HARDWARE IMPLEMENTATION OF THE SALSA20 AND PHELIX STREAM CIPHERS

JUNJIE YAN

Hardware Implementation of the Salsa20 and Phelix Stream Ciphers

by

@ Junjie Yan

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF ENGINEERING

in

Faculty of Engineering and Applied Science

MEMORIAL UNIVERSITY OF NEWFOUNDLAND

October 2007

St. John's Newfoundland

Acknowledgements

First of all I would like to thank my supervisor Dr. Howard Heys for his constant guidance and support. Without his supervision, this thesis would not exist.

I would also like to thank Dr. Cheng Li. His courses, especially LSI Design (Engr. 8863) and Digital Systems (Engr. 5865) not only made me familiar with the hardware design flow but also the use of VHDL as design language and input for logic synthesis. They are critical tools for my research.

I am very grateful to Canadian Microelectronic Corporation (CMC) as well as Mentor Graphics for their help in providing design tools and the access to state-of-the-art manufacturing technologies.

I thank Namin Yu for the useful literature she provided. With the literature as guidance for chapter organization, grammars and format, I saved a lot of time in terms of thesis writing.

Many thanks to Huiju Cheng for being a great lab partner, travel mate and my best friend at MUN. Our cooperation resulted in more ideas in my research, which was truly rewarding.

I thank Peter Vandrish for his insightful comments and criticism and for careful reading of my papers before publishing. I very much appreciated the pleasant and fruitful time studying together with him.

Furthermore, I would like to express my gratitude to Avalon Microelectronics Incorporation, who has provided me with a fabulous job as an FPGA designer, making it possible for me to continue my work and study on hardware implementation in a broader range of applications.

Junjie Yan

July 15, 2007

Abstract

This thesis investigates the hardware implementation and statistical analysis of new stream ciphers, Phelix and Salsa20. Both are candidates for the eSTREAM project, a project highlighting the state of stream cipher design and analysis.

From a physical technology perspective, hardware implementation methodology consists of Application Specific Integrated Circuit (ASIC) design and Field Programmable Gate Array (FPGA) design. When high performance is required, an ASIC is typically chosen as the implementation platform. However, FPGA platforms have become increasingly popular due to their flexibility and a diminishing performance tradeoff as compared with ASIC technology. Following this trend we have developed two versions of Salsa20, one for deployment on an ASIC, the other for an FPGA. The cipher Phelix is studied for application to ASIC environment.

Implementing a cipher requires detailed knowledge of the cryptographic algorithm itself, particularly the underlying arithmetic. In the case of Phelix and Salsa20, both of which are composed of several simple operations: 32-bit addition, bitwise addition (exclusive or) and rotation, the most important operation is the 32-bit addition, for which we have investigated multiple structures for the adders and compared them in both speed and area. Different adder architectures are chosen for different designs, and the basic criteria is the concern of speed or area the overall implementation consumes.

Two structures for Phelix have been implemented, one is a high speed design and the other one is aimed at compactness. The simulation results shows that it consumes about 12,000 two-input NAND gates in the compact design and achieves more than one Gbps throughput in the high speed design. The speed of the compact design is 260 Mbps and the area of the high speed design is 64,200 two-input NAND gates. Up to four different structures are investigated for Salsa20 as extra considerations are given to the utilization of FPGA. The proposed VLSI implementations achieve a data throughput up to 4.8 Gbps, and a compact FPGA design uses 194 slices and 4 memory blocks in a Xilinx device. The proposed designs in the thesis serve mainly as a quick evaluation of their hardware performance; hence, further architectural optimizations are certainly possible.

Security analysis is an important concern in cipher designs. Thus, we have applied certain statistical tests, which are publicly available in the NIST (National Institute of Standards and Technology) test suite to test various sequences produced by using the Phelix and Salsa20 algorithms. Since the test suite has not considered the relationship between key, IV, internal state and the keystream, we also applied six novel tests to examine the ciphers. Two strategies are employed to interpret the test results: the examination of the proportion of sequences that pass a statistical test and the distribution of P-values to check for uniformity. NIST gives the definition of P-value: the probability that a perfect random number generator would have produced a sequence less random than the sequence that was tested. The experimental results show that both Salsa20 and Phelix have passed the tests in NIST, considering that P-value less than 0.01 indicate a possible weakness. An easily understood deviation is observed in the correlation test for the last internal state (the state after 9 double rounds) and the keystream in Salsa20. However, how this could be exploited in an attack is an open question.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
CBC	Cipher-block Chaining
CFB	Cipher Feedback
ECB	Electronic Codebook
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GB	Gigabyte
Gbps	Gigabits per second
GF	Galois Field
IP	Intellectual Property
IV	Initialization Vector
LFSR	Linear Feedback Shift Register
MAC	Message Authentication Code
Mbps	Megabits per second
MUX	Multiplexer
NIST	Institute of Standards and Technology
OFB	Output Feedback
RAM	Random Access Memory
S-box	Substitution Box (or vectorial Boolean function)
VPN	Virtual Private Network
XOR	Exclusive OR

Chapter 1 An Introduction to Cryptography

1.1 Brief History of cryptography

Cryptography is a fascinating topic related to confidentiality, authenticity and integrity of information and its origin. The earliest cryptography can be traced back to the Egyptians 4000 years ago. At that time, cryptography was concerned solely with message confidentiality, such as encryption, which concerns the process of converting original information (plaintext) into indistinguishable gibberish (ciphertext). The most striking development of cryptography came with the proliferation of the computers and communication systems since 1960s. During this time, when humans communicate with each other by using digital signals instead of traditional written language symbols, large volumes of information exchanging over untrusted medium, such as the Internet, make the classical cryptographic methods out of date; security becomes a tremendously important issue to deal with. The search for new encryption schemes and improvements to existing information security mechanisms and cryptanalysis continues at a rapid pace.

In 1978, the first practical public-key encryption algorithm was discovered, known as RSA, which is based on the intractability of factoring large integers. During 1970s, IBM designed the Data Encryption Standard (DES), the most common symmetric key cryptography scheme used today. DES has been used extensively in electronic commerce. Since 1975, some people have noted that the key size of DES is too small. Moreover, some others worried about NSA's involvement. In 1987, the well known stream cipher RC4 was created by Ronald Rivest. To generate a pseudorandom keystream, the cipher makes use of a secret internal state consist of 2064 bits and two 8-bit index pointers. The plaintext is combined with the keystream using the exclusive-or (XOR) function. In 1997, NIST launched a project to find primitives that were suitable to replace DES. Among the candidates, the Rijndael algorithm was selected at last in 2001 and has become the Advanced Encryption Standard (AES). Recently, quantum computing techniques in cryptography have attracted a lot of attention as a new area of research. It is believed that quantum mechanical principles used in computation might significantly outperform the current prevailing cryptography methods.

Nowadays, the rapid growth of electronic applications and business based on the Internet has fueled the need for cryptographic methods to protect information processing.

1.2 Information Security

A very important issue related to cryptography is information security. Both of them share the common goals of protecting information. However, the scope of information security is larger than cryptography. Over the centuries, information security does not only use cryptography to mask usable information, but also includes the process of protecting data from unauthorized access, use, disclosure, destruction, modification, or disruption [1]. Information security requires a vast range of methods and the technical means is typically provided by cryptography.

Some objectives are frequently used interchangeably in information security, computer security and information assurance. For example, *Data Integrity* is one of the objectives that are used in all of the three fields. It ensures that the information has not been altered by unauthorized or unknown means. Some other objectives include:

1. ***Message Authentication***: corroborates of the identity of an entity.
2. ***Signature***: binds information to an entity.
3. ***Validation***: provides timeliness of authorization to use information.
4. ***Witnessing***: verifies the existence of information by an entity.

More objectives associated with information security are listed in [6].

Information security was mostly applied in military in the past, such as the old Caesar Cipher [70] and the Playfair Cipher [70], which was widely used by the British and U.S armies in World War I. World War II probably brought about most advancement in information security. Also, it witnessed the formalized classification of data based on the information sensitivity [61].

The core principles of information security include confidentiality, integrity and availability, which are known as the CIA Triad. Confidentiality is a service used to prevent information to be accessed, used or disclosed by unauthorized parties. Approaches providing confidentiality range from physical protection to cryptographic algorithms. Integrity is used to avoid unauthorized alteration of data. Data alteration includes substitution, insertion and deletion. Availability means that all the resources used to process and protect the information are available and work correctly when needed.

1.3 Background on Functions

While this thesis will look into the details of implementation of different functions in selected ciphers, a familiarity with basic mathematical functions that are widely used in cryptography will be helpful.

1.3.1 One-way Function

A function *f* is called a one-way function if *f(x)* is easy to compute but hard to invert. "Easy to compute" means computationally feasible or equivalently, one can compute the function in polynomial time. "Hard" in the context refers to average case complexity [6].

The existence of a one-way function is partly dependent on an open conjecture $P \neq NP$, where P is the set of decision problems that are known to be solvable in polynomial time, and NP is the set of decision problems that can be solved by a non-deterministic Turing machine in polynomial time [51]. In other words, if $P = NP$, any function that can be computed in polynomial time can be inverted in polynomial time, which means that one-way functions do not exist. However, it is not know whether $P \neq NP$ is the sufficient condition of the existence of one-way function.

Many useful cryptographic primitives like pseudorandom number generators are based on the existence of a one-way function. Obviously, it is not known whether a one-way function candidate is indeed one-way at present. Some famous candidates supported by current research results include integer factorization and discrete logarithm. The widely used RSA cryptosystem is an example that utilizes integer factorization, while the Digital Signature Algorithm (DSA) [82] uses the discrete logarithm problem.

1.3.2 Permutation

In a permutation, a set of objects or symbols are rearranged into distinguishable sequences. Permutations are invertible functions that are often used as a basic component in various cryptographic constructs, particularly in symmetric key cryptographic algorithms to encrypt large volumes of data. That is because the permutation is easy to be implemented and works fast in hardware implementations since bit or byte level permutations do not require any extra resources but simply reorder logic signals. DES and triple DES use bit level permutations to spread the redundancy of the plaintext over the ciphertext.

However, in software implementations, bit level permutations bring two challenges [63]. In existing RISC processors, $O(n)$ [*] instructions are generally required in generic ways to achieve any one of $n!$ permutations, that is the reason for old processors not supporting arbitrary bit level permutations except a restricted subset known as rotation. The other challenge is that the instructions may need more than two word-sized operands and/or produce more than one word-size result. For example, multiplying two 1024-bit operands in RSA requires two 16-word operands, if the computer is 64-bit word based.

1.3.3 Substitution

In cryptography, substitution is a method that substitutes the units of plaintext with ciphertext according to an alphabet; the "units" might be one symbol or a mixture of symbols. The receiver decrypts the ciphertext by performing an inverse substitution.

The substitution function can be compared with transposition or permutation functions. The difference is that, in a transposition procedure, what changes is the order of the plaintext units, while in a substitution procedure, the units of the plaintext are retained in the same position in the ciphertext, but the units themselves are changed.

Old substitution ciphers are often vulnerable to frequency analysis [10]. Hence, most of them, such as a simple affine cipher, are no longer in serious use. However, the concept of substitution is still being used today. From an unusual perspective, some modern bit-oriented ciphers (e.g. DES) can be considered as substitution ciphers with an enormously large binary alphabet. Additionally, some smaller substitution tools known as an S-boxes are often used for confusion, which obscures the relationship between the plaintext and the ciphertext.

If a substitution scheme and a permutation scheme are used serially in a cipher, then it is called SP – network, or substitution-permutation network (SPN). AES is a case in

[*] O(n) is called big O notation in computational complexity theory. It is often used to describe the relationship between the size of the input data and the running time or memory consumption of an algorithm [24].

point [56]: in the substitution step, each byte in the process is replaced with its entry in a fixed lookup table, known as an S-box; in the permutation step, bytes in each row are shifted cyclically to the left.

1.3.4 Exclusive OR

Exclusive OR, also known as XOR, is a bitwise operator from binary mathematics. In modern ciphers, XOR is often used to mix key bits into the cipher data. It can be denoted as:

$$C = P \oplus K$$

where $\oplus$ denotes XOR operation, C is ciphertext, P is plaintext and K is key. According to the principles of XOR, the decryption process is merely reapplying the key as below:

$$P = C \oplus K$$

If the key or keystream is as long as the message, the system is similar to the one-time pad that is theoretically unbreakable. Besides, XOR is simple to implement and computationally inexpensive. It is provable that the uncertainty in attempting to guess the keystream is equal to that of directly guessing the plaintext. The security of the one-time pad is based on this. Clearly, when the length of plaintext is very long, it is impractical to maintain and distribute the keystreams. As a result, stream ciphers are introduced and developed by the loose inspiration from the one-time pad. More details will be discussed later.

1.3.5 Modular Addition

Many cryptographic primitives include modular *addition* because addition mod 2^n is a nonlinear transformation over GF(2) and the operation is fast in both software and hardware. GF(2) is the Galois Field of elements 0 and 1. Nonlinear transformation is of great importance in cryptography as it makes functions hard to invert.

Keeping with the popularity of addition in ciphers, Klimov and Shamir proposed T-function in 2002 [5]. T-function employs *addition* mixed with *multiplication* and *or* in a certain way to update every bit the internal state. The authors of [57] investigated the probability distribution of the carry chain for integer addition. More literature that looked into modular addition for various aspects can be found in [33], [32], [13] and [37].

1.3.6 Modular Multiplication

Modular multiplication is widely used in cryptography as it has good diffusion properties [76]. It is a multiplication performed over a finite field. The most straightforward method for performing modular multiplication is to compute the remainder on division by the modulus. This is referred to as the classical modular multiplication algorithm.

Since most applications are based on the binary representation system, the modulus is often a power of two (as in RC6). Therefore, some efficient algorithms without explicitly carrying out the classical modular reduction step are widely exploited. For example, in [76] a technique is proposed, which efficiently implements 2^n+1 prime modulus operation by using only two additional additions and one multiplication.

1.4 Symmetric-key/Private-Key Cryptography

There are many ways to divide cryptographic primitives. Figure 1.1 provides a schematic listing of the primitives considered.

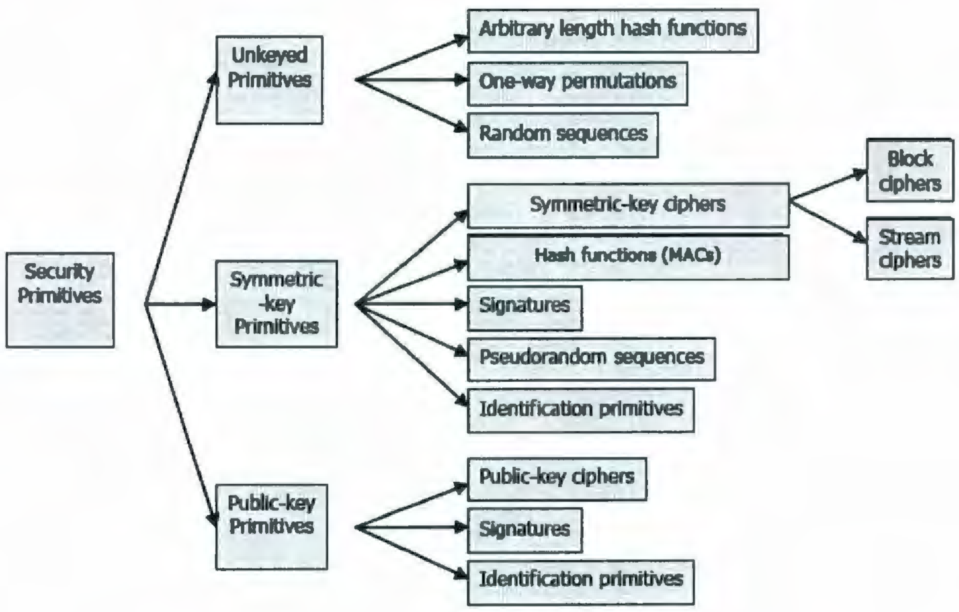


Figure 1.1 A Taxonomy of Cryptographic Primitives [6]

Among the list shown above, our main concern is symmetric-key cryptography and public-key cryptography. Symmetric-key/Private-Key cryptography was the only kind of encryption publicly known until 1976 [73]. In this method, both sides of information communication share the same secret key, or different keys that are related in an easily computable way. The scenario can be described by the block diagram of Figure 1.2.

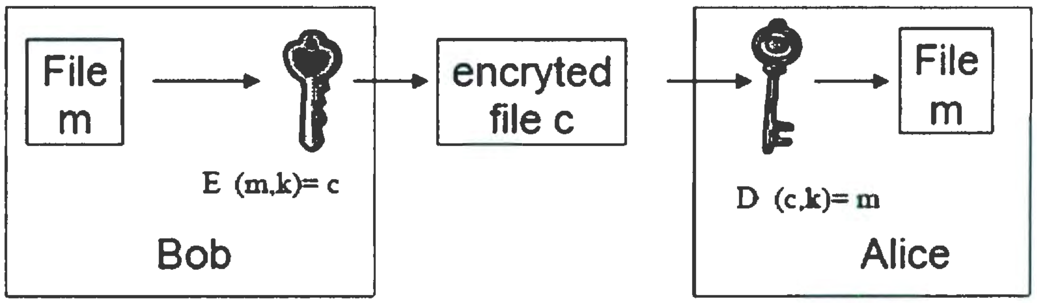


Figure 1.2 Two-party Communications Using Symmetric Key Cryptography

It is assumed that both the sender Alice and the receiver Bob know the encryption/decryption scheme. The ciphertext is transmitted through an insecure channel, which is possibly eavesdropped by a third party Eve.

An encryption algorithm E is employed to encrypt the plaintext m with the secret key k; c is the resulting output as the ciphertext. After Alice receives the ciphertext, the corresponding decryption algorithm D is used with the same secret key k to reveal the original plaintext m.

Symmetric-key encryption can be divided into stream ciphers and block ciphers.

1.4.1 Block Ciphers

A block cipher is a symmetric-key cipher that operates on data in blocks. The input plaintext and the output ciphertext have fixed lengths, often 64, 128 or 256 bits. Another input is the secret key as shown below:

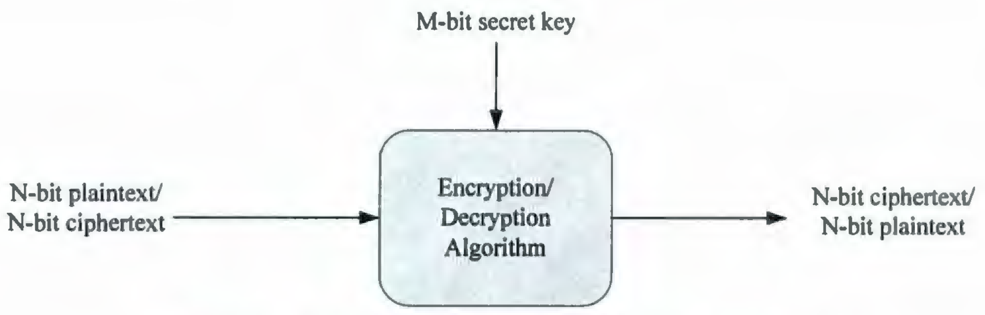


Figure 1.3 Encryption/Decryption Process of Block Ciphers

The size of key is closely related to the security of the cipher. Typically, the key size relates to the effort and time needed to decrypt it by brute force. It is widely accepted that a key should be large enough to prevent a brute force attack. Different cryptographic systems may have different key sizes but with the same level of security in relation to other non-brute force attacks.

For a specific block cipher, the length of a data block is fixed. When the message is longer than one block size, a mode of operation is required. Some modes of operation allow block ciphers to operate on a message of arbitrary length. The earliest modes described in the literature include Electronic Codebook mode (ECB), Cipher Block Chaining mode (CBC), Cipher Feedback mode (CFB), and Output Feedback mode (OFB). Another aspect to consider for messages coming in a variety of lengths is padding, which pads the final block before encryption. The simplest padding is to add null bytes or bits to the plaintext to make it a multiple of the block size. More padding methods can be found in [81].

Block ciphers are widely used in many applications and cryptosystems. An old and prevailing block cipher was the Data Encryption Standard (DES) with a block size of 64 bits and a key size of 56 bits. In October 2000, the National Institute of Standard and Technology (NIST) selected Rijndael algorithm for a standard known as the Advanced Encryption Standard (AES) [72] with the objective to replace DES. The result is a block cipher that is capable of supporting a block size of 128 bits and key sizes of 128, 192, and 256 bits.

1.4.2 Stream Cipher

Stream ciphers treat the plaintext bit by bit or byte by byte continuously and generate one bit/byte of ciphertext at a time. Much of the popularity of stream ciphers is due to the theory of one-time pad; originally known as the Vernam cipher [28].

Typically a stream cipher has a key setup phase and a pseudorandom bit generation phase (keystream generation). In the key setup phase, a secret key and a known initial vector IV are fed into the keystream generator to generate the initial internal state of the cipher and expand the original key when needed. In the pseudorandom bit generation phase, the keystream generator creates pseudo-random sequences to XOR with the plaintext bits. The encryption and decryption processes are shown in Figure 1.4.

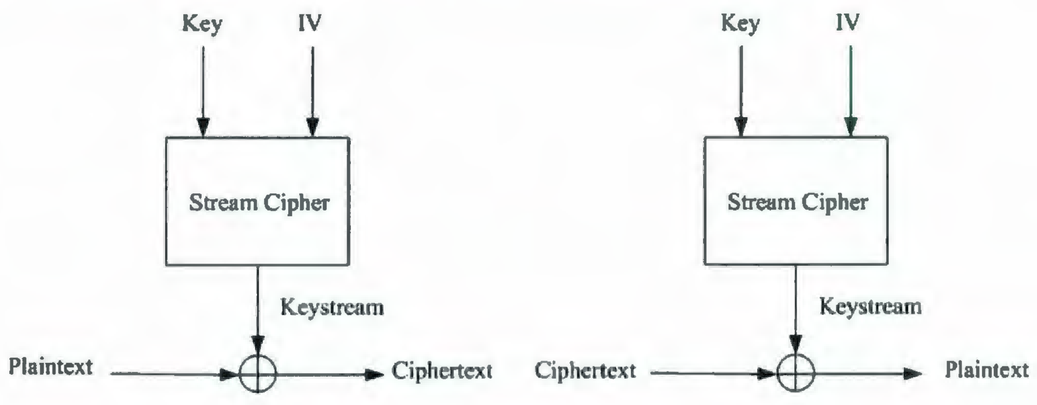


Figure 1.4 Encryption/Decryption Process of Stream Ciphers

As shown above, the sender and the receiver must be exactly synchronous. If digits are altered or removed from the message during transmission, it might cause decryption failure. However, several schemes can be used to rebuild synchronization. For example, regular points in the output can be selected and added with tags, which functions like a marker to inform the receiving side.

Many stream cipher designs are based on linear feedback shift registers (LFSRs) [80] as shown in Figure 1.5. LFSRs by themselves are trivially breakable but non-linear functions such as the use of clock-controlled generators [80] can be added to increase security. LFRS based stream ciphers are very popular since it is easy for them to be implemented in hardware, and their properties are well-understood.

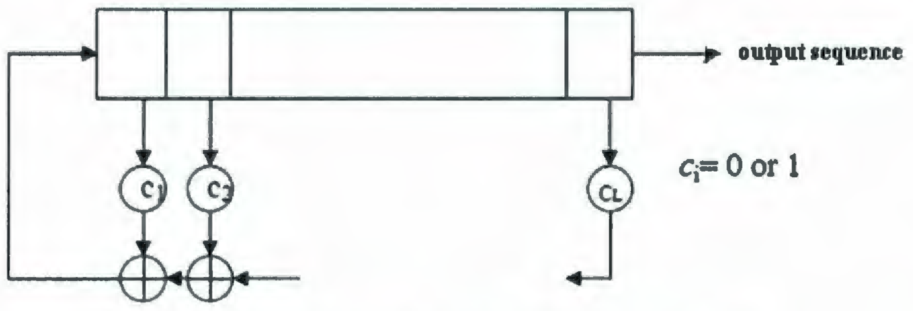


Figure 1.5 A Linear feedback shift registers (LFSR)

Sometimes, a stream cipher can be derived from a block cipher. For example, a block cipher in the output feedback (OFB) mode generates keystream blocks, which are then XORed with the plaintext to produce the ciphertext. Counter mode is another method to turn a block cipher into a stream cipher. It has similar characteristics to OFB; the difference is that it allows a random access within the keystream during decryption because each encryption block operates on independent input, contrary to OFB, in which every output feedback block cipher operation depends on all previous ones. An example for counter mode is Salsa20 [20], a new stream cipher that will be introduced in Chapter 4. It generates the next block of keystream by making use of successive values of a counter.

Even when derived from a block cipher, most stream ciphers in practical use are still generated independently from the plaintext. It is error propagation free, but also has a drawback of requirement for synchronization. However, some stream ciphers violate the model: the keystream depends on the plaintext. An example is Phelix [22]. The authors claim that the basic reason for this violation is to incorporate message authentication "for free".

The security of a stream cipher significantly depends on the period of the keystream, that is, the size of the keystream before it starts to repeat itself. It is a very practical concern and tradeoff between the security and the size of the keystream should be made based on the requirement of the targeted application.

Compared with block ciphers, stream ciphers are more suitable for applications where plaintext comes in bursts, since the former works on blocks of fixed length, leading to a choice between transmission efficiency or implementation complexity.

Although it is impossible to conclude which one is more superior, block ciphers have received more attention when comparing the proceedings of the major cryptography conferences. This imbalance may due to the preoccupation of the block cipher Data Encryption Standard (DES) [70], which had mastered the area of symmetric key cryptography for many years.

In October 2004, the state of stream cipher design and analysis were highlighted by the ECRYPT Network of Excellence in Cryptology, which initiated a workshop to develop a project called the ECRYPT Stream Cipher Project [34], with the goal to identify new stream ciphers that might become suitable for widespread adoption.

1.5 Asymmetric-key/Public-key cryptography

Public-key cryptography is also known as asymmetric cryptography. It uses a pair of keys (a secret key and a public key) instead of one secret key during the encryption and decryption process. The public key, as the name indicates, may be widely distributed, while the secret key is kept private. Only with the corresponding secret key, the plaintext encrypted with the public key can be decrypted.

Public key cryptography is often divided into two categories based on their applications: public key ciphers and digital signatures, which are shown in Figure 1.6 and Figure 1.7, respectively. Only one transmission direction is shown in the figures since the other one is exactly the same except that the names "Bob" and "Alice" exchange.

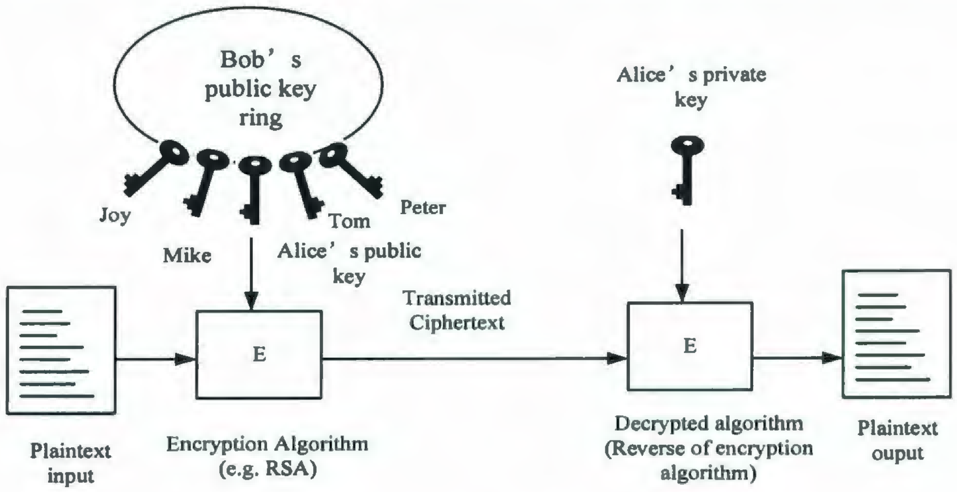


Figure 1.6 Encryption/Decryption Process for Public Key Ciphers

Public key ciphers are used to ensure confidentiality, since only the receiver's secret key can decrypt the message. To the contrary, in a digital signature, any one that holds the sender's public key can verify a message signed with the sender's secret key. This method is used for message authenticity.

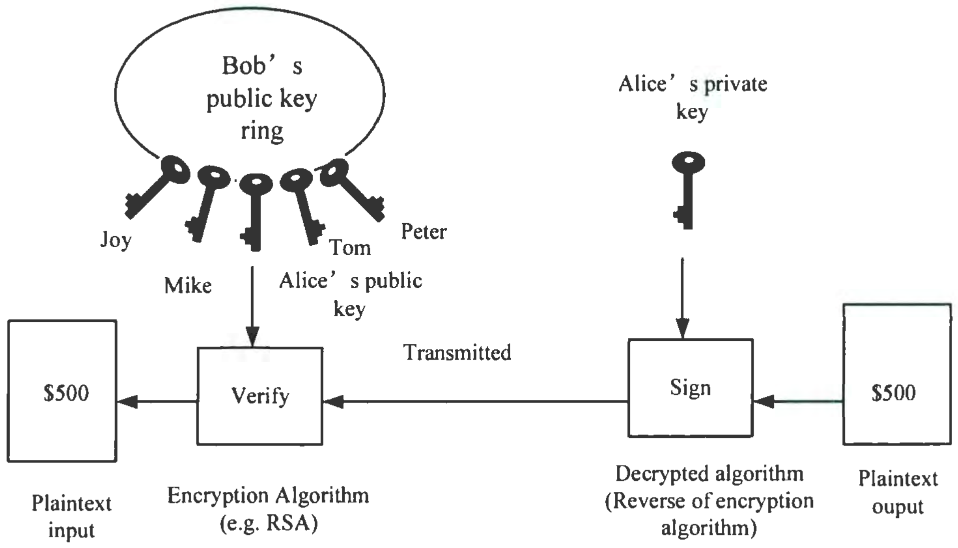


Figure 1.7 Sign/Verify Process for Digital Signature

Normally, a symmetric-key algorithm runs much faster than a public-key algorithm because of fewer computations, but public-key algorithms can facilitate key distribution. For example, if there is a communication group of n people, $n(n-1)/2$ secret keys are required to ensure security in a symmetric-key system, and they should be changed regularly during distribution. It indicates that symmetric keys need to be distributed in an authentic and confidential manner. But in a public-key system, only authenticity is considered, thus, it simplifies key management. To take advantage of both, symmetric-key algorithms and public-key algorithms are not typically used alone. In modern cryptographic algorithm implementations, they are often used as a combination.

1.6 Cryptanalysis

Although breaking codes and ciphers has a very long history, the systematic study of cryptanalysis is relatively recent. In 1920, William Friedman firstly proposed the word "*Cryptanalysis*" for the methods and study to obtain the encrypted message, without knowing the secret information, such as a secret key. The development of computers allows more complex encryption schemes related to binary format information. However, it has also facilitated cryptanalysis.

During many years, the main concern in cryptography is to propose and implement good ciphers that stay ahead of cryptanalysis. Normally, a "good" cipher does not indicate absolute security, but is based on the standard that breaking it requires an effort that makes cryptanalysis too inefficient and impractical.

Nowadays, many types of attacks on cryptographic systems have been invented. Most of them can be categorized into two branches:

- A passive attack: threatens message confidentiality since the adversary can monitor the communication channel.
- An active attack: not only threatens message confidentiality but also data integrity since the adversary might add, delete, or alter the original message that is transmitted through the communication channel.

The objective of cryptanalysis can be recovering plaintext from ciphertext, or even deducing the decryption key. Based on the assumptions about how much information could be obtained by the adversary, cryptanalysis can be performed under the attacks below:

- Ciphertext-only attack: the cryptanalyst only observes a collection of ciphertexts, and tries to deduce the decryption key.
- Known-plaintext attack: the cryptanalyst has a group of ciphertexts and their corresponding plaintext.
- Chosen-plaintext (chosen-ciphertext) attack: the cryptanalyst can access the ciphertext (plaintext) for an arbitrary chosen plaintext (ciphertext).
- Adaptive chosen-ciphertext attack: improved version of chosen-ciphertext attack since the cryptanalyst chooses ciphertext based on information learned from previous requests.

Practically, determining whether an attack is successful depends on the amount of resources it requires, or more specific, time complexity and space complexity. The former could be from a measurement on the number of basic computer instructions; the latter often indicates the amount of storage required to perform the attack. One of the most important assumptions in modern cryptography is Kerckhoffs' Principle [11]: In assessing the security of a cryptosystem, on should always assume the enemy knows the details of the cipher being used. As a result, the security of a cryptosystem should be based on the key instead of the encryption/decryption algorithm it uses.

1.7 Summary

This chapter introduces various aspects of cryptography. Emphasis has been placed on the basic issues of block cipher and stream cipher. The discourse on the rudiments of cryptography leads us gradually to move deeper into the implementation and performance evaluation of ciphers. One of the major inspirations for working on hardware implementation of specific ciphers is the rapidly increasing demand for different hardware designs for various applications, some of which, such as cell phones, take compactness as most important factor in real use while others, such as virtual private network (VPN) applications, prefer high speed.

The following chapter is about software implementation and hardware implementation design and methodologies. Selected topics about software/hardware co-designs and considerations for trade-offs can be found too.

Chapter 2 Cipher Implementation

A cryptosystem can be developed by implementing one or several cryptographic algorithms either on general-purpose microprocessor [45] or on ASICs/FPGAs. Typically, the former is called software implementation, and the latter is known as hardware implementation.

Also, to match the challenges of modern applications that have different requirements about speed and area efficiency, hardware/software codesign for cryptographic systems as a standard design technique has attracted much attention in recent years.

2.1 Software Implementation

Software implementation is a very flexible method to realize encryption/decryption algorithms. Most software implementations are based on a general-purpose processor and its corresponding instruction set. The algorithm is translated into a group of instructions, which will be accessed one by one, decoded into machine language and executed to fulfill encryption/decryption tasks.

One important factor that decides the efficiency of software implementation is the basic underlying architectures, such as the word size of the processor. This is evident if one looks into the performance of AES on different platforms [29]. In the eSTREAM project, most stream ciphers are simple designs composed of a series of simple operations. Thus, when their corresponding codes run on a general processor, the hits or misses on L1 cache that is a typical component of today's RISC processors can have a significant impact on the performance.

Another factor is the software language used. In most of the cases, ciphers implemented by assembly language generally produce better performance compared with interpreted language [39]. That is why most benchmarks to measure the performance of algorithms tend to choose low-level language for implementation. However, it does not mean giving up high-level language implementations. Some languages like Java have priority in flexibility with an interpreter, and they are very suitable for a wide range of cross-platform software implementations. In our hardware implementation, we also use a

Java software implementation to verify the generated keystreams.

Sometimes, bad coding style may have a significant impact on the performance. For example, executing the code "c = 8 × a" can be much slower than executing an equivalent code "c = a << 3", where "<<" means shift left.

Nowadays, a majority of software implementations for symmetric-key cryptographic algorithms have a speed level of several hundred Mbps for the throughput.

2.2 Hardware Implementation

For the last decade, high-speed applications such as virtual private network (VPN) applications and secure e-commerce web servers have gained increasing acceptance in the industry. To sustain the high throughput, the demand for high-speed encryption is also rapidly increasing. A hardware implementation typically runs faster than its software counterpart. The main reason is that an application specific chip contains none of the baggage necessary to execute non-cryptographic workloads. Hence, many block ciphers implemented in hardware achieve throughput in Gbps by taking advantage of parallelism. For example, a full pipelined AES-128 hardware implementation that runs at 200 MHz has a throughput of 128 bits × 200 MHz = 25.6 GB/s.

However, some ciphers are sequential in nature, such as LFSRs based stream ciphers. Hence, they can not be pipelined. Thus, the throughput is directly decided by the clock frequency. In general, the throughput can be defined as below [44]:

$$\text{Throughput} = N \times \text{clock frequency}$$

where N is the width of the output processed per clock cycle.

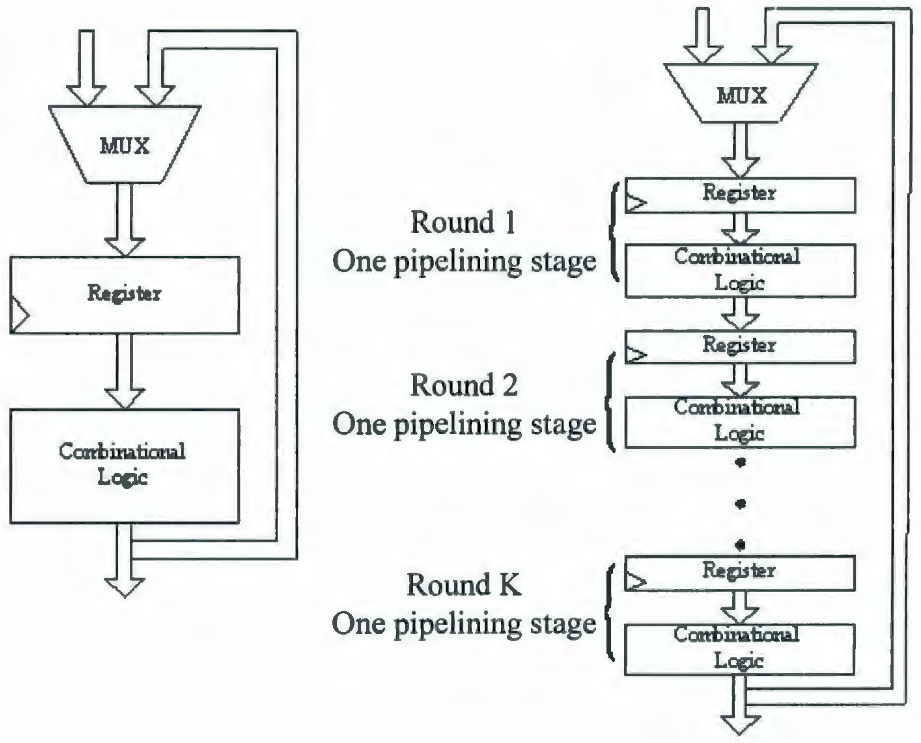
The clock frequency typically depends on the critical path, that is, the longest path between registers. When the critical path is relatively long, higher N can compensate to improve throughput.

Except the advantage in speed, hardware implementation provides a suitable level of security related to side-channel attacks [15], which are based on considerable technical knowledge of the physical system, rather than the weakness of the algorithm itself. For instance, the timing information or the power leaks could provide the information for cryptanalysis. Thus, the well-studied underlying structure of a general processor saves much effort for cryptanalyst. Alternatively, a hardware implementation can be designed to be explicitly resistant to side channel attacks.

2.2.1 Methodology

2.2.1.1 Traditional Methodology Used for Non-feedback Cipher Modes

The traditional methodology for efficient implementations of secret key block ciphers is shown in Figure 2.1.



a) one round, no pipelining b) K round pipelining

Figure 2.1 Architectures Used for Non-feedback Cipher Modes

Figure 2.1.(a) is known as a basic iterative structure. Based on the basic iterative architecture, a reasonable estimation for speed and area for a single round can be estimated. From these estimations, the pipeline stages can be decided without exceeding the available hardware resources. Usually, a single stage is the implementation for a basic round. Figure 2.1.(b) is a partial outer-round pipelined structure applied for limited resources. Extra registers are inserted between any two stages. In this way, with a K-stage pipeline, K blocks of data can be processed by the circuit at the same time. At the end of a clock cycle, the output of each block will be stored in those registers. The throughput as well as the area increase proportionally to the number of stages, while the latency for encryption and decryption remains the same as in a basic iterative structure.

Although pipelining increases the speed of the implementation significantly, equalizing data and control path latency are very important issues to guarantee the overall efficiency. Moreover, it cannot be used with the standard feedback modes such as OFB and CFB since the output of one iteration of the encryption/decryption process must be available before the next iteration can start. In our design, both iterative structure and pipelined structure are investigated for comparison on speed and area.

2.2.1.2 ASIC Design Flow & FPGA Design Flow

From a physical technology perspective, hardware implementation methodology consists of ASIC design and FPGA design. The cost of designing application specific integrated circuits (ASICs) is increasing every year. Issues such as non-recurring engineering (NRE) cost, clock tree synthesis, and time-to-market delays can have significant impact on ASIC design.

In hardware implementations, major parts such as CPU and memories need to be connected by extra custom electronic circuitry. Glue logic is designed to do this job. Since the first programmable device with glue logic, both the speed and the density have increased dramatically. Hence, today's field programmable gate arrays (FPGAs) play a central role in digital hardware implementations. With the support for numerous EDA tools, designers tend to choose FPGA when the volume of the product is less than millions of chips. Typical ASIC and FPGA design flows are shown in Figure 2.2.

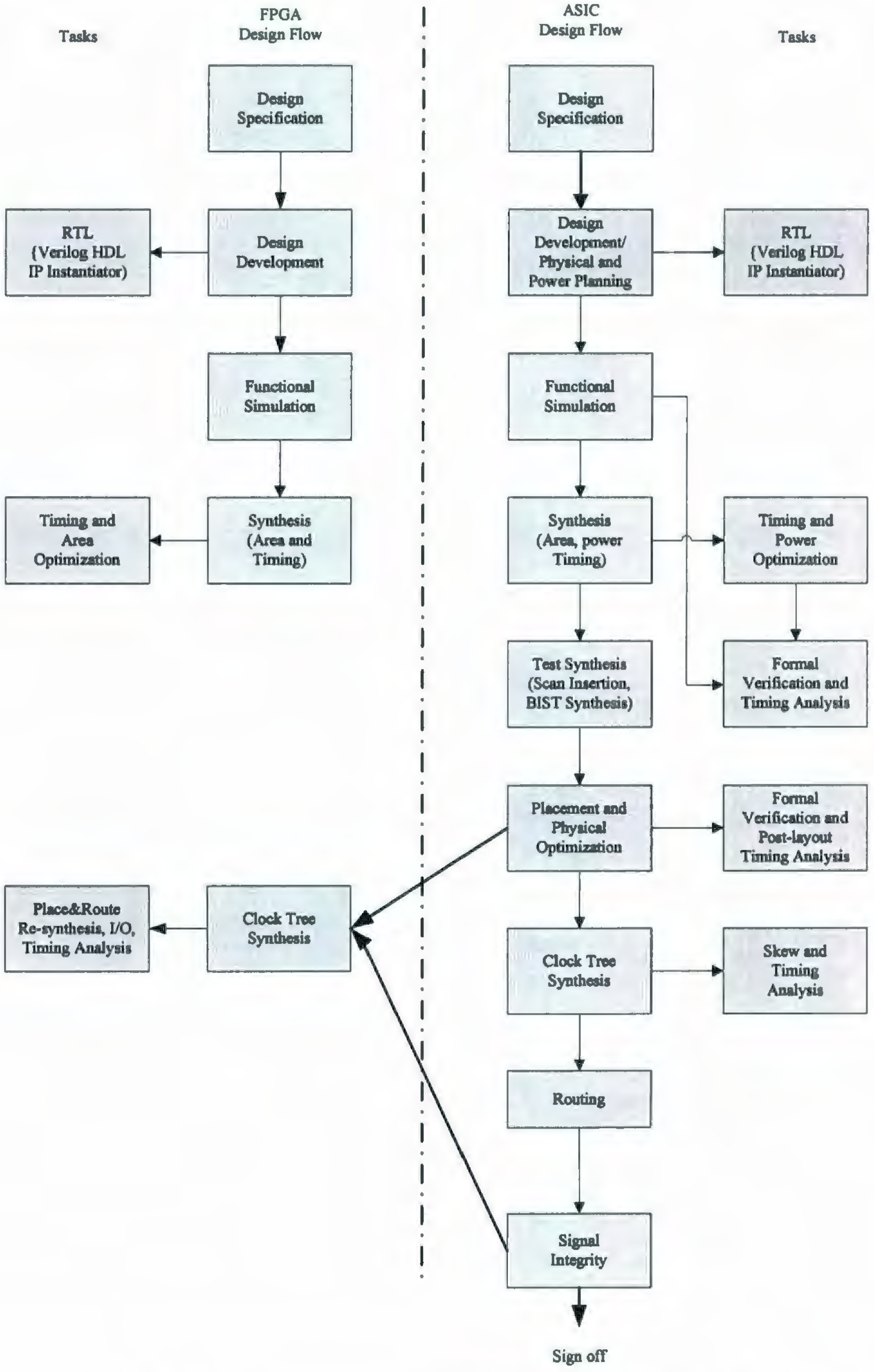

Tasks
FPGA Design Flow
ASIC Design Flow
Tasks
Design Specification
Design Specification
RTL (Verilog HDL IP Instantiator)
Design Development
Design Development/ Physical and Power Planning
RTL (Verilog HDL IP Instantiator)
Functional Simulation
Functional Simulation
Timing and Area Optimization
Synthesis (Area and Timing)
Synthesis (Area, power Timing)
Timing and Power Optimization
Test Synthesis (Scan Insertion, BIST Synthesis)
Formal Verification and Timing Analysis
Placement and Physical Optimization
Formal Verification and Post-layout Timing Analysis
Place&Route Re-synthesis, I/O, Timing Analysis
Clock Tree Synthesis
Clock Tree Synthesis
Skew and Timing Analysis
Routing
Signal Integrity
Sign off

Figure 2.2 Comparison of ASIC & FPGA Design Flows [9]

The ASIC back-end design involves more tasks than FPGA design. In both of them, an initial design idea has to go through several steps before it is completely implemented in hardware or chips. In our design, we followed the guidelines shown in Figure 2.2, because formalized flows can improve design debugging capabilities.

In the design specification step, we used a divide-and-conquer strategy called top-down methodology, while in more concrete design development stage we chose the modular design approach bottom-up methodology. More technical details can be found in [79].

Normally, for a design with sequential logic, we have two choices before RTL coding: synchronous design or asynchronous design. In synchronous design, there is a global clock to control all registers. Asynchronous design is widely used in communication system, where two or more different clock frequencies result in different clock domains, which require extra logic for synchronization. The main disadvantage is the existence of race conditions. In ASIC libraries, registers with asynchronous built-in "reset" pin consume more area and appear slower, therefore, all of our ASIC designs use registers with an external gate on the data path for a reset. That is, the reset is routed through the data pin. The clock must be running when the reset signal is asserted. However, it is not a problem for our FPGA design as there is already a reset tree in place. All registers have a built-in reset, so no extra care is needed.

Clock gating is used in our ASIC designs for power optimization as memories have been used and data can be accessed only when the Write/Read enable signal is asserted. Read enable signal and write enable signal could be either separate or a same signal. Figure 2.3 shows an example for gated clock. Glitches may appear in such a design. However, the simulation results show that the period of the clock is long enough to ignore the glitches. FPGA designs have no such potential dangers since they have a clock enable pin that can be used to avoid the gate clocking.

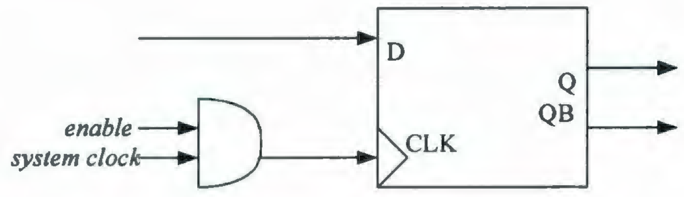


Figure 2.3 Gated Clock for Register Bit

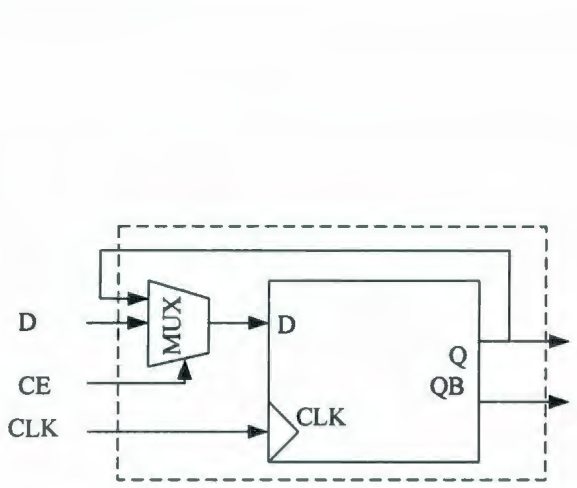


Figure 2.4 Typical Flip Flop Cell in FPGAs

2.2.2 Main Concerns in Hardware Implementation

Before mid 1990s, the limits in VLSI technology and clock rate made the pursuit for speed in a hardware implementation become a general goal. However, with the fast development of VLSI design, the achievement of sheer speed occurs much easier than before. As a result, the considerations for cost and security of cryptographic implementation start to play a more important role in many applications.

- **Speed**

The most well-known and universal techniques used to speed up the hardware implementation are pipelining and parallel processing, which allow multiple blocks of data processed simultaneously. However, neither of them is suitable for the cipher feedback modes (e.g. CBC and CFB mode), as one iteration of the encryption/decryption process needs the results of the previous one.

Moreover, different schemes are often applied based on the concrete arithmetic operations required in a cryptographic algorithm. For example, modular exponential operation is key part in public-key cryptography as the operands are usually very large. For the sake of speed, the number of time-consuming modular multiplications should be reduced as many as possible. A systolic array for modular multiplication is presented using the algorithm of P. L. Montgomery in [16]. Some other speed-up methods include Barret-Booth's method and [3] and Brickell's algorithm [25].

In some ciphers, such as Salsa20 and Phelix, which we implemented in hardware, the dominant cost operation is addition. Therefore, the subject of various adder structures is introduced and investigated in Chapter 4.

Many ciphers take advantage of S-boxes, an array lookup table using an input-dependent index. A case in point is the AES algorithm. The contents of an S-box is the multiplicative inverse in Galois Field (2^8), combined with an affine transformation over GF(2). Unfortunately, GF arithmetic approach yields up to 15 XOR gate delays [65] though it is area efficient. In most Gbps applications, a single look-up table is constructed. Till now, a twisted binary decision diagrams (TBDD) approach is the fastest reported so far [53], where the fanout of signals is distributed in all of the S-boxes.

Obviously numerous speed improvement methods may be done considering the properties of a specific algorithm.

- **Cost (area and power consumption)**

"Cost" can mean different things in real life. However, when it comes to hardware implementation, it typically indicates chip area and power consumption. Sometimes, when subtle aspects are considered, "cost" may include the actual cost financially.

The first step to evaluate the area cost of a cipher is to define a metric for comparing digital logic. Traditionally, transistors are the atomic device in digital design. Table 2.1 provides the area complexity in terms of transistors for a standard cell library from [69].

Table 2.1 Area complexity of CMOS standard cells

Components	Transistors
2-input NAND	4
2-input AND	6
2-input XOR	12
D Flip Flop	26
2:1 MUX	12

However, the popular metric nowadays is a count of the number of a standard two-input NAND gates that would be equivalent to the area of the design, since cell-based methodology makes it possible for hardware designers to focus on the high-level (logical function) aspect of digital-design. Table 2.2 is the truth table for NAND gate's behavior; Figure 2.5 is the NAND gate symbol that is in common use.

Table 2.2 The Truth Table of NAND Gate

INPUTS		OUTPUT
A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

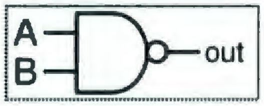


Figure 2.5 NAND Gate Symbol

For most portable devices such as mobile phones, which run on battery, power consumption is a major concern, even more important than compactness requirement, because the development of the techniques for energy storage is far behind that for cell technology [43]. Another concern related to power consumption is the cooling system of the chip. For some futuristic applications like smart card, the chip runs at ultra-low power. Sequential logic is carefully employed since flip flops require a clock to drive them, which consumes extra power. The relative power consumption for various CMOS components is illustrated in Table 2.3.

Table 2.3 Power Consumption of 0.18 μm CMOS Standard Cells [43]

Components	Normalized Power
2-input NAND	1
2-input AND	2.14
2-input XOR	3.36
D Flip Flop	22.55
2:1 MUX	2.77

The methods used to reduce power can be divided into two categories: technology-based approaches and architecture-based approaches. Technology-based approaches focus on reducing the voltage. It is well known that for a CMOS circuit, the power dissipation is due to three aspects: dynamic power, short-circuit power and leakage power. Among them, dynamic power is the largest. It is given by the equation shown below:

$$P_{dyamic} = N\,(f \times C_L \times V^2_{dd})$$

where f is the clock rate, C_L is the switched capacitance, V_{dd} is the supply voltage and N is the number of gate switches. Thus, power reduction can be achieved by voltage decrease.

However, Kakumu and Kinugawa's critical voltage provides an lower bound on the supply voltage [48]. Therefore, it is more attractive to make efforts on architectural improvement. Moreover, architecture-based approaches often make compensation for the reduced circuit speed that is due to the lower operating voltage (CMOS gate delay increases according to the voltage decrease).

The authors of [4] illustrate several architectural methods. One of them is a parallel structure, which makes sacrifice in area. This can be seen from Figure 2.6.

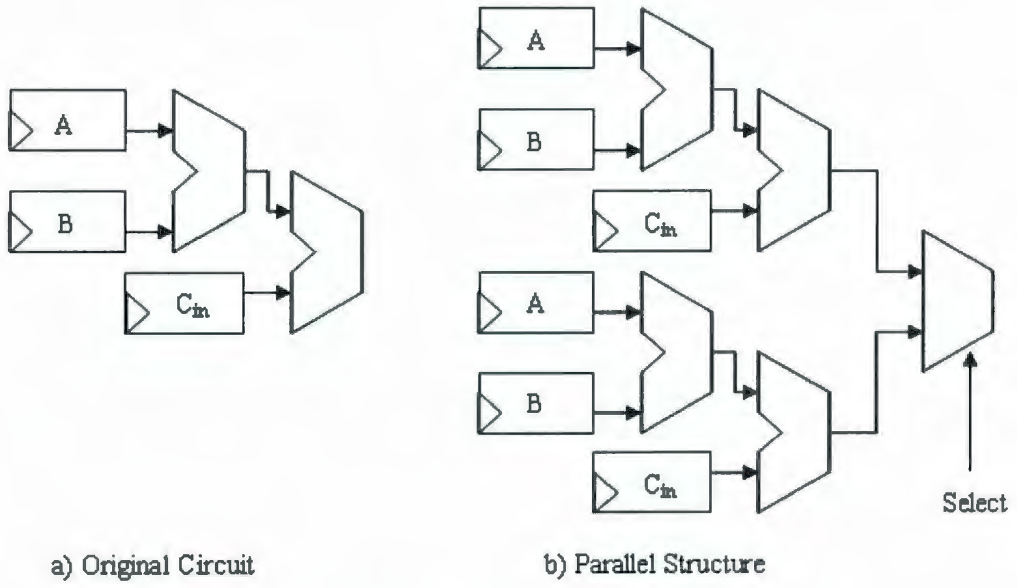


Figure 2.6 Parallel Scheme for Power Reduction [4]

The circuit shown in Figure 2.6.b runs at half of the original frequency with the same throughput. It should be note that the power consumption is not doubled though it looks like that with a doubled circuit structure. Assuming the original power consumption is P_1

$= N (f_1 \times C_{L1} \times V^2_{dd\ 1})$, then the new power assumption is $P_2 = 2.15N (f_1/2 \times C_{L1} \times (0.58V_{dd\ 1})^2) \approx 0.36P_1$.

- **Security/ Implementation Attacks**

In addition to implementation efficiency (i.e. speed and area), security is very important. Implementation or side channel attacks are targeted on the possible weaknesses in specific implementation platforms instead of in the algorithm. These attacks include power analysis [59] and fault attacks [38].

In 1996 Kocher et al. introduced the concept of a timing attack, which is one of the general classes in side-channel attacks, by measuring how much time different computations take to perform to reveal sensitive information. For example, with detailed knowledge of a cryptographic algorithm, it is sometimes possible to determine the length of the key by watching data transmission with the CPU. Two years later Kocher et al. [59] proved that the power consumption in a cryptographic circuit could reveal the secret information too. In [38], a comprehensive study at fault attacks is provided. In a fault attack, errors are injected to the cryptographic core, while the resulting outputs reflect the faults. Current methods to avoid fault attacks, especially on symmetric ciphers, include linear error-detecting codes, such as Reed-Solomon code [64]. Rather than focusing on the overall possible faults, they concentrate on a certain group of possible faults for practical reason.

However, side-channel attacks do not threaten stream ciphers as much as they do to block ciphers. This is because, in a stream cipher, the keystream process is typically independent of externally known data. In [15], the authors pointed out that side-channel attack related publications on stream ciphers have only aimed at RC4 stream cipher. Attacks on other stream ciphers, especially LFSRs based ones are not available in the public literature yet.

2.3 Software/Hardware Codesign

Considering the advantages and disadvantages of software and hardware implementations, it is believed that some hybrid co-designs can provide excellent performance while maintaining the flexibility. Smart cards that are used in secure financial applications and e-commerce is an example of software/hardware codesign. In a smart card, the most computationally intensive blocks run in targeted circuit to provide the speed and more physical security, and other blocks run on a general processor for flexibility.

The main concern of co-designs is which parts of the cipher are mapped into hardware and software. There are two trends. One aims at adding hardware resources to a general purpose processor [45]. For example, a new instruction set is also added to the processor to facilitate cryptographic implementation. The other trend is to divide cryptographic algorithms into two stand-alone components; one runs on general processor and the other one runs on additional core. In [60], a stream cipher is modeled abstractly, as shown in Figure 2.7. It indicates that the stream cipher kernel and the iterated state variables are mapped into hardware (HW) while software (SW) provides the initial key and the nonce, which is often a pseudo-random number used only once. The keystream and the plaintext stream processing are also handled by software.

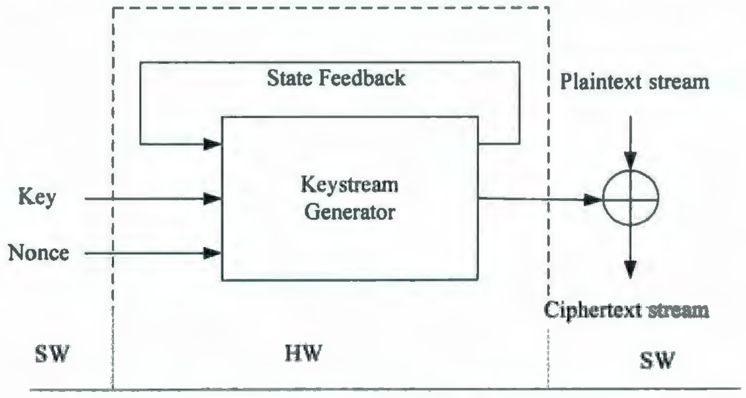


Figure 2.7 Example Scenario for a Stream Cipher [60]

FPGAs are practically attractive to use in co-designs. The overwhelming problem to resolve before implementing a software/hardware codesign is how to split the work between the general processor and the application specific circuit. This problem includes many aspects, some of which can be very subtle. However, the decisions on the work which is allocated to FPGAs are relatively simpler to make because of FPGAs' inherent flexibility. FPGAs are very suitable to process large amounts of data, which indicates that a complex computational component or a bus interface can be implemented in them.

A key factor in deciding how much computational task to assign to an FPGA is the “logic gate equivalence”. It is the total number of logic gates that can work at the same time. Figure 2.8 shows a typical structure of FPGA logic block. However, in practice many blocks will not work in a computation even if they are supposed to. It is largely because of improper routing, which is normally done by EDA tools instead of manually. The problem of routing problem for real implementation is studied by Inuani and Saul in [49].

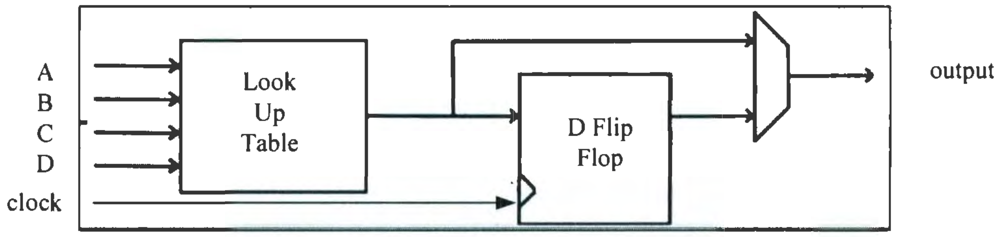


Figure 2.8 Typical FPGA Logic Block

One of the most attractive uses of FPGA in co-designs is the soft processors which can be configured to suit different applications. For example, the Virtex chips from Xilinx have 32-bit built-in soft processor known as MicroBlaze [52], which provides high speed hardware/software interfaces. This technique allows the designers to take an FPGA as a simple cross-compiler. In [36], the authors investigated the implementation of several most typical cryptographic algorithms based on MicroBlaze. They found that the flexibility offered by FPGAs can be used to notably increase the throughput of a software/hardware hybrid system.

2.4 Summary

This chapter addresses the methodologies and main concerns for software and hardware implementations, respectively. In addition, it draws analogies between typical FPGA and ASIC design processes.

Various trade-offs are very important. In software implementation, it is often reasonable to generate ciphertext for block cipher or keystream for stream cipher in blocks, whose size equals a multiple of the word size of the processor. As a result, the available bandwidth in the system can get optimum use.

Software inherently has the ability for fine-granular control on the internal configuration and behavior of a cryptographic algorithm by using particular instructions.

However, performing such operations would require extra memory and execution time which decrease the speed of the implementation. In hardware implementation, the typical trade-off concern is the speed-area tradeoff. Considering the pipeline structure in Figure 2.1, it is obvious that with more pipeline stages there is more speed gain but higher area consumption.

The following two chapters illustrate the details of our hardware designs for Phelix and Salsa20 stream ciphers. Both of the ciphers were claimed to be designed with the special emphasis on their suitability for not only software implementations but also hardware implementations.

Chapter 3 Hardware Implementation of the Phelix Stream Cipher

3.1 Introduction of Phelix Stream Cipher

Phelix [22] is claimed to be a high-speed stream cipher. It is selected for both software and hardware performance evaluation by the eSTREAM project. The cipher supports an 8-bit to 256-bit length key and a 128-bit nonce to generate the keystream bits. The plaintext is incorporated during the computation to produce a built-in Message Authentication Code (MAC).

Our goal in implementing Phelix is to find out a reasonable synthesis result for two extreme situations: the compactness it can achieve without considering the throughput; the speed/throughput it can achieve without considering the area. Since circuits implemented in FPGAs are at least ten times larger and three times slower than the custom implementations [67], we have chosen ASIC-based approach to implement the two proposed designs for Phelix: compact Phelix implementation and high speed implementation. The basic features underlying FPGAs are explored in the next chapter that introduces various structures for Salsa20 as the property of Salsa20 is more suitable for a FPGA-based implementation, such as a relatively big requirement for memory compared with some popular stream ciphers.

It should be noted that in block diagrams representing the designs, the input and output are not real I/Os in a system. The designs are core based and the input and output could be internally interfaced with external circuits.

3.1.1 Algorithm

Phelix is targeted at 32-bit platforms. It is composed of simple operations: addition modulo 2^{32}, exclusive or, and rotation by a fixed number of bits. There are 5 words that are updated during each round, and 4 "old" words are stored in memory to be used in the keystream output function.

One block that produces one word of keystream consists of two "half-block" functions

H, which is defined as:

```
Function $H(w_0, w_1, w_2, w_3, w_4, K_0, K_1)$
Begin
    $w_0 := w_0 \boxplus (w_3 \oplus K_0);$    $w_3 := w_3 \lll 15;$
    $w_1 := w_1 \boxplus w_4;$                $w_4 := w_4 \lll 25;$
    $w_2 := w_2 \oplus w_0;$                  $w_0 := w_0 \lll 9;$
    $w_3 := w_3 \oplus w_1;$                  $w_1 := w_1 \lll 10;$
    $w_4 := w_4 \boxplus w_2;$                $w_2 := w_2 \lll 17;$

    $w_0 := w_0 \oplus (w_3 \boxplus K_1);$   $w_3 := w_3 \lll 30;$
    $w_1 := w_1 \oplus w_4;$                  $w_4 := w_4 \lll 13;$
    $w_2 := w_2 \boxplus w_0;$                $w_0 := w_0 \lll 20;$
    $w_3 := w_3 \boxplus w_1;$                $w_1 := w_1 \lll 11;$
    $w_4 := w_4 \oplus w_2;$                  $w_2 := w_2 \lll 5;$
    Return $(w_0, w_1, w_2, w_3, w_4);$
End.
```

Figure 3.1 H Function

The bitwise exclusive-or of two words, denoted as "$\oplus$", is the sum of the words with carries suppressed. The symbol "<<<" represents left rotation, and "$\boxplus$" represents addition modulo 2^{32}.

During the encryption, defining one step as a complete step for Phelix to update five active state words (w_0, w_1, w_2, w_3, w_4), at ith step, two 32-bit secret subkeys ($X_{i,0}$, $X_{i,1}$) and one plaintext word P_i are applied to executions of the H function. The generation of the subkeys is introduced in Section 3.3.5.

Selected details of the algorithm will be discussed in the analysis of Phelix cipher main components. For more insight of those components, refer to the initial paper [22].

3.1.2 Security

The authors of Phelix claimed that there is no attack against Phelix with less than 2^{128} operations. Actually, Phelix is a strengthened version of an earlier cipher, Helix [55], which was targeted by two attacks in 2004. One is an adaptive chosen-plaintext attack created by Muller [26]. The other one was also published by Muller, which is a distinguishing attack [26]. The Phelix primitive was largely motivated to increase the security against the distinguishing attack. It expands Helix's 160-bit internal state to 288 bits.

However, in November 2006, Hongjun Wu and Bart Preneel published a paper titled "Differential Attacks against Phelix" [35]. This paper was largely derived from the differential attack against Helix stream cipher to recover the key. Their work mainly aims at weak nonces and keys. They concluded that Phelix fails to strengthen Helix since the computational complexity of the attack is much less than that of the attack against Helix. However, a good cipher does not imply absolute security, and the confidence level in the amount of security is not merely dependent on several results of attacks. Selecting a potential prevalent cipher is even more complicated, and security is only one consideration among a large pool.

3.1.3 Previous Work on Hardware Implementation on Phelix

Until now there has been insufficient work on the hardware implementations of the candidate stream ciphers proposed for eSTREAM project. Most discussion and comparison focus on cryptanalysis. However, the authors of [68] demonstrated the FPGA performance of Phelix. The results are shown in Table 3.1. In order to avoid specific metric for individual devices, the authors of [68] chose the number of equivalent gates to measure area, where one gate is a two-input NAND gate (6 transistors).

Table 3.1 FPGA Hardware performance of Phelix [68]

Xilinx Chip	Slices	Throughput Mbit/s	Gate Equiv Estimate	Implementation Description
XC2S100-5	1198	960.0	20404	(A) full-round 160-bit design, as per developers paper
XC2S100-5	1077	750.0	18080	(B) half-round 160-bit design
XC2S30-5	264	3.2	12314	(C) 32-bit data path

The software/hardware codesign of Phelix has been studied in an undergraduate course at Virginia Tech called "Introduction to Codesign" [60]. They implement two identical half function blocks as the hardware component.

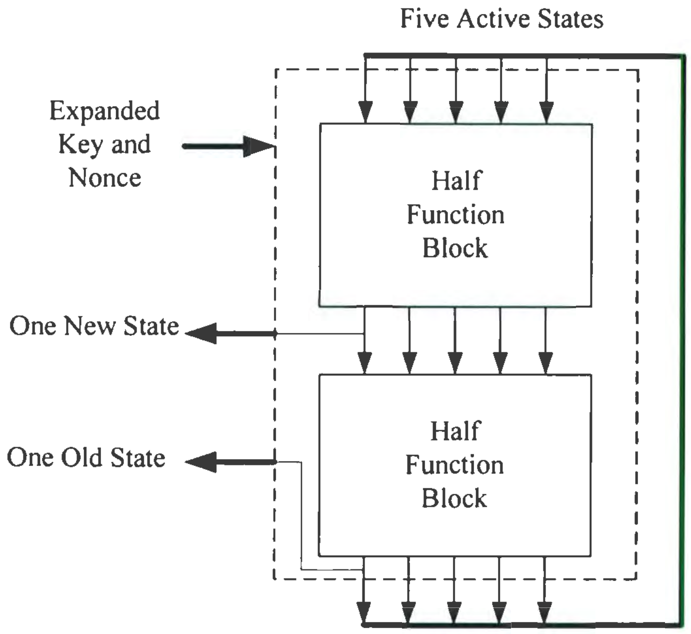


Figure 3.2 Block Diagram of Hardware Accelerator for Phelix

The hardware design of one student group shown in Figure 3.2 is of pure combinational logic, which can accomplish one state updating process in a single clock cycle, and leave the whole control complexity in software.

Another two teams are more aggressive since the one implemented a half block as the basic iterative H function circuit, and the other one implemented eight double H function blocks as a parallel structure to relieve the communication bottleneck between software and hardware. Obviously, the synthesis result for the eight-full-H-block is quite large. A careful trade-off decision should be made. Table 3.2 shows the results for those three structures. All of them are implemented targeted on Xilinx Spartan3e FPGAs.

Table 3.2 Results of Phelix Co-design

Version	Area (slices)	Frequency(MHz)
8x Full Block	4301	7.7
Full Block	1190	48.6
Half Block	459	76

Unfortunately, the results are not good for comparison since the authors only provide the frequency instead of the throughput.

3.2 Compact ASIC Structure of Phelix

3.2.1 Top Level Design

The Phelix stream cipher can be implemented in many ways. The proposed compact structure focuses on function sharing to optimize the area. Figure 3.3 illustrates a minimal ASIC implementation consisting of one round of encryption and a memory recording the four old states. The specifications of the main blocks are given below:

n_expand: converts a 128-bit input nonce to the 256-bit working nonce.

key_mix: converts a variable-length input key to the fixed-length working key.

subkey_gen: generates subkeys $X_{i,0}$, $X_{i,1}$ for each block.

ini_dp: decides the input of H_func. For the first eight blocks (initialization phase), the generated keystream is discarded.

H block: performs function H $(w_0,w_1,w_2,w_3,w_4, K_0,K_1)$.

FIFO: the "first in, first out" memory that stores the old states.

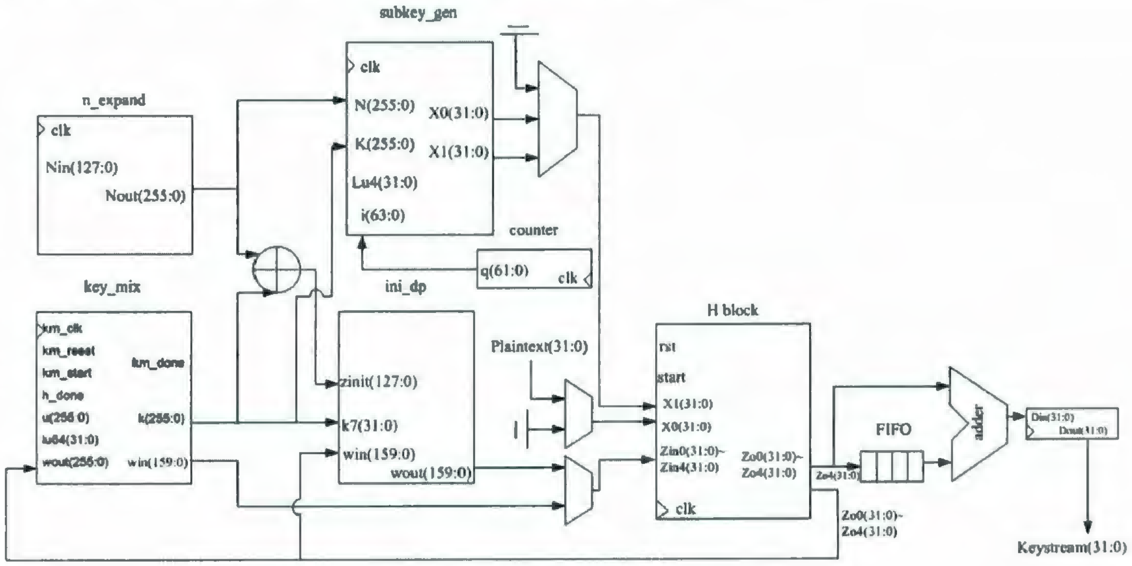


Figure 3.3 Phelix Compact Structure

The top level controller can be presented by a finite state machine shown in Figure 3.4. When the system is turned on with the power plugged in or the *Reset* signal is triggered, the controller transfers to *Idle* state. When the *start* signal is asserted, the encryption circuit starts to work.

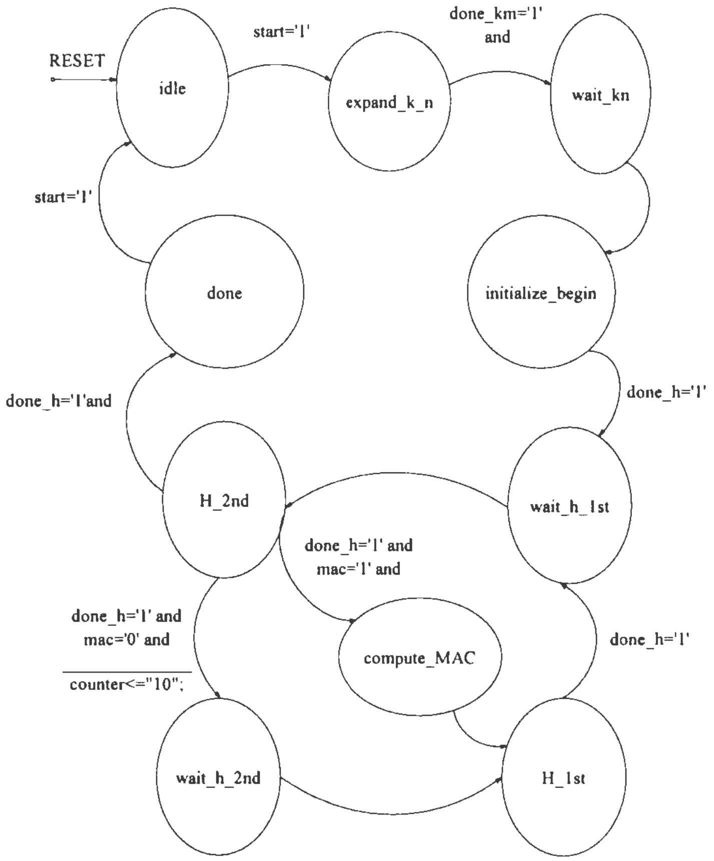


Figure 3.4 State Diagram of the Top Level Controller

The specifications of the states are given below:

expand_kn: Convert a input key/nonce to the fixed-length working key/nonce.

wait_kn: While stopping *key_mix* block and *n_expand* block, enable *subkey_gen* block, putting the first old state into the FIFO.

initialize_begin: Set the initial five states of the H function, and enable the counter, which provides the block number.

wait_h_1st: Generate the keystream, and increase the block number.

H_1st: Do the function

$(Y_0^{(i)}, Y_1^{(i)}, Y_2^{(i)}, Y_3^{(i)}, Y_4^{(i)}) := H(Z_0^{(i)}, Z_1^{(i)}, Z_2^{(i)}, Z_3^{(i)}, Z_4^{(i)}, 0, X_{i,0})$, where Y, X and Z are 32-bit words, and i represents the block number.

H_2nd: Do the function

$(Z_0^{(i+1)}, Z_1^{(i+1)}, Z_2^{(i+1)}, Z_3^{(i+1)}, Z_4^{(i+1)}) := H(Y_0^{(i)}, Y_1^{(i)}, Y_2^{(i)}, Y_3^{(i)}, Y_4^{(i)}, P_i, X_{i,1})$, where P_i is the plaintext word.

wait_h_2nd: Store Z_4 in the FIFO after each round.

Compute_MAC: Begin the MAC generation process.

Since this design objective is sensitive to area, each sub-component is chosen to simply minimize the area. For instance, the rotation is realized by reordering the interconnections between logic cells instead of using shift registers. The discussion presented in this section concerns the structure of each main component that is implemented for the Phelix cipher. These basic components include 32-bit adder, H function block, key mixing block, nonce-expanding block and subkey generator.

3.2.2 32-bit Adder

Compared with other components, the 32-bit adder is the most expensive operation because of the speed and the area it consumes. The most convenient way to perform this operation is to use a ripple carry adder (RCA).

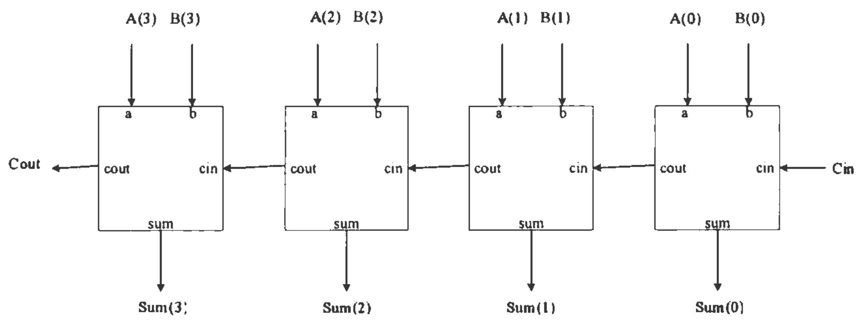


Figure 3.5 A 4-bit RCA

RCA is a straightforward adder and the layout is quite simple, which facilitates fast design. However, the speed of an adder mainly depends on the time taken by the carry chain. Since each full single-bit adder has to wait for the carry bit to be propagated from the previous adder, the delay is very significant, especially in 32-bit computation.

Our design was developed for low resource, so the simple 32-bit RCA is used by sacrificing the throughput. The output is unregistered.

3.2.3 H Function Block

H function block and the adder are the basic function sharing components. Figure 3.6 is a block diagram that consists of two H functions:

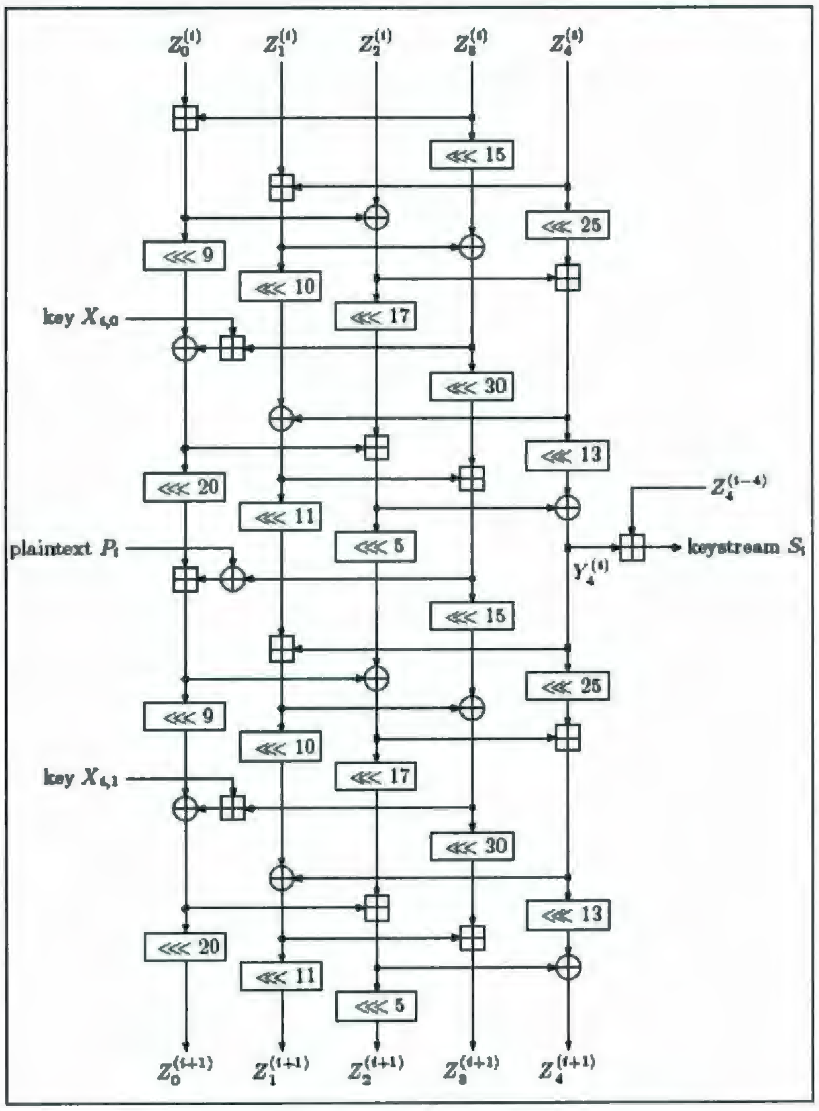


Figure 3.6 One block of Phelix Encryption [22]

Based on the sources of the input data, the H function block is used in four phases.

- In the key mixing process, the H function is used to create an R function that generates eight 32-bit working keys.

```
Function R($w_0$,$w_1$,$w_2$,$w_3$)
Begin
Local Variable $w_4$ := l(U) + 64; // l(U) is the number of words in the input key.
($w_0$,$w_1$,$w_2$,$w_3$,$w_4$) := H($w_0$,$w_1$,$w_2$,$w_3$,$w_4$,0,0);
($w_0$,$w_1$,$w_2$,$w_3$,$w_4$) := H($w_0$,$w_1$,$w_2$,$w_3$,$w_4$,0,0);
Return ($w_0$,$w_1$,$w_2$,$w_3$);
End.
```

- In the initialization phase, the input of the H function is set as:

  $Z_j^{(-8)} := K_{j+3} \oplus N_j$ for j = 0, 1, 2, 3

  $Z_4^{(-8)} := K_7$

  $Z_4^{(i)} := 0$ for i = -12, -11, …, -9

  $P_i := 0$ for i = -8, -7, …, -1

  where i represents the block number, $Z_j^{(i)}$ is the jth word of the ith block, P_i is the ith word of the plaintext.

- After the initialization, H function is employed as below:

  $(Y_0^{(i)}, Y_1^{(i)}, Y_2^{(i)}, Y_3^{(i)}, Y_4^{(i)}) := H(Z_0^{(i)}, Z_1^{(i)}, Z_2^{(i)}, Z_3^{(i)}, Z_4^{(i)}, 0, X_{i,0})$

  $(Z_0^{(i+1)}, Z_1^{(i+1)}, Z_2^{(i+1)}, Z_3^{(i+1)}, Z_4^{(i+1)}) := H(Y_0^{(i)}, Y_1^{(i)}, Y_2^{(i)}, Y_3^{(i)}, Y_4^{(i)}, P_i, X_{i,1})$

  The ciphertext words are computed by $C_i := P_i \oplus S_i$, where $Si := Y_4^{(i)} + Z_4^{(i-4)}$ and i ranges from 0 to 2^{64}-1.

- Just after the last word of the plaintext is encrypted, the internal state word $Z_0^{(i)}$ will be XORed with the constant value 0x912d94f1, and the modified state is a new input for the H function block to start the post-mixing process, which computes the MAC.

The simplified block diagram of H function is shown below:

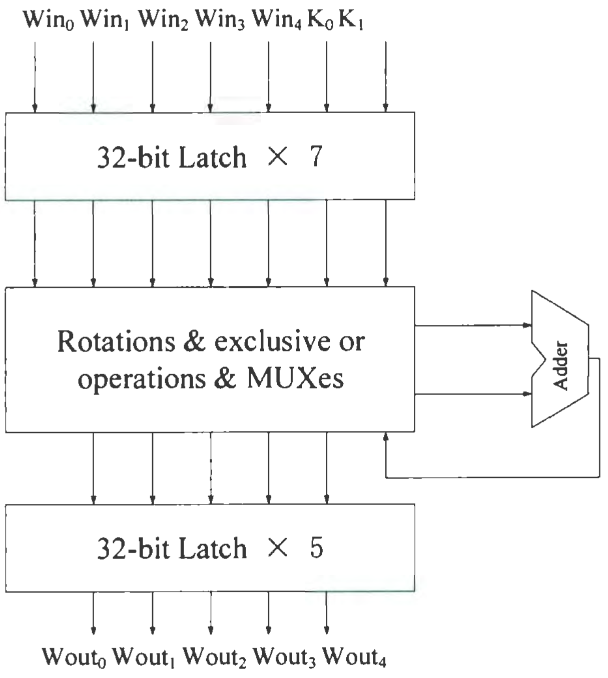

Win_0 Win_1 Win_2 Win_3 Win_4 K_0 K_1
32-bit Latch × 7
Rotations & exclusive or operations & MUXes
Adder
32-bit Latch × 5
$Wout_0$ $Wout_1$ $Wout_2$ $Wout_3$ $Wout_4$

Figure 3.7 Simplified Datapath of the H Function Block

The multiplexers (MUXes) connected to the input ports of the adder in the circuit are of six input sources and a single output for adder sharing. To increase the speed of the encryption, we could design additional logic to perform the H function. It would require more adders and 32-bit exclusive-or function blocks that can work in parallel. However, it will dramatically increase the size of the H function circuit since the adder is the largest component compared with other simple function blocks, such as a 32-bit register.

The complete block function is illustrated in Figure 3.8. The five input words are lathed when the H function block starts to work. Using a latch instead of a register might cause noisy inputs: if there is any glitch on the input of the latch, then, it will be propagated directly to the output. However, the area consumed by a latch is typically less than that consumed by a register. Two six-to-one MUXes are used to select the input for the single adder in the datapath of the H function block. The selection signals are given by an FSM. The results from the adder are stored in six latches.

3.2.4 Key Mixing Block

The key mixing block maps a variable-length input key to eight workings keys. Define function R as shown below:

```
Function R($w_0$, $w_1$, $w_2$, $w_3$)
    Begin
        Local Variable $w_4$ := l(U) + 64;
        ($w_0$, $w_1$, $w_2$, $w_3$, $w_4$) := H($w_0$, $w_1$, $w_2$, $w_3$, $w_4$, 0, 0);
        ($w_0$, $w_1$, $w_2$, $w_3$, $w_4$) := H($w_0$, $w_1$, $w_2$, $w_3$, $w_4$, w4, 0, 0);
        Return($w_0$, $w_1$, $w_2$, $w_3$);
    End
```

Then, the key mixing process is a recursion:

$(K_{4i}, K_{4i+1}, \ldots, K_{4i+3}) := R(K_{4i+4}, K_{4i+5}, \ldots, K_{4i+7}) \oplus (K_{4i+8}, K_{4i+9}, \ldots, K_{4i+11})$ for $i = 7, 6, \ldots, 0$, where K_i represents a 32-bit word. The words $(K_0, K_1, \ldots, K_7)$ forms the working keys of the cipher.

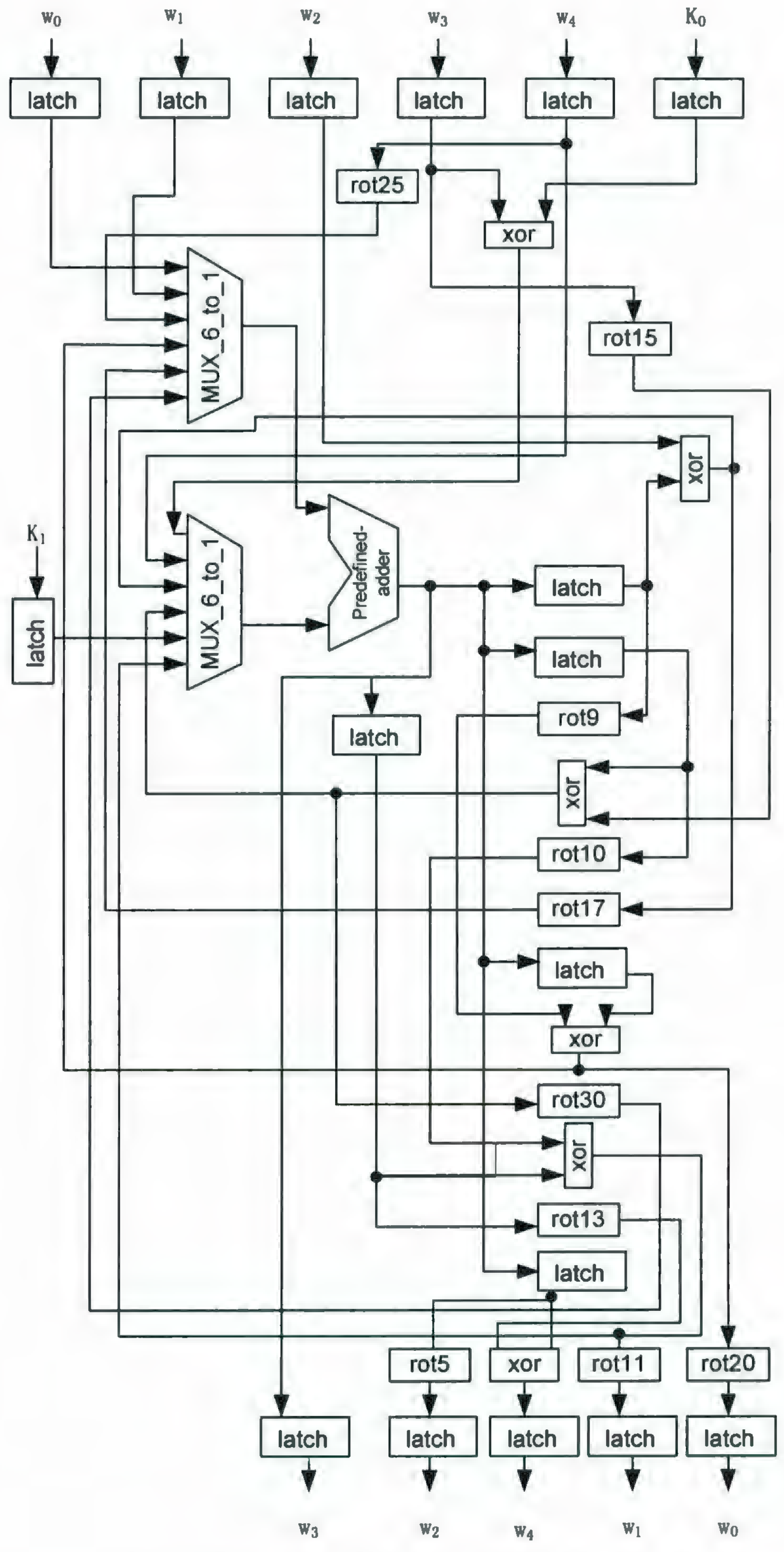


Figure 3.8 Complete Datapath of the H Function Block

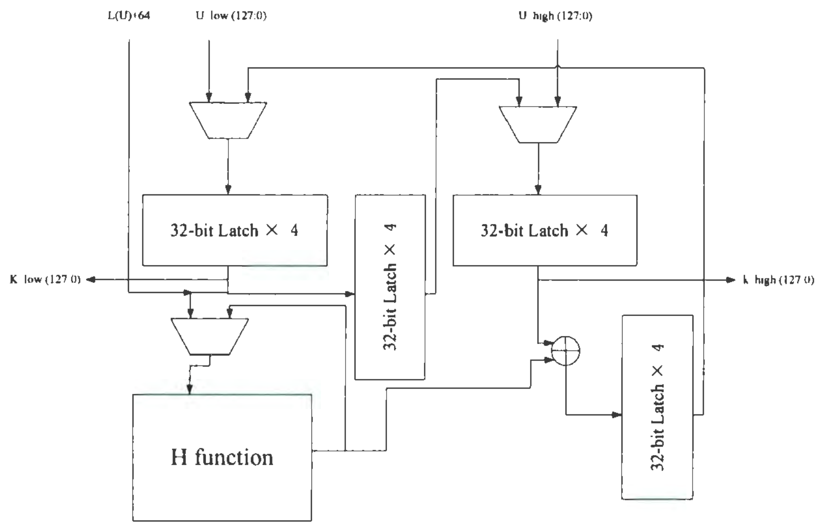


Figure 3.9 Datapath of the Key Mixing Block

There are four latch blocks (32-bit × 4) in the circuit. Two of them record the result of each recursion, including the final working keys; the other two are for the temporary variables. Although the H function block is shown in key mixing block's datapath, it is not exclusive but a basic public component in the top level structure.

3.2.5 Nonce Expanding Block

The nonce-expanding block is to extend a 128-bit nonce to the fixed 256-bit words by defining $N_k := (k \bmod 4) - N_{k-4} \pmod{2^{32}}$ for k = 4, 5, …, 7.

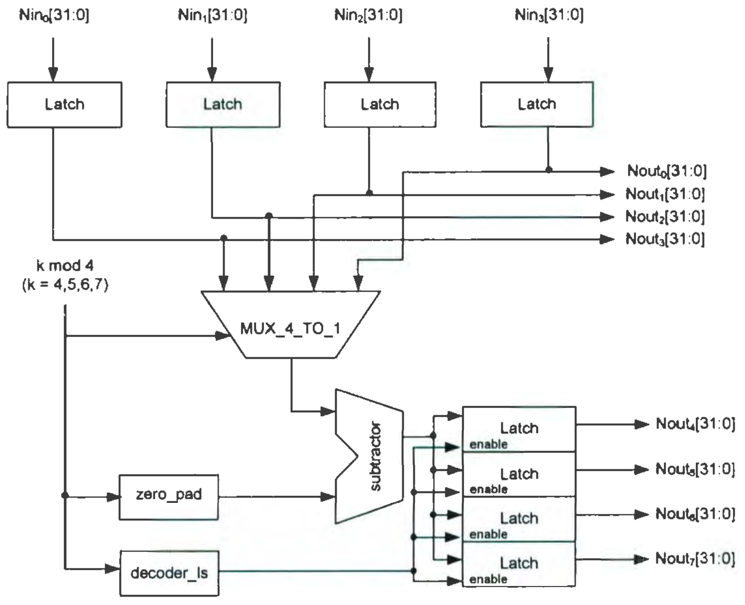


Figure 3.10 Datapath of Nonce Expanding Block

The eight outputs of the block diagram are the expanded nonce $N_0, N_1, \ldots, N_7$. Usually, the modulo operation is implemented such that division with remainder is calculated each time. In this way, it can be slower and a waste of area. But for the case in Phelix nonce expanding, (k mod 4) equals (k and 3) in a bitwise operation. Thus we simply use an AND gate array instead of divider and comparator for efficiency.

When k ranges from 4 to 7, (k mod 4) is of exactly 3 bits. So, it requires a *zero_pad* block to extend it to a 32-bit word for the 32-bit subtractor. The result of (k mod 4) can be used to notify one output latch to load the data at each clock cycle by using a block called "*decoder_ls*".

3.2.6 Subkey Generator

The computation of subkeys can be done either on-chip or off-chip. If it is realized off-chip, in which subkey generation is performed outside and then is downloaded into the circuit memory, it requires only a memory. However, it may affect the security of the device. Thus, we implement the subkey generator block on-chip and provide additional block of hardware for this operation.

According to [22], the subkey words for block i are defined by

$$X_{i,0} := K_{i \bmod 8}$$

$$X_{i,1} := K_{(i+4) \bmod 8} + N_{i \bmod 8} + X_i' + i + 8$$

$$X_i' = \begin{cases} \lfloor (i+8)/2^{32} \rfloor & if (i \bmod 4) = 3 \\ 4 \times l(U) & if (i \bmod 4) = 1 \\ 0 & others \end{cases}$$

Only when the working keys and the expanded nonce are ready to be used, the system controller can send a *start* signal to the subkey generator.

It is easy to notice that all indices such as (i mod 8) and ((i+4) mod 8) are decided by the three least significant bits of the block number i. It indicates that we can take use of these three bits as the select signals for the three MUXes, whose outputs are $K_{i \bmod 8}$, X_i' and $N_{i \bmod 8}$, respectively. The details are shown below:

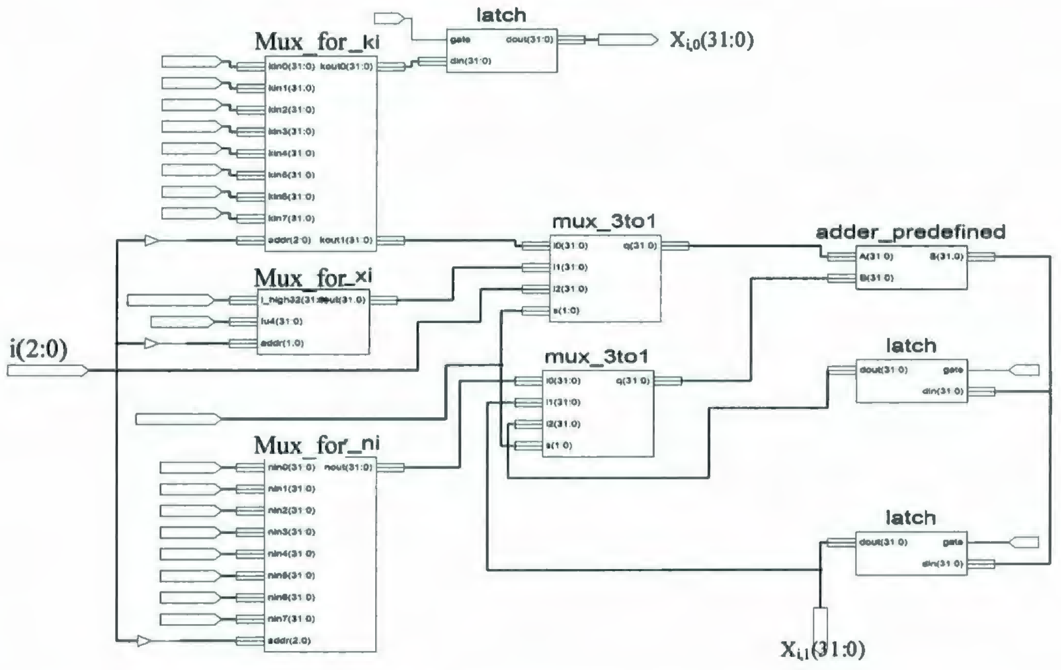


Figure 3.11 Datapath of Subkey Generator

The block number starts from -8 in [22]. However, we start the block number from 0 in the initialization phase. Therefore, there is no need for adder to operate on the counter output i and the constant 8.

3.3 High Speed ASIC Structure of Phelix

The high speed implementation of Phelix stream cipher is not an overall change of the compact design; it only removes the features that will achieve compactness with the sacrifice of speed. For example, in the compact version, only a single carry ripple adder is used in H function block, while in the high speed version, six Kogge-Stone adders [46] are implemented for each addition. Various adder structures will be introduced in the following chapter since the structures of Salsa20 have chosen more adders to cater to the corresponding features.

The figure below illustrates the top level of high speed Phelix implementation. To make it easy for understanding, we ignore some obvious I/O, such as the input reset. It is a synchronous design. Thus, all *clk* signals shown on each sub-block are driven by a global clock signal. In Sections 3.3.1~3.3.4, we will give the details of each sub-block shown in the top level block diagram in a bottom-up way.

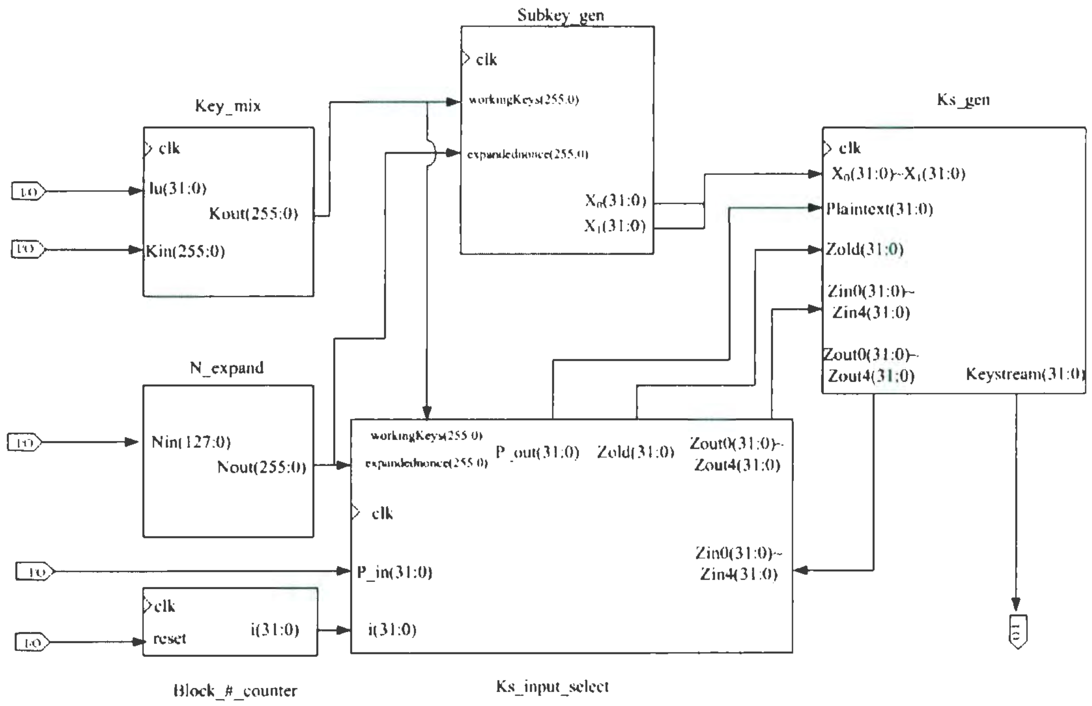


Figure 3.12 Phelix High Speed Structure

3.3.1 H Function Block

The H function block is the most important component that decides the speed of the whole implementation. It is used in four different phases as we have introduced in the previous section. They are: key mixing phase, initialization phase, normal encryption phase and MAC generation phase. To speed up H function block, we split the six add operations into two groups: the group one includes ks_adder0, ks_adder1, ks_adder2, ks_adder3 and the group two includes ks_adder4 and ks_adder5. Since the second group's inputs depend on the output of the first group, they can not work in parallel. The main cost from the input to the output is the time consumed by the two adders. In this design, the combinational logic data path in front of the output registers can be divided into two clock cycles. However, no internal registers are added. It is controlled by a D flip flop, whose output connects to the load enable signal of the output registers. Therefore, the output is loaded every two clock cycles. The alternative method is to remove the D flip flop and make the whole H function completed in a single clock cycle. But it is not an efficient way as it may cause clock cycle waste when doing other work, such as XOR operation.

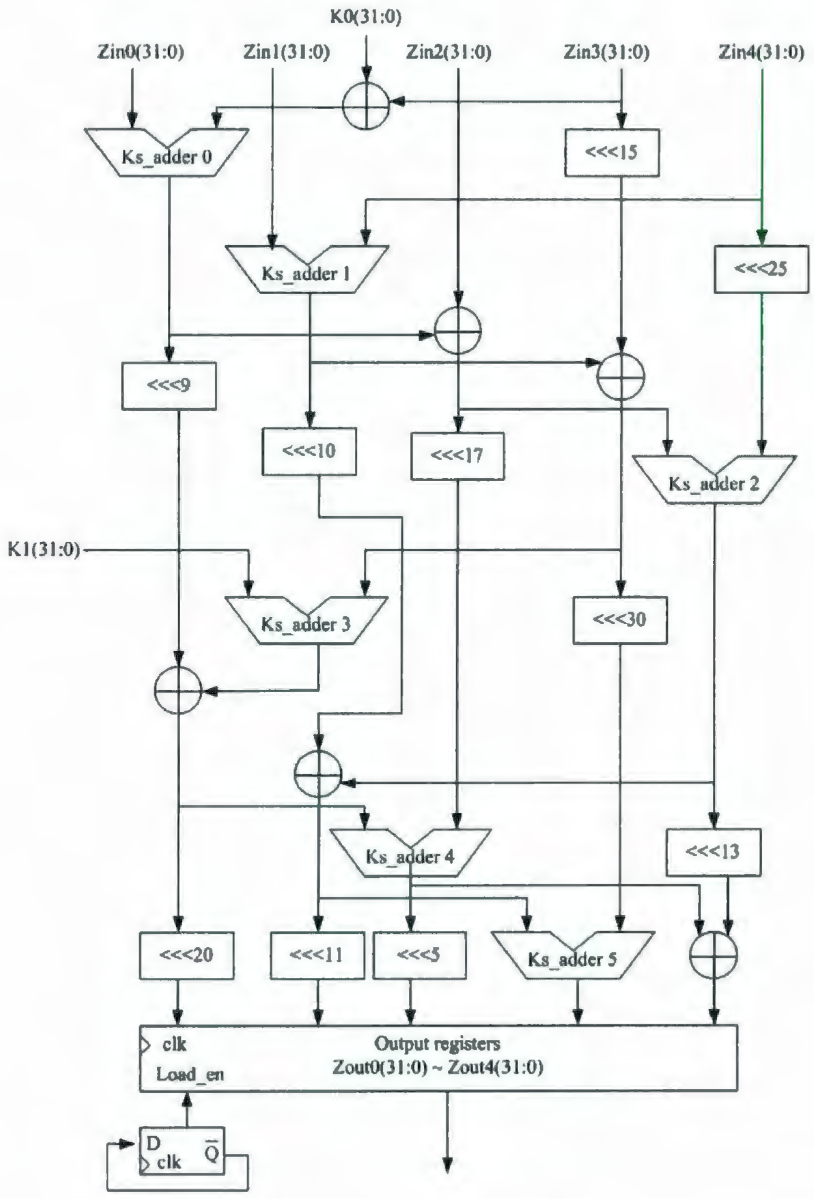


Figure 3.13 H Function Block

3.3.2 Key Mixing Block

Key mixing block converts a variable-length input key to the fixed-length working keys. Each working key is a 32-bit word. The main components in key mixing block is called R function block, which is introduced in section 3.2.4.

Firstly, it expands the input key into eight 32-bit words by padding with zeros to the most significant bits, if the length of the input key is smaller than 256 bits. Then, it performs eight iterations; each one consists of an R function and an XOR function to produce an array of four new words as shown below:

$$(K_{4i},\dots,K_{4i+3}) = R(K_{4i+4},\dots,K_{4i+7}) \oplus (K_{4i+8},\dots K_{4i+11}) \quad i = 7, 6, \dots, 0$$

R function block is composed of two H blocks and it takes four clock cycles to accomplish one loop since each H function takes two. Consequently, 32 clock cycles (8×4) are required for the key mixing phase to produce eight working keys. There are eight R function blocks in key mixing block in total.

3.3.3 Nonce Expanding Block

As shown in the top level block diagram, there is no *clk* as an input for *N_expand* block. It is pure combinational logic. Nonce expanding only happens in the initialization phase. So, it is not a very critical module for high speed design, which is focused on quickly producing keystream bits. The type of adder/subtractor in this block can be either ripple carry or Kogge-Stone.

3.3.4 Keystream Generation Block

The subkey generator in our design works "on the fly". That means the subkey computation is performed during the time when the keystream generator core (*ks_gen*) works on the previous plaintext word instead of pre-computed in the initialization phase and stored in a large memory. The structure of subkey generator is similar to the compact design, except that the high speed design uses Kogge-Stone adder.

The *ks_gen* block requires four clock cycles to generate one keystream word. The reason is the same as that in the R function block: there are two H function blocks included. The data path of *ks_gen* is illustrated in Figure 3.14. Two H blocks are concatenated. The start signal for the second one is delayed by two clock cycles for signal alignment.

The *ks_gen* block not only works in the normal keystream generation phase but also the initialization phase. So, its inputs are given by a *ks_input_sel* block as shown in the top level. The *ks_input_sel* block is very similar to *ini_dp* block in the Phelix compact design. It takes one clock cycle to make an input selection corresponding to the block number which is given by a counter. When the block number is between zero to seven and the start signal is asserted, it performs an initialization phase selection; when the block number is larger than seven, it does nothing but pass the current input to the output directly. The FIFO that is used to store the previous four old states is combined in the *ks_input_sel* block.

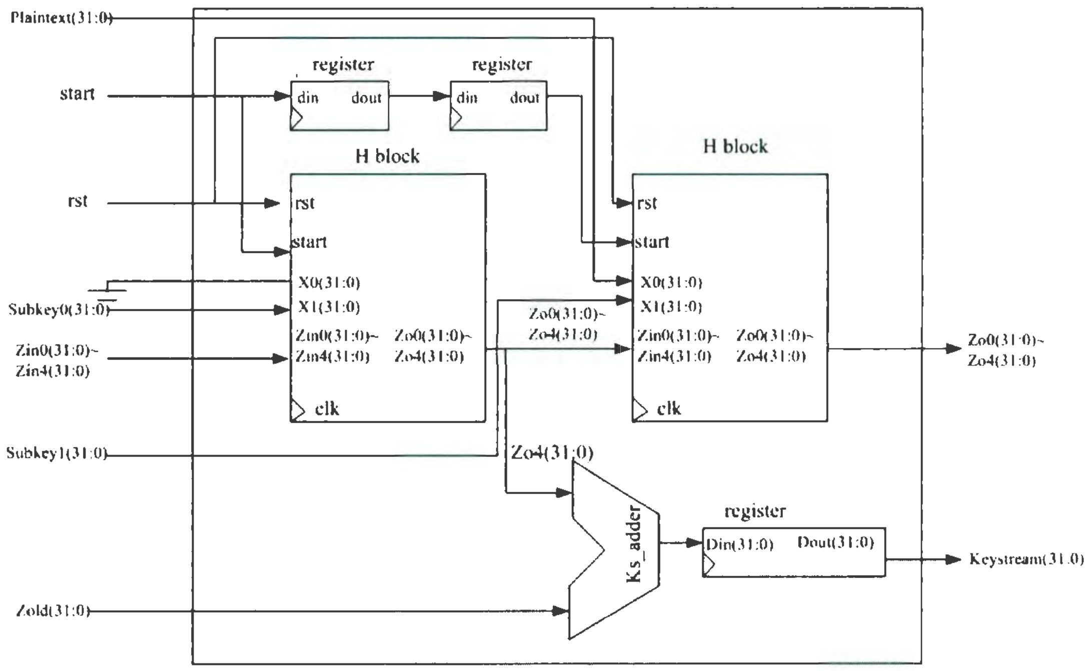


Figure 3.14 Block Diagram of Keystream Generator

3.3.5 The Controller

One tricky task to design a controller is to define the states in the system. If we define the states according to all control signals that the sub-components need, then any different combination of the signals could result in a new state, and the state transitions can cause a fair degree of "spaghetti- factor" when trying to follow the line of execution. Finite state machines are an adopted artificial intelligence technique, therefore, the representation of the states could be more abstract instead of catering to any trivial needs of sub-component, say, the selection signal of a MUX. With this in mind, we employed the idea that is often seen in framer cores used in communication systems. A typical example is in [54], which describe an implementation to perform basic word alignment and deframing for SONET/SDH system [23]. Usually, a frame has two indices, row and column. In hardware implementation, they are global signals and sent to every functional component, such as the error monitor. Those functional components perform the corresponding tasks depending on the current row and column indices. Since in an ideal synchronous circuit the behavior of the whole circuit can be predicted exactly, they can cooperate well. In our design, we change the row and column indices to block number. With a correct block number, at a specific time, the *key_mix* block knows whether it needs to generate the working keys; the subkey generator knows how to produce the subkeys; the core knows whether to send *keystream_valid* signal. The counter that generates the block number does not increment at every clock cycle but every four cycles as the encryption/decryption core requires four cycles to process one plaintext word.

This method is greatly simplified to make the design more readable and easier for maintenance. There is no explicit central controller. To some extent, it can be considered as an FSM decomposition scheme. For the modules in the system, the input condition is the block number, the decisions they could make are the state machine related behavior. However, it might consume more resources since the sub-components have their own decision making system and are more independent.

3.4 Synthesis Results of Phelix

To our knowledge, there are no published ASIC implementations results for the Phelix, but a rough estimation from the authors of [22], is that the cipher can achieve speed of at least 200 MBps with 20,000 gates for the area. The targeted technology is not specified.

Synthesis results of our design for Phelix are illustrated in Table 3.3. It consumes about 12,000 two-input NAND gates in the compact design and achieves more than one Gbps throughput in the high speed design.

Table 3.3 ASIC Implementation Results of Phelix

ASIC Device	Throughput (Mbps)		# of 2-input Nand gates	
0.18 μ CMOS	Compact Design	High Speed Design	Compact Design	High Speed Design
	260.0	1,440.0	12,400	64,200

3.5 Summary

In this chapter, we implement two structures for Phelix stream cipher: compact design and high speed design. Both of them are targeted for 0.18 micron CMOS technology. As we expected, the H function is a core part of the entire circuit, and it determines the encryption speed. In our compact design, we only use a single adder in this function block and divide the circuit into several layers of combinational logic separated by latches. The high speed design removes all sharing components and allocates them to the function blocks exclusively. Moreover, it has no separate controller, when the input *reset* is low and *start* is asserted, all functional blocks cooperate according to the global clock signal. After the initialization phase, when the first word of the plaintext has been processed, the core sends out a *keystream_valid* signal to notify the peripheral devices. Table 3.3 shows that the high speed design consumes about two times more of area than the compact one does, but it comes with a four times higher throughput. Sample VHDL code for the Phelix cipher implementation is contained in Appendix A.

The following chapter will investigate various hardware implementations for Salsa20.

Chapter 4 Hardware Implementation of the Salsa20 Stream Cipher

4.1 Introduction of Salsa20 Stream Cipher

Salsa20 stream cipher is another candidate of the eSTREAM project. It is claimed to provide high security, and is composed of several simple operations that are similar to Phelix. The core of Salsa20 is a hash function, encrypting a 512-bit block of plaintext by hashing the key (128-bit), nonce (64-bit), and a sequence number (64-bit) to a 512-bit output used as the keystream.

In hardware implementations, FPGA designs typically consume more resources and run slower than their ASIC counterparts. However, development in the features of FPGA, such as intellectual property (IP) integration and high-speed I/O interconnects, has allowed FPGAs to play an important role in digital designs. Therefore, we have designed a compact FPGA hardware implementation for Salsa20, whose requirement for memory is relatively larger compared with other candidates. To find out two extreme situations in compactness and speed, we also designed the corresponding ASIC structures for Salsa20. In addition, for comparison we implement a basic iterative design, which incorporates trade-offs between area and speed.

4.1.1 Algorithm

The main function in the Salsa20 core is called the quarterround function. Defining y as a 4-word sequence then quarterround(y) is a 4-word sequence. If $y = (y_0;y_1;y_2;y_3)$, then quarterround(y) = $(z_0;z_1;z_2;z_3)$, where y_i and z_i are 32-bit words, $i \in \{0,1,2,3\}$, and

$$z_1 = y_1 \oplus ((y_0 + y_3) <<< 7)$$
$$z_2 = y_2 \oplus ((z_1 + y_0) <<< 9)$$
$$z_3 = y_3 \oplus ((z_2 + z_1) <<< 13)$$
$$z_0 = y_0 \oplus ((z_3 + z_2) <<< 18)$$

The exclusive-or (XOR) of two words, denoted as " $\oplus$ ", is the sum of the words with carries suppressed. The symbol "<<<" represents left rotation by the indicated number of bits, and "+" represents addition modulo 2^{32}.

If we consider the 64-byte input block x = (x[0], x[1], ..., x[15]) as a 4×4 matrix of 32-bit words, the four elements in each row and each column will be modified by the quarterround function ten times, respectively. After that, the output is added with the original values, producing a 16-word keystream.

In short, the keystream generation process can be shown as the flow in Figure 4.1:

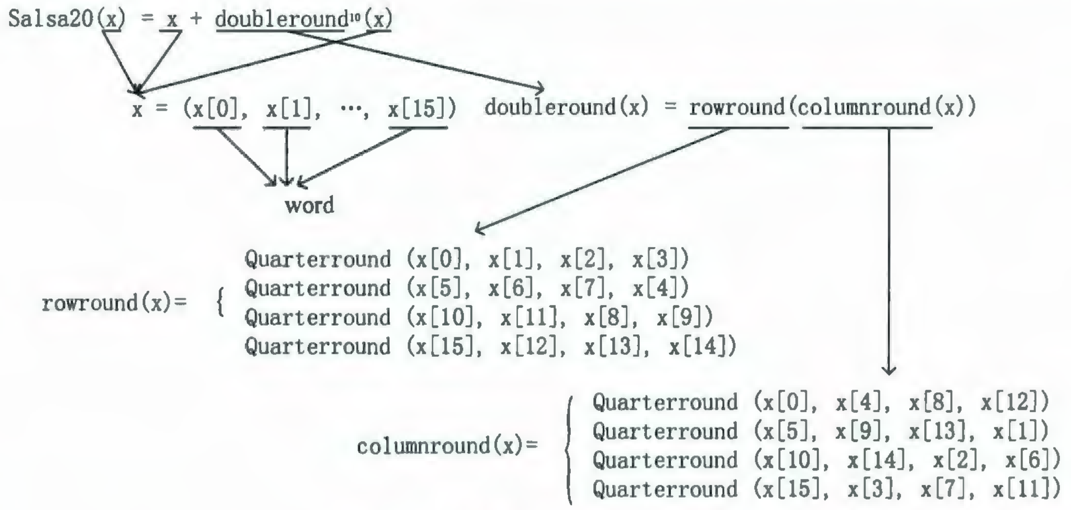

Salsa20(x) = x + doubleround¹⁰(x)
x = (x[0], x[1], ···, x[15])
doubleround(x) = rowround(columnround(x))
word
rowround(x)= Quarterround (x[0], x[1], x[2], x[3]) Quarterround (x[5], x[6], x[7], x[4]) Quarterround (x[10], x[11], x[8], x[9]) Quarterround (x[15], x[12], x[13], x[14])
columnround(x)= Quarterround (x[0], x[4], x[8], x[12]) Quarterround (x[5], x[9], x[13], x[1]) Quarterround (x[10], x[14], x[2], x[6]) Quarterround (x[15], x[3], x[7], x[11])

Figure 4.1 Salsa20 Keystream Generation

4.1.2 Security

In 2006, Crowley Paul Crowley reported a differential cryptanalysis of the 5-round Salsa20 model [58] and won Bernstein's US$1000 prize for "most interesting Salsa20 cryptanalysis". Later, Fisher et al. reported a 6-round Salsa20 model in [66]. The most recent cryptanalysis of Salsa20 is in [78], which found a significant bias in the differential probability for Salsa20's 4th round internal state, yielded by assigning single bit differences to the initial vector which may be freely chosen by an attacker. However, the attack in [78] was only targeted on Salsa20/r ($5 \leqslant r \leqslant 8$) instead of the full round Salsa20/20, where r is the number of rounds.

4.1.3 Previous Work on Hardware Implementation on Salsa20

Salsa20 was selected as one of the focus eSTREAM candidates for both Profile 1 (software) and Profile 2 (hardware) in Phase 2, and it received the highest voting score at the end of Phase 2 [66]. However, it was not selected for Profile 2 in Phase 2 since it was not considered to be suitable for very resource-constrained hardware implementations. This also coincides with our conclusion in Section 4.3 though we have not been aware of any other pure hardware implementation results except rough estimation.

The Salsa20 co-design was studied in an undergraduate course at Virginia Tech called "Introduction to Codesign" [60]. Through the reference C code, it can be found that in those co-designs, a component, which is a hardware accelerator is implemented targeted on Xilinx Spartan3e FPGA.

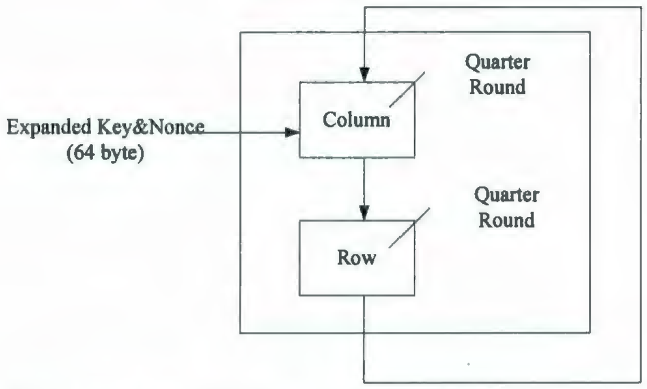


Figure 4.2 Block Diagram of Hardware Accelerator for Salsa20

It is clear that the co-design splits the double round function into two isolated blocks (column block and row block) to speed up the circuit. Table 4.1 shows that the speed up is limited. The reason is that during the execution, the output of the rowround function must be moved in and out for each iteration. It is a bottleneck for the overall performance.

Table 4.1 Results of Salsa20 Co-design [60]

Cipher	C lines	GEZEL lines	Area slices	Speedup
Salsa20	220	533	568	1.2

Of particular interest is the last column in the table that illustrates the resulting performance improvement after introducing the hardware accelerator to the pure software design. However, neither the throughput of the software design nor that of the co-design is available in the literature.

4.2 Analysis of Salsa20 Cipher Main Components

Considering the simple operations like addition modulo 2^{32}, exclusive or, and rotation by a fixed number of bits from the bottom level of the structure of Salsa20 circuit, the speed-and-area tradeoff in the ASIC design have been discussed in the previous chapter. However, we have not only looked into ASIC structures (fast structure, basic iterative structure and compact structure) for Salsa20, but also a compact structure in context of FPGA design.

Basically, FPGA is reconfigurable and more flexible compared with ASIC design. It has distinct properties such as rich sequential logic resources on-chip and fast carry chain in the configurable logic blocks. As a result, we present the performance of various adder structures and their effects on the cipher performance in this section. Also, the main second level components like quarterround block, memory block are discussed for both ASIC and FPGA implementation.

4.2.1 32-bit Adder

In the literature, there exist plenty of stream ciphers whose internal states or output keystreams are based on modular additions. This is partly due to the fact that, when implemented in software, they are able to produce outputs at a very high speed. However, adder structures are of great concern in hardware implementation, especially in those where addition is the most responsible factor to decide the overall speed.

As discussed in the previous chapter, a ripple carry adder (RCA) consumes least area compared with other adder structures in ASIC design. Therefore, we still use RCA for compact ASIC implementation for Salsa20. However, it is very important to consider the actual targeted FPGA device to achieve the highest levels of performance in both speed and area. For example, the number of rows and columns of configurable logic blocks (CLBs) can affect the partitioning of the implemented adders. Nowadays, most commercial FPGAs have provided specific arithmetic hardware resources. Figure 4.3 is a carry logic diagram integrated in Xilinx Virtex 2.5 V FPGA.

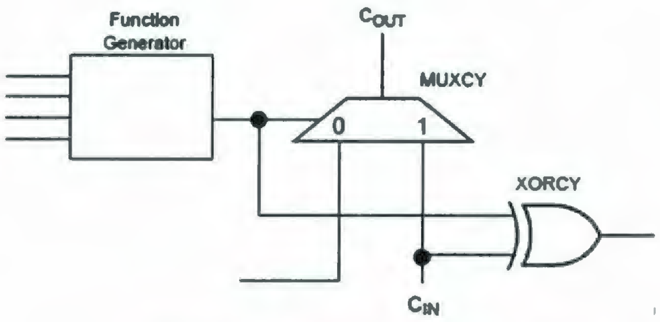


Figure 4.3 Carry Logic Diagram [74]

The carry chain contains a 2-input multiplexer (MUXCY) and an XOR (XORCY) gate. The function generator is a simple look up table (LUT). In a single CLB, there are two LUTs and each LUT can be implemented as a partial adder.

It is obvious that the predefined adder targeted in Virtex 2.5 V FPGA is a carry skip adder [74]. Table 4.2 shows the results of our implementations for various 32-bit adders in terms of speed and area. It can be seen from the table that in hardware designs, the growth in area consumption might not be linear to the increase of the speed.

Table 4.2 Comparison of Adder Implementations

on a Xilinx Virtex-IITM 2V250FG256 Board

Primitive Component	Critical Path Delay (ns)	Area (CLBs)
carry ripple adder	31.89	31
carry skip adder (using the dedicated carry logic)	6.87	16
carry look-ahead adder	23.68	82
CSA with add-one circuit	23.31	40
Sklansky parallel prefix adder	15.77	78
Brent-Kung parallel prefix adder	15.77	79
Kogge-Stone parallel prefix adder	7.72	110

All of the adder structures listed above are applicable to general purpose designs, though some of them might not look like to be. For example, in our FPGA implementation of carry ripple adder structure, the speed is more than four times slower than the fastest parallel structure. It looks very unlikely to be actually implemented in industry. However, a different target technology can result in a significant difference. In [84], the authors also give a comparison of various adder structures, such as carry ripple adder and carry look-ahead adder. Instead of FPGA, they chose the 0.7μm CMOS technology to implement all of the designs. Based on their implementation results, the carry ripple adder is not the slowest one. Besides, the carry skip adder has larger area requirements than others for all bit sizes.

The proposed adder implementations targeted on FPGA demonstrate the importance of taking advantage of the dedicated carry logic. In the FPGA, in which the dedicated resources are so much faster than the general purpose routing, it is very hard, impossible in many cases, to build an adder that is faster than the built in adder structure with the same area consumption. Moreover, without the use of the built-in carry chain logic, even if we implement a carry-skip adder of the same structure, we need two LUTs per full adder instead of one and speed degradation will appear too because of the irregular routing. Based on the results, we choose the Xilinx vendor predefined adder for the compact FPGA design of Salsa20.

When it comes to fast implementation, the parallel prefix adder is used. To illustrate parallel prefix adders, we should understand the prefix graph first. The basic idea is looking for cases in which carry out of a set of bits is the same as carry in. The carry produced at the ith stage is given as follows:

$$C_{i+1} = x_i y_i + (x_i \oplus y_i) C_i$$

where x_i and y_i are the inputs of the ith stage. The equation can be interpreted as stating that a carry is asserted if both operand bits are 1, and the carry from the previous stage is propagated if one of x_i and y_i is 0. Therefore, we have G_i and Pi denoting the generation and propagation at the ith stage:

$$C_i = G_{i-1} + P_{i-1} C_{i-1}$$

Define an associative binary operator $\triangle$ as:

$$y_0 = x_0$$
$$y_i = x_i \Delta y_{i-1}$$

We can get the most important property:

$$x_1 \Delta (x_2 \Delta (x_3 \Delta x_4)) = (x_1 \Delta x_2) \Delta (x_3 \Delta x_4)$$

Thus,

$$C_i = G_{i:0}\,; \qquad (G,P)^k_{i:j} = (G,P)^{k-1}_{i:q+1}\,\Delta\,(G,P)^{k-1}_{q:j}$$

where k is the level within the hierarchical structure and i, j define the range of the sub block. For example, in a 16-bit Ripple Carry Adder, which is represented graphically in Figure 4.5, The black circle is the specific carry operator $\triangle$. To graphically illustrate how to generate (G, P) pair by using $\triangle$, one of the black circles shown in Figure 4.5 is marked by ** and shown below:

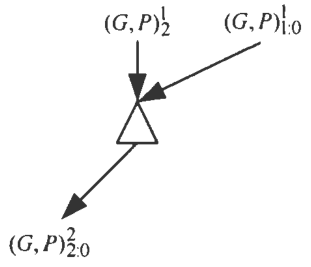


Figure 4.4 An Example of the carry operator $\triangle$

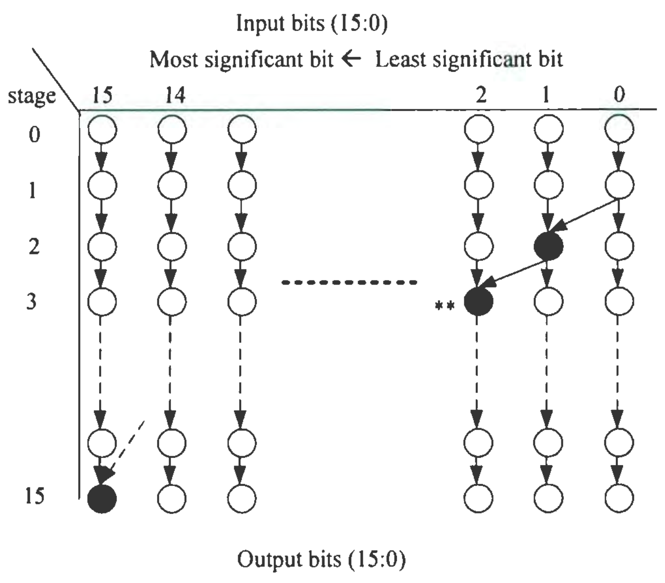


Figure 4.5 16-bit Ripple Carry Adder

The more black circle (operator $\triangle$) there are, the more area the adder structure requires. From our FPGA synthesis results, we find that the fastest parallel prefix adder is the Kogge-Stone adder, which has minimum logic depth (the number of stages shown in Figure 4.6) and full binary tree with minimum fan-out, resulting in a fast adder but with a large area.

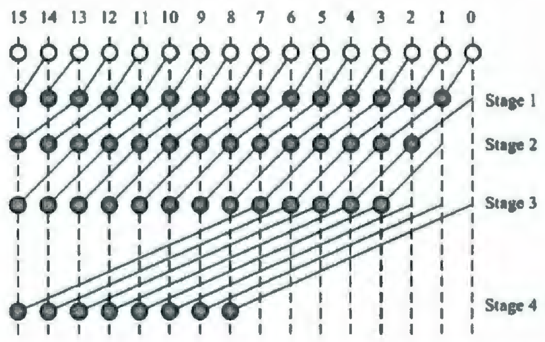


Figure 4.6 16-bit Kogge-Stone Parallel Prefix Adder

For the moderate basic iterative structure, more attention is paid to the tradeoff between speed and area. Therefore, carry select adder (CSA) is more suitable since it can be considered as a tradeoff between the adders shown above. There are many improvements focusing on increasing the efficiency of CSA [68-72]. In [41], the authors use the first zero finding circuit and MUX to reduce the area without speed penalty.

The carry select scheme divides the adder into three blocks as shown below. Sum and carry values are generated for possible carry values. Those values are fed into a multiplexer, who selects the correct sum and carry-out for the next stage.

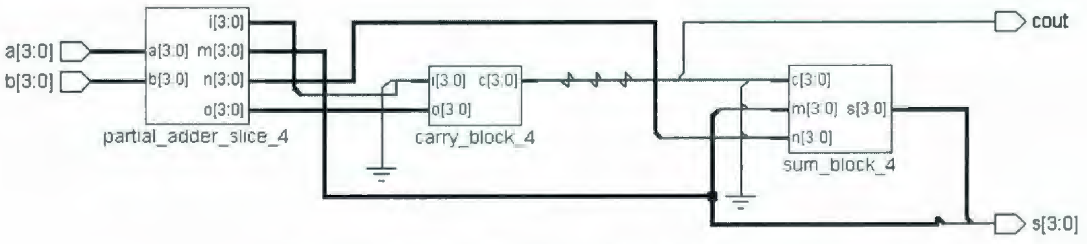
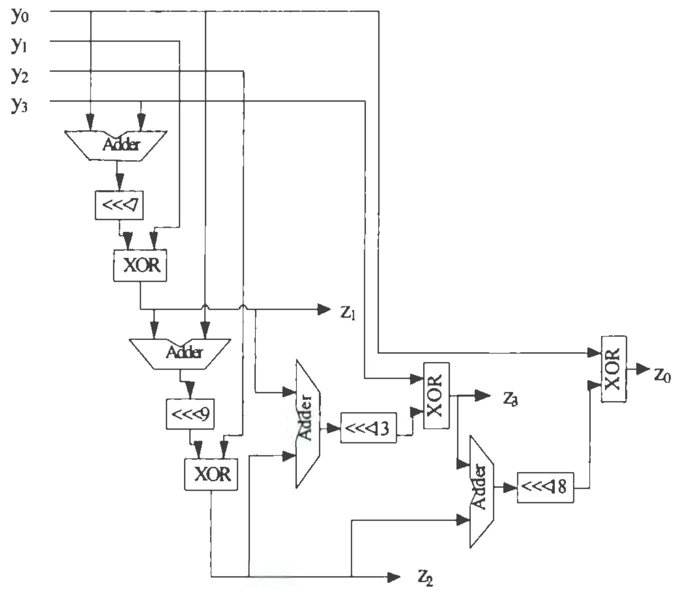


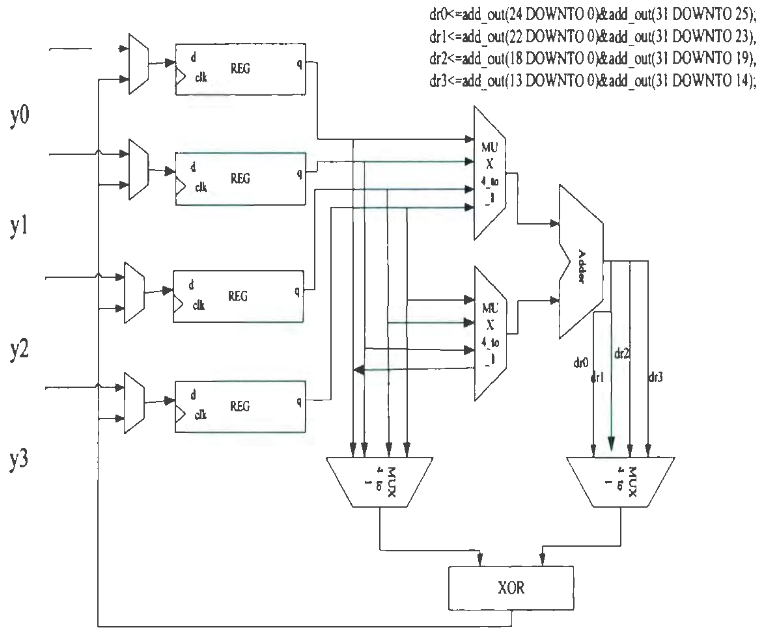
Figure 4.7 Block diagram of a 4-bit carry select adder

4.2.2 Quarterround Block

A single quarterround function regards its 16-byte input as an array of 4 words. We can choose between implementing the function using purely combinational design and using sequential logic with registers to divide the combinational circuit into small but faster sub-components. Both designs are illustrated in Figure 4.8.



(a) Combinational design



```
dr0<=add_out(24 DOWNTO 0)&add_out(31 DOWNTO 25);
dr1<=add_out(22 DOWNTO 0)&add_out(31 DOWNTO 23),
dr2<=add_out(18 DOWNTO 0)&add_out(31 DOWNTO 19),
dr3<=add_out(13 DOWNTO 0)&add_out(31 DOWNTO 14);
```

(b) Sequential design

Figure 4.8 Datapath of Quarterround Block

The combinational design shown above does not illustrate the main registers to store the final results (z_0, z_1, z_2, z_3). If those registers are counted, it takes 2,900 2-input NAND gates and the critical path cost 20.72 ns, while the sequential version takes 2,050 2-input

NAND gates and the critical path is 7.47 ns. Although the sequential design of quarterround function block works at a higher clock frequency, it spends 10 clock cycles to finish the four modifications for the input words at a time (Figure 4.8.b). In terms of those results, it is reasonable to choose the faster combinational quarterround function as the basic block for the fast ASIC structure of Salsa20 and the sequential version for the compact structure.

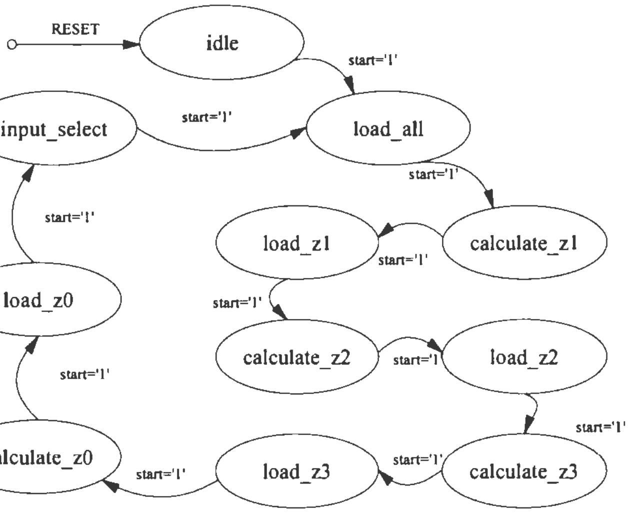


Figure 4.9 Finite State Machine of the Sequential Quarterround Block

The finite state machine in Figure 4.9 shows how the sequential quarterround block works. When the *start* signal is asserted, the controller transits from *idle* state to *load_all* state, in which the selection signals for the four MUXes in front of the registers are asserted (Figure 4.8.b). At the next positive edge of clock signal, the inputs selected by the MUXes have been held constant for specific period, called the setup time. Thus, the registers load all of them without metastability. In the quarterround block, there are only four registers requiring clock trigger. The states *load_*Z_i (i = 1, 2, 3, 4) perform data loading, as the name implies. In *calculate_*Z_i (i = 1, 2, 3, 4) states, the selection signals for all MUXes in the datapath are provided and kept stable during those clock cycles. Quarterround block works in a sequential manner as the calculation of Z_i (i = 1, 2, 3, 4) depends on the previous one.

4.2.3 Memory Block

Salsa20 encryption performs 320 invertible modifications for its 64-byte input data. The resulting words are added to the original ones to produce a 64-byte keystream.

Obviously, it requires memory to store the original input as well as the temporary data after each quarterround function.

Memory is usually a significant expense in most applications. Today's advanced FPGAs provide rich on-chip memories, which are maturely designed for compactness and speed. If properly employed, it can lead to a significant improvement in the latency of the overall design.

The Xilinx Virtex FPGA provides synchronous read/write block RAM in its primitive library. Each port of the block ram can be independently configured as a read/write port. In our FPGA design, four block rams are used. Each one is configured to 16-bit data width. The concatenation is shown in Figure 4.10:

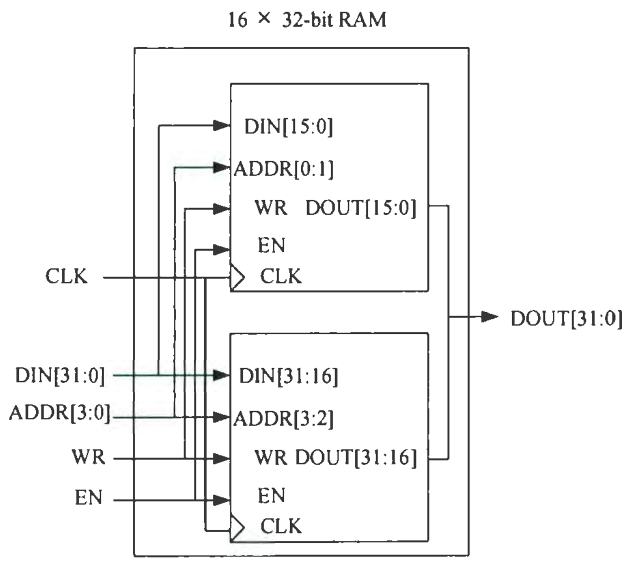
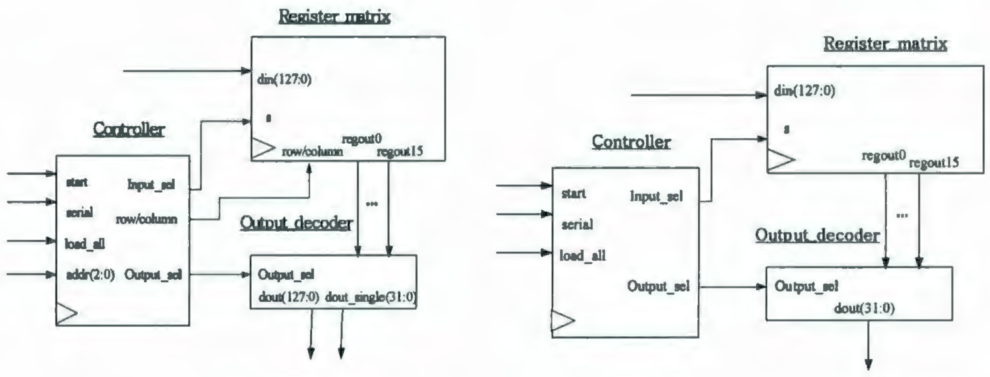


Figure 4.10 A 32-bit Wide RAM

The memories for Salsa20 ASIC compact structure are more complicated. Unlike FPGA design, we do not use the same structure for all memory blocks, since the control logic for the memory that only works as a buffer to store the original input without modifying during the keystream generation is much simpler than the other one, which will update the contents from the quarterround function block.



(a) Mem0 (b) Mem1

Figure 4.11 Block Diagram of Memory Blocks in ASIC implementation

The two types of memories are shown in Figure 4.11, named Mem0 and Mem1, respectively. The capacity for both memories is 16 × 32-bit. But the controller of Mem0 has a row/column select signal to tell the register matrix which data array to update. This is decided by the property of quarterround function: row modification and column modification are interleaved. The 32-bit width output ports are only used at the end of ten double rounds. Each 32-bit sum is a word of keystream.

The synthesis result in Table 4.3 illustrates the assumed difference. Mem0 consumes more area and runs slower because of the more complicated control logic. Mem0 stores the 16 original 32-bit words, and its contents will be modified after each quarterround function. Mem1 stores the 16 original 32-bit words, which will not be touched until the quarterround function block has been used for twenty times.

Table 4.3 Synthesis Results for Different Memory Blocks

COMPONENTS	AREA (# of NAND gates)	CRITICAL PATH	CRITICAL PATH PASSING TIME (ns)
Mem0 (modify)	5,820	FSM-output_sel-dout_dingle	2.22
Mem1 (no_modify)	4,662	FSM-output_sel- dout	2.09

4.2.4 Control Unit based on Various Datapath

4.2.4.1 Controller of the Compact ASIC Structure

In a traditional controller consisting of a Finite State Machine (FSM) and combinational output logic, a large number of states can dramatically impact the logic equations, number of gates, and clock rate. A state consists of the condition (inputs trigger), state transition and outputs (control signals). There are two main methods of FSM design, focusing on handling where to generate the outputs. They are Moore Machine, whose outputs are associated with the current state of the device, and Mealy Machine, whose outputs are associated with not only the current state but also the input signals.

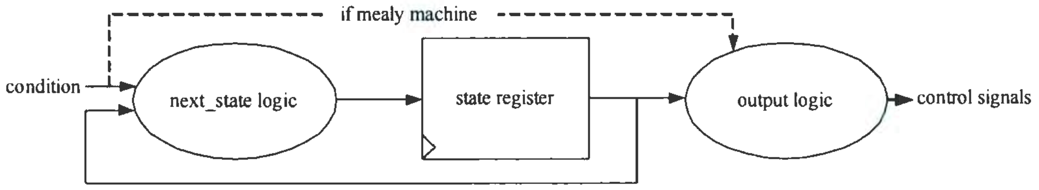


Figure 4.12 Block Diagram of FSM

Due to their simple structure and predictability, FSMs are easy to implement. An FSM can be implemented by different encoding methods. In our design, we used the most popular encoding scheme called one-hot encoding. One-hot encoding requires a D flip flop for each state, while another encoding method called binary encoding only requires $\lceil \log_2 N \rceil$ D flip flops for N states. It seems like that the latter is more efficient. However, decoding an encoded binary FSM leads to increased logic levels between states, while in a one-hot encoded FSM, the states are already in the decoded format and, consequently, only simple combinational logic is needed as a part of FSM.

The design of controllers significantly depends on the structure of datapath. In compact ASIC design of Salsa20, the datapath consists of two 32-bit $\times 16$ memory blocks and one quarterround function block.

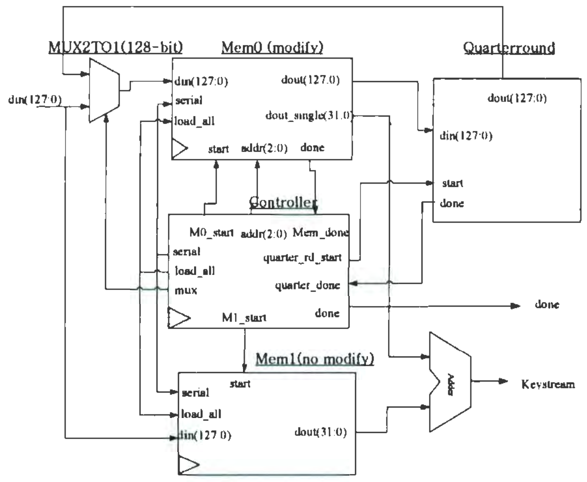


Figure 4.13 Datapath of Compact ASIC Structure for Salsa20

The specifications of the main blocks are given below:

- Quarterround: performs quarterround function.
- Mem0(modify): stores the 16 original 32-bit words, and its contents will be modified after each quarterround function.
- Mem1(no modify): stores the 16 original 32-bit words, which will not be touched until the quarterround function block has been used for twenty times.

Based on the datapath of the figure above, it is easy to define five states that output different control signals.

In *idle* state, nothing happens; arbitrary control signals for the datapath are sent to the datapath. The controller is transited to *load_rows* state if the *start* signal is asserted. At this state, one row (128 bits) of the 512-bit data block is loaded into the registers in parallel. The quarterround block starts to work at *quarter_en* state. The output of the quarterround block is fed back to the memory at *load*_z state. The controller goes to *add* state if twenty rounds have been accomplished. After *add* state, in which the modified data is added with the original one to produce the keystream, a new process for the next input data block can begin.

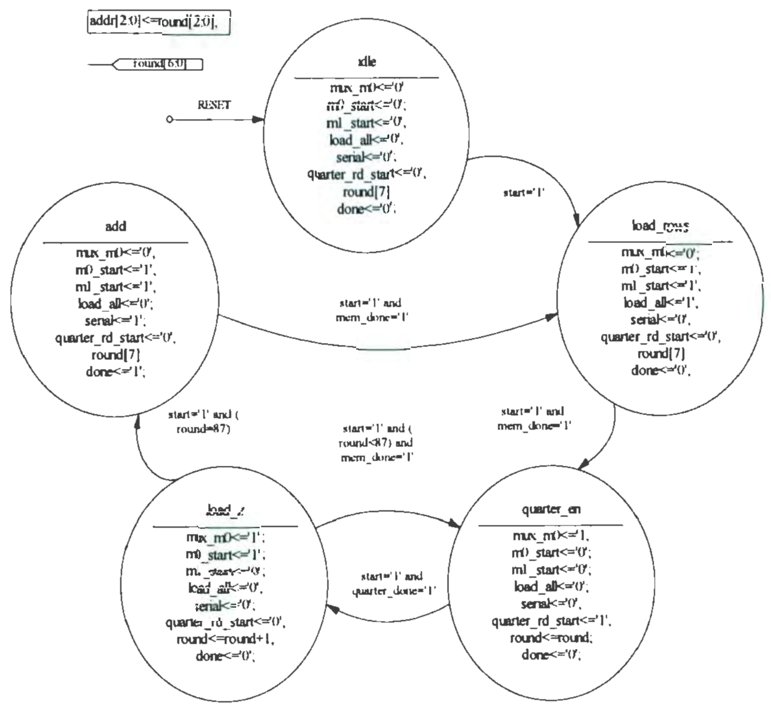

addr[2:0]<=round[2:0];
round[6:0]
RESET
idle
start='1'
load_rows
add
start='1' and mem_done='1'
start='1' and (round=87)
start='1' and (round<87) and mem_done='1'
start='1' and mem_done='1'
load_z
quarter_en
start='1' and quarter_done='1'

Figure 4.14 FSM of Compact ASIC Structure for Salsa20

4.2.4.2 Controller of the Basic Iterative and High Speed ASIC Structure

The datapath of an iterative structure consists of four quarterround function blocks, since the four rows or the four columns are encrypted independently. The control unit is simply a combination of a counter and a comparator.

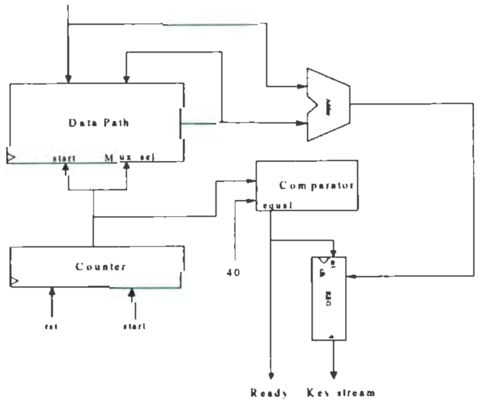


Figure 4.15 Salsa20 Basic Iterative ASIC Structure

After the *start* signal is asserted, the quarterround function begins to work on the 4×4 data matrix during the odd clock cycles. The data matrix performs a transpose function during every even clock cycle. It takes 40 clock cycles to finish the whole data modification (encryption).

The controller of the high speed ASIC structure is exactly the same as the basic iterative structure as the former can be seen as a concatenation of the latter.

4.2.4.3 Controller of the Compact FPGA Structure

Since our compact design implemented in FPGA employs two 32×16 bits predefined memory and reuses the adder for compactness, the number of states required to produce different control signals for the datapath is largely increased. It is composed of ten row functions, ten column functions, and an addition for one data block, and each row or column function requires four states for a row or column (since each modification for an element in a row or column needs different control signals).

To decrease the number of states and simplify the controller structure, the proposed design uses a partial microprogramming controller. A complete microprogramming controller [14] consists of address generator, address register, ROM, instruction register as the figure below:

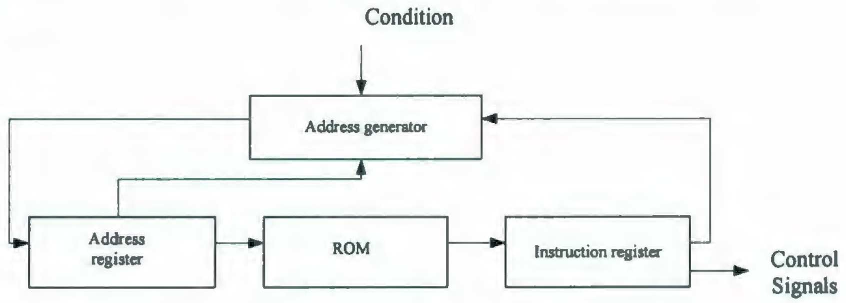


Figure 4.16 General Structure of Microprogramming Approach

In our design, we keep FSM, whose output is not the control signals but the states. A 6-bit counter is used to generate the address. The control information is loaded into the memory in the initialization phase, and the controller sends out a microinstruction to the datapath every clock cycle. Here, the microinstruction for Salsa20 are the read/write enable signal, memory address, selection signals for the MUXs, and a ready signal to notify other functional blocks to receive the keystream. It is a flexible method, especially for further improvement, because program changes only cause slight difference in the memory, which contains control information. The controller also eases the pressure on the designer when the design needs flexibility.

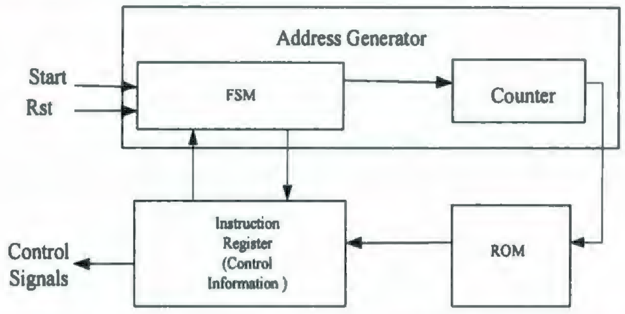


Figure 4.17 Controller of Salsa20 (FPGA implementation)

4.3 The Synthesis Results of Salsa20

All designs are simulated by using Modelsim and synthesized by using LeonardoSpectrum evaluation edition. Synthesis results for Salsa20 are illustrated in Table 4.4. Table 4.5 shows a comparison between ciphers from the previous work done in [47]; the device that was employed was Xilinx FPGA 2V250fg256 [75], which is used by us too.

Table 4.4 Implementation Results of Salsa20

Structure	Compact (ASIC)	Basic Iterative (ASIC)	Fast (ASIC)	Compact (FPGA)
Device	0.18μ CMOS	0.18μ CMOS	0.18μ CMOS	Xilinx FPGA 2V250fg256 [71]
Throughput	71.2 Mbps	255 Mbps	4.8 Gbps	38 Mbps
Area	14,100 2-input Nand gates	23,408 2-input Nand gates	470,000 2-input Nand gates	194 CLB slices + 4 Block RAMs
Throughput/ Area	5.05 Mbps/kgates	10.90 Mbps/kgates	10.21 Mbps/kgates	0.20 Mbps/slices

Table 4.5 Cipher performance and area comparison [47]

Cipher	Area (slices)	Frequency (MHz)	Throughput (Mb/s)	Throughput / Area
A5/1	32	188.3	188.3	5.88
W7	608	96.0	768.0	1.26
E0	895	189.0	189.0	0.21
Helix	418	32.0	1024.0	2.45
RC4	140	60.8	120.8	0.86
Salsa20*	194	96.0	38.0	0.20

The implementation result of Salsa20* in Table 4.5 is obtained from our design. It is unsurprising that Salsa20 based on a compact context is slower and consumes more area than other popular stream ciphers, since it performs a large number of invertible modifications, each of which changes one word of the matrix in a sequential manner. However, the high speed ASIC design achieves a more acceptable throughput to area ratio. The best throughput to area ratio comes from the basic iterative design as it has considered the trade-off between speed and area.

4.4 Summary

In this chapter, we implement Salsa20 in various structures. Both ASIC and FPGA implementations are considered. The proposed VLSI implementations achieve a data throughput up to 4.8Gbps targeted for 0.18 micron CMOS technology, and a compact FPGA design uses 194 slices and 4 memory blocks in a Xilinx device. The FPGA results are compared with other stream ciphers, considering two major quality metrics: area and speed.

Salsa20 has now been removed from the focus list in terms of hardware performance. However, the document of eSTREAM project shows that they only compare the selected candidates for extremely resource constrained hardware environments. It is apparent that Salsa20 can be implemented for very high speeds in digital hardware. Moreover, it is hard to predict the influence of the FPGA design tools and the HDL design entry method on the synthesis results. Considering that Salsa20 has received the highest weighted voting score of any Profile 1 algorithm at the end of Phase 2, we believe that it is reasonable to provide more comparisons for this cipher in terms of hardware performance.

Chapter 5 Statistical Test of Salsa20 and Phelix

5.1 Introduction

Statistical testing is one of the topics in the theory of statistics, which is based on the idea that a sample from the set of all possible outcomes of the experiment will be 'typical' to analyze the whole set. In this chapter, we will focus on one particular field of statistical testing: randomness testing. Such testing is important because common cryptosystems require key/keystream to be generated in a random fashion. We have employed a test suite provided by the National Institute of Standards and Technology (NIST) [8] specifically for the tests of cryptographic sequences. NIST is a non-regulatory US agency, whose mission is to promote measurement science, standards, and technology. It provides an empirical statistical test suite discussing particular standards of testing random and pseudorandom number generators. In this test suite, sixteen recommended statistical tests are provided namely; frequency, block frequency, runs, longest run of ones in a block, matrix rank, discrete Fourier transform, non-overlapping template matchings, overlapping template matchings, universal test, Lempel-Ziv compression, linear complexity, serial, cumulative sums, approximate entropy, random excursions, random excursions and variants. We have chosen four of them to apply to the sample sequences generated by Salsa20 and Phelix. The corresponding algorithms and the test results will be discussed later.

Considering that the tests from NIST are not particularly designed for testing the security of stream ciphers, in other words, they do not consider the internal states, input key, or IV of the ciphers; we have divided our study into two phases. In the first phase, keystreams are generated from each cipher and tested by using the selected tests from the NIST test suite. In the second phase, several structural tests that consider the properties of stream ciphers are applied to both Phelix and Salsa20.

5.2 General Discussion

In this section, we shall see how to utilize probability distributions to carry out a statistical test. Also, some basic concepts in statistical testing will be introduced.

5.2.1 Randomness

For a real random generator, the properties of the output sequences have to meet some standards. A variety of statistical test algorithms can be applied to a keystream to evaluate whether it meets those randomness requirements. A random bit sequence could be interpreted as that the probability of '0' or '1' happens in the sequence should be exactly 0.5. Furthermore, the bits are independent. Randomness is a probabilistic property. In other words, it is the property of a random sequence can be characterized and described in terms of probability. There is no "complete" set of tests for judging whether a sequence is random or not, because each test only assesses the presence or absence of a particular "pattern", sometimes called test statistic, which, if detected, would indicate that the sequence is random based on a certain hypothesis. To understand test statistic, we must look into the concepts of null and alternative hypothesis first. A null hypothesis, which is denoted as H_0 supposes that the sequence being tested is random. Any other hypothesis is called the alternative hypothesis. For each applied test, a decision or conclusion is derived that accepts or rejects the null hypothesis.

5.2.2 Test Statistic and P-value

Having null and alternative hypotheses, the next step is to find out a statistic which shows up any departure from H_0. There is not only one type of statistic that can be chosen. For example, some tests, such as T test [18] chooses a statistic noted as T,

$$T = \frac{X_n - \mu}{S_n/\sqrt{n}}.$$

where $\overline{X}_n$ is the sample mean, μ is the expected mean, *n* is the size of the sample and

$$S_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}\left(X_i - \overline{X}_n\right)^2,$$

which is the sample variance.

Some tests choose different statistics. In chi-square goodness-of-fit test [18], the test statistic is often written as

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

where O_i is an observed frequency; E_i is an expected (theoretical) frequency, asserted by the null hypothesis.

The computed test statistic should be compared with a critical value determined by a theoretical reference distribution of this statistic under the null hypothesis. If the test statistic value exceeds the critical value, the null hypothesis for randomness is rejected. Otherwise, the null hypothesis (the randomness hypothesis) is not rejected (i.e., the hypothesis is accepted). A result which is unlikely to occur if H_0 is true is called a significant result. Consequently, the significance level of a test is the maximum probability, assuming the null hypothesis, that the statistic would be observed. The P-value is the probability that the null hypothesis will be rejected in error when it is true. It can be described as P (reject H_0 | H_0 is true) mathematically. In practice, a significance level α is set prior to a test. If P-value $\geq \alpha$, then the null hypothesis is accepted. If P-value $< \alpha$, then the alternative hypothesis is accepted. The parameter α is typically chosen in the range [0.001, 0.01]. In the NIST test suite [8], α has been set to 0.01. It indicates that one would expect 1 sequence in 100 sequences to be rejected by the test if the sequence is random.

An example including the related concepts is given below:

Assume there is a random sample of size *n*, taken from a normal distribution with an unknown mean μ. The observed mean is x. Then what we are interested in is whether x differs significantly from a particular value μ_0 for μ. Thus the null hypothesis is given by H_0: $\mu = \mu_0$. If the standard deviation is known, and the value is σ, then we have the test statistic:

$$z = \frac{x - \mu_0}{\sigma / \sqrt{n}}$$

If H_0 is true, then z is a standard normal variable. The observed value of z is denoted as z_0. The probability P ($|z| \geq z_0$) is represented by the shaded area in Figure 5.1. It is the P-value. Given an α of value 0.01, we can get a value c from P ($|z| > c) = \alpha$. If $|z_0|$ is greater than $|c|$, then H_0 will be rejected.

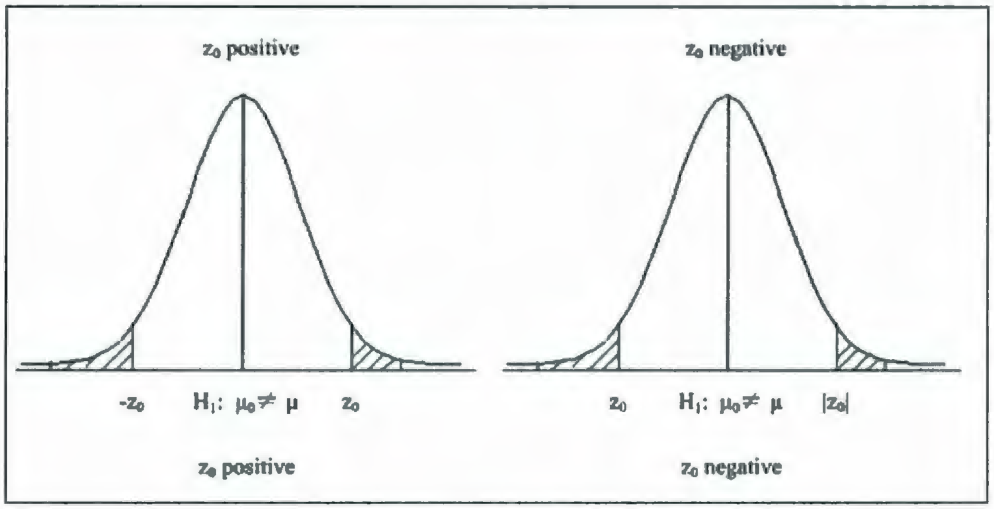


Figure 5.1 An Example of Statistical Test [18]

5.3 Test Model

Figure 5.2 shows the model for statistical test. The sample generator could be Salsa20 or Phelix keystream generator or other derived generator, because the data sequences we have tested are not only the keystream, but also some new sequences that would represent certain properties or correlations between key, IV, internal states and a part of keystream. A large amount of sequences produced by the generator are used as the input of the selected test algorithm, which will calculate the test static and the corresponding P-value. Based on the P-value, we can decide whether the sequences have passed the test. The decision does not indicate weak keys or internal states recovery, but can be used for distinguishing the sequence being tested from a truly random one.

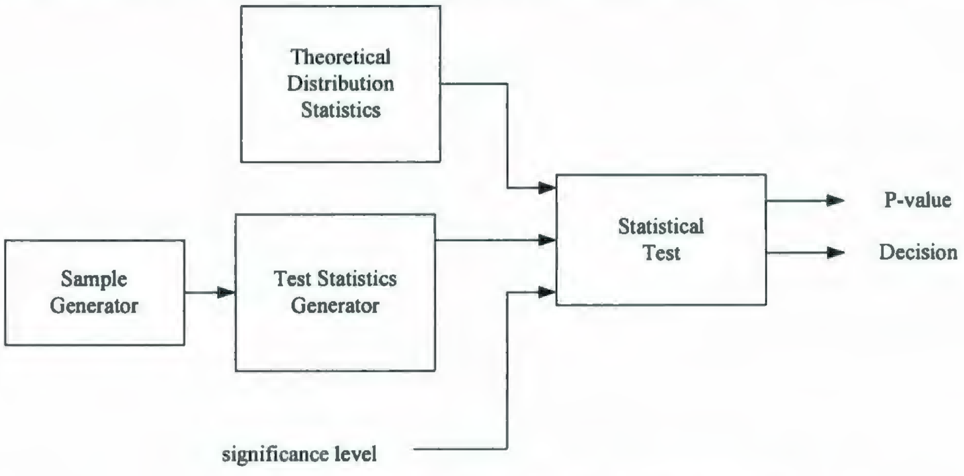


Figure 5.2 Model for Statistical Hypothesis Test

5.4 Keystream Tests

In this section, randomness properties of the output keystream are examined. We generate a large amount of keystream and apply certain statistical tests. Details of the test algorithms are introduced in each sub-section.

5.4.1 Frequency (Monobit) Test

The frequency test is one of the most basic tests, which focuses on the proportion of zeros and ones in a sample sequence. The number of ones and zeroes are expected to n/2; n is the length of the input sequence. The test algorithm is described below:

```
Input sequence s;
for i ← 0 to n-1 do    //n is the length of s
   if s[i]= 0 then x[i] := -1;
   if s[i]= 1 then x[i] := +1;
$S_n = \sum_{i=0}^{n-1} x[i]$;
$S_{obs} := |S_n|/\sqrt{n}$ ; //Compute the test statistic.
P-value := erfc($S_{obs}/\sqrt{2}$ ); //erfc is the complementary error function
```

Figure 5.3 Frequency Test Algorithm

This test makes use of De Moivre-Laplace theorem, which is defined as the fact that for a sufficiently large number, say n, of independent Bernoulli trials [42], the distribution of S_{obs} (the binomial sum) is approximately a standard normal distribution.

According to the Central Limit Theorem,

$$\lim_{n\to\infty} P(S_{obs} \le z) = \phi(z) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} e^{-u^2/2} du$$

where $\varphi(z)$ is the cumulative probability function of the standard normal distribution. The complementary error function of z is defined as

$$erfc(z) = \frac{1}{\sqrt{2\pi}} \int_{z}^{\infty} e^{-u^2} du$$

Then the corresponding P-value equals erfc($S_{obs}/\sqrt{2}$), since the P-value is $2[1-\phi(|S_{obs}|)]$.

5.4.2 Frequency Test Within a Block

The focus of this test is similar to the frequency test, but it divides the whole sequence into M sub-blocks. The frequency test (monobit) can be seen as a special case, where M=1, of block frequency test. The test algorithm is described below:

```
Input sequence s;
// Partition s into M non-overlapping blocks, n is the length of the sequence.
// N is the length of the sub-block.
M=$\lfloor n/N \rfloor$;
for i ← 0 to M-1 do    //n is the length of s
    p[i] = $\frac{\sum_{j=0}^{N-1} s[j]}{N}$; // calculate the proportion of ones in each N-bit block
$\chi^2_{(obs)} = 4N\sum_{i=0}^{M-1}(p[i]-0.5)^2$;
P-value = igamc (M/2, $\chi^2_{(obs)}/2$); // igamc is the incomplete gamma function
```

Figure 5.4 Block Frequency Test Algorithm

It is recommended that the length of each sequence to be tested should be at least 100. The block size N is selected such that $N \geq 20$, $N > 0.01n$ and $M < 100$. In our test, n=512, N=20 and M=25. If n is not a multiple of the selected N, M is the floor of (n/N), the largest integer that is not greater than (n/N).

The function noted as igamc( ) in Figure 5.4 is called incomplete Gamma function, which is derived from Gamma function. They are based on an approximation formula [8].

$$\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}dt \tag{5.1}$$

$$igamc(a,x) = 1 - \frac{1}{\Gamma(a)}\int_0^x e^{-t}t^{a-1}dt \;=\; \frac{1}{\Gamma(a)}\int_x^\infty e^{-t}t^{a-1}dt \tag{5.2}$$

The formula (5.1) is Gamma function and (5.2) is incomplete Gamma function. More details can be found in the Handbook of Applied Mathematical Functions [46]. The statistic $\chi^2_{(obs)}$ is referred to chi-square distribution with M degrees of freedom.

5.4.3 Discrete Fourier Transform Test

The discrete Fourier transform (DFT) is often used for Fourier analysis of finite-domain discrete-time signals. It is widely employed to analyze the frequencies in a sampled signal. The DFT statistical test focuses on the peak heights in the discrete Fourier transform obtained from the sample sequence.

If the sequence is random, then the proportion of peaks that exceed the 95% threshold should not be significantly different than 5%.

```
Input sequence s;
for i ← 0 to n-1 do    //n is the length of s
   if s[i]= 0 then x[i] := -1;
   if s[i]= 1 then x[i] := +1;
S = DFT(x); // A sequence of complex variables is produced.
// S' is the first n/2 elements in S. Since the complex conjugate of s[i] equals s[n-i],
// the other half of S is ignored.
M = modulus(S');
$T = \sqrt{3n}$ ; the 95 % peak height threshold value
$N_0$ = 0.95n/2; the expected theoretical (95 %) number of peaks that are less than T.
N1 = the observed number of peaks that are less than T
$d = \frac{N_1 - N_0}{\sqrt{n \times 0.95 \times 0.05 / 2}}$ ; test statistic
P-value = erfc ( |d|/2 ); //erfc is the complementary error function
```

Figure 5.5 Discrete Fourier Transform Test Algorithm

According to the central limit theorem, the distribution of the value of the test statistic d can be considered as standard normal distribution N (0, 1) when n is large enough.

Though the report [8] claims that the default threshold value of the DFT test, which is $\sqrt{3n}$, is not correct, the correct value of the variance of the test statistic remains unsolved. Therefore, we still use the default value in our tests.

5.4.4 Runs Test (Wald-Wolfowitz Test)

In this test, a run is defined as an uninterrupted sequence of identical bits. For instance, "111100011100" is divided in four runs, two of which consist of "1" and the others of "0". If there are too many runs, the data is likely to alternate in a nonrandom order.

Runs test can be used to test the randomness of a sequence, by measuring whether the number of runs of various lengths is as expected.

```
Input sequence s;
$\pi$:= (number of ones in s)/n; //n is the sequence length
$\tau$:= $2/\sqrt{n}$ ;
if ($|\pi - 1/2| > \tau$) then break; // the test should not be run under this circumstance
                                       //because of a failure to pass the Frequency test.
else
    $V_n(obs) := \sum_{k=1}^{n-1} r(k) + 1$; // where r(k)=0 if s(k)=s(k+1), and r(k)=1 otherwise.
P-value := $\mathrm{erfc}\left(\frac{|V_n(obs) - 2n\pi(1-\pi)|}{2\sqrt{2n}\pi(1-\pi)}\right)$ // erfc is the complementary error function
```

Figure 5.6 Runs Test Algorithm

Considering an oscillation as a change from a one to zero or vice versa, a large value for $V_n(obs)$ would indicate an oscillation between substrings is too fast; a small value would indicate that the oscillation is too slow.

5.4.5 Experimental Results and Analysis

NIST has adopted two approaches to interpret the test results: the examination of the proportion of sequences that pass a statistical test and the uniform distribution of P-values [8].

5.4.5.1 Proportion of Sequences Passing a Test

A P-values pass proportion is directly related to α, the significance level. For example, given an α = 0.01, if 1000 sequences are tested, and 998 of them have P-values ≥ 0.01, then the pass proportion is 0.998. The low and high bounds can be computed using the formula:

$$1-\alpha \pm 3\sqrt{\frac{\alpha(1-\alpha)}{m}}$$, where m is the number of tested sequences.

If the proportion falls in this interval, H_0 is accepted.

For m = 1024, and α = 0.01, the lower bound of the P-values pass proportion is 0.980672, and the higher bound is 0.999328. The results for each test applied to 1024 keystream sequences are given in Table 5.1.

Table 5.1 the Result of Proportion of Sequences Passing a Test

Cipher	Test			
Salsa20	Frequency	Block Frequency	Discrete Fourier Transform	Runs
	Proportion	Proportion	Proportion	Proportion
	0.972656	0.992187	0.985351	0.994328
Phelix	Frequency	Block Frequency	discrete Fourier transform	Longest Run
	Proportion	Proportion	Proportion	Proportion
	0.982359	0.993342	0.980963	0.998799

There is one P-value corresponding to the application of a statistical test on a single sequence, and in our tests, 1024 (2^{10}) keystream sequences of length 512 (2^9) bits are generated from both Salsa20 and Phelix respectively. Therefore, there are 1024 P-values for each test. Table 5.1 shows that only the proportion of Salsa20's keystream sequences passing the frequency test is out of the acceptable interval.

5.4.5.2 Uniform Distribution of P-values

To examine the uniformity of the distribution of a set of P-values, we have to learn about the chi-square goodness-of-fit test (also know as the χ^2 goodness-of-fit test). It is one of the most widely used statistical tests when one wants to see if the observed frequencies of multiple mutually exclusive categories are significantly different from those which could be expected as random.

Suppose that $y_1, y_2, \ldots, y_k$ and $p_1, p_2, \ldots, p_k$ are the observed and expected frequencies for the ones that have appeared in the k different n-bit keystreams. Then, the chi-square statistic for the test is:

$$\chi^2 = \sum_{1 \le s \le k} \frac{(y_s - np_s)^2}{np_s}, \quad \sum_{s=1}^{k} p_s = 1$$

If the observed value of χ^2 is χ^2_0, then the P-value equals the probability $P(\chi^2 \ge \chi^2\_0)$. More details of the χ^2 goodness-of-fit test could be found in [18].

One of the common misunderstandings about P-value is that it is the probability that the null hypothesis is true. In fact, it does not attach probabilities to hypotheses, but the probability that a perfect random number generator would have produced a sequence less random than the sequence that is tested. In those tests that we have chosen from the NIST suite, the computed test statistics could be anywhere on the x-coordinate of a standard distribution. The distribution of P-values is supposed to have the property of uniformity. Appendix G gives an example of how those P-values distribute. The idea of evaluating the uniformity of the obtained P-values is to divide the interval between 0 and 1 into 10 equal sub-intervals, and the P-values that lie within each sub-interval are counted and tested via an application of the χ^2 goodness-of-fit test. In this way, the P-values' P-value is obtained. The test statistic is

$$\chi^2 = \sum_{i=0}^{9} \frac{(F_i - 1/10)^2}{1/10}$$

where F_i is the frequency of P-values in sub-interval i, and m is the number of the tested sequences. A new P-value is calculated by the function *igamc* $(9/2, \chi^2/2)$. Not like the previous P-values, the new P-value tends to be close to one as $(F_i - 1/10)$ is very small based on the test results. For a good random sequence, the new P-value should be close to 1. Table 5.2 illustrates all test statistics we have observed and their corresponding P-values. All new P-values obtained are very close to 1. The values shown in the table are not the accurate ones, but the rounded numbers, as they are enough to illustrate the uniform distribution.

Table 5.2 The Result of Uniform Distribution of P-values

Cipher	Test							
Salsa20	Frequency		Block Frequency		Discrete Fourier Transform		Runs	
	χ^2	New P-value	χ^2	New P-value	χ^2	New P-value	χ^2	New P-value
	0.14427	0.99998	0.12797	0.99999	0.02583	0.99999	0.13556	0.99999
Phelix	Frequency		Block Frequency		discrete Fourier transform		Longest Run	
	χ^2	New P-value	χ^2	New P-value	χ^2	New P-value	χ^2	New P-value
	0.10589	0.99999	0.01785	0.99999	0.00509	0.99999	0.14339	0.99999

5.5 Correlation Tests

The tests from NIST are not particularly designed for testing the security of stream ciphers. They simply focus on testing the randomness look of the keystream sequence. In other words, they do not consider the cipher structure including the internal states, input key, and IV of the ciphers. The authors of [50] applied four structural randomness tests to the stream ciphers of eSTREAM, including Phelix and Salsa20. They are *Key/Keystream Correlation Test* that considers the correlation between key and the corresponding keystream using a fixed IV, *IV/Keystream Correlation Test* that considers the correlation between IV and the corresponding keystream using a fixed key, *Frame Correlation Test* that considers the correlation between keystreams using different IV values, and *Diffusion Test* that examines the diffusion property of each bit of key and IV. They also introduced another two tests, which are not applied but left as a future study: *Internal State Correlation test* and *Internal State/Keystream Correlation Test*. Each of the six tests produces new sequences of data to which randomness tests can be applied. In [50], the randomness of the sequences was tested by considering the appropriateness of the fit to the binomial distribution, using the χ^2 goodness of fit test.

In this section, we apply the four randomness test algorithms introduced in 5.4 to the new sequences to extend the result. Only the chi-square goodness-of-fit test is employed in [50] for evaluation. Additionally, we apply the last two statistical tests of [50], which were not investigated for Phelix and Salsa20.

5.5.1 Testing New Sequences

5.5.1.1 Sequence Generation Process

In this section, we describe the process to generate the sequences to be tested for the first few tests mentioned in [50]. To evaluate various correlations for stream ciphers, new sample sequences representing particular properties of the stream cipher are required. According to [50], the first two types of sequences are obtained by the operation

$$\text{keystream}[0 \text{ to } i-1] \oplus s[0 \text{ to } i-1]$$

where keystream[0 … i-1] are the first i bits of the keystream, and s[0 … i-1] represents i-bit key for *key/keystream correlation sequence tests* and i-bit IV for *IV/keystream correlation sequence tests*, respectively.

The algorithm to generate the frame correlation sequence is shown below:

```
Randomly choose key, IV;
for i ← 1 to N    // N=1024
do
    // L=512, encryption could be Salsa20 or Phelix keystream generator
    Ks[i] = first L bit of encryption (IV, key);
    Increment IV;
for i ← 1 to L
do
   for j ← 1 to N
   do
       sequence[i] = sequence[i] & ks[j][i]; // & is concatenation
return sequence
```

Figure 5.7 Frame Correlation Sequence Generation Algorithm

In Figure 5.7, L is an arbitrary length of the keystream and N is the number of repetitions to produce a keystream with fixed 256-bit key and incremented values of IV. L sequences are initialized to empty before operation. The sequence sequence[i] indicates the ith bit of the sequence, ks[j][i] indicates the ith bit of the jth keystream segment (j ranges from 1 to N and i ranges from 1 to L). It is illustrated in Figure 5.8.

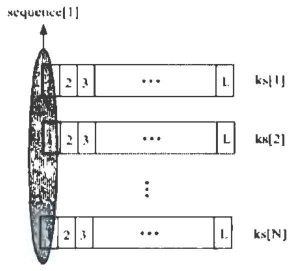


Figure 5.8 Diagram of Frame Correlation Sequence

The last type of sequence generated aims at the diffusion property of each bit of key and IV. That is, each bit of IV and key should affect the keystream with equal probability.

```
M is a (K+V)×L zero matrix;   // k is the length of the key; V is the length of the IV
M= $\begin{bmatrix} M[1] \\ M[2] \\ \vdots \\ M[K+V] \end{bmatrix}$
for i ← 1 to N            //N=1024
  do
    Randomly choose key and IV
    Keystream[i] := encryption(IV, key);
    for j ← 1 to K
      do
        key' := change one bit of the original key;
        d[j] := Keystream[i] ⊕ encryption (IV, key);
        M[j] := M[j] + d[j];
    for j ← 1 to V
      do
        IV' := change one bit of the original IV;
        d[j] := Keystream[i] ⊕ encryption (IV', key);
        M[K+j] := M[K+j] + d[j];
Return M;
```

Figure 5.9 Diffusion Sequence Generation Algorithm

5.5.1.2 Experimental Results and Analysis

The evaluation strategies used for new sequences are the same as that used in previous section 5.3: the proportion of sequences passing a test and the distribution of P-values which should be uniform between 0 and 1. The results are summarized in Table 5.3 and 5.4.

Table 5.3 Proportion of Sequences Passing a Test

acceptable range [0.980672, 0.999328]

Tests		Frequency	Block Frequency	Discrete Fourier Transform	Runs
Cipher	Sequences	proportion	proportion	proportion	proportion
Salsa20	S1	0.996094	0.997070	0.984375	0.991210
	S2	0.985352	0.991211	$\boxed{0.979492}$	0.988281
	S3	$\boxed{0.923828}$	$\boxed{0.953125}$	$\boxed{0.113281}$	$\boxed{0.863281}$
	S4	0.989583	0.992188	0.981771	0.989583
Phelix	S1	0.979492	0.993164	0.985352	0.989258
	S2	0.992188	0.993164	0.990234	0.988281
	S3	$\boxed{0.890623}$	$\boxed{0.949219}$	$\boxed{0.123782}$	$\boxed{0.890625}$
	S4	0.994792	0.989583	$\boxed{0.973958}$	0.984375

- S1: Key/Keystream Correlation Sequence (Number of Sequences = 1024, Sequence Length = 256)
- S2: IV/Keystream Correlation Sequence (Number of Sequences = 1024, Sequence Length = 128)
- S3: Frame Correlation Sequence (Number of Sequences = 512, Sequence Length = 1024)
- S4: Diffusion Sequence (Number of Sequences = 384, Sequence Length = 512, Repeat 1024 times)

As the results in Table 5.3 indicate, we obtained some deviations from the acceptable range of the proportions that pass the tests, and those deviation values are circled in the table. The Frame Correlation Sequence of both Salsa20 and Phelix fails all the four tests and is particularly poor for the DFT test. This may indicate that Salsa20 and Phelix do not satisfy the necessary property as the frames generated using consecutive IVs are correlated. Whether this can be exploited particularly in a cryptanalysis is not clear and merits further study, particularly in relation to the DFT test. For further investigation, we have provided the source code that is related to frame correlation sequence generation and testing process in the Appendix F.

Table 5.4 The Result of Uniform Distribution of P-values

Tests		Frequency		Block Frequency		discrete Fourier transform		Runs	
Cipher	Sequence	χ^2	New P-value	χ^2	New P-value	χ^2	New P-value	χ^2	New P-value
Salsa20	S1	0.07397	1.00000	0.01177	1.00000	0.16953	1.00000	0.012363	1.00000
	S2	0.16499	1.00000	0.01313	1.00000	1.28315	0.99846	0.04609	1.00000
	S3	0.43272	0.99998	0.13541	1.00000	8.61479	0.47356	0.58081	0.99994
	S4	0.03461	1.00000	0.02770	1.00000	0.21745	1.00000	0.03706	1.00000
Phelix	S1	0.04198	1.00000	0.01442	1.00000	0.14113	1.00000	0.01063	1.00000
	S2	0.15280	1.00000	0.00397	1.00000	1.34406	0.99815	0.01900	1.00000
	S3	0.49475	0.99997	0.12518	1.00000	8.76799	0.45896	0.43684	0.99998
	S4	0.03339	1.00000	0.05225	1.00000	0.26139	1.00000	0.04072	1.00000

- S1: Key/Keystream Correlation Sequence (Number of Sequences = 1024, Sequence Length = 256)
- S2: IV/Keystream Correlation Sequence (Number of Sequences = 1024, Sequence Length = 128)
- S3: Frame Correlation Sequence (Number of Sequences = 512, Sequence Length = 1024)
- S4: Diffusion Sequence (Number of Sequences = 384, Sequence Length = 512, Repeat 1024 times)

The same situation has happened in the proportion results: deviation is obtained only when the discrete Fourier transform test is applied to the frame correlation sequences. However, it is hard to conclude that whether there are structural weakness in Salsa20 and Phelix or in the test itself, since the frame correlation sequences from both Salsa20 and Phelix have passed the frequency test, the block frequency test and the runs test, while the other three different sequences have passed the discrete Fourier transform test as well.

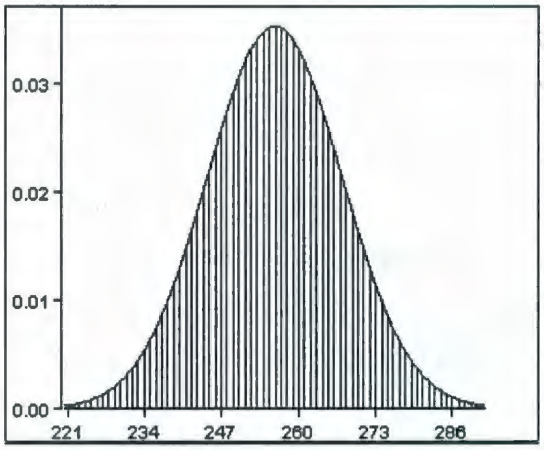
5.5.2 Keystream/Internal States Correlation Test

The purpose of the test in this section is to evaluate the correlation between the internal states and the first k bits of the keystream [50]. The main idea is that at any time if the internal states have a distinguishing property such as low/high weight, the resulted keystream should behave randomly in terms of its weight. Both Salsa20 and Phelix support various key and IV lengths. We used key size of 256 bits and IV size of 128 bits. For further analysis, other key and IV sizes may be considered.

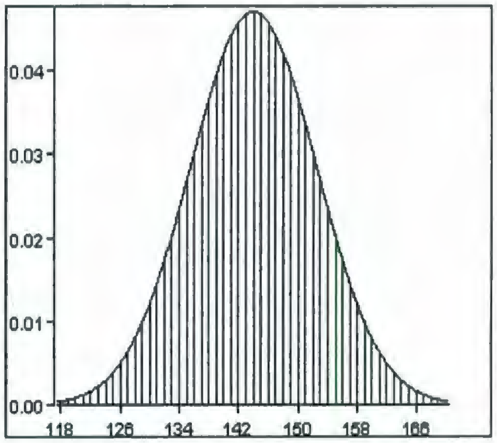
5.5.2.1 Algorithm

In this test, k represents the size of an internal state memory and m k-bit keystreams are generated from m randomly generated keys and a fixed IV. During the encryption, each state (a k-bit data block) is stored to XOR with the keystream. The obtained results are calculated for their weights, and the m weights are grouped into categories. Then, the observed frequencies are calculated and the chi-square goodness-of-fit test is applied to calculate P-values.

Figure 5.10 shows the theoretical distributions for k=512 (the length of Salsa20's internal state) and for k=288 (the length of Phelix's internal state) based on binomial distribution.



a. k=512, p=0.5



b. k=288, p=0.5

Figure 5.10 Theoretical Distribution

The category limits for Phelix are chosen as {0 – 118, 119 – 126, 127 – 134, 135 – 142, 143 – 150, 151 – 158, 159 – 166, 167 - 287}. In general, if the random variable follows the binomial distribution with size k and variance p, we write Binomial (k, p). If the encryption algorithm is secure, the distribution of the weights is Binomial (k, 0.5). Cumulative probabilities are calculated depending on the categories. They are observed frequencies, which will be compared to the expected ones in the chi-square test.

```
Fix IV;
Initialize weights w to zero;
for i ← 1 to m
  do
    randomly generate a key;
    keystream := encryption(IV, key);
    for j ← 1 to n
      do
        internal_state[i][j] := encryption_round(current_state);
        // i is the index for i^th keystream; j is the index for j^th state
        k_xor_s[i][j] := internal_state[i][j] XOR keystream;
        w[i][j] := number of ones in k_xor_s;
        // group the weights into the selected categories
        for i ← 1 to C    // C is the number of categories
          if w[i][j] is in category[k]
            W[k] := W[k] + 1;
    Chi-square of Goodness-of-fit test on W;
```

Figure 5.11 keystream/Internal States Correlation Test Algorithm

In the algorithm above, the function encryption round is defined as a double round function [20] for Salsa20 or two half functions [22] for Phelix.

5.5.2.2 Experimental Results and Analysis

Table 5.5 shows the test result for Salsa20. 2^{15} keystreams are generated according to the different input keys of 256-bit length. Each is related to ten internal states. One internal state consists of 512 bits. IV is the concatenation of the input nonce and the sequence number. In our test, it is chosen as {57, -121, 9, -62, -105, 60, 1, -10} & {0, 0, 0, 0, 0, 0, 0, 0}, where & indicates concatenation and each number separated by comma is a byte. The values of the test statistic and the P-values are shown below.

Table 5.5 Test Results for Salsa20

Internal State Number	χ^2	P-value
1	0.008030	0.928598
2	0.008700	0.925686
3	0.008364	0.927129
4	0.005190	0.942567
5	0.008252	0.927618
6	0.008017	0.928654
7	0.007110	0.932803
8	0.006595	0.935277
9	0.006992	0.933361
10	19.16823	1.196899E-5

It can be seen from the above that the P-value for the correlation of the last internal state and the keystream is far less than the significance level, which is 0.01 in the NIST test suite. It is not surprising when we look back into the process of keystream generation.

The four constants in [20] are C_0, C_1, C_2, and C_3, where C_0 = (101, 120, 112, 97), C_1 = (110, 100, 32, 51), C_2 = (50, 45, 98, 121), and C_3 = (116, 101, 32, 107). If each of k_0, k_1 and IV is a16-byte sequence, then

$X = (C_0, k_0, C_1, IV, C_2, k_1, C_3)$;

keystream = Salsa20(X)

= X + (final internal state)

Therefore, the correlation between the keystream and the final internal state significantly depends on X. If IV starts from 0 or other small number, a segment of continuous zeros will appear in X, resulting a small P-value. If IV is changed to {-84, 38, -7, -99, 112, 73, -116, 102, -50, -75, 18, 10, 84, -55, 93, -15}, then the last χ^2 value equals 0.17045 and the corresponding P-value is 0.679709.

Table 5.6 shows the test result for Phelix. 2^{15} keystreams are generated according to the different input keys of 256-bit length. There are nine internal states for each keystream generation process. One internal state consists of 288 bits.

Table 5.6 Test Results for Phelix

Internal State Number	χ^2	P-value
1	0.013279	0.908200
2	0.013104	0.908864
3	0.011896	0.913147
4	0.013117	0.908816
5	0.014341	0.904678
6	0.013360	0.907982
7	0.011528	0.914497
8	0.012151	0.912226
9	0.013619	0.907097

Phelix has 9 states for a keystream: 5 "active" states and 4 "old" states that are only used in the keystream output function. Therefore, keystreams of 288 (32×9) bits are generated using a fixed randomly chosen IV for each key.

According to the test result, we did not obtain deviations from the expected values as all of them are much higher than the significance level 0.01.

5.5.3 Internal States Correlation Test

5.5.3.1 Algorithm

The purpose of this test is to evaluate the correlation between internal states generated from similar IVs. Firstly, key and nonce (for Salsa20, nonce is a part of IV, and for Phelix, nonce equals IV) are chosen randomly and at each round the internal state is stored. Each state is of n bits. With incremented values of nonce this procedure is repeated M-1 times. Therefore, a matrix of size M × n is obtained at the end; each element in this matrix is an internal state vector. The column weights (number of ones in each column) of the matrix are calculated. Then, the chi-square goodness-of-fit tests are applied. The process of the sequence generation is very similar to frame correlation sequence generation.

Taking Salsa20 as an example, it has 10 internal states and each is 512 bits. Starting with an IV of value 0 and repeating the keystream generation by incrementing IV by one at each time for (2^{10}-1) times, we obtain the matrix shown in Figure 5.12. The column weights are calculated for the chi-square goodness-of-fit test.

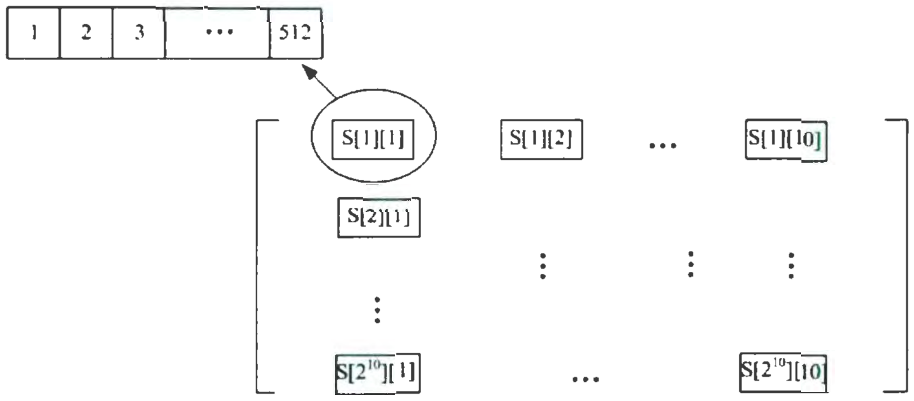


Figure 5.12 Salsa20 Internal States Matrix

```
Fix IV;
Nonce = 0;
// for Salsa20, nonce is a part of IV; for Phelix, nonce equals IV
Initialize weights w to zero;
for i ← 1 to m // m = 1024
    if i > 1
        IV = IV + 1;
    for j ← 1 to n // n is the number of the states
        do
            internal_state[i][j] := encryption_round(current_state);
            // i is the index for i^th keystream; j is the index for j^th state
for j ← 1 to n
  for k ← 1 to L // L is the size of one internal state
    for i ← 1 to m
        if internal_state[i][j][k] = 1
            w[L× (j-1) + k] := w[L× (j-1) + k] + 1;
            // group the weights into the selected categories
            Chi-square of Goodness-of-fit test on w;
```

Figure 5.13 Internal States Correlation Test Algorithm

5.5.3.2 Experimental Results and Analysis

For the internal states correlation test, m = 2^{10} keystream are generated but ignored. Only the internal states are stored and form a matrix. Salsa20 has ten internal states while Phelix has nine. Table 5.7 and 5.8 shows the test results.

Table 5.7 Test Results for Salsa20

Internal State Number	χ^2	P-value
1	0.008037	0.9286
2	0.008920	0.9248
3	0.008359	0.9272
4	0.008193	0.9279
5	0.007252	0.9321
6	0.008030	0.9286
7	0.006932	0.9336
8	0.006893	0.9338
9	0.006542	0.9355
10	0.006992	0.9334

Table 5.8 Test Results for Phelix

Internal State Number	χ^2	P-value
1	0.015103	0.9022
2	0.013360	0.9080
3	0.012151	0.9122
4	0.013340	0.9080
5	0.013465	0.9076
6	0.010523	0.9183
7	0.012350	0.9115
8	0.013514	0.9109
9	0.012380	0.9114

According to the test results, no significant weaknesses are found for either Salsa20 or Phelix as all of them are much higher than the significance level 0.01.

5.6 Conclusions

In this study, we use four of the empirical statistical tests given in the NIST suite to analyze the randomness of the Salsa20 and Phelix keystreams. Since the test suite did not consider the relationship between key, IV, internal state and the keystream, we also applied two novel tests, which are proposed in [50] to examine the ciphers. In addition, we provide the results of uniform distribution of P-values as well as the proportion of sequences passing a test.

The experimental results shows that both Salsa20 and Phelix have passed the tests in NIST, considering that P value less than 0.01 indicate a possible weakness. Significant deviation is observed in the correlation test for the last internal state (the state after 9 double rounds) and the keystream in Salsa20, but this is easily explained when considering the cipher structure and it remains to be seen whether this can be exploited on cryptanalysis. Also, both Salsa20 and Phelix failed the frame correlation test for DFT test and this may indicate that Salsa20 and Phelix does not satisfy the necessary properties as the frames generated using consecutive IVs are correlated. However, how this could be exploited in an attack is an open question.

Chapter 6 Conclusions and Future Work

6.1 Summary of Research

In this thesis, we deal with hardware implementations for two stream ciphers proposed for the eSTREAM project: Salsa20 and Phelix. Both of them are claimed to be suitable for software and hardware implementation since they are built on a series of simple operations: 32-bit addition, bitwise addition (XOR) and rotation operations. No S-box is needed.

Generally speaking, Salsa20 is a hash function in counter mode. The hash function performs 320 invertible word modifications. Each modification is an XOR operation on two 32-bit words, or a rotated sum of two 32-bit words. The resulting data block is added to the original input as the last step to accomplish one keystream generation process for 512 bits. Each data block representing the internal state is of 512 bits. In our design, it is a 4×4 register or RAM matrix, with 32-bit size for each entry. There is no correlation between any two 512-bit keystreams.

Considering that the needs of different applications in communication systems demand different structures for cryptographic algorithms in hardware, we investigated four different structures for Salsa20: compact ASIC structure, basic iterative ASIC structure, fast ASIC structure and a compact FPGA structure. An ASIC implementation is typically faster and more compact compared with an FPGA implementation. However, FPGA provides flexibility with its configurable logic. In terms of choosing one targeted device for hardware implementation of a protocol or algorithm or a whole System-on-Chip (SoC), many factors should be considered, such as time-to-market, chip area, time to working silicon, cost considerations, etc. In our study, we choose Xilinx Vertex FPGA for the proposed Salsa20 compact structure because Salsa20 requires memory to store the original input data as well as the temporary data after each quarterround function, while FPGAs provide rich on-chip memories, which are designed for compactness and speed. If properly employed, it can lead to a significant improvement in the latency of the overall design. A micro-programming controller is used to replace the traditional finite state machine in the controller. This method translates physical design into programming. Further improvement can be done by simply adding or modifying the microinstructions. Here, the microinstruction for Salsa20 is of fixed-length binary digits, which include the read/write enable signal, memory address, selection signals for the MUXes, and a ready signal to notify other functional blocks to receive the keystream. They can be mapped to higher programming language easily. However, since it is a small design, it is unnecessary to do the mapping specification.

The other three structures are designed and synthesized by using 0.18 micron CMOS technology. Several schemes are used for speed increase or area decrease. For example, in Salsa20 fast ASIC structure, full pipeline structure has been implemented and each stage out of twenty consists of one round of Salsa20's encryption process. Each stage is exactly the same as the Salsa20 basic iterative structure, and controlled by a single central controller. In the basic iterative structure, parallelism is applied: four quarterround function blocks in the datapath operate on the four rows or columns of the data matrix simultaneously and independently. A common reused module in the designs is the adder, more details could be found in the section 4.2.2, Figure 4.8. The throughput of Salsa20 ranges from 38 Mbps implemented in FPGA requiring 194 CLB slices to 4.8 Gbps implemented in ASIC requiring the equivalence of 468,160 2-input NAND gates.

The synthesis results illustrate that the throughput to area ratio for Salsa20 compact FPGA structure is only 0.20. It is an unsurprising small value compared with other popular stream cipher implementation results. Recalling the original encryption algorithm and the compact FPGA design of Salsa20, it is obvious that it performs a large number of invertible modifications, each of which changes one word of the matrix in a sequential manner. This is the main reason for the lower throughput of the compact structure.

Phelix is a high-speed stream cipher with built-in message authentication code (MAC) functionality. The same as Salsa20, it is composed of a series of simple operations: addition modulo 2^{32}, exclusive or, and rotation by a fixed number of bits. There are 5 words that are updated during each round, and 4 "old" words are stored in memory to be used in the keystream output function. One block that produces one word of keystream consists of two "half-block" functions H.

The H function block is very critical in terms of performance as it is used in four phases. Since the proposed compact implementation for Phelix aims at compactness, we use a single 32-bit carry ripple adder in the H block and divide the circuit into several layers of combinational logic separated by latches or registers. The high speed implementation removes the features that facilitate compactness with the sacrifice of throughput. Also, it employs the fastest adder we have evaluated.

The other components included in the top level hierarchy are key mixing block, nonce-expanding block, subkey generator and a buffer to store four old states. Vendor supported libraries are employed since it is an efficient way to do the implementation by using the primitives. The throughput of the compact ASIC design for Phelix is 260 Mbps targeted for 0.18 micron CMOS technology, and the corresponding area is equivalent to 12,366 2-input NAND gates. The high speed implementation achieves a throughput of 1,440.0 Mbps with an equivalent area of 64,200 2-input NAND gates.

To sum up, the proposed designs can be divided by two methods; the first one is based on the different technologies (FPGA design and ASIC design) and the second one is based on the implementation schemes, either for compactness or for high speed. The table below is the comparison of FPGA design's advantages and ASIC design's advantages. One's advantage implies the other's corresponding shortcoming.

Table 6.1 Comparison of FPGA Design and ASIC Design

FPGA Design Advantages	ASIC Design Advantages
● No NRE (non recurring expenses)	● Less power consumption
● Use of embedded processors, which is very difficult to implement in an ASIC	● Less area requirement ● Lower per-unit cost
● Flexibility in the design process (reprogramability)	● Higher internal clock frequency

The advantages and disadvantages of the different schemes are listed below:

- Basic iterative structure

Advantages: The required hardware resources are decremented as only one round of the cipher. Considered as a basic stage, basic iterative structure is easy to be expanded.

Disadvantages: This architecture has higher register-to-register delay because of the MUX added in front of the input registers and so we have to either split one clock cycle into 2 or decrease the frequency.

- High speed structure based on full-pipelining

Advantages: This structure has the highest throughput.

Disadvantages: The overall latency of a pipelined structure is slightly lower than in a non-pipelined equivalent. This is due to the fact that extra registers must be added to the data path of a pipelined structure. However, in our design, the result of each stage, which is a basic iterative structure has already been registered. Therefore, there are no extra registers added.

- Compact structure based on module reuse

Advantages: It decreases the area required, especially when the reused module is a significant cost in terms of area consumption in the whole design.

Disadvantages: extra multiplexers and other control logics have to be implemented, and this could decrease the clock frequency.

- Compact structure based on the employment of FPGA's generic module (e.g. RAM)

Advantages: The rich on-FPGA block RAMs can store more than several lookup table's worth of memory elements and they are typically have higher densities and faster access times compared with lookup-table-configured memories. Besides, their timing characteristics are more predictable.

Disadvantages: In FPGAs with the block RAMs of very big size, it could be a waste to configure a block RAM for a pure 64 bytes data block.

It is hard to conclude the advantages and disadvantages for the high speed structure based on faster major functional module (e.g. parallel adder) in a system. But problems may occur due to the module itself. For example, high speed carry look-ahead adders may have fan-in and fan-out difficulties, which indicate that the design could have difficulties to drive a large number of inputs or outputs. For more details of fan-in/fan-out, refer to [85].

Although stream ciphers are inspired by the one-time pad (OTP) theory, which was proved to be perfectly secure, stream ciphers make deviation from OTP, since they do not meet the requirement that the key stream is at least the same length as the plaintext, and generated completely at random. The keystreams from a stream cipher algorithm are produced by deterministic generators. Thus, an attacker may be able to recover a part of secret key or internal state by using the improved distinguishing attack.

In our study, we use various statistical tests given in the NIST suite to analyze the randomness of the Salsa20 and Phelix stream ciphers. They are the frequency test, the frequency test within a block, the discrete Fourier transform test and the runs test. It is assumed that the keystreams should be indistinguishable from a random sequence and secure enough to keep the key and the internal states secret.

Since the test suite did not consider the relationship between key, IV, internal state and the keystream, we applied two novel tests presented in [50]: keystream/internal states correlation test and internal states correlation test to examine the ciphers. We also generate four new types of sequence to look into more possible correlations. They are key/keystream correlation sequence, IV/keystream correlation sequence, frame correlation sequence and diffusion sequence.

The experimental results show that both Salsa20 and Phelix keystreams have passed the tests in NIST. For each experiment, the significance level was fixed at 0.01, which implies that, ideally, no more than one binary sequence should be rejected for each sample of 100 binary sequences evaluated by a statistical test. Deviation is observed in the correlation test for the last internal state (the state after 9 double rounds) and the keystream in Salsa20. It is explained in Section 5.4.2. In Section 5.5.1, the frame correlation sequences from either Salsa20 or Phelix do not pass the discrete Fourier transform test. It is not because of the sample size as we have applied multiple lengths ranging from 2^{10} to 2^{20} for this single test. More study is needed in this case to explain the phenomenon.

6.2 Future Work

Hardware implementation and analysis for new stream ciphers are interesting research areas and there are many aspects that should be taken into account. Basically, there are two most important factors to consider: (1) hardware complexity, in FPGA designs, measured by the number of configurable slices, and in ASIC designs, measured by the number of standard cell NAND gates; (2) speed or throughput. Although in the eSTREAM project, the hardware implementations are dedicated for the low end environment, which indicates that the compactness is the most important consideration, more studies on the trade-offs between area and speed of the hardware implementation for a chosen cipher is undoubtedly necessary since the ultimate goal is wide spread adoption.

Another potential future work is the cipher co-design based on FPGAs. As we know, FPGAs provide a very flexible platform for cipher implementations. The high speed I/O interface, embedded IP cores, block RAMs, and especially the configurable microprocessor like Xilinx's MicroBlaze form a suitable environment for hardware/software co-designs of cryptographic systems. Unlike sheer software implementation or hardware implementation, co-design is a relatively new area. Since traditional design methodologies are not adequate to address the co-design challenges, people often fall into pitfalls when they try to gain the advantages of both while neglecting the communication between them and a proper mapping process to divide the cryptographic system into software part and hardware part. To implement a cipher efficiently by using co-design methodology, the bottlenecks of both software and hardware should be taken into account. Moreover, additional analysis on programming techniques may be helpful for the overall performance.

The randomness of pseudorandom sequences could be evaluated by using statistical tests. The NIST suite provides a relatively comprehensive collection of generic tests. We only used four of them. It typically spans more properties that a good cryptographic algorithm should satisfy by applying the whole suite. These properties include any detectable correlation between plaintext/ciphertext pairs, any detectable bias due to single bit changes to either a plaintext or a 128-bit key, in addition to many others. The NIST suite has ignored the internal structures of stream ciphers. Statistical tests taking the internal structure, key or IV loading phases into account are very limited compared with sole keystream tests. The correlations between key, IV, internal state and keystream are important factors in the design of a stream cipher since availability of the keystream and IV should not leak any information about the internal state or secret key. More attention could be paid in this area.

Reference

[1] 44 U.S.C § 3542 (b)(1) (2006), available at http://www.law.cornell.edu/uscode/html/uscode44/usc_sec_44_00003542----000-.html

[2] A. Beaumont-Smith and C. Lim. Parallel prefix adder design. Proc. 15th IEEE Symposium on Computer Arithmetic, pages 218–225, 2001.

[3] A. Booth, A signed binary multiplication technique, Quarterly Journal of Mechanics and Applied Mathematics (1951), pp. 236–240.

[4] A. Chandrakasan and R. Brodersen. Low-Power CMOS design. IEEE Press, 1998.

[5] A. Klimov, A. Shamir (2003). "Cryptographic Applications of T-functions", Selected Areas in Cryptography, SAC 2003, LNCS 3006: 248-261, Springer-Verlag.

[6] A. Menezes, P. van Oorschot, and S. Vanstone, Handbook of Applied Cryptography, pp.3~5, CRC Press, 1996.

[7] A. Menezes, P. van Oorschot, and S. Vanstone, Handbook of Applied Cryptography, pp.43~45, CRC Press, 1996.

[8] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. A statistical test suite for random and pseudorandom number generators for cryptographic applications. 2001. http://www.nist.gov.

[9] ALTERA, ASIC to FPGA Design Methodology & Guidelines, July 2003, ver.1.0

[10] Abraham Sinkov, Elementary Cryptanalysis : A Mathematical Approach, The Mathematical Association of America, 1966. ISBN 0-88385-622-0.

[11] Auguste Kerckhoffs, La cryptographie militaire, Journal des sciences militaires, vol. IX, pp. 5–38, Janvier 1883, pp. 161–191, Février 1883.

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

[13] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C.Hall, N. Ferguson, The Twofish Encryption Algorithm: A 128-Bit Block Cipher," John Wiley & Sons, April 1999, ISBN: 0471353817.

[14] B.W. Bomar, "Implementation of Microprogrammed Control in FPGAs", IEEE Transactions on Industrial Electronics, vol. 49, pp. 415-422, April, 2002.

[15] C. Rechberger. Side channel analysis of stream ciphers. Master's thesis, Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Inffeldgasse 16a, A-8010 Graz, Austria, 2004.

[16] C.D. Walter, Systolic modular multiplication, IEEE Transactions on Computers 42 (1993) (3), pp. 376–378

[17] Chen, Y., Hai Li, Roy, K., Chena-Kok Koh, Cascaded carry-select adder (C/sup 2/SA): a new

structure for low-power CSA design, Low Power Electronics and Design, 2005. ISLPED apos;05. Proceedings of the 2005 International Symposium on Volume , Issue , 8-10 Aug. 2005 Pages 115 – 118

[18] Chris Chatfield, Statistics for Technology: A Course in Applied Statistics, Second Edition, Chapman & Hall/CRC publishers, 1978.

[19] Claude E. Shannon and Warren Weaver: The Mathematical Theory of Communication. The University of Illinois Press, Urbana, Illinois, 1949. ISBN 0-252-72548-4

[20] D. Bernstein, "The Salsa20 Stream Cipher", presented at Symmetric Key Encryption Workshop, Aarhus, Denmark, May, 2005. Also available at www.ecrypt.eu.org/stream/salsa20.html.

[21] D. Coppersmith, S. Halevi, and C. S. Jutla. Cryptanalysis of stream ciphers with linear masking. In CRYPTO, pages 515–532, 2002.

[22] D. Whiting, B. Schneier, and S. Lucks , "Phelix - Fast Encryption and Authentication in a Single Cryptographic Primitive", presented at Symmetric Key Encryption Workshop, Aarhus, Denmark, May, 2005. Also available at www.ecrypt.eu.org/stream/phelixp2.html.

[23] David R. Smith, Digital Transmission Systems, Third Edition, Kluwer Academic Publishers, 2004.

[24] Donald Knuth. The Art of Computer Programming, Volume 1: Fundamental Algorithms, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4. Section 1.2.11: Asymptotic Representations, pp.107–123.

[25] E.F. Brickell, A survey of hardware implementation of RSA. In: G. Brassard, Editor, Advances in Cryptology, Proceedings of CRYPTO'98, Lecture Notes in Computer Science vol. 435, Springer-Verlag (1989), pp. 368–370.

[26] Frédéric Muller, Differential Attacks against the Helix Stream Cipher, FSE 2004, pp94–108

[27] G.J. Simmons, editor. Contemporary Cryptology, The Science of Information Integrity. IEEE, New York, 1992.

[28] G.S. Vernam. Cipher printing telegraph systems for secret wire and radio telegraphic communications. J. Am. Inst. Elec. Eng., 55:109-115, 1926.

[29] Guido Bertoni1, Luca Breveglieri1, Pasqualina Fragneto, Marco Macchetti,and Stefano Marchesin, "Efficient Software Implementation of AES on 32-Bit Platforms," in Cryptographic Hardware and Embedded Systems - CHES 2002,, pp. 159-171, B.S. Kaliski Jr., .K. Ko, C. Paar.

[30] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," Cryptology ePrint Archive, Report 2004/10, 2004.

[31] H. Englund and T. Johansson. A new simple technique to attack filter generators and related ciphers. In Selected Areas in Cryptography, pages 39–53, 2004.

[32] H. Lipmaa, J. Wallén, P. Dumas, On the Additive Differential Probability of Exclusive-Or," Fast Software Encryption 2004 (B. Roy, W. Meier, eds.), vol. 3017 of LNCS, pp. 317-331,

Springer-Verlag, 2004.

[33] H. Lipmaa, S. Moriai, Efficient Algorithms for Computing Differential Properties of Addition," FSE 2001 (M. Matsui, ed.), vol. 2355 of LNCS, pp. 336-350, Springer-Verlag, 2002.

[34] Homepage for the eSTREAM project: www.ecrypt.eu.org/stream

[35] Hongjun Wu and Bart Preneel, "Differential-Linear Attacks against the Stream Cipher Phelix", available at http://www.ecrypt.eu.org/stream/phelixp2.html

[36] I. Gonzalez1, F.J. Gomez-Arribas, Ciphering algorithms in MicroBlaze-based embedded systems, IEE Proceedings - Computers and Digital Techniques -- March 2006 -- Volume 153, Issue 2, p. 87-92

[37] J. Wallén, Linear Approximations of Addition modulo 2n, Fast Software Encryption 2003 (T. Johansson, ed.), vol. 2887 of LNCS, pp. 261-273, Springer-Verlag, 2003.

[38] J.J. Hoch and A. Shamir, Fault analysis of stream ciphers, In Marc Joye and Jean-Jacques Quisquater, editors, Chryptographic Hardware and Embedded Systems —CHES 2004, volume 3156 of LNCS, pages 240–253. Springer-Verlag, 2004.

[39] John Waldron, Introduction to RISC Assembly Language Programming, Addison Wesley, 1998. ISBN 0-201-39828-1

[40] K. Hamano, F. Satoh, and M. Ishikawa, "Randomness test using discrete Fourier transform," Technical Report 6841, Technical Research and Development Institute, Japan Defense Agency, Sept. 2003.

[41] K. Rawwat, T. Darwish, and M. Bayoumi, A low power carry select adder with reduces area, Proc. Of Midwest Symposium on Circuits and Systems, pp. 218-221, 2001.

[42] Kai Lai Chung, Elementary Probability Theory with Stochastic Processes.New York: Springer-Verlag, 1979.

[43] Kumar S., Lemke K., Paar C. , Some thoughts about Implementation properties of stream ciphers, SASC2004, 14-15.10.2004

[44] L. Batina, J. Lano, N. Mentens, B. Preneel, I. Verbauwhede, "Energy, Performance, Area versus Security Trade-offs for Stream Ciphers," In ECRYPT Workshop, SASC - The State of the Art of Stream Ciphers, pp. 302-310, 2004

[45] L. Wu, C. Weaver, T. Austin, "CryptoManiac: A Fast Flexible Architecture for Secure Communication", Proceedings of 28th Annual International Symposium on Computer Architecture, 2001.

[46] M. Abramowitz and I. Stegun, Handbook of Mathematical Functions, Applied Mathematics Series. Vol. 55, Washington: National Bureau of Standards, 1964; reprinted 1968 by Dover Publications, New York.

[47] M. D. Galanis, P. Kitsos, G. Kostopoulos, N. Sklavos, O. Koufopavlou, and C.E. Goutis "Comparison of The Hardware Architectures and FPGA Implementations of Stream Ciphers ",

In proceeding of 11th IEEE International Conference on Electronics, Circuits and Systems, (ICECS 2004), Tel-Aviv, Israel, December 13-15, 2004.

[48] M. Kakumu and M. Kinugawa, "Power-supply voltage impact on circuit performance for half and lower submicrometer CMOS LSI," IEEE Trans. Electron Devices, vol. 37, no. 8, pp. 1902-1908, Aug. 1990.

[49] Maurice Kilavuka Inuani and Jonathan Saul, Technology mapping of heterogeneous LUT-based FPGAs, In Luk et al. [LCG97], pages 223–234.

[50] Meltem Sonmez Turan, Ali Doganaksoy, Cagdas Calik, Statistical Analysis of Synchronous Stream Ciphers, eSTREAM, ECRYPT Stream Cipher Project, 2005. www.ecrypt.eu.org/stream/papersdir/2006/012.pdf

[51] Michael Garey, and David S. Johnson: Computers and Intractability: A Guide to the Theory of NP-Completeness. New York: W. H. Freeman & Co., 1979.

[52] MicroBlaze Hardware Reference Guide, Xilinx User Guide, available from the Xilinx website support.xilinx.com, March 2002.

[53] Morioka, S. and Satoh, A., A 10Gbps Full-AES Crypto Design With a Twisted BDD S-Box Architecture, IEEE Tran. on VLSI Systems, Vol. 12, No. 7, July 2004, pp 686-691.

[54] Nick Sawyer, Word Alignment and SONET/SDH Deframing, Xilinx application notes, available at http://direct.xilinx.com/bvdocs/appnotes/xapp652.pdf

[55] Niels Ferguson, Doug Whiting, Bruce Schneier, John Kelsey, Stefan Lucks and Tadayoshi Kohno, Helix: Fast Encryption and Authentication in a Single Cryptographic Primitive, Fast Software Encryption - FSE 2003, pp330–346

[56] Niels Ferguson, Richard Schroeppel, Doug Whiting (2001). "A simple algebraic representation of Rijndael". Proceedings of Selected Areas in Cryptography, 2001, Lecture Notes in Computer Science: pp. 103–111, Springer-Verlag. Retrieved on 2006-10-06.

[57] O. Staffelbach, W. Meier, Cryptographic Significance of the Carry for Ciphers Based on Integer Addition," Crypto '90 (A. Menezes, S. A. Vanstone, eds.), vol. 537 of LNCS, pp. 601-614, Springer-Verlag, 1991.

[58] P. Crowley: "Truncated Differential Cryptanalysis of Five Rounds of Salsa20," SASC 2006 - Stream Ciphers Revisited, Workshop Record, pp.198-202, 2006. Available at http://www.ecrypt.eu.org/stvl/sasc2006/

[59] P. Kocher, J. Jaffe, B. Jun, "Differential Power Analysis," technical report, 1998; later published in Advances in Cryptology - Crypto 99 Proceedings, Lecture Notes In Computer Science Vol. 1666, M. Wiener, ed., Springer-Verlag, 1999.

[60] Patrick Schaumont and Ingrid Verbauwhede, "Hardware/software codesign for stream ciphers", presented at Symmetric Key Encryption Workshop, January, 2007. Also available at http://www.ecrypt.eu.org/stream/hw.html

[61] Quist, Arvin S. (2002). Security Classification of Information, Volume 1. Introduction, History, and Adverse Impacts. Oak Ridge Classification Associates, LLC. Retrieved on 2007-01-11.

[62] R. Hashemian, "A New Design for High Speed and High-Density Carry. Select Adders", 43. rd. Midwest Symposium on Circuits and Systems,. Lansing, Michigan, August 8-11, 2000.

[63] Ruby B. Lee, Xiao Yang, and Zhijie Shi, Validating Word-Oriented Processors for Bit-level Permutations and Multi-word Operations in Pervasive Secure Computing Paradigms, Princeton University Department of Electrical Engineering Technical Report CE-L2002-004, November 2002.

[64] S. B. Wicker and V. K. Bhargava, Reed-Solomon Codes and their Applications, IEEE Press, New York, 1994.

[65] Satoh, A., Morioka, S., Takano, K., and Munetoh, S., A Compact Rijndael Hardware Architecture with S-Box optimization," Asiacrypt 2001, LNCS 2248, pp.239–254, 2001

[66] Simon Fischer, Willi Meier, C?me Berbain, Jean-Francois Biasse, Matt Robshaw, Non-Randomness in eSTREAM Candidates Salsa20 and TSC-4, Indocrypt 2006

[67] Steven Brown, Rovert Francis, Johnathan Rose, and Zvonko Vranesic. Field Programmable Gate Arrays. Kluwer Academic Publishers, 1992

[68] T. Good, W. Chelton, M. Benaissa, Review of stream cipher candidates from a low resource hardware perspective, available at www.ecrypt.eu.org/stream/papersdir/2006/016.pdf

[69] VLSI Computer Architecture, Arithmetic, and CAD Research Group – Department of Electrical Engineering, IIT, Chicago, IL. IIT Standard Cells for AMI 0.5um and TSMC 0.25um/0.18um (Version 1.6.0), 2003.

[70] W. Stallings, Cryptography and Network Security Principles and Practices, Prentice Hall press, third edition, 2003.

[71] W. Stallings, Data and Computer Communications, 8th Edition, Prentice Hall, 2007.

[72] W. Stallings, "The Advanced Encryption Standard", CRYPTOLOGIA, Volume XXVI, NO. 3, July 2002.

[73] Whitfield Diffie and Martin Hellman, New Directions in Cryptography, IEEE Transactions on Information Theory, vol. IT-22, Nov. 1976, pp: 644-654.

[74] Xilinx Inc., San Jose, Calif., "Design Tips for HDL Implementation of Arithmetic Functions," 2000, www.xilinx.com.

[75] Xilinx Inc., San Jose, Calif., "Virtex, 2.5 V Field Programmable Gate Arrays," 2003, www.xilinx.com.

[76] Xuejia Lai. On the Design and Security of Block Ciphers. Hartung-Gorre Verlag, 1992.

[77] Y. Kim and L-S Kim, .64-bit carry-select adder with reduced area,. Electronics Letters, vol. 37, pp. 614-615, May 2001

[78] Yukiyasu Tsunoo, Teruo Saito, Hiroyasu Kubo, Tomoyasu Suzaki and Hiroki Nakashima,

"Differential Cryptanalysis of Salsa20/8". Available at http://www.ecrypt.eu.org/stream/papersdir/2007/010.pdf

[79] Z. Navabi, VHDL analysis and modeling of digital systems, McGraw-Hill press, second edition, 1998.

[80] Thomas Beth and Fred Piper, The Stop-and-Go Generator. EUROCRYPT 1984, p88-92.

[81] http://en.wikipedia.org/wiki/Padding_(cryptography)

[82] S. Vanstone , A. Menezes, P. van Oorschot, Handbook of Applied Cryptography, pp.425, CRC Press, 1996.

[83] Adam Osborne, An Introduction to Microcomputers Volume 1 Basic Concepts,2nd Edition, Osborne-McGraw Hill, Berkely California, 1980, ISBN 0-931988-34-9 pg1-1

[84] A.Th. Schwarzbacher and J.P. Silvennoinen and P.A. Comiskey, Benchmarking CMOS Adder Structures, *Irish Systems and Signals Conference,* Cork, Ireland, pp. 231-234, June 2002.

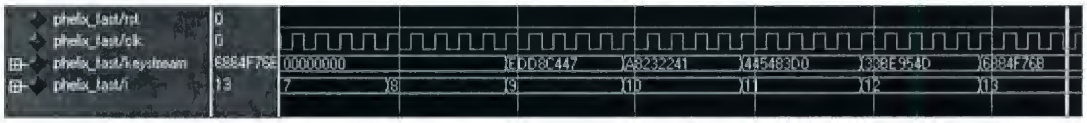
[85] E. Andrew Parr, The Logic Designer's Guidebook, NY: McGraw-Hill 1984, pp. 21.

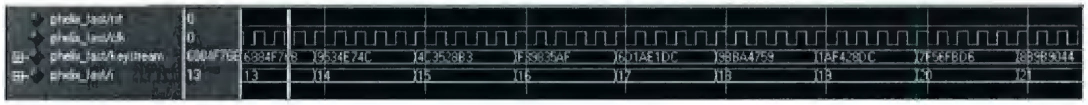
Appendix A: Selected Simulations Results for Phelix

The simulation results were obtained from both fast and compact structures. The values of the parameters for them are chosen differently, but the input length is the same. That is, 64 bytes of the plaintext, 32 bytes of the inputkey and 16 bytes of the nonce.

Phelix_fast

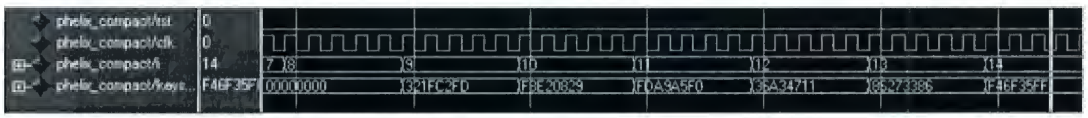
```
plaintext:
30379186 a409778e a1f4a193 adcbc457 8064e117 611ef4db 89764619 7e06d964
a530c522 fa9ea710 12461967 d023bde7 9c28d69b 6579d4d1 160ce27 13d46583
inputkey:
eb840c37 e6b27430 550ca430 789b6a0e 7e498f2f 68dcd5fc cce57a89 a5ea440f
nonce:
f9d2067f c9022dad cabd09cd e00f8d36
workingKey:
c6a2c00d f47a93b7 a2b77454 43db9eb0 7cccd425 6934d90d 38c5b13f 2ac7d22e
keystream:
edd8c447 a8232241 445483d0 33be954d 6884f76b 9534e74c 4c3528b3 f39835af
6d1ae1dc 9bba4759 1af428dc 7f56fbd6 8b9b9044 c7a8a9e 54a8092c 566e2940
```

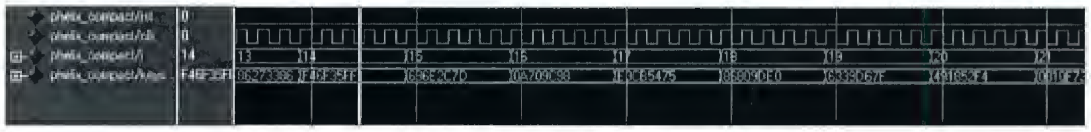




Phelix_compact

```
plaintext:
5007fc86 1a0ea26b dc20138 1bbe0e0d 45dc22b5 bcfa2acf a1b16166 dca89b47
3ad67e03 5d1b9ad6 7b689ee9 b68f8426 828097b7 a8f3d351 b663478a bb5ea400
inputkey:
4dc4ee80 b9b95ddb e9605b1f 717bdb6b 28302ad6 25bff123 3ce75d00 5eb17d3
nonce:
19a2717f 3f07588b 368a6972 4e01d6ec
workingKey:
df9d8059 3bf6d9d6 ee1964bd 4a63c4d8 d173df6 2f531cad 7accd14d ad571d89
keystream:
321fc2fd f8e20829 fda9a5f0 36a34711 86273386 f46f35ff 696e2c7d a709c98
e3cb5475 86809de0 6339d67f 491852f4 810f728 d9c5ee3b 1d6b4a64 83f0f5ca
```

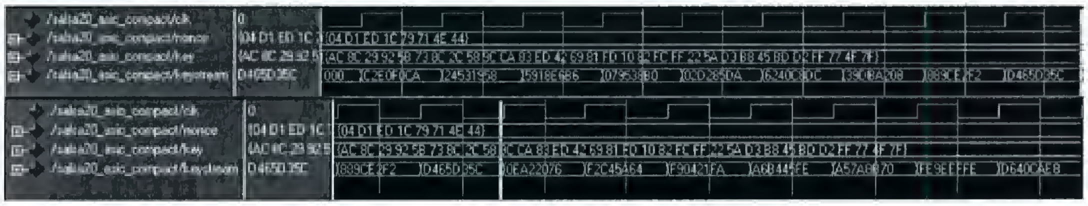




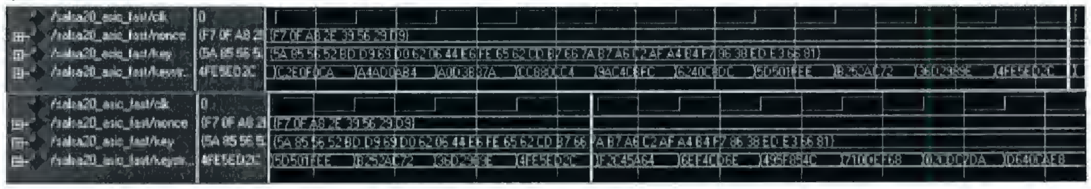
Appendix B: Selected Simulations Results for Salsa20

Simulation results obtained from various structures are given. Either the nonce or the sequence number consists of 8 bytes. The keystream shown in the waveform is the result of one encryption process, which generates a 64-byte data block.

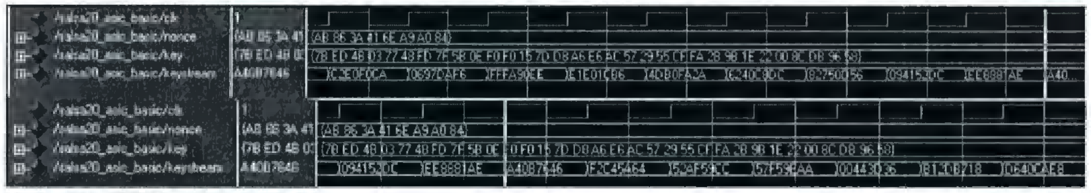
Salsa20_asic_compact
Nonce & SequenceNumber: 4 d1 ed 1c 79 71 4e 44 ae e9 32 6a 3b 10 51 87
Key: ac 8c 29 92 5b 73 8c 2c 58 9c ca 83 ed 42 69 81 fd 10 82 fc ff 22 5a d3 b8 45 bd d2 ff 77 4f 7f
Keystream: c2e0f0ca 24531958 5918e6b6 79538b0 2d285da 6240c8dc 39dba208 889ce2f2 d465d35c ea22076 f2c45a64 f90421fa a6b445fe a57a8b70 fe9eeffe d640cae8


/salsa20_asic_compact/clk
/salsa20_asic_compact/nonce
/salsa20_asic_compact/key
/salsa20_asic_compact/keystream

Salsa20_asic_fast
Nonce & SequenceNumber: f7 f a8 2e 39 56 29 d9 4f 4c 69 1b 96 f6 f2 a7
Key: 5a 85 56 52 bd d9 69 d0 62 6 44 e6 fe 65 62 cd b7 66 7a b7 a6 c2 af a4 b4 f7 86 38 ed e3 66 81
Keystream: c2e0f0ca a4ad0ab4 a0d3b37a cc880cc4 9ac4cbfc 6240c8dc 5d501fee b252ac72 36d2989e 4fe5ed2c f2c45a64 6ef4cd6e 495f854c 710def68 2cdc7da d640cae8

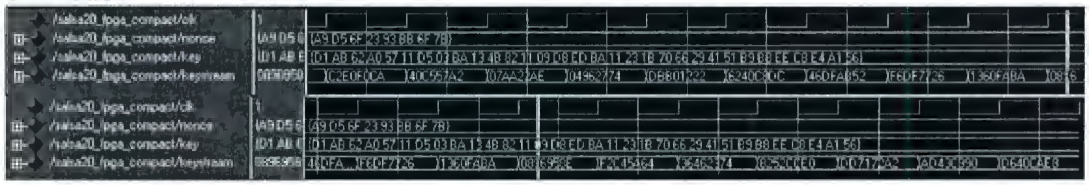


Salsa20_asic_basic_iterative
Nonce & SequenceNumber: ab 86 3a 41 6e a9 a0 84 d7 40 44 f7 23 bb 5 52
Key: 7b ed 4b 3 77 48 fd 7f 5b e f0 f0 15 7d d8 a6 e6 ac 57 29 55 cf fa 2b 9b 1e 22 0 8c db 96 58
Keystream: c2e0f0ca 697daf6 fffa90ee e1e01cb6 4db0fa2a 6240c8dc 82750d56 94152dc ee8881ae a40b7646 f2c45a64 52af59cc 57f59eaa 443d36 b12db718 d640cae8



Salsa20_FPGA_compact

```
Nonce & SequenceNumber: a9 d5 6f 23 93 bb 6f 7b 5d 7d b0 89 c7 4a 4b 4
Key: d1 ab 62 a0 57 11 d5 3 ba 13 4b 82 11 9 d8 ed ba 11 23 1b 70 66 29 41
51 b9 b8 ee c8 e4 a1 56
Keystream: c2e0f0ca 40c557a2 7aa22ae 4962774 dbb01222 6240c8dc 46dfab52
f6df7726 1360faba 896958e f2c45a64 36462374 8252cce0 dd7172a2 ad43c990
d640cae8
```


/salsa20_fpga_compact/clk
/salsa20_fpga_compact/nonce
/salsa20_fpga_compact/key
/salsa20_fpga_compact/keystream

Appendix D: Selected Source Code for Phelix

```
library IEEE;
use IEEE.std_logic_1164.all;

package mypackage is

       type byte_vector is array (natural range<>) of bit_vector(7 downto 0) ;
       type word_vector is array (natural range<>) of bit_vector(31 downto 0) ;
       type word_vector_std is array (natural range<>) of std_logic_vector(31 downto 0) ;

       function bv2slv (b:bit_vector) return std_logic_vector;

end mypackage;

package body mypackage is

    function bv2slv (b:bit_vector) return std_logic_vector is
       variable result: std_logic_vector(31 downto 0);
    begin
       for i in 0 to 31 loop
           case b(i) is
               when '0' => result(i) := '0';
               when '1' => result(i) := '1';
           end case;
       end loop;
```

```
        return result;
    end;
end mypackage;

-----------------------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.All;

entity KSadder is
generic ( size : natural := 32 ) ;
port ( A, B : in Std_Logic_Vector ( size -1 downto 0 ) ;
sum : out Std_Logic_Vector ( size -1 downto 0 ) ;
Cout : out Std_Logic ) ;
end KSadder ;

architecture structural of KSadder is
-- G(i)(j), P(i)(j) : "group Generate", "group Propagate". i = group left position, j = group
right position
type Tr is array (size -1 downto 0) of Std_Logic_Vector (size -1 downto 0) ;
signal G, P : Tr ;

procedure half_adder
(signal G, P : out Std_Logic; signal A, B : in Std_Logic) is
begin G <= A and B; P <= A xor B; end half_adder;

procedure BK
(signal GO, PO : out Std_Logic; signal GI1, PI1, GI2, PI2 : in Std_Logic) is
begin GO <= GI1 or ( PI1 and GI2 ); PO <= PI1 and PI2; end BK;

function koggestone ( i , j : integer) return integer is
variable p : integer ;
begin
p := 2 ;
while p <= i - j loop
p := p + p ;
end loop ;
if j = 0 or i - j + 1 = p then
return i + 1 - p/2 ;
else
return 0 ;
end if ;
end koggestone ;

begin

-- "half_adder" cells row
half_adders    : for i in size -1 downto 0 generate
```

```
half_adder( G(i)(i) , P(i)(i) ,     A(i) , B(i) ) ;
end generate half_adder_row ;

-- operator
for_i : for i in size -1 downto 1 generate
for_j : for j in i -1 downto 0 generate
if_kg : if koggestone (i, j) > 0 generate
BK( G(i)(j), P(i)(j), G(i)(koggestone(i, j)), P(i)(koggestone(i, j)), G(koggestone(i, j) -1)(j),
P(koggestone(i, j) -1)(j) ) ;
end generate if_kg ;
end generate for_j ;
end generate for_i ;

Cout <= G(size -1)(0) ;
-- "XOR" gates row
XOR_row   : for i in size -1 downto 1 generate
Sum(i) <= P(i)(i) xor G(i-1)(0) ;
end generate XOR_row ;
Sum(0) <= P(0)(0) ;

end structural ;
-----------------------------------------------------------------------------------------------------------
-- H block for high speed

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity h_block is
  port ( clk,rst, start  : in std_logic;
         zin0 : in std_logic_vector(31 downto 0);
         zin1 : in std_logic_vector(31 downto 0);
         zin2 : in std_logic_vector(31 downto 0);
         zin3 : in std_logic_vector(31 downto 0);
         zin4 : in std_logic_vector(31 downto 0);
         xin0 : in std_logic_vector(31 downto 0);
         xin1 : in std_logic_vector(31 downto 0);
         zo0 : out std_logic_vector(31 downto 0);
         zo1 : out std_logic_vector(31 downto 0);
         zo2 : out std_logic_vector(31 downto 0);
         zo3 : out std_logic_vector(31 downto 0);
         zo4 : out std_logic_vector(31 downto 0)
        );
end h_block;

architecture rtl of h_block is
    signal zi0, z_10, z_20, z_30, z_40,
           zi1, z_11, z_21, z_31, z_41,
        zi2, z_12, z_22, z_32, z_42,
        zi3, z_13, z_23, z_33, z_43, z_53, z_63,
        zi4, z_14, z_24, z_34, z_44,
        x0, x1
    : std_logic_vector(31 downto 0);
    -- six adders, six useless cout
    signal cout0, cout1, cout2, cout3, cout4, cout5
    : std_logic;
    signal flag : integer := 0;
    component Ksadder
        generic ( size : natural := 32 ) ;                          -- size: Number of bits
        port ( A, B : in Std_Logic_Vector ( size -1 downto 0 ) ;  -- A,B: addends
        S : out Std_Logic_Vector ( size -1 downto 0 ) ;     -- S: Sum;
        Cout : out Std_Logic ) ;                             -- carry out
    end component ;

    -- left rotation
    function lrotate( din : std_logic_vector(31 downto 0);
                      n   : integer)
    return std_logic_vector is
    variable dout: std_logic_vector(31 downto 0);
    begin
        dout := din((31-n) downto 0) & din(31 downto (32-n));
        return dout;
    end lrotate;

begin
            zi0 <= zin0 when flag = 0;
            zi1 <= zin1 when flag = 0;
            zi2 <= zin2 when flag = 0;
            zi3 <= zin3 when flag = 0;
            zi4 <= zin4 when flag = 0;
            x0 <= xin0 when flag = 0;
            x1 <= xin1 when flag = 0;
    z_13 <= zi3 xor x0;
    add_z_10: KSadder port map (zi0, z_13, z_10, cout0);
    z_23 <= lrotate (zi3, 15);
    add_z_11: KSadder port map (zi1, zi4, z_11, cout1);
    z_14 <= lrotate (zi4, 25);
    z_12 <= zi2 xor z_10;
    z_20 <= lrotate (z_10, 9);
    z_33 <= z_23 xor z_11;
    z_21 <= lrotate (z_11, 10);
    add_z_24: KSadder port map (z_14, z_12, z_24, cout2);
    z_22 <= lrotate (z_12, 17);

    add_z_43: KSadder port map (z_33, x1, z_43, cout3);
    z_30 <= z_20 xor z_43;
    z_53 <= lrotate (z_33, 30);
    z_31 <= z_21 xor z_24;
    z_34 <= lrotate (z_24, 13);
    add_z_32: KSadder port map (z_22, z_30, z_32, cout4);
    z_40 <= lrotate (z_30, 20);
    add_z_63: KSadder port map (z_53, z_31, z_63, cout5);
    z_41 <= lrotate (z_31, 11);
    z_44 <= z_34 xor z_32;
    z_42 <= lrotate (z_32, 5);
syn_in: process(rst, clk, start)
       begin
         if (rst = '1')then
             flag <= 0;
         elsif (clk='1' and clk' event) then
             if (flag = 0 and start = '1') then
                 flag <= 1;
             else
                 flag <= 0;
             end if;
         end if;
     end process syn_in;
syn_out: process (clk, flag)
        begin
            if (clk = '1' and clk'event ) then
               if (flag = 1)then
            -- output registers
            zo0 <= z_40;
            zo1 <= z_41;
            zo2 <= z_42;
            zo3 <= z_63;
            zo4 <= z_44;
               end if;
        end if;
    end process syn_out;

end rtl;
```

-- H block for compactness

```
library ieee;
use ieee.std_logic_1164.ALL;

entity H_func_dp is
   port ( k0      : in    std_logic_vector (31 downto 0);
          k1      : in    std_logic_vector (31 downto 0);
          ls0     : in    std_logic;
          ls1     : in    std_logic;
          ls2     : in    std_logic;
          ls3     : in    std_logic;
          ls4     : in    std_logic;
          ls5     : in    std_logic;
          ls6     : in    std_logic;
          mux_sel : in    std_logic_vector (2 downto 0);
          w00     : in    std_logic_vector (31 downto 0);
          w01     : in    std_logic_vector (31 downto 0);
          w02     : in    std_logic_vector (31 downto 0);
          w03     : in    std_logic_vector (31 downto 0);
          w04     : in    std_logic_vector (31 downto 0);
          w40     : out   std_logic_vector (31 downto 0);
          w41     : out   std_logic_vector (31 downto 0);
          w42     : out   std_logic_vector (31 downto 0);
          w43     : out   std_logic_vector (31 downto 0);
          w44     : out   std_logic_vector (31 downto 0));
end H_func_dp;

architecture BEHAVIORAL of H_func_dp is
   signal ain0       : std_logic_vector (31 downto 0);
   signal ain1       : std_logic_vector (31 downto 0);
   signal aout       : std_logic_vector (31 downto 0);
   signal k0_in      : std_logic_vector (31 downto 0);
   signal w00_in     : std_logic_vector (31 downto 0);
   signal w01_in     : std_logic_vector (31 downto 0);
   signal w02_in     : std_logic_vector (31 downto 0);
   signal w03_in     : std_logic_vector (31 downto 0);
   signal w03_xor_k0 : std_logic_vector (31 downto 0);
   signal w04_in     : std_logic_vector (31 downto 0);
   signal w10        : std_logic_vector (31 downto 0);
   signal w11        : std_logic_vector (31 downto 0);
   signal w12        : std_logic_vector (31 downto 0);
   signal w13        : std_logic_vector (31 downto 0);
   signal w14        : std_logic_vector (31 downto 0);
   signal w20        : std_logic_vector (31 downto 0);
   signal w21        : std_logic_vector (31 downto 0);
   signal w22        : std_logic_vector (31 downto 0);
   signal w23        : std_logic_vector (31 downto 0);
   signal w23_add_k1 : std_logic_vector (31 downto 0);
   signal w24        : std_logic_vector (31 downto 0);
   signal w30        : std_logic_vector (31 downto 0);
   signal w31        : std_logic_vector (31 downto 0);
   signal w32        : std_logic_vector (31 downto 0);
   signal w33        : std_logic_vector (31 downto 0);
   signal w34        : std_logic_vector (31 downto 0);
   signal w40_in     : std_logic_vector (31 downto 0);
   signal w41_in     : std_logic_vector (31 downto 0);
   signal w42_in     : std_logic_vector (31 downto 0);
   signal w44_in     : std_logic_vector (31 downto 0);
   signal XLXN_21    : std_logic_vector (31 downto 0);
   component latch
      port ( gate : in    std_logic;
          din  : in    std_logic_vector (31 downto 0);
          dout : out   std_logic_vector (31 downto 0));
end component;

component mux_6to1
   port ( i0 : in    std_logic_vector (31 downto 0);
          i1 : in    std_logic_vector (31 downto 0);
          i2 : in    std_logic_vector (31 downto 0);
          i3 : in    std_logic_vector (31 downto 0);
          i4 : in    std_logic_vector (31 downto 0);
          i5 : in    std_logic_vector (31 downto 0);
          s  : in    std_logic_vector (2 downto 0);
          q  : inout std_logic_vector (31 downto 0));
end component;

component rot15
   port ( din  : in    std_logic_vector (31 downto 0);
          dout : out   std_logic_vector (31 downto 0));
end component;

component rot25
   port ( din  : in    std_logic_vector (31 downto 0);
          dout : out   std_logic_vector (31 downto 0));
end component;

component adder_predefined
   port ( A : in    std_logic_vector (31 downto 0);
          B : in    std_logic_vector (31 downto 0);
          S : out   std_logic_vector (31 downto 0));
end component;

component rot9
   port ( din  : in    std_logic_vector (31 downto 0);
          dout : out   std_logic_vector (31 downto 0));
end component;

component rot10
   port ( din  : in    std_logic_vector (31 downto 0);
          dout : out   std_logic_vector (31 downto 0));
end component;

component rot17
   port ( din  : in    std_logic_vector (31 downto 0);
          dout : out   std_logic_vector (31 downto 0));
end component;

component rot30
   port ( din  : in    std_logic_vector (31 downto 0);
          dout : out   std_logic_vector (31 downto 0));
```

```
    end component;

    component rot13
       port ( din  : in    std_logic_vector (31 downto 0);
              dout : out   std_logic_vector (31 downto 0));
    end component;

    component rot20
       port ( din  : in    std_logic_vector (31 downto 0);
              dout : out   std_logic_vector (31 downto 0));
    end component;

    component rot11
       port ( din  : in    std_logic_vector (31 downto 0);
              dout : out   std_logic_vector (31 downto 0));
    end component;

    component rot5
       port ( din  : in    std_logic_vector (31 downto 0);
              dout : out   std_logic_vector (31 downto 0));
    end component;

    component xor_array
       port ( a : in    std_logic_vector (31 downto 0);
              b : in    std_logic_vector (31 downto 0);
              c : out   std_logic_vector (31 downto 0));
    end component;

begin
    latch_w10 : latch
       port map (din(31 downto 0)=>aout(31 downto 0),
                 gate=>ls0,
                 dout(31 downto 0)=>w10(31 downto 0));

    latch_w11 : latch
       port map (din(31 downto 0)=>aout(31 downto 0),
                 gate=>ls1,
                 dout(31 downto 0)=>w11(31 downto 0));

    latch_w23K1 : latch
       port map (din(31 downto 0)=>aout(31 downto 0),
                 gate=>ls3,
                 dout(31 downto 0)=>w23_add_k1(31 downto 0));

    latch_w32 : latch
       port map (din(31 downto 0)=>aout(31 downto 0),
                 gate=>ls4,
                 dout(31 downto 0)=>w32(31 downto 0));
```

```
mux_ain0 : mux_6tol
   port map (i0(31 downto 0)=>w00_in(31 downto 0),
             i1(31 downto 0)=>w01_in(31 downto 0),
             i2(31 downto 0)=>w14(31 downto 0),
             i3(31 downto 0)=>w23(31 downto 0),
             i4(31 downto 0)=>w22(31 downto 0),
             i5(31 downto 0)=>w33(31 downto 0),
             s(2 downto 0)=>mux_sel(2 downto 0),
             q(31 downto 0)=>ain0(31 downto 0));

mux_ain1 : mux_6tol
   port map (i0(31 downto 0)=>w03_xor_k0(31 downto 0),
             i1(31 downto 0)=>w04_in(31 downto 0),
             i2(31 downto 0)=>w12(31 downto 0),
             i3(31 downto 0)=>XLXN_21(31 downto 0),
             i4(31 downto 0)=>w30(31 downto 0),
             i5(31 downto 0)=>w31(31 downto 0),
             s(2 downto 0)=>mux_sel(2 downto 0),
             q(31 downto 0)=>ain1(31 downto 0));

XLXI_83 : rot15
   port map (din(31 downto 0)=>w03_in(31 downto 0),
             dout(31 downto 0)=>w13(31 downto 0));

XLXI_84 : rot25
   port map (din(31 downto 0)=>w04_in(31 downto 0),
             dout(31 downto 0)=>w14(31 downto 0));

XLXI_86 : adder_predefined
   port map (A(31 downto 0)=>ain1(31 downto 0),
             B(31 downto 0)=>ain0(31 downto 0),
             S(31 downto 0)=>aout(31 downto 0));

XLXI_115 : rot9
   port map (din(31 downto 0)=>w10(31 downto 0),
             dout(31 downto 0)=>w20(31 downto 0));

XLXI_117 : rot10
   port map (din(31 downto 0)=>w11(31 downto 0),
             dout(31 downto 0)=>w21(31 downto 0));

XLXI_130 : latch
   port map (din(31 downto 0)=>aout(31 downto 0),
             gate=>ls2,
             dout(31 downto 0)=>w24(31 downto 0));

XLXI_131 : rot17
   port map (din(31 downto 0)=>w12(31 downto 0),
             dout(31 downto 0)=>w22(31 downto 0));
```

```
XLXI_134 : rot30
   port map (din(31 downto 0)=>w23(31 downto 0),
             dout(31 downto 0)=>w33(31 downto 0));

XLXI_136 : rot13
   port map (din(31 downto 0)=>w24(31 downto 0),
             dout(31 downto 0)=>w34(31 downto 0));

XLXI_138 : rot20
   port map (din(31 downto 0)=>w30(31 downto 0),
             dout(31 downto 0)=>w40_in(31 downto 0));

XLXI_142 : rot11
   port map (din(31 downto 0)=>w31(31 downto 0),
             dout(31 downto 0)=>w41_in(31 downto 0));

XLXI_144 : rot5
   port map (din(31 downto 0)=>w32(31 downto 0),
             dout(31 downto 0)=>w42_in(31 downto 0));

latch_win0 : latch
   port map (din(31 downto 0)=>w00(31 downto 0),
             gate=>ls5,
             dout(31 downto 0)=>w00_in(31 downto 0));

latch_win1 : latch
   port map (din(31 downto 0)=>w01(31 downto 0),
             gate=>ls5,
             dout(31 downto 0)=>w01_in(31 downto 0));

latch_win2 : latch
   port map (din(31 downto 0)=>w02(31 downto 0),
             gate=>ls5,
             dout(31 downto 0)=>w02_in(31 downto 0));

latch_win3 : latch
   port map (din(31 downto 0)=>w03(31 downto 0),
             gate=>ls5,
             dout(31 downto 0)=>w03_in(31 downto 0));

latch_win4 : latch
   port map (din(31 downto 0)=>w04(31 downto 0),
             gate=>ls5,
             dout(31 downto 0)=>w04_in(31 downto 0));

latch_k0 : latch
   port map (din(31 downto 0)=>k0(31 downto 0),
             gate=>ls5,
            dout(31 downto 0)=>k0_in(31 downto 0));

latch_k1 : latch
   port map (din(31 downto 0)=>k1(31 downto 0),
            gate=>ls5,
            dout(31 downto 0)=>XLXN_21(31 downto 0));

XLXI_157 : latch
   port map (din(31 downto 0)=>aout(31 downto 0),
            gate=>ls6,
            dout(31 downto 0)=>w43(31 downto 0));

XLXI_158 : latch
   port map (din(31 downto 0)=>w42_in(31 downto 0),
            gate=>ls6,
            dout(31 downto 0)=>w42(31 downto 0));

XLXI_159 : latch
   port map (din(31 downto 0)=>w44_in(31 downto 0),
            gate=>ls6,
            dout(31 downto 0)=>w44(31 downto 0));

XLXI_160 : latch
   port map (din(31 downto 0)=>w41_in(31 downto 0),
            gate=>ls6,
            dout(31 downto 0)=>w41(31 downto 0));

XLXI_161 : latch
   port map (din(31 downto 0)=>w40_in(31 downto 0),
            gate=>ls6,
            dout(31 downto 0)=>w40(31 downto 0));

xor_w10 : xor_array
   port map (a(31 downto 0)=>k0_in(31 downto 0),
            b(31 downto 0)=>w03_in(31 downto 0),
            c(31 downto 0)=>w03_xor_k0(31 downto 0));

xor_w12 : xor_array
   port map (a(31 downto 0)=>w10(31 downto 0),
            b(31 downto 0)=>w02_in(31 downto 0),
            c(31 downto 0)=>w12(31 downto 0));

xor_w30 : xor_array
   port map (a(31 downto 0)=>w20(31 downto 0),
            b(31 downto 0)=>w23_add_k1(31 downto 0),
            c(31 downto 0)=>w30(31 downto 0));

xor_w31 : xor_array
   port map (a(31 downto 0)=>w21(31 downto 0),
              b(31 downto 0)=>w24(31 downto 0),
              c(31 downto 0)=>w31(31 downto 0));

   xor_w44 : xor_array
      port map (a(31 downto 0)=>w34(31 downto 0),
              b(31 downto 0)=>w32(31 downto 0),
              c(31 downto 0)=>w44_in(31 downto 0));

   xor_23 : xor_array
      port map (a(31 downto 0)=>w11(31 downto 0),
              b(31 downto 0)=>w13(31 downto 0),
              c(31 downto 0)=>w23(31 downto 0));

end BEHAVIORAL;

LIBRARY ieee;
USE ieee.std_logic_1164.all;


ENTITY SHELL_H_CTR IS
    PORT (CLK,RESET,start: IN std_logic;
        done,ls0,ls1,ls2,ls3,ls4,ls5,ls6,mux_sel0,mux_sel1,mux_sel2 : OUT std_logic
            );

END;

ARCHITECTURE BEHAVIOR OF SHELL_H_CTR IS
    TYPE type_sreg IS (H_done,idle,w10_gen,w10_load,w11_gen,w11_load,w23K1_gen,
        w23K1_load,w24_gen,w24_load,w32_gen,w32_load,w44_gen);
    SIGNAL sreg, next_sreg : type_sreg;
    SIGNAL ls : std_logic_vector (6 DOWNTO 0);
    SIGNAL mux_sel : std_logic_vector (2 DOWNTO 0);
BEGIN
    PROCESS (CLK, RESET, next_sreg)
    BEGIN
        IF ( RESET='1' ) THEN
            sreg <= idle;
        ELSIF CLK='1' AND CLK'event THEN
            sreg <= next_sreg;
        END IF;
    END PROCESS;

    PROCESS (sreg,start)
    BEGIN

        CASE sreg IS
            WHEN H_done =>
                done<='1';
                mux_sel <= (std_logic_vector'("101"));
        ls <= (std_logic_vector'("1000000"));
        next_sreg<=idle;
    WHEN idle =>
        done<='0';
        mux_sel <= (std_logic_vector'("000"));
        ls <= (std_logic_vector'("0100000"));
        IF ( start='1' ) THEN
            next_sreg<=w10_gen;
         ELSE
            next_sreg<=idle;
        END IF;
    WHEN w10_gen =>
        done<='0';
        mux_sel <= (std_logic_vector'("000"));
        ls <= (std_logic_vector'("0000001"));
        IF ( start='1' ) THEN
            next_sreg<=w10_load;
         ELSE
            next_sreg<=w10_gen;
        END IF;
    WHEN w10_load =>
        done<='0';
        mux_sel <= (std_logic_vector'("000"));
        ls <= (std_logic_vector'("0000001"));
        IF ( start='1' ) THEN
            next_sreg<=w11_gen;
         ELSE
            next_sreg<=w10_load;
        END IF;
    WHEN w11_gen =>
        done<='0';
        mux_sel <= (std_logic_vector'("001"));
        ls <= (std_logic_vector'("0000010"));
        IF ( start='1' ) THEN
            next_sreg<=w11_load;
         ELSE
            next_sreg<=w11_gen;
        END IF;
    WHEN w11_load =>
        done<='0';
        mux_sel <= (std_logic_vector'("001"));
        ls <= (std_logic_vector'("0000010"));
        IF ( start='1' ) THEN
            next_sreg<=w24_gen;
         ELSE
            next_sreg<=w11_load;
        END IF;
    WHEN w23K1_gen =>
        done<='0';
    mux_sel <= (std_logic_vector'("011"));
    ls <= (std_logic_vector'("0001000"));
    IF ( start='1' ) THEN
        next_sreg<=w23K1_load;
     ELSE
        next_sreg<=w23K1_gen;
    END IF;
WHEN w23K1_load =>
    done<='0';
    mux_sel <= (std_logic_vector'("011"));
    ls <= (std_logic_vector'("0001000"));
    IF ( start='1' ) THEN
        next_sreg<=w32_gen;
     ELSE
        next_sreg<=w23K1_load;
    END IF;
WHEN w24_gen =>
    done<='0';
    mux_sel <= (std_logic_vector'("010"));
    ls <= (std_logic_vector'("0000100"));
    IF ( start='1' ) THEN
        next_sreg<=w24_load;
     ELSE
        next_sreg<=w24_gen;
    END IF;
WHEN w24_load =>
    done<='0';
    mux_sel <= (std_logic_vector'("010"));
    ls <= (std_logic_vector'("0000100"));
    IF ( start='1' ) THEN
        next_sreg<=w23K1_gen;
     ELSE
        next_sreg<=w24_load;
    END IF;
WHEN w32_gen =>
    done<='0';
    mux_sel <= (std_logic_vector'("100"));
    ls <= (std_logic_vector'("0010000"));
    IF ( start='1' ) THEN
        next_sreg<=w32_load;
     ELSE
        next_sreg<=w32_gen;
    END IF;
WHEN w32_load =>
    done<='0';
    mux_sel <= (std_logic_vector'("100"));
    ls <= (std_logic_vector'("0010000"));
    IF ( start='1' ) THEN
        next_sreg<=w44_gen;
```

```
                ELSE
                    next_sreg<=w32_load;
                END IF;
            WHEN w44_gen =>
                done<='0';
                mux_sel <= (std_logic_vector'("101"));
                ls <= (std_logic_vector'("1000000"));
                IF ( start='1' ) THEN
                    next_sreg<=H_done;
                 ELSE
                    next_sreg<=w44_gen;
                END IF;
            WHEN OTHERS =>
        END CASE;

    END PROCESS;

        ls6 <= ls(6);
        ls5 <= ls(5);
        ls4 <= ls(4);
        ls3 <= ls(3);
        ls2 <= ls(2);
        ls1 <= ls(1);
        ls0 <= ls(0);
        mux_sel2 <= mux_sel(2);
        mux_sel1 <= mux_sel(1);
        mux_sel0 <= mux_sel(0);
END BEHAVIOR;

LIBRARY ieee;
USE ieee.std_logic_1164.all;


ENTITY H_CTR IS
    PORT (ls : OUT std_logic_vector (6 DOWNTO 0);
        mux_sel : OUT std_logic_vector (2 DOWNTO 0);
        CLK,RESET,start: IN std_logic;
        done : OUT std_logic);
END;

ARCHITECTURE BEHAVIOR OF H_CTR IS
    COMPONENT SHELL_H_CTR
        PORT (CLK,RESET,start: IN std_logic;
            done,ls0,ls1,ls2,ls3,ls4,ls5,ls6,mux_sel0,mux_sel1,mux_sel2 : OUT
                std_logic);
    END COMPONENT;
BEGIN
    SHELL1_H_CTR : SHELL_H_CTR PORT MAP (CLK=>CLK,RESET=>RESET,start=>start,done
        =>done,ls0=>ls(0),ls1=>ls(1),ls2=>ls(2),ls3=>ls(3),ls4=>ls(4),ls5=>ls(5),ls6
        =>ls(6),mux_sel0=>mux_sel(0),mux_sel1=>mux_sel(1),mux_sel2=>mux_sel(2));
END BEHAVIOR;
```

Appendix E: Selected Source Code for Salsa20

```
---- ASIC_Compact

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY ieee;
USE ieee.std_logic_unsigned.all;


ENTITY SHELL_CONTRO IS
    PORT (CLK,mem_done,quarter_done,RESET,start: IN std_logic;
        done,load_all,mux_m0,m0_start,m1_start,quarter_rd_start,round0,round1,round2,
            round3,round4,round5,round6,serial : OUT std_logic);
END;

ARCHITECTURE BEHAVIOR OF SHELL_CONTRO IS
    TYPE type_sreg IS (add,idle,load_rows,load_z,quarter_en);
    SIGNAL sreg, next_sreg : type_sreg;
    SIGNAL round : std_logic_vector (6 DOWNTO 0);

BEGIN
    PROCESS (CLK, RESET)
    BEGIN
        IF ( RESET='1' ) THEN
            sreg <= idle;
        ELSIF CLK='1' AND CLK'event THEN
            sreg <= next_sreg;
        END IF;
    END PROCESS;

    PROCESS (sreg, mem_done,quarter_done,start,round)
    BEGIN

        next_sreg<=add;

        CASE sreg IS
            WHEN add =>
                IF ( start='1' AND mem_done='1' ) THEN
                    next_sreg<=load_rows;
                 ELSE
                    next_sreg<=add;
                END IF;
            WHEN idle =>
                IF ( start='1' ) THEN
                next_sreg<=load_rows;
             ELSE
                next_sreg<=idle;
           END IF;
        WHEN load_rows =>
           IF ( start='1' AND mem_done='1' ) THEN
                next_sreg<=quarter_en;
             ELSE
                next_sreg<=load_rows;
           END IF;
        WHEN load_z =>
           IF ( start='0' ) THEN
                next_sreg<=load_z;
           ELSIF (round=87 and mem_done='1') THEN
                next_sreg<=add;
           ELSIF (round<87 and mem_done='1') THEN
                next_sreg<=quarter_en;
           END IF;
        WHEN quarter_en =>
           IF ( start='1' AND quarter_done='1' ) THEN
                next_sreg<=load_z;
             ELSE
                next_sreg<=quarter_en;
           END IF;
        WHEN OTHERS =>
    END CASE;

END PROCESS;

PROCESS( sreg )
    BEGIN

        CASE sreg IS
        WHEN add =>
             mux_m0<='1';
               m0_start<='1';
               m1_start<='1';
               load_all<='0';
               serial<='1';
               quarter_rd_start<='0';
               done<='1';
               round <= (std_logic_vector'("0000111"));
        WHEN idle =>
               mux_m0<='0';
               m0_start<='0';
               m1_start<='0';
               load_all<='0';
               serial<='0';
               quarter_rd_start<='0';
               done<='0';
               round <= (std_logic_vector'("0000111"));
        WHEN load_rows =>
               mux_m0<='0';
               m0_start<='1';
               m1_start<='1';
               load_all<='1';
```

```
                    serial<='0';
                    quarter_rd_start<='0';
                    done<='0';
                    round <= (std_logic_vector'("0000111"));
                WHEN load_z =>
                   mux_m0<='1';
                    m0_start<='1';
                    m1_start<='0';
                    load_all<='0';
                    serial<='0';
                    quarter_rd_start<='0';
                    done<='0';
                    IF ( round<87 ) THEN
                        round <= round + std_logic_vector'("0000001");
               ELSE
                  round <= round;
                    END IF;
                WHEN quarter_en =>
                   mux_m0<='1';
                    m0_start<='0';
                    m1_start<='0';
                    load_all<='0';
                    serial<='0';
                    quarter_rd_start<='1';
                    done<='0';
                    round <= round;
                WHEN OTHERS =>
          END CASE;
     END PROCESS;

     PROCESS (round)
     BEGIN
          round0 <= round(0);
          round1 <= round(1);
          round2 <= round(2);
          round3 <= round(3);
          round4 <= round(4);
          round5 <= round(5);
          round6 <= round(6);
     END PROCESS;
END BEHAVIOR;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY ieee;
USE ieee.std_logic_unsigned.all;

ENTITY CONTRO IS
     PORT (addr : OUT std_logic_vector(2 DOWNTO 0);
          CLK,mem_done,quarter_done,RESET,start: IN std_logic;
          done,load_all,mux_m0,m0_start,m1_start,quarter_rd_start,serial : OUT std_logic);
END;

ARCHITECTURE BEHAVIOR OF CONTRO IS
   SIGNAL round : std_logic_vector (6 DOWNTO 0);
```

```
        COMPONENT SHELL_CONTRO
            PORT (CLK,mem_done,quarter_done,RESET,start: IN std_logic;
                done,load_all,mux_m0,m0_start,m1_start,quarter_rd_start,round0,round1,round2,
                    round3,round4,round5,round6,serial : OUT std_logic);
        END COMPONENT;
BEGIN
        addr<=round(2 DOWNTO 0);
        SHELL1_CONTRO : SHELL_CONTRO PORT MAP (CLK=>CLK,mem_done=>mem_done,
            quarter_done=>quarter_done,RESET=>RESET,start=>start,done=>done,load_all=>
            load_all,mux_m0=>mux_m0,m0_start=>m0_start,m1_start=>m1_start,quarter_rd_start=>
            quarter_rd_start,round0=>round(0),round1=>round(1),round2=>round(2),round3=>
            round(3),round4=>round(4),round5=>round(5),round6=>round(6),serial=>serial);
END BEHAVIOR;

library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;


entity quarterround is
    port ( CLK      : in       std_logic;
            d0      : in       std_logic_vector (31 downto 0);
            d1      : in       std_logic_vector (31 downto 0);
            d2      : in       std_logic_vector (31 downto 0);
            d3      : in       std_logic_vector (31 downto 0);
            RST     : in       std_logic;
            start : in      std_logic;
            done    : out      std_logic;
            z0      : inout std_logic_vector (31 downto 0);
            z1      : inout std_logic_vector (31 downto 0);
            z2      : inout std_logic_vector (31 downto 0);
            z3      : inout std_logic_vector (31 downto 0));
end quarterround;

architecture BEHAVIORAL of quarterround is
    signal mux2      : std_logic;
    signal mux4      : std_logic_vector (1 downto 0);
    signal reg_ld : std_logic_vector (3 downto 0);
    component quarter
        port ( clock                : in      std_logic;
                clear               : in      std_logic;
                d2                  : in      std_logic_vector (31 downto 0);
                reg_sel0            : in      std_logic;
                reg_sel1            : in      std_logic;
                reg_sel2            : in      std_logic;
                reg_sel3            : in      std_logic;
                mux4_sel_xorin1 : in      std_logic_vector (1 downto 0);
                mux4_sel_xorin0 : in      std_logic_vector (1 downto 0);
                mux4_sel_ain1       : in      std_logic_vector (1 downto 0);
                d3                  : in      std_logic_vector (31 downto 0);
                d1                  : in      std_logic_vector (31 downto 0);
                d0                  : in      std_logic_vector (31 downto 0);
                mux2_sel            : in      std_logic;
                z0                  : inout std_logic_vector (31 downto 0);
                z2                  : inout std_logic_vector (31 downto 0);
                z3                  : inout std_logic_vector (31 downto 0);
                z1                      : inout std_logic_vector (31 downto 0);
                mux4_sel_ain0      : in        std_logic_vector (1 downto 0));
    end component;

    component FSM_QUA
        port ( CLK              : in       std_logic;
                RESET           : in       std_logic;
                start            : in       std_logic;
                done             : out     std_logic;
                mux2_input : out     std_logic;
                mux4              : out     std_logic_vector (1 downto 0);
                reg_ld           : out     std_logic_vector (3 downto 0));
    end component;

begin
    XLXI_1 : quarter
        port map (clear=>RST,
                    clock=>CLK,
                    d0(31 downto 0)=>d0(31 downto 0),
                    d1(31 downto 0)=>d1(31 downto 0),
                    d2(31 downto 0)=>d2(31 downto 0),
                    d3(31 downto 0)=>d3(31 downto 0),
                    mux2_sel=>mux2,
                    mux4_sel_ain0(1 downto 0)=>mux4(1 downto 0),
                    mux4_sel_ain1(1 downto 0)=>mux4(1 downto 0),
                    mux4_sel_xorin0(1 downto 0)=>mux4(1 downto 0),
                    mux4_sel_xorin1(1 downto 0)=>mux4(1 downto 0),
                    reg_sel0=>reg_ld(0),
                    reg_sel1=>reg_ld(1),
                    reg_sel2=>reg_ld(2),
                    reg_sel3=>reg_ld(3),
                    z0(31 downto 0)=>z0(31 downto 0),
                    z1(31 downto 0)=>z1(31 downto 0),
                    z2(31 downto 0)=>z2(31 downto 0),
                    z3(31 downto 0)=>z3(31 downto 0));

    XLXI_2 : FSM_QUA
        port map (CLK=>CLK,
                    RESET=>RST,
                    start=>start,
                    done=>done,
                    mux2_input=>mux2,
                    mux4(1 downto 0)=>mux4(1 downto 0),
                    reg_ld(3 downto 0)=>reg_ld(3 downto 0));

end BEHAVIORAL;

library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;


entity quarter is
    port ( clear                  : in       std_logic;
            clock                  : in       std_logic;
            d0                      : in       std_logic_vector (31 downto 0);
        d1                  : in     std_logic_vector (31 downto 0);
        d2                  : in     std_logic_vector (31 downto 0);
        d3                  : in     std_logic_vector (31 downto 0);
        mux2_sel            : in     std_logic;
        mux4_sel_ain0       : in     std_logic_vector (1 downto 0);
        mux4_sel_ain1       : in     std_logic_vector (1 downto 0);
        mux4_sel_xorin0 : in    std_logic_vector (1 downto 0);
        mux4_sel_xorin1 : in    std_logic_vector (1 downto 0);
        reg_sel0            : in     std_logic;
        reg_sel1            : in     std_logic;
        reg_sel2            : in     std_logic;
        reg_sel3            : in     std_logic;
        z0                  : inout std_logic_vector (31 downto 0);
        z1                  : inout std_logic_vector (31 downto 0);
        z2                  : inout std_logic_vector (31 downto 0);
        z3                  : inout std_logic_vector (31 downto 0));
end quarter;

architecture BEHAVIORAL of quarter is
    signal ain0              : std_logic_vector (31 downto 0);
    signal ain1              : std_logic_vector (31 downto 0);
    signal reg0_in           : std_logic_vector (31 downto 0);
    signal reg1_in           : std_logic_vector (31 downto 0);
    signal reg2_in           : std_logic_vector (31 downto 0);
    signal reg3_in           : std_logic_vector (31 downto 0);
    signal r7                : std_logic_vector (31 downto 0);
    signal r9                : std_logic_vector (31 downto 0);
    signal r13               : std_logic_vector (31 downto 0);
    signal r18               : std_logic_vector (31 downto 0);
    signal sum               : std_logic_vector (31 downto 0);
    signal xor_in0           : std_logic_vector (31 downto 0);
    signal xor_in1           : std_logic_vector (31 downto 0);
    signal zi                : std_logic_vector (31 downto 0);
    component adder_predefined
        port ( A : in     std_logic_vector (31 downto 0);
               B : in     std_logic_vector (31 downto 0);
               S : out    std_logic_vector (31 downto 0));
    end component;

    component rotation7
        port ( din   : in     std_logic_vector (31 downto 0);
               dout : out     std_logic_vector (31 downto 0));
    end component;

    component rotation9
        port ( din   : in     std_logic_vector (31 downto 0);
               dout : out     std_logic_vector (31 downto 0));
    end component;

    component rotation13
        port ( din   : in     std_logic_vector (31 downto 0);
               dout : out     std_logic_vector (31 downto 0));
    end component;

    component rotation18
        port ( din   : in     std_logic_vector (31 downto 0);
                dout : out      std_logic_vector (31 downto 0));
    end component;

    component mux_2to1
        port ( s   : in       std_logic;
                i0 : in       std_logic_vector (31 downto 0);
                i1 : in       std_logic_vector (31 downto 0);
                q   : inout std_logic_vector (31 downto 0));
    end component;

    component mux_4to1
        port ( i0 : in       std_logic_vector (31 downto 0);
                i1 : in       std_logic_vector (31 downto 0);
                i2 : in       std_logic_vector (31 downto 0);
                i3 : in       std_logic_vector (31 downto 0);
                s   : in       std_logic_vector (1 downto 0);
                q   : inout std_logic_vector (31 downto 0));
    end component;

    component reg
        port ( clk : in       std_logic;
                clr : in       std_logic;
                s     : in       std_logic;
                d     : in       std_logic_vector (31 downto 0);
                q     : inout std_logic_vector (31 downto 0));
    end component;

    component xor_array
        port ( a : in       std_logic_vector (31 downto 0);
                b : in       std_logic_vector (31 downto 0);
                c : out      std_logic_vector (31 downto 0));
    end component;

begin
    adder : adder_predefined
        port map (A(31 downto 0)=>ain0(31 downto 0),
                    B(31 downto 0)=>ain1(31 downto 0),
                    S(31 downto 0)=>sum(31 downto 0));

    dr7 : rotation7
        port map (din(31 downto 0)=>sum(31 downto 0),
                    dout(31 downto 0)=>r7(31 downto 0));

    dr9 : rotation9
        port map (din(31 downto 0)=>sum(31 downto 0),
                    dout(31 downto 0)=>r9(31 downto 0));

    dr13 : rotation13
        port map (din(31 downto 0)=>sum(31 downto 0),
                    dout(31 downto 0)=>r13(31 downto 0));

    dr18 : rotation18
        port map (din(31 downto 0)=>sum(31 downto 0),
                    dout(31 downto 0)=>r18(31 downto 0));

    mux0 : mux_2to1
    port map (i0(31 downto 0)=>d0(31 downto 0),
              i1(31 downto 0)=>zi(31 downto 0),
              s=>mux2_sel,
              q(31 downto 0)=>reg0_in(31 downto 0));

mux1 : mux_2to1
    port map (i0(31 downto 0)=>d1(31 downto 0),
              i1(31 downto 0)=>zi(31 downto 0),
              s=>mux2_sel,
              q(31 downto 0)=>reg1_in(31 downto 0));

mux2 : mux_2to1
    port map (i0(31 downto 0)=>d2(31 downto 0),
              i1(31 downto 0)=>zi(31 downto 0),
              s=>mux2_sel,
              q(31 downto 0)=>reg2_in(31 downto 0));

mux3 : mux_2to1
    port map (i0(31 downto 0)=>d3(31 downto 0),
              i1(31 downto 0)=>zi(31 downto 0),
              s=>mux2_sel,
              q(31 downto 0)=>reg3_in(31 downto 0));

mux4_ain0 : mux_4to1
    port map (i0(31 downto 0)=>z0(31 downto 0),
              i1(31 downto 0)=>z1(31 downto 0),
              i2(31 downto 0)=>z2(31 downto 0),
              i3(31 downto 0)=>z3(31 downto 0),
              s(1 downto 0)=>mux4_sel_ain0(1 downto 0),
              q(31 downto 0)=>ain0(31 downto 0));

mux4_ain1 : mux_4to1
    port map (i0(31 downto 0)=>z3(31 downto 0),
              i1(31 downto 0)=>z0(31 downto 0),
              i2(31 downto 0)=>z1(31 downto 0),
              i3(31 downto 0)=>z2(31 downto 0),
              s(1 downto 0)=>mux4_sel_ain1(1 downto 0),
              q(31 downto 0)=>ain1(31 downto 0));

mux4_xor_in0 : mux_4to1
    port map (i0(31 downto 0)=>z1(31 downto 0),
              i1(31 downto 0)=>z2(31 downto 0),
              i2(31 downto 0)=>z3(31 downto 0),
              i3(31 downto 0)=>z0(31 downto 0),
              s(1 downto 0)=>mux4_sel_xorin0(1 downto 0),
              q(31 downto 0)=>xor_in0(31 downto 0));

mux4_xor_in1 : mux_4to1
    port map (i0(31 downto 0)=>r7(31 downto 0),
              i1(31 downto 0)=>r9(31 downto 0),
              i2(31 downto 0)=>r13(31 downto 0),
              i3(31 downto 0)=>r18(31 downto 0),
              s(1 downto 0)=>mux4_sel_xorin1(1 downto 0),
              q(31 downto 0)=>xor_in1(31 downto 0));

register0 : reg
```

```
        port map (clk=>clock,
                    clr=>clear,
                    d(31 downto 0)=>reg0_in(31 downto 0),
                    s=>reg_sel0,
                    q(31 downto 0)=>z0(31 downto 0));

    register1 : reg
        port map (clk=>clock,
                    clr=>clear,
                    d(31 downto 0)=>reg1_in(31 downto 0),
                    s=>reg_sel1,
                    q(31 downto 0)=>z1(31 downto 0));

    register2 : reg
        port map (clk=>clock,
                    clr=>clear,
                    d(31 downto 0)=>reg2_in(31 downto 0),
                    s=>reg_sel2,
                    q(31 downto 0)=>z2(31 downto 0));

    register3 : reg
        port map (clk=>clock,
                    clr=>clear,
                    d(31 downto 0)=>reg3_in(31 downto 0),
                    s=>reg_sel3,
                    q(31 downto 0)=>z3(31 downto 0));

    xor_gate_array : xor_array
        port map (a(31 downto 0)=>xor_in1(31 downto 0),
                    b(31 downto 0)=>xor_in0(31 downto 0),
                    c(31 downto 0)=>zi(31 downto 0));

end BEHAVIORAL;

----ASIC_basic_iterative
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity quarterround is
    port(rst, start, clk:    in std_logic;
            mux_sel, regs_sel: in std_logic_vector(3 downto 0);
            y0, y1, y2, y3: in    std_logic_vector(31 downto 0);
            z0, z1, z2, z3: out std_logic_vector(31 downto 0)
            );
end entity;

architecture rtl of quarterround is
    component xor_array
    PORT (          a: in STD_LOGIC_VECTOR (31 downto 0);
            b: in STD_LOGIC_VECTOR (31 downto 0);
            c: out STD_LOGIC_VECTOR (31 downto 0) );
    end component;

    component mux_2to1
```

```
      GENERIC (N: INTEGER :=32);
      PORT(i0, i1: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
            s: IN STD_LOGIC;
            q: INOUT STD_LOGIC_VECTOR(n-1 DOWNTO 0));
       end component;

      component reg
      GENERIC (N: INTEGER :=32);
      PORT(clk,clr,s: IN    STD_LOGIC;
            d: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            q: INOUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
      end component;

      type regcolumn is array(0 to 3) of std_logic_vector(31 downto 0);
      signal reg_in, reg_out,
                 add_out, dr, z: regcolumn;


      begin
       MUX0: mux_2to1 port map (y0, z(0), mux_sel(0), reg_in(0));
       MUX1: mux_2to1 port map (y1, z(1), mux_sel(1), reg_in(1));
       MUX2: mux_2to1 port map (y2, z(2), mux_sel(2), reg_in(2));
       MUX3: mux_2to1 port map (y3, z(3), mux_sel(3), reg_in(3));

           REGS: for i in 0 to 3 generate
             REGI: reg port map ( clk, rst, regs_sel(i),
                                           reg_in(i),reg_out(i));
             end generate;
             add_out(0)<=reg_out(0)+reg_out(3);
             add_out(1)<=reg_out(1)+reg_out(0);
             add_out(2)<=reg_out(2)+reg_out(1);
             add_out(3)<=reg_out(3)+reg_out(2);
             dr(0)<=add_out(0)(24 DOWNTO 0)&add_out(0)(31 DOWNTO 25);
             dr(1)<=add_out(1)(22 DOWNTO 0)&add_out(1)(31 DOWNTO 23);
             dr(2)<=add_out(2)(18 DOWNTO 0)&add_out(2)(31 DOWNTO 19);
             dr(3)<=add_out(3)(13 DOWNTO 0)&add_out(3)(31 DOWNTO 14);

             z1G: xor_array port map (reg_out(1), dr(0), z(1));
             z2G: xor_array port map (reg_out(2), dr(1), z(2));
             z3G: xor_array port map (reg_out(3), dr(2), z(3));
             z0G: xor_array port map (reg_out(0), dr(3), z(0));

             z0<=reg_out(0); z1<=reg_out(1);
             z2<=reg_out(2); z3<=reg_out(3);
        end rtl;

library ieee;
use ieee.std_logic_1164.all;

entity fsm is
           port (clk, rst, start: in std_logic;
                  round: in std_logic_vector(4 downto 0);
                  s: out std_logic_vector(6 downto 0);
                  ready: out std_logic);
end entity;
```

```
architecture rtl of fsm is
        type state is (idle, paral_load, z1, z2, z3, z0, done);
        signal ps, ns: state;
begin
        state_reg: process (rst, clk)
        begin

                if (rst = '1') then
                        ps <= idle;
                elsif (clk = '1' and clk'event) then
                        ps <= ns;
                end if;
        end process;

        state_transaction: process (start, ps, round)
        begin
                if start = '1' then
                      case ps is
                    when idle =>
                            ready<='0';
                            ns <= paral_load;
                    when paral_load => ns <= z1;
                    when z1 => ns <= z2;
                    when z2 => ns <= z3;
                    when z3 => ns <= z0;
                    when z0 =>
                              if (round = "10100") then ns <= done;
                              else ns <= paral_load;
                              end if;
                    when done => ready <= '1';
                    end case;
                end if;
            end process;

            --output_decode
            with ps select
            s <= "0000001" when idle,
                      "0000010" when paral_load,
                      "0000100" when z1,
                      "0001000" when z2,
                      "0010000" when z3,
                      "0100000" when z0,
                      "1000000" when done;
end rtl;
```

---- FPGA_compact

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE ieee.std_logic_unsigned.ALL;

ENTITY controller IS
        PORT (clk,clkfast, rst, start: IN STD_LOGIC;
                    cddr: INOUT STD_LOGIC_VECTOR(1 DOWNTO 0);
                    sm: INOUT STD_LOGIC_VECTOR(5 DOWNTO 0);
```

```
                addr: INOUT STD_LOGIC_VECTOR(7 DOWNTO 0);
                wren: OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
                s_reg: INOUT STD_LOGIC_VECTOR(3 DOWNTO 0);
                ready: INOUT STD_LOGIC);
END ENTITY;

ARCHITECTURE rtl OF controller IS
        COMPONENT fsm
        --ready is the signal indicating 20 quarterround funtions are done for the input
        PORT (clk, rst, start, ready, en: IN STD_LOGIC;
                s: OUT STD_LOGIC_VECTOR(10 DOWNTO 0));
        END COMPONENT;

        COMPONENT pulse_gen
            PORT ( clk,rst, trigger: IN STD_LOGIC;
                pulse: OUT STD_LOGIC);
        END COMPONENT;

        COMPONENT counter
        PORT (clk, clr: IN STD_LOGIC;
                q: INOUT STD_LOGIC_VECTOR(5 DOWNTO 0));
        END COMPONENT;

        COMPONENT table
        PORT (addr: IN STD_LOGIC_VECTOR(5 DOWNTO 0);
                output: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
        END COMPONENT;

        COMPONENT output
            PORT (start, clk: IN STD_LOGIC;
                s : IN STD_LOGIC_VECTOR(10 DOWNTO 0);
                count: IN STD_LOGIC_VECTOR(5 DOWNTO 0);
                addr_ram: IN STD_LOGIC_VECTOR(3 DOWNTO 0);

                change_s: OUT STD_LOGIC;
                cddr: INOUT STD_LOGIC_VECTOR(1 DOWNTO 0);
                sm: INOUT STD_LOGIC_VECTOR(5 DOWNTO 0);
                addr: INOUT STD_LOGIC_VECTOR(7 DOWNTO 0);
                wren: OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
                s_reg: INOUT STD_LOGIC_VECTOR(3 DOWNTO 0);
                ready,key_ready : OUT STD_LOGIC);
        END COMPONENT;

        SIGNAL s: STD_LOGIC_VECTOR(10 DOWNTO 0);
        SIGNAL addr_ram: STD_LOGIC_VECTOR(3 DOWNTO 0);
        SIGNAL count: STD_LOGIC_VECTOR(5 DOWNTO 0);
        SIGNAL change_state, trigger,
                pulse0, pulse1, pulse,change_to_add: STD_LOGIC;

        BEGIN
            pulse<=pulse0 OR pulse1;
            state_machine: fsm PORT MAP(clk, rst, start, change_to_add, change_state,s);
            pulseGen0: pulse_gen PORT MAP(clkfast, rst, s(1), pulse0);
            pulseGen1: pulse_gen PORT MAP(clkfast, rst, s(2), pulse1);
            countr: counter PORT MAP (clk, pulse, count);
            truth_table: table PORT MAP (count, addr_ram);
        output_logic: output PORT MAP (start, clkfast, s, count,
                                        addr_ram, change_state, cddr,
                                        sm, addr, wren, s_reg,
                                        change_to_add, ready);

    END rtl;

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY fsm IS
    --ready is the signal indicating 20 quarterround funtions are done for the input
    PORT (clk, rst, start, ready, en: IN STD_LOGIC;
          s: OUT STD_LOGIC_VECTOR(10 DOWNTO 0));
END ENTITY;

ARCHITECTURE rtl OF fsm IS
    TYPE state IS (idle, initialize, zc0, zc1, zc2, zc3, zr0, zr1, zr2, zr3, add);
    SIGNAL ps, ns: state;
BEGIN
    state_reg: PROCESS (rst, clk, en, ns)
    BEGIN

        IF (rst = '1') THEN
            ps <= idle;
        ELSIF (en='1') THEN
            IF (clk = '1' AND clk'event) THEN
            ps <= ns;
             END IF;
        END IF;
    END PROCESS;

    state_transaction: PROCESS (start, ready, ps)
    BEGIN
        IF start = '1' THEN
           CASE ps IS
          WHEN idle => ns <= initialize;
          WHEN initialize => ns <= zc0;
          WHEN zc0 => ns <= zc1;
          WHEN zc1 => ns <= zc2;
          WHEN zc2 => ns <= zc3;
          WHEN zc3 => ns <= zr0;
          WHEN zr0 => ns <= zr1;
          WHEN zr1 => ns <= zr2;
          WHEN zr2 => ns <= zr3;
          WHEN zr3 =>
               IF (ready='1') THEN ns <= add;
               ELSE ns <= zc0;
               END IF;
          WHEN add => ns <= initialize;
          END CASE;
        ELSE
          ns <= ps;
        END IF;
      END PROCESS;
```

```
        --output_decode
        WITH ps SELECT
        s <= "00000000001" WHEN idle,
             "00000000010" WHEN initialize,
             "00000000100" WHEN zc0,
             "00000001000" WHEN zc1,
             "00000010000" WHEN zc2,
             "00000100000" WHEN zc3,
             "00001000000" WHEN zr0,
             "00010000000" WHEN zr1,
             "00100000000" WHEN zr2,
             "01000000000" WHEN zr3,
             "10000000000" WHEN add;
END rtl;
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY pulse_gen IS
    PORT ( clk,rst, trigger: IN STD_LOGIC;
           pulse: OUT STD_LOGIC);
END ENTITY;

ARCHITECTURE rtl OF pulse_gen IS
    SIGNAL q0, qb1: STD_LOGIC;
    BEGIN
        DFF0:PROCESS(clk, rst,trigger)
        BEGIN
            IF rst='1' THEN
                q0<='0';
            ELSIF(clk'event AND clk='1')THEN
                q0<=trigger;
            END IF;
        END PROCESS;

        DFF1: PROCESS(clk,rst,q0)
        BEGIN
            IF rst='1' THEN
                qb1<='1';
            ELSIF(clk'event AND clk='1')THEN
                qb1<= NOT q0;
            END IF;
        END PROCESS;

        pulse <= q0 AND qb1;
    END rtl;
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE ieee.std_logic_unsigned.ALL;

ENTITY table IS
    PORT (addr: IN STD_LOGIC_VECTOR(5 DOWNTO 0); --64 entries
          output: OUT STD_LOGIC_VECTOR(3 DOWNTO 0)); --16*8 bits
END ENTITY;

ARCHITECTURE rtl OF table IS
    SUBTYPE WORD IS STD_LOGIC_VECTOR(3 DOWNTO 0);
```

```
      TYPE ROM IS ARRAY (0 TO 63) OF WORD;

      CONSTANT content: ROM := ( "0000", "0011", "0001", "0001",
                                 "0001", "0000", "0010", "0010",
                                 "0010", "0001", "0011", "0011",
                                 "0011", "0010", "0000", "0000",

                                 "0101", "0100", "0110", "0110",
                                 "0110", "0101", "0111", "0111",
                                 "0111", "0110", "0100", "0100",
                                 "0100", "0111", "0101", "0101",


                                 "1010", "1001", "1011", "1011",
                                 "1011", "1010", "1000", "1000",
                                 "1000", "1011", "1001", "1001",
                                 "1001", "1000", "1010", "1010",

                                 "1111", "1110", "1100", "1100",
                                 "1100", "1111", "1101", "1101",
                                 "1101", "1100", "1110", "1110",
                                 "1110", "1101", "1111", "1111"
);

      BEGIN
        output <= content(conv_integer(addr));
    END rtl;
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE ieee.std_logic_unsigned.ALL;

ENTITY output IS
      PORT (start, clk: IN STD_LOGIC;
            s : IN STD_LOGIC_VECTOR(10 DOWNTO 0);
            count: IN STD_LOGIC_VECTOR(5 DOWNTO 0);
            addr_ram: IN STD_LOGIC_VECTOR(3 DOWNTO 0);

            change_s: OUT STD_LOGIC;
            cddr: INOUT STD_LOGIC_VECTOR(1 DOWNTO 0);
            sm: INOUT STD_LOGIC_VECTOR(5 DOWNTO 0);
            addr: INOUT STD_LOGIC_VECTOR(7 DOWNTO 0);
            wren: OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
            s_reg: INOUT STD_LOGIC_VECTOR(3 DOWNTO 0);
            key_stream_en: OUT STD_LOGIC;
            ready,key_ready : OUT STD_LOGIC);
END ENTITY;

ARCHITECTURE rtl OF output IS
SIGNAL round:STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL round_no_en, dff: STD_LOGIC;

BEGIN
change_s <= (count(3)AND count(2)AND count(1)AND count(0))OR
                    (s(0) AND start) ;
                 --   OR ((s(2) or s(3)OR s(4)or s(5)OR s(6) or s(7)OR s(8) OR s(9)) AND count(1)AND
count(0));
```

```
control_signals: PROCESS (dff, s, count, addr_ram, round)--sm, cddr, addr, wren, s_reg, done
VARIABLE addr_half: STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
    CASE s IS
        WHEN "00000000001"=> --idle
            cddr <= "00";
            sm <= (OTHERS=>'0');
            addr <= "00000000";
            wren <= "00";
            ready <= '0';
        WHEN "00000000010"=> --initialize
            wren <= "11";
            addr_half:= count(3 DOWNTO 0);
            addr <= addr_half & addr_half;
            CASE count IS
                WHEN "000000"=> --constant0 from the ROM
                    cddr <= "00";
                    sm (1 DOWNTO 0) <= "00";
                WHEN "000001"=> -- key
                    sm (1 DOWNTO 0) <= "01";
                WHEN "000101"=> -- constant1
                    cddr <= "01";
                    sm (1 DOWNTO 0) <= "00";
                WHEN "000110"=> -- nonce
                    sm (1 DOWNTO 0) <= "01";
                WHEN "001010"=> -- constant2
                    cddr <= "10";
                    sm (1 DOWNTO 0) <= "00";
                WHEN "001011"=> -- key
                    sm (1 DOWNTO 0) <= "01";
                WHEN "001111"=> -- constant3
                    cddr <= "11";
                    sm (1 DOWNTO 0) <= "00";
                WHEN OTHERS=> NULL;
                END CASE;
        WHEN "10000000000"=> --add
           sm(2) <= '1'; --the operands of the adder are from RAM0 and RAM1
           sm(3) <= '1'; --short cut from RAM0
           addr <= count(3 DOWNTO 0) & count(3 DOWNTO 0);
           wren <= "00";
           --key_ready is asserted when round=10 (10 double quarterround)
            key_ready<= '1';
        WHEN OTHERS=> --hash function
             -- input of ram0 is from the datapath
             -- operands of the adder are from ram0
            ready <=    round(3)AND (NOT round(2)) AND
                                    round(1)AND (NOT round(0));
            sm(3 DOWNTO 0) <= "0010";
            wren(1) <= '0'; --ram1 cannot be written
            IF (count(1 downto 0)="11")THEN
                wren(0) <= '1'; --write back from datapath
            ELSE wren(0) <= '0';
            END IF;

            IF ((s(2)OR s(3)OR s(4)OR s(5))='1')THEN --column
              CASE addr_ram IS
            WHEN"0000" => addr(3 DOWNTO 0) <= "0000";
            WHEN"0001" => addr(3 DOWNTO 0) <= "0100";
            WHEN"0010" => addr(3 DOWNTO 0) <= "1000";
            WHEN"0011" => addr(3 DOWNTO 0) <= "1100";
            WHEN"0100" => addr(3 DOWNTO 0) <= "0001";
            WHEN"0101" => addr(3 DOWNTO 0) <= "0101";
            WHEN"0110" => addr(3 DOWNTO 0) <= "1001";
            WHEN"0111" => addr(3 DOWNTO 0) <= "1101";
            WHEN"1000" => addr(3 DOWNTO 0) <= "0010";
            WHEN"1001" => addr(3 DOWNTO 0) <= "0110";
            WHEN"1010" => addr(3 DOWNTO 0) <= "1010";
            WHEN"1011" => addr(3 DOWNTO 0) <= "1110";
            WHEN"1100" => addr(3 DOWNTO 0) <= "0011";
            WHEN"1101" => addr(3 DOWNTO 0) <= "0111";
            WHEN"1110" => addr(3 DOWNTO 0) <= "1011";
            WHEN"1111" => addr(3 DOWNTO 0) <= "1111";
            WHEN OTHERS => addr(3 DOWNTO 0) <= "0000";
            END CASE;
           ELSE    -- row
             addr(3 DOWNTO 0)<= addr_ram;
           END IF;

            CASE count(3 downto 0) IS
                  WHEN "0011"=> sm(5 downto 4)<= "00";
                  WHEN "0111"=> sm(5 downto 4)<= "01";
                  WHEN "1011"=> sm(5 downto 4)<= "10";
                  WHEN "1111"=> sm(5 downto 4)<= "11";
                  WHEN OTHERS=> NULL;
            END CASE;

      END CASE;
  END PROCESS;

s_regs: PROCESS(s, dff)
BEGIN
   CASE s IS
      WHEN "00000000001"=> --idle
         s_reg<=(OTHERS=>'0');
      WHEN "00000000010"=> --initialize
           s_reg <= "0000"; -- hold the regs
      WHEN "10000000000"=> --add
           s_reg<="0110";
           key_stream_en<= dff;
      WHEN OTHERS =>            --hash function
           s_reg(3) <= '1' ;
         CASE count(1 downto 0) IS
                WHEN "00"=>
                     IF dff = '0' THEN
                        s_reg(2 downto 0) <= "000" ;
                     ELSIF dff = '1' THEN
                        s_reg(2 downto 0) <= "001" ;
                     END IF;
                WHEN "01"=>
                     IF dff = '0' THEN
                        s_reg(2 downto 0) <= "000" ;
                     ELSIF dff = '1' THEN
```

```
                    s_reg(2 downto 0) <= "010" ;
                 END IF;
            WHEN "10"=>
                 IF dff = '0' THEN
                    s_reg(2 downto 0) <= "000" ;
                 ELSIF dff = '1' THEN
                    s_reg(2 downto 0) <= "100" ;
                 END IF;

            WHEN OTHERS=> NULL;
            END CASE;
        END CASE;
END PROCESS;


 -- increase round_no when count=63
 round_no_en<=s(9) AND count(0) AND count(1) AND count(2) AND
                count(3) AND count(4)AND count(5);
 round_number: PROCESS (s(1), round_no_en) --clear, increase
                BEGIN
                    IF s(1)='1' THEN
                        round<=(OTHERS=>'0');
                    ELSIF(round_no_en'event AND round_no_en='1') THEN
                         round <= round + 1;
                    END IF;
               END PROCESS;
 flipflop: PROCESS (s(1), clk)--dff begins to work when initialization begins
               BEGIN
                   IF s(1)='1' THEN
                       dff <= '1';
                   ELSIF (clk' event AND clk='1') THEN
                        dff <= not dff;
                   END IF;
              END PROCESS;

END rtl;
```

Appendix F: Selected Codes for Statistical Tests

```
package tests;

import generators.Ks_multiple;
import generators.PhelixException;
import tests_algorithms.BlockFrequency;
import tests_algorithms.DiscreteFourierTransform;
import tests_algorithms.Frequency;
import tests_algorithms.Runs;
import util.MyMath;
import util.Utility;
```

```
public class TestAll {
        /** It generate m data sequences.
         * The resulting sequence is tested by the four methods from the test suite
         */
        public static void main(String[] args) throws PhelixException {
            /*Etype:          0-Salsa20
             *                1-Phelix
             */
            int Etype = 0;

            /*Stype:          0-Key/Keystream Correlation Sequences
             *                1-IV/Keystream Correlation Sequences
             *                2-Frame Correlation Sequences
             *                3-Diffusion Sequences
             *                4-Keystream Sequences
             */
            int Stype=2;

            /*Ttype:          0-Frequency Test
             *                1-Block Frequency Test
             *                2-DiscreteFourierTransform Test
             *                3-Runs Test
             */
            int Ttype=2;

            int pow = 10;   // # of the generated sequences=Math.pow(2, pow)

            double[] p_value;
            double newP_value;
            double[] f = new double[81];
            int pass_No = 0;
            double pass_Frequency = 0;
            for(int i=0; i<10; i++){
                f[i]=0.0;
            }


            String[] sequence = null;
            Ks_multiple ks_gen = null;

            switch (Etype){
            case 0: System.out.println("Test Salsa20");break;
            case 1: System.out.println("Test Phelix");break;
            }

            if (Stype<4)
                ks_gen = new Ks_multiple(pow, Etype, Stype);
            else
                ks_gen = new Ks_multiple(pow, Etype, 0);
            switch (Stype){
              case 0:
                  System.out.println("test key/Keystream Correlation Sequences");
                  sequence = ks_gen.ks_k;
                  break;
              case 1:
```

```
            System.out.println("test IV/Keystream Correlation Sequences");
            sequence = ks_gen.ks_IV;
            break;
        case 2:
            System.out.println("test Frame Correlation Sequences");
            sequence = ks_gen.frame;
            break;
        case 3:
            System.out.println("test Diffusion Correlation Sequences");
            sequence = ks_gen.diffusion;
            break;
        case 4:
            System.out.println("test Keystream Sequences");
            sequence = ks_gen.ks_str;
            break;
    }
    int sL = sequence.length;
    p_value = new double[sL];

    switch (Ttype){
        case 0:
            System.out.println("Frequency Test: ");
            for(int i=0; i<sL; i++){
                p_value[i] = new Frequency(sequence[i]).pvalue;

                Utility.group(p_value[i], f);
                pass_No = Utility.pass(p_value[i], pass_No);
            }
            break;
        case 1:
            System.out.println("BlockFrequency Test: ");
            for(int i=0; i<sequence.length; i++){
                p_value[i] = new BlockFrequency(20, sequence[i]).pvalue;
                Utility.group(p_value[i], f);
                pass_No = Utility.pass(p_value[i], pass_No);
            }
            break;
        case 2:
            System.out.println("DFT Test: ");
            for(int i=0; i<sequence.length; i++){
                p_value[i] = new DiscreteFourierTransform(sequence[i]).pvalue;
                Utility.group(p_value[i], f);
                pass_No = Utility.pass(p_value[i], pass_No);
            }
            break;
        case 3:
            System.out.println("Runs Test: ");
            for(int i=0; i<sequence.length; i++){
                p_value[i] = new Runs(sequence[i]).pvalue;
                Utility.group(p_value[i], f);
                pass_No = Utility.pass(p_value[i], pass_No);
            }
            break;
    }

    double x;
```

```
                double chi=0;
                for (int i=0; i<10; i++){
                        f[i] = f[i]/sL;
                        x = f[i] - 0.1;
                        chi += x * x / 0.1;
                }
                newP_value = MyMath.igamc(4.5, chi/2);

                java.text.DecimalFormat    df5    =    new    java.text.DecimalFormat("##0.00000");
                java.text.DecimalFormat    df6    =    new    java.text.DecimalFormat("##0.000000");
                System.out.println("chi = "+df5.format(chi));
                System.out.println("newp_value = "+df5.format(newP_value));
                pass_Frequency = (double)pass_No/sL;
                System.out.println("proportion = "+df6.format(pass_Frequency));

        }

}
ackage tests_algorithms;

import util.MyMath;

public class DiscreteFourierTransform {

        public double pvalue;

        public DiscreteFourierTransform(String str){

                int          n = str.length();
                int          count = 0;
                double       upperBound = MyMath.sqrt(3*n);
                double       percentile, N_l, N_o, d;
                double[] X_real = new double[n];
                double[] X_imag = new double[n];
                double[] S = new double[n];       //significance

                for( int i=0; i<n; i++ )
                        X_real[i] = (double) MyMath.checkStringMinus(str)[i];
                MyMath.dft(X_real);
                for( int i=0; i<n/2; i++ ){
                        X_real[i] = MyMath.gr[i];
//                      System.out.println("gr"+i+":"+X_real[i]);
                        X_imag[i] = MyMath.gi[i];
                        S[i] = Math.sqrt( Math.pow(X_real[i],2) + Math.pow(X_imag[i],2) );
//                      System.out.println("S"+i+":"+S[i]);
                }
//              System.out.println("upperBound"+":"+upperBound);
                for(int i=0; i<n/2; i++ )
                        if ( S[i] < upperBound )
                                count++;
                percentile = (double)count/(n/2)*100;
//              System.out.println("percentile"+":"+percentile);
                N_l = (double) count;             /* number of peaks less than h = sqrt(3*n) */
//              System.out.println("N1"+":"+N_l);
                N_o = (double) 0.95*n/2.;
                //d = (N_l - N_o)/sqrt(n/2.*0.95*0.05);
```

```
		d = (N_1 - N_o)/Math.sqrt(n/4.0*0.95*0.05);
//		System.out.println("d"+":"+d);
		pvalue = MyMath.erfc(Math.abs(d)/Math.sqrt(2.));

	}


}
package generators;

import java.math.BigInteger;
import java.util.Arrays;

import util.Utility;

public class Ks_multiple {
	public int[][] key;
	public int[][] IV;
	public int[][] ks;
	public String[] ks_str;
	public String[] ks_k;
	public String[] ks_IV;
	public String[] frame;
	public String[] diffusion;
	public int ksLen_Salsa = 16;	//in words(32 bit)
	public int ksLen_Phelix = 16;	//in words(32 bit)


	public void encryptPhelix(byte[]REF_KEY, byte[]REF_IV, byte[]REF_PTXT, byte[]ctxt, int[][]ks, int
i ) throws PhelixException{
		Phelix phx;
		phx = new Phelix();
	phx.init();
		phx.setupKey(REF_KEY, 0, REF_KEY.length * 8, Phelix.PHELIX_MAC_SIZE);
		phx.setupNonce(REF_IV, 0);
		ctxt = makeOutputBuffer(REF_PTXT.length, 0);
		phx.encryptBytes(REF_PTXT, 0, ctxt, 0, REF_PTXT.length);
		ks[i] = phx.ks;
	}

	public void encryptSalsa(byte[]REF_KEY, byte[]REF_IV, int[][]ks, int i){
		Salsa20 salsa = new Salsa20(REF_KEY, REF_IV);
		key[i]= salsa.kW;
		salsa.keystream_gen();
		ks[i] = salsa.ks;
	}
	public String tonbitString(int[] data){
		String temp = null;
		String str = null;
		for (int i=0; i<data.length; i++) {
			temp = Integer.toBinaryString(data[i]);
			while(temp.length()<32)
			temp = "0".concat(temp);
			if (i==0)
				str = temp;
			else
```

```
                    str = str.concat(temp);
            }
            return str;
    }

    public void to32bitStrings(int[][] data, String[] str){
            for (int i=0; i<data.length; i++){
                    str[i] = tonbitString(data[i]);
            }

    }

    public static byte[] makeOutputBuffer(int nLen, int nExtraLen)
     {
            byte[] result = new byte[nLen + nExtraLen];
            Arrays.fill(result, (byte)0xcc);
            return result;
     }
    /**
     *
     * @param pow: the power of 2
     * @param generator: 0 indicates Salsa20, 1 indicates Phelix
     * @param type: 0-keystream or Key/Keystream Correlation Sequence.
     *                  1-IV/Keystream Correlation Sequence.
     *                  2-Frame Correlation Sequences.
     *                  3-Diffusion Sequence
     * @throws PhelixException
     */
    public Ks_multiple(int pow, int generator, int type) throws PhelixException{
            int[][] temp = null;
            int m = (int) Math.pow(2, pow);     //# of the generated sequences

            if (generator == 1){ //Phelix
                    this.ks= new int[m][ksLen_Phelix];
                    ks_str = new String[m];
                    byte[] REF_KEY = new byte[60];      //256-bit key
                    byte[] REF_IV = new byte[44]; //128-bit IV
                    byte[] REF_PTXT = new byte[ksLen_Phelix*4] ;
                    byte[] ctxt = null;
                    this.key = new int[m][10];
                    java.util.Random rKey = new java.util.Random(),
                                    rIV=new java.util.Random(),
                                    rPTXT =new java.util.Random();
                    rPTXT.nextBytes(REF_PTXT); //plaintext fixed

                    if (type == 0){
                    ks_k = new String[m];
                    temp = new int[m][key[0].length];
                            rIV.nextBytes(REF_IV); //IV fixed
                            for (int i=0; i<m; i++){
                                    rKey.nextBytes(REF_KEY );

                                    int j=0;
                                    for (int k=0;k<8;k++){
                                     j=k*4;
                                     this.key[i][k]=
                        Utility.byteToInt(REF_KEY[j+3],      REF_KEY[j+2],      REF_KEY[j+1],
REF_KEY[j]);
                }

                    encryptPhelix(REF_KEY, REF_IV, REF_PTXT, ctxt, this.ks, i );

                    for(j=0; j<key[i].length; j++)
                     temp[i][j] =ks[i][j]^key[i][j]; //keystream XOR key
                   }
              to32bitStrings(this.ks, this.ks_str);
              to32bitStrings(temp, this.ks_k);
              System.out.println("Number of Sequences = "+ks_k.length+", " +
                     "Sequence Length = "+ks_k[0].length());
          }
         if (type == 1){
          ks_IV = new String[m];
          IV = new int[m][6];
          temp = new int[m][IV[0].length];
                rKey.nextBytes(REF_KEY); //key fixed
                for (int i=0; i<m; i++){
                    rIV.nextBytes(REF_IV );

                    int j=0;
                    for (int k=0;k<4;k++){
                     j=k*4;
                     this.IV[i][k]=
                          Utility.byteToInt(REF_IV[j+3], REF_IV[j+2], REF_IV[j+1], REF_IV[j]);
                }

                    encryptPhelix(REF_KEY, REF_IV, REF_PTXT, ctxt, this.ks, i );
                    for(j=0; j<IV[i].length; j++)
                     temp[i][j] = ks[i][j]^IV[i][j]; //keystream XOR IV
                }
              to32bitStrings(this.ks, this.ks_str);
              to32bitStrings(temp, this.ks_IV);
            System.out.println("Number of Sequences = "+ks_IV.length+", " +
                          "Sequence Length = "+ks_IV[0].length());
          }
         if (type == 2){
          ks_IV = new String[m];
          IV = new int[m][6];
          frame = new String[512]; //the arbitrary length of the frame is 512
                rKey.nextBytes(REF_KEY); //key fixed
              // This procedure is repeated pow(2,10) times with incremented values of IV.
                for (int i=0; i<m; i++){
                 if (i<256){
                      REF_IV[0] = (byte)i;
                      REF_IV[73] = (byte)0;
                }

                 else {
                      REF_IV[73] = (byte)i;
                      REF_IV[0] = (byte)0;
                }
                 for (int iv_index=2; iv_index<REF_IV.length; iv_index++){
                      REF_IV[iv_index]=0;
```

```
        }

            encryptPhelix(REF_KEY, REF_IV, REF_PTXT, ctxt, this.ks, i );
          }
      to32bitStrings(this.ks, this.ks_str);


      for(int fi=0; fi<512; fi++){ //the arbitrary length of the frame is 512
       for (int mi=0; mi<m; mi++){
             if (mi==0)
                   frame[fi] = String.valueOf(ks_str[mi].charAt(0));
             else
                   frame[fi]     =     frame[fi].concat(String.valueOf(ks_str[mi].charAt(fi)));

         }
      }
    }
  if (type == 3){
   byte[] newKey, newIV;
   int sLen = 256 + 128; //k+v sequences
   diffusion = new String[sLen];
   byte ei = 1;
   int ei_p = 0;   //the position of the bit that will change
   int byte_p=0; //the position of the byte that will change
   int[] ks_old = new int[44],
          ks_new = new int[44];
   String ks_old_str = null;
   String ks_new_str = null;
   BigInteger b1, b2;
   int[][] diff_int = new int[sLen][44];
   for (int i=0; i<sLen; i++){
         for(int j=0; j<16; j++)
         diff_int[i][j] = 0;
   }

   for (int i=0; i<m; i++){
         /*
          * Random key and IV values are chosen.
          * Using this key and IV, a keystream of length L=ksLen_Phelix *4 bits is generated
          */
         rKey.nextBytes(REF_KEY); //32 bytes
         rIV.nextBytes(REF_IV);     //16 bytes
         encryptPhelix(REF_KEY, REF_IV, REF_PTXT, ctxt, this.ks, i );
         ks_old= ks[i];
         ks_old_str = tonbitString(ks_old);

         /*
          * By changing each bit of key and IV, new keystreams are generated.
          * These keystreams are XORed with the original keystream.
          * Each obtained value is added with the value of diffusion[j]
          */
         for (int j=0; j<256; j++){
               ei_p = j % 8;
               byte_p = j/8;
               newKey = REF_KEY;
               newKey[byte_p] ^= (int) Math.pow(2, ei_p);
```

```
                        encryptPhelix(newKey, REF_IV, REF_PTXT, ctxt, this.ks, i );
                        ks_new = ks[i];
                            ks_new_str = tonbitString(ks_new);

                        b1 = new BigInteger(ks_old_str, 2);
                        b2 = new BigInteger(ks_new_str, 2);

                        diffusion[j] = b1.add(b2).toString(2);
                        while (diffusion[j].length()<512)
                            diffusion[j] = "0".concat(diffusion[j]);
                    }
                    for (int j=0; j<128; j++){
                        ei_p = j % 8;
                        byte_p = j/8;
                        newIV = REF_IV;
                        newIV[byte_p] ^= (int) Math.pow(2, ei_p);

                        encryptPhelix(REF_KEY, newIV, REF_PTXT, ctxt, this.ks, i );

                        ks_new= ks[i];
                        ks_new_str = tonbitString(ks_new);
                        b1 = new BigInteger(ks_old_str, 2);
                        b2 = new BigInteger(ks_new_str, 2);

                        diffusion[256+j] = b1.add(b2).toString(2);
                        while (diffusion[256+j].length()<512)
                            diffusion[256+j] = "0".concat(diffusion[256+j]);
                    }
                }



                }


            }

        if (generator == 0){ //Salsa20
        this.ks= new int[m][ksLen_Salsa];
        ks_str = new String[m];
        byte[] REF_KEY = new byte[60];      //256-bit key
            byte[] REF_IV = new byte[44]; //128bit IV
            java.util.Random rKey = new java.util.Random(),
                                rIV = new java.util.Random();
            rKey.nextBytes(REF_KEY); //fixed
            rIV.nextBytes(REF_IV); //fixed

            this.key = new int[m][10];

            if (type == 0){
              ks_k = new String[m];
              temp = new int[m][key[0].length];
                  rIV.nextBytes(REF_IV); //IV fixed
                for (int i=0; i<m; i++){
                        rKey.nextBytes(REF_KEY );

                        int j=0;
                        for (int k=0;k<8;k++){
                         j=k*4;
                         this.key[i][k]=
                              Utility.byteToInt(REF_KEY[j+3],      REF_KEY[j+2],      REF_KEY[j+1],
REF_KEY[j]);
                      }

                        encryptSalsa(REF_KEY, REF_IV, this.ks, i );

                        for(j=0; j<key[i].length; j++)
                         temp[i][j] =ks[i][j]^key[i][j]; //keystream XOR key
                       }
                  to32bitStrings(this.ks, this.ks_str);
                  to32bitStrings(temp, this.ks_k);
                }
               if (type == 1){
                ks_IV = new String[m];
                IV = new int[m][6];
                temp = new int[m][IV[0].length];
                      rKey.nextBytes(REF_KEY); //key fixed
                  for (int i=0; i<m; i++){
                        rIV.nextBytes(REF_IV );

                        int j=0;
                        for (int k=0;k<4;k++){
                         j=k*4;
                         this.IV[i][k]=
                              Utility.byteToInt(REF_IV[j+3], REF_IV[j+2], REF_IV[j+1], REF_IV[j]);
                      }

                        encryptSalsa(REF_KEY, REF_IV, this.ks, i );
                        for(j=0; j<IV[i].length; j++)
                         temp[i][j] = ks[i][j]^IV[i][j]; //keystream XOR IV
                      }

                  to32bitStrings(this.ks, this.ks_str);
                  to32bitStrings(temp, this.ks_IV);
                }
               if (type == 2){
                int l=ksLen_Salsa*32;
                ks_IV = new String[m];
                IV = new int[m][6];
                frame = new String[512];
                      rKey.nextBytes(REF_KEY); //key fixed
                  // This procedure is repeated pow(2,10) times with incremented values of IV.
                  for (int i=0; i<m; i++){
                    if (i<256){
                         REF_IV[0] = (byte)i;
                         REF_IV[73] = (byte)0;
                    }

                    else {
                         REF_IV[73] = (byte)i;
                REF_IV[0] = (byte)0;
            }
            for (int iv_index=2; iv_index<REF_IV.length; iv_index++){
                REF_IV[iv_index]=0;
            }

            encryptSalsa(REF_KEY, REF_IV, this.ks, i );
              }
          to32bitStrings(this.ks, this.ks_str);
          for(int fi=0; fi<l; fi++){ //the arbitrary length of the frame is l
          for (int mi=0; mi<m; mi++){
              if (mi==0)
                          frame[fi] = String.valueOf(ks_str[mi].charAt(0));
                      else
                          frame[fi]   =   frame[fi].concat(String.valueOf(ks_str[mi].charAt(fi)));

                  }
     }
     }
   if (type == 3){
    byte[] newKey, newIV;
    int sLen = 256 + 128; //k+v sequences
    diffusion = new String[sLen];
    byte ei = 1;
    int ei_p = 0;    //the position of the bit that will change
    int byte_p=0; //the position of the byte that will change
    int[] ks_old = new int[44],
              ks_new = new int[44];
    String ks_old_str = null;
    String ks_new_str = null;
    BigInteger b1, b2;
    int[][] diff_int = new int[sLen][44];
    for (int i=0; i<sLen; i++){
            for(int j=0; j<16; j++)
            diff_int[i][j] = 0;
    }

    for (int i=0; i<m; i++){
            /*
             * Random key and IV values are chosen.
             * Using this key and IV, a keystream of length L=ksLen_Salsa *4 bits is generated
             */
            rKey.nextBytes(REF_KEY); //32 bytes
            rIV.nextBytes(REF_IV);      //16 bytes
            encryptSalsa(REF_KEY, REF_IV, this.ks, i );
            ks_old= ks[i];
            ks_old_str = tonbitString(ks_old);

            /*
             * By changing each bit of key and IV, new keystreams are generated.
             * These keystreams are XORed with the original keystream.
             * Each obtained value is added with the value of diffusion[j]
             */
            for (int j=0; j<256; j++){
                ei_p = j % 8;
                byte_p = j/8;
```

```
                                        newKey = REF_KEY;
                                        newKey[byte_p] ^= (int) Math.pow(2, ei_p);
                                        encryptSalsa(newKey, REF_IV, this.ks, i );
                                        ks_new = ks[i];
                                            ks_new_str = tonbitString(ks_new);
                                        b1 = new BigInteger(ks_old_str, 2);
                                        b2 = new BigInteger(ks_new_str, 2);

                                        diffusion[j] = b1.add(b2).toString(2);
                                        while (diffusion[j].length()<512)
                                            diffusion[j] = "0".concat(diffusion[j]);
                                    }
                                    for (int j=0; j<128; j++){
                                        ei_p = j % 8;
                                        byte_p = j/8;
                                        newIV = REF_IV;
                                        newIV[byte_p] ^= (int) Math.pow(2, ei_p);

                                        encryptSalsa(REF_KEY, newIV, this.ks, i );
                                        ks_new= ks[i];
                                        ks_new_str = tonbitString(ks_new);
                                        b1 = new BigInteger(ks_old_str, 2);
                                        b2 = new BigInteger(ks_new_str, 2);
                                        diffusion[256+j] = b1.add(b2).toString(2);
                                        while (diffusion[256+j].length()<512)
                                            diffusion[256+j] = "0".concat(diffusion[256+j]);
                                    }
                                }
                            }
                    }
                }
            }
```

Appendix G: A Test Example to Illustrate the Distribution of P-values

```
Test Salsa20
Frequency Test:
1024 independent sequences (1024 keys and 1024 nonce are generated randomly),
each is of 512 bits, generate 1024 P-values:

0.53197 0.90052 0.31731 0.70766 0.70766 0.90052 0.80259 0.45325
0.31731 0.61708 0.31731 0.80259 0.70766 0.90052 0.26059 0.21130
0.31731 0.10416 0.10416 0.31731 0.61708 1.00000 0.06079 0.53197
0.31731 0.31731 0.26059 0.31731 0.61708 0.90052 0.21130 0.90052
0.61708 0.06079 0.90052 0.80259 0.31731 0.26059 1.00000 0.53197
0.53197 0.90052 0.80259 0.38157 0.90052 0.26059 0.53197 0.21130
0.90052 0.90052 0.53197 0.21130 0.10416 0.31731 1.00000 0.61708
0.80259 0.70766 0.90052 0.90052 0.21130 0.38157 0.61708 0.31731
0.26059 0.70766 0.21130 0.31731 0.45325 0.38157 0.90052 0.90052
0.10416 0.31731 0.10416 0.31731 0.70766 0.26059 0.61708 0.53197
```

```
0.70766 1.00000 0.13361 0.45325 1.00000 0.06079 0.00866 0.16913
0.53197 0.31731 0.70766 0.21130 0.26059 0.61708 0.13361 0.13361
0.70766 0.70766 0.53197 0.70766 0.80259 0.26059 0.53197 0.26059
0.61708 0.53197 0.80259 0.80259 0.70766 0.10416 0.26059 0.70766
0.04550 0.70766 0.31731 0.53197 0.70766 0.80259 0.53197 0.80259
0.13361 0.16913 1.00000 0.31731 0.61708 0.13361 0.53197 0.80259
1.00000 0.21130 0.70766 0.90052 1.00000 0.61708 0.70766 0.61708
1.00000 0.26059 0.31731 0.61708 0.53197 0.70766 0.70766 0.61708
0.80259 0.31731 0.10416 0.61708 1.00000 0.70766 0.80259 0.70766
0.38157 0.80259 0.53197 0.90052 0.45325 0.70766 0.45325 0.70766
0.13361 0.45325 0.08012 0.53197 0.70766 0.31731 0.90052 0.53197
0.38157 0.53197 0.70766 0.45325 0.90052 0.70766 0.45325 0.61708
0.26059 0.10416 0.26059 0.70766 0.31731 0.31731 0.53197 0.70766
0.61708 0.90052 0.53197 0.90052 0.26059 0.21130 0.21130 0.53197
0.31731 0.45325 0.45325 0.61708 0.26059 0.16913 0.53197 0.61708
0.90052 0.45325 0.80259 1.00000 0.26059 0.61708 0.70766 0.80259
0.26059 0.26059 0.80259 0.80259 0.70766 0.80259 0.38157 0.61708
0.80259 0.38157 0.08012 0.80259 0.08012 0.38157 0.00866 0.04550
0.53197 0.70766 0.26059 0.31731 0.06079 0.10416 0.61708 0.70766
0.26059 0.80259 1.00000 0.80259 1.00000 0.26059 0.45325 0.45325
0.61708 0.53197 0.80259 0.90052 0.00115 0.61708 0.03359 0.61708
0.53197 0.38157 0.31731 0.90052 0.31731 0.04550 0.38157 0.90052
0.31731 0.45325 0.70766 0.61708 0.53197 0.70766 0.45325 1.00000
0.08012 0.45325 0.45325 0.61708 0.70766 1.00000 1.00000 0.53197
0.31731 0.53197 0.53197 0.26059 0.70766 1.00000 0.10416 0.38157
0.31731 0.31731 0.13361 0.90052 0.02445 0.90052 0.10416 1.00000
0.21130 0.90052 0.08012 1.00000 0.38157 0.06079 0.53197 0.21130
0.10416 0.90052 0.26059 0.90052 0.90052 1.00000 0.13361 0.31731
0.10416 0.31731 0.21130 0.61708 0.70766 1.00000 0.16913 0.03359
0.38157 0.53197 0.21130 0.26059 0.70766 0.70766 0.16913 0.01755
0.45325 0.21130 0.80259 1.00000 0.13361 0.90052 0.38157 0.38157
0.21130 0.16913 0.21130 0.80259 0.90052 0.13361 0.90052 0.70766
0.90052 0.90052 0.26059 0.26059 1.00000 0.08012 0.26059 0.26059
0.13361 0.38157 0.70766 1.00000 0.61708 1.00000 0.45325 0.61708
0.38157 0.90052 1.00000 0.70766 1.00000 0.38157 0.10416 0.45325
0.70766 0.70766 0.31731 0.21130 0.80259 0.70766 0.53197 0.13361
0.53197 0.03359 0.06079 0.80259 0.70766 0.38157 0.08012 0.61708
0.06079 0.53197 0.61708 0.31731 0.16913 0.80259 0.70766 0.16913
0.13361 0.26059 0.45325 0.90052 0.80259 0.04550 0.16913 0.38157
0.61708 0.06079 0.70766 0.38157 0.70766 0.38157 0.21130 0.70766
0.80259 0.00596 0.10416 0.04550 0.08012 0.45325 0.21130 1.00000
0.16913 0.16913 0.13361 0.26059 0.31731 0.70766 0.06079 0.80259
0.21130 0.08012 0.70766 0.38157 0.80259 0.70766 0.21130 1.00000
0.80259 1.00000 0.61708 0.61708 0.61708 0.45325 0.21130 0.38157
0.90052 0.70766 0.45325 0.45325 0.53197 0.02445 0.38157 0.70766
0.61708 0.16913 0.70766 0.26059 0.13361 0.61708 0.90052 0.31731
0.70766 0.45325 0.90052 1.00000 0.61708 1.00000 0.61708 0.90052
1.00000 1.00000 0.53197 0.53197 0.53197 0.53197 0.90052 0.31731
0.10416 0.80259 0.70766 0.45325 0.45325 0.53197 0.02445 0.38157
0.31731 0.45325 1.00000 0.45325 0.04550 0.26059 0.70766 0.61708
0.61708 0.61708 0.90052 0.53197 0.53197 0.70766 0.31731 0.26059
0.26059 0.03359 0.80259 0.01755 1.00000 0.80259 0.08012 0.90052
0.53197 0.21130 0.80259 1.00000 0.45325 0.21130 0.90052 0.70766
0.31731 0.61708 0.31731 0.45325 0.26059 0.70766 0.38157 0.38157
0.90052 0.10416 0.06079 0.26059 0.80259 0.53197 0.26059 0.26059
0.13361 0.45325 0.53197 0.70766 0.61708 0.90052 0.90052 0.90052
0.70766 0.45325 0.10416 0.61708 0.53197 0.80259 0.80259 0.31731
```

```
0.13361 0.80259 0.06079 0.90052 0.90052 0.70766 0.70766 0.16913
0.16913 0.70766 0.26059 0.90052 0.21130 0.21130 0.80259 0.45325
0.53197 0.61708 0.01755 0.26059 0.61708 0.80259 0.38157 0.38157
0.80259 0.53197 0.80259 0.26059 0.21130 0.45325 0.53197 0.61708
0.53197 1.00000 0.80259 0.80259 0.70766 0.04550 0.53197 0.38157
0.90052 0.31731 0.08012 0.16913 0.80259 0.53197 1.00000 0.70766
0.45325 0.01755 0.90052 0.70766 1.00000 0.10416 0.45325 0.80259
0.53197 0.38157 0.90052 0.13361 0.31731 0.80259 0.16913 0.26059
0.31731 0.01755 0.90052 0.53197 0.90052 0.02445 0.70766 0.90052
0.70766 0.02445 0.10416 0.61708 0.13361 0.01242 0.80259 0.31731
0.61708 0.00404 0.61708 0.80259 0.70766 1.00000 0.16913 0.45325
0.90052 0.38157 0.31731 0.80259 0.26059 0.80259 0.31731 0.16913
0.61708 0.53197 0.13361 0.80259 0.16913 0.26059 0.16913 0.61708
0.90052 0.70766 0.90052 0.80259 0.61708 0.80259 0.61708 0.10416
0.38157 0.90052 0.53197 0.38157 0.38157 0.26059 0.21130 0.38157
0.38157 0.26059 0.90052 0.70766 0.26059 0.31731 0.80259 0.61708
0.80259 0.90052 0.90052 0.21130 0.16913 0.61708 0.90052 0.31731
0.13361 0.38157 0.70766 0.61708 0.61708 0.80259 0.90052 0.61708
0.21130 0.38157 0.90052 0.06079 1.00000 0.16913 0.16913 1.00000
0.00596 0.06079 0.70766 0.26059 0.38157 0.45325 0.26059 0.00178
0.61708 0.53197 0.13361 1.00000 0.45325 0.10416 0.80259 0.80259
0.70766 0.26059 0.38157 0.80259 0.31731 0.61708 0.04550 0.61708
0.00404 0.70766 0.70766 0.10416 0.13361 0.13361 0.70766 0.13361
0.80259 0.61708 0.26059 0.45325 0.38157 0.61708 0.53197 1.00000
0.70766 0.53197 0.61708 0.45325 0.38157 0.90052 0.90052 0.45325
0.03359 0.10416 0.31731 0.08012 0.53197 0.70766 0.26059 0.61708
0.45325 1.00000 0.61708 0.16913 0.26059 0.45325 0.13361 0.45325
0.53197 0.53197 0.21130 0.70766 0.45325 0.53197 0.53197 0.53197
1.00000 1.00000 0.61708 0.45325 0.90052 0.31731 1.00000 0.53197
1.00000 0.31731 0.10416 0.70766 0.45325 0.26059 0.61708 0.38157
0.90052 0.08012 0.38157 0.90052 0.53197 0.10416 0.80259 0.61708
0.80259 0.61708 1.00000 0.31731 0.21130 0.70766 1.00000 0.45325
1.00000 0.80259 0.31731 0.45325 0.38157 0.70766 0.16913 0.53197
0.61708 0.26059 0.70766 0.80259 0.04550 0.70766 1.00000 0.70766
0.80259 0.61708 1.00000 0.13361 0.90052 0.31731 0.16913 0.80259
1.00000 0.21130 0.80259 0.01755 0.04550 0.90052 0.38157 0.61708
0.70766 0.31731 0.61708 0.90052 0.61708 0.53197 1.00000 0.26059
0.70766 0.45325 0.61708 0.31731 1.00000 0.38157 0.00596 0.80259
0.80259 0.21130 0.90052 0.70766 0.01755 1.00000 0.61708 0.70766
1.00000 0.26059 0.10416 0.90052 0.61708 0.10416 0.70766 0.90052
0.13361 0.80259 0.26059 0.38157 0.10416 0.08012 0.16913 0.21130
0.13361 0.53197 0.90052 0.45325 0.04550 0.31731 0.31731 0.21130
0.45325 0.06079 0.31731 0.08012 0.10416 0.53197 0.80259 0.90052
0.53197 0.53197 0.61708 0.38157 0.16913 0.31731 0.31731 0.13361
0.38157 0.26059 0.53197 1.00000 0.21130 0.80259 0.13361 0.13361
0.45325 0.61708 0.31731 0.26059 0.53197 0.90052 0.61708 0.90052
0.38157 0.04550 0.00404 0.38157 0.53197 0.45325 0.90052 0.90052
0.80259 0.70766 0.10416 0.53197 0.06079 0.31731 0.31731 0.90052
0.80259 0.53197 0.10416 0.38157 0.38157 0.45325 1.00000 0.53197
0.90052 0.21130 0.53197 0.61708 0.16913 0.45325 1.00000 0.06079
0.10416 0.10416 0.53197 0.31731 0.45325 0.53197 0.61708 0.13361
0.45325 0.08012 0.90052 0.08012 0.31731 0.61708 0.70766 0.13361
0.45325 0.70766 0.38157 0.61708 0.70766 0.13361 1.00000 0.80259
0.26059 0.90052 0.13361 0.38157 0.26059 0.26059 0.16913 0.21130
0.90052 0.21130 0.31731 0.31731 0.45325 0.08012 0.31731 0.80259
0.13361 0.61708 0.61708 0.70766 0.45325 0.80259 0.70766 0.10416
0.53197 0.31731 0.26059 0.01755 0.45325 1.00000 0.01242 0.08012
```

```
0.70766 0.08012 0.00866 0.26059 1.00000 0.45325 0.13361 0.80259
1.00000 0.16913 0.61708 0.90052 0.16913 0.31731 0.70766 0.26059
1.00000 0.61708 0.06079 0.38157 0.31731 0.70766 0.61708 0.31731
0.80259 1.00000 0.61708 0.21130 0.90052 0.02445 0.70766 0.53197
```