Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl

2003

MSc THESIS

Hardware Acceleration of the SRP Authentication Protocol

Peter Theodoor Groen

Abstract

CE-MS-2003-15

In Wireless Local Area Networks (WLANs), data is transferred through the ether. To prevent that unauthorized users read or modify data, security is needed, not only to secure data transmission but also to authorize users and machines to the network. A secure log-on procedure can be created with the help of cryptographic functions. However, these functions are often computationally intensive which results in a slow log-on procedure. This work focuses on the design of the hardware acceleration of a WLAN authentication proceduce. The hardware acceleration is used in the Secure Remote Password (SRP) authentication protocol, but can be used in other security related computations as well. The experimental platform is TUTWLAN, a WLAN platform being developed at Tampere University of Technology which runs on the Altera Excalibur development board that contains a chip with programmable hardware and a microprocessor. On a microprocessor, approximately 99 percent of the execution time of the SRP protocol is consumed by one single function: the modular exponentiation. Thus a significant speedup can be obtained by implementing this function in hardware. In this work the exponentiation function is implemented in the Altera programmable logic and used in the SRP protocol running on the microprocessor. In addition, proposals for further hardware speedup of the exponentiation design are presented. The results show that a full modular exponentiation with inputs of 1024 bits can be performed in less than 40 ms using less than 10.000 logic elements (i.e., 4-input lookup tables and registers) on the Excalibur programmable logic. By using the implemented hardware accelerator in the SRP protocol, the execution time is reduced by a factor of 4. An additional factor of 5 speedup (totaling a factor of 20) is possible by implementing the fastest design proposed in this work.

Institute of Digital and Computer Systems

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Hardware Acceleration of the SRP Authentication Protocol

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL ENGINEERING

by

Peter Theodoor Groen
born in Leidschendam, Netherlands

Computer Engineering Laboratory
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Hardware Acceleration of the SRP Authentication Protocol

by Peter Theodoor Groen

Abstract

In Wireless Local Area Networks (WLANs), data is transferred through the ether. To prevent that unauthorized users read or modify data, security is needed, not only to secure data transmission but also to authorize users and machines to the network. A secure log-on procedure can be created with the help of cryptographic functions. However, these functions are often computationally intensive which results in a slow log-on procedure.

This work focuses on the design of the hardware acceleration of a WLAN authentication proceduce. The hardware acceleration is used in the Secure Remote Password (SRP) authentication protocol, but can be used in other security related computations as well. The experimental platform is TUTWLAN, a WLAN platform being developed at Tampere University of Technology which runs on the Altera Excalibur development board that contains a chip with programmable hardware and a microprocessor.

On a microprocessor, approximately 99 percent of the execution time of the SRP protocol is consumed by one single function: the modular exponentiation. Thus a significant speedup can be obtained by implementing this function in hardware. In this work the exponentiation function is implemented in the Altera programmable logic and used in the SRP protocol running on the microprocessor. In addition, proposals for further hardware speedup of the exponentiation design are presented. The results show that a full modular exponentiation with inputs of 1024 bits can be performed in less than 40 ms using less than 10.000 logic elements (i.e., 4-input lookup tables and registers) on the Excalibur programmable logic. By using the implemented hardware accelerator in the SRP protocol, the execution time is reduced by a factor of 4. An additional factor of 5 speedup (totaling a factor of 20) is possible by implementing the fastest design proposed in this work.

Computer Engineering Laboratory
Codenumber: CE-MS-2003-15

Acknowledgements

This thesis is the result of a research project as partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering at Delft University of Technology. The research project has been carried out in the Institute of Digital and Computer Systems at Tampere University of Technology in Finland. The goals of this work could never have been reached without the help of other people, whom I would like to thank in this section.

First I would like to thank my colleagues in the Institute of Digital and Computer Systems at Tampere University of Technology for assisting me in both research work and social contacts. Special thanks goes to Professor Timo Hämäläinen and my advisor Panu Hämäläinen for helping me through the toughest moments of the research project. Thanks to Panu and also Jari Nikara for supplying me with some basic goods and advises to survive the Finnish winter and summer.

Then I would like to thank Professor Stamatis Vassiliadis of the Computer Engineering group at Delft University of Technology for his support before, during and after my stay at Tampere University of Technology. Thanks to my advisor at Delft University, Ben Juurlink, for giving me all the freedom and independence.

Working on a research project is a great thing to do. Nevertheless this would not have been possible without spending time with my family and friends. Many thanks to my parents, brothers and sister for their love and support. Special thanks to Teppo Riihimäki, the Settlers of Catan players, the summer trainees, Sanna Nyrhi and the family Heikola. Many thanks to my friends in Delft for keeping my mailbox busy during my stay in Finland. Finally I would like to say thanks to all other people I have met during my exchange at Tampere University of Technology and with whom I have had some of the best moments of my life.

Delft, The Netherlands Peter Groen
August, 2003

Contents

List of Figures

List of Tables

List of Abbreviations

AHB	Advanced microcontroller bus architecture High-performance Bus
AKE	Asymmetric Key Exchange
ASIC	Application Specific Integrated Circuit
CLB	Configurable Logic Block
ESB	Embedded System Block
FPGA	Field Programmable Gate Array
fmax	maximum frequency
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
LAB	Logic Array Block
LAN	Local Area Network
LE	Logic Element
LED	Light Emitting Diode
LUT	Look Up Table
MAC	Medium Access Control
OSI	Open System Interconnect
PC	Personal Computer
PCI	Peripheral Component Interconnection
PLD	Programmable Logic Device
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
DPRAM	Dual Ported Random Access Memory
RSA	public key encryption algorithm by Rivest, Shamir and Adleman
SHA-1	Secure Hash Algorithm
SRP	Secure Remote Password
TCP	Transmission Control Protocol
TUTWLAN	Tampere University of Technology Wireless LAN
WLAN	Wireless Local Area Network
WEP	Wired Equivalent Privacy
UART	Universal Asynchronous Receiver/Transmitter

Introduction 1

In recent years wireless Local Area Networks (WLANs) have gained popularity. They can be used to connect computers when building a wired network is impossible or inconvenient, e.g., when users are moving or when the network is set up only temporarily. Because data is transferred through the ether, security is needed not only to prevent that data is intercepted by unauthorized users but also to authorize users and computers to the network. One way to obtain security is by using cryptographic operations. These operations, however, often require time-consuming computations that slow down network speed and increase log-on time. Furthermore, the cryptographic implementations may be required to support several algorithms as well as updating the implementation. Trading chip area and power-consumption for execution speed depending on the application requirements may also be profitable. High speed with flexibility can be obtained by using reconfigurable hardware for the time-consuming operations and a microprocessor for the remaining parts of the security implementation.

The objective of this work is to accelerate the authentication protocol of a WLAN developed at Tampere University of Technology. The protocol employed is the Secure Remote Password (SRP) authentication protocol [30]. It makes extensive use of hash and modular exponentiation functions. Both of them are often used in cryptography and require large amount arithmetic operations. The aim is to find and design appropriate hardware accelerators that can be called from software routines in the SRP protocol.

The WLAN, called TUTWLAN, is designed on a development platform supplied by Altera which combines an ARM9 microprocessor and a programmable logic device (PLD) in one chip. The modular exponentiation is much harder to compute than the hash function. The first objective is therefore to speed up the log-on protocol by designing a modular exponentiation for the Altera PLD and have it working with the software part running on the ARM9. The second objective is to investigate how more speed can be obtainded at a cost of a higher area consumption by modifications and additions to the accelerated exponentiation.

This thesis is structured as follows. An introduction to WLANs and security issues is given in Chapter 2. Chapter 3 describes the SRP log-on protocol and its performance in software. Chapter 4 explains the time-consuming functions in the protocol. The implemented hardware accelerator is described in Chapter 5. Chapter 6 presents the area consumption and execution time of the implemented design. The design can be modified in several ways to obtain a further reduction of the execution time. These modifications are discussed in Chapter 7. Conclusions and possibilities for future work can be found in Chapter 8.

2 WLANs and Security

This chapter gives an introduction about WLANs and security issues. Section 2.1 explains some basics of cryptology, section 2.2 gives a brief description of WLANs and the IEEE 802 WLAN standard and section 2.3 describes the TUTWLAN project at the Digital and Computer Systems laboratory of Tampere University of Technology.

2.1 Secure Transmission

A message is called secure if it can only be understood by those who are authorized to do so. The process of disguising a message is called encryption, the process of turning it back into the original message is called decryption. The science of keeping messages secure is called cryptography. Encryption and decryption can be done by a mathematical function that is not publicly revealed. The disadvantage of this is that a new function is needed every time a new connection is established. Using a special function that works with a variable called 'key' to encrypt and decrypt can solve this problem.

The key-based algorithms can be grouped in two categories: symmetric and public-key (or asymmetric) algorithms. In symmetric algorithms, the encryption and decryption key can be derived from each other. The keys of both the sending and the receiving party have to be kept secret. The public-key algorithms, on the contrary, allow one key to be publicly known. Only the decryption key has to be kept secret. This secret key cannot be derived easily from the public encryption key. It is similar to the working principle of a mailbox, anyone can deliver messages but only the key-holder can open the box.

Public-key algorithms are often used to distribute keys for symmetric algorithms. They can also be used to digitally sign a document. Signing can be done by using the private key to encrypt the message while anyone can read it by using the public key. Public-key algorithms usually require more computations than symmetric algorithms. They have to make use of one-way functions. These are functions that are easy to compute in one way but hard in the reverse way. Only someone who possesses the secret key can compute the reverse function easily [23, 1].

Communication is often done with session keys to increase security. These keys have a short lifetime and are different each time the communication channel is opened. These session keys have to be exchanged in a secure way which can lead to complicated procedures. Another issue is the log-on procedure needed to establish a connection. Both parties have to know the other one's identity to ensure that they are exchanging information to the right person. Often this is done by using a password.

The log-on procedure or authentication and exchange of the session keys can be performed by exchanging information between the parties in a series of steps, called a protocol. One of these protocols is the Secure Remote Access (SRP) protocol proposed by Thomas Wu [30] that is used for authentication in the Tampere University of Technology's WLAN platform. SRP is explained in detail in the next chapter.

A commonly used function in cryptography is the one-way hash function [23, 26]. It computes a fixed length value from a variable input string. One-way hash means that it is easy to compute a hash value from an input string but hard to compute the input string from the hash value. This function can be used to verify that someone really has a certain piece of data without actually sending it. One of the best known is the SHA-1 [24]. The hash function produces a hash of 160 bits out of an input message with length smaller than 2^{64}. Any change to a message in transit will, with very high probability, result in a different message digest, and the signature will fail to verify

2.2 Wireless Local Area Networks

Wireless Local Area Networks (WLANs) enable quick network setup and topology changes. They can be useful in both public (such as libraries and airports) as well as private areas (such as homes and meeting rooms), especially in those places where no good wired LAN is available and users are moving regularly.

A network consists of several layers [28]. The Open system Interconnect (OSI) model distinguishes 7 different layers. The lowest four layers are the physical layer, the data link layer, the network layer and the transport layer. For WLANs, the Transmission Control Protocol (TCP) is in the transport layer and the Internet Protocol (IP) is the network layer. The physical layer consists of a radio interface. Controlling and managing the WLAN is performed at the data link layer. The Institute of Electrical and Electronics Engineers (IEEE) has specified a standard for the data link layer and the physical layer for WLANs, known as IEEE 802.11b [15]. This standard specifies a Medium Access Control (MAC) protocol for the data link layer and provides a radio link speed of 11 Mbit/s at a radio frequency of 2.4 GHz [14].

Basically two types of topologies can be distinguished in WLAN: infrastructure and independent (ad-hoc). The infrastructure operates with a central server to which all terminals can connect, the independent established a connection between two individual terminals [28]. Since new terminals can freely move into the WLAN coverage area, good security is needed for both data transfer and authentication of users. The 802.11b standard provides a mechanism for authentication and encryption via the Wired Equivalent Privacy (WEP) protocol [15]. However, the WEP protocol is suffering from some security vulnerabilities [7].

2.3 The TUTWLAN Project

Research on the TUTWLAN project has been carried out in the Institute of Digital and Computer Systems at Tampere University of Technology (TUT). The work started in 1997 and consists of different areas of local wireless data communications. The first TUTWLAN platform was aimed at delivering a reduced complexity low-cost WLAN [18]. The demonstrator platform interfaces with the host through the PCI bus. The TUTWLAN MAC module (TUTMAC) [13] runs in a Digital Signal Processor (DSP). Interfaces with PCI and radio have been implemented in FPGAs. Improvements to this first TUTWLAN have been added in a second version [27].

Currently, a third version is being developed. The available hardware for the third TUTWLAN version is the Excalibur development platform manufactured by Altera [2]. It houses an ARM9 processor and programmable logic. The programmable logic contains lookup tables (LUT), registers and RAM blocks in which custom circuitry can be configured. The purpose of the TUTWLAN is to develop a scalable system, enabling a wide range of applications such as multimedia laptops, small handheld devices and wireless sensors. The application area introduces different quality of service requirements. This includes different levels of security by data encryption algorithms and authentication protocols. The SRP protocol will be used as a secure authentication protocol in TUTWLAN.

3 The SRP Authentication Protocol

The Secure Remote Password (SRP) protocol [30] is an authentication and key-exchange protocol suitable for secure password verification and session key generation over an insecure communication channel such as in WLANs. In this chapter this protocol is described and its performance in Software is measured.

3.1 The Steps in the SRP Protocol

SRP employs a method to exchange session keys called Asymmetric Key Exchange (AKE) [30]. In this method verifiers are stored instead of the plain-text passwords. The verifiers are computed from the passwords using a one-way mathematical function. The password and verifier correspond to private and public keys with the difference that the verifier is kept secret on the server instead of being publicly known. A stolen verifier is not sufficient to be able to log-on because the password is still needed. To establish a secure connection it is sufficient when one side (server) stores the verifier(s) while the other (client) computes the verifier from a user given password.

The initial setup of SRP goes as follows. A password is read in. First, the user enters a password. Then a verifier is computed from the password and a random password salt is generated. Next the user name, salt and verifier are stored in the database on the server. Now the client is ready to authenticate to the server. The SRP protocol operates as follows.

1. Client sends user name.
2. Server looks up this client's verifier and salt. The salt is sent to the client. The arithmetic is modular thus the client needs to know the modulus that is used for this session. It is sent together with the salt.
3. Client generates a random number, computes a public key from it, and sends it to the server.
4. Server generates a random number, computes a public key from it, and sends it to the client.
5. Both client and server compute the session key S.

6. Both sides hash their session key for use in the verification procedure.

7. Verification of the client's key: variable M_1 is computed and sent from the client to the server. Server verifies it.

8. Verification of server's key: variable M_2 is computed and sent from the server to the client. Client verifies it.

Figure 3.1 shows the functions that are computed in each SRP step on the server and client. All computations are performed modulo N. The modulus N is a large prime number with a length of hundreds of bits. The authentication is successful, if M_1 computed in Step 7 and M_2 in Step 8 are identical on the client as well as the server side.

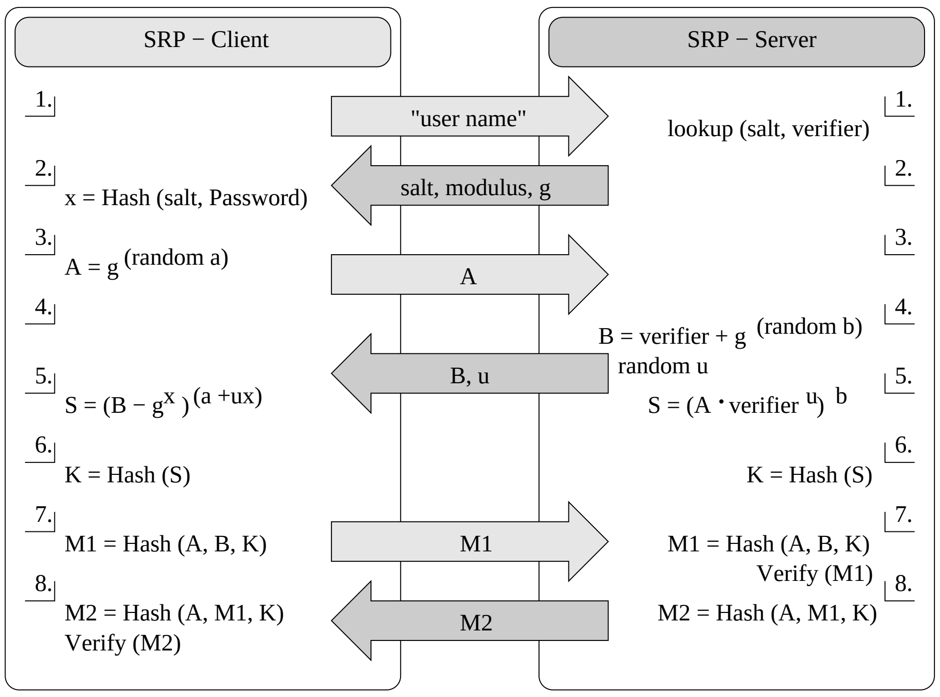

Figure 3.1: Steps in the SRP protocol [30].

In some of the steps, a one-way hash function is employed. The Secure Hash Algorithm (SHA-1) [24] is used for this. It produces a 160-bit hash value from a string of variable length. In addition, the protocol uses an interleaved hash that computes 2 hash functions and yields 320 bits. The first hash function is applied to the even bytes and the second to the odd bytes of the input. The hash outputs are byte-wise interleaved to get the final result.

The session keys are computed in Step 5 in Figure 3.1. Client and server do not possess the same variables but nevertheless they are able to compute the same session key. This is shown in Table 3.1. The verifier is computed from the password as $\mathit{verifier} = g^x$. Computation of the session keys requires modular exponentiations

Client	Server
$S = (B - g^x)^{(a+ux)}$	$S = (A \times \mathit{verifier}^u)^b$
$= ((\mathit{verifier} + g^b) - g^x)^{(a+ux)}$	$= (g^a \times g^{x^u})^b$
$= (g^x + g^b - g^x)^{(a+ux)}$	$= g^{a^b} \times g^{x^{u^b}}$
$= (g^b)^{(a+ux)}$	$= g^{(ab)} \times g^{(ubx)}$
$= g^{(ab+ubx)}$	$= g^{(ab+ubx)}$

Table 3.1: Equivalence of the session keys.

with large numbers in the exponent. Typically a and b are 256-bit values which translates to two modular exponentiations with large exponents on both server and client. The first exponentiation is in the computations of g^a and g^b and the second in the computation of S.

3.2 Security Analysis of SRP

Modular exponentiations are used frequently in cryptography because it is hard to compute the exponent given the base and result. The security of SRP is based on the idea that S cannot easily be computed from the information that is sent between the client and the server. In other words, someone who knows g^a and g^b cannot compute g^{ab} in polynomial time. This is because the mathematical structure of SRP is sufficiently similar to the Diffie-Hellman problem [30, 9]. The security in the Diffie-Hellman problem, also known as the discrete logarithmic problem, depends on solving the inverse of $P = g^x$ which is believed to be computationally infeasible: computation would take years when thousands of computers are used together to solve the problem. This only holds if the size of the exponent and modulus are sufficiently large. For now, a modulus of 512 bits is needed for minimum security but 1024 or 2048 is recommended [30].

Guessing a password does not help an intruder in finding g^{ab}. A passive attacker is unable to construct a session key by using the transmitted information and a guessed password because he is still missing the verifier. Furthermore an attacker cannot verify guessed passwords in any way which makes the SRP insensitive to password dictionary attacks on transmitted data.

Because a and b are random variables, previously used session keys cannot be computed even if the password is revealed, only a new session key can be computed in this case. Messages encrypted in the past with an old session key cannot be recovered by knowing the current password or session key. This is known as forward secrecy. Thus the SRP protocol is suitable to setup a secure connection.

3.3 Software Performance of SRP

The SRP protocol can be implemented in C using a library that allows large variables to be processed and contains optimized C/C++/Assembly routines. The

Table 3.2: SRP parameter sizes in bits for different moduli [12].

Parameter	N = 512	N = 1024	N = 2048
user name, u	32	32	32
password (x)	64 (160)	64 (160)	64 (160)
salt s	80	80	80
generator g	8	8	8
verifier v	512	1024	2048
random a, b	256	256	256
public A, B	512	1024	2048
key K, M1, M2	320	320	320

MIRACL library developed by Shamus Software Ltd [25] is very suitable for this since it has assembly support for the ARM processors. This library contains routines for cryptographic functions including modular exponentiation with Montgomery Reduction (see next chapter). The software performance of the SRP protocol has been evaluated for the ARM9 in [12].

Performance of SRP on the ARM9 processor was re-measured in this work to find out how many clock cycles each step in the SRP protocol needs to execute. This allows a comparison of the hardware accelerated SRP with the pure software SRP and shows the share of the modular exponentiations in the overall execution time. The Altera Excalibur development platform houses an ARM9 processor running on 200 MHz [2]. The performance is measured in the ARM AXD debugger simulator [3] since clock cycles can be measured accurately with this software. In addition, a counter is configured in the Programmable Logic Device (PLD) to measure the clock cycles while the protocol is running in the development board. The counter is started and stopped by writing a byte to the Dual Ported RAM (DPRAM). These hardware measurements are used to confirm the results of the simulation in the AXD debugger. The SRP parameters and their bit sizes are listed in Table 3.2. These sizes are the same as used in [12]. Measurements are done for moduli of 512, 1024 and 2048 bits. The size of A, B and the *verifier* increase with the length of the modulus, all other parameters are fixed for all moduli. The clock cycles counted per step of the SRP protocol are shown in the graph in Figure 3.2. Exact values are given in Appendix A.

Figure 3.2 shows that certain steps of the protocol consume almost all of the time. These are the steps that contain modular exponentiations with large exponents. The client needs more time than the server because the verifieris computed by exponentiation of the hashed password. The cycles for this computation are in Step 1 in the figure. This indicates that accelerating the modular exponentiation would be the most effective way to speed up the SRP protocol. Another option is performing the hash function in hardware. The hash function is used in almost every step of the protocol but the time it consumes is so small compared to the time required by the modular exponentiation that the benefit of a hardware implementation is limited.

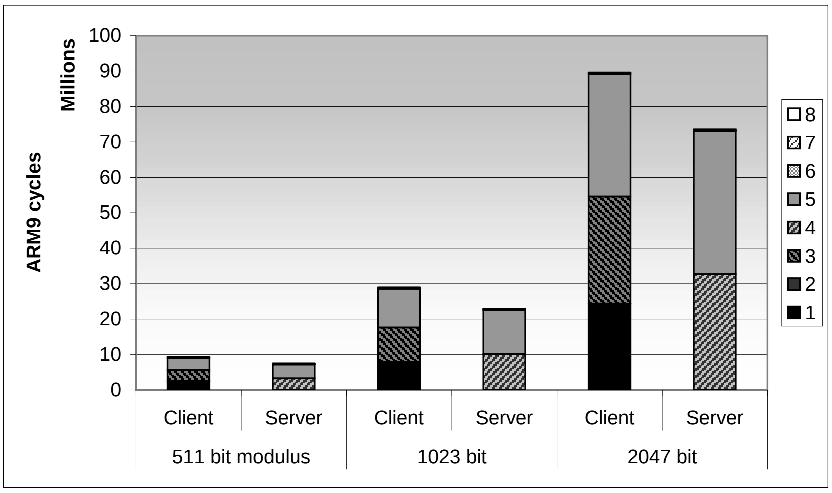

Figure 3.2: Clock cycles of the eight steps in the SRP protocol on ARM9 processor.

4 Hardware Acceleration

The SRP protocol obtains its security from modular exponentiations. This function consumes most of the clock cycles and is studied further to design appropriate hardware accelerators for the SRP protocol. In Section 4.1 the modular exponentiation function is described, Section 4.2 describes the Montgomery multiplication method and Section 4.3 discusses existing hardware implementations. Possibilities to accelerate the hash function are briefly described in Section 4.4.

4.1 Modular Exponentiation

An exponentiation basically consists of a series of multiplications. For example 3^5 can be computed as $3\times3\times3\times3\times3$. This is not the most efficient way to compute an exponentiation. A more efficient way to do this is by using the square and multiply algorithm [16]. 3^5 is now computed as $3^{2^2} \times 3$. There are two different methods to apply the square and multiply algorithm. Start with the most significant bit of the exponent or start with the least significant bit. Starting with the least significant bit requires one extra multiplication but the advantage is that squaring and multiplying can be done in parallel. Algorithm 4.1 describes the right-to-left method for exponentiation: bits are processed from least significant (rightmost) to most significant (leftmost). Example 4.1 illustrates how 3^5 is computed with this method.

Algorithm 4.1 *Right-to-left Square and Multiply [16].*
Input parameters:
X: number to be exponented consisting of m bits
E: exponent of n bits, where e_i is the i-th bit
N: modulus of m bits
Output parameters:
$P = X^E \bmod N$

```
1.  $Z = X,\ P = 1$
2.  FOR $i = 0$ to $n - 1$ DO
3.     IF $e_i = 1$ THEN $P = Z \cdot P \bmod N$ (multiplication)
4.     $Z = Z^2 \bmod N$ (square)
5.  END FOR
```

Example 4.1 *Square and Multiply of* $3^5 \bmod 15$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{5 = 101_{bin} \Rightarrow e_0 = 1, e_1 = 0, e_2 = 1} \\
 & Z = 3,\ P = 1 \\
i = 0: & P = (Z \cdot P) \bmod 15 = (3 \cdot 1) \bmod 15 = 3 \\
 & Z = (Z \cdot Z) \bmod 15 = (3 \cdot 3) \bmod 15 = 9 \\
i = 1: & Z = (Z \cdot Z) \bmod 15 = (9 \cdot 9) \bmod 15 = 81 \bmod 15 = 6 \\
i = 2: & P = (Z \cdot P) \bmod 15 = (6 \cdot 3) \bmod 15 = 18 \bmod 15 = 3 \\
 & Z = (Z \cdot Z) \bmod 15 = (6 \cdot 6) \bmod 15 = 36 \bmod 15 = 1
\end{array}
$$

This algorithm needs only $\lfloor \log_2(E) \rfloor + v(E)$ multiplications to computer X^E while the simple multiplications method needs E multiplications, where $v(E)$ is the number of ones in the binary representation of E. The maximum number of multiplications will be $n + n = 2n$, where n is the number of bits in the exponent in binary representation. The number of squares is always n while the number of multiplications is only $0.5n$ on average. A hardware implementations can benefit from this and obtain an execution time of $1.5n$ on average. More speed-up can be obtained by considering processing several bits of E per multiplication. This will reduce the total number of multiplications and can thus decrease the execution time. The method is called M-ary and is explained in more detail in Chapter 7.

As can be seen from Algorithm 4.1, every multiplication needs to be performed modular N. This can be done by taking the remainder of division by N. Division are an expensive operation. They can be performed by a series of subtractions, which yields a significant slowdown of the execution time of the algorithm.

Barret has defined a way to do this more efficiently [11]. In his algorithm, the division is replaced by multiplications. Also Brickell has found a way to do the subtractions faster [8]. An even better efficiency can be obtained by Montgomery's method [19, 17, 10]. This is method is explained in detail in the next section.

4.2 Montgomery Arithmetic

Montgomery transforms the input values to a special residue (called N-residues) to avoid the expensive divisions. The transformations take time so Montgomery becomes most useful for series of arithmetic operations such as in modular exponentiations.

The operands are transformed into these residues before starting to perform multiplications and afterward the residue is transformed back to normal representation. An N-residue of X is defined as $(X \cdot R) \bmod N$ where R is chosen to be a power of two such that $R > N$. The key idea is to do divisions by R instead of the regular modulus N which replaces the expensive division after each multiplication by an inexpensive division by a power of 2 (i.e. a rightshift). Montgomery has defined Algorithm 4.2 to do multiplications using N-residues. The output of the algorithm is $(T \cdot R^{-1}) \bmod N$ where the input is T.

Algorithm 4.2 *Montgomery reduction [19].*
Input parameters:

N: modulus, $N' = -N^{-1}$
R: radix, such that $R > N$ *and* R *co-prime to* N
T: input, where $0 \leq T < RN$
Output:
$t : (T \cdot R^{-1}) \bmod N$

```
function REDC(T)
  1.  $m = (T \bmod R)N' \bmod R$
  2.  $t = (T + m \cdot N)/R$
  3.  IF $t \geq N$ THEN $t - N$
      ELSE $t$
```

The steps in Algorithm 4.2 can be described as follows. In Step 1 is determined how many multiples of the modulus should be added to T in Step 2. This addition of m times the modulus N will make sure that the output of the division by R is an integer. The modular inverse of N is N^{-1} in $NN^{-1} = 1$ where the modulus is R, $N' = -N^{-1}$ is the negative modular inverse of N. The modular inverse of R is R^{-1} in $RR^{-1} = 1$ where the modulus is N. When these are combined it results in the condition $RR^{-1} - NN' = 1$ which must hold for Algorithm 4.2 [19].

In Step 2, $T + mN$ can be as large as $RN + RN$, so $0 \leq t < 2N$. The *IF* in Step 3 assures that the output of the algorithm is smaller than the modulus N. The radix and modulus need to be co-prime (no common divider greater than one) to guarantee the existence of an inverse. Thus the algorithm works for any odd modulus N which makes the radix co-prime to N. The output of this algorithm is the input times $R^{-1} \bmod N$ which keeps the output within the size of the inputs.

A product of residues of X and Y is computed as:

$$\begin{aligned} z &= REDC((XR \bmod N)(YR \bmod N)) && (4.1)\\ &= (XYR^2)R^{-1} && (4.2)\\ &= XYR \bmod N && (4.3) \end{aligned}$$

Transformation to and from Montgomery-residues can be done by using *REDC*. The initial values X and Y have to be multiplied with $R^2 \bmod N$. Transformation from residues back to normal representation can be done by multiplication with 1. Now only $R^2 \bmod N$ has to be computed using slow non-Montgomery multiplications. This has to be performed only once for a given modulus. Example 4.2 illustrates Algorithm 4.2 by computing $6 \cdot 6 \bmod 13$ using $REDC(T)$.

Example 4.2 *Montgomery modular multiplication.*

Constant for a given modulus:
$N = 13,\ R = 16$
$N' \bmod R = -5 \bmod 16 = 11,$
$R^2 \bmod 13 = 16^2 \bmod 13 = 9$

Transformation to residue, multiply with $R^2 \bmod N$*:*
$T = 6 \cdot 9 = 54$
1. $6 \cdot 9 \bmod 16 = 6$
$m = 6 \cdot 11 \bmod 16 = 2$
2. $t = (54 + 2 \cdot 13)/16 = 5$

The actual multiplication:
$T = 5 \cdot 5 = 25$
1. $25 \bmod 16 = 9$
$m = 9 \cdot 11 \bmod 16 = 3$
2. $t = (25 + 3 \cdot 13)/16 = 4$

Back to normal representation, multiply with 1:
$T = 4 \cdot 1 = 4$
1. $4 \bmod 16 = 4$
$m = 4 \cdot 11 \bmod 16 = 12$
2. $t = (4 + 12 \cdot 13)/16 = 10 \Rightarrow 6 \cdot 6 \bmod 13 = 10$

The divisions in Example 4.2 can be implemented very efficiently in hardware now. All that needs to be done is take the four least significant bits in step 1 and perform a four bit right-shift in step 2. The expensive divisions are avoided.

A multiplication of two large numbers requires a lot of area to implement in hardware. Therefore the multiplication cannot be done in one single step. It is perfomed as an addition of products. Each product is the result of a part of the multiplier multiplied with the full multiplicand. A part can be as small as one bit. If l is the number of bits of the multiplier, a full multiplication requires l additions of a multiplication of 1 bit of the multiplier with the full multiplicand. If k bits of the multiplier are processed (radix-2^k), a full multiplication requires l/k multiplications (see Example 4.3). Algorithm 4.2 can be transformed to Algorithm 4.3 to realize a radix-2^k multiplier that is called l/k times [10, 5].

Example 4.3 *A multiplication can be split into smaller multiplications.*

```
   110 = 6 multiplicand   110                 110
   110 = 6 multiplier     110                 110
-------                 ------              ------
   000  (110 x 0)         1100 (110 x 10)   100100 (110 x 110)
  1100  (110 x 1)        11000 (110 x 01)
------- +               ------ +
  1100                  100100
 11000  (110 x 1)
------- +
100100 = 36
radix-2 multiplier   radix-4 multiplier   radix-8 multiplier
```

Algorithm 4.3 *Radix-*2^k *Montgomery modular multiplication [10, 5].*
$j = l/k$*; q consisting of k bits*

Input parameters:
N: modulus of m *bits* $N < R$, $N' = -N^{-1}$
A: multiplier of $a \cdot k = l$ *bits,* $A < N$, $A = \sum_{i=0}^{l-1} a_i (2^k)^i$
B: multiplicand of l *bits,* $B < N$
Output parameters:
S: $ABR^{-1} \bmod N$, *R such that* $R = 2^{jk}$ *and R co-prime to N*

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

The algorithm can be further improved by replacing the *IF* statement in Step 6 by two extra iterations where $a_i = 0$. Two extra division in step 4 will make the result always smaller than the modulus. The output value of Algorithm 4.3 is $ABR^{-1} \bmod N$. The algorithm is suitable for implementation on reconfigurable hardware because large multipliers and divisions are avoided. The Montgomery algorithm allows a pipelined implementation because the next computation does not have to wait for the most significant bits in the determination of q_i. This makes Montgomery suitable for processing multiplications with very large moduli. It also allows for changes in radix to trade speed for area. Several designs that use Montgomery multiplications for programmable logic have been reported. A review of them is given in the next section.

4.3 Existing Designs for Modular Exponentiation

For the implementation of modular exponentiation in hardware, several approaches are available. Existing designs of modular exponentiations for Field Programmable Gate Array (FPGA) platforms have been studied and are briefly described below.

There are basically two methods to compute a Montgomery modular multiplication. One is the redundant representation, addition to a binary representation is only done for the final result to prevent very long carry propagations. Intermediate results are kept in the carry-save state. In redundant representation, carries and sums are kept separated after each iteration. In an array approach, a multiplication is done by a large number of parallel units that each compute a couple of bits of the result. This is a line of parallel processing units, in which values (q_i and a_i in Algorithm 4.3) are 'pumped' through the units while carries are moved to the next unit in each clock cycle.

Non-Montgomery modular multiplication methods are available as well. Johan Groszschaedl [11] uses Barret's modular reduction but did not implement it on an FPGA.

Blum and Paar's design is one of the fastest implementation of Montgomery

Modular Exponentiation on an FPGA [5, 4]. They have developed a resource efficient and a speed efficient design for Xilinx FPGA, both based on an optimized version of Algorithm 4.3. The speed efficient design is a radix 16 (k=4) implementation in which 16 multiples of the modulus and multiplicand B are precomputed, stored in RAM and addressed with q_i and a_i respectively [6]. Their implementations are based on an array approach in which multiplication and squaring are performed interleaved in the same array. Their radix-2 implementation runs in approximately $2n$ cycles and their radix-16 in almost $0.5n$ cycles where n is the length of both the exponent and ground. Others have modified the array of processing units design for Montgomery modular exponentiation which allows their implementation to run in $1.5n$ cycles [29].

Another possibility is to have separate arrays for multiplying and squaring. Naturally this will cost twice as most area to have the full exponentiation done in n cycles instead of $2n$. In[22] two arrays are used in an implemented on Xilinx FPGAs. This design needs four times as much resources as Blum's radix-16 design while it is only 20% faster.

A comparison between an array of processing untis and an iterative array can be found in [20]. It turns out that the processing units approach takes about 1.5 times more resources for large modulus at a benefit of at least a 2 times faster execution time. In [5] is argued that the array of processing units needs more resources than redundant approach to store the signals between units in the array but is easier to route and can therefore be faster than a redundant representation on FPGA. [21] also describes both approaches but has not made an implementation in hardware.

The problem in additions and multiplications with big input values is transportation of the carries through the result of each step in the computations. One way to solve this is to represent the results in redundant representation. Extra register-bits are allocated to store the carries instead of transporting them all the way to the most significant bit. This method works fine in for example ASIC hardware implementations , but it might cause routing problems in reconfigurable hardware [5]. For reconfigurable hardware, a split-up into smaller units is more likely to turn into a design that can be routed successfully. A higher maximum frequency (fmax) can be expected.

4.4 Hash Function

The hash function is often used in the SRP protocol. This function can be implemented in HW. The effect on the overall performance will be small though since the exponentiation takes almost all of the execution time. Hardware implementations of hash function have already been implemented by many others. The best known hash is the (SHA1-160) [24] in which the input cannot easily be derived from the outputs. Altera is selling a ready made implementation of this function for their programmable logic platforms.

Hashing of a number with SHA-1 to 160 bit for 1024 bit is done in approximately 38,350 cycles in the ARM9. In addition to this comes initialization, the final result and conversion from MIRACL variables to bytes. These conversions are also needed for HW exponentiation. Clock cycles can be saved by hashing in hardware. Altera supplies an IP core for the SHA-1 [2] that is suitable for the Excalibur device's PLD. The SHA-1 IP core can hash 512 bits in 335 clock cycles, thus 1024 bits in less than 700 cycles. This would translate to approximately 3000 cycles in ARM9. For a 1024 bit modulus, about 100,000 processor cycles can be saved this way. This is far less than what is needed for a modular exponentiation which takes about 10,000,000 cycles.

The core consumes 1377 LEs, 6 ESBs and has a maximum frequency (fmax) of 78MHz. The function can share the communication channels between ARM and PLD with the exponentiation function. This leaves only data type conversions, subtractions, additions and single multiplications to the software. Since this is an off-the-shelf available function that saves a relatively small bumber of clock cycles, it is not studied any further in this thesis.

Implementation of Modular Exponentiation 5

The most promising way to accelerate the SRP protocol is to perform the modular exponentiation function in the reconfigurable hardware. In this work this function is implemented in the Programmable logic Device (PLD) of the Altera Excalibur platform. Section 5.1 gives on overview of the hardware. Section 5.2 and 5.3 describe the implementation of a radix-2 modular exponentiation in an array of processing units. The array splits the design into small blocks. The radix-2 design includes communication with the ARM9 processor and integration in the SRP protocol.

5.1 Altera Excalibur Platform

The chip on the Altera Excalibur device consists of a 32-bit RISC ARM922 processor operating at up to 200 MHz combined with a programmable logic device (PLD). Processor and PLD are connected by an embedded stripe. Communication between stripe and PLD goes through two dual ported RAM blocks (one for the smallest Excalibur device) or through AHB master/slave ports. The PLD can also interrupt the ARM directly through 6 interrupt lines. Figure 5.1 shows the structure of the stripe in Excalibur devices.

The stripe contains 2 AHB busses. The first bus, AHB1 connects the processor to the stripe and the dual ported RAMs. Communication through the dual ported RAMs can be at 8, 16 or 32 bits. This allows for fast read and writes of large amount of data between PLD and ARM processor. The AHB1 also contains a bridge to the second bus, AHB2. The second bus operates on a lower frequency and connects the AHB master/slave ports in the PLD to the processor. The AHB2 bus also provides a connection to UART and external memory (flash, SDRAM). These memory blocks can contain PLD configuration data and/or ARM application code. The configurations can be loaded onto the PLD by instructions from the ARM, which allows changes in the PLD configuration during operation. An overview of the specifications of the Altera Excalibur devices is given in Table 5.1 [2].

The structure of the PLD in the Excalibur is equal to Altera's APEX20K devices at an internal voltage of 1.8. The Apex devices are based on a large number of Logic Elements (LEs). Each LE contains a 4-input look-up table (LUT) and one register which means that any boolean function of up to 4 bits can be implemented in one LE. It can also be configured to operate in a special arithmetic mode which

Table 5.1: Excalibur devices.

Device	EPXA1	EPXA4	EPXA10
Processor	ARM922T	ARM922T	ARM922T
Maximum operating frequency	200 MHz	200 MHz	200 MHz
Single-port RAM	32kbytes	128kbytes	256kbytes
Dual-port RAM	16kbytes	64bytes	128bytes
Typical gates	100,000	400,000	1,000,000
Logic Elements (LEs)	4,160	16,640	38,400
Embedded system Blocks (ESBs)	26	104	160
Maximum system gates	263.200	1,052,000	1,772,000
Maximum user I/O pins	246	488	711

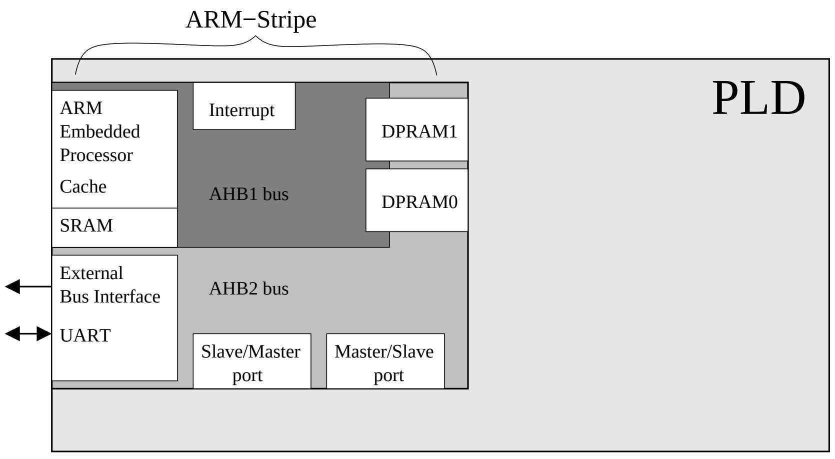

Figure 5.1: The embedded Stripe in Altera Excalibur.

has only two inputs but allows for fast propagation of carries between neighboring LEs. A register is available directly after each LUT. The output of the LUT can be routed through the register, around the register or both. In addition, several neighboring LEs can be cascaded together to act as a bigger one by using the cascade in/out ports of an LE.

Ten LEs are grouped in a Logic Array Block (LAB). Such a LAB is surrounded by 2 interleaved interconnect channels to transfer signals between LEs in adjacent LABs (Figure 5.2). LABs are grouped together in a MegaLAB which consists of 24 LABs in the largest Excalibur device (EPXA10) and 16 in the smaller ones. Each MegaLAB also contains an Embedded System Block (ESB). An ESB can directly be connected to an adjacent LAB and one ESB holds up to 2048 Dual-ported RAM bits in one of the following possible configurations: (addresses x in/outputs) 128x16, 256x8, 512x4, 1024x2 or 2048x1. ESBs can be used for storage of data or configured to a special kind of large LUTs.

MegaLABs are connected by Fast-Track interconnects and grouped in an array of 4 columns and x rows, where x depends on the size of the chip. For the EPXA10, the total number of LEs in one MegaLAB is 240. Its PLD contains 40 rows, each row consisting of 4 MegaLABs, resulting in a total of 38,400 LEs and 160x2048 ESB-bits.[2].

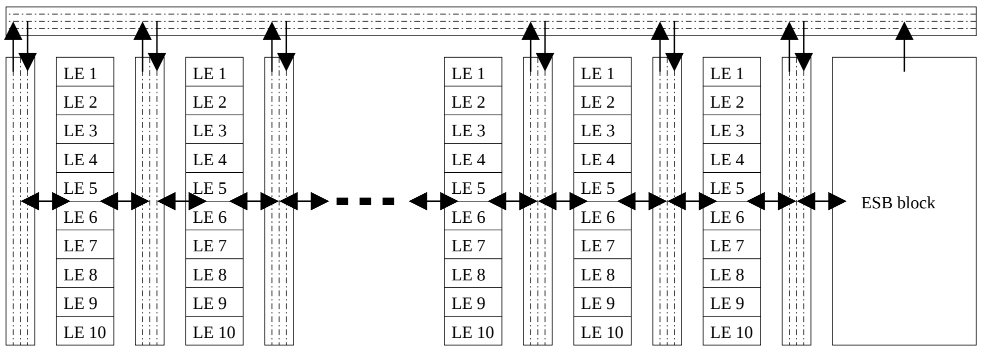

Figure 5.2: MegaLAB in Altera APEX PLD.

5.2 Radix-2 Modular Multiplication Algorithm

The aim is to design and implement a Montgomery modular exponentiation with an architecture as simple as possible so that no routing problems will arise. This should result in a high maximum frequency and an acceptable area consumption. The second requirement is that this design should be easy expandable to make it suitable for use in faster designs.

For the design, a slightly modified version of the radix-2^k Montgomery algorithm (Algorithm 4.3) is used. The *IF* statement in Step 6 is removed at a cost of 2 extra iterations of the for-loop. This guarantees that the output will be smaller than two times the modulus if the inputs are smaller than two times the modulus. The addition in Step 4 is avoided by multiplying B by 2. The product $a_iB \bmod 2$ will thus be 0 for all B. In Step 3, B can be taken out of the division.

The modulus N must be odd to be co-prime to the Montgomery radix R, the inverse of N in Algorithm 4.3 for the radix-2 algorithm becomes 1 since $N' = -N^{-1} \bmod 2 = 1$ for every odd N. No inverse needs to be computed. Algorithm 5.1 shows the modified algorithm.

Algorithm 5.1 *Radix-2 Montgomery modular multiplication [5].*
q consisting of 1 bit
Input parameters:
N: modulus of m bits $N < R$, $R = 2^{m+2}$
A: multiplier of $l + 1$ bits, $A < 2N$
B: multiplicand of $l + 1$ bits, $B < 2N$

Output parameters:
S: $ABR^{-1} \bmod (2N)$

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

The fourth step of Algorithm 5.1 contains the arithmetic operations that can be implemented in an array of processing units in the PLD. Two multipliers and two adders are needed in each unit to implement the multiplications and addititions. Further registers and multiplexors will be needed to store values and to select the appropriate input. The algorithm needs $m + 2$ iterations to compute a full multiplication, where m is the bit-length of the multiplier A.

The design can be split into smaller parts by utilizing an array of processing units. It consists of small processing units that each compute a part of the final result of the multiplication. One of the input variables is loaded in the units and the other transported through the array of processing units. Blum and Paar have designed a resource efficient design of radix-2 and a speed efficient design of radix 16, both based on Algorithm 4.3. However, their designs are optimized for Xilinx FPGAs instead of Altera's PLD.

Some differences between Xilinx and Altera can be distinguished. The Xilinx LE is called CLB and is approximately twice as large as an Altera LE. The Xilinx CLB contains two 4-input LUTs and a third one to combine the results from those LUTs. Two registers are available in each CLB after the LUTs registers. The Altera LE contains only one 4-input LUT and one register. The PLD on the Excalibur development board possesses a large number of RAM bits in ESB blocks, the XC4000 chip that is used by Blum in [5] is capable of configuring its CLBs as RAM modules. Therefore a new design, based on the processing units approach as used by Blum and Paar on the Xilinx FPGA, is presented below.

5.3 Implementation of Radix-2 Modular Exponentiation

5.3.1 Exponentiation Unit

The computations are performed in an array of processing units as depicted in Figure 5.3. The multiplicand B is loaded into the units from a central point, all other signals are moved between neighboring units. Algorithm 5.1 defines that all intermediate values are at most two times as large as the modulus. This means that, internally everything should be 1025 bits to allow a 1024-bit modulus. To simplify the design, the maximum modulus is set to one bit less: $2^n - 1$. Moduli will have a lengths of for example 511, 1023 or 2047 bits. This implies that the lenght of all intermediate values in the array is a power of two which simplifies the structure of the design. The design is such that it can be quite easily changed to any modulus $2^n - 1$ for n up to eleven.

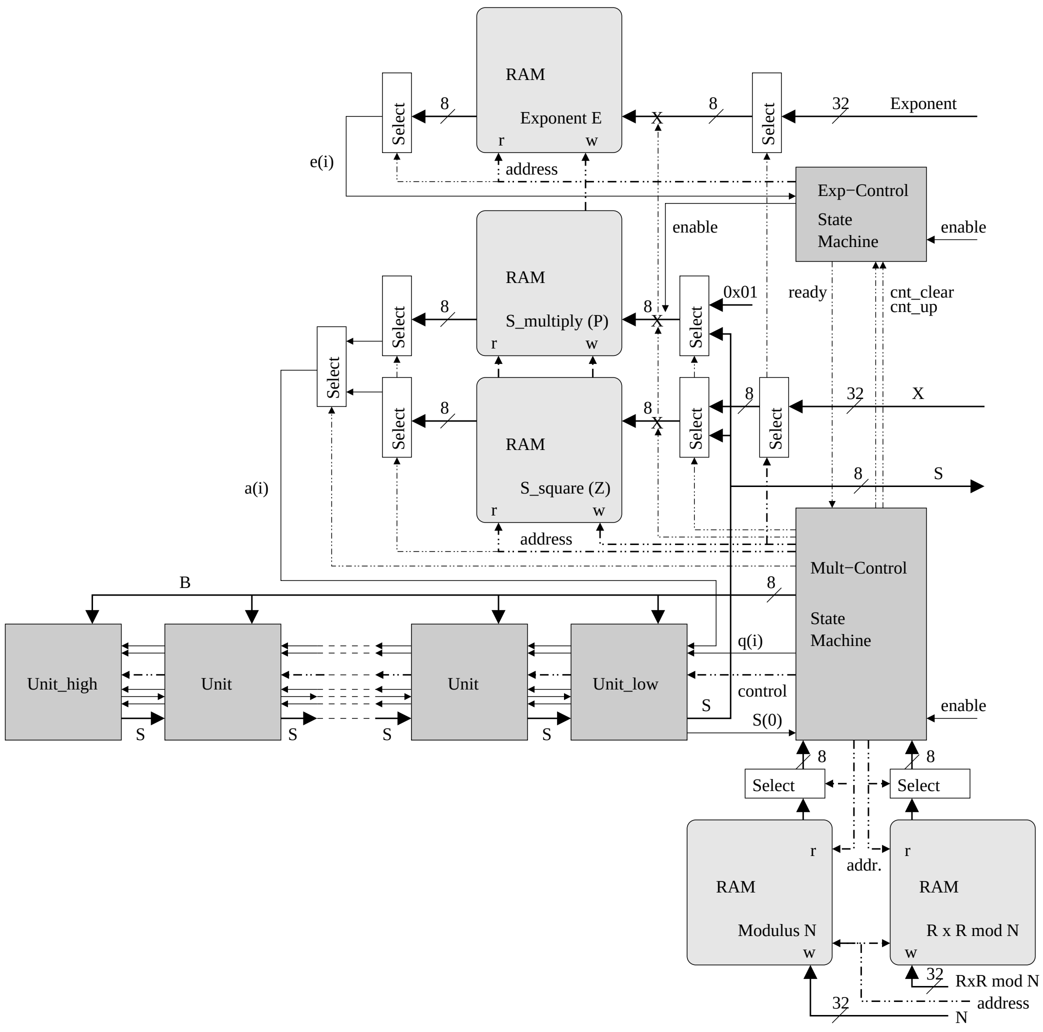

Figure 5.3: Array of processing units radix-2 exponentiation in PLD.

Figure 5.3 shows five RAM blocks. The N and R block store the modulus N and the Montgomery radix R^2 mod N. The Montgomery radix is used to transform variables into residues. The upper block stores the exponent E and the two blocks in the middle of the figure store the results of each squaring and multiplication. Two control machines are included in the design. The upper machine (Exp-control) contains a counter to generate the address to select the appropriate bit of the Exponent E. The lower machine (Mult-control) controls the array and operates as follows.

1. Startup.

2. Start loading modulus into units.
3. Load remainder of modulus into units.
4. Load exponent and base from stripe-DPRAM to ESB/RAM on PLD.
5. Start loading base B into units.
6. Repeat the following steps i+3 times (i=max length: 32, 512, 1024, etc.):
   (a) Load for first multiplication (repeated), simultaneously obtain previous results from the array.
   (b) Load for second multiplication (repeated), obtain results, don't use them if Exp[i]='0'.
   (c) Initialize processing units for next multiplication and start reading results.
7. end

The procedure to compute $X^E \bmod N$ using Montgomery is as follows: X is transformed to Montgomery-residue. This takes one Montgomery multiplication. Then the Square and Multipy algorithm is applied. The final output P is transformed back to normal representation which takes another Montgomery multiplication. The computation of an exponentiation is divided into four episodes: 0. loading of modulus from DPRAM in processing units and exponent to ESB/RAM. 1. transformation to residues and automatically continue with the Square and Multiply algorithm. 2. Load the number '1' in the processing units and transform back to normal presentation 3. transportation of the results back to the processor. A 2-bit counter is configured that moves up every time when a new episode starts. The counter acts like a second layer state-machine which reduces the states in the first layer state machine to nine. A detailed state diagram of this state machine is included in Appendix B.

In the radix-2 algorithm, only one bit of the multiplier is processed per multiplication thus one iteration of the Multiply and Square algorithm takes $(l+3) \times 2$ cycles. All iterations together take $n \times (l+3) \times 2$ cycles. A full exponentiation (exponent and modulus have the same length), including transformations, takes $(n+2) \times (n+3) \times 2$ cycles, where n is the bit-length of the modulus, ground and exponent.

5.3.2 Processing Units

Each unit contains a certain amount of bits of the multiplicand B in the multiplication of A and B. The number of bits per unit is an important factor in the maximum frequency (fmax) since it directly affects the routing complexity and longest path. More bits per unit saves registers but increase the complexity and longest path.

The most suitable size of a processing unit is determined as follows. Any processing unit needs at least 8 registers to store carries, control-bits, a_i and qi. Thus processing less than 8 bits per unit would result in an overhead of at least one LE per processed bit. This reduces if the unit size increases. There are 10 LEs in a LAB which allows for efficient implementation of 8 bit carry-propagation adders. Processing 16 bits could be an option, but the carry delay will cause a significant increase in the longest path. Processing 16 instead of 8 bits per unit saves only 0.5 LE per bit. This saving may not be enough to compensate the reduction in fmax due to more complicated routing. Thus the number of bits of B per unit is 8 in this design.

Figure 5.4 shows the structure of one processing unit. It computes $S_{i+1} = (S_i + q_iN)/2 + a_iB$. *Adder_1* computes $(S_i + q_iN)$ and *Adder_2* adds a_iB. The block labeled *Box* computes the 9th bit of the addition of S and q_iN. This bit is needed as extra input for the second adder because of the rightshift between the adders. For the highest unit, *Adder_2* needs to be one bit larger and the box will contain a 2-bit adder instead of 1-bit. A unit has to go perform the following steps to perform a Montgomery multiplication.

1. The modulus N (8 bits per unit) is loaded via B in Multiplexer+Reg_1. Multiplier_1 is set on multiplication by 1 to let the modulus pass through unchanged while the registers in Reg_1 are reading the data.

2. The multiplicand B (8 bit per unit) of Algorithm 5.1 is loaded via B in Multiplexer+Reg_1. This operand is the same for both squaring and multiplication step of the exponentiation Algorithm 4.1.

3. The multipliers a_i are moved through the units at one bit per clock (radix-2).

4. The result of the first multiplication (squaring) are fed back to Multiplexer+Reg_2 for the next iteration.

5. After the completion of the multiplications, both results are moved to Multiplexer+Reg_3. The result of the first multiplication (squaring) is sent to the previous unit and also to Multiplexer+Reg_1. The result of the second multiplication is moved to the previous unit only.

6. Step 2 to 5 are repeated to compute an exponentiation. In Step 2, S is loaded instead of B.

All signals are registered before they leave a unit. Thus a signal needs one clock cycle to move to the higher neighbor. The higher neighbor uses these signals and shifts its lowest bit back to the lower unit afterwards. This allows for a two-cycle operating mode with interleaved squaring and multiplying. The unit could be transformed to a single-cycle version by not registering the bit of S that is transported to the lower unit. In a single-cycle version, the square and multiply steps in the exponentiation algorithm are performed in series. No long carry chains through the whole array are created as long as the size of the adder is at least twice

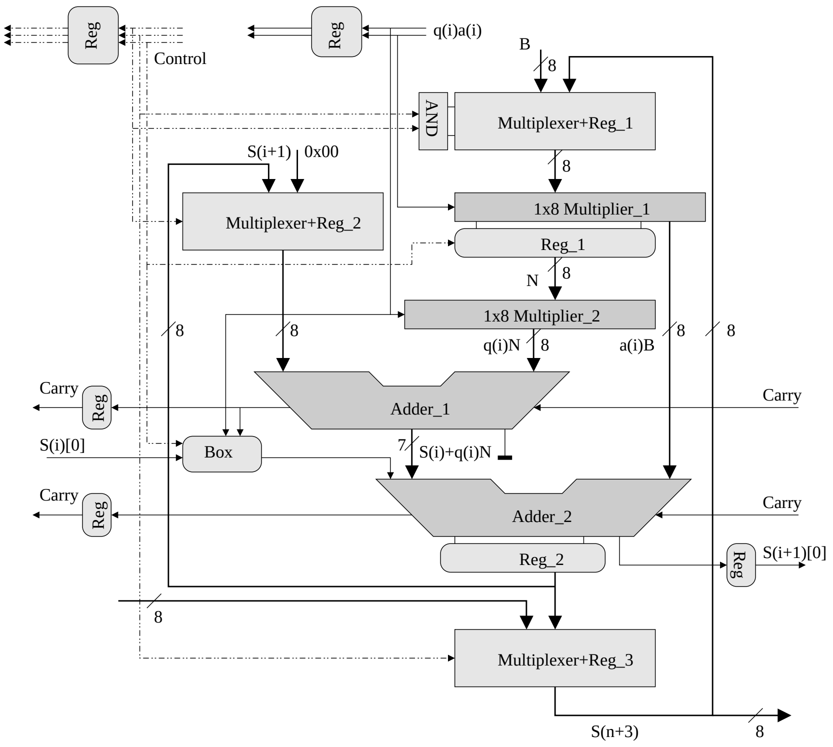

Figure 5.4: 8-bit radix-2 processing unit.

as large as the number of bits of S that are moved to the lower unit. For an 8-bit wide unit, up to 4 bits or radix-16 can be computed in one-cycle mode.

Each unit needs 70 LEs when implemented in the Altera APEX PLD. The LEs needed per component in a processing unit are listed in Table 5.2. The registers after the adder are combined in one LE. The registers that store the modulus N are placed directly after the 1×8 Multiplier so that they fit in the same LE. During initial loading of the modulus, bits are moved through both multiplexer+Reg_1 and 1×8 Multiplier_1 and stored in the Reg_1. *1x8 Multiplier_1* and *1x8 Multiplier_2* are actually no real multipliers since one of the inputs is only one bit width. If that bit is zero, then the outputs are zeros, if that bit is one, the outputs are the same as the inputs. The VHDL code for a processing unit is included in Appendix B.

5.3.3 Communication with the ARM Processor

Communication between the PLD and the ARM processor can be done through DPRAM or AHB slave/master ports. The DPRAM ports allow for quick access to large amount of data. Since the exponentiation function needs large numbers before computation can start and the result has to be written back afterwards, the DPRAM ports are most suited for this data transfer. For maximum performance, both DRPAM ports are configured for 32-bit data size, so reading of an exponent

Table 5.2: LE consumption per 8-bit processing unit.

	LEs	#	Totals
multiplexer+reg:	8	3	24
1x8 multiplier:	8	2	16
carry-adder:	10	2	20
8-bit registers:	-		-
registers:	1		8
box:	2		2
Total:			70

and base of 1023 bits takes 32 clock cycles. The disadvantage of using both ports at maximum width is that more LEs are needed to read and write data. A 32-bit port needs about 4 times more LEs are than an 8-bit port. LEs are mainly consumed by registers and control that is needed to move the data to the array. Since the design is part of TUTWLAN and therefore might have to share the DPRAM ports with other applications in the PLD. The configuration of the DPRAM ports can be adapted if necessary. Because the LE consumption would still be small relative to the amount that is needed for the multiplication array, both DPRAM ports are configured for 32 bits.

The hardware modular exponentiation has to be enabled/disabled by sending a signal to an AHB-slave in the PLD. This allows other parts of the TUTWLAN to freely use DPRAM when no exponentiations are computed. In this implementation, the DPRAM is used from the first read of inputs till the the last write of results. The design can be modified to allow other modules to use DPRAM also during a computation. An extra ESB/RAM block is needed in that case to store the final result of a computation instead of writing it back to DPRAM directly from the array.

Four parameters are needed to perform exponentiation. The modulus M, the Montgomery radix R^2, a base and and an exponent. The read and write process is managed by a state machine (PLD_control in Figure 5.5). Base and exponent are stored in ESB/RAM blocks, and the addresses are generated by the exponentiation control unit. Reading of N and R^2 is controlled by a separate control unit that generates the address. They are stored in ESB/RAM blocks. They have to be read only when a new modulus is used. The state machine PLD_control operates as follows:

1. Reset/startup upon receive of enable signal from AHB-slave.
2. Read signal from DPRAM and select between regular computation or loading of N and R^2 (go to step 5).
3. Connect DPRAM to exponentiation hardware and start computation.
4. At finish of computations (episode 3): write results back to DPRAM.

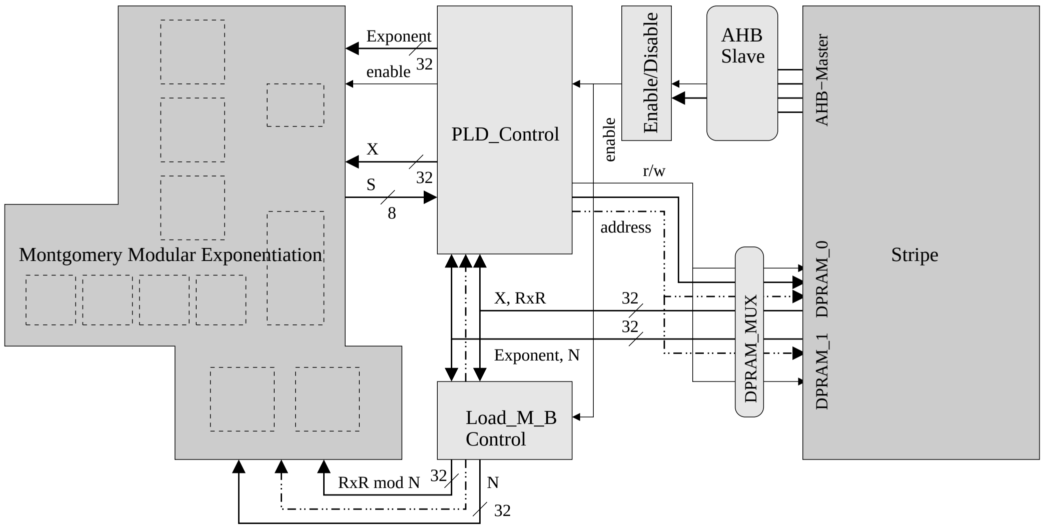

Figure 5.5: Communication between ARM and Exponentiation.

5. Read N and R^2 from DPRAM and load into ESB/RAM.
6. Write to DPRAM to tell ARM that operation is finished.

5.3.4 Software Routines

The hardware exponentiation is controlled by the software running on the ARM. The software consists of three routines:

1. Read N, compute R^2 mod N and store them in DPRAM.
2. Read Base and Exponent, move them to DPRAM, start accelerator.
3. Read results from DPRAM.

The radius R needs to be transfered to Montgomery domain (R^2 mod N) by traditional divisions. A hardware implementation would require a small array that can do subtractions and comparison at 32 bits per clock cycle. Processing 1024 bit costs 32 cycles. This process has to be repeated 1024 times (half of the size of R^2 for a modulus that has the same size as R). Since it is likely that the modulus is not frequently changed in the SRP protocol, this operation is performed in software. Changing the modulus requires computing new verifiers for each client and updating the whole server database (it is assumed that the whole database uses the same modulus).

The SRP implementation processes variables in a special large format supplied by the MIRACL library. Before moving variables to DPRAM, they are transformed to strings, then transformed to a series of 32-bit integers and stored in DPRAM. Then the PLD is released from reset and started by writing a number to the lowest DPRAM address. In the third routine, the result is read from DPRAM after the

PLD has set a value to the lowest DPRAM address to indicate that results can be read. The results are transformed from integers to string and then back to the MIRACL big format.The routines are included in Appendix B. The UART port is configured to output to a terminal on a regular PC. Data can be input and output through this terminal.

6 Experimental Results

The radix-2 modular exponentiation design has been implemented and tested on theAltera EPXA10 Excalibur development board. The area consumption and execution time measurements are discussed in Section 6.1. The performance of the accelerated SRP protocol can be found in Section 6.2.

6.1 Implemented Radix-2 Design

The radix-2 design has been tested on the Altera Excalibur EPXA10 development board for an array length of 128, 512, 1024 and 2048 bits. Thus the modulus can be at maximum 127, 511, 1023 and 2047 respectively due to extra bit that is needed in intermediate results. Testing is done by comparing the software and hardware result where a series of random values is selected for the input parameters. A terminal in the PC (e.g. Hyper-terminal) is used to input parameters and display results. The ARM processor is connected with a PC through the on-board UART port. The accelerator works correctly when the results from the modular exponentiation returned by the MIRACL exponentiation function are identical to the results from the hardware modular exponentiation in the PLD.

During the experiments, the ARM processor was configured at a frequency of 50 MHz and the PLD at 33 MHz. The Quartus software was used together with the ARM Developer Suite [2, 3] to compile and simulate the designs. The hardware part of the design is implemented in VHDL and Altera standard components from the Quartus II software. Leonardo Spectrum is used for synthesis. The Quartus software produces a .hex file that can be downloaded into the flash memory of the Excalibur development platform. The compiler returns values for the maximum frequency and the number of LEs needed in its compilation report. These values are shown in Table 6.1.

Table 6.1: Clock cycles for radix-2 designs implemented in Excalibur.

Modulus	LEs	RAM-bits	fmax
511 bit	5149	3584	65.37 MHz/15.279ns
1023 bit	9644	5120	65.11 MHz/15.3ns
2047 bit	18186	10240	58.01 MHz/ 17.234ns

The Quartus compiler appeared to be very sensitive to small changes. Even one extra register can result in a decrease in fmax of 10 MHz. Another was that some compilations ran successfully but nevertheless produced a design that did not work in the Excalibur board. This was solved by minor changes in the VHDL code. Because of these limitations and long compilation times the 2047 bit version has not been tested for its whole input range on the Excalibur development board. The compilation of the 2047 bit version shows how far the maximum frequency drops when a larger percentage of LEs is used. From Table 6.1 can be seen that the frequency for 2047 bit is lower than for both 511 and 1023 bit designs. The 511 and 1023 bit designs have been tested and are working correctly on the development board for the randomly selected inputs. The 511 bit design was also compiled on 1/8 of the PLD. In that part, 95% of the LEs was used at an fmax of still 55 MHz.

Table 6.2: LE consumption for the radix-2 modular exponentiation.

	LEs	#	Totals
Processing units	70	128	8960
State machine 1 (mult-control):			178
State machine 2 (exp-control):			26
32to8 bit multiplexors:	8	2	16
16to8 bit multiplexors:	8	2	16
8to1 bit multiplexors:	5	4	20
others:			10
Total:			9226

Logic Element (LE) usage can be verified by comparing the results of Table 6.1 with the computations. The total LE usage can be computed by taking the LE usage of each component in the design times the number of times it is used. For 1023 bit, the computation is done as in Table 6.2. The control between stripe and array is not included in this computation. This accounts for the remaining 9644-9226=418 LEs. The large amount of LEs needed for transportation is caused by registers to store the large data-sizes from both DPRAM ports. Two times 32 bits have to be registered several times, addressed and sent through multiplexers in the state machines. The focus was on a maximum transfer speed because the read and write procedures only cost little amount of LEs relative to the array of processing units.

The amount of clock cycles required to perform a full modular exponentiation was measured with a counter in the PLD. The counter starts when the software enables the hardware exponentiation and stops immediately after the results have been written to the dual ported RAM. In Table 6.3 is argued how many clock cycles a computation should take. Table 6.4 shows how many have been measured by the counter and how many full exponentiations can be performed per second. For exponents of 1023 bits the difference in execution time between software and hardware becomes noticeable in the Excalibur device.

The performance of the design in the Excalibur can be compared with the perfomance of Blum and Paar's design in the Xilinx FPGA. The results of both designs are shown in Table 6.5. The PLD in the Excalibur operates on a lower voltage than the Xilinx device used by Blum and Paar (XC4000 series) and therefore has a smaller execution time T and a higher maximum frequency. Area consumption is complicated to compare because the structure of the Xilinx FPGA is different from Altera PLD. A Xilinx CLB contain about two times as much logic as an Altera LE. In area consumption, Xilinx and Altera are not much different for this modular exponentiation design.

Table 6.3: Clock cycles for radix-2 exponentiations with p the size of the array.

Loading M into array	2
Start loading M into array	$p/8-2$
Read A and E from DPRAM into ESB/RAM	$p/8$
Start loading B into array	2
Computations - squaring	$(p+3)(p+2)$
Computations - multiplying	$(p+3)(p+2)$
Clearing of registers after each MMM	$2(p+2)$
Loading of results to DPRAM	$p/4$
Total	$2(p+4)(p+2)+0.25p$

Table 6.4: Clock cycles for a full radix-2 modular exponentiation.

Modulus	clock cycles/mod.exp	msec./mod.exp.	mod.exp./sec.
511 bit	530,704	8.24	65M/530,704 = 121.3
1023 bit	2,109,968	32.4	65M/2,109,968 = 30.86
2047 bit	8,414,224	145.0	58M/8,414,224 = 6.894

Table 6.5: Comparison with modular exponentiation design of Blum and Paar.

	Radix-2 design		Blum Radix-2 [4]		Blum Radix-16 [4]	
Modulus	LEs	T (ms)	CLBs	T (ms)	CLBs	T (ms)
511 bit	5149	8.24	2555	9.38	3413	2.93
1023 bit	9644	32.4	4865	40.50	6633	11.95
2047 bit	18186	145.0	-	-	-	-

6.2 Performance of Accelerated SRP

The radix-2 modular exponentiation in the PLD can be integrated in the SRP protocol C-code for the ARM9 processor. Exponents a and b in the SRP protocol are at most 256 bit width (see Table 3.2) which is much less than the full modulus of 511, 1023 or 2047 bits. The hardware exponentiation can be configured (Exp-control module) such that the execution time increases proportional with the size of the exponent as long as it is a power of 2. The exponent address-counter can be stopped as soon as all bits of the exponents have been processed. These modifications are implemented and it results in the amount of clock cycles for an exponentiation in the SRP protocol for a basic radix-2, 1-ary design as listed in Table 6.6.

Figure 6.1 shows the execution times of the SRP protocol with hardware accelerated exponentiation. The measurements are included in Appendix A. g^x, g^a, g^b and the exponentiation for the session-key S are computed in hardware. From the figure can be seen that computation of the session-key takes more time than the other exponentiations. This is caused by additions and multiplications in the computation of S. The server computes v^u in Step 5 of the SRP protocol, where u is 32 bits width, in software. The execution time of this step is significant but performing it in hardware would not be faster because the hardware always runs through the maximum length of the exponent. Also in g^x the exponent is smaller (160 bits). The efficiency of the hardware accelerator would be a little higher when the hardware is adapted to stop after all bits of the exponent have been processed. This requires extra information to be send to the hardware exponentiation unit together with the enable signal. The Exp-control counter (Figure 5.3) can use this information to stop counting when enough bits of the exponent have been processed. This would reduce the total execution time of the server so much that it becomes smaller than the client's execution time.

Table 6.6: Modular exponentiation when the exponent is 256 bits.

Modulus	Clock cycles HW PLD	Clock cycles SW. ARM
511 bit	265,352	3,091,610
1023 bit	527,492	9,804,060
2047 bit	1,051,778	31,083,956

The ARM9 can run at maximum at 200 MHz and the PLD up to the fmax depicted in Table 6.1. For comparison, the frequency of the ARM9 is set on 200 MHz and the PLD on 50 MHz. The difference in execution time as a result of the hardware acceleration is shown in Figure 6.2. The speed-up increases with the size of the modulus. For a modulus of 1023 bits, the hardware implementation of the exponentiation is about four times faster than the software implementation. Since exponentiation takes most of the execution time of the SRP a speed-up of this function will result in a similar speed-up of the whole protocol.

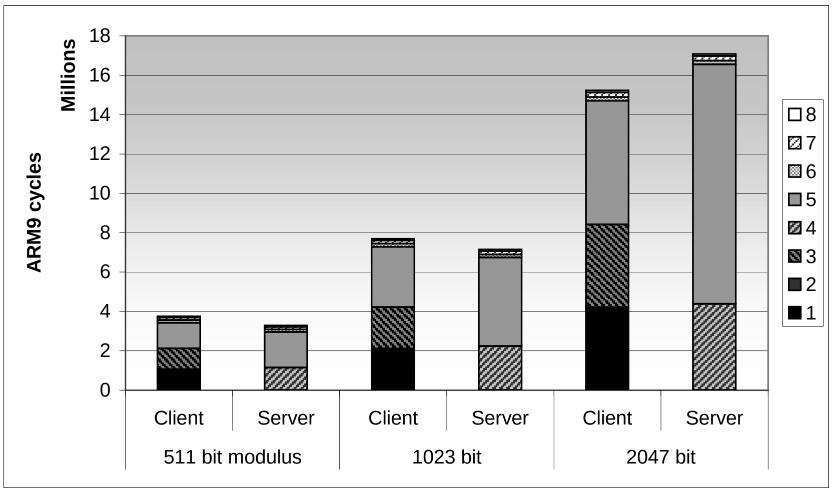

Figure 6.1: The eight steps of SRP with accelerated exponentiation.

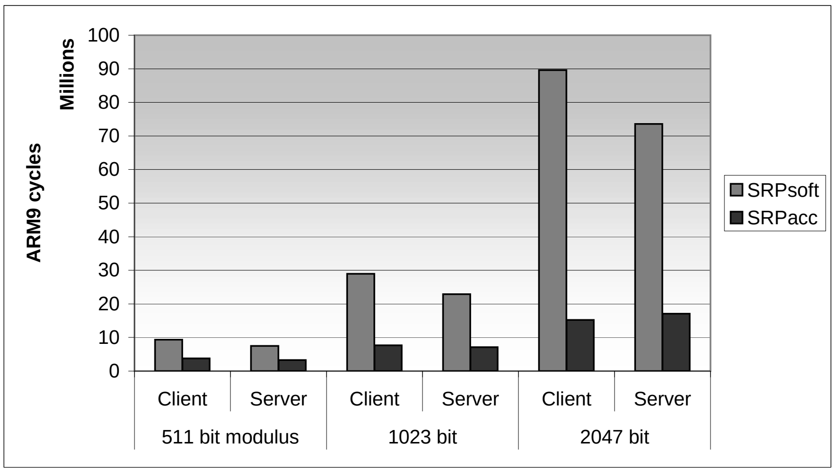

Figure 6.2: SRP with (SRPacc) and without HW acceleration (SRPsoft).

Extensions to Implemented Exponentiation 7

The implementation of the radix-2 exponentiation has provided knowledge about the characteristics of the chip and the development software. The array of processing units occupies only 70 LEs per 8 bits of the modulus which leaves large percentage of the Excalibur chip's resources unused. The design can be upgraded to compute modular exponentiations in less clock cycles. The algorithms in Chapter 4 allow the design of faster accelerators. This chapter presents possible modifications to the implemented radix-2 design to make it faster (Section 7.1 and 7.2). Clock cycle and area consumption estimates are given in Section 7.3

7.1 Reduction of Clock Cycles

The radix-2 array architecture performs multiplications at one bit of the multiplier per cycle and computes two multiplications simultaneously. A full multiplication takes $2 \times (m + 3)$ cycles where m is the bit-length of the modulus. According to the Square and Multiply Algorithm this needs to be repeated n times to perform a full exponentiation where n is the bit-length of the exponent. As demonstrated in the previous chapter, the radix-2 design needs at minimum $2 \times (m+3) \times (n+2)$ clock cycles to complete a full modular exponentiation $X^E \bmod N$. The execution time can be reduced in the following ways:

1. Compute more than 1 bit per Montgomery multiplication (higher radix).
2. Compute more than 1 bit per iteration of the exponentiation (M-ary).
3. Compute multiplications and squaring in parallel.

A fourth way to reduce clock cycles is to skip the multiplications in the Multiply and Square algorithm if the i-th bit of the exponent in Algorithm 4.1 is a zero. This can reduces the average execution time by $0.5n$. Drawback is that the worst case execution time is not improved and the chance that all bits are zero decreases quickly when several bits of the exponent are selected per iteration, as happens in M-ary.

7.1.1 Higher Radix

In higher radix, several bits of the multiplicand A are inserted into the units instead of only one. Radix-16 means that 4 bits are processed per clock. A radix-16 Montgomery multiplication is designed for Xilinx FPGA in [5]. Approximately 4 times less clock cycles are needed to do a full multiplication. Contrary to the radix-2 design, the inverse modulus is needed as in Algorithm 4.3. The inverse modulus N' mod 2^4 can be easily computed as follows:

Algorithm 7.1 *Mod-inverse for radix-2^4 [22].*

```
1.  $y_1 = 1$
2.  FOR $i = 2$ to $4$ DO
3.      IF $2^{i-1} < x \times y_{i-1} \bmod 2^i$ THEN $y_i = y_{i-1} + 2^{i-1}$
4.      ELSE $y_i = y_{i-1}$
5.  END FOR
6.  RETURN $y_4$
```

Example 7.1 *Mod-inverse x=13 at radix 16.*

$2^1 > 13 \bmod 4 = 1 \quad \Rightarrow \quad y_2 = 1$
$2^2 < 13 \bmod 8 = 5 \quad \Rightarrow \quad y_3 = 1 + 4 = 5$
$2^3 > 13 \cdot 5 \bmod 16 = 1 \quad \Rightarrow \quad y_4 = 5 \Rightarrow$ *inverse of* $13 \bmod 16$ *is 5*

The time to compute the inverse is negligible since the inverse is small compared to the modulus. The higher radix Montgomery multiplication algorithm can be derived from Algorithm 4.3 in the same way as was done for the radix-2 design [6]. This results in Algorithm 7.2.

Algorithm 7.2 *Radix-2^k Montgomery modular multiplication [6].*
$N* = (N' \bmod 2^4)N$
q consisting of k bit
Input parameters:
N: modulus of m bits $N < R$, $R = 2^{(k(m+2))}$
A: multiplier of $l + k = k \cdot a$ bits, $A < 2^k N$, $A = \sum_{i=0}^{l-1} a_i (2^k)^i$
B: multiplicand of $l + k$ bits, $B < 2^k N$
Output parameters:
S: $ABR^{-1} \bmod (2^k N)$

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

Where $N*$ is the modulus times its negative inverse N' as determined in Algorithm 7.1. This removes the multiplication with N' from step 3 in Algorithm 4.3 which simplifies computation of q_i. Because division are done by 2^k, the output can be $2^k - 1$ times larger than the modulus N and subtractions of N are needed to get the correct result. These final subtractions can be done in software since they are needed only once per exponentiation.

Most of the implemented radix-2 design can also be used for a higher-radix version. The 1x8 multipliers in Figure 5.4 should be enlarged to 2x8 for radix-4 or 4x8 for radix-16. A radix-16 would require two 4x8 bit multipliers in each processing unit. This, however is rather expensive in terms of LE consumption: more than 24 LEs for a 2x8 multiplier and more than 50 LEs for a 4x8 multiplier.

To omit the need for multipliers, all possible outputs of the products a_iS and q_iN in Algorithm 4.3 can be pre-computed and stored in ESB/RAM. For a 1023 bit exponentiation, 2048 bits have to be read from RAM per clock cycle, which requires 128 ESB blocks where each of them can access 16 bits per clock cycle. The largest Excalibur device has 160 RAM/ESB blocks of 2048 bits. This is enough space to store up to 128 multiples of both products a_iS and q_iN where the length of the modulus is 1023 bits. One block storesthe multiples of a 16 bit part of either a_iS or q_iN.

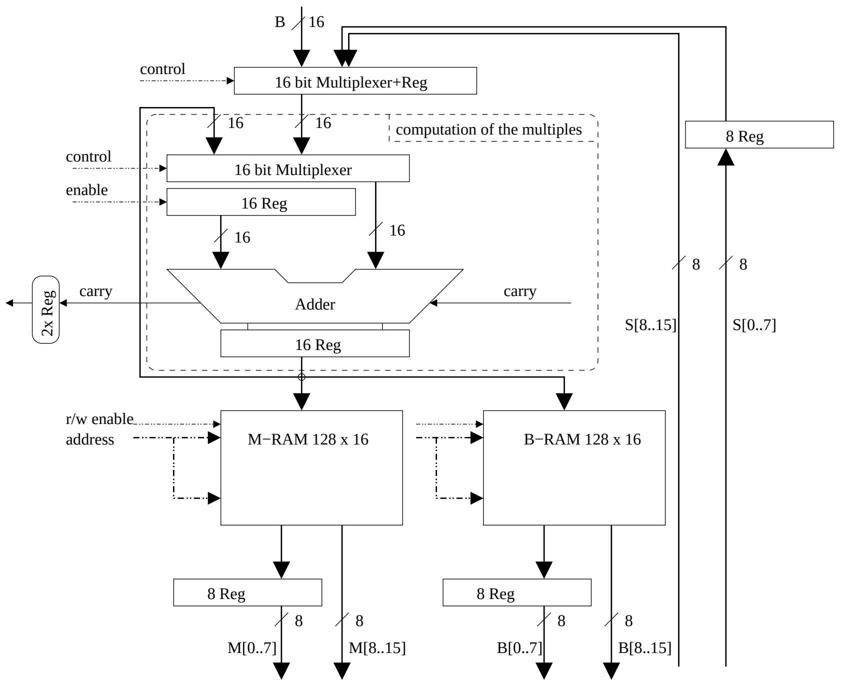

Figure 7.1: Storage of higher radix multiples in ESB/RAM blocks.

The outputs of a_iS have to be pre-computed for each iteration of the Square and Multiply algorithm. For a radix-2, 1 multiple is needed, radix-4, 3 multiples, radix-8, 7 multiples and so on. Thus extra clock cycles are required for the reloading process. Computation of the multiples is done by consecutive addition. One 16-bit multiplexor and a 16-bit adder are used to perform the pre-computations. The multiplexor selects between the first input and a sum of first inputs. The additions

Table 7.1: Additional LE consumption per 16 bits to go from radix-2 to radix-16.

16 bit multiplexers:	32
16 bit carry-adder:	20
enables:	10
registers:	24
addressing (ai and qi):	16
obsolete 1x8 multipliers:	-32
obsolete multiplexer ai/qi:	-20
Total LEs per 16 bit:	50

and storage are shown in Figure 7.1. Two of these expansion units are needed to store the multiples of N and B for two 8 bit processing units. The extra LEs that are needed per 16 bits to transform the radix-2 design into a radix-16 are listed in Table 7.1. Addressing for the RAM blocks takes 2 LEs per bit of a_i and q_i per processing unit. Together, approximately 50 extra LEs are needed per 16 bits of the modulus. For a modulus of 1023 bits and radix-16 this will be about 3200 extra LEs.

Up to radix-16, the delay caused by computing the new multiples is rather small relative to the actual multiplications. For example for a 1024 bit radix 16 the multiplication takes $(1024/4) \times 2 = 512$ cycles while reloading of multiples takes 16 cycles. For radix higher than 16 the reloading time becomes significant because the number of multiples increases with the power of two.

A radix 2^6 or 2^8 can be considered. The storage of a_i and q_i between processing units will cost more registers: for 2^8, 16 registers per unit are needed. Another issue is that $2^8 = 256$ multiples do not fit anymore in the chip's RAM blocks. This can be solved by storing only the outputs of a_iS and q_iN for odd values of a_i and q_i. The outputs for the even values can be computed by reading the appropriate odd value from RAM and add S respectively N one time. This requires extra adders and multiplexors. Per 16 bits of the modulus, one 16-bit adder is needed in the pre-computations and four 8-bit adders and four 8-bit multiplexors are needed in the post-computations.

7.1.2 M-ary Method

The M-ary algorithm processes several bits of the exponent per multiplication in the Square and Multiply algorithm. Algorithm 5.3.2 can be adapted to Algorithm 7.3 to allow processing of 2 bits per Multiply and Square [16].

Algorithm 7.3 *2-ary Square and Multiply.* $P = X^E \bmod N$ *[16].*
Input parameters:
X*: number to be exponented consisting of* m *bits*
E*: exponent of* n *bits*
N*: modulus of* m *bits*
Output parameters:

$P = X^E \bmod N$

```
1.  $Z = X,\ P_0 = 1,\ P_1 = 1,\ P_2 = 1$
2.  FOR $i = 0$ to $n - 1$ DO
3.      IF $e_i = 01$ THEN $P_0 = Z \cdot P_0 \bmod N$ (multiplication)
        IF $e_i = 10$ THEN $P_1 = Z \cdot P_1 \bmod N$ (multiplication)
        IF $e_i = 11$ THEN $P_2 = Z \cdot P_2 \bmod N$ (multiplication)
4.      $Z = Z^2 \bmod N$ (square)
5.      $Z = Z^2 \bmod N$ (square)
6.  END FOR
7.  $P_1 = P_1 \cdot P_2,\ \ P_0 = P_0 \cdot P_1$
8.  $P = P_2 \cdot P_1 \cdot P_0$
```

Processing 2 instead of 1 bit of the exponent reduces the number of clock cycles with almost 25%. The M-ary method requires parameter P in Algorithm 4.1 to be stored several times. If, for example, 2 bits of the exponent are processed per multiplication, three Ps have to be stored. The first P is modified if the bits are '01', the second if the bits are '10' and the third if the bits are '11'. In the end $P2 = P2 \cdot P3$ and $P1 = P1 \cdot P2$ and $P = P3 \cdot P2 \cdot P1$ are computed to get the final result. These multiplications are like regular multiplications in the already available array of processing units. For example $X^{31} = X^{011111bin}$ returns $P1 = X^{16}$, $P2 = X^0$, $P3 = X^5$ in step 5. After step 6 of Algorithm 7.3 this will become $P1 = X^{21}$, $P2 = X^5$, $P3 = X^5$ and $P = X^{31}$ after step 7. The M-ary method can be used on any number of bits but at a certain point so many post-multiplications are needed that the benefit disappears. The M-ary method can easily be used to process 2, 3 or 4 bits of the exponent at once, the amount of post-multiplications is still acceptable.

Implementation of the M-ary method requires an extra ESB/RAM block for each P and LEs to build multiplexors for selection of the appropriate P. Since at most 8 bits are selected at each clock cycle, the cost of the multiplexor is small: a 2-ary with 3 multiples requires 16 LES, a 4-ary, with 15 multiples requires 80 LEs on hardware to select the appropriate P. Furthermore, the control unit may increases in size, especially due to at least two extra states to cover the extra post-multiplications.

7.1.3 Two Parallel Arrays of Processing Units

Next to M-ary and higher radix, 2 parallel arrays, one for squaring and one for multiplying can be considered. This generally requires twice as much area at a speed gain of a factor 2. Parallel arrays require the processing units to be single cycle. This is possible up to radix-16 when 4 out of 8 result bits are connected to the lower neighboring unit without being registered. Another modification is that instead of shifting between the adders in a processing unit, the entire inputs are shifted. An extra benefit of parallel squaring and multiplying is that squaring can be done in higher radix, while the M-ary method can reduce the number of multiplications simultaneously.

Table 7.2: Clock cycles for high radix/M-ary exponentiations where p the bit-length of the modulus.

—serial square and multiply—

Radix	pre-mult	1-ary	2-ary	4-ary
2^1	-	2pp	1.5pp	1.25pp
2^2	4p	1pp	.75pp	.63pp
2^4	16p	.5pp	.38pp	.33pp
2^6	64p	.33pp	.25pp	.21pp
2^6*	32p*	.33pp	.25pp	.21pp
2^8*	128p*	.25pp	.19pp	.16pp

—parallel square and multiply—

Radix	pre-mult	1-ary	2-ary	4-ary
2^1	-	1pp	-	-
2^2	4p	-	.5pp	-
2^4	16p	-	-	.25pp

7.2 Combination of M-ary and Higher Radix

The clock cycles can be computed by combination of the M-ary, higher radix and parallel methods. The pre and post-multiplications are not included in these formulas.

1. M-ary: $p \times p + p \times p/$number-of-bits-of-exp
2. Higher radix: $2 \times p \times p/$number-of-bits-per-mult
3. parallel: $2 \times p \times p/2$

An overview of combinations of the techniques and how they can reduce the number of clock cycles is given in Figure 7.2. Variable p in the figure is $n + 3$, where n is the length of modulus, ground and exponent. The three extra iterations are needed to cover of Montgomery multiplications and the extra multiplications needed for transformation to and from Montgomery residues.

The radix marked with a * are using ESB/RAM blocks and extra adders to compute the multiples. For radix higher than 2, p can be 1024 bits at maximum because of the limited port size in the ESB/RAM blocks. A p of 2048 would be possible to implement for radix-4, radix-16 and even radix-64 but at a twice as long execution time due to a maximum port size of 16 on the 160 ESB/RAM blocks. This option is therefore not considered. For radix 2^6 and 2^8, the pre-multiplication is a serious factor in the total execution time. Radix higher than 2^4 does not allow the 8-bit wide processing unit to operate in single-cycle mode. Thus the cycles for 2-ary and 4-ary, radix higher than 16 are only valid if a 16-bit wide processing unit is implemented first.

Extension to 2 and 4-ary need post-computations: 6 Montgomery multiplications for 2-ary and 30 for the 4-ary or 6.180 and 30.900 cycles respectively when computed in a radix-2 array of processing units with length of 1024 bits.

7.3 Execution Time and Area Usage Estimates

For the faster designs, clock cycles and area consumption have been computed. The faster designs have not been implemented thus verification on the Excalibur board is not possible and no fmax can be determined for these designs. Table 7.1 displays the additional LE consumption in the array per 8 bits if the radix-2 is transformed to a radix-16. This includes the extra registers that are required to transport a_i and q_i between units. For a radix-16, these are both 4 bits wide, thus 8 registers or LEs per unit are needed. The state machine will become larger to cover the pre-multiplications. Final subtractions to get the result smaller than the modulus are needed, this has to be done only once per exponentiation and therefore should be computed in software. The radix marked with a * need three extra 8-bit adders and 2 8-bit multiplexors per processing unit because only half of the multiples is stored in these designs.

Table 7.3: Extra LE consumption for higher radix expansions.

	radix-4	radix-16	radix-64	radix-64*	radix-256*
pre-comp. and storage	25	25	25	25	25
extra aiqi regs.	-4	0	4	4	8
post-computations	-	-	-	40	40
Total LEs for 128 units (1024 bits)	2688	3200	3712	8832	9344

The higher-ary designs are not so demanding in terms of LE usage as higher radix designs. The expansion need mostly RAM blocks and in addition a multiplexor to select the appropriate RAM block and some extra control, mostly to deal with the extra multiplications after the regular multiplications. A multiplexor to select 1 out of 3 in a 2-ary takes 2 LEs. This for 8 RAM outputs comes to 16 LEs. Selecting 1 out of 15 for 4-ary costs 10 LEs, or 80 LEs for 8 RAM outputs. Addressing and extra control for M-ary takes approximately another 200 LEs. Figure 7.2 and 7.3 show the ESB and the LE usage respectively for a 1024 bit full modular exponentiation when the higher radix and M-ary methods are applied.

The number of clock cycles required by the methods sketched above can be computed by combining the formulas for M-ary, higher radix and parallel (Table 7.2). Cycles have been estimated for radix-4, 16, 64, radix-64 and 256 with only the odd half of the possible products stored in RAM and parallel arrays for radix-2, 4 and 16. The results are shown in Figure 7.4 (see Appendix A for formulas). Pre and post-computations are included in the execution times. Since the longest

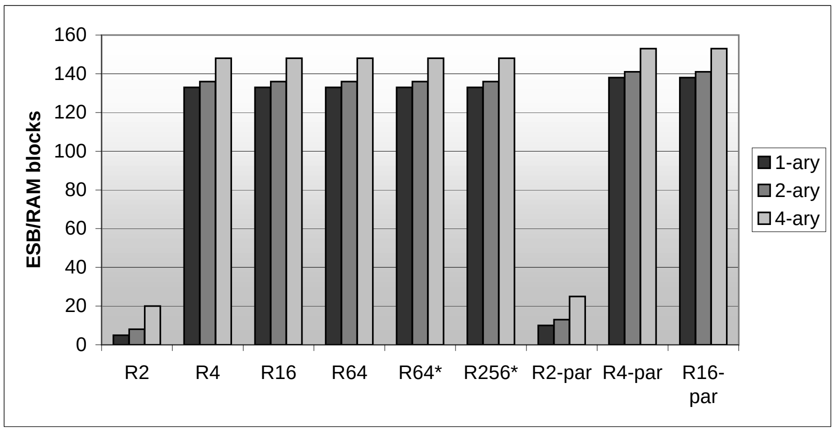

Figure 7.2: ESB usage for 1023 bit full exponentiations.

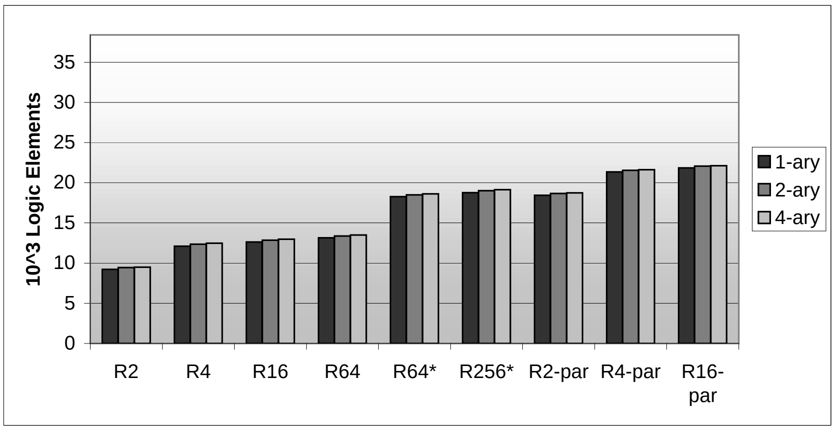

Figure 7.3: LE usage for 1023 bit full exponentiations.

path within the units does not increase significantly for higher radix and not at all for M-ary, it is assumed that a clock frequency of 50 MHz is possible (25% smaller than the maximum frequency of the radix-2 design). The figure shows that the improvement for radices beyond 16 is small. Also the effect of the M-ary method decreases for higher radices due to larger number of cycles that is needed for pre/post-multiplications.

By using higher radix and M-ary (Figure 7.4) a reduction in clock cycles of a factor five can be obtained. The fastest design is the radix-16, 4-ary parallel which requires less than 290.000 clock cycles, followed by radix-64, reduced pre-computations and radix 256, at 390.000 cycles. Radix 64 and 256 could produce better results for 2-ary and 4-ary if the processing units width is extended to 16 bits. This however results in an increased longest path and a serious decrease in fmax. During a test with an array of 16 bit processing unit, compilations in

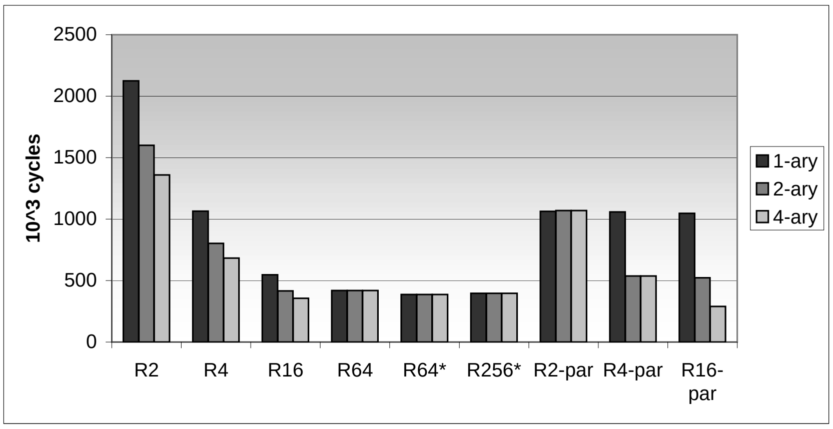

Figure 7.4: Clock cycles for 1023 bit full exponentiation.

the Quartus showed a decrease in fmax. For a 1024 bit array consisting of 16 bit processing units the fmax dropped below the 45 MHz. Thus higher-ary is not considered a useful option to obtain a smaller execution time.

All designs fit on the EPXA10 device, the largest Excalibur device. The radix-2 designs for up to 1023-bit modulus also fit on the EPXA4 device. For higher radix designs, the EPXA10 supplies enough ESB/RAM blocks up to 1023-bits modulus. Expansion to 2047-bit data-width is only possible for radix-2 with 1-ary, 2-ary and 4-ary designs and will take approximately 4 times as many cycles as the radix-2. The fastest design requires less than 290.000 clock cycles at an area consumption of about 25,000 LEs and nearly all ESB/RAM blocks in the PLD. It uses radix-16, combined with 4-ary and two parallel arrays. The design needs over 7 times less clock cycles than the radix-2 design for a 1023 bits modulus, base and exponent. For a clock frequency of 40Mhz in the PLD, the design is still 5 times faster than the radix-2 design and 20 times faster than the pure software exponentiation in the ARM9 at 200 Mhz.

Conclusions and Future Work 8

The SRP protocol is a password authentication protocol suitable for WLANs. With this protocol it is possible to setup a secure connection in a network where any computer can freely move into the coverage area. The SPR protocol is part of the TUTWLAN, a WLAN platform that is being developed at Tampere University of Technology. This WLAN is being implemented on the Altera Excalibur platform which consists of an ARM9 processor and a Programmable Logic Device (PLD). Heavy functions in the protocol require a large amount of clock cycles to compute. This mainly concerns modular exponentiations with large modulus. In this work the execution time of the SRP authentication protocol was reduced by designing hardware accelerators. The modular exponentiation function was implemented in the PLD of the Excalibur platform and connected with the ARM9 processor.

8.1 Conclusions

The hardware accelerated SRP protocol is about 4 times faster for a 1023-bit modulus than a software implementation. For larger moduli the gain increases. A multiplier consisting of an array of radix-2 processing units was designed in the PLD. It can be configured to perform modular exponentiations using moduli of 127, 255, 511, 1023, or 2047 bits. Where the modulus is 1023 bits uses less than 10,000 LEs or about 1/4th of the PLD of the largest Excalibur chip is needed. A full modular exponentiation with inputs of 1023 bits is performed in 2.1M clock cycles with a maximum frequency of about 65 MHz. The hardware exponentiation can be used continuously. The number of full exponentiations that can be computed per second is 30 for a 1023 bit modulus and 60 for a 511-bit modulus.

Faster hardware exponentiations are possible by using higher radix, M-ary and parallel arrays of processing units. The fastest design is one that uses radix-16, 4-ary and has 2 parallel arrays and needs over 7 times less clock cycles than the implemented radix-2 design. This design would run in approximately 290,000 cycles at an area consumption of at most 25,000 LEs and nearly all ESB/RAM blocks in the PLD. The maximum frequency will be lower due to a higher complexity. Nevertheless, if it drops to 40 MHz, it would still be 5 times faster than the radix-2 design. The SRP protocol, which mostly contains modular exponentiation can become about a factor 20 times faster than the pure software implementation on the ARM9 processor, running at 200 MHz.

8.2 Future Work

The faster designs can be implemented for the Altera PLD in future work. An implementation of the radix-16, 4-ary, 2 parallel modular exponentiation might be one of the fastest designs for a modular exponentiation in reconfigurable hardware. M-ary can be added by transforming the units from two-cycle to one cycle units to allow separate computation of multiplication and squaring. Further a new state machine and 3-15 ESB/RAM blocks are needed. Higher radix is more complicated because almost all ESB/RAM blocks are needed in the design and each block needs to be connected to two different processing units. In between multiples need to be computed. Other possible improvements are:

- Modify the design such that it can handle powers-of-two moduli instead of powers of two minus one. This would require one extra processing unit and a modified state machine.

- Make the designs more flexible. The design can be optimized such that it can adapt to the size of the ground and exponent instead of always asuming the maximum size. This would make the hardware more suitable if for example RSA public key encryption is used next to the authentication procedure. The exponentiation can work on exponentiations of different sizes without reconfiguration.

- Perform hashing in reconfigurable hardware. An intellectual property SHA-1 hash function is available from Altera.

- The design presented here can be compiled for other, newer Altera PLDs. This can possible result in faster designs. Newer architectures may have special logic elements that can be configured as a multiplier. In the Excalibur devices, multipliers cannot be implemented efficiently and certainly not as many as are required to feed an entire array of processing units. A PLD with a large number of multipliers allows for new, faster designs for modular exponentiations. Also more ESB/RAM blocks can be useful.

- Communication between PLD and ARM9 is chosen to be as fast as possible in this thesis. The price is a complicated read and load procedure consuming hundreds of LEs. The area consumption of the loading procedure can be reduced by about 75 % when only eight bits are loaded per clock instead of two times 32 bits. Also the possibility to start reading to PLD while the ARM9 is still writing data to higher addresses could be studied.

- So far only arrays of processing units have been used in the design. It was chosen mainly to minimize the risk of routing problems while still be able to implement multipliers and adders for large bit-lengths. Taking a closer look to other types of modular multipliers can be interesting in future research.

Bibliography

[1] P.C. van Oorschot A.J. Menezes and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press Inc., Oct 1996, ISBN 0849385237.

[2] *Altera Homepage*, www.altera.com, 2003, Excalibur HW Reference Manual, APEX 20K Programmable Logic Device Family Data Sheet, Application Note 142 Using the Embedded Stripe Bridges.

[3] *ARM Ltd. Homepage*, www.arm.com, 2003.

[4] T. Blum, *Modular Exponentiation on Reconfigurable Hardware*, Master's thesis, Worchester Polytechnic Institute, Apr 1999.

[5] T. Blum and C. Paar, *Montgomery Modular Exponentiation on Reconfigurable Hardware*, 14th IEEE Symposium on Computer Arithmetic, IEEE, Apr 1999, pp. 70–77.

[6] ———, *High Radix Montgomery Modular Exponentiation on Reconfigurable Hardware*, IEEE Transactions on Computers, vol. 50(7), IEEE, Jul 2001.

[7] N. Borisov, I. Goldberg, and D. Wagner, *Intercepting mobile communications: The insecurity of 802.11*, 7th Annual International Conference on Mobile Computing and Networking, Jul 2001.

[8] E.F. Brickell, *A Fast Modular Multiplication Algorithm With Application To Two Key Cryptography*, Advances in Cryptology - CRYPTO'82 (Santa Barbara, USA), University of California, Aug 1982, pp. 51–60.

[9] W. Diffie and M.E. Hellman, *New Directions in Cryptography*, IEEE Transactions on Information Theory, IEEE, Nov 1976, 22(6), pp. 644–654.

[10] S.E. Eldridge and C.D. Walter, *Hardware Implementation of Montgomery's Modular Multiplication Algorithm*, IEEE Transactions on Computers, IEEE, Jul 1993, 42(6), pp. 693–699.

[11] J. Groszschaedl, *High-Speed RSA Hardware based on Barret's Modular Reduction Method*, Cryptographic Hardware and Embedded Systems (CHES 2000), Lecture Notes in Computer Science, vol. 1965/2000, Springer-Verlag Heidelberg, Aug 2000.

[12] P. Hämäläinen, M. Hännikäinen, M. Niemi, and T. Hämäläinen, *Performance Evaluation of Secure Remote Password Protocol*, IEEE International Symposium on Circuits and Systems (Scottsdale, Arizona, USA), vol. 3, IEEE, May 2002, pp. 29–32.

[13] M. Hännikäinen, M. Knuutila, and et al, *TUTMAC: A Medium Access Control Protocol for a New Multimedia Local Area Network*, IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC 1998) (Boston, USA), vol. 2, IEEE, Sep 1998, pp. 592–596.

[14] M. Hännikäinen, M. Niemi, and T. Hämäläinen, *Performance of the Ad-hoc IEEE 802.11b Wireless LAN*, International Conference on Telecommumications (ICT 2002) (Beijing, China), vol. 1, Jun 2002, pp. 938–945.

[15] *IEEE Standard 802.11b*, http://ieeexplore.ieee.org, 1999.

[16] D.E. Knuth, *The art of computer programming: Seminumerical algorithms*, vol. 2, Addison-Wesley, 2002.

[17] C.K. Koc, T. Acar, and B.S. Kaliski, *Analyzing and Comparing Montgomery Multiplication Algorithms*, IEEE Micro, IEEE, Jun 1996, (16)3, pp. 29–33.

[18] K. Marjo, M. Hännikäinen, T. Hämäläinen, and et al, *Configurable Platform for a Wireless Multimedia Local Area Network*, International Workshop on Mobile Multimedia Communications (MoMuC 1998) (Berlin, Germany), Oct 1998, pp. 301–306.

[19] P.L. Montgomery, *Modular Multiplication Without Trial Division*, Mathematics of Computation, Apr 1985, 44(170), pp. 519–521.

[20] N. Nedjah and L. de Macedo Mourelle, *Two Hardware Implementations for the Montgomery Modular Multiplication: Sequential versus Parallel*, 15th Symposium on Integrated Circuits and Systems Design (Porto Alegre, Brazil), Sep 2002.

[21] Juri Poldre, Kalle Tammemäe, and Marek Mandre, *Modular Exponent Realization on FPGAs*, 8th International Workshop (Tallinn, Estonia), Lecture Notes in Computer Science, vol. 1482, Springer-Verlag Heidelberg, Aug 1998.

[22] T. Ristimäki and J. Nurmi, *Implementation of a Fast 1024-bit RSA Encryption on Platform FPGA*, 6th IEEE International Workshop on Design and Diagnostics of Electronics Circuits and Systems (DDECS'03) (Poznan, Poland), Apr 2003.

[23] B. Schneier, *Applied Cryptography Second Edition*, John Wiley & Sons. Inc., 1996, ISBN 0471128457.

[24] *The Secure Hash Standard*, Federal information processing standards (fips) - publication 180-1, National Institute of Standards and Technology (NIST), USA, 1995.

[25] Shamus Software Ltd, Dublin, Ireland, *M.I.R.A.C.L. Users Manual*, Sep 2002.

[26] W. Stallings, *Network and Internetwork Security: Principles and Practice*, Prentice Hall, 1995.

[27] K. Tikkanen, M. Hännikäinen, T. Hämäläinen, and J. Saarinen, *Advanced Prototype Platform for a Wireless Local Area Network*, The European Signal Processing Conference (EUSIPCO 2000), vol. 4, Sep 2000, pp. 2309–2312.

[28] I. Walrand, *Communication Networks, a First Course*, 2nd ed., McGraw-Hill, 1998, ISBN 0256174040.

[29] C.D. Walter, *An Overview of Montgomery's Multiplication Technique: How to make it Smaller and Faster*, Cryptographic Hardware and Embedded Systems (CHES 1999) (Worcester, Massachusetts, USA), Lecture Notes in Computer Science, Springer-Verlag Heidelberg, Aug 1999.

[30] T. Wu, *The Secure Remote Password protocol*, Internet Society Network and Distributed Systems Security Symposium (NDSS) (San Diego, California, USA), Mar 1998, pp. 97–111.

Measurements and Computations A

The charts in the thesis have been created from measurements and computations. They are described in this Appendix.

A.1 SRP with and without Hardware Acceleration

The performance of the SRP protocol is measured with the ARM simulator. The ARM simulator is part of the AXD Debugger and computes the clock cycles that instruction would take in the ARM processor. The clock cycles are measured per step of the SRP protocol and shown in Figure A.1. Figure A.2 shows the cycles when the exponentiation function is computed in the Progammable Logic Device.

The hardware exponentiation is included as follows. The counter in the AXD Debugger is stopped around the exponentiation function in the SRP protocol. This is replaced by the cycles, measured by the counter in the PLD during a computation of an exponentiation. The counter starts when the control of the array of processing units (Mult-control module) is released from the reset state and stops as soon as it comes back in the reset state. Thus it starts counting when the data is ready in DPRAM and the software had written the number '8' to DPRAM0 address0 to tell the communication control (PLD-control module) to startup computations and stops exactly after the last 32 bits of the result have been written to DPRAM. The results of the counter are outputted to the LEDs on the Excalibur board from where its binary value can be read.

The clock cycles measured by the counter in the PLD are converted to an equivalence of ARM9 cycles. The maximum frequency of the ARM is 200MHz, the PLD is set on 50 MHz, even though it could run up to 65 MHz for some designs. The clock cycles from the counter are multiplied by four and added to the number of cycles measured in the AXD Debugger without the exponentiations.

Measurements are in figure A.2. Computation of S takes more cycles because the value in the table includes also the additions and multiplications in the computation of S.In the server this also includes a 32-bit exponentiation. This explains why those steps take more cycles than a bare exponentiation in step 3.

Clock Cycles measurements of the SRP Protocol simulated on ARM at 200MHz

		512 bit	1024 bit	2048 bit
Client				
	1 g^x	2482340	7936101	24358576
	2 Hash(salt, password)	0	0	0
	3 g^a	3093120	9683980	30203674
	4	0	0	0
	5 S=(B-g^x)^(a+ux)	123872	425977	1597652
		3285128	10509900	32930832
	6 Hash(S)	134474	157241	184043
	7 M1	134564	168096	239589
	8 M2	67990	82733	106403
Server				
	1 lookup	0	0	0
	2	0	0	0
	3	0	0	0
	4 v+g^b	3235469	10070518	32563263
	u= hash(B)_32bit	25042	36737	60364
	5 S=(A*verifier^u)^b	740823	2395184	7958763
		3172099	9981805	32447235
	6 Hash(S)	129279	153977	180820
	7 M1	134564	168096	239589
	8 M2	67990	82733	106403

Figure A.1: Software performance of SRP on ARM9.

Clock Cycles measurements of SRP simulated with accelerated exponentiations (50/200 MHz)

		512 bit	1024 bit	2048 bit
Client				
	1 g^x	1061408	2109968	4207112
	2 Hash(salt, password)	0	0	0
	3 g^a	1061408	2109968	4207112
	4	0	0	0
	5 S=(B-g^x)^(a+ux)	1298309	3064040	6288361
	6 Hash(S)	134474	157241	184043
	7 M1	134564	168096	239589
	8 M2	67990	82733	106403
Server				
	1 lookup	0	0	0
	2	0	0	0
	3	0	0	0
	4 v=g^b, u	1149820	2235418	4383504
	5 S=(A*verifier^u)^b	1802231	4505152	12165875
	6 Hash(S)	129279	153977	180820
	7 M1	134564	168096	239589
	8 M2	67990	82733	106403

Figure A.2: Performance of SRP with hardware accelerated exponentiation.

A.2 Higher Radix and M-ary

The area and clock cycle computations for higher radix and M-ary design are computed as explained in the thesis. The formulas that are used are in table A.1, A.2 and A.3. As an example the formulas are applied to a 1024 bit (modulus of 1023 bit) exponentiation. The results are shown in figure A.3,A.4 and A.5. Radix higher than 16 cannot be used in combination with M-ary for the 8-bit processing

Table A.1: Cycles and LE for higher radix, 1-ary

	Cycles	LEs
R2	$p \times 2 + p \times q \times 2/1$	$r2$
R4	$p \times 4 + p \times q \times 2/2$	$r2 + (25 - 4) \times 128 + state$
R16	$p \times 16 + p \times q \times 2/4$	$r2 + (25) \times 128 + state$
R64	$p \times 64 + p \times q \times 2/6$	$r2 + (25 + 4) \times 128 + state$
R64*	$p \times 32 + p \times q \times 2/6$	$r2 + (25 + 4) \times 128 + state + 5 \times 1024$
R256*	$p \times 128 + p \times q \times 2/8$	$r2 + (25 + 8) \times 128 + state + 5 \times 1024$
R2par	$p \times 2 + p \times q/1$	$r2 \times 2$
R4par	$p \times 4 + p \times q/2$	$r2 \times 2 + (25 - 4) \times 128 + state$
R16par	$p \times 16 + p \times q/4$	$r2 \times 2 + (25) \times 128 + state$

Table A.2: Cycles and LE for higher radix, 2-ary

	Cycles
R2	$p \times 2 + (p \times q \times 1.5 + prepost1)/1$
R4	$p \times 4 + (p \times q \times 1.5 + prepost1)/2$
R16	$p \times 16 + p \times q \times 1.5 + prepost1)/4$
R64	$p \times 64 + p \times q \times 2/6$
R64*	$p \times 32 + p \times q \times 2/6$
R256*	$p \times 128 + p \times q \times 2/8$
R2par	$p \times 2 + (p \times q + prepost1)/1$
R4par	$p \times 4 + (p \times q + prepost1)/2$
R16par	$(p \times q + prepost1)/2$
	LEs
R2	$r2 + ary2$
R4	$r2 + (25 - 4) \times 128 + state + ary2 + 16$
R16	$r2 + (25) \times 128 + state + ary2 + 32$
R64	$r2 + (25 + 4) \times 128 + state + ary2 + 32$
R64*	$r2 + (25 + 4) \times 128 + state + 5 \times 1024 + ary2 + 32$
R256*	$r2 + (25 + 8) \times 128 + state + 5 \times 1024 + ary2 + 32$
R2par	$r2 \times 2 + ary2 + 16$
R4par	$r2 \times 2 + (25 - 4) \times 128 + state + ary2 + 16$
R16par	$r2 \times 2 + (25) \times 128 + state + ary2 + 16$

unit described in the thesis. These alternatives are therefore not economical to implement. A square is put around these combinations in the figures.

The ESB/RAM block usage in table A.4 are computed as follows. A basic design costs 5 different RAM blocks with a maximum capacity of 2048 bits and port size of 16 bits. For a higher radix design 128 extra blocks are needed to store the multiples of M and Z from the multiply and square algorithm. For 2-ary, 3 outputs need to be stored, for 4-ary, 15 outputs need to be stored.

Table A.3: Cycles and LE for higher radix, 4-ary

	Cycles
R2	$p \times 2 + (p \times q \times 1.25 + prepost2)/1$
R4	$p \times 4 + (p \times q \times 1.25 + prepost2)/2$
R16	$p \times 16 + p \times q \times 1.25 + prepost2)/4$
R64	$p \times 64 + p \times q \times 2/6$
R64*	$p \times 32 + p \times q \times 2/6$
R256*	$p \times 128 + p \times q \times 2/8$
R2par	$p \times 2 + (p \times q + prepost2)/1$
R4par	$p \times 4 + (p \times q + prepost2)/2$
R16par	$p \times 16 + (p \times q + prepost2)/4$
	LEs
R2	$r2 + ary4$
R4	$r2 + (25 - 4) \times 128 + state + ary4 + 80$
R16	$r2 + (25) \times 128 + state + ary4 + 160$
R64	$r2 + (25 + 4) \times 128 + state + ary4 + 160$
R64*	$r2 + (25 + 4) \times 128 + state + 5 \times 1024 + ary4 + 160$
R256*	$r2 + (25 + 8) \times 128 + state + 5 \times 1024 + ary4 + 160$
R2par	$r2 \times 2 + ary4 + 80$
R4par	$r2 \times 2 + (25 - 4) \times 128 + state + ary4 + 80$
R16par	$r2 \times 2 + (25) \times 128 + state + ary4 + 80$

Table A.4: Variables for the 1024 bit exponentiation.

	Execution time	LE usage	
n	size of array (128, 512, 1024, 2048)	r2	9226
p	n+6 (ground)	state	200
q	n+6 (exponent)	ary2	200
prepost1	6180	ary4	360
prepost2	30900		

execution time - 1024 bit array

	Radix	pre-comp	1-ary	2-ary	4-ary
1	R2	2060	2123860	1599590	1359085
2	R4	4120	1065020	802885	682633
4	R16	16480	546930	415863	355736
6	R64	65920	419553	419553	419553
6	R64*	32960	386593	386593	386593
8	R256*	131840	397065	397065	397065
1	R2-par	2060	1062960	1069140	1093860
2	R4-par	4120	1058840	537660	550020
4	R16-par	16480	1046480	522725	289430

*=precompute and extra adders

Figure A.3: Clock cycles for 1024 bit full exponentiation.

ESB usage

	Radix	1-ary	2-ary	4-ary
1	R2	5	8	20
2	R4	133	136	148
4	R16	133	136	148
6	R64	133	136	148
6	R64*	133	136	148
8	R256*	133	136	148
1	R2-par	10	13	25
2	R4-par	138	141	153
4	R16-par	138	141	153

*=precompute and extra adders

Figure A.4: ESB usage for 1024 bit full exponentiation.

LE usage -1024 bit array

	Radix	1-ary	2-ary	4-ary
1	R2	9226	9442	9506
2	R4	12114	12346	12474
4	R16	12626	12858	12986
6	R64	13138	13370	13498
6	R64*	18258	18490	18618
8	R256*	18770	19002	19130
1	R2-par	18452	18668	18732
2	R4-par	21340	21556	21620
4	R16-par	21852	22068	22132

*=precompute and extra adders

Figure A.5: LE usage for 1024 bit full exponentiation.

Radix-2 Design B

The radix-2 basic design has been implemented and tested on an EPXA10 Excalibur development board. This appendix contains figures of the state machines and block schematics of the design. It also includes the VHDL code of the processing unit and C code of the software in ARM9.

B.1 Software C code in ARM9

The software routines can read or modify MIRACL big variables. The modular exponentiation is spit in three parts. One to read the modulus M and RxR mod M trough the DPRAM to the PLD (modloading). This will prepare the hardware to do regular modular exponentiations. The regular ground and exponent are loaded to PLD in the second routine (exploading). The result is read back from DPRAM to a big variable in the third routine (resreading).

```
------------main.c(1024bit)------------
#include <stdio.h>
#include <string.h>
#include <miracl.h>
#include "uartcomm.h"
#include "..\stripe.h"

#define width unsigned long int
width DPRAM_DATA_A[32];//unused global variable
width DPRAM_DATA_E[32];//unused global variable
width DPRAM_DATA_M[32];//unused global variable
width DPRAM_DATA_B[32];//unused global variable
width DPRAM_Result[32];//unused global variable

void EnableIRQ(void);
void delay(unsigned int);

static long unsigned int DPRAM_DATA[32];//'temp' variable
static big g, N, E, V, u, u2, u3;
static miracl *mip;

int main(void)
```

```
{
  volatile unsigned int* const PLD_Reg0_Address =
(volatile unsigned int* const) EXC_PLD_BLOCK0_BASE;
  char str [10];
  uart_init();
  EnableIRQ(); // Enable processor interrupts

  mip = mirsys(256, 0);//Miracl initialization

// *** software and hardware modular exponentiation***

  N = mirvar( 0);
  g = mirvar( 0);
  E = mirvar( 0);
  V = mirvar( 0);
  u = mirvar( 0);
  u2 = mirvar( 0);
  u3 = mirvar( 0);

  *PLD_Reg0_Address = 0x00;// put HW exponentiation trough AHB slave in reset
  printf ("START of MME exponentiation\r\n");
  *PLD_Reg0_Address = 0xAA;// enable HW exponentiation trough AHB slave in PLD

  mip->IOBASE = 16;// radix of the modulus at input
//printf (" IOBASE is 16 \r\n");
  modloading(128, DPRAM_DATA, N, u2, u, u3);

// ***start of loop***
  forever
  {

// ***input of variables and HW modular exponentiation***
mip->IOBASE = 10;// radix of the base and exponent at input
//printf (" IOBASE is 10 \r\n");
exploading(128, DPRAM_DATA, g, E);

mip->IOBASE = 16;
resreading(128, DPRAM_DATA, V);

mip->IOBASE = 10;
printf( "\r\nHardware Result = ");
cotnum( V, stdout);

// ***software modular exponentiation***
powmod( g, E, N, V );
printf( "\r\nSoftware Result = ");

cotnum( V, stdout );
printf("\r\nEnd");
}
// ***end of loop***
```

```
  mirkill( N);
  mirkill( g);
  mirkill( E);
  mirkill( V);
  mirkill( u);
  mirkill( u2);
  mirkill( u3);

// *** end of modular exponentiation***
  return 0;
}
```

The three routines modloading, exploading and resreading are as follows. Modloading reads the modulus (big N) and writes it to DRPAM1 address 96-127. Then it computes RxR mod M from it (big u, u2, u3) and writes it to DPRAM0 address 96-127. Expreading reads ground (big g) and exponent (big E) to DPRAM0 address 32-63 respectively to DPRAM1 address 32-63. The hardware is enabled from these functions by writing to an AHB slave in the PLD. During enable, all control communication is done by writing to or reading from DPRAM0 address 0. After computations, the hardware writes the result directly to DPRAM0 64-95. Resreading puts this result to big V.

```
------------modloading------------
int modloading(int sizeinbytes, long unsigned int* DPRAM_DATA,
 big NN, big uu2, big uu, big uu3)
{
  int RR, len, i;
  char mod0[256];
  char str1[257];
  volatile width* DPRAM0_ADDR;
  volatile width* DPRAM_ADDR_M;
  volatile width* DPRAM_ADDR_B;
  DPRAM0_ADDR = (volatile unsigned long int*) EXC_DPSRAM_BLOCK0_BASE;
  DPRAM_ADDR_M = (volatile unsigned long int*) EXC_DPSRAM_BLOCK1_BASE + 96;
  DPRAM_ADDR_B = (volatile unsigned long int*) EXC_DPSRAM_BLOCK0_BASE + 96;


  *DPRAM0_ADDR = 1;  // PLD_control is kept in first state
// in here, a signal can be sent through the stripe-to-PLD bridge to
// activate the PLD part, currently the PLD part is activated in main.
  for (i = 0 ; i < 32; i++) // set global var. DPRAM_DATA to 0
  {
DPRAM_DATA[i]= 0;
  }

// ***loading modulus routine*** (big NN, uu2)

// the modulus is inputted as a string with radix IOBASE and stored in big NN
  RR=sizeinbytes*8;
  printf (" %d ",RR);
  printf ("\r\n Input modulus M (any ODD number): ");
  scanf ("%s",mod0);
  printf ("%s", mod0);
  cinstr( NN, mod0 );

// load modulus to DPRAM
  len = big_to_bytes(sizeinbytes ,NN ,str1, TRUE);
  copybytes2(sizeinbytes, str1, DPRAM_DATA);
  for (i = 0 ; i < 32; i++)
  {
*(DPRAM_ADDR_M + i) = DPRAM_DATA[i]; // write Modulus to DPRAM1 96+i
  }

// computing R = 2^(m+2)
  expint( 2, (RR+1), uu);
  printf( "\n\r U = ");
  cotnum( uu, stdout );

// old fashioned division to compute R*R mod M
  multiply( uu, uu, uu2 );
  divide( uu2, NN, uu3 );
  printf( "\n\r R*R mod M = ");
  cotnum( uu2, stdout );

// load R to DPRAM
  len = big_to_bytes(sizeinbytes, uu2,str1, TRUE);//## exactly 64 bytes in str2
  copybytes2(sizeinbytes, str1, DPRAM_DATA);
  for ( i = 0 ; i < 32; i++)
  {
*(DPRAM_ADDR_B + i) = DPRAM_DATA[i]; // write R to DPRAM0 96+i
  }

// PLD starts reading, software waits until PLD has finished reading from DPRAM
  *DPRAM0_ADDR = 0; // PLD_control will move to second state
  delay(100);
  *DPRAM0_ADDR = 10; // PLD can start reading M and R from DPRAM1/0 address 96
  while (*DPRAM0_ADDR != 3)
  {
 delay(100); // PLD sets DPRAM status to 3 when M and R have been read
  }
  *DPRAM0_ADDR = 1; // PLD_control is kept in first state

  printf ("\r\n Finished writing M and R to PLD");
// a PLD-deactivation signal through stripe-to-PLD bridge can be added here
  *DPRAM0_ADDR = 0;
  return 0;
}
------------exploading------------
int exploading(int sizeinbytes, long unsigned int* DPRAM_DATA, big gg, big EE)
{
  char gen[128];
  char expon[128];
  char str1[129];
  int len, i;
  volatile width* DPRAM0_ADDR;
  volatile width* DPRAM_ADDR_A;
  volatile width* DPRAM_ADDR_E;
  DPRAM0_ADDR = (volatile unsigned long int*) EXC_DPSRAM_BLOCK0_BASE;
  DPRAM_ADDR_E = (volatile unsigned long int*) EXC_DPSRAM_BLOCK1_BASE + 32;
  DPRAM_ADDR_A = (volatile unsigned long int*) EXC_DPSRAM_BLOCK0_BASE + 32;

  while (*DPRAM0_ADDR == 1) // read status flag, should be 0 if PLD is not busy
  {
delay(1000);
  }
  *DPRAM0_ADDR = 1; // PLD_control is kept in first state
// in here, a signal should be sent through the stripe-to-PLD bridge to
// activate the PLD part, currently the PLD part is always activated
  for (i = 0 ; i < 32; i++) // initialize global var. DPRAM_DATA to 0
  {
DPRAM_DATA[i]= 0;
  }

// ***loading  base and exponent routine*** (big g and E)
  printf ("\r\n Input the base A: ");
  scanf ("%s",gen);
  printf (" %s ",gen);
  printf ("\r\n Input exponent E: ");
  scanf ("%s",expon);
  printf (" %s ",expon);
  cinstr( EE, expon );
  cinstr( gg, gen );

// load base to DPRAM
  len = big_to_bytes(sizeinbytes, gg,str1, TRUE);//## exactly 64 bytes in str1
  copybytes2(sizeinbytes, str1, DPRAM_DATA);
  for (i = 0 ; i < 32; i++)
  {
*(DPRAM_ADDR_A + i) = DPRAM_DATA[i]; // write Base A to DPRAM0 32+i
  }

// load exponent to DPRAM
  len = big_to_bytes(sizeinbytes, EE,str1, TRUE);//## exactly 64 bytes in str1
  copybytes2(sizeinbytes, str1, DPRAM_DATA);
  for (i = 0 ; i < 32; i++)
  {
*(DPRAM_ADDR_E + i) = DPRAM_DATA[i]; // write Exponent E to DPRAM1 32+i
  }

// PLD can start reading from DPRAM (after both base and exponent are in DPRAM completely)
  *DPRAM0_ADDR = 0; // PLD_control will move to second state
  printf ("\r\n Finished writing A and E to RAM");
  *DPRAM0_ADDR = 8; // PLD can start reading E and A from DPRAM1/0 address 32
```

```
// NOTE: any routine between this and 'resreading' should not
// write to DPRAM1/0 address 32-63 or DPRAM0 address 64-95
  return 0;
}
------------resreading------------
int resreading(int sizeinbytes, long unsigned int* DPRAM_DATA, big VV)
{
  char str1[300];
  int i = 0;
  volatile width* DPRAM0_ADDR;
  volatile width* DPRAM0_ADDR_R;
  DPRAM0_ADDR = (volatile unsigned long int*) EXC_DPSRAM_BLOCK0_BASE;
  DPRAM0_ADDR_R = (volatile unsigned long int*) EXC_DPSRAM_BLOCK0_BASE + 64;

// wait until PLD has finished computations and written results to DPRAM
  while (*DPRAM0_ADDR != 3)
  {
 delay(1000); // PLD sets DPRAM status to 3 after Result to DPRAM0 64
  }
  *DPRAM0_ADDR = 1; // PLD_control is kept in first state

// ***reading resultsfrom DPRAM routine*** (big V)
  for (i = 1 ; i < 32; i++) // initialize global var. DPRAM_DATA to 0
  {
DPRAM_DATA[i]= 0;
  }

  for (i = 0 ; i < sizeinbytes/4; i++)// ##4->4x32 bits, RR = size of modulus/32
  {
DPRAM_DATA[i] = *(DPRAM0_ADDR_R + i);// read Result from DPRAM0 64+i
  }
  i = 0;
  printf ("\r\n Finished reading results from DPRAM\r\n");

// a PLD-deactivation signal through stripe-to-PLD bridge to be added here
  *DPRAM0_ADDR = 0;
// putting result to a 'big' variable
  bytescopy2(sizeinbytes, DPRAM_DATA, str1);
  cinstr( VV, str1);
  return 0;
}
```

The miracl library provides functions to transfer from a string to big or from big to string. Copybytes2 copies from a character string to integers of 32 bit for storage in DRPAM. Bytescopy2 copies from 32 bit integers back to a string.

```
-----------string_to_integers------------
void copybytes2(int length, char* KKK, long unsigned int* P)
{
/*
* reads the first 'length' characters to long unsigned integers
```

```
*
* receives the 'length' of the input string
* receives a character string of at least 'length' characters of 1 byte each
* outputs an array of 'length'/4 integers of 32 bit each (long unsigned integers)
* the first character of the input string should be smaller than 128
*/
  int i, int jconst, j, h;
  long unsigned int Ptemp, Ptot;
  jconst=length/4;
  j=jconst-1;
  Ptemp= KKK[0];
  if (Ptemp>127)
  {
printf("\r\nWARNING: input value is too big (carry overflow in HW) \r\n");
  }
  for (h=0; h<jconst; h++)
  {
Ptot=0;
for (i=0; i < 4; i++)
{
Ptemp = KKK[i+h*4];
Ptot = Ptemp + Ptot;
if (i != 3)
Ptot = Ptot * 256;
}
P[j]=Ptot;
j--;
  }
}
------------integers_to_string------------
void bytescopy2(int length, long unsigned int* P, char* KKK)
{
/*
* reads an array of 'length'/4 long unsigned integers to a character string
*
* receives the 'length' of the output string
* receives an array of 'length'/4 integers of 32 bit each (long unsigned integers)
* outputs a character string of 'length' char. of 4 bits each (hexadecimal numbers)
*
*/
  int qi, jconst;
  char character2[10];
  char KK[300];
  jconst=(length/4)-1;
  sprintf(KK, "%08lX", P[jconst]);//## first long int to 8xhex-char: 15-> 16x32 bit
  for (qi=jconst-1; qi>=0; qi--) //## others to 8xhex-char: 14-> 16x32 bit
  {
sprintf(character2, "%08lX", P[qi]);
strcat(KK, character2);
  }
  strcpy(KKK,KK);
```

```
}
```

B.2 VHDL Code Processing Unit

The actual computations are performed in the processing units. The VHDL code of the processing unit is included below.

```
---------------------------------------------------
--- multiplexer/register - 8 bit
---------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity Enable_reg1 is
port( data_inS    : in std_logic_vector(7 downto 0);
      SelectZeros : in std_logic;
      data_out_reg: out std_logic_vector(7 downto 0);
      CLK, RESET  : in std_logic);
end Enable_reg1;

architecture behaviour of Enable_reg1 is
signal data_signal: std_logic_vector(7 downto 0);
begin
  process(SelectZeros, data_inS)
  begin
    if(SelectZeros = '1') then
      data_signal <= "00000000";
    else
      data_signal <= data_inS;
    end if;
  end process;
  process(CLK, RESET)
  begin
    if (RESET = '1') then
      data_out_reg <="00000000";
    elsif (CLK'event and CLK = '1' )then
      data_out_reg <= data_signal;
    end if;
  end process;
end behaviour;

---------------------------------------------------
--- multiplexer/register - 8 bit, enable
---------------------------------------------------
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;

entity Enable_reg2 is
port( Data_inB    : in std_logic_vector(7 downto 0);
      Data_inS    : in std_logic_vector(7 downto 0);
      SelectR     : in std_logic;
      data_out_reg: out std_logic_vector(7 downto 0);
      CLK, RESET, ena: in std_logic);
end Enable_reg2;

architecture behaviour of Enable_reg2 is
signal data_signal: std_logic_vector(7 downto 0);
begin
  process(SelectR, ena, data_inB, Data_inS)
  begin
    if(SelectR = '1' and ena = '1') then
      data_signal <= data_inB;
    else
      data_signal <= data_inS;
    end if;
  end process;
  process(CLK, RESET, ena)
  begin
    if (RESET = '1') then
      data_out_reg <="00000000";
    elsif (CLK'event and CLK = '1' )then
      if ena='1' then
        data_out_reg <= data_signal;
      end if;
    end if;
  end process;
end behaviour;

---------------------------------------------
--- 1x8 multiplier (compare) and register
---------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity multiply_with_one is
port( selectbit    : in std_logic;
      data_in      : in std_logic_vector(7 downto 0);
      data_out     : out std_logic_vector(7 downto 0);
      data_out_reg : out std_logic_vector(7 downto 0);
      CLK, RESET   : in std_logic;
      ena          : in std_logic);
end multiply_with_one;

architecture comparing of multiply_with_one is
signal data: std_logic_vector(7 downto 0);
begin
  process(data_in, selectbit)
  begin
    case selectbit is
      when '1' => data <= data_in;
      when '0' => data <= "00000000";
      when others => data <= "11111111";
    end case;
  end process;
  data_out <= data;
  process(CLK, RESET, ena)
  begin
    if (RESET = '1') then
      data_out_reg <="00000000";
    elsif (CLK'event and CLK = '1' )then
    if ena='1' then
      data_out_reg <= data;
    end if;
  end if;
end process;
end comparing;

-----------------------------------------------------
--- 8-bit binary unsigned adder -
-----------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

entity n_bit_adder is
generic(n : integer := 7);   --no. of bits less one
port( data_a, data_b : in std_logic_vector(n downto 0);
      sum      : out std_logic_vector(n downto 0);
      carry_in : in std_logic;
      carry_out: out std_logic);
end n_bit_adder;

architecture adding of n_bit_adder is
signal sum_S, data_a_S, data_b_S: std_logic_vector(n+1 downto 0);
```

```
begin
  data_a_S(n downto 0) <= data_a;
  data_a_S(n+1) <= '0';
  data_b_S(n downto 0) <= data_b;
  data_b_S(n+1) <= '0';
  sum_S <= data_a_S + data_b_S + carry_in;
  sum <= sum_S(n downto 0);
  carry_out <= sum_S(n+1);
end adding;

---------------------------------------------
--- 8 BIT PROCESSING UNIT (version 2)
---------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity bit_multiplication8_2 is
port( B             : in std_logic_vector (7 downto 0);
      SS_in         : in std_logic_vector (7 downto 0);
      SS_out        : out std_logic_vector (7 downto 0);
      Ai_qi, SC_BMC_in: in std_logic_vector(1 downto 0);
      Ai_qi_out, SC_BMC_out_r: out std_logic_vector(1 downto 0);
      Shift_in      : in std_logic;
      Shift_out_r   : out std_logic;
      CLK           : in std_logic;
      EnableBRM_in  : in std_logic_vector(2 downto 0);
      EnableBRM_out: out std_logic_vector(2 downto 0);
      RESET         : in std_logic);
end bit_multiplication8_2;

architecture algorithm_4 of bit_multiplication8_2 is
component n_bit_adder is
  generic(n : integer := 7);
  port( data_a, data_b: in std_logic_vector(n downto 0);
        carry_in    : in std_logic;
        carry_ou t  : out std_logic;
        sum         : out std_logic_vector(n downto 0));
end component;
component multiply_with_one is
  port( selectbit   : in std_logic;
        data_in     : in std_logic_vector(7 downto 0);
        data_out    : out std_logic_vector(7 downto 0);
        data_out_reg:out std_logic_vector(7 downto 0);
        CLK, RESET  : in std_logic;
        ena        : in std_logic);
end component;
component Enable_reg1 is
  port( data_inS   : in std_logic_vector(7 downto 0);
        SelectZeros: in std_logic;
        data_out_reg:out std_logic_vector(7 downto 0);
        CLK, RESET : in std_logic);
end component;
component Enable_reg2 is
  port( Data_inB   : in std_logic_vector(7 downto 0);
        Data_inS   : in std_logic_vector(7 downto 0);
        SelectR    : in std_logic;
        data_out_reg: out std_logic_vector(7 downto 0);
        CLK, RESET, ena: in std_logic);
end component;

signal AiB, qiM, x, xx: std_logic_vector(7 downto 0);
signal SC_BMC_out: std_logic_vector(1 downto 0);
signal EnableBRM_ouS: std_logic_vector(2 downto 0);
signal Shift_out: std_logic;
signal Snew, S_reg, S_out_signal, S_signal, S_new_signal:std_logic_vector(7 downto 0);
signal Breg, M, Mreg : std_logic_vector(7 downto 0);
signal M8 : std_logic;

begin
  minus1: Enable_reg2 port map
    (B, S_out_signal, EnableBRM_in(1), Breg, CLK, RESET, EnableBRM_in(2));
  minus2: Enable_reg1 port map
    (S_reg, EnableBRM_in(2), S_new_signal, CLK, RESET);
  nolla: multiply_with_one port map
    (Ai_qi(1), Breg, AiB, M, CLK, RESET, EnableBRM_in(0));
  kaks:  multiply_with_one port map
    (Ai_qi(0), M, qiM, Mreg, CLK, RESET, EnableBRM_in(0));
  kolm: n_bit_adder port map
    (qiM, S_new_signal, SC_BMC_in(0), SC_BMC_out(0), x);

  Shift_out <= Snew(0);
  xx(6 downto 0) <= x(7 downto 1);
  xx(7) <= (Ai_qi(0) and M8) xor SC_BMC_out(0) xor (Shift_in and (not EnableBRM_ouS(2)));

  nelj: n_bit_adder port map
    (AiB, xx, SC_BMC_in(1), SC_BMC_out(1) , Snew);

  select_res: process(EnableBRM_in(1), S_reg, SS_in)
  begin
```

```
    if (EnableBRM_in(1) ='0') then
      S_signal <= S_reg;
    else
      S_signal <= SS_in;
    end if;
  end process;

  SS_out <= S_out_signal;
  EnableBRM_out <= EnableBRM_ouS;

  synch: process(CLK, RESET, EnableBRM_in)
  begin
    if (RESET = '1') then
      Ai_qi_out <= "00";
      SC_BMC_out_r <= "00";
      Shift_out_r <= '0';
      S_reg <= "00000000";
      S_out_signal <= "00000000";
      EnableBRM_ouS <= "000";
      M8 <= '0';
    elsif (CLK'event and CLK = '1')then
      Ai_qi_out <= Ai_qi;
      SC_BMC_out_r <= SC_BMC_out;
      Shift_out_r <= Shift_out;
      S_reg <= Snew;
      S_out_signal <= S_signal;
      EnableBRM_ouS <= EnableBRM_in;
      if EnableBRM_in(0) = '1' then
        M8 <= B(0);
      end if;
    end if;
  end process;
end algorithm_4;
---------------------------------------------
```

B.3 Quartus Block Design Files

The hardware is designed in three different levels. The first level controls the communication between exponentiation and PLD. The second level connects the units and control within the exponentiation function. The VHDL descriptions of the various components are the third level.

arm_top.bdf (first level) tryout.bdf (second level)

B.4 State Machines

Figure B.3 shows the state diagram that controls reading from and writing to the Dual Ported RAM in the Excalibur Stripe. This statediargam communicates with the software code in the previous section.

In figure B.4, iaddress is the i-th address of the results S_multiply and S_square when it's written to the ESB/RAM block. CounterB and CounterM are used to loed respectively base B and Modulus M from ESB to processing units. The ESB read address is set by j. iSS is the write enable signal for the ESB/RAM blocks.

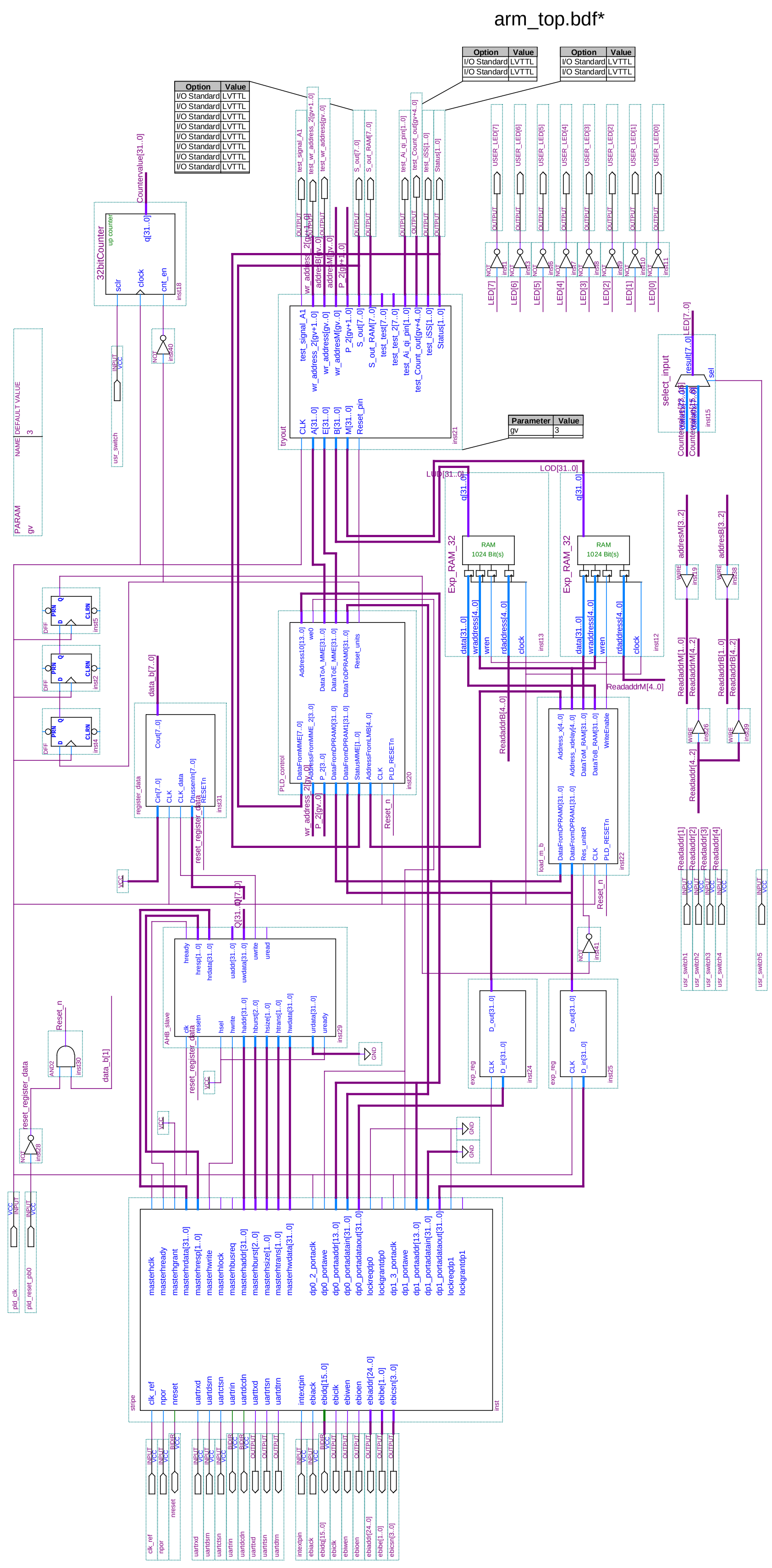

Figure B.1: communication between ARM and exponentiation hardware.

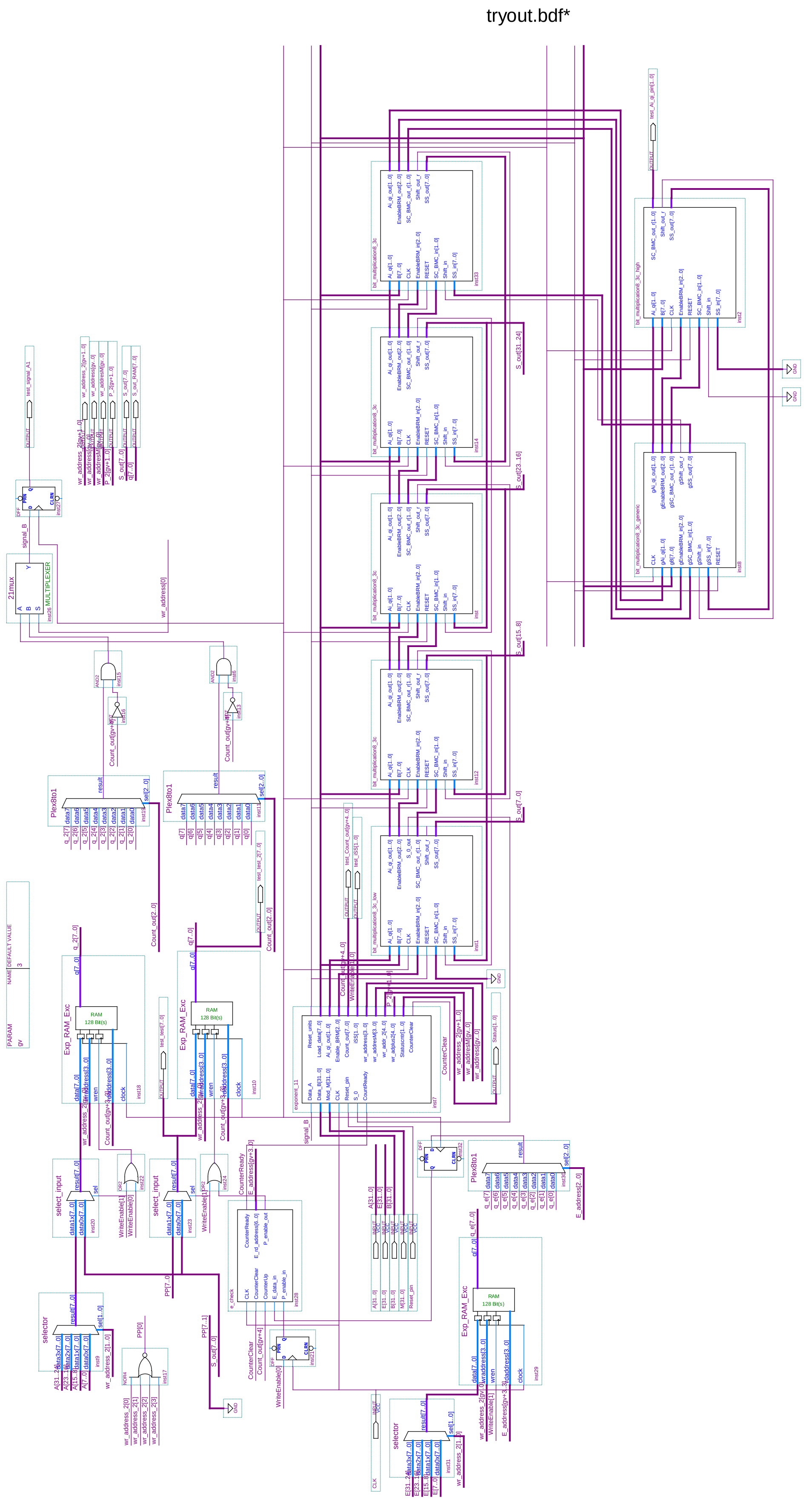

Figure B.2: 128 bit modular exponentiation array and control.

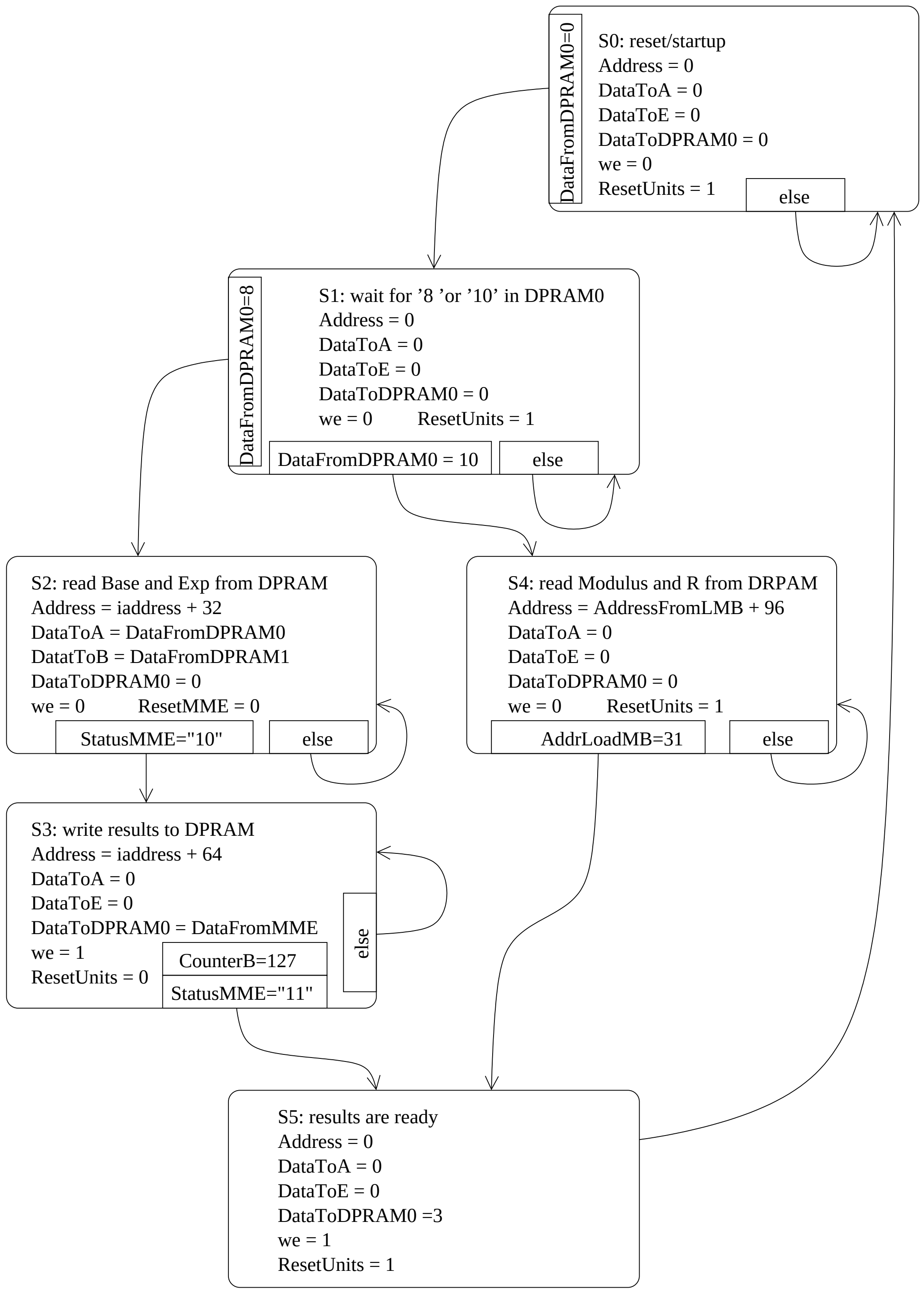

Figure B.3: Communication with ARM state diagram

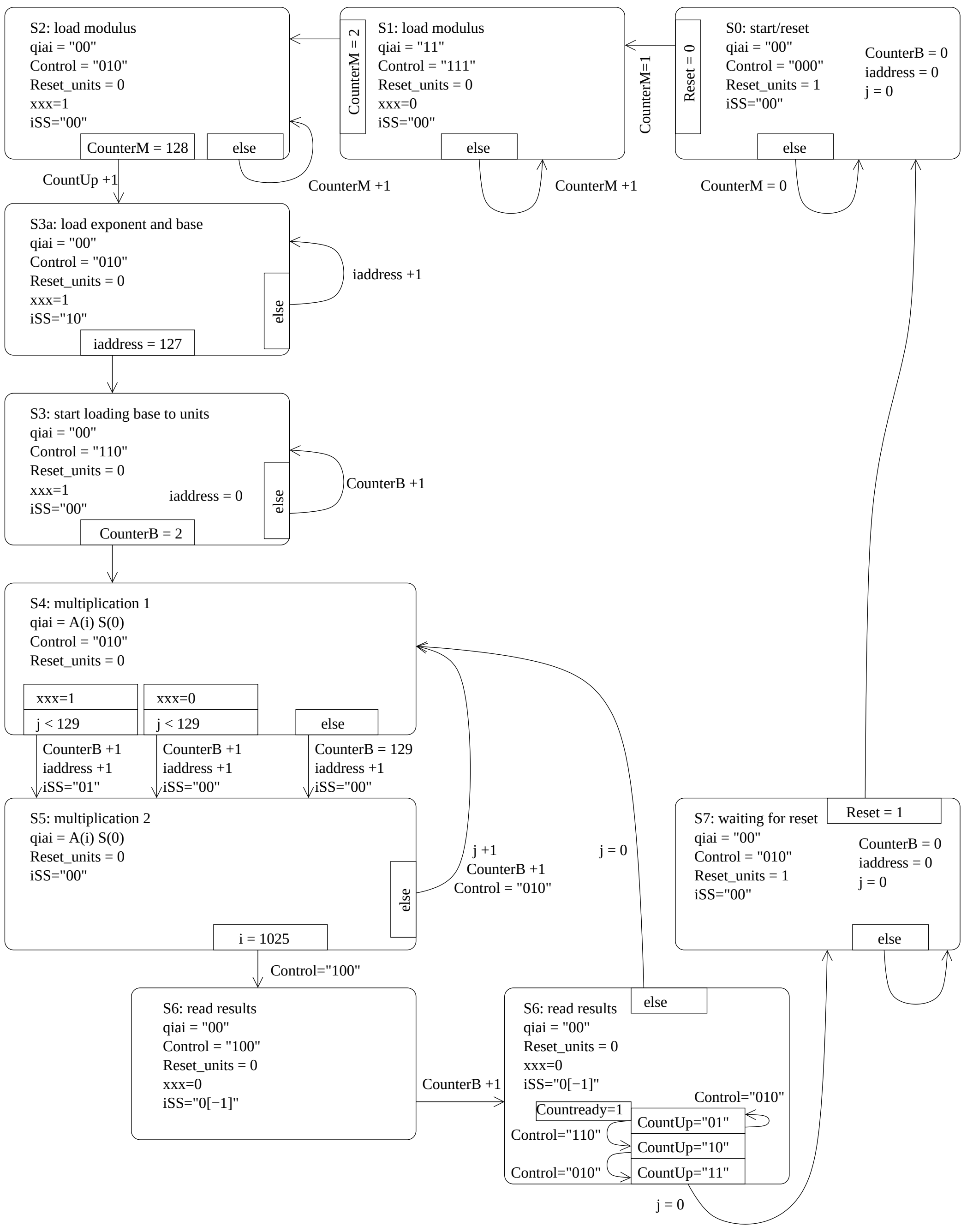

Figure B.4: Exp_Control state diagram

Curriculum Vitae

Peter Groen syntyi 1977 Leidschendamissa, Alankomaissa. Peter Groen on kasvanut lähellä Leidenin kaupunkia, Amsterdamin ja den Haagin välissä. Lapsuutensa ensimmäiset kymmenen vuotta Peter Groen asui Voorschotenissa, vuonna 1987 hän muutti perheensä kanssa Koudekerk a/d Rijn:iin. Vuonna 1996 Peter Groen siirtyi Delftiin ja aloitti opinnot Deltin Teknillisessä Yliopistossa. Opiskeltuaan vuoden konetekniikan osastolla hän siirtyi opiskelemaan sähkötekniikkaa ja valitsi pääopintosuunnakseen Computer Engineering. Opiskeluidensa ohella Peter Groen on työskennellyt Hollannin postilla sekä ollut mukana sähkökillan toiminnassa osallistuen ETV-lehti- ja symposium-toimintaan. Vuonna 2001 Peter Groen teki työharjoittelua Tokiossa, Japanissa. Hän auttoi Omronia Face ID Access Control system:ssä. Suomeen Peter Groen saapui helmikuussa 2002. Hän teki diplomityötään Tampereen Teknillisessä Yliopistossa toukokuuhun 2003 saakka, jolloin hän palasi takaisin Delftiin. Peter Groen työskenteli Tampereella Digital and Computer Systems:in laitoksella aiheenaan: hardware acceleration for an authentication protocol for WLAN. Hän suunnitteli tehtävät Alteran FPGA:ta varten. Peter Groen valmistuu Delftin Teknillisestä Yliopistosta syksyllä 2003 diplomi-insinööriksi.

Peter Groen was born in 1977 in Leidschendam, the Netherlands. He lived near the city of Leiden during the first 10 years of his life. Then his family moved to Koudekerk a/d Rijn, a village to the east of Leiden. In 1996 Peter Groen moved to Delft and started his studies at Delft University of Technology. He studied Mechanical Engineering for one year, then moved to Electrical Engineering and specialized in the field of Computer Engineering. During his years in Delft, Peter Groen worked part-time at the Dutch Royal Mail and took part in the organization of the periodical and symposium of the guild of electrical engineering students. In 2001, Peter Groen did a trainee-ship at Omron in Tokyo, Japan. He assisted in the development of their Face ID Access Control system. In February 2002, he moved to Finland to do his MSc thesis project at Tampere University of Technology. The project included the design of hardware accelerators in Altera FPGA for a WLAN authentication protocol. Peter Groen graduated from Delft University of Technology in autumn 2003.