A Secure Single Clock Cycle

Reconfigurable Field Programmable Gate Array

by

James Millar

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES
*****
COPYRIGHT PERMISSION PAGE

A SECURE SINGLE CLOCK CYCLE RECONFIGURABLE FIELD PROGRAMMABLE GATE ARRAY

BY

JAMES MILLAR

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University

of Manitoba in partial fulfillment of the requirements of the degree

of

Master of Science

JAMES MILLAR © 2003

Abstract

Smaller transistors have resulted in more circuitry in the same area, operating at increasing speeds. The ability to place what seems like an endless amount of transistors with increasing density on a microchip is what made this project possible. While past Field Programmable Gate Array (FPGA) designs remained inactive during chip reconfiguration, I have designed an FPGA capable of reconfiguring its hardware design (or function) in just one clock cycle. This feat was accomplished by using a set of standby register banks and one active register bank for fast reconfiguration. The ability to save state information from the active registers into one of the standby registers is also possible. All registers are connected as a scan chain so as to make the FPGA testable for manufacturing errors. In order to secure the intellectual property of downloaded hardware designs, my FPGA decrypts encrypted hardware designs for its programmable registers. This is accomplished using both public key and secret key techniques for flexible and quick decryption suitable in a data broadcast environment. During the public key design process, it was also discovered that a more efficient method of encoding sequential (vs. parallel) circuitry in behavioural Very High Speed Integrated Circuit Hardware Description Language (VHDL) is doable by creating a small extension to the existing language.

Acknowledgements

I would like to thank my advisor Dr. Robert McLeod for the inspiring idea of creating a reconfigurable FPGA in one clock cycle, and his contacts in industry which allowed me to obtain financial support for my thesis work. I would like to acknowledge from Nortel Networks Rod Whitford, Kent Felske, and especially Mohamed Zaid for providing me with financial aid. In addition to Nortel's help, Jonathan Rose of Altera, provided funding for two summer students, Clint Stuart and Doug Cornelson, to work on the FPGAs initial routing scheme.

On a personal note, I would like to thank my family, especially my parents and grandparents for all of their financial support and encouragement. Lastly, I would like to thank my wife, Ellie Tsai, for helping to provide all of the food and shelter during my studies until I may find employment.

Table of Contents:

List of Figures

Chapter 1 Introduction

Overview of FPGAs

When computational circuitry was first implemented on silicon, custom design techniques were the sole method of production. These circuits generally had one purpose and were termed "ASICs" or Application Specific Integrated Circuits. As time passed, companies began to look at ways to reduce their up front costs for smaller projects by taking advantage of the lower costs of mass production. Today a variety of techniques exist including the use of Standard Cells, Mask-Programmed Gate Arrays (MPGAs), and Field Programmable Gate Arrays (FPGAs). While the list is by no means exhaustive, the first two provide semi-custom designers with the ability to create relatively high-speed designs with lower up front costs than a full custom design. FPGAs on the other hand provide designers with an inexpensive, mass produced, general purpose hardware device for rapid development and deployment on even the smallest of scales.

FPGAs come in two flavours, mainly those that are one-time-programmable and those that are reconfigurable over their lifetime. One-time-programmable (OTP) FPGAs are designed with what are termed "anti-fuses" as their configuration is set by making, rather than breaking, electrical connections as a household fuse would do. This is accomplished at first by having the anti-fuse reside in a high-impedance state prior to being programmed, and then in a permanently low impedance state following programming. The popular technique for reconfigurable FPGAs uses static RAM (SRAM) based technology and provides the ability to indefinitely reprogram the device which has some obvious advantages, namely:

- hardware designs can be upgraded (e.g. new standards to be implemented)
- a field technician may not be required to upgrade computing systems
- designs can be tested and retested without any additional fixed costs
- designs can be implemented immediately

Field programmable gate arrays (FPGA) were first produced by Xilinx Corporation and have since been manufactured by Altera, Advanced Micro Devices (AMD), QuickLogic and many others [1]. The general structure of an FPGA varies from manufacturer to manufacturer, however, the basic resource requirements are similar. Each FPGA must have a means of connecting to its pins, and this is accomplished through input/output blocks of logic (IOB). These IOBs must then connect to routing switches or interconnect blocks (IC's). IOBs must also connect to configurable logic blocks (CLB) or functional units (FU). A generalized diagram of an FPGA is provided in Figure 1 and various designs on this theme may be extracted from it [1,2,3,4,5,6].

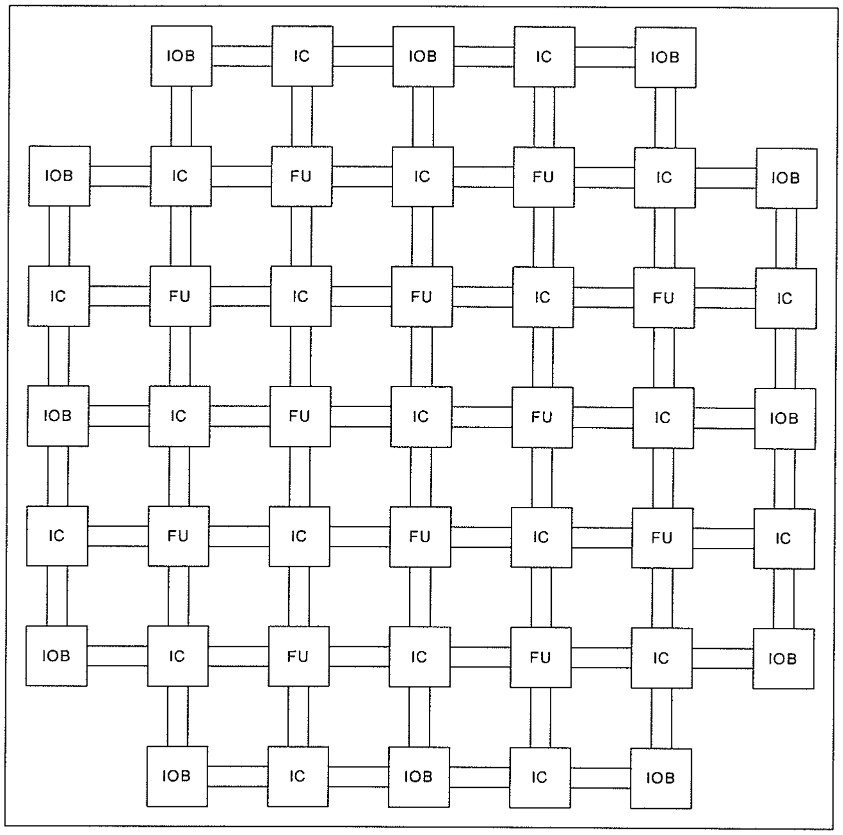

Figure 1 : Generalized FPGA Architecture

Modern IOBs not only connect the FPGA logic to its pins, but also provides some routability around a pin. Pin routability allows for greater flexibility when port pins and their functionality are fixed, and design changes are expected to occur. FU's (or CLB's) contain the logic that implements the application. Each FU may contain look-up tables (LUTs) and memory devices such as flip flops or simply basic NAND and NOR gates. Routing resources may have the flexibility to connect one block (IOB or FU) to its nearest neighbour or, in some cases, for faster data throughput, they may connect to a few blocks away.

At this point in time, one must begin to realize that no FPGA is useful without some form of computer automated design (CAD) tools. Since all of the actual hardware resides on an FPGA, designs are always written using a high level language such as Verilog or Very High Speed Integrated Circuit Hardware Description Language (VHDL). Once the design is implemented in a hardware description language (HDL), it is first simulated then synthesized. Synthesis is the process by which the HDL is converted into a stream of bits for downloading into the FPGA for the purpose of configuration. This bit stream can be quite long as FPGAs typically have hundreds of thousands to millions of switches to set on or off in order to be configured properly. The time to download such a configuration depends on its length, the clock speed, and whether or not the bit stream was loaded in parallel blocks or serially. A serial stream is often used as it requires fewer pins to implement and therefore represents a cost savings.

FPGA Application Issues

A variety of common characteristics of FPGAs include:

- the FPGA is designed using custom layout techniques
- configuration of the gate arrays do not allow for continuous application operation
- one configuration is configured at a time
- lack of an on-chip method for saving or passing on previous configuration state information
- lack of intellectual property (IP) protection.

This research project focuses on solving some of the common issues plaguing current FPGA designs and in the process, offers new or enhanced features in a flexible, easy to modify manner.

Thesis Objectives

Numerous FPGA designs have been proposed by various universities and companies using custom design techniques. Few, if any, have been fully specified in VHDL. To the best of this author's knowledge, no FPGA design has been fully specified in VHDL. By specifying an FPGA in VHDL, companies will have the ability to test new configurations fully through both simulation and by using another FPGA evaluation board as the base hardware. Furthermore, the designed FPGA will be enhanced by allowing the reconfiguration of designs in just one clock cycle. This will eliminate the downtime in a system that uses FPGAs intensively for its computation. Current designs must wait for large bitstreams to download and be reconfigured. Furthermore, the registers holding configurations in waiting (hereto referred to as the "standby registers") can be multiple in depth and extract state information from a previously running design. This should allow for smoother transitions between various reconfigurations and will be referred to as "soft-reconfiguration". Designs that do not make use of previous state information will result in what will be termed "hard-reconfigurations". An example of a soft-reconfiguration would be if the next to be reconfigured hardware design were initialized with the previous design's state information. This soft-reconfiguration would allow for some continuity of information flow between the designs. In the case of a hard-reconfiguration, individual designs to be reconfigured would have no knowledge of what had happened before them and run as they were first synthesized. Since the FPGA being developed is to be created similar to SRAM based designs, the question of intellectual property (IP) protection must be solved. IP protection is to be done using a combination of public and secret key techniques. A host of basic tools is also to be developed to automate much of the configuration bitstream generation. The FPGA design will be based on a scan design technique borrowed from testability or "Design For Test" (DFT) technology. Finally, recommendations to enhance the VHDL language for more rapid development are presented.

The final name given to this, single-clock-cycle reconfigurable FPGA is "RFPGA". Where R stands for "*Our*" FPGA and "*Reconfigurable*" FPGA. It also distinguishes our FPGA (RFPGA) from the FPGA that was used to target RFPGA to. The FPGA to target will be referred to as "Virtex", the trademarked name given to it by Xilinx Corporation. With the decryption units included, the top entity is called "ssFPGA", or the secure, single clock cycle reconfigurable FPGA.

Since no FPGA was actually fabricated, RFPGA should be considered a prototype, or design concept FPGA.

RFPGA was designed and simulated in Synopsys [7] and ModelSim [8]. The Xilinx ISE [9] development environment was used for synthesis and an XSV800 FPGA evaluation board by XESS [10] Corporation, containing a Xilinx Virtex chip, was targeted.

Chapter 2 Literature Review

Field Programmable Gate Arrays

While a variety of FPGA architectures exist [1], it is the academic variety that have been analyzed the most in public literature and are therefore easiest to learn from.

One such published FPGA is called "Triptych" [11] at the University of Washington in U.S.A.. The Triptych architecture was a custom design that tried to make better use of chip area by using routable logic blocks (RLBs). That is, routing functionality and implementation logic were on the same block. In doing so, the authors of Triptych hoped to minimize the tradeoff between creating an easily routable chip and high logic density. In order to make their design feasible, routing software capable of finding short paths between the necessary logic in a design was required as arbitrary and flexible routing was not efficient. In their research, they claim that most designs fit this criteria.

There are two primary disadvantages to the Triptych architecture that I can see. The first is that digital circuits only operate at the fastest speed of the slowest connection. It therefore only takes one poorly routed connection to slow an entire FPGA's operation. The second involves design upgrades. If an FPGA lacks a flexible routing architecture, then design upgrades requiring pin functions to remain fixed may be challenging to implement. It only takes one impossible upgrade to upset a customer.

Another FPGA named "LEGO" [3,4] was developed at the University of Toronto (U of T) in Canada. The FPGA designs of the U of T team were built on years of testing basic circuit blocks for ever increasing speed. Tests were done on logic-blocks to determine where it would be best to have hard-wired connections to increase speed without sacrificing too much in flexibility [12]. Other tests demonstrated that a four-input look-up-table (LUT) was the optimal choice for speed and chip area [13]. The final chip's design layout is similar to that of a Xilinx XC4000 chip. Each configurable logic block (CLB) has its four sides surrounded by a connection block (CB). Each connection block then connects to the nearest switch block for more generalized routing. Such a design allows for a flexible routing architecture and is easily repeatable over the entire chip. Repeatable designs often make for better use of silicon, hence the basic design of "LEGO" was chosen to build on for this thesis' "RFPGA" chip.

While some FPGA designs include special circuitry such as video decoders and phase-locked-loop circuitry, there is at least one design that has generalized such circuitry within a system of FPGAs to create a kind of supercomputing machine. Such a system was described as having a "Programmable Active Memory" [14] (PAM) by its authors. The authors claim that "At comparable cost, the computing power virtually available in a PAM exceeds that of conventional processors by a factor 10 to 1000" [14]. A system of over twenty FPGAs were coupled together with external SRAM and a system processor to allow for at least twelve hardware designs to time-share the FPGA hardware. While the system provides a means to generalize software functions onto hardware, it lacks communication back to an external processor making it difficult for state information of various routines to be passed easily from one design to the next. The reference is also not specific on how this might occur. The authors do however state to have proven the PAM concept through a variety of designs targeted for their system.

In the ASIC community, the design concept often implemented is Design For Testability or "DFT". The testability of a design is based on its controllability and observability. Controllability refers to how easily a circuit's elements can be set to various states by a test engineer. Observability is how easily one can monitor the circuit's output once those elements have been set. In general, scan designs result in the replacement of a circuit's flip-flops by the scan-design flip-flop (SDFF). The SDFF is connected in much the same way as the original flip-flop with additional wires that also make for a separate shift-register chain during testing. Therefore, a test vector would be first scanned in through the SDFF, then the vector is applied to the circuit for one clock cycle allowing the test vector to propagate through the asynchronous elements. Next, the resulting vector is scanned out of the chip for comparison. In this manor, synchronous and asynchronous circuitry are separated and both controllability and observability are increased. Popular techniques for DFT are described in [19] including IBM's level sensitive scan design (LSSD) approach and NEC's scan path [28].

When creating a testable chip, it is useful to follow some simple guidelines. The below guidelines are compiled from [19,20,21].

1. Avoid Logic Redundancy. For a designer using a high level programming language, this can be a challenge. The primary reason being that as an end user of synthesis software, unless you have a written guarantee from the company supplying you of that software, most designers may never know the details of the synthesis. Most synthesis software should automatically remove redundancy, but this is no guarantee.
2. Separate Sequential and Asynchronous Logic. Through separation, one is able to apply a test vector for one clock cycle, and read the result back without further interference. This increases both controllability and observability and breaks all combinational logic feedback loops that may result in circuit instability.
3. Avoid Gated Clocks. Some designers control clocks with gates to reduce logic area, however this results in difficult to test circuits. There are two reasons why it becomes difficult. The first is that an error in the gate controlling the clock may make it impossible to determine where in the

circuit the actual error is coming from. The second is that gated clocks are not controllable.

4. Design for Easy Initialization. Each flip-flop should have a reset input so that all sequential logic starts at a known state. In doing so, testing can begin immediately and does not require any test setup vectors. The reset should also extend to a chip's external pin.

Cryptography

Cryptography, "the art of writing in code or cipher" [15], has been gaining increasing importance for a variety of legitimate or illigitimate organisations. War probably spearheaded some of cryptography's modern beginnings. Examples of early cryptographic techniques included simple letter substitution algorithms and rotor machines, to the newer electronic stream and block ciphers as well as the famous Rivest-Shamir-Adleman (RSA) public key algorithm. Each of these techniques and many more are well described in Bruce Schneier's book "Applied Cryptography" [16]. While Schneier's book focuses on providing a clear explanation of the concepts of cryptography, another text, "Handbook of Applied Cryptography" [17] by Menezes, van Oorschot, and Vanstone takes a more academic approach and provides detailed mathematical algorithms for each type of cryptography.

Prior to the design of a cryptographic system, one should ensure at least the classical laws of a cryptographic system design by Kerckhoff [18] are followed, namely:

1. The system must be practically, if not mathematically, undecipherable;
2. It must not be required to be secret, and it must be able to fall into the hands of the enemy without inconvenience;
3. Its key must be communicable and retainable without the help of written notes, and changeable or modifiable at the will of the correspondents;
4. It must be applicable to telegraphic correspondence;
5. It must be portable, and its usage and function must not require the concourse of many people;
6. Finally, it is necessary, seeing the circumstances that the application commands, that the system be easy to use, requiring neither mental strain nor the knowledge of a long series of rules to observe.

There are two basic types of cryptography, symmetric and asymmetric. In symmetric cryptography, the keys are termed "secret keys" and their value is guarded at all times because the sending key and receiving key is the same. In asymmetric cryptography, the sending key is different from the receiving key. The sending key is often termed the "public key" and the receiving key is called the "private key". The public key for encryption can be known by anyone, and the private key for decryption, is only known to the receiver. Often, systems are designed so the human users don't even know what their secret or private key is. This added secrecy of the decryption key can help destroy the chance of the ancient arts of bribery, sex, and torture in unlocking encrypted data.

Symmetric Key Cryptography

A considerable amount of work has been done regarding symmetric cryptography, that is, a cryptosystem which relies on its key being the same for encryption and decryption. Unfortunately, this doesn't mean that it is well understood. A simple search of the IEEE database of designs will yield numerous proposals and successful attacks. It is therefore very important to fully understand the mathematics behind symmetric key cryptography before creating one's own algorithm. Learning however can be divided into various types of symmetric ciphers. There are block ciphers, stream ciphers as well as memory and memoryless systems. This thesis will concentrate on stream cipher design.

The ultimate stream cipher was not invented recently, in fact, it was invented in 1917 by Mauborgne and Vernan [16]. This cipher system was termed the "One-time pad" or OTP for short. To create an OTP assuming alphabetical letters are to be transmitted only, one could write out in a completely random fashion the letters of the alphabet on a pad of paper. Starting with the first letter on the pad and the first letter of the message to be encrypted, one could add their alphabetical location together, take mod 26 and that would be the encrypted character. The next letter to be encrypted would use the next letter on the pad and so on. The pad would only be used once for encryption and then destroyed. Such a system has not been found to be breakable due to the fact the sequence of letters on the pad is totally random. An example is given in Figure 2.

PAD (alphabetical location)	A 1	B 2	Z 26	E 5	F 6	Q 17	M 13	H 8	S 19	D 4
Plaintext Message (alphabetical location)	H 8	I 9	H 8	O 15	S 19	I 9	L 12	V 22	E 5	R 18
Sum	9	11	34	20	25	26	25	30	24	22
Mod 26	9	11	8	20	25	0	25	4	24	22
Decryption: ((above)+26-PAD)mod26	8 H	9 I	8 H	15 O	19 S	9 I	12 L	22 V	5 E	18 R

Figure 2 : Alphabet based One Time Pad (OTP)

If a one-time pad system were used more than once, it would then be possible for someone to compare messages for similarities and decrypt the plaintext. Correlations between popular code words or common characters such as the letter "E" in English, may allow a determined cryptanalyst to decode the entire plaintext. However, when used correctly, the OTP is considered unbreakable.

If a computer were to use an OTP, it could simply exclusive-OR (XOR) a random stream of 1's and 0's with the plaintext message's binary representation. The decryption operation would be similar – an XOR of the pad with the ciphertext would yield the plaintext.

Although OTPs have been extensively used in the past, their potential size and difficulty in transferring the pad to the end user ensures their use is limited to relatively small applications. The problem therefore is to create a seemingly random sequence of bits with a small and easily transferable piece of information or "key" which can be used to define the longer sequence. Such systems exist in the form of stream ciphers.

Stream ciphers may come in a variety of forms including "Linear Feedback Shift Registers" (LFSRs), which this thesis will concentrate on, and cellular automata. An LFSR is composed of a series of registers that usually shift their contents one bit at a time in one direction only. The contents of specific shift registers are then combined to generate a bit to fill the void left from a previous shift as shown in Figure 3.

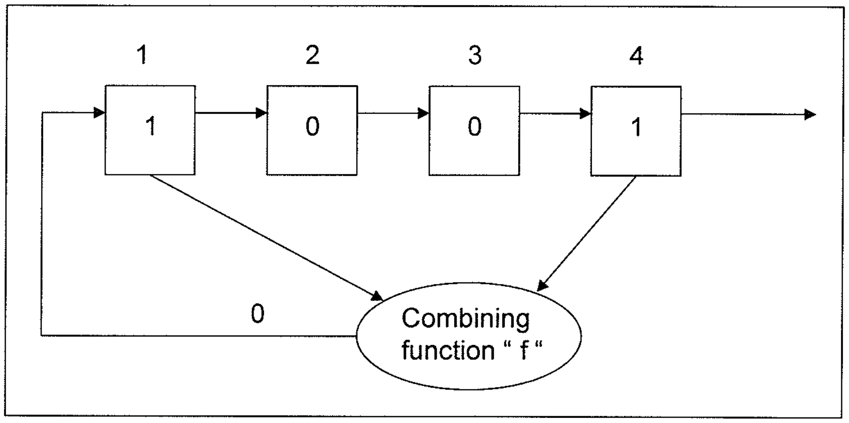

Figure 3 : BASIC LFSR

When the shift register is clocked, after one cycle, the current contents of 1001 would become 0100 in Figure 3. LFSR's have a varying number of registers and different combining functions to produce different sequences of 1's and 0's. If an LFSR is composed of "m" shift registers and is capable of cycling through all possible states, then this LFSR is said to have a maximal-period, and the resulting output sequence is termed an m-sequence. In order to generate such a sequence, the combining function "f" must be composed of a primitive polynomial. Primitive polynomials are said to be "irreducible" just as common prime numbers are not factorable (beyond 1 multiplied by themselves). The definition "a polynomial $f(x) \varepsilon$ $Zp[x]$ of degree $m \geq 1$ is said to be irreducible over Zp if it cannot be written as a product of two polynomials in Zp[x] each having degree less than m" can be found in [41] along with techniques for testing and generating primitive polynomials. The testing and generation of primitive polynomials is outside the scope of this thesis, however, they are required to create useful results for LFSR designs in cryptography.

Much of the basic understanding for LFSR design has been well summarized by Golomb [45] and Rueppel [42], a condensed version of which will be used here for the purpose of providing basic definitions and understanding.

Let "p" be a prime number, and "Fp" represent the set of integers {0,1,...,p-1}. Given the algebraic system <Fp, +, •> we use symbols a and b to represent Fp's integers for:

$a + b = Rp\ (a+b)$

$a \bullet b = Rp(a\ b)$

to define a Galois field of order p, denoted GF(p). The remainder is defined as Rp when an integer n is divided by the field's maximum prime number p. For a stream cipher based on flip flops, p equals 2 and the set of integers is {0,1}. In this field termed GF(2), arithmetic operation Rp is termed "modulo-2 arithmetic". As has been mentioned, polynomials are used to describe the feedback taps of an LFSR. These polynomials in GF(p) are written as:

$$a(X) = a_0X^0 + a_1X^1 + a_2X^2 + \ldots + a_nX^n$$

where: a_i is the leading coefficient and

X is called the indeterminate.

Definitions for polynomial addition and multiplication are given in [42] on page 18 and are worth repeating here so the reader may understand their linkage to LFSR design.

Addition is performed by:

$a(X) + b(X) = SUM[(a_i + b_i)*X^i]$ for i=0 to n

Multiplication is performed by:

$a(X)b(X) = SUM(c_k x^k)$ for k=0 to (n+m)

where

$c_k=SUM(a_i \bullet b_j)$ for (i+j=k, and 0<=i <= n, 0<=j<=m)

and

n and m are the degrees of a(X) and b(X) respectively.

Multiplication of course introduces us to division where a(X) divides c(X) if c(X)=a(X)b(X). If a(X) or b(X) is a simple constant value of GF(p), then c(X) is said to be an irreducible polynomial. That is, we have a primitive polynomial.
Useful polynomials are those that can no longer be factored and are termed irreducible polynomials. If an LFSR is of length r, and its feedback polynomials are irreducible, then the shift register is guarenteed to go through all 2^r-1 possible states before repeating. Factoring is a challenging operation that will not be explained here, however long division which may result in factoring is the same with numbers. Examples of polynomial division are given in [44] and a software program to generate irreducible polynomials can be found in [43].

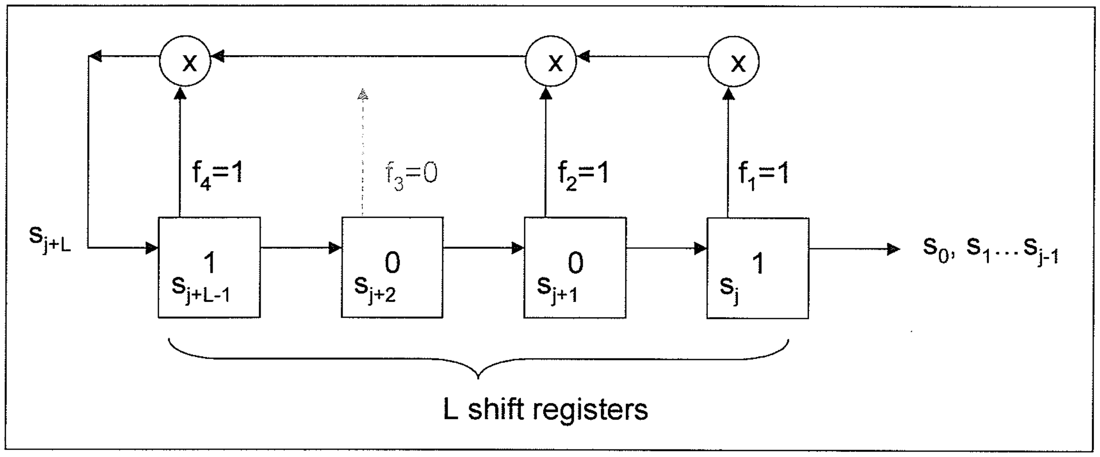

Figure 4 : Generalized LFSR

A generalized LFSR as shown in Figure 4 can be used to explain the mathematical operation of LFSRs.

The feedback polynomial coefficients are represented by 'f' while the state of the shift registers is defined by 's'. There are a total of L shift registers in the design. Given we are working in GF(2), addition or multiplication can be accomplished with the use of an XOR gate as is done in our design. The feedback polynomial can be described by:

$F(D) = 1 + f_1D^1 + f_2D^2 + f_3D^3 + f_4D^4$

where the exponent of the power D represents the delay, and D is referred to as the 'indeterminate' and f represents the feedback coefficients. Substituting in their values yields:

$F(D) = 1 + D + D^2 + D^4$

To determine the sequence of digits 's', the coefficients of F(D) is multiplied by the state of each shift register or:

$s_{j+L} = -SUM(-f_i s_{j+L-i})$ for i=1 to L, j>=0

or simply $s_{j+L} = SUM(f_i s_{j+L-i})$ mod 2, for i=1 to L, j>=0

There are two points worth mentioning here. The first is that an LFSRs output is highly predictable and must be combined with others to form a less predictable bit stream for cryptography. The second is that this combination of LFSRs must produce as close to a truly random sequence as possible so as to limit the analyzability of the stream. Golomb's [45] popular definition of the randomness of an ideal coin toss is formulated as three postulates:

1. The number of heads is approximately equal to the number of tails.
2. Runs of consecutive heads or of consecutive tails frequently occur, with short runs being more frequent than long runs.
3. Random sequences possess a special kind of auto-correlation function, peaked in the middle and tapering off rapidly at the ends.

Golomb's definition however generally describes an LFSR using an irreducible polynomial for feedback. Rueppel however recommends that randomness should include some measure of *unpredictability* and suggests the Berlekamp-Massey LFSR synthesis algorithm[46] be used as a test. As the synthesis algorithm generates a larger LFSR to represent a finite bit stream, s, this may indicate that the stream's digits are less predictable or, the linear complexity of the sequence has increased. Rueppel goes on to provide his own mathematical clarification of unpredictability. Rueppel's mathematical definitions are necessary to ensure that trivial exceptions like a stream of zero's followed by a single one, are not deemed secure by an algorithm like Berlekamp-Massey's.

The basic fundamentals of a stream cipher system are illustrated in Figure 5. Once the LFSR has been seeded with a key, its output is then passed through a non-linear function. Non-linear functions generate the unpredictability necessary to make a stream cipher secure. Such a function may be generated on the existing state of the LFSRs, the outputs of a few different LFSRs, or by means of a custom LFSR clocking scheme.

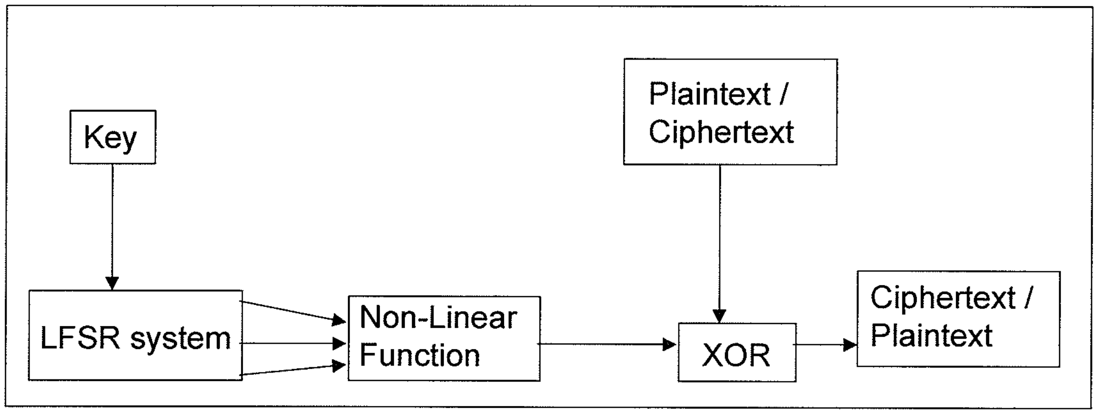

Figure 5 : Basic Stream Cipher System

The output of the stream cipher occurs just before the XOR stage in Figure 5. XOR is used to combine the plaintext with the stream cipher output yielding the ciphertext. The exact same operation is applied to turn ciphertext into plaintext at the receiving end.

In general stream ciphers may be synchronous or self-synchronizing. Self-synchronizing ciphers will self-correct errors that may occur due to noise or attacks. Synchronous ciphers have no such ability, however, this also makes them immune to many attacks such as injection, deletion or replay of the cipher text since any of these would result in a permanent loss of synchronization [42].

Public Key / Asymmetric Cryptography

Public Key cryptography has been around long enough that its initial patent by the RSA group has expired that it may be applied by anyone. The first known publication of public key cryptography came from Diffie and Hellman [29] in 1976. Their paper, "New Directions in Cryptography", gave a theoretical description of how public key cryptography was to work. Unfortunately, they gave neither software nor hardware designs for their concept. Diffie and Hellman didn't get the patent on it (but they did for key exchange [30,31]). It wasn't until two years later Rivest, Shamir, and Adleman pieced together the first public key cryptography algorithm and signature scheme known as "RSA" [32,33]. To this day, RSA has remained a secure algorithm with no efficient technique for breaking the scheme. Extensive documentation and a variety of implementations have been published. The RSA web site however provides one of the best sources of summarized information at http://www.rsasecurity.com/ .

The basic technique for public key cryptography is as follows:

1. Define p and q to be two large prime numbers chosen at random, and of equal length.
2. Let n = p * q.
3. Define e to be the encryption key, and set it to a number relatively prime to (p-1)*(q-1).
4. Compute the decryption key, d, as : e*d=1 mod ((p-1)(q-1)) using the extended Euclidean Algorithm.
5. The pair (e,n) are the public encryption keys.
6. "d" is the private decryption key, and requires "n" to work.
7. The encryption formula is:
   $c_i = m_i^e \bmod n$,
   where: m is the message, and m is smaller than n, but fixed in binary digit length.
   the subscript "i" refers to each message or ciphertext block.
8. The decryption formula is:
   $m_i = c_i^d \bmod n$

While RSA has never been broken, failures to properly implement the RSA protocol would result in a weak form of cryptography. Bruce Schneier explains in [16] a variety of attacks against the protocol including:

a) Chosen ciphertext attack,
b) Common modulus attack,
c) Low encryption exponent attack,
d) Low decryption exponent attack,

e) Attack on encrypting and signing.

Each form of attack requires careful consideration [34,35] when implementing RSA.

Fast Modular Multiplication for Public-Key Cryptography

Modular multiplication involves simplifying the expression: (a*b)mod n. In order to carry out modular multiplication, one may try the following classical method:

1. c<=a*b.
2. Determine c mod n.
    2a. d= remainder (c/n), where d is the answer.

Note, in order to perform c mod n, a division operation is performed. Division on a computer is generally slow with the exception of base 2 division which involves simple right shift operations. This can be a problem for cryptographic systems that must compute a^e mod n, where e is a large exponent. While this problem can be easily managed by repetitive multiplication and division [16], it is still slow to compute. This problem was later solved by the mathematical scientist, Peter Montgomery [22]. "Montgomery Multiplication" as it has since been referred to requires intensive division for only set up and completion of operations. Division during the Montgomery process only involves simple shifts. Therefore, Montgomery's method is generally well suited for public key cryptography and has seen wide application in research.

Montgomery's method also works for any residue system. To perform Montgomery multiplication, some precomputation is required where:

1. For N>1, where N represents the residue class, modulo N.
2. R is chosen as the system radix and is greater than N as well as coprime to N. In order for computation to be quick, R should be a power of the machine word size.
3. Compute R^{-1} such that $R*R^{-1} \bmod N = 1$. Note, R^{-1} is the multiplicative inverse of R.
4. Compute N^{-1} such that $R*R^{-1} - N* N^{-1} = 1$.

Prior to computing $a^e \bmod N$, "a" must first be converted to the chosen residue system R.

5. A <= a*R mod N
6. X <= 1*R mod N

Borrowing a simplified algorithm from [23] we can complete the computation of $a^e \bmod n$ with:

```
7a. i<=j-1 downto 0
    7b. X <= MonPro(X,X)
     7c. if $e_i$=1 then X<=MonPro(X,A)

8. return x<=MonPro(X,1)
```

where *MonPro* is:

```
function MonPro(A,B)
 i.   t <= A*B
 ii.  u <= [t+(t * $N^{-1}$ mod R) * N ] / R
 iii. if u $\geq$ N then return (u – N) , else return u
```

Looking at line (ii) of the function MonPro, we have at first glance what may appear to be two computationally intensive operations, namely "mod R" and division by R. Remember, R is a power of the machine word size. To simplify things, lets assume $R=2^b$, where b is an integer greater than zero. Then, "Z mod R" simply involves selecting the least significant "b" bits of Z.

For example, compute "$22_{10} \bmod 16_{10}$".

$22_{10}=10110_2$
$16_{10}={2_{10}}^4$
Therefore, $10110_2 \bmod {2_{10}}^4 = 0110_2$ or 6_{10}.

From the example it can be seen that for the base 10 power 2^4, one only needs the 4 least significant binary digits to obtain the modulus. Mod R, where R is related to the machine word size is simple to compute (for any base).

The division by R is easier to explain. There are simply "b" logical shifts to the right to complete the computation.

Numerous theoretical enhancements have been made on Montgomery Multiplication as well as custom hardware designs [23,24,25,26,27] and many, many more. To the author's best knowledge, the published hardware designs to date are all custom. Usually, the only significant improvement comes from performing the multiplication and reduction steps in parallel. Others may provide performance enhancements by placing hardware elements close together in what's termed a "systolic array", a variation of a pipelining technique.

Data Broadcast Cryptography

The concept of broadcast encryption is somewhat of a Mecca for the business community with its practical considerations being the devil. Broadcast encryption is when a specific subset of users from a community receive encrypted data for which only they can decrypt. This simple idea has wide ranging applications in business.

One of the biggest failures of broadcast encryption technology, is also the perfect example as to where one would like to deploy such technology. Digital Versatile Discs (DVD), if you don't have them, you will. It appears to be that every video store around is renting an increasing number of DVDs. The movie industry would love to prevent the unauthorised copying and distribution (piracy) of its product through some form of broadcast encryption. Broadcast encryption would allow every user who has legally obtained a copy to view the movie, and those who attempt to illegally obtain a copy fail.

The DVD cipher system is called the "Contents Scrambling System" or CSS for short. Putting together a group of sub keys and master keys allows one to descramble the contents of a DVD. A complete description and cryptanalysis of the algorithm is given in reference [36]. Why did CSS fail so badly? The designers of the CSS algorithm broke a golden rule of cryptography. That is, for maximum security, one should only use published algorithms that have seen only failed attacks over a period of time.

Broadcast encryption was first published by Berkovits in 1991 [37]. There are a variety of researchers who continue to look into the problem of broadcast encryption. In Canada, D. Stinson [50,51] of the University of Waterloo has published papers on the topic. Overseas, considerable attention has been given by Taiwan Universities [38,39,40].

For this thesis, Chiou and Chen's broadcast encryption algorithm [40] was implemented in Java. Chiou and Chen's algorithm was used to encrypt the secret session key, and then ssFPGAs hardware design decrypted the secret key so the application could then be decrypted using a Gunther Alternating Step Generator (G-ASG).

One of the primary problems left unsolved by such encryption algorithms is the greater the number of end users, the more setup information needs to be sent. Therefore, each scheme must include a secure key distribution system to ensure setup overhead doesn't become unmanageable. Considerable research is still required to make broadcast encryption more practical.

Chapter 3 RFPGA

What's In a Name?

It is worth mentioning again the name of this FPGA came about after examining its function and how it was going to be tested. The "R" in RFPGA stands for both "Our" and "Reconfigurable". It is "Our" FPGA because the initial design for the switch boxes, c-blocks, s-blocks, and I/O blocks were done by two summer students and the rest, including modifications to the summer student's work, was done by myself. The FPGA is also designed to be reconfigurable in a single clock cycle and so we needed a suitable letter to represent that. RFPGA (Our FPGA) is therefore the one we designed, while Virtex is used to refer to the FPGA I would target RFPGA to for synthesis. Virtex is a trademark of Xilinx Corporation.

The remainder of the project involved the creation of a public key decryption unit and a Gunther Alternating Step Generator. RFPGA and the decryption units are managed by a controller entity. Each of these entities are then wired together with the super-entity "ssfpga" – or the Secure Single clock cycle RFPGA (ssFPGA).

RFPGA Requirements

As mentioned in the introduction chapter, RFPGA was required to overcome certain FPGA problems and at the same time, achieve new functionality. The resulting prototype has the following features:

- Reconfigure its functionality in just one clock cycle.
- Save a state worth of information.
- Designed with easily modifiable VHDL source code.
- Provide intellectual property protection.
- Designed using a scannable architecture for testability.

The highest level view of such a design is illustrated in Figure 6. Each major component was built as a separate VHDL hierarchy, and then simply tied together through one 'super entity' called "ssfpga.vhd". This allowed for simplified testing and modeling of the circuitry at every stage of the design cycle.

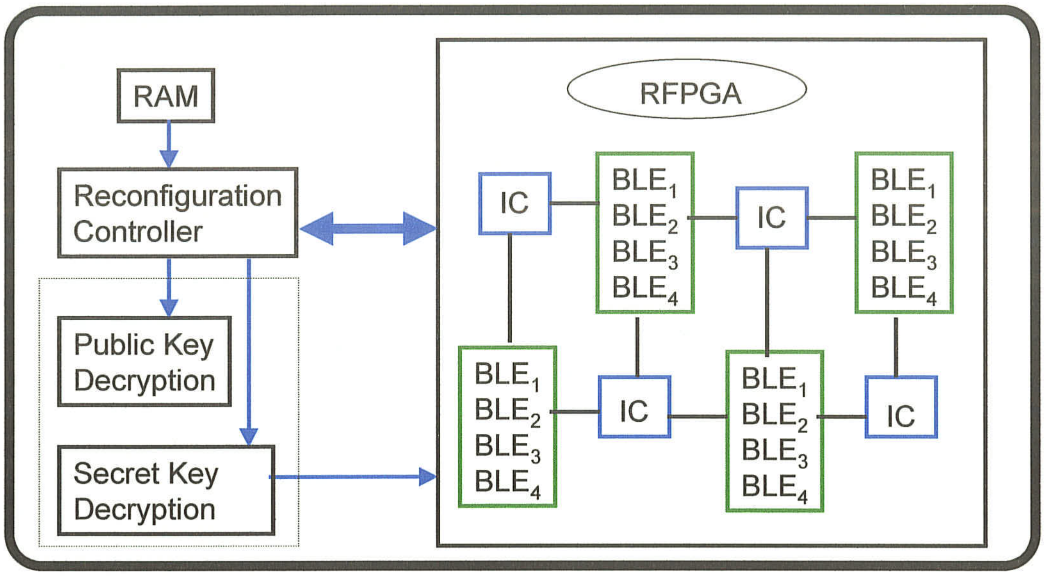

Figure 6 : High Level ssFPGA View

acronyms: IC – interconnect, BLE – Basic Logic Element

RFPGA Design Outline

ssFPGA has three major components and a super-entity to tie each one together.

1) RFPGA.
2) The Decryption Unit.
3) The Reconfiguration Controller Unit.

Within the RFPGA unit lie 3 functional units, which subdivide as follows:

1) External Pin Connecting Input/Output Blocks (IOBs) (outside Figure 6)
   a) top, bottom, left, right
2) Interconnect Blocks (ICs)
   a) Switch Blocks
      i) corner blocks
      ii) mid (top, bottom, left & right blocks)
      iii) middle blocks
   b) Connection Blocks

        i) Vertical or Y connection blocks
        ii) Horizontal or X connection blocks
3) Configurable Logic Blocks (CLBs)
    a) Basic Logic Elements (BLEs)

Each of the aforementioned functional units are then tied together through a routing architecture entity. Within this entity it is possible to specify the dimensions of RFPGA. This way, as new physical layer technologies develop (e.g. enhancements to silicon fabrication technology) and more transistors or metal layers are packed per unit area, larger chips may easily be reproduced. The primary variables for modifying the size and functionality of RFPGA include:

1) size.
2) twidth (track width).
3) IntClusterSize.
4) ScanWidth.

Through simple modification of these variables, one may effortlessly play with the speed, CLB coarseness, chip area, and routing capabilities of RFPGA. The "size" variable determines the overall "grid size" of RFPGA. For example, to design RFPGA with 3 CLB's across and 3 CLB's vertically, set size to "3". For a much larger RFPGA, set size to 50 for a 50x50 CLB FPGA. Note however, that as the RFPGA chip size increases, so may its routing requirements. That is, it may be more difficult to find a path from one CLB to another in a larger design. To alleviate this problem, one may increase "twidth" or the track width between all blocks. This will allow for greater routing capability between all IC's and CLBs, much the same as adding extra highways to a province would do. Increasing IntClusterSize will allow the user to pack more BLE's within a CLB. In doing so, one may make better use of available chip area and also increase the speed of implemented designs. Speed increases are possible this way because all routing within a CLB is fully connected – which of course makes the job of any routing software considerably less challenging as well. Note however, additional programming of the "c_block_x.vhd" unit is required for interfacing with multiple BLE outputs. If a greater number of standby scan chains are desired, then modify the ScanWidth variable. A value of two represents one active chain, and one standby chain. A value of X units represents 1 active chain and (X-1) standby chains. Increasing the number of standby scan chains allows one to reconfigure downloaded hardware designs in a single clock cycle to a greater selection of designs and increases the ability of the system to store more states of a given hardware design. By storing more states one can use RFPGA to emulate hardware designs as software more easily by creating an artificial stack of state information and functions.

The next major component is the decryption unit. The decryption unit is composed of the following two major entities:

1) Gunther inspired Alternating Step Generator (G-ASG).
2) Montgomery Enhanced / Broadcast Decryption Public Key (PK) Unit.

In addition to modifying RFPGAs dimensions, the decryption units have their own scalability constants such as:

1) vbits.
2) SRSize.

The constant "vbits" or "vector-bits" allows for a variation in the public-key decryption routines' variable size. All variables use this same size, therefore, vbits must represent the largest required word size for the output of any given calculation. "SRSize" or "Shift Register Size" is used to vary the length of the shift registers associated with the Gunther ASG. A larger shift register size would allow for a larger key and therefore greater security.

Within the Gunther ASG are third generation single clock cycle reconfigurable blocks (RB3) as opposed to RFPGAs fourth generation blocks (RB4). By using RB3's I was able to create a design with configurable feedback connections by reusing previously designed hardware. This allows for faster turnaround time in the design cycle as well as reducing the maintenance costs of redesign.

The original intention of the public key decryption unit was to be able to work with algorithms designed for broadcast encryption. In the end, the mathematics for such a unit turned out to be backwards compatible with existing RSA techniques and hence little modification was required on my part. The PK unit however was designed using Montgomery's technique for quick modular operations as mentioned in the literature review section.

Lastly, the control unit is required to automate the setup and running of RFPGA and the decryption unit. The controller is capable of detecting a reset condition, and know which designs to load into what RFPGA standby registers as well as when to reconfigure them active. The specific capabilities of the controller unit will be outlined later. Note, the ssfpga unit is the top entity which has little function other than to tie each of the three major subunits together as one entity.

RFPGA components

The heart and primary focus of this thesis was to develop an FPGA that would reconfigure in a single clock cycle. Initial designs led to a reconfigurable block named "RB3" which, much later in the project, evolved into "RB4". RB3 is still used in the G-ASG decryption section, while RB4 is the only scannable flip-flop used in the FPGA. Since RB3 is a simpler design, it will be described first.

"RB3"

RB3 was designed to be a simple and easily modifiable scannable block capable of reconfiguration in one clock cycle. Its components include two D-flip-flops as well as two multiplexers as shown in Figure 7. Each flip-flop was carefully included to ensure that scan operation was not controlled at the clock input. This meant that all flip-flop I/O had to be retimed through the multiplexers. Therefore, the standby chain, which is at the top of Figure 7, retimes its output back through a multiplexer. When data is to be scanned in, the multiplexer is configured to select the input from the previous shift registers output, and the data marches through the chain until retimed again. The active chain at the bottom of Figure 7 operates in a similar fashion, but with greater functionality. The multiplexer controlling the active chain allows for one of the following inputs:

1) operational data from a user's circuit design.
2) retiming of circuit data.
3) reconfiguration from the standby chain to the active chain.
4) test data to be scanned through the chain.

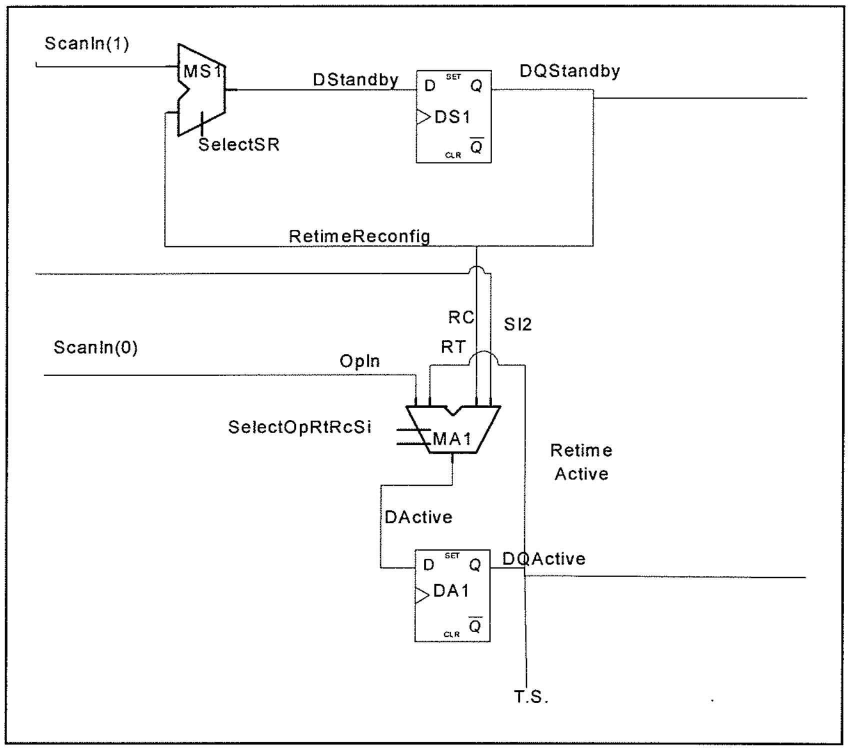

Figure 7 : RB3

While RB3 was simple and easily incorporated into higher level entities, some time during the testing stages of the project it became apparent that RB3 would not suffice for the final RFPGA design. One of RB3's biggest drawbacks was the possibility that a short circuit could occur in the final design. This is because the state of the active chain would be initially undetermined resulting in buses with logic 0 and 1 on the same line. RB3's other disadvantage was in its lack of support for a reset switch, and multiple banks of standby registers. This meant a larger reconfigurable block would have to be designed, but it would prove necessary for RFPGAs stable operation.

"RB4"

RB4 solved the problems of RB3 and provided some enhancements. Starting with the standby chain at the top of Figure 8, there are multiple scan chains each controlled by their own separate multiplexer. By having separate control lines, one is able to scan in different designs for reconfiguration. Furthermore, the number of standby chains is easily controlled by the ScanWidth constant which allows the implementer to tradeoff useable chip area against the maximum number of stored designs and state information.

State Information from a currently running design can be saved in any one of the standby designs through the multiplexer controlled by the SelectRtSs (Select Retime or Save State) line. This multiplexer allows an application to substitute retimed data with that of active data thus allowing for data to be reused or passed to other applications. Note, this control line should be configured with the SelectSiRt control line in mind. SelectSiRt is set to "Si" or "Scan In" when loading the standby registers, while the "Rt" or retime setting will pass the output of the SelectRtSs multiplexer through to the flip-flop.

Due to the multiple standby chains, there had to be a simple technique for selecting which data was to be reconfigured. This could be accomplished through either a multiplexer or tri-state buffers onto a bus. It was decided that tri-state buffers would be used so as to reduce area requirements. These tri-state buffers are controlled using the SelectSout input line. Note that tri-state switches are not always mapped efficiently onto a targetted FPGA, but since RFPGA is meant to be fabricated as an ASIC, it was deemed appropriate.

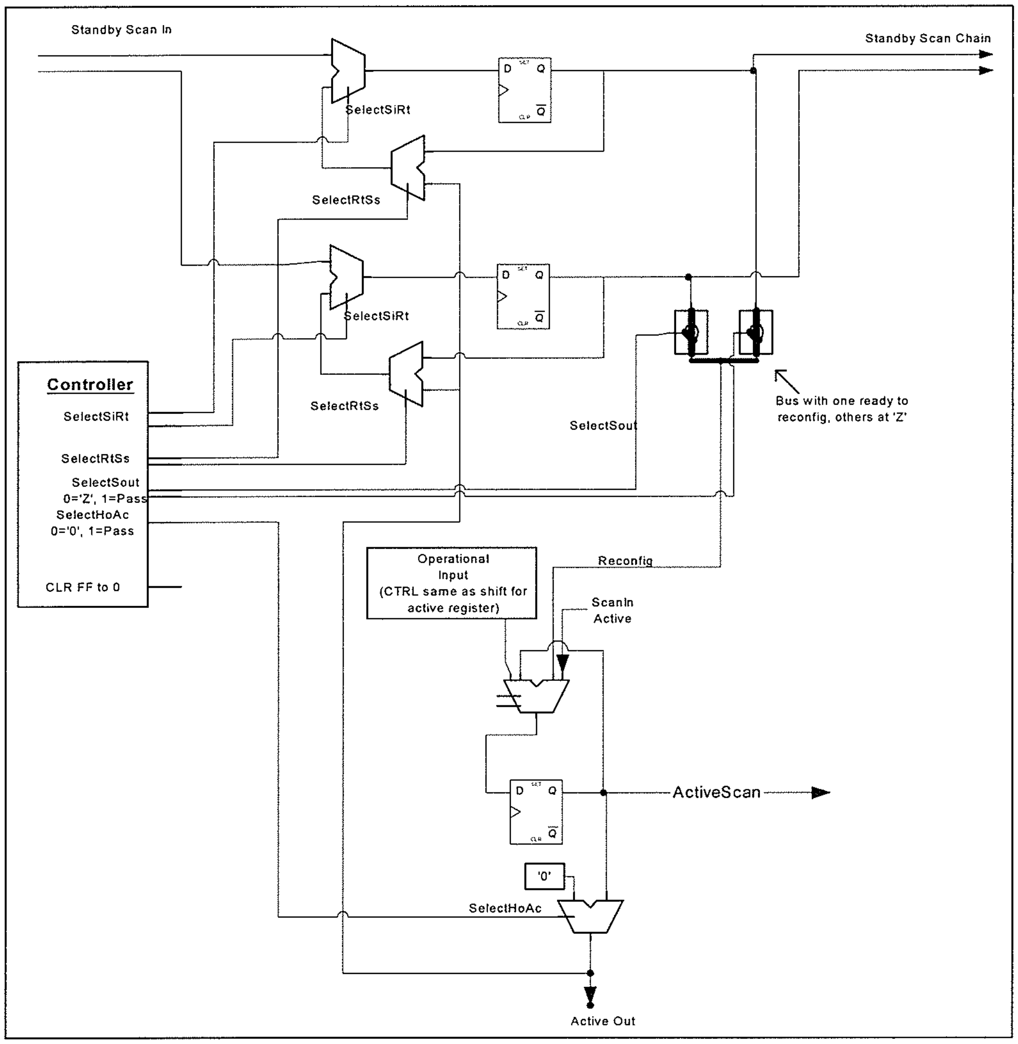

Figure 8 : RB4 with Controller Hook-Up on the side

The final major enhancement was to allow the output of the active chain to be set to zero. This is helpful during the setup stage and avoids any possible chance of short-circuits while scanning is taking place. Once RFPGA is loaded, the active D-flip-flop's output is allowed to present itself to the switch it is setting on or off, or in the case of a tri-state, it could be the high-impedance value represented as a 'Z'. Each flip-flop also has a reset switch and can be cleared to zero. The control line to the multiplexer that selects a zero as output, or the value of the active flip-flop is SelectHoAc.

It should be noted that any data to pass through to the active flip-flop is controlled by a line called SelectOpRtRcSi. For this line, "Op" allows operational data to flow through to the active flip flop. This is to allow for synchronous hardware applications. If "Rt" is selected, then the output of the active flip-flop is simply retimed, or kept in a holding pattern. "Rc" invokes a reconfiguration from one of the selected standby registers, and Si is used when data is to be scanned directly into the active chain for testing purposes.

RB4 was then incorporated into a basic logic element (BLE), which is then built into a configurable logic block (CLB). Each BLE is similar in design to what's found on U of T's LEGO or Xilinx' XC4000 logic elements[3,4,9]. There can be as many BLE's per CLB as is desired. One such BLE is shown in Figure 9.

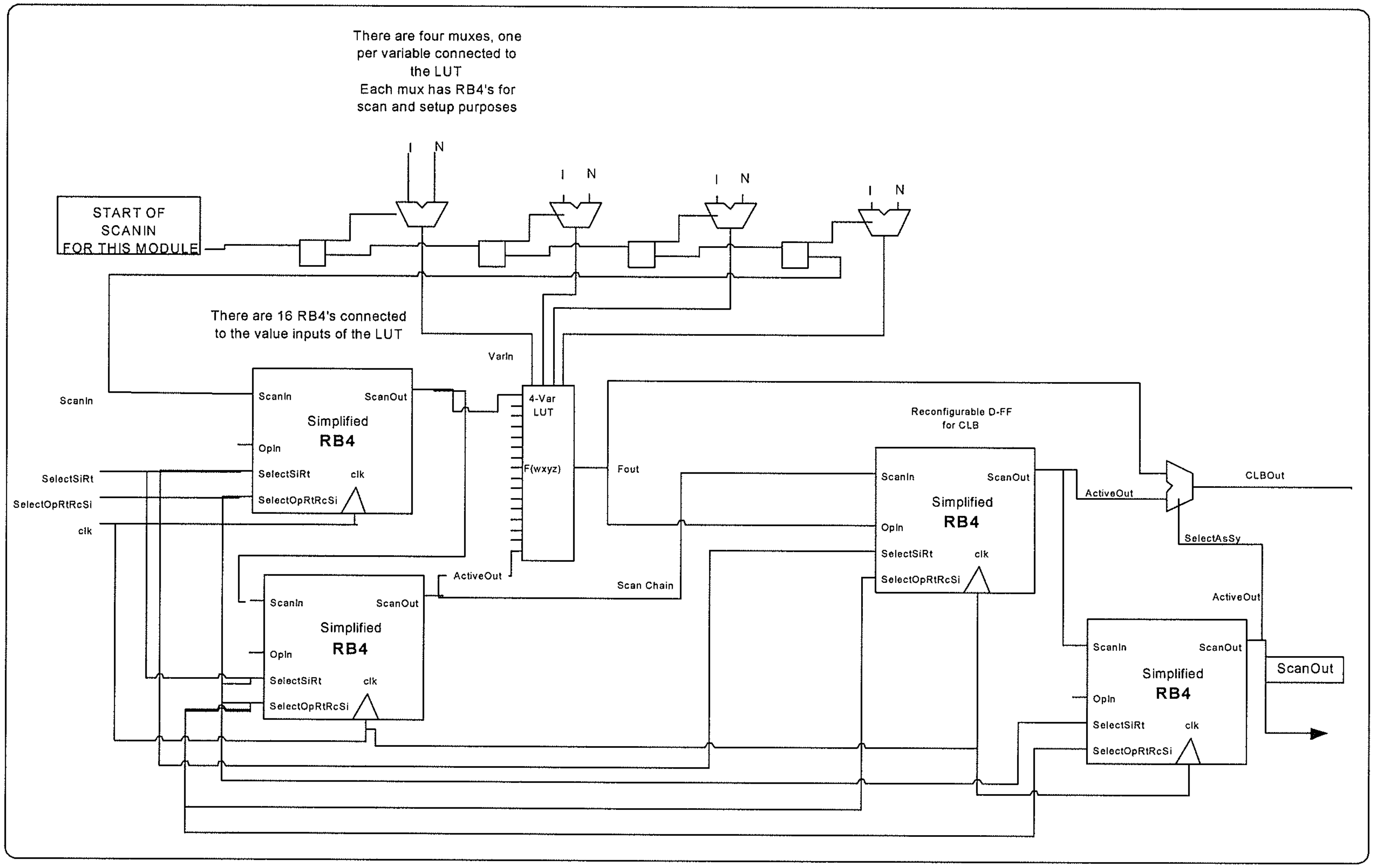
There are four muxes, one per variable connected to the LUT
Each mux has RB4's for scan and setup purposes
I N
START OF SCANIN FOR THIS MODULE
There are 16 RB4's connected to the value inputs of the LUT
VarIn
ScanIn
SelectSiRt
SelectOpRtRcSi
clk
ScanIn ScanOut
Simplified RB4
OpIn
clk
4-Var LUT
F(wxyz)
Fout
ActiveOut
Scan Chain
Reconfigurable D-FF for CLB
CLBOut
SelectAsSy
ScanOut

Figure 9 : Basic Logic Element within a Configurable Logic Block and Scan Chain.

The BLE of Figure 9 illustrates how RB4 is also used as part of the scan chain. The scan chain begins with the first input multiplexer at the top of the diagram where it is used to hold the selected values for input to the look-up table (LUT). The number of RB4's used here is dependent on the size of the generated multiplexer. The next chain of RB4's is down the left side of the LUT. There are 16 RB4's, only two of which are shown attached to the LUT. These RB4's active output values are what determines the LUT's function. The output of the LUT can then be accessed in asynchronous fashion or be passed through another RB4 to be used in a synchronous fashion. The asynchronous or synchronous path ways are selected by a multiplexer, just before the final output stage. The aforementioned RB4 is also part of the scan chain, and comes just before the final RB4 which sets the final multiplexer's selection value. In this configuration, the scan chain can apply in one clock cycle a new value to the circuitry, and then scan it out of the chip for test evaluation.

Switching Component

During the FPGA design, some components were easily simulated, but not synthesizable. While the synthesis software should have been able to extract a circuit, it probably wasn't sophisticated enough, hence a more structural redesign was required. The most notable redesign was the switching unit. The initial switching unit's design described the flow of data from one port to another, given input from a control line. The initial design also lacked clarity in describing how the data was to be handled by the circuitry connected to the entity's 'in/out' ports for the synthesizer. The in-out ports circuitry had to be changed to clearly allow data to flow bidirectionaly, using circuitry behind the ports that is unidirectional. Figure 10 illustrates how this was done. There are now four groups of two tri-State buffers to allow data to flow in or out of the top, bottom, left or right side of each basic switch. If the control line turns on a switch, data is allowed to flow through, otherwise, the tri-state enters a high impedance state. By only allowing one buffer to allow data in and one to allow data out, there is no conflict on the bus (represented as a ring in the centre). Furthermore, by specifying the exact type of circuitry to be used and how it interfaces with in/out ports, the design is easily synthesized.

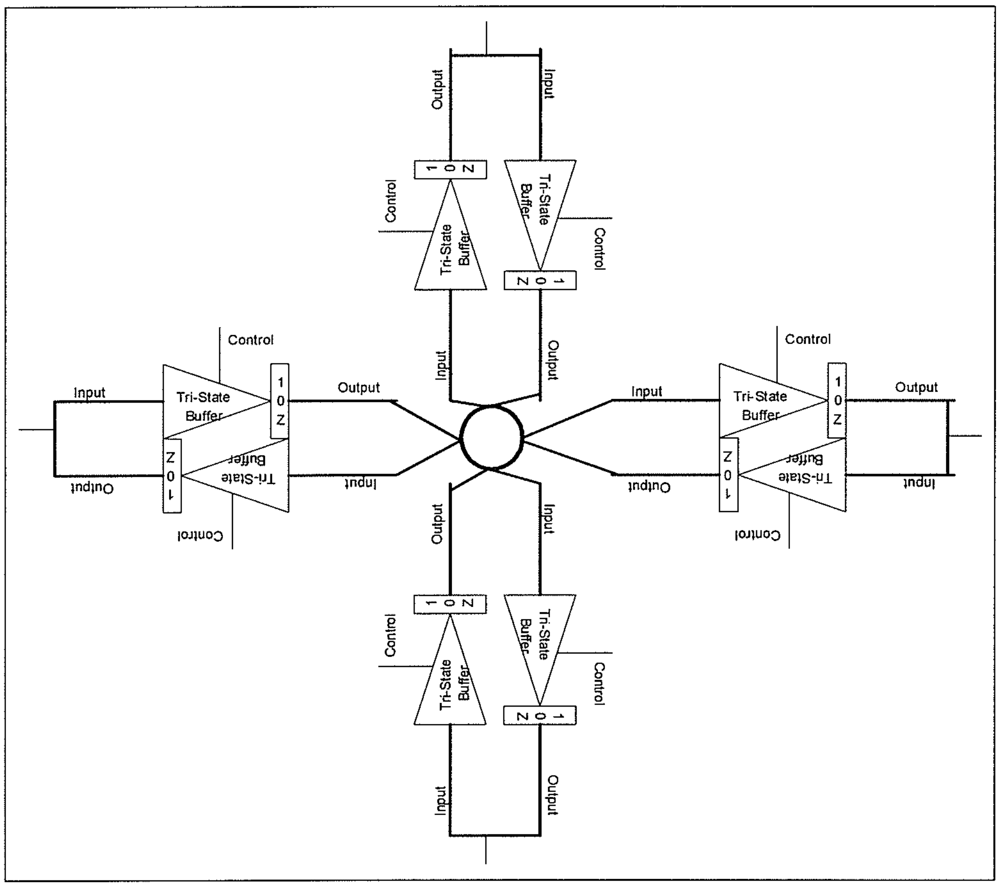

Figure 10 : Latest switch_6 entity

Each switch_6 entity is then combined to form a switch block as shown in Figure 11. The switch block is the major interconnect (IC) device in the FPGA allowing for routing in any direction. Each FPGA IC uses the same number of tracks on the top, bottom, left and right hand sides to ensure they line up when all put together. The final design of the individual switch_6 entity was simple to implement, and easy to replicate for the creation of the larger switch block.

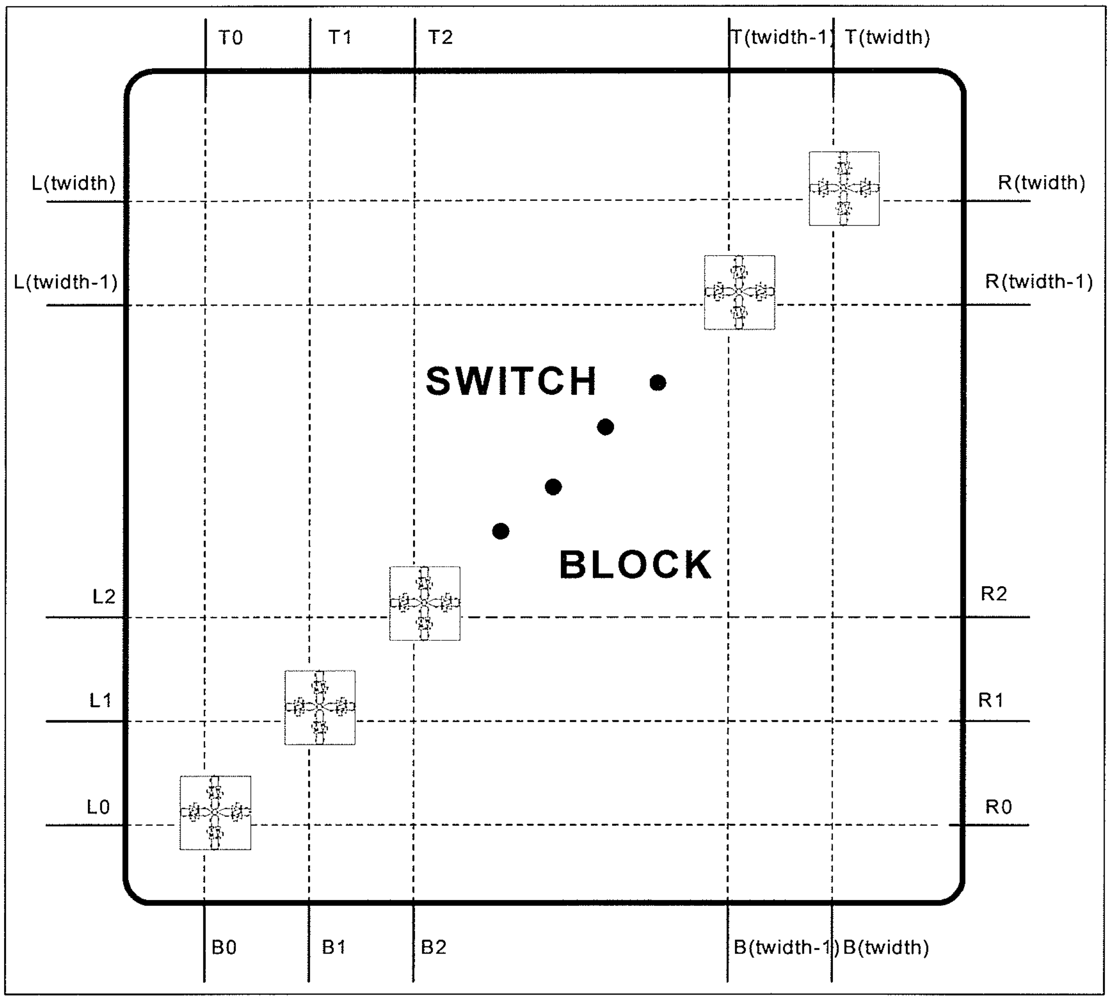

Figure 11 : Switch Block

The switch block was built as simple as possible in order to fit on the target FPGA, the Virtex. Switch blocks used in most practical FPGAs contain the ability to route over larger distances than simply from one switch block to the next nearest routing entity or CLB. A Xilinx 4000 series switch is similar to that designed in Figure 11, with the exception that wires may also run past 2 or 4 CLBs at a time [52]. By varying the distance over which a signal may travel before passing through a tri-state buffer or pass transistor, speed is increased.

According to Betz et al. [53] consideration should also be given to using a mix of tri-state and pass transistors for optimal speed and area requirements. Betz et al conclude that a mix of "50% - 83% pass-transistor switched wires..having a length of 4 or 8" with the rest of the routing being performed by tri-state buffers for shorter wires provides the best trade-off. For this project, only tri-state buffers were used as there is no VHDL statement to specify pass transistor instantiation.

Betz et al. [53] also analyze different switch block topologies. Each of the topologies involve switching from one track input over to a subset of other track outputs. For this project, if a signal entered on the left hand side on track 4, then it will leave on track 4 of the top, right, or bottom side. This is not a limitation since the track the signal initally travels on is determined by the horizontal connection block, c_block_x, after leaving a CLB.

The c-block-x and c-block-y routing units are shown in Figure 12. It should be noted that if a wire is labed with a "0", it is an output, if it is labelled with a "1", it is an input. For example, TC0 is an output, TC1 is an input. Each c_block_x connects TC0, TC1, and BC0 to horizontal tracks or wires. Each c_block_y connects LC0 and RC0 to vertical tracks or wires.

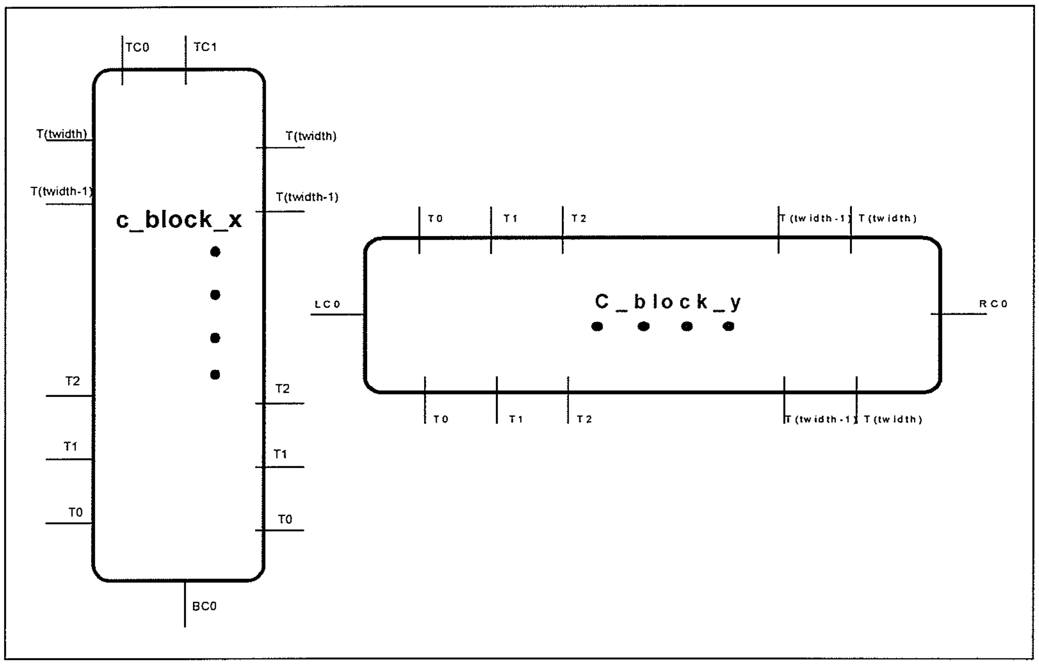

Figure 12 : c_block_x and c_block_y

The input/output (IO) units for the left, right, top and bottom sides of RFPGA connect all internal circuitry to the pins of the microchip. Track connections to each block also pass data through to the nearest switch block. This allows for some routing around a pin.

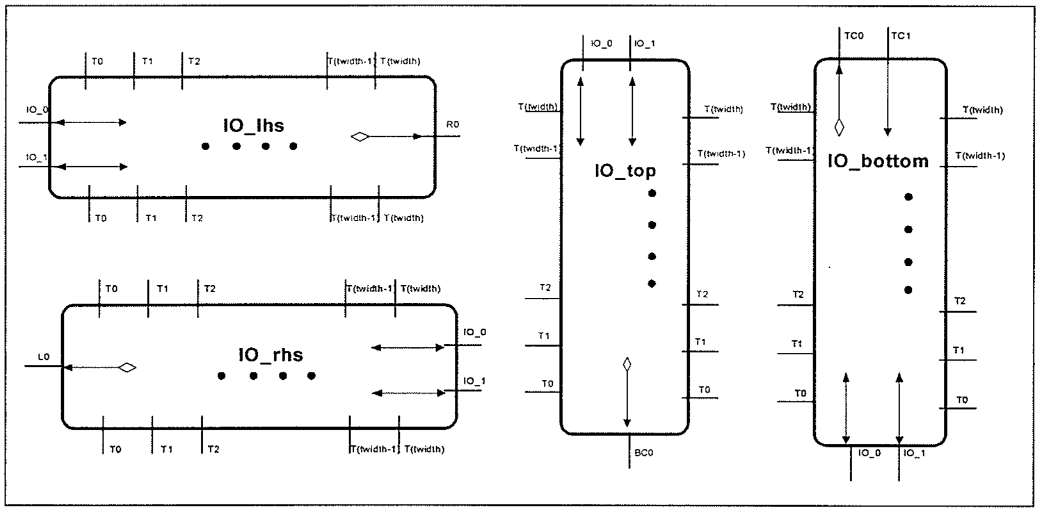

Figure 13 : Various input/output blocks

While RFPGAs BLE's were built with the scannable RB4 flip-flop, the remaining components had RB4 added later. The addition of an RB4 scan chain, and the order in which each RB4 appears is illustrated in each of the following component figures. Each of the component figures are necessary viewing when trying to map a hardware design onto RFPGA using the place and route tool "RLocate" developed for RFPGA. The arrow in each figure represents the direction data is clocked through the scan chain, and each number represents the local switch to be set in a chain segment. Each local switch is in fact an RB4 to be set on or off.

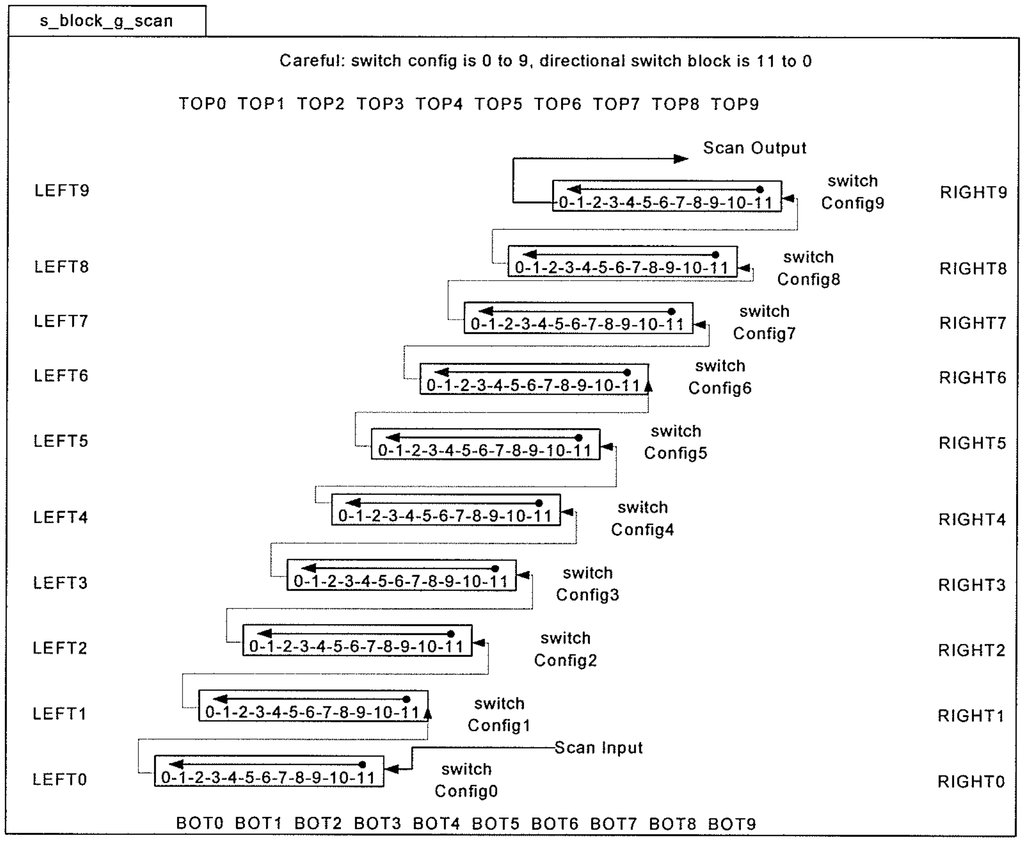

Figure 14 : Switch Block Scan Chain

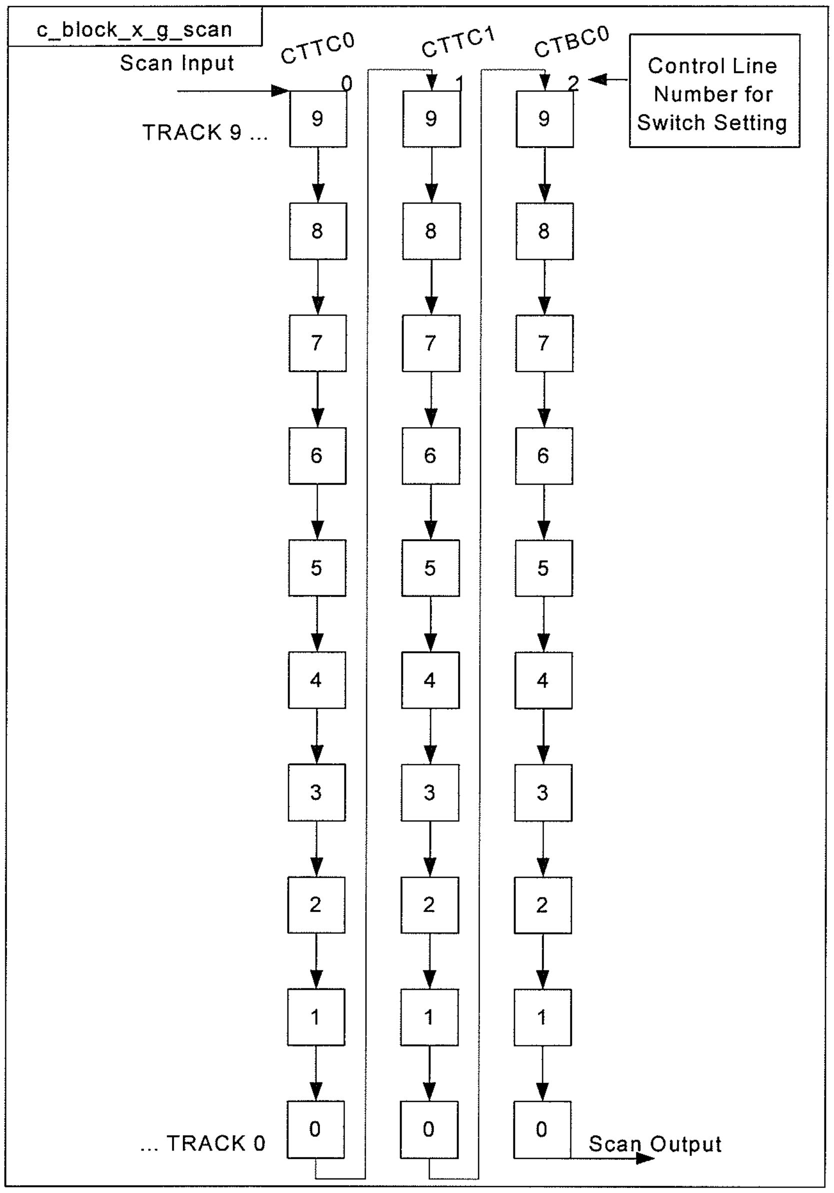

Figure 15 : C Block X Scan Chain

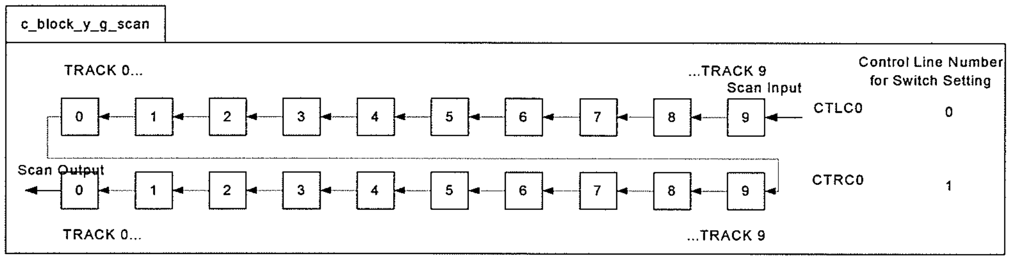

Figure 16 : C Block Y Scan Chain

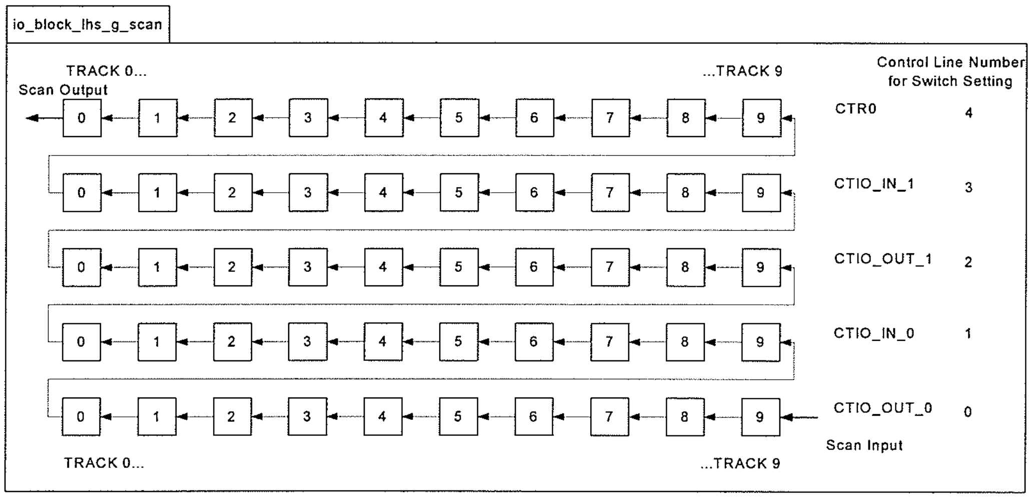

Figure 17 : IO Block, Left Hand Side Scan Chain

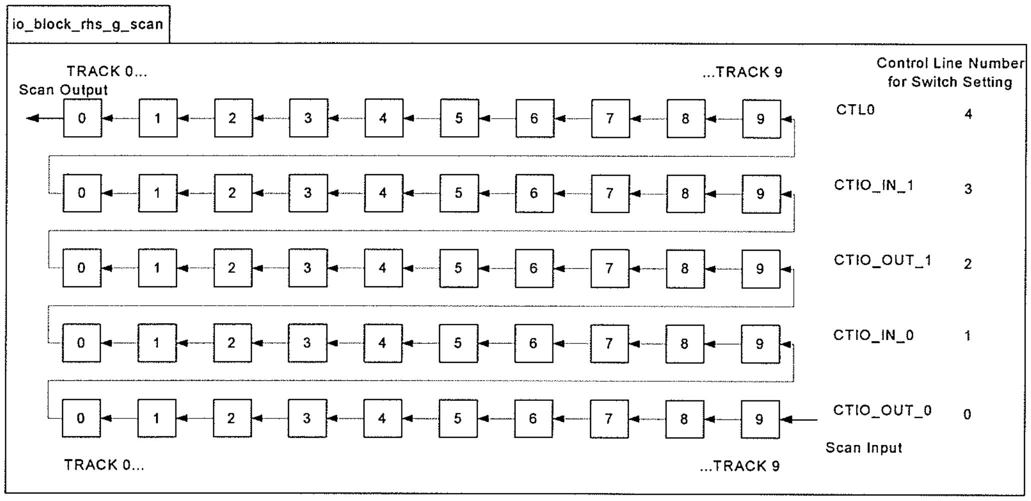

Figure 18 : IO Block Right Hand Side Scan Chain

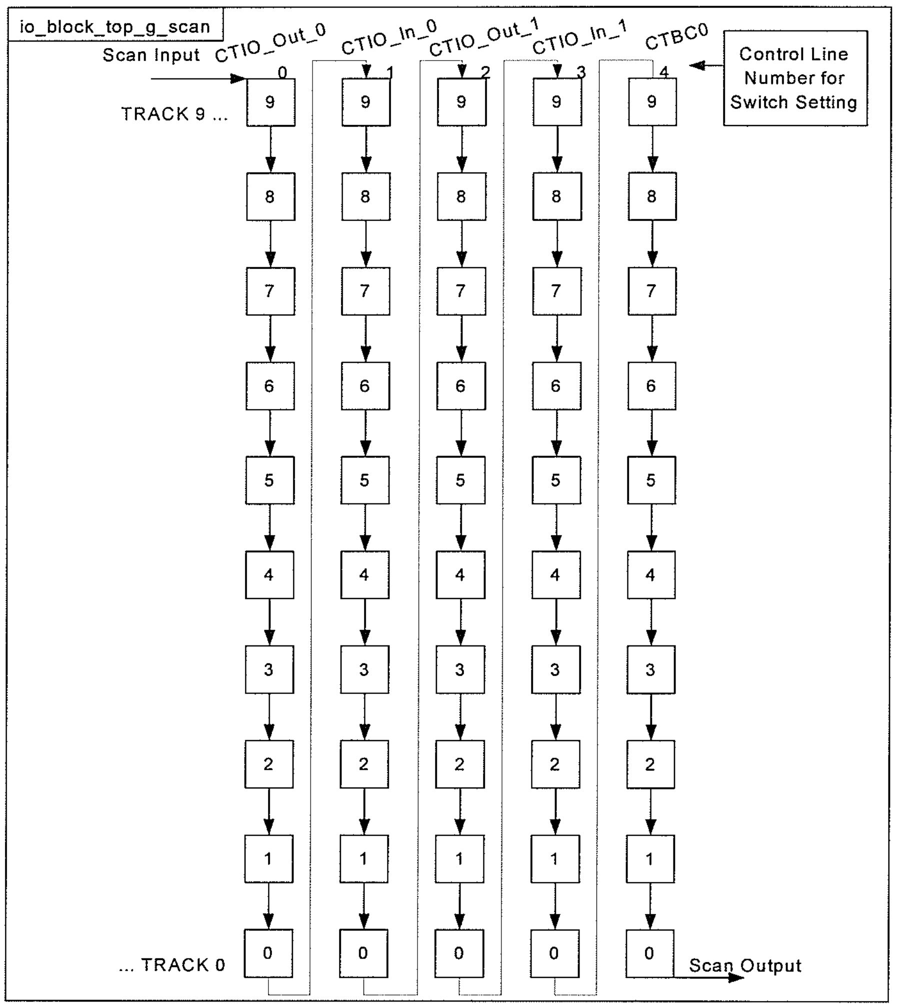

Figure 19 : IO Block Top Scan Chain

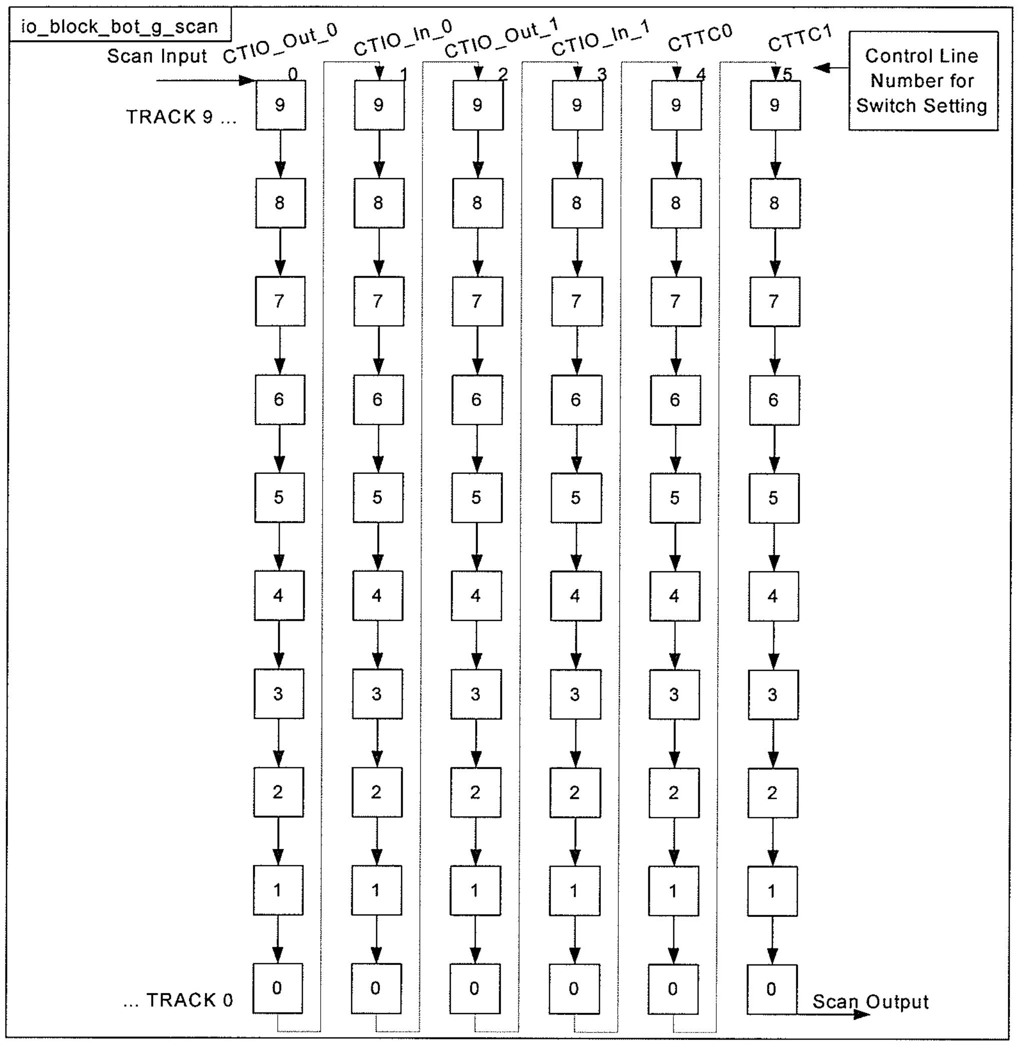

Figure 20 : IO Block Bottom Scan Chain

RFPGA assembly

Component Connectivity Example

When size=3, a 3x3 FPGA would be fabricated.

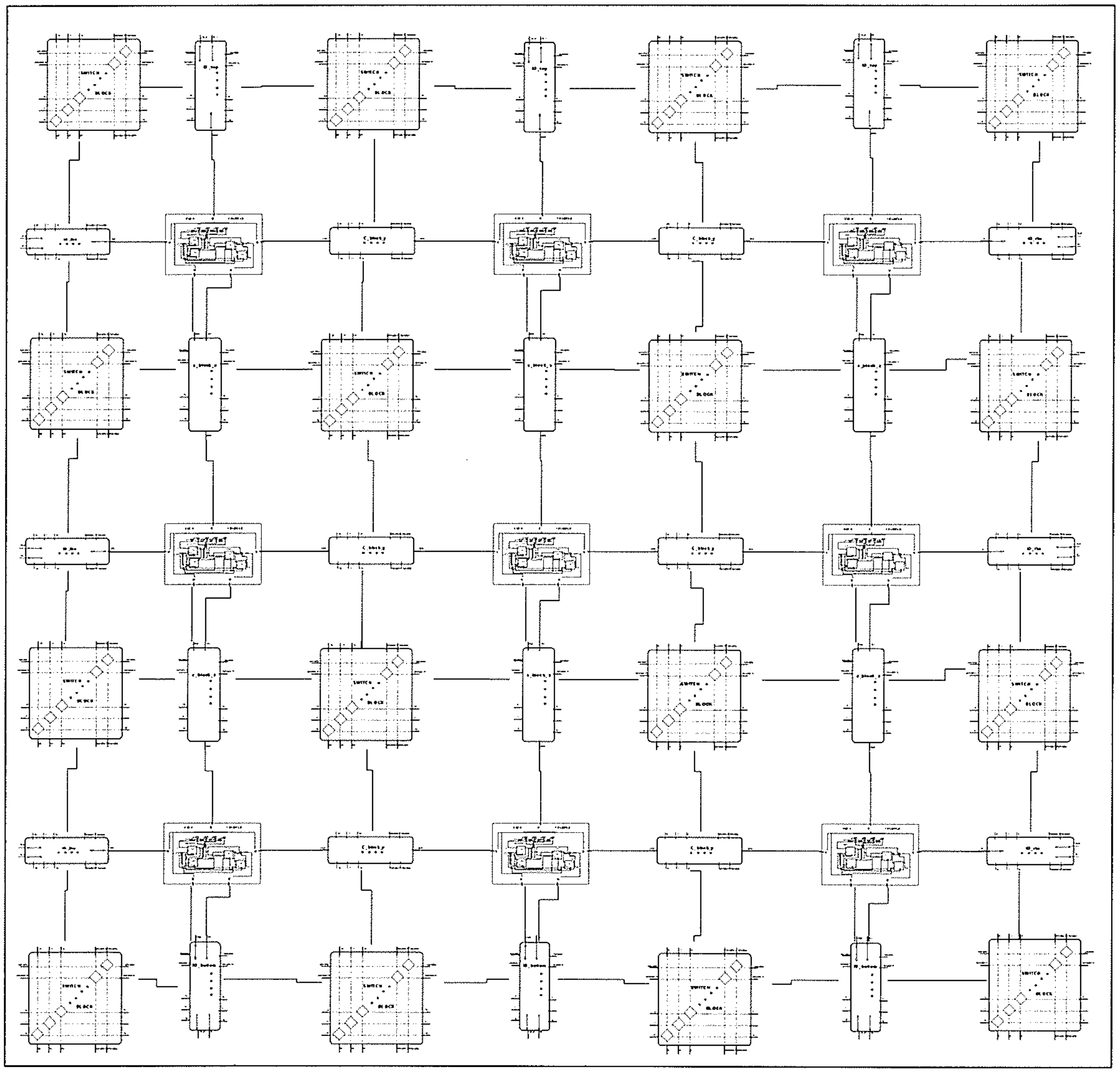

Figure 21 : Complete 3x3 FPGA

Since even the small 3x3 RFPGA of Figure 21 contains enough components to make it difficult to decipher on an 8 1/2 x 11 inch sheet of paper, I will describe the connectivity here. The majority of the squares are switch blocks and are the ones with the diamond on a diagonal. An example switch block can be found in any one of the four corners (minimum). On the outer rim of RFPGA are the rectangular IO blocks. The top, bottom, left, and right IO blocks are on the top, bottom, left and right of the figure respectively. "c_block_y" or "cby'"s for short are the thin horizontal connection blocks while the "c_block_x'"s or "cbx'"s for short are the thin vertical connection blocks. The remainder are the rectangular and slightly fatter CLB's which appear to have a complex design within themselves. This example RFPGA is termed a "3x3" RFPGA because it has 3 rows of CLB's and 3 columns of CLB's. To generate a 3x3 RFPGA, set the "size" variable to equal 3.

Connectivity out of each switch block is performed through a set of parallel wires termed "tracks". Therefore all lines emanating from a switch block are of a certain width specified by the variable "twidth" and are numbered from zero to twidth. All other lines represent single wires. The remaining detail is left to be deciphered from the source code comments or an explanation from this author.

Throughout all the various functional units of RFPGA, there are RB4s to set switches on or off to allow circuits to be created. That is, there is a complete scan chain in addition to what is visible in Figure 21 for RFPGA configuration and testing purposes. Supporting tools designed specifically for the RFPGA project are included to program RFPGA.

RFPGA support tools

Bitstream Generator

Following the creation of a VHDL application for RFPGA, one requires a bit stream to be scanned into the chip to make that application run. While the bitstream generator was not a core part of this Masters project, it is a necessary evil. Without the bitstream generator, an FPGA with a track width of 10 and size of three, there would be 3120 RB4's to set to a 1 or 0. To use the bitstream generator program, Rlocate, one must already have an idea as to what the final circuit is to look like. Then, with the help of a diagram of RFPGA, manually map their circuit to it. Circuit mapping is performed by selecting only the RB4's to be set to a logic '1'. Rlocate will then generate the final bitstream for download into RFPGA.

Bitstream Encryption and Decryption

Once the bitstream has been generated, it must be encrypted. This is because RFPGA is expecting to decrypt whatever is to be downloaded.

To support the encryption process there are two pieces of software. The first is a Java tool used to generate the public key information according to Chiou and Chen's broadcast encryption protocol [40]. This tool can be used to encrypt the secret key used by the Gunther ASG and represents the first part of RFPGAs downloadable datastream. The second tool is some VHDL code that can be used to generate the pseudo-random bitstream via the G-ASG and combine it with the hardware application bitstream for RFPGA to configure. The final two encrypted data streams are then combined and placed in memory for use by RFPGA when required.

Decryption is done in a three phase process. The first phase is to use RFPGAs private key to obtain the secret key. Once the secret key has been obtained, it is programmed into the Gunther ASG. At this point, the encrypted hardware design bitstream is ready to be read from memory and downloaded into RFPGA. At the same time this is happening the Gunther ASG's output is being

XOR'd with the bitstream and turned into its original format (unencrypted) for use on RFPGA.

Chapter 4 RFPGA Decryption Unit

Decryption Unit Requirements and Overview

Keeping in mind Kerckhoff's[18] ideals for a cryptographic system, it was decided that a hybrid public-key / secret key system be designed. If the system was solely based on public-key theory, then the decryption of the bitstream would be too slow for most applications on RFPGA. If the system was based totally on a secret key design, then it might limit RFPGAs applications. By going with a hybrid approach, the best of public key and secret key systems are utilized. That is, anyone can send an encrypted message to RFPGA easily while the speed of decryption will remain fast. To perform the decryption, the secret key is encrypted using Chiou and Chen's broadcast encryption scheme [40] encoded in Java. This allows the secret key to be sent as a broadcast message if necessary to a variety of end users. The secret key is then decrypted using the RSA scheme [32]. This RSA scheme however is augmented by the mathematical theory of Montgomery's quick modular multiplication algorithm [22]. The decrypted secret key is then passed to the stages of the Gunther ASG for use in decrypting the hardware design.

Public Key Design

There have been many public key cryptosystems designed as of today. Most of the literature found in an IEEE search will focus on improving the speed of the basic public key operation which is "a^e mod n", where a is the text (or ciphertext), e is the exponent for encryption (or decryption), and n is the modulus. The engineering research to date has been a variety of custom hardware designs that are designed once, and require another custom redesign for modifications. It has been said that "Hardware is called hardware not because it is hard to understand, but hard to change". Custom designs take this notion to the extreme. For this project, a slightly higher level of design is performed in VHDL. VHDL is a software programming language that allows for quick and easy modification of algorithms just like in a high-level software programming language such as Java or C. VHDL code can also be targeted to general purpose, reprogrammable FPGAs for the ultimate in hardware upgradeability, or MPGA's and ASICs allowing for simple price / performance / maintainability tradeoffs depending on the final application. Therefore, this open-source project will allow for quick improvements to hardware designs using current mathematical theory.

For RFPGA, the secret session key is encrypted using Chen and Chiou's algorithm previously mentioned. This secret key is also used to encrypt the downloadable hardware design. This is how the hardware design's intellectual property is protected. Once the encrypted hardware design has been downloaded into RFPGA, RFPGA's decryption units decrypt and configure each design. Another primary objective for providing a public key system was to learn about behavioural hardware description in VHDL. Prior to designing the public key system, all hardware was encoded using a structural technique. Structural techniques are used to specify and connect specific components together, much like drawing a circuit's wires, multiplexers and flip-flops out by hand on paper. The behavioural technique however is akin to a high-level programming language like Java or C which only requires a description of what is to happen. Since the specification of basic digital components is minimized or non-existent, the synthesis software is required to do more work in generating the final digital circuit. This extra effort however results in some trade-offs for the hardware designer such as:

1) Quick method for rapid application development.
2) Easy to understand, and therefore lower software maintenance costs [47].
3) Potentially reduced circuit speed due to an inefficient synthesis.
4) Potentially massive area requirements for a given behavioural description.
5) A lack of visible control over the exact circuit implementation.

The first two reasons for producing behavioural as opposed to structural source code may be so overwhelmingly positive for most companies that they may choose to always write behavioural code. In fact, Fjelstad and Hamlen [47] reported that approximately half of all the time spent maintaining code was actually spent simply trying to understand what had been previously written. It is therefore imperative that in order to reduce maintenance costs, one should write easily, humanly readable code. Generally speaking, a behavioural description of what is to happen is easier to understand, than adding basic components and connecting them with wires. In fact, structural code is most dislike pseudo-code, a common, easy to comprehend, method for documenting existing code functions.

In terms of the ease of maintaining a digital design, I propose that the higher-level the design, the easier it is to maintain. The following list starts from low-level design techniques and proceeds to the high-level methods:

1) Custom design involving the drawing layout of individual polygons. representing different layers and materials of the manufacturing process.
2) Standard Cell design involving the layout of known components.
3) Structural HDL involving the software layout of known components.
4) Behavioural HDL involving the description of a hardware's function.

There is however one problem with behavioural design that could be eliminated with a change to the VHDL specification.

The initial design of the public key decryption unit was totally behavioural. It was also theoretically possible, but not practical, to synthesize. Unfortunately, VHDL has a tendency to synthesize the described circuitry in parallel. VHDL also requires that the bounds of every "for" or "while" loop be well defined. In many cases, this is a blessing, and produces fast circuitry, however it is area inefficient. VHDL allows little room to produce albeit slower, but area efficient behavioural code involving loops. For completeness sake, there is a simple way to produce sequential code through the use of variable assignments, but this has its limitations. What is required is a modification of the VHDL specification to produce area efficient behavioural code which is *easy to maintain*. To give an example as to the extent of the problem, I will first explain what happened for the first public-key HDL code I wrote.

Sample Code:

```
for x = 1 to 1000
  for y = 1 to 1000
    a=x+y
    procedure MathSubroutine(a,b,c,x,y)
  endloop y
endloop x
```

In the sample code, the loops of x and y are static which is what VHDL requires to synthesize a circuit. Since the code generates parallel circuitry, the two statements within the nested loop are synthesized (generated physically) 1000*1000 times. Furthermore, MathFunction (lets say) includes one or two for-loops. One may also wish to base these for-loops on x and y; so that the loop ranges are to be dynamic. Some VHDL synthesizers will reject these dynamic ranges, others will produce circuitry for the entire possible range of a dynamic loop. Either way, one can easily see how a chip's area can quickly become consumed. In the public-key crypto routine I created, many MathFunctions are required within many loops. The nested loops were impractical for the VHDL synthesizer to produce a usable circuit.

To solve this problem, the code had to be completely rewritten, which brings us a step closer to a solution to VHDL's automatic parallelization problem.

It is possible in VHDL to create simple counters, which when coupled with 'if / then / else' statements, will mimic the function of a for or while-loop. In doing so, VHDL creates only the counter and associated "if" condition circuitry, but it does not generate all possible solutions in parallel. To do this, a state machine is required and the "if" statements are used to determine which state is to be executed next. Furthermore, one other technique had to be employed to reduce the circuitry generated by VHDL. That is, the procedures had to be turned into entities. This is because the procedures themselves call other procedures with for-loops which all need to be broken down into state machines with counters and "if/then/else" statements. With the creation of entities, a technique for reusing the same entity (which would only be generated once) was employed. This was done by creating basic-language-like 'gosub' routines. This is illustrated in the following code and associated comments:

```
when Current_State=Initialize
        x:=0;
        Current_State=Start                      // go to next state after clock tick
when Current_State=Start
        x:=x+1                                   //add 1 to x counter as for-loop would
do
        if (x<1001) then                         //check for-loop limits
                Current_State=Xloop1             //if positive, continue Xloop1
        else
                Current_State=After_Xloop        //if negative, quit this for-loop
        end if

when Current_State=Xloop1
        a=x+y                                    //inside Xloop1, execute this code
        Current_State=MathSubroutine             //call a subroutine entity
        Return_State=Xloop2                      //remember where to return control to

when Current_State=Xloop2                        //where control returns to after sub
        ...do some things...
        Current_State=Start                      //continue the for loop

when Current_State=After_Xloop                   //we've reached the end of the code
        ...do some things....
        ...end the program...

// "Subroutines" ****************************

when Current_State=MathSubroutine                //Math subroutine code********
        Input_1 = a
        Input_2 = b ....etc...
        Input_Control = Run_Math_Function_Now    //tell Math subroutine to execute
        Current_State=Hold_Until_Done

when Current_State=Hold_Until_Done               //wait until Math sub is finished
        if Output_MathSubroutine=Done
                Current_State=Return_State       //if Math done, return to main code
        else
                Current_State=Hold_Until_Done    //if Math unfinished, stay here longer
        end if
```

The actual code is only slightly more involved than the pseudo-code above. It, however, powerfully illustrates the following points:

1) Subroutines do not need to be synthesized for each case of a for-loop
2) Chip area requirements are significantly less than the VHDL-natural parallel implementation
3) Counters allow for dynamic loop assignment
4) The translation from parallel VHDL code to versatile sequential VHDL code is simple for a computer to do.
5) The translation from parallel to sequential is currently more involved than it should be for a programmer to do.

The problem and solution should now be obvious. If a programmer doesn't require speed as a top priority, there should be a simple means within VHDL to implement chip-area efficient circuitry that is sequential in nature and behavioural as well. When I moved from the original code, to the sequential code, I had to structurally add in the math subroutines. A low level feat that a behavioural programmer may shun. Each step I took to resolve the problem can be automated, and give the full look and feel to the designer of behavioural programming.

I propose the following enhancement to the VHDL language as a result of my work. To implement sequential behavioural code, one should have the option of sequential for and while-loop statements. These sequential statements would also have the advantage of dynamic loop limit assignment as described previously. To accomplish this, one could create two new statements in VHDL. Simply by concatentating the letter "s" in front of the VHDL reserved words: "for", "while", and "procedure" would direct the synthesizer to produce sequential code as above. In doing so, the code would be far easier to interpret by those expected to maintain the hardware. This would result in significant cost reductions for potentially numerous applications. To understand the exact implementation details, I refer the reader to both the pre and post sequential public-key source code written as part of this thesis, and available from the author.

Secret Key Design

The purpose of the public key unit is to decrypt a secret session key. This secret session key is then implanted within the Gunther Alternating Step Generator (G-ASG). Gunther's ASG has been proven to be both mathematically secure and to have stood the test of time. To this date, this author has not read of a suitable attack to break the Gunther ASG. The author himself has only found one attack, "a correlation attack ...[that] ... does not substantially reduce the effort to break the system." [48] Gunther's ASG as he published it, is shown in Figure 22.

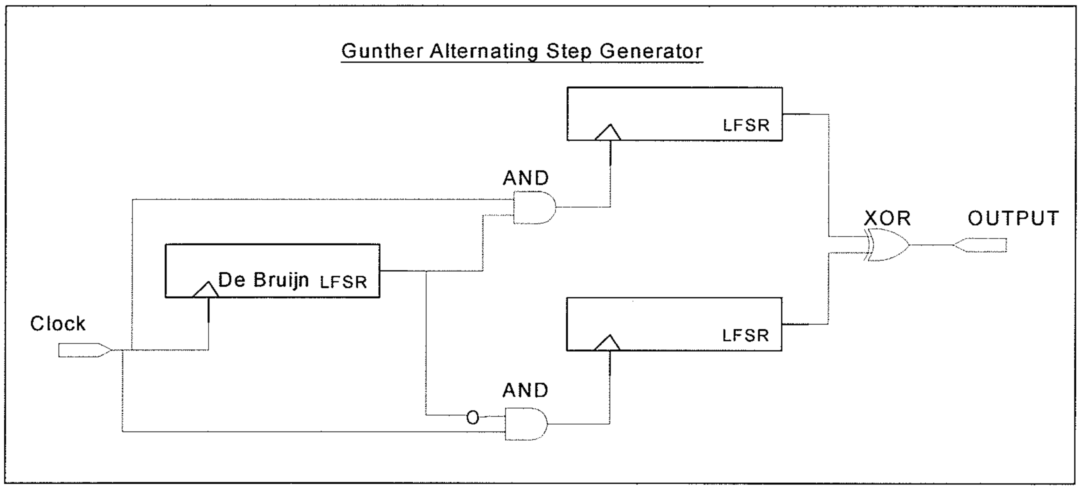

Figure 22 : Gunther's original Alternating Step Generator

For this thesis, there were two design changes from other Gunther designs. Prior to discussing the design change I will review Kerckhoff's [18] first two postulates from the literature review section, namely:

1. The system must be practically, if not mathematically, undecipherable.
2. It must not be required to be secret, and it must be able to fall into the hands of the enemy without inconvenience.

While Gunther's design has been shown time and again to be undecipherable, the modification I make to the G-ASG slightly stretches the limits of postulate #2. Postulate #2 states that you should be able to provide the encrypting and deciphering schemes to others, and no one without the key should be able to break it. For my modified G-ASG, the traditionally fixed feedback polynomials of the shift registers are now programmable and thus part of the key itself. This has been made possible through the use of FPGA SRAM technology and the fact the secret key can be easily transported and programmed into the G-ASG by use of public-key crypto techniques. All the mathematics proven by Gunther and others still apply. There is simply more information kept securely unknown from a potential attacker, and Kerckhoff's principles are maintained.

While the mathematics of the programmable LFSR remains sound, and the flexibility increases, there is one tradeoff, area. The area requirements for a programmable LFSR are approximately twice that of the non-programmable kind. This is because for each feedback loop there must be an associated RB3. The feedback RB3 is used to determine whether or not the current state of an individual LFSR register is to be multiplied with the LFSR output state. Given today's fabrication technology, the extra area required should be a minor issue, and even less so in the future.

In addition to designing a more flexible G-ASG, the hardware design is considered more practical as the shift register's clock inputs do not have any asynchronous logic before them. If a clock is gated, errors may result (especially at increasing clock speeds) with changes in temperature and fabrication process. The solution to this is accomplished by adding multiplexers before each RB3. When new data is to be shifted in, the ssFPGAs controller module selects the input line instead of the retime line on the De Bruijn LFSR as can be seen in Figure 23.

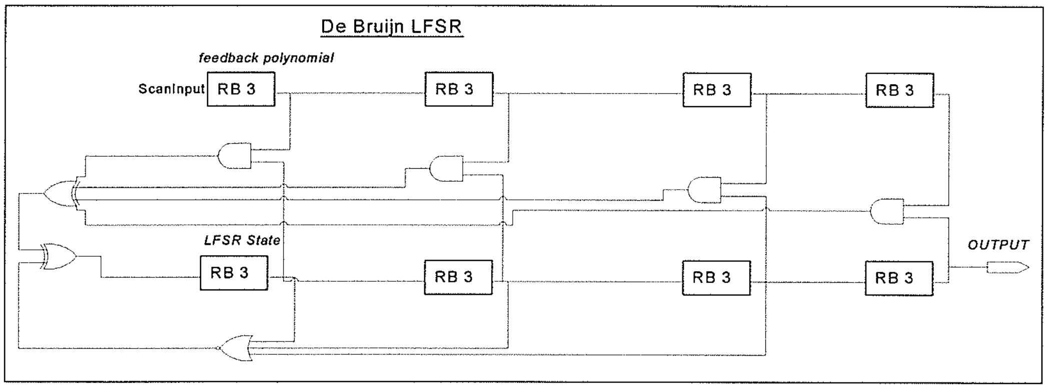

Figure 23 : Programmable De Bruijn LFSR

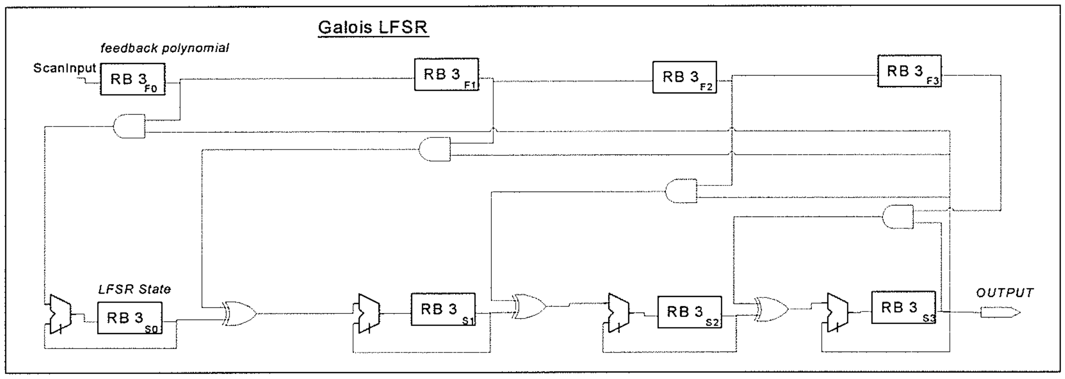

Figure 24 : Programmable Galois LFSR

The multiplexer's select line of the two instantiated Galois LFSRs in Figure 24 is connected to the output of the De Bruijn LFSR through an AND gate like in Figure 22. At a higher level, all LFSR's are ultimately managed by a controller entity. The purpose of which will be clarified later. The clock for the De Bruijn and Galois LFSR's enters directly into each scannable flip-flop, RB3 - as a proper design should, with no gates between itself and the source. The final design of the Gunther Alternating Step Generator is shown in Figure 25.

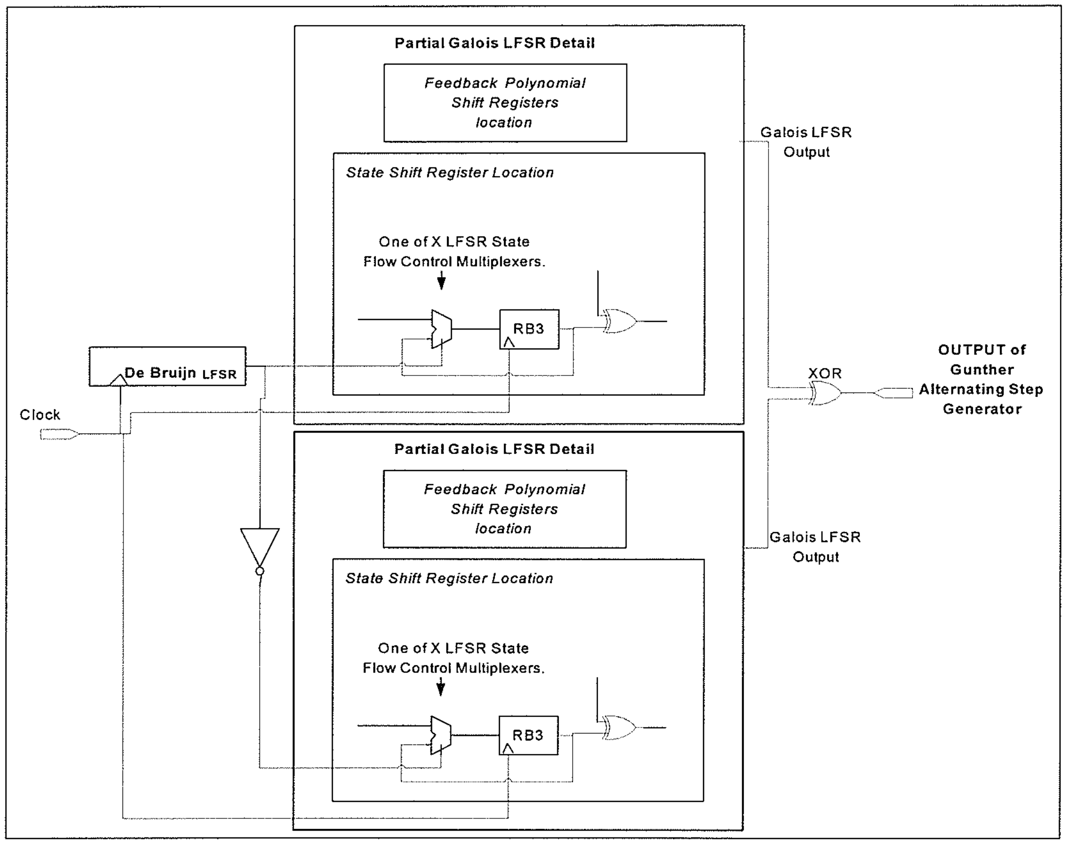
Partial Galois LFSR Detail
Feedback Polynomial Shift Registers location
State Shift Register Location
One of X LFSR State Flow Control Multiplexers.
RB3
Galois LFSR Output
De Bruijn LFSR
Clock
XOR
OUTPUT of Gunther Alternating Step Generator
Partial Galois LFSR Detail
Feedback Polynomial Shift Registers location
State Shift Register Location
One of X LFSR State Flow Control Multiplexers.
RB3
Galois LFSR Output

Figure 25 : Designed Gunther Alternating Step Generator

LFSR – Linear Feedback Shift Register

Chapter 5 RFPGA Controller Unit

Controller Requirements

The controller entity is required to manage the resources of RFPGA and the decryption units. The entire project termed ssFPGA, has three distinct entities requiring management control, namely:

1) public-key decryption unit,
2) Gunther Alternating Step Generator secret-key decryption unit,
3) RFPGA.

Within RFPGA the following tasks must be managed by the controller:

1) reset conditions,
2) detect a request for a new hardware design,
3) download a new hardware design,
4) save the state of an active hardware design,
5) reconfiguration to the new hardware design.

To accomplish these tasks, three state machines exist. One to reset ssFPGA, one to load and run new designs, and the other to monitor the run-time requests of RFPGA.

In order to facilitate the communication between the controller entity and the public-key entity, a handshaking protocol had to be developed. This is because the public-key entity was designed using a behavioural model, and has the ability to complete its computations early depending upon the complexity of the computation. While knowledge was learned from Gutberlet and Rosenstiel's [49] work, the final handshaking routine is my own.

Controller Design

While the controller has numerous states of detailed operation, only the high level workings will be explained here. For a detailed explanation, the reader is referred to the actual source code, "controller.vhd".

The controller can be in a reset mode, idle, or be initiating a run or load command.

One of the controller's first responsibilities is to initialize the public-key's fast modular multiplier's constants. These constants are used in the Montgomery Multiplication process described in the literature review section of this document. The constants are the private decryption key, the modulus, and the precalculated inverse of the modulus which are stored in variables PrivateKey, Modulus, and PreCalculatedNpri respectively. Each of these constants were calculated using a Java program that implements the broadcast encryption [40] algorithm.

The next step is to load the first hardware design of the first memory bank. RFPGA was designed such that there are 2^4 possible memory banks, each of which stores one hardware design. Within a memory bank, data can be accessed by a 16 bit vector to grab a block of 4 bits of information. Note, that all values are easily modifiable given the design is in the high-level programming language, VHDL.

The first blocks of data to be loaded are for the public-key decryption unit. The results of the public-key unit are simply the secret key for the Gunther Alternating Step Generator (G-ASG). The secret key bits are first decrypted and then they are clocked into the scan chain of the G-ASG. Once the G-ASG is programmed for the design to be downloaded, the reset finite state machine (FSM) checks to see if it can fit more hardware designs into RFPGAs scan chain. If for example the width of the standby registers is 2, then 2 hardware designs will be downloaded. Furthermore, those two designs will be the first two designs in the memory made accessible to the controller. Once the reset FSM has filled all of the standby registers it runs the first hardware design and enters an idle mode called "LastState".

Once the reset FSM is in LastState, it is possible for the downloaded hardware design to issue commands to the controller module. The commands come from an output port on RFPGA and can be:

1) "00" Idle,
2) "01" Run or
3) "10" Load.

These commands are termed "port commands". The initial design of RFPGA allowed for commands to also come from within the application itself as opposed to using external pins, however, this would have made testing more difficult. The change however does not in any way reduce the capability of the existing design.

Given RFPGA was designed as a general purpose chip capable of storing a subset of designs from RAM for later use, there had to be a way to determine which design uses what standby register. The question is important because fixed designs are not capable of knowing dynamically exactly where in the standby registers the next-to-be-activated hardware design resides. To solve this probelm two different options were possible.

1) A request could be made as to where the location of application ID "XYZ123" resides,
2) A fixed location for each design.

Option 2) was selected for simplicity and to ensure the hardware would reconfigure in just one clock cycle. Furthermore, option 1) would have resulted in an elaborate scheme to save state information within another design.

Thus, the commands listed previously are actually augmented with an application number, which is also the memory bank of the stored hardware design, and whether or not current state information is to used by that application. Therefore a full command from the ports of an application to the controller unit would look like (for example):

01|101|1

The first number, "01", is the Run command. The next number "101" means run Application #5. The last digit, if a '1' means to use the current application's state information otherwise, simply reconfigure and run application #5.

Chapter 6 Results Discussion

Results

Reconfigurable Blocks

RFPGA was originally designed with only RB3, (or "reconfigurable block, 3rd version"). This initial implementation resulted in the possibility of short circuits being created on busses prior to a reconfiguration attempt.

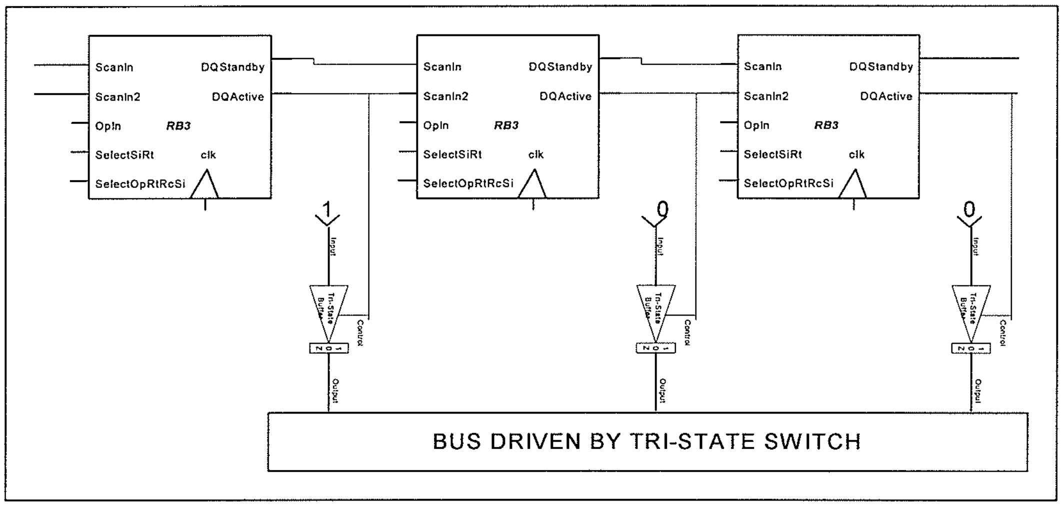

Figure 26 : RB3 and Bus Short Circuiting Problem

Figure 26 shows that the tri-state switch may take on one of three values, 1,0, or 'Z' which represents a high-impedance state. If one tri-state is outputting a logic "0" or 0 volts, and the other a logic '1' or some positive voltage, then a short circuit on the bus may result. This near-sighted problem was rectified with the advent of RB4, a scannable flip-flop that's capable of keeping the tri-states at a constant value, prior to reconfiguration. The RFPGA was then programmed to interpret a control value of logic '0' from RB4 to mean a high-impedance state on the tri-state switch.

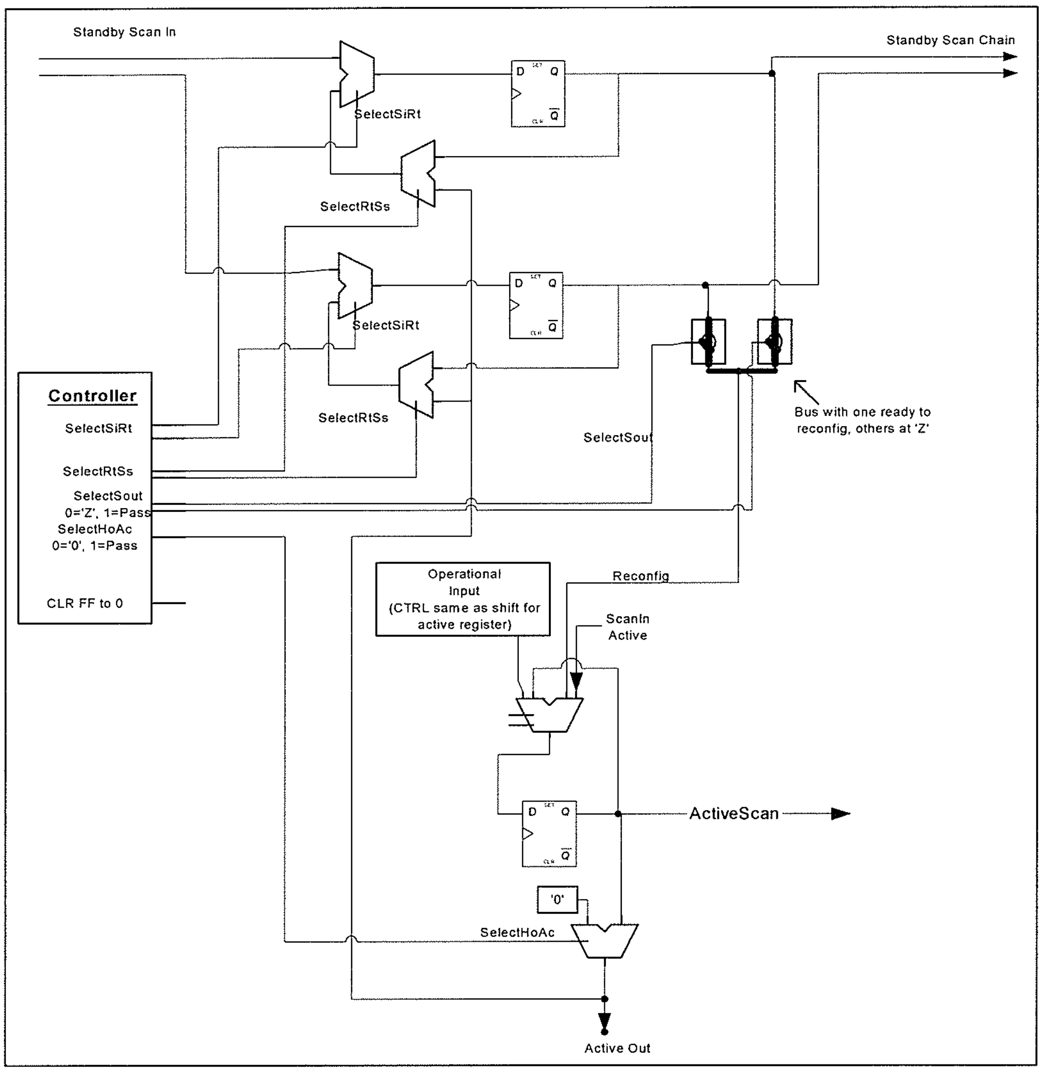

Figure 27 : RB4, explanation of "SelectHoAc" Hold/Active Multiplexer Purpose

This is accomplished by simply placing a multiplexer before the "Active Out" stage as can be seen in the lower portion of Figure 27. Prior to RFPGAs initial reconfiguration, the output multiplexer is set to zero. All tri-state switches connected to the multiplexer's output will output a high-impedance state, or 'Z' during this time. This high impedance state on the bus is what prevents short circuits during changes to values stored in RB4.

Another deficiency of RB3 was that it was limited to one standby register. This decreases the flexibility of a VHDL design, and other potential applications. As the number of standby registers increases, so does the number of potential hardware designs one may switch to. In order to activate one hardware design over another from the standby registers, a bus system, as seen in the middle of Figure 27, was created. Using the SelectStby (or "Select Standby") array, one may choose a standby register's output to impress upon the "Reconfig" input of the multiplexer controlling the active D-flip-flop. All tests showed this to be successful.

In order to make RFPGA more suitable for real-time computing applications, some state information from previous calculations had to be stored. This was accomplished by adding a second retiming multiplexer to the standby registers which is controlled by the SelectRtSs() line. By allowing one of the inputs to be from the Active Out flip-flop, it is possible to feed current state information into one of the standby chains. This results in a potentially more application-rich FPGA design.

Tests were also performed to determine the maximum frequency of a given reconfigurable logic block (RB). Figure 28 illustrates the changes in RB clock speed based on the type of RB and the number of standby registers per RB.

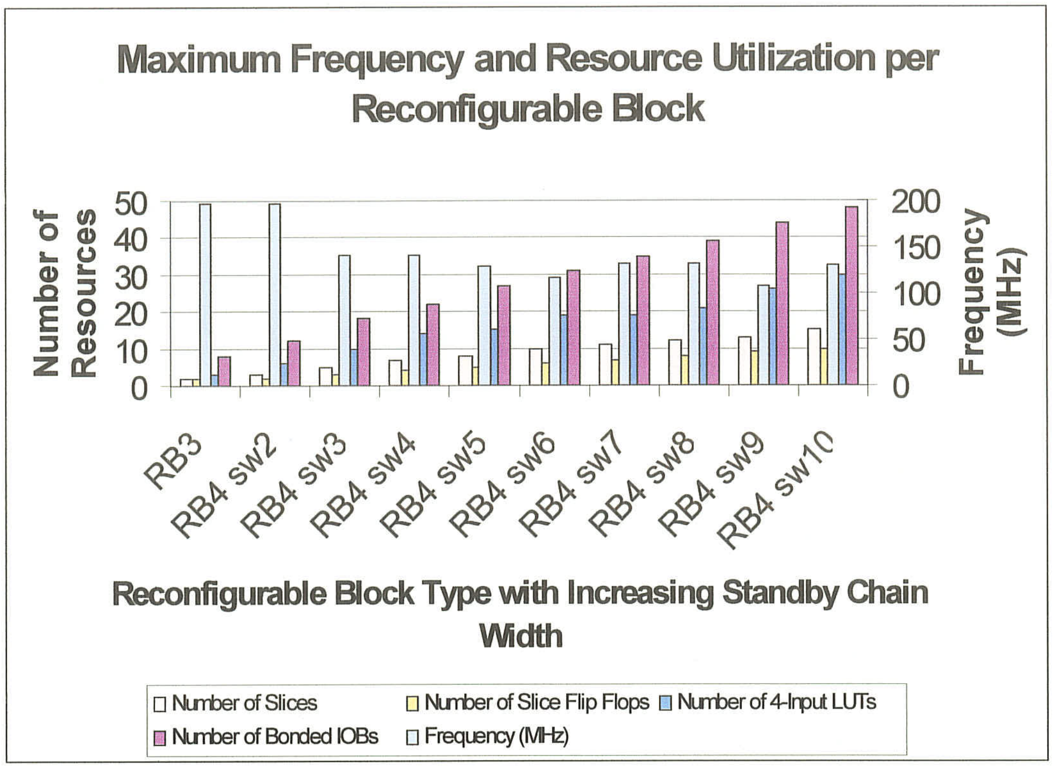

Figure 28 : Reconfigurable Block Clock Speed Comparisons

Figure 28 shows that as the number of standby registers increases, the speed of RB operation decreases. This is normal for two reasons. The first is that we are targeting the Virtex FPGA and any larger circuit must connect to other Virtex CLB's termed "slices". As the number of slices increase, so do the number of pass transistors a signal must traverse which ultimately decreases the speed of circuit operation. The second is that RB4 itself includes a bus between the standby registers and the active register. A bus may also decrease the speed of operation, especially since the driving component's size (and therefore drive power) remain constant. Furthermore, a tri-state switch and bus system are not always efficiently targeted on an FPGA resulting in many multiplexers to simulate the system.

It should of course be noted that both blocks RB3 and RB4 are fully capable of reconfiguring from one hardware design to the next in just one clock cycle. Tests were performed at the initial design stage of RB3 and RB4 as well as with the entire system using ModelSim version 5.6a.

Public Key Decryption unit

The original behavioural description of the public-key algorithm using Montgomery Multiplication was not synthesizable due to its size. The original description resulted in a massive amount of parallel circuitry being created which the synthesis software could not process.

The revised code using a versatile sequential programming process was synthesizable using a reduced maximum decryption block data size (vbits) equal to 24 instead of the usual 148 bit width. It had the following summary results:

```
Number of Slices:                    2429  out of   9408    25%
Number of Slice Flip Flops:          2546  out of  18816    13%
Number of 4 input LUTs:              4504  out of  18816    23%
```

The maximum frequency of the public key unit was calculated by the synthesis software at: `43.995MHz,` for the Virtex FPGA I had.

All simulations of the new and original public key code performed flawlessly.

ModelSim Simulation Results

ModelSim version 5.6a was the primary vehicle for producing simulation results. The first result worth observing is the heart of RFPGA, RB4. This simulation demonstrates RB4's ability to be used as a scannable flip-flop, reconfigure in one clock cycle, and to pass information from an active register back into a standby register. The key control lines for managing this functionality are SelectSiRt, SelectRtSs, SelectSout, SelectHoAc, SelectOpRtRcSi, each of which were previously described in the *"RFPGA Components"* section of this document. The results of this simulation are shown in Figure 29 : RB4 Simulation Results.

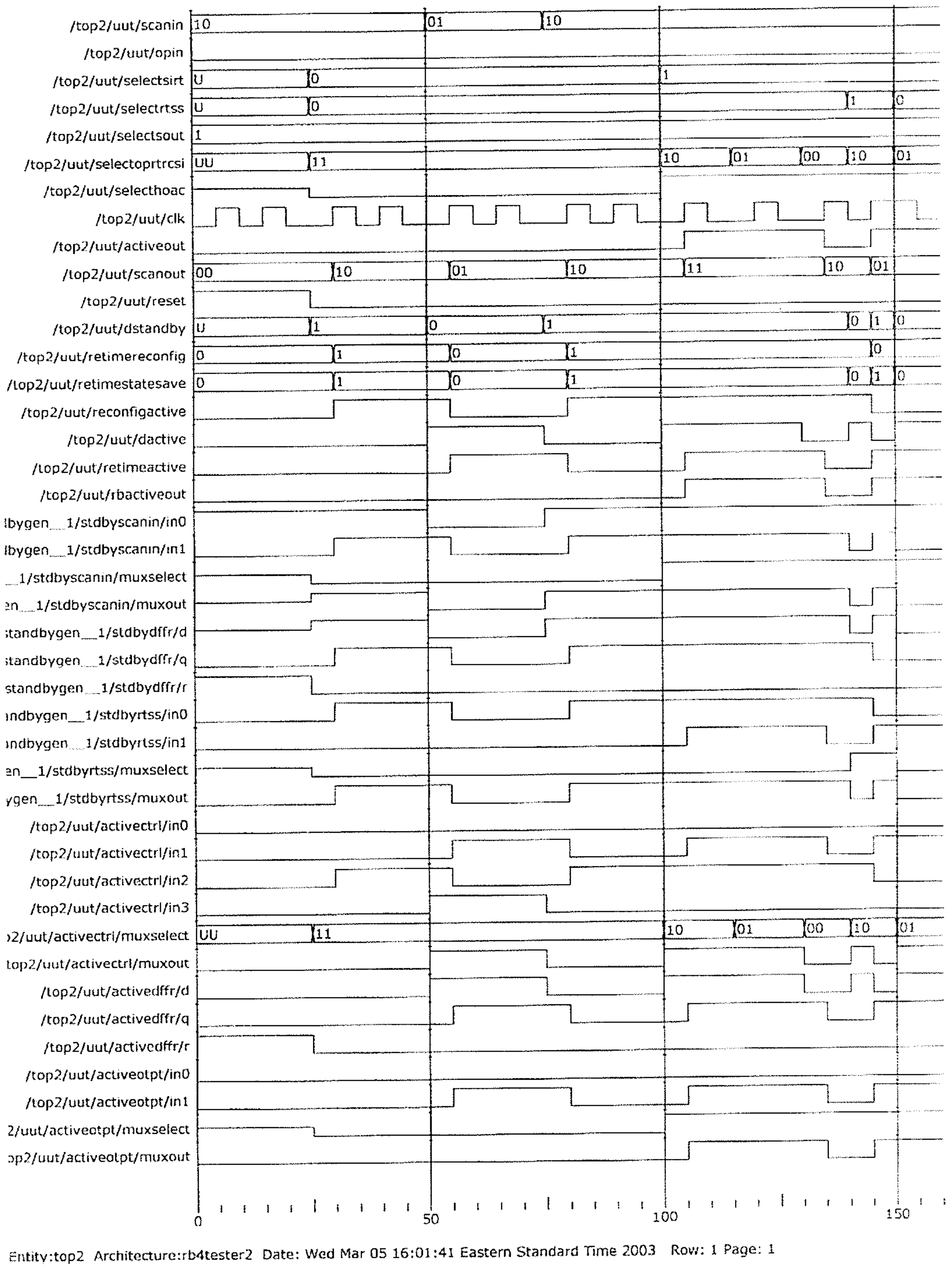

Figure 29 : RB4 Simulation Results

Observe that in Figure 29 through port input "scanin", a '1' is scanned into the standby register, and a '0' into the active register at the start of the simulation. At the rising edge of the clock just after "SelectOpRtRcSi" goes from "Si" to "Rc", the active D-FF transitions from a '0' to a '1' demonstrating single clock cycle reconfiguration as can be seen from the "ActiveOut" and "ScanOut" signals. The data is then retimed by selecting "Rt" on the "SelectOpRtRcSi" line before selecting the operational input or "Op". Since the input to the "Op" port, OpIn, was set to '0', this data then flows through the active flip-flop during the next clock cycle, and hence "ActiveOut" becomes zero. At the next clock cycle, the '1' in the standby register is reconfigured back into the active D flip-flop, while at the same time, the '0' in the active, goes into the standby register. This is most easily seen by looking at the "ScanOut" vector which flips from "10", to "01". By performing this operation, the simulation proves that RB4 is capable of saving active flip-flop memory into a standby register while reconfiguring to another state. This is the feature that allows data to be shared from one configuration to the next, and is in fact the "soft-reconfiguration" described in the first chapter of this thesis.

In Figure 30, one can see an application running on RFPGA. Since the synthesis tools used were home-made and primitive, only a simple AND gate was targetted to RFPGA. The first input is the first subheading under i_chanx, and the second input is also the last subheading but under i_chany. The output is the first subheading under i_chany.

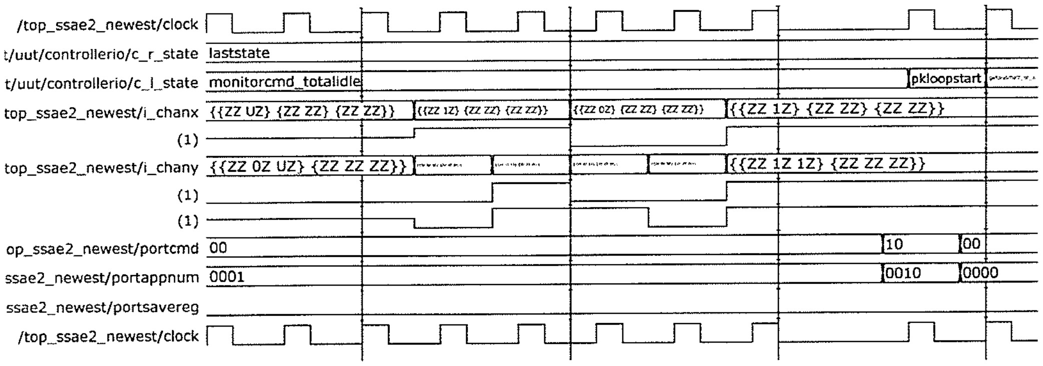

Figure 30 : Run of AND Gate on RFPGA with Port Command

Following the AND gate test, an external port command was applied to the controller. This command was issued through the three ports named PortCmd, PortAppNum, and PortSaveReg. The port command encoded for "PortCmd" as '01', orders RFPGA to run a new hardware design. This command may only last for one clock cycle before returning to an idle state such as '00'. The port application number or the "PortAppNum" signal was set to the value 2, or '10' in binary. This tells the controller to run the second application into the standby registers. If the controller was told to load another application, then the formula for determining which standby scan chain to map the hardware design into is given by:

```
ScanChainNumOp := ( (CONV_INTEGER(UNSIGNED( PortAppNum ))-1)  mod (ScanWidth-1)) + 1;
```

The results of the second hardware design are shown in Figure 31.

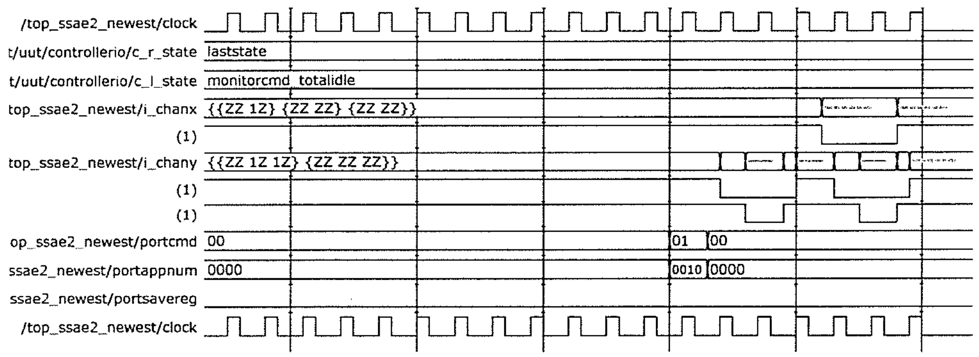

Figure 31 : Run of AND Gate with Synchronous Output (through RB4)

The second application is the same as the first with the exception that the AND gate's output is run through the scannable flip-flop RB4. These two simulations demonstrate RFPGAs ability support both asynchronous and synchronous applications while supporting single clock cycle reconfiguration.

Chapter 7 Conclusions and Future Work

Conclusions

RFPGA has been successfully designed to provide reconfiguration of its function in just one clock cycle. This will allow future designers to run more functionality on one chip with RAM, than any single ASIC could manage. RFPGA's quick reconfiguration capability may find uses in supercomputing, and custom computing applications requiring the cost effective use of microchips, with a high degree of flexibility.

The entire FPGA system, "ssFPGA" (entity "ssfpga") has been shown in ModelSim to successfully download, decrypt, load, and run hardware designs into RFPGA. The decryption of the hardware designs was accomplished using a hybrid scheme of both asymmetric and symmetric key cryptography so as to adhere to Kerckhoff's principles. The asymmetric key cryptography is based on Chiou and Chen's broadcast encryption methodology [40], RSA [32,33], and Montgomery's modular multiplication method [22]. The symmetric key cryptography was based on Gunther's Alternating Step Generator [48] with some enhancements.

The final reconfigurable logic block (RB4), may be generated with as many standby registers as is required or is feasible. It also allows for one to save an existing state, in a standby register, for later use by another. In doing so, information can be transferred to other designs, or stored on a stack-like interface allowing for a soft reconfiguration from one design to the next. This allows the hardware design for each function to be written in behavioural VHDL and to be targeted to RFGPA thus permitting data to be passed to various hardware designs. Since the VHDL code is being run as hardware, RFPGA creates a platform for a potential supercomputer to run faster than a straight software implementation. Since all of this occurs on one chip, it may offer significant improvements over the PAM architecture discussed in the literature review.

The design of the public-key decryption unit of the system brought insight into one of VHDL's shortcomings. That is, hardware being executed in sequence using behavioural code with procedure calls is not as simple a task as it should be. Suggestions were made and pseudo-code was given as to how this could be made more effortlessly. The inclusion of the recommendations into the VHDL language specification would make behavioural programming easier and more relevant to area efficient design segments, and the reduction of maintenance costs. I refer the reader to the actual source code to see how this was accomplished.

Future Work

RFPGA was mostly synthesized using the Xilinx ISE XST synthesis tools. Future students may wish to ensure adequate RAM requirements before attempting to synthesize the entire system from the top entity, "ssfpga.vhd". Each entity below ssfpga, however, synthesized successfully. The entire design simulated successfully for various tests.

Future work should involve the creation of synthesis tools for applications to run on top of RFPGA. Such tools would allow for the easy linking of one hardware design's states to the subsequent design's invocation. Currently, the placement and routing of designs is done mostly manually, and therefore, only small test designs can be synthesized.

The public key decryption unit was written as behavioural code for ease of modification. Future students wishing to modify the design may opt to allow for computations based on a subset of each math function's maximum output. In doing so, variable lengths of integers can be computed, and hence the design will be in use longer as cryptographic applications require greater security. Routines to shift subsets of data in and out of the basic math routines would be required as well. Any common computational speed inefficiencies in the Montgomery implementation should also be addressed. The primary one being the parallelization of multiplication and reduction routines which is currently not done, but is common in many Montgomery custom designs.

The Gunther Alternating Step Generator could also be altered to a programmable length. Currently, the length is easily set using a constant, but that constant is set at synthesis time and not modifiable after that. Everything else is programmable. By allowing for a dynamic point to read off the last bit out of the register, one could offer a slightly more flexible alternative. Currently, the non-linear combining function is a simple addition operation. While this has proven to be sufficient, one may modify it to another programmable LFSR to incorporate future enhancements in non-linear function research.

Appendices

Acronym List

ASG	Alternating Step Generator
ASIC	Application Specific Integrated Circuit
BLE	Basic Logic Element
CAD	Computer Automated Design
CB	Connection Block
CLB	Configurable Logic Block (see also BLE)
CSS	Contents Scrambling System
DFT	Design For Testability
DVD	Digital Versatile Disc
FPGA	Field Programmable Gate Array
FU	Functional Units
G-ASG	Gunther ASG
HDL	Hardware Description Language
IC	Interconnect Switch
IOB	Input Output (Logic) Block
IP	Intellectual Property
LFSR	Linear Feedback Shift Register
LSSD	Level Sensitive Scan Design (by IBM)
LUT	Look-Up Table
MPGA	Mask Programmable Gate Array
OTP	One Time Pad
OTP	One Time Programmable
PAM	Programmable Active Memory
PK	Public Key
PLA	Programmable Logic Array
RAM	Random Access Memory
RB	Reconfigurable Logic Block – any generation
RB3	Reconfigurable Logic Block – third generation
RB4	Reconfigurable Logic Block – fourth generation
RFPGA	R <=> Our, R<=>Reconfigurable (see also SCCRFPGA) (RFPGA is the entire design)
RLB	Routable Logic Block (for Triptych FPGA Architecture)
SCCR-FPGA	Single Clock Cycle Reconfigurable FPGA (see also RFPGA)
SDFF	Scan Design Flip Flop (a generic term)
SRAM	Static Random Access Memory
SRSize	Shift Register Size
twidth	Track Width
U of M	University of Manitoba (in Canada)
U of T	University of Toronto (in Canada)
vbits	vector bits
VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC HDL (Very High Speed Integrated Circuit Hardware Description Language)
XOR	Exclusive – OR (Boolean logic operation)

References

1. Stephen D. Brown, Robert J. Francies, Jonathan Rose, Zvonko G. Vranesic, "Field Programmable Gate Arrays", Kluwer Academic Publishers, Boston, 1993.
2. Peter Alfke, "Dynamic Reconfiguration",Xilinx XAPP093, November 10,1997.
3. Paul Chow, Soon Seo, Jonathan Rose, Kevin Chung, Gerard Paez-Monzon, Immanuel Rahardja, "The Design of a SRAM-Based Field-Programmable Gate Array – Part I: Architecture", IEEE Transactions on VLSI Systems, Vol. 7, No.2, June 1999.
4. Paul Chow, Soon Seo, Jonathan Rose, Kevin Chung, Gerard Paez-Monzon, Immanuel Rahardja, "The Design of a SRAM-Based Field-Programmable Gate Array – Part II: Circuit Design and Layout", IEEE Transactions on VLSI Systems, Vol. 7, No.2, June 1999.
5. Ethan Mirsky, Andre DeHon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources", FPGAs for Custom Computing Machines, Proceedings, IEEE, 1996
6. Scott Hauck, "The Roles of FPGAs in Reprogrammable Systems", Proceedings of the IEEE. Vol. 86, No. 7, April 1998.
7. Synopsys Website: http://www.synopsys.com/
8. Model Technology Website: http://www.model.com/
9. Xilinx Corporation Website: http://www.xilinx.com/
10. XESS Corporation Website: http://www.xess.com/
11. Gaetano Borriello, Carl Ebeling, Scott Hauck, Steven Burns, "The Triptych FPGA Architecture", IEEE Transactions on VLSI Systems, Vol. 3, No. 4, December 1995.
12. Kevin Chung, "Architecture and synthesis of field-programmable gate arrays with hard-wired connections", Ph.D. dissertation, Dept. Electrical and Computer Engineering, University of Toronto, Canada 1994.
13. J. Rose, R. Francis, D. Lewis, P. Chow, "Architecture of Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency", IEEE Journal of Solid State Circuits, Vol. 25, No 5, October 1990.
14. Jean Vuillemin, Patrice Bertin, Didier Roncin, Mark Shand, Herve Touati, Philippe Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age", IEEE Transactions on VLSI Systems, Vol. 4, No. 1, March 1996.
15. "The New Lexicon Webster's Encyclopedic Dictionary of the English Language, Canadian Edition", Lexicon Publications, 1988.
16. Bruce Schneier, "Applied Cryptography, 2nd Edition", Wiley, 1996.
17. Alfred Menezes, Paul van Oorschot, Scott Vanstone, "Handbook of Applied Cryptography", CRC Press, June 2001edition.
18. Auguste Kerckhoffs, "La cryptographie militaire", Journal des sciences militaires, vol. IX, pp. 5-83, Jan. 1883, pp. 161-191, Feb. 1883
19. Paul Bardell, William McAnney, Jacob Savir, "Built-In Test for VLSI: Pseudorandom Techniques", John Wiley & Sons Inc., 1987.

20. Benson Cheung, L. T. Wang, "The Seven Deadly Sins of Scan-Based Designs", ISD, August 1997. (http://www.eedesign.com/editorial/1997/test9708.html)
21. Ken Jaramillo, Subbu Meiyappan, "10 tips for successful scan design", EDN Access, February 2000.
22. Peter L. Montgomery, "Modular Multiplication Without Trial Division", Mathematics of Computation, Vol 44, No 170, April 1985, pages 519-521.
23. Cetin Kaya Koc, Tolga Acar, Burton S. Kaliski Jr., "Analyzing and Comparing Montgomery Multiplication Algorithms", RSA Laboratories, IEEE Micro, June 1996.
24. Braden Phillips, "Modular Multiplication in the Montgomery Residue Number System", IEEE Signals, Systems and Computers, 2001. Conference Record of the Thirty-Fifth Asilomar Conference on , Volume: 2 , 2001, pages 1637-1640.
25. Alan Daly, William Marnane, "Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic", ACM FPGA '02, February 24-26, 2002.
26. C. K. Koc, "Montgomery reduction with even modulus",IEE Proceedings, Computers and Digital Techniques, pages 314-316, Sept 1994.
27. J. Oh, S. Moon, "Modular multiplication method", IEEE Proceedings, Copmut. Digit. Tech., Vol. 145, No 4, July 1998.
28. Shigehiro Funatsu, Masato Kawai, Akihiko Yamada, "Scan Design at NEC", NEC Corp., IEEE Design & Test of Computers, June 1989.
29. Diffie, Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory Vol 22, 1976, 644-654.
30. Hellman, Diffie, Merkle, "Cryptographic Apparatus and Method", U.S. Patent #4,200,7700, April 29, 1980.
31. Hellman, Diffie, Merkle, "Cryptographic Apparatus and Method", Canadian Patent #1,121,480, April 6, 1982.
32. R.L. Rivest, A. Shamir, and L.M. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM (2) 21, 1978, 120-126.
33. http://www.rsasecurity.com/ (see the FAQ section for general cryptography information), The RSA Company.
34. Moore, "Protocol Failures in Cryptosystems", Proceedings of the IEEE, vol.76, No. 5, May 1998.
35. Moore, "Protocol Failures in Cryptosystems", Contemporary Cryptology: The Science of Information Integrity, G.J. Simmons, ed., IEEE Press, 1992, pages 541-558.
36. http://www-2.cs.cmu.edu/~dst/DeCSS/Gallery/ , Touretzky, D. S. (2000) Gallery of CSS Descramblers, Computer Science Department, Carnegie Mellon University
37. S. Berkovits, "How to Broadcast a Secret", Advances in Cryptology – Eurocrypt 1991, lecture Notes in Computer Science, 1991, vol. 547, pages 536-541
38. Yuh-Min Tseng, Jinn-Ke Jan, "Cryptanalysis of Liaw's Broadcasting Cryptosystem", Pergamon, Computers and Mathematics with Applications Vol 41, 2001, pages 1575-1578

39. Chen, Chang , "Secure Information Broadcasting Scheme using Embedded Locks", Computer Systems Science and Engineering, vol. 10, No.2, Apr, 1995, p 67-74.
40. Chiou, Chen, "Secure Broadcasting Using the Secure Lock", IEEE Transactions on Software Engineering, Vol. 15, No. 8, August 1998.
41. Alfred Menezes, Paul van Oorschot, Scott Vanstone, "Handbook of Applied Cryptography", CRC Press, June 2001edition, page 154, section 4.5 "Irreducible polynomials over Zp".
42. Rainer A. Rueppel, "Analysis and Design of Stream Ciphers", Springer-Verlag, (c) 1986.
43. http://wims.unice.fr/~wims/fr_tool~algebra~primpoly.en.html Xiao Gang, "Primpoly", University of Nice, France.,
44. http://www.sosmath.com/algebra/factor/fac01/fac01.html Helmut Knaust, "Polynomial Long Division", S.O.S. Mathematics.
45. S. Golomb, "Shift Register Sequences", Holden-Day, San Francisco, California, 1967.
46. J. Massey, "Shift-Register Synthesis and BCH Decoding", IEEE Transactions on Information Theyory, v IT-15, January 1969.
47. R. K. Fjelstad, W. T. Hamlen, "Application Program Maintenance Study: Report to Our Respondents", Proceedings Guide 48, Philidelphia PA, 1979.
48. C. G. Gunther, "Alternating Step Generators Controlled by De Bruijn Sequences", Eurocrypt '87, published in 1988.
49. www.fzi.de/sim/publications/1994002-paper.pdf Gutberlet, Rosenstiel, "Timing Preserving Interface Transformations for the Synthesis of Behavioural VHDL.", , Forschungszentrum Informatik (FZI), University of Tuebingen, no publication date given.
50. C. Blundo, L. Frota Mattos and D. R. Stinson, "Trade-offs between communication and storage in unconditionally secure schemes for broadcast encryption and interactive key distribution.", Lecture Notes in Computer Science 1109 (1996), 387-400, (Advances in Cryptology - CRYPTO '96).
51. D. R. Stinson and R. Wei., "Key preassigned traceability schemes for broadcast encryption.", Lecture Notes in Computer Science 1556 (1999), 144-156, (SAC '98 Proceedings).
52. J. Armstrong, F. Gray, "VHDL Design Representation and Synthesis", 2nd Ed., (c) 2000, Prentice Hall.
53. V. Betz, J. Rose, A. Marquardt, "Architecture and CAD for Deep-Submicron FPGAs", (c)1999, Kluwer Academic Publishers, 2nd printing, 2000.