DESIGN OF A FLEXIBLE CRYPTOGRAPHIC HARDWARE MODULE

ANDREW W.H. HOUSE

DESIGN OF A FLEXIBLE CRYPTOGRAPHIC HARDWARE MODULE

BY

© ANDREW W. H. HOUSE

A thesis submitted to the
School of Graduate Studies
in partial fulfillment of the
requirements for the degree of
Master of Engineering

FACULTY OF ENGINEERING AND APPLIED SCIENCE

MEMORIAL UNIVERSITY OF NEWFOUNDLAND

September 2004

St. John's Newfoundland

Your file Votre référence
ISBN: 0-494-02346-5
Our file Notre référence
ISBN: 0-494-02346-5

Abstract

The research presented in this thesis focuses on the design of a flexible cryptographic hardware module capable of implementing a variety of private-key cryptographic algorithms and their modes of operation using different implementation methodologies such as iteration and pipelining. The design of the SHERIF cryptographic hardware module was motivated by the difficulties inherent in implementing cryptographic algorithms: software implementation is easy and flexible, but offers low performance, whereas hardware implementations offer high performance but are difficult to design and are generally inflexible.

The design of the SHERIF architecture was driven by an analysis of several leading block ciphers and hash functions which identified six basic operations that could be used to implement most block ciphers and hash functions. Configurable components were developed to implement each of those operations, and these components were arranged into processing elements capable of implementing a single round of most of the algorithms under consideration. These processing elements were then integrated into a top-level system with complex data control mechanisms to provide added flexibility.

A sample pipelined implementation of the AES algorithm Rijndael has been successfully simulated. Synthesis results in 0.18 μm CMOS technology suggest the device would have an area of approximately 10 million gates and have a clock speed of 4.78 MHz, leading to a throughput of 611.84 Mbps for the sample implementation of Rijndael. These results demonstrate the flexibility and performance of the system.

The current SHERIF architecture offers greater flexibility and ease of use than existing cryptographic hardware modules, but there are still many areas in which it can be improved through future research. A number of avenues of future research have been identified that will improve system speed, integration, and flexibility.

Acknowledgments

This thesis owes its existence to the encouragement, support and inspiration of many people, and I would like to acknowledge them here.

I would like to thank my supervisor, Dr. Howard Heys, for his support, guidance, and boundless patience during my research over the past few years. Also, the funding he provided, along with Memorial University's School of Graduate Studies, was vital in allowing me to pursue this research.

I would also like to thank all of the instructors of the graduate courses I've taken during my Masters program, from whom I've learned a lot, some of which was even directly applicable to my research. Also, I would like to thank Nolan White of the Department of Computer Science for maintaining the CAD tools provided by the CMC and solving all my problems in short order.

Lastly, I would like to thank my fellow graduate student colleagues, both those in the CERL lab and without, for the support, advice, and friendship that made my graduate studies so enjoyable, and thanks to my friends and family for their support in school and life.

Contents

List of Tables

List of Figures

List of Abbreviations and Symbols

3DES	Triple DES
3GPP	3rd Generation Partnership Project
ABM	Automated Banking Machine
AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
CBC	Cipher-Block Chaining
CFB	Cipher Feedback
CLA	Carry Look-Ahead Adder
CLB	Configurable Logic Block
CLU	Control Logic Unit
CMOS	Complementary Metal-Oxide Semiconductor
CRA	Carry-Ripple Adder
CSA	Carry-Save Adder
DES	Data Encryption Standard
DPU	Data Path Unit

ECB Electronic Codebook

FPGA Field Programmable Gate Array

FSM Finite State Machine

Gbps gigabits per second

HDL Hardware Description Language

IPsec Internet Protocol Security

IV Initialization Vector

LFSR Linear Feedback Shift Register

LRU Least Recently Used

LSB Least Significant Bit

LUT Lookup Table

Mbps megabits per second

MUL Multiplier Unit

MSB Most Significant Bit

NESSIE New European Schemes for Signatures, Integrity, and Encryption

NIST United States National Institute of Standards and Technology

NSA United States National Security Agency

OFB Output Feedback

PE Processing Element

PE-FUNC Processing Element with Functional Units and Networked Connections

PIO Programmable I/O

PLA Programmable Logic Array

PROM Programmable Read-Only Memory

QoS Quality of Service

RISC Reduced Instruction Set Computer

RN Routing Node

SAGE Security Algorithms Experts Group

SLAB Sub-Layered Architectural Block

SoC System On Chip

SIMD Single Instruction Multiple Data

SSL Secure Sockets Layer

SHERIF Security Hardware Enhanced for Rapid Implementation and Flexibility

VLIW Very Long Instruction Word

VPN Virtual Private Network

Chapter 1

Introduction

Modern telecommunications technology forms the foundation of today's global society. Evidence supporting this bold statement can be seen in the rapid pace of cultural and technological development since the introduction of modern communication technology.

The first communication technology that might be considered *modern* was the telegraph. Once a telegraph network was in place, it allowed the transmission of text messages over long distances with minimal human effort. This was followed by telephones, to allow transmission of voice messages, and then radio, television, satellite communication, early computer networks, cellular phones, and the Internet. Each new technology opened new avenues of communication between disparate people and groups, helping build a global society and adding yet more complexity to human social interaction.

Today, the world is awash with communication technology. Cellular phones can now frequently connect to the Internet and provide e-mail and web browsing, and have long provided text-messaging and paging capabilities. E-mail and instant messaging have replaced the postal system for personal written communication. High speed Internet access is becoming more common. Overall, the various communication technologies are starting to converge, so that voice, video, text, image, and other data may be communicated over the same network.

Modern communication technology makes it easy and cheap to communicate across great distances, including national boundaries. The effect this has on society is profound, though not everyone is convinced that it is a good effect. Nonetheless, modern telecommunications has become an important part of many people's lives, integrated into their daily routine, and this introduces a host of new problems to deal with.

First, let us consider the kinds of communication networks people frequently deal with in the developed world.

1.1 Communication Networks

The most ubiquitous modern communication network is the telephone network. Initially designed to carry analog voice data, it has evolved into an elaborate computer-controlled network to manage hundreds of millions of calls around the world on a daily basis [1]. In the core of the network, at least, the voice data is converted from analog to digital to facilitate easier processing and transmission.

The telephone network is more an interconnection of different networks, run by different organizations. Regional companies build local phone networks to service the end user, and these smaller networks are connected into larger and larger systems that allow national and international telephone communication. As such, telephone networks are built on a set of international standards to ensure interoperability.

The typical visualization of the telephone network consists of the end-user, who has a telephone handset or end-system. The end-user's telephone is connected to a central office (frequently called an *exchange*). Other end-users, typically from the same geographic region, would be connected to this same central office. The central office can connect calls between its end-users, or to other nearby central offices for calls that are between more distant end-users. For long distance calls, the central office will connect to the long-haul network (or *backbone*), which interconnects different groups of central offices over long distances [1].

Cellular phone systems are integrated into the traditional telephone network as shown in Figure 1.1, but the end-system is quite different. End-users have wireless handsets which communicate with a *base station* responsible for providing service to a particular area of coverage (called a *cell*). The base station essentially has the role of a small central office, serving only cellular users in its area, and connects to a central office of the wired network. Base stations must not only handle regular telephone services, but also work together to track the location of users (to allow proper routing of incoming calls) and to transfer service between each other as users move into different areas of coverage [1].

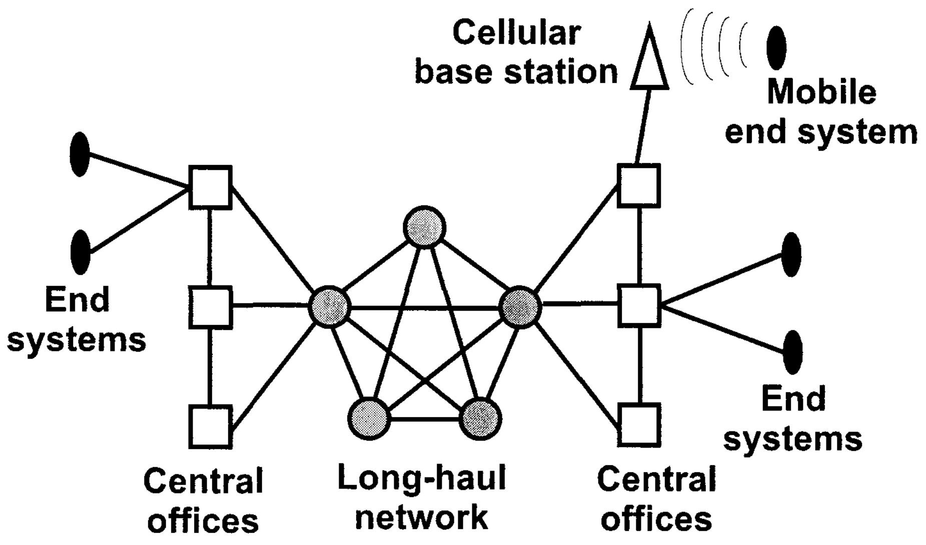


Figure 1.1: Telephone and Cellular Network

The telephone system is an example of a *circuit-switched* network. In essence, when a call is made, a dedicated path through the network is reserved for that call, and is guaranteed to be available for the duration of the call. Thus, the telephone network provides Quality of Service (QoS) with respect to voice communication [1]. Consequently, the telephone network is substantially different from the other major computer network that has become dominant in people's day-to-day lives – the Internet.

The Internet, illustrated in Figure 1.2, is a computer network primarily used for the transfer of data traffic. More accurately, it is a network of networks, hierarchically arranged into larger and larger systems [1]. Because of the flexibility of the end-systems – easily programmed computers – the data transmitted over the Internet can be for a variety of purposes: text messages on bulleting boards, web pages, e-mail, file transfer, video and audio streaming, and even real-time voice communication.

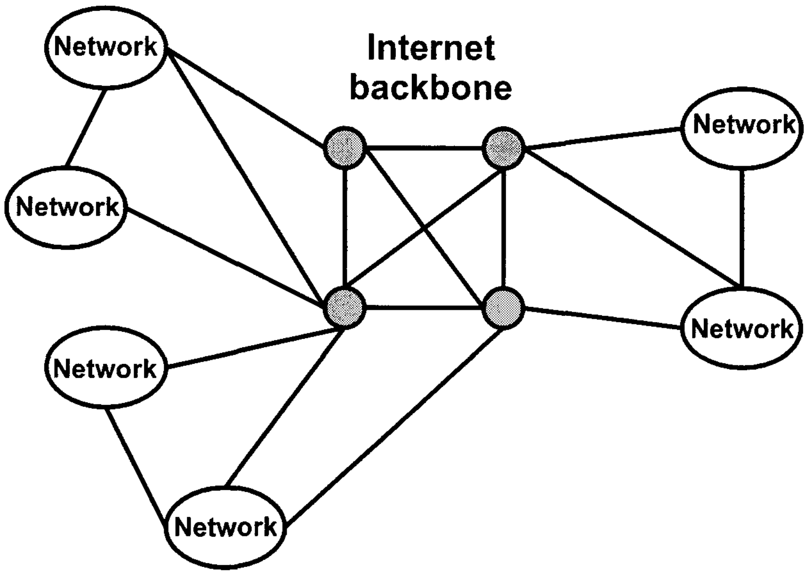


Figure 1.2: The Internet

The Internet is a *packet-switched* network – it breaks up the data to be transmitted into smaller chunks (called *packets*), and those chunks are transmitted individually [1]. Furthermore, the Internet is not centrally controlled. It has a distributed architecture, so routers in each member network simply know neighbouring network nodes and which neighbour is likely to be on the best path to a given destination. This data is constantly updated based on actual network conditions, and so packets traveling from one destination to another may not all follow the same route through the network [1].

The Internet does not guarantee QoS. Rather, it promises best effort. The store-and-forward routing approach means that delay through the network is unknown, and that packets can be lost if the network is subject to a lot of traffic [1]. However, even with this unreliability, the Internet architecture is useful for many applications, and consequently has seen widespread adoption [1].

Of course, other networks invade people's daily lives "below the radar", in such a way that they do not even realize they are using them. Automated Banking Machines (ABMs) communicate over a private network with the the bank's central servers, and via the Interac network with other financial institutions to provide banking services and debit purchasing at retailers. Modern credit card authorization systems work in much the same way. Satellite television, phone, and Internet are newer entries into the list of ubiquitous networks. New network architectures that seek to combine voice, video, and data (such as Asynchronous Transfer Mode (ATM) networks) are constantly in development, as well [1]. As a society, people have become used to the presence of communication devices and networks, and incorporate them into nearly every aspect of their daily lives. This has serious implications.

1.2 Security Concerns

People have always been concerned about the secrecy of their communications, though not in an obvious way. Whispered conversation and personal diaries have been the primary means of private thought. Governments and other organizations often have a more pressing need for secrecy – even Julius Caesar was said to use a basic cipher to protect his messages [2]. Military, diplomatic, and economic/financial information have all been candidates for secrecy at an organizational level, resulting in secret codes like Caesar's shift cipher [2] or the well-known rotor-based Enigma machine used by the German military in World War II [3]. The general populace, however, has had little need to be concerned with telecommunications security until recently.

The latter part of the Twentieth century has seen an explosion in the use of computing and telecommunications. From what was initially just the public telephone network (which was difficult to eavesdrop on due to physical access barriers and its central control), telecommunications has grown to encompass: computer databases full of personal information; banking done via automated machines, the phone, or Internet; cellular phones; e-commerce; interactive television services; and more besides. This is problematic in several ways.

The control of some of these newer forms of communication (such as the Internet) is decentralized by the very nature of their design. This has some advantages in terms of reliability, since a failure at a single point does not bring down the whole network, but it does mean that every computer through which communication passes might not be trustworthy. For most types of communication, this is not a concern, but when using the Internet for operations like banking, shopping, or even file transfer, it poses a security risk. Other technologies such as cellular phones and wireless Internet access have no physical access restrictions, since anyone with the appropriate receiving equipment can read the messages out of the air. Depending on the nature of the conversation or Internet use, this can pose a security risk.

Each of the different technologies has their own respective security risks. Those risks, however, are more important than ever before due to the way people are integrating them into their lives. Telecommunications technology has an effect on the day-to-day routine of many people; it has become a part of how people function in society. As such, it is relied upon more than ever before. The more it is used, the greater the need for security becomes. Cellular phones are now commonly used to conduct business deals and conversations, by law enforcement, and by individuals doing banking. The Internet is used for everything from personal e-mail and instant messaging to banking, tax submission, shopping, university registration, and business networking. When handling monetary transactions, personal information, and business secrets, security of communications becomes a major concern.

This begs the question, "What exactly *is* security?"

1.3 Types of Information Security

Information security is an extremely broad field that is difficult to define. It includes such things as virus protection, access control, user identification, privacy, control of data, and much more besides. In terms of communications, the definition is generally more specific: communication security should keep transmitted data private from everyone but the intended recipient, prevent the data from being altered in transit, and make sure that the sender and receiver are who they say they are. *Cryptography* is the main means by which this kind of security is provided [3].

Cryptography concerns itself with the study of several aspects of information security, namely privacy or confidentiality, data integrity, entity authentication, and data origin authentication [3]. Even though it is a subset of information security, cryptography is still a broad field. Cryptography has four primary goals [3].

1. *Confidentiality* involves keeping the content of information secret except to those authorized to view it [3].

2. *Data integrity* ensures that the information has not been altered by an unauthorized party, including insertion of additional information, deletion of existing information, or substitution of new information over the original [3].

3. *Authentication* relates to identification – it verifies that two parties involved in a communication are who they say they are (entity authentication) and that the source of the data is what it is supposed to be (data origin authentication) [3].

4. *Non-repudiation* essentially provides a guarantee that neither party can deny involvement in a communication (also called a transaction) that has already occurred. This frequently involves a trusted third party [3].

There are a number of tools (or cryptographic *primitives* [3]) that can be used to meet these goals, which can be broadly classified as unkeyed, private-key, or public-key primitives [3]. A *key* in this context is a value used to control access to protected information, just like a physical key controls access to locations or storage areas. In either case, only authorized parties – who are given access to a key – can fully access the secured information or location, and so much of the security of the scheme relies on protecting the key.

The four cryptographic goals described above are commonly met using a combination of cryptographic primitives. *Ciphers* are used to provide confidentiality, and *hash functions* and *digital signatures* can be used to provide data integrity, authentication, and non-repudiation services [3]. Modern cryptography has been designed to work with digital data, and the discussion here is in the digital context.

1.3.1 Block Ciphers, Stream Ciphers, and Public-Key Ciphers

Ciphers are mathematical algorithms that transform the original data (called a *plaintext*) into a different set of data (called a *ciphertext*) that has no obvious relation to the original plaintext. This process of transformation, called *encryption*, is performed at the originating end of a communication channel and uses a key as described above.

The ciphertext is then transmitted over the channel. At the receiving end, a key is used to *decrypt* the ciphertext and recover the original plaintext. If a good cipher is used, only the people with keys – that is, only authorized entities – can view the content of the information. Should any malicious party intercept the transmission, it would be useless to them since they would only intercept the ciphertext, and without the key they cannot recover the original message. This is illustrated in Figure 1.3, where E represents encryption and D represents decryption.

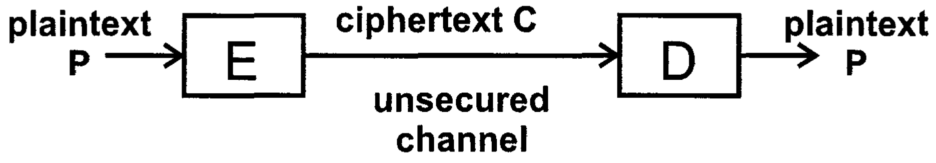


Figure 1.3: Encrypted Communication

There are two basic classes of ciphers: *public-key ciphers* and *symmetric-key ciphers.* Public-key ciphers have two keys, a public key and a private key. The public key is used to encrypt the plaintext into ciphertext, and the private key to decrypt and recover the original plaintext [3]. Thus, an entity will generate a public key and private key pair, and make the public key freely available. Anyone wishing to securely communicate with that entity simply encrypts the message with its public key, and then only that entity can use the corresponding private key to decrypt the message [3]. (Some public-key ciphers work in such a way that either key may be used to encrypt, and the other decrypt. This is more relevant to authentication than confidentiality). The security of public-key ciphers depends on the keys – it should be infeasible to determine the private key from knowledge of the public key [3]. Public key cryptography is illustrated in Figure 1.4.

Symmetric-key ciphers (also called private-key ciphers) use a single key for both

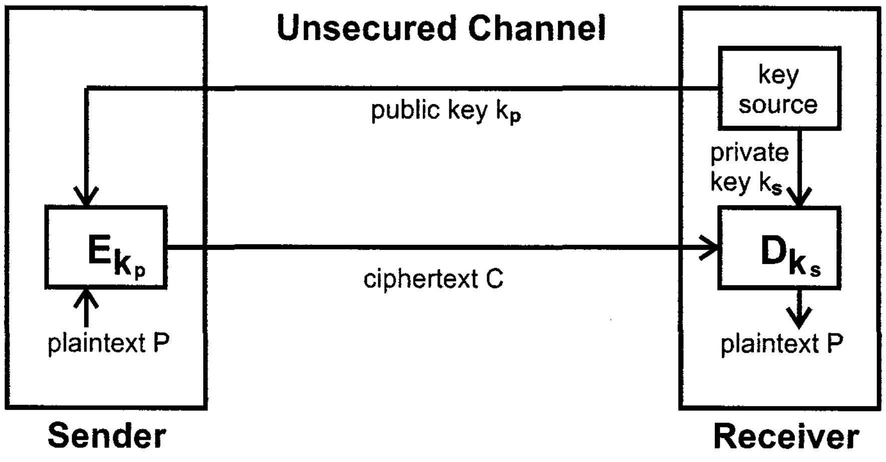


Figure 1.4: Public Key Cryptography

encryption and decryption. A message encrypted with a particular key must be decrypted with the same key. Thus, the key must be kept secret from all parties except the sender and receiver in order to maintain security [3]. This is illustrated in Figure 1.5.

Symmetric-key ciphers further break down into two sub-categories: *stream ciphers* and *block ciphers* (shown in Figure 1.6). They are basically similar, but stream ciphers operate on one plaintext symbol at a time, whereas block ciphers operate on groups of plaintext symbols, or blocks [3]. Stream ciphers typically consist of a mathematical function that generates a stream of bits that gets mixed with the plaintext stream, one bit at a time, via the exclusive-OR (XOR) operation [3]. Block ciphers transform blocks of plaintext (say, 64- or 128-bits at a time) into an output block of ciphertext of the same size [3].

If a single input bit to a stream cipher changes, only a single output bit will change. However, if a single input bit to a block cipher changes, many of the output bits should

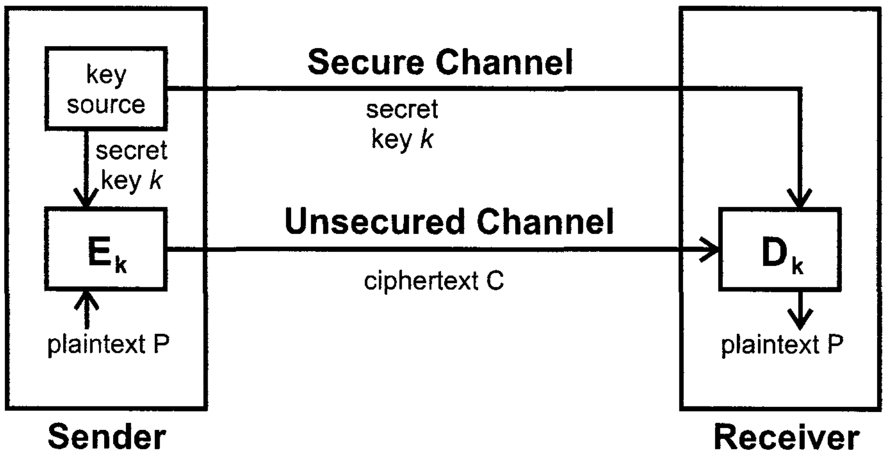


Figure 1.5: Private Key Cryptography

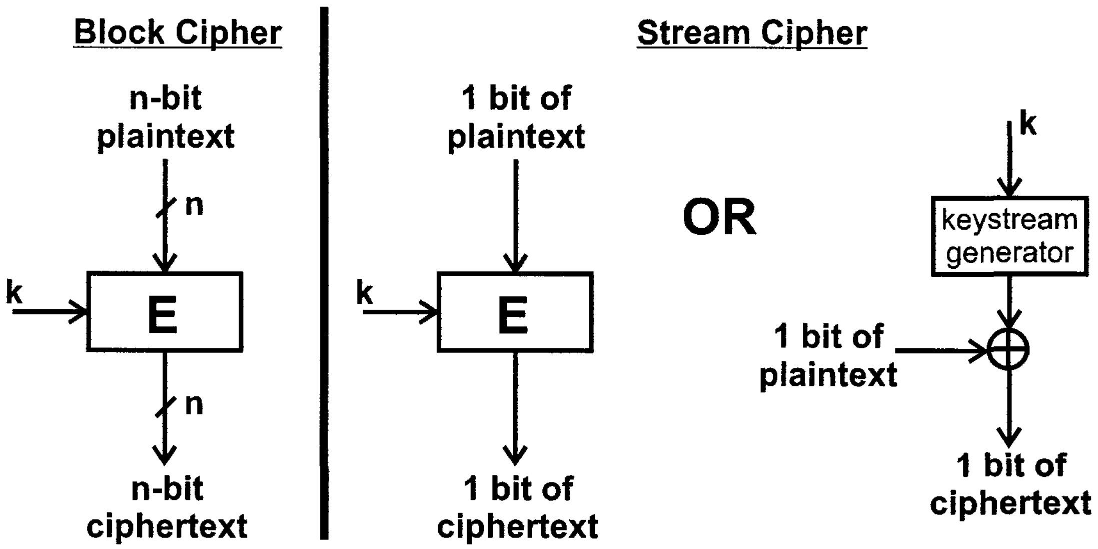


Figure 1.6: Comparison of Block Ciphers and Stream Ciphers

change [3]. These differing properties suggest that stream ciphers and block ciphers each have their own strengths and weaknesses, depending on their application. It should also be noted that block ciphers can be converted into stream ciphers quite easily [3]. Block ciphers will be discussed in more detail in Section 1.3.2.

1.3.2 Block Cipher Modes of Operation

An unfortunate characteristic of block ciphers is that identical blocks of data, encrypted with the same key, will produce identical ciphertext blocks [3]. Since data that repeats might be commonly encrypted (such as in communication protocol headers, for example), it might provide statistical information to attackers or reveal other information about the plaintext. Thus, several different modes of operation of block ciphers are used [3]. These modes are illustrated in Figure 1.7 for both encryption and decryption.

The normal mode of operation is Electronic Codebook (ECB) mode [3]. This is how most block ciphers are defined, and has the problem described above. The alternate modes of operation seek to avoid the disadvantages of ECB mode, and have their own properties related to error propagation and chaining dependencies that makes them appropriate in different situations.

Cipher-Block Chaining (CBC) mode alters the input plaintext to the cipher by XORing it with the ciphertext of the previous block encrypted by the cipher [3]. The first block of data to be encrypted is XORed with an Initialization Vector (IV). Thus, identical plaintext blocks will be modified to appear different via the XOR operation, and thus the ciphertext produced by the identical plaintexts will be different [3]. This mode is appropriate for streams of related data that are encrypted and decrypted together.

Cipher Feedback (CFB) mode essentially turns a block cipher into a stream cipher, using the cipher as a generating function for a random-seeming stream of bits [3]. It

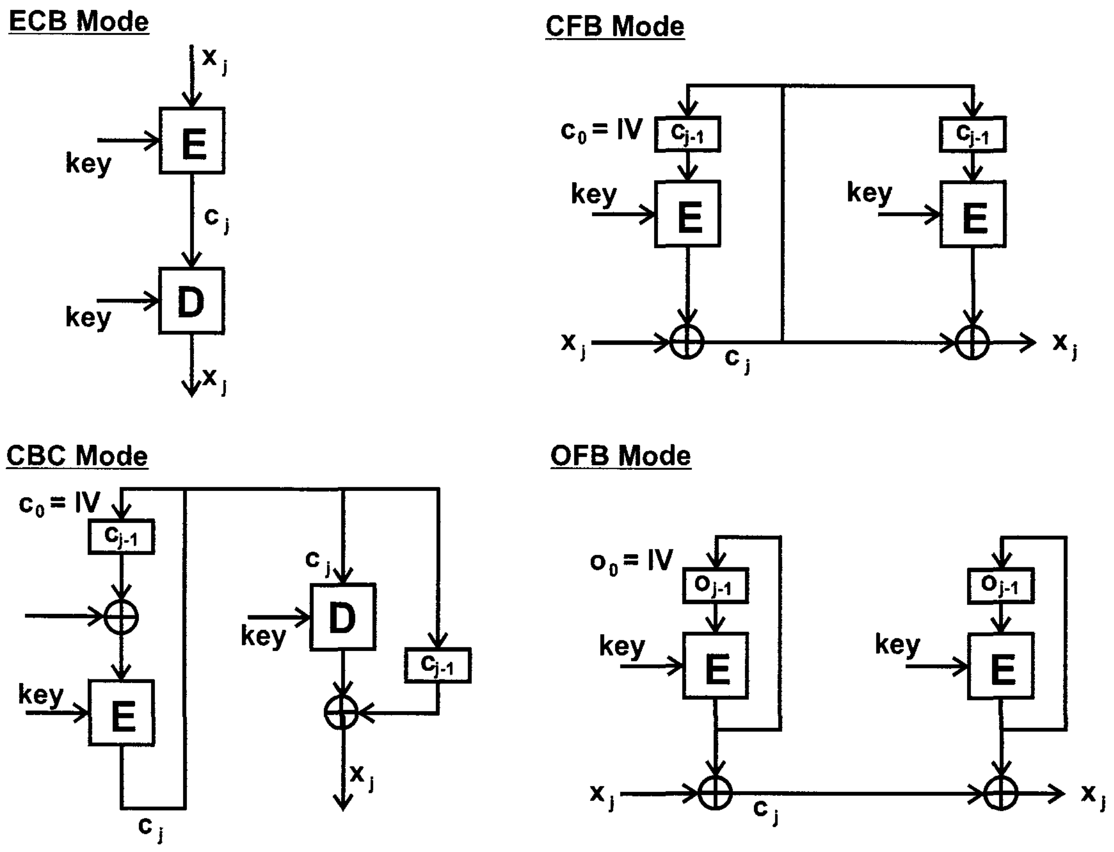


Figure 1.7: Block Cipher Modes of Operation

uses an IV as the input into the cipher, and the output (or some part of it) is XORed with the message plaintext to produce a ciphertext for transmission. The transmitted ciphertext (or some part thereof) is then used as the input to the cipher for the next block of data [3]. Output Feedback (OFB) mode is similar, but rather than using the ciphertext as the input to the cipher for the next block of data, it uses the block cipher output directly [3].

These modes of operation are important, and are specified in many common protocols. Thus, support of these block cipher modes of operation is crucial for any block cipher implementation.

1.3.3 Hash Functions and Signatures

Entity authentication (or identification) can be provided in several ways. Its goal is to verify the identify of parties involved in a transaction [3]. This service can be provided along with confidentiality, but also in its absence. Certain types of confidentiality provide entity authentication automatically – for example, in a communication which uses a symmetric-key cipher, entity authentication is implicit because only the sender and authorized receiver should have access to the secret key. Public-key cryptosystems provide a degree of entity authentication as well – only the intended recipient can recover the encrypted message, though the recipient cannot be sure of the identity of the sender [3]. However, in situations where entity authentication is desired in the absence of confidentiality, other primitives such as *hash functions* and *digital signatures* can be used.

Hash functions are efficient algorithms that convert a message of any length into a fixed-length value called a *hash-value* [3]. Typically, the length of the message is much greater than the length of the hash. They essentially create shortened representations of the entire message, although the message is not recoverable from the hash-value since the function maps from an infinite set of messages onto a finite set of hash-values.

Useful hash functions have properties such that it is computationally infeasible to find two messages that produce the same hash-value [3], and as a result, small changes in the input to the hash function leads to significant changes in the hash-value.

Hash functions can be keyed or unkeyed. Unkeyed hash functions will always produce the same output given the same input, whereas keyed hash functions mix the key information into the hash-value, so keyed hash functions will produce different hash-values when using different keys [3].

As stated above, hash functions can be used to provide authentication services, as well as data integrity services. The sender of a message calculates its hash-value, and appends this value to the message when it is sent. The receiver calculates the hash of the received message, and compares it to the transmitted hash-value to authenticate.

Unkeyed hash functions are thus primarily useful in providing data integrity with respect to channel errors rather than a malicious attacker, since an attacker could alter the message and replace the original hash-value with a new one matching the new message.

Keyed hash functions, on the other hand, provide data integrity (since any modification to the message will cause the receiver to calculate a different hash-value and thus reject the message) *and* both data origin authentication and entity authentication (since the sender and receiver must share a secret key used by the hash function, and thus the identity of the sender and origin of the data is known) [3]. The assertion of the shared secret key is the basis of the security of the authentication services provided by keyed hash functions.

Digital signatures are used similarly to hash functions, but do not require shared secret knowledge in order to authenticate. A digital signature is essentially a process that allows an entity to bind its identity to a piece of information [3]. The entity uses a signing function that transforms the message as well as some private information held by the entity into a tag or *signature* that is much like a hash-value [3]. The signing function might also include temporal or other supplementary information. The signature is transmitted along with the message. The receiver can verify the identity of the sender by running a publicly-known verification function on the signature, which will either verify or deny the identity of the sender. The verification will fail if either the message has been altered from what was originally signed, or if the signature has been altered [3]. In this way, it ensures the authenticity of the sender, the integrity of the data, and consequently the origin of the data.

The private information incorporated into the signature might be considered to be a type of private key, and the verification function might be considered to make use of public information about the entity, such as a public key. Digital signatures only provide one-way verification – they identify the signing entity and the origin of the data, but the sender knows nothing of the receiver. Nevertheless, digital signatures are another tool used to provide entity authentication, data origin authentication, and data integrity.

Non-repudiation services are provided implicitly as part of all of the preceding cryptographic tools. Confidentiality and authentication using shared secret keys ensure that all parties involved in a transaction are known, and that no malicious parties could impersonate them. Even in the absence of confidentiality, authentication and digital signatures provide data integrity and entity identification which disallows the possibility of denying a transaction.

Thus, it can be seen that the main cryptographic goals can be met with just a small set of cryptographic primitives. Unfortunately, theory often has difficulty being applied to the real world, and there are a host of issues which make implementing information security a truly daunting task.

1.4 Applications of Cryptography

Cryptography has long been relevant to military communications, to the extent that modern military communications must support a variety of cryptographic systems and thus programmable encryption systems are desired [4]. In recent years, the need for cryptography in business and private communication has become equally important as communication moves from "secure" channels such as letter mail or the wired telephone system, which are both operated by trusted agencies, to more accessible mediums such as the Internet or cellular telephony. Cryptography is needed not just to preserve privacy, but also to verify business transactions and protect financial information and trade secrets.

Cryptography is used in everything from digital cell phones and ABMs to Virtual Private Networks (VPNs) and Secure Sockets Layer (SSL) transactions over the Internet. These different systems all have different protocols, and often use different cryptographic algorithms to provide security. This introduces some interesting issues.

The primary issue with respect to cryptography is the distribution of the keys. Block ciphers require the sender and receiver to have the same secret key. Public-key cryptography, which is much slower than private-key, can be used to exchange secret keys, and then private-key ciphers can be used for the bulk of communication. There are a number of key exchange protocols designed for this [3, 2, 5], and to allow interoperability, a number of standard security protocols have been defined. For example, secure Internet transactions typically use SSL, whereas VPN applications (which use the Internet) generally use the IPsec protocol [5].

Unfortunately, cryptographic systems are difficult to implement properly [5]. Software implementations of cryptography are easy, and offer flexibility, but are often open to attack from other malicious software. Additionally, software implementations are too slow for many applications, since they only offer throughput on the order of tens to low hundreds of megabits per second (Mbps) [6], which is insufficient for high-bandwidth communications applications.

Hardware implementations of cryptography offer much greater speed, ranging from the mid hundreds of Mbps to several gigabits per second (Gbps) [7], as well as offering a degree of physical security that software lacks. Such implementations, whether in the form of a cryptographic Application Specific Integrated Circuit (ASIC) or an algorithm implemented in an Field Programmable Gate Array (FPGA), are difficult and time-consuming to develop, requiring specialized hardware design skills. ASIC implementations tend to lack flexibility as well, in that they generally cannot be changed once manufactured, whereas with some degree of effort, FPGAs can be.

In today's communication landscape, with new applications being developed all the time that must exist alongside legacy applications and different systems and protocols, the need for a flexible cryptographic solutions is obvious. Flexible systems capable of dealing with different algorithms and protocols are highly desirable, and such a hardware system would have a wide range of applications.

1.5 Motivation and Proposed Solution

As can be seen, common methods of implementing cryptographic algorithms each have their respective good and bad points. Software implementations are easy and flexible, but result in low throughput. Hardware implementations, such as in ASICs or FPGAs, provide high throughput, but require a lengthy and difficult design process.

The purpose of this research is to develop a middle-ground, something that combines the best features of both common forms of implementation: the ease of software implementation and the speed of hardware. Unfortunately, to achieve this, compromises will have to be made.

The rationale for such a project is two-fold. Firstly, such a system will lighten the workload of developers, allowing them to integrate relatively fast security hardware into their products more easily than doing their own FPGA designs, and with more flexibility than an pre-packaged ASIC implementation, thus allowing changes and fixes after production has begun. This will save time and money. Secondly, use of such a system benefits the end-users since if a weakness is found in the currently-implemented security algorithm, the system could be reconfigured more easily that implementing a new FPGA implementation. This combination of the ease of use and flexibility of software with the speed of hardware might in many instances be a desirable combination.

Thus, the goal of this research is to design a cryptographic hardware module that can be configured for a variety of different algorithms, which provides greater throughput than software implementation, and requires less design time than standard hardware implementation. It should also support standard block cipher modes of operation, and different algorithm implementation methodologies such as pipelining and iteration. The cryptographic hardware module, entitled Security Hardware Enhanced for Rapid Implementation and Flexibility (SHERIF), is intended for use in high-throughput communications systems, and thus must be able to implement several different algorithms and provide a significant degree of throughput to be considered a successful prototype.

The discussion of this research is presented in the subsequent chapters, as outlined below.

- Chapter 2 provides a concise but thorough review of existing cryptographic implementations, with particular attention paid to reconfigurable hardware architectures and special-purpose cryptographic modules.
- Chapter 3 introduces and analyzes the six major cryptographic algorithms around which the subsequent design was based.

- Chapter 4 details the design and implementation of the basic operational components and data control components that were used to build our higher-level system.
- Chapter 5 describes the overall system architecture of our cryptographic hardware module, showing how it is constructed from the components detailed in Chapter 4 and demonstrating the reasoning behind the architecture and control scheme.
- Chapter 6 describes the design and use of the software configuration utility written to aid the end-user in configuring the cryptographic hardware module to implement specific algorithms.
- Chapter 7 details the functional testing of the device, focusing on the implementation and validation of a pipelined version of the AES cryptographic algorithm, and also discusses synthesis results.
- Chapter 8 investigates possible directions for further research related to the cryptographic hardware module, and draws conclusions about the existing design.

The appendices contain samples the VHDL code for the SHERIF cryptographic hardware module, Java code for the software configuration utility, and overall configuration details.

Chapter 2

Current Cryptographic Hardware

Before designing a new cryptographic hardware module, it is necessary to consider existing implementations and platforms, from pure software to dedicated ASICs, FPGAs, and special-purpose cryptographic hardware modules. Knowledge of such implementations guided design efforts for the SHERIF cryptographic hardware module under development.

2.1 Standard Algorithm Implementations

Note that the following analysis is focused on performance as measured by throughput, where throughput is defined as the number of bits per second (bps) that the algorithm implementation can process. Sometimes throughput is also reported in bytes per second. Other considerations such as code size (for software), power consumption, area (for hardware), and latency are not considered.

2.1.1 Software Implementations

The easiest means of implementing cryptographic algorithms is in software. Software implementations are flexible, and general-purpose microprocessors can run software to implement any cryptographic algorithm. The key drawback to software implementation is that it is comparatively slow, meaning it is too slow for certain applications. For example, a 600 MHz processor is incapable of encrypting data fast enough to saturate a T3 communication line using Triple DES (3DES) encryption [8]. Furthermore, speed of operations is clearly dependent on the speed of the processor, and certain operations such as permutations are very slow to implement in software.

Still, software implementation is viable and cost-effective in many applications, and thus is well-studied. In particular, during the AES selection process sponsored by United States National Institute of Standards and Technology (NIST) [9] and the NESSIE standardization process [10], software implementations of the candidate algorithms were extensively developed and tested. Table 2.1 summarizes software speeds for several interesting algorithms from the two standardization efforts. Note that the AES candidate numbers were converted from average cycle counts per block when encrypting 128 blocks [6] to get the throughput.

Algorithm	**Platform**	**Throughput**
Rijndael	ANSI C (450 MHz Pentium II)	98.3 Mbps
RC6	ANSI C (450 MHz Pentium II)	122 Mbps
Serpent	ANSI C (450 MHz Pentium II)	21.1 Mbps
Twofish	ANSI C (450 MHz Pentium II)	101.6 Mbps
MARS	ANSI C (450 MHz Pentium II)	72.18 Mbps
Camellia	assembly (700 MHz Pentium III)	290.9 Mbps
Camellia	Java (1 GHz Pentium III)	161.4 Mbps

Table 2.1: Software Implementation Results from AES Development and NESSIE

More recent software implementations have in many cases improved upon these performance figures, due to increasing microprocessor speeds and better understanding of the algorithms themselves. For example, recent software implementation speeds of the AES are listed in [11]. Highly optimized software on very high end microprocessors (such as Intel's Pentium 4 running at 3.2 GHz) can achieve impressive speeds, as high as 1537.9 Mbps, and other high end microprocessors range in performance from the mid-to-high hundreds of Mbps. The key downside to such software implementations is that they achieve such performance by running on expensive and power-hungry high end microprocessors, and thus even if such a software implementation on a high end processor could satisfy the requirements of a communication system, it might not be cost-effective to do so, especially in systems where space or power consumption are important.

Despite the ever-increasing speed capabilities of software implementation, dedicated hardware will still be needed for the foreseeable future to support new applications requiring higher and higher data rates, such as high definition real-time video. Perhaps more importantly, while software implementations may offer sufficient speed for end-users in whatever applications they may need, such capabilities do not scale to large networks and the vast amounts of data that they must transport. Thus, the need for hardware implementations to offer greater speed than software is apparent. These software numbers are provided primarily as a point of comparison for hardware implementations.

2.1.2 Dedicated Hardware Implementations

Hardware implementations of cryptographic algorithms are widely varied, and encompass both ASIC and FPGA technology. Both types of hardware typically provide much greater speeds than capable in software, but share (to an extent) the same lengthy and complex development cycles.

There are other advantages to hardware implementation as well. In [12] and [13] the need for encryption algorithm agility in ATM networks is discussed. Algorithm agility means that a system must be able to handle multiple algorithms, since there are several defined in the ATM security standard. While software running on a microprocessor inherently offers such flexibility, it would likely be incapable of supporting the high data rates needed for an ATM switch. The architecture presented in [12] proposes a number of encryption hardware pipelines in parallel to support the different algorithms in the specification. The various pipelines are fed by an input sorting queue. This approach avoids software or reconfigurable logic overhead, and achieves aggregate speeds of up to 21.2 Gbps when running at 100 MHz [12], far beyond the performance available at that clock speed for a software implementation.

ASICs such as the above tend to provide the greatest performance, but once manufactured, they cannot be fixed, altered, or updated. They are totally inflexible, and have a longer and more costly development process than FPGAs. Only when manufactured in large quantities do ASICs become cost-effective [14].

FPGAs typically provide lower performance than ASICs, but have the advantage of shorter and less costly development cycles since less work is needed before the design can be mapped into a standard FPGA part. Also, since FPGAs are programmable, it is possible for FPGA implementations of cryptographic algorithms to be updated or replaced if the final system is designed to accommodate this, which offers much greater flexibility than ASICs. This flexibility also means that, as hardware technologies improve, it is easier to re-implement an algorithm in a newer FPGA than it is to produce a new ASIC.

Both ASICs and FPGAs have their benefits and drawbacks. Cryptographic algorithm implementations in both have been studied extensively. The five finalist algorithms for the AES were implemented in hardware by the National Security Agency in the United States [7] in both iterative and pipelined implementations. Similarly, New European Schemes for Signatures, Integrity, and Encryption (NESSIE) candidates such as Camellia [15] included extensive hardware performance results in their submission. Table 2.2 shows several hardware performance values for a number of algorithms that were AES and NESSIE candidates. Most of the devices are implemented in various sizes of Complementary Metal-Oxide Semiconductor (CMOS) technology.

Algorithm	**Implementation Details**	**Throughput**
Rijndael	pipelined (0.5μ CMOS)	5163 Mbps
Rijndael	iterative (0.5μ CMOS)	443.2 Mbps
RC6	pipelined (0.5μ CMOS)	2171 Mbps
RC6	iterative (0.5μ CMOS)	103.8 Mbps
Serpent	pipelined (0.5μ CMOS)	8030 Mbps
Serpent	iterative (0.5μ CMOS)	202.3 Mbps
Twofish	pipelined (0.5μ CMOS)	2278 Mbps
Twofish	iterative (0.5μ CMOS)	104.6 Mbps
MARS	pipelined (0.5μ CMOS)	2189 Mbps
MARS	iterative (0.5μ CMOS)	56.7 Mbps
Camellia	unrolled (0.18μ Mitsubishi)	3200 Mbps
Camellia	iterative (0.18μ Mitsubishi)	1881.25 Mbps
Camellia	pipelined (Xilinx VirtexE FPGA)	6749.99 Mbps

Table 2.2: Hardware Implementation Results from AES Development and NESSIE

A unified hardware implementation of Camellia [16] and AES [17] is presented in [18]. The unified implementation used structural similarities between the two algorithms to reduce hardware where possible, and algorithm-specific hardware where necessary. It provided throughput for AES of 469.22 Mbps and 794.05 Mbps when optimized for area and speed respectively, and 661.18 Mbps and 1118.89 Mbps for Camellia [18]. In comparisons to separate implementations cited in that paper, the unified architecture was slightly slower than independent implementations of the algorithms, but had a lower overall area. The independent implementation of AES was listed as having a throughput of 548.68 Mbps when optimized for area, and 875.28 Mbps when optimized for speed, whereas the independent implementation of Camellia had throughput of 1094.04 Mbps when optimized for area, and 1616.14 Mbps when optimized for speed [18]. All implementations were in 0.13 μm CMOS technology.

The results presented here are in no means considered to be comprehensive. Rather, they are given to provide a general context for the performance of the system under design.

2.2 Commercial Cryptographic Co-Processors

There are a number of commercial products available that generally refer to themselves as cryptographic co-processors. They generally focus on increasing the speed of public-key cryptography, whether it is RSA encryption or digital signatures. Although architectural details are scarce on many of the products, they mostly seem to be system-level products (rather than single-chip solutions) that are used to off-load the processing from the main processor in the system. Most of them seem to be based around a general microprocessor with special hardware to implement the cryptographic algorithms. Also, all of the commercial products are very application-specific – most are designed to accelerate the IPsec or SSL security protocols.

There are three general types of security chips or devices [19].

1. Coprocessors exist as part of a larger system, and off-load the cryptographic duties from the main processor, freeing it for other tasks. This sort of arrangement is limited by the communication between the main processor and coprocessor [19].

2. Inline security processors are used directly in the communications path, and thus are forced to take on additional processing tasks beyond security. However, they allow a significant improvement in speed and throughput [19].

3. Network processors with integrated/embedded security functions are the third class of devices and can basically be seen as a combination of coprocessor and inline processor, or a consolidation of an inline security processor with the other communication components. Such integrated processors bring all the network and security functionality into a single device to allow line-rate operation [19].

These different approaches can be seen in the many varied commercial devices currently on the market.

2.2.1 Motorola's S1 Family

Motorola [20] has a family of security processors, the S1 family, consisting of the MPC180, MPC184, MPC185, and MPC190 [21]. The processors are targeted toward edge routers, wireless base stations, e-commerce servers, broadband access equipment, and more [22], and are designed to support standard internet security protocols such as IPsec, IKE, WTLS/WAP and SSL/TLS [23]. The MPC185 also supports 3rd Generation Partnership Project (3GPP) protocols [24].

To support these protocols, the processors must support algorithms such as public-key cryptography (RSA, Diffie-Hellman, and elliptic curve), various modes of DES (and AES in later processors), and message digests (including SHA-1, MD4, and MD5) [25]. While the specific features of each processor differ, their general architecture is the same.

Rather than provide a general architecture optimized for cryptography, the S1 family processors incorporated dedicated hardware for the supported algorithms onto a single chip. The architecture of the MPC185 provides a good example of this [24]. It has 10 execution units plus a random number generator. There are two public key execution units, which offer programmable support of RSA and Diffie-Hellman as well as elliptic curve cryptography [2], and several modes of each. Likewise, there are two DES execution units offering DES and 3DES in a variety of modes and configurations. Two AES execution units provide support for various modes of Rijndael, and the two message digest execution units support various modes of SHA-1 as well as MD4 and MD5. It even provides stream cipher support in the form of a single ARC4 execution unit (compatible with the RC4 algorithm), and support for the 3GPP algorithm Kasumi (for both encryption and authentication) via a dedicated execution unit.

The estimated performance numbers for the MPC185-supported block ciphers, as shown in [24], are included in Table 2.3. Note that overall performance improves as the amount of data being processed increases. This is due to the memory access required by the processors operating within a larger system.

Data Size	DES-CBC	3DES-CBC	AES	MD5	SHA-1
64 byte	204	168	180	177	162
128 byte	355	260	281	311	279
256 byte	562	358	391	472	411
512 byte	815	449	489	636	540
1024 byte	1051	513	557	770	639
1536 byte	1164	538	585	828	681

Table 2.3: Block Cipher and Hash Function Estimated Throughput (in Mbps) from Motorola MPC185 Security Processor Technical Summary

2.2.2 The IBM 4758 Secure Coprocessor

IBM [26] produces the 4758 secure coprocessor, described in [27, 28]. The 4758 has a two-fold purpose: to accelerate cryptographic functions and always operate correctly despite physical attack [28]. Thus, much of the design effort for the 4758 went into the physical tamper-resistance of the device, rather than an innovative cryptographic architecture.

The 4758 secure coprocessor incorporates a Data Encryption Standard (DES) acceleration unit (the device predates AES selection), a modular math unit, and a general-purpose 486 microprocessor to provide cryptographic support [28]. Later revisions of the device offer SHA-1 and Triple-DES execution units as well [29].

Given that the focus of the device is on providing a secure operating environment rather than high cryptographic performance, it might be expected that the performance will not be optimal. In [30], the latest models are described as supporting up to 175 1024-bit key RSA operations per second, which is quite good, but the DES encryption throughput is 15.3 MBytes/section (approximately 122.4 Mbps), which is slow compared to many other DES implementations [31]. The performance may be partly limited by the PCI interface of the device, which has limited bandwidth, but the focus of the device on protecting the secret/private keys, authenticating operations, and providing a secure execution environment (not just for cryptography [28, 32]) likely contributes to its comparatively lackluster performance.

2.2.3 Harris Corporation Sierra

The RF Communications Division of the Harris Corporation has produced a programmable cryptographic device to meet military and law enforcement needs as described in [4]. The Sierra device is aimed at providing Type 1, Type 3, and Type 4 encryption for wireless devices [33], and thus is designed to maximize flexibility while providing sufficient performance.

Wireless applications typically have lower bandwidth requirements than wired networks, though modern wireless systems can have throughput of several megabits per second [4]. The Sierra module provides both flexibility and sufficient speed through use of a dedicated ASIC called Palisades [33] which incorporates a variety of functionality (including cryptography).

The Palisades ASIC is based around an ARM7T Reduced Instruction Set Computer (RISC) processor core, which provides most of the device's flexibility and functionality. However, the processor is supplemented by several cryptographic blocks to accelerate cryptographic operations: two Type 1 encryption hardware blocks (for military encryption support), a DES encryptor, a 128-bit multiplier, a randomizer function, and even Linear Feedback Shift Registers (LFSRs) for stream cipher support. These blocks are tightly coupled to the processor core, allowing great flexibility, performance, and support for many different modes and algorithms [33].

This sort of architecture is fairly common in terms of programmable cryptographic hardware; most programmable solutions rely on general microprocessors with extensions to speed cryptographic operations. Several further examples of this will be seen in subsequent sections.

2.2.4 Cavium Networks

Cavium Networks [34] offers a series of security processors supporting the IPsec and SSL protocols, as described in [35]. This device runs at 500 MHz and has high-bandwidth I/O which contribute to its ability to achieve very high peak throughput, as shown in Table 2.4 [35].

Operation	Peak Bandwidth
3DES	70 Gb/s
AES (with 256-bit key)	61 Gb/s
ARC4	28 Gb/s
MD5	56 Gb/s
SHA-1	89 Gb/s

Table 2.4: Peak bandwidth of Cavium IPsec/SSL processor at 500 MHz

These very high throughput numbers are primarily due to the somewhat unique architecture of the device. It consists of 28 execution units, each of which is programmable [35]. Each execution unit consists of a 16-bit microcode processor engine with 12 kB of code storage, which controls the 64-bit Arithmetic Logic Unit (ALU) and the cryptographic units and provides high level protocol processing. Hardware support is provided for 3DES, AES, and ARC4, MD5, and SHA-1. A hardware modular multiplier is also included to support public-key operations via the microcode processor [35]. The large number of execution units, and the large bus connecting them all, combined with fast chip I/O, leads to some very impressive performance figures. However, the microcode processors are limited, and thus, while this device provides excellent support for IPsec and SSL algorithms, it is not easily expandable to new applications.

2.2.5 Other Commercial Solutions

There are many other cryptographic hardware products commercially available, a number of which are discussed in [19] and [36].

Many commercial cryptographic accelerators focus on accelerating public-key operations, since they are more computationally intensive than the block ciphers which are used for bulk encryption. As such, they are often packaged as add-in boards for larger systems. Such cryptographic accelerators from nCipher and Rainbow Technologies are discussed in [36]. The nCipher product is an add-in or a larger computer system, whereas the Rainbow product is attached to the same network as other systems, allowing different forms of acceleration. Both are targeted at improving the speed of SSL transactions, primarily in terms of the public-key setup, although that includes some block cipher components as well [36].

Broadcom's CryptoNetX line of security processor devices work with host processors, although the company provides them on fully realized add-in boards as well. Like most commercial security hardware vendors, their chips such as the BCM5840/5841, BCM5850, and BCM5851 support the standard protocols such as IPsec and SSL [19]. The BCM5840/5841 devices are meant to support IPsec for VPNs, and thus they support DES, 3DES, AES, MD5, and SHA-1 in hardware, allowing 2.4 Gbps and 4.8 Gbps data rates, respectively [19]. The BCM5850 and BCM5821 support the protocols and public key cryptography needed for SSL, with the BCM5821 accelerating the RSA public-key cryptography (allowing 4000 1024-bit transactions per second) as well as providing hardware support for DES, 3DES, ARC4, MD5, and SHA-1 [19].

Corrent has a series of security chips and boards as well, including the CR7020 and CR7120. These devices accelerate SSL or IPsec operations using special hardware for DES, 3DES, AES, ARC4, MD5, and SHA-1, as well as having on-chip exponentiator circuits for public-key algorithms. The CR7020 runs at 1.5 Gbps, while the CR7120 runs at 3 Gbps [19].

HiFN has a variety of cryptographic devices available. Their coprocessors range from the 7855, which supports IPsec protocols and algorithms at rates up to 650 Mbps, up to the 7955 processor, which supports a greater degree of public-key cryptography as well as supporting the IPsec algorithms at speeds of up to 756 Mbps [19]. They also have several in-line security processors, the 8300 and 8350 FlowThrough processors, which are meant to be used directly in a communication system datapath, rather than as a coprocessor. They support IPsec applications with public-key operations and standard private-key encryption and authentication, as well as an IKE stack. The 8300 runs at 600 Mbps, while the 8350 operates up to 4 Gbps [19].

Layer N Networks is a relatively new company that has developed a fast SSL chip, using a simplified mathematical algorithms to speed up public-key cryptography. Their UltraLock chip is an in-line processor, and thus incorporates protocol processing hardware for TCP/IP along with the security hardware [19]. It has a control unit to handle the SSL and TLS protocols (using RSA encryption), and a separate cryptography controller which interfaces to a hardware execution unit supporting SHA-1, MD5, DES, 3DES, RC4, and AES. This architecture allows it to implement security at the a rate of 1 Gbps [19].

Intel's IPX2850 network processor is primarily intended to support network processing, but incorporates security features as well, such as 3DES, AES, and SHA-1, and allows speeds of up to 10 Gbps [19]. Software support allows the device to implement either IPsec or SSL/TLS, and makes this processor somewhat more flexible in terms of the applications it supports than most of the other devices considered here, but the cryptographic algorithms themselves are fixed [19].

Overall, the commercial solutions overwhelmingly tend to offer flexibility through incorporation of a general processor supplemented by additional cryptographic hardware for specific algorithms. Thus, while excellent performance may be achieved in current applications, expandability to new algorithms, protocols, or applications is limited.

2.3 Implementation on Reconfigurable Platforms

While dedicated hardware implementations provide high speed, they lack the flexibility of software implementations. Straightforward FPGA-based hardware implementations offer almost as much speed as ASIC implementations, but with the added benefit of being reprogrammable, and yet the development effort is similar in many respects. Also, once operating, an FPGA tends to be static, although newer FPGAs allow dynamic reconfiguring of themselves (or parts thereof). A number of cryptographic hardware systems have been proposed to take advantage of this.

In [37], a reconfigurable computing system based on an FPGA is described, and implementation of the DES algorithm is used as an example of its operation. While their configuration management scheme (in which FPGA configurations are multiplexed from ROMs into the FPGA) provides good support for an algorithm-agile cryptography, it is still just a general reconfigurable system that could be used for virtually any application.

The PipeRench architecture described in [38] is also a general reconfigurable system used to implement cryptographic algorithms. It supports the concept of virtual hardware, in which parts of its configuration may be swapped for others to effectively support a design larger than the resources of the device. This degree of flexibility makes implementation of a variety of cryptographic algorithms feasible, and allows them to share the same execution platform.

An extension to the PipeRench architecture optimized for cryptographic operations, called PipeRench+, is also described in [38]. The PipeRench+ system is identical to the baseline PipeRench architecture except for the addition of a small memory accessible from each of the computational *stripes*, in order to support larger table lookups than the standard PipeRench. However, PipeRench is not fully configurable; some algorithms cannot be mapped onto the device, although they can still be accelerated by it if custom instructions are implemented in a co-processing situation [38].

The CryptoBooster is a cryptographic coprocessor described in [39]. It is optimized for implementation with FPGAs, and meant to work within a host system. It incorporates a number of session control features to handle the details of each task it is asked to perform. However, the core functionality lies in the hardware implementation of the ciphers in the FPGA part of the system. CryptoBooster supports the partial reconfiguration capabilities of the FPGA, and allows the session controller and adapter to configure the FPGA as required by the session setup. Though designed to support cryptographic operations, it is still based around general reconfigurable hardware.

Another FPGA-based algorithm-agile cryptographic coprocessor is presented in [40]. The FPGA in this system houses the cryptographic processor, which is configured with data from a non-volatile memory block called the algorithm library. Thus, a number of algorithm configurations can be stored in the algorithm library, and depending on how the system is controlled, different algorithm configurations can be loaded into the FPGA. This provides a flexible platform and allows algorithm agility. However, there is still the difficulty in designing such FPGA configurations.

Overall, the use of FPGA-based reconfigurable computing may be an effective solution to the problem of flexibility and good performance. However, they are typically board-level solutions, rather than single-chip, and apart from providing an FPGA platform, are not optimized for cryptographic processing.

2.4 Generalized Cryptographic Hardware

While much study has been performed on implementation of individual cryptographic operations, such as ROMs for use as s-boxes [41], there has also been some academic effort toward the development of generalized cryptographic hardware capable of implementing a variety of algorithms, apart from those implementing cryptography in general reconfigurable systems. Many of these architectures propose extensions of general microprocessor architectures, adding instructions that accelerate cryptographic operations in order to speed up software implementations of algorithms using optimized hardware.

An example of this in its most basic form is shown in [42], which proposes the addition of a bit permutation instruction to a general microprocessor. Such an instruction would enable significant speedup over a normal processor when implementing cryptographic operations that require bitwise permutations, such as DES [43].

A more extreme example of this microprocessor-supplemental approach is presented in [44], which adds an instruction to the user-customizable ARC processor that implements an entire round of the DES algorithm. This single instruction vastly increases the speed of DES and 3DES encryption by providing custom hardware and reducing the number of needed memory reads/writes, allowing an speedup of 47 for DES and a speedup of 92 for 3DES [44]. The new DES round instruction also allowed a dramatic reduction in code size. For a processor speed of 200 MHz, 3DES performance was estimated to be 337 Mbps [44].

These sorts of processor extensions have their limitations. Adding a bit permutation instruction is only useful at accelerating algorithms that use bit permutations, and does nothing to address the word-size disparity between the microprocessor (most of which are 32-bit or less) and modern block ciphers (most of which are 128-bit or more). In the case of the specialized DES instruction, it certainly allows speedup of DES, but has no flexibility advantage over a dedicated hardware implementation.

However, a number of different architectures have been proposed that extend this concept even further.

2.4.1 Crypto-CPU

The Crypto-CPU is presented [45] and [46]. It is designed as a processor for low-power communications, and as such much of the design effort went into minimizing power consumption. However, it does offer specialized hardware to speed cryptography.

Based around the MIPS architecture, is uses several modern computer architecture techniques such as superscalar execution and explicit parallelism. In addition to providing a good general-purpose RISC processor, it provides a substitution unit (in the form of SRAM memories) for implementing s-boxes, as well as a programmable bit expansion/permutation unit [45].

The presence of general purpose ALU operations along with the specialized cryptography extensions gives the Crypto-CPU a significant performance boost in cryptographic operation in comparison to other low-power processors. Running at a speed of 100 MHz, the Crypto-CPU offers approximately five times the throughput of the other devices (running at 120 MHz and 133 MHz respectively); at 200 MHz, the performance gap widens to almost a factor of ten. The Crypto-CPU even compares favourably to a general-purpose microprocessor such as an Intel Celeron running at 850 MHz [45].

2.4.2 CryptoManiac

In [8] a number of additions are proposed to a basic microprocessor instruction set to support symmetric-key ciphers. In addition to the standard ALU operations, the extensions include rotation instructions, instructions for constant rotations followed by an XOR operation, modular multiplication instructions, substitution instructions accessing a 256-entry by 32-bit lookup table, and an instruction to implement a partial 64-bit permutation capable of generating 8 bits of the permutation output each time it is executed. These additional instructions improved encryption operation performance over a baseline processor by 59% for processors which already have rotate instructions, and 74% for processors without [8].

The CryptoManiac architecture [47] is partially derived from the instruction set operations presented in [8]. It is a processor based around a 4-wide 32-bit Very Long Instruction Word (VLIW) processor optimized for cryptographic operation, meaning it has no cache and only simple branch prediction. To integrate more easily into communication systems, the device is session oriented, meaning that it can operate for multiple channels of data in parallel or sequence.

Compared to many of the architectures seen so far, the CryptoManiac has a somewhat unique design. It interfaces with a host processor via an input queue which buffers requests for the services of the device (create private key session, delete private key session, and encrypt/decrypt data). There are a number of CryptoManiac Processing Elements (PEs) in parallel, and the requests are sent to them by a scheduler. The scheduler maintains sessions where possible, and uses a Least Recently Used (LRU) allocation scheme to send new sessions to the PEs. The outputs of the PEs are sent to an output queue. The system also has a storage element called the keystore which is used to hold key and substitution data, and is shared between all of the PEs to allow context switching [47].

Naturally, the CryptoManiac PEs are the core to the functionality of the system. Each processing element is a 4-wide 4-stage VLIW processor which has a local instruction memory to store the instructions for the algorithms the PE is implementing. Instructions are fetched and branches predicted in the first stage, decoded and registers accessed in the second stage, processed through the four parallel 32-bit functional units in the third stage (which also may access data memory), and then results are written to the appropriate places in the final stage [47].

The functional units can execute a limited number of instructions, derived from those in [8], and consisting of Boolean AND and XOR, addition, subtraction, rotation, s-box lookups, and multiplication (regular or modular). The instructions are classified into three types: tiny, short, and long. Tiny and short instructions can be executed simultaneously in various combinations, whereas the long instructions (multiplication) can only be executed by themselves [47].

The four 32-bit functional units are nearly identical, each consisting of a logical unit (providing XOR and AND logic) followed by a 32-bit adder, 32-bit rotator, and 1 kB s-box cache in parallel, all of which are followed by another logical unit. Two of the four functional units also have a pipelined 32-bit multiplier in parallel to all the rest [47]. Clearly, such a system offers great flexibility.

The estimated speed of the CryptoManiac is 360 MHz, and performance numbers suggest a speed improvement of anywhere from 32% to 290% over a 600 MHz Alpha 21264 microprocessor, depending on the algorithm. For example, an implementation of Rijndael on the CryptoManiac can achieve an encryption rate of up to 64 megabytes per second, more than 2.25 times faster than the corresponding software implementation [47]. This is a significant performance boost, and the architecture has the flexibility to support other algorithms as well.

2.4.3 Programmable Processor for Cryptography

In [48], the design of a high speed programmable architecture to handle a variety of cryptographic algorithms is presented. The architecture is based around several functional units controlled by a control unit ROM. Hardware units for addition/subtraction and XOR are provided, as well as an EPROM to implement substitutions and permutations. These operations primarily support private-key ciphers.

A modular multiplication and exponentiation unit is also provided to support public-key ciphers such as RSA. The unit implements a loop-unrolled version of the

Montgomery multiplication algorithm to achieve higher throughput [48].

The different functional units are controlled by a control unit which receives instructions, bit lengths, and round numbers as inputs from a master processor. The control unit has a ROM which stores processing cycle counts for the different instructions and different bit lengths. The control unit takes over from the master processor sending control signals to the various functional units, and then writes the valid output after the necessary number of clock cycles [48]. This processor is thus very much like a microprogrammed controller – a very small, limited, task-specific microprocessor – rather than an extended general microprocessor.

The processor was fabricated in 2 micron technology, and managed to achieve a speed of 77 MHz. This allowed it to achieve a DES throughput of 44 megabytes per second and a throughput of 300 kilobits per second for RSA encryption. The authors suggest that if the design were fabricated in submicron technology (for example, 0.18 μm CMOS technology) much higher clock speeds and throughputs could be achieved [48].

2.4.4 Reconfigurable Public Key Architecture

Though beyond the scope of this thesis, which focuses on the design of an architecture to support private-key functionality, [49] offers an interesting reconfigurable processor for public-key cryptography. Despite the technical difficulties of implementing public-key cryptography in hardware, in some respects the implementation of a reconfigurable general-purpose public-key cryptographic architecture is easier than a private-key system simply because the class of base operations is much more limited. The design in [49] is based around implementing those few basic problems, defined in the IEEE P1363 public key cryptography standard: integer factorization, discrete logarithms, and elliptic curves.

This Domain-Specific Reconfigurable Cryptographic Processor (DSRCP) is designed around a microcode controller that interfaces with a 32 by 32-bit reconfigurable datapath. There is also dedicated SHA-1 hardware to satisfy the requirements of the IEEE P1363 standard. The DSRCP receives instructions from an external controller, which it then decodes in the controller. This is because some of the instructions use others in their implementation. Thus, when an instruction is decoded, the necessary microcode is fetched from a ROM and used to control the datapath [49].

The reconfigurable datapath has a local register file, a fast adder unit, a comparator unit, and reconfigurable unit. The reconfigurable unit can be reconfigure on the fly to one of three operations: Montgomery multiplication/reduction, multiplication in $GF(2^n)$, and $GF(2^n)$ inversion. The system is also accommodating to different bit widths (though larger bit sizes naturally require more iterations in the hardware) [49]. It can be used to implement elliptic curve computations, mathematics of finite fields, and exponentiation of large numbers – all basic operations of various forms of public-key cryptography. This smaller set of basic operations makes such a reconfigurable processor much easier to develop for public-key cryptography than the widely varied private-key cryptographic algorithms.

2.5 Summary of Cryptographic Hardware Architectures

A variety of cryptographic hardware architectures have been surveyed in this chapter. Dedicated hardware implementations tend to achieve speed through brute force, efficiency, and parallelism, with fully loop-unrolled and pipelined implementations offering the greatest performance but least flexibility, since they cannot even be used in different modes of operation without a performance hit.

Implementations in programmable hardware such as FPGAs are similar, offering high speeds but little flexibility other than the ability to reprogram with a different algorithm. However, a number of FPGA-based reconfigurable platforms offer a much greater degree of flexibility, whether specifically meant for cryptography or not.

Non-FPGA-based programmable solutions mostly take the form of extended microprocessors. Either extra instructions are added to a general microprocessor to accelerate cryptographic operations, or else smaller processors supporting only cryptography-oriented instructions are proposed. Such systems offer a high degree of flexibility and a significant speedup over pure software implementations, but often cannot approach the throughput levels of dedicated hardware.

The SHERIF cryptographic hardware module will take a somewhat novel approach to implementing a flexible cryptographic platform. Its design will be developed in the following chapters.

Chapter 3

Algorithm Survey and Analysis

To design a hardware architecture optimized for cryptographic operations, it is necessary to determine what capabilities need to be provided and optimized so as to give such a cryptographic hardware module an advantage over a general-purpose microprocessor. This can only be determined by study of the algorithms the device must implement.

There are a multitude of cryptographic algorithms in existence; many are detailed in [3] and [50]. New algorithms are also being developed on an ongoing basis. Analysis of all such cryptographic algorithms would prove impossible, so it is necessary to limit the scope of the algorithms that must be considered.

It is possible to remove algorithms from consideration by applying several broad criteria. Algorithms that are outdated, or that have been shown to have severe weakness to specialized attacks, will be discounted immediately. Likewise, algorithms that are not widely used will be discounted, since a cryptographic hardware module to implement such algorithms would likely be economically infeasible.

This leaves algorithms that are widely used and/or standardized to be considered. Standardized algorithms are selected by government institutions such as NIST [51] or through open processes such as NESSIE [10]. Thus, the analysis will focus on a small set of algorithms that have received much public study and have been standardized or were viable candidates in a standardization process. Furthermore, the algorithms are all symmetric-key block ciphers or hash functions.

3.1 Algorithms Under Consideration

NIST has long been responsible for setting security standards for the U.S. Government. The current standard for data confidentiality, known as the AES, is the algorithm Rijndael [17]. It replaces the older DES [43], although DES is still used in many applications. NIST also specifies the hash function SHA-1 [52] which is based on the older hash function MD5 [3]. These standardized algorithms are natural choices for consideration. MD5 will be excluded from consideration since SHA-1 is a close derivative of it.

The NESSIE selection process was in progress at the start of this research, and thus a number of algorithms considered to be strong contenders were selected for consideration: Camellia [16], RC6 [53], SAFER++$_{128}$ [54], and SHACAL (which is based on the SHA-1 hash function) [55].

At the conclusion of the NESSIE project, Camellia, AES, and a version of SHACAL were recommended algorithms [56]. No attacks against or weaknesses of RC6 or SAFER++$_{128}$ were discovered, but they were not recommended due to intellectual property concerns in the case of RC6, and concerns about certain structural elements and low security margins in SAFER++$_{128}$ [56]. Nevertheless, they are included in this analysis as representatives of modern cipher design, rather than standardized algorithms.

The following sections examine these algorithms in detail. However, the algorithms as depicted are only meant to describe the general structure of the algorithms, and do not necessarily provide all the specific details needed for implementation. For example, details on the S-boxes or bit permutations and expansions will be glossed over.

3.1.1 Notation

In the following algorithm descriptions, some specialized notations are used to ensure uniformity among the descriptions.

- $SBOX()$ is used to represent a table lookup. Different-valued s-boxes will be differentiated by unique numbers or subscripts (SBOX1, SBOX2, and so on).
- $\oplus$ is the XOR operation.
- & is used to concatenate bit strings.
- $N_{(X)}$ represents byte X of bit string N, where the least significant byte is byte 0.
- $N_{(X...Y)}$ represents bytes X to Y of bit string N, where the least significant byte is byte 0.
- $\leftarrow$ represents assignment.
- $\wedge$, $\vee$, and $\neg$ represent bitwise Boolean AND, OR, and NOT respectively.
- $N <<< X$ represents a left rotation of bit string N by X bits. $N >>> X$ is a right rotation by X bits.
- $+$, $-$, and $\times$ represent addition, subtraction, and multiplication, respectively.

The notation can be used in various combinations. Other unique notations are described in the algorithms in which they appear.

3.1.2 AES

The AES algorithm, Rijndael, is relatively simple as described in [17]. Like most block ciphers, it iterates through a number of identical rounds, applying different round keys each time. It is somewhat unique in the way the specification describes it, since it organizes the input data into a two-dimensional 4×4 array of bytes. The four main operations – SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey() – operate on this array.

The SubBytes() operation is a non-linear byte substitution on each byte of the input [17]. This sort of structure is commonly known as an S-box or substitution table, although it can be implemented with boolean logic as well.

The ShiftRows() operation is a fixed permutation of the bytes of the input data. The input bytes are rearranged into a different order.

The AddRoundKey() operation is a simple bitwise exclusive-or (XOR) of the input data and the current round key.

The MixColumns() operation is perhaps the most complex. Each 32 bit word of the input is considered a polynomial over $GF(2^8)$ and multiplied modulo $x^4 + 1$ with $a(x) = \{03\}\,x^3 + \{01\}\,x^2 + \{01\}\,x + \{02\}$. This can also be viewed as a matrix multiplication, and there are a number of ways of implementing this operation. This operation produces a result in which each output but is the XOR of a subset of the input bits.

AES has a variable key size. The version described in this analysis has a 128-bit key size and 10 rounds. Other key sizes simply require a greater number of rounds and a different key schedule.

As mentioned above, the standard description of Rijndael organizes the plaintext into an array. To make comparisons to other algorithms easier, it was necessary to write the algorithm in terms of operations on a 128-bit vector. This is shown in Algorithm 3.1.1.

Algorithm 3.1.1 AES (Rijndael) Encryption

```
Input: 128-bit plaintext $M$.
Input: 11 128-bit round keys $w_0$ to $w_{10}$.

$N \leftarrow M \oplus w_0$

for $i = 1$ to $9$ do
  $N \leftarrow \text{SBOX}(N_{(15)})$ & $\text{SBOX}(N_{(14)})$ & ... & $\text{SBOX}(N_{(0)})$

  $N \leftarrow N_{(15)}$ & $N_{(10)}$ & $N_{(5)}$ & $N_{(0)}$ & $N_{(11)}$ & $N_{(6)}$ & $N_{(1)}$ & $N_{(12)}$
        & $N_{(7)}$ & $N_{(2)}$ & $N_{(13)}$ & $N_{(8)}$ & $N_{(3)}$ & $N_{(14)}$ & $N_{(9)}$ & $N_{(4)}$

  $N \leftarrow N_{(15...12)} \times [\{03\}\,x^3 + \{01\}\,x^2 + \{01\}\,x + \{02\}]$ modulo $x^4 + 1$
        & $N_{(11...8)} \times [\{03\}\,x^3 + \{01\}\,x^2 + \{01\}\,x + \{02\}]$ modulo $x^4 + 1$
        & $N_{(7...4)} \times [\{03\}\,x^3 + \{01\}\,x^2 + \{01\}\,x + \{02\}]$ modulo $x^4 + 1$
        & $N_{(3...0)} \times [\{03\}\,x^3 + \{01\}\,x^2 + \{01\}\,x + \{02\}]$ modulo $x^4 + 1$

  $N \leftarrow N \oplus w_i$
end for

$N \leftarrow \text{SBOX}(N_{(15)})$ & $\text{SBOX}(N_{(14)})$ & ... & $\text{SBOX}(N_{(0)})$

$N \leftarrow N_{(15)}$ & $N_{(10)}$ & $N_{(5)}$ & $N_{(0)}$ & $N_{(11)}$ & $N_{(6)}$ & $N_{(1)}$ & $N_{(12)}$
      & $N_{(7)}$ & $N_{(2)}$ & $N_{(13)}$ & $N_{(8)}$ & $N_{(3)}$ & $N_{(14)}$ & $N_{(9)}$ & $N_{(4)}$

Output: 128-bit ciphertext $C \leftarrow N \oplus w_{10}$
```

The AES decryption algorithm, given in Algorithm 3.1.2, is similar.

Algorithm 3.1.2 AES (Rijndael) Decryption

```
Input: 128-bit ciphertext $C$.
Input: 11 128-bit round keys $w_0$ to $w_{10}$.

$N \leftarrow C \oplus w_{10}$

for $i = 9$ down to 1 do
    $N \leftarrow N_{(15)}$ & $N_{(2)}$ & $N_{(5)}$ & $N_{(8)}$ & $N_{(11)}$ & $N_{(14)}$ & $N_{(1)}$ & $N_{(4)}$
            & $N_{(7)}$ & $N_{(10)}$ & $N_{(13)}$ & $N_{(0)}$ & $N_{(3)}$ & $N_{(6)}$ & $N_{(9)}$ & $N_{(12)}$

    $N \leftarrow \text{SBOX}^{-1}(N_{(15)})$ & $\text{SBOX}^{-1}(N_{(14)})$ & ... & $\text{SBOX}^{-1}(N_{(0)})$

    $N \leftarrow N \oplus w_i$

    $N \leftarrow N_{(15...12)} \times [\{0b\}\, x^3 + \{0d\}\, x^2 + \{09\}\, x + \{0e\}]$ modulo $x^4 + 1$
            & $N_{(11...8)}\, [\times \{0b\}\, x^3 + \{0d\}\, x^2 + \{09\}\, x + \{0e\}]$ modulo $x^4 + 1$
            & $N_{(7...4)}\, [\times \{0b\}\, x^3 + \{0d\}\, x^2 + \{09\}\, x + \{0e\}]$ modulo $x^4 + 1$
            & $N_{(3...0)}\, [\times \{0b\}\, x^3 + \{0d\}\, x^2 + \{09\}\, x + \{0e\}]$ modulo $x^4 + 1$
end for

$N \leftarrow N_{(15)}$ & $N_{(2)}$ & $N_{(5)}$ & $N_{(8)}$ & $N_{(11)}$ & $N_{(14)}$ & $N_{(1)}$ & $N_{(4)}$
        & $N_{(7)}$ & $N_{(10)}$ & $N_{(13)}$ & $N_{(0)}$ & $N_{(3)}$ & $N_{(6)}$ & $N_{(9)}$ & $N_{(12)}$

$N \leftarrow \text{SBOX}(N_{(15)})$ & $\text{SBOX}(N_{(14)})$ & ... & $\text{SBOX}(N_{(0)})$

Output: 128-bit plaintext $M \leftarrow N \oplus w_0$
```

The key schedule of AES, which takes the initial 128-bit key and expands it into the 11 round keys used in encryption and decryption, is shown in Algorithm 3.1.3.

3.1.3 DES

DES is the old NIST standard [43], which has been officially replaced by the AES. However, it is certain that DES is still in use in existing systems, or in modified versions such as Triple-DES [3]. DES is a Feistel cipher structure, with a 64-bit block size and a 56-bit key size, and 16 rounds. The DES encryption algorithm is given in Algorithm 3.1.4. Note that the details of the bitwise permutations and expansions are not given. Also, note that the S-boxes in DES take a 6-bit input and produce a 4-bit

Algorithm 3.1.3 AES (Rijndael) Key Expansion

```
Input: 128-bit key $K$.
Input: 32-bit round constants $R_i = [x^{i-1}, \{00\}, \{00\}, \{00\}]$ over $GF(2^8)$.
Output: 11 128-bit round keys $w_0$ to $w_{10}$.
Comments: Let temp be a 32-bit variable.

$w_0 \leftarrow K$

for $i = 1$ to 10 do
  temp $\leftarrow (w_{i-1})_{(3...0)}$

  temp $\leftarrow$ temp$_{(2...0)}$ & temp$_{(3)}$

  temp $\leftarrow$ SBOX $\left(\text{temp}_{(3)}\right)$ & SBOX $\left(\text{temp}_{(2)}\right)$
           & SBOX $\left(\text{temp}_{(1)}\right)$ & SBOX $\left(\text{temp}_{(0)}\right)$

  temp $\leftarrow$ temp $\oplus R_i$

  $(w_i)_{(15...12)} \leftarrow \text{temp} \oplus (w_{i-1})_{(15...12)}$

  $(w_i)_{(11...8)} \leftarrow (w_i)_{(15...12)} \oplus (w_{i-1})_{(11...8)}$

  $(w_i)_{(7...4)} \leftarrow (w_i)_{(11...8)} \oplus (w_{i-1})_{(7...4)}$

  $(w_i)_{(3...0)} \leftarrow (w_i)_{(7...4)} \oplus (w_{i-1})_{(3...0)}$
end for
```

output, which will account for the apparent size mismatch in the given description.

Algorithm 3.1.4 DES Encryption

```
Input: 64-bit plaintext $M$.
Input: 16 48-bit round keys $k_1$ to $k_{16}$
Comment: Let $L$ and $R$ be 32-bit variables.
Comment: Let $X$ be a 48-bit variable.

$N \leftarrow \mathrm{IP}(M)$ (initial permutation)
$L \leftarrow N_{(7...4)}, R \leftarrow N_{(3...0)}$

for $i = 1$ to 16 do
  temp $\leftarrow R$
  $X \leftarrow \mathrm{E}(R)$ (bit expansion of $R$ to 48 bits)
  $X \leftarrow X \oplus k_i$
  $R \leftarrow \mathrm{SBOX1}(X_{(\text{Bits 47 to 42})})$ & $\mathrm{SBOX2}(X_{(\text{Bits 41 to 36})})$ & $\mathrm{SBOX3}(X_{(\text{Bits 35 to 30})})$
     & $\mathrm{SBOX4}(X_{(\text{Bits 29 to 24})})$ & $\mathrm{SBOX5}(X_{(\text{Bits 23 to 18})})$ & $\mathrm{SBOX6}(X_{(\text{Bits 17 to 12})})$
     & $\mathrm{SBOX7}(X_{(\text{Bits 11 to 6})})$ & $\mathrm{SBOX8}(X_{(\text{Bits 5 to 0})})$
  $R \leftarrow \mathrm{P}(R)$ (bit permutation $P$)
  $L \leftarrow$ temp
end for

Output: 64-bit ciphertext $C \leftarrow \mathrm{IP}^{-1}(R \mathbin{\&} L)$ (inverse permutation)
```

Decryption with DES is identical, since this is a Feistel cipher, and thus simply requires that the keys be applied in the reverse order. This is shown in Algorithm 3.1.5.

The 16 48-bit subkeys are determined from the 56-bit key using the key schedule described in Algorithm 3.1.6. Note that it is also possible to generate the round keys within each round of computation, rather than doing all computations up front. The 56-bit key is actually represented with 64 bits, where the extra bits are odd parity bits.

3.1.4 Camellia

The version of Camellia [16] under consideration has a 128-bit block size, a 128-bit key size, and 18 rounds. The key schedule converts the 128-bit key into 26 64-bit subkeys.

Algorithm 3.1.5 DES Decryption

```
Input: 64-bit ciphertext $C$.
Input: 16 48-bit round keys $k_1$ to $k_{16}$
Comment: Let $L$ and $R$ be 32-bit variables.
Comment: Let $X$ be a 48-bit variable.

$N \leftarrow \mathrm{IP}(M)$ (initial permutation)
$L \leftarrow N_{(7...4)}, R \leftarrow N_{(3...0)}$

for $i = 16$ down to 1 do
  temp $\leftarrow R$
  $X \leftarrow \mathrm{E}(R)$ (bit expansion of $R$ to 48 bits)
  $X \leftarrow X \oplus k_i$
  $R \leftarrow \mathrm{SBOX1}(X_{(\text{Bits 47 to 42})})$ & $\mathrm{SBOX2}(X_{(\text{Bits 41 to 36})})$ & $\mathrm{SBOX3}(X_{(\text{Bits 35 to 30})})$
     & $\mathrm{SBOX4}(X_{(\text{Bits 29 to 24})})$ & $\mathrm{SBOX5}(X_{(\text{Bits 23 to 18})})$ & $\mathrm{SBOX6}(X_{(\text{Bits 17 to 12})})$
     & $\mathrm{SBOX7}(X_{(\text{Bits 11 to 6})})$ & $\mathrm{SBOX8}(X_{(\text{Bits 5 to 0})})$
  $R \leftarrow \mathrm{P}(R)$ (bit permutation $P$)
  $L \leftarrow$ temp
end for

Output: 64-bit plaintext $M \leftarrow \mathrm{IP}^{-1}(R$ & $L)$ (inverse permutation)
```

Algorithm 3.1.6 DES Key Schedule

```
Input: 64-bit key $K$ including 8 odd parity bits.
Output: 16 48-bit round keys $k_1$ to $k_{16}$
Comment: Let $T$ be a 56-bit variable.
Comment: Let $L$ and $R$ be 28-bit variables.
Comment: Let $v_i = 1$ when $i \in [1, 2, 9, 16]$, else $v_i = 2$.

$T \leftarrow \mathrm{PC1}(K)$ (permuted choice 1, selects the 56 key bits and permutes them)
$L \leftarrow T_{(\text{Bits 55 to 28})}$
$R \leftarrow T_{(\text{Bits 27 to 0})}$

for $i = 1$ to 16 do
  $L \leftarrow L <<< v_i$
  $R \leftarrow R <<< v_i$
  $k_i \leftarrow \mathrm{PC2}(L$ & $R)$ (permuted choice 2, selects and permutes 48 bits for subkey)
end for
```

The specification for Camellia uses a series of nested function calls to describe the algorithm, but to allow easy comparison to other algorithms, it was necessary to rewrite the description as is shown in Algorithm 3.1.7.

The decryption operation of Camellia is identical to encryption, since it is a Feistel cipher [3]. Thus, the same algorithm is used, but the keys are applied in the reverse order. The subkeys are determined by a key schedule based on the cipher, which is shown in Algorithm 3.1.8.

3.1.5 SAFER++$_{128}$

SAFER++$_{128}$ is a specific variation of the SAFER++ algorithm which has a block size of 128-bits, a key size of 128-bits, and 7 rounds [54]. SAFER++$_{128}$ has some interesting operations, making it rather different from most of the other algorithms considered. There are several operations in particular that bear mentioning in terms of how the are actually implemented.

There is a nonlinear layer in which certain bytes are are used to compute an exponential value which replaces them, and other bytes are used to compute logarithmic values. The exponential computation replaces byte x with $45^x \mathrm{mod} 257$ with the convention that when x has a value of 128, then $45^{128} \mathrm{mod} 257 = 256$ is represented by 0. The logarithmic computation replaces a byte y with $\log_{45} y$ with the convention that when $y = 0$, then $\log_{45} 0$ is represented by 128.

However, such computations are complex. We can replace them by pre-computing the results for possible values of x and y and using S-boxes to implement these operations. This S-box representation of these operations will be used in the description of SAFER++$_{128}$ given the algorithm descriptions below.

The other unique operation in SAFER++$_{128}$ is called the 4-PHT in the specification [54]. This operation is an invertible linear transformation that operates on 32

Algorithm 3.1.7 Camellia Encryption

```
Input: 128-bit plaintext $M$.
Input: 4 64-bit subkeys $kw_t$ for $t \in (1, 2, 3, 4)$.
Input: 18 64-bit subkeys $k_r$ for $r \in (1, 2, 3, \dots, 18)$.
Input: 4 64-bit subkeys $kl_v$ for $v \in (1, 2, 3, 4)$.

Comment: Let $L$, $R$, $LN$, $L_{temp}$, $R_{temp}$, and temp be 64-bit variables.
Comment: Let $LH$ and $RH$ be 32-bit variables.

$N \leftarrow M \oplus (kw_1 \,\&\, kw_2)$
$L \leftarrow N_{(15...8)}$
$R \leftarrow N_{(7...0)}$

for $r = 1$ to 18 do
  temp $\leftarrow L$
  $L \leftarrow L \oplus k_r$
  $L \leftarrow$ SBOX1($L_{(7)}$) & SBOX2($L_{(6)}$) & SBOX3($L_{(5)}$) & SBOX4($L_{(4)}$)
          & SBOX2($L_{(3)}$) & SBOX3($L_{(2)}$) & SBOX4($L_{(1)}$) & SBOX1($L_{(0)}$)
  $LN_{(7)} \leftarrow L_{(7)} \oplus L_{(5)} \oplus L_{(4)} \oplus L_{(2)} \oplus L_{(1)} \oplus L_{(0)}$
  $LN_{(6)} \leftarrow L_{(7)} \oplus L_{(6)} \oplus L_{(4)} \oplus L_{(3)} \oplus L_{(1)} \oplus L_{(0)}$
  $LN_{(5)} \leftarrow L_{(7)} \oplus L_{(6)} \oplus L_{(5)} \oplus L_{(3)} \oplus L_{(2)} \oplus L_{(0)}$
  $LN_{(4)} \leftarrow L_{(6)} \oplus L_{(5)} \oplus L_{(4)} \oplus L_{(3)} \oplus L_{(2)} \oplus L_{(1)}$
  $LN_{(3)} \leftarrow L_{(7)} \oplus L_{(6)} \oplus L_{(2)} \oplus L_{(1)} \oplus L_{(0)}$
  $LN_{(2)} \leftarrow L_{(6)} \oplus L_{(5)} \oplus L_{(3)} \oplus L_{(1)} \oplus L_{(0)}$
  $LN_{(1)} \leftarrow L_{(5)} \oplus L_{(4)} \oplus L_{(3)} \oplus L_{(2)} \oplus L_{(0)}$
  $LN_{(0)} \leftarrow L_{(7)} \oplus L_{(4)} \oplus L_{(3)} \oplus L_{(2)} \oplus L_{(1)}$
  $L \leftarrow R \oplus LN$
  $R \leftarrow$ temp

  if $r = 6$ or $r = 12$ then
    $L_{temp} \leftarrow L$, $R_{temp} \leftarrow R$
    $k_1 \leftarrow kl_{2r/6-1}$, $k_2 \leftarrow kl_{2r/6}$
    $RH \leftarrow (((L_{temp})_{(7...4)} \wedge (k_1)_{(7...4)}) <<< 1) \oplus (L_{temp})_{(3...0)}$
    $LH \leftarrow (RH \vee (k_1)_{(3...0)}) \oplus (L_{temp})_{(7...4)}$
    $L \leftarrow LH \,\&\, RH$
    $LH \leftarrow ((R_{temp})_{(3...0)} \vee (k_2)_{(3...0)}) \oplus (R_{temp})_{(7...4)}$
    $RH \leftarrow ((LH \wedge (k_2)_{(7...4)}) <<< 1) \oplus (R_{temp})_{(3...0)}$
    $R \leftarrow LH \,\&\, RH$
  end if
end for

Output: 128-bit ciphertext $C \leftarrow (R \,\&\, L) \oplus (kw_3 \,\&\, kw_4)$
```

Algorithm 3.1.8 Camellia Key Schedule

```
Input: 128-bit key $K$.
Input: 64-bit key constants $kc_1 \dots kc_4$ = [a09e667f3bcc908b, b67ae8584caa73b2, c6ef372fe94f82be, 54ff53a5f1d36f1c].
Output: 4 64-bit subkeys $kw_t$ for $t \in (1,2,3,4)$, 18 64-bit subkeys $k_r$ for $r \in (1,2,3,\dots,18)$, and 4 64-bit subkeys $kl_v$ for $v \in (1,2,3,4)$.
Comment: Let $L$, $R$, $LN$, and temp be 64-bit variables, and let $KL$ and $KA$ be 128-bit variables.

$KL \leftarrow K$
$L \leftarrow K_{(15\dots8)}$
$R \leftarrow K_{(7\dots0)}$

for $r = 1$ to 4 do
  temp $\leftarrow L$
  $L \leftarrow L \oplus kc_r$
  $L \leftarrow$ SBOX1($L_{(7)}$) & SBOX2($L_{(6)}$) & SBOX3($L_{(5)}$) & SBOX4($L_{(4)}$)
          & SBOX2($L_{(3)}$) & SBOX3($L_{(2)}$) & SBOX4($L_{(1)}$) & SBOX1($L_{(0)}$)
  $LN_{(7)} \leftarrow L_{(7)} \oplus L_{(5)} \oplus L_{(4)} \oplus L_{(2)} \oplus L_{(1)} \oplus L_{(0)}$
  $LN_{(6)} \leftarrow L_{(7)} \oplus L_{(6)} \oplus L_{(4)} \oplus L_{(3)} \oplus L_{(1)} \oplus L_{(0)}$
  $LN_{(5)} \leftarrow L_{(7)} \oplus L_{(6)} \oplus L_{(5)} \oplus L_{(3)} \oplus L_{(2)} \ominus L_{(0)}$
  $LN_{(4)} \leftarrow L_{(6)} \oplus L_{(5)} \oplus L_{(4)} \oplus L_{(3)} \oplus L_{(2)} \ominus L_{(1)}$
  $LN_{(3)} \leftarrow L_{(7)} \oplus L_{(6)} \oplus L_{(2)} \oplus L_{(1)} \oplus L_{(0)}$
  $LN_{(2)} \leftarrow L_{(6)} \oplus L_{(5)} \oplus L_{(3)} \oplus L_{(1)} \oplus L_{(0)}$
  $LN_{(1)} \leftarrow L_{(5)} \oplus L_{(4)} \oplus L_{(3)} \oplus L_{(2)} \oplus L_{(0)}$
  $LN_{(0)} \leftarrow L_{(7)} \oplus L_{(4)} \oplus L_{(3)} \oplus L_{(2)} \oplus L_{(1)}$
  $L \leftarrow R \oplus LN$
  $R \leftarrow$ temp

  if $r = 2$ then
    $L \leftarrow KL_{(15\dots8)} \oplus L$
    $R \leftarrow KL_{(7\dots0)} \oplus R$
  end if
end for

$KA \leftarrow (L \ \& \ R)$
$kw_1 \leftarrow (KL <<< 0)_{(7\dots4)}$, $kw_2 \leftarrow (KL <<< 0)_{(3\dots0)}$, $kw_3 \leftarrow (KA <<< 111)_{(7\dots4)}$,
$kw_4 \leftarrow (KA <<< 111)_{(3\dots0)}$
$kl_1 \leftarrow (KA <<< 30)_{(7\dots4)}$, $kl_2 \leftarrow (KA <<< 30)_{(3\dots0)}$, $kl_3 \leftarrow (KL <<< 77)_{(7\dots4)}$,
$kl_4 \leftarrow (KL <<< 77)_{(3\dots0)}$
$k_1 \leftarrow (KA <<< 0)_{(7\dots4)}$, $k_2 \leftarrow (KA <<< 0)_{(3\dots0)}$, $k_3 \leftarrow (KL <<< 15)_{(7\dots4)}$,
$k_4 \leftarrow (KL <<< 15)_{(3\dots0)}$, $k_5 \leftarrow (KA <<< 15)_{(7\dots4)}$, $k_6 \leftarrow (KA <<< 15)_{(3\dots0)}$,
$k_7 \leftarrow (KL <<< 45)_{(7\dots4)}$, $k_8 \leftarrow (KL <<< 45)_{(3\dots0)}$, $k_9 \leftarrow (KA <<< 45)_{(7\dots4)}$,
$k_{10} \leftarrow (KL <<< 60)_{(3\dots0)}$, $k_{11} \leftarrow (KA <<< 60)_{(7\dots4)}$, $k_{12} \leftarrow (KA <<< 60)_{(3\dots0)}$,
$k_{13} \leftarrow (KL <<< 94)_{(7\dots4)}$, $k_{14} \leftarrow (KL <<< 94)_{(3\dots0)}$, $k_{15} \leftarrow (KA <<< 94)_{(7\dots4)}$,
$k_{16} \leftarrow (KA <<< 94)_{(3\dots0)}$, $k_{17} \leftarrow (KL <<< 111)_{(7\dots4)}$, $k_{18} \leftarrow (KL <<< 111)_{(3\dots0)}$
```

bits. It is given as a matrix multiplication, but the specification also provides an alternate implementation using 6 8-bit additions modulo 2^8. In the specification below, the term 4-PHT is used for brevity, but can be replaced with the implementation shown in Algorithm 3.1.9.

Algorithm 3.1.9 4-PHT

```
Input: 32-bit input $X$ composed of individual bytes $[a, b, c, d]$.
Comment: Let $A, B, C, D$ be intermediate 8-bit variables.
$D \leftarrow d + a + b + c$ modulo 256
$C \leftarrow c + D$ modulo 256
$B \leftarrow b + D$ modulo 256
$A \leftarrow a + D$ modulo 256
Output: 32-bit $Y \leftarrow A \;\&\; B \;\&\; C \;\&\; D$
```

Note that there is also an inverse of the 4-PHT, which is referred to as 4-IPHT, which is implemented similarly [54].

The encryption operation of SAFER++$_{128}$ is shown in Algorithm 3.1.10. In this description, 4-PHT is abbreviated as a function called PHT.

Decryption is similar to encryption, but requires the same operations in a different order, as shown in Algorithm 3.1.11. IPHT is used to represent the inverse of the 4-PHT transform.

Each round uses 2 128-bit subkeys giving 14 round keys, plus one additional subkey is used at the end. The subkeys are generated via Algorithm 3.1.12.

3.1.6 RC6

The RC6 algorithm is fully parameterizable [53], but the version illustrated here is known as RC6-32/20/16, meaning that the algorithm operates on 128-bit blocks, uses a 128-bit key, and has 20 rounds. RC6 is something of a software-oriented algorithm – inclusion of operations like multiplication and addition play to the strengths of general-purpose microprocessors, while they prove quite costly to custom hardware implementations. RC6 encryption is described in Algorithm 3.1.13.

Algorithm 3.1.10 SAFER++$_{128}$ Encryption

```
Input: 128-bit input plaintext $M$.
Input: 15 128-bit subkeys, $k_1$ to $k_{15}$.

$N \leftarrow M$.
for $i = 1$ to $7$ do
    $N \leftarrow \left(N_{(15)} \oplus (k_{2i-1})_{(15)}\right)$ & $\left(N_{(14)} + (k_{2i-1})_{(14)}\right)$ & $\left(N_{(13)} + (k_{2i-1})_{(13)}\right)$
        & $\left(N_{(12)} \oplus (k_{2i-1})_{(12)}\right)$ & $\left(N_{(11)} \oplus (k_{2i-1})_{(11)}\right)$ & $\left(N_{(10)} + (k_{2i-1})_{(10)}\right)$
        & $\left(N_{(9)} + (k_{2i-1})_{(9)}\right)$ & $\left(N_{(8)} \oplus (k_{2i-1})_{(8)}\right)$ & $\left(N_{(7)} \oplus (k_{2i-1})_{(7)}\right)$
        & $\left(N_{(6)} + (k_{2i-1})_{(6)}\right)$ & $\left(N_{(5)} + (k_{2i-1})_{(5)}\right)$ & $\left(N_{(4)} \ominus (k_{2i-1})_{(4)}\right)$
        & $\left(N_{(3)} \oplus (k_{2i-1})_{(3)}\right)$ & $\left(N_{(2)} + (k_{2i-1})_{(2)}\right)$ & $\left(N_{(1)} + (k_{2i-1})_{(1)}\right)$
        & $\left(N_{(0)} \oplus (k_{2i-1})_{(0)}\right)$

    $N \leftarrow \mathrm{SBOX}_{exp}\left(N_{(15)}\right)$ & $\mathrm{SBOX}_{log}\left(N_{(14)}\right)$ & $\mathrm{SBOX}_{log}\left(N_{(13)}\right)$
        & $\mathrm{SBOX}_{exp}\left(N_{(12)}\right)$ & $\mathrm{SBOX}_{exp}\left(N_{(11)}\right)$ & $\mathrm{SBOX}_{log}\left(N_{(10)}\right)$
        & $\mathrm{SBOX}_{log}\left(N_{(9)}\right)$ & $\mathrm{SBOX}_{exp}\left(N_{(8)}\right)$ & $\mathrm{SBOX}_{exp}\left(N_{(7)}\right)$
        & $\mathrm{SBOX}_{log}\left(N_{(6)}\right)$ & $\mathrm{SBOX}_{log}\left(N_{(5)}\right)$ & $\mathrm{SBOX}_{exp}\left(N_{(4)}\right)$
        & $\mathrm{SBOX}_{exp}\left(N_{(3)}\right)$ & $\mathrm{SBOX}_{log}\left(N_{(2)}\right)$ & $\mathrm{SBOX}_{log}\left(N_{(1)}\right)$
        & $\mathrm{SBOX}_{exp}\left(N_{(0)}\right)$

    $N \leftarrow \left(N_{(15)} + (k_{2i})_{(15)}\right)$ & $\left(N_{(14)} \oplus (k_{2i})_{(14)}\right)$ & $\left(N_{(13)} \oplus (k_{2i})_{(13)}\right)$
        & $\left(N_{(12)} + (k_{2i})_{(12)}\right)$ & $\left(N_{(11)} + (k_{2i})_{(11)}\right)$ & $\left(N_{(10)} \oplus (k_{2i})_{(10)}\right)$
        & $\left(N_{(9)} \oplus (k_{2i})_{(9)}\right)$ & $\left(N_{(8)} + (k_{2i})_{(8)}\right)$ & $\left(N_{(7)} + (k_{2i})_{(7)}\right)$
        & $\left(N_{(6)} \oplus (k_{2i})_{(6)}\right)$ & $\left(N_{(5)} \oplus (k_{2i})_{(5)}\right)$ & $\left(N_{(4)} + (k_{2i})_{(4)}\right)$
        & $\left(N_{(3)} + (k_{2i-1})_{(3)}\right)$ & $\left(N_{(2)} \oplus (k_{2i})_{(2)}\right)$ & $\left(N_{(1)} \oplus (k_{2i})_{(1)}\right)$
        & $\left(N_{(0)} + (k_{2i})_{(0)}\right)$

    $N \leftarrow N_{(7)}$ & $N_{(10)}$ & $N_{(13)}$ & $N_{(0)}$ & $N_{(15)}$ & $N_{(2)}$ & $N_{(5)}$ & $N_{(8)}$
        & $N_{(11)}$ & $N_{(14)}$ & $N_{(1)}$ & $N_{(4)}$ & $N_{(3)}$ & $N_{(6)}$ & $N_{(9)}$ & $N_{(12)}$

    $N \leftarrow \mathrm{PHT}\left(N_{(15\ldots12)}\right)$ & $\mathrm{PHT}\left(N_{(11\ldots8)}\right)$ & $\mathrm{4PHT}\left(N_{(7\ldots4)}\right)$ & $\mathrm{PHT}\left(N_{(3\ldots0)}\right)$

    $N \leftarrow N_{(7)}$ & $N_{(10)}$ & $N_{(13)}$ & $N_{(0)}$ & $N_{(15)}$ & $N_{(2)}$ & $N_{(5)}$ & $N_{(8)}$
        & $N_{(11)}$ & $N_{(14)}$ & $N_{(1)}$ & $N_{(4)}$ & $N_{(3)}$ & $N_{(6)}$ & $N_{(9)}$ & $N_{(12)}$

    $N \leftarrow \mathrm{PHT}\left(N_{(15\ldots12)}\right)$ & $\mathrm{PHT}\left(N_{(11\ldots8)}\right)$ & $\mathrm{PHT}\left(N_{(7\ldots4)}\right)$ & $\mathrm{PHT}\left(N_{(3\ldots0)}\right)$
end for

Output: 128-bit ciphertext $C \leftarrow \left(N_{(15)} \oplus (k_{15})_{(15)}\right)$ & $\left(N_{(14)} + (k_{15})_{(14)}\right)$
    & $\left(N_{(13)} + (k_{15})_{(13)}\right)$ & $\left(N_{(12)} \oplus (k_{15})_{(12)}\right)$ & $\left(N_{(11)} \oplus (k_{15})_{(11)}\right)$
    & $\left(N_{(10)} + (k_{15})_{(10)}\right)$ & $\left(N_{(9)} + (k_{15})_{(9)}\right)$ & $\left(N_{(8)} \oplus (k_{15})_{(8)}\right)$
    & $\left(N_{(7)} \oplus (k_{15})_{(7)}\right)$ & $\left(N_{(6)} + (k_{15})_{(6)}\right)$ & $\left(N_{(5)} + (k_{15})_{(5)}\right)$
    & $\left(N_{(4)} \oplus (k_{15})_{(4)}\right)$ & $\left(N_{(3)} \oplus (k_{15})_{(3)}\right)$ & $\left(N_{(2)} + (k_{15})_{(2)}\right)$
    & $\left(N_{(1)} + (k_{15})_{(1)}\right)$ & $\left(N_{(0)} \oplus (k_{15})_{(0)}\right)$
```

Algorithm 3.1.11 SAFER++$_{128}$ Decryption

```
Input: 128-bit ciphertext $C$.
Input: 15 128-bit subkeys, $k_1$ to $k_{15}$.

$N \leftarrow \left(C_{(15)} \oplus (k_{15})_{(15)}\right)$ & $\left(C_{(14)} - (k_{15})_{(14)}\right)$ & $\left(C_{(13)} - (k_{15})_{(13)}\right)$
    & $\left(C_{(12)} \oplus (k_{15})_{(12)}\right)$ & $\left(C_{(11)} \oplus (k_{15})_{(11)}\right)$ & $\left(C_{(10)} - (k_{15})_{(10)}\right)$
    & $\left(C_{(9)} - (k_{15})_{(9)}\right)$ & $\left(C_{(8)} \oplus (k_{15})_{(8)}\right)$ & $\left(C_{(7)} \oplus (k_{15})_{(7)}\right)$
    & $\left(C_{(6)} - (k_{15})_{(6)}\right)$ & $\left(C_{(5)} - (k_{15})_{(5)}\right)$ & $\left(C_{(4)} \oplus (k_{15})_{(4)}\right)$
    & $\left(C_{(3)} \oplus (k_{15})_{(3)}\right)$ & $\left(C_{(2)} - (k_{15})_{(2)}\right)$ & $\left(C_{(1)} - (k_{15})_{(1)}\right)$
    & $\left(C_{(0)} \oplus (k_{15})_{(0)}\right)$

for $i = 7$ down to $1$ do
  $N \leftarrow \mathrm{IPHT}\left(N_{(15\ldots12)}\right)$ & $\mathrm{IPHT}\left(N_{(11\ldots8)}\right)$ & $\mathrm{IPHT}\left(N_{(7\ldots4)}\right)$ & $\mathrm{IPHT}\left(N_{(3\ldots0)}\right)$

  $N \leftarrow N_{(11)}$ & $N_{(6)}$ & $N_{(13)}$ & $N_{(0)}$ & $N_{(7)}$ & $N_{(14)}$ & $N_{(1)}$ & $N_{(5)}$
      & $N_{(15)}$ & $N_{(2)}$ & $N_{(9)}$ & $N_{(4)}$ & $N_{(3)}$ & $N_{(10)}$ & $N_{(5)}$ & $N_{(12)}$

  $N \leftarrow \mathrm{IPHT}\left(N_{(15\ldots12)}\right)$ & $\mathrm{IPHT}\left(N_{(11\ldots8)}\right)$ & $\mathrm{IPHT}\left(N_{(7\ldots4)}\right)$ & $\mathrm{IPHT}\left(N_{(3\ldots0)}\right)$

  $N \leftarrow N_{(11)}$ & $N_{(6)}$ & $N_{(13)}$ & $N_{(0)}$ & $N_{(7)}$ & $N_{(14)}$ & $N_{(1)}$ & $N_{(5)}$
      & $N_{(15)}$ & $N_{(2)}$ & $N_{(9)}$ & $N_{(4)}$ & $N_{(3)}$ & $N_{(10)}$ & $N_{(5)}$ & $N_{(12)}$

  $N \leftarrow \left(N_{(15)} - (k_{2i})_{(15)}\right)$ & $\left(N_{(14)} \oplus (k_{2i})_{(14)}\right)$ & $\left(N_{(13)} \oplus (k_{2i})_{(13)}\right)$
      & $\left(N_{(12)} - (k_{2i})_{(12)}\right)$ & $\left(N_{(11)} - (k_{2i})_{(11)}\right)$ & $\left(N_{(10)} \oplus (k_{2i})_{(10)}\right)$
      & $\left(N_{(9)} \oplus (k_{2i})_{(9)}\right)$ & $\left(N_{(8)} - (k_{2i})_{(8)}\right)$ & $\left(N_{(7)} - (k_{2i})_{(7)}\right)$
      & $\left(N_{(6)} \oplus (k_{2i})_{(6)}\right)$ & $\left(N_{(5)} \oplus (k_{2i})_{(5)}\right)$ & $\left(N_{(4)} - (k_{2i})_{(4)}\right)$
      & $\left(N_{(3)} - (k_{2i-1})_{(3)}\right)$ & $\left(N_{(2)} \oplus (k_{2i})_{(2)}\right)$ & $\left(N_{(1)} \oplus (k_{2i})_{(1)}\right)$
      & $\left(N_{(0)} - (k_{2i})_{(0)}\right)$

  $N \leftarrow \mathrm{SBOX}_{log}\left(N_{(15)}\right)$ & $\mathrm{SBOX}_{exp}\left(N_{(14)}\right)$ & $\mathrm{SBOX}_{exp}\left(N_{(13)}\right)$
      & $\mathrm{SBOX}_{log}\left(N_{(12)}\right)$ & $\mathrm{SBOX}_{log}\left(N_{(11)}\right)$ & $\mathrm{SBOX}_{exp}\left(N_{(10)}\right)$
      & $\mathrm{SBOX}_{exp}\left(N_{(9)}\right)$ & $\mathrm{SBOX}_{log}\left(N_{(8)}\right)$ & $\mathrm{SBOX}_{log}\left(N_{(7)}\right)$
      & $\mathrm{SBOX}_{exp}\left(N_{(6)}\right)$ & $\mathrm{SBOX}_{exp}\left(N_{(5)}\right)$ & $\mathrm{SBOX}_{log}\left(N_{(4)}\right)$
      & $\mathrm{SBOX}_{log}\left(N_{(3)}\right)$ & $\mathrm{SBOX}_{exp}\left(N_{(2)}\right)$ & $\mathrm{SBOX}_{exp}\left(N_{(1)}\right)$
      & $\mathrm{SBOX}_{log}\left(N_{(0)}\right)$

  $N \leftarrow \left(N_{(15)} \oplus (k_{2i-1})_{(15)}\right)$ & $\left(N_{(14)} - (k_{2i-1})_{(14)}\right)$ & $\left(N_{(13)} - (k_{2i-1})_{(13)}\right)$
      & $\left(N_{(12)} \oplus (k_{2i-1})_{(12)}\right)$ & $\left(N_{(11)} \oplus (k_{2i-1})_{(11)}\right)$ & $\left(N_{(10)} - (k_{2i-1})_{(10)}\right)$
      & $\left(N_{(9)} - (k_{2i-1})_{(9)}\right)$ & $\left(N_{(8)} \oplus (k_{2i-1})_{(8)}\right)$ & $\left(N_{(7)} \oplus (k_{2i-1})_{(7)}\right)$
      & $\left(N_{(6)} - (k_{2i-1})_{(6)}\right)$ & $\left(N_{(5)} - (k_{2i-1})_{(5)}\right)$ & $\left(N_{(4)} \oplus (k_{2i-1})_{(4)}\right)$
      & $\left(N_{(3)} \oplus (k_{2i-1})_{(3)}\right)$ & $\left(N_{(2)} - (k_{2i-1})_{(2)}\right)$ & $\left(N_{(1)} - (k_{2i-1})_{(1)}\right)$
      & $\left(N_{(0)} \oplus (k_{2i-1})_{(0)}\right)$
end for

Output: 128-bit plaintext $M \leftarrow N$
```

Algorithm 3.1.12 SAFER++$_{128}$ Key Schedule

```
Input: 128-bit key $K$
Input: 128-bit constants $B_2$ to $B_15$.
Output: 15 128-bit subkeys $k_0$ to $k_15$.
Comment: $XE, XO$ be 136-bit (17-byte) variables.

Comment: Let P be parity byte of $K$ and concatenate:
$XE \leftarrow XO \leftarrow (K \& P)$

$k_1 \leftarrow K$

for $i = 1$ to 7 do
  Rotate each individual byte of XO 6 bits to the left.
  Let temp $\leftarrow$ bytes $18 - 2i$ to $3 - 2i$ mod 17 from XO (allowing wraparound).
  $k_{2i+1} \leftarrow$ temp $+ B_{2i+1}$ (addition is done byte-by-byte modulo 256)

  Rotate each individual byte of XE 3 bits to the left.
  Let temp $\leftarrow$ bytes $17 - 2i$ to $2 - 2i$ mod 17 from XE (allowing wraparound).
  $k_{2i} \leftarrow$ temp $+ B_{2i}$ (addition is done byte-by-byte modulo 256)
end for
```

Algorithm 3.1.13 RC6 Encryption

```
Input: 128-bit input plaintext $M$.
Input: 44 32-bit round keys $S_0 \ldots S_{43}$.
Comment: Let $W, X, Y, Z$ be 32-bit variables.
Comment: Let $t, u$ be temporary 32-bit variables.
$W \leftarrow M_{(15...12)}$
$X \leftarrow M_{(11...8)}$
$Y \leftarrow M_{(7...4)}$
$Z \leftarrow M_{(3...0)}$
$X \leftarrow X + S_0$
$Z \leftarrow Z + S_1$
for $i = 1$ to 20 do
  $t \leftarrow (X \times (2X + 1)) <<< 5$
  $u \leftarrow (Z \times (2Z + 1)) <<< 5$
  $W \leftarrow ((W \oplus t) <<< u) + S_{2i}$
  $Y \leftarrow ((Y \oplus u) <<< t) + S_{2i+1}$
  $(W, X, Y, Z) \leftarrow (X, Y, Z, W)$
end for
$W \leftarrow W + S_{42}$
$Y \leftarrow Y + S_{43}$
Output: 128-bit ciphertext $C \leftarrow W \& X \& Y \& Z$
```

RC6 decryption, shown in Algorithm 3.1.14, is similar to encryption, but not identical.

Algorithm 3.1.14 RC6 Decryption

```
Input: 128-bit ciphertext $C$.
Input 44 32-bit round keys $S_0 \ldots S_{43}$.
Comment: Let $W, X, Y, Z$ be 32-bit variables.
Comment: Let $t, u$ be temporary 32-bit variables.
$W \leftarrow C_{(15 \ldots 12)}$
$X \leftarrow C_{(11 \ldots 8)}$
$Y \leftarrow C_{(7 \ldots 4)}$
$Z \leftarrow C_{(3 \ldots 0)}$
$Y \leftarrow Y - S_{43}$
$W \leftarrow W - S_{42}$
for $i = 20$ down to 1 do
  $(W, X, Y, Z) \leftarrow (Z, W, X, Y)$
  $u \leftarrow (Z \times (2Z + 1)) <<< 5$
  $t \leftarrow (X \times (2X + 1)) <<< 5$
  $Y \leftarrow ((Y - S_{2i+1}) >>> t) \oplus u$
  $W \leftarrow ((W - S_{2i}) >>> u) \oplus y$
end for
$Z \leftarrow Z - S_1$
$X \leftarrow X - S_0$
Output: 128-bit plaintext $M \leftarrow W \,\&\, X \,\&\, Y \,\&\, Z$
```

The large number of round keys are generated by a rather involved key schedule, shown in Algorithm 3.1.15.

3.1.7 SHA-1

The secure hash algorithm SHA-1 is a NIST standard [52] derived from the older MD4 and MD5 algorithms [3]. There are other standardized variations as well, such as SHA-384 and SHA-512, but this analysis focuses on basic SHA-1.

SHA-1 uses word-oriented operations – that is, it works on input data in groups of 32 bits. It takes an arbitrary length message (so long as the length is $< 2^{64} - 1$) in 512-bit blocks and produces a final 160-bit hash value. Should the message size not be a multiple of 512 bits, it is padded according to a scheme described in [52] and [2].

Algorithm 3.1.15 RC6 Key Schedule

```
Input: 128-bit key $K$.
Output: 44 32-bit round keys $S_0 \ldots S_{43}$.
$L_0 \leftarrow K_{(15\ldots12)}$
$L_1 \leftarrow K_{(11\ldots8)}$
$L_2 \leftarrow K_{(7\ldots4)}$
$L_3 \leftarrow K_{(3\ldots0)}$
$S_0 \leftarrow b7e15163$
for $k = 1$ down to 43 do
  $S_k \leftarrow S_{k-1} + 9e3779b9$
end for
$A \leftarrow B \leftarrow j \leftarrow i \leftarrow 0$
for $s = 1$ to 132 do
  $A \leftarrow S_i \leftarrow (S_i + A + B) <<< 3$
  $B \leftarrow L_j \leftarrow (L_j + A + B) <<< (A + B)$
  $i \leftarrow (i + 1)\mathrm{mod}44$
  $j \leftarrow (j + 1)\mathrm{mod}4$
end for
```

Each block is processed in sequence, the result of the hash function's operation on earlier blocks being incorporated into the processing of later blocks. This is shown in Algorithm 3.1.16, which is similar to the description in [2].

SHACAL

SHACAL is something of a unique algorithm in that it is an encryption algorithm based on the hash function SHA-1 [55]. It makes use of the fact that the compression function of SHA-1 (the function that is iterated 80 times) is invertible. Thus, SHA-1 can be used for encryption if a secret key is input to the algorithm as the message, the plaintext is used as the initial value, and the final addition with the initial values is skipped [55]. Thus, SHACAL is a block cipher that has a 160-bit block and a key size of up to 512 bits.

Algorithm 3.1.16 SHA-1 Hash Function

```
Input: properly padded message $y = M_1 \& M_2 \& \dots \& M_n$, where each $M_i$ is a
512-bit block.
Comment: Let $A, B, C, D, E$ be 32-bit variables.

Comment: Let $H_0 \leftarrow 67452301$, $H_1 \leftarrow EFCDAB89$, $H_2 \leftarrow 98BADCFE$,
$H_3 \leftarrow 10325476$, $H_4 \leftarrow C3D2E1F0$

for $i = 1$ to $n$ do
  Comment: Let $M_i$ be defined as a concatenation of 16 32-bit words,
  $M_i = W_0 \& W_1 \& \dots \& W_{15}$

  for $t = 16$ to 79 do
    Comment: Create additional words for subsequent use,
    $W_t \leftarrow (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) <<< 1$
  end for
  $A \leftarrow H_0$, $B \leftarrow H_1$, $C \leftarrow H_2$, $D \leftarrow H_3$, $E \leftarrow H_4$

  for $t = 0$ to 79 do
    if $0 \leq t \leq 19$ then
      Comment: Let $K = 5A827999$.
      temp $\leftarrow (A <<< 5) + ((B \wedge C) \vee ((\neg B) \wedge D)) + E + W_t + K$
    else if $20 \leq t \leq 39$ then
      Comment: Let $K = 6ED9EBA1$.
      temp $\leftarrow (A <<< 5) + (B \oplus C \oplus D) + E + W_t + K$
    else if $40 \leq t \leq 59$ then
      Comment: Let $K = 8F1BBCDC$.
      temp $\leftarrow (A <<< 5) + ((B \wedge C) \vee (B \wedge D) \vee (C \wedge D)) + E + W_t + K$
    else if $60 \leq t \leq 79$ then
      Comment: Let $K = CA62C1D6$.
      temp $\leftarrow (A <<< 5) + (B \oplus C \oplus D) + E + W_t + K$
    end if
    $E \leftarrow D$
    $D \leftarrow C$
    $C \leftarrow B <<< 30$
    $B \leftarrow A$
    $A \leftarrow$ temp
  end for

  $H_0 \leftarrow H_0 + A$, $H_1 \leftarrow H_1 + B$, $H_2 \leftarrow H_2 + C$,
  $H_3 \leftarrow H_3 + D$, $H_4 \leftarrow H_4 + E$
end for

Output: 160-bit hash value HASH $\leftarrow H_0 \& H_1 \& H_2 \& H_3 \& H_4$
```

3.2 Algorithm Analysis

A cursory examination of the preceding algorithms indicates a few common traits characteristic to block ciphers and hash functions. Chief among these similarities is the fact that all of the algorithms are round-oriented, iterating through the same structure multiple times to improve security. This type of iterative behaviour is present in most (if not all) block ciphers not included in this study as well, and is an important aspect on which to build. Within each round, the operations are usually sequential, occurring in a distinct order. This, too, can be exploited.

All of the modern block ciphers operate on 128-bit blocks, with the legacy cipher DES the only one operating on 64-bit blocks. The hash function SHA-1 operates on 512-bit blocks and produces 160-bit output hash values, although its predecessors MD4 and MD5 produce 128-bit hash values. Thus, a standard block size of 128 bits seems to be a reasonable choice.

The specifics of each algorithm under consideration are drastically different. Each algorithm was analyzed to identify the basic primitive operations on which it is built, noting the type of operation, size and type of operands, and frequency of occurrence. Any operations with special implementation techniques, such as the 4-PHT in SAFER++$_{128}$ [54] were interpreted in terms of their implementation details. The results of this analysis are shown in Table 3.1. The values in this table are for the whole algorithm, not just a single round.

As can be seen in Table 3.1, there is only a relatively small set of operations used in these widely varying algorithms. This suggests that a reconfigurable cryptographic hardware module is feasible, since it need only support this set of operations. However, it should be noted that the different algorithms have different operand sizes in some cases, so this needs to be accounted for in any reconfigurable design.

- All of the algorithms studies use bitwise Boolean operations, usually the

		AES (10-round)	**Camellia** (18-round)	**RC6** (20-round)	**SAFER++$_{128}$** (7-round)	**SHA-1** (80-round)	**DES** (16-round)
XOR	Bit Size	128	128, 64, 32	32	8	32	48, 32
	# ops.	11	2, 36, 8	40	112	272	16, 16
AND	Bit Size		32			32	
	# ops.		4			100	
OR	Bit Size		32			32	
	# ops.		4			60	
NOT	Bit Size					32	
	# ops.					20	
Rotation/	Bit Size		32	32		32	
Shift	# ops.		4	120		160	
Addition/	Bit Size			32	8	32	
Subtraction	# ops.			84	448	325	
Multiplication	Bit Size			32			
	# ops.			40			
S-boxes/	Bit Size	8×8	8×8		8×8		6×4
LUTs	# ops.	160	144		112		128
Byte	Bit Size	16-byte	16-byte	16-byte	16-byte	20-byte	8-byte
Permutations	# ops.	10	18	20	14	80	16
Bit	Bit Size						64, 32
Permutations	# ops.						2, 16
Linear	Bit Size	32	64				
Transformation	# ops.	36	18				
Bit	Bit Size						32-to-48
Expansion	# ops.						16

Table 3.1: Basic Operations of Block Ciphers and Hash Functions

exclusive-or (XOR), but some also using AND, OR, and NOT. Operand sizes range from 8 bits to 128 bits.

- Several algorithms use rotations, which are usually called circular shifts. In all cases, the shifts work on 32 bits at a time.

- Addition or subtraction modulo 2^n where n is the bit size of the operands is also common. In most instances, the operands are 32-bit, but 8-bit operands are also used.

- Multiplication is only present in RC6, in the form of 32-bit $\times$ 32-bit multiplication modulo 2^{32}.

- Substitutions in the form of S-boxes are also common, appearing mostly as 8-bit $\times$ 8-bit substitutions. These substitutions are also frequently referred to as LUTs.

- Permutation of data bytes is also common, although often it is 32-bit words that are reordered, not the bytes directly. This operation is something of a data routing issue, and will be considered as such.

- Bitwise permutations of the data appear only in DES, since it is an inefficient operation for software. Nevertheless, it must be considered.

- Linear transformations such as the MixColumns() operation in AES [17] or the P() function in Camellia [16] are becoming more common in modern block ciphers. In these operations, each output bit is produced from the XOR of a subset of the input bits.

- DES also uses bitwise expansions, taking a 32-bit string and expanding it to a 48-bit string by duplicating certain bit values.

To support the algorithms considered in this study, a reconfigurable cryptographic hardware module would have to support all of the above operations.

3.3 Summary of Algorithm Analysis

This chapter has detailed the analysis of a number of cryptographic algorithms. This analysis identified a small number of core operations necessary to implement the algorithms under consideration. These basic operations included bitwise Boolean operations, rotations and shifts, addition and subtraction, multiplication, substitutions and permutations, and linear transformations. These operations form the basis of the system designed in the following chapters.

Chapter 4

Component Implementation

Having identified the set of operations desired in a flexible cryptographic hardware module, it was then necessary to design hardware components to implement these operations. Since the most common block size of the studied algorithms was 128 bits, that size will be used as the input size to the hardware components. Some components such as Boolean operations, addition/subtraction, multiplication, and shifting will have two 128-bit inputs, whereas other components such as the substitutions and linear transforms will have only a single 128-bit input.

The algorithm survey also showed that the Boolean operations, addition/subtraction, and LUTs all operated on 8-bit operands at some point. Thus, it was decided to make those components configurable and usable at the byte-level, and if larger operand sizes are needed, they must be accommodated by combining byte-level operations. Thus, component of this type will have 16 8-bit operations available, which can be combined to allow processing of larger operands.

Multiplication, shifting/rotation, linear transformations, bit permutations, and bit expansions all operate on 32-bit or larger operands. Thus, these components will be configurable and usable at the word-level, providing 4 32-bit operations which can be combined to allow processing of larger operands.

Configuration of the components is accomplished via control signals. Setting values on the control lines will set a particular configuration of the component. Discussion of the source of those values can be found in Section 4.3. The components must also be able to be disabled or bypassed, since not every component will be used in every algorithm implementation.

4.1 Basic Operational Components

There are six basic components which can be implemented to provide all of the functionality specified in the algorithm survey.

- The Boolean component handles XOR, AND, OR, and NOT operations.
- The Shifter component handles shifts and rotations.
- The Add/Sub component handles addition and subtraction.
- The Multiplier component handles multiplication.
- The LUT component handles substitutions/S-boxes.
- The XORnet component handles bit permutations, bit expansions, and linear transformations.

Note that byte permutations are considered to be a part of data routing, and are discussed in Section 4.2.2.

The following sections discuss each of the components in detail.

4.1.1 Boolean Logic Component

The basic Boolean operations – XOR, AND, OR, and NOT – are trivial to implement in hardware, as they are the basic gates on which most hardware is built. However, it is desirable to have a Boolean component that can be configured for any of the four specified operations. This can be achieved by using a 4-to-1 multiplexer to select which operation is desired, as shown in Figure 4.1. There is also a 2-to-1 multiplexer shown to allow all the Boolean logic to be bypassed if desired. The values of the control signals to the multiplexers are part of the component's configuration.

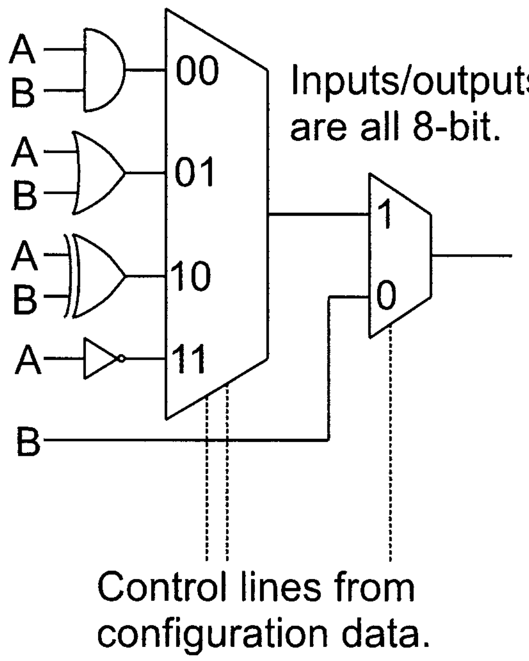

A
B
A
B
A
B
A
B
00
01
10
11
1
0
Inputs/outputs are all 8-bit.
Control lines from configuration data.

Figure 4.1: 8-bit Boolean Logic Component

Since they are always used as bitwise operations, the Boolean operations scale easily – two 8-bit XORs in parallel are equivalent to a 16-bit XOR. Note that when referring to an n-bit Boolean operation, it is implicitly understood that it means n 2-input gates in parallel. Thus 16 8-bit operations in parallel will be sufficient to meet the needs specified by the algorithm analysis and earlier in this chapter, and this is shown in Figure 4.2. Note that the control lines for all 16 are separate, so each byte can be configured for a different operation. Thus the overall component takes

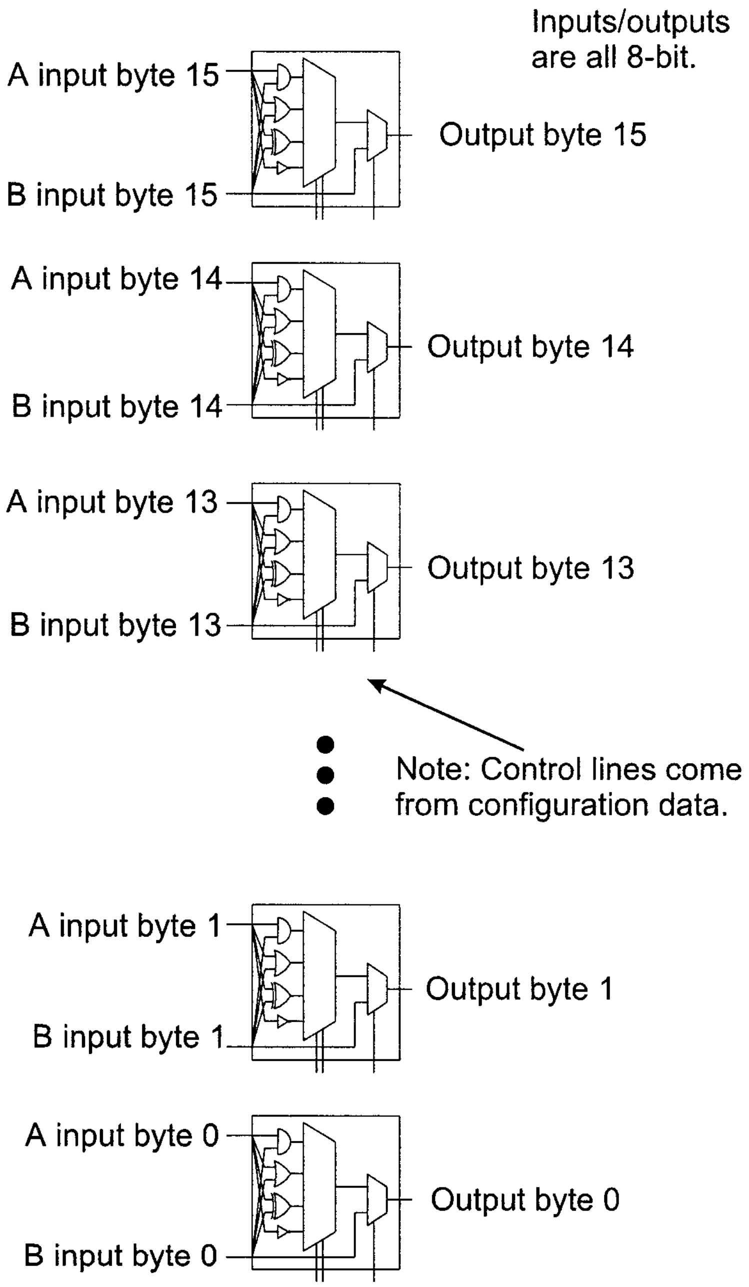


Figure 4.2: Full Boolean Logic Component

two 16-byte inputs and produces one 16-byte output which is the result of each byte being subjected to a bitwise Boolean operation.

A sample of the VHDL code implementing the core Boolean logic component described here can be found in Appendix A, Section A.1, on page 196.

4.1.2 Shifter Component

The shifter component is built around a 32-bit rotation/shift component that can be configured either for rotation (circular shift) or shifting operation, as well as shift/rotate left or right. All shifts are logical – that is, they shift in 0 values. The control signals of the shifter are provided by the configuration, but the amount of the shift/rotation is controlled by a 5-bit input that comes from the datapath.

The 32-bit rotation/shift takes place in a single clock cycle because it is implemented as a series of layers of multiplexers, as shown in Figure 4.3. Each layer's multiplexers are controlled by a single bit of the 5-bit shift amount. If the controlling bit is 0, no shift occurs. If the controlling bit is 1, a shift occurs of size 2^n where n is the bit position of the controlling bit: the most significant bit of the amount is in position 4, the least significant is in position 0.

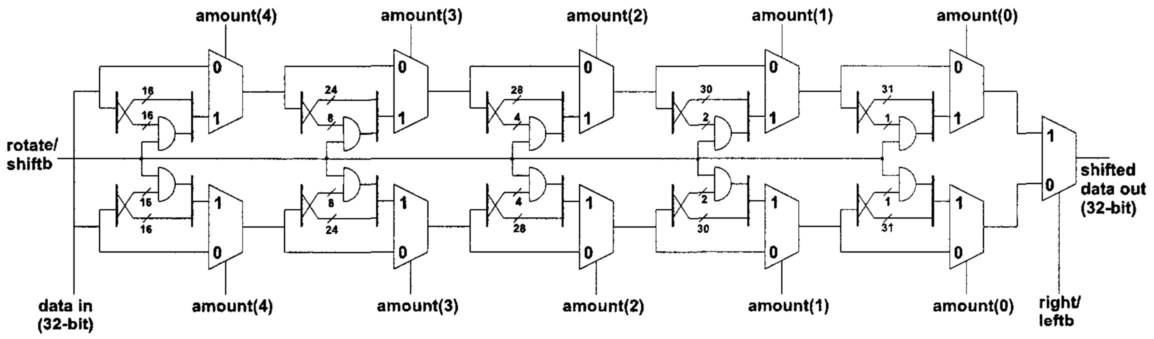


Figure 4.3: 32-bit Shift/Rotate Component

Four such 32-bit shifter components are used in parallel to service a 128-bit datapath block. This can be seen in Figure 4.4. Thus, the overall component takes 4

32-bit data input and 4 5-bit shift amounts and produces 4 32-bit outputs. Note that bypass functionality is also included.

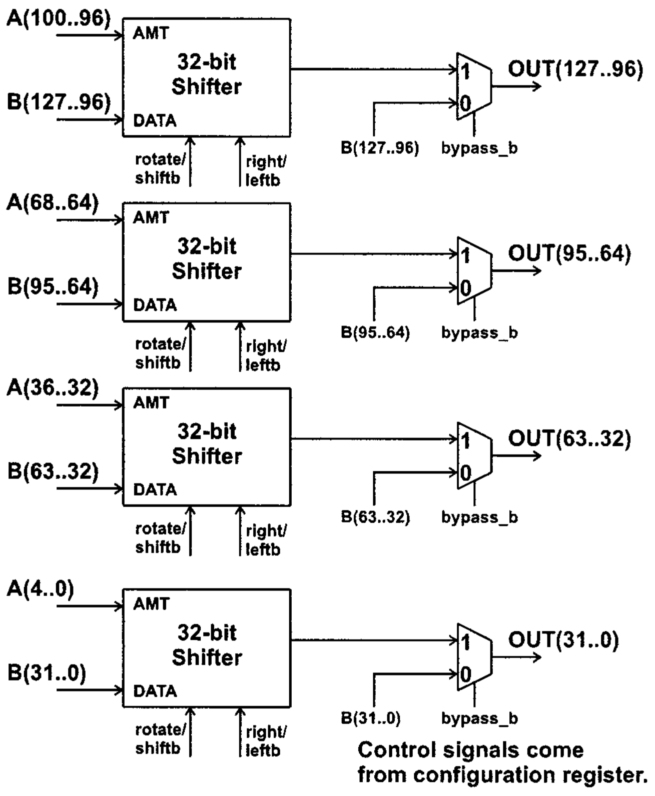


Figure 4.4: Full Shift/Rotate Component

This general shifter architecture is generalizable. An n-bit shifter requires $\log_2(n)$ layers of multiplexers for shifting (in each shift direction), plus an extra multiplexer to select direction. If we consider each 1-bit 2-to-1 multiplexer to be 3 gates, then, accounting for the multiplexers and the necessary AND gates, the implementation of an n-bit bidirectional rotation/shift-in-0 unit requires $6n\log_2(n) + 5n - 2$ gates and $\log_2(n) + 2$ memory elements to store control data.

A barrel shifter could also be used for the shifter component. This would have the added benefit of supporting general bitwise permutations, but would require a significantly larger number of control bits. However, since bitwise permutations are now rarely used in modern algorithms, and because they can be implemented by another basic component, it was decided to use the shifter described above for the prototype proof-of-concept implementation. Future revisions to this design may incorporate a barrel shifter into the system in order to benefit from its greater functionality.

Sample VHDL code used in implementing the core shifter component described here can be found in Appendix A, Section A.2, starting on page 197.

4.1.3 Add/Subtract Component

This component proved most difficult to design even though addition and subtraction are well-studied algorithms, since a variety of operand sizes was required. The natural inclination to use adders of the largest size needed – which can naturally perform smaller additions – was problematic due to the *quantity* of additions required. SAFER++$_{128}$ [54] requires over 400 8-bit additions, and providing that many 32-bit adders (which is the size required by RC6 [53]) would be prohibitive.

Thus, the add/subtract component must be designed around fast 8-bit adders (which can also be used to perform 2's complement subtraction). The basic 8-bit adders can be cascaded together to form larger adders. Whether the adders are cascaded or not is determined by the values of control lines from the configuration. Thus each 8-bit adder component can be configured for addition or subtraction, and cascaded or not cascaded, which affects how the carry-in to each adder is handled. This setup is illustrated in Figure 4.5. Note that the specific implementation of the adder circuit is unimportant, and can be left to the synthesis tool. If the synthesis timing constraints are aggressive enough, it will likely be synthesized as a Carry Look-Ahead Adder (CLA).

It would seem natural to directly cascade these blocks together to allow larger operand sizes, but the structure of SAFER++$_{128}$ requires additions in sequence to compute the 4-PHT [54]. That is, the output of one 8-bit adder is the input to the

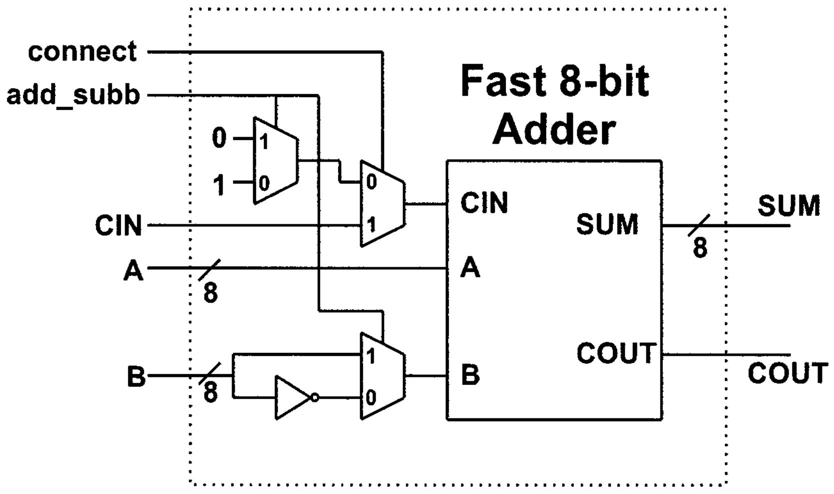


Figure 4.5: 8-bit Add/Subtract Block

next. Rather than require the use of multiple add/subtract components to support this, a further architectural change can be made. A multiplexer can be placed on the input of each adder in the cascade sequence after the first, which selects one of the adder inputs between the current data input and the outputs of the preceding adder. Note that this requires multiplexers of increasing size the further down the cascade it is placed. This scheme is illustrated in Figure 4.6. A maximum of 8 adders can be cascaded together, as shown. Note that the control signals to each of the 8-bit adders and to the various multiplexers are not shown in the figure, but are driven by the configuration data register.

The add/subtract component consists of 16 8-bit adders (to process a 128-bit block), arranged in two banks of 8 adders each. The adders in each bank may be cascaded together to form up to one 64-bit adder, two 32-bit adders, and so on. Furthermore, in each adder bank, every adder after the first can either use the datapath input as one of its inputs, or else use the output of any preceding adder in the same bank. Subtraction is configured in the same way. This overall system can be seen in Figure 4.7. Note that each individual byte out output from the adders can be

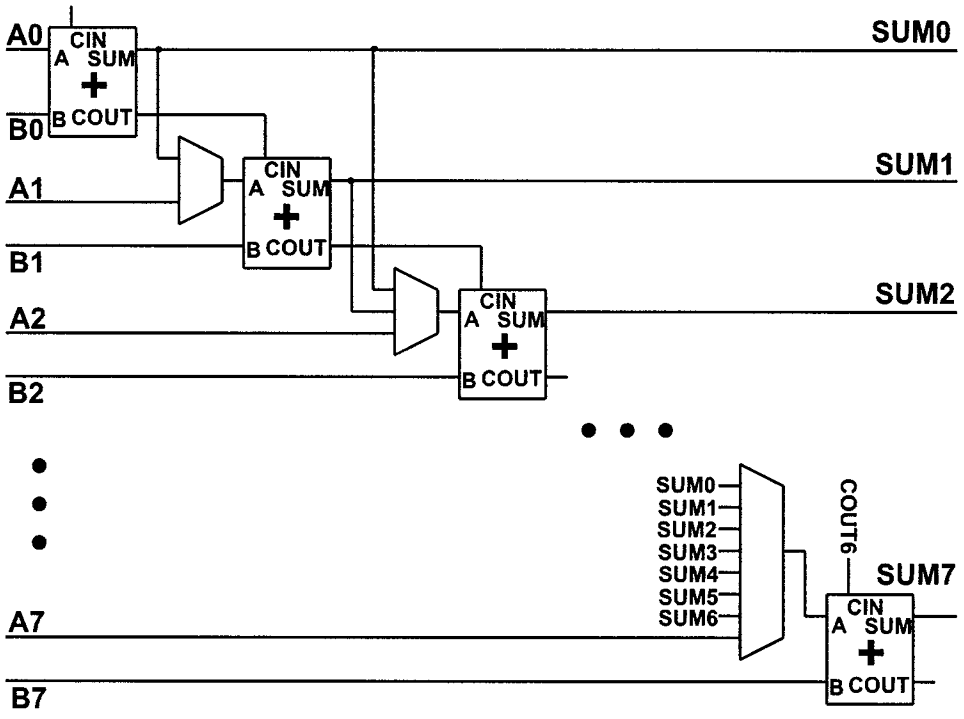


Figure 4.6: One Add/Subtract Bank

bypassed using a layer of multiplexers at the output.

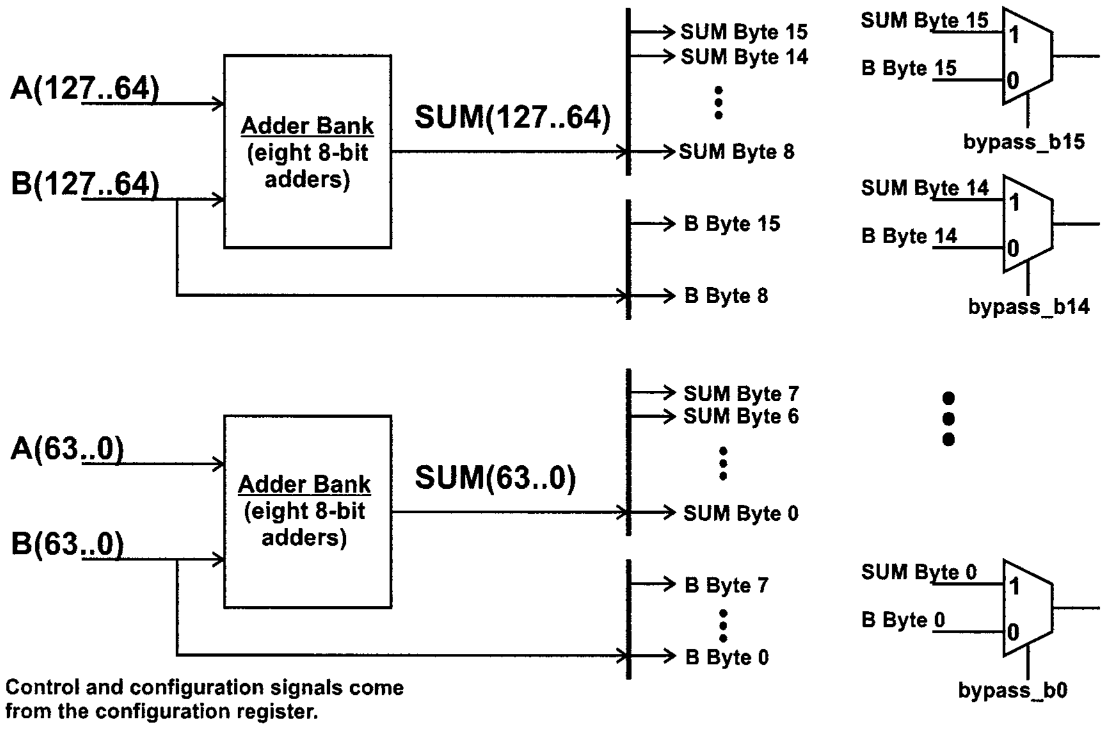


Figure 4.7: Core Add/Subtract Component

Sample VHDL code used in implementing the 8-bit add/subtract component and the 64-bit adder bank described here can be found in Appendix A, Section A.3, starting on page 198.

4.1.4 Multiplier Component

Of the algorithms considered, only RC6 [53] uses multiplication. However, since it is a fairly fundamental operation, it is conjectured that new algorithms might also incorporate multiplication, and so a multiplier component is included despite its current limited use.

The basic operation is 32-bit $\times$ 32-bit multiplication modulo 2^{32}, meaning that the output is only 32 bits, not 64. Multiplication is a costly component to implement, especially when required to compute the product in a single clock cycle. Thus, a 32-bit Wallace tree multiplier was used since it provides a good balance between speed and complexity, and, as shown in [57], more than half of its hardware complexity can be removed since the operation is modulo 2^{32}.

Wallace tree multiplication works by using 32-bit Carry-Save Adders (CSAs) to add the partial products of the multiplicand and multiplier through a tree of CSAs until there are only two values are left, which are then added normally [58, 59] (modulo 2^{32} for RC6). CSAs are much like Carry-Ripple Adders (CRAs) except that, rather than connecting the carries from one full adder to the next, the carry is sent directly to the output [58]. Thus, a 32-bit CSA has 3 32-bit inputs (say a, b and carry-in vector) and 2 32-bit outputs (sum and carry-out vectors). This allows all the full adders to process the input in parallel and improve speed, rather than waiting for the carry to ripple. In the Wallace tree multiplier, the carry values eventually get added in by a later layer of CSAs.

The structure of the Wallace tree used in this multiplier component is shown in Figure 4.8. It takes two 32-bit inputs and produces one 32-bit output.

The partial products for the multiplication $A \times B$ are generated as shown in Figure 4.9. Note that in this figure, the boxes labeled "$A << N$" (where N is some number) represent a left shift-in-0 operation. The partial products are the result of a logical AND of these shifted versions of the multiplicand A and the corresponding bits in the multiplier B (that is, each of the 32 bits of the shifted A value are ANDed with the same bit of B). Only the least significant 32 partial products are generated since this is an implementation of modulo 2^{32} multiplication.

Four such multipliers are used in parallel to allow processing of a 128-bit block. This structure is shown in Figure 4.10. Note that the bypass functionality is still provided, albeit at the word-level. This overall component takes two 4-word inputs and produces a single 4-word output, where each output word is the product of the

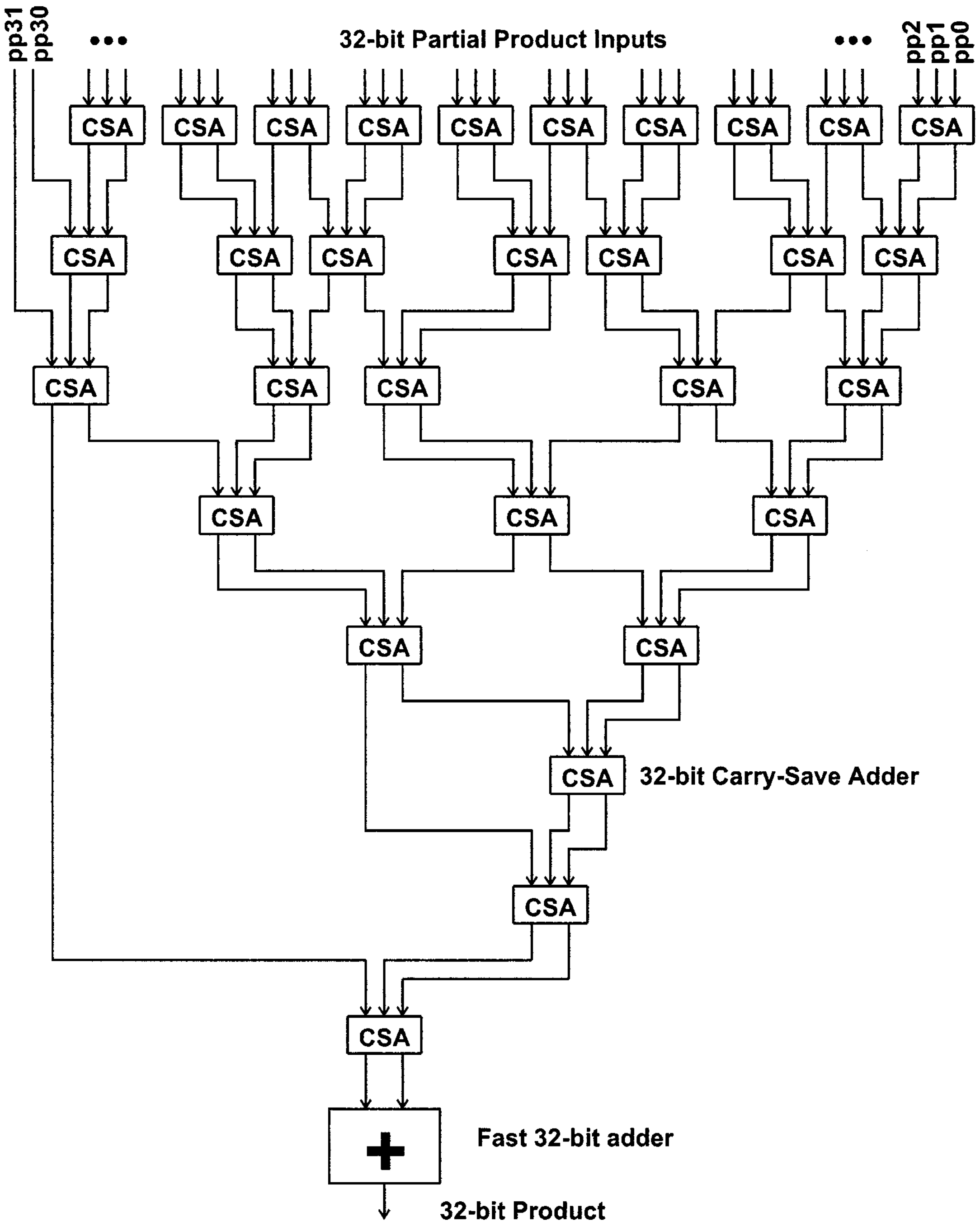


Figure 4.8: 32-bit × 32-bit Wallace Tree Multiplier

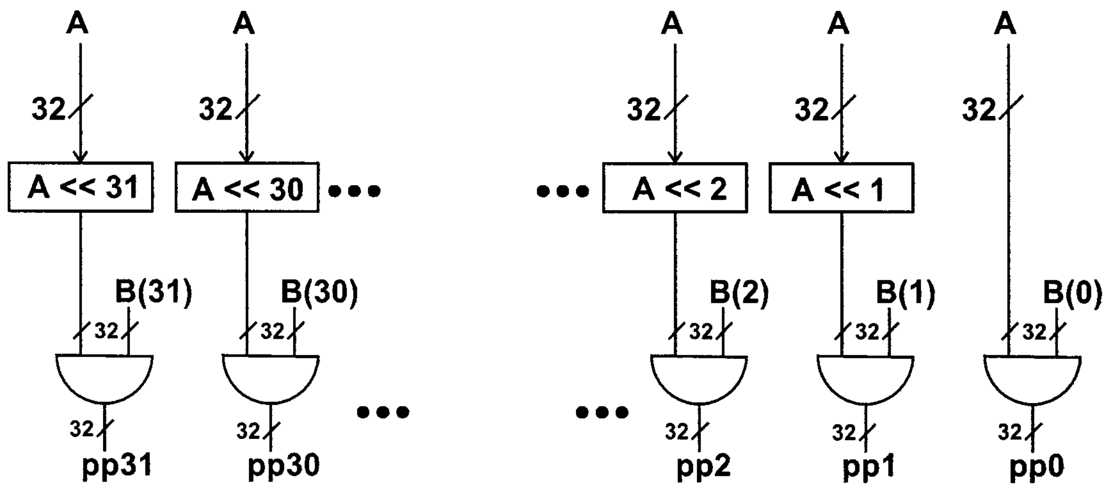


Figure 4.9: Partial Product Generation

corresponding two input words (modulo 2^{32}).

4.1.5 LUT Component

This component performs substitution operations. While many modern algorithms may define their substitutions mathematically as Boolean functions, in the general case it is easiest to implement such substitutions (or S-boxes) as a look-up table (LUT). For our LUT component, an array of sixteen 8×8 LUTs is used, which is sufficient for any of the algorithms under consideration. Each LUT takes an 8-bit input and produces the corresponding 8-bit substitution value as output.

Normally, LUTs would be implemented as memory. The standard forms of memory produced by memory compiler tools introduce a degree of latency – for example, one clock cycle may be required to latch and decode the memory address, and another clock cycle to fetch the data from memory. To avoid this latency, the LUTs have been implemented as banks of flip-flops which are selected via a large multiplexer. The input data controls the multiplexer, which selects 8 bits of output data from the

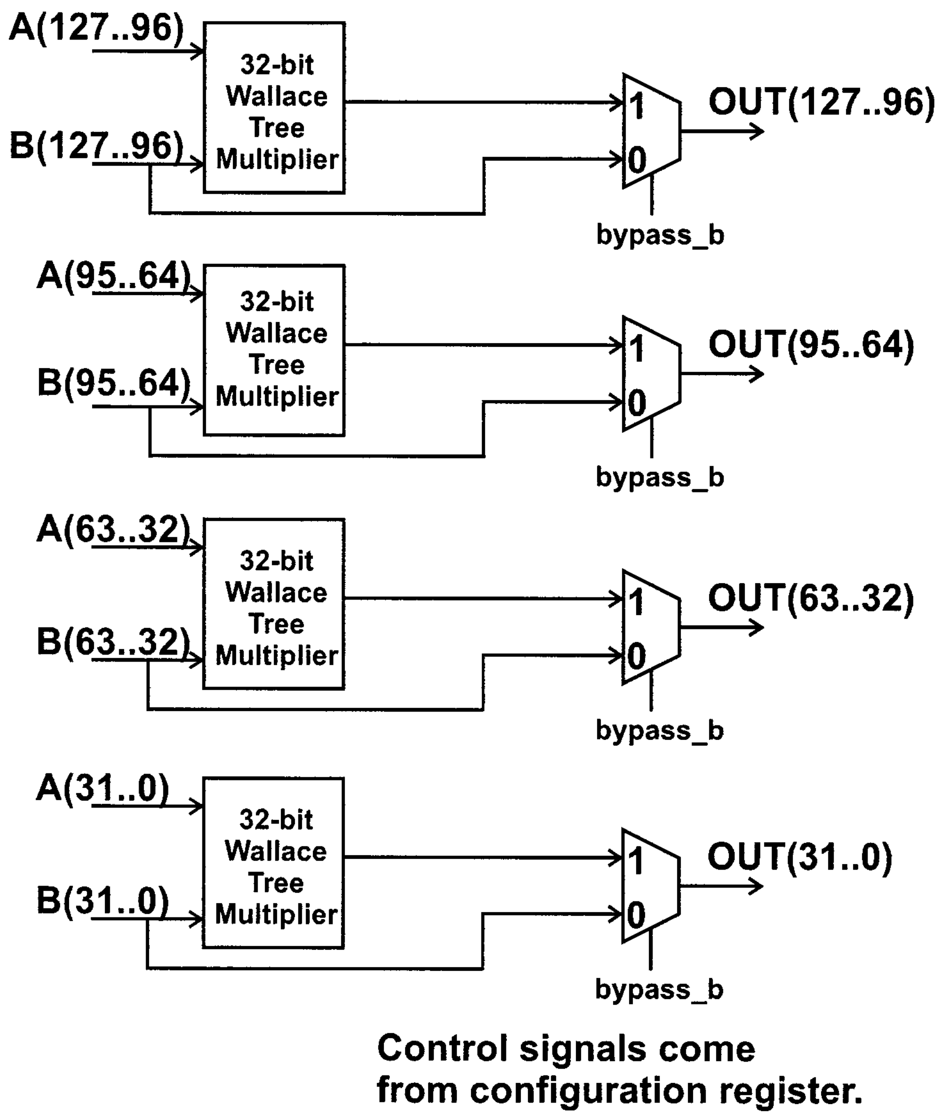

A(127..96)
B(127..96)
32-bit Wallace Tree Multiplier
1
0
OUT(127..96)
bypass_b
A(95..64)
B(95..64)
32-bit Wallace Tree Multiplier
1
0
OUT(95..64)
bypass_b
A(63..32)
B(63..32)
32-bit Wallace Tree Multiplier
1
0
OUT(63..32)
bypass_b
A(31..0)
B(31..0)
32-bit Wallace Tree Multiplier
1
0
OUT(31..0)
bypass_b
Control signals come from configuration register.

Figure 4.10: Full Multiplier Component

flip-flops. This scheme is highly inefficient in terms of area, but is sufficient for prototyping purposes. The substitution values are stored in the flip-flops, which allows the component to be configured to support the different S-boxes required by the various different algorithms. The currently implemented form of the LUT is shown in Figure 4.11.

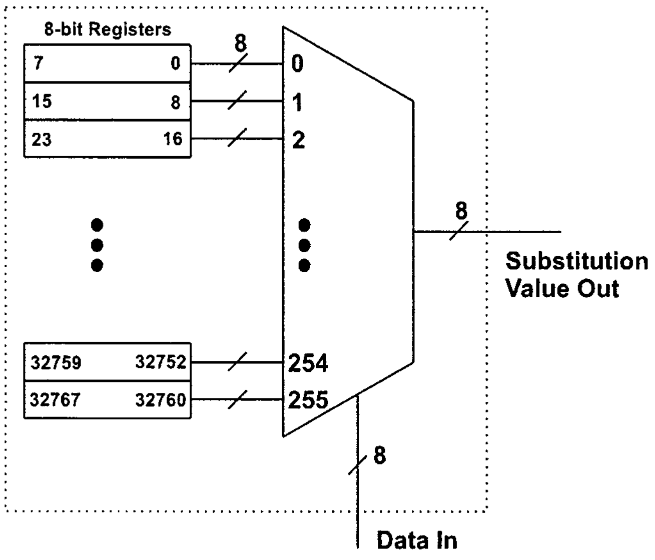


Figure 4.11: Single 8×8 LUT

As stated above, each LUT has an 8-bit input which therefore allows it to select one of 256 possible 8-bit outputs. This means that a single LUT stores a total of $256 \times 8 = 2048$ bits. Thus, the overall LUT component stores 16 times this amount, which is 32768 bits or 4 kB. That is a significant amount of data, which must be set by the configuration. Having 32768 control lines available to set the flip-flops of the LUT would be infeasible, so instead the data is shifted in to the flip-flops one bit at a time during configuration. In essence, the registers of the LUT components are configuration registers, and are part of the configuration chain. This is discussed more completely in Section 4.3.

The overall representation of the component is shown in Figure 4.12. Note that there is only one data input, and that multiplexers can be used to bypass each LUT.

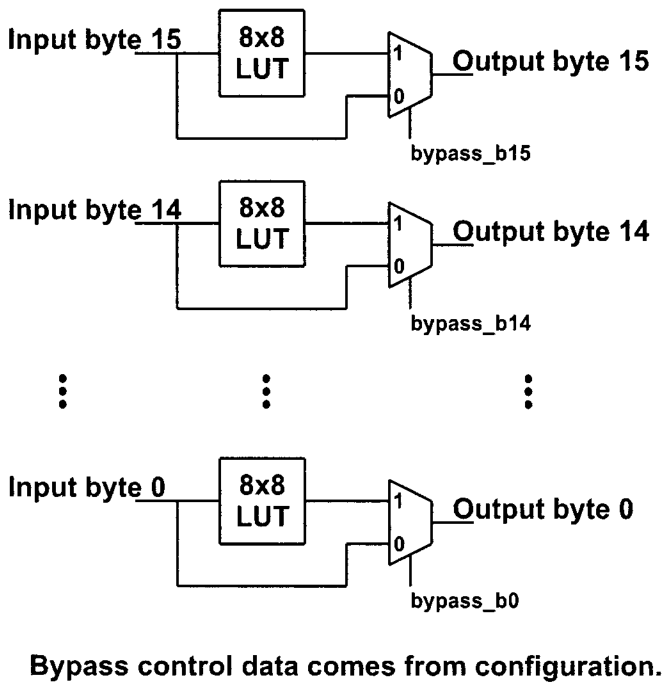


Figure 4.12: Core LUT Component

4.1.6 XORnet Component

The final component, called the XORnet, is perhaps the most general of all the components in the system. It serves to implement a variety of operations, such as the Galois field multiplication by a constant (the MixColumns() operation of AES [17]), bitwise permutations and bit expansions in DES [43], and other bitwise manipulations and linear transformations (such as Camellia's $P()$ function [16]).

The XORnet component operates such that each output bit is generated from the XOR of any number of the input bits. The logic to generate a single output bit is a tree of 2-input XOR gates, with an initial layer of AND gates to allow selection of the desired inputs. This is shown in Figure 4.13. The structure is simply repeated in parallel to produce the output for each desired output bit, so a 32-bit XORnet would repeat the structure in Figure 4.13 32 times.

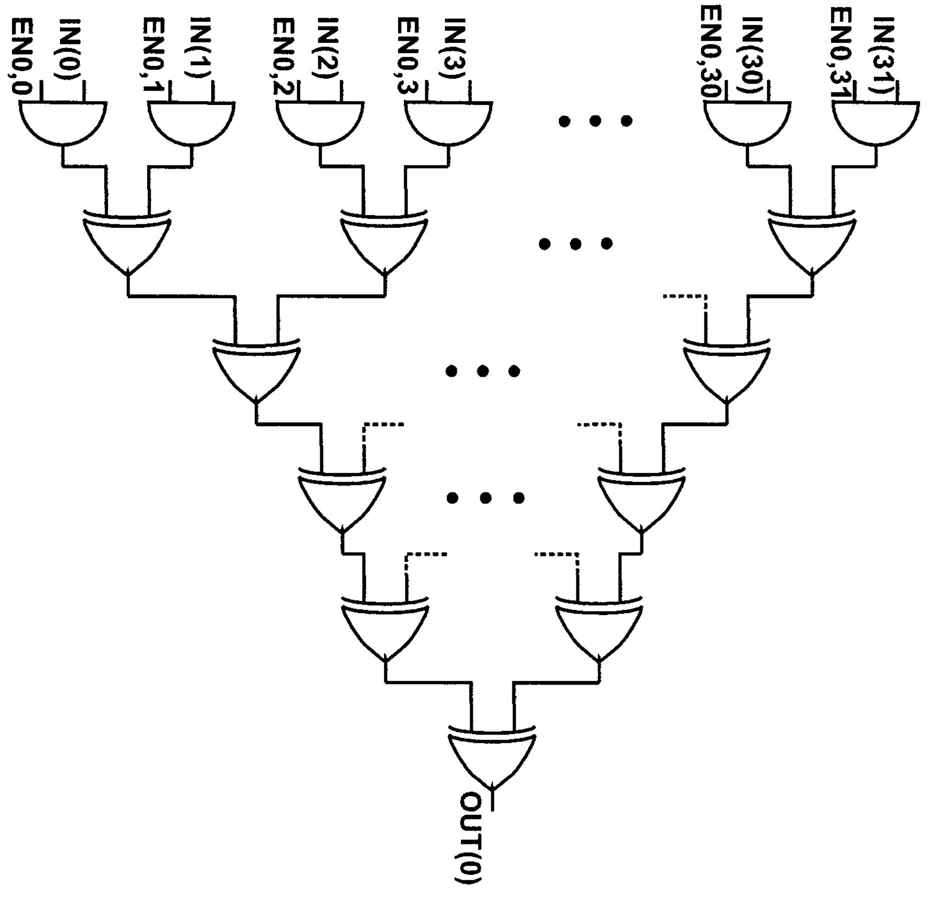


Figure 4.13: Logic to generate single output bit for XORnet.

It is shown in [60] that multiplication by a constant in a Galois field can be implemented in the above fashion. Such operation also mirrors the definition of Camellia's P() function [16]. With respect to bitwise permutations and expansions, it is trivial to see that the XORnet can implement these by having the output bits consist of the XOR of only the desired input bit to be permuted/expanded to that output. Likewise, the XORnet can also implement fixed shifts and rotations, though such operations are more efficiently done in the shifter component.

The major difficulty introduced by this component is that it requires a large amount of configuration data (the enable signals seen in Figure 4.13) – each output bit requires a number of configuration bits equal to the number of input bits to allow potential selection of any of the inputs. Thus, the memory requirement for storing configuration data for the XORnet becomes a major problem – a 32-bit XORnet (which has a 32-bit input and 32-bit output) requires 1024 bits of configuration data.

The hardware cost of the XORnet, in terms of gate count and memory bits, is quite easy to quantify. An n-bit XORnet requires $n \times (2n - 1)$ gates and n^2 bits of memory. Thus, a 32-bit XORnet (the minimum size we need) requires 2016 gates and 1024 bits of control data, and a 64-bit XORnet requires 8128 gates and 4096 bits of control data.

Several approaches involving implementing larger XORnets out of smaller ones were tried. For example, an 8-bit XORnet can be implemented as a monolithic block, or else it can be implemented from 4 4-bit XORnets. This is illustrated in Figure 4.14. The inputs to the smaller XORnets must be set appropriately, and the outputs of each group of XORnets XORed together, but it is feasible.

It was found that the gate count and memory requirements for, say, a 32-bit XORnet implemented out of 8-bit XORnets were exactly the same (note that the 32-bit XORnet requires 16 8-bit XORnets to implement it). Thus, neither monolithic nor compartmentalized designs give any size advantage. However, implementing large XORnets out of smaller ones gives the bonus of greater flexibility – with a monolithic 32-bit XORnet, 4 8-bit XORnets can be emulated, but with the compartmentalized design, in which the 32-bit XORnet is realized from 16 8-bit XORnets, there are 4 times as many 8-bit XORnets available. The only added hardware cost to this approach is the addition of some extra control logic.

AES requires 32-bit XORnets [17], whereas Camellia requires a 64-bit XORnet [16]. DES requires at most a 64-bit XORnet. Thus, it makes the most sense to base

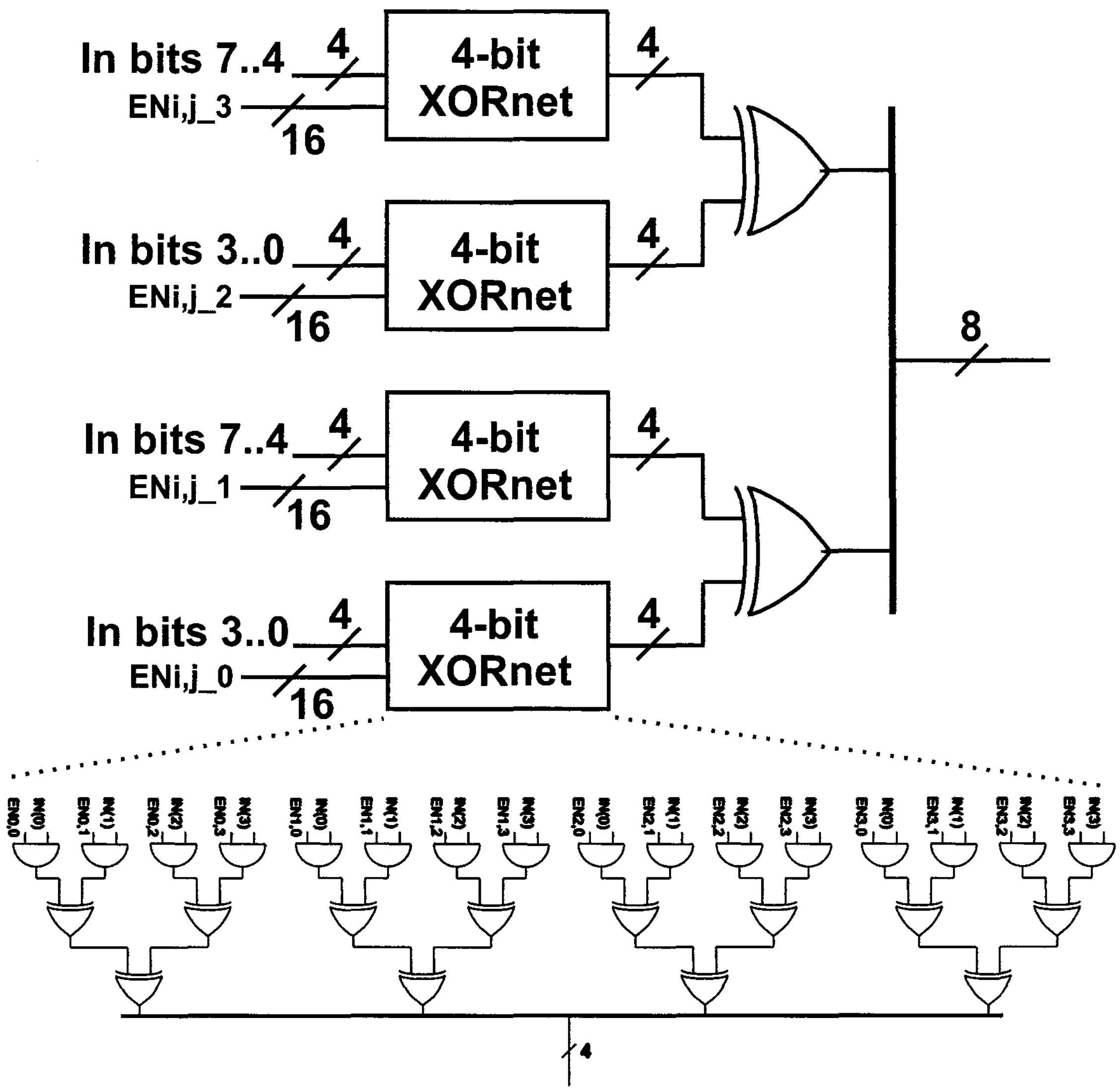


Figure 4.14: Building an 8-bit XORnet from 4 4-bit XORnets.

the implementation of the component around 32-bit XORnets.

The XORnet component consists of 4 32-bit XORnets in parallel. To meet the requirements of Camellia and DES, the option of combining the 4 32-bit XORnets into a single 64-bit XORnet exists by repeating the 64-bit input across the second two XORnets and then running the outputs of all four through an XOR tree to produce a single 64-bit output. This overall arrangement is shown in Figure 4.15. The XORnet is a unary operation, so there is only one 4-word input that produces a 4-word output. Note that the bypass multiplexers operate at the word-level, as well.

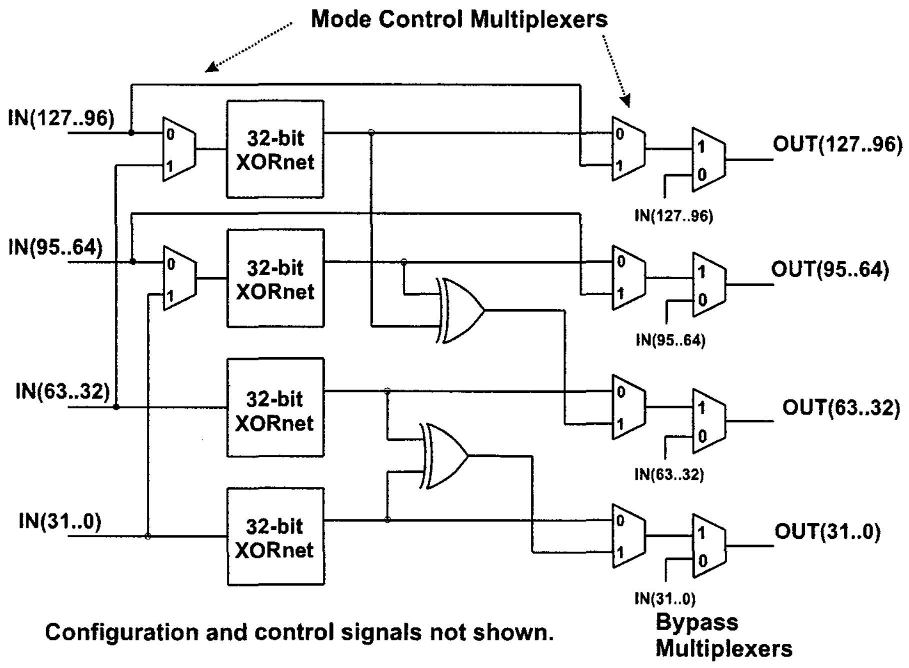


Figure 4.15: Core XORnet Component

4.2 Data Routing and Selection

With the basic operational components designed, it is necessary to consider data routing to those components. While the general form of data processing in block ciphers and hash functions is sequential, not all data is necessarily processed at the same time. For example, in DES [43] and Camellia [16], which are Feistel ciphers, half of the input data is processed and used to update the other half, while that half is swapped into a new position. Thus, it is necessary to have some sort of temporary data storage, to allow input data to be held until needed.

Rather than use an actual memory element, which would require complex addressing and decoding, the concept of the scratch path is introduced (named after a scratch pad of paper often used to hold temporary workings). It is a set of data lines parallel to the datapath, onto which the datapath values may be switched. The scratch path bypasses all computational components completely. When the values on the scratch path are needed, they can be switched back into the datapath on a byte-by-byte basis.

Also, the basic operations often need input other than the data being enciphered. For example, almost every block cipher mixes a round key with the data being enciphered via an XOR operation. Thus, data from two different sources (datapath and key) must be routed to the inputs of a Boolean component. Similarly, several algorithms use shifts by a constant value. Since the Shifter component has an input to allow control of the shift amount, it is clearly necessary for the constant value (stores in some configuration memory) to be routed to the shifter control inputs. These are only two simple examples of how data other than message data may be used by the basic components. Thus, there needs to be a way to switch key data and constant data onto the datapath as well so that the components can operate on data from these different sources. All of this functionality is encapsulated in the input switch.

4.2.1 Input Switch

The input switch is designed to switch the needed data into the operational component and scratch path. An instance of this switch is meant to precede each basic component, to allow each to be have its inputs fully configurable.

The data inputs to the basic operations are selected at the byte level by 4-to-1 multiplexers, selecting between the current datapath input, the scratch path input, a constant value specific to the configuration of the component being switched into, and a key value from the key memory (this is discussed further in Section 5.1.2). This multiplexer arrangement is shown in Figure 4.16.

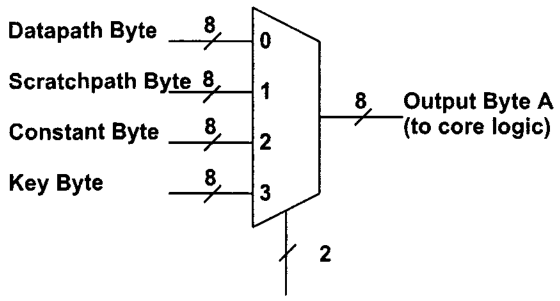


Figure 4.16: Single 8-bit Input Switch Datapath Multiplexer

The new scratch path values are selected at the byte level by 2-to-1 multiplexers, selecting between current datapath inputs and current scratch path inputs. Thus, data on the scratch path can be kept there or replaced with new datapath data. The scratch multiplexer is shown in Figure 4.17.

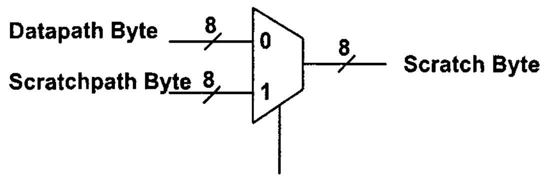


Figure 4.17: Single 8-bit Input Switch Scratch Path Multiplexer

The overall input switch requires 16 8-bit 4-to-1 multiplexers to switch the data path, and 16 8-bit 2-to-1 multiplexers to switch the scratch path. Thus, overall, it takes four 128-bit inputs (datapath, scratch path, constant value, and key value) and produces two 128-bit outputs (datapath and scratch path) where each datapath output byte is selected between corresponding bytes of the four inputs, and where each scratch path output byte is selected between corresponding bytes of the datapath and scratch inputs. This is shown in Figure 4.18. Note that the control lines to the multiplexers are naturally driven by the configuration data.

4.2.2 Byte Reordering

In the algorithm survey, byte permutations were identified as a common operation among many of the algorithms. Such an operation can be considered a simple rerouting of data, and as such its implementation is considered here. Note that byte reordering is not just useful for explicit byte permutation operations, but also for when portions of the input data need to by processed by other parts of the datapath hardware.

Byte reordering is implemented using an 8-bit 16-to-1 multiplexer to select which byte of the 128-bit input gets switched onto a particular byte of the output. This is illustrated in Figure 4.19.

The overall byte reordering component consists of 16 such multiplexers in parallel, as shown in Figure 4.20. Note that this arrangement allows duplication of output bytes, since more than one of the multiplexers may select the same input byte to switch to the output. That is why this component is referred to as a byte reordering component rather than strictly as a byte permutation component.

This component is meant to be instanced at each of the basic component outputs, to allow reordering of the datapath values before data reaches the next input switch. The multiplexer control lines are driven by the configuration data.

A conceptual flaw exists in the design of the byte reordering multiplexer, in that the 4-bit control signal for the multiplexer selects the input byte corresponding to

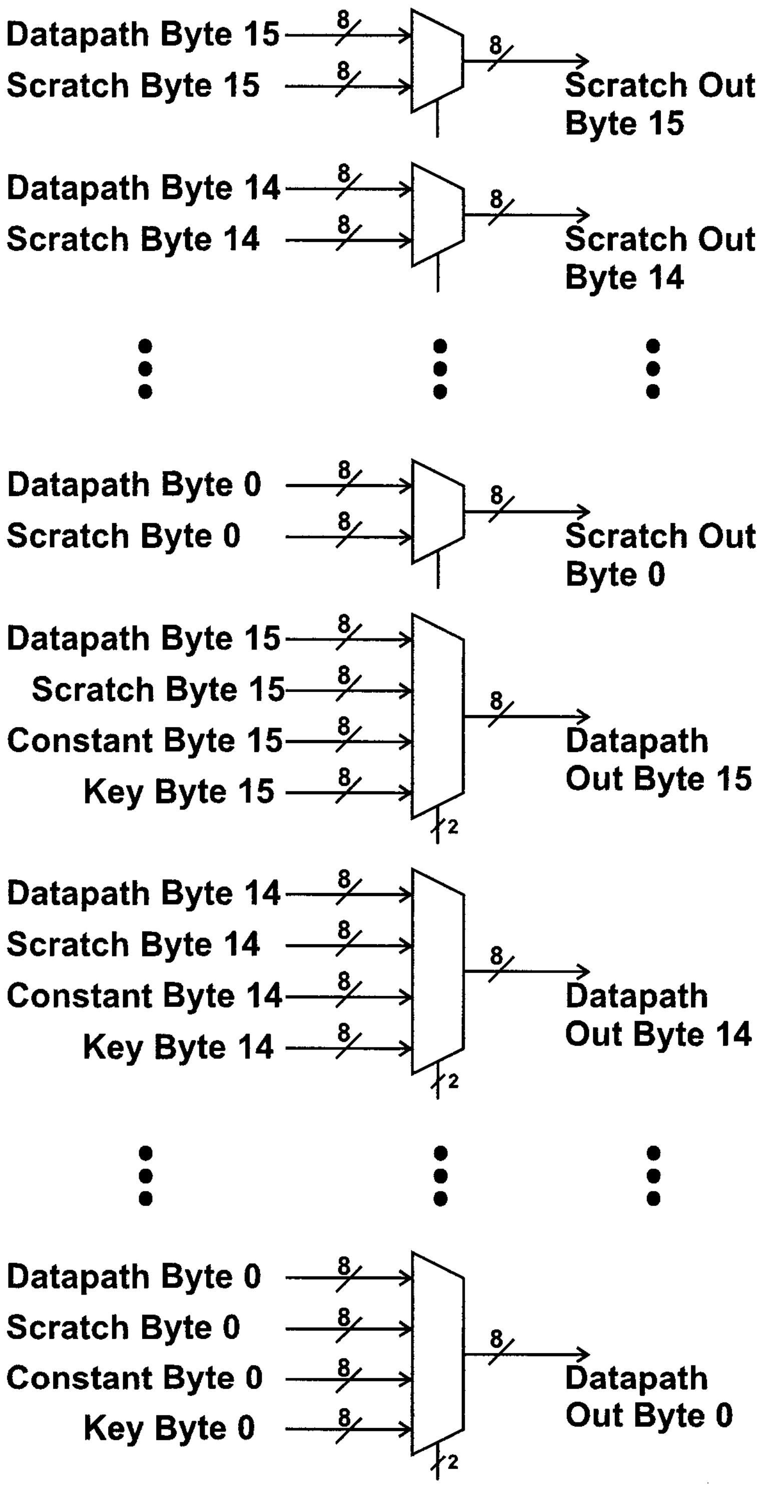


Figure 4.18: Overall Input Switch

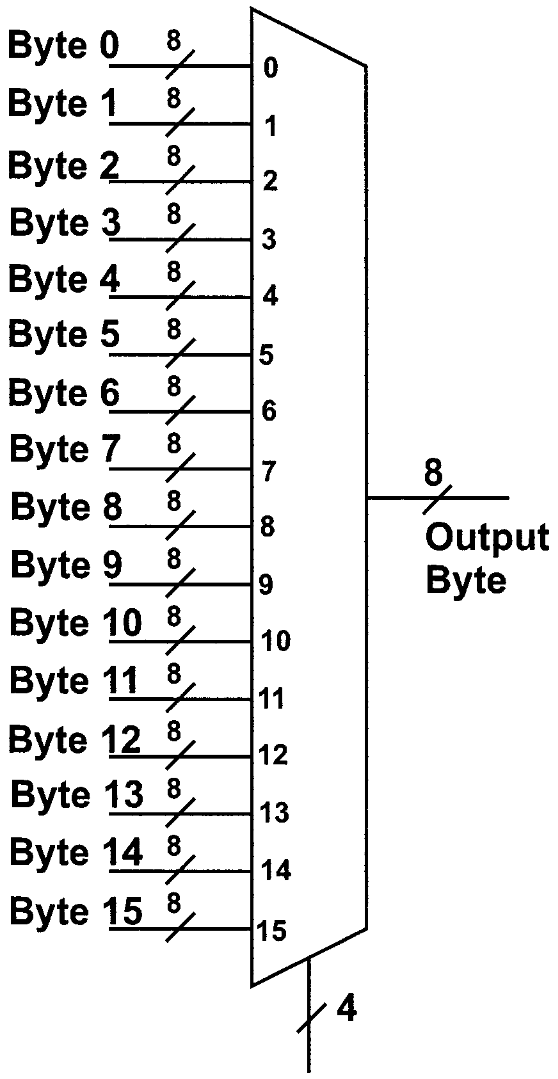


Figure 4.19: 8-bit 16-to-1 Multiplexer for Byte Reordering

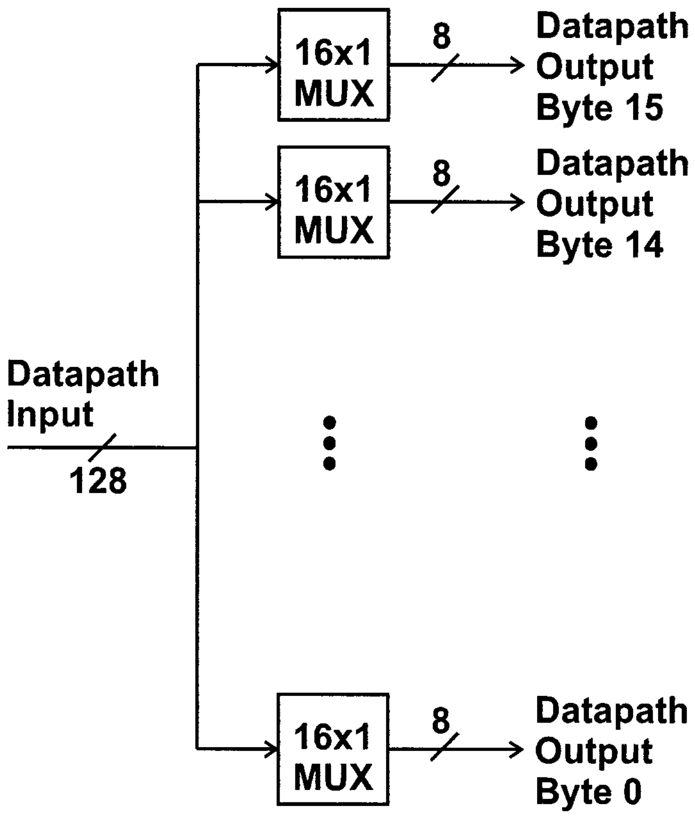


Figure 4.20: Overall Byte Reordering Component

its value. That means that, in the default mode, when all configuration values are zero, all sixteen of the byte reordering multiplexers will select input byte 0 onto their outputs, rather than each multiplexer passing through its corresponding input. This negates the desired bypass functionality for the zero configuration. However, since there was no logical mapping of the input bytes to the multiplexer ports that made sense other than each byte going into its correspondingly-numbered port, it was decided to leave the multiplexers as they are and enforce bypass configuration of the byte reordering component in the software configuration utility.

4.3 Configurability

The fact that each component is configurable has been mentioned frequently. Most of the configuration of the components mentioned in preceding sections is the result of using multiplexers to select and control routing of data to the components. The control lines for these multiplexers are driven by configuration data.

Also, the LUTs can be configured to hold arbitrary data, and arbitrary constant values can be provided to every component for selection by the input switch. These kinds of data are part of the configuration data as well.

Thus, the configuration data for a component is a bit sequence containing control signals and data values. All values of the configuration data are needed at all times, so storing them in memory is infeasible. Thus, the configuration data must be stored in a flip-flop or register.

The configuration register is a generic construct of n bits, where n is the number of bits of configuration data required. It is essentially a shift register. Before system operation, when the system is in configuration mode, the configuration bits are shifted into the system serially, until the entire configuration register is filled. Once configuration is complete, the values are held in the register, and are available thereafter unless the system is reset or goes back into configuration mode.

An example of a configuration register is shown in Figure 4.21. It outputs all of its data in parallel, but only loads data serially, on the positive edge of the configuration clock signal, and only when the configuration mode control signal is asserted.

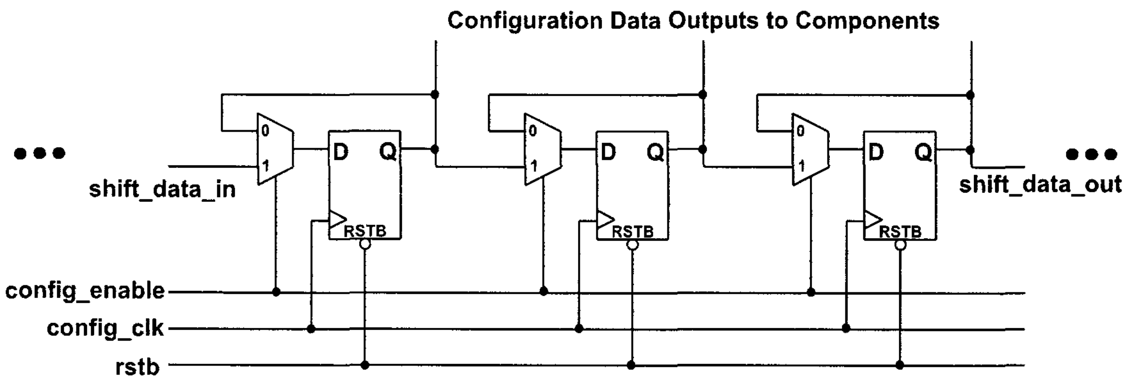


Figure 4.21: General Configuration Register Circuit

The configuration register will be cascadeable, so that the configuration register for one component will connect to the next component, and so on, so that the entire system configuration can be viewed as a concatenation of individual component configurations.

Note that the clock signal driving the configuration register is not the same as the overall system clock. That is because when in configuration mode, the only relevant behaviour or the system is the shifting of configuration data, which can be done at a much higher clock rate than regular operation due to the close proximity of flip-flop elements. Thus, we can use a faster configuration clock and configure the device more quickly. When not in configuration mode, the value is held on the flip-flop output, so the disparity between the configuration clock and the system clock is irrelevant. This is very similar to the concept of a scan chain for production testing of ASICs.

Depending on what component is being configured, the configuration register will be of different sizes. In subsequent diagrams, configuration registers will be represented as shown in Figure 4.22, with the bit size specified.

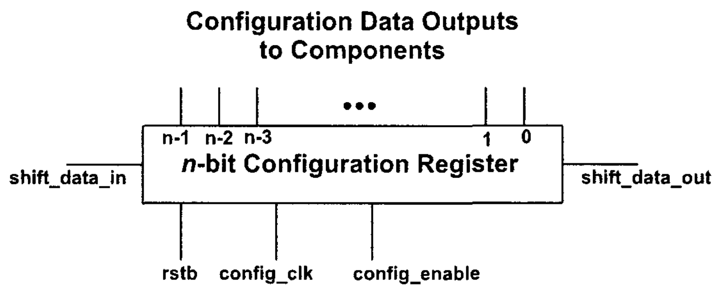

Configuration Data Outputs to Components
n-1 n-2 n-3
1 0
n-bit Configuration Register
shift_data_in
shift_data_out
rstb config_clk config_enable

Figure 4.22: Configuration Register Block Diagram

Sample VHDL code implementing a 256-bit configuration register can be found in Appendix A, Section A.4, on page 199. Other configuration register sizes are discussed in the following section.

Note that the flip-flops in the LUT component implementation are connected to each other like a configuration register. Thus, the LUT component acts as its own configuration register, and the LUT data is shifted in as part of the system configuration.

4.4 Basic Operational Building Block

As mentioned in preceding sections, the input switch is meant to be instanced before every computational component, and the byte reordering after. It makes sense, then, to group the input switch, specific component, byte reordering, and their corresponding configuration register into a SLAB. This will make higher level system design much easier.

Naturally, there will be six different SLABs, one for each basic component. Thus, each SLAB will have a different size of configuration register. For the Boolean, Add/-Sub, Shifter, and Multiplier SLABs, the general structure will be as shown in Figure 4.23. Note that any bit signal not labeled is assumed to be 128 bits, and clock, reset, and enable signals are not shown.

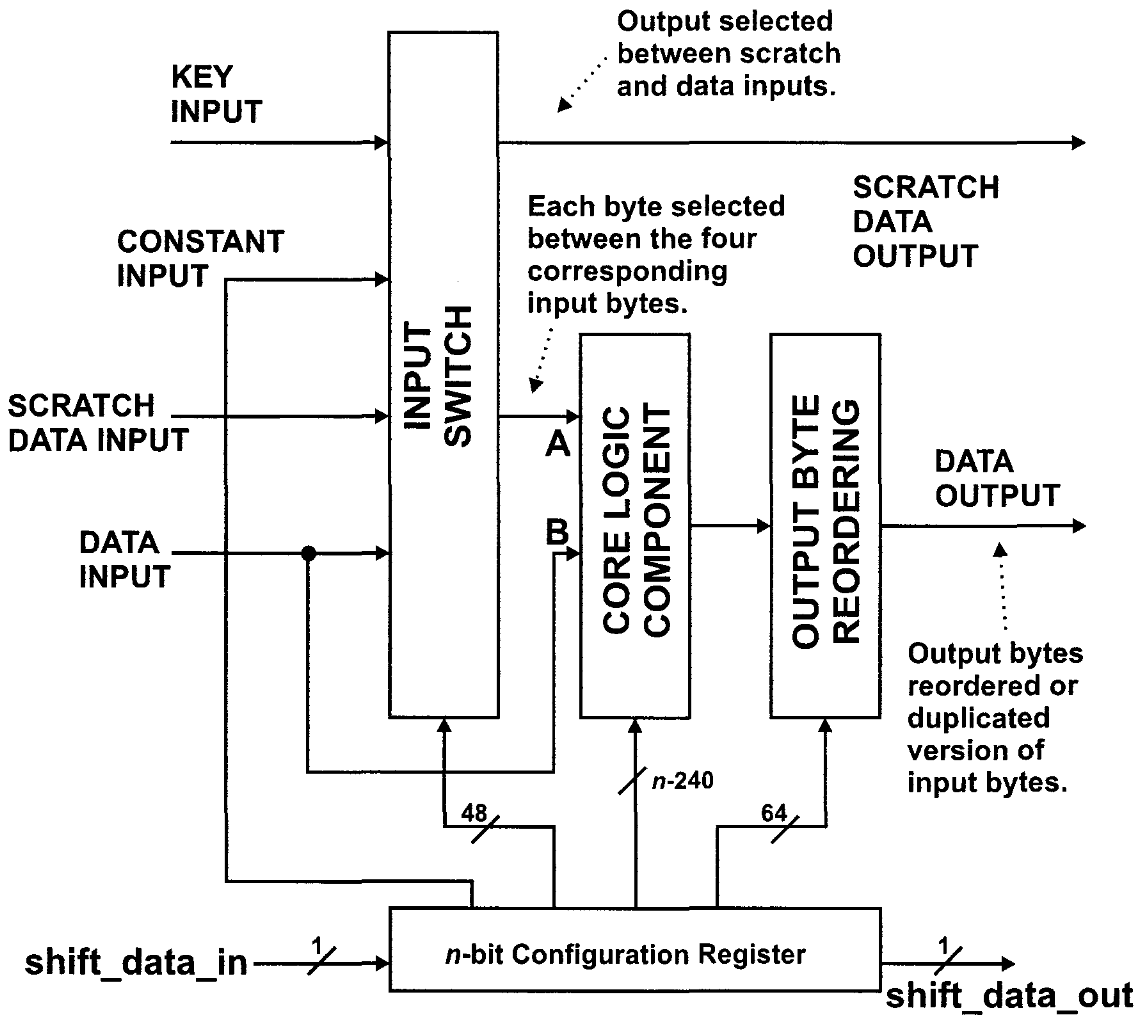

KEY INPUT
CONSTANT INPUT
SCRATCH DATA INPUT
DATA INPUT
INPUT SWITCH
Output selected between scratch and data inputs.
SCRATCH DATA OUTPUT
Each byte selected between the four corresponding input bytes.
A
B
CORE LOGIC COMPONENT
OUTPUT BYTE REORDERING
DATA OUTPUT
Output bytes reordered or duplicated version of input bytes.
48
n-240
64
shift_data_in
1
n-bit Configuration Register
1
shift_data_out

Figure 4.23: Binary Operation SLAB

The SLAB has two primary 128-bit inputs (datapath and scratch data) plus configuration register I/O and a 128-bit key input, and two 128-bit outputs (again, datapath and scratch data). Internally, note that the input switch takes four 128-bit inputs (datapath, scratch path, constant data from the configuration register, and key data) plus control signals from the configuration register, and it generates two 128-bit outputs, the scratch output (which goes directly to the SLAB scratch output) and the data output (which feeds into the primary input of the core operation).

The second input to the core operation is directly connected from the datapath input of the SLAB. Thus, only one of the two operands of these operations is switched by the input switch. This still allows sufficient flexibility to meet the design requirements.

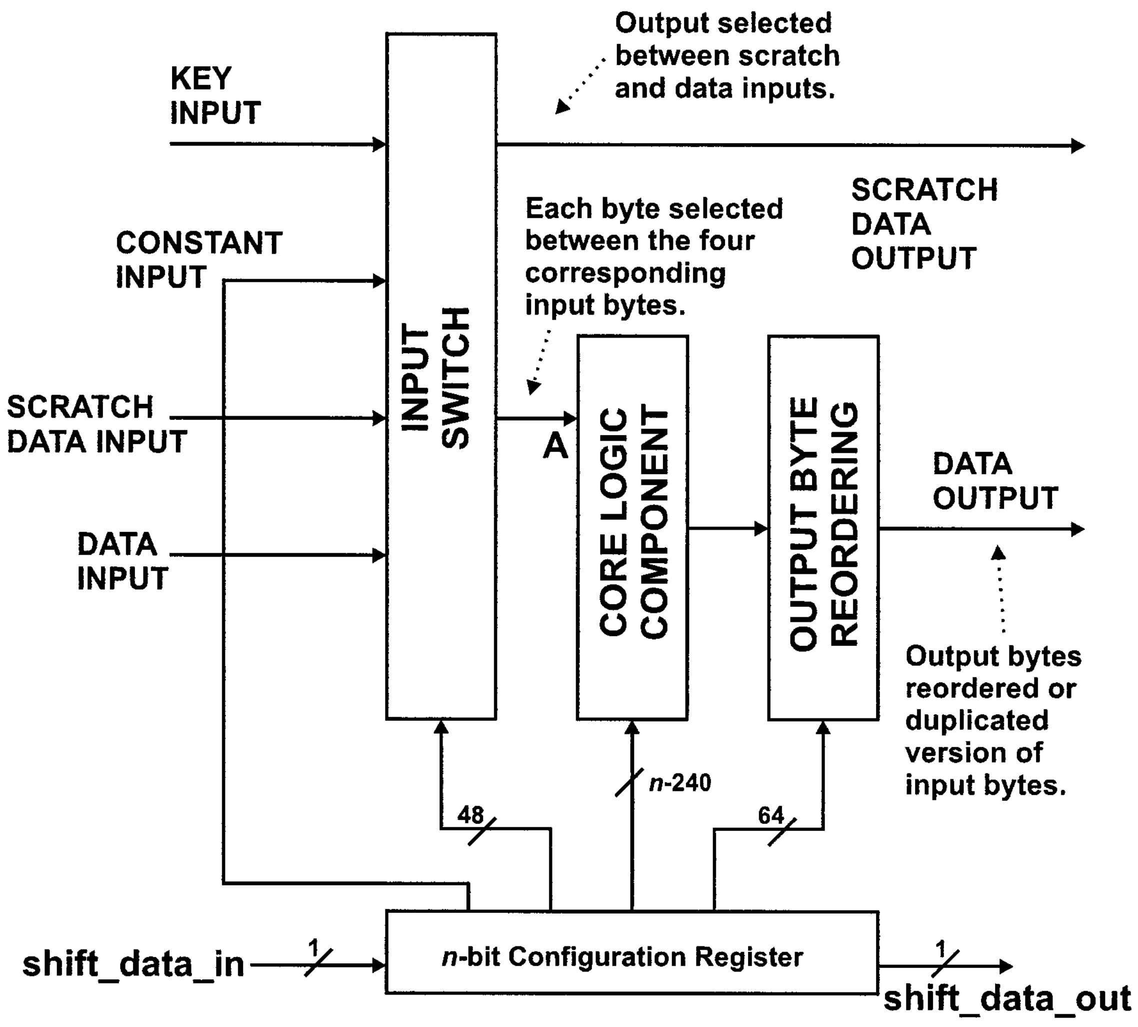
The core operations have two 128-bit inputs plus control signal inputs driven by the configuration register. Details of these core operations have been discussed in prior sections. The core operation produces a single 128-bit output which is fed into the byte reordering component (which is controlled by the configuration register). The 128-bit output of the byte reordering component is connected directly to the SLAB output.

The LUT and XORnet SLABs are slightly different in that they are unary operations, only having a single 128-bit input. Thus, there is no second input to the core operation, and so the only input is fully switched by the input switch. This arrangement is shown in Figure 4.24. Note that any bit signal not labeled is assumed to be 128 bits, and clock, reset, and enable signals are not shown.

4.5 Configuration Formats of Basic SLABs

Each of the basic SLABs will have different sizes of configuration register since each operation works different. However, there are some commonalities.

It was previously mentioned that each component has a constant value associated with it, that is selected by the input switch. This 128-bit constant value is stored in the configuration register as the 128 Most Significant Bits (MSBs). Immediately



Output selected between scratch and data inputs.
KEY INPUT
CONSTANT INPUT
SCRATCH DATA INPUT
DATA INPUT
INPUT SWITCH
Each byte selected between the four corresponding input bytes.
SCRATCH DATA OUTPUT
A
CORE LOGIC COMPONENT
OUTPUT BYTE REORDERING
DATA OUTPUT
Output bytes reordered or duplicated version of input bytes.
48
n-240
64
shift_data_in
1
n-bit Configuration Register
1
shift_data_out

Figure 4.24: Unary Operation SLAB

after the constant value, the 48 control bits for the input switch are stored. After the input switch control bits, the 64 control bits for the byte reordering are stored. Then, finally, the specific control bits for the core component are stored. Thus, the 240 MSBs of each configuration register hold the same function, whereas the remaining Least Significant Bits (LSBs) vary according to the specific operation.

4.5.1 Boolean SLAB

The Boolean SLAB configuration register is 288 bits in size, meaning the 48 LSBs control the core Boolean component. Those bits are split into 16 groups of three, the groups corresponding to each of the 16 byte-operations. Two of the 3 bits in each group select the operation; the third bit controls bypass. This format is more accurately described in Appendix B, Section B.1 on page 200.

4.5.2 Shifter SLAB

The Shifter SLAB configuration register is 252 bits in size, meaning the 12 LSBs control the core Shifter component. Those bits are split into 4 groups of 3 bits each, the groups corresponding to each of the 32-bit shifters. Those bits control left/right, rotate/shift, and bypass operation. This format is more accurately described in Appendix B, Section B.2 on page 200.

4.5.3 Add/Sub SLAB

The Add/Sub SLAB configuration register is 324 bits in size, meaning the 84 LSBs control the core Add/Sub component. The 84 bits are cut in half, each controlling one of the adder banks. In each 42 bit control vector, 2 bits are set aside for each of the 8 adders to control add/subtract and connect/not connect, and the remaining bits control the input multiplexers on the adders. This format is more accurately described in Appendix B, Section B.3 on page 201.

4.5.4 Multiplier SLAB

The Multiplier SLAB configuration register is 244 bits in size, meaning the 4 LSBs control the core Multiplier component. Each of the 4 bits controls the bypass functionality of one of the four 32-bit multipliers. This format is more accurately described in Appendix B, Section B.4 on page 202.

4.5.5 LUT SLAB

The LUT SLAB is somewhat different from the others, because a large part of its configuration is stored within the flip-flops of the LUT components themselves, which can function as configuration registers. Thus, the SLAB structure for the LUT does not quite match that in Figure 4.24, since the configuration register, rather than connect directly to the configuration output of the SLAB, must connect to the configuration input of the LUT core, and the core's configuration output connects to the configuration output of the SLAB. This is illustrated in Figure 4.25.

The configuration register in this SLAB is 256 bits in size, but the overall configuration for this component is 33024 bits in size. In the configuration register itself, the 16 LSBs control the core LUT component, each of the 16 bits controlling the bypass functionality of one of the 8×8 LUTs. The other 32768 bits of configuration data are stored directly into the flip-flops of the LUT.

This format is more accurately described in Appendix B, Section B.5 on page 202.

4.5.6 XORnet SLAB

The XORnet SLAB configuration register is 4341 bits in size, meaning the 4101 LSBs control the core XORnet component. Of those 4101 bits, 4096 are the enable

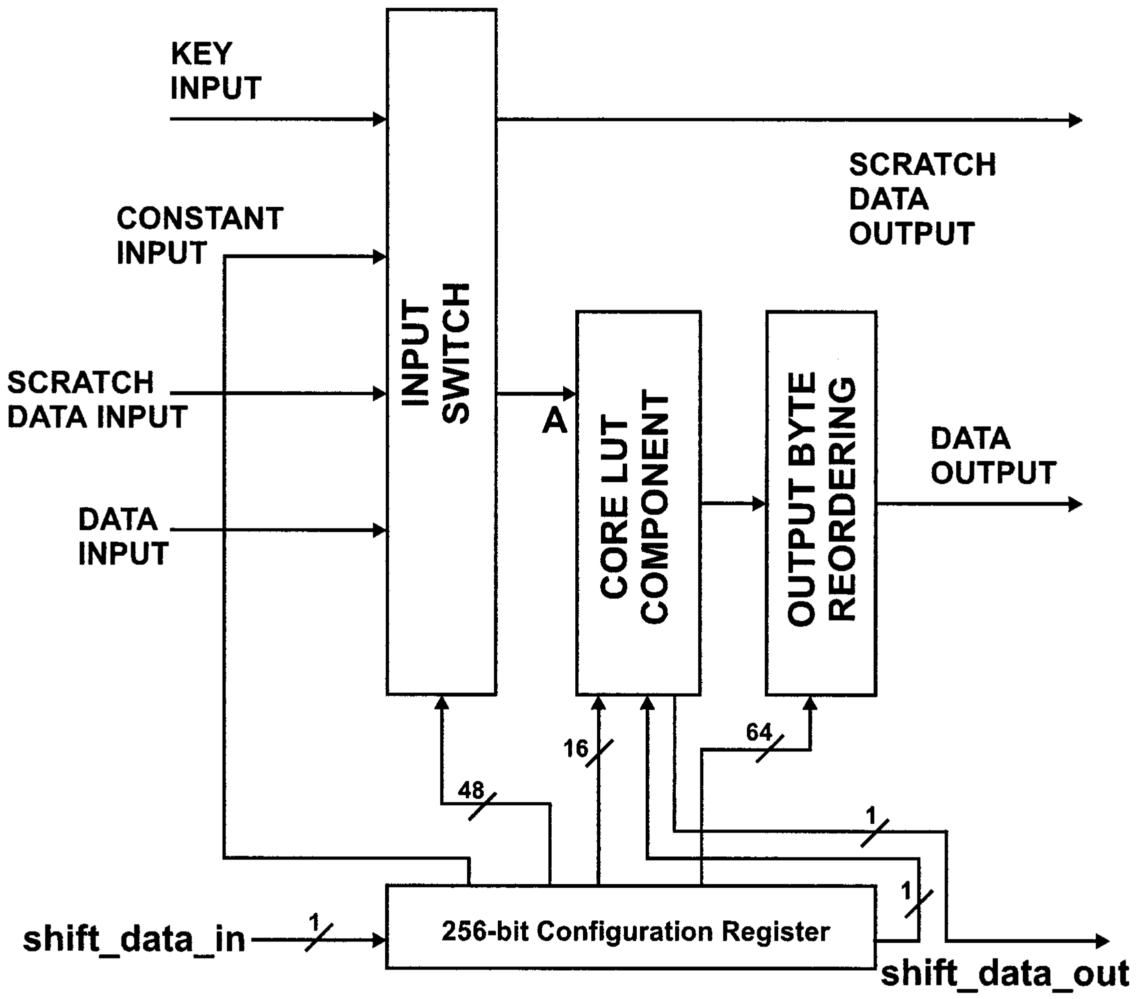


Figure 4.25: LUT SLAB

signals for the four 32-bit XORnets, with 1024 bits controlling each XORnet. Of the remaining 5 control bits, 1 bit selects whether the XORnet component is operating in 32-bit mode or 64-bit mode, and the other 4 bits control the bypass for each of the four 32-bit XORnets. This format is more accurately described in Appendix B, Section B.6 on page 203.

4.6 Summary of Component Implementation

This chapter has discussed the design of hardware components to implement the basic operations identified by the algorithm survey in the previous chapter. The six components described in this chapter – the Boolean SLAB, Shifter SLAB, Add/Sub SLAB, Multiplier SLAB, LUT SLAB, and XORnet SLAB – not only the desired operations, but also are configurable to handle different operand sizes and dataflow options, thanks to the incorporation of the input switch and byte reordering components. These basic components form the basis of the overall cryptographic hardware system described in the next chapter.

Chapter 5

System Architecture

In the previous chapter, designs for configurable cryptographic hardware operations were presented, providing a basis from which to build a reconfigurable cryptographic hardware module. The arrangement of those components into a working, flexible system is a non-trivial problem.

The design of the overall system architecture is influenced by the structure of the algorithms under consideration. Those algorithms, despite having vastly different organization, are all round-oriented, iterating through the same operations over and over again to provide security. Furthermore, within each round, operations are generally sequential. The system architecture must be built with this in mind.

Thus, it was decided to use a coarse-grained reconfigurable solution, with each processing element (PE) designed to be able to implement a single round of most of the algorithms considered. Since AES would likely be the most common algorithm implemented, the proof-of-concept system presented here will have ten PEs in order to implement a fully-pipelined, loop-unrolled version of AES. Algorithms requiring more rounds will therefore require iterative implementation, and thus the system architecture must accommodate that as well.

This general architecture depends almost entirely on the flexibility of the PE, which must be able to implement one round of almost all of the algorithms desired.

The arrangement of the PEs is also a factor in system flexibility. In any real application, block ciphers are often used in modes other than ECB mode, and thus providing system support for different modes of operation is highly desirable. Since pipelining provides no benefit to most block cipher modes other than ECB, even algorithms that *could* be pipelined might be better suited to iterative implementation.

The key advantage to iterative implementation is that it improves resource usage, especially when block ciphers are used in CBC, OFB, and CFB modes. In each of those modes, encryption (or decryption) of a block of data is dependent upon the result of the encryption of the preceding block. In other words, one block must be completely processed for the next block can be processed. In pipelined implementations, this means that the pipeline must be stalled until the first block is encrypted, and then the result can be used to start processing the next block. This leads to most of the active hardware in the pipeline processing invalid data most of the time.

A more efficient use of resources in such scenarios is to iterate the data through the same round hardware (in our case, the PE). There will be little difference in performance since subsequent blocks must wait for the current block to finish in any case, but much less hardware will be used. For the particular system under design, that means that the other PEs are free to use in other ways, such as implementing other algorithms in parallel which will allow an increase in overall system throughput. Parallel implementation will be supported through the addressing scheme, so that data can be written to one PE, and while that data is being iteratively processed, a separate, unrelated block of data (perhaps from a separate communication channel) could be written to a different PE for processing. Thus, support for iteration allows maximal use of hardware resources and greater aggregate throughput, though individual messages will see little speedup.

Consequently, the proof-of-concept system developed in the rest of this chapter has been designed with two primary goals in mind.

- The processing elements must be capable of implementing a single round of most algorithms.
- The system must offer support for pipelining and iteration, as well as different modes of operation.

These design goals raise several issues, which are explored throughout the rest of the chapter.

5.1 Processing Element

The processing element that forms the basis of the overall system is called the Processing Element with Functional Units and Networked Connections (PE-FUNC), or more commonly, just the PE. It is intended to be built from from the SLABs described in Section 4.4, and to be able to implement a single round of most of the algorithms under consideration. Thus, the design of the PE-FUNC must be guided by the nature of the algorithms themselves.

Table 5.1 shows the rounds of each algorithm broken down into a sequential list of the basic operations that are used. From this table, several important characteristics become evident.

First and foremost, Boolean operations are used frequently in many different orders within the rounds of almost all of the algorithms. Their high rate of occurrence and extremely variable positioning with the rounds suggests that there must be a large number of Boolean SLABs spread throughout the PE.

Secondly, add/subtract and shifter operations are also used often, albeit with less frequency than basic Boolean operations. They are sometimes used in sequence, with the shift preceding the addition/subtraction. Thus, it is necessary to provide a number of shifter and add/subtract components, spread throughout the PE. The number of add/subtract and shifter components would be significantly less than the

AES	**Camellia**	**RC6**	**SAFER++$_{128}$**	**SHA-1**	**DES**
LUTs	Boolean	Shifter	Boolean	*during initialization*	XORnet
Byte Reorder	LUTs	Add/Sub	Add/Sub	Boolean ($\times$3)	Boolean
XORnet	XORnet	Multiply	LUTs	Shifter	LUTs
Boolean	Boolean	Shifter	Boolean	*during 80 rounds*	XORnet
	and on every 6th round	Boolean	Add/Sub	Shifter	
	Boolean	Shifter	Byte Reorder	Boolean (3 to 5 times)	
	Shifter	Add/Sub	Add/Sub	Add/Sub ($\times$2)	
	Boolean ($\times$6)	Byte Reorder	Byte Reorder		
	Shifter		Add/Sub		
	Boolean				

Table 5.1: Algorithm Round Breakdown into Sequence of Basic Operations

number of Boolean components due to their relatively less common occurrence and variability of positioning.

Lastly, the three "big" operations (multiplication, LUT, and XORnet) are used infrequently, generally only once in a given round. Thus, if the rest of the PE is sufficiently flexible, providing a single component for each of these operations should be sufficient to meet the needs of the system.

5.1.1 Organization of Components

A sequence of 29 SLABs was determined to be sufficient to meet the needs of most of the algorithms under consideration, while still providing flexibility to implement other algorithms. Note that the use of 29 SLABs results in extensive overprovision of basic components since most algorithms need only four to six basic operations in a given round. This overprovision provides great flexibility and avoids the need for any complex data switching within the PE.

Each PE-FUNC has one LUT SLAB, one Multiplier SLAB, one XORnet SLAB, four Shifter SLABs, four Add/Sub SLABs, and 18 Boolean SLABs, arranged as shown in Figure 5.1. Clocks, enable signals, and input/output registers are not shown.

In Section 4.2, the scratch path was introduced to the SLAB in parallel to the datapath. Both can be clearly seen in Figure 5.1. The datapath values obviously come from the datapath input of the PE-FUNC itself, but the scratch path value must be initially introduced into the first SLAB in the PE. Thus, a 4-to-1 multiplexer allows the user to configure the initial scratch path data between a copy of the datapath, a constant value from the PE-FUNC configuration register, or one of two possible key values. The key values come from the key memory, which is discussed in the following section.

The inputs and outputs of the PE-FUNC are registered, but in the proof-of-concept implementation there is no pipelining inside the PE-FUNC. It has a 129-bit

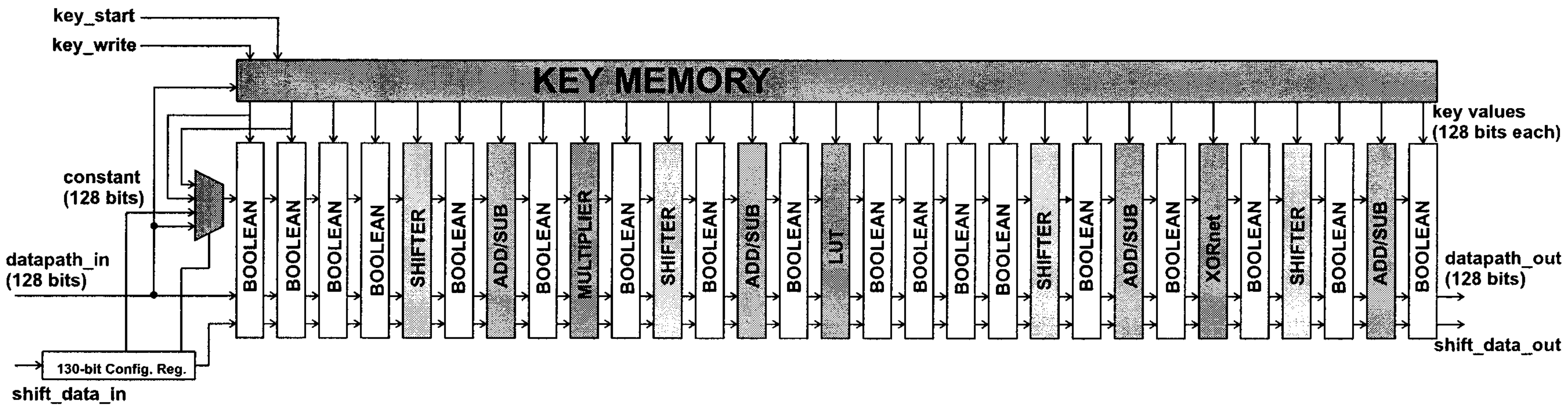


Figure 5.1: Processing Element Component Organization

datapath input (the 129th bit is asserted when valid data is being input), key memory write and key memory start control signals, configuration register inputs and outputs that are connected directly to the configuration registers internal to the PE-FUNC, and a 129-bit datapath output. This I/O layout is illustrated in Figure 5.2. Note that the valid bit is directly passed from the input register to the output register, rather than through the processing logic, which is why it does not appear in Figure 5.1.

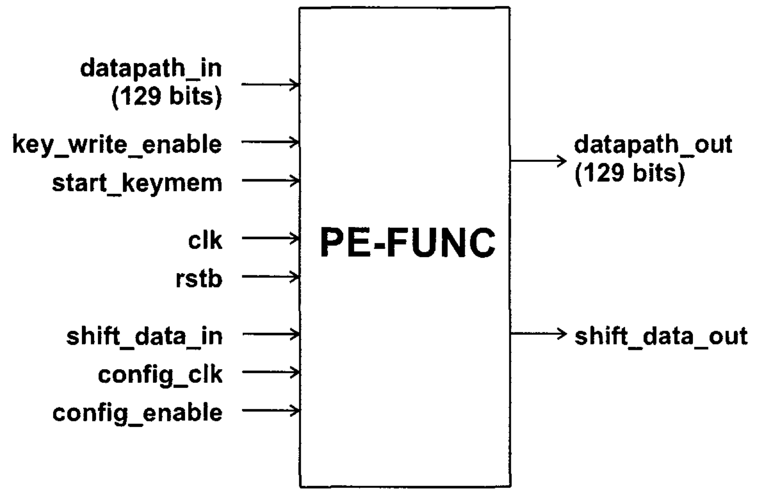


Figure 5.2: Processing Element Input/Output Signals

This arrangement of SLABs, in conjunction with the key memory discussed in the next section, allows great flexibility.

5.1.2 Key Memory Design and Operation

Management of the subkeys of a cryptographic is a major concern. As seen in Chapter 3, the key schedules of cryptographic algorithms can frequently be more complex that the algorithms themselves. This is usually mitigated by the fact that a single key is usually used to encrypt many blocks of data, and thus the computational costs of computing the subkeys is amortized over many encryptions. In other words, it can be done up front, and once finished, does not slow processing down any further.

In keeping with this ideology, and to simplify technical matters, the SHERIF cryptographic hardware module will not readily support implementation of key schedules. Rather, it requires that the key schedules be pre-computed by an external processor, and the resulting subkeys must then be loaded in to the PEs. These subkeys are loaded into a special component in each PE-FUNC called the key memory.

Some of the algorithms use 32-bit subkeys; others full 128-bit subkeys. Many of the algorithms require multiple subkeys in each round. Thus, the key memory must be capable of providing at least two different 128-bit keys to the SLABs. Therefore the key memory has an independent output to connect to each SLAB's key input. Each of those outputs can be configured to provide one of the two key values. The key memory also provides those two key values to the scratch path input multiplexer mentioned above.

Such a component would be straightforward in design, if not for the fact that the overall system is intended to support both pipelined, loop-unrolled implementations of cryptographic algorithms, as well as iterative implementations. Thus, the key memory must be capable of providing different subkey values during different clock cycles (but still no more than two 128-bit values at a time).

A comparison of the the subkey requirements of the algorithms under consideration is shown in Table 5.2. Note that SHA-1 [52] is excluded since it is not a keyed hash function. A key memory size of 1664 bits is selected, allowing up to 13 128-bit subkeys to be available to a given PE. This means that algorithms like SAFER++$_{128}$ [54] cannot be implemented in a single PE, but in the case of SAFER++$_{128}$, a single round would likely require two PEs anyway, due to the sequence in which the adders are used.

This problem is solved by implementing the key memory as a large shift register,

Algorithms	Number/Size of Keys	Approx. Freq. of Keys	Total Bits
AES	11 128-bit round keys	1 per round	1408
Camellia	26 64-bit round keys	1 per round (sometimes 3)	1664
RC6	44 32-bit round keys	2 per round	1408
SAFER++$_{128}$	15 128-bit round keys	2 per round	1920
DES	16 48-bit round keys	1 per round	768

Table 5.2: Algorithm Subkey Requirements

as shown in Figure 5.3.

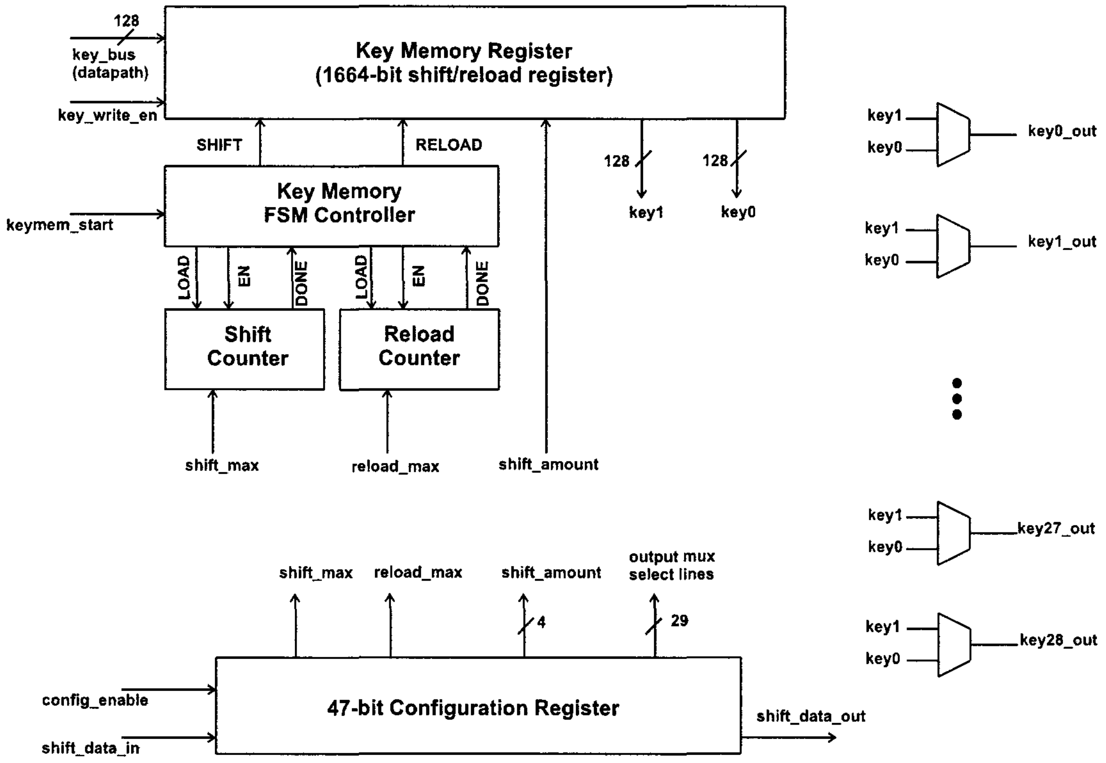


Figure 5.3: Key Memory Architecture

When the key memory write signal is asserted, the key memory assumes the datapath input to the PE-FUNC is subkey data, and writes that 128-bits into the MSB end of the shift register after shifting the current contents down by 128-bits. Thus, 13 clock cycles are needed to fill the key memory. Also, note that the output of the key memory is taken from the LSB end of the shift register. That means that even if only one 128-bit key is needed in the key memory, it still takes 13 clock cycles to shift it into the right position.

The two key values available at any given time come from the least significant 256-bits of the key memory register. Each of the 29 SLAB outputs multiplexes between the two, according to the configuration set in the configuration register.

To support iterative implementations, the key memory must be able to shift new values into the output position during operation. This shifting is controlled by a state machine. When in the idle state, the state machine prevents the key memory from shifting. However, if it receives a key memory start signal assertion, the state machine begins operation and waits a number of clock cycles set by the user (up to 127), then shifts the register values right (by some multiple of 32-bit words, up to 8, with the shift amount being configured by the user), allowing new subkey values to enter the output range of the shift register. The number of clock cycles between shifts is determined by a shift counter that resets at each shift. Another counter increments every time a shift occurs, essentially counting the rounds of the algorithm, and when it reaches its limit as configured by the user, the state machine reverts to idle mode and resets the contents of the shift register. The instant reset is possible through the use of a non-shifting 1664-bit register in parallel with the shifting register, that simply holds the original value of the key memory for use in resets.

The counters are used to account for the latency introduced by the registers at the input and output of the PE. The presence of those registers means that, in an iterative algorithm implementation, each iteration takes multiple clock cycles, and thus the subkey data for each iteration must be synchronized with its proper round.

This design of the key memory allows the user to configure its operation to accommodate pipelined algorithm implementations (which generally require no shifting behaviour of the key memory), single-PE iterative implementations (which require new subkeys every round), and multiple-PE iterative implementations, where the algorithm is implemented in two or more PEs, and must iterate through them. In this last case, each PE's key memory only needs to hold part of the key schedule.

5.2 Processing Fabric

The arrangement of PEs into a processing fabric was also a point of concern. Several different approaches were considered, including a 2D array as discussed in [61]. The array structure was abandoned due to difficulties in coordinating data transfers between adjacent PEs. A linear, sequential arrangement of PEs was selected since it mimics the linear ordering of the rounds of the algorithms being considered. The key drawback to this arrangement is that the system can only handle data blocks of 128 bits, so implementations of most hash functions like SHA-1 are infeasible.

As described in previous sections, the PEs have very simple I/O, and while the key memories of each PE are capable of handling iterative algorithm implementations, nothing else about the PEs themselves handles feedback or iteration. Thus, Routing Nodes (RNs) are implemented between all of the PEs, as shown in Figure 5.4.

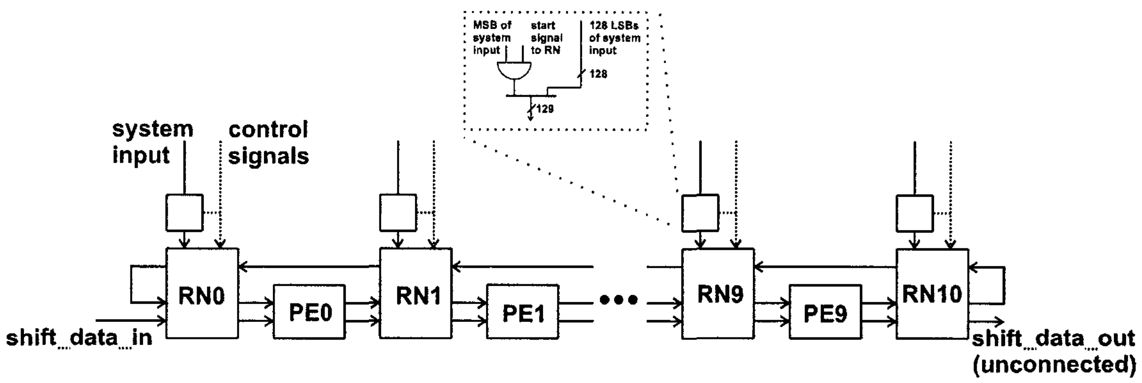


Figure 5.4: SHERIF Processing Fabric

The routing nodes control the flow of data within the system, routing data from the system input to the PEs, and from the PEs to the system output or to other PEs via the datapath or feedback signals. The routing nodes are key to completing the functionality of the SHERIF system, since they handle all of the data routing and storage needs that the PE-FUNCs do not.

5.2.1 Routing Nodes

The routing nodes would seem to have a simple task in routing the data between the various PEs, but it turned out that coordinating the data flow was a very difficult task. The design of the PE-FUNC was kept simple, but that off-loaded responsibility for control and other complex tasks to the routing node.

The basic responsibility of the routing node is to switch or route data between the PEs, the system input, and the system output. However, it must be sufficiently configurable to handle both pipelined and iterative implementations. Furthermore, if the SHERIF system is to support different modes of operation (see Section 1.3.2) for its block cipher implementations, then the routing nodes must handle this as well, since all of the modes are defined as being external to the ciphers themselves [3]. Other functionality off-loaded from the PE-FUNC includes selection of the key memory start signal, and controlling key writes.

Design and Operation of the Routing Node

Figure 5.5 shows the basic interface of the routing node. It has three inputs (datapath in, feedback in, and system in) and three corresponding outputs (datapath out, feedback out, and system out). The three outputs are switched independently of each other. It has a number of control signal inputs to indicate the type of the current system input data: key write assertion means that a key value is being written to the key memory of the following PE, IV write means that an IV is being written into the routing node, and stream write means that the data is part of an ongoing encryption.

The implications of these signals will be discussed later.

There is also an 11-bit start vector input, driven by the address decoder at the system level. The individual bits of this correspond to the individual start signals generated by the address decoder when data is addressed to particular routing nodes. Those signals are used at the top level to set the valid bit (MSB) of the system input for each routing node, but all of the signals are available to each routing node to allow the internal state machines to start when data is addressed to *any* of the routing nodes, if so desired.

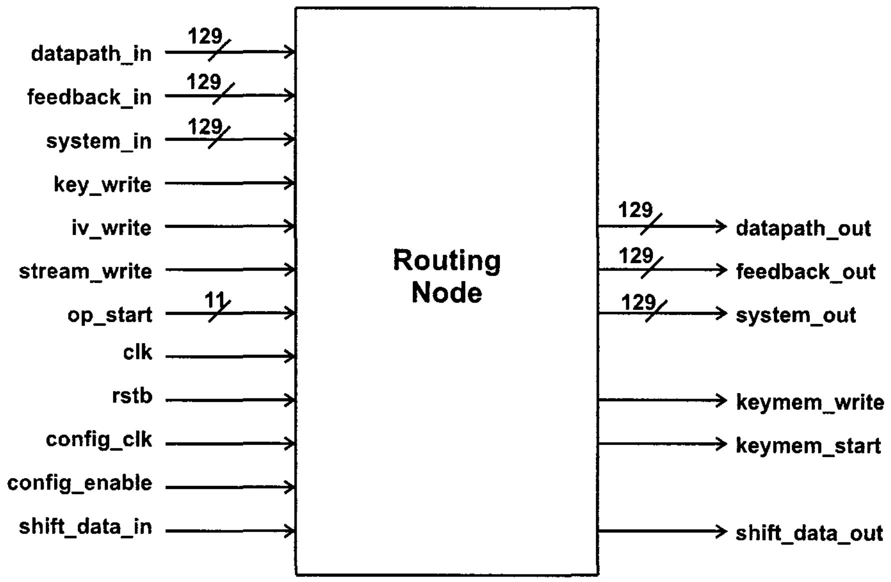


Figure 5.5: Routing Node I/O Interface

The routing node is configurable, so it has the same standard configuration register interface as well. Also, apart from the data output described above, there are two output control signals (key write and key start) which connect to the key memory control signals in the following PE-FUNC. At this point, many of the top level routing node control signals seem extraneous. Their function will become apparent as the implementation of the routing node is discussed in detail.

Figure 5.6 shows the architectural details of the routing node. Note that the external I/O in this figure has been rearranged so that all the inputs are on the left and all the output are on the right, in order to make the diagram more understandable. Also, many of the control and configuration signals are not shown to preserve clarity. Details of the routing node configuration register are given in Figure 5.7.

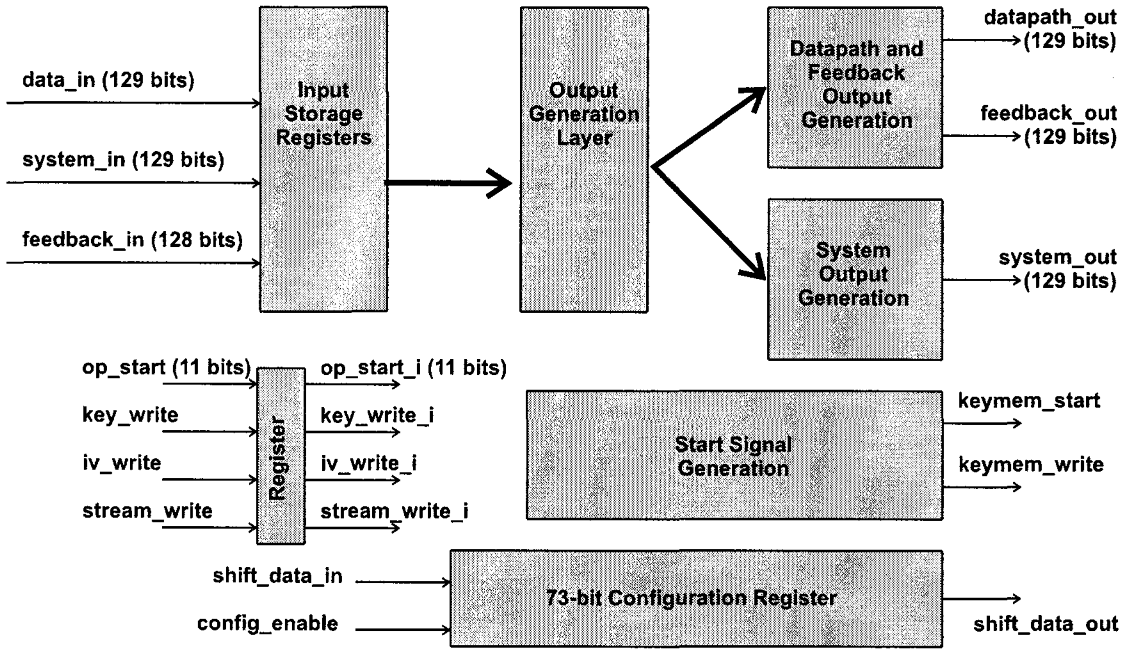


Figure 5.6: Top Level Routing Node Implementation

The data inputs to the routing node are registered immediately, although there are two extra registers for storing special data as well. This is illustrated in Figure 5.8, and discussed in detail later.

Each of the routing node's data outputs (datapath, feedback, and system) are selected independently by 8-to-1 multiplexers, meaning each output can be one of eight possible options despite the fact that there are only three data inputs. These extra values are available to support different modes of operation; beyond outputting the three data inputs directly, the routing node can also output the XOR of any of

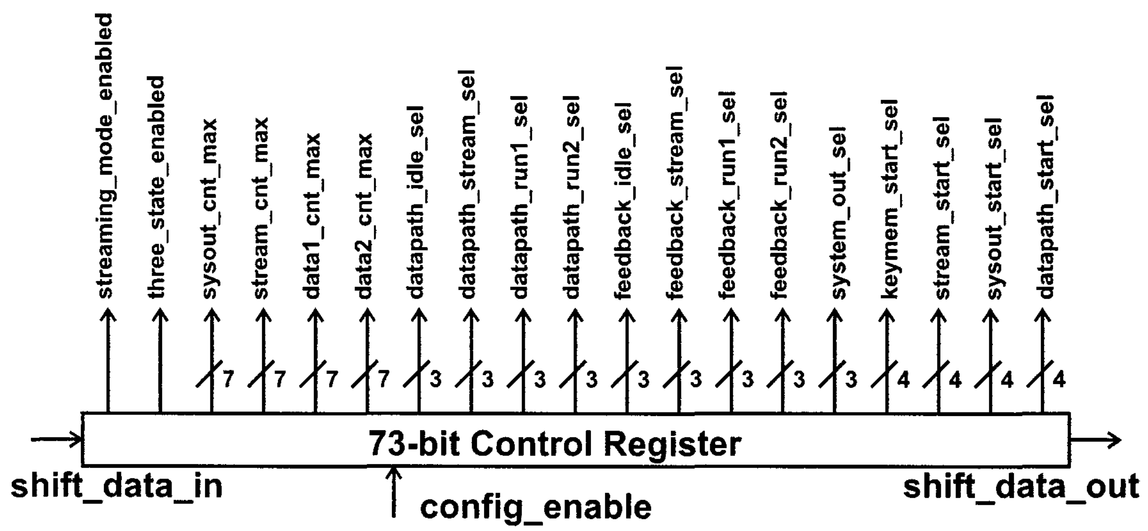


Figure 5.7: Routing Node Configuration Register Detail

the inputs with each other or an IV (in certain combinations). The possible output values for the system output, datapath output, and feedback output are as follows.

1. System Input
2. Feedback Input
3. Datapath Input
4. Datapath Input XOR IV
5. Datapath Input XOR Feedback Input
6. Datapath Input XOR System Input
7. System Input XOR Feedback Input
8. System Input XOR IV

Note that the IV value in the above XOR operations comes from either the IV register or the stream register, depending on whether the stream write signal is asserted. This

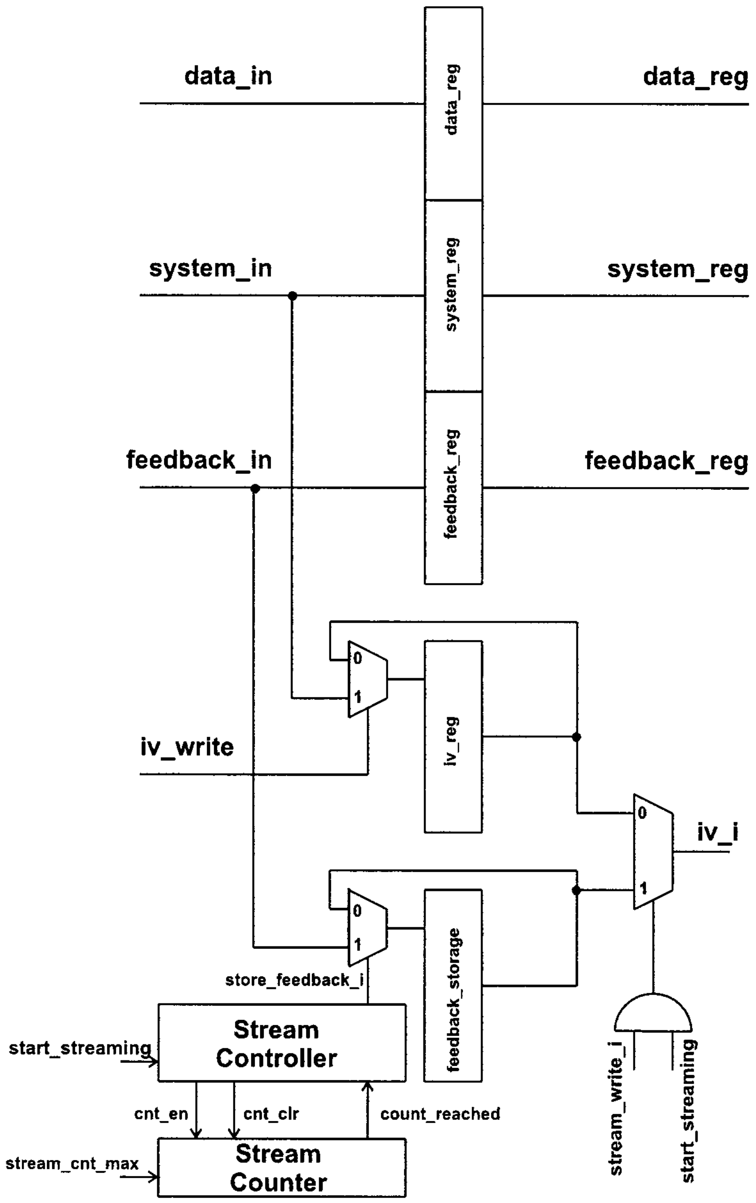


Figure 5.8: Routing Node Input Registers

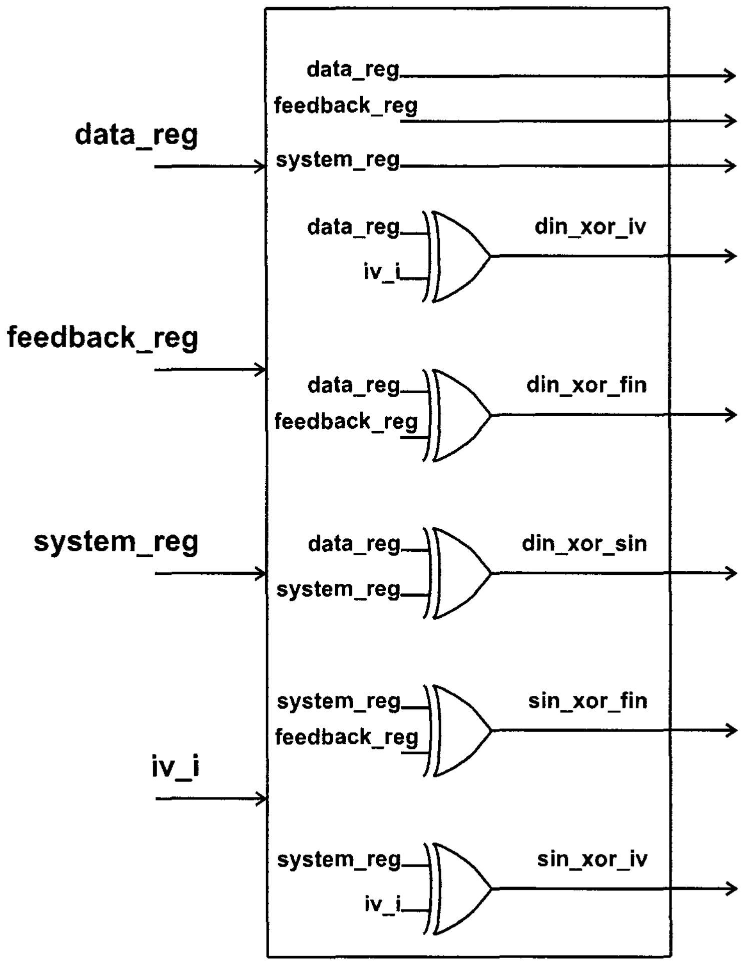


Figure 5.9: Routing Node Output Signal Generation

is the first example of where the routing node operation gets complex. This values are generated as shown in Figure 5.9.

Stream Controller

The IV register can be written to by asserting the IV write signal, which causes the data on the system input to be stored into the register. This action also invalidates the system input value stored in the system input register. Thus, an IV can be set and changed on-the-fly. When implementing an algorithm in CBC mode, for example, the user can write the IV and set the datapath output to the system input XOR IV option, to perform the XOR on the plaintext. The next plaintext block to be encrypted, however, must be XORed with the ciphertext produced by the encryption of the previous plaintext block. That ciphertext can be sent via the feedback path when the encryption is done, so that it may be used in the XOR with the next plaintext. Rather than require the external controlling processor to keep precise track of how many clock cycles are needed to encrypt one block through whatever algorithm is implemented, and then ensure that the next plaintext is written into the routing node just as the previous ciphertext arrives at its feedback input, a simple state machine controller is used.

This state machine controller, called the stream controller, uses a 7-bit counter. When the state machine receives a start signal (selected from one of the 11 possible signals by a multiplexer controlled by values in the configuration register), it starts counting clock cycles. When it reaches the maximum value, as set by the user in the configuration register, it takes the current feedback input value and writes it into the stream register before returning to an idle state. Thus, the stream controller can be configured to know how many clock cycles until the valid ciphertext will return to the feedback input, so that it can be stored, and when the external processor writes the next plaintext block in the same encryption process (at some point coincident with or after this), it can assert the stream write signal to ensure that the datapath output will be the datapath input XORed with the stored ciphertext in the stream register, rather than the still-stored value in the IV register. The stream controller is shown in Figure 5.8.

System Output Controller

The system output also requires a little management. While it is only necessary to select one of the eight values for the system output, it is also necessary to specify

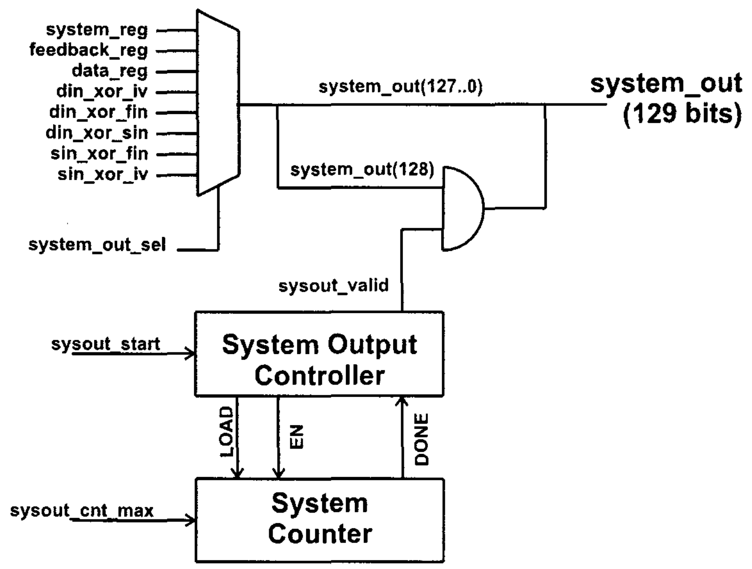


Figure 5.10: System Output Generation and Control

when that value is valid. Relying on the MSB of the signal, the valid bit, will not always work, since in an iterative implementation, the same block of valid data will pass through a routing node several times before the algorithm is complete.

Thus, it is necessary to use a state machine controller, similar to the stream controller, to validate the output data. This system output controller uses a 7-bit counter. Once triggered by a start signal or valid input data (selected by a multiplexer controlled by configuration register values), it counts until it reaches the configured maximum value and outputs a validation pulse that is ANDed with the valid bit of the selected output signal value. The counter should be configured to count the number of clocks from the start signal until the output value will be available, and this ensures only valid data is sent to the system output, and only at the appropriate times. This system is illustrated in Figure 5.10.

The operation of the system output controller should make apparent the necessity of having every start signal available to each routing node. The routing node that actually sends data to the system output will usually be a different node from the one where the data is entered into the system, and thus, if each routing node only knew its own start signal value, the system output controller would never get a start signal. By making all start signals available to each routing node, a great deal of flexibility is facilitated. Also, the output controller start signal can be configured to use any valid data coming into the node, and if the count is set to 0, it allows data to pass directly to the system output. This is useful for pipelined algorithm implementations.

Datapath and Feedback Output Controller

It has already been stated that the datapath and feedback outputs are selected by 8-to-1 multiplexers. Unfortunately, they cannot be used as simply as the system output. For example, in an iterative implementation of an algorithm would require the datapath input to come from the system input on the first iteration, but on subsequent iterations, the datapath output would have to come from the feedback input. Thus, depending on how an algorithm is implemented, a single configuration of the datapath output value will be insufficient. Similar examples can be shown for the feedback output.

To solve this problem, a state machine controller is used to control the output multiplexers for the datapath and feedback outputs. This controller, called the datapath controller, is rather more complex than the system output or stream controller. It uses two 7-bit counters as well.

The datapath controller can be configured to operate in either 2-state mode or 3-state mode. In 2-state mode, it transitions from an idle state to a running state upon receiving a start signal (which is selected from the 11 possible start signals), and stays in the running state until the first counter reaches its set maximum, at which time it will return to the idle state. If the 3-state mode is enabled, it will transition from idle to the first running state as in 2-state mode, and then to a second running state, where it will stay until the second counter reaches its configured maximum, at

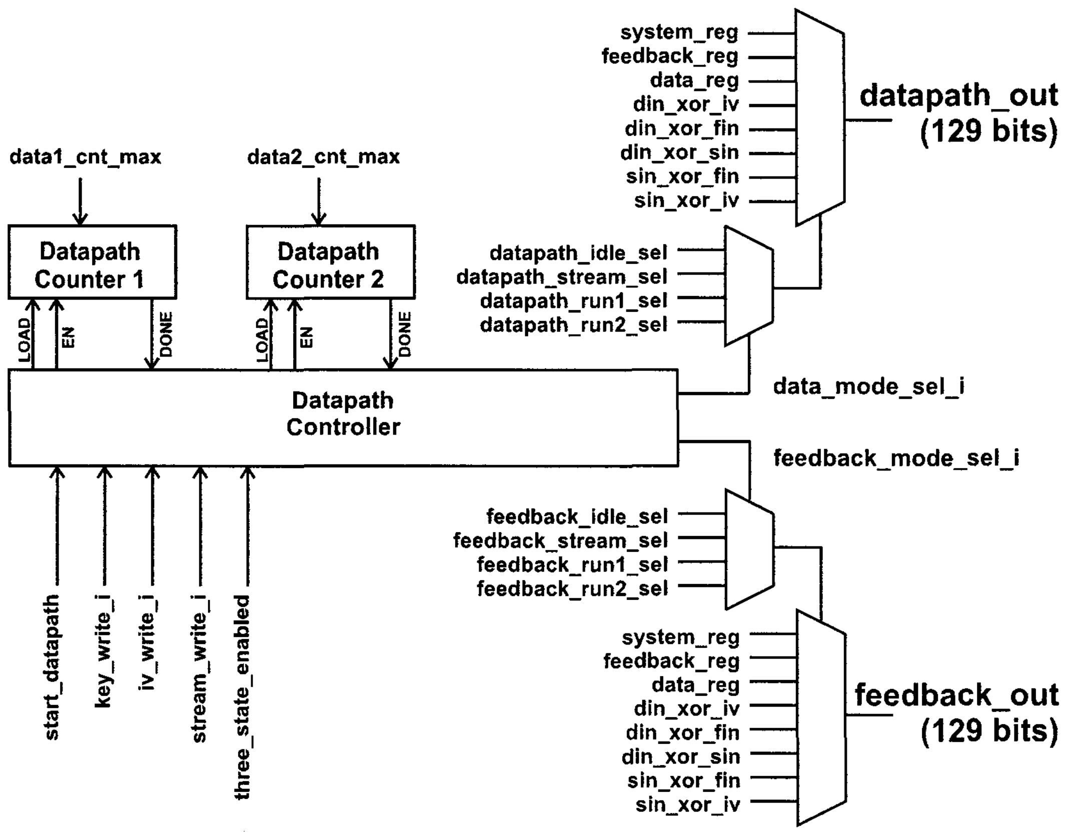


Figure 5.11: Datapath and Feedback Output Control

which time it will return to idle.

In each of these states, both the datapath output and feedback output may have different output values selected. Furthermore, if streaming is enabled in the configuration, then in the idle state, a different output value again may be selected for each output (which is used via the XOR with IV when a stream write occurs). Thus, the datapath output may have up to four different values over the course of the controller's operation, as might the feedback output. The output values for each state are set in the configuration register. The datapath and output feedback control architecture is shown in Figure 5.11.

This setup allows the previously mentioned iterative example to be implemented. For an iterative algorithm, the datapath output for the idle state would be the system input, and in the running state, the datapath output would be the feedback input. It would stay in the running state for a number of clock cycles as set in the configuration register, counted by the counter, until the algorithm is done, at which time it would return to the idle state to wait for the next input.

Key Memory Control

As previously mentioned, to keep the key memory and PE-FUNC design simple, some of its required complexity was off-loaded to the routing node. In particular, generation of the key memory start signal is performed in the routing node. The routing node also passes the key memory write enable signal along to the PE-FUNC.

The key memory start signal is selected via a large multiplexer controlled by the routing node configuration register. The start signal can be selected from any of the 11 routing node start signals, no start signal at all can be selected (for non-iterative implementations), or the valid bit of the datapath output can be used as the key memory start signal.

The key memory start signal, as well as other internal start signals, are generated as shown in Figure 5.12.

5.3 Overall System Architecture and Control

The overall system architecture for the SHERIF cryptographic hardware module is shown in Figure 5.13. Note that the top level inputs and outputs are registered, as is each of the inputs to the output controller and routing nodes, and both the inputs and outputs of all of the PE-FUNCs are registered as well.

At this level, the system architecture is quite simplistic. The processing fabric

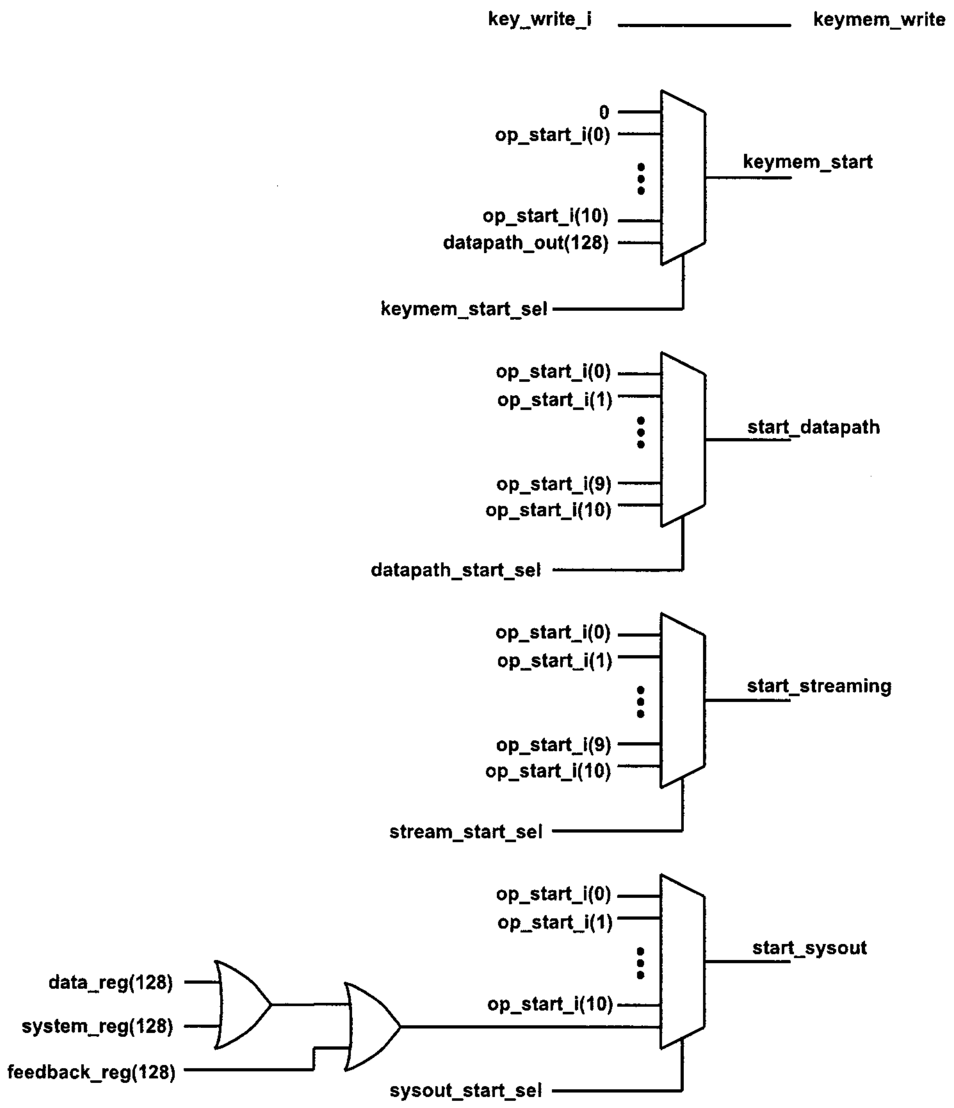


Figure 5.12: Generation of Start Signals in Routing Node

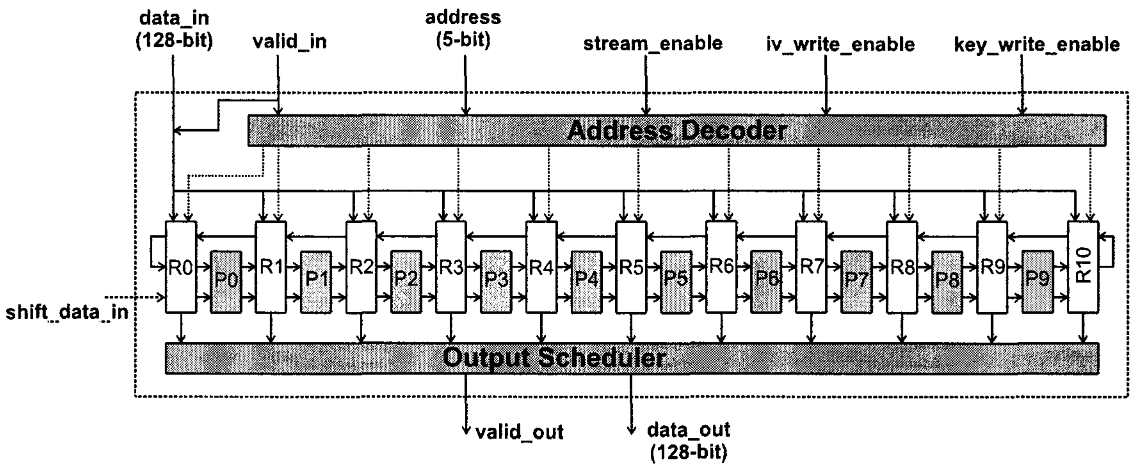


Figure 5.13: SHERIF System Diagram

has been discussed in detail in prior sections; the address decoder and system output arbiter are covered in the following sections. In brief, the address decoder handles generation of control signals to the routing nodes, whereas the system output arbiter manages switching all of the routing node system output signals onto the single SHERIF output bus. All the configuration and data processing occurs in the processing fabric – the other top-level components are merely to support the routing nodes and PE-FUNCs.

5.3.1 Address Decoder

The address decoder module is relatively straightforward in terms of its role. It processes the address values from the system input, as well as the write control signals, in order to drive the control signals going to each of the routing nodes. Thus, it takes the 5-bit address, as well as the key write, IV write, stream write, and valid bit as inputs, and generates start signals, key write signals, IV write signals, and stream write signals to each of the routing nodes. All of the start signals are sent to each routing node, as mentioned above, but at the top level, individual start signals are used to to set the valid bit of the data going into each routing node.

The address decoder works on a priority system. If the key write signal is asserted, it takes highest priority, so any other valid write inputs are ignored. Next in priority is the IV write signal, followed by the stream write signal, and then regular writes. The address decoder also has a rudimentary multicast decode mode: if the address "11111" is input, it is interpreted as being a simultaneous write to all the routing nodes, and so the appropriate control signals are sent to *all* routing nodes.

5.3.2 Dataflow Control

Once the data has entered the processing fabric, all dataflow control is entirely dependent on the routing nodes. Because the routing nodes are all independent, it is up to the user configuring the entire system to ensure that the data flows properly based on the configurations of the independent routing nodes and processing elements.

Pipelined/loop-unrolled implementations are easiest to implement with respect to controlling the dataflow; each routing node is simply set to a fixed configuration, and the system output controller just counts until data has made it from the input routing node to the final routing node. Iterative implementations are more difficult, requiring accurate timing of iterations and processing to ensure that everything works together properly. Ideally, a means of abstracting the overall dataflow control would be desirable.

5.3.3 Output Control

The system output is governed by an output bus arbiter. Since it is theoretically possible to have multiple algorithms running in parallel on different groups of PEs, it might happen that two algorithms finish at the same time, and thus both algorithms have to output their respective results to the system output. The output bus arbiter resolves contention for the single system output in such cases.

The output bus arbiter stores valid output signals from the PEs, and selects which valid output is switched onto the system output in a round-robin fashion. It can switch one value to the output every clock cycle, so as long as the average number of valid outputs is not greater than one per cycle, the output bus arbiter can keep up. It is the responsibility of the algorithm implementation to ensure that particular implementations do not generate valid outputs too frequently, to ensure that a prior output value is not overwritten before it has a chance to be switched to the output bus.

The selection algorithm works by checking the storage registers for each routing node input into the arbiter, in order, from 0 to 10. The first register that it finds with valid data in it, it switches to the system output. On the next clock cycle, it continues its check of the registers, starting from the register just *after* the register previously output, to ensure equal opportunity for data from all routing nodes. Note that the register checking is a combinational process, done within the switching clock cycle, which allows the system's high throughput.

5.3.4 Designing for Expandability

The architecture described in this section has been in reference to the proof-of-concept design that was implemented, using only ten processing elements (to allow pipelined implementation of the AES algorithm) and having no pipeline registers internal to the PEs. It should be noted, however, that the system components were designed and implemented with expandability in mind.

The address decoder decodes a 5-bit address to access the routing nodes, with the address "11111" accessing all of the nodes simultaneously. Thus, the address space has room for up to 31 routing nodes, which means 30 processing elements. Thus, the processing fabric can easily be expanded by up to a factor of three by logically extending the existing address decoder and processing fabric.

Similarly, since all the routing node and key memory state machine timing is based on counter values set by the system configuration, adding pipeline registers inside the PE-FUNC will only require algorithm implementations to change by increasing the set counter values to account for the added latency. Thus, further refinements and variations of the design in the future will be easily implemented, with simple redesign of only the address decoder and output arbiter.

5.4 Summary of System Architecture

This chapter has described the organization of the basic operational components developed in the previous chapter into a processing element capable of being configured to implement a single round of a variety of algorithms. These processing elements were then integrated into a sample system which allowed support for different modes of operation, as well as different implementation styles such as pipelining and iteration.

The proof-of-concept SHERIF system designed in this chapter is oriented toward supporting implementation of a pipelined version of the AES algorithm Rijndael [17], and thus has 10 PEs integrated into the system, with dataflow between them controlled by 11 routing nodes. The routing nodes are very complex, and responsible for the system's ability to support pipelining, iteration, and different modes of operation. The next chapter details a software configuration utility designed to help manage the complexity of the design and allow easy implementation of cryptographic algorithms on the proof-of-concept SHERIF system.

Chapter 6

System Configuration

The hardware design of the SHERIF cryptographic hardware module has been described in detail in the preceding chapters. The stated goal of this device is to provide superior performance to software implementation of cryptographic algorithms with greater ease of implementation than custom hardware implementations. While the device architecture has clearly been designed to fulfill the speed requirements, no mention has been made of how the SHERIF module is to be configured; that is, issues related to ease of implementation have not been dealt with until now.

In discussion of the SLABs in Section 4.4 and in Appendix B, the bit strings needed to to configure each component were outlined in detail. Thus, implementation of an algorithm on the SHERIF cryptographic hardware module entails creating a bit string to configure all of the components in the whole system. Hence, ease of implementation of cryptographic algorithms will depend on the ease of creation of the configuration bit strings.

The SHERIF cryptographic hardware module is configured much like an FPGA, in that the configuration is loaded in upon system start-up via the configuration I/O interface as shown in Figure 6.1. Either a dedicated start-up circuit or an external processor must assert the config_enable input, and then, on each clock cycle of the configuration clock, a single bit of the system configuration is shifted in to the shift_data_in input from a serial memory device such as a serial Programmable Read-Only Memory (PROM). The configuration bit string is shifted in from LSB to MSB.

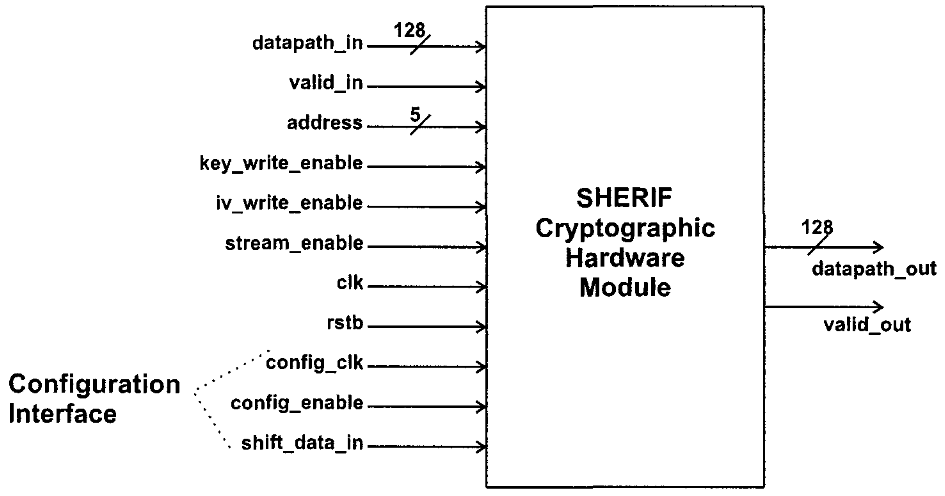


Figure 6.1: SHERIF Device Configuration I/O

Note that the configuration clock is separate from the main system clock. That is because the simplicity of the configuration logic and the close proximity of each configuration register means that the configuration register can run much faster than the entire system. Thus, by clocking the configuration logic separately, a much faster clock can be used to allow for faster configuration times. However, the same clock *can* be used for both system clock and configuration clock if so desired.

The proof-of-concept system described in this thesis contains 10 PEs and 11 routing nodes. The PEs require 45274 configuration bits each, and the routing nodes require 73 bits each. Thus, the total configuration size for the SHERIF cryptographic hardware module is 453543 bits. To generate such a configuration by hand would arguably be more difficult than designing a hardware implementation, and certainly more error-prone. The solution to this problem is to develop a software configuration utility to allow users to configure the SHERIF cryptographic hardware module.

6.1 Software Configuration Utility

The purpose of the SHERIF Configuration Utility is to provide a graphical front end to the configuration process, providing a more human-readable interface than strings of 0's and 1's and restricting users to valid configuration string choices. Most computer systems with windowing user interfaces share common interface components, such as combo boxes (whereby users select options from a drop-down list), check boxes, and text boxes. These standard interface components can allow users to implement algorithms on the SHERIF device more easily than custom hardware, and possibly more easily than software.

The SHERIF Configuration Utility was written in the Java programming language [62] using the free NetBeans 3.6 Integrated Development Environment [63]. The Java language was selected due to its ease of use and cross-platform capabilities, as well as the comprehensive Swing windowing library which provides all the desired user interface controls. The NetBeans IDE made graphical user interface development easy, allowing the creation of user interface forms with virtually no coding. The only programming directly required was the underlying reading, validation, and manipulation of the data represented by the components.

The configuration utility was designed in a modular fashion, with separate user interface forms implemented for each of the components to be configured. A ConfigurationString class was implemented to encapsulate the behaviour needed by the configuration forms: it had to store the strings, overwrite the string or parts of them, and allow them to be read as a whole, in part, or one bit at a time for saving to a file.

The forms for each component are accessed hierarchically. Each of the forms was implemented as a modal form, meaning that when it is active, the user cannot switch to a different form directly, but has to close the existing form or cause it to launch a new one. This was done to ensure the user is able to keep track of what is currently being configured.

The main form of the configuration utility is shown in Figure 6.2. It consists of buttons representing each of the PEs and routing nodes, each of which launches a new configuration form to configure that particular component. Configuration of individual components is covered in the following sections. This form could be further refined by offering combo boxes instead of buttons to allow selection between the different routing nodes and PEs. Such a modification would simplify the interface, and at the same time make it scalable to systems with different numbers of PEs.

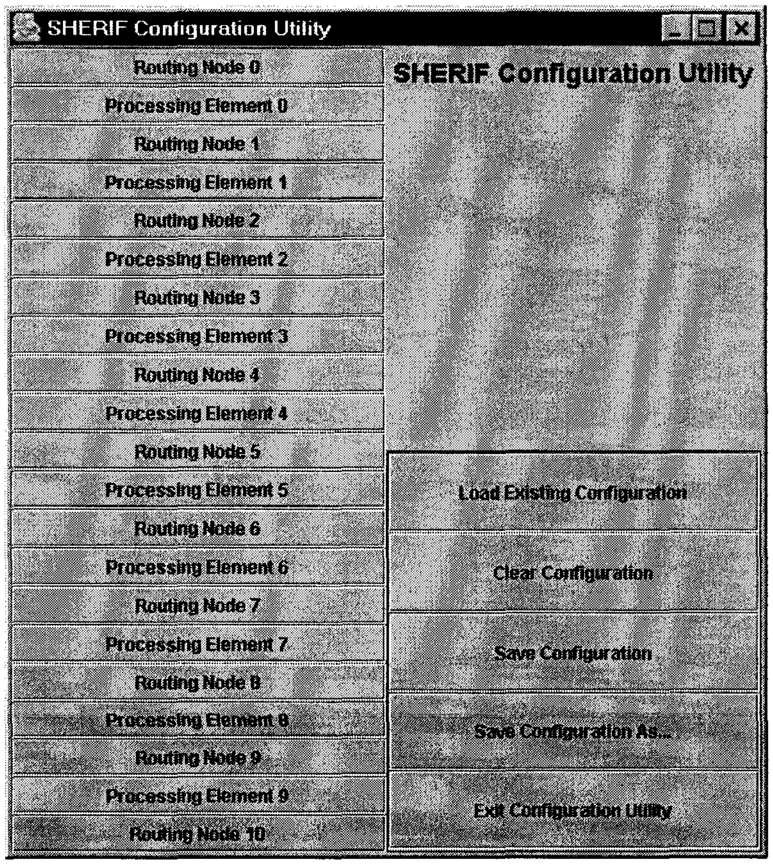


Figure 6.2: SHERIF Configuration Utility Main Form

The main form also includes basic functionality such as the ability to save the configuration to a text file, or to read in an existing configuration.

The system configuration is saved as a standard text file. The first line contains the text "SHERIFCONFIG", and the last line the text "END SHERIFCONFIG". In between these lines are the 453543 configuration bits, all stored in a single line, from the LSB at the beginning of the line to the MSB at the end. This is so that the text file, when used in functional simulation of the SHERIF cryptographic hardware module, can be read with a single line read and then shifted in to the configuration one bit at a time. Should the SHERIF device be fabricated, the configuration utility would have to be extended to support generation of a binary configuration file that could be programmed into a PROM.

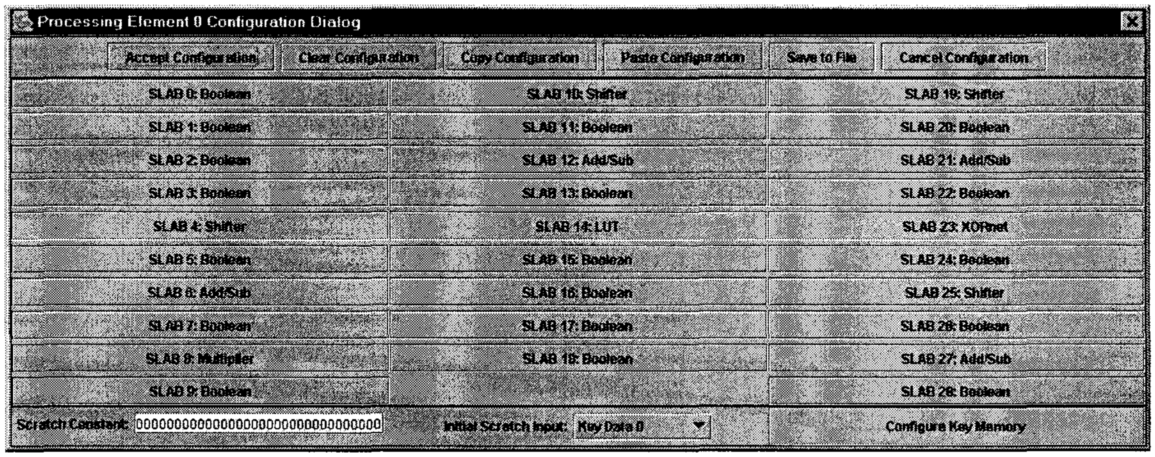
Overall the system is designed so that each form generates a small configuration based on its controls. When the user accepts the configuration of the component, the configuration is validated and then passed back to a parent form, where the configuration string is integrated into the larger configuration (which might itself then be accepted and passed to another parent and integrated to its configuration). Following this strictly hierarchical design process greatly simplified implementation, allowing easy use of the forms for the different components as needed and ensuring that reading values from the user interface was kept as simple as possible.

Thus, the main form in Figure 6.2 has 10 ConfigurationString objects representing the PEs, and 11 representing the routing nodes. When one of the buttons is clicked to launch the configuration form for a PE or routing node, the main form passes a reference to the configuration string it holds to that newly created form, and that form, once its configuration is accepted, writes the new configuration values back into the string held by its parent. When saving the overall configuration to a text file, the 21 different strings are combined into one temporary ConfigurationString object that is then written to the file.

The operation of the software is more obvious when the individual forms are considered, in the following sections.

6.2 PE-FUNC Configuration Form

When the PE-FUNC configuration form is launched from the main form by clicking one of the PE buttons, a new form as shown in Figure 6.3 is displayed.



Scratch Constant: 00000000000000000000000000000000

Initial Scratch Input: Key Data 0

Figure 6.3: Top-level PE-FUNC Configuration Form

It contains 29 buttons to launch configuration forms for the individual SLABs, a button to launch the key memory configuration form, and some standard buttons at the top of the form. The only items configurable at this level are the scratch path input selection and the top-level PE-FUNC constant value.

A combo box is used to select the input to the scratch path between two key values, the aforementioned constant value, or the datapath value. A simple text box is provided to allow the user to enter the desired constant value in hexadecimal notation. The software verifies the length and format of the user-entered values to ensure they are valid.

The standard control buttons that appear on every component configuration form are "Accept Configuration", "Clear Configuration", "Copy Configuration", "Paste Configuration", and "Cancel Configuration".

"Accept Configuration" allows the user to accept and store the changes they have made to the configuration. It reads the data from the components into the configuration string (after validating it, of course), then writes the configuration string back into the parent configuration and closes the form, returning window focus to the parent form.

"Clear Configuration" sets the configuration string to the default, all zeros. "Cancel Configuration" discards any changes that have not been accepted and closes the form, returning focus to the parent. It does provide a warning before doing so, however.

"Copy Configuration" and "Paste Configuration" work together. Copying stores the current configuration to a component-specific clipboard held by the parent. Then, whenever in a form of the same type, the value can be pasted in to quickly configure the new, different form, or overwrite its existing configuration.

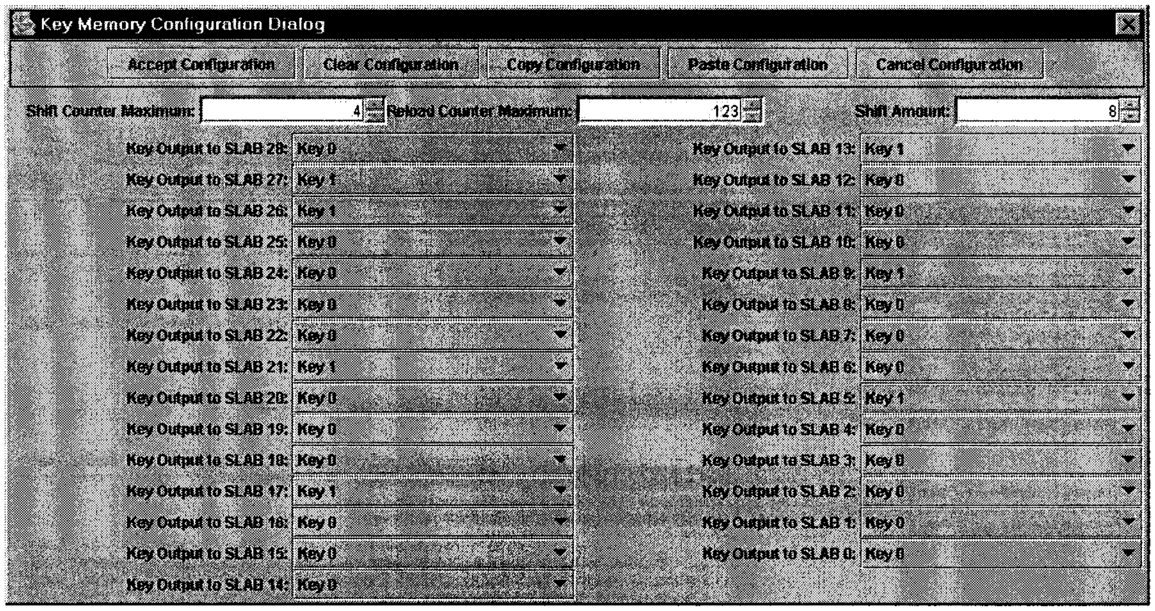
For example, in the PE-FUNC configuration form for PE0, the user might set a configuration, and click "Copy". Then, the user might accept the configuration, and from the main window, open the PE1 form, and click "Paste" to give PE1 the same configuration as PE0. Note that the copied configuration includes the configuration of all the sub-forms, as well. If the user tried to paste into a routing node form, however, nothing would happen. The clipboards are form-specific, and only copy and paste between forms of the same type.

The PE-FUNC configuration form has an additional control button, allowing the PE-FUNC configuration to be saved to a file. This ability to save just a single PE-FUNC configuration was included for testing and debugging purposes.

The key memory configuration form is detailed in the following section. Configuration forms of the individual SLABs are covered in Section 6.3.

6.2.1 Key Memory Configuration Form

The key memory configuration form, which is launched from the PE-FUNC configuration form, is shown in Figure 6.4. It is a relatively straightforward form, having the standard control buttons at top.



Shift Counter Maximum: 4 Reload Counter Maximum: 123 Shift Amount: 8

Key Output to SLAB 28:	Key 0	Key Output to SLAB 13:	Key 1
Key Output to SLAB 27:	Key 1	Key Output to SLAB 12:	Key 0
Key Output to SLAB 26:	Key 1	Key Output to SLAB 11:	Key 0
Key Output to SLAB 25:	Key 0	Key Output to SLAB 10:	Key 0
Key Output to SLAB 24:	Key 0	Key Output to SLAB 9:	Key 1
Key Output to SLAB 23:	Key 0	Key Output to SLAB 8:	Key 0
Key Output to SLAB 22:	Key 0	Key Output to SLAB 7:	Key 0
Key Output to SLAB 21:	Key 1	Key Output to SLAB 6:	Key 0
Key Output to SLAB 20:	Key 0	Key Output to SLAB 5:	Key 1
Key Output to SLAB 19:	Key 0	Key Output to SLAB 4:	Key 0
Key Output to SLAB 18:	Key 0	Key Output to SLAB 3:	Key 0
Key Output to SLAB 17:	Key 1	Key Output to SLAB 2:	Key 0
Key Output to SLAB 16:	Key 0	Key Output to SLAB 1:	Key 0
Key Output to SLAB 15:	Key 0	Key Output to SLAB 0:	Key 0
Key Output to SLAB 14:	Key 0		

Figure 6.4: Key Memory Configuration Form

There are 29 combo boxes allowing the user to select whether Key 0 (the least significant 128 bits of the key memory register) or Key 1 (bits 255 down to 128 of the key memory register) is sent to their respective SLABs.

There are also 3 spinner components, which allow selection of integer values within a specified range. Here, the first spinner allows the user to set the maximum value of the shift counter (which counts the clock cycles between shifts of the key memory register) to a value in the range of 0 to 127. The next spinner allows the user to set the maximum value of the reload counter (which counts the number of shifts until key memory reset) in the same range. The third spinner ranges from 0 to 8, and controls the size of the key memory shift, with the value representing the number of 32-bit words by which the key memory register will shift when the shift counter reaches its maximum.

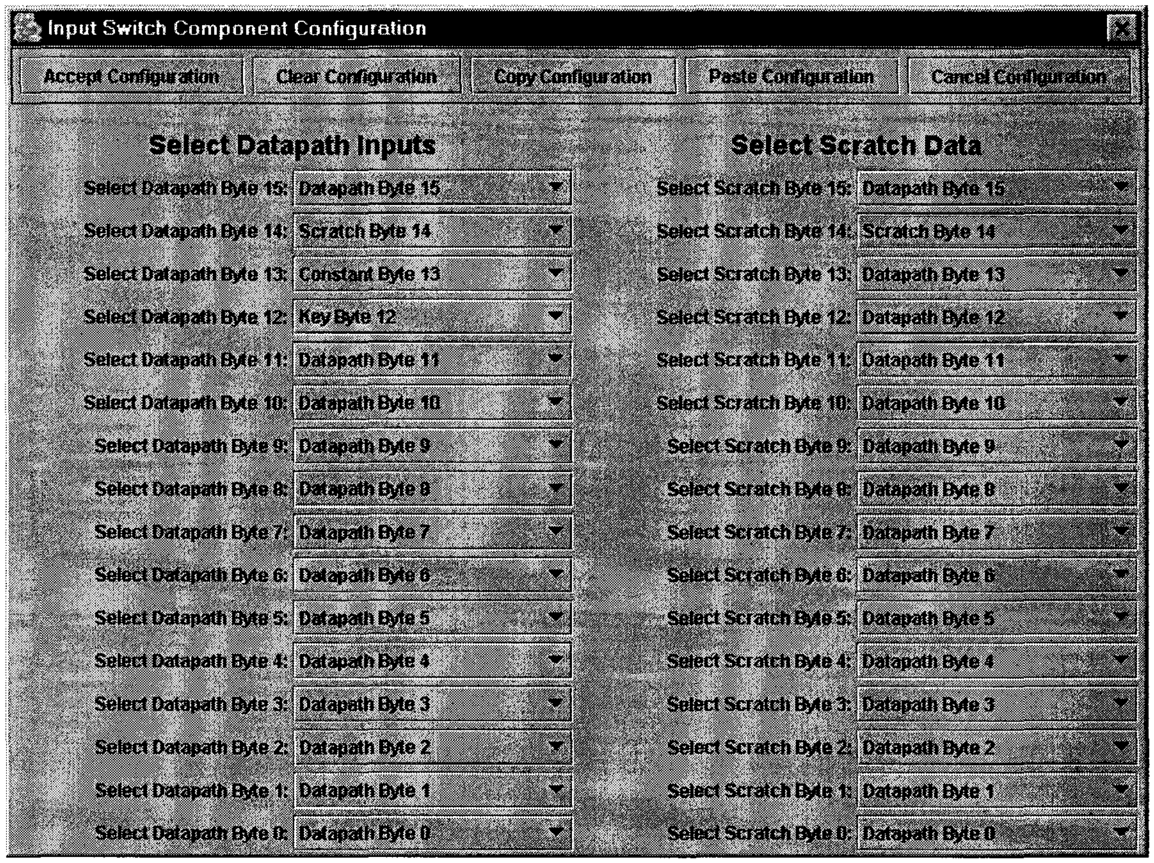
6.3 SLAB Configuration Forms

Each of the six SLABs has its own configuration form. However, since all SLABs incorporate input switch and byte reordering components, they all use identical forms to configure those parts of the SLAB. Each of the six forms has buttons to launch configuration forms for the input switch and byte reordering.

The input switch configuration form is shown in Figure 6.5. It has the standard controls at the top, and 32 combo boxes. The 16 combo boxes on the left control the multiplexers on the datapath of the input switch, allowing the user to select for each of the 16 datapath bytes whether a datapath byte, scratch data byte, constant byte, or key byte gets passed to the core component. The rightmost 16 combo boxes select the data to be sent to the scratch path output, either the current scratch input, or else the current datapath input.

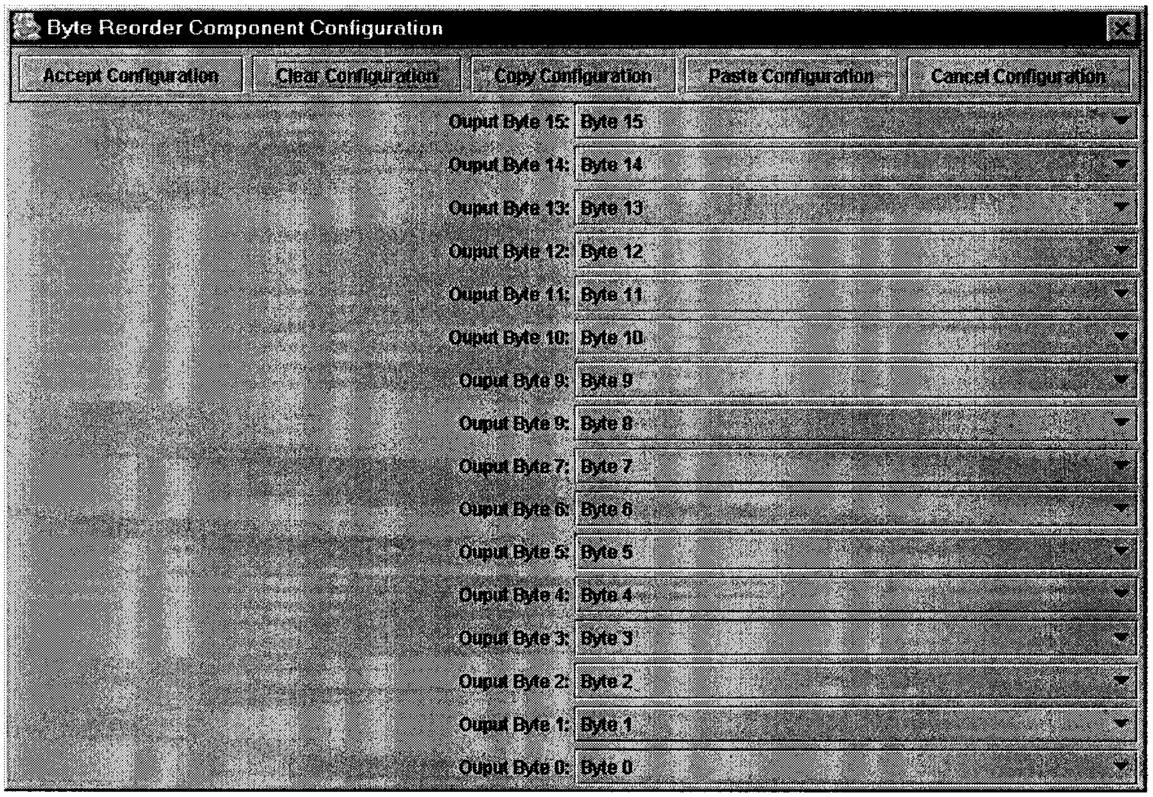
The byte reordering configuration form shown in Figure 6.6 is somewhat similar, having the standard controls across the top. It has only 16 combo boxes, which allow the user to configured the multiplexers of the byte reorder component. Each combo box represents an output byte, and any of the 16 input bytes can be selected for each output.

The following sections discuss the individual SLAB configuration forms, which incorporate the two forms discussed above as sub-forms.



Select Datapath Inputs		Select Scratch Data	
Select Datapath Byte 15:	Datapath Byte 15	Select Scratch Byte 15:	Datapath Byte 15
Select Datapath Byte 14:	Scratch Byte 14	Select Scratch Byte 14:	Scratch Byte 14
Select Datapath Byte 13:	Constant Byte 13	Select Scratch Byte 13:	Datapath Byte 13
Select Datapath Byte 12:	Key Byte 12	Select Scratch Byte 12:	Datapath Byte 12
Select Datapath Byte 11:	Datapath Byte 11	Select Scratch Byte 11:	Datapath Byte 11
Select Datapath Byte 10:	Datapath Byte 10	Select Scratch Byte 10:	Datapath Byte 10
Select Datapath Byte 9:	Datapath Byte 9	Select Scratch Byte 9:	Datapath Byte 9
Select Datapath Byte 8:	Datapath Byte 8	Select Scratch Byte 8:	Datapath Byte 8
Select Datapath Byte 7:	Datapath Byte 7	Select Scratch Byte 7:	Datapath Byte 7
Select Datapath Byte 6:	Datapath Byte 6	Select Scratch Byte 6:	Datapath Byte 6
Select Datapath Byte 5:	Datapath Byte 5	Select Scratch Byte 5:	Datapath Byte 5
Select Datapath Byte 4:	Datapath Byte 4	Select Scratch Byte 4:	Datapath Byte 4
Select Datapath Byte 3:	Datapath Byte 3	Select Scratch Byte 3:	Datapath Byte 3
Select Datapath Byte 2:	Datapath Byte 2	Select Scratch Byte 2:	Datapath Byte 2
Select Datapath Byte 1:	Datapath Byte 1	Select Scratch Byte 1:	Datapath Byte 1
Select Datapath Byte 0:	Datapath Byte 0	Select Scratch Byte 0:	Datapath Byte 0

Figure 6.5: Input Switch Configuration Form

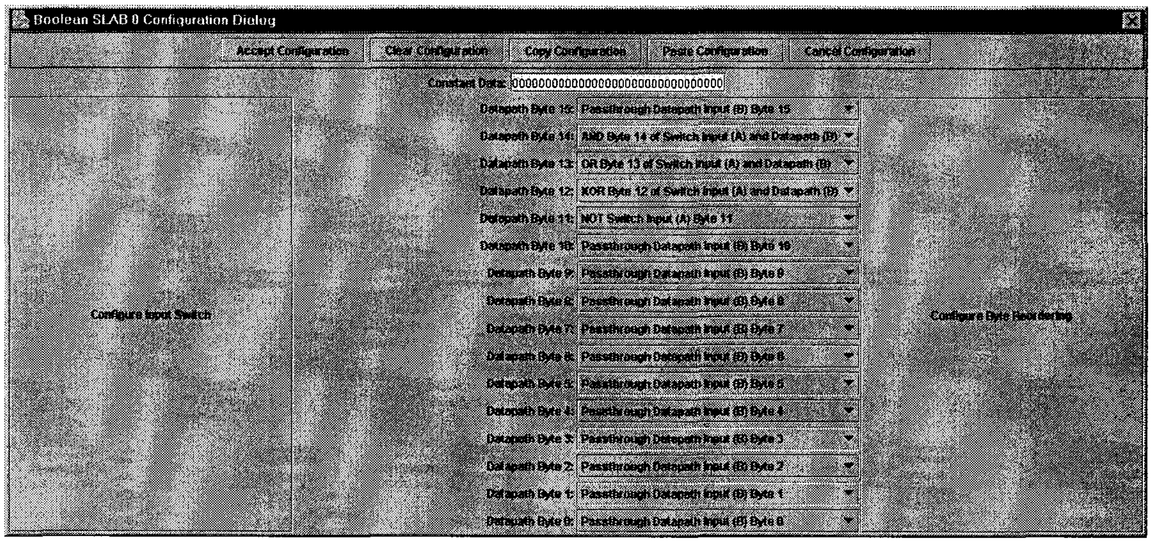


Ouput Byte 15:	Byte 15
Ouput Byte 14:	Byte 14
Ouput Byte 13:	Byte 13
Ouput Byte 12:	Byte 12
Ouput Byte 11:	Byte 11
Ouput Byte 10:	Byte 10
Ouput Byte 9:	Byte 9
Ouput Byte 9:	Byte 8
Ouput Byte 7:	Byte 7
Ouput Byte 6:	Byte 6
Ouput Byte 5:	Byte 5
Ouput Byte 4:	Byte 4
Ouput Byte 3:	Byte 3
Ouput Byte 2:	Byte 2
Ouput Byte 1:	Byte 1
Ouput Byte 0:	Byte 0

Figure 6.6: Byte Reordering Configuration Form

6.3.1 Boolean SLAB Configuration Form

The Boolean SLAB configuration form, shown in Figure 6.7, is relatively straightforward. It has the standard control buttons at the top, and buttons to launch configuration of the input switch and byte reordering forms. The core logic is configured by the 16 combo boxes, which allows each output byte to be selected between the default bypass mode, or the boolean operations AND, OR, and XOR of the datapath and input switch values, or the inverse of the input switch values. The SLAB constant can be entered via the text box in hexadecimal notation.

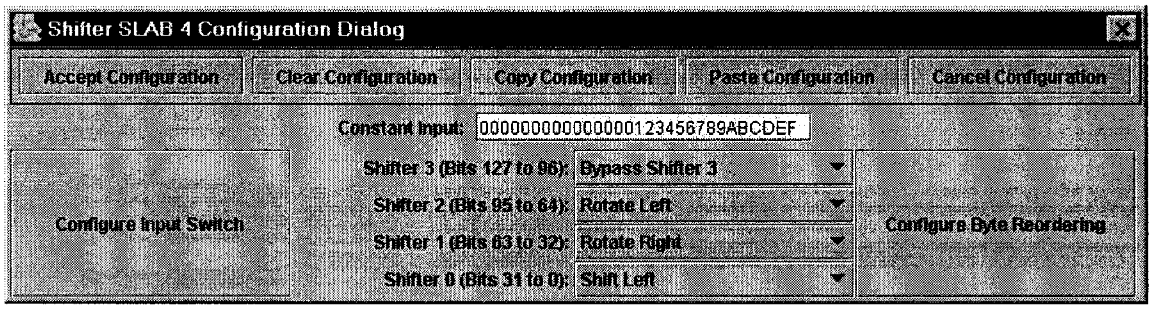


Constant Data:	00000000000000000000000000000000
Datapath Byte 15:	Passthrough Datapath Input (B) Byte 15
Datapath Byte 14:	AND Byte 14 of Switch Input (A) and Datapath (B)
Datapath Byte 13:	OR Byte 13 of Switch Input (A) and Datapath (B)
Datapath Byte 12:	XOR Byte 12 of Switch Input (A) and Datapath (B)
Datapath Byte 11:	NOT Switch Input (A) Byte 11
Datapath Byte 10:	Passthrough Datapath Input (B) Byte 10
Datapath Byte 9:	Passthrough Datapath Input (B) Byte 9
Datapath Byte 8:	Passthrough Datapath Input (B) Byte 8
Datapath Byte 7:	Passthrough Datapath Input (B) Byte 7
Datapath Byte 6:	Passthrough Datapath Input (B) Byte 6
Datapath Byte 5:	Passthrough Datapath Input (B) Byte 5
Datapath Byte 4:	Passthrough Datapath Input (B) Byte 4
Datapath Byte 3:	Passthrough Datapath Input (B) Byte 3
Datapath Byte 2:	Passthrough Datapath Input (B) Byte 2
Datapath Byte 1:	Passthrough Datapath Input (B) Byte 1
Datapath Byte 0:	Passthrough Datapath Input (B) Byte 0

Figure 6.7: Boolean SLAB Configuration Form

6.3.2 Shifter SLAB Configuration Form

The Shifter SLAB configuration form is shown in Figure 6.8. It has the standard controls across the top, as well as the SLAB-standard text box for entry of the constant value in hexadecimal notation and buttons to launch input switch and byte reordering forms. Since the shifter is based around 32-bit shift components, only 4 combo boxes are needed to configure the component. Each combo box selects between bypass,



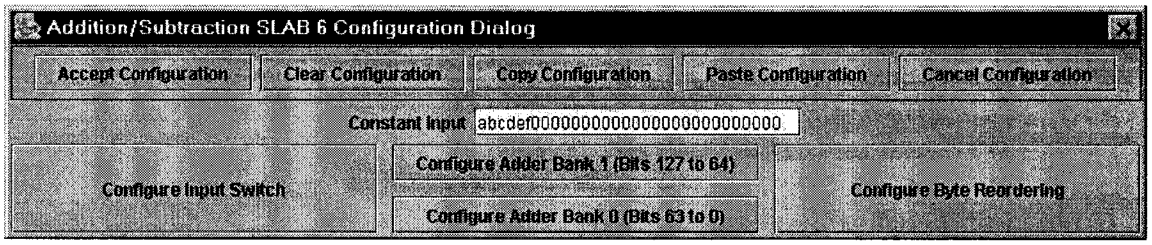
Constant Input: 00000000000000000123456789ABCDEF
Shifter 3 (Bits 127 to 96): Bypass Shifter 3
Shifter 2 (Bits 95 to 64): Rotate Left
Shifter 1 (Bits 63 to 32): Rotate Right
Shifter 0 (Bits 31 to 0): Shift Left

Figure 6.8: Shifter SLAB Configuration Form

left/right shift, and left/right rotation for its corresponding 32-bit input.

6.3.3 Add/Sub SLAB Configuration Form

The Add/Sub SLAB configuration form, shown in Figure 6.9, is a bit more complex than the previous two SLABs. It does, however, have the same standard controls, constant value text box, and input switch and byte reordering buttons.

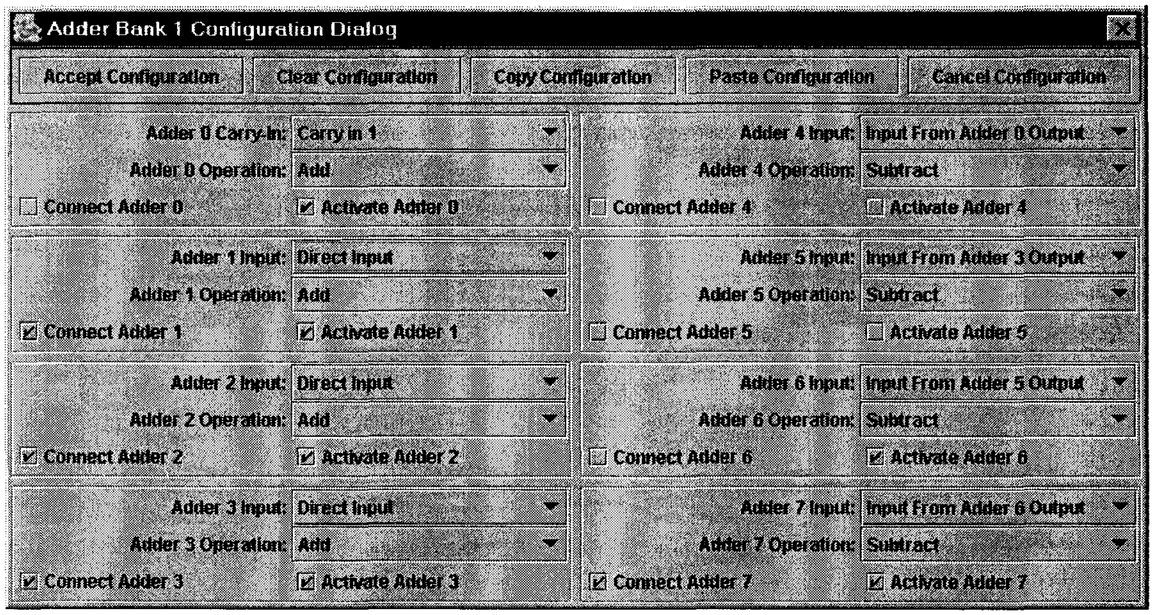

Addition/Subtraction SLAB 6 Configuration Dialog
Accept Configuration
Clear Configuration
Copy Configuration
Paste Configuration
Cancel Configuration
Configure Input Switch
Configure Adder Bank 1 (Bits 127 to 64)
Configure Adder Bank 0 (Bits 63 to 0)
Configure Byte Reordering

Constant Input: abcdef00000000000000000000000000

Figure 6.9: Add/Sub SLAB Configuration Form

The configuration of the adders themselves is rather extensive, since there are a number of options. Including that many components on the SLAB form would be unwieldy, so a separate sub-form was designed to configure the two banks of 8 adders each. This form is shown in Figure 6.10.

The adder bank configuration form is fairly complex. It has the standard controls at the top, and then is broken into eight panels, one to configure each of the adders.



Adder 0 Carry-In: Carry in 1
Adder 0 Operation: Add
Connect Adder 0 [✓ Activate Adder 0]

Adder 1 Input: Direct Input
Adder 1 Operation: Add
[✓ Connect Adder 1] [✓ Activate Adder 1]

Adder 2 Input: Direct Input
Adder 2 Operation: Add
[✓ Connect Adder 2] [✓ Activate Adder 2]

Adder 3 Input: Direct Input
Adder 3 Operation: Add
[✓ Connect Adder 3] [✓ Activate Adder 3]

Adder 4 Input: Input From Adder 0 Output
Adder 4 Operation: Subtract
Connect Adder 4 Activate Adder 4

Adder 5 Input: Input From Adder 3 Output
Adder 5 Operation: Subtract
Connect Adder 5 Activate Adder 5

Adder 6 Input: Input From Adder 5 Output
Adder 6 Operation: Subtract
Connect Adder 6 [✓ Activate Adder 6]

Adder 7 Input: Input From Adder 6 Output
Adder 7 Operation: Subtract
[✓ Connect Adder 7] [✓ Activate Adder 7]

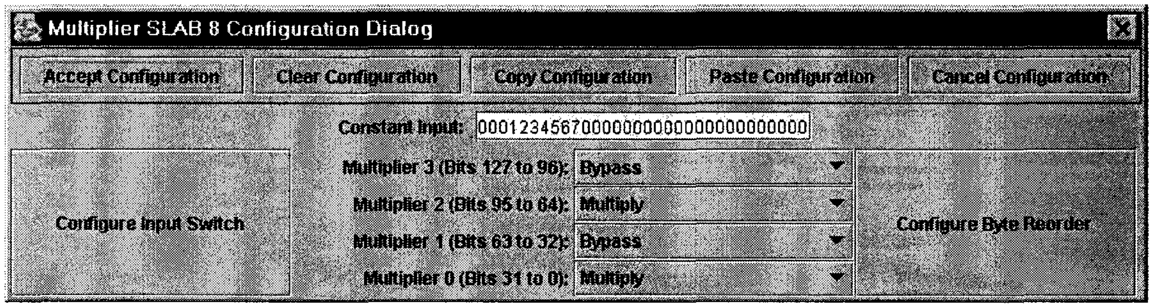
Figure 6.10: Configuration Form for Bank of 8 Adders

Each of the panels has check boxes to allow the user to select whether the adder is active or not, and whether is is connected to the preceding adder (that is, whether is uses the carry-in). Each panel also has a combo box allowing the component to be switched from addition to subtraction. The remaining combo box differs slightly between the panels.

In the first panel, the topmost combo box allows selection of the carry-in to the first adder. In the rest of the panels, however, the topmost combo box allows selection of one of the adder inputs between the direct input and the outputs of the preceding adders. Thus, the combo box for each adder gets more and more options. The full flexibility of the add/sub component is made available through this form.

6.3.4 Multiplier SLAB Configuration Form

The Multiplier SLAB configuration form, shown in Figure 6.11, is quite simple. It has the standard controls, constant value input text box, and input switch/byte reorder buttons. It then has four combo boxes to configure the four 32-bit multipliers by either bypassing them or using them.



Constant Input: 00012345670000000000000000000000
Multiplier 3 (Bits 127 to 96): Bypass
Multiplier 2 (Bits 95 to 64): Multiply
Multiplier 1 (Bits 63 to 32): Bypass
Multiplier 0 (Bits 31 to 0): Multiply

Figure 6.11: Multiplier SLAB Configuration Form

6.3.5 LUT SLAB Configuration Form

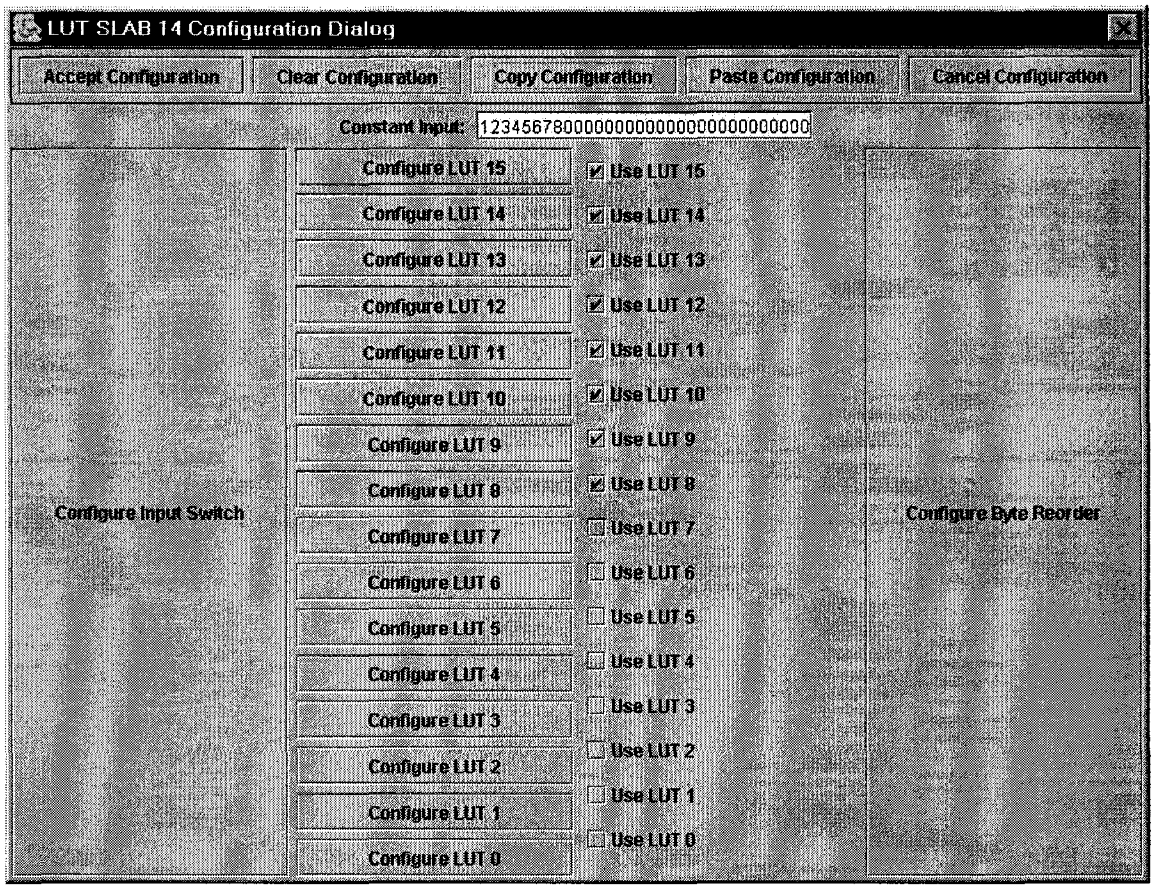
The LUT SLAB configuration form is shown in Figure 6.12. It has the standard controls, constant value input text box, and input switch/byte reorder buttons, as well as 16 buttons and 16 check boxes. The check boxes select which LUTs are in use, and the buttons configure the LUTs.

The configuration interface of the actual LUTs themselves is complex, and so was relegated to a sub-form to avoid cluttering the screen. A sub-form to configure the individual 8×8 LUTs was implemented, as shown in Figure 6.13.

The form is built around a 16×16 table of two-digit hexadecimal values. Conceptually, an 8-bit input selects a row and column from the table, and the value in the cell at the intersection of the row and column is the output value. The first 4 bits of the 8-bit input select the row; the last 4 bits select the column.

To configure the LUT, the user simply enters the desired values into the table in hexadecimal notation. The software validates the length and value of the user inputs.

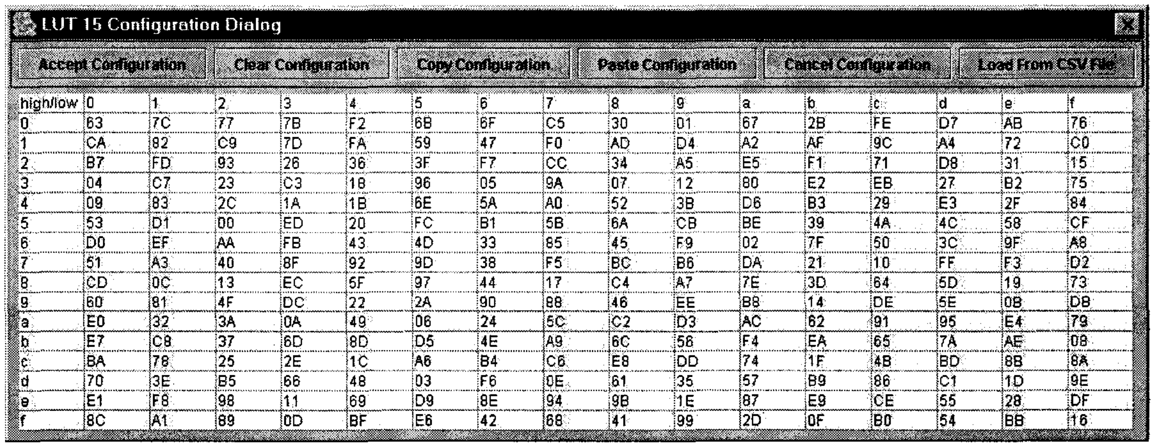
This form has the standard controls at the top, plus one extra, "Load From CSV File". Since entering 256 table values can be tedious, and since many algorithm



Constant Input: 12345678000000000000000000000000

- ☑ Use LUT 15
- ☑ Use LUT 14
- ☑ Use LUT 13
- ☑ Use LUT 12
- ☑ Use LUT 11
- ☑ Use LUT 10
- ☑ Use LUT 9
- ☑ Use LUT 8
- ☐ Use LUT 7
- ☐ Use LUT 6
- ☐ Use LUT 5
- ☐ Use LUT 4
- ☐ Use LUT 3
- ☐ Use LUT 2
- ☐ Use LUT 1
- ☐ Use LUT 0

Figure 6.12: LUT SLAB Configuration Form



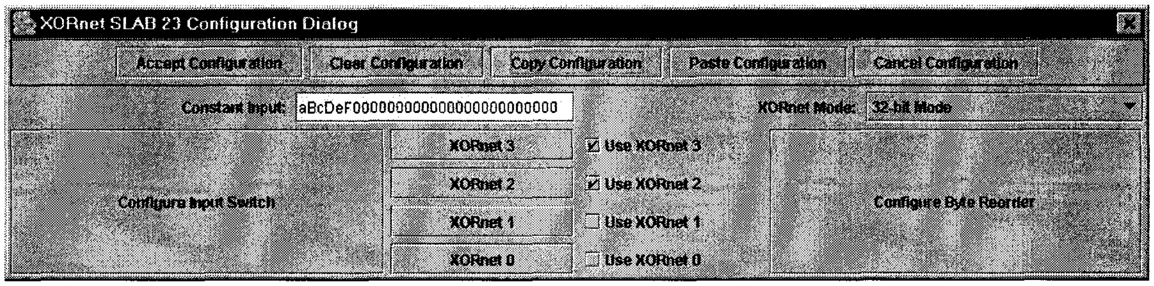
high/low	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
a	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
b	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
c	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
d	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
e	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
f	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figure 6.13: 8 × 8 LUT Configuration Form

specifications provide some sort of representation of LUT values in a format that can be converted into a text file, an additional piece of functionality was provided to the 8×8 LUT form to allow it to read in such text files, in CSV (comma separated values) format. The data can be read from a text file, saving the user time and removing the possibility of transcription error. The text file, however, must have a specific format: 256 values, each value being 2 hexadecimal characters, arranged in 16 lines of 16 values each, where the values are separated by commas.

6.3.6 XORnet SLAB Configuration Form

The XORnet SLAB configuration form is shown in Figure 6.14. It is relatively simple, with standard controls at the top, buttons for the input switch and byte reorder configuration forms, and a text box for constant value input.



Constant Input: aBcDeF00000000000000000000000000

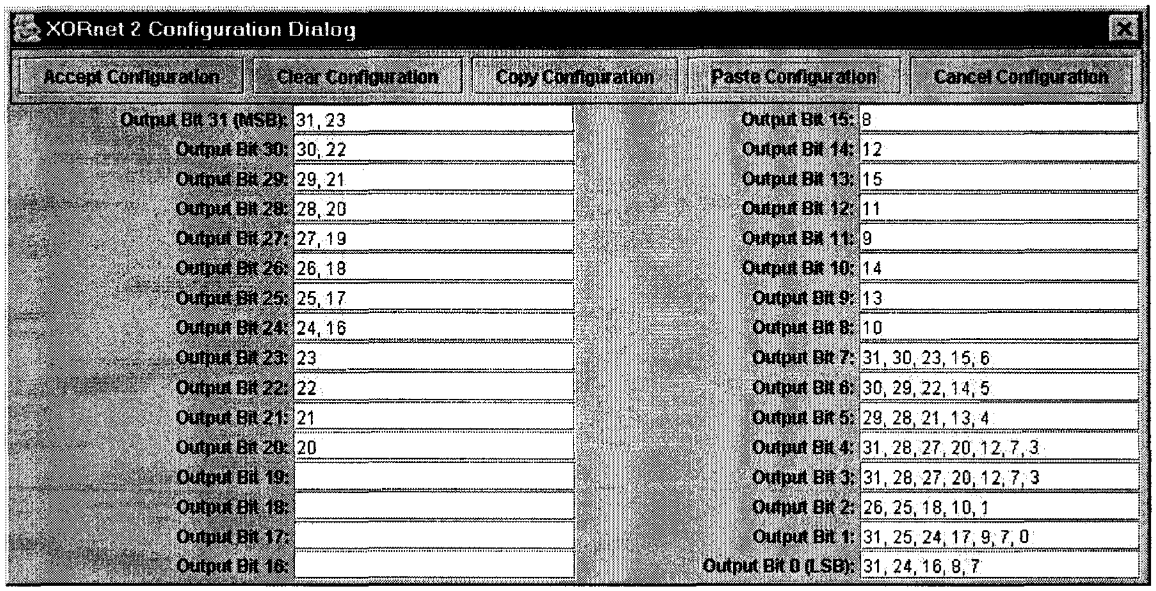
XORnet Mode: 32-bit Mode

☑ Use XORnet 3
☑ Use XORnet 2
☐ Use XORnet 1
☐ Use XORnet 0

Figure 6.14: XORnet SLAB Configuration Form

The form also provides a combo box to select the mode of operation of the XORnet component. There are two possible modes, "32-bit Mode" and "64-bit Mode". "32-bit Mode" allows the XORnet component to operate as 4 independent 32-bit XORnets. "64-bit Mode" uses all four 32-bit XORnets to operate as a single 64-bit XORnet.

There are four buttons which launch forms to configure the individual 32-bit XORnets themselves, and four check boxes to select which XORnets are to be active. The configuration form for a single 32-bit XORnet is shown in Figure 6.15.



Output Bit 31 (MSB):	31, 23	Output Bit 15:	8
Output Bit 30:	30, 22	Output Bit 14:	12
Output Bit 29:	29, 21	Output Bit 13:	15
Output Bit 28:	28, 20	Output Bit 12:	11
Output Bit 27:	27, 19	Output Bit 11:	9
Output Bit 26:	26, 18	Output Bit 10:	14
Output Bit 25:	25, 17	Output Bit 9:	13
Output Bit 24:	24, 16	Output Bit 8:	10
Output Bit 23:	23	Output Bit 7:	31, 30, 23, 15, 6
Output Bit 22:	22	Output Bit 6:	30, 29, 22, 14, 5
Output Bit 21:	21	Output Bit 5:	29, 28, 21, 13, 4
Output Bit 20:	20	Output Bit 4:	31, 28, 27, 20, 12, 7, 3
Output Bit 19:		Output Bit 3:	31, 28, 27, 20, 12, 7, 3
Output Bit 18:		Output Bit 2:	26, 25, 18, 10, 1
Output Bit 17:		Output Bit 1:	31, 25, 24, 17, 9, 7, 0
Output Bit 16:		Output Bit 0 (LSB):	31, 24, 16, 8, 7

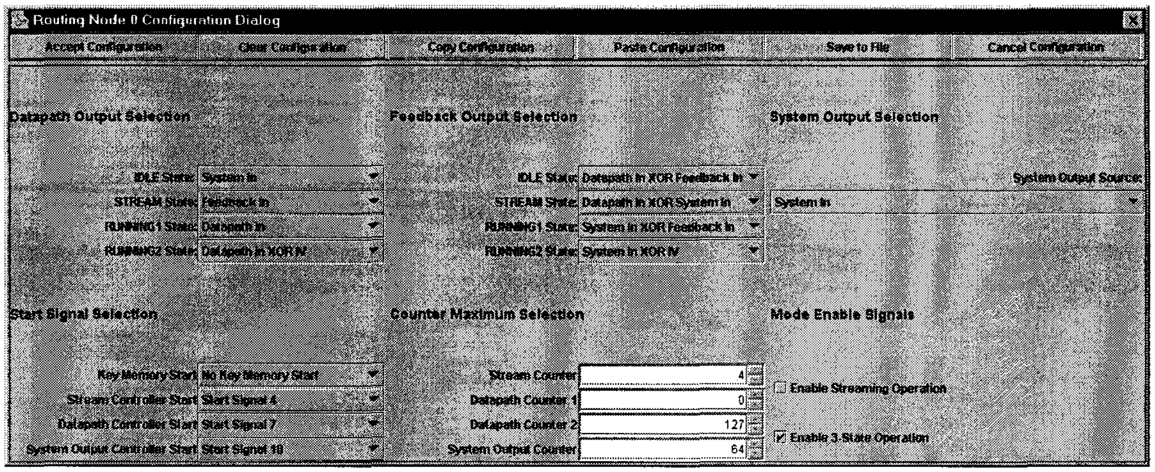
Figure 6.15: 32-Bit XORnet Configuration Form

This single XORnet configuration form has standard controls and 32 text boxes. The 32 text boxes correspond to the 32 output bits of the component. The component is configured by typing values into the text boxes (separated by commas). The values must be decimal values in the range of 0 to 31, and represent which input bits are XORed together to produce an output for that bit. Thus, if the text box labeled "Output Bit 1" had the values "31, 17, 4" in it, it would mean that output bit 1 is generated by input bit 31 XORed with input bit 17 XORed with input bit 4. Each of the output bits is independent of the others. An empty text box means the output for that bit will be zero.

This type of interface is the simplest way to provide the flexibility needed for the XORnet. Each output bit can be the XOR of any of the input bits, but this text box based interface avoids the tedium of switching 32 bits on and off for each output bit, and allows only the desired ones to be typed in.

6.4 Routing Node Configuration Form

The routing node configuration form is shown in Figure 6.16. Though the operation of the routing node is very complex to understand, configuring it is relatively straightforward. It just requires selection of a large number of unrelated options.



Datapath Output Selection

IDLE State: System In
STREAM State: Feedback In
RUNNING1 State: Datapath In
RUNNING2 State: Datapath In XOR IV

Feedback Output Selection

IDLE State: Datapath In XOR Feedback In
STREAM State: Datapath In XOR System In
RUNNING1 State: System In XOR Feedback In
RUNNING2 State: System In XOR IV

System Output Selection

System Output Source: System In

Start Signal Selection

Key Memory Start: No Key Memory Start
Stream Controller Start: Start Signal 4
Datapath Controller Start: Start Signal 7
System Output Controller Start: Start Signal 18

Counter Maximum Selection

Stream Counter: 4
Datapath Counter 1: 0
Datapath Counter 2: 127
System Output Counter: 64

Mode Enable Signals

☐ Enable Streaming Operation
☑ Enable 3-State Operation

Figure 6.16: Routing Node Configuration Form

The combo boxes for datapath output and feedback output allow the user to select what values will be output during each state of the main operation. Note that the four options presented do not match the actual states the controller goes through. The four actual states are RESET, IDLE, RUNNING1, and RUNNING2. However, the combo boxes present IDLE, STREAM, RUNNING1, and RUNNING2. The routing node behaviour for the actual RESET and default (non-streaming) IDLE states are selected by the IDLE State combo box. The STREAM State combo box selects the behaviour of the actual IDLE state when data is input in streaming mode. The RUNNING1 and RUNNING2 state combo boxes select the behaviour of the corresponding actual states.

The system output if fixed for every state, but has the same eight options as the datapath output and feedback output. The outputs can be selected from the system input, feedback input, datapath input, datapath input XORed with the IV, datapath input XORed with feedback input, datapath input XORed with the system input, system input XORed with the feedback input, or the system input XORed with the IV. Note that when in streaming mode, the IV is replaced with the feedback storage register value.

There are two check boxes used to enable streaming operation and three-state operation of the main datapath controller. Three-state operation means that both RUNNING1 and RUNNING2 states will be used, whereas normally only RUNNING1 will be used.

There are four combo boxes to select the various start signals. The system-level address decoder generates start signals when it writes to a particular routing node. However, all of the start signals are available to all the routing nodes. Thus, the start signals that trigger the operation of the various controlling state machines (streaming controller, datapath controller, and system output controller) can be selected from any of the system's 11 possible start signals. Also, the signal that is sent to the PE-FUNC to start the key memory operation can also be selected, though it has an extra option whereby no start signal may be selected, thus preventing the key memory from shifting at all (which would be used in non-iterative implementations). The key memory may also be started by the input of valid data.

Lastly, there are four spinner controls to set the maximum value of the state machine counters. The stream counter selects how many clock cycles the stream controller waits before storing the feedback input into the feedback storage register. The system output counter selects how many clock cycles the controller waits after its start signal before validating the system output. The two datapath counters count the number of clock cycles the system remains in states RUNNING1 and RUNNING2. All the counters are limited to the range 0 to 127 (7-bit counters).

The routing node configuration form has standard controls at the top, as well as an additional control button, allowing the routing node configuration to be saved to a file. This ability to save just a single routing node configuration was included for testing and debugging purposes.

6.5 Summary of System Configuration

This chapter has presented the design, implementation, and operation of the SHERIF software configuration utility. This configuration software is vital to the ease-of-use of the SHERIF system, and is responsible for managing, validating, and compartmentalizing the configurations input by the user, as well as abstracting away from the actual hardware (to a limited extent). By providing a graphical user interface to represent the various hardware structures and limiting user options to only valid choices, the software configuration utility makes algorithm implementation relatively quick and simple. This will be seen in the next chapter, which shows the software configuration utility in use to create a sample implementation of the AES algorithm Rijndael [17].

Chapter 7

Functional Testing and Synthesis Results

Having designed a flexible cryptographic hardware platform and its associated configuration software, it is naturally desirable to test whether the prototype proof-of-concept implementation works. Ideally, with sufficient time, each of the algorithms considered in Chapter 3 would be implemented on the SHERIF cryptographic hardware module, but time and resource constraints precluded that possibility. Only a minimal amount of testing could be carried out, but such simple tests were very informative in identifying problems with system design and organization.

As will be seen in subsequent sections, the proof-of-concept implementation of the SHERIF cryptographic architecture is very large in terms of silicon area and the number of signals in the system. This made complete synthesis infeasible on the available workstations, and also made functional simulation difficult, since there were so many signals and so much logic to simulate. Thus, it was not possible to perform system-level functional testing, nor to provide the most optimized, accurate synthesis results. However, individual components of the system have been functionally tested independently of each other, a sample pipelined functional implementation of the AES algorithm Rijndael [17] has been simulated one round at a time, and synthesis has been performed on individual components, allowing conclusions to be drawn about overall system flexibility, area, and performance.

7.1 Functional Testing

In general, functional testing of the system proved difficult because of the large number of signals in the system. This is not always a crippling problem, but in the SHERIF system, configuration must be performed before operational simulation. Simulating the configuration process is very time-consuming, requiring more than 450 000 clock cycles of simulation before any valid operational simulation can be done. Such a simulation would require more than 12 hours of real-time simulation in a non-interactive mode.

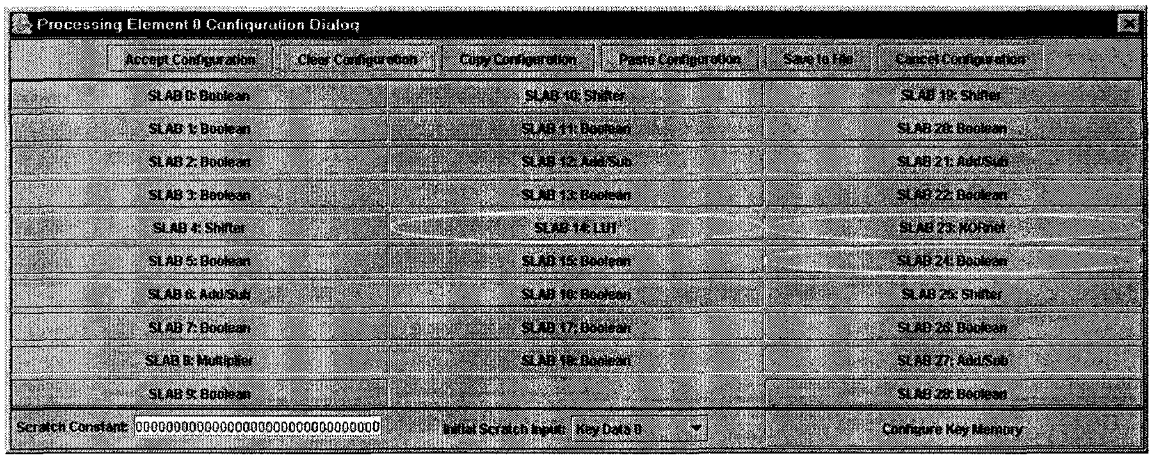
Fortunately, the individual PE-FUNCs and routing nodes can be simulated independently from the entire system in reasonable amounts of time. This means that an algorithm implementation on the SHERIF architecture can be tested by simulating each routing node and PEs independently, in sequence, verifying that each component works properly and forwards the correct data to the next component.

The AES algorithm Rijndael was chosen as the target for functional testing since it is a recent standard that will see frequent use. A pipelined implementation was chosen for its simplicity and greater throughput. The details of implementation and simulation are in the following subsections.

7.1.1 Pipelined Implementation of AES

The AES algorithm Rijndael [17] is an ideal candidate for implementation on the SHERIF architecture because its basic operations directly map onto the SLABs within the PE-FUNC. Round key addition is done via the XOR mode of the Boolean SLABs, substitutions by the LUT SLAB, and the XORnet implements the MixColumns() operation.

The first step in creating the pipelined AES implementation was to create a single round implementation in one of the PEs. A normal round of AES uses the SLABs



SLAB 0: Boolean
SLAB 1: Boolean
SLAB 2: Boolean
SLAB 3: Boolean
SLAB 4: Shifter
SLAB 5: Boolean
SLAB 6: Add/Sub
SLAB 7: Boolean
SLAB 8: Multiplier
SLAB 9: Boolean
SLAB 10: Shifter
SLAB 11: Boolean
SLAB 12: Add/Sub
SLAB 13: Boolean
SLAB 14: LUT
SLAB 15: Boolean
SLAB 16: Boolean
SLAB 17: Boolean
SLAB 18: Boolean
SLAB 19: Shifter
SLAB 20: Boolean
SLAB 21: Add/Sub
SLAB 22: Boolean
SLAB 23: XORnet
SLAB 24: Boolean
SLAB 25: Shifter
SLAB 26: Boolean
SLAB 27: Add/Sub
SLAB 28: Boolean

Scratch Constant: 00000000000000000000000000000000
Initial Scratch Input: Key Data 0

Figure 7.1: SLABs used in a normal AES round.

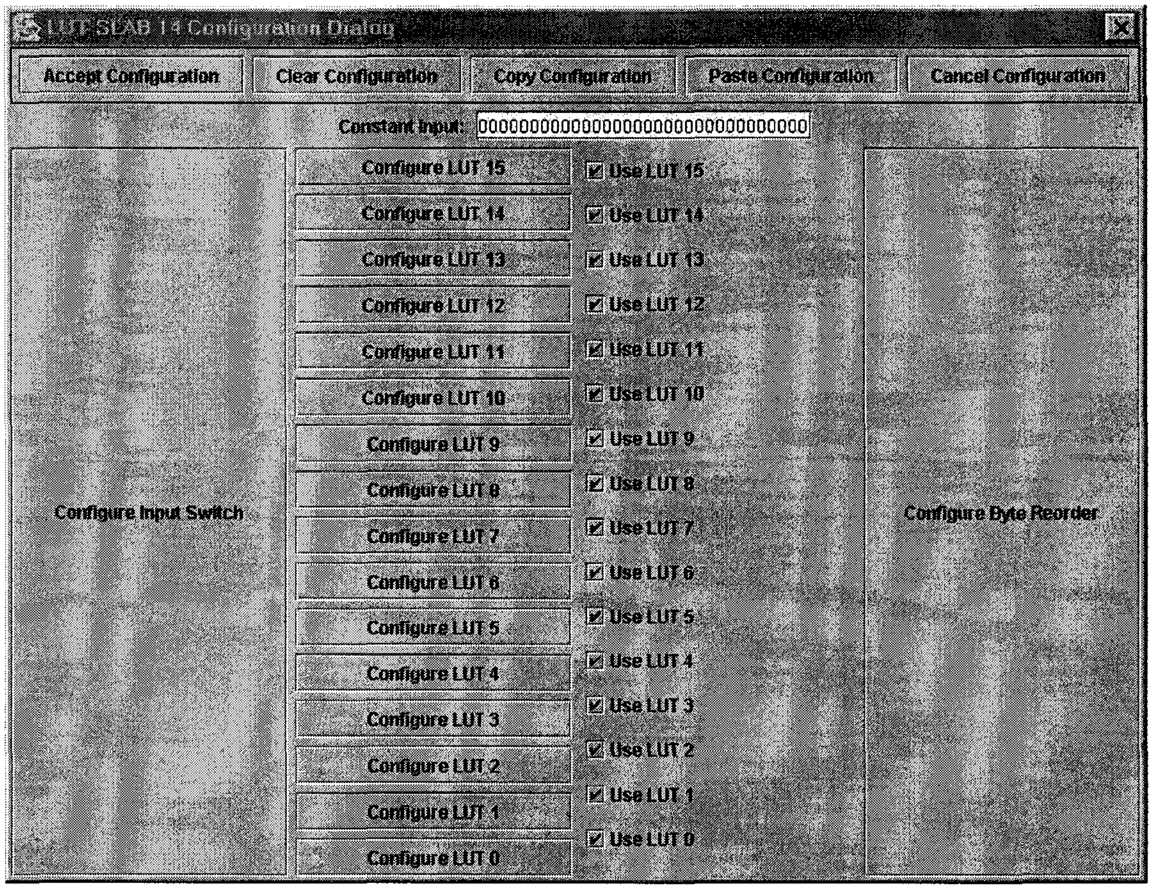
highlighted in Figure 7.1. Before implementing any of the algorithm's operations, all of the byte reorder components had to be manually set to pass through data directly, due to the slight design error discussed in Section 4.2.2.

The S-boxes are implemented directly in the LUT SLAB, in which all of the LUTs are used as shown in Figure 7.2. Each LUT has identical values, loaded in from a text file copied directly from the AES specification [17]. This is shown in Figure 7.3.

The ShiftRows() operation from the specification is just a byte reordering. Thus, the byte reordering component of the LUT SLAB can be used to implement this. The implementation is shown in Figure 7.4.

Unfortunately, the MixColumns() operation, defined as multiplication by a constant in a Galois field, is not so easy to implement. It is known from [60] and other sources that such operations *can* be implemented with a series of XORs, but the details of implementation are still needed before the XORnet SLAB can be used. The columns in the AES specification match up the each of the 32-bit words in the algorithm representation in Chapter 3.

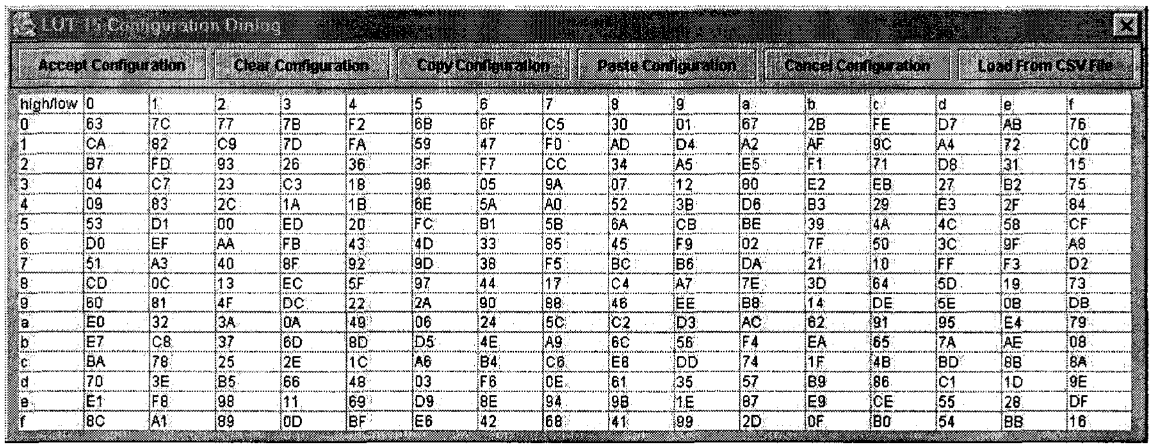
The operation is defined as follows, noting that the numbers inside curly braces



Constant Input: 00000000000000000000000000000000

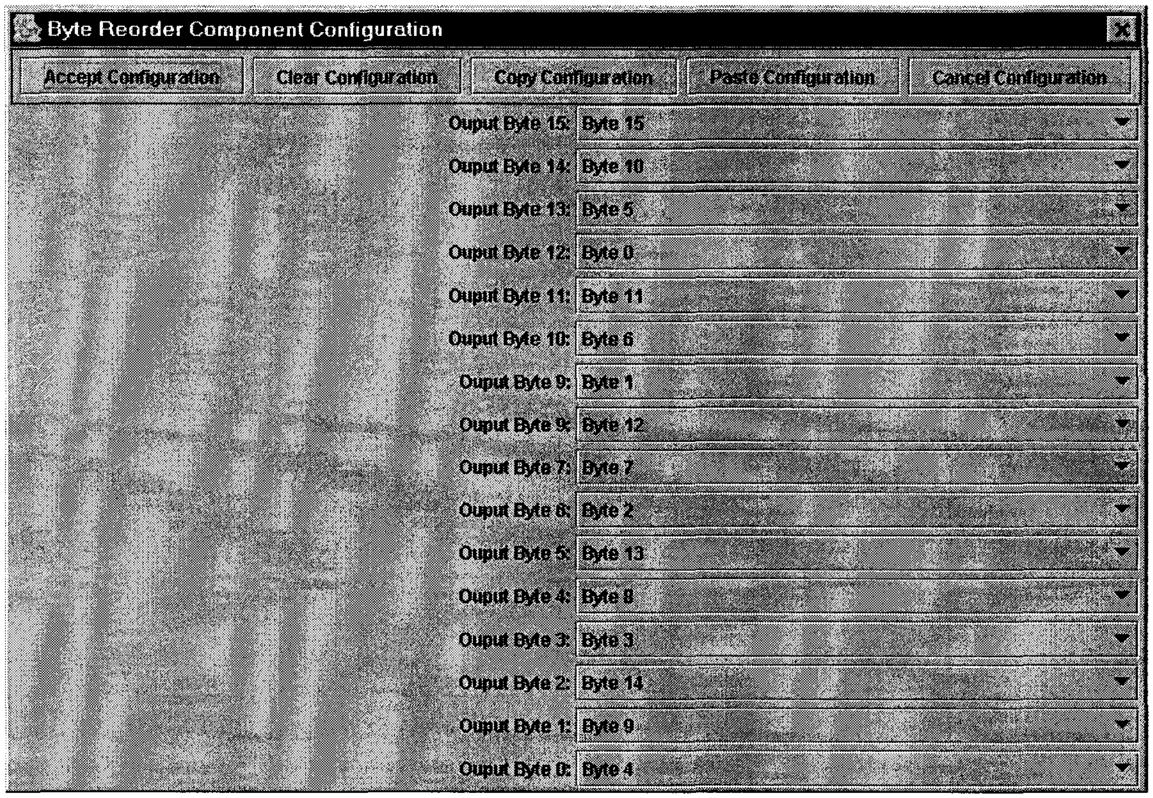
✓ Use LUT 15
✓ Use LUT 14
✓ Use LUT 13
✓ Use LUT 12
✓ Use LUT 11
✓ Use LUT 10
✓ Use LUT 9
✓ Use LUT 8
✓ Use LUT 7
✓ Use LUT 6
✓ Use LUT 5
✓ Use LUT 4
✓ Use LUT 3
✓ Use LUT 2
✓ Use LUT 1
✓ Use LUT 0

Figure 7.2: Setting the LUTs to active



high/low	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
a	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
b	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
c	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
d	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
e	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
f	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figure 7.3: Filling out the LUT



Ouput Byte 15:	Byte 15
Ouput Byte 14:	Byte 10
Ouput Byte 13:	Byte 5
Ouput Byte 12:	Byte 0
Ouput Byte 11:	Byte 11
Ouput Byte 10:	Byte 6
Ouput Byte 9:	Byte 1
Ouput Byte 9:	Byte 12
Ouput Byte 7:	Byte 7
Ouput Byte 6:	Byte 2
Ouput Byte 5:	Byte 13
Ouput Byte 4:	Byte 8
Ouput Byte 3:	Byte 3
Ouput Byte 2:	Byte 14
Ouput Byte 1:	Byte 9
Ouput Byte 0:	Byte 4

Figure 7.4: Byte Reordering implementing ShiftRows()

are hexadecimal values,

$$R = D \times \left[\{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}\right] \mod x^4 + 1 \tag{7.1}$$

where D and R are 32-bit values representing the data input to the MixColumns() operation and the result of the MixColumns() operation, respectively.

This can also be expressed as the following matrix operation [17], where r_3 to r_0 are the most significant to least significant bytes of R above, and d_3 to d_0 are the most significant to least significant bytes of D above:

$$\begin{bmatrix} r_3 \\ r_2 \\ r_1 \\ r_0 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} d_3 \\ d_2 \\ d_1 \\ d_0 \end{bmatrix} . \tag{7.2}$$

This can be multiplied out as per a normal matrix operation, at least insofar as multiplying rows by columns. However, it is also possible to take advantage of the fact that in the defined field, $\{03\} \cdot d = \{02\} \cdot d \oplus d$. This gives expressions for each byte of output for this operation:

$$\begin{aligned} r_3 &= \{02\} \cdot d_3 \oplus \{02\} \cdot d_2 \oplus d_2 \oplus d_1 \oplus d_0 \\ r_2 &= d_3 \oplus \{02\} \cdot d_2 \oplus \{02\} \cdot d_1 \oplus d_1 \oplus d_0 \\ r_1 &= d_3 \oplus d_2 \oplus \{02\} \cdot d_1 \oplus \{02\} \cdot d_0 \oplus d_0 \\ r_0 &= \{02\} \cdot d_3 \oplus d_3 \oplus d_2 \oplus d_1 \oplus \{02\} \cdot d_0 . \end{aligned}$$

In [64], a simple means of solving the $\{02\} \cdot d$ operation is described which is dependent on the value of d:

$$\{02\} \cdot d = \begin{cases} \left(d_{(6..0)} \,\&\, 0\right) & \text{if } d < \{80\} \\ \left(d_{(6..0)} \,\&\, 0\right) \oplus \{1\mathrm{B}\} & \text{if } d \geq \{80\} \end{cases}$$

where $d_{(6..0)}$ represents the seven least significant bits of the input byte d.

Thus, if the byte value is less than $\{80\}$, the operation is a simple left shift-in-0. If the byte value is greater than or equal to $\{80\}$, the result is the same shift, but then XORed with the value $\{1B\}$. Although this solution is defined in terms of bytes, the XORnet component can easily be used to compute the fixed shift required. Furthermore, noting that $d \geq \{80\}$ means that $d =$ 1XXXXXXX, so the MSB of the byte can be used to determine whether or not the XOR with $\{1B\}$ is needed, and can in fact be used directly to implement it in the XORnet by XORing the MSB with the appropriate bits so that it forms the value $\{1B\}$ when the MSB is 1. Thus, the four equations can be written out bitwise as follows, where $d_{(i)}$ represents bit position i of input data word D from Equation 7.1, $r_{j(k)}$ represents bit position k in byte r_j of result R from Equation 7.1, and $R_{(n)}$ represents bit position n of output result R from Equation 7.1:

$$
\begin{aligned}
R_{(31)} = r_{3(7)} &= d_{(23)} \oplus d_{(15)} \oplus d_{(7)} \oplus d_{(30)} \oplus d_{(22)} \\
R_{(30)} = r_{3(6)} &= d_{(22)} \oplus d_{(14)} \oplus d_{(6)} \oplus d_{(29)} \oplus d_{(21)} \\
R_{(29)} = r_{3(5)} &= d_{(21)} \oplus d_{(13)} \oplus d_{(5)} \oplus d_{(28)} \oplus d_{(20)} \\
R_{(28)} = r_{3(4)} &= d_{(20)} \oplus d_{(12)} \oplus d_{(4)} \oplus d_{(27)} \oplus d_{(19)} \oplus d_{(31)} \oplus d_{(23)} \\
R_{(27)} = r_{3(3)} &= d_{(19)} \oplus d_{(11)} \oplus d_{(3)} \oplus d_{(26)} \oplus d_{(18)} \oplus d_{(31)} \oplus d_{(23)} \\
R_{(26)} = r_{3(2)} &= d_{(18)} \oplus d_{(10)} \oplus d_{(2)} \oplus d_{(25)} \oplus d_{(17)} \\
R_{(25)} = r_{3(1)} &= d_{(17)} \oplus d_{(9)} \oplus d_{(1)} \oplus d_{(24)} \oplus d_{(16)} \oplus d_{(31)} \oplus d_{(23)} \\
R_{(24)} = r_{3(0)} &= d_{(16)} \oplus d_{(8)} \oplus d_{(0)} \oplus d_{(31)} \oplus d_{(23)}
\end{aligned}
$$

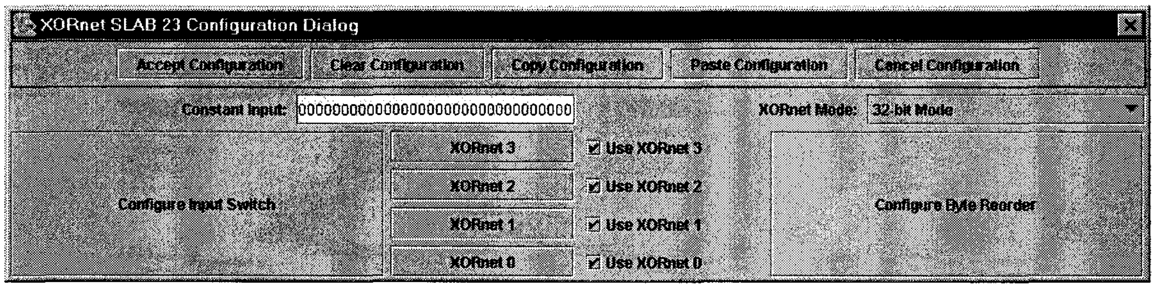
$$
\begin{aligned}
R_{(23)} = r_{2(7)} &= d_{(31)} \oplus d_{(15)} \oplus d_{(7)} \oplus d_{(22)} \oplus d_{(14)} \\
R_{(22)} = r_{2(6)} &= d_{(30)} \oplus d_{(14)} \oplus d_{(6)} \oplus d_{(21)} \oplus d_{(13)} \\
R_{(21)} = r_{2(5)} &= d_{(29)} \oplus d_{(13)} \oplus d_{(5)} \oplus d_{(20)} \oplus d_{(12)} \\
R_{(20)} = r_{2(4)} &= d_{(28)} \oplus d_{(12)} \oplus d_{(4)} \oplus d_{(19)} \oplus d_{(11)} \oplus d_{(23)} \oplus d_{(15)} \\
R_{(19)} = r_{2(3)} &= d_{(27)} \oplus d_{(11)} \oplus d_{(3)} \oplus d_{(18)} \oplus d_{(10)} \oplus d_{(23)} \oplus d_{(15)} \\
R_{(18)} = r_{2(2)} &= d_{(26)} \oplus d_{(10)} \oplus d_{(2)} \oplus d_{(17)} \oplus d_{(9)} \\
R_{(17)} = r_{2(1)} &= d_{(25)} \oplus d_{(9)} \oplus d_{(1)} \oplus d_{(16)} \oplus d_{(8)} \oplus d_{(23)} \oplus d_{(15)} \\
R_{(16)} = r_{2(0)} &= d_{(24)} \oplus d_{(8)} \oplus d_{(0)} \oplus d_{(23)} \oplus d_{(15)}
\end{aligned}
$$

$$
\begin{aligned}
R_{(15)} = r_{1(7)} &= d_{(31)} \oplus d_{(23)} \oplus d_{(7)} \oplus d_{(14)} \oplus d_{(6)} \\
R_{(14)} = r_{1(6)} &= d_{(30)} \oplus d_{(22)} \oplus d_{(6)} \oplus d_{(13)} \oplus d_{(5)} \\
R_{(13)} = r_{1(5)} &= d_{(29)} \oplus d_{(21)} \oplus d_{(5)} \oplus d_{(12)} \oplus d_{(4)} \\
R_{(12)} = r_{1(4)} &= d_{(28)} \oplus d_{(20)} \oplus d_{(4)} \oplus d_{(11)} \oplus d_{(3)} \oplus d_{(15)} \oplus d_{(7)} \\
R_{(11)} = r_{1(3)} &= d_{(27)} \oplus d_{(19)} \oplus d_{(3)} \oplus d_{(10)} \oplus d_{(2)} \oplus d_{(15)} \oplus d_{(7)} \\
R_{(10)} = r_{1(2)} &= d_{(26)} \oplus d_{(18)} \oplus d_{(2)} \oplus d_{(9)} \oplus d_{(1)} \\
R_{(9)} = r_{1(1)} &= d_{(25)} \oplus d_{(17)} \oplus d_{(1)} \oplus d_{(8)} \oplus d_{(0)} \oplus d_{(15)} \oplus d_{(7)} \\
R_{(8)} = r_{1(0)} &= d_{(24)} \oplus d_{(16)} \oplus d_{(0)} \oplus d_{(15)} \oplus d_{(7)}
\end{aligned}
$$

$$
\begin{aligned}
R_{(7)} = r_{0(7)} &= d_{(31)} \oplus d_{(23)} \oplus d_{(15)} \oplus d_{(30)} \oplus d_{(6)} \\
R_{(6)} = r_{0(6)} &= d_{(30)} \oplus d_{(22)} \oplus d_{(14)} \oplus d_{(29)} \oplus d_{(5)} \\
R_{(5)} = r_{0(5)} &= d_{(29)} \oplus d_{(21)} \oplus d_{(13)} \oplus d_{(28)} \oplus d_{(4)} \\
R_{(4)} = r_{0(4)} &= d_{(28)} \oplus d_{(20)} \oplus d_{(12)} \oplus d_{(27)} \oplus d_{(3)} \oplus d_{(31)} \oplus d_{(7)} \\
R_{(3)} = r_{0(3)} &= d_{(27)} \oplus d_{(19)} \oplus d_{(11)} \oplus d_{(26)} \oplus d_{(2)} \oplus d_{(31)} \oplus d_{(7)} \\
R_{(2)} = r_{0(2)} &= d_{(26)} \oplus d_{(18)} \oplus d_{(10)} \oplus d_{(25)} \oplus d_{(1)} \\
R_{(1)} = r_{0(1)} &= d_{(25)} \oplus d_{(17)} \oplus d_{(9)} \oplus d_{(24)} \oplus d_{(0)} \oplus d_{(31)} \oplus d_{(7)} \\
R_{(0)} = r_{0(0)} &= d_{(24)} \oplus d_{(16)} \oplus d_{(8)} \oplus d_{(31)} \oplus d_{(7)} \ .
\end{aligned}
$$

The top level configuration of the XORnet SLAB is shown in Figure 7.5, whereas the implementation of the MixColumns() operation itself is shown in Figure 7.6.

The round key addition is quite straightforward. The input switch of a Boolean

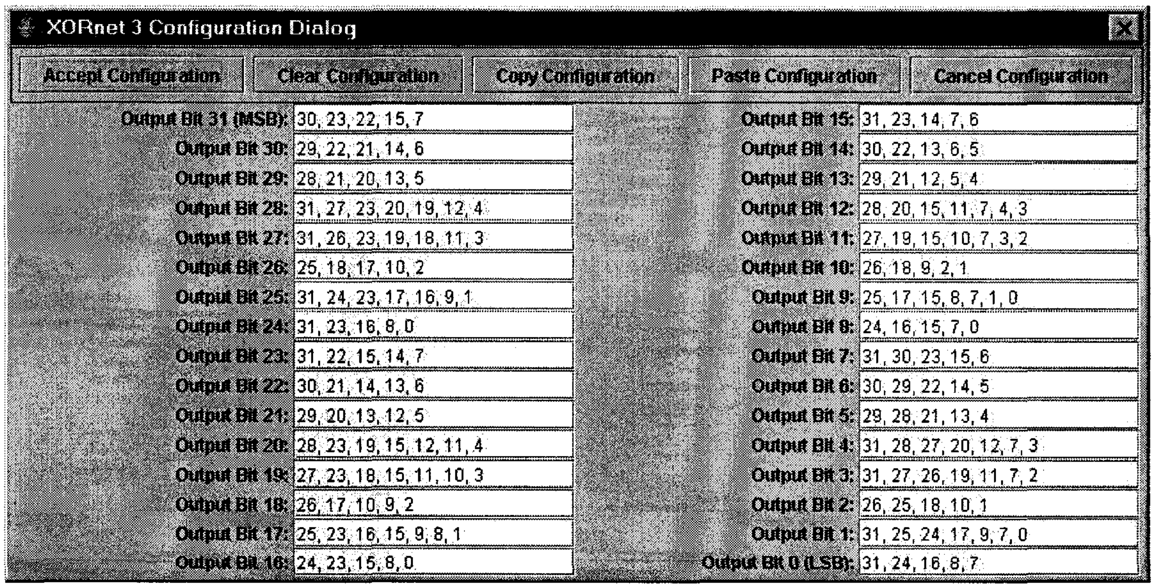


Constant Input: 00000000000000000000000000000000

XORnet Mode: 32-bit Mode

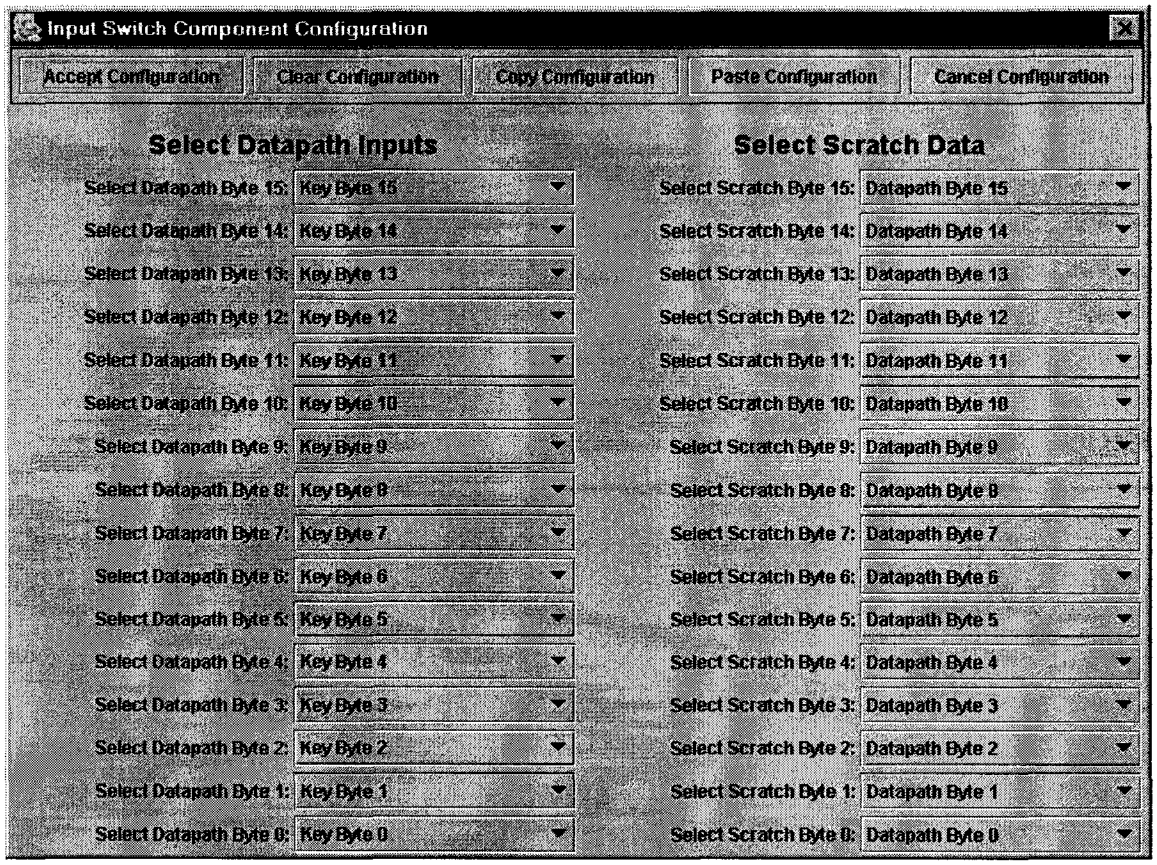
☑ Use XORnet 3
☑ Use XORnet 2
☑ Use XORnet 1
☑ Use XORnet 0

Figure 7.5: Top-level XORnet SLAB selection



Output Bit 31 (MSB):	30, 23, 22, 15, 7	Output Bit 15:	31, 23, 14, 7, 6
Output Bit 30:	29, 22, 21, 14, 6	Output Bit 14:	30, 22, 13, 6, 5
Output Bit 29:	28, 21, 20, 13, 5	Output Bit 13:	29, 21, 12, 5, 4
Output Bit 28:	31, 27, 23, 20, 19, 12, 4	Output Bit 12:	28, 20, 15, 11, 7, 4, 3
Output Bit 27:	31, 26, 23, 19, 18, 11, 3	Output Bit 11:	27, 19, 15, 10, 7, 3, 2
Output Bit 26:	25, 18, 17, 10, 2	Output Bit 10:	26, 18, 9, 2, 1
Output Bit 25:	31, 24, 23, 17, 16, 9, 1	Output Bit 9:	25, 17, 15, 8, 7, 1, 0
Output Bit 24:	31, 23, 16, 8, 0	Output Bit 8:	24, 16, 15, 7, 0
Output Bit 23:	31, 22, 15, 14, 7	Output Bit 7:	31, 30, 23, 15, 6
Output Bit 22:	30, 21, 14, 13, 6	Output Bit 6:	30, 29, 22, 14, 5
Output Bit 21:	29, 20, 13, 12, 5	Output Bit 5:	29, 28, 21, 13, 4
Output Bit 20:	28, 23, 19, 15, 12, 11, 4	Output Bit 4:	31, 28, 27, 20, 12, 7, 3
Output Bit 19:	27, 23, 18, 15, 11, 10, 3	Output Bit 3:	31, 27, 26, 19, 11, 7, 2
Output Bit 18:	26, 17, 10, 9, 2	Output Bit 2:	26, 25, 18, 10, 1
Output Bit 17:	25, 23, 16, 15, 9, 8, 1	Output Bit 1:	31, 25, 24, 17, 9, 7, 0
Output Bit 16:	24, 23, 15, 8, 0	Output Bit 0 (LSB):	31, 24, 16, 8, 7

Figure 7.6: XORnet configuration for MixColumns()



Select Datapath Inputs

Select Datapath Byte 15: Key Byte 15
Select Datapath Byte 14: Key Byte 14
Select Datapath Byte 13: Key Byte 13
Select Datapath Byte 12: Key Byte 12
Select Datapath Byte 11: Key Byte 11
Select Datapath Byte 10: Key Byte 10
Select Datapath Byte 9: Key Byte 9
Select Datapath Byte 8: Key Byte 8
Select Datapath Byte 7: Key Byte 7
Select Datapath Byte 6: Key Byte 6
Select Datapath Byte 5: Key Byte 5
Select Datapath Byte 4: Key Byte 4
Select Datapath Byte 3: Key Byte 3
Select Datapath Byte 2: Key Byte 2
Select Datapath Byte 1: Key Byte 1
Select Datapath Byte 0: Key Byte 0

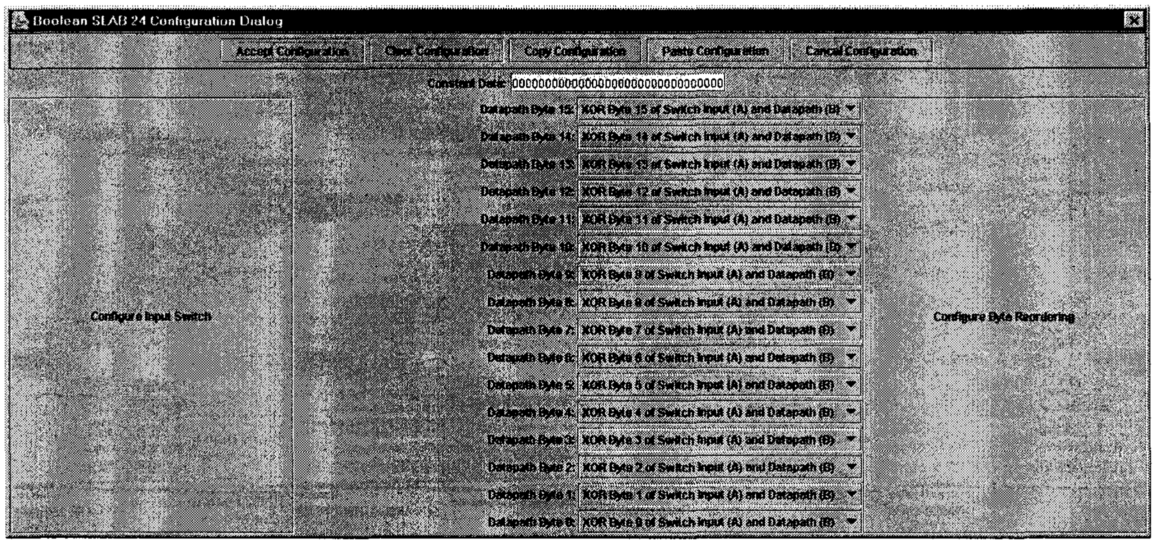
Select Scratch Data

Select Scratch Byte 15: Datapath Byte 15
Select Scratch Byte 14: Datapath Byte 14
Select Scratch Byte 13: Datapath Byte 13
Select Scratch Byte 12: Datapath Byte 12
Select Scratch Byte 11: Datapath Byte 11
Select Scratch Byte 10: Datapath Byte 10
Select Scratch Byte 9: Datapath Byte 9
Select Scratch Byte 8: Datapath Byte 8
Select Scratch Byte 7: Datapath Byte 7
Select Scratch Byte 6: Datapath Byte 6
Select Scratch Byte 5: Datapath Byte 5
Select Scratch Byte 4: Datapath Byte 4
Select Scratch Byte 3: Datapath Byte 3
Select Scratch Byte 2: Datapath Byte 2
Select Scratch Byte 1: Datapath Byte 1
Select Scratch Byte 0: Datapath Byte 0

Figure 7.7: Round key addition input switch configuration

SLAB is configured to select all key bytes, and then all of the gates are set to the XOR operation. Thus, the key bytes are XORed with the incoming datapath bytes. The configuration of the input switch is shown in Figure 7.7, and the configuration of the Boolean logic itself in Figure 7.8. This finishes the implementation of a single round of Rijndael.

The above round implementation is copied and pasted into each of the other PEs. Once that was done, to account for the initial key addition and the lack of MixColumns() in the final round, slight changes were made to the configurations of the first PE and the last PE. In the first PE, to handle the initial key addition, another Boolean SLAB (SLAB 0) was used to add in the first round key. This also required changing the key memory output to that SLAB from key 0 to key 1, and



Constant Data: 00000000000000000000000000000000
Datapath Byte 15: XOR Byte 15 of Switch Input (A) and Datapath (B)
Datapath Byte 14: XOR Byte 14 of Switch Input (A) and Datapath (B)
Datapath Byte 13: XOR Byte 13 of Switch Input (A) and Datapath (B)
Datapath Byte 12: XOR Byte 12 of Switch Input (A) and Datapath (B)
Datapath Byte 11: XOR Byte 11 of Switch Input (A) and Datapath (B)
Datapath Byte 10: XOR Byte 10 of Switch Input (A) and Datapath (B)
Datapath Byte 9: XOR Byte 9 of Switch Input (A) and Datapath (B)
Datapath Byte 8: XOR Byte 8 of Switch Input (A) and Datapath (B)
Datapath Byte 7: XOR Byte 7 of Switch Input (A) and Datapath (B)
Datapath Byte 6: XOR Byte 6 of Switch Input (A) and Datapath (B)
Datapath Byte 5: XOR Byte 5 of Switch Input (A) and Datapath (B)
Datapath Byte 4: XOR Byte 4 of Switch Input (A) and Datapath (B)
Datapath Byte 3: XOR Byte 3 of Switch Input (A) and Datapath (B)
Datapath Byte 2: XOR Byte 2 of Switch Input (A) and Datapath (B)
Datapath Byte 1: XOR Byte 1 of Switch Input (A) and Datapath (B)
Datapath Byte 0: XOR Byte 0 of Switch Input (A) and Datapath (B)

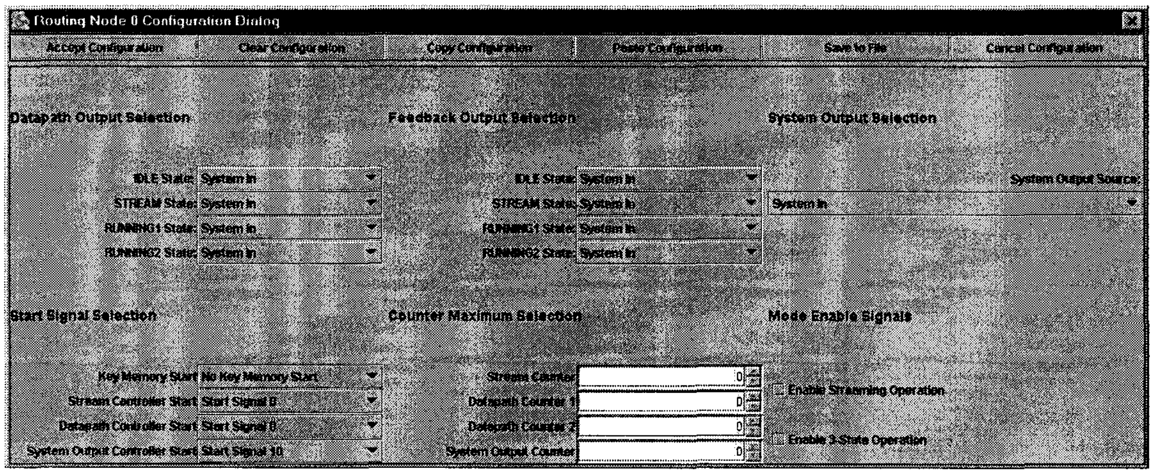
Figure 7.8: Boolean SLAB configured for AES round key addition

thus also requires the external system responsible for generating the subkeys to write the initial key *after* the regular round key. In the final round PE, the XORnet was disabled to bypass the final MixColumns().

With the rounds implemented, all that remains is to configure the routing nodes. Routing node configuration is fairly straightforward for pipelined implementations, as they do not have to handle any iterative data. Thus, the first routing node must simply be configured to put the system input onto the datapath for all possible states, to ensure the data written into the system gets processed. This is shown in Figure 7.9.

Routing nodes 1 through 9 are all configured identically, and share the simple task of passing the datapath input to the datapath output in all modes of operation. This configuration is shown in Figure 7.10.

The final routing node is responsible for sending the encrypted data to the system output. The system output controller is responsible for ensuring only valid, completely processed data gets sent to the system output. As such, it is based on a state machine and counter that counts the number of cycles from the start of processing,



Datapath Output Selection

IDLE State: System In
STREAM State: System In
RUNNING1 State: System In
RUNNING2 State: System In

Feedback Output Selection

IDLE State: System In
STREAM State: System In
RUNNING1 State: System In
RUNNING2 State: System In

System Output Selection

System Output Source: System In

Start Signal Selection

Key Memory Start: No Key Memory Start
Stream Controller Start: Start Signal 0
Datapath Controller Start: Start Signal 0
System Output Controller Start: Start Signal 10

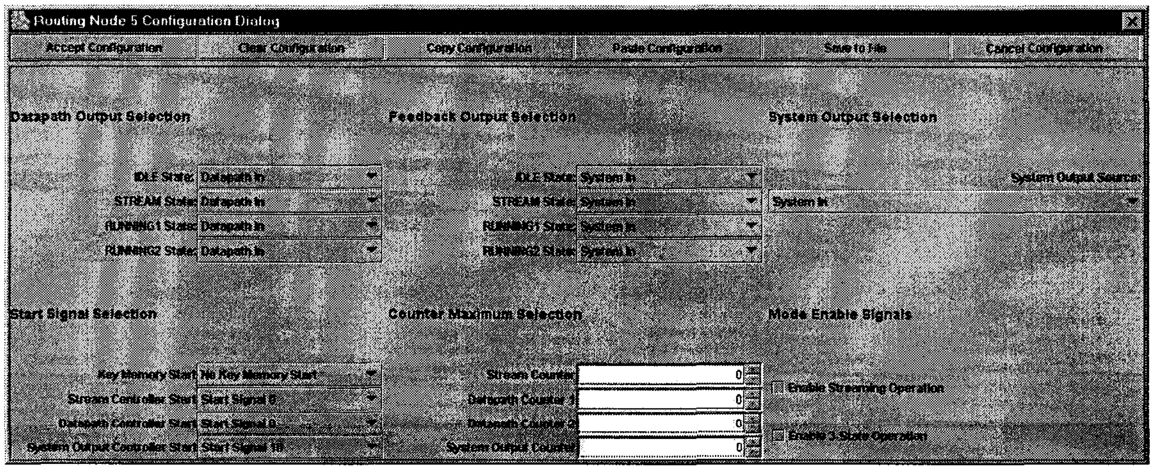
Counter Maximum Selection

Stream Counter: 0
Datapath Counter 1: 0
Datapath Counter 2: 0
System Output Counter: 0

Mode Enable Signals

☐ Enable Streaming Operation
☐ Enable 3-State Operation

Figure 7.9: Initial routing node for AES



Datapath Output Selection

IDLE State: Datapath In
STREAM State: Datapath In
RUNNING1 State: Datapath In
RUNNING2 State: Datapath In

Feedback Output Selection

IDLE State: System In
STREAM State: System In
RUNNING1 State: System In
RUNNING2 State: System In

System Output Selection

System Output Source: System In

Start Signal Selection

Key Memory Start: No Key Memory Start
Stream Controller Start: Start Signal 0
Datapath Controller Start: Start Signal 0
System Output Controller Start: Start Signal 10

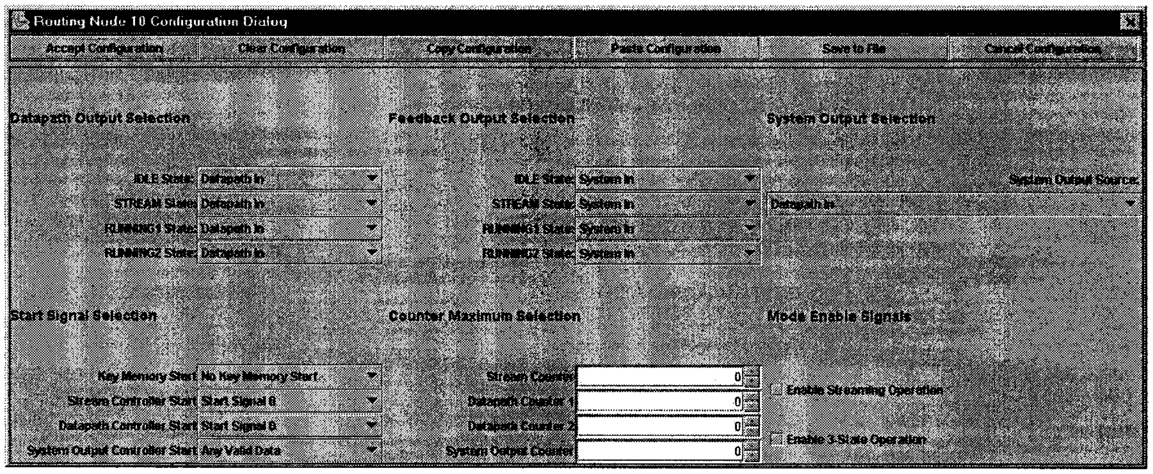
Counter Maximum Selection

Stream Counter: 0
Datapath Counter 1: 0
Datapath Counter 2: 0
System Output Counter: 0

Mode Enable Signals

☐ Enable Streaming Operation
☐ Enable 3-State Operation

Figure 7.10: Middle routing nodes for AES



Datapath Output Selection
IDLE State: Datapath In
STREAM State: Datapath In
RUNNING1 State: Datapath In
RUNNING2 State: Datapath In

Feedback Output Selection
IDLE State: System In
STREAM State: System In
RUNNING1 State: System In
RUNNING2 State: System In

System Output Selection
System Output Source: Datapath In

Start Signal Selection
Key Memory Start: No Key Memory Start
Stream Controller Start: Start Signal 0
Datapath Controller Start: Start Signal 0
System Output Controller Start: Any Valid Data

Counter Maximum Selection
Stream Counter: 0
Datapath Counter 1: 0
Datapath Counter 2: 0
System Output Counter: 0

Mode Enable Signals
☐ Enable Streaming Operation
☐ Enable 3-State Operation

Figure 7.11: Final routing node for AES

and then passes any valid data in the routing node to the system output at the end of that time. This behaviour of the system output controller is not desirable for a pipelined implementation, however, since it can only be counting from the start of one block of data being processed. To support the high throughput of pipelined implementation, the start signal for the system output controller must be selected as any valid data input, and the maximum count set to 0 to ensure that the controller will validate any valid data that comes into the routing node that is destined for the system output. This configuration is shown in Figure 7.11.

The configurations of each PE and routing node were saved to separate files for piecewise simulation. The entire configuration was also saved for future simulation.

7.1.2 Simulation of Pipelined AES

Once the configuration files for each PE and routing node were generated, they could be simulated independently of each other, using test vectors provided in Appendix C of the AES specification [17]. The provided test vectors are especially useful since they show the result of every operation performed during encryption, and thus subkey values and the data values after each round are available. The given test vector for encryption takes a plaintext block 0x00112233445566778899AABBCCDDEEFF (the "0x" prefix denotes hexadecimal notation) and a key 0x000102030405060708090A0B0C0D0E0F to produce a ciphertext 0x69C4E0D86A7B0430D8CDB78070B4C55A.

To verify the pipelined implementation of AES, first each of the routing nodes was simulated independently to verify the correct operation. In a pipelined implementation, the operation of the routing nodes is trivial, and it was easily verified that the data is routed correctly by each of the routing node configurations. Verification of the PEs themselves would be a much more difficult task.

Simulation of the PEs first required the PE to be configured from the prepared configuration files. Configuration was a slower process to simulate for the AES configuration than for earlier component tests since the greater variety of the configuration data caused a lot more signal switching to be simulated, thus slowing down execution time. In fact, more than 99% of the simulation time was for configuration.

Once configuration was completed, the test vectors were applied. First, subkey values were written in to the PE's key memory. This required thirteen clock cycles. After that, valid test vectors could be applied, with valid output data appearing once clock cycle later due to latency introduced by registers at the inputs and outputs of the PE-FUNC.

For example, to simulate the first round of the pipelined AES implementation, a PE was configured using the data saved from the PE0 configuration form in the software configuration utility. Then, the VHDL testbench applied key values: 0xD6AA74FDD2AF72FADAA678F1D6AB76FE, followed by 0x000102030405060708090A0B0C0D0E0F twelve times. Only the first of those twelve repeated values are actually used; the rest are needed to shift the valid key data into the proper location in the key memory register, and could just as easily be all zeros. However, if simulating the entire SHERIF system, using the same second key value (which is actually the first subkey applied) as padding allows some slight efficiency since the same padding value can be written to the other key memories at once, even though they only require one subkey value. Thus, each PE's key memory would have a unique subkey value written in, and then all PEs would have the padding/first key value written twelve times.

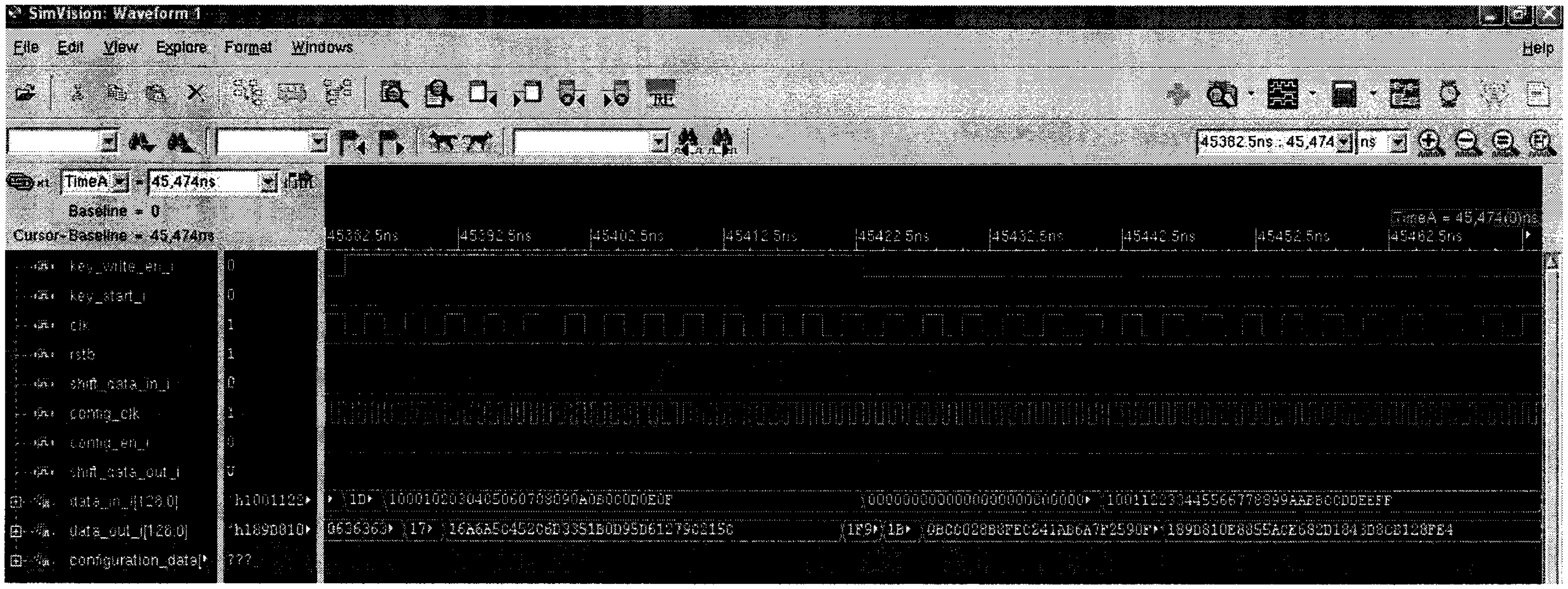
Once the key data has been configured, the input test vector can be applied as valid data. Thus 0x00112233445566778899AABBCCDDEEFF is input to the system with an extra valid bit prepended to it, and one clock cycle later the output value should be 0x89D810E8855ACE682D1843D8CB128FE4 (also with a valid bit prepended). As can be seen in the simulation waveform results in Figure 7.12, the configuration for the first round of the AES algorithm simulates successfully.

Subsequent rounds of the pipelined implementation were simulated in a similar fashion, using the subkey values and intermediate values from the example vectors in [17]. They were simulated in non-interactive mode to speed up the simulation time, and stored their results in text files to allow verification of correct operation. For each of the remaining nine rounds, operation was verified against the expected results from the example vectors in [17], with the output of the final round simulation being 0x69C4E0D86A7B0430D8CDB78070B4C55A (with a valid bit prepended).

Thus, even though it was not possible to simulate the operation of the whole system, it is still possible to consider the PE datapath verified for a pipelined implementation of the AES algorithm Rijndael.

7.2 Synthesis Results

While functional simulation proved difficult on the available computing resources, complete synthesis proved virtually impossible. Early attempts at synthesizing a PE-FUNC ran for nearly two weeks with no result, and ultimately the synthesis process



```
key_write_en_i          0
key_start_i             0
clk                     1
rstb                    1
shift_data_in_i         0
config_clk              1
config_en_i             0
shift_data_out_i        U
data_in_i[128:0]        'h1001122▸   ▸ 1D▸ 1000102030405060708090A0B0C0D0E0F   000000000000000000000000▸ 100112233445566778899AABBCCDDEEFF
data_out_i[128:0]       'h189D810▸   0636363▸ 17▸ 16A6A5C45208D3351B0D95D6127902156   1F9▸ 1B▸ 0BC0028B8FEC241AB6A7F2590F▸ 189D810E8855AC E682D1843D8CB128FE4
configuration_data[▸    ???
```

Figure 7.12: Waveform for Simulation of AES Round 1

was killed to free resources for other tasks. The synthesis attempt took so long because the PE-FUNC is primarily an extremely large pool of combinational logic (estimated at more than 800 layers of logic gates), with only the inputs and outputs registered. Synthesis tools typically have difficulty synthesizing such large designs.

The normal route to synthesizing large designs is to synthesize smaller components first, then use those as building blocks for synthesizing the total design. Unfortunately, the PE-FUNC is the smallest element with registered I/O, and so synthesis of its component SLABs individually would be sub-optimal since there would be no way to constrain the speed of the components without having any sequential logic. Adding more pipelining to the internal stages of the PE-FUNC and thereby breaking it into several components might allow successful synthesis of the entire PE-FUNC, but that does not help synthesis of the proof-of-concept design currently implemented.

While more powerful computing resources could potentially complete synthesis of the existing PE-FUNC, the difficulty of doing so suggests the design is *extremely* large and inefficient in terms of area, and that improvements in this respect would be desirable. Still, it is necessary to at least find area and speed estimates for the existing design, so a compromise is made.

Each of the basic components of the system was synthesized individually, but with their inputs and outputs registered to allow for constraining the timing of each component. A set of registers only was also synthesized, so that the area of the registers can be subtracted out of the area results for each component to give an approximate area of the unregistered components as they would appear in the actual PE-FUNC.

To facilitate this synthesis, VHDL code was written to instantiate each component of interest as a registered component. These "stub" components were then synthesized. To make the synthesis of these stubs easier, all of the configuration registers (which are already easily-constrained sequential logic) were synthesized first, as was the 8×8 LUT, which is primarily configuration register logic with a large combinational multiplexer on the outputs.

The following components were synthesized. The stub components are registered versions of the SLABs in the case of the basic operation components, and thus encapsulate the configuration register of the component, the core logic, the input switch, and the byte reordering. The other stubs are just registered versions of the named components. The input switch and byte reordering were also synthesized as stubs to allow an idea of how much area *they* consume overall.

- register_stub (to allow subtracting out of register area)
- config_register_47bit
- config_register_73bit
- config_register_130bit
- config_register_244bit
- config_register_252bit
- config_register_256bit
- config_register_288bit
- config_register_324bit
- config_register_4341bit
- lut_8x8
- address_decoder_stub
- output_scheduler_stub
- routing_node_stub

- key_memory_stub
- boolean_stub
- shifter_stub
- adder_stub
- multiplier_stub
- xornet_stub
- lut_stub
- input_switch_stub
- byte_reorder_stub

Results and analysis of this test synthesis are presented in the following subsections.

7.2.1 Area

All synthesis was performed using 0.18 μm CMOS technology. All area values are given in square microns (μm^2). Approximate gate counts are determined by dividing the total area by 12.97 (μm^2), which is an experimentally determined value for the area of a 2-input NAND gate in the target technology.

First, a set of input/output registers was synthesized to provide reference values. The synthesized register_stub component had three 128-bit input registers and two 128-bit output registers, and synthesized with a total cell area of 54121 (approximately 4172 gates), which suggests a single 128-bit register would have an area of approximately 10825 μm^2 (approximately 835 gates) and a single 1-bit register would have an area of approximately 85 μm^2 (roughly 7 gates). These values can be used to subtract out the area of the registers from the other synthesized stub components.

Configuration Register:	**Area (μm^2):**	**Gates:**
config_register_47bit	15022	1158
config_register_73bit	25003	1927
config_register_130bit	40485	3121
config_register_244bit	75949	5855
config_register_252bit	78555	6056
config_register_256bit	79921	6162
config_register_288bit	89044	6865
config_register_324bit	100582	7755
config_register_4341bit	1374537	105978
lut_8x8	234537	18083

Table 7.1: Configuration Register Synthesis Results (Area)

Next, the configuration registers were synthesized independently. The area results of this synthesis are given in Table 7.1. Note that synthesis results for the independently-synthesized 8×8 LUT are also included here, since the LUT is essentially a configuration register.

The area synthesis results for each of the component stubs are shown in Table 7.2. The table also shows estimated areas for the core logic by subtracting out the area of the input and output registers added to the stub components.

The area for a single PE-FUNC can be estimated from the component areas. Each PE-FUNC contains a 130-bit configuration register, a key memory, 18 Boolean SLABs, 4 Add/Subtract SLABs, 4 Shifter SLABs, 1 Multiplier SLAB, 1 LUT SLAB, and 1 XORnet SLAB, 2 129-bit input/output registers, plus an input multiplexer on the scratch path. The scratch path input multiplexer will be smaller than the input switch with each SLAB, but for the sake estimating the area it contributes to the

Component Stub:	Total Area (μm^2):	Register Area:	Core Area:	Core Gates:
input_switch_stub	96273	69004	27268	2102
byte_reorder_stub	79490	27060	52430	4042
address_decoder_stub	6988	4397	2591	199
output_scheduler_stub	332465	130906	201559	15540
routing_node_stub	442203	66806	375397	28943
key_memory_stub	1519422	324897	1194524	92099
boolean_stub	319784	54121	265662	20482
shifter_stub	319641	54121	265520	20471
adder_stub	353601	54121	299480	23090
multiplier_stub	619822	54121	565700	43616
xornet_stub	1778557	54121	1724436	132955
lut_stub	4039512	54121	3985391	307277

Table 7.2: Component Stub Synthesis Results (Area)

overall PE-FUNC, the input switch area will be used.

$$\begin{aligned}\text{Total PE-FUNC Area} &= (18 \times 265662) + (4 \times 265520) \\ &\quad +(4 \times 299480) + (565700) \\ &\quad +(1724436) + (3985391) \\ &\quad +(40485) + (27268) + (21817) \\ &= 13407036\ \mu m^2 \\ &= \text{approximately } 1,033,696 \text{ gates}\end{aligned}$$

With an estimation of the total PE-FUNC area, it is possible to estimate the size of the entire system (designed to be capable of implementing a pipelined version of AES), based on 10 PEs, 11 routing nodes, the address decoder, output arbiter, and the need for registering 137 bits at the inputs and 129 bits at the outputs. The values for the routing nodes, address decoder, and output arbiter were shown in Table 7.2 along with the areas of the basic components used to calculate the PE-FUNC area.

$$\begin{aligned}\text{Total SHERIF Area} &= (10 \times 13407036) + (11 \times 375397) \\ &\quad +(2591) + (201559) + (22494) \\ &= 138426378\ \mu m^2 \\ &= \text{approximately } 10,672,813 \text{ gates}\end{aligned}$$

Thus, each PE-FUNC is approximately one million gates in size, and the whole system would be on the order of ten million gates.

7.2.2 Timing

Since it was not possible to synthesize the entire system, performance of the SHERIF cryptographic hardware module can only be estimated based on the timing analysis of the individually-synthesized components.

When synthesizing the stub components, the configuration clock was constrained to a 2 ns period, whereas the main clock was constrained to a 7 ns clock period. While synthesis was not always able to meet these requirements, reports on the slack (either positive or negative) allow estimation of the clock speed at which each component could successfully run.

Table 7.3 shows each synthesized component and its largest usable slack for both the configuration clock and main clock. A negative slack indicates that the timing requirement could not be met, and is exceeded by the given value. A positive slack indicates that the timing requirements were easily met, with the given value to spare. Thus, the latency of a component is given by the clock period minus the slack. Note that most of the configuration registers are incorporated into the larger components in the table, and thus their timing is given as part of their parent component.

The critical path for the entire system will be the PE-FUNC datapath, since it is essentially one large pool of combinational logic. The latency of the PE-FUNC datapath can be estimated from the component latencies in Table 7.3.

Component:	**Clock Period (ns):**	**Slack (ns):**	**Latency (ns):**
address_decoder_stub	7	0.57	6.43
output_scheduler_stub	7	-3.64	10.64
routing_node_stub	7	-0.48	7.48
	2	-4.23	6.23
key_memory_stub	7	0.61	6.39
	2	-4.16	6.16
boolean_stub	7	0.61	6.39
	2	-4.28	6.28
shifter_stub	7	0.57	6.43
	2	-4.30	6.30
adder_stub	7	-4.25	11.25
	2	-4.31	6.31
multiplier_stub	7	-3.36	10.36
	2	-4.29	6.29
xornet_stub	7	0.57	6.43
	2	-4.63	6.63
lut_stub	7	0.61	6.39
	2	-8.24	10.24

Table 7.3: Synthesis Results (Timing)

$$\begin{aligned}
\text{PE-FUNC latency} &= (18 \times \text{Boolean latency}) \\
&\quad +(4 \times \text{Shifter latency}) \\
&\quad +(4 \times \text{Add/Sub latency}) \\
&\quad +\text{Multiplier latency} \\
&\quad +\text{LUT latency} \\
&\quad +\text{XORnet latency} \\
&= (18 \times 6.39) + (4 \times 6.43) \\
&\quad +(4 \times 11.25) + 10.36 + 6.39 + 6.43 \\
&= 208.92 \text{ ns}
\end{aligned}$$

A latency of 208.92 ns suggests a that a datapath clock frequency of approximately 4.78 MHz would be the upper limit of the clock speed. Any other components running on the main clock will have more than enough time to complete operation with such a long clock period being used.

The configuration registers have a worst-case latency of just over 10 ns, so running the configuration clock with a period of 15 or 20 ns should be acceptable. Note that this means the configuration clock can run more than 10 times faster than the main clock, if so desired.

7.2.3 Synthesis Analysis

The implementation of the PE-FUNC is clearly the focus of this system, as it constrains the rest of the system to operate at its speed and consumes most of the area on the device. Most of the PE-FUNC area, however, comes not from the data processing hardware, but rather from configuration registers and data routing hardware.

The configuration registers alone make up approximately 20% of the total PE-FUNC area; if the LUTs are included in the count, the amount jumps to almost 59% of the area. The input switches and byte reorder components account for almost 17% of the PE-FUNC area. Thus, almost 76% of the device area is not directly used for data computation, but is necessary to offer the desired level of flexibility.

Using an estimated clock speed of 4.78 MHz, the pipelined implementation of the AES algorithm should achieve a throughput of 128 bits/cycle $\times$ 4.78 MHz $=$ 611.84 Mbps. This offers speed comparable to many software implementations, and certainly superior to software running on processors of equivalent speed, but is a little slow compared to some dedicated ASIC implementations or software running on high-end microprocessors.

7.3 Summary of Testing and Results

This chapter has discussed the simulation and synthesis results for the SHERIF cryptographic hardware architecture. While the vast size and complexity of the current architecture made it infeasible to do system-level functional simulation or synthesis, it was possible to simulate and synthesize the major components of the design in order to determine the feasibility of the overall design.

A sample implementation of a pipelined version of the AES algorithm Rijndael was used to demonstrate the operation of the SHERIF hardware architecture and some of its flexibility. While implementation of other algorithms would be desirable to fully verify the flexibility of the system, time constraints forced them to be relegated to future work.

The basic components of the SHERIF architecture were synthesized in 0.18 μm

CMOS technology, and based on component synthesis, the total area of the proof-of-concept SHERIF cryptographic hardware module is estimated to be 138426378 μm^2 or approximately 10.7 million gates for a system implemented to allow a fully pipelined implementation of the AES algorithm. By far the largest contributors to the area of the device are the PE-FUNCs, each of which has an area of 13407036 μm^2 or $1,033,696$ gates. The large size of the PE-FUNC constrains the speed of operation, limiting the system to running at approximately 4.78 MHz.

Overall, while the SHERIF architecture seems to offer the desired degree of flexibility in terms of its PEs, this flexibility comes at the cost of large area and low speed of operation. Future research efforts may find ways to overcome these problems.

Chapter 8

Future Directions and Conclusions

The design of the SHERIF cryptographic hardware module has been a challenging, but ultimately rewarding, project. While the SHERIF architecture has been designed to provide great flexibility in implementing cryptographic algorithms, it provides that flexibility at the expense of area and speed. Nevertheless, the current version of the SHERIF architecture provides a basis on which future, more powerful and more efficient versions may be built, and has suggested several directions in which such research may go.

8.1 Conclusion

The primary focus of this research has been on the design of a flexible cryptographic hardware module. Implementation of the SHERIF cryptographic hardware module has been done solely as a proof-of-concept for the design ideas presented, and as such its performance in terms of area and speed is sub-optimal.

Careful analysis of several leading block cipher and hash function algorithms led to the identification of six basic operations that are sufficient to implement a wide variety of algorithms. Functional components were designed and implemented to allow these operations to be configured to support the necessary variety of operand sizes. These were then arranged into a PE capable of being configured to implement a single round of most of the considered algorithms. The PE-FUNC proved to be a success in terms of flexibility, but its size was also responsible for the overall system's less-than-desirable performance.

Most of the design difficulties arose from the flexibility that was designed into the system: the large amounts of configuration data, the complexity and amount of data routing, and the extreme redundancy which lead to the large device area and slow speed of operation. Despite these drawbacks, the SHERIF system offers a great deal of flexibility, not only in terms of the variety of algorithms that can be implemented, but also in the fact that it supports different kinds of implementations, such as pipelined and iterative, as well as multiple implementations in parallel. This degree of flexibility is far greater than that offered by any other cryptographic hardware platform, and superior to many software platforms as well.

The large area of the SHERIF device (roughly 138426378 μm^2 or $10,672,813$ gates) and its slow clock speed of 4.78 MHz leads to somewhat disappointing performance compared to custom hardware, but its throughput of 611.84 Mbps for a pipelined implementation of Rijndael is on par with or superior to all but the fastest software implementations on the most modern microprocessors. Thus, even this sub-optimal system implementation offers some performance benefits, and with further research, the SHERIF architecture could achieve higher speeds and smaller area.

Thus, the design of the SHERIF system and the PE-FUNC can be considered a success, offering a significant degree of flexibility in hardware implementation of cryptographic operations. While current performance levels are somewhat low, there are many possibilities for future that may overcome this drawback and add even greater power and flexibility to the SHERIF architecture.

8.2 Future Research Possibilities

The key drawbacks to the current SHERIF architecture are its large size and low estimated speed. While the device would significantly outperform general-purpose microprocessors running at the same clock speed, the low estimated speed means that microprocessors anywhere from 10 to 100 times faster could be used, to potentially give equivalent (if not greater) performance. The large area of the device means that it would be costly to fabricate, and thus a much faster microprocessor may be the more cost-effective option. Consequently, most of the future research possibilities revolve around reducing the device area and improving the speed, but some also offer suggestions for improving the flexibility of the system.

8.2.1 Further Functional Testing

Further functional testing of the current SHERIF architecture would provide greater insight into the operation of the PEs and the system as a whole. Verifying the applicability of the current architecture in implementing the different algorithms discussed in Chapter 3 would identify any dataflow issues in the current architecture and thus could guide future revisions of the SHERIF device. Furthermore, testing the applicability of the SHERIF architecture against algorithms not considered in the initial survey would be valuable in determining how flexible the system can be.

8.2.2 Pipelining

Increasing the degree of pipelining within the PE-FUNC would greatly improve the clock speed and throughput for loop-unrolled, pipelined implementations of algorithms. The routing nodes and key memory were designed to handle future pipelining via their counters, which have a configurable maximum value (up to 127). The drawback is that latency – the number of clock cycles between data being applied and a result becoming available – would increase. Thus, iterative implementations would see little overall improvement, since they require more of the faster clock cycles before completion.

The added benefit to further pipelining is that it would allow smaller parts of the PE-FUNC to be synthesized independently, thus allowing greater optimization and reducing area somewhat. Whether this would offset the additional area required for the pipeline registers is unknown.

Careful placement of pipelining registers after the first Add/Sub SLAB, just before the LUT SLAB, and just before the third Add/Sub SLAB would allow a clock speed of approximately 18 MHz with a latency of 4 clock cycles (as opposed to the current 1 cycle latency). For the pipelined implementation of AES discussed in Chapter 7, this would boost throughput to 2304 Mbps, which would be a significant performance enhancement.

8.2.3 Reducing Device Area

This recommendation is perhaps trivial, but anything that could be done to reduce the device area would make the architecture more cost-effective, synthesizeable, and amenable to higher clock speeds. Migrating synthesis to 0.13 μm CMOS technology would help somewhat, but the key to reducing area will be in optimizing the design of the components.

8.2.4 Elimination of Redundancy

There is a large degree of redundancy in the current SHERIF architecture, as could be seen in the AES implementation in which 26 out of the 29 SLABs were unused in a normal round. In that case, almost 90% of the PE-FUNC resources (comprising roughly 50% of the area) were not being used.

Unfortunately, the flexibility offered by the SHERIF architecture comes from its redundancy, since it offers relatively simplistic routing within the PE-FUNC itself. The overprovision of resources – especially the Boolean SLABs – could be reduced with a more complex and flexible data routing scheme. Also, the large number of input switched and byte reorder components could be reduced by pulling them out of the SLABs and treating them as separate components. Any such changes would require careful analysis to ensure that flexibility is preserved.

8.2.5 Basic Component Optimization

There is a lot of room for optimization of individual components. For example, the LUTs could be implemented with embedded memory, which offers a smaller area than the current flip-flop based implementation but introduces access latency. This latency could be mitigated by or subsumed into any pipelining that is done, thus hiding its effect somewhat. Implementing the LUTs as an SRAM would require a different configuration methodology, since it would no longer function as a configuration register.

The adder component might be improved by using only four banks of 4 adders each that cascade together, rather than two banks of 8. This would reduce the longest-path delay through the device without posing a problem to most algorithms, as well as reducing control logic (since only 3 prior outputs would be multiplexed into the fourth adder) and the number of needed configuration bits. Furthermore, in the basic 8-bit adder component, replacing the multiplexer that switches between b and NOT b with an array of XOR gates each controlled by the inverse of the add_subb control signal would allow a reduction in the total logic, as would replacing the carry-in multiplexer with the inverse of the add_subb control signal. These would provide minor improvements in area and speed, but in aggregate such minor improvements may be significant.

The Boolean logic component also has room for optimization. For example, the

NOT gate could be removed, since the XOR gate can be used with a constant value of all ones to provide an inverting effect. Without the NOT gate taking up one of the 4-to-1 multiplexer inputs, the bypass functionality can be incorporated into that multiplexer, thus saving a configuration bit and eliminating the need for the 2-to-1 bypass multiplexer. While this is a minimal improvement, given the large number of Boolean SLABs in the device, it may prove beneficial.

Components such as the multiplier, XORnet, shifter, and adder components might also benefit from internal pipelining to reduce the depth of the combinational logic. Again, this introduces potentially undesirable latency. Further optimizations are not obvious, but are certainly possible.

8.2.6 Allowing Partial Configurations

In simulation, most of the time is spent configuring the device. In reality, even if configuration takes a long time, once configured, the device can then operate until powered down. This is an acceptable constraint, but could be improved.

In any algorithm implementation, most of the configuration bits will be zero. In fact, there will be very large portions of the configuration that are zero-valued. Configuration could be much faster if parts of the device could be configured independently, rather than via shifting bits in one at a time through the configuration registers. If each configuration register could be written directly, then after system reset, the configuring circuit or processor would only need to write values to the configuration registers that have non-zero values, with the result of needing to write fewer bits than the shifting configuration scheme.

The added benefit of allowing partial configuration beyond shortening the configuration time is that a controlling processor could configure one part of the SHERIF device while another is processing data. Thus, it would allow a degree of dynamic reconfigurability, since the configuration could be changed on-the-fly.

Adding such a capability would be costly in terms of area and complexity, however, and would require a significant degree of modification to the existing SHERIF architecture.

8.2.7 Dynamic Configurations

While performing preliminary investigation of iterative implementations of several algorithms, a design problem became apparent that had initially been overlooked. Almost every algorithm considered has a few operations at the very beginning and/or very end that do not fall within their standard round structure. Thus, a single round implemented in a PE-FUNC would not be able to perform those extra operations, and consequently any iterative implementation would also require the use of PE-FUNCs before and/or after the iterative round PE-FUNC to implement those extra operations. This is wasteful of system resources, since two or three PEs are needed to do what should be implementable by one.

If the PE-FUNC configurations could change while still running, it would be possible to create iterative implementations that incorporated any additional operations into the first round, then while the data was being fed back to the input, it could change configuration to a regular round. This regular round configuration could be held for the middle rounds, and then for the final round a third configuration could be loaded to handle any extra final operations. This capability might be tied to the partial configurability described above, or it could be implemented with an entirely different system. It would increase the effectiveness of iterative implementations, as well as offer an ever greater amount of flexibility in general.

8.2.8 Different Dataflow Approaches

The slowness of the current SHERIF architecture is inherent to its design – despite any pipelining or other optimizations, it will always be relatively slow due to the large amount of logic in its datapath. Thus, future versions of the SHERIF architecture should investigate alternate dataflow approaches.

For example, using non-regular PEs might be beneficial. In this scenario, each PE is smaller and more specialized than the PE-FUNC, supporting a small set of the overall operations. The processing fabric would have a large number of highly interconnected PEs of this type, moving closer to a fine-grained reconfigurable solution. Such an architecture may help reduce the redundancy of the design, and could potentially allow different paths through the system to allow for different degrees of throughput. It would likely have its own currently-unforeseen problems, however.

Another alternate approach that bears consideration is to implement the basic components as functional units in a small microcontroller-style architecture. This would allow a much higher clock speed, and would allow greater flexibility since the algorithms themselves will be written as microcode instructions. While such a microprocessor-oriented architecture would have lower throughput (since pipelining would be impossible), there would be virtually no redundancy, and allowing partial configurations would be easier. Thus, the overall area of such an architecture would be quite small – likely smaller than the estimated area of the PE-FUNC in the current design – and so many could be instantiated in parallel on a chip and increase throughput through parallelism. The added benefit of adapting the SHERIF architecture to a microprocessor-like structure in comparison to other processor-oriented cryptographic hardware is that the SHERIF architecture would maintain the large datapath size (128-bit) and configurability, allowing a greater degree of flexibility and speed.

8.2.9 Subkey Generation

In the future, it would be desirable for the SHERIF architecture to support generation of subkeys directly within the device itself, rather than requiring an external processor to pre-compute them. This could be supported in a System On Chip (SoC) fashion, in which a general-purpose microprocessor core is incorporated into the design, or via a more specialized processing element such as described in the preceding section. The subkey generation unit would have to interface directly with the key memory components in the current architecture, but could be interfaced with alternate architectures in different ways.

8.2.10 Support for Other Cryptographic Operations

Once a version of the SHERIF architecture exists that offers reasonable area and performance, it may be worthwhile to extend support from block ciphers and hash functions to stream ciphers and public key cryptography. This would require development of additional processing elements such as LFSRs to support stream ciphers, and modular multiplication and exponentiation to support public key operations. These components would likely have separate datapaths from the block cipher support, since they require very different models of computation. Integrating such components, however, would make the SHERIF architecture truly flexible.

8.2.11 Improved Configuration Software

While the existing configuration software serves its purpose well, it is very closely tied to the current SHERIF proof-of-concept implementation and thus does now easily allow modifications to the SHERIF architecture without redesigning parts of the software. Thus, a more easily expandable version of the configuration software would be desirable.

The configuration software could also be modified to generate VHDL code for configuration registers with initial values set to the desired configuration values. Such VHDL code would have to be compiled and integrated with the system whenever configuration changes were made. When simulating, they would initialize the SHERIF system to the set configuration, and therefore allow functional testing of the algorithm implementation much more easily, since system configuration would not have to be simulated. This would save hundreds of thousands of clock cycles of simulation time, and might allow simulation of the entire system to be simulated, rather than just individual PEs and routing nodes.

Furthermore, while it handles all the bit-flipping and setting, it still puts all the responsibility for timing on the person implementing the algorithm, which is tricky to handle, especially for iterative implementations. If the configuration software could provide a means of implementation that abstracted away from the specific architectural details and timing, without having to provide high-level language support, it would make the algorithm designer's task much easier.

In fact, abstracting further away from the SHERIF hardware would open up the possibility of using the configuration software with other reconfigurable hardware devices. For example, if the configuration software allowed specification of algorithms in a general sense, it might then generate synthesizable VHDL code that could be run through an FPGA synthesis tool, allowing easy creation of FPGA implementations and providing greater efficiency and flexibility than the existing SHERIF system. Such a software system would clearly move away from being a simple configuration utility and into the realm of being a cryptographic hardware development environment, which could be the topic of much further research.

8.3 Final Remarks

This thesis has investigated the design and implementation of a flexible cryptographic hardware module. While the estimated performance of the prototype SHERIF architecture is poor in terms of area and speed, this research has been valuable in uncovering a number of problems that must be overcome before such a cryptographic hardware module could be feasible for production. Some of these problems are specific to cryptographic hardware; others are general problems of reconfigurable computing. Despite all the difficulties and drawbacks, however, the SHERIF architecture looks promising, and with further research could one day provide flexibility, ease-of-use, and sufficient speed to make it a viable alternative to conventional hardware or software implementation of cryptographic algorithms.

References

[1] S. Keshav, *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network.* Don Mills, Ontario: Addison-Wesley, 1997.

[2] D. R. Stinson, *Cryptography: Theory and Practice.* New York: Chapman & Hall/CRC, second ed., 2002.

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

[4] J. M. Trinidad, "Programmable encryption for wireles and network applications," in *MILCOM 2002 Proceedings*, vol. 2, October 7 – 10 2002.

[5] N. Ferguson and B. Schneier, *Practical Cryptography.* Indianapolis: Wiley Publishing, Inc., 2003.

[6] L. E. B. III, "Efficiency testing of ANSI C implmentations of round 1 candidate algorithms for the advanced encryption standard." Online at `http://csrc.nist.gov/CryptoToolkit/aes/round1/r1-ansic.pdf`, October 13 1999.

[7] B. Weeks, M. Bean, T. Rozylowicz, and C. Ficke, "Hardware performance simulations of round 2 advanced encryption standard algorithms," tech. rep., National Security Agency, May 15 2000. Online at `http://csrc.nist.gov/CryptoToolkit/aes/round2/NSA-AESfinalreport.pdf`.

[8] J. Burke, J. McDonald, and T. Austin, "Architectural support for fast symmetric-key cryptography," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operations Systems*, 2000.

[9] "Advanced encryption standard (aes) development effort." Online at `http://csrc.nist.gov/CryptoToolkit/aes/index2.html`., February 2001.

[10] "New european schemes for signatures, integrity, and encryption (NESSIE)." Online at `http://www.cryptonessie.org`.

[11] H. Lipmaa, "Aes/rijndael: speed." Online at `http://www.tcs.hut.fi/~helger/aes/rijndael.html`, 2004.

[12] A. G. Wassal and M. A. Hasan, "A vlsi architecture for atm algorithm-agile encryption," in *Proceedings, Ninth Great Lakes Symposium on VLSI* (R. J. Lomax and P. Mazumder, eds.), (Ann Arbor, Michigan, USA), IEEE Computer Society, March 4 – 6 1999.

[13] T. D. Tarman, R. L. Hutchinson, L. G. Pierson, P. E. Sholander, and E. L. Witzke, "Algorithm-agile encryption in atm networks," *IEEE Computer*, vol. 31, pp. 57 – 64, September 1998.

[14] M. J. S. Smith, *Application-Specific Integrated Circuits.* Toronto: Addison-Wesley, 1997.

[15] "Performance of camellia." Online at `https://www.cosic.esat.kuleuven.ac.be/nessie/updatedPhase2Specs/camellia/camelliaperformance.pdf`, August 31 2001.

[16] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima,

and T. Tokita, "NESSIE submission: Camellia." Online at http://www.cryptonessie.org/workshop/submissions/camellia.zip, September 2000.

[17] NIST, "Federal information processing standards publication 197 – Advanced Encryption Standard." Online at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, November 2001.

[18] A. Satoh and S. Morioka, "Unified hardware architecture for 128-bit block ciphers aes and camellia," in *Cryptograhic Hardware and Embedded Systems – CHES 2003*, vol. 2779 of *LNCS*, Springer-Verlag Heidelberg, 2003.

[19] L. E. Frenzel, "Cryptochips help eliminate the security bottleneck," *Electronic Design*, March 17 2003.

[20] Motorola, Inc. Online at http://www.motorola.com.

[21] Motorola, Inc., "S1 family." Online at http://e-www.motorola.com/webapp/sps/site/taxonomy.jsp?nodeId=03DnXMl3ZGSbK47721.

[22] Motorola, Inc., "MPC185 product summary page." Online at http://e-www.motorola.com/webapp/sps/site/prod_summary.jsp?code=MPC185&nodeId=01DFTQ42497721.

[23] Motorola, Inc., "MPC190 product summary page." Online at http://e-www.motorola.com/webapp/sps/site/prod_summary.jsp?code=MPC190&nodeId=03DnXMl3ZGSbK47721.

[24] Motorola, Inc., "MPC185 security processor technical summary." Online at http://e-www.motorola.com/files/32bit/doc/data_sheet/MPC185TS.pdf., February 2003.

[25] Motorola, Inc., "MPC190 security processor technical summary." Online at

http://e-www.motorola.com/files/32bit/doc/data_sheet/MPC190TS.pdf., February 2003.

[26] IBM. Online at http://www.ibm.com.

[27] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart, "Building the ibm 4758 secure coprocessor," *IEEE Computer*, vol. 34, pp. 57–66, October 2001.

[28] S. W. Smith and S. Weingart, "Building a high-performance, programmable secure processor," *Computer Networks*, vol. 31, pp. 831 – 860, 1999.

[29] IBM, "Ibm hardware – IBM PCI cryptographic processor." Online at http://www-306.ibm.com/security/cryptocards/html/overhardware.shtml.

[30] IBM, "Product summary – IBM PCI cryptographic coprocessor." Online at http://www-306.ibm.com/security/cryptocards/html/overproduct.shtml.

[31] A. W. H. House and H. M. Heys, "FPGA implementation of ATM encryption algorithms," in *Proceedings of NECEC 2000, the Tenth Newfoundland Electrical and Computer Engineering Conference*, (St. John's, Newfoundland, Canada), Memorial University of Newfoundland, November 2000.

[32] S. W. Smith, E. R. Palmer, and S. Weingart, "Using a high-performance, programmable secure coprocessor," in *Proceedings, Financial Cryptography, Second International Conference FC'98* (R. Hirschfeld, ed.), vol. 1465 of *LNCS*, (Anguilla, British West Indies), Springer, February 23 – 25 1998.

[33] K. S. Spring, "High grade communications security: Reprogrammable technologies for commercial wireless devices," in *Proceedings of SPIE Conference on Enforcement and Security Technologies* (A. T. D. Persia and J. J. Pennella, eds.), vol. 3575, (Boston, Massachusetts), November 1998.

[34] Cavium Networks. Online at `http://www.cavium.com/`.

[35] D. Carlson, D. Brasili, A. Hughes, A. Jain, T. Kiszely, P. Kodandapani, A. V. adn T. Xanthopoulos, and V. Yalala, "A high performance ssl ipsec protocol aware security processor," in *Digest of Technical Papers: 2003 IEEE International Solid-State Circuits Conference*, (San Francisco, California), February 9 – 13 2003.

[36] K. Deats, "Hardware focus – cryptographic accelerators," *HP Professional*, vol. 13, pp. 10 – 11, February 1999.

[37] T. Yamaguchi, T. Hashiyama, and S. Okuma, "A study on reconfigurable computing system for cryptography," in *Proceedings, 2000 IEEE International Conferency on Systems, Man and Cybernetics*, (Nashville, Tennessee, USA), October 8 – 11 2000.

[38] R. R. Taylor and S. C. Goldstein, "A high-performance flexible architecture for cryptography," in *Cryptographic Hardware and Embedded Systems – CHES'99* (Ç. K. Koç and C. Paar, eds.), vol. 1717 of *LNCS*, Springer-Verlag, 1999.

[39] E. Mosanya, C. Teuscher, H. F. Restrepo, P. Galley, and E. Sanchez, "Cryptobooster: A reconfigurable and modular cryptographic coprocessor," in *Cryptographic Hardware and Embedded Systems – CHES'99* (Ç. K. Koç and C. Paar, eds.), vol. 1717 of *LNCS*, Springer-Verlag, 1999.

[40] C. Paar, B. Chetwynd, T. Connor, S. Y. Deng, and S. Marchant, "An algorithm-agile cryptographic co-processor based on fpgas," in *Proceedings of teh SPIE*

Conference on Reconfigurable Technology: FPGAs for Computing and Applications, (Boston, Massachusetts), September 1999.

[41] M. Hileeto and S. J. Simmons, "A low-power reduced-area rom architecture for cryptographic algorithms," in *2000 Canadian Conference on Electrical and Computer Engineering*, vol. 1, IEEE, May 7 – 10 2000.

[42] Z. Shi and R. B. Lee, "Bit permutation instructions for accelerating software cryptography," in *Proceedings, IEEE International Conference on Application-Specific Systems, Architectures, and Processors* (E. E. Swartzlander, G. A. Jullien, and M. J. Schulte, eds.), (Boston, Massachusetts), IEEE Computer Society, July 10 – 12 2000.

[43] NIST, "Federal information processing standards publication 46-3 – Data Encryption Standard." Online at `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`, October 1999.

[44] P. Davies and S. Robsky, "Customized processor extension speeds network cryptology," *Electronic Design*, pp. 83 – 88, September 16 2002.

[45] M. Lewis and S. Simmons, "A VLSI implementation of a cryptographic processor," in *CCECE 2003 - CCGEI 2003*, (Montréal), IEEE, May 2003.

[46] M. D. T. Lewis, "A comparative study of ciphers and their vlsi implementation for low power communications," Master's thesis, Queen's University, Kingston, Ontario, Canada, December 2001.

[47] L. Wu, C. Weaver, and T. Austin, "Cryptomaniac: A fast flexible architecture for secure communication," in *Proceedings, 28th Annual International Symposium on Computer Architecture*, 2001.

[48] S. S. Raghuram and C. Chakrabarti, "A programmable processor for cryptography," in *Proceedings, ISCAS 2000 - IEEE International Symposium on Circuits and Systems*, (Geneva, Switzerland), IEEE, May 28 – 31 2000.

[49] J. Goodman and A. Chandrakasan, "An energy efficient reconfigurable public-key cryptography processor architecture," in *Cryptographic Hardware and Embedded Systems - CHES 2000* (Ç. K. Koç and C. Paar, eds.), vol. 1965 of *LNCS*, Springer-Verlag, 2000.

[50] B. Schneier, *Applied Cryptography.* John Wiley and Sons, Inc., 1996.

[51] "National institute of standards and technology." Online at `http://www.nist.gov.`, 2004.

[52] NIST, "Federal information processing standards publication 180-2 – Secure Hash Standard." Online at `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf`, August 2002.

[53] J. Jonsson and B. Kaliski, "NESSIE submission: RC6." Online at `http://www.cryptonessie.org/workshop/submissions/rc6.zip`, September 2000.

[54] J. L. Massey, G. H. Khachatrian, and M. K. Kuregian, "NESSIE submission: SAFER++." Online at `http://www.cryptonessie.org/workshop/submissions/safer++.zip`, September 2000.

[55] H. Handschuh and D. Naccache, "NESSIE submission: SHACAL." Online at `http://www.cryptonessie.org/workshop/submissions/shacal.zip`, September 2000.

[56] NESSIE Consortium, "Portfolio of recommended cryptographic primitives." Online at `http://www.cryptonessie.org/deliverables/decision-final.pdf`.

[57] M. Riaz, "The hardware implementation of private-key block ciphers," Master's thesis, Memorial University of Newfoundland, St. John's, Newfoundland and Labrador, Canada, 2000.

[58] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs.* Oxford University Press, 2000.

[59] J. E. Earl E. Swartzlander, *Computer Arithmetic*, vol. 1. IEEE Computer Society Press, second ed., 1990.

[60] L. Xiao and H. M. Heys, "Hardware design and analysis of block cipher components," in *Information Security and Cryptology – ICISC 2002*, Lecture Notes in Computer Science 2587, pp. 164–181, Springer-Verlag, 2002.

[61] A. W. H. House and H. M. Heys, "Preliminary design of a flexible cryptographic hardware module," in *Proceedings of the Twelfth Newfoundland Electrical and Computer Engineering Conference*, (St. John's, Newfoundland, Canada), Memorial University of Newfoundland, November 2002.

[62] Sun Microsystems, "J2SE 1.4.2 Documentation." Online at `http://java.sun.com/j2se/1.4.2/reference/docs/index.html`., July 2004.

[63] "NetBeans." Online at `http://www.netbeans.org`., 2004.

[64] J. McCaffrey, "Keep your data secure with the new advanced encryption standard." Online at `http://msdn.microsoft.com/msdnmag/issues/03/11/AES/default.aspx`, November 2003.

Appendix A

VHDL Code for Basic Components

A.1 Boolean Component

```
-- Drive Data Outputs
drive_datapath_out15:
PROCESS (a_in15, b_in15, dcon15)
BEGIN
  CASE dcon15 IS
    WHEN "100" =>
      -- a AND b
      datapath_out(127 DOWNTO 120) <= a_in15 AND b_in15;
    WHEN "101" =>
      -- a OR b
      datapath_out(127 DOWNTO 120) <= a_in15 OR b_in15;
    WHEN "110" =>
      -- a XOR b
      datapath_out(127 DOWNTO 120) <= a_in15 XOR b_in15;
    WHEN "111" =>
      -- NOT a
      datapath_out(127 DOWNTO 120) <= NOT a_in15;
    WHEN "000" =>
      -- pass b (the datapath)
      datapath_out(127 DOWNTO 120) <= b_in15;
    WHEN OTHERS =>
      datapath_out(127 DOWNTO 120) <= b_in15;
  END CASE;
END PROCESS drive_datapath_out15;
```

A.2 Shifter Component

```
-- Left shift operation
dir_left:
PROCESS (data_in, shift_amt, data_mask16, data_mask8,
         data_mask4, data_mask2, data_mask1,
         llayer5, llayer4, llayer3, llayer2)
BEGIN
  IF (shift_amt(4) = '1') THEN
    -- rotate/shift left by 16 positions
    llayer5 <= data_in(15 DOWNTO 0) & (data_mask16 AND data_in(31 DOWNTO 16));
  ELSE
    llayer5 <= data_in;
  END IF;

  IF (shift_amt(3) = '1') THEN
    -- rotate/shift left by 8 positions
    llayer4 <= llayer5(23 DOWNTO 0) & (data_mask8 AND llayer5(31 DOWNTO 24));
  ELSE
    llayer4 <= llayer5;
  END IF;

  IF (shift_amt(2) = '1') THEN
    -- rotate/shift left by 4 positions
    llayer3 <= llayer4(27 DOWNTO 0) & (data_mask4 AND llayer4(31 DOWNTO 28));
  ELSE
    llayer3 <= llayer4;
  END IF;

  IF (shift_amt(1) = '1') THEN
    -- rotate/shift left by 2 positions
    llayer2 <= llayer3(29 DOWNTO 0) & (data_mask2 AND llayer3(31 DOWNTO 30));
  ELSE
    llayer2 <= llayer3;
  END IF;

  IF (shift_amt(0) = '1') THEN
    -- rotate/shift left by 1 position
    from_left <= llayer2(30 DOWNTO 0) & (data_mask1 AND llayer2(31));
  ELSE
    from_left <= llayer2;
  END IF;
END PROCESS dir_left;
```

A.3 Add/Sub Component

A.3.1 Sample Code for 8-bit Adder Block

```
set_cin:
PROCESS (carry_in, connect, add_subb)
BEGIN
  -- Determine carry-in
  IF (connect = '1') THEN
    cin <= carry_in;
  ELSE
    IF (add_subb = '0') THEN
      cin <= '1';
    ELSE
      cin <= '0';
    END IF;
  END IF;
END PROCESS set_cin;

-- invert b input if subtracting
set_b_input:
PROCESS (b, add_subb)
BEGIN
  IF (add_subb = '0') THEN
    b_internal <= NOT b;
  ELSE
    b_internal <= b;
  END IF;
END PROCESS set_b_input;

-- Perform the addition
adder:
PROCESS (a, b_internal, cin)
BEGIN
  -- Perform the addition
  y_prime <= ("0" & a) + b_internal + cin;
END PROCESS adder;

-- Separate adder outputs into data out and carry out
drive_output:
PROCESS (y_prime)
BEGIN
  y <= y_prime(7 DOWNTO 0);
  carry_out <= y_prime(8);
END PROCESS drive_output;
```

A.3.2 Sample Code for part of 64-bit Adder Block

```
adder_inst4: add_sub_8bit
  PORT MAP (--
           -- data inputs a, b, carry-in
           --
           a => a_in4,
           b => b_in4,
           carry_in => cout3,
           --
           -- control inputs connect, add_subb
           --
           connect => conn4,
           add_subb => add_subb4,
           --
           -- data outputs y and carry-out
           --
           y => sum4,
           carry_out => cout4);
```

```
mux4:
PROCESS (s4, a_in4x, sum0, sum1, sum2, sum3)
BEGIN
   CASE s4 IS
     WHEN "000" =>
       -- direct input
       a_in4 <= a_in4x;
     WHEN "001" =>
       -- adder 0 output
       a_in4 <= sum0;
     WHEN "010" =>
       -- adder 1 output
       a_in4 <= sum1;
     WHEN "011" =>
       -- adder 2 output
       a_in4 <= sum2;
     WHEN "100" =>
       -- adder 3 output
       a_in4 <= sum3;
     WHEN OTHERS =>
       a_in4 <= a_in4x;
   END CASE;
END PROCESS mux4;
```

A.4 Configuration Register Components

```
reg_store:
PROCESS (config_clk, rstb)
BEGIN
  IF (rstb /= '1') THEN
    -- Clear the lookup table
    reg <= (OTHERS => '0');
  ELSIF (config_clk 'EVENT AND config_clk = '1') THEN
    IF (config_enable = '1') THEN
      -- shift out least significant config bit
      --shift_data_out <= reg(0);
      -- shift in new config bit as MSB
      reg <= shift_data_in & reg(255 DOWNTO 1);
    ELSE
      -- hold register value
      reg <= reg;
    END IF;
  END IF;
END PROCESS reg_store;

-- Drive data output
data_out <= reg;
shift_data_out <= reg(0);
```

Appendix B

Configuration Formats

B.1 Boolean SLAB Configuration Format

The Boolean SLAB Configuration is 288 bits, partitioned as follows:

- Most significant 128 bits are constant data (bits 287 down to 160).
- Next 48 bits control the input switch (bits 159 down to 112).
  - The most significant 16 bits of the 48 control scratch path multiplexers, selecting between datapath and scratch values.
  - The least significant 32 bits control datapath multiplexers, each 2 bit group selecting between datapath, scratch, constant, and key values.
- Next 64 bits control the byte reordering (bits 111 down to 48).
  - Each 4 bits of the 64 selects which of the input bytes is switched to the output byte corresponding to the position of the bits.
- Least significant 48 bits control core Boolean component (bits 47 down to 0).
  - Each 3 bit group corresponds to an output byte, the MSB controls bypass, while the LSBs selected between AND, OR, XOR, and NOT operations.

B.2 Shifter SLAB Configuration Format

The Shifter SLAB Configuration is 252 bits, partitioned as follows:

- Most significant 128 bits are constant data (bits 251 down to 124).
- Next 48 bits control the input switch (bits 123 down to 76).
  - The most significant 16 bits of the 48 control scratch path multiplexers, selecting between datapath and scratch values.
  - The least significant 32 bits control datapath multiplexers, each 2 bit group selecting between datapath, scratch, constant, and key values.

- Next 64 bits control the byte reordering (bits 75 down to 12).
  - Each 4 bits of the 64 selects which of the input bytes is switched to the output byte corresponding to the position of the bits.
- Least significant 12 bits control core Shifter component (bits 11 down to 0).
  - The most significant 4 bits are bypass control for each 32-bit output.
  - The least significant 8 bits control the shifters, with each 2 bit group controlling rotate/shift and right/left.

B.3 Add/Sub SLAB Configuration Format

The Add/Sub SLAB Configuration is 324 bits, partitioned as follows:

- Most significant 128 bits are constant data (bits 323 down to 196).
- Next 48 bits control the input switch (bits 195 down to 148).
  - The most significant 16 bits of the 48 control scratch path multiplexers, selecting between datapath and scratch values.
  - The least significant 32 bits control datapath multiplexers, each 2 bit group selecting between datapath, scratch, constant, and key values.
- Next 64 bits control the byte reordering (bits 147 down to 84).
  - Each 4 bits of the 64 selects which of the input bytes is switched to the output byte corresponding to the position of the bits.
- Least significant 84 bits control core Add/Sub component (bits 83 down to 0).
  - The most significant 34 bits control the most significant adder bank.
    * Bits 33 down to 31 control input multiplexer to adder 7.
    * Bits 30 down to 28 control input multiplexer to adder 6.
    * Bits 27 down to 25 control input multiplexer to adder 5.
    * Bits 24 down to 22 control input multiplexer to adder 4.
    * Bits 21 down to 20 control input multiplexer to adder 3.
    * Bits 19 down to 18 control input multiplexer to adder 2.
    * Bit 17 controls input multiplexer to adder 1.
    * Bit 16 controls carry in to adder 0.
    * Each 2 bits fro 15 down to 0 control connect and add_subb signals for corresponding adders.
  - The next 34 bits control the least significant adder bank.
    * Bits 33 down to 31 control input multiplexer to adder 7.

        * Bits 30 down to 28 control input multiplexer to adder 6.
        * Bits 27 down to 25 control input multiplexer to adder 5.
        * Bits 24 down to 22 control input multiplexer to adder 4.
        * Bits 21 down to 20 control input multiplexer to adder 3.
        * Bits 19 down to 18 control input multiplexer to adder 2.
        * Bit 17 controls input multiplexer to adder 1.
        * Bit 16 controls carry in to adder 0.
        * Each 2 bits fro 15 down to 0 control connect and add_subb signals for corresponding adders.
    - The least significant 16 bits control bypass for each output byte.

B.4 Multiplier SLAB Configuration Format

The Multiplier SLAB Configuration is 244 bits, partitioned as follows:

- Most significant 128 bits are constant data (bits 243 down to 116).
- Next 48 bits control the input switch (bits 115 down to 68).
    - The most significant 16 bits of the 48 control scratch path multiplexers, selecting between datapath and scratch values.
    - The least significant 32 bits control datapath multiplexers, each 2 bit group selecting between datapath, scratch, constant, and key values.
- Next 64 bits control the byte reordering (bits 67 down to 4).
    - Each 4 bits of the 64 selects which of the input bytes is switched to the output byte corresponding to the position of the bits.
- Least significant 12 bits control core Multiplier component (bits 3 down to 0).
    - The 4 bits are bypass control for each 32-bit output.

B.5 LUT SLAB Configuration Format

The LUT SLAB configuration can be considered to have two parts, the control configuration at the SLAB level, and the actual LUT data itself. They will be considered separately.

The LUT SLAB Configuration is 256 bits, partitioned as follows:

- Most significant 128 bits are constant data (bits 255 down to 128).
- Next 48 bits control the input switch (bits 127 down to 80).
    - The most significant 16 bits of the 48 control scratch path multiplexers, selecting between datapath and scratch values.

  - The least significant 32 bits control datapath multiplexers, each 2 bit group selecting between datapath, scratch, constant, and key values.

- Next 64 bits control the byte reordering (bits 79 down to 16).
  - Each 4 bits of the 64 selects which of the input bytes is switched to the output byte corresponding to the position of the bits.

- Least significant 12 bits control core LUT component (bits 15 down to 0).
  - Each of the 16 bits controls bypass for corresponding output bytes.

The LUT Data Configuration is 32768 bits, partitioned as follows:

- Each 8 bits of data represents the output corresponding to the equivalent address input. Thus, if the address "11111111" was input to the LUT, the most significant 8 bits of the 32768 would be selected for the output; similarly, if the address "00000000" was input, the least significant 8 bits would be selected.

B.6 XORnet SLAB Configuration Format

The XORnet SLAB Configuration is 4341 bits, partitioned as follows:

- Most significant 128 bits are constant data (bits 4340 down to 4213).
- Next 48 bits control the input switch (bits 4212 down to 4165).
  - The most significant 16 bits of the 48 control scratch path multiplexers, selecting between datapath and scratch values.
  - The least significant 32 bits control datapath multiplexers, each 2 bit group selecting between datapath, scratch, constant, and key values.
- Next 64 bits control the byte reordering (bits 4164 down to 4101).
  - Each 4 bits of the 64 selects which of the input bytes is switched to the output byte corresponding to the position of the bits.
- Least significant 4101 bits control core XORnet component (bits 4100 down to 0).
  - The most significant bit selects mode of operation (1 for 64-bit mode, 0 for 32-bit mode).
  - The next 4 bits control bypass for corresponding 32-bit outputs.
  - The next 1024 bits (4095 down to 3072) control output bit generation for the most significant XORnet. Each group of 32 bits selects which input bits contribute to generating the corresponding single output bit.

- The next 1024 bits (3971 down to 2048) control output bit generation for the next XORnet. Each group of 32 bits selects which input bits contribute to generating the corresponding single output bit.
- The next 1024 bits (2047 down to 1024) control output bit generation for the next XORnet. Each group of 32 bits selects which input bits contribute to generating the corresponding single output bit.
- The least significant 1024 bits (1023 down to 0) control output bit generation for the least significant XORnet. Each group of 32 bits selects which input bits contribute to generating the corresponding single output bit.

B.7 Key Memory Configuration Format

The Key Memory Configuration is 47 bits, partitioned as follows:

- Most significant 7 bits are maximum count of the shift counter (bits 46 down to 40).
- Next 7 bits are the maximum count of the reload counter (bits 39 down to 33).
- Next 4 bits are the shift amount (bits 32 down to 29).
- Least significant 29 bits control select between key0 and key1 for each of the 29 SLABs.

B.8 PE-FUNC Configuration Format

The PE-FUNC Configuration is 45274 bits, partitioned as shown in Figure B.1.

The 130-bit top level configuration shown as the most significant bits of the PE-FUNC configuration is partitioned as follows:

- Most significant 128 bits are constant data (bits 129 down to 2).
- Least significant 2 bits select initial scratch input between SLAB 0 key, SLAB 1 key, constant, or datapath values.

B.9 Routing Node Configuration Format

The Routing Node Configuration is 73 bits, partitioned as follows:

- Bit 72 enables streaming operation.
- Bit 71 enables three-state operation.
- Bits 70 downto 64 are the maximum count for system output counter.
- Bits 63 downto 57 are the maximum count for the stream counter.

Top-level config (130)	Keymem config (47)
Boolean config (288)	Boolean config (288)
Boolean config (288)	Boolean config (288)
Shifter config (252)	Boolean config (288)
Add/Sub config (324)	Boolean config (288)
Multiplier config (244)	Boolean config (288)
Shifter config (252)	Boolean config (288)
Add/Sub config (324)	Boolean config (288)
LUT config (256)	LUT data (32768)
Boolean config (288)	Boolean config (288)
Boolean config (288)	Boolean config (288)
Shifter config (252)	Boolean config (288)
Add/Sub config (324)	Boolean config (288)
XORnet config (4341)	Boolean config (288)
Shifter config (252)	Boolean config (288)
Add/Sub config (324)	Boolean config (288)

Figure B.1: PE-FUNC Configuration Format

- Bits 56 downto 50 are the maximum count for data counter 1.
- Bits 49 downto 43 are the maximum count for data counter 2.
- Bits 42 downto 40 select the datapath output for the IDLE state.
- Bits 39 downto 37 select the datapath output for the STREAM condition.
- Bits 36 downto 34 select the datapath output for the RUNNING1 state.
- Bits 33 downto 31 select the datapath output for the RUNNING2 state.
- Bits 30 downto 28 select the feedback output for the IDLE state.
- Bits 27 downto 25 select the feedback output for the STREAM condition.
- Bits 24 downto 22 select the feedback output for the RUNNING1 state.
- Bits 21 downto 19 select the feedback output for the RUNNING2 state.
- Bits 18 downto 16 select the system output value.
- Bits 15 downto 12 select the start signal for the key memory.
- Bits 11 downto 8 select the start signal for the stream controller.
- Bits 7 downto 4 select the start signal for the system output.
- Bits 3 downto 0 select the start signal for the datapath.

B.10 System Configuration Format

The system configuration is 453543 bits and consists of an interleaving of routing nodes and processing elements. For the proof-of-concept system, the overall configuration is as follows:

- Routing Node 0 Configuration
- PE-FUNC 0 Configuration
- Routing Node 1 Configuration
- PE-FUNC 1 Configuration
- Routing Node 2 Configuration
- PE-FUNC 2 Configuration
- Routing Node 3 Configuration
- PE-FUNC 3 Configuration

- Routing Node 4 Configuration
- PE-FUNC 4 Configuration
- Routing Node 5 Configuration
- PE-FUNC 5 Configuration
- Routing Node 6 Configuration
- PE-FUNC 6 Configuration
- Routing Node 7 Configuration
- PE-FUNC 7 Configuration
- Routing Node 8 Configuration
- PE-FUNC 8 Configuration
- Routing Node 9 Configuration
- PE-FUNC 9 Configuration
- Routing Node 10 Configuration