Comparison of Scalable Montgomery Modular Multiplication Implementations Embedded in Reconfigurable Hardware

Milos Drutarovský, Martin Simka, Viktor Fischer

To cite this version:

Milos Drutarovský, Martin Simka, Viktor Fischer. Comparison of Scalable Montgomery Modular Multiplication Implementations Embedded in Reconfigurable Hardware. Acta Electrotechnica et Informatica, 2006, 6 (2), pp.37-45. ujm-00288982

HAL Id: ujm-00288982
https://ujm.hal.science/ujm-00288982

Submitted on 25 Jul 2008

COMPARISON OF SCALABLE MONTGOMERY MODULAR MULTIPLICATION IMPLEMENTATIONS EMBEDDED IN RECONFIGURABLE HARDWARE

[*]Miloš DRUTAROVSKÝ, [*]Martin ŠIMKA, [**]Viktor FISCHER
[*]Department of Electronics and Multimedia Communications, Technical University of Košice,
Park Komenského 13, 041 20 Košice, Slovak Republic,
E-mail: {Milos.Drutarovsky, Martin.Simka}@tuke.sk
[**]Laboratoire Traitement du Signal et Instrumentation, Unité Mixte de Recherche CNRS 5516, Université Jean Monnet,
10, rue Barrouin, 42000 Saint-Etienne, France, E-mail: fischer@univ-st-etienne.fr

SUMMARY

This paper presents a comparison of possible approaches for an efficient implementation of Multiple-word radix-2 Montgomery Modular Multiplication (MM) on modern Field Programmable Gate Arrays (FPGAs). The hardware implementation of MM coprocessor is fully scalable what means that it can be reused in order to generate long-precision results independently on the word length of the originally proposed coprocessor. The first of analyzed implementations uses a data path based on traditionally used redundant carry-save adders, the second one exploits, in scalable designs not yet applied, standard carry-propagate adders with fast carry chain logic. As a control unit and a platform for purely software implementation an embedded soft-core processor Altera NIOS is employed. All implementations use large embedded memory blocks available in recent FPGAs. Speed and logic requirements comparisons are performed on the optimized software and combined hardware-software designs in Altera FPGAs. The issues of targeting a design specifically for a FPGA are considered taking into account the underlying architecture imposed by the target FPGA technology. It is shown that the coprocessors based on carry-save adders and carry-propagate adders provide comparable results in constrained FPGA implementations but in case of carry-propagate logic, the solution requires less embedded memory and provides some additional implementation advantages presented in the paper.

Keywords: *Altera, FPGA, modular multiplication, Montgomery exponentiation, RSA, ECC, scalable architecture, NIOS*

1. INDRODUCTION

Many popular cryptographic algorithms, such as the RSA, ElGamal, Elliptic curve cryptography (ECC), Diffie-Hellman etc. [1] include 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 $AB \bmod M$ was developed by P.L. Montgomery [2].

A scalable MM design methodology in prime Galois Fields (GF) introduced in [3] forms the basis of approaches presented in the paper. This design methodology based on Carry-Save Adders (CSA) [4] allows using a fixed-area modular multiplication circuit for performing multiplication of (virtually) unlimited precision operands in radix-2. 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 Gate Arrays (FPGAs). Results presented in literature, e.g. [6-8] are mainly concentrated to systolic-like implementations that provide very fast but less flexible solution. Current FPGAs provide an alternative hardware platform even for system-level integration of complete cryptographic systems. A System on a Configurable Chip (SoCC) typically includes an embedded processor with a set of dedicated coprocessors. For SoCC a highly flexible (although typically slower) scalable MM coprocessor could be more attractive than the one with the fixed length.

Principal questions that motivated this paper are:

1. Is CSA-based data path the best option for a scalable MM implementation in modern FPGAs?
2. What is the best organization for a scalable architecture for given constrained FPGA resources?

To answer these questions, we consider such design aspects as the architecture, the effect of the word length, the number of pipelined stages, the size of Embedded Memory Blocks (EMBs), etc. on Altera FPGAs. Although these results are vendor specific they can be generalized also for other FPGAs (e.g. Xilinx).

This paper presents the results originally presented in [9] and is organized as follows: Section 2 gives a brief discussion on the scalability of an arithmetic unit in the context of FPGA application. Section 3 introduces a Montgomery method of MM, used notation and applied algorithmic optimization. Multiple-word radix-2 Montgomery multiplication algorithms suitable for scalable implementation are described in Section 4. Section 5 describes how the underlying architecture of the target FPGA may be utilized to produce an optimized design within constrained FPGA resources. Implementation results including final speed and area requirements of the hardware MM coprocessor designs as well as the pure software solution based on the embedded

Altera NIOS processor are presented in Section 6. Finally, concluding remarks are presented in the Section 7.

2. SCALABILITY OF COPROCESSOR ARCHITECTURE IN FPGA

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

Separation of the processing element with data path and the memory block is the first fundamental difference from FPGA designs optimized for fixed-length operands – e.g. [7, 8]. RAM in modern FPGAs is implemented in dedicated part of the device in the form of Embedded Memory Blocks (EMBs). In Altera devices they have size of 2 or 4 kbits [10-14]. The EMB could be an ideal component to build a memory for a scalable MM coprocessor since its size is comparable to typical RSA operand sizes. FPGAs typically contain relatively large number of EMBs, which can be configured as true dual-port memories. Therefore the proposed MM coprocessor designs for FPGA will exploit these EMBs for data storing.

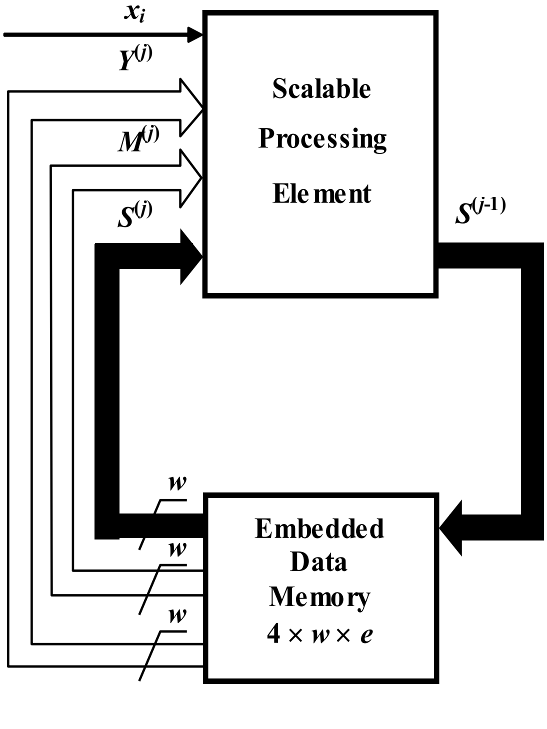


Fig. 1 Architecture of a general scalable coprocessor based on a separated memory and a processing element with w-bit data path width

3. MONTGOMERY MULTIPLICATION IN MONTGOMERY DOMAIN

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

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

where $r = 2^n$, $X,Y < M < r$ and M is an n-bit number. The algorithm is applicable for any modulus M provided that $\mathrm{GCD}(M,r) = 1$.

3.1. Montgomery exponentiation

Basic MM (1) can be used especially for efficient computation of modular exponentiation by the standard Montgomery exponentiation algorithm [1] ($E = (e_{t-1},\ldots,e_0)_2$ with $e_{t-1} = 1$, all other variables are k-bit integers) described in Alg. 1.

Alg. 1 The Montgomery exponentiation algorithm.

```
$\tilde{X} = \mathrm{MM}(X, R^2 \bmod M) = XR \bmod M$
$A = R \bmod M$
for $i = t-1$ downto 0 do
  $A = \mathrm{MM}(A, A)$
  if $e_i = 1$ then
    $A = \mathrm{MM}(A, \tilde{X})$
  end if
end for
$A = \mathrm{MM}(A, 1)$
```

The Montgomery multiplication is especially efficient when a sequence of MM is used in a row. The input X is transformed to Montgomery domain $\tilde{X}$ (or Montgomery base) in step 1 of Alg. 1. And then we use Montgomery form of MM instead of regular MM. The final result is transformed back to the real domain by Montgomery multiplication by 1 (since for any X, we have $\mathrm{MM}(\tilde{X},1) = X$).

3.2. Radix-2 Montgomery multiplication

The basic radix-2 Montgomery multiplication algorithm for m-bit operands $X = (x_{m-1},\ldots,x_1,x_0)$, Y, and M is given as Alg. 2[1]. Alg. 2 is suitable for the hardware implementation because it is composed of simple operations: a word-by-bit multiplication, right 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.

[1] In the rest of the paper we use m as a parameter for length of the input operands X,Y and M. The parameter n will be used for setting the $r = 2^n$. Optimized implementations use $n > m$.

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

```
$S_0 = 0$
for $i = 0$ to $m-1$
    $q_i = (S_i + x_i Y) \bmod 2$
    if $(q_i = 1)$ then
        $S_{i+1} = (S_i + x_i Y + M)/2$
    else
        $S_{i+1} = (S_i + x_i Y)/2$
    end if
end for
if $S_m \ge M$ then
    $S_m = S_m - M$        // the final correction step
end if
$Z = S_m$
```

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 the Alg. 2, several optimizations of original MM are taken from references [15, 16] to formulate the Alg. 3 (such formulation is used also for fast implementation, e.g. in [7, 8]).

Alg. 3 The optimized radix-2 Montgomery multiplication algorithm for $(m+3)$-bit operands $X = (0, 0, x_m, x_{m-1}, \ldots, x_1, x_0)$, Y and M.

```
$S_0 = 0$
$\hat{Y} = 2Y$
for $i = 0$ to $m+2$
    $q_i = S_i \bmod 2$
    if $(q_i = 1)$ then
        $S_{i+1} = (S_i + x_i \hat{Y} + M)/2$
    else
        $S_{i+1} = (S_i + x_i \hat{Y})/2$
    end if
end for
$Z = S_{m+3}$
```

In the Alg. 3

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

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

where $r = 2^{m+3}$, $X, Y < 2M$ (two times larger than in Alg. 2 so can be reused for the following MM in Alg. 1) and $2^{m-1} < M < 2^m$ is an m-bit number (the same as in the Alg. 2). Note that $\hat{Y}$ in (3) is the left shifted version of Y, with $\hat{y}_0 = 0$. This modification simplifies the computation of q_i compared to the Alg. 2. The loop of the Alg. 3 is executed three more times ($m+3$) than in the Alg. 2, what ensures that the inequalities

$$S_i < 3M, \quad i = 0, 1, \ldots, m+2 \qquad (4)$$

and

$$Z = S_{m+3} = \mathrm{MM}(X, Y) = \\ = \left(XY2^{-m-3}\right) \bmod M < 2M \qquad (5)$$

always hold.

The result of $Z = \mathrm{MM}(X, Y)$ in the Alg. 3 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 Alg. 2) and makes a pipelined execution of the Montgomery exponentiation algorithm much simpler.

In typical applications (e.g. RSA, ECC), input operands X, Y are transformed into Montgomery domain by pre-multiplication with a factor $2^{2m} \bmod M$ (Alg. 2) or $2^{2m+6} \bmod M$ (Alg. 3). The final MM with value 1 ($A = \mathrm{MM}(A, 1)$ in Alg. 1) makes the final result smaller than M (without any final correction step in Alg. 3) and provides the result $XY \bmod M$ [7, 16].

4. MULTIPLE-WORD RADIX-2 MONTGOMERY MULTIPLICATION ALGORITHM

Operations in Alg. 2 and Alg. 3 are performed on full-precision operands and do not provide scalability shown in Fig. 1. A 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 2. An extra bit used in the calculation of e_1 is required since it is known that S_i (the internal variable of radix-2 algorithm) is in the range $[0, 2M-1]$ where M is the modulus [3]. Thus the computations of Alg. 2 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.

The Alg. 3 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 the Alg. 3. The operand 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 Alg. 3. Note that in practical configurations $m \ge 1024$ (for RSA) or $m \ge 160$ (for ECC), so the difference in number of cycles is not significant. On the other hand, the possibility to remove correction unit from hardware design of Alg. 3 greatly simplifies the hardware design.

A scalable algorithm in which operands Y (multiplicand) and M (modulus) are 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:

$$\begin{aligned} M &= \left(M^{(e-1)},\ldots,M^{(1)},M^{(0)}\right), \\ Y &= \left(Y^{(e-1)},\ldots,Y^{(1)},Y^{(0)}\right), \\ S &= \left(S^{(e-1)},\ldots,S^{(1)},S^{(0)}\right), \\ X &= \left(x_{m-1},\ldots,x_1,x_0\right), \end{aligned} \tag{6}$$

where the words are marked with superscripts and the bits are marked with subscripts. The concatenation of vectors a and b is presented as (a,b). A particular range of bits in a vector a from position i to position j, $j > i$ is represented as $a_{j\ldots i}$. The bit position i of the k^{th} word of a is represented as $a_i^{(k)}$. The details of the MWR2MM algorithm (Alg. 4, in this paper it will be referred to as MWR2MM_CSA) are given in [3].

Alg. 3 can be transformed to a multiple word form (Alg. 5) in a similar way (here referred to as MWR2MM_CPA as its implementation is based on application of Carry-Propagate Adders (CPA)). In Alg. 5, the notation from (6) 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 $x_i Y$ [3].

MWR2MM_CSA and MWR2MM_CPA share the same data dependencies. Detailed analysis of algorithm implementation can be found in [3, 5]. It can be directly applied to MWR2MM_CPA algorithm, too. The main result of this analysis – the possibility to operate in pipelined stages is used by FPGA 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: Typical FPGAs provide the EMBs with dual port memory feature and configurable word lengths up to 16 bits. Since the capacity of EMBs is sufficient for typical RSA or ECC operands, in constrained designs it makes sense to use only one ($w \le 16$) or two ($w \le 32$) EMBs per algorithm variable. Usage of more EMBs per variable is not reasonable for currently used operands (say, up to 4096 bits) and EMB sizes. This is especially important for constrained SoCC designs.

Alg. 4 The multiple word radix-2 Montgomery multiplication MWR2MM_CSA algorithm.

```
$S = 0$
for $i = 0$ to $m-1$ do
  $\left(C, S^{(0)}\right) = x_i Y^{(0)} + S^{(0)}$
  if $S_0^{(0)} = 1$ then
    for $j = 1$ to $e-1$ do
      $\left(C, S^{(j)}\right) = C + x_i Y^{(j)} + M^{(j)} + S^{(j)}$
      $S^{(j-1)} = \left(S_0^{(j)}, S_{w-1..1}^{(j-1)}\right)$
    end for
    $S^{(e-1)} = \left(C, S_{w-1..1}^{(e-1)}\right)$
  else
    for $j = 1$ to $e-1$ do
      $\left(C, S^{(j)}\right) = C + x_i Y^{(j)} + S^{(j)}$
      $S^{(j-1)} = \left(S_0^{(j)}, S_{w-1..1}^{(j-1)}\right)$
    end for
    $S^{(e-1)} = \left(C, S_{w-1..1}^{(e-1)}\right)$
  end if
end for
```

Alg. 5 The multiple word radix-2 Montgomery multiplication MWR2MM_CPA algorithm.

```
$S = 0$
$\hat{Y} = 2Y$
for $i = 0$ to $m+3$ do
  $C = 0$
  $q_i = S^{(0)}$
  for $j = 1$ to $e-1$ do
    $\left(C, S^{(j)}\right) = C + x_i \hat{Y}^{(j)} + q_i M^{(j)} + S^{(j)}$
    $S^{(j-1)} = \left(S_0^{(j)}, S_{w-1..1}^{(j-1)}\right)$
  end for
  $S^{(e-1)} = \left(C, S_{w-1..1}^{(e-1)}\right)$
end for
```

2. To increase the number of pipelined stages N_p: 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 a case of FPGA with limited routing resources for a large w). However, significant increase of the number of pipelined stages necessitates a reduction of the complexity of one stage.

The main difference between the MWR2MM_CPA and the MWR2MM_CSA algorithms is a non-redundant representation of S_i variables in case of the MWR2MM_CPA with the following consequences:

1. The MWR2MM_CPA algorithm uses less (only 80% of MWR2MM_CSA) memory resources for the same operand sizes.

2. The MWR2MM_CPA algorithm does not require the final correction unit (MWR2MM_CSA algorithm requires at least the final conversion to the non-redundant form).

3. The MWR2MM_CPA algorithm allows a simpler computation of internal q_i variable that can (potentially) allow simplification of the architecture of the MWR2MM_CPA Processing Element (PE).

4. The MWR2MM_CSA PE is always faster than MWR2MM_CPA one because it does not use the carry at all. The MWR2MM_CPA PE is slower but uses less LEs (so within the same FPGA resources, more MWR2MM_CPA PE pipelined stages can be used, what can in turn speed up the complete solution).

It is clear, that the speed of the MWR2MM_CPA PE depends significantly on the word-length (the length of carry chain). However, we can suppose that up to a certain word-length $w \le w_{\max}$ the speed of the 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 whether the $w_{\max}$ is larger than 16 (or 32) bits required for economical usage of EMB resources in current FPGAs as it was explained in Section 2. This is analyzed in the next section.

5. MWR2MM PROCESSING ELEMENTS COMPARISON

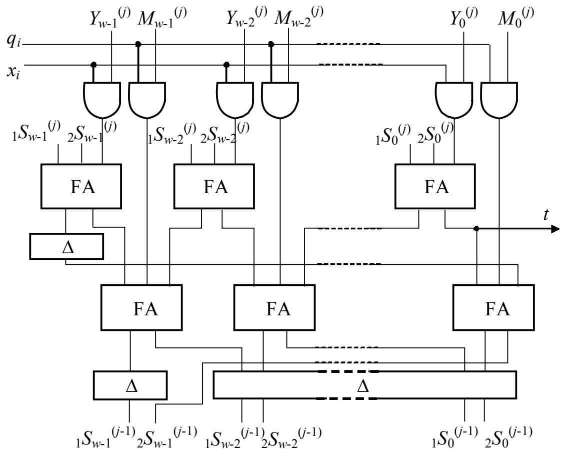
The whole computational complexity of both algorithms lies in the three additions of w-bit operands for computation of S_{i+1}. In the following part we describe pipelined implementations of the Alg. 4 and 5 optimized for constrained FPGA resources.

5.1. Architecture of processing elements

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]. A w-bit PE architecture implementing the MWR2MM_CSA algorithm (CSA_PE) based on Full Adders (FAs) is depicted in Fig. 2.

In order to reduce the storage and arithmetic hardware complexity, the operands 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 ${}_1S$, ${}_2S$.

Conversion into binary representation is done only 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 FPGA as it was done in [17, 18].



$Y_{w-1}^{(j)}$ $M_{w-1}^{(j)}$
$Y_{w-2}^{(j)}$ $M_{w-2}^{(j)}$
$Y_0^{(j)}$ $M_0^{(j)}$
q_i
x_i
${}_1S_{w-1}^{(j)}$ ${}_2S_{w-1}^{(j)}$
${}_1S_{w-2}^{(j)}$ ${}_2S_{w-2}^{(j)}$
${}_1S_0^{(j)}$ ${}_2S_0^{(j)}$
FA
FA
FA
t
Δ
FA
FA
FA
Δ
Δ
${}_1S_{w-1}^{(j-1)}$ ${}_2S_{w-1}^{(j-1)}$
${}_1S_{w-2}^{(j-1)}$ ${}_2S_{w-2}^{(j-1)}$
${}_1S_0^{(j-1)}$ ${}_2S_0^{(j-1)}$

Fig. 2 Block diagram of a CSA-based w-bit MWR2MM processing element CSA_PE based on full adders (FAs) [3]

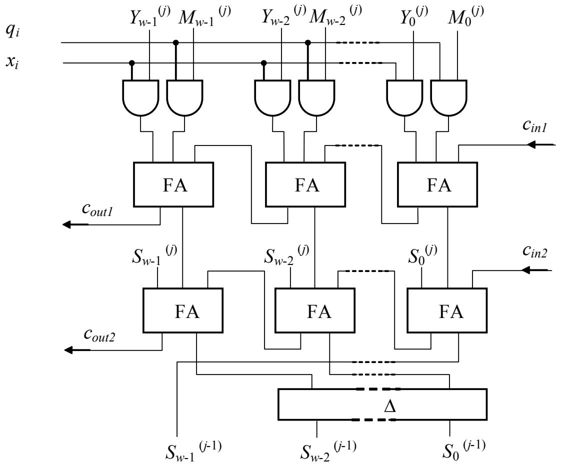


Fig. 3 Block diagram of a new CPA-based w-bit MWR2MM processing element CPA_PE based on FAs

Recent FPGAs contain high speed interconnecting lines between adjacent logic blocks which are designed to provide efficient carry propagation.

The PE architecture presented in this paper is optimal for implementation on any FPGA that has dedicated carry logic capability (e.g. modern Altera and Xilinx FPGAs). 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) occupies smaller area than that used for MWR2MM_CSA (Fig. 2) and uses less embedded memory.

5.2. Pipelined organization of PEs

The main advantage of the scalable architecture lies in the fact that the PEs can be easily repeated to increase throughput [3]. In the pipelined version of the multiplier N_p slightly modified PEs (some registers have to be added to allow temporary data storage) are connected in a cascade (for CPA_PEs see Fig. 4).

The maximum number of pipelined stages is found as:

$$N_{p_{\max}} = \left\lceil \frac{e+1}{2} \right\rceil \tag{7}$$

To keep the control logic simple the number of the stages N_p is restricted to values that divide the number of words e. In this way, the output of the last stage after computation is finished is the final result. The total computation time T (in clock cycles) when $N_p \le N_{p_{\max}}$ stages are connected in the cascade is

$$T = \left\lceil \frac{ew}{N_p} \right\rceil e + 2N_p \tag{8}$$

5.3. Performance analysis of scalable CSA_PE and CPA_PE

Tab. 1 and Tab. 2 give results of the MWR2MM_CSA and the MWR2MM_CPA PEs implementations (including data storage registers necessary for the pipelined version) in different Altera FPGAs for various word lengths w. The results have been obtained with Altera Quartus development system, version 4.0. PEs have been implemented in parameterized VHDL code. 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 these tables. With the exception of the CPA_PE implemented in the ACEX family, the two solutions are technologically independent (as far as the area occupation is concerned).

Tab. 1 PE sizes and speeds for old style Altera FPGAs

Device	CPA_PE			CSA_PE		
	w (bits)	Size (LEs)	Speed (MHz)	w (bits)	Size (LEs)	Speed (MHz)
ACEX [10] EP1K100-1	8	66	161	8	81	232
	16	130	129	16	161	202
	32	258	99	32	321	170
APEX [11] EP 20K160-1	8	59	161	8	81	232
	16	115	129	16	161	202
	32	227	99	32	321	170

Tab. 2 PE sizes and speeds for new style Altera FPGAs

Device	CPA_PE			CSA_PE		
	w (bits)	Size (LEs)	Speed (MHz)	w (bits)	Size (LEs)	Speed (MHz)
CYCLONE [13] EP1C20-6	8	59	277	8	81	304
	16	115	235	16	161	304
	32	227	221	32	321	304
STRATIX [14] EP1S10-6	8	59	271	8	81	304
	16	115	248	16	161	304
	32	227	214	32	321	304

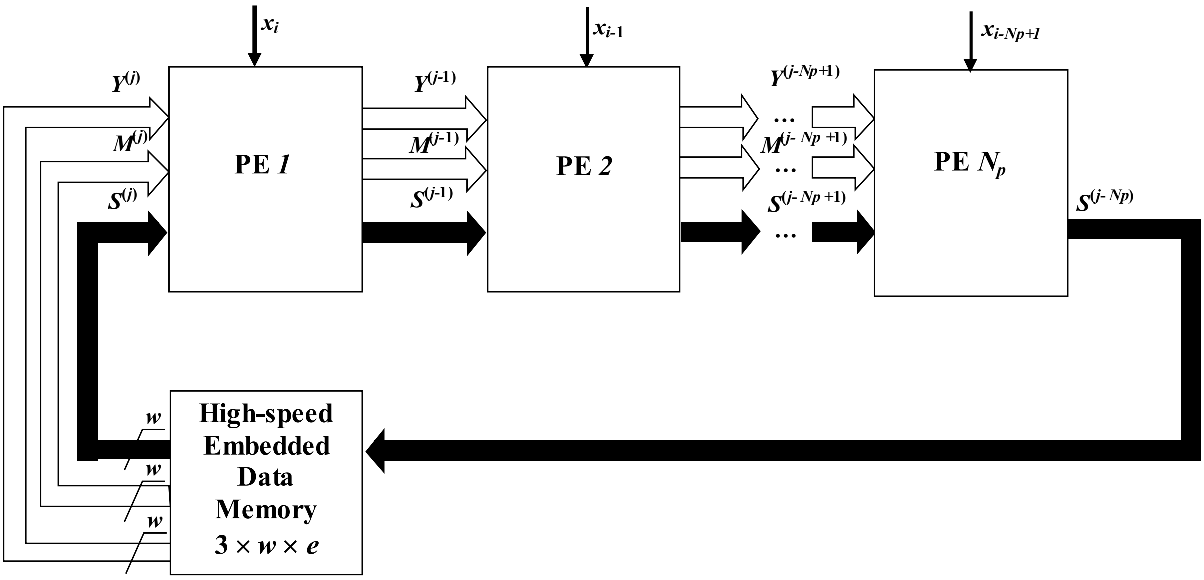


Fig. 4 Pipelined organization of the Montgomery MM coprocessor based on N_p CPA_PEs connection and separated embedded data memory

The size (expressed as a number of logic elements (LEs) in Altera FPGAs) of the block depends almost linearly on the word length w. The CSA_PE occupies always more resources than the 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 varies 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 an enhanced carry chain. The 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 the CPA_PE comparable to the CSA_PE (it is about 10 to 30% slower). Since CPA_PE is about 20% smaller, we can improve the final speed by increasing number of pipelined stages N_p. However, this approach does not seem to be adequate for the word lengths $w > 32$ bits.

6. 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 SoCC 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 larger logic resources and eventually some data memory. In a mixed hardware-software design, parallel and time critical operation can be done in the hardware (dedicated coprocessor) and complex sequential and control operations in the software (hard- or soft-core processor). In the proposed SoCC 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 [19] 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 cycles[2]. 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.

6.1. Execution times and implementation details

a) 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 [20] was used as the best method for given NIOS RISC architecture [21]. Tab. 3 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.

Tab. 3 Execution times of software implementation of MM on Altera NIOS development board (with APEX EP20K200-2X FPGA)

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

b) In the mixed hardware-software design multiplication is realized in the hardware as a coprocessor. 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 (with area occupation - 1275 LEs), which was powerful enough to realize the necessary control operations. The coprocessor was based on a 16-bit ($w = 16$) MWR2MM_CSA PE with $N_p = 6$ pipelined stages. The coprocessor has thus occupied additional 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 $F_{\text{NIOS}} = 50$ MHz and the coprocessor at $F_{\text{CSA\_MM}} = 150$ MHz. Times necessary for Montgomery multiplication and squaring are presented in Tab. 4.

Tab. 4 Execution times of hardware-software implementation of MM on Altera NIOS development board (with APEX EP20K200-2X FPGA) for MWR2MM_CSA PE

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

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

c) 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 $N_p = 9$ 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 $F_{\mathrm{NIOS}} = 50$ MHz and the coprocessor at $F_{\mathrm{CPA\_MM}} = 100$ MHz. The results obtained for this configuration are presented in Tab. 5.

Tab. 5 Execution times of hardware-software implementation of MM on Altera NIOS development board (with APEX EP20K200-2X FPGA) for MWR2MM_CPA PE

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

7. CONCLUSIONS

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

Both variants of the Montgomery multiplier (MWR2MM_CSA and MWR2MM_CPA) were successfully applied also in system implementations of RSA algorithm [18] and Elliptic Curve Method for factorization [22] (in this case Xilinx Virtex FPGA was chosen as a target platform). Thanks to the scalability of the MM the implementations are highly flexible and provide effective utilization of FPGAs resources.

ACKNOWLEDGEMENT

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

REFERENCES

[1] J.A. Menezes, P.C. Oorschot, S.A. Vanstone: Handbook of Applied Cryptography, CRC Press, New York, 1997.

[2] P.L. Montgomery: Modular Multiplication without Trial Division. Math. Computation, (44), pp. 519-521, 1985.

[3] A.F. Tenca, C.K. Koc: A Scalable Architecture for Montgomery Multiplication, In C.K. Koc and C. Paar, editors, Cryptographic Hardware and Embedded Systems, Lecture Notes in Computer Science, No.1717, pp. 94-108, Springer, Berlin, Germany, 1999.

[4] C.K. Koc: RSA Hardware Implementation. RSA Laboratories, version 1.0, pp. 1-28, August 1995, www.rsa.com.

[5] A.F. Tenca, C.K. Koc: A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm, IEEE Transactions on Computers, No 9., (52), pp. 1215-1221, September 2003.

[6] S.E. Eldridge, C.D. Walter: Hardware Implementation of Montgomery's Modular Multiplication Algorithm. IEEE Transactions on Computers, (42), pp. 693-699, June 1993.

[7] T. Blum, C. Paar: Montgomery Modular Exponentiation on Reconfigurable Hardware. Proceedings of the 14th IEEE Symposium on Computer Arithmetic, Adeline, Australia, pp. 70-77, 1999.

[8] A. Daly, W. Marnane: Efficient Architectures for Implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic. Proceedings of the 2002 ACM/SIGDA 10th International Symposium on Field-Programmable Gate Arrays FPGA'02, Monterey, California, USA, February 2002.

[9] M. Drutarovský, V. Fischer, M. Šimka: Comparison of Two Implementations of Scalable Montgomery Coprocessor Embedded in Reconfigurable Hardware. In Proceedings of the XIX Conference on Design of Circuits and Integrated Systems - DCIS 2004, pp. 240-245, Bordeaux, France, November 24-26, 2004.

[10] ACEX 1K Programmable Logic Device Family. Data Sheet, May 2003, ver. 3.4. www.altera.com

[11] APEX 20K Programmable Logic Device Family. Data Sheet, March 2004, ver. 5.1. www.altera.com

[12] APEX II Programmable Logic Device Family. Data Sheet, August 2002, ver. 3.0. www.altera.com

[13] Cyclone Device Handbook, Volume 1, March 2005, ver. 1.6. www.altera.com

[14] Stratix Device Handbook Volume 1, January 2005, ver. 3.2. www.altera.com

[15] C. D. Walter: Systolic Modular Multiplication, IEEE Transactions on Computers, no. 3, (42), pp. 376–378, March 1993.

[16] Colin D. Walter: Montgomery's Multiplication Technique: How to Make It Smaller and Faster. In C.K. Koc and C. Paar, editors, Cryptographic Hardware and Embedded Systems, Lecture Notes in Computer Science, No.1717, pp. 80-93, Springer, Berlin, Germany, 1999.

[17] M. Drutarovský, 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, pp. 132–135, November 2001.

[18] V. Fischer, 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, pp. 327–332, November 2001.

[19] NIOS 3.0 CPU, Data Sheet, November 2004, ver. 2.2. www.altera.com

[20] C. K. Koc, T. Acar, B. S. Kaliski, Jr.: Analyzing and Comparing Montgomery Multiplication Algorithms, IEEE Micro, no. 3, (16) pp. 26–33, 1996.

[21] V. Frolek: Implementation of Asymmetric Encryption Algorithms in Reconfigurable Circuits, Master’s thesis, Technical University of Košice, Department of Electronics and Multimedia Communications, January-May 2002.

[22] J. Pelzl, M. Šimka, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovský, V. Fischer, C. Paar: Area-time Efficient Hardware Architecture for Factoring Integers with the Elliptic Curve Method, IEE Proceedings - Information Security, Special Issue on Cryptographic Algorithms and Architectures for System-on-Chip, Vol. 152, No. 1, pp. 67-78, 2005.

BIOGRAPHIES

Miloš Drutarovský was born in 1965 in Prešov, Slovak Republic. He received the MSc degree in radioelectronics and PhD degree in electronics from Technical University of Košice, Slovak Republic, in 1988 and 1995, respectively. He defended his habilitation work - Digital Signal Processors in Digital Signal Processing in 2000. He is currently working as an Associated Professor at the Department of Electronics and Multimedia Communications, Technical University of Košice. His current research interests include applied cryptography, digital signal processing, and algorithms for embedded cryptographic architectures.

Martin Šimka was born in 1979 in Košice. He received his MSc degree in electronics and telecommunications in 2002 after defending his Master's Thesis - Conception of connection of embedded processor to arithmetic coprocessor in SOPC Altera. Currently he is a PhD student at the Department of Electronics and Multimedia Communications, Technical University of Košice and his main research area is an implementation of cryptographic blocks on FPGAs.

Viktor Fischer received the MSc and PhD degrees in electronics from Technical University of Košice, Slovak Republic, in 1981 and 1991, respectively. From 1982 to 1991 he was an Assistant Professor at the Department of Electronics, Technical University of Košice. Since 1991, he has been working at the Jean Monnet University of Saint-Etienne, France, as an Invited Professor in electronics and computer science. In the Laboratory Traitement du Signal et Instrumentation (TSI), UMR 5516 CNRS/University of Saint-Etienne, he works on signal and image processing, information security and embedded cryptographic systems. He is also currently working with company Micronic (Košice, Slovak Republic) oriented towards the development and production of data security hardware and software.