A Systolic Hardware Architectures of Montgomery Modular Multiplication for Public Key Cryptosystems

Amine MRABET[1,5], Nadia EL-MRABET[1,2], Ronan LASHERMES[2], Jean Baptiste RIGAUD[2], Belgacem BOUALLEGUE[6], Sihem MESNAGER[1,4] and Mohsen MACHHOUT[3]

[1]University of Paris8 - France
[2]Ecole des Mines de St Etienne - France
[3]University of Monastir - Tunisia
[4]University of Paris XIII, CNRS, UMR 7539 LAGA - France
[5]National Engineering School of Tunis - Tunisia
[6]King Khalid University - Saudi Arabia

Abstract The arithmetic in a finite field constitutes the core of Public Key Cryptography like RSA, ECC or pairing-based cryptography. This paper discusses an efficient hardware implementation of the Coarsely Integrated Operand Scanning method (CIOS) of Montgomery modular multiplication combined with an effective systolic architecture designed with a Two-dimensional array of Processing Elements. The systolic architecture increases the speed of calculation by combining the concepts of pipelining and the parallel processing into a single concept. We propose the CIOS method for the Montgomery multiplication using a systolic architecture. As far as we know this is the first implementation of such design. The proposed architectures are designed for Field Programmable Gate Array platforms. They targeted to reduce the number of clock cycles of the modular multiplication. The presented implementation results of the CIOS algorithms focuses on different security levels useful in cryptography. This architecture have been designed in order to use the flexible DSP48 on Xilinx FPGAs. Our architecture is scalable and depends only on the number and size of words. For instance, we provide results of implementation for 8, 16, 32 and 64 bit long words in 33, 66, 132 and 264 clock cycles. We highlight the fact that for a given number of word, the number of clock cycles is constant.

Keywords: Hardware Implementation, Modular Multiplication, Montgomery Algorithm, CIOS method, Systolic Architecture, DSP48.

1 Introduction

Since 1976, many Public Key Cryptosystems (PKC) have been proposed and all these cryptosystems based their security on the difficulty of some mathematical problem. The hardness of this underlying mathematical problem is essential for security. Elliptic Curve Cryptosystems which were proposed by Koblitz [11] and Miller [15], RSA [19] and the Pairing-Based Cryptography[10] are examples of PKCs. All these systems rely on an efficient finite field multiplication. As a consequence, the development of efficient architecture for modular multiplication has been a very popular subject of research. In 1985, Montgomery has presented a new method for modular multiplication [16]. It's one of the most suitable algorithm for performing modular multiplications in hardware and software implementations. The efficient implementation of the Montgomery modular multiplication in hardware was considered by many authors [17,9,3,6,18,20]. There are a variety of ways to perform the Montgomery multiplication, considering if multiplication and reduction are separated or integrated. The separated approach consists in first performing the product and then the Montgomery reduction. It was presented in 1996 by Koç and Tolga in [13]. This method is called the Separated Operand Scanning method (SOS). On the contrary, the integrated approach is characterized by an alternation between multiplication and reduction. Several integrated approaches are presented in [13]: the Coarsely Integrated Operand Scanning Method (CIOS), the Finely Integrated Operand Scanning Method (FIOS), the Finely Integrated Product Scanning Method (FIPS) and the Coarsely Integrated Hybrid Scanning Method (CIHS). According to Koç and Tolga in [13] the CIOS method is a scalable word-based method for Montgomery multiplication, and it is the most efficient algorithm that integrates the multiplication with reduction steps. A systolic array architecture [14,21] is one possibility for the implementation of the Montgomery algorithm in hardware [20,18,17,3]. These architectures offer Processing Elements (PE) array where each Processing Element performs arithmetic computation additions and multiplications. In accordance with the number of words used, the architecture can employ a variable number of PEs. The systolic architecture uses very simples Processing Elements. As a consequence, the systolic architecture decreases the needs for logic elements in hardware implementations. Our contribution in this work is to combine a systolic architecture, which is assumed to be the best choice for FPGA implementation, with the CIOS method of Montgomery modular multiplication. We optimize the number of clock cycles required to compute a n-bit Montgomery multiplication and we reduce the utilization of FPGA resources. We have implemented the modular multiplication in a fixed number of clock cycles. To the best of our knowledge, this is the first time that a hardware or a software multiplier of modular Montgomery multiplication, suitable for various security level, is performed in just 33 clock cycles. Furthermore, as far as we know, our work is the first one dealing with systolic architecture and CIOS method over large prime characteristic finite fields. This paper is organized as follows: Section 2 discusses related state-of-the-art works. Section 3 presents the Montgomery modular multiplication algorithm. The proposed architectures and results are presented in Section 4 and Section 5. Finally, the conclusion is presented in Section 6.

2 Brief state of the art

In hardware design, the systolic architecture [14] is a pipelined network arrangement of Processing Elements (or cells). It is a specialized form of parallel design. Each cell compute the data which is coming as input and calculate data independently. In [21] the authors proposed a systolic design for FPGA implementation. Several works are devoted to the implementation of the Montgomery multiplication [2,13,17,16,18,3,6,8,9,20]. The first ones to our knowledge who proposed a systolic array are Iwamura, Matsumoto and Imai [8,9]. They presented a systolic architecture that can execute a modular exponentiation using Montgomery multiplications. In [20] Tenca and Koç introduced a pipelined Montgomery modular multiplication, which has the ability to work in any given operand precision and which is adjustable to any chip area. Harris et al. in [4] improve the result of [20] using a systolic architecture for the Montgomery multiplication. Siddika Berna Örs, Lejla Batina, Bart Preneel and Joos Vandewalle presented in [17] a modular exponentiation based on the modular Montgomery. In [18] Guilherme Perin, Daniel Gomes Mesquita and Jõao Baptista Martins proposed a comparison between two modular multiplication architectures: a systolic and a very high-radix multiplexed implementation. Their approach uses a radix-16 and radix-32 decomposition. Both implementations targeted a Virtex-4 and a Virtex-5 FPGA. (A radix-n word is a word of size n.) Their work is the latest and the most efficient describing the use of a systolic approach for the Montgomery multiplication. We briefly recall the definition of a systolic architecture before a summary of their work. A systolic architecture is a pipelined network arrangement of PEs called cells. It is a specialized form of parallel computing, where cells compute the data which is coming as input and store them independently. A systolic architecture is an array composed of matrix-like rows of cells. Each PE shares the information with its neighbours immediately after processing. Cell at each step takes input data from one or more neighbours. The systolic architecture proposed in the work [18] is composed of s Processing Elements distributed in a one-dimensional array. The number s is the number of words. At each iteration of the Montgomery Algorithm, the words are read from an external memory (BRAM) and passed to their architecture. To evaluate the number of clock cycles for a Montgomery multiplication in the systolic architecture, they have to consider the first s cycles to read the input operands from RAM memories. Furthermore the first iteration of algorithm also needs s clock cycles. Finally the remaining iterations of algorithm are performed in $4 \times s$ clock cycles. As a consequence, this architecture requires a $6 \times s (= s + s + 4 \times s)$ clock cycles. For the multiplexed architecture, the first steps are identical to thus of the systolic architecture $(2 \times s)$. The number of clock cycles required to remaining iterations of Montgomery Algorithm is $6 \times s$ clock cycles. In order to perform the multiplexed architecture the algorithm requires $8 \times s (= 2 \times s + 6 \times s)$ clock cycles.

3 Montgomery Multiplication

Algorithm 1: Montgomery Modular Multiplication

Input: p an odd prime, $n = \lceil log_2(p) \rceil$, $R = 2^n$, $p' = -p^{-1} mod\ R$, $M(a)$, $M(b) \in \mathbb{F}_p$
Output: $M(ab) \mod p$
1 $\gamma \leftarrow M(a) \times M(b)$
2 $\delta \leftarrow \gamma \times p' \mod R$
3 $T \leftarrow \frac{\gamma + \delta \times p}{R}$
4 **If** $T \geq p$ **then** $T \leftarrow T - p$
5 **return** T

The Montgomery Multiplication Algorithm for large prime characteristic finite fields [16] is a method for performing modular multiplication without needing to divide by the modulus. In cryptography, the Montgomery Algorithm is the most used modular multiplication to perform the operation $a \times b \mod p$. The Montgomery multiplication transforms the division by p into several divisions by a power of 2, which consists only in shifts in hardware and software implementation. Furthermore, the Montgomery multiplication among large numbers can be constructed using a radix representation of the numbers. Let p be an odd prime number. Let $n = \lceil log_2(p) \rceil$ be the length of the binary decomposition of p. We choose the base of numeration to be $R = 2^n$, such that $p < R$. As p and R are coprime, we can define $p' = -p^{-1} \mod R$. The choice of R is motivated by the facts that $\gcd(R, p) = 1$ and reductions and divisions by R must be efficient. As R is a power of 2, divisions are right shifts and the modulo operation is a simple assignment of the first n-bit. Montgomery multiplication is performed with numbers represented in the Montgomery representation. The conversion from ordinary domain to Montgomery domaine detailed in Table 1 The map $M : a \in \mathbb{F}_p \to aR \in \mathbb{F}_p$ is a bijection and a field isomorphism of $\mathbb{F}_p$. For any element a of $\mathbb{F}_p$, the product $aR \in \mathbb{F}_p$ is called

Ordinary Domain	$\Longleftrightarrow$	Montgomery Domain
a	$\longleftrightarrow$	M(a)=a·R mod p
b	$\longleftrightarrow$	M(b)=b·R mod p
a·b	$\longleftrightarrow$	M(a·b)=a·b·R mod p

Table 1: Conversion between Montgomery and Ordinary Domains

the Montgomery representation of a in basis R and it is denoted $M(a)$. We describe the Montgomery multiplication in Algorithm 1. The Montgomery multiplication computes $M(a) \times M(b)$ and gives as result $M(ab)$.

3.1 CIOS Method

The Coarsely Integrated Operand Scanning (CIOS) method presented in Algorithm 2, improves the Montgomery Algorithm by integrating the multiplication and reduction. More specifically, instead of computing the product $a \cdot b$, then reducing the result, this method allows an alternation between iterations of the outer loops for multiplication and reduction. The integers (p, a and b) are seen as lists of s words of size w. In order to perform this algorithm we need an array T of size only $s+2$. The intermediate results are stored in T. The final result of the CIOS algorithm is composed by the $s+1$ least significant words of this array. The alternation between multiplication and reduction is possible since the value of m (in line 11 of the Algorithm 2) in the i^{th} iteration of the outer loop for reduction depends only on the value $T[j]$, which is computed by the i^{th} iteration of the outer loop for the multiplication. In order to perform the multiplication, we have modified the CIOS algorithm of [13] and designed this method with a systolic architecture. Indeed, instead of using an array to store the intermediate result, we replace T by Input and Output signals for each Processing Element. As a consequence, our design uses fewer of multiplexers and then we have better results considering the number of slices.

4 Hardware Implementation

4.1 Block DSP in Xilinx FPGAs

Modern FPGA devices like Xilinx Virtex-4, Virtex-5 and Artix-7 as well as Altera Stratix FPGAs have been equipped with arithmetic hardcore extensions to accelerate digital signal processing applications. These function DSP blocks can be used to build a more efficient implementation interms of performance and reduce at the same time the demand for areas. DSP blocks can be programmed to perform basic arithmetic functions, multiplication, addition and subtraction of unsigned integers. Figure 1 shows the generic DSP structure in advanced FPGAs. DSP can operate on external Input A,B and C as well as on feedback values from P or result PCIN.

Algorithm 2: CIOS algorithm for Montgomery multiplication [13]

Input: $p < 2^K$, $p' = -p^{-1} mod\ 2^w$, w, s , $K = s \cdot w$:bit length, $R = 2^K$, $a, b < p$
Output: $a \cdot b \cdot R^{-1}$ mod p

$T \leftarrow Null$;
for $i \leftarrow 0$ ***to*** $s-1$ **do**
  $C \leftarrow 0$;
  for $j \leftarrow 0$ ***to*** $s-1$ **do**
    $(C, S) \leftarrow T[j] + a[i] \cdot b[j] + C$
    $T[j] \leftarrow S$
  $(C, S) \leftarrow T[s] + C$
  $T[s] \leftarrow S$
  $T[s+1] \leftarrow C$
  $C \leftarrow 0$;
  $m \leftarrow T[0] \cdot p' \bmod 2^w$
  $(C, S) \leftarrow T[0] + m \cdot p[0]$
  for $j \leftarrow 1$ ***to*** $s-1$ **do**
    $(C, S) \leftarrow T[j] + m \cdot p[j] + C$
    $T[j] \leftarrow S$
  $(C, S) \leftarrow T[s] + C$
  $T[s-1] \leftarrow S$
  $T[s] \leftarrow T[s+1] + C$
return T;

4.2 Proposed Architecture

The idea of our design is to combine the CIOS method of Montgomery Modular multiplier presented in [13] with a two-dimensional systolic architecture in the model of [7,21]. As seen in section 3.1, the CIOS method is an alternation between iterations of the loops for multiplication and reduction. The concept of the two-dimensional systolic architecture presented in Section 2 combines an identical Processing Elements with local connections, which take external inputs and handle them with a predetermined manner in a pipelined fashion. This new architecture is directly based on the arithmetic operations of the CIOS method of Montgomery Algorithm. The arithmetic is performed in a radix-w base (2^w). The input operands are processed in s words of w bits. We present many versions of this method. We illustrate our design for $s = 8$, $s = 16$, $s = 32$ and a $s = 64$ architectures, respectively denoted NW-8 (for Number of Words), NW-16, NW-32 and NW-64. Before the descriptions of the architectures NW-8 and NW-16, we begin with a generic description of our systolic architecture. Our proposed architectures for the implementation of the Montgomery modular multiplication is detailed in this section. We describe it in detail as well as the different Processing Element behaviours. In order to have less of states in our Final State Machine (FSM), we divided our Algorithm 2 of Montgomery on five kinds of PE noted:

- cells alpha denoted α;
- cells beta denoted β;
- cells gamma denoted γ;
- cells alpha final denoted α_f;
- cells gamma final denoted γ_f.

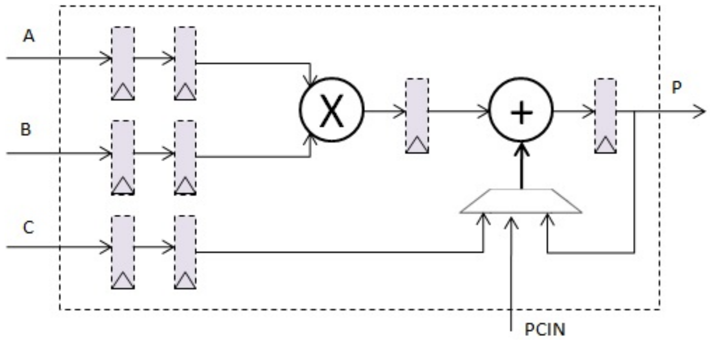

Figure 1: Structure of DSP block in modern FPGA device.

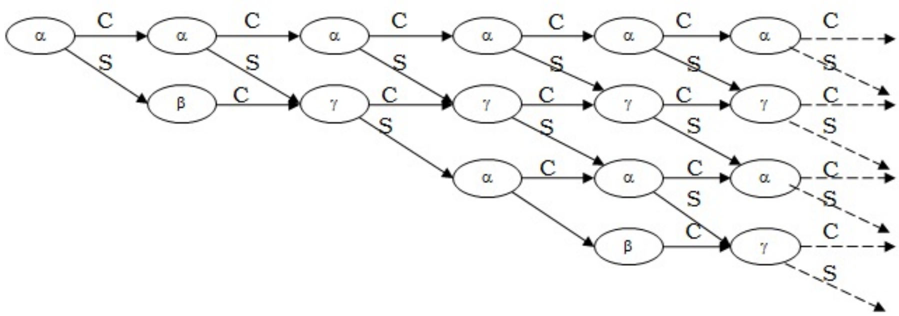

Figure 2: data dependency in general systolic architecture.

Figure 2 presents the dependency of the different cells. Below we describe precisely each cells. The letters MSB stand for the Most Significant Bits and LSB for the Least Significant Bits. In our notation the letter C denote the MSB of the results and the letter S the LSB.

1. alpha : Presented by the lines 4 and 5 in the Algorithm 2 and detailed in Algorithm 3 . The PEs alpha are scalable according to the NW in the design. We use this cell to perform the multiplication step. The input of the cell alpha are: S_In provided by the previous step, C_In provided by the previous step, $a[i]$: The words of the operand a, and $b[j]$: The words of the operand b. The output of the cell alpha are: S provided to the next step and C provided to the next step.
2. beta : Presented by the lines 9, 10 and 11 in the Algorithm 2 and detailed in Algorithm 4. The input of the cell beta are: S_In provided by the previous step, $p[0]$: The first word of the modulo p and p': predefined. The output of the cell beta are: m provided to the next step and C provided to the next step.
3. gamma : Presented by the lines 13 and 14 in the Algorithm 2 and detailed in Algorithm 5. The PEs gamma are scalable according to the NW in the design. We use this cell to perform the reduction step. The input of the cell gamma are: S_In provided by the previous step, C_In provided by the previous step, $p[j]$: The words of the modulo p and m provide by the cell beta. The output of the cell gamma are: S provided to the next step and C provided to the next step.

4. alpha_final : Presented by the lines lines 6, 7 and 8 in the Algorithm 2 and detailed in Algorithm 6. The input of the cell alpha_final are: S_In provided by the previous step and C_In provided by the previous step. The output of the cell alpha_final are: $S1$ provided to the next step and $S2$ provide to the next step.
5. gamma_final : Presented by the lines 15, 16 and 17 in the Algorithm 2 and detailed in Algorithm 7. The input of the cell gamma_final are: $S1\_In$ provided by the previous step, $S2\_In$ provided by the previous step and C_In provided by the previous step. The output of the cell gamma_final are: $S1$ provided to the next step and $S2$ provided to the next step.

Algorithm 3: Cell alpha

Input: $a[i]$, $b[j]$, C_In, S_In
Output: C, S
$tmp1 \leftarrow S\_In + C\_In$
$tmp2 \leftarrow a[i] \cdot b[j]$
$tmp2 \leftarrow tmp2 + tmp1$
$C \leftarrow$MSB($tmp2$)
$S \leftarrow$LSB($tmp2$)
return C, S;

Algorithm 4: Cell beta

Input: S_in, $p[0]$, $p' = -p^{-1} mod\ 2^w$
Output: C, m
$tmp1 \leftarrow S\_in \cdot p'$
$m \leftarrow LSB(tmp1)$
$tmp1 \leftarrow p[0] \cdot m$
$tmp1 \leftarrow S\_in + tmp1$
$C \leftarrow$MSB($tmp1$)
return C, m;

Algorithm 5: Cell gamma

Input: $p[i]$, m, C_in, S_in
Output: C, S
$tmp1 \leftarrow S\_in + C\_in$
$tmp2 \leftarrow p[i] \cdot m$
$tmp2 \leftarrow tmp2 + tmp1$
$C \leftarrow$MSB($tmp2$)
$S \leftarrow$LSB($tmp2$)
return C, S;

Algorithm 6: Cell alpha_final

Input: C_in, S_in

Output: $S1$, $S2$

$tmp1 \leftarrow S\_in + C\_in$

$S1 \leftarrow$LSB($tmp1$)

$S2 \leftarrow$MSB($tmp1$)

return C, S;

Algorithm 7: Cell gamma_final

Input: C_in, $S1\_in$, $S2\_in$

Output: $S1$, $S2$

$tmp1 \leftarrow S1\_in + C\_in$

$S1 \leftarrow$LSB($tmp1$)

$S2 \leftarrow$MSB($tmp1$)

$S2 \leftarrow S2\_in + S2$

return $S1$, $S2$;

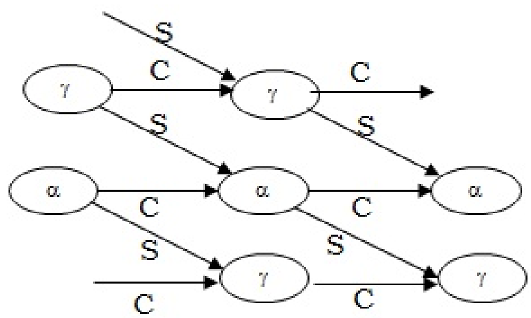

Figure 3: PEs of Systolic Architecture in two-dimensional array.

This organization allows us to optimize the number of clock cycles. Each Processing Element in Figure 10 is responsible for performing arithmetic operations. The different Processing Elements establish communication with the control block (FSM) as shown in Figure 9 by receiving starts signals at each state of Montgomery Algorithm iteration. Each PE sends a done signal to the FSM at each end of the calculation. The final result is a concatenation of the last output of gamma and gamma_final PEs. The structure of all PEs have a combinational behaviour.

4.3 Internals architectures of cells

In this section we will describe the internals architectures of PEs used in these designs. Our five cells are designed in order to use DSP(s) blocks.

Description of the cell α As illustrated in Figure 4, the multiplication between $a[i]$ and $b[j]$ words returns a $2w$ bits result. This result is added thereafter to S_α_In. This latter is the least significant bits of the result of Processing Element gamma, which is provided through the output multiplexer. The last add is also added to C_α_In. The C_α_In is the most significant bits of the result of the previous Processing Element alpha, which is provided also through an output of a second multiplexer. The different inputs outputs of the PE alpha are presented in Figure 9. The most significant bits of the result of alpha is propagated to the multiplexer to fix the next PE of alpha. Whereas the least significant bits are propagated to an other multiplexer to fix the next PE of gamma. After each computation of the alpha PE a shift in the input b is triggered.

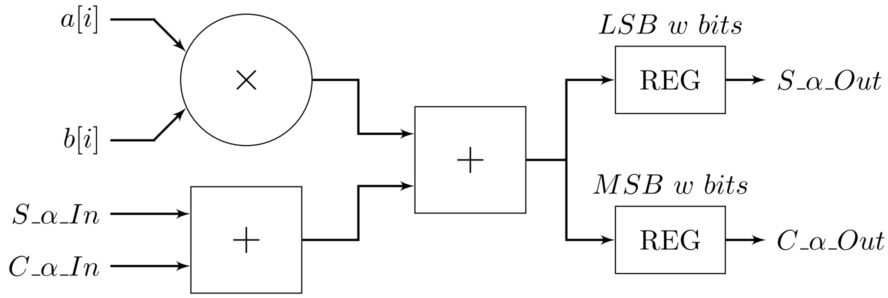

Figure 4: Alpha Processing Element internal architecture.

Description of the cell $\boldsymbol{\beta}$ According to our algorithm 4 and as illustrated in Figure 5, the zero index word of p ($p[0]$) and p' are provided to this beta Processing Element. The number p' corresponds the modular inverse of p modulo 2^w. The multiplication between p' and S_β_In returns a $2w$ bits result, where only the least significant bits of this multiplication is multiplied by the first word of p and returns a $2w$ bits result. Finally, this result is added to a w bits word S_β_In. Only the most significant bit part of this result is used in the next gamma PE. The different inputs/outputs of PE beta are presented in Figure 9.

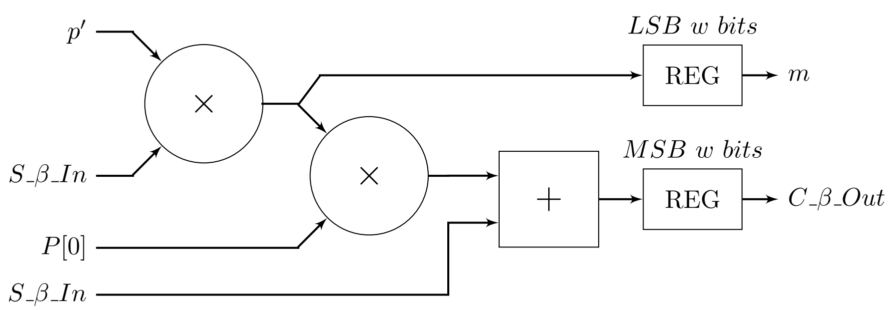

Figure 5: Beta Processing Element internal architecture.

Description of the cell $\boldsymbol{\gamma}$ As illustrated in Figure 6, the multiplication between m and $p[j]$ words returns a $2w$ bits result. This latter is added thereafter to S_γ_In. The number S_γ_In

corresponds to the least significant bits of the result of Processing Element alpha, which is provided through an output multiplexer. This add is also added to C_γ_In, where C_γ_In is the most significant bits of the result of the previous Processing Element gamma. This PE gamma is provided also through an output of a second multiplexer. The different inputs/outputs of the gamma PE are shown in Figure 9. The most significant bits of result are propagated to the multiplexer to fix the next PE of gamma. Whereas the least significant bits are propagated to an other multiplexer to fix the next PE of alpha.

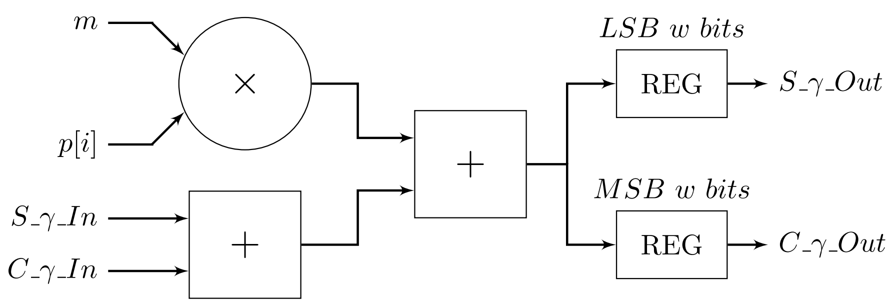

Figure 6: Gamma Processing Element internal architecture.

Description of the cell $\boldsymbol{\alpha_f}$ The cell α_f corresponds to the final α computed at the end of the line correspond to the multiplication step. In the PE alpha_final, the $S\_\alpha\_f\_In$ added to C_α_f returns a $2w$ bits result as presented in Figure 7.

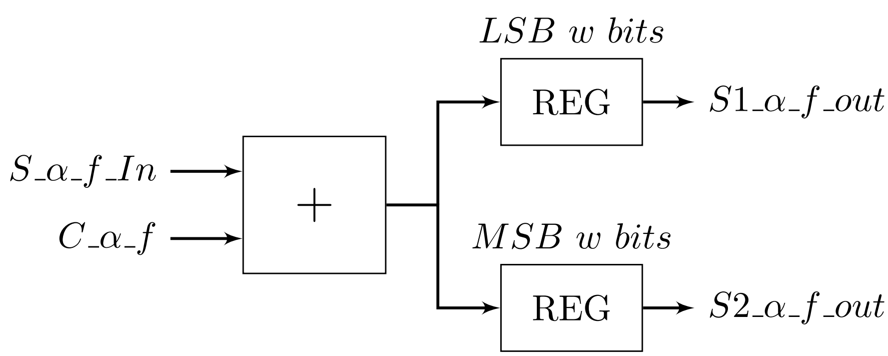

Figure 7: Alpha_f Processing Element internal architecture.

Description of the cell $\boldsymbol{\gamma_f}$ The cell γ_f corresponds to the final γ computed at the end of the line correspond to the reduction step. For Processing Element gamma_final, $S1\_\gamma\_f\_In$ is added to C_γ_f, the result is a $2w$ bits. The least significant bits of the last result is added to $S2\_\gamma\_f\_In$. The internal architecture of the gamma_final type PE is presented in Figure 8.

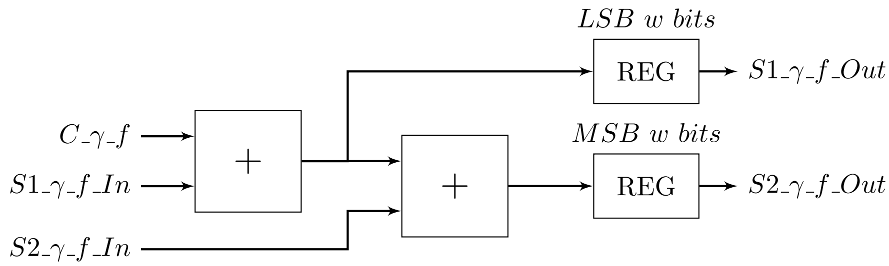

Figure 8: Gamma_f Processing Element internal architecture.

In the remainder of this section we detail our design for a $s = 8$ and a $s = 16$ architectures, respectively denoted NW-8 and NW-16.

4.4 Our architectures

Firstly, we will start with the NW-8 architecture which contains 3 PEs of type alpha and 3 of type gamma. With this design we can compute a modular multiplication in 33 clock cycles. Secondly we will present the NW-16 architecture that is composed by 6 PEs of type alpha and 6 PEs of type gamma. And we can perform a modular multiplication with this architecture in 66 clock cycles. Similarly, in order to implement the NW-32 architecture and the NW-64 architecture we need every time to double the number of cells. We provide a comparison of our architectures at the end of this section.

NW-8 Architecture In this architecture, the operands and the modulo are divided in 8 words as illustrated in Figure 10. The NW-8 architecture is composed of 9 Processing Elements distributed in a two-dimensional array. Every Processing Elements are responsible for the calculus involving w bits words of the input operands. For example, for a 256 bits modular multiplication with NW-8, the operands are split in 8 words of 32 bits which results in a two-dimensional array of 9 Processing Elements. The 9 Processing Elements are divided in the following manner: 3 cells alpha, 1 cell alpha_final, 1 cell beta, 3 cells gamma, et 1 cell gamma_final. Those choices were made in order to optimize the number of states in our FSM. As seen in section 2 each PE in the N-dimensional array is connected to 2N data In/Out paths for communicating with 2N PEs in the N-dimensional array. Since we are working with two-dimensional elements, each PE in our design is connected to 4 data paths, 2 Input and 2 Output as presented in Figure 3. In this architecture, the Processing Elements are designed with finite state machines (FSM). The control block communicates with the PEs and shift registers through starts signals. The Figure 9 presents an overview of our architecture. For more technical details the Figure 20 presents the differents PEs with input/output. The shift register is designed to provide the required words for a modular multiplication to the PEs. The Processing Element alpha requires words $a[i]$ and $b[j]$ of the operands a and b, on the other side the Processing Element gamma required a words of the operand p. Thus, these operands are defined in the package body. At the end of the Montgomery modular multiplication, the control block provides the multiplication result $a \cdot b \cdot R^{-1} \mod p$ through the outputs of the last gamma and gamma_final Processing Elements. To evaluate the number of clock cycles for a CIOS method of

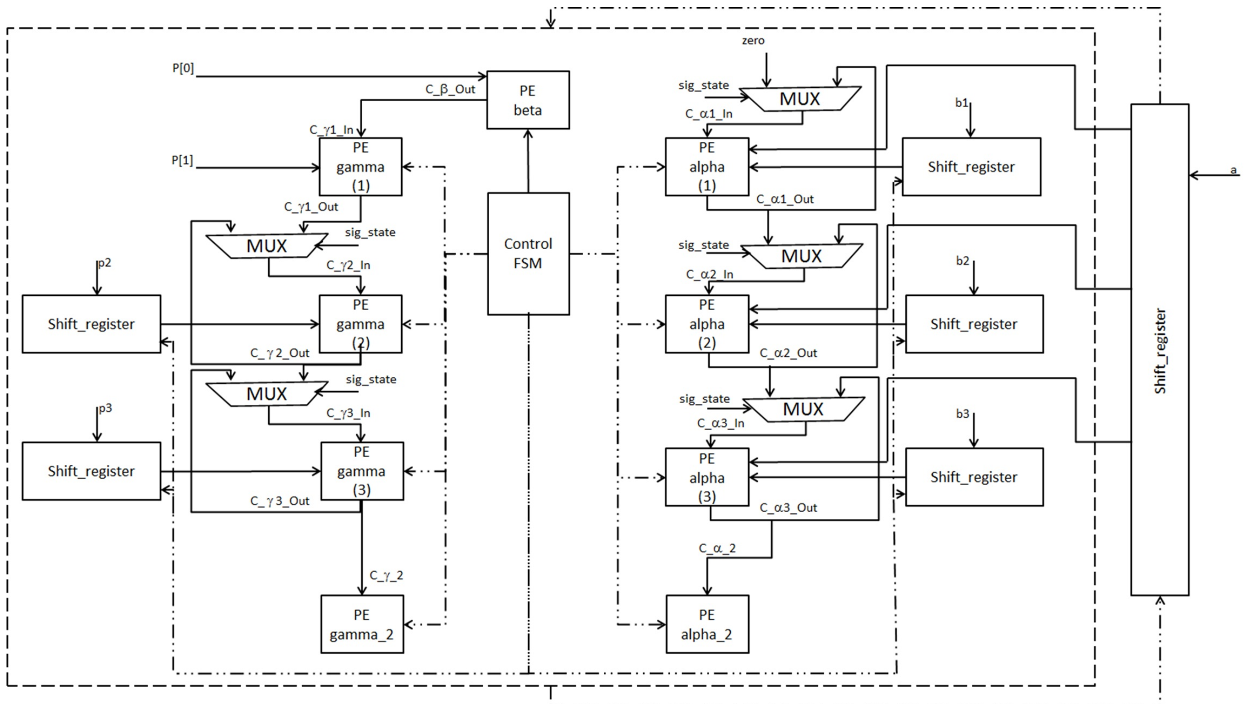

Figure 9: Proposed Montgomery modular multiplication architecture.

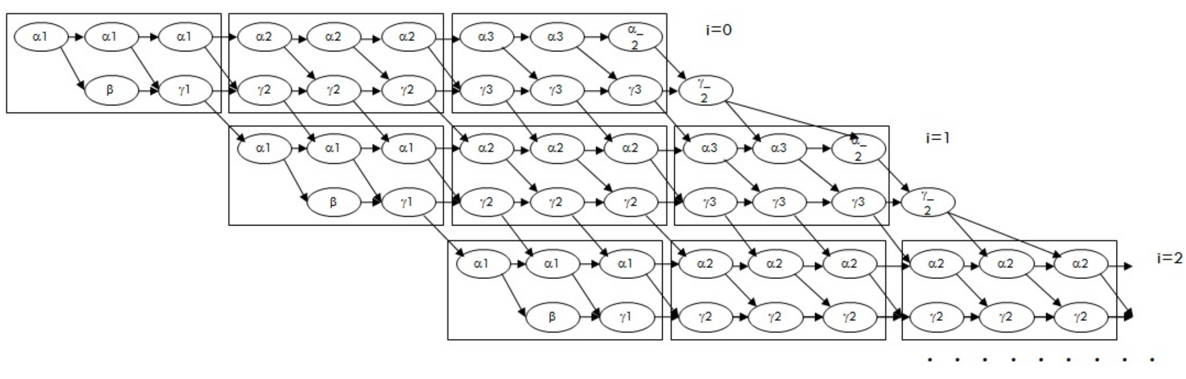

Figure 10: The data dependency graph of the proposed new Systolic Architecture with a Tow-dimensional array of Processing Elements (NW-8).

modular multiplication in NW-8, the first parameter is max{number of alpha, number of gamma}=3, it means that our design can handle three iterations of i at the same time as illustrated in Figure 10. Implying that our algorithm require to loop $s+3$ times. we can performing our design in 33 clock cycles since our design requires three states $(33 = 3 \times (s+3))$. The different results of this architecture in bit-length 256 are given in Table 2. And we illustrate an execution of this architecture in the appendix B.1

Artix-7	DSP	Frequency (MHz)	Clock cycle
MMM (s=8/K=256)	31	105.275	33
Alpha	4	291.023	1
Gamma	4	291.023	1
Beta	4	388.350	1
Alpha_final	1	459.918	1
Gamma_final	2	442.811	1

Table 2: Implementations of cells and MMM (NW-8).

NW-16 Architecture In this architecture, the operands and the modulo are divided in 16 words. The NW-16 architecture is designed in the same way as the NW-8. This example illustrates the scalability of our design. The NW-16 architecture is composed of 15 Processing Elements distributed in a two-dimensional array, where every Processing Elements are responsible for the calculus involving w bits words of the input operands. The 15 Processing Elements are divided like this: 6 cells alpha, 1 cell alpha_final, 1 cell beta, 6 cells gamma et 1 cell gamma_final. We can remark that the number of PEs of type alpha and gamma are the double of the number for NW-8. As said previously, the number of other PE type (alpha_final, beta, gamma_final) remains unchanged whatever the number of words in the design. In order to evaluate the number of clock cycles of the NW-16 architecture, the first parameter is max{number of alpha, number of gamma}=6, implying that our algorithm requires to loop $s+6$ times. We can perform the multiplication with our design in 66 clock cycles since our design requires three states $(66 = 3 \times (s+6))$. The different results of this architecture in bit-length 256 are given in Table 3.

NW-32 Architecture In this architecture, the operands and the modulo are divided in 32 words. The NW-32 architecture is composed of 27 Processing Elements distributed in a two-dimensional array, where every Processing Elements are responsible for the calculus involving w bits words of the input operands. The 27 Processing Elements are divided like this: 12 cells alpha, 1 cell alpha_final, 1 cell beta, 12 cells gamma et cell gamma_final. In order to evaluate the number of clock cycles of the NW-32 architecture, the first parameter as we have seen previously is max{number

Artix-7	DSP	Frequency (MHz)	Clock cycle
MMM (s=16/K=256)	29	145.892	66
Alpha	2	379.341	1
Gamma	2	379.341	1
Beta	2	453.104	1
Alpha_final	1	459.918	1
Gamma_final	2	442.811	1

Table 3: Implementations of cells and MMM (NW-16).

of alpha, number of gamma}=12, implying that our algorithm require to loop $s + 12$ times. We can perform the multiplication with our design in 132 clock cycles since our design requires three states ($132 = 3 \times (s + 12)$).

NW-64 Architecture In this architecture, the operands and the modulo are divided in 64 words. The NW-64 architecture is composed of 51 Processing Elements distributed in a two-dimensional array, where every Processing Elements are responsible for the calculus involving w bits words of the input operands. The 51 Processing Elements are divided like this: 24 cells alpha, 1 cell alpha_final, 1 cell beta, 24 cells gamma et 1 cell gamma_final. In order to evaluate the number of clock cycles of the NW-64 architecture, the first parameter is max{number of alpha, number of gamma}=24, implying that our algorithm require to loop $s + 24$ times. We can perform the multiplication with our design in 264 clock cycles since our design requires three states ($264 = 3 \times (s + 24)$).

Architectures comparison The Table 4 explains a comparison between the different architectures. Number of clock cycles for every architecture equal to 3 × (s+nb), such that nb=max{number of cells alpha, number of cells gamma}, implying that our algorithm require to loop $s + nb$ times. It is interesting to notice that all our architectures are scalable and targeting the different security levels useful in cryptography.

5 Results

The Table 5 summarizes the FPGA results postimplementation of the proposed versions of modular multiplication architectures. We present a results for the both architectures NW-8 and NW-16. The designs were described in hardware description languages (VHDL) and synthesized for Artix-7 and Virtex-5 Xilinx FPGAs. In order to check the correctness of the result, we compare the results given by the FPGA with the sage code. We present the different results after implementation of bit-length k which are given in Table 5. These circuits have the advantage of suitability to various applications with different bit lengths like RSA, ECC and pairings. As it is shown in Table 5, an

CIOS	s=8	s=16	s=32	s=64
K=256	32	16	8	4
K=512	64	32	16	8
K=1024	128	64	32	16
K=2048	256	128	64	32
Clock cycles= 3 × (s+nb)	33	66	132	264
Number of cells	6 +3	12 +3	24 +3	48 +3

Table 4: comparison of our architectures

interesting property of our design is the fact that the clock cycles are independent from the bit length. This property gives to our design the advantage of suitability to different security level. In order to implement the modular Montgomery multiplication for fixed security level, we must choose the most suitable architecture. The results presented in this work are compared with the previous work [18,17,5,4] in the Table 6. We could notice that our results are better then [18] considering every point of comparison i.e. the number of slice and the number of clock cycles. Considering the number of slices, we recall that [18] used an external memory to optimize the number of slices used by their algorithms. Considering the comparison with [17], our design requires less number of slices, and a better frequency and we really improve the number of clock cycles. Our design performed the Montgomery multiplication in 66 clock cycles for the 512 and 1024 bit length corresponding to AES-256 and AES-512 security level, while [17] performed the multiplication in 1540 clock cycles for the AES-256 security level and 3076 for the AES-512 security level.

6 Conclusion

In this paper we have presented an efficient hardware implementation of the CIOS method of Montgomery multiplication Algorithm over large prime characteristic finite fields $\mathbb{F}_p$. We give the results of our design after routing and placement using a Artix-7 and Virtex-5 Xilinx FPGAs. Our systolic implementations is suitable for every implementation implying a modular multiplication, for example RSA, ECC and pairing-based cryptography. Our architectures and the designs were matched with features of the FPGAs. The NW-8 design presented a good performance considering $latency \times area$ efficiency. This architecture can run for all the bit length corresponding to classical security levels (128, 256, 512 or 1024 bits) in just 33 clock cycles. On the other hand the NW-16 perform the same bit length in 66 clock cycles, but improve in area compared to NW-8 work. Our systolic design using this method CIOS is scalable for other number of words.

Artix 7- Nexys 4

	NW-8			NW-16		
	128	256	512	256	512	1024
Freq MHz	198	106	65	146	106	65
cycles	33	33	33	66	66	66
Slice Registers	487	870	1614	1123	2164	4208
Slice LUTs	355	809	2650	846	1789	5242
Slices	206	352	878	402	798	2072
DSP	19	31	87	29	57	161

Table 5: illustration of the scalabilty of our architecture.

Xilinx FPGAs

	Our works A7		**Our works V5**		[18] V5		[17] VE		[5] VII	[4] VII	[12] V		[1] K7 and V5	
	512	**1024**	**512**	**1024**	512	1024	512	1024	1024	1024	512	1024	512 K7	512 V5
Freq MHz	**106**	**65**	**97**	**65**	95	130	95.229	95.620	116.4	119	72.1	79.2	176	123
Cycles	**66**	**66**	**66**	**66**	96	384	1540	3076	1088	1167	–	–	66	66
Speed μs	**0.622**	**1.013**	**0680**	**1.015**	1.010	2.953	16.031	32.021	9.34	9.80	–	–	0.374	0536
Slice Registers	**2164**	**4208**	**3046**	**6072**	3876	6642	–	–	–	–	–	–	5076	4960
Slice LUTs	**1789**	**5242**	**1781**	**5824**	–	–	2972	5706	9319	9271	3125	6243	8757	10877
BRAM	0	0	0	0	128	256	–	–	–	–	–	–	0	0

Table 6: Comparaison of our work with state-of-art implementations.

References

[1] Karim Bigou and Arnaud Tisserand. Single base modular multiplication for efficient hardware rns implementations of ecc. page 123–140, September 2015.

[2] Junfeng Fan, K. Sakiyama, and I. Verbauwhede. Montgomery modular multiplication algorithm on multi-core systems. In *Signal Processing Systems, 2007 IEEE Workshop on*, pages 261–266, Oct 2007.

[3] Arash Hariri and Arash Reyhani-Masoleh. Bit-serial and bit-parallel montgomery multiplication and squaring over gf2m. *IEEE Transactions on Computers*, 58(10):1332–1345, 2009.

[4] D. Harris, R. Krishnamurthy, M. Anders, S. Mathew, and S. Hsu. An improved unified scalable radix-2 montgomery multiplier. In *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pages 172–178, June 2005.

[5] Miaoqing Huang, K. Gaj, and T. El-Ghazawi. New hardware architectures for montgomery modular multiplication algorithm. *Computers, IEEE Transactions on*, 60(7):923–936, July 2011.

[6] Miaoqing Huang, Kris Gaj, Soonhak Kwon, and Tarek El-Ghazawi. An optimized hardware architecture for the montgomery multiplication algorithm. In Ronald Cramer, editor, *Public Key Cryptography – PKC 2008*, volume 4939 of *Lecture Notes in Computer Science*, pages 214–228. Springer Berlin Heidelberg, 2008.

[7] Kuan i Lee. *Algorithm and VLSI architecture design for H.264/AVC Inter Frame Coding.* PhD thesis, National Cheng Kung University, Tainan, Taiwan, 2007.

[8] Keiichi Iwamura, Tsutomu Matsumoto, and Hideki Imai. High-speed implementation methods for rsa scheme. In RainerA. Rueppel, editor, *Advances in Cryptology — EUROCRYPT' 92*, volume 658 of *Lecture Notes in Computer Science*, pages 221–238. Springer Berlin Heidelberg, 1993.

[9] Keiichi Iwamura, Tsutomu Matsumoto, and Hideki Imai. Systolic-arrays for modular exponentiation using montgomery method. In RainerA. Rueppel, editor, *Advances in Cryptology — EUROCRYPT' 92*, volume 658 of *Lecture Notes in Computer Science*, pages 477–481. Springer Berlin Heidelberg, 1993.

[10] Antoine Joux. A one round protocol for tripartite diffie–hellman. *Journal of Cryptology*, 17(4):263–276, 2004.

[11] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[12] Saadat Pourmozafari Kooroush Manochehri and Babak Sadeghiyan. Montgomery and rns for rsa hardware implementation. page 29(5):849–880, December 2010.

[13] C.K. Koç, Tolga Acar, and Jr. Kaliski, B.S. Analyzing and comparing montgomery multiplication algorithms. *Micro, IEEE*, 16(3):26–33, Jun 1996.

[14] H.T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, Jan 1982.

[15] VictorS. Miller. Use of elliptic curves in cryptography. In HughC. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer Berlin Heidelberg, 1986.

[16] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[17] S.B. Ors, L. Batina, B. Preneel, and J. Vandewalle. Hardware implementation of a montgomery modular multiplier in a systolic array. pages 8 pp.–, April 2003.

[18] Guilherme Perin, Daniel Gomes Mesquita, and João Baptista Martins. Montgomery modular multiplication on reconfigurable hardware: Systolic versus multiplexed implementation. *Int. J. Reconfig. Comput.*, 2011:6:1–6:10, January 2011.

[19] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[20] Alexandre F. Tenca and Çetin Kaya Koç. A scalable architecture for montgomery multiplication. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 1999.

[21] Mahendra Vucha and Arvind Rajawat. Design and fpga implementation of systolic array architecture for matrix multiplication. volume 26, page 18–22, July 2011. Full text available.

A Appendix

A.1 Code Sage NW-8

```
#NW-8 Algoritm
s=8
p'
p=[p0,p1,p2,p3,p4,p5,p6,p7]
b=[b0,b1,b2,b3,b4,b5,b6,b7]
a=[a0,a1,a2,a3,a4,a5,a6,a7]
T=[0,0,0,0,0,0,0,0,0,0]
for i in range (s):
    C_S=0
    for j in range (0,s):
        C_S=T[j]+a[i]*b[j]+(C_S>>32)
        T[j]=C_S%(2^32)
    C_S=T[s]+(C_S>>32)
    T[s]=C_S%(2^32)
    T[s+1]=C_S>>32
    m=(T[0]*p')%(2^32)
    C_S=T[0]+m*p0
    for j in range (1,s):
        C_S=T[j]+m*p[j]+(C_S>>32)
        T[j-1]=C_S%(2^32)
    C_S=T[s]+(C_S>>32)
    T[s-1]=C_S%(2^32)
    T[s]=T[s+1]+(C_S>>32)
```

A.2 Code Sage NW-16

```
#NW-16 Algoritm
s=16
p'
p=[p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15]
b=[b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15]
a=[a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15]
T=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
for i in range (s):
    C_S=0
    for j in range (0,s):
        C_S=T[j]+a[i]*b[j]+(C_S>>16)
        T[j]=C_S%(2^16)
    C_S=T[s]+(C_S>>16)
    T[s]=C_S%(2^16)
    T[s+1]=C_S>>16
    m=(T[0]*p')%(2^16)
    C_S=T[0]+m*p0
    for j in range (1,s):
        C_S=T[j]+m*p[j]+(C_S>>16)
        T[j-1]=C_S%(2^16)
    C_S=T[s]+(C_S>>16)
    T[s-1]=C_S%(2^32)
    T[s]=T[s+1]+(C_S>>16)
```

B architecture

B.1 Execution

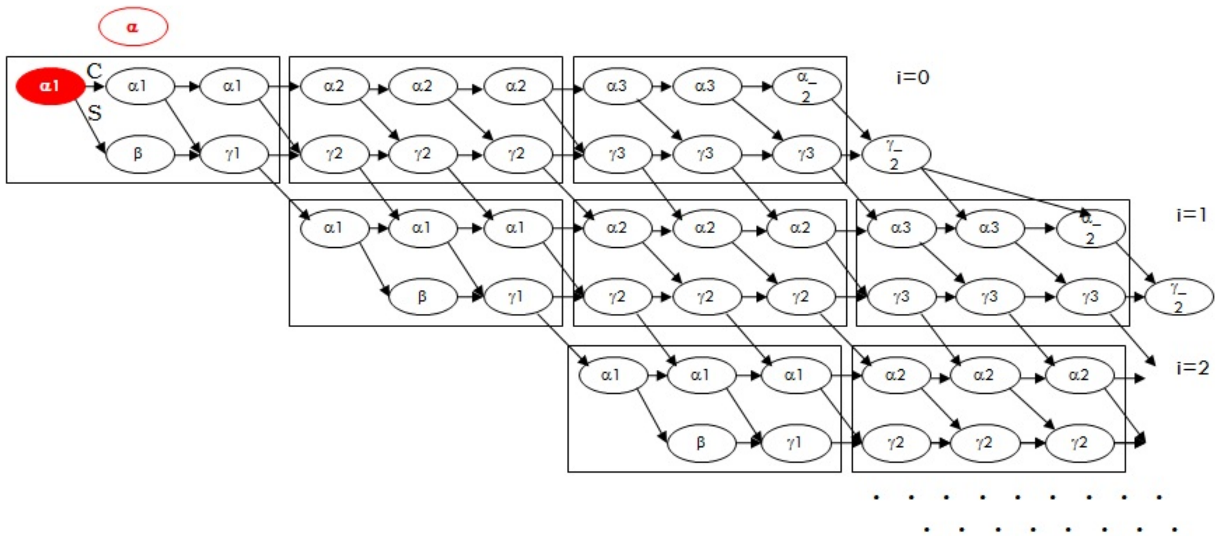

Figure 11: Step 1.

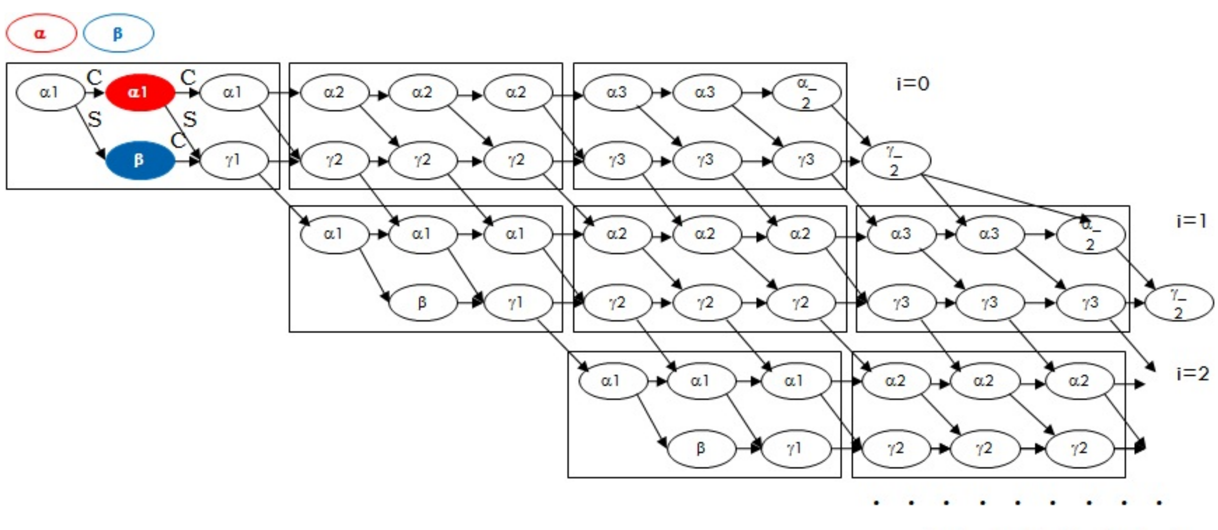

Figure 12: Step 2.

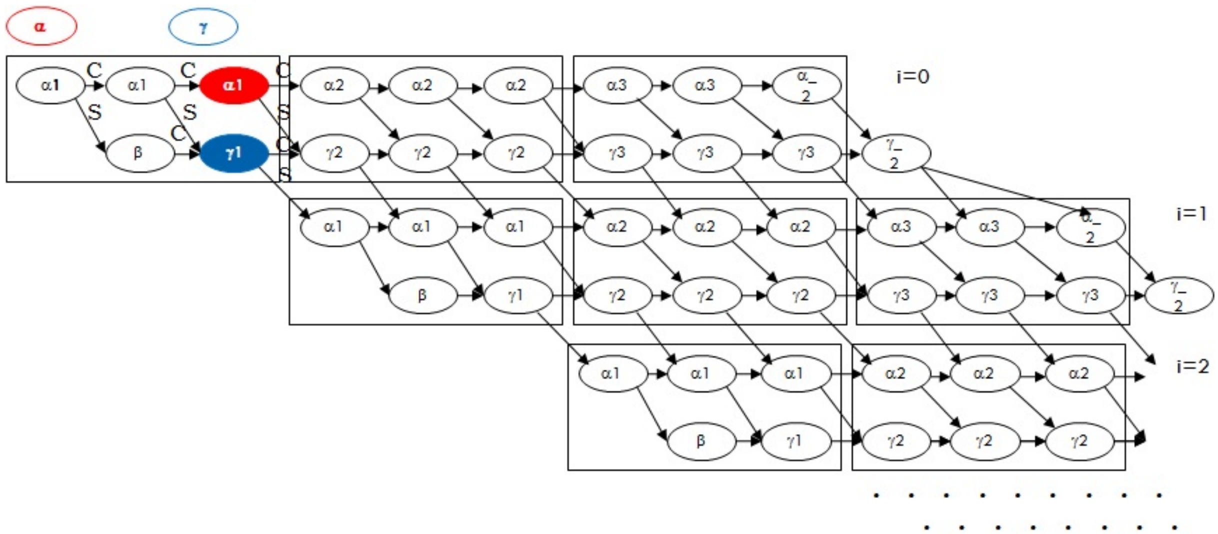

Figure 13: Step 3.

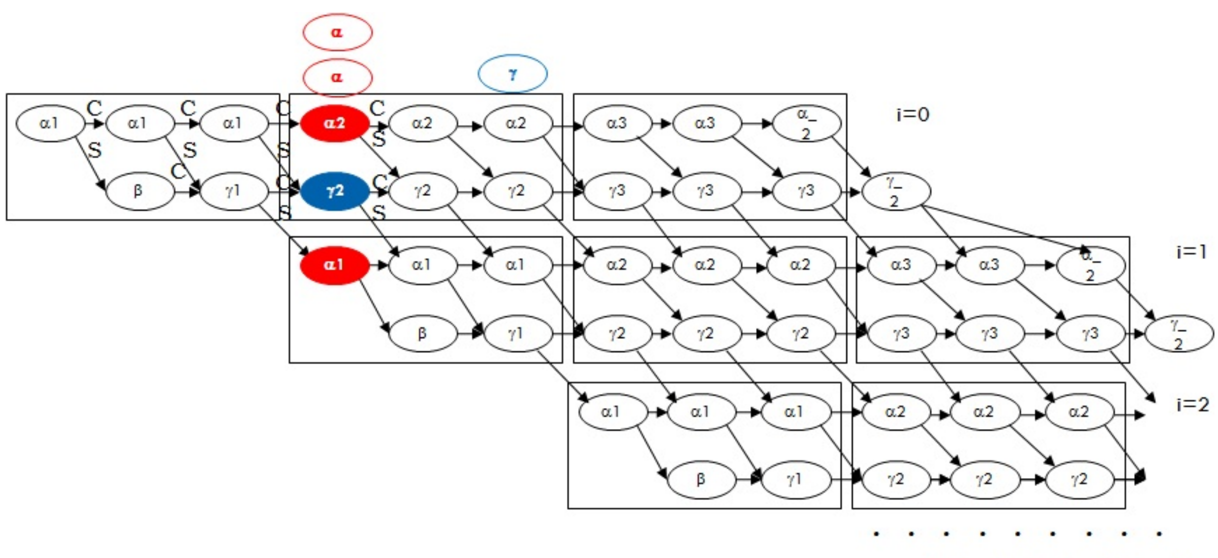

Figure 14: Step 4.

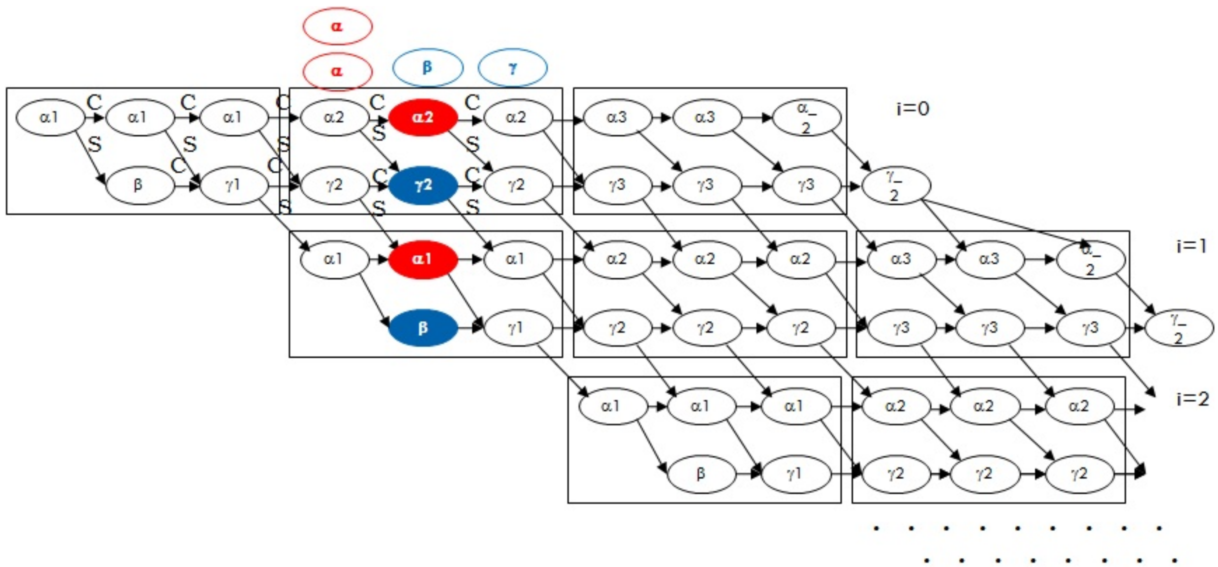

Figure 15: Step 5.

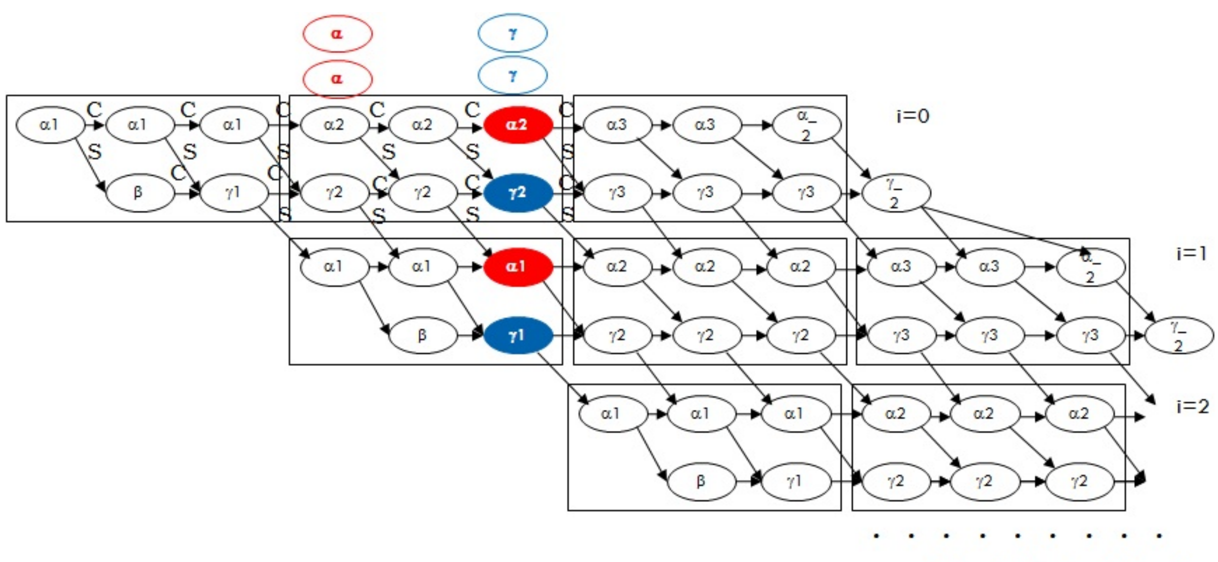

Figure 16: Step 6.

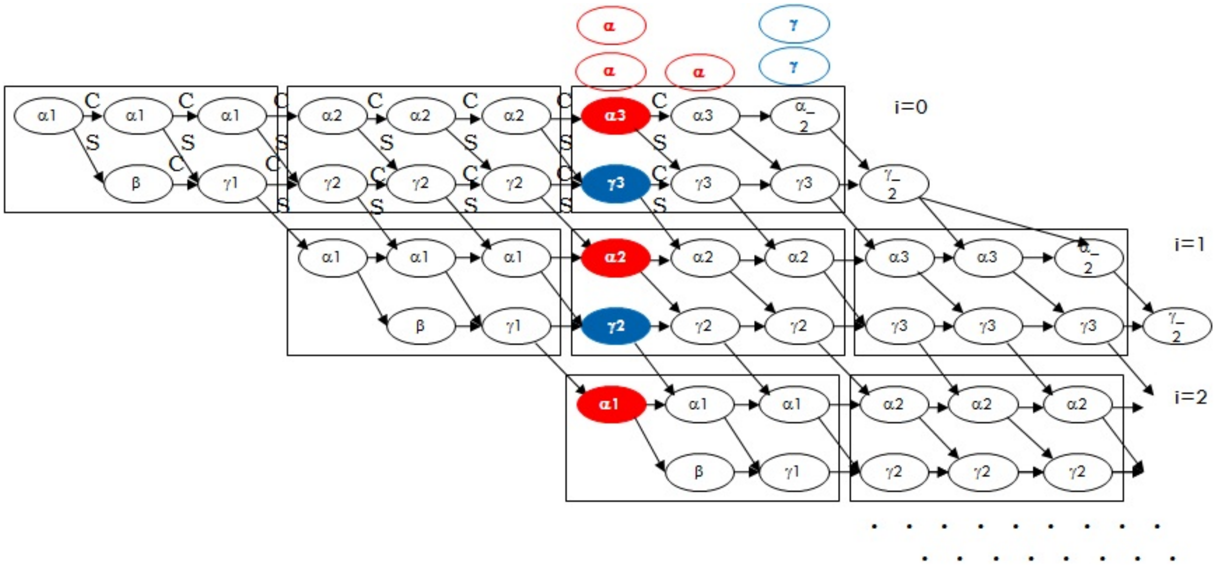

Figure 17: Step 7.

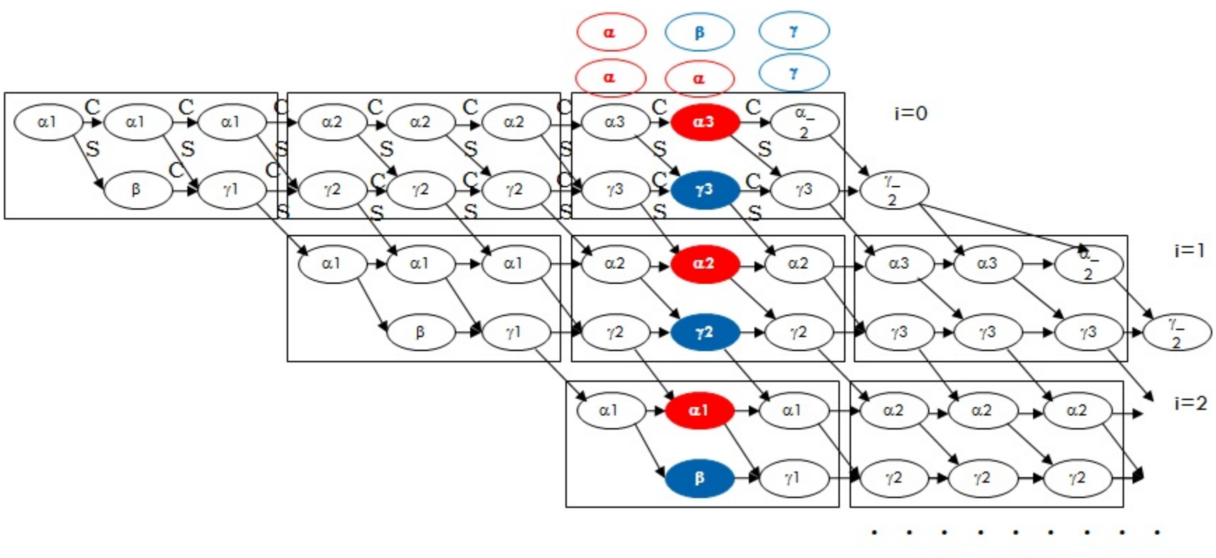

Figure 18: Step 8.

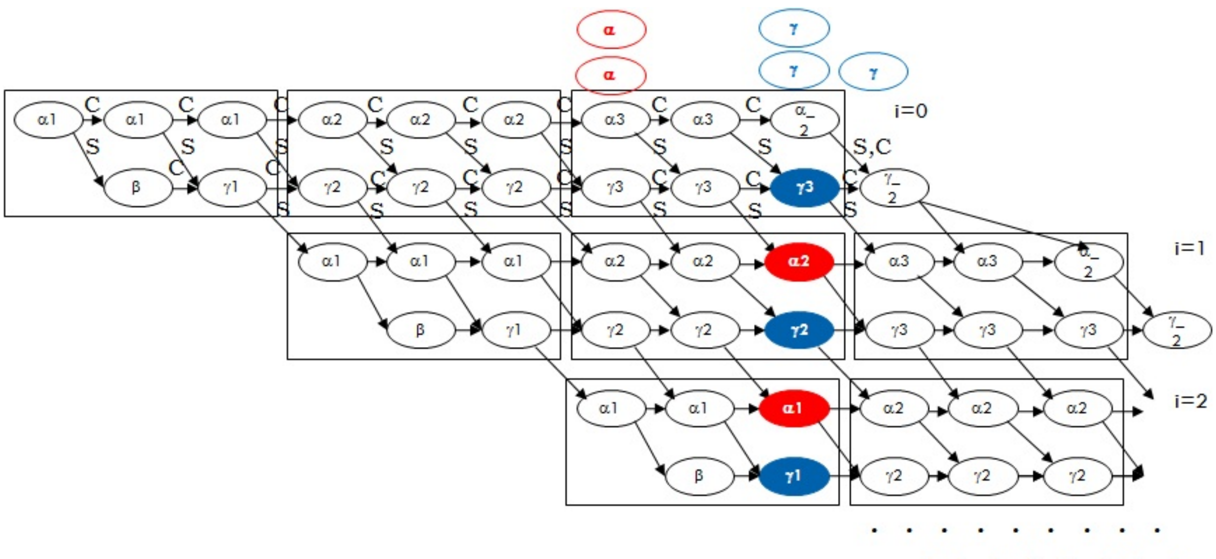

Figure 19: Step 9.

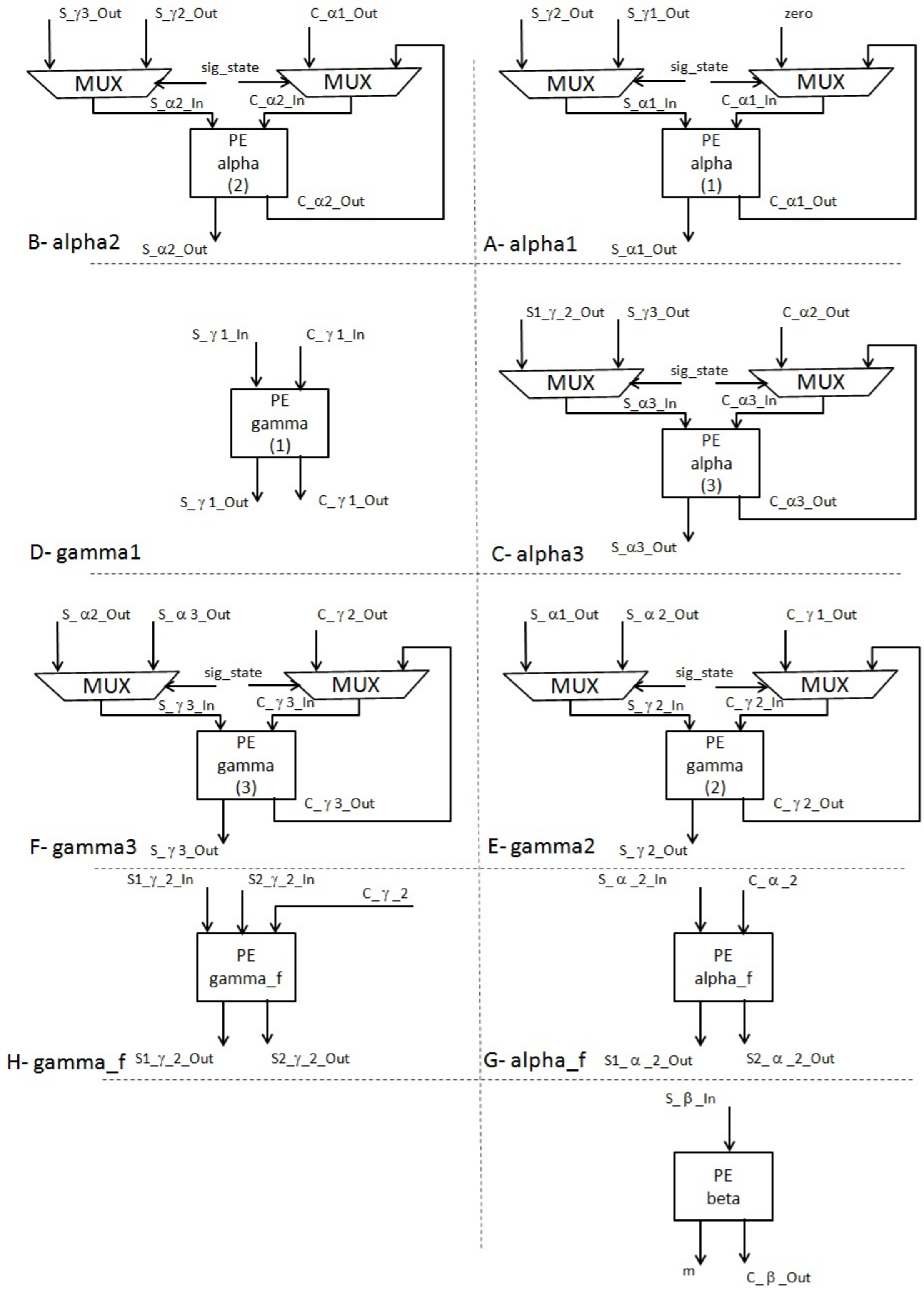

Figure 20: All Processing Elements.