1999-09-03

Modular Exponentiation on Reconfigurable Hardware

Thomas Blum
Worcester Polytechnic Institute

Modular Exponentiation on Reconfigurable Hardware

by
Thomas Blum

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical Engineering

April 8th, 1999

Approved:

Prof. Christof Paar
ECE Department
Thesis Advisor

Prof. Yusuf Leblebici
ECE Department
Thesis Committee

Prof. Fred J. Looft
ECE Department
Thesis Committee

Prof. John Orr
ECE Department Head

Abstract

It is widely recognized that security issues will play a crucial role in the majority of future computer and communication systems. A central tool for achieving system security are cryptographic algorithms. For performance as well as for physical security reasons, it is often advantageous to realize cryptographic algorithms in hardware. In order to overcome the well-known drawback of reduced flexibility that is associated with traditional ASIC solutions, this contribution proposes arithmetic architectures which are optimized for modern field programmable gate arrays (FPGAs). The proposed architectures perform modular exponentiation with very long integers. This operation is at the heart of many practical public-key algorithms such as RSA and discrete logarithm schemes. We combine two versions of Montgomery modular multiplication algorithm with new systolic array designs which are well suited for FPGA realizations. The first one is based on a radix of two and is capable of processing a variable number of bits per array cell leading to a low cost design. The second design uses a radix of sixteen, resulting in a speed–up of a factor three at the cost of more used resources. The designs are flexible, allowing any choice of operand and modulus.

Unlike previous approaches, we systematically implement and compare several versions of our new architecture for different bit lengths. We provide absolute area and timing measures for each architecture on Xilinx XC4000 series FPGAs. As a first practical result we show that it is possible to implement modular exponentiation at secure bit lengths on a single commercially available FPGA. Secondly we present faster processing times than previously reported. The Diffie–Hellman key exchange scheme with a modulus of 1024 bits and an exponent of 160 bits is computed in 1.9 ms. Our fastest design computes a 1024 bit RSA decryption in 3.1 ms when the Chinese remainder theorem is applied. These times are more than ten times faster than any reported software implementation. They also outperform most of the hardware–implementations presented in technical literature.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

It is widely recognized that security issues will play a crucial role in many future computer and communication systems. A central tool for achieving system security is cryptography. For performance as well as for physical security reasons it is often required to realize cryptographic algorithms in hardware. Traditional ASIC solutions, however, have the well-known drawback of reduced flexibility compared to software solutions. Since modern security protocols are increasingly defined to be *algorithm independent*, a high degree of flexibility with respect to the cryptographic algorithms is desirable. A promising solution which combines high flexibility with the speed and physical security of traditional hardware is the implementation of cryptographic algorithms on reconfigurable devices such as FPGAs and EPLDs. In the case of public-key schemes, algorithm independence can mean not only a change of the actual cryptographic algorithm but also a change of parameters such as bit length, modulus, or exponents. One application, dealt with in this report, includes arithmetic architectures for modular exponentiation with very long integers which is at the heart of most modern public-key schemes. Most notably, both RSA and discrete logarithm-based (e.g., Diffie-Hellman key exchange or the Digital Signature Algorithm, DSA) schemes require modular long number exponentiation.

The challenge at hand is to design arithmetic architectures for operands with up to 1024 bits on current FPGAs. The very long word lengths prohibit the application of many proposed architectures as they would result in unrealistically large resource requirements. In this thesis we derive two modular exponentiation architectures which combine Montgomery's modular reduction scheme and novel systolic array architectures. The systolic array architecture requires considerably fewer logic resources than many other systolic array architectures for modular arithmetic. This is crucial, as one of our goals was to derive solutions that can fit into a single FPGA, a design goal that has many cost and design advantages over multi–FPGA solutions. Another important objective was to systematically implement various architecture options for different bit lengths and compare performance and resource usage.

1.2 Thesis Goals

Based on the general considerations in the previous section we defined the following goals for the thesis research:

1. Implementation of a 1024–bit modular exponentiation architecture in a single commercially available FPGA device. 1024 bits is the recommended bit size for RSA and discrete logarithm systems and thus highly relevant for practical applications. The computation time of our architecture should be close to a previously reported architecture that used 16 FPGAs [33].

2. It should be investigated which of the two proposed general architecture options, based on systolic arrays and a redundant representation, is best suited for modular exponentiation architectures on FPGAs.

3. To find an optimal resource usage and computation time trade–off for the FPGA architectures that will be designed.

4. A resource efficient FPGA design should be developed which allows the implementation of a 1024 bit architecture at moderate costs.

5. It should be investigated weather the high–radix Montgomery modular multiplication algorithm proposed in [9] can be used for modular exponentiation architectures on FPGAs.

6. Develop and implement a design that is considerably faster than any previously reported FPGA architecture and reaches speeds similar to the fastest design reported in technical literature [27].

1.3 Thesis Outline

This thesis is structured as follows. In Chapter 2, we summarize some of the previous work on modular exponentiation. Chapter 3 describes three families of algorithms in public–key cryptography, and the modular arithmetic needed for their implementation. Chapter 4 describes algorithms for modular exponentiation and multiplication and some simplifications and speed-ups for their hardware implementation. In Section 5 we summarize some of the relevant features of the Xilinx XC4000 FPGA series. Based on these features we derive some characteristics for our architectures. Chapter 6 outlines our architecture for modular exponentiation, optimized for low resource usage. Chapter 7 describes an architecture optimized for speed. Chapter 8 describes our methodology and tools that were used for this research. Chapter 9 posts the timing and area results obtained. A comparison to other architectures and an outlook conclude this thesis.

Chapter 2

Previous Work

In the following, we will summarize relevant previous work in the field of modular multiplication. Most presented approaches are based on an algorithm proposed by Peter Montgomery in 1985 [19], either in conjunction with a redundant number representation or in a systolic array architecture. Solutions using other algorithms have also been presented.

2.1 Montgomery Reduction and Redundant Representation

Applying Montgomery's algorithm, the cost of a modular exponentiation is reduced to a series of additions of very long integers. To avoid the carry propagation in multiplication/addition architectures several solutions have been proposed in the literature. They either use Montgomery's algorithm, in combination with a redundant radix number system [26, 33, 7, 9, 36] or a Residue Number System [2].

In [7] Montgomery's modular multiplication algorithm is adapted for an efficient hardware implementation. A gain in speed results from a faster clock, due to simpler combinatorial logic. Compared to previous techniques based on Brickell's Algorithm [4], a speed-up factor of two is reported.

The Research Laboratory of Digital Equipment Corp. in Paris implemented modular exponentiation architectures on FPGAs [33, 26]. They utilized an array of 16 XILINX 3090 FPGAs. Compared to XILINX 4000 series in terms of flip–flops, this is equivalent to a chip with 5100 configurable logic blocks (CLBs). In terms of logic resources this is equivalent to a chip of 4000 CLBs. In their work they used several speed-up methods [26] including the Chinese remainder theorem, asynchronous carry completion adder, and a windowing exponentiation method. The implementation computes a 970bit RSA decryption at a rate of 185kb/s (5.2ms per 970 bit decryption) and a 512 bit RSA decryption in excess of 300 kb/s (1.7ms per 512 bit decryption). A drawback of this solution is that the binary representation of the modulus is hardwired into the logic representation so that the architecture has to be reconfigured with every new modulus.

The problem of using high radices in Montgomery's modular multiplication algorithm is the more complex determination of the quotient. This behavior made a pipelined execution of the algorithm impossible. Reference [9] rewrites the algorithm and avoids thereby any operation involved in the quotient determination. The necessary pre–computation has to be done only once for a given modulus.

Reference [36] proposes a novel VLSI architecture for Montgomery's modular multiplication algorithm. The critical path that determines the clock speed is pipelined. This is done by interleaving each iteration of the algorithm. Compared to previous propositions, an improvement of the time–area product of a factor two is reported.

Reference [2] describes a new approach using a Residue Number System (RNS). The algorithm is implemented with n moduli in the RNS on n reasonably simple processors. The resulting processing time is $O(n)$.

2.2 Montgomery Reduction and Systolic Arrays

There have been a number of proposals for systolic array architectures for modular arithmetic. However, no implementations have been reported to our knowledge. In [8] a VLSI solution is presented where a modular multiplication is calculated in $(4m + 1) \cdot 3m/2$ clock cycles, where m is the number of bits of the modulus. That is approximately four times more cycles than in a conventional solution. In terms of resources, this design would be suitable for FPGA.

Similar two-dimensional systolic arrays are presented in [10, 35, 36]. For a radix of two they all propose an $m \times m$ matrix of one bit processing elements. With this configuration $2m$ modular multiplications are calculated at the same time and the theoretical throughput is one modular multiplication per clock cycle. In terms of resources, such a solution is not feasible in either VLSI or FPGA for the bit length required in public-key algorithms. Even implementing only one row of processing elements, (resulting in m times slower throughput) into presently available FPGAs is difficult in terms of resources.

In reference [30] a linear systolic array was obtained by systematically mapping a two-dimensional graph model onto a one-dimensional systolic array.

Reference [15] describes an architecture based on one row of processing elements and a radix of two. Squarings and multiplications are computed in parallel. The system requires n systolic processing elements for an n–bit modular exponentiation, and the resulting execution time is $2n^2$ clock cycles.

2.3 Other Work

References [4, 25, 32, 31] describe different algorithms for modular multiplication avoiding costly division. Reference [3] compares these algorithms. An overview of previously presented architectures for VLSI implementations and their underlying algorithms for modular integer arithmetic is also provided in this contribution. Reference [5] summarizes the chips available in 1990 for performing RSA encryption.

In Reference [34] a generalization of [4] is presented and some conclusions are drawn about the choice of the radix.

Reference [29] proposes a radix–4 hardware algorithm. A redundant number representation is used and the propagation of carries in additions is therefore avoided. A processing speed–up of about six times compared to previous work is reported.

More recently an approach [39] has been presented that utilizes pre-computed complements of the modulus and is based on the iterative Horner's rule. Compared to Montgomery's algorithms these approaches use the most significant bits of an intermediate result to decide which multiples of the modulus to subtract. The drawback of these solutions is that they either need a large amount of storage space or many clock cycles to complete a modular multiplication. The authors attempted to overcome the later problem by a higher clock frequency which is possible due to a simplified modulo reduction operation.

2.4 Implementations

To our knowledge, the fastest reported software implementation of modular exponentiation [37] computes RSA decryption with a 1024–bit modulus in 43 ms.

In Reference [23] a table with several VLSI hardware implementations for RSA is published. The fastest chip computes RSA decryption with a 512–bit modulus in 8 ms. These chips are somewhat dated, though. More recently an ASIC implementation has been reported [16] that computes RSA decryption with a 1024–bit modulus in 150 ms. However, the author claims in [27] that 1024–bit exponentiation architectures with 10 ms computation time are available. This time corresponds to an RSA computation time of 2.5 ms if the Chinese remainder theorem is used for speeding–

up the computation. Only one FPGA implementation of RSA has been reported in technical literature so far [33]. 970-bit RSA decryption is computed in 5.2 ms in this approach. A detailed comparison of the modular exponentiation architectures that we develop in this thesis with the previously reported implementations will be given in Chapter 9.

Chapter 3

Preliminaries: Public–Key Algorithms

In this chapter we review the three most popular families of public key algorithms. Information on secure key length is given as well as speed-up methods proposed in the literature. We will show that all algorithm families are based on modular long number arithmetic.

3.1 RSA

RSA was proposed by Rivest, Shamir and Adleman [21] in 1978. The private key of a user consists of two large primes p and q and an exponent D. The public key consists of the modulus $M = p \times q$, $M = \sum_{i=0}^{m-1} m_i 2^i$, $m_i \in \{0, 1\}$ and an exponent E such that $E = D^{-1} \bmod (p-1)(q-1)$, $E = \sum_{i=0}^{n-1} e_i 2^i$, $e_i \in \{0, 1\}$. In the remainder of this thesis we assume that E can be represented by n bits, and M can be represented by m digits. To encrypt a message X the user computes:

$$Y = X^E \bmod M$$

Decryption is done by calculating:

$$X = Y^D \bmod M$$

The identical operations are used for the RSA digital signature scheme. In order to thwart currently known attacks, the modulus M and thus X and Y should have a length of 768 – 1024 bits. Both encryption and decryption require algorithms for computing a modular exponentiation.

For speeding up encryption the use of a short exponent E has been proposed [13]. Recommended by the International Telecommunications Union ITU is the the Fermat prime $F_4 = 2^{16}+1$. Using F_4, the encryption is executed in only 17 operations. Other short exponents proposed include $E = 3$ and $E = 17$.

Obviously the same trick can not be used for decryption, as the decryption exponent D must be kept secret. But using the knowledge of the factors of $M = q \times p$, the Chinese Remainder Theorem [20] can be applied by the decrypting party. Two $m/2$ size modular exponentiations and an additional recombination instead of one m size modular exponentiation are computed in this case. Each modular exponentiation of length $m/2$ takes 1/4 of the time required for an m – bit exponentiation (see Chapter 4). If both exponentiations are performed serially, an over–all speed–up factor of two is achieved. If they are performed in parallel, a speed–up factor of four is achieved.

3.2 Algorithms Based on the Discrete Logarithm Problem in Finite Fields

The best known public–key schemes based on the discrete logarithm problem in finite fields are the Diffie-Hellman key exchange scheme, the Digital Signature Algorithm (DSA) and the ElGamal encryption scheme (see, e.g., [17]). As an example, we present below the Diffie-Hellman key exchange scheme, proposed in 1976 by W. Diffie and M.E. Hellman [6].

The goal of this protocol is to establish a secret session key between to parties over an insecure channel. The two parties, Alice and Bob, want to establish a secret key without Oscar, the adversary, being able to compute this key. During the setup phase Alice and Bob obtain the public parameters p and α. Parameter p is a large prime and α a primitive element in Z_p^* or a subgroup of Z_p^*.

The algorithm proceeds as follows:

1a) Alice generates a random key:		1b) Bob generates a random key:
$a_A \in \{2, 3, \ldots p-1\}$ (private)		$a_B \in \{2, 3, \ldots p-1\}$ (private)
2a) Alice computes her public key:		2b) Bob computes his public key:
$\beta_A = \alpha^{a_A} \bmod p$ (public)		$\beta_B = \alpha^{a_B} \bmod p$ (public)
3a) Alice sends β_A to Bob	$\xrightarrow{\beta_A}$	
	$\xleftarrow{\beta_B}$	3b) Bob send β_B to Alice
4a) Alice computes:		4b) Bob computes:
$K_s = \beta_B^{a_A} = (\alpha^{a_B})^{a_A} = \alpha^{a_B \cdot a_A} \bmod p$		$K_s = (\alpha^{a_A})^{a_B} = \alpha^{a_A \cdot a_B} \bmod p$

After the final stage of the algorithm, Alice and Bob share a session key K_s. Oscar cannot regenerate the session key from the public parameters α, β_A, and β_B because the two random integers a_A and a_B, generated by Alice and Bob are private and were never transmitted over the insecure channel.

The computational complexity of the algorithm lies in steps 2 and 4, the computation of a modular exponentiation. The index–calculus method is the currently best known attack against discrete logarithm-based schemes. In order to thwart this attack, the modulus p and thus α should have a length of 768–1024 bits, and even longer bit lengths are recommended for highly sensitive applications. If α generates a subgroup of order n, the exponents a_A, a_B can be restricted to $0 < a_A, a_B < n$. In practice, a 160 bit exponent can be used with moduli up to 1024 bit.

3.3 Elliptic Curves

Elliptic Curve public–key cryptosystems were proposed independently in 1986/1987 by Victor Miller [18] and Neil Koblitz [14]. We restrict ourselves in the following to curves over prime fields, as opposed to curves over extension fields such as $GF(2^m)$. An elliptic curve is a set of all pairs (x,y), $x, y \in Z_p$, that fulfill the equation:

$$y^2 \equiv x^3 + ax + b \bmod p$$

To perform an addition of two points $P_1 = (x_1,y_1)$, $P_2 = (x_2,y_2)$, $P_3 = P_1 + P_2 = (x_3,y_3)$ we need to compute the following equations:

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda \cdot (x_1 - x_3) - y_1 \end{aligned}$$

$$\lambda = \begin{cases} \frac{y_2-y_1}{x_2-x_1} \bmod p, & \text{if } P_1 \neq P_2 \text{ (addition)} \\ \frac{3x_1^2+a}{2y_1} \bmod p, & \text{if } P_1 = P_2 \text{ (doubling)} \end{cases}$$

The complexity of this operation is two multiplications and one inversion (point–addition) or three multiplications and one inversion (point–doubling), if we ignore additions and subtractions. The inversion is very costly to implement. To optimize the addition of two points by avoiding the inversion, the use of projective coordinates has been proposed. A projective point (X,Y,Z) in the projective plane can be identified with a point (x,y) in the affine plane. The homogeneous elliptic curve is a set of all points (X,Y,Z) that fulfill the equation:

$$ZY^2 \equiv X^3 + aXZ^2 + bZ^3 \bmod p$$

The addition formulae are now [12]:

addition:
$$X_3 = VA$$
$$Y_3 = U(V^2X_1Z_2 - A) - V^3Y_1Z_2$$
$$Z_3 = V^3Z_1Z_2$$

where $U = Y_2Z_1 - Y_1Z_2$, $V = X_2Z_1 - X_1Z_2$, $A = U^2Z_1Z_2 - V^2T$, $T = X_2Z_1 + X_1Z_2$

doubling:
$$X_3 = 2SH$$
$$Y_3 = W(4F - H) - 8E^2$$
$$Z_3 = 8S^3$$

where $S = Y_1Z_1$, $W = 3X_1^2 + aZ_1^2$, $E = Y_1S$, $F = X_1E$, $H = W^2 - 8F$

To perform an elliptic curve projective space addition we have to compute 15 multiplications, and 12 multiplications are needed for a doubling operation.

A method similar to Algorithm 4.1 combines additons and doublings to a general point multiplication, $e \cdot P$, that is, addition of the point P e–times to itself. Point multiplication is the core operation in elliptic curve public key crypto–systems. If we use a modulus and operands of length $m+1$ bits, m doublings and an average of $m/2$ additions have to be executed.

The currently best known attack against elliptic curve public key crypto–systems uses the Silver–Pohlig–Hellmann algorithm [24] together with Pollard's rho method. In order to thwart this attack, the modulus p and thus X, Y, and Z should have a length of at least 160 bits. We note that this operand bit length is considerably shorter than in the case of RSA or DL schemes.

Chapter 4

Preliminaries: Modular Exponentiation

In this chapter we review the square & multiply algorithm, which is the most popular algorithm for modular exponentiation. Secondly we develop versions of Montgomery's modular multiplication algorithm, which are well suited for hardware implementations.

4.1 Square & Multiply Algorithm

The public–key schemes described in Chapter 3 are based on modular exponentiation or repeated point addition. Both operations are in their most basic forms done by the square and multiply algorithm [13].

Algorithm 4.1 *compute* $Z = X^E \bmod M$*, where* $E = \sum_{i=0}^{n-1} e_i 2^i$, $e_i \in \{0,1\}$

```
1.  $Z = X$
2.  FOR $i = n-2$ down to 0 DO
3.  $Z = Z^2 \bmod M$
4.  IF $e_i = 1$ THEN $Z = Z \cdot X \bmod M$
```

```
5.  END FOR
```

Algorithm 4.1 takes $2(n-1)$ operations in the worst case and $1.5(n-1)$ on average. To compute a squaring and a multiplication in parallel we can use the following version of the square & multiply algorithm [36]:

Algorithm 4.2 *computes* $P = X^E \bmod M$*, where* $E = \sum_{i=0}^{n-1} e_i 2^i$*,* $e_i \in \{0, 1\}$

```
1.  $P_0 = 1$, $Z_0 = X$
2.  FOR $i = 0$ to $n-1$ DO
3.  $Z_{i+1} = Z_i^2 \bmod M$
4.  IF $e_i = 1$ THEN $P_{i+1} = P_i \cdot Z_i \bmod M$
    ELSE              $P_{i+1} = P_i$
5.  END FOR
```

Algorithm 4.2 takes $2n$ operations in the worst case and $1.5n$ on average. A speed–up can be achieved by applying the l – ary method [13] which is a generalization of Algorithm 4.1. The l – ary method processes l exponent bits at the time. The drawback here is that $(2^l - 2)$ multiples of X have to be precomputed and stored. A reduction to 2^{l-1} pre–computations is possible. The resulting complexity is roughly n/l multiplications and n squaring operations.

4.2 Montgomery Reduction

As shown in the previous section, modular exponentiation is reduced to a series of modular multiplications and squaring steps. The algorithm for modular multiplication described below has been proposed by P. L. Montgomery in 1985 [19]. It is a method for multiplying two integers modulo M, while avoiding division by M. The idea is to transform the integers in m-residues and compute the multiplication with these m-residues. Finally we transform back to the normal representation. This approach is only beneficial if we compute a series of multiplications in the transform domain (e.g., modular exponentiation).

To compute the Montgomery multiplication, we chose a radix $R > M$, with $\gcd(M, R) = 1$. Division by R has to be inexpensive, thus an optimal choice is $R = 2^m$ if we assume that $M = \sum_{i=0}^{m-1} m_i 2^i$. The m-residue of x is $xR \bmod M$. We also compute $M' = -M^{-1} \bmod R$. Now we define a function MRED(T) that computes $TR^{-1} \bmod M$: This function computes the normal representation of T, given T is an m-residue.

Algorithm 4.3 *MRED(T): computes a Montgomery reduction of T*
$T < RM$, $R = 2^m$, $M = \sum_{i=0}^{m-1} m_i 2^i$, $\gcd(M, R) = 1$

```
1.  $U = TM' \bmod R$
2.  $t = (T + UM)/R$
3.  IF      $t \geq M$   RETURN  $t - M$
    ELSE                 RETURN  $t$
```

The result of MRED(T) is $t = TR^{-1} \bmod M$. For the proof of this equation, see [19].

Now we consider a multiplication of two integers a and b in the transform domain, where their respective representations are $(aR \bmod M)$ and $(bR \bmod M)$. To acquire the result $(abR \bmod M)$ we feed their product into MRED(T):

$$MRED((aR \bmod M) \cdot (bR \bmod M)) = abR^2R^{-1} = abR \bmod M$$

For a modular exponentiation we can repeat this step numerous times according to Algorithm 4.1 or 4.2 to get the final result $ZR \bmod M$ (Algorithm 4.1) or $P_nR \bmod M$ (Algorithm 4.2). We finally feed one of these values into MRED(T) to get the result $Z \bmod M$ or $P_n \bmod M$.

The initial transform step still requires costly modular reductions. To avoid the division involved, we can take the following approach. First we compute $R^2 \bmod M$ using division. This step needs to be done only once for a given cryptosystem. To get a and b in the transform domain we run MRED($a \cdot R^2 \bmod M$) and MRED($b \cdot R^2 \bmod$

M) to get $aR \bmod M$ and $bR \bmod M$. Obviously, any variable can be transformed in this manner.

We now consider a hardware implementation of Algorithm 4.3: To compute step 2 we need an $m \times m$–bit multiplication and a $2m$–bit addition. The intermediate result can have as many as $2m$ bits. Instead of computing U at once, we can compute one digit of an r–radix representation at a time. We have to chose a radix r, such that $\gcd(M, r) = 1$ [28]. Division by r has to be inexpensive, thus an optimal choice is $r = 2^k$. All variables are now represented in a basis–r representation. Another improvement is to include the multiplication $A \times B$ in the algorithm.

Algorithm 4.4 *[7] Montgomery Modular Multiplication for computing* $A \cdot B \bmod M$, *where* $M = \sum_{i=0}^{m-1} (2^k)^i m_i,\ m_i \in \{0, 1 \ldots 2^k - 1\}$;
$B = \sum_{i=0}^{m-1} (2^k)^i b_i,\ b_i \in \{0, 1 \ldots 2^k - 1\}$;
$A = \sum_{i=0}^{m-1} (2^k)^i a_i,\ a_i \in \{0, 1 \ldots 2^k - 1\}$;
$A, B < M$; $M < R = 2^{km}$; $M' = -M^{-1} \bmod 2^k$; $\gcd(2^k, M) = 1$

```
1.  $S_0 = 0$
2.  FOR $i = 0$ to $m - 1$ DO
3.  $q_i = (((S_i + a_i B) \bmod 2^k) M') \bmod 2^k$
4.  $S_{i+1} = (S_i + q_i M + a_i B)/2^k$
5.  END FOR
6.  IF      $S_m \geq M$ RETURN $S_m - M$
    ELSE              RETURN $S_m$
```

The output of Algorithm 4.4 is $S_m = ABR^{-1} \bmod M$. Considering a radix $r = 2^k$, we need at most two $k \times k$– bit multiplications and a k–bit addition to compute step 3. For step 4 two $k \times m$– bit multiplications and two $m + k$–bit additions are needed. The maximal bit length of S is reduced to $m + k + 2$ bits, compared to the $2m$ bits of Algorithm 4.3. In Section 4.3 we review further improvements of Algorithm 4.4 for the case of $r = 2$. Section 4.4 treats the algorithm for larger radix.

4.3 Montgomery Multiplication for Radix Two

Algorithm 4.5 is a simplification of Algorithm 4.4 for radix $r = 2$. For the radix $r = 2$, the operations in step 3 of Algorithm 4.4 are done modulo 2. The modulus M must be odd due to the condition $\gcd(M, 2^k) = 1$. It follows immediately that $M \equiv 1 \bmod 2$. Hence $M' \equiv -M^{-1} \bmod 2$ also degenerates to $M' = 1$. Thus the multiplication by $M' \bmod 2$ in step 3 can be omitted.

Algorithm 4.5 *[7] Montgomery Modular Multiplication (Radix $r = 2$) for computing* $A \cdot B \bmod M$, *where* $M = \sum_{i=0}^{m-1} 2^i m_i$, $m_i \in \{0, 1\}$;
$B = \sum_{i=0}^{m-1} 2^i b_i$, $b_i \in \{0, 1\}$;
$A = \sum_{i=0}^{m-1} 2^i a_i$, $a_i \in \{0, 1\}$;
$A, B < M$; $M < R = 2^m$; $\gcd(2, M) = 1$

```
1.  $S_0 = 0$
2.  FOR $i = 0$ to $m - 1$ DO
3.  $q_i = (S_i + a_i B) \bmod 2$
4.  $S_{i+1} = (S_i + q_i M + a_i B)/2$
5.  END FOR
6.  IF      $S_m \geq M$ RETURN $S_m - M$
    ELSE                 RETURN $S_m$
```

The final comparison and subtraction in step 6 of Algorithm 4.5 would be costly to implement, as an m bit comparision is very slow or expensive in terms of resource usage . It would also make a pipelined execution of the algorithm impossible. It can easily be verified that $S_{i+1} < 2M$ always holds if A, $B < M$. S_m, however, can not be reused as input A or B for the next modular multiplication. If we perform two more executions of the for loop with $a_{m+1} = 0$ and inputs A, $B < 2M$, the inequality $S_{m+2} < 2M$ is satisfied. Now, S_{m+2} can be used as input B for the next modular multiplication. We just allow S to have two more bits for intermediate results.

To further reduce the complexity of Algorithm 4.5, B can be shifted up by one position, i.e., multiplied by two [7]. This results in $a_i \cdot B \bmod 2 = 0$ and the addition in step 3 is avoided. In the update of S_{i+1} we replace $(S_i + q_iM + a_iB)/2$ by $(S_i + q_iM)/2 + a_iB$. The cost of this simplification is one more execution of the loop with $a_{m+2} = 0$. The algorithm below comprises the just mentioned optimizations.

Algorithm 4.6 *[7] MONT_R2(A,B): Montgomery Modular Multiplication (Radix $r = 2$) for computing $A \cdot B \bmod M$, where $M = \sum_{i=0}^{m-1} 2^i m_i$, $m_i \in \{0, 1\}$;*
$B = \sum_{i=0}^{m} 2^i b_i$, $b_i \in \{0, 1\}$;
$A = \sum_{i=0}^{m+2} 2^i a_i$, $a_i \in \{0, 1\}$, $a_{m+1} = 0$, $a_{m+2} = 0$;
$A,B < 2M$, $M < R = 2^{m+2}$; $\gcd(2, M) = 1$

```
1.  $S_0 = 0$
2.  FOR $i = 0$ to $m + 2$ DO
3.  $q_i = S_i \bmod 2$
4.  $S_{i+1} = (S_i + q_i \cdot M)/2 + a_i \cdot B$
5.  END FOR
```

The algorithm above calculates $S_{m+3} = (2^{-(m+2)}AB) \bmod M$. To get the correct result we need an extra Montgomery modular multiplication by $2^{2(m+2)} \bmod M$. However, if further multiplications are required as in exponentiation algorithms, it is better to pre–multiply all inputs by the factor $2^{2(m+2)} \bmod M$. Thus every intermediate result carries a factor 2^{m+2}. We just need to Montgomery multiply the result by "1" to eliminate that factor.

The final Montgomery multiplication with "1" insures that our final result is smaller than M. Consider Algorithm 4.6 with $B < 2M$ and $A = (0, \ldots, 0, 1)$. We will get $S_1 = a_0 \cdot B = B < 2M$. As all remaining $a_i = 0$, we get at most $S_{i+1} = (S_i + M)/2 \to M$. If only one $q_i = 0$ $(i = 1, 2 \ldots m + 2)$, then $S_{i+1} = S_i/2 < M$ (probability: $1 - 2^{-(m+2)}$).

The computational complexity of Algorithm 4.6 lies in the two additions of m bit operands for computing S_{i+1}. Recall that $m \approx 160 - 1024$ is of great interest in public–key algorithms. As the propagation of m carries is too slow and an equivalent carry look ahead logic requires to many resources, two different strategies have been pursued in literature:

1. Redundant representation: The intermediate results are kept in redundant form. Resolution into binary representation is only done at the very end and for feeding the intermediate result back as a_i in Algorithm 4.6.

2. Systolic Arrays: Typically m processing units calculate 1 bit per clock cycle. The computed carries, q_i and a_i are "pumped" through the processing units. As these signals have to be distributed only between adjacent processing units, a faster clock speed and a resulting higher throughput should be possible. The cost is a higher latency and possibly more resources.

4.4 High–Radix Montgomery Algorithm

The goal of this section is to improve Algorithm 4.4 to make it suitable for a hardware implementation. At first we avoid the costly comparison and subtraction of step 6. The output S_m has to be small enough to be fed back in the algorithm as A or B. We change the conditions to $4M < 2^{km}$ and $A, B < 2M$. This results in $S_m < 2M$ as needed for further processing. The penalty is two more executions of the loop (see also Section 4.3 for $k = 1$).

In Section 4.3 the multiplication in the quotient q_i determination of Algorithm 4.4 was avoided. This is also possible for higher radixes [9]. M has to be transformed to $\tilde{M} = (M' \bmod 2^k)M$. This step must be performed only once for a given crypto–system. The conditions for the algorithm are $4\tilde{M} < 2^{km}$ and $A, B < 2\tilde{M}$ now. Thus for the same bit length of M the loop has to be executed one more time. An other penalty is a larger range of S_m. Algorithm 4.7 comprises the above mentioned improvements.

Algorithm 4.7 *[9] Montgomery Modular Multiplication for computing* $A \cdot B \bmod M$, *where* $M = \sum_{i=0}^{m-3}(2^k)^i m_i,\ m_i \in \{0, 1 \ldots 2^k - 1\}$;
$\tilde{M} = (M' \bmod 2^k)M,\ \tilde{M} = \sum_{i=0}^{m-2}(2^k)^i \tilde{m}_i,\ \tilde{m}_i \in \{0, 1 \ldots 2^k - 1\}$;
$B = \sum_{i=0}^{m-1}(2^k)^i b_i,\ b_i \in \{0, 1 \ldots 2^k - 1\}$;
$A = \sum_{i=0}^{m-1}(2^k)^i a_i,\ a_i \in \{0, 1 \ldots 2^k - 1\}$;
$A, B < 2\tilde{M}$; $4\tilde{M} < 2^{km}$; $M' = -M^{-1} \bmod 2^k$

```
1. $S_0 = 0$
2. FOR $i = 0$ to $m - 1$ DO
3. $q_i = (S_i + a_i B) \bmod 2^k$
4. $S_{i+1} = (S_i + q_i \tilde{M} + a_i B)/2^k$
5. END FOR
```

The quotient q_i determination complexity can further be reduced by replacing B by $B \cdot 2^k$. Since $a_i B \bmod 2^k \equiv 0$, step 3 is reduced to $q_i = S_i \bmod 2^k$. The addition in step 3 is avoided at the cost of an additional iteration of the loop, to compensate for the extra factor 2^k in B. A Montgomery algorithm optimized for hardware implementation is shown below:

Algorithm 4.8 *[9] MONT_RH(A,B):Montgomery Modular Multiplication for computing* $A \cdot B \bmod M$, *where* $M = \sum_{i=0}^{m-3}(2^k)^i m_i,\ m_i \in \{0, 1 \ldots 2^k - 1\}$;
$\tilde{M} = (M' \bmod 2^k)M,\ \tilde{M} = \sum_{i=0}^{m-2}(2^k)^i \tilde{m}_i,\ \tilde{m}_i \in \{0, 1 \ldots 2^k - 1\}$;
$B = \sum_{i=0}^{m-1}(2^k)^i b_i,\ b_i \in \{0, 1 \ldots 2^k - 1\}$;
$A = \sum_{i=0}^{m}(2^k)^i a_i,\ a_i \in \{0, 1 \ldots 2^k - 1\},\ a_m = 0$;
$A, B < 2\tilde{M}$; $4\tilde{M} < 2^{km}$; $M' = -M^{-1} \bmod 2^k$

```
1. $S_0 = 0$
2. FOR $i = 0$ to $m$ DO
3. $q_i = (S_i) \bmod 2^k$
4. $S_{i+1} = (S_i + q_i \tilde{M})/2^k + a_i B$
5. END FOR
```

The result of Algorithm 4.8 is $S_{m+1} = ABR^{-1} \bmod M$ where $R = 2^{km} \bmod M$. We perform the same pre–computations as mentioned in Section 4.3: Pre–multiply all inputs by the factor $2^{2km} \bmod M$. Thus every intermediate result carries a factor 2^{km}. We just need to Montgomery multiply the final result by 1 to eliminate that factor.

The final Montgomery multiplication with 1 makes sure our final result is smaller than $\tilde{M}$. Consider Algorithm 4.8 with $B < 2\tilde{M}$ and $A = (0, \ldots, 0, 1)$. We will obtain $S_1 = a_0 \cdot B = B < 2\tilde{M}$. As all remaining $a_i = 0$, we get at most $S_{i+1} = (S_i + (2^k - 1)\tilde{M})/2^k \rightarrow \tilde{M}$. If only one $q_i \neq 2^k - 1$, $(i = 1, 2 \ldots m)$, $q_i \in \{0, 1 \ldots 2^k - 1\}$, then $S_{i+1} < \tilde{M}$. $\tilde{M}$, however, can be $2^k - 1$ times larger than M and the same is true for the result of a modular exponentiation S_{m+1}. The last quotient q_m and the factor $M' \bmod 2^k$ might determine the number of times M has to be subtracted from S_{m+1}. This behavior, however, has not been studied in this thesis. We assume that the final comparison and eventual modular reduction step is performed outside of our design.

Algorithm 4.8 is used for the architecture described in Chapter 7. The designs we implemented have moduli of 160, 256, 512, 768, and 1024 bits. The radix chosen was $r = 2^4 = 16$. For the rest of this thesis we use the following convention for specifying the complexity: $M = \sum_{i=0}^{m-1} (2^k)^i m_i \Rightarrow \tilde{M} = \sum_{i=0}^{m} (2^k)^i \tilde{m}_i$, $B = \sum_{i=0}^{m+1} (2^k)^i b_i$, $A = \sum_{i=0}^{m+2} (2^k)^i a_i$, $a_{m+2} = 0$. The loop in Algorithm 4.8 is executed $m + 3$ times and the response is $S_{m+3} = ABR^{-1} \bmod M$, where $R = 2^{k(m+2)} \bmod M$. Thus the pre–computation factor is $2^{2k(m+2)} \bmod M$.

The computational complexity of Algorithm 4.8 lies in the two additions of $m + k$ bit operands for computing S_{i+1}. Another costly operation is the computation of the multiples of M and B in step 4.

Chapter 5

General Design Considerations

In this chapter we present some of the relevant features of the Xilinx XC4000 Series FPGAs and introduce a metric for FPGA cost and performance evaluation. Based on these features we derive some characteristics for our architectures. The pros and cons of two different approaches to implement Montgomery's algorithm are exhibited.

5.1 Xilinx XC4000 Series FPGAs

5.1.1 Configurable Logic Blocks

An FPGA device consists of three types of reconfigurable elements, the Configurable Logic Blocks (CLBs), I/O blocks (IOBs) and routing resources [38].

Figure 5.1.1 shows the structure of a CLB. An XC4000 CLB is made up of three look–up tables (LUT) F, G, and H, two flip-flops and programmable multiplexers. Any boolean function of 5 inputs, any two functions of 4 inputs and some functions of up to 9 inputs can be computed in one CLB. The multiplexers can route the outputs of the look–up tables directly to the outputs or to the flip-flops. In the first case the flip-flops can be utilized to store direct inputs. This is important for a design with a large amount of registers. The same CLB can be used to store two bits and compute two independent logic functions. Thus a considerable amount of resources can be

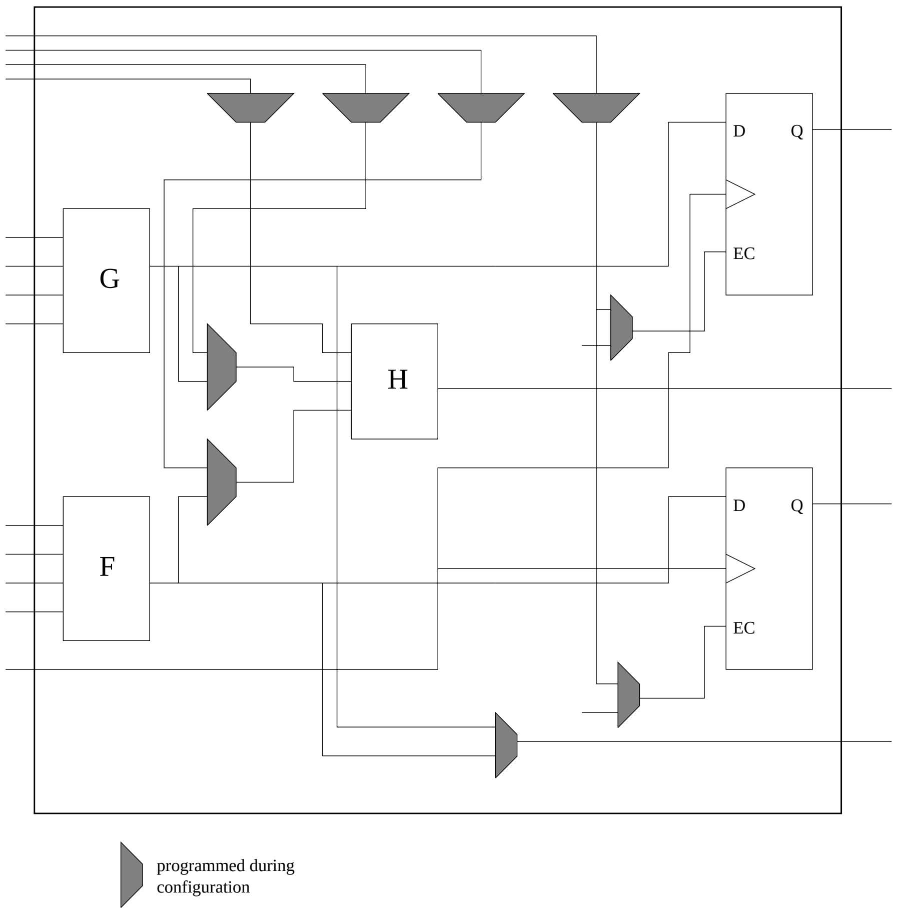


Figure 5.1: XC4000 CLB Structure [38]

saved as will be shown in Sections 6.2 and 7.2.

5.1.2 Routing Topologies

Programmable routing resources connect the CLBs and IOBs into a network. The structure of XC4000 FPGA devices is shown in Figure 5.2. A matrix of switch boxes is placed over the CLB array. These switch boxes make it possible to connect any two CLBs together.

Routing in the XILINX FPGA is accomplished through a hierarchal structure.

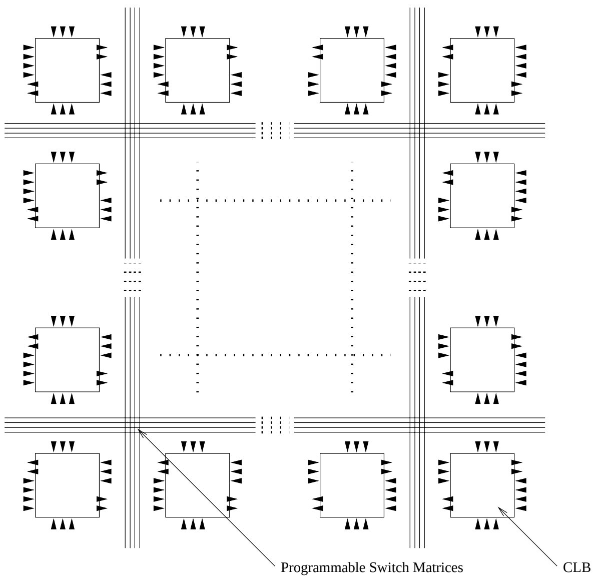


Figure 5.2: Xilinx FPGA structure [38]

Each row or column of routing lines between CLBs has a number of different types of lines. These include single, double, quad, long, and global lines. Single lines route signals between adjacent CLBs. Double lines stretch over two CLBs. For the architectures described in Chapters 6 and 7, devices of the XC4000XL and XC4000XV families were used. Table 5.1 shows their routing resources per CLB:

The numbers show that a large amount of connections are available for closely located CLBs. On the other hand routing gets increasingly difficult for CLBs further apart. In Section 5.2.1 we will discuss some consequences this issue causes for the choice of our architectures. For a detailed description of the routing structure inside the XILINX FPGA, refer to [38].

Table 5.1: Routing per CLB in XC4000XL and XC4000XV devices [38]

	Vertical	Horizontal
Singles	8	8
Doubles	4	4
Quads	12	12
Longlines	10	6
Direct Connects	2	2
Globals	8	0
Carry Logic	1	0

5.1.3 Special Features of the XC4000 Family

The XC4000 devices contain dedicated, hard–wired carry logic to both accelerate and condense arithmetic functions such as adders and counters [38]. An n bit ripple carry adder is implemented in $n/2+2$ CLBs. The ripple carry outputs are routed between CLBs on high speed dedicated paths. The maximum delay from the operand input to the sum output of a N–bit adder is approximately:

$$t_{pd} = 4.5 + N \cdot 0.35 \ [ns]$$

For an n–bit counter, the minimum clock period is approximately:

$$t_{clk-clk} = 9.5 + N \cdot 0.35 \ [ns]$$

These values vary slightly for different devices and speed grades.

Another very useful feature of the XC4000 devices is the possibility to implement RAM in CLBs. A single CLB can be programmed as a 16×2 bit or 32×1 bit ROM/RAM or as a 16×1 bit Dual Port RAM. RAM with larger address width requires considerably more resources. Table 5.2 shows the amount of CLBs used for implementing 64×2–bit, 128×2–bit, and 256×2–bit RAM and DP RAM blocks.

Table 5.2: Amount of CLBs used for RAM and DP RAM blocks on XC4000 devices

	64×2–bits CLBs	128×2–bits CLBs	256×2–bits CLBs
RAM	6	12	24
DP RAM	16	32	64

5.1.4 Cost and Speed Evaluation

In previous work [36, 35, 7] related to modular arithmetic architectures, the gate count model has been used for cost evaluation and the gate delay model for speed evaluation. This is not appropriate for FPGAs. As the functional unit of an FPGA is the CLB, we evaluate the cost (C) in number of CLBs. The operation time (T) consists of logic delay in the CLBs and routing delay and is obtained from Xilinx's Timing Analyzer software. As a third parameter we use the time–area product (TA). It is defined by time multiplied by cost.

5.2 Architectures Suitable for FPGAs

5.2.1 Systolic Array vs. Redundant Representation

As described in Section 4.3, there have been two principle approaches proposed to compute Montgomery modular multiplication.

1. Avoid the carry propagation delays by keeping intermediate results in redundant representation. Resolution into binary representation is only done at the very end and for feeding the intermediate result back as a_i in Algorithm 4.8.

2. Systolic Arrays: Processing units compute successive values for a single digit position. The computed carries, q_i and a_i, are "pumped" through the processing units.

A solution following approach 1 has already been implemented in FPGAs [33]. A matrix of 16 FPGAs has been used. The second approach using systolic arrays has drawn considerable attention in the research community. However, no architectures that specifically target FPGAs have been reported, nor are there reports of ASIC implementations of such systolic architectures.

The question at hand is which solution is better suited for an FPGA implementation. To answer this we first take a closer look at the equation we have to implement (Section 4.4):

$$\begin{aligned} q_i &= S_i \bmod 2^k \\ S_{i+1} &= (S_i + q_i \cdot \tilde{M})/2^k + a_i \cdot B \end{aligned} \tag{5.1}$$

Equation (5.1) is executed iteratively to compute a modular multiplication. A series of modular multiplications combine to a modular exponentiation according to the square & multiply Algorithms 4.1 or 4.2. Additionally a pre–computation and post–computation are necessary (Section 4.3 and 4.4).

An architecture comprising these computations has two major parts:

1. The arithmetic part computes Equation (5.1). The operands B, $\tilde{M}$ and S_i must be stored and multiples of $\tilde{M}$ and B have to be calculated. Furthermore two additions have to be computed. To avoid a long carry chain, we can divide the adder into units that typically compute one digit of S_{i+1} in Equation (5.1). If the units compute one iteration of (5.1) at the same time, the result S_{i+1} must be kept in redundant representation. A separate adder is needed to resolve the redundancy of S_{i+1} for further processing. In a systolic array approach the units compute an iteration of 5.1 successively. The overflow of a unit is fed as carry to the next unit. In both approaches we have to globally feed q_i and a_i to all units. The quotient q_i is computed once per iteration in the least significant unit. The result of the modular multiplication S_{m+3} must be stored and reused

as operand A for the successive squaring. Therefore all digits of S_{m+3} need to be distributed to all units of the design.

2. A control and storage part contains the finite state machine (FSM) that controls the execution of the square & multiply algorithm and the pre– and post–computations. We also need storage elements for the pre–computation factor, the exponent and operand A. A is processed serially and has to be distributed digit by digit globally over the arithmetic part.

Clearly, the arithmetic part will utilize most of the resources. Considerations concerning its design will decide between an systolic array and a redundant representation architecture. Some considerations concerning the control part are discussed in Subsection 5.2.2.

Two recent MS theses [22] and [11] in the same field found that a major problem when implementing designs in FPGAs, is the availability of enough routing resources. As discussed in Section 5.1.2, the problem worsens if there are many connections between CLBs which are far apart. When considering an implementation of an *architecture* with *redundant representation* there are some routing issues to deal with:

To compute Equation (5.1) with a radix $r = 2^k$, q_i and a_i, each k bits wide and some control signals have to be distributed globally over the arithmetic part. As all redundant digits of S_{i+1} are computed concurrently, the signals have to arrive at their destination at the same time. The resulting high fan–out causes also considerable propagation delays. Another issue is feeding B and $\tilde{M}$ to the locations where they are stored and processed. To avoid the full size bus that feeds these signals in parallel, we need a systolic approach at least for the loading of B and $\tilde{M}$. Lastly, the redundant result S_{m+1} of a modular multiplication has to be resolved and distributed as A for further processing. As the low order bits of A are needed first, the resolution of the most significant bits is not critical in terms of propagation delay. The routing, however, is an issue as the full length result has to be stored and distributed all over the arithmetic part.

Routing in a *systolic array architecture* is much less critical:

In a systolic array architecture we have typically m units, each processing k bits. The k signals q_i and a_i and the control signals are "pumped" through the units, from register to register. k bits of S_{i+1} are computed per clock cycle. Only k bits of the result S_{m+1} of a modular multiplication are valid at the time. Thus with an additional k bit register per unit we can "pump" the result back through the units. We can further store the result in RAM, k bits at the time, for further processing. So far no connections stretch over more than one unit. If reasonably small units are designed routing is not problematic. B and $\tilde{M}$ can be loaded over k bit wide buses. The load signal propagates through the units and activates the clock enable of one unit per cycle. Thus only $2k$ signals have to be routed all over the design.

Summarizing the last two paragraphs, three major advantages of a systolic array architecture were found.

routing resources Most connections are within one unit or are stretching to an adjacent unit. The availability of enough routing resources is much less an issue compared to an architecture where signals have to be distributed all over the design.

propagation delay A higher clock frequency is possible due to less additional routing delays of long paths.

synchronous design A fully synchronous design is possible, as we do not need an asynchronous carry completion adder as described in [33], to resolve the redundant representation of the result S.

In contrast to these three advantage, we possibly need more resources to implement a systolic array. A considerable amount of registers is needed for "pumping" the operand A, the quotient Q, the control word and the result S consecutively through the units. CLBs used for implementing registers, however, can be reused for logic functions as pointed out in Section 5.1.1.

5.2.2 State Machine and Storage Elements

We chose a synchronous methodology for all designs presented in this thesis. In the synchronous approach, the clock period is determined by the longest combinatorial delay between two registers.

One of the major goals of this work was to design a high–speed modular exponentiation architecture. This can be done by reducing the amount of iterations in Algorithm 4.8 by choosing a high radix. Secondly the propagation delay of Step 4 in the same algorithm needs to be optimized. This step is typically computed in one clock cycle and determines the minimum clock period.

The FSM and central storage elements have to be designed in such a way that their minimum clock period is smaller than the combinatorial delay of the arithmetic part. To speed–up the FSM as much as possible "one–hot encoding" was used. Each state is represented with an individual bit, resulting in decreased logic complexity associated with each state [1]. On the other hand more registers are needed. We accept this minor disadvantage as the FSM uses only a small percentage of the whole design.

In the central storage elements we need to store data serially. Such data includes the pre–computation factor, the exponent, and intermediate results of the square & multiply algorithm. The usage of RAM saves a large amount of resources compared to registers. A 1024 bit exponent stored in a 16×64-bit RAM uses 32 CLBs, while an equivalent register needs 512 CLBs. RAM with address width larger than 4 bit require more resources and feature larger access delays, due to additional address decoding. The RAM have to be designed in such a way that their access time does not determine the minimal clock period of the design.

Chapter 6

Design 1: A Resource Efficient Architecture

In this chapter we describe our first architecture. The goal was to design an area efficent architecture using Algorithm 4.6. As target devices we use the Xilinx XC4000 family as described in Chapter 5. The results of the actual implementation of the architecture will be described in Chapter 9.

6.1 Design Overview

A general radix 2 systolic array as proposed in [10, 35, 8] utilizes m times m processing elements, where m is the number of bits of the modulus and each element processes a single bit. $2m$ modular multiplications can be processed simultaneously, featuring a throughput of one modular multiplication per clock cycle and a latency of $2m$ cycles. As this approach would result in unrealistically large CLB counts for the bit length required in modern public–key schemes, we implemented only one row of processing elements. With this approach two modular multiplications can be processed simultaneously and the performance reduces to a throughput of two modular multiplications per $2m$ cycles. The latency remains $2m$ cycles.

The second consideration was the choice of the radix $r = 2^k$. Increasing k reduces the amount of steps to be executed in Algorithm 4.8. Such an approach, however, requires more resources: The main expense lies in the computation of the 2^k multiples of M and B in Algorithm 4.8. They can either be pre-computed and stored in RAM or calculated by a multiplexer network as proposed in Reference [9]. Clearly, the CLB count becomes smallest for $r = 2$, as no multiples of M or B have to be calculated or pre–computed.

Using a radix $r = 2$, the following equation has to be implemented (Algorithm 4.6):

$$S_{i+1} = (S_i + q_i \cdot M)/2 + a_i \cdot B,\ q_i, a_i \in \{0, 1\}$$

To further reduce the required number of CLBs we took the following measures:

1. A considerable amount of CLBs are used in the overhead of a processing element (unit). At least a_i, q_i and two control bits have to be stored and decoded in each unit. If only one bit is processed per unit, the overhead is required m times. In order to save resources we implemented units that process u = 4,8,16 bits. With this approach we need only m/u instead of m units, and a considerable amount of overhead can be saved. For processing more than one bit per unit large adders are needed. Thus we expect the processing time and resource requirements to increase exponentially. The possibility to use the fast carry ripple adder as described in Section 5.1.3, however, causes processing time and CLB count to grow only proportionally.

2. Computing u bits per unit, the operands S_i, B and M have all u bits in each unit. The result S_{i+1} has a maximum of $u + 2$ bits, u result bits and two carry bits that are fed to the next unit. Instead of using two adders and computing the two additions serially in each execution of the loop, we pre-compute $B + M$ and store the result in a register. The resulting carry is fed to the next unit. Thus we need only one adder, that adds S_i to 0, B, M or $B + M$. This adder

is also used for computing $B + M$. With this approach the result S_{i+1} is only $u + 1$ bits wide and just one carry bit is fed to the next unit.

Similar to the approach in [15] we compute squarings and multiplications in parallel. As explained in Section 6.3, this measure fully utilizes every cycle.

Design 1 can be divided hierarchically into three levels.

Processing Element Computes u bits of a modular multiplication.

Modular Multiplication An array of processing elements computes a modular multiplication.

Modular Exponentiation Combine modular multiplications to a modular exponentiation according to Algorithm 4.2.

In the following we describe the system with a bottom–up approach.

6.2 Processing Elements

Figure 6.1 shows the implementation of a processing element.

In the processing elements we need the following registers:

- *M-Reg* (u bits): storage of the modulus
- *B-Reg* (u bits): storage of the B multiplier
- *B+M-Reg* (u bits): storage of the intermediate result $B + M$
- *S-Reg* ($u + 1$ bits): storage of the intermediate result (inclusive carry)
- *S-Reg-2* ($u - 1$ bits): storage of the intermediate result
- *Control-Reg* (3 bits): control of the multiplexers and clock enables

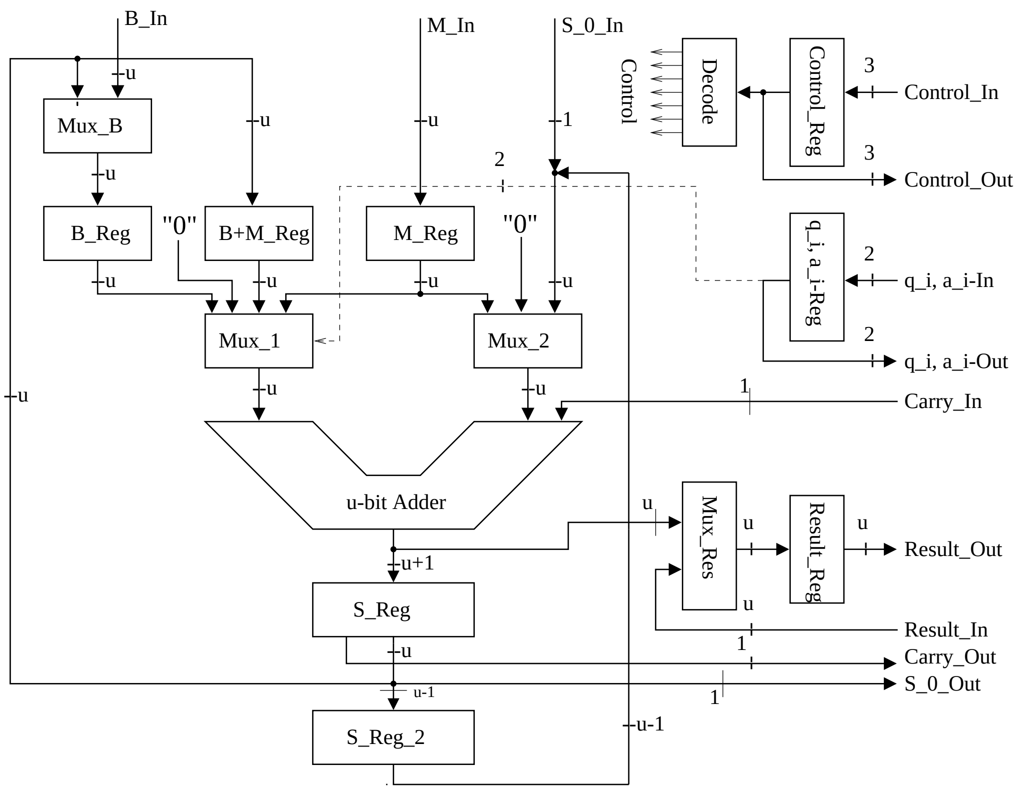


Figure 6.1: Processing element (unit) of Design 1 that computes $S_{i+1} = (S_i + q_i \cdot M)/2 + a_i \cdot B,\ q_i, a_i \in \{0,1\}$

- a_i,q_i (2 bits): multiplier A, quotient Q
- *Result-Reg* (u bits): storage of the result at the end of a multiplication

The registers need a total of $(6u + 5)/2$ CLBs, the adder $u/2 + 2$ CLBs, the multiplexers $4 \cdot u/2$ CLBs, and the decoder 2 CLBs. The possibility of re–using registers for combinatorial logic allows some savings of CLBs. Mux_B and Mux_{Res} are implemented in the CLBs of *B-Reg* and *Result-Reg*, Mux_1 and Mux_2 partially in *M-Reg* and *B+M-Reg*. The resulting costs are approximately $3u + 4$ CLBs per u–bit processing unit. That is 3 to 4 CLBs per bit, depending on the unit size u.

Let's compare this expense to the resources needed for a one bit unit implementation ($u = 1$). We would need a total of seven bit register space (M, B, a_i, q_i, control(2) and result) plus eventually a $B + M$ register and a 4-bit input – 3 bit output (2 carries, result) adder. Together with one or two CLBs for decoding the control word and multiplexing, we would have a total of 6 or 7 CLBs per unit. With such a large amount of CLBs we can implement a much faster architecture, as we will see in Chapter 7.

Before a unit can compute a modular multiplication, the system parameters have to be loaded. M is stored into *M-Reg* of the unit. At the beginning of a modular multiplication, the operand B is loaded from either *B-in* or *S-Reg*, according to the select line of multiplexer *B-Mux*. The next step is to compute $M + B$ once and store the result in the *B+M-Reg*. This operation needs two clock cycles, as the result is clocked into *S-Reg* first. The select lines of Mux_1 and Mux_2 are controlled by a_i or the control word respectively.

In the following $2(m + 2)$ cycles a modular multiplication is computed according to Algorithm 4.6. Multiplexer Mux_1 selects one of its inputs 0, M, B, $B + M$ to be fed in the adder according to the value of the binary variables a_i and q_i. Mux_2 feeds the $u - 1$ most significant bits of the previous result $S\text{-}Reg_2$ plus the least significant result bit of the next unit (division by two/shift right) into the second input of the adder. The result is stored in *S-Reg* for one cycle. The least significant bit goes into the unit to the right (division by two / shift right) and the carry to the unit to the left. In this cycle a second modular multiplication is calculated in the adder, with updated values of $S\text{-}Reg_2$, a_i and q_i. The second multiplication uses the same operand B but a different operand A.

At the end of a modular multiplication, S_{m+3} is valid for one cycle at the output of the adder. This value is both stored into *Result-Reg*, as fed via *S-Reg* into *B-Reg*. The result of the second multiplication is fed into *Result-Reg* one cycle later.

6.3 Modular Multiplication

Figure 6.2 shows how the processing elements are connected to an array for computing an m–bit modular multiplication. To compute $S_{i+1} = (S_i + q_i \cdot M)/2 + a_i \cdot B$ with $M = \sum_{i=0}^{m-1} 2^i \cdot m_i$, we need $m/u + 1$ units. $Unit_1 \ldots Unit_{(m/u)-1}$ are designed as described in Section 6.2. $Unit_0$ has only $u-1$ B inputs as B_0 is added to a shifted value $S_i + q_i M$. The result bit $S\text{-}Reg_0$ is always zero according to the properties of Montgomery's algorithm. $Unit_{m/u}$ processes the most significant bit of B and the temporary overflow of the intermediate result S_{i+1}. There is no M input into this unit.

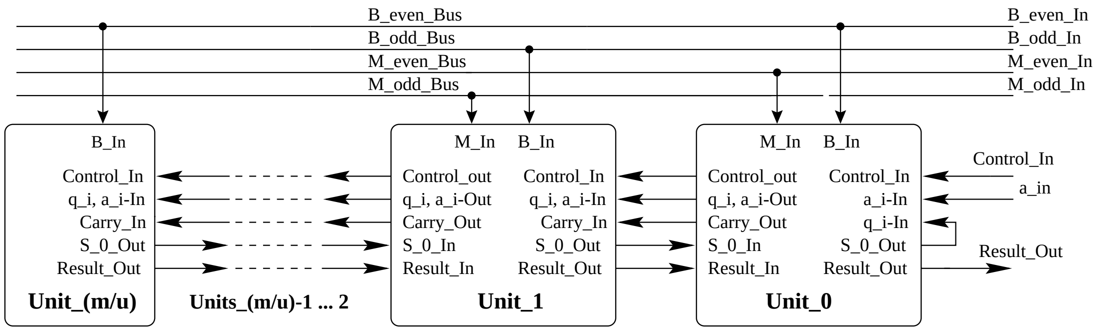


Figure 6.2: Systolic Array for modular multiplication

The inputs and outputs of the units are connected to each other in the following way. The control word, q_i and a_i are pumped from right to left through the units. The result is pumped from left to right. The *carry-out* signals are fed to the *carry-in* inputs to the right. Output *S_0_Out* is always connected to input *S_0_In* of the unit to the right. This represents the division by 2 of the equation.

At first the modulus M is fed into the units. To allow enough time for the signals to propagate to all the units, M is valid for two clock cycles. We use two *M-Buses*, the *M-even-Bus* connected to all even numbered units $unit_0$, $unit_2 \ldots$ $unit_{m/u}$, and the *M-odd-Bus* connected to all odd numbered units $unit_1$, $unit_3 \ldots$ $unit_{(m/u)-1}$. This approach allows to feed u bits of $\tilde{M}$ to the units per clock cycle. Thus it takes m/u cycles to load the full modulus M.

The operand B is loaded similarly. The signals are also valid for two clock cycles. We also use two *B-Buses*, the *B-even-Bus* connected to all even numbered units $unit_0$, $unit_2 \ldots unit_{m/u}$, and the *B-odd-Bus* connected to all odd numbered units $unit_1$, $unit_3 \ldots unit_{(m/u)-1}$.

After the operand B is loaded, the computation of Algorithm 4.6 can begin. Starting at the rightmost $unit_0$, the control word, a_i, and q_i are fed into their registers. The adder computes *S-Reg-2* plus B, M, or $B + M$ in one clock cycle according to a_i and q_i. The least significant bit of the result is read back as q_{i+1} for the next computation. The resulting carry bit, the control word, a_i and q_i are pumped into the unit to the left, where the same computation takes place in the next clock cycle. In such a systolic fashion the control word, a_i, q_i, and the carry bits are pumped from right to left through the whole unit array. The division by two in Algorithm 4.6 leads also to a shift–right operation. The least significant bit of a unit's addition (S_0) is always fed back into the unit to the right. After a modular multiplication is completed, the results are pumped from left to right through the units and consecutively stored in RAM for further processing.

A single processing element computes u bits of $S_{i+1} = (S_i + q_i \cdot M)/2 + a_i \cdot B$ of Algorithm 4.6. In clock cycle i, $unit_0$ computes bits $0 \ldots u-1$ of S_i. In cycle $i+1$, $unit_1$ uses the resulting carry and computes bits $u \ldots 2u-1$ of S_i. $Unit_0$ uses the right shifted (division by 2) bit u of S_i (S_0) to compute bits $0 \ldots u-1$ of S_{i+1} in clock cycle $i+2$.

Clock cycle $i+1$ is unproductive in $unit_0$ while waiting for the result of $unit_1$. This inefficiency is avoided by computing squares and multiplications in parallel according to Algorithm 4.2. Both p_{i+1} and z_{i+1} depend on z_i. We therefore store the intermediate result z_i in the B–Registers and feed z_i and p_i into the a_i input of the units for squaring and multiplication.

6.4 Modular Exponentiation

6.4.1 Data Flow

Figure 6.3 shows how the array of units is utilized for modular exponentiation. At the heart of our design is a finite state machine (FSM) with 17 states. An *idle* state, four states for loading the system parameters, and four times three states for computing the modular exponentiation. The actual modular exponentiation is executed in four main states, *pre-computation*$_1$, *pre-computation*$_2$, *computation*, and *post-computation*. Each of these main states is subdivided in three sub–states, *load-B*, *B+M*, and *calculate-multiplication*. The control word fed into *control-in* is encoded according to the states. The FSM is clocked at half the clock rate. The same is true for loading and reading the RAM and DP RAM elements. This measure makes sure the maximal propagation time is in the units. Thus the minimal clock cycle time and the resulting speed of a modular exponentiation relates to the effective computation time in the units and not to the computation of overhead.

Before a modular exponentiation is computed, the system parameters have to be loaded. The modulus M is read $2u$ bits at the time from I/O into *M-Reg*. Reading starts from low order bits to high order bits. M is fed from *M-Reg* u bits at the time alternatively to *M-even-Bus* and *M-odd-Bus*. The signals are valid two cycles at a time. The exponent E is read 16 bits at the time from I/O and stored into *Exp-RAM*. The first 16 bit wide word from I/O specifies the length of the exponent in bits. Up to 64 following words contain the actual exponent. The pre–computation factor $2^{2(m+2)}$ mod M is read from I/O $2u$ bits at the time. It is stored into *Prec-RAM*.

In state *Pre-compute*$_1$ we read the X value from I/O, u bits per clock cycle, and store it into *DP RAM Z*. At the same time the pre–computation factor $2^{2(m+2)}$ mod

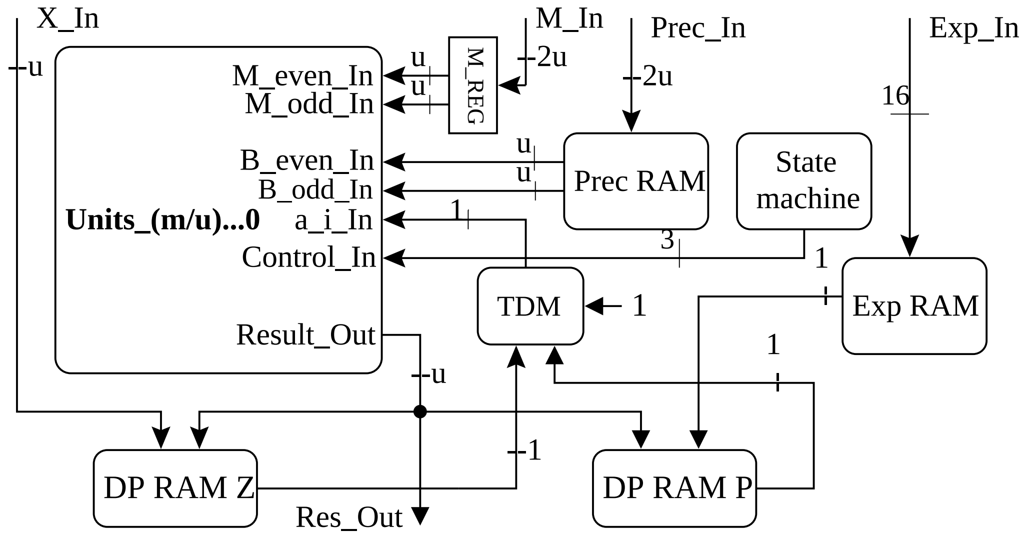


Figure 6.3: Design for a modular exponentiation

M is read from *Prec RAM* and fed u bits per clock cycle alternatively via the *B-even-Bus* and *B-odd-Bus* to the B–registers of the units. In the next two clock cycles, $B + M$ is calculated in the units.

Now we begin calculating the pre–computation. The initial values of Algorithm 4.2 are $P_0 = 1$ and $Z_0 = X$. Both values have to be multiplied by $2^{2(m+2)} \bmod M$. This can be done in parallel as both multiplications use a common operand $2^{2(m+2)} \bmod M$, that is already stored in B. The time division multiplexing unit (TDM) reads X from *DP RAM Z* and multiplexes X and $1, 0 \ldots 0$ on the a_i–bus into the units.

After $2(m+3)$ clock cycles the low order bits of the result of MONT_R2($2^{2(m+2)} \bmod M$, X)= $X \cdot 2^{m+2} \bmod M$ appear at *Result-Out* and are stored in *DP RAM Z*. The low order bits of the result of MONT_R2($2^{2(m+2)} \bmod M$, 1)= $2^{m+2} \bmod M$ appear at *Result-Out* one cycle later and are stored in *DP RAM P*. This process repeats for $2m$ cycles, until all digits of the two results are saved in *DP RAM Z* and *DP RAM P*. The result $X \cdot 2^{m+2} \bmod M$ is also stored in the B-registers of the units.

In state *pre-compute*$_2$ the actual computation of Algorithm 4.2 begins. For both calculations of Z_1 and P_1 we use Z_0 as an operand. This value is stored in the B-registers. The second operand Z_0 or P_0 respectively, is read from *DP RAM Z* and *DP RAM P* and "pumped" via *TDM* as a_i into the units. After another $2(m+3)$ clock cycles the low order bits of the result of Z_1 and P_1 appear at *Result-Out*. Z_1 is stored in *DP RAM Z*. P_1 is needed only if the first bit of the exponent e_0 is equal to "1". Depending on e_0, P_1 is either stored in *DP RAM P* or discarded.

In state *compute* the loop of Algorithm 4.2 is executed $n-1$ times. Z_i in *DP RAM Z* is updated after every cycle and "pumped" back as a_i into the units. P_i in *DP RAM P* is updated only if the relevant bit of the exponent e_i is equal to "1". In this way always the last stored P_i is "pumped" back into the units.

After the processing of e_{n-1}, the FSM enters state *post-compute*. $P_n = X^E \bmod M \cdot 2^{m+2} \bmod M$ is stored in *DP RAM P* now. To eliminate the factor 2^{m+2} (Section 4.3) from the result P_n, we compute a final Montgomery multiplication MONT(P_n, "1"). First the vector $0, 0, \ldots 0, 1$ is fed alternatively via the *B-even-Bus* and *B-odd-Bus* into the B–registers of the units. P_n is "pumped" from *DP RAM P* as a_i into the units. After state *post-compute* is executed, u bits of the result $P_n = X^E \bmod M$ are valid at the I/O port. Every two clock cycles another u bits appear at I/O. State *pre-compute*$_1$ can be re–entered immediately now for the calculation of another X value.

A full modular exponentiation is computed in $2(n+2)(m+4)$ clock cycles. That is the delay it takes from inserting the first u bits of X into the device until the first u result bits appear at the output. At that point, another X value can enter the device. With a additional latency of m/u clock cycles the last u bits appear on the output bus.

6.4.2 Function Blocks

In this subsection we explain the function blocks in Figure 6.3: *DP RAM P*, *DP RAM Z*, *EXP RAM*, and *Prec RAM*.

Figure 6.4 shows the design of *DP RAM Z*. An $m/u \times u$ bit DP RAM is at the heart of this unit. It has separate write (A) and read ($DPRA$) address inputs. The

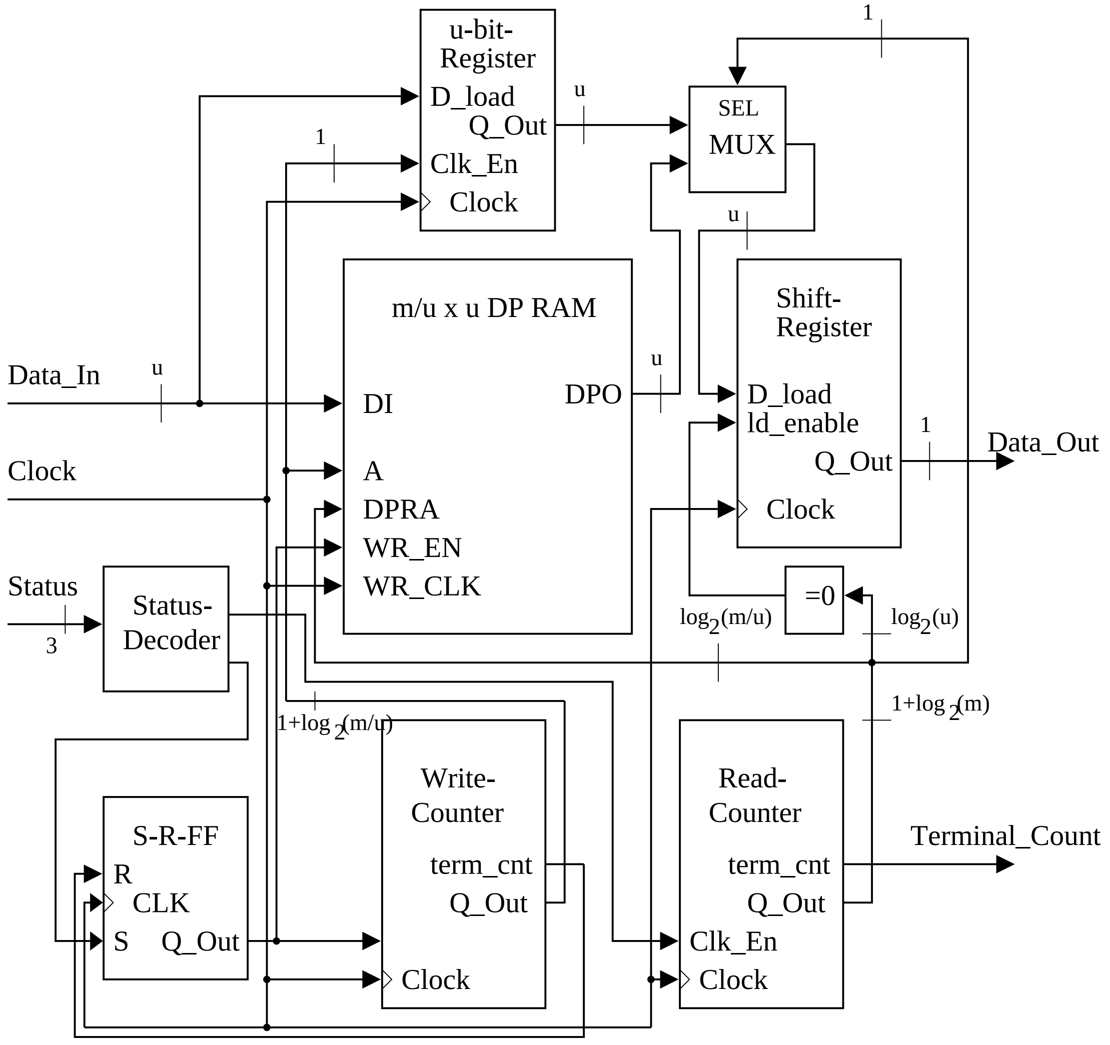


Figure 6.4: DP RAM Z Unit

write-counter counting up to m/u computes the write address (A). The *write-counter* starts counting (*clock-enable*) in sub–states *B-load* when the first u bits of Z_i appear at *data_in*. At the same time the enable signal of the DP RAM is active and data is stored in DP RAM. *Terminal-count* resets *count–enable* and *write–enable* of DP RAM when m/u is reached. The *read-counter* is enabled in the sub–states *compute*. When *read-counter* reaches its upper limit $m+2$, *terminal-count* triggers the FSM to transit into sub-state *B-load*. The $log_2(m/u)$ most significant bits of the *read-counter* value (*q_out*) address *DPRA* of the DP RAM. Every u cycles another value stored in the DP RAM is read. This value is loaded into the shift register when the $log_2(u)$ least significant bits of *q_out* reach zero. The next u cycles u bits appear bit by bit at the serial output of the shift register. The last value of z_i is stored in a u–bit register. This measure allows us to select an $m/u \times u$–bit DP RAM instead of an $2m/u \times u$–bit DP RAM ($m = 2^x$, $x = 8, 9, 10$).

DP RAM P works almost the same way. It has an additional input e_i, that activates the *write-enable* signal of the DP RAM in the case of $e_i = 1$.

Figure 6.5 shows the design of *Exp RAM*. For the simulation results please refer to Figure C.9 in the appendix. In the first cycle of the *load-exponent* state, the first word is read from I/O and stored into the 10–bit register. Its value specifies the length of the exponent in bits. In the next cycles the exponent is read 16–bit at a time and stored in RAM. The storage address is computed by a 6–bit *write counter*. At the beginning of each *compute* state the 10–bit *read counter* is enabled. Its 6 most significant bits compute the memory address. Thus every 16th activation, a new value is read from RAM. This value is stored in the 16–bit *shift–register* at the same time (when the 4 least significant bits of *read counter* are equal to zero). When *read counter* reaches the value specified in the 10–bit register, the *terminate* signal triggers the FSM to enter state *post-compute*.

Figure 6.6 shows the design of *Prec RAM*. In state *load–pre–factor* the precomputation factor is read $2u$ bits at the time from I/O and stored in RAM. A

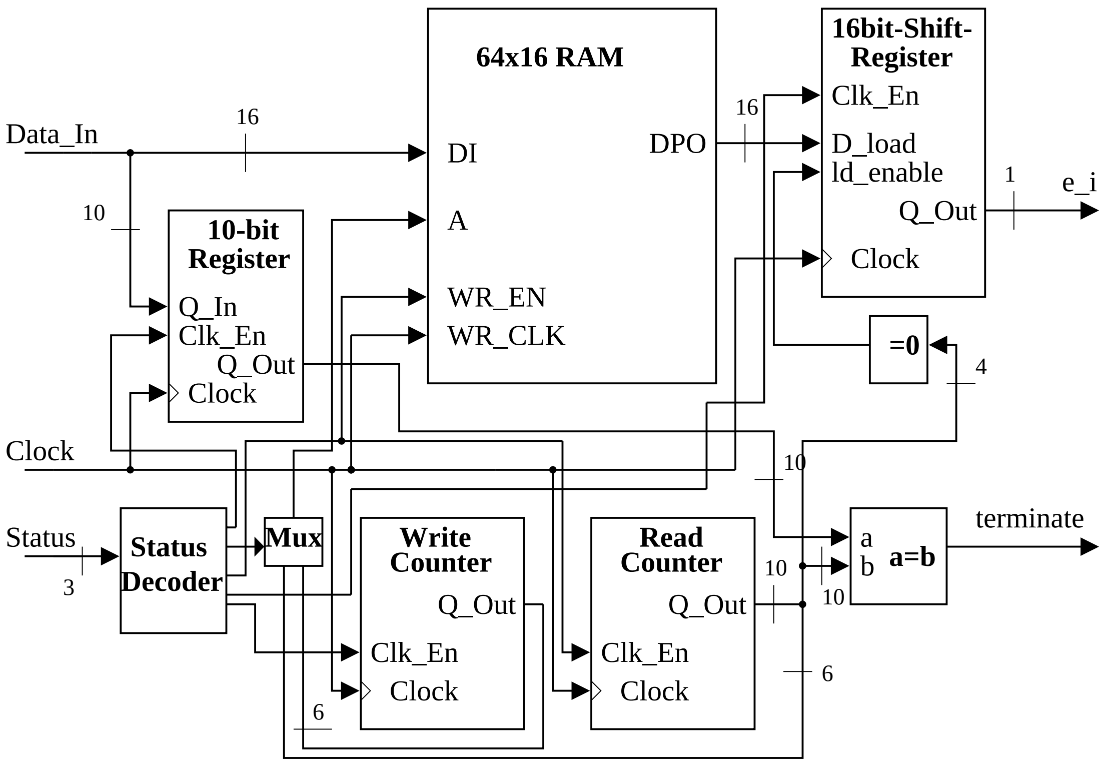


Figure 6.5: Exp RAM Unit

counter that counts up to $m/2u$ addresses the RAM. When all $m/2u$ values are read, the *terminal-count* signal triggers the FSM to leave state *load–pre–factor*. In state *pre–compute*$_1$ the pre–computation factor is read from RAM and fed to the B–registers of the units. The counter is incremented each clock cycle and $2u$ bits are loaded in the $2u$–bit register. From there u bits are fed on *B-even-bus* each positive edge of the clock. On the negative clock edge, u bits are fed on the *B-odd-bus* via the u–bit register.

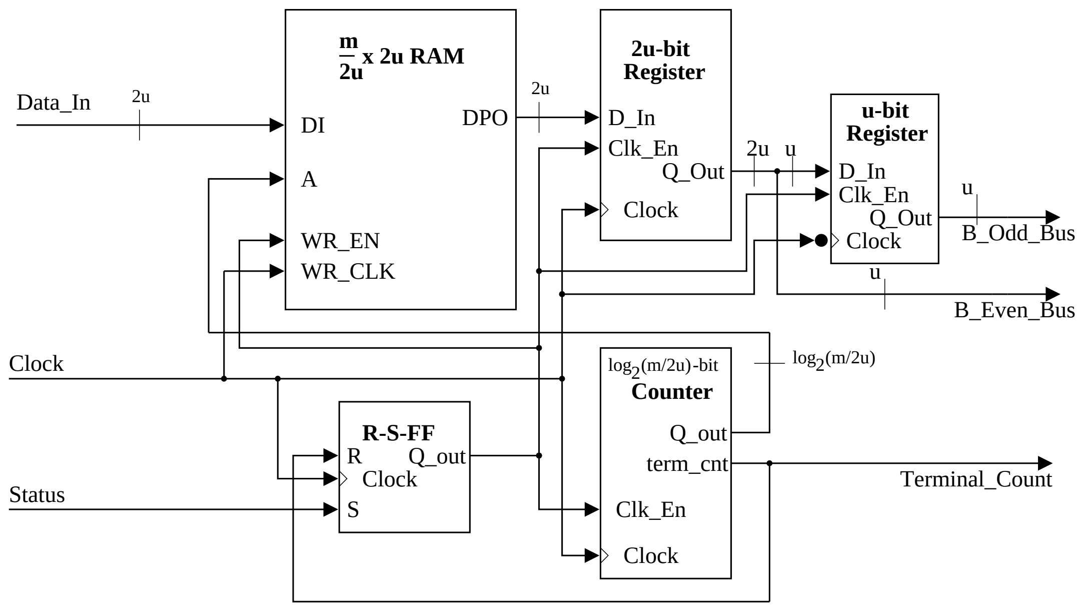


Figure 6.6: Prec RAM Unit

Chapter 7

Design 2: A Speed Efficient Architecture

In this chapter we describe our second architecture. The goal was to design a speed efficient architecture using Algorithm 4.8 with a larger radix. As target devices we use the Xilinx XC4000 family as described in Chapter 5. The results of the actual implementation of the architecture will be described in Chapter 9.

7.1 Design Overview

Design 1, described in Chapter 6, was optimized in terms of resource usage. In order to speed–up the design two approaches can be taken. We can either try to reduce the cycle time, or the number of cycles per modular multiplication. Reduction of the cycle time can be achieved by computing one instead of u bits per unit. Compared to a design with 4 bit units, the resulting speed–up of approximately 20% (without three ripple carry delays) comes at the expense of additional 30% resources. Thus the time–area product becomes worse. Section 4.4 describes how the number of steps per modular multiplication can be reduced. Using a radix $r = 2^k$, $k > 1$, reduces the number of steps in Algorithm 4.6 by a factor k. The following computation of

Algorithm 4.8 has to be executed $m+3$ times ($i = 0\ to\ m+2$):

$$q_i = S_i \bmod 2^k$$

$$S_{i+1} = (S_i + q_i \cdot \tilde{M})/2^k + a_i \cdot B$$

where $B = \sum_{i=0}^{m+1} (2^k)^i \cdot b_i$;
$A = \sum_{i=0}^{m+2} (2^k)^i \cdot a_i,\ a_{m+2} = 0$;
$\tilde{M} = \sum_{i=0}^{m} (2^k)^i \cdot \tilde{m}_i$;
$M = \sum_{i=0}^{m-1} (2^k)^i \cdot m_i$;
Please note that q_i and a_i are digits with k bits. One of the major problems when implementing this equation is computing multiples of B and $\tilde{M}$. Reference [9] proposes a multiplexer network. This approach is not suitable for a systolic array implementation into FPGA because of the following reasons:

1. For a radix of 2^2 the multiplexer could be implemented in one CLB per bit length, but already a radix of 2^4 uses more than four CLBs per bit. This would result in unrealistically large CLB counts for secure bit length.

2. In a systolic array we typically compute k bits per processing element. With a multiplexer solution the internal bit length becomes $2k$ resulting in twice as much costs for adders and registers.

To avoid the doubling of the internal bit length of a unit the following approach which is optimized for the CLB architectures at hand can be taken.

1. Pre-compute the multiples of B and $\tilde{M}$ at the beginning of the execution of Montgomery's algorithm and store the results for further use.

2. Let the carries of this pre–computations propagate to the units to the left.

If a unit processes k bits, the stored multiples will also have k bits and the internal bit length will not exceed $k+2$ bits (addition of 3 operands). The cost is additional

2^k clock cycles for calculating the 2^k multiples of B. For small k values, this expense is negligible compared to the total amount of $2(m+3)$ cycles for the whole algorithm. As the value of $\tilde{M}$ is changed infrequently, we can compute its multiples externally and store these values into the units. This approach saves some additional CLBs. As storage elements we can either use registers or RAM elements. For k larger than 2, registers are not suitable as they utilize one CLB per 2 stored bits. RAM elements are very efficient up to an address width of 4 bits. Their implementation requires only one CLB per two bits data width (2 CLBs for a 16×4 bit RAM). The resource requirements grow rapidly, though, for larger address width. A 64×6 bit implementation ($k = 6$) utilizes 18 CLBs, a 256×8 bit implementation ($k = 8$) utilizes 96 CLBs (see Table 5.2). Both would result in unrealistically large CLB counts for secure bit length. Additionally the 2^6 or even 2^8 clock cycles for computing the multiples of B are not negligible any more. To achieve an optimal time–area product we implemented therefore an architecture with a radix $r = 2^4$. We compute 4 bits per processing element. The multiples of $\tilde{M}$ are computed externally once and stored in the units. Similar to Design 1, we use square and multiply Algorithm 4.2 and compute squares and multiplications in parallel.

Design 2 can be divided hierarchically into three levels.

Processing Element Computes 4 bits of a modular multiplication.

Modular Multiplication An array of processing elements computes a modular multiplication.

Modular Exponentiation Combines modular multiplications to a modular exponentiation according to Algorithm 4.2.

In the following we describe the system with a bottom–up approach.

7.2 Processing Elements

Figure 7.1 shows the implementation of a processing element.

$$S_{i+1} = (S_i + q_i \cdot \tilde{M})/2^k + a_i \cdot B$$

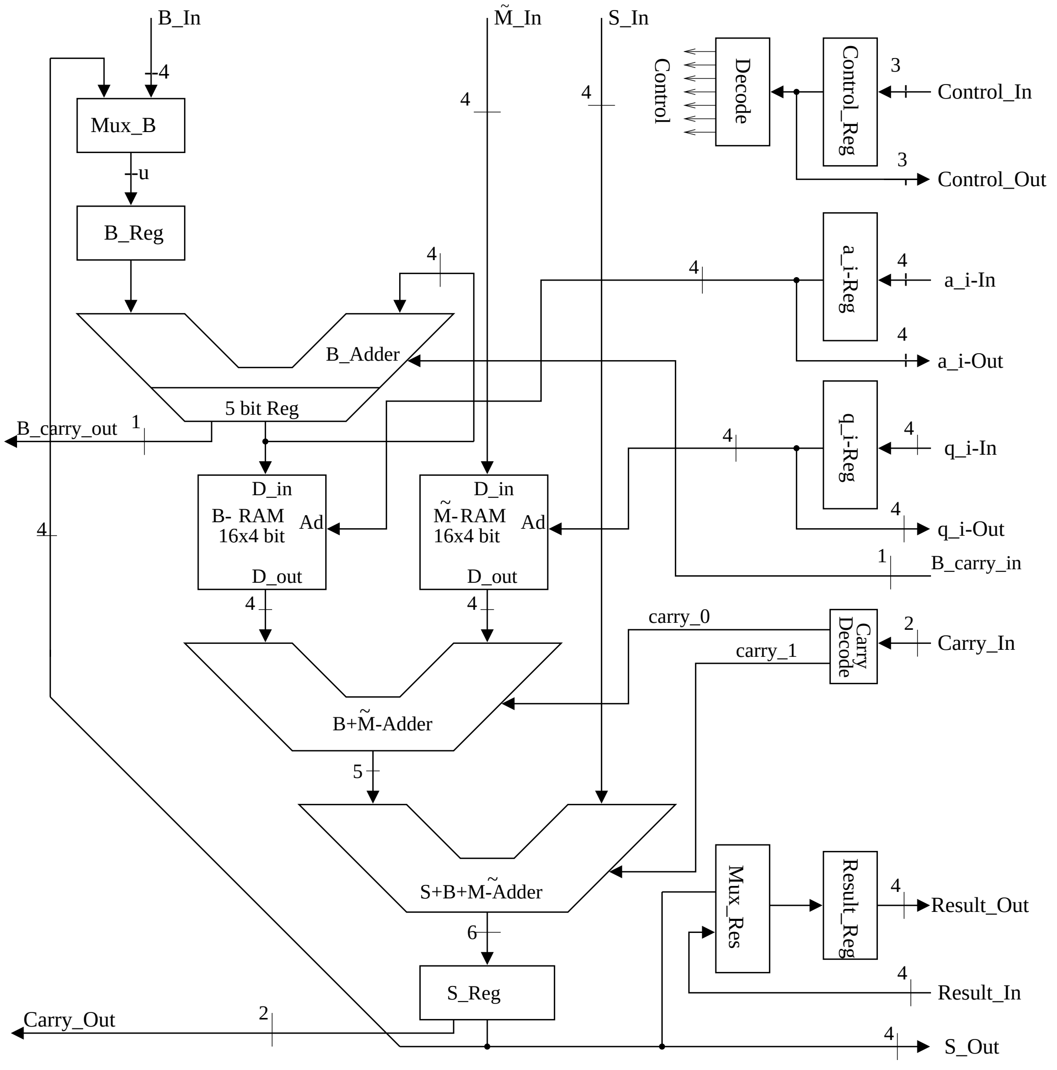


Figure 7.1: Processing Element (unit)

The following elements are needed:

- *B-Reg* (4 bits): storage of the B multiplier
- *B-Adder-Reg* (5 bits): storage of multiples of B
- *S-Reg* (4 bits): storage of the intermediate result S_i (Algorithm 4.8)
- *Control-Reg* (3 bits): control of the multiplexers and clock enables
- a_i*-Reg* (4 bits): multiplier A
- q_i*-Reg* (4 bits): quotient Q
- *Result-Reg* (4 bits): storage of the result at the end of a multiplication
- *B-Adder* (4 bits): Adds B to the previously computed multiple of B
- *B+*$\tilde{M}$*-Adder* (4 bits): Adds a multiple of $\tilde{M}$ to a multiple of B
- *S+B+*$\tilde{M}$*-Adder* (5 bits): Adds the intermediate result S_i to $B + \tilde{M}$
- *B-RAM* (16x4 bits): Stores 16 multiples of B
- $\tilde{M}$*-RAM* (16x4 bits): Stores 16 multiples of $\tilde{M}$

For a timing model of the processing element shown in Figure 7.1 please refer to Figure C.2 in the appendix.

The registers need a total of 14 CLBs, the adders 13 CLBs and the RAM blocks 4 CLBs. The possibility of re–using registers for combinatorial logic allows some savings of CLBs. Thus a processing element utilizes a total of 24 CLBs, which is equal to 6 CLBs per processed bit.

Before a unit can compute a modular multiplication, the system parameters have to be loaded. The relevant bits of the multiples of $\tilde{M}$ are loaded into $\tilde{M}$*-RAM* starting from $\tilde{M}$, $2 \cdot \tilde{M}$ to $15 \cdot \tilde{M}$. Each multiple of $\tilde{M}$ is valid on the input $\tilde{M}$*-in* for 2 clock cycles. The write enable signal for $\tilde{M}$*-RAM* is decoded from *Control-In*. The address input q_i is incremented every two cycles.

At the beginning of a modular multiplication, the operand B is loaded from either *B-in* or *S-Reg*, according to the select line of multiplexer *B-Mux*. This value is stored in *B-RAM* while address input a_i is equal to one. The write enable signal for *B-RAM* is decoded from *control-in*. After each clock cycle, B is added to the accumulated B multiple and stored in *B-RAM*, while a_i is incremented by one. The resulting carry propagates to the adjacent unit. The pre–computation and storage of the multiples of B takes 16 clock cycles. For a simulation of this behavior please refer to Figure C.1.

When computing the modular multiplication, the relevant multiples of B and $\tilde{M}$ are addressed by a_i and q_i. Both values are added up and added to *S-in* (S_i in Algorithm 4.8). The result of these additions will not exceed 47 ($3{\cdot}15+$*(carry-in* $= 2$)). Thus *carry-out* $\in \{0, 1, 2\}$. This behavior saves us another full adder. The incoming *carry-in* is decoded in the *carry-decode* unit and the resulting signals *carry_0* (*carry-in* $= 2$) and *carry_1* (*carry-in* $= 1$ *or* 2) can be fed in the *carry-in* inputs of the two adders. A simulation of a modular multiplication can be found in Figure C.2 in the appendix.

At the end of a modular multiplication the result S_{m+3} appears in *S-Reg*. This value is stored via *Mux-Res* into *Result-Reg* and via *B-Mux* into *B-Reg* for further processing (see Figure C.3 in the appendix).

7.3 Modular Multiplication

Figure 7.2 shows how the processing elements are connected to an array for computing a full size modular multiplication. To compute $S_{i+1} = (S_i + q_i \cdot \tilde{M})/16 + a_i \cdot B$ with operand $B = \sum_{i=0}^{m+1} 16^i \cdot b_i$, we need $m + 3$ units. Units $1 \ldots m + 1$ are designed as described in Section 7.2. *Unit*$_0$ does not have a B input as B is not shifted by 4 bits (division by 16) in above mentioned equation. The four result bits *S-Reg*$_{3\ldots0}$ are

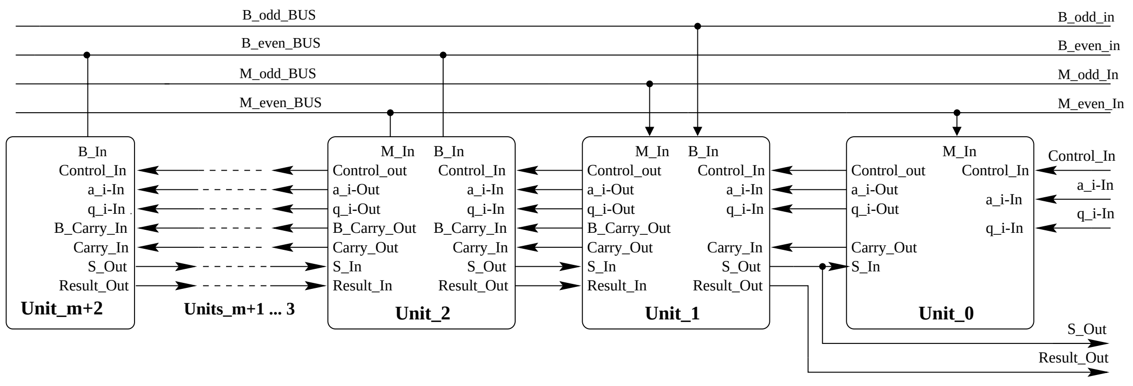


Figure 7.2: Systolic array for modular multiplication

always equal to zero according to the properties of Montgomery's algorithm. $Unit_{m+2}$ on the other hand does not have an M input. It processes the most significant bit of B and the temporarily occurring overflow of S_{i+1}.

The inputs and outputs of the units are connected to each other in the following way. The control word, q_i and a_i are pumped from right to left through the units. The result is pumped from left to right. The *carry-out* signals are fed to the *carry-in* inputs to the right. Output *S_out* is always connected to input *S_in* of the unit to the right. This represents the division by 16 of the computation.

At first the multiples of the modulus $\tilde{M}$ are fed into the units. To allow enough time for the signals to propagate to all the units, the modulus $\tilde{M}$ is valid for two clock cycles. We use two $\tilde{M}$*-buses*, the $\tilde{M}$*-even-bus* connected to all even numbered units $unit_0$, $unit_2 \ldots unit_{m+2}$, and the $\tilde{M}$*-odd-bus* connected to all odd numbered units $unit_1$, $unit_3 \ldots unit_{m+1}$. This approach allows to feed 4 bits of $\tilde{M}$ to the units per clock cycle. It takes $m+2$ cycles to feed one multiple of $\tilde{M}$ into all the units, a total of $15 \cdot (m+2)$ cycles to load all multiples of $\tilde{M}$. As this step is executed only when the modulus is changed, the long load time can be accepted.

The operand B is loaded similarly. The signals are also valid for two clock cycles. We also use two *B-buses*, the *B-even-bus* connected to all even numbered units

$unit_2$, $unit_4 \ldots unit_{m+2}$, and the *B-odd-bus* connected to all odd numbered units $unit_1$, $unit_3 \ldots unit_{m+1}$. In cycle 1 the control word with state *load-B* is fed into *control-in* of $unit_0$ and is valid for two cycles. In cycle 2, the control word propagates to $unit_1$. In cycle 2 and 3, bits $b_0 \ldots b_3$ are fed on the *B-odd-bus* and stored in $unit_1$. In cycle 3 and 4, bits $b_4 \ldots b_7$ are fed on the *B-even-bus* and stored in $unit_2$. Also in cycle 3 the control word with state *multiple-B* is fed into *control-in* of $unit_0$ and is valid for 16 cycles. $Unit_1$ computes the multiples of B in cycles 4 to 20, $unit_2$ in cycles 5 to 21. In cycle 19, $Unit_0$ starts to compute a squaring according to Algorithm 4.8. Therefore bits $a_0 \ldots a_3$ of the operand A are fed in a_i-*in*, and the control word with state *calculate-multiplication* in *control-in*. In cycle 20, $unit_1$ uses the resulting carries and computes bits $0 \ldots 3$ of S_1. These bits are fed back either in *S-in*, as well as in q_i-*in* of $unit_0$. $Unit_0$ is ready to compute the second loop of the squaring S_2 in cycle 21.

Clock cycle 20 is unproductive in $unit_0$ while waiting for the result *S-out* of $unit_1$. This inefficiency is avoided by computing squares and multiplications in parallel according to Algorithm 4.2. Both p_{i+1} and z_{i+1} depend on z_i. We therefore store the intermediate result z_i in the B–Registers and feed z_i and p_i into the a_i input of the units for squaring and multiplication.

In cycle $20 + 2(m + 2)$ bits $0 \ldots 3$ of S_{m+3} are available at *result-out* of $unit_1$. Thus the first modular multiplication takes $2m + 22$ cycles from feeding $b_0 \ldots b_3$ on the bus until the result is available. Further squaring operations take $2m + 20$ cycles. While the last loop of the modular multiplication is computed, the result of the squaring is stored in *B-Reg* in only one cycle. Thus the two cycles for loading B from the bus are not needed.

For the simulation of the systolic array please refer to Figures C.4, C.5 and C.6 in the appendix.

7.4 Modular Exponentiation

7.4.1 Data Flow

Figure 7.3 shows how the array of units is utilized for modular exponentiation. For simulation results please refer to Figures C.7, C.8 and C.9 in the appendix.

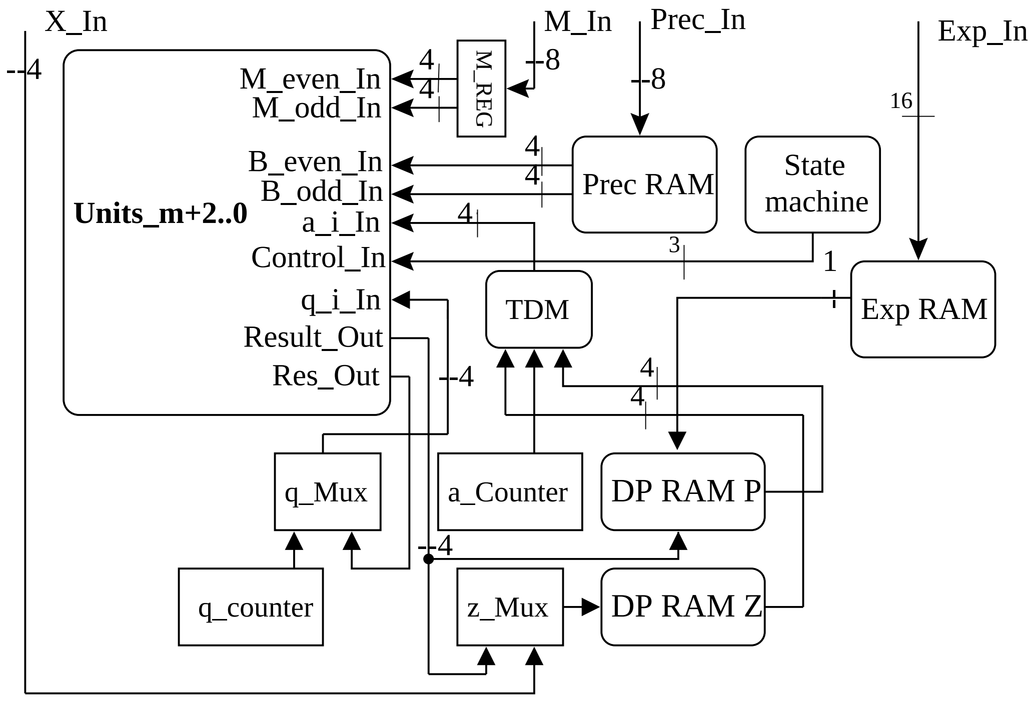


Figure 7.3: Design for a modular exponentiation

At the heart of our design is a finite state machine (FSM) with 17 states. An *idle* state, four states for loading the system parameters, and four times three states for computing the modular exponentiation. The actual modular exponentiation is executed in four main states, *pre-computation*$_1$, *pre-computation*$_2$, *computation*, and *post-computation*. Each of these main states is subdivided in three sub–states, *load-B*, *multiple-B*, and *calculate-multiplication*. The control word fed into *control-in* is encoded according to the states. The FSM is clocked at half the clock rate. The same is true for loading and reading the RAM and DP RAM elements. This measure makes sure the maximal propagation time is in the units. Thus the minimal clock cycle time and the resulting speed of a modular exponentiation relates to the effective computation time and not to the computation of overhead.

Before a modular exponentiation is computed, the system parameters have to be loaded. Multiples of the modulus $\tilde{M}$ have to be computed externally and are read eight bit at a time from I/O in $\tilde{M}$*-Reg*. Reading starts from low order bits to high order bits, and from $\tilde{M}$ to $15 \cdot \tilde{M}$. From $\tilde{M}$-Reg the multiples of $\tilde{M}$ are fed 4 bits at the time alternatively to $\tilde{M}$*-even-bus* and $\tilde{M}$*-odd-bus*. The address for the $\tilde{M}$*-RAM* of the units is generated by the *q-counter* and fed via *q-Mux* into q_i*-In*. After loading a full multiple of $\tilde{M}$, *q-counter* is incremented. The exponent E is read 16 bits at the time from I/O and stored into *exp-RAM*. The first 16 bit wide word from I/O specifies the length of the exponent in bits. Up to 64 following words contain the actual exponent. The pre–computation factor $2^{8(m+2)} \bmod M$ is read from I/O eight bits at the time. It is stored into *prec-RAM*.

In state *pre-compute*$_1$ we read the X value from I/O, 4 bits per clock cycle, and store it into *DP RAM Z*. At the same time the pre-computation factor $2^{8(m+2)} \bmod M$ is read from *prec-RAM* and fed 4 bits per clock cycle alternatively via the *B-even-bus* and *B-odd-bus* to the B–registers of the units. In the next 16 clock cycles, the multiples of $B = 2^{8(m+2)} \bmod M$ are calculated.

Now we begin calculating the pre–computation. The initial values of Algorithm 4.2 are $P_0 = 1$ and $Z_0 = X$. Both values have to be multiplied by the pre–computation factor. This can be done in parallel as both multiplications use a common operand $2^{8(m+2)} \bmod M$, that is already stored in B. The time division multiplexing unit (TDM) reads X from *DP RAM Z* and multiplexes X and $1, 0 \ldots 0$ on the a_i–bus into the units. After $2(m+2)$ clock cycles the low order bits of the result of MONT_HR($2^{8(m+2)} \bmod M,\ X$)$=\ X \cdot 2^{4(m+2)} \bmod M$ appear at *result-out* and are stored in *DP RAM Z*. The low order bits of the result of MONT_HR($2^{8(m+2)}$ mod M, 1)= $2^{4(m+2)}$ mod M appear at *result-out* one cycle later and are stored in *DP RAM P*. This process repeats for $2(m+2)$ cycles, until all digits of the two results are saved in *DP RAM Z* and *P*. The result $X \cdot 2^{4(m+2)}$ mod M is also being stored in the B-registers of the units. For the simulation of this behavior please refer to Figure C.7 in the appendix.

In state *pre-compute*$_2$ the actual computation of Algorithm 4.2 begins. For both calculations of Z_1 and P_1 we use Z_0 as an operand. This value is stored in the B-registers. The second operand Z_0 or P_0 respectively, is read from *DP RAM Z* and *P* and "pumped" via *TDM* as a_i into the units (Figure C.8 in the appendix). After another $2(m+2)$ clock cycles the low order bits of the result of Z_1 and P_1 appear at *result-out*. Z_1 is stored in *DP RAM Z*. P_1 is needed only if the first bit of the exponent e_0 is equal to "1". Depending on e_0, P_1 is either stored in *DP RAM P* or discarded.

In state *compute* the loop of Algorithm 4.2 is executed $n-1$ times. Z_i in *DP RAM Z* is updated after every cycle and "pumped" back as a_i into the units. P_i is updated only if the relevant bit of the exponent e_i is equal to "1". In this way always the last stored P_i is "pumped" back into the units.

After the processing of e_{n-1}, the FSM enters state *post-compute*. $P_n = X^E \cdot 2^{4(m+2)}$ mod M is stored in *DP RAM P* now. To eliminate the factor $2^{4(m+2)}$ from the result P_n, we compute a final Montgomery multiplication MONT_HR(P_n, "1"). First the vector $0, 0, \ldots 0, 1$ is fed alternatively via the *B-even-bus* and *B-odd-bus* into the B–registers of the units. P_n is "pumped" from *DP RAM P* as a_i into the units. After state *post-compute* is executed the result X^E mod M is ready at the I/O port. 4 bits appear every two clock cycles. State *pre-compute*$_1$ can be re–entered immediately now for the calculation of another X value.

A full modular exponentiation is computed in $(n+2)(2m+20)$ clock cycles. That is the delay it takes from inserting the first 4 bits of X into the device, until the first 4 result bits appear at the output. At that point, another X value can enter the device. With an additional latency of $m + 2$ clock cycles the last 4 bits appear on the output bus.

7.4.2 Function Blocks

In this subsection we explain the function blocks in Figure 6.3: *DP RAM P* and *DP RAM Z*. The modules *EXP RAM* and *Prec RAM* are explained in Section 6.4.2.

Figure 7.4 shows the design of *DP RAM Z*. An $m \times 4$ bit DP RAM is at the heart of this unit. It has separate write (A) and read ($DPRA$) address inputs. Two counters

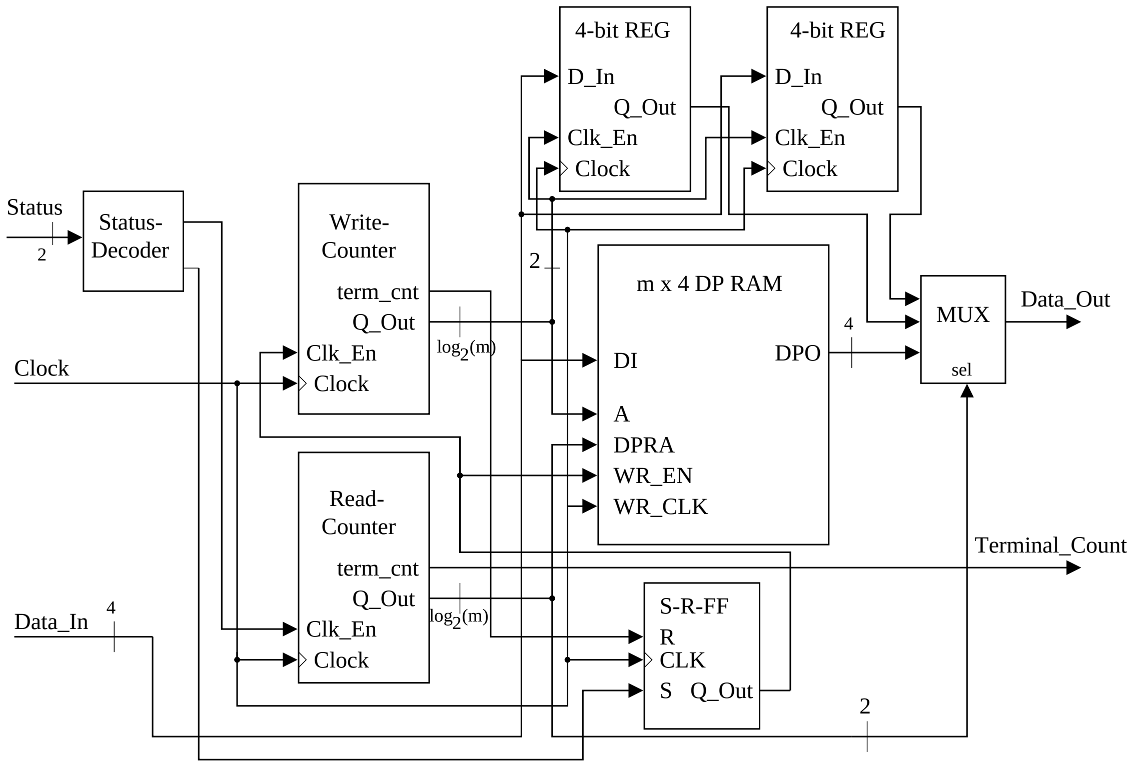


Figure 7.4: DP RAM Z Unit

that count up to $m + 2$ compute these addresses. The *write-counter* starts counting (*clock- enable*) in sub–states *B-load* when the first digit of Z_i appears at *data_in*. At the same time the enable signal of the DP RAM is active and data is stored in DP RAM. When $m+2$ is reached, the *terminal-count* signal of the write-counter resets the two enable signals. The *read-counter* is enabled in sub–states *compute*. The data of DP RAM is addressed by *q_out* of the *read-counter* and appears immediately at *DPO*. When *read-counter* reaches $m+2$, *terminal-count* triggers the FSM to transit into sub-state *B-load*. The last two values of z_i are stored in a 4–bit register each. This measure allows us to choose a 100% utilized $m \times 4$–bit DP RAM instead of an only 50% utilized $2m \times 4$–bit DP RAM.

DP RAM P works almost the same way. It has an additional input e_i, that activates the *write-enable* signal of the DP RAM in the case of $e_i =$ "1″.

Chapter 8

Methodology

In our implementation we adopted the following design approach that resulted in fast verification of gate level netlists as well as back annotated designs:

1. Design entry
2. Logic verification
3. Synthesis
4. Place and Route
5. Timing Verification

The entire design, with the exception of vendor specific soft macros, was entered in VHDL format. Once the design was developed in VHDL, boolean logic and major timing errors were verified by simulating the gate level description with Synopsys VHDL analyzer (vhdlan) and VHDL debugger (vhdldbx) version 1998.08. The next step involved the synthesis of the VHDL code with Synopsys Design Compiler (fpga_analyzer) version 1998.08. The output of this step was an optimized netlist describing the gate level design in XILINX format. The most time consuming step was the compilation of the synthesized design with the place and route tools available from Xilinx. This process was accomplished with the XILINX Design Manager tools version M1.5.19. The final step of the design flow was to verify the design once again but this time with the physical net, CLB, and pad delays introduced when the design was placed into a specific device. This was accomplished with the same test benches and simulation models that were used during the logic verification stage. Synopsys (vhdldbx) was used once again to verify back-annotated designs. The timing results from Chapter 9 were all computed by the Xilinx timing analyzer and verified by the Synopsis vhdl debugger. They were not verified with an actual chip.

8.1 Xilinx Synopsys Interface

Figure 8.1 presents a flow chart diagram of the design flow with Xilinx-Synopsys-Interface (XSI) tools. The XSI tools provide for a transition between results obtained from Synopsys synthesis and the Xilinx place and route tools. The XSI module includes all libraries necessary for Synopsys `fpga_analyzer` to interpret gates into logical blocks so that synthesis can be performed at this level. The design_ware libraries provided by Xilinx are automatically instantiated when possible. Synthesis results include report files on area and timing utilization, design netlist and constraints that are used in the place and route process, and Synopsys design files that describe the entire system.

8.2 Simulation and Verification

Verification of the design is done at two points. First, it is applied to the initial VHDL design. This verifies only the logic without delays. The input to this verification process is a test bench written in VHDL and the actual VHDL design. The results are compared to test vectores generated by `Maples`.

The post place and route verification uses the same test bench. The VHDL input

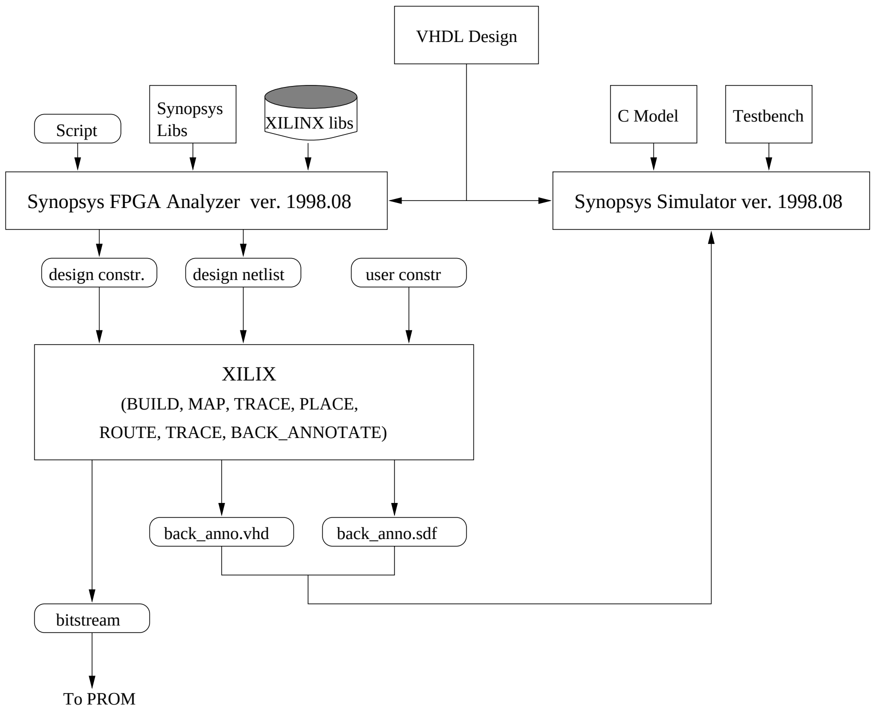


Figure 8.1: Design flow

model to this stage is different. Here the VHDL model is obtained from the XILINX place and route tools. This VHDL model includes a separate file defining all net, CLB, and port delays associated with the placed design. Once again, verification process involves testing all vectors against test vectors generated by `Maples`. A sample test bench can be found in Appendix A.

8.3 Synthesis

To synthesize our designs, script files were developed that could be launched from within the `fpga_analyzer`. These scripts would elaborate, compile, optimize the design, and prepare report summaries. The output of the synthesis tools is a design netlist and constraints file. A sample script file is provided in Appendix B.

8.4 Place and Route

The input to the place and route tools is a design netlist and constraints file generated by Synopsys, as well as a possible user constraints file. The user constraints have higher priority over the Synopsys constraints and may include additional constraints relaxing the clock period or implementing pin assignment. The output of this process is a bit-stream file that can be used to directly program the device and the back-annotated design that can be simulated for timing verification (Figure 8.1).

Chapter 9

Results

9.1 Design 1

We implemented Design 1 for various bit lengths and unit widths. Table 9.1 shows our results in terms of used CLBs (C), clock cycle time (T) and the time–area product (TA).

Table 9.1: Design 1: CLB usage, minimal clock cycle time, and time–area product of modular exponentiation architectures on Xilinx FPGAs

	160 bit			256 bit			512 bit		
	C	T	TA	C	T	TA	C	T	TA
u	[CLBs]	[ns]	[CLB·μs]	[CLBs]	[ns]	[CLB·μs]	[CLBs]	[ns]	[CLB·μs]
4	**951**	**17.3**	**16.4**	**1307**	**17.5**	**22.8**	**2555**	**17.7**	**45.2**
8	**820**	**19.6**	**16.0**	**1122**	**19.8**	**22.2**	**2094**	**19.1**	**39.9**
16	**790**	**21.1**	**16.6**	**1110**	**21.7**	**24.0**	**2001**	**21.8**	**43.6**

	768 bit			1024 bit		
	C	T	TA	C	T	TA
u	[CLBs]	[ns]	[CLB·μs]	[CLBs]	[ns]	[CLB·μs]
4	**3745**	**19.1**	**71.5**	**4865**	**19.2**	**93.4**
8	**3132**	**19.4**	**60.7**	**4224**	**23.4**	**98.8**
16	**2946**	**21.6**	**63.6**	**3786**	**23.7**	**89.7**

The majority of CLBs is used in the units. In Section 6.2 we derived an approximation of $3u + 4$ CLBs per unit which proves to be correct for large designs. The overhead consists mainly of RAM, dual port RAM, shift registers, counters and the state machine. Counters and their decoding for addressing RAM and dual port RAM are more costly for larger designs. On the other hand, we used the same state machine for all designs in Table 9.1.

The clock cycle time T in Table 9.1 is the propagation delay from *B-Reg* through mux_1 and the carries of the adder to the registered carry, plus the setup time of the flip-flop. We compare this delay to the optimal cycle time calculated by the Xilinx timing analyzer; for a 4–bit unit the delay with optimal routing is 10.5 ns (256 and 512 bit designs) and 12.7 ns (768 and 1024 bit designs); for an 8–bit unit 11.2 ns and 13.7 ns and for a 16–bit unit 12.8 ns and 15.5 ns. The larger designs were implemented in larger FPGA devices featuring different delay specifications. Otherwise we expect the same cycle times for designs with the same unit size. The additional routing delay is between 50% and 80% above the optimal propagation delay. For designs up to 768 and 1024 ($u = 4$) bits it remains approximately constant; it deteriorates for 1024 bit designs with unit sizes $u = 8$ and $u = 16$. The same can be said about the place and route time: we experienced run–times of a couple of hours on a AMD–K6–2/300 MHz PC for designs up to 768 and 1024 ($u = 4$) bits, up to a week for the 1024 ($u = 8$ and $u = 16$) bit designs. Different design methods, such as hard–macros for a single unit, would probably improve routing delay and place and route time.

The time–area product shows that designs with 8–bit units are generally most efficient.

Table 9.2 shows our results for a full length modular exponentiation. The purpose of this table is to compare our design to previous propositions. None of the popular public key schemes as described in Chapter 3 requires computing a modular exponentiation with equal size exponent and modulus. A full modular exponentiation with

Table 9.2: Design 1: CLB usage and execution time for a full modular exponentiation

	512 bit		768 bit		1024 bit	
u	C CLBs	T [ms]	C CLBs	T [ms]	C CLBs	T [ms]
4	**2555**	**9.38**	**3745**	**22.71**	**4865**	**40.50**
8	**2094**	**10.13**	**3123**	**23.06**	**4224**	**49.36**
16	**2001**	**11.56**	**2946**	**25.68**	**3786**	**49.99**

an n bit exponent and an m bit modulus is computed in $2(n+2)(m+4)$ clock cycles.

9.2 Design 2

Table 9.3 shows our results of Design 2 in terms of used CLBs (C), clock cycle time (T) and the time–area product (TA).

Table 9.3: Design 2: CLB usage, minimal clock cycle time, and time–area product of modular exponentiation architectures on Xilinx FPGAs

160 bit			256 bit			512 bit		
C CLBs]	T [ns]	TA [CLB·μs]	C [CLBs]	T [ns]	TA [CLB·μs]	C [CLBs]	T [ns]	TA [CLB·μs]
1219	**20.8**	**25.4**	**1818**	**21.3**	**38.7**	**3413**	**20.7**	**70.6**

768 bit			1024 bit		
C CLBs]	T [ns]	TA [CLB·μs]	C [CLBs]	T [ns]	TA [CLB·μs]
5071	**20.1**	**101.9**	**6633**	**21.9**	**145.2**

The time–area products of Table 9.3 are between 50% and 70% larger compared to those of Table 9.1. Design 2, however, is far more efficent than Design 1, as we gain a speed–up of approximately a factor 4 by computing a modular exponentiation with 4 times fewer cycles.

The majority of CLBs is expended in the units, that is 6 CLBs per bit of the modulus. The overhead consists mainly of RAM, DP RAM, counters, registers, and the state machine. Between 300 CLBs for the 160–bit design and 500 CLBs for the 1024–bit design are used for overhead.

The clock cycle time T in Table 9.3 is the access delay $q_i \rightarrow$ *D_out* of the $\tilde{M}$-RAM or $a_i \rightarrow$ *D_out* of the B-RAM plus the delay through the two adders to the registered carry in *S_Reg*, plus the setup time of the flip-flop (see Figure 7.1). We compare this delay to the optimal cycle time calculated by the Xilinx timing analyzer; for the smaller designs (160–512 bits) the delay with optimal routing is 14.7 ns, for the larger designs 15.7 ns. The larger designs were implemented in larger FPGA devices featuring different delay specifications. Otherwise we expected the same cycle times for all designs as the difference between designs lies in the amount of units. The additional routing delay is about 30% above the optimal propagation delay. This is considerably better than the additional routing delay of Design 1. The structure with a RAM block and two adders in series seems to be less of a routing problem than the register–multiplexer–adder structure in Design 1.

Table 9.4: Design2: CLB usage and execution time for a full modular exponentiation

	512 bit		768 bit		1024 bit	
design	C CLBs	T [ms]	C CLBs	T [ms]	C CLBs	T [ms]
2	**3413**	**2.93**	**5071**	**6.25**	**6633**	**11.95**
1	**2555**	**9.38**	**3745**	**22.71**	**4865**	**40.50**

Table 9.4 shows our results for a full length modular exponentiation. A full modular exponentiation with an n bit exponent and an m digit modulus is computed in $2 \cdot (n+2)(m+10)$ clock cycles. For an easy comparison we included the fastest Design 1 in Table 9.4. A speed–up of approximately a factor 3.5 is gained using Design 2.

9.3 Application to RSA

Table 9.5 shows our results from the tables above, applied to RSA. The encryption time is calculated for the F_4 exponent, requiring $2 \cdot 19(m+4)$ clock cycles for Design 1 and $2 \cdot 19(m+10)$ clock cycles if Design 2 is used. Please note that $M = \sum_{i=0}^{m-1}(2^k)^i m_i$ for $k = 1$ in Design 1 and $k = 4$ in Design 2.

Table 9.5: Application to RSA: Encryption

		512 bit		1024 bit	
design	u	C CLBs	T [ms]	C CLBs	T [ms]
1	4	**2555**	**0.35**	**4865**	**0.75**
	8	**2094**	**0.37**	**4224**	**0.91**
	16	**2001**	**0.43**	**3786**	**0.93**
2		**3413**	**0.11**	**6633**	**0.22**

For decryption we apply the Chinese remainder theorem. We either decrypt m bits with an $m/2$ bit architecture serially, or with two $m/2$ bit architectures in parallel. The first approach uses only half as many resources, the later is twice as fast.

Table 9.6: Application to RSA: Decryption

		512 bit $2 \cdot 256$ serial		512 bit $2 \cdot 256$ parallel		1024 bit $2 \cdot 512$ serial		1024 bit $2 \cdot 512$ parallel	
design	u	C CLBs	T [ms]	C CLBs	T [ms]	C CLBs	T [ms]	C CLBs	T [ms]
1	4	**1307**	**4.69**	**2614**	**2.37**	**2555**	**18.78**	**5110**	**10.18**
	8	**1122**	**5.31**	**2244**	**2.56**	**2094**	**20.26**	**4188**	**12.41**
	16	**1110**	**5.82**	**2220**	**2.92**	**2001**	**23.12**	**4002**	**12.52**
2		**1818**	**1.62**	**3636**	**0.79**	**3413**	**5.87**	**6826**	**3.10**

9.4 Application to Algorithms Based on the Discrete Logarithm Problem

Table 9.7 applies our results to encryption and decryption for algorithms based on the descrete logarithm problem. As the exponent is only 160 bits long, computation is about six times faster than for a full 1024 bit modular exponentiation.

Table 9.7: CLB usage and execution time for algorithms based on the DL–problem

		512 bit		768 bit		1024 bit	
design	u	C CLBs	T [ms]	C CLBs	T [ms]	C CLBs	T [ms]
1	4	**2555**	**2.97**	**3745**	**4.77**	**4865**	**6.39**
	8	**2094**	**3.19**	**3123**	**4.85**	**4224**	**7.79**
	16	**2001**	**3.64**	**2946**	**5.4**	**3786**	**7.89**
2		**3413**	**0.93**	**5071**	**1.31**	**6633**	**1.89**

9.5 Application to Elliptic Curves

Section 3.3 states that an elliptic curve projective space addition can be performed in 15 operations, and 12 operations are needed for a doubling. As operations we count multiplications only. Additions, subtractions and shift operations are neglected.

With both our designs we can compute two multiplications in parallel, under the condition that the same operand B is used in both multiplications. For example $B(C + D)$ can be computed in parallel. The operation for adding and doubling are listed in Tables 9.8 and 9.9. As many steps as possible are computed in parallel, including the pre– and post–computations. In an addition 16 operations are executed in series, 8 can be done in parrallel.

For a point doubling we need 14 operations in series and 8 can be done in parallel. In a general point multiplication a series of addings and doublings are executed. Thus

Table 9.8: Operations for a point addition

Common Operand	Multiplication 1		Multiplication 2		Type of Operation
	Operand	Result	Operand	Result	
$2^{2k(m+2)}$	X_1	X_1'	X_2	X_2'	pre-computation
$2^{2k(m+2)}$	Y_1	Y_1'	Y_2	Y_2'	pre-computation
$2^{2k(m+2)}$	Z_1	Z_1'	Z_2	Z_2'	pre-computation
Z_1'	Y_2'	$Z_1'Y_2'$	X_2'	Z_1X_2'	
Z_2'	Y_1'	$Z_2'Y_1'$	X_1'	Z_2X_1'	
Z_2'	Z_1'	$Z_2'Z_1'$			
V'	V'	$(V^2)'$			
$(V^2)'$	T'	$(V^2)'T'$	V'	$(V^3)'$	
$(V^2)'$	Z_2X_1'	$(V^2)'Z_2X_1'$			
$(V^3)'$	$Z_2'Z_1'$	Z_3'	$Z_2'Y_1'$	$(V^3)'Z_2'Y_1'$	
U'	U'	$(U^2)'$			
$(U^2)'$	$Z_2'Z_1'$	A'			
$(V^2)'Z_2X_1' - A$	U'	Y_3'			
V'	A'	X_3'			
1	X_3'	X_3	Y_3'	Y_3	post-computation
1	Z_3'	Z_3			post-computation

the pre-computations have to be done only once before the first operation, and the post-computations after the last operation. The total amount of clock cycles for a normal addition is therefore $11 \cdot 2(m+4)$ if Design 1 is used, and $11 \cdot 2(m+10)$ cycles with Design 2. A doubling without pre– and post–computations is executed in $8 \cdot 2(m+4)$ cycles (Design 1), or $8 \cdot 2(m+10)$ cycles (Design 2). We assume that all values are stored externally. The subtractions, additions and shift operations could be done internally with almost no additonal resource requirements. As we process only u bit (Design 1) or 4 bit (Design 2) at a time, addition or subtraction could be done serially. In the doubling operation we have multiplications by eight. Thus the

Table 9.9: Operations for a Doubling

Common	Multiplication 1		Multiplication 2		Type of
Operand	Operand	Result	Operand	Result	Operation
$2^{2k(m+2)}$	X_1	X_1'	X_2	X_2'	pre-computation
$2^{2k(m+2)}$	Y_1	Y_1'	Y_2	Y_2'	pre-computation
$2^{2k(m+2)}$	Z_1	Z_1'	Z_2	Z_2'	pre-computation
$2^{2k(m+2)}$	a	a'			Pre-computation
Z_1'	Y_1'	S'	Z_1'	$(Z_1^2)'$	
S'	S'	$(S^2)'$	Y_1'	E'	
X_1'	X_1'	$(X_1^2)'$	E'	F'	
$(Z_1^2)'$	a'	W'			
W'	W'	$(W^2)'$			
E'	E'	$(E^2)'$			
$4F' - H$	W'	Y_3'			
S'	$(S^2)'$	$(S^3)'$	$2H'$	X_3'	
1	X_3'	X_3	Y_3'	Y_3	post-computation
1	Z_3'	Z_3			post-computation

operands are getting larger than specified in Chapter 4 ($A, B > 2M$). Therefor an at least 4 bit larger architecture has to be chosen. The resulting additional resource requirements and clock cycles however can be neglected.

Table 9.10 shows the total execution time for the basic operations in elliptic curve cryptosystems. Table 9.11 shows the average time needed for a general point multiplication in an elliptic curve cryptosystems. The modulus and the operands are 160 bits long. Thus 160 doublings and an average of 80 additions are computed. It should be noted that advanced point multiplication algorithms (see, e.g., [37]) can reduce the number of point additions considerably at the cost of additional memory for pre–computed points.

Table 9.10: Application to Elliptic Curves: Execution time for point addition and doubling (160 bits)

		Addition	Doubling
design	u	T [μs]	T [μs]
1	4	**62**	**39**
	8	**71**	**45**
	16	**76**	**48**
2		**23**	**15**

Table 9.11: Application to Elliptic Curves: Execution time for a general point multiplication (160 bits)

design	u	T [ms]
1	4	**11.3**
	8	**12.8**
	16	**13.8**
2		**4.2**

Chapter 10

Comparison and Outlook

We compare our fastest RSA 512/1024 bit designs of Table 9.6 to the fastest soft- and hardware solutions we found in the literature [33, 26, 37]. Our 0.8 ms decryption time is about 11 times faster than the 512 bit software implementation (9.1 ms) on a 150MHz Alpha [26]. The fastest 1024 bit software implementation [37] of 43.3 ms running on a PPro–200 based PC is about 14 times slower than our best result (3.1 ms).

Most reported hardware implementations of modular arithmetic are somewhat dated, making a fair comparison difficult. It is nevertheless interesting to look at previously reported performances. The fastest reported FPGA design [33] (1.7 ms for a 512 bit modulus and 5.2 ms for a 970 bit modulus) is a factor 2.1/1.8 slower than ours (2.8 ms for a 970 bit modulus). It is possible, however, that their solution upgraded to currently available FPGA technology, would reach similar speeds. A drawback of the solution in [33] is, however, that the binary representation of the modulus is hardwired into the logic representation so that the architecture has to be reconfigured with every new modulus. The user of such an implementation needs to own the full development tools for synthesis, placing and routing of FPGAs, if RSA with different moduli should be executed. Our design stores the modulus, the exponent and the pre–computation factor in registers and RAM. A second advantage of our design is that it is implemented into one device instead of a matrix of 16 devices. Using currently available FPGA technology the design [33] would probably also fit in a single device.

The fastest ASIC solution was presented in [16]. A 1024–bit modular exponentiation with a 1024–bit exponent is performed in an average of 10 ms (50% bits equal to "1") without applying the Chinese remainder theorem. Our Design 2 processes the same computation in 11.9 ms, where average and worst case computation times are the same.

To improve our design in terms of speed the following conclusions can be drawn.

1. Choice of radix $r = 2^k$: We believe that the radix $r = 2^4 = 16$ chosen for Design 2 is the optimal choice for optimizing the time–area product in a Xilinx XC4000 FPGA implementation. Smaller radixes are simpler to implement at the drawback of more clock cycles. Larger radixes result in very large resource requirements for the computation of the $2^k - 1$ multiples of the operand B.

2. To fully utilize each clock cycle we compute squaring operations and multiplications in parallel. Fully utilize is not quite correct, though, as we have to compute only an average of $n/2$ multiplications. We could speed-up our design by a factor 1.33 on average if we use Algorithm 4.1 and compute two modular exponentiations in parallel. The worst case timing improvement is zero however. The drawback of this approach is that two operands B and their multiples have to be stored. The units would need two 32×4 bit RAM blocks instead of two 16×4 bit RAM blocks and two additional registers. Thus the additional resource requirement is one CLB per computed bit of the modulus.

3. Exponent: In Section 4.1 the l–ary method was discussed. This method is particularly useful if we choose an approach as discussed in the last paragraph. For $l = 2$ we have to store only two additional values X^2 and X^3. The worst

case execution time is improved by a factor 2, the average time by a factor 1.5.

4. Implementation of our designs in devices from other FPGA vendors. We might be able to run the designs at faster clock frequencies using different devices.

An architecture as proposed in (2) and (3) would barely fit into the largest available device of the Xilinx XC4000 family. The combined resulting speed-up is a factor two compared to the tables in Chapter 9.

Appendix A

Test Bench Sample

```
-- VHDL Architecture TB_example_hr.vhd
-- Tests the functionality of the monXb_r4.vhd designs (X=160,256,512,768,1024).
-- Test_vectors: M = '1...1', 2M = '2...2', 3M = '3...3' ... 15M = 'F...F'
--               E = 1101101010101010101 (MSB-LSB)
--               Pre-computation factor = '1','2','2','3','3'... (digit_0, digit_1 ...)
--               X = '2','3','4' ... (digit_0, digit_1 ...)
--
--Created:
--        by - Thomas Blum
--        at - 02/03/99


Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_textio.all;
use  work.array_types.all;


entity E is
GENERIC(
width : positive := w_idth;   -- unit width
slices : positive := s_lices;  -- number of units
length : positive := l_ength -- length of modulus
    );
end E;


ARCHITECTURE A OF E IS

-- Architecture declarations
```

```
     CONSTANT clk_prd_f : time := 20 ns;
     CONSTANT clk_prd_s : time := 40 ns;
     CONSTANT del_prd   : time := 5 ns;

SIGNAL modulus_in : std_logic_vector(7 DOWNTO 0);
SIGNAL ttn : std_logic_vector(7 DOWNTO 0);
SIGNAL data_in : std_logic_vector(3 DOWNTO 0);
SIGNAL CLOCK_F : std_logic;
SIGNAL CLOCK_S : std_logic;
SIGNAL reset : std_logic;
SIGNAL exponent_in : std_logic_vector(15 DOWNTO 0);
SIGNAL encrypt_in : std_logic;
SIGNAL load_mod_in : std_logic;
SIGNAL load_exp_in : std_logic;
SIGNAL load_ttn_in : std_logic;
SIGNAL result_out : std_logic_vector(3 DOWNTO 0);
SIGNAL zero_out : std_logic_vector(3 DOWNTO 0);

--internal clock signal

SIGNAL iclk_f : std_logic;
SIGNAL iclk_s : std_logic;

--clock procedure

        PROCEDURE wait_clock(CONSTANT clk_ticks:integer) IS
        VARIABLE i : integer := 0;
        BEGIN
                FOR i IN 1 TO clk_ticks*2 LOOP
                        WAIT UNTIL iclk_s'EVENT;
                END LOOP;
        END wait_clock;

PROCEDURE wait_clock_f(CONSTANT clk_ticks:integer) IS
        VARIABLE i : integer := 0;
        BEGIN
                FOR i IN 1 TO clk_ticks LOOP
                        WAIT UNTIL iclk_s'EVENT;
                END LOOP;
        END wait_clock_f;


--test component declaration

        component mon160b_r4
PORT(
modulus_in : IN std_logic_vector(7 DOWNTO 0);
ttn : IN std_logic_vector(7 DOWNTO 0);
data_in : IN std_logic_vector(3 DOWNTO 0);
CLOCK_F : IN std_logic;
CLOCK_S : IN std_logic;
reset : IN std_logic;
exponent_in : IN std_logic_vector(15 DOWNTO 0);
encrypt_in : IN std_logic;
load_mod_in : IN std_logic;
load_exp_in : IN std_logic;
load_ttn_in : IN std_logic;
result_out : OUT std_logic_vector(3 DOWNTO 0);
zero_out : OUT std_logic_vector(3 DOWNTO 0)
);

END COMPONENT;

BEGIN

--test component instantiation

UUT: mon160b_r4
PORT MAP(
modulus_in      =>      modulus_in,
ttn             =>      ttn,
data_in         =>      data_in,
CLOCK_F         =>      CLOCK_F,
CLOCK_S         =>      CLOCK_S,
reset           =>      reset,
exponent_in     =>      exponent_in,
encrypt_in      =>      encrypt_in,
load_mod_in     =>      load_mod_in,
load_exp_in     =>      load_exp_in,
load_ttn_in     =>      load_ttn_in,
result_out      =>      result_out,
zero_out        =>      zero_out);

--testbench procedure

flow_process: PROCESS

-- Process declarations

variable i : integer := 0;


BEGIN

--*****************************************************
-- initialize signals and reset: '0' -> '1' -> '0'
--*****************************************************
```

```
reset   <= '0';
data_in   <= "0000";
ttn   <= "00000000";
exponent_in <= "0000000000000000";
load_mod_in <= '0';
load_exp_in <= '0';
load_ttn_in <= '0';
encrypt_in <= '0';

modulus_in  <= "00000000";
load_mod_in <= '0';
wait_clock_f(1);
wait for del_prd;
reset <= '1';
wait_clock_f(1);
wait for del_prd;
reset   <= '0';
wait_clock_f(1);
wait for del_prd;

--****************************************************
-- load modulus: This is painful but has to be done
--               only once for a given system
--****************************************************

load_mod_in <= '1';
wait_clock(1);
wait for del_prd;
modulus_in  <= "00010001";   -- 1M (units 0 ... m)
load_mod_in <= '0';
wait_clock(length/(2*width));
wait for del_prd;

load_mod_in <= '1';
wait_clock(1);
wait for del_prd;
modulus_in  <= "00100010";   -- 2M (units 0 ... m)
load_mod_in <= '0';
wait_clock(length/(2*width));
wait for del_prd;

load_mod_in <= '1';
wait_clock(1);
wait for del_prd;
modulus_in  <= "00110011";   -- 3M (units 0 ... m)
load_mod_in <= '0';
wait_clock(length/(2*width));
wait for del_prd;
```

```
load_mod_in <= '1';
wait_clock(1);
wait for del_prd;
modulus_in  <= "01000100";   -- 4M (units 0 ... m)
load_mod_in <= '0';
wait_clock(length/(2*width));
wait for del_prd;

load_mod_in <= '1';
wait_clock(1);
wait for del_prd;
modulus_in  <= "01010101";   -- 5M (units 0 ... m)
load_mod_in <= '0';
wait_clock(length/(2*width));
wait for del_prd;

load_mod_in <= '1';
wait_clock(1);
wait for del_prd;
modulus_in  <= "01100110";   -- 6M (units 0 ... m)
load_mod_in <= '0';
wait_clock(length/(2*width));
wait for del_prd;

load_mod_in <= '1';
wait_clock(1);
wait for del_prd;
modulus_in  <= "01110111";   -- 7M (units 0 ... m)
load_mod_in <= '0';
wait_clock(length/(2*width));
wait for del_prd;

load_mod_in <= '1';
wait_clock(1);
wait for del_prd;
modulus_in  <= "10001000";   -- 8M (units 0 ... m)
load_mod_in <= '0';
wait_clock(length/(2*width));
wait for del_prd;

load_mod_in <= '1';
wait_clock(1);
wait for del_prd;
modulus_in  <= "10011001";   -- 9M (units 0 ... m)
load_mod_in <= '0';
wait_clock(length/(2*width));
wait for del_prd;

load_mod_in <= '1';
```

```
wait_clock(1);
wait for del_prd;
modulus_in  <= "10101010";   -- 10M (units 0 ... m)
load_mod_in <= '0';
wait_clock(length/(2*width));
wait for del_prd;

load_mod_in <= '1';
wait_clock(1);
wait for del_prd;
modulus_in  <= "10111011";   -- 11M (units 0 ... m)
load_mod_in <= '0';
wait_clock(length/(2*width));
wait for del_prd;

load_mod_in <= '1';
wait_clock(1);
wait for del_prd;
modulus_in  <= "11001100";   -- 12M (units 0 ... m)
load_mod_in <= '0';
wait_clock(length/(2*width));
wait for del_prd;

load_mod_in <= '1';
wait_clock(1);
wait for del_prd;
modulus_in  <= "11011101";   -- 13M (units 0 ... m)
load_mod_in <= '0';
wait_clock(length/(2*width));
wait for del_prd;
load_mod_in <= '1';
wait_clock(1);
wait for del_prd;
modulus_in  <= "11101110";   -- 14M (units 0 ... m)
load_mod_in <= '0';
wait_clock(length/(2*width));
wait for del_prd;

load_mod_in <= '1';
wait_clock(1);
wait for del_prd;
modulus_in  <= "11111111";   -- 15M (units 0 ... m)
load_mod_in <= '0';
wait_clock(length/(2*width));
wait for del_prd;
wait_clock(1);
wait for del_prd;
modulus_in  <= "00000000"; --end load modulus
wait_clock(1);
```

```
wait for del_prd;

--**************************************************
-- load exponent
--**************************************************

load_exp_in <= ’1’;    --prepare load exponent: load_input->hi
wait_clock(1);
wait for del_prd;
exponent_in  <= "0000000000010011";     --counter_value / nuber of bits in exponent
wait_clock(1);
wait for del_prd;
exponent_in  <= "1101010101010101";     --16 bit exponent
wait_clock(1);
wait for del_prd;
exponent_in  <= "0000000000000110";     --16 bit exponent
load_exp_in <= ’0’; --load_input->low
wait_clock(1);
wait for del_prd;
exponent_in  <= "0000000000000000";     --16 bit exponent
wait_clock(1);
wait for del_prd;

--**************************************************
-- load ttn
--**************************************************

load_ttn_in <= ’1’;
wait_clock(1);
wait for del_prd;
ttn   <= "00100001"; -- ls-digits of precomputation factor: ’1’, ’1’
wait_clock(1);
wait for del_prd;

for i in 1 to (length/(2*width)) loop
ttn   <= ttn + "00010001";   -- add ’1’, ’1’ to next digit
wait_clock(1);
wait for del_prd;
end loop;

load_ttn_in <= ’0’;
wait_clock(1);
wait for del_prd;
ttn   <= "00000000";
wait_clock(1);
wait for del_prd;
```

```
--*****************************************************
-- start encryption
--*****************************************************

encrypt_in <= '1';
wait_clock(2);
wait for del_prd;
data_in <= "0010";     -- ls-digit of X = '2'
wait_clock(1);
wait for del_prd;

for i in 0 to (length/(width)) loop
data_in   <= data_in + "0001";   - add '1' to next digit of X
wait_clock(1);
wait for del_prd;
end loop;

data_in <= "0000";
wait_clock(100000);

   END PROCESS flow_process;

clock_gen_f : PROCESS
BEGIN
   iclk_f <= '1';
   WAIT FOR clk_prd_f/2;
   iclk_f <= '0';
   WAIT FOR clk_prd_f/2;
END PROCESS clock_gen_f;
CLOCK_f <= iclk_f;

clock_gen_s : PROCESS
BEGIN
   iclk_s <= '1';
   WAIT FOR clk_prd_s/2;
   iclk_s <= '0';
   WAIT FOR clk_prd_s/2;
END PROCESS clock_gen_s;
CLOCK_S <= iclk_s;

END A;

--architecture configuration

configuration CFG_TB_mont_BEHAVIORAL of E is
        for A
        end for;
end CFG_TB_mont_BEHAVIORAL;
```

Appendix B

Synosys Script

```
/*  Sample Script for Synopsys to Xilinx Using   */
/*  FPGA Compiler targeting an XC4000XL device    */
/*  Set the name of the design"s top-level        */

TOP = mon160b_r4
F1  = package_r4
F2  = module_pack
F3  = unit4b_r4ram
F4  = f_unit4b_r4ram
F5  = e_unit4b_r4ram
F6  = exp_ram_64x16s
F7  = mu_block_ram_40x4dp
F8  = sq_block_ram_40x4dp
F9  = state_mach
F10 = ttn_ram_20x8s

/*  Set the name of the design"s LOGIBLOX        */

F11  = reg_2b
F12  = reg_3b
F13  = reg_4b
F14  = reg_5b
F15  = reg_6b
F16  = reg_8b
F17  = reg_10b
F18  = reg_16b
F19  = shift16bit
F20  = clk_div_a
F21  = count_4b_15m
F22  = count_5b_20m
F23  = count_6b_u
F24  = count_6b_42m
F25  = count_10b_u
```

```
F26  = ram4x4
F27  = ram5x4
F28  = ram32x8s
F29  = ram64x16s
F30  = ram64x4dp

designer = "Thomas Blum"
company  = "WPI Crypto Group"
part     = "4085XLBG432-09"

/* Read the LOGIBLOX.       */

read -format edif "WORK/" + F11 + ".edn"
read -format edif "WORK/" + F12 + ".edn"
read -format edif "WORK/" + F13 + ".edn"
read -format edif "WORK/" + F14 + ".edn"
read -format edif "WORK/" + F15 + ".edn"
read -format edif "WORK/" + F16 + ".edn"
read -format edif "WORK/" + F17 + ".edn"
read -format edif "WORK/" + F18 + ".edn"
read -format edif "WORK/" + F19 + ".edn"
read -format edif "WORK/" + F20 + ".edn"
read -format edif "WORK/" + F21 + ".edn"
read -format edif "WORK/" + F22 + ".edn"
read -format edif "WORK/" + F23 + ".edn"
read -format edif "WORK/" + F24 + ".edn"
read -format edif "WORK/" + F25 + ".edn"
read -format edif "WORK/" + F26 + ".edn"
read -format edif "WORK/" + F27 + ".edn"
read -format edif "WORK/" + F28 + ".edn"
read -format edif "WORK/" + F29 + ".edn"
read -format edif "WORK/" + F30 + ".edn"

/* Analyze and Elaborate the design file.       */

analyze -format vhdl "sim_rtl/" + F1 + ".vhd"
analyze -format vhdl "sim_rtl/" + F2 + ".vhd"
analyze -format vhdl "sim_rtl/" + F3 + ".vhd"
analyze -format vhdl "sim_rtl/" + F4 + ".vhd"
analyze -format vhdl "sim_rtl/" + F5 + ".vhd"
analyze -format vhdl "sim_rtl/" + F6 + ".vhd"
analyze -format vhdl "sim_rtl/" + F7 + ".vhd"
analyze -format vhdl "sim_rtl/" + F8 + ".vhd"
analyze -format vhdl "sim_rtl/" + F9 + ".vhd"
analyze -format vhdl "sim_rtl/" + F10 + ".vhd"
analyze -format vhdl "sim_rtl/" + TOP + ".vhd"

elaborate TOP
```

```
/*Set the current design to unit4b_r4ram level.        */

        current_design F3

/* Don't touch the logiblox*/

set_dont_touch ("b_ram_inst")
set_dont_touch ("m_ram_inst")
set_dont_touch ("a_i_reg")
set_dont_touch ("q_i_reg")
set_dont_touch ("res_reg_inst")
set_dont_touch ("control_reg")
set_dont_touch ("b_in_reg")
set_dont_touch ("b_mult_reg")
set_dont_touch ("result_reg")

/*Set the current design to the f_unit4b_r4ram level.        */

        current_design F4

/* Don't touch the logiblox*/

set_dont_touch ("m_ram_inst")
set_dont_touch ("a_i_reg")
set_dont_touch ("q_i_reg")
set_dont_touch ("res_reg_inst")
set_dont_touch ("control_reg")
set_dont_touch ("result_reg")

/*Set the current design to the e_unit4b_r4ram level.        */

        current_design F5

/* Don't touch the logiblox*/

set_dont_touch ("b_ram_inst")
set_dont_touch ("res_reg_inst")
set_dont_touch ("res_reg_2_inst")
set_dont_touch ("control_reg")
set_dont_touch ("b_in_reg")
set_dont_touch ("b_mult_reg")
set_dont_touch ("result_reg")

/*Set the current design to the exp_ram_64x16s level.        */

        current_design F6

/* Don't touch the logiblox*/
```

```
set_dont_touch ("ram_inst")
set_dont_touch ("cnt_ram_wr")
set_dont_touch ("cnt_ram_rd")
set_dont_touch ("count_val_reg")
set_dont_touch ("shift_reg")

/*Set the current design to the mu_block_ram_40x4dp level.        */

        current_design F7

/* Don't touch the logiblox*/

set_dont_touch ("dp_ram_y")
set_dont_touch ("count_wrt")
set_dont_touch ("count_rd")
set_dont_touch ("data_in_reg_f")
set_dont_touch ("data_in_reg_s")
set_dont_touch ("data_r")

/*Set the current design to the sq_block_ram_40x4dp level.        */

        current_design F8

/* Don't touch the logiblox*/

set_dont_touch ("dp_ram_y")
set_dont_touch ("count_wrt")
set_dont_touch ("count_rd")
set_dont_touch ("data_in_reg")
set_dont_touch ("data_r")

/*Set the current design to the state_mach level.        */

        current_design F9

/* Don't touch the logiblox*/

set_dont_touch ("cnt_a")

/*Set the current design to the ttn_ram_20x8s level.        */

        current_design F10

/* Don't touch the logiblox*/

set_dont_touch ("ram_inst")
set_dont_touch ("cnt_ram")
set_dont_touch ("output_reg_even")
set_dont_touch ("output_reg_odd")
```

```
/*Set the current design to the top level.        */

        current_design TOP

/* Don't touch the logiblox*/

set_dont_touch ("mod_even")
set_dont_touch ("mod_odd")
set_dont_touch ("ttn_reg")
set_dont_touch ("reg_exp")
set_dont_touch ("clk_divider")

remove_constraint -all

/* Uniquify the design and reset the schematic */

uniquify
create_schematic -size infinite -gen_database

/* include timming and area constraints */

remove_constraint -all
remove_clock -all
create_clock -period 20 -waveform {0 10} CLOCK_F
create_clock -period 40 -waveform {20 40} CLOCK_S
group_path -critical_range 10000 -default
set_input_delay 0 -clock CLOCK_F { all_inputs()}
set_output_delay 0 -clock CLOCK_F { all_outputs()}
set_input_delay 0 -clock CLOCK_S { all_inputs()}
set_output_delay 0 -clock CLOCK_S { all_outputs()}
set_operating_conditions WCCOM

/* Indicate which ports are pads.    */

set_port_is_pad "*"
set_pad_type -no_clock all_inputs()
set_pad_type -clock CLOCK_F
set_pad_type -clock CLOCK_S
set_pad_type -slewrate LOW all_outputs()
insert_pads

/* link */

link

/* Synthesize the design.*/
```

```
compile -boundary_optimization -map_effort high

/* Write the design report files.                  */

report_fpga > "reports/" + TOP + ".fpga"
report_timing > "reports/" + TOP + ".timing"
report_constraint -verbose > "reports/" + TOP + ".cnst"

/* Write out an intermediate DB file to save state */
write -format db -hierarchy -output "db/" + TOP + "_compiled.db"

/* Replace CLBs and IOBs primitives (XC4000E/EX/XL only)    */
replace_fpga

/* Set the part type for the output netlist.      */
set_attribute TOP "part" -type string part

/* Write out the intermediate DB file to save state*/
write -format db -hierarchy -output "db/" + TOP + ".db"

/* Write out the timing constraints                */
ungroup -all -flatten
write_script > "dc/" + TOP + ".dc"

/* Save design in XNF format as <design>.sxnf      */
write -format xnf -hierarchy -output "sxnf/" + TOP + ".sxnf"

/* XILINX primitive to convert Synopsys design constraints to Xilinx format*/
sh /usr/local/xilinx/bin/sol/dc2ncf "dc/" + TOP + ".dc"
```

Appendix C

Simulation Results

In this chapter the simulation results are shown of Design 2, with a modulus of 160 bits. For a more detailed description of the data flow, please refer to Sections 7.2, 7.3, and 7.4.

C.1 Processing Elements

The following sequence of three figures shows the pre place-and-route simulation results for processing element_3. Figure C.1 shows the loading of the pre–computation factor into B and the calculation of its multiples. Figure C.2 shows the first cycles of the two modular multiplications. In Figure C.3 finally, the last cycles of these operations are shown, the storing of the first multiplication result in *B-reg*, and the calculation of the multiples of B.

The signals shown in the simulation sequence are as follows:

Figure: C.1

`CLOCK_F`	$\Rightarrow$	Fast clock signal, system clock.
`CLOCK_S`	$\Rightarrow$	Slow clock signal`(CLOCK_F/2)`.
`control`	$\Rightarrow$	Current state encoded.
`b_in`	$\Rightarrow$	Operand B.
`control_in`	$\Rightarrow$	`control` input for unit$_3$.

`b`	$\Rightarrow$	Input B registered.
`a_i`	$\Rightarrow$	Operand A.
`write_ram_b`	$\Rightarrow$	Write enable signal for B-RAM.
`b_m_reg`	$\Rightarrow$	Data input of B-RAM (`b_p_mult`registered).
`b_p_mult`	$\Rightarrow$	Temporarily result of B-multiplication.
`c_b_in`	$\Rightarrow$	Carry input B-multiplication.
`c_b_out`	$\Rightarrow$	Carry output B-multiplication.

Figure: C.2

`carry_in`	$\Rightarrow$	Carry input from unit to the right.
`q_in`	$\Rightarrow$	Quotient Q.
`res_in`	$\Rightarrow$	Result S of an iteration from unit to the left.
`modulus`	$\Rightarrow$	Multiple of M according to q_i ($1M = 1$).
`a_t_b`	$\Rightarrow$	Multiple of B according to a_i ($1B = 2$).
`mod_p_b`	$\Rightarrow$	`modulus` plus `a_t_b` plus `carry_0`.
`mod_p_b_p_s`	$\Rightarrow$	`mod_p_b` plus `res_in` plus `carry_1`.
`carry_out`	$\Rightarrow$	Bits 4 and 5 of the result `mod_p_b_p_s` registered.
`res_out`	$\Rightarrow$	Bits 3 to 0 of the result `mod_p_b_p_s` registered.
`res_reg`	$\Rightarrow$	Bits 5 to 0 of the result `mod_p_b_p_s` registered.

Figure: C.3

`result_in`	$\Rightarrow$	Result of a modular multiplication from unit to the left.
`result_out`	$\Rightarrow$	Result of a modular multiplication `mod_p_b_p_s` or `result_in` to unit to the right .
`load_b`	$\Rightarrow$	Write enable signal for B-reg.
`b`	$\Rightarrow$	Result of squaring stored in B-reg.

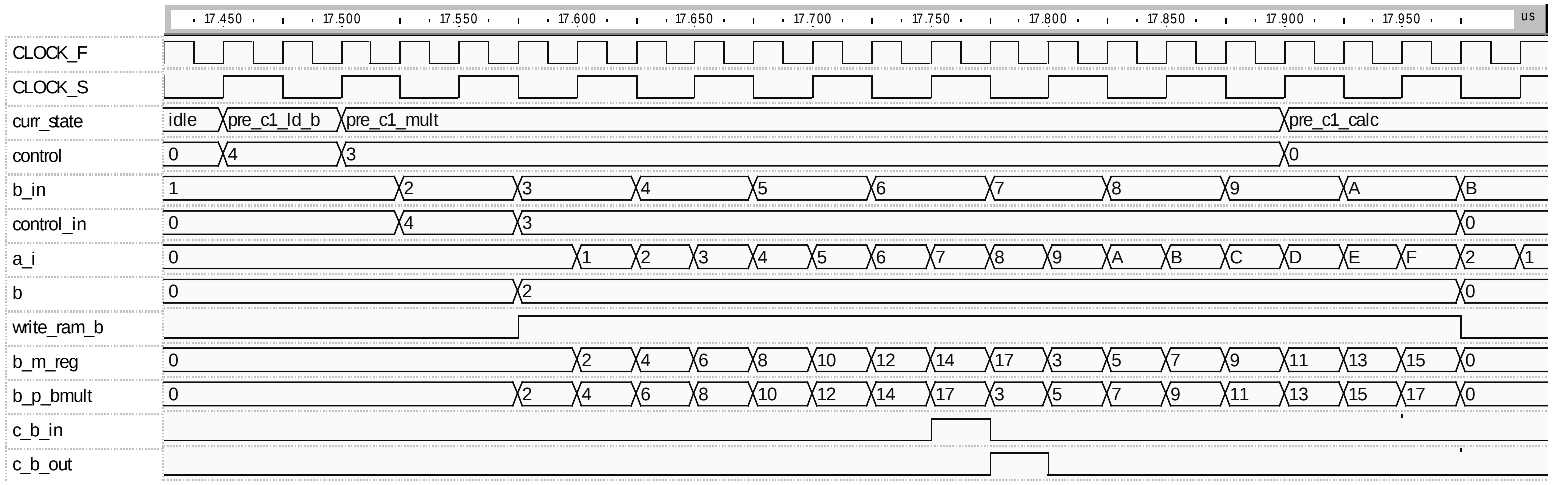


Figure C.1: Processing Element: Loading of the pre-computation factor and calculation of its multiples

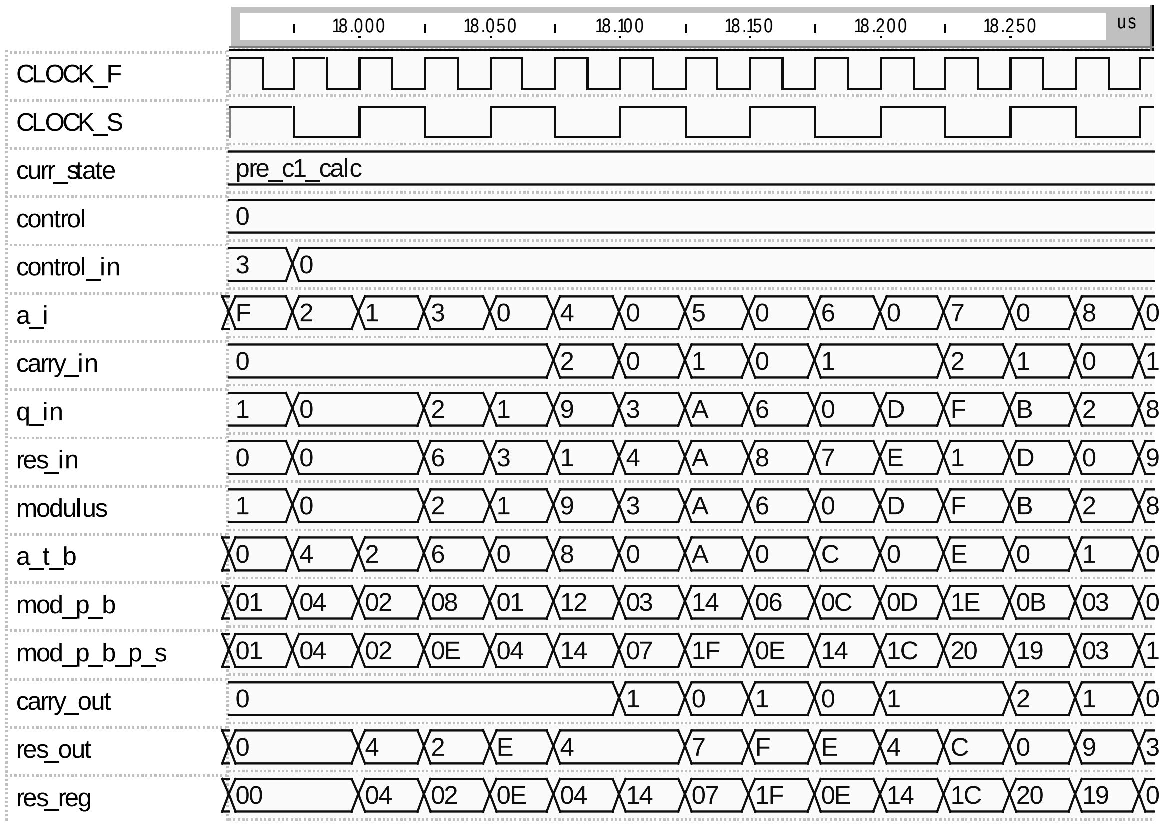


Figure C.2: Processing Element: Computation of two modular multiplications (first cycles)

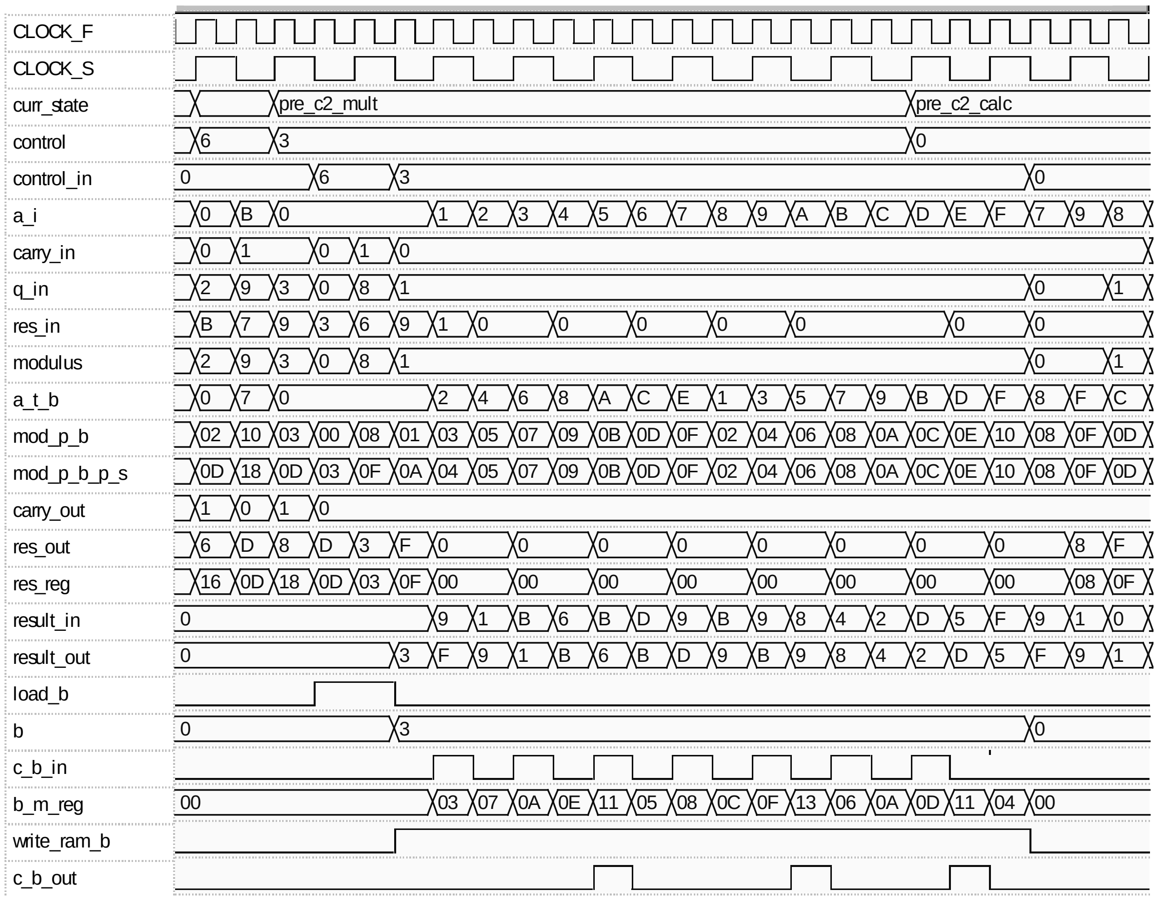


Figure C.3: Processing Element: Computation of two modular multiplications (last cycles).

C.2 Systolic Array

The following sequence of three figures shows the pre place-and-route simulation results for the systolic array. Figure C.4 shows the loading of the pre–computation factor into B, the calculation of its multiples, and the first cycles of the two modular multiplications. Figure C.5 shows the last cycles of these operations in units 0,1 and 2, the storing of the first multiplication result in *B-reg*, and calculation of the multiples of B. In Figure C.6 finally, the end of the modular multiplications in the units 41,42 and 43 is shown.

The signals shown in the simulation sequence are as follows:

Figure: C.4

`CLOCK_F`	$\Rightarrow$	Fast clock signal, system clock.
`CLOCK_S`	$\Rightarrow$	Slow clock signal(`CLOCK_F/2`).
`b_even`	$\Rightarrow$	bus B to even numbered units.
`b_odd`	$\Rightarrow$	bus B to odd numbered units.
`control_in`	$\Rightarrow$	Inputs control of units 0 to 2 and 42,43.
`a_in`	$\Rightarrow$	Inputs operand A of units 0 to 2 and 42,43.
`res_out`	$\Rightarrow$	Result of unit$_1$ (reused as q_i in next iteration).
`q_in`	$\Rightarrow$	Inputs quotient Q of units 0 to 2 and 42,43.

Figures: C.5 and C.6

`result_out`	$\Rightarrow$	Modular multiplication results of units 0 to 3 and 41,42,43 pumped trough the systolic array.

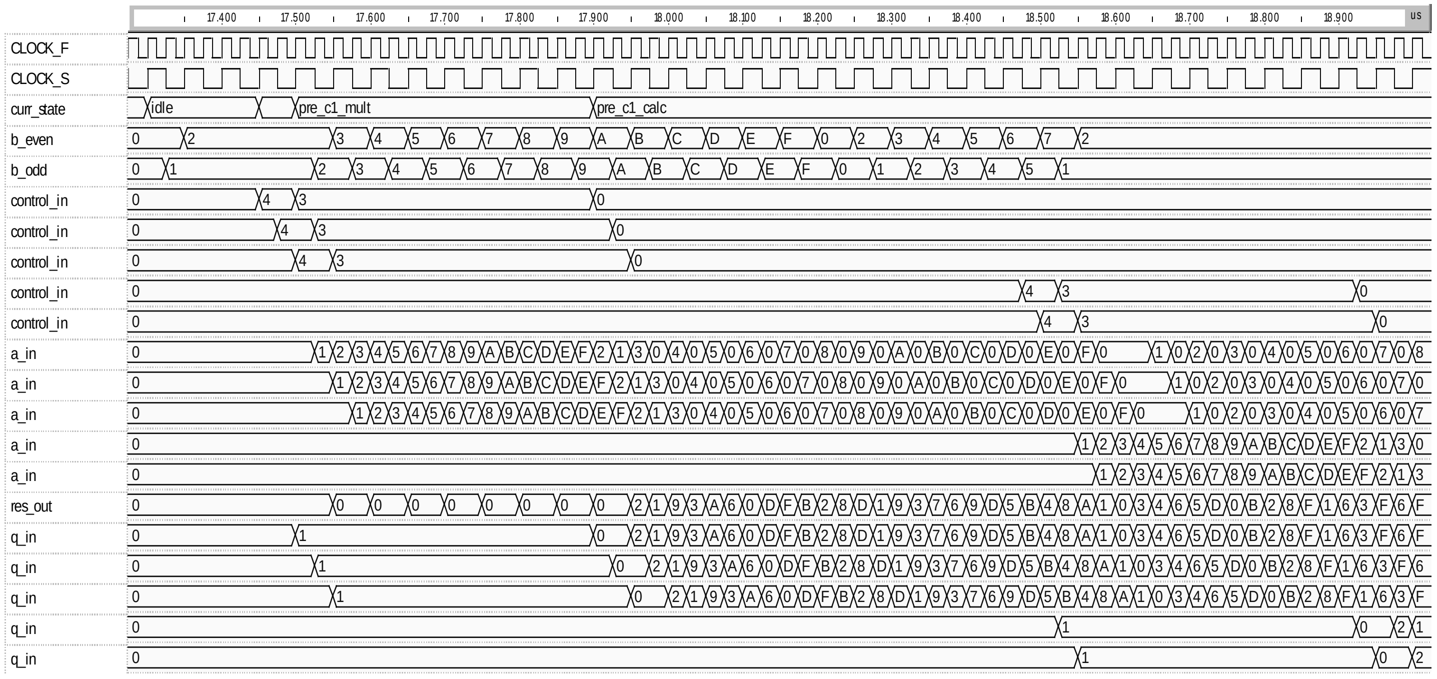


Figure C.4: Systolic Array: Beginning of the pre-computation

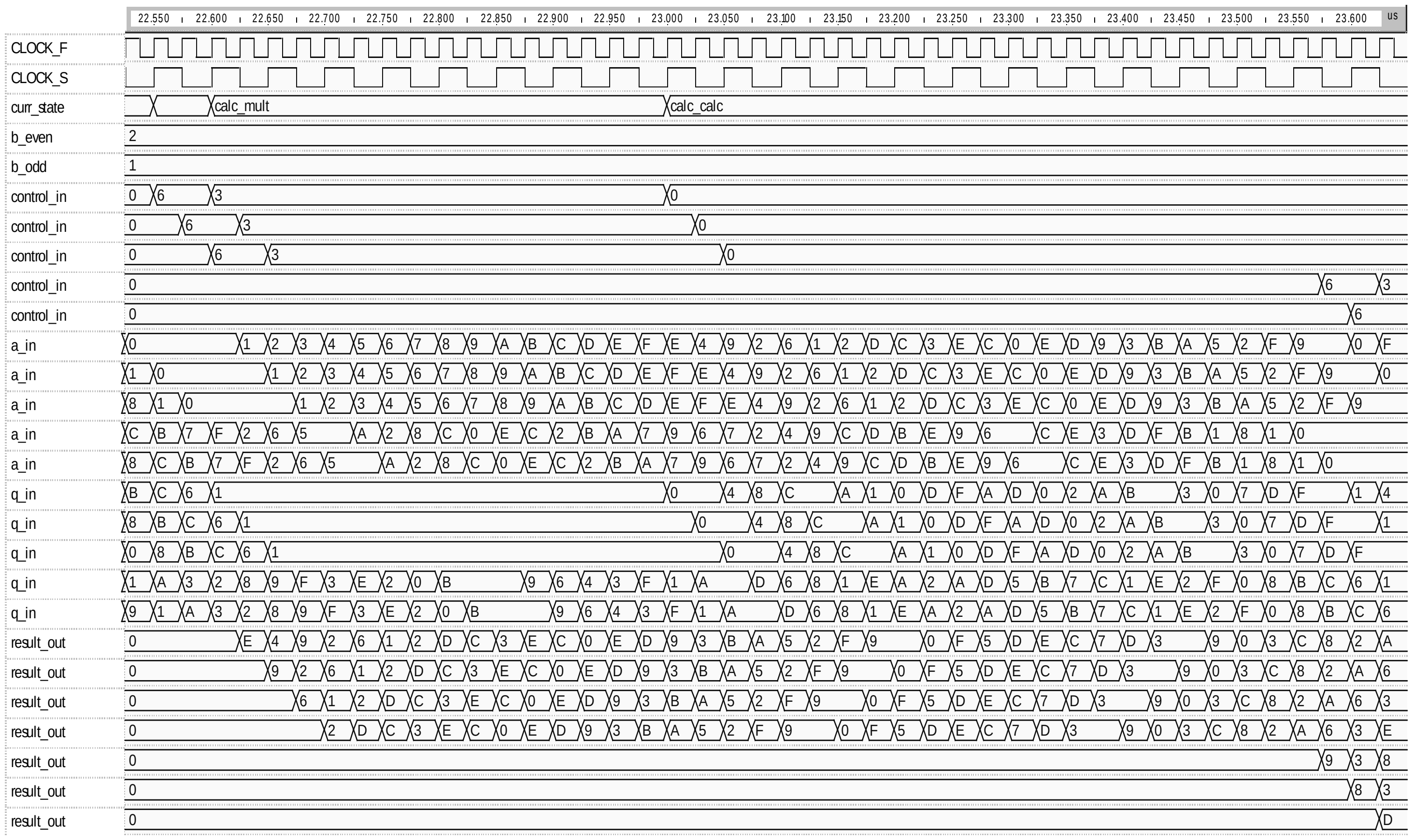


Figure C.5: Systolic Array: End of the pre-computation in units 0,1,2

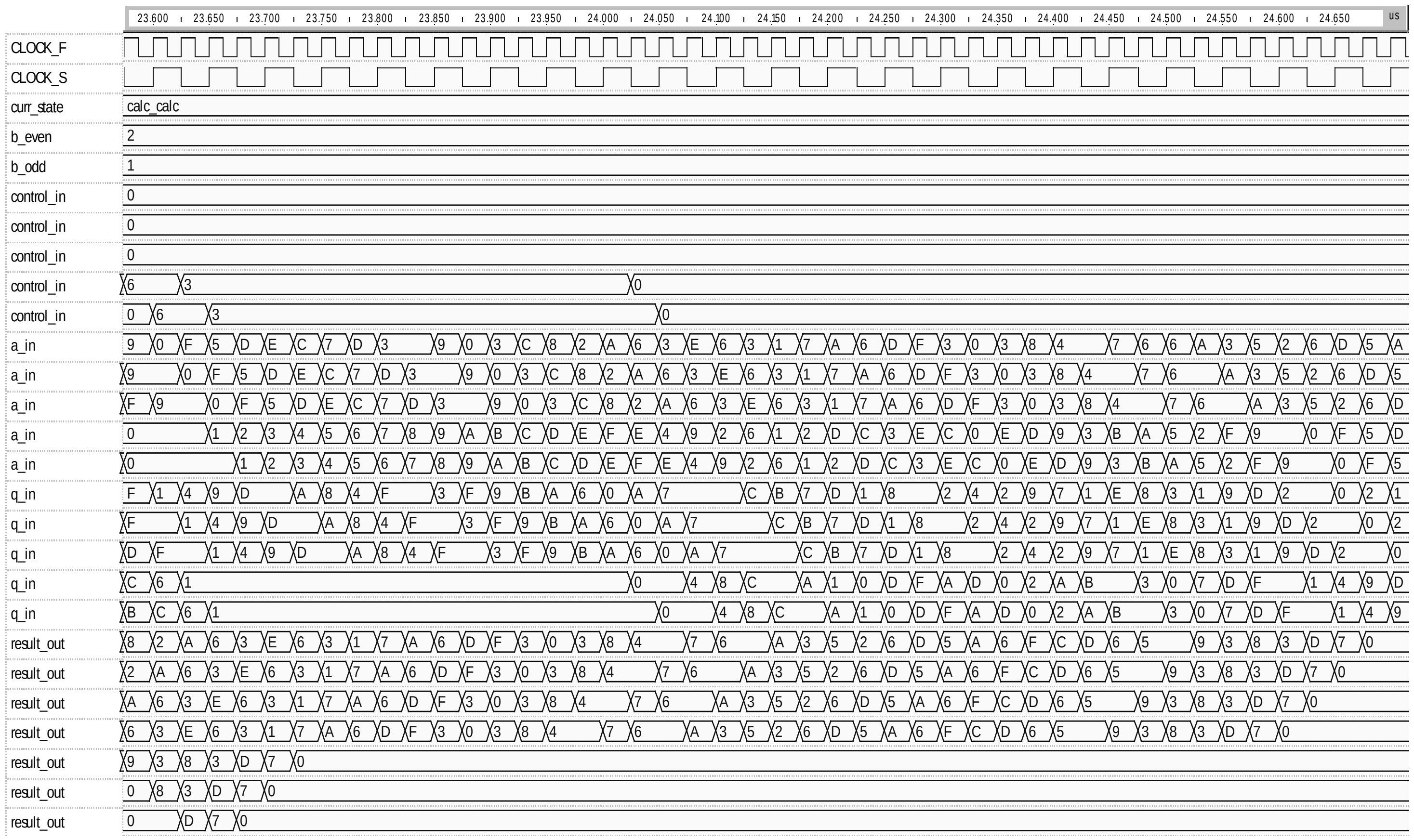


Figure C.6: Systolic Array: End of the pre-computation in units 41,42,43

C.3 Modular Exponentiation

The following sequence of three figures shows the pre place-and-route simulation results of the modular exponentiation. Figures C.7 and C.8 show the pre-computation. In Figures C.9 an overview of a 19-bit modular exponentiation is given.

The signals shown in the simulation sequence are as follows:

Figure: C.7

`CLOCK_F`	$\Rightarrow$	Fast clock signal, system clock.
`CLOCK_S`	$\Rightarrow$	Slow clock signal(`CLOCK_F/2`).
`data_in`	$\Rightarrow$	X value to encrypt/decrypt from input.
`b_even`	$\Rightarrow$	bus B to even numbered units.
`b_odd`	$\Rightarrow$	bus B to odd numbered units.
`control_in`	$\Rightarrow$	Control word to systolic array.
`a_i`	$\Rightarrow$	Operand $A(= X)$ to to systolic array.
`q_i`	$\Rightarrow$	Quotient Q to systolic array.
`y_out`	$\Rightarrow$	Data output of `DP RAM Z`.

Figure: C.8

`result_out`	$\Rightarrow$	Result from systolic array.
`y_out`	$\Rightarrow$	Data output of `DP RAM Z`.
`y_out`	$\Rightarrow$	Data output of `DP RAM P`.
`max_count_sq`	$\Rightarrow$	Terminal count signal from `DP RAM Z`.

Figure: C.9

`data_in`	$\Rightarrow$	Data input of `Exp RAM`:
		1.value: number of bits e_i of exponent (19).
		2.value: first word of exponent $(e_{15} \dots e_0)$.

		3.value: second word of exponent $(e_{18} \dots e_{16})$.
`exponent`	$\Rightarrow$	Bit e_i of exponent.
`finish`	$\Rightarrow$	Terminal signal in `Exp RAM` (see Figure 6.5).
`sig_pre_c1`	$\Rightarrow$	Signal for state `pre-computation`$_1$.
`sig_pre_c2`	$\Rightarrow$	Signal for state `pre-computation`$_2$.
`sig_calc`	$\Rightarrow$	Signal for state `computation`.
`sig_post`	$\Rightarrow$	Signal for state `post-computation`.
`sig_load`	$\Rightarrow$	Signal for sub–state `load`.

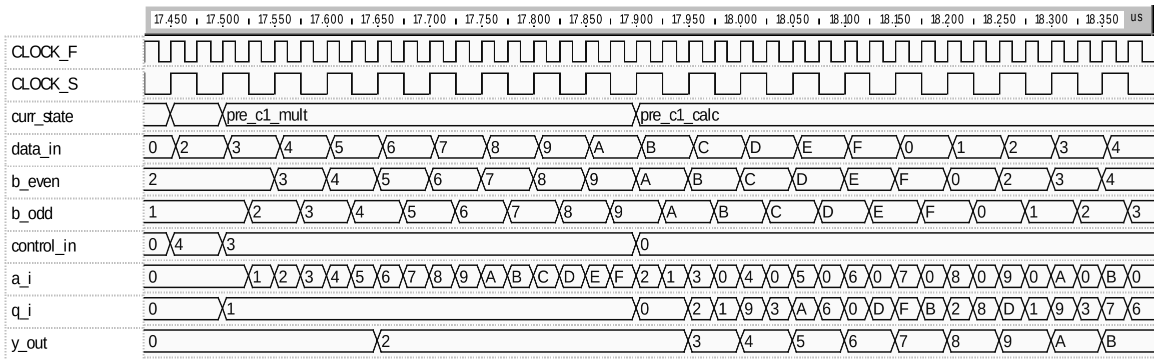


Figure C.7: Modular Exponentiation: Beginning of pre-computation

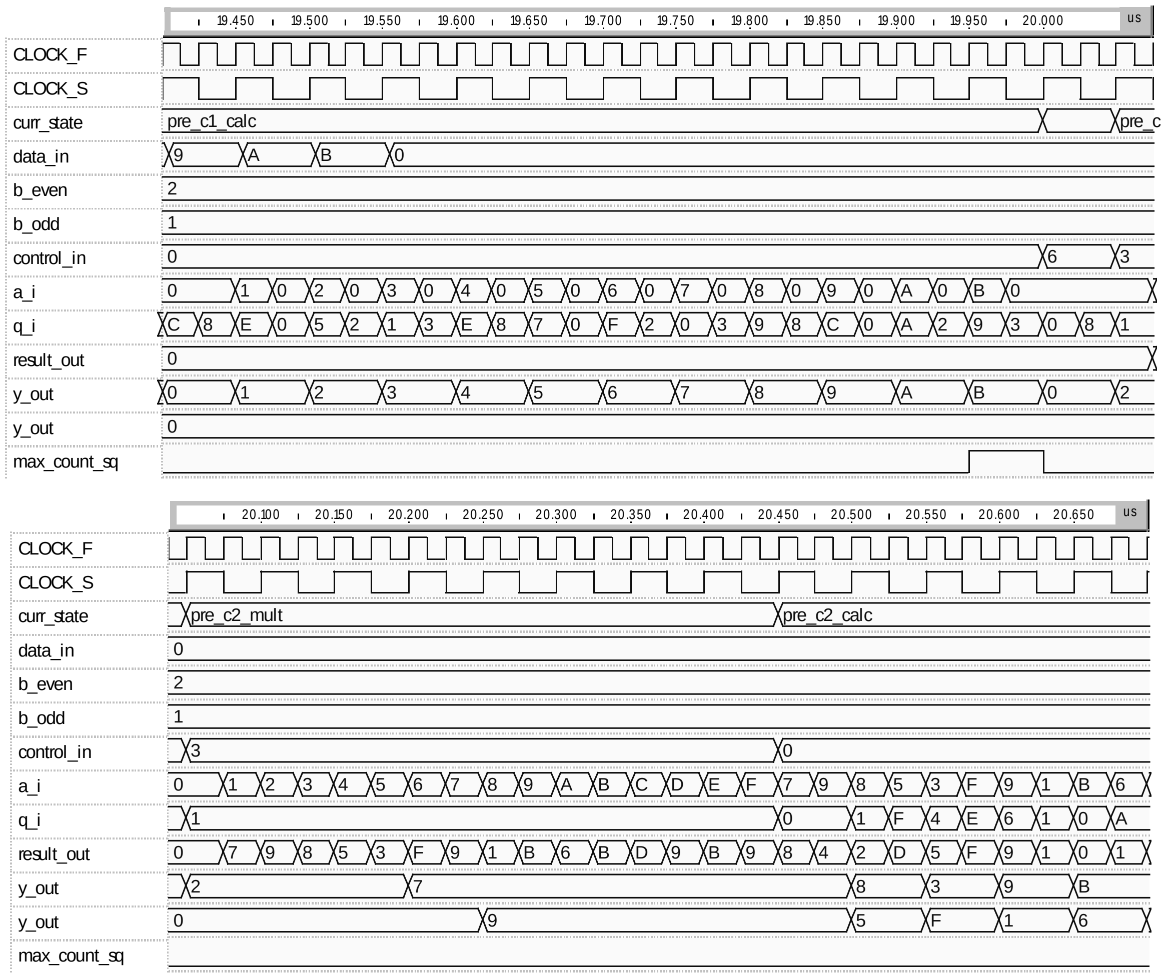
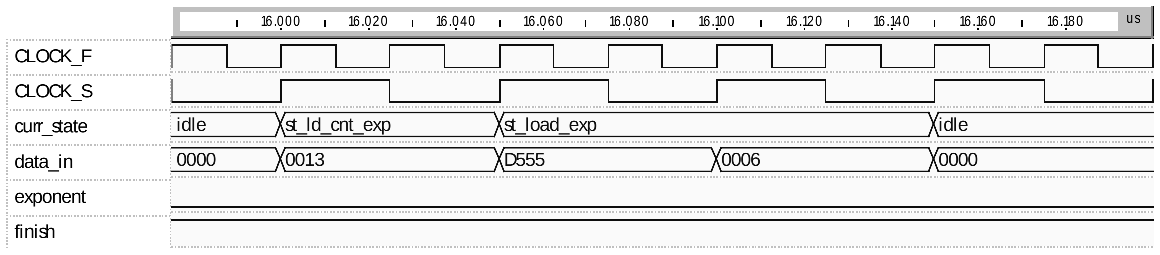


Figure C.8: Modular Exponentiation: End of pre-computation and beginning of Z_1, P_1 calculation



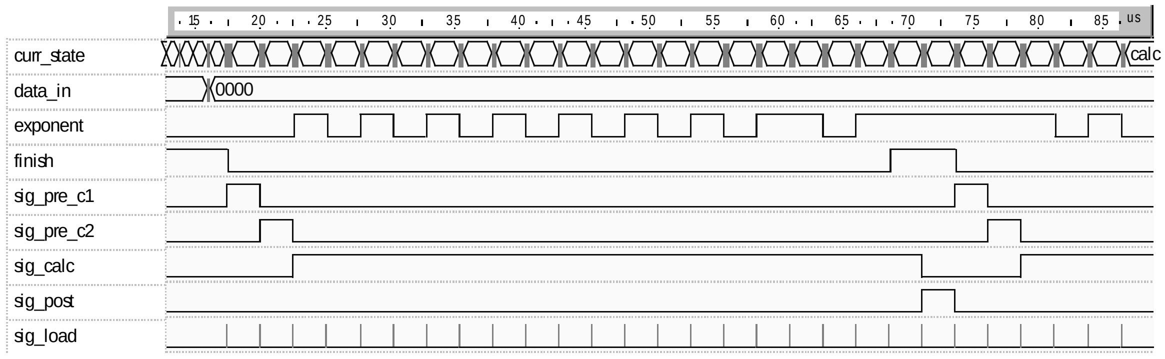


Figure C.9: Modular Exponentiation: Loading the exponent and computation of a modular exponentiation with a 19–bit exponent

Bibliography

[1] P. Alfke and B. New. *Implementing State Machines in LCA Devices.* Xilinx, Inc., San Jose, CA. Xilinx Application Note XAPP 027.001.

[2] J. Bajard, L. Didier, and P. Kornerup. An RNS Montgomery modular multiplication algorithm. *IEEE Transactions on Computers*, 47(7):766–76, July 1998.

[3] T. Beth and D. Gollmann. Algorithm engineering for public key algorithms. *IEEE Journal on Selected Areas in Communications*, 7(4):458–65, May 1989.

[4] E.F. Brickell. A fast modular multiplication algorithm with application in public key cryptography. In *Advances in Cryptology — CRYPTO '82*, pages 51–60. Chaum et al., Eds. New York, Plenum Press, 1983.

[5] E.F. Brickell. A survey of hardware implementations of RSA. In *Advances in Cryptology — CRYPTO '89*, pages 368–70. Springer-Verlag, 1990.

[6] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, 1976.

[7] S. E. Eldridge and C. D. Walter. Hardware implementation of Montgomery's modular multiplication algorithm. *IEEE Transactions on Computers*, 42(6):693–699, July 1993.

[8] W. Gai and H. Chen. A systolic linear array for modular multiplication. In *2nd International Conference on ASIC*, pages 171–4, 1996.

[9] H.Orup. Simplifying quotient determination in high-radix modular multiplication. In *Proceedings 12th Symposium on Computer Arithmetic*, pages 193–9, 1995.

[10] K. Iwamura, T. Matsumoto, and H. Imai. Montgomery modular-multiplication method and systolic arrays suitable for modular exponentiation. *Electronics and Communications in Japan, Part 3*, 77(3):40–51, March 1994.

[11] J.-P. Kaps. High speed FPGA architectures for the Data Encryption Standard. Master's thesis, ECE Dept., Worcester Polytechnic Institute, Worcester, USA, May 1998.

[12] K.Koyama and Y.Tsuruoka. Speeding up elliptic cryptosystems by using a signed binary window method. In *Advances in Cryptology — EUROCRYPT '92*, pages 345–357, May 1992.

[13] D.E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms.* Addison-Wesley, Reading, Massachusetts, 2nd edition, 1981.

[14] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[15] P. Kornerup. A systolic, linear-array multiplier for a class of right-shift algorithms. *IEEE Transactions on Computers*, 43(8):892–8, August 1994.

[16] N. Lange. Single-chip implementation of a cryptosystem for financial applications. *Lecture Notes in Computer Science*, 1318:135–44, 1997.

[17] A. J. Menezes, P.C.van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

[18] V. Miller. Uses of elliptic curves in cryptography. In *Lecture Notes in Computer Science 218: Advances in Cryptology — CRYPTO '85*, pages 417–426. Springer-Verlag, Berlin, 1986.

[19] P.L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–21, April 1985.

[20] J.J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public–key cryptosystem. *Electronics Letters*, 18:905–7, October 1982.

[21] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–6, Feb. 1978.

[22] M. Rosner. Elliptic curve cryptosystems on reconfigurable hardware. Master's thesis, ECE Dept., Worcester Polytechnic Institute, Worcester, USA, May 1998.

[23] B. Schneier. *Applied Cryptography.* Wiley & Sons, 2nd edition, 1995.

[24] S.C.Pohlig and M.E. Hellman. An improved algorithm for computing logarithms over gf(p) and its cryptographyc significance. *IEEE Transactions on Information Theory*, 24(1):106–110, January 1978.

[25] H. Sedlak. An rsa cryptography processor. In *Proceedings of EUROCRYPT '87*, pages 127–42. Springer-Verlag, 1987.

[26] M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In *Proceedings 11th IEEE Symposium on Computer Arithmetic*, pages 252–259, 1993.

[27] Sican GmbH. RSA public key algorithm. http://www.sican.de/internet/homepage_internet_d.html.

[28] S.R.Dusse and B.S.Kaliski. A cryptographic library for the motorola dsp 56000. In *Advances in Cryptology — EUROCRYPT '90*, pages 230–244. Springer-Verlag, 1991. LNCS 740.

[29] N. Takagi. A radix-4 modular multiplication hardware algorithm efficient for iterative modular multiplications. In *Proceedings 10th IEEE Symposium on Computer Arithmetic*, pages 35–42, 1991.

[30] A. Tiountchik. Systolic modular exponentiation via Montgomery algorithm. *Electronic Letters*, 34(9):874–5, April 1998.

[31] A. Vandemeulebroecke. A single chip 1024 bits RSA processor. In *Advances in Cryptology — EUROCRYPT '89*, pages 219–36. Springer-Verlag, 1990.

[32] M. Vielhaber. Entwurf und Layout eines RSA-Prozessors auf einer Chipkarte. Master's thesis, University of Karlsruhe, Karlsruhe, Germany, 1988.

[33] J.E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H.H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, Mar 1996.

[34] C.D. Walter. Fast modular multiplication using 2-power radix. *International Journal of Computer Mathematics*, 39(1–2):21–8, 1991.

[35] C.D. Walter. Systolic modular multiplication. *IEEE Transactions on Computers*, 42(3):376–8, March 1993.

[36] P.A. Wang. New VLSI architectures of RSA public key cryptosystems. In *Proceedings of 1997 IEEE International Symposium on Circuits and Systems*, volume 3, pages 2040–3, 1997.

[37] E. De Win, S. Mister, B. Preneel, and M. Wiener. On the performance of signature schemes based on elliptic curves. In *Algorithmic Number Theory Symposium III*, pages 252–266. Springer-Verlag, 1998.

[38] Xilinx, Inc., San Jose, CA. *The Programmable Logic Data Book*, 1996.

[39] J. Yong-Yin and W.P. Burleson. VLSI array algorithms and architectures for RSA modular multiplication. *IEEE Transactions on VLSI Systems*, 5(2):211–17, June 1997.