Indexing Large Permutations in Hardware

Jacob H. Odom

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Peter M. Athanas, Chair
Thomas L. Martin
Joseph G. Tront

May 9, 2019
Blacksburg, Virginia

Keywords: permutations, combinatorics, hardware acceleration, Fisher-Yates,
Knuth-Shuffle

Indexing Large Permutations in Hardware

Jacob H. Odom

(ABSTRACT)

Generating unbiased permutations at run time has traditionally been accomplished through application specific optimized combinational logic and has been limited to very small permutations. For generating unbiased permutations of any larger size, variations of the memory dependent Fisher-Yates algorithm are known to be an optimal solution in software and have been relied on as a hardware solution even to this day. However, in hardware, this thesis proves Fisher-Yates to be a suboptimal solution. This thesis will show variations of Fisher-Yates to be suboptimal by proposing an alternate method that does not rely on memory and outperforms Fisher-Yates based permutation generators, while still able to scale to very large sized permutations. This thesis also proves that this proposed method is unbiased and requires a minimal input. Lastly, this thesis demonstrates a means to scale the proposed method to any sized permutations and also to produce optimal partial permutations.

Indexing Large Permutations in Hardware

Jacob H. Odom

(GENERAL AUDIENCE ABSTRACT)

In computing, some applications need the ability to shuffle or rearrange items based on run time information during their normal operations. A similar task is a partial shuffle where only an information dependent selection of the total items is returned in a shuffled order. Initially, there may be the assumption that these are trivial tasks. However, the applications that rely on this ability are typically related to security which requires repeatable, unbiased operations. These requirements quickly turn seemingly simple tasks to complex. Worse, often they are done incorrectly and only appear to meet these requirements, which has disastrous implications for security. A current and dominating method to shuffle items that meets these requirements was developed over fifty years ago and is based on an even older algorithm refer to as Fisher-Yates, after its original authors. Fisher-Yates based methods shuffle items in memory, which is seen as advantageous in software but only serves as a disadvantage in hardware since memory access is significantly slower than other operations. Additionally, when performing a partial shuffle, Fisher-Yates methods require the same resources as when performing a complete shuffle. This is due to the fact that, with Fisher-Yates methods, each element in a shuffle is dependent on all of the other elements. Alternate methods to meet these requirements are known but are only able to shuffle a very small number of items before they become too slow for practical use. To combat the disadvantages current methods of shuffling possess, this thesis proposes an alternate approach to performing shuffles. This alternate approach meets the previously stated requirements while outperforming current

methods. This alternate approach is also able to be extended to shuffling any number of items while maintaining a useable level of performance. Further, unlike current popular shuffling methods, the proposed method has no inter-item dependency and thus offers great advantages over current popular methods with partial shuffles.

Dedication

This work is dedicated to Samantha and Zoey. They literally stayed by my side through the entirety of this work and my academic career to provided support through frustrations in the way only dogs can. In appreciation of their support, the name of the proposed method this paper introduces, CRGE (kôrgē), was inspired by them.

Acknowledgments

First and foremost, I would like to thank my mother, Keena Garns. Without her undeserved support and trust I would have never been able to start my academic career or be where I am today. I will be forever grateful and indebted to her for not giving up on me.

I would also like to thank my research advisor Dr. Peter Athanas for his guidance, motivation and support. He was always willing to set aside time for me and clearly showed his desire for me to succeed in all areas through his actions.

I would like to thank the National Science Foundation for their Scholarship for Service program. This program provided me support in pursuing a graduate degree, helped me discover my passion for information security and connected me with my current employer.

Lastly, thank you to Dr. Thomas Martin and Dr. Joseph Tront for begin accommodating while serving on my advisory committee.

My appreciation to all mentioned cannot be overstated.

Contents

List of Figures x

List of Tables xii

1 Introduction 1

2 Background Mathematics 3

2.1 Definitions . . . 3

2.2 Notation . . . 5

2.3 Number Systems and Number Representation . . . 6

2.4 Permutations . . . 7

2.5 Ideal Permutation Generator . . . 8

3 Popular Current Methods for Indexing Permutations 10

3.1 Memory-less Approaches . . . 10

3.2 Memory Dependent Approaches . . . 12

3.2.1 Fisher-Yates . . . 12

3.2.2 Knuth-Shuffle . . . 13

3.2.3 Random-Sort . . . 14

4 Proposed Method **17**

4.1 Visual Representation 17

4.2 Mathematical Representation 20

4.3 Proof of Soundness 20

5 Methodology **22**

5.1 Evaluation Method 22

5.2 Tested Designs 23

5.2.1 Combinational Designs 23

5.2.2 Knuth-Shuffle Designs 24

5.2.3 Knuth-Shuffle Design Cycle Count (Worst Case) 29

5.2.4 CRGE Designs 30

6 Results **37**

6.1 Evaluation of I2P Implementation 37

6.2 Comparison of Knuth Shuffle Implementations 39

6.3 Comparison of CRGE Implementations 44

6.4 Comparison of Approaches 47

6.5 Comparison of AREA (Small n) 47

6.6 Comparison of AREA (Large n) 48

6.7 Comparison of Performance (Small n) 49

6.8 Comparison of Performance (Large n) 49

6.9 Comparison of Running Time 50

7 Additional CRGE Uses and Future Work **52**

7.1 Partial Permutation Model 52

7.2 Distributed Model 53

7.3 High Throughput Design 54

7.4 Software Implementation 55

8 Conclusion **56**

Bibliography **58**

List of Figures

4.1 Example visual representation of CRGE 18

4.2 Example transformation 18

5.1 I2P four element permutation circuit [1] 24

5.2 Knuth-Shuffle naive four element permutation circuit 25

5.3 Knuth-Shuffle RAM output w/initialization four element permutation circuit 26

5.4 Knuth-Shuffle shift register output w/"dirty bit" logic four element permutation circuit 28

5.5 CRGE basic "compact" design four element permutation circuit 31

5.6 CRGE precomputational design four element permutation circuit 34

5.7 CRGE shift register design four element permutation circuit 35

6.1 I2P implementation: area vs. n 38

6.2 I2P implementation: maximum clock frequency vs. n 39

6.3 Knuth-Shuffle implementations: area vs. small n 40

6.4 Knuth-Shuffle implementations: maximum clock frequency vs. small n . . . 41

6.5 Knuth-Shuffle implementations: area vs. large n 42

6.6 Knuth-Shuffle implementations: maximum clock frequency vs. large n 43

6.7 CRGE implementations: area vs. small n 44

6.8 CRGE implementations: maximum clock frequency vs. small n 45

6.9 CRGE implementations: area vs. large n 45

6.10 CRGE implementations: maximum clock frequency vs. large n 46

6.11 Approach representatives: area vs. small n 47

6.12 Approach representatives: area vs. large n 48

6.13 Approach representatives: maximum clock frequency vs. small n 49

6.14 Approach representatives: maximum clock frequency vs. large n 50

6.15 Approach representatives: worst case running time vs. small n 51

6.16 Approach representatives: worst case running time vs. large n 51

7.1 CRGE partial permutation design 52

7.2 CRGE distributed design 53

7.3 CRGE high throughput design 54

List of Tables

5.1 Worst case cycle complexity to complete and read a permutation for Knuth-Shuffle designs 30

5.2 Input/output table of f_2 function for CRGE basic "compact" design 32

5.3 Partial element calculated by each computational block by cycle for the CRGE "compact" design four element permutation 33

Chapter 1

Introduction

A permutation in layman terms is simply a rearrangement or shuffling of a collection of items. In computing, predetermined and static permutations appear in a multitude of places ranging from cryptographic standards [2][3] to memory access optimization [4][5]. There also exists a large range of applications that require producing permutations dynamically at run-time based on a provided or random-source input. However, efficiently producing permutations in an unbiased manner can prove to be difficult [6].

Current popular methods of dynamically producing unbiased permutations suffer from major drawbacks. These drawbacks hinder their ability to efficiently produce large permutations. The major drawbacks to current methods stem from inter-dependencies between permutation elements. Within these current methods, elements are computed one at a time, where each element's result is dependent on the results of all of the previously computed elements. This fact has several negative impacts. First, current methods have a complexity growth that results in computing large permutations requiring an unfeasible amount of resources and performance degradation that makes producing even small permutations inefficient. Secondly, this inter-dependency requires permutations to be generated and treated as singular objects instead of as a collection of elements which ultimately limits scale-ability. To mitigate the complexity issue, current higher performance methods utilize memory. While this use of memory allows for larger permutations and higher performance, it also ultimately sets an upper bound on the maximum performance of these methods based on memory access times

and delays. These higher performance methods also do nothing to address or eliminate the inter-dependency issue and thus will always face a scalability problem when large enough permutations are considered.

To combat the complexity and scalability issues of current methods while also avoiding the performance penalties of utilizing memory, this thesis propose an alternate approach to dynamically producing large unbiased permutations. The proposed approach is able to produce permutations element by element completely independently of each other. This thesis will introduce the proposed approach to dynamically generating unbiased permutations and show the proposed approach does, in fact, produce unbiased permutations. Further, this thesis will show the proposed approach out performs current popular methods by comparing implementations on a selected Field Programmable Gate Array (FPGA). Finally, this thesis will also provide guidance on how to take advantage of the proposed approach's element independent nature when producing partial permutations and when scaling to larger permutations that would otherwise exceed the capabilities of a device.

The remainder of this paper is outlined as follows. In Chapter 2 any relevant background mathematics required to thoroughly understand the problem and proposed alternate approach is provided while establishing the mathematical notation used within. In Chapter 3 an overview of currently established approaches to the problem is provided. In Chapter 4 the proposed approach is introduced and formal verification of the proposed approach is provided. In Chapter 5 the used representative implementations for each approach and the methodology used to evaluate and compare the performance of each approach's representative implementations is described. In Chapter 6 an analysis of the results is presented. Chapter 7 provides guidance on extending the proposed method to any sized permutations and capitalizing on its benefits for generating partial permutations. Lastly, Chapter 8 contains a reflection on everything covered throughout this thesis and concluding remarks.

Chapter 2

Background Mathematics

This chapter provides any relevant background information necessary to understand any presented mathematics. This thesis attempts to make no assumptions on the level of background mathematical knowledge of the reader. However, the goal of this chapter is to only provided the cursory level of information required to follow discussed topics and not necessarily to serve as a thorough reference. Therefore, a dedicated reader seeking a deeper understanding in a particular area my still need to consult outside resources.

2.1 Definitions

This section provides definitions to relevant mathematical terms used throughout this thesis. It mostly serves as a quick and convenient reference for the remainder of this chapter.

Definition 2.1. Set: Any collection of objects specified in a well-defined manner.

Definition 2.2. Element: An item in a particular set.

Definition 2.3. Domain: The input set for which a function is defined.

Definition 2.4. Codomain: The set into which all of the output of the function is constrained to fall.

Definition 2.5. Mapping: A more general term for a function. Commonly used when the domain or codomain may or may not be a subset of $\mathbb{C}$.

Definition 2.6. Bijection: A mapping in which every element in the domain induces exactly one element in the codomain and for every element in the codomain there is exactly one element in the domain from which it is induced.

Definition 2.7. Permutation: A specific ordered arrangement of a set. Often defined as a bijection mapping from a set to itself.

Definition 2.8. Group: A set G with an operation $*$ that satisfies the following:

1. $\forall a, b, c \in G \ (a * b) * c = a * (b * c)$
2. There is an element $e \in G$ such that $\forall a \in G \ a * e = a = e * a$. Call this element e the identity element of G.
3. $\forall a \in G$, there is an element $a^{-1} \in G$ such that $a * a^{-1} = e = a^{-1} * a$. Call this element a^{-1} the inverse of a.

Definition 2.9. Mapping Composition: Given mappings $f : A \mapsto B$ and $g : B \mapsto C$ the composite mapping operator applied to f and g denoted as $g \circ f$ results in a mapping $A \mapsto C$ defined as:

$$(g \circ f)(x) = g(f(x)) \ \forall x \in A$$

Definition 2.10. Cartesian Product: The Cartesian product of two sets A and B denoted as $A \times B$ is the set of all ordered pairs (a, b) such that $a \in A$ and $b \in B$.

Definition 2.11. Modulo Operation: The modulo operation maps $\mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{Z}$. Given $(a, b) \in \mathbb{Z} \times \mathbb{Z}$ the modulo operation is denoted $a \bmod b$ and is defined as the unique r such that $a = qb + r$ where $q, r \in \mathbb{Z}$ and $0 \leq r < b$.

2.2 Notation

This section contains and defines any mathematical notation used throughout this thesis.

$\{\}$	Unordered set*
$()$	Order set
$\in$	Element of
$\forall$	For All
$\mathcal{S}$	Uppercase letters distinguish a set
$\mathbb{Z}$	The set of integers $\{..., -2, -1, 0, 1, 2, ...\}$
$\mathbb{C}$	The set of complex numbers
$\mathbb{N}$	The set of natural numbers $\{1, 2, ...\}$
f	Lowercase letters distinguish a function or mapping
σ	Specifically used for naming a mapping representative of a permutation. In a context where multiple permutations are discussed, subscripts are used to differentiate permutations.
$\mathbb{Z}_i$	**The group formed from the set $\{0, 1, ..., i-1\}$ with the $*$ operator defined as given $a, b \in \{0, 1, ..., i-1\}$ $a * b = (a + b) \bmod i$

* This is not to be confused with the common Verilog and circuit notation. Within these contexts, $\{\}$ will refer to concatenation as one would expect.

** $i \in \mathbb{Z}$

2.3 Number Systems and Number Representation

While the origins of the modern number system are debated, it's adoption is mostly credited to Leonardo of Pisa during the thirteenth century [7, p. 277-280]. The Hindu-Arabic number system, as the modern number system is commonly referred, is a base ten positional system. The base, or radix, refers to the number of different symbols required to represent all numbers while the term positional refers to the fact that the order or position of these symbols determines their value [8, p. 36].

Any computer scientist or engineer is likely to be familiar with several other alternative based positional systems such as base two binary, base eight octal and base sixteen hexadecimal. With a positional system one can uniquely represent any value by:

$$n = d_k \cdot b^k + d_{k-1} \cdot b^{k-1} + ... + d_2 \cdot b^2 + d_1 \cdot b + d_0 \tag{2.1}$$

where n is the value to represent, b is the base, $\forall i \; d_i \in \{0, 1, ..., b-1\}$ and k is minimal.

One can even go a step farther and define a positional system in a more general way, such that each digit has a separately defined base. Such a positional system is referred to as a mixed radix system. To be able to represent a value in a mixed radix system Equation 2.1 must be generalized to:

$$n = d_k \cdot \Pi_{i=0}^{k-1} b_i + d_{k-1} \cdot \Pi_{i=0}^{k-2} b_i + ... + d_2 \cdot \Pi_{i=0}^{1} b_i + d_1 \cdot b_0 + d_0 \tag{2.2}$$

Where $\forall i \; b_i$ is the base of d_i. It is easy to verify that Equation 2.2 is consistent with Equation 2.1 and one's intuitions. However, for Equation 2.2 to uniquely represent values, for all $i > 0 \; b_i \geq b_{i-1}$ must be true. The Mayan Number System provides an example of a mixed radix number system that fails to meet this requirement and results in non-unique

value representation [7, p. 7-8][9]. Using Equation 2.2, consider a mixed radix system where every digit's base is one greater than its position resulting in:

$$n = d_k \cdot \Pi_{i=0}^{k-1}(i+1) + d_{k-1} \cdot \Pi_{i=0}^{k-2}(i+1) + ... + d_2 \cdot \Pi_{i=0}^{1}(i+1) + d_1 \cdot 1 + d_0 \tag{2.3}$$

$$n = d_k \cdot \Pi_{i=1}^{k} i + d_{k-1} \cdot \Pi_{i=1}^{k-1} i + ... + d_2 \cdot \Pi_{i=1}^{2} i + d_1 \cdot 1 + d_0 \tag{2.4}$$

$$n = d_k \cdot k! + d_{k-1} \cdot (k-1)! + ... + d_2 \cdot 2! + d_1 \cdot 1! + d_0 \cdot 0! \tag{2.5}$$

Where $\forall i \ d_i \in \{0, 1, ..., i\}$. The mixed radix system under consideration and presented in Equation 2.5 was named the Factorial Number System by Knuth [10, p. 192]. The Factorial Number System provides a convenient way to number, or index, permutations and will be used as such throughout this thesis. You may notice that d_0 can only ever be zero thus cannot convey any meaningful information about a value. As such, d_0 will be excluded when representing a value in the factorial number system when the digit is not directly referenced. The following notation will also be used when referring to a number in the factorial number system: digits will be separated by commas and d_1 will carry the subscript "!". For example, to represent the number 13 one would write $2, 0, 1_!$ since $13 = 2 \cdot 3! + 0 \cdot 2! + 1 \cdot 1!$.

2.4 Permutations

Put simply, a permutation is a bijection mapping from a set to itself. Often the symbols σ or π are used to label a mapping that represents a permutation. For the context of this thesis, only permutations on the set $\{0, 1, ..., n-1\}$ where n is the number of elements to permute are considered. Working with permutations of this set easily extend to permutations of any n element set by composing the permutation with an additional bijection mapping. Restricting considered permutations to permutations on the set $\{0, 1, ..., n-1\}$ offers a convenient way

to represent a permutation as the ordered set $(\sigma(0), \sigma(1), ..., \sigma(n))$. As such, the ordered set $(\sigma(0), \sigma(1), ..., \sigma(n))$ and the actual mapping σ will be considered to be equivalent and will be used interchangeably as convenient. The set of all n element permutations will be refer to as S_n. The permutation $(0, 1, .., n-1) \in S_n$ is the identity permutation as it makes no change to the order of the elements to which it is applied. One can verify that the set S_n forms a group with the operation $*$ defined as the composite mapping operator where e is the identity permutation.

2.5 Ideal Permutation Generator

For this thesis an ideal permutation generator will be defined as follows:

Definition 2.12. Ideal Permutation Generator: Any mapping with a co-domain of S_n that also holds the following properties:

1. **Truly unbiased:** Given an unknown arbitrary element of the domain, every permutation in S_n must be equally likely to be the resulting permutation.
2. **Deterministic:** The permutation generator should always produce the same permutation for a particular input.
3. **The domain is minimal:** There is no ideal permutation generator with a smaller domain.
4. **Elements of the domain can easily be produced:** For any n, an element of the domain can be produced or generated with constant complexity.

The presented definition of an ideal permutation generator is important for and focused at applications. In efforts to not exclude certain applications of permutations; the definition was made as restrictive as possible. It is also typically easier to relax constraints than it is to apply them. Therefore, one could possibly modify an ideal permutation generator to operate in an advantageous manner when one or more requirements are not strictly needed. The opposite, typically, would be significantly more difficult or impossible. For example, in Section 3.2.3 the reader will see the deceptive difficulties in achieving both properties 1 and 2. One can also note that properties 1, 2 and 3 together imply an ideal permutation generator is a bijection mapping. Thus, it is possible to show a given method is an n element ideal permutation generator if and only if it is a bijection mapping to S_n and an arbitrary element of the domain can be generated in constant time and memory. This fact will be used when proving a permutation generator is ideal.

Property 4 mainly serves to eliminate any trivial or naive generators from being considered ideal. For example, consider an identity permutation generator where the domain is S_n and the generator simply returns any input permutation. It is easy to verify that such a generator would satisfy the first three requirements. However, such a generator would be unproductive, useless and off loads the entirety of the computational complexity of the generator into producing an item in its domain. Thus, Property 4 was included to eliminate any such generators from being considered ideal.

Chapter 3

Popular Current Methods for Indexing Permutations

This chapter serves to introduce the contemporary methods of generating unbiased permutations discussed in this thesis. Where applicable, relevant background information is also provided. For each method that is later evaluated, a justification is provided to show the method meets the requirements of Definition 2.12 and thus qualifies as an ideal permutation generator within the context of this thesis. Additionally, an example of a commonly misused method that fails to meet the requirements of an ideal permutation generator is presented to demonstrate common pitfalls when designing an ideal permutation generator.

3.1 Memory-less Approaches

Memory-less approaches simply refer to methods of producing unbiased permutations that, by design, do not fundamentally rely on memory to operate. In other words, it would be possible for the design to be implemented solely with combinational logic capable of functioning properly, albeit at a relatively low clock frequency, over a single clock cycle.

While these methods are functional distinct from the memory based methods to be discussed in Section 3.2, fundamentally all current popular methods of generate unbiased permutations follow a similar logical approach. Generally, these approaches can be divided into a series

of logical blocks. Starting with an identity permutation, at each block a remaining element is selected based upon the input index. This selected element is removed and the remaining elements are passed to the next block. The order the elements are selected and removed defines the generated permutation.

Memory-less approaches offer several distinct advantages. Since they are able to function as pure combinational logic they can be implemented as a relatively simple circuit. They can also be used in instances where memory or even a clock are impractical or unfeasible, such as extremely low power devices or where no continuous or reliable source of power exists.

Of course the same properties that provide advantages in certain situations are also the source of disadvantages in others. The nature of the designs requires similar or identical logic to be implemented a large number of times. This causes a high complexity growth in terms of area. Likewise, commonly, components are used in the logic that scale poorly as the number of inputs increases. This not only exacerbates the area growth but also adversely affects delays resulting in an undesirable time complexity growth.

Since there is a wide range of ways one could implement this selection and removal process in digital logic, it could be argued that there are nearly an endless number of distinct ways to implement a memory-less approach. To constrain and focus this thesis, a single memory-less design was selected that provides a clever implementation of the selection process and properly conveys the typical memory-less approach. It should be stressed that this choice was made with the intentions of showcasing a strong representative of memory-less approaches. Therefore, this thesis operates under the implied assumption that an arbitrarily selected memory-less approach would result in similar conclusions being drawn and they are not a result this choice.

To represent current combinational designs, the design described in [1] (I2P) was selected.

I2P uses a repeated block structure that can be extended to generated permutations for any size n. Each block in I2P takes the current state of the calculated permutation and an index $[0, n_r! - 1]$ where n_r is the number of permutation elements yet to be computed. At each block, based on the input index, a remaining element is selected and moved to the top of the remaining elements, while the remaining elements above the selected element are shift down to fill the space. It should be clear that I2P makes a bijection mapping from $[0, n! - 1]$ to S_n. I2P takes an input index in the range $[0, n! - 1]$ that can be generated in constant time with constant memory. Therefore, one can conclude I2P is an ideal permutation generator.

3.2 Memory Dependent Approaches

Naturally, an alternate approach to generating unbiased permutations while mitigating the complexity issues of purely combinational circuits is to utilize sequential logic. A single instance of the set of values to permute is stored in memory. Algorithmically, the memory contents are manipulated over several cycles. The result is the memory holding a permutation of the originally stored values.

3.2.1 Fisher-Yates

Originally published in 1938, [11] contains a memory dependent method of producing a random permutation labeled as Example 12. However, it is more commonly referred to as simply Fisher-Yates after the authors' names. The method centers around randomly selecting and removing an entry from a list. This procedure is repeated on the remaining values of the list until the list contains one entry, which can simply be removed. The permutation is the list of entries in the order they were removed. This method, when implemented as originally

described, is intended to be performed by hand utilizing a table of previously produced random values. This, however, is very impractical for all but the smallest of permutations. To mitigate this issue, larger sets are first randomly divided in a number of bins. Each bin containing more than one value has its values permuted as before. The order of the bins and the order of their values is the resulting permutation. This process can also be repeated recursively so that at any one stage the number of bins and maximum number of values in a bin doesn't exceed a set threshold. This method, however, can result in requiring an input larger than the theoretical minimum and is thus not ideal.

3.2.2 Knuth-Shuffle

In 1964, Richard Durstenfeld proposed an improvement to Fisher-Yates known by its published name, Algorithm 235 [12]. However, Durstenfeld's method is most commonly known from its use in Knuth's work [10, p. 145-146] and as such is often referred to as the Knuth-Shuffle even though this implies incorrectly crediting Knuth with its creation. The method is an improvement of Fisher-Yates, optimizing the method for a computer implementation. The method differs from Fisher-Yates by instead of removing a value from the list, the value is simply swapped with the value at the ending index. The ending index starts at the end of the list and is decreased each swap. When the ending index is the first item in the list the algorithm terminates and the list contains a permutation of its values. The advantage Knuth-Shuffle has over Fisher-Yates in a computer implementation is it is optimal in memory usage. The potentially wasted space discarded values would occupy in the list is shifted to the end and used to hold the resulting permutation. For this reason, Knuth-Shuffle is categorized as an in-place shuffle and is known to be optimal in software.

Due to its prevalence, uses of Knuth-Shuffle in the past few years are easy to find in image

encryption [13][14][15], file encryption [16], steganography [17], block ciphers [18] and communications [19][20]. While variations of the Fisher-Yates algorithm offer efficient ways to produce an unbiased permutation, like most circuits that utilize memory, when implemented in hardware they are ultimately hindered by memory access times and delays. Even when high performance memory is used, such as on chip Block RAM, one can expect critical paths to always be found in memory to memory paths.

Similarly, as with I2P, it should be clear that Knuth-Shuffle is a bijection mapping from an index value in the range $[0, n! - 1]$ expressed in the factorial number system to S_n. Likewise, such an input value can be generated in constant time and memory. Therefore, Knuth-Shuffle is an ideal permutation generator.

3.2.3 Random-Sort

The misleading simplicity of Fisher-Yates and Knuth-Shuffle may cause the reader to believe that producing an ideal permutation generator is fairly straight forward. To demonstrate this is not the case, consider the random sort, an alternative method that is commonly but erroneously cited as being deterministic and unbiased. As it will soon be clear to the reader, attempts to overcome this method's short comings ultimately exacerbate the issue and fail. While a formal analysis and complete critique of random sort could fill a paper in its own right, this thesis only aims to show how deceptively difficult it is to achieve the properties in Definition 2.12.

The random sort method is extremely straight forward. For each element in the set you wish to permute you assign a random key value. The elements of the set are then sorted based on their key values. It should be clear that this does in fact produce a permutation. One can also show that with this method every permutation is possible:

Proof. Assume this is not the case and some permutation in S_n is not produced by random sort, call it $\bar{\sigma}$. Since a group can be defined from S_n and $\bar{\sigma} \in S_n$ we have $\bar{\sigma}^{-1} \in S_n$. Assign key values to the elements of the permuted set according to $\bar{\sigma}^{-1}$ such that the first element's key is $\bar{\sigma}^{-1}(0)$, the second's key is $\bar{\sigma}^{-1}(1)$ and so forth. Sorting the keys will produce $\bar{\sigma}$, a contradiction.

$\therefore$ random sort produces all permutations in S_n. □

When one attempts to show this method is unbiased though, issues quickly arise. To see this, first consider S_2. Since computing resources are limited, one must set a bound on the range of random values used for keys. The actual bound has no bearing on the argument, just the fact that it exists is all that is important. Thus, consider random values on the arbitrary range $[0, m-1]$ and consider the elements to permute as a and b with random keys k_a and k_b. If $k_a < k_b$ then the result is (a, b). If $k_b < k_a$ then the result is (b, a). If $k_a = k_b$, for now assume the sort does not swap the elements as they are already sorted and thus the result is (a, b). Counting the number of values for k_b that produce (a, b) one finds 1 when $k_a = 0$, 2 when $k_a = 1$, ..., m when $k_a = m - 1$. This gives a total of $\Sigma_{i=1}^{m} i$ key pairs. Counting the number of values for k_b that produce (b, a) one finds $m-1$ when $k_a = 0$, $m-2$ when $k_a = 1$, ..., 0 when $k_a = m - 1$. This gives $\Sigma_{i=1}^{m-1} i$ total key pairs. Therefore, when given a random input (a, b) is more likely than (b, a). The same issue exists if the sort swaps on equal keys except (b, a) will be more likely than (a, b).

Randomly deciding to swap on equal keys may seem like a quick and obvious solution however, this is once again not the case. First, there is the issue of requiring additional input information to convey the random swaps taken if the method is to be deterministic. Secondly there are more nuanced implications depending on which sort algorithm is used. Consider a sorting algorithm like pivot sort and a set of all equal keys. On average, the keys will be partitioned into equal groups on each side of a pivot element. Due to this, pivot elements

will always have a bias away from the ends of the permutation. The only conclusion one can draw is if permutations are to be unbiased, repeating key values must be excluded. This of course violates, Property 4 of Definition 2.12. Worse though, even if the requirements are relaxed severely and one only wishes to have the method be unbiased to a high probabilistic degree issues are encountered. First, one must use a large range of values, $[0, m-1]$, for the keys to avoid repeated values at a high enough rate. One must also have $m^n - \Pi_{i=1}^{n-1} i$ be divisible (or have an insignificant remainder) by $n!$. These requirements lead to a domain that uncomprehendingly explodes in size even for small n and still doesn't manage to be truly unbiased. Based on this one can conclude that the properties of Definition 2.12 are essentially mutual exclusive when looking at the random sort method.

Chapter 4

Proposed Method

To best describe the proposed method for generating an unbiased permutation from an index, first a visual representation of the proposed method is presented. This visual representation accurately portrays one form of the underlying mathematics and is the source for the name Independent Permutation Element Retrieval Through Cyclic Rotations of Group Elements (CRGE [kôrgē]). Next, a generalized purely mathematical representation is covered. Finally, a proof of the proposed method's soundness as an ideal permutation generator is provided.

4.1 Visual Representation

As stated before, this visual representation is the source of the proposed method's name. In addition, this visual representation also provides insight into how the method was developed as it demonstrates the first iteration of the method's development. The method was later generalized into CRGE's current form as described in the next section, Section 4.2.

For example purposes, consider a permutation for $n = 5$ and index of $4, 2, 2, 1_!$, in the factorial number system. Start with an identity permutation, Figure 4.1a. Then, consider only the first two elements, and circularly rotate them right by the value of d_1 of the index, Figure 4.1b. Then, repeat with the first three elements using d_2 of the index, Figure 4.1c. This pattern continues, Figure 4.1d, until ultimately rotating the entire permutation using the most significant digit of the index d_4, Figure 4.1e. One could stop now and consider this

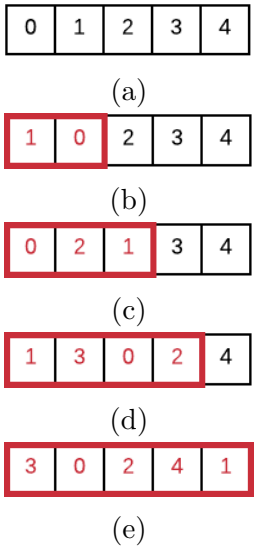

Figure 4.1: Example visual representation of CRGE

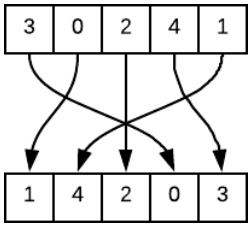

Figure 4.2: Example transformation

result the desired permutation as it can easily be shown to be an ideal permutation generator. However, applying an additional transformation to this result ultimately produces a much more useful method. For the transformation, simply calculate the inverse permutation by exchanging each element's value and position. For example, $\sigma(0)$ in the example has the value '3', this indicates the value $\sigma(3)$ in the resulting permutation is '0'. The transformation performed on the example is in Figure 4.2 and shows the final produced permutation.

Understanding the reasoning behind applying the final inverse transformation is key to understanding how CRGE actually functions. This final inverse transformation is such an important addition because it provides the insight that now the permutation elements can be calculated completely independently. It turns out that surprisingly this was always secretly the case, the transformation just makes this fact more apparent.

When calculating an element of a permutation one tends to be concerned with tracking the values of the element at each step in computing the permutation. Since this is essentially how all current popular methods compute elements, it can be difficult to deviate from this way of viewing the problem. But, this is also the cause for the inter-dependency of current methods since the value of all other elements must be known to track which value a particular element will hold next. If instead this idea is flipped on its head and what position a value is at each step is tracked, the values of the remainder of the elements don't need to be known. Also, if it is known that the result is in fact a permutation then no two values can be at the same position and there is no need to be concerned with the position of any other elements either. Therefore, no knowledge of the rest of the permutation is needed. Of course this will only compute what element holds a particular value which is the same as computing the inverse of a permutation! By adding this inverse transformation to the visual representation, a transformation is ultimately removed from when the calculations based on position instead of value are done.

4.2 Mathematical Representation

CRGE can provide a flexible and generalized mathematical representation of an n element permutation generator. For the mathematical representation the resulting permutation is calculated element by element. First, a set of functions is defined:

$$F = \{f_i : \mathbb{Z}_{i+1} \times \mathbb{Z}_{i+1} \mapsto \mathbb{Z}_{i+1} | i \in \mathbb{Z}, 0 \leq i \leq n-1\}$$

Where $\forall f_i \in F \ \forall a_1, a_2, b_1, b_2 \in \mathbb{Z}_{i+1}$

1. $f_i(a_1, b_1) = f_i(a_1, b_2) \implies b_1 = b_2$
2. $f_i(a_1, b_1) = f_i(a_2, b_1) \implies a_1 = a_2$

Then, given an index in the factorial number system with digits labeled $d_{n-1}, ..., d_2, d_1, d_0$ a permutation $\sigma \in S_n$ can be generated with CRGE by $\sigma(i) = f_{n-1}(...(f_{i+1}(f_i(i, d_i), d_{i+1})...), d_{n-1})$

4.3 Proof of Soundness

To prove CRGE is an ideal permutation generator, the same pattern as with the current popular methods is used. First, it is shown to be a bijection mapping from an integer index $[0, n! - 1]$ to a permutation in S_n. Then, an argument that CRGE does in fact satisfy Property 4 of Definition 2.12 is provided.

Proof. Let $n \in \mathbb{N}$. We will use induction on n. For each n let F be a set of functions as defined above, $d \in \{0, ..., n! - 1\}$ with digits in the factorial number system labeled $d_{n-1}, ..., d_1, d_0$ and define $\sigma(i) = f_{n-1}(...(f_{i+1}(f_i(i, d_i), d_{i+1})...), d_{n-1})$ for $0 \leq i \leq n - 1$

Suppose $n = 2$. $d = d_1 \in \{0, 1\}$. Without loss of generality assume $d_1 = 0$. We have $\sigma_0(0) = f_1(f_0(0, d_0), d_1)$ but $\mathbb{Z}_1 = \{0\} \implies f_0(0, d_0) = 0$ means $\sigma_0(0) = f_1(0, 0)$. We also have $\sigma_0(1) = f_1(1, 0)$. We know by our definition of f_1, $\sigma_0(0) \neq \sigma_0(1)$ since d_1 is held constant. Let $\sigma_0 = (a, b)$ $a, b \in \{0, 1\}, a \neq b$. Also by our definition of f_1, if $d_1 = 1$ it must be the case that $\sigma_1(0) = b$ and $\sigma_1 = (b, a)$. Thus applying CRGE to F forms a bijection $d \mapsto S_2$ when $n = 2$.

Now assume applying CRGE to F forms a bijection $d \mapsto S_n$ and consider the case for $n + 1$. By assumption for any d if we look at the partial computation of $\sigma(i)$ for $0 \leq i \leq n - 1$, $f_{n-1}(...(f_{i+1}(f_i(i, d_i), d_{i+1})...), d_{n-1})$ we have a permutation in S_n, call it σ_p. By the definition of f_n since σ_p is a permutation $f_n(\sigma_p(i_1), d_n) = f_n(\sigma_p(i_2), d_n) \implies i_1 = i_2$ also since $n \notin \{0, 1, ..., n - 1\}$ we have $f_n(\sigma_p(i), d_n) \neq f_n(n, d_n)$ for $0 \leq i \leq n - 1$. Therefore, $\sigma(i) = f_n(...(f_{i+1}(f_i(i, d_i), d_{i+1})...), d_n)$ is a permutation. We also know by our definition of f_{n-1} and f_n we have $n! \cdot (n + 1) = (n + 1)!$ permutations and applying CRGE to F forms a bijection $d \mapsto S_{n+1}$. □

Since CRGE is a bijection, one can conclude properties 1, 2 and 3 of Definition 2.12 are satisfied. To generate an element in the domain of CRGE one simply needs to produce a value $[0, n! - 1]$ which can be accomplished in constant time and with constant memory. Therefore, one can conclude CRGE satisfies Property 4 of Definition 2.12 and CRGE is an ideal permutation generator by definition.

Chapter 5

Methodology

5.1 Evaluation Method

To evaluate the performance of CRGE when compared to current popular methods of generating unbiased permutations, several implementations of current permutation methods and of CRGE were created. The designs were then compared based on area determined by percentage of device Adaptive Logic Module (ALM) utilization, performance determined by maximum operating frequency and speed determined by worst case time to compute then read a full permutation at various sizes of n. To fairly evaluate each approach, multiple design implementations were used to compensate for design decisions that may favor particular metrics at the cost of others.

The Intel Arria 10, model 10AX115N1F45E1SG FPGA, here after refer to as simply the Testing Device (TD), was the used device for all evaluations. TD was selected based on operating in the fastest class (1) and simultaneously offering a large amount of General Purpose Input/Output (GPIO) pins (768), ALM (427,200) and total memory bits (55,562,240). TD's memory bits are split between M20k RAM Blocks and Memory Logic Array Blocks (MLAB). Quartus Prime Pro version 18.1.0 was the tool chain used for every build. Each design was built twice for every value of n, once with tool chain options optimized for performance and once with tool chain options optimized for area. The values $n \in \{2, 3, ..., 10, 16, 32\}$ were used to evaluate the designs at small n. For higher n, powers of two greater than 32 where

used until a design failed to build. For each design and value of n, the results for performance and area were taken individually from the better of the two optimized builds.

When evaluating the performance of the designs, clock frequencies that exceeded the maximum speed of the device were considered valid. This allows for a better comparison of each implementation's potential and only impacts the smallest values of n. Also, this allows for more information to be inferred when considering alternate devices.

5.2 Tested Designs

This section provides a detailed description of every implemented design. Each design was tested and verified to function correctly through use of automated test benches. Each test bench instances a design for $n = 10$ and exhaustively tests every valid index value while verifying that:

1. The index produces a permutation
2. The produced permutation has not been produced by any previously tested index
3. The resulting permutation is held constant for a random number of cycles

5.2.1 Combinational Designs

As mention in Section 3.1, to represent current combinational designs, I2P was faithfully implemented. I2P's repeated block structure, Figure 5.1, leads to an easy to implement design that can be generated for any size n. The repeated block structure also easily allows for pipe-lining of the design by storing the state at the end of each block. To maximize the performance of the tested implementation, pipe-lining was used. The pipe-lined design

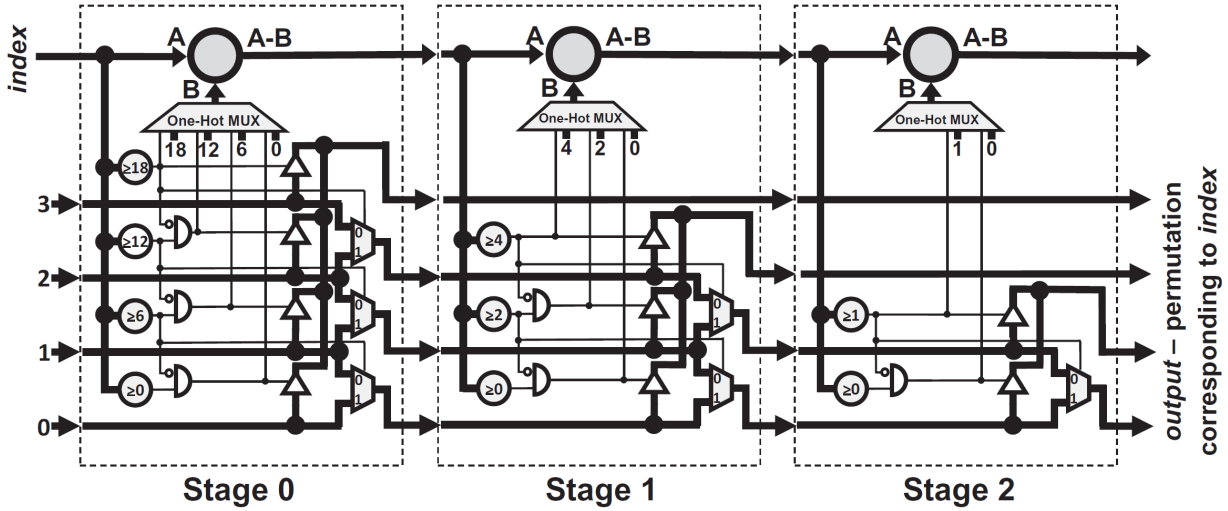

Figure 5.1: I2P four element permutation circuit [1]

results in a latency of $n-1$ cycles, and a throughput of one complete n element permutation per cycle. The index at each block is represented as a single value in the range $[0, n_r! - 1]$ as described above and is sized based on hard coded, predetermined values.

5.2.2 Knuth-Shuffle Designs

For the Knuth-Shuffle designs, three general approaches were implemented. One is a "naive" approach that simply utilizes a register array for memory. The other two approaches utilize Block RAM for memory but differ in how they manage output values. One uses a second dedicated Block RAM to store the output values, resulting in random access to the output permutation but with a read cycle delay and the limitation of only being able to read element values one at a time. The second utilizes a large shift register for the output permutation, and as such does not suffer any of the previous drawbacks but suffers from additional required area. For each design that utilizes Block RAM two variations of how to initialize the Block RAM also exist.

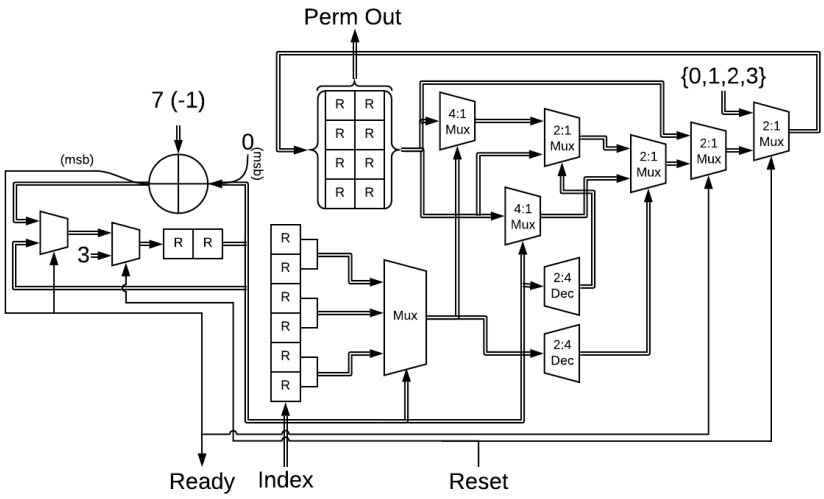

Figure 5.2: Knuth-Shuffle naive four element permutation circuit

Naive Approach (RAM-less)

Figure 5.2 provides an example circuit of the Knuth-Shuffle naive implementation for $n = 4$. The circuit has two inputs, the `index` and a `reset` signal. The `index` is of the form of a flat array of $n - 1$ elements each of size $\lceil \log_2 n \rceil$. The array holds the $n - 1$ digits necessary to represent a value $[0, n! - 1]$ in the factorial number system. The circuit also has two outputs. The output permutation of the circuit, `Perm Out`, is represented as a two dimensional array for simplification while in reality it is flat. The circuit also outputs a `ready` signal. This signal is held low while the permutation is being computed and is driven high during the first clock cycle the permutation is complete. This signal holds the results of `Perm Out` constant until the next `reset` signal is received and is output to allow for simpler integration into a larger design along with simpler exhaustive implementation testing. The circuit also features a `counter` of size $\lceil \log_2 n \rceil$.

The functionality of the circuit is straight forward as one would expect with a naive implementation. On `reset`, `counter` is initialized to $n - 1$ and the permutation is initialized to

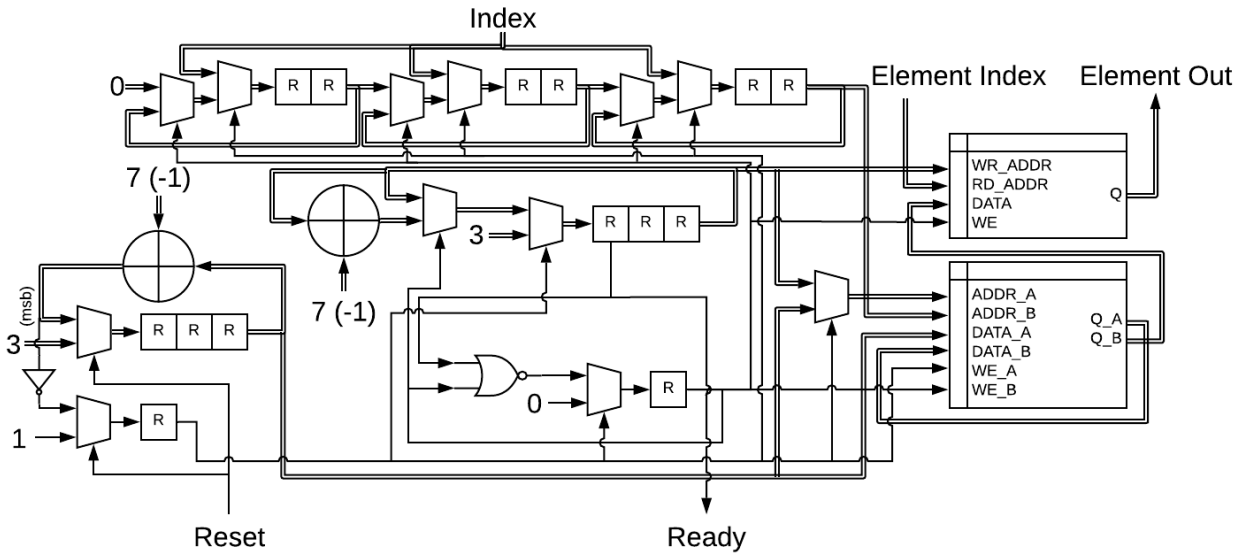

Figure 5.3: Knuth-Shuffle RAM output w/initialization four element permutation circuit

the identity permutation. On the rising clock edge of every cycle `counter` is decrease while the value of the permutation at `counter` and `index[counter]` (the index digit associated with counter's value) exchange locations. Once `counter` becomes zero, `ready` is driven high, which prevents further updates of `counter` and `Perm Out`.

RAM Based with Dedicated RAM for Output Values

Two variations of the Knuth-Shuffle with dedicated RAM output were implemented. One spends additional clock cycles to first initialize the RAM. The other implementation uses a "dirty bit" register for each RAM address and initializes values on the fly as needed while computing the permutation.

Figure 5.3 provides a circuit implementation of Knuth-Shuffle with dedicated RAM and initialization. The circuit can be deconstructed into several components; `counter_init`, `counter`, a shift register, a simple dual port RAM block and a full dual port RAM block. The full dual port RAM serves as a scratch pad to store the current state of the remaining

permutation as it is computed. The simple dual port RAM stores the completed permutation as it is computed and provides access to read out the permutation's elements. On `reset`, `counter_init` is initialized to $n-1$ and a `hold` register is set high. The `hold` register holds the remainder of the circuit in a reset state during initialization. During initialization, the write enable signal to `port A` of the dual port RAM is held high while it's address and data signals are `counter_init`. Each cycle `counter_init` is decreased, effectively initializing the full dual port RAM to the identity permutation. Once `counter_init` is zero, the next cycle the `hold` register is driven low, disconnecting `counter_init` from the full dual port RAM and starting the remainder of the circuit.

While the remainder of the circuit is held in reset, the index is stored into the shift register while `counter` is initialized to $n-1$. Once the circuit starts, it alternates between read and write cycles. On a read cycle, the values at the address of `counter` and the address of the next index digit are read on `port A` and `port B` of the full dual port RAM respectively. On a write cycle several actions happen. First, the shift register and `counter` are staged to update for the next read cycle. Secondly, the value from `port B` is written to address `counter` in the simple dual port RAM while the value from `port A` is written to `port B` which still points to the same address as on the read cycle. Once the counter reaches negative one, `ready` is driven high and the circuit is held in a read cycle to prevent any further memory writes.

The implementation without initialization is functional identically with the exception of how the full dual port RAM initialization is handled. The "dirty bit" logic is presented in Section 5.2.2 and can be easily applied in exchange for the initialization logic discussed here. As such, full details for the alternate implementation are exclude. It is also important to note that this design is distinct from the others as it carries additional constraints on its output permutation. While all other designs offer unrestricted random access to the resulting permutation, this design is limited to random access to a single element per cycle.

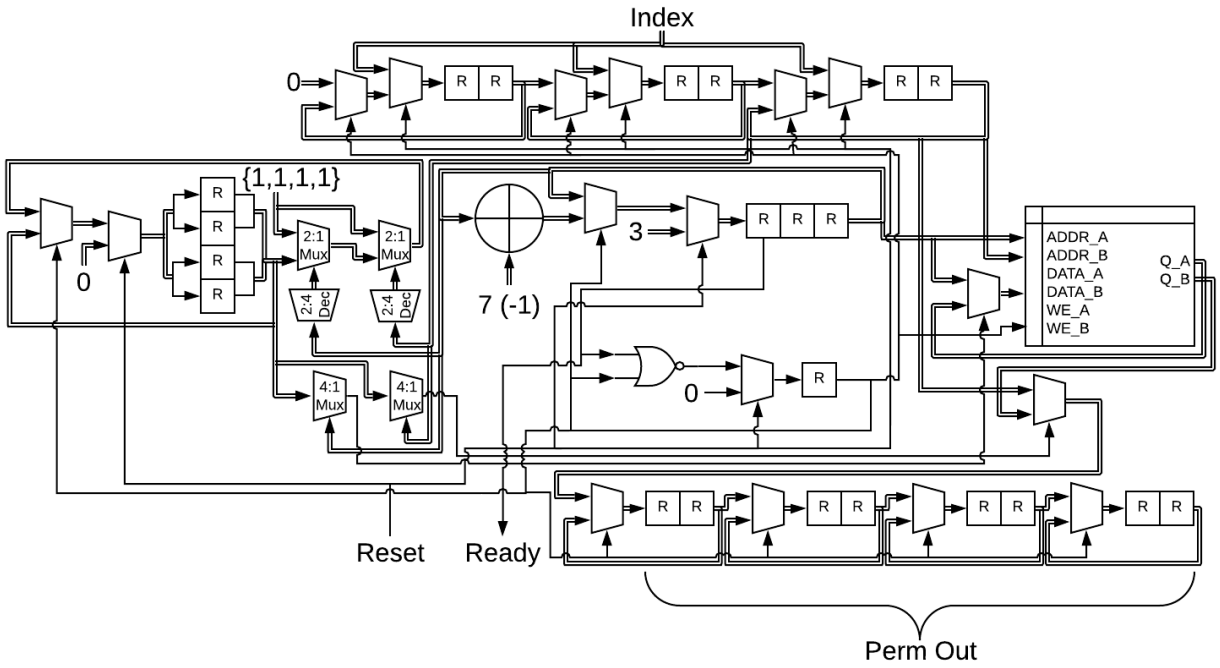

Figure 5.4: Knuth-Shuffle shift register output w/"dirty bit" logic four element permutation circuit

RAM Based with Dedicated Shift Register for Output Values

In an equivalent manner as with Knuth-Shuffle with dedicated RAM output, two variations of the Knuth-Shuffle with shift register output were implemented. The logic that demonstrates initializing the RAM block was covered in Section 5.2.2 and can be found within Figure 5.3. It is trivially to apply identical initialization logic in exchange for the "dirty bit" logic presented here. As such, the initialization implementation will not be covered.

Figure 5.4 provides a circuit implementation of Knuth-Shuffle with a shift register output that utilizes a "dirty bit" to handle memory initialization on the fly. The circuit can be decomposed into several main components; the "dirty bit" logic, a shift register for the input `index`, a shift register for the output permutation `Perm Out`, a full dual port RAM block and a `counter`. The "dirty bit" logic operates in a simple manner. For each permutation element, a "dirty bit" register exists. These registers are initialized to zero and are used to

track if the values at the element addresses in the RAM block are valid. Whenever an address is written to, it's corresponding "dirty bit" is driven high, indicating it is valid. When an address is read, if the corresponding "dirty bit" is zero, it indicates the address has yet to be written to and is not valid. For any non-valid address, the correct value is known to be the value to which it would have been initialized, which is simply the address.

On `reset` the "dirty bit" registers are all initialized to zero, the `counter` is initialized to $n-1$ and the `index` is side loaded into its shift register. As described in Section 5.2.2, the circuit then cycles between read and write cycles operating in a similar manner. On a read cycle, the values at addresses `counter` and the current `index` digit are read on `port A` and `port B` respectively. On a write cycle, several actions happen. The output shift register, `Perm Out` is advanced forward. If the "dirty bit" for the value at address `counter` is high, `port A` is written to `Perm Out` otherwise, `counter` is simply written to `Perm Out`. If the "dirty bit" for the value at the `index` digit's address is high, `port B` is written to `address A` otherwise, the `index` digit is simply written to `address A`. The "dirty bit" registers for `counter` and the index digit are then driven high, marking them as valid. Lastly, `counter` is staged to be decreased and the `index` shift register is staged to advance for the next read cycle. Once `counter` becomes negative one, the `ready` signal is driven high and the circuit is held in a read cycle preventing any further updates.

5.2.3 Knuth-Shuffle Design Cycle Count (Worst Case)

Table 5.1 contains the worst case number of cycles required to compute and read out an n element permutation for each Knuth-Shuffle implementation. The naive implementation swaps a single element per cycle starting at element $n-1$ and ending at element 1, totaling $n-1$ cycles. The initialized implementations spend n cycles initializing the RAM. Each

Table 5.1: Worst case cycle complexity to complete and read a permutation for Knuth-Shuffle designs

Design	Cycle Complexity
Naive	$n-1$
Initialized RAM Output	$4n$
"Dirty Bit" RAM Output	$3n$
Initialized Shift Register Output	$3n$
"Dirty Bit" Shift Register Output	$2n$

RAM implementation performs a swap every two cycles and must swap n elements, since element 0 must be manually written to output. In the worst case scenario, one would want to read elements in order from 0 to $n-1$. This requires an additional n cycles to read the values after the permutation is computed for the RAM output implementations. Totaling the applicable cycles for each RAM implementation gives the totals shown.

5.2.4 CRGE Designs

For CRGE designs, three alternate approaches were implemented. A basic "compact" approach that aims to reduce total register utilization by reusing registers used to compute the permutation to also store the resulting permutation. An approach that builds on the previous but also includes additional adder logic in aims of speeding up computation. Lastly, a straight forward approach that simply uses an additional shift register to store the output permutation.

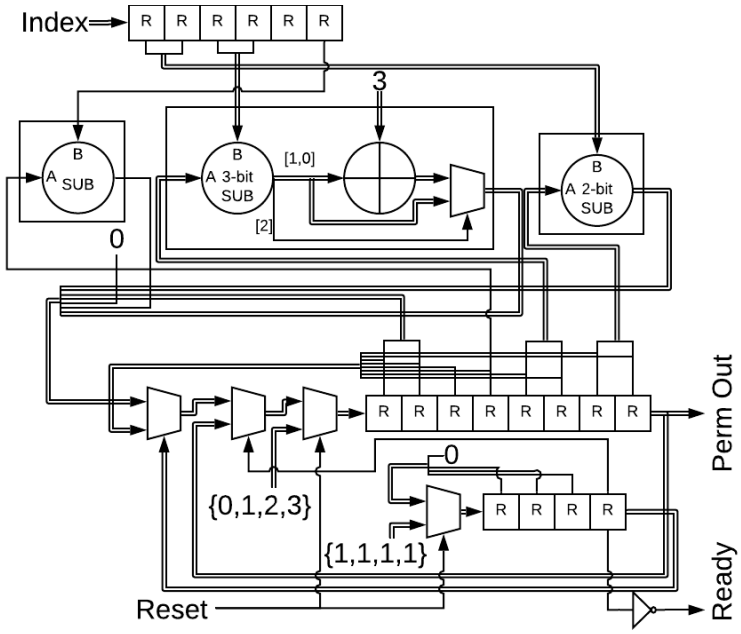

Figure 5.5: CRGE basic "compact" design four element permutation circuit

Basic "Compact" Implementation

Figure 5.5 provides a circuit implementation of the CRGE basic "compact" design for a four element permutation. For large n, the amount of registers required to store the resulting permutation alone becomes dominating. Using a similar amount of registers to store intermediate values during permutation computation could significantly increase the size of a resulting circuit. As such, this design aims to minimize register utilization by re-purposing the registers used to store intermediate values to also hold the resulting permutation. The circuit can be decomposed into the input index, $n-1$ computation blocks, a circular shift register to store the permutation and a shift register to manage enable bits for the computation blocks. To understand the computation blocks and the circuit as a whole, first the F function must be defined as:

$$f_i(x, d_i) = x - d_i \bmod (i+1)$$

Table 5.2: Input/output table of f_2 function for CRGE basic "compact" design

x	d_2	Output
0	0	0
0	1	2
0	2	1
1	0	1
1	1	0
1	2	2
2	0	2
2	1	1
2	2	0

It should be clear that this meets the requirements of an F function as outlined in Section 4.2. Thus, based on the results of Section 4.3, produces an ideal permutation generator. Each computation block computes a specific f_i. From left to right the computation blocks in Figure 5.5 represent f_1, f_2 and f_3. Each f_i function can be implemented as a subtractor, adder and multiplexer (MUX). The subtractor performs $x - d_i$ while the adder and MUX perform the mod$(i + 1)$ operation. The result of the subtractor will always be between $-i$ and i. If this result is negative then adding $(i + 1)$ is equivalent to performing the mod$(i + 1)$ operation otherwise, since $i < i + 1$ taking no action is equivalent to performing the mod$(i + 1)$ operation. For blocks where $i = 2^k - 1$ for some integer k the adder and MUX can be optimized away. In such blocks, if the subtractor result is negative then adding $i + 1 = 2^k$ is the equivalent to flipping the value's sign bit, thus simply ignoring this bit is equivalent to performing the mod $(i+1)$ operation. To further demonstrate the functionality of the computation blocks, Table 5.2 provides the output of f_2 for every possible input. On reset, the permutation is initialized to the identity permutation and the enable bits are all initialized to one. Each cycle, the computation blocks calculate a portion of an elements F function for an element currently located at static locations in the output permutation and the elements in the permutation rotate right circularly, effectively moving through each block required to calculate its F function. At the same time the enable bits are shifted

Table 5.3: Partial element calculated by each computational block by cycle for the CRGE "compact" design four element permutation

Cycle	f_1 Block	f_2 Block	f_3 Block
1	$f_1(1)$	$f_2(2)$	$f_3(3)$
2	$f_1(0)$	$f_2(f_1(1))$	$f_3(f_2(2))$
3	*disabled*	$f_2(f_1(0))$	$f_3(f_2(f_1(1)))$
4	*disabled*	*disabled*	$f_3(f_2(f_1(0)))$
5	*disabled*	*disabled*	*disabled*

right with a zero shifted in to disable the computation blocks in order, allowing already computed elements to bypass the blocks unmodified. Once all enable bits have shifted out, the ready signal is driven high and future updates are prevented. At this point all elements have rotated through the entire permutation and arrived back at their correct position. To better understand this behavior Table 5.3 contains the partial elements calculated by each block each cycle. For compactness, the d_i values passed to each function are omitted.

Precomputational Optimizations

Figure 5.5 provides a circuit implementation for the CRGE precomputational design for a four element permutation. In the CRGE "compact" design the critical path through a computational block is two adders and a MUX. The CRGE precomputational design aims to reduce this critical path to a single adder and a MUX, hopefully increasing the performance of the design.

This design is identical to the previously described "compact" design except for the F function used. In light of this, only the F function and optimization will be discussed here while Section 5.2.4 should be consulted for any additional details. The F function used in the CRGE precomputational design follows:

$$f_i(x, d_i) = x + d_i \bmod (i + 1)$$

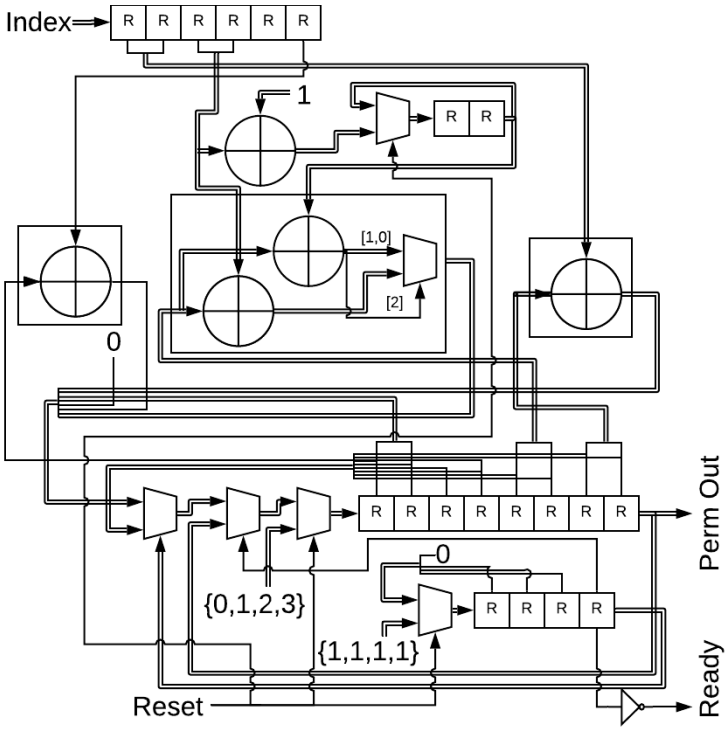

Figure 5.6: CRGE precomputational design four element permutation circuit

Similar to the case with the CRGE basic design, due to the possible ranges of x and d_i, the $\mathrm{mod}(i+1)$ operation equates to subtracting $i+1$ if $x+d_i$ exceeds i and taking no action otherwise. In attempts to reduce the critical path in a computation block, with the CRGE precomputational design both possible actions for the $\mathrm{mod}(i+1)$ operation are computed in parallel and later which action was correct is decided. Outside of the computation block, on reset $d_i-(i+1)$ is calculated and stored. Within the computation block, $x+d_i$ and $x+(d_i-(i+1))$ are computed in parallel. If the result of $x+(d_i-(i+1))$ is positive, it indicates the $\mathrm{mod}\,(i+1)$ operation would have resulted in a subtraction and $x+(d_i-(i+1))$ is the correct value. Otherwise the $\mathrm{mod}(i+1)$ operation would have taken no action and $x+d_i$ is the correct value. This process is further optimized through the realization of if $x+(d_i-(i+1))$ is positive there will be a carry to its most significant bit to unset the sign bit. Therefore, the sign bit in $d_i-(i+1)$ is simply ignored and only a check for a carry

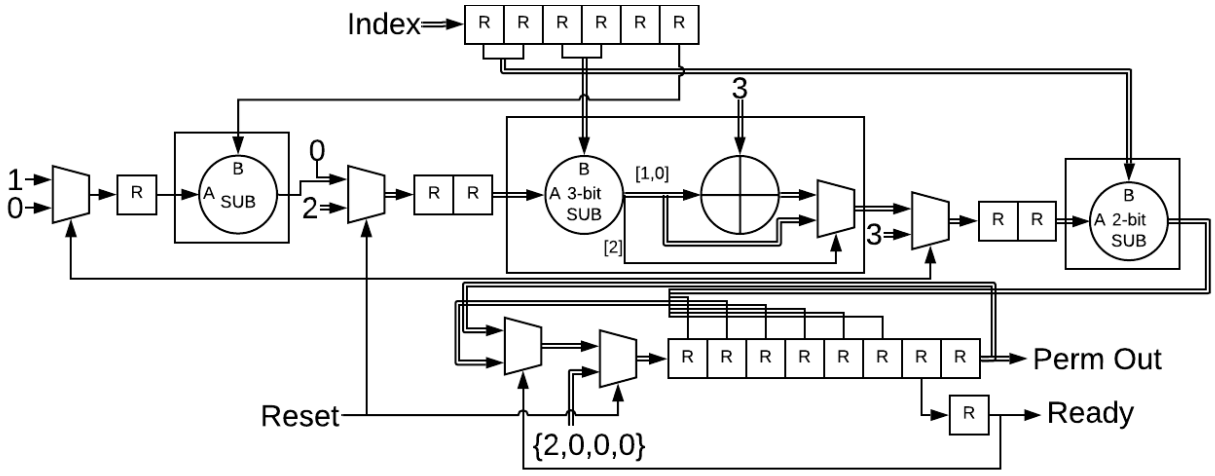

Figure 5.7: CRGE shift register design four element permutation circuit

out in $x + (d_i - (i + 1))$ is done. Following this logic, since the sign bit of $d_i - (i + 1)$ is of non-concern this operation can be optimized. For example, in Figure 5.6 $d_2 - 3$ is replaced with $d_2 + 1$. It can be easily verified that, when viewed from the two least significant bits, these operations are equivalent. Following similar logic in Section 5.2.4, the computation blocks where $i = 2^k - 1$ for some integer k can be optimized in an equivalent way to the CRGE basic design.

Dedicated Shift Register for Output Values

Figure 5.7 provides a circuit implementation for the CRGE shift register design for a four element permutation. The computational blocks in this design are identical to those of the basic "compact" implementation. Thus, the computational blocks will not be covered here but can be reference in Section 5.2.4. The CRGE shift register design implementation is quite straight forward. The output of each computation block is stored as an intermediate value which serves as input for the following computation block. The f_0 function is represented simply as a MUX that returns zero after any non-reset cycle. The final computation block, whose output is completed elements, is stored in a shift register instead of an intermediate

value. This shift register stores the computed permutation and serves as the output for the design. On reset, the intermediate values are initialized to represent the identity permutation and the output shift register is initialized to zeros except a single bit of the first element. Once this set bit reaches the final element of the shift register, it marks the final needed computation cycle. This sets the ready register on the next cycle which prevents further updates to the shift register.

Chapter 6

Results

This chapter provides the evaluation results of the area required and performance of each evaluated design. First, each approach is evaluated separately to compare implementations. Then, the best performing implementations for each metric and approach are used to compare the approaches. In all cases, performance and area are evaluated separately for small n and large n.

6.1 Evaluation of I2P Implementation

Figure 6.1 provides a graph of required area, based on the percent of device ALMs used, at $n \in \{2, 3, ..., 10, 16, 32, 64\}$ for the I2P implementation. The required area results for I2P are similar to those reported in [1], varying by at most three ALM for $n < 16$. However, this thesis' implementation requires significantly fewer ALMs at $n \geq 16$. No explicit implementation details are provided in [1], so it can only be speculated that these discrepancies arise from possible differences in tool chain versions, placement effort or hardware differences between the Stratix IV and Arria 10. Figure 6.1 does clearly indicate I2P has unmanageable growth in resource requirements for larger n as one would expect to find with any combinational approach. At $n = 16$ the design requires less than 1% of the boards total resources. By doubling n to 32, resource usage at least doubles to 2%. Another doubling of n to 64 causes resource usage to increase nearly ten fold to 21%. At $n = 128$, the build fails once the tool

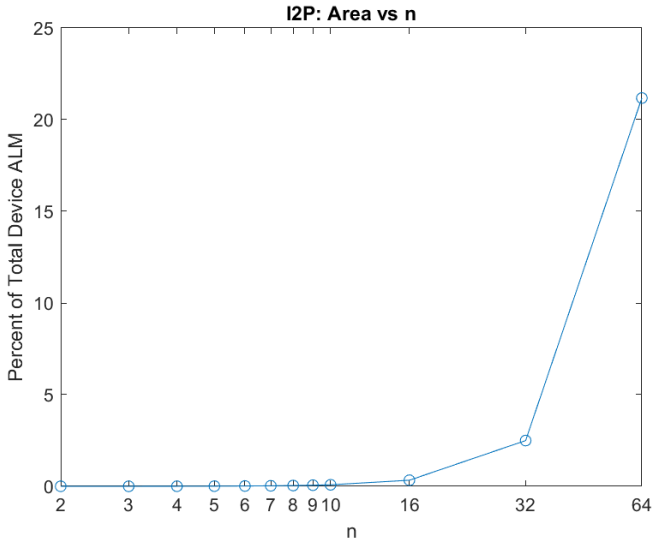

Figure 6.1: I2P implementation: area vs. n

chain realizes the design will require in excess of 150% of the boards resources. This appears to be a cut off in the tool chain where the developers believed there is no amount of effort that would be productive in fitting the design. Figure 6.2 provides a graph of maximum clock frequency at $n \in \{2, 3, ..., 10, 16, 32, 64\}$ for the I2P implementation. The results here follow a fairly similar trajectory as with the results in [1]. This thesis' implementation significantly outperforms the authors' of I2P for all n; however, the margin significantly decreases as n increases. For the smallest n, this thesis' implementation exceeds [1] by over 475%. This advantage decreases to slightly over 260% at $n = 8$ and ultimately is only near 140% at $n = 32$. Although the authors of [1] fail to report on $n = 64$, if this trend continues, the results would continue to converge. It is easier to explain this difference in performance as a result of differences in hardware as the transistor size of the Stratix IV is twice that of the Arria 10.

Based on the level of correlation between the presented results and the results reported in [1], the implementation used in this thesis appears to be a fair representative of I2P on the Stratix 10.

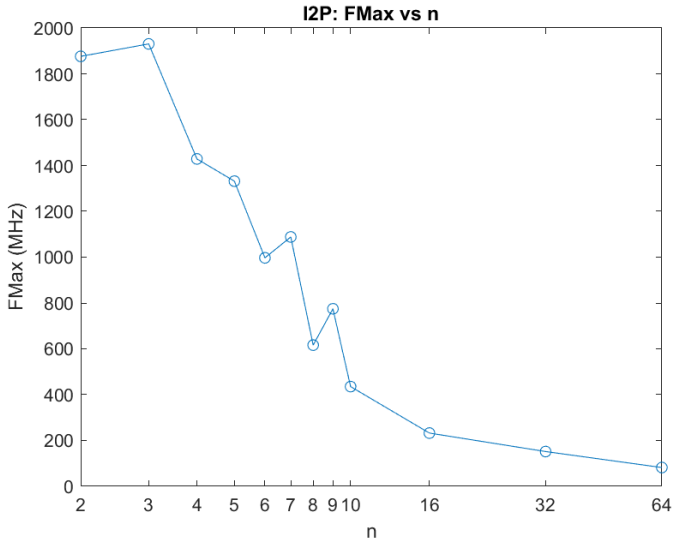

Figure 6.2: I2P implementation: maximum clock frequency vs. n

6.2 Comparison of Knuth Shuffle Implementations

Figure 6.3 provides a graph of required area, based on the percent of device ALMs used at $n \in \{2, 3, ..., 10, 16, 32\}$ for every Knuth-Shuffle implementation. Clearly, as one may have expected, the naive approach's required area grows significantly faster when compared to implementations that utilize Block RAM. Although insignificant, it is interesting to note at the smallest n the naive implementation is actually smaller than some of the Block RAM implementations.

Several trends with the Block RAM designs can also be noted. The designs with initialization are consistently smaller than their counterparts without initialization. This result should not be surprising as initializing the Block RAM uses both simpler logic and fewer registers than the "dirty bit" approach. When paired by initialization method, the shift register output designs are consistently and significantly larger than the Block RAM output designs. Again, this result should be expected as a shift register utilizes ALMs while a Block RAM does not. Figure 6.3 does seem to imply one results that may not have been expected. Since

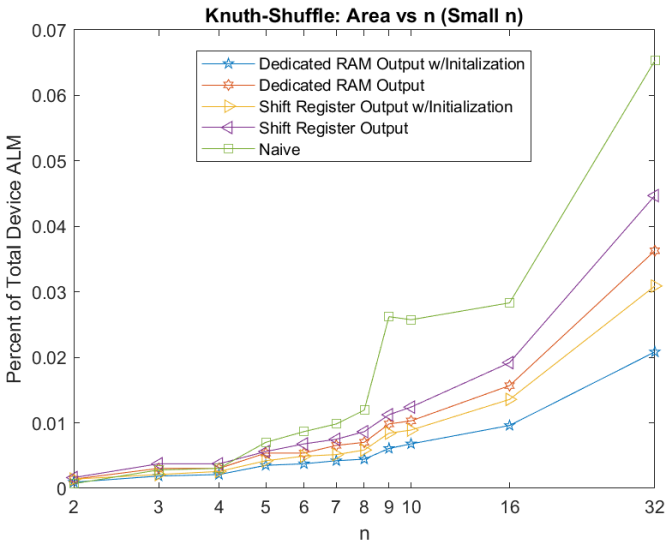

Figure 6.3: Knuth-Shuffle implementations: area vs. small n

the "dirty bit" Block RAM output design is consistently larger than the initialized shift register output design, it seems the "dirty bit" logic is larger than a shift register capable of storing the entire permutation. Figure 6.4 provides a graph of maximum clock frequency at $n \in \{2, 3, ..., 10, 16, 32\}$ for every Knuth-Shuffle implementation. For $n < 8$ the naive design is actually the fastest. This can be attributed to the fact that all the Block RAM designs are limited by the theoretical maximum RAM speed of 730 MHz [21]. Since the speed of the RAM designs remain nearly constant as n increases it is a further indication that RAM speed is the limiting factor in these designs. Also, the initialized Block RAM designs are faster than their "dirty bit" counterparts. As with the area results, this should be expected. Also, when paired by initialization method, the shift register output designs are faster than the Block RAM output designs. This can be attributed to RAM to RAM writes exacerbating the bottle neck in clock speed the RAM already causes. As with the area results, again Figure 6.4 seems to imply something interesting about the designs. The initialized Block RAM design and the "dirty bit" shift register output design have near identical results. This could indicate that the performance cost of the "dirty bit" logic is similar to the performance cost

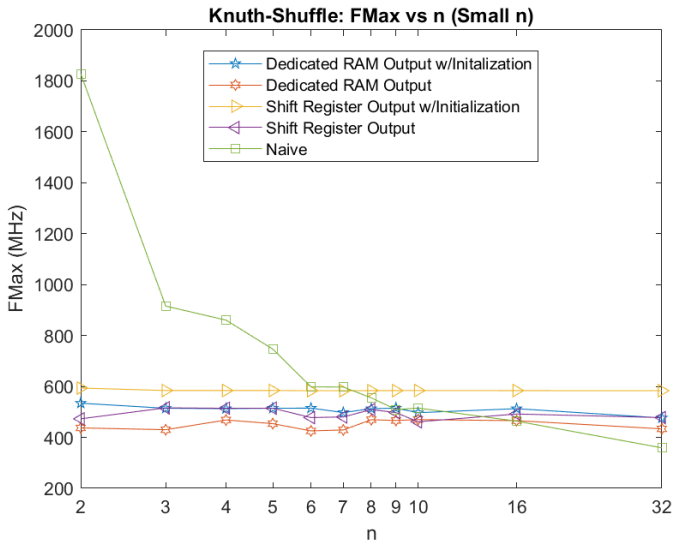

Figure 6.4: Knuth-Shuffle implementations: maximum clock frequency vs. small n

of the Block RAM's limiting RAM to RAM write. Figure 6.5 provides a graph of required area, based on the percent of device ALMs used at $n \in \{2^x | x \in \mathbb{Z}, 6 \leq x \leq 14\}$ for every Knuth-Shuffle implementation. The first thing that can be noted about these results are the naive implementation stops at $n = 2048$. While the total resource usage at this point is only at 11%, the Quartus tool chain still failed to complete any larger builds. The issue the tool chain faces with the naive implementation is with routing congestion. With routing effort exceeding 2.5, build attempt time becomes nearly unmanageable. At the same time, the tool chain is unable to reduce wire utilization below 150% between multiple regions. Although routing effort could be further increased, it is safe to conclude any resulting design will have severely degraded performance. Based on this, any larger naive designs will be assumed to have performance below any set practical threshold. In light of this, it is still possible to comment on the growth rate of the naive solution. At larger n the area of the naive solution continues accelerate its rate of divergence from the other designs and remains to be the largest design for all applicable n.

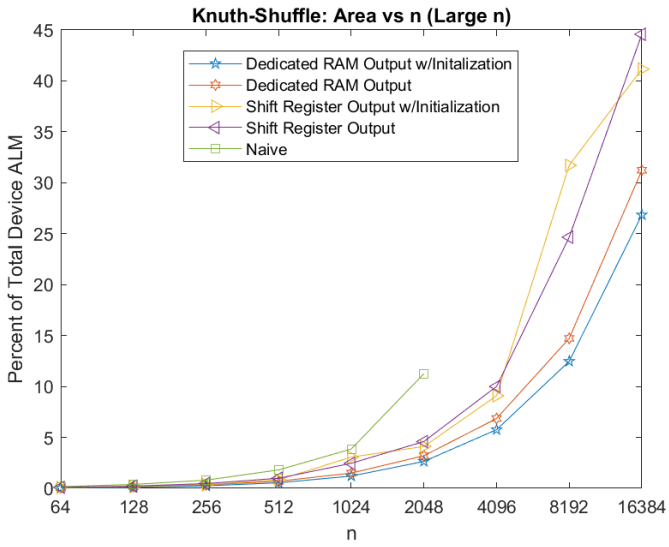

Figure 6.5: Knuth-Shuffle implementations: area vs. large n

For the Block RAM designs, most trends from smaller n continue. As before, the shift register output designs are larger than the Block RAM output designs. However unlike before, as n is increased both shift register output designs deviate from both Block RAM output designs; also, generally, the "dirty bit" designs are still larger than their initialized counterparts. As n is increased, though, this difference has become less pronounced and at $n = 8192$ the initialized shift register design is actually larger than the "dirty bit" shift register design.

Based on this data, the initialized RAM output design is the best choice for any n when minimizing resource usage is the primary focus. Figure 6.6 provides a graph of maximum clock frequency at $n \in \{2^x | x \in \mathbb{Z}, 6 \leq x \leq 14\}$ for every Knuth-Shuffle implementation. As previously addressed, the naive design was only able to produce results up to $n = 2048$. At this point the design is already only able to operate at roughly $\frac{1}{10}$ to $\frac{1}{7}$ of the clock frequency of all other design. This further supports the previous assumption that at larger n, the naive design would fail to produce any practical results and pursuing larger n builds is an unproductive use of computing resources.

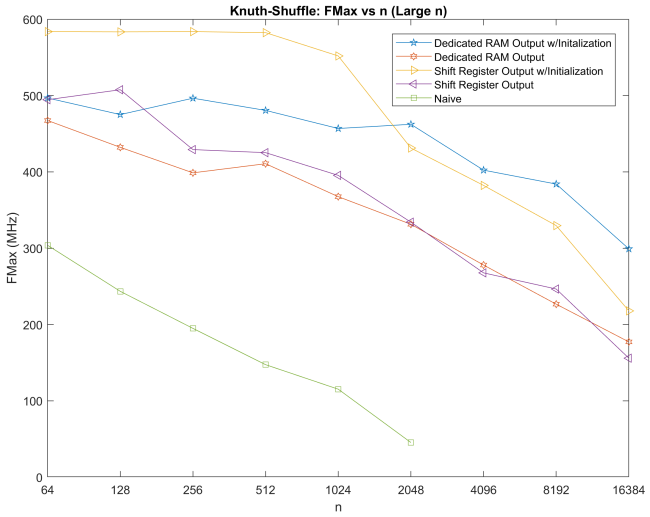

Figure 6.6: Knuth-Shuffle implementations: maximum clock frequency vs. large n

At larger n, the "dirty bit" designs continue to perform worse than their initialized counterparts. As n increases, eventually the RAM is no longer the limiting factor in each design. This seems to happen first for the "dirty bit" designs as they appear to dip in performance first and last for the initialized shift register output design as it remains at a constant frequency until $n = 1024$. While the initialized shift register output design initially continues to be the best performer, ultimately at $n > 1024$, the initialized Block RAM output design becomes the best performer. This seems to coincide with the performance bottleneck shifting away from the RAM. While initially, one may have assumed the initialized Block RAM output would be the best performer, one may also find it surprising how long it was outperformed.

When maximizing clock frequency is a priority; if $n \geq 1024$ then the initialized RAM output design is generally ideal otherwise the initialized shift register output is clearly the best choice.

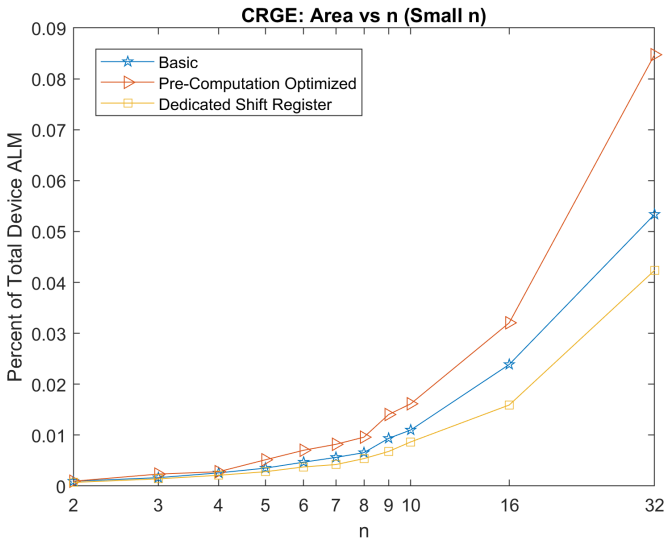

Figure 6.7: CRGE implementations: area vs. small n

6.3 Comparison of CRGE Implementations

Figure 6.7 provides a graph of required area, based on the percent of device ALMs used at $n \in \{2, 3, ..., 10, 16, 32\}$ for every CRGE implementation. At the smallest n the resource requirements for all three designs are fairly consistent. However, for $n > 4$ the precomputational design begins to diverge from the other two, growing more steeply. At $n > 8$ the other two designs begin to diverge from each other as well, with the basic "compact" design clearly outpacing the shift register design in growth. Figure 6.8 provides a graph of maximum clock frequency at $n \in \{2, 3, ..., 10, 16, 32\}$ for every CRGE implementation. At small n the basic "compact" and the shift register designs follow a very similar trend in performance with the shift register design generally being the better performer by a small margin. While across the same n, the precomputational design is clearly the worst performer. At $n > 8$ though, the performance of all three designs appears to converge. Figure 6.9 provides a graph of required area, based on the percent of device ALMs used at $n \in \{2^x | x \in \mathbb{Z}, 6 \leq x \leq 13\}$ for every CRGE implementation. The trend from smaller n continues with the precompu-

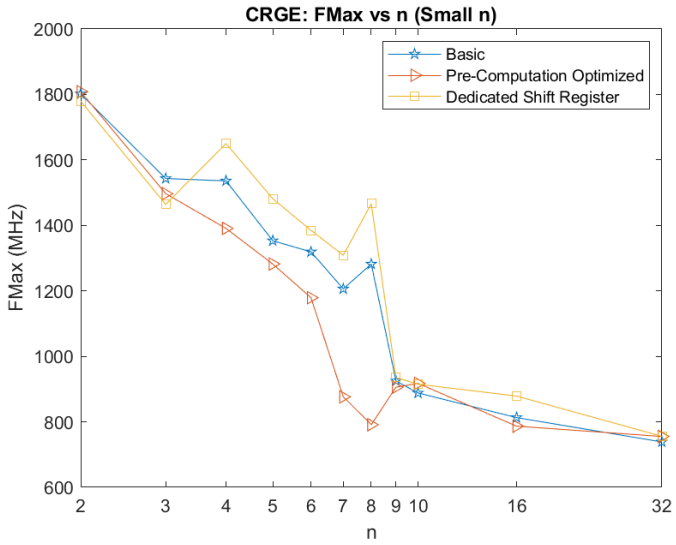

Figure 6.8: CRGE implementations: maximum clock frequency vs. small n

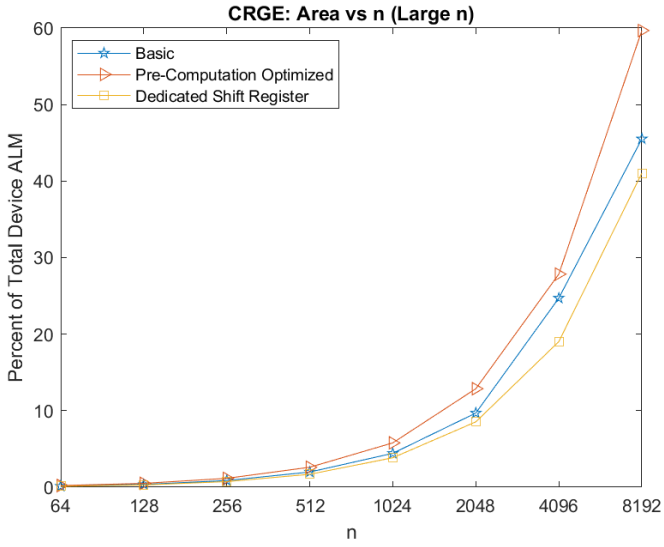

Figure 6.9: CRGE implementations: area vs. large n

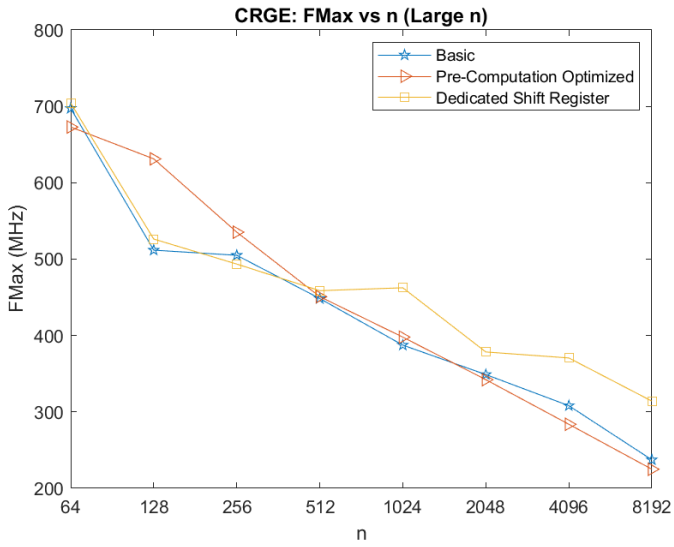

Figure 6.10: CRGE implementations: maximum clock frequency vs. large n

tational design being the largest and the shift register design being the smallest. However, the divergence between the designs originally observed is much less pronounced.

Based on these results, the shift register design is exclusively the best choice for any n when minimizing resource usage is the primary focus. Figure 6.10 provides a graph of maximum clock frequency at $n \in \{2^x | x \in \mathbb{Z}, 6 \leq x \leq 13\}$ for every CRGE implementation. At larger n the performance of all CRGE designs is surprisingly close with a few exceptions. At $n = 128$ and $n = 256$ the precomputatal design is the best performer by a reasonable margin. However, above $n = 512$ the shift register becomes the clear best performer by a large margin.

While no design is consistently the best choice for all n when maximizing clock frequency is a priority, the shift register design is generally the best choice.

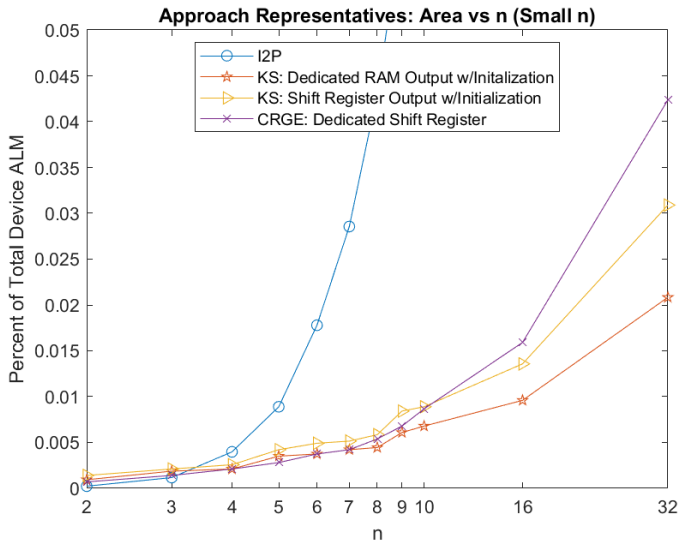

Figure 6.11: Approach representatives: area vs. small n

6.4 Comparison of Approaches

This section will compare the area and performance between the best representatives of each approach. For the cases where the Block RAM output Knuth-Shuffle design is the best representative, an additional Knuth-Shuffle design will be included in the comparison as well. This is done to account for the additional constraints the Block RAM output design carries since, it may not always be a viable option even if it is the best design for a given metric and n.

6.5 Comparison of AREA (Small n)

Figure 6.11 provides a graph of required area, based on the percent of device ALMs used at $n \in \{2, 3, ..., 10, 16, 32\}$ for the smallest representative implementations at small n for each approach. Here it is shown how aggressively the combinational approach, I2P, outpaces the other approaches in growth rate at small n. Granted, one must keep in mind that I2P has

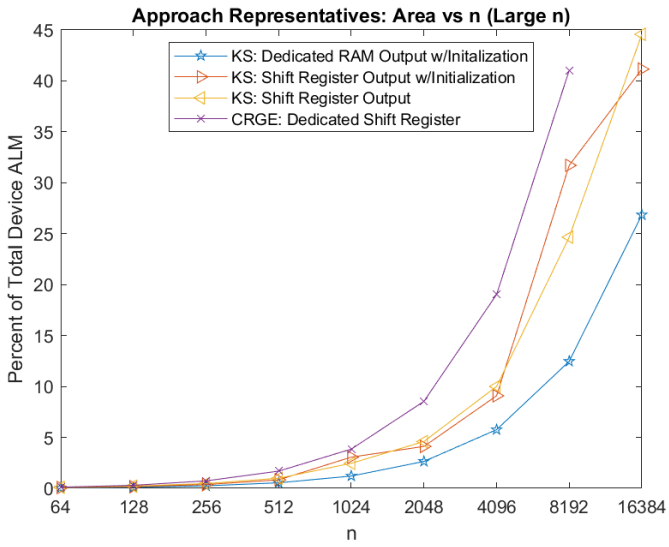

Figure 6.12: Approach representatives: area vs. large n

a throughput of one permutation per cycle while the other implementations take multiple cycles per element.

6.6 Comparison of AREA (Large n)

Figure 6.12 provides a graph of required area, based on the percent of device ALMs used at $n \in \{2^x | x \in \mathbb{Z}, 6 \leq x \leq 13\}$ for the smallest representative implementations at large n for each approach. These results should be expected. CRGE implemented with a shift register ranges from about 110% to 190% over the resource cost of the Knuth-Shuffle shift register output. The disparity in resource utilization is even worst when the CRGE implementation is compared with the initialized RAM output Knuth-Shuffle and the CRGE implementation ranges from a 240% to 330% increase in resource utilization.

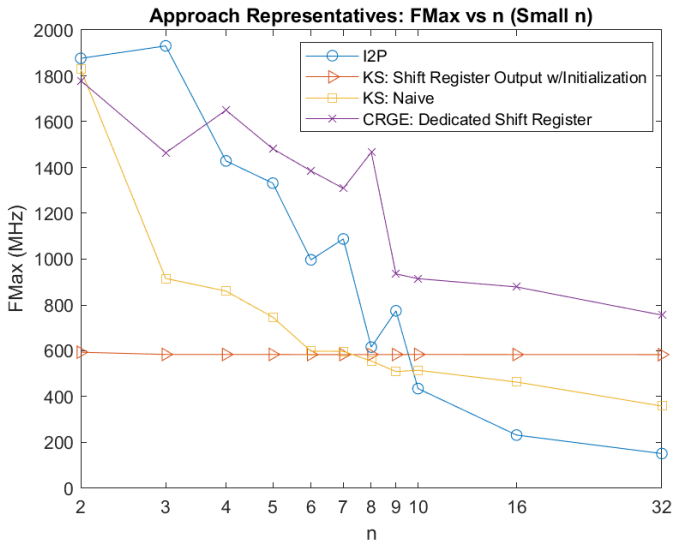

Figure 6.13: Approach representatives: maximum clock frequency vs. small n

6.7 Comparison of Performance (Small n)

Figure 6.13 provides a graph of maximum clock frequency at $n \in \{2, 3, ..., 10, 16, 32\}$ of the best performing representative implementations at small n for each approach. Here one can see, outside of the smallest n, CRGE implemented with a shift register out performs all other approaches. At $n \geq 9$, while the CRGE implementation significantly outperforms the naive Knuth-Shuffle implementation, they appear to degrade in performance at identical rates. It should also be noted, as n approaches 32, the performance advantage CRGE has over the shift register output Knuth-Shuffle implementation quickly approaches zero.

6.8 Comparison of Performance (Large n)

Figure 6.14 provides a graph of maximum clock frequency at $n \in \{2^x | x \in \mathbb{Z}, 6 \leq x \leq 14\}$ of the best performing representative implementations at large n for each approach. At $n \geq 256$, the performance of the CRGE implementations finally drops below the performance of the

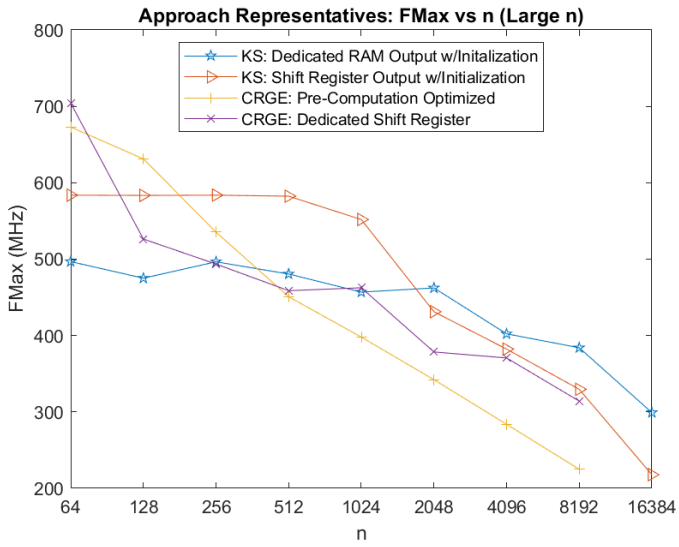

Figure 6.14: Approach representatives: maximum clock frequency vs. large n

initialized shift register output Knuth-Shuffle implementation. However, above $n = 1024$, the performance of CRGE implemented with a shift register nearly matches that of the Knuth-Shuffle shift register output.

6.9 Comparison of Running Time

Together, Figure 6.15 and Figure 6.16 provide graphs of the worst case running time to compute and read an entire permutation for the best performing representative of each approach at all n. Surprisingly, even though the initialized shift register output Knuth-Shuffle performed very well, outperforming most designs at most n, it turns out to perform far worse than the CRGE implementations and even the naive Knuth-Shuffle when running time is considered. At $n \geq 64$, CRGE is able to compute a permutation in 25% to 45% of the time it takes the best performing Knuth-Shuffle implementations. While CRGE is unable to produce permutations above 8192, one can still speculate about the results at higher n. It appears from Figure 6.16 that if the designs where able to build for larger n, the running time advantage CRGE provides would only continue to grow.

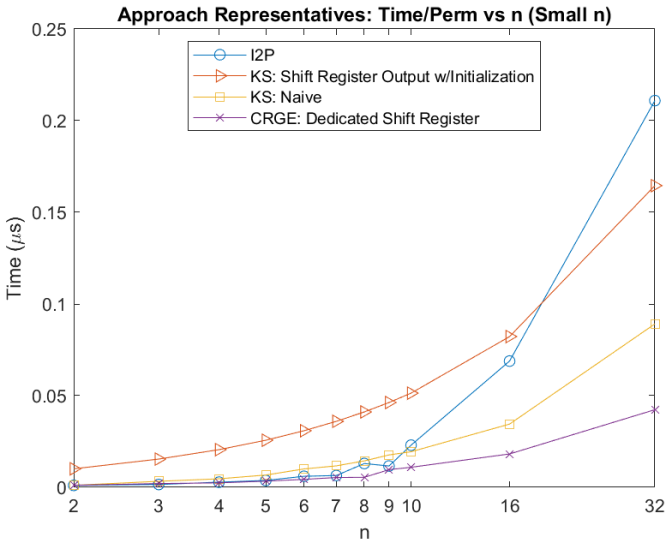

Figure 6.15: Approach representatives: worst case running time vs. small n

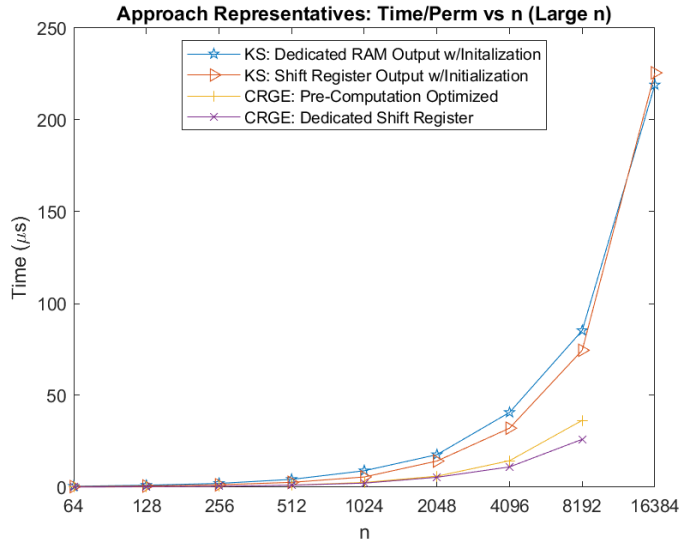

Figure 6.16: Approach representatives: worst case running time vs. large n

Chapter 7

Additional CRGE Uses and Future Work

CRGE has several additional advantages over other permutation methods yet to be explored. Here high-level proposals for additional models that can exploit these additional advantages are demonstrated.

7.1 Partial Permutation Model

Figure 7.1 presents a high level template for a design to use CRGE to produce partial permutations. Details specific to a chosen F function were omitted for clarity and flexibility but should be easy to deduce from a full permutation design. For each of the desired elements

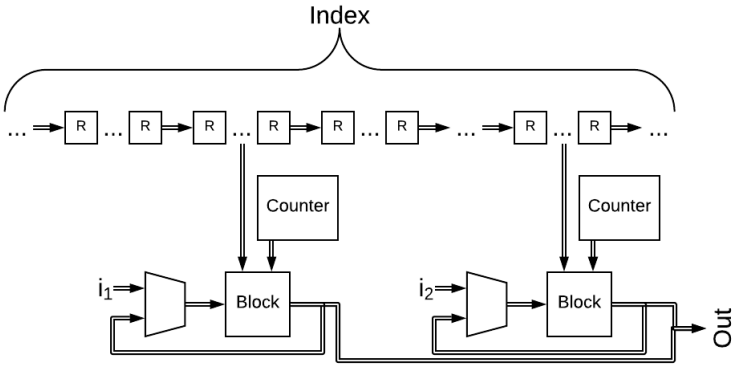

Figure 7.1: CRGE partial permutation design

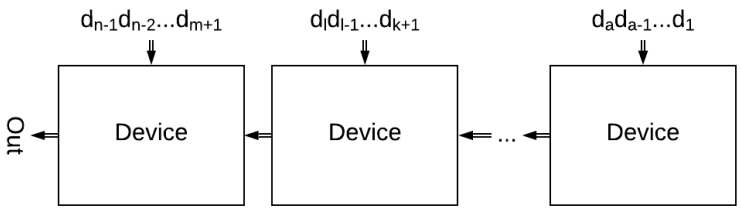

Figure 7.2: CRGE distributed design

in the partial permutation, a block and counter are instanced. The input index is side loaded into a shift register such that $d_{n-1} = 0$ and $d_i = d_{i+1}$ on each cycle update. A block for element i takes d_i in the index as input. On reset, the block also takes i as an input, otherwise it takes its previous output. The counter is used to adjust the functionality of the block each cycle to match the functionality of $f_i, f_{i+1}, ..., f_{n-1}$ and may vary in implementation details based on the specific F function used. For example, to match the functionality of the F function used in the CRGE shift register design presented in Section 5.2.4 the counter would serve as the modulus input to the block's adder. It can easily be confirmed that this design results in each block computing $f_{n-1}(...(f_{i+1}(f_i(i, d_i), d_{i+1})...), d_{n-1})$ as desired.

7.2 Distributed Model

Figure 7.2 presents a high level template for a distributed design intended for computing permutations of very high n. This design is intended to compute permutations where n is too large for a complete implementation or possibly even a complete index to fit on a single device. Each device instead follows a design similar to the CRGE shift register design presented in Section 5.2.4 but only implements a portion. Each device implements a continuous subset of the blocks and only receives the index digits associated with its blocks. Instead of a device storing the output of its ending block in a shift register, the output is provided to the next device where it serves as input to the next device's first block.

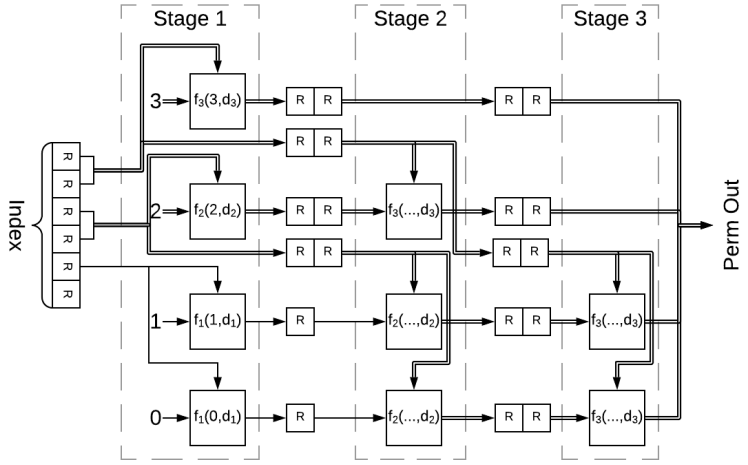

Figure 7.3: CRGE high throughput design

The final device outputs the permutation elements as they are computed. To maximize the performance of the system additional analysis would be required to find an optimal distribution of blocks between devices. This thesis predicts that attempting to balance peak performance with minimizing the number of devices required for a particular n could be achieved by decreasing the amount of total device utilization in proportion to the size of the largest block on a particular device. This will result in the first few devices implementing a large number of blocks at or near device capacity and then having the amount of utilization decrease for subsequent devices as the size of the implemented blocks increases.

7.3 High Throughput Design

Figure 7.3 presents a high throughput CRGE design template for four element permutations. As with the previous additional CRGE designs, implementation specifics and details were omitted for clarity and flexibility. Each stage, s, receives the index digits $d_{n-1}, d_{n-2}, ..., d_s$ as input and computes the next f_i for each permutation element. Section 4.3 showed $f_0(0, d_0)$ must be zero and thus computing element zero starts with $f_1(0, d_1)$. Although no proper

analysis was conducted for this design accurate speculations may still be possible. The performance should very closely align with that of the CRGE shift register design for the same n, since they most likely have equivalent critical paths. The resource requirements are a bit harder to estimate but if one counts the number of block instances one finds $n + (n-1) + (n-2) + ... + 2 \approx n(n+1)/2$. If the case of $n = 32$ is considered, one can set a fair upper limit by considering the resource utilization for $n = 32 \cdot 33/2 \approx 512$ of the CRGE shift register design. This will be an over estimate since at $n = 512$ the blocks are significantly larger. Using this over estimate, based on the results in Figure 6.11, Figure 6.13 and Figure 6.9 one can predict the high throughput CRGE design should significantly exceed the performance of I2P and require fewer resources while still meeting the permutation per cycle throughput I2P offers.

7.4 Software Implementation

While the focus of this thesis has been on reconfigirable hardware implementations, this limits CRGE's use on conventional computers that lack specialized hardware. However, with proper F function selection, CRGE's design has the potential to lend itself well to vector processors. Examples of vector processors can be found very commonly in even the most mundane of modern conventional computers through processor instruction set extensions, graphical coprocessors, and graphical processing units (GPU). Further research is needed in this area to determine the performance of CRGE in such environments when compared with the performance of a software implementation of Knuth-Shuffle. Since Knuth-Shuffle is the undeniable dominating method in software, it is the only current method needed to be considered in such a comparison.

Chapter 8

Conclusion

This thesis proposed an alternate approach to generating unbiased permutations.

In Chapter 2, the relevant background mathematics required to fully understand the problem and described solutions was provided. The problem was also clearly defined and constrained by defining an ideal permutation generator.

In Chapter 3, current popular methods for generating unbiased permutations were covered. Memory-less approaches along with their advantages and disadvantages were discussed. I2P, the selected example of a memory-less approach, was described and I2P was confirmed to qualify as an ideal permutation generator. The history and description of the Knuth-Shuffle was also provided. The Knuth-Shuffle's advantages and disadvantages were then discussed and Knuth-Shuffle was confirmed to qualify as an ideal permutation generator. The chapter ended by describing the Random-Sort method and demonstrating that it failed to qualify as an ideal permutation generator.

In Chapter 4 the proposed method, CRGE was presented. First, the chapter described how CRGE was developed and then provided a formal mathematical description of CRGE. Lastly, the chapter provided a formal mathematical proof that CRGE was indeed an ideal permutation generator.

In Chapter 6, the results of the evaluations outlined in Chapter 5 were presented and discussed. It was shown that outside of a few edge cases the shift register implementation was

surprisingly the best CRGE implementation in terms of both area and performance. This thesis showed that CRGE significantly outperforms I2P in terms of both area and performance. Further, this thesis showed that although CRGE has a growth complexity constant slightly below twice that of Knuth-Shuffle and operates at a clock speed slightly below Knuth-Shuffle it is able to produce permutations significantly faster than the Knuth-Shuffle implementations.

In Chapter 7, templates for implementing CRGE to produce partial permutations using minimal area, to distribute designs for permutations that exceed device resource limitations across multiple devices and to achieve a throughput of one permutation per cycle were provided. All such achievements being impossible with Knuth-Shuffle due to the inter-element dependencies of the Knuth-Shuffle algorithm. Therefore, based on this fact combined with the results drawn from the data in Chapter 6 this thesis concludes that the proposed method, CRGE, to be an optimal unbiased permutation generator compared to current popular methods.

Bibliography

[1] J. T. Butler and T. Sasao, "Hardware index to permutation converter," *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, May 2012.

[2] National Institute of Standards and Technology, *FIPS PUB 46-2: Data Encryption Standard (DES)*. Dec. 1993.

[3] National Institute of Standards and Technology, *FIPS PUB 197: Advanced Encryption Standard (AES)*. Nov. 2001.

[4] J. Takala and T. Jarvinen, "Stride permutation access in interleaved memory systems," 11 2003.

[5] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000.*

[6] M. Waechter, K. Hamacher, F. Hoffgaard, S. Widmer, and M. Goesele, "Is your permutation algorithm unbiased for n≠2m?," pp. 297–306, 2012.

[7] D. M. Burton, *The history of mathematics: an introduction*. McGraw-Hill, 2011.

[8] J. S. Tanton, *Encyclopédia of mathematics*. Facts On File, 2008.

[9] A. Shell-Gellasch and P. J. Freitas, "When a number system loses uniqueness: The case of the maya," *Mathematical Association of America*, 2012.

[10] D. E. Knuth, *"Art of computer programming. Volume 2, Seminumerical algorithms"*, pp. 145–146. Addison-Wesley, 1997.

[11] R. A. Fisher and F. Yates, *Statistical tables for biological, agricultural, and medical research*, pp. 37–38. Hafner Pub. Co., 1963.

[12] R. Durstenfeld, "Algorithm 235: Random permutation," *Communications of the ACM*, vol. 7, no. 7, p. 420, 1964.

[13] S. Saeed, M. Sarosh Umar, M. Athar Ali, and M. Ahmad, "Fisher-Yates Chaotic Shuffling Based Image Encryption," *arXiv e-prints*, Oct 2014.

[14] S. Saeed, M. S. Umar, M. A. Ali, and M. Ahmad, "A gray-scale image encryption using fisher-yates chaotic shuffling in wavelet domain," pp. 1–5, May 2014.

[15] T. K. Hazra and S. Bhattacharyya, "Image encryption by blockwise pixel shuffling using modified fisher yates shuffle and pseudorandom permutations," pp. 1–6, Oct 2016.

[16] T. K. Hazra, R. Ghosh, S. Kumar, S. Dutta, and A. K. Chakraborty, "File encryption using fisher-yates shuffle," pp. 1–7, Oct 2015.

[17] S. Alam, S. M. Zakariya, and N. Akhtar, "Analysis of modified triple - a steganography technique using fisher yates algorithm," pp. 207–212, 2014.

[18] M. Tayel, G. Dawood, and H. Shawky, "Block cipher s-box modification based on fisher-yates shuffle and ikeda map," pp. 59–64, Oct 2018.

[19] V. Shokeen, M. Yadav, and P. Kumar Singhal, "Comparative analysis of flr approach based inverse tree and modern fisher-yates algorithm based random interleavers for idma systems," pp. 447–452, Jan 2018.

[20] M. Yadav, P. R. Gautam, V. Shokeen, and P. K. Singhal, "Modern fisher–yates shuffling based random interleaver design for scfdma-idma systems," *Wireless Personal Communications*, vol. 97, no. 1, p. 63–73, 2017.

[21] Intel, "Intel arria 10 device datasheet," November 2018.