A HDL Generator for Flexible and Efficient Finite-Field Multipliers on FPGAs

Joël Cathébras(✉) and Roselyne Chotin

Sorbonne Université, CNRS, LIP6, 75005 Paris, France
joel.cathebras@gmail.com, roselyne.chotin@lip6.fr

Abstract. In this paper we propose a HDL generator for finite-field multipliers on FPGAs. The generated multipliers are based on the CIOS variant of Montgomery multiplication. They are designed to exploit finely the DSPs available on most FPGAs, interleaving independent computations to maximize throughput and DSP's workload. Beside their throughput-efficiency, these operators can dynamically adapt to different finite-fields by changing both operand width and precomputed elements.

From this flexible and efficient operator base, our HDL generator allows the exploration of a wide range of configurations. This is a valuable asset for specialized circuit designers who wish to tune state-of-the-art IPs and explore design space for their applications.

Keywords: Finite-field multiplier · FPGA design · Design space exploration

1 Introduction

In hardware design, when considering specific fields of application, FPGA targets are particularly attractive today and found in many hardware acceleration solutions. A classical step in the development of specialized hardware is the exploration of design space to make architectural choices [1,13]. This exploration may be necessary both at system's level and at IP's level. Exploration tools that allow different IP configurations to be tested are therefore valuable assets for digital circuit designers. This is also true for cryptography which is more and more present in our digital applications.

Modern cryptography is often build upon finite-field arithmetic. As in classical arithmetic, multiplication is an expensive operation and optimizations of multipliers are often the subject of researches and explorations [9,11,12].

In 2017 and 2018 Gallin and Tisserand [5,7] proposed a FPGA implementation of a Finely-Pipelined Modular Multiplier (FPMM) based on the CIOS variant [8] of the Montgomery modular multiplication [10]. It makes fine use of hardware resources present in FPGAs while exploiting in depth the characteristics of the chosen algorithm. Their operator has a good throughput per area ratio

© Springer Nature Switzerland AG 2021
J. C. Bajard and A. Topuzoğlu (Eds.): WAIFI 2020, LNCS 12542, pp. 75–91, 2021.
https://doi.org/10.1007/978-3-030-68869-1_4

compared to the state of the art due to a pipeline interleaving several independent computations. Their approach is quite parametrizable but the developed generator [6] is restricted to the parameters that were suitable for embedded elliptic and hyperelliptic curve cryptography.

This paper presents our work of building up an extended generator for modular multiplier based on the FPMM's approach. Our main scientific contribution consists in the practical generalization of this operator. Another contribution is a new functionality: the ability to dynamically change, to a certain extent, the width of the finite-field elements handled by the operator. When enabled, this feature increases the flexibility of the original operator for dynamically reconfiguring the finite-field over which the multiplier is operating. This feature could be interesting in different contexts. For instance, a crypto-processor that implements several primitives requiring finite-fields of different widths (e.g. RSA, ECC, HECC, etc.). Another example may be the implementation of modulus switching homomorphic encryption schemes (e.g. BGV [2]), which leads to a regular decrease in the width of underlying finite-field arithmetic.

2 Preliminaries

2.1 Notations

Throughout this paper we will use the following notations. An element of a large finite field, as well as the prime number that defines it, are in upper case and bold (e.g. $\mathbf{A} \in \mathbb{Z}_\mathbf{P}$). The width in bits of these elements is noted N. The Montgomery constant is M-bit wide and noted as $\mathbf{R}$.

The radix considered for Montgomery multiplication algorithm is 2^Ω. The Ω-bit elements are just in upper-case, not bold. Hence, a finite field element is decomposed into s elements of width Ω-bit (i.e. $\mathbf{A} = \{A_0, ..., A_{s-1}\}$).

The width of the basic arithmetic considered in this paper is constrained by the hardware resources available on a FPGA. We note here ω the width of a DSP slice's input words. These "basic words" are written in lower-case, and k denotes the number of them needed to write a Ω-bit word (i.e $A = \{a_0, ..., a_{k-1}\}$).

2.2 DSP Slices

FPGAs are chips made of a grid of configurable basic hardware blocks, along with a configurable interconnection network. Within these basic hardware blocks are elementary resources allowing among other things: combinatorial logic (e.g. Look-Up-Tables), data storage (e.g. Flip-Flops), and clock generation for synchronous circuits [4]. With the growth of size and performance required for circuits to be programmed, more complex hardware blocks have been added to the bestiary (e.g. BRAM, DSP, μP core, ...). In this work we are particularly interested in DSP slices.

A DSP is a basic hardware block that embedded a small multiplier, accumulators and cascading capabilities. They were historically designed for digital

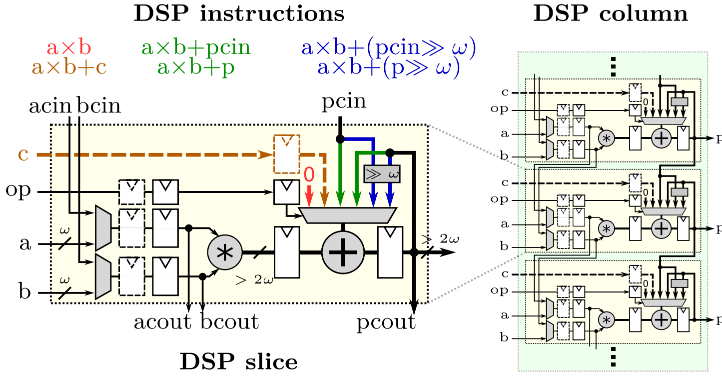


Fig. 1. DSP slice's characteristics used for our finite-fields multipliers. (Color figure online)

signal processing but could be efficiently used for finely-tuned arithmetic. They can achieve interesting running frequencies (up 700 MHz for last FPGAs).

Figure 1 presents the main DSP characteristics that have been exploited for our contributions. DSP are usually grouped in columns of half a dozen to a few hundred. This grouping facilitates cascading of operations and propagation of intermediate results within columns.

A single slice can be configured to perform one or several instructions multiplexed in time. In the latter, an operation code *op* is used to specify which instruction is issued. In this paper we are interested in three types of instructions : $a \times b$ (red) for a simple ω-bit multiplication, $a \times b + p$ (green) for a ω-bit multiplication followed by an accumulation, and $a \times b + (p \gg \omega)$ (blue) where the accumulated value is previously right-shifted by ω bits. For DSP slices that do not have this right-shift capability (like DSP48A slices in some Xilinx FPGA families), it is still possible to do so with external wires to the DSP. This makes use of the c port designed for $a \times b + c$ instructions (brown). It may require extra cycles in cascading operations to achieve the maximal running frequencies.

3 Previous Works on Finely-Pipelined Modular Multiplier

The main ideas behind the Finely-Pipelined Modular Multiplier (FPMM) are brought by Gallin and Tisserand's works [5,7]. They are introduced in this section but the details of the FPMM design comes in Sect. 4 along with our generalization of this approach.

Latency Optimized CIOS Algorithm Without Final Subtraction. The FPMM operator is based on the Coarsely Integrated Operand Scanning (CIOS) version of Montgomery multiplication [8], without final subtraction [14]. It is designed as

a pipelined interleaving several independent computations. The resulting algorithm is presented in Algorithm 1.

As a remainder, Montgomery multiplication computes the product modulo $\mathbf{P}$ of two elements $\mathbf{A}$ and $\mathbf{B}$ in Montgomery form (i.e. scaled by $\mathbf{R} = 2^M$), and return the result $\mathbf{T}$ in Montgomery form. For FPMM, $\mathbf{P}$ is taken less than $\mathbf{R}/4$ to avoid the final subtraction of the original Montgomery algorithm (consequently $N \leq M - 2$). In practice, the operands are considered to be M-bit integers decomposed into $s > 1$ words of Ω-bit.

Algorithm 1: Latency optimized CIOS algorithm

```
    Require: P = {P_0, ..., P_{s-1}} ; P' = -P^{-1} mod 2^Ω ; 4P < R(= 2^M).
    Input: A = {A_0, ..., A_{s-1}} ; B = {B_0, ..., B_{s-1}}.
    Output: T = {T_0, ..., T_{s-1}} with T = (AB · R^{-1}) mod P and 0 ≤ T < 2P.
 1  begin
 2  |   for i = 0 to s − 1 do
 3  |   |   for j = 0 to s − 1 do                          /* L1 stage */
 4  |   |   |   (D, U_j) ← A_i × B_j + T_j + D
 5  |   |   end
 6  |   |   Q_i ← (V_0 × P') mod 2^Ω    /* L2 stage: V_0 = (A_i × B_0 + T_0) mod 2^Ω */
 7  |   |   for j = 0 to s − 1 do                          /* L3 stage */
 8  |   |   |   (C, T_{j-1}) ← Q_i × P_j + U_j + C
 9  |   |   end
10  |   |   T_{s-1} ← T_{-1}^{(n)}                         /* i.e. C + D */
11  |   end
12  |   return T = {T_0, ..., T_{s-1}}
13  end
```

The particularities of the proposed implementation are visible at line 6 and 10, and comes from the proposed pipeline. For line 6, the original CIOS algorithm computes Q_i from U_0, but also resets D for each new upper-loop's iteration. The equivalent behaviour is achieved with V_0 that is extracted from the L1 stage computation. For line 10, the authors have demonstrated in [5] that the summation of remaining upper-words from L1 and L3 stages (i.e. C and D) is actually propagated in the immediately successive computation. In different terms, the upper word T_{s-1} is actually the lower word in the immediately successive (and independent) computation $T_{-1}^{(n)}$. We do not go into the details of the demonstration and invite the reader to take a look at the original paper.

Interleaving Independent Computations. The latency of an outer-loop iteration is noted α. It is defined from the input of $\mathbf{T}$ in L1 stage to its retro-propagation at the end of L3 stage. This latency depends on the choice of Ω w.r.t. the decomposition of multiplications onto DSP slices. For typical applications α is larger than s, leaving room for interleaving $\sigma = \lceil \alpha/s \rceil$ independent computations in the outer-loop's pipeline, while increasing its latency by $l_T = \sigma \cdot s - \alpha$. Thus, increasing by $s \cdot l_T$ cycles a single modular multiplication. We note "slot" the space taken by a single computation in the pipeline.

Figure 2 presents the interleaving principle. When a slot is unused, a new computation can be requested. The storage of $\mathbf{A}$ and $\mathbf{B}$ in local memories and to start their sub-word's read routine takes l_{in} cycles. Then, each outer-loop

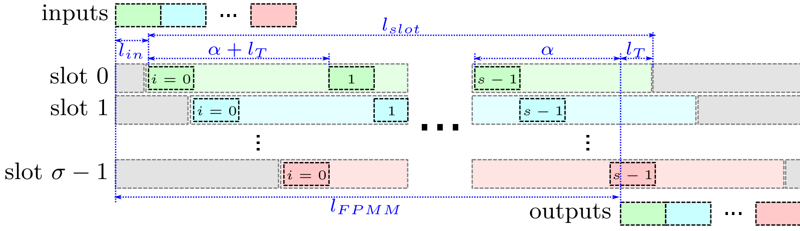


Fig. 2. Interleaving principle of the Finely-Pipelined Modular Multiplier.

iteration takes $\alpha + l_T$ cycles. While waiting for intermediate results, iterations of the other slots are performed.

Motivations for Extended Works. In [5], the authors explored FPMM implementations on different FPGAs from Xilinx. They varied several implementation parameters such as the number of slots σ or the type of memory used (BRAM or LUT based). However, their FPMM generator [6] is restricted to Ω being 2ω, which reduces FPMM's application ranges.

Therefore, our motivations are to generalize the FPMM principles to a wider range of configurations. In particular, we look to aim for larger finite fields, which require to choose $\Omega = k\omega$ with $k \geq 2$.

When studying the FPMM operator, we realized that it should be able to dynamically change the width of handled finite fields. Indeed, once parameter $\Omega = k\omega$ is chosen, FPMM's data-path is mainly fixed. The handled operands' width (M) drives s, σ and l_T parameters, which mainly impact control path. Thus, control path may be somehow duplicated for different width and a mode signal may select the current one.

4 FPMM's Model for HDL Generation

This section presents the generalized FPMM operator. It includes design modifications made to implement the multi-width feature.

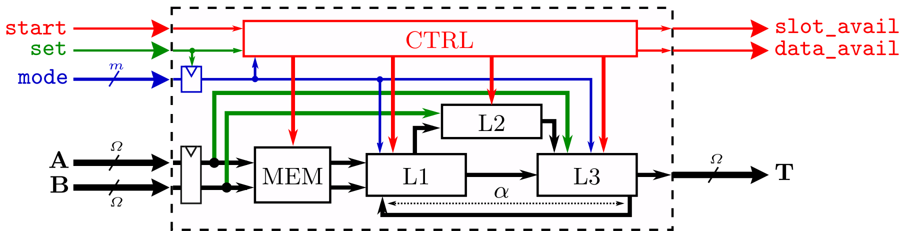


Fig. 3. FPMM top module.

4.1 FPMM Top Module

At top level, the FPMM operator is composed of five sub-modules: CTRL, MEM and the L1, L2 and L3 stages (Fig. 3). It has two operating phases: setup and run. A setup phase allows width mode and precomputed elements to be changed. When in run phase, FPMM handles up to σ independent multiplications, with σ depending on the current width mode.

MEM sub-module consists of two dual-port memories. Each of them can store up to $\max(s \cdot \sigma)$ Ω-bit words. Memory accesses are managed by CTRL while it is orchestrating the computations of the σ slots.

4.2 L1 and L3 Sub-modules

Regarding Algorithm 1, L1 and L3 stages are very close from each others. They both realize, in a i-indexed upper-loop, a j-indexed lower-loop of s iterations performing a multiply-accumulate operation of the form $(H_j, L_j) \leftarrow E_i \times F_j + G_j + H_{j-1}$. H_j and L_j are respectively the upper and lower resulting words, and E_i is constant for a whole lower loop.

Similarities of L1 and L3 Sub-modules. In practice, the multiply-accumulate operation is decomposed in three computation's sub-parts:

$$(H'_j, L'_j) \leftarrow E_i \times F_j \tag{1}$$

$$(c^{(1)}, V_j) \leftarrow L'_j + G_j \tag{2}$$

$$(c^{(2)}, L_j) \leftarrow V_j + H_{j-1} \tag{3}$$

The result L_j is the resulting lower word at the end of all sub-parts. The most significant word H_j is composed of H'_j and two carry bits $c^{(1)}$ and $c^{(2)}$ (i.e. $H_j = H'_j + c^{(1)} + c^{(2)}$).

To illustrate the following discussion on hardware implementation, we rely on Fig. 4. Equation 1 is broken down into ω-bit operations to be mapped onto DSP slices. It results in k^2 DSPs cascaded in space to fully pipeline the multiplication. Multiplexing in time with fewer DSPs would have reduced hardware utilization, but would have increased latency and impacted the whole operator's performances.

Due to this cascading, the sub-results of Eq. 1 are produced with some delay from each others. Each sub-result is used in further computation as soon as possible. Consequently, Eq. 2 is also broken down into k sub-additions with carry propagations (Fig. 4).

Given the Ω-bit multiplication algorithm, the mapping of sub-operations onto DSP slices is only dependent on k and DSP characteristics. All latencies resulting from the DSP cascade are known, and from them the hardware models of L1 and L3 stages are derived. To point out our contributions here: we managed to express hardware's models of each computation sub-parts depending on k and DSP characteristics. The automatic generation of HDL code is then possible from this modelisation.

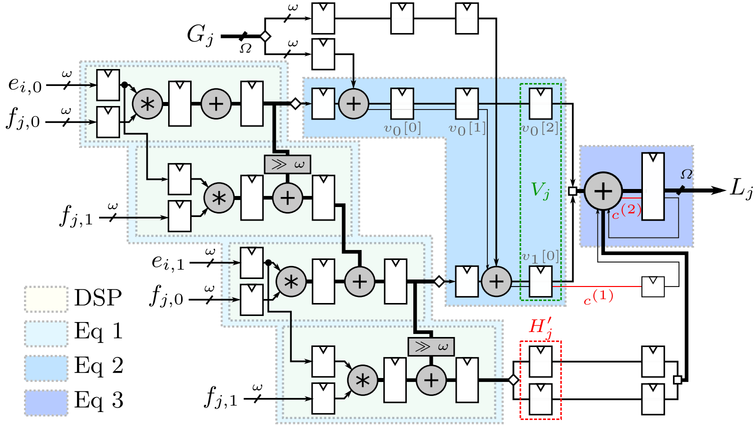
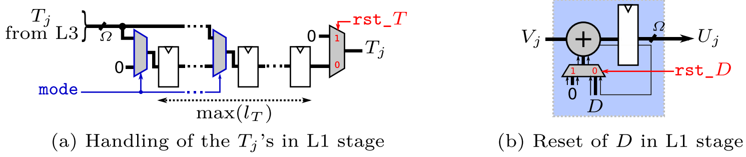


Fig. 4. Multiply-accumulate's pipeline in L1 and L3 sub-modules with $\Omega = 2\omega$.

L1 Stage Specificities. For the L1 stage, $(H_j, L_j, E_i, F_j, G_j)$ are identified with (D, U_j, A_i, B_j, T_j) from Algorithm 1. There are three specificities for L1 stage: handling the T_j's retro-propagated from L3 stage, reset D after a setup phase, and early propagation of V_0 to the L2 stage.



(a) Handling of the T_j's in L1 stage (b) Reset of D in L1 stage

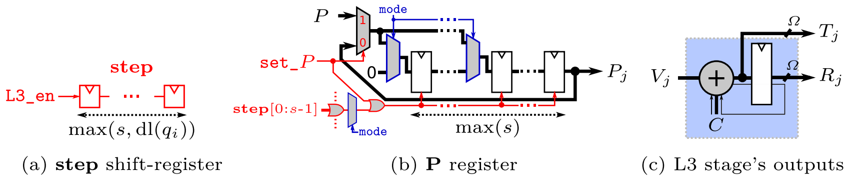
Fig. 5. Specificities of L1 stage.

Figure 5 illustrates design solutions for the first two specificities. In a multi-width context, different width mode may require different latencies l_T. The `mode` signal selects the shift register's depth according to current one (Fig. 5a). In addition, the T_j's are reset by the CTRL module (`rst`_T signal) whenever a new finite-field multiplication starts.

After a setup phase, the propagations of intermediate results through immediatly successive slots is mixed up. To restart a proper propagation, D is reset for the first slot going through the L1 stage after a setup phase (Fig. 5b).

Finally, the early propagation of V_0 is acheived by picking appropriately its sub-words from their delay lines. More details are given in Sect. 4.3.

L3 Stage Specificities. For the L3 stage, $(H_j, L_j, E_i, F_j, G_j)$ are identified with $(C, T_{j-1}, Q_i, P_j, U_j)$ from Algorithm 1. There are four specificities for the L3 stage. The first three are in the management of operands Q_i, P_j's and U_j's. The fourth one is the handling of outputs.



(a) **step** shift-register (b) **P** register (c) L3 stage's outputs

Fig. 6. Specificities of L3 stage.

Figure 6a presents the shift register **step**. It is fed by a `L3_en` signal coming from CTRL module to restart L3 operations at each new slot.

Due to L2 sub-module's architecture (presented in the next section), the Q_i input is received with delays between its sub-words. These delays are known and depend on the decomposition of Ω-bit multiplication onto DSPs. A write enable signal extracted from **step** (not shown) is generated for each sub-word to register them at appropriate time.

Figure 6b presents the storage module for **P**. It is composed of $\max(s)$ registers of Ω-bit that are reprogrammed whenever the prime **P** is changed (`set_`P). The depth used corresponds to the current width mode.

To synchronize the U_j's from L1 with Q_i input, an artificial latency l_U may be required depending on the FPMM configuration (not shown).

Finally, two L3 stage's outputs are differentiated (Fig. 6c): T_j feeds L1 stage for further iteration, and R_j feeds the final result port.

4.3 L2 Sub-module

L2 stage performs $Q_i = V_0 \times P' \bmod 2^{\Omega}$, with V_0 coming from L1, and P' being precomputed. For convenience, V_0 and Q_i are noted V and Q.

Given Ω-bit multiplication's decomposition, result modulo 2^{Ω} requires only $k(k+1)/2$ sub-multiplications. Moreover, L2 stage has s cycles to reuse the hardware before the next slot's data arrive. Consequently, only $\left\lceil \frac{k(k+1)}{2s} \right\rceil$ DSPs are

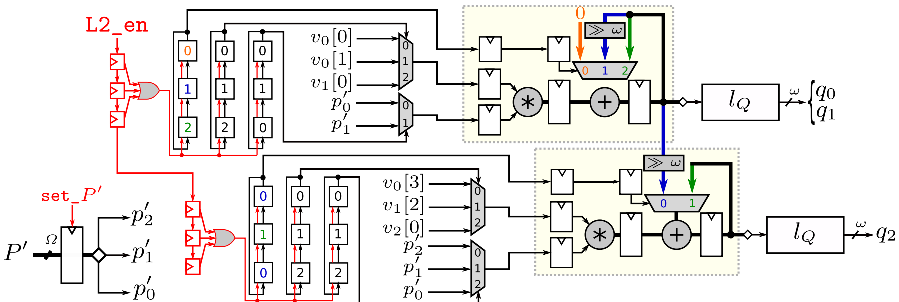


Fig. 7. Example of L2 stage for $\Omega = 3\omega$ and $s = 3$.

instantiated (minimum s in that case of multi-width mode). Figure 7 illustrates L2 stage for $k = 3$ and $s = 3$. The `L2_en` signal issued by CTRL restarts the sequence of operations each time a new slot arrives. Each DSP is configured with up to s different instructions, depending on the sub-multiplications it is taking care of.

As introduced earlier, V sub-words are extracted from L1 stage's data-path with appropriate delay. For instance, $v_i[d]$ is the d-th register delaying the result of the i-th sub-addition in Eq. 2's data-path (Fig. 4).

Outputs $(q_0, ..., q_{k-1})$ are progressively generated by appropriate DSPs, and stored in L3 stage as seen in L3 stage's specificities. Depending on the configuration, a latency l_Q may be required to synchronise L1 and L2's data-paths.

4.4 CTRL Sub-module

As FPMM's data-path handles slot-wise computations, CTRL module is ryhtmed over s cycles, depending on the current width mode (Fig. 8).

During a setup phase, CTRL updates precomputed values P' and $\mathbf{P}$ stored in L2 and L3 stages. It propagates `set_`P' (Fig. 7) and `set_`P (Fig. 6) appropriately (not shown for convenience).

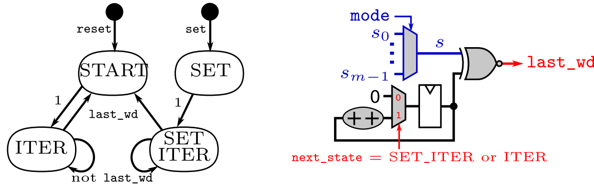


Fig. 8. Control FSM and slot cadencing generation.

During run phase, CTRL handles the succession of slots with the help of a control pipeline presented in Fig. 9. Whenever a `last_wd` is triggered, a verification is made of whether the next slot is free or already in use. This information is gathered from a shift register **slot_occ** presented latter.

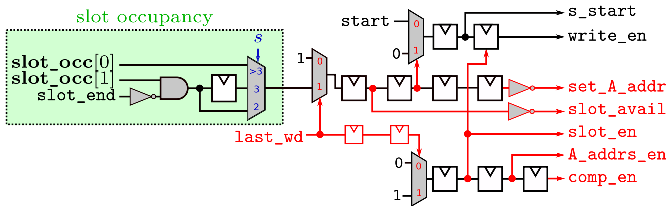


Fig. 9. Control pipeline orchestrated by `last_wd` signal.

From this, the FPMM signals any free slot (`slot_avail`) allowing a new computation to be required (`start`), in that case the storage of computation operands is issued (`write_en`). The other signals are used to handle slot and address managements.

Figure 10 presents the four shift registers for management of slots and addresses. They are all of depth max(σ), and the portion currently used depends on current width mode.

To the left, the shift register **slot_occ** memories the current slot occupancy. It is paired with **iter_count** that memories the current upper-loop's iteration of each slot (i-indexed). At a slot's last iteration the corresponding elements are reset in both shift registers (`slot_end` signal).

To the right, **base_addrs** stores for each slot the base address where operands are stored in MEM. It is reset with appropriate precomputed values during a setup phases (not shown here). **A_addrs** stores the address where A_i element is read for each slot current upper-loop's iteration. Read address for the B_j's is incremented at each cycle from the current slot's base address.

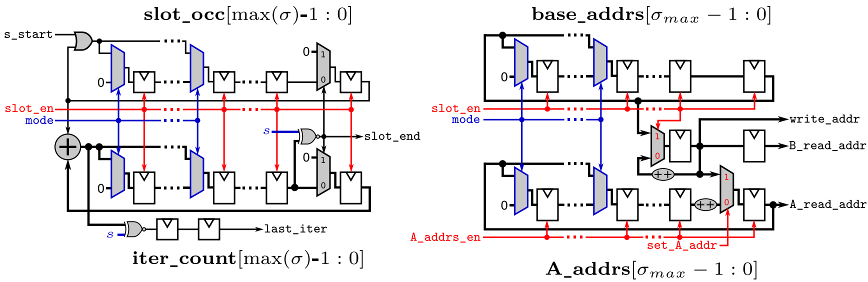


Fig. 10. Control shift registers for slot and address managements.

Finally, the different data path's control signals are delayed from the control signals shown in Fig. 9:

- `rst`_T: `write_en` delayed by $l_{MEM} + 4$ cycles.
- `rst`_D: `s_start` delayed by $1 + l_{MEM} + l_{L1}$ cycles, only for the first slot after a setup phase, there is no reset otherwise.
- `L2_en`: `comp_en` delayed by $l_{MEM} + 4$ cycles.
- `L3_en`: `L2_en` delayed by $l_Q + 4$ cycles.
- `data_avail`: (`comp_en` & `last_iter`) delayed by $l_{MEM} + 3 + \alpha$ cycles.

With l_{MEM} being the read latency of MEM, and l_{L1} the latency of L1 stage.

5 Implementation and Exploration Results

Some concrete implementation results are presented in this section. Each generated design has been tested with tens of random stimuli for each possible width

mode. Regarding FPGA implementations, a first "place and route" while targeting the theoretically possible maximum frequency (limited by BRAM or DSP) was done for each design. If the timings are not met, performance optimizers from FPGA manufacturer are run. If a design still fails to meet propagation delays, we lower its targeted clock frequency and repeat the previous steps.

We named designs after their k parameter and the FPGA resource used to implement its internal memories - B for BRAM and D for LUT. The operand widths M for which the different designs are generated are made explicit in the different discussions.

5.1 Outlines for FPMM Design Space Exploration

This section gives the FPMM's practical limitations, as well as general observations on its performances and its FPGA utilization as a function of sizing parameters.

Parameter Ranges. The generator imposes by design s and σ to be greater than or equal to two. Thus, a given parameter k implies boundaries on operand widths (i.e. M). Given a FPGA target, the choice of k implies Ω and α, and M is restricted to the range from $\Omega + 1$ (for $s > 1$) to $(\alpha - 1) \cdot \Omega$ (for $\sigma > 1$).

Table 1. (M, s, σ)'s ranges and number of cascaded DSP depending on k. Considering DSP48E2 from Xilinx Ultrascale+, with $\omega = 17$ (unsigned) and internal right-shifting operation.

k	Ω	α	l_Q/l_U	M range	s range	σ range	Cascaded DSP
2	34	14	0/4	35–442	2–13	7–2	4
3	51	18	0/0	52–867	2–17	9–2	9
4	68	32	7/0	69–2108	2–31	16–2	16
5	85	50	16/0	86–4165	2–49	25–2	25
6	102	72	27/0	103–7242	2–71	36–2	36
7	119	98	40/0	120–11543	2–97	49–2	49
8	136	128	55/0	137–17272	2–127	64–2	64

In addition, an FPGA target provides a limit to the depth of DSP cascades, depending on the size of the DSP columns. Thus, k must be less than the square root of the deepest possible cascade. This limitation can be somehow circumvented by cascading across DSP columns, but our generator does not handle this limitation case at the moment. An example of parameter ranges for various k is given in Table 1.

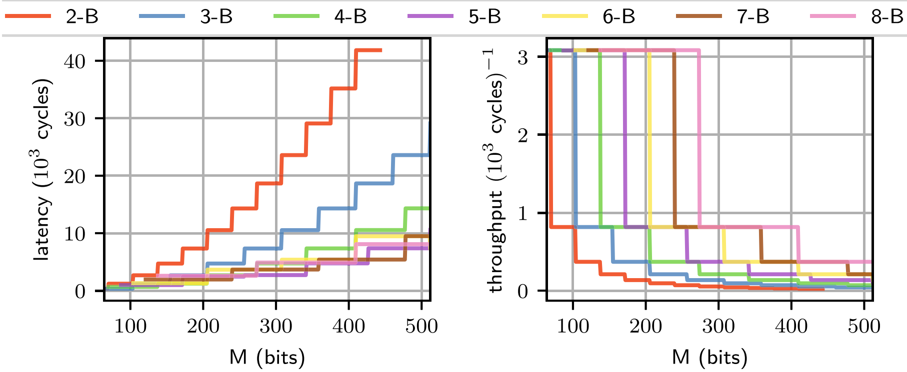


Fig. 11. Theoretical performance exploration for $M \in [64; 512]$.

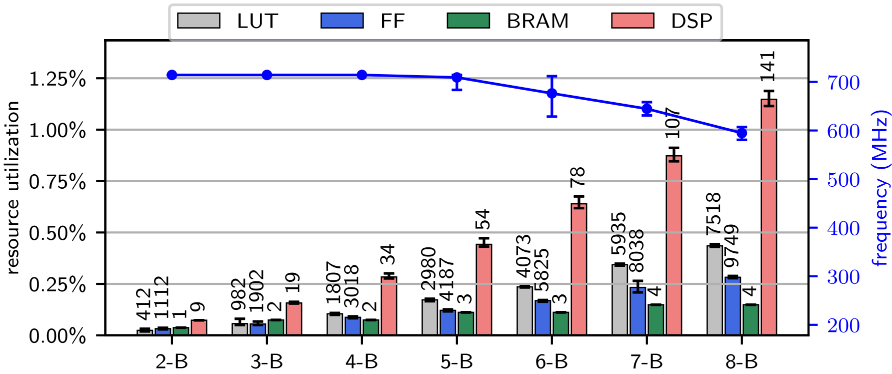


Fig. 12. Xilinx's Ultrascale+'s usage for $M \in [64; 512]$.

Theoretical Performances. Regarding latency, it takes $l_{FPMM} = l_{in} + (s-1)(\alpha + l_T) + \alpha$ cycles for a modular multiplication to be performed. For a given operand width M, a larger parameter k increases α but reduces s, thus an appropriate k may be found to reduce latency. Regarding throughput, the interleaving of independent computations allows the operator to output up to one operation every s^2 cycles[1]. A larger k always improves throughput by lowering s.

Figure 11 plots latency and throughput as a function of the operand width M for $k \in [2; 8]$. These are theoretical performances as it does not consider the running frequency achieved on a specific FPGA target. The stepped shapes are due to FPMM designs being actually dependent on (s, σ) pairs that are constants for M in $](s-1)\Omega; s\Omega]$. Nevertheless, these plots suggest that increasing k is a valid approach to improve both latency and throughput for larger M.

[1] When fully loaded with instructions, σ operations are outputed every $s^2\sigma$ cycles.

Resource Utilization. To give an overview of resource usage as a function of the parameter k, we implemented all designs possible with $M \in [64; 512]$ for each $k \in [2; 8]$. The targeted FPGA is an Ultrascale+ from Xilinx.

Figure 12 displays for each k averages of resource usage and running frequency, as well as deviations around these averages across designs (i.e. same k and different M). Designs are relatively small, and are able to reach important running frequency. The DSP are the limiting resources and their utilization is quadratic with the growth of k. The impact of M is rather insignificant on resource utilizations. For $k < 5$ the running frequency reach the upper limit imposed by BRAMs. The complexity of the design increasing for larger k, the maximum frequency achievable drops to 600 MHz (for $k = 8$).

5.2 Exploration's Examples for (H)ECC

In this section, we consider the use case of elliptic and hyper-elliptic curve cryptography to compare with previous works, and in particular [5] and [9]. For the sake of comparison, the FPGA target is now a Virtex-7 and the operand widths are 128-bit and 256-bit.

Table 2. Comparison of our generated FPMMs with [5] and [9].

M	Work	Name	CLB/LUT/FF	BRAM	DSP	Freq. MHz	Lat. ns	Thr. μs^{-1}
128	[9]	MA16	455/1182/1305	6	21	350	77.0	17.5
	[5]	F44B	325/545/725	2	9	528	141.8	33.1
	Our	2-B	305/406/1074	1	9	558	125.5	34.9
	Our	3-B	448/866/1785	2	20	396	156.4	44.0
	Our	4-B	839/1803/2823	2	37	373	193.2	93.2
	[5]	F44D	306/600/758	–	9	633	118.5	39.6
	Our	2-D	261/448/1218	–	9	539	129.8	33.7
	Our	3-D	497/921/1921	–	20	406	152.7	45.1
	Our	4-D	858/1902/2958	–	37	385	187.1	96.2
256	[9]	MA16	661/1770/2172	10	37	372	99.5	13.3
	[5]	F28B	296/556/743	2	9	528	270.3	8.3
	Our	2-B	481/578/1315	1	9	548	244.7	8.6
	Our	3-B	500/898/1732	2	19	414	280.3	11.5
	Our	4-B	860/1821/2829	2	35	367	370.9	22.9
	[5]	F28D	291/674/787	–	9	598	238.8	9.4
	Our	2-D	372/629/1401	–	9	527	254.3	8.2
	Our	3-D	466/956/1866	–	19	395	293.7	11.0
	Our	4-D	853/1913/2965	–	35	362	375.6	22.6

In Table 2 comparisons are regrouped according to M and the type of resources used to memorize input operands. Note that designs 2-[B/D] for

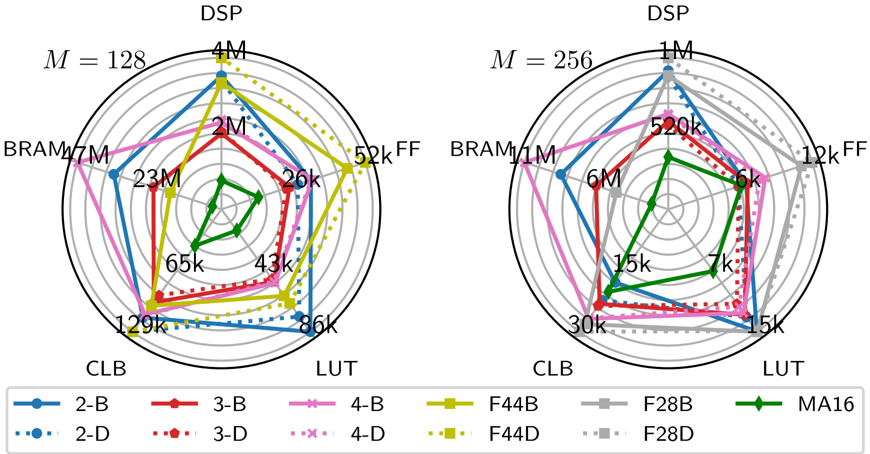


Fig. 13. TPAR comparisons on a Virtex-7 from Xilinx for $M \in \{128, 256\}$.

$M = 128$ (resp. $M = 256$) are actually our FPMM versions of F44[B/D] (resp. F28[B/D]), and are suppose to be very similar.

General Observations. A first observation is that increasing k makes the FPMM reach higher possible throughputs than the state of the art - more than twice the throughput of F44[D/B] and F28[B/D] with our 4-[B/D]. A second observation is that compared to the equivalent designs from [5] our FPMMs use more flip-flops and have a lower running frequency for designs with LUT based memories (e.g. F44D Vs 2-D). This is certainly due to specific optimizations for the case $k = 2$ that we given up to facilitate the generalization to larger k. A final observation concerns the operating frequency which seems to not be as stable with k growth as for Ultrascale+ target. One possible explanation could be the improvement of the configurable logic block carry-chains of the 7 Series for the Ultrascales (from 4-bit to 8-bit long). It may help to maintain the speed of the arithmetic as the operands' width grows. We did not investigate deeper this observation.

Throughput per Area Ratio. We then compare the different configurations through a typical metric that merges the information of throughput and hardware cost. This metric is called Throughput Per Area Ratio (TPAR) and gives a number of operations per second and per unit of utilized resources.

Figure 13 displays TPAR on Virtex-7 for 128 and 256 bits operators. According to the limiting resource (DSP) our best implementations are with $k = 2$. We believe that the the higher operating frequencies mainly explains this result.

From the flip-flops point of view, we find the consequence of the increased usage compared to the original works F44[D/B] and F28[D/B]. Nevertheless, all our different implementations remain relevant compared to Ma et al. [9] (MA16) approach. It has the closest performances to the FPMM's ones among the works to which [5] compared itself.

5.3 Multi-width FPMM

The multi-width feature allows an architect to implement a single operator to handle different operand widths. Among others, applications could be encryption circuits delivering several types of cryptographic primitives (e.g. RSA, DH, ECC, HECC,...), or different level of security.

To illustrate the gain brought by multi-width FPMM in such context, let's consider an application of elliptic and hyper elliptic curve cryptography with two different security levels, namely 128 and 256 bits. We further add the reasonable assumption that multiplication is the application's bottleneck and that the operator is working full time. We note p the fraction of time spent performing 128-bit multiplications and $(1-p)$ the 256-bit one. We now compare a multi-width FPMM against two other implementation choices: a 256-bit single-width FPMM used for both operand widths, and two different single-width FPMMs, one for each operand width.

Table 3. Comparison between single-width and multi-width FPMMs for $k = 2$ and $M = 128, 256$ on Virtex-7 (V7) and Ultrascale+ (US+).

	Parameters		Single-width				Multi-width			
	k	$M/s/\sigma$	CLB/LUT/FF	BRAM	DSP	f	CLB/LUT/FF	BRAM	DSP	$f_{\#}$
V7	2	128/4/4	305/406/1074	1	9	557.7	372/547/1234	1	9	515.2
		256/8/2	481/578/1315	1	9	547.6				
US+	2	128/4/4	116/383/1074	1	9	714.3	135/532/1229	1	9	714.3
		256/8/2	152/552/1332	1	9	714.3				

Table 3 shows implementation results of single-widths and multi-width FPMM on Virtex-7 and Ultrascale+. Compared to the first implementation choice, the multi-width FPMM improves throughput of 128-bit operations. The application speedup is then derived from the fraction of time p spent on these small operands[2]. For instance, with p equals to 0.25, 0.5 and 0.75 the respective speedups are ×1.65, ×2.35 and ×3.06 on Virtex-7 and ×1.75, ×2.5 and ×3.25 on Ultrascale+. These improvements require no increase of resource utilization.

Compared to the second implementation choice, the multi-width FPMM does not improve performances on Ultrascale+, and is 6% slower on Virtex-7 due to the loss in running frequency. Nonetheless, it reduces by roughly half the utilization of FPGA's ressources.

In conclusion, the benefits of the multi-width option are case-critical when several operand widths must coexist in the same application.

6 Conclusion

This paper presented our realization of an extended FPMM generator. Its main scientific contributions are the generalization of the FPMM operator presented

[2] Speedup $= \frac{f_{\#}\left(\frac{p}{4^2}+\frac{1-p}{8^2}\right)}{\frac{f}{8^2}}$.

by Gallin and Tisserand [5] and the addition of a multi-width feature. The generator is provided as a python package under GPL3 licence [3]. The purpose is to propose a design space exploration tool for FPGA implementation of modular arithmetic.

Although the core principles of the FPMM are implemented, our generator is currently limited to Xillinx FPGA families. We have good reason to believe that extension to other types of FPGAs should not be a major problem. Indeed, the FPMM pre-requisites on DSP are typical characteristics of these arithmetic units, non-specific to Xilinx's ones.

A direct improvement of our generator would be the integration of design optimizations for $k = 2$ from [5]. Another area for improvement could be the mapping of Ω-bit multiplication onto DSPs. Karatsuba's algorithm would certainly reduce the number of DSPs for large k. Nevertheless, it remains to identify the impact on α and the differential in utilization of other hardware resources.

Ackowledgments. We would like to thank Arnaud Tisserand for our interesting exchanges and his encouragement to publish these results; as well as the anonymous reviewers for their pertinent and welcome remarks and suggestions.

References

1. Bossuet, L., Gogniat, G., Philippe, J.L.: Exploration de l'espace de conception des architectures reconfigurables. **25**(7), 921–946. https://doi.org/10.3166/tsi.25.921-946
2. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference on ITCS 2012. ACM Press (2012). https://doi.org/10.1145/2090236.2090262
3. Cathébras, J., Chotin, R.: Finely Pipelined Modular Multiplier (FPMM). https://gitlab.lip6.fr/roselyne/fpmm/. Accessed 12 Aug 2020
4. Deschamps, J.P., Sutter, G.D., Cantó, E.: Guide to FPGA Implementation of Arithmetic Functions. Springer, Netherlands (2012). https://doi.org/10.1007/978-94-007-2987-2
5. Gallin, G., Tisserand, A.: Generation of finely-pipelined GF(P) multipliers for flexible curve based cryptography on FPGAs, pp. 1–12. https://doi.org/10.1109/TC.2019.2920352
6. Gallin, G., Tisserand, A.: Hyper-Threaded Modular Multipliers (HTMM). https://sourcesup.renater.fr/www/htmm/. Accessed 25 Feb 2020
7. Gallin, G., Tisserand, A.: Hyper-threaded multiplier for HECC. In: 2017 51st Asilomar Conference on Signals, Systems, and Computers. IEEE (2017). https://doi.org/10.1109/acssc.2017.8335378
8. Koc, C.K., Acar, T., Kaliski, B.: Analyzing and comparing montgomery multiplication algorithms. IEEE Micro **16**(3), 26–33 (1996). https://doi.org/10.1109/40.502403
9. Ma, Y., Liu, Z., Pan, W., Jing, J.: A high-speed elliptic curve cryptographic processor for generic curves over GF(p). In: Lange, T., Lauter, K., Lisoněk, P. (eds.) SAC 2013. LNCS, vol. 8282, pp. 421–437. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43414-7_21

10. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**(170), 519 (1985). https://doi.org/10.1090/s0025-5718-1985-0777282-x
11. Morales-Sandoval, M., Diaz-Perez, A.: Scalable GF(P) montgomery multiplier based on a digit-digit computation approach. IET Comput. Digit. Tech. **10**(3), 102–109 (2016). https://doi.org/10.1049/iet-cdt.2015.0055
12. Mrabet, A., et al.: A scalable and systolic architectures of montgomery modular multiplication for public key cryptosystems based on DSPs. J. Hardw. Syst. Secur. **1**(3), 219–236 (2017). https://doi.org/10.1007/s41635-017-0018-x
13. Pimentel, A.D.: Exploring exploration: a tutorial introduction to embedded systems design space exploration. IEEE Des. Test **34**(1), 77–90 (2016). https://doi.org/10.1109/mdat.2016.2626445
14. Walter, C.: Montgomery exponentiation needs no final subtractions. Electron. Lett. **35**(21), 1831 (1999). https://doi.org/10.1049/el:19991230