Generation of Finely-Pipelined GF(P ) Multipliers for Flexible Curve based Cryptography on FPGAs

Gabriel Gallin, Arnaud Tisserand

To cite this version:
Gabriel Gallin, Arnaud Tisserand. Generation of Finely-Pipelined GF(P ) Multipliers for Flexible Curve based Cryptography on FPGAs. IEEE Transactions on Computers, Institute of Electrical and Electronics Engineers, 2019, 68 (11), pp.1612-1622. 10.1109/TC.2019.2920352. hal-02141260

HAL Id: hal-02141260
https://hal.archives-ouvertes.fr/hal-02141260
Submitted on 27 May 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Generation of Finely-Pipelined GF($P$) Multipliers for Flexible Curve based Cryptography on FPGAs

Gabriel Gallin and Arnaud Tisserand, Senior Member, IEEE

Abstract—In this paper, we present modular multipliers for hardware implementations of (hyper-)elliptic curve cryptography on FPGAs. The prime modulus $P$ is generic and can be configured at run-time to provide flexible circuits. A finely-pipelined architecture is proposed for overlapping the partial products and reductions steps in the pipeline of hardwired DSP slices. For instance, 2, 3, or 4 independent multiplications can share the hardware resources at the same time to overlap internal latencies. We designed a tool, distributed as open source, for generating VHDL codes with various parameters: width of operands, number of logical multipliers per physical one, speed or area optimization, possible use of BRAMs, target FPGA. Our modular multipliers lead to, at least, 2 times faster as well as 2 times smaller circuits than state of the art operators.

Index Terms—Modular arithmetic, Montgomery multiplication, elliptic curve cryptography, arithmetic operator generation.

I. INTRODUCTION

Designing efficient modular arithmetic is key for implementing asymmetric cryptography. In such applications, multiplication is the most important operation for speed and cost purpose. Several algorithms and architectures have been proposed. Here, we deal with modular multipliers for GF($P$) used in implementations of (hyper-)elliptic curve cryptography ((H)ECC) on embedded systems. In our cryptographic applications, we target generic primes (i.e. $P$ has a non-specific binary representation) and field sizes about 128 bits for HECC and 256 bits for ECC. We do not consider field sizes smaller than 100 bits or larger than 400 bits for our applications. In order to provide flexible operators, the modulus $P$ can be defined at run time. This allows us to easily support various curves and applications in one circuit.

We focus on FPGAs embedding hardwired DSP slices dedicated to operations such as $a \times b \pm c$ over small integers (typically 10 to 20 bits). To exploit them at a high frequency, one has to use the dedicated pipeline registers inside each DSP slice. This can be tricky in iterative algorithms where the number of operations per iteration is smaller than the pipeline depth (leading to idle stages in the schedule). Hyper-threading has been proposed in this type of situation, see for instance [11]. One physical unit is simultaneously shared by several logical units for overlapping internal latencies.

In [2], we proposed finely-pipelined modular multipliers (FPMMs) for 128-bit HECC designed by hand for a few FPGAs (the URL is given in the references Section). Below, we extend these results with new optimizations, a wider parameter space and more supported FPGAs. The parameter space is so wide that we decided to design a tool, distributed as open source [3], to generate optimized FPMMs in VHDL. Compared to [2], our tool allows us to explore multipliers with:

- flexible prime $P$ defined at run time ($P$ was fixed in [2]);
- fewer DSP slices and a shorter computation time;
- a higher ratio between the operations throughput and the area cost compared to state of the art solutions;
- a higher frequency (close to the maximum allowed by the technology, e.g. 502 MHz upon 550 MHz in Virtex-5);
- a larger parameter space at design time: width of operands, number of logical multipliers per physical one, more supported FPGAs, potential use of BRAMs, and new area/speed optimizations.

Thanks to our finely-pipelined architecture, various optimizations and our exploration tool, we can propose more efficient multipliers than the state of the art. As an example, for a 128-bit prime modulus on a Virtex-7 FPGA, our best multiplier (code name F44D below) only requires 9 DSP slices, 600 LUTs and 151 clock cycles at 633 MHz to compute 8 modular multiplications (239 ns). As a comparison, the best operator from the state of the art (code name MA16) requires 21 DSP slices, 1182 LUTs, and 167 clock cycles at 350 MHz (478 ns). At least, our most efficient solutions are 2 times faster as well as 2 times smaller than state of the art one for typical applications in HECC.

Section II recalls the state of the art. Our FPMM operators and tool are described in Sections III and IV respectively. Implementation results and comparisons are reported and analyzed in Section V. Finally, Section VI concludes the paper.

Notations:

- modular multiplication $A \times B \mod P$ with $P$ prime;
- $n$ the width of $P$ (e.g. 128, 256 bits);
- $m$ the width of field elements ($m > n$) in Montgomery domain decomposed into $s$ words of $w$ bits ($m = sw$);
- $\sigma$ the number of logical multipliers (LMs) per FPMM;
- Clock cycles are denoted cycles and abbreviated cc;
- $\lambda$ the multiplication latency (i.e. duration in cc);
- $\tau$ the interval between 2 multiplications in a LM (in cc);
- LSW (MSW) stands for least (most) significant word.

II. STATE OF THE ART IN MODULAR MULTIPLICATION

In this work, we do not consider specific modulus forms such as (pseudo)-Mersenne primes where the reduction can be performed using a few shifts and additions. These specific
primes lead to fast (H)ECC implementations (see for instance [4]), but they are limited to only one field characteristic. Our target applications require the support of several curves and fields then we only deal with generic primes.

Several algorithms and architectures for modular multiplications with generic primes have been proposed. Some solutions use the interleaved algorithm from Blakley [5]: e.g. [6, 7] and [8]. Other solutions use the reduction method from Barrett [9]: e.g. [10]. Montgomery Modular Multiplication (MMM) [11] algorithm and its numerous variants are widely used for both software and hardware implementations.

Thanks to operands and results represented in Montgomery domain, MMM performs the modular reduction by the help of two multiplications, a few additions, and one shift: e.g. [11], [12]. In [13], Walter proposed to remove the final subtraction in the original MMM by enlarging the domain such that the internal parameter \( R \) is \( R > 4P \) (instead of \( R > 2P \)).

In [14], five MMM variants are compared. Among them, the Coarsely Integrated Operand Scanning (CIOS) algorithm was proposed for software implementations with small computation time and memory requirements, see Algo. 1. CIOS efficiently interleaves partial products and partial reductions steps in a regular decomposition using high-radix subwords (while Blakley algorithm uses interleaving but with very small subwords). Efficient CIOS implementations on FPGAs have been proposed in [15], [16], [17] using DSP slices. Among MMM solutions, the CIOS algorithm, and its variants, are widely used in cryptographic applications.

The high-radix algorithms proposed in [12] relax data dependencies inside iterations and simplify the quotient computation by enlarging the internal datapath with a wider Montgomery domain. Later, [18] and [19] used this idea to design FPGA implementations with increased internal parallelism and reduced computation time. The 256-bit ECC processor over generic \( GF(P) \) presented in [19] is one of the fastest on FPGAs. Their multiplier requires 37 DSP slices. The reported frequency is 250 or 291 MHz for Virtex-4 or Virtex-5 respectively (the authors state that the critical path is in the multiplier), but no detailed area metrics are reported.

As the results from [19] are very good, we reproduced their multiplier on more FPGAs and other sizes. In our first work [2], we only implemented the 128-bit version of the multiplier from [19]. Since this first work, we added a 256-bit version of their multiplier. Our implementation results of the multipliers from [19] are reported in Table II and denoted MA16. They are very close to the original results from [19]. For 256 bits, we also need 37 DSP slices, and we have very close frequencies on the same FPGAs. Only our internal schedule is slightly different: for one MMM our implementation requires 37 cycles instead of 35, but the interval between 2 multiplications is reduced to 28 cycles instead of 29. Our re-implementation of the multiplier from [19] is similar to the original one. Thanks to all the details provided in [19], we were able to reproduce their results with a good accuracy. This was not possible for other works.

Iterative Digit-Digit Montgomery Multiplication (IDDMM) was proposed in [20] for 256-bit to 1024-bit generic primes. This solution was inspired from CIOS. For comparisons, we use their results for 256-bit multipliers on Virtex-5 with 64-bit internal datapath (solution denoted below MO64).

In [21], a modified IDDMM was proposed on Virtex 7. For comparisons, we use their results for 256-bit multipliers with 32-bit (denoted AM32) and 64-bit (AM64) internal datapaths. Systolic architectures have been proposed for MMM in [22] and frequently optimized: [23], [24] and [25]. For comparisons, we use some results from [26] where systolic CIOS implementations are reported for 128 and 256-bit multipliers. Internal decompositions into 8 and 16 words have been proposed and are denoted below MR8 and MR16.

Using the CIOS and FIOS (Finely Integrated Operand Scanning) algorithms from [14], the work presented in [27] recently proposed area-efficient MMMs for ECC on low-power IGLOO 2 FPGAs. For comparisons, we use their results with CIOS (denoted MAS1) and FIOS (denoted MAS2). Their units embed both MMM and modular addition/subtraction.

Other solutions have been proposed for ASIC implementations where small multipliers (such as DSP slices) cannot be used efficiently. For instance, the Multiple-Word Radix-2 Montgomery Multiplication (MWR2MM) proposed in [28] uses iterations with products of very small 2 or 4-bit words by the full multiplicand. The works [29], [30], [31] and [32] provide interesting results for ASIC implementations. Below, we only target FPGA implementations with DSP slices. Then, we do not consider this type of MMM variant.

III. FINELY-PIPELINED MODULAR MULTIPLIERS

A. Fine pipeline for speeding-up \( GF(P) \) multipliers

We use the CIOS algorithm from [14] without final subtraction from [13], see Algo. 1 because it is simple and regular. In the outer loop (index \( i \)), the internal computations have sequential dependencies which are difficult to map in DSP slices without “bubbles” in the pipeline at a high frequency. Two typical solutions in the literature mitigate this dependency problem: i) using a more complex algorithm with relaxed dependencies (e.g. [12]); ii) reusing some hardware resources in different tasks with a more complex control (e.g. [27]).

Internal values are represented on \( m \) bits, instead of \( n \) for “raw” \( GF(P) \) elements due to the Montgomery domain. Based on the CIOS algorithm from [14] without final subtraction from [13], we select \( R > 4P \) (where \( R = 2^m \)). Then \( m \) is slightly larger than \( n \). In most of FPGAs, hardwired units are not wide enough for cryptographic sizes (128 or 256 bits in our applications). In our target FPGAs, BRAMs words are at most 36-bit wide while DSP slices are 18 × 18 or 18 × 25 bits units. Then processing each \( GF(P) \) element would require many parallel BRAMs and DSP slices. But using many BRAMs is useless since the number of intermediate \( GF(P) \) elements is small in ECC and HECC (up to 10 and 20 respectively). In FPPM, we decompose \( m \)-bit elements into \( w \)-bit smaller words processed sequentially (\( w \ll m \)). We denote \( s \) the number of \( w \)-bit words required for each operand with \( m = sw \). This processing scheme efficiently fits DSP slices and BRAMs in our FPGAs. It also reduces the interconnect area in a complete cryptoprocessor. Finally, it leads to higher clock frequencies.
We use CIOS for its simplicity and regularity properties combined to a finely-pipelined architecture to overlap internal latencies in the hardwired pipeline of DSP slices. A physical FPMM supports \( \sigma \) logical multipliers for independent MMMs simultaneously. In FPMM, operand loading and result output are limited to one LM per cycle (see Figure 1).

The behavior for \( \sigma = 3 \) LMs and \( s = 2 \) words per element is illustrated in Figure 1 (C means cycle):

- **C1**: load \( w \)-bit words \((a_0, b_0)\) for the first operation;
- **C2**: load \((a_1, b_1)\) and start \( A \times B \) in LM1;
- **C3**: load \((c_0, d_0)\) for the second operation;
- **C4**: load \((c_1, d_1)\) and start \( C \times D \) in LM2;
- **C5**: load \((e_0, f_0)\) for the last operation;
- in next cycles, all LMs compute \( \sigma \) independent products;
- **C6**: output the \( w \)-bit word \( AB_0 \) of the result;
- **C\& + 1**: output the word \( AB_1 \) of the result;
- the next 4 cycles output products \( CD \) and \( EF \).

After loading the operands (e.g. \( a_0, \ldots, a_{s-1}, b_0, \ldots, b_{s-1} \)), all pipeline stages in each DSP slice perform intermediate computations for all independent MMMs at different clock cycles.

We denote \( \lambda \) the latency between the loading of the first word of the operands and the output of the last word of the result (i.e. \( \delta+1 \) cycles in our example).

The interval between two successive MMMs in the same LM, denoted \( \tau \), is less than \( \lambda \). New operands can be loaded in the first stage of the pipeline while the last stages are finishing the current MMM. In our example on Figure 1, this corresponds to \( \tau = \delta - 1 \) cycles.

In the CIOS Algo. 1. the computations at iteration \( i \) of the outer loop are decomposed into 3 dependent tasks:

- **Task 1**: lines 3–5 compute the multiplication of word \( a_i \) by \( B \) (with accumulation of previous iteration \( i-1 \));
- **Task 2**: line 6 computes the “reduction quotient” \( q_i \) for the Montgomery algorithm;
- **Task 3**: lines 8–10 compute the product \( q_i \times P \) (with an accumulation of the value computed in task 1).

Between task 3 at iteration \( i \) and task 1 at iteration \( i + 1 \), we discard the least significant word (see Sec. III-A for implementation details) accordingly to the Montgomery algorithm.

Our FPMM architecture, presented in Fig. 2, is designed to fit this decomposition into 3 tasks. Each task is handled by a dedicated block.

With pipelined DSP slices, the result from task 3 is not immediately available in task 1 for the next iteration \( i + 1 \). We need to wait for the pipeline depth. Our FPMM solution is inspired from hyper-threading to overlap this delay. With \( \sigma \) independent MMMs in one FPMM, we can always fill all stages without any “bubbles” in the pipeline. Hyper-threading principle is illustrated in Fig. 3 for a 3-stage iterative pipeline.

In the top of Fig. 3 only 1 MMM (in green) is computed in the pipeline, leading to idle stages. In the bottom of Fig. 3 idle stages are used to compute 2 other independent MMMs (in red and blue) at the same time and all stages are used after the initial latency.

**B. Selection of parameters \( s \) and \( w \)**

Both \( s \) and \( w \) impact the performances. \( s \) is the number of iterations in both outer and inner loops in Algo. 1. In our
case, the minimal possible value for $s$ is 2 (this ensures the correct use of the CIOS principle). A larger $s$ leads to a larger latency $\lambda$ (it grows as $O(s^2)$). The decomposition into $w$ bits requires $w \times w$ bits partial products. $w$ must be as large as possible to efficiently fill the operand width in the DSP slices and reduce $s$. Currently, we do not consider “rectangular” DSP configurations such as $18 \times 25$ bits (we plan to see how we can exploit those asymmetric operators in the future). We use $18 \times 18$ bits DSP slices (designed for two’s complement). For unsigned integers, one has to set the MSB to 0 and only use the 17 LSBS (as stated in the FPGA documentation). Then $w$ should be 17 bits or a multiple of 17 to use the full capacity of the DSP slices. We explored several values for $w$: 17, 34, 51 and 68 bits, with respectively 1, 4, 9 and 16 DSP slices for $w \times w$ bits partial products.

For $w = 17$ bits, we need only one DSP slice per block (3 in total for one FPMM). One BRAM is large enough to store both operands (e.g. $A$ and $B$) for each LM. But the latency $\lambda$ is very large due to a large $s$.

For $w$ larger than 36 bits, we need several BRAMs per operand (with a very low memory usage, for instance HECC with $w = 68$ only requires 2 words). $w$ larger or equal to 51 bits leads to huge circuits (at least 9 DSP slices per block) and a lower frequency. Then we do not consider large $w$ parameters.

We select $w = 34$ bits. We need 4 DSP slices in each block for tasks 1 and 3 as illustrated in Fig. 2. A complete FPMM requires 11 DSP slices. All words are small enough to fit into one single BRAM. The capacity of the smallest target BRAM is 9 Kb on Spartan FPGAs. The 2 BRAMs in a FPMM allow to store more than 200 $w$-bit words. They can store at least 50 $w$-bit words and $w$ is 9 Kb on Spartan FPGAs. The 2 BRAMs in a FPMM allow to store more than 200 $w$-bit words. They can store at least 50 $w$-bit words and $w$ is 9 Kb on Spartan FPGAs.

Internal loops in tasks 1 and 3 are processed in $s$ cycles. One extra cycle is required for final carry propagation in each task: line 5 for task 1 and line 10 for task 3 in Algo. 1. We used this $s + 1$ cycles schedule in our previous work [2]. In Section III-F we propose a new improvement to remove this extra cycle in tasks 1 and 3.

For the block dedicated to task 2, we showed in [2] that only 3 DSP slices are required to compute $q_i$ due to the reduction modulo $2^w$. In Section III-F we propose an improved version of this hardware block with only one DSP slice.

### C. Selection of parameter $\sigma$

Parameter $\sigma$ must be at least greater than 2 for pipelining several logical multiplications in the operator. The outer loop (index $i$) in Algo. 1 task 1 $\rightarrow$ task 2 $\rightarrow$ task 3 $\rightarrow$ task 1, colored in red on Figure 2 has a duration of $\alpha$ cycles. The minimal value for $\alpha$ is 15 cycles, coming from the FPMM internal architecture and configuration of DSP slices in hardware blocks for tasks 2 and 3. This value only depends on words size $w$. It does not depend on the field size (e.g. 128, 256 bits) and on the number $s$ of words.

The inner loops in tasks 1 and 3 require $s + 1$ cycles each to load and start the partial products. In order to fill all stages without “bubbles”, one needs to select $\sigma$ such that:

$$\sigma = \left\lceil \frac{\alpha}{s + 1} \right\rceil.$$  

If $\sigma$ is too big, result from task 1 must be delayed to wait until task 1 starts all intermediate computations in all LMs. Then $\sigma(s + 1) - \alpha$ extra registers must be inserted. This would lead to a larger FPMM without any speed gain. If $\sigma$ is too small, there are $\alpha - \sigma(s + 1)$ “bubbles” in the pipeline. This leads to under-utilizing some hardware resources.

The 128-bit architecture, $s = 4$ and $w = 34$, without the improvement from Section III-F, is illustrated in Fig. 2. Fortunately, $s + 1 = 5$ leads to a minimal $\sigma = 3$ without need to increase $\alpha$ (no extra register).

For 256-bit, $s = 8$ and $w = 34$, the minimal $\alpha$ is not a multiple of $s + 1 = 9$. To stay close to the minimal $\alpha = 15$, two $\sigma$ values are possible: 1 and 2. With $\sigma = 1$, we have the same architecture but 6 “bubbles” in the pipeline (40% of time is then wasted). With $\sigma = 2$, we have to add 3 extra $w$-bit registers between task 3 and task 1. Then there is no “bubble” in the pipeline.

Parameter $\sigma = 3$ or 2, respectively for 128 or 256-bit MMM, is the smallest possible value. However, a larger $\sigma$ (and larger circuit due to extra registers) may lead to a better speed-area trade-off. We explore different values for $\sigma$ in Section IV.

### D. Differences with our first FPMM published in [2]

Our first FPMM [2] was manually implemented only for 128-bit GF($P$) fields for the fixed parameters $w = 34$ bits, $s = 4$ words and $\sigma = 3$ LMs. The prime $P$ was a constant fixed at design time. Implementations were done on Virtex-4, Virtex-5 and Spartan-6 FPGAs. It was more efficient than state the field size ($P$)

The current paper presents the tool (see Section IV) we developed for generating various FPMMs, in VHDL, with several major improvements:

- Various widths $n$ of GF($P$) elements specified at design time (128 and 256 bits examples are reported below);
- Flexible prime $P$ defined at run time for a fixed GF($P$) width $n$ (see subsection III-F);
- Various values of the parameter $\sigma$;
- Area reduction with 9 DSP slices instead of 11 (see optimization in subsection III-F);
Input: \( A = \sum_{i=0}^{s-1} a_i 2^{iw} \), \( B = \sum_{j=0}^{s-1} b_j 2^{jw} \), \( P = \sum_{j=0}^{s-1} p_j 2^{jw} \) such that \( 0 \leq A, B < 2P \).

Requires: \( 4P < 2^m \) and \( -P^{-1} \equiv (v_0 \times P') \mod 2^w \) for \( j = 0 \) to \( s - 1 \) do
\[
(c, t_{j-1}) \leftarrow u_j + q_i \times p_j + c
\]
\( t_{s-1} = t(n)_{-1} \)
return \( T = \sum_{j=0}^{s-1} t_j 2^{jw} \)

- At the end of task 3 (line 7 in Algo. 2):
\[
a_i \times B + T_{i-1} + q_i \times P + u^{(p)}_i + c^{(p)} \equiv t^{(n)}_{s-1} \mod 2^w
\]

Where all variables with \( (p) \) as exponent are the values computed in the previous independent MMM.

Input: \( A = \sum_{i=0}^{s-1} a_i 2^{iw} \), \( B = \sum_{j=0}^{s-1} b_j 2^{jw} \), \( P = \sum_{j=0}^{s-1} p_j 2^{jw} \) such that \( 0 \leq A, B < 2P \).

Requires: \( 4P < 2^m \) and \( P' = -P^{-1} \mod 2^w \)

Output: \( T \equiv (A B 2^{-m}) \mod P, 0 \leq T < 2P \)
begin
for \( i = 0 \) to \( s - 1 \) do
for \( j = 0 \) to \( s - 1 \) do
\[
(d, u_j) \leftarrow v_j \mod 2^w
\]
for \( j = 0 \) to \( s - 1 \) do
\[
(c, t_{j-1}) \leftarrow u_j + q_i \times p_j + c
\]
\( t_{s-1} = t(n)_{-1} \)
return \( T = \sum_{j=0}^{s-1} t_j 2^{jw} \)

Algorithm 2: Proposed CIOS algorithm without final subtraction (based on [14] and [13]) modified for latency optimization. Value \( t^{(n)}_{s-1} \) in line 8 is the value of \( t_{s-1} \) computed at iteration \( i \) in the next LM.

In the original CIOS Algo. 1, computations of \( w \times m \) bits partial products \( (a_i \times B \) and \( q_i \times P) \) in inner loops (index \( j \)) require 4 clock cycles, plus 1 extra clock for carry propagation (words of index \( j = s = 4 \)).

In our optimized Algo. 2, the result \( t_{s-1} \) of the first LM at iteration \( j = 4 \) and the result \( t_{s-1} \) of the second LM at iteration \( j = 0 \) share the same \( w \)-bit word. This optimization is possible as the LSW \( t_{s-1} \) of the result is always zero and is discarded in MMM. We use this word during each outer loop iteration to store intermediate values for the next LM. In the following, \( (c) \) and \( (v) \) denote respectively the values computed in the current and next LM.

Below, we show that sharing one \( w \)-bit word between successive LMs to store two different intermediate values throughout intermediate computations does not modify the expected results. We need to verify that \( t^{(n)}_{s-1} = t^{(c)}_{s-1} \) thanks to the properties of the MMM algorithm. We also verify that overlapping the MSW \( t_{s-1} \) of the current LM with the LSW \( t_{s-1} \) of the next LM does not create “hidden” carries between results of successive independent MMMs.

In Algo. 2, the carry \( d \) and \( c \) are propagated between successive LMs. As in CIOS Algo. 1, we define \( u_s = d \) after the last iteration of the first inner loop (lines 2 to 4 in Algo. 2), and \( t_{s-1} = u_s + c \) after the last iteration of the second inner loop (lines 6 and 7 in Algo. 2). To compute \( T_i \) the current LM, we need the value \( t^{(n)}_{s-1} \), computed at iteration \( j = 0 \) of the second inner loop in the next LM:
\[
t^{(n)}_{s-1} = \left( u^{(n)}_0 + q^{(n)}_i \times p_j + c^{(c)} \right) \mod 2^w.
\]
Due to the propagation of carry words $u_s$ between successive LMs, we have
\[ u_0(n) = \left( T_{i-1}^{(n)} + a_i^{(n)} \times b_0^{(n)} + u_s^{(c)} \right) \mod 2^w. \] (2)

However, the “reduction quotients” $q_i$ are computed before the propagation of the $u_s$ carry words, and then
\[ q_i^{(n)} = \left( T_{i-1}^{(n)} + a_i^{(n)} \times b_0^{(n)} \times P^i \right) \mod 2^w. \] (3)

Putting together equations (1), (2) and (3), we obtain:
\[ t_{i-1}^{(n)} = \left( T_{i-1}^{(n)} + a_i^{(n)} \times b_0^{(n)} + u_i^{(c)} \times p_0 + c^{(c)} \right) \mod 2^w. \]
\[ = \left( T_{i-1}^{(n)} + a_i^{(n)} \times b_0^{(n)} + q_i^{(n)} \times p_0 \right) \]
\[ + \left( u_s^{(c)} + c^{(c)} \right) \mod 2^w. \] (4)

Due to Montgomery algorithm, the LSW is equal to 0:
\[ T_{i-1} + a_i \times b_i + q_i \times p_0 \equiv 0 \mod 2^w. \]

Then, from equation (4), the value $t_{i-1}^{(n)}$ is now
\[ t_{i-1}^{(n)} = \left( u_s^{(c)} + c^{(c)} \right) \mod 2^w. \]

Value $u_s^{(c)}$ is computed as:
\[ u_s^{(c)} = \left[ \frac{T_i^{(c)} + a_i^{(c)} \times B^{(c)} + u_s^{(p)}}{2^m} \right]. \]

From properties of MMM algorithm, we have:
\[ \left\{ \begin{array}{l}
2^m > 4P \\
0 \leq B, T_{j-1} < 2P < 2^{m-1} \\
0 \leq a_i < 2^w
\end{array} \right. \] (5)

Then the value of $u_s^{(c)}$ is bounded by
\[ 0 \leq u_s^{(c)} < 2^{w-1}. \] (6)

Similarly, the value of $c$ is
\[ c^{(c)} = \left[ \frac{\sum_{j=0}^{s-1} \left( u_j^{(c)} \times 2^j \right) + u_s^{(p)} + a_i^{(c)} \times P + c^{(p)}}{2^m} \right]. \]

From bounds in Eq. (5) we have:
\[ 0 \leq c^{(c)} \leq 2^{w-2}. \] (7)

We now use the bounds for $u_s^{(c)}$ and $c^{(c)}$ in equations (6) and (7) to bound the value of $t_{i-1}^{(n)}$:
\[ 0 \leq t_{i-1}^{(n)} < 2^w. \]

We then have:
\[ t_{i-1}^{(n)} = \left( u_s^{(c)} + c^{(c)} \right) \mod 2^w \]
\[ = u_s^{(c)} + c^{(c)} \]
\[ = t_{s-1}^{(c)}. \]

and the result of MMM is
\[ T^{(c)} = t_{i-1}^{(n)} \times 2^{(s-1)w} + \sum_{j=0}^{s-2} t_j^{(c)} \times 2^j w \]
\[ = \sum_{j=0}^{s-1} t_j^{(c)} \times 2^j w, \]

which is less than $2^m$ as $0 \leq t_j \leq 2^w, \forall j \in [0, s-1]$. Results for independent MMMs in successive LMs then do not overlap, and are the same as in original algorithm.

This optimization removes the “bubble” but requires a modified architecture and control to deal with the overlap indicated at line 8 of Algo. 2 and illustrated in Fig. 4. Fortunately, it leads to a more regular control without any measurable area overhead. The latency reduction impacts the FPMM pipeline configuration. We now have: $\sigma = \lceil \alpha / (s + 1) \rceil$.

G. Optimization of the DSP slices configuration

All the DSP slices in [2] are configured with a 3-stage pipeline as illustrated in Fig. 2. However, the first DSP slice in task 1 (top left in Fig. 2) does not use the hardwired adder.

We provide two FPMM variants depending on the configuration of the first DSP slice:

- **Fast version** with a 3-stage pipeline to reach a higher frequency (only this version was available in [2]);
- **Small version** with only 2-stage pipeline. The total latency decreases by one cycle, but the frequency in the DSP slice also drops a little. This optimization reduces the delay for operands loading in the 3 next DSP slices and allows us to remove 3 $w$-bit registers.

Both variants have some interest depending on the parameters and the target FPGA (see examples in Sec. 4).

H. Flexible P definition at run time

Unlike most of MMM literature solutions, our new FPMM allows us to change $P$ at run time. The parameters $n$, $m$, $w$, and $s$ are fixed at design time, but $P$ and $P'$ can be defined and loaded at run time in the configured FPGA.

Our new FPMM has now two running modes:

- **Setup**: $P$ and $P'$ are loaded and all LMs are reset;
- **Run**: MMMs are performed as described above for the current $P$.

FPMM in “run” mode reads 3 $w$-bit words from $P$ at every clock cycle in task 3. This would require at least 2 $w$-bit dual-port BRAMs (with a very few words). Storing $P$ in the operands BRAMs is not possible since they are used at every cycle for the LMs. Only $w$ bits are required to store $P'$.

We chose to store $P$ in $w$ shift registers of depth $s$ bits (each one uses only one LUT since $s$ is small). For $P'$, we use a single $w$-bit register (in regular flip-flops).

In “setup” mode, $P$ and $P'$ are loaded into dedicated registers (respectively using $A$ and $B$ FPMM inputs). This only requires $s$ cycles and a few additional LUTs and flip-flops.
I. Validation

At the arithmetic level, the only difference between our FPMM and the original CIOS without final addition is the latency reduction presented and proved in Sec. III-F. At the architecture and implementation levels, we performed very intensive simulations for the validation of our operators (see Sec. IV-V). We used predefined operands and millions of random ones with success.

IV. VHDL Generator

We developed a tool dedicated to the automatic generation of FPMMs for many parameter combinations. Our tool, available on Unix platforms as open source [3], is based on Bash scripts and Python programs.

The generator input is a text file with the complete FPMM specification: width $w$ of operands, internal decomposition parameters $(s, w)$, number $\sigma$ of LMs, target FPGA, operand storage in BRAMs or DRAMs, enable/disable latency reduction optimization (see Sec. III-F), fast or small version (see Sec. III-G). After verification of the specification consistency, the generator produces a set of VHDL files ready for implementation of the selected FPMM on the target FPGA. The generator also reports some computed properties such as $\tau$ and $\lambda$.

In order to help design space exploration, we also provide in [3] scripts for synthesis, implementation and validation steps. The complete flow, illustrated in Fig. 5 (with steps in numbered circles), currently supports the following Xilinx tools: ISE 14.7, ISim simulator and SmartXplorer.

Steps 1 and 2 respectively correspond to the user specification and VHDL generation.

FPGA synthesis and implementation are performed in step 3. SmartXplorer selects the best implementation after many runs of the place and route tool (set to 100 by default).

Step 4 validates the mathematical and functional behavior of the produced FPMM through very intensive simulations (with millions of random and selected operands). The theoretical expected results are computed by the internal math library from the Sage open source mathematical software available at the URL [http://www.sagemath.org/]. This library uses state of the art software algorithms to compute the modular multiplication. In order to intensively test the generated operators, our tool also generates VHDL codes for simulating the obtained FPMMs and automatically compares their results to the mathematical reference from Sage at bit level and cycle level.

Finally, the last step produces a final report with the main implementation (time and area) and validation results.

Users can easily adapt our generator and flow for their CAD tools and target FPGAs.

V. Implementation Results and Comparisons

A. Exploration of FPMMs

![Fig. 5. FPMM generator, in green, and its utilization flow. Commercial CAD tools are in blue and the Sage mathematical program in grey.](image)

**Algorithm 3:** Exploration of FPMM parameters for a given maximal size $n$ of primes $P$. $\Lambda(k)$ is the total latency for $k$ MMMs in FPMM (in cc).

```plaintext
Input : $n$, version $\in \{\text{fast, small}\}$
Output : FPMM parameters and performances for various $m$, $w$, $s$, $\sigma$, $\alpha$
begin
  if version = fast then
    $\varepsilon \leftarrow 8$
  else
    $\varepsilon \leftarrow 7$
  for $w \in \{17, 34, 51, 68\}$ do
    $s \leftarrow \lceil \frac{w+2}{\varepsilon} \rceil$
    if latency_reduction then
      $\theta \leftarrow s$
    else
      $\theta \leftarrow s + 1$
    $m \leftarrow s \times w$
    $\sigma_{\min} \leftarrow 2 \times \lceil \frac{w}{17} \rceil^2 + 4 \times \lceil \frac{w}{17} \rceil - 1$
    $\sigma_{\min} \leftarrow \lceil \frac{\sigma_{\min}}{w} \rceil$
    for $i = 0$ to $4$ do
      $\sigma \leftarrow \sigma_{\min} + i$
      $\alpha \leftarrow \sigma \times \theta$
      $\lambda \leftarrow \alpha \times \sigma \times (s - 1) + \alpha_{\min} + s + \varepsilon$
      $\tau \leftarrow \sigma \times s$
      $\Lambda(k) \leftarrow \tau \left\lfloor \frac{s-1}{\tau} \right\rfloor + \theta \left( k - 1 \mod \sigma \right) + \lambda$
      for target $\in \{V4, V5, S6, V7\}$ do
        IMPLEMENT(version, $w$, $s$, $\sigma$, $\theta$, target)
        Report results in Database
end
```

In Algo. 3, we describe the exploration of FPMM parameters for a given size $n$ of prime $P$ and a given configuration of the DSP slices (i.e. fast or small, see Sec. III-G). In this algorithm, $\varepsilon$ is the delay in clock cycles between the loading of the operands words $a_0$ and $b_0$ and computing the first $u_0$ at end of task 1 in one LM. Delay $\varepsilon$ is composed by:
- 2 cc to write $a_0$ and $b_0$ to operands memories;
- 2 cc to read $a_0$ and $b_0$ from operands memories to Task 1;
- 3/2 cc in first DSP in Task 1, in fast/small version resp.;
- 1 cc to compute $u_0$.

For each size $n$ of $P$, we explored several architectures for FPMM, based on various decompositions of operands into $w$-bit words (line 5 to 21 in Algo. 3). From width $w$ of
words, FPMM version and the selected optimisations, we can
determine the main characteristics of each explored FPMM:
• width $m$ of operands in FPMM (line 11 in Algo. [3]);
• minimal duration $\alpha_{min}$ of the outer loop (line 12 in
Algo. [3]);
• minimal number $\sigma_{min}$ of LMs (line 13 in Algo. [3]).

We implemented explored FPMMs on 4 different FPGAs
for various numbers $\sigma$ of LMs (line 14 to 21 in Algo. [3]).
We also evaluated their performances in terms of latency $\lambda$
(line 17 in Algo. [3]) and computation time $\Delta$ for 8 MMMs.

### B. FPGA specific optimizations of FPMM

We implemented many FPMMs on Virtex-4, 5, 7 and
Spartan-6 from Xilinx: denoted V4, V5, V7, S6 for
XC4VLX100, XC5VLX110T, XC7VX690T and XC6SLX75.
These FPGAs embed different types of DSP slices, listed in
Table I. Reported maximum frequencies come from Xilinx
official documentation: [33] for V4; [34] for V5; [35] for S6;

<table>
<thead>
<tr>
<th>FPGA</th>
<th>type of DSP slice</th>
<th>FPGA reference</th>
<th>speed grade</th>
<th>max. freq. MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtex-4</td>
<td>DSP48E1</td>
<td>XC4VLX100</td>
<td>-12</td>
<td>500</td>
</tr>
<tr>
<td>Virtex-5</td>
<td>DSP48E</td>
<td>XC5VLX110T</td>
<td>-3</td>
<td>550</td>
</tr>
<tr>
<td>Spartan-6</td>
<td>DSP58A1</td>
<td>XC6SLX75</td>
<td>-3</td>
<td>590</td>
</tr>
<tr>
<td>Virtex-7</td>
<td>DSP48E1</td>
<td>XC7VX690T</td>
<td>-3</td>
<td>741</td>
</tr>
</tbody>
</table>

Fig. 6. LUTs vs. computation time (8 MMMs) trade-offs for 128-bit FPMMs
on V7 (top) and V4 (bottom). Red lines are Pareto fronts of trade-offs space.
Table II
Implementation results for a selection of the most interesting 128 and 256-bit FPMMs.

<table>
<thead>
<tr>
<th>FPGA Spec.</th>
<th>slice / LUT / FF</th>
<th>BRAM</th>
<th>DSP freq. MHz</th>
<th>( \lambda )</th>
<th>time [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>F44B</td>
<td>512 / 424 / 766</td>
<td>2⊕ 9</td>
<td>387 75</td>
<td>391</td>
<td></td>
</tr>
<tr>
<td>S44B</td>
<td>486 / 389 / 745</td>
<td>2⊕ 9</td>
<td>320 74</td>
<td>468</td>
<td></td>
</tr>
<tr>
<td>F48B</td>
<td>257 / 397 / 728</td>
<td>2⊕ 9</td>
<td>483 75</td>
<td>313</td>
<td></td>
</tr>
<tr>
<td>S44B</td>
<td>242 / 369 / 703</td>
<td>2⊕ 9</td>
<td>397 74</td>
<td>378</td>
<td></td>
</tr>
<tr>
<td>S44B</td>
<td>241 / 382 / 757</td>
<td>0</td>
<td>349 75</td>
<td>433</td>
<td></td>
</tr>
<tr>
<td>F48B</td>
<td>203 / 321 / 671</td>
<td>2⊕ 9</td>
<td>320 74</td>
<td>469</td>
<td></td>
</tr>
<tr>
<td>S44D</td>
<td>199 / 416 / 721</td>
<td>0</td>
<td>334</td>
<td>449</td>
<td></td>
</tr>
<tr>
<td>F48B</td>
<td>325 / 545 / 725</td>
<td>2⊕ 9</td>
<td>528 75</td>
<td>286</td>
<td></td>
</tr>
<tr>
<td>S44B</td>
<td>306 / 600 / 758</td>
<td>0</td>
<td>633</td>
<td>239</td>
<td></td>
</tr>
<tr>
<td>S48B</td>
<td>287 / 523 / 683</td>
<td>2⊕ 9</td>
<td>481 74</td>
<td>312</td>
<td></td>
</tr>
</tbody>
</table>

Table III
Results for 128 and 256-bit standalone multipliers from recent literature (* denotes estimation based on paper values).

<table>
<thead>
<tr>
<th>ref.</th>
<th>FPGA</th>
<th>slice / LUT / FF</th>
<th>BRAM</th>
<th>DSP freq. MHz</th>
<th>( \lambda )</th>
<th>time [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>MA16</td>
<td>V4</td>
<td>879 / 1201 / 1311</td>
<td>6⊕ 9</td>
<td>252 27</td>
<td>663</td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td>540 / 1600 / 1280</td>
<td>6⊕ 9</td>
<td>210</td>
<td>795</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V7</td>
<td>455 / 1182 / 1305</td>
<td>6⊕ 9</td>
<td>350</td>
<td>478</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MR8</td>
<td>A7</td>
<td>206 / 255 / 487</td>
<td>0</td>
<td>19 33</td>
<td>1333*</td>
<td></td>
</tr>
</tbody>
</table>

For 8 MMMS and 128-bit generic primes on V7, our best FPMM (F44D) is 2 times faster than the best state of the art multiplier (MA16). F44D is also much smaller than MA16: 9 extra cycles.

Figure 8 depicts the schedule for 8 independent MMMS. On the right, we report the number of DSP slices, BRAMs, logic slices, LUTs, and FFs, as well as the frequency. Green ratios in the last column are normalized w.r.t. the F44D results. The area variations among S44B, F44B and F44D are smaller than \( \pm 10\% \). MA16 is larger and slower. Figure 8 clearly illustrates why our finely-pipelined architecture is efficient for multiple independent MMMS.

In Table IV we present the achievable throughput (MM\( \cdot \)s\(^{-1}\)) per area ratio (TPAR) as a global efficiency metric for state-of-the-art multipliers and some of our most efficient FPMMs. For instance, respectively 129k and 37k FPMMs can be computed per second per logic slice for our FPMM and for MA16 from [19] on Virtex-7 and 128-bit operands. Figure 9 illustrates TPAR for MA16 from [19] and our best trade-offs on Virtex-7 and 128-bit operands. In this figure, the further from center means the higher TPAR, and thus the best hardware efficiency. Our finely-pipelined solution clearly leads to arithmetic units with a better utilization.
of the hardware resources without “bubbles” in the pipeline for simple and regular algorithms such as the CIOS.

For 256 bits, only the MA16 multiplier is at most 1.5 times faster but it requires 4.1 times more DSP slices and 3.9 times more LUTs. All other multipliers are both slower and larger.

### D. Applications to HECC

In [37], we used our first 128-bit FPMMs from [2] to design cryptoprocessors for HECC over GF($P$). Below, we present new results for these HECC cryptoprocessors where we use our new flexible FPMMs. Table V provides implementation results for 3 cryptoprocessors based on the architectures from [37] and our new F44B FPMM (see above). These architectures are composed with

- arithmetic units: adder(s)/subtractor(s), F44B FPMM(s);
- 1 or 2 central data memories to store intermediate values;
- 1 interconnect between memories and arithmetic units;
- 1 central control unit with a program memory.

Architecture A2 corresponds to a small architecture with only 1 adder/subtractor unit, 1 FPMM and 1 data memory. A3 corresponds to a parallel architecture with 2 adder/subtractor units, 2 FPMMs and 1 data memory. Finally, A4 is a clustered architecture, with 2 adder/subtractor units, 2 FPMMs and 2 data memories (see [37] for schematics). In Table V, $\tilde{w}$ is the width of the interconnect. More details on these architectures can be found in [37].

Thanks to our FPMMs, our HECC cryptoprocessors reach high frequencies with reduced area compared to the best equivalent state-of-the-art ECC cryptoprocessors. For example, compared to the very efficient ECC solution for generic primes from [19], our smallest cryptoprocessors are at least 2 times smaller for the same computation time.

Compared to our first HECC cryptoprocessors published in [37], our new architectures based on the new F44B are both smaller and faster.

### VI. Conclusion and Future Prospects

We propose finely-pipelined modular multipliers (FPMMs) for FPGA implementations with DSP slices. FPMM internal control, inspired from hyper-threading, allows us to more efficiently fill the DSP slices pipeline. Our first FPMMs presented in [2] were limited to 128 bits and led to larger and slower circuits. In this paper, we propose more advanced FPMMs and a tool for generating them for various sizes...
and FPGAs, leading to both faster and smaller circuits. For instance, we get 2 to 3 times smaller designs for the same speed compared to the state of the art [19]. Our generator is available as open source at [3] (with some VHDL codes and implementation results for many parameters).

In the future, we plan to extend our tool to support other FPGA vendors and rectangular DSP slices (e.g. 25 × 18 bits). For other types of applications, we also plan to investigate other field sizes (e.g. n < 100 or n > 400 bits) with other w values. Finally, we plan to study FPMM architectures with protections against side channel and fault injection attacks.

ACKNOWLEDGMENT

This work was done in the HAH project (http://www.h-a-h.cominlabs.u-bretagne-loire.fr/) partially funded by Labex CominLab, Labex Lebesgue and Brittany Region. We sincerely thank Xilinx for University Program donations. We also thank the anonymous Reviewers for their valuable comments.

REFERENCES

Arnaud Tisserand (PhD’97, M’00, SM’06) is senior researcher at CNRS (French National Center for Scientific Research) in computer science in Lab-STICC laboratory. His research interests include computer arithmetic, computer architecture, digital security, VLSI and FPGA design, design automation, low-power design and applications in applied cryptography, scientific computing, digital signal processing. He is senior member of the IEEE (SSCS, CAS).