Worcester Polytechnic Institute
Digital WPI

Masters Theses (All Theses, All Years)

Electronic Theses and Dissertations

2002-04-30

Versatile Montgomery Multiplier Architectures

Gunnar Gaubatz
Worcester Polytechnic Institute

Versatile Montgomery Multiplier Architectures

by

Gunnar Gaubatz

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

by

April, 2002

Approved:

Dr. Berk Sunar Thesis Advisor ECE Department	Dr. Fred J. Looft Thesis Committee ECE Department
Dr. John A. McNeill Thesis Committee ECE Department	Dr. John A. Orr Department Head ECE Department

Abstract

Several algorithms for Public Key Cryptography (PKC), such as RSA, Diffie-Hellman, and Elliptic Curve Cryptography, require modular multiplication of very large operands (sizes from 160 to 4096 bits) as their core arithmetic operation. To perform this operation reasonably fast, general purpose processors are not always the best choice. This is why specialized hardware, in the form of cryptographic co-processors, become more attractive.

Based upon the analysis of recent publications on hardware design for modular multiplication, this M.S. thesis presents a new architecture that is scalable with respect to word size and pipelining depth. To our knowledge, this is the first time a word based algorithm for Montgomery's method is realized using high-radix bit-parallel multipliers working with two different types of finite fields (unified architecture for $GF(p)$ and $GF(2^n)$).

Previous approaches have relied mostly on bit serial multiplication in combination with massive pipelining, or Radix-8 multiplication with the limitation to a single type of finite field. Our approach is centered around the notion that the optimal delay in bit-parallel multipliers grows with logarithmic complexity with respect to the operand size n, $\mathcal{O}(\log_{3/2} n)$, while the delay of bit serial implementations grows with linear complexity $\mathcal{O}(n)$.

Our design has been implemented in VHDL, simulated and synthesized in 0.5μ CMOS technology. The synthesized net list has been verified in back-annotated timing simulations and analyzed in terms of performance and area consumption.

Preface

In this thesis I describe research work I performed in the Cryptography and Information Security Lab during my graduate studies at WPI. This work would not have been possible without the support of many people. I would like to use this place to express my most sincere gratitude to all those who have made this possible.

First and foremost I would like to thank my advisor Prof. Berk Sunar for the advice, guidance, trust, and—last not least—the funding he has provided me with. I feel honored by being able to work with him and look forward to a continued research relationship for my Ph.D.

I became involved with Cryptography in my first semester of graduate studies at WPI, when I took my first course on the subject with Prof. Christof Paar. I would like to thank him for his excellent lectures, his enthusiasm and the opportunity to work with him.

I am very grateful to the members of my thesis committee, Prof. Fred Looft and Prof. John McNeill, for their support, advice and time, especially since the latter is usually in short supply.

Thanks must also go out to my colleagues in the CRIS lab Colleen O'Rourke, Adam Elbirt, Selçuk Baktir and Seth Hardy, for the good spirit and friendship. A big “Thank you!” to my friends and roommates Jens-Peter Kaps and Pavan Reddy for interesting night-long discussions and general friendship.

Finally, and most importantly, I want to thank my parents Erwin and Ingeborg Gaubatz and my sister Corinna for their unconditional love and support they provide me with. It means a lot to me.

To all of you thank you very much!

Worcester, Massachusetts, May 2002

Gunnar Gaubatz

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many Public Key Cryptographic (PKC) algorithms, such as RSA, Diffie-Hellman, and Elliptic Curve Cryptography, require modular multiplication of very large operands (sizes from 160 to 4096 bits) as their core arithmetic operation. To perform this operation reasonably fast, general purpose processors are not always the best choice. This is why specialized hardware, e.g. in the form of cryptographic co-processors, become more attractive.

Based upon the analysis of recent publications on hardware design for modular multiplication, this M.S. thesis presents a new architecture that is scalable with respect to word size and pipelining depth. To our knowledge this is the first time a word based algorithm for Montgomery's method is realized using high-radix bit-parallel multipliers that can perform two different types of arithmetic, (1) integer arithmetic for operations in rings $\mathbb{Z}_n$ or finite fields $GF(p)$, and (2) binary polynomial arithmetic for finite fields $GF(2^n)$ in a single unified architecture.

Earlier designs have relied mostly on bit serial multiplication in combination with massive pipelining, or Radix-8 multiplication with a limitation to integer arithmetic. Our approach is centered around the notion that the optimal delay in bit-parallel multipliers grows with logarithmic complexity with respect to the operand size n, e.g. $\mathcal{O}(\log_{3/2} n)$, while the delay of bitserial implementations grows with linear complexity $\mathcal{O}(n)$. Based on this observation we expect our design to be comparable in performance with other designs, and ultimately outperform them for large values of w.

1.1 Motivation

Since its conception in 1976 by Whitfield Diffie and Martin Hellman [DH76] *Public Key Cryptography* has come a long way. Many competing algorithms and standards have been proposed and implemented. The entire concept of “eCommerce” is based on the availability of reliable and secure methods for not only encryption, but also authentication, and integrity.

With the ongoing digital revolution and advances in high performance computing, powerful desktop computer systems are available to almost everybody at low cost. While there has always been a demand for hardware implementations of public key cryptography, the volume has risen dramatically in recent years, due to a paradigm shift in communications, from wirebound to wireless. New and small handheld devices with low power consumption and more and more features keep appearing. Those devices do not possess the computing power of desktop computers, but still require strong security mechanisms. Here is where specialized cryptographic hardware comes into play. With the aforementioned multitude of different algorithms and standards, it is essential for any such hardware to support the necessary arithmetic primitives needed by those algorithms.

1.1.1 Modular Multiplication in Public Key Cryptosystems

The majority of the currently established Public-Key Cryptosystems (RSA, Diffie-Hellman, Digital Signature Algorithm (DSA), Elliptic Curves (ECC), etc.) require modular multiplication in finite fields as their core operation which accounts for up to 99% of the time spent for encryption and decryption. In order to improve the performance of the overall cryptosystem, it is therefore crucial to optimize modular multiplication.

One method of modular multiplication that is particularly suitable for those cryptosystems mentioned above is Montgomery Multiplication. It is a method that avoids the division that is usually necessary for finding the remainder, at the cost of an additional multiplication. Since division is much more costly than multiplication, this method represents a significant improvement over regular modular multiplication.

1.1.2 Keysizes and Complexity of Public Key Schemes

Current Public Key schemes are computationally very expensive. On one side this has to do with the complexity of the operations involved, e.g. modular multiplication, modular inversion, etc. On the other hand, the length of the operands involved in such operations is much larger than the word size of traditional microprocessors. They range between 160 bits for Elliptic Curve-based cryptosystems and 2048 bits or more for RSA or Diffie-Hellman.

Operations on such large operands naturally need to be broken down into word based multi-precision operations. The speed complexity of operations like multiplication is given as $\mathcal{O}(n^2)$, meaning that the time necessary for multiplication grows quadratically with the operand length n in words.

To illustrate the complexity a little more, the following short example shows an estimate x of the number of integer multiplications necessary for the central operation of two popular Public Key Schemes. One is a 160 bit Elliptic Curve scalar point multiplication and the other a 1024 bit RSA modular exponentiation. For more information on selecting key sizes for cryptographic applications, see [LV00]. For simplicity we assume a word size w of 32 bits and only count simple integer multiplications and nothing else. Although modular squaring can be implemented faster than multiplication, no distinction is made in this case. The number of integer multiplications p necessary for one full precision modular multiplication can be approximated as $2n^2$, where $n = \lceil N/w \rceil$. The average number K of modular multiplications necessary for either ECC or RSA is based on simple shift-add-methods.

	ECC	RSA
Operand length N	160 bits	1024 bits
Words / operand n	5	32
# ModMul K	2,280	1,536
# Integer Mul $p \approx 2n^2$	50	2048
Total # Mul $x = pK$	114,000	3,145,728

This table shows that the number of integer multiplications necessary to perform a single encryption is 114,000 for ECC and over 3 million for RSA. From these numbers it should be evident that public key cryptography is a time consuming application, especially on low-powered hardware for mobile use.

1.2 Algorithm Agility and Reusability

A number of different public key algorithms are in use today. To ensure compatibility with the rest of the world, cryptographic applications have to support a large portion of those algorithms. While software implementations are often easy to upgrade and to adapt to new algorithms or larger key sizes, the same is not necessarily true for hardware implementations.

Algorithm agility, the ability to support many different algorithms with the same architecture, is an important concept in the field of cryptography. The security of most cryptographic algorithms is not proven, but merely presumed to be intractable with currently available computing power. Moore's law therefore plays an important role in estimating key lengths for long term security. Also, one can never be entirely sure that better methods for cryptanalysis, than those currently known, do not exist. Quite recently a theoretical attack on RSA, based on an improved scheme for factoring integers, has been proposed in [Ber01] which, if practical, could render RSA keys of less than $\sim$1500 bits insecure.

Instead of implementing a complete cryptographic algorithm in hardware, it is often better to simply build universal arithmetic units. Several of such units for certain complex operations commonly found in cryptographic algorithms, can be integrated into a microcontroller or microprocessor. The programmability of the processor provides the flexibility of this approach, the specialization of the arithmetic unit provides the performance, and the universality of the arithmetic primitives enhances the reusability.

1.3 Scalability

Another benefit of our design is that it is a fully scalable architecture, meaning on one hand, that parameters like wordsize and pipelining depth can be chosen arbitrarily.

This gives implementors the flexibility to fit the design for various applications with differing area, timing or power constraints.

Consider the example of a network processor inside a stand-alone router. While power consumption is not much of an issue here, high-speed performance certainly is, so a large word size and deep pipeline would be the appropriate choice of parameters.

Consider a different example, like a handheld device. It should be small for easy handling, consume as little power as possible for long battery life and have a different purpose than just encrypting data. In such a setting there is not much room left for an area intensive modular multiplier. But since we have a scalable architecture, it is possible to fit in a small scale version of the design which can then assist the main CPU leaving more processing power for the actual handheld applications.

A second meaning of scalability refers to the concept of arbitrary operand size, even after implementation in hardware. The operand size is directly related to the security level of the crypto algorithm. If, for example, a certain security level of a crypto algorithm becomes inadequate due to improved computing power available to attackers, increasing the operand size usually increases the security level, given the algorithm itself is unbroken. Therefore scalable hardware is less prone to become obsolete due to demands for higher security.

1.4 Thesis Outline

After a short introduction into the mathematics of finite field arithmetic in the second chapter, the third chapter will present some of the earlier works in this field. Different concepts will be analyzed for useful ideas as well as for possible drawbacks.

Following that, the general idea of Montgomery's algorithm and more specific details of its implementation will be presented in chapter four. Chapter five will explore the different levels of parallelism that the algorithm offers to hardware designers.

Chapter six follows a bottom-up scheme of presenting the reader with the system architecture of the design that has been developed as part of this thesis. Chapter seven talks briefly about implementation issues and details, before the performance results and the analysis thereof are thoroughly discussed in chapter eight and summarized with the conclusions in chapter nine.

Chapter 2

Mathematical Background: Finite Field Arithmetic

The purpose of this chapter is to give the reader a short introduction into the mathematics of finite field arithmetic, without getting into too many details. A basic knowledge of set theory and abstract algebra is assumed.

2.1 Definitions

Definition 1 (Rings) *[MvOV97] A* ring $(R, +, \times)$ *consists of a set R with two binary operations $+$ (addition) and $\times$ (multiplication) on R, satisfying the following axioms:*

1. *$(R, +)$ is an Abelian group with identity denoted 0.*

2. *The operation $\times$ is associative. That is, $a \times (b \times c) = (a \times b) \times c$ for all $a, b, c, \in R$.*

3. *There is a multiplicative identity denoted 1, with* $1 \neq 0$*, such that* $1 \times a = a \times 1 = a$ *for all* $a \in R$*.*

4. *The operation* $\times$ *is* distributive *over* $+$*. That is,* $a \times (b + c) = (a \times b) + (a \times c)$ *and* $(b + c) \times a = (b \times a) + (c \times a)$ *for all* $a, b, c \in R$*.*

The ring is a commutative ring *if* $a \times b = b \times a$ *for all* $a, b \in R$*.*

An example of a ring is the set of integers $\mathbb{Z}$.

Definition 2 (Fields) *[MvOV97] A* field *is a commutative ring in which all non-zero elements have multiplicative inverses.*

This means that for all elements $a \in R \setminus \{0\}$ there is another element a^{-1} from the same set such that $a \times a^{-1} = 1$. Any ring for which this condition is not fulfilled is therefore not a field. The set of integers $\mathbb{Z}$, for example, is not a field, since the only two elements that have a multiplicative inverse in $\mathbb{Z}$ are -1 and 1. On the other hand, the ring $\mathbb{Z}_p$ with addition and multiplication performed modulo p is a field if and only if p is prime.

Definition 3 (Field Characteristic) *[MvOV97] The* characteristic *of a field is* 0 *if* $\overbrace{1 + 1 + \cdots + 1}^{m}$ *is never equal to* 0 *for any* $m \geq 1$*. Otherwise, the characteristic of the field is the least positive integer* m *such that* $\sum_{i=1}^{m} 1$ *equals* 0*.*

Definition 4 (Polynomial Rings) *[MvOV97] If* R *is a commutative ring, then a* polynomial *in the indeterminate* x *over the ring* R *is an expression of the form*

$$f(x) = a_n x^n + \cdots + a_2 x^2 + a_1 x + a0$$

where each $a_i \in R$ *and* $n \geq 0$. *The element* a_i *is called the* coefficient *of* x^i *in* $f(x)$. *The largest integer* m *for which* $a_m \neq 0$ *is called the* degree *of* $f(x)$, *denoted* $\deg f(x)$; a_m *is called the* leading coefficient *of* $f(x)$.

Definition 5 (Finite Fields) *[MvOV97] A* finite field *is a field* F *which contains a finite number of elements. The* order *of* F *is the number of elements in* F.

Some facts about finite fields [MvOV97]:

1. If F is a finite field, then F contains p^m elements for some prime p and integer $m \geq 1$.
2. For every prime power order p^m, there is a unique (up to isomorphism) finite field of order p^m. This field is denoted by $\mathbb{F}_{p^m}$, or sometimes by $GF(p^m)$.

2.2 Arithmetic over Prime Fields $GF(p)$

Arithmetic over prime fields $GF(p)$ is in principle simply a combination of integer arithmetic with intermittent modular reduction steps whenever the result grows larger than the modulus M. Just like regular integer arithmetic it depends on carry propagation. Modular multiplication in its simplest form requires trial division for finding the multiple of the modulus that needs to be subtracted from the result and is therefore inherently slow. For this reason the Montgomery multiplication algorithm is used in this thesis as a faster alternative.

2.3 Arithmetic over Binary Extension Fields $GF(2^n)$

Finite fields $GF(p^m)$ with $m > 1$ are often represented in polynomial basis representation. The special case where $p = 2$ is usually referred to as binary extension fields. This class of finite fields is particularly suitable for arithmetic on digital computers because of the straightforward representation of coefficients as binary bit strings.

Arithmetic in binary extension fields has different properties than prime fields, but is structurally very similar. The role of the prime modulus is adopted by an irreducible polynomial $f(x)$ of degree m.

2.3.1 Addition

Addition of two binary polynomials is performed as the addition of its coefficients modulo two without any carries

$$A(x) + B(x) = \sum_{i=0}^{m} (a_i + b_i)x^i \bmod 2$$

which in terms of logic circuits directly translates into XOR combinations of the coefficients

$$A(x) + B(x) = \sum_{i=0}^{m} (a_i \oplus b_i)x^i$$

It is obvious that addition of binary polynomials can be implemented in hardware very efficiently. Subtraction is the exact same operation, since each coefficient is its own additive inverse.

2.3.2 Multiplication

Multiplication in $GF(2^n)$ is only slightly more complex:

$$A(x)B(x) = \sum_{i=0}^{m}\sum_{j=0}^{m}(a_i b_j)x^{i+j} \bmod f(x)$$

With $p = 2$ the partial products $a_i b_j$ are the outputs of simple logical AND gates in hardware and summed up as before by XOR gates, according to their position in the resulting polynomial.

Modular reduction takes place by adding (subtracting) $f(x)x^{k-m}$ repetitively to the result, as long as the degree of the result $k \geq m$. This approach is very simplistic, but achieves the desired effect. A more efficient method for modular reduction is available in form of Montgomery's algorithm. Only a couple of minor adaptions are necessary to make this integer arithmetic algorithm work in conjunction with binary polynomials, as shown in [KA98].

In conclusion it can be said that except for the modular reduction, binary polynomial arithmetic is very similar in structure to integer arithmetic. The only big difference is the absence of any sort of carry propagation, which makes this type of arithmetic so attractive for high speed hardware implementations.

Chapter 3

Previous Work

There have been numerous designs implementing modular multiplications over the years. This chapter describes some of them and contains a detailed analysis of their merits and drawbacks.

Not all of these designs are based on Montgomery multiplication, and are therefore not directly comparable to our design. Still there are other aspects in those designs that are worthy of further investigation, like scalability issues and the use of systolic arrays.

3.1 Montgomery Based Multiplier Designs

Three of the proposed architectures that will be discussed in this section are based on Montgomery's algorithm for modular multiplication. It is not surprising that they appear almost identical, given the fact that they all originated from the same team of researchers at Oregon State University and are based one upon the other. Still there are a couple of details distinguishing the three designs which are worth mentioning.

3.1.1 Bitserial Integer Multiplier

The foundation for this series of multiplier architectures was laid in 1999 by A. Tenca and Ç. Koç's original paper [TK99]. It was centered around the ideas of the word based Montgomery Multiplication algorithm for finite fields GF(p) known from traditional software implementations, but the multiplication itself was performed in a bitserial fashion. Two design choices play a key role in why the design performs adequately fast despite the fact that the use of bitserial multiplication requires a lot of clock cycles:

1. Carry propagation during the addition of partial products is deferred until the very end of the algorithm by using Carry Save Adders extensively and keeping the result in redundant representation throughout most of the algorithm. This makes it possible to use wordsizes of up to 128 bits without causing a significant degradation of the clock frequency.

2. Massive pipelining achieves a high degree of parallelization at the cost of a negligible start-up latency.

One of the outstanding advantages of this architecture is the level of scalability that it displays. First of all the wordsize of the datapath is configurable and no major re-design is necessary. Secondly the number of pipeline stages is configurable as well, and adding a stage comes at nearly no cost, since it only requires an instantiation of one further macro-cell. These two parameters, wordsize and pipeline-depth, give the implementor a lot of choice with regards to area and speed. The notion of scalability also plays an important role once the design is manufactured. The word based algorithm around which this design is based does not, in principle, limit the maximum operand size it can handle. In practice, of course, issues like the size of certain storage elements and counters do impose a maximum wordsize. For practical applications, however, the demand for larger operands is predictable for the near future, so that provisions to accomodate them can be made easily.

The main drawbacks of the architecture are the fact that operand conversions from integer to the Montgomery domain representation and vice-versa are necessary before, respectively, after the multiplication. This, however, is a general drawback of Montgomery's method in opposition to other techniques. Also, this becomes negligible once a sufficient number of modular multiplications need to be performed in a row, e.g. like in a modular exponentiation for the RSA cryptosystem.

Another flaw of the design is the high number of clock cycles needed to compute a full modular multiplication in comparison to that of other techniques. This has to do mainly with the bitserial approach that was chosen in favour of a high radix design.

Finally, this first version of the design is suitable only for integer arithmetic modulo N in a ring $\mathbb{Z}_N$. It does not address other types of finite field arithmetic in use by modern cryptosystems like arithmetic over binary extension fields $GF(2^n)$. This type of arithmetic is frequently used in *Elliptic Curve Cryptosystems (ECC)*.

3.1.2 Unified Bitserial Integer and Polynomial Multiplier

The first extension of this basic architecture by E. Savaş was published in August 2000 [STK00]. The paper describes a modification of the architecture's arithmetic kernel which, in addition to integer arithmetic, also allows computation on binary extension fields $GF(2^n)$ at the cost of only a slight increase of the gate count.

Despite this added functionality the basic drawbacks mentioned in connection with the original design remain. Conversion between number systems are still necessary and the number of clock cycles is high.

3.1.3 Radix-8 Integer Multiplier

G. Todorov et. al. [TTK01] published another modification in May 2001. As before it is based upon the original architecture by Tenca and Koç, but this time investigates the use of high radix multiplication as an alternative to bitserial multiplication.

Unfortunately the radix chosen by the team is fixed to 8, so that w bits of one operand are multiplied by only 3 bits of the second operand in each arithmetic unit. The speed improvements over the bitserial design are only marginal, considering that the radix-8 design was synthesized using 0.5μ CMOS technology instead of 1.2μ as in the earlier paper. The results do not show the anticipated performance improvement over bitserial multiplication that are expected from high-radix designs, simply because the increase in radix was not high enough. The attempt at reducing the increased complexity of adding up partial products by using Booth-Recoding [Boo51] on the multiplier input, produces such a huge overhead in delay that it hides the benefits of the higher throughput in bits per clock cycle.

Apart from the controversial [OVL96] benefits of Booth recoding for reducing the delay of small multipliers, it additionally makes the design of a unified Montgomery multiplier tremendously complicated. Techniques like Booth recoding conceptually rely on the notion of carry propagation, and don't work for multiplication of binary polynomials. As a consequence the proposed design had to drop the support of $GF(2^n)$ arithmetic, which prevents its use in algorithm agile cryptographic processing units.

3.2 Alternative Schemes for Modular Multiplication

In August 2000 J. Großschädel presented his work [Gro00] on an alternative modular multiplier architecture, which is based on Barrett's technique for modular reduction.

The main advantage of Barrett's method is that it operates in the regular number system and no transformation into a residue number system is necessary. The basic principle behind this technique is to compute an estimated quotient $\tilde{q} = \left\lfloor \frac{P}{M} \right\rfloor$ which is used to subtract a multiple of the modulus $\tilde{q}M$ from the most significant bits of the product P. This produces a nearly complete reduction of the product. The full reduction is computed in the final step by repeatedly subtracting the modulus and comparing the result.

The drawback of this technique is the fact that the quotient estimation works on the most significant portion of the result, which makes it very difficult to come up with a scalable architecture to allow arbitrary precision modular multiplication. As expected the design proposed by Großschädel turns out to be fixed in precision to a maximum of 1024 bit operands.

The main purpose of this architecture was to serve as the core for modular exponentiation in an RSA crypto accelerator chip. The limitation to this specific application essentially prevented a more flexible design capable of operating on both types of fields $GF(p)$ and $GF(2^n)$.

On the other hand, concentrating on one particular application brought about a very fast architecture which only needs 227 clock cycles to perform one modular multiplication, while running at a clock frequency of 200 MHz. This is made possible by a very large partial parallel multiplier of size 1056 × 16 bits, based on a systolic array structure. The huge area requirements of this design approach, however, are obvious.

In May 2001 the same author presented an entirely new architecture [Gro01] that completely relies on a bitserial shift-add method for multiplication combined with repetitive subtraction of the modulus. Like in the previous design, the subtraction of the modulus is based on estimations made from the most significant portion of the intermediate result. This time, however, the architecture provides the necessary mechanisms to perform multiplication on two different types of fields, the afforementioned Galois Fields $GF(p)$ and $GF(2^n)$. Similar to the bitserial multiplication in the Tenca-Koç design, carry propagation is deferred until the end of the multiplication by using Carry Save Adders and keeping the result in redundant representation.

Once again, one of the main drawbacks of the design is the evaluation of the most significant portion of the result for modular reduction, which resulted in a full-precision data-path implementation. Again the design is not scalable with respect to arbitrary precision operand sizes. Due to the bitserial multiplication strategy the number of clock cycles is larger than that of the previous architecture.

On the positive side this increase in the clock cycle count also greatly reduces the area requirements. Additionally this design also works with a regular number representation.

3.3 Scalable versus Fixed Precision Architectures

As is evident from the last section, the general algorithm of how to perform modular multiplication is irrelevant – a lot of different methods exist and all of them work. The real issue is the flexibility that an architecture provides in terms of scalability and choice of parameters like area requirements, speed and support of different types of arithmetic.

The cost-effectiveness of a hardware architecture is determined not only through the cost associated with development and manufacturing, but also through the duration of its use. This in turn is highly dependend on the flexibility the architecture exhibits in different circumstances. If, hypothetically speaking, advances in cryptanalysis suggest that the RSA cryptosystem with keylengths of 1024 bits prove to be not adequate for long term security any more, then the keysize needs to be extended. Architectures which do not scale to meet the new demands have to be replaced and thus create costs.

3.4 Unified Architectures

The flexibility of an architecture to work with different types of arithmetic is a key feature for modern information security applications. A wealth of competing cryptographic algorithms exist and have been standardized. Supporting a broad range of these algorithms is no longer optional, but a necessity. The most promising model of addressing this issue in the design of a cryptographic co-processor is to add efficient arithmetic and logic primitives to a standard microprocessor / -controller architecture. This ensures an upgrade path to support future algorithms and changes to existing schemes, while preserving the speed advantages of a specialized design.

By combining the support for integer and binary polynomial arithmetic into one single unified architecture, as done in [STK00] and [Gro01], less area is needed for the same functionality.

Chapter 4

Montgomery Multiplication

In 1985 Peter L. Montgomery proposed a method [Mon85] for modular multiplication using Residue Number System (RNS) representation of integers. It replaces the costly division operation usually needed to perform modular reduction by simple shift operations, at the cost of having to transform the operands into the RNS before the operation and re-transforming the result thereafter.

A radix R is selected to be two to the power of a multiple of the machine word size and greater than the modulus, i.e. $R = 2^{kw} > M$. For the algorithm to work R and M need to be relatively prime, i.e. must not have any common non-trivial divisors. With R a power of two, this requirement is easily satisfied by selecting an odd modulus. This also fits in nicely with the cryptographic algorithms that we are targetting, where the modulus is either a prime – always odd with the exception of 2

– or the product of two primes and therefore odd as well.

RNS representations of integers are called M-residues and are usually denominated as the integer variable name with a bar above it. An integer a is transformed into its corresponding M-residue $\bar{a}$ by multiplying it by R and reducing modulo M. The back-transformation is done in an equally straight-forward manner by dividing the residue by R modulo M. Thus we have the following equations as transformation rules between the integer and the RNS domain:

$$\bar{a} = aR \pmod{M} \tag{4.1}$$

$$a = \bar{a}R^{-1} \pmod{M} \tag{4.2}$$

Montgomery Multiplication can be written simply as the product of two M-residues divided by the radix modulo M:

$$\bar{c} = \bar{a}\bar{b}R^{-1} \pmod{M} \tag{4.3}$$

Division by the Radix is necessary to make the result again an M-residue. This becomes more obvious as we expand the equation in the following way, in which we also introduce the function name $MM(Op_1, Op_2)$ for the Montgomery Multiplication algorithm:

$$\begin{aligned}\bar{c} &= MM(\bar{a},\bar{b}) \\ &= \bar{a}\bar{b}R^{-1} \pmod{M} \\ &= aRbRR^{-1} \pmod{M} \\ &= (ab)R \pmod{M} \\ &= cR \pmod{M}\end{aligned}$$

Assuming we have an implementation for of the MM algorithm at our disposal, it looks as if we still need a method to perform regular modular reduction if we want to transform integer variables into their respective M-residues. However, once the precision, and therefore the radix R, is fixed for the implementation, we can use the pre-computed constant $R^2 \pmod{M}$ in conjunction with the MM algorithm for transformation purposes:

$$\begin{aligned}\bar{a} &= MM(a,R^2) \\ &= aR^2R^{-1} \pmod{M} \\ &= aR \pmod{M}\end{aligned} \qquad (4.4)$$

$$\begin{aligned}a &= MM(\bar{a},1) \\ &= aRR^{-1} \pmod{M}\end{aligned} \qquad (4.5)$$

The benefits of Montgomery Multiplication over classical methods involving division are not overly evident for applications with only a few modular multiplications. However, for algorithms in which a lot of modular multiplications need to be performed with respect to the same modulus, the performance gain is much more obvious, since the ratio between transformation overhead and actual modular arithmetic is much lower.

For the sake of simplicity we will drop the ”bar” notation for distinguishing M-residues from integers throughout the remainder of this thesis, since the transformation to and from the RNS is not of significance here. When it becomes necessary to distinguish the two domains, extra indication will be provided.

4.1 General Algorithm

Algorithm 1 outlines an implementation of Montgomery's method for the single precision case. A multiple precision word based version will be presented later. This particular example illustrates well the separate multiplication and reduction steps of the method.

As a prerequisite this algorithm expects a value M' that like the modulus M itself, may be treated as a constant, because it rarely changes. This value M' is part of the main trick behind Montgomery's method: it is used in conjunction with the lower half of the product P to compute the number of multiples U of the modulus M that need to be added to P to make its lower half become zero. Note that an addition of an integer multiple of the modulus does not change the congruence between the result and the product P. Since now the lower half is all zero, we can safely shift the

Algorithm 1: Single precision Montgomery multiplication

```
Require: $a, b \in \mathbb{Z}_M$, $n = \lceil \log_2 M \rceil$, $R = 2^n$, $M' = -M^{-1} \pmod R$
$P = ab$
$U = (P \bmod R)M' \bmod R$
$c = (P + UM)/R$
if $c \geqslant M$ then
    $c = c - M$
end if
```

result to the right, which is equivalent to a division by R.

It must be noted without going into any further details that the result c might not always be fully reduced with respect to the modulus M. Therefore it might be necessary to perform a final subtraction of the modulus. Depending on the algorithm employing Montgomery Multiplication, however, in some cases this may be delayed until the final step which transforms the M-residue result back to integer form.

4.2 Word-based Algorithms

In practice, primitive arithmetic operations such as multiplication and addition are limited to a certain word size w. Operands of cryptographic algorithms, on the other hand, tend to be very large, so that multiple precision arithmetic comes into play.

The simplest way of adapting Montgomery's algorithm to large operand sizes would hence be, to just replace every arithmetic operation by its multi-precision equivalent. More efficient ways to achieve the same are analyzed and presented in [KAK96].

The criteria for selecting the most suitable algorithm is not limited to the number of multiplication operations alone. The specific architecture targeted for the implementation also plays an important role. While the "Coarsely Integrated Operand Scanning" (CIOS) method is the most suitable one for implementation on a standard PC, certain Digital Signal Processors (DSPs) feature special arithmetic operations and multiple memory busses which make an implementation of the "Finely Integrated Product Scanning" (FIPS) method a much better choice.

4.3 Finely Integrated Operand Scanning (FIOS) Algorithm

In a custom hardware implementation the amount of speed-up compared to general purpose processors and software mostly relies on the level of parallelization that can be achieved. For the selection of a particular Montgomery Multiplication algorithm this means that data dependencies between parallel arithmetic units and storage of intermediate results need to be kept to a minimum and local. From this perspective the most suitable algorithm is a slight variant of the "Finely Integrated Operand

Scanning" (FIOS) method.

4.3.1 Bitserial FIOS Method

A bitserial word based version of the FIOS Montgomery algorithm has first been proposed in [TK99]. It was later refined to also work on binary extension fields [STK00] with only minor architectural changes.

The following conventions are used in the explanation of the algorithm: Entire words of an operand are type-set in upper-case and referenced with their index in square brackets starting from zero. Thus $A[3]$ stands for the fourth word of A. Single operand bits are indicated using lower-case operand names with the index as a subscript, i.e. b_8 references the ninth bit in B.

As can be seen in Algorithm 2 the multiplication and reduction steps are tightly integrated. First one word of A is multiplied by a single bit of operand B and added with the previous round's intermediate result D. The least significant bit is examined and stored for use during the remainder of the inner i-loop. In case the bit is set, the odd modulus M will be added to the product in order to zero out the least significant bit. This makes it possible to shift the result one bit to the right without losing information. Writing back a word of the intermediate result D to memory is delayed by one step (line 8) in order to include the next round's least significant bit that gets shifted in.

Algorithm 2: Bitserial word based version of the Montgomery algorithm

```
$D = 0$ {initialize all words of the result}
for $j = 0$ to $n - 1$ do
    $(c, S) = A[0]b_j + D[0]$
    $u = s_0$
    $(c, S) = (c, S) + M[0]u$
    for $i = 1$ to $e - 1$ do
        $(c, T) = c + A[i]b_j + D[i] + M[i]u$
        $D[i - 1] = (t_0, S_{w-1 \ldots 1})$
        $S = T$
    end for
end for
```

The low area requirements of this algorithm make it very attractive for hardware implementations in which size and/or power consumption are the critical constraints. Bitwise multiplication can be implemented by a single AND gate per bit position. Furthermore the costly propagation of carries produced by the additions can be postponed until the very end of the algorithm by keeping intermediate results in Carry Save notation.

4.3.2 High-Radix FIOS Method

Algorithm 2 can be easily modified from bitserial multiplication to high radix digit multiplication, as shown in Algorithm 3. The intrinsic complexity of digit multipliers, however, significantly increases the area requirements of hardware implementations. On the other hand it also helps to decrease the number of clock cycles it takes to complete the algorithm. Furthermore, as the word size w of the digit multiplier increases, the clock period only grows logarithmically due to parallelizable addition of partial products.

Apart from multiplier complexity, the major change in the new algorithm is that not only the least significant *bit* of the result is to be made zero by adding M. An additional multiplication is necessary to compute the factor U, which represents the number of multiples of M to be added, to zero out the entire least significant *word* of the result. Accordingly the result can be shifted w bits to the right without losing

Algorithm 3: High Radix version of the Montgomery algorithm

```
$D = 0$ {initialize all words of the result}
for $j = 0$ to $e - 1$ do
    $(C, S) = A[0]B[j] + D[0]$
    $U = SM'_0 \pmod{2^w}$
    $(C, S) = (C, S) + M[0]U$
    $(C, S) >> w$
    for $i = 1$ to $e - 1$ do
        $(C, S) = (C, S) + A[i]B[j] + D[i] + M[i]U$
        $D[i - 1] = S$
        $(C, S) >> w$
    end for
    $D[n - 1] = S$
end for
```

data.

4.3.3 Complexity Comparison

As it has been stated before, the more complex structure of digit multipliers significantly increases the area requirements of a hardware implementation of Montgomery's Algorithm. That said, however, it is also true that more data can be processed in fewer clock cycles, and that the clock period grows logarithmically instead of linearly. These observations lead to the conclusion that a Montgomery architecture based on high radix digit multipliers can perform *asymptotically* better than its bitserial counterpart. The emphasis of the last sentence is on asymptotical, since additional overhead in practical implementations of digit multipliers might obscure this theoretical observation for word sizes below a certain threshold.

Chapter 5

Inherent Parallelism in Montgomery Multiplication

The common trade-off when it comes to implementation of an algorithm in hardware versus one in software is that flexibility is sacrificed for speed. In many cases, however, the use of a particular algorithm is very specific to a certain application, so that a loss of flexibility is a low price to pay for the performance improvement. An additional benefit is that the hardware solution can be optimized for reduced power consumption, since only a subset of all the features available on general purpose processors will be necessary.

There are a number of different ways to improve on the performance of complex operations in hardware. While logic and arithmetic operations take at least one clock cycle each in software implementations, multiple logic operations can be combined into a single clock cycle in custom built hardware. Intermediate results can be stored in fast local registers instead of in a standard register file which may be placed far away. Loop-Unrolling may be used to perform multiple iterations of a task in a single clock cycle where this would help in balancing the critical paths of different tasks. Data independent shift operations or even permutations can be hardwired and therefore cost virtually nothing, while they are slow in software.

Perhaps the most efficient way of speeding up complex operations in hardware, however, is through the utilization of inherent parallelisms that a particular algorithm offers. Identifying these parallelisms is only the first step, during which automated tools for algorithm analysis and transformation might be helpful. It should be noted, however, that the success rate of such tools is limited and often even thorough analysis by hand is difficult.

In the following sections three possible ways are identified, how to parallelize the Montgomery algorithm at different levels. The degree of parallelism that can be achieved varies with the data dependency of a particular level. Sometimes, when real simultaneity is impossible due to dependence on output from an earlier step, processes can still be overlapping in time, e.g. through the use of pipelining. Depending on the definition this can still be viewed as a form of parallelism, so it is included in this chapter.

The first and innermost level of parallelism can be found inside the high radix digit multiplier, which is a core component of this architecture. One level higher the parallel computation of the product $A[i]B[j]$ and the product used for reduction $UM[i]$ are computed completely in parallel, once the initialization phase is over. Finally, pipelining of multiple MM Units constitutes yet another level of parallelism.

5.1 Unified Digit Multiplier Core

Multiplication as an arithmetic operation in hardware has been studied well. Many different methods and architectures have been proposed and built. A very general characterization shows serial multipliers on one side and bit-parallel designs on the other side of the area-delay trade off spectrum.

Bitserial multiplication is usually very easy to implement, requires little area and introduces very little delay into the critical path. However, due to the iterative nature of the operation the number of clock cycles needed for a $n \times n$ bit multiplication grows linearly with the operand size n. Even if we ignore the problem of carry propagation for a moment, there are physical limits to the fastest clock speed that can be achieved in a particular technology, and it is thus doubtful if bitserial multiplication is the best choice when it comes to performance.

Parallel multiplier designs are typically much more complex and tend to have a longer critical path which in turn limits the maximum clock speed. However, the parallel generation of partial products also enables their addition in a parallelized fashion, using tree structures. The addition of all partial products can be performed in one single long clock cycle. The delay introduced, e.g. by a Wallace tree structure for n partial products, has the lowest bound of $\log_{3/2} n$ levels of full adders.

This observation leads to the conclusion that parallel multipliers can outperform serial architectures in terms of speed, at the cost of more complex hardware. On the other hand increased complexity and irregularities in the hardware design also introduce additional delays due to longer wires. Obviously there must be an optimum operand size for the parallel multiplier.

5.1.1 A Detailed Look at Integer Multiplication

In very generic terms the process of multiplication can be broken up into two basic steps, (1) generation of partial product terms, and (2) addition of partial products.

Partial Product Generation The first step basically consists of multiplying each bit of the first operand with each bit of the second operand and aligning the result in the correct bit position (column) for the second step. Single bit multiplication is a very inexpensive operation as it only takes a simple AND gate.

Addition of Partial Products Compared to the simplicity of step one, the second step is much more complex. Adding up partial product words involves carry propagation which is a classic problem in computer arithmetic. The problem of designing fast and efficient adder structures has been studied over a long time and many architectures have been proposed. Common to them all is again the trade-off between area and delay.

Digit Multipliers As we move from bitserial towards high radix digit multipliers, the problem of adding the partial products becomes even more complex. Fully propagating all carries up to the most significant bitposition in each addition step is not feasible and also not necessary. Alternative solutions make use of partial carry propagation through a technique known as *Carry Save Addition* which defers the full carry propagation until the end of the operation. Partial carries that are generated at one level along the way are only passed on to the immediate next stage, one level lower. These partial carries are sometimes also called *horizontal carries.* The addition process of step (2) can therefore be split up into the two sub-processes (a) *column compression*, and (b) *final addition.* The typical architecture of a digit multiplier is depicted in Figure 5.1

Column Compression The amount of delay that is introduced by column compression greatly depends on the topology of the compression network. These compression networks are typically constructed from so-called (3,2) counters, which are technically the same as full adders. The name is derived from the number of in-

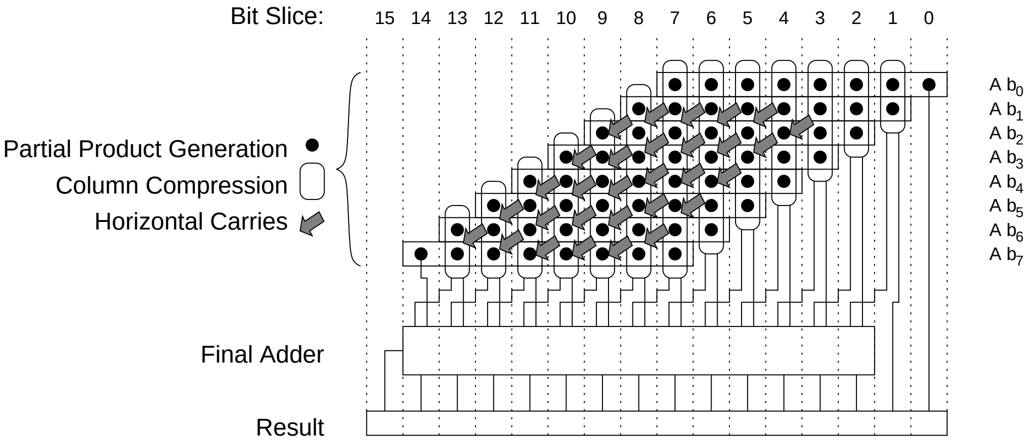

Figure 5.1: 8 × 8-bits digit multiplier

puts and outputs, respectively, and the fact that the two output bits, sum and carry, interpreted as a two bit integer give the count of active input bits.

The combination of two (3,2) counters as shown in Figure 5.2 constitutes an element known as a [4:2] compressor. Note that in actuality it has five inputs and three outputs, but one of each is used for horizontal carry propagation and not counted towards compression ratio. Higher order compressors can be built in a similar way.

Most of the regular topologies proposed for column compression are based upon these two building blocks. A good example is the double array topology shown in figure 5.3. The problem with simple regular topologies is that, although they often improve the delay of column compression, it is still linearly dependend on the multiplier word size.

The more complex types of regular topologies are tree topologies, such as *Binary*

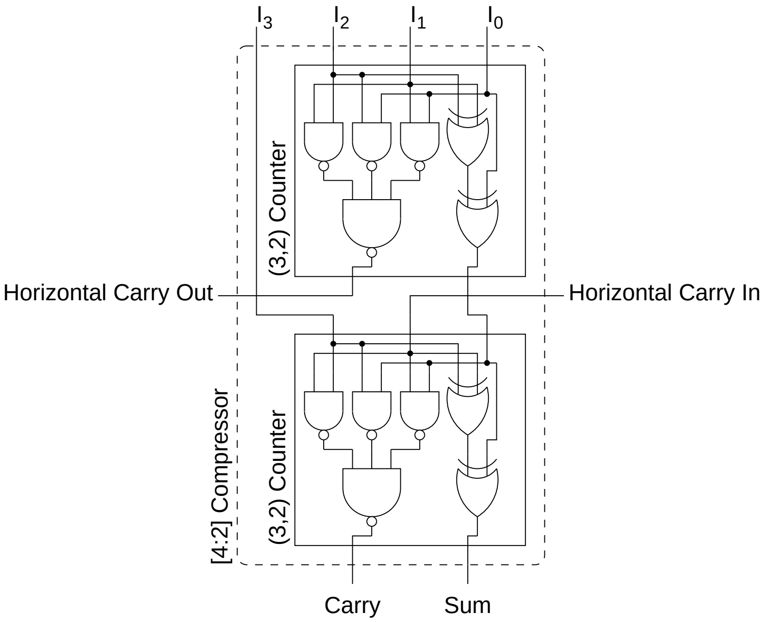

Figure 5.2: [4:2] Compressor constructed from two (3,2) counters

Tree, Balanced Delay Tree and *Overturned Staircase Tree.* In the case of a balanced delay tree, for example, the delay only grows with complexity $\mathcal{O}(2\sqrt{n})$.

The biggest problems of more complex topologies compared to simpler ones are their increased wire lengths. As the feature sizes of technologies shrink below 0.5μ, the delay caused by parasitic resistance and capacitance of long wires begin to dominate the gate delay.

Complex compression topologies would ideally require a three dimensional layout in which each column can accommodate a two dimensional tree structure. The floorplan for current VLSI technologies, however, is limited to two dimensions, and so the layout has to be flattened into a single bit slice, moving components away from

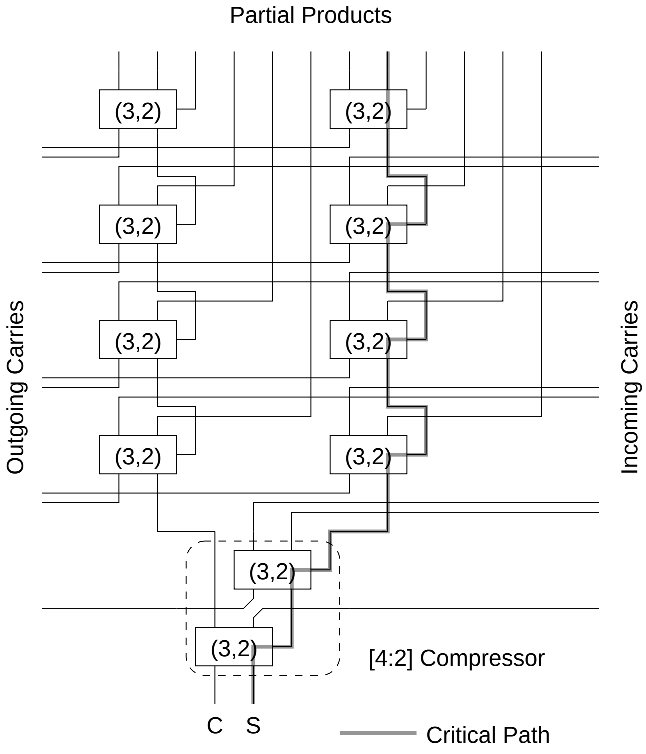

Figure 5.3: Double array column compression topology

each other. This problem becomes even more apparent with irregular topologies, like *Wallace Trees.*

Wallace trees are composed entirely of (3,2) counters and were first proposed in [Wal64]. They achieve the highest degree of parallelization possible and their delay only grows with complexity $\mathcal{O}(\log_{3/2} n)$. Traditionally Wallace Trees were not embraced by designers, because they are much harder to design and layout due to their irregular structure, as mentioned before.

In recent times, however, algorithmic layout and placement has been investigated as a possible solution to this problem. Oklobdzija et al. [OVL96] were the first to propose an algorithm that honors the difference between fast and slow inputs and outputs of (3,2) counters. However, it did not consider the influence of different wirelengths on the overall delay along the critical path. Later versions of such an algorithm that also address wire delays were presented in [FO01].

Final Adder Column compression produces as output the product $A \times B$ in redundant Carry Save form, i.e. each column ends with two outputs, *carry* and *sum*. To bring it back into non-redundant binary integer representation, all *sum* outputs must be added to the *carry* outputs using the *Final Adder* which completes the task of fully propagating all the remaining carries up to the MSB.

Since the Carry output is one bit position more significant than the Sum, it has to enter the final adder shifted to the left by one. In hardware this is easily done by appropriately wiring the two components, as can be seen in Figure 5.1.

5.1.2 Special Case: Carryless Multiplication

In our goal to build an architecture that is suitable for multiplication of integers as well as binary polynomials, we have to modify the digit multiplier slightly. As has been pointed out in chapter 2 the main difference between integer and binary polynomial arithmetic is that the latter does not have to deal with carries at all.

All computations between coefficients of the same order are performed modulo

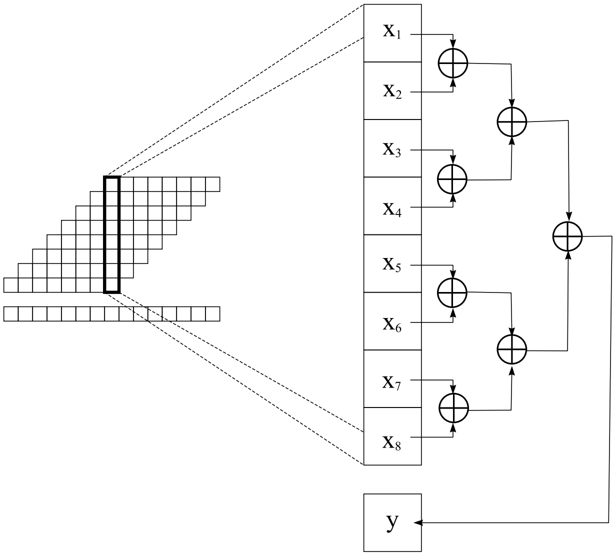

Figure 5.4: Optimal topology for binary polynomials: XOR Tree

2. This property has the beneficial effect that addition can be performed by simply XOR'ing the two operands.

The modifications to the multiplier circuit which allow both integer and polynomial operations are relatively simple and only affect the column compression section and, to a lesser degree, the final adder. Since carries are not relevant for arithmetic modulo 2, we must take care that they are not propagated from one bit slice to the next–otherwise they would get added to the result. We therefore insert controlled gates between neighboring columns that form a sort of carry blockade for each horizontal carry that is generated in a bit slice. The cost for this modification is nearly negligible. An AND gate controlled by the mode signal for each horizontal carry is all that is needed. If the value of the mode signal is a logical zero, any incoming carry will be inhibited from passing through. The final adder is not necessary either. We therefore pick the final result straight from the sum output of the column compression section instead from the final adder. A multiplexer controlled by the mode signal delivers the correct result to the output.

5.2 Parallel Computation of Product and Reduction

Going back to algorithm 3 we see that the main operation performed in the inner loop is an addition of four terms, of which two are multiplications. Once the initialization phase–where the factor U is computed–is over, both products can be computed completely independent from one another, as they have no common terms. The first product $A[i]B[j]$ is the actual product of the M-residues $\bar{a}\bar{b} = aRbR \bmod M$, while the second product is responsible for the reduction $\bar{a}\bar{b}R^{-1} \pmod{M}$. It performs the reduction by adding such a multiple of M to the first product that the lowest word becomes zero and shifting right by w bits is possible. The right shift essentially is a partial division by R, which is a multiple of the word size (see Alg. 1).

Since both products are completely independent from one another, it is possible to build an MM unit that sports two separate unified multiplier cores. Computing the two products in parallel reduces the clock period significantly and also simplifies the control logic, since the multipliers are dedicated to one task.

5.3 Pipelining

Closer examination of the word based Montgomery algorithm shows that each iteration of the inner loop only works on a limited data set. More specifically, for any iteration $i = 0 \ldots e - 1$ the intermediate result $D[i - 1]$ is only dependent on the previous value of $D[i]$. While the output of the first iteration is all zero from the reduction, the second iteration produces the new value for $D[0]$, the third iteration computes $D[1]$, and so forth, until the inner loop is finished.

5.3.1 Pipelining of Multiple Arithmetic Units

In a software implementation on a processor with only one arithmetic unit $D[i - 1]$ would have to be stored in memory temporarily, until the next iteration of the *outer* loop. Since all memory operations impose a speed penalty on the algorithm, it is better if it can be avoided. In a hardware implementation where the number of available arithmetic units is in the hand of the designers, this is much easier to do. A couple of arithmetic units can be placed in a row, connecting the in- and outputs with the intermediate results, as in Figure 5.5. A new iteration of the outer loop is started whenever the first result of a preceding unit enters the next.

This process is called *pipelining* and apart from minimizing memory accesses its

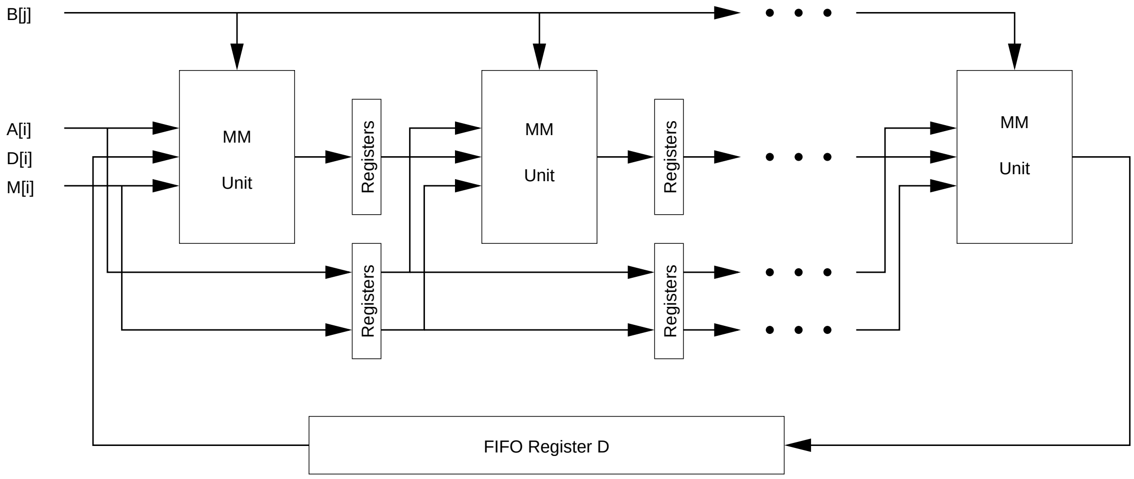

Figure 5.5: Pipelining of multiple arithmetic units

biggest advantage is that it also parallelizes the execution of an algorithm. For a pipeline depth of p arithmetic units or pipeline stages, p iterations are started one after another with a short delay, thereby overlapping them in time. Only when the first result $D[0]$ leaves the last stage, it has to buffered in memory until the first stage becomes available again, for a new round.

5.3.2 Pipelining within Arithmetic Units

Pipelining Montgomery's algorithm is not restricted to combining several arithmetic units alone. Albeit shorter, another pipeline can be created by splitting the datapath inside the unit into two separate stages, like shown in Figure 5.6. The first stage contains the two parallel multipliers, while the second stage adds the results of the multipliers and a shifted version of the previous result. Partitioning the task in such a way reduces the critical path to that of one multiplier. It also increases the number of clock cycles necessary for executing the inner loop by one. For a sufficiently large number of iterations in the inner loop, however, the effects of this additional cycle on the total delay become negligible and the overall benefit of having a shorter clock period prevails.

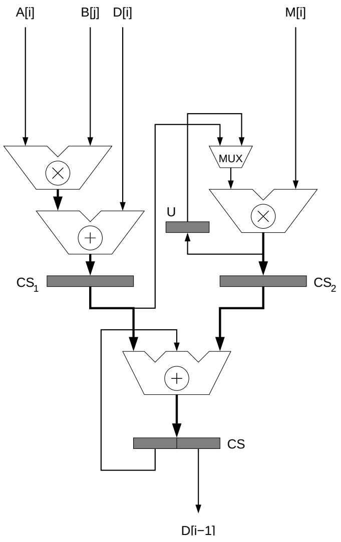

Figure 5.6: Pipelined arithmetic unit

As can be seen from Algorithm 3, the inner loop is preceded by an initialization step in which the value U is computed. Data dependencies that are present only during this initialization step, introduce a start-up latency of five clock cycles before the first value $D[0]$ is passed on to the next pipeline stage. Once initialization is complete, however, a new word $D[i]$ is computed with each cycle.

Chapter 6

System Architecture

In this chapter architectural details of an actual implementation of a unified high radix Montgomery multiplier design will be discussed. Some of these details have already been mentioned in the previous chapter, but only to a degree that was necessary to convey the ideas about differences to other architectures.

Since the algorithm after which this Montgomery multiplier was modeled, has already been explained in some detail, the following sections will present detailed views of the hierarchical building blocks in a bottom-up order. In other words, the core function blocks at the bottom of the hierarchy will be presented first, followed by the next higher blocks which are based around the core elements or combine them, and so on.

6.1 Unified Digit Multiplier

At the center of the architecture is the double core of unified digit multipliers. As indicated in the previous chapter the two multipliers compute the product and its reduction modulo M completely in parallel, once the initialization phase is over.

6.1.1 Partial Product Array

At one point during the initialization phase, which will be discussed a little later in this chapter, it is necessary to add the $2w$ bits wide product $A[i]B[j]$ and the w bits wide intermediate result $D[i]$ within the same clock cycle. Using a separate adder circuit to accomplish this would result in an increased clock period with a negative impact on performance. One alternative would be to add it one cycle later, but that would increase the start-up latency.

The best solution therefore is to incorporate the operand that has to be added into the partial product array of the digit multiplier. Since the delay only increases logarithmically with the height of the partial product array, one additional term in the array is insignificant. The only part of the multiplier affected by this modification is the column compression tree. The final adder remains entirely unchanged. On the other hand this "little trick" avoids the complexity of having another adder circuit with its own carry propagation related issues. The difference between a regular digit multiplier and one that incorporates an additional w bits wide word in the array is that the maximum depth of the extended array is increased by one and this difference becomes less significant as w increases.

6.1.2 Column Compression

The column compression layer of the unified digit multiplier is implemented using modified Wallace trees for each bit slice. These sum up the partial products using Carry-Save addition and produce the result in redundant sum and carry representation. Each level of the trees has a certain number of horizontal carries coming in from the neighboring bit slice and produces outgoing horizontal carries entering the next bit slice. The number of incoming and outgoing carries depends largely on which half of the partial product array the considered bit slice is in. For the first half in which the height h of the array increases from left to right, the incoming carries is $h-4$, or 0 for the first three bit slices. The number of outgoing carries is exactly one more $h-3$, except for the first two columns. For the second half of the array the height suddenly drops by two, because the additional w bits wide word only covers the lower half of the array, and continues to decrease linearly by one per column. Here the number of incoming carries is $h-1$, and that of outgoing carries $h-2$. As an example Figure 6.1 shows the Wallace tree for compression of the center column in an 8×8 bit multiplier's extended partial product array.

The Wallace trees for each bit slice have been designed by hand in a fashion that

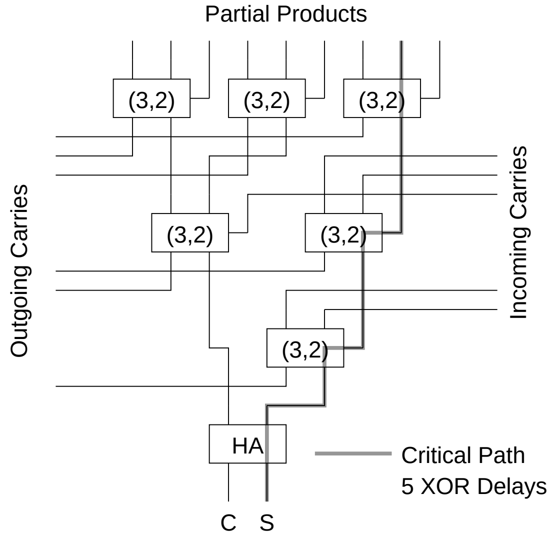

Figure 6.1: Wallace tree for reducing 9 PP's (5 horiz. carries)

follows the principles of the algorithm proposed in [OVL96]. This algorithm does not consider the influence of wire lengths on the delay of the tree, but since this thesis does not cover the entire design flow including place and route, it does not matter. Evaluation of the designs was based on the number of XOR delays that any given path contains. The path between the two inputs A, B and the Sum output of a full adder were counted as two XOR delays, while the path from the Carry input to the Sum output only contributes a single XOR delay. In cases where only two partial trees had to be connected, a half adder was used instead of a full adder to further reduce the path delay.

Even though this design methodology does not yield the optimal result for the binary polynomial case, it achieves the desired bound of $\mathcal{O}(\log_{3/2} w)$ XOR delays for integer multiplication. Nevertheless, polynomial multiplication is still faster than the integer case, since the final adder can be bypassed.

6.1.3 Selective Carry Propagation

In the case of polynomial multiplication the horizontal carries generated in any bit slice must not be passed on to the following columns. In order to control this behavior by a signal specifying the type of arithmetic, additional circuitry is necessary. As already described in the previous chapter, the simple solution is to insert rows of AND gates between bit slices that are controlled by the field selection signal F_SEL. Horizontal carries are propagated through the gates only if $F\_SEL = 1$. Otherwise the outputs will be zero.

Compared to the large number of gates that the multiplier consumes in its regular configuration, the increase in number of gates caused by the addition of this feature is irrelevant, yet the functionality of the whole circuit is improved dramatically.

6.1.4 Final Adder

The purpose of the Final Adder is to convert the redundant carry-save representation of the product back into non-redundant form. To do so it has to add the sum outputs from all columns but the first to all the carry outputs shifted left by one position. The first sum output from column 0 is directly taken as the final result. The carry vector is $2w - 1$ bits long and the sum vector $2w - 2$ bits. The final non-redundant result is of length $2w$.

The straightforward approach to implement this final adder would be to simply choose the fastest available adder circuit for $2w-1$ bits and place it after the column compression network. The usual choice of adders would include Carry-Select adders and members of the Carry-Look-Ahead family. The problem of this approach is, however, that the fast adder architectures consume a much higher number of logic gates per bit than simpler circuits such as Ripple Carry Adders.

The typical delay given for these fast adders relies upon the assumption that all input operands are available at the same time. However, if we take a closer look at the arrival time profile of the column compression outputs, we realize that the middle columns have the largest delay, while the columns at the edges arrive first. Based on this observation that were first made in [OVL96] we can build a hybrid final adder based on the arrival profile of the column compression layer.

For this purpose we partition the delay profile into a zone with growing delay, a plateau section where the delay is nearly constant, and a region of decreasing delay. For the first region we can use a simple ripple adder for as long as the carry ripple time is less than the delay of the compression tree. As soon as both delays approach the same value, a fast adder such as a Carry-Look-Ahead continues the addition and covers the rest of the plateau section and a little of the zone with decreasing arrival times. For the last section a simpler adder type can be used in conjunction with a Carry Select type of mechanism. Two simple adders of the same type can start adding up the same inputs, with the only difference being the value of the incoming carry. When the real carry in is available from the fast adder, it can be used to select the correct result using a multiplexer.

This technique helps reduce both the delay and the gate count of the final adder, by overlapping the final addition with column compression and reducing the size of the adder in the critical section of the delay profile.

6.2 MM Engine

The MM Engine is the combination of the two unified digit multipliers with a three input adder into a two stage pipelined block. The name was chosen, because this component delivers the computational power of the arithmetic unit, but it relies on control signals coming in from the outside, much like the engine of a car delivers the movement, but needs to be controlled by either the driver or the cruise control.

Each of the two digit multipliers is followed by a register. A *mul_enable* signal for each of these is used to control its behavior to either accept a new result from the multiplier at the time of a clock edge, or preserve its value. These two registers mark the boundary of the first pipeline stage of the engine.

In the second stage, a three input unified adder sums up the results of both multipliers of width $2w$ along with a third value, which is the feed-back of the upper $w+1$ bits of the previous cycle's result shifted to the right by w positions. The adder operates in two steps:

1. One row of half and full adders combine the three input operands into carry-save notation. Since one operand is only $w + 1$ bits long, half adders combine the upper $w - 1$ bits of the full-size operands, while full adders combine the common portion. The sum outputs of the adder cells now contain the pure XOR sum as necessary for the polynomial arithmetic mode.

2. In order to provide carry propagation as needed in the integer arithmetic mode, a carry select adder of width $2w$ adds the sum outputs with the carries shifted left by one bit position, much in the same way it was done with the multipliers' final adders. The only difference here is that the arrival of the operands is close to homogeneous and using a combination of different adders does not give any advantage.

The final $2w + 1$ bits wide result is selected to be either the sum outputs of the half/full adders in step 1, or the outputs of the carry select adder, depending on the *F_SEL* signal. Again, the added functionality requires only very little additional gates, in this case the multiplexers necessary for bypassing the carry select adder.

The critical path of the second pipeline stage is clearly shorter than that of the first stage, which includes the digit multiplier. Balancing the two stages in terms of delay would certainly be beneficial for achieving a higher clock frequency, but it is connected with a number of other problems, like additional latency during the initial computation of the parameter U. Another possibility might be to add a third pipeline stage by partitioning the multipliers into two balanced stages. Again this would mean increased latency during initialization, but it would also influence the clock period in a positive way. Such ideas have not been investigated any further yet, but could prove interesting for further research.

6.3 Pipeline Stage (MM Unit)

The MM Unit serves as a kind of shell for the MM Engine by adding a finite state machine controlling the data path. It also adds registers for holding the parameter U once it is computed, and for a local copy of the input $B[j]$ which does not change in the course of executing the inner loop of the algorithm. Another important element in the unit is the word counter. It is used for keeping track of the iteration count and changing states accordingly. The counter's start value is not fixed to any particular value, but has to be loaded once the unit is started up. This feature is crucial to the scalability of the design. If it were fixed to a specific value the design could not be used for any different operand sizes. This flexibility is very important in the context of algorithm agility, the possibility to run different algorithms with different sets of parameters and security settings.

Upon reset the MM unit enters the idle state *MM_IDLE*. This is the state when the unit is inactive. Apart from being the reset state, *MM_IDLE* is also entered after all inner loop iterations have finished. The only task for the unit in the idle state is to wait for a clock edge while the control signal $START\_NEXT$ is low. Then it enters the first of the three states in the initialization phase, *MM_INIT0*.

6.3.1 Initialization Phase

During the initialization phase of the unit, the control logic sets up the data path for computation of the parameter U, which is then stored in a local register for use throughout the inner loop.

For the three steps of this initialization phase only one multiplier is active at a time, due to data dependencies. Once completed, however, both multiplier cores are utilized in every cycle throughout the inner loop.

The three initialization cycles are as follows (CS_1 and CS_2 denote the double precision outputs from the multipliers, before they are added to form CS):

MM_INIT0 Multiplier 1 computes $CS_1 = A[0]B[j] + D[0]$

MM_INIT1 Multiplier 2 computes $U = S_1M' \pmod{2^w}$

MM_INIT2 Multiplier 2 computes $CS_2 = UM[i]$

These three steps only take place inside the first pipeline stage of the unit. CS_1

is kept inside the latch during steps 2 and 3. Thereby, one cycle later, CS_1 and CS_2 can be added together in the second stage of the unit, while at the same time the next set of inputs, $A[1]$, $B[j]$, $D[1]$ and $M[1]$, enters the first stage.

Note that the initialization step does not produce any output, yet. The idea behind Montgomery Multiplication is to add such a multiple of the modulus to the product, that the lowest word becomes zero. Thereby we can safely shift the result to the right without loosing any data, i.e. divide by R.

6.3.2 Execution Phase

During the last of the three initialization steps, when the unit is in state *MM_INIT2*, the unit loads the number of words to be computed into the counter, and with the next positive clock edge changes from initialization phase to execution phase. The unit then is in the state *MM_RUN*, where it produces a new intermediate result $D[i]$ with each clock cycle.

The first new output $D[0]$ is available after the second set of inputs, $A[1], M[1]$ and the former $D[1]$, has passed through the unit, i.e. after the second cycle of the execution phase has completed. The output can either be stored in a temporary register and reused in the following runs of the unit, or it can be fed into a subsequent unit that is started with a delay of five cycles. In the latter case it is necessary to have $D[0]$ available to the next unit during its first cycle.

However, the input to the next pipeline stage requires the second word $D[1]$ three cycles after $D[0]$, since the new unit goes through its own initialization phase of computing U in three steps. After that each subsequent word $D[i]$ again enters the next stage with every clock cycle.

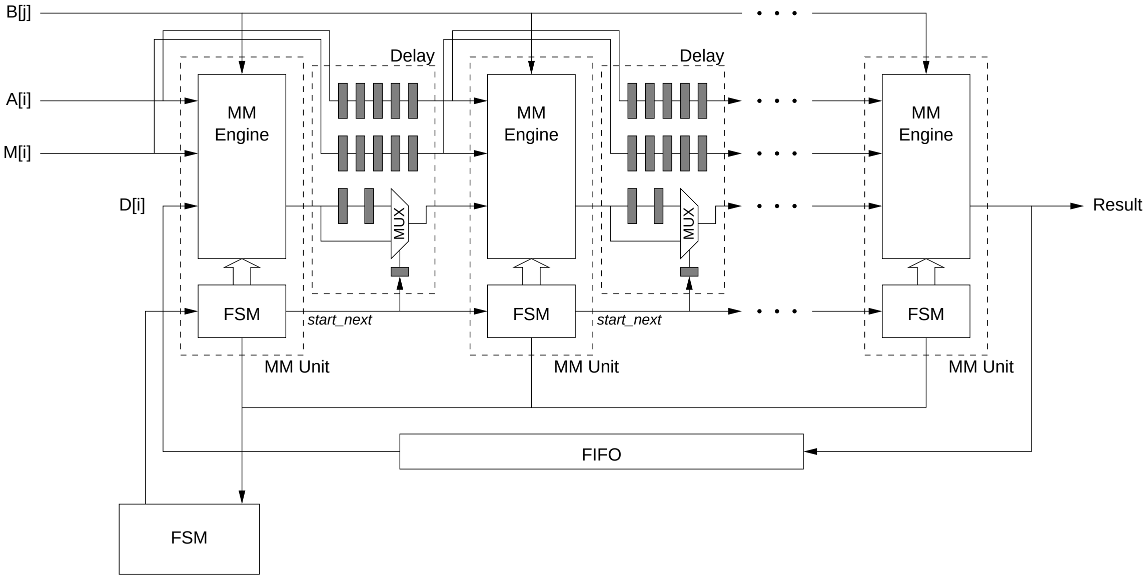

Figure 6.2: Pipeline structure

Another thing that should be noted is that the inputs $A[i]$ and $M[i]$ pass through each pipeline stage without change. This means they need only be read from memory once, for the first stage, and can then be passed from stage to stage with the right amount of delay. This greatly simplifies the memory access scheme and also reduces the load on those wires.

Based on these timing requirements, which are illustrated in Table 6.1, we define a group of delay registers to be inserted between neighboring pipeline stages. The portion responsible for the delay of $D[i]$ consists of 2 registers of width w. It additionally features a bypass multiplexer which directly passes $D[0]$ on to the next pipeline stage the instant it is available at the output of the previous stage. All the other words $D[i], i = 1 \dots e-1$ are delayed by the two registers. The remaining registers provide a constant delay of five clock cycles for the data path of $A[i]$ and $M[i]$, which branches off from the input to the first stage and is routed around the unit. A detailed view of the pipelining structure is provided in Fig. 6.2.

6.4 FIFO Buffer Queue

The task of the FIFO buffer queue depicted in figure 6.2 is to act as cyclic temporary storage for the intermediate result words $D[i]$. During the course of the algorithm these are being worked on over and over again, until the final result is ready. Since the number of words is variable, there is no way to tell how long a particular word needs to be delayed before it can be fed back into the first pipeline stage. Therefore a FIFO of sufficient storage capacity is necessary. What size is sufficient depends on the maximum operand length the Montgomery multiplier is supposed to handle, as well as the word size and number of pipeline stages of the design.

From the pipeline timing in table 6.1 it is clear to see how for each pipeline stage $D[0]$ leaves the unit at the time $D[3]$ enters it. Hence there are three words $D[i]$ in

	Stage 1			Stage 2			Stage 3		
Clk	in	temp	out	in	temp	out	in	temp	out
0	$A[0], B[0], D[0]$								
1	M'	S_1, CS_1							
2	$M[0]$	U							
3	$A[1], M[1], D[1]$	CS_2							
4	$A[2], M[2], D[2]$	CS_1, CS_2	0						
5	$A[3], M[3], D[3]$	CS_1, CS_2	$D[0]$	$A[0], B[1], D[0]$					
6	...	CS_1, CS_2	$D[1]$	M'	S_1, CS_1				
7		...	$D[2]$	$M[0]$	U				
8			$D[3]$	$A[1], M[1], D[1]$	CS_2				
9			...	$A[2], M[2], D[2]$	CS_1, CS_2	0			
10				$A[3], M[3], D[3]$	CS_1, CS_2	$D[0]$	$A[0], B[2], D[0]$		
11				...	CS_1, CS_2	$D[1]$	M'	S_1, CS_1	
12					...	$D[2]$	$M[0]$	U	
13						$D[3]$	$A[1], M[1], D[1]$	CS_2	
14						...	$A[2], M[2], D[2]$	CS_1, CS_2	0
15							$A[3], M[3], D[3]$	CS_1, CS_2	$D[0]$
16							...	CS_1, CS_2	$D[1]$
17								...	$D[2]$
18									$D[3]$
19									...

Table 6.1: Pipeline organization and timing

each unit, if the unit is in its execution phase. Between neighboring units there are chains of delay registers for timing the data. The chain responsible for delaying $D[i]$ is exactly two registers long. This means that a fully working pipeline in which all p units are in execution phase can hold a maximum of $N_{p_{max}}$ words of $D[i]$, where

$$N_{p_{max}} = 2(p-1) + 3p = 5p - 2. \tag{6.1}$$

The number of words D that are being processed the unit is $e = \left\lceil \frac{N}{w} \right\rceil$, where N denotes the operand size in bits. In the beginning of the computation, all words of the temporary result are zero, so they do not have to be stored explicitly. Only after having passed through the pipeline at least once, they have to be temporarily stored in the FIFO, until the first stage is free again. Therefore the maximum number of words that must fit inside the FIFO is exactly the difference between the total number of words and the maximum number of words inside the pipeline.

$$N_{FIFO} = e - N_{p_{max}} = \left\lceil \frac{N}{w} \right\rceil - (5p - 2) \tag{6.2}$$

To be on the safe side and to also allow for extra precision in case the need arises, this value should be only considered as the absolute bare minimum of storage space in the FIFO. It is recommended to allow for extra space when designing the FIFO.

Chapter 7

Implementation

This chapter discusses the design process that has been followed in the course of realizing the Montgomery multiplier. The typical flow for custom digital ASIC design has been covered only partially, simply because it was not the purpose of this work to produce actual silicon. Rather it can be viewed as a proof of concept that the architecture is feasible for an efficient hardware implementation of Montgomery's algorithm.

7.1 Design Methodology

An algorithmic description of a particular version of Montgomery's method, the *Finely Integrated Operand Scanning (FIOS)* method, in pseudo-code served as the basis for the hardware. It was thoroughly analyzed for potential parallelization. Different arrangements of arithmetic units and data paths have been investigated in the process.

In order to implement the arithmetic cores like the digit multipliers and adders efficiently, it was necessary to investigate the latest findings in integer multiplier design. Books on Computer Arithmetic, e.g. [Kor93] and [FO01], were a good starting point, that was augmented by further literature search on the subject. The IEEE Xplore database proved to be an invaluable tool for this purpose, as well as the Google World Wide Web search engine.

Once the architecture was specified, the components of the design were described the in structural *VHDL* code. This approach makes the performance of the design less dependent on the synthesis features of the VHDL compiler suite. Only the components containing a finite state machine were expressed in a mixture of structural and algorithmic description and state machine encoding was left to the compiler.

7.1.1 Functional Verification

At several times during the design process, the correct function of the components has been verified using a VHDL testbench. A testbench essentially is a piece of behavioral VHDL code without any signals to the outside, that instantiates the component that is to be tested, also called *Device Under Test (DUT)*, feeds specific data to its inputs (test vectors or patterns) and reads back the results, comparing them to the expected results.

Testing a complex computer arithmetic architecture presents a special challenge to the designer of a testbench. The test patterns that have to be generated can not just simply be random vectors, but need to make sense in terms of the computation that is to be performed. Next to the input test patterns there also have to be the corresponding output vectors for comparison. In addition to the correct values, the correct timing of the data, with respect to the clock cycle in which it enters or leaves the circuit, is also very important. The architecture consists of many stages of logic and intermediate register stages, therefore the output of results does not necessarily coincide with the input of test vectors.

Functional verification does not take into account the different critical path delays a design has. The critical path delay is directly related to the maximum clock frequency a design can run with, and will only be determined at the time of synthesis.

7.1.2 Synthesis

After the design has been successfully verified, the next step is to synthesize it into a net list using elementary gates provided by the technology vendor. For the purposes of this thesis the 0.5μ AMI_05 CMOS technology from Mentor Graphics' *ASIC Design Kit (ADK)* [Gra] was found to be adequate, since it allows for a good comparison of this design with the previously proposed ones.

Exemplar Logic's LeonardoSpectrum has been used to synthesize the design into a net list and to produce preliminary timing estimates. Preliminary, because the synthesis tool does not take care of placement and routing of the elementary gates, so the exact wire delay from interconnects cannot be determined. Still, the critical path delay reported by the synthesis tool can give a good indication of what the expected performance of the design will be like, since a linear wire-delay model adds a global amount of wire delay based on the area of the design. At feature sizes at and above 0.5μ wire delay is not such a critical issue as long as most wires are local and kept short, which is the case in most of this design.

The result of the synthesis are typically two files, (1) structural VHDL code of the flattened design based on the components of the specific technology, and (2) timing information in Standard Delay Format (SDF). For every basic elementary gate in the design the SDF file contains the path delay for every possible path between input and output ports. The structural VHDL output is usually used in conjunction with an SDF file to perform back-annotated timing simulation.

7.1.3 Back-Annotated Timing Simulation

The synthesized design is run through the functional simulation again, using the same testbench as before, only this time timing information is provided in form of the SDF file and the delay of every signal can be inspected. In this step timing related problems can be detected much better than in a pure functional simulation. It is also useful for verifying the maximum clock frequency reported by the synthesis tool.

7.2 Test Pattern Generation

The test vectors for the testbench were created using Maple V. It first generates a random set of numbers of length N bits which are then transformed into the Montgomery Residue Number System (RNS). Then the numbers are split into single words of the size w and stored in an array and in the testbench stimuli file. The stimuli file contains all the necessary input words in the right order as the design expects them. This file is read by the testbench using VHDL file operations, and the test stimuli are fed to the top level design.

The Maple testbench generator program continues by performing the exact same algorithm 3 that is the basis for the architecture. Thus all steps produce exactly the same intermediate results that are expected from the design. These intermediate results are written out into a different file for comparison with the simulation results that occur during computation of the Montgomery product. This makes the debugging process much easier and enables the verification of the interaction between pipeline stages. At the current time the results are still inspected manually, but an automatic testbench verification is possible.

Chapter 8

Results

This chapter presents and discusses the results obtained from the synthesis tools and from extrapolation for a number of different parameter settings. Specifically this means delay and area efficiency for several configurations with different choices of wordsize w, numbers of pipeline stages p and optimization efforts.

Since the design has so far only implemented for the word sizes $w = 4$ and 8, the timings for $w = 16$ and 32 have been extrapolated. The equation used for extrapolation, Eq. 8.3, takes into account the individual contribution of each separate component to the overall delay, based on the word size w. These delay characteristics have been taken directly from established literature [Kor93] [FO01] and are believed to be acurate. For example, the error between estimated and reported critical path for word sizes $w = 4$ and 8 is less than 3%, so that the estimated critical path delay for $w = 16$ and 32 is considered to lie within an acceptable margin of error.

The following performance evaluation will discuss different aspects of the design's capabilities by not only presenting the speed in different configurations, but also by taking area consumption into account.

8.1 Performance Evaluation

Performance of an arithmetic architecture can be evaluated in a multitude of ways, depending on its expected use. The most common performance evaluation certainly is to measure the time necessary for computation of a benchmark test. The rules are simple, the faster a design performs, the better it is. However, there are cases in which this rule does not apply. Certain applications and environments in which low power consumption and/or a small footprint on the chip area are regarded more important than pure number crunching capabilities, the *time* × *area product* has become a vital measure of comparability. With the growing popularity of mobile handheld appliances and increased deployment of smartcards this method of evaluation is on the rise, while simple benchmarks, purely focused on speed, tend to be important to high-performance computing applications only.

The overall speed of the architecture in this thesis is dependent on the number of clock cycles needed to perform a multiplication for a given operand size, but also on the clock period that can be achieved. While the former is purely a parameter of the configuration (w, p), the latter is also dependent on the chosen technology and optimization efforts of the synthesis tool.

Interestingly enough the clock period does not seem to be dependent on the number of pipeline stages p in the design. This appears logical since the data path of each arithmetic unit ends in a set of registers which represents a boundary for the critical path. Exact replication of the same unit over and over again therefore should not have significant influence on the clock period. This observation, however, was made without taking influences of wire delays into consideration, so the real impact of the parameter p is unknown. Nevertheless, the potential number of long wires is limited to certain control signals and the clock tree, and compensation for these effects is possible.

8.1.1 Influence of Pipelining on Performance

This first part of the performance analysis focusses on the influence of pipelining on the performance of the architecture in connection with the word size of the data path. The direct influence of the word size on the speed of the architecture, however, is not considered here, only its saturating influence on pipeline effectiveness. The number of pipeline stages p and the word size w of the system data path are put into context with the total number of clock cycles Z_{clk} necessary for performing a single Montgomery multiplication with N bits of precision.

The number of clock cycles Z_{clk} can be expressed as in the following equation:

$$Z_{clk} = \left\lceil \frac{N}{wp} \right\rceil \left(\left\lceil \frac{N}{w} \right\rceil + 5 \right) + 5 \left(\left\lceil \frac{N}{w} \right\rceil + (p-1) \bmod p \right) - 1 \tag{8.1}$$

The equation can be broken down in the following way:

1. $\left\lceil \frac{N}{wp} \right\rceil$ is the number of times the intermediate result cycles through the pipeline to complete the outer loop of the algorithm

2. $\left(\left\lceil \frac{N}{w} \right\rceil + 5 \right)$ denotes the number of clock cycles used for the inner loop including the five clock cycles initial latency

3. $5 \left(\left\lceil \frac{N}{w} \right\rceil + (p-1) \bmod p \right)$ accounts for the additional cycles spent on pipeline delay

4. -1 is the final stage correction

Note that for some values of p the utilization of the pipeline starts to saturate, i.e. when the complete operands fit into the pipeline. The first pipeline stage then has to wait until the last stage starts producing output again. This point of saturation is reached when the condition $5p > \left\lceil \frac{N}{w} \right\rceil + 5$ is fulfilled. In that case the number of clock cycles is increased by the number of unused or idle clock cycles Z_{idle} which can be expressed as

$$Z_{idle} = \left\lfloor \frac{N-1}{wp} \right\rfloor \left(5p - \left\lceil \frac{N}{w} \right\rceil - 4 \right) \tag{8.2}$$

An example with real numbers helps to illustrate the impact that additional pipeline stages have on reducing the total number of clock cycles. Two different operand sizes of real world importance have been selected to give a further hint of how the architecture is expected to perform in a given application. Two tables give the number of clock cycles for a given pipeline depth.

The first case with $N = 256$ bits (table 8.1) corresponds to an application involving an *Elliptic Curve Cryptosystem* with a high security setting. The second example with $N = 1024$ bits (table 8.2) is relevant to modular exponentiation based systems like the *Digital Signature Algorithm (DSA)* or the *RSA* cryptosystem, with a medium level of security. More information on the relationship between operand size and security level of different cryptosystems can be found in [LV00].

p	1	2	3	4	5	6	7	8	9	10
$w = 4$	4415	2212	1517	1118	911	773	689	586	551	497
$w = 8$	1183	596	411	310	263	226	199	194	194	194
$w = 16$	335	172	125	98	98	97	97	96	96	96
$w = 32$	103	56	49	48	48	48	48	47	47	47

Table 8.1: Number of clock cycles for 256-bit operands

The effect of pipeline saturation is nicely visible for large values of w in table 8.1. For relatively short operand sizes like $N = 256$ it therefore makes less sense to have

p	1	2	3	4	5	6	7	8	9	10
$w = 4$	66815	33412	22445	16718	13571	11237	9671	8386	7583	6810
$w = 8$	17023	8516	5723	4270	3467	2930	2531	2162	1999	1763
$w = 16$	4415	2212	1517	1118	911	773	689	586	551	497
$w = 32$	1183	596	411	310	263	226	199	194	194	194

Table 8.2: Number of clock cycles for 1024-bit operands

a long pipeline in conjunction with a wide data path. For the second case with $N = 1024$, however, the advantage of a long pipeline is obvious, since the speed-up factor is almost directly proportional to the number of stages.

8.1.2 Influence of the Word Size on Performance

The performance of the architecture is dependent on the word size of the system data path in two different ways. One of them is apparent from the structure of the parallel digit multiplier core. The partial product array is basically a skewed matrix of $w \times w$ bits. If w increases, this means that more partial products need to be added together, thereby adding to the critical path. By using a tree based column compression technique, the critical path only grows logarithmically with w.

The second dependence on w is not as apparent, and is hidden in the final adder that is responsible for carry propagation. As in the case of column compression the delay introduced by the addition techniques used in the final adder grows logarithmically with w.

The total delay the multiplier is composed of consists of (1) the delay of partial product generation t_{pp}, (2) multiplier column compression t_{cc}, (3) final addition t_{fa} and (4) multiplexing according to field selection t_{mux}. The result of the multiplication needs to be stored in a register, therefore we also have to take into account the setup time and clock to Q propagation delay of a typical D-Flipflop. For the total critical path delay in the case of $GF(p)$ this gives us

$$\begin{aligned} t_{cp} &= t_{pp} + t_{cc} + t_{fa} + t_{mux} + t_{FF} \\ &= t_g(2 + 4\left\lceil \log_{3/2} w \right\rceil + 4(\left\lceil \log_3 w \right\rceil - 1) + 4) + t_{FF} \end{aligned} \tag{8.3}$$

Thus, for a technology dependent average gate delay of $t_g = 0.6ns$ and $t_{FF} = 1.5ns$ (AMI 0.5μ slow CMOS process) we can estimate the critical path delay to be 21.9 ns for a word size of $w = 8$. Similarly, after applying the equation above for $w = 16$, we get an estimate of 26.7 ns for the total delay. Please note, that in the case of $GF(2^n)$ the final adder is not used, so there is a performance improvement of t_{fa} which will allow us to run the circuit at an even higher speed.

Equation 8.3 was found to approximate the clock period in dependence on the word size w with appropriate accuracy for extrapolation purposes. Using the actual timing reports from implementations for $w = 4$ and 8 the error between delay reported by the synthesis tool and delay estimated using equation 8.3 was determined as:

For $w = 4$:

$$T_{synth} = 16.72\text{ns}$$

$$T_{est} = 17.10\text{ns}$$

$$\Delta = \frac{|T_{synth} - T_{est}|}{T_{synth}} = 2.3\%$$

For $w = 8$:

$$T_{synth} = 22.47\text{ns}$$

$$T_{est} = 21.90\text{ns}$$

$$\Delta = \frac{|T_{synth} - T_{est}|}{T_{synth}} = 2.5\%$$

The error between estimated and actual reported clock period is less than 3%. This is good enough for using the equation to extrapolate the clock period for word sizes that could not be implemented as part of this thesis.

Design	Clock period [ns]
$w = 4$ (Speed opt.)	16.72
$w = 4$ (Area opt.)	23.64
$w = 8$ (Speed opt.)	22.47
$w = 8$ (Area opt.)	32.57
$w = 16$ (extrapol.)	26.7
$w = 32$ (extrapol.)	33.9
$w = 64$ (extrapol.)	38.7

Table 8.3: Clock periods for different word sizes

Table 8.3 lists the clock periods found, either empirically or by extrapolation, for different sizes w and optimizing strategies. By optimizing for little area instead of

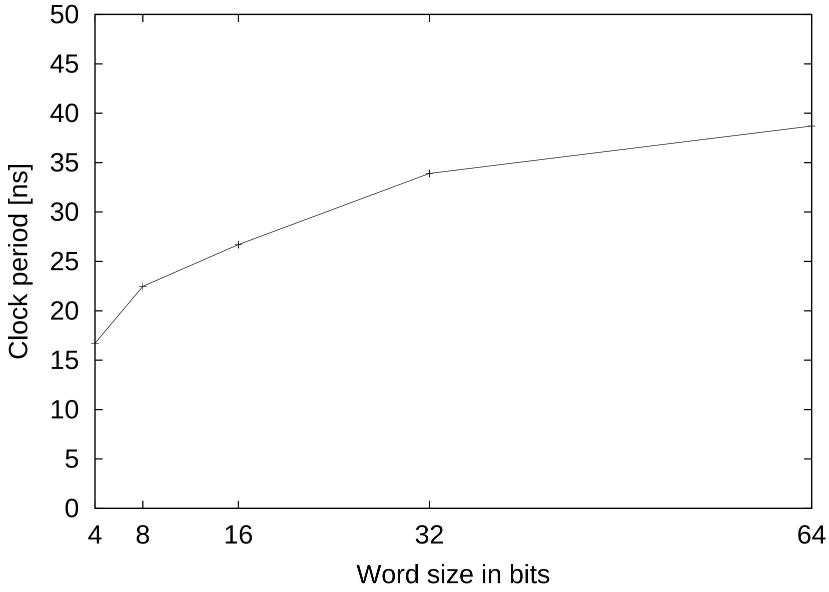

Figure 8.1: Clock period with respect to word size

for minimal delay, the synthesis tool tries to re-use as many gates as possible, so that the total number of gates is decreased. This strategy sacrifices speed over chip area and might be the ideal choice in cases where the design has special low-power requirements, as in mobile applications. The graph in figure 8.1 shows nicely how the clock period only increases logarithmically with the word size.

Another important benchmark by which the efficiency of a design can be measured is the amount of area it consumes. Table 8.4 lists the area requirements of the design with varying data path word sizes for different numbers of pipeline stages. For $w = 16$ and 32 the required area is unknown since those word sizes have not been implemented yet. What is known, however, is the general structure of the architecture, which can be used in extrapolating the anticipated chip area. Based on the reported areas from implemented designs and a concept of how the word size influences the area of particular components of the architecture, equation 8.4 was used for estimation purposes. For a fixed value of w the area is almost exactly linearly dependent on p, as is evident from figure 8.2.

$$A_{MM}(w,p) = 25 \cdot w^2 + (306 \cdot p - 77.5)w + 190 \cdot p + 300 \tag{8.4}$$

Design Area (equivalent gates)				
p	Word size w			
	4	8	16 (est.)	32 (est.)
1	1785	3901	10546	33402
2	3619	8159	22032	68984
4	7251	16632	45004	140148
8	14502	33584	90948	
16	29014	67488		
32	58038			

Table 8.4: Area for different word sizes

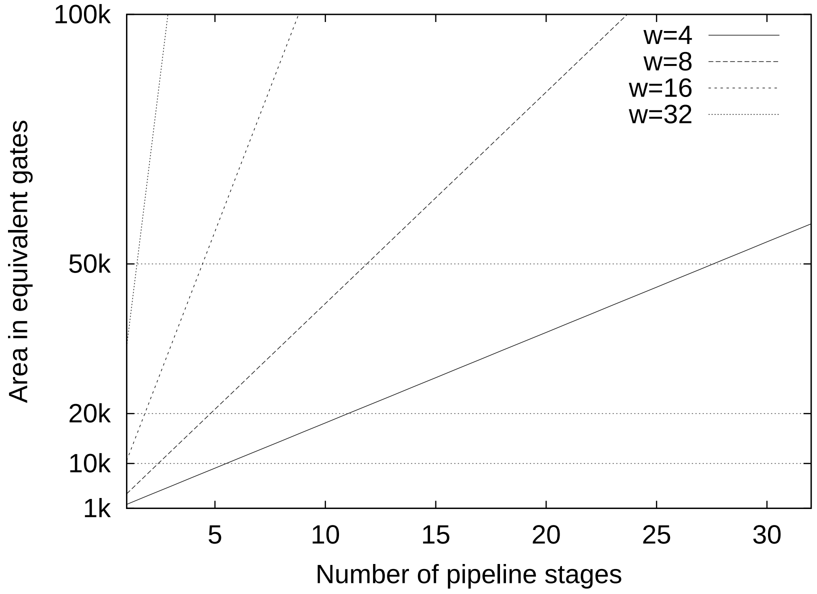

Figure 8.2: Area requirements of different (w,p) configurations

8.2 Analysis of Results

Using the clock periods of a couple of design choices from table 8.3 and the number of clock cycles from tables 8.1 and 8.2, the total time necessary to compute a Montgomery product for precision N is easily computed. The results have been plotted to a graph and included as Figure 8.3 and Figure 8.4.

The influence of pipelining is clearly visible in the hyperbolic shape of the curve. As expected, designs with a larger radix perform superior compared to those with a small w. At the same time, however, the speed improvements through pipelining saturate much faster, so that adding more stages does not produce further speed-up.

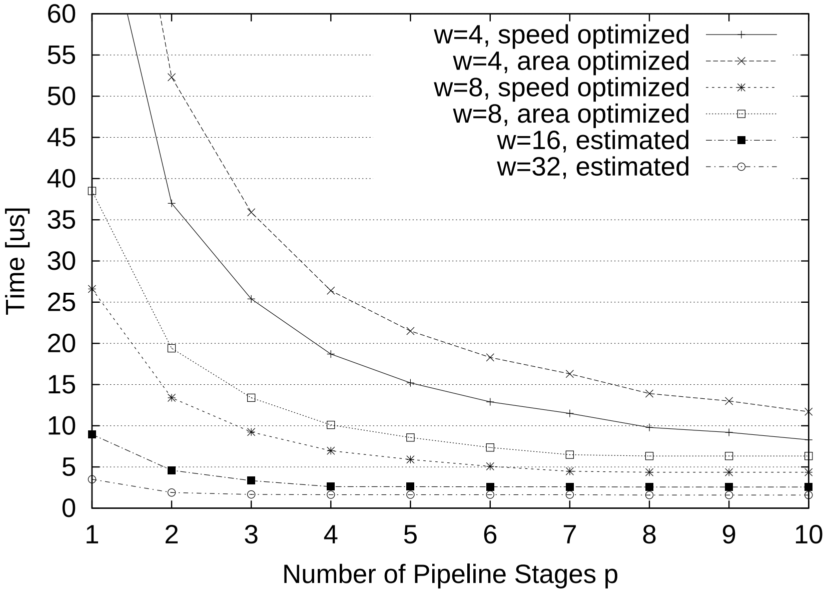

Figure 8.3: Total time for 256 bit operands

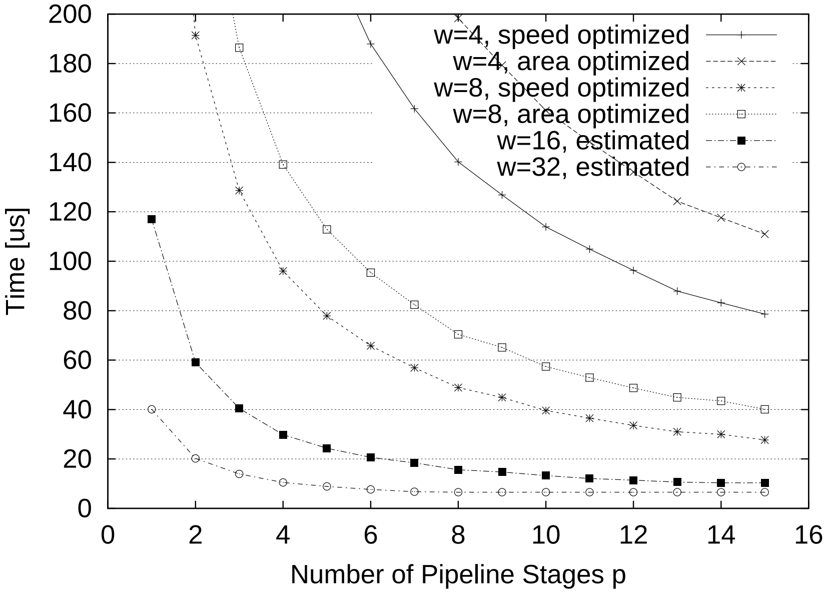

Figure 8.4: Total time for 1024 bit operands

8.2.1 Speed

The maximum speed that can be achieved with the new architecture in certain configurations is able to compete with and outperform the previously proposed bitserial and Radix-8 Montgomery based designs. The minimum total time needed for a 256 bit Montgomery product sinks below $2\mu s$ in configurations where w is high. Unfortunately this does not necessarily mean that the design is preferable over a bitserial design in every case. The area consumption increases dramatically with the word size. In order make a valid statement about the efficiency of a configuration, the time × area product needs to be evaluated for each configuration.

8.2.2 Time × Area Product

The time × area product gives an idea about how much chip area needs to be sacrificed for a certain targeted speed. Using the mathematical toolkit Maple and the formulae for time and area requirements developed so far, the values of the time × area product for all configurations $w = 4 \dots 32, p = 1 \dots 20$ have been computed for $N = 256, 1024$ and sorted by ascending area requirements. The resulting graphs in figure 8.5 and figure 8.6 give a clear overview of the tradeoff that is involved.

In the first graph for $N = 256$ we can see that the area necessary for achieving a time of less than $2\mu s$ is close to 70,000 equivalent gates. This number of gates is much too high for applications with limited available area. If we settle for a little

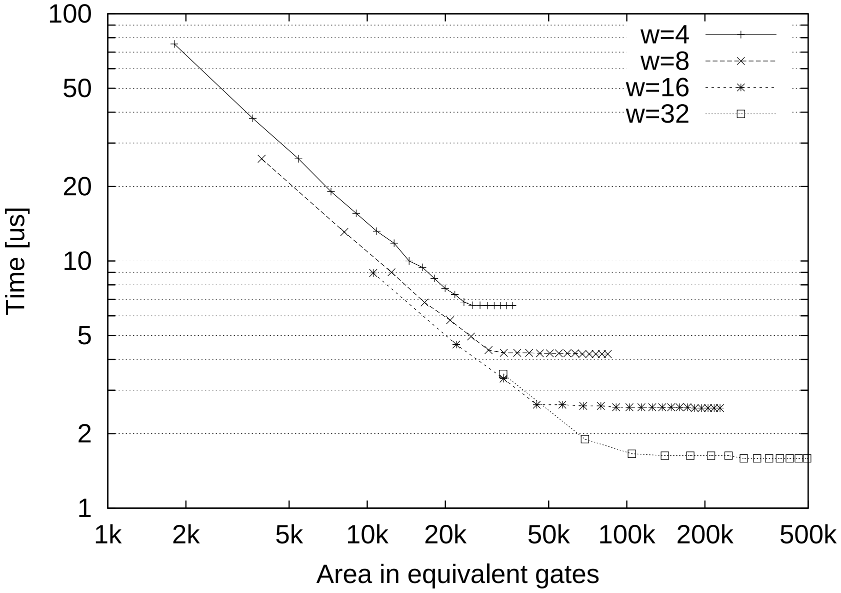

Figure 8.5: Time - area tradeoff for 256 bit operands

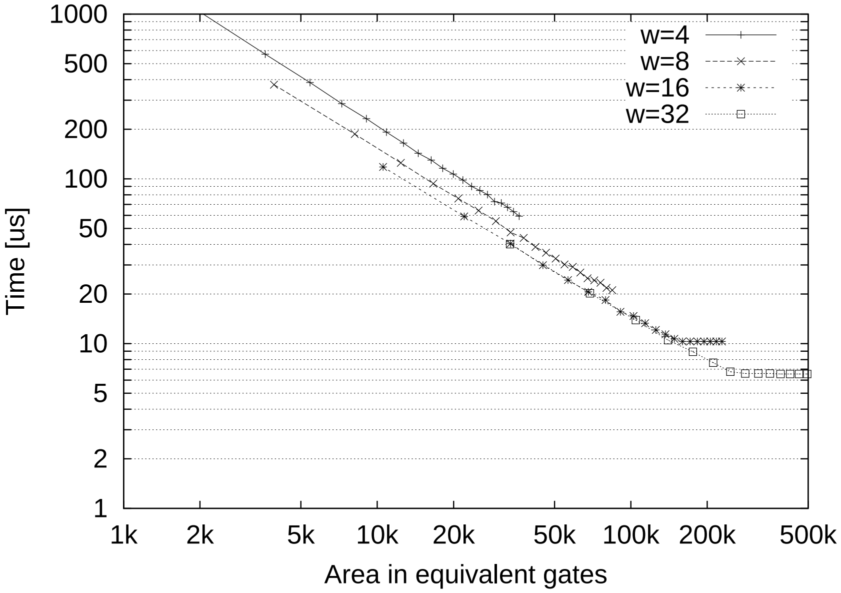

Figure 8.6: Time - area tradeoff for 1024 bit operands

less performance, such as 4.5μs, the area requirements shrink to 22,000 gates. This is still a lot when compared with the timings from the bitserial design which reaches its maximum speed at 3.2μs for 256 bits, albeit at area costs of about 13,000 gates. We see a similar pattern in the graph for $N = 1024$, where our design reaches an execution time of 40μs with an area of 33,000 gates. It does not matter much that the absolute minimum time achieved with our design is 6.6μs, about twice as fast at almost seven times the area.

A couple of remarks on our design:

- The word size w influences both the width and, most importantly, the depth of the digit multipliers. The radix-8 design presented in [TTK01] has a fixed depth of three bits and only varies the width. Therefore a larger word size does not influence the clock period as much as in our design, and more bits can be processed in the same clock cycle at a higher speed. Making the word size independent of the multiplier depth is one item to be investigated in further research.

- Except for the bitserial design in [STK00] the two other architectures can not handle binary polynomial arithmetic. While its support does not contribute significantly to the latency, it does not come for free in terms of area cost. Our design is able to handle both types of arithmetic, and offers a huge potential of further flexibility. At the time of writing this thesis an attempt [O'R02] is under way to extend this architecture to additionally handle the NTRU cryptosystem's *Star Multiplication* of polynomial rings.

- The bitserial and the radix-8 design both defer carry propagation until the very end of the algorithm, while our design fully propagates the carries in each clock cycle. This has a negative effect on the clock period and therefore is the prime target for future optimization.

- In contrast to the previous designs, where the start-up latency of each pipeline stage is two clock cycles, our architecture has a latency of five. This in effect reduces the potential for parallelization through pipelining. The same goes for the higher radix due to a high w. The number of words $e = \lceil \frac{N}{w} \rceil$ that are processed in the inner loop of the algorithm is inversely proportional to the word size, thereby reducing the maximum pipeline length.

Chapter 9

Conclusions

A new architecture for modular multiplication based on Montgomery's algorithm has been proposed, which combines positive features from previously proposed architectures with recent advances in digit multiplier design. The result is a highly scalable design with the ability perform integer and binary polynomial modular arithmetic at high speeds.

Analysis of previous work proved to be essential and forms the foundation for the proposed new architecture. The most important result of this analysis was to identify the bit serial approach to multiplication as the limiting factor. By basing our contribution on bit-parallel digit multipliers we tried to get around this limitation and achieved promising results.

Although the design is still in its early stages and a number of improvements are necessary to reach peak performance, the point has been made that high radix multiplication is a viable alternative to bit serial multiplication. The architecture has been implemented in VHDL, synthesized and tested successfully. Preliminary timing analysis shows that even at this early stage the performance is acceptable. The main bottlenecks have been identified and can thus be tackled with an improved design.

The flexibility of the architecture stems from the fact that the radix and therefore the word size of the data path is not limited, and neither is the number of pipeline stages. This makes the design fit in a large number of application contexts ranging from high performance network processor down to hand held appliances. Once the performance bottlenecks have been removed, the architecture is a real contender.

9.1 Further Research

As mentioned earlier the architecture presented in this thesis is an early prototype and there are some issues left that need to be addressed to improve performance:

- Making the word size and therefore the multiplier width independent from the multiplier depth to improve the throughput,
- Deferring carry propagation until the very end of the algorithm by deploying carry save addition as much as possible
- Reducing the start-up latency of the initialization step to improve the pipeline effectiveness

The ultimate goal of our research is to create a flexible hardware solution capable of addressing the current and future needs for security of information. As advances are being made in the field of cryptanalysis, the security parameters of cryptographic systems, such as key sizes, but also complete algorithms, need to adapt. To protect investment in the infrastructure of information systems, scalable and flexible architectures become increasingly important. The presented research is one of the first steps towards this goal, but many more must follow.

A number of primitives for various types of arithmetic need to be identified and implemented efficiently in an *algorithm agile* security processor. The list includes, but is not limited to

- modular squaring of integers and polynomials
- modular inversion
- modular convolution of polynomials
- new types of operations, e.g. data dependent permutations, etc.

Bibliography

[Ber01] D. J. Bernstein. Circuits for integer factorization: A proposal. http://cr.yp.to/papers.html#nfscircuit, 2001.

[Boo51] A. D. Booth. A Signed Binary Multiplication Technique. *Quarterly Journal of Mechanics and Applied Mathematics*, pages 236–240, June 1951.

[DH76] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, 1976.

[FO01] M. Flynn and S. Oberman. *Advanced Computer Arithmetic Design.* John Wiley & Sons, INC., New York, 2001.

[Gra] Mentor Graphics. ADK HTML Data Book AMI 0.5 Micron. http://www.mentor.com/partners/hep/AsicDesignKit/dsheet/ami05-databook.html.

[Gro00] J. Großschädl. High-Speed RSA Hardware Based on Barret's Modular Reduction Method. In Çetin K. Koç and Christof Paar, editors, *Work-*

shop on Cryptographic Hardware and Embedded Systems — CHES 2000, volume LNCS 1965, pages 191–203, Worcester, Massachusetts, USA, August 17–18 2000. Springer-Verlag.

[Gro01] J. Großschädl. A Bit-Serial Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^m)$. In Ç. Koç, D. Naccache, and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2001*, pages 206–223, Paris, France, May 2001. Springer-Verlag.

[KA98] Ç. K. Koç and T. Acar. Montgomery Multplication in $GF(2^k)$. *Design, Codes, and Cryptography*, 14(1):57–69, 1998.

[KAK96] Ç. K. Koç, T. Acar, and B. Kaliski. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, pages 26–33, June 1996.

[Kor93] I. Koren. *Computer Arithmetic Algorithms.* Prentice-Hall, 1993.

[LV00] A. Lenstra and E. Verheul. Selecting Cryptographic Key Sizes. In Hideki Imai and Yuliang Zheng, editors, *Third International Workshop on Practice and Theory in Public Key Cryptography — PKC 2000*, volume LNCS 1751, Berlin, 2000. Springer-Verlag.

[Mon85] P. L. Montgomery. Modular Multiplication without Trial Division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[MvOV97] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Boca Raton, Florida, USA, 1997.

[O'R02] C. M. O'Rourke. Efficient NTRU Implementations. Master's thesis, ECE Department, Worcester Polytechnic Institute, Worcester, Massachusetts, USA, May 2002. Work in Progress.

[OVL96] V. G. Oklobdzija, D. Villeger, and S. S. Liu. A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach. *IEEE Transactions on Computers*, 45(3):294–306, March 1996.

[STK00] E. Savaş, A. F. Tenca, and Ç .K. Koç. A Scalable and Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^m)$. In Çetin K. Koç and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2000*, pages 277–292, Berlin, Germany, LNCS 1965 2000. Springer-Verlag.

[TK99] A. F. Tenca and Ç. K. Koç. A Scalable Architecture for Montgomery Multiplication. In Ç. Koç and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 1999*, volume LNCS 1717, pages 94–108, Worcester, Massachusetts, USA, August 1999. Springer-Verlag.

[TTK01] A. F. Tenca, G. Todorov, and Ç. K. Koç. High Radix Design of a Scalable Modular Multiplier. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2001*, pages 189–205, Paris, France, May 14–16 2001. Springer-Verlag.

[Wal64] C. S. Wallace. A Suggestion for a Fast Multiplier. *IEEE Transactions on Electronic Computers*, pages 14–17, February 1964.