Hardware Complexity of Modular Multiplication and Exponentiation

Jean Pierre David, *Member*, *IEEE*, Kassem Kalach, and Nicolas Tittley

Abstract—Large integer Modular Multiplication (MM) and Modular Exponentiation (ME) are the foundation of most public-key cryptosystems, specifically RSA, Diffie-Helleman, ElGamal, and the Elliptic Curve Cryptosystems. Thus, MM algorithms have been studied widely and extensively. Most of the work is based on the well-known Montgomery Multiplication Method and its variants, which require standard multiplication operations. Despite their better complexity orders, Karatsuba and FFT algorithms seem to rarely be used for hardware implementation. In this paper, we review their hardware complexity and propose original implementations of MM and ME that become useful for 24-bit operators (Karatsuba algorithm) or 373-bit operators (FFT algorithm).

Index Terms—Cryptography, multiplication, modular arithmetic, hardware complexity.

1 Introduction

The ever-increasing needs in data communication and the expansion of Internet services, namely, financial applications and electronic commerce, have made security a major concern. Private-key cryptography is unfortunately impractical for such applications because of its main drawback: the Key Distribution Problem. Public-key cryptography [1], introduced in 1976, was the first bright solution and so far has been shown to be the only convenient security technique for such applications.

Since then, many public-key cryptosystems have been designed and implemented. The most dominant realizations are RSA, ElGamal, and Elliptic Curve Cryptosystems (ECCs). These systems base their security on the computational difficulty of solving some mathematical problems in modular arithmetic.

All current public-key cryptosystems use Modular Multiplication (MM), and most of them require Modular Exponentiation (ME). Thus, the performance of any public-key cryptosystem is primarily determined by the efficiency of the MM algorithm (multiplication and remainder operations) and its implementation. Unfortunately, the complexity of the classical method for multiplying integers is $O(N^2)$ and the same is true for the modulo operation. Besides, for security reasons, these algorithms are applied to very large integers (1,024 bits or higher). This is an important obstacle to attaining high throughput, which is a vital criterion in network applications.

A number of papers have been published on the optimization of MM. Most of the contributions are based on the Montgomery Multiplication Method (MMM) [2], which achieves MM and ME with classical multiplication and power-of-2 division. Many variants (that is, the combination of the MMM with a particular mathematical or hardware technique) have also been proposed. We mention, for example, the Residue Number System [3], [4], [5], Chinese Remainder Theorem [6], [7], Redundant Number System [8], [9], High Radix [10], Systolic Array [10], [11], Carry Save Adders [12], [13], and Lookup Tables [14]. Nevertheless, the intrinsic complexity of these approaches relies on the multiplication and is $O(N^2)$.

Other methods exist to multiply (large) integers. Karatsuba and Ofman proposed a recursive algorithm, which is about $O(N^{1.58})$ [15], and the Fast Fourier Transform (FFT) [16] has the smallest known complexity, $O(N\log(N))$, for integer multiplication. Fig. 1 illustrates MM complexity order for both classical and Montgomery representations.

This figure will be detailed in the following sections. The important point here is to notice that an MM can have the same complexity order as a standard multiplication, which can be as low as $O(N^{1.58})$ or $O(N\log(N))$. Nevertheless, these orders hide constants that make the algorithms useful only beyond a given threshold. The present paper reviews the hardware complexity (HC) of squaring, multiplication, modular squaring (MS), MM, and ME with classical, Karatsuba, and FFT-based operators. An important contribution is also proposed in the hardware optimization of FFT-based operators.

Given these implementations, we propose to answer the following question: "What are the thresholds that determine when classical, Karatsuba, or FFT-based operators should be used to minimize the HC?" After an extensive search, the only FFT-based hardware implementation that we have found is a recent work that aims at multiplying ultra-large numbers (12,000,000 digits) [17]. Our results

- *J.P. David is with the Département de Informatique et Recherche Operationnelle, Université de Montréal and the Ecole Polytechnique de Montréal, Genie Electrique, C.P. 6079, succursale Centre-ville Montréal H3C 3A7, Québec, Canada. E-mail: jpdavid@polymtl.ca.*
- *K. Kalach is with the Crypto Group, DICE, Université Catholique de Louvain, Place due Levant, 3 (Batiment Maxwell), 1348 Louvain-La-Neuve, Belgium. E-mail: kalach@nacara.ucl.ac.be.*
- *N. Tittley is with the Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, CP 6128, Succ. Centre-Ville, Montréal, Québec, Canada H3C 3J7. E-mail: nicolas.tittley@umontreal.ca.*

Manuscript received 8 Aug. 2006; revised 24 Nov. 2006; accepted 2 Jan. 2007; published online 29 May 2007.
Recommended for acceptance by J.-C. Bajard.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0308-0806.
Digital Object Identifier no. 10.1109/TC.2007.1084.

0018-9340/07/$25.00 © 2007 IEEE Published by the IEEE Computer Society

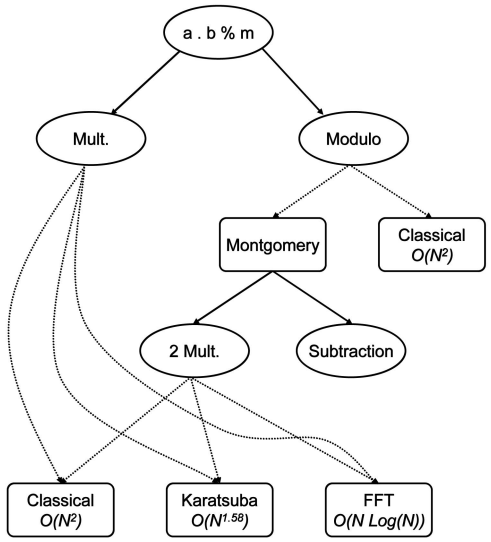

Fig. 1. Modular multiplication complexity order.

demonstrate that, to multiply two integers, the FFT method is better than the classical one for 346-bit operands and better than the Karatsuba one for 1,824-bit operands, whereas the Karatsuba method starts to be more efficient than the classical one for 16-bit operands. We have obtained similar results for modular operations (MM and ME).

The rest of this paper is organized as follows: Section 2 presents the ME algorithm and defines our approach to measure its HC. The first refinement step leads to MM and MS algorithm implementations, which are studied in Section 3. The second refinement step is presented in Section 4, where various algorithms and implementations of the square and multiplication operations are described. Section 5 is dedicated to the study of FFT-based operators and, in particular, details our original hardware implementation. The HC of squarer, multiplier, MS, MM, and ME have been computed for several architectures and operand sizes. Section 6 presents and compares the results, including a preliminary FPGA implementation. Section 7 concludes this work.

2 Modular Exponentiation Complexity

2.1 Modular Exponentiation Computation

ME has the form

$$y = x^e \bmod m, \tag{1}$$

where x is the base, e is the exponent, and m is the modulus. We consider that all integers are unsigned N-bit numbers.

One of the most efficient techniques to compute exponentiation is the square-and-multiply algorithm (Algorithm 1), which is also called Binary Exponentiation. It drastically reduces both the number of operations and the memory required to perform ME. The exponent e must first be converted into its binary notation. That is, e can be written as

$$e = (e_{N-1}e_{N-2}\dots e_1e_0) = \sum_{i=0}^{N-1} e_i 2^i. \tag{2}$$

TABLE 1
Atomic Device Complexity

Identifier	Complexity
θ_{gate}	One 2-input gate
θ_{mux}	One 2-input multiplexer
θ_{HA}	One Half Adder
θ_{FA}	One Full Adder

Bits e_i are scanned from left to right. An MS is performed at each step. Depending on the e_i value, a subsequent MM is done. The following equation illustrates this strategy:

$$x^{13} = x^{1101b} = (((1^2x)^2x)^2)^2x. \tag{3}$$

To minimize the ME's HC, we propose studying the possible implementations of MS and MM according to Fig. 1. Nevertheless, we must first define what we call "HC." This is detailed in the next section.

Algorithm 1 Modular Binary Exponentiation (MBE).
1: Input: x, e, m
2: Output: $y = x^e \bmod m$
3: $y = 1$
4: **for** $i = N - 1$ downto 0 **do**
5: $\quad y = y^2 \bmod m$
6: $\quad$ **if** $(e_i = 1)$ **then**
7: $\quad\quad y = y \cdot x \bmod m$
8: $\quad$ **end if**
9: **end for**
10: return y

2.2 Measure of Algorithms Complexity

The abstraction level that we address is the theoretical number of atomic devices required to implement an algorithm. The atomic devices that we consider are simple gates (AND, OR, XOR), two-input multiplexer, Half Adder (HA), and Full Adder (FA). Their intrinsic complexity, which is technology dependent, is denoted θ_{device} (Table 1). The routing, the delays, the power consumption, and the potential logic redundancy that depend on the actual implementation are not taken into account at this level.

All of our algorithms are refined until they can be implemented as a network of such basic devices in order to measure their HC, which is expressed by their $\theta_{algo}()$ function:

$$\theta_{algo}() = \dots\theta_{gate} + \dots\theta_{mux} + \dots\theta_{HA} + \dots\theta_{FA}. \tag{4}$$

We consider only pure combinatorial implementations. The information can, however, be easily extrapolated to pipelined or functionally pipelined implementations when possible. In this context, multiplications and divisions by constant powers of 2 are free.

2.3 Power-of-2 Sum Complexity

Many algorithms require the implementation of power-of-2 sums. That is,

$$result = \sum_i x_i \cdot 2^{w_i}, \tag{5}$$

TABLE 2
Complexity of Basic Functions

Identifier	Complexity
$\theta_{ADD}(w_a, w_b, w_r)$	$\theta_{P2Sum}(\dots)$
$\theta_{ADD}(w_a, w_b)$	$\theta_{ADD}(w_a, w_b, \max(w_a, w_b)+1)$
$\theta_{ADD}(w_a)$	$\theta_{ADD}(w_a, w_a)$
$\theta_{SUB}(\dots)$	$\theta_{ADD}(\dots)$ (Subtracter)
$\theta_{ADDC}(w_a), \theta_{SUBB}(w_a)$	$w_a\theta_{FA}$ (with Carry/Borrow)
$\theta_{CSTADD}(w_r)$	$w_r\theta_{HA}$ (Constant adder)
$\theta_{2-CPL}(w_r)$	$w_r\theta_{HA}$ (Two's complement)

where x_is are binary input variables with associated weight w_i.

The complexity $\theta_{P2Sum}(\{x_i\})$ is computed by a fixed-point algorithm (Algorithm 2) that tries to allocate FA when possible and HA otherwise until the whole sum is computed.

Algorithm 2 power-of-2 sum complexity.

1: Input: x_i, w_i
2: Output: complexity in terms of θ_{HA} and θ_{FA}
3: Start: $C_{HA} = C_{FA} = 0$
4: Let $c_i =$ number of $x_k | w_k = i$
5: **while** $(\exists i | (c_i > 1))$ **do**
6: **if** $(\exists j | (c_j \geq 3))$ **then**
7: $c_j = c_j - 2$; $c_{j+1} = c_{j+1} + 1$; $C_{FA} = C_{FA} + 1$
8: **else**
9: let $j | c_j = 2$ and $\forall k < j$, $c_k < 2$
10: $c_j = c_j - 1$; $c_{j+1} = c_{j+1} + 1$; $C_{HA} = C_{HA} + 1$
11: **end if**
12: **end while**
13: return $(C_{HA}\,\theta_{HA} + C_{FA}\,\theta_{FA})$

We define the θ_{algo} (Table 2) for basic functions that are trivially implemented by HA and FA atomic resources. The variables w_a and w_b represent the widths of operands a and b, respectively, whereas w_r represents the required result width.

3 Modular Multiplication

This section presents standard and Montgomery methods to compute the MM and details their HC. The MM is given by

$$p = a \cdot b \bmod m. \tag{6}$$

3.1 Standard Modular Multiplication (SMM)

This method consists of computing the multiplication and then computing the modulus. It is detailed in Algorithm 3. The SMM's HC is concentrated in lines 3 and 6 because shift resources are free. Multiplication can be implemented in various ways, as shown in the next section. Test and subtraction at line 6 are actually a single subtractor resource (the test is the subtracter's borrow). A multiplexer is used to choose between p and $p-m$ values. This block is replicated $N+1$ times to implement the FOR structure. A hardware implementation is given in Fig. 2. The HC is

$$\theta_{MODULO}(N) = (N+1)(\theta_{SUB}(N) + \theta_{MUX}(N)), \tag{7}$$

$$\theta_{SMM}(N) = \theta_{MULT}(N) + \theta_{MODULO}(N). \tag{8}$$

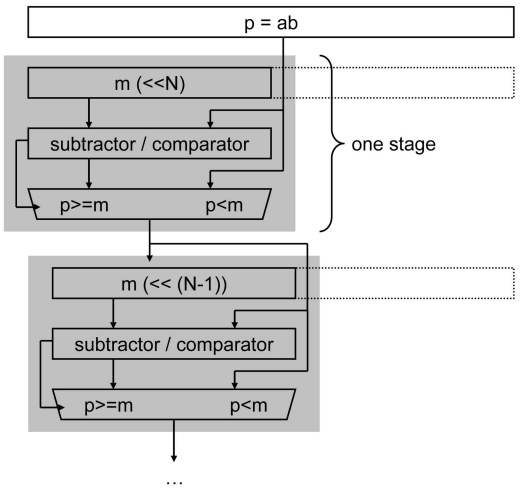

Fig. 2. Standard Modulo Multiplication.

Algorithm 3 Standard Modular Multiplication (SMM).

1: Input: a, b, m
2: Output: $a \cdot b \bmod m$
3: $p = a \cdot b$
4: $m = m \ll N$
5: **for** $i = N$ downto 0 **do**
6: if $(p \geq m)$ $p = p - m$
7: $m = m \gg 1$
8: **end for**
9: return p

3.2 The Montgomery Multiplication Method

Introduced in 1985 [2], Montgomery's MM algorithm is one of the most efficient methods to perform ME. Let m be a modulo of N bits and r be defined as $r = 2^N$. Parameters m and r must be relatively prime in order to have unique inverses (r^{-1}, m^{-1}) in $\mathbf{Z_m}$ and $\mathbf{Z_r}$, respectively. This necessary condition for the MMM is satisfied for any odd m since r is a power of 2.

Given an integer $0 \leq a < m$, the Montgomery representation (also called the m-residue) with respect to r is defined by

$$\overline{a} = a \cdot r \bmod m. \tag{9}$$

The Montgomery Reduction (MR) is the inverse transform of the m-residue representation. It is defined by

$$a = \mathrm{MR}(\overline{a}) = \overline{a} \cdot r^{-1} \bmod m, \tag{10}$$

where $\overline{a}$ is the m-residue of a, and r^{-1} is the inverse of r in $\mathbf{Z_m}$.

Given the m-residues $\overline{a}$ and $\overline{b}$, Montgomery proposes an algorithm to compute the m-residue of $a \cdot b$ with power-of-2 divisions only (due to the special form of r). This is detailed in Algorithm 4. The squaring version of the algorithm is straightforward and will be called Montgomery Square Reduction (MSR).

Algorithm 4 Montgomery Product Reduction (MPR).

1: Input: $\overline{a}$, $\overline{b}$,
2: Output: $\overline{c} = \overline{a \cdot b}$
3: $u = \overline{a} \cdot \overline{b}$
4: $v = u \cdot (-m^{-1}) \bmod r$

5: $\overline{c} = (u + v \cdot m)/r$
6: **if** $(\overline{c} \geq m)$ **then**
7: $\quad \overline{c} = \overline{c} - m$
8: **end if**
9: Return $\overline{c}$

This algorithm is not efficient at all for a single MM because it requires m-residue transformations (for operands) and reduction (for result). Besides, constants r^{-1} and $(-m^{-1})$ must be precomputed. Nevertheless, when the same data is involved in a large set of MM, this algorithm has the asymptotical complexity of a multiplication and becomes very interesting for large numbers since the standard modulo operation is $O(N^2)$.

The HC of this algorithm is composed of two full multipliers (lines 3 and 5), one N-bit truncated multiplier (line 4), one $2N$-bit adder (line 5), and one N-bit subtractor/multiplexer stage similar to the one used in SMM (lines 6 and 7). The HC is

$$\begin{aligned}\theta_{REDUCT}(N) = &\theta_{MULT}(N) + \theta_{MULT}(N, N, N) \\ &+ \theta_{ADD}(2N) + \theta_{SUB}(N) \\ &+ \theta_{MUX}(N),\end{aligned} \tag{11}$$

$$\theta_{MSR}(N) = \theta_{SQUARE}(N) + \theta_{REDUCT}(N), \tag{12}$$

$$\theta_{MPR}(N) = \theta_{MULT}(N) + \theta_{REDUCT}(N). \tag{13}$$

The squarer and multiplier complexities are analyzed in the next section. Combining MSR/MPR with MBE, MEM is presented in Algorithm 5. This algorithm has N stages. Each stage is composed of one MSR and one conditional MPR. The HC of one stage is

$$\theta_{MEM}(N) = \theta_{MPR}(N) + \theta_{MSR}(N) + \theta_{MUX}(N). \tag{14}$$

Algorithm 5 Montgomery Exponentiation Method (MEM).
1: Input: x, e, m
2: Output: $y = x^e \bmod m$
3: $r = -m^{-1}$ (using the extended euclidean algorithm)
4: $\overline{x} = x \cdot r \bmod m$
5: $\overline{y} = 1 \cdot r \bmod m$
6: **for** $i = N - 1$ downto 0 **do**
7: $\quad \overline{y} = MSR(\overline{y})$
8: $\quad$ **if** $(e_i = 1)$ **then**
9: $\quad\quad \overline{y} = MPR(\overline{x}, \overline{y})$
10: $\quad$ **end if**
11: **end for**
12: $y = MPR(\overline{y}, 1)$
13: return y

4 Multiplication Algorithms

4.1 The Standard Multiplication Algorithm (SMA)

Let a and b be two N-bit numbers:

$$a = (a_{N-1}a_{N-2}\dots a_0) = \sum_{i=0}^{N-1} a_i 2^i, \tag{15}$$

Fig. 3. The Standard Multiplication Algorithm.

$$b = (b_{N-1}b_{N-2}\dots b_0) = \sum_{i=0}^{N-1} b_i 2^i. \tag{16}$$

The standard (classical) multiplication method computes the partial products $p_{ij} = a_i b_i$. Each product has a weight 2^{i+j}. Then, all products are summed (according to their weight), resulting in a $2N$-bit number. This is illustrated in Fig. 3 for 2-bit integers.

The last row of this table is the total sum of the partial products, giving the product of a by b. This method is presented in Algorithm 6, where partial products are iteratively computed and summed at Step 8, known as the inner product operation.

Algorithm 6 SMA.
1: Input: a, b
2: Output: $p = ab$
3: Variable: C(Carry) and S(Sum)
4: Initially, $p_i = 0$ for all $i = 0, 1, \cdots, 2N - 1$
5: **for** $i = 0$ to $N - 1$ **do**
6: $\quad \mathrm{C} = 0$
7: $\quad$ **for** $j = 0$ to $N - 1$ **do**
8: $\quad\quad (\mathrm{C}, \mathrm{S}) = p_{i+j} + b_i a_j + C$
9: $\quad\quad p_{i+j} = S$
10: $\quad$ **end for**
11: $\quad p_{i+N} = C$
12: **end for**
13: return $(p_{2N-1}p_{2N-2}\dots p_0)$

This algorithm's complexity is $O(N^2)$. It is asymptotically the slowest one in the literature but also the simplest. Its HC can thus be given as

$$\theta_{SMA}(N) = \theta_{P2Sum}(\{p_{i,j}\}) + N^2\theta_{GATE}. \tag{17}$$

The general form of the SMA (including r-bit truncated multiplications) complexity is

$$\begin{aligned} &i \in [0, w_a - 1], \\ &j \in [0, w_b - 1], \\ &P = \{p_{i,j} | (i + j) < w_r\}, \\ \theta_{SMA}(w_a, w_b, w_r) = &\theta_{P2Sum}(P, w_r) \\ &+ \#P\ \theta_{GATE}. \end{aligned} \tag{18}$$

The Standard Squaring Algorithm (SSA) is also $O(N^2)$, but has a smaller complexity because $p_{i,j} = p_{j,i}$ and $p_{i,j} + p_{j,i} = 2p_{i,j}$, which is simply a change of weight of $p_{i,j}$.

4.2 The Karatsuba Algorithm

Karatsuba is a Russian mathematician who proposed a recursive multiplication algorithm with a complexity of about $O(N^{1.58})$. Let a and b be two N-bit numbers and let $l = N/2$. Initially, a and b are split into two equal-sized parts as follows:

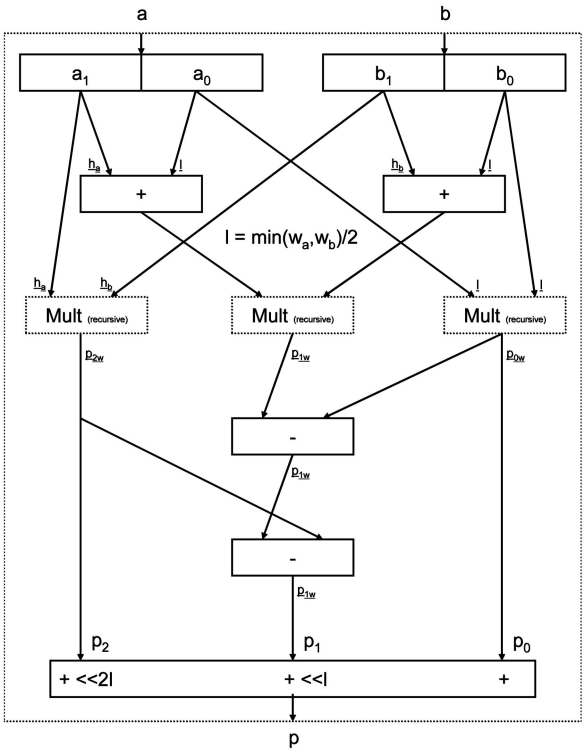

Fig. 4. Karatsuba multiplication implementation.

$$a = 2^l a_1 + a_0, \quad b = 2^l b_1 + b_0. \tag{19}$$

Thus, a_0 and b_0 contain the l least significant bits and a_1 and b_1 contain the most significant bits. The value 2^l denotes the base β of this system. Karatsuba's algorithm transforms the multiplication of a and b into the multiplications of half-sized numbers a_0, a_1, b_0, and b_1, illustrated as follows:

$$\begin{aligned} p &= a \cdot b \\ &= (2^l a_1 + a_0).(2^l b_1 + b_0) \\ &= 2^{2l}(a_1 b_1) + 2^l(a_1 b_0 + a_0 b_1) + a_0 b_0 \\ &= 2^{2l} p_2 + 2^l p_1 + p_0. \end{aligned} \tag{20}$$

This formalism is simply a recursive version of SMA and its complexity is still $O(N^2)$, but Karatsuba noticed that p_1 could be expressed differently as

$$p_1 = a_1 b_0 + a_0 b_1 = (a_0 + a_1) \cdot (b_0 + b_1) - p_0 - p_2. \tag{21}$$

The two multiplications and the addition previously required are replaced by one multiplication, two additions, and two subtractions, leading to a global complexity order of $N^{\log_2(3)}$. Karatsuba's algorithm is consequently asymptotically more efficient than the classical one. However, the latter is better for small-sized numbers. Therefore, a hybrid algorithm is often used. It consists of applying Karatsuba's algorithm until the coefficient widths become lower than the threshold at which point it is more interesting to use the classical method. This algorithm is described in Algorithm 7 and its hardware implementation is illustrated in Fig. 4.

Algorithm 7 Karatsuba Multiplication Algorithm (KMA).
1: Inputs: a (w_a bits), b (w_b bits)
2: Output: p (w_r bits) $= a \cdot b$
3: **if** (w_a, w_b and w_r are too small) **then**
4: return $\mathrm{SMA}(a, b)$
5: **end if**
6: $a_0 = a \bmod 2^l$; $a_1 = a/2^l$
7: $b_0 = b \bmod 2^l$; $b_1 = b/2^l$
8: $p_0 = \mathrm{KMA}(a_0, b_0)$
9: $p_2 = \mathrm{KMA}(a_1, b_1)$
10: $p_1 = \mathrm{KMA}(a_0 + a_1, b_0 + b_1) - p_0 - p_2$
11: $p = 2^{2l} p_2 + 2^l p_1 + p_0$
12: return p

We propose extending this algorithm to arbitrary widths (including r-bit truncated multiplications). To be able to compute p_1 from p_2 and p_0 when $w_r < w_a + w_b$, the required widths p_{0w}, p_{1w}, and p_{2w} are computed as follows:

$$l = \frac{\min(w_a, w_b)}{2}, \; h_a = w_a - l, \; h_b = w_b - l, \tag{22}$$

$$p_{0w} = 2l, \tag{23}$$

$$p_{1w} = \min(l + \max(h_a, h_b) + 1, w_r - l), \tag{24}$$

$$p_{2w} = \min(h_a + h_b, \max(p_{1w}, w_r - 2l)). \tag{25}$$

The HC of this algorithm is

$$\begin{aligned} \theta_{KMA}(w_a, w_b, w_r) = {} & \theta_{KMA}(l) + \theta_{KMA}(h_a, h_b, p_{2w}) \\ & + \theta_{ADD}(l, h_a) + \theta_{ADD}(l, h_b) \\ & + \theta_{KMA}(\max(l, h_a) + 1, \max(l, h_b) + 1, p_{1w}) \\ & + \theta_{SUB}(p_{1w}, p_{0w}, p_{1w}) + \theta_{SUB}(p_{1w}, p_{2w}, p_{1w}) \\ & + \theta_{P2Sum}(p_2, p_1, p_0). \end{aligned} \tag{26}$$

The HC of the Karatsuba Squaring Algorithm (KSA) can be easily deduced by considering that $w_b = w_a$ and $h = h_a = h_b$. It is given by the following:

$$\begin{aligned} \theta_{KSA}(w_a, w_r) = {} & \theta_{KSA}(l) + \theta_{KSA}(h, p_{2w}) \\ & + 2\theta_{ADD}(l, h) \\ & + \theta_{KMA}(\max(l, h) + 1, \max(l, h) + 1, p_{1w}) \\ & + \theta_{SUB}(p_{1w}, p_{0w}, p_{1w}) + \theta_{SUB}(p_{1w}, p_{2w}, p_{1w}) \\ & + \theta_{P2Sum}(p_2, p_1, p_0). \end{aligned} \tag{27}$$

5 FFT Implementation of MM and ME

5.1 The Principle of FFT Multiplication

The multiplication of large integers using the FFT algorithm is based on the polynomial multiplication:

$$A(x) = \sum_{i=0}^{S/2-1} a_i x^i, \tag{28}$$

$$B(x) = \sum_{i=0}^{S/2-1} b_i x^i, \tag{29}$$

$$P(x) = A(x) \cdot B(x) = \sum_{i=0}^{S-1} p_i x^i. \tag{30}$$

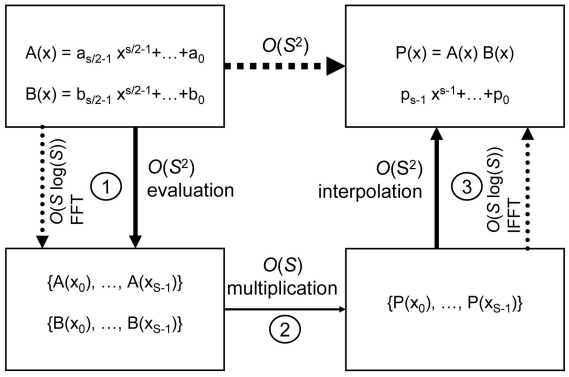

Fig. 5. Polynomial multiplication.

Given two numbers A and B, the first step consists of defining the $\{a_i\}$, $\{b_i\}$, and a base β satisfying the following system:

$$A = A(x)|_{x=\beta}, \tag{31}$$

$$B = B(x)|_{x=\beta}, \tag{32}$$

$$\Rightarrow P = P(x)|_{x=\beta}. \tag{33}$$

A degree S polynomial $P(x)$ is completely defined by S pairs $(x_i, P(x_i))$. Its coefficients can be computed by interpolation. The polynomial multiplication by FFT is a three-step process, as illustrated in Fig. 5:

1. Evaluate $A_i = A(x_i)$ and $B_i = B(x_i)$.
2. Compute products $(x_i, P(x_i)) = (x_i, A_i \cdot B_i)$.
3. Interpolate $\{(x_i, P(x_i))\}$ to find the p_i.

Evaluation and interpolation processes are generally $O(S^2)$. However, for a specific set of x_i $(0 \leq i < S)$, the discrete Fourier transform (DFT) and its inverse (IDFT) can compute them in $O(S\log(S))$ time by applying, respectively, the FFT and IFFT algorithms:

$$S = 2^k, w = \sqrt[\frac{1}{2}S]{-1} \Rightarrow w^S = 1, \tag{34}$$

$$\{x_i\} = w^i, 0 \leq i < S, \tag{35}$$

$$FFT \equiv A_i = \sum_{k=0}^{S-1} a_k w^{ik}, 0 \leq i < S, \tag{36}$$

$$IFFT \equiv a_i = \frac{1}{S}\sum_{k=0}^{S-1} A_k w^{-ik}, 0 \leq i < S. \tag{37}$$

Fig. 6 illustrates a well-known implementation of the FFT and IFFT algorithms based on the Butterfly operator. This implementation is fully covered in the signal processing literature. In our context (HC evaluation), we only need to consider the following:

1. The Butterfly operator is computed as follows:

$$ru = lu + w^i ld, \tag{38}$$

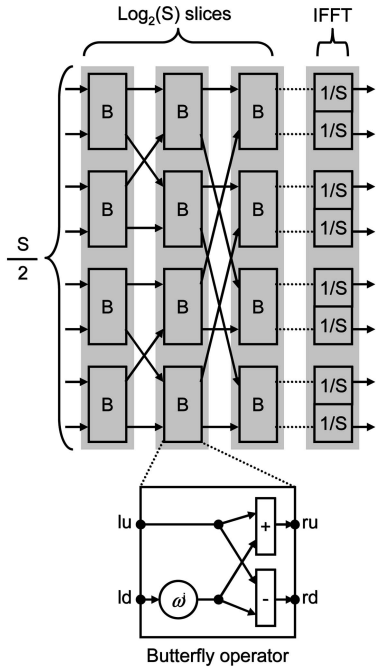

Fig. 6. FFT and IFFT implementation $(S = 8)$.

$$rd = lu - w^i ld. \tag{39}$$

2. FFT can be implemented by $\log_2(S)$ slices of $\frac{S}{2}$ instances of the Butterfly operator.
3. IFFT can be implemented in a similar way (with $\{X_i\} = w^{-i}$), but it must be followed by a slice of $\frac{1}{S}$ operators.

The FFT and IFFT algorithms rely on specific properties of unity roots, which are complex numbers in the general case. Nevertheless, complex numbers are not good candidates for implementation in hardware because operands have fixed widths and it is not possible to maintain full precision. Modular arithmetic also offers multiple unity roots, x_i, that can be exploited in these algorithms and which are better implemented in hardware. However, specific conditions must be met to guarantee that FFT can be applied. The theory behind these conditions is quite complex and out of the scope of this paper. We present here a very summarized justification. The interested reader can find a more formal justification in [18]:

1. $S = 2^k$ and S^{-1} must exist in $\mathbf{Z_m}$:
   $\Rightarrow$ modulo m must be odd.
2. a_i, $b_i \leq \beta - 1 \Rightarrow p_i \leq \frac{S}{2}(\beta - 1)^2$:
   $\Rightarrow m$ must be $> \frac{S}{2}(\beta - 1)^2$.
3. FFT and IFFT are used:
   $\Rightarrow \exists w | w^{\frac{S}{2}} = -1$.

The following simple example will help illustrate the process. We intend to multiply 4,321 by 8,765. For clarity, we will use decimal formalism, so $\beta = 10$ and $S = 8$:

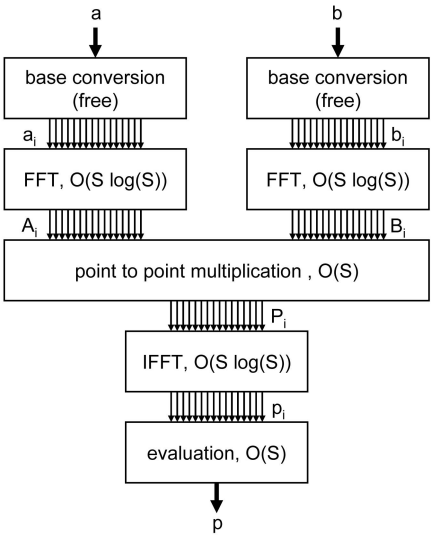

Fig. 7. FFT multiplication.

$$\begin{aligned} &m \text{ must be odd and } > 4(9)^2 = 324 \\ &\Rightarrow\ w = 6, m = w^4 + 1 = 1297 \text{ is a valid modulo} \end{aligned} \tag{40}$$

$$\begin{aligned} A(x) &= 1 + 2x + 3x^2 + 4x^3, \\ A &= FFT(1, 2, 3, 4, 0, 0, 0, 0) \\ &= (10, 985, 1, 223, 349, 1, 295, 530, 70, 734), \end{aligned}$$

$$\begin{aligned} B(x) &= 5 + 6x + 7x^2 + 8x^3, \\ B &= FFT(5, 6, 7, 8, 0, 0, 0, 0) \\ &= (26, 724, 1, 223, 1, 097, 1, 295, 1, 087, 70, 1, 003), \end{aligned} \tag{41}$$

$$\begin{aligned} P &= AB = (260, 1, 087, 288, 238, 4, 242, 1, 009, 803), \\ p_i &= (5, 16, 34, 60, 61, 52, 32, 0), \\ P(x) &= 5 + 16x + 34x^2 + 60x^3 + 61x^4 + 52x^5 + 32x^6. \end{aligned}$$

Finally, noticing that

$$A(x)B(x)|_{x=10} = 4,321 \cdot 8,765 = P(x)|_{x=10} = 37,873,565, \tag{42}$$

the reader will be easily convinced that multiplying large numbers is equivalent to multiplying polynomials. This is detailed in the following section.

5.2 FFT-Based Multiplication

We consider the multiplication of N-bit numbers. Binary numbers can be expressed in base $\beta = 2^l$ at no cost since it consists of grouping their bits per packet of l bits from the least significant bits to the most significant bits. The operation to compute $P(x)|_{x=2^l}$ (final evaluation) is a little more complex because it requires additions, but the complexity is only $O(S)$. Thus, the multiplication of two integers can have the same complexity order as the multiplication of two polynomials. The algorithm for multiplying two binary integers by FFT is described in Algorithm 8 and its combinatorial architecture is presented in Fig. 7.

Algorithm 8 FFT-based multiplication algorithm (FMA).

1: Input: a, b N-bit integers
2: Output: $p = ab$
3: define parameters l, k, and S such as
4: $S = 2^k$, $lS \geq 2N$
5: define valid modulo m and root w
6: compute $\{a_i\}$ and $\{b_i\}$
7: apply FFT to get $\{A_i\}$ and $\{B_i\}$ (in $\mathbf{Z_m}$)
8: compute $P_i = A_i B_i$ (in $\mathbf{Z_m}$)
9: apply IFFT to get $\{p_i\}$ (in $\mathbf{Z_m}$)
10: return $p = P(x)|_{x=2^l}$

Remembering that $\forall i \geq \frac{S}{2}$, $a_i = 0$ and $b_i = 0$, thus rendering the first slice of FFT useless, the HC of FMA is therefore

$$\begin{aligned} \theta_{FMA}(N) = &2\frac{S}{2}(\log(S) - 1)\theta_{Butterfly}(v) + \\ &S(\theta_{MM}(v+1) + \theta_{INV}(S)) + \\ &\frac{S}{2}\log(S)\theta_{Butterfly}(v) + \theta_{EVAL} \\ &\text{with } v \text{ defined as follows.} \end{aligned} \tag{43}$$

For the FFT-based squaring algorithm (FSA) or the FFT-based constant multiplication algorithm (FCMA), only one FFT must be computed. The complexity is reduced accordingly.

Brassard and Bratley [18] have proposed special m and w values that allow efficient implementation of the multiplier. We propose an original architecture based on these parameters that tends to minimize the global HC. This is fully detailed in the following points.

5.2.1 Brassard and Bratley Special m and w Values

These are given by

$$w = 2^r, v = r\frac{S}{2}, m = 2^v + 1, \tag{44}$$

with r chosen to satisfy $m > \frac{S}{2}(\beta - 1)^2$. This particular form leads to power-of-2 w^i defined by

$$w^i = 2^{ri} \text{ for } 0 \leq ri < \frac{S}{2}, \tag{45}$$

$$w^i = -2^{ri - \frac{S}{2}} \text{ for } \frac{S}{2} \leq ri < S. \tag{46}$$

The value w is theoretically a $(v+1)$-bit unsigned integer and so are the operations involved in FFT and IFFT. In order to simplify the hardware architecture, we propose using $(v+1)$-bit *signed* integers with redundant logic. Attention must be paid to the fact that the value 2^v is now represented by -1. All other values (except 0) have two possible representations (positive and negative).

5.2.2 Modulo Operator

Any number x expressed in base 2^v has the form

$$x = \sum_i x_i 2^{vi}. \tag{47}$$

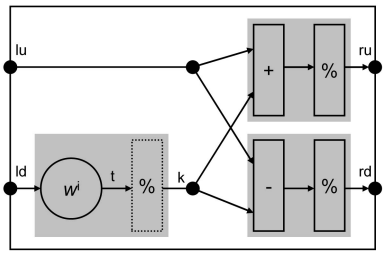

Fig. 8. The butterfly operation in modular arithmetic.

Considering that

$$2^v \bmod m = -1 \Rightarrow x = \left(\sum_i x_i(-1)^i\right) \quad \text{in } \mathbf{Z_m}. \tag{48}$$

For $(2v+1)$-bit signed integers and due to the sign bit, we have

$$x \bmod m = x[v-1\ldots 0] - x[2v-1\ldots v] - x[2v] \bmod m, \tag{49}$$

which is a $(v+1)$-bit signed integer that can be computed with a simple v-bit subtracter (with borrow) operator. The reader should note that this would not have been possible with unsigned arithmetic, which justifies our choice. Thus, the HC of the modulo operator for up to $(2v+1)$-bit (signed) operands is

$$\theta_{MOD}(w_r) = \theta_{SUBB}(v) \qquad \forall w_r \leq 2v+1. \tag{50}$$

5.2.3 The Butterfly Operator

The Butterfly operator in modular arithmetic is illustrated in Fig. 8. Thanks to our optimized parameters, the multiplication by w^i is reduced to a constant shift (from 0 to $v-1$ bits) and the negative values simply switch the adder and subtracter resources. Thus, t is actually a (maximum) $2v$-bit signed integer whose $(v+1)$ first bits are significant. The modulo operator following the multiplier is not useful anymore. The subsequent adder and subtracter can be implemented by $(v+1)$-bit adder/subtracter operators producing $(2v+1)$-bit signed integers that perfectly map the modulo operator requirements defined above. The Butterfly HC can therefore be written as

$$\theta_{BTF}(v) = \theta_{ADD}(v+1) + \theta_{SUB}(v+1) + 2\theta_{SUBB}(v), \tag{51}$$

which is a remarkable result because the complexity theoretically required for unspecified m and w^i parameters would require one MM and two modular adder/subtracter resources.

5.2.4 The Internal MM Operator and $\frac{1}{s}$ Operator

The FMA requires S instances of a $(v+1)$-bit MM. This multiplier can be recursively implemented by the FMA. Nevertheless, in this paper, we only consider single-level FMA because the operand size required for multilevel implementation is beyond what is achievable in a single combinatorial Integrated Circuit (IC). Thus, we consider that this multiplier is implemented by KMA or SMA and that it is followed by a modulo-m operator.

The multiplier takes inputs in the range $[-2^v, 2^v-1]$ and produces a result in the range $[-2^{2v}+2^v, 2^{2v}]$, which is a $(2v+2)$-bit signed integer.

In order to have a $(2v+1)$-bit signed integer (required by our modulo operator) and to anticipate the $\frac{1}{S}$ operator, we propose implementing the product's negation, followed by the modulo-m operator already described.

The implementation of the $\frac{1}{S}$ operator (for IFFT) is also very simplified in our context:

$$m = 2^v + 1 \Rightarrow 1 = -2^v(. \bmod m), \tag{52}$$

$$\frac{1}{S = 2^k} = -2^{v-k}(. \bmod m). \tag{53}$$

Since the negation is already done (in the multiplication), the $\frac{1}{S}$ operator simply becomes a (free) shift followed by our modulo-m operator. The HC is

$$\begin{aligned}\theta_{MM}(v+1) + \theta_{INV}(S) = \\ \theta_{MULT}(v+1) + \theta_{2-CPL}(v+1) + 2\theta_{MOD}(v+1).\end{aligned} \tag{54}$$

5.2.5 Final Evaluation

The final evaluation consists of computing

$$P(x) = \sum_{i=0}^{S-1} p_i x^i|_{x=2^l}, \tag{55}$$

where p_is are the outputs of the IFFT, which are modulo-m integers represented by redundant $(v+1)$-bit signed integers. The values p_i must first be converted into modulo-m integers by adding m to each negative value. This can be done with a multiplexer and a constant adder resource. The final result is obtained by a sum of shifted values, which is simply a power-of-2 sum. Noticing that $p_{S-1} = 0$, the evaluation HC is

$$\begin{aligned}\theta_{EVAL} = (S-1)(\theta_{MUX}(v+1) + \theta_{CSTADD}(v+1)) \\ + \theta_{P2Sum}(\ldots).\end{aligned} \tag{56}$$

5.3 FFT-Based MM and ME

The MSR and MPR algorithms combined with FSA and FMA are very powerful because, among the three multiplications, two involve precomputed constants. Therefore, one can also precompute the FFT of these constants and apply FCMA. The MEM combined with FMA is even better because all of the multiplications involved are square or (precomputed) constant multiplications. Algorithms 9 and 10 are, respectively, the FFT-based implementation of MPR and MEM.

Algorithm 9 FFT-based MPR (FMPR).

1: Input: $\overline{Y} = FFT(\overline{y})$, $\overline{X} = FFT(\overline{x})$
2: Output: $\overline{y \cdot x} \bmod m$
3: Constants: $m = modulo$, $r = 2^N > m$
4: Precomputed: $l = -m^{-1}$, $L = FFT(l)$, $M = FFT(m)$
5: $U = \overline{Y} \cdot \overline{X}$(point to point)
6: $u = \text{evaluate}(IFFT(U))$
7: $V = FFT(u \bmod r) \cdot L$(point to point)
8: $v = \text{evaluate}(IFFT(V)) \bmod r$
9: $W = FFT(v) \cdot M$(point to point)

TABLE 3
Atomic Device Weights for Global Hardware Complexity

Device	Weight
Gate	1
Multiplexer	3
HA	2
FA	5

10: $w = \text{evaluate}(IFFT(W))$
11: $\overline{y} = (u + w)/r$
12: **if** $(\overline{y} \geq m)$ **then**
13: $\quad \overline{y} = \overline{y} - m$
14: **end if**
15: return $\overline{y}$

Algorithm 10 FFT-based MEM (FMEM).
1: Input: x, e, m
2: Output: $y = x^e \bmod m$
3: **if** $(e = 0)$ **then**
4: $\quad$ return 1
5: **else if** $(e = 1)$ **then**
6: $\quad$ return x
7: **end if**
8: $l = -m^{-1}$ (using the extended euclidean algorithm)
9: $\overline{x} = x \cdot r \bmod m$
10: $\overline{y} = 1 \cdot r \bmod m$
11: $L = FFT(l)$
12: $M = FFT(m)$
13: $\overline{X} = FFT(\overline{x})$
14: **for** $i = N - 1$ downto 0 **do**
15: $\quad \overline{Y} = FFT(\overline{y})$
16: $\quad \overline{y} = FMPR(\overline{Y}, \overline{Y})$
17: $\quad$ **if** $(e_i = 1)$ **then**
18: $\quad\quad \overline{Y} = FFT(\overline{y})$
19: $\quad\quad \overline{y} = FMPR(\overline{Y}, \overline{X})$
20: $\quad$ **end if**
21: **end for**
22: $\overline{Y} = FFT(\overline{y})$
23: $y = FMRM(\overline{Y}, FFT(1))$
24: Return y

TABLE 4
Hardware Complexity: Standard Implementation

N	gate	mux	HA	FA	HC
Square					
16	120	0	15	105	675
32	496	0	31	465	2883
64	2016	0	63	1953	11907
128	8128	0	127	8001	48387
256	32640	0	255	32385	195075
512	130816	0	511	130305	783363
1024	523776	0	1023	522753	3139587
2048	2096128	0	2047	2094081	12570627
4096	8386560	0	4095	8382465	50307075
Multiplication					
16	256	0	16	224	1408
32	1024	0	32	960	5888
64	4096	0	64	3968	24064
128	16384	0	128	16128	97280
256	65536	0	256	65024	391168
512	262144	0	512	261120	1568768
1024	1048576	0	1024	1046528	6283264
2048	4194304	0	2048	4190208	25149440
4096	16777216	0	4096	16769024	100630528
Modular Square					
16	120	272	32	360	2800
32	496	1056	64	1488	11232
64	2016	4160	128	6048	44992
128	8128	16512	256	24384	180096
256	32640	65792	512	97920	720640
512	130816	262656	1024	392448	2883072
1024	523776	1049600	2048	1571328	11533312
2048	2096128	4196352	4096	6288384	46135296
4096	8386560	16781312	8192	25159680	184545280
Modular Multiplication					
16	256	272	33	479	3533
32	1024	1056	65	1983	14237
64	4096	4160	129	8063	57149
128	16384	16512	257	32511	228989
256	65536	65792	513	130559	916733
512	262144	262656	1025	523263	3668477
1024	1048576	1049600	2049	2095103	14676989
2048	4194304	4196352	4097	8384511	58714109
4096	16777216	16781312	8193	33546239	234868733
Modular Exponentiation					
16	376	560	65	839	6381
32	1520	2144	129	3471	25565
64	6112	8384	257	14111	102333
128	24512	33152	513	56895	409469
256	98176	131840	1025	228479	1638141
512	392960	525824	2049	915711	6553085
1024	1572352	2100224	4097	3666431	26213373
2048	6290432	8394752	8193	14672895	104855549
4096	25163776	33566720	16385	58705919	419426301

6 Implementation Results

In this section, we present the HC results obtained for several combinations of algorithms, implementations, and operand widths. These complexities are expressed in terms of atomic complexities θ_{gate}, θ_{mux}, θ_{HA}, and θ_{FA}. A global HC (computed from Table 3) is also given to enable the comparison of the different versions. These weights are based on the number of two-input gates required to implement each atomic device.

6.1 Classical Implementation HC

The HCs of square, multiplication, MS, MM, and ME are computed for the classical implementations. For ME, only one stage is computed. Results are presented in Table 4.

6.2 Karatsuba Implementation HC

The HCs of square, multiplication, MSR, MPR, and MEM are computed for the Karatsuba implementation. This means that, each time a squarer or a multiplier is involved, its complexity is computed by the Karatsuba algorithm (in a recursive way). Only one stage of MEM is computed. The reader must recall that MSR, MPR, and MEM require a Montgomery representation. Results are presented in Table 5. The rightmost column indicates the ratio of the Karatsuba implementation HC to the standard classical implementation HC (%(S)).

TABLE 5
Hardware Complexity: Karatsuba Implementation

N	gate	mux	HA	FA	HC	%(S)
Square						
16	120	0	15	105	675	100
32	496	0	31	465	2883	100
64	2016	0	63	1953	11907	100
128	6233	0	637	7250	43757	90
256	19028	0	2800	24504	147148	75
512	57685	0	10170	78967	472860	60
1024	174162	0	34036	247718	1480824	47
2048	524551	0	109139	764630	4565979	36
4096	1577542	0	341445	2336579	13943327	28
Multiplication						
16	214	0	36	218	1376	98
32	683	0	131	810	4995	85
64	2106	0	433	2739	16667	69
128	6413	0	1377	8822	53277	55
256	19388	0	4280	27648	166188	42
512	58405	0	13130	85255	510940	33
1024	175602	0	39956	260294	1556984	25
2048	527431	0	120979	789782	4718299	19
4096	1583302	0	365125	2386883	14247967	14
Montgomery Square Reduction						
16	499	16	67	460	2981	106
32	1768	32	194	1804	11272	100
64	5989	64	694	6714	41139	91
128	18492	128	2846	23169	140413	78
256	56527	256	10124	75802	456553	63
512	171706	512	33671	240444	1442804	50
1024	519407	1024	107867	748265	4479538	39
2048	1566876	2048	337827	2300326	13750304	30
4096	4718069	4096	1042970	7015306	41892827	23
Montgomery Product Reduction						
16	593	16	88	573	3682	104
32	1955	32	294	2149	13384	94
64	6079	64	1064	7500	45899	80
128	18672	128	3586	24741	149933	65
256	56887	256	11604	78946	475593	52
512	172426	512	36631	246732	1480884	40
1024	520847	1024	113787	760841	4555698	31
2048	1569756	2048	349667	2325478	13902624	24
4096	4723829	4096	1066650	7065610	42197467	18
Montgomery Exponentiation (1 stage)						
16	1092	48	155	1033	6711	105
32	3723	96	488	3953	24752	97
64	12068	192	1758	14214	87230	85
128	37164	384	6432	47910	290730	71
256	113414	768	21728	154748	932914	57
512	344132	1536	70302	487176	2925224	45
1024	1040254	3072	221654	1509106	9038308	34
2048	3136632	6144	687494	4625804	27659072	26
4096	9441898	12288	2109620	14080916	84102582	20

6.3 FFT Implementation HC

The HC is computed for the FFT implementation devices. This means that, each time a squarer or a multiplier is involved, its complexity is computed by the FFT algorithm. This algorithm is theoretically recursive because it requires S multipliers of smaller size. Nevertheless, only one level of FFT is actually implemented in our context because further levels are best implemented by the Karatsuba or the classical architectures. The FFT architecture is not as flexible as other architectures due to the close relationship between r, S, m, and N. For a given r, the HC is a step function, where steps are defined by $S = 2^k$, as illustrated in Table 6, for $r = 1, 2$ and 3. It is up to the designer to choose the best combination of r and S satisfying the required N. In this paper, we only present results for $r = 1$ to save space. The following details the case where $r = 1$ and $S = 64$:

TABLE 6
Operand Sizes for the FFTI Due to the Step Function

	$r=1$		$r=2$		$r=3$	
S	l	N	l	N	l	N
8	1	1→4	3	1→12	5	1→20
16	2	5→16	6	13→48	10	21→80
32	6	17→96	14	49→224	22	81→352
64	13	97→416	29	225→928	45	353→1440
128	29	417→1856	61	929→3904	93	1441→5952
256	60	1857→7680	124	3905→15872	188	5953→24064
512	124	7681→31744	262	15873→64512	380	24065→97280

$$S = 64 \Rightarrow m = 2^{32} + 1, \tag{57}$$

$$\beta \leq \sqrt{\frac{2m}{S}} = 11,585 \Rightarrow l \leq \log_2(\beta) = 13, \tag{58}$$

$$\text{result width} \leq 64.13 = 832, \tag{59}$$

$$\text{operand width} \leq 32.13 = 416. \tag{60}$$

The Butterfly operator HC is available in Table 7, while full implementation results are reported in Table 8. The rightmost two columns indicate the ratio of the FFT implementation HC to standard (%(S)) and to Karatsuba (%(K)) implementation HC.

6.4 Hardware Complexity Comparison

Table 9 compares the HC of square, multiplication, MM, and ME (one stage). For MM and ME, the classical implementations require the classical binary representation, whereas the Karatsuba and FFT implementations require the Montgomery representation. N values have been chosen to be the best or the worst cases for FFT implementation when $r = 1$ (due to the step function).

Results demonstrate that the Karatsuba implementation quickly produces lower complexities than the classical implementation. Any application requiring operands greater than or equal to 32 bits should replace the classical implementation with the Karatsuba implementation and, potentially, the FFT implementation.

The Karatsuba implementation and the FFT implementation compete to minimize the HC. The FFT implementation minimizes the HC of 1,856-bit operands operations, but the Karatsuba implementation minimizes the HC of 1,857-bit operands and both implementations are better than the classical implementation. The exact thresholds are presented in Table 10.

Although the Karatsuba implementation seems to offer the best complexity for up to 1,825-bit operands, one must recall that it requires a complete combinatorial or pipelined

TABLE 7
Butterfly Operator Hardware Complexity

$v+1$	gate	mux	HA	FA	HC
5	0	0	2	8	44
9	0	0	2	12	64
17	0	0	2	16	84
33	0	0	2	20	104
65	0	0	2	24	124
129	0	0	2	28	144
257	0	0	2	32	164

TABLE 8
Hardware Complexity: FFT Implementation

N	gate	mux	HA	FA	HC	%(S)	%(K)
Square							
1→4	87	35	146	442	2694	9978	9978
5→16	582	135	526	2576	14919	2210	2210
17→96	4354	527	1890	14392	81675	302	316
97→416	33793	2079	7289	82103	465123	90	139
417→1856	266241	8255	29145	491895	2808771	27	72
1857→7680	1625857	32895	247969	2954741	16994185	10	44
7681→31744	9849857	131327	1775393	17451509	101052169	3	27
Multiplication							
1→4	207	35	170	642	3862	6034	6034
5→16	1302	135	590	3904	22407	1591	1628
17→96	8066	527	2818	22648	128523	236	392
97→416	47169	2079	14329	126327	713699	69	196
417→1856	281345	8255	78681	695159	3939267	19	97
1857→7680	1679105	32895	441761	3824629	21784457	6	56
7681→31744	10048513	131327	2541857	21175797	121405193	2	33
Constant Multiplication							
1→4	207	35	154	514	3190	4984	4984
5→16	1302	135	542	3136	18471	1312	1342
17→96	8066	527	2690	18552	107787	198	328
97→416	47169	2079	14009	105847	610659	59	167
417→1856	281345	8255	77913	596855	3446211	17	85
1857→7680	1679105	32895	439969	3365877	19487113	6	50
7681→31744	10048513	131327	2537761	19078645	110911241	2	30
Montgomery Square Reduction							
1→4	501	109	456	1480	9140	5314	5050
5→16	3186	421	1612	8894	52143	1862	1749
17→96	20486	1677	7272	51782	298971	295	353
97→416	128131	6653	35309	295043	1693923	89	165
417→1856	828931	26621	184973	1691171	9734595	26	83
1857→7680	4984067	106365	1127909	9709533	56106645	9	49
7681→31744	29946883	425725	6850917	55704029	323446037	3	29
Montgomery Product Reduction							
1→4	621	109	480	1680	10308	4932	4728
5→16	3906	421	1676	10222	59631	1688	1620
17→96	24198	1677	8200	60038	345819	269	377
97→416	141507	6653	42349	339267	1942499	80	184
417→1856	844035	26621	234509	1894435	10865091	23	91
1857→7680	5037315	106365	1321701	10579421	60896917	7	53
7681→31744	30145539	425725	7617381	59428317	343799061	2	31
Montgomery Constant Product Reduction							
1→4	621	109	464	1552	9636	4611	4420
5→16	3906	421	1628	9454	55695	1576	1513
17→96	24198	1677	8072	55942	325083	253	354
97→416	141507	6653	42029	318787	1839459	76	174
417→1856	844035	26621	233741	1796131	10372035	22	87
1857→7680	5037315	106365	1319909	10120669	58599573	7	51
7681→31744	30145539	425725	7613285	57331165	333305109	2	30
Montgomery Exponentiation (1 stage)							
1→4	1122	222	920	3032	18788	4781	4571
5→16	7092	858	3240	18348	107886	1691	1608
17→96	44684	3450	15344	107724	624342	271	353
97→416	269638	13722	77338	613830	3534630	82	170
417→1856	1672966	55098	418714	3487302	20112198	23	85
1857→7680	10021382	220410	2447818	19830202	114729258	8	50
7681→31744	60092422	883194	14464202	113035194	656846378	3	30

implementation (because of recursion deployment). Such an implementation involves tens of millions of gates and this is usually not acceptable in most cryptographical circuits.

The FFT implementation produces lower complexities than the classical implementation for higher thresholds, but it offers the advantage of being implementable with a functional pipeline. A circuit composed of one modulo-m multiplier (with m defined as shown above), $\frac{3}{2}\log(S)$ modulo-m Butterfly operators, and one accumulator can perform a multiplication in S clock cycles at the price of a few extra memory elements and multiplexers to implement functional pipelining.

6.5 Preliminary FPGA Implementation

FPGAs are programmable devices capable of implementing complex designs equivalent to millions of gates. Moreover, recent FPGAs already contain hundreds of small multiplier cores that can be interconnected with custom logic to build more complex devices. Our methodology to measure a circuit's complexity cannot be applied straightforwardly to such a device because they contain atomic subdevices that are more complex than a gate, a multiplexer, an HA, or an FA. Nevertheless, if thresholds are different, complexity orders remain and this should be observable.

TABLE 9
Hardware Complexity Comparison: Standard, Karatsuba, and FFT Algorithms for Square, Multiplier, MS, MM, and ME

N	S	l	SMA (1)	KMA (2)	FMA (3)	(2)/(1) %	(3)/(1) %	(3)/(2) %
Square								
96	32	6	27075	25859	81675	96	302	316
97	64	13	27648	26492	465123	96	1682	1756
416	64	13	516675	334152	465123	65	90	139
417	128	29	519168	336353	2808771	65	541	835
1856	128	29	10323075	3908038	2808771	38	27	72
1857	256	60	10334208	3913901	16994185	38	164	434
7680	256	60	176901123	38251605	16994185	22	10	44
7681	512	124	176947200	38269299	101052169	22	57	264
Multiplication								
96	32	6	54528	32819	128523	60	236	392
97	64	13	55678	33660	713699	60	1282	2120
416	64	13	1035008	364968	713699	35	69	196
417	128	29	1039998	367385	3939267	35	379	1072
1856	128	29	20653568	4049830	3939267	20	19	97
1857	256	60	20675838	4055915	21784457	20	105	537
7680	256	60	353832960	38831701	21784457	11	6	56
7681	512	124	353925118	38849581	121405193	11	34	313
Modular Square (Standard or Montgomery representation)								
96	32	6	101280	84755	298971	84	295	353
97	64	13	103402	86746	1688181	84	1633	1946
416	64	13	1903200	1023829	1693923	54	89	165
417	128	29	1912362	1029943	9708693	54	508	943
1856	128	29	37890240	11784291	9734595	31	26	83
1857	256	60	37931082	11800019	56001831	31	148	475
7680	256	60	648798720	114819105	56106645	18	9	49
7681	512	124	648967690	114865734	323012903	18	50	281
Modular Multiplication (Standard or Montgomery representation)								
96	32	6	128733	91715	345819	71	269	377
97	64	13	131432	93914	1936757	71	1474	2062
416	64	13	2421533	1054645	1942499	44	80	184
417	128	29	2433192	1060975	10839189	44	445	1022
1856	128	29	48220733	11926083	10865091	25	23	91
1857	256	60	48272712	11942033	60792103	25	126	509
7680	256	60	825730557	115399201	60896917	14	7	53
7681	512	124	825945608	115446016	343365927	14	42	297
One stage of Modular Exponentiation (Standard or Montgomery representation)								
96	32	6	230301	176758	624342	77	271	353
97	64	13	235125	180951	3522189	77	1498	1946
416	64	13	4325981	2079722	3534630	48	82	170
417	128	29	4346805	2092169	20056077	48	461	959
1856	128	29	86116541	23715942	20112198	28	23	85
1857	256	60	86209365	23747623	114502161	28	133	482
7680	256	60	1474552317	230241346	114729258	16	8	50
7681	512	124	1474936341	230334793	655907921	16	44	285

The following results demonstrate that our approach is fully functional and give some insight into what is achievable in an FPGA. Nevertheless, the complete optimization of such an implementation is out of the scope of this paper and could be explored in a future paper.

We implemented multipliers of different sizes on a Stratix II device manufactured by Altera [19]. Each circuit has been synthesized by the QuartusII software [19]. Table 11 reports the synthesis results of a naive implementation that lets the QuartusII tool build the multiplier itself and our Karatsuba-based implementation. Due to the presence of 9-bit digital signal processor (DSP) blocks capable of implementing 18-bit multipliers, we limited the Karatsuba recursion to such multipliers. Table 12 reports the synthesis results for our FFTI. All of the circuits have been tested on random test vectors and gave successful results.

TABLE 10
Thresholds of Equal Complexity

Algorithm	K/S	FFT/S	FFT/K
Square (N)	80	969	1519
(HC)	18551	2808771	2808771
Multiply (N)	16	346	1824
(HC)	1376	713699	3939267
MM (N)	24	373	1750
(HC)	7842	1941725	10863183
ME (N)	28	376	1677
(HC)	19335	3533070	20105217

TABLE 11
Multiplier FPGA Implementation: Naive/Karatsuba

	Naive implementation			Karatsuba-based implementation		
Size	LC	9-bit DSP	Delay(ns)	LC	9-bit DSP	Delay(ns)
32x32 →64	0	8	14	149	6	20
64x64 →128	213	32	23	745	24	30
128x128 →256	5700	96	38	2828	78	44
256x256 →512	36193	288	64	9669	246	68

TABLE 12
Multiplier FPGA Implementation: Naive/FFT

	Naive implementation			FFT-based implementation		
Size	LC	9-bit DSP	Delay(ns)	LC	9-bit DSP	Delay(ns)
4x4 →16	32	0	11	544	6	31
16x16 →32	0	2	12	3141	31	46
96x96 →192	501	72	31	16788	64	73
416x416 →832	70180	768	91	79438	512	253

7 Conclusion

The complexity of squaring, multiplication, MS, MM, and ME depend on the algorithm used to implement them. the Karatsuba implementation and the FFT implementation are well known in software applications but seem to be rarely used in hardware due to recursion (for the Karatsuba implementation) and too large integer requirements (for the FFT implementation).

We have performed an in-depth analysis of the HC involved by Karatsuba and FFT-based implementations of the cited operators. We have also proposed a very optimized implementation of FFT-based operators. Results show that recursion deployment leads to efficient implementation for as small as 16-bit operands (Karatsuba implementation), whereas our optimized FFT implementation starts to outperform the classical implementation for 346-bit operands for simple multiplication. For 1,856-bit integers, one stage of ME in FFTI only requires 23 percent of the classical implementation hardware for the same stage.

Current cryptographical applications require 1,024-bit operands (and beyond) and the need for larger keys continuously increases. We think that the FFT implementation will take an important place in future design, at least for functionally pipelined architectures or specialized arithmetic logic units (ALUs, with dedicated modulo-m operations) coupled to standard processors.

References

[1] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Trans. Information Theory,* vol. 22, no. 6, pp. 644-654, 1976.

[2] P. Montgomery, "Modular Multiplication without Trial Division," *Math. Computation,* vol. 44, no. 170, pp. 519-521, 1985.

[3] K. Posch and R. Posch, "Modulo Reduction in Residue Number Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 6, no. 5, pp. 449-454, 1995.

[4] J. Bajard, L. Didier, and P. Kornerup, "An RNS Montgomery's Modular Multiplication Algorithm," *IEEE Trans. Computers,* vol. 47, no. 2, pp. 167-178, Feb. 1998.

[5] J. Bajard, L. Didier, and P. Kornerup, "Modular Multiplication and Base Extensions in Residue Number Systems," *Proc. 15th IEEE Symp. Computer Arithmetic (ARITH '01),* pp. 59-65, 2001.

[6] J.-J. Quisquater and C. Couvreur, "Fast Decipherment Algorithm for RSA Public-Key Cryptosystem," *Electronics Letters,* vol. 18, pp. 905-907, 1982.

[7] J. Grosschädl, "The Chinese Remainder Theorem and Its Application in a High-Speed RSA Crypto Chip," *Proc. 16th Ann. Computer Security Application Conf. (ACSAC '00),* pp. 384-393, 2000.

[8] S.E. Eldridge and C.D. Walter, "Hardware Implementation of Montgomery's Modular Multiplication Algorithm," *IEEE Trans. Computers,* vol. 42, no. 6, pp. 693-699, June 1993.

[9] H. Orup, "Simplifying Quotient Determination in High-Radix Modular Multiplication," *Proc. 12th Symp. Computer Arithmetic (ARITH '95),* pp. 193-199, 1995.

[10] T. Blum and C. Paar, "High-Radix Montgomery Modular Exponentiation on Reconfigurable Hardware," *IEEE Trans. Computers,* vol. 50, pp. 759-764, 2001.

[11] A.F. Tenca and Ç.K. Koç, "A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm," *IEEE Trans. Computers,* vol. 52, no. 9, pp. 1215-1221, Sept. 2003.

[12] C. McIvor, M. McLoone, J.V. McCanny, A. Daly, and W. Marnane, "Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architectures," *Proc. 37th Ann. Asilomar Conf. Signals, Systems, and Computers,* 2003.

[13] C. McIvor, M. McLoone, and J.V. McCanny, "Modified Montgomery Modular Multiplication and RSA Exponentiation," *IEE Proc.—Computers and Digital Techniques,* vol. 151, pp. 402-408, 2004.

[14] V. Bunimov, M. Schimmler, and B. Tolg, "A Complexity-Effective Version of Montgomery's Algorithm," *Proc. 29th Ann. Int'l Symp. Computer Architecture (ISCA '02) Workshop Complexity Effective Designs,* 2002.

[15] A.A. Karatsuba and Y. Ofman, "Multiplication of Multidigit Numbers on Automata," *Soviet Physics Doklady,* vol. 7, pp. 595-596, 1963.

[16] J.W. Cooley and J.W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Computation,* vol. 19, pp. 297-301, 1965.

[17] S. Craven, C. Patterson, and P. Athanas, "Super-Sized Multiplies: How Do FPGAs Fare in Extended Digit Multipliers?" *Proc. Seventh Ann. Conf. Military and Aerospace Programmable Logic Devices (MAPLD '04),* 2004.

[18] G. Brassard and P. Bratley, *Algorithmics Theory and Practice.* Prentice Hall, 1988.

[19] Altera, Altera Corp., http://www.altera.com, 2006.

Jean Pierre David received the degree in electrical engineering from the Université de Liège, Belgium, in 1995 and the PhD degree from the Université Catholique de Louvain, Belgium, in 2002. He has been an assistant professor at the Université de Montréal for three years and is currently an assistant professor at the École Polytechnique de Montréal. His research interests include reconfigurable computing, high-level synthesis, hardware-software codesign, and their applications. He is a member of the IEEE.

Kassem Kalach received the bachelor's degree in applied mathematics from the Lebanese University in 1996 and the master's degree in computer science from the University of Montreal in 2006. His research interests include cryptography, smart cards, and computer security.

Nicolas Tittley received the bachelor's degree from the Université de Montréal in 2006. His research interests include grid computing, network availability, and advanced algorithms.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.