DESIGN AND IMPLEMENTATION OF ENCRYPTION ALGORITHMS IN A COARSE GRAIN RECONFIGURABLE ENVIRONMENT

JASON P. RHINELANDER

Your file Votre référence
ISBN: 0-612-93054-8
Our file Notre référence
ISBN: 0-612-93054-8

DESIGN AND IMPLEMENTATION OF ENCRYPTION ALGORITHMS IN A COARSE GRAIN RECONFIGURABLE ENVIRONMENT

BY

JASON P. RHINELANDER

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

FACULTY OF ENGINEERING AND APPLIED SCIENCE

MEMORIAL UNIVERSITY OF NEWFOUNDLAND

2003

MASTER OF ENGINEERING THESIS

OF

JASON P. RHINELANDER

APPROVED:

thesis Committee

Major Professors _____

_____

_____

DEAN OF THE SCHOOL OF GRADUATE STUDIES

MEMORIAL UNIVERSITY OF NEWFOUNDLAND

2003

Abstract

In early 2000, Chameleon Systems Incorporated and Memorial University formed a research agreement to evaluate the viability of the Chameleon Systems CS2112 Reconfigurable Communications Processor (RCP) for use in implementing popular cryptographic algorithms. The CS2112 has a coarse grain reconfigurable architecture, capable of run time reconfigurability.

The benefit of coarse grain reconfigurable architectures is that they can offer many of the flexibilities found in software, such as reprogrammability and ease of modification to implementation, while giving performance advantages of speed and hardware encapsulation.

This research involves examining the implementation characteristics of two popular symmetric key block ciphers, RC5 and RC6, and two popular cryptographic hash algorithms, MD5 and SHA-1 with respect to the CS2112.

RC5 was designed as an iterative loop and then expanded to provide a parallel pipeline to maximize the usage of the reconfigurable fabric. RC6 was designed as an iterative loop and a pipeline. For both hash algorithms, initial designs were drafted and performance figures were estimated from experience gained through simulation and testing on a CS2112 development board.

By implementing these algorithms, the architecture of the CS2112 was evaluated for its suitability for cryptographic applications. Moreover, the reconfigurable fabric of the CS2112 was evaluated with respect to its support for the primitive operations that are required for cryptographic algorithms.

The conclusions of this research and reccomendations for future research are directly related to resource use on the CS2112. In particular, support for control and datapath logic, memory space, and global communication resources within the CS2112 were all design constraints. More specifically, it would be advantageous to have direct support for accessing memory without using datapath resources. Also hardware support for data dependent logical rotations and unsigned integer multiplications would greatly save resource usage and increase performance. Finally the design process for the CS2112 was sometimes time intensive and cumbersome, especially with respect to layout and placement of reconfigurable logic. Advances in the area of automatic placement and layout for coarse grain primitives would benefit the design process for the CS2112 greatly.

Acknowledgements

I would like acknowledge the sources of help that I have received while pursuing my Masters' degree in Electrical Engineering. I would like to thank my supervisors, Dr. Howard Heys, and Dr. Ramachandran Venkatesan, not only for their superb guidance throughout my work, but for introducing me to the field of cryptography and hardware design. I would also like to thank Dr. Mark Rollins from Chameleon Systems Incorporated for his technical support and guidance.

This thesis would not have been possible without the sources of funding from: The School Of Graduate Studies at Memorial University, Dr. Heys and Dr. Venkatesan, Chameleon Systems Inc., and the Government of Newfoundland and Labrador.

I would like to thank numerous colleagues and friends for their assistance and support throughout the duration of my Masters research. I would like to thank my parents for their constant support and guidance in all my endeavours. My colleague Andrew Cook for his help and a "fresh view on things". My special friend Cindy O'Driscoll, for all her proofreading and grammatical expertise.

Contents

Abstract ii

Acknowledgements iv

Acknowledgements iv

Table of Contents v

List of Tables ix

List of Figures x

List of Abbreviations xiii

1 Introduction 1

1.1 Motivation for Research 5

1.2 Research Scope 6

1.3 Thesis Outline 7

2 Block Ciphers, Hash Functions and Applications 8

2.1 Symmetric Key Block Ciphers 8

2.1.1 Electronic Codebook Mode (ECB) 10

2.1.2 Cipher-block Chaining Mode (CBC) 10

2.2 Hash Functions 11

2.3 Description of an HMAC 12
2.4 The RC5 Block Cipher 15
2.5 The RC6 Block Cipher 16
2.6 The MD5 Algorithm 18
2.7 The SHA-1 Algorithm 21
2.8 An Example Application: The IP Security Protocol (IPSec) 24
2.8.1 Authentication Header Protocol 25
2.8.2 Encapsulating Security Payload Protocol 26
2.9 Concluding Remarks 28

3 Hardware Architecture **29**
3.1 Software vs. Hardware Algorithms 29
3.2 Application-Specific Integrated Circuits 30
3.2.1 Designs and Performance 32
3.3 Field-Programmable Gate Arrays 33
3.3.1 Implementations and Performance 35
3.4 Coarse Grain Reconfigurable Architecture 36
3.4.1 Survey of Existing Technologies 41
3.4.2 Cryptographic Applications 42
3.5 Chameleon Systems CS2112 43
3.5.1 CS2112 High Level Architecture Description 44
3.5.2 CS2112 Data Path Unit 46
3.5.3 CS2112 Local Store Memory 48
3.5.4 CS2112 Multiplier 48
3.5.5 CS2112 Control Logic Unit 50
3.5.6 Design Process For The CS2112 Fabric 51

4 Symmetric Block Cipher Design and Implementation **55**

4.1 Diagram Use 55

4.2 RC5 Designs 57

4.2.1 RC5 Simple Iterative Design 57

4.2.2 Two Half-Round, Full Slice Version of RC5 64

4.2.3 Full Fabric RC5 Design 68

4.2.4 Summary of RC5 Results 69

4.3 RC6 Designs 70

4.3.1 Unsigned 32-bit Integer Multiplication 70

4.3.2 Iterative RC6 Design 73

4.3.3 Pipeline Primitives 75

4.3.4 Pipelined Multiplication 78

4.3.5 RC6 Full Pipelined Design 81

4.3.6 Summary of RC6 Designs 87

4.4 Summary 87

5 Evaluation of Message Digest Algorithms **89**

5.1 MD5 Implementation 89

5.1.1 Performance and Usage Estimates for MD5 91

5.2 SHA-1 Implementation 93

5.2.1 Recursive Array Expansion 93

5.2.2 SHA-1 Auxiliary Function Design 95

5.2.3 Full SHA-1 Datapath 96

5.2.4 Performance and Resource Usage of SHA-1 99

5.3 Comparison of SHA-1 and MD5 Implementations 99

5.4 Summary 100

6 Summary and Conclusions **101**

6.1 Summary of Results 101

6.2 CS2112 Architectural and Support Features 102

6.3 Considerations For Future Work 103

List of References **105**

Appendices **A-1**

A Sample Verilog Code for Selected Modules **A-1**

A.1 RC5 Testbench A-1

A.2 Iterative RC5 Top Level Module A-1

A.3 Iterative RC5 Controller Module A-2

A.4 Iterative RC5 Datapath Module A-6

A.5 Unsigned Integer Multiplier Module Controller A-12

A.6 Unsigned Integer Multiplier Module Datapath A-14

A.7 Verilog Testbench For Controlling RC5 Iterative Pipeline A-19

B ANSI C Code for Select Implementations **B-2**

B.1 RC5 C Code For Testing B-2

B.2 RC6 C Code For Testing B-4

List of Tables

3.1 Results from AES candidates in ASIC technology. 32

3.2 Simulation results from RC6 ASIC designs using 128 bit keys. 33

3.3 Some results of FPGA simulations of AES candidates. 35

3.4 Some results of FPGA implementations of RC6. 36

3.5 Survey of existing coarse grain reconfigurable technologies. 41

4.1 Resource usage for the simple iterative version of RC5. 63

4.2 Timing information for the simple iterative version of RC5. 64

4.3 Resource usage for the two half-round design of RC5. 67

4.4 Timing for the full slice design of RC5. 68

4.5 Resource use for the full fabric version of RC5 (control logic excluded). 69

4.6 Resource estimates for a single slice of RC6 in the fabric. 75

4.7 Timing information for the RC6 pipeline. 84

4.8 Resource usage for a fully pipelined RC6. 85

4.9 Control logic resource usage for a fully pipelined RC6 design. 86

4.10 Summary of block ciphers on the CS2112. 88

5.1 Resource usage for preliminary MD5 implementation. 93

5.2 Delay through MD5 datapath. 93

5.3 Resource utilization of SHA-1. 99

5.4 Summary hash algorithms on the CS2112. 100

6.1 Summary of designs on the CS2112. 101

List of Figures

1.1 Description of encryption with respect to a data network. 3

2.1 Block diagram of secure communications. 9

2.2 ECB mode. 10

2.3 CBC mode. 11

2.4 Operation of HMAC. 14

2.5 Flow diagram of the RC5 block cipher. 16

2.6 Illustration of a simplified 8-bit left data dependent rotation. 16

2.7 Flow diagram of the RC6 block cipher. 17

2.8 Procedure of initial processing arbitrary length message. 18

2.9 Looking into the H_MD5 function. 19

2.10 Operations involved in a single step of H_MD5. 20

2.11 Procedure of processing arbitrary length message. 21

2.12 A decomposition of the H_SHA1 function. 22

2.13 A decomposition of a step in the H_SHA1 function. 23

2.14 AH packet format. 26

2.15 ESP packet format. 27

3.1 A view of the ASIC design process. 31

3.2 Abstracted internal FPGA structure. 34

3.3 Flexibility verses performance of hardware technologies. 37

3.4 Examples of 2D mesh connections. 38

3.5 An example of a linear array configuration. 38

3.6 An example of a crossbar configuration. 39
3.7 An example of a KressArray configuration. Multiple communication schemes between processing elements are used. 39
3.8 Chameleon cipher chip, designed for encryption. 42
3.9 Process of swapping active and background fabrics. 43
3.10 High level block diagram of CS2112 components 44
3.11 High level decomposition of reconfigurable fabric. 45
3.12 Communication arrangement between processing elements. 46
3.13 Datapath unit block diagram. 47
3.14 Multiplier block diagram. 49
3.15 CLU communication and interaction with processing elements. 50
3.16 Design flow for the CS2112. 52
3.17 Screen capture of the graphical floorplanner. 53
4.1 Examples of configured DPU structures. 56
4.2 Two methods for describing memory structures containing one LSM and one DPU. 57
4.3 Abstracted block digram of simple iterative RC5 design 59
4.4 Structural diagram of data dependent rotation. 61
4.5 C~SIDE floorplanner screen shot of simple iterative RC5 design. . . . 63
4.6 High level abstraction of the two half-round design of RC5. 65
4.7 Four DPU implementation of the data dependent rotation. 66
4.8 Screen capture of the two half-round RC5 fabric function. 67
4.9 Screen capture of the full fabric RC5 implementation. 69
4.10 Creating a 32-bit unsigned integer multiplier. 71
4.11 Iterative multiplier setup. 73
4.12 Need of delay in a pipeline. 77
4.13 First-in, first-out queue setup. 78

4.14 Pipelined multiplier module. 79
4.15 Multiplier and controller interaction. 81
4.16 Fixed logical rotation by five bits. 82
4.17 One full round of RC6. 83
4.18 Description of control and datapath interaction. 85
4.19 RC6 pipeline floorplan. 86
5.1 Implementation of F and G functions. 90
5.2 Implementation of H and I functions. 91
5.3 A proposed MD5 datapath for one step of H_MD5. 92
5.4 Recursive expansion of $W[0..15]$ to $W[0..79]$. 94
5.5 Auxiliary function implementation. 96
5.6 Auxiliary function implementation. 97
5.7 SHA-1 datapath design. 98

AES Advanced Encryption Standard

AH Authentication Header

ALU Arithmetic Logic Unit

ARC Argonaught RISC Core

ASIC Application-Specific Integrated Circuit

CBC Cipher-block Chaining Mode

CLB Configurable Logic Block

CLU Control Logic Unit

CMOS Complementary Metal Oxide Semiconductor

CSM Control Store Memory

DES Data Encryption Standard

DMA Direct Memory Access

DPU Data Path Unit

DSA Digital Signature Algorithm

DSP Digital Signal Processing

ECB Electronic Codebook Mode

ESP Encapsulating Security Payload

FIR Finite Impulse Response

FPGA Field-Programmable Gate Array

FSM Finite State Machine

HDL Hardware Description Language

HMAC Hashed Message Authentication Code

ICV Integrity Check Value

IOB Input Output Block

IP Internet Protocol

IPSec IP Security Protocol

LSM Local Store Memory

MAC Message Authentication Code

MD Message Digest

MSP Message Security Protocol

NESSIE New European Schemes for Signatures, Integrity, and Encryption

NIST National Institute of Standards and Technology

PE Processing Element

PGP Pretty Good Privacy

PIO Programmable Input Output

PLA Programmable Logic Array

RCP Reconfigurable Communications Processor

RTL Register Transfer Level

S-HTTP Secure Hypertext Transfer Protocol

SA Security Association

SRB State Register Block

SSL Secure Sockets Layer

VPN Virtual Private Network

Chapter 1

Introduction

It is hard to imagine today's society without modern communication systems. While it is a necessity for people to communicate with each other to share information, the way in which this is undertaken has changed dramatically since the advent of global voice and data networks.

The Internet has grown at an exponential rate in the last decade. Not only are people using the Internet as a medium for communication, but the transmission and storage of sensitive data has also seen increased usage. Online banking is a good example of both the transmission and storage of sensitive data. A person must transmit their account number along with a password to access their personal information and accounts stored on the bank's computers. By the year 2007 it is estimated that 30% of Americans will use online banking and in the salary range of $50,000-$75,000, usage will be 50% [1]. With respect to the Internet economy, quarterly growth figures between 1999 and 2000 were estimated to be between 20% and 30% [2]. Corportations are utilizing Virtual Private Networks (VPNs) to connect remote locations to the private infrastructure of the company network.

While modern communication through global networks has increased communication efficiency, it has also become easy for people to intercept data traveling across shared networks such as the Internet. For example, a packet sniffer is a system that

looks at traffic flowing across a network so that a third party can view private information. A packet sniffer is a common way to obtain user IDs and passwords [3].

Cryptography can be used to provide security to information flowing through a publicly accessible network. Most cryptographic applications lie in the digital world of computers, but cryptography has a much older past. Given below are some interesting facts about the history of cryptography [4]:

- The first known emergence of a cryptographic substitution cipher occurred around 1900 B.C. in a town called Meneu Khufu, near the river Nile. Some unique hieroglyphic symbols were used in place of normal ones.
- Ancient Egyptians used substitutions of hieroglyphs, and the use became more popular with the increasing occurrence of tombs.
- Mechanical encryption devices were used extensively in World War II for the encryption of military and political messages.
- Some of the first mechanical computers were invented and used by Marian Rejewski to break codes produced from the German WWII Enigma machine [5].

The use of data networks has been increasing at an astonishing rate and with this growth in use, there is a need to secure private information across the network. Within the scope of data security, encryption plays a large role. Encryption provides data confidentiality, but there is also a need for the following security services [6]:

- Access Control: Maintains privileged access to information.
- Data Integrity: Prevents unauthorized modification of information.
- Authentication and Replay Prevention: Verifies a sender's identity and prevents unauthorized re-transmission of information.

- Scalable Key Management: Allows the secure deployment of cryptographic keys.
- Accountability and Non-repudiation: Maintains the identity of sender and prevents deniability.

The support of data integrity within a data network prevents the modification of data while it is in transit across the network. A data integrity service over a network often uses a hash function to produce a Message Digest (MD) of a message. A digital signature technique such as the Digital Signature Algorithm (DSA) also uses a hash algorithm in its operation [7]. Figure 1 is a typical hierarchy with respect to encryption in a data network.

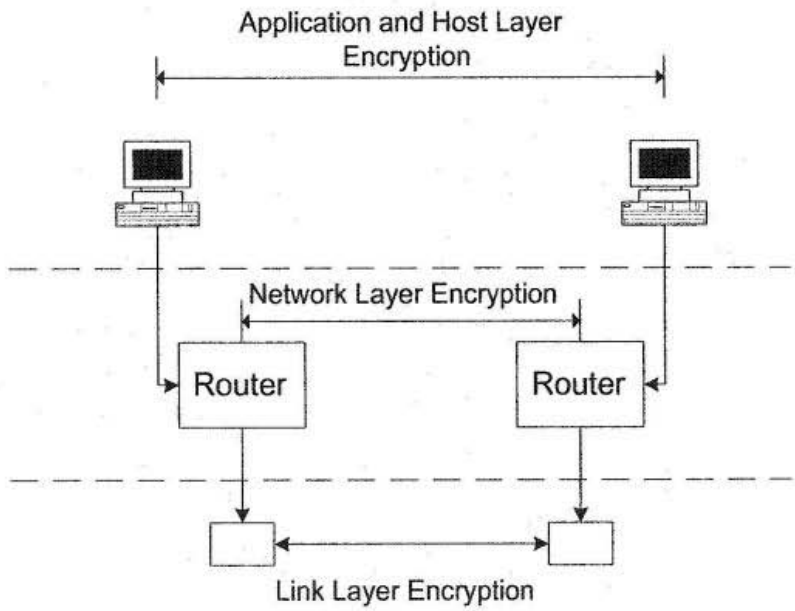

Figure 1.1: Description of encryption with respect to a data network.

Encryption algorithms, or ciphers, can be implemented in hardware or software. Some examples of popular secure communication protocols that are targeted for software encryption are: Secure Hypertext Transfer Protocol (S-HTTP), Pretty Good Privacy (PGP), Message Security Protocol (MSP), Secure Sockets Layer (SSL) and

IP Security Protocol (IPSec). Software encryption implementations are slower than hardware implementations. Hardware implementations are used in high bandwidth, low latency environments such as the link layer of a data network [6]. In addition to potentially higher performance compared to software, hardware based encryption can often be more secure than software. It is harder for an attacker to obtain information about the cipher during operation [8].

There are various ways of implementing encryption algorithms in hardware. One way is through a special cryptographic processor that can be configured for various algorithms. One example can be found in the CryptoManiac device [9]. CryptoManiac is a cryptographic co-processor and is designed to speed software encryption. Broadcom offers two cryptographic co-processors (BCM5840/41) that work at the host level to aid encryption speeds of software [10]. Another way to use hardware encryption is through an Application-Specific Integrated Circuit (ASIC) or Field-Programmable Gate Array (FPGA) technology.

The field of ASICs is a broad one. ASICs can be full custom integrated circuits or semi-custom. A full-custom ASIC engineer will design some or all of the logic, circuits and layout for a particular chip. In most application areas full-custom ASICs are not as popular as they once were, but they are growing in the area of integrated analog and digital ASICs [11]. Semi-custom ASICs are designed using standard cells that provide functionality as simple as logic gates to the level of complexity of microprocessor cores. Once the design is simulated and laid out it can be fabricated. The fabrication of an ASIC involves the masking of layers of silicon similarly to standard integrated circuits. Once the design is fabricated it cannot be changed.

An ASIC implementation is specifically designed for the cipher(s) of choice and has the main advantage of speed. Another advantage of ASICs is that the designer has complete control over placement, and is limited only by the design rules imposed by the fabrication process. Cost, design time and lack of flexibility are some

disadvantages of ASIC design.

FPGAs are a more flexible way of designing algorithms in hardware. FPGAs were developed initially as a fast way to prototype cells to be used in ASIC designs. Since then FPGAs have grown in size and capability allowing designers to implement market products in FPGAs. FPGAs contain programmable logic devices that are set by anti-fuse or static random access memory technology. The matrix of programmable logic cells are connected together by a network of wires allowing communication between cells [12].

An FPGA solution is considered to be a fine grain reconfigurable solution and will allow faster design cycles because designs can be re-burned or reconfigured without restarting the whole design process as with ASICs. A disadvantage of FPGA implementation is that routing can have overhead and can be problematic [13].

Newer coarse grain architectures are emerging to exploit the advantages of hardware while simultaneously offering the flexibility of software. Run-time reconfigurable processing, ease of modification, and quick turn around times in design and testing, are advantages of coarse grain architectures. Unlike an FPGA, coarse grain architectures can use datapaths that are greater than 1-bit [13].

1.1 Motivation for Research

The Chameleon Systems Inc. CS2112 RCP is a coarse grain reconfigurable architecture [14] designed for communication and Digital Signal Processing (DSP) applications. The performance of coarse grain reconfigurable architectures with respect to cryptographic applications can be dependent on resources and characteristics offered by the specific reconfigurable product. With the increased need for secure communication, commerce, faster transmission speeds, and increased traffic over public networks, there is a need for implementation of new ciphers in hardware. A survey of companies

in 2003 shows that medium to large sized businesses utilize hardware based security services over software based methods [15].

The Chameleon Systems CS2112 offers a reconfigurable environment for encryption that gives the security and performance of hardware while offering the flexibilities of software. Since the CS2112 is a general purpose communications processor with support for any of the arithmetic and logical operation found in encryption, a research agreement was developed between Memorial University and Chameleon Systems to investigate and evaluate the performance of popular ciphers on the CS2112 [16].

It is the purpose of this research to not only investigate the performance of popular encryption algorithms on the CS2112, but to investigate where the CS2112's architecture is inadequate to support these algorithms efficiently and to determine the advantages of using the CS2112 for security applications. Results of this research were reported back to Chameleon Systems Inc. for future design considerations.

1.2 Research Scope

The purpose of this thesis is to investigate the suitability of symmetric key encryption and cryptographic hash algorithms on the Chameleon Systems RCP. Two symmetric key block ciphers that were explored are RC5 [17] and RC6 [18]. Hash functions that were explored are MD5 [19] and SHA-1 [7]. Various design methods of these algorithms were addressed along with testing and performance evaluation.

Before addressing the topic of this research, the reader will be given adequate background in cryptography, hardware implementation technologies for encryption algorithms, and a high level description of the CS2112 architecture. RC5 was the first algorithm to be investigated and this focused on an iterative approach to cipher implementation. Optimizations to this design were carried out to maximize use of the CS2112. Next, RC6 was evaluated with a pipelined design optimized for high

speed operation. MD5 and SHA-1 were designed based on information gained from designing RC5 and RC6 on the CS2112. When design and testing were completed the CS2112 is evaluated for its suitability for supporting the selected algorithms with respect to its processing resources and support for cryptographic primitives.

1.3 Thesis Outline

This thesis follows the outline below:

- Chapter One is an introduction to the research conducted with the CS2112.
- Chapter Two will introduce the reader to symmetric key block ciphers, cryptographic hash functions, and give an example of a popular protocol that makes use of both types of algorithms.
- Chapter Three will focus on different hardware implementation technologies while giving some examples of cryptographic applications and performance. The CS2112 is also introduced in Chapter Three outlining its architecture and target application areas.
- Chapter Four describes implementation and performance of symmetric key block ciphers on the CS2112.
- Chapter Five describes the design and performance of hash functions on the CS2112.
- Chapter Six provides conclusions with respect to the CS2112 and gives some recommendations on coarse grain architectural support for cryptography in relation to the CS2112.

Chapter 2

Block Ciphers, Hash Functions and Applications

The purpose of this chapter is to give an overview of symmetric key block ciphers, hash functions, and their applications such as the IP Security Protocol. Primitive operations that ciphers and hash functions utilize also will be discussed.

2.1 Symmetric Key Block Ciphers

When a message or plaintext is encrypted using an encryption algorithm, it is computationally infeasible to extract the information from the ciphertext unless the corresponding decryption algorithm is used. Cryptology is the field that is made up of Cryptography and Cryptanalysis. Cryptography is the field that involves mapping plaintext to ciphertext in a secure fashion. The purpose of Cryptanalysis is to test the security of ciphers by decrypting encrypted messages in a method not intended by the decryption algorithm, in effect testing the security of the encryption. A block cipher is a function that mathematically maps an n-bit plaintext block to an n-bit ciphertext block, with the block size defined to be n [20].

Extra information, called a key, is required to execute an encryption algorithm. If the same key is used for both encryption and decryption it is called a symmetric

key cipher [17]. The use of a symmetric key block cipher to transmit an encrypted message is illustrated in Figure 2.1 [20].

The total possible number of keys is defined as the keyspace and the security of a cipher is related to both the keyspace and n. A cipher is unconditionally secure if ciphertext blocks and plaintext blocks are statistically independent. In general, an increase in block size and/or in size of the keyspace will increase the implementation cost of the cipher [20].

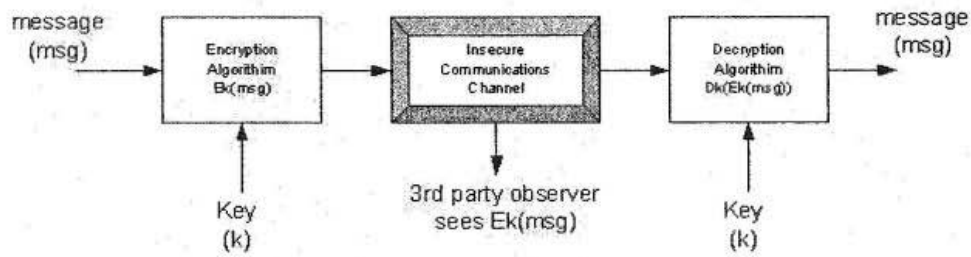

Figure 2.1: Block diagram of secure communications.

Iterated round ciphers involve a sequential loop of an internal function (a round), involving blocks of plaintext. A round consists of simple cryptographic operations such as additions and data dependent rotations. Each round uses a subkey that is derived from the original key that is mixed with the data. Virtually all block ciphers are iterated, and RC5 and RC6 operate in this fashion.

Block ciphers may be used in different modes of operation. Modes of operation arise when encrypting a message that is longer than n-bits [20]. Two modes of operation will be discussed, Electronic Codebook Mode (ECB) and Cipher-block Chaining Mode (CBC). ECB is a mode that does not involve feedback of a previous operation, while CBC requires feedback from its previous operation. For the two modes of operation discussed, a brief explanation of the advantages and disadvantages with respect to security and error recovery will be presented.

2.1.1 Electronic Codebook Mode (ECB)

ECB is illustrated in Figure 2.2. The symbol n is the block size in bits, X_i represents the i-th block of plaintext, C_i represents the i-th block of ciphertext, $e()$ represents the encryption function, $d()$ represents the decryption function, and k represents the key.

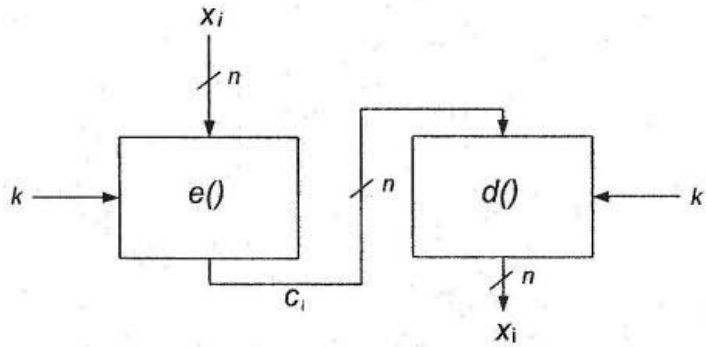

Figure 2.2: ECB mode.

When a message is more then n-bits long, it is sectioned into n-bit blocks, each block is encrypted separately, and decryption is carried out in a similar fashion. ECB mode has the advantage of being the simplest encryption mode. An error in a transmitted encrypted block will result in a full plaintext block being decrypted in error on the receiver's end. ECB mode has the disadvantage that it does not hide data patterns [20]. In ECB mode an observer can view ciphertext across an insecure network and may sometimes have knowledge of the plaintext that is being transmitted. The observer can then build a library of plaintext-ciphertext pairs allowing partial decryption of future messages.

2.1.2 Cipher-block Chaining Mode (CBC)

CBC is illustrated in Figure 2.3. CBC mode starts with an initial value or IV vector and subsequent encryptions are carried out with the use of the previous ciphertext

block. The IV vector is required because this mode uses feedback. When the first plaintext block is encrypted there is no previous ciphertext block to use, therefore a value is provided externally so that the operation can proceed. CBC has the advantage of hiding patterns in plaintext values.

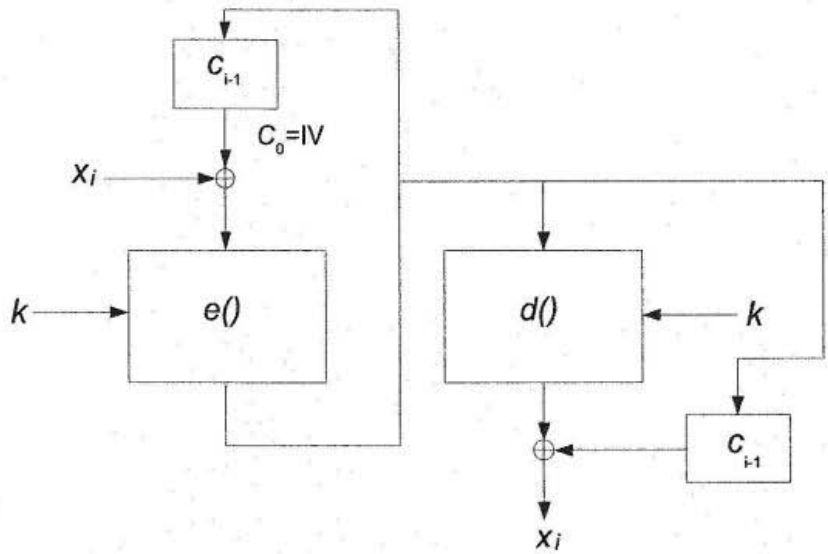

Figure 2.3: CBC mode.

Some of the disadvantages of CBC mode are that an error in the transmitted ciphertext block will result in an incorrect deciphering of C_i and C_{i+1}. In addition, the order of plaintext blocks matter because the decryption requires the receiver to have the previous block of ciphertext.

2.2 Hash Functions

Hash functions take a finite arbitrary length input and output a fixed length message digest or simply a hash of the message. In other words, a hash function will map an arbitrary ranged input to a fixed and smaller ranged output [20]. Hash functions are used in both cryptographic and non-cryptographic applications.

Cryptographic hash functions are one way functions, meaning that you cannot get the original input based solely on the output. A collision occurs when two inputs map to the same output value. The output value of a hash function is regarded as a compact digital image or representation of the input data. For cryptographic applications, a hash function must exhibit strong and weak collision avoidance. To explore the concept of strong and weak collision avoidance, the hash function will be defined as $h()$, the input to the function is x and another input to the function is x' (different than x). Strong collision avoidance is observed if it is computationally infeasible to find both x and x' such that $h(x) = h(x')$. Weak collision avoidance is observed if given x, finding x' such that $h(x) = h(x')$ is infeasible [20].

Hash functions are primarily used in data integrity schemes and may be keyed or not keyed. A keyed hash function will take two inputs (data and a secret key) and produce one output referred to as a Message Authentication Code (MAC). A hash function that is not keyed can be configured as MACs producing an Hashed Message Authentication Code (HMAC).

For the purposes of this research single input, single output hash functions used for authentication schemes will be addressed.

2.3 Description of an HMAC

HMAC is a mechanism for message authentication that utilizes a cryptographic hash function. MACs provide a way to check the integrity of information transmitted across an insecure medium [21]. The HMAC scheme uses a cryptographic hash function and a secret key. The input information to an HMAC is the message to be authenticated, and the secret key. It is assumed that only the sender and receiver has access to the secret key. To describe the operation of HMAC, the following definitions are made [21]:

- $H()$, a cryptographic hash function that processes an arbitrary length message formatted into length B-byte blocks.
- L, byte length of hash function output. It is assumed that L will be less than B.
- K, the secret key used in the HMAC. The secret key is of variable length and any length that is less than B will have zero bytes appended to the end of the key. For key $(K1)$ with length greater than B the following will occur, $H(K1) = K2$. $K2$ has a length of L, and will be used as the secret key.
- *ipad*, the value $0x36$ repeated B times.
- *opad*, the value $0x5c$ repeated B times.

Figure 2.4 illustrates the operation of an HMAC. In illustrations, adjoining blocks of data represent the appending of two blocks into a single, larger block. The $\oplus$ operator is the bit-wise XOR operation.

The construction of an HMAC provides two functions. The integrity of the message is protected because the cryptographic hash function provides a digital fingerprint of the original text. A message cannot be forged because a secret key is mixed into the HMAC. HMACs are used in the Authentication Header (AH) protocol within IPSec.

The performance of the HMAC depends on the underlying hash function and it is desired to use hash functions that will perform well in software. The main goal is to make HMACs scalable to faster or more secure hash functions in the future [21].

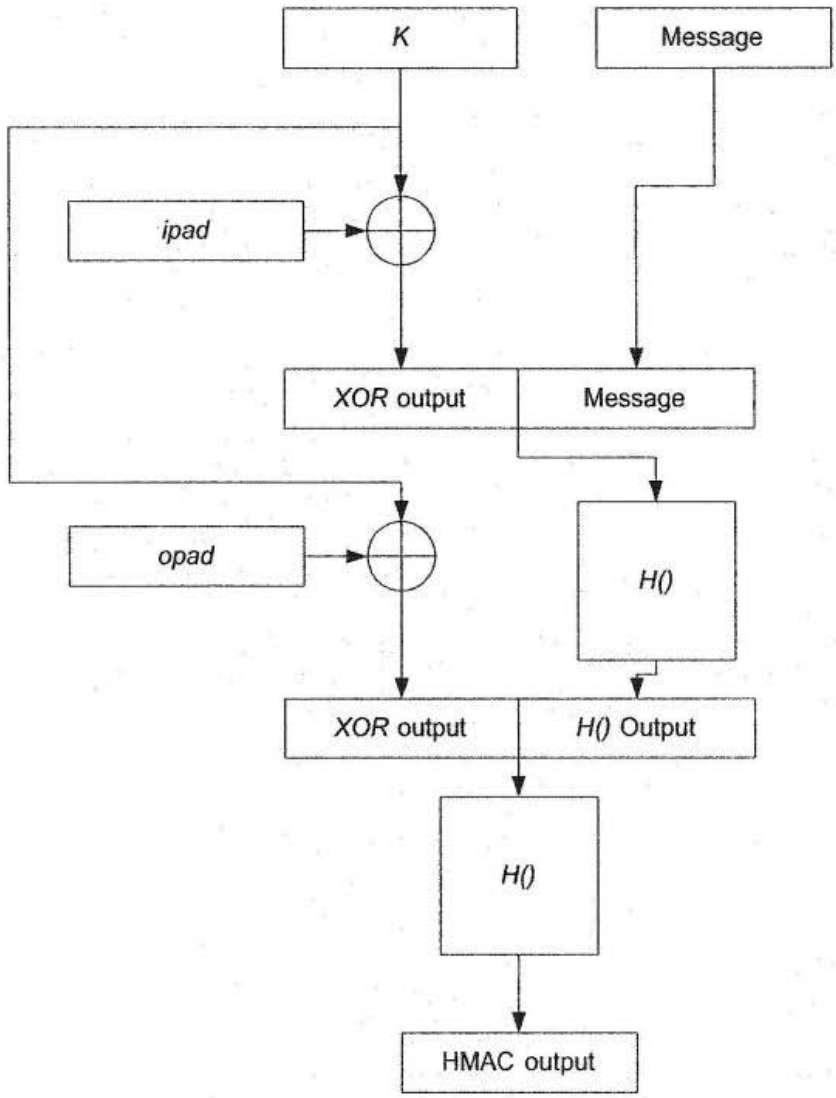

Figure 2.4: Operation of HMAC.

2.4 The RC5 Block Cipher

Ronald L. Rivest developed RC5 [17] at the MIT Laboratory for Computer Science and it is a trademark of RSA Data Security. RC5 is an extremely compact cipher and is suitable for both hardware and software implementations. Listed below are some more notable characteristics of RC5 [17]:

- RC5 is a symmetric block cipher. The same cryptographic key is used for both encryption and decryption. The ciphertext and plaintext are of fixed bit length.
- RC5 uses operations and instructions that are commonly found on typical microprocessors.
- RC5 is iterative and can have a variable number of rounds.
- RC5 uses little memory so that it is useful with smart cards, mobile computing platforms, micro controllers, and other low memory environments.
- RC5 makes use of data dependent rotations as one of its diffusion primitives.
- RC5 is parameterized as RC5 - $w/r/b$. The word size is defined as w and the block size is $2w$. The number of iterative rounds is given by r, and b specifies the key size in bytes. For this research RC5-32/12/16 will be used.

The RC5 cipher is illustrated in Figure 2.5. The parameters A and B are 32-bit blocks of plaintext. The array $S[0..2r+1]$ is composed of 32-bit words that are created by manipulation of the initial key. The $+$ operation is mod 2^{32} addition and L $\lll$ R is the data dependent bit-wise left rotation of L by the amount R. To further illustrate the process of a data dependent left rotation, a simplified operation is given in Figure 2.6.

The bit-wise XOR operation is defined as $\oplus$. The i parameter is used to indicate which round the algorithm is in: for example, the statement repeat $i = 1..r$ is the

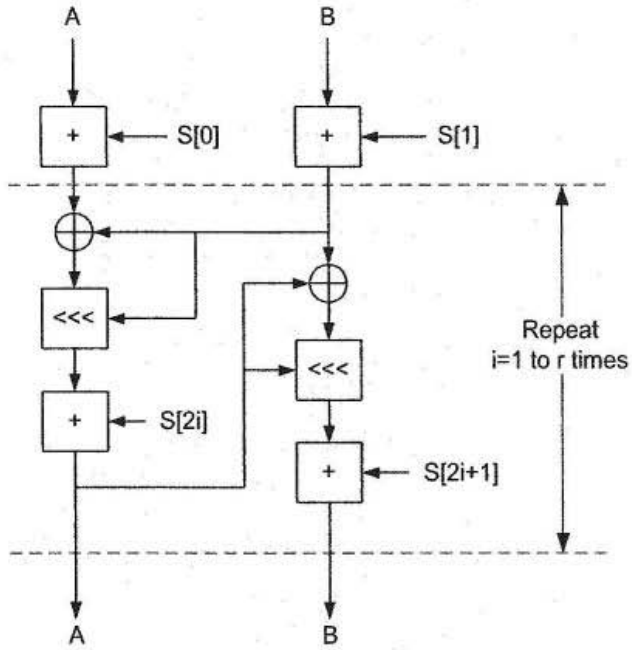

Figure 2.5: Flow diagram of the RC5 block cipher.

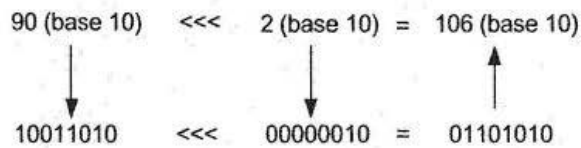

Figure 2.6: Illustration of a simplified 8-bit left data dependent rotation.

equivalent of a loop where the variable i is incremented by one each round. Therefore in the second round of RC5, $S[4]$ and $S[5]$ are used in the algorithm.

2.5 The RC6 Block Cipher

RC6 was a submission to the National Institute of Standards and Technology (NIST) for consideration as a candidate for the Advanced Encryption Standard (AES) in 1998 (Rijndael was chosen to be the algorithm for AES [22]). RC6 was also considered for the New European Schemes for Signatures, Integrity, and Encryption (NESSIE)

specification, but did not make it through to the final round of selections due to ongoing intellectual property rights issues [23]. RC6 is a direct evolution of RC5.

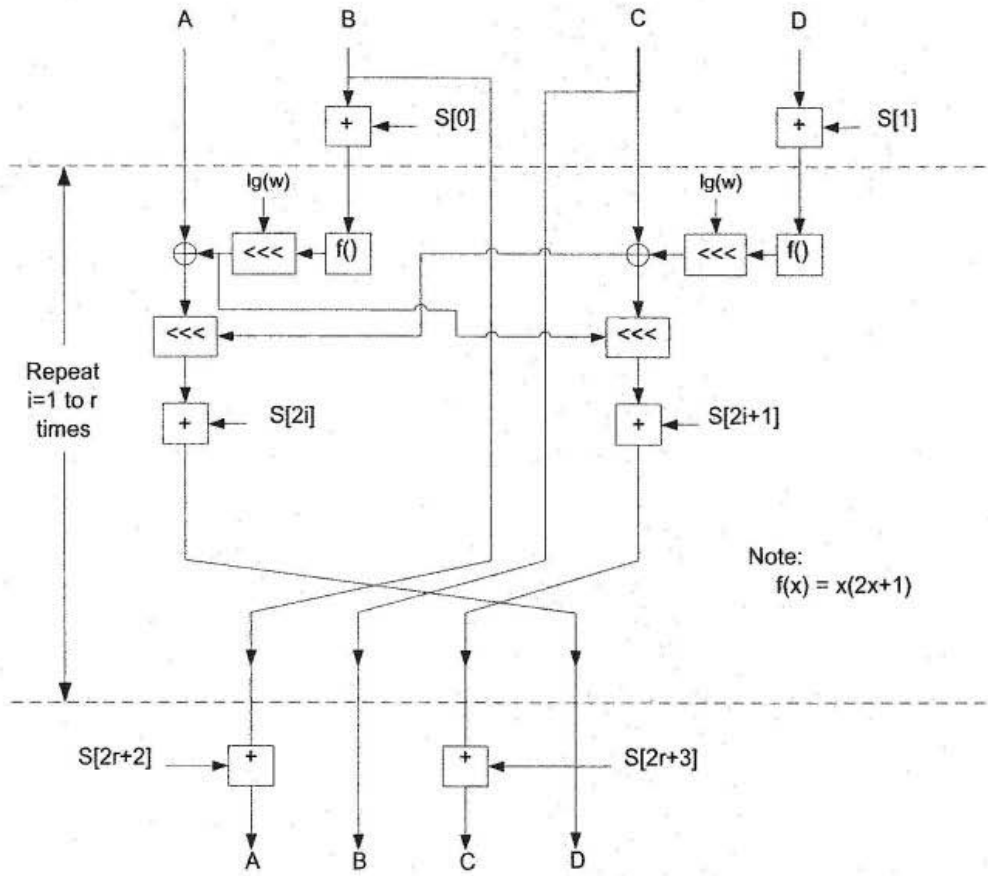

Figure 2.7: Flow diagram of the RC6 block cipher.

Since RC6 is an advancement of RC5 there are various similarities. RC6 is parameterizable like RC5 with the same parameter format, namely, RC6-$w/r/b$. All of the operations that are used in RC5 are also found in RC6. The operation of RC6 is shown in Figure 2.7 [18]. From the flow diagram the differences in RC6 are evident. There is a left rotation by $lg(w)$, where $lg()$ is the $log_2()$ operation. The use of four w-bit input blocks of plaintext denoted as A, B, C and D make RC6 a 128-bit block cipher when w=32. There is a permutation of the data blocks at the end of each

round. The biggest difference, especially from the viewpoint of performance is the operation $f()$, which represents the following relationship, $f(x) = x(2x+1) \bmod 2^{32}$. Hence, $f()$ requires an unsigned integer multiplication operation. For this research, RC6-32/20/16 will be used.

2.6 The MD5 Algorithm

MD5 is a cryptographic hash function that takes a message of arbitrary length and produces a 128-bit message digest. MD5 exhibits weak and strong collision avoidance, and is a popular hash algorithm that has found much use in Internet based message authentication [19].

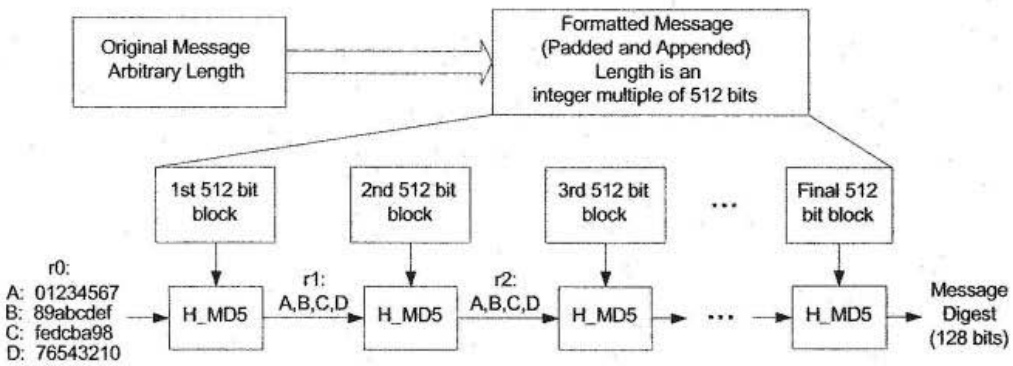

Figure 2.8: Procedure of initial processing arbitrary length message.

Once the message of arbitrary length has been formatted into N 512-bit blocks as shown in Figure 2.8, it is passed into the compression algorithm denoted as H_MD5. The inputs to the compression function are one 512-bit block of the formatted arbitrary length message, and four initialized registers, A, B, C, and D. The outputs of the function are the modified values of the registers that were used by the input. These four registers act as temporary buffers for subsequent calls to the compression function. The values $r0, r1, r2...rN-1$ are used to denote the state of the four registers before and after a call to the compression function where $N-1$ would be the

last call to H_MD5. When each block of the message is processed in this fashion, the values of the registers make up the 128-bit message digest or hash value [19].

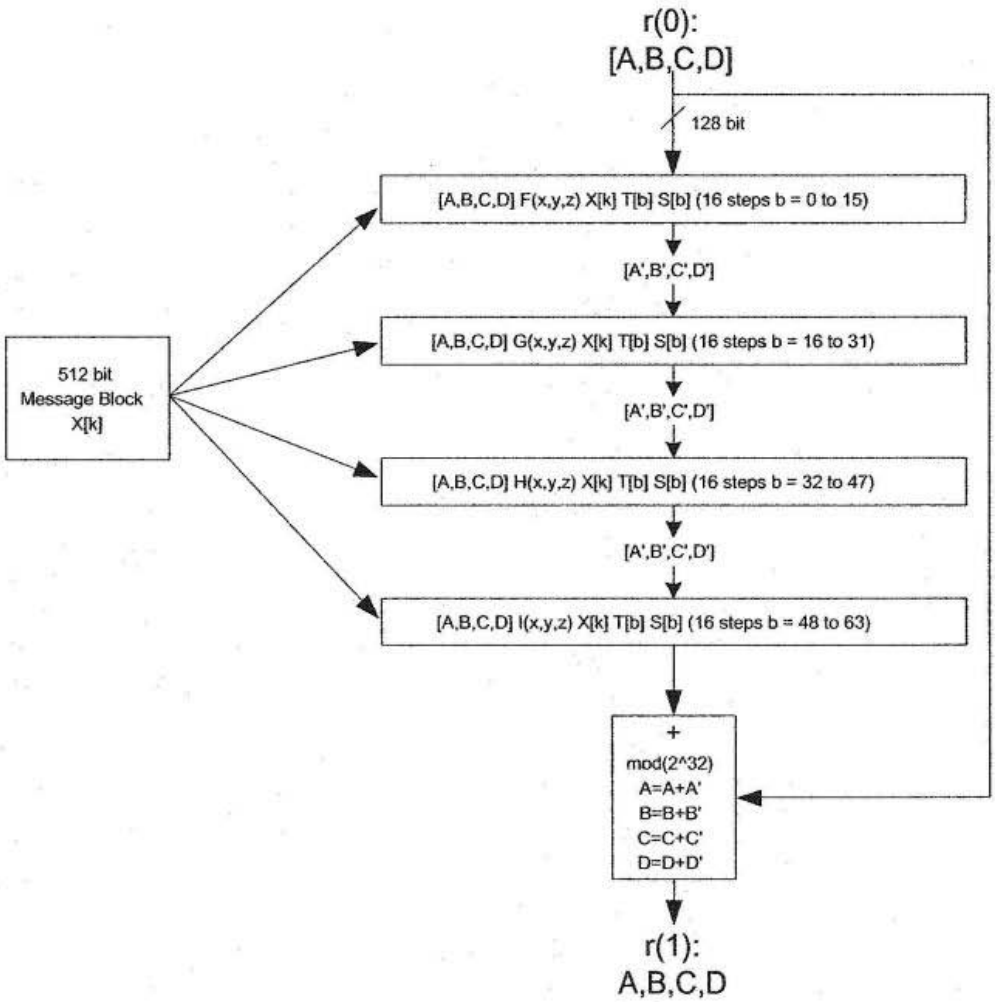

Figure 2.9: Looking into the H_MD5 function.

There are 64 steps involved with the execution of H_MD5, as shown in Figure 2.9. Each grouping of 16 steps are abstracted into a block showing inputs, outputs, and data values involved. The F,G,H and I functions denote auxiliary functions that take in three 32-bit values and output a single 32-bit value. These operations are solely made up of bit-wise operations (AND, OR and XOR). The registers are re-assigned

to during the execution represented by a superscript " ' ". The final operation is to add the old values of the 32-bit registers to the new modified values.

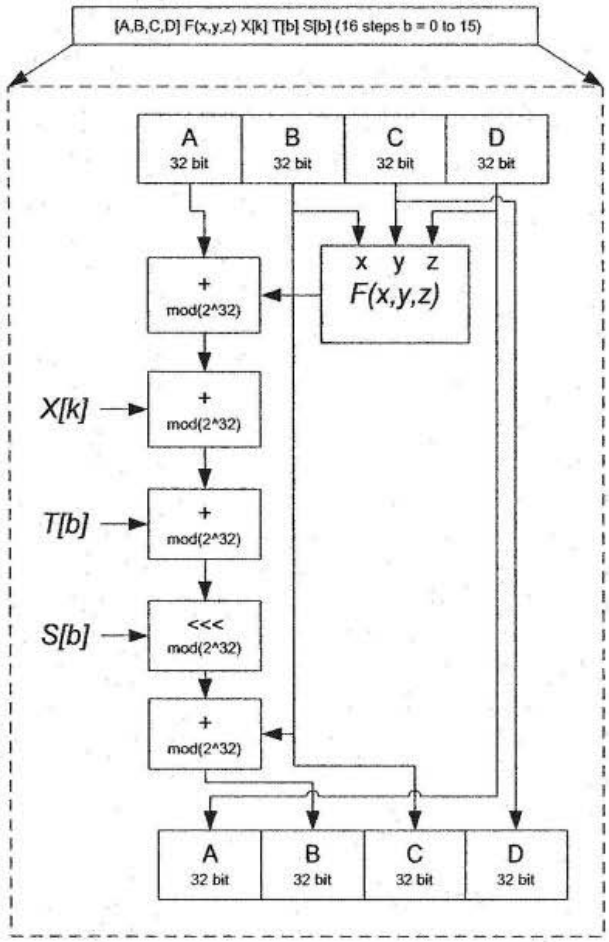

Figure 2.10: Operations involved in a single step of H_MD5.

The array $T[0..63]$ is an array of 32-bit values derived from the $sin()$ function. The array $S[0..63]$ contains values for the data dependent rotation operation. The array $X[0..15]$ contains 32-bit words of the 512-bit message block that was passed into the compression function. The mapping of b to k is accomplished by using a permutation mod 16 operation, therefore elements of X are used and reused during the execution of H_MD5.

2.7 The SHA-1 Algorithm

SHA-1 [7] is a hash algorithm used in digital signature schemes and for HMACs within IPSec [24]. SHA-1 uses primitives from MD4, and there are similarities between MD5 and SHA-1. SHA-1 generates a 160-bit message digest from a message of arbitrary length [7] and this process is illustrated in Figure 2.11.

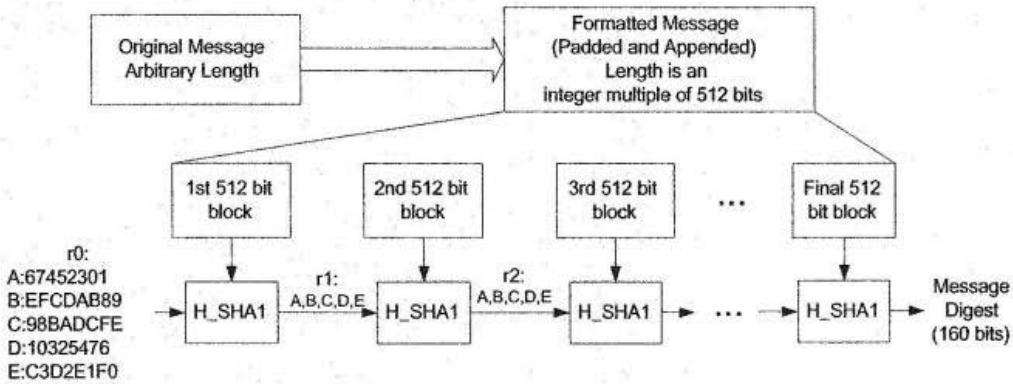

Figure 2.11: Procedure of processing arbitrary length message.

The compression function is defined as H_SHA1. With each call to the compression function, five 32-bit registers are used as input, labeled A,B,C,D, and E. A 512-bit block of the arbitrary length message is also used as an input to the compression function. The output of the function are the five registers modified from their original values and they are used as the input to the next call to H_SHA1. Once all of the message blocks are processed the final value of the registers make up the 160-bit message digest [7].

The compressing function is decomposed in Figure 2.12. The 512-bit block is stored in the 32-bit array element $W[0..15]$, and is expanded to a 80 element array of 32-bit values. The process of expanding W is recursive and only depends on W. There are 80 steps in total for running the compression function. The 80 steps have been further divided into groups of 20 steps. Each one of these sub-groupings is illustrated as a rectangle with input, output and other parameters listed inside. The

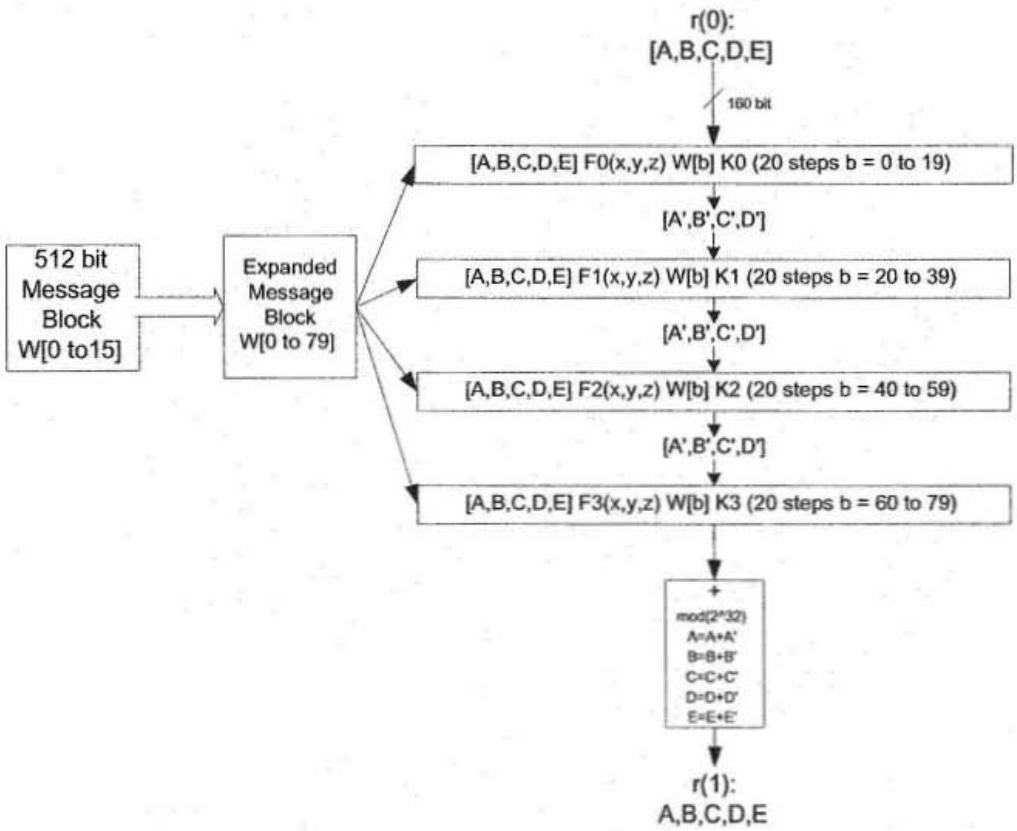

Figure 2.12: A decomposition of the H_SHA1 function.

functions $F0$,$F1$,$F2$, and $F3$, take as input three 32-bit values and output a single 32-bit value using bit-wise operations. The K parameter is a 32-bit constant that is used for that particular grouping of steps. The operation of the first block is further decomposed in Figure 2.13 to show the operations that take place in each step. The specific operations involved are addition, rotation, and bit-wise logical operations contained in $F0$ [7]. The other three groupings use $F1$, $F2$, and $F3$ respectively.

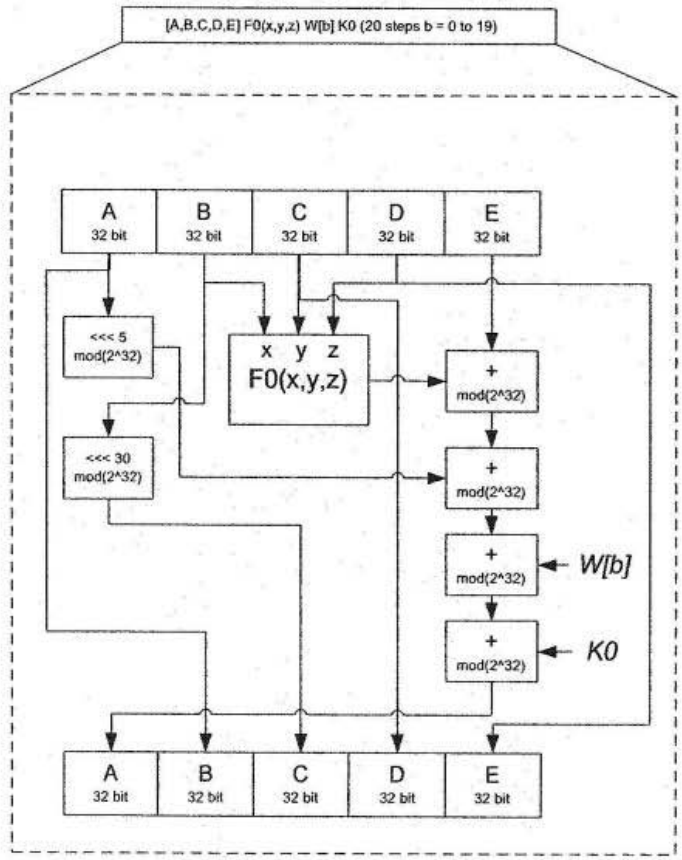

Figure 2.13: A decomposition of a step in the H_SHA1 function.

For the purposes of this research it is important to look directly at the implementation of the H_SHA1 function.

2.8 An Example Application: The IP Security Protocol (IPSec)

The following section is an overview of how symmetric key block ciphers and hash algorithms are used in popular applications to provide general security to communications over the Internet. IPSec is the proposed standard with respect to the security architecture for the Internet protocol (IP) [25]. IPSec allows the implementation of VPNs through the establishment of secure tunnels [26]. The biggest advantage of a VPN over the Internet is that businesses can abandon private dial-in and leased communication lines in favor of using more popular public connection methods. Partners, suppliers, and customers can be seamlessly connected over a private network that is based upon a public medium [26]. IPSec will be examined in terms of how and where symmetric block ciphers and hash functions are utilized. IPSec is designed to be algorithm-independent while offering a set of required algorithms for operation on different platforms [25]. IPSec provides the following functionality to IP based networks, including the Internet [25]:

- Access Control
- Connectionless Integrity
- Data Origin Authentication
- Protection Against Replay Attacks
- Confidentiality

For the purpose of this research it is important to look at how and where IPSec is implemented. IPSec can be applied to the host level (often in software) or in conjunction with an Internet gateway or router (possibly in hardware). Three implementation possibilities arise:

1. Into the native Internet Protocol (IP) implementation at the host level or at the gateway level, requiring access to IP source code [25].

2. Underneath the existing IP protocol stack referred to as "Bump-in-the-stack" or BITS. Implementation is transparent to existing architecture [25].

3. In external hardware referred to as "Bump-in-the-wire" or BITW. BITW is often used in business and military applications and is usually IP addressable. The BITW arrangement can act as a host or a security gateway (possibly both) [25]. The BITW scheme will often require high throughput and a hardware solution.

IPSec works in either a transport mode or tunnel mode. In either case there are Security Associations (SAs) associated with each connection. In the creation of SAs there are three types of protocols that work together to provide security services: Encapsulating Security Payload (ESP), AH, and various key management protocols [26].

2.8.1 Authentication Header Protocol

The role of the IP Authentication Header is to provide data origin authentication, data integrity, and protection against replay attacks. To protect against replay attacks, the receiver must check the sequence number of the incoming packets [24].

Algorithm options for the AH protocol are: HMAC MD5 or HMAC SHA-1. The protocol authenticates the entire packet with the exception of the destination address [26]. The format of the AH contains various fields as described in Figure 2.14. The field "Authentication Data" contains the computation of the Integrity Check Value (ICV). The ICV is the output of the specific HMAC algorithm used. The length of the "Authentication Data" field is variable but its length must be a multiple of

32-bits for IP version-4 or 64-bits for IP version-6 [24], while non-multiple lengths are explicitly padded [24].

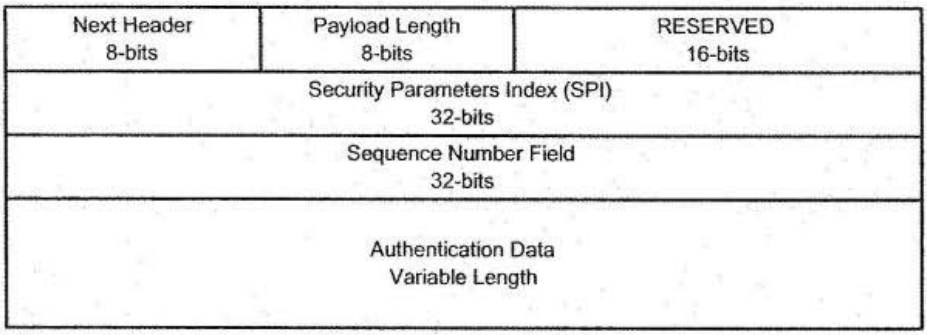

Figure 2.14: AH packet format.

HMACs are distinguished by underlying cryptographic hash functions. HMACs are keyed algorithms and they provide data origin authentication and data integrity that are dependent on the distribution of their key. In other words, if a packet is sent from one party to another, and the receiver uses its key and deems the HMAC is correct it shows two things: first that the HMAC must have been added by the sender (and is not a forgery) [27], and second, the data within the packet has not been modified. The performance of a HMAC is dependent on the performance of the underlying cryptographic hash function [27] which is within the scope of this research.

2.8.2 Encapsulating Security Payload Protocol

The ESP protocol provides confidentiality, data origin authentication, connectionless integrity, an anti-replay service, and limited traffic flow confidentiality [28]. ESP is designed to be algorithm-independent and some cipher options are: Data Encryption Standard (DES), 3DES, RC5, Blowfish, IDEA, and Cast and RC6 [26] [8]. ESP also supports optional message authentication within its protocol. To provide interoperability between different implementations the following algorithms are mandatory [28]:

- DES in CBC mode.
- HMAC MD5
- HMAC SHA-1
- Null Authentication Algorithm
- Null Encryption Algorithm

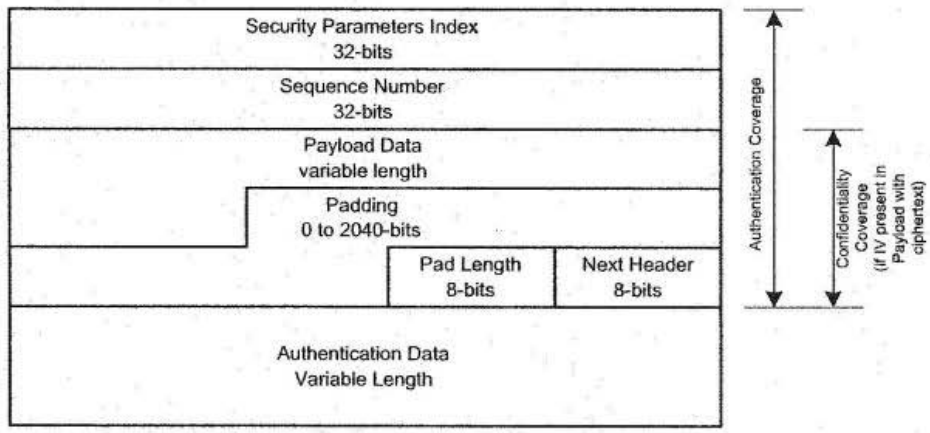

Figure 2.15: ESP packet format.

The format for the ESP packet is outlined in Figure 2.15 and is provided to illustrate the steps involved when using the ESP protocol. When encrypting an ESP packet the following steps take place [28]:

1. The original data packet is encapsulated into the ESP Payload field. If in transport mode, only the upper layer transport protocol information is encapsulated. In tunnel mode the entire data packet is encapsulated into the ESP packet.
2. The Padding field is added as required by the protocol.
3. The result is encrypted using a specified cipher in a specific mode, such as CBC. Encrypted fields are Payload Data, Padding, Pad Length, and Next Header

fields. An IV vector may or may not be added to the Payload field. This depends on the mode of operation and settings defined when the SA is created by the specific connection.

The speed of the cipher is variable depending on the type, and how it is used internally (e.g. number of rounds executed). For BITW implementations of IPSec, it is advantageous to have fast cipher designs available so that data throughput can be high.

2.9 Concluding Remarks

In this chapter, block ciphers and hash algorithms were introduced and explained in basic detail to provide the reader an understanding of some of the operations that take place when implementing these algorithms. IPSec was also described to provide a general view of why there is a need for fast hardware based ciphers and hash functions. It is important next to focus on hardware implementation methods and how they can be applied to cryptographic applications.

Chapter 3

Hardware Architecture

In this chapter, various hardware platforms for encryption algorithms and hash functions are discussed. The performance advantages of hardware verses software applications of encryption will be discussed briefly. Next ASIC, FPGA, and coarse grain technologies will be outlined and illustrated, with applications to encryption algorithms that are of interest to this research. A high level description of the Chameleon Systems CS2112 RCP is given with the design process required by the architecture.

3.1 Software vs. Hardware Algorithms

Cryptographic applications can be implemented in many different forms and across various levels within a data network. The previous chapter showed that ciphers and hash algorithms use many arithmetic operations (such as multiplication) and bit-wise manipulations (such as rotation and permutation). For these applications, specialized hardware is a faster choice than the use of software, which uses general purpose processors. For example, a 600 MHz processor is incapable of encrypting a T3 communications line (45 Mbit/s) with 3DES [29]. There is cryptographic co-processor hardware that can aid the encryption speeds of general software platforms with respect to protocols such as IPSec. For example, Broadcom's CryptoNetX line

of products offer full IPSec support at speeds up to 4.8 Gbit/s [10].

The lack of architectural support for some cryptographic operations is one reason software based encryption on general purpose processors is relatively slow compared to direct hardware implementations. It has been shown that adding architectural support to a general purpose processor has improved encryption speeds. For instance, a 59%-74% speed up in encryption was encountered by adding instruction set support for fast substitutions, general permutations, rotates and modular arithmetic [29]. Overall, ciphers that rely heavily upon substitutions and permutations (such as DES) benefit from having higher memory access times and bandwidth, while ciphers that are based heavily upon arithmetic operations (such as RC5 and RC6) benefit from architectural support for their operations, such as rotation and multiplication [29].

3.2 Application-Specific Integrated Circuits

ASICs are a technology that is applicable to a wide variety of design areas. When implementing a design using ASIC technology, a designer will use computer libraries and simulation tools typically encapsulated within a Hardware Description Language (HDL). Once simulation of the design is complete, the design enters the placement a routing phase. The placement and routing phase is highly dependent upon the technology and process used to fabricate the chip. A general design flow can be seen in Figure 3.1 [30]. Two of the most popular HDLs are Verilog and VHDL. In the simulation stages, a behavioral description of the design is created. The designer can then break each block down into smaller units, possibly to the level of transistors and gates [30]. There are many design processes that can automate certain parts of this process depending on the particular process and provided libraries. The final stage of the design process is to send the placed design for fabrication. Once the chip is fabricated it can be tested and depending on the results the whole process may need

to be repeated if re-design is required. Industry examples of ASIC designs show that it takes on average two to three repetitions of the design flow in Figure 3.1 to meet design specifications [31].

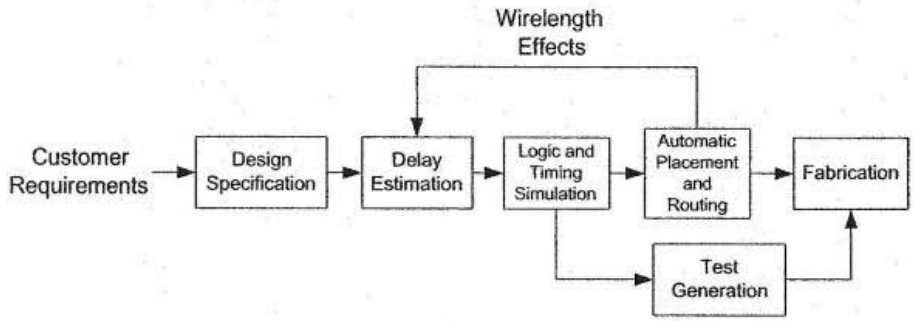

Figure 3.1: A view of the ASIC design process.

There are both advantages and disadvantages to using ASICs for implementation. The use of standard cells in a design allow the designer to be removed from some of the underlying technology that is used for fabrication.

ASICs work at higher operating speeds than other technologies and work well with integrated technologies such as analog cores, microprocessor cores, and high speed IO [32]. ASICs also use less power than other implementation methods [31]. There are industry arguments that there is a widening gap between process technology, speed of operation, and ASIC cells that make the process of ASIC design more difficult and expensive [31]. Problems arise with an increase in operating speed. As speed increases, transmission line effects start emerging into the connections between cells. With older technological processes, many of the effects could be hidden inside the standard cells provided by the manufacturer. Expansion of tool sets and tool automation are required to address this problem [31]. Long design and fabrication times are the biggest drawbacks to ASIC technology.

3.2.1 Designs and Performance

While the DES cipher is a popular algorithm to consider for implementation and performance, there are other algorithms that have been gaining much attention for their applicability to ASIC technology. There has been ongoing work with the finalists for the AES to replace DES. Of these ciphers, the Rijndael algorithm was chosen over ciphers such as RC6, IDEA, Serpent, Mars and Twofish. In the evaluation of these ciphers, simulations were conducted using Mitsubishi Electric's 0.35 micron Complementary Metal Oxide Semiconductor (CMOS) ASIC design library [33]. The AES candidates were designed and simulated to explore bottlenecks in performance with respect to ASICs, and they do not represent optimizations for performance. The figures in Table 3.1 are important for exploring what primitive operations affect the performance of popular ciphers [33].

Cipher	Performance (Gbit/s)
MARS	0.2256
RC6	0.204
Rijndael	1.95
Serpent	0.9316
Twofish	0.3941

Table 3.1: Results from AES candidates in ASIC technology.

This study also made particular mention to operations were most time consuming and had the largest effect on performance [33]. With respect to Rijndael, substitutions, unsigned addition, and bit-wise logical operations had the biggest impact. With respect to RC6, unsigned integer multiplications had the biggest impact to performance [33]. The unsigned integer multiplication within RC6 is of interest to this research.

Another investigation into the suitability of RC6 in ASIC technology was conducted [34] [35]. Two versions of RC6 were used, a pipelined version optimized for

performance, and an iterative version that allows different modes of operation. Simulations were performed using 0.5 micron CMOS technology [35], and were evaluated with respect to throughput, number of transistors used, and chip area used. The evaluations are given in Table 3.2.1 [35]. Results from the simulations suggest that the performance of RC6 was dependent on the multiplication operation.

RC6 Design	Max Speed (Gbit/s)	Transistors	Max Area (sq mm)
Iterative	0.10	450000	0.023
Pipelined	2.10	9000000	0.52

Table 3.2: Simulation results from RC6 ASIC designs using 128 bit keys.

3.3 Field-Programmable Gate Arrays

Field-programmable gate arrays (FPGAs) provide a hardware platform that can be considered a cross between software (use of programmable general purpose processing units), and specific hardware implementations (ASICs). FPGAs are an example of a reconfigurable hardware technology. Reconfigurable solutions can potentially provide a faster operating platform than software, while offering greater flexibility than ASIC technology [36].

FPGAs were conceptually designed to fit between programmable array logic (PAL) devices and maskable programmable gate arrays or MPGAs. FPGAs are like PAL devices in that they can be programmed using an electrical current. FPGAs are similar to MPGAs in that they can accommodate complex designs within a single device while utilizing arrays of logical gates [36]. Usually an FPGA is connected to a SRAM device that contains configuration bits for the design. When the configuration bits are transferred to the appropriate points on the chip, gates are configured for use depending on where the software mapper places the design within the chip's reconfigurable architecture. Overall the performance penalty for designs in an FPGA versus an ASIC are on an order of five to ten times [37].

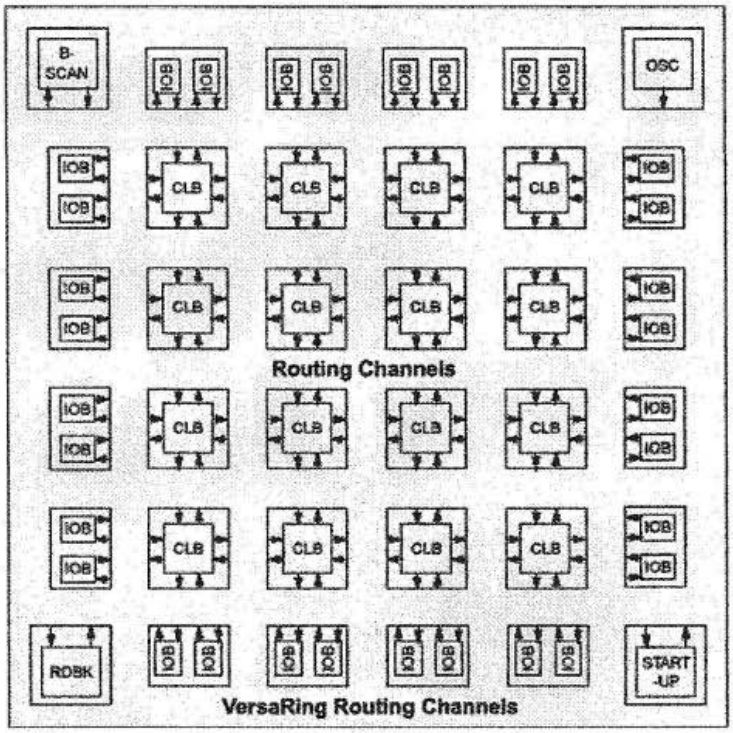

Figure 3.2: Abstracted internal FPGA structure.

The structure of the basic building blocks within a typical FPGA are illustrated in Figure 3.2 [38]. This is taken from the Spartan family FPGA data sheet from Xilinx Inc. The Configurable Logic Blocks (CLBs) illustrated are used to implement the users logic, and the Input Output Blocks (IOBs) are used to provide communication between the internal structure of the FPGA and its external pins [38]. FPGAs are deemed to be fine grain reconfigurable devices.

While FPGAs do not perform as fast as ASICs, their reconfigurable characteristics make FPGAs a very attractive platform for application. In today's marketplace the development cycle of an ASIC is often longer than the market lifetime of a product. For instance, the time required to prototype an FPGA is a few weeks while it takes months to fabricate an ASIC. Another advantage of FPGAs are that they can be reprogrammed, whereas an ASIC needs to be replaced with a new chip when design modifications are required [39].

With ASICs, a design is first simulated and synthesized for a particular technology and then the netlist is passed to an ASIC design house for physical placement and routing. All FPGA design, simulation, and implementation is done by the designer. The placement and routing stage is accomplished by software called a mapper and is dependent on the particular FPGA used. The process of mapping can be problematic depending on application needs [39].

3.3.1 Implementations and Performance

Preliminary design of AES finalists was conducted in a similar fashion to Section 3.2. The ciphers were designed to analyze potential bottlenecks in performance based on primitive operations. The results shown in Table 3.3 are without the presence of performance enhancing structures such as pipelining [40]. As shown earlier, the main bottleneck observed for RC6 was the multiplication operation and this is of specific interest with the design of RC6.

Cipher	Performance Gbit/s
Serpent	0.3394
Rijndael	0.3315
Twofish	0.1773
RC6	0.1039
MARS	0.0398

Table 3.3: Some results of FPGA simulations of AES candidates.

A detailed survey of the design of the AES candidates on FPGAs was conducted using various methodologies. The evaluations were based on the Xilinx Virtex XCV1000 device with a 40 MHz design constraint. Iterative designs of RC6 using feedback and non-feedback modes, and optimized pipelines using feedback and non-feedback modes are given in Table 3.4 [8].

The findings show how pipelining only has an advantage to performance when non-feedback modes are used, such as ECB cipher mode. During this study the authors

RC6 Design	Speed (Mbit/s)	CLBs Used (%)
Iterative Non-feedback	97.4	52.1
Iterative Feedback	97.4	52.1
Pipelined Non-feedback	2400	88.3
Pipelined Feedback	126.5	26

Table 3.4: Some results of FPGA implementations of RC6.

found that the two biggest architectural challenges for RC6 were the implementation of the multiplication and data dependent rotation operations. The authors made note that the multiplication operation is actually a squaring and an addition operation ($X(2X + 1) = 2X^2 + X$). Within an FPGA the operation can be created with an array squarer with summed partial products [8]. Summed partial products are used in a different manner for the analysis of the multiplication operation used in this research.

An evaluation of the suitability of RC6 and CAST-256 for the Xilinx XC4000 family of FPGA is also conducted [41]. This study achieved an RC6 implementation with a speed of 37 Mbit/s utilizing 91% of the resources of the device. With respect to the target device, the overall conclusions were the following [41]:

- The multiplication operation was a primary source for resource utilization.
- Performance was affected by the multiplication operation.
- Pipelining is difficult due to hardware complexity.
- Larger devices will be needed to achieve faster performance.

3.4 Coarse Grain Reconfigurable Architecture

Coarse grain reconfigurable hardware offers a new approach to designs in a hardware environment. FPGAs were defined to be fine grain architectures because their reconfigurable elements (CLBs) had datapath widths of one bit [13]. The biggest

problem with FPGAs is that their fine grain nature yields much routing overhead [13]. Figure 3.3 illustrates the flexibility vs. performance relationship between hardware technologies [13].

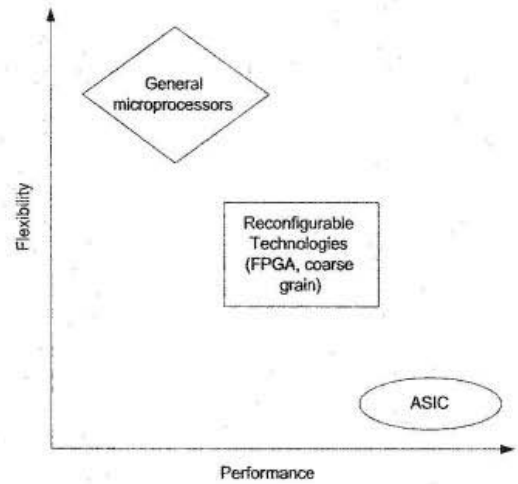

Figure 3.3: Flexibility verses performance of hardware technologies.

Coarse grain architectures commonly support Processing Elements (PEs) that have word level operations (for example, addition and subtraction may be supported within one PE). Fast reconfiguration times can allow runtime reconfiguration. Some coarse grain architectures allow multiple PEs to be combined to make larger data width PEs. Combining two 32-bit adders to make a 64 bit adder is an example of this. Coarse grain architectures allow combining PEs with much less overhead than fine grain platforms, yielding a multi-granular architecture [13].

One of the biggest design challenges of coarse grain architectures is the way in which the PEs communicate with each other. The connection of multiple PEs form what is often referred to as a "fabric". A fabric may be mesh based as shown in Figure 3.4, where PEs will have connection to their neighbors, either side by side, four ways or eight ways.

Processing elements can also be connected in linear arrays, illustrated in Figure 3.5. Within the linear array, each element is connected to its neighbour in a line,

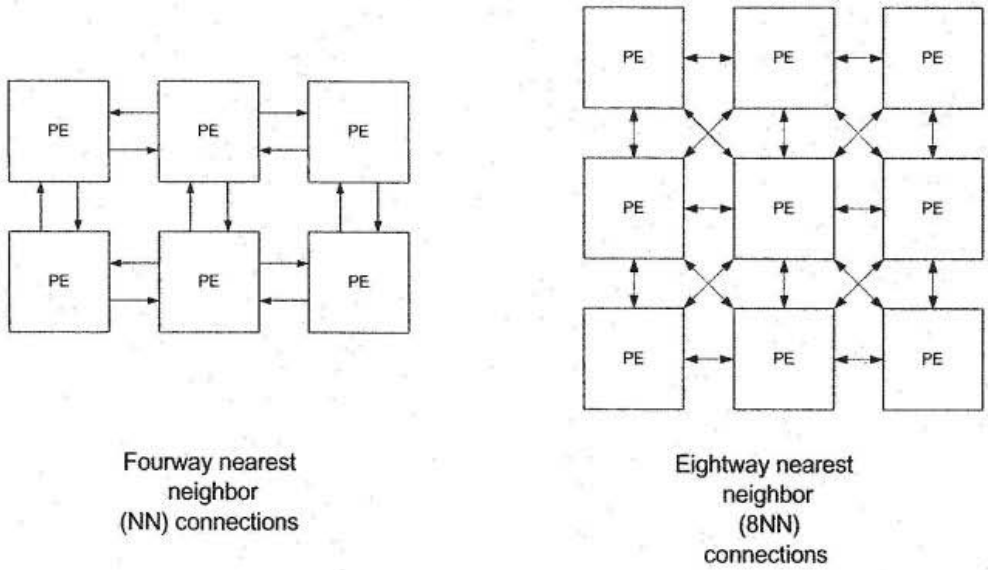

Figure 3.4: Examples of 2D mesh connections.

and this arrangement can be used for development of pipelined architectures. If a pipeline must branch, the PEs must have some type of two dimensional realization in their structure [13].

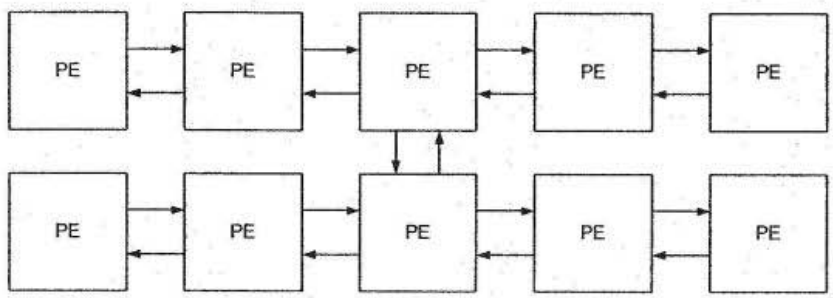

Figure 3.5: An example of a linear array configuration.

Finally, multiple layer crossbar switching can be used to connect PEs together, shown in Figure 3.6. While a crossbar can enable many PEs in a fabric to communicate with each other, it is usually costly to do this. Some coarse grain architectures use partial crossbars to communicate. In Figure 3.6, bidirectional pathways are established where communication pathways overlap.

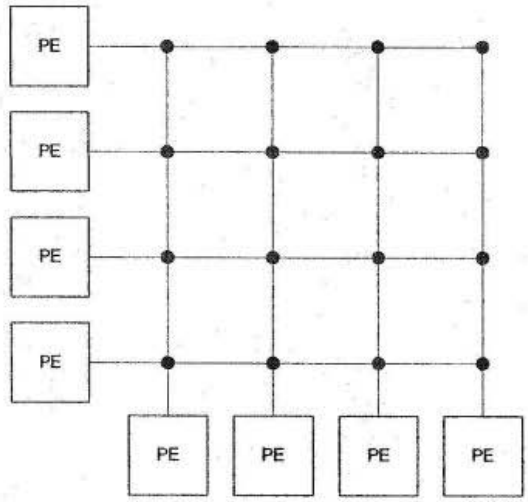

Figure 3.6: An example of a crossbar configuration.

Processing elements may exhibit one or more of the described types of interconnection and communication pathways, and they may be unidirectional or bi-directional in nature. The use of local and global communication buses can also be used in combination with these structures. A KressArray style of architecture will support this particular arrangement, and is illustrated in Figure 3.7 [42].

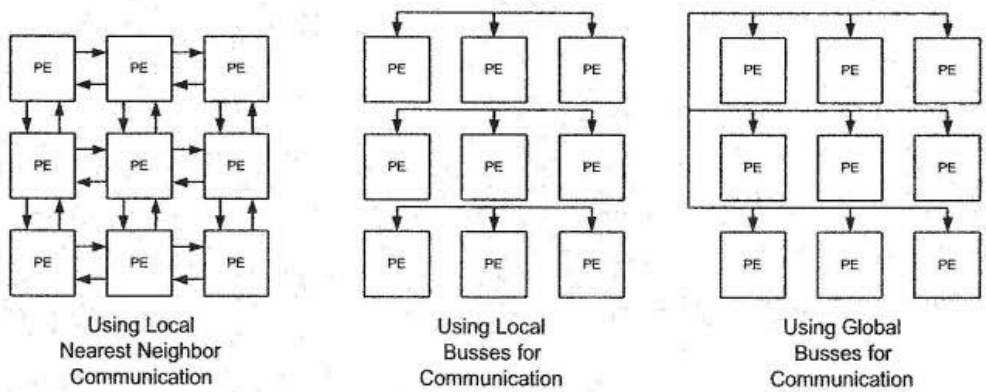

Figure 3.7: An example of a KressArray configuration. Multiple communication schemes between processing elements are used.

Coarse grain architectures have varying applications. One application area is DSP [42]. Many operations and algorithms used in DSP utilize multiplication and addition

operations. For example, to create a Finite Impulse Response (FIR) filter, a designer could configure and connect processing elements together and then flow the input signal through the datapath [14]. If the architecture is runtime reconfigurable, the filter parameters could be changed while processing data, creating an adaptive filter.

With respect to symmetric ciphers and hash functions, many operations are similar to that of DSP. An array of PEs can form an encryption array or a pipeline to implement the cipher of choice. While being runtime reconfigurable, keys can be changed or a new cipher put into place while running. Limiting factors for use with respect to ciphers and hash functions consist of lack of support within the PEs for some required operations, lack of PE resources to accommodate the cipher, and lack of communication resources within the reconfigurable fabric.

Coarse grain architectures begin to show a divergence from their fine grain cousins when addressing application space. A fine grain technology is rather universal in nature and is meant for deployment in many areas such as DSP, cryptography, control applications and telecommunications applications. Coarse grain architectures will be more suited for a particular area, that is, one architecture may be suited for DSP while another will be more suited for multimedia applications [13].

Coarse grain architectures exhibit some disadvantages over fine grain architectures. As with FPGAs, processing elements can be left unused if there are not enough communication resources available. An unused PE in a reconfigurable architecture is a greater loss of computational resources than in an FPGA, since PEs are larger than CLBs and are far fewer in number [42]. Another problem can arise if operations and bit-manipulations require word lengths different than the data width supported by the PEs, resulting in a waste of computational resources. Power consumption in general should be less than that of FPGA on a per implementation basis, but the communication network must be carefully designed to support low power consumption [42].

There are various methods for mapping designs onto a reconfigurable fabric. From the examination of various architectures, the mapping method is highly dependent on the type of PEs and the communication arrangement used. Some architectures will go from a HDL description of an algorithm and automatically placed PE elements to implement the design (with possible user interaction). Other architectures will try to translate a high level description of a design in C or C++ to a hardware configuration within the chip [13].

3.4.1 Survey of Existing Technologies

Surveys of existing coarse grain technologies exist and Table 3.5 is a compilation of some existing technologies [13] [14]. The architecture field in the table represents the general communication structure within the reconfigurable fabric and granularity refers to the communication word width between processing elements.

Name	Architecture	Granularity
MorphICs	2D Array	Undisclosed
Chameleon CS2112	2D Array	32-bit
DReAM	2D Array	8 and 16-bit
CHESS	Hexagon Mesh	4-bit
MorphoSys	2D Mesh	16-bit
REMARC	2D Mesh	16-bit
PipeRench	1D Array	128-bit
Pleiades	Mesh and Crossbar	multi granular
Garp	2D Mesh	2-bit
RAW	2D Mesh	8-bit multi granular
Matrix	2D Mesh	8-bit multi granular
RaPID	1D Array	16-bit
Colt	2D Array	1 and 16-bit inhomogeneous
KressArray	2D Mesh	selectable multiple NN
DP-FPGA	2D Array	1 and 4-bit
PADDI-2	Crossbar	16-bit
PADDI	Crossbar	16-bit

Table 3.5: Survey of existing coarse grain reconfigurable technologies.

3.4.2 Cryptographic Applications

Thus far, there has not been a reconfigurable architecture design specifically for the use of cryptography. DSP is a large field that requires many bit-level operations and often requires high speeds of operation.

There has been some work into dynamically reconfigurable cipher cores in which the reconfigurable nature of the chip has had cryptography as a primary design goal. Chameleon (not to be confused with Chameleon Systems and the CS2112) is a cipher implemented on a chip with architecture that contains reconfigurable features specifically for bit-level encryption operations [43]. A block diagram illustrating important components of the Chameleon cipher chip is given in Figure 3.8.

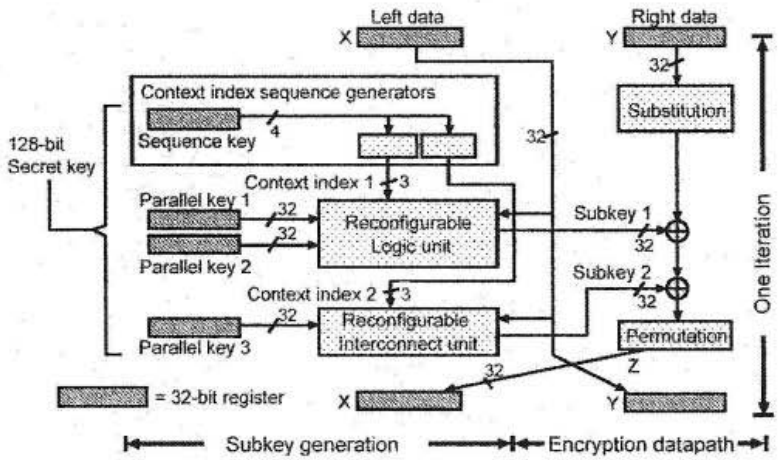

Figure 3.8: Chameleon cipher chip, designed for encryption.

The Chameleon cipher operates on 64-bits of plaintext and uses a 64-bit key. The reconfigurable blocks within the architecture are used to generate new subkeys during execution. The purpose of the Chameleon cipher chip was to add flexibility to the hardware design, a benefit of utilizing a coarse grain reconfigurable architecture [43].

3.5 Chameleon Systems CS2112

The Chameleon Systems CS2112, also referred to as a Reconfigurable Communications Processor, is an integrated system on a chip that contains a 32-bit coarse grain reconfigurable fabric. The CS2112 was designed in 2000 with high speed signal processing applications in mind. The architecture is designed to allow high speed parallel processing of information, fast design to market time, and flexibility with respect to individual application needs.

The benefit of the CS2112 is that it is a coarse grain reconfigurable system on a chip. As discussed in Section 3.4, runtime reconfigurability allows the designer to apply different algorithms during operation of the chip. The CS2112 incorporates two reconfigurable fabrics; one is not currently processing and is referred to as the background plane, while the other is actively processing and is referred to as the active plane. The swapping of active and background planes is described in Figure 3.9 and can be accomplished in one clock cycle [14]. The swapping of planes allows the user to load the background plane from external memory during runtime.

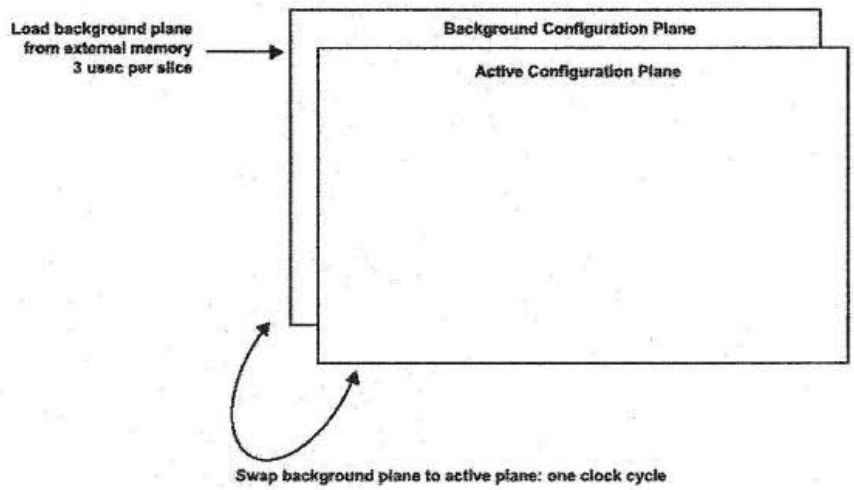

Figure 3.9: Process of swapping active and background fabrics.

The CS2112 coarse grain processing elements can also be reconfigured individually during runtime based on control logic that is associated with each section of the reconfigurable fabric. One of the advantages that the CS2112 has over traditional ASICs is that it is reconfigurable, and that it is reconfigurable at runtime makes it more useful than FPGAs.

3.5.1 CS2112 High Level Architecture Description

A high level description of the CS2112 is illustrated in Figure 3.10. The components of the CS2112 communicate over a 128-bit Roadrunner Bus. The Argonaught RISC Core (ARC) processor provides the CS2112 with a general microprocessor on chip to control the reconfigurable fabric, run user code, and control the other components of the CS2112. The ARC processor has been optimized for the CS2112, and it employs a four-stage pipeline and 64 general purpose 32-bit registers. There is a 8k byte instruction cache, and a 4k byte data memory. The Direct Memory Access (DMA) subsystem contains sixteen channels for transferring data amongst the various modules within the CS2112 [14].

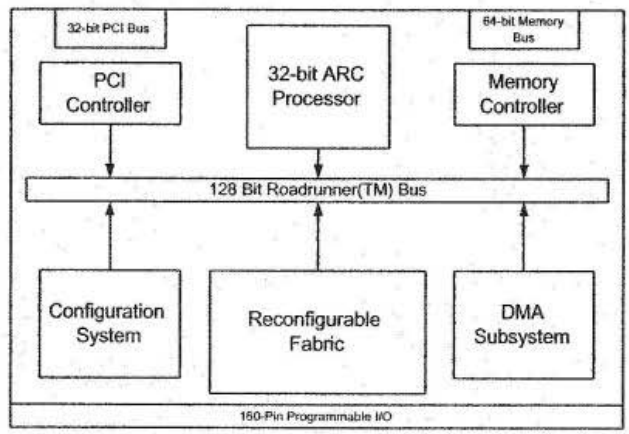

Figure 3.10: High level block diagram of CS2112 components

The CS2112 has four banks of 40 Programmable Input Output (PIO) pins which give the chip its highest IO bandwidth of 3.2 Gbit/s. When all four banks are used the total bandwidth of the CS2112 is 12.8 Gbit/s and is the highest data transfer speed of the RCP. The PIO pins of the CS2112 allow it to be integrated into larger systems such as interfaces with FPGAs, analog to digital/digital to analog blocks and external memory modules.

The reconfigurable fabric illustrated in Figure 3.11 is broken into four slices, each containing three tiles. Each tile contains seven Data Path Units (DPUs) and two 16 by 24-bit multipliers. The basic PE building blocks within the fabric are the DPUs, with added support from multipliers and memories. The Local Store Memorys (LSMs) in the CS2112 fabric provides a memory structure so the DPUs can store and retrieve data.

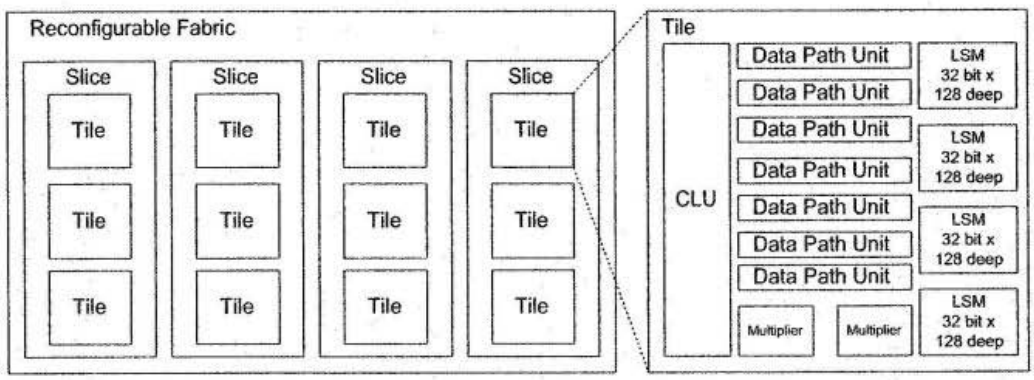

Figure 3.11: High level decomposition of reconfigurable fabric.

The CS2112 DPU structure has multiple inputs from local, global, and feedback communication pathways and buses, giving a communication arrangement shown in Figure 3.12 [14]. A DPU can communicate locally with 8 DPUs downward or 7 upward, otherwise global routing must be used through vertical and horizontal buses. If global routing is used, each tile has eight 32-bit data buses while each slice has three groups of three 32-bit data buses (one per-tile). The process of placing DPUs

in a design is done through a graphical floor planner that comes with the software tool set (C~SIDE tools) for the CS2112.

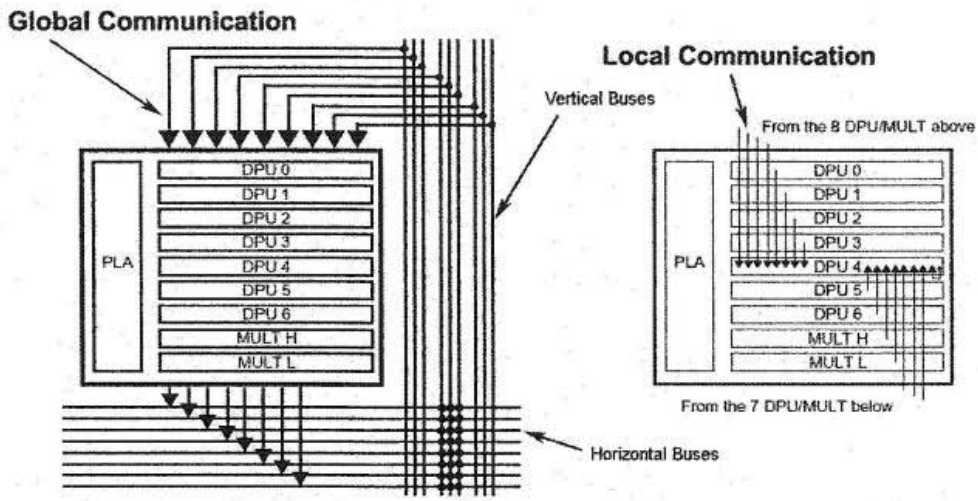

Figure 3.12: Communication arrangement between processing elements.

Each tile contains a Control Logic Unit (CLU) that provides support for design of Finite State Machines (FSMs). FSMs are used to control the flow and sequencing of the DPUs that make up the datapath. Each DPU also has an associated Control Store Memory (CSM) (located in the CLU). The CSM stores the various configurations for the DPU and during runtime the DPU can be reconfigured by the CLU. The Programmable Logic Array (PLA) is a component within the CLU that provides combinational logic for the user's FSMs.

3.5.2 CS2112 Data Path Unit

The most important functional building block within the CS2112 fabric is the DPU. Figure 3.13 is a detailed block diagram illustrating the various inputs, outputs, and functional blocks within a DPU.

The inputs to a DPU are abstracted to eight inputs on both the A and B sides.

Depending on the routing setup, these sixteen inputs originate from the local and global interconnects illustrated in Figure 3.13. To use a local interconnect the input must come from the output register of a DPU that is within seven DPUs below, or eight DPUs above, otherwise a global interconnect must be used. When implementing for the CS2112, the designer must keep in mind that communication resources will place restrictions on the design. It is important to note that a DPU must be used to address and transfer data from an LSM. In this process only even numbered DPUs can read from an LSM, and odd numbered DPUs can write to an LSM.

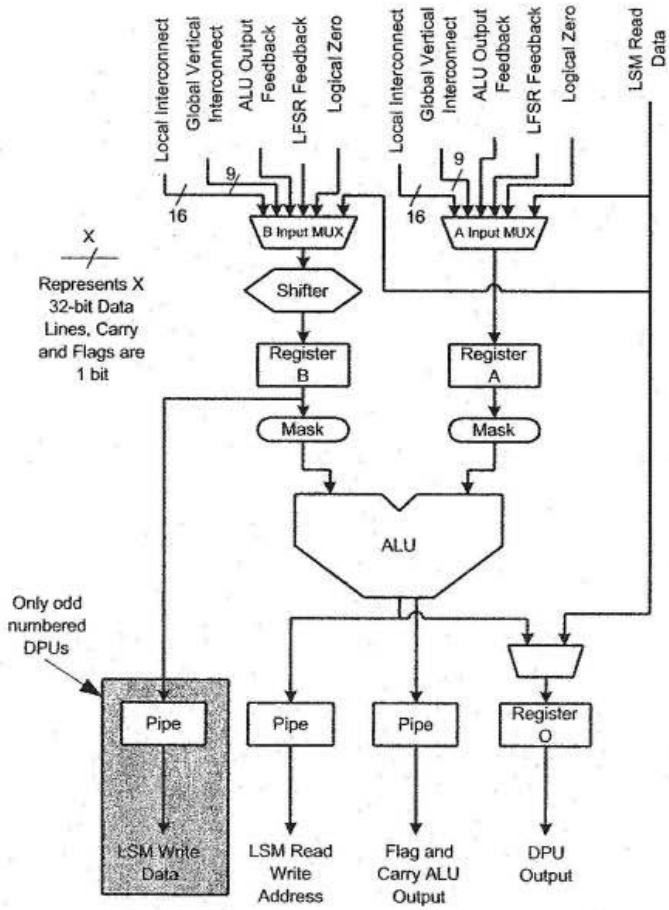

Figure 3.13: Datapath unit block diagram.

The CS2112 DPU contains three 32-bit registers, one on each input, and one on the output of the DPU. These registers can be configured to hold their value or to load a new value. The registers on the inputs can be bypassed while the output register cannot [14]. The delay of data flowing through a DPU is dependent on whether the input registers are loaded or not. The delay will be two clock cycles if the registers are enabled, or one clock cycle if they are bypassed.

The Arithmetic Logic Unit (ALU) within the DPU structure supports C and Verilog operations. Some of the bit-level operations that the ALU supports are addition, subtraction, bit-wise logical operations, and equality testing. The ALU can also be set to pass data through without modification. In addition to ALU functionality, each DPU contains a 32-bit barrel shifter that is also capable of bit-wise AND/OR masking, word swapping, byte swapping, and word duplication. The configuration of a DPU is done through the Verilog HDL. Section 3.5.6 provides a more detailed look at how the fabric is programmed during the design phase.

3.5.3 CS2112 Local Store Memory

Within each tile on the CS2112 fabric are four LSMs. Each LSM is 32-bits wide and 128 locations deep. The LSM primitive can be connected together to provide deeper memories (using a special chain input/output). With the assistance of an extra DPU, wider memories can also be configured, but for the purposes of this research these special memory configurations were not utilized.

3.5.4 CS2112 Multiplier

While many DSP algorithms require multiplication, it is not common in the area of cryptography due to the fact that it is a computationally intensive operation. For this research, the CS2112 multipliers will be used with the RC6 cipher. Each tile within the fabric contains two multipliers illustrated in Figure 3.14 [44].

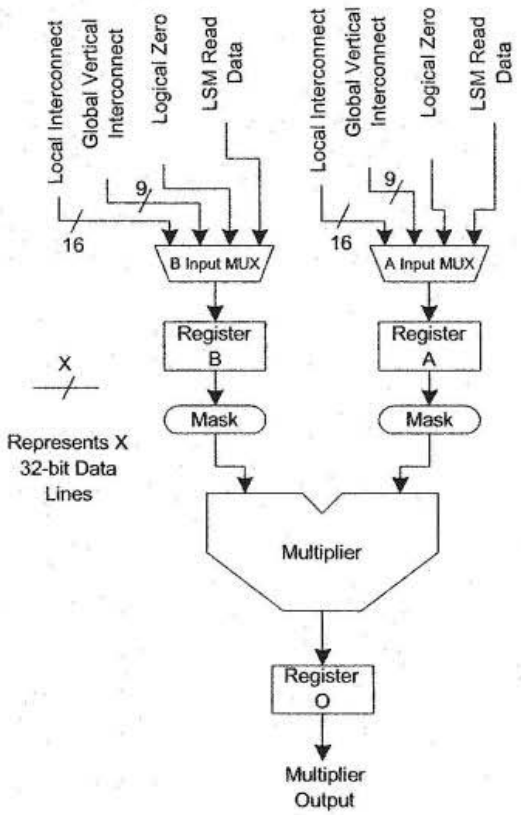

Figure 3.14: Multiplier block diagram.

Since the CS2112 was designed with DSP applications in mind, the hardware multipliers on the fabric are designed to operate on fixed point two's complement numbers [14]. The multipliers on the fabric will use two 16-bit operands to produce a signed 32-bit operand. With respect to RC6, the multiplication operation required is an unsigned integer multiplication (mod 2^{32}). There will be further discussion on how the multiply operation was implemented on the fabric for RC6 in Section 4.3.1.

3.5.5 CS2112 Control Logic Unit

The CLU is the control mechanism for the reconfigurable fabric of the CS2112. The CLU provides control over the DPU configurations, synchronous state machines and conditional operation. There is one CLU in each tile on the fabric and communication pathways are illustrated in Figure 3.15 [14]. The following are key components of the CLU.

- Control Store Memories (CSMs)
- State Register Blocks (SRBs)
- PLAs
- MUXing Plane

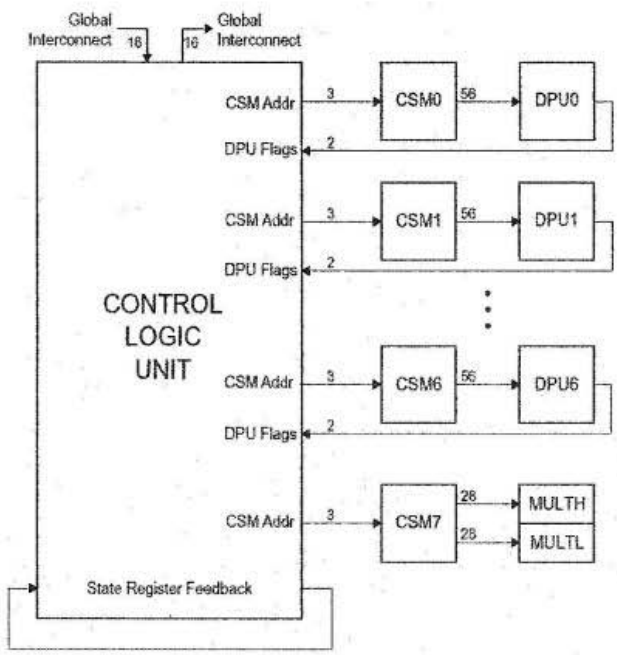

Figure 3.15: CLU communication and interaction with processing elements.

The CSM stores the configuration information for each DPU. They are eight locations deep and provide eight different DPU configurations per DPU. The multipliers also have CSMs associated with them which allow four configurations per multiplier. There are eight SRBs in a CLU and the SRBs are used to register the PLA outputs. The PLA has 16 inputs, 32 outputs and 32 product terms that make up the required control logic. The MUXing plane controls the inputs to the PLA and inputs can come from various sources including outputs from other PLAs in the same slice (local communication), PLAs in another slice (global communication), or feed back from DPU flag signals. For example, if a DPU was implemented as a counter, when the count was completed the DPU can communicate back to the CLU through a flag signal. PLAs can communicate with each other globally by the use of broadcast registers.

3.5.6 Design Process For The CS2112 Fabric

Designing for the CS2112 involves the use of a Verilog simulator, waveform viewer, and GNU C programming language tools (gcc, gdb etc.), and the C~SIDE software tool set developed for the CS2112. Figure 3.16 is a block diagram illustrating the design flow for the CS2112.

Chameleon recommends that the starting point for development is a C function of the algorithm that will be used [14]. The development process illustrated in Figure 3.16 occurs in a Unix environment, with the exception of testing with the the CS2112 development board which is based in a Windows NT4 environment.

There are certain guidelines which are required when converting a C function to a fabric function. The C function must not call any functions itself and if other functions are called, the code must be restructured. Data can be passed by the use of function arguments aligned to 128-bit boundaries, and the use of global values are invalid. Finally, there is no floating point between the conversion from a C function to a fabric function.

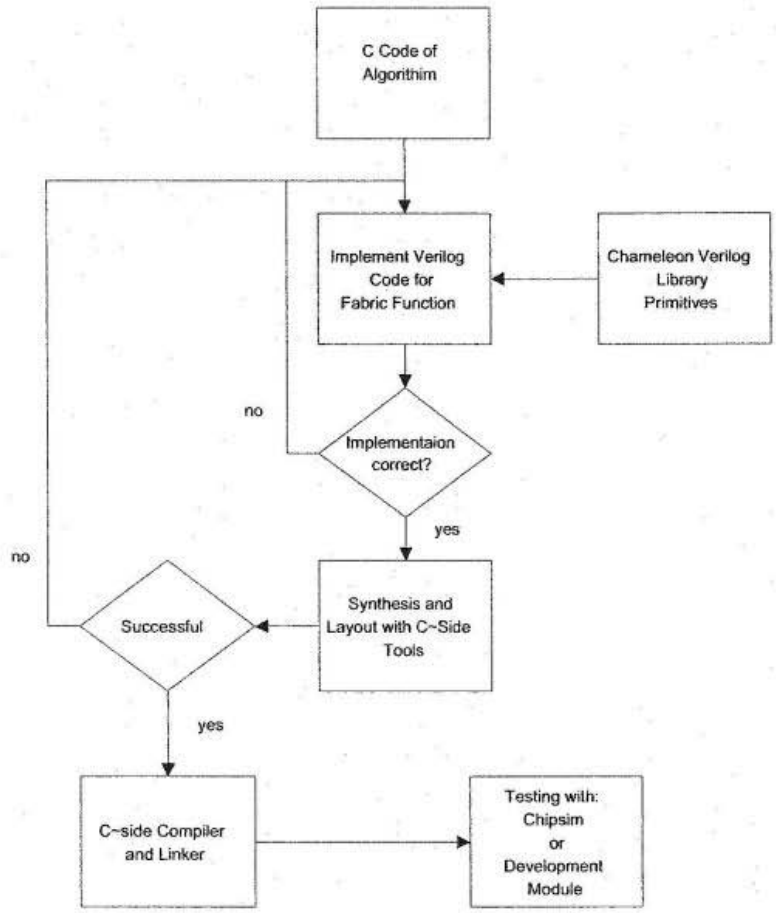

Figure 3.16: Design flow for the CS2112.

The ARC processor is responsible for setting up and running the fabric. Fabric parameters are set up through CS2112 BIOS function calls. Data is passed into the fabric through streaming DMA (transfer while the fabric is running), or through independent DMA (transfer before and after the fabric has run). Once the fabric is set up, a start signal is sent telling the control logic it can begin the operation. Once the fabric is finished, it sends a done signal to the ARC. Data is transferred out of the fabric in the same manner it was passed in [14]. To execute the fabric function the ARC calls a function that is defined by a *#pragma* statement. Appendix B.1 and Appendix B.2 are examples of C code that the ARC processor runs within the

CS2112 to control the fabric.

The implementation stage in Verilog requires the writing of structural Verilog code to represent the function. The designer has much flexibility at this point because the basic building blocks are DPUs, LSMs, and MULTs. Once the datapath is created by wiring together the primitives, control logic is written as Register Transfer Level (RTL) state machines with input signals coming from DPU flag outputs. The DPU module is configured by ORing instruction mnemonics together to give a configuration bit stream that is stored within a CSM. The CSM will use the bit stream to configure the fabric at runtime. Designing for the CS2112 requires that all Verilog code be encapsulated within a top level module that only receives only start, done, reset and clock signals. Appendix A.2 gives an example of a top level module definition. Within the top level module is the logic for the datapath and controller. Appendix A.3 gives example Verilog code for controller design, and Appendix A.4 gives example Verilog code for datapath implementation.

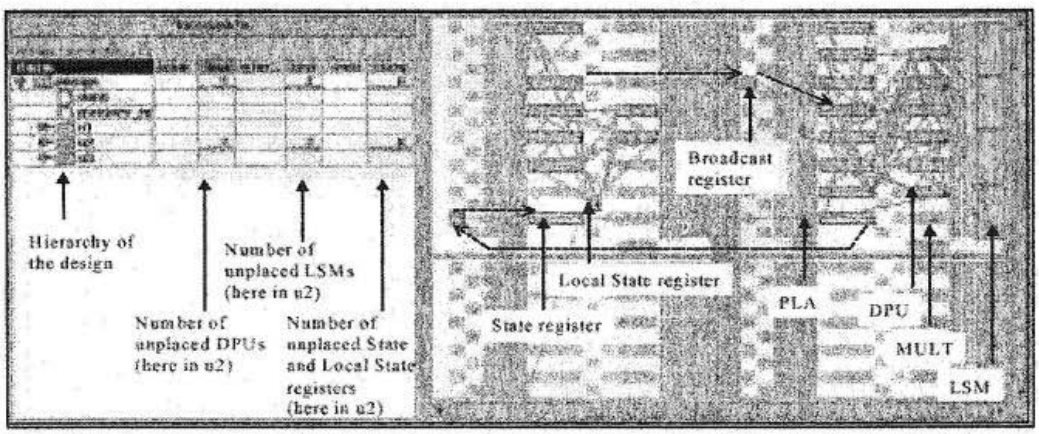

Figure 3.17: Screen capture of the graphical floorplanner.

Once the Verilog design is tested and deemed to be correct through a waveform viewer, the appropriate Verilog modules are loaded into the C~SIDE tools. Using these tools, the design is checked for timing violations and it can be carried on to the layout phase. The software tool set tries to place cells for layout within the fabric

using a simplistic approach that is successful for small designs.

Larger designs require the user to layout the design by hand. Figure 3.17 is an illustration of the C~SIDE interactive graphical floorplanner. Once the design is laid out and is deemed to be "routable" by the floorplanner, the design can be either simulated with the chip simulator that comes with the tools, or tested on a development board provided by Chameleon Systems. Both methods were used to test implemented designs in this research.

Chapter 4

Symmetric Block Cipher Design and Implementation

In this chapter, the designs and results of RC5 and RC6 are given. Various design methodologies and issues with the architecture of the CS2112 are discussed and there are a total of five different designs between RC5 and RC6. Conventions used within diagrams of datapath and control structures are outlined.

For the purpose of research conducted with the CS2112, two design strategies were employed for the implementation of each cipher. Both strategies provided a wide survey of performance, ease of implementation, and efficient use of fabric resources. With RC5, an iterative design approach was used as well as a pseudo-pipelined approach that used an iterative design as its basic building block. With respect to RC6, a more efficient pipelined approach was employed, a design method that lends itself toward the intended structure of the CS2112.

4.1 Diagram Use

For designs on the CS2112, many diagrams are used to describe algorithm design in the fabric. Design structures within the reconfigurable fabric are illustrated as block diagrams to show component interaction. Figure 4.1 shows some examples of how

DPUs are used in design diagrams.

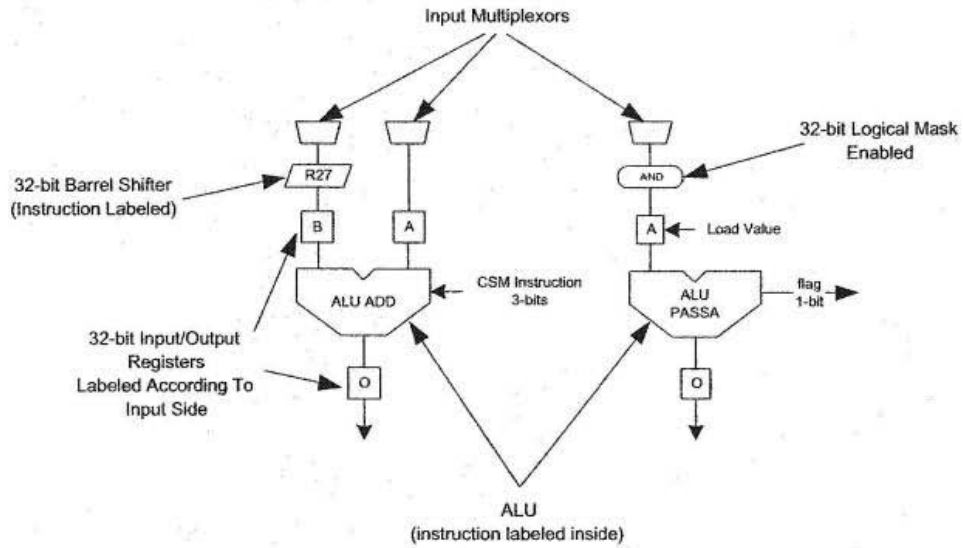

Figure 4.1: Examples of configured DPU structures.

If a component within a DPU is configurable, the configuration is shown inside the structure. The barrel shifter (illustrated on the left) is configured to shift the input word twenty-seven bits to the right. The logical mask structure (illustrated on the right) is configured for an AND mask with a vector that is initialized in the A side register. Initialized registers are shown with an arrow intersecting horizontally with the value labeled. ALU instructions are labeled within the structure and if an ALU has multiple configurations they are all illustrated within the diagram. A flag output from the ALU is represented by a horizontal arrow coming out of the ALU, while DPU CSM instructions are given by a horizontal arrow into the ALU structure.

Memory structures are abstracted in two different forms, as shown in Figure 4.2. Both structures contain a fabric LSM for memory space, and a DPU for memory address generation. Within the structure, information is given about the memory module and its particular use to the algorithm being implemented. For the purposes of this research, all array elements within the memory are 32-bit words.

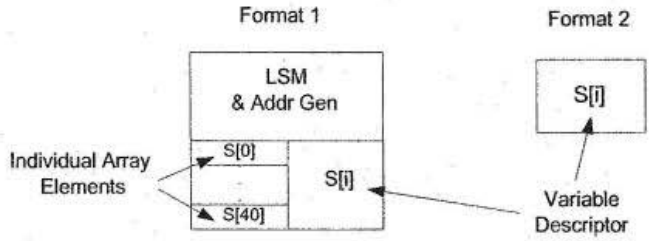

Figure 4.2: Two methods for describing memory structures containing one LSM and one DPU.

Some complex multiple DPU structures within a datapath are illustrated as blocks, with their respective operation labeled within. This is to reduce the complexity of the descriptions and to aid in the legibility of the diagrams.

4.2 RC5 Designs

The following subsections illustrate various implementations of RC5 on the CS2112. The designs are presented in the order in which they were created, with subsequent designs developed as more experience was gained with the C~SIDE tools and in simulation with Cadence VerilogXL.

4.2.1 RC5 Simple Iterative Design

For the first design and analysis of RC5 on the CS2112, a simplistic design approach was used to both gain familiarity with the development environment, and maximize the chances of a successful design. The simple iterative design of RC5 operates as follows:

1. Two 32-bit words of plaintext are used as input into the cipher as scalars from the ARC processor. These words of plaintext are stored in DPU input registers for simplicity, and are immediately accessible by the datapath.

2. A half of a round of RC5 is implemented in the fabric.

3. The associated control logic either configures a DPU to operate on the data and pass the information to the next stage or it configures a DPU to hold its value.

4. Once a half round is completed the values swap and are loaded into the top of the datapath for the next half round. Once both half rounds are completed, the round description given in Figure 2.5 is finished.

5. When an appropriate number of rounds have been processed the controller will assert the done signal and the ARC processor will transfer the ciphertext from the appropriate registers as return values from the fabric function call.

A high level abstraction of the controller and datapath for the simple iterative RC5 design is shown in Figure 4.3. The data dependent rotation is shown with inputs, outputs, and control signals labeled. The controller is an FSM in Verilog, and was mapped to the CLU within the fabric through the C~SIDE tools.

Initially the datapath was designed based on the required operations of the algorithm. Multiple DPUs were used to build the data dependent rotation module because this operation is not supported within the architecture of a single DPU. By using an iterative approach, each operation in the algorithm can be translated to a DPU configuration. As lines of code are executed in a sequential program, data flows through the datapath in the form of a hot spot of activity, while DPUs above and below the datapath mostly hold previous values.

The roles of the datapath elements for implementation of the algorithm described in Figure 2.5 are as follows:

- DPU1: $A + S[0]$ and holds the modified value for the completion of the round.
- DPU2: $A + S[1]$ and holds the modified value for the completion of the round.
- DPU3: Bit-wise exclusive OR ($\oplus$) operation required by RC5.

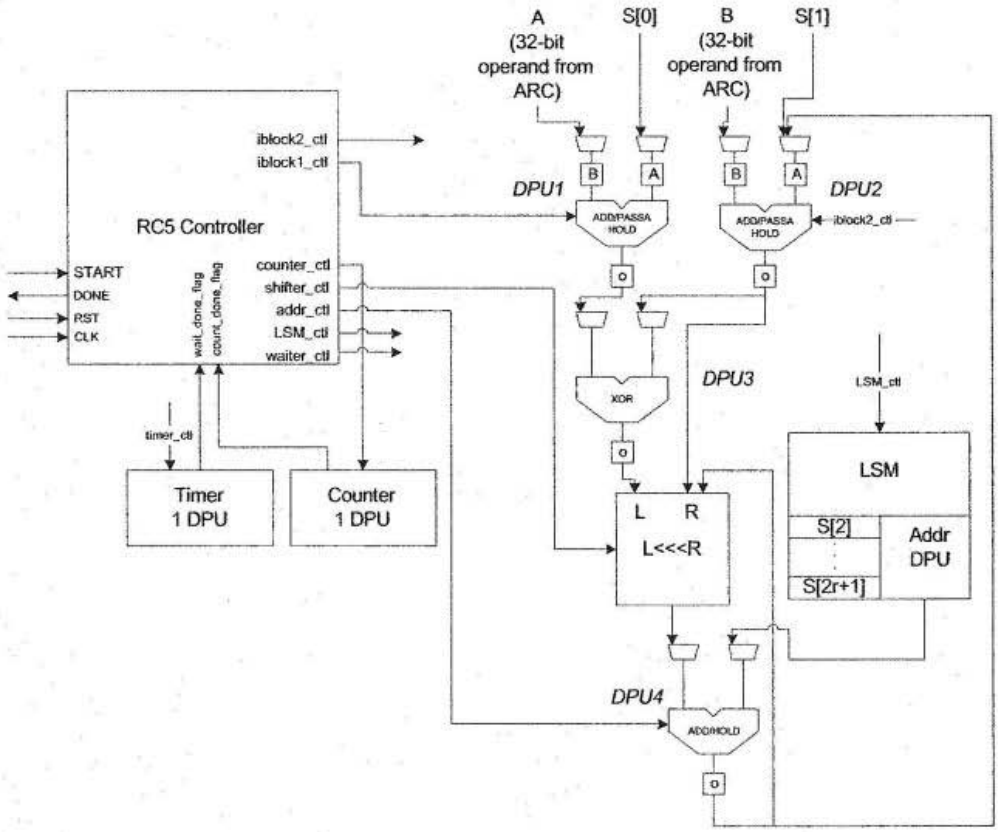

Figure 4.3: Abstracted block digram of simple iterative RC5 design

- DPU4: Addition of a subkey value to the data.

There are advantages to using this iterative approach in the design. Firstly, since many DPUs are holding previous values, problems with race hazards and mistiming due to latency in the datapath are simplified. Overall fewer DPUs are used than in a pipelined approach because DPUs are not used solely for the purpose of delay, as is required for a pipelined design.

A disadvantage to a simple iterative approach is that the datapath needs to be configured while running by the controller as the hot spot of activity progresses. A complex controller results from timing the configuration of the datapath in this fashion. For example, the two DPUs that store the initial words of plaintext, A and B,

must initially add the first two subkeys to the plaintext and then hold the result. Later in the execution of a round, these DPUs must load in the processed words from the bottom of the datapath and store the values in their registers. These are three different roles that the same physical structures must play during the operation of RC5. The controller must be synchronized to switch configurations with the progression of these roles. The task of designing a complex controller can become quite cumbersome for the CS2112 and should be avoided for larger designs. Smaller sub-controllers should be used or a less complex datapath should be designed. Also since there is usually a hot spot of activity traveling through the datapath, the fabric is under-utilized, which results in a lower performance than a pipelined approach.

The performance of a design on the CS2112 can be estimated based on the amount of latency through the datapath. If a DPU uses its input registers, it takes two clock cycles to provide the output. It takes one clock cycle to load the output register with the output (this register cannot by bypassed), and one clock cycle to load the input registers. If the input registers are not used it takes one clock cycle for a DPU to produce its output. If data is to be accessed from a LSM it takes three clock cycles for the address generator DPU to produce the data from when the address was input to the DPU. Writing to a LSM is immediate and takes one clock cycle.

For the implementation of the data dependent rotation, five DPUs were used. Since there is not a data dependent rotation primitive in C, the following definition was used for the left data dependent rotation [17].

```
#define ROTL(x,y) (((x)<<(y&(w-1))) | ((x)>>(w-(y&(w-1)))))
```

The $\ll$ is the left logical shift operation, $\gg$ is the right logical shift operation, & is the bit-wise logical AND operation, and | is the bitwise logical OR operation. The definition for the rotation module within RC5 was taken directly from the above C declaration.

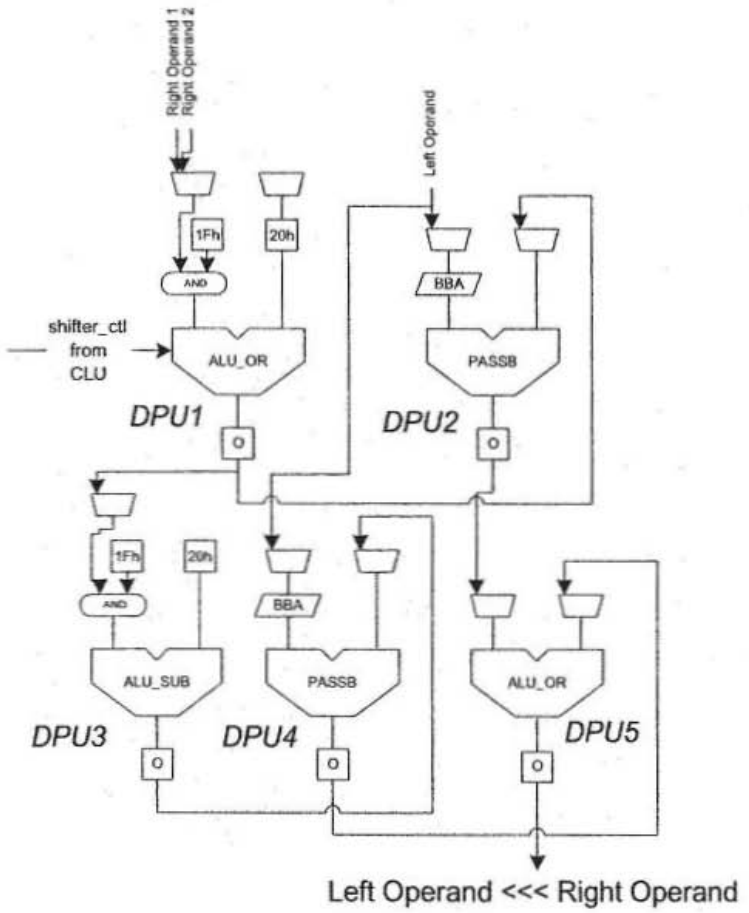

Figure 4.4: Structural diagram of data dependent rotation.

There are two possible inputs for the right operand into the rotation module. The control logic was designed to use right operand one during the first round, then use right operand two during subsequent rounds. This design choice was made to correct a glitch that would occur due to mistiming in the datapath. To control this behaviour, a control signal is used from the CLU into the appropriate DPU and can be seen in Figure 4.4. The initial values of registers are labeled within the register structures and are in hexadecimal. The BBA configuration of the barrel shifter will shift the input by the value on the A input. If bit six of the input is one, the shift will be left. If bit 6 is set to 1, then the shift is right. The following is a more detailed

explanation of the function of each structure within the rotation module:

- DPU1: $y\&(w-1)$ required by the rotation. ALU_OR is used here to set bit six to '1'.
- DPU2: $(x) << (y\&(w-1)$.
- DPU3: $w-(y\&(w-1)$, and uses an AND mask to set bit six to '0' for a right shift.
- DPU4: $(x) >> (w-(y\&(w-1)))$.
- DPU5: The final bit-wise OR operation for the rotation.

The control logic and the above datapath were simulated using Cadence VerilogXL. The design was then imported into the C~SIDE tools and timing and design rules were checked. Automatic Placement was attempted but was unsuccessful, therefore manual routing of DPUs, LSMs, and control state registers was performed. The C~SIDE tools do not attempt to re-arrange placement upon encountering a non-routable design. A design will be deemed routable if all communication paths between DPUs, LSMs through local and global communication are available and valid. An annotated illustration of the floor plan layout of the simple iterative design of RC5 is given in Figure 4.5.

After routing of the design, the C code was modified to generate ciphertext from within the ARC processor and from a call to the fabric function. The ciphertext was then compared to verify correct operation. The design was verified with the chip simulator for the CS2112 and on the Chameleon Systems CS2112 development module. The resource usage of the simple iterative version of RC5 is given in Table 4.1. Section A.4 is the Verilog description of the datapath. The control logic for the datapath required 13 states with seven 3-bit output signals, and two 1-bit flag signals

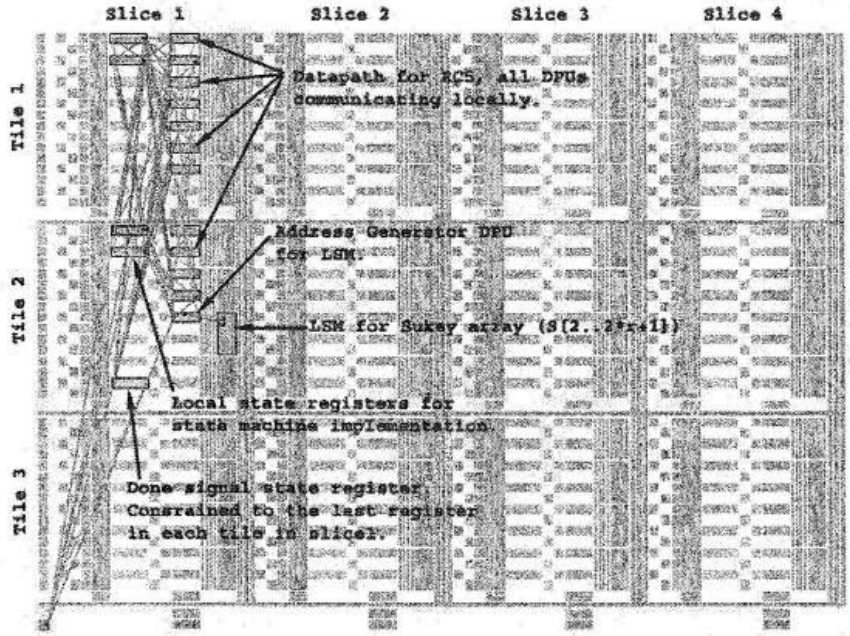

Figure 4.5: C~SIDE floorplanner screen shot of simple iterative RC5 design.

from the datapath (not including start and done signals), and is illustrated in Section A.3. Both the datapath and controller are abstracted to a top level module that the C~SIDE tools will interpret as the fabric function, and this module is described in Section A.2.

Resource	Count				Total
	Slice 0	Slice 1	Slice 2	Slice 3	
DPU	12	0	0	0	12
LSM	1	0	0	0	1
MUL	0	0	0	0	0

Table 4.1: Resource usage for the simple iterative version of RC5.

After the testing of this design, simulation results and waveforms from the development module were used to measure the performance of the design. Table 4.2 shows both the total number of clock cycles from the start of the fabric to when the done signal is sent to the ARC processor and the number of clock cycles required for the data dependent rotation. The time required for the rotation is variable between

two and three clock cycles because the operands may arrive at the module inputs at different times changing the critical path through the module.

RC5 Iterative	Clock Cycles
Start To Done Time	157
Data Dependent Rotation	2 to 3

Table 4.2: Timing information for the simple iterative version of RC5.

Based on a 100 MHz clock in the fabric, the number of clock cycles between the start and done signal and considering that 64 bits of data are being processed during operation, the simple iterative version of RC5 was determined to operate at 40.7 Mbit/s.

4.2.2 Two Half-Round, Full Slice Version of RC5

The next phase of research involved the application of RC5 such that it more efficiently used the resources of the reconfigurable fabric. Using a variation of the design in Section 4.2.1, two half rounds were placed onto a slice allowing two separate plaintext pairs to be processed in parallel. A high-level abstraction of the two round implementation of RC5 is given in Figure 4.6.

Based on the simple iterative design, some changes had to be made to allow this design to fit into one slice. The simple half-round uses twelve DPUs in total. A slice in the fabric contains 21 DPUs over three tiles. The following changes were needed to fit the design into a single slice:

- One DPU was removed from the data dependent rotation.
- The control state machine was re-designed to operate with only one counter, removing one DPU from the design.

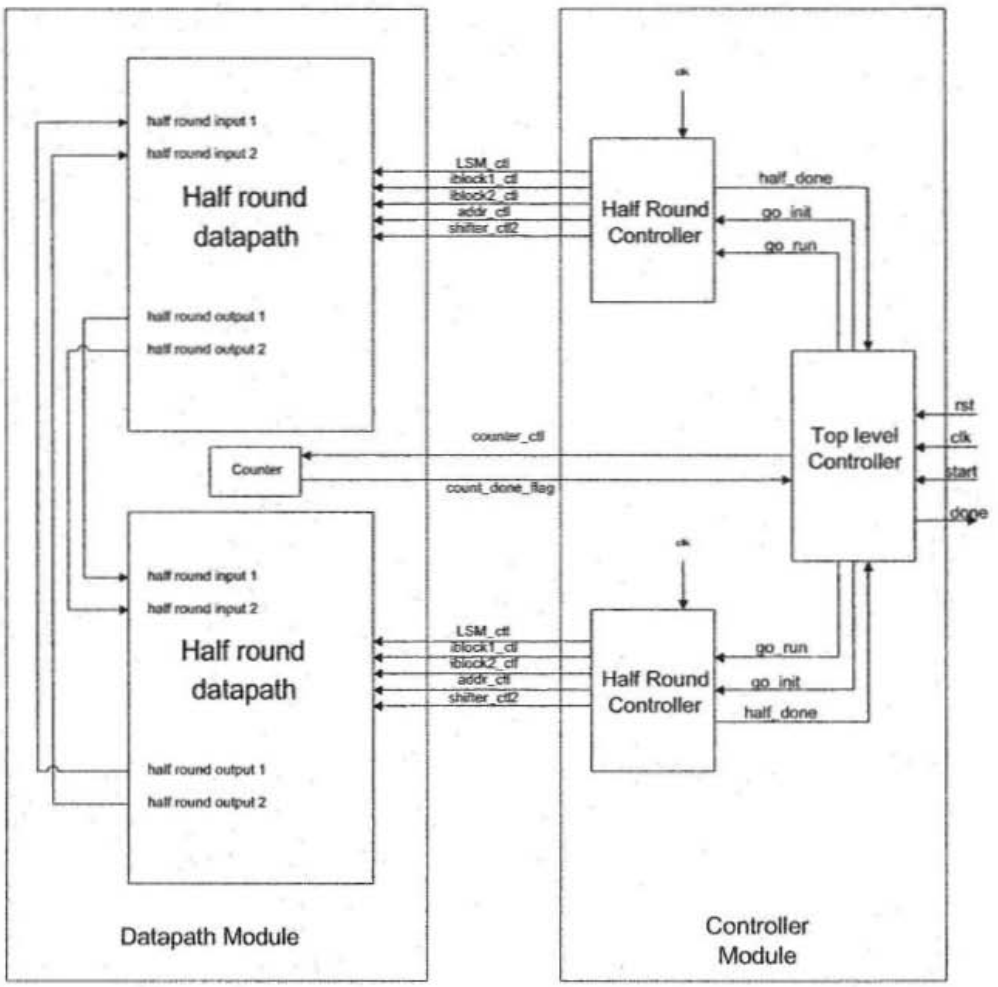

Figure 4.6: High level abstraction of the two half-round design of RC5.

A DPU was removed from the data dependent rotation by feeding back and changing configurations of another DPU within the rotation module. To do this, a DPU within the the rotation module must be sequenced by the controller during the execution of every round. An illustration of the four DPU data dependent rotation is given in Figure 4.7. From the modification of the rotation, the controller and the datapath elements required by the control logic, the full slice implementation of RC5 was accomplished using 19 DPUs in total. Resource usage is given in Table 4.3 for this design.

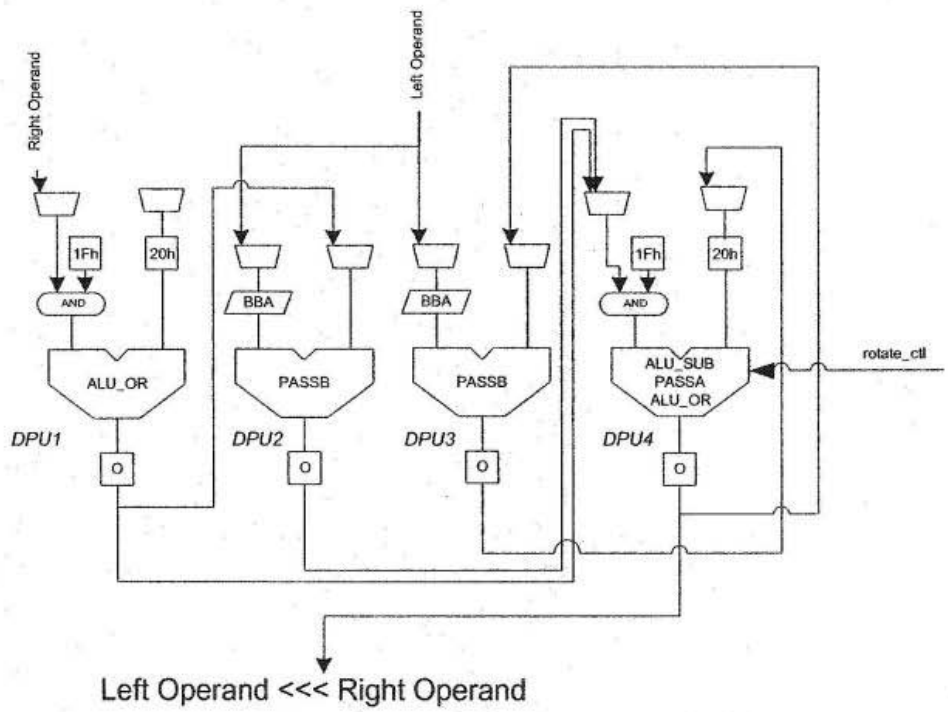

Figure 4.7: Four DPU implementation of the data dependent rotation.

The DPUs described in Figure 4.7 have the following roles:

- DPU1: $y\&(w-1)$ required by the rotation. ALU_OR is used here to set bit six to '1'.
- DPU2: $(x) << (y\&(w-1)$.

- DPU3: $(x) >> (w - (y\&(w-1)))$
- DPU4: Configurations: (1) uses bit-wise OR to provide the final rotation output, (2) provides DPU3 with $(w - (y\&(w-1))$ and (3) uses an AND mask to set bit six to '0' for a right shift.

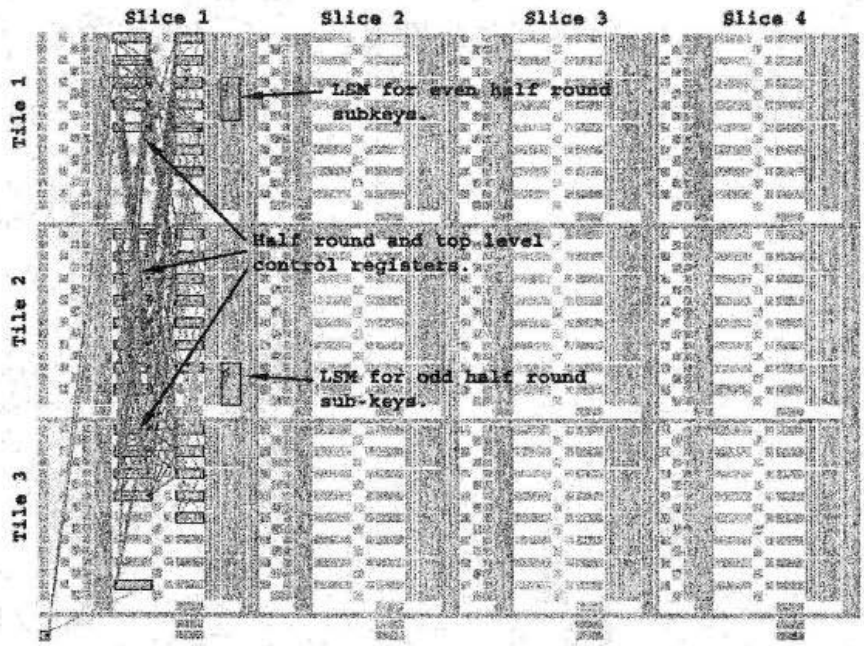

Figure 4.8: Screen capture of the two half-round RC5 fabric function.

By interleaving and mixing the locations of DPUs from both half rounds, the entire datapath makes use of local communication. Global broadcasting should only be used when needed, and requires receiving DPUs to have registered inputs.

Resource	Count				Total
	Slice 0	Slice 1	Slice 2	Slice 3	
DPU	19	0	0	0	19
LSM	2	0	0	0	2
MUL	0	0	0	0	0

Table 4.3: Resource usage for the two half-round design of RC5.

The design was fully implemented in Verilog and successfully laid out with the

floor planner in the C~SIDE tool set. Simulation results were used to measure performance. Table 4.4 shows both the total number of clock cycles from the start of the fabric to when the done signal is sent to the ARC processor and the number of clock cycles required for the data dependent rotation.

RC5 Full Slice	Clock Cycles
Start To Done Time	196
Data Dependent Rotation	3

Table 4.4: Timing for the full slice design of RC5.

The full slice version takes two blocks of 64-bit plaintext (128-bit total input) and has a throughput of 65.3 Mbit/s (with a fabric operating frequency of 100MHz). The performance figure is not quite double of the simple iterative design because processing time is lost when the top level controller receives a count done signal from the counter, and when the control signals are sent to the datapath to start the next round.

4.2.3 Full Fabric RC5 Design

Maximum use of the fabric was investigated by copying the design from Section 4.2.2 to the remaining three slices. Each slice iterates for three rounds and passes its data off to the next stage. This style is a pipelined method that uses an iterative design as its basic building block. The function of each half round module is different from Section 4.2.2. Instead of passing the data down locally to a half round module in the same slice, it broadcasts the data to the next slice. Therefore this design can be considered to be two separate RC5 pipelines. A floor plan layout of the full fabric design with the top-level control logic excluded is shown in Figure 4.9.

Simulation of this design was carried out by using the datapath and control logic associated with each half round. Top level control was done by the assertion of the

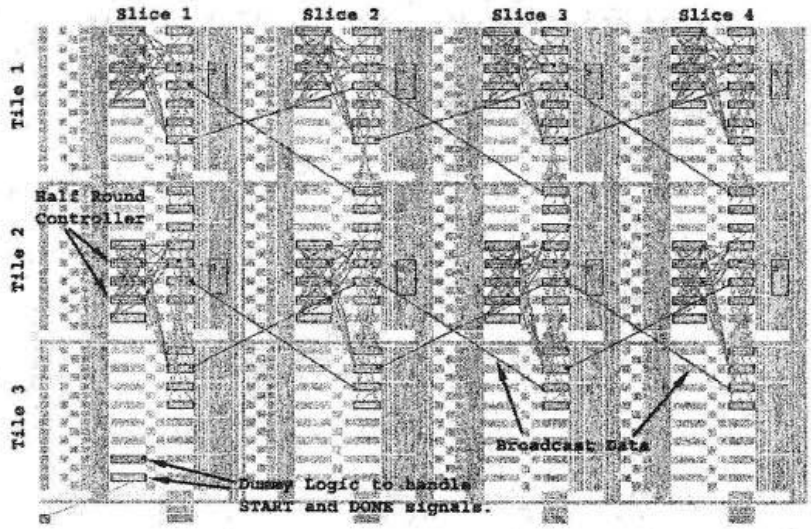

Figure 4.9: Screen capture of the full fabric RC5 implementation.

control signals from the test bench. The test bench for the design is given in Section A.7 to illustrate the role a top level controller would need to play if implemented on the CS2112 and to allow for full simulation of the datapath. From the simulation results, it takes 214 clock cycles to receive the done signal from the fabric, and since the pipeline accommodates a total of 512 bits of plaintext, a throughput of 237 Mbit/s is achieved.

Resource	Count				Total
	Slice 0	Slice 1	Slice 2	Slice 3	
DPU	18	18	18	18	72
LSM	2	2	2	2	8
MUL	0	0	0	0	0

Table 4.5: Resource use for the full fabric version of RC5 (control logic excluded).

4.2.4 Summary of RC5 Results

The three versions of RC5 thus far have all used an iterative half-round building block. The first simple iterated version utilized 14.3% of the total DPUs within the fabric. RC5 was designed for operation on a 32-bit general processing platform,

such as in desktop computers and smartcards [17]. The fabric of the CS2112 applies RC5 well because the data width between its processing elements is 32-bits wide and the operations supported within the DPU are also 32-bit operations. The biggest restriction to a fast iterative RC5 design is that there is no data dependent rotation operation contained within a DPU. The data dependent rotation accounts for roughly 42% of the resources used by all three designs. Performance is increased by arranging the iterative half-rounds to provide a multistage pipeline. This method of encryption does not allow the cipher to be used in a feedback mode such as CBC mode.

4.3 RC6 Designs

The following sections describe the work done with RC6 on the CS2112. The fully implemented design of RC6 uses knowledge gained from RC5 on the CS2112. The most challenging component of the design of RC6 on the CS2112 is performing the $X(2X + 1) \bmod 2^{32}$ operation during the execution of a round. There are more operations to be performed in a round of RC6 than in RC5, therefore it will be assumed that more resources will be used on the fabric of the CS2112.

4.3.1 Unsigned 32-bit Integer Multiplication

The reconfigurable fabric of the CS2112 has two multipliers present in each tile. These multipliers operate on signed floating point values. With respect to RC6, the multipliers were used in the 16-bit mode, where two 16-bit operands multiply to generate a signed 32-bit result. The operation in RC6 is the unsigned integer multiplication $\bmod 2^{32}$. That is, two 32-bit operands multiply together to give a 32-bit result. To produce a $\bmod 2^{32}$ result, just the least significant 32-bits are taken.

To produce a 32-bit unsigned multiplier out of 16-bit signed multipliers, the operands must undergo special processing comprised of two steps. The first step

forms a partial product with the sign bits masked. The second step is to incorporate the effect of the sign bits into the partial product. The 32-bit operands must be split into four 16-bit operands. The most significant bit of each 16-bit operand must be masked because these represent sign bits to the multipliers on the fabric, and not magnitude bits. The 16-bit masked operands must then be multiplied and the results summed together after appropriate shifting to preserve magnitude of the intended multiplication. This process is illustrated in Figure 4.10 with the results of the 16-bit multiplications represented by 32-bit variables temp1, temp2, and temp3.

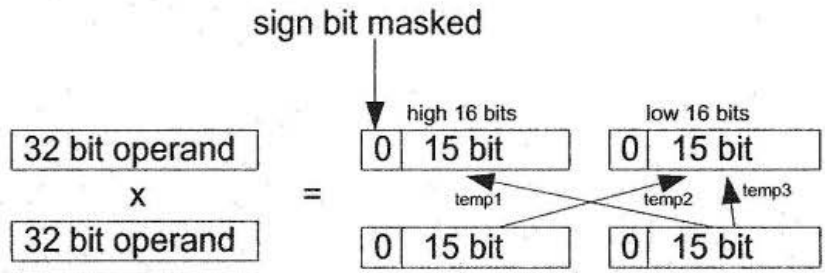

Figure 4.10: Creating a 32-bit unsigned integer multiplier.

Variables temp1 and temp2 must be logically shifted by 16-bit positions to account for the magnitude of the result. When temp1, temp2, and temp3 are summed we have a partial result that represents the unsigned integer multiplication mod 2^{32} of two 32-bit numbers, without the contribution of the sign bits that were masked off.

Consider now the effect of the sign bits that were masked off to obtain the initial partial product. The multiplication operation in RC6 is $X(2X+1) \bmod 2^{32} = (2X^2 + X) \bmod 2^{32}$ and can be viewed as squaring X, left shifting the result by one and an addition operation. Consider now the contribution of the sign bits to the operation $X^2 \bmod 2^{32}$. First let X' be defined as X with bits 16 and 32 being set to zero. The following expressions can be written:

$$X^2 \bmod 2^{32} = \left(X' + S_L 2^{15} + S_H 2^{31}\right)\left(X' + S_L 2^{15} + S_H 2^{31}\right) \bmod 2^{32} \quad (4.1)$$

$$\begin{aligned} X^2 \bmod 2^{32} &= (X')^2 + S_L(2^{15}X') + S_H(2^{31}X') + S_L(2^{30}) \\ &\quad + S_L S_H(2^{46}) + S_L(2^{15}X') + S_H(2^{31}X') \\ &\quad + S_H S_L(2^{46}X') + S_H(2^{62}) \bmod 2^{32} \end{aligned} \quad (4.2)$$

$$X^2 \bmod 2^{32} = (X')^2 + S_L(2^{16}X') + S_H(2^{32}X') + S_L(2^{30}) \bmod 2^{32} \quad (4.3)$$

$$X^2 \bmod 2^{32} = (X')^2 + S_L(2^{16}X') + S_L(2^{30}) \bmod 2^{32} \quad (4.4)$$

S_L and S_H are the values of the sign bits of X. Any terms with powers higher than 2^{31} drop out because the multiplication is mod 2^{32}. The final result of the X^2 simplifies to a X' term, and depending on the state of the sign bit, the addition of a shifted version of X' and a constant. The value $(X')^2$ is the partial product developed by the addition of shifted temp1, shifted temp2, and temp3. S_H has no effect in calculating X^2. It is worth noting that temp1 and temp2 are equivalent because we are calculating X^2. Based on the above result the following can be applied:

$$X(2X+1) \bmod 2^{32} = 2\left(X^2\right) + X \bmod 2^{32} \quad (4.5)$$

$$X(2X+1) \bmod 2^{32} = 2\left((X')^2 + S_L(2^{16}X') + S_L(2^{30})\right) + X \bmod 2^{32} \quad (4.6)$$

$$X(2X+1) \bmod 2^{32} = 2(X')^2 + S_L(2^{17}X') + S_L(2^{31}) + X \bmod 2^{32} \quad (4.7)$$

Given the arithmetic expression in Equation 4.7 for performing the $X(2X+1)$ operation with the sign bits masked, the datapath for the 32-bit unsigned integer multiplication using signed floating point 16-bit multipliers can be created in the reconfigurable fabric.

4.3.2 Iterative RC6 Design

Based on design experience with RC5, an investigation of performance of an iterative design of RC6 was undertaken. Many of the operations in RC6 are similar to RC5, therefore resource utilization and performance estimates can provide a rough guide when making implementation decisions. Figure 4.11 is an iterative design of a DPU configuration of Equation 4.7 from Section 4.3.1. The control logic resource usage was not addressed for the multiplier because datapath resources were deemed to be the biggest design restriction.

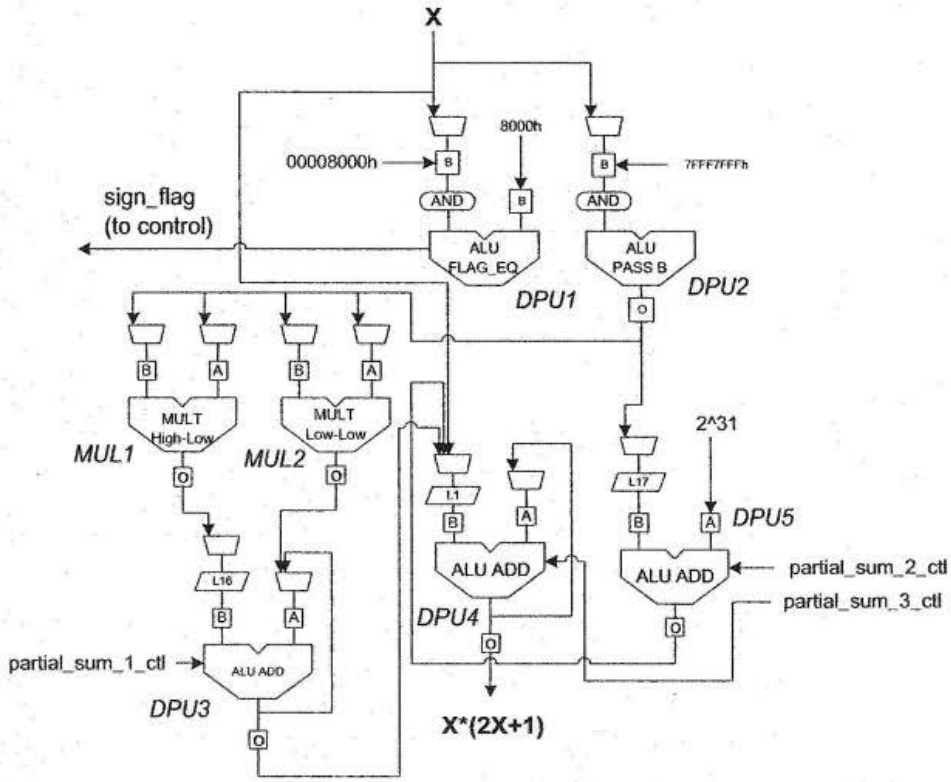

Figure 4.11: Iterative multiplier setup.

The number of DPUs required for the design of the multiplication module is estimated to be seven. Five DPUs are used in the multiplication, while 2 more are used for the fixed rotation of $\lll lg(w)$ required right after the multiplication

operation [17]. The following are the specific datapath elements required for an iterative unsigned multiplier:

- DPU1: Tests the state of bit 16. This information must be passed to the control module for the multiplier so that Equation 4.7 can be implemented.
- DPU2: Masks the sign bits of the input operand. The DPU set bit 16 and 32 to zero.
- DPU3: Creates the summation of temp1, temp2, and temp3. The DPU also provides the 16-bit left shift to add the proper magnitude to temp1 and temp2.
- DPU4: Adds in contribution of the sign bit. The DPU must either add or pass data depending on value of the sign bit.
- DPU5: Adds $S_L(2^{17}X') + S_L(2^{31})$ from Equation 4.7.
- MUL1: Multiplies high and low segments of the operand and produces temp1 and temp2 from Section 4.3.1. The operation being performed is $2(X')^2$, therefore temp1 and temp2 are equivalent.
- MUL2: Multiplies low segments of the operand and produces temp3 from Section 4.3.1.

A preliminary iterative datapath was designed. The resource usage for a single slice of RC6 is outlined in Table 4.6. Some DPUs can be removed by using one data dependent rotation module and by introducing control logic allowing it to accept multiple inputs from the datapath. Overall throughput would be decreased by such a choice because the rotation would be used for one part of the round and then another, effectively doubling the amount of time required by the rotation.

Based on Figures 4.7 and 4.11 it will take approximately 14 clock cycles for one round of RC6. Based on a 100MHz clock for the fabric and 20 rounds of operation, the

Resource	Count				Total
	Slice 0	Slice 1	Slice 2	Slice 3	
DPU	20	0	0	0	20
LSM	2	0	0	0	2
MUL	2	0	0	0	2

Table 4.6: Resource estimates for a single slice of RC6 in the fabric.

upper limit is 45.7 Mbit/s. If this design were to be copied to the remaining 4 slices and implemented to a parallel pipelined design as in Section 4.2.3, the throughput would be 182.8 Mbit/s.

4.3.3 Pipeline Primitives

Thus far, all designs have been designed in an iterative approach, or have used an iterative design as a basic building block. An investigation of a pipelined approach for RC6 was conducted. By considering a pipelined or rolled out approach, the most important design focus is to maximize the use of DPUs within a design. One of the main applications of the CS2112 is to focus on DSP applications where data flows through the fabric. An example of this type of application would be the use of a multiple tap, finite impulse response digital filter. For a pipelined version of RC6, ideally plaintext data blocks will enter the pipeline every clock cycle. With each clock cycle, each DPU will perform some operation on the data, and pass it on by the next clock cycle. The difference between this type of design approach and that of Section 4.2.3 is that the pipelined elements are individual DPUs, rather than groupings of DPUs.

A pipelined approach can provide an easier design approach with respect to the fabric of the CS2112. With a pipelined approach, most DPUs will have only one configuration in that they will perform the operation on new data every clock cycle, unless the pipeline is stalled waiting for information. Therefore, control logic is simplified from the viewpoint of managing various DPUs. Control resources may be

occupied with respect to the specific state of data on the pipeline. For example, in the case of the unsigned integer multiplication in RC6, it is advantageous to have the data stream through the multiplier.

In Section 4.3.2, the iterative multiplier would send sign bit information back to the controller which would then use control signals back to the datapath for correct operation. To prevent stalling the data flowing through the multiplier, the controller for the multiplier needs to retain sign bit information of incoming data. When the data reaches the part of the pipeline where the sign bit state matters, the controller will switch the DPU configuration. The controller needs a FIFO queue that is N spaces deep where N represents the number of clock cycles from when the sign bit is detected to where the information is relevant in the pipeline. It will be seen how this requirement is satisfied in the design of the pipelined RC6 multiplier module.

With an iterative design there is a hot spot of computational activity traveling through the datapath. Some DPUs are conducting relevant work while others are just holding their value for use elsewhere in the datapath. A pipelined approach also has DPU elements such that their sole purpose is to hold data. These DPUs are provided to delay data flowing through the pipeline so that it reaches all portions of the datapath at the proper times. Figure 4.12 illustrates a pipeline that will perform the following operation $D = A + (B + C)$ with and without DPUs used for delay.

The output values on the left pipeline are incorrect because there is no delay on the A operand. The addition of B and C takes one clock cycle and A must be delayed by one cycle as well. Since adding clock cycle delays to a pipeline is an important part of synchronizing the data flowing through the pipeline, it is important to see how delay was used in the design of a pipelined version of RC6. A one clock cycle delay is employed by using a DPU to pass its input value and load the output register [14]. Two clock cycle delays are created by using one DPU and having it load its respective input register, pass its value on, and load its output register. It takes one

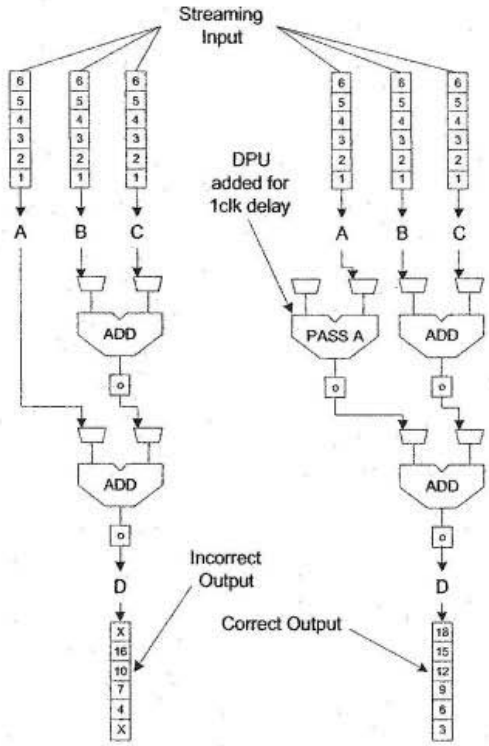

Figure 4.12: Need of delay in a pipeline.

clock cycle to load a DPU register, and by stringing together DPUs we can create an arbitrary N clock cycle delay [14].

DPUs are a fundamental resource within the fabric of the CS2112 and it is wasteful to create a 20 clock cycle delay using 10 DPUs. Not only would this occupy approximately 12% of DPUs in the fabric but it would add to the complexity of creating a routable design. A more space efficient way to implement an arbitrary length clock cycle delay is to use two DPUs and an LSM to create a first-in, first-out data queue. Data can be read from an LSM every clock cycle, therefore the queue must be N spaces deep with each space holding a 32-bit data value. Figure 4.13 is an illustration of the fabric resources involved with the creation of a FIFO data queue.

The DPU illustrated on the left side of the LSM in Figure 4.13 is a write address generator that writes to addresses $4N$ ahead of the read address generator, which

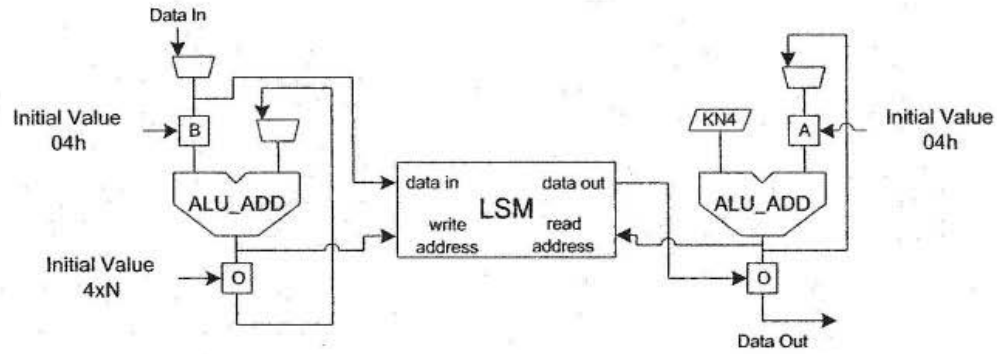

Figure 4.13: First-in, first-out queue setup.

is illustrated to the right of the LSM. The LSMs are addressed in byte wide locations, when the port size is set to 32-bits the address must be incremented by four places. Physical addressing for the LSMs is mod 2^9 allowing the FIFO queue to operate without any control logic. The FIFO queue can be 128 spaces deep resulting in a maximum delay of 128 clock cycles with this setup. There is a three clock cycle latency from when a read request is given and when information comes out of the read address generator accessing the LSM. A minimum of a three clock cycle delay can be used with the setup illustrated in Figure 4.13, but since it requires more resources than using DPUs for such a small delay (two DPUs and a LSM verses two DPUs), this setup is only useful for delays greater than four clock cycles.

4.3.4 Pipelined Multiplication

Figure 4.14 is a block diagram illustrating the pipelined unsigned integer multiplication module and a Verilog description can be found in Section A.6. The square blocks represent delay elements with their delay value illustrated within the block.

The multiplication module did not use the FIFO data queue structure as illustrated in Figure 4.13. These structures will be needed in the RC6 datapath, due to long delay requirements. The function of the remaining DPUs are described as follows:

- DPU1: Tests the state of bit 16. This information must be passed to the control module for the multiplier so that Equation 4.7 can be implemented.
- DPU2: Masks the sign bits of the input operand. The DPU will set bits 16 and 32 to zero.

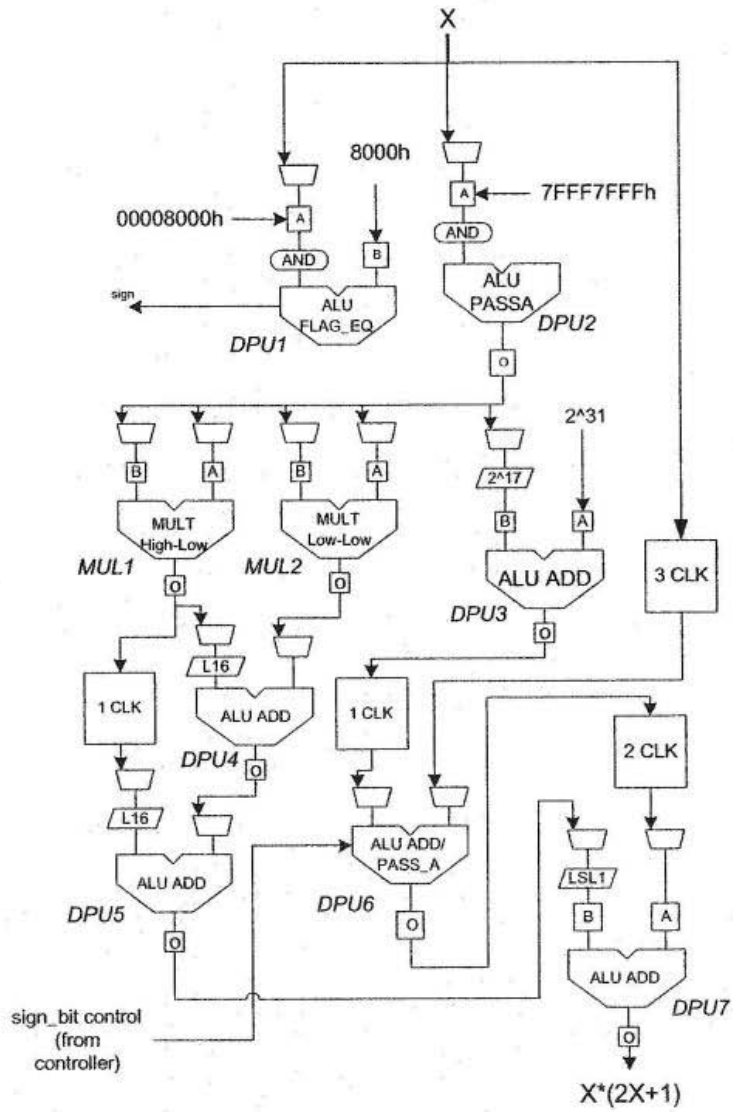

Figure 4.14: Pipelined multiplier module.

- DPU3: Adds $S_L(2^{17}X') + S_L(2^{31})$ from Equation 4.7.
- DPU4: Adds temp1 and temp3 variable in the multiplication process.
- DPU5: Adds temp2 to the partial sum of temp1 and temp3.
- DPU6: Adds the contribution of the sign bit. The DPU passes through X only if the sign bit is 0 and adds $S_L(2^{17}X') + S_L(2^{31})$ from DPU3 if sign bit is 1.
- DPU7: Creates the final summation to produce $X(2X + 1) \bmod 2^{32}$.
- MUL1: Creates temp1 and temp2 partial multiplication products because temp1 is equivalent to temp2.
- MUL2: Creates temp3 partial multiplication products.

An important part of the multiplier module is the control logic associated with controlling the process of the multiplication depending on the state of the sign bit. As stated previously, there is need for a control queue that will keep track of the sign bit of incoming data. Figure 4.15 is a high level illustration of how the pipelined multiplier and controller interact.

The control module was created by defining a finite state machine that took the sign bit in as an input. The queue assigns to three state registers within the controller, with the third state register being the output of the controller back into the datapath. The Verilog description of the multiplier controller is in Section A.5.

The control register queue actually contains the required CSM instructions for the multiplier datapath. In all states within the controller, the sign flag is checked and then the first position within the control queue is assigned. With each clock cycle the CSM instructions are moved forward and assignment to the next state register is made. The output of the control queue can then be tied directly into the controlling DPU in the multiplier datapath.

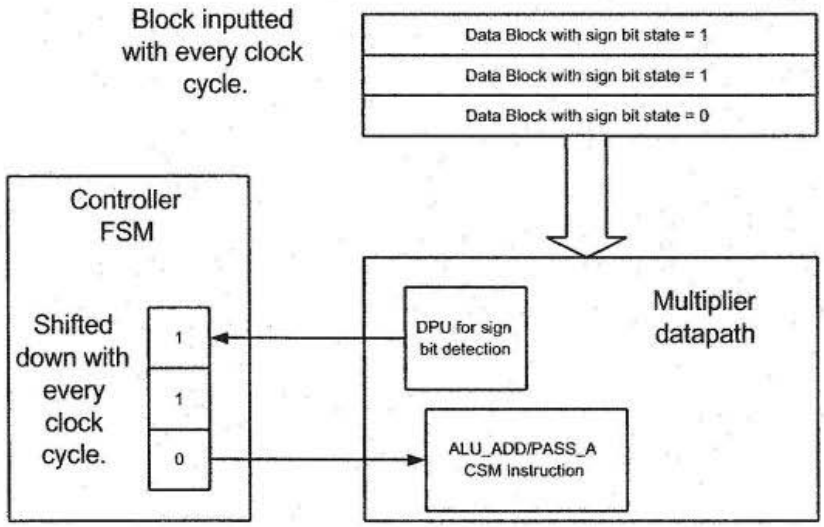

Figure 4.15: Multiplier and controller interaction.

After the operation of $X(2X+1)$ is performed, RC6 calls for a static rotation of the product of the multiplier module. The rotation is $X(2X+1) \lll lg(w)$ where $lg(w)$ is the log base 2 of w (which is 32 in this implementation), resulting in a rotation by five bit positions to the left. Figure 4.16 illustrates the fixed rotation structure.

4.3.5 RC6 Full Pipelined Design

A good way to conceptualize the pipelined datapath for RC6 is an assembly line with a finite amount of space. Given the available resources of the reconfigurable fabric, the pipeline was designed to allow one round of RC6. The pipeline can fill up with independent plaintext blocks until the depth of the pipeline is reached. If it takes N clock cycles for an input block to reach the end of the pipeline, we can fit an additional $N-1$ blocks of data in behind the initial block. Once the pipeline is filled, the output block and every block thereafter will feed back into the input of the pipeline until each block of data has been passed through the appropriate number of rounds. In effect the pipeline becomes circular for the remaining rounds. Top level control logic is required to increment subkey values, initialize input and output

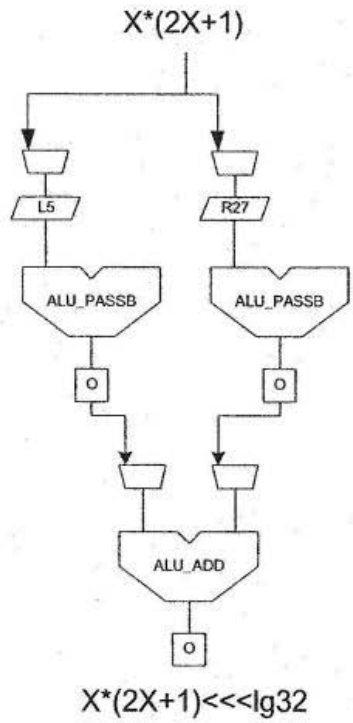

Figure 4.16: Fixed logical rotation by five bits.

memories and broadcast other datapath control signals across the fabric. The RC6 pipeline assumes that the key is the same for all words in the pipeline. If different keys were required for data inside the pipeline, redesign of the datapath, controller and C code, that the ARC processor uses to derive the subkeys, would be required.

Figure 4.17 is an abstracted diagram of the RC6 datapath. Delay elements are given as square blocks, multiplier/fixed rotation, and data dependent rotations are abstracted as blocks with their respective operations labeled within. Other elements are drawn as DPU structures. The input words of 32-bit plaintext are labeled A,B,C, and D with the output blocks labeled in the same fashion. The diagram illustrates one round of RC6 and does not show the feedback of the output of the datapath into the input, nor does it show the input and output memories involved with the operation.

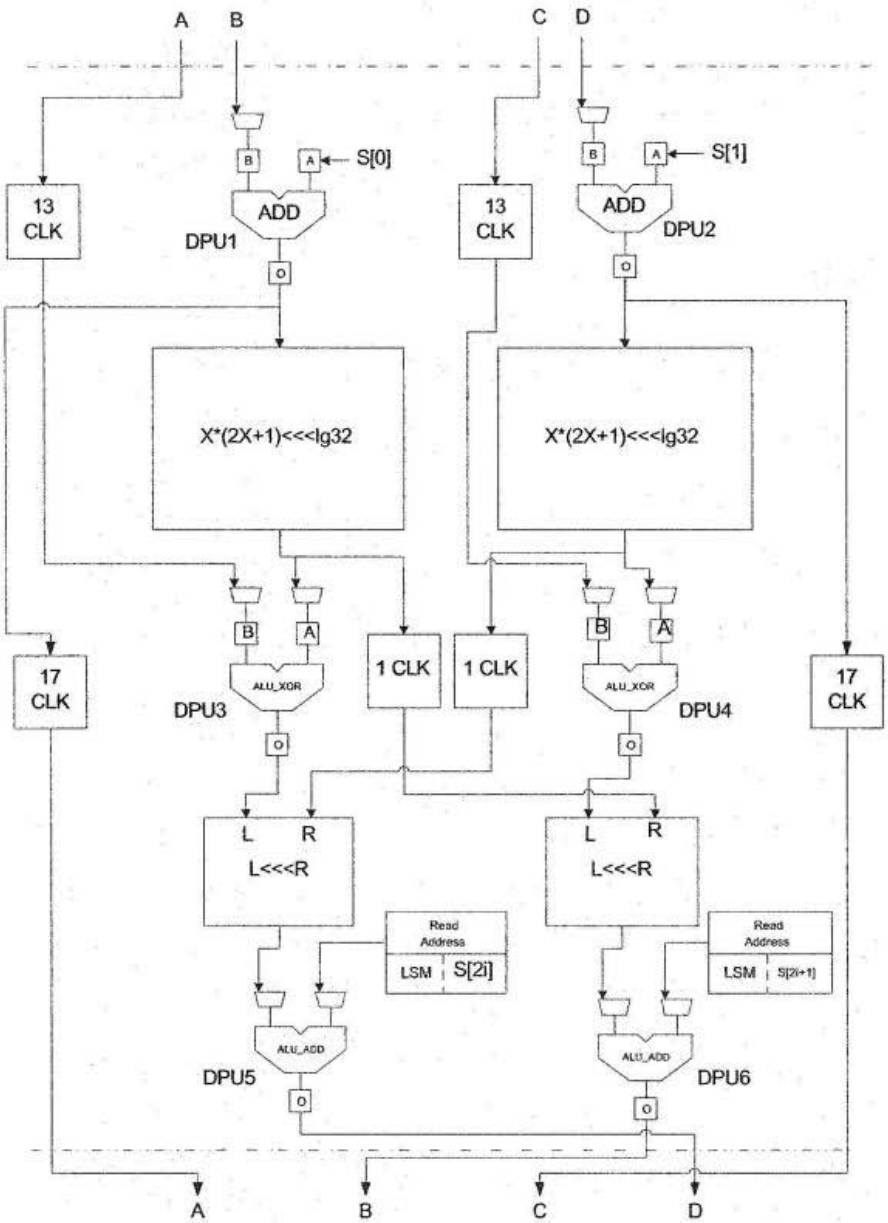

Figure 4.17: One full round of RC6.

The function of the following DPUs in the pipeline of Figure 4.17 are described as follows:

- DPU1: Adds $S[0]$ to B of the plaintext in the first pass and in subsequent passes it will register the data.
- DPU2: Adds $S[1]$ to D of the plaintext in the first pass and in subsequent passes it will register the data.
- DPU3: Executes XOR operation required in Figure 2.7.
- DPU4: The bit-wise exclusive OR operation required in Figure 2.7.
- DPU5: Adds the $S[2i]$ subkey to the data.
- DPU6: Adds the $S[2i+1]$ subkey to the data.

Figure 4.18 is high level block diagram showing all control logic and how they communicate with the datapath module. Since the RC6 pipeline is spread across the entire fabric, some signals from the top level controller must be broadcast across the fabric. In Figure 4.18, these signals are shown as dashed lines. Broadcasted signals enter into a small controller (two or three states) that control their local DPUs.

The pipeline for RC6 is 19 clock cycles deep. This is evident from the enabled registers on the initial DPU (input and output, giving two clock cycles of delay) and the 17 clock cycle delay element that provides the output. The timing information for this datapath can be found in Table 4.7.

RC6 Pipeline Timing	Clock Cycles
Start to Done Time	407
Data Dependent Rotation	3
Multiplication	9
Fixed Rotation	2

Table 4.7: Timing information for the RC6 pipeline.

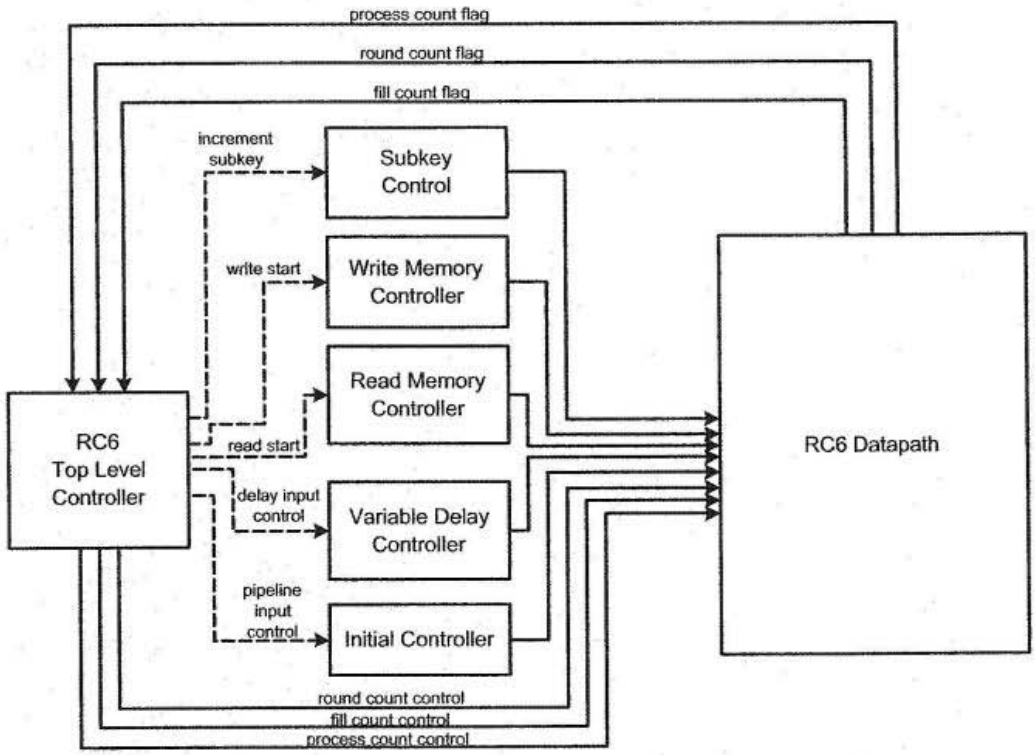

Figure 4.18: Description of control and datapath interaction.

Figure 4.19 is a layout floor plan of the reconfigurable fabric captured from the floorplanner in the C~SIDE tool set. Slices two and three are mainly used for the pipelined multipliers. Slices one and four contain counters and datapath logic for rotation, addition, and read/write memories for operation. Slice four contains read and write memories for operation and remaining datapath logic.

Resource	Count				Total
	Slice 0	Slice 1	Slice 2	Slice 3	
DPU	19	21	21	16	77
LSM	4	3	3	4	14
MUL	0	2	2	0	4

Table 4.8: Resource usage for a fully pipelined RC6.

Table 4.8 is a summary of the resources used within the fabric for the pipelined RC6. This version heavily utilizes the fabric, especially with respect to DPU usage,

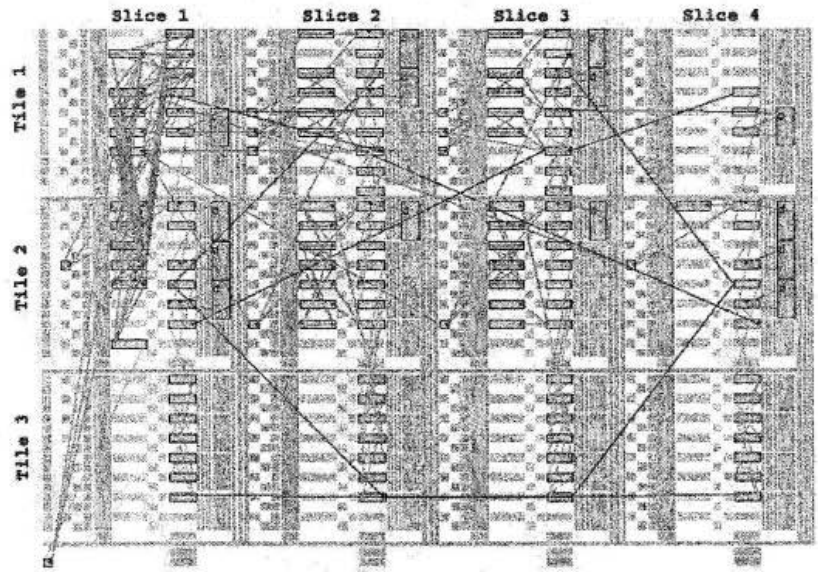

Figure 4.19: RC6 pipeline floorplan.

and in general would constrain larger designs of this nature. Control logic for the pipelined version of RC6 consisted of the resources illustrated in Table 4.9. The C code for RC6, along with the hardware call to the CS2112, for testing within Chipsim or in the development module, can be found in Section B.2.

Controler FSM	Instantiations	States	Internal Registers	Inputs	Outputs
Multiplier	2	7	6 (3-bit)	1 (1-bit)	1 (3-bit)
Read LSM	4	2	2 (3-bit), 2 (1-bit)	1 (1-bit)	1 (3-bit)
Write LSM	4	2	2 (3-bit), 2 (1-bit)	1 (1-bit)	1 (3-bit)
Subkey	2	2	2 (3-bit), 2 (1-bit)	1 (1-bit)	1 (3-bit)
Delay	2	1	2 (3-bit), 2 (1-bit)	1 (1-bit)	1 (3-bit)
RC6 Top	1	24	2 (5-bit), 6 (3-bit), 12 (1-bit)	3 (1-bit)	8 (3-bit)

Table 4.9: Control logic resource usage for a fully pipelined RC6 design.

The pipelined version of RC6 was fully simulated, tested using the CS2112 simulator, and the development board. Based on waveform output support of the CS2112 development board and simulation results, the RC6 pipeline operates at 597.5 Mbit/s, encrypting 19 sets of plaintext at once in the circular pipeline.

4.3.6 Summary of RC6 Designs

The CS2112s architecture works well with a pipelined design because the datapath is more efficiently used. With an iterative design, a hot spot of activity iterates through the datapath with only a small portion of the datapath elements performing operations while the rest are holding data. With respect to the pipelined version of RC6, the datapath is more efficiently used because most of the DPUs are doing useful work each clock cycle, with the exception of DPUs and LSMs used for delay.

The biggest restriction to the performance of RC6 is that the unsigned integer multipliers take up roughly 50% of the resources of the fabric. Having unsigned integer multipliers on the fabric would save resources. It would be advantageous to have a method of accessing LSMs without using DPUs to generate read and write addresses. If this were the case, 18 DPUs (8 for read/write memories, 8 for FIFO delay elements, and 2 for subkey storage) accounting for 21.4% of the DPUs on the fabric, would be available for computational purposes.

4.4 Summary

This chapter contained various methods and design philosophies for implementing symmetric block ciphers on the CS2112. As illustrated in Table 4.10, RC5 was designed with an iterative nature in mind. RC5 first started with a simplistic version, and built up to a design that fully utilized the resources of the reconfigurable fabric. Fabric resources were used more effectively with a pipelined version (with an iterative core) of RC5, and speed increased significantly.

RC6 was evaluated in both an iterative and a pipelined fashion. A full fabric pipelined design was implemented once it was deemed that fabric resources would be adequate to accommodate a full pipelined round of RC6. The pipelined structure of RC6 yielded the highest speed of all designs.

Implementation	DPUs Used	Speed Mbit/s
RC5 Simple Iterative	12	40.7
RC5 Two Half Round	19	65.3
RC5 Full Fabric	72	237
RC6 Iterative Full Slice	20	45
RC6 Iterative Full Fabric	80	182.8
RC6 Pipelined Full Fabric	77	597.5

Table 4.10: Summary of block ciphers on the CS2112.

The next chapter will explore the viability of hash algorithms within the reconfigurable fabric of the CS2112. While hash functions have many of the primitive operations of symmetric block ciphers, due to the amount of processing required only the compression function of SHA-1 and MD5 were implemented.

Chapter 5

Evaluation of Message Digest Algorithms

It is the purpose of this chapter to provide an evaluation of the suitability of the reconfigurable architecture of the CS2112 with respect to two popular message digest algorithms. MD5, as discussed in Section 2.6, will be the first algorithm explored. SHA-1 which was discussed in Section 2.7 will be the second algorithm discussed. With respect to both algorithms, an estimate of resource usage along with implementation issues will be addressed.

5.1 MD5 Implementation

Based on the complexity of the MD5 algorithm, it is feasible to use an iterative kernel that will provide functionality for the compression function H_MD5 only. The ARC processor will have to properly format the arbitrary length message for use with the fabric function. Within the function, there are a set of auxiliary functions that are bit-wise operation and are used within different steps of H_MD5. The auxiliary functions are defined as follows [19]:

- $F(X,Y,Z) = X\&Y|NOT(X)\&Z$
- $G(X,Y,Z) = X\&Z|Y\&NOT(Z)$

- $H(X,Y,Z) = X \oplus Y \oplus Z$
- $I(X,Y,Z) = Y \oplus (X|NOT(Z))$

The reconfigurable fabric can be used for these functions, according to the depiction in Figures 5.1 and 5.2, showing the datapath of the auxiliary functions. The solid black lines represent clock cycle delays of data flow. Where possible, data should be synchronized for easier timing of the datapath.

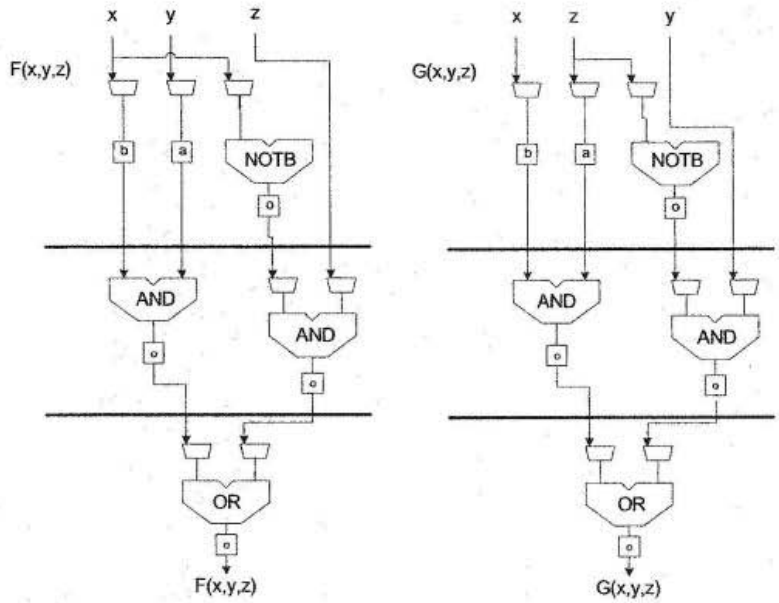

Figure 5.1: Implementation of F and G functions.

A preliminary design of the datapath for the compression function H_MD5 with the variable rotation and auxiliary function abstracted, is shown in Figure 5.3. The variable rotation is the version used in RC6 using five DPUs. The five DPU version of the variable rotation uses one more than the iterative version found in the RC5 designs, but does not require control logic. The rotation amounts for the execution of H_MD5 are values that are stored within a LSM. The setup of T and X are done by the ARC processor initially when the kernel call is made. In Figure 5.3 it is the

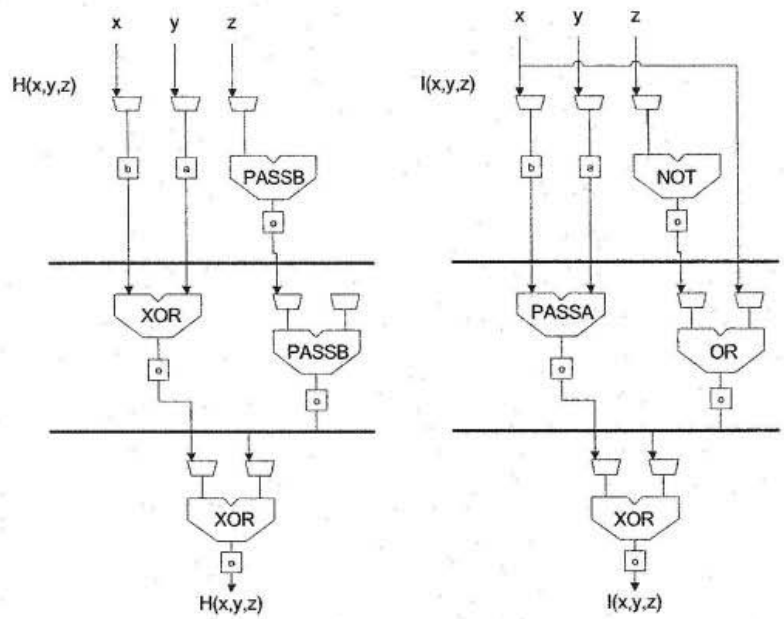

Figure 5.2: Implementation of H and I functions.

role of four DPUs to store the registered values A, B, C, and D. The role of these four DPUs is to buffer and permute the A, B, C, and D in each step and to do the final addition after the four rounds are completed. The original value is stored in the A register of each DPU while the permuted and feedback values are stored in the B register of the DPUs. The addition is performed by adding the registers together.

5.1.1 Performance and Usage Estimates for MD5

A rough estimate of hardware usage is given in Table 5.1. These estimates are based upon preliminary investigation of a datapath. As the final design was not implemented, estimates of CLU resource usage for the MD5 datapath are not available.

The design almost fills up an entire slice in the fabric. As was discovered with RC5 and RC6, an iterative solution does not make efficient use of the fabric from a computational point of view, but the size of the fabric and the nature of MD5 does not allow for a fully pipelined version of the algorithm. This preliminary analysis

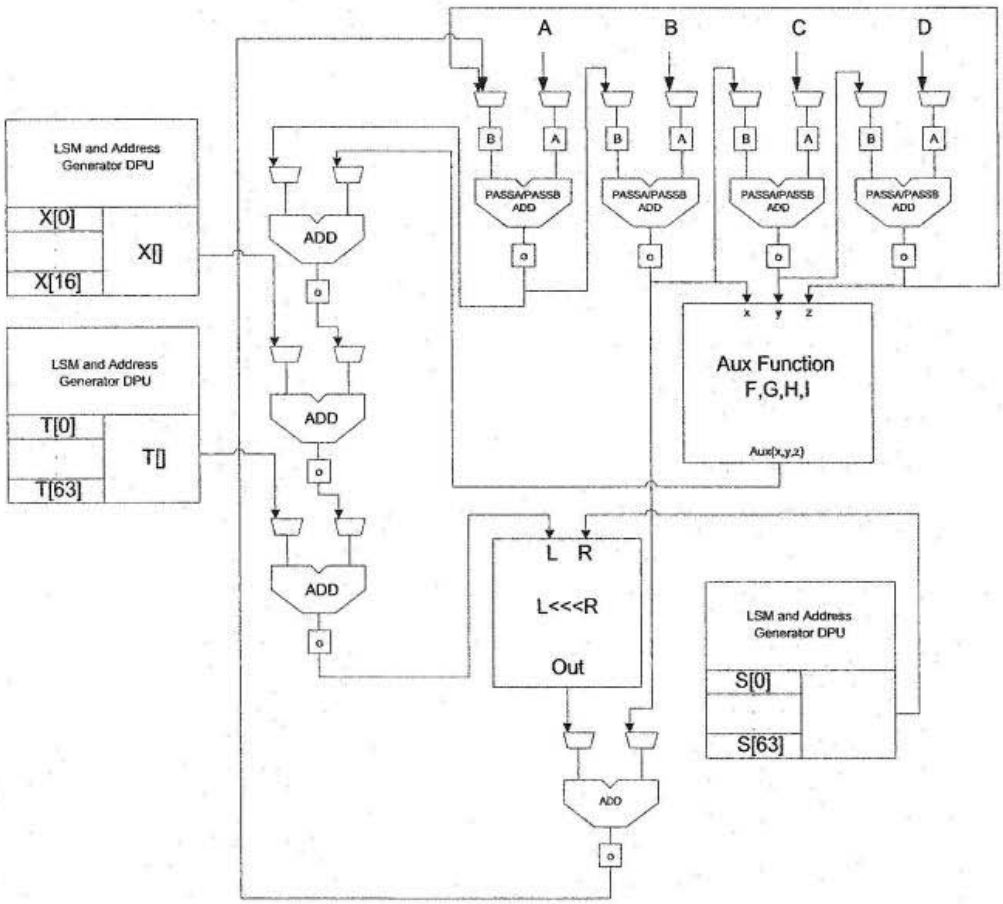

Figure 5.3: A proposed MD5 datapath for one step of H_MD5.

does not take into account the DPUs used for control purposes, such as counters. There is a strong possibility that a solely finite state machine controller will not be realized within the state bits of one slice's CLU, and the MD5 design will span two slices. The number of DPUs in the critical path of data flow will determine how fast MD5 will run when the hardware function call is made. Table 5.2 is an analysis of the number of clock cycles required to get through the flow of the datapath.

It takes approximately 13 clock cycles to get through one of sixteen steps in a round, with four total rounds in MD5. Therefore, it will take about 832 clock cycles to process one 512-bit block of the formatted message. Assuming a 100 MHz

Function	Number of DPUs
Rotation	5
Auxiliary Function	4
Registered Inputs for Inits	4
Other DPUs For Other Arithmetic Ops	4
DPUs for LSM Address Generators	3
LSM Memories Used	3
TOTAL	20

Table 5.1: Resource usage for preliminary MD5 implementation.

Delay Through Datapath	Clock Cycles
Initial Stage	2 clk
Auxiliary Function	3 clk
Three Additions	3 clk
Variable Rotation	4 clk
Final Addition	1 clk
Total	13 clk

Table 5.2: Delay through MD5 datapath.

clock in the fabric and operating on a 512-bit message block, the throughput is 61.5 Mbit/s. Software calls to the hardware function H_MD5 will be made in processing the arbitrary length message, therefore the actual performance of the algorithm will be considerably slower due to latency in the ARC processors' execution, and DMA latency of transferring the appropriate data to and from the fabric.

5.2 SHA-1 Implementation

SHA-1 is partially based on MD5 and as a result has many similarities. Section 2.7 outlines the SHA-1 algorithm. As with MD5, it is practical to implement one of 80 steps involved with compression function H_SHA1 as an iterative fabric function.

5.2.1 Recursive Array Expansion

During the execution of H_SHA1, $W[0..15]$ is recursively expanded into an 80 element array by the following operation:

$$W[t] = W[t-3] \oplus W[t-8] \oplus W[t-14] \oplus W[t-16] \, (t > 15)$$

Since this occurs with each call to H_SHA1, it makes sense to provide this function into the hardware of H_SHA1. Figure 5.4 illustrates a datapath for the expansion of W.

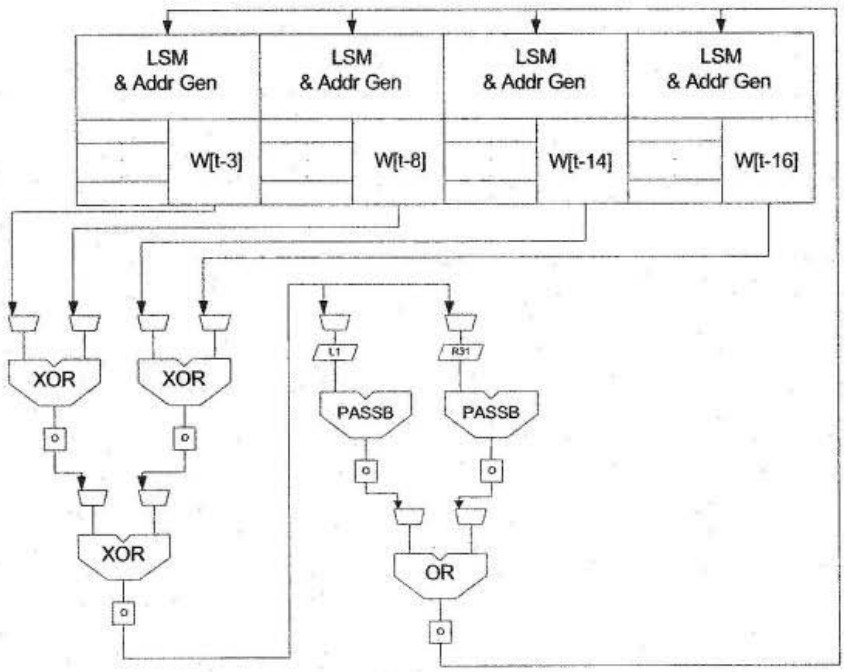

Figure 5.4: Recursive expansion of $W[0..15]$ to $W[0..79]$.

The operation of this module can be considered in two ways: first the expansion of W can be performed, and then the rest of the algorithm can be carried out. A total of 64 entries (entries 16 to 79) need to be filled, with each entry taking 4 clock cycles. The overhead of reading from an LSM is only at the start resulting in a one-time cost of three clock cycles. Therefore the total time to fill W is 259 clock cycles.

The second method of filling W is that it can be performed while other parts of the

datapath are processing. For performance analysis, it must be determined whether the module that expands W will be able to stay ahead and supply $W[t]$ values for all 80 steps of the kernel function. This point of view looks to expand W in parallel with the main datapath of H_SHA-1. The module contains $W[t]$ values for $t = 0$ to 15, so for the first 15 steps the datapath can proceed. During this time a entry in W can be filled every 4 clock cycles. The timing of the rest of the datapath must be determined next.

5.2.2 SHA-1 Auxiliary Function Design

As with MD5, SHA-1 makes use of bitwise auxiliary functions during its execution. These are defined as follows [7]:

- $F0(X, Y, Z) = (X\&Y)|((NOT(Y))\&Z)$
- $F1(X, Y, Z) = X \oplus Y \oplus Z$
- $F2(X, Y, Z) = (X\&Y)|(X\&Z)|(Y\&Z)$
- $F3(X, Y, Z) = X \oplus Y \oplus Z$

Figure 5.5 and 5.6 are flow diagrams illustrating configurations of each DPU involved with the implementation. The black line separates clock cycles of timing. Note that $F2$ reuses a DPU by using a feedback to the ALU.

A significant difference with respect to SHA-1 from MD5 is the auxiliary function $F3$. The computation takes 4 clock cycles for $F3$ because ALU feedback is utilized so that a DPU can be reused. The choice to use ALU feedback was made to minimize resource usage. With respect to a performance analysis it will be assumed that four clock cycles are needed to execute the auxiliary function, since this is the worst case time for this execution.

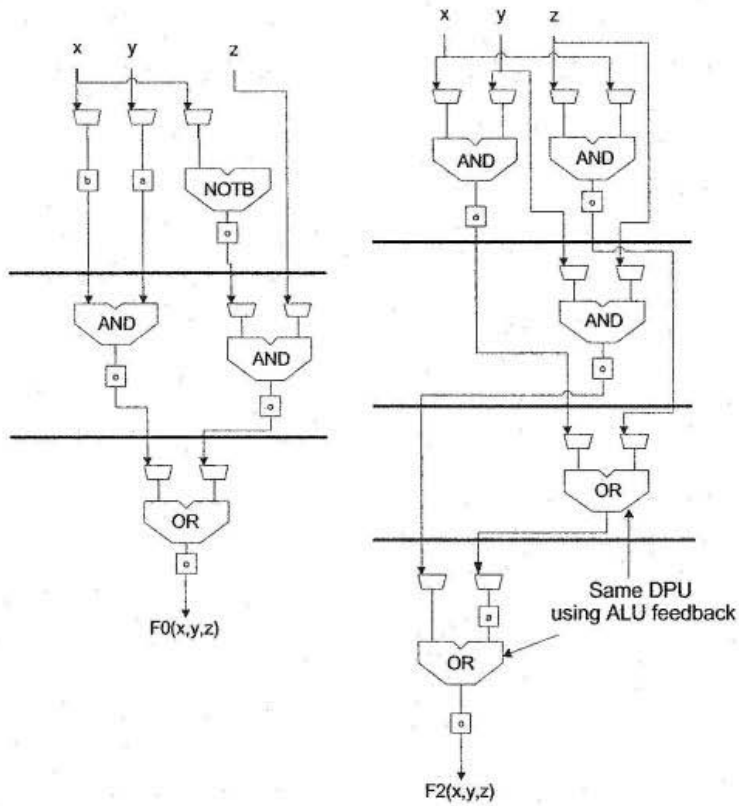

Figure 5.5: Auxiliary function implementation.

5.2.3 Full SHA-1 Datapath

Figure 5.7 illustrates a full iterative design for one step of 80 for the H_SHA1 kernel function. Some of the differences between MD5 and the SHA-1 that cause SHA-1 to use more resources than MD5 are:

- The use of static rotation instead of a variable rotation. This will use less resources than a data dependent rotation.
- A different auxiliary function definition than MD5. In the case of SHA-1, the auxiliary function takes one extra clock cycle, and requires some control logic for ALU feedback mode. This will affect performance.

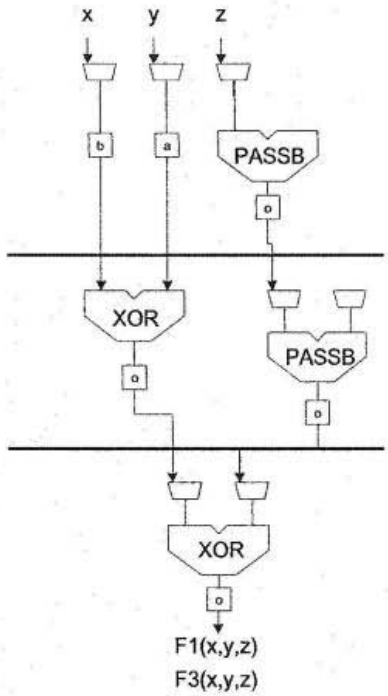

Figure 5.6: Auxiliary function implementation.

- The recursive expansion of the W array which is derived from formatted arbitrary length message. The method of expanding W is recursive and is derived from the 512-bit message block. The implementation of this requires a sizable portion of resources within the fabric.

- The use of five 32-bit registers to produce a 160-bit message digest adds some extra complexity and resource usage to the design.

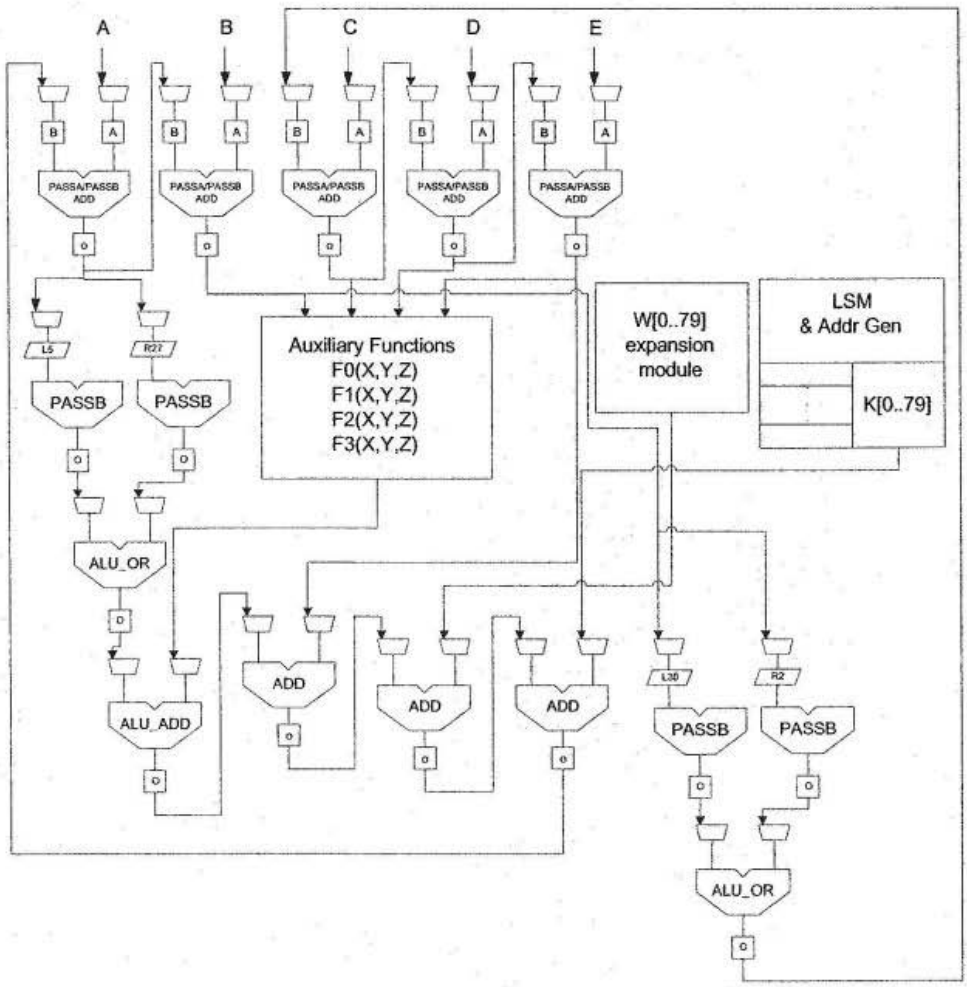

Figure 5.7: SHA-1 datapath design.

5.2.4 Performance and Resource Usage of SHA-1

Table 5.3 is a summary of the resource utilization of the design outlined in Section 5.2.3. A considerable amount of control logic will be required to operate the configuration changes of the auxiliary functions and to operate the expansion of W. Two slices will be used with SHA-1, including the expansion of W. There is enough time (four clock cycles are required to provide one W entry) for the expansion of W to occur while the datapath is processing through the rest of the compression function, which takes approximately 10 clock cycles. Therefore, each call to the kernel function will take 800 clock cycles if the expansion of W is conducted in parallel. Assuming 100MHz clock for the fabric we can expect about 64 Mbit/s of throughput by processing a 512-bit message block in 800 clock cycles.

Function	Number of DPUs Used
W[] Expansion	10
Auxiliary Function	4
Top Level Datapath	16
Total	30
LSMs Used	5

Table 5.3: Resource utilization of SHA-1.

5.3 Comparison of SHA-1 and MD5 Implementations

Since MD5 and SHA-1 are similar algorithms, it is useful to make comparisons between their implementations in evaluating the CS2112. The performance of both SHA-1 and MD5 are similar and are in the range of 60 Mbit/s to 65 Mbit/s. The biggest deviation is the amount of resources used when implementing the algorithm. MD5 uses an array of 32-bit words from the $sin()$, function while SHA-1 uses an

array of 32-bit words from a recursive expansion of elements from the 512-bit message block. The implementation of expanding W in the fabric function is the reason SHA-1 has more resource utilization than MD5.

5.4 Summary

This chapter explored the design of hash algorithms on the CS2112. Resource usage and performance figures are given in Table 5.4. Hash algorithms exhibit many of the primitive operations that can be found in symmetric key block ciphers. Due to the size of the compression functions used in both algorithms in this chapter, an iterative solution was the only design choice. The next chapter will summarize the results of this research and provide some insight into algorithm implementation, optimization, and specific architectural considerations for the CS2112.

Design	DPUs Used	Speed Mbit/s
MD5 Iterative Single Slice	20	61.5
SHA-1 Iterative Two Slice	30	64

Table 5.4: Summary hash algorithms on the CS2112.

Chapter 6

Summary and Conclusions

6.1 Summary of Results

The results for all cryptographic algorithms on the CS2112 are found in Table 6.1.

Implementation	DPUs Used	% of Total DPUs	Performance Mbit/s
RC5 Simple Iterative	12	14.30%	40.7
RC5 Two Half Round Full Slice	19	22.60%	65.3
RC5 Full Fabric	72	85.70%	237
RC6 Iterative Full Slice	20	23.80%	45
RC6 Iterative Full Fabric	80	95.20%	182.8
RC6 Pipelined Full Fabric	77	91.70%	597.5
MD5 Iterative Single Slice	20	23.80%	61.5
SHA-1 Iterative Two Slice	30	35.70%	64

Table 6.1: Summary of designs on the CS2112.

The pipelined version of RC6 was the best performer of the block ciphers with a speed more than twice that of the pipelined (with an iterative core) version of RC5. Both versions used roughly the same amount of DPUs, while RC5 has a more simplistic iterative round structure than RC6.

The structure of the CS2112 fabric is more suited to the pipelined or unrolled implementation of ciphers. The application space for the CS2112 is for streaming DSP [14], and telecommunications applications [13], making pipelining the most efficient use of resources.

The performance of the hash algorithms in this research was almost equivalent.

However, SHA-1 used more resources due to extra processing within the fabric. The hash algorithms were similar in structure and were iterative in nature due to limitations imposed by the available resources within the fabric.

6.2 CS2112 Architectural and Support Features

The fabric of the CS2112 is rich in operational features and many of these features can be applied to the area of cryptography. Many of the arithmetic operations can be found within one reconfigurable unit in the fabric, the datapath unit. Operations such as the bit-wise exclusive OR, logical masking operations, and barrel shifting can also be accomplished with one DPU.

With respect to the CS2112 the following features were found to be lacking within the reconfigurable fabric:

- Support for a single clock cycle data dependent rotation within one DPU. The designs required 5 DPUs or 4 DPUs with possible associated control logic to implement a data dependent rotation. The design of a data dependent rotation also took more than one clock cycle to complete the operation. If this could be accomplished within one clock cycle, the speedup of the data dependent rotation would be 2 to 3 times, and resource usage will be decreased by 75%-80%.

- Support for a unsigned 32-bit integer multiplication structure. RC6 requires this operation and of all the resources used for the pipelined RC6 design, roughly 50% of the fabric was utilized for the unsigned integer multipliers. If a one-clock cycle 32-bit unsigned multiplication module were to be used, a speedup of 9 times would be achieved for the multiplication.

- While memory requirements were adequate for iterative block ciphers, the need of a DPU for address generation when accessing an LSM is wasteful.

All algorithms implemented in this research were originally designed to perform well on general purpose processors. The algorithms use 32-bit words, with 32-bit operations and manipulations. The fabric of the CS2112 has features which exploit the characteristics of algorithms that were created with software in mind. The results from this research cannot be expanded to algorithms designed for specific architectures and platforms because no such algorithms were investigated.

Communication within the fabric is a mix of local and global buses in a two dimensional arrangement, allowing for the creation of pipelined and iterative structures. With respect to reconfigurable architectures in general, the data width of processing elements is an important feature. For all algorithms explored in this research, primitive operations were all carried out as 32-bit operations, allowing easy translation to the reconfigurable fabric of the CS2112.

The use of communication and control resources was a design consideration for all the algorithms, but did not cause problems that required redesign of the datapath. The C~SIDE tool set provided by Chameleon Systems gives a full implementation platform, including simulation fabric mapping tools. The biggest drawback with using the C~SIDE tools was that the mapper did not use any intelligent mapping algorithms and for designs exceeding five or more DPUs, all mapping was done manually. The process of mapping also included placement of control logic.

6.3 Considerations For Future Work

The study of symmetric key block ciphers and hash algorithms was carried out for the purposes of determining the suitability of the Chameleon Systems CS2112 for hardware based cryptographic algorithms. In 2003 Chameleon Systems ceased to exist as a corporate entity, and the CS2112 and related technology from Chameleon has not found its way into the market. The suitability of coarse grain architectures

with respect to run time reconfigurability, quick design, shorter time to market, and functional flexibility, remain as motivating factors for further research in the design of cryptographic algorithms in coarse grain environments.

References

[1] *Online Banking Goes Mainstream in United States.* NUA Web Site: http://www.nua.ie/surveys.

[2] *The Internet Economy Indicators.* Web Site: http:// www. internetindicators. com/keyfindings.html.

[3] E. A. Fisch and G. B. White, *Secure Computers and Networks: Analysis, Design, and Implementation.* CRC Press, 2000.

[4] D. Kahn, *The Code Breakers: The Story of Secret Writing.* Scribner, 1996.

[5] S. Singh, *The Code Book.* Doubleday, 1996.

[6] S. E. Forrester, "Security in data networks," *BT Technology Journal*, vol. 16, no. 1, pp. 52–75, 1998.

[7] *Federal Information Processing Standards Publication 180-1 1995 April 17 Announcing the Standard for Secure Hash Standard.* Nation Institute of Standards and Technology, 1995.

[8] B. C. AJ Elbrit, W Yip and C. Paar, "An FPGA implementation and performance evaluation of AES block cipher candidate algorithm finalists," in *AES3: The Third Advanced Encryption Standard Candidate Conference*, 2000.

[9] L. Wu, C. Weaver, and T. Austin, "CryptoManiac: a fast flexible architecture

for secure communication," in *Proceedings of the 28th annual international symposium on on Computer architecture*, pp. 110–119, ACM Press, 2001.

[10] L. E. Frenzel, "Cryptochips: Help eliminate the security bottleneck," *Electronic Design*, March 2003.

[11] M. J. S. Smith, *ASICs...The Web Site.* http:// www-ee.eng.hawaii.edu / msmith/ASICs /HTML/ASICs.htm.

[12] A. Dandalis, V. K. Prasanna, and J. D. P. Rolim, "A comparative study of performance of aes final candidates using FPGAs," in *AES3: The Third Advanced Encryption Standard Candidate Conference*, 2000.

[13] R. Hartenstein, "Coarse grain reconfigurable architecture (embedded tutorial)," in *Proceedings of the conference on Asia South Pacific Design Automation Conference*, pp. 564–570, ACM Press, 2001.

[14] *Chameleon Systems CS2112 User Manual.* Chameleon Systems Incorporated, 2001.

[15] *Integrated IPSec/MPLS Services and SSL-Based VPNs Fuel Solid Growth in VPN.* Infonetics Research: http: // www.infonetics.com/ resources/.

[16] *Chameleon Systems Inc. - Memorial University Research Agreement.* Chameleon Systems Inc., October 2000.

[17] R. L. Rivest, "The RC5 encryption algorithim," in *Proceedings of the 1994 Leuven Workshop on Fast Software Encryption*, pp. 86–96, 1995.

[18] R. L. Rivest, M. Robshaw, R. Sidney, and Y. Yin, *The RC6 Block Cipher.* 1998.

[19] R. L. Rivest, *The MD5 Message Digest Algorithm.* 1992.

[20] S. A. V. Alfred J. Menezes, Paul C. van Oorschot, *Handbook of Applied Cryptography.* CRC Press, 1997.

[21] H. Krawczyk, M. Bellare, and R. Canetti, *HMAC: Keyed-Hashing for Message Authentication.* RFC Editor, 1997.

[22] *National Institute of Standards and Technology.* NIST Website: http://www.nist.gov.

[23] *Computer Security and Industrial Cryptography.* NESSIE Web Site: https://www.cosic.esat.kuleuven.ac.be/.

[24] S. Kent and R. Atkinson, *IP Authentication Header.* RFC Editor, 1998.

[25] S. Kent and R. Atkinson, *Security Architecture for the Internet Protocol.* RFC Editor, 1998.

[26] R. Younglove, *IPSec: What Makes It Work.* 2000.

[27] C. Madson and R. Glenn, *The Use of HMAC-MD5-96 within ESP and AH.* RFC Editor, 1998.

[28] S. Kent and R. Atkinson, *IP Encapsulating Security Payload (ESP).* RFC Editor, 1998.

[29] J. Burke, J. McDonald, and T. Austin, "Architectural support for fast symmetric-key cryptography," in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems,* pp. 178–189, ACM Press, 2000.

[30] J. P. Huber and M. W. Rosneck, *Successful ASIC Design The First Time Through.* Van Nostrand Reinhold, 1991.

[31] L. Stok and J. Cohn, "There is life left in ASICs," in *Proceedings of the 2003 international symposium on physical design*, pp. 48–50, ACM Press, 2003.

[32] R. A. Rutenbar, M. Baron, T. Daniel, R. Jayaraman, Z. Or-Bach, J. Rose, and C. Sechen, "(when) will FPGAs kill ASICs? (panel session)," in *Proceedings of the 38th conference on Design automation*, pp. 321–322, ACM Press, 2001.

[33] T. K. Tetsuya Ichikawa and M. Matsui, "Hardware evaluation of the AES finalists," in *AES3: The Third Advanced Encryption Standard Candidate Conference*, 2000.

[34] T. R. Bryan Weeks, Mark Bean and C. Ficke, "Hardware performance simulations of round 2 advanced encryption standard algorithms," in *AES3: The Third Advanced Encryption Standard Candidate Conference*, 2000.

[35] T. R. Bryan Weeks, Mark Bean and C. Ficke, "Hardware performance simulations of round 2 advanced encryption standard algorithms (presentation)," in *AES3: The Third Advanced Encryption Standard Candidate Conference*, 2000.

[36] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys (CSUR)*, vol. 34, no. 2, pp. 171–210, 2002.

[37] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: A survey," *Journal of VLSI Signal Processing*, vol. 28, pp. 7–27, 2001.

[38] *Spartan and Spartan-XL Famalies Field Programmable Gate Array Datasheet.* Xilinx Incorporated, 2002.

[39] C. Ajluni, "Field programmable gate arrays just aren't for prototyping anymore.," *Electronic Design*, April 2000.

[40] K. Gaj and P. Chodowiec, "Comparison of the hardware performance of the

AES candidates using reconfigurable hardware," in *AES3: The Third Advanced Encryption Standard Candidate Conference*, 2000.

[41] M. Riaz and H. Heys, "The FPGA implementation of the RC6 and CAST-256 encryption algorithms," in *IEEE Canadian Conference on Electrical and Computer Engineering*, May 1999.

[42] R. W. Hartenstein, T. Hoffmann, and U. Nadeldinger, "Design-space exploration of low power coarse grained reconfigurable datapath array architectures," in *Proceedings of the 10th International Workshop on Integrated Circuit Design, Power and Timing Modeling, Optimization and Simulation*, pp. 118–128, Springer-Verlag, 2000.

[43] Y. Mitsuyama, Z. Andales, T. Onoye, and I. Shirakawa, "A dynamically reconfigurable hardware-based cipher chip," in *Proceedings of the conference on Asia South Pacific Design Automation Conference*, pp. 11–12, ACM Press, 2001.

[44] *Chameleon Systems CS2112 Data Book.* Chameleon Systems Incorporated, 2001.

Appendix A

Sample Verilog Code for Selected Modules

A.1 RC5 Testbench

```
// TESTBENCH FOR RC5 CIPHER KERNEL
// Used with Verilog to verify operation before synthesis.
// Jason Rhinelander
module rc5tb;
reg clk,rst,start;
wire done;
rc5top rc5(clk,rst,start,done);
initial clk <= 1;
always @(clk) clk <= #5 ~clk;
initial begin
#0
  `include "rc5Keys.include"
rst <= 1;
start <=0;
#10
rst <= 0;
#10
start <= 1;
#10
start <= 0;
     #2000;
     $finish;
   end
initial begin
     $shm_open("rc5top.shm");
     $shm_probe("AS", rc5);
   end
   //initial begin
     //$monitor($$time,,,"rst=%b start=%b done=%b",rst,start,done);
   //end
endmodule
```

A.2 Iterative RC5 Top Level Module

```
// TOP LEVEL MODULE REQUIRED BY CS2112 ARCHITECTURE
module rc5top(clk,rst,start,done);
input clk;
input rst;
input start;
output done;

wire [2:0] LSM_ctl,waiter_ctl,iblock1_ctl,shifter_ctl,iblock2_ctl
,addr_ctl,counter_ctl;
wire wait_done_flag,count_done_flag;
```

```
#100
go_run_3 <= 1'b1;
#10
go_run_3 <= 1'b0;

#100
go_run_3 <= 1'b1;
#10
go_run_3 <= 1'b0;

#100
go_run_3 <= 1'b1;
#10
go_run_3 <= 1'b0;

#100
go_run_3 <= 1'b1;
#10
go_run_3 <= 1'b0;

#100
go_run_external_4 <= 1'b1;
#10
go_run_external_4 <= 1'b0;

#100
go_run_4 <= 1'b1;
#10
go_run_4 <= 1'b0;

#100
go_run_4 <= 1'b1;
#10
go_run_4 <= 1'b0;

#100
go_run_4 <= 1'b1;
#10
go_run_4 <= 1'b0;

#100
go_run_4 <= 1'b1;
#10
go_run_4 <= 1'b0;

#100
go_run_4 <= 1'b1;
#10
go_run_4 <= 1'b0;

    #500;
    $finish;
  end
initial begin
    $shm_open("test.shm");
    $shm_probe("AS", rc5_pipeline_tb);
  end
endmodule
```

```
rc5dp dp(
.clk(clk),
.rst(rst),
.iblock1_ctl(iblock1_ctl),
.iblock2_ctl(iblock2_ctl),
.addr_ctl(addr_ctl),
.shifter_ctl(shifter_ctl),
.LSM_ctl(LSM_ctl),
.waiter_ctl(waiter_ctl),
.counter_ctl(counter_ctl),
.wait_done_flag(wait_done_flag),
.count_done_flag(count_done_flag)
);

rc5ctl ctl(
.clk(clk),
.rst(rst),
.start(start),
.iblock1_ctl(iblock1_ctl),
.iblock2_ctl(iblock2_ctl),
.addr_ctl(addr_ctl),
.LSM_ctl(LSM_ctl),
.shifter_ctl(shifter_ctl),
.waiter_ctl(waiter_ctl),
.counter_ctl(counter_ctl),
.wait_done_flag(wait_done_flag),
.count_done_flag(count_done_flag),
.done(done)
);
endmodule
```

A.3 Iterative RC5 Controller Module

```
// RC5 Datapath controler
// Jason Rhinelander
module rc5ctl(clk,rst,start,LSM_ctl, waiter_ctl,iblock1_ctl,iblock2_ctl,addr_ctl,
counter_ctl, shifter_ctl, wait_done_flag, count_done_flag,done);
/*
Just as a refence, just assign to next_state;
parameter
IDLE = 4'b0000,
INIT = 4'b0001,
R1 = 4'b0010,
R2 = 4'b0011,
R3 = 4'b0100,
R4 = 4'b0101,
R5 = 4'b0110,
R6 = 4'b0111,
R7 = 4'b1000,
R8 = 4'b1001,
R8b = 4'b1010,
R9 = 4'b1011,
DONE = 4'b1100;
*/
input clk;
input rst;
input start;
output [2:0] LSM_ctl;
output [2:0] iblock1_ctl;
output [2:0] iblock2_ctl;
output [2:0] shifter_ctl;
output [2:0] addr_ctl;
output [2:0] waiter_ctl;
output [2:0] counter_ctl;
output done;
input wait_done_flag;
input count_done_flag;
```

```
reg [2:0] LSM_ctl;
reg [2:0] waiter_ctl;
reg [2:0] iblock1_ctl;
reg [2:0] iblock2_ctl;
reg [2:0] shifter_ctl;
reg [2:0] addr_ctl;
reg [2:0] counter_ctl;
reg done;
reg [3:0] current_state;
reg [3:0] next_state;
reg [2:0] next_LSM_ctl;
reg [2:0] next_waiter_ctl;
reg [2:0] next_iblock1_ctl;
reg [2:0] next_iblock2_ctl;
reg [2:0] next_shifter_ctl;
reg [2:0] next_addr_ctl;
reg [2:0] next_counter_ctl;
reg next_done;
//DEFINE THE SEQUENTIAL BLOCK FOR THE CONTROLLER
always @(posedge clk)
begin
if (rst==1) begin
current_state <= 4'b0000;
LSM_ctl <= 3'b000;
waiter_ctl <= 3'b000;
iblock1_ctl <= 3'b000;
iblock2_ctl <= 3'b000;
addr_ctl <= 3'b000;
counter_ctl <= 3'b001;
shifter_ctl <= 3'b000;
done <= 1'b0;
end else begin
current_state <= next_state;
LSM_ctl <= next_LSM_ctl;
waiter_ctl <= next_waiter_ctl;
iblock1_ctl<= next_iblock1_ctl;
iblock2_ctl<= next_iblock2_ctl;
counter_ctl <= next_counter_ctl;
addr_ctl <= next_addr_ctl;
shifter_ctl <= next_shifter_ctl;
done <= next_done;
end
end
// Combinatorial block for the controler module
always @(current_state or start or wait_done_flag or count_done_flag) begin
case(current_state)
//ASSIGN TO OUTPUT LINES
4'b0000: begin
if (start) begin
next_state = 4'b0001;
next_LSM_ctl = 3'b010;
next_iblock1_ctl = 3'b010;
next_iblock2_ctl = 3'b010;
next_waiter_ctl = 3'b000;
next_counter_ctl = 3'b001;
next_addr_ctl = 3'b000;
next_shifter_ctl = 3'b000;
next_done = 1'b0;
end else begin
next_state = 4'b0000;
next_LSM_ctl = 3'b000;
next_iblock1_ctl = 3'b000;
next_iblock2_ctl = 3'b000;
next_waiter_ctl = 3'b000;
next_counter_ctl = 3'b001;
next_addr_ctl = 3'b001;
next_shifter_ctl = 3'b000;
next_done = 1'b0;
```

```
end
end
4'b0001: begin
next_state = 4'b0010;
next_LSM_ctl = 3'b010;
next_iblock1_ctl = 3'b010;
next_iblock2_ctl = 3'b010;
next_waiter_ctl = 3'b000;
next_counter_ctl = 3'b001;
next_addr_ctl = 3'b001;
next_shifter_ctl = 3'b000;
next_done = 1'b0;
end
4'b0010: begin
next_state = 4'b0011;
next_LSM_ctl = 3'b010;
next_iblock1_ctl = 3'b010;
next_iblock2_ctl = 3'b010;
next_waiter_ctl = 3'b001;
next_counter_ctl = 3'b001;
next_addr_ctl = 3'b001;
next_shifter_ctl = 3'b000;
next_done = 1'b0;
end
4'b0011: begin
next_state = 4'b0100;
next_LSM_ctl = 3'b010;
next_iblock1_ctl = 3'b010;
next_iblock2_ctl = 3'b010;
next_waiter_ctl = 3'b001;
next_counter_ctl = 3'b000;
next_addr_ctl = 3'b001;
next_shifter_ctl = 3'b000;
next_done = 1'b0;
end
4'b0100: begin
if(wait_done_flag) begin
next_state = 4'b0101;
next_LSM_ctl = 3'b001;
next_iblock1_ctl = 3'b010;
next_iblock2_ctl = 3'b010;
next_waiter_ctl = 3'b001;
next_counter_ctl = 3'b001;
next_addr_ctl = 3'b001;
next_shifter_ctl = 3'b000;
next_done = 1'b0;
end else begin
next_state = 4'b0100;
next_LSM_ctl = 3'b010;
next_iblock1_ctl = 3'b010;
next_iblock2_ctl = 3'b010;
next_waiter_ctl = 3'b001;
next_counter_ctl = 3'b001;
next_addr_ctl = 3'b001;
next_shifter_ctl = 3'b000;
next_done = 1'b0;
end
end
4'b0101: begin
next_state = 4'b0110;
next_LSM_ctl = 3'b010;
next_iblock1_ctl = 3'b010;
next_iblock2_ctl = 3'b010;
next_waiter_ctl = 3'b000;
next_counter_ctl = 3'b001;
next_addr_ctl = 3'b000;
next_shifter_ctl = 3'b000;
next_done = 1'b0;
```

```
end
4'b0110: begin
next_state = 4'b0111;
next_LSM_ctl = 3'b010;
next_iblock1_ctl = 3'b001;
next_iblock2_ctl = 3'b010;
next_waiter_ctl = 3'b001;
next_counter_ctl = 3'b001;
next_addr_ctl = 3'b001;
next_shifter_ctl = 3'b001;
next_done = 1'b0;
end
4'b0111: begin
next_state = 4'b1000;
next_LSM_ctl = 3'b001;
next_iblock1_ctl = 3'b010;
next_iblock2_ctl = 3'b010;
next_waiter_ctl = 3'b001;
next_counter_ctl = 3'b001;
next_addr_ctl = 3'b001;
next_shifter_ctl = 3'b001;
next_done = 1'b0;
end
4'b1000: begin
if(wait_done_flag) begin
next_state = 4'b1001;
next_LSM_ctl = 3'b010;
next_iblock1_ctl = 3'b010;
next_iblock2_ctl = 3'b001;
next_waiter_ctl = 3'b000;
next_counter_ctl = 3'b001;
next_addr_ctl = 3'b000;
next_shifter_ctl = 3'b001;
next_done = 1'b0;
end else begin
next_state = 4'b1000;
next_LSM_ctl = 3'b010;
next_iblock1_ctl = 3'b010;
next_iblock2_ctl = 3'b010;
next_waiter_ctl = 3'b001;
next_counter_ctl = 3'b001;
next_addr_ctl = 3'b001;
next_shifter_ctl = 3'b001;
next_done = 1'b0;
end
end
4'b1001: begin
if(count_done_flag) begin
next_state = 4'b1010;
next_LSM_ctl = 3'b010;
next_iblock1_ctl = 3'b010;
next_iblock2_ctl = 3'b010;
next_waiter_ctl = 3'b000;
next_counter_ctl = 3'b001;
next_addr_ctl = 3'b001;
next_shifter_ctl = 3'b000;
next_done = 1'b1;
end else begin
next_state = 4'b1011;
next_LSM_ctl = 3'b010;
next_iblock1_ctl = 3'b010;
next_iblock2_ctl = 3'b001;
next_waiter_ctl = 3'b000;
next_counter_ctl = 3'b001;
next_addr_ctl = 3'b001;
next_shifter_ctl = 3'b000;
next_done = 1'b0;
end
```

```
end
4'b1010: begin
next_state = 4'b1100;
next_LSM_ctl = 3'b010;
next_iblock1_ctl = 3'b010;
next_iblock2_ctl = 3'b010;
next_waiter_ctl = 3'b000;
next_counter_ctl = 3'b001;
next_addr_ctl = 3'b001;
next_shifter_ctl = 3'b000;
next_done = 1'b1;
end
4'b1011: begin
next_state = 4'b0010;
next_LSM_ctl = 3'b010;
next_iblock1_ctl = 3'b010;
next_iblock2_ctl = 3'b001;
next_waiter_ctl = 3'b000;
next_counter_ctl = 3'b001;
next_addr_ctl = 3'b001;
next_shifter_ctl = 3'b000;
next_done = 1'b0;
end
default: begin
 next_state = 4'b1100;
 next_LSM_ctl = 3'b010;
 next_iblock1_ctl = 3'b010;
 next_iblock2_ctl = 3'b010;
 next_waiter_ctl = 3'b000;
 next_counter_ctl = 3'b001;
 next_addr_ctl = 3'b001;
 next_shifter_ctl = 3'b000;
 next_done = 1'b1;
end
endcase
        end
endmodule
```

A.4 Iterative RC5 Datapath Module

```
// rc5 encryption data path definitions.
// June 5th 2001
//
// Jason Rhinelander
`include "CS2112_Instructions.include"
Template instantiaion
defparam dpu1.A_REG_INITIAL_VALUE = 32'h0;
defparam dpu1.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu1.O_REG_INITIAL_VALUE = 32'h0;

defparam dpu1.INSTRUCTION_0 = ();

CS2112_DPU dpu1(

      INPUTS
      .clk(clk),
      .rst(rst),
      .a_in0(),
      .b_in0(),

      DPU OUTPUT
      .dpu_output(),

      CSM ADDRESS
      .csm_addr(),

      FLAG OUTPUTS
```

```
      .flag_high(),
      .flag_low(),

      LSM CONNECTIONS
      .data_to_lsm(),
      .data_from_lsm(),
      .lsm_addr(),
      .lsm_write_en(),

      CARRY LOGIC
      .carry_in(),
      .carry_out()
);
*/

//Module description for a variable circular shifter.
module varCirShifter(clk,rst,shifter_ctl,lop,rop,rop2,outdata);
input clk;
input rst;
input [31:0] lop;
input [31:0] rop;
input [31:0] rop2;
input [2:0] shifter_ctl;
output[31:0] outdata;
wire [31:0] cir1, cir2, cir4, cir5;

// DATA PATH elements for variable circular shifting.
defparam variable_cir_shift_A.B_REG_INITIAL_VALUE = 32'h0000001f;
defparam variable_cir_shift_A.A_REG_INITIAL_VALUE = 32'h00000020;
defparam variable_cir_shift_A.INSTRUCTION_0 = (`A_ZERO_IN | `OPA_REG |
`B0_IN | `SHIFT_OFF | `OPB_AND_MASK | `ALU_OR |`OUT_ALU | `LOAD_0_REG);

defparam variable_cir_shift_A.INSTRUCTION_1 = (`A_ZERO_IN | `OPA_REG | `B1_IN |
       `SHIFT_OFF | `OPB_AND_MASK | `ALU_OR |
       `OUT_ALU | `LOAD_0_REG);
CS2112_DPU variable_cir_shift_A(
      .clk(clk),
      .rst(rst),
      .a_in0(),
      .b_in0(rop),
      .b_in1(rop2),
      .dpu_output(cir1),
      .csm_addr(shifter_ctl),
      .flag_high(),
      .flag_low(),
      .data_to_lsm(),
      .data_from_lsm(),
      .lsm_addr(),
      .lsm_write_en(),
      .carry_in(),
      .carry_out()
);



defparam variable_cir_shift_B.INSTRUCTION_0 = (`A0_IN | `OPA_NO_REG |
`B0_IN | `OPB_NO_REG | `SHIFT_BBA | `OUT_ALU | `LOAD_0_REG);
CS2112_DPU variable_cir_shift_B(
      .clk(clk),
      .rst(rst),
      .a_in0(cir1),
      .b_in0(lop),
      .dpu_output(cir5),
      .csm_addr(3'b000),
      .flag_high(),
      .flag_low(),
      .data_to_lsm(),
```

```
      .data_from_lsm(),
      .lsm_addr(),
      .lsm_write_en(),
      .carry_in(),
      .carry_out()
);


//NEED TO MODIFY THIS BLOCK SO THAT BIT 6 OF (Y&(W-1)) IS CLEARED
defparam variable_cir_shift_C.A_REG_INITIAL_VALUE = 32'h00000020;
defparam variable_cir_shift_C.B_REG_INITIAL_VALUE = 32'h0000001f;
defparam variable_cir_shift_C.INSTRUCTION_O = (`AO_IN |`OPA_REG | `BO_IN
 | `OPB_AND_MASK | `ALU_SUB | `OUT_ALU | `LOAD_O_REG);
CS2112_DPU variable_cir_shift_C(
      .clk(clk),
      .rst(rst),
      .a_in0(),
      .b_in0(cir1),
      .dpu_output(cir2),
      .csm_addr(3'b000),
      .flag_high(),
      .flag_low(),
      .data_to_lsm(),
      .data_from_lsm(),
      .lsm_addr(),
      .lsm_write_en(),
      .carry_in(),
      .carry_out()
);

defparam variable_cir_shift_D.INSTRUCTION_O = (`AO_IN | `OPA_NO_REG | `BO_IN | `OPB_NO_REG |
       `SHIFT_BBA | `OUT_ALU | `LOAD_O_REG);
CS2112_DPU variable_cir_shift_D(
      .clk(clk),
      .rst(rst),
      .a_in0(cir2),
      .b_in0(lop),
      .dpu_output(cir4),
      .csm_addr(3'b000),
      .flag_high(),
      .flag_low(),
      .data_to_lsm(),
      .data_from_lsm(),
      .lsm_addr(),
      .lsm_write_en(),
      .carry_in(),
      .carry_out()
);


// OR the two values together, finished circular shift here.
defparam variable_cir_shift_E.INSTRUCTION_O = (`AO_IN | `OPA_NO_REG | `BO_IN |
`OPB_NO_REG |`SHIFT_OFF | `ALU_OR | `OUT_ALU | `LOAD_O_REG);
CS2112_DPU variable_cir_shift_E(
      .clk(clk),
      .rst(rst),
      .a_in0(cir4),
      .b_in0(cir5),
      .dpu_output(outdata),
      .csm_addr(3'b000),
      .flag_high(),
      .flag_low(),
      .data_to_lsm(),
      .data_from_lsm(),
      .lsm_addr(),
      .lsm_write_en(),
      .carry_in(),
```

```
      .carry_out()
);

endmodule


// MODULE DESCRIPTION FOR THE RC5 DATA PATH
module rc5dp(clk, rst, iblock1_ctl, iblock2_ctl,
addr_ctl,shifter_ctl,LSM_ctl, waiter_ctl, counter_ctl,
wait_done_flag, count_done_flag);

input clk;
input rst;
input [2:0] LSM_ctl;
input [2:0] iblock1_ctl;
input [2:0] iblock2_ctl;
input [2:0] addr_ctl;
input [2:0] shifter_ctl;
input [2:0]waiter_ctl;
input [2:0]counter_ctl;
output wait_done_flag;
output count_done_flag;

// internal wiring
wire [31:0] w1, w2, w3, w4, w5;
wire [31:0] lsm_read_data, lsm_read_addr, memData;

// ADD configuration...
// CAN USE THE A AND B REGS TO LOAD IN THE VALUES FOR PLAINTEXT (A) AND S[0].
defparam iblock1.A_REG_INITIAL_VALUE = 32'h9bbbd8c8;
defparam iblock1.B_REG_INITIAL_VALUE = 32'h0;

// INITIAL ADD OPERATION TO LOAD VALUES
defparam iblock1.INSTRUCTION_0 = (`A0_IN | `OPA_REG | `B0_IN | `OPB_REG |
       `SHIFT_OFF | `ALU_ADD | `OUT_ALU | `LOAD_0_REG);
// pass A
defparam iblock1.INSTRUCTION_1 = (`A1_IN | `OPA_NO_REG | `LOAD_A_REG |
`B0_IN | `OPB_NO_REG |`SHIFT_OFF | `ALU_PASSA | `OUT_ALU | `LOAD_0_REG);

// HOLD INSTRUCTION
defparam iblock1.INSTRUCTION_2 = (`A_ALU_IN | `OPA_REG | `LOAD_A_REG | `B0_IN  |
       `SHIFT_OFF | `ALU_PASSA | `OUT_ALU | `LOAD_0_REG);
CS2112_DPU iblock1(
      .clk(clk),
      .rst(rst),
      .a_in0(),
      .a_in1(w5),
      .b_in0(),
      .dpu_output(w1),
      .csm_addr(iblock1_ctl),
      .flag_high(),
      .flag_low(),
      .data_to_lsm(),
      .data_from_lsm(),
      .lsm_addr(),
      .lsm_write_en(),
      .carry_in(),
      .carry_out()
);
// ADD configuration...
// CAN USE THE A AND B REGS TO LOAD IN THE VALUES FOR PLAINTEXT (B) AND S[1].
defparam iblock2.A_REG_INITIAL_VALUE = 32'h1a37f7fb;
defparam iblock2.B_REG_INITIAL_VALUE = 32'h0;

// INITIAL ADD OPERATION TO LOAD VALUES
defparam iblock2.INSTRUCTION_0 = (`A0_IN | `OPA_REG | `B0_IN | `OPB_REG |
       `SHIFT_OFF | `ALU_ADD | `OUT_ALU | `LOAD_0_REG);
// pass A
```

```
defparam iblock2.INSTRUCTION_1 = (`A1_IN | `OPA_NO_REG| `LOAD_A_REG |
 `BO_IN | `OPB_NO_REG |`SHIFT_OFF | `ALU_PASSA | `OUT_ALU | `LOAD_O_REG);

// HOLD INSTRUCTION
defparam iblock2.INSTRUCTION_2 = (`A_ALU_IN | `OPA_REG | `LOAD_A_REG
| `BO_IN  |`SHIFT_OFF | `ALU_PASSA | `OUT_ALU | `LOAD_O_REG);
CS2112_DPU iblock2(
      .clk(clk),
      .rst(rst),
      .a_in0(),
      .a_in1(w5),
      .b_in0(),
      .dpu_output(w2),
      .csm_addr(iblock2_ctl),
      .flag_high(),
      .flag_low(),
      .data_to_lsm(),
      .data_from_lsm(),
      .lsm_addr(),
      .lsm_write_en(),
      .carry_in(),
      .carry_out()
);
// ADD configuration...
defparam add1.INSTRUCTION_0 = (`AO_IN | `OPA_NO_REG | `LOAD_A_REG | `BO_IN |
`OPB_NO_REG| `LOAD_B_REG |`SHIFT_OFF | `ALU_ADD | `OUT_ALU | `LOAD_O_REG);

// HOLD instruction
defparam add1.INSTRUCTION_1 = (`AO_IN | `OPA_REG | `OPB_REG | `BO_IN  |
       `SHIFT_OFF | `ALU_ADD | `OUT_ALU | `LOAD_O_REG);
CS2112_DPU add1(
      .clk(clk),
      .rst(rst),
      .a_in0(memData),
      .b_in0(w4),
      .dpu_output(w5),
      .csm_addr(addr_ctl),
      .flag_high(),
      .flag_low(),
      .data_to_lsm(),
      .data_from_lsm(),
      .lsm_addr(),
      .lsm_write_en(),
      .carry_in(),
      .carry_out()
);
// XOR configuration...
defparam xor1.INSTRUCTION_0 = (`AO_IN | `OPA_NO_REG | `BO_IN | `OPB_NO_REG |
       `SHIFT_OFF | `ALU_XOR | `OUT_ALU | `LOAD_O_REG);
CS2112_DPU xor1(
      .clk(clk),
      .rst(rst),
      .a_in0(w1),
      .b_in0(w2),
      .dpu_output(w3),
      .csm_addr(3'b000),
      .flag_high(),
      .flag_low(),
      .data_to_lsm(),
      .data_from_lsm(),
      .lsm_addr(),
      .lsm_write_en(),
      .carry_in(),
      .carry_out()
);
// LSM Address generator
defparam addrGen.A_REG_INITIAL_VALUE = 32'h0;
defparam addrGen.B_REG_INITIAL_VALUE = 32'h0;  //LOAD VALUE OF STARTING ADDRESS
```

```
defparam addrGen.O_REG_INITIAL_VALUE = 32'h0;

// load instruction
defparam addrGen.INSTRUCTION_0 = (`A_ALU_IN | `LOAD_A_REG | `OPA_REG |
`OPB_REG| `ALU_PASSB | `OUT_LSM | `LOAD_O_REG);
// Run instruction
defparam addrGen.INSTRUCTION_1 = (`A_ALU_IN | `LOAD_A_REG | `OPA_REG |
`OPB_NO_REG| `ALU_ADD | `OUT_LSM | `LOAD_O_REG | `KN_4);
// Hold instruction
defparam addrGen.INSTRUCTION_2 = (`A_ALU_IN | `HOLD_A_REG | `OPA_REG |
`OPB_NO_REG | `ALU_PASSA | `OUT_LSM | `KN_4);
CS2112_DPU addrGen(
      .clk(clk),
      .rst(rst),
      .a_in0(),
      .b_in0(),
      .dpu_output(memData),
      .csm_addr(LSM_ctl),
      .flag_high(),
      .flag_low(),
      .data_to_lsm(),
      .data_from_lsm(lsm_read_data),
      .lsm_addr(lsm_read_addr),
      .lsm_write_en(),
      .carry_in(),
      .carry_out()
);
// Need LSM to contain the S[] array.
defparam memArray.OFFSET = 0;
defparam memArray.ADDR_MATCH = 4'h0;
defparam memArray.ADDR_MATCH_ENABLE_MASK = 4'h0;
defparam memArray.WRITE_PORT_WIDTH = `LSM_PORT_SIZE_32;
defparam memArray.READ_PORT_WIDTH = `LSM_PORT_SIZE_32;
CS2112_LSM memArray(
.clk(clk),
.lsm_write_addr(),
.lsm_write_data(),
.lsm_write_en(),
.lsm_read_addr(lsm_read_addr),
.lsm_read_data(lsm_read_data),
.chain_data_in(32'h0)
);
//Counter is used for keeping track of the number of rounds completed.
//
defparam counter.B_REG_INITIAL_VALUE = 32'h0000000c;
// inc instruction
defparam counter.INSTRUCTION_0 = (`A_ALU_IN | `LOAD_A_REG | `OPA_REG |
 `OPB_NO_REG |`ALU_ADD | `OUT_ALU | `LOAD_O_REG | `KN_1);
// Hold instruction
defparam counter.INSTRUCTION_1 = (`A_ALU_IN | `LOAD_A_REG |`OPA_REG |
`OPB_REG | `ALU_PASSA | `OUT_ALU | `LOAD_O_REG | `FLAG_EQ);
// reset instruction
defparam counter.INSTRUCTION_2 = (`A_ZERO_IN | `LOAD_A_REG | `OPA_REG |
`OPB_NO_REG | `ALU_PASSA | `OUT_ALU | `LOAD_O_REG);
CS2112_DPU counter(
      .clk(clk),
      .rst(rst),
      .a_in0(),
      .b_in0(),
      .dpu_output(),
      .csm_addr(counter_ctl),
      .flag_high(count_done_flag),
      .flag_low(),
      .data_to_lsm(),
      .data_from_lsm(),
      .lsm_addr(),
      .lsm_write_en(),
      .carry_in(),
```

```
      .carry_out()
);
// Need a waiting block for the controller to wait for the circular
//shift to be completed
defparam waiter.A_REG_INITIAL_VALUE = 32'h00000002;
defparam waiter.B_REG_INITIAL_VALUE = 32'h00000000;
// Reset and hold instruction
defparam waiter.INSTRUCTION_0 = (`B_ZERO_IN |`HOLD_A_REG |`LOAD_B_REG
|`ALU_PASSB |`OUT_ALU);
// run instruction
defparam waiter.INSTRUCTION_1 = (`B_ALU_IN | `HOLD_A_REG | `LOAD_B_REG
| `OPA_REG | `OPB_REG |`ALU_INC | `OUT_ALU | `FLAG_EQ);
CS2112_DPU waiter(
      .clk(clk),
      .rst(rst),
      .a_in0(),
      .b_in0(),
      .dpu_output(),
      .csm_addr(waiter_ctl),
      .flag_high(wait_done_flag),
      .flag_low(),
      .data_to_lsm(),
      .data_from_lsm(),
      .lsm_addr(),
      .lsm_write_en(),
      .carry_in(),
      .carry_out()
);
varCirShifter shifter(clk,rst,shifter_ctl,w3,w2,w5,w4);
endmodule
```

A.5 Unsigned Integer Multiplier Module Controller

```
// Multiplier datapath controller
`define IDLE 3'd0
`define MUL1 3'd1
`define MUL2 3'd2
`define MUL3 3'd3
`define MUL4 3'd4
`define MUL5 3'd5
`define MUL6 3'd6

module us_multiplier_ctl(
clk,
rst,
sign_flag,
output_ctl);

input clk;
input rst;
input sign_flag;
output [2:0] output_ctl;
reg [2:0] output_ctl;
reg [2:0] current_state;
reg [2:0] next_state;
reg [2:0] sign_flag_delay1;
reg [2:0] sign_flag_delay2;
reg [2:0] sign_flag_delay3;


//DEFINE THE SEQUENTIAL BLOCK FOR THE CONTROLLER
always @(posedge clk)
begin
if (rst==1) begin
```

```
current_state <= `IDLE;
sign_flag_delay1 = 3'b000;
sign_flag_delay2 = 3'b000;
sign_flag_delay3 = 3'b000;
output_ctl = 3'b000;

end else begin
current_state <= next_state;
output_ctl = sign_flag_delay3;
sign_flag_delay3 = sign_flag_delay2;
sign_flag_delay2 = sign_flag_delay1;
end
end

// Combinatorial block for the controler module
always @(current_state) begin
case(current_state)

//ASSIGN TO OUTPUT LINES
`IDLE: begin
next_state = `MUL1;
if(sign_flag==1'b1) begin
sign_flag_delay1 = 3'b001;
end else begin
sign_flag_delay1 = 3'b000;
end
end

`MUL1: begin
next_state = `MUL2;
if(sign_flag==1'b1) begin
sign_flag_delay1 = 3'b001;
end else begin
sign_flag_delay1 = 3'b000;
end

end

`MUL2: begin
next_state = `MUL3;
if(sign_flag==1'b1) begin
sign_flag_delay1 = 3'b001;
end else begin
sign_flag_delay1 = 3'b000;
end
end

`MUL3: begin
next_state = `MUL4;
if(sign_flag==1'b1) begin
sign_flag_delay1 = 3'b001;
end else begin
sign_flag_delay1 = 3'b000;
end
end

`MUL4: begin
next_state = `MUL5;
if(sign_flag==1'b1) begin
sign_flag_delay1 = 3'b001;
end else begin
sign_flag_delay1 = 3'b000;
end
end


`MUL5: begin
next_state = `IDLE;
```

```
if(sign_flag==1'b1) begin
sign_flag_delay1 = 3'b001;
end else begin
sign_flag_delay1 = 3'b000;
end
end

endcase
        end

endmodule
```

A.6 Unsigned Integer Multiplier Module Datapath

```
// Unsigned multiplier module for rc6. Note this multiplier only works
for the operation of
// x*(2x+1)
`include "CS2112_Instructions.include"
module us_multiplier_dp(clk, rst,op_in,sign_flag, res_2_ctl,multiplier_output);
input clk;
input rst;
input [31:0] op_in;
input [2:0] res_2_ctl;
output sign_flag;
output [31:0] multiplier_output;
wire [31:0] w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,delay;

defparam sign_detect1.B_REG_INITIAL_VALUE = 32'h00008000;
defparam sign_detect1.A_REG_INITIAL_VALUE = 32'h00008000;
defparam sign_detect1.INSTRUCTION_0=(`AO_IN|`OPA_AND_MASK|`BO_IN|`OPB_REG|
`HOLD_A_REG|`HOLD_B_REG|`SHIFT_OFF|`ALU_PASSA|`OUT_ALU|`HOLD_0_REG|`FLAG_EQ);
CS2112_DPU sign_detect1(
        .clk(clk),
        .rst(rst),
        .a_in0(op_in),
        .b_in0(),
        .dpu_output(),
        .csm_addr(3'b000),
        .flag_high(sign_flag),
        .flag_low(),
        .data_to_lsm(),
        .data_from_lsm(),
        .lsm_addr(),
        .lsm_write_en(),
        .carry_in(),
        .carry_out()
);

defparam sign_detect2.A_REG_INITIAL_VALUE = 32'h7fff7fff;
defparam sign_detect2.INSTRUCTION_0=(`AO_IN|`OPA_AND_MASK|`BO_IN|
`OPB_REG|`HOLD_A_REG|`HOLD_B_REG|`SHIFT_OFF|`ALU_PASSA|`OUT_ALU|`LOAD_0_REG);
CS2112_DPU sign_detect2(
        .clk(clk),
        .rst(rst),
        .a_in0(op_in),
        .b_in0(),
        .dpu_output(w1),
        .csm_addr(3'b000),
        .flag_high(),
        .flag_low(),
        .data_to_lsm(),
        .data_from_lsm(),
        .lsm_addr(),
        .lsm_write_en(),
```

```
        .carry_in(),
        .carry_out()
);

defparam low_low_mul.INSTRUCTION_0 = (`MUL_A0_IN | `MUL_LOAD_A_REG |
`MUL_B0_IN|`MUL_LOAD_B_REG|`MUL_A_LOW_16|`MUL_B_LOW_16 |`MUL_OUT|`MUL_LOAD_0_REG);
defparam low_low_mul.A_REG_INITIAL_VALUE = 32'h00000000;
defparam low_low_mul.B_REG_INITIAL_VALUE = 32'h00000000;

CS2112_MUL low_low_mul(
        .clk(clk),
        .rst(rst),
        .a_in0(w1),
        .b_in0(w1),
        .mult_output(w3),
        .csm_addr(3'b000)
);

defparam high_low_mul.INSTRUCTION_0 = (`MUL_A0_IN |`MUL_LOAD_A_REG |
`MUL_B0_IN|`MUL_LOAD_B_REG |`MUL_A_HI_16 |`MUL_B_LOW_16 |`MUL_OUT |`MUL_LOAD_0_REG);
defparam high_low_mul.A_REG_INITIAL_VALUE = 32'h00000000;
defparam high_low_mul.B_REG_INITIAL_VALUE = 32'h00000000;
CS2112_MUL high_low_mul(
        .clk(clk),
        .rst(rst),
        .a_in0(w1),
        .b_in0(w1),
        .mult_output(w2),
        .csm_addr(3'b000)
);

// DPU NAME: delay_1: 1 clk delay
defparam delay_1.INSTRUCTION_0=(`A0_IN|`OPA_NO_REG|`B0_IN|`OPB_REG|
`HOLD_A_REG|`LOAD_B_REG|`SHIFT_OFF|`ALU_PASSB|`OUT_ALU|`LOAD_0_REG);
CS2112_DPU delay_1(
        .clk(clk),
        .rst(rst),
        .a_in0(),
        .b_in0(w2),
        .dpu_output(w4),
        .csm_addr(3'b000),
        .flag_high(),
        .flag_low(),
        .data_to_lsm(),
        .data_from_lsm(),
        .lsm_addr(),
        .lsm_write_en(),
        .carry_in(),
        .carry_out()
);

//=================================================================
// DPU NAME: delay_1: 2 clk delay
defparam add3delay.INSTRUCTION_0=(`A0_IN|`OPA_REG|`B0_IN|`OPB_REG|
`HOLD_A_REG|`LOAD_B_REG|`SHIFT_OFF|`ALU_PASSB|`OUT_ALU|`LOAD_0_REG);
CS2112_DPU add3delay(
        .clk(clk),
        .rst(rst),
        .a_in0(),
        .b_in0(w7),
        .dpu_output(delay),
        .csm_addr(3'b000),
        .flag_high(),
        .flag_low(),
        .data_to_lsm(),
        .data_from_lsm(),
        .lsm_addr(),
        .lsm_write_en(),
```

```
        .carry_in(),
        .carry_out()
);

// 2 clk delay
defparam delay_3.INSTRUCTION_0=('A0_IN|'OPA_REG|'B0_IN|'OPB_REG|'LOAD_A_REG|'LOAD_B_REG|
'SHIFT_OFF|'ALU_PASSB|'OUT_ALU|'LOAD_0_REG);
CS2112_DPU delay_3(
        .clk(clk),
        .rst(rst),
        .a_in0(),
        .b_in0(op_in),
        .dpu_output(w9),
        .csm_addr(3'b000),
        .flag_high(),
        .flag_low(),
        .data_to_lsm(),
        .data_from_lsm(),
        .lsm_addr(),
        .lsm_write_en(),
        .carry_in(),
        .carry_out()
);

// 2 clk delay
defparam delay_2.INSTRUCTION_0=('A0_IN|'OPA_REG|'B0_IN|'OPB_REG|'LOAD_A_REG|'LOAD_B_REG|
'SHIFT_OFF|'ALU_PASSB|'OUT_ALU|'LOAD_0_REG);
CS2112_DPU delay_2(
        .clk(clk),
        .rst(rst),
        .a_in0(),
        .b_in0(w9),
        .dpu_output(w8),
        .csm_addr(3'b000),
        .flag_high(),
        .flag_low(),
        .data_to_lsm(),
        .data_from_lsm(),
        .lsm_addr(),
        .lsm_write_en(),
        .carry_in(),
        .carry_out()
);

// 1 clk delay
defparam delay_4.INSTRUCTION_0=('A0_IN|'OPA_NO_REG|'B0_IN|'OPB_NO_REG|'LOAD_A_REG|
'LOAD_B_REG|'SHIFT_OFF|'ALU_PASSB|'OUT_ALU|'LOAD_0_REG);
CS2112_DPU delay_4(
        .clk(clk),
        .rst(rst),
        .a_in0(),
        .b_in0(w11),
        .dpu_output(w10),
        .csm_addr(3'b000),
        .flag_high(),
        .flag_low(),
        .data_to_lsm(),
        .data_from_lsm(),
        .lsm_addr(),
        .lsm_write_en(),
        .carry_in(),
        .carry_out()
);

defparam add_1.INSTRUCTION_0=('A0_IN|'OPA_REG|'B0_IN|'OPB_REG|'LOAD_A_REG|'LOAD_B_REG|
'LSL|'SHFT_AMT_16|'ALU_ADD|'OUT_ALU|'LOAD_0_REG);
CS2112_DPU add_1(
        .clk(clk),
```

```
        .rst(rst),
        .a_in0(w3),
        .b_in0(w2),
        .dpu_output(w5),
        .csm_addr(3'b000),
        .flag_high(),
        .flag_low(),
        .data_to_lsm(),
        .data_from_lsm(),
        .lsm_addr(),
        .lsm_write_en(),
        .carry_in(),
        .carry_out()
);

defparam add_2.INSTRUCTION_0=('A0_IN|'OPA_REG|'B0_IN|'OPB_REG|'LOAD_A_REG|
'LOAD_B_REG|'LSL|'SHFT_AMT_16|'ALU_ADD|'OUT_ALU|'LOAD_0_REG);
CS2112_DPU add_2(
        .clk(clk),
        .rst(rst),
        .a_in0(w5),
        .b_in0(w4),
        .dpu_output(w6),
        .csm_addr(3'b000),
        .flag_high(),
        .flag_low(),
        .data_to_lsm(),
        .data_from_lsm(),
        .lsm_addr(),
        .lsm_write_en(),
        .carry_in(),
        .carry_out()
);

defparam add_3.INSTRUCTION_0=('A0_IN|'OPA_REG|'B0_IN|'OPB_REG|'LOAD_A_REG|
'LOAD_B_REG|'LSL|'SHFT_AMT_1|'ALU_ADD|'OUT_ALU|'LOAD_0_REG);
CS2112_DPU add_3(
        .clk(clk),
        .rst(rst),
.a_in0(delay),
        .b_in0(w6),
        .dpu_output(w12),
        .csm_addr(3'b000),
        .flag_high(),
        .flag_low(),
        .data_to_lsm(),
        .data_from_lsm(),
        .lsm_addr(),
        .lsm_write_en(),
        .carry_in(),
        .carry_out()
);

defparam res_1.A_REG_INITIAL_VALUE = 32'h80000000;
defparam res_1.INSTRUCTION_0=('A0_IN|'OPA_REG|'B0_IN|'OPB_REG|'HOLD_A_REG|
'LOAD_B_REG|'LSL|'SHFT_AMT_17|'ALU_ADD|'OUT_ALU|'LOAD_0_REG);
CS2112_DPU res_1(
        .clk(clk),
        .rst(rst),
        .a_in0(),
        .b_in0(w1),
        .dpu_output(w11),
        .csm_addr(3'b000),
        .flag_high(),
        .flag_low(),
        .data_to_lsm(),
        .data_from_lsm(),
        .lsm_addr(),
```

```
        .lsm_write_en(),
        .carry_in(),
        .carry_out()
);

defparam res_2.INSTRUCTION_0=('A0_IN|'OPA_NO_REG|'B0_IN|'OPB_NO_REG|
'LOAD_A_REG|'HOLD_B_REG|'SHIFT_OFF|'ALU_PASSA|'OUT_ALU|'LOAD_0_REG);
defparam res_2.INSTRUCTION_1=('A0_IN|'OPA_NO_REG|'B0_IN|'OPB_NO_REG|
'LOAD_A_REG|'HOLD_B_REG|'SHIFT_OFF|'ALU_ADD|'OUT_ALU|'LOAD_0_REG);
CS2112_DPU res_2(
        .clk(clk),
        .rst(rst),
        .b_in0(w10),
.a_in0(w8),
        .dpu_output(w7),
        .csm_addr(res_2_ctl),
        .flag_high(),
        .flag_low(),
        .data_to_lsm(),
        .data_from_lsm(),
        .lsm_addr(),
        .lsm_write_en(),
        .carry_in(),
        .carry_out()
);


defparam fixed1.INSTRUCTION_0=('A0_IN|'OPA_NO_REG|'B0_IN|'OPB_NO_REG|
'HOLD_A_REG|'HOLD_B_REG|'LSL|'SHFT_AMT_5|'ALU_PASSB|'OUT_ALU|'LOAD_0_REG);
CS2112_DPU fixed1(
        .clk(clk),
        .rst(rst),
        .a_in0(),
        .b_in0(w12),
        .dpu_output(w13),
        .csm_addr(3'b000),
        .flag_high(),
        .flag_low(),
        .data_to_lsm(),
        .data_from_lsm(),
        .lsm_addr(),
        .lsm_write_en(),
        .carry_in(),
        .carry_out()
);

defparam fixed2.INSTRUCTION_0=('A0_IN|'OPA_NO_REG|'B0_IN|'OPB_NO_REG|
'HOLD_A_REG|'HOLD_B_REG|'LSR|'SHFT_AMT_27|'ALU_PASSB|'OUT_ALU|'LOAD_0_REG);
CS2112_DPU fixed2(
        .clk(clk),
        .rst(rst),
        .a_in0(),
        .b_in0(w12),
        .dpu_output(w14),
        .csm_addr(3'b000),
        .flag_high(),
        .flag_low(),
        .data_to_lsm(),
        .data_from_lsm(),
        .lsm_addr(),
        .lsm_write_en(),
        .carry_in(),
        .carry_out()
);

defparam fixed3.INSTRUCTION_0=('A0_IN|'OPA_NO_REG|'B0_IN|'OPB_NO_REG|'HOLD_A_REG|
'HOLD_B_REG|'SHIFT_OFF|'ALU_OR|'OUT_ALU|'LOAD_0_REG);
CS2112_DPU fixed3(
```

```
        .clk(clk),
        .rst(rst),
        .a_in0(w14),
        .b_in0(w13),
        .dpu_output(multiplier_output),
        .csm_addr(3'b000),
        .flag_high(),
        .flag_low(),
        .data_to_lsm(),
        .data_from_lsm(),
        .lsm_addr(),
        .lsm_write_en(),
        .carry_in(),
        .carry_out()
);
endmodule
```

A.7 Verilog Testbench For Controlling RC5 Iterative Pipeline

```
module rc5_pipeline_tb;

reg clk;
reg rst;

wire [31:0] stage1_to_2a;
wire [31:0] stage1_to_2b;
wire [31:0] stage1_to_2c;
wire [31:0] stage1_to_2d;

wire [31:0] stage2_to_3a;
wire [31:0] stage2_to_3b;
wire [31:0] stage2_to_3c;
wire [31:0] stage2_to_3d;

wire [31:0] stage3_to_4a;
wire [31:0] stage3_to_4b;
wire [31:0] stage3_to_4c;
wire [31:0] stage3_to_4d;

reg [31:0] external_input1;
reg [31:0] external_input2;
reg [31:0] external_input3;
reg [31:0] external_input4;

wire [31:0] external_output1;
wire [31:0] external_output2;
wire [31:0] external_output3;
wire [31:0] external_output4;

reg  go_init_1,
go_run_1,
go_run_external_1;

reg  go_init_2,
go_run_2,
go_run_external_2;

reg  go_init_3,
go_run_3,
go_run_external_3;

reg  go_init_4,
go_run_4,
go_run_external_4;
```

```
wire [2:0]  LSM_ctl_1;
wire [2:0] iblock1_ctl_1;
wire [2:0] iblock2_ctl_1;
wire [2:0] addr_ctl_1;
wire [2:0]  shifter_ctl2_1;
wire  half_done_1;

wire [2:0]  LSM_ctl_2;
wire [2:0] iblock1_ctl_2;
wire [2:0] iblock2_ctl_2;
wire [2:0] addr_ctl_2;
wire [2:0]  shifter_ctl2_2;
wire  half_done_2;

wire [2:0]  LSM_ctl_3;
wire [2:0] iblock1_ctl_3;
wire [2:0] iblock2_ctl_3;
wire [2:0] addr_ctl_3;
wire [2:0]  shifter_ctl2_3;
wire  half_done_3;

wire [2:0]  LSM_ctl_4;
wire [2:0] iblock1_ctl_4;
wire [2:0] iblock2_ctl_4;
wire [2:0] addr_ctl_4;
wire [2:0]  shifter_ctl2_4;
wire  half_done_4;


rc5_half_round_ctl ctl1(
.clk(clk),
.rst(rst),
.go_init(go_init_1),
.go_run(go_run_1),
.go_run_external(go_run_external_1),
.LSM_ctl(LSM_ctl_1),
.iblock1_ctl(iblock1_ctl_1),
.iblock2_ctl(iblock2_ctl_1),
.addr_ctl(addr_ctl_1),
.shifter_ctl2(shifter_ctl2_1),
.half_done(half_done_1)
);

rc5_half_round_ctl ctl2(
.clk(clk),
.rst(rst),
.go_init(go_init_2),
.go_run(go_run_2),
.go_run_external(go_run_external_2),
.LSM_ctl(LSM_ctl_2),
.iblock1_ctl(iblock1_ctl_2),
.iblock2_ctl(iblock2_ctl_2),
.addr_ctl(addr_ctl_2),
.shifter_ctl2(shifter_ctl2_2),
.half_done(half_done_2)
);

rc5_half_round_ctl ctl3(
.clk(clk),
.rst(rst),
.go_init(go_init_3),
.go_run(go_run_3),
.go_run_external(go_run_external_3),
.LSM_ctl(LSM_ctl_3),
.iblock1_ctl(iblock1_ctl_3),
.iblock2_ctl(iblock2_ctl_3),
.addr_ctl(addr_ctl_3),
```

```
.shifter_ctl2(shifter_ctl2_3),
.half_done(half_done_3)
);

rc5_half_round_ctl ctl4(
.clk(clk),
.rst(rst),
.go_init(go_init_4),
.go_run(go_run_4),
.go_run_external(go_run_external_4),
.LSM_ctl(LSM_ctl_4),
.iblock1_ctl(iblock1_ctl_4),
.iblock2_ctl(iblock2_ctl_4),
.addr_ctl(addr_ctl_4),
.shifter_ctl2(shifter_ctl2_4),
.half_done(half_done_4)
);

rc5_slice1_dp stage1(
.clk(clk),
.rst(rst),
.even_iblock1_ctl(iblock1_ctl_1),
.even_iblock2_ctl(iblock2_ctl_1),
.even_addr_ctl(addr_ctl_1),
.even_shifter_ctl2(shifter_ctl2_1),
.even_LSM_ctl(LSM_ctl_1),
.odd_iblock1_ctl(iblock1_ctl_1),
.odd_iblock2_ctl(iblock2_ctl_1),
.odd_addr_ctl(addr_ctl_1),
.odd_shifter_ctl2(shifter_ctl2_1),
.odd_LSM_ctl(LSM_ctl_1),
.input1(external_input1),
.input2(external_input2),
.input3(external_input3),
.input4(external_input4),
.out1(stage1_to_2a),
.out2(stage1_to_2b),
.out3(stage1_to_2c),
.out4(stage1_to_2d)
);

rc5_slice2_dp stage2(
.clk(clk),
.rst(rst),
.even_iblock1_ctl(iblock1_ctl_2),
.even_iblock2_ctl(iblock2_ctl_2),
.even_addr_ctl(addr_ctl_2),
.even_shifter_ctl2(shifter_ctl2_2),
.even_LSM_ctl(LSM_ctl_2),
.odd_iblock1_ctl(iblock1_ctl_2),
.odd_iblock2_ctl(iblock2_ctl_2),
.odd_addr_ctl(addr_ctl_2),
.odd_shifter_ctl2(shifter_ctl2_2),
.odd_LSM_ctl(LSM_ctl_2),
.input1(stage1_to_2a),
.input2(stage1_to_2b),
.input3(stage1_to_2c),
.input4(stage1_to_2d),
.out1(stage2_to_3a),
.out2(stage2_to_3b),
.out3(stage2_to_3c),
.out4(stage2_to_3d)
);

rc5_slice3_dp stage3(
.clk(clk),
.rst(rst),
.even_iblock1_ctl(iblock1_ctl_3),
```

```
.even_iblock2_ctl(iblock2_ctl_3),
.even_addr_ctl(addr_ctl_3),
.even_shifter_ctl2(shifter_ctl2_3),
.even_LSM_ctl(LSM_ctl_3),
.odd_iblock1_ctl(iblock1_ctl_3),
.odd_iblock2_ctl(iblock2_ctl_3),
.odd_addr_ctl(addr_ctl_3),
.odd_shifter_ctl2(shifter_ctl2_3),
.odd_LSM_ctl(LSM_ctl_3),
.input1(stage2_to_3a),
.input2(stage2_to_3b),
.input3(stage2_to_3c),
.input4(stage2_to_3d),
.out1(stage3_to_4a),
.out2(stage3_to_4b),
.out3(stage3_to_4c),
.out4(stage3_to_4d)
);

rc5_slice4_dp stage4(
.clk(clk),
.rst(rst),
.even_iblock1_ctl(iblock1_ctl_4),
.even_iblock2_ctl(iblock2_ctl_4),
.even_addr_ctl(addr_ctl_4),
.even_shifter_ctl2(shifter_ctl2_4),
.even_LSM_ctl(LSM_ctl_4),
.odd_iblock1_ctl(iblock1_ctl_4),
.odd_iblock2_ctl(iblock2_ctl_4),
.odd_addr_ctl(addr_ctl_4),
.odd_shifter_ctl2(shifter_ctl2_4),
.odd_LSM_ctl(LSM_ctl_4),
.input1(stage3_to_4a),
.input2(stage3_to_4b),
.input3(stage3_to_4c),
.input4(stage3_to_4d),
.out1(external_output1),
.out2(external_output2),
.out3(external_output3),
.out4(external_output4)
);

initial clk <= 1;

always @(clk) clk <= #5 ~clk;

initial begin
#0
`include "rc5KeysStage1a.include"
`include "rc5KeysStage1b.include"
`include "rc5KeysStage2a.include"
`include "rc5KeysStage2b.include"
`include "rc5KeysStage3a.include"
`include "rc5KeysStage3b.include"
`include "rc5KeysStage4a.include"
`include "rc5KeysStage4b.include"

rst <= 1'b0;
#10
rst <= 1'b1;
#20
rst <= 1'b0;
#10
external_input1 <= 32'h9bbbd8c8;
external_input3 <= 32'h9bbbd8c8;
#10
go_run_external_1 <= 1'b1;
external_input2 <= 32'h1a37f7fb;
```

```
external_input4 <= 32'h1a37f7fb;
#10
go_run_external_1 <= 1'b0;

#100
go_run_1 <= 1'b1;
#10
go_run_1 <= 1'b0;

#100
go_run_1 <= 1'b1;
#10
go_run_1 <= 1'b0;

#100
go_run_1 <= 1'b1;
#10
go_run_1 <= 1'b0;

#100
go_run_1 <= 1'b1;
#10
go_run_1 <= 1'b0;

#100
go_run_1 <= 1'b1;
#10
go_run_1 <= 1'b0;

#100
go_run_external_2 <= 1'b1;
#10
go_run_external_2 <= 1'b0;

#100
go_run_2 <= 1'b1;
#10
go_run_2 <= 1'b0;

#100
go_run_2 <= 1'b1;
#10
go_run_2 <= 1'b0;

#100
go_run_2 <= 1'b1;
#10
go_run_2 <= 1'b0;

#100
go_run_2 <= 1'b1;
#10
go_run_2 <= 1'b0;

#100
go_run_2 <= 1'b1;
#10
go_run_2 <= 1'b0;

#100
go_run_external_3 <= 1'b1;
#10
go_run_external_3 <= 1'b0;

#100
go_run_3 <= 1'b1;
#10
go_run_3 <= 1'b0;
```

Appendix B

ANSI C Code for Select Implementations

B.1 RC5 C Code For Testing

```
/*
ANSI C Implmentation of RC5-w/r/b encryption cipher.
Taken from "The RC5 Encryption Algorithim", Ronald L. Rivest
MIT Laboratory for Computer Science.

Modified May 25th, 2001
Jason Rhinelander

Modified July 9th, 2001
Jason Rhinelander
C code will make function call and verify correct output.
*/

#include <stdio.h>

typedef unsigned long int WORD;     // 4 bytes in WORD

#define w 32 // word size in bits
#define r 12 // number of rounds
#define b 16 // number of bytes in key
#define c 4 // number of words in key
// c = max(1,ciel(8*b/w)
#define t 26 // size of table S = 2*(r+1)


WORD S[128] __attribute__((aligned (16)));
WORD S2[128] __attribute__((aligned (16)));
WORD pt[2] __attribute__((aligned (16)));
WORD ct1[2] __attribute__((aligned (16)));
int ct2[2] __attribute__((aligned (16)));
unsigned char key[b] __attribute__((aligned (16)));
WORD P=0xb7e15163;
WORD Q=0x9e3779b9;  // Magic constants for generation of
// the subkeys.
```

```
#pragma CMLN_FUNC_DEF rc5top(int in dp.iblock1.dpu.a,
int in dp.iblock2.dpu.a,int in dp.memArray.lsm[128],
int out *dp.iblock1.dpu.o ,int out *dp.add1.dpu.o)

// Need to define rotation operators. Note x must be unsigned to get
// logical right shift.
#define ROTL(x,y) (((x)<<(y&(w-1))) | ((x)>>(w-(y&(w-1)))))
#define ROTR(x,y) (((x)>>(y&(w-1))) | ((x)<<(w-(y&(w-1)))))
// The encryption function.
void rc5_encrypt(WORD *pt, WORD *ct) // NB: 2 words in pt and ct
{
WORD C;
WORD D;
WORD i, A=pt[0]+S[0],B=pt[1]+S[1];

for (i=1;i<=r;i++){
A=(ROTL(A^B,B))+S[2*i];
B=(ROTL(B^A,A))+S[2*i+1];
}

ct[0] = A;
ct[1] = B;
}

// The Decryption function.
void rc5_decrypt(WORD *ct, WORD *pt)
{
WORD i,A=ct[0],B=ct[1];

for(i=r;i>0;i--){
B=ROTR(B-S[2*i+1],A)^A;
A=ROTR(A-S[2*i],B)^B;
}

pt[1] = B-S[1];
pt[0] = A-S[0];
}

// Setup function for the S array.
void rc5_setup(unsigned char *Key)
{
WORD i;
WORD j;
WORD k;
WORD u=w/8;
WORD A;
WORD B;
WORD L[c];

// Init L and then S then mix key into S.
for(i=b-1,L[c-1]=0; i!=-1; i--){ L[i/u] = (L[i/u]<<8)+Key[i]; }
for(S[0]=P,i=1; i<t; i++){ S[i] = S[i-1]+Q; }
for(A=B=i=j=k=0;k<3*t;k++,i=(i+1)%t,j=(j+1)%c)
```

```
{
A = S[i] = ROTL(S[i]+A+B,3);
B = L[j] = ROTL(L[j]+A+B,(A+B));
}

for(i=0;i<t-2;i++){S2[i]=S[i+2];}
}


/* Any other code following this is for testing purposes
(ex generation the S[] array)  */

int main(){

int i;
pt[1]=0x21A5DBEE;
pt[0]=0x154B8F6D;


//for(i = 0; i<b; i++){key[i] = 0x00;}

key[15] = 0x91;
key[14] = 0x5F;
key[13] = 0x46;
key[12] = 0x19;
key[11] = 0xBE;
key[10] = 0x41;
key[9] = 0xB2;
key[8] = 0x51;
key[7] = 0x63;
key[6] = 0x55;
key[5] = 0xA5;
key[4] = 0x01;
key[3] = 0x10;
key[2] = 0xA9;
key[1] = 0xCE;
key[0] = 0x91;

rc5_setup(key);

/* Put the S[] into the lsm for the hardware call */

rc5_encrypt(pt,ct1);
#pragma CMLN_FUNC_CALL rc5top() SLICES=(0:1)
rc5top(S[0],S[1],S2,&ct2[0],&ct2[1]);

if((ct1[0]==ct2[0]) && (ct1[1]==ct2[1])){asm volatile ("mov r8, 0x10");}
else{asm volatile ("mov r8, 0x20");}
}
```

B.2 RC6 C Code For Testing

```
//#include "RC6.cmln.h"
```

```
/*
ANSI C Implmentation of RC6-w/r/b encryption cipher.
KERNAL MODIFIED CODE for testing
10/10/02

This file is a full testable implementation of RC6
The key must be the same for all blocks of plaintext.

In the Main() an easy way to change the plaintext values
that are going into the cipher are to change the
arithmetic parameters that are modifying the
seed values.
*/

#include <stdio.h>

typedef unsigned long int WORD;     // 4 bytes in WORD

#define w 32 // word size in bits
#define r 20 // number of rounds
#define b 16 // number of bytes in key
#define c 4 // number of words in a byte
// c = max(1,ciel(8*b/w)
#define t 44 // size of table S = 2*(r+1)


WORD S[128] __attribute__((aligned (16))); // global visibility
WORD evenS[128] __attribute__((aligned (16)));
WORD oddS[128] __attribute__((aligned (16)));
WORD A[128] __attribute__((aligned (16)));
WORD B[128] __attribute__((aligned (16)));
WORD C[128] __attribute__((aligned (16)));
WORD D[128] __attribute__((aligned (16)));

WORD ctA[128] __attribute__((aligned (16)));
WORD ctB[128] __attribute__((aligned (16)));
WORD ctC[128] __attribute__((aligned (16)));
WORD ctD[128] __attribute__((aligned (16)));

WORD ctAfab[128] __attribute__((aligned (16)));
WORD ctBfab[128] __attribute__((aligned (16)));
WORD ctCfab[128] __attribute__((aligned (16)));
WORD ctDfab[128] __attribute__((aligned (16)));

// Magic constants for generation of
// the S[]
WORD P=0xb7e15163, Q = 0x9e3779b9;

#pragma CMLN_FUNC_DEF rc6top(in int dp.rcounter.dpu.a,
in int dp.ArdMem.rdMemLSM.lsm[128],
in int dp.BrdMem.rdMemLSM.lsm[128],in int dp.CrdMem.rdMemLSM.lsm[128],
in int dp.DrdMem.rdMemLSM.lsm[128],in int dp.init1.dpu.a,
in int dp.init2.dpu.a,in int dp.final1.dpu.a,
in int dp.final2.dpu.a,in int dp.OddSubkeyMem.lsm[128],
```

```
in int dp.evenSubkeyMem.lsm[128],
out int dp.AwrMem.wrMem.lsm[128],out int dp.BwrMem.wrMem.lsm[128],
out int dp.CwrMem.wrMem.lsm[128],out int dp.DwrMem.wrMem.lsm[128])

// Need to define rotation operators. Note x must be unsigned to get
// logical right shift.
#define ROTL(x,y) (((x)<<(y&(w-1))) | ((x)>>(w-(y&(w-1)))))
#define ROTR(x,y) (((x)>>(y&(w-1))) | ((x)<<(w-(y&(w-1)))))


// The encryption function.
void rc6_encrypt(WORD *pt, WORD *ct) // NB: 2 words in pt and ct
{

WORD i, B=pt[1]+S[0],D=pt[3]+S[1],A=pt[0],C=pt[2];
WORD temp1,temp2,temp3;

for (i=1;i<=r;i++){
temp1 = ROTL((B*(2*B+1)),5);
temp2 = ROTL((D*(2*D+1)),5);
A=(ROTL(A^temp1,temp2))+S[2*i];
C=(ROTL(C^temp2,temp1))+S[2*i+1];
temp3=A;
A=B;
B=C;
C=D;
D=temp3;
}

A = A + S[2*r+2];
C = C + S[2*r+3];
ct[0] = A;
ct[1] = B;
ct[2] = C;
ct[3] = D;
}

// Setup function for the S array.
void rc6_setup(unsigned char *Key)
{
// NOTE ENTER KEY HERE!
WORD L[c]={0x00000000,0x00000000,0x00000000,0x00000000};
WORD A,B;
int a=0,s=0,i=0,j=0,v=0,u=w/8;
S[0]=P;
for(a=1; a<=(2*r+3); a++){S[a]=S[a-1]+Q;}
A= 0;
B= 0;
v = 3 * maxOf(c,2*r+4);
for(s = 1;s<=v;s++){
A = ROTL(S[i]+A+B,3);
S[i] = A;
B = ROTL(L[j]+A+B,A+B);
L[j] = B;
```

```
i = (i+1)%(2*r+4);
j = (j+1)%c;
}
}

int maxOf(int op1, int op2){
if(op1>op2){return op1;}
else{return op2;}
}

/*Any other code following this is for testing purposes (
ex generation the S[] array)*/
int main(void){
// create arrays for input buffers
WORD i,j;
WORD ptIn[4];
WORD ctOut[4];
char* key;
int pass;
WORD rounds;

rounds = 0x00000014;
pass = 0;

//pseudo random plaintext
A[0]=0xff758600;
B[0]=0x0112fe50;
C[0]=0x0023eff5;
D[0]=0xff455980;
for(i=1;i<128;i++){
A[i]=(A[i-1]+0xef456234)%0xffffffff;
B[i]=(0x459001ff+A[i])%0xffffffff;
C[i]=(0x023ef031+B[i])%0xffffffff;
D[i]=(0x00260081+C[i])%0xffffffff;
}

rc6_setup(key);
for(i=2;i<(t-2);i++){
if((i%2)==0){evenS[(i-2)/2] = S[i];}
else{oddS[(i-3)/2] = S[i];}
}

for(i=0;i<19;i++){

ptIn[0]=A[i];
ptIn[1]=B[i];
ptIn[2]=C[i];
ptIn[3]=D[i];
rc6_encrypt(ptIn,ctOut);
ctA[i]=ctOut[0];
ctB[i]=ctOut[1];
ctC[i]=ctOut[2];
ctD[i]=ctOut[3];
```

```
}
// now make the hardware call.
#pragma CMLN_FUNC_CALL rc6top() SLICES=(0:4)
rc6top(rounds,A,B,C,D,S[0],S[1],S[t-2],S[t-1],oddS,evenS,ctAfab,
ctBfab,ctCfab,ctDfab);

/* We will now compare values for correctness
Range of valid data [block0 -> block18]:

ctAfab[25] -> ctAfab[43]
ctBfab[23] -> ctBfab[41]
ctCfab[25] -> ctCfab[43]
ctDfab[23] -> ctDfab[41]

*/

for(i=0;i<19;i++){
if(ctA[i]==ctAfab[2+i] && ctB[i]==ctBfab[i] && ctC[i]==ctCfab[2+i]
&& ctD[i]==ctDfab[i]){
pass=pass+1;
}
}

if(pass==19){asm volatile ("mov r8, 0x10");}
else{asm volatile ("mov r8, 0x20");}
return 0;
}
```