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, 2019, 68 (11), pp.1612-1622. ⟨10.1109/TC.2019.2920352⟩. ⟨hal-02141260⟩

HAL Id: hal-02141260

https://hal.science/hal-02141260v1

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.

HAL Authorization

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 $\mathrm{GF}(P)$ 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 [1]. 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 $A \times B \bmod P$ 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 ($m = sw$);
- σ the number of *logical multipliers* (LMs) per FPMM;
- Clock cycles are denoted cycles and abbreviated cc;
- λ the multiplication latency (*i.e.* duration in cc);
- τ 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

G. Gallin is with CNRS, IRISA UMR 6074, University Rennes and INRIA Centre Rennes - Bretagne Atlantique, France.

A. Tisserand is with CNRS, Lab-STICC UMR 6285 and University South Brittany, in Lorient, France. (arnaud.tisserand@univ-ubs.fr).

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 $\mathrm{GF}(P)$ 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 III 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 $\mathrm{GF}(P)$ 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" $\mathrm{GF}(P)$ 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 $\mathrm{GF}(P)$ element would require many parallel BRAMs and DSP slices. But using many BRAMs is useless since the number of intermediate $\mathrm{GF}(P)$ elements is small in ECC and HECC (up to 10 and 20 respectively). In FPMM, 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 σ *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 σ independent products;
- Cδ: output the w-bit word AB_0 of the result;
- C$\delta + 1$: output the word AB_1 of the result;
- the next 4 cycles output products CD and EF.

After loading the operands (*e.g.* $a_{0,\ldots,s-1}, b_{0,\ldots,s-1}$), all pipeline stages in each DSP slice perform intermediate computations for all independent MMMs at different clock cycles.

We denote λ 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 τ, is less than λ. 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;

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 $P' = -P^{-1} \bmod 2^w$
Output : $T \equiv (A\, B\, 2^{-m}) \bmod P$, $0 \leq T < 2P$
begin

$t_{0\ldots s-1} \leftarrow 0$
1 **for** $i = 0$ **to** $s - 1$ **do**
2 $d \leftarrow 0$
3 **for** $j = 0$ **to** $s - 1$ **do**
4 $(d, u_j) \leftarrow t_j + a_i \times b_j + d$
5 $u_s \leftarrow d$ } Task 1
6 $q_i \leftarrow (u_0 \times P') \bmod 2^w$ } Task 2
7 $c \leftarrow 0$
8 **for** $j = 0$ **to** $s - 1$ **do**
9 $(c, t_{j-1}) \leftarrow u_j + q_i \times p_j + c$
10 $(c, t_{s-1}) \leftarrow u_s + c$ } Task 3
11 **return** $T = \sum_{j=0}^{s-1} t_j\, 2^{jw}$

Algorithm 1: CIOS algorithm without final subtraction (based on [14] and [13]).

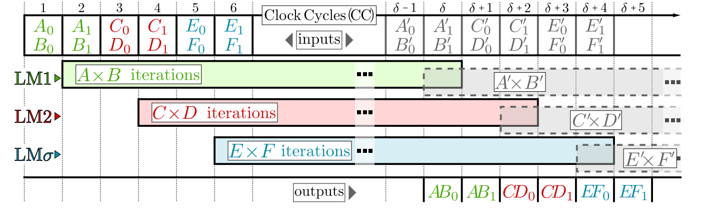

Fig. 1. Illustration of operands transfers and computations in FPMM for $\sigma = 3$ and $s = 2$. Operands pairs (A, B), (C, D) and (E, F) and resulting products AB, CD and EF are decomposed into w-bit words.

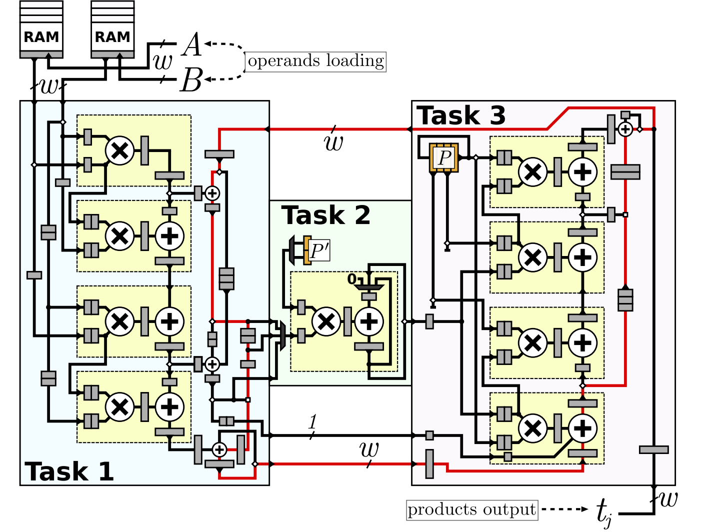

Fig. 2. FPMM architecture, without all control details, for $\sigma = 3$ and $s = 4$ (*i.e.* 128-bit MMM). Blue, green and purple boxes are hardware blocks for each task in Algo. 1. Yellow boxes are DSP slices (with internal pipeline), and grey boxes are registers. Critical path for the outer loop is in red.

- 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-F 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 σ 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

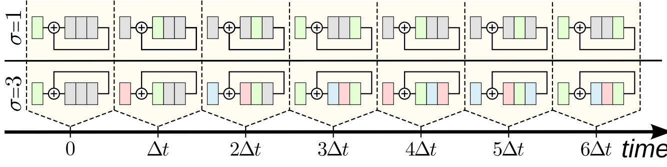

Fig. 3. Illustration of hyper-threading principle in a 3-stage iterative pipeline for $\sigma = 1$ (top) and $\sigma = 3$ (bottom). Colors represent different independent operations.

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 λ (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 18×25 bits (we plan to see how we can exploit those asymmetric operators in the future). We use 18×18 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 λ 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 pairs of 128-bit HECC operands or 25 pairs for 256-bit ECC. In practice, the number of logical multipliers σ is way smaller than those bounds (see Subsection III-C). Setting $w = 34$ bits is the most efficient solution in our FPGAs for both 128-bit HECC ($s = 4$) and 256-bit ECC ($s = 8$).

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-E, we propose an improved version of this hardware block with only one DSP slice.

C. Selection of parameter σ

Parameter σ 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 α cycles. The minimal value for α 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 σ such that:

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

If σ is too big, result from task 3 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 σ 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 α (no extra register).

For 256-bit, $s = 8$ and $w = 34$, the minimal α is not a multiple of $s + 1 = 9$. To stay close to the minimal $\alpha = 15$, two σ 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 σ (and larger circuit due to extra registers) may lead to a better speed-area trade-off. We explore different values for σ in Section V.

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 of the art. But it was not possible to explore various sizes and parameters by hand.

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-H);
- Various values of the parameter σ;
- Area reduction with 9 DSP slices instead of 11 (see optimization in subsection III-E);

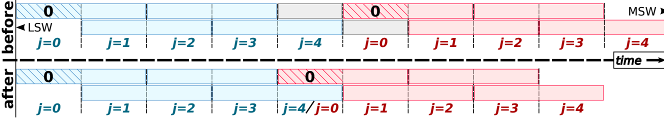

Fig. 4. Schedule example for 2 MMMs (colors) *before* and *after* latency reduction (rectangles are w-bit words computed at iterations of index j).

- Reduced latency λ of $s \times s$ cycles instead of $s \times (s+1)$ (see optimization in subsection III-F);
- Frequency increased up to 20 % (and now close to the maximal frequency of the DSP slices or BRAMs) with improved control;
- Alternative architectures inside block for task 1 (see subsection III-G);
- More supported FPGA families.

As in [2], we still support architectures with either BRAMs or DRAMs for operands memories.

E. Reduction of the number of DSP slices in block 2

The block dedicated to task 2 computes the "reduction quotient" q_i from the Montgomery algorithm for each iteration of the outer loop (line 6 in Algo. 1). One $w \times w$ bits partial product modulo 2^w must be computed for each q_i. For $w = 34$, this only requires 3 DSP slices instead of 4 due to the reduction modulo 2^w (see [2] for details). These 3 DSP slices were used with different configurations.

We analyzed the schedule of all internal operations. We were able to optimize the control and spread the three 17×17 multiplications at different cycles in only one DSP slice without any penalty on the global latency α. Now the only DSP slice uses 3 different configurations at different cycles. The DSP slice inputs are also selected among several possible internal values at the right cycle by the new internal control.

This optimization reduces the number of DSP slices from 3 to 1 in block 2 (from 11 to 9 in a complete FPMM). It requires a few new registers (less than 10 bits in total), 2 new w-bit multiplexers, but it does not reduce the frequency.

F. Latency reduction

In the FPMM from [2], inner loops in tasks 1 and 3 both require s cycles for operand loading plus 1 extra cycle for the carry propagation into the MSW for task 1 (u_s at line 5 in Algo. 1) and task 3 (t_{s-1} at line 10). It introduces a "bubble" in the DSP slices pipeline at the transition between 2 consecutive MMMs in 2 LMs as illustrated on top of Fig. 4.

We propose a modification in the control to remove this "bubble" as illustrated on the bottom of Fig. 4. Our modification targets the carry propagation in lines 5 and 10 in Algo. 1. Modified CIOS algorithm is described in Algo. 2, and the new control now produces:

- At the end of task 1 (line 4 in Algo. 2):

$$a_i \times B + T_{i-1} + u_s^{(\mathrm{p})}$$

- At the end of task 2 (line 5 in Algo. 2):

$$q_i = \left((a_i \times B + T_{i-1}) \bmod 2^w \times P' \right) \bmod 2^w$$

- At the end of task 3 (line 7 in Algo. 2):

$$\underbrace{a_i \times B + T_{i-1} + q_i \times P}_{T_i \times 2^w} + \underbrace{u_s^{(\mathrm{p})} + c^{(\mathrm{p})}}_{t_{s-1}^{(\mathrm{p})}}$$

Where all variables with $^{(\mathrm{p})}$ 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 $0 \le A, B < 2P$
Requires: $4P < 2^m$ and $P' = -P^{-1} \bmod 2^w$
Output : $T \equiv (A\,B\,2^{-m}) \bmod P$, $0 \le T < 2P$
begin

```
    t_{0...s-1} ← 0; d ← 0; c ← 0
1   for i = 0 to s − 1 do
2       for j = 0 to s − 1 do                    } Task 1
3           v_j ← t_j + a_i × b_j
4           (d, u_j) ← v_j + d
5       q_i ← (v_0 × P') mod 2^w                 } Task 2
6       for j = 0 to s − 1 do                    } Task 3
7           (c, t_{j−1}) ← u_j + q_i × p_j + c
8       t_{s−1} = t_{−1}^(n)
9   return T = Σ_{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 $t_{-1}^{(\mathrm{n})}$ in line 8 is the value of t_{-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_{-1} of the second LM at iteration $j = 0$ share the same w-bit word. This optimization is possible as the LSW t_{-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, $^{(\mathrm{c})}$ and $^{(\mathrm{n})}$ 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 $t_{-1}^{(\mathrm{n})} = t_{s-1}^{(\mathrm{c})} < 2^w$ 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_{-1} of the next LM does not create "hidden" carries between results of successive independent MMMs.

In Algo. 2, carries 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 in the current LM, we need the value $t_{-1}^{(\mathrm{n})}$, computed at iteration $j = 0$ of the second inner loop in the next LM:

$$t_{-1}^{(\mathrm{n})} = \left(u_0^{(\mathrm{n})} + q_i^{(\mathrm{n})} \times p_0 + c^{(\mathrm{c})} \right) \bmod 2^w. \qquad (1)$$

Due to the propagation of carry words u_s between successive LMs, we have

$$u_0^{(\mathrm{n})} = \left(T_{i-1}^{(\mathrm{n})} + a_i^{(\mathrm{n})} \times b_0^{(\mathrm{n})} + u_s^{(\mathrm{c})}\right) \bmod 2^w. \tag{2}$$

However, the "reduction quotients" q_i are computed before the propagation of the u_s carry words, and then

$$q_i^{(\mathrm{n})} = \left((T_{i-1}^{(\mathrm{n})} + a_i^{(\mathrm{n})} \times b_0^{(\mathrm{n})} \times P'\right) \bmod 2^w. \tag{3}$$

Putting together equations (1), (2) and (3), we obtain:

$$\begin{aligned} t_{-1}^{(\mathrm{n})} &= (T_{i-1}^{(\mathrm{n})} + a_i^{(\mathrm{n})} \times b_0^{(\mathrm{n})} + u_s^{(\mathrm{c})} + q_i^{(\mathrm{n})} \times p_0 + c^{(\mathrm{c})}) \bmod 2^w \\ &= \Big((T_{i-1}^{(\mathrm{n})} + a_i^{(\mathrm{n})} \times b_0^{(\mathrm{n})} + q_i^{(\mathrm{n})} \times p_0) \\ &\quad +(u_s^{(\mathrm{c})} + c^{(\mathrm{c})})\Big) \bmod 2^w. \end{aligned} \tag{4}$$

Due to Montgomery algorithm, the LSW is equal to 0:

$$T_{i-1} + a_i \times b_0 + q_i \times p_0 \equiv 0 \bmod 2^w.$$

Then, from equation (4), the value $t_{-1}^{(\mathrm{n})}$ is now

$$t_{-1}^{(\mathrm{n})} = \left(u_s^{(\mathrm{c})} + c^{(\mathrm{c})}\right) \bmod 2^w.$$

Value $u_s^{(\mathrm{c})}$ is computed as:

$$u_s^{(\mathrm{c})} = \left\lfloor \frac{T_{i-1}^{(\mathrm{c})} + a_i^{(\mathrm{c})} \times B^{(\mathrm{c})} + u_s^{(\mathrm{p})}}{2^m} \right\rfloor.$$

From properties of MMM algorithm, we have:

$$\begin{cases} 2^m > 4P \\ 0 \le B, T_{i-1} < 2P < 2^{m-1} \\ 0 \le a_i < 2^w \end{cases} \tag{5}$$

Then the value of $u_s^{(\mathrm{c})}$ is bounded by

$$0 \le u_s^{(\mathrm{c})} < 2^{w-1}. \tag{6}$$

Similarly, the value of c is

$$c^{(\mathrm{c})} = \left\lfloor \frac{\sum_{j=0}^{s-1} \left(u_j^{(\mathrm{c})}\, 2^{jw}\right) + u_s^{(\mathrm{p})} + q_i^{(\mathrm{c})} \times P + c^{(\mathrm{p})}}{2^m} \right\rfloor.$$

From bounds in Eq. (5) we have:

$$0 \le c^{(\mathrm{c})} \le 2^{w-2}. \tag{7}$$

We now use the bounds for $u_s^{(\mathrm{c})}$ and $c^{(\mathrm{c})}$ in equations (6) and (7) to bound the value of $t_{-1}^{(\mathrm{n})}$:

$$0 \le t_{-1}^{(\mathrm{n})} < 2^w.$$

We then have:

$$\begin{aligned} t_{-1}^{(\mathrm{n})} &= \left(u_s^{(\mathrm{c})} + c^{(\mathrm{c})}\right) \bmod 2^w \\ &= u_s^{(\mathrm{c})} + c^{(\mathrm{c})} \\ &= t_{s-1}^{(\mathrm{c})}, \end{aligned}$$

and the result of MMM is

$$\begin{aligned} T^{(\mathrm{c})} &= t_{-1}^{(\mathrm{n})}\, 2^{(s-1)w} + \sum_{j=0}^{s-2} t_j^{(\mathrm{c})}\, 2^{jw} \\ &= \sum_{j=0}^{s-1} t_j^{(\mathrm{c})}\, 2^{jw}, \end{aligned}$$

which is less than 2^m as $0 \le t_j < 2^w$, $\forall j \in [0, s-1]$. 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: $\sigma = \lceil \alpha/s \rceil$ cycles instead of $\lceil \alpha/(s+1) \rceil$.

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. V).

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.

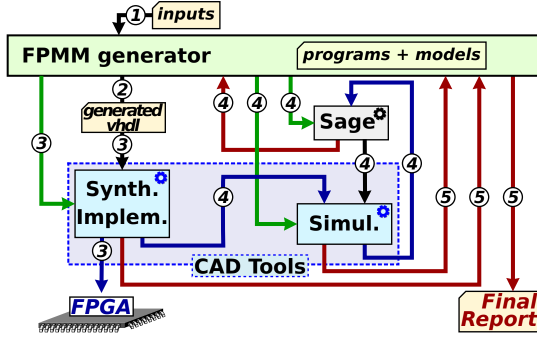

Fig. 5. FPMM generator, in green, and its utilization flow. Commercial CAD tools are in blue and the Sage mathematical program in grey.

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). 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 m of operands, internal decomposition parameters (s, w), number σ of LMs, target FPGA, operand storage in BRAMs or DRAMs, enable/disable latency reduction optimization (see III-F), fast or small version (see 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 τ and λ.

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 compare 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

Input : n, *version* $\in \{fast, small\}$
Output : FPMM parameters and performances for various m, w, s, σ, α
begin
1 **if** $version = fast$ **then**
2    $\varepsilon \leftarrow 8$
3 **else**
4    $\varepsilon \leftarrow 7$
5 **for** $w \in \{17, 34, 51, 68\}$ **do**
6    $s \leftarrow \left\lceil \frac{n+2}{w} \right\rceil$
7   **if** *latency_reduction* **then**
8      $\theta \leftarrow s$
9   **else**
10      $\theta \leftarrow s + 1$
11    $m \leftarrow s \times w$
12    $\alpha_{\min} \leftarrow 2 \times \left\lceil \frac{w}{17} \right\rceil^2 + 4 \times \left\lceil \frac{w}{17} \right\rceil - 1$
13    $\sigma_{\min} \leftarrow \left\lceil \frac{\alpha_{\min}}{w} \right\rceil$
14   **for** $i = 0$ **to** 4 **do**
15      $\sigma \leftarrow \sigma_{\min} + i$
16      $\alpha \leftarrow \sigma \times \theta$
17      $\lambda \leftarrow \alpha \times \sigma \times (s-1) + \alpha_{\min} + s + \varepsilon$
18      $\tau \leftarrow \alpha \times s$
19      $\Lambda(k) \leftarrow \tau \left\lfloor \frac{k-1}{\sigma} \right\rfloor + \theta\ (k - 1 \bmod \sigma) + \lambda$
20   **for** *target* $\in \{\mathrm{V4}, \mathrm{V5}, \mathrm{S6}, \mathrm{V7}\}$ **do**
21   `IMPLEMENT`$(version, w, s, \sigma, \theta, target)$
22   `Report results in Database`

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).

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, ε 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 ε 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 α_{min} of the outer loop (line 12 in Algo. 3);
- minimal number σ_{min} of LMs (line 13 in Algo. 3).

We implemented explored FPMMs on 4 different FPGAs for various numbers σ of LMs (line 14 to 21 in Algo. 3), We also evaluated their performances in terms of latency λ (line 17 in Algo. 3) and computation time Λ 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; and [36] for V7.

Internal pipelines of DSP slices in our FPMMs are optimized for each FPGA in order to reach the highest frequency allowed by the hardware. The best configurations of DSP slices for each FPMM implemented on a specific FPGA are selected after implementation of various possible internal pipelines in each of these DSP slices.

C. Discussions and Comparisons

Below we report implementation results for some 128 and 256 bits multipliers (more complete results are available in our generator website [3]). For area metrics, we report the number of DSP slices, logic slices, LUTs, flip-flops (FFs) and BRAMs. For timing aspects, we report the latency λ for one MMM (in cycles), the clock frequency and the computation time for 8 independent MMMs to illustrate internal parallelism benefits (based on the HECC application from [37]).

FPMM specifications are abbreviated by 1 letter, 2 digits and 1 letter: i) F/S for fast/small version; ii) σ in $\{3,4\}$ or $\{2,3,4\}$ resp. for $n = 128$ or 256 bits; iii) s or $s{+}1$ cycles for the delay between 2 LMs with or without latency reduction; iv) B/D for BRAM/DRAM operands storage. For instance, F45B means a fast version with $\sigma = 4$ LMs, $s+1 = 5$ cycles between LMs (*i.e.* no latency reduction) and BRAMs.

Figure 6 presents area/time trade-offs for all 128-bit FPMMs on V4 and V7. Remember that all FPMMs use $w = 34$ bits and require 9 DSP slices. On V7, the smallest FPMM is S44B while F44B and F44D are the fastest ones with and without BRAMs. On V4, the smallest one is still S44B, the fastest

TABLE I
PROPERTIES OF DSP SLICES EMBEDDED IN SELECTED FPGAS.

FPGA	type of DSP slice	FPGA reference	speed grade	max. freq. MHz
Virtex-4	DSP48	XC4VLX100	-12	500
Virtex-5	DSP48E	XC5VLX110T	-3	550
Spartan-6	DSP48A1	XC6SLX75	-3	390
Virtex-7	DSP48E1	XC7VX690T	-3	741

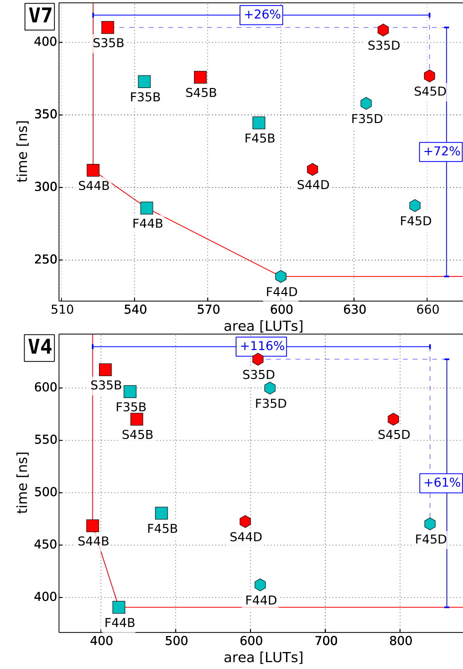

Fig. 6. LUTs *vs.* computation time (8 MMMs) trade-offs for 128-bit FPMMs on V7 (top) and V4 (bottom). Red lines are Pareto fronts of trade-offs space.

one is F44B but now F44D is not anymore on the Pareto front. Without our generator, it would be difficult to identify the best specification for one application.

When exploring FPMMs for different FPGA families, it is also very difficult to predict the impact of some parameters. On a recent V7, the fastest solution uses DRAMs while on an older V4, it uses BRAMs. On V7 (based on LUT-6), the area variations among all FPMMs specifications are only 26 % but 116 % on V4 (based on LUT-4).

Table II reports a selection of the most interesting FPMMs on more FPGAs for both 128 and 256-bit finite fields. Bold values indicate the best trade-off metrics and BRAM capacity is denoted $^{\ominus}$ for 9Kb and $^{\oplus}$ for 18Kb.

Our generator helps cryptographic applications designers to explore many trade-offs and select the best FPMM for each FPGA and optimization target (area *vs.* speed).

Table III reports results for recent efficient multipliers, with generic primes, from the state of the art (see Sec. II): MA16 we reimplemented from [19] (on all FPGAs), MR8/16 from [26] (on A7 for Artix-7), AM32/64 multipliers from [21] (on V7), and MO64 multiplier from [20] (on V5). We also report the recent, very small, multipliers MAS1/2 from [27] on IGLOO 2 (I2) but we do not have this FPGA for comparison.

Comparing the raw results from Tables II and III is not easy. Then, we illustrate the main differences between our FPMMs and state of the art multipliers using figures.

Figure 7 compares the computation time for several independent 128-bit multiplications. For only one or two MMMs, MA16 is more efficient than FPMM. But for more operations, FPMM is always faster. Our multipliers lead to a stair-shaped evolution since before reaching σ MMMs in one FPMM, launching a new multiplication is almost free: it only adds

TABLE II
IMPLEMENTATION RESULTS FOR A SELECTION OF THE MOST INTERESTING 128 AND 256-BIT FPMMS.

n	FPMM spec.	FPGA	slice / LUT / FF	BRAM	DSP	freq. MHz	λ cc	time 8M ns
128 bits	F44B	V4	**512** / **424** / **766**	$2^{\oplus}$	9	387	75	391
	S44B		**486** / **389** / **745**	$2^{\oplus}$	9	320	74	468
	F44B	V5	**257** / **397** / **728**	$2^{\oplus}$	9	483	75	313
	S44B		**242** / **369** / **703**	$2^{\oplus}$	9	397	74	378
	F44D	S6	**241** / **382** / **757**	0	9	349	75	433
	S44B		203 / **321** / **671**	$2^{\ominus}$	9	320	74	469
	S44D		**199** / 416 / **721**	0	9	334		449
	F44B	V7	325 / **545** / 725	$2^{\oplus}$	9	528	75	286
	F44D		**306** / **600** / **758**	0	9	633		239
	S44B		**287** / **523** / **683**	$2^{\oplus}$	9	481	74	312
256 bits	F28B	V4	**523** / **455** / **786**	$2^{\oplus}$	9	370	143	1445
	F48B		**611** / **504** / **829**	$2^{\oplus}$	9	387	255	1383
	S28B		534 / **422** / **766**	$2^{\oplus}$	9	317	142	1682
	F28B	V5	**240** / **433** / **751**	$2^{\oplus}$	9	497	143	1076
	F48B		**282** / **480** / **794**	$2^{\oplus}$	9	502	255	1065
	S28B		245 / **408** / 730	$2^{\oplus}$	9	397	142	1346
	S48B		**232** / 414 / **667**	$2^{\oplus}$	9	398	254	1342
	F28B	S6	205 / **335** / 743	$2^{\ominus}$	9	320	143	1671
	F28D		**209** / **415** / **777**	0	9	349		1533
	S28B		**175** / 358 / **690**	$2^{\ominus}$	9	320	142	1668
	F28B	V7	296 / **556** / 743	$2^{\oplus}$	9	528	143	1013
	F28D		**314** / **634** / **778**	0	9	598		895
	F48D		**291** / 674 / 787	0	9	552	255	969
	S28B		301 / **552** / **703**	$2^{\oplus}$	9	480	142	1112

s extra cycles.

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 DSP slices instead of 21 and 600 LUTs instead of 1182.

Figure 8 depicts the schedule for 8 independent MMMs in various 128-bit multipliers. 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.

TABLE III
RESULTS FOR 128 AND 256-BIT STANDALONE MULTIPLIERS FROM RECENT LITERATURE (* DENOTES ESTIMATION BASED ON PAPER VALUES).

n	Ref.	FPGA	slice / LUT / FF	BRAM	DSP	freq. MHz	λ cc	time 8M ns
128 bits	MA16	V4	879 / 1201 / 1311	$6^{\oplus}$	21	252	27	663
		V5	440 / 1027 / 1310	$6^{\oplus}$		292		571
		S6	540 / 1600 / 1280	$6^{\ominus}$		210		795
		V7	455 / 1182 / 1305	$6^{\oplus}$		350		478
	MR8	A7	206 / 255 / 487	0	19	198	33	1333*
256 bits	MA16	V4	1466 / 1998 / 2204	$10^{\oplus}$	37	250	37	932
		V5	698 / 1860 / 2172	$10^{\oplus}$		292		798
		S6	741 / 1941 / 2159	$10^{\ominus}$		177		1319
		V7	661 / 1770 / 2172	$10^{\oplus}$		372		626
	AM32	V7	- / 1917 / -	0	9	225	114	4049*
	AM64		- / 2343 / -	0	32	161	76	3776*
	MO64	V5	- / 699 / 333	$4^{\oplus}$	33	68	24	2814*
	MR16	A7	402 / 846 / 1123	0	29	146	66	3617*
	MR8		352 / 809 / 870	0	31	106	33	2490*
	MAS1	I2	- / 505 / 257	1	1	250	583	18656*
	MAS2		- / 680 / 341	2	2		312	9984*

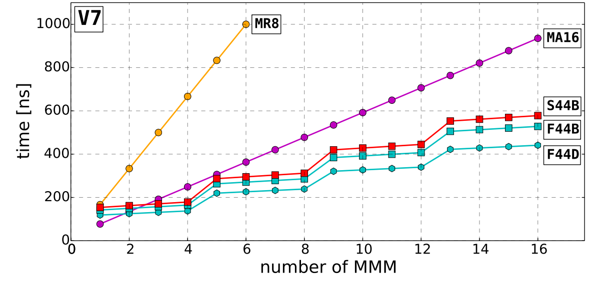

Fig. 7. Computation time for several numbers of MMMs and 128-bit multipliers on a Virtex-7. MA16 and MR8 are results from the state of the art ([19] and [26] respectively). The multipliers S44B, F44B and F44D are our solutions defined in Sec. V-C.

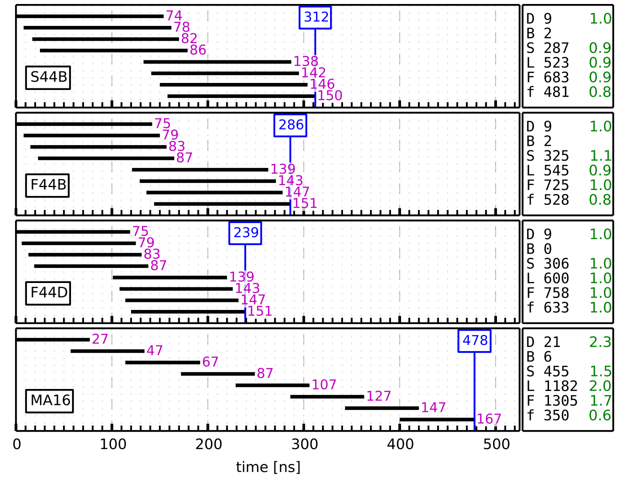

Fig. 8. Schedule details and area/time metrics for various efficient 128-bit multipliers on V7. Output timings are in purple (in cycles) and in blue for the last of 8 MMMs (in ns).

Exploring various DSP slice configurations using our generator allows us to almost reach the maximum frequency of the DSP slices or BRAMs. For instance, MA16 operates at 350 MHz while F44D reaches 633 MHz (V7 DSP slices have a maximum frequency about 740 MHz). Our generator allows us to discard the F44B specification since it reaches the maximum frequency of the BRAMs (about 530 MHz).

In Table IV we present the achievable *throughput* ($\mathrm{MMM} \cdot \mathrm{s}^{-1}$) *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 MMMs 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

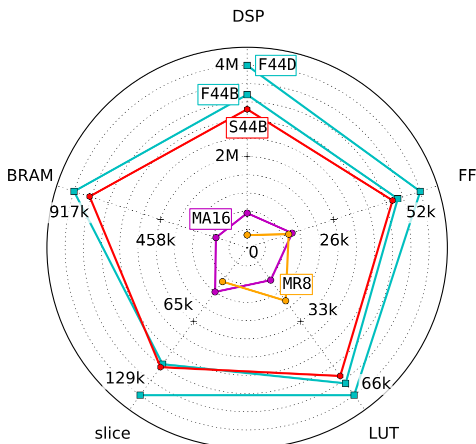

Fig. 9. Illustration of hardware efficiency, in $\mathrm{MMM}\cdot\mathrm{s}^{-1}$ per hardware resource unit (further from center is better).

TABLE IV

THROUGHPUT PER AREA RATIO (TPAR) FOR RECENT STATE-OF-THE-ART MULTIPLIERS AND OUR BEST FPMMS ON VARIOUS FPGAS. MEAN TPAR IS COMPUTED FOR ALL HARDWARE RESOURCES EXCEPT BRAMS.

n	FPGA	mult.	slice / LUT / FF ($\times 10^3$)	BRAM ($\times 10^6$)	DSP ($\times 10^6$)	mean ($\times 10^6$)
128 bits	V4	MA16	14.3 / 10.5 / 9.6	2.1	0.6	0.16
		F44B	47.2 / 57.0 / 31.5	12.1	2.7	0.70
	V5	MA16	33.2 / 14.2 / 11.2	2.4	0.7	0.19
		F44B	117.4 / 76.0 / 41.4	15.1	3.4	0.90
	S6	MA16	19.4 / 6.6 / 8.2	1.7	0.5	0.13
		S44D	104.9 / 50.2 / 29.0	-	2.3	0.62
		F44D	90.5 / 57.1 / 28.8	-	2.4	0.65
	A7	MR8	29.1 / 23.5 / 12.3	-	0.3	0.10
	V7	MA16	38.4 / 14.8 / 13.4	2.9	0.8	0.22
		F44D	129.2 / 65.9 / 52.1	-	4.4	1.20
256 bits	V4	MA16	6.1 / 4.5 / 4.1	0.9	0.2	0.06
		F28B	11.1 / 12.7 / 7.4	2.9	0.6	0.17
		F48B	9.9 / 12.0 / 7.3	3.0	0.7	0.18
	V5	MO64	n.a. / 4.1 / 8.5	0.7	0.1	0.03
		MA16	14.9 / 5.6 / 4.8	1.0	0.3	0.77
		F28B	32.4 / 17.9 / 10.3	3.9	0.9	0.23
		F48B	27.8 / 16.3 / 9.9	3.9	0.9	0.23
	S6	MA16	8.5 / 3.2 / 2.9	0.6	0.2	0.05
		F28D	26.1 / 13.1 / 7.0	-	0.6	0.16
	A7	MR16	5.5 / 2.6 / 2.0	-	0.1	0.02
		MR8	9.1 / 4.0 / 3.7		0.1	0.03
	V7	AM64	n.a. / 0.9 / n.a.	-	0.1	0.03
		AM32	n.a. / 1.0 / n.a.		0.2	0.11
		MA16	20.1 / 7.5 / 6.1	1.3	0.4	0.10
		F28D	29.8 / 14.7 / 12.0	-	1.0	0.27

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 $\mathrm{GF}(P)$. 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.

TABLE V

FPGA IMPLEMENTATION RESULTS FOR COMPLETE 128-BIT HECC CRYPTOPROCESSORS, USING F44B FPMM, ON VIRTEX-4 (V4), VIRTEX-5 (V5), SPARTAN-6 (S6) AND ZYNQ-7 (Z7). HECC ARCHITECTURE A2 EMBEDS ONLY ONE FPMM, WHILE ARCHITECTURES A3 AND A4 INTEGRATE TWO FPMMS. SEE [37] FOR DETAILS ON THESE HECC CRYPTOPROCESSORS.

FPGA	archi. type	$\tilde{w}$ bits	slice / LUT / FF	BRAM	DSP	freq. MHz	time ms
V4	A2	34	1081 / 863 / 1689	4	9	327	0.54
	A4	34	2447 / 1699 / 3255	7	18	328	0.39
	A3	136	3492 / 3959 / 5251	9	18	290	0.37
V5	A2	34	558 / 783 / 1653	4	9	386	0.45
	A4	34	1019 / 1413 / 3182	7	18	378	0.34
	A3	136	1657 / 2658 / 5170	9	18	356	0.30
S6	A2	34	382 / 911 / 1619	4	9	298	0.59
	A4	34	809 / 1565 / 3120	7	18	276	0.46
	A3	136	1182 / 3128 / 5040	9	18	238	0.45
Z7	A2	34	463 / 855 / 1619	4	9	347	0.50
	A4	34	747 / 1475 / 3020	7	18	360	0.36
	A3	136	1143 / 3147 / 5033	9	18	322	0.33

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-bretagneloire.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

[1] D. Koufaty and D. T. Marr, "Hyperthreading technology in the netburst microarchitecture," *IEEE Micro*, vol. 23, no. 2, pp. 56–65, Mar. 2003.

[2] G. Gallin and A. Tisserand, "Hyper-threaded multiplier for HECC," in *Asilomar Conf. on Signals, Systems and Computers*. IEEE, Oct. 2017, camera ready at https://hal.archives-ouvertes.fr/hal-01620046/.

[3] ——, "VHDL generator for hyper-threaded modular multipliers," https://sourcesup.renater.fr/htmm/, May 2018.

[4] T. Guneysu and C. Paar, "Ultra high performance ECC over NIST primes on commercial FPGAs," in *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, vol. 5154, Aug. 2008, pp. 62–78.

[5] G. R. Blakley, "A computer algorithm for calculating the product A*B modulo M," *IEEE Trans. on Comp.*, vol. C-32, no. 5, pp. 497 – 500, May 1983.

[6] S. Ghosh, D. Mukhopadhyay, and D. Chowdhury, "High speed Fp multipliers and adders on FPGA platform," in *Conf. Design and Architectures for Signal and Image Processing (DASIP)*, Oct. 2010, pp. 21–26.

[7] K. Javeed, X. Wang, and M. Scott, "Serial and parallel interleaved modular multipliers on FPGA platform," in *Internat. Conf. on Field Programmable Logic and Applications (FPL)*, Sep. 2015, pp. 1–4.

[8] ——, "High performance hardware support for elliptic curve cryptography over general prime field," *Microprocessors and Microsystems*, vol. 51, pp. 331–342, Jun. 2017.

[9] P. Barrett, "Communications authentication and security using public key encryption: A design for implementation," Ph.D. dissertation, University of Oxford, 1984.

[10] M. Knezevic, F. Vercauteren, and I. Verbauwhede, "Speeding up Barrett and Montgomery modular multiplications," 2009. [Online]. Available: https://homes.esat.kuleuven.be/~fvercaut/papers/bar_mont.pdf

[11] P. L. Montgomery, "Modular multiplication without trial division," *Math. of Comp.*, vol. 44, no. 170, pp. 519–521, Apr. 1985.

[12] H. Orup, "Simplifying quotient determination in high-radix modular multiplication," in *Symp. on Computer Arithmetic (ARITH)*. IEEE, Jul. 1995, pp. 193–199.

[13] C. D. Walter, "Montgomery exponentiation needs no final subtractions," *Electronics Letters*, vol. 35, no. 21, pp. 1831–1832, Oct. 1999.

[14] C. K. Koc, T. Acar, and B. S. Kaliski, Jr., "Analyzing and comparing Montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, Jun. 1996.

[15] C. McIvor, M. McLoone, and J. McCanny, "FPGA Montgomery multiplier architectures - a comparison," in *IEEE Symp. Field-Programmable Custom Computing Machines (FCCM)*, Apr. 2004, pp. 279–282.

[16] ——, "FPGA Montgomery modular multiplication architectures suitable for ECCs over GF(p)," in *IEEE Internat. Symp. on Circuits and Systems (ISCAS)*, vol. 3, May 2004, pp. 509–512.

[17] M. McLoone, C. McIvor, and J. McCanny, "Coarsely integrated operand scanning (CIOS) architecture for high-speed Montgomery modular multiplication," in *IEEE Internat. Conf. on Field-Programmable Technology*, Dec. 2004, pp. 185–191.

[18] D. Suzuki, "How to maximize te potential of FPGA resources for modular exponentiation," in *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Sep. 2007, pp. 272–288.

[19] Y. Ma, Z. Liu, W. Pan, and J. Jing, "A high-speed elliptic curve cryptographic processor for generic curves over GF(p)," in *Internat. Workshop on Selected Areas in Cryptography (SAC)*, vol. 8282, Aug. 2013, pp. 421–437.

[20] M. Morales-Sandoval and A. Diaz-Perez, "Scalable GF(p) Montgomery multiplier based on a digit-digit computation approach," *IET Computers & Digital Techniques*, vol. 10, no. 3, pp. 102–109, May 2016.

[21] D. Amiet, A. Curiger, and P. Zbinden, "Flexible FPGA-based architectures for curve point multiplication over GF(p)," in *Euromicro Conf. on Digital System Design (DSD)*, Aug. 2016, pp. 107–114.

[22] C. D. Walter, "Systolic modular multiplication," *IEEE Trans. on Comp.*, vol. 42, no. 3, pp. 376–378, Mar. 1993.

[23] T. Blum and C. Paar, "High-radix Montgomery modular exponentiation on reconfigurable hardware," *IEEE Trans. on Comp.*, vol. 50, no. 7, pp. 759–764, Jul. 2001.

[24] S. Ors, L. Batina, B. Preneel, and J. Vandewalle, "Hardware implementation of a Montgomery modular multiplier in a systolic array," in *Internat. Parallel and Distributed Processing Symp.*, Apr. 2003, pp. 184–191.

[25] C. McIvor, M. McLoone, and J. McCanny, "High-radix systolic modular multiplication on reconfigurable hardware," in *IEEE Internat. Conf. on Field-Programmable Technology*, Dec. 2005, pp. 13–18.

[26] A. Mrabet, N. El-Mrabet, R. Lashermes, J.-B. Rigaud, B. Bouallegue, S. Mesnager, and M. Machhout, "A scalable and systolic architectures of Montgomery modular multiplication for public key cryptosystems based on DSPs," *Journal of Hardware and Systems Security*, vol. 1, no. 3, pp. 219–236, Sep. 2017.

[27] P. M. Massolino, L. Batina, R. Chaves, and N. Mentens, "Area-optimized Montgomery multiplication on IGLOO2 FPGAs," in *Internat. Conf. Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–4.

[28] A. F. Tenca and C. K. Koc, "A scalable architecture for Montgomery multiplication," in *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, vol. 1717, Aug. 1999, pp. 94–108.

[29] A. F. Tenca, G. Todorov, and C. K. Koc, "High-radix design of a scalable modular multiplier," in *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, vol. 2162, May 2001, pp. 185–201.

[30] A. F. Tenca and C. K. Koc, "A scalable architecture for modular multiplication based on Montgomery's algorithm," *IEEE Trans. on Comp.*, vol. 52, no. 9, pp. 1215–1221, Sep. 2003.

[31] A. F. Tenca and L. A. Tawalbeh, "An efficient and scalable radix-4 modular multiplier design using recoding techniques," in *Asilomar Conf. on Signals, Systems Computers*, vol. 2, Nov. 2003, pp. 1445–1450.

[32] M. Huang, K. Gaj, and T. El-Ghazawi, "New hardware architectures for Montgomery modular multiplication algorithm," *IEEE Trans. on Comp.*, vol. 60, no. 7, pp. 923–936, Jul. 2011.

[33] *Virtex-4 FPGA Data Sheet 302: DC and Switching Characteristics*, Xilinx, Sep. 2009, v3.7.

[34] *Virtex-5 FPGA Data Sheet 202: DC and Switching Characteristics*, Xilinx, Dec. 2014, v5.4.

[35] *Spartan-6 FPGA Data Sheet 162: DC and AC Switching Characteristics*, Xilinx, Jan. 2015, v3.1.1.

[36] *Virtex-7 T and XT FPGAs Data Sheet 183: DC and AC Switching Characteristics*, Xilinx, Apr. 2017, v1.27.

[37] G. Gallin, T. O. Celik, and A. Tisserand, "Architecture level optimizations for Kummer based HECC on FPGAs," in *Internat. Conf. on Cryptology in India (IndoCrypt)*, Dec. 2017.

PLACE
PHOTO
HERE

Gabriel Gallin received the Master Degree in Computer Science in Paris, France, in 2013. He defended his PhD in Computer Science in 2018 in the IRISA laboratory in France. His research interests include computer arithmetic, computer architecture and digital security with applications in applied cryptography.

PLACE
PHOTO
HERE

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).