High-Throughput Hardware Architecture for the SWIFFT / SWIFFTX Hash Functions

Tamás Györfi[1,2], Octavian Creţ[2], Guillaume Hanrot[3], and Nicolas Brisebarre[3]

[1] National Instruments Romania, Cluj-Napoca branch,
B-dul Corneliu Coposu, nr. 167A, et.I,
Cluj Napoca, Romania
tamas.gyorfi@ni.com

[2] Technical University of Cluj-Napoca - Romania, Computer Science Department,
26-28 George Bariiu street,
400027 Cluj-Napoca, Romania
Octavian.Cret@cs.utcluj.ro

[3] LIP/AriC, (CNRS – ENS Lyon – INRIA – UCBL), ENS Lyon,
46, allée d'Italie,
F-69364 Lyon Cedex 07, France
{Guillaume.Hanrot,Nicolas.Brisebarre}@ens-lyon.fr

Abstract. Introduced in 1996 and greatly developed over the last few years, Lattice-based cryptography offers a whole set of primitives with nice features, including provable security and asymptotic efficiency. Going from "asymptotic" to "real-world" efficiency seems important as the set of available primitives increases in size and functionality. In this present paper, we explore the improvements that can be obtained through the use of an FPGA architecture for implementing an ideal-lattice based cryptographic primitive. We chose to target two of the simplest, yet powerful and useful, lattice-based primitives, namely the SWIFFT and SWIFFTX primitives. Apart from being simple, those are also of central use for future primitives as Lyubashevsky's lattice-based signatures. We present a high-throughput FPGA architecture for the SWIFFT and SWIFFTX primitives. One of the main features of this implementation is an efficient implementation of a variant of the Fast Fourier Transform of order 64 on $\mathbb{Z}_{257}$. On a Virtex-5 LX110T FPGA, we are able to hash 0.6GB/s, which shows a ca. 16× speedup compared to SIMD implementations of the literature. We feel that this demonstrates the revelance of FPGA as a target architecture for the implementation of ideal-lattice based primitives.

Keywords: Lattice-based cryptography, Provably secure, Hardware accelerator, FPGA, FFT, Hash functions

1 Introduction

Lattice-based cryptography (see [25] for a beautiful survey) has been developing at a quick pace over the last fifteen years since the seminal paper by Ajtai [2]. Its main attractive features, among others (including no known quantum attack at the time this paper is written) are probably rigorous asymptotic security analyses and asymptotic efficiency.

Such an analysis has been obtained through the identification of two specific and versatile problems, SIS [3] and LWE [30,31], such that the *average* instance of those problems is at least as hard as the *worst-case* instance of a lattice problem. This lattice problem is actually a slight relaxation of the problem of finding a shortest non-zero vector of a lattice in the ℓ^2 norm, known to be NP-hard under randomized reductions. The cryptographer then only has to prove the equivalence of breaking the cryptosystem and solving a random instance of either SIS or LWE to get a strong asymptotic security guarantee.

As for efficiency, it should be noted that the basic operation in lattice-based cryptography is a matrix-vector product, which has quadratic complexity in the size of the vector. This complexity was actually lowered to quasi-linear through the introduction of structured matrices, idea which originated in the well-known NTRU cryptosystem [16] and was revived by Micciancio, who introduced matrices with a specific block-Toeplitz structure, namely *negacyclic structure*[1]. As the product of such a matrix by a vector amounts to computing a product of two polynomials modulo $X^n + 1$, one can get a quasi-linear computation time via the use of an FFT-type algorithm for multiplying polynomials. This has given rise to the definition and study of the hardness of structured variants of the LWE [23] and SIS [21,27] problems, which have proven to be as versatile as their purely linear counterparts and to enjoy similar security reductions, while offering cryptographic primitives with asymptotic quasi-linear complexity.

However, envisioning the deployment of lattice-based cryptography requires a practical, not only asymptotic, assessment of its parameters in order to reach a good efficiency/security compromise. By using the explicit reductions from the cryptosystems to classical lattice problems such as SIVP_γ (a variant of the Shortest nonzero Vector Problem, see e.g. [25]), one can derive rather precise estimates for security of lattice-based primitives, using the significant body of work performed on lattice basis reduction and enumeration of short vectors. Such studies of lattice problems and the impact they have on parameters choice for lattice-based cryptography have been made on a variety of platforms, see e.g. for CPUs [5, 8, 11, 19, 25], for GPUs [14, 15, 18] and for FPGA [9]. However, the assessment of the efficiency of (ideal-) lattice-based primitives does not seem to have given rise to much interest, except in software (CPU), though significant activity has been observed over the last months concerning homomorphic encryption, see e.g. [12, 26, 32]. In particular, this raises an important question on what various architectures (CPU, GPU, FPGA) can bring (advantages and drawbacks) as far as implementation of ideal-lattice-based primitives are concerned.

The main purpose of this paper is to study the relevance of FPGA implementation for an ideal-lattice-based cryptography primitive, and to compare it with highly optimized software implementation. In order to do this, we describe a high-throughput architecture for the simplest, though powerful and useful, of the lattice-based primitives, namely the SWIFFT [22] hash function and its improved version SWIFFTX [4]. The former enjoys a *security proof* for its collision-resistance properties (which is both desirable and highly unusual for a hash function), whereas the latter is an extension of the former, adding some scrambling steps to avoid undesirable linearity properties. Further, SWIFFT is used in other important cryptographic primitives such as Lyubashevsky's

[1] A matrix is said to be negacyclic if its coefficients are constant along each diagonal, up to a sign change when the diagonal wraps around

signature algorithm[2] [20]. The present work can thus be considered as a first step towards the implementation of more advanced lattice-based primitives. Our major contributions to the high throughput hardware implementation of the SWIFFT and SWIFFTX hash functions can be summarized as follows:

– Pipelined implementation of the modular Fast Fourier Transform over $\mathbb{Z}_p$, $p = 2^k + 1$, using the diminished-one number system;
– Packing the SWIFFT function descriptor constant coefficient multipliers into BlockRAM primitives;
– DSP overclocking for hardware reuse;
– Pipelined implementation of the ConvertToBytes function.

The paper is organized as follows: Section 2 presents an overview of the SWIFFT hash function introduced in [22], and briefly argues the choice to stick to the original SWIFFT parameters. Section 3 describes our high-throughput hardware implementation for the modular FFT. Though we restrict to the original SWIFFT parameters, it must be pointed that the FFT is a basic building block for all ideal-lattice based cryptography. Section 4 shows the proposed architecture for the SWIFFT hash function. Section 5 presents the hardware architecture for the SWIFFTX hash function which uses SWIFFT as its basic building block. Experimental results are shown in Section 6, while concluding remarks and future work are presented in Section 7.

2 The SWIFFT hash function

2.1 Description of SWIFFT

Let n be a power of 2, p a prime number such that $2n|p-1$, and R be the ring $\mathbb{Z}_p[\alpha]/(\alpha^n+1)$, which can be identified to the set of polynomials over $\mathbb{Z}_p$ with degree $< n$. Let m be an integer such that $m \geq \log p$. The SWIFFT hash function with parameters n, m, p takes as input m elements of R, and computes a linear combination of those with prescribed coefficients $\mathbf{A} = (A_0, \ldots, A_{m-1}) \in R^m$, namely

$$\mathrm{SWIFFT}_{\mathbf{A}}(x_0, \ldots, x_{m-1}) = \sum_{i=0}^{m-1} A_i \times x_i, \qquad (1)$$

where, in order to hash a binary string, the input message μ is cut into chunks of mn bits, which are in turn interpreted as the binary polynomials

$$(x_0, \ldots, x_{n-1}) = (\mu_0+\mu_1\alpha+\cdots+\mu_{n-1}\alpha^{n-1}, \mu_n+\cdots+\mu_{2n-1}\alpha^{n-1}, \ldots, \mu_{(m-1)n}+\cdots+\mu_{mn-1}\alpha^{n-1}) \in R^m.$$

In particular, SWIFFT hashes mn bits to $n\lceil\log p\rceil$ bits.

[2] Note that though Lyubashevsky does not directly point SWIFFT, one of the basic operations of his signature algorithm amounts to compute $A \cdot y \mod q$, which is the same if A has a suitable structure.

2.2 FFT aspects of SWIFFT

As pointed in [22], the polynomial products in Equation (1) can be efficiently computed using (part of) a $2n$-order Discrete Fourier Transform (DFT) performed over $\mathbb{Z}_p$ (sometimes referred to as *Number Theoretic Transform* (NTT), see eg. [29]). Let ω be a primitive $2n$-th root of unity lying in some extension field $\mathbb{F}$ of $\mathbb{Z}_p$. Define $\mathrm{ODFT}_{2n,\omega}(P)$ by

$$\mathrm{ODFT}_{2n,\omega}\left(\sum_{i=0}^{n-1} u_i X^i\right) = \left(\sum_{j=0}^{n-1} \omega^{(2k+1)j} u_j\right)_{0\le k\le n-1} = \left(\sum_{j=0}^{n-1} \left(\omega^2\right)^{kj} \left(\omega^j u_j\right)\right)_{0\le k\le n-1}.$$

As suggested by the last equality, this modified DFT can be computed by the classical FFT algorithm [6], with order n and primitive root ω^2, applied to the polynomial $\sum_{i=0}^{n-1} \omega^i u_i X^i$. This requires a quasi-linear number of operations in the field $\mathbb{F}$. Further, ODFT is suited to computing polynomial products modulo $X^n + 1$ (our goal here), since it is readily checked that $\mathrm{ODFT}_{2n,\omega}(P \cdot Q \bmod (X^n+1)) = \mathrm{ODFT}_{2n,\omega}(P) \star \mathrm{ODFT}_{2n,\omega}(Q)$, where $\star$ stands for the componentwise multiplication of vectors.

The setting where $p = 1 \bmod 2n$ is especially favourable since in that case $\mathbb{Z}_p$ contains a $2n$-th primitive root of unity ω, so that we have $\mathbb{F} = \mathbb{Z}_p$. All the underlying FFT computation is thus performed by arithmetic over $\mathbb{Z}_p$. Further note that such moduli (powers of 2 plus one – recall that n is a power of 2) are also well-suited to efficient implementation of modular reduction.

A slight change of definition of the SWIFFT function allows for further optimizations, see [22]. Indeed, as the multipliers A_i are fixed in advance, it is possible to store their Fourier representation instead of evaluating it[3]. Also, since ODFT is a bijection between $\mathbb{Z}_p^n$ and polynomials over $\mathbb{Z}_p$ of degree $< n$, the hash value can be represented by the value $\mathrm{ODFT}(\mathrm{SWIFFT}(u_0, \ldots, u_{n-1}))$ (in other words, in the Fourier domain) and there is no need to compute the inverse FFT; in the sequel, we shall denote this output FFT vector by $(Z_0, \ldots, Z_{n-1})$.

Based on the above observations, the algorithm for computing the hash value can be formulated as follows:

```
Input  : String to be hashed, seen as an element of $\{0,1\}^{n\times m}$.
Output : The hash value $(Z_0, ..., Z_{n-1}) \in \mathbb{Z}_p^n$
'Diffusion' step:
    Multiply the $i^{th}$ row of the input matrix by $\omega^i$;
    Apply the ODFT over each column of the resulting matrix:
        $(Y_{0,j}, ..., Y_{n-1,j}) = ODFT(\omega^0 \cdot x_{0,j}, ..., \omega^{n-1} \cdot x_{n-1,j})$.
'Confusion' step:
    Multiply the $i^{th}$ column by the element $A_{i,j}$;
    Compute the sum of the elements on a row:
        $Z_i = \sum_{j=0}^{m-1} A_{i,j} \times Y_{i,j}$.
```

Algorithm 1: The SWIFFT hash function algorithm

[3] or, instead, to choose the Fourier transform $\hat{A}_i$ uniformly at random instead of choosing A_i, both giving rise to the same distribution.

2.3 Parameter choice for FFT/SWIFFT

Based on security as well as performance considerations, the authors of [22] propose the following concrete parameters for the hash function: $n = 64, m = 16, p = 257$. Indeed, as was previously pointed, since $p = 1 \bmod 2n$, for these parameter values, one can use $\omega = 42 \in \mathbb{Z}_p$, and the function maps a binary input of length $m \times n = 1024$ bits to an output vector in the range $\mathbb{Z}_p^n$ that can be represented on 520 bits.

The original paper and the survey [25] argue that those parameters guarantee strong security (the latter mentionning a 100-bit equivalent for the symmetric security corresponding to enumeration attacks). Recent work by Buchmann and Lindner [5] argues that those parameters only provide roughly 68 bit of symmetric security. However, this security analysis accounts for the resistance to pseudo-collisions (roughly speaking, collisions where the x_i are "small" but not restricted to $\{0, 1\}$), whereas the security level for resistance to real collisions seems to remain higher than 100 bits. For this reason, but also because we wanted to stick to the original parameters of SWIFFT in order to be able to plug the latter into SWIFFTX, we stick to those parameters values in the sequel.

3 Hardware architecture for the FFT

The performance of the modular FFT depends on the efficient implementation of the basic modular arithmetic operators such as addition and multiplication in $\mathbb{Z}_p$. This section describes the proposed implementation of the modular arithmetic operators, and, based on them, presents a high-throughput design for the FFT.

3.1 Modular Arithmetic Operators Implementation

All arithmetic operations involved in the transform are performed over the $\mathbb{Z}_p$ field. We shall use here the fact that the standard value of p for SWIFFT, 257, is equal to $2^8 + 1$. Note that this is a strong requirement, as conjecturally, only finitely many (and probably very few) prime numbers of that form (the so-called Fermat primes) do exist.

As proposed in [1], an efficient way to represent numbers modulo $p = 2^k + 1$ is to apply the *diminished-one number system*, where a non-zero number A is represented as $A_{\text{dim}} := A - 1$ on $\log_2(p)$ bits, while number zero is indicated by a special flag $A.z$. We will denote the tuple $(A_{\text{dim}}, A.z)$ by A'.

Modular Addition and Subtraction Modular addition in this system can be carried out as follows [1]:

$$S_{\text{dim}} = (A_{\text{dim}} + B_{\text{dim}} + 1)[2^k + 1] = (A_{\text{dim}} + B_{\text{dim}} + \overline{c_{out}})[2^k], \qquad (2)$$

where S_{dim} is the diminished-one representation of the sum $A + B$ while c_{out} represents the carry-out bit of the addition $A_{\text{dim}} + B_{\text{dim}}$. Besides implementing Equation (2), the modular adder must have a zero-detection circuit that asserts the zero flag. The design of the modular adder is shown in Figure 1.

Modular subtraction can be carried out by adding the modular complement of B_{dim} which is equivalent with its ones' complement representation.

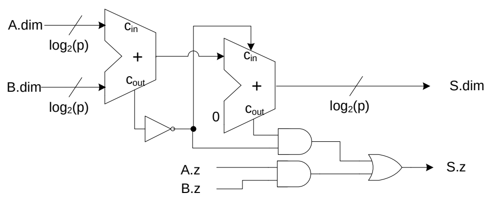

Fig. 1. Architecture of the modular adder unit based on diminished-one representation

Modular Multiplication The FFT requires the evaluation of polynomials at the n roots of unity, i.e. $\omega^0, \omega^2, \ldots, \omega^{2n-2}$. Besides additions and subtractions, this evaluation requires multiplication of the polynomial coefficients by the roots of unity at a certain power that can be considered as constants ($\omega = 42$ for our parameter choice).

We considered several modulo $2^k + 1$ multiplier architectures based on the techniques proposed in [7] and [35]. These techniques can be divided into four classes as follows:

1. Multiplication by means of lookup tables, where the operands form the address of the lookup table and the content of the table stores the product.
2. Quarter-squared method that uses lookup tables for evaluating $(A/2)^2$. Then, the product of two arbitrary numbers is computed based on the equation:
$$(A \times B)[2^k + 1] = \left(\left(\frac{A+B}{2}\right)^2 - \left(\frac{A-B}{2}\right)^2\right)[2^k + 1].$$
3. Multiplication by $k \times k$ bits multiplier using the following equation:
$$(A \times B)[2^k + 1] = ((A \times B) \mod 2^k - (A \times B) \text{ div } 2^k)[2^k + 1]. \tag{3}$$
4. Rewrite modular multiplication as a sum of partial products. If $A = \sum_{i=0}^{k-1} 2^i a_i$ and $B = \sum_{i=0}^{k-1} 2^i b_i$:
$$(A \times B)[2^k + 1] = (\sum_{i=0}^{k-1} (PP_i + 1) + 2)[2^k + 1],$$
$$PP_i = a_i \cdot b_{k-i-1} \ldots b_0 \overline{b}_{k-1} \ldots \overline{b}_{k-i} + \overline{a}_i \cdot \underbrace{0 \ldots 0}_{k-i} \underbrace{1 \ldots 1}_{i}.$$

The multiplication methods were evaluated for the cases when both operands are variable and also when one of the operands is constant. The area utilizations of the multiplication methods expressed as the number of 6-input LUTs are given in Table 1.

Based on the concrete parameters for SWIFFT ($k = 8$), we implemented the modular multiplication by ω^{2i} by means of lookup tables. Figure 2 shows the design of this multiplier, where the i^{th} ROM block stores the diminished-one representation of the product resulting from the multiplication of all non-zero numbers modulo p by ω^{2i}. The zero flag of the product is asserted only if the input zero flag is asserted (ω^{2i} is non-zero for all i). Thus, the zero flag does not have to be stored and no logic resources are required for generating it.

Table 1. Number of FPGA lookup tables (LUT) required for modulo (2^k+1) multiplier

Method	Variable inputs		Constant coefficient	
	$k=8$	$k=16$	$k=8$	$k=16$
Lookup tables	8×10^3	1×10^9	32	16×10^3
Quarter-squared	158	32×10^3	32	16×10^3
Multiplier-based	105	345	35	63
Partial product-based	72	145	N/A	N/A

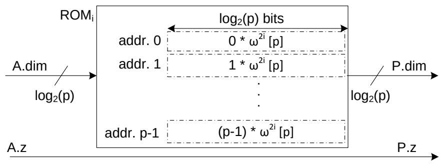

Fig. 2. Constant coefficient modular multiplier implemented by means of lookup tables

3.2 Overall FFT Architecture

Most of the FFT architectures reported in the literature apply a recursive decomposition of the problem into multiple butterfly stages as shown in Figure 3 [6]. The hardware implementation can be iterative, serial or pipelined, mostly depending on the size n of the transform, and the input pattern [17,24]. Also, a *distributed serial arithmetic* approach [28,33] would be feasible in this particular case.

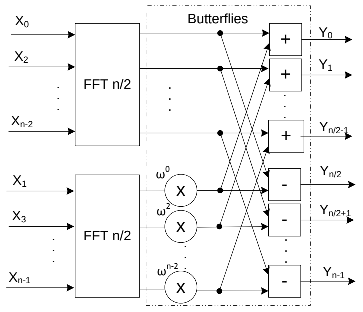

Fig. 3. Traditional implementation of the FFT

Considering the size of the transform required for SWIFFT ($n = 64$) and that the algorithm applies the transform multiple times, we opted for a Radix-2 decomposition pipelined architecture consisting of $\log_2(n)$ butterfly stages where each stage consists of $n/2$ butterflies. The control logic for the FFT is implemented using the *valid-ready handshaking protocol*, as shown in Figure 4.

The butterfly scheme consists in the previously presented modular addition-subtraction unit and constant coefficient multiplier. Thus, all operations inside the FFT are carried out in the diminished-one number system and there is no need to convert the intermediary values back to their normal representation. The throughput of the proposed implementation is 1 cycle / sample.

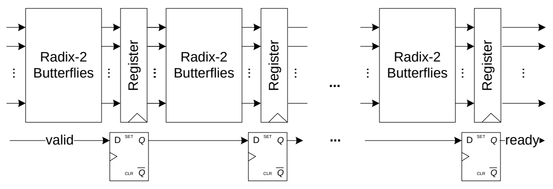

Fig. 4. Pipelined FFT architecture

Note that this FFT architecture could still be optimized in terms of latency - one possible improvement could consist in reducing the overall latency by implementing an FFT of degree 8 by means of lookup tables instead of the traditional decomposition into FFTs of degree 4 and 2. This would reduce the latency by 2 cycles but it would increase the FPGA device resource consumption.

4 Hardware architecture for SWIFFT

As presented in Section 2, the SWIFFT hash function performs FFTs m times to compute the Fourier coefficients of each polynomial of degree $< n$ formed by the input message bits. The overall time needed to execute all the FFT transforms for SWIFFT is $m + \log_2(n)$ cycles.

4.1 Multiplication by the function descriptor matrix

After obtaining the Fourier coefficients $Y_{0,j}, ..., Y_{n-1,j}$ of the polynomials formed by input coefficients $X_{0,j}, ..., X_{n-1,j}$, these coefficients are multiplied by the function descriptor elements $A_{0,j}, ..., A_{n-1,j}$. The resulting products are accumulated for $j = 0, ..., m-1$ to obtain Z_i:

$$Z_i = \sum_{j=0}^{m-1} A_{i,j} \times Y_{i,j}. \tag{4}$$

Considering that $Y_{i,j}$ is represented in diminished-one format, Equation (4) becomes:

$$Z'_i = \sum_{j=0}^{m-1} (A_{i,j} \times Y'_{i,j} + 1) + \sum_{j=0}^{m-1} A'_{i,j}. \tag{5}$$

The pipelined implementation of the FFT proposed in Section 3 computes a new set of Fourier coefficients in each clock cycle. Hence, Equation (5) can be efficiently implemented in hardware as a Multiply-Accumulate unit (MAC) consisting of a pipelined modular multiplier, modular adder and the internal register. The Fourier coefficients generated throughout the m consecutive clock cycles are multiplied by their corresponding function descriptor element and the product is accumulated using the diminished-one modular adder. The second term of Equation (5) is a constant and can be set as the synchronous reset value of the internal register.

The hash value Z'_i (represented in the Fourier domain) is obtained after m iterations of the MAC. The internal register has to be reset in the following clock cycle to start a new operation. This reset implies inserting a stall cycle in the FFT pipeline. The overall throughput of the MAC thus becomes $m+1$ cycles per sample.

Figure 5 shows the SWIFFT architecture that consists of the FFT and MAC components.

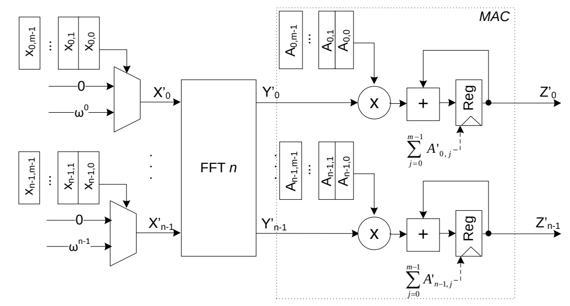

Fig. 5. Overall hardware architecture for the SWIFFT hash function

We considered two approaches for implementing the multipliers inside the Multiply-Accumulate Unit (first term in Equation (5)) having different area and performance trade-offs. *In the first approach*, we treat the function descriptors $A_{i,j}$ as constants and implement constant coefficients multipliers by means of lookup tables, similar to the one shown in Figure 2. The benefit of this approach is that the memory can store the modulo reduced products in diminished-one format and thus multiplication is executed in one clock cycle. However, this option also implies implementing $m \times n$ memories of size $2^k \times (k+1)$ bits.

Based on the parameters proposed for SWIFFT in [22], this is still feasible: the memories evaluating the product of all the Fourier coefficients at a certain index i can be efficiently packed into a single Xilinx BlockRAM primitive inside the FPGA chip. Thus, we obtain a multiplier structure as shown in Figure 6.

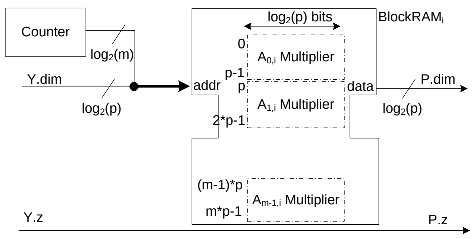

Fig. 6. Multiplier by function descriptor elements $A_{0,i}, ..., A_{m-1,i}$ packed into BlockRAM primitive

If the size of the function descriptor $A_{i,j}$ is larger or if there are multiple function descriptors (as in the case of the SWIFFTX hash function [4]) using the ROM-based method is unattractive due to excessive storage requirements.

The second approach consists in applying an unsigned $k \times k$ bits multiplier and computing the modulo reduced product based on Equation (3). An efficient solution for implementing the multiplier is to use the DSP primitives dedicated for this purpose. The DSP48E slices of a Xilinx FPGA can operate at a higher frequency compared to other primitives - in order to reduce the overall number of required DSP48E slices, it is possible to overclock these primitives to execute two multiplications per cycle [34]. Registers are inserted at the input and the output of the DSP48E blocks which, together with the embedded registers, form double synchronizers that transfer data from/to the double-frequency clock domain. The final architecture of the modular multiplier is shown in Figure 7. Thus, the overall number of DSP48E blocks required for SWIFFT is $n/2$.

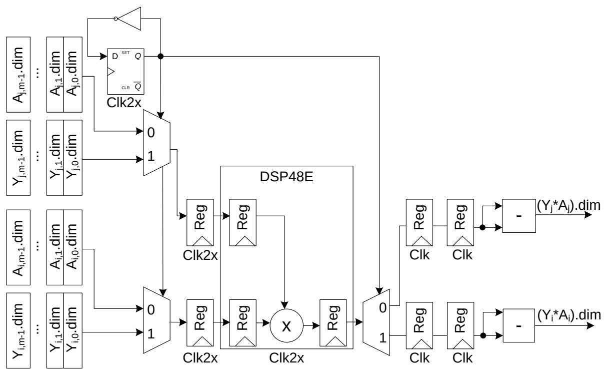

Fig. 7. Implementation of modulo $2^k + 1$ multiplier using overclocked DSP48E primitive

4.2 Overall SWIFFT Architecture

The overall hardware architecture for SWIFFT is shown in Figure 5. We opted for the BlockRAM-based implementation of the MAC multipliers (the first approach) having one cycle latency. The input message is read in n-bit blocks through m consecutive cycles. The input multiplexers implement the multiplication of the input polynomial coefficient bits by ω^i. The output of the multiplexer is already represented in the diminished one format. The FFT starts processing one n-bit block of the input message in each clock cycle and has a latency of $\log_2(n)$ cycles. The MAC accumulates the Fourier coefficients to generate the final hash value. The overall execution time of SWIFFT is $\log_2(n) + m + 2$ cycles. The hash value obtained is represented in the diminished-one format - converting it back to the normal representation is covered in the next section.

5 Hardware architecture for the SWIFFTX hash function

The SWIFFTX hash function [4] uses SWIFFT as its basic building block. The function consists of three layers as shown in Figure 8, whose detailed functionality and design rationale are described in [4]:

1. An inner layer of 3 parallel invocations of SWIFFT on the same input data with different function descriptor matrices;
2. An intermediary layer that converts the output of the inner layer from modulo p to binary and applies S-boxes to break the linearity property of SWIFFT;
3. An outer layer consisting of a single invocation of SWIFFT on the output of the intermediary layer.

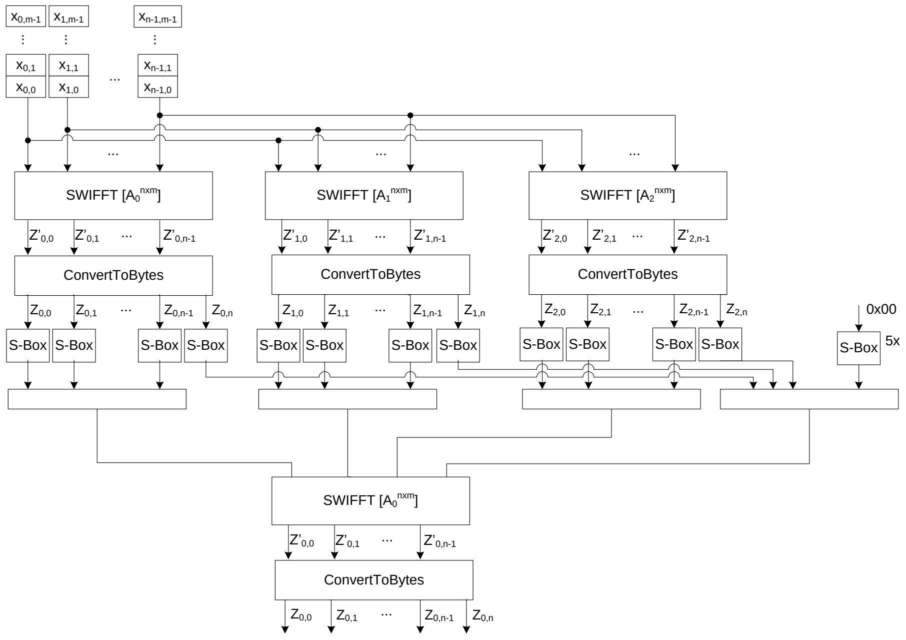

Fig. 8. The SWIFFTX compression function

The proposed hardware implementation follows a data-flow architecture where all the three layers are placed on a single FPGA chip and interconnected by pipeline registers. This leads to an increased area/throughput ratio, but it also results in a high-throughput architecture. In order to process the input message at a fixed rate without additional storage requirements, the three SWIFFT invocations in the inner layer have to be executed in parallel. Considering the SWIFFT architecture proposed in Section 4.2 (Figure 5), it is clear that the component implementing the FFT can be shared across the three instantiations. Hence, only the multiply-accumulate units that perform the linear combination by the function descriptor matrices have to be instantiated three times.

Thus, the architecture of the inner layer implies $3 \times n$ modular multiplier blocks operating in parallel. For the concrete parameters of SWIFFTX ($p = 2^8 + 1$), this can be efficiently implemented using $3n/4$ DSP48E blocks considering that:

1. the DSP48E blocks are overclocked to multiply two Fourier coefficients per cycle;
2. the DSP48E input port is divided into upper and lower 16-bit words, so that it multiplies one Fourier coefficient by two elements from different function descriptors, $A^0_{i,j}$ and $A^1_{i,j}$.

The output of the inner layer is comprised of elements in $\mathbb{Z}_{257}$ that need to be converted into binary qauntities for further use. This is done by the so-called ConvertToBytes function introduced in [4] which is an injective mapping of $n = 64$ elements of $\mathbb{Z}_{257}$ into 65 bytes. The ConvertToBytes function performs the change of base from 257 to 256 by taking groups of 8 elements $Z'_i, ..., Z'_{i+7} \in \mathbb{Z}_{257}$ and producing 8 elements $Z_i, ..., Z_{i+1} \in \mathbb{Z}_{256}$ and a bit b based on the following formula:

$$\sum_{i=0}^{7} Z'_i \times 257^i = \sum_{i=0}^{7} Z_i \times 256^i + b \times 256^8. \tag{6}$$

Then, each of the outputs $Z_i \in \mathbb{Z}_{256}$ are fed to an S-box that performs a simple permutation over $\{0,1\}^8$. The additional bits b generated by the execution of the ConvertToBytes function are combined over 8 groups to form a byte. This combined byte is then also permuted by the S-box. Finally, the algorithm executes SWIFFT over the output bytes produced by the S-box.

Based on Equation (6), $Z_i \in \mathbb{Z}_{256}$ can be expressed as shown in the equations below:

$$\sigma_{-1} = 0, \;\; \sigma_i \times 256 + Z_i = \sum_{k=i}^{7} (\binom{k}{i} \times Z'_k) + \sigma_{i-1} \text{ for } i = 0, \ldots, 7, \;\; b = \sigma_7. \tag{7}$$

As all Z'_k become available in the same cycle, we implement Equation (7) as a pipelined accumulator using a binary tree of 16-bit adders to compute Z_i. The generic design for Z_3 is shown in Figure 9.

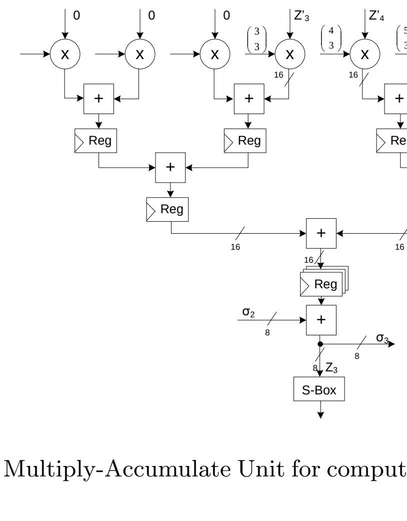

Fig. 9. Pipelined Multiply-Accumulate Unit for computing Z_3 in the ConvertToBytes function

The binomial sum from Equation (7) is computed in 3 cycles. For computing Z_i, σ_{i-1} becomes available after $i + 3$ cycles - we insert i pipeline registers on the datapath in order to compensate the delay. The proposed hardware architecture for the intermediary layer consists of a pipelined implementation of the ConvertToBytes function over one group of 8 elements and 8 S-box instances implemented as lookup tables that permute the outputs of the ConvertToBytes function.

The connection between the inner and intermediary layers is realized through 8 parallel-in serial-out shift registers. The output of the intermediary layer is fed to the outer layer instantiation of SWIFFT.

The overall SWIFFTX architecture is shown in Figure 10. Note that the SWIFFTX hash function has as its final step the so-called FinalTransform procedure. This procedure converts the final 520-bits output of the hash function into 512 uniformly distributed bits. Due to resource limitation, this transform step is still implemented in software. Considering that it is invoked only after the entire message has been hashed, this would not have a significant impact on the function's throughput when hashing long messages.

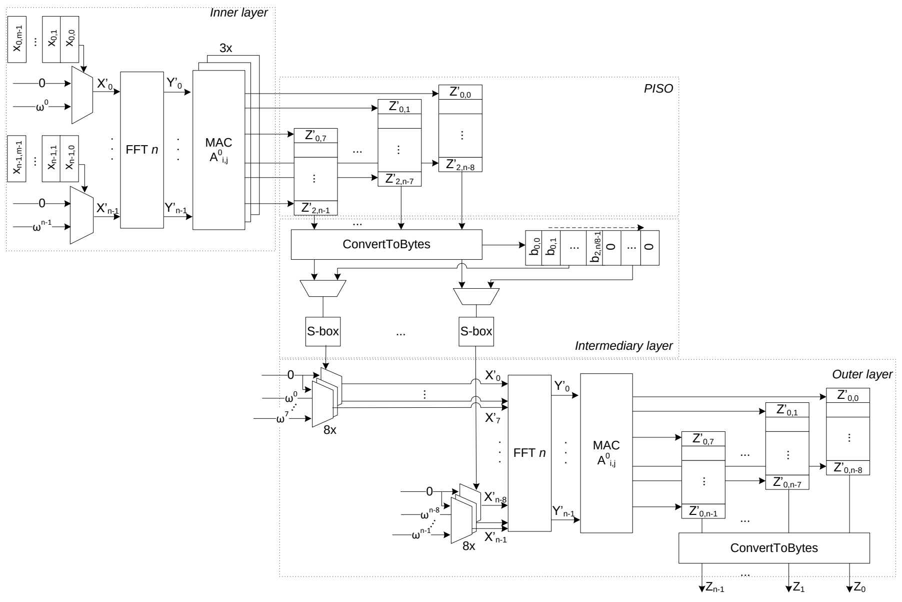

Fig. 10. Hardware architecture for the SWIFFTX hash function

6 Experimental Results and Analysis

The hardware architectures for the FFT, SWIFFT and SWIFFTX, respectively presented in Sections 3, 4 and 5 were implemented in VHDL and tested on a Xilinx ML509 platform featuring a Virtex-5 LX110T FPGA. For synthesis and place-and-route we used the ISE 13.4 design suite. The whole architecture was thoroughly simulated in ModelSIM and tested in the field. For implementing SWIFFT, the parameters of the function were set as proposed in [22], i.e. $n = 64, m = 16, p = 257$ and $\omega = 42$.

The function descriptor multipliers are implemented using the BlockRAM-based approach that has one cycle latency. For SWIFFTX, parameter m is set to 32 as proposed in [4] and the function descriptor multipliers are implemented in DSP48E blocks. Both hash functions use the same implementation of the FFT.

The resource utilizations obtained for the FFT, SWIFFT and SWIFFTX [4] hash functions are presented in Table 2. In our implementation, the FFT core is optimized for maximum throughput - this is achieved by the Radix-2 pipelined decomposition consisting of $(n/2) \cdot \log_2(n)$ butterfly operators suited for small values of n ($n \leq 64$). If n is further increased, it will be necessary to explore different techniques for executing the FFT. On the other hand, increasing parameter m impacts only the execution time of the algorithm but it does not affect the resource requirements significantly.

Table 2. Resource utilization for FFT, SWIFFT and SWIFFTX cores on Virtex-5 LX110T

Core	Slices	BlockRAM	DSP48E	Cycles/sample	Latency (cycles)	Frequency (MHz)
FFT	3,639 (21%)	68 (23%)	0 (0%)	1	$\log_2(n)$	150
SWIFFT	4,004 (23%)	96 (32%)	0 (0%)	$m+1$	$m+\log_2(n)+1$	150
SWIFFTX	16,645 (96%)	69 (23%)	64 (100%)	$m+1$	$2(\log_2(n)+m+1)+n/2+2$	120

In terms of the execution rate, the implementation of the FFT yields a throughput of 1 cycle / sample with a latency of $\log_2(n)$ cycles. SWIFFT consists of m executions of the FFT and the accumulation of the resulting Fourier coefficients. Before starting to process a new message block, a pipeline stall has to be inserted in the FFT - this is required to afford a reset cycle for the Multiply-Accumulate unit. Thus, the next hashing operation can start after $m+1$ cycles. SWIFFTX instantiates SWIFFT both in the inner and outer layers and the ConvertToBytes function in the intermediary and outer layers. Due to pipelining, the next message block can be hashed after $m+1$ cycles, as in the case of SWIFFT. The ConvertToBytes function processes a group of 8 elements, thus it is executed $3n/8+1$ times in the intermediary layer and $n/8$ times in the outer layer - this increases the latency of the hash function. The execution rates and latencies are summarized in Table 2.

For SWIFFTX, the DSP48E blocks are clocked at 240 MHz (twice the operational frequency of the rest of the design). The synthesis on Virtex6 and Virtex7 FPGA devices yields the same maximal operating frequency but the device utilization ratio is better due to larger lookup tables.

Based on the information in Table 2 we compute the throughput of the SWIFFT and SWIFFTX hash functions for long messages when operating in chaining mode. As shown in Table 3, the proposed hardware implementations of the two hash functions achieve a significantly higher (at least one order of magnitude) speed compared to the SIMD optimized software implementations. We feel that the main meaning of this comparison is the fact that FPGA is a suitable architecture for the implementation of (ideal) lattice-based cryptographic primitives.

[4] The area utilization excludes the FinalTransform procedure.

Table 3. Throughput of the SWIFFT and SWIFFTX hardware implementation

Core	Proposed FPGA-based implementation (MB/s)	SIMD software implementation (MB/s)	Speedup
SWIFFT	530	40 [22]	13.3x
SWIFFTX	607	37 [4]	16.4x

Compared to the leading commercial cores [13] and state-of-the-art implementations of the SHA-256 and SHA-512 hash functions (which remain de-facto standards in the industry) [10], the proposed hardware implementation of the SWIFFT and SWIFFTX hash functions achieve a higher throughput due to pipelining. On the other hand, pipelining also has an impact on area requirement - as given in Table 4, the SWIFFT and SWIFFTX hash functions are not as efficient regarding the resource utilization compared to the hardware implementation of other hashing algorithms. However, a remarkable aspect is that the implementation still fits in a single FPGA chip of medium size. Compared to the SHA-3 finalists, SWIFFT and SWIFFTX give a stronger security guarantee by proving that finding collisions in SWIFFT is at least as difficult as finding short vectors in ideal lattices. However, based on Table 4, it can be seen that this guarantee comes at a higher hardware cost.

Table 4. Comparison of the proposed hardware implementation of the SWIFFT and SWIFFTX hash function with published implementation of the SHA-2 and SHA-3 finalist hash functions

Core	Compression ratio	Max. clock (MHz)	Throughput (Gbps)	Slices
SHA-256 core reported in [10]	2	177	1,4	396
SHA-512 core reported in [10]	2	159	2.01	798
Helion Fast SHA-256 core [13]	2	221	1.71	319
Helion Fast SHA-512 core [13]	2	190	2.37	608
Proposed implementation of SWIFFT	2	150	4.24	4,004
Proposed implementation of SWIFFTX	4	120	4.85	16,645
BLAKE core reported in [10]	2	210	7.55	3,495
Groestl core reported in [10]	2	243	12.48	2,971
JH core reported in [10]	2	348	8.29	2,312
Keccak core reported in [10]	2	276	12.52	2,123
Skein core reported in [10]	2	230	5.34	1858

7 Conclusion and Future Work

This paper presented a pipelined architecture for the FFT performed over $\mathbb{Z}_p$ using the diminished-one number system. Based on the FFT, a high throughput hardware implementation for the SWIFFT [22] and SWIFFTX [4] lattice-based cryptographic hash functions is proposed. The implementations of both hash functions were simulated and tested in real hardware (Xilinx Virtex5 FPGA chips) and achieve a significant speedup (more than one order of magnitude) compared to the SIMD software implementations reported in [22] and [4]. Compared to the hardware implementation of the SHA-256 and SHA-512 hash functions, the proposed implementation of SWIFFT and SWIFFTX obtain a higher throughput, though with increased area utilization. The throughput to area ratio of SWIFFT and SWIFFTX are below the ratio of the hardware implementations for the SHA-3 finalist functions.

As mentioned in the introduction, the proposed implementations for SWIFFT and SWIFFTX are optimized for the concrete parameters proposed in [22] and are not scalable regarding area requirements. However, it should be pointed that our architecture can, up to minor modifications, accomodate slightly larger values of n, such as the last parameter set of Table 1 of [5], in order to reach larger levels of security: one just needs to use the decomposition of an FFT of size $2n$ into two FFTs of size n, at a cost of a factor of 2 loss in throughput. An increase of m can similarly be dealt with, up to a linear loss in throughput.

Our implementation choice however makes heavy use of the choice $p = 257$. One might think of exploring more generic or scalable architectures for the modular FFT. Since this modular FFT is the building block of all lattice-based primitives, this would be of great use for future work. We intend to continue in this direction in a forthcoming paper regarding Lyubashevsky's signature scheme [20].

References

1. D. P. Agrawal and T. R. N. Rao. Modulo $(2^n + 1)$ arithmetic logic. *IEEE Journal on Electronic Circuits and Systems*, 2(6):186–188, 1978.
2. M. Ajtai. Generating Hard Instances of Lattice Problems (Extended Abstract). In *STOC*, pages 99–108. ACM, 1996.
3. M. Ajtai. Generating Hard Instances of the Short Basis Problem. In *Proceedings of the 1999 International Colloquium on Automata, Languages and Programming (ICALP 1999)*, volume 1644 of *LNCS*, pages 1–9. Springer, 1999.
4. Y. Arbitman, G. Dogon, V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. SWIFFTX: A Proposal for the SHA-3 Standard. Submission to NIST, 2008. Available at `http://www.eecs.harvard.edu/~alon/PAPERS/lattices/swifftx.pdf`.
5. J. Buchmann and R. Lindner. Secure Parameters for SWIFFT. In *INDOCRYPT*, volume 5922 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009.
6. J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.
7. A. V. Curiger, H. Bonnenberg, and H. Kaeslin. Regular VLSI architectures for multiplication modulo $(2n + 1)$. *IEEE Journal of Solid-State Circuits*, 26(7):990–994, 1991.
8. Ö. Dagdelen and M. Schneider. Parallel Enumeration of Shortest Lattice Vectors. In *Proc. of Euro-Par' 2010*, volume 6272 of *LNCS*, pages 211–222. Springer, 2010.
9. J. Detrey, G. Hanrot, X. Pujol, and D. Stehlé. Accelerating lattice reduction with FPGAs. In *Proc. of LATINCRYPT*, volume 6212 of *LNCS*, pages 124–143. Springer, 2010.
10. K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif. Comprehensive evaluation of high-speed and medium-speed implementations of five SHA-3 finalists using Xilinx and Altera FPGAs. 3rd SHA-3 candidate conference, Mar 2012.
11. N. Gama and Phong Q. Nguyen. Predicting Lattice Reduction. In *Proc of Eurocrypt 2010*, volume 6110 of *LNCS*, pages 31–51. Springer.
12. C. Gentry and S. Halevi. Implementing Gentry's Fully-Homomorphic Encryption Scheme. In *Proc. of EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2011.
13. Helion, Cambridge, UK, `http://heliontech.com/fast_hash.htm`. *Fast Hash Core Family for Xilinx FPGA*.
14. J. Hermans, M. Schneider, J. Buchmann, F. Vercauteren, and B. Preneel. Parallel Shortest Lattice Vector Enumeration on Graphics Cards. In *Proc. of AfricaCrypt 2010*, volume 6055 of *LNCS*, pages 52–68. Springer.

15. J. Hermans, F. Vercauteren, and B. Preneel. Speed Records for NTRU. In *Proc. of CT-RSA 2010*, pages 73–88, 2010.
16. J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A Ring-Based Public Key Cryptosystem. In *Proc. of ANTS 3*, pages 267–288, 1998.
17. W. R. Knight and R. Kaiser. A Simple Fixed-Point Error Bound for the Fast Fourier Transform. *IEEE Trans. on Acoust., Speech, Signal Processing*, 27(6):615–620, 1979.
18. P-C. Kuo, M. Schneider, Ö. Dagdelen, J. Reichelt, J. Buchmann, C-M. Cheng, and B-Y. Yang. Extreme Enumeration on GPU and in Clouds. In *Proc. of CHES'2011*, volume 6917 of *LNCS*, pages 176–191. Springer, 2011.
19. R. Lindner and C. Peikert. Better key sizes (and attacks) for LWE-based encryption. In *Proc. of CT-RSA 2011*, volume 6558 of *LNCS*, pages 319–339. Springer.
20. V. Lyubashevsky. Lattice signatures without trapdoors. In *Proc. of Eurocrypt 2012*, volume 7237 of *LNCS*, pages 738–755. Springer, 2012.
21. V. Lyubashevsky and D. Micciancio. Generalized Compact Knapsacks Are Collision Resistant. In *Proc. ICALP (2)*, volume 4052 of *LNCS*, pages 144–155. Springer, 2006.
22. V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. SWIFFT: a modest proposal for FFT hashing. In K. Nyberg, editor, *Fast Software Encryption – FSE 2008*, number 5086 in Lecture Notes in Computer Science, pages 54–72. Springer, 2008.
23. V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *Proc. of EUROCRYPT*, volume 6110 of *LNCS*, pages 1–23. Springer, 2010.
24. U. Meyer-Baese. *Digital Signal Processing with Field Programmable Gate Arrays (Signals and Communication Technology)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.
25. D. Micciancio and O. Regev. Lattice-Based Cryptography. In *Post-Quantum Cryptography, D. J. Bernstein, J. Buchmann, E. Dahmen (Eds)*, pages 147–191. Springer, 2009.
26. M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *Proc. of CCSW*, pages 113–124. ACM, 2011.
27. C. Peikert and A. Rosen. Efficient Collision-Resistant Hashing from Worst-Case Assumptions on Cyclic Lattices. In *Proc. TCC 2006*, volume 3876 of *LNCS*, pages 145–166. Springer, 2006.
28. A. Peled and B. Liu. A New Hardware Realization of Digital Filters. *IEEE Trans. on Acoust., Speech, Signal Processing*, 22:456–474, 1974.
29. J. M. Pollard. The Fast Fourier Transform in a finite field. *Mathematics of Computation*, 25(114):365–374, 1971.
30. O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proc. of STOC*, pages 84–93. ACM, 2005.
31. O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6), 2009.
32. N. P. Smart and F. Vercauteren. Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes. In *Public Key Cryptography*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443. Springer, 2010.
33. S. A. White. Applications of Distributed Arithmetic to Digital Signal Processing. *IEEE ASSP Magazine*, 6(3):4–19, 1989.
34. Xilinx Inc., `http://www.xilinx.com/support/documentation/user_guides/ug193.pdf`. *Virtex-5 FPGA XtremeDSP Design Considerations.*
35. R. Zimmermann. Efficient VLSI implementation of modulo $(2^n + 1)$ addition and multiplication. In I. Koren and P. Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 158–167. IEEE Computer Society Press, Los Alamitos, CA, 1999.