Cryptographic Primitives On Reconfigurable Platforms

Tsoi Kuen Hung

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of

Master of Philosophy

in

Computer Science & Engineering

Abstract

There are increasing demands for cryptographic systems due to the rapid adoption of electronic commerce and personal privacy concerns. Hardware based cryptographic systems offer improved speed, lower power consumption, smaller footprint and perhaps higher security over purely software based systems. Field Programmable Gate Array (FPGA) technology offers a good compromise between the speed of VLSI based implementations and the short development times and adaptability of software systems.

This study illustrates that FPGAs are suitable for cryptographic systems by implementing several cryptographic primitives. In particular, high performance FPGA-based implementations of secret key, public key, key search and random number generation systems were developed. The study also evaluates different architecture and system parameters which will affect the performance of the designs.

The secret key algorithm implemented was the IDEA block cipher and a deeply pipelined architecture was employed to achieve a throughput of 592Mbps. A variable radix systolic Montgomery multiplier was developed to speed up implementations of the RSA public key algorithm, offering an efficient way to estimate the performance and area tradeoffs of a long integer multiplier by varying the radix. In order to demonstrate the ability of FPGAs for cryptanalysis, an RC4 key search engine was developed which can search a 40-bit key within 2 days and achieves performance which is 58 times faster than a 1.5GHz Intel Pentium 4 machine. Finally, an area optimized random number generator using the Blum Blum Shub algorithm was implemented. This 1024-bit BBS RNG can generate a secure random sequence using less than 3% of a XCV1000E FPGA chip.

Acknowledgments

I want to thank my supervisor Prof. LEONG, Heng Wai Philip for his help and guidances through my master studies. I would also want to thank Prof. LEE, Kin Hong and Prof. Wei Keh-wei Victor for their suggestions and to be my thesis marker.

Thanks also for Mr. Cheung Yu Hoi for his help in the IDEA project and Mr. Norris Leong for his helpful guidance in tools and circuit designing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Cryptosystems are important.

The aim of cryptography is to secure information so that only the intended parties can read the data. Cryptosystems had been developed for centuries. The advance of computer technologies and popularity of personal computers provides a large base on which cryptographic applications are installed. The recent popularity of the Internet and e-commerce have made strong demands on cryptography. Cryptosystems today are all around our lives including banking systems using 3DES, identification systems using PKI (Public Key Infrastructure), entertainment systems using encrypted storages and even systems in electronic car locks. Developments in cryptography have been growing faster then ever before due to increased research.

Hardware cryptographic platforms are helpful.

Many cryptographic algorithms are based on specialized arithmetic computations such as finite field arithmetic. For clients that only perform cryptographic computations occasionally, the central processing unit (CPU) in a PC is sufficient. However the work load on a server that will handle thousands of requests per second many be unacceptably large. In addition, clients which have very limited computing resources, such as smartcards, mobile phones and handheld computers may not have sufficient computing power. Special hardware cryptosystems can offer higher performance than conventional CPUs. In addition, cryptosystems implemented in software may have lower security than tamper proof hardware devices [And01].

FPGAs are suitable for building hardware cryptosystems.

The tradition way of building cryptographic hardware is using application specific integrated circuit (ASIC) technology. This methodology has many disadvantages including high small volume cost, long design to product time, difficulties in debugging and not able to adapt new changes after the system is built. A new way to solve these problem is to build the design on FPGA platforms. FPGA chips provide sufficient logic and storage elements on which complex algorithms can be built. The reconfigurable characteristic makes it easier to adapt new cryptographic protocols even after the hardware is installed. One aim of this research work is to demonstrate that FPGA platforms are suitable for hardware cryptosystems. It is also shown that FPGA systems are good for cryptanalysis applications.

Architecture differences affect performance.

For the same algorithm, different architectures can be applied to achieve different design objectives. One of the most important issues are the cost-performance tradeoffs. Cost in hardware design can be interpreted in different ways including logic area, memory storage, power consumption, etc. Another commonly encountered problem is the parallel and serial tradeoff. One can build a deeply pipeline system which use many cycles under a fast clock or a massively parallel system that can process multiple data in one cycle. Even for the same design, different parameters such as the radix used can dramatically change the system performance.

In this work, different ways to implement cryptographic algorithms using flexible architectures will be explored.

1.2 Objectives

The main objective of this research work was to develop primitive building blocks for FPGA based hardware cryptosystems. The details research aim are:

- Implement different cryptographic algorithms including public key, block and stream ciphers and compare their performance with hardware and software based systems. This is used to support the argument that building a cryptosystem completely on an FPGA platform is possible.
- Explore different system architectures and evaluate the impact on performance.
- Provide experimental results so that a designer can choose system parameters (e.g. radix, area, throughput) which are most suitable for the given application.

1.3 Contributions

In this work, a set of primitive components for building cryptosystems on hardware platforms were implemented and evaluated. The primitives include an IDEA block cipher; a long integer multiplier core for the efficient implementation of RSA public key cryptographic algorithms; an RC4 stream cipher and also a random number generator. In the implementations, different architectures were explored and various optimization methods were used.

The main contributions of this dissertation are as follows:

- A deeply pipelined IDEA block cipher was designed [MvOV01]. The IDEA core offers 592Mbps throughput under 50MHz clock rate. The number of rounds instantiated can be varied to meet different area constraints without violating the 8.5 rounds minimum requirement in the algorithm. The upper bound of the speed is also examized. This was developed in collaboration with Mr. O.Y.H. Cheung and my duty was to design the pipeline states and control. This implementation of IDEA was, at the time of publication, the fastest reported implementation to date in FPGA technology.

- A variable radix systolic Montgomery modular multiplier was developed [Mon85]. The performance tradeoff for different radixes could thus be easily examined. Results for different radixes were measured and provide a fast way for designers to select a design given the required resources and performance.
- An RC4 key search engine was developed for cryptanalysis purposes [RSA00]. Its architecture was one of the first to exploit the massive memory bandwidth in an FPGA for cryptanalysis applications. The RC4 key search engine developed in this work is the fastest reported implementation in any technology and 58 times faster than a 1.5GHz Intel Pentium 4 CPU.
- A compact and secure random number generator using the BBS (Blum Blum Shub) algorithm [LMM86] was developed for cryptographic applications. For a 1024-bit modulo, this design consumed less than 3% of an XCV1000E chip. The results passed various RNG tests including the NIST RNG test suite [A. 01] and the Diehard test [Mar02].

1.4 Thesis Organization

In Chapter 2, an introduction to cryptographic algorithms and the FPGA design flow are presented. Also, a review of related work on cryptographic hardware is given. Chapter 3 presents implementations of the IDEA block cipher which use deep pipelining technique and Chapter 4 presents an implementation of a variable radix systolic Montgomery multiplier which can be used in RSA cryptosystems. A design of a massively parallel RC4 engine is described in Chapter 5. Chapter 6 shows an implementation of random number generator using a free oscillator and the BBS algorithm. The results of the above designs are presented and evaluated in Chapter 7. Finally, the conclusion and directions for future research are presented in Chapter 8.

Chapter 2

Background and Review

2.1 Introduction

To construct primitive building blocks for a hardware cryptosystem, the cryptographic algorithms must be studied. Also, we need to be familiar with the hardware platform, i.e. FPGA systems in our research, to achieve the best performance from them. To facilitate our studies and evaluate our results, related work on cryptographic algorithms and hardware cryptosystems are reviewed in this chapter.

This chapter is organized as follows. Section 2.2 presented the architecture of common cyrptosystems. The types of cryptographic algorithms are also explained in this section. In section 2.3, the applications of cryptography are presented. The structure and characteristics of FPGA platforms are explained in section 2.4. The last section presented the reviewed related work.

2.2 Cryptographic Algorithms

The major concern of cryptography is to secure information from being intelligible to whom it is not intended. A general view of a cryptosystem for communications is shown in Figure 2.1. Some common terms used in cryptography are:

Encryption transformation of data into a form that is unreadable without some appropriate knowledge.

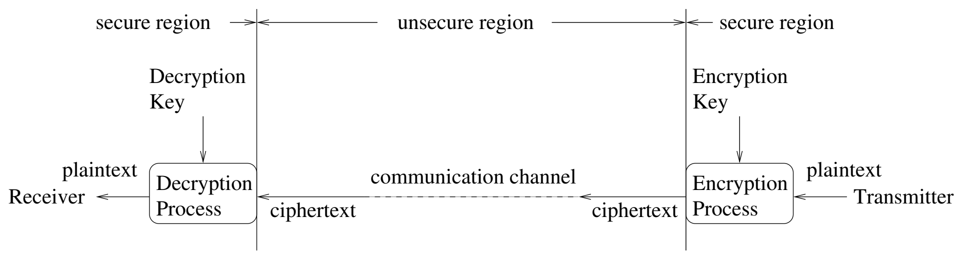


Figure 2.1: Encrypting and decryption processes.

Decryption reverse of encryption; transformation of encrypted data to the original data with the assistance of some appropriate knowledge.

Cipher process, either in form of a software program or hardware circuit to perform encryption.

Plaintext data to be encrypted.

Ciphertext data after being encrypted.

Key secret information used during encryption and decryption.

Various algorithms have been proposed for cryptographic systems and these can be divided into two major classes: *public key* algorithms and *secret key* algorithms. In secret key algorithms, all keys are kept secret and shared by the parties involved. In public key algorithms, the are two keys involved. One is the *public key* which is made public. The other is the *private key* which is known only by the person that can decrypt the message.

Secret Key Algorithms

Secret key algorithms are also referred to as symmetric key algorithms. In these algorithms, the decryption key can be generated directly from, or is exactly the encryption key, so both keys should be kept secret.

Secret key algorithms can be further classified into two subclasses: *block ciphers* and *stream ciphers*. The block ciphers accept a data block of fixed size as input in each iteration and produce an output block of the same size. It is a bijective function (one-to-one mapping) from the input plaintext block to the output ciphertext block. Stream ciphers consider the input as a stream of bytes and produce an output byte stream by combining the input with the generated *key stream*. In stream ciphers, the output of a fixed input depends on the input value, the key value as well as the time the input enters the cipher. That means two identical and adjacent input blocks will be encrypted to different output blocks by a stream cipher.

Common block cipher algorithms include: DES (Data Encryption Standard) [MvOV01], AES (Advanced Encryption Standard), RC2 (Rivest's Cipher 2) [RSA00], RC5 (Rivest's Cipher 5) [MvOV01], IDEA (International Data Encryption Algorithm) [MvOV01], Secure And Fast Encryption Routine (SAFER) [MvOV01], Blowfish [Sch93] and CAST-128 (Carlisle Adams and Stafford Tavares) [AT93]. Examples of stream ciphers include: RC4 (Rivest's Cipher 4) [RSA00], Software-optimized Encryption Algorithm (SEAL) [MvOV01] and VRA (Venkatesan, Rajagopalan and Aiello) [ARV95]. In fact, there are many other stream ciphers based on the Linear Feedback Shift Registers (LFSR) algorithm [MvOV01]. The processing speed of a stream cipher is usually faster than that of a block cipher.

There are several modes under which block ciphers can operate. The most commonly used modes are the Electronic Codebook (ECB) mode and the Cipher Block Chaining (CBC) mode [Sch96]. In ECB mode, the block ciphers are used directly and the one-to-one mapping from plaintext block to ciphertext block is maintained. In CBC mode, the output ciphertext block is fed back to the input port and XORed with the next plaintext block to be encrypted. By doing so, the encryption of a block is dependent upon the previous block and the one-to-one mapping characteristic is eliminated, improving security.

A disadvantage of secret key algorithms is that the parties performing encryption and decryption must somehow exchange keys. That means a secure channel must be available for the key exchange to take place before the secure channel for data can be established.

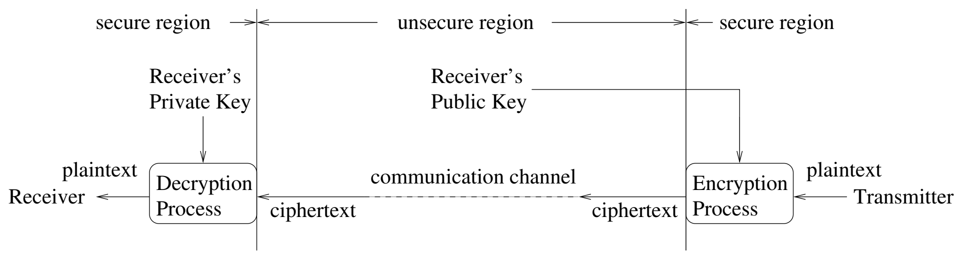


Figure 2.2: Public key cryptography.

Public Key Algorithms

Public key algorithm was first introduced by Whitfield Diffie and Martin Hellman [DH76] in 1976. Figure 2.2 illustrates the idea behind public key cryptography. The difference with Figure 2.1 is that the public key, which is used as the encryption key here, is publicly available. The message encrypted using a public key can only be decrypted by the corresponding private key. So anyone can send an encrypted message but only the person with private key can decrypt it. This property can be used to solve the key exchange problem associated with secret key algorithms. The security of public key system depends on the difficulty of deriving the private key from the public key.

Examples of public key algorithm include Diffie-Hellman [DH76], RSA [RSA78], ElGamal [ElG85], and Merkle-Hellman knapsack [MH78].

In public key systems, the public key of a receiver is shared between sender and receiver. Since this information is available to everyone, it is not necessary to have some trusted means of key distribution before secure data communication provided that the public keys are associated with their owners in a trusted manner. The disadvantage of public key algorithms is that they require more computation to achieve similar security compared with secret key algorithms.

2.3 Cryptographic Applications

Cryptographic algorithms are used in a wild range of applications; including the SSL (Secure Sockets Layer) [Net02] developed by Netscape Communications Co., SSH (Secure Shell) [TTTM02], IPSec (IP Security Protocol) [Req02c], Kerberos [Req02a] and PGP (Pretty Good Privacy) [Req02b]. Besides networking applications, cryptographic algorithms are also applied to data storage. UNIX system provide system tools to encrypt files [Ope02], word processors such as Microsoft Word [Mic02] and Adobe Acrobat [Ado02] integrate the encryption/decryption functions internally. Also, many data compressing tools such as Zip [BK02] and RAR [RAR02] provide encryption. On static storages such as DVDs, cryptographic algorithms are used to protect copyright by preventing illegal copying.

The following example illustrates the typical usage of cryptographic algorithms on networking environment. Client C wants to download a file from server S through an insecure network. The client should obtain the server's public key in advance. Before the file can be accessed, the client sends the server its request and public key encrypted using the server's public key. Since only the server has the private key to decrypt this request, this will authenticate the server's identity. Then the server will send a session key to the client, encrypted by client's public key. After the client gets the session key, a secure communication channel is established for client authentication and file transferring using either a block or stream cipher.

Note that public key cryptography, which is computationally expensive, is used for the key exchange and involves a small amount of data. A secret key algorithm is used for the potentially large amount of data involved in the file transfer. This scheme reduces the amount of computation required by the server, thus improving its overall efficiency without sacrificing security.

In this research, 4 cryptographic primitives were developed. By integrating these primitives in a single design, the tasks in the above example can be implemented in a single chip hardware cryptosystem. The Montgomery multiplier can be

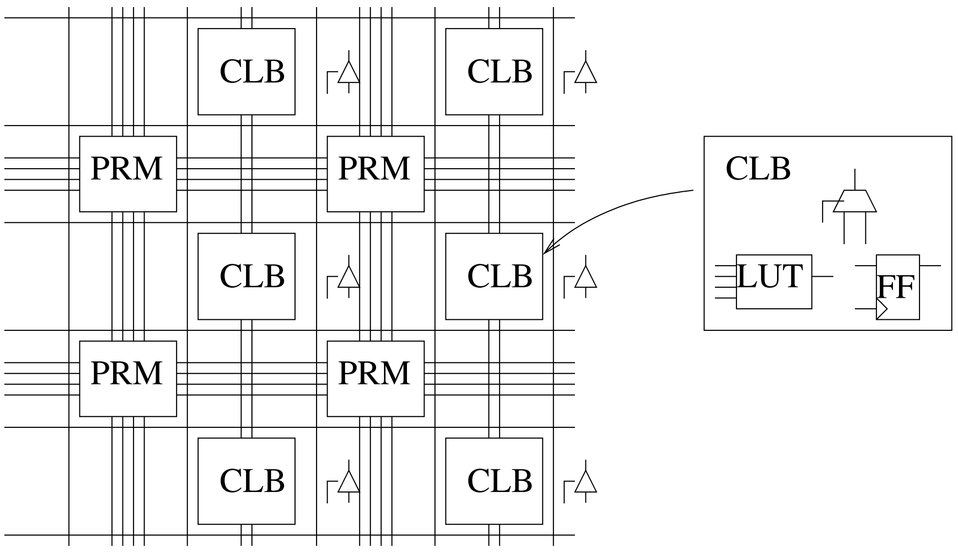


Figure 2.3: FPGA structure.

used to perform the RSA public key authentication while the BBS random number generator can be used to generate the session key for the RC4 or IDEA secret key ciphers. Actually, the designs can be used in many scenarios ranging from high-end server side cryptographic accelerators to low power consumption handheld devices.

2.4 Modern Reconfigurable Platforms

The experiments in this research were implemented on Field Programmable Gate Arrays (FPGA). FPGA is VLSI chip with some special features. The structure of an FPGA is a 2-D array of Configurable Logic Blocks (CLB) surrounded by connection wires. There are some primitives such as lookup tables (LUT) and flip-flops (FF) inside the CLB. The functions of these primitives and connections between them can be configured for different designs. Programmable routing matrices (PRM), implemented in static RAMs, are used to connect the I/O ports of the CLBs. A general structure outline of FPGAs is shown in Figure 2.3.

The advantages of FPGA designs over traditional VLSI designs are:

- Fast design to product time and chips can be reused for different designs.
- Easy simulation and debugging. Software simulator and debugger provide efficient methods of finding bugs and estimating of performance.
- It is possible to use the same FPGA hardware platform for many different cryptographic protocols. This make the design flexible and extensible.
- Low cost prototyping for early designs which are subjected to be changed.
- Design can be upgraded after deployment without hardware replacement.

Today's advanced FPGA chips also offer a lot special components such as large memory blocks (BlockRAM) and fast carry chains between adjacent logic blocks [Xil02a]. Dedicated multipliers and clock distribution lines can also be found in some designs. Due to the overheads associated with providing programmable logic, FPGA designs usually have lower clock rate and lower logic density than traditional VLSI designs using the same technology.

The FPGA design flow is shown in Figure 2.4. The design entry can be either schematic capture or synthesis via a Hardware Description Language (HDL). The schematic flow is more intuitive for small designs while the HDL flow provides an efficient way to implement and manage large and complex designs.

In the synthesis approach, a netlist is generated describing the logic functions and their interconnections. The functions are then mapped to the logic primitives of the target FPGA platform. The placement of logic primitives and routing of connections are altered to find an optimized solution which will meet the constraints stated by the designer. The implementation process generates a bitstream representing the configuration of the FPGA, which can be download to the chip.

There are many cryptographic protocols that implementing all of them in a single FPGA chip is not feasible. Once the applications agree on some protocols, they seldom change. The dynamic reconfiguration capability of the FPGAs are suitable for such scenarios. An example is secure network communications. Designers can

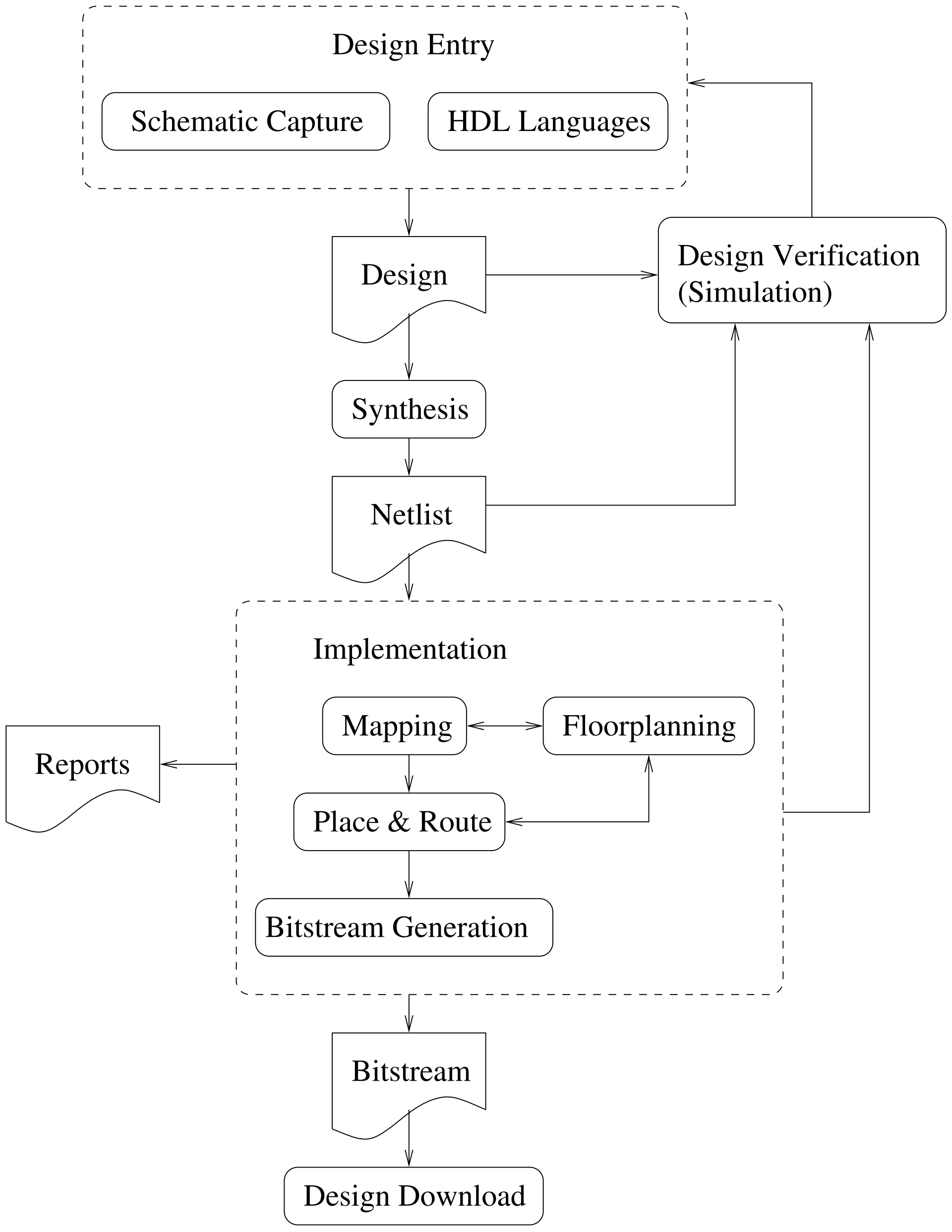


Figure 2.4: FPGA design flow.

	FPGA	Microprocessor
Unit Price	high	medium
Power Consumption	low	high
Operation Frequency	low	high
Memory Bandwidth	high	low
Hardware Parallelism	high	low
Customized for Application	yes	no
Design Time	high	low
Commondity Item	no	yes

Table 2.1: Cost/performance tradeoffs between FPGA and microprocessor.

implement all kinds of cryptographic protocols in different designs targeting the same hardware (the FPGA chip). After the software determines which protocol will be used, the bit stream of the corresponding design will be downloaded to the FPGA. By doing so, a limited hardware resource can serve a wide range of requirements.

Table 2.1 summarizes the cost/performance tradeoffs of FPGA comparing to microprocessor designs. It shows that, at the expense of lower clock rate and higher price, FPGA designs can achieve higher performance than the microprocessor counterparts through higher degrees of parallelism and customization.

2.5 Review of Related Work

2.5.1 Montgomery Multiplier

The Montgomery method [Mon85] is used in all high performance hardware and software modular multipliers. Montgomery multiplier implementations are reviewed. In 1993, Peter Kornerup [Kor93] proposed an algorithm for computing $x^e \ mod \ m$ using as high-radix redundant number system. The algorithm required $\lceil \frac{log_2 m+2}{k} \rceil + 1$ cycles per multiplication in radix 2^k. This algorithm was adapted in our design. In the same year, M. Shand and J. Vuillemin [SV93] form Paris Research Laboratory

(PRL) presented a fast FPGA based implementation of RSA system with 1Mbps throughput for 521-bit keys. This design was implemented on the PAM (Programable Active Memory) system with a matrix of 16 FPGA and the modulus used was hardwired in the design.

A VLSI implementation with lookup table quotient estimation by Che-Han Wu *et al.* [WSW$^+$99] showed that higher radix implementations could achieve speed improvement at the expense of hardware overhead. $2n(\frac{n}{rk} + 4)(k + 1)$ cycles are required for the radix-2^r algorithm where n is the size of modulus and k is the size of partitioned multiplier. The highest radix evaluated in that work was radix-16 which used 10% more logic than radix-2 system based on COMPASS cell library. In 1997, Colin D. Walter [Wal97] showed that the product $Time \times Area$ should be independent of the choice of radix for the best implementations of repeated addition.

There has been a lot of research on systolic Montgomery multipliers. Colin D. Walter [Wal93] proposed a radix-2 system with $2n + 2$ cycles latency for an n-bit multiplication. This design used a two dimensional array of systolic cells. For a 500-bit RSA design, this system was estimated to use about 4×10^6 gates. Peter Kornerup presented another linear systolic Montgomery multiplier [Kor94] which had a similar latency. Our implementation of variable radix Montgomery multiplier was based on this structure. The estimate throughput of this design is 100kbps under a 100MHz clock. In 2000, Wei-Chang Tsai *et al.* [TSW00] introduced two new systolic architectures for modular multiplication. Simulation using 0.35 μm CMOS technology shows that these 3-D systolic arrays had faster computing speed. The double-layer system consuming 240k transistors achieved 244kbps for 1024-bit RSA and the non-interlaced system consuming 209k transistors achieved 241kbps throughput.

Besides the one presented by Shand, other implementations of Montgomery multiplier based on FPGA platforms were reviewed. T. Blum and C. Paar [BP99] implemented a Montgomery exponentiation unit in systolic array architecture. Radix-4, 8, 16 were implemented and Chinese remainder theorem was used for decryption. An 1024-bit RSA decryption, on a Xilinx XC4000 series, requires 10.18ms, 12.41ms and 12.52ms respectively for radix-4, 8, 16. A. Tiountchik and E. Trichina [TT00] designed a 132-bit radix-2 linear systolic Montgomery multiplier on an Xilinx XC6000 FPGA chip. For this design, at least 4 XC6000 FPGAs are required for a 512-bit key with estimated bit rate 800Kbps. M. K. Hani *et al.* [HTSH00] presented a complete RSA system using Walter's systolic structure on Altera FLEX10KE FPGA system. The core performance of this design was not reported.

2.5.2 IDEA Cipher

The block cipher we consider in this work is the IDEA cipher. Although IDEA involves only simple 16-bit operations, software implementations of this algorithm still cannot offer the encryption rate required for on-line encryption in high-speed networks. Ascom's implementation of IDEA (Ascom are the holders of the patent on the IDEA algorithm) achieves 0.37×10^6 encryptions per seconds, or an equivalent encryption rate of 23.53Mbps, on an Intel Pentium II 450MHz machine. Implementation of IDEA using the Intel MMX multimedia instructions was proposed by Helger [Lip98] and achieves 0.51×10^6 encryption per seconds or a equivalent encryption rate 32.9Mbps, on an Intel Pentium II 233MHz machine. Our optimized software implementation running on a Sun Enterprise E4500 machine with twelve 400MHz Ultra-IIi processor, performs 2.30×10^6 encryptions per second or an equivalent encryption rate of 147.13Mbps, still cannot be applied to applications such as encryption for 155Mbps Asynchronous Transfer Mode (ATM) networks or Gigabit Ethernet.

Hardware implementations offer significant speed improvements over software implementations by exploiting parallelism among operators. In addition, they are likely to be cheaper, having lower power consumption and smaller footprint than a high speed software implementation. A design of an IDEA processor which achieves 528 Mb/sec on four XC4020XL devices was proposed by Mencer et. al. [MMF98]. The first VLSI implementation of IDEA was developed and verified by Bonnenberg et. al. in 1992 using a 1.5 μm CMOS technology [BCF+91]. This implementation had an encryption rate of 44Mbps. In 1994, VINCI, a 177Mbps VLSI implementation of the IDEA algorithm in 1.2 μm CMOS technology, was reported by Curiger et. al. [CBZ+93, ZCB+94]. A 355Mbps implementation in 0.8 μm technology of IDEA was reported in 1995 by Wolter et. al. [WMSL95], followed by a 424Mbps single chip implementation of 0.7 μm technology by Salomao et. al. [SAF98] was reported. In 2000, Leong et. al. proposed a 500Mbps bit-serial implementation of IDEA on an Xilinx Virtex XCV300-6 FPGA which is scalable on larger devices [LCTL00]. Later, Goldstein et. al reported an implementation on the PipeRench FPGA which achieves 1013Mbps [GSB+00]. A commercial implementation of IDEA called the IDEACrypt Kernel developed by Ascom achieves 720Mbps [Asc99b] at 0.25 μm technology. The implementation derived from the IDEACrypt Kernel, called the IDEACrypt Coprocessor, has a throughput of 300Mbps [Asc99a].

2.5.3 RC4 Key Search

There have been two previously reported FPGA based RC4 key search machines. In 1996, Goldberg and Wagner proposed an RC4 search engine using an Altera RIPP10 board which had 8 FLEX8000 chips and four static RAM chips [GW96]. Their design could perform 4 parallel searches and each unit required 1286 cycles per key. Kundarewich *et al.* proposed a key search engine using a single Altera EPF10K20 complex programmable logic device (CPLD). In their implementation, each search unit required 1304 cycles per key and 5 parallel searches could be made at 10MHz [KWH99].

2.5.4 Secure Random Number Generator

Many VLSI based RNGs (Random Number Generator) have been developed and evaluated. The randomness of the generators may be based on electronic noise [PC00], thermal noise [ZH01], oscillator noise [RRK98] or radioactive decay [hot02]. Most of these implementations include an amplified noise source and digital scrambling logic.

Real random number generators based on chaotic systems, provided a very compact structure on standard CMOS designs. In 2001, Toni Stojanovski *et al.* [sPK01] implemented a chaos-based RNG in a 0.8 μm standard COMS chip utilizing switched current techniques. The estimate output bit rate of this design is 1 Mbps. Also on 0.8 μm CMOS process, Andrea Gerosa *et al.* [GBP01] implemented a RNG based on a chaotic system. Their design with a pipelined ADC (analog-to-digital converter) occupied $2.2mm^2$ silicon area.

There are many methods to generate pseudo random sequences. In 1986, Wolfram [Wol86] proposed a method to generate random numbers by connected cellular automatas (CA). In the method, output of a CA is a function of the current outputs of nearby CAs. This method is very suitable for hardware implementation where concurrent operations are easy to achieved. P.D. Hortensius *et al.* [HMC89] proposed a VLSI implementation of 1-D cellular automata in parallel structure. The 30-bit hybrid CA design was about 2.1 times larger than a 30-bit LFSR (linear feedback shift register) RNG while offering better randomness and faster clock rate due to the nearest neighbor wiring. While selecting a suitable connection scheme and automata function is not trivial in higher dimension designs for more randomness.

As many cryptographic accelerating hardware are FPGA based, it is more desirable to have FPGA PRNG (Pseudo Random Number Generator) modules instead. Barry Shackleord *et al.* presented RNGs based on neighborhood-of-four cellular automata [STCS01]. The design made use of the 4-input LUTs in Xilinx FPGA to fully utilize the hardware and can generate 64-bit random numbers at frequency as high as 230MHz. Another FPGA implementation of PRNG was introduced by Robert K. Watkins *et al.* in 2001 [WIF01]. Their design used a Genetic Algorithm (GA) to generate a set of PRNGs. FIPS-140 was used as fitness function in the evolution. This design, implemented on XESS XSV800 Virtex prototyping board, relied on the reconfiguration ability at run time. The final product of the evolution is a PRNG.

It is not possible to prove a sequence is random. Some basic tests were introduced by Knuth [Knu81]. A compact and preliminary test suite was defined in FIPS-140 by the National Institute of Standards and Technology (NIST). NIST proposed a more comprehensive random and pseudo random number generator test suite for cryptographic applications in 2001 [A. 01]. Another well known RNG test suite is the Diehard test developed by Marsaglia [Mar02]. This is widely considered the most stringent RNG test.

2.6 Summary

In this chapter, the algorithms and architectures commonly used in cryptosystems has been introduced. The applications of cryptography in various fields were presented. Also, the characteristics and design flow of modern FPGA platforms were explained. Related work on Montgomery multipliers, IDEA block ciphers, RC4 key searching engines and secure random number generators were reviewed.

Chapter 3

The IDEA Cipher

3.1 Introduction

The proposed Encryption Standard (PES) is a block cipher introduced by Lai and Massay [LM90] in 1990. It was then improved by the Lai, Massay and Murphy [LMM91] in 1991. This version, with stronger security against differential analysis and truncated differentials [HL94, Knu95, Bor97], was called the Improved PES (IPES). IPES was renamed to be the International Data Encryption Algorithm (IDEA) in 1992. Claims have been made that the algorithm is the most secure block encryption algorithm in the public domain [Sch96]. Except for weak key attacks, the current best attack is by brute force on the 128-bit key space [Sch96].

In this chapter, an IDEA cipher implementation on an FPGA platform is described. With a single core, this design can achieve a throughput of 592Mbps encryption rate using a 50MHz operating frequency. The design is heavily pipelined to maximize the throughput. A method for pre-computing keys is used in this design to save logic resources. It can be used as a hardware accelerator in a cryptosystem such as for Secure Shell (ssh) data transfer and Virtual Private Networks (VPN).

This chapter is organized as follows. In Section 3.2 the IDEA algorithm as well as algorithms for multiplication modulo $2^{16} + 1$ are described. The implementation of IDEA cipher and interface are presented in Section 3.3. A summary is given Section 3.4.

3.2 The IDEA Algorithm

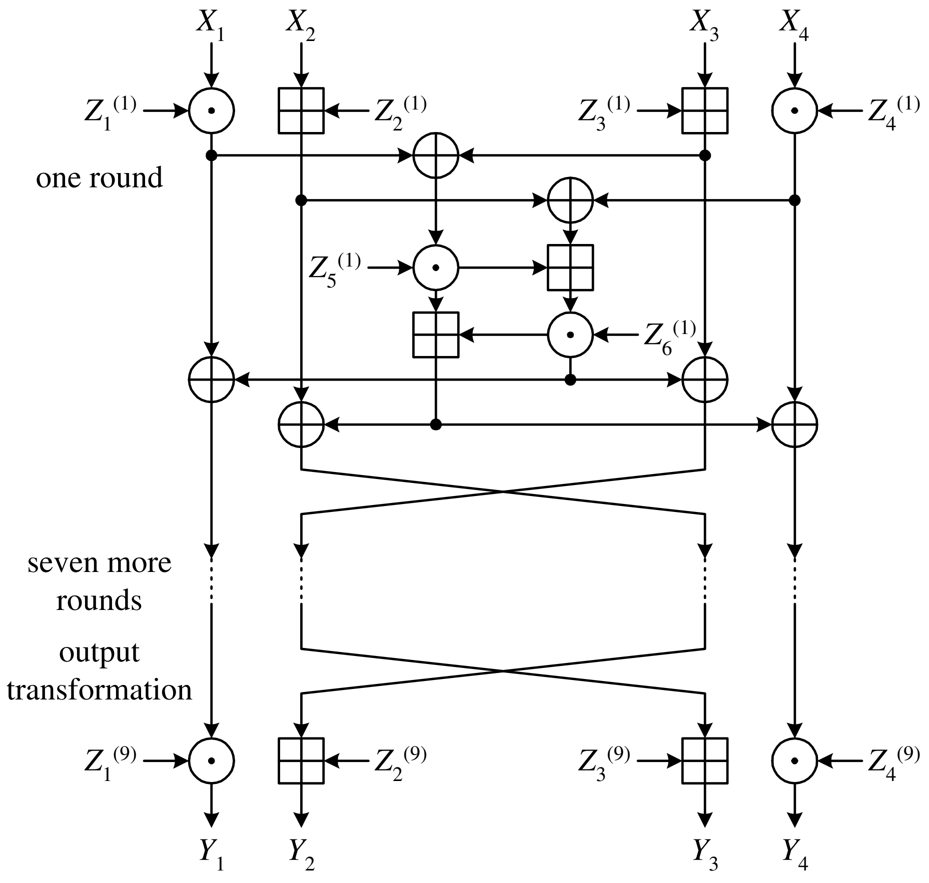
IDEA is a secret-key block cipher. The keys for both encryption and decryption must be kept secret from unauthorized persons. Since the two keys are symmetric, one can divide the decryption key from the encryption one or vice versa. The size of the key is fixed to be 128 bits and the size of the data block which can be handled in one encryption/decryption process is fixed to 64 bits. All data operations in the IDEA cipher are in 16-bit unsigned integers. When processing data which is not a integer multiple of 64-bit block, padding is required.

The security of IDEA algorithm is based on the mixing of three different kinds of algebraic operations: XOR, addition and modular multiplication. This section will explain the top structure of the cipher followed by the key scheduling algorithm and the S-box ($mod\ 2^{16} + 1$) algorithm.

3.2.1 Cipher Data Path

The IDEA block cipher shown in Figure 3.1 is based on a Feistel structure [Nyb96]. There are 8 identical rounds and an output transformation block in the original IDEA data path. The output transformation, as shown in Figure 3.1, is actually the upper half of a round with some inputs interchanged. We will reference this as half round in the rest of the text. These iterative rounds are used to make differential attacks more difficult.

The 64-bit input plaintext, *X*, is divided into four 16-bit sub-blocks, X_1 to X_4. After encryption, the four sub-blocks, Y_1 to Y_4 are concatenated to the 64-bit ciphertext. For every full round, six 16-bit subkeys, Z_1 to Z_6 are used. The half round block only use 4 subkeys. A key block is used to store the 128-bit input key. The subscript in the sub-key is the order which it is extracted from the key block. The superscript of the sub-key is the round number in which it is used. For example, Z_5^1 is the fifth sub-key used in round 1. The decryption process shares the same data path with different subkeys.



⊕ bitwise XOR of 16-bit sub-blocks

⊞ addition modulo 2^{16} of 16-bit integers

⊙ multiplication modulo 2^{16}+1 of 16-bit integers with the zero sub-block corresponding to 2^{16}

Figure 3.1: Block diagram of the IDEA algorithm.

For the IDEA algorithm in Electronic Codebook mode (ECB) [Sch96], there is no loop in the data dependency graph which implies that a deep pipeline technique can be applied.

3.2.2 S-Box: Multiplication Modulo $2^{16}+1$

The IDEA algorithm uses $mod\ 2^{16}+1$ multiplication as the main mixing operation. It maps from the domain $Z_{2^{16}}$ to $Z^*_{2^{16}+1}$. In the mapping, $0 \in Z_{2^{16}}$ is mapped to $2^{16} \in Z^*_{2^{16}+1}$. By doing so, the cardinality of $Z^*_{2^{16}+1}$ is still 16, making it the same as the other two operations. The original algorithm requires a domain transformation before and after the $\odot$ operation. Since this operation is the most computational intensive one in the IDEA algorithm, one of the design considerations was to optimize it. Many methods has been developed to speed up this calculation [CBK91]. In this design, we adapted the method proposed by Meier and Zimmerman [MZ91] which uses a modulo 2^n adders with bit-pair recoding. This algorithm is explained in the following pseudo code.

```
uint16 mulmod(uint16 x, uint16 y) {
   uint16 xd, yd, th, tl;
   uint32 t;
   xd = (x - 1) & 0xFFFF;
   yd = (y - 1) & 0xFFFF;
   t = (uint32) xd * yd + xd + yd + 1;
   tl = t & 0xFFFF;
   th = t >> 16;
   return (tl - th) + (tl <= th);
}
```

This algorithm requires a total of five additions and subtractions, one 16-bit multiplication and one comparison. However, in IDEA one of the operands of a modular multiplication operation is always a subkey, so the second subtraction can be eliminated if the associated subkeys are pre-decremented.

3.2.3 Key Schedule

In the key schedule, 52 subkeys are generated from a 128-bit input key. The subkeys are formed by rotating the input key. The key scheduling process is:

- Order the 52 subkeys as $Z_1^{(1)}, \ldots, Z_6^{(1)}$, $Z_1^{(2)}$, …, $Z_6^{(2)}$, …, $Z_1^{(8)}$, …, $Z_6^{(8)}$, $Z_1^{(9)}$, …, $Z_4^{(9)}$.
- Partition the 128-bit input key into eight 16-bit blocks. Assign them to the first 8 subkeys, $Z_1^{(1)}$ to $Z_2^{(2)}$, directly.
- Rotate the input key left by 25 bits to form a new key block. Another 8 subkeys can then be generated.
- Repeat the rotation process whenever the subkeys in the current key block is used up.

The decryption subkeys $Z'^{(r)}_i$ can be computed from the encryption subkeys with reference to Table 3.1.

3.3 FPGA-based IDEA Implementation

3.3.1 Multiplication Modulo $2^{16} + 1$

Modular multiplication operations dominate the computation time in IDEA algorithm. A careful and optimized design in this part can improve the complete design significantly. Figure 3.2 shows the structure of modulo multiplication operations using the algorithm decrypted in Section 3.2.2.

The adders in level 2 and the subtracters in level 4 are both implemented as 16-bit adders with carry input. If we subtract 1 from every subkey in the pre-compute

	$r=1$	$2 \leq r \leq 8$	$r=9$
$Z'^{(r)}_1$	$(Z_1^{(10-r)})^{-1}$	$(Z_1^{(10-r)})^{-1}$	$(Z_1^{(10-r)})^{-1}$
$Z'^{(r)}_2$	$-Z_2^{(10-r)}$	$-Z_3^{(10-r)}$	$-Z_2^{(10-r)}$
$Z'^{(r)}_3$	$-Z_3^{(10-r)}$	$-Z_2^{(10-r)}$	$-Z_3^{(10-r)}$
$Z'^{(r)}_4$	$(Z_4^{(10-r)})^{-1}$	$(Z_4^{(10-r)})^{-1}$	$(Z_4^{(10-r)})^{-1}$
$Z'^{(r)}_5$	$Z_5^{(9-r)}$	$Z_5^{(9-r)}$	N/A
$Z'^{(r)}_6$	$Z_6^{(9-r)}$	$Z_6^{(9-r)}$	N/A

Table 3.1: IDEA decryption subkeys $Z'^{(i)}_r$ derived from encryption subkeys $Z_r^{(i)}$. $-Z_i$ and Z_i^{-1} denote additive inverse modulo 2^{16} and multiplicative inverse $2^{16}+1$ of Z_i respectively.

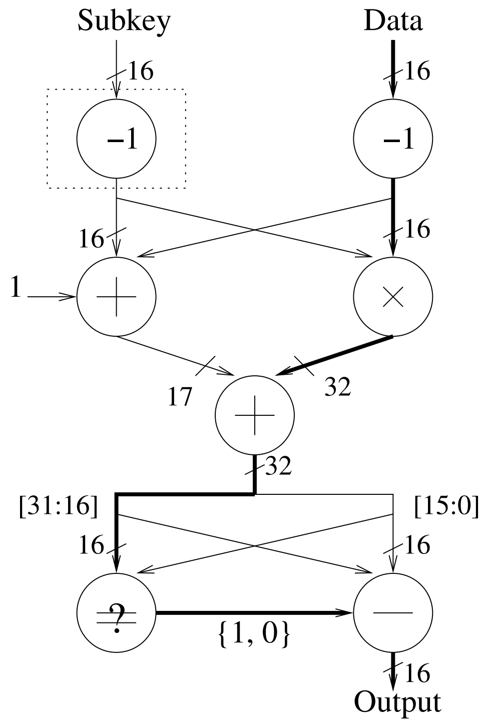


Figure 3.2: Block diagram of the multiplication modulo $2^{16}+1$.

process, the subtracter in the dashed box can be eliminated. This will save some logic resources but cannot speed up the design since there is still a subtracter in the same level. The operations on the same horizontal level can be carried out in parallel. An exception is in the last level where the carry input of the adder depends on the result of the comparator.

As shown in Figure 3.2, the critical path is indicated with a thick line. To get the best performance in this module, the 16-bit multiplier is constructed by the Xilinx CORE Generator [Xil00b] to maximize the throughput. The generated multiplier, with output registers, has a latency of 4 clock cycles. To match the throughput of the multiplier, every level is pipelined such that there are 7 cycles latency in the modulo $2^{16} + 1$ component.

3.3.2 Deeply Pipelined IDEA Core

The architecture of the IDEA core has been shown in Figure 3.1. The $\odot$ operators are replaced by the component described in Section 3.3.1. There are 3 modular multiplication in the critical path in a full round. Delays and output registers are used to balance the critical path. To save area, delays are implemented with the Xilinx Virtex SRL16E shift register primitives [Xil99, GA99].

The architecture of the IDEA core after inserting the delays is shown in Figure 3.3. The numbers in the circles represent the number of delay cycles added to the path. There are 22 cycles of latency introduced in a full round and 7 cycles in a half round. There for, a complete IDEA cipher has totally 183 cycles delay. Since every single logic path has combinational delay shorter than that in the multiplier from CoreGen, the throughput of the design is bound by the speed of the generated multiplier. Thus, a faster multiplier implementation can improve the design.

In ECB mode, the design is fully pipelined and a 64-bit plaintext block can be processed every cycle since there are no feedback paths. When encrypting a large amount of data, the latency is not important.

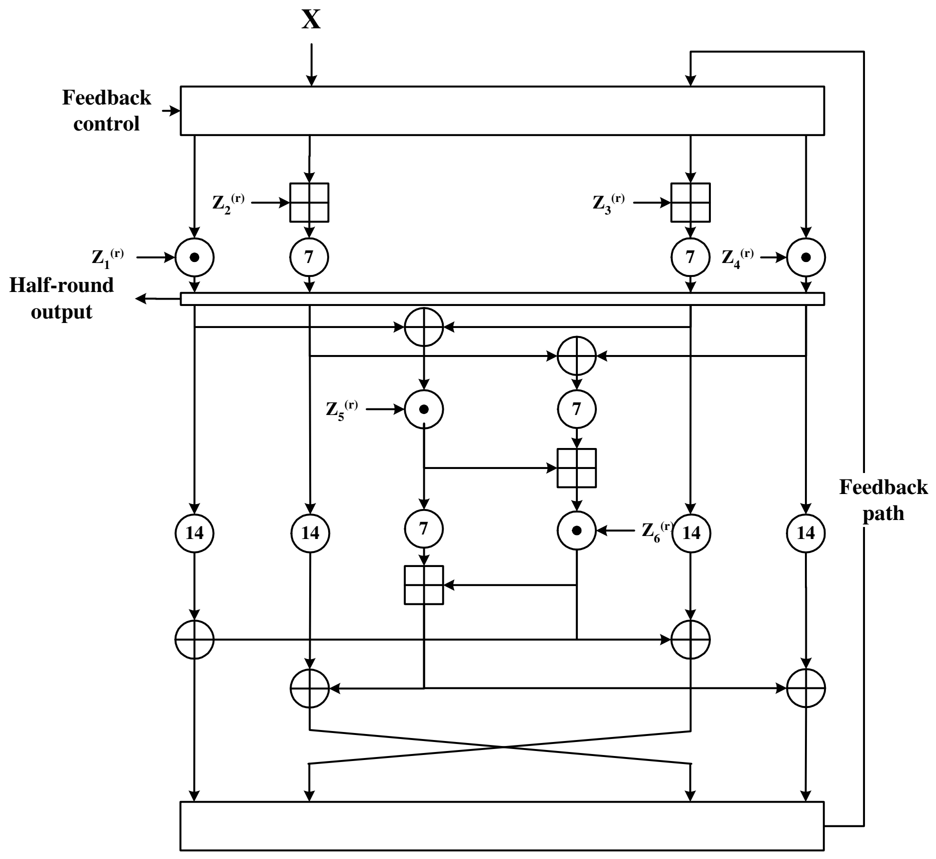


Figure 3.3: Architecture of the bit-parallel IDEA core.

3.3.3 Area Saving Modification

The design can be modified to fit into a chip which is not large enough for the complete eight and a half rounds. To use minimum logic resources, only one round is used to perform the cipher function. To enable this configuration, a feedback control must be added to the data input port in the full round design. As shown in Figure 3.3, The there two possible inputs to the core: the plaintext block and the output of the previous round. The operation is as follows:

- Initially, the feedback control logic selects the plaintext block, X, as the input.
- 22 plaintext blocks are accepted to fill the pipeline.
- Feedback logic then selects the outputs of the round as input. By doing so, the data passes through the round logic again.
- After 183 cycles, the half-round output is sampled to obtain the ciphertext blocks. At the same time, the feedback control accepts plaintext input again.

Figure 3.4 illustrates the above procedure. The number of rounds instantiated can be varied in the design. Trade off between area consumption and performance thus can be easily evaluated. For example, if only one round is instantiated, the scheme saves up to a factor of 8.5 in area.

3.3.4 Key Block in Memory

The key block storing the 128-bit input key needs to be shifted right 25 bits after the current subkeys are used up. The number of subkeys generated by the 128-bit key block, 8, is not a multiple of the subkeys required in a round. These mean the round computation must stall waiting for another set of subkeys after shifting. This either makes the control unnecessarily complex or introduces a irregular round design. The key block must be restored to the initial state after a data block is processed. The 25-bit shift prohibits the restoring procedure to be implemented by

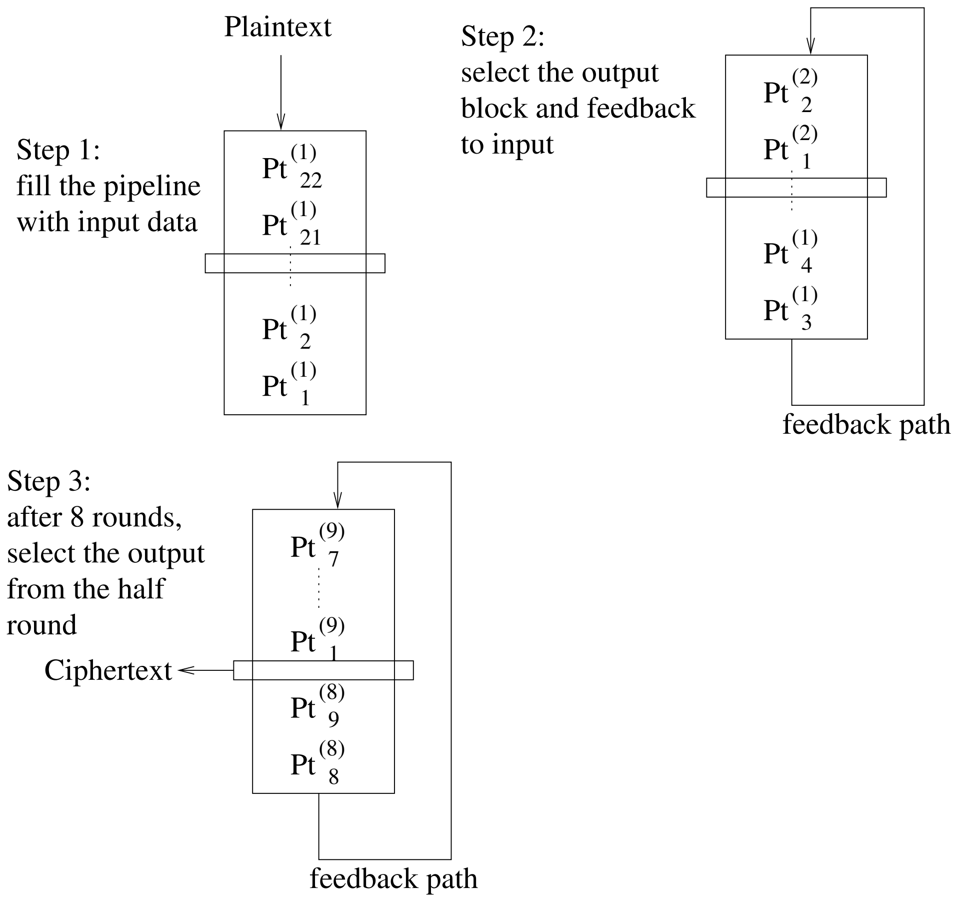


Figure 3.4: Operating procedures of pipelined IDEA core. $Pt_n^{(k)}$ is the n^{th} plaintext in the k^{th} round.

regular shifting. Also, the utilization rate of the key scheduling logic is very low. Since design has computed all the required 52 subkeys after the first data block, it is unwise to repeat the computation for every following blocks.

To avoid all these problems, the 52 subkeys are pre-computed in software and stored in memory (SRL16Es) in the design. The bus width of the key memory is 6×16 bits which will provide the keys for a complete round. Just before entering the next round, the key memory is rotated to prepare the next 6 subkeys. The advantages of this design are:

- No stall stages are required inside a round, thus simplifying control.
- The round design is regular by assuming the required subkeys are always ready.
- The subkeys are computed only once, in software.
- The logic for key scheduling is minimized to only several memories.
- To eliminate the -1 operation of subkey in multiplication $mod\ 2^{16}$, some subkeys should have 1 subtracted from them in the pre-compute stage. The scheduling process combines this in software. By doing so, 4 16-bit adders, i.e. about 32 SLICEs, are eliminated in each round.

The above design is suitable for an area optimized IDEA core with less than eight and a half rounds. For the design with all required rounds, no rotation in memory is required and the subkeys can embedded in the operators for further optimization. The drawback of the pre-computed key design is that changing key in a design requires runtime configuration.

3.3.5 Pipelined Key Block

For the deeply pipelined IDEA core, the values of subkeys are also pipelined. To save storage area, shifting is used instead of pipelining. At the start, only the first 4 subkeys, $Z_1^{(n)}$ to $Z_4^{(n)}$ are shifted. After 7 clock cycles, $Z_5^{(n)}$ is shifted. After another 7 clock cycles, $Z_6^{(n)}$ is shifted. This scheme ensures that the corresponding subkeys are changed at the same time that the data blocks for the current round reach the position in the pipeline where the subkeys are used.

3.3.6 Interface

The interface to the host system is simple. The host system writes a 64 bits plaintext to the FPGA. The hardware sets a flag bit after it receives the data. This flag is passed along the same pipeline as the data so that when the flag reaches the end of the pipeline, the ciphertext is ready at the output port, and the host system can read the encrypted message. This flag also serves as a busy flag for the input.

3.3.7 Pipelined Design in CBC Mode

In CBC mode, the input block must be XORed with the ciphertext of the previous block and the design can only accept one plaintext block after the previous one reaches the end of the pipe. This reduces the throughput by a factor of 183 and so the deep pipeline technique does not have any advantages in this mode.

Since the block cipher is usually used in network communication software, such as SSH, on the server side, there may be multiple connections using the same encryption protocol. The server can input plaintext blocks belonging to different connections (and hence different encryptions) in the pipelined design. There is no data dependency in these blocks even in CBC mode. By doing so, the utilization rate of the hardware is increased and the overall performance is improved. This method requires the software to select the correct key scheduling components for different connections. Assuming there are large number of concurrent connections such that the pipeline can be filled, this scheme can achieve a 22 times speedup in CBC mode.

3.4 Summary

A high-performance implementation of the IDEA block cipher was presented in this chapter. The deeply pipelined implementation achieved an encryption rate of 592Mbps using a 50MHz clock. An area saving design which iterates over a number of rounds and can be used in smaller chips was also implemented.

Chapter 4

Variable Radix Montgomery Multiplier

4.1 Introduction

Modular-multiplication (*mul-mod*) is an operation commonly used in security related applications. Such applications include the RSA public key system and secure random number generation. This class of computations usually involves vary large numbers which range from 512 to 2048 bits in size. This chapter will address some methods to improve the computation speed in a hardware implementation.

Various algorithms have been developed to improve the speed of *mul-mod* computations. For example, the Montgomery method [Mon85] speeds up the calculation by converting the inputs to a 2^k-residue system. In the Montgomery method, simple truncation and bit masks are used instead of trial division, to compute the remainder. A systolic array implementation can achieve high performance by dividing a complex algorithm into small and regular parts. These small systolic cells will work together to produce the final results of relatively higher speed. Combining these two techniques can result in significantly improvements in the performance of a *mul-mod* implementation.

The configuration parameters of the systolic structure will affect the performance of the design. One of the most important parameters is the radix used inside the systolic cell. Different radixes will result in different size and speed. In this chapter, we explore the relationship between the radix parameter and the size/speed.

This chapter was organized as following. Section 4.2 introduced the algorithms of RSA cryptography. Section 4.3 and 4.4 explained the Montgomery algorithm and systolic array structure. The structure radix-2^k Montgomery multiplier core was presented in section 4.6. The implementation details was shown in section 4.7. Finally, the systolic architecture and brief results are summarized.

4.2 RSA Algorithm

RSA [RSA78] is a secure public key cryptography standard used widely in many applications. It was first invented in 1977 by Ronald Rivest, Adi Shamir, and Leonard Adleman. The RSA algorithm can be applied in both encryption and authentication systems.

The basic idea of RSA is quite simple. There are two large prime numbers: p and q. These two primes are kept secretly from unauthorized parties. The modulus N is the product of the p and q:

$$N = p \times q.$$

By selecting a public exponent, e, such that:

$$gcd(e, (p-1)(q-1)) = 1,$$

the receiver can compute the private exponent from these parameters:

$$d = e^{-1} \; mod \; ((p-1)(q-1)).$$

However, if p and q are not known, N must be factorized to obtain the private exponent, and such a factorization is intractable using current technology for larger N.

To speed up the encryption process, the public exponent, *e*, is chosen to be small in practical applications. The ITU-T (International Telecommunications Union, ITU-T) suggests values of *e* such as $2^1 + 1$ and $2^{16} + 1$.

The encryption/decryption processes can be represented by the following equations (where C is the ciphertext and M is the message):

$$C = M^e \; mod \; N, \; M = C^d \; mod \; N$$

The size of the RSA key is given by the number of bits of the modulus *N*. For reasonable security, common practice is to use RSA keys greater than 1024 bits in length.

4.3 Montgomery Algorithm – $A \times B \; mod \; N$

The security of RSA is based on the difficulty in factoring integers which are the product of two large primes. Modular multiplication of large integers also pose difficulties using general hardware. One of the most efficient ways to compute the modular multiplication, $(A \times B) \; mod \; N$, is the Montgomery algorithm [Mon85].

The Montgomery algorithm converts the input numbers into a special residual system. By doing this, the computation of $(mod \; N)$ is transformed to be $(mod \; 2^n)$, where *n* is the bit width of *N* such that $2^{n-1} \leq N < 2^n$. The algorithm actually computes $n - 2$ bit data. Let $R = 2^n$. R', the inverse of R modulo N, is computed by $R'R = 1 \; mod \; n$. Assume the *R'* and *N'* are computed in advance such that $RR' - NN' = 1$.

Transform the inputs in the following way:

$$A' = (A \times R) \; mod \; N \text{ and } B' = (B \times R) \; mod \; N.$$

Then the modular multiplication

$$P' = (A' \times B' \times R^{-1}) \; mod \; N$$

can be computed by:

```
t := A′ * B′;
m := (t * N′) mod R;
u := (t + m * N)/R;
if u >= n then
        return u - n
else
        return u;
```

The above procedure involves only the division and mod of *R*, where R is a power of 2, greatly simplifying the computation since they become shifts and bit masks respectively. The final result can be obtained by converting *u* back to a normal number system.

The implementations of Montgomery algorithm can be divided into two different strategies: redundant representation and systolic array. The latter will be discussed in detail in later sections.

4.4 Systolic Array Structure

Hardware implementations of large integer computations can be done efficiently using a systolic array. This design style is characterized by high clock rates and is implemented using simple processing elements.

Each systolic cell has the same structure and is responsible for a small portion of the number. The radix of the cell can vary in different designs. Higher radixes will consider more bits at a time with reduced number of clock cycles. This is at the expense of increased logic count and reduced clock frequency. One extreme is the fully parallel design which considers all bits at a time. Another extreme is the radix-2 design which considers only 1 bit per cell.

It is a challenge to find the optimal radix to fulfill design objectives. Redesigning the systolic cell to check the performance for every radix is not efficient. In this work, a general radix systolic multiplier cell is proposed. It uses a 2's complement number system and the radix can be changed by setting a few parameters. The design was developed in VHDL so it can be synthesized on different hardware. Developers can optimize the critical part of the model for special target platform. It is a fast and reliable way to find out the optimal are/performance tradeoff.

4.5 Radix-2^k Core

In this section, the original method of systolic Montgomery design from Peter Kornerup [Kor94] is described for completeness. The method to extend the design to variable radix is then described.

4.5.1 The Original Kornerup Method (Bit-Serial)

```
for i := 0 to n
        step_1:    q := S mod 2;
        step_2:    S := (S + qN) div 2 + aB;
end for;
```

The proof of the correctness of this algorithm is given in [Kor93]. This algorithm is only suitable for computing with an odd modulus *N* which presents no problems in practice. To make the algorithm suitable for a systolic implementation, step_2 can be modified as follows:

$$S := \lfloor\frac{S-q}{2}\rfloor + q\lfloor\frac{N+1}{2}\rfloor + aB \tag{4.1}$$

Since $q = S\ mod\ 2$, then $S - q$ is always even i.e. $\frac{S-q}{2}$ has no remainder. Since N is an odd number, then $N + 1$ is always even i.e. $\frac{N+1}{2}$ has no remainder. The first term in (4.1) is generated by right shifting the previous result. The second term is generated by pre-computing $\frac{N+1}{2}$. The last term is the product of B and the LSB from A. A is shifted in every iteration and an a new a is generated then. The actual computation in hardware is to sum up 3 numbers (all n bits in width) within a clock cycle. Each systolic cell sums parts of the numbers and stores the carries for next clock cycle.

4.5.2 The Radix-2^k Method

```
for i := 0 to n/k
        step_1:    q := (S*N') mod (2^k);
        step_2:    S := (S + qN) div (2^k) + aB;
end for;
```

In this method, the step 2 can be modified as:

$$S := \lfloor\frac{S}{2^k}\rfloor + q\lfloor\frac{N}{2^k}\rfloor + aB + \lfloor\frac{(S \; mod \; 2^k) + q(N \; mod \; 2^k)}{2^k}\rfloor \tag{4.2}$$

The last term (referenced as '*f*' in the rest of this paper) will be within the range of [0, $2^k - 1$]. This suggests that the computation is still a sum of 3 inputs except the k LSBs.

The structure of a n-bit (actually n+2-bit in hardware) radix-2^k Montgomery multiplier is shown in Figure 4.1. The extra bits in hardware are to eliminate the need for final reduction after Montgomery multiplication.

The inputs and outputs of the top level entity in Figure 4.1 are shown in table 4.1. F-cell in the figure computes the last term in Equation 4.2. Let $j = \log_2 k$. In every clock cycle, one systolic cell outputs j bits of S in the corresponding location.

4.5.3 Time-Space Relationship of Systolic Cells

The figure 4.2 shows the time-space relationship of the systolic cells. It shows that by inserting registers between cells, one can control the computation of iterations in which a cell performs according to the cell's location in the array.

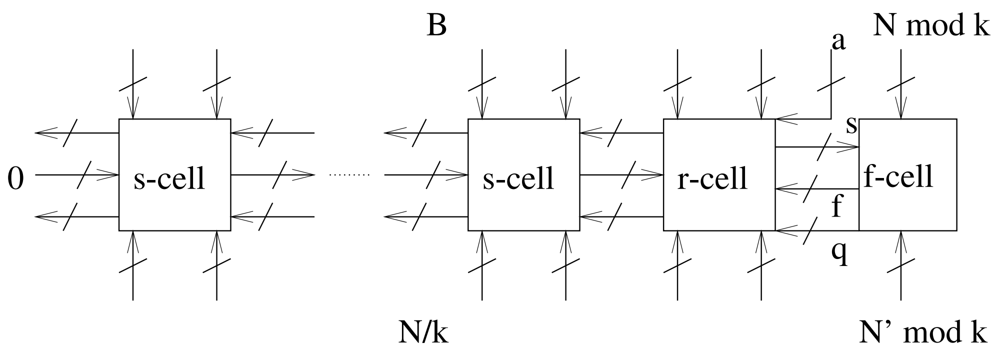


Figure 4.1: Top level overview of multiplier. All signals in this figure are k-bit in width.

name	direction	general width
B, N	in	n+2
A	in	n (bit-length)
$N'mod2$	in	$log_2(radix)$
S_out	out	$log_2(radix)$

Note: A is shifted $log_2(radix)$ LSBs to the r-cell every clock cycle.

Table 4.1: Inputs and Outputs of Top Level Entity.

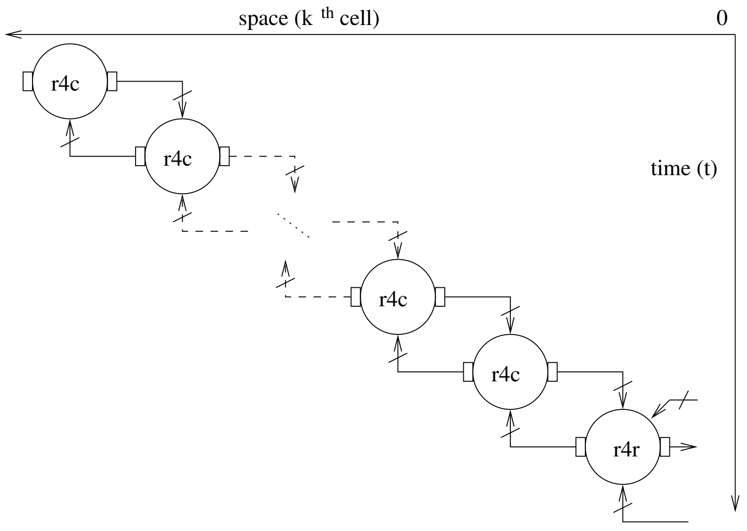


Figure 4.2: Space and time relationship of the systolic array.

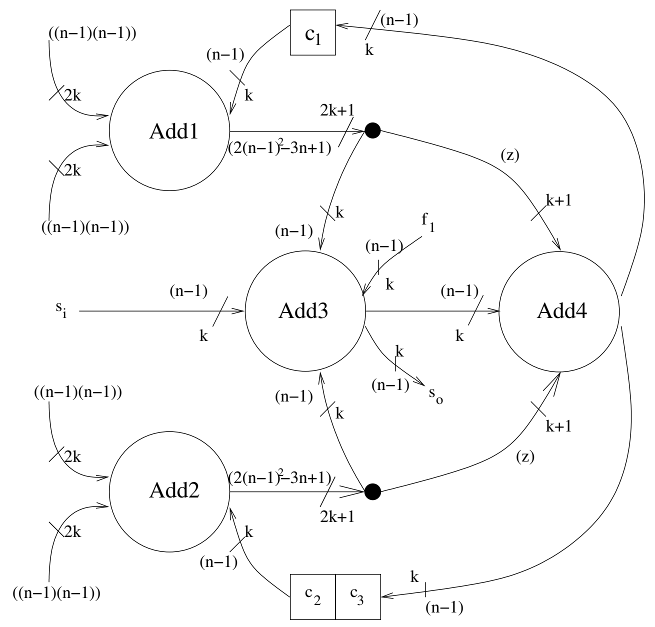


Figure 4.3: Generalized data path of radix-2^k cell.

4.5.4 Design Correctness

The correctness of the design depends on handling carries and matching the bit widths of each of the terms. Figure 4.3 shows the bit width of each component of an r-cell. The numbers in the parentheses are the maximum values passed though the paths and $n = 2^k$. It can be seen from in the figure that no overflow will occur on any signal.

4.6 Implementation Details

A set of VHDL files was developed to describe the behaviors of the components in the model. The width of I/O and internal signals were controlled by generic attributes of the entities. So there are no hard coded bit widths in the system. The radix and bit width of the design are controlled by two constants in the top level entity. These two parameters propagate through out the design in a top down fashion. By changing the value of the two constants in the top level entity, the complete design can be transformed to any radix base as needed. Since the generic attributes are resolved during the synthesis stage, this will not introduce any overhead on the target hardware. The bottleneck of the design is the multiplier used to compute aB and qN. To make the design as flexible as possible, the default VHDL operator, '*', was used. Designers are free to replace the multiplier used, provided that the I/O and timing requirements are matched. For example, the fast 18-bit unsigned multipliers in the Xilinx VirtexII chip can be used to replace the ordinary '*' operator in VHDL.

4.7 Summary

In this chapter, a method to construct a variable radix systolic Montgomery multiplier hardware was presented. The width of the datapath of these designs can be changed via generic parameters. By doing so, the radixes in the multiplier can be adjusted by designer. This will help to measure the performance of different designs efficiently.

Chapter 5

Parallel RC4 Engine

5.1 Introduction

In this chapter, an implementation of the alleged RC4 cipher which achieves significant performance improvement over a microprocessor implementation is presented. RC4 is used for encryption in products such as the secure sockets layer (SSL) protocol, the secure shell (SSH) protocol, the wired equivalent privacy (WEP) algorithm (part of the IEEE 802.11b wireless LAN security standard), Lotus Notes, Oracle Secure SQL, Microsoft Office and Adobe Acrobat (Acrobat 4.x or older). Furthermore, the key size is often limited to 40 bits due to US export restrictions.

A brute force key search can be used to determine the key used to encrypt a message by trying every possible key to decrypt the message. Such a key search is trivially parallelizable and successful key searches using loosely coupled microprocessors in a distributed computing approach have successfully been applied to the 56-bit DES algorithm and the 56-bit RC5 algorithm.

Application specific integrated circuits (ASICs) have also been used by the Electronic Frontier Foundation (EFF) to implement a DES key search engine, called "Deep Crack", which could search 88 billion keys per second [Ele98]. The machine solved the "Blaze Challenge" and the RSA Laboratories DES–III challenge, the latter on January 1999 in 22 hours [RSA99]. One limitation of an ASIC based implementation is that they are hardwired for specific problems.

There have been two previously reported FPGA based RC4 key search machines. In 1996, Goldberg and Wagner proposed an RC4 search engine using an Altera RIPP10 board which had 8 FLEX8000 chips and four static RAM chips [GW96]. Their design could perform 4 parallel searches and each unit required 1286 cycles per key. Kundarewich *et. al.* proposed a key search engine using a single Altera EPF10K20 complex programmable logic device (CPLD). In their implementation, each search unit required 1304 cycles per key and 5 parallel searches could be made at 10MHz [KWH99].

The RC4 implementation described in the chapter integrates the key search controller and 96 parallel RC4 decryption engines on a single Xilinx Virtex XCV1000E FPGA (much larger FPGA devices are already available). Although the RC4 implementation operates at a clock frequency which is an order of magnitude lower than that of the latest microprocessors, the FPGA implementation achieves a significant speedup due to the following features:

- Parallelism in the implementation of the RC4 core allows several operations to be completed in a single cycle.
- On-chip resources were used to achieve a very low latency, high bandwidth memory interface
- The memory used was dual-ported, allowing for higher memory transfer efficiency.
- Floorplanning was used to minimize interconnect delays
- A large number of the encryption cores were used in parallel.

In this implementation, each search unit requires approximately 800 cycles per key and 96 such units are integrated on a single FPGA.

The rest of the chapter is organized as follows: in Section 5.2, the RC4 and key search algorithms are described. Section 5.3 describes the architecture of the RC4 implementation. Implementation details are presented in Section 5.4. Finally, there is a summary on this work in Section 5.5.

5.2 Algorithms

5.2.1 RC4

RC4 is a stream cipher designed by Ron Rivest and was originally proprietary to RSA Data Security [Sch96]. The algorithm was leaked anonymously to the Cypherpunks mailing list in 1994. The RC4 algorithm generates a key dependent pseudorandom number sequence of arbitrary length.

In the description below, two 256 byte arrays are used, namely the *K-block*, *K* and the *S-block*, *S*. Note that the *K-block* does not change during the encryption process.

The RC4 algorithm can be divided into 2 phases: a key scheduling phase and the pseudorandom number generator (PRNG) phase. Both phases must be performed for every new key.

In the key scheduling phase, a scrambling process is used to produce a key dependent permutation of $0, 1, \ldots 255$ in the *S* array. In the initialization stage, the *S* array is set to the identity permutation using the formula $S[i] = i(i = 0, 1 \ldots 255)$ and the *K* array is set to the *key*, repeating as necessary to fill the array. The *S* array is scrambled by selecting two indices i and j and then swapping $S[i]$ and $S[j]$. In pseudocode form, the key schedule is computed as follows:

```
keyschedule()
{
    /* initialization */
    for i = 0 to 255
        s[i] = i;
    /* scrambling */
    j = 0;
    for i = 0 to 255
    {
        j = j + K[i] + S[i];
        swap S[i] and S[j];
    }
}
```

The PRNG phase is similar to the key schedule. Indices i and j are selected and $S[i]$ and $S[j]$ swapped. The output of the PRNG is the value of the *S* array indexed by $S[i]+S[j]$ (i.e. $S[S[i]+S[j]]$). Encryption or decryption is achieved by performing an exclusive-OR of the pseudorandom number output with the plaintext or ciphertext respectively. The pseudocode below shows the process for encryption of the plaintext in the *pt* array, the result being written to the ciphertext array *ct*:

```
prng()
{
    i = 0;
    j = 0;
    while not end of stream
    {
        i = (i + 1) mod 256;
        j = (j + S[i]) mod 256;
        swap S[i] and S[j];
        t = S[i] + S[j];
        ct[i] = pt[i] xor S[t];
    }
}
```

5.2.2 Key Search

The design described in this chapter performs a known plaintext attack via a key search [Sch96]. In a known plaintext attack, it is assumed that the ciphertext as well as the corresponding plaintext is available and one wishes to deduce the key used for encryption. The same architecture, with additional filtering logic (e.g. to detect if the message is 7-bit ascii) could be used for a ciphertext only attack.

If the plaintext and ciphertext are known and n bytes in length, checking that the ciphertext, ct, when decrypted using a key k is the same as the plaintext pt, is equivalent to checking if the first n bytes of the PRNG produces the sequence *pt* xor *ct*.

If N RC4 key search units are available, i is an index used to identify each RC4 key search unit, and $rc4(cxp, k)$ checks to see if the PRNG produced with key k gives cxp, the key search procedure can be described in psuedocode form as:

```
keysearch()
{
    k = 0;
    cxp = pt xor ct;
    forever
    {
        for i = 0 to N-1 (in parallel)
        {
            found = rc4(cxp, k + i)
            if (found(i))
                return k + i;
        }
        k = k + N;
    }
}
```

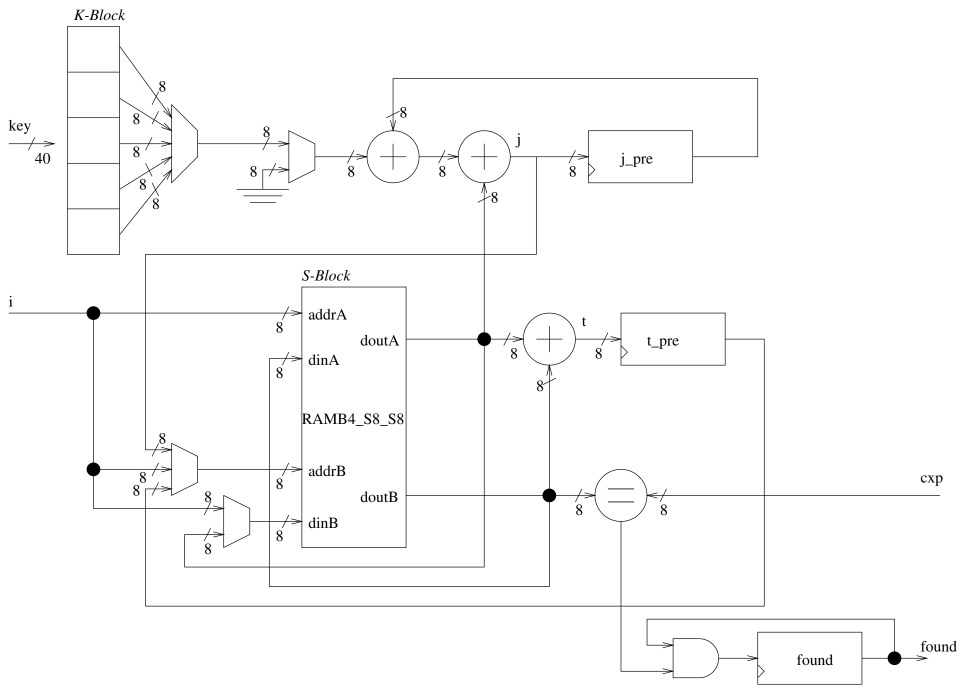


Figure 5.1: Datapath of the RC4 cell.

5.3 System Architecture

5.3.1 RC4 Cell Design

The datapath of a single RC4 cell is shown in Figure 5.1. The core component of the RC4 cell is the S-block for the *S* array, which is implemented using a 4096-bit on-chip Block RAM [Xil00a], configured as an 8-bit wide dual port memory. Since the RC4 algorithm requires only $8 \times 256 = 2048$ bits of memory for the *S* array, the Block RAM is divided into two halves via the most significant bit of the address. As the key scrambling phase for a new key is being computed in one half of the RAM, initialization for the next key is done in the other half. This scheme saves 256 cycles and hereafter, this combined initialization and scrambling step will be referred to as the key schedule phase.

Each iteration of the key schedule phase requires 3 clock cycles as shown in Figure 5.2. In the first clock cycle, *i* is passed into port A of the Block RAM as an

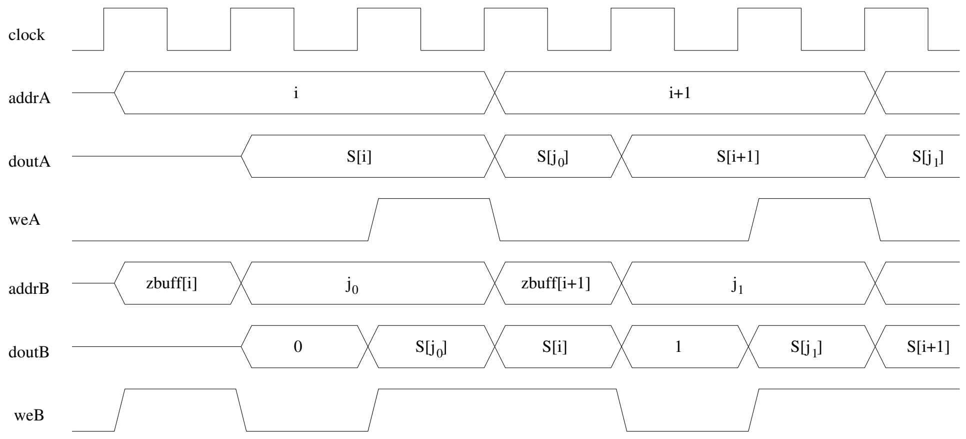


Figure 5.2: Timing diagram of the block RAM during the key schedule phase.

address, and the initialization of *S* for the next key is done at the same time via port B. In the second clock cycle, the value of S[i] becomes available and *j* is computed. In the last clock cycle, S[j] is available and the contents of S[i] and S[j] are swapped and written back to *S*. The total clock cycles required for the key schedule pahse are 768 ($= 3 \times 256$).

The PRNG phase (see Figure 5.3) also requires 3 clock cycles per iteration, hence a total of $768 + 3n$ cycles are required to test each key (for an n byte long ciphertext). Operations in this phase are similar to those of the key schedule phase except that S does not require initialization. The *t* value is ready (as the *t_pre* signal) in the first clock cycle of the next iteration. The output, *s[t]*, is read and compared with the *cxp* value in following cycle.

A possible memory contention problem exists in the last clock cycle of each iteration in the key schedule and PRNG phases, since it is possible that both ports attempt to write the same data to the same address, producing unpredictable results [Xil00a]. To avoid this conflict, a comparator is added to the RC4 cell (not shown in the schematic) so that if *i* and *j* are equal, the write enable to the memory is disabled. The operation performed in this clock cycle is to swap *S[i]* and *S[j]*. If the array indexes are the same, there will be no swap and no data losses.

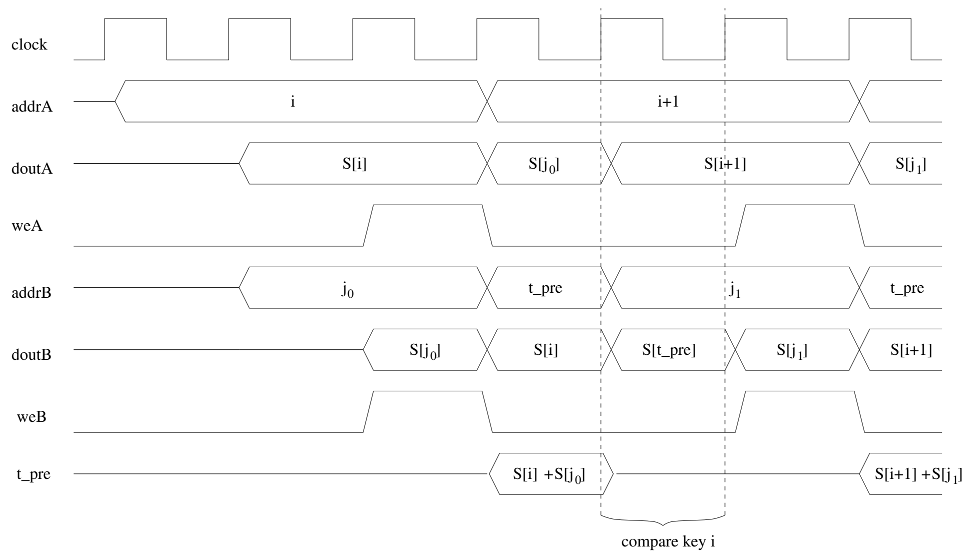


Figure 5.3: Timing Diagram of Block RAM in PRNG Phase

Finally, a latch called *found* in the RC4 cell is used to indicate whether the key being tested matches the plaintext. This latch is cleared if the byte produced by a decryption does not match cxp (as described in Section 5.2.2). Should the latch remain high after all bytes of the plaintext have been tested, the key being tested is the desired key.

5.3.2 Key Search

The top level block diagram of the design is shown in figure 5.4. All RC4 cells are identical. Each cell accepts a key input and sets a flag if the input is a valid key. There is one global key register which is initialized by the host and routed to all RC4 cells. A local key is computed in each cell by summing the global key with a cell offset, which is a unique value ranging from 0 to 95. By using this scheme, the RC4 cell array can process 96 different keys in parallel, after which, 96 is added to the global key.

All RC4 cells share a common control unit, implemented as a simple finite state machine (FSM). This unit controls the state of all the RC4 cells, updates the global

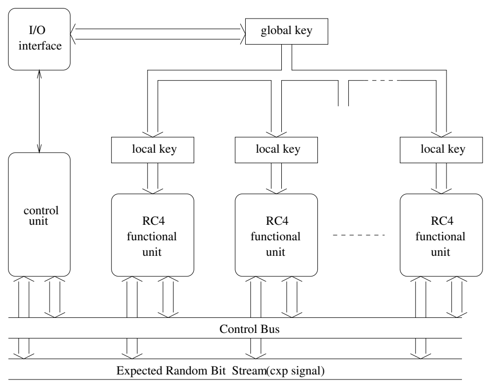


Figure 5.4: Block diagram of parallel RC4 key search machine.

key and also provides the interface to a host computer (discussed in Section 5.3.3).

5.3.3 Interface

In the host/key search engine interface protocol, the host must download the expected PRNG sequence *cxp* and then the start key value for the key search. After the search engine receives the start key, it works independently of the host, testing new keys until it detects that the *found* flag of an RC4 cell has been asserted (in which case the FSM halts). The host then can read the global key and offset which produces *cxp*. The host and key search engine communicate via a set of 3 64-bit read and 2 64-bit write registers.

The interface protocol is detailed in Table 5.1. Write registers are used by the host to send the start key (w0) and expected PRNG sequence, cxp (w1) to the key search engine. After the key has been found, the host can read back the global key value (r0), and the offset of the RC4 cell which asserted the *found* flag (r1, r2).

5.4 Implementation

The design is modularized and floorplanning was done to reduce implementation time as well as improve the maximum frequency of the design. In this section,

Step	Action	Register	State
1	Host writes expected PRNG sequence (cxp)	w1	idle
2	Host writes start key	w0	start
3	Host polls flag registers	r1, r2	searching
4	Search engine writes 96-bit offset	r1, r2	halt
5	Host reads global key	r0	idle
6	Host reads offset	r1, r2	idle

Table 5.1: Host/key search engine handshaking protocol.

details of the implementation are presented.

5.4.1 RC4 cell

There are 8 major components inside an RC4 cell, the dual port RAMs and the 40-bit local key registers being excluded from the RC4 cell for reasons described in Section 5.4.2. The names of the components and their functions are listed in Table 5.2.

The RC4 cell was designed to fit into a $4\ row \times 6\ column$ Virtex-E configurable logic block (CLB) array. All components are structural HDL descriptions containing only primitives provided by the Xilinx library. The physical placement of components were fixed using relative location (RLOC) attributes. The complete cell is a RPM (Relationally Placed Macro) which can be instantiated multiple times in the top level design. The block diagram in Figure 5.5 shows the layout of components within the RC4 cell. In the figure, the small rectangular boxes represent a slice (two logic cells, where each logic cell contains a 4 input lookup table) and two adjacent slices form a Virtex-E CLB. This scheme ensures low local routing delays.

The multiplexer for the *K_unit*, which is used to select a byte from a 40-bit key, is implemented using tristate buffers (TBUFs) and do not use CLB resources. This scheme replaces the large multiplexer in Figure 5.1 and reduces both logic and routing resources.

name	function
D_unit	8-bit 2-to-1 MUX select portB data input
A_unit	8-bit 3-to-1 MUX select portB address input
F_unit	8-bit compare and registers generate *found signal*
W_unit	8-bit compare detect Block RAM address conflict
I_unit	combinational logic to control MSB of portB address
J_unit	two 8-bit adders with registers outputs compute the *j value*
T_unit	8-bit adder with registered outputs compute the *t value*
K_unit	5-to-1 8-bit mux (using tristate buffers) select byte from K-block

Table 5.2: Components inside an RC4 cell.

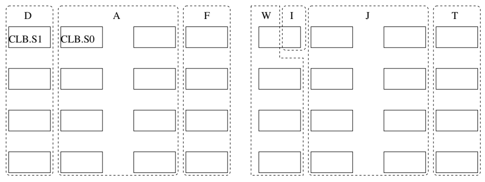


Figure 5.5: Block diagram showing component placement within an RC4 cell.

5.4.2 Floorplan

On the XCV1000E FPGA, the 96 Block RAMs are grouped into 6 columns. Adjacent Block RAMs are separated by 4 rows of CLBs. The RC4 cell described in Section 5.4.1 was designed to have the same pitch as the Block RAM and hence, each of the 96 RC4 cells is placed adjacent to a Block RAM which is used for the *S-block*.

The 40-bit local key is another module used in the design. This module is a 40-bit adder with registered outputs and is used to latch the sum of the global key and the offset of the RC4 cell. To avoid breaking the fast carry chain, this module is implemented as a column which is 20 slices (or 5 RC4 cells) high (see Figure 5.6). Five local keys are grouped together and placed perpendicular to their corresponding RC4 cells as shown in Figure 5.6. Since the local key modules have no direct connections to the Block RAMs, placing them away from the Block RAM column does not increase the routing delay. Since the TBUFs and CLBs are independent, the RC4 cell overlaps with the local key module in a section where the RC4 cell only uses TBUFs and the local key module only uses the CLBs.

Figure 5.7 shows the floorplan of the completed design. It can be seen that the RC4 cells and local key modules are placed close to the Block RAM columns. The control unit is located in the center where the distance to all RC4 cells is minimized.

5.5 Summary

In this chapter, a highly parallelized RC4 key search engine based on FPGA device was presented. Both high level design using VHDL and low level optimizations using floorplanning tools are implemented in the design.

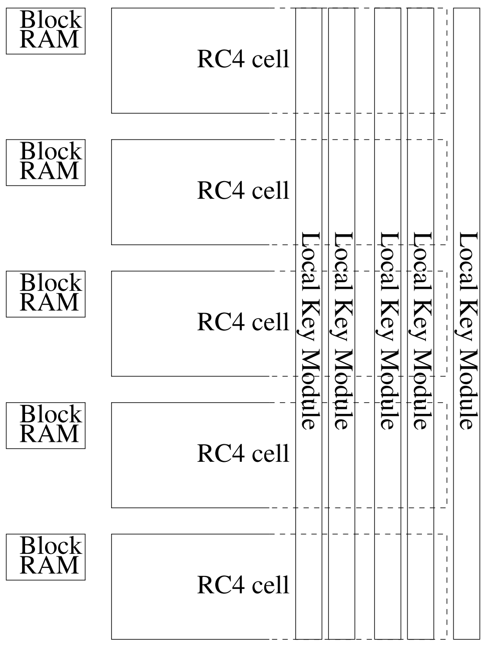


Figure 5.6: Block RAM, RC4 Cell and local key module placement.

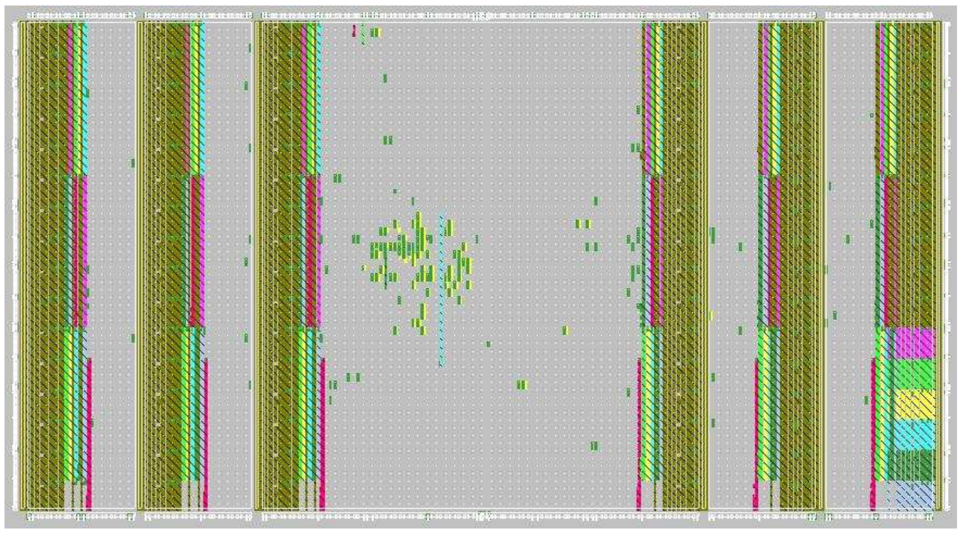


Figure 5.7: Floorplan of the completed design.

Chapter 6

Blum Blum Shub Random Number Generator

6.1 Introduction

Random number generators (RNG) are important in many security related applications such as key generation in cryptography [Ram89] and challenge generation in authentication protocols [TTTM02]. Most cryptographic systems rely on the unpredictability and irreproducibility of generated random sequences. Other applications using random number extensively are circuit testing, computer-based gaming, modeling of genetic systems and simulation. The main purpose of this work is to study a highly random hardware RNG based on a bit serial implementation.

There are two classes of random number generators: *real random number generators (RRNG)* and *pseudo random number generators (PRNG)*. Both RRNGs and PRNGs can produce a random bit stream for external use. The RRNG makes use of a non-deterministic source which may be the electronic noise, thermal noise or even radioactive decay [hot02]. PRNGs generate pseudo random numbers based on a deterministic algorithm. PRNGs require a starting state value called a *seed*. A common practice is to seed the PRNG using a RRNG and then use the PRNG to generate random numbers for the application.

There are two major performance criteria for RNGs: randomness and generation rate. A good RNG for cryptographic applications must be able to generate an unpredictable (at least to the external world) random sequence. On the other hand, the RNG must generate random numbers fast enough for various applications such as SSH servers and PKI based e-commerce systems.

This chapter introduces an implementation of a cryptographically secure hardware random number generator which can generate random numbers at an average rate of 211bps. The design includes one RRNG and one PRNG. The random sequence is actually a BBS (Blum Blum Shub) [LMM86] sequence and takes a random number generated by the RRNG as a seed. It is desirable to have a small, flexible and modular RNG, since cost, footprint and ease of interfacing are improved.

The chapter is organized as follows. The real random number generator algorithm was first introduced in Section 6.2. Then the BBS algorithm was described in Section 6.3. The top architecture of the RNG and the design details were presented in Section 6.4 and Section 6.5. The last section gave a summary of the design.

6.2 RRNG Algorithm

The physical random number source used is the phase-noise of a free-running oscillator. We chose this source since it has the least external components for an FPGA and the circuit can be powered directly by the FPGA chip. There are two clocks in the design: a slow and unstable external clock, F_l and a fast and accurate internal clock, F_h. This is achieved by using an edge-triggered D-type flip-flop with F_l as clock input and feeding the F_h to the data input. By doing so, the two square waves are mixed together to produce a output F_r. Figure 6.1 shows this structure.

The output rate of this method depends on the slow clock, F_l, which is deliberately designed to have high phase noise. Since this clock is not stable and the frequency varies with time, the throughput of the device is not fixed.

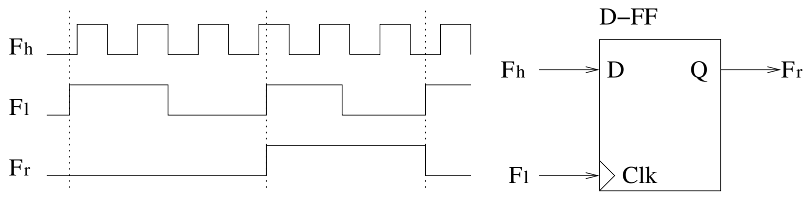


Figure 6.1: Oscillator sampling using D-type flip-flop.

There are several factors which affect the quality of the randomness of the algorithm. The first situation is that the duty cycle of clock F_l may not be 50%. In this situation, F_r will have unequal probability of being '0' or '1'. A parity filter which will even the number of '0' and '1' in a bit stream was applied to F_r. It can be shown [RRK98] that the probability of a '1' generated by the filter is $0.5 - 2^{n-1}(p - 0.5)^n$ where p is the probability of '1' in raw random stream F_r and n is the number of flip-flops in the parity filter. As n increases, the value of the expression tends to 0.5. The second factor is the selection of clock frequency. If the variation of the period in F_l is not large enough, there will be correlation between bits and so the value of the output can be predicted to some extent from the previous values. Previous research [RRK98] has shown that the standard deviation of the period of F_l should at least be 0.75 times the period of F_h. A third factor affecting the quality of the RNG is the random source itself. As there are both periodic and aperiodic electro-magnetic noise inside a computer system, there may be a patten in the output sequence as the result of coupling of periodic noise. The source of periodic noise includes the mains AC power supply, the master clocks on various bus systems, nearby wireless communication devices, etc. There is no way to eliminate this factor since the developing environment and the target environment are different and subject to change. The only way to test the quality of the RRNG is to test the results before installation and reject if it fails to pass the test suit.

6.3 PRNG Algorithm

The BBS algorithm [LMM86] was used in this design due to its high security. Some believe that the BBS algorithm is the most secure PRNG method available [VMD98]. The security of BBS is based on its long period and the difficulty in predicting the sequence even if all previously generated bits are known. Despite the strong security of the algorithm, the BBS sequence generator is simple and easy to understand. The following equation generates the BBS sequence X_i where i is a positive integer.

$$X_{i+1} = X_i^2 \; mod \; M$$

The *M* used here is a product of two large prime numbers p and q, which both have a remainder of 3 when divided by 4. X_0 is a seed which is co-prime with *M*. As proofed in [LMM86], a deterministic algorithm to compute the unique quadratic residue $X_{-1} \; mod \; M$ such that $(X_{-1})^2 \; mod \; M = X_0$ requires the knowledge of the prime factors of M. So M needs not to be kept secret as long as p and q are kept secret.

This algorithm is appropriate for use in cryptographic applications. Since large integer arithmetic is involved, it is slow comparing with other PRNGs. However, it has a strong security proof [LMM86], which relates the quality of the generator to the difficulty of integer factorization. The output of the generator is formed from the $log_2(log_2 M)$ least significant bits of X_i. The original algorithm only outputs 1 (least significant) bit per iteration. But Vazirani and Vazirani [VV84] showed that we can safely use at least $log_2(log_2 M)$ bits and the prediction of this sequence is as hard as factoring *M*. The typical bit width of *M* is 512 or 1024. Using a larger size will increase the number of available bits in each iteration, however, this is at the expense of larger storage area and computing power requirements.

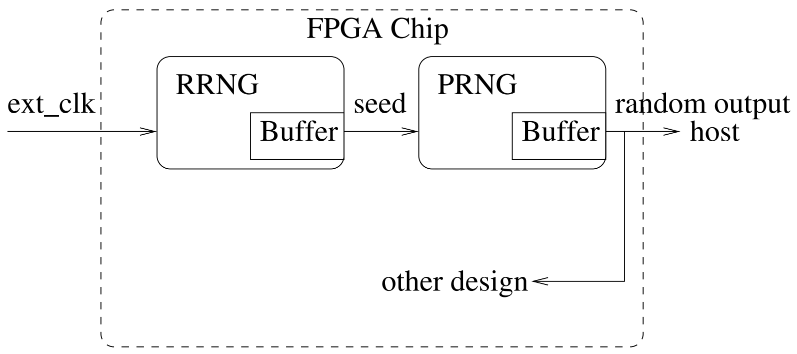


Figure 6.2: Overview of the RNG and PRNG.

6.4 Architectural Overview

A complete RNG was designed on a single FPGA chip with few external components. Since the target is to be used for cryptographic hardware, the circuit size of the generator should be as small as possible, leaving more logic resources for other functions. The generated data can be stored in a buffer for other logic on the same chip or read by the host system as a direct random source for software applications.

The design can be separated into two parts: the RRNG part and the PRNG part. Fig 6.2 shows the relation between the two parts. The RRNG first fills its buffer with random bits. This buffer will then be used as a seed in the PRNG part if constraints are met. The output of PRNG is also stored in a buffer which can be read by a host computer or other modules on the same chip. These two parts work independently.

6.5 Implementation

In this section, the implementation details are presented and the considerations behind the implementation are explained. The implemented design includes a 1024-bit BBS PRNG. The size of the BBS algorithm can be easily changed by appending more registers and increasing counter size. The modulus M in BBS algorithm is hardwired in the design.

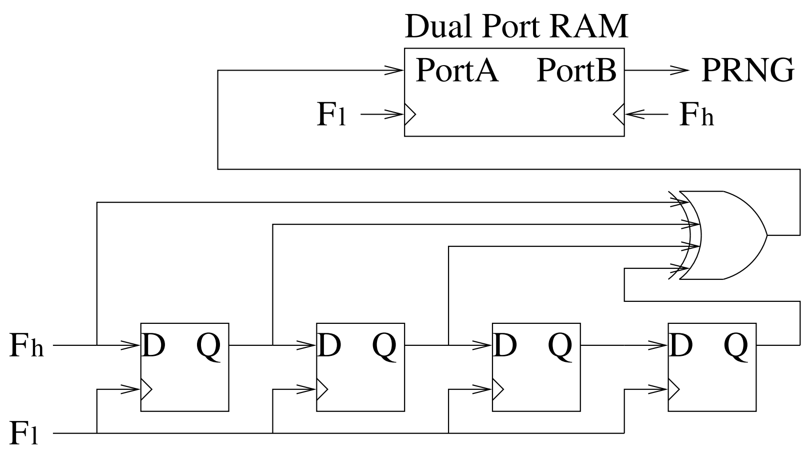


Figure 6.3: RRNG circuit.

6.5.1 Hardware RRNG

The RRNG circuit inside FPGA is shown in Figure 6.3. Two clocks are used in the design. F_h is a 100MHz high frequency clock generated by the PC DIMM interface. This clock is also the master clock for other parts in the design. F_l is a low frequency clock generated by an RC circuit which ranges from 225Hz to 1MHz. This RC circuit is constructed outside the FPGA chip and is sensitive to electronic and thermal noise. A variable resistor was used for testing the circuit using different clock rate. The digital mixing of F_h and F_l was implemented as shown in Figure 6.1. We call the output of this circuit, F_r, the raw random bit stream. A parity filter with 4 stages was applied to the raw random bit stream to accomplish the duty cycle bias. This is necessary since experiment results show that the duty cycle of F_h was approximately 54%.

The dual port BlockRAM acts as both a buffer storage and interface. The random bit stream is written to the memory through one port under F_l. The PRNG circuit reads this stream through another port under F_h. The RRNG circuit also contains a counter (not shown in Fig 6.3) whose output is used as the address for the BlockRAM.

The RRNG circuit starts generating a real random bit stream after power up.

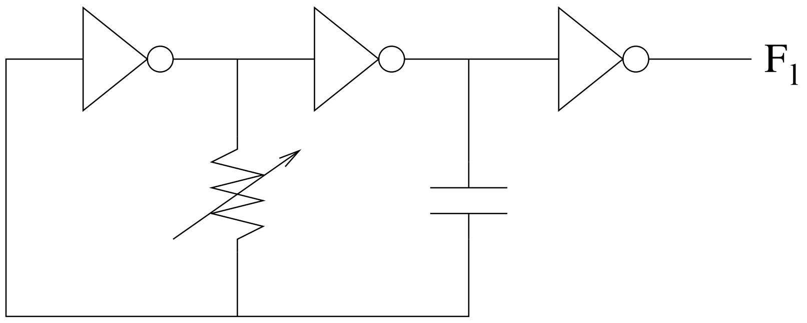


Figure 6.4: Circuit of External Clock.

When the buffer is full, the write enable is dis-asserted and the contents of the buffer remain unchanged. It will then assert a *full* signal to other parts of the circuit indicating that there are new random data in the buffer. When the PRNG requires a new stream of random bits, the circuit is *reset* and the process restarts. This is necessary since the stored random data may not pass the BBS seed validation. In this case, the BBS PRNG discards the current data and requests new data.

Fig 6.4 shows the design of the low frequency oscillator. The inverters used in the circuit are from a TTL 74LS74 chip. The charge on the capacitor and the resistance of the resistor will be affected by the background noise. This is the source of randomness in the design.

6.5.2 BBS PRNG

Fig 6.5 shows the data path of BBS PRNG. Note that all datapaths are one bit in width.

The computation part of the PRNG is a bit serial ALU. The signal *op* selects its operation modes:

$$ALUoutput = \begin{cases} A + B + C_{in} & \text{if op = 0 and sub = 0} \\ B - A & \text{if op = 0 and sub = 1} \\ B + C_{in} & \text{otherwise} \end{cases}$$

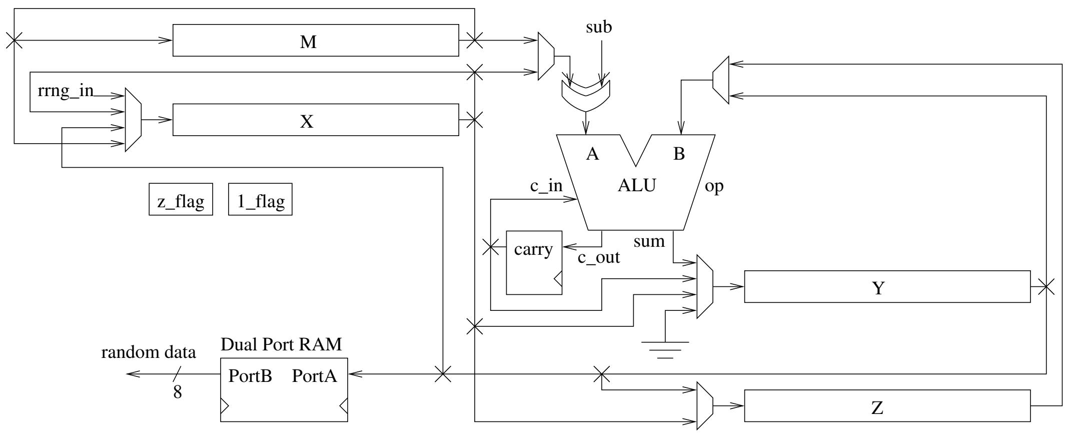


Figure 6.5: Circuit of BBS PRNG.

There are 4 1024-bit shift registers in the design: M, X, Y and Z. Register M stores the value of *M* which will not be changed. Register X stores the value of X_i. This value is initialized to a random seed from the RRNG and refreshed after each iteration. Register Y and Z can be combined to form a 2048-bit register, register YZ, to store the temporary results of ALU operations. All registers are constructed by cascading SRL16E components to reduce area consumption. The SRL16E is a single LUT configured as a 16-bit shift register with enable.

There are two internal flag registers: 0_flag and 1_flag. When the output of ALU is 0, the 0_flag is set. Else if the ALU result is 1, the 1_flag is set. These two flags are examined by the control FSM (finite state machine). The FSM also requires 3 counters (not shown in the circuit). Two of them are 10-bit counters which are used for arithmetic operations. The other one is used for storing random data to the buffer and should be 4 bits (i.e. $\lceil log_2 log_2 1024 \rceil$) in size.

The BBS PRNG performs three functions: seed validation, multiplication and modulo operations.

Seed Validation

One requirement for the BBS algorithm is that the seed, X_0 must be co-prime

Register	Value before validation	Value after validation
M	*M*	*M*
X	*M*	don't care
Y	X_0	0 or 1
Z	X_0	X_0

Table 6.1: Contents of registers in validation process.

with the modulo *M*. Euclid's method [Knu81] of finding $gcd(X_0, M)$ was performed. The following pseudo code shows the algorithm used:

```
seed_validation() {
get_seed:
    x = read(RRNG);
    M = modulus;
gcd_sub:
    M = M - x;
    if (M == 1) return(seed = x);
    if (M == 0) goto get_seed;
    if (M < 0) M = M + x;
    swap x, M;
    goto gcd_sub;
}
```

Table 6.1 shows the content of every registers before and after validation process starts.

The addition and subtraction in the process are performed in a serial manner. The LSB of register X and Y are passed to ALU as operand and the registers are shifted to right after each clock cycle. A single +/- operation will consume 1024 clock cycles then. To test if $y - x$ is smaller than zero, the carry bit is checked. Since we assume all operands (X_0 and *M*) are positive, the carry bit should be zero after subtraction if the result is also positive.

Register	Value before Mul	Value after Mul
M	M	M
X	X_i	X_i
Y	0	$(X_i^2)_{[511:256]}$
Z	X_i	$(X_i^2)_{[255:0]}$

Table 6.2: Contents of registers in multiplication process.

Multiplication

Table 6.2 shows the values of all registers before and after the multiplication process. The following pseudo code shows the procedures of performing a multiplication. Note that 1024 cycles are required to perform an *add* in the line labeled L1.

```
multiplication() {
    repeat 1024 times {
        LSB(Z) = 1 then
L1:         Y = Y + X;
        else Y = Y;
        shift_right_one_bit(YZ);
    }
}
```

Modulo

The result in register YZ is now X_i^2. The design then subtracts the contents of register Y by that of register M. Y is recovered if a negative result is generated. Register YZ is then shifted left by 1 bit and the process repeated. After 1024 iterations, the value stored in register Y is the result of $X_i^2 \ mod \ M$. The following pseudo code performs the *Mod* operation.

```
mod(M, YZ) {
    repeat 1024 times {
```

Register	Value before Mod	Value after Mod
M	M	M
X	X_i	X_i
Y	$(X_i^2)_{[511:256]}$	$X_i^2 \ mod \ M$
Z	$(X_i^2)_{[255:0]}$	don't care

Table 6.3: Contents of registers in Mod process.

```
            Y = Y - M;
            if (Y < 0) Y = Y + M;
L1:         YZ = shift_left_1bit(YZ);
      }
}
```

Table 6.3 shows the values of registers before and after the *Mod* process.

The algorithm is simple division using the paper-and-pencil method except that the quotient is not stored. There are faster methods for finding the remainder but the design described can be implemented in a manner which utilizes very little circuit area.

One implementation detail should be noted. In the pseudo code, register YZ is shifted left by one bit. In actual hardware this is not possible since the shift register is implemented by SRL16E components which can only shift in one direction. In most other operations including the validation, multiplication and backup, the registers shift in the right direction. The line labeled L1 is the only exception in the design. Our decision here was to make the hardware simple and uniform. This shift left operation is transformed to a shift right of 2047 bits through the ALU. We may implement the Y and Z registers in D-type flip-flops with selected inputs and leave the X and M registers unchanged. This will simplify the control unit but the size of the design will grow. As a SRL16E can replace 16 shift registers, the size of register Y and Z will grow by a factor of 16. Considering that we have only 4 registers in the complete design, this will make the design about 4.5 times larger. Thus this approach was not used.

Restoring Register X

After the *Mod* operation, the result, $X_i^2 \ mod \ M$, is then stored in register Y. We need to restore the registers' value to prepare for the next iteration. This is done by copying Y to X and Z. At the same time, zero is shifted in Y. The flag registers and carry register should also be cleared. After the restoring, the values in the registers are the same as list in the second column in Table 6.2. At the beginning of restoring, 10 LSBs in register Y are also shifted to the buffer in PRNG. This is the pseudo random bit stream generated in this iteration.

6.5.3 Interface

The PRNG buffer is also implemented in a dual port BlockRAM. As the pseudo random bit stream is shifting in one port, the host or other circuit in the same FPGA can read the random data through another port with 8-bit bus. The PRNG will assert a *full* signal when the BlockRAM is filled up. After detecting the *full* signal, the external design or host can start reading the data. There is also a *reset* signal which is used to clear the *full* signal.

Since random numbers are required, overwriting the data in the buffer when the buffer is full will not affect the randomness of the result. Thus no other handshaking circuit is needed. To read a continuous sequence from the design for evaluation, a double buffer method is used. As the host reads one buffer, the PRNG is writing to the other.

6.6 Summary

In this chapter, two serial RNGs were introduced. The RRNG senses the external noise and produces a raw random bit stream. This raw random data is used as a seed in the PRNG which uses the BBS algorithm. The output pseudo random bit stream is stored in buffers and can be used by either host or other circuit in the same chip. The entire generation process does not involve any CPU operations or use any memory storage in the host system. Thus security of the design is enhanced since the RNG state can be kept internal to the FPGA device. A serial architecture, which reduces circuit size, admittedly at the expense of speed, was used.

Chapter 7

Experimental Results

7.1 Design Platform

The results presented in this chapter are based on the following environment unless otherwise specified:

Design Entry VHDL

Target FPGA Platform Xilinx VirtexE family (XCV1000E-6)

Target Prototype Platform Pilchard

Simulation Tools Synopsys VSS 2000.12

Synthesis Tools Synopsys FPGA Compiler II

Implementation Tools Xilinx ISE 4.1i

Host Platform Linux on Pentium III 800

Software Drivers Language ANSI C

The prototypes are based on the Pilchard platform (Figure 7.1) [PMO$^+$01] which uses the SDRAM bus instead of the PCI bus used in conventional FPGA boards. We used this platform for the following reasons:

Simple Interface For most designs, the interfacing signals required are minimized. Besides clocks, only data I/O and read write signals are used. This simple design also provide more logic resources for the algorithm.

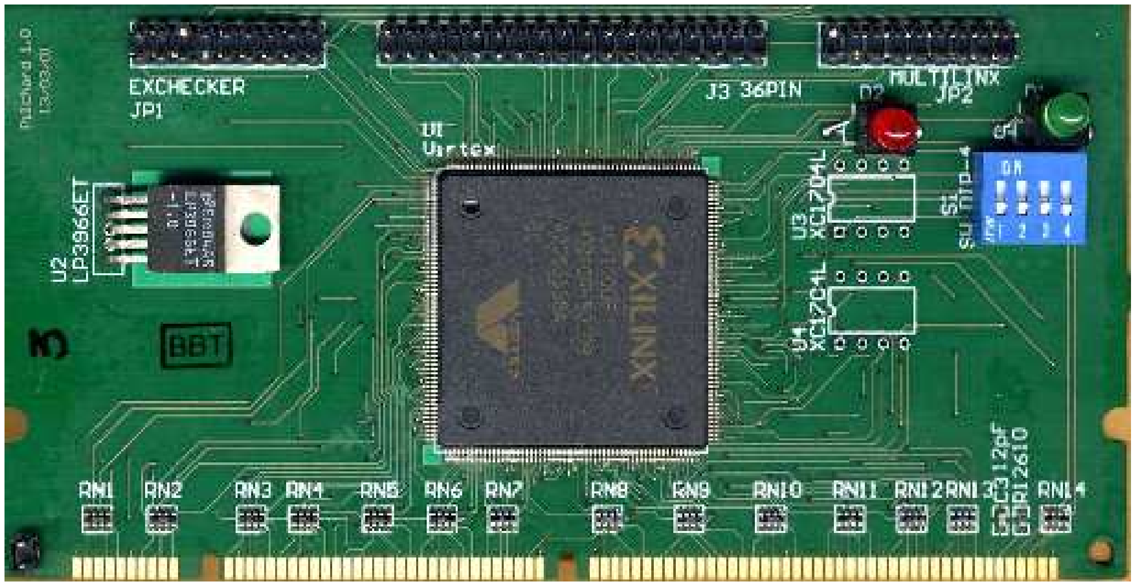


Figure 7.1: Photograph of the Pilchard board.

High Throughput The Pilchard platform interfaces to the host PC through the system memory bus. Since this bus is more highly coupled with the CPU than the traditionally used PCI bus, latency is reduced and bandwidth is increased.

Suitable FPGA Chip Onboard The Xilinx Virtex chip onboard provides useful components including DLLs (Delay Lock Loop), BlockRAMs, tristate buffers, etc. FPGAs of different sizes can be mounted on the board.

7.2 IDEA Cipher

In the experiment, two kind of IDEA cores was implemented. One is the flat (unpipelined) version without any intermediate registers and the other is the deeply pipelined version. The results of these two versions are compared in this section.

	SLICEs	Logic Utilization
flat core	848	6%
flat design (8 rounds + 1 half round)	7919	64%
pipelined core	1269	10%
pipelined design (1 round area optimized)	1363	11%

Table 7.1: Size of IDEA design.

	flat design	pipelined design
Number of cores	1	1
Clock rate (MHz)	14.5	83.7
Encryption per second ($\times 10^6$)	1.6	9.3
Encryption rate(Mb/sec)	102	592
Latency (clock cycles)	9	1407 =($175 \times 8 + 7$)

Table 7.2: Speed of IDEA Design.

7.2.1 Size of IDEA Cipher

The sizes of a single IDEA core in both versions as well as the sizes of the complete design are shown in Table 7.1. In the complete design, the interface to the host system are included. The flat version core is smaller due to the absence of pipeline registers.

In the both designs, there is a shift-register component for storing the sub keys. Since these registers were implemented in SRL16E primitives, it takes only 96 half slices (48 slices).

7.2.2 Performance of IDEA Cipher

Table 7.2 shows the speed of different IDEA implementations. Both designs use only one round of IDEA core and the key is stored in subkey memories. For both designs, increasing the number of round cores in a cipher will not affect the latency but allow the design to process more data at the same time. Thus the average throughput of the designs is increased.

Data from the host are written directly to the core using a burst mode transfer of 175 64-bit plaintext blocks. This is the latency of the pipelined core. By doing so, the pipeline is filled up and the performance is optimized. After the latency period of the design, the ciphertext is written to a buffer implemented in BlockRAM. The results are read by the host from the IDEA processor by doing a burst mode transfer of the contents of the BlockRAM. The decryption process is similar except the ciphertext is written to the IDEA core and the plaintext appears in the BlockRAM. The key component is also reconfigured for the decryption process.

Further improvement could be achieved by floorplanning. It is also possible to increase the encryption rate by scaling, i.e. to place multiple IDEA ciphers in parallel on a single chip. When using a platform with larger (> 64-bit) data bus, this scale up method can increase the throughput linearly.

7.3 Variable Radix Systolic Array

For an n-bit Montgomery multiplication using a radix 4 system, we must use an $(n+2)$-bit multiplier to ensure that $0 \leq S \leq 2N$, eliminating the need for reduction after each multiplication. Such a multiplier requires $\lceil \frac{n+2}{4} \rceil$ systolic cells. The data should be packed with leading 0s before passed to the multiplier. After $\frac{n+2}{2} + 1$ clock cycles, the result is computed and stored in the multiplier. Another $\frac{n+2}{2} + 1$ clock cycles are needed to shift out the remaining digits of the result. In total, n+4 clock cycles are needed for a complete multiplication. One systolic cell (either r4c or r4r) has four 2-bit output multipliers, two 4-bit adders, one 3-bit adder and one 2-bit adder. The critical path is from the 2-bit output multiplier to the 4-bit adder and then to the 3-bit adder.

To test the correctness of the design, a test bench was developed. The test bench contains only the 3 main components listed in the top level entity of Chapter 4. The interface is implemented such that the input parameters can be changed easily. The driver will test the hardware for all the possible cases by checking against the results

radix	max. freq. (MHz).	size (SLICEs)
2	143.205	10
4	77.797	35
8	61.132	76
16	49.801	81
32	41.599	115
64	41.064	141
128	35.120	180
256	37.448	209
512	34.400	265
1024	30.487	312
2048	29.444	364
4096	29.727	419
8192	30.723	485
16384	29.702	542
32768	29.847	632
65536	30.234	677

Table 7.3: Measurement of different radixes (one systolic cell).

generated by software. The test showed that the implementations was correct.

The performances of different radixes have been evaluated in this section. Table 7.3 show the figures reported by implementation tools.

The performance of the design is evaluated by how fast it can compute a multiplication. The time required to finish a modular multiplication is $t = \frac{clockcycles}{frequency}$. The number of clock cycles needed for a radix-2^k is $2(\lceil\frac{n+2}{k}\rceil + 1)$.The relation between performance and radix n is represented in Fig. 7.2. The area of systolic cell also changes with radix. Fig. 7.3 shows the relation between them.

The results show that the higher the radix, the better the performance. When the radix changes from 2^k to 2^{k+1}, the bit width of the multipliers used inside the systolic cell changes from k to $k+1$. This will introduce more logic levels on the critical path of the design. And so the maximum frequency decreases with increasing radix. At the same time, the number of clock cycles needed to compute the product is reduced. The ratio of clock cycles of radix-k and radix-[k+1] is about

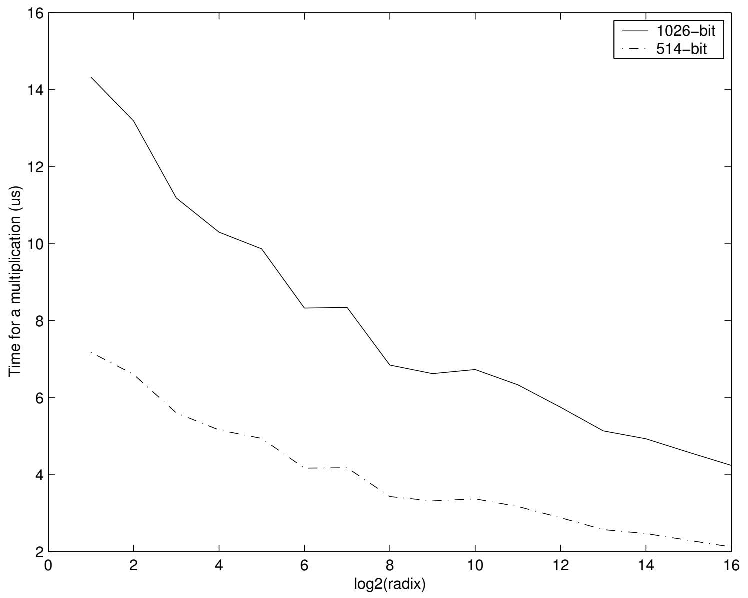


Figure 7.2: Performance and radix relation.

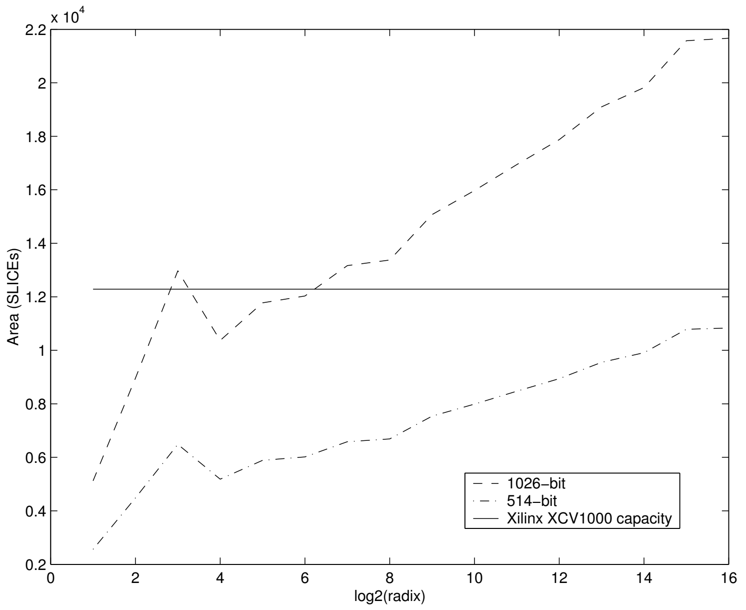


Figure 7.3: Area and radix relation.

radix	cells for a 1024-bit	cells which can fit on an		
	multiplication	XCV300	XCV600	XCV1000
2	513	307	691	1228
4	257	87	197	351
8	171	40	90	161
16	128	37	85	151
32	103	26	60	106
64	85	21	49	87
128	74	17	38	68
256	64	14	33	58
512	57	11	26	46
1024	52	9	22	39
2048	47	8	18	33
4096	43	7	16	29
8192	40	6	14	25
16384	37	5	12	22
32768	35	4	10	19
65636	33	4	10	18

Note: The second column is the number of cells required for a 1024-bit multiplication.

Table 7.4: Number of systolic cells for different Virtex FPGA chips.

$\frac{k+1}{k}$. When k is not too large, this ratio is more significant than the increase in logic levels. When k becomes vary large, e.g. ≥ 128, the logic levels are more significant than clock count. The performance will be decreasing in this range. The second range was not observed since the area requirements were already unrealistically large before it was reached.

The area is an important limiting factor. If the design cannot fit in a single chip, the timing performance is greatly reduced by the inter chip communication. In a n-bit radix-2^k system, there are $\lceil \frac{n+2}{2k} \rceil$ systolic cells. For example, the XCV1000E chip can only support up to 351 radix-4 systolic cells and 58 radix-256 systolic cells. Table 7.4 shows the possible number of cells which can be placed on different Virtex FPGA chips. The number may be different if special hardware, e.g. the fast multipliers in VirtexII [Xil02b], are used.

DLLs	1	out of	4	25%
BLOCKRAMs	96	out of	96	100%
SLICEs	5178	out of	12288	42%
TBUFs	4608	out of	12544	36%

Table 7.5: Device utilization summary.

7.4 Parallel RC4 Engine

An implementation of the RC4 key search engine was synthesized and implemented. The design was successfully tested on the Pilchard platform by performing key searches on randomly generated 40-bit keys. The performance was compared with an optimized software implementation on various general purpose microprocessors.

The RC4 engine containing 96 RC4 cells was designed for 50MHz operation as reported by the Xilinx timing analyzer. The system RAM bus interface operates at 100MHz. Resource utilization as reported by the implementation tools are listed in Table 7.5. Since each RC4 core requires $768+3n$ cycles to test a key and $n = 8$ was used, a single RC4 key is tested in 792 cycles ($15\mu S$). Hence the average encryption time when all 96 cells operate in parallel is 165ns.

An optimized software implementation of the RC4 algorithm was used to compare the speed of the RC4 key search engine with that of a contemporary microprocessor. The key is generated and stored in memory and the size of expected pseudo random bit stream was 8 bytes. The speed measurements (for 1000 encryptions) only consider the computation time and involve no I/O operations. The GNU GCC compiler v2.9 was used to compile the program source using the '-O3' optimization flag. The speed of the microprocessor based implementation is compared with that of the FPGA implementation in Table 7.6. The 50MHz FPGA implementation is approximately 60 times faster than the 1.5GHz Pentium 4 implementation.

Table 7.7 shows the time required to search a complete 40-bit and 56-bit RC4 key space. Since FPGA chips with more logic resources and faster clock rate are

Platform	Frequency MHz	Time us	Normalized Time
Sun Ultra IIi	400	49456	299
SGI R12000A	400	11318	68.6
Intel P4	1500	9618	58.3
This work	**50**	**165**	**1**

Table 7.6: RC4 Encryption Speed on Different Platforms.

Platform	40-bit key hours	56-bit key years
Sun Ultra IIi	15084	113007
SGI R12000A	3451	25861
Intel Pentium 4	1361	10269
This work	**50**	**377**

Table 7.7: Time required for an RC4 key search.

already available, the performance of the FPGA RC4 key search engine can be further improved. A Xilinx XCV3200E has double the number of block RAMs, and the XC2V8000 can contain 672 RC4 engines.

7.5 BBS Random Number Generator

7.5.1 Size

The size of the design is quite small that it uses less than 3% of the logic resources in the FPGA chip. Table 7.8 is the report on hardware sources used by the complete 1024-bit design including interface to host.

7.5.2 Speed

The `TRACE` tool reports minimum clock period to be 9.648ns. To simplify the design, 100MHz clock is used for this implementation. The frequency of the external

Name of resource	count	Utilization
External GCLKIOBs	2	50%
BLOCKRAMs	2	2%
SLICEs	347	3%

Table 7.8: Device utilization summary.

clock is variable and independent of the design throughput.

For a n-bit design, n^2 clock cycles are used for a single multiplication. The product will be $2n$ bits. Subtracting from the MSB to compute $X^2 \; mod \; M$ requires n iterations. Assuming half of the iterations require only subtraction and shifting while the other half of them require an extra addition (recover process), a *mod* operation requires $3.5n^2$ clock cycles. After the *mod* operation, the registers must be reinitialized so another n clock cycles are needed.

To generate a value in the random sequence, $4.5n^2+n$ clock cycles are required, where n is the size of the modules in term of bit. For a 256-bit design, 4719616 clock cycles are required. That means a 1024-bit random value is generated every 47.2ms. Since only the last 10 bits are used as random data, the throughput of the design is 211bps.

7.5.3 External Clock

The randomness of the RNG depends on the frequency fluctuation of the external clock [RRK98]. The mean frequency of the external clock does not affect the randomness when there is a large gap between frequencies of the two clocks (2 to 3 degree of orders in this design). The peak-to-peak voltage of the external clock is 3.45 ± 0.05V. The duty cycle of the clock various from 51% to 54% for high and low frequencies respectly. There is no trival frequency drift in time domain. The most important characteristic of the external clock is the frequency variation. The frequency variation ranges in $\pm 15Hz$.

7.5.4 Random Performance

The experimental results are tested using the NIST test suite (version 1.4) [A. 01] and the Diehard Random Test [Mar02]. The hardware RRNG and the BBS PRNG were tested independently.

For the NIST test suit, the test sequences were 1M bits in size. This size is larger than the usual 20000 bits since some of the tests (e.g. Random Excursions, etc.) require more then 10^6 bits data for a single pass. The sample size, i.e. the number of bit sequences to pass the tests is 50. The hardware RNG performance under different external clock frequencies are presented in Table 7.9. Table 7.10 shows the tests applied to the RNG outputs and the input parameters used. The significant level α was chosen to be 0.01. If the calculated *P-value* is larger than 0.01, the test is passed. All these parameters were set according to the recommendation in the NIST documents [A. 01].

This result indicates that the random sequences from both the RRNG and PRNG can pass all the tests applied. In both cases, there are failed results for some patterns in Aperiodic-Template test. But this will only affect the pass rate of the test.

In the Diehard test suit, all random sequences generated by the BBS PRNG can pass all the tests. For the hardware RRNG, only the sequences generated for external clock frequencies lower than 3kHz can pass the test. Using an external clock with frequency higher than this value will result in failure in all tests. The reason is that the noise variation in frequency decreases with increasing frequency and hence the randomness is affected.

7.6 Summary

This chapter presented the results for the implemented designs and compared them with other software and hardware implementations. The results of these implementations were summarized in Table 7.11. In each case the FPGA implementation was

Ext clk (Hz)	pass all tests	
	RNG	PRNG
225	YES	YES
500	YES	YES
750	YES	YES
1k	YES	YES
2k	YES	YES
25k	YES	YES
50k	YES	YES
75k	YES	YES
100k	YES	YES
250k	YES	YES
500k	YES	YES
750k	YES	YES
1M	YES	YES

Table 7.9: RNG test results (NIST).

Name of Test	Parameters
Frequency	N/A
Block Frequency	blk=10500
Cumulative Sums	N/A
Runs	N/A
Longest Run of Ones	N/A
Rank	N/A
Discrete Fourier Transform	N/A
Nonperiodic Template Matchings	blk=9
Overlapping Template Matchings	blk=9
Universal Statistical	blk=7
	ini.=1280
Approximate Entropy	blk=5
Random Excursions	N/A
Random Excursions Variant	N/A
Serial	blk=5
Lempel-Ziv Complexity	N/A
Linear Complexity	blk=500

Table 7.10: Input parameters for NIST test.

Design	Measurement	Our Performance	Software
IDEA cipher	through put	592Mbps	5.9Mbps
Montgomery Multiplier	through put (1024-bit design)	256Mbps (radix-2^{16})	11.4Mbps
RC4 key search	Time to search 40-bit key space	50 hours	377 hours
RRNG	through put	250kpbs	-
PRNG	through put (1024-bit design)	211bps	-

Table 7.11: Performance summary. The software implementations are all based on an Intel P4 1.5GHz PC and compiled by GCC (v2.95.3) with '-O3' enabled. The RSA and IDEA speeds are reported by OpenSSL (v0.9.6c).

significantly faster than a software implementation on an Intel Pentium 4 1.5GHz PC. Apart from the speed advantages, FPGAs also offer benefits in terms of footprint and power consumption over microprocessors which are important in embedded applications. The parameters affecting the performance were also presented. The performance of the designs are measured with throughput and area consumption. The results of the test suits applied to the RNG were listed.

Chapter 8

Conclusion

Through design examples, this thesis illustrated that the FPGA platform is suitable for building high performance hardware cryptographic systems. The FPGA designs can adapt various algorithms in various architectures. Several problems have been addressed:

Parallel and Serial Trade Off

The parallel implementation of the IDEA block cipher and the serial implementation of the BBS PRNG are two extreme examples of parallel and serial architectures. The parallel structure was improved by deep pipelining and increasing the utilization of hardware. The throughput of the improved design was 592Mbps. The serial implementation of BBS was extremely small, using less than 3% of an FPGA chip.

Tradeoffs between parallel and serial extremes are important. Within the given resources, developers always want to achieve the highest performance. The variable radix Montgomery multiplier implementation offers the flexibility of evaluating the tradeoff. By synthesizing designs using different radixes, a developer can choose an optimal radix for his/her design based on the area and timing constraints. Without these measurements, designers can only estimate the performance of the systolic design based on experience, or they may implement different designs to compare the results. This work provides an efficient yet accurate way for designers to predict the performance of a systolic Montgomery multiplier on an FPGA platform. The results show that the larger the radix, the faster the design. However, area also increases with radix and the area requirements exceed the resources available on an XCV1000 device for radixes larger than 2^{16}.

High Performance Parallel Structure

This research also evaluated a parallel computing structure on the system level. The RC4 implementation presented in this thesis had a massively parallel structure in which 96 RC4 cores are placed on a single FPGA chip. Actually more cores can be used since the current design uses less than 50% of the logic area. This structure is about 58 times faster than a software implementation on a Pentium 4 1.5GHz CPU. The parallel architecture proposed in this thesis shows that an FPGA design can have much higher performance than a general purpose processors.

Modern microprocessor have a higher clock frequency than FPGAs, however, there are two large limitations in microprocessor implementations: low memory bandwidth and less processing units. The memory units in FPGA chips are divided in to small pieces and scatted all around the chip. Each of these units has their own I/O channel and can work independently, while most microprocessor systems have only one memory channel which in controlled centrally. Speed improvement in microprocessor system depends on the caching and pre-fetching. If data cannot fit in the cache and the process of the data is so simple that the processing time are shorter than the data accessing time, the microprocessor will waste a lot of time in idle waiting. On the other hand, FPGA design can fully utilize the on chip memories through dedicate data processing units. The superscaler and pipelined architecture in a microprocessor which can achieve parallelism to some degree. However, this is limited by data dependencies and the number of ALUs. The number of processing units in the RC4 FPGA design is much higher than that of a microprocessor. Since all the cores have their own data and logic, data dependencies do not affect the parallelism.

Improved Security and Efficiency

The increasing needs of high quality and efficient random number generator raised from various fields including business, academic, consumer products. Since many cryptography systems use FPGA chips as hardware accelerator, a build in random number generator is definitely a requirement. In the design presented, the random seed is generated by sampling the frequency of an external clock. For a sufficiently slow external clock, which has large amounts of jitter, the resulting random numbers can pass the stringent DIEHARD test. The BBS PRNG is one of the most secure pseudo random number generators and is suitable for cryptography related applications. The proposed serial design has a particularly small area utilization.

FPGA designs can offer sufficient computing power for today's cryptographic applications. The design architecture can be varied to adapt new algorithm or different design constraints. From high performance cryptanalysis systems to small size serial RNG, the adaptability and usability of FPGAs in cryptographic applications have been shown.

8.1 Future Development

The examples presented in this thesis can be integrated together to form a complete hardware cryptographic solution. By modifying open source security applications such as OpenSSL or integrating to the system level security protocols such as PAM (Pluggable Authentication Modules), this cryptography system on a chip can offer improvements both in speed and security. Finally, the reconfigurability of FPGA chips makes it possible to prepare all possible cryptography protocols, but only those currently used by the system need be downloaded to the hardware, saving logic resources and hence cost.

Bibliography

[A. 01] A. Rukhin, el. *A Statistical Test Suit For Random and Pseudorandom Number Generators for Cryptographic Applications*. NIST Special Publication 800-22, 2001.

[Ado02] Adobe System Inc. *Add security to PDF documents*, 2002. http://www.adobe.com/epaper/tips/acr5secure/pdfs/acr5secure.pdf.

[And01] Ross Anderson. *Security Engineering - a Guide to Building Dependable Distributed Systems*. John Wiley & Sons, 2001.

[ARV95] W. Aiello, S. Rajagopalan, and R. Venkatesan. Design of practical and provably good random number generators. In *Proceedings of the Sixth Annual ACM-SLAM Symposium on discrete Algorithm*, pages 1–9, 1995.

[Asc99a] Ascom. *IDEACrypt Coprocessor Data Sheet*, 1999. http://www.ascom.ch/infosec/downloads/IDEACrypt_Coprocessor.pdf.

[Asc99b] Ascom. *IDEACrypt Kernel Data Sheet*, 1999. http://www.ascom.ch/infosec/downloads/IDEACrypt_Kernel.pdf.

[AT93] C. Adams and S. Tavares. Designing s-boxes for ciphers resistant to differential cryptanalysis. In *Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography, Rome, Italy*, pages 181–190, 1993.

[BCF$^+$91] H. Bonnenberg, A. Curiger, N. Felber, H. Kaeslin, and X. Lai. VLSI implementation of a new block cipher. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computer and Processors*, pages 501–513, 1991.

[BK02] Eli Biham and Paul C. Kocher. *Known Plaintext Attack on the PKZIP Stream Cipher*, 2002. ftp://ripem.msu.edu/pub/crypt/docs/kocher-pkzip-attack.txt.

[Bor97] J. Borst. Differential-linear cryptanalysis of IDEA. ESAT–COSIC Technical Report 96–2, Department of Electrical Engineering, Katholieke Universiteit Leuven, February 1997.

[BP99] T. Blum and C. Paar. Montgomery modular exponentiation on reconfigurable hardware. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 70–77, April 1999.

[CBK91] A. V. Curiger, H. Bonnenberg, and H. Kaeslin. Regular VLSI architectures for multiplication modulo $2^n + 1$. *IEEE Journal of Solid-State Circuits*, 26(7):990–994, July 1991.

[CBZ$^+$93] A. Curiger, H. Bonnenberg, R. Zimmerman, N. Felber, H. Kaeslin, and W. Fichtner. VINCI: VLSI implementation of the new secret-key block cipher IDEA. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 15.5.1–15.5.4, 1993.

[DH76] W. Diffie and M.E. Hellman. New directoins in cryptography. *IEEE Transactions on Information Theory 22*, pages 644–654, 1976.

[Ele98] Electronic Frontier Foundation. *Cracking DES*. O'Reilly, 1998.

[ElG85] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory 3*, pages 469–472, 1985.

[GA99] M. George and P. Alfke. *Linear Feedback Shift Registers in Virtex Devices*. Xilinx, Inc., August 1999. Application Note XAPP210, Version 1.0.

[GBP01] A. Gerosa, R. Bernardini, and S. Pietri. A fully integrated 8-bit, 20 mhz, truly random numbers generator, based on a chaotic system. In *SSMSD. 2001 Southwest Symposium on Mixed-Signal Design*, pages 87–92, 2001.

[GSB$^+$00] S. C. Goldstein, H. Schmit, M. Budiu, M. Moe, and R. R. Taylor. PipeRench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, April 2000.

[GW96] I. Goldberg and D. Wagner. Architectural considerations for cryptographic hardware. *http://www.cs.berkeley.edu/∼iang/issac/hardware/*, 1996.

[HL94] M. Hellman and S. Langford. Differential-linear cryptanalysis. In *Advances in Cryptology, Proceedings of Eurocrypt 1994*, pages 26–36, 1994.

[HMC89] P.D. Hortensius, R.D McLeod, and H.C. Card. Parallel random number generation for VLSI systems using cellular automata. *IEEE Transactions on Computers*, 38(10):1466–1473, Oct. 1989.

[hot02] *HotBits: Genuine random numbers, generated by radioactive decay*, 2002. http://www.fourmilab.ch/hotbits/.

[HTSH00] M.K. Hani, Siang Lin Tan, and N. Shaikh-Husin. FPGA implementation of RSA public-key cryptographic coprocessor. In *Proceedings of TENCON 2000*, volume 2, pages 6–11, 2000.

[Knu81] D. Knuth. *The Art of Computer Programming: Vol. 2, Seminumerical Algorithms*. Addison-Wesley, 1981.

[Knu95] L. R. Knudsen. Truncated and higher order differentials. In *Proceedings of the Second International Workshop on Fast Software Encryption*, pages 196–211, 1995.

[Kor93] P. Kornerup. High-radix modular multiplication for cryptosystems. In *11th Symposium on Computer Arithmetic*, pages 277–283, Jul 1993.

[Kor94] P. Kornerup. A systolic, linear-array multiplier for a class of right-shift algorithms. *IEEE Transactions on computers*, 43:892–898, Aug 1994.

[KWH99] P.D. Kundarewich, S.J.E. Wilton, and A.J. Hu. A CPLD–based RC4 cracking system. In *IEEE Proceedings of the Canadian Conference on Electrical and Computer Engineering*, pages 397–401, 1999.

[LCTL00] M. P. Leong, O. Y. H. Cheung, K. H. Tsoi, and P. H. W. Leong. A bit-serial implementation of the international data encryption algorithm (IDEA). In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 122–131, April 2000.

[Lip98] Helger Lipmaa. IDEA: A cipher for multimedia architectures. In *Selected Areas in Cryptography '98*, pages 253–268, August 1998.

[LM90] X. Lai and J. Massay. A proposal for a new block encryption standard. In *Advances in Cryptology, Proceedings of Eurocrypt 1990*, pages 389–404, 1990.

[LMM86] L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on computing*, 15(2), 1986.

[LMM91] X. Lai, J. Massay, and S. Murphy. Markov ciphers and differential cryptanalysis. In *Advances in Cryptology, Proceedings of Eurocrypt 1991*, pages 17–38, 1991.

[Mar02] George Marsaglia. *DIEHARD: a battery of tests for random number generators*, 2002. http://stat.fsu.edu/ geo/diehard.html.

[MH78] R.C. Merkle and M.E. Hellman. Hiding information and signatures in trapdoor knapsacks. *IEEE Transactions on Information Theory 24*, pages 525–530, 1978.

[Mic02] Microsoft Co. *Office XP Document Security*, 2002. http://www.microsoft.com/Office/techinfo/enterprisezone/itcolumn07.asp.

[MMF98] O. Mencer, M. Morf, and M. J. Flynn. Hardware software tri-design of encryption for mobile communication units. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 5, pages 3045–3048, May 1998.

[Mon85] P. Montgomery. Modular multiplication without trial division. In *Mathematics of Computation*, volume 44, pages 519–521, Apr 1985.

[MvOV01] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 5th edition, 2001.

[MZ91] C. Meier and R. Zimmerman. A multiplier modulo $(2^n + 1)$. Diploma thesis, Institut für Integrierte Systeme, ETH, Zürich, Switzerland, February 1991.

[Net02] Netscape Communications Co. *The SSL Protocol*, 2002. http://wp.netscape.com/eng/ssl3/draft302.txt.

[Nyb96] K Nyberg. Generalized feistel networks. *Advances in Cryptology, Asiacrypt'96 Springer-Verlag*, pages 91–104, 1996.

[Ope02] The Open Group. *The Single UNIX Specification, Version 2*, 2002. http://www.opengroup.org/onlinepubs/7908799/xsh/crypt.html.

[PC00] Craig S. Patrie and J. Alvin Connelly. A noise-based ic random number generator for applications in cryptography. In *IEEE Transactions on Circuits and Systems – I: fundamental Theory and Application*, volume 47, pages 615–621, 2000.

[PMO$^+$01] P.H.W. Leong, M.P. Leong, O.Y.H. Cheung, T. Tung, C.M. Kwok, M.Y. Wong, and K.H. Lee. Pilchard – a reconfigurable computing platform with memory slot interface. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2001.

[Ram89] R. Ramaswamy. Application of a key generation and distribution algorithm for secure communication in open systems interconnection architecture. In *Security Technology, 1989. Proceedings. 1989 International Carnahan Conference on, 1989*, pages 175–180, 1989.

[RAR02] RAR Lab. *RAR Archiver*, 2002. http://www.rarlab.com.

[Req02a] Request For Comments. *The Kerberos Network Authentication Service (V5)*, 2002. http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc1510.html.

[Req02b] Request For Comments. *OpenPGP Message Format*, 2002. http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc2440.html.

[Req02c] Request For Comments. *Security Architecture for the Internet Protocol*, 2002. http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc2401.html.

[RRK98] R.C. Fairfield, R.L. Mortenson, and K.B. Coulthart. An LSI Random Number Generator (PRN). *Springer-Verlag*, pages 203–230, 1998.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. In *Communication ACM*, volume 21, pages 120–126, Feb 1978.

[RSA99] RSA Labs. DES III Challenge. *http://www.rsa.com/rsalabs/des3/*, 1999.

[RSA00] RSA Labs. *FAQ*, 2000. http://www.rsasecurity.com/rsalabs/faq/index.html.

[SAF98] S. L. C. Salomao, V. C. Alves, and E. M. C. Filho. HiPCrypto: A high-performance VLSI cryptographic chip. In *Proceedings of the Eleventh Annual IEEE ASIC Conference*, pages 7–11, 1998.

[Sch93] B. Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *Proceedings of 1st Internal Workshop on Fast Software Encryption, Springer-Verlag*, pages 191–204, 1993.

[Sch96] B. Schneider. *Applied Cryptography*. John Wiley & Sons, second edition, 1996.

[sPK01] Toni stojanovski, Johnny Pil, and Ljupco Kocarev. Chaos-based random number generators. Part II: practical realization. *IEEE Transactions on Circuits and Systems – I: fundamental Theory and Application*, 48(3):382–385, March 2001.

[STCS01] Barry Shackleford, Motoo Tanaka, Richard J. Carter, and Greg Snider. FPGA implementation of neighborhood-of-four cellular automata random number generators. Technical report, HP Labs Technical Reports, 2001.

[SV93] M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In *Proceedings., 11th Symposium on Computer Arithmetic*, pages 252 –259, 1993.

[TSW00] Wei-Chang Tsai, C.B. Shung, and Sheng-Jyh Wang. Two systolic architectures for modular multiplication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(1):103 –107, Feb. 2000.

[TT00] A. Tiountchik and E. Trichina. Modular exponentiation on fine-grained fpga. In *Proceedings. 13th Symposium on Integrated Circuits and Systems Design*, pages 139 –143, 2000.

[TTTM02] T. Rinne, T. Ylonen, Tero Kivinen, and M Saarinen Sami. *SSH Authentication Protocol.* Network Working Group, Interrnet-Draf, Internet Engineering Task Force (IETF), 2002.

[VMD98] F. Montoya Vitini, J. Monoz Masque, and A. Peinado Dominguez. Bound for linear complexity of BBS sequences. In *Electronics Letters*, volume 34, pages 450–451, 1998.

[VV84] U. Vazirani and V. Vazirani. Efficient and secure pseudo-random number generation. In *Advances in Cryptology – CRYPTO '84, Lecture Notes in Computer Science No. 196, Springer-Verlag*, pages 193–202, 1984.

[Wal93] Colin D. Walter. Systolic modular multiplication. *IEEE Transactions on Computers*, 42(3):376–378, March 1993.

[Wal97] Colin. D. Walter. Space/time trade-offs for higher radix modular multiplication using repeated addition. *IEEE Transactions on Computers*, pages 139–141, Feb 1997.

[WIF01] Robert K. Watkins, Jason C. Isaacs, and Simon Y. Foo. Evolvable random number generators: A schemata-based approach. In *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 469–473, 2001.

[WMSL95] S. Wolter, H. Matz, A. Schubert, and R. Laur. On the VLSI implementation of the international data encryption algorithm IDEA. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, volume 1, pages 397–400, 1995.

[Wol86] S. Wolfram. Random sequence generation by cellular automata. *Advances in Applied Mathematics*, 7:123 – 169, 1986.

[WSW$^+$99] Che-Han Wu, Ming-Der Shieh, Chien-Hsing Wu, Ming-Hwa Sheu, and Jia-Lin Sheu. A VLSI architecture of fast high-radix modular multiplication for RSA cryptosystem. In *ISCAS '99. Proceedings of the 1999 IEEE International Symposium on Circuits and Systems*, volume 1, pages 500–503, 1999.

[Xil99] Xilinx, Inc. *Xilinx Libraries Guide*, 1999.

[Xil00a] Xilinx. *Using the Virtex Block SelectRAM+ Features*, 2000. Applications Note XAPP130.

[Xil00b] Xilinx, Inc. *Xilinx Coregen Reference Guide*, 2000. Version 3.1i.

[Xil02a] Xilinx Inc. *Virtex-E Extended Memory: Detailed Functional Description*, 2002.

[Xil02b] Xilinx Inc. *Virtex-II: Detailed Functional Description*, 2002.

[ZCB$^+$94] R. Zimmermann, A. Curiger, H. Bonnenberg, H. Kaeslin, N. Felber, and W. Fichtner. A 177Mb/sec VLSI implementation of the international data encryption algorithm. *IEEE Journal of Solid-State Circuits*, 29(3):303–307, March 1994.

[ZH01] Huang Zhun and Chen Hongyi. A truly random number generator based on thermal noise. In *Proceedings of 4th International Conference on ASIC*, pages 862–864, 2001.

Publications

Full Length Conference Papers

- K.H. Tsoi, K.H. Lee and P.H.W. Leong: A Massively Parallel RC4 Encryption Engine, FCCM 2002;

- K.H. Tsoi, O.Y.H. Cheung and P.H.W. Leong: A Variable-Radix Systolic Montgomery Multiplier, FCCM 2002;

- O. Y. H. Cheung, K. H. Tsoi, Philip Heng Wai Leong, M. P. Leong: Tradeoffs in Parallel and Serial Implementations of the International Data Encryption Algorithm IDEA. CHES 2001: pp. 333-347 2001

- M.P. Leong, O.Y.H. Cheung, K.H. Tsoi, and P.H.W. Leong, A Bit-Serial Implementation of the International Data Encryption Algorithm IDEA", Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa Valley, California USA, pp. 122-131, 2000