This is an author produced version of Throughput/Area-efficient ECC Processor Using Montgomery Point Multiplication on FPGA.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/92952/

Article:

http://dx.doi.org/10.1109/TCSII.2015.2455992
Throughput/Area Efficient ECC Processor using Montgomery Point Multiplication on FPGA

Zia-Uddin-Ahamed Khan, Student Member, IEEE, Mohammed Benaissa, Senior Member, IEEE

Abstract—High throughput while maintaining low resource is a key issue for Elliptic Curve Cryptography (ECC) hardware implementations in many applications. In this paper, an ECC processor architecture over Galois Fields is presented that achieves the best reported throughput/area performance on FPGA to date. A novel segmented pipelining digit serial multiplier is developed to speed up ECC point multiplication. To achieve low latency, a new combined algorithm is developed for point addition and point doubling with careful scheduling. A compact and flexible distributed RAM based memory unit design is developed to increase speed while keeping area low. Further optimisations were made via timing constraints and logic level modifications at the implementation level. The proposed architecture is implemented on Virtex4 (V4), Virtex5 (V5) and Virtex7 (V7) FPGA technologies and respectively achieved throughout/slice figures of 19.65, 65.30 and 64.48 (10⁶ / (Seconds x Slices)).

Index Terms—Elliptic Curve Cryptography (ECC), Point Multiplication (PM), Field Programmable Gate Array (FPGA), Throughput per Area (throughput/area), Efficiency.

I. INTRODUCTION

Public key based information security networks use cryptography algorithms such as Elliptic Curve Cryptography (ECC) and RSA. ECC has emerged recently as an attractive replacement to the established RSA due to its superior strength-per-bit and reduced cost for equivalent security [1].

High speed ECC is a requirement for matching real-time information security, however, in many applications the hardware resource implications may be prohibitive and the required high speed performance would need to be achieved within a restricted resource performance.

FPGA based Hardware acceleration of ECC has seen a surge of interest recently. There are several state of the art FPGA implementations aimed at the high speed end of the design space [7 -13]. Most of these however use increased hardware resource to achieve the speed improvements sacrificing overall efficiency in terms of the throughput/area metric; such efficiency is desirable in many emerging low resource applications in particular in wireless communications. Area optimised high speed ECC design is challenging; there are requirements of algorithmic optimisation, careful scheduling to reduce clock cycles, size of multiplier, critical delay of the logic, and pipelining issues [7], [9].

In ECC, scalar point multiplication (PM) is the main operation. The PM can be implemented over either prime fields, GF(p) or binary extension fields, GF(2ⁿ) adopting either projective coordinates or affine coordinates. Binary extension fields called also finite fields (FFs) are more suited to hardware implementation due to their lower complexity FF multipliers, simple FF adder and single clocked FF squaring circuits. Projective coordinates are suited to throughput/area efficient ECC designs, where the costly inversion operation is avoided and the inversion operation required to convert projective into affine coordinates can be achieved by multiplicative inversion [2], [6].

ECC computations in the projective coordinates system are based on large operand finite field operations of which multiplication is the most frequently performed. The high speed performance of ECC designs therefore would depend mainly on the performance of the FF multipliers. Digit serial FF multipliers are often used to reduce latency; popular multipliers here include the direct method based multipliers and Karatsuba [7], [10]. If the field size is m and the digit size is w of a digit serial multiplier, then the number of clock cycles for each FF multiplication is s + c, where s = m/w, and c is for clock cycles due to data read-write operations. Thus, large digit multipliers can reduce clock cycles (latency) with increasing complexities of area and critical path delay. The critical path delay can be reduced using pipelining with some extra latency [9].

In this paper, we present an area-time (throughput/slice) efficient ECC processor over binary fields in projective coordinates on FPGA. We implement the Lopez-Dahab (LD) modified Montgomery algorithm for fast PM. We demonstrate a new “no idle cycle” [7] combined point operations (point addition and point doubling) algorithm to remove idle clock cycles in between two successive point operations. We schedule point operations very carefully to avoid the idle clock cycles due to data dependency, read-write operations, and pipelining. In addition, our efficient arithmetic circuit includes a digit serial multiplier, an adder and a square circuit. The presented arithmetic unit can support on-the-fly addition and square operations while performing FF multiplication. Moreover, we present an improved Most Significant Digit (MSD) serial multiplier utilizing segmented pipelining similar to the Least
Our proposed area optimized high throughput architecture is presented in Fig. 1. The design consists of an efficient arithmetic unit, an optimised memory unit and a dedicated control unit.

A. Segmented Pipelining Based Digit Serial Multiplier

The arithmetic unit design consists of a novel most significant digit (MSD) serial multiplier, a square and adder circuit as shown in Fig. 1. The performance of ECC depends mainly on the performance of the Digit serial multiplier in particular the speed.
TABLE I
LATENCY, CRITICAL PATH DELAY AND RESOURCES OF DIGIT SERIAL MULTIPLIERS OVER GF(2^m)

<table>
<thead>
<tr>
<th>Ref</th>
<th>Latency, cc</th>
<th>Critical path delay</th>
<th>#XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>[15]</td>
<td>2C</td>
<td>((2 + \log_2 d)T_X)</td>
<td>m + C(2d + S_1 + S_3) + d + S_2</td>
</tr>
<tr>
<td>[16]</td>
<td>m^log_2 a</td>
<td>((2 + 3b_1 g_m)T_X)</td>
<td>(20m^{\log_2 a} - 2m - \frac{1}{4}m^{\log_2 a} - \frac{1}{2} + 2m - 2)</td>
</tr>
<tr>
<td>[17]</td>
<td>2C</td>
<td>((2 + \log_2 d)T_X)</td>
<td>CS_2</td>
</tr>
<tr>
<td>Ours</td>
<td>(\left\lfloor \frac{m}{d} \right\rfloor)</td>
<td>(T_{MUX} + (\log_2 (n + k)) T_X)</td>
<td>m^{\log_2 a}</td>
</tr>
</tbody>
</table>

\(n = s = \# segments, d = \) digit size, \(S_1 = \left\lfloor \frac{m}{2} \right\rfloor (2.5d^{\log_2 a} - 3d + 0.5) + d^{\log_2 a} - d, S_2 = \left\lfloor \frac{m}{2} \right\rfloor (2d^{\log_2 a} - 2d), S_3 = \left\lfloor \frac{m}{2} \right\rfloor (2d^{\log_2 a}), C = \left\lceil \frac{m}{2} \right\rceil T_{MUX} = 2 \times 1 \) Mux delay.

Algorithm 2 Proposed combined loop operation of the LD

Montgomery point multiplication with careful scheduling

For \(i\) from \(1\) to \(2\) do If \(k_i = 1\) then

If \(k_{i+1} = 1\) then Point addition \(P(X_i, Z_i) = P(X_{i+1}, Z_{i+1})\) and Point Doubling: \(Q(X_{i+1}, Z_{i+1}) = 2Q(X_i, Z_i)\).

\(St1: Z_i \leftarrow X_i, Z_{i+1} \leftarrow Z_i;\)
\(St2: X_{i+1} \leftarrow X_i + Z_i, Z_{i+1} \leftarrow Z_i - T \leftarrow T_{\text{MUX}};\)
\(St3: X_i \leftarrow X_i, Z_{i+1} \leftarrow Z_i + T, T \leftarrow T_{\text{MUX}};\)
Conversion Step: same as Algorithm 1.

of the multiplier for a targeted level of latency. Digit serial multiplication for the high speed ECC implementation and tended to be either in direct form (i.e. MSD serial Multiplier) [10] or in bit parallel form (i.e. Karatsuba multiplier) [8], [9]. There are some advantages of Karatsuba multiplication over MSD multiplication. A Karatsuba FF multiplication takes \(s\)-1 cycles, where \(s = m/\omega\), and is suitable for pipelining. An MSD FF multiplication takes \(m + 1\) cycles where the extra clock cycle delay is due to the reduction register [2], [5] [9], and [10]. However, a pipelined Karatsuba multiplier based ECC implementation has been shown to achieve a lower clock frequency than a direct digit serial multiplier based implementation [7-8], [10].

For large MSD digit serial based ECC, pipelining is required which can affect latency in the point multiplications. In this work, we apply segmented pipelining to improve performance in MSD multiplication. In the segmented pipelining approach, a \(wxm\) digit serial multiplication is broken into sub digit serial multiplications called segmented multiplications \(w_1x_m, w_2x_m,...,w_nx_m\), where \(w = w_1 + w_2 + ... + w_n\). The segmented multiplication product is first saved in the register (Reg) before reduction into \(m\) bits using an interleaved reduction similar to that in the bit serial multiplier in [2]. The reduced \(m\) bit output of the reduction unit is saved in another Reg to use in the next cycles reduction or output. Thus, the proposed multiplication takes \(s + 2\) clock cycles where 1 extra clock cycle is due to the segmented pipelining, and the other additional clock cycle for pipelining after the reduction unit. A new input of the multiplier is inputs in every \(s\) clock cycles. Thus, a real time reset is required in every \(s\) cycles. We use multiplexers to select zero for segmented and save one clock cycle for the FF multiplication. Finally, the segmented pipelined multiplier takes one clock cycle for \(n\) segmentations without increasing area (slices) on the FPGA. The unused flip flops (FFs) in the combinational circuit of the multiplier are utilized in the pipelining [8].

To evaluate our proposed segmented multiplier, area and time complexity analysis is performed and presented in Table 1 which also includes comparison to state of the art digit serial multipliers reported in [15],[16], and [17]. For \(s = 4\) or less, our proposed multiplier shows same or better latency using similar or less resources. However, a key advantage of our proposed architecture is that we are able to achieve higher speed for the same (or less) area and the same (or less) latency; this is because our critical path delay can be modulated by the number of segmentations (\(n\)) with extra Flip Flop (FFs). The value of \(n\) defines the critical path delay of the multiplier. The path delay is either \(T_A + (\log_2 (d) T_X\) for the GF2MUL (M) or \(T_{MUX} + (\log_2 (n + k) T_X\) for the reduction part (Rd). Thus, our critical path delay can be optimised (to achieve the desirable high speed) by choosing an optimum number of segmentations (\(n\)). To generalise, from Table 1, the best figure latency for a field multiplication [15, 17] is \(2 \times \left\lfloor \frac{m}{d} \right\rfloor\), our multiplier’s latency is \(\frac{m}{d}\).

As a rule of thumb, therefore as long as \(m<4d\), our multiplier would achieve comparable or better latency figure. But what is crucial is that for comparable (less or higher) latency say and same digit size, our design can achieves improved critical path delay \(T_A + (\log_2 (\frac{d}{w}) T_X\) in our case (due to GF2MUL) compared to \(T_A + (\log_2 (d) T_X\) in [15,17] using an optimum segment size without increasing the latency of the multiplier. Thus, utilising similar area, our multiplier can achieve higher speed. At the extreme, the use a full precision multiplier (\(d = m\)) with an optimised segmentation would thus lead to the highest speed.

B. Optimized Memory Unit

High speed and flexible design for the memory unit can improve performance. We consider an optimised distributed RAM based memory unit. There is an \(8xm\) size register file in a unit, one \(m\) bit register (accumulator) and one shift register (ShiftReg). The \(8xm\) register file consists of one \(m\) bit input that can load data in any location of the register file, two \(m\) bit output buses (A bus and B bus) that can access data from any location of the register file. The shift register can store data from any location of the register file to provide \(w\) size digit (bi) multiplier for the FF multiplication. The accumulator can save a result from the arithmetic unit or new data from the register file to do a square operation. The accumulator and square circuit are connected such that repeated squaring can be done without saving in the register file. The repeated squaring improves
latency of multiplicative inversion as proposed in [6]. The memory unit is smartly accessible to write, read shifting operation in any location. The easy accessibility of the memory reduces the number of temporary registers for the PM. The memory unit consumes very low area to provide high speed data access.

C. Scheduling for point operations

In this paper, we propose new scheduling in the combined LD Montgomery point multiplication as shown in Algorithm 2. To schedule for no idle cycles, we combine the point addition and point doubling algorithms for the current value of $K_i = 1$ as shown in Algorithm 2. We observe that the product of the last multiplication is $X_1$ if $k_1 = 1$ or $X_2$ if $k_1 = 0$. Thus, the first multiplication of the loop should be independent of the last multiplication. For example, if the last product is $X_1$ then the next operands of multiplication are $X_2$ and $Z_1$. Otherwise, the next operands will be $X_1$ and $Z_2$. Thus, the first multiplication depends on the last $k_1$ which means the $k_{i+1}$ bit as shown in Algorithm 2.

Fig. 2 illustrate the proposed no idle state schedule using a 41-bit digit size FF multiplier. The 41 bit digit size FF multiplier takes $M = 4$ cycles for actual multiplication, and $c = 4$, with 2 clock cycles for pipelining and 2 clock cycles for unloading from and loading to the memory unit. In a loop, the point operation in the projective coordinates system requires 6 multiplications. To ensure no idle state in the multiplication, no new multiplication is started at every 4 clock cycles. Thus, two consequent but independent multiplications are overlapping each other as shown in Fig. 2 for $k_i = 1$ and $k_{i-1} = 1$.

Again, the adder circuit placed in the common data path is capable of doing addition concurrently. The square operation takes three cycles with 1 cycle to save in the accumulator, 1 clock cycle for squaring, and 1 clock cycle for loading. Repeated squaring can be done without storing in the register file. Thus, double squaring takes 4 clock cycles. Total Latency of the ECC is shown in the Table II.

![Fig. 2. Proposed careful scheduling (4 Clock cycles/multiplication)](image)

V. IMPLEMENTATION ON FPGA AND RESULTS

Our proposed efficient ECC processor is implemented over $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$, and $GF(2^{571})$, on different FPGA technologies namely Virtex4 (LX25_12 for f163, and LX100_12 for f233 to f571)), Virtex5 (XC5VLX50_3 for f163), and Virtex7 (VX550T_3 for f163, and V585_T for f233 to f571) using Xilinx tools versions 13.2 and 14.5 respectively. The design was implemented on Virtex4 and Virtex5 technologies to allow for a fair comparison to most relevant works, and on the Virtex7 to evaluate the performance on the newer technology. We present the implementation results after place and route in Table III. The Xilinx tools were used to set high speed properties and put subsequent timing constraints to improve the area-time product. The implementation results after place and route of our ECC designs are summarized in Table III. Table IV also includes area-time performance and comparison to state of the art.

As shown in Table IV, the main contribution of the segmentation in the multiplier is an increase in the clock frequency while segmentation in the multiplier is an increase in the clock frequency while

![Table II: Latency of ECC for m/µs = 4, MUL = M/M+, ADD = 1, SQR = 2](image)

![Table III: FPGA Implementation Results After Place & Route in Virtex7](image)
throughput/area efficiency. Particularly, our 3 segmented based design shows 65% better efficiency than [7]. Our f571 achieves 180 MHz speed while the work in [7] operates at a max speed of 107 MHz. One potential option of improving the area performance of [7] is to deploy an area efficient Karatsuba multiplier [16]; however, this would be at the expense of increased critical path delay. Another optimized ECC in [8] used full length (164 bit) word serial Karatsuba multiplier with pipelining and implemented on Virtex4 and Virtex5. The work in [8] uses four times bigger multiplier than ours to achieve 11.55 and 29.96 throughput/area on Virtex4 and Virtex5, respectively. Our 3 segmented 41 bit multiplier based design on virtex4 is 70% and the 2 segmented 41 bit multiplier based design on Virtex5 is 118% better than [8]. In [10], the reported best throughput/area efficiency is based on three 33 bit multipliers based ECC on Virtex5 shows 9.86 in throughput/LUTs ((1x10^6/s)/LUTs). Our 2 segmented multiplier based ECC shows 17.9 in (1x10^6/s)/LUTs is 82% better than the reported most efficient design in [10]. The hardware results presented in [11], [12], [13], and [14] use parallel multipliers to speed up their ECC designs show poor throughput/area efficiency due to the large area consumed. Finally, our single multiplier (41 bit) based ECC implementation on Virtex7 takes 10.51 µs for point multiplication is faster than the reported high speed work in [7], [9], [13], [14], and the work on the Virtex4 reported in [8], and is comparable to the work in [12] while of course using much lower resources.

VI. CONCLUSION

We proposed a highly efficient FPGA ECC processor design for high speed applications over GF(2^m). Key contributions include a novel high performance segmented pipelining MSD multiplication, a smart no-idle state scheduling that enables the clock cycles for loop operations in the point multiplication to depend only on the actual clock cycles of the FF multiplications, and a highly optimized memory unit design.

To our knowledge, our design achieves the best throughput/area efficiency figure on FPGA reported to date. The best throughput/area design achieved a figure of 65.30 (1x10^6/s)/(slices) that is performing an ECC point multiplication in 14.06 µs time whilst utilising only 1089 slices of area. The fastest design achieved 10.51 µs for a point multiplication using only 1476 slices.

REFERENCES


### TABLE IV

<table>
<thead>
<tr>
<th>NAME</th>
<th>Slides (Sls)</th>
<th>LUTs (Slices)</th>
<th>FPs</th>
<th>Clock Cycles (MHz)</th>
<th>Throughput (Kbps)</th>
<th>Throughput (Kbps)/Slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>GF571</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F51(V4)</td>
<td>34892</td>
<td>66594,6445</td>
<td>14250,107</td>
<td>133</td>
<td>0.22</td>
<td></td>
</tr>
<tr>
<td>F51(V5)</td>
<td>3515</td>
<td>10195, -</td>
<td>1414, 147</td>
<td>9.50</td>
<td>29.96</td>
<td></td>
</tr>
<tr>
<td>F51(V5)</td>
<td>6536</td>
<td>17305, 4075</td>
<td>3379, 262</td>
<td>12.90</td>
<td>11.86</td>
<td></td>
</tr>
<tr>
<td>Ours(2, V4)</td>
<td>1089</td>
<td>3958, 1522</td>
<td>4168, 296</td>
<td>14.06</td>
<td>65.30</td>
<td></td>
</tr>
<tr>
<td>Ours(3, V5)</td>
<td>1476</td>
<td>4721, 1886</td>
<td>4168, 397</td>
<td>10.51</td>
<td>64.48</td>
<td></td>
</tr>
<tr>
<td>F51(V4)</td>
<td>34892</td>
<td>66594,6445</td>
<td>14250,107</td>
<td>133</td>
<td>0.22</td>
<td></td>
</tr>
<tr>
<td>F51(V5)</td>
<td>3515</td>
<td>10195, -</td>
<td>1414, 147</td>
<td>9.50</td>
<td>29.96</td>
<td></td>
</tr>
<tr>
<td>F51(V5)</td>
<td>6536</td>
<td>17305, 4075</td>
<td>3379, 262</td>
<td>12.90</td>
<td>11.86</td>
<td></td>
</tr>
<tr>
<td>Ours(2, V4)</td>
<td>1089</td>
<td>3958, 1522</td>
<td>4168, 296</td>
<td>14.06</td>
<td>65.30</td>
<td></td>
</tr>
<tr>
<td>Ours(3, V5)</td>
<td>1476</td>
<td>4721, 1886</td>
<td>4168, 397</td>
<td>10.51</td>
<td>64.48</td>
<td></td>
</tr>
</tbody>
</table>