operation. To assure the operations' correctness, the inputs' domains must be the same. Therefore, the data is converted from the MD to the OD and the operation is computed using the OD's representation each time the operands' domains do not match. This pass introduces instructions that perform the conversions in order to assure that all the MDs can be used while keeping data consistency. Another task of this pass is to identify the constants that are involved in the operations of any of the MDs so as to annotate each constant and assure that there is an instance of this constant stored in memory in the appropriate domains.

Remove Inefficient Basis Extensions. The BE used in the reductions and domains' conversions is the most computationally demanding operation supported by the framework. Therefore, the BE should be avoided unless strictly required or if it does not have a remarkable impact in the performance. This pass detects the BE that can be avoided, namely the ones that result from the reduction of multiplications by constants. In this case, the constant can be a small number and the width of the result of the multiplication be only slightly higher than the nonconstant operand itself, i.e., the dynamic range required by this multiplication with or without the BEs following it will be almost the same. Currently, the criterion for removing the BE in these cases requires the constant to be smaller than the channel width k defined by the user.

Evaluate Dynamic Ranges. After inserting all of the program's modifications, it is possible to evaluate the required dynamic ranges. One of the tasks of this pass is to find the RNS bases that support the required dynamic ranges M_1 and M_2 : (i) M_1 of B_1 must be greater than the maximum number to be represented during the program's execution; and (ii) B_2 is obtained such that $M_2 > M_1$ and $\gcd(M_2, M_1) = 1$. In practice, the bases are obtained by identifying the values $m_{e,i} = 2^k - c_{e,i}$ for increasingly higher

values of $c_{e,i}$ that are pairwise coprime with all the other moduli that are already part of the bases. By considering A and B with S_A and S_B bits, respectively, the dynamic range for the: (i) addition is $\max(S_A, S_B) + 1$; (ii) multiplication is $S_A + S_B$; (iii) reduction of A is $S_{\lceil R \rceil}$ where $R = (2^{S_A} + (Q + M_1)N)M_1^{-1}$; due to the algorithms' properties Q should be replaced by $Q + M_1$ for dynamic range evaluation purposes (see Appendix A.3). In other applications such as fixed point digital signal processing, operand bit-width estimation can be accomplished using semianalytical approaches and error models [Lee et al. 2006]. In the proposed framework the dynamic range is accomplished by taking advantage of the JIT compilation supported by LLVM which allows the execution of the program corresponding to the LLVM IR. By taking into account the aforementioned operation's dynamic ranges and assuming the inputs have its maximum bit-width size defined, it is possible to accurately evaluate the required dynamic range. Note that the dynamic range also depends on other properties associated with the BE, and the dynamic range of the operations depends on the dynamic range estimated so far. Therefore, the dynamic range evaluation may have to be computed iteratively until the dynamic range size stabilizes.

Extract Conditions. The framework allows the designer to specify the test of conditions in the program. However, since the architecture does not foresee the possibility of data evaluation, each condition must not depend on the input data. With this, the compiler is able to solve all the conditions at compile time which, recalling that the dynamic ranges may depend on these conditions, allows the framework to accurately compute the required dynamic range without the need of overestimating its value. The JIT compilation feature is also used to run the program and store the Boolean values resulting from the conditions in the exact sequence they will occur during execution. With this, it is possible to obtain a string of zeros and ones and pass this string to the back-end which can control the program flow according to the content of this string.

Extract Register and Constant Map. The final step accomplished by the framework evaluates the number of registers and constants required by the program. The program's variables and constants are assigned to addresses in ascending order. Two consecutive addresses are assigned to each variable to store the value modulo $m_{e,1}$ and $m_{e,2}$, respectively. All the address management is made by this pass, which also assures that addresses for temporary variables are reused in order to keep the required number of memory locations as small as possible.

3. BACK-END DETAILS

After optimizing the LLVM IR representation for the RNS-based operations and extracting constants and conditions, it is then possible to generate an optimized RNS implementation. The CRNS was constructed to be easy to add at the back-end new target systems that can benefit from the RNS properties. Currently the framework is supporting two different back-end classes of devices.

- Compute Unified Device Architecture (CUDA) enabled Nvidia GPUs, by deploying an optimized kernel specified with the PTX language and a C template for the host using the CUDA Application Programming Interface (API).
- Reconfigurable Hardware- Application-Specific Integrated Circuit (ASIC), by deploying a fully functional processor specified in a technology independent HDL and the required microcode to program it. With this back-end the framework operates as a High-Level Synthesis tool that can be accommodated in an FPGA or ASIC design flow.

ALGORITHM 1: Optimized RNS reduction.**Require:** $Mr_{e,1}, Mr_{e,2}$.**Ensure:** $Mu_{e,1} = (RM_1 \bmod N)_{e,1}$ and $Mu_{e,2} = (RM_1 \bmod N)_{e,2}$.

- 1: In ch. e , 1: $q_{e,1} = \left| Mr_{e,1} \lambda_{e,1} \right|_{m_{e,1}}$
- 2: In ch. e , 2: $q_{e,2} = \left| \left(\sum_{j=1}^{h_1} q_{j,1} \lambda_{2ej} + \hat{\alpha}_1 \right) \lambda_{3e} \right|_{m_{e,2}}$
- 3: In ch. e , 2: $Mu_{e,2} = \left| (Mr_{e,2} + q_{e,2}) \lambda_{4e} \right|_{m_{e,2}}$
- 4: In ch. e , 2: $q_{e,1} = \left| Mu_{e,2} \lambda_{5e} \right|_{m_{e,2}}$
- 5: In ch. e , 1: $Mu_{e,1} = \left| \left(\sum_{j=1}^{h_2} q_{j,1} \lambda_{6ej} + \hat{\alpha}_2 \right) \lambda_{7e} \right|_{m_{e,1}}$
- 6: **return** $Mu_{e,1}, Mu_{e,2}$.

The following paragraphs briefly describe the key properties concerning the implementation of modular operations used by the back-end to deploy the implementation.

Addition, Subtraction, and Multiplication. The greatest advantage of RNS is that these operations in SMA correspond exactly to the same operations in CMA, i.e. parallel operations in all the RNS channels $m_{e,i}$ of the basis B_i . Therefore, each channel computes the very same operation in the channel, followed by a reduction also in the channel. This reduction relies on the property $a \equiv a_H 2^k + a_L \equiv a_H c_{e,i} + a_L \bmod m_{e,i}$, for $m_{e,i} = 2^k - c_{e,i}$ (see Appendix A.2 for details).

Reduction / Basis Extension Offset. The reduction is the most complex operation supported in RNS by the proposed framework. The reduction U of R is obtained as $MU = (MR + QN)/M_1$, where MX stands for the representation of X in an MD; the value X is said to be in the OD. The representation of the variables in an MD is required to efficiently compute a reduction in RNS. Algorithm 1 is an optimized reduction algorithm deduced in Appendix A.3, where the values λ correspond to precomputed constants. The purpose of this algorithm is to assure that $U \equiv R \bmod N$. The MD representations $MR = R(M_1^2 \bmod N)$ and $MU = (UM_1 \bmod N)$ are used to ease the computation. The value Q is a constant that can be efficiently computed with CMA in B_1 and assures that $(MR + QN)$ is a multiple of M_1 so that the division (in practice a multiplication by an inverse) does not introduce error. Therefore, given that QN is a multiple of N , $(MR + QN)/M_1 \equiv RM_1^2/M_1 + 0 \bmod N$ which means that $MU \equiv_M R \bmod N$. The name BE arises from the impossibility to represent M_1 in B_1 and therefore the data has to be extended to a basis B_2 to compute the inverse of M_1 . By construction, the herein proposed framework always uses the same number of moduli ($m_{e,f}$) in both basis B_1 and B_2 . Algorithm 1 shows the sequence of operations to be computed in each channel's arithmetic units for the basis B_1 and B_2 . The steps 1 and 2 of Algorithm 1 compute and convert the value Q to the basis B_2 and the steps 4 and 5 convert the result obtained in step 3 back to the basis B_1 . Since there are no simultaneous operations modulo $m_{e,1}$ and $m_{e,2}$ in the same channel e , the resources assigned to each channel can be used to compute with both moduli. Algorithm 1 uses the values $\hat{\alpha}_j$ that can be computed as

$$\hat{\alpha}_j = \left\lfloor \sum_{e=1}^{h_i} \frac{\text{trunc}_{t_j}(q_{e,1})}{2^{k-t_j}} + \Phi_j \right\rfloor / 2^{t_j}, \quad (1)$$

where trunc_{t_j} sets the $k - t_j$ least significant bits of $q_{e,1}$ to zero. The constants t_j and Φ_j are precomputed values that depend on the RNS basis used in the computations (see Appendix A.1 for details).

Input/Output conversions. The required input data X can be converted to RNS with the direct computation of $x_{e,i} = X \bmod m_{e,i}$ for each channel e of the basis B_i . Considering S_X the size of X in bits and kX_j the j -th word of the binary representation of X split in k -bit words, the forward conversion can be computed as

$$x_{e,i} = X \bmod m_{e,i} = \sum_{j=1}^{\lceil S_X/k \rceil} {}^kX_j \left(2^{k(j-1)} \bmod m_{e,i} \right) \bmod m_{e,i}, \quad (2)$$

where $2^{k(j-1)} \bmod m_{e,i}$ are precomputed constants.

The reverse conversion can be accomplished by using a scalable method based on the Chinese Remainder Theorem (CRT). This method can be applied by reusing the constant values used in Algorithm 1, step 4. For the value U represented in B_2 , $u_{e,2}$, the residues $q_{e,1} = |u_{e,2} \lambda_{5e}|_{m_{e,2}}$ can be computed and the final value of the conversion obtained as

$$U = \sum_{e=1}^{h_2} q_{e,1} M_{e,2} - \hat{\alpha}_2 M_2, \quad M_{e,2} = \frac{M_2}{m_{e,2}}. \quad (3)$$

Note that the value $\hat{\alpha}_2$ can be computed with the same method used for Algorithm 1, i.e. as in (1). As in the forward conversion, some of the constants, namely $M_{e,2}$ and M_2 , are much larger than the arithmetic that is computed in each channel. Therefore, they can be split in k -bit words as ${}^kM_{e,2,j}$ and ${}^kM_{2,j}$, respectively, so they can be easily handled with the resources available in the channel. Appendix A.1 provides more details about the methods for computing the aforementioned conversions.

3.1. GPU Back-End

The utilization of RNS arithmetic in GPUs has been successfully exploited in the literature [Szerwinski and Güneysu 2008; Antão et al. 2011]. These implementations suggested that not only can an RNS approach enhance the throughput of the operations being held, but also reduce the latency of such operations. However, the proposals in the related state-of-the-art consist of static implementations (using CUDA and Open Computing Language (OpenCL)) targeting a given algorithm. The CRNS framework allows one to address any given algorithm based on MA and obtain an implementation in a fully automated way. Furthermore, CRNS is comprehensive as it includes in the implementation the RNS forward and reverse conversions. To the best of the authors knowledge, there are no previous GPU implementations of these operations reported in the literature. On top of that, the CRNS implementations are deployed using the PTX language which is similar to an assembly language for Nvidia GPUs, enabling the fine-tuning of the implementation for these devices. Also, CRNS provides a complete and fully functional C template using the CUDA API for the GPU's host, which allows the user to load the PTX kernel effortlessly.

The support for general-purpose computation with Nvidia GPUs started with the Tesla architecture [Lindholm et al. 2008] which has evolved to slightly different architectures but all sharing the same basic attributes. Concerning the back-end implementation, the relevant information about the architecture is the supported parallel hierarchy: a GPU kernel is computed by several threads grouped in blocks. The threads within the same block run in the same multiprocessor and can therefore share the same resources, namely a fast shared memory and a local cache that can efficiently operate as a constants' memory. In order to extract full advantage of the microprocessors the threads are grouped in 32 simultaneous running threads (warp). The full power of a multiprocessor is not completely exploited if less than 32 threads are simultaneously available to run. Data dependencies or divergent control flows between the different

Fig. 4. Schedule of the operations to compute a BE according to Algorithm 1. In order to assure correctness, several synchronizations need to be inserted each time a thread needs data that was previously computed by any other thread. The values $\hat{\alpha}_j$ are computed by a single thread using (1), as a simple accumulation and shift of all the truncated contributions of each thread. The data sharing between all the running threads is assured by the device's shared memory.

threads in the same block can limit the number of threads ready to run and should therefore be avoided to maximize performance. In CRNS the user can decide the desired level of parallelism: (i) by setting the RNS channel width k so that the CRNS can derive the number of threads required to support the algorithm's dynamic range; (ii) by specifying the number of operations $N_{\text{ops. per block}}$ to be run within a block (same multiprocessor); and (iii) by specifying the number of blocks N_{blocks} . Hence, the user can run $N_{\text{ops. per block}} \times N_{\text{blocks}}$ instances of the same algorithm in the GPU for different input datasets. Note that some combinations of k , $N_{\text{ops. per block}}$ and N_{blocks} can exhaust the available resources in the GPU or in a multiprocessor (e.g., shared memory). For these configurations, the user is informed by the framework through the emission of error messages suggesting the user to modify the parameters.

CRNS is the first tool that automatically produces optimized PTX to compute the RNS-related arithmetic which translates to several advantages in terms of performance when compared to implementations obtained with high-level languages. An advantage of the PTX description is that registers can be more efficiently assigned and reused. The value for the channel width k can be any value but the target Nvidia GPU have a 32-bit datapath. Therefore, for any value $k \neq 32$, multiprecision arithmetic has to be implemented which requires the application of masks and operations with carry flags. Although the manipulation of carry flags is not possible using the usual C description supported by CUDA, the PTX instructions make this possible. On the other hand, the utilization of PTX with operations requiring the carry flags results in several data dependencies, e.g. by propagating the carry in a multiprecision addition. This can limit the efficiency of the program by exposing the deep pipeline in each of the several scalar processors inside a GPU's multiprocessor. Nevertheless, assigning a large number of threads (more channels or more operations per block) allows the scheduler to interleave the execution of threads each time a data dependency is identified among the running threads, mitigating the impact of the data dependencies in the performance.

The implementation of the addition, subtraction, and multiplication in each thread is straightforward as discussed in Section 2. Each thread can easily handle the carry values and use the built-in instructions for 32-bit addition, subtraction, multiplication (high and low part) and shift (left and right) for this purpose. Also, the RNS forward conversion can be easily implemented since it corresponds to a sequence of modular multiplications and additions for each thread (channel).

For the BE and reverse conversion, the implementation is more complex given that the threads need to cooperate. Figure 4 presents the scheduling of the operations on each thread for the computation of the BE taking into account the steps of Algorithm 1. In the related state-of-the-art, a slightly different method has been used to compute the BE which avoids the computation of $\hat{\alpha}_1$, reducing the size of the divergent section in the BE computation [Antão et al. 2011]. Nevertheless, the CRNS implements

Fig. 5. Schedule of the operations to compute the RNS reverse conversion with several threads. Each thread computes its contribution to each position of the final result, and then a single thread computes the correction offset $\hat{\alpha}_2 M_2$ and accumulates all the contributions.

the presented approach because it assures a lower dynamic range than the one in Antão et al. [2011].

Figure 5 presents the operations' schedule for the RNS reverse conversion resulting directly from the aforementioned reverse conversion method in (3). Note that Figure 5 presents the thread's contribution to the final result as k -bit words. In order to enhance the efficiency, these contributions are computed as 32-bit words instead, which is the width of the datapath of the target GPUs.

The following is a summary of the steps that need to be taken in order to use the CRNS with the GPU backend:

- modify the provided C template in order to set the desired inputs and interface it with any desired application;
- compile the template against the CUDA API library;
- run the obtained program in a machine containing a compatible Nvidia GPU. The PTX kernel file generated by the CRNS must be available given that it is JIT compiled to the GPU binary.

3.2. HDL Back-End

Two different approaches have been proposed to accelerate MA based on RNS with dedicated hardware [Schinianakis et al. 2009; Nozaki et al. 2001]. In Schinianakis et al. [2009] an accelerator for EC cryptography over a finite field $GF(p)$ with p prime was proposed. The architecture of the accelerator consists of three main components: (i) an RNS forward converter, (ii) a register file connected with buses to the channel's arithmetic units, and (iii) an RNS reverse converter followed by a projective to affine converter (EC coordinates conversion). The multiplication is performed by a Horner scheme and no optimizations are introduced regarding the reductions. This accelerator performs an initial conversion of the input data to RNS, uses the dedicated arithmetic units to implement the modular operations with the operands stored in the registers and, finally, converts the obtained results back to the binary weighted representation. An accelerator for the modular exponentiation, a main building block of the RSA cryptographic algorithm, was proposed in Nozaki et al. [2001]. The architecture of this accelerator is composed by as many multiply-accumulate units as the number of RNS channels, connected to each other in a ring shape. Each one of these units is fed by a Random Access Memory (RAM) that stores the input-dependent data and a Read-Only Memory (ROM) that stores the required constants. This architecture implements the RNS version of the Montgomery Modular Multiplication (MMM) by using a BE scheme such as the one presented in Appendix A.3. In order to handle the required conversions between bases, the conversion offset values $\hat{\alpha}_j$, similar to the ones in (1), are computed by a unit present in the logic that implements each channel. This architecture was upgraded to handle the EC point multiplication, a key operation

Fig. 6. Architecture supported by the proposed HDL back-end. It consists of several REs each one responsible for computing all the arithmetic for one of the RNS channels in basis B_1 and B_2 . The architecture also contains one input buffer and one output buffer, implemented as FIFO queues, that are the only components the designer has to interface with. There is a dedicated unit (' α accumulator') that computes the RNS conversion offset ($\hat{\alpha}_j$) required for the BE method as in (1). The control is assured by the framework and broadcast to all the REs.

of EC-based cryptography [Guillermin 2010]. Although these architectures allow the utilization of the RNS properties, they consist of dedicated implementations that can only support a particular application.

The CRNS HDL back-end supports the implementation of a fully functional and programmable accelerator specified with an HDL language able to compute the desired application. With this back-end, the production of the HDL specification with the CRNS can be used as the first step of a standard hardware design flow, preceding the synthesis of the HDL specification to the target technology. The architecture supported by the proposed framework depicted in Figure 6 is similar to the architectures in Nozaki et al. [2001] and Guillermin [2010] but with the following main improvements: (i) the Ring Elements (REs) contain a simple pipeline with several data forwarding paths instead of a regular pipeline designed and balanced to suit a specific application; (ii) the REs are microcontrolled with a sequence of instructions generated by the framework for the target application; (iii) the REs not only support the standard modular operations but also the forward and reverse conversions; and (iv) an accumulator and FIFO buffers are introduced to implement the required conversions, providing a simple interface. Figure 7 depicts the structure of the RE used in the architecture in Figure 6. The RE includes a dual port RAM, which is directly addressed by the broadcasted instructions and is responsible for storing input/output and temporary data, as well as the RNS constants. The control of the REs is directly included in the instructions that feed the hardware structure. Note that no multiprecision computation is directly instantiated within the RE given that the physical implementation of the arithmetic units is to be decided by the synthesis tool.

The reverse conversion is computed as specified in Figure 5 with the difference that: (i) the computation is performed by the REs instead of threads, (ii) the data is shared not recurring to a memory but traveling in the ring, and (iii) the final data accumulation is accomplished by a hardwired accumulator, that feeds the output buffer with the data already in the binary-weighted format.

Figure 8 presents a schematic of the computation of the BE in Algorithm 1 with the architecture in Figure 6. The shape of the architecture is mainly motivated by the need to improve the performance of the BE, which is the most demanding operation in the RNS-based arithmetic. This operation has $O(h_i^2)$ complexity ($O(h_i)$ in a single

Fig. 7. The RE: the basic processing element of the HDL back-end's architecture. The REs includes the required resources for computing all the arithmetic. These resources include a binary multiplier and adder, as well as two bitwise operation blocks which are responsible for multiplexing, complementing, and padding the operands. A pipeline is used, along with some data forwarding paths, to reduce the effective number of clock cycles per operation and increase the throughput.

Fig. 8. Schematic distribution of the BE workload presented in Algorithm 1 by the REs available in the architecture implemented with the HDL back-end. In this algorithm, each channel e needs to gather data from all the other channels j . The ring shape of the architecture optimizes the computation of $\sum_{j=1}^{h_1} q_{j,1} \lambda_{ej}$ enabling the forwarding of the values $q_{j,1}$ to the next RE after it is used by the current one. When a value $q_{j,1}$ finishes a complete tour around the ring, it is assured that all the REs have gathered all the information they need to proceed with the computation.

channel), with h_i the number of channels, due to the summation in the steps 2 and 5 of Algorithm 1.

Another, yet not less important, characteristic of the architecture is the scalability. It is straightforward to trade-off between the number of REs and the datapath's size inside the REs, which introduces extra flexibility in the design and allows the designer to explore the best configuration given the timing and resource constraints.

The following is a summary of the steps that need to be taken in order to use the CRNS with the HDL backend:

- instantiate the resulting HDL specification in a top-level design (more than one instance can be used to enhance throughput);
- synthesize the HDL specification for the target technology;

Table I. Experimental Setup Used in the Evaluation of the CRNS

Ref.	GPU	Host CPU	CUDA API
580GTX	Nvidia GeForce 580 GTX (Fermi arch 2.0)	Intel Core 2 Quad Q9550	version 3.2
285GTX	Nvidia GeForce 285 GTX (Tesla arch 1.3)	Intel Core i7 3930K	version 4.2
8800GTX	Nvidia GeForce 8800 GTX (Tesla arch 1.0)	AMD Opteron 170 Dual Core	version 3.2

(a) GPU Back-end

Ref.	Xilinx FPGA part	Synthesis and place and route tools
V4	Virtex 4 - xc4vxsx55ff1148-12	Synopsis Synplify - version E-2010.09-SP2
V5	Virtex 5 - xc5v1x220ff1760-2	Xilinx ISE tools - version 12.4

(b) HDL Back-end

—write the required inputs as k -bit words in the accelerator's input FIFO, stream the framework's generated microcode in the accelerator (one instruction per clock cycle), and retrieve the results as k -bit words from the output FIFO.

4. USING THE FRAMEWORK

Cryptography is probably the most prominent application that can be addressed with the proposed framework. Hence, in order to illustrate the utilization of the framework and to conduct an experimental assessment of its characteristics, two typical cryptographic operations are herein addressed: the modular exponentiation used in the RSA [Rivest et al. 1978] and the EC point multiplication [Antão et al. 2011]. The first development step with the proposed framework is the generation of an appropriate description of the target algorithm using the framework's programming language. In Figure 3 is presented the implementation in the framework's programming language of a 1024-bit modular exponentiation. Concerning the EC point multiplication, Appendix C presents a sample program for this application, which consists of an EC point multiplication over an underlying 224-bit finite field. Note that in Figure 3, some of the constants' middle part was omitted to simplify the representation. The implementation of the EC point multiplication is more complex given that additions, subtractions, multiplications, and conditions are required. The second development step is to use the framework to generate the implementation for the given program. In this step, the user has to select the back-end that the framework will use to generate the implementation. As discussed in the previous sections, the framework is extensible enough to support several devices. Only two back-ends are currently supported: (i) GPUs (Nvidia), through the generation of an optimized PTX description of the algorithm, and (ii) reconfigurable hardware, through the generation of a technology-independent HDL specification of a processor and the microcode required to program it. In the following subsections, results obtained for each of these back-ends are assessed and discussed. Table I summarizes the experimental setup used. For all tested configurations, the implementation files were obtained in less than 3 seconds. The following are some of the error messages that can be generated during this process (note that the program parsing errors are not presented in this list but are also emitted).

- POSTPROCESSING ERROR (58): Condition is depending on the inputs.* The input program has a dependency from the input data that was triggered at line 58 of the program.
- Device shared memory of 16KB is not enough to support the amount of operations specified (35KB).* The number of operations per block or the channel width may have to be changed. The amount of shared memory in the target GPU (16KB) is not enough to store all the required data (35KB).

Fig. 9. Latency and throughput results of the CRNS implementation using the GPU back-end (580GTX) for a single block and different numbers of operations per block. All configurations' results comprise the host to/from GPU data transfers. Due to the interleaved execution of the threads caused by the data dependencies in the multiprecision arithmetic, the latency figures for different OPB are almost overlapped. Therefore, the throughput is approximately proportional to the value OPB .

—It is not possible to assign a valid RNS basis operation with such a small channel size. The size of the channel set by the user is too small to find a basis or to comply with the mathematical conditions in the BE offset computation.

4.1. GPU Back-End

In this subsection the results for the GPU back-end are discussed. To use this back-end, the user has to provide along with the program (i) the channel width k , (ii) the number of blocks B to be used during the computation, and (iii) the number of operations per block OPB . With these three variables the tool is able to define all the levels of parallelism to be exploited in the GPU: (i) with larger k less threads are required to operate a single instance of the user's algorithm but higher precision arithmetic is required for each thread; (ii) B specifies how many multiprocessors of the GPU are to be used; and (iii) OPB specifies the number of parallel instances of the algorithm that are run in a single multiprocessor (block). The analysis herein presented focuses on the performance of the implementation of a single block, given that there is no cooperation between blocks. Therefore, in particular for the target algorithms which are not data intensive, the execution time for one block will be similar to the execution time of 2 blocks and the maximum throughput will be close to the throughput of a single block multiplied by the number of multiprocessors available in the GPU.

Figure 9 presents the experimental latency and throughput results for the two target algorithms for a single block and the experimental setup 580GTX. The modular exponentiation is referred to as RSA. Note that for $OPB = 20$ and $OPB = 12$ there are some channel width values that do not present results. This is due to the lack of enough resources, namely shared memory, for those configurations. From Figure 9(a) one can conclude that the latency is almost invariant for different values of OPB . This observation has a direct connection with the PTX implementation of the multiprecision arithmetic, which results in several data dependencies and, consequently,

Fig. 10. Latency results for the CRNS implementation using the HDL back-end. The minimum latency is a compromise between the number of REs and their operating frequency.

pipeline exposure. Given that the GPU has a fast scheduler and thread switcher, each time there is a data dependency which prevents the running thread from continuing, another thread is issued during the time the pipeline would be stalling. Therefore, for configurations with larger values of OPB , there will be more threads available to run, but this will have almost no impact in the latency, given that all operations in the block will be running interleaved, occupying the time slots the other threads would be stalling. The same effect can be observed in Figure 9(b). This effect allows one to conclude that the GPU implementation may not be as competitive in terms of latency as it is in terms of throughput. Due to the latency behavior properties, the values of throughput in Figures 9(c) and 9(d) are approximately proportional to OPB .

Concerning the behavior of the implementation performance with the channel width k in Figure 9, there are values k that result in local latency minimums (throughput maximums). These values are usually the values of k that are multiple of 32, the GPU's datapath size. E.g., in Figures 9(c) and 9(d) the most competitive configurations are obtained for $k = 32$ and $k = 64$, respectively. Nevertheless there are also other values with similar behavior. These values usually result in a more efficient exploitation of the computation power available by distributing the threads by the GPUs warps (the 32 threads that can run simultaneously) more efficiently.

4.2. HDL Back-End

In order to use the HDL back-end the user only needs to specify the channel width k along with the input program. After obtaining the resulting files the HDL can be synthesized to a given technology (see Table Ib for the tools used in the experiments). After the implementation is obtained the user can easily integrate the accelerator in any design by interfacing with the FIFO buffers and streaming the framework's generated microcode into the accelerator to execute the algorithm. Figure 10 presents latency results for the two addressed applications. In this case, the accelerator only performs a single operation at once, therefore the throughput T can be obtained from the latency L as $T = 1/L$. Concerning latency, low-width channels correspond to more channels, which results in more clock cycles to implement the BE (for the same dynamic range, the ring size increases while the channel width decreases), and also more clock cycles to reduce inside the RE (maximum size of $c_{e,i}$ in $m_{e,i} = 2^k - c_{e,i}$ also increases with the number of channels). For larger-width channels the number of cycles to compute the program will decrease but, since the complexity of the REs increase, the operating frequency decreases. Hence, as Figure 10(a) shows, the minimum latency is achieved at $k = 64$. In Figure 10(b) a similar behavior is observed and the latency tends to change its monotony for $k = 96$.

4.3. Discussion

In Sections 4.1 and 4.2, results for two typical operations that can be implemented with the CRNS are presented. In order to contextualize the obtained results, they

Table II. CRNS Best Results Summary for the Experimental Setup in Table I between All the Configurations and Comparison with the Reference Designs

	GPU Back-end			HDL Back-end		
	ref.	Latency [ms]	T.put [ops./s] ^a	ref.	Latency [ms]	T.put [ops./s] ^b
EC point mult.	[Antão et al. 2011]	12.0	1,252	[Guillermin 2010]	0.68 ^c	1470 ^c
	580GTX	12.9	2,208	V4	5.77	173.4
	(improvement)	(-1.08)	(+1.76)		(-8.48)	(-8.48)
	285GTX	63.43	151.6	V5	6.41	156.0
Modular exp.	8800GTX	65.2	146.9			
	[Szerwinski and Güneysu 2008]	144	67.75	[Nozaki et al. 2001]	4.2 ^d	238 ^d
	8800GTX	119	23.68	V4	5.31	188.4
	(improvement)	(+1.21)	(-2.86)		(-1.26)	(-1.26)
	285GTX	116	24.41	V5	3.54	282.3
	580GTX	17.9	703.2			

^aThe throughput is evaluated for a single multiprocessor.

^bThe HDL back-end only runs a single operation at once, thus the latency is the inverse of the throughput.

^cThe results were obtained for an Altera Stratix II technology and a 256-bit wide prime field.

^dThe results were obtained for a 0.25 μ m CMOS technology.

The improvement figures refer to the experimental setup with closer characteristics to the reference implementations.

are compared with related state-of-the-art implementations that share common characteristics with the implementations that can be developed with CRNS, i.e. are programmable solutions and/or employ RNS-based computation. For the experimental setup herein employed, Table II summarizes the results for the most competitive configurations in terms of channel width k and, for the GPU back-end, B and OPB , along with the aforementioned related state-of-the-art results. Note that none of the related state-of-the-art implementations computes the RNS forward and reverse conversions. In Antão et al. [2011] is presented a GPU implementation of the EC point multiplication using RNS and the BE method to compute the modular reduction. To the best of the authors' knowledge, this implementation provides the lowest latency and higher throughput known to date for GPU platforms. The authors were able to obtain results for the same experimental setup (see Table Ia) employed in the experimental evaluation herein presented. In Szerwinski and Güneysu [2008] the evaluation of several BE methods is accomplished and results for both an RNS and non-RNS implementation of the modular exponentiation are stated. The lowest latency of 144 ms is obtained with the RNS method but the highest throughput per multiprocessor is 67.75 ops/s, which is obtained with the non-RNS method. These results are stated for a Nvidia 8800 GTS GPU that corresponds to the first GPU supporting the Tesla architecture. The results for the 8800GTX experimental setup suggest an improvement of 21% in latency but lower throughput. Regarding Antão et al. [2011], the comparisons suggest a different conclusion with similar latency values but 76% improvement for the throughput. Given that CRNS requires storing the same amount of constants in the shared memory independently of the number of instances of the algorithm running in each multiprocessor, for devices with less shared memory resources the ratio of memory assigned to the constants is higher, and less memory is available to run more parallel instances and increase throughput. Therefore, the CRNS is more competitive for the more recent GPU architectures (see results for the implementations 285GTX and 580GTX in Table II).

The HDL-based implementations more than halve the latency of the GPU implementations but cannot supersede them in terms of throughput. Given that the dedicated hardware implementations in Nozaki et al. [2001] and Guillermin [2010] refer to different technologies than the ones herein addressed (see Section 3.2 for a detailed

description of these implementations), a fair direct comparison cannot be supported both in terms of latency and resources. Furthermore, the CRNS implementation is technology independent and it is not fine-tuned as the reference implementations. Therefore, the results in Table II pretend only to contextualize the obtained results. In order to complement this contextualization, the following presents a summary of the resource utilization of the CRNS implementation regarding Guillermin [2010], which also refers to an FPGA implementation: the CRNS implementation requires 12 Block RAMs (18 KB each), 64 DSPs, and 3435 Slices, 3%, 12%, and 13%, of the total available in the device, respectively; and the implementation in Guillermin [2010] requires 96 DSPs and 9177 ALM (the slice equivalent for Altera devices).

Concerning the two target devices, the CRNS is able to support the automatic and fast generation of the implementation. Furthermore, for the GPU, it also supersedes the performance of other implementations proposed in the literature.

5. CONCLUSIONS

This article proposes the CRNS, aiming to automate the design and implementation of MA-based algorithms. The proposed framework currently supports two different back-ends that produce optimized PTX code for GPUs and the HDL specification of a fully functional and technology-independent accelerator, respectively. The design is developed from a target algorithm specified using a high-level programming language. Furthermore, the proposed framework supports RNS arithmetic which introduces a step forward toward the efficient parallelization of the applications. The application of the RNS arithmetic is completely transparent to the designer, since the framework's tools handle all the computation required to implement a custom algorithm with RNS arithmetic. Furthermore, the implementation files are obtained in a few seconds overcoming all the issues in the designing and verification of a dedicated implementation. On top of that, the CRNS framework suggests superseding the throughput marks of the related state-of-the-art implementations by a factor of 1.76 for a Nvidia Fermi architecture.

Framework evaluation release. The current version of the framework is available for downloading and evaluation at <http://sips.inesc-id.pt/~sfan/prototypes/crns>.

APPENDIXES

A. RNS OPERATIONS

The main operations to be supported in a Modular Arithmetic (MA) algorithm are the addition/subtraction, multiplication and reduction as well as the conversion of Residue Number System (RNS) from/to binary. Nevertheless, these two last operations are far more complex and are thoroughly discussed in the following.

A.1. RNS Forward/Reverse Conversions

Forward conversion corresponds to the conversion of the input data to the RNS representation at the program's startup. The conversion of an input X is usually accomplished by computing for every channel e of the basis B_i : $x_{e,i} = X \bmod m_{e,i}$. Because, in general, $X \geq 2^k$, for the forward conversion the following should be computed for the channel e :

$$x_{e,i} = \left(\sum_{j=1}^{\lceil S_X/k \rceil} {}^kX_j 2^{k(j-1)} \right) \bmod m_{e,i} = \sum_{j=1}^{\lceil S_X/k \rceil} {}^kX_j \left(2^{k(j-1)} \bmod m_{e,i} \right) \bmod m_{e,i}, \quad (4)$$

where S_X is the size of X in bits and kX_j is the j -th word of the binary representation of X split in k -bit words. Considering (4), in order to obtain the RNS representation of X ,

the constants $2^{k(j-1)} \bmod m_{e,i}$ can be precomputed and stored in the accelerator and the conversion is accomplished with a few RNS channel modular multiply-and-accumulate operations.

For the reverse conversion the Chinese Remainder Theorem (CRT) can be used which states for the RNS representation of X in h_i channels e of a given basis B_i :

$$X = \sum_{e=1}^{h_i} \xi_{e,i} M_{e,i} - \alpha M_i, \quad \alpha < n, \quad \xi_{e,i} = \left\lfloor \frac{x_{e,i}}{M_{e,i}} \right\rfloor_{m_{e,i}}, \quad M_{e,i} = \frac{M_i}{m_{e,i}}. \quad (5)$$

In order to the conversion be successful, the value of α in (5) should be computed such that $0 \leq X < M$. The method proposed in Kawamura et al. [2000] involves a fixed point successive approximation approach. This method observes that for $X < M$, therefore $\frac{X}{M} < 1$, and (5) can be rewritten as

$$\sum_{e=1}^{h_i} \frac{\xi_{e,i}}{m_{e,i}} = \alpha + \frac{X}{M_i} \Rightarrow \alpha = \left\lfloor \sum_{e=1}^{h_i} \frac{\xi_{e,i}}{m_{e,i}} \right\rfloor. \quad (6)$$

Since (6) requires costly divisions by $m_{e,i}$, an approximation ($\hat{\alpha}$) is suggested

$$\hat{\alpha} = \left\lfloor \sum_{e=1}^{h_i} \frac{\text{trunc}_t(\xi_{e,i})}{2^k} + \beta \right\rfloor = \left\lfloor \sum_{e=1}^{h_i} \frac{\text{trunc}_t(\xi_{e,i})}{2^{k-t}} + \Phi \right\rfloor / 2^t, \quad (7)$$

where $\text{trunc}_t(\xi_{e,i})$ sets the $k - t$ least significant bits of $\xi_{e,i}$ to zero, with $t < k$, and $\Phi = \lfloor 2^t \beta \rfloor$. The parameter β (and consequently Φ) is a corrective term that should be carefully chosen such that $\alpha = \hat{\alpha}$. A set of inequalities that allow choosing good values for β were stated, supported on the maximum initial approximation errors.

$$\epsilon = \max_e \left(\frac{2^k - m_{e,i}}{2^k} \right), \quad \delta = \max_e \left(\frac{\xi_{e,i} - \text{trunc}(\xi_{e,i})}{m_{e,i}} \right) \quad (8)$$

In Kawamura et al. [2000], Theorems A.1 and A.2 are stated concerning the computation of the value $\hat{\alpha}$, its relationship with β , the number of channels h_i , and the errors stated in (8).

THEOREM A.1. *If $0 \leq h_i(\epsilon + \delta) \leq \beta < 1$ and $0 \leq X < (1 - \beta)M_i$, then $\alpha = \hat{\alpha}$.*

THEOREM A.2. *If $\beta = 0$, $0 \leq h_i(\epsilon + \delta) \leq 1$ and $0 \leq X \leq M_i$, then $\hat{\alpha} = \alpha$ or $\hat{\alpha} = \alpha - 1$.*

Theorem A.2 refers to the computation of a nonexact, although bounded, value of $\hat{\alpha}$ that allows a nonexact RNS to binary conversion, while Theorem A.1 refers to an exact conversion. Considering that the basis B_i one wants to convert from has $m_{\min}$ as the minimum value of $m_{e,i} = 2^k - c_{e,i}$, the value $\Delta = h_i(\epsilon + \delta)$ used in the definition of Theorems A.1 and A.2 can be obtained as

$$\Delta = h_i(\epsilon + \delta) = h_i \left(\frac{2^k - m_{\min}}{2^k} + \frac{2^{k-t} - 1}{m_{\min}} \right) = \frac{h_i}{m_{\min}} (2^k + 2^{k-t} - m_{\min} - 1). \quad (9)$$

Summarizing, Theorem A.1 allows obtaining suitable values of Φ in (7) such that $\alpha = \hat{\alpha}$, and Theorem A.2 bounds the maximum error of $\hat{\alpha}$ (+1) if Φ is set to zero, thus an extra term M_i may exist in (3).

Another challenge concerning the reverse conversion is the mapping of this conversion to the resources assigned to the channel arithmetic in order to avoid the utilization of a dedicated unit for this purpose. In other words, the challenge is to map the reverse conversion to the k -bit arithmetic used in each one of the RNS channels. Each channel e is able to compute the values $\xi_{e,i} = \lfloor x_{e,i} \rfloor_{M_{e,i}^{-1} \bmod m_{e,i}}$ and consequently a value of α can

be computed by accumulating the contributions of each channel by using the method in (7). Each channel can also contribute to the final value in (3) if the constants ${}^k(M_{e,i})_j$ such that $M_{e,i} = \sum_{j=1}^{S_{M_{e,i}}/k} {}^k(M_{e,i})_j 2^{kj}$ are available to each channel e , where $S_{M_{e,i}}$ is the size in bits of $M_{e,i}$. With these constants, the values $\xi_{e,i} {}^k(M_{e,i})_j$ can be accumulated obtaining k bits of the result at once. The contribution that depends on α has to be computed by one of the channels that should have available the constants ${}^k(-M_i)_j$ such that $-M_i = \sum_{j=1}^{S_{M_i}/k} {}^k(-M_i)_j 2^{kj}$. This channel can compute $\alpha {}^k(-M_2)_j$ as a contribution to the final accumulation.

A.2. RNS Addition/Subtraction and Multiplication

Addition, subtraction, and multiplication in an RNS channel e is straightforward and relies on the corresponding binary operations followed by the final reduction in the channel if the result is not in the range $[0, m_{e,i} - 1]$. For addition and subtraction, a further addition/subtraction with $m_{e,i}$ is accomplished for this reduction, as presented in Algorithm 2.

ALGORITHM 2: RNS channel addition/subtraction.

Require: $a_{e,i}$, $b_{e,i}$ and $m_{e,i}$.

Ensure: $r_{e,i} = a_{e,i} \pm b_{e,i} \bmod m_{e,i}$.

- 1: $r_{e,i} = a_{e,i} \pm b_{e,i}$;
 - 2: **if** $r_{e,i} \geq m_{e,i}$ **or** $r_{e,i} < 0$ **then** $r_{e,i} = r_{e,i} \mp m_{e,i}$;
 - 3: **return** $r_{e,i}$.
-

In the multiplication (see Algorithm 3) the reduction stage is more elaborated, since for a k -bit channel the value to be reduced is bounded by $r' < 2^{2k} - 1$. Hence, r' can be rewritten as $r' = {}^k r'_2 2^k + {}^k r'_1$ and since $2^k = m_{e,i} + c_{e,i}$ and $m_{e,i} \equiv 0 \bmod m_{e,i}$, therefore $r' \equiv {}^k r'_2 c_{e,i} + {}^k r'_1 \bmod m_{e,i}$.

ALGORITHM 3: RNS channel multiplication.

Require: $a_{e,i}$, $b_{e,i}$ and $m_{e,i}$.

Ensure: $r_{e,i} = a_{e,i} b_{e,i} \bmod m_{e,i}$.

- 1: $r' = a_{e,i} b_{e,i}$;
 - 2: **while** $r' \geq 2m_{e,i}$ **do** $r' = {}^k r'_2 c_{e,i} + {}^k r'_1$;
 - 3: $r_{e,i} = r'$;
 - 4: **if** $r_{e,i} > m_{e,i}$ **then** $r_{e,i} = r_{e,i} - m_{e,i}$;
 - 5: **return** $r_{e,i}$.
-

A.3. RNS Reduction

The reduction modulo N as herein discussed is usually introduced as part of a modular multiplication algorithm known as Montgomery Modular Multiplication (MMM) [Montgomery 1985]. Addressing this problem, an RNS variation of the MMM that relies in a Basis Extension (BE) method [Bajard et al. 2001] has been proposed.

In RNS, reduction modulo the dynamic range M_1 (it must comply with $\gcd(M_1, N) = 1$) is an easy task since it relies on the modulo operation in each one of the RNS channels of the basis B_1 . In order to apply this method, the input data should be represented in the alternative Montgomery Domain (MD) (see Appendix B). The correspondence between the MD and the data's Original Domain (OD) is bijective, hence any operation performed in the MD has a correspondence in the OD. In order to define the MD, the

modulo N arithmetic and the RNS dynamic range M_1 should be defined. For an element A in the OD its correspondence in the MD is given by ${}_MA = A(M_1 \bmod N) \bmod N$. For the addition and subtraction of two elements in the MD it is easy to see a direct correspondence with the OD since ${}_MA + {}_MB = (A+B)(M_1 \bmod N)$. For the multiplication of two elements in the MD, the result is given by ${}_MR = {}_MA{}_MB = (AB)(M_1^2 \bmod N)$, which includes an extra factor $M_1 \bmod N$ in the result. This factor is corrected by assuring that the multiplication is followed by a reduction operation that computes ${}_MR = {}_MRM_1^{-1} \bmod N$. Also, in order to keep a unified reduction method in RNS, each time the reduction is performed after an addition or subtraction, the input value to be reduced should be multiplied by $M_1 \bmod N$. Hence, for a value R to be reduced in the OD, the input of the reduction in the MD has the form $R(M_1^2 \bmod N)$. The reduction result ${}_MU$ is obtained by computing

$${}_MU = ({}_MR + QN)/M_1. \quad (10)$$

Note that since $QN \equiv 0 \bmod N$, (10) corresponds to ${}_MU \equiv RM_1 \bmod N$ as desired. The purpose of the value QN is to assure that $({}_MR + QN)$ is a multiple of M_1 , which allows the division by M_1 to introduce no approximation error in the result. Assuring that $({}_MR + QN)$ is a multiple of M_1 is equivalent to $({}_MR + QN) \equiv 0 \bmod M_1$, which is the same as computing ${}_Mr_{e,1} + q_{e,1}n_{e,1} \equiv 0 \bmod m_{e,1}$ for all the RNS channels e of B_1 . With the aforementioned property, Q is computed in the RNS channels as $q_{e,1} = {}_Mr_{e,1}n_{e,1}^{-1} \bmod m_{e,1}$.

One of the drawbacks of this reduction algorithm is that M_1 cannot be represented in the moduli-set with dynamic range M_1 . This means that an extra basis B_2 with dynamic range M_2 higher than M_1 , with $\gcd(M_2, M_1) = 1$ and $\gcd(M_2, N) = 1$, is required to compute (10). This is the reason for the BE method's designation, since the original basis is extended to allow computing (10). The use of the extra basis B_2 requires the data to be reduced (${}_MR$) to be also represented in B_2 and the value of Q to be converted from B_1 to B_2 . This conversion is performed with the method presented in Appendix A.1, with the main difference that for each channel e the operations are computed modulo $m_{e,2}$. After computing the result ${}_MU$ in B_2 , it has to be converted to B_1 and the reduction algorithm is complete. Algorithm 4 presents in detail the steps of the described reduction algorithm for each RNS channel.

ALGORITHM 4: RNS reduction.

Require: ${}_Mr_{e,1}, {}_Mr_{e,2}$.

Ensure: ${}_Mu_{e,1} = (RM_1 \bmod N)_{e,1}$ and ${}_Mu_{e,2} = (RM_1 \bmod N)_{e,2}$.

- 1: In ch. e , 1: $q_{e,1} = \left| {}_Mr_{e,1}n_{e,1}^{-1} \right|_{m_{e,1}}$
 - 2: In ch. e , 1: $\xi_{e,1} = \left| q_{e,1} \left| M_{e,1}^{-1} \right|_{m_{e,1}} \right|_{m_{e,1}}$
 - 3: In ch. e , 2: $q_{e,2} = \left| \sum_{j=1}^{h_1} \xi_{j,1} \left| M_{j,1} \right|_{m_{j,1}} - \hat{\alpha}_1 M_{1e,2} \right|_{m_{e,2}}$
 - 4: In ch. e , 2: ${}_Mu_{e,2} = \left| ({}_Mr_{e,2} + q_{e,2}n_{e,2})M_1^{-1} \right|_{m_{e,2}}$
 - 5: In ch. e , 2: $\xi_{e,2} = \left| {}_Mu_{e,2} \left| M_{e,2}^{-1} \right|_{m_{e,2}} \right|_{m_{e,2}}$
 - 6: In ch. e , 1: ${}_Mu_{e,1} = \left| \sum_{j=1}^{h_2} \xi_{j,2} \left| M_{j,2} \right|_{m_{j,2}} - \hat{\alpha}_2 M_2 \right|_{m_{e,1}}$
 - 7: **return** ${}_Mu_{e,1}, {}_Mu_{e,2}$.
-

In Algorithm 4 two conversions are performed using (3): from B_1 to B_2 with $\hat{\alpha}_1$ and from B_2 to B_1 with $\hat{\alpha}_2$. For the former conversion, the value to be converted is Q , which

is in the range $[0, M_1 - 1]$, whereas for the latter conversion the value converted is ${}_M U$ in (10). Given the range of Q , the conditions of Theorem A.1 cannot be verified, thus $\hat{\alpha}_1$ should be obtained from (7) in the conditions of Theorem A.2 instead, with the associated error. Therefore, for the computation of $\hat{\alpha}_1$ the value Φ in (7) should be set to zero and the minimum value of t such that $0 \leq \Delta \leq 1$ should be used (Theorem A.2). Note that the minimum possible value of t results in the operands' size in the accumulation in (7) to be minimum as well. For ${}_M U$ it is possible to obtain $\hat{\alpha}_2$ under the conditions of Theorem A.1, i.e., a value β such that $0 \leq {}_M U < (1 - \beta)M_2$ exists. Therefore, the minimum value of t such that $\max({}_M U) < (1 - \Delta)M_2$ should be computed and $\hat{\alpha}_2$ obtained from (7) using $\Phi = \lfloor 2^t \Delta \rfloor$.

Algorithm 4 can be further optimized by merging constants so as to reduce the required precomputed constants and temporary results. Algorithm 1 is an optimized version of Algorithm 4 that comprises the merging of the constants as: $\lambda_{1e} = | - n_{e,1}^{-1} M_{e,1}^{-1} |_{m_{e,1}}$, $\lambda_{2ej} = | - M_{j,1} M_1^{-1} |_{m_{e,2}}$, $\lambda_{3e} = | - M_1 n_{e,2} |_{m_{e,2}}$, $\lambda_{4e} = | M_1^{-1} |_{m_{e,2}}$, $\lambda_{5e} = | M_{e,2}^{-1} |_{m_{e,2}}$, $\lambda_{6ej} = | - M_{j,2} M_2^{-1} |_{m_{e,1}}$, and $\lambda_{7e} = | - M_2 |_{m_{e,1}}$.

B. MONTGOMERY DOMAINS

The MDs are defined as domains where the operands involved in MA modulo N_i can be alternatively represented to enhance the efficiency of these operations in RNS, namely the reduction. The input variables of an algorithm are said to be in their OD, thus conversions between MDs and the OD are required. Considering a variable X in its OD represented in an RNS basis B_1 , the corresponding variable ${}_M X$ in the MD associated to the modulus N_i is ${}_M X = X(M_1 \bmod N_i) \bmod N_i$. The inverse conversion of domain can be performed as $X = {}_M X(M_1^{-1} \bmod N_i) \bmod N_i$. There is an isomorphism between any MD and the OD, therefore operations in the MD have a direct relationship with the operations in the OD.

- For efficiency reasons, the reduction $U = R \bmod N$ is usually computed recurring to a method that reduces the results but at the same time inserts a term $M_1^{-1} \bmod N_i$ in the result (see Appendix A.3). Therefore, in order to assure that the output of this operation is in the required MD the input ${}_M R$ of the reduction is replaced by ${}_M R(M_1 \bmod N_i)$ so that the output is ${}_M R(M_1 \bmod N_i)(M_1^{-1} \bmod N_i) \bmod N_i = {}_M R \bmod N_i$.
- For the addition $R = A + B \bmod N_i$ the correspondence is direct, i.e., ${}_M A + {}_M B \bmod N_i = (A + B)(M_1 \bmod N_i) \bmod N_i = {}_M R$. For the subtraction the rationale is the same.
- The multiplication $R = AB \bmod N_i$ in the MD results in $AB(M_1^2 \bmod N_i) \bmod N_i = {}_M R(M_1 \bmod N_i)$. Therefore, every time a multiplication is performed the factor $M_1 \bmod N_i$ should be corrected. This correction can be performed using the aforementioned reduction operation in the MD without using the factor $M_1^{-1} \bmod N_i$. The utilization of the reduction to correct this factor is also useful to keep the required dynamic ranges as low as possible. The correction can be avoided if one of the operands ${}_M A$ or ${}_M B$ is used in the OD, which would result in $AB(M_1 \bmod N_i) = {}_M R$.

Using the aforementioned operations, it is possible to operate the conversions between OD and MDs. The conversion from OD to MD is accomplished with a multiplication of the operand to be converted X by $M_1^2 \bmod N_i$ followed by a reduction as

$${}_M X = X(M_1^2 \bmod N_i)(M_1^{-1} \bmod N_i) \bmod N_i = X(M_1 \bmod N_i) \bmod N_i, \quad (11)$$

while the conversion from an MD to OD is obtained with a single reduction.

$$X = {}_M X(M_1^{-1} \bmod N_i) \bmod N_i = X(M_1 \bmod N_i)(M_1^{-1} \bmod N_i) \bmod N_i \quad (12)$$

<pre> 01. (Xp,Zp) = (int224 point) 02. { 03. int p = 26959946 ... 66298881; 04. int c = 0x5b056c ... 218291fb; 05. int b = 0xb4050a ... 2355ffb4; 06. int scalar = 26959946 ... 22368062; 07. 08. /* mask to use with the scalar */ 09. int M = 0x400000 ... 00000000; 10. 11. /* declare the variables */ 12. int xg,A,B,C,D,E,F,G,H,Xq,Zq; 13. 14. /* initialization of the computation */ 15. xg = point % p; Xp = xg; 16. Zp = 1; A = xg * xg % p; B = b * xg % p; 17. C = A + 3; C = C * C % p; A = xg * A % p; 18. Xq = C - 8*B; Zq = 4*(A - 3*xg + b); 19. </pre>	<pre> 20. while(M) 21. { 22. int M2 = M & scalar; 23. 24. if(M2) //scalar[i] == 1 25. { 26. A = Xq*Zp % p; B = Xp*Zq % p; C = Xq*Xp % p; 27. D = Zq*Zp % p; E = Xq*Xq % p; F = Zq*Zq % p; 28. H = b*Zq % p; 29. Zp = A-B; Xp = A+B; C = C+3*D; A = E+3*F; 30. D = D*Xp % p; Xp = C*C % p; Zp = Zp*Zp % p; 31. A = A*A % p; B = F*H % p; E = E*Xq % p; 32. F = Xq*F % p; G = E+B-3*F; 33. D = b*D % p; Zp = xg*Zp % p; B = Xq*B % p; 34. F = Zq*G % p; 35. Xp = Xp-4*D; Xq = A-8*B; Zq = 4*F; 36. } 37. else {...} //scalar[i] == 0 38. M = M >> 1; 39. } 40. } </pre>
--	---

Fig. 11. EC point multiplication algorithm (Montgomery Ladder) described using the proposed framework supported language. In this program point is an EC point's coordinate and scalar is the integer to multiply the point with.

Note that the MDs depend on N_i , hence there is an MD for each N_i used in a given algorithm. Since the operands need to be in the same MD for the operations to be correct, conversions between MDs may also be required, which can be accomplished with a conversion from the current MD to OD followed by a conversion from the OD to the required MD.

C. ELLIPTIC CURVE POINT MULTIPLICATION PROGRAM

In Figure 11 is presented an algorithm for computing EC point multiplication often referred to as Montgomery Ladder [Antão et al. 2011]. This algorithm receives the point information as a single coordinate (point) and returns the resulting projective coordinates (xp and zp). The scalar to multiply the point with is specified as a constant and browsed during the algorithm with the help of a mask. One of the characteristics of this algorithm is that its duration (number of operations) does not depend on the scalar, being therefore resistant against time attacks, because the operations being computed in the if scope are similar to the ones computed in the else scope.

REFERENCES

- ANTÃO, S., BAJARD, J.-C., AND SOUSA, L. 2011. RNS based elliptic curve point multiplication for massive parallel architectures. *The Comput. J. Oxford J.* 55, 5, 629–647.
- ANTÃO, S., CHAVES, R., AND SOUSA, L. 2009. Compact and flexible microcoded elliptic curve processor for re-configurable devices. In *Proceeding of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*. IEEE, 193–200.
- BAJARD, J.-C., DIDIER, L.-S., AND KORNERUP, P. 2001. Modular multiplication and base extensions in residue number systems. In *Proceeding of the IEEE Symposium on Computer Arithmetic (ARITH)*. IEEE, 59–65.
- FLEX. 2012. Flex: The fast lexical analyzer. <http://flex.sourceforge.net/>
- GNU - BISON. 2012. Bison - GNU parser generator. <http://www.gnu.org/software/bison/>
- GNU - GMP. 2012. The GNU multiple precision arithmetic library. <http://gmplib.org/>
- GUILLERMIN, N. 2010. A high speed coprocessor for elliptic curve scalar multiplications over Fp. In *Proceeding of the Conference Cryptographic Hardware and Embedded Systems (CHES'10)*, S. Mangard and F.-X. Standaert, Eds. Springer, 48–64.
- KAWAMURA, S., KOIKE, M., SANO, F., AND SHIMBO, A. 2000. Cox-Rower architecture for fast parallel Montgomery multiplication. In *Proceeding of the Conference on EUROCRYPT00*, B. Preneel, Ed. Springer, 523–538.
- LEE, D.-U., GAFFAR, A. A., CHEUNG, R. C. C., MENCER, O., LUK, W., AND CONSTANTINIDES, G. A. 2006. Accuracy-Guaranteed bit-width optimization. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 25, 10, 1990–2000.

- LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* 28, 2, 39–55.
- LLVM. 2012. The LLVM compiler infrastructure. <http://llvm.org/>
- MAHLKE, S., RAVINDRAN, R., SCHLANSKER, M., SCHREIBER, R., AND SHERWOOD, T. 2001. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *IEEE Trans. Comput.-Aid. Des. Integ. Circ. Syst.* 20, 11, 1355–1371.
- MOHAN, P. V. A. 2002. *Residue Number Systems: Algorithms and Architectures*. Kluwer Academic Publishers, Norwell, MA.
- MONTGOMERY, P. L. 1985. Modular multiplication without trial division. *Math. Comput.* 44, 170, 519–521.
- NOZAKI, H., MOTOYAMA, M., SHIMBO, A., AND KAWAMURA, S. 2001. Implementation of RSA algorithm based on RNS Montgomery multiplication. In *Proceeding of the Conference on Cryptographic Hardware and Embedded Systems (CHES'01)*, C. K. Koç, D. Naccache, and C. Paar, Eds. Springer, 364–376.
- RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM* 21, 2, 120–126.
- SCHINIANKIS, D. M., FOURNARIS, A. P., MICHAIL, H. E., KAKAROUNTAS, A. P., AND STOURAITIS, T. 2009. An RNS implementation of an F_p elliptic curve point multiplier. *IEEE Trans. Circ. Syst.* 56, 6, 1202–1213.
- SZERWINSKI, R. AND GÜNEYSU, T. 2008. Exploiting the power of GPUs for asymmetric cryptography. In *Proceeding of the Conference on Cryptographic Hardware and Embedded Systems (CHES'08)*, E. Oswald and P. Rohatgi, Eds. Springer, 79–99.

Received June 2012; revised September 2012; accepted October 2012