Optimizations and Hardware Implementations for Composited de Bruijn Sequence Generators

by

Bo Yang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

A binary de Bruijn sequence with period 2^n is a sequence in which every length-n subsequence occurs exactly once. de Bruijn sequences have randomness properties that make them attractive for pseudorandom number generators. Unfortunately, it is very difficult to find de Bruijn sequence generators with large periods (*e.g.*, 2^{64}) and most known de Bruijn sequence construction techniques are computationally quite expensive. In this thesis we present a set of optimizations that reduces the computational complexity of the de Bruijn sequence generators constructed by the *composited construction* technique, which is the most effective one we know. We call optimized composited de Bruijn sequence generators "OcDeb". An original (k, n)-composited de Bruijn sequence generator generates a sequence with period 2^{n+k} and uses $\mathcal{O}(k^2 + nk)$ bit operations. Our optimizations reduce this to $\mathcal{O}(k\log(k) + \log(n))$ operations, allow retiming, and enable parallel implementations that produce multiple bits per clock cycle while reusing some combinational hardware. Our optimizations are formulated in lemmas and theorems with proofs. The benefits of OcDeb-*k*-*n* over (k, n)-composited de Bruijn sequence generators are demonstrate by comprehensive results in a 65nm CMOS ASIC library. For example, before place-and-route, an instance of OcDeb-*32*-*32* has a period of 2^{64}, an area of 656 GE and a maximum performance of 1.67 Gbps, representing 1.7× and 29.4× improvement on area and performance respectively over the previous implementation method presented by Mandal and Gong; with parallelization, this instance can achieve 8.30 Gbps with an area of 1229 GE. An instance of OcDeb-*512*-*32* has a period of 2^{544}, an area of 7949 GE, and a maximum performance of 1.43 Gbps.

Acknowledgements

I would like to thank my supervisor Mark Aagaard for all the help and support. He is a great supervisor. I would also like to thank Prof. Guang Gong for the meaningful discussions we had about this work and Kalikinkar Mandal for the patient explainations on composited de Bruijn sequence generators. Many thanks to Yin Tan for answering my questions about cryptography and math and Nusa Zidaric for the careful proof reading and help with Inkscape. Finally, I want to thank my parents for the endless love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Random number generators (RNGs) are a critical building block for security. They are used to generate secret keys, initialization vectors, padding values, and random masks for applications in mutual authentication, digital signatures, symmetric key cryptography public key cryptography, and side-channel-attack countermeasures.

Low quality random numbers can be the weak link that compromises the overall security of a system. For example, due to the poor quality of the RNG in Sony's PlayStation 3, an attacker found two digital signatures that used the same random number, which then allowed the attacker to find the master key of the PS3, and thereby allowed pirated software to be run on the PS3 [5]. Due to a misconfiguration of the random number generator in South Korea's Citizen Digital Certificate smart card, researchers were able to recover the private keys of 184 cards [1].

Random number generators can be characterized as either non-deterministic (true random number generators, TRNGs) or deterministic (pseudo random, PRNG). TRNGs rely on variability effects in manufacturing, environmental conditions, and aging. PRNGs are algorithms designed to satisfy statistical properties of randomness.

Two characteristics of the quality of a random number generator are the period of the sequence it generates and the "randomness" of the numbers in the sequence. A key design characteristic of a PRNG is the number of bits of internal state, because an n-bit deterministic system can have a maximum period of 2^n. The ultimate mathematical definition of randomness, Kolmogorov complexity [26], defines the randomness of a string to be the size of the smallest program that can generate the string. Kolmogorov complexity is uncomputable. Practical assesments of randomness include properties that can either be proved mathematically or evaluated empirically by analyzing long sequences of numbers.

Common empirical tests include Marsaglia's DIEHARD from 1995 [32, 33], L'Ecuyer and Simard's TESTUO1 [23] from 2007, and NIST's SP800-22 from 2010 [44]. The NIST tests are aimed specifically at cryptographically secure PRNGs.

For anything other than small PRNGs, statistical tests can give only approximate answers. The long debate about a possible trapdoor in Dual_EC_DRBG before it was finally discredited [40, 22, 3]; and the weaknesses in the Sony PS3 and South Korean identity cards all demonstrate that analyzing PRNGs is very difficult.

PRNGs are often built from feedback shift registers. Linear feedback shift registers (LFSRs) have guaranteed periods, but poor randomness. The behaviour of nonlinear feedback shift registers (NLFSRs) is much more complicated and there are relatively few theoretical results. For most NLFSRs, both the period and the randomness must be estimated empirically.

De Bruijn sequences have guaranteed mathematical attributes of randomness: balance, tuple-distribution, and high linear complexity. These randomness properties make de Bruijn sequence generators attractive for PRNGs. Unfortunately, finding a de Bruijn sequence generator with a long period is computationally infeasible: the largest generator found so far has period of $2^{32} - 1$ [14]. Algorithmic techniques to construct de Bruijn sequence generators result in PRNGs that are unreasonably large and slow. The best known technique is Mandal and Gong's composited construction [28], which also has the advantage of having security analysis.

In this thesis, we focus on optimizing and implementing composited de Bruijn sequence generators. Our optimized composited de Bruijn sequence generators, hereafter called OcDeb, are functionally equivalent to Mandal and Gong's composited construction, thereby inheiriting their randomness properties and security analysis, but are significantly more efficient: achieving a $51\times$ improvement in performance/area for a generator with period 2^{64}. Asymptotically, the complexity (number of bit operations) is reduced from $\mathcal{O}(k^2 + nk)$ to $\mathcal{O}(k \log(k) + \log(n))$.

The thesis begins with an overview of de Bruijn sequences construction techniques and focuses on the composited construction technique in Chapter 2. The main body of the thesis consists of the optimizations that reduce the number of bit operations, the retiming optimization and the optimization for efficient parallelization in Chapter 3. Details on implementation including discusion on two tricks of implementation are in Chapter 4. Comprehensive hardware implementation results of OcDeb family are in Chapter 5. For future work, please go to Chapter 6.

Chapter 2

Background

We gather background knowledge necessary to understand this thesis. Section 2.1 introduces de Bruijn sequences and especially our notations that we will use consistently across the whole thesis. Section 2.2 is a literature review for algorithms to generate de Bruijn sequences. Composited construction stands out for its superior complexity in bit operations. Section 2.3 details it.

2.1 Preliminaries

Our notation and definitions for Galois fields and vectors are shown below. Sequences in this thesis all have a binary alphabet.

$\mathbb{F}_2$	The Galois field with two elements $\{0, 1\}$.
$\mathbb{F}_{2^m}$	The Galois field with 2^m elements.
$+$	Addition over $\mathbb{F}_2$ (XOR).
$[j, i]$	The set of natural numbers from j downto i, inclusive.
x_i	The i^{th} element in vector x.
$x_{[j:i]}$	The vector consisting of elements j *down to* i of vector x.
$x_{[:i]}$	The vector consisting of elements from the end of x down to i.
iff	A shorthand for if and only if.
$\mathbb{Z}$	The integer set.
Z_o^n	A set of odd integers from 1 to n.
Z_e^n	A set of even integers from 1 to n.
$v^{(i)}$	v's value after i clock cycles. v can be either a function or a variable.
$\log()$	The logarithm's base is 2 unless otherwise stated.

To make our math more consistent with our figures, we write vectors and ranges of natural numbers in *decreasing* order. To minimize notational clutter, we overload operations on elements of $\mathbb{F}_2$, Booleans (truth values), and predicates (functions that return truth values).

Definition 1. *A binary sequence (x) with period 2^n is a* de Bruijn sequence *iff every length-n subsequence ($x_{[i+n-1:i]}$) occurs exactly once in one period.*

Definition 2. *[15] A binary sequence (x) with period $2^n - 1$ is a* span-n sequence *iff every* non-zero *length-n subsequence ($x_{[i+n-1:i]}$) occurs exactly once in one period. It is also called a* modified de Bruijn sequence *iff it is obtained from a de Bruijn sequence with period of 2^n by deleting one of the 0s in the subsequence of n 0s.*

For example, **0**0011101**0**0011101... is a de Bruijn sequence with period 8 because every subsequence of length 3, which are $000, 001, 010, 011, 100, 101, 110, 111$, occurs exactly once in one period. By removing one of the 0s in the subsequence of 3 0s, we have **0**0111010011101.... It is called modified de Bruijn sequence, which is also a span-n sequence.

2.2 Algorithms to Construct de Bruijn Sequence Generators

Generating de Bruijn sequences has been a mathematical problem studied mainly in two directions, generating all or as many as possible de Bruijn sequences and efficiently generating de Bruijn sequences. A graphical approach was used by de Bruijn to prove that the total number of different de Bruijn sequences with a period 2^n is $2^{2^{n-1}-n}$ in [4]. Approaches that are able to generate all de Bruijn sequences can be found in [18, 36, 37, 27]. In this thesis, we are more interested in feedback shift register(FSR) based algorithms because of their efficiency. FSR (shown in Figure 2.1) consists of a series of FLIP-FLOPs connected in serial and a feedback function f generating new values for the shift register. The feedback function f may take any subset of the FLIP-FLOPs as inputs. If f is linear, the FSR is also called linear feedback shift register (LFSR). Otherwise, it is called non-linear feedback shift register (NLFSR).

A simple such algorithm to generate a de Bruijn sequence is appending 0 after $n-1$ consective 0s in a span-n sequence of a period $2^n - 1$. This can be achieved by adding a product of negation of $n-1$ stages of the n-stage shift register to the feedback function. This approach is not suitable for cryptographic applications, and the reasons are twofold. If the span-n sequence is generated by an LFSR, a string of $(2n+1)$ consecutive bits is sufficient

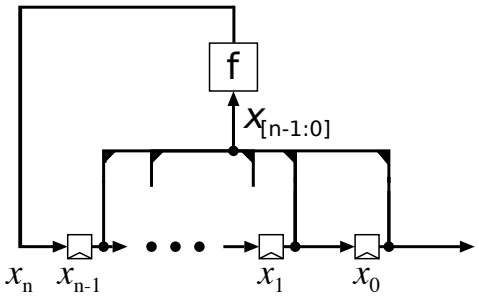


Figure 2.1: Generic feedback shift register

for an adversary to generate the complete sequence of 2^n bits. If the span-n sequence is generated by an NLFSR, the period of de Bruijn sequence is not large enough for most of the cryptographic applications. It is because NLFSRs are acquired by exhaustive search and the maximum period found so far is $2^{32} - 1$ [14].

Fredricksen and Kessler in [11] presented an algorithm based on lexicographic compositions which requires a storage unit linear in n. Later, Fredricksen and Maiorana proposed a more general technique to generate k-ary de Bruijn sequences of period k^n [12]. A similar algorithm is in [42] with modest computational advantage.

Cycle joining is one of the well-known tools for generating de Bruijn sequences in which a de Bruijn sequence is constructed by joining a number of cycles produced by a feedback shift register [21, 19, 25]. In [21], Jansen *et al.* presented a cycle joining algorithm for generating de Bruijn sequences whose feedback function is the sum of the original feedback function and a feedback function for joining the cycles of the original feedback function. The storage requirement for this method is $3n$ bits, and each bit of the de Bruijn sequences is generated within $4n$ cycles.

Another technique used for generating de Bruijn sequences is using D-morphism [24] by Lempel, which constructs a de Bruijn sequence of period 2^{n+1} as follows. First, two D-morphic preimages of the de Bruijn sequence of period 2^n are computed and then, the de Bruijn sequence of period 2^{n+1} is obtained by concatenating these two preimages at a conjugate pair. It will be explained more in Section 2.3. Annexstein presented a software implementation and it requires $O(2^n)$ bit operations [2]. In [6], Chang *et al.* presented a method for implementing D-homomorphism for producing de Bruijn sequences of long period. Games' generalized construction [13] is also based on Lempel's method. Mykkeltveit *et al.* [38] studied the composition of recurrence relations and represented the Lempel construction in terms of compositions of recurrence relations.

In [28], Mandal and Gong refined Mykkeltveit *et al.*'s construction and studied the composited construction for generating cryptographically strong de Bruijn sequences of order $(n+k)$ from span n sequences. For the composited construction, the feedback function of a de Bruijn sequence is composed of k-th order composition of the feedback function of span-n sequences and the sum of the product-of-sum terms which is the expensive part. According to their studies, in order to generate a cryptographically strong de Bruijn sequence of order $(n+k)$, the underlying span-n sequence needs to be long and have optimal or near optimal linear complexity. Mandal and Gong [29] further analyzed the security of the composited de Bruijn sequences from D-morphic point of view and also proposed an iterative technique for the implementation. The amount of storage required to implement the feedback function is $(n + k - 1)$ and their implementation outputs one bit in every k clock cycles, where k is the composition degree and 2^{n+k} is the period of the constructed de Bruijn sequence.

Table 2.1 summarizes storage and bit operation complexity information of many of these de Bruijn sequence generation techniques. The storage corresponds to the feedback function, instead of the whole sequence generator which would need $n+k$ extra storage for the shift register. Among them, composited construction [28] has the best computational complexity per output bit, which makes it especially suitable for generating de Bruijn sequences with long periods. We will show that our work significantly improves the complexity, resulting the first practical hardware implementation of de Bruijn sequence generators.

Table 2.1: Comparison of different de Bruijn sequence construction techniques in storage, throughput and computation complexity

	Storage	**Bit ops per cycle**	**Throughput**	**Total bit ops per output bit**
Fredricksen [10]	$3(n+k)$	$\mathcal{O}(n+k)$ †	$\frac{1}{n+k}$	$\mathcal{O}((n+k)^2)$ †
Jansen *et al.* [21]	$3(n+k)$	$\mathcal{O}(n+k)$ †	$\frac{1}{4(n+k)}$	$\mathcal{O}((n+k)^2)$ †
Annexstein [2]	–	$\mathcal{O}(2^{n+k})$	$\frac{1}{5}$	$\mathcal{O}(2^{n+k})$
Chang *et al.* [6]	–	1	$\frac{1}{3k(n+k)}$	$3k(n+k)$
Mandal and Gong [29]	$n+k-1$	$\mathcal{O}(n+k)$	$\frac{1}{k}$	$\mathcal{O}(k^2+nk)$

Results are given in terms of n and k where the period of the de Bruijn sequence is 2^{n+k} to enable comparison between the composited construction and other techniques. To translate the descriptions of the non-composited constructions in the paragraphs above to the table, replace n with $n+k$.

† The number of bit operations is an estimate based upon the cited paper.

2.3 Composited Construction of de Bruijn Sequences

We restate composited construction method from [28], including changing notations and creating new notations. New notations are informative in suggesting their meanings from our perspective. In this section, they are created to represent a complex formula for easy reference. Besides, some notations from [28] are changed to fit our taste. As it will become more clear, the combination of the superscript and subscript of a notation forms a pair of coordinates in figures. We also refer the subscript as index.

Moreover, we show visualizations of composited operations and the overall structure of composited de Bruijn sequence generators.

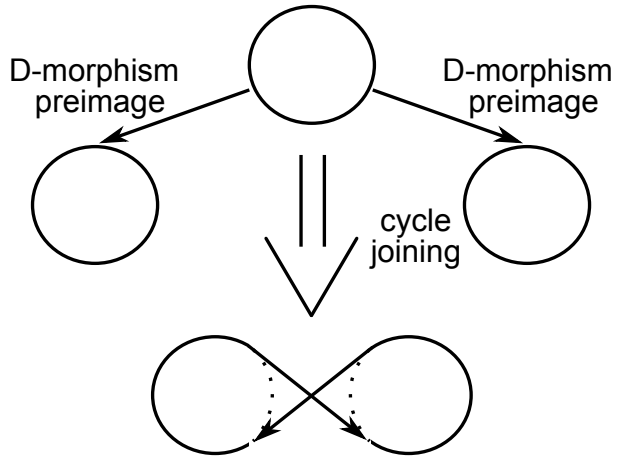


Figure 2.2: Overview of Composited Construction

Composited construction is based on Lempel's D-Homomorphism [24]. It is used to construct long de Bruijn sequences from a short de Bruijn sequence. Figure 2.2 shows the overview of this technique. Let D be the function, mapping $\mathbb{F}_{2^n}$ to $\mathbb{F}_{2^{n-1}}$, defined by

$$D(x_{[n-1:0]}) = (x_{n-1} + x_{n-2}, x_{n-2} + x_{n-3}, \ldots, x_1 + x_0).$$

The function D is 2-to-1 mapping. From a de Bruijn sequence of period 2^n, we first compute the two preimages of length $n+1$ for each subsequence of n under the function D. The two preimages are two distinct 2^n-cycles. Then we join these two cycles at a conjugate pair to make a 2^{n+1}-cycle. A conjugate pair is formed by two subsequences of the same length, that differ in their leftmost element, namely $(x_{n-1}, x_{n-2}, \ldots, x_0)$ and $(x_{n-1}, x_{n-2}, \ldots, x_0+1)$, e.g. $(1,0,1,0,1,0)$ and $(1,0,1,0,1,1)$. We can repeat this process to create longer de Bruijn sequences.

For example, given a de Bruijn sequence . . . 00110011, we compute every subsequence's two preimages. For example, the subsequence 11 has two preimages 101 and 010, shown in Figure 2.3. In the end, we end up with two seperate cycles consisting of the preimages we compute with no overlapping in subsequences. The cycles shown in Figure 2.3 are annotated with their corresponding sequences, respectively . . . 11011101 and . . . 00100010. We pick a conjugate pair $(0, 1, 0)$ and $(0, 1, 1)$, where we join two cycles to get the de Bruijn sequence with period 2^3. We can do this over and over to construct de Bruijn sequence with period $2^4, 2^5$ and so on.

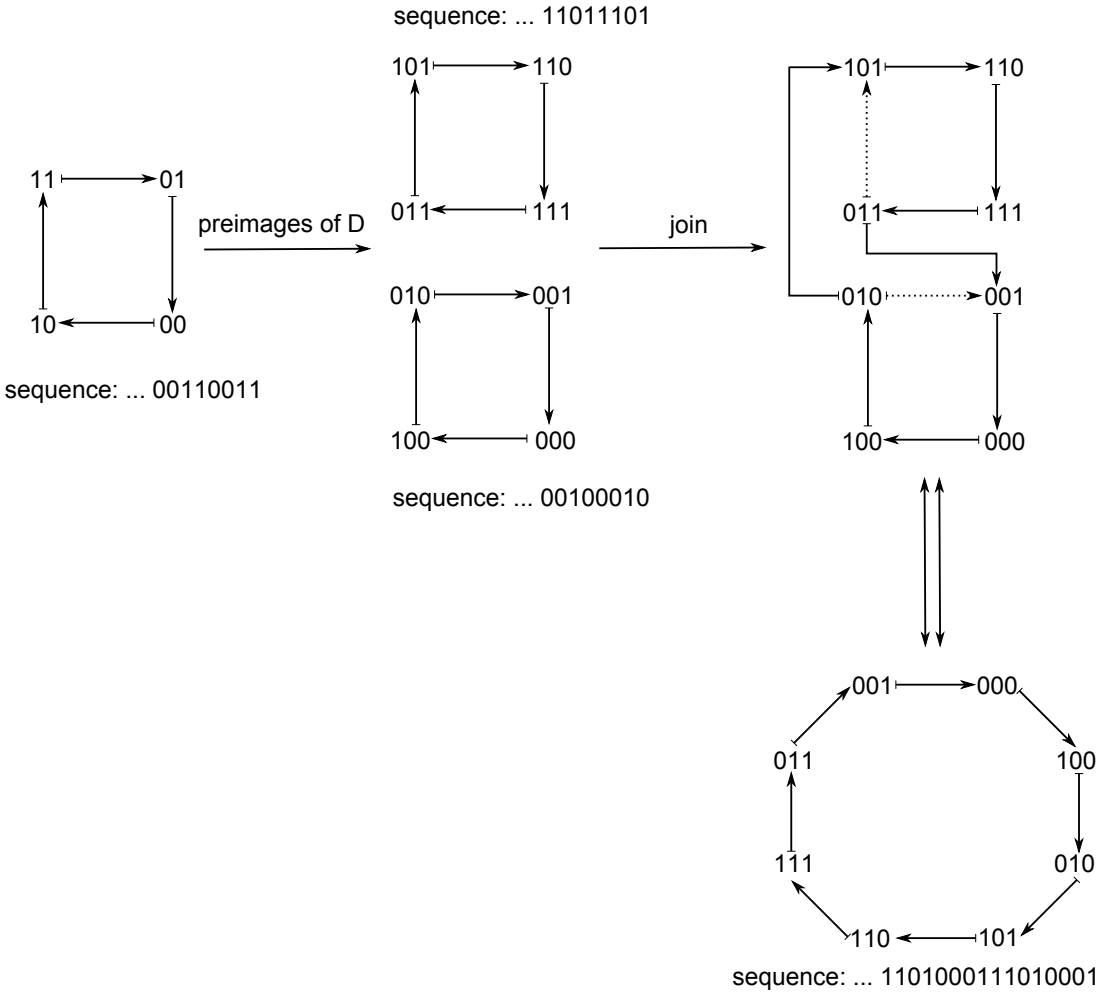


Figure 2.3: An Example of Composited Construction

In summary, starting with a de Bruijn sequence of period 2^n, by applying composited construction k times, we will end up with a de Bruijn sequence of period 2^{n+k}.

We now proceed to discuss how composited construction changes the boolean function/relation that defines the feedback value of the shift register that generates the starting de Bruijn sequence.

2.3.1 Composition Operation

Note that our definition of composition operation, shown belown, is different from function composition.

Definition 3. *Let g be a function of n inputs, i.e. $x_{[n-1:0]}$ and f be a function of a boolean vector, i.e. $x_{[:0]}$. Then*

$$(g \circ f)(x_{[n-1:0]}) = g(f(x_{[:n-1]}), \ldots, f(x_{[:0]}))$$

Composition operation consists of an operator, denoted by $\circ$ and two functions as operands. $g \circ f$ (Definition 3) applies f to *each* argument of g. If g is a function of n arguments, $g \circ f$ will consist of n copies of f, each producing one argument for g. The argument to each instance of f is offset by 1.

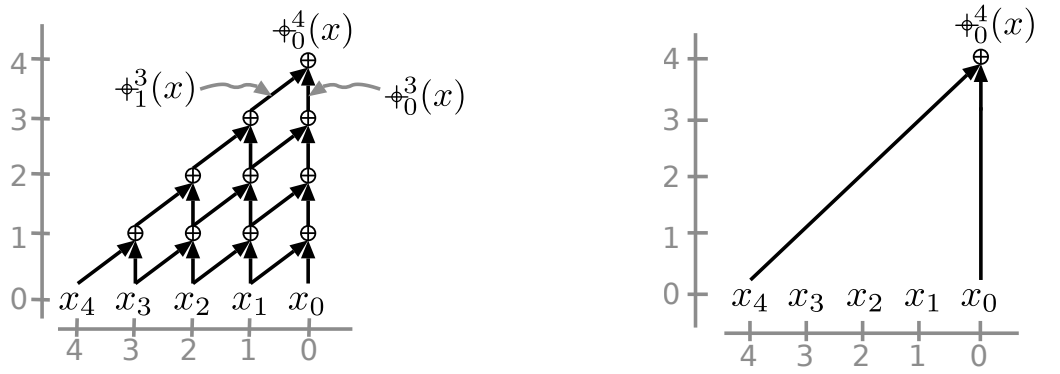
Let us now formalize a summation operation that will be used throughout this thesis.

Definition 4.
$$\begin{aligned}
\oplus_0^1(x) &= x_0 + x_1 \\
\oplus_0^k(x) &= \oplus_0^1 \circ \oplus_0^{k-1}(x) \\
&= \oplus_0^1((..., \oplus_2^{k-1}(x), \oplus_1^{k-1}(x), \oplus_0^{k-1}(x))) \\
&= \oplus_1^{k-1}(x) \ + \ \oplus_0^{k-1}(x) \\
\oplus_n^k(x) &= \oplus_0^k(x_{[:n]}), \ \textit{where } k \in \mathbb{Z} \textit{ and } k \geq 2
\end{aligned}$$

Remark 1 (The connection between function D and $\oplus_0^1$)**.**
Let I be an identity function, mapping $\mathbb{F}_{2^n}$ to $\mathbb{F}_{2^n}$. Then,

$$D(x_{[n-1:0]}) = I \circ \oplus_0^1(x_{[n-1:0]})$$

Let k be 2 in Definition 4, we have $\oplus_0^2(x) = \oplus_0^1 \circ \oplus_0^1(x) = \oplus_0^1(\ldots, x_2 + x_1, x_1 + x_0) = x_0 + x_2$, which can be derived from Definition 3 by replacing f and g with $\oplus_0^1(x)$. Its higher order extension $\oplus_0^k(x)$, where $k > 2$, can be derived by applying Definition 3 multiple times and replacing f with the new intermediate result each time and g with $\oplus_0^1(x)$. In $\oplus_n^k(x)$, the subscript n slices the input vector and picks the higher half as the input to the function $\oplus_0^k$, i.e. $\oplus_n^k(x) = \oplus_0^k(x_{[:n]})$. $k-1$ is how many times we apply the composition operation , that is $\oplus_n^k(x) = \underbrace{\oplus_n^1 \circ \oplus_0^1 ... \circ \oplus_0^1}_{\#\, of \circ : k-1}$. Plus, n and k are the coordinates when we draw $\oplus_n^k(x)$ in a picture.



(a) Recursive

(b) Direct

Figure 2.4: Computation of $\oplus_0^4(x)$

There are two manners of computing $\oplus_n^k(x)$, which are recursive way and direct way, as shown in Figure 2.4a and Figure 2.4b respectively. We call the Figure 2.4a a mesh structure of XOR, which is derived from the recursive definition of $\oplus_n^k(x)$ in Definition 4. It is worthwhile to notice that intermediate results remain in the structure. As a result, we do not need another mesh structure to compute $\oplus_0^3(x)$ while we have the mesh for $\oplus_0^4(x)$. This reflects mesh's characteristic of being able to naturally utilize reusing to some extend. This is one advantage over the direct manner. For the direct manner, we just directly expand $\oplus_0^k(x)$ to elements from vector x according to Definition 4, and cancel elements when the rule, $v + v = 0$, where v is in $\mathbb{F}_2$, is applicable. For example,

$$\begin{aligned}
\oplus_0^4(x) &= \oplus_1^3(x) + \oplus_0^3(x) \\
&= \oplus_2^2(x) + \oplus_1^2(x) + \oplus_1^2(x) + \oplus_0^2(x) \\
&= \oplus_3^1(x) + \oplus_2^1(x) + \oplus_1^1(x) + \oplus_0^1(x) \\
&= (x_3 + x_4) + (x_2 + x_3) + (x_1 + x_2) + (x_0 + x_1) \\
&= x_0 + x_4
\end{aligned}$$

Figure 2.4b shows the computation of $\oplus_0^4(x)$ in such way. For the direct manner, one interesting question would be how many elements are left after expansion and all applicable cancellations? To answer it, we first observe one special case when k is a power of 2.

Lemma 1. *If k is a power of 2, then $\oplus_n^k(x) = x_n + x_{n+k}$.*

Proof. It can be proved by induction. Assume when $k = 2^j$ where j is a non-negative

integer, we have $\oplus_n^k(x) = x_n + x_{n+2^j}$. Then, when $k = 2^{j+1}$,

$$\begin{aligned}
\oplus_n^k(x) &= \oplus_n^{2^j} \circ \oplus_0^{2^j}(x) \\
&= \oplus_n^{2^j}(y), \text{where y is a vector and } y_i = \oplus_i^{2^j}(x)). \\
&= y_n + y_{n+2^j} \\
&= \oplus_n^{2^j}(x) + \oplus_{n+2^j}^{2^j}(x) \\
&= x_n + x_{n+2^j} + x_{n+2^j} + x_{n+2\times 2^j} \\
&= x_n + x_{n+2^{j+1}} \\
&= x_n + x_{n+k}
\end{aligned}$$

□

Theorem 1. *The number of bit elements from x for $\oplus_n^k(x)$ is $2^{H(k)}$, where $H(k)$ is the Hamming weight of the binary representation of k and n is an arbitrary natural number.*

Proof. Theorem 1 can be proved in the following way. First, the integer k is written as a sum of $H(k)$ powers of 2, $k = 2^{i_1} + 2^{i_2} + \ldots + 2^{i_{H(k)}}$, where $i_1 < i_2 < \ldots < H(k)$.

$$\begin{aligned}
\oplus_n^k(x) &= \oplus_n^{2^{i_{H(k)}}}(y), \text{ where } y_i = \oplus_i^{2^{i_{H(k)-1}}} \circ \cdots \circ \oplus_0^{2^{i_1}} \circ \oplus_0^{2^{i_1}}(x) \\
&= y_n + y_{n+2^{i_{H(k)}}}, \text{ according to Lemma 1} \\
&= \oplus_n^{2^{i_{H(k)-1}}}(z) + \oplus_{n+2^{i_{H(k)}}}^{2^{i_{H(k)-1}}}(z), \text{ where } z_i = \oplus_i^{2^{i_{H(k)-2}}} \circ \cdots \circ \oplus_0^{2^{i_1}} \circ \oplus_0^{2^{i_1}}(x) \\
&= z_n + z_{n+2^{i_{H(k)-1}}} + z_{n+2^{i_{H(k)}}} + z_{n+2^{i_{H(k)}}+2^{i_{H(k)-1}}}, \text{ according to Lemma 1} \\
&= \ldots
\end{aligned}$$

We can continue until all left on the right side of the equation are elements from vector x. Because of the fact that a sum of different powers of 2 is not a power of 2, no terms in each step above have the same index. Therefore no cancellation $v + v = 0$ is possible. We can see how in each line the number terms to add together doubles. Finally, after eliminating every function $\oplus$ on the right side, the total number of elements from x is $2^{H(k)}$. Accordingly, the number of XOR operations is $2^{H(k)} - 1$. □

Useful corollaries of Theorem 1 are:

Corollary 1. *The number of* XOR *operations for $\oplus_n^k(x)$ is $2^{H(k)} - 1$, where k and n can be arbitrary natural numbers.*

Corollary 2. *In computing* $\oplus_i^k$ *and* $\oplus_{i+\delta}^k$, $\delta \in \mathbb{Z}$ *and* $\delta \geq 1$, *direct computation requires fewer* XOR *operations than recursive computation.*

Proof. The recursive computation of $\oplus_{i+\delta}^k$ requires mesh in a shape of right triangle. To further compute $\oplus_i^k$, we need to extend this triangular mesh on the right with a rectangular mesh whose width is δ, height is the same as the triangular mesh. The total number of XOR operations needed for each way are as follows.

Direct Computation: $2(2^{H(k)} - 1) = 2 \times 2^{H(k)} - 2 \leq 2(k-1) - 2$

Recursive Computation: $(k-1+k-2+\ldots+1) + k\delta = \frac{k^2}{2} - \frac{k}{2} + k\delta$, where $k\delta$ accounts for the rectangular mesh that extends the triangular mesh which has $\frac{k^2}{2} - \frac{k}{2}$ XOR operations.

Because the cost of direct computation in terms of XOR operations is independent of δ, while for recursive computation, the cost increases as δ increases, if Corollary 2 is true for $\delta = 1$, then it is true for any $\delta \geq 1$. So we check if the direct computation is better than the recursive computation when $\delta = 1$.

$$\begin{aligned} \frac{k^2}{2} - \frac{k}{2} + k - 2(2^{H(k)} - 1) &\geq \frac{k^2}{2} + \frac{k}{2} - (2(k-1) - 2) \\ &\geq \frac{k^2}{2} - \frac{3k}{2} + 4 \\ &> 0, \text{ for any } k \geq 1 \end{aligned}$$

□

Corollary 3 (Indexes of the elements from x in $\oplus_n^k(x)$)**.** *Let* $k = 2^{i_1} + 2^{i_2} + \ldots + 2^{i_{H(k)}}$ *and* $i_1 < i_2 < \ldots < i_{H(k)}$. *The indexes of the* x*'s elements that appear in* $\oplus_n^k(x)$ *are in one-to-one correspondence to the objects/elements of the power set of* $\{2^{i_1}, 2^{i_2}, \ldots, 2^{i_{H(k)}}\}$. *The mapping that maps a set to an index is defined as follows. The index corresponding to the set* $\{idx_1, idx_2, idx_3, \ldots\}$ *is* $n + idx_1 + idx_2 + idx_3 + \ldots$. *If a set is empty, the corresponding index is* n.

For example, to compute the indexes of x elements in $\oplus_3^5(x)$. We start by writing $5 = 1 + 4$. The power set of $\{1, 4\}$ is $\{\{\}, \{1\}, \{4\}, \{1, 4\}\}$. The corresponding indexes are $3, 3+1, 3+4, 3+1+4$, that is $3, 4, 7, 8$. Therefore,

$$\oplus_3^5(x) = x_3 + x_4 + x_7 + x_8.$$

The hints to prove Corollary 3 can be found in the proof of Theorem 1.

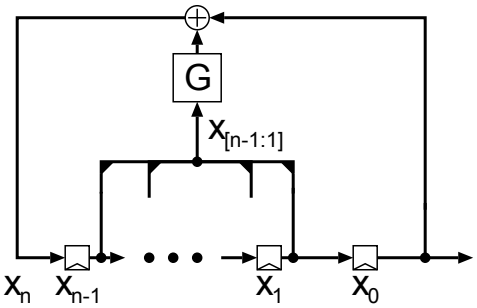
2.3.2 Composited de Bruijn Sequence Generators

Now we continue constructing composited de Bruijn sequence generators. We begin by defining a function used to identify the all 0 subsequence.

Definition 5. *All*$0_n(x) = 1$ *iff all elements of* $x_{[n:1]}$ *are* 0*:*

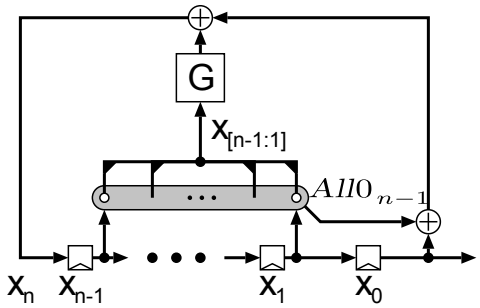
$\forall\, i \in [n, 1]\ .\ x_i = 0.$

Assuming Figure 2.5 is a span-n sequence generator, the composited construction begins by creating a de Bruijn-sequence generator with period 2^n (Figure 2.6) by adding *All*0_{n-1}, which tests if bits $x_{[n-1:1]}$ are 0 (Definition 5).



$x_n + G(x_{[n-1:1]}) + x_0$

Figure 2.5: A span-n sequence generator



$x_n + G(x_{[n-1:1]}) + x_0$
$+ All0_{n-1}(x_{[n-1:1]})$

Figure 2.6: A de Bruijn sequence generator

We describe the feedback structure of shift registers with *feedback expressions*. Given a feedback expression $g(x)$ for an n-stage NLFSR, the term x_n represents the output of the feedback and the input to the shift register. Factoring x_n out of $g(x)$ to create an equation of the form $x_n = \ldots$ results in a feedback function. In the figures, the function G is drawn with all $n-1$ bits of $x_{[n-1:1]}$ as inputs. In reality, G almost always is a function of a relatively few bits (*e.g.*, 10 for the $(k = 40, n = 24)$ example in this thesis). We use gray regions in figures to identify sets of nodes that are inputs to a function such as *All*0. Next we define a function to detect the subsequence of alternating 1 and 0 with 1 appearing at odd indexes.

Definition 6. $X_n(x) = 1$ *iff all of the elements with even indexes from* $x_{[n:1]}$ *are* 0 *and all of the elements with odd indexes are* 1*.*

Remark 2. *With $X_n(x) = 1$, we specify the position to join two preimages (i.e. two sequences). More specifically, if we can find a subsequence $x_{[n+i-1:i]}$ that has alternating 1,0 pattern with 1 appearing at the odd indexes in either of the two preimages, then we insert the other preimage into the first one after the subsequence.*

In [28], Mandal and Gong restate the theorem from [38] as follows.

Theorem 2. *Let $g(x_{[n:0]}) = x_0 + x_n + f(x_{[n-1:1]})$, such that $g(x) = 0$ generates a de Bruijn sequence. Then*

$h(x_{[n+1:0]}) = (g \circ \pitchfork(x)) + X_n(x_{[n-1:1]}) = 0$ generates de Bruijn sequences with period 2^{n+1}.

Remark 3. *We relate Theorem 2 to the composited construction process (Figure 2.2) by pointing out that the preimages of the sequence generated by feedback relation $g(x_{[n:0]}) = 0$ satisfy $g \circ \pitchfork(x_{[n:0]}) = 0$. In other words, $g \circ \pitchfork(x_{[n:0]}) = 0$ generates two different cycles depending on the initial state. X_n joins two different cycles into a full and complete cycle that is a de Bruijn sequence.*

$All0_n^k$ and X_n^k are extensions of $All0_n$ and X_n, which were previously defined in Definition 5 and Definition 6 respectively, to support composited de Bruijn sequence generators. $All0_n^k$ and X_n^k check nodes from row k instead of the bottom row as $All0_n$ and X_n do.

Definition 7. *$All0_n^k(x) = 1$ iff all of the elements from the input vector to $All0_n^k$ after k times of composition with $\pitchfork_0^1$ are 0:*

$$All0_n^k(x) = All0_n \circ \pitchfork_0^k(x) = \sum_{i=i}^{n} (\pitchfork_i^k(x) + 1)$$

Definition 8. *$X_n^k(x) = 1$ iff all of the elements with even indexes from the input vector to X_n^k after an order-k composition are 0 and all elements with odd indexes are 1:*

$$X_n^k(x) = X_n \circ \pitchfork_0^k(x) = \sum_{i \in Z_o^n} \pitchfork_i^k(x) \sum_{i \in Z_e^n} (\pitchfork_i^k(x) + 1)$$

By applying this fundamental Theorem 2 repeatedly on a relation that will generate a de Bruijn sequence of shorter period, we will construct an NLFSR that will generate a de Bruijn sequence with long period. More concretely, suppose $x_n + G(x_{[n-1:1]}) + x_0 = 0$ generates a span-n sequence with period $2^n - 1$, then we know $g(x_{[n:0]}) = x_n + G(x_{[n-1:1]}) + All0_{n-1}(x_{[n-1:1]}) + x_0 = 0$ generates a de Bruijn sequence with period 2^n. After repeatedly applying Theorem 2 k times, and based on Definition 3, Definition 7, Definition 8 we have:

$$
\begin{aligned}
g \circ \dashv\!\!\oplus\!\!\dashv_0^k(x) &= g(\dashv\!\!\oplus\!\!\dashv_n^k(x), ..., \dashv\!\!\oplus\!\!\dashv_0^k(x)) \\
&= \dashv\!\!\oplus\!\!\dashv_n^k(x) + G(\dashv\!\!\oplus\!\!\dashv_{n-1}^k(x), ..., \dashv\!\!\oplus\!\!\dashv_1^k(x)) \\
&+ All0_{n-1}^k(x) + \sum_{i=0}^{k-1} X_{n+k-1-i}^i(x) + \dashv\!\!\oplus\!\!\dashv_0^k(x) \\
&= 0
\end{aligned}
$$

It is called an (k, n)-composited construction. Its feedback relation is:

$$
\dashv\!\!\oplus\!\!\dashv_n^k(x) + G(\dashv\!\!\oplus\!\!\dashv_{n-1}^k(x), ..., \dashv\!\!\oplus\!\!\dashv_1^k(x)) + All0_{n-1}^k(x_{[n-1:1]}) + \sum_{i=0}^{k-1} X_{n+k-1-i}^i(x) + \dashv\!\!\oplus\!\!\dashv_0^k(x) = 0
$$

For simplicity, we define:

Definition 9. *$ccdB_k^n(x)$ is the feedback expression for a (k, n)-composited de Bruijn sequence generator. Its original feedback expression is $G + x_0 + x_n$:*

$$
mesh_k^n(x) = All0_{n-1}^k(x) + \sum_{i=0}^{k-1} X_i^{n+k-1-i}(x)
$$

$$
\begin{aligned}
ccdB_k^n(x) = &\dashv\!\!\oplus\!\!\dashv_n^k(x) + G\left(\dashv\!\!\oplus\!\!\dashv_{n-1}^k(x), ..., \dashv\!\!\oplus\!\!\dashv_1^k(x)\right) + \\
&\dashv\!\!\oplus\!\!\dashv_0^k(x) + mesh_k^n(x)
\end{aligned}
$$

Figure 2.7 shows an $(k = 6, n = 4)$ composited construction. In an order-k composition, each x_i in the feedback expression is replaced by $\dashv\!\!\oplus\!\!\dashv_i^k(x)$. The $All0_n$ term becomes $All0_n^k$ (Definition 7), to include $\dashv\!\!\oplus\!\!\dashv_0^k$. Also, a new term is added, which examines each instance of $\dashv\!\!\oplus\!\!\dashv_i^j(x)$ for $j \in [k-1, 0]$ and $i \in [n+k-1, 1]$. This new term checks if an odd number of vectors of $\dashv\!\!\oplus\!\!\dashv_i^j(x)$ terms at the same level of recursion (same value for j) have an alternating 1,0 pattern with 1 appearing at odd indexes (Definition 8).

2.3.3 Three Baseline Implementations of (k, n)-composited de Bruijn Sequence Generators

For evaluating the benefits of our optimizations, we present three baseline implementations of composited de Bruijn sequence generators. The first two baseline implementations are different in the way of computing nodes, $\dashv\!\!\oplus\!\!\vdash_i^j(x)$ for $j \in [k-1, 0]$ and $i + j \leq n + k - 1$, which are needed in the computation of $mesh_n^k$.

If we compute each of these nodes recursively (Figure 2.4a), we will end up having a mesh of XOR gates as shown in Figure 2.7. Then we explictly compute X, $All0$ and G terms. Figure 2.7 illustrates this mesh-based implementation. The signal x_{10}, which carries the feedback value, appears implicitly in the term $\dashv\!\!\oplus\!\!\vdash_4^6(x)$. We factor it out by adding x_{10} on both sides of the equation so that we can create an equation of the form $x_{10} = \ldots$.

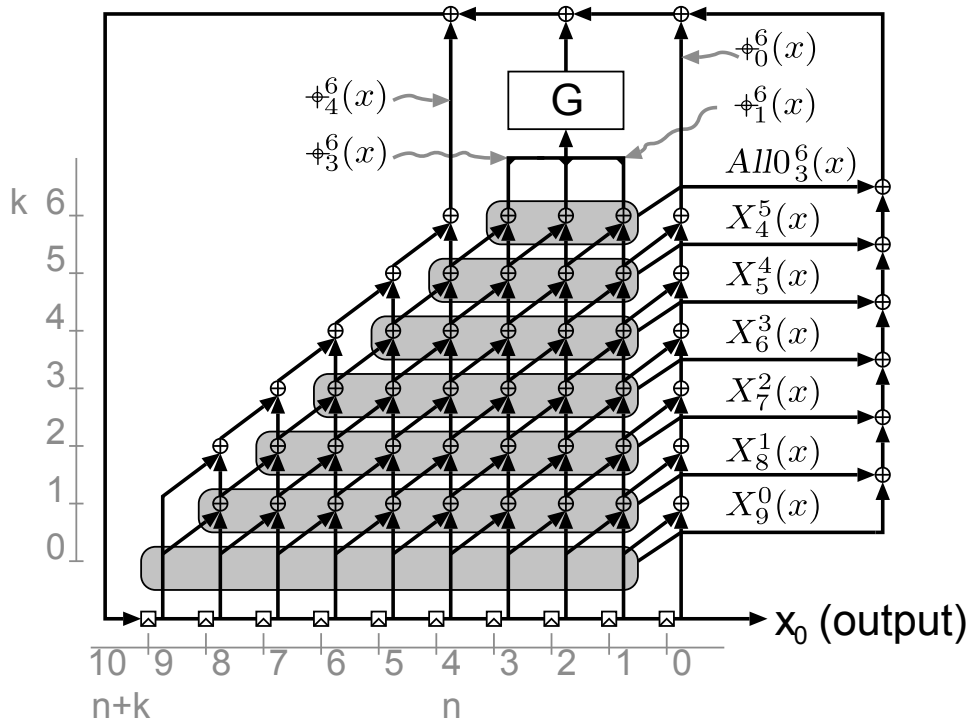


Figure 2.7: A $(6, 4)$-composited de Bruijn sequence generator

This mesh-based implementation of an (k, n)-composited de Bruijn sequence generator requires a mesh with $n+k-1$ nodes at the bottom, $k+1$ rows, and $n-1$ nodes at the top,

for a total of $(2n+k-2)(k+1)/2$ XOR gates in the XOR mesh structure. So the overall complexity of computing the mesh nodes in terms of bit-level operations is $\mathcal{O}(k^2+nk)$.

An alternative approach would be to compute each node separately and directly as in Figure 2.4b. With the simplification $v+v=0$, where v is a variable over $\mathbb{F}_2$, the minimum number of XOR gates to compute $\dot{+}_i^j(x)$ is reduced to $2^{H(j)}-1$, where H(j) denotes the Hamming weight of j and $0 \leq j \leq k$, $i+j \leq n+k-1$. Therefore, each node now requires minimum computation resources to be calculated. It is called direct implementation. The rest of the computation of the feedback function is the same as the mesh-based implementation. From now on, for our convenience, we call $\dot{+}_i^j(x)$ (where $0 \leq j \leq k$, $i+j \leq n+k-1$) nodes from the mesh withouth the implication that they are computed recursively, i.e. using the mesh. In such way, nodes with the same superscript(subscript) are from the same row(column) of the mesh and nodes with the highest(lowest) possible superscript are from the top(bottom) row. The diagonal of a mesh are made of the head node of each row, which are the nodes whose superscript plus subscript is equal to $n+k-1$.

Another baseline implementation is presented in [29]. It is essentially a serialized version of the mesh-based implementation. In each cycle, this serialized implementation computes one level of the mesh as well as the funtions (X or $All0$) applied on the nodes of this level. This serialization is possible because the computation of each level of the mesh only requires the nodes from the level right below it and neither of X and $All0$ needs nodes accross multiple levels of the mesh as inputs. As far as computational complexity is concerned, it uses $\mathcal{O}(n+k)$ bit operations but requires around k (i.e. the height of the mesh) clock cycles to generate one bit as output. Therefore, the number of bit operations per output bit is still $\mathcal{O}(k^2+nk)$ [29].

2.3.4 Cryptographic Applications

Though it is out of scope of this thesis, security is in the center of cryptographically strong pseudorandom number generators(CSPRNG). Refer to [7] for introductions to randomness, security analysis on PRNG, [35] for a brief survey of CSPRNGs. In [29] Mandal and Gong proved that composited de Bruijn sequence generators are cryptographically strong. Particularly, as far as linear complexity is concerned, they showed that if the starting span-n sequence (Figure 2.5) has the optimal linear complexity, which is 2^n-2, then the de Bruijn sequence will have a near-optimal linear complexity of $2^{n+k}-C$, where C is a relatively small constant that is dependent upon the starting span-n sequence. They also showed empirically that the span-$(n+k)$ subsequence of the de Bruijn sequence has

optimal or near-optimal linear complexity for values of $n+k$ ranging from 11 to 20. If this guideline is not followed, in the worst case G would be linear, the starting span-n sequence would be an m sequence, and then the resulting span-$(n+k)$ subsequence of the de Bruijn sequence would have a linear complexity of just $n+k$.

Due to the fact that composited de Bruijn sequence generators generate random number directly from its internal states, they are not directly suitable for applications that expose the random numbers in plaintext, such as EPC protocol for RFID tags [20]. In such applications, an adversary can extract the internal state from the sequence of random numbers OcDeb-*k*-*n* generates and then run the OcDeb algorithm to compute future (and possibly even previous) random numbers. The countermeasure is adding filtering functions. But the security of the filtering function needs to be analyzed just as with other such PRNGs. For applications that keep random numbers as secrets (e.g. in public-key cryptography [43], the randomly generated private key is kept secret and then used to generate the public key that is exposed), composited de Bruijn sequence generators can be used directly.

2.4 Summary

We introduced de Bruijn sequences and various construction techniques in Section 2.1 and Section 2.2. Among them, composited construction is the most efficient one. A (k,n)-composited de Bruijn sequence generator has a period 2^{n+k} and is built by applying composited construction to a span-n sequence generator with period 2^n-1. (k,n)-composited de Bruijn sequence generators need $n+k-1$ bits storage space and $\mathcal{O}(k^2+nk)$ bit operations. We also presented composited de Bruijn sequence generators' architecture, which inspired our optimizations that will be shown in the next chapter. Two different ways to compute nodes in the mesh of the architecture were discussed and the corresponding cost in XOR gates for each way was shown. We presented three different baseline implementations of composited de Bruijn sequence generators, which are direct implementation, mesh-based implementation, as well as Mandal and Gong's implementation. The first two differ in the way of computing nodes in the mesh. Mandal and Gong's implementation is essentially a serialized version of the mesh-based implementation. In Section 2.3.4, we briefly discussed the security aspect of composited de Bruijn sequence generators. Particularly, we pointed out that composited de Bruijn sequence generators need to be added filter functions in order to be used in applications where random numbers are exposed.

Chapter 3

Optimizations

Our optimizations are focused on the summation of the functions applied on each row of the mesh, (i.e. $mesh_n^k(x)$, the mesh and functions are shown in Figure 3.1), because this summation includes all of the overhead from composited construction. These are mathematical/algorithmic optimizations that are applicable to both hardware and software implementations. Computation of $mesh_n^k(x) = All0_{n-1}^k(x) + \sum_{i=0}^{k-1} X_i^{n+k-1-i}(x)$ examines every nodes in the mesh. In Theorem 3 (Section 3.1.3), we show that we can find a much simpler expression that is functionaly equivalent to it but only examines the top row nodes and the diagonal nodes instead of the whole mesh. The intermediate steps towards Theorem 3 can be found in Section 3.1.1 and Section 3.1.2. Theorem 4 (Section 3.1.5) states that for any function over the bottom row nodes, we can always find an equivalent function over other nodes in the mesh. Its corollary Corollary 4 shows that we can also examine nodes of an interior diagonal instead of examining the diagonal nodes. An interior diagonal is made of nodes that have same distance to the diagonal. Corollary 4 opens opportunities for hardware retiming. Besides, we show that these two functions (one examining the diagonal nodes, one examining nodes of an interior diagonal) are either the same or related to each other by a simple relation. Combined with this fact, Corollary 4 also enables efficient parallelization. These two applications of Corollary 4 are shown in Section 3.1.5. In Section 3.2, we compute the new complexity in bit operations after our optimizations. Inevitably we will introduce new definitions and notations. Symbols' superscript and subscript serve as coordinates in figures and a subscript is also referred to as an index.

We name optimized composited de Bruijn sequence generators "OcDeb", and call optimzed (k, n)-composited de Bruijn sequence generators "OcDeb-k-n". Efficient hardware implementations of OcDeb-k-n will be detailed in the next chapter.

3.1 Mathematical Optimizations

In this section, we present mathematical optimizations that reduce the absolute number of bit operations as well as the complexity so that the optimized composited construction technique will scale better in terms of increasing k and n. We express our optimizations in theorems and prove their correctness by proving the theorems. We depend on the mesh to illustrate our optimizations.

There are three key insights behind our mathematical optimizations.

- At most one row can satisfy X (have an alternating 1s and 0s pattern and 1 appears at odd indexes). This later becomes Lemma 2 in Section 3.1.1.
- We can detect if there is a row that satisfies X by looking at just the top row and the leftmost node of every row. This becomes Lemma 7 in Section 3.1.2.
- We can use a counter to detect if the top row of nodes are all 0 and all 1. This can be found in Section 3.1.4.

Together these optimizations allow us to look at only the leftmost node of each row, that is the diagonal nodes. Figure 3.7 shows the optimized version of Figure 2.7.

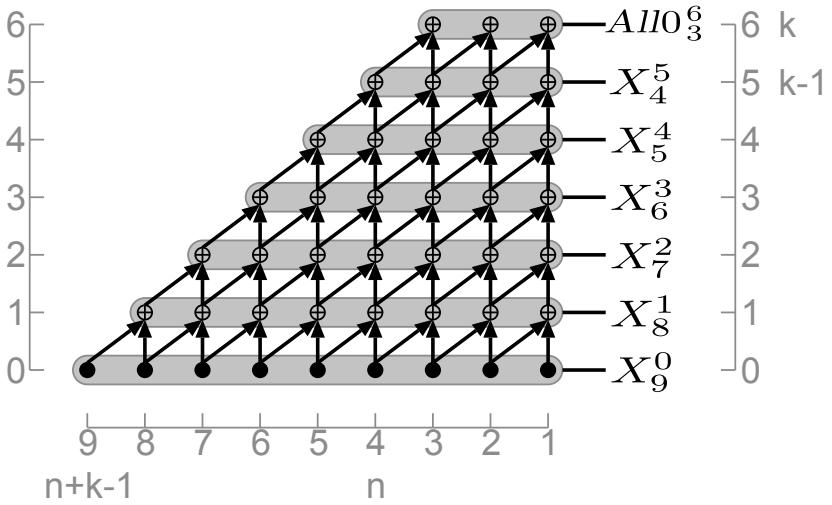


Figure 3.1: The mesh of $(6, 4)$-composited de Bruijn sequence generator

3.1.1 Only One Row Can Satisfy X

Now we present our first observation on the mesh structure. Recall that the function X checks whether the nodes are alternating 1s and 0s with 1 appearing at odd indexes.

Lemma 2. *At most one row satisfies X.*

Here we sketch out the reasoning behind Lemma 2 by introducing the following helper lemmas.

Lemma 3. *For any $k_1 \in \{\, x \in \mathbb{Z} \mid x \geq 0 \,\}$, if the row k_1 satisfies X (alternating 1s and 0s pattern with 1 appearing at odd indexes), then the row $k_1 + 1$ is all 1s.*

Lemma 4. *For any $k_1 \in \{\, x \in \mathbb{Z} \mid x \geq 0 \,\}$, if row k_1 is all 1s, then row $k_1 + 1$ is all 0s.*

Lemma 5. *For any $k_1 \in \{\, x \in \mathbb{Z} \mid x \geq 0 \,\}$, if row k_1 is all 0s, then row $k_1 + 1$ is all 0s.*

Lemma 6. *For any $k_1 \in \{\, x \in \mathbb{Z} \mid x > 0 \,\}$, if all of the nodes row k_1 are 0, then row $k_1 - 1$ does not satisfy X.*

Lemmas 3–6 are easily proved using the behaviour of an XOR gate, including the fact that with an XOR gate we can deduce the value of an input from the values of the output and the other input.

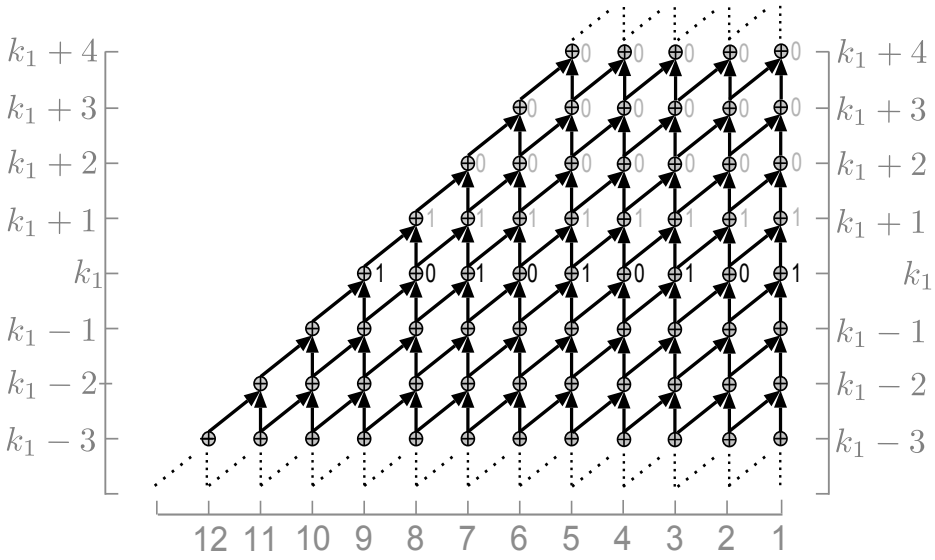


Figure 3.2: An example of Lemma 2

In Figure 3.2, without losing generlity, let us say row k_1 satisfies X. Also for clarity, we do not draw the associated functions on each row. Therefore, the XOR nodes from left to right are 101010101. We can then deduce the other nodes above it and their values are labled in grey colors in the picture. It is not hard to see Lemmas 3–5 are correct.

We apply these lemmas inductively to prove Lemma 2, which says that at most one row satisfies X. Intuitively, if a row k satisfies X, that is row k consists of alternating 1s and 0s with 1 at odd indexes, then the row above it is all 1s and all higher rows are all 0s, which prevents another row from satisfying X. With Lemma 2, we have reduced the problem of determining if an odd number of rows satisfy X to determining if there is any row that satisfies X.

3.1.2 Diagonal Optimization for a Specific Row

We now show that we can determine if a row k_1 satisfies X by checking if the top row is all 0s and if the leftmost nodes on the diagonal from row k and above satisfy a particular pattern (Lemma 7). Nodes on the diagonal are $\dot{+}^{k_1}_{n_1}(x)$, where $k_1 + n_1 = n + k - 1$ and k_1 ranges from 0 to k.

We use *Diag*0 (Definition 10) to examine the nodes on a diagonal. On top of them, we define *Pat* (Definition 11). They are all functions of a sequence x, which is left out intentionally for simplicity when there is no ambiguity. Examples of *Diag*0 are shown in Figure 3.3a and Figure 3.3b.

Definition 10. $Diag0^{k_1,k_2}_{n_2}(x) = \prod_{i=k_1}^{k_2} (\dot{+}^{i}_{n_2+k_2-i}(x) + 1)$*, meaning the diagonal (head nodes) on rows* $k_1, k_1 + 1, \ldots, k_2$ *are all 0.*

(a) Function $Diag0^{3,4}_5$ (b) Function $Diag0^{2,4}_5$

Figure 3.3: Examples of *Diag*0: (a) $Diag0^{3,4}_5$ and (b) $Diag0^{2,4}_5$

Definition 11.

$$Pat_{n_2}^{k_1,k_2}(x) = \begin{cases} Diag0_{n_2}^{k_1+2,k_2} \cdot (\oplus_{n_1-1}^{k_1+1}(x)+1) \cdot \oplus_{n_1}^{k_1}, & \textit{when } n_1 \textit{ is odd} \\ Diag0_{n_2}^{k_1+2,k_2} \cdot \oplus_{n_1-1}^{k_1+1}(x) \cdot (\oplus_{n_1}^{k_1}(x)+1), & \textit{otherwise} \end{cases}$$

where $n_1 = k_2 + n_2 - k_1$.

$Pat_{n_2}^{k_1,k_2} = 1$ *iff the head nodes on rows* $k_1 \dots k_2$ *satisfy:*

- *the head nodes of rows* $k_1+2 \dots k_2$ *are all 0.*
- *the head node of row* k_1+1 *is 1.*
- *the head node of row* k_1 *is 1(0) if the head node of row* k_1 *is in an odd(even) column.*

In other words, $Pat_{n_2}^{k_1,k_2}$ is 1 iff the head nodes on row $k_1, k_1+1 \dots, k_2$ match the regular expression $11[0]\{k_2-k_1-2\}$ when $k_2+n_2-k_1$ is odd or $01[0]\{k_2-k_1-2\}$ when $k_2+n_2-k_1$ is even.

For example, shown in Figure 3.4a and Figure 3.4b, $Pat_5^{1,4} = 1$ iff the heads nodes on row $1, 2, 3, 4$ are 0100; $Pat_5^{0,4} = 1$ iff the heads nodes on row $0, 1, 2, 3, 4$ are 11000.

(a) Function $Pat_5^{1,4}$ (b) Function $Pat_5^{0,4}$

Figure 3.4: Examples of Pat: (a) $Pat_5^{1,4}$ and (b) $Pat_5^{0,4}$

Lemma 7. $X_{n_1}^{k_1}$ *is 1 iff there exists a row* k_2 *so that* $k_2 \geq k_1+2$*,* $All0_{n_2}^{k_2} = 1$ *and* $Pat_{n_2}^{k_1,k_2} = 1$*, where* $n_2 = n_1 + k_1 - k_2$*.*

Proof. Figure 3.2 and Figure 3.5 provide intuitive impression on the forward direction and the reverse direction of the lemma respectively. Figure 3.5 is an instantiation of the reverse direction of Lemma 7 with $k_2 = k_1 + 4$, $n_1 = 9$ and $n_2 = 5$. With the precondition that

$All0_{n_2}^{k_2} = 1$ and $Pat_{n_2}^{k_1,k_2} = 1$, we know the values of the nodes on row k_2 and the diagonal nodes from k_1 to k_2. They are labeled with their values in black color. We can now proceed interatively from left to right and top to bottom to compute the other nodes between row k_1 and k_2. Their values are labeled in the picture with grey color instead. We can see row k_1 indeed satisfies X.

To prove the forward direction of Lemma 7, we begin with $X_{n_1}^{k_1}$, then use Lemma 3 to show that row $k_1 + 1$ is all 1s, use Lemma 4 to show that row $k_1 + 2$ is all 0s, then use Lemma 5 inductively to conclude that all rows above $k_2 + 2$ are all 0s. At this point we know that any row $k_2 \geq k_1 + 2$ is all 0s and the diagonals from k_1 to k_2 satisfy $Pat_{n_2}^{k_1,k_2}$.

To prove the reverse direction of Lemma 7, we use the fact that the value of an input to an XOR gate can be determined from the values of the output and the other input. We know that row k_2 is all 0s and we know the values of all the diagonal nodes on rows $k_2 \ldots k_1$. Proceeding iteratively from left to right and top to bottom, we can compute the value of each node down to row k_1 and show that row k_1 must satisfy X. All of the nodes are 0 until we encounter the first diagonal node that is 1, which then forces that entire row to be 1s. We then apply Lemma 8 to show that the next row down satisfies X. □

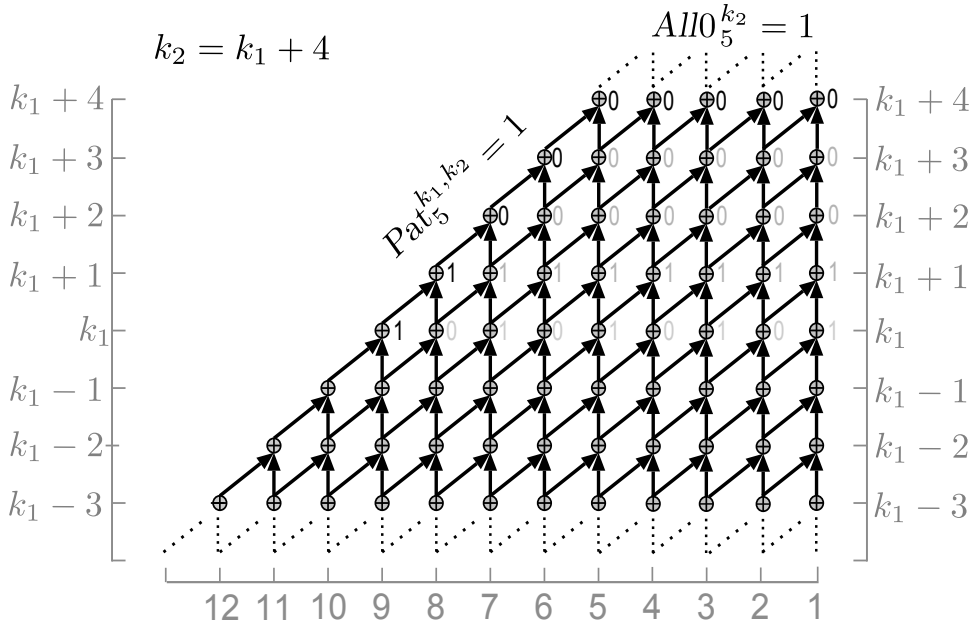


Figure 3.5: An example of Lemma 7

Lemma 8 strengthens Lemma 3, which was just an implication.

Lemma 8. *A row satisfies X iff*

1. *the row above is all 1s, and*
2. *the head element of the row is 1(0) if the index of the head element is odd(even).*

3.1.3 Diagonal Optimization for the Complete Mesh

We now combine Lemma 2 and Lemma 7 to prove important Theorem 3, which reduces the problem of determining if the mesh satisfies $mesh_n^k$ to just examining the top row and the diagonal. To present Theorem 3, we first define another function $SomePat_{n_2}^{k_2}$.

Definition 12. $SomePat_{n_2}^{k_2} = 1$ *iff there is a diagonal segment starting at a row* $k_1, 0 \leq k_1 \leq k_2 - 2$ *such that* $Pat_{n_2}^{k_1,k_2} = 1$.

With the help of regular expression language, we can rephrase Definition 12. $SomePat_{n_2}^{k_2} = 1$ iff the diagonal nodes from bottom to the top match $([01][01])^*110^*$ or $[01]([01][01])^*010^*$ when $n_2 + k_2$ is odd. Similary, for $n_2 + k_2$ is even, $SomePat_{n_2}^{k_2} = 1$ iff the regular expression $([01][01])^*010^*$ or $[01]([01][01])^*110^*$ are matched.

Theorem 3. $mesh_n^k(x) = 1$ *iff one of two conditions is true:*

1. *The top row (row k) is all 0s and none of the rows is the starting point for a diagonal segment that satisfies* Pat.
2. *The top row (row k) is all 1s and the head node on row $k-1$ is 1 if n is odd.*

$$mesh_k^n = \begin{cases} SomePat_{n-1}^k + 1, & when\ All0_{n-1}^k = 1 \\ \not{+}_n^{k-1} = 1\ (resp.\ 0), & when\ All1_{n-1}^k = 1\ and \\ & n\ is\ odd\ (resp.\ even) \\ 0, & otherwise \end{cases}$$

Proof. We now sketch the proof of Theorem 3. Using Lemma 2, which says that at most one row satisfies X, we know that there are only two cases such that $mesh_n^k(x)$ is satisfied:

1. $All0_{n-1}^k$ is 1 and no lower row satisfies X.

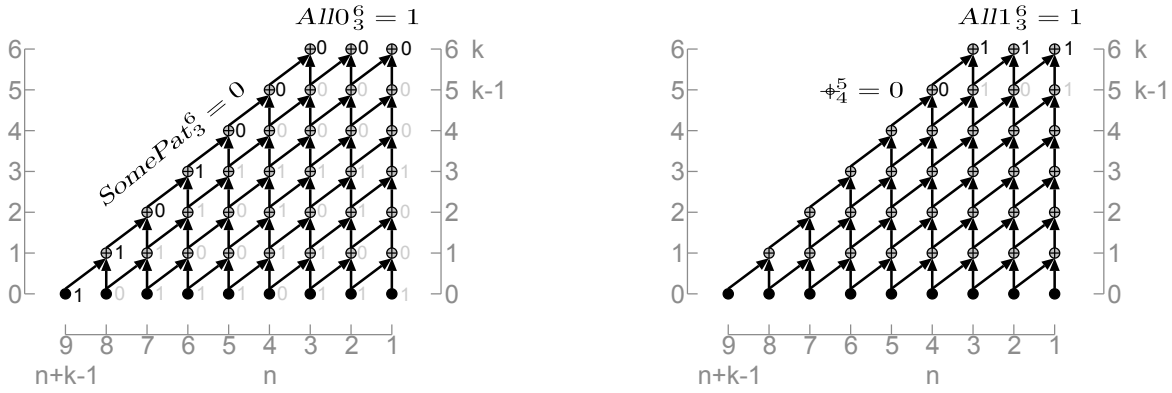
2. $All0^k_{n-1}$ is 0 and exactly one lower row satisfies X.

For the first case, from Lemma 7, we know that no row between 0 and $k-2$ satisfies X iff $SomePat^k_{n-1} = 0$ or $All0^k_{n-1} = 0$. Because we admit that $All0^k_{n-1} = 1$, $SomePat^k_{n-1} = 0$, which matches the top line of the right-side of Theorem 3.

For the second case, where $All1^k_{n-1} = 1$, the only possible row that can satisfy X without causing the top row to be all 0s is the row just below the top row. It is not hard to see that the second row from the top satisfies X iff the top row is all 1s ($All1^k_{n-1}$) and the head element of the second row from the top, which is $\dotplus^{k-1}_n$, has the correct value (is 1 if the index of the head element is odd).

We draw Figure 3.6a and Figure 3.6b to help understand the forward direction of Theorem 3. Both figures show the mesh structure of $(6, 4)$-composited de Bruijn sequence generator. In Figure 3.6a, we assign values to the diagonal nodes so that,

$$SomePat^4_5 = Pat^{0,4}_9 \text{ or } Pat^{1,4}_8 \text{ or } Pat^{2,4}_7 \text{ or } Pat^{3,4}_6 = 0.$$



(a) An example of the 1.case in Theorem 3 (b) An example of the 2.case in Theorem 3

Figure 3.6: Examples of Lemma 7

We assign 0s to the top row nodes so that $All0^6_3 = 1$. These values are labeled in black colors labeled on the figure. Similarly, we do the same in Figure 3.6b with the known precondition that $All1^6_3 = 1$ and $\dotplus^5_4 = 0$. In each figure, with known values of the nodes, we calculate the rest nodes and it is easy to verify that no rows in Figure 3.6a satisfies X and in Figure 3.6b only row 5 satisfies X but the top row(row 6 doesn't satisfy $All0^6_3$). Therefore, in both pictures, $mesh^6_3 = 1$. □

3.1.4 All-Zeroes State Machine

Up to this point, we have eliminated the need to examine any of the interior nodes in the XOR mesh. Because the input to the XOR mesh is a shift register, each node in a row of the mesh is simply a shifted version of its neighbor to the left. We replace the explicit test that the n nodes in the top row are all 0 (all 1) with a simple counter that counts the number of consecutive clock cycles that the head node in the top row is 0 (or 1) and is compared with n. More details of this state machine can be found in Section 4.2.1.

Figure 3.7 illustrates the diagonal optimization (the optimization based on Theorem 3) and the optimization using counters for OcDeb-*6-4*. Because our optimizations avoid the need to compute the interior nodes of the composited construction, it now becomes cheaper to compute each node separately, hence Figures 3.7–3.10 do not show the mesh.

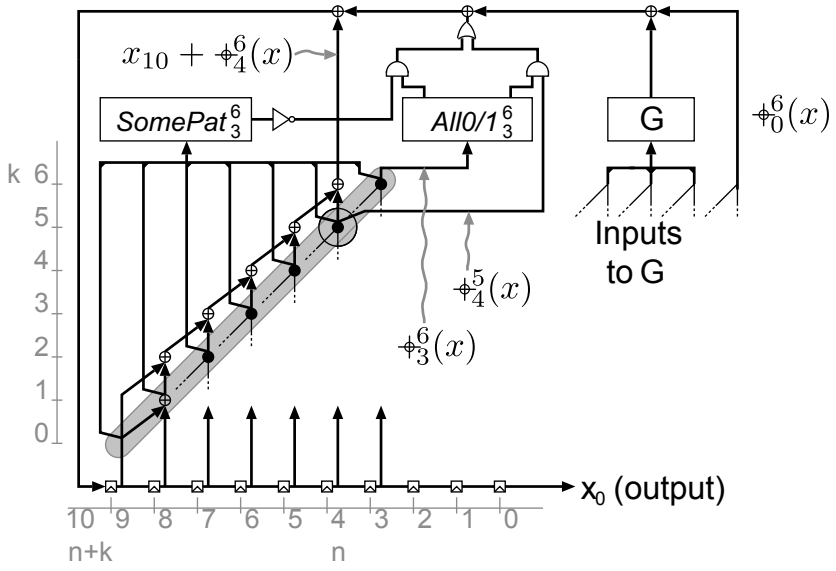


Figure 3.7: The architecture of $(6, 4)$-composited de Bruijn generator after all-zeroes state machine optimization

3.1.5 Parallelization and Retiming

A set of nodes is independent iff we can not determine one node's value from other nodes in the same set. Lemma 9 presents an easy way to test independency if the nodes come from a mesh.

Lemma 9. *Any set consisting of n distinct nodes from the mesh, which are referred to by integers $n, \ldots, 2, 1$ and their values are denoted with variables $v_n, \ldots, v_2, v_1$, is independent iff for all subsets with at least 2 nodes, the following holds: let $i_k, \ldots, i_2, i_1$ be the nodes of the subset, where $k \in \mathbb{Z}, 2 \leq k \leq n$ and $1 \leq i_1 < i_2 < \ldots < i_k \leq n$, then the inequality, $v_{i_k} + v_{i_{k-1}} + \ldots + v_{i_2} + v_{i_1} \neq 0$, is satisfied.*

For example, by Definition 4, we know that $\clubsuit_3^4 = \clubsuit_3^3 + \clubsuit_4^3$, thus

$$\clubsuit_3^4 + \clubsuit_3^3 + \clubsuit_4^3 = 0.$$

Therefore, any set consisting of $\clubsuit_3^4$, $\clubsuit_3^3$ and $\clubsuit_4^3$ is not independent.

Theorem 4. *In (k, n)-mesh, for any $n+k-1$ independent nodes, denoted with $v_{n+k-1}, \ldots, v_2, v_1$, and for any function $Q(x_{[n+k-1:1]})$, we can always find a function F such that $Q(x_{[n+k-1:1]}) = 1$ iff $F(v_{[n+k-1:1]}) = 1$.*

Proof. The proof of Theorem 4 begins by realizing that each node in the mesh can be also written as $\clubsuit_i^j(x)$, $i \geq 1$, which is a sum of elements from x. Therefore, we have $n + k - 1$ linear equations as follows.

$$\begin{aligned}
v_{n+k-1} &= \clubsuit_{i_{n+k-1}}^{j_{n+k-1}}(x) \\
v_{n+k-2} &= \clubsuit_{i_{n+k-2}}^{j_{n+k-2}}(x) \\
\ldots &= \ldots \\
v_2 &= \clubsuit_{i_2}^{j_2}(x) \\
v_1 &= \clubsuit_{i_1}^{j_1}(x), \text{ where } i_1, i_2, \ldots, i_{n+k-1} \geq 1.
\end{aligned}$$

Because $v_{n+k-1}, \ldots, v_2, v_1$ are independent, the equations are linearly independent. With $n + k - 1$ independent linear equations, we can solve exactly $n + k - 1$ unkown variables. The unknown variables are elements from vector x of size $n + k$ and x_0 is not in the mesh, so the unkown variables have to be $x_{n+k-1}, \ldots, x_2, x_1$, that is, the bottom row nodes. We solve the equations and get answers for $x_{n+k-1}, \ldots, x_2, x_1$ which are represented as functions of the vector $v_{[v+k-1:1]}$ made of the known variables, $v_{n+k-1}, \ldots, v_2, v_1$.

$$\begin{aligned}
x_{n+k-1} &= f_{n+k-1}(v_{[n+k-1:1]}) \\
x_{n+k-2} &= f_{n+k-2}(v_{[n+k-1:1]}) \\
\ldots &= \ldots
\end{aligned}$$

$$x_2 = f_2(v_{[n+k-1:1]})$$
$$x_1 = f_1(v_{[n+k-1:1]})$$

$f_{n+k-1}, \ldots, f_2, f_1$ are just a summation of some of its arguments. We use them to construct the function F such that $\mathit{SomePat}^k_{n-1}(x) = 1$ iff $F(v_{[n+k-2:0]}) = 1$:

$$F(v_{[n+k-1:1]}) = Q((f_{n+k-1}(v_{[n+k-1:1]}), \ldots, f_2(v_{[n+k-1:1]}), f_1(v_{[n+k-1:1]}))$$

□

The core of the proof is showing that there are exactly $n+k-1$ independent nodes in (k, n)-mesh. It is intuitively true by the fact that the bottom row of the mesh has $n+k-1$ nodes and we can build up the mesh from the bottom row.

Though in the proof of Theorem 4, we treated a selected set of $n+k-1$ independent nodes as known nodes, as a matter of fact, $x_{[n+k-1:0]}$ are the only known nodes for the $(n+k)$-stage shift register. This is why the functions defined are in terms of the vector x. Particularly, let us take a look up $\mathit{SomePat}^k_{n-1}(x)$. Note that x_0 is never used in the function, so we can also write it as $\mathit{SomePat}^k_{n-1}(x_{[n+k-1:1]})$. We described this function as checking patterns on the diagonal nodes (Definition 12), but it is actually a function over the vector x. To understand this, we view the computation of $\mathit{SomePat}^k_{n-1}(x_{[n+k-1:1]})$ as two steps. The first step is to compute the diagonal nodes from the vector x. The second step is to check patterns on the diagonal nodes. Based on Theorem 4, we find an equivlent function that takes $n+k-1$ independent nodes from the mesh as inputs. But we still have to compute these $n+k-1$ nodes from x. In such way, we essentially make the function we found have x as inputs, thus become a tranformation of the original $\mathit{SomePat}^k_{n-1}(x_{[n+k-1:1]})$. This transformation can be benefitial and entail simplifications. Our experience shows that it can increase clockspeed and save area in hardware implementations (Algorithm 2, Algorithm 6). Moreover, noticing that $\mathit{SomePat}^k_{n-1}$ is only used in conjuction with $\mathit{All0}^k_{n-1}$, by utilizing this fact and the temporal relation between nodes in the mesh, we are able to find some functions that are equivalent to $\mathit{SomePat}^k_{n-1}$ and more importantly they are not transformations of $\mathit{SomePat}^k_{n-1}$. It will be show in the following corollary. We will also show how to apply the corollary in parallelizing OcDeb and increasing OcDeb performance in hardware.

Corollary 4. *If* $\mathit{All0}^k_{n-1} = 1$*, for all* $1 + i < n$*,* $\mathit{SomePat}^k_{n-1} = 1$ *iff* $\mathit{SomePat}^k_{n-1-i} = 1$*.*

Proof. In Corollary 4, the $n+k-1$ independent nodes we pick are the $n-1$ nodes from the top row(nodes that are examined by $\mathit{All0}^k_{n-1}$) and k nodes from

any interior diagonal(nodes that are examined by $SomePat^k_{n-1-i}$). However, we still need to prove these $n+k-1$ nodes are independent from each other. As in the proof of Theorem 4, we can write down $n+k-1$ linear equations and each equation's left hand is just one of the selected nodes. It is not hard to see that given values of the top row nodes and nodes from an interior diagonal, we can compute the values of the remaining nodes in the mesh including $n+k-1$ nodes on the bottom row. In other words, the $n+k-1$ equations must have solutions for the $n+k-1$ elements from x which are $x_{n+k-1}, \ldots, x_2, x_1$. Therefore, the equations are independent, and accordingly the selected nodes which are the equations' left hand sides, are independent. Moreover, because the top row nodes are all 0, the bottom row nodes $x_{n+k-1}, \ldots, x_2, x_1$ are sums of only interior digonal nodes in the solution of the equations. Let the interior digonal nodes be $v_k, \ldots, v_2, v_1$, then

$$\begin{aligned} x_{n+k-1} &= f_{n+k-1}(v_{[k:1]}) \\ x_{n+k-2} &= f_{n+k-2}(v_{[k:1]}) \\ \ldots &= \ldots \\ x_2 &= f_2(v_{[k:1]}) \\ x_1 &= f_1(v_{[k:1]}) \end{aligned}$$

There is a temporal relation between the interior diagonal nodes and the diagonal nodes. Specifically, ${v_k}^{(i)}, \ldots, {v_2}^{(i)}, {v_1}^{(i)}$, that is ${v_{[k:1]}}^{(i)}$, corresponds to the diagonal nodes. If we pick the diagonal nodes and the top row nodes as the $n+k-1$ independent nodes, the corresponding F is

$$\begin{aligned} F(v_{[k:1]}) &= SomePat^k_{n-1-i}((f_{n+k-1}({v_{[k:1]}}^{(i)}), \ldots, f_2({v_{[k:1]}}^{(i)}), f_1({v_{[k:1]}}^{(i)})) \\ &= SomePat^k_{n-1}((f_{n+k-1}(v_{[k:1]}), \ldots, f_2(v_{[k:1]}), f_1(v_{[k:1]})) \end{aligned}$$

To summarize, $SomePat^k_{n-1-i}(x) = 1$ iff $F(v_{[k:1]}) = 1$, which is also

$SomePat^k_{n-1-i}(x) = 1$ iff $SomePat^k_{n-1}((f_{n+k-1}(v_{[k:1]}), \ldots, f_2(v_{[k:1]}), f_1(v_{[k:1]})) = 1$.

By replacing $(f_{n+k-1}(v_{[k:1]}), \ldots, f_2(v_{[k:1]}), f_1(v_{[k:1]})$ with $(x_{n+k-1}, \ldots, x_2, x_1)$, we finally get

$$SomePat^k_{n-1-i}(x) = 1 \text{ iff } SomePat^k_{n-1}(x) = 1.$$

□

Recall that $SomePat^k_{n-1}$ checks if the diagonal nodes satisfy a pattern. On the other hand, $SomePat^k_{n-1-i}$ checks if nodes of an interior diagonal that is made of the i-th(counting from the left) node on each row satisfy a pattern. Figure 3.9 shows *SomePat* examining an interior diagonal and Figure 3.10 shows *SomePat* examining the diagonal of OcDeb-*6-4*. The exact pattern depends on the parity of the sum of the superscript and subscript of *SomePat* (Definition 12). Therefore, $SomePat^k_{n-1}$ and $SomePat^k_{n-1-i}$ may check nodes against different patterns.

We will illustrate two applications of Corollary 4.

Enabling Efficiently Parallelizing OcDeb

We use $x_{n-1} \dashv\!\!\rightarrow x_{n-2} \dashv\!\!\rightarrow \ldots \dashv\!\!\rightarrow x_1 \dashv\!\!\rightarrow x_0 \dashv\!\!\rightarrow$ to denote an n-stage shift register, v to denote the feedback value and $\mapsto$ to denote feeding the value to the shift register. For example, $v \mapsto x_{n-1} \dashv\!\!\rightarrow x_{n-2} \dashv\!\!\rightarrow \ldots \dashv\!\!\rightarrow x_1 \dashv\!\!\rightarrow x_0 \dashv\!\!\rightarrow$ denotes that $x_{n-1}, x_{n-2}, \ldots, x_1, x_0$ is a shift register with v as the feedback value to the stage x_{n-1}. Another concrete example is that the shift register on the left side of Figure 3.8 can be represented with this notation:

$$v \mapsto x_3 \dashv\!\!\rightarrow x_2 \dashv\!\!\rightarrow x_1 \dashv\!\!\rightarrow x_0 \dashv\!\!\rightarrow$$

Shift register is the bottom row of a mesh. Refer to Figure 3.7 and others for how we draw shift register. Also recall that $v^{(i)}$ represents v's value after i clock cycles. To achieve a parallelism of degree d, we decompose the original shift registers to an array of (n/d)-stage shift registers. The array has a length of d and for simplicity, we assume d divides n [1].

The (n/d)-stage shift registers in the array run in parallel and the adjacent registers in them were d positions apart in the original shift register. Mathematically,

the original shift register: $v \mapsto x_{n-1} \dashv\!\!\rightarrow x_{n-2} \dashv\!\!\rightarrow \ldots \dashv\!\!\rightarrow x_1 \dashv\!\!\rightarrow x_0 \dashv\!\!\rightarrow$

the array of shift registers after parallezation:

$$\begin{aligned}
v &\mapsto x_{n-d} \dashv\!\!\rightarrow x_{n-d-d} \dashv\!\!\rightarrow \ldots \dashv\!\!\rightarrow x_0 \dashv\!\!\rightarrow \\
v^{(1)} &\mapsto x_{n-d+1} \dashv\!\!\rightarrow x_{n-d-d+1} \dashv\!\!\rightarrow \ldots \dashv\!\!\rightarrow x_1 \dashv\!\!\rightarrow \\
\ldots &\mapsto \ldots \\
v^{(d-2)} &\mapsto x_{n-2} \dashv\!\!\rightarrow x_{n-2-d} \dashv\!\!\rightarrow \ldots \dashv\!\!\rightarrow x_{d-2} \dashv\!\!\rightarrow \\
v^{(d-1)} &\mapsto x_{n-1} \dashv\!\!\rightarrow x_{n-1-d} \dashv\!\!\rightarrow \ldots \dashv\!\!\rightarrow x_{d-1} \dashv\!\!\rightarrow .
\end{aligned}$$

[1] If d does not divide n, shift register from the array will not have the same length and the array looks different from the one below. However, our techniques introduced here for efficient parallelization still work.

For example, let us double the throughput of the 4-stage shift register with the feedback function $v = x_1 + x_2 + x_3$, shown in Figure 3.8. Mathematically, the array of shift registers will be,

$$v \mapsto x_2 \shortrightarrow x_0 \shortrightarrow$$
$$v^{(1)} \mapsto x_3 \shortrightarrow x_1 \shortrightarrow .$$

v's value in the next clock cycle, $v^{(1)} = v(x_{[:1]}) = x_2 + x_3 + v$.

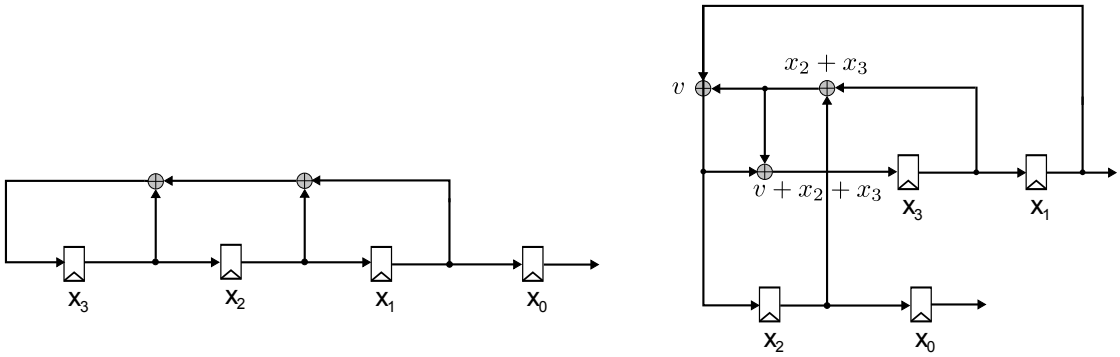


Figure 3.8: An example of parallelizing feedback shift register

Also shown in Figure 3.8, achieving a parallelism of 2 does not necessarily mean we have another complete copy of the feedback function as they may share common sub-expressions, for example $x_2 + x_3$ is shared by two feedback functions in the picture. Identifying and utilizing the common sub-expressions among the feedback functions to save computations is important for an efficient parallelization.

Parallelization increases delay when the stage that has the feedback value as input is one of the inputs to the feedback function. In Figure 3.8, originaly, the longest combinational path only has 2 XOR gates connected in serial. But after parallelization, the number becomes 3 which happens on the path from x_3 to its input port.

And of course, simplifications may exist. For our simple example,

$$\begin{aligned} v^{(1)} &= v(x_{[:1]}) \\ &= x_2 + x_3 + v \\ &= x_2 + x_3 + x_1 + x_2 + x_3 \\ &= x_1 \end{aligned}$$

Let us look at $mesh_n^k$ now. Again, we assume the feedback value is v in the composited de Bruijn sequence generator. By $v \,\&\, x$, we concatenate v with vector x. For (k, n)-composited de Bruijn sequence generators,

$$\begin{aligned} mesh_n^k(x) &= All0_{n-1}^k(x) + \sum_{j=0}^{k-1} X_{n+k-1-j}^j(x) \\ mesh_n^{k\,(1)}(x) &= mesh_n^k(v \,\&\, x_{[:1]}) \\ &= All0_{n-1}^k(v \,\&\, x_{[:1]}) + \sum_{j=0}^{k-1} X_{n+k-1-j}^j(v \,\&\, x_{[:1]}) \end{aligned}$$

$v \,\&\, x_{[:1]}$ is offset by 1 from x. As a result, same index i into $v \,\&\, x_{[:1]}$ and x will have different indexes when they are put into the same coordinate system. To be exact, the indexes are $i + 1$ and i respectively, having the opposite parity. If we assume $n + k$ is an even number, then $X_{n+k-1}^0(x) = x_{n+k-1}(x_{n+k-2} + 1) \dots (x_2 + 1)x_1$, while $X_{n+k-1}^0(v \,\&\, x_{[:1]}) = v(x_{n+k-1} + 1)x_{n+k-2} \dots x_2(x_1 + 1)$. We do not find common sub-expressions between them. For higher value X, for example,

$$\begin{aligned} X_{n+k-j-1}^j(x) &= X_{n+k-j-1}^0 \circ \dot{+}_0^j(x) \\ &= X_{n+k-j-1}^0(\dot{+}_{n+k-j-1}^j(x) \dots, \dot{+}_1^j(x), \dot{+}_0^j(x)), \\ X_{n+k-j-1}^j(v \,\&\, x_{[:1]}) &= X_{n+k-j-1}^0 \circ \dot{+}_0^j(v \,\&\, x_{[:1]}) \\ &= X_{n+k-j-1}^0(\dot{+}_{n+k-j}^j(v \,\&\, x), \dot{+}_{n+k-1-j}^j(x) \dots, \dot{+}_2^j(x), \dot{+}_1^j(x)). \end{aligned}$$

$\dot{+}_1^j(x), \dot{+}_2^j(x), \dots, \dot{+}_{n+k-j-1}^j(x)$ are part of the inputs to the function $X_{n+k-j-1}^0$ and they are shared.

$All0$ function is very amenable to parallelization as will be seen below.

$$\begin{aligned} All0_{n-1}^k(x) &= (\dot{+}_1^k(x) + 1)(\dot{+}_2^k(x) + 1) \dots (\dot{+}_{n-1}^k(x) + 1) \\ All0_{n-1}^k(v \,\&\, x_{[:1]}) &= (\dot{+}_2^k(x) + 1)(\dot{+}_3^k(x) + 1) \dots (\dot{+}_{n-1}^k(x) + 1)\dot{+}_n^k(v \,\&\, x) \end{aligned}$$

More shared terms can be found between $mesh_n^{k\,(2)}$ and $mesh_n^k$, but we will not go into details. For OcDeb-k-n, with the help of our optimizations, most of the terms are shared even between $mesh_n^{k\,(1)}$ and $mesh_n^k$.

$$
\begin{aligned}
mesh_n^k(x) &= \Big(\big(SomePat_{n+1}^{k-2}(x)+1\big)All0_{n-1}^{k}(x)\Big) \text{ or } \Big(\dotplus_n^{k-1}(x)All1_{n-1}^{k}(x)\Big),\\
mesh_n^k(x)^{(1)} &= mesh_n^k(v \,\&\, x_{[:1]})\\
&= \Big(\big(SomePat_{n+1}^{k-2}(v \,\&\, x_{[:1]})+1\big)All0_{n-1}^{k}(v \,\&\, x_{[:1]})\Big) \text{ or}\\
&\quad \Big(\dotplus_n^{k-1}(v \,\&\, x_{[:1]})All1_{n-1}^{k}(v \,\&\, x_{[:1]})\Big)\\
&= \Big(\big(SomePat_{n}^{k-2}(v \,\&\, x_{[:1]})+1\big)All0_{n-1}^{k}(v \,\&\, x_{[:1]})\Big) \text{ or}\\
&\quad \Big(\dotplus_{n+1}^{k-1}(v \,\&\, x)All1_{n-1}^{k}(v \,\&\, x_{[:1]})\Big), \text{ according to Corollary 4.}
\end{aligned}
$$

We already know that $All0_{n-1}^{k}$ is very amenable to parallelization. So is $All1_{n-1}^{k}$. $SomePat_{n+1}^{k-2}(x)$ checks nodes of the diagonal, while $SomePat_n^{k-2}(x)$ checks the interior diagonal that is adjacent to the diagonal. And according to Corollary 4, we have the relation,

$$SomePat_{n+1}^{k-2}(x) = SomePat_n^{k-2}(x).$$

$SomePat_n^{k-2}(v \,\&\, x_{[:1]})$ and $SomePat_{n+1}^{k-2}(x)$ check patterns on the same diagonal because the input of $SomePat_n^{k-2}(v \,\&\, x_{[:1]})$ have the shift register's values in the next clock cycle. When they check the same diagonal, their relation becomes

$$SomePat_n^{k-2}(v \,\&\, x_{[:1]}) = (SomePat_n^{k-2}(x)+1)(1+\prod_{j=0}^{k}(\dotplus_{n+k-1-j}^{j}(x)+1))$$

$1+\prod_{j=0}^{k}(\dotplus_{n+k-1-j}^{j}(x)+1)$ is equal to 1 when not all diagonal nodes are 0s. It may seem to be not shared and thus seem to be a big overhead. However, our implementation of $SomePat_{n+1}^{k-2}(x)$ already has the computation of $\prod_{j=2}^{k}(\dotplus_{n+k-1-j}^{j}(x)+1)$.

To conclude, our optimizations enable efficient parallelization of OcDeb by promoting sharing of computations among feedback functions.

Enabling Retiming

In this section, we illustrate Corollary 4's positive impact on applying retiming in hardware implementation. Retiming is a technique that helps improve clockspeed.

In hardware implementations, *All*0 and *All*1 can be computed by a state machine while *SomePat* and G are purely combinational. The critical path in the design from Figure 3.7 goes through *SomePat*. In this section we prove Corollary 4, which says that *SomePat* may be applied to any diagonal of the mesh: either the leftmost diagonal as shown in Figure 3.7, or an interior diagonal as shown in Figure 3.9.

We use Corollary 4 to increase the clockspeed and reduce the area of parallel implementations.

We increase the clockspeed by using Corollary 4 to allow *SomePat* to use the first interior diagonal (Figure 3.9), then use conventional retiming to shift the inputs back to the left and add a register on the output of *SomePat* (Figure 3.10). Figure 3.10 is the final diagram for our optimized composited de Bruijn sequence generator of throughput 1. For our OcDeb-*24-40* generator, retiming increased our clockspeed from 1.02 GHz to 1.35 GHz because the critical path is cut into two paths with shorter delay by putting the register after *SomePat*.

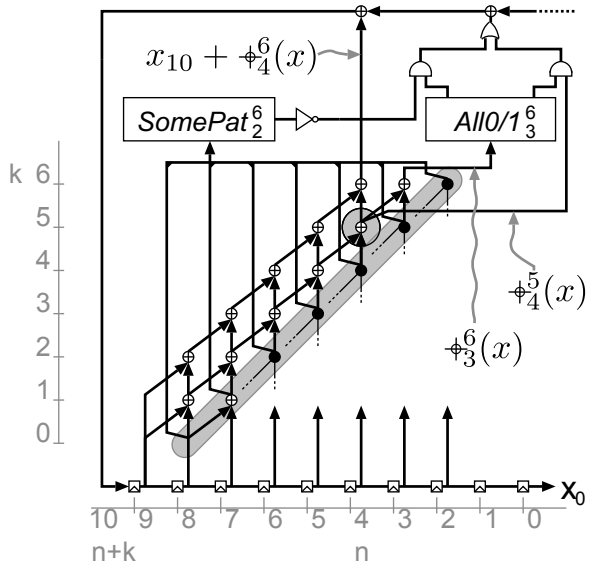


Figure 3.9: Preparation for retiming: shift diagonal to right

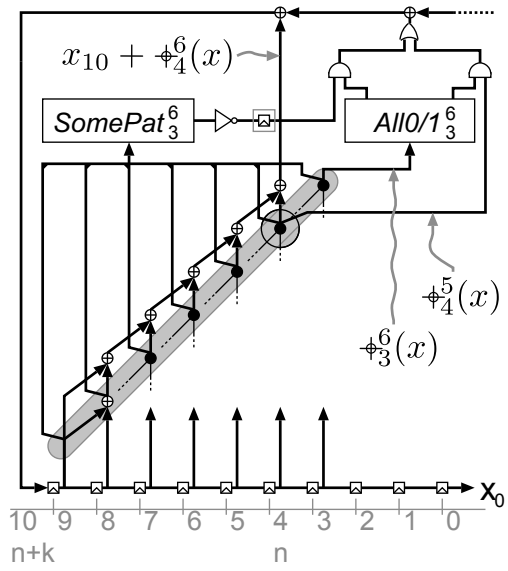


Figure 3.10: After retiming: an architecture of OcDeb-*6-4*

Retiming is a common technique used in hardware implementations to increase clock-speed. The role of our optimizations, especially Corollary 4, is to empower this technique to be applied in a way that would not be correct without Corollary 4's support.

3.2 The New Complexity

Let us calculate how many bit operations are needed for the computation of $mesh_n^k = All0_{n-1}^k(x) + \sum_{j=0}^{k-1} X_{n+k-1-i}^j(x)$ for OcDeb-*k*-*n*. We will calculate the complexity based on the calculation of $mesh_n^k$ described by Theorem 3 as well as the way to compute All0/1 suggested in Section 3.1.4.

For OcDeb-*k*-*n*, recall that we need to only examine the leftmost node in $k+1$ rows and the technique we use to implement a set of nodes is to compute the simplified expression for each node. We know that the simplified expression for $\not\oplus_i^k(x)$ includes $2^{H(k)} - 1$ XOR gates (Corollary 1). We use *len*(k) to denote the length of the vector for k (number of bits needed to represent k). Recall that log ()'s base is 2.

$$len(k) = \begin{cases} \log(k) + 1, & \text{when } k \text{ is a power of 2} \\ \lceil \log(k) \rceil, & \text{otherwise} \end{cases}$$

The Hamming weight of an integer is the number of 1s in the integer's binary representation. There are $\binom{l}{i}$ possible integers of length l and having i 1s. This is a counting problem that is analogous to counting how many ways there are to give out i cans to l people with the rule that each person can have at most 1 can.

We compute the diagonal nodes from bottom to top implying that before we compute another diagonal node, all diagonal nodes below it have already been computed. We take common sub-expression elimination into consideration when computing the complexity of bit operations. Common sub-expression elimination is a technique that identifies shared sub-expressions among formula or expressions so that the shared sub-expressions are calculated only once instead of being calculated repeatedly in every formula that has them.

For example, $\not\oplus_4^7(x)$ is the diagonal node we want to compute now and we have already computed $\not\oplus_5^6(x), \not\oplus_6^5(x), \ldots, \not\oplus_{10}^1(x)$. We find out that there are sub-expressions (common subexpressions are underlined) in $\not\oplus_5^6(x)$ and $\not\oplus_5^6(x)$ that also appear in $\not\oplus_4^7(x)$.

$$\oplus_4^7(x) = \underline{x_5 + x_7 + x_9 + x_{11}} + \underline{x_6 + x_{10}} + x_8 + x_4$$
$$\oplus_5^6(x) = \underline{x_5 + x_7 + x_9 + x_{11}}$$
$$\oplus_6^5(x) = \underline{x_6 + x_{10}} + x_7 + x_{11}$$

We generalize the above example. Let's say we want to compute the diagonal node on the k_1's row, $\oplus_{n_1}^{k_1}$.

$$k_1 = 2^{i_1} + 2^{i_2} + \ldots + 2^{i_{H(k_1)}},\ i_1 < i_2 < \ldots < i_{H(k_1)}.$$

It shares common sub-expressions with some diagonal nodes below it. They are:

$$\oplus_{n_1+2^{i_1}}^{k_1-2^{i_1}}, \oplus_{n_1+2^{i_2}}^{k_1-2^{i_2}}, \ldots, \oplus_{n_1+2^{i_{H(k_1)}}}^{k_1-2^{i_{H(k_1)}}}.$$

Recall that based on Corollary 3, there is one-to-one correspondence between x elements' indexes and the objects from the power set of $\{2^{i_1}, 2^{i_2}, \ldots, 2^{i_{H(k)}}\}$. Same x elements have the same index.

Between $\oplus_{n_1}^{k_1}$ and $\oplus_{n_1+2^{i_1}}^{k_1-2^{i_1}}$, to compensate the difference in the subscript, we want to pick the objects from the power set of $\{2^{i_1}, 2^{i_2}, \ldots, 2^{i_{H(k)}}\}$ that contain 2^{i_1}. There are $2^{H(k)-1}$ such objects of the power set, corresponding to $2^{H(k)-1}$ x elements that are shared by these two nodes, resulting $2^{H(k)-1} - 1$ shared XOR operations. We proceed to consider the shared computations between $\oplus_{n_1}^{k_1}$ and $\oplus_{n_1+2^{i_2}}^{k_1-2^{i_2}}$. Similarly, to make up for the difference in the subscript, we look for objects from the power set that have 2^{i_2} in them. Besides, we do not want 2^{i_1} in these objects becaues we just considered this case. There are $2^{H(k)-2}$ such objects of the power set, corresponding to $2^{H(k)-2}$ x elements that are shared by $\oplus_{n_1}^{k_1}$ and $\oplus_{n_1+2^{i_2}}^{k_1-2^{i_2}}$ excluding the x elements that are shared between $\oplus_{n_1}^{k_1}$ and $\oplus_{n_1+2^{i_1}}^{k_1-2^{i_1}}$. Then we know there are $2^{H(k)-2} - 1$ more shared XOR operations. We can continue and in the end, the total number of shared XOR operations is,

$$2^{H(k)-1} - 1 + 2^{H(k)-2} - 1 + \ldots + 2^{H(k)-H(k)} - 1 = 2^{H(k)} - H(k) - 1.$$

According to Corollary 1, to directly compute $\oplus_{n_1}^{k_1}$, we need $2^{H(k)} - 1$ XOR operations. Taking the common sub-expression into consideration, we realize the actual number of XOR operations is

$$2^{H(k)} - 1 - (2^{H(k)} - H(k) - 1) = H(k).$$

Recall that for $0 \leq i \leq len(k)$, we have $\binom{len(k)}{i}$ rows with Hamming weight i, which means $\binom{len(k)}{i}(2^i - 1)$ XOR gates. Hence, we have $\sum_{i=0}^{len(k)} \binom{len(k)}{i} i$ XOR gates all together.

When k is a power of 2, the exact total number of XOR gates is:

$$1 + \sum_{i=0}^{\log(k)} \binom{\log(k)}{i} i = 2^{\log(k)-1}\log(k) = \frac{k\log(k)}{2}$$

When $k \neq 2^{len(k)}$, the upper bound on the number of XOR gates is $\sum_{i=0}^{\lceil\log(k)\rceil} \binom{\lceil\log(k)\rceil}{i} i$ which can be simplified to $2^{\lceil\log(k)\rceil-1}\lceil\log(k)\rceil$ in the same way. Because $\lceil\log(k)\rceil$ is no smaller than $\log(k)$ and $f(x) = 2^{x-1}x$ monotonically increases after x is bigger than 0, the upper bound of both cases is $2^{\lceil\log(k)\rceil-1}\lceil\log(k)\rceil$.

To acquire total XOR gates' $\mathcal{O}()$ notation in terms of variable k, we eliminate ceilings in the upper bound expression based on the fact that $\lceil\log(k)\rceil < \log(k) + 1$. After we replace $\lceil\log(k)\rceil$ with $\log(k) + 1$ in the recently computed upper bound, which is $2^{\lceil\log(k)\rceil-1}\lceil\log(k)\rceil$, we get $\mathcal{O}(k\log(k))$. Therefore, the complexity to compute the diagonal nodes is $\mathcal{O}(k\log(k))$.

As described in Section 2.3.1, an alternative to computing each node individually and directly is to compute them recursively, or alternatively speaking, with a mesh. To compute the $k+1$ nodes on the diagonal with a mesh, we need to construct a triangular mesh with base of $k+1$ nodes. Each row has 1 fewer node than the row right below it and the quantity of XOR needed to derive a row from the row below it is the length of it. The base row is made of shift register, so no XOR is needed to compute its nodes' values. Therefore, the total number of XOR in the mesh is $k+k-1+\ldots+2 = (k+2)(k-1)/2 = k^2/2+k/2-1$, which is asymptotically larger than the the complexity $\mathcal{O}(k\log(k))$. For real world instances where typical values of k are relatively large, e.g. 32, 40, the number of XOR in the mesh is larger than the XOR needed to compute nodes of the diagonal one by one independently.

The state machine that produces values for $All0^k_{n-1}$ and $All1^k_{n-1}$ includes a counter and comparators. All of the components of the state machine have a number of bit operations that is linear in terms of the length of the counter, which is $\log(n)$. The number of bit operations in computing *SomePat* after the diagonal nodes being computed is linear in k. This will become clear in Section 4.2.4.

In summary, the upper bound on the number of XOR gates required in the computation of $mesh^k_n$ is $\mathcal{O}(k\log(k))$. For the state machine that produces values for $All0^n_{k-1}$ and $All1^n_{k-1}$, the complexity of bit operations is $\mathcal{O}(\log(n))$. Thus, with our optimizations, we have reduced the asymptotic number of bit-operations for the composited construction technique from $\mathcal{O}(k^2+nk)$ to $\mathcal{O}(k\log(k)+\log(n))$, with small overhead in storage, $\mathcal{O}(\log n)$,

for the counter that is used to compute $All0$ and $All1$.

Theorem 5. *For OcDeb-k-n, the number of bit operations used to compute* $mesh_n^k$ *has an upper bound of* $\mathcal{O}(k \log(k) + \log(n))$.

3.3 Notes

The above complexity analysis is based on computing the diagonal nodes directly and seperately. Our way to find common sub-expressions for diagonal nodes yields the minimal number of XOR gates required to calculate them. Later in Section 4.1.6, we exploit the trade-offs between space and computation in order to compute the diagonal nodes even more efficiently in area and performance. The main challenge in doing this is identifying places where it is beneficial. It is complicated because of the fact that common sub-expression elimination and trading space for computation often compete with each other in the sense that one may disallow the other. We will simplify this complication in Section 4.1.6 by taking only a subset of all common sub-expressions into consideration so that identifying places where trading space for computation is beneficial becomes manageable.

Now we conclude the optimization chapter. In short, our optimizations exploit the characteristics of the mesh structure to simplify the fairly complex boolean formula of $mesh_n^k$. Especially Corollary 4 which states that SomePat may be applied to interior diagonals as well extends Theorem 3 and makes our optimizations even more powerful. We show its usage in hardware retiming and parallization as examples. The optimizations are stated in theorems and their key proof steps are provided. We intentionaly try to leave out implementation details when possible. In the next chapter, we show how to efficiently implement it in hardware.

Chapter 4

Hardware Implementations

Up to now, we have presented the backbone of efficient and fast de Bruijn sequence generator design. In this section, we provide tricks (Section 4.1), details on hardware implementations and rationale for some decisions we made to achieve even smaller and faster hardware (Section 4.2).

4.1 Tricks

Recall that we use $v^{(i)}$ to denote v's value after i clock cycles and v can be either a function or a variable.

Also recall that $x_{n-1} \twoheadrightarrow x_{n-2} \twoheadrightarrow \ldots \twoheadrightarrow x_1 \twoheadrightarrow x_0 \twoheadrightarrow$ denotes a shift register of n stages, and x_{n-1} is connected to data input. Then for any non-negative integer n_1, k_1, i and δ such that $i + \delta \leq n - 1$ and $k_1 + n_1 + \delta \leq n - 1$, we have:

$$\begin{aligned} x_i^{(\delta)} &= x_{i+\delta}, \text{ and} \\ \text{⧺}_{n_1}^{k_1}{}^{(\delta)} &= \text{⧺}_{n_1+\delta}^{k_1}. \end{aligned}$$

The second equation holds because as defined in Definition 4 $\text{⧺}_{n_1+\delta}^{k_1} = \text{⧺}_{n_1}^{k_1}(x_{[:\delta]})$, where the right hand side has a shift register x shifted to the right δ positions as the input. Examples can be found in the following sections.

4.1.1 Trade-off Between Storage and Computation

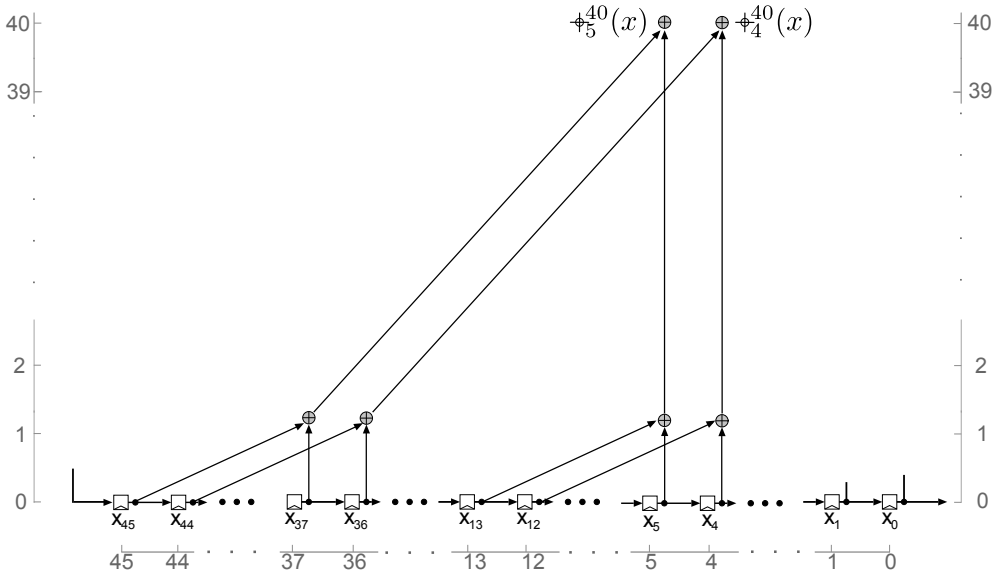
Replacing computation (i.e. combinational logic gates) with storage (i.e. FLIP-FLOPs)and vice versa is possible when the computation has some temporal properties.

Example1. To compute $\oplus_5^{40}$ and $\oplus_4^{40}$, having analyzed in Corollary 2, it is cheaper to compute each one's value directly in its most simplified form,

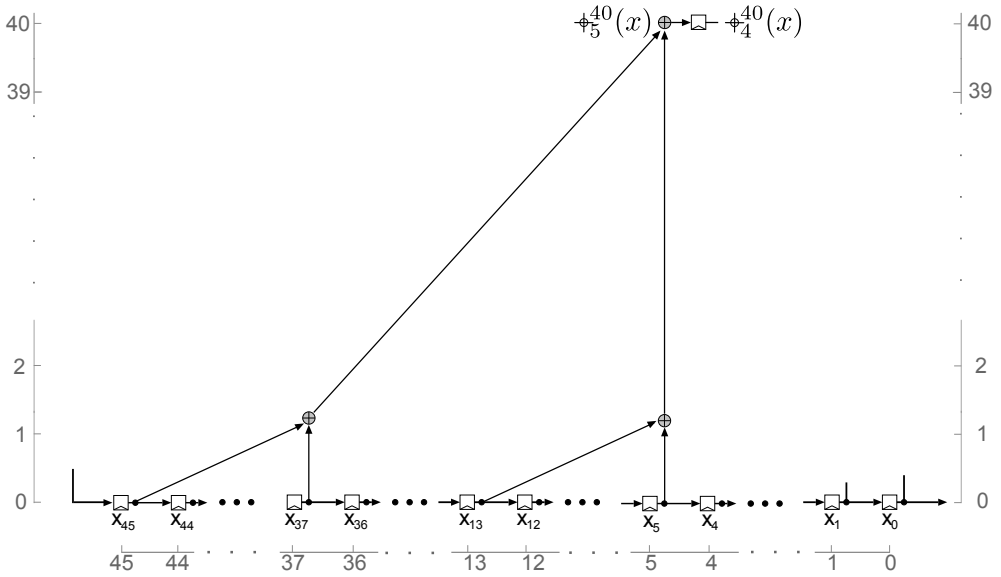
$$\begin{aligned} \oplus_5^{40} &= x_5 + x_{13} + x_{37} + x_{45}, \\ \oplus_4^{40} &= x_4 + x_{12} + x_{36} + x_{44}. \end{aligned}$$

In CMOS 65nm technology, a two-input XOR gate is approximately 2 GE while a simple FLIP-FLOP is about 4 GE. $\oplus_4^{40}$ and $\oplus_5^{40}$ have adjacent indexes and as seen in Figure 4.1a, they are adjacent nodes. As it is more efficient to compute each node directly, we delete the mesh structure in Figure 4.1a. However, the coordinate system is still preserved. $\oplus_4^{40}$ is equal to registered $\oplus_5^{40}$ and it can save area if $\oplus_4^{40}$ is computed in such a way (Figure 4.1b) because 1 FLIP-FLOP is smaller than 3 XOR gates. We may think of this technique as replacing combinational logic with sequential logic or trading space for computations. We rewrite the above as follows.

$$\begin{aligned} \oplus_5^{40} &= x_5 + x_{13} + x_{37} + x_{45}, \\ \oplus_4^{40(1)} &= \oplus_5^{40}. \end{aligned}$$



(a) The direct way to compute $\oplus_5^{40}$ and $\oplus_4^{40}$



(b) Computing $\oplus_4^{40}$ with a FLIP-FLOP from $\oplus_4^{40}$

Figure 4.1: Computing (a) $\oplus_5^{40}$ and $\oplus_4^{40}$ directly and (b) $\oplus_4^{40}$ with a FLIP-FLOP

4.1.2 Common Sub-expression Elimination

Common sub-expression elimination identifies shared terms among formula so that the shared sub-expressions are only calculated once instead of being calculated repeatedly in every formula that has them. In Section 3.2, we use our knowledge of the common sub-expressions among diagonal nodes to compute a tight upper bound of bit operations for OcDeb-k-n. Here we focus on how it will affect our decisions on where to apply the trick which we call "replacing combinational logic with sequential logic". Besides that, we also look at common sub-expressions among nodes on the same row.

Example 2. We are about to compute $\boxplus_5^{38}$ and $\boxplus_3^{38}$.

$$\begin{aligned} \boxplus_5^{38} &= x_5 + x_7 + x_9 + x_{11} + x_{37} + x_{39} + x_{41} + x_{43}, \\ \boxplus_3^{38} &= x_3 + x_5 + x_7 + x_9 + x_{35} + x_{37} + x_{39} + x_{41}. \end{aligned}$$

If put into a figure, we will see $\boxplus_3^{38}$ and $\boxplus_5^{38}$ are two nodes apart on the same row. It is tempting to utilize this fact, which also can be translated to $\boxplus_3^{38}$'s current value is equal to $\boxplus_5^{38}$'s value two clock cycles ago. In such way, we need 7 XOR gates to compute $\boxplus_5^{38}$ and 2 FLIP-FLOPs to compute $\boxplus_5^{38}$, which requires 22 GE altogether. However, there is a better way. Observing there are shared expression between $\boxplus_5^{38}$ and $\boxplus_3^{38}$ above, we introduce a fresh variable to represent the shared expression, $v = x_5 + x_7 + x_9 + x_{37} + x_{39} + x_{41}$. Then we rewrite the original formula as follows:

$$\begin{aligned} v &= x_5 + x_7 + x_9 + x_{37} + x_{39} + x_{41}, \\ \boxplus_5^{38} &= x_{11} + x_{43} + v, \\ \boxplus_3^{38} &= x_3 + x_{35} + v. \end{aligned}$$

In this way, we need 9 XOR gates and 0 FLIP-FLOP, resulting in 18 GE area.

4.1.3 Using the Two Techniques Together

The following example will show that it is possible and could be beneficial if we apply the two aforementioned techniques together.

Example 3. We are about to compute $\boxplus_4^{7}$ and $\boxplus_5^{6}$.

$$\begin{aligned} \boxplus_4^{7} &= x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} + x_{11}, \\ \boxplus_5^{6} &= x_5 + x_7 + x_9 + x_{11}. \end{aligned}$$

As in Example 2, but without the need to introduce fresh variables, we rewrite the above:

$$\begin{aligned} \boxplus_4^{7} &= x_4 + x_6 + x_8 + x_{10} + \boxplus_5^{6}, \\ \boxplus_5^{6} &= x_5 + x_7 + x_9 + x_{11}. \end{aligned}$$

As in Example 1, we realize that $x_4 + x_6 + x_8 + x_{10}$ has $\dotplus_5^6$'s value from the previous clock cycle and it is the same as $\dotplus_4^6$. Thus, we are able to further rewrite them.

$$\begin{aligned} \dotplus_4^7 &= \dotplus_4^6 + \dotplus_5^6, \\ \dotplus_5^6 &= x_5 + x_7 + x_9 + x_{11}, \\ \dotplus_4^{6(1)} &= \dotplus_5^6. \end{aligned}$$

$\dotplus_4^7 = \dotplus_4^6 + \dotplus_5^6$ can also be derived based on the definition of $\dotplus_n^k$ (in Definition 4). In this way, we get the minimum area which is 4 XOR gates and 1 FLIP-FLOP, resulting in 12 GE altogether.

4.1.4 Our Goal

These tricks can be used to efficiently (in terms of area) compute nodes on the diagonal and inputs to function G (a subset of top row nodes). For OcDeb-k-n, we state computation tasks:

Computing the inputs to G: $\dotplus_{i_1}^k, \dotplus_{i_2}^k, \ldots, \dotplus_{i_n}^k$, where $i_1 > i_2 > \ldots > i_n$

Computing the nodes on the diagonal: $\dotplus_{n+k-1}^0, \dotplus_{n+k-2}^1, \ldots, \dotplus_{n-1}^k$.

Our goal is to find a way to do the computations so that it will lead to the smallest possible area. The techniques at hand, namely common sub-expression elimination and replacing combinational logic with sequential logic, are already described in the Example 1 to 3. We will consider each computation task seperately in Section 4.1.6 and Section 4.1.7.

4.1.5 Discussion

In real world, the actual area of hardware library components(such as XOR gate, FLIP-FLOP) depends on versatile factors, for example, fan-in and fan-out. Besides, there are variants of basic libarary components, such as 3-input XOR gate which is smaller than 2 2-input XOR gates. Synthesis tools also have the ability to do common sub-expression elimination as well as the ability to restructure the cricuit to meet user-specified area and clockspeed goals. However, the tools are not able to apply our other optimization, which we phrased as "replacing combinational logic with sequential logic". Essentially, we use simplified area estimation XOR gate and FLIP-FLOP and knowledge of where common sub-expression elimination could happen to find the places where substituting the

combinational logic with sequential logic can lead to area saving. Most of the time, we entrust the synthesis tools with optimizing combinational logic including applying common sub-expression elimination, as it usually turns out to work better based on our experience.

4.1.6 Computing the Nodes on the Diagonal

We propose Algorithm 1, which takes in integer k, that is the degree of composition and print out messages that will instruct how to compute the nodes on the diagnal. Note that we do not compute the leftmost node on the top row though it is on the diagonal. This is because the diagonal nodes computed here are the inputs to *SomePat* and *SomePat* is only used in conjuction with *All*0 which implies that the top row nodes including the leftmost one are all 0s.

Algorithm 1 is a greedy algorithm. For each i, we compare four options of computing $\pitchfork^i_{n+k-1-i}$, and then pick the one leading to the smallest area.

Option 1. $\pitchfork^i_{n+k-1-i} = \pitchfork^i_{n+k-1-i}$.
It represents that we unfold it according to Definition 4 to its most simplified form which is a summation of elements from x.

Option 2. $\pitchfork^i_{n+k-1-i} = \pitchfork^{i-2^j}_0 \circ \pitchfork^{2^j}_{n+k-1-i}$, where $i > 2^j$ and $i < 2^{j+1}$.
It represents that we treat a function of degree i as two functions of lower degrees, $i - 2^j$ and 2^j, composited together. We unfold the function of degree $i - 2^j$, which results in a summation of terms from $\pitchfork^{2^j}_{n+k-1-i}, \pitchfork^{2^j}_{n+k-i}, \pitchfork^{2^j}_{n+k+1-i} \cdots, \pitchfork^{2^j}_{n+k-1-2^j}$.

Option 3. $\pitchfork^i_{n+k-2-i}{}^{(1)} = \pitchfork^i_{n+k-1-i}$.
$\pitchfork^i_{n+k-2-i}$'s value in the next clock cycle is the current value of $\pitchfork^i_{n+k-i-1}$. Thus, in hardware, $\pitchfork^i_{n+k-2-i}$ is the output of a register whose input is $\pitchfork^i_{n+k-1-i}$.

Option 4. $\pitchfork^i_{n+k-1-i}{}^{(1)} = \pitchfork^{i-1}_{n+k-i} + \pitchfork^{i-1}_{n+k-i-1}$.
It is usually chosen when $\pitchfork^{i-1}_{n+k-i}$'s value is known.

Algorithm 1 Computing the Diagonal Nodes

1: **function** EFF-DIAG-V1(k)
2: $i \leftarrow 0$
3: $j \leftarrow 0$
4: **while** $i < k$ **do**
5: **if** $i < 2^{j+1}$ **then**
6: **if** $i \mod 2 = 0$ **then**
7: **if** $i = 0$ **then**
8: print "$\phi^{i}_{n+k-1-i} = \phi^{i}_{n+k-1-i}$"
9: **else**
10: print "$\phi^{i}_{n+k-1-i} = \phi^{i-2^j}_{0} \circ \phi^{2^j}_{n+k-1-i}$"
11: **end if**
12: **else**
13: print "$\phi^{i}_{n+k-1-i} = \phi^{i-1}_{n+k-i} + \phi^{i-1}_{n+k-i-1}$"
14: **if** ff-cost($\phi^{i-1}_{n+k-i-1}$) $\leq$ xor-cost($\phi^{i-1}_{n+k-i-1}$) **then**
15: print "$\phi^{i-1}_{n+k-1-i}{}^{(1)} = \phi^{i-1}_{n+k-i}$"
16: **else**
17: print "$\phi^{i-1}_{n+k-1-i} = \phi^{i-1-2^j}_{0} \circ \phi^{2^j}_{n+k-1-i}$"
18: **end if**
19: **end if**
20: **else**
21: print "$\phi^{i}_{n+k-1-i} = \phi^{i}_{n+k-1-i}$"
22: $j \leftarrow j + 1$
23: **end if**
24: $i \leftarrow i + 1$
25: **end while**
26: **end function**

Let us explain key steps in the Algorithm 1.

Line 5: Tests whether $i = 2^{j+1}$. If this condition is true, Option 1 is better than Option 2 based on Lemma 10.

Line 6: Tests whether i is even. When i is odd, Option 4 is chosen where we exercise common sub-expression elimination (Line 13). For example, $\phi^{5}_{3} = \phi^{4}_{4} + \phi^{4}_{3}$, so ϕ^{5}_{3} share ϕ^{4}_{4} with ϕ^{4}_{4}. Note that ϕ^{4}_{4} is computed prior to ϕ^{5}_{3} and there are no more shared expressions between ϕ^{4}_{4} and ϕ^{4}_{3}.

Line 14: Option 3 is explicitly compared to Option 2 for the computation of $\oplus^{i-1}_{n+k-i-1}$ when i is odd. Because $i-1$ is even at Line 14, xor-cost computes the area of computing $\oplus^{i-1}_{n+k-i-1}$ by Option 2. Based on Corollary 5, the total number of XOR gates is $2^{H(i-1)-1}$, where $2^j < i < 2^{j+1}$. ff-cost of $\oplus^{i-1}_{n+k-i-1}$ is 1 FLIP-FLOP because it is next to $\oplus^{i-1}_{n+k-i}$, which has already been computed. With the information of the area corresponding to XOR gate and FLIP-FLOP, we can unify the metrics of the area of XOR gate and FLIP-FLOP and then compare these two options. We model the area of XOR and FLIP-FLOP as constants, though this deviates from the reality.

Lemma 10. *For the computation of $\oplus^{i}_{n+k-1-i}$, Option 1 is preferred over Option 2 iff i is a power of 2.*

Proof. We prove Lemma 10 by induction.

Base Step: When $i = 2^j + 1$, according to Corollary 1, Option 1 requires $2^{H(2^j+1)} - 1 = 3$ XOR gates. According to Option 2, $\oplus^{i}_{n+k-1-i}$ is computed by

$$\oplus^{1}_{0}((\ldots, \oplus^{2^j}_{n+k-i}, \oplus^{2^j}_{n+k-1-i})) = \oplus^{2^j}_{n+k-i} + \oplus^{2^j}_{n+k-1-i}.$$

Note that $\oplus^{2^j}_{n+k-i}$ can also be rewritten as $\oplus^{i-1}_{n+k-i}$, which has already been computed because it is a node on the diagonal and it is below the node we are currently considering. Therefore, we only need to compute $\oplus^{2^j}_{n+k-i-1}$, which requires $2^{H(2^j)} - 1 = 1$ XOR gates. In total, we need $1 + 2^{H(i-2^j)} - 1 = 2$ XOR gates. This is one XOR gate fewer than Option 1, therefore prefered.

Induction Step: Assume that we have found i_1 such that for all $2^j + 1 \leq i_1 < 2^{j+1} - 1$, Option 2 is preferred. It implies that we have computed

$$\oplus^{2^j}_{n+k-1-2^j}, \oplus^{2^j}_{n+k-2-2^j}, \ldots, \oplus^{2^j}_{n+k-i_1-1}.$$

To compute $\oplus^{i_1+1}_{n+k-i_1-2}$ by Option 2, two steps are needed. The first step is to compute $\oplus^{2^j}_{n+k-(i_1-1)-1}$, which requires $2^{H(2^j)} - 1 = 1$ XOR gates. The second step is to compute $\oplus^{i_1+1}_{n+k-i_1-2}$ from

$$\oplus^{2^j}_{n+k-1-2^j}, \oplus^{2^j}_{n+k-2-2^j}, \ldots, \oplus^{2^j}_{n+k-i_1-1}, \oplus^{2^j}_{n+k-i_1-2}.$$

$2^{H(i_1+1-2^j)} - 1$ XOR gates are needed for that. Therefore, in total, Option 2 requires

$$2^{H(i_1+1-2^j)} - 1 + 1 = 2^{H(i_1+1-2^j)} \text{ XOR gates.}$$

Because $2^j < i_1 + 1 < 2^{j+1}$, $2^{H(i_1+1-2^j)}$ is simplified to $2^{H(i_1+1)-1}$. The number of XOR gates required by Option 1 is $2^{H(i_1+1)} - 1$, which is larger than $2^{H(i_1+1)-1}$ for $2^j < i_1 + 1 < 2^{j+1}$ with arbitrary positive j.

□

Corollary 5. *Option 2 requires* $2^{H(i)-1}$ XOR *gates to compute* $\oplus^i_{n_1}$*, where* j *is a positve integer and* $2^j < i < 2^{j+1}$*. It holds for arbitrary non-negtive integer* n_1*.*

Corollary 5 is extracted from an intermediate result in the proof of Lemma 10.

Having said in Section 4.1.5 synthesis tools are more likely to do a better job than us in optimizing combinational logics, we propose Algorithm 2, which is a variation of Algorithm 1 and relieves most of our burden of extracting and considering common sub-expressions. In other words, Algorithm 2 does not take Option 2 into account. If the composition degree i is an even number, we compute it directly. Otherwise, we treat i as $(i-1)+1$ so that it can be computed by a summation of two composiiton operations of degree $i-1$. The one with a smaller index can be computed either directly or by registering the one with larger index, depending on the comparison of the area costs, ff-cost and xor-cost. ff-cost is still 1 FLIP-FLOP and xor-cost now becomes $2^{H(i-1)} - 1$ XOR gates. Our experience shows that Algorithm 2 almost always returns a better way (in terms of both hardware area and performance) to compute the diagonal nodes than Algorithm 1. Besides, Algorithm 2 is simpler to implement. Therefore, it is chosen to instruct us how to compute the diagonal nodes.

Algorithm 2 Computing the Diagonal Nodes

1: **function** EFF-DIAG-V2(k)
2: $i \leftarrow 0$
3: **while** $i < k$ **do**
4: **if** $i \mod 2 = 0$ **then**
5: print "$\oplus^i_{n+k-1-i} = \oplus^i_{n+k-1-i}$"
6: **else**
7: print "$\oplus^i_{n+k-1-i} = \oplus^{i-1}_{n+k-i} + \oplus^{i-1}_{n+k-i-1}$"
8: **if** ff-cost($\oplus^{i-1}_{n+k-i-1}$) $\leq$ xor-cost($\oplus^{i-1}_{n+k-i-1}$) **then**
9: print "$\oplus^{i-1}_{n+k-1-i}{}^{(1)} = \oplus^{i-1}_{n+k-i}$"
10: **else**
11: print "$\oplus^{i-1}_{n+k-1-i} = \oplus^{i-1}_{n+k-1-i}$"
12: **end if**
13: **end if**
14: **end while**
15: **end function**

Moreover, we present a snippet of code (Algorithm 3) that can be used to extend Algorithm 2 or Algorithm 1 to improve their effectiveness for larger k. It provides another place where replacing combinational logic with FLIP-FLOPs is applicable. $Thres$ (on Line 1) is a constant whose value depends on the hardware library (the technology) used. For CMOS 65nm, $Thres = 3$ works well. It helps to bring OcDeb-*128-24*'s area down to 1987 GE from 2079 GE, with clockspeed increasing from 518 MHz to 571 MHz. The results are from Synopsys DC_Shell logic synthesis with low optimization efforts and a clockspeed target of 250 MHz.

Algorithm 3 A Patch for Efficiently Computing the Diagonal

1: **if** $(i-2) \mod 4 = 0$ and $H(i) \geq Thres$ **then**
2: print "$\oplus^{i}_{n+k-1-i} = \oplus^{i-2}_{n+k+1-i} + \oplus^{i-2}_{n+k-1-i}$"
3: print "$\oplus^{i-2}_{n+k-1-i}{}^{(1)} = \oplus^{i-2}_{n+k-i}$"
4: **end if**

As final remarks for efficiently computing the diagonal nodes, we would like to point out that the simplicity of both algorithms, Algorithm 1 and Algorithm 2 comes from sacrificing the completeness of common sub-expressions considered. In Algorithm 1, we only recognize the common sub-expressions that match the nodes from i-th row, where i can be any powers of 2, and the common sub-expressions that match one of the first two nodes on even rows. In Algorithm 2, we only recognize the common sub-expressions that match one of the first two nodes on even rows. In comparison, we are more considerate about recognizing common sub-expressions when computing the new complexity in Section 3.2. In fact, following the approach in Section 3.2, for each node on the diagonal, we can find the maximum number of XOR gates that can be shared by lower nodes. Though we have a very simplified view of where common sub-expressions exist, Algorithm 1 and Algorithm 2 both work well in decreasing hardware area and increasing the clockspeed.

Please refer to Section 4.2.3 for concrete examples of Algorithm 1, Algorithm 2.

4.1.7 Computing the Inputs to the G

Because different G could have different subset of the top row nodes as inputs, proposing an algorithm to find efficient ways to compute the inputs to G is more complex than doing the same with the nodes of the diagonal. However, this issue can be mitigated in practice. On one hand, for an instance of OcDeb with known G, we have more insights in how to efficiently compute the inputs to G. On the other hand, we usually have the freedom to

choose a prefered k to make sure $H(k)$ is small in order to achieve a smaller area for each input to G. Especially, when k is a power of 2, for aribitrary i, $\not\oplus_i^k = x_i + x_{i+k}$ according to Lemma 1. This extremely simple expression for $\not\oplus_i^k$ disallows our two optimization techniques because for arbitrary $i_1 \neq i_2$, $\not\oplus_{i_1}^k$ and $\not\oplus_{i_2}^k$ do not share more than 1 elements from vector x, and 1 XOR gate is almost always smaller than 1 FLIP-FLOP. [1]

4.2 Modules

We will present **All0/1, G, Diag,** and **SomePat** modules in details. Table 4.1 lists each module's area for OcDeb-*32-32*. We can see that the most expensive module of the mesh computation is Diag, followed by SomePat. More results are available in the dedicated results chapter.

Table 4.1: Implementation area results for modules of OcDeb-*32-32*

		Mesh					Shift Register	G
		All0/1	Diag	SomePat	In-G	Total		
OcDeb-*32-32*	GE	62.5	108.8	87.2	28	295.5	381	22.5
	%	8.0%	14.0%	11.2%	3.6%	38.0%	49.0%	2.9%

In-G: inputs to G G: the original feedback functions that generates the starting span-n sequence

4.2.1 Module All0/1

As stated in Theorem 3, we need to determine whether the nodes on the top row are all 0s and whether they are all 1s. All0/1 module serves this purpose by having two signals as outputs. The left output will be high when the nodes on the top row are all 0 while the right one will be high when the nodes on the top row are all 1s.

We propose Algorithm 4 as an alternative to directly calculating whether the nodes are all 0s and whether they are all 1s with AND gates NOT gates. Algorithm 4 has better area

[1] An exception: for FPGA, FLIP-FLOPs may be free when the combinational logics need more logic blocks than FLIP-FLOPs.

Algorithm 4 Using Finite State Machine to Compute All0 and All1

1: **procedure** ALL0-1-FSM(in) ▷ for OcDeb-k-n
2: $state^{(1)} \leftarrow in$ ▷ in is the leftmost node on the top row
3: **if** $in \neq state$ **then**
4: $cnt^{(1)} \leftarrow 1$
5: **else if** $cnt < n - 1$ **then** ▷ if the input is the same as the previous input
6: $cnt^{(1)} \leftarrow cnt + 1$ ▷ increase the counter by 1
7: **else**
8: $cnt^{(1)} \leftarrow cnt$
9: **end if**
10: **if** $cnt = n - 2$ and $in = 1$ and $state = 1$ **then**
11: $All0 \leftarrow 0$
12: $All1 \leftarrow 1$
13: **else if** $cnt = n - 2$ and $in = 0$ and $state = 0$ **then**
14: $All0 \leftarrow 1$
15: $All1 \leftarrow 0$
16: **else**
17: $All0 \leftarrow 0$
18: $All1 \leftarrow 0$
19: **end if**
20: **end procedure**

and shorter delay. It works best for large n and inferior k, because this algorithm only needs to compute the first node and uses a counter of length $\log(n)$ to count consective 1s and consective 0s. It may not be as good as the direct computation when n is small, k's Hamming weight is small or the number of inputs to G is large. For example, for a parallel OcDeb-*32-32* with a throughput of 2, it is better to simply directly compute All0/1 because 32's Hamming weight is 1, and the number of inputs to G is doubled because of the parallelism. Logic synthesis results obtained by Design Compiler show area decreases from 785 GE to 747 GE with a slight decrease in clockspeed from 961.5 GHz to 934.6 GHz.

In Algorithm 4, when $state = 0(1)$, cnt is the number of consective 0(1)s starting from the second node on the top row towards right.

4.2.2 Module G

Function G is the original feedback function of a shift register that generates the starting span-n sequence. To build efficient OcDeb-k-n with long period, the following attributes of G are desired:

- The computation of G is simple.
- The number of inputs to G is small.
- The span-n sequence generated by G is long, in other words, n is large.

Fewer inputs to G makes the area that is used to compute them smaller. OcDeb-k-n has a period of 2^{n+k}. Consider that our goal is to build a de Bruijn sequence generator with period longer than 2^C, where C is an abitrary positive integer. When we have a larger n, we are able to choose a smaller k, which is the parameter that contributes most to the complexity, $\mathcal{O}(k \log(k) + \log(n))$, and has the most affect on the area.

Table 4.2: Two examples of the original feedback function G

	Feedback Expression	**Source**	**Note**
n=24	$x_0 + x_{24} + WG5_1 + WG5_2$	[30]	It is composed of a summation of two WG5 transformations [16, 17]. It has sub-optimal linear complexity.
n=32	$x_0 + x_2 + x_6 + x_7 + x_{12} +$ $x_{17} + x_{20} + x_{27} + x_{30} +$ $x_3x_9 + x_{12}x_{15} + x_4x_5x_{16} + x_{32}$	[14]	This forms an NLFSR that generates the span-n sequence with the maximum period found so far.

$$WG5_1 = x_1 + x_2 + x_3 + x_4 + x_6 + x_2x_3 + x_2x_4 + x_3x_6$$
$$WG5_2 = x_{10} + x_{10}x_{17} + x_{13}x_{15} + x_{15}x_{21} + x_{10}x_{13}x_{15} + x_{10}x_{13}x_{21} + x_{10}x_{15}x_{21} + x_{10}x_{17}x_{21} + x_{13}x_{15}x_{21} + x_{13}x_{17}x_{21} + x_{15}x_{17}x_{21}$$

Table 4.2 shows two examples of original feedback function G, both used in our implementations. We implemented $WG5_1$ and $WG5_2$ with two arrays of size 2^5, filled with pre-computed values. Each entry corresponds to a possible input. The area of G for $n = 24$ is 27.7 GE and the area of G for $n = 32$ is 22.5 GE. Both results exclude the area used for

computing the inputs. The results were obtained from logic synthesis using CMOS 65nm process. Other possible G candidates can be found in [8, 9, 30, 41].

4.2.3 Module Diag

Refer to Section 4.1.6 for an efficient way to compute the diagonal nodes, Section 3.1.5 for how to increase clock speed by applying retiming technique, which is enabled by Corollary 4. Recall we show in Section 3.2 that the computation of the diagonal nodes is upper bounded by $\mathcal{O}(k \log(k))$ and is the dominating part of the overall computation of $mesh_n^k$. Although both Algorithm 1 and Algorithm 2 are more efficient than the way of computing diagonal nodes we use to derive the complexity in Section 3.2, they are not better asymptotically. In this section, we present a concrete application of Algorithm 1 and Algorithm 2 to OcDeb-*32-32* with CMOS 65nm as the target technology. Recall that for CMOS 65nm, one FLIP-FLOP is around 4 GE while one XOR is around 2 GE. The results of the algorithms are shown in Figure 4.2 and Figure 4.3. Note that in the figures, we do not show the mesh of XOR gates because computing the diagonal nodes according to Algorithm 1 or Algorithm 2 is way more efficient than mesh. We only keep the nodes from the mesh that we are going to compute. The bottom nodes are from the shift register and we only draw the stages that are needed for the computation of diagonal nodes. We tilt the Y-axis so that it is easier to tell which row each node belongs to. Every node, except the nodes directly from the shift register, has a background denoting how it is calculated based on the corresponding algorithm. We always compute the diagonal nodes on odd rows by Option 4, which is summing the diagonal node below it and that node's right neighbour. Figure 4.2 and Figure 4.3 differ in computation of diagonal ndoes on even rows. In Figure 4.2 (Algorithm 1), they are computed based on Option 2 except for the rows whose height is a power of 2, where they are computed by Option 1. In Figure 4.3 (Algorithm 2), the diagonal nodes on all even rows are computed directly, i.e. by Option 1 (recall that Option 2 is removed in Algorithm 2). Note that on the rows whose height is a power of two, we also need to compute internal nodes that are required by the computation of higher diagonal nodes by Option 2. For example, in Figure 4.2, we need to compute the internal diagonal node at $(43, 16)$ because it is needed by the computation of the diagonal node at $(43, 20)$. As for the second leftmost node on each even row, it is calculated based on Option 3 with the exception on rows whose height is a power of 2, where it is computed by Option 1.

Recall that to compute a node, let us say $\oplus_i^j$, Option 1 and Option 2 require $2^{H(j)} - 1$ and $2^{H(j-1)}$ XOR gates respectively, while Option 3 and Option 4 require 1 FLIP-FLOP and 1 XOR gate respectively. For Algorithm 1 in Figure 4.2, 71 XOR gates and 11 FLIP-FLOPs are required in total, while for Algorithm 2 in Figure 4.3, 85 XOR gates and 11 FLIP-FLOPs are

required in total. As discussed in Section 4.1.5, hardware synthesis tools have the ability to apply common sub-expression elimination and most of the time do a better job than us. Option 2 which recognizes a type of common sub-expressions is lacking in Algorithm 2. Despite the theoretical count, which favours the computation shown in Figure 4.2, practical results show that the way to compute diagonal nodes shown in Figure 4.3 will actually lead to a smaller area than the one shown in Figure 4.2, due to much more aggressive common sub-expression elimination by synthesis tools.

Application of Algorithm 1 on OcDeb-*32-32* is shown here.

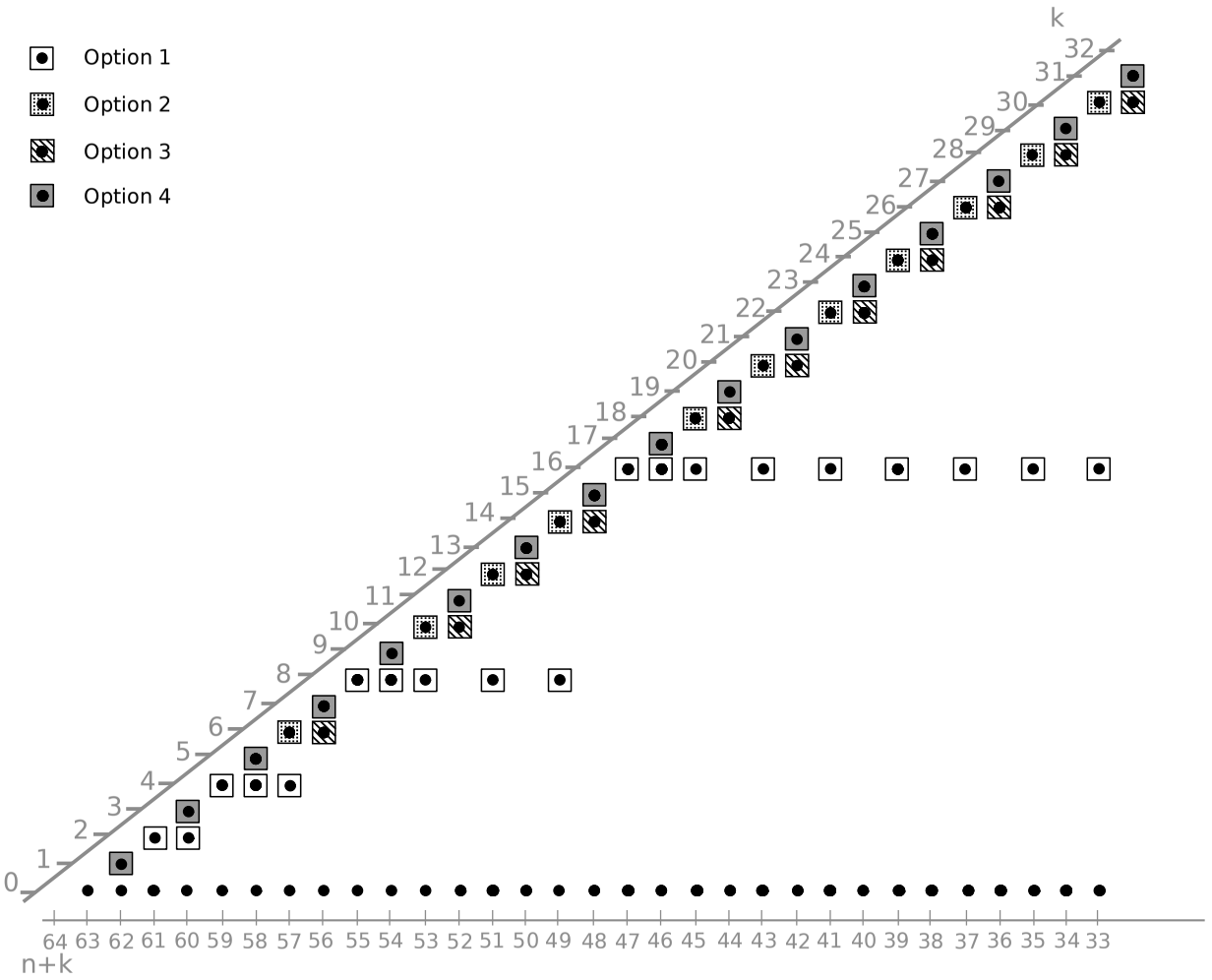


Figure 4.2: Computing the diagonal nodes of OcDeb-*32-32* based on Algorithm 1

Application of Algorithm 2 on OcDeb-*32-32* is shown here.

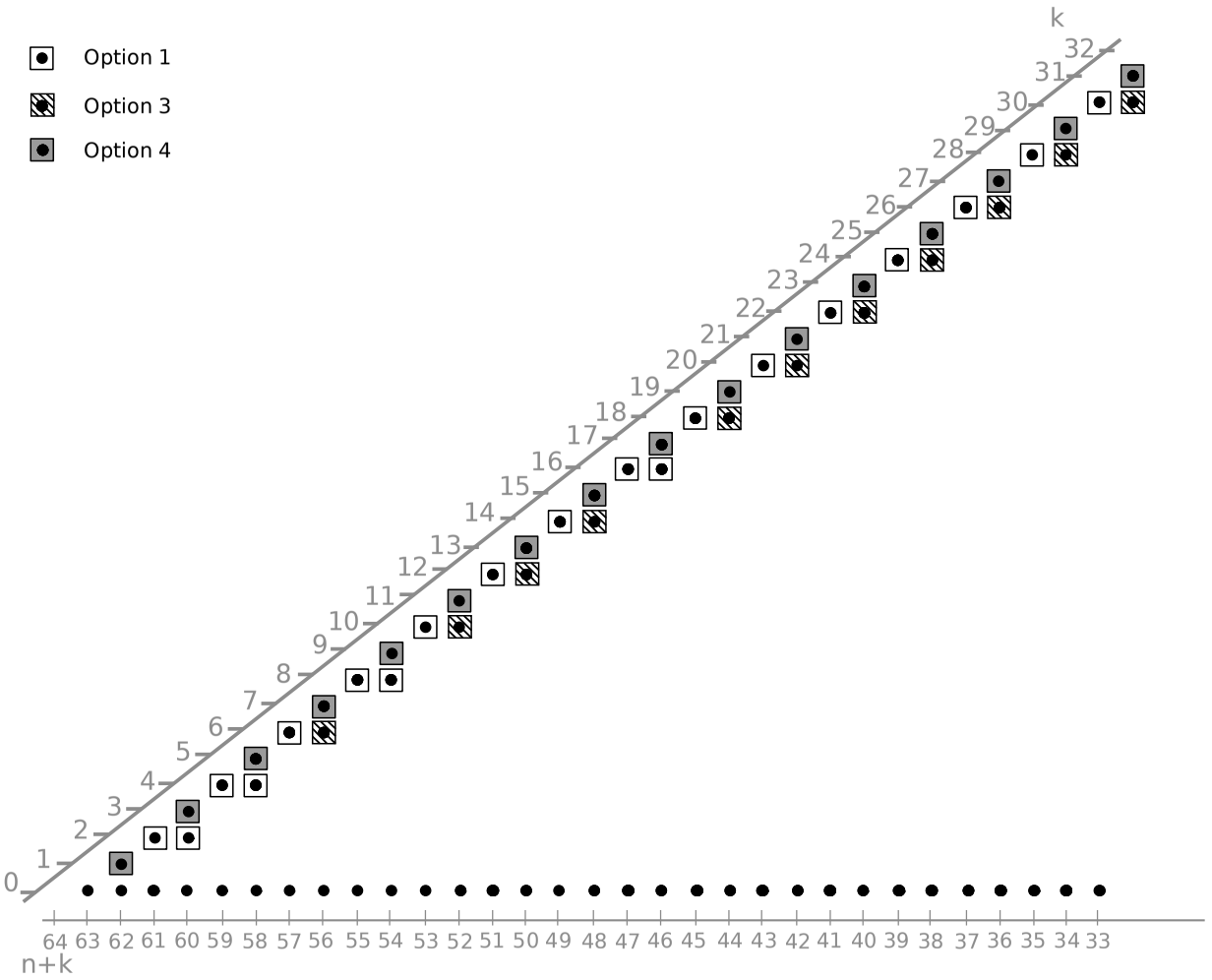


Figure 4.3: Computing the diagonal nodes of OcDeb-*32-32* based on Algorithm 2

4.2.4 Module SomePat

For OcDeb-*k*-*n* where $n+k$ is even, SomePat (Definition 12) is 1 iff the diagonal nodes from bottom to the top row match the regular expression $([01][01])^*110^*$ or $[01]([01][01])^*010^*$. Similary, for OcDeb-*k*-*n* where $n+k$ is odd, SomePat is 1 iff the diagonal nodes from bottom to the top row match the regular expression $([01][01])^*010^*$ or $[01]([01][01])^*110^*$.

For example, for OcDeb-*4*-*4*, we can group possible pattern-matches into four categories: ([01][01]){3}01, ([01][01]){2}11$\underline{0}$, ([01][01]){1}01$\underline{00}$ and 11$\underline{000}$. Consective 0s are underlined. We use *diag*(0),. . . ,*diag*(4) to represent OcDeb-*4*-*3*'s diagonal from the lowest to the highest row, then SomePat becomes:

$$
\begin{aligned}
\mathit{SomePat}_3^4 = &\Big(\mathit{diag}(4) \text{ AND NOT } \mathit{diag}(3)\Big) \text{ OR} \\
&\Big(\underline{\text{NOT } \mathit{diag}(4)} \text{ AND } \big(\mathit{diag}(3) \text{ AND } \mathit{diag}(2)\big)\Big) \text{ OR} \\
&\Big(\underline{\text{NOT } \mathit{diag}(4) \text{ AND NOT } \mathit{diag}(3)} \text{ AND } \big(\mathit{diag}(2) \text{ AND NOT } \mathit{diag}(1)\big)\Big) \text{ OR} \\
&\Big(\underline{\text{NOT } \mathit{diag}(4) \text{ AND NOT } \mathit{diag}(3) \text{ AND NOT } \mathit{diag}(2)} \text{ AND } \big(\mathit{diag}(1) \text{ AND } \mathit{diag}(0)\big)\Big)
\end{aligned}
$$

Recall that to compute the feedback value (Theorem 3), SomePat is in conjuction with *All*0 , which implies the leftmost node on the top row(which is *diag*(4) in the above example) is 0. With this knowledge, we can simplify the above example to:

$$
\begin{aligned}
\mathit{SomePat}_3^4 = &\Big(\mathit{diag}(3) \text{ AND } \mathit{diag}(2)\Big) \text{ OR} \\
&\Big(\underline{\text{NOT } \mathit{diag}(3)} \text{ AND } \big(\mathit{diag}(2) \text{ AND NOT } \mathit{diag}(1)\big)\Big) \text{ OR} \\
&\Big(\underline{\text{NOT } \mathit{diag}(3) \text{ AND NOT } \mathit{diag}(2)} \text{ AND } \big(\mathit{diag}(1) \text{ AND } \mathit{diag}(0)\big)\Big)
\end{aligned}
$$

Algorithm 5 generalizes the above computation of SomePat. It creates a vector of length $k-2$, named "*nand_diag*", to store the computation results of checking for consective 0s. It takes in diagonal nodes from bottom to the second highest row as inputs, which are denoted as *diag*(0), . . . , *diag*(k-1). The vector *nand_diag*is obtained as:

$$
\begin{aligned}
&\mathit{nand\_diag}(0) = \text{NOT } \mathit{diag}(\text{k-1}) \\
&\mathit{nand\_diag}(\text{i}) = \mathit{nand\_diag}(\text{i-1}) \text{ AND NOT } \mathit{diag}(\text{k-i-1}),\ 0 < i \leq k-3
\end{aligned}
$$

$\mathit{SomePat}_k^n$ updates itself in every interation of the loop that starts from Line 7 to Line 15 by considering one category of regular expression matches. When Algorithm 5 exits the loop,

Algorithm 5 Calculating *SomePat* with the Diagonal Nodes as Inputs

```
1: procedure CALC-SOMEPAT-V1(in)                    ▷ for OcDeb-k-n and n+k is even
2:     nand_diag(0) ← NOT diag(k − 1)
3:     for i ← 1, k − 3 do
4:         nand_diag(i) = nand_diag(i − 1) AND NOT diag(k − i − 1)
5:     end for
6:     SomePat_k^n ← 0
7:     for i ← 0, k − 3 do
8:         if k − i − 3  mod 2 = 0 then
9:             SomePat_k^n ← SomePat_k^n OR
10:                (nand_diag(i) AND diag(k − i − 2) AND diag(k − i − 3))
11:        else
12:            SomePat_k^n ← SomePat_k^n OR
13:                (nand_diag(i) AND diag(k − i − 2) AND NOT diag(k − i − 3))
14:        end if
15:    end for
16: end procedure
```

the diagonal nodes have been checked for all possible matches of the regular expression. Though we make an assumption that $n + k$ is even for Algorithm 5, we can easily make it work for odd $n + k$ by exchanging Line 10 and Line 13.

From Algorithm 1 and Algorithm 2, we know that diagonal ndoes with odd degrees are calculated by a summation of two neighbouring nodes right below them. More specifically, for even i, $\not{\phi}_{j-1}^{i+1} = \not{\phi}_{j}^{i} + \not{\phi}_{j-1}^{i}$, or after renaming, $diag(i+1) = diag(i) + diag(i)^{(-1)}$. $diag(i)^{(-1)}$ holds the value of $diag$(i) in the previous clock cycle which is the value of the node on the right of $diag$(i). For simplicity, we refer to the adjacent nodes on the right to the diagonal ndoes as $i\_diag(0)$, $i\_diag(2)$, $i\_diag(4)$, ... These nodes compose an internal diagonal. Now $diag(i+1) = diag(i) + diag(i)^{(-1)}$ can be also written as $diag(i+1) = diag(i) + i\_diag(i)$. Note that this was done for even i only.

Inspired by this way of computing the diagonal, we present Algorithm 6 to calculate SomePat. In comparison with Algorithm 5, it checks two categories of possible pattern-matches with a single slightly more complicated expression. For example, in Algorithm 5, checking whether the diagonal nodes match ([01][01]){2}11$\underline{0}$ or ([01][01]){1}01$\underline{00}$ is done by evaluating

$$\Big(\underline{\text{NOT } diag(4)} \text{ AND } \big(diag(3) \text{ AND } diag(2)\big)\Big) \text{ OR}$$
$$\Big(\underline{\text{NOT } diag(4) \text{ AND NOT } diag(3)} \text{ AND } \big(diag(2) \text{ AND NOT } diag(1)\big)\Big),$$

while in Algorithm 6, it is achieved by evaluating a simpler expression:

$$\underline{\text{NOT } diag(4)} \text{ AND } diag(2) \text{ AND } \Big(\text{NOT } i\_diag(2) \text{ OR NOT}\big(diag(0) \text{ XOR } i\_diag(0)\big)\Big)$$

Algorithm 6 Calculating *SomePat* with the Diagonal Nodes as Inputs

```
1: procedure CALC-SOMEPAT-V2(in)          ▷ for OcDeb-k-n and n and k are even
2:     nand_diag(0) ← NOT diag(k − 2) AND NOT i_diag(k − 2)
3:     for i ← 2; i ← i + 2; k − 4 do
4:         nand_diag(i) ← nand_diag(i − 2) AND NOT diag(k − 2 − i) AND NOT i_diag((k −
   2 − i)
5:     end for
6:     SomePat_k^n ← diag(k − 2) AND NOT i_diag(k − 2)
7:     for i ← 4; i ← i + 2; k − 4 do
8:         tmp ← nand_diag(k-i) AND diag(i − 2) AND
9:                 (NOT i_diag(i − 2) OR NOT (diag(i − 4) XOR i_diag(i − 4)))
10:        SomePat_k^n ← SomePat_k^n OR tmp
11:    end for
12:    SomePat_k^n ← SomePat_k^n OR (nand_diag(k − 4) AND diag(0) AND NOT i_diag(0))
13: end procedure
```

Keeping $diag(2) + i\_diag(2) = diag(3)$ and $diag(0) + i\_diag(0) = diag(1)$ in mind, it is not hard to prove that the two expressions are equivalent. Algorithm 6 is essentially simplified Algorithm 5. In the algorithm, we assume n and k are both even. But it is easy to modify it for other parity combinations of n and k.

Assume n and k are both even. Algorithm 5 uses up $k-3+2(k-2) = 3k-7$ AND gates, $k-2$ OR gates and k XOR gates. Among them, $\frac{k-2}{2}$ XOR gates are used to compute diagonal nodes with odd degrees which are elements from the set $\{\, diag(2i+1) \mid 0 \leq i \leq \frac{k-2}{2} \,\}$. On the contrary, Algorithm 6 uses up $1 + 2 \times \frac{k-4}{2} + 1 + 2 \times \frac{k-6}{2} + 2 = 2k - 6$ AND gates, $2 \times \frac{k-6}{2} + 1 = k - 5$ OR gates and $\frac{k-6}{2}$ XOR gates. Table 4.3 summerizes the above results. As seen in the table, the number of bit operations is linear in k. So SomePat is not the dominating part of the computation of $mesh_n^k$, which has been shown to have complexity $\mathcal{O}(k \log(k) + \log(n))$ in Section 3.2.

Table 4.3: Gate Count of SomePat module implementations based on Algorithm 5 and Algorithm 6 respectively

	and	**or**	**xor**
Algorithm 5	$3k-7$	$k-2$	$\frac{k-2}{2}$
Algorithm 6	$2k-6$	$k-5$	$\frac{k-6}{2}$

4.3 Summary

This section described hardware implementations of OcDeb. Firstly, we introduced two tricks, namely the common sub-expression elimination and replacing combinational logic with sequential logic. We realized that synthesis tools have the ability to apply common sub-expression elimination to some extent but are not capable to use the other trick. We also found out that most of the times, doing common sub-expression elimination ourselves and then feeding the synthesis tool with processed formula does not work better than simply providing the synthesis tool with the raw/original form of formula. Therefore, we only apply the necessary sub-expression elimination on our end and focus on finding the places where replacing combinational logic with sequential logic is beneficial. It is a complex task which requires the knowledge of all locations of the common sub-expressions to make the best decision on where to apply the other trick, that is replacing combinational logic with sequential logic. We presented our solutions to a refined problem, which only involves the diagonal nodes. Algorithm 1 and Algorithm 2 are not optimal solutions but work very well in practice. We also presented implementations details for other modules including $All0/1$, G and $SomePat$. In the next chapter, we will present the implementation results.

Chapter 5

Hardware Implementation Results

To demonstrate the effectiveness of our optimizations, we compare OcDeb with three other implementations of composited construction de Bruijn sequence generators (already introduced in Section 2.3.3): the direct implementation of the mesh, the recursive implementation of the mesh(Figure 2.7) and a serialized implementation that computes one row of the mesh in each clock cycle [29]. We perform a detailed analysis of n and k's impact on area and performance for OcDeb-k-n with periods up to 2^{32} (Section 5.2). We compare the asymptotic complexity of OcDeb against other algorithms to generate de Bruijn sequences (Section 5.3), and compare OcDeb instances to other CSPRNGs (Section 5.4). Results of parallel implementations are presented in Section 5.5. Last but not least, in Section 5.6 we highlight several extra carefully optimized instances of OcDeb that fulfill various demanding requirements/needs.

All results are obtained from logic synthesis with either low or ultra high optimization effort level with clock gating enabled for an STMicroelectronics 65 nm cell library using nominal delay values. We use Synopsys Design Compiler for logic synthesis. Because our main goal is to evaluate effects of optimizations and correlate implementation results with complexity results, not to benchmark the actual implementation against competing PRNGs, we consider logic synthesis sufficient for us and did not do place and route.

5.1 Effectiveness of Optimizations in Hardware

We conduct comparison between OcDeb with three other implementations in area and performance accross a variety of k ranging up to 512. Recall that the sequence generated by OcDeb-k-n is 2^{k+n} has period 2^{k+n}. Results are in Figure 5.1a and Figure 5.1b.

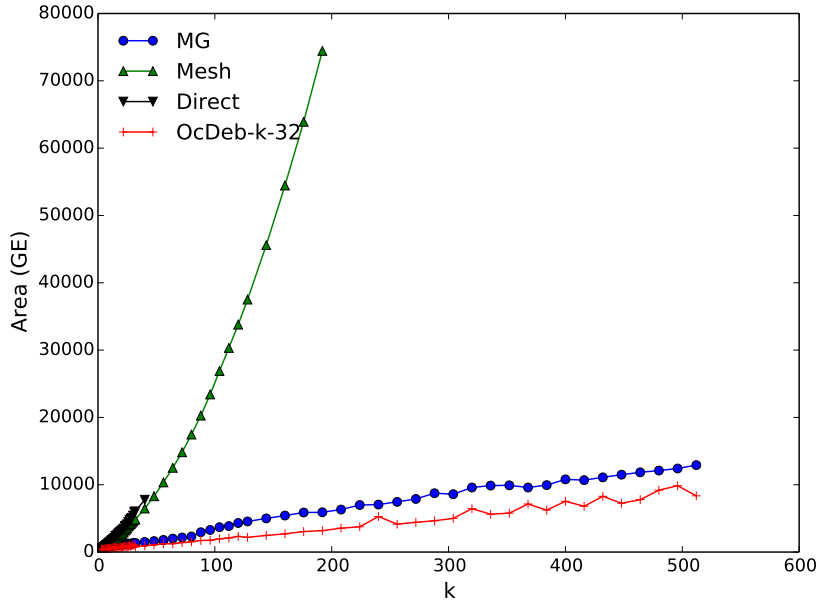
They are obatined from logic synthesis by Synopsys Design Compiler with the optimization effor level set to low. Logic synthesis on the direct implementations runs out of memory when $k \geq 40$ and we kill logic synthesis on the mesh based implementations when $k \geq 208$ because they do not terminate within 1 hour. The sampled ks are $1, 2, 3, \ldots, 30, 31, 32, 40, 48, 56, \ldots, 120, 128, 144, 160, \ldots, 496, 512$. From Figure 5.1a, we observe that our implementation of OcDeb is dramatically smaller than the direct and mesh-based implementation, and that it scales much better with k, which aligns with our complexity analysis. Mandal and Gong (MG) implementations also scale well with k and have area that is usually around 1.5X OcDeb's area. However, recall that MG's throughput is $\frac{1}{k}$, implying very low performance, as show in Figure 5.1b

Refer to Section 5.2 for detailed interpretation of OcDeb's figure. Table 5.1 provides more details on $(k = 32, n = 32)$. Results of each module's area are not available from logical synthesis at ultra high optimization level because at this optimization level, modules are unfold for better optimization.

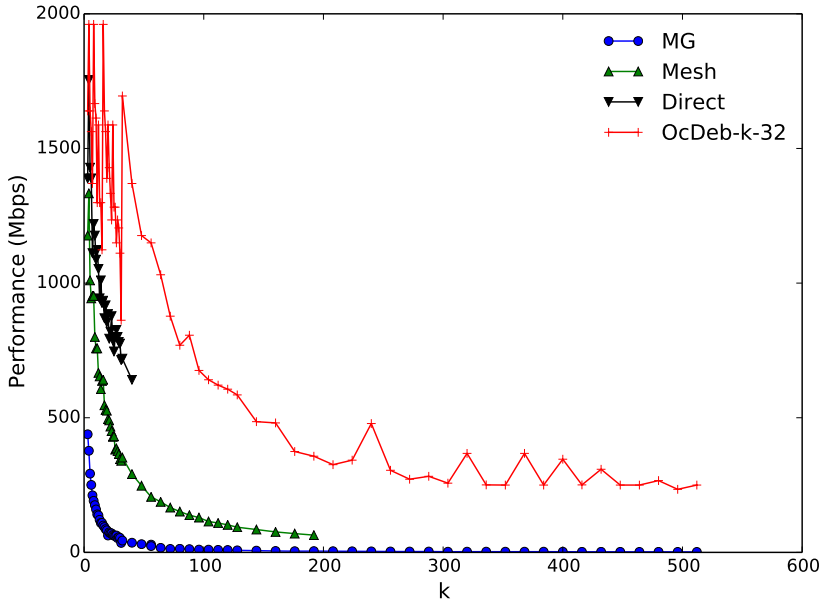
Table 5.1: Detailed comparison of OcDeb-*32-32* with baseline implementations in area and performance

	Direct		Mesh-based		MG		OcDeb-*32-32*	
	simple	ultra	simple	ultra	simple	ultra	simple	ultra
% of area								
SR	6.4	–	7.8	–	28.3	–	49.0	–
G	0.4	–	0.5	–	1.7	–	2.9	–
mesh	92.4	–	90.5	–	64.3	–	38.0	–
Area (GE)	6049	5077	4876	4255	1347	1121	777	656
ClkSpd (GHz)	0.72	0.64	0.46	0.95	1.45	1.82	1.69	1.67
Tput	1	1	1	1	$\frac{1}{32}$	$\frac{1}{32}$	1	1
Optimality	0.12	0.13	0.09	0.22	0.03	0.05	2.17	2.54
Rel Optimality	1.0	1.0	0.75	1.7	0.25	0.38	18.1	19.5

SR: shift-register G: original feedback Tput: throughput Perf = ClkSpd×Tput
simple: logic synthesis with low optimization effort level
ultra: logic synthesis with ultra optimization effort level plus clock gating
Optimality: Perf/Area Rel optimality: relative optimality



(a) Comparison in area



(b) Comparison in performance

Figure 5.1: Comparison of OcDeb with baseline implementations in (a) area and (b) performance across a series of k

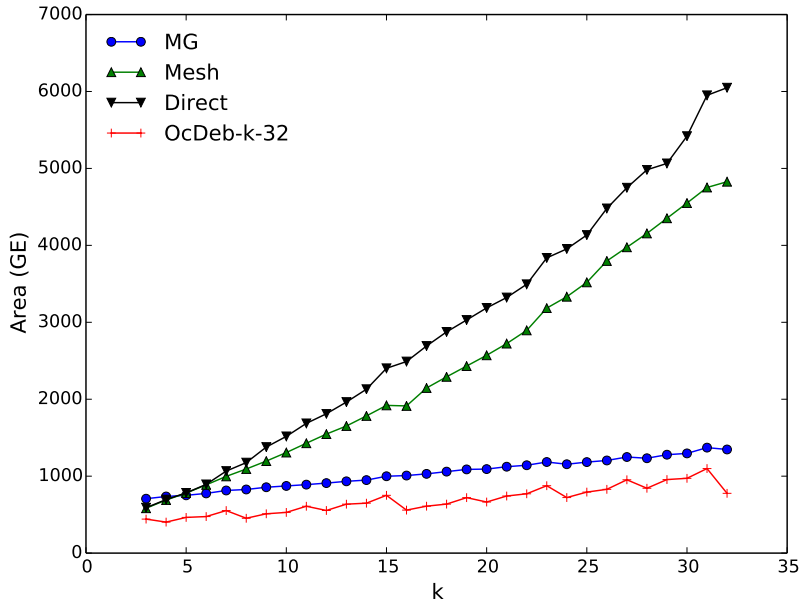
5.2 The Impact of Parameters k and n On OcDeb's Area and Performance

We pay extra attention to $3 \leq k \leq 32$ in Figure 5.1a and Figure 5.1b as it will become clear that the pattern of figures repeats between consective powers of 2. Figure 5.2a and Figure 5.2b enlarge the details of OcDeb figures with k ranging from 3 to 32. We can see that OcDeb's area and performance are inversely correlated in the sense that when the area becomes larger the performance gets worse. For OcDeb, the fluctuations seen in Figure 5.2a and Figure 5.2b are results of the variations of the cost to compute inputs to G after k-degree composition. The function G of OcDeb-k-*32* has 14 inputs. After k-degree composition, each input to G will become a term that has $2^{H(k)} - 1$ XOR gates. When k is between 2^i and $2^{i+1} - 1$ inclusive, where i is an integer, $2^{H(k)}$ is the largest at $k = 2^{i+1} - 1$ and smallest at $k = 2^i$. This explains the local peaks in Figure 5.2a at $k = 3, 7, 15, 31$, in Figure 5.2b at $k = 4, 8, 16, 32$. When k is an even number, $2^{H(k+1)} = 2^{H(k)+1}$, which explains some rises at odd k and falls at even k. One of the exceptions is at $k = 22$, where the hardware cost does not decrease comparing with $k = 21$. This happens because the area saved by computing G's inputs is not as much as the overhead area incurred by increasing k by 1.

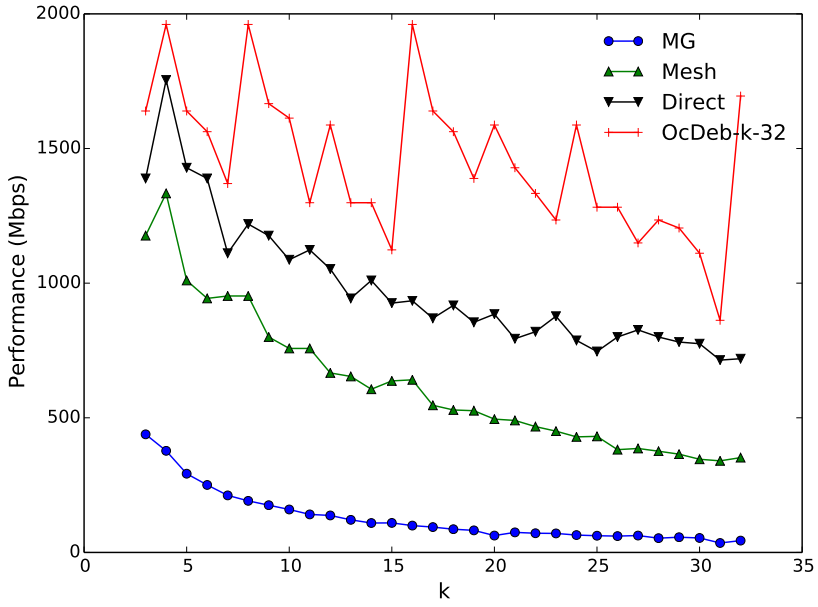
Now, we extend the discussion to the parameter n. In Figure 5.2, we show performance and area results of two categories of instances:

- OcDeb-k_1-*18* with k_1 ranging from 9 to 35 and its G has 5 inputs.
- OcDeb-k_2-*24* with k_2 ranging from 3 to 29 and its G has 10 inputs.

We connect two instances from different categories with concrete lines when they have the same period and label the line with the size of the internal state. The higher an instance is in Figure 5.2, the better its performance, while the closer to the left, the smaller the area. An instance has a better optimality, measured by Performance/Area, proportional to the angle between x-axis and the imaginary line connecting the instance with the origin. We observe that for most of the cases, OcDeb-k_1-*24* is posited higher and closer to the left than the OcDeb-k_2-*18* with the same period. Exceptions appear at $k_1 + 18 = k_2 + 24 = 34$, $k_1 + 18 = k_2 + 24 = 51$ and so on, where the corresponding k_2 are 10, 27 and the corresponding k_1 are 16, 33. Either of these k_1 has a smaller Hamming weight than the respective k_2.



(a) Parameter k's impact on OcDeb's area



(b) Parameter k's impact on OcDeb's performance

Figure 5.2: Parameter k's impact on OcDeb's (a) area and (b) performance

In conclusion, parameter k dominates the area and peformance of OcDeb. Almost at all times, a larger n is prefered as it either leads to a OcDeb with longer period with the same k or a smaller k while keeping the same period. Also we prefer parameter k with small Hamming weight. However, we would like to mention that NLFSRs with long periods are hard to find. Besides, optimal or close to optimal linear complexity of the starting span-n sequence is required to construct a more secure OcDeb [29].

5.3 Comparisons of the Complexity of de Bruijn Sequences Generators

Table 5.2 summarizes the storage and time complexity of different methods to construct de Bruijn sequences, showing how we improve upon the previously best results of Mandal and Gong. It is almost same as Table 5.2 with one more table row for OcDeb.

Table 5.2: Comparison of OcDeb-k-n with other techniques in storage and computation complexity

	Storage	**Complexity**
Fredricksen [10]	$3(n+k)$	$\mathcal{O}((n+k)^2)$ †
Jansen *et al.* [21]	$3(n+k)$	$\mathcal{O}((n+k)^2)$ †
Annexstein [2]	—	$\mathcal{O}(2^{n+k})$
Chang *et al.* [6]	—	$3k(n+k)$
Mandal,Gong [29]	$n+k-1$	$\mathcal{O}(k^2+nk)$
$*$ this thesis	$n+k-1+\log(n)$	$\mathcal{O}(k\log(k)+\log(n))$

† The complexity is an estimate based upon the cited paper.

Results are given in terms of n and k where the period of the de Bruijn sequence is 2^{n+k} to enable comparison between the composited construction and other techniques.

Table 5.3: Comparison of OcDeb with PRNGs used in RFID tags in terms of area and peformance

	State (bits)	**Area** (GE)
LAMED [39]	64	1585
OcDeb-*40-24*	64	869
Warbler [45]	65	464
Melia-Segui [34]	16	761
OcDeb-*32-32*	64	656

5.4 Comparisons with Other Lightweight CSPRNGs

An active area of research today is developing PRNGs for lightweight cryptography: characterized by applications such as RFID tags, wireless sensor network devices, and small embedded systems with limited computational resources. The goal is to reduce the area of PRNGs while maintaining reasonable levels of performance and security. Examples of lightweight PRNGs include LAMED [39], Warbler [31], and Melia-Segui [34]. Table 5.3 list their area and one instance of OcDeb that has comparable security level, which is measured by the number of state bits. From the table, we can tell that OcDeb-*32-32* is smaller than all of them except Warbler. However, Warbler's maximum clockspeed is 1.43 GHz and its throughput is 1/5. Hence, the performance is 0.286 Gbps, which is much slower compared with 1.67 Gbps of OcDeb-*32-32*.

5.5 Results of Parallel Implementations

Table 5.4 shows the area and performance data for OcDeb-*32-32* with parallelization of 1, 2, and 10. We run Design Compiler with ultra high optimization effort and clock gating turned on. For each parallelism, we report two extremes of the tradeoff between area and clockspeed. They are acquired by manipulating the clock period target for logic synthesis.

To increase the throughput, we compute future feedback values which requires moving NLFSR's tap positions to the left. As parallelism increases, the clock speed decreases,

Table 5.4: Implementation results for parallel OcDeb-*32-32*

Parallelism	**Area** GE	**ClkSpd** GHz	**Performance** Gbps	**Optimality** Perf/Area
1	724	2.50	2.50	3.45
1	656	1.67	1.67	2.54
2	798	2.00	4.00	5.01
2	697	1.15	2.30	3.30
10	1229	0.83	8.30	6.75
10	990	0.52	5.20	5.25

because the first stage of the shift register is in the feedback logic. Thus, the delay through the combinational logic inevitably increases as the parallelism increases.

5.6 Highlighted Instances

In this section, we provide results for four instances of OcDeb. The results were obtained by setting Design Compiler's optimization effort level to ultra high and enabling clock gating.

Table 5.5: Highlighted instances: OcDeb-32-32-xx, OcDeb-512-32-xx

Instance	**Sequence Period**	**Area** GE	**Clockspeed** GHz	**Performance** Gbps
OcDeb-*32-32*-la	2^{64}	656	1.67	1.67
OcDeb-*32-32*-hp	2^{64}	1229	0.83	8.30
OcDeb-*512-32*-hpnp	2^{544}	7949	1.43	1.43

OcDeb-*32-32*-la. Low area. It uses Algorithm 2 to compute the diagonal nodes, a state machine to compute All0/1 (Algorithm 4), and Algorithm 6 to compute *SomePat*. It is not paralleled.

OcDeb-*32-32*-hp. High performance. It has a parallelism of 10. The Hamming Weight of integer 32 is just 1. Parallelizing Algorithm 4 to a degree of 10 is fairly complex and leads to a slow and large circuite. So we simply compute All0/1 explicitly.

OcDeb-*512-32*-hpnp [1]**.** We use Algorithm 2 and Algorithm 3 to save area. We apply retiming aggressively to improve clockspeed and depend on Theorem 4 to guarantee its correctness. It is not paralleled but the result we show is obtained by setting constraints for the clock period in a way to achieve high clockspeed at the cost of area.

[1]It is a different implementation from the one that can be found in Figure 5.1a and Figure 5.1b. For OcDeb-*512-32*-hpnp, we take extra measures to ensure the performance is comparable to OcDeb instances with smaller k. The measures include inserting register in the comptuation of $SomePat$, puting register after the diagonal nodes whose computation is expensive.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Psuedorandom number generators are critical building blocks but also often the overlooked ones in cryptographic systems. de Bruijn sequences have good randomness properties making them attractive for pseudorandom number generators. Existing techniques to generate de Bruijn sequences with realistic periods (e.g., 2^{64}) are too expensive and mostly implemented in software. The most effective technique so far would be Mandal and Gong's composited construction. However, our hardware implementation results show that it still does not have competitive area and performance with respect to other PRNGs, such as LAMED, Warbler, Melia-Segui. So optimizations are needed.

We approached the optimization problem by drawing the architecture of composited de Bruijn sequence generators directly based on its feedback expression. This revealed a regular structure consisting of XOR gates, and we called this structure a mesh. Each row of the mesh has one XOR gate fewer than the row below. Nodes on each row (except the top row) are inputs to a function X_i^j (function X checks alternating 0 and 1), where j and i are determined by the coordinates of the leftmost node on the row. Nodes on the top row are inputs to the function $All0$, which checks whether all of the top row nodes are 0. The summation of all functions associated with each row (i.e. $mesh_n^k$) is the most expensive part of the composited construction, and because of that it is the target of our optimizations. Later, we found some interesting relations among the functions, for example, there is at most one row that can satisfy X. Another important realization is that when all top row nodes are 0, we only need the nodes on a segment of the diagonal (that is composed of the leftmost nodes on each row) to determine whether a row satisfies X or not. With this, we

eliminate the need to compute the internal nodes of the mesh. In the end, we concluded that we can compute the summation of the functions associated with rows of the mesh (i.e. $mesh_n^k$) using only the diagonal nodes and the top row nodes. We called this diagonal optimization. In the diagonal optimization, we created a function called *SomePat* to check certain patterns on the diagonal nodes. A presence of those patterns means $mesh_n^k = 0$. For the top row, we only check whether the nodes are all 0 or all 1, and for that we can use a counter that counts the consective 0s and consective 1s, hence we only need the leftmost node on the top row instead of all nodes on the top row. The diagonal optimization reduces the number of bit operations and also improves it asymptotically, from $\mathcal{O}(k^2 + nk)$ to $\mathcal{O}(k \log(k) + \log(n))$. Because the savings in bit operations are dramatic, we called optimized de Bruijn sequence generators "OcDeb" to distinguish it from the original ones. The other important optimization of OcDeb we presented states that for any function of the base row nodes, we can find an equivalent function on any $n+k-1$ independent nodes from the mesh. A concrete application of this optimization is its corollary, which states that *SomePat* can be applied to any interior diagonal instead of the diagonal. *SomePat* on an interior diagonal is either the same as it is on the diagonal or can be related to it in a simple manner. This is very useful in applying retiming and efficient parallelization. Retiming is a hardware optimization technique involving inserting FLIP-FLOP into the longest combinational path to improve performance. Both retiming and parallelization requires computing future values which implies moving the inputs of the functions to the left direction of the mesh. Because of the corollary we just mentioned, we can offset it by firstly moving the inputs of *SomePat* to the right. Our optimizations are formulated in lemmas and theorems with proofs.

In the hardware implementation, we found that it is inefficient to compute the diagonal nodes with the XOR gates in the mesh. Instead, it is more efficient to compute each of them individually and directly based on the definition. Later we presented two algorithms that improve this approach by utilizing two tricks, the common sub-expression elimination and replacing combinational logic with sequential logic (or trading space for computation). The two tricks are useful throughout the implementation, but they sometimes compete with each other.

OcDeb family of pseudorandom number generators shows dramatic improvement over other composited construction implementations that we considered. An instance of OcDeb-*32-32* shows 72× improvement in optimality (performance/area) over Mandal and Gong's implementation that focuses on low area. Besides, we show our lightweight instances have very competitive area results compared with other lightweight pseudorandom number genarators. For example, an instance of OcDeb-*32-32* has area of 656 GE and performance of 1.67 Gbps, which makes it smaller than LAMED and Melia-Segui all of which are de-

signed for lightweight applications. Moreover, OcDeb offers a wide specture in security (sequence period), performance and area and our optimizations ensure we do not have pay too much overhead in others while pursuing one or two extremes among security, performance and area. Our parallel instance of OcDeb-*32-32* can offer 8.30 Gbps performance at the cost of area 1229 GE. Our instance of a period of 2^{544} has an area of 7949 GE and performance of 1.43 Gbps.

6.2 Future Work

One way to widen OcDeb's application areas in cryptography is by adding a filter function to the output. This would allow us to use OcDeb without exposing its internal state. In the thesis, we adopt an approach that only involves increasing the parameter k in order to get an instance of OcDeb with long period, alternatively, we can increase n or n and k at the same time. It means we use a small instance of OcDeb as the starting feedback shift register. It will be interesting to see whether this approach will yield better results. Besides, analysis on the relation between clockspeed and the degree of parallelism and especially the realation between performance ($clockspeed \times throughput$) and the degree of parallelism will be very helpful in further pushing OcDeb on the performance side. Last but not the least, efficient software implementation of OcDeb is valuable. Our optimizations are also applicable in software implementations and new software oriented optimizations are expected.

References

[1] *Factoring RSA keys from certified smart cards: Coppersmith in the wild*. Springer, 2013.

[2] F.S. Annexstein. Generating de Bruijn sequences: An efficient implementation. *IEEE Transactions on Computers*, 46(2):198–200, February 1997.

[3] James Ball, Julian Borger, and Glenn Greenwald. Revealed: how us and uk spy agencies defeat internet privacy and security. *The Guardian*, 6, 2013.

[4] de NG Bruijn. A combinatorial problem. *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen. Series A*, 49(7):758, 1946.

[5] Segher Bushing, H. M. Cantero, and S. Peter. PS3 epic fail, December 2010. *Chaos Communication Congress*.

[6] T. Chang, B. Park, Y. H. Kim, and I. Song. An efficient implementation of the D-homomorphism for generation of de Bruijn sequences. 45(4):1280–1283, May 1999.

[7] Lidong Chen and Guang Gong. *Communication System Security*. 2012.

[8] Elena Dubrova. A list of maximum period nlfsrs. *IACR Cryptology ePrint Archive*, 2012:166, 2012.

[9] Elena Dubrova. A list of maximum period NLFSRs. Technical Report 166, Cryptology ePrint Archive, 2012.

[10] H. Fredricksen. A class of nonlinear de Bruijn cycles. 19(2):192–199, September 1975.

[11] Harold Fredricksen and Irving Kessler. Lexicographic compositions and debruijn sequences. *Journal of Combinatorial Theory, Series A*, 22(1):17–30, 1977.

[12] Harold Fredricksen and James Maiorana. Necklaces of beads in k colors and k-ary de bruijn sequences. *Discrete Mathematics*, 23(3):207–210, 1978.

[13] R. A. Games. A generalized recursive construction for de Bruijn sequences. 29(6):843–850, September 1983.

[14] B.M. Gammel, R. Gottfert, and O. Kniffler. An NLFSR-based stream cipher. pages 1–4, 2006.

[15] Solomon W. Golomb. On the classification of balanced binary sequences of period 2^{n-1}. 26(6):730–732, 1980.

[16] Solomon W. Golomb and Guang Gong. *Signal Design for Good Correlation*. 2005.

[17] Guang Gong and Youssef A.M. Cryptographic properties of the welch-gong transformation sequence generators. 48(11):2837–2846, November 2002.

[18] Irwin John Good. Normal recurring decimals. *Journal of the London Mathematical Society*, 1(3):167–169, 1946.

[19] E. R. Hauge and T. Helleseth. De Bruijn sequences, irreducible codes and cyclotomy. *Discrete Mathematics*, 159(1–3):143–154, November 1996.

[20] GS1 EPCglobal Inc. EPC radio-frequency identity protocols, generation-2 uhf rfid, ver2.0.0.

[21] C.J.A. Jansen, W.G. Franx, and D.E. Boekee. An efficient algorithm for the generation of de Bruijn cycles. 37(5):1475–1478, September 1991.

[22] Jeff Larson, Nicole Perlroth, and Scott Shane. Revealed: The nsas secret campaign to crack, undermine internet security. *Pro-Publica, September*, 2013.

[23] Pierre L'Ecuyer and Richard Simard. Testu01: Ac library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)*, 33(4):22, 2007.

[24] A. Lempel. On a homomorphism of the de Bruijn graph and its applications to the design of feedback shift registers. *IEEE Transactions on Computers*, C-19(12):1204–1209, 1970.

[25] C. Li, X. Zeng, C. Li, and T. Helleseth. A class of de Bruijn sequences. 60(12):7955–7969, 2014.

[26] Ming Li and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer Science & Business Media, 2013.

[27] Jacobus Hendricus Lint. *Combinatorial Theory Seminar, Eindhoven University of Technology*. Springer-Verlag, 1974.

[28] K. Mandal and G. Gong. Cryptographically strong de Bruijn sequences with large periods. volume 7707, pages 104–118, 2012.

[29] Kalikinkar Mandal and Guang Gong. Cryptographic D-morphic analysis and fast implementations of composited de Bruijn sequences. Technical Report 27, University of Waterloo, CACR, 2012.

[30] Kalikinkar Mandal and Guang Gong. Probabilistic generation of good span-n sequences from nonlinear feedback shift registers. Technical Report 06, University of Waterloo, CACR, 2012.

[31] Kalikinkar Mandal and Guang Gong. Warbler: A lightweight pseudorandom number generator for EPC C1 Gen2 passive RFID tags. *Int'l Jour. RFID Security and Cryptography*, 2(1-4):82–91, March 2013.

[32] George Marsaglia. A current view of random number generators. In *Computer Science and Statistics, Sixteenth Symposium on the Interface. Elsevier Science Publishers, North-Holland, Amsterdam*, pages 3–10, 1985.

[33] George Marsaglia, Wai Wan Tsang, et al. Some difficult-to-pass tests of randomness. *Journal of Statistical Software*, 7(3):1–9, 2002.

[34] Joan Melia-Segui, Joaquin Garcia-Alfaro, and Jordi Herrera-Joancomarti. Analysis and improvement of a pseudorandom number generator for EPC Gen2 tags. In *Financial Cryptography and Data Security*, pages 34–46. Springer, 2010.

[35] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[36] Frederic J Mowle. Relations between p n cycles and stable feedback shift registers. *Electronic Computers, IEEE Transactions on*, (3):375–378, 1966.

[37] Frederic J Mowle. An algorithm for generating stable feedback shift registers of order n. *Journal of the ACM (JACM)*, 14(3):529–542, 1967.

[38] J. Mykkeltveit, Siu M-K., and Tong P. On the cycle structure of some nonlinear shift register sequences. 43:202–215, 1979.

[39] Pedro Peris-Lopez, Julio Cesar Hernandez-Castro, Juan M Estevez-Tapiador, and Arturo Ribagorda. LAMED — a PRNG for EPC Class-1 Generation-2 RFID specification. *Computer Standards and Interfaces*, 31(1):88–97, 2009.

[40] Nicole Perlroth, Jeff Larson, and Scott Shane. Nsa able to foil basic safeguards of privacy on web. *The New York Times*, 5, 2013.

[41] Tomasz Rachwalik, Janusz Szmidt, Robert Wicik, and Janusz Zabłocki. Generation of nonlinear feedback shift registers with special-purpose hardware. In *Communications and Information Systems Conference (MCC), 2012 Military*, pages 1–4. IEEE, 2012.

[42] Anthony Ralston. A new memoryless algorithm for de bruijn sequences. *Journal of Algorithms*, 2(1):50–62, 1981.

[43] Ronald L Rivest, Adi Shamir, and Len Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[44] A. Rukhin, J. Soto, J. Nechvatal, E. Barker, S. Leigh, M. Levenson, D. Banks, A. Heckert, J. Dray, S. Vo, M. Smid, M. Vangel, A. Heckert, and L.E. Iii. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical Report Special Publication 800-22 Rev. 1a, April 2010.

[45] Gangqiang Yang, Mark D Aagaard, and Guang Gong. Efficient hardware implementations of the warbler pseudorandom number generator. 2015.