

Comparison of Two Implementations of Scalable Montgomery Coprocessor Embedded in Reconfigurable Hardware

Miloš Drutarovský, Viktor Fischer, and Martin Šimka

Abstract—This paper presents a comparison of two possible approaches for the efficient implementation of a scalable Montgomery Modular Multiplication (MM) coprocessor on modern Field Programmable Logic Devices (FPLDs). The first implementation uses data path based on traditionally used redundant carry-save adders, the second one exploits standard carry-propagate adder with fast carry chain logic not yet used in fully scalable designs. Both implementations use large embedded memory blocks available in recent FPLDs. Speed and logic requirements comparisons are performed on the optimized designs. The issues of targeting a design specifically for a FPLD are considered taking into account the underlying architecture imposed by the target FPLD technology. It is shown that carry-save adder is not an optimal building block for constrained scalable MM coprocessor in modern FPLDs.

Index Terms—Altera, FPGA, modular multiplier, Montgomery multiplication, RSA, scalable architecture, NIOS.

I. INTRODUCTION

MANY popular cryptographic algorithms, such as the RSA, ElGamal, Diffie-Hellman, etc. [1], make extensive use of modular exponentiation of long integers. However, it is a very slow operation when performed on a general-purpose computer since current typical operands (e.g. for RSA) have 1024, 2048, or more bits. The modular exponentiation is achieved by repeated modular multiplications. An efficient Modular Multiplication (MM) algorithm for the calculation of $(A \times B) \bmod M$ was developed by P. L. Montgomery [2]. A scalable MM design methodology in prime fields $GF(p)$ introduced in [3] forms the basis of the approaches presented in this paper. This design methodology based on Carry-Save Adders (CSA) [4] allows using a fixed-area modular multiplication circuit for performing MM of operands with (virtually) unlimited precision. The design tradeoffs for the best ASIC performance in a limited chip area of ASIC gates were analyzed in [3], [5].

A cheap and flexible modular exponentiation hardware accelerator can be also achieved using Field Programmable Logic Devices (FPLDs). Results presented in literature, e.g. [6]–[8] are mainly concentrated to systolic-like implementations that provide a very fast but less flexible solution.

M. Drutarovský and M. Šimka are with the Department of Electronics and Multimedia Communications, Technical University of Košice, Park Komenského 13, 04120 Košice, Slovakia (email: {Milos.Drutarovsky, Martin.Simka}@tuke.sk).

V. Fischer is with the Laboratoire Traitement du Signal et Instrumentation, Unité Mixte de Recherche CNRS 5516, Université Jean Monnet, 10, rue Barrouin, 42000 Saint-Etienne, France (email: fischer@univ-st-etienne.fr).

Current FPLDs provide an alternative hardware platform even for system-level integration of a cryptographic hardware. A System on a Configurable Chip (SOCC) can typically include an embedded processor with a set of dedicated coprocessors. For such a system a highly flexible (although typically slower) scalable MM coprocessor could be more attractive than a fixed length dedicated one.

The principal questions that motivated this paper are:

Is a CSA-based data path the best option for a scalable MM implementation in the modern FPLDs?

What is the best organization for the scalable architecture for the given FPLD resources?

To reply these questions, we consider such design aspects as an architecture, an effect of a word length, the number of pipelined stages, a size of Embedded Memory Blocks (EMBs), etc. on Altera FPLDs. Although these results are vendor specific, we believe that they can be generalized for other FPLDs.

This paper is organized as follows: Section 2 gives a brief discussion of the scalability of an arithmetic unit in the context of FPLDs application. Section 3 introduces the Montgomery method of MM, used notation and a possible algorithmic optimization. Multiple-word radix-2 MM algorithms suitable for scalable implementation are described in Section 4. Section 5 describes how the underlying architecture of the target FPLD may be utilized to produce an optimized design within constrained FPLD resources. Implementation results including final speed and area occupation of the hardware MM coprocessor designs as well as the software solution based on the embedded NIOS processor are presented in Section 6. Finally, concluding remarks are presented in Section 7.

II. SCALABILITY OF THE COPROCESSOR'S ARCHITECTURE IN FPLD

An arithmetic (cryptographic) unit is called scalable if it can be reused or replicated in order to generate long-precision results independently of the data precision for which the unit was originally designed [3]. The typical scalable coprocessor consists of two separate blocks - memory and w -bit data path as shown in Fig. 1.

Separation of the data path and the memory is the first fundamental difference from FPLD designs optimized for fixed-length operands, e.g. [7], [8]. There are large blocks of RAM (EMBs) available in modern FPLDs. They have the size of 2 or 4 kbits [9]–[13] in Altera FPLDs. The EMB

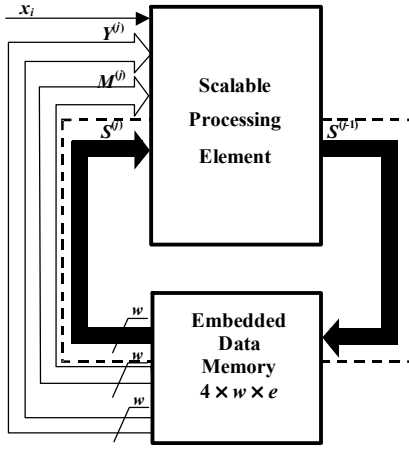


Fig. 1. Architecture of a general scalable coprocessor based on separate memory and w -bit datapath.

could be an ideal component to build a memory for a scalable MM coprocessor since its size is comparable to typical RSA operand sizes. FPLDs typically contain relatively large number of EMBs, which can be configured as true dual-port memories. Therefore our MM coprocessor design for FPLD will exploit these EMBs for data storing.

III. MONTGOMERY MULTIPLICATION

In the following text, the notation from [3], [5] is used. Given two integers X and Y , and the prime modulus M , the original Montgomery multiplication algorithm computes

$$Z = MM(X, Y) = (XYr^{-1}) \bmod M, \quad (1)$$

where $r = 2^n$, $X, Y < M < r$ and M is an n -bit number. This algorithm works for any modulus M provided that $\text{GCD}(M, r) = 1$. The basic radix-2 Montgomery multiplication algorithm for m -bit operands $X = (x_{m-1}, \dots, x_1, x_0)$, Y , and M is given as Algorithm 1.

Algorithm 1 The basic radix-2 Montgomery multiplication algorithm for m -bit operands $X = (x_{m-1}, \dots, x_1, x_0)$, Y , and M

```

1:  $S_0 = 0$ 
2: for  $i = 0$  to  $m - 1$  do
3:    $q_i = (S_i + x_i Y) \bmod 2$    ← is even?
4:    $S_{i+1} = (S_i + x_i Y + q_i M) / 2$ 
5: end for
6: if  $S_m \geq M$  then
7:    $S_m = S_m - M$    ← the final correction step
8: end if
9:  $Z = S_m$ 

```

This algorithm is suitable for hardware implementation because it is composed of simple operations: a word-by-bit multiplication, bit-shift (division by 2), and an addition. The test of an even condition is also very simple to implement; it consists of checking the least significant bit of the partial sum S_{i+1} followed by decision if the addition of M is required.

The described formulation of radix-2 algorithm was used as the starting point for derivation of scalable MM presented in [3], [5]. Instead of direct usage of Algorithm 1, several optimizations of original MM are taken from reference [14] to formulate Algorithm 2 (such formulation is used for fast implementation, e.g. in [7], [8]).

Algorithm 2 Optimized radix-2 Montgomery multiplication algorithm

```

1:  $S_0 = 0$ 
2:  $\hat{Y} = 2Y$ 
3: for  $i = 0$  to  $m + 2$  do
4:    $q_i = (S_i) \bmod 2$    (LSB of  $S_i$ )
5:    $S_{i+1} = (S_i + x_i \hat{Y} + q_i M) / 2$ 
6: end for
7:  $Z = S_{m+3}$ 

```

In Algorithm 2

$$X = \sum_{i=0}^m x_i 2^i = (0, 0, x_m, x_{m-1}, \dots, x_1, x_0) < 2M, \quad (2)$$

$$\hat{Y} = \sum_{i=0}^m \hat{y}_i 2^{i+1} = (y_m, \dots, y_1, y_0, 0) < 4M, \quad (3)$$

where $r = 2^{m+3}$, $X, Y < 2M$, and $2^{m-1} < M < 2^m$ is an m -bit number (the same as in the Algorithm 1). Note that \hat{Y} in (3) is the left shifted version of Y , with $\hat{y}_0 = 0$. This measure simplifies the computation of q_i compared to Algorithm 1. The loop of Algorithm 2 is executed three more times than in Algorithm 1 and it ensures that the inequalities $S_i < 3M$, $i = 0, 1, \dots, m + 2$ and $Z = S_{m+3} = MM(X, Y) = (XYr^{-m-3}) \bmod M < 2M$ always hold. The result of $Z = MM(X, Y)$ can thus be reused as an input X and Y for the next MM. This modification avoids the originally proposed final correction step (comparison and subtraction shown in the Algorithm 1) and makes a pipelined execution of the algorithm possible.

In typical applications (e.g. RSA), input operands X , Y are pre-multiplied by a factor $2^{2m} \bmod M$ (Algorithm 1) or $2^{2m+6} \bmod M$ (Algorithm 2). The final MM with value 1 makes the final result smaller than M (with probability $1 - 2^{-(m+2)}$ as shown in [7]) and provides the result $XY \bmod M$.

IV. MULTIPLE-WORD RADIX-2 MONTGOMERY MULTIPLICATION ALGORITHM

Operations in Algorithm 1 and Algorithm 2 are performed on full-precision operands and do not provide scalability shown in Fig. 1. Scalable algorithm requires a word-oriented processing. Let us consider w -bit words. For operands with m -bit precision, $e_1 = \lceil (m+1)/w \rceil$ words are required for Algorithm 1. An extra bit used in the calculation of e_1 is required since it is known that S_i (internal variable of radix-2 algorithm) is in the range $[0, 2M-1]$ where M is the modulus [3]. Thus the computations of Algorithm 1 must be done with an extra bit of precision. The input operands will need an

extra 0 bit value at the leftmost bit position in order to have the precision extended to the correct value.

Algorithm 2 requires $e_2 = \lceil (m+3)/w \rceil$ words in order to support extended range of input variables X , Y , and internal variable S_i . Note that in many practical configurations $e_1 = e_2$ and no additional words are required for Algorithm 2. The operands X will need two extra 0 bit values at the leftmost bit positions in order to have the precision extended to the $m+3$ cycles required by Algorithm 2. Note that in practical configurations $m \geq 1024$, so the difference in number of cycles is insignificant. On the other hand, the possibility to remove correction unit from hardware design of Algorithm 2 can simplify the hardware design.

A scalable algorithm in which the operand Y (multiplicand) is scanned word-by-word, and the operand X (multiplier) is scanned bit-by-bit was proposed in [3], [5]. It is called Multiple Word Radix-2 Montgomery Multiplication algorithm (MWR2MM) and it uses the following vectors:

$$\begin{aligned} M &= (M^{(e-1)}, \dots, M^{(1)}, M^{(0)}) \\ Y &= (Y^{(e-1)}, \dots, Y^{(1)}, Y^{(0)}) \\ S &= (S^{(e-1)}, \dots, S^{(1)}, S^{(0)}) \\ X &= (x_{m-1}, \dots, x_1, x_0) \end{aligned} \quad (4)$$

where the words are marked with superscripts and the bits are marked with subscripts. The concatenation of vectors a and b is represented as (a, b) . A particular range of bits in a vector a from position i to position j , $j > i$ is represented as $a_{j..i}$. The bit position i of the k -th word of a is represented as $a_i^{(k)}$. The details of the MWR2MM algorithm (in this paper it will be referred to as MWR2MM.CSA) are given in [3]. Algorithm 2 can be transformed to a multiple word form (referred here as MWR2MM.CPA) in a similar way.

Algorithm 3 The multiple word radix-2 Montgomery multiplication MWR2MM.CPA algorithm

```

1:  $S = 0$ 
2:  $\hat{Y} = 2Y$ 
3: for  $i = 0$  to  $m + 3$  do
4:    $C = 0$ 
5:    $q_i = S^{(0)}$ 
6:   for  $j = 1$  to  $e - 1$  do
7:      $(C, S^{(j)}) = C + x_i \hat{Y}^{(j)} + q_i M^{(j)} + S^{(j)}$ 
8:      $S^{(j-1)} = (S_0^{(j)}, S_{w-1..1}^{(j-1)})$ 
9:   end for
10:   $S^{(e-1)} = (C, S_{w-1..1}^{(e-1)})$ 
11: end for

```

In Algorithm 3, the notation from (4) is used with the exception of X which is given by (2). The MWR2MM.CPA algorithm features the same basic characteristics as the original MWR2MM algorithm. Thus, the MWR2MM.CSA as well as MWR2MM.CPA imposes no constraints on the precision of operands. The carry variable C must be from the set $\{0, 1, 2\}$. This condition is imposed by the addition of the three vectors S , M , and $x_i \hat{Y}$ [3].

MWR2MM.CSA and MWR2MM.CPA share the same data dependencies. Detailed analysis of potential parallelism and investigation of parallel organizations suitable for an MWR2MM.CSA algorithm implementation can be found in [3], [5]. It can be directly applied to the MWR2MM.CPA algorithm, too. The main result of this analysis - the possibility to operate in pipelined stages is used by FPLD implementations proposed in this paper. There are two possible approaches how to increase the speed of both algorithms:

- 1) To increase the word length w . Current FPLDs provide EMBs with dual port memory feature and configurable word lengths up to 16 bits. Since the capacity of EMBs is sufficient for typical RSA operands, it makes sense to use only one ($w \leq 16$) or two ($w \leq 32$) EMBs per variable. Usage of more EMBs per variable is for currently used operands (say, up to 4096 bits) and EMB sizes not reasonable. This is especially important for constrained SOCC designs.
- 2) To increase the number of pipelined stages. The hardware structure for both solutions is relatively simple and fast. An addition of several pipelined stages can increase the overall speed, especially if the access to the embedded memory is a bottleneck (as it is in a case of FPLD with limited routing resources for a large w). However, significant increase of number of pipelined stages necessitates a reduction of the complexity of one stage.

The main difference between the MWR2MM.CPA and MWR2MM.CSA algorithms is a non-redundant representation of all variables with the following consequences:

- 1) MWR2MM.CPA algorithm uses less (only 80% of MWR2MM.CSA) memory resources for the same operand sizes.
- 2) MWR2MM.CPA algorithm does not require final correction unit (MWR2MM.CSA algorithm requires at least final conversion to a non-redundant form).
- 3) MWR2MM.CPA algorithm allows a simpler computation of internal variable q_i that can allow (potentially) to simplify architecture of MWR2MM.CPA Processing Element (PE).
- 4) MWR2MM.CSA PE is always faster than MWR2MM.CPA one because it does not use carry at all. MWR2MM.CPA PE is slower but uses less Logic Elements (LEs) (so potentially within the same FPLD resources more MWR2MM.CPA PE pipelined stages can be used, what can in turn speed up the solution).

It is clear, that the speed of MWR2MM.CPA PE depends significantly on the word-length (the length of the carry chain). However, we can suppose that up to a certain word-length, $w \leq w_{max}$ the speed of MWR2MM.CPA PE is not critical, because the final speed is dominated by the embedded memory access time. The value w_{max} may differ between technologies due to the different routing and distinct physical layout. The unanswered question is if the w_{max} is larger in current FPLDs than 16 (or 32) bits required for economical usage of EMB resources as was explained in Section 2. This is analyzed in the next section.

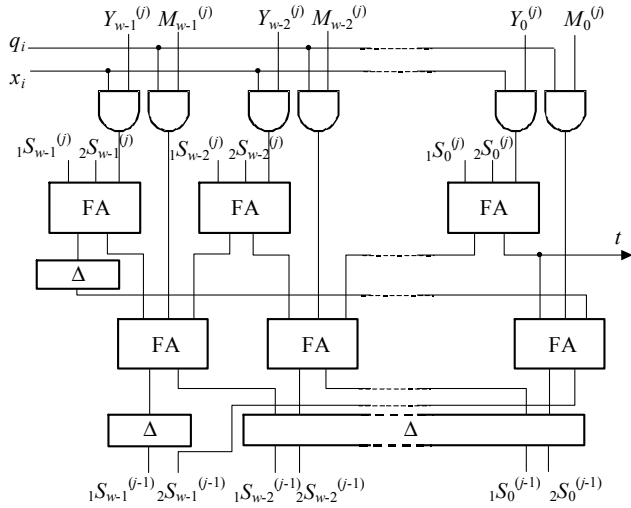


Fig. 2. Block diagram of CSA-based w -bit MWR2MM processing element CSA_PE based on full adders FA.

V. MWR2MM PROCESSING ELEMENTS PERFORMANCE

The whole computational complexity of both algorithms lies in the three additions of w -bit operands for computing S_{i+1} . As the propagation of w carries is (in general) too slow and an equivalent carry look-ahead logic requires too many resources, implementation of MWR2MM_CSA in [3] uses redundant carry-save representation [4]. MWR2MM_CSA w -bit Processing Element (CSA_PE) architecture based on Full Adders (FAs) is depicted in Fig. 2.

In order to reduce the storage and arithmetic hardware complexity, X , Y , and M are available in a non-redundant form. The intermediate internal sum S is received and generated in the redundant carry-save form ${}_1S, {}_2S$. A nice feature of this solution is an independence of the cycle time from the word length w . The cycle time may increase for larger w as a result of the broadcast problem only, it will not depend on the arithmetic operation itself. Intermediate results S_i are kept during computation in redundant form ${}_1S, {}_2S$. Conversion into binary representation is only done at the very end for feeding the intermediate result back as X or Y for a new computation (e.g. next iteration of modular exponentiation). The redundant representation of variables requires twice as much memory as a non-redundant representation. This is a drawback of the MWR2MM_CSA algorithm. However, it can be easily mapped into FPLD as it was done in [15], [16].

Recent FPLDs contain high speed interconnect lines between adjacent logic blocks which are designed to provide an efficient carry propagation. The PE architecture presented in this paper is optimal for implementation on any FPLD that has dedicated carry logic capability (e.g. modern Altera and Xilinx FPLDs). The basic organization of the data path consists of two layers of conventional Carry-Propagated Adders (CPA) as shown in Fig. 3.

MWR2MM Processing Element (CPA_PE) shown in Fig. 3 occupies smaller area than that used for MWR2MM_CSA (Fig. 2). The main advantage of the scalable architecture lies in the fact that the PEs can be easily repeated to increase throughput [3]. In this pipelined version several slightly modified PEs

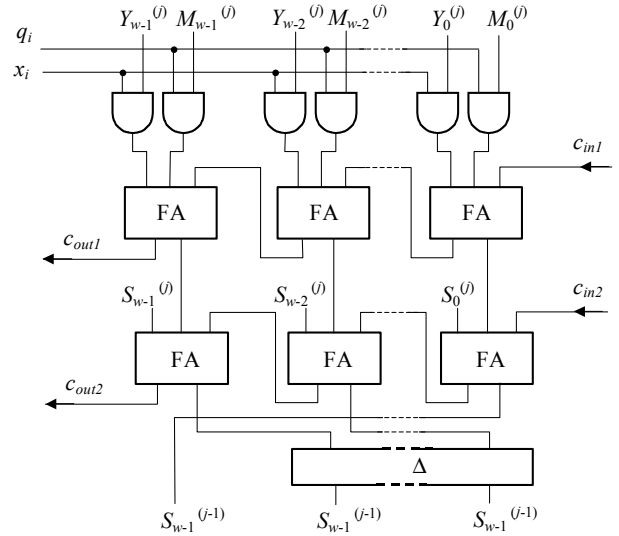


Fig. 3. Block diagram of CPA-based w -bit MWR2MM processing element CPA_PE based on full adders FA.

TABLE I
PE SIZES AND SPEEDS FOR OLD STYLE ALTERA FPLDS.

Device	CPA_PE			CSA_PE		
	w (bits)	Size (LEs)	Speed (MHz)	w (bits)	Size (LEs)	Speed (MHz)
ACEX [9]	8	66	161	8	81	232
EP1K100-1	16	130	129	16	161	202
	32	258	99	32	321	170
APEX [10]	8	59	161	8	81	232
EP20K160-1	16	115	129	16	161	202
	32	227	99	32	321	170

(some registers have to be added to allow temporary data storage) are connected in a cascade (see Fig. 4).

Tables I and II give the results of MWR2MM_CSA and MWR2MM_CPA PEs implementations (including data storage registers necessary for the pipelined version) in different Altera FPLDs for various word lengths, w . The results have been obtained with Altera Quartus development system, version 2.2. PEs have been implemented in VHDL. Carry chains have been realized using the *lpm_add_sub* function from the Library of Parameterized Modules (LPM) – a technology-independent library of logic functions that are parameterized to achieve scalability and adaptability.

There are several interesting facts that can be seen in

TABLE II
PE SIZES AND SPEEDS FOR NEW STYLE ALTERA FPLDS.

Device	CPA_PE			CSA_PE		
	w (bits)	Size (LEs)	Speed (MHz)	w (bits)	Size (LEs)	Speed (MHz)
CYCLONE [12]	8	59	277	8	81	304
EP1C20-6	16	115	235	16	161	304
	32	227	221	32	321	304
STRATIX [13]	8	59	271	8	81	304
EP1S10-6	16	115	248	16	161	304
	32	227	214	32	321	304

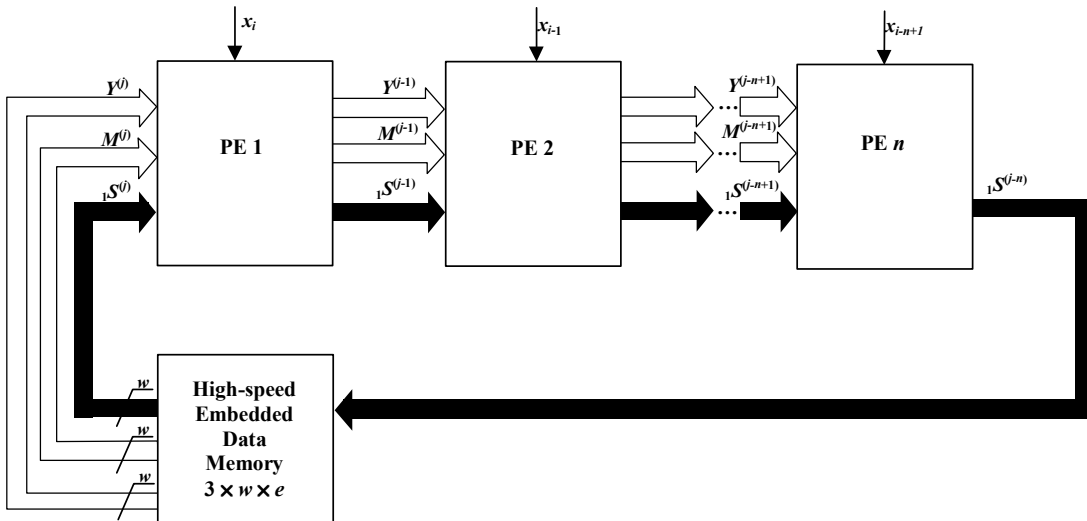


Fig. 4. Pipelined organization of the Montgomery modular multiplier coprocessor based on n -stage processing elements connection and separated embedded data memory.

these tables. With the exception of CPA_PE implemented in the ACEX family, the two solutions are technologically independent (as far as the area occupation is concerned). The size (in LEs) of the block depends almost linearly on the word length w . CSA_PE occupies always more resources than that of CPA_PE. The most important fact concerns the speed of the PEs. As it could be expected, the CSA_PE is always faster and the speed vary either only slightly (for old families) or almost not at all (for recent families, probably due to enhanced routing possibilities) with the word length w . However, the speed of the CPA_PE in the older families decreases significantly with the word length (about 40% from 8 bits to 32 bits). Recent Altera devices use enhanced carry chain. So-called carry-select chain uses the redundant carry calculation (hardwired) to increase the speed of carry functions. This feature enables to get processing times for CPA_PE comparable to CSA_PE (it is about 10 to 30% slower). Since CPA_PE is about 20% smaller, we can improve the final speed increasing number of pipelined stages. However, this approach does not seem to be adequate for word lengths $w > 32$ bits.

VI. IMPLEMENTATION RESULTS

An advantage of the use of the SOCC is that hardware and software solutions can be compared in a better way. In the system on the chip both software and hardware solutions occupy the same resources. The fully software solution needs relatively large logic resources and small memory resources to implement the processor and sometimes large memory to implement the program. The fully hardware solution needs greater logic resources and eventually some data memory. In a mixed hardware-software design, parallel and time critical operation can be done in hardware (coprocessor) and complex sequential and control operations in software (processor). In the system on the chip design the speedup factor of the coprocessor application in relationship to the entirely software-based solution can be measured quite easily: both implementations use the same embedded processor, e.g. Altera NIOS soft core.

Altera NIOS CPU [17] is a pipelined general-purpose RISC microprocessor. NIOS supports both 32-bit and 16-bit architectural variants. Both of them use 16-bit instructions. The processor has a five-stage pipelined structure with separate instruction and data-memory blocks (Harvard memory architecture). NIOS can include up to 512 internal general-purpose registers. The compiler uses the internal registers to accelerate subroutine calls and a local variable access. A 32-bit NIOS CPU can optionally be configured to include a hardware integer multiplier. This hardware is used by the MUL instruction to compute 32-bit result in three clock cycles¹. This option is not supported in the 16-bit NIOS instruction set. In order to obtain realistic comparisons, 32-bit NIOS CPU with hardware supported MUL instruction was used for software implementation.

We have realized three different systems: a) the first one was based on a fully software solution implemented on a 32-bit NIOS processor, b) the second version used 16-bit NIOS processor and the pipelined MWR2MM.CSA coprocessor, c) the third version used 16-bit NIOS and the pipelined MWR2MM.CPA coprocessor.

- 1) The software implementation of the MM algorithm has been written in the NIOS assembly language by using known optimization techniques for target processor. The Separated Operand Scanning (SOS) MM method [18] was used as the best method for given NIOS RISC architecture [19]. Table III shows the software MM timings for this fully software solution at 50 MHz. 32-bit NIOS processor had 2137 LEs and hardware integer multiplier (for MUL instruction) had 446 LEs.
- 2) In the mixed hardware-software design multiplication is realized in the hardware. Therefore we did not need the 32-bit version of the NIOS core. To reduce resources usage the second design has used the 16-bit NIOS processor, which was powerful enough to realize the necessary

¹When using the MUL option with Altera STRATIX devices, the hardware multiplier uses the STRATIX DSP blocks for implementation.

TABLE III

EXECUTION TIMES OF SOFTWARE IMPLEMENTATION OF MM ON ALTERA NIOS DEVELOPMENT BOARD (WITH APEX EP20K200-2X FPLD).

Length ($e \times w$)	Method	Multiplication (ms)	Squaring (ms)
1024	SOS32MEM	2.40	1.87
2048	SOS32MEM	9.47	7.24

TABLE IV

EXECUTION TIMES OF MIXED HARDWARE-SOFTWARE IMPLEMENTATION OF MM ON ALTERA NIOS DEVELOPMENT BOARD (WITH APEX EP20K200-2X FPLD) FOR MWR2MM_CSA PE.

Length ($e \times w$)	Method	Multiplication (ms)	Squaring (ms)
1024 = 64×16	MWR2MM_CSA	0.073	0.073
2048 = 128×16	MWR2MM_CSA	0.291	0.291

control operations. It had 1275 LEs. The coprocessor was based on a 16-bit ($w = 16$) MWR2MM_CSA PE with 6 pipelined stages. The coprocessor has thus occupied 1290 LEs. So the total area occupation of the second solution was comparable to that of the first solution. The processor has been clocked at 50 MHz and the coprocessor at 150 MHz. Times necessary for Montgomery multiplication and squaring are presented in Table IV.

- 3) The third design was the same as the second one except for type of the coprocessor. It was based on a 16-bit ($w = 16$) MWR2MM_CPA PE with 9 pipelined stages, so that it has occupied about the same area as the coprocessor based on MWR2MM_CSA PE in the previous design. The processor has been clocked at 50 MHz and the coprocessor at 100 MHz. The results obtained for this configuration are presented in Table V.

VII. CONCLUSION

In this paper we have evaluated two implementation methods of a scalable hardware Montgomery multiplier embedded in Altera FPLDs. It was shown that PE based on the conventional carry-propagated adders provides comparable speed results in implementation with constrained FPLD resources as originally proposed PE based on carry-save adders. The new method uses only 80% of the embedded FPLD memory resources required for the coprocessor based on carry-save adders. The proposed implementation method can be applied also for FPLDs from other vendors since it uses building

TABLE V

EXECUTION TIMES OF MIXED HARDWARE-SOFTWARE IMPLEMENTATION OF MM ON ALTERA NIOS DEVELOPMENT BOARD (WITH APEX EP20K200-2X FPLD) FOR MWR2MM_CPA PE.

Length ($e \times w$)	Method	Multiplication (ms)	Squaring (ms)
1024 = 64×16	MWR2MM_CPA	0.069	0.069
2048 = 128×16	MWR2MM_CPA	0.278	0.278

blocks generally available in modern FPLDs - high-speed dual-port embedded memories and fast carry-propagated logic.

ACKNOWLEDGMENT

This work has been done in the frame of the project CrypArchi included in the French national program ACI Cryptologie (project number CR/02 2 0041) and the project VEGA 1/1057/04.

REFERENCES

- [1] J. A. Menezes, P. C. Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. New York: CRC Press, Oct. 1996. [Online]. Available: <http://www.cacr.math.uwaterloo.ca/hac/>
- [2] P. L. Montgomery, "Modular multiplication without trial division," *Math. Computation*, vol. 44, pp. 519–521, 1985.
- [3] A. F. Tenca and C. K. Koc, "A scalable architecture for Montgomery multiplication," in *Cryptographic Hardware and Embedded Systems – CHES'99*, ser. Lecture Notes in Computer Science, C. K. Koc and C. Paar, Eds., no. 1717. Berlin, Germany: Springer-Verlag, Aug. 1999, pp. 94–108.
- [4] C. K. Koc, "RSA hardware implementation," RSA Laboratories, RSA Data Security, Inc., Tech. Rep., Aug. 1995.
- [5] A. F. Tenca and C. K. Koc, "A scalable architecture for modular multiplication based on Montgomery's algorithm," *IEEE Transactions on Computers*, vol. 52, no. 9, pp. 1215–1221, Sept. 2003.
- [6] S. E. Eldridge and C. D. Walter, "Hardware implementation of Montgomery's modular multiplication algorithm," *IEEE Trans. Comput.*, vol. 42, no. 6, pp. 693–699, 1993.
- [7] T. Blum and C. Paar, "Montgomery modular exponentiation on reconfigurable hardware," in *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, Koren and Kornerup, Eds. Los Alamitos, CA: IEEE Computer Society Press, April 1999, pp. 70–77.
- [8] A. Daly and W. Marnane, "Efficient architectures for implementing Montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic," in *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays FPGA'02*, Monterey, California, USA, Feb. 2002.
- [9] *ACEX 1K Programmable Logic Device Family, Data Sheet*, Altera Corporation, Sept. 2001, ver. 3.3.
- [10] *APEX 20K Programmable Logic Device Family, Data Sheet*, Altera Corporation, Feb. 2002, ver. 4.3.
- [11] *APEX II Programmable Logic Device Family, Data Sheet*, Altera Corporation, Aug. 2002, ver. 3.0.
- [12] *Cyclone Programmable Logic Device Family, Data Sheet*, Altera Corporation, Mar. 2003, ver. 1.1.
- [13] *Stratix Programmable Logic Device Family, Data Sheet*, Altera Corporation, Dec. 2002, ver. 3.0.
- [14] C. D. Walter, "Systolic modular multiplication," *IEEE Transactions on Computers*, vol. 42, no. 3, pp. 376–378, Mar. 1993.
- [15] M. Drutarovský and V. Fischer, "Implementation of scalable montgomery multiplication coprocessor in Altera reconfigurable hardware," in *Proceedings of the International Conference on Signal Processing and Multimedia Communications*, Košice, Slovakia, Nov. 2001, pp. 132–135.
- [16] V. Fischer and M. Drutarovský, "Scalable RSA processor in reconfigurable hardware – a SoC building block," in *Proceedings of XVI. Conference of Design of Circuits and Integrated Systems – DCIS 2001*, Porto, Portugal, Nov. 2001, pp. 327–332.
- [17] *NIOS 2.2 CPU, Data Sheet*, Altera Corporation, Sept. 2002, ver. 1.3.
- [18] C. K. Koc, T. Acar, and B. S. Kaliski, Jr., "Analyzing and comparing Montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, 1996.
- [19] V. Frolek, "Implementation of asymmetric encryption algorithms in reconfigurable circuits," Master's thesis, Technical University of Košice, Department of Electronics and Multimedia Communications, Jan.-May 2002.