HLS-Based Methodology for Fast Iterative Development Applied to Elliptic Curve Arithmetic

Simon Pontie, Alban Bourge, Adrien Prost-Boucle, Paolo Maistri, Olivier Muller, Régis Leveugle, Frédéric Rousseau

HLS-Based Methodology for Fast Iterative Development Applied to Elliptic Curve Arithmetic

Simon Pontie, Alban Bourge, Adrien Prost-Boucle, Paolo Maistri, Olivier Muller,
Régis Leveugle and Frédéric Rousseau
Univ. Grenoble Alpes, TIMA, F-38000 Grenoble
CNRS, TIMA F-38000 Grenoble
{firstname.name}@imag.fr

***Abstract*—High-Level Synthesis (HLS) is used by hardware developers to achieve higher abstraction in circuit descriptions. In order to shorten the hardware development time via HLS, we present an adjustment of the Iterative and Incremental Design (IID) methodology, frequently used in software development. In particular, our methodology is relevant for the development of applications with unusual complexity: the method was applied here to the development of large modular arithmetic, commonly used for cryptography applications (e.g., Elliptic Curves). Rapid feedback on circuit characteristics is used to evaluate deep architectural changes in short time, greatly reducing the time-to-market with respect to hand-made designs. In addition, our approach is highly flexible, since the same generic high-level description can be used to produce an entire set of circuits, each with different area/performance trade-offs. Thanks to the proposed approach, any change to the initial specification (e.g., the curve used) is also very fast, while it may require a large effort in the case of hand-made designs.**

I. Introduction

Current trends are pushing hardware description towards higher abstraction. In this context, High-Level Synthesis (HLS) tools allow for a concise description of circuits [1]. They offer the best trade-off between precision, the asset of lower level description languages such as VHDL or Verilog, and the rapid prototyping often associated with higher level description languages such as C or C++. These higher-level languages are historically used in the software development field, even if the differences between hardware and software design are numerous. Hardware development is a task involving more tools and steps than software development in the sense that the path toward a functional prototype is longer. Moreover, architectural changes often involve a consequent rewriting and testing workload. On the other hand, incremental and iterative methods (also called agile methods) are often applied in the software field to cope with these issues [2]. These methods are of multiple types, but they have a few rules in common including the objective of having a working prototype as fast as possible. The following design iterations aim at improving the existing version without breaking anything. Such methods are not easily applicable to hardware development due to the usage of verbose *hardware description languages*.

This paper presents the results of an Incremental and Iterative Design (IID) method applied to the design of a generic Elliptic Curve Cryptography (ECC) crypto-coprocessor, based on an HLS generation flow. HLS is commonly used for some mainstream applications, but the specific case of ECC requires architectures based on uncommon operations (e.g., large integer multiplications or reductions modulo a large prime) that are less suitable for current HLS tools optimizations. In this particular case, it is not straightforward to consider a fully automatic HLS flow giving results that can be compared with the state of the art. This case study is focusing on this example to highlight how an HLS flow can still be used and how an IID method can compensate for some of HLS drawbacks.

By combining HLS and IID during development, we were able to iterate quickly in order to test multiple architectural changes until obtaining a satisfactory solution. Additionally, the generated circuit is described in a high-level language. Therefore, it is very versatile as it supports updating its parameters before synthesis. Choosing the supported elliptic curve or how large integers are partitioned remains easy. Finally, software tests during the development of prototypes give quick feedbacks before hardware tests validate the generated circuits.

The paper is organized as follows. We will first describe the motivations of this work. Then, we will present the proposed design flow and how the IID method is used. The results are then discussed in Section IV.

II. Motivations

In the first place, the objective was to obtain quickly an efficient but versatile coprocessor for Elliptic Curve Cryptography. This case study was chosen because it is a difficult case for HLS tools, which are usually efficient for classical integer operations. Indeed, very large integers are generally used in cryptography. We will first provide some details about this application to emphasize development choices before summarizing the bases of our methodology.

A. Designing and Updating an ECC Coprocessor

Using Elliptic curves for asymmetric cryptography was initially proposed in 1985 by Miller [3] and Koblitz [4]. Most elliptic curves used in cryptography are described by quadratic equations on two (or more) variables, where each term is defined over a binary or a prime Galois field. To use such cryptographic mechanism, participants should agree on one particular elliptic curve which defines a related security level. Due to recent progress in mathematical attacks against curves over binary fields [5], elliptic curves over prime field seem to be a safer

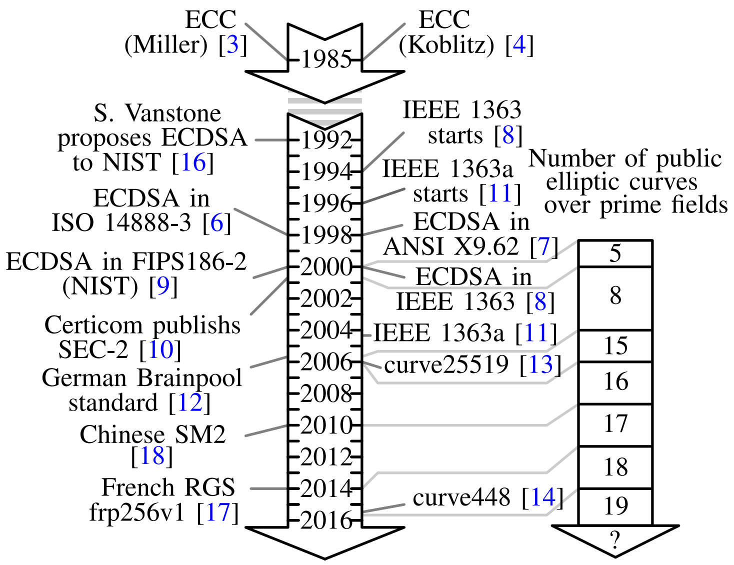

Fig. 1: ECC history.

choice. Therefore, we focus on curves over $GF(p)$ (prime field) in this paper. Figure 1 illustrates the ECC history by underlining some major events. Real life utilizations date back to the early 21st century after governmental and international standardizations, but also recommendations from consortiums [6]–[11]. This was followed in October 2005 by a new standard with an alternative curve generation method [12]. Recently, new curves with specific parameters were proposed [13], [14] and are already used in some applications. It is even possible for anyone to send random data in order to participate in the seed for a new curve generation [15]. Other elliptic curves exist but are rarely used or with less extensive support by tools. Figure 1 illustrates that new curves are regularly proposed. In the end, there is still no unanimous consensus about which curve should be used.

In this context, focusing a hardware development on a specific elliptic curve could lead to the design of an outdated circuit at short term. Our proposition is therefore to use reconfigurable devices (FPGA) to prevent this. The proposed building process is compatible with all elliptic curves over prime field even if each built bitstream is specialized for a specific curve. This permits to efficiently minimize the resource usage, but requires to provide a simple method to update the supported curve. This is possible thanks to an automatic building process as explained in section II-C. The specialization for a selected curve is done during this build. This automatic process should also permit to adapt the ratio performance/area. For instance, from the same input description, we are able to produce very cheap (but slow) crypto processors, or a fast but more resource consuming one.

ECC can be used for authentication, cyphering, or secret exchange protocols. All are based on the same time consuming operation: the scalar multiplication, which is a one-way function as no known sub-exponential algorithm currently exists to reverse it. Since this is the bottleneck of our application, the case study in this paper is focused on this operation. The

```
Require:
    $P$ = a point of the elliptic curve
    $O$ = the infinite point
    $k$ = a $n$-bit width scalar = $(k_{n-1}k_{n-2}...k_0)_2$
Ensure: $Q = k \times P$
 1: $Q \leftarrow O$
 2: for $i = n-1$ to 0 do
 3:     $R \leftarrow 2 \times Q$                     ▷ point doubling
 4:     $S \leftarrow R + P$              ▷ point addition always done
 5:     if $k_i = 1$ then
 6:         $Q \leftarrow S$              ▷ Write-back: $Q = 2 \times Q + P$
 7:     else
 8:         $Q \leftarrow R$                  ▷ Write-back: $Q = 2 \times Q$
 9:     end if
10: end for
11: return $Q$
```

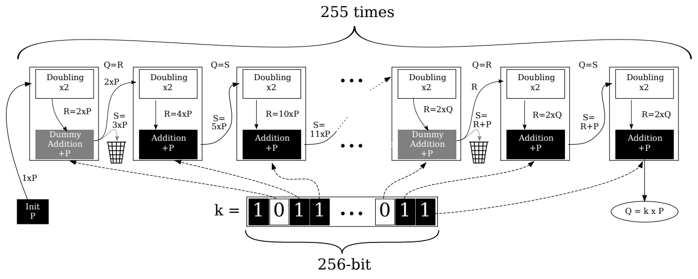

Fig. 2: The Double-And-Add Always ECSM algorithm: point doublings in white and point additions in black or grey

elliptic curve scalar multiplication (ECSM) is not a classical integer multiplication, as it is defined between a secret scalar (classically a 256-bit integer) and a point of the elliptic curve, whose coordinates are described by large integer values. Several algorithms for the ECSM exist, but they are usually computed by using the same two basic point operations: point addition and point doubling. These operations are not the classical doubling or addition on integer, because their operands are actually points of an elliptic curve.

Although we will not consider here the threat of an attacker with physical access to the device, remote attacks to the device are often realistic. If an attacker can communicate with the device, he will be able to measure the computation time. This can be an important information for accelerating cryptanalysis [19], and this is why only constant time algorithms are usually chosen. For the ECSM, the simplest constant time algorithm is the Double-And-Add Always that is illustrated in Figure 2. In order to compute the ECSM kP, the scalar coefficient k is scanned from the most significant to the less significant bit. For each bit, the point accumulator Q is doubled and stored in R. A point addition $P + R$ is always computed and stored in S because of constant time requirement. The write-back in the accumulator Q depends on the bit k_i value (black or grey boxes in Figure 2). For a 256-bit scalar, the algorithm will always require 255 point doublings and 255 point additions.

Specific formulas are used to compute coordinates of the addition and doubling resulting points. Performance of these formulas depends on the chosen point representation. For performance considerations, we have selected the Jacobian representation [20]. The formulas require large integer operations over the prime field $GF(p)$: addition, subtraction, inversion, and multiplication, all modulo p. The prime number p can be different for each elliptic curve. Some curves over prime fields may have specific p values in order to implement more efficient modular reductions [9], but there are also curves with generic primes and thus less efficient reductions. Since we want to support any curve over arbitrary prime fields, the latter approach is thus chosen. In order to minimize the impact of this choice on global performance, the values are represented in the Montgomery domain, which allows us using a Montgomery multiplier and thus avoiding the time consuming reduction operation [21]. This multiplier is sequential as the result is computed by splitting the operands into smaller digits which are processed one cycle at a time. The elementary multiplication between two digits is computed in 1 clock cycle and defines the main critical path. The smaller the digit, the higher the operating frequency, but the more cycles the full integer multiplication will require. Evaluating the optimal digit width is therefore a difficult problem.

B. Fast Design Method

Cryptographic choices impose some constraints but due to many freedom degrees, there are always a lot of non-imposed implementation details. These details can have important effects on the overall performance which can be difficult to anticipate. We thus need a method allowing a fast manual exploration of the different solutions with relevant and precise indicators. This means that performance or cost (area) indicators should be quickly available to fix design choices as soon as possible. That is why we should have a functional system in a short time, even if sub-optimal, which we will be able to improve later by incremental and iterative development steps. The core of our method is hence to import and adapt IID from the software field in order to try and rewrite as many circuit descriptions as required to reach the target. This adaptation may not seem straightforward for hardware development but it will eventually allow a fast design process.

C. Using HLS for IID

To adapt an IID method to hardware development, the key idea is to use an HLS flow in order to benefit from its advantages. First, a designer needs precise feedback on the resources and the performance of his circuit. He also needs this information as soon as possible during the design process, in order to react and rewrite or adapt the source code. HLS allows quickly producing a functional circuit as the source code can be compiled and tested like any software. On the other hand, the same source code can also go through an HLS flow, where the tool can provide useful information to the developer, such as the area of the circuit and performance estimation. With such data, the developer can immediately change the source code and both test functionality and estimate performance. One

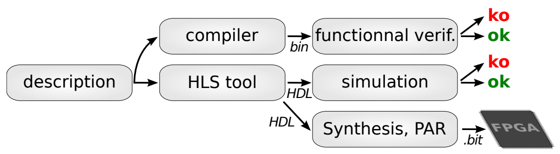

Fig. 3: Generic design flow and tool chain requirements.

should note that these iterations are not the classical HLS user choices (e.g., loop unrolling) but source code modifications in order to choose for instance a different algorithm.

Secondly, HLS allows for writing complex circuits in a more concise way compared to a classical method with Hardware Description Languages (HDLs). A minor variation in the source code given to an HLS tool can lead to significant architectural changes that would take a large amount of manpower in order to apply them in HDL.

Eventually, higher-level languages allow for a very powerful set of tools (the C preprocessor for instance) giving the possibility to produce versatile circuits. As seen earlier, in the ECC context and because of the rapid evolution of the curve set, such a feature is mandatory. In our case, as an example, the digit width was never fixed during the development. It was kept as a parameter that permits to generate a set of crypto-processors for a curve with different area/performance ratios. In the end, the designer can select the circuit satisfying his constraints among the set of generated crypto-processors.

For all these reasons, our proposal is focused on describing a hardware circuit in a high-level language (not a HDL) to feed an HLS tool. Figure 3 gives an overview of the different steps that the source code goes through during the design flow. It is exploited in two independent build chains: the software one, used to quickly test the functionality, and the hardware one, to check the hardware functionality (slower than in software) and produce a FPGA-ready bitstream. The tedious debug of hardware functionality is (at least partially) replaced by a faster software debug.

D. Summary of Needed Operations and Constraints

The top operation is the ECSM. There are hierarchical dependencies between needed operations:

- The scalar multiplication: a special multiplication between a scalar and a point of the elliptic curve. The scalar length is close to the length of the prime p.
- Basic point operations: point doubling and point addition.
- Operations over $GF(p)$: **multiplication**, subtraction, addition and inversion of large integers modulo a large prime number. The typical size of the prime and of the operands is currently 256-bit for a reasonable security level, but it may increase in the future.

The sub-section IV-A presents our design method on a limited part of the crypto-processor: the multiplication over the prime field $GF(p)$, the most important operation used in ECSM. On the other hand, we compare our final circuits to the state of the art at the top-level (the ECSM) in Section IV-B. In addition, three constraints must be respected:

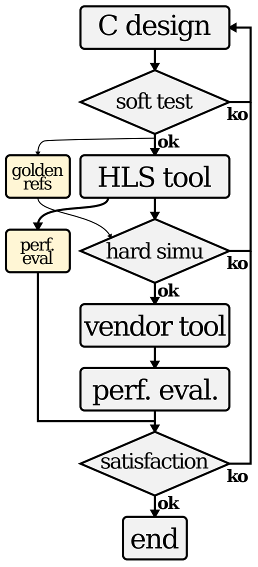

Fig. 4: Flowchart of HLS design methodology.

- All operations must be computed in constant time to protect the circuit against timing attacks.
- The elliptic curve must be a parameter (chosen before synthesis). This permits an easy update when the supported curve of a design is outdated or a new one is chosen. This also permits to generate a set of circuits supporting different curves.
- The digit width must also be a parameter (chosen before synthesis) because the optimal size is difficult to anticipate before the place and route step. Additionally, this permits to generate a set of circuits with different area/performances ratios.

III. HLS and IID

In this section the proposed design flow and the IID method are discussed in more details. The tools used for building the ECC coprocessor are then presented.

A. Design Flow

In this paragraph, we describe the method used during hardware design. Figure 4 represents the flowchart of the design phases. The first step is to describe the hardware design using the C language (actually, a subset of ANSI C cf. III-B). This version can be compiled with `gcc` for instance in order to be executed on a standard general purpose processor. The outputs of the executed program are checked to assert the algorithmic correctness of the program. As long as the program outputs are wrong, there are corrections to include in the C source code. When the modifications are done, the C source code is given to the HLS tool which produces a hardware description of the program in standard HDL. The HLS tool can give some performance information about the circuit. If these performances are not satisfactory, the designer can try improving the C source code. The output of the HLS tool is then given to an HDL simulator for verification and another corrective iteration can occur. Eventually, the designer can use any tool chain, according to the vendor of the targeted FPGA, to generate the final circuit and obtain precise information that can lead to further performance-oriented iterations.

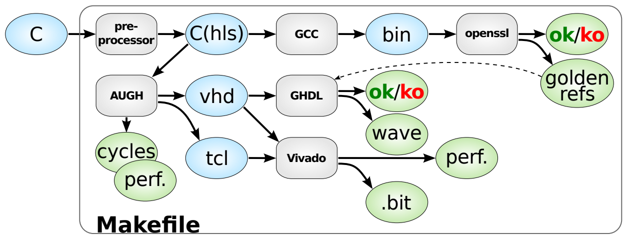

Fig. 5: Set of tools used to setup design flow.

B. Tools

Keeping the developed application up to date with current curves is a major concern. In order to take advantage of newer FPGAs as they are released, the tools used to build the presented circuit are relatively independent of the target. Moreover, this tool chain is Free and Open Source Softwares (FOSS) for the most part. However, the need for a final estimation of the design choices and the FPGA implementation imply the usage of vendor tools so that specific FPGAs can be targeted, and the design evaluated and built for this specific chip. In this paper, a Xilinx chip is targeted, hence, the Xilinx tool suite is used further in this work. Note that any tool suite from other FPGA vendors could have been used instead.

Figure 5 references all the tools used in the studied design flow. This entire flow presented in III-A is encapsulated in an automated environment handled by `make`. The first step is to modify the C input to adapt the code to the chosen parameters (particularly the curve type). This code is given to `gcc` for compilation, the produced binary is tested, and the correct outputs are kept as golden references. In parallel, the HLS tool Augh [22] generates the circuit in VHDL and reports estimation of the final circuit latency and performance (i.e. frequency). It also generates a TCL script in order to automatize the final circuit generation through Xilinx's Vivado.

Augh is used in this paper for the following reasons:

- it produces a VHDL output very quickly
- it is vendor agnostic and produces code for various FPGA families
- it performs a design space exploration automatically, the user is not involved in the hardware circuit optimization process
- Eventually, Augh is an academic free and open source software, hence results are reproducible.

Augh takes a subset of ANSI-C as input and produces VHDL files.

IV. Evaluation of the design method

How the results were obtained is here as much important as what the results are. Hence, performance of every obtained circuit are analysed with respect to the development timeline.

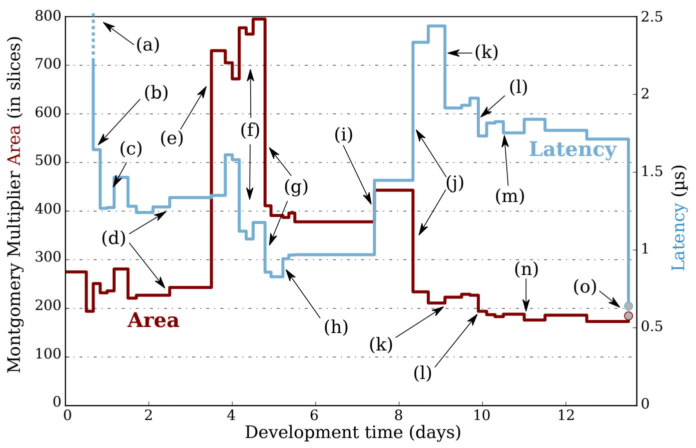

Fig. 6: Area and latency of the Montgomery multiplier vs. time for 256-bit operands and 32-bit digits.

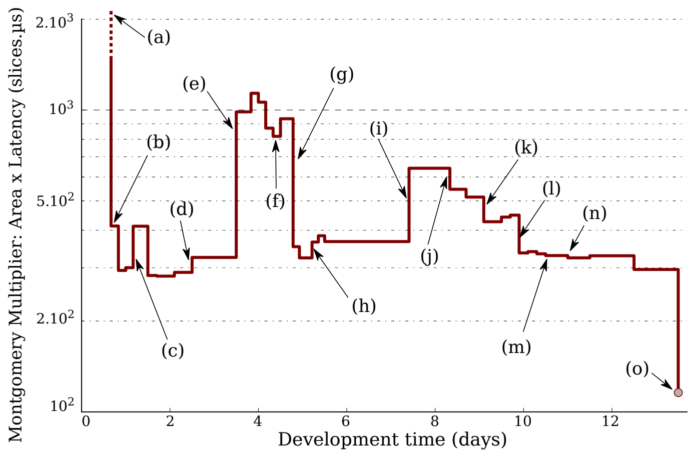

Fig. 7: Area × latency of the Montgomery multiplier vs. time for 256-bit operands and 32-bit digits.

The mere results are also analysed with a non-exhaustive state of the art comparison, in order to validate the obtained architecture.

A. Architecture Performances

In this paragraph, we will discuss Figure 6 which presents the performance and the cost of the obtained Montgomery multiplier implementations relatively to a three-weeks timeline. Version controlled code allowed us to plot such data. It should be noted that with the IID method, every obtained circuit is a fully functional ECC coprocessor. However, we focus on the Montgomery multiplier in order to provide the simplest illustration of our development method. All field multipliers support arbitrary digit width, but for ease of clarity only those based on 32-bit digits are represented.

During the first days of development, a functional software application was developed. Automatic test based on the classical cryptography library `openssl` were also implemented to prevent non functional circuits. The first functional hardware circuit was obtained after this period: (a) in Figures 6 and 7. This circuit consumed few resources because parallelism is not identifiable in the first written code. For the same reason, this implementation was inefficient. Quickly (b), useless carry propagations were identified and suppressed: computation time was reduced by 10 times. The multiplier algorithm contains several accumulation and bit shifting operations. The next improvement (c), illustrated in Figure 8, was to merge the bit shifting with the previous operation by modifying the write-back destination. Right immediately, the C code was relaxed to permit parallelism between partial products and a Montgomery specific digit multiplication before shifting.

After some time, a latency increase can be observed (d). The final step of our multiplication is a conditional subtraction, which is always computed to avoid timing attacks. The write-back is thus disabled when this operation is not required. So far, the digits of the multiplier operands are incrementally scanned: in order to reduce routing, operands were hence copied into local rotating buffers (e). However, it appears that the area cost of these rotating buffers is very large. Merging two close loops

```
Require:
    $a = (a_{m-1}, ..., a_2, a_1, 0)$
Ensure: $a' = a >> \text{digit width} = (0, a'_{m-2}, ..., a'_1, a'_0)$
1: $a_0 = 0$
2: for $i = 1$ to $m-1$ do                      ▷ Compute a
3:     $a_i = ...$
4: end for
5: for $i = 0$ to $m-2$ do                      ▷ Shift a
6:     $a'_i = a_{i+1}$
7: end for

                    ⇓

1: for $i = 0$ to $m-1$ do                      ▷ Compute and shift a
2:     if $i > 0$ then
3:         $a'_{i-1} = ...$
4:     end if
5: end for
6: $a'_{m-1} = 0$
```

Fig. 8: Description of the iteration (c).

offered a latency reduction by increasing the parallelism, but it also increases the occupied area (f). After two days, it was clear that our rotating buffers do not fit well on FPGA targets, hence they were abandoned. Anticipating this fact was not obvious. Dedicated memory components exist in our FPGA target (Xilinx Virtex 7 family): RAM blocks were therefore introduced in the design (g). This significantly reduced the area occupied by the Montgomery multiplier. In addition, these memories are large enough to store several integers. Operating frequency also improved slightly.

The Montgomery multiplier requires a register for the accumulator, which was hence moved into the BRAMs of its first operand thus simplifying the multiplier usage (h). This reduced the occupied area but was unfavourable for performances. Loop unrolling was useful for easiest parallelism of multiplications without data dependency, but it had a significant cost on area and it indirectly slew down the system frequency due too additional

```
Require:
    $a = (a_{m-1}, ..., a_0) \quad b = (b_{m-1}, ..., b_0)$
Ensure: $a \times b >> m \times$ digit width $\mod p$
1: for $i = 0$ to $m - 1$ do
2:     $r_b = b_i$                                  ▷ Read $b_i$
3:     for $j = 0$ to $m - 1$ do                    ▷ Body without data dep.
4:         $... = a_j \times r_b + ...$             ▷ Read $a_j$
5:     end for
6: end for

                         ⇓

1: for $k = -2$ to $m^2 - 1$ do                     ▷ Body without data dep.
2:     if $k >= 0$ then
3:         $... = a_j \times r_b + ...$             ▷ Read $a_j$
4:     end if
5:     every m cycles $r_b$ update: $r_b = b_i$     ▷ Read $b_i$
6:     compute $i$ and $j$ indexes for the next cycle
7: end for
```

Fig. 9: Description of the iteration (k).

routing. After having studied the generated unrolled scheduling, rewriting an internal loop of our multiplier for executing only operations without data dependency was easy (i). This code describes a hardware pipeline, hence loop unrolling for this particular portion of the code was forbidden (j). The pipelined loop is in a second loop: each loop is used to scan one of the operands. For each digit of the second operand, the final write-back of the pipeline was stalled. Figure 9 illustrates the loop merging (k). Less pipeline stalls reduce the pipeline latency, significantly reducing the global latency of our multiplier.

Finally, the Montgomery multiplier requires a conditional final subtraction, which has been deleted by relaxing the constraint on the range of integers (l). RAM blocks are inferred by Xilinx's Vivado from the VHDL generated by Augh whenever a register linked to a memory is found (i.e. an array in C). However, the synthesis tool sometimes chose to absorb this register into an input register of a DSP block instead of using it as output register for a RAM block. After solving this problem, some additional data was moved from LUTRAM to BRAM (m). The latency was reduced thanks to a shorter critical path, and some area was saved by using a more compact digit index coding (n). After a final compression of pipeline stages, the final circuit was built (o).

A lot of radical hardware architecture changes happened during the development: moving data into BRAMs; pipelining the Montgomery multiplier; algorithmic update; deletion of the final subtraction in the multiplier. There has also been changes in several parts of the design out of the multiplier. However, all these modifications were always related to small iterations in the C source code. A lot of small improvements that were done by iterative development on our high-level description would have been very time consuming in classical hardware design.

B. Related Work

In this paragraph, the results of the final architecture for computing the scalar multiplication (ECSM) are compared with state-of-the-art solutions. This architecture contains the final Montgomery multiplier obtained after the iterations previously detailed. It also contains other operations over $GF(p)$, point addition, point doubling, and the top-level scalar multiplication. All these components have been quickly designed using the same methodology.

ECSM computation time is selected for performance comparison. To evaluate the complete circuit cost, two metrics are proposed. The first one (named *unified metric*) takes into account BRAMs, DSPs, and slices. The number of BRAMs and DSPs is converted in equivalent-slices to allow a fair comparison. Conversion ratios are computed for each FPGA device as the ratio between the number of available slices and the number of available BRAMs and DSPs. The unified metric as seen in Figure 11 is the maximum of occupied equivalent-slices by BRAMs, DSPs, and really occupied slices. This unified metric and the convertion ratio can be calculated with the following formulas:

$$r_D = \frac{\#\text{slices}_{\text{target}}}{\#\text{DSPs}_{\text{target}}} \qquad r_B = \frac{\#\text{slices}_{\text{target}}}{\#\text{BRAMs}_{\text{target}}}$$

$$\text{unif. metric} = max(\#\text{slices}, r_D.\#\text{DSPs}, r_B.\#\text{BRAMs})$$

In the case of our target (xc7v585t) the ratios become:

$$r_{D\_585t} = \frac{91050}{1260} \simeq 72.3 \qquad r_{B\_585t} = \frac{91050}{1292} \simeq 70.5$$

If we consider the last circuit obtained after the iterations and synthesized for a 256-bit elliptic curve with a digit width of 32 bits (cf. Table I), the unified metric can be calculated as follows:

$$\text{unif. metric} = max(633, r_{D\_585t} \times 8, r_{B\_585t} \times 1) = 633$$

Another metric taking into account DSPs and BRAMs is proposed in [23]. Unfortunately, it uses FPGA-specific architectural information that is tedious to gather for the different targets presented here. The second circuit cost metric is classic: it is the number of occupied slices, where DSPs and BRAMs are not taken into account. The former metric is more adapted to the case where all system parts use BRAMs, DSPs and slices. In Figure 11, area is represented by the latter, which is for cases where BRAMs and DSPs are not used by other parts of the system. Thus, they can be considered as carrying no cost and only slices are included in the area cost. In order to compare area fairly, the study is limited to Xilinx Virtex families 5, 6 and 7 because in all these targets each slice has 4 6-input LUTs. For equitable performance comparisons, only implementations supporting 256-bit elliptic curves over the prime field are represented.

In order to obtain interesting area/performance ratios, multiplications over $GF(p)$ must be sequential. [24] exhibited the advantages of using DSP in this context on a Virtex 5 target. The number of occupied slices can be reduced from 9100 to

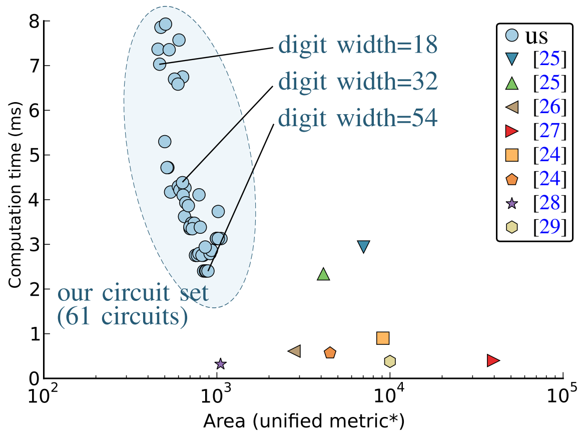

Fig. 10: ECSM Implementation efficiency vs. area for state-of-the-art circuits and our circuit set using final Montgomery multiplier implementation (*: *unified metric* is defined in IV-B)

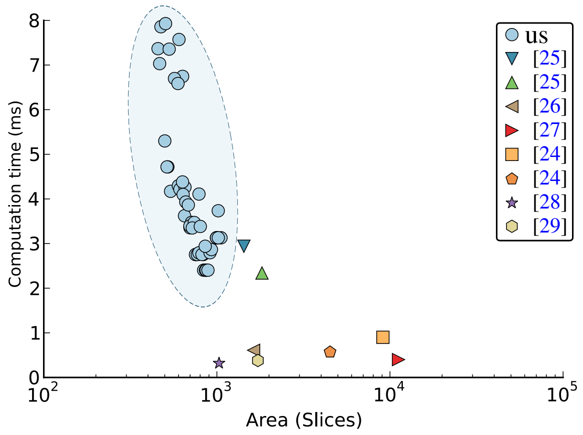

Fig. 11: ECSM Implementation efficiency vs. area (slices) for state-of-the-art circuits and our circuit set using final Montgomery multiplier implementation

4505 by using 16 DSPs. This number of DSPs is equal to 4320 equivalent-slices ($16 \times r_D = 16 \times \frac{51840}{192}$ for a target xc5vlx330). The usage of these specific components is reasonable because it is close to the number of occupied slices. In the same way, computation time is reduced to 0.59 ms for each scalar multiplication. This work is based on a manual tiling of DSPs by taking into account the asymmetry of these components. This tiling step should be repeated if the implemented circuit has to support a larger prime. In addition, the modular reduction is specific to a special prime number [9] and is not compatible with random curves.

To avoid the reconfiguration required to support multiples curves, it is possible to design crypto-processors supporting several curves. For example, [27] proposes an implementation supporting 5 NIST curves with different prime widths. It is based on modular operations over a restricted set of prime fields $GF(p)$. The proposed circuit is efficient (scalar multiplication in 0.4 ms for a 256-bit curve on a Virtex 6 target). On the other hand, a lot of resources are required because the design also supports a 521-bit curve: 11200 slices, 289 DSPs, and 128 BRAMs. In addition, the number of DSPs is large comparing to occupied slices. Our proposed area metric in Figure 10 is 39657 equivalent-slices whereas only 11200 slices are occupied. For an optimal resource occupation, this design could be used in conjunction with an application that does not consume DSPs. In order to minimize area, our choice is different because the generated designs support only one curve but an automatic and fast process permits easier curve update.

Another method for increasing the frequency is to use RNS arithmetic [26]. This can be more complex than classical arithmetic but more efficient due to shorter critical path (0.612 ms for a 256-bit scalar multiplication on a Kintex 7). In order to support a new curve with another prime, however, a new RNS base must be generated.

All our generated designs use less resources than the other circuits. Our most efficient circuit occupies a little less slices than designs proposed by [25] on Kintex 7. In this paper, a $GF(p)$ Montgomery Multiplier is used as well. The development was done at the RTL level whereas we used HLS and IID methods. For comparable performances, our circuits consume less area due to the larger exploration that is done by the HLS tool. Althought these circuits use as much slices as our best circuits, the unified metric (in Figure 10) illustrates their intense DSP usage. It should be pointed out that performance and area overheads in their circuits come in part from a protection against physical attacks.

It is possible to increase the Montgomery multiplier throughput. For example, a previous implementation on Virtex 5 target used quotient pipelining to achieve more parallelism [29]. This permits to compute a scalar multiplication in 0.376 ms with 1725 slices and 37 DSPs. If we want to generate more efficient circuits by increasing occupied resources, another iteration in our development may integrate this improvement on the multiplier pipeline. A simpler improvement may also be developed to increase parallelism, i.e. compute at the same time several multiplications. Data dependencies in basic point operations allow this but in order to minimize area this was left as future work.

Another very efficient circuit was proposed by Sasdrich and Günneysu on Zynq 7 target [28]. Data rolls between two memories through the arithmetic operator. The multiplier is based on a school method with an efficient reduction because the field is based on a pseudo Mersenne prime. This implementation can support only one specific curve (Montgomery curve). For this particular case, the number of prime field multiplications required for computing one point doubling and one point addition is reduced from 26 to 10. It seems difficult to achieve the same performance when targeting generic curves. A future iteration in the development focusing on specific curves and keeping our actual solution as fall-back for generic curves is always possible, but out of the scope of this paper.

C. Generated Set

Our generated circuits support an arbitrary generic elliptic curve contrary to some cited designs [27], [28]. We focused on 256-bit curves for comparison to the state-of-the art,

TABLE I: Area and performance of the scalar multiplication over curve P-256 [10] for an extract of our generated circuits

Digit width (bits)	**Slices**	**DSPs**	**BRAMs** (36 kb)	**Freq.** (MHz)	**computation time (ms)**
4	417	0	1	326	97.8
18	467	2	1	356	7.03
32	633	8	1	277	4.38
54	859	18	2	271	**2.4**
64	1019	32	2	208	3.13

but other security levels can be chosen. In addition, several crypto-processors with different area/performance ratios can be generated by modifying the digit width used in the large integer partitioning. The target is a Virtex 7 FPGA (XC7V585T) and all results are post place and route. Our circuits in Figures 10 and 11 are generated with digit widths from 4 to 64 on curve P-256 [10]. With the same high-level source code, our generated circuits occupy from 417 to 1019 slices (respectively with digit width 4 and 64) and can compute a scalar multiplication on 256-bit from 97.8 ms to 2.4 ms (respectively with digit width 4 and 54). A subset of all the generated circuits is characterized in Table I. The design with best performance uses 54-bit digits and computes the scalar-multiplication in 2.4 ms (651000 cycles at 271 MHz). The 64-bit digits is faster in term of cycles but the critical is longer hence computation time is stretched. Our designs consume less area than other designs thanks to the proposed method, while achieved performances are close to equivalent hand-made designs.

D. *Discussion*

The present contribution is focusing on FPGA for various reasons. A highly versatile architecture was required given the fast changing trends in elliptic curve standards. Nonetheless, it is possible to apply the presented method to any hardware design that lacks HLS support. Indeed, when a satisfying architecture is found after few iterations, the hardware developer can directly write this kind of architecture in HDL and proceed to the low level improvements that differentiates the output of an HLS tool from a fully hand-made design. With this solution, the developer can avoid potentially costly architecture changes and have final results quicker than with a classical method.

V. Conclusion

The reported case study demonstrates the efficiency of the proposed approach, inspired by some principles of Agile development methods. Thanks to HLS, powerful and flexible circuits can be obtained quickly, with various performance/area trade-offs. The proposed flow also enabled the generation of a highly versatile Elliptic Curve Cryptography coprocessor, adapted to any existing and future curve specification.

Future works include the parallelization of some architecture parts. This improvement could allow us to reach state-of-the-art latencies. Our verification based on a cryptographic library is application specific. Better interoperability by using standardized verification methodology could be beneficial. Moreover, proving equivalence of designs between iterations could further improve verification. Taking into account side channel vulnerabilities (in addition to constant computation time) during the high-level specification is another objective.

References

[1] P. Coussy and A. Morawiec, *High-level synthesis*. Springer, 2010.

[2] C. Larman and V. R. Basili, "Iterative and incremental development: A brief history," *Computer*, no. 6, pp. 47–56, 2003.

[3] V. Miller, "Use of elliptic curves in cryptography," in *Advances in CryptologyCRYPTO85 Proceedings*. Springer, 1986, pp. 417–426.

[4] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[5] N. Koblitz and A. Menezes, "A riddle wrapped in an enigma," IACR Cryptology ePrint Archive, Report 2015/1018, Tech. Rep., 2015.

[6] "Information technology - security techniques - digital signatures with appendix - part 3: Discrete logarithm based mechanisms," IOS, Geneva, Switzerland, ISO 14888-3, 1998.

[7] X.-F. S. American National Standards Institute, "Ansi x9.62, public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ECDSA)," September 1998.

[8] I. S. Assoc. *et al.*, "IEEE std 1363-2000, IEEE standard specifications for public-key cryptography," 2000.

[9] P. FIPS, "186-2. digital signature standard (DSS)," *National Institute of Standards and Technology (NIST)*, 2000.

[10] S. Certicom, "Sec 2: Recommended elliptic curve domain parameters," *Proceeding of Standards for Efficient Cryptography, Version*, vol. 1, 2000.

[11] I. S. Assoc. *et al.*, "IEEE std 1363a-2004, IEEE standard specifications for public-key cryptography - amendment 1: Additional techniques," 2000.

[12] E. Brainpool, "Ecc brainpool standard curves and curve generation," 2005, http://www.ecc-brainpool.org/download/Domain-parameters.pdf.

[13] D. J. Bernstein, "Curve25519: new diffie-hellman speed records," in *Public Key Cryptography-PKC 2006*. Springer, 2006, pp. 207–228.

[14] M. Hamburg, "Ed448-goldilocks, a new elliptic curve," Cryptology ePrint Archive, Report 2015/625, 2015, http://eprint.iacr.org/2015/625.

[15] A. K. Lenstra and B. Wesolowski, "A random zoo: sloth, unicorn, and trx," Cryptology ePrint Archive, Report 2015/366, 2015, http://eprint.iacr.org/.

[16] S. Vanstone, "Responses to NISTs proposal," *Communications of the ACM*, vol. 35, no. 35, pp. 50–52, 1992.

[17] ANSSI, "Mécanismes cryptographiques: Règles et recommandations concernant le choix et le dimensionnement des mécanismes cryptographiques," 2014, http://www.ssi.gouv.fr/uploads/2015/01/RGS_v-2-0_B1.pdf.

[18] "China cryptography administration: SM2 EC recommended parameters," 2010, http://www.oscca.gov.cn/UpFile/2010122214836668.pdf.

[19] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems," in *Advances in CryptologyCRYPTO96*. Springer, 1996, pp. 104–113.

[20] H. Cohen *et al.*, *Handbook of elliptic and hyperelliptic curve cryptography*. CRC press, 2005.

[21] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[22] A. Prost-Boucle, O. Muller, and F. Rousseau, "Fast and standalone design space exploration for high-level synthesis under resource constraints," *Journal of Systems Architecture*, vol. 60, pp. 79–93, 2014.

[23] E. Vansteenkiste, "Analyzing the divide between FPGA academic and commercial results," in *FPT*, 2015.

[24] D. B. Roy *et al.*, "Tile before multiplication: An efficient strategy to optimize DSP multiplier for accelerating prime field ECC for NIST curves," in *51st Design Automation Conference*. ACM, 2014, pp. 1–6.

[25] A.-T. Donda, "Efficient implementation of a generic coprocessor for elliptic curve cryptography on reconfigurable hardware," 2015, http://www.torsten-schuetze.de/reports/MasterThesis_Donda.pdf.

[26] J.-C. Bajard and N. Merkiche, "Double level montgomery cox-rower architecture, new bounds," in *Smart Card Research and Advanced Applications*. Springer, 2014, pp. 139–153.

[27] H. Alrimeih and D. Rakhmatov, "Fast and flexible hardware support for ECC over multiple standard prime fields," *VLSI Systems, IEEE Transactions on*, vol. 22, no. 12, pp. 2661–2674, 2014.

[28] P. Sasdrich and T. Güneysu, "Efficient elliptic-curve cryptography using curve25519 on reconfigurable devices," in *Reconfigurable Computing: Architectures, Tools, and Applications*. Springer, 2014, pp. 25–36.

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