# Hardware Implementations of the Lightweight Welch-Gong Stream Cipher Family using Polynomial Bases

by

Marat Sattarov

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, 2019

## Author's Declaration

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**

In this thesis we develop a parametrized generic hardware implementation for the Welch-Gong (WG) stream cipher family for low power and low cost applications. WG stream ciphers operate over finite fields $GF_{2^m}$, and are comprised of Linear Feedback Shift Register (LFSR) and non-linear WG transformation as filtering function. These stream ciphers provide mathematically proven keystream properties. We begin with design of individual components that perform cryptographic functions. Then we construct WG transformation using these components and perform analysis of dependency between design parameters and circuit area pre place-and-route for ASIC and two FPGAs. We also explored a second implementation approach that uses constant arrays or lookup tables generated with GAP by Zidaric. Finally, instances of the complete cipher of different sizes from WG-5 to WG-16 that output from 1 to 32 bits / cycle are shown, and their performance and area is analyzed for 65 nm CMOS technology post place-and-route.

**Acknowledgements**

I express my profound gratitude to Professor Mark Aagaard for being an incredible supervisor. His experience, attention to others and excellent sense of humour made my time in the university a pure joy. It was a true honour to be his student. I would also like to personally thank Nusa Zidaric for her great work on constant array implementations in GAP and many hours of discussions and explanations. My appreciation extends to Professor Guang Gong, Dr. Kalikinkar Mandal and other members of the ComSec group.

## Dedication

This thesis is dedicated to my mother, Venera Maylenova. Her endless love, wisdom and support continues to give me strength and motivation, and helps to make the right decisions in life.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Communication security is an increasingly important area as our lives become more and more dependant on computers and technology. Smart devices for all kinds of applications emerge around the world under the *Internet-of-Things* (IoT) paradigm, and *Radio Frequency Identification* (RFID) tags with various functionality are used in daily life. One of the aspects of communication security is confidentiality, commonly achieved by data encryption using a cryptographic tool called a cipher.

Ciphers can be categorized by how the keys are used (asymmetric and symmetric key algorithms) and by how data encryption is performed (block ciphers and stream ciphers). Asymmetric key algorithms, also known as public key algorithms, use different keys to encrypt and decrypt the data, while symmetric key algorithms use the same key for both encryption and decryption. Block ciphers perform encryption over fixed size of data by passing the blocks though an encryption algorithm. Stream ciphers do encryption continuously by generating a keystream that is later XORed with data bit-by-bit. Asymmetric ciphers incorporate computationally intensive algorithms that require a lot of processing power to run and are not suitable for constrained environments like low power IoT devices or RFID tags.

The thesis is a part of the WG-lite project of the ComSec group in the Electrical & Computer Engineering department of the University of Waterloo. WG ciphers are based on the WG transformation and provide a keystream with mathematically proven randomness properties. Previously, only specific instances of WG ciphers have been implemented in hardware and software. The main contribution of the thesis is the development of a universal generic implementation of the whole family. It allows us to compare instances of different sizes directly as well as study design trade-offs between universal and highly

optimized manual implementations of the same instances.

As a part of this thesis, over 26000 different instances were synthesized for extensive evaluation of how different design parameters affect cipher area, one of the most important performance metrics for low power and low cost applications.

Two implementation approaches are used in this thesis - discrete components and constant arrays, which are introduced in Section 2.3.3, a part of Chapter 2 that includes necessary background information and related work. Chapter 3 is dedicated to discrete components implementations. Each of the components performs a specific arithmetic operation and is discussed in a separate section. In Chapter 4 we perform an extensive analysis of the effect of field size, field defining polynomial, implementation approaches and implementation technologies on WG transformation area pre place-and-route. We also propose a way to reduce the search space to find a field defining polynomial that provides the smallest WG transformation area. We select optimal parameters and implement complete WG ciphers of different sizes in Chapter 5. We present instances with 1 bit / cycle output and two options of instances with multiple bits / cycle output, provide performance metrics post place-and-route and compare them with other implementations. In Chapter 6 we give a conclusion and talk about future work possibilities.

# Chapter 2

# Background and Related Work

## 2.1 Mathematical Background

In this section we introduce groups; finite fields, specifically the binary extension fields, and define polynomial basis, which will be the focus of this thesis.

### 2.1.1 Groups and Finite Fields

A group $G$ is a non-empty set with a binary operation $+$, that satisfies the following criteria:

- $\forall\, x, y \in G :\ x + y \in G$ (closure)
- $\forall\, x, y, z \in G :\ x + (y + z) = (x + y) + z$ (associativity)
- $\exists\, e \in G,\ \forall\, x \in G :\ x + e = x$ (identity)
- $\forall\, x \in G,\ \exists\, y \in G :\ x + y = y + x = e$ (inverse)

Group $G$ is called an Abelian group if the binary operation $+$ is commutative:

- $\forall\, x, y \in G :\ x + y = y + x \in G$ (commutativity)

A field $F$ is a set with two operations called addition ($+$) and multiplication ($\cdot$) (and two corresponding identity elements 0 and 1 respectively) with the following conditions [1, 2]:

- $F$ is an Abelian group under addition and therefore satisfies the group criteria (closure, associativity, identity and inverse) for addition
- All elements are associative under multiplication, i.e. $\forall\, x, y, z \in F :\ x{\cdot}(y{\cdot}z) = (x{\cdot}y){\cdot}z$
- There exists a multiplicative identity element 1, i.e. $\forall\, x \in F :\ x \cdot 1 = x$
- Every element except additive identity 0 has multiplicative inverse, i.e. $\forall\, x \in F \setminus \{0\},\ \exists\, y \in F \setminus \{0\} :\ x \cdot y = 1$, i.e. $x^{-1} = y$
- Multiplication is distributive over addition, i.e. $\forall\ x, y, z \in F :\ x \cdot (y + z) = x \cdot y + x \cdot z$

A finite field is a field $F$ with a finite number of elements, and this number is the *order* of the field $F$ [2]. A finite field or *Galois* field $GF_q$ has $q$ elements and it only exists if $q = p^m$, where $p$ is a prime number and $m$ is a positive integer [1].

### 2.1.2 Binary Extension Fields and Polynomial Basis

A field with $p$ elements $GF_p$, where $p$ is a prime number, is called prime field. Elements of this field are integers $\{0, 1, 2, 3, \ldots, p-1\}$. Additions and multiplications in $GF_p$ are regular integer operations followed by reduction *mod p*. A field $GF_{p^m}$ is called an extension field and has $p^m$ elements. If $p = 2$, such a field is called a binary extension field $GF_{2^m}$. The elements of $GF_{2^m}$ can be represented as binary polynomials of degree less than $m$. Similar to how prime fields are constructed by *mod p* reduction, binary extension field can be constructed using reduction by a binary irreducible polynomial $P(x)$ of degree $m$. Addition of polynomials is performed by modulo-2 addition of the corresponding coefficients and multiplication is a regular polynomial multiplication followed by reduction by $P(x)$ [3].

A polynomial basis is defined by an irreducible polynomial $P(x)$. If we take a root $\alpha$ of this polynomial ($P(\alpha) = 0$), then the set $\{1, \alpha, \alpha^2, \alpha^3, \ldots, \alpha^{m-1}\}$ forms a polynomial basis. Every element of the field $GF_{2^m}$ can be represented in this basis using binary coefficients $a_i$:

$$A = \sum_{i=0}^{m-1} a_i \cdot \alpha^i$$

Since each coefficient is binary, such a representation is very convenient to implement on intrinsically binary digital hardware. Addition in polynomial basis is very simple and cheap – all we need is $m$ XOR gates to perform the modulo-2 addition of each corresponding coefficient. Multiplication, however, requires both polynomial multiplication and

reduction, which makes it a more costly operation. Therefore, a reasonable approach in cryptographic hardware optimization is to try to minimize the number of multiplications for a given circuit to keep its area small. Another common operation is squaring. Squaring can be implemented using a multiplier with two united inputs. In polynomial basis, such a multiplier is reduced to a much simpler circuit by the synthesis tools.

It is important to mention normal basis $N = \{\beta, \beta^q, \beta^{q^2}, \ldots, \beta^{q^{m-1}}\}$, which is formed by a normal element $\beta$ and its conjugates with respect to $GF_2$. From the hardware perspective, normal basis provides essentially free squaring in a form of rotation of all coefficients, which is just a rearrangement of wires inside a circuit. Multipliers, however, are more complex than in polynomial basis.

The trace function on an element $\alpha$ with respect to the underlying subfield $GF_q$ is a function that maps $GF_{q^m} \rightarrow GF_q$:

$$Tr_{GF_q}^{GF_{q^m}}(\alpha) = \sum_{i=0}^{m-1} \alpha^{q^i} = \alpha^{q^0} + \alpha^{q^1} + \alpha^{q^2} + \cdots + \alpha^{q^{m-1}}$$

If the subfield $GF_q$ is prime, the function is called absolute trace [4]. In this thesis trace functions will be denoted as $Tr_1^m$ for convenience.

## 2.2 The WG Cipher

WG (Welch-Gong) is a family of parametrized stream ciphers that provide mathematically proven randomness properties of the keystream such as guaranteed period, $k$-tuple distribution, balance property, run property, ideal two-level autocorrelation and large linear span [5, 6]. A WG stream cipher consists of a Linear Feedback Shift Register (LFSR), a WG permutation function with optional decimation and an absolute trace function, as shown in Figure 2.1. The generated keystream is XORed with plaintext *i_text* to provide ciphertext *o_text*.

The WG permutation can be referred to as the WGP and if optional decimation $d$ is present it can be referred to as DWGP. For simplicity, we will always use DWGP notation, and refer to the case without decimation as DWGP with $d = 1$ later in this thesis. DWGP followed by the absolute trace function will be referred to as the WG transformation with decimation, or DWGT.

WG ciphers operate over $GF_{2^m}$ extension fields with $m \bmod 3 \neq 0$ condition, where $m$ is a **field dimension**. WG instantiations with low values of $m$ are smaller and consume less power, while the ones with higher $m$ values provide higher security [6].

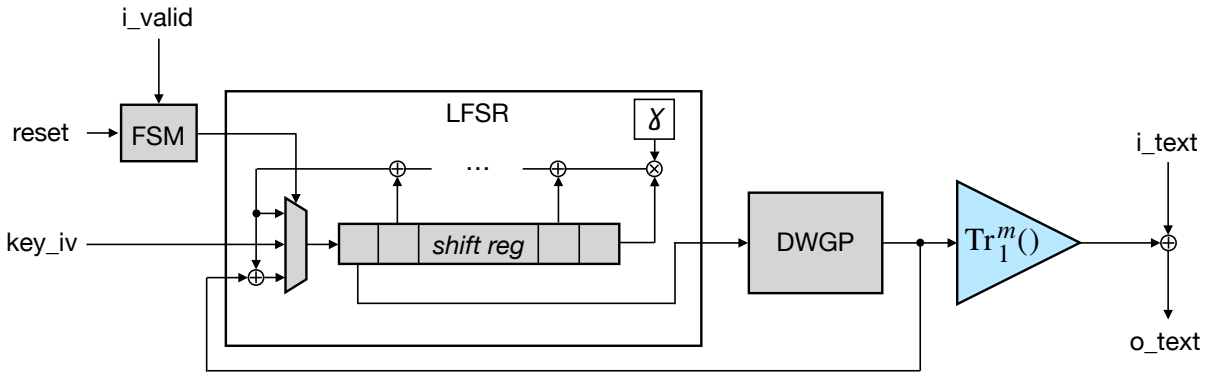


Figure 2.1: WG cipher structure

Decimation is used to increase the algebraic degree and the algebraic immunity of the cipher [7]. The DWGP function is defined as:

$$DWGP(A) = h(A^d + 1) + 1,$$

where

$$h(A) = A + A^{r_1} + A^{r_2} + A^{r_3} + A^{r_4}$$

is a permutation polymonial. The exponents are:

$$r_1 = 2^k + 1,\ r_2 = 2^{2\cdot k} + 2^k + 1,\ r_3 = 2^{2\cdot k} - 2^k + 1,\ r_4 = 2^{2\cdot k} + 2^k - 1$$

where $3 \cdot k = 1 \pmod{m}$. DWGT can be expressed as:

$$DWGT(A) = Tr_1^m(DWGP(A)),$$

i.e. by applying the absolute trace function $Tr_1^m()$ over DWGP. For $d = 1$, it is possible to rewrite the equation for $h(A)$ as:

$$\begin{aligned} h(A) &= A + (A^{2^k} \cdot A) + (A^{2^{2\cdot k}} \cdot A^{2^k} \cdot A) + (A^{2^{2\cdot k}} \cdot A^{-2^k} \cdot A) + (A^{2^{2\cdot k}} \cdot A^{2^k} \cdot A^{-1}) \\ &= A + A \cdot (A^{2^k} + (A^{2^{2\cdot k}} \cdot A^{2^k}) + (A^{2^{2\cdot k}} \cdot (A^{-1})^{2^k})) + (A^{-1} \cdot (A^{2^{2\cdot k}} \cdot A^{2^k})) \end{aligned}$$

to reduce the number of multiplications from 7 to 5 and the number of very expensive inversion operations from 2 to 1.

## 2.3 Implementation Techonologies

Hardware was implemented using VHDL. The following sections briefly describe the details, including which software was used and how we achieved a high degree of design automation.

### 2.3.1 ASIC library and FPGAs

In this thesis we perform pre place-and-route (logic) synthesis for extensive area analysis using STMicroelectronics 65 nm CMOS ASIC library as well as two FPGAs: STRATIX IV and MAX 10. The place-and-route synthesis is done for several complete instances of WG ciphers using the 65 nm CMOS library.

Quick specifications of the FPGAs are given in Table 2.1. The main differences we are concerned with are the number of configurable *look-up tables* (LUT), which are sometimes referred to as *logic elements* (LE) in various sources, as well as their input and output size. The LUT is a core atomic component of an FPGA and is used to implement logic functions. For ASICs, different gates from an extensive library (NOT, NAND, NOR, XOR, AND, OR, etc.) are used to construct digital circuits. In FPGAs, no such gates exist. Instead, each LUT has a fixed number of input bits (for example, 4 or 6) and a fixed number of output bits (for example, 1 or 2) and can be configured to represent an arbitrary truth table. If a particular logic function requires more inputs or outputs than a single LUT can provide, two of more LUTs will be dedicated to it. FPGAs also contain dedicated flip-flops (registers) and memory bits.

| | **Intel / Altera STRATIX IV FPGA** | **Intel / Altera MAX 10 FPGA** |
|---|---|---|
| **Model** | EP4SGX70HF35 | 10M08SAE144C8GES |
| **LUTs** | 58080 (8 input bits, 2 output bits)* | 8064 (4 input bits, 1 output bit) |
| **Registers** | 58080 | 8064 |
| **Memory Bits** | 6617088 | 387072 |

* certain restrictions apply [8]

Table 2.1: STRATIX IV and MAX 10 FPGAs specifications

The reason why we chose ASIC as a main implementation for area profiling is because logic synthesis area results for ASIC are almost directly proportional to physical (post place-and-route) area – unless target density is pushed to its absolute limit for each particular

design, which is not the case in this thesis. A density value can be found to work well for all designs that results in low increase from logic synthesis area to physical synthesis area (like 5.3% for density 0.95) without unreasonable increase in delay due to congestion. Place-and-route on FPGA, however, is fundamentally different because routing is much harder due to constrained resources in a form of fixed architecture of look-up tables and limited amount of interconnects in-between. For example, it is common to have LUTs completely blocked (i.e. impossible to use because all interconnects around them are being used for other cells). Moreover, synthesis tools naturally do not put as much effort in efficient resource allocation and routing when design is significantly smaller than the FPGA. That, combined with different LUT and Slice designs between different FPGAs means that logic synthesis results are less useful than for ASIC as they are specific to a particular FPGA model and do not consistently predict physical synthesis results, and physical synthesis results are still very specific to the FPGA model or family.

### 2.3.2 The tools for synthesis, simulation and automation

CAD software used in this thesis:

- ***Synopsys Design Compiler*** - logic (pre place-and-route) synthesis for ASIC implementations
- ***Cadence Encounter*** by *Cadence Design Systems* - physical synthesis (post place-and-route) for ASIC implementations and power consumption analysis
- ***Model Technology ModelSim SE-64*** - design simulation at various steps like program simulation to verify functional behaviour, logic simulation for pre place-and-route synthesis results and physical simulation for post place-and-route. ModelSim is able to perform timing simulations with *.sdf* (standard delay file) generated by synthesis tools. The output of ModelSim in a form of *.vcd* (value change dump) file can then be used for precise power analysis in Cadence Encounter.
- ***Mentor Graphics Precision RTL*** - logic synthesis for STRATIX IV and MAX 10 FPGAs

In order to make it possible to synthesize and evaluate thousands of various instances, we had to take advantage of an extensive library of proprietary python scripts, which were developed and are maintained by Dr. Mark D. Aagaard. We refer to them as UW tools. They interface with the CAD software through the command line interface and make the

process easy, fast and reliable. Brief description of UW tools functionality is presented below:

- **.uwp file** - UW project file which contains the paths to all design files, testbenches, necessary libraries (packages) and the name of top level entity and its architecture.
- **uw-com** - compile design files or a project
- **uw-sim** - simulate a project in ModelSim
- **uw-synth** - synthesize design files or a project. Possible arguments include:
  - **--logic** - perform pre place-and-route synthesis
  - **--chip** - perform place-and-route synthesis
  - **-O0, -O1, -O2,** etc. – optimization level. A template with different settings for synthesis tools. *-O2* option is used exclusively in this thesis as the best option to minimize area
  - **--board** - choose the implementation technology. Possible options include *cmos65nm*, *stratixiv*, *lstep* and other supported ASIC libraries or FPGAs
  - **-d** - density for place-and-route synthesis, indicates area target w.r.t. pre place-and-route area. *-d=0.95* (95%) option is used in all cases as the highest density that *a*) works consistently for all instances without violations and *b*) does not result in unreasonable increase in circuit delay due to congestion caused by inability of the CAD tools to route wires optimally without violating area requirements.
  - **-t** - target delay for logic and physical synthesis in nanoseconds. *-t=100* is chosen as a way to allow synthesis tools to focus only on area minimization. For several design instances that have delays close to 100 *ns* and 200 *ns*, *-t=200* and *-t=300* flag is used instead.
  - **-G** - generic parameters passed to top level entity
  - **-C** - configuration constants, that allow to have placeholders in any file type within a project. For example, using *-C FIELD_DIR=pb_11/pb_11_2* as an argument in a *uw-synth* command will substitute placeholders *$FIELD_DIR* and *${FIELD_DIR}* with the text *pb_11/pb_11_2*, which makes it possible to define a folder path inside *.vhd* or *.uwp* files.

- **uw-loop** - cycle through synthesis *(uw-synth)*, timing simulation *(uw-sim)* and power analysis
- **uw-report** - read data (area, performance, power, generic parameters used, etc.) from report directories that were automatically created by *uw-synth* and export the data as *.csv* file.
- **uw-clean** - delete temporary folders and files left after synthesis or simulation
- **uw-plot** - create charts from data in *.csv* files

In addition to the above, a large collection of custom python scripts is built on top of the core UW tools library. Those scripts are typically used for batch synthesis or batch data processing and are tailored to specific needs. While all of the core tools were already present before the work on this thesis has begun, a number of custom scripts had to be written or modified while working on this thesis as well as minor changes to the core UW tools.

Several VHDL testbenches for individual components were used in this thesis. Also, a robust testbench environment developed by Dr. Mark Aagaard was used for complete WG stream cipher instances. Test vectors for individual components as well as for all WG cipher instances were provided by Zidaric using GAP packages from [9]. Simulation of WG ciphers included key and initialization vector (IV) loading phase, initialization phase and running phase to generate 1024 bits of keystream for comparable power analysis results among different instances.

### 2.3.3 Constant Array and Discrete Components

There are two different approaches in implementing cryptographic functions in VHDL. The first approach is using discrete components, which is thoroughly discussed in Chapter 3 and is the main focus of this thesis. Such components include squarers, multipliers, generic exponentiations and inversions that perform actual finite field arithmetic over the inputs in real time. The second approach is using constant array, when a cryptographic block as a look-up table with an input and an output. For each input value the output value is written as a VHDL constant. The values are pre-computed and the corresponding VHDL modules generated by Zidaric using GAP packages from [9].

## 2.4 Related Work

In this section we discuss existing work on hardware implementations of various instances of WG ciphers as well as two stream ciphers from the eSTREAM competition [10] - Grain and Trivium. As this thesis is mostly concentrated on ASIC (65 nm CMOS), at least when it comes to complete cipher implementations, it will be also the focus of this review.

### 2.4.1 Hardware Implementations of WG

The first implementation of WG cipher was done for WG-29 [11], which was a cipher submitted to the *eSTREAM* competition in 2005. In that work both hardware and software implementations were done using type II optimal normal basis for efficient field arithmetic. The multi-output WG architecture called MOWG was proposed and implemented in [12]. MOWG-7, MOWG-11 and MOWG-29 generate 3, 6 and 17 bits / cycle respectively using a different MOWG transformation with deep pipelining compared to 1 bit / cycle for standard WG architecture. An improved MOWG-29 as well as a new more efficient implementation of the standard WG-29 were done in [13]. Area reduction and speed increase for the standard WG-29 were achieved due to the discovery of useful properties of the trace function and elimination of 4 multipliers. In [14], WG-29 and WG-16 ciphers were implemented using polynomial basis, and in addition to standard WG, pipelined and serialized architectures were explored. Pipelining allows to increase maximum clock speed at a cost of area increase due to extra registers. Serializing decreases circuit area by having fewer key components (like multipliers) and performing the same amount of computation over several clock cycles instead of 1 clock cycle, while reusing these components. Also, implementations using regular multipliers and Karatsuba multipliers are shown. The approach that lead to significant reduction in number of multipliers and therefore area is similar to [13]. WG-16 was implemented using only 6 multipliers, while standard technique used in this thesis yields 12.

For WG-29 (ONB II) implementation in [13] and WG-29 (PB) and WG-16 (PB) implementations [14], the DWGP signal is not present explicitly during the run phase. In order to recover the DWGP signal for initialization phase, one initialization round takes 3 cycles for WG-16 (PB), 7 cycles for WG-29 (PB) and 3 cycles for WG-29 (ONB II), during which some internal components, including a multiplier, are reused. This approach provides smaller area at the cost of a longer initialization phase.

WG-5 and WG-8, two lightweight cipher instances for applications like RFID tags, were

| Cipher | Basis | State / Key (bits) | Bits / cycle | d | Area (GE) | Max freq. (MHz) | Max tput (Mbps) | Max T/A (Mbps/GE) | Power ($\mu$W) | Energy (pJ/bit) | Source |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 65 nm CMOS ASIC library: | | | | | | | | | | | |
| WG-29 | ONB II | 319/128 | 1 | 1 | 33180 | 144 | 144 | 0.00434 | 7280 (140 MHz) | 52.00 (140 MHz) | [11] † |
| WG-29 | ONB II | 319/128 | 1 | 1 | 19892 | 169 | 169 | 0.00850 | 4450 (140 MHz) | 31.79 (140 MHz) | [13] † |
| MOWG-29 | | | | | 26261 | 151 | 2567 | 0.09775 | 5890 (140 MHz) | 2.47 (140 MHz) | |
| WG-29 | PB | 319/128 | 1 | 1 | 17165 | 202 | 202 | 0.01177 | - | - | [14] † |
| WG-29 ‡ | | | 1 | | 21190 | 917 | 917 | 0.04328 | | | |
| WG-29 †† | | | 1/6 | | 7050 | 610 | 101 | 0.01433 | | | |
| WG-16 | PB | 512/256 | 1 | 1057 | 9103 | 189 | 189 | 0.02076 | | | |
| WG-16 ‡ | | | 1 | | 11795 | 1149 | 1149 | 0.09741 | | | |
| WG-16 †† | | | 1/6 | | 5267 | 680 | 113 | 0.02145 | | | |
| WG-16 | PB * | 512/256 | 1 | 1057 | 8060 | 193 | 193 | 0.02395 | - | - | [14] † |
| WG-16 ‡ | | | | | 10681 | 1370 | 1370 | 0.12827 | | | |
| WG-16 †† | | | 1/6 | | 5026 | 714 | 119 | 0.02368 | | | |
| WG-16 ‡ | TFB | 512/256 | 1 | 1057 | 26300 | 2440 | 2440 | 0.09278 | - | - | [15] |
| | | | | | 10900 | 880 | 880 | 0.08073 | | | |
| | | | | | 9900 | 330 | 330 | 0.03333 | | | |
| WG-16 ‡ | TFB | 512/256 | 1 | 1057 | 12031 | 552 | 552 | 0.04588 | 25500 (552 MHz) | 46.12 (552 MHz) | [16] |
| WG-8 | PB ** | 160/80 | 1 | 19 | 1786 | 500 | 500 | 0.27996 | - | - | [17] |
| | | | 11 | | 3942 | 610 | 6710 | 1.70218 | | | |
| 130 nm CMOS ASIC library: | | | | | | | | | | | |
| WG-5 | PB *** | 160/80 | 1 | 1 | 1229 | - | - | - | 0.78 (0.1 MHz) | 7.80 (0.1MHz) | [6] |
| | | | | 11 | 1235 | | | | 0.79 (0.1 MHz) | 7.80 (0.1MHz) | |
| | | | 2 | 1 | 1350 | | | | 0.84 (0.1 MHz) | 4.20 (0.1MHz) | |
| | | | | 11 | 1360 | | | | 0.85 (0.1 MHz) | 4.20 (0.1MHz) | |
| 180 nm CMOS ASIC library: | | | | | | | | | | | |
| WG-5 | PB *** | 160/80 | 1 | 1 | 1361 | - | - | - | - | - | [6] |
| | | | | 11 | 1373 | | | | | | |
| | | | 2 | 1 | 1508 | | | | | | |
| | | | | 11 | 1521 | | | | | | |

**d** = decimation exponent; **tput** = throughput; **T/A** = throughput over area

† synthesis results only pre place-and-route

‡ pipelined implementation

†† serialized implementation

* Karatsuba multiplier was used

** implemented using constant array

*** implemented using GAP-derived equation

Table 2.2: Hardware implementations of WG stream cipher

proposed in [6] and [17] respectively. WG-5 had DWGP implemented using equations from a symbolic algebra program. Versions with 1 bit / cycle and 2 bits / cycle using a single copy of the LFSR and two copies of the combinational datapath were implemented and were extensively compared to competing ciphers, outperforming them. WG-8 had DWGP implemented using a constant array and 3 different tower field constructions. It turned out that the constant array version provided significantly smaller area compared to the tower field versions for that field size.

WG-16 with 2 different tower field constructions and extensive pipelining was explored in [15] and achieved frequency of 2.5GHz using the 65 nm CMOS ASIC library. Another pipelined WG-16 with tower field construction is presented in [16].

A compilation of hardware implementations of WG is shown in Table 2.2. The WG cipher family in this thesis is done using generic VHDL code and the results can be used as a baseline for comparison with previous work, where hand-optimized solutions were presented. It makes it possible to quantify the performance trade-offs (in terms of area, throughput, power, etc.) between these two approaches. Moreover, having consistent impementation allows us to compare the performance of WG instances using various field sizes directly without the need to take unique optimizations into account. Our implementation results are presented in Sections 5.2.1 and 5.3.3 and a more in-depth comparison with previous work is given in Section 5.4.

### 2.4.2 Hardware Implementations of Grain

Grain was proposed by Hell, Johansson & Meier [18] as a stream cipher for low-power and low-performance environments. It consists of an LFSR, a Non-linear Feedback Shift Register (NLFSR) and a filtering function. Two instances with 80-bit and 128-bit internal state are called Grain and Grain-128.

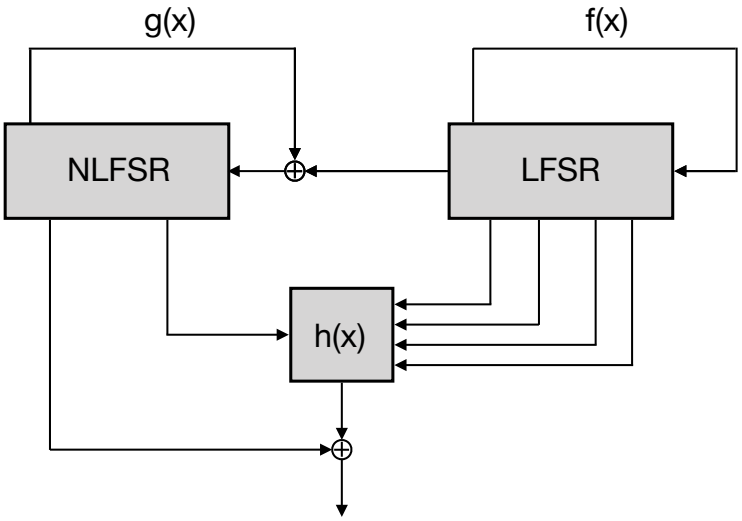


Figure 2.2: Internal structure of Grain stream cipher [18]

The internal structure is shown in Figure 2.2. $f(x)$ is a linear feedback, $g(x)$ is a non-linear feedback. The NLFSR state is updated by XOR-ing $g(x)$ with an output from the LFSR. The cipher outputs 1 bit / clock cycle. It is possible to increase throughput by shifting both shift registers by more than one stage at a time, which requires multiple copies of feedback functions and the filtering function. The initialization phase is 160 cycles for Grain and 256 cycles for Grain-128. Cipher operation and design decisions are discussed in [18] and [19].

| ASIC library | Cipher | Area | Max freq. (MHz) | Bits / cycle | Max tput (Mbps) | Max T/A (Mbps/$\mu$m$^2$) | Max T/A (Mbps/GE) | Source |
|---|---|---|---|---|---|---|---|---|
| 250 nm | Grain | 119821 $\mu m^2$ | 300 | 16 | 4475 | 0.0373 | - | [20] |
| 180 nm | Grain | 1410 GE | - | 1 | - | - | - | [6] |
| | | 1585 GE | | 2 | | | | |
| 130 nm | Grain | 1259 GE | - | 1 | - | - | - | [6] |
| | | 1393 GE | | 2 | | | | |
| | | 1294 GE | 724.6 | 1 | 724.6 | - | 0.5600 | [21] |
| | | 1678 GE | 694.4 | 4 | 2777.6 | | 2.1465 | |
| | | 2191 GE | 632.9 | 8 | 5063.2 | | 2.3109 | |
| | | 3239 GE | 617.3 | 16 | 9876.8 | | 3.0493 | |
| | Grain-128 | 1857 GE | 925.9 | 1 | 925.9 | - | 0.4986 | |
| | | 2129 GE | 584.8 | 4 | 2339.2 | | 1.0987 | |
| | | 2489 GE | 581.3 | 8 | 4650.4 | | 1.8684 | |
| | | 3189 GE | 540.5 | 16 | 8648.0 | | 2.7118 | |
| | | 4617 GE | 452.5 | 32 | 14480.0 | | 3.1362 | |
| 90 nm | Grain | 4911 $\mu m^2$ | 565 | 1 | 565 | 0.1150 | - | [22] * |
| | | 10548 $\mu m^2$ | 495 | 16 | 7920 | 0.7508 | | |

tput = throughput; T/A = throughput over area

* synthesis results only pre place-and-route

Table 2.3: Hardware implementations of Grain stream cipher

Results of Grain hardware implementations from difference sources are compiled in Table 2.3.

### 2.4.3 Hardware Implementations of Trivium

The Trivium stream cipher construction is based on block cipher design principles. A 288-bit internal state is formed by an 80-bit secret key and an 80-bit initialization vector (IV) in the following format:

$$
\begin{aligned}
(s_1, s_2, \ldots, s_{93}) &\leftarrow (K_{80}, K_{79}, \ldots, K_1, 0, \ldots, 0) \\
(s_{94}, s_{95}, \ldots, s_{177}) &\leftarrow (IV_{80}, IV_{79}, \ldots, IV_1, 0, \ldots, 0) \\
(s_{178}, s_{179}, \ldots, s_{288}) &\leftarrow (0, \ldots, 0, 1, 1, 1)
\end{aligned}
$$

and after initialization phase provides $2^{64}$ bits of key stream through iterative process that extracts 15 bits from the state, updates 3 bits of the state and generates 1 output bit $z_i$. The next iteration begins after the state bits are rotated [23].

The internal structure of Trivium is shown in Figure 2.3. A list of hardware implementations of Trivium by various sources is compiled in Table 2.4.

| **ASIC library** | **Area** | **Max freq. (MHz)** | **Bits / cycle** | **Max tput (Mbps)** | **Max T/A (Mbps/$\mu$m$^2$)** | **Max T/A (Mbps/GE)** | **Source** |
|---|---|---|---|---|---|---|---|
| 250 nm | 144128 $\mu m^2$ | 312 | 64 | 18568 | 0.1288 | - | [20] |
| 180 nm | 2530 GE | - | 1 | - | - | - | [6] |
| | 2569 GE | | 2 | | | | |
| 130 nm | 2088 GE | - | 1 | - | - | - | [6] |
| | 2122 GE | | 2 | | | | |
| | 2580 GE | 327.9 | 1 | 327.9 | - | 0.1271 | [21] |
| | 2627 GE | 574.7 | 2 | 1149.4 | | 0.4375 | |
| | 2705 GE | 473.9 | 4 | 1895.6 | | 0.7008 | |
| | 2952 GE | 471.7 | 8 | 3773.6 | | 1.2783 | |
| | 3166 GE | 467.3 | 16 | 7476.8 | | 2.3616 | |
| | 3787 GE | 350.9 | 32 | 11288.8 | | 2.9809 | |
| | 4921 GE | 348.4 | 64 | 22297.6 | | 4.5311 | |
| 90 nm | 7428 $\mu m^2$ | 840 | 1 | 840 | 0.1131 | - | [22] * |
| | 13440 $\mu m^2$ | 800 | 64 | 51200 | 3.8095 | | |

tput = throughput; T/A = throughput over area

* synthesis results only pre place-and-route

Table 2.4: Hardware implementations of Trivium stream cipher

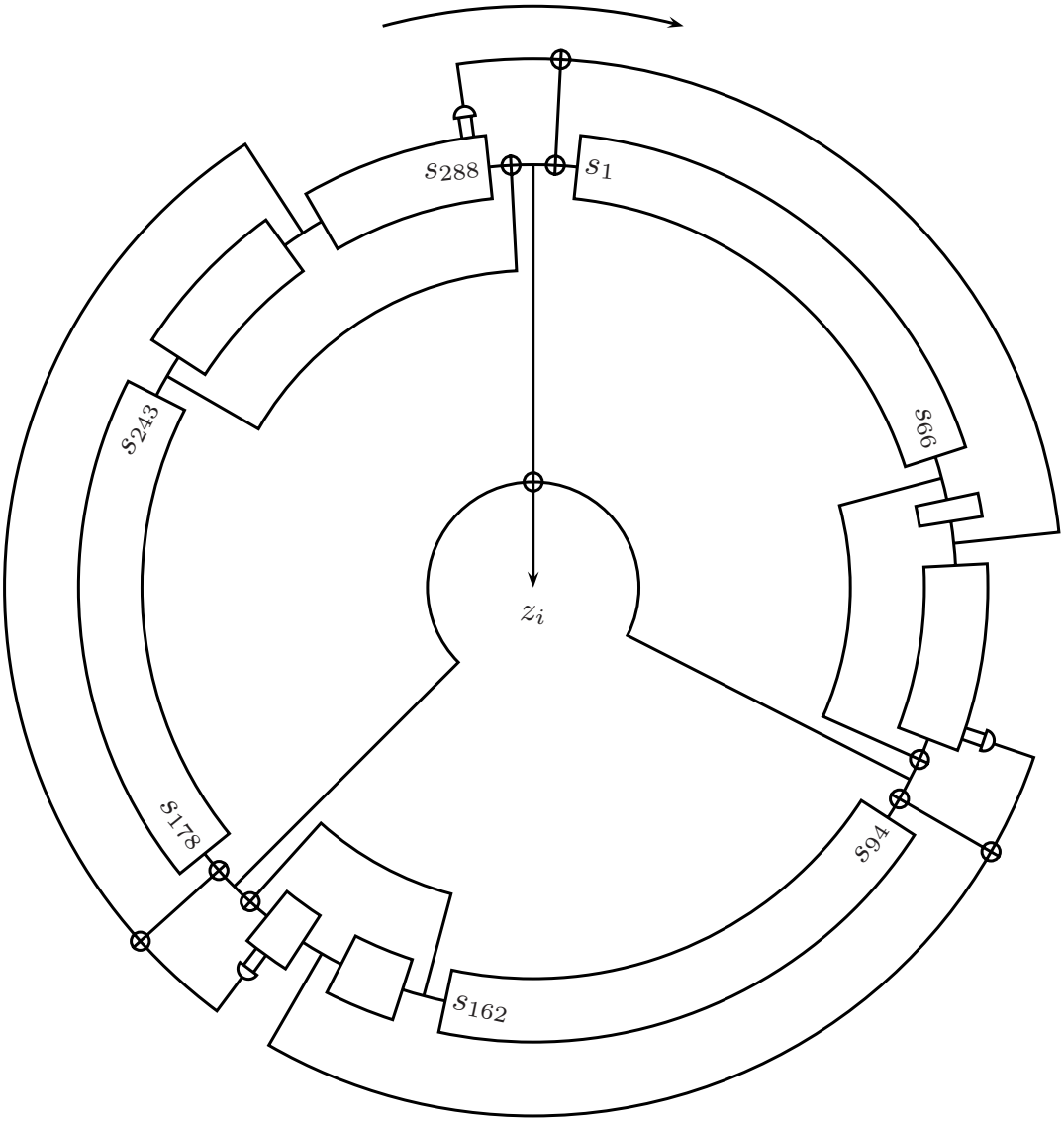


Figure 2.3: Internal structure of Trivium stream cipher [23]

# Chapter 3

# Discrete Components Implementation of DWGP / DWGT

This chapter is dedicated to parametrized implementations of DWGP and DWGT and their building blocks, such as multipliers, squarers and exponentiations. These blocks perform finite field arithmetics in polynomial basis. Instances that are built using this approach are referred to as *"discrete components implementations"* or simply *"comp"* in certain plots.

## 3.1 Multiplier

A multiplier is a complex building block and significantly affects the area of the WG cipher. As a part of the WG-lite project [24], different members of ComSec group wrote VHDL implementations of combinational discrete multipliers for polynomial basis using different algorithms (Karatsuba, Montgomery, Interleaved) and type 2 optimal normal basis multiplier.

This thesis use the classic combinational discrete multiplier with brief comparison with Karatsuba [25] multiplier. Other multiplier architectures were considered as well. The Karatsuba multiplier [25] was chosen for evaluation because it has clear potential to give area advantage over the classic multiplier even in a fully combinational implementation.

The classic multiplier can be described as a regular polynomial multiplication followed by a reduction, which brings the result back to the extension finite field $GF_{2^m}$, where $m = \text{field\_sz}$ (Figure 3.1).

| Multiplier (mul_comb.vhd) | | | |
|---|---|---|---|
| | **Kind** | **Type** | **Description** |
| **field_sz** | Generic | *integer* | dimension of binary extension field |
| **field_poly** | Generic | *std_logic_vector (field_sz downto 0)* | field defining polynomial |
| **i_a** | Input | *std_logic_vector (0 to field_sz - 1)* | first input |
| **i_b** | Input | *std_logic_vector (0 to field_sz - 1)* | second input |
| **o_z** | Output | *std_logic_vector (0 to field_sz - 1)* | output |

Table 3.1: Parameters for the multiplier entity

A multiplier entity description is given in Table 3.1.

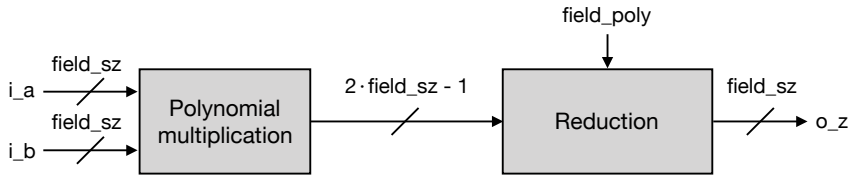


Figure 3.1: Principal structure of multiplier

Figure 3.2 shows an example reduction matrix for the field defining polynomial $f(x) = x^5 + x^2 + 1$, where $[a_0\ a_1\ a_2\ \ldots\ a_8\ a_9]$ is s a product of polynomial multiplication with length $2 \cdot m - 1$, and $[b_0\ \ b_1\ \ b_2\ \ b_3\ \ b_4]$ is a is a final multiplication result after reduction. The red line in reduction matrix corresponds to the field defining polynomial. It can be read as $x^5 = x^2 + 1$, starting with 1 for a constant term, 0 for $x$, 1 for $x^2$, 0 for $x^3$ and 0 for $x^4$. The two last lines, for example, correspond to $x^8 = x^3 + x^2 + 1$ and $x^9 = x^4 + x$ respectively. The effect of field defining polynomial on the classic multiplier area is studied in Section 4.1.

As we can see in Table 3.2, the Karatsuba multiplier gives the highest area advantage for ciphers using larger fields like WG-16 and this advantage gradually decreases with field size. The lowest input size for which it is beneficial to do Karatsuba decomposition is 8 bits. That means if we need to multiply two 7-bit vectors, it is better to use the classic multiplier instead. Our implementation iteratively performs Karatsuba decomposition and then instantiates the classic multiplier once the input size drops below 8 bits. Actual area for both multiplier architectures is shown in Figure 3.3

In this thesis classic multiplier will be used for area studies due to its more consistent

$$\begin{bmatrix} a_0 & a_1 & a_2 & \dots & a_8 & a_9 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} b_0 & b_1 & b_2 & b_3 & b_4 \end{bmatrix}$$

Figure 3.2: Using a reduction matrix to obtain the final multiplication result

| | **Reference (classic)** | **Levels of decomposition** | | |
|---|---|---|---|---|
| | | **1** | **2** | **3** |
| **WG-5** | 71 GE | +16.9% | +40.8% | – |
| **WG-7** | 141 GE | +6.4% | +17.7% | – |
| **WG-8** | 197 GE | **−4.1%** | +4.1% | – |
| **WG-10** | 288 GE | **−5.2%** | +5.2% | +25.0% |
| **WG-11** | 349 GE | **−3.4%** | −2.3% | +22.6% |
| **WG-13** | 510 GE | **−12.4%** | −1.4% | +10.4% |
| **WG-14** | 587 GE | **−11.4%** | −8.0% | +2.6% |
| **WG-16** | 783 GE | −13.7% | **−16.6%** | −8.4% |

Table 3.2: Area change for the Karatsuba multipliers w.r.t. the classic multiplier (%)

behaviour across different fields and our focus on smaller fields. Another reason why Karatsuba was not used as a main architecture is given in the following Section 3.2.

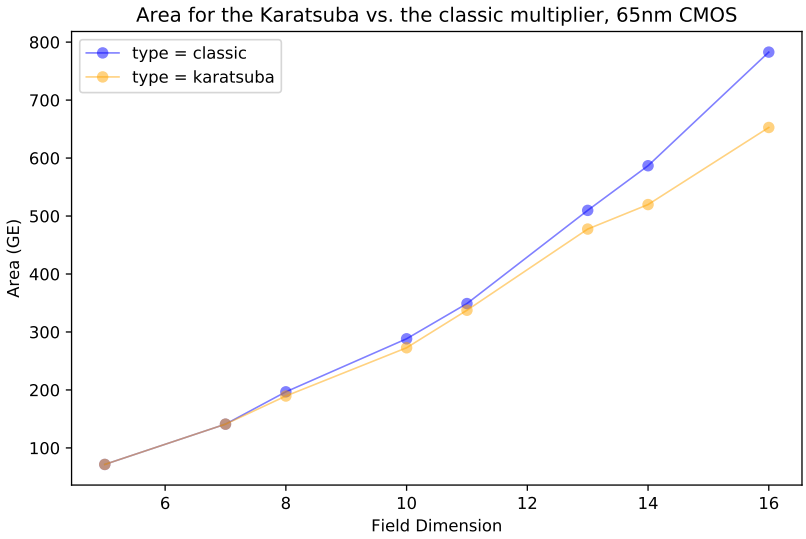


Figure 3.3: Area comparison of the Karatsuba and the classic multipliers (65 nm CMOS)

## 3.2 Squarer

Squaring is the most frequent operation inside the WG cipher and has to be implemented in an efficient way. Entity description can be found in Table 3.3.

| **Squarer (sqr_comb.vhd)** | | | |
|---|---|---|---|
| | **Kind** | **Type** | **Description** |
| **field_sz** | Generic | *integer* | dimension of binary extension field |
| **field_poly** | Generic | *std_logic_vector (field_sz downto 0)* | field defining polynomial |
| **i_a** | Input | *std_logic_vector (0 to field_sz - 1)* | input |
| **o_z** | Output | *std_logic_vector (0 to field_sz - 1)* | output |

Table 3.3: Parameters for the squarer entity

Unlike in normal basis, where squaring is just a cyclic shift, the polynomial basis implementation is not as straightforward. Let's take an element of WG-10 with the polynomial basis and the field defining polynomial $f(x) = x^{10} + x^3 + 1$ as an example:

$$A = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9)$$

All coefficients belong to the ground binary field, so it is possible to express $A^2$ like this:

$$A^2 = \sum_{i=0}^{9} a_i \cdot x^{2 \cdot i} = (x^{18} \cdot a_9 + x^{16} \cdot a_8 + x^{14} \cdot a_7 + x^{12} \cdot a_6 + x^{10} \cdot a_5) + (x^8 \cdot a_4 + x^6 \cdot a_2 + x^2 \cdot a_1 + a_0)$$

$$\begin{aligned}
A_L &= x^8 \cdot a_4 + x^6 \cdot a_2 + x^2 \cdot a_1 + a_0 \\
A_H &= x^{18} \cdot a_9 + x^{16} \cdot a_8 + x^{14} \cdot a_7 + x^{12} \cdot a_6 + x^{10} \cdot a_5 \\
&= (x^8 \cdot a_9 + x^6 \cdot a_8 + x^4 \cdot a_7 + x^2 \cdot a_6 + a_5) \cdot x^{10} \\
&= (x^8 \cdot a_9 + x^6 \cdot a_8 + x^4 \cdot a_7 + x^2 \cdot a_6 + a_5) \cdot (x^3 + 1) \\
&= (x^{11} + x^8) \cdot a_9 + (x^9 + x^6) \cdot a_8 + (x^7 + x^4) \cdot a_7 + (x^5 + x^2) \cdot a_6 + (x^3 + 1) \cdot a_5 \\
&= (x^8 + x^4 + x) \cdot a_9 + (x^9 + x^6) \cdot a_8 + (x^7 + x^4) \cdot a_7 + (x^5 + x^2) \cdot a_6 + (x^3 + 1) \cdot a_5
\end{aligned}$$

We can rewrite:

$$\begin{aligned}
A^2 &= A_H + A_L \\
&= x^9 \cdot a_8 + x^8 \cdot (a_9 + a_4) + x^7 \cdot a_7 + x^6 \cdot (a_8 + a_3) + x^5 \cdot a_6 + x^4 \cdot (a_9 + a_7 + a_2) \\
&\quad + x^3 \cdot a_5 + x^2 \cdot (a_6 + a_1) + x \cdot a_9 + (a_5 + a_0)
\end{aligned}$$

In terms of hardware, such single squarer requires six XOR gates as shown in Figure 3.4. Each XOR gate consists of 6 transistors and occupies the circuit area of 1.5 GE.

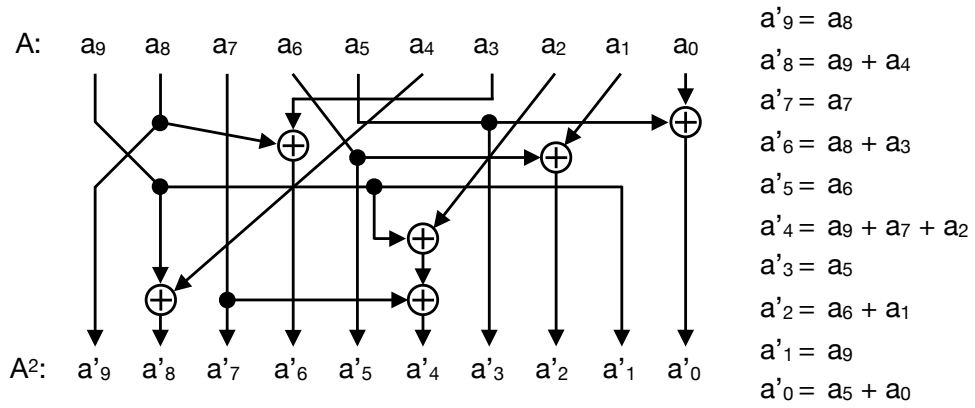


Figure 3.4: Squaring hardware for WG-10, $f(x) = x^{10} + x^3 + 1$

In WG generally there are chains of subsequent squarers in various parts of the design. For example, WG-10 has chains of length 1, 2, 3, 4 and 7 squarers. A chain that consists of

several squarers can be optimized due to different XOR gates cancelling each other out and due to the appearance of common sub-expressions. For example, a chain of 5 subsequent squarers will not have 5 times more XOR gates. It is, however, unnecessary to perform such optimizations by hand as they will be done automatically by the synthesis tools. Area results (pre place-and-route) for chains of squarers of different length are shown in Figure 3.5. In the figure, each line represents a different field. Chain length varies between 1 (single squarer) and $2^{m-1}$, where $m$ = field_sz. The area rises as the number of squarers in chain increases until reaching saturation somewhere in the middle, and then the area starts to decrease, with the chain of length $2^{m-1}$ being very close in area to a single squarer. The chain with length $2^m$ is a simple wire, because $A^{2^m} = A$, and therefore is not shown. Chains with length above $2^m$ do not exist because $A^{2^{m-1+n}} = A^n$, where $n$ is an integer number.

We can also see from Figure 3.5 that the chains of squarers do not exceed 140 GE even for WG-16. That is equal to the area of one classic multiplier for WG-7 according to Figure 3.3. The same multiplier for WG-16 has the area of 783 GE.

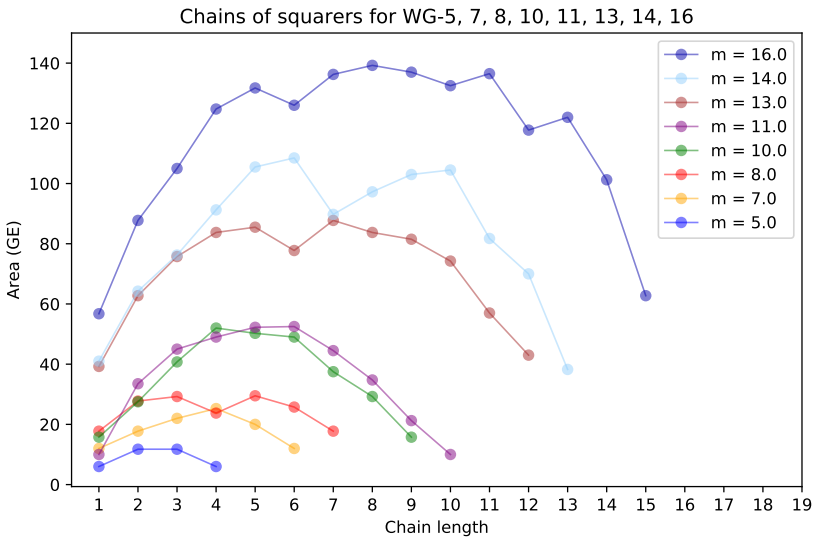


Figure 3.5: Area for chains of squarers using the classic multiplier (65 nm CMOS)

An important implementation detail is that the squarer component in polynomial basis in terms of VHDL code is a combinational classic multiplier with merged inputs – one input signal goes to both multiplier inputs. Synthesis tools were capable of reducing the

complex multiplier down to a very small squaring circuit. Such approach allows for generic code without the need to manually derive the squaring equation for each polynomial basis or even write separate VHDL code for squarer hardware. Squarer for normal basis is a cyclic rotation by 1, a trivial piece of hardware (wire) written as another architecture of the same entity.

The same, however, can not be said about the Karatsuba multiplier. We were unable to force the Design Compiler to simplify the squaring equations reliably, which resulted in huge area for long chains of squarers as can be seen in Figure 3.6. The data without WG-16 and WG-14 is shown in Figure 3.7 to highlight the behaviour for smaller fields.

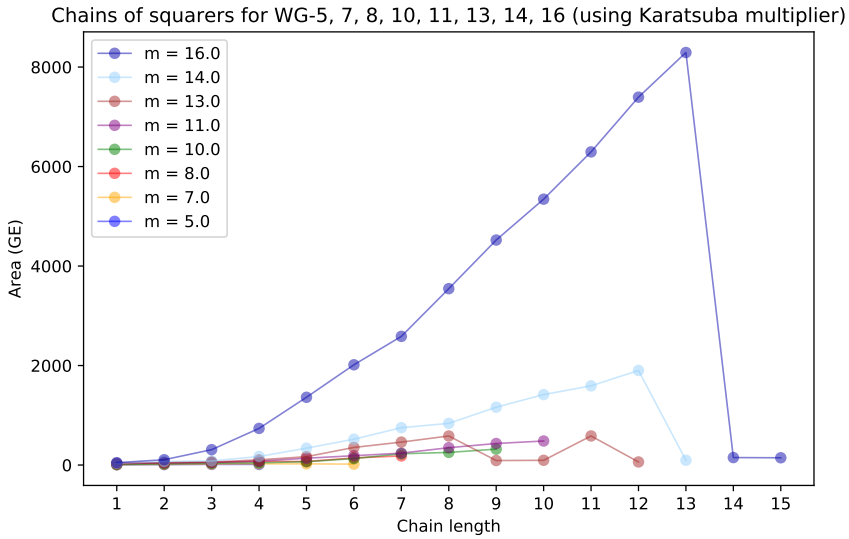


Figure 3.6: Area for chains of squarers using the Karatsuba multiplier (65 nm CMOS)

Such behaviour means that it will be impossible to simply substitute the classic multiplier by the Karatsuba multiplier in our design. In order to get consistent results for area profiling, it was decided to stick with the classic multiplier despite an area disadvantage when used as an actual multiplier. In order to incorporate the Karatsuba multiplier in large WG instances later on without massive area penalty, we still have to keep using the classic multipliers for the squarers. The implementation results for WG ciphers using the Karatsuba multipliers are presented in Table 5.3 on page 71.

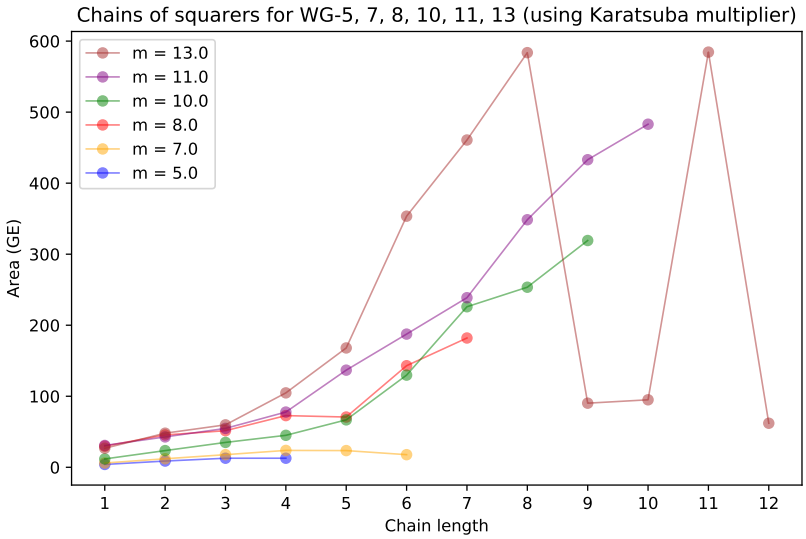


Figure 3.7: Area for chains of squarers using the Karatsuba mul., zoomed in (65 nm CMOS)

## 3.3 Exponentiation

Exponentiation $a^d$ in WG is used in a form of fixed exponent ($d$) as natural number and unknown base $a$ ($a \in GF_{2^m}$, where $m$ = field_sz). The exponent value is provided as a generic constant and the base is an m-bit input std_logic_vector. The output is the exponentiation result, also an m-bit std_logic_vector representing an element of the extension field $GF_{2^m}$. The exponentiation component is purely combinational. Entity description is in Table 3.4.

| **Exponentiation (exp_comb.vhd)** | | | |
|---|---|---|---|
| | **Kind** | **Type** | **Description** |
| **field_sz** | Generic | *integer* | dimension of binary extension field |
| **field_poly** | Generic | *std_logic_vector (field_sz downto 0)* | field defining polynomial |
| **d** | Genetic | *integer* | exponent value |
| **i_a** | Input | *std_logic_vector (0 to field_sz - 1)* | exponent base |
| **o_z** | Output | *std_logic_vector (0 to field_sz - 1)* | exponentiation result |

Table 3.4: Parameters for the exponentiation entity

Hardware components for exponentiation with the fixed exponent are built using lower level building blocks – multipliers and squarers. An algorithm that arranges and connects these blocks one-by-one in a chain is written as a VHDL function making the file self-contained – it does not depend on any external software or tool. Two algorithms were implemented and compared for the number of multipliers both theoretically and empirically.

Algorithm 1 is a straightforward square-and-multiply approach. It is documented in [26] and requires $||d|| - 1$ squaring operations and $H(d) - 1$ multiplications, where $d_2$ is an exponent value in binary representation, $||d||$ is the number of bits that are needed to represent $d$ in binary form and $H(d)$ is Hamming weight.

Algorithm 1: Let $A$ be an input, $Temp$ an intermediate value and $d_2$ an exponent value encoded in the binary form with its MSB = 1 (no leading zeros). Let $Temp = A$. In each iteration Algorithm 1 processes one bit of the exponent starting from the second most significant bit. If the bit is 0, then $Temp = Temp^2$, if the bit = 1, then $Temp = A \cdot Temp^2$.

Let's look at an example. $A^{203}$ is a decimation block for WG-11, which uses the smallest field size where discrete components have an area advantage over the constant array implementations.

$A^{203}$ example for Algorithm 1:

| | |
|---|---|
| $d_{10}$ | $= 203$ |
| $d_2$ | $= 11001011$ |
| # of multipliers: $H(d_2)$ | $= 4$ |
| # of squarers: | $= 7$ |

| | | | |
|---|---|---|---|
| $A \cdot A^2$ | $= A^3$ | Iter. 1 ($bit_7 = 1$) | 1 multiplier, 1 squarer |
| $(A \cdot A^2)^2$ | $= A^6$ | Iter. 2 ($bit_6 = 0$) | 1 squarer |
| $((A \cdot A^2)^2)^2$ | $= A^{12}$ | Iter. 3 ($bit_5 = 0$) | 1 squarer |
| $A \cdot (((A \cdot A^2)^2)^2)^2$ | $= A^{25}$ | Iter. 4 ($bit_4 = 1$) | 1 multiplier, 1 squarer |
| $(A \cdot (((A \cdot A^2)^2)^2)^2)^2$ | $= A^{50}$ | Iter. 5 ($bit_3 = 0$) | 1 squarer |
| $A \cdot ((A \cdot (((A \cdot A^2)^2)^2)^2)^2)^2$ | $= A^{101}$ | Iter. 6 ($bit_2 = 1$) | 1 multiplier, 1 squarer |
| $A \cdot (A \cdot ((A \cdot (((A \cdot A^2)^2)^2)^2)^2)^2)^2$ | $= A^{203}$ | Iter. 7 ($bit_1 = 1$) | 1 multiplier, 1 squarer |

Result: $A^{203} = A \cdot (A \cdot ((A \cdot (((A \cdot A^2)^2)^2)^2)^2)^2)^2$, 4 multipliers, 7 squarers Hardware built by the algorithm is shown in Figure 3.8.

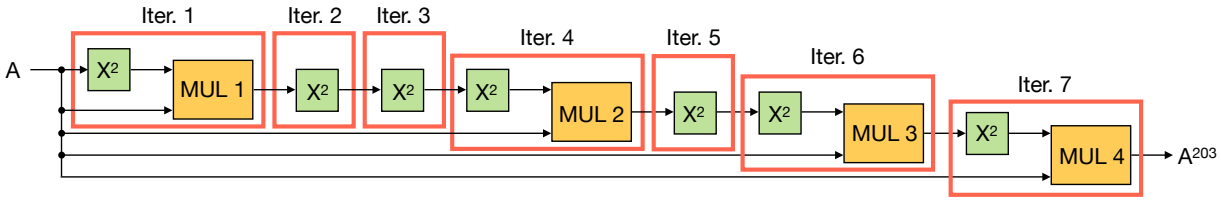


Figure 3.8: Decimation block hardware for WG-11 using Algorithm 1, d = 203

Algorithm 2 is based on a proposal by Hasan in [27] called "Efficient S&M based Inversion (using NB)". While the original algorithm by Hasan is given only for inversion in normal basis, it can be merged with Algorithm 1 and applied to general exponentiation in polynomial basis with great results. Algorithm 2: Let $A$ be an input, $d$ an exponent value and $Temp$ an intermediate exponent value. Let $Temp = d$ . Each iteration Algorithm looks at the value of $Temp$ and does one of the following steps:

- if $Temp = 2^{2 \cdot n} - 1$, it performs a substitution $A^{2^{2 \cdot n}-1} = A^{2^n-1} \cdot (A^{2^n-1})^{2^n}$, which is a hardware block that takes $A^{2^n-1}$ as input, squares it $n$ times and then multiplies the result with $A^{2^n-1}$. New $Temp = 2^n - 1$. This step gives advantage over Algorithm 1 due to more efficient decomposition of the exponent with fewer multipliers.
- if $Temp = odd$, but simultaneously $Temp \neq 2^{2 \cdot n} - 1$, the algorithm performs a substitution $A^{Temp} = A \cdot (A^{0.5 \cdot (Temp-1)})^2$, which is a hardware block that takes $A^{0.5 \cdot (Temp-1)}$ and $A$ as inputs, squares the first input once and multiplies it by $A$. New $Temp = 0.5 \cdot (Temp - 1)$. This step is essentially a part of regular square and multiply Algorithm 1.
- if $Temp = even$, the algorithm just adds a square block. New $Temp = 0.5 \cdot Temp$. This is also step taken directly from Algorithm 1.

$A^{203}$ example for Algorithm 2:

| | | |
|---|---|---|
| $A^{203} = A \cdot A^{202} = A \cdot (A^{101})^2$ | Iter. 1 (203 is odd) | 1 multiplier, 1 squarer |
| $A^{101} = A \cdot (A^{50})^2$ | Iter. 2 (101 is odd) | 1 multiplier, 1 squarer |
| $A^{50} = (A^{25})^2$ | Iter. 3 (50 is even) | 1 squarer |
| $A^{25} = A \cdot (A^{12})^2$ | Iter. 4 (25 is odd) | 1 multiplier, 1 squarer |
| $A^{12} = (A^6)^2$ | Iter. 5 (12 is even) | 1 squarer |
| $A^6 = (A^3)^2$ | Iter. 6 (6 is even) | 1 squarer |
| $A^3 = A \cdot A^2$ | Iter. 7 (3 is odd) | 1 multiplier, 1 squarer |

Result: $A^{203} = A \cdot (A \cdot ((A \cdot (((A \cdot A^2)^2)^2)^2)^2)^2)^2$, 4 multipliers, 7 squarers. The corresponding hardware is shown in Figure 3.9.

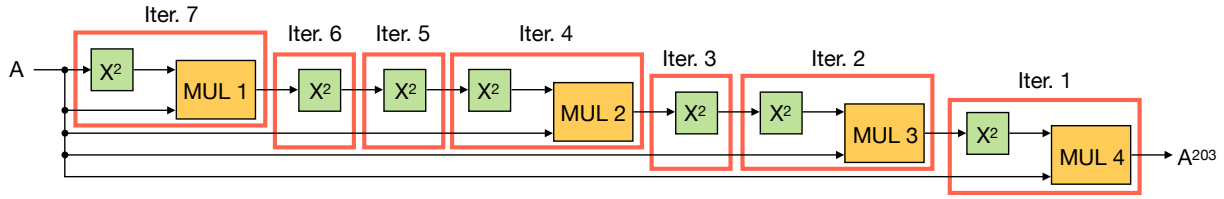


Figure 3.9: Decimation block hardware for WG-11 using Algorithm 2, d = 203

As we can see from Figure 3.8 and Figure 3.9, both algorithms provide exactly the same result. The only difference is that Algorithm 1 goes from MSB down to LBS and Algorithm 2 goes from LSB to MSB (in a form of checking whether the current intermediate exponent is even or odd).

Algorithm 2 allows to reduce the number of multipliers from $m-1$ to $\lfloor log_2(m-1) \rfloor + H(m-1) - 1$ [27] when used for exponent $d = 2^m - 2$ that corresponds to inversion according to Fermat's Little Theorem ($A^{-1} = A^{2^m-2}$, where $m$ = field_sz). Squaring operation is essentially free in the normal basis (rotation) but requires several XOR gates in the polynomial basis. But it is still much cheaper than multiplication, and therefore an algorithm that minimizes the number of multiplications is equally useful for the polynomial basis exponentiation as it is for the normal basis exponentiation.

However, it is important to show that Algorithm 2 sometimes requires fewer multiplications than Algorithm 1 (when exponent values $d = 2^m - 2$ occur at any decomposition step) and never requires more multiplications than Algorithm 1. Figure 3.10 shows actual exponentiation block area after logic synthesis for different exponent values in WG-11 field using the polynomial basis with the field defining polynomial $f(x) = x^{11} + x^2 + 1$. Each dot is an exponent (range from 1 to 2046), with X coordinate being area using Algorithm 2 and Y coordinate being the area using Algorithm 1. As we can see, the Algorithm 2 is always at least as efficient as the Algorithm 1. Every dot that lies above the Y = X line corresponds to an exponent for which Algorithm 2 gives noticeable area advantage.

Exponents in the Figure 3.10 are clustered in multiple groups with equal gaps between them. These gaps correspond to addition of 1 extra multiplier. Exponents within one group have the same number of multipliers but different number of squarers.

In order to take advantage of the Algorithm 2, a term in form of $A^{2^{2 \cdot n}-1}$ must occur somewhere during its execution. It allows for efficient substitution with only one multiplier:

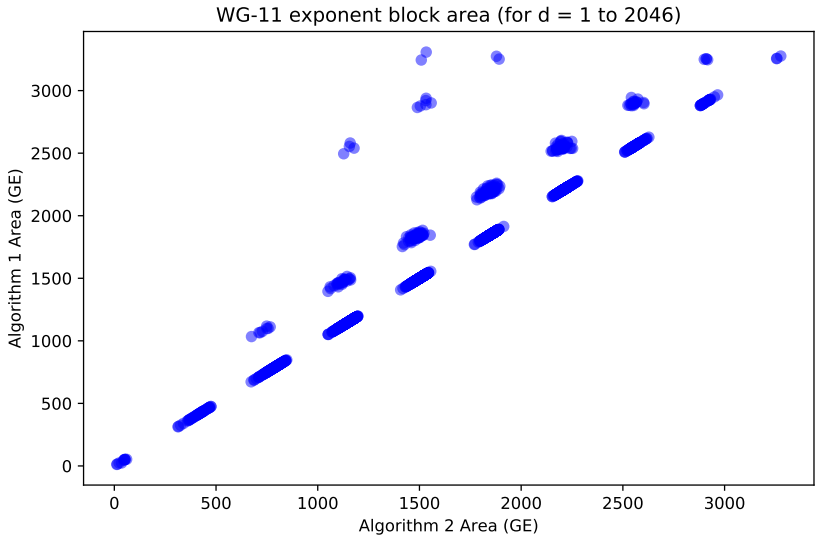


Figure 3.10: Area comparison between Algorithm 1 and 2 for exponentiation, WG-11

$$\begin{aligned} A^{2^{2\cdot n}-1} &= A^{(2^n-1)\cdot(2^n+1)} \\ &= (A^{2^n-1})^{2^n} \cdot A^{2^n-1} \end{aligned}$$

compared to decomposition of the same term by Algorithm 1 using $n$ multipliers:

$$\begin{aligned} A^{2^{2\cdot n}-1} &= A \cdot (A^{2^{2\cdot n}-2}) \\ &= A \cdot (A^{2^{2\cdot n-1}-1})^2 \\ &= A \cdot (A \cdot A^{2^{2\cdot n-1}-2})^2 \\ &= A \cdot (A \cdot (A^{2^{2\cdot n-2}-1})^2)^2 \\ &= A \cdot (A \cdot (A \cdot A^{2^{2\cdot n-2}-2})^2)^2 \\ &= A \cdot (A \cdot (A \cdot (A^{2^{2\cdot n-3}-1})^2)^2)^2 \\ &= \dots \end{aligned}$$

Now let's consider an example that fully takes advantage of the Algorithm 2. For that it is a good idea to choose an inversion.

$A^{-1} = A^{2^{11}-2} = A^{2046}$ (inversion in $GF_{2^{11}}$) example for Algorithm 1:

$d_{10} = 2046$
$d_2 = 11111111110$
# of multipliers: $H(d_2) = 9$
# of squarers: 10

| | | | |
|---|---|---|---|
| $A \cdot A^2$ | Iter. 1 | $(bit_{10} = 1)$ | 1 mult., 1 sq. |
| $A \cdot (A \cdot A^2)^2$ | Iter. 2 | $(bit_9 = 1)$ | 1 mult., 1 sq. |
| $A \cdot (A \cdot (A \cdot A^2)^2)^2$ | Iter. 3 | $(bit_8 = 1)$ | 1 mult., 1 sq. |
| $A \cdot (A \cdot (A \cdot (A \cdot A^2)^2)^2)^2$ | Iter. 4 | $(bit_7 = 1)$ | 1 mult., 1 sq. |
| $A \cdot (A \cdot (A \cdot (A \cdot (A \cdot A^2)^2)^2)^2)^2$ | Iter. 5 | $(bit_6 = 1)$ | 1 mult., 1 sq. |
| $A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot A^2)^2)^2)^2)^2)^2$ | Iter. 6 | $(bit_5 = 1)$ | 1 mult., 1 sq. |
| $A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot A^2)^2)^2)^2)^2)^2)^2$ | Iter. 7 | $(bit_4 = 1)$ | 1 mult., 1 sq. |
| $A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot A^2)^2)^2)^2)^2)^2)^2)^2$ | Iter. 8 | $(bit_3 = 1)$ | 1 mult., 1 sq. |
| $A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot A^2)^2)^2)^2)^2)^2)^2)^2)^2$ | Iter. 9 | $(bit_2 = 1)$ | 1 mult., 1 sq. |
| $(A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot A^2)^2)^2)^2)^2)^2)^2)^2)^2)^2$ | Iter. 10 | $(bit_1 = 1)$ | 1 mult., 1 sq. |

Result: $A^{2046} = (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot (A \cdot A^2)^2)^2)^2)^2)^2)^2)^2)^2)^2$, 9 multipliers, 10 squarers. Hardware for this example is shown in Figure 3.11.

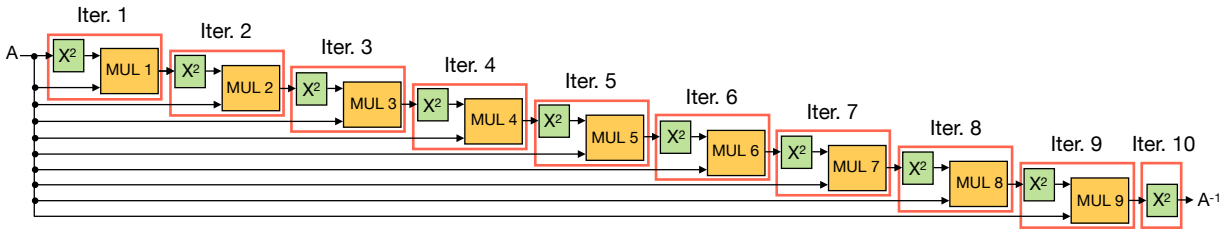


Figure 3.11: Inversion block hardware for WG-11 using Algorithm 1, d = 2046

Here is the same $A^{-1} = A^{2^{11}-2} = A^{2046}$ (inversion in $GF_{2^{11}}$) example for Algorithm 2:

| | | |
|---|---|---|
| $A^{2^{11}-2} = (A^{2^{10}-1})^2 = (A^{(2^5-1)\cdot(2^5+1)})^2 = ((A^{2^5-1})^{2^5} \cdot A^{2^5-1})^2$ | Iter. 1 | 1 mult., 5 sq. |
| $A^{2^5-1} = A \cdot A^{2^5-2} = A \cdot (A^{2^4-1})^2$ | Iter. 2 | 1 mult., 1 sq. |
| $A^{2^4-1} = A^{(2^2-1)\cdot(2^2+1)} = (A^3)^{2^2} \cdot A^3$ | Iter. 3 | 1 mult., 2 sq. |
| $A^3 = A \cdot A^2$ | Iter. 4 | 1 mult., 1 sq. |

Result: $A^{2^{11}-2} = ((A \cdot ((A \cdot A^2)^{2^2} \cdot (A \cdot A^2))^2)^{2^5} \cdot A \cdot ((A \cdot A^2)^{2^2} \cdot (A \cdot A^2))^2)^2$, 4 multipliers, 9 squarers. Hardware for this example is shown in Figure 3.12.

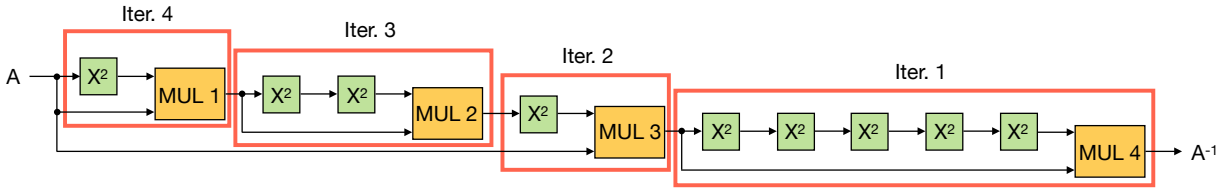


Figure 3.12: Inversion block hardware for WG-11 using Algorithm 2, d = 2046

For this exponent value the Algorithm 2 was able to reduce the number of multipliers from 9 to 4 and the number of squarers from 10 to 9. Overall, implementation of the Algorithm 2 allows the use of the same generic exponentiation block everywhere in design without worrying about special cases like inversion.

## 3.4 Trace

Absolute trace in a form of:

$$Tr_1^m(\alpha) = \sum_{i=0}^{m-1} \alpha^{2^i} = \alpha + \alpha^2 + \alpha^{2^2} + \cdots + \alpha^{2^{m-1}}$$

is implemented directly in VHDL. After the first term $\alpha$, each term in the equation above is a chain of squarers with length from 1 to $m-1$. It maps an *input* of length $m$ bits to an *output* of length 1 bit. The trace equation above simplifies down to a few (from 1 to $m-1$) bitwise XOR operations depending on the field defining polynomial $f(x)$.

However, it was observed that *Design Compiler* was unable to fully simplify the trace circuit for WG-13 and above, despite giving correct functionality. This could happen due to the high complexity of the circuit, which makes it possible for a tool to get stuck in a local minimum while trying to simplify the equation. In order to resolve this problem, Zidaric computed the final trace equation in GAP. That equation was implemented directly in VHDL for each field and field defining polynomial. This is not an ideal, but a necessary workaround until the solution to the problem is found. A comparison of two different trace implementations for WG-13, 14 and 16 is shown in Table 3.5.

Penalty for complete WG ciphers is discussed in Section 5.2.1 on page 70. Parameters for the trace entity are shown in Table 3.6.

| | Trace area | |
|---|---|---|
| | **GAP-derived equation** | **Chains of squarers** |
| **WG-13** | 13 GE | 60 GE |
| **WG-14** | 13 GE | 658 GE |
| **WG-16** | 2 GE | 247 GE |

Table 3.5: Trace area for 2 different approaches (65 nm CMOS)

| **Trace (trace_comb.vhd)** | | | |
|---|---|---|---|
| | **Kind** | **Type** | **Description** |
| **field_sz** | Generic | *integer* | dimension of binary extension field |
| **field_poly** | Generic | *std_logic_vector (field_sz downto 0)* | field defining polynomial |
| **i_a** | Input | *std_logic_vector (0 to field_sz - 1)* | Input |
| **o_z** | Output | *std_logic* | Absolute trace |

Table 3.6: Parameters for the trace entity

## 3.5 DWGP and DWGT

DWGP is a complex block that combines decimation and WG permutation. Figure 3.13 shows the DWGT (DWGP + trace) hardware. WGP is split into two separate blocks - simple and compose - for the sake of code clarity and consistency with other implementations. Intermediate DWGP output is required for the cipher initialization mode.

| **DWGP block (dwgp.vhd, dwgp-comp.vhd)** | | | |
|---|---|---|---|
| | **Kind** | **Type** | **Description** |
| **d_exp** | Generic | *integer* | decimation exponent value |
| **i_x** | Input | *std_logic_vector (0 to field_sz - 1)* | input signal |
| **o_wgp** | Output | *std_logic_vector (0 to field_sz - 1)* | output signal |

Table 3.7: Parameters for the DWGP entity

Tables 3.7 and 3.8 show parameters for the DWGP and DWGT entities respectively. Entity and architecture declarations are in separate files because two architectures are used within the project (-comp, or implementation using discrete components, and -const_array, or implementation using constant array).

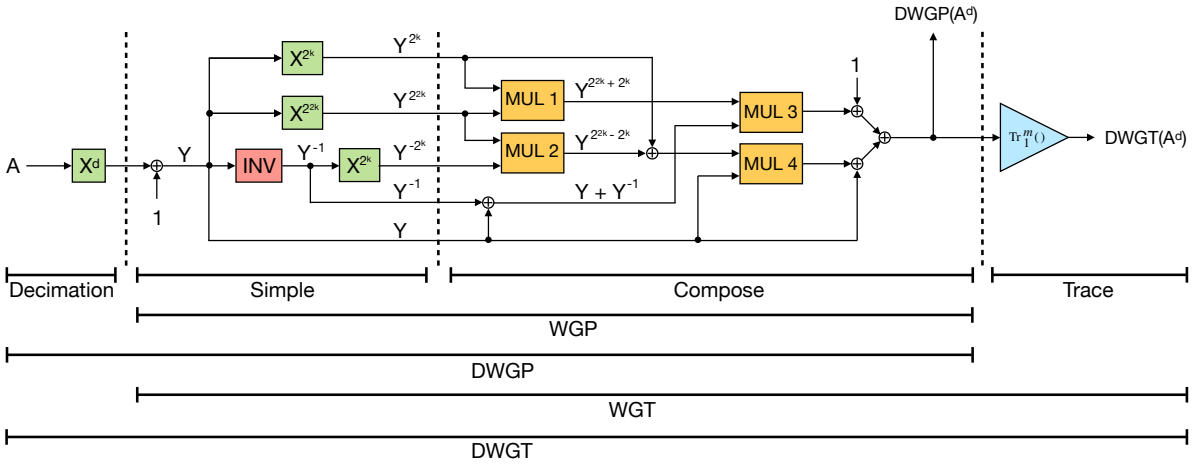


Figure 3.13: DWGP / DWGT hardware

| DWGT block (dwgt.vhd, dwgt-comp.vhd) | | | |
|---|---|---|---|
| | **Kind** | **Type** | **Description** |
| **d_exp** | Generic | *integer* | decimation exponent value |
| **i_x** | Input | *std_logic_vector (0 to field_sz - 1)* | input signal |
| **o_wgt** | Output | *std_logic* | output signal (1-bit keystream) |

Table 3.8: Parameters for the DWGT entity

Discrete components implementation instantiates all individual building blocks from the files included in project. This allows, for example, swap one multiplier architecture for another without making any changed in the code. Inclusion of such trivial instances as *add_one*, which adds a constant 1 to the signals, is necessary to support other bases – like normal basis - in the future. In polynomial basis, addition of 1 in terms of hardware is an inversion of the least significant bit of the input signal, while in normal basis it is an inversion of all bits.

INV component, that is shown as a red rectangle in Figure 3.13, is the inversion, i.e. exponentiation to the power of $2^m - 2$.

# Chapter 4

# DWGP Area Results & Analysis

In this chapter we evaluate area for DWGP / DWGT for implementations using discrete components and constant array (see Section 2.3.3 for more details). There is a difference between these two approaches in circuit area and how it changes based on field size and field defining polynomial. An in-depth study of these implementations is presented in sections 4.1 and 4.2 and more than 26000 DWGP instances were synthesized.

Changing the field defining polynomial affects the DWGP area. That means it is important to find the polynomial that gives the smallest DWGP area and ultimately the smallest WG cipher area. One way to find the polynomial is to do the exhaustive search by synthesizing the DWGP for all the polynomials that exist for a given field size. This approach is reasonable with the ciphers that use smaller fields like WG-5 or WG-10. As field size increases, the number of polynomials increases and the synthesis time of each DWGP increases as well. For example, it takes almost 2 months of compute time to synthesize DWGP for all 2048 primitive polynomials that exist for $GF_{2^{16}}$ on a typical workstation PC and it is impractical for larger fields. Alternatively, it might be possible to find a metric that correlates with area.

In [28], Zidaric proposed to use the Hamming weights of field defining polynomials, derived matrices, and pre- and post-PAR synthesis results for profiling of various arithmetic blocks. In the WG-lite project, this idea was extended to compare simple metrics with the DWGP area. The metrics used are the Hamming weight of field defining polynomial, Hamming weight of reduction matrix, reduction matrix area (before place-and-route) and multiplier area (before place-and-route). The first two can be quickly precomputed, while the latter two require running synthesis tools. Using such a metric would allow us either pinpoint the best polynomial or to reduce the search space. This thesis provides in-depth

correlation analysis between the aforementioned metrics and the DWGP: we show that reduction matrix Hamming weight is indeed a good metric to reduce search space for the smallest area polynomial. Most examples use WG-14 to better illustrate the behaviour due to a high number of primitive field defining polynomials available (756). And the reason we chose WG-14 over WG-16 is consistency – DWGP for WG-16 implemented with discrete components does not fit on the MAX 10 FPGA.

Area analysis in these sections is done with synthesis results pre place-and-route for 65 nm CMOS ASIC library, STRATIX IV FPGA and MAX 10 FPGA.

## 4.1 DWGP Implementations Using Discrete Components

This section is dedicated to area analysis of the DWGP implemented using discrete components. We discovered that this approach results in high area variability, and finding the polynomial that corresponds to the smallest DWGP provides a noticeable area decrease of more than 10% w.r.t. the mean area for most field sizes. We will evaluate each implementation technology separately, and then compare the results for ASIC and FPGAs.

### 4.1.1 65 nm CMOS ASIC library

Figure 4.1 shows correlation between DWGP area and reduction matrix area for each field defining polynomial. Similarly, Figure 4.2 shows correlation between DWGP area multiplier area for WG-14. The smallest reduction matrix and the smallest multiplier both correspond to the smallest DWGP. This behaviour is consistent across different libraries and from WG-5 to WG-16. It is reasonable to assume that the same can be said about ciphers using larger fields.

There is no significant difference between using either reduction matrix area or multiplier area to find the smallest DWGP area, aside from the fact that reduction matrix takes less time to synthesize. However, both approaches share the same disadvantage. They require exhaustive search through all primitive polynomials. Much faster synthesis times compared to DWGP can make it practical for a few more larger fields (ciphers above WG-16), but the process still should not scale well. That is why it is important to have an alternative metric that does not require running the synthesis tools.

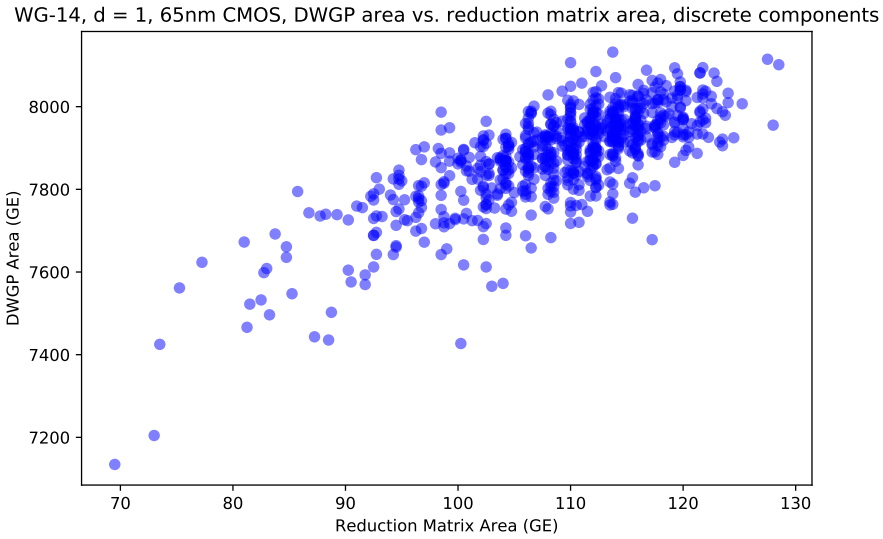


Figure 4.1: DWGP vs. red. matrix area (WG-14, 65 nm CMOS, d = 1, discrete comp.)

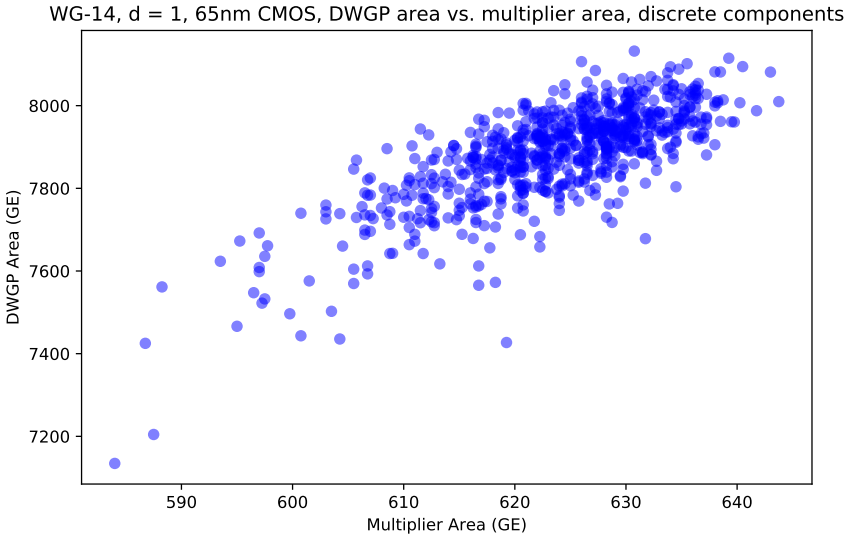


Figure 4.2: DWGP area vs. multiplier area (WG-14, 65 nm CMOS, d = 1, discrete comp.)

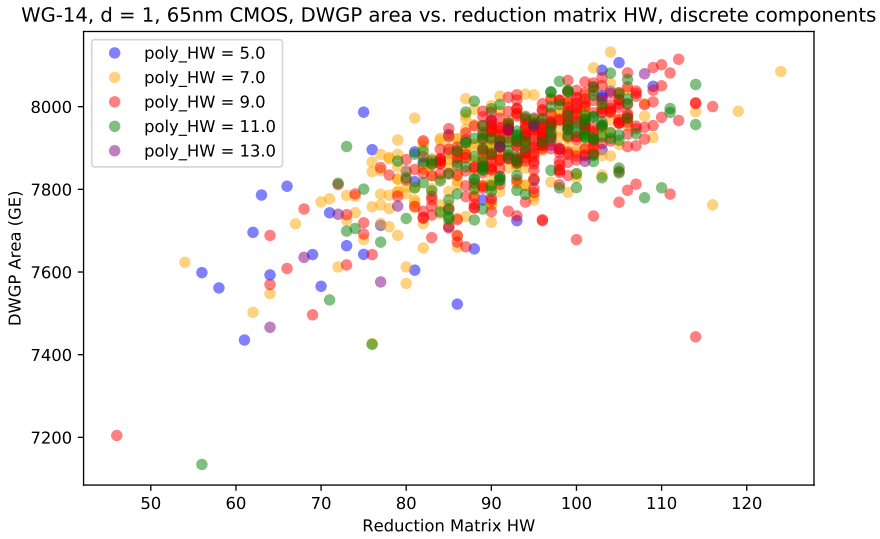


Figure 4.3: DWGP area vs. Hamming weights (WG-14, 65 nm CMOS, d = 1, disc. comp.)

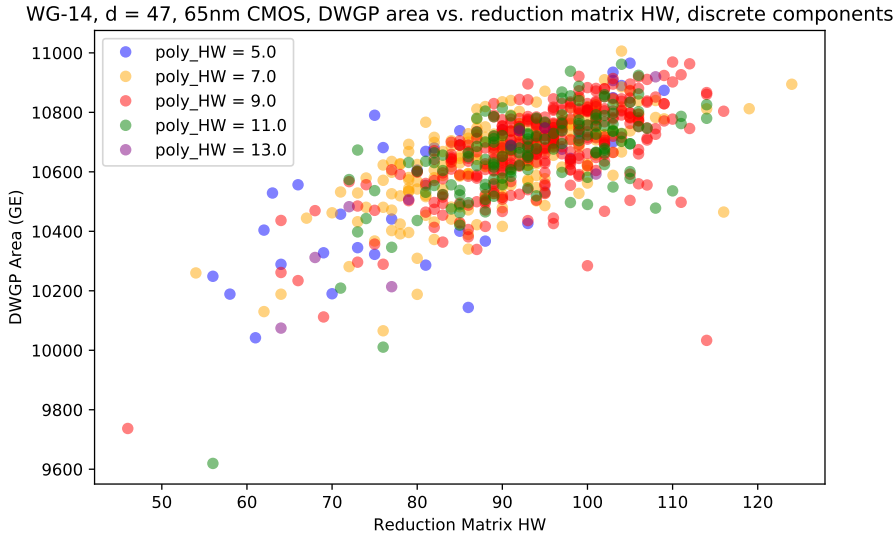


Figure 4.4: DWGP area vs. Hamming weights (WG-14, 65 nm CMOS, d = 47, disc. comp.)

In Figure 4.3 we can see correlation between DWGP size and Hamming weights of both field defining polynomial and reduction matrix. Like for two previous figures, all 756 primitive field defining polynomials for $GF_{2^{14}}$ are shown.

The smallest DWGP area corresponds to the field defining polynomial with Hamming weight = 11, and the second and third correspond to the ones with Hamming weight of 9 and 11 respectively. Only the 5th smallest DWGP corresponds to a polynomial with minimal Hamming weight of 5, with area increase of 4% compared to the smallest DWGP. Choosing to explore only polynomials with the lowest Hamming weight reduces the search space from 756 to 42 polynomials.

Reduction matrix Hamming weight, on the other hand, correlates better with DWGP area. If we evaluate field defining polynomials one by one, starting with the one with the smallest reduction matrix Hamming weight, and go in ascending order, it will take only 3 attempts to find the polynomial that corresponds to the smallest DWGP area for WG-14. The number of attempts for other field sizes can be found in Figure 4.20 on page 47.

Figure 4.4 shows the same data for the case of decimation exponent = 47 (d = 47). As can be inferred from the chart, introduction of decimation hardware noticeably increases DWGP area, however the relation between different polynomials does not change a lot. This consistency allows us to proceed with area analysis of only DWGP without decimation, and it greatly reduces the number of instances needed to be synthesized, making a more in-depth approach feasible.

Figure 4.5 shows an effect of decimation on DWGP area with discrete components implementation more clearly. The closer the plot is to a straight line, the more consistent is the effect of decimation. Also, we are mostly interested in the lower left quadrant, where DWGP instances with the smallest area are located.

DWGP area distributions in percentage above the minimal area for all field dimensions are shown in Figure 4.6 (without decimation) and Figure 4.7 (with decimation). Decimation values used are shown in Table 4.1 on page 56. As can be seen from the plots, there is a significant difference from 10% to more than 15% between the absolute minimum DWGP area and the mean value, or up to 10 standards deviations. That means there exist a few polynomials that provide a significant area decrease compared to the rest of them - they correspond to the sharp spikes down in distribution plots. It makes searching for such polynomials worthwhile.

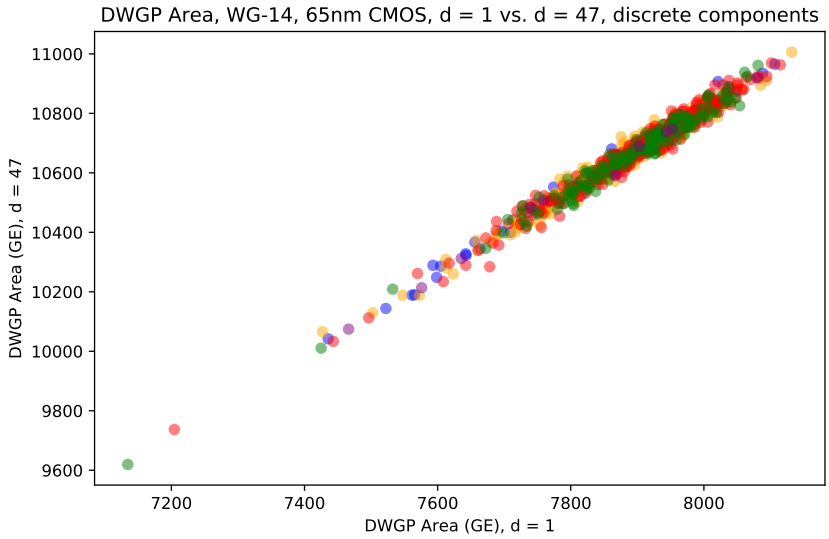


Figure 4.5: Decimation effect on DWGP area (WG-14, 65 nm CMOS, discrete components)

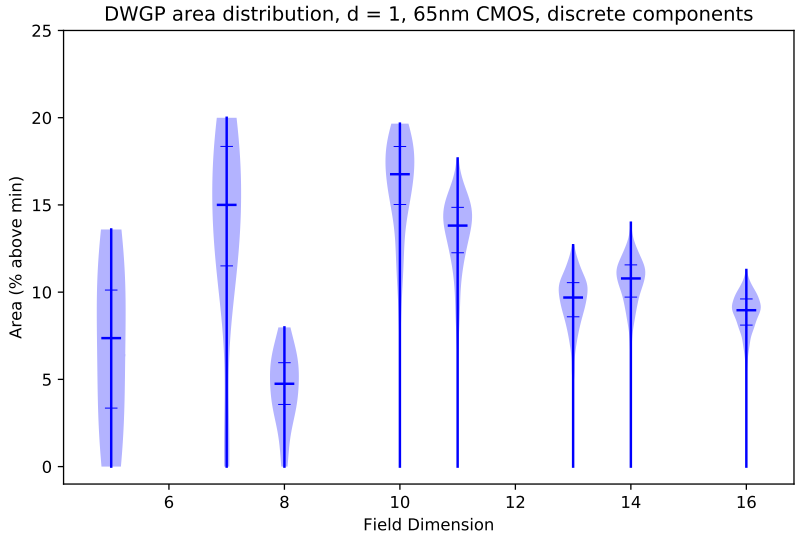


Figure 4.6: DWGP area distribution (d = 1, 65 nm CMOS, discrete components)

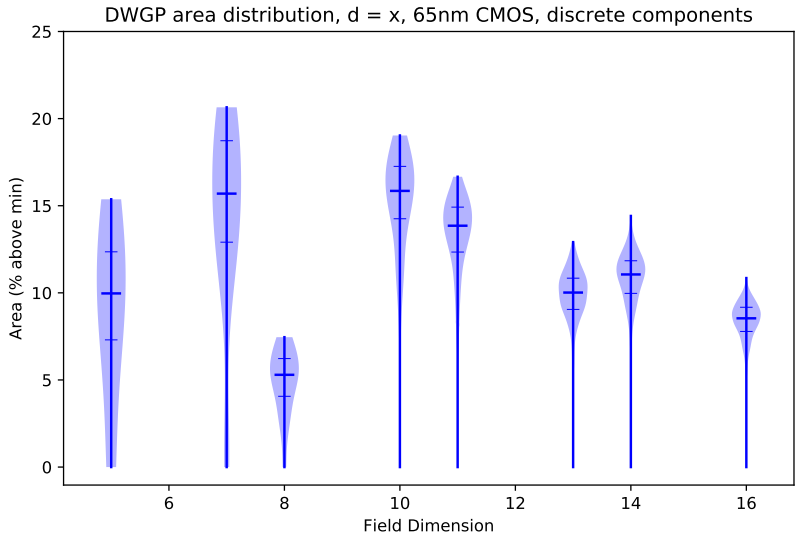


Figure 4.7: DWGP area distribution (d = x, 65 nm CMOS, discrete components)

### 4.1.2 STRATIX IV FPGA

Let's look at the same results for STRATIX IV FPGA. Depending on the axis scale, it is possible to observe more discrete nature of plots, as the smallest area unit is 1 look-up table (LUT) and is atomic.

For STRATIX IV FPGA, the correlation between DWGP area and Reduction Matrix area (Figure 4.8), Multiplier Area (Figure 4.9) and Reduction Matrix Hamming weight (Figure 4.10) is much less pronounced compared to 65 nm CMOS. The first two figures have noticeable discretization on the horizontal axis, which makes it difficult to perceive.

In Figure 4.10 we can observe that, unlike for 65 nm CMOS in Figure 4.3, most polynomials are grouped at the bottom. In all STRATIX IV charts in this section there are several outlying data points - surprisingly, the polynomials that have the smallest reduction matrix Hamming weight often provide very large area from 20% to 40% above the top threshold, below which the majority ($> 90\%$) of data points are located.

The impact of decimation on DWGP area for STRATIX IV, shown in Figure 4.11, is similar to what we observed for 65 nm CMOS in Figure 4.5 - in fact, the correlation is even stronger for STRATIX IV implementations.

Figures 4.12 and 4.13 show DWGP area distributions in percentage above the minimal

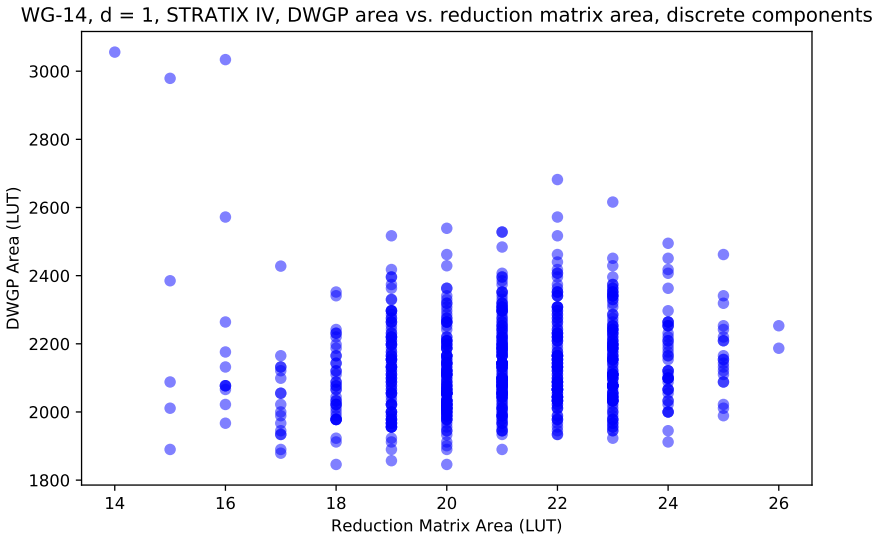


Figure 4.8: DWGP vs. reduction matrix area (WG-14, STRATIX IV, d = 1, disc. comp.)

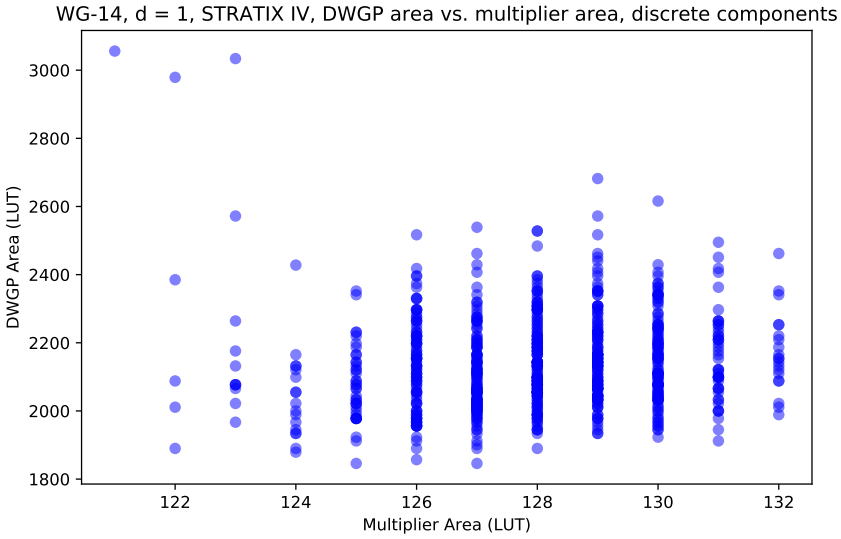


Figure 4.9: DWGP vs. multiplier area (WG-14, STRATIX IV, d = 1, discrete comp.)

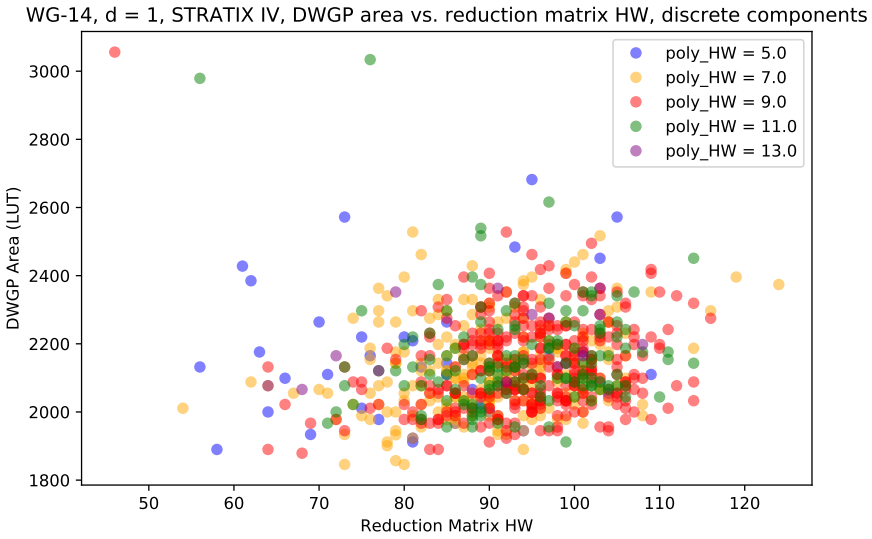


Figure 4.10: DWGP area vs. Hamming weights (WG-14, STRATIX IV, d = 1, d. comp.)

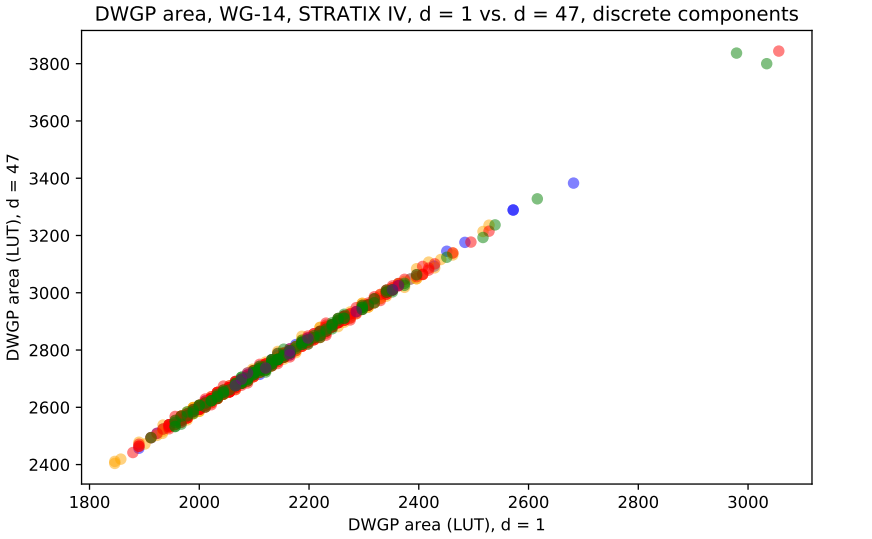


Figure 4.11: Decimation effect on DWGP area (WG-14, STRATIX IV, discrete comp.)

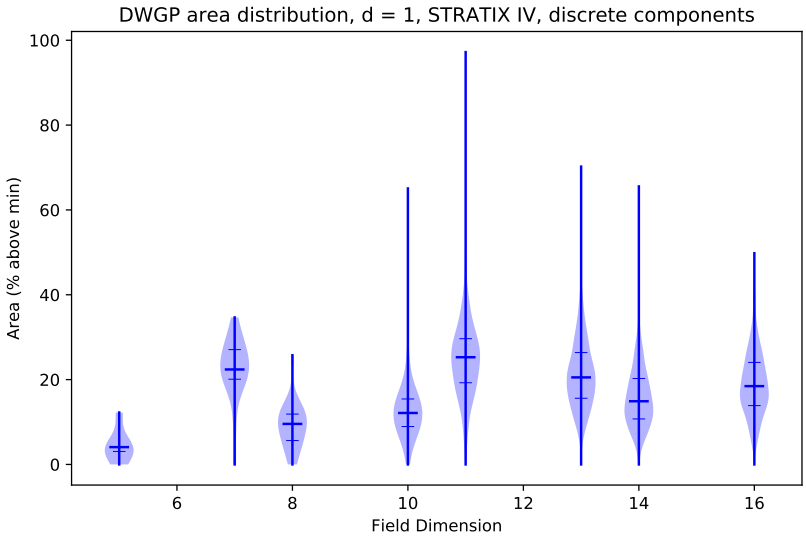


Figure 4.12: DWGP area distribution (d = 1, STRATIX IV, discrete components)

area without decimation and with decimation. Compared to discrete components implementations using 65 nm CMOS in the previous section, the distributions for STRATIX IV look the opposite. The majority of the polynomials provide small DWGP area within 2-3 standard deviations from the minimum. A few polynomials result in significant area increase and corresponds to spikes up in the plots. However, it does not mean that searching for the best polynomials is less important for STRATIX IV than it is for 65 nm CMOS. If we look at the percentage difference, we notice much higher area variability, and the mean area is often  20% higher than minimum area.

There is also a risk to end up with anomalously high area with the increase in the range from +65% to almost +100% for some field sizes if area profiling is not done correctly - as already stated in the previous paragraph, these polynomials often have the smallest reduction matrix Hamming weights. As this thesis is focused on 65 nm CMOS as a main technology, at least when it comes to implementations of complete WG cipher instances, we did not search for an explanation of this behaviour.

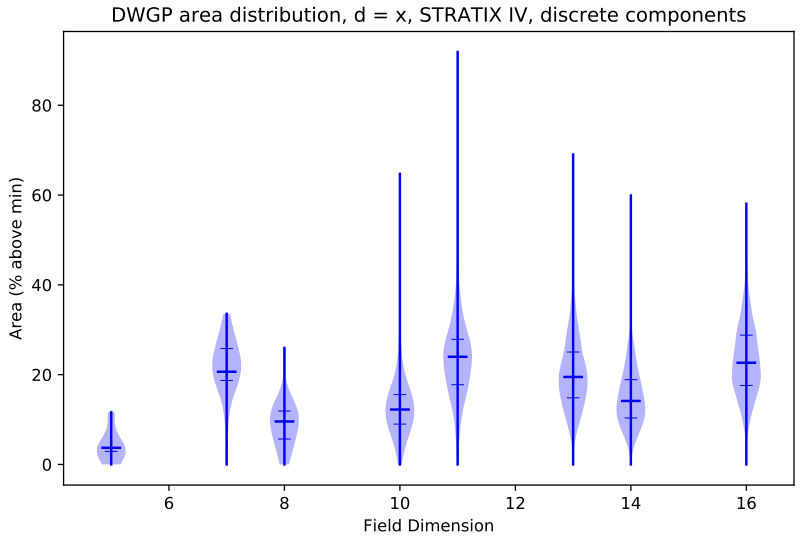


Figure 4.13: DWGP area distribution (d = x, STRATIX IV, discrete components)

### 4.1.3 MAX 10 FPGA

The MAX 10 board, however, shows more correlation for the same charts. All 3 charts have noticeable inclination from bottom left to top right.

Figure 4.14 and Figure 4.15 show area correlation between DWGP and reduction matrix and between DWGP and multiplier respectively. It is possible to observe the same outlying data points as in Figure 4.8 and Figure 4.9 for STRATIX IV. However, the same polynomial with the smallest reduction matrix Hamming weight that resulted in surprisingly large DWGP area for STRATIX IV (see Figure 4.10) gives DWGP area close to mean value for MAX 10 (see Figure 4.16).

The behaviour observed in decimation impact chart in Figure 4.20 is similar to STRATIX IV implementations.

Finally, Figures 4.18 and 4.19 show DWGP area distribution in percentage above the minimal area without decimation and with decimation. The overall picture seems to be similar to we saw in STRATIX IV results in Section 4.1.2.

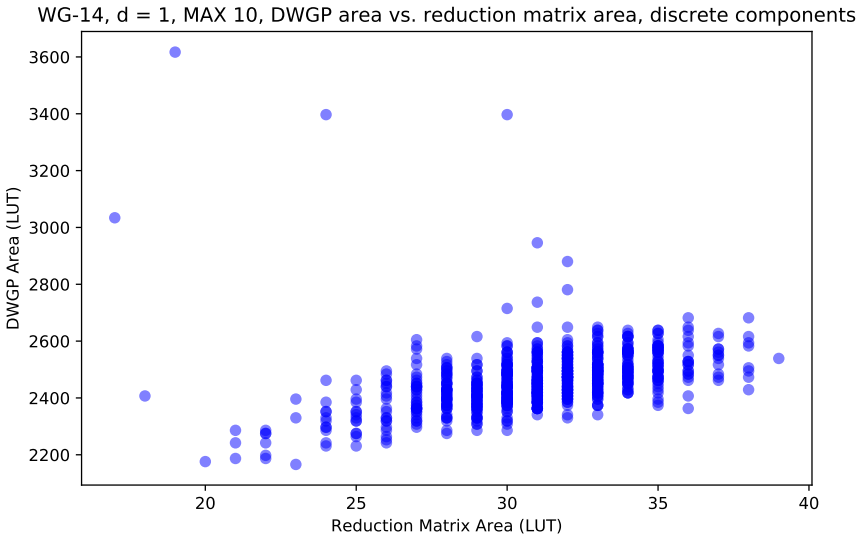


Figure 4.14: DWGP vs. reduction matrix area (WG-14, MAX 10, d = 1, discrete comp.)

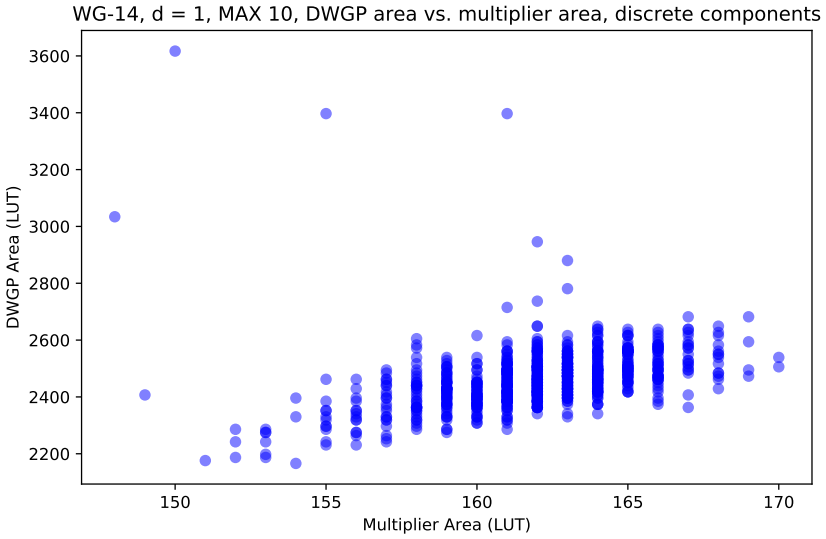


Figure 4.15: DWGP vs. multiplier area (WG-14, MAX 10, d = 1, discrete components)

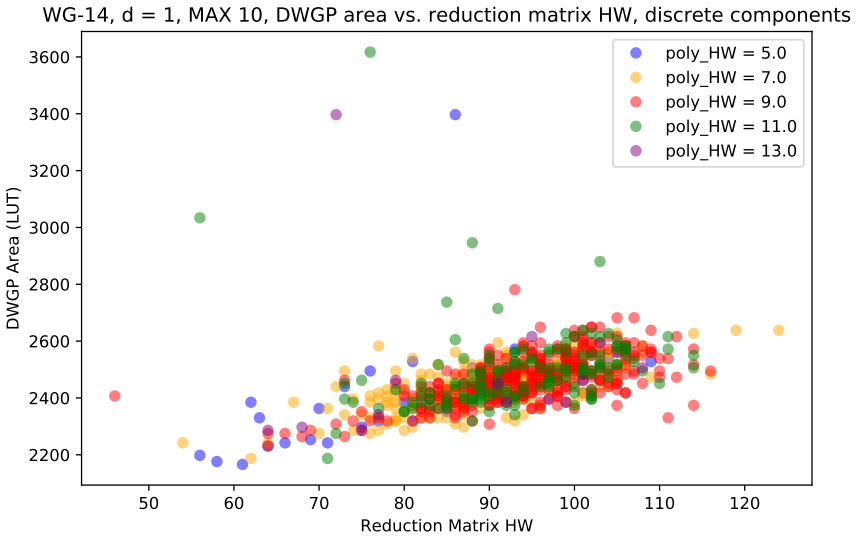


Figure 4.16: DWGP area vs. Hamming weights (WG-14, MAX 10, d = 1, discrete comp.)

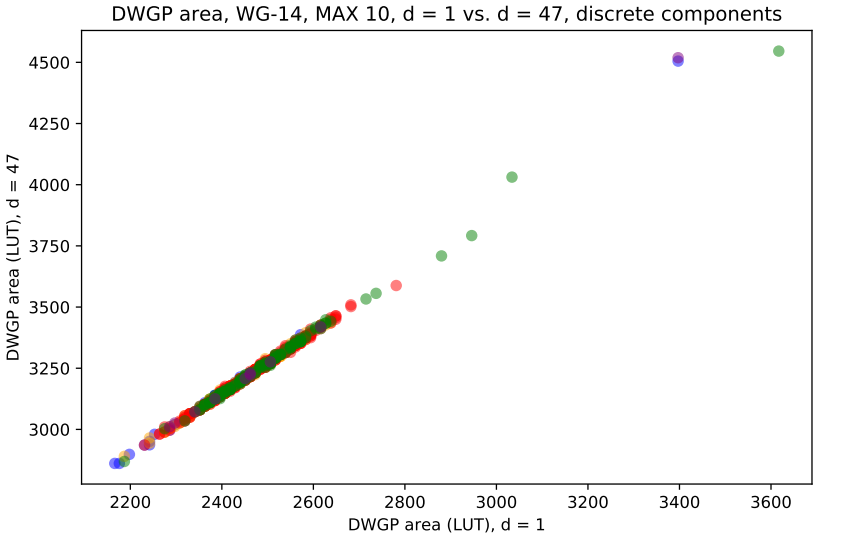


Figure 4.17: Decimation effect on DWGP area (WG-14, MAX 10, discrete components)

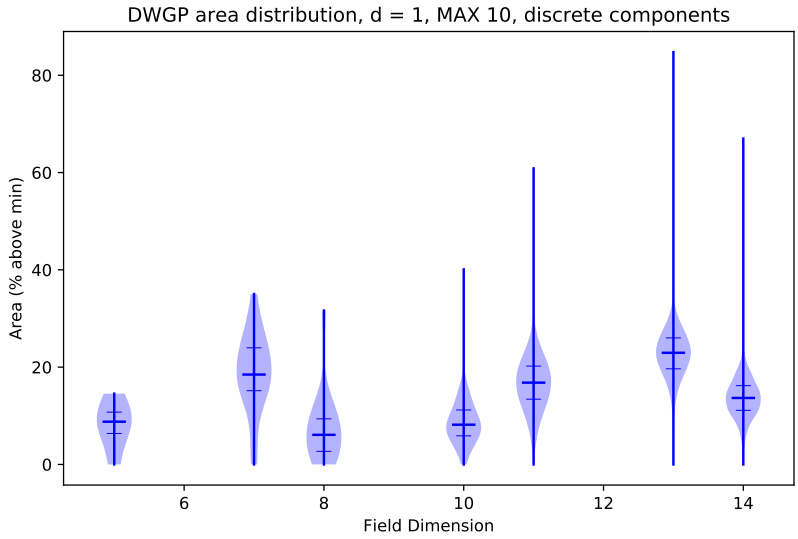


Figure 4.18: DWGP area distribution (d = 1, MAX 10, discrete components)

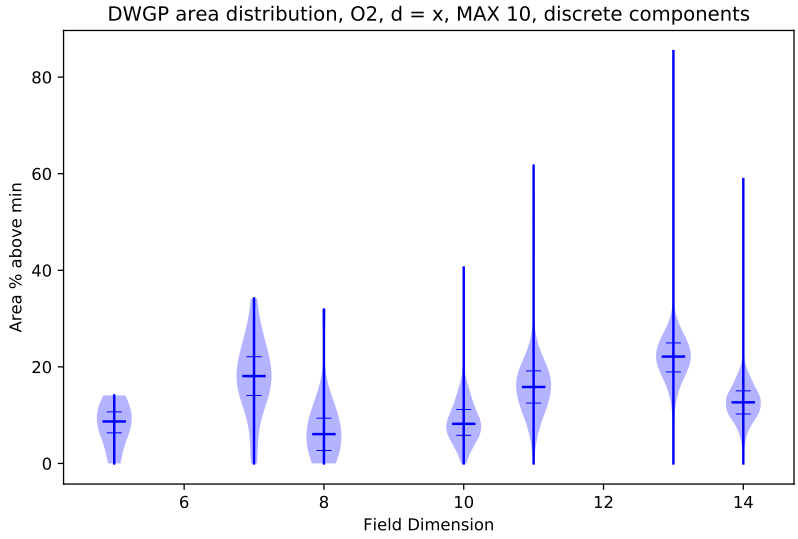


Figure 4.19: DWGP area distribution (d = x, MAX 10, discrete components)

### Brief Summary for Discrete Components Implementations

It is clear from the sections above that reduction matrix Hamming weight can be used as a good indicator to find the field defining polynomial that corresponds to the smallest DWGP area implemented with discrete components. We can take advantage of good correlation and use it to reduce the search space. If we sort all field defining polynomials by their reduction matrix Hamming weight, then go through the list in ascending order and synthesize DWGP for each of them, it will take us several attempts before we stumble upon the one that results in the smallest DWGP area. In this work, all field defining polynomials were evaluated, so it is possible to find an exact number of attempts required for each field dimension and each library. This number, when extrapolated to larger fields, may be used as a reference of how many polynomials should be included in exhaustive search. If too few attempts are made, the polynomial for smallest DWGP area will not be found. If too many attempts are made, the search time will increase.

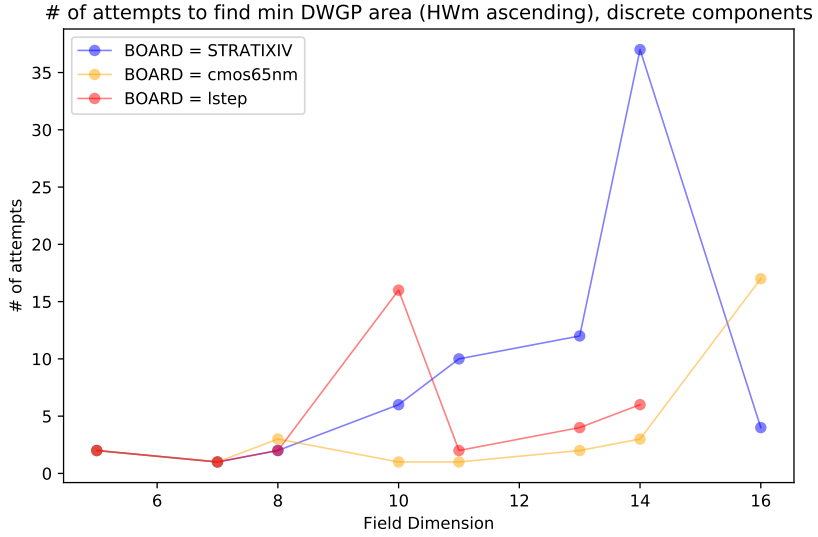


Figure 4.20: Number of attempts needed to find the smallest DWGP (d = 1, disc. comp.)

The number of attempts needed for different fields and different libraries/devices (65 nm CMOS, STRATIX IV and MAX 10) is shown in Figure 4.20. If we extrapolate the results to larger fields, where exhaustive search in the entire space of primitive polynomials is not feasible, we can conclude that it only takes to explore a fraction of it after precomputing the reduction matrix Hamming weight and sorting all the polynomials based on it.

Even for the worst case (STRATIX IV, WG-14), it only takes 37 attempts (out of 768 primitive polynomials), and it drops to 4 for WG-16 (out of 2048 primitive polynomials). The data point for MAX 10 and WG-16 is not shown because the design does not fit on that particular FPGA.

It is, however, very important to note that polynomials which gave the smallest DWGP area for each implementation technology most of the time were different (see further discussion on page 62 and example in Figure 4.38 on page 63).

## 4.2 DWGP Implementations Using Constant Array

As a part of the work, it is important to assess how switching to constant array implementations affects DWGP area minimization. It turns out that the results are not as promising as with discrete components. There is very low correlation on 65 nm CMOS, which makes it hard to minimize the area. On both FPGAs, however, finding the best polynomial is not needed because area variation is either zero (STRATIX IV) or very low (MAX 10).

### 4.2.1 65 nm CMOS ASIC Library

Figure 4.21, 4.22 and 4.23 show much lower correlation between DWGP area and all of the 3 metrics for 65 nm CMOS library. Therefore, using reduction matrix Hamming weight as a metric to reduce search space by eliminating some polynomials is not going to give significant advantage. Refer to the following summary chapter for specific numbers and their implications.

Decimation impact on DWGP area for WG-8 is shown in Figure 4.24. In order to make a direct comparison with discrete components implementation, refer to Figure 4.25, which shows that WG-8 provides the same behaviour as WG-14 in Figure 4.5. Inclusion of decimation is no longer providing a small consistent increase in area. Decimation completely changes the values in constant array and its effect on area is unpredictable.

As can be inferred from Figure 4.24, there are cases when the same field defining polynomial gives relatively small DWGP area without decimation and relatively large DWGP area with decimation, and vice versa. There is also no significant change in maximum / minimum DWGP area for different decimation values, while with discrete components decimation is a separate hardware block with consistent area penalty.

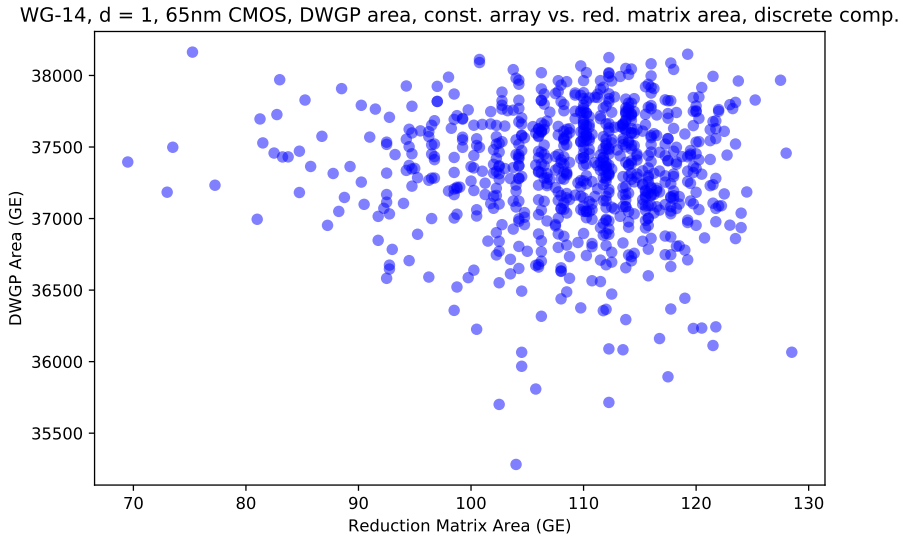


Figure 4.21: DWGP vs. reduction matrix area (WG-14, 65 nm CMOS, d = 1, const. array)

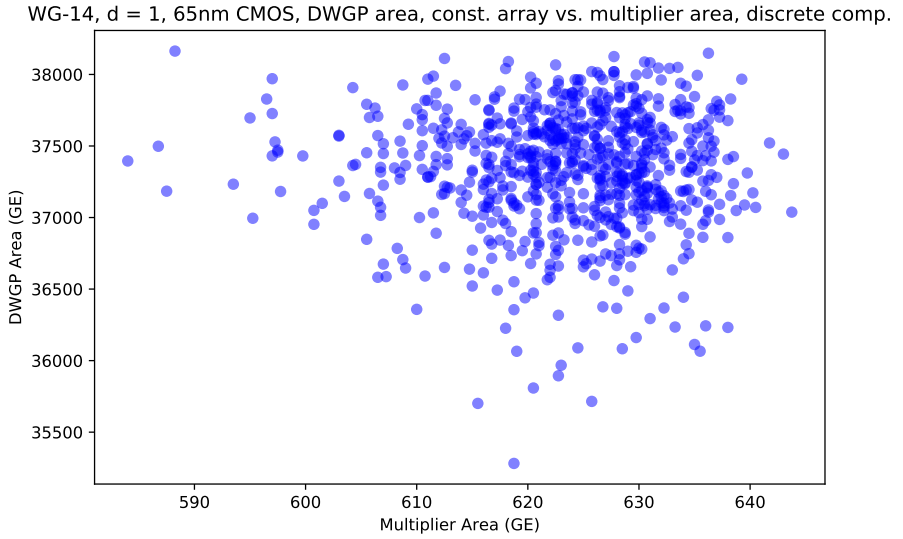


Figure 4.22: DWGP vs. multiplier area (WG-14, 65 nm CMOS, d = 1, constant array)

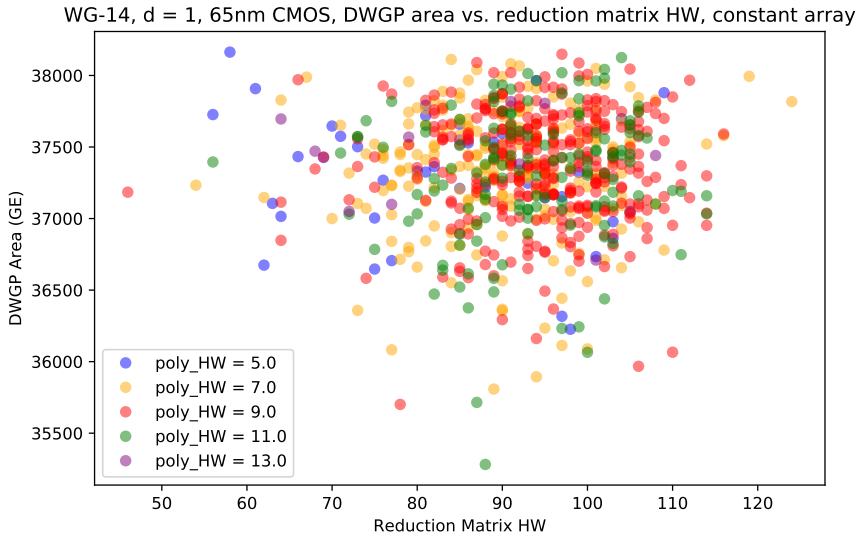


Figure 4.23: DWGP area vs. Hamming weights (WG-14, 65 nm CMOS, d = 1, const.)

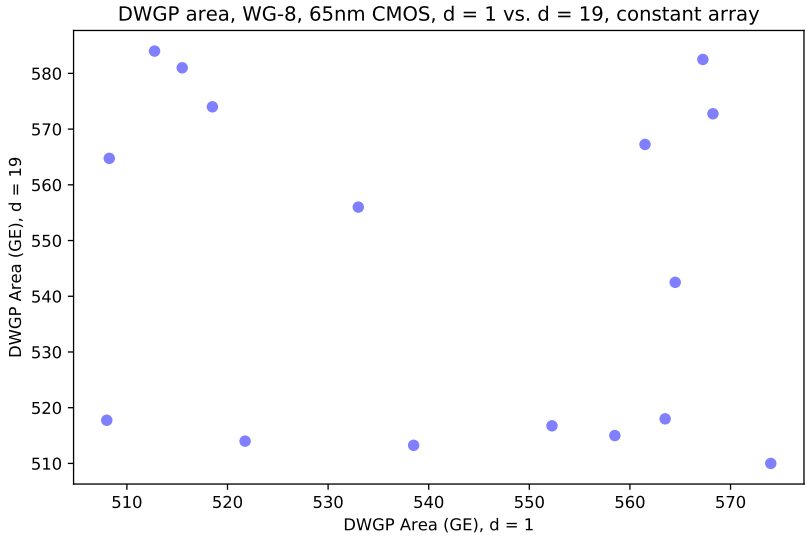


Figure 4.24: Decimation effect on DWGP area (WG-8, 65 nm CMOS, constant array)

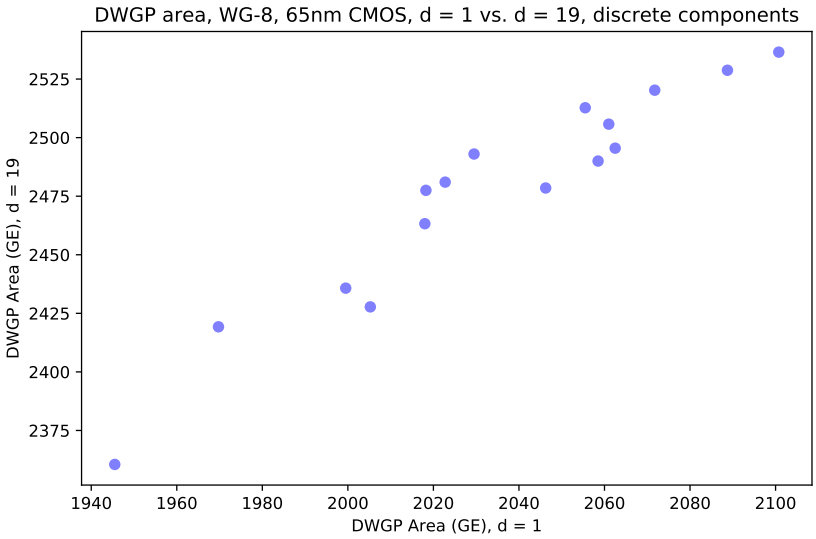


Figure 4.25: Decimation effect on DWGP area (WG-8, 65 nm CMOS, discrete components)

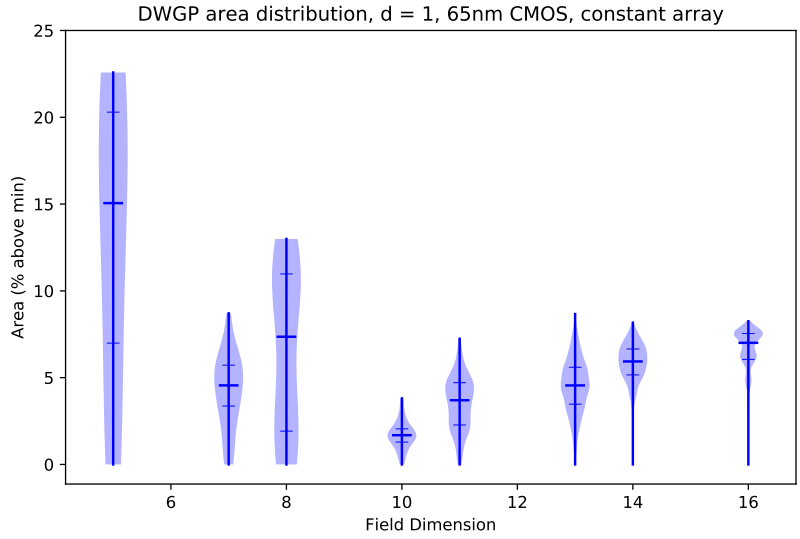


Figure 4.26: DWGP area distribution (d = 1, 65 nm CMOS, constant array)

DWGP area distribution in Figure 4.26 shows a significantly lower area variation compared to discrete components implementations. For instances that use large field sizes (for example, WG-14 and WG-16) we can see familiar spikes down that correspond to a few polynomials that give sharp decrease in the area compared to the mean value. However, the difference between mean and absolute minimum does not exceed 7% (except for WG-5). This means that while finding the best polynomial is much harder for constant array implementations due to low correlation of the DWGP area with reduction matrix Hamming weight, it is also less important due to lower area variability.

### 4.2.2 STRATIX IV FPGA

STRATIX IV provides the same DWGP area with constant array implementaion for all field defining polynomials.

### 4.2.3 MAX 10 FPGA

For the MAX 10 board, WG-11 is used as an example instead of WG-14. The reason behind this is that DWGP for WG-13 (and higher) in constant array implementations do not fit on this FPGA. The results are shown in Figure 4.27, 4.28 and 4.29.

The MAX 10 does not provide the same DWGP area for all field defining polynomials, unlike STRATIX IV. It could be explained by MAX 10 having smaller LUTs in comparison. Despite that, it is obvious that area variation is very small if we look at Figure 4.30, where DWGP area distribution is shown for the case without decimation. High variability for WG-5 can be explained - DWGP area for this case varies from 30 to 33 LUTs, which is actually insignificant in practice. Standard deviation of only several LUTs is also the case for other field sizes. While formally we never observed the same zero variability as with STRATIX IV, we can proceed as if it was zero.

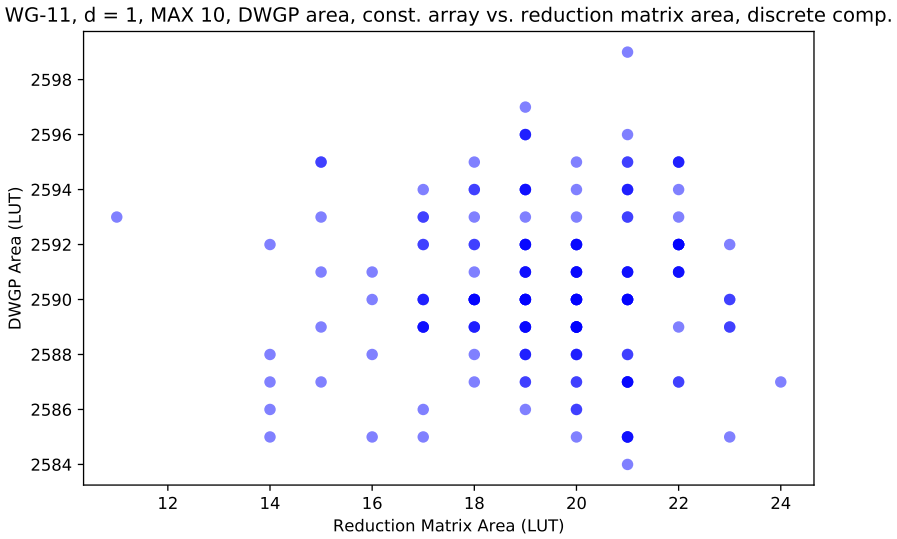


Figure 4.27: DWGP area vs. reduction matrix area (WG-11, MAX 10, d = 1, const. array)

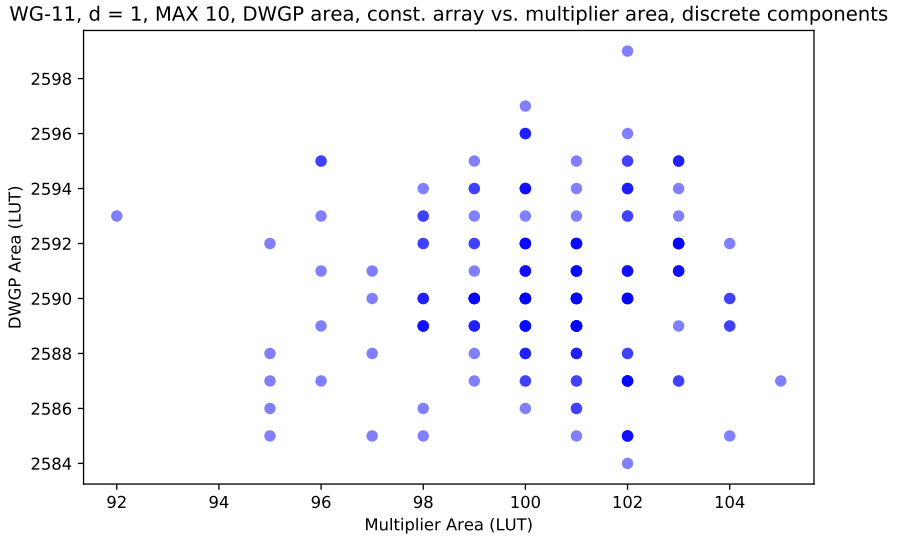


Figure 4.28: DWGP area vs. multiplier area (WG-11, MAX 10, d = 1, constant array)

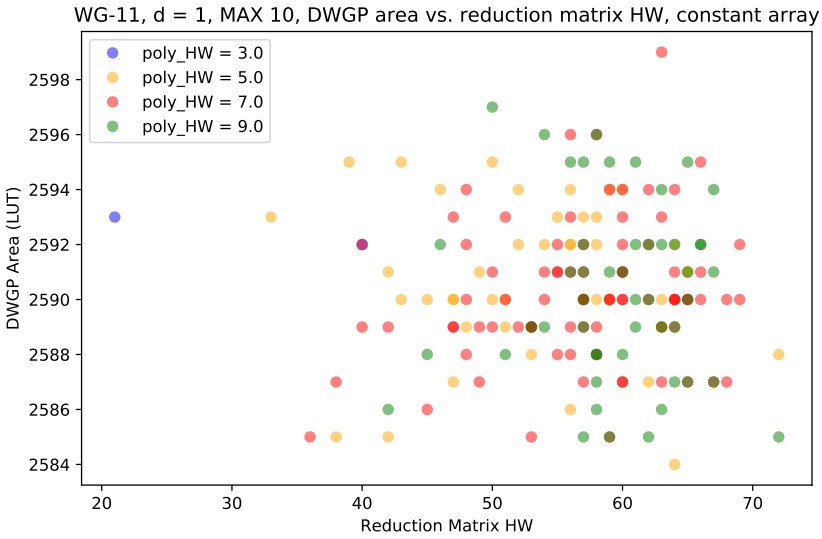


Figure 4.29: DWGP area vs. Hamming weights (WG-11, MAX 10, d = 1, constant array)

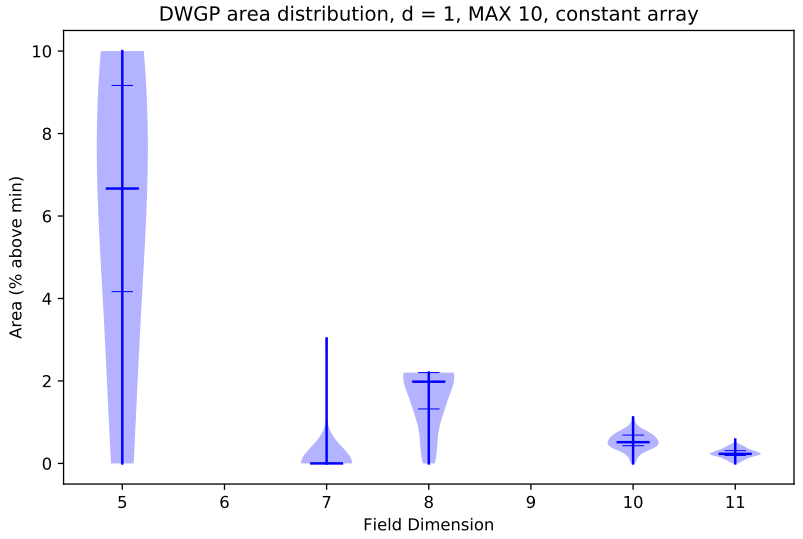


Figure 4.30: DWGP area distribution (d = 1, MAX 10, constant array)

### 4.2.4 Brief Summary for Constant Array Implementations

The number of attempts needed for constant array implementations and different fields and different libraries (65 nm CMOS, STRATIX IV and MAX 10) is shown in Figure 4.31. The significant values for 65 nm CMOS library are caused by low correlation shown in previous charts. However, those numbers are not going to matter in practice. As shown is the following chapter, constant array implementations of DWGP are only useful for small fields (WG-5, WG-7, WG-8 and WG-10) and become impractically large afterwards. The data points for MAX 10 and WG-13, WG-14 and WG-16 are not shown because these designs are too large and do not fit on that FPGA. The number of attempts for STRATIX IV is 0 for all fields because there is no difference in area for different field defining polynomials. Moreover, despite high value of attempts for WG-11 and MAX 10 board, the actual area varies from 2584 to 2599 LUTs, with mean value of 2590.37 and standard deviation of 2.84. This corresponds well with the trend that all constant array implementations on FPGA have 0 or very low variation in absolute units (LUTs).

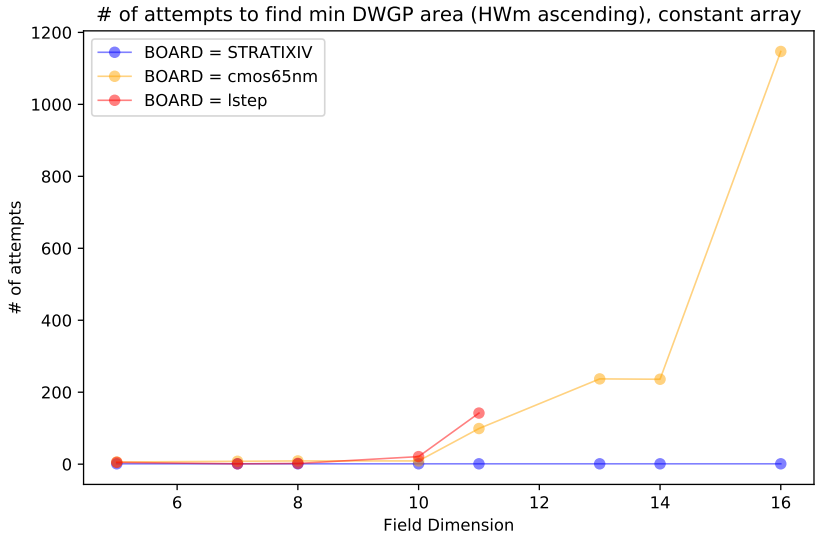


Figure 4.31: Number of attempts needed to find the smallest DWGP (d = 1, const. array)

## 4.3 DWGP Area Summary: Discrete Components vs. Constant Array

The discrete components implementations increase in area quadratically with field dimension. The constant array implementations increase in area exponentially with field dimension. Table 4.1 includes pre place-and-route area results for the smallest DWGP. Implementations using discrete components and constant array are presented separately, and two decimation values are shown for discrete components. A list of field defining polynomials that provide the smallest DWGP area for each implementation is compiled in Table 4.2. Constant array implementations for FPGAs are not listed because of extremely low variation – there is no practical difference between different polynomials (standard deviation is 0 for STRATIX IV and 1-3 LUT for MAX 10 for all field sizes). Results for WG-16 and MAX 10 for discrete components implementation are not provided because the design does not fit on the FPGA.

| | **d** | **Discrete Components** | | | | | | **Constant Array** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **DWGP** | | | | **DWGT** | | **DWGP** | | | | **DWGT** | |
| | | **min** | **max** | **mean** | **std** | **min*** | **max *** | **min** | **max** | **mean** | **std** | **min *** | **max *** |
| **WG-5** | **1** | 426 | 473 | 446 | 21 | 125 | 183 | 46 | 57 | 53 | 4 | 9 | 13 |
| | **11** | 537 | 619 | 586 | 29 | 201 | 218 | | | | | | |
| **WG-7** | **1** | 1157 | 1390 | 1316 | 66 | 904 | 1250 | 247 | 268 | 257 | 6 | 53 | 57 |
| | **63** | 1711 | 2064 | 1957 | 101 | 1435 | 1940 | | | | | | |
| **WG-8** | **1** | 1937 | 2078 | 2035 | 42 | 1641 | 2081 | 508 | 574 | 542 | 24 | 90 | 87 |
| | **19** | 2360 | 2536 | 2477 | 46 | 2064 | 2553 | | | | | | |
| **WG-10** | **1** | 2520 | 3018 | 2928 | 87 | 2007 | 2816 | 2173 | 2257 | 2211 | 18 | 300 | 309 |
| | **73** | 3167 | 3770 | 3645 | 105 | 2650 | 3544 | | | | | | |
| **WG-11** | **1** | 3134 | 3674 | 3567 | 81 | 2556 | 3560 | 5104 | 5470 | 5282 | 79 | 520 | 498 |
| | **203** | 4634 | 5407 | 5248 | 119 | 4033 | 5290 | | | | | | |
| **WG-13** | **1** | 5642 | 6366 | 6170 | 99 | 6511 | 5737 | 18163 | 19731 | 18985 | 289 | 1532 | 1528 |
| | **195** | 7271 | 8209 | 7982 | 124 | 7359 | 8390 | | | | | | |
| **WG-14** | **1** | 7136 | 8102 | 7885 | 116 | 7696 | 8911 | 35299 | 38164 | 37338 | 414 | 2752 | 2748 |
| | **47** | 9619 | 11006 | 10665 | 161 | 10204 | 11796 | | | | | | |
| **WG-16** | **1** | 11052 | 12302 | 12036 | 145 | 11259 | 12512 | 127171 | 137624 | 135513 | 1596 | 8673 | 8578 |
| | **1057** | 12948 | 14353 | 14029 | 159 | 13145 | 14574 | | | | | | |

**d** = decimation exponent

* min/max DWGT = DWGT area for the field defining polynomial that corresponds to min/max DWGP

Table 4.1: DWGP and DWGT area in GE, pre place-and-route (65 nm CMOS)

| | Field defining polynomial, $f(x)$ | ASIC / FPGA | Implementation |
|---|---|---|---|
| **WG-5** | $x^5 + x^3 + 1$ | 65 nm CMOS | comp |
| | $x^5 + x^4 + x^3 + x + 1$ | | const |
| | $x^5 + x^3 + 1$ | STRATIX IV | comp |
| | $x^5 + x^3 + 1$ | MAX 10 | comp |
| **WG-7** | $x^7 + x + 1$ | 65 nm CMOS | comp |
| | $x^7 + x^6 + x^5 + x^3 + x^2 + x + 1$ | | const |
| | $x^7 + x + 1$ | STRATIX IV | comp |
| | $x^7 + x + 1$ | MAX 10 | comp |
| **WG-8** | $x^8 + x^6 + x^4 + x^3 + x^2 + x + 1$ | 65 nm CMOS | comp |
| | $x^8 + x^6 + x^5 + x^2 + 1$ | | const |
| | $x^8 + x^4 + x^3 + x^2 + 1$ | STRATIX IV | comp |
| | $x^8 + x^4 + x^3 + x^2 + 1$ | MAX 10 | comp |
| **WG-10** | $x^{10} + x^3 + 1$ | 65 nm CMOS | comp |
| | $x^{10} + x^5 + x^3 + x^2 + 1$ | | const |
| | $x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + 1$ | STRATIX IV | comp |
| | $x^{10} + x^9 + x^7 + x^6 + x^4 + x^3 + x^2 + x + 1$ | MAX 10 | comp |
| **WG-11** | $x^{11} + x^2 + 1$ | 65 nm CMOS | comp |
| | $x^{11} + x^{10} + x^8 + x^7 + x^6 + x^4 + x^2 + x + 1$ | | const |
| | $x^{11} + x^7 + x^5 + x^3 + 1$ | STRATIX IV | comp |
| | $x^{11} + x^8 + x^5 + x^2 + 1$ | MAX 10 | comp |
| **WG-13** | $x^{13} + x^{12} + x^{11} + x^9 + x^6 + x^5 + x^4 + x^2 + 1$ | 65 nm CMOS | comp |
| | $x^{13} + x^{12} + x^{11} + x^9 + x^7 + x^6 + x^3 + x^2 + 1$ | | const |
| | $x^{13} + x^{11} + x^9 + x^5 + 1$ | STRATIX IV | comp |
| | $x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + 1$ | MAX 10 | comp |
| **WG-14** | $x^{14} + x^{13} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^3 + x + 1$ | 65 nm CMOS | comp |
| | $x^{14} + x^{13} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^6 + x^3 + x^2 + 1$ | | const |
| | $x^{14} + x^{12} + x^9 + x^6 + x^5 + x^2 + 1$ | STRATIX IV | comp |
| | $x^{14} + x^8 + x^3 + x^2 + 1$ | MAX 10 | comp |
| **WG-16** | $x^{16} + x^{14} + x^{12} + x^{10} + x^8 + x^7 + x^6 + x^5 + x^4 + x^2 + 1$ | 65 nm CMOS | comp |
| | $x^{16} + x^{15} + x^{14} + x^{11} + x^5 + x^4 + x^2 + x + 1$ | | const |
| | $x^{16} + x^{10} + x^7 + x^6 + 1$ | STRATIX IV | comp |

**comp** = discrete components; **const** = constant array

Table 4.2: Field defining polynomials for the smallest DWGP area (d = 1)

An interesting observation is that DWGT implemented with constant array is smaller than DWGT done using discrete components for all field sizes. We did not perform DWGT area profiling, and the *min* and *max* values for DWGT area listed in Table 4.1 correspond

to the same field defining polynomial that gave the smallest DWGP area. The massive difference in area between constant array DWGP and DWGT can be explained by a much smaller output size – 1 bit instead of $m$ bits, where $m$ is a field dimension.

Constant array DWGT is smaller than discrete components DWGT as well as DWGP of any implementation for all fields evaluated. The biggest area difference is observed for WG-11, for which constant array DWGT has 9.3 times lower area than DWGP. As a reminder, DWGP for WG-11 is built using discrete components. As the field size increases, the area difference gets lower because DWGP area grows quadratically with field dimension, while DWGT constant array area grows exponentially. This area advantage still remains even for WG-16 (1.5x times). Extrapolating the trend shows that WG-16 is indeed using the larger field, for which constant array DWGT gives area advantage over DWGP built with discrete components, which is shown in Figure 4.32. And as can be inferred from the same table, DWGT built with discrete components does not give any advantage over DWGP using the same implementation - synthesis tools seem to lose the ability to efficiently analyze and simplify complex circuits after certain size.

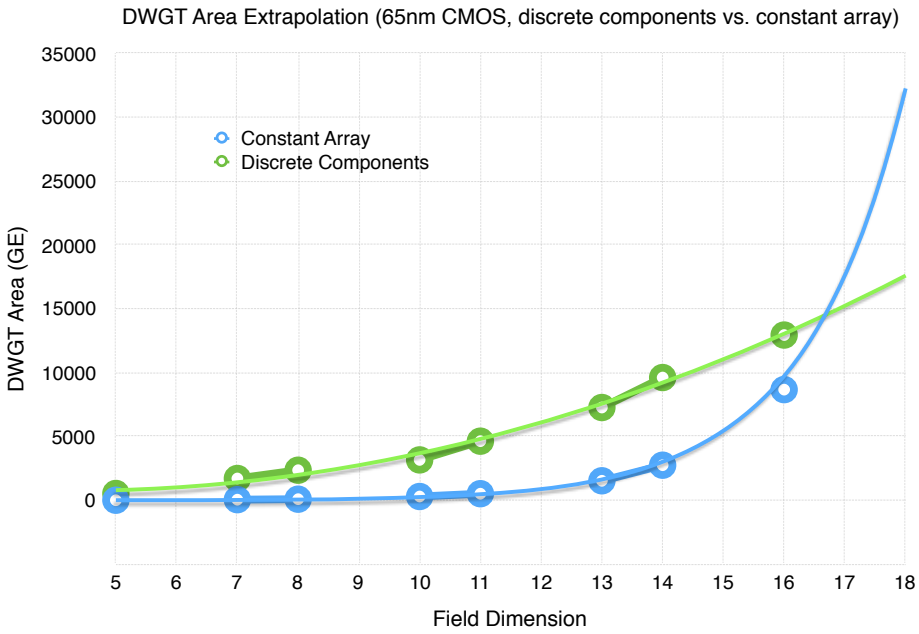


Figure 4.32: Extrapolation of DWGT area, const. array vs. discrete comp. (65 nm CMOS)

Another important note is that constant array DWGP followed by a trace equation or trace component comprised of squarers does not achieve the same low area as straight-

forward constant array DWGT implementation for fields larger than 10. The results are compiled in Table 4.3. Note that polynomials for smallest discrete components DWGP area were used to conduct the test, which explains a minor mismatch with data from Table 4.1. There is still a noticeable area decrease of −81% for DWGP + trace equation w.r.t. the straightforward constant array DWGP implementation for WG-11 (and the area penalty compared to the pure constant array DWGT is +94%), but for WG-13 and higher even than is no longer the case. For them DWGP + trace component area is actually slightly larger than pure DWGP due to the extra gates needed for the trace equation. DWGP + trace equation provides lower area, but still significantly above the pure constant array DWGT. A possible explanation is that trace components end up too large and complicated and synthesis tools evaluate and optimize them only individually. In the case of DWGP + trace equation, it might be that synthesis tools evaluate DWGP first, finding and implementing as many common subexpressions as possible for each DWGP output bit. Then, when trace equation effectively discards most of the DWGP output bits, large portion of the DWGP hardware still remains. In order to match straightforward constant array DWGT, it has to be re-evaluated again. That is why in Section 5.3 we only use actual constant array DWGT for all field sizes.

| | **DWGP (reference)** | **DWGT** | **Constant array DWGP +** | |
|---|---|---|---|---|
| | | | **trace equation** | **trace component** |
| **WG-5** | 49 | 12 | 12 | 12 |
| **WG-7** | 257 | 53 | 53 | 53 |
| **WG-8** | 552 | 94 | 94 | 105 |
| **WG-10** | 2190 | 305 | 305 | 341 |
| **WG-11** | 5406 | 517 | 1004 | 1026 |
| **WG-13** | 19014 | 1532 | 11972 | 19075 |
| **WG-14** | 37395 | 2762 | 21846 | 37904 |
| **WG-16** | 137058 | 8593 | 17259 | 137283 |

Table 4.3: Constant array DWGT area using 3 different approaches (d = 1, 65 nm CMOS)

The smallest DWGP area for discrete components vs. constant array implementations is also plotted in Figure 4.33 for 65 nm CMOS. Note, that logarithmic scale was used. As we can see, the crossover point is somewhere between WG-10 and WG-11. Constant array implementations provide smaller area for field sizes below the crossover point, while discrete components give smaller area for field sizes above the crossover point.

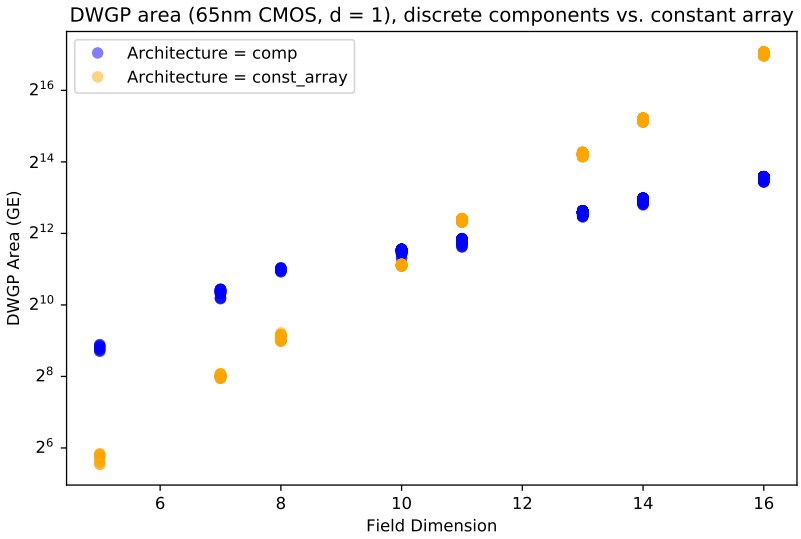


Figure 4.33: DWGP area for discrete components vs. constant array (d = 1, 65 nm CMOS)

The same plot for STRATIX IV is shown in Figure 4.34. Surprisingly, the crossover point moved up to slightly above WG-13. The opposite happened for MAX 10 - the crossover point moved down to slightly below WG-10 according to Figure 4.35.

Minimal area for DWGP with discrete components implementation for 65 nm CMOS library could be up to 10 standard deviations lower than mean area (see Figures 4.6 and 4.7). Typically, there are a few standalone polynomials that provide significantly lower area than the rest. The constant array DWGP implementations for the 65 nm CMOS library (except for WG-7, 8 and 16, see Figure 4.26) have noticeably smaller difference between mean area and absolute minimum compared to discrete components implementation, despite having similar standard deviation values in %. WG-16 only shows modest decrease in area variation for constant array implementations, while WG-7 and WG-8 show the opposite effect, where area variation for discrete components implementations is lower (see Figure 4.26). Variability for constant array implementations is zero with STRATIX IV and close to zero in practice with MAX 10.

Using reduction matrix hamming weight to find field defining polynomial is effective for DWGP implemented with discrete components. This method does not work for constant array implementations. However, constant array implementations are used for smaller fields, where exhaustive search is practical.

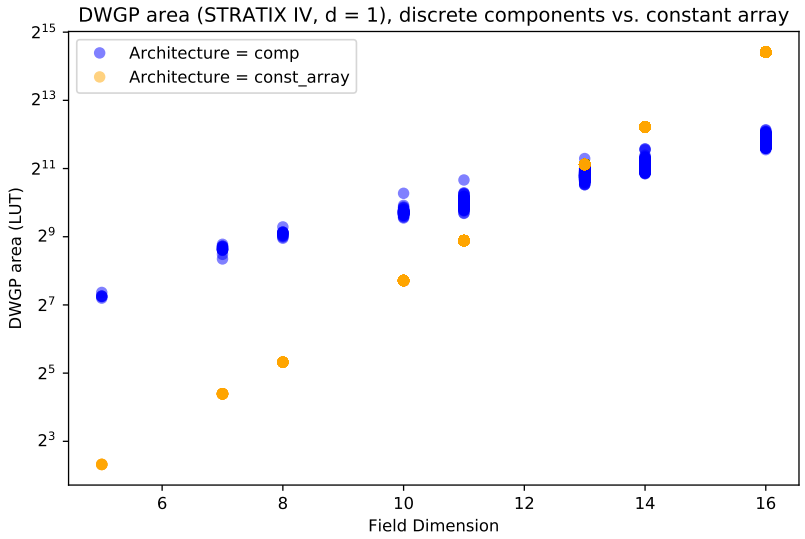


Figure 4.34: DWGP area for discrete comp. vs. const. array (d = 1, STRATIX IV)

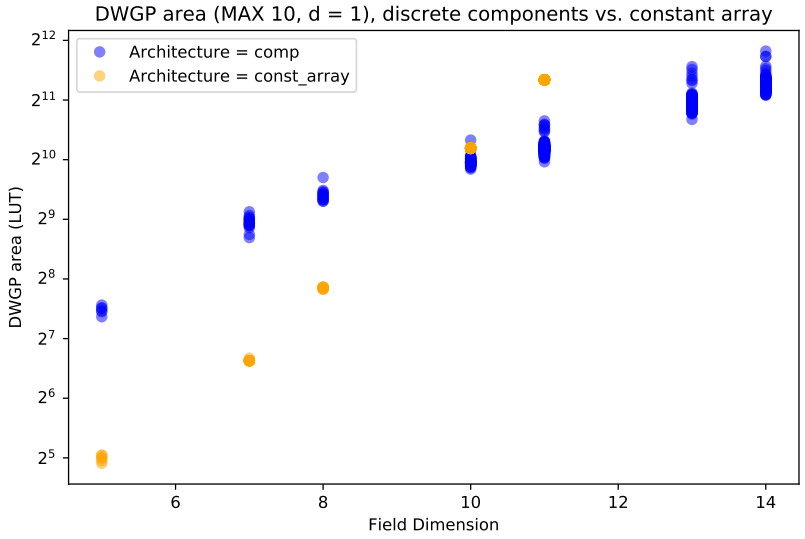


Figure 4.35: DWGP area for discrete components vs. constant array (d = 1, MAX 10)

The last step is to look for correlation between 65 nm CMOS and FPGA implementations and for correlation between discrete components and constant array implementations.

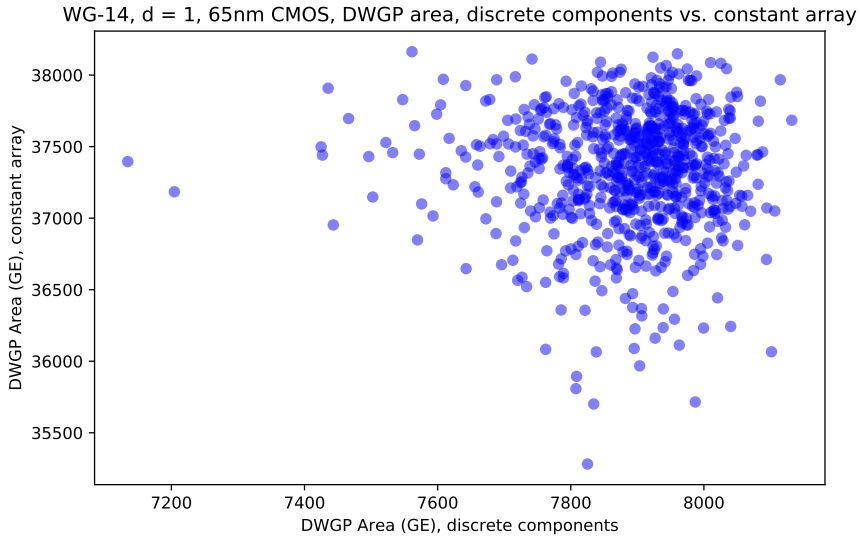


Figure 4.36: DWGP area for WG-14, 65 nm CMOS (discrete comp. vs. constant array)

Figure 4.36 shows DWGP discrete components area vs. constant array area for each primitive polynomial for WG-14 using 65 nm CMOS library. The bottom left quadrant of the chart is empty, which means that there are no polynomials that result in the smallest DWGP for both implementations simultaneously. Polynomials from top left quadrant (there are very few of them) provide small discrete components implementation and large constant array implementation, while polynomials from bottom right quadrant give the opposite. Very dense top right quadrant corresponds to the majority of polynomials that give large DWGP area for both implementations. This dense area corresponds to the peak in distribution in Figures 4.6 and 4.7 for field dimension $m = 14$. The conclusion is that it is necessary to find best field defining polynomial for both implementations separately.

The same plot for MAX 10 and WG-11 is shown in Figure 4.37. It is important to note that variation on Y axis (constant array DWGP implementation area) is very small. While bottom left quadrant is not completely empty, the overall conclusion is the same as for 65 nm CMOS if the absolute smallest DWGP area for both implementations is required. There are, however, few "sweet-spot" polynomials, that give very small area for both implementations. And, of course, one may consider very small variation in DWGP area for constant array insignificant in practice.

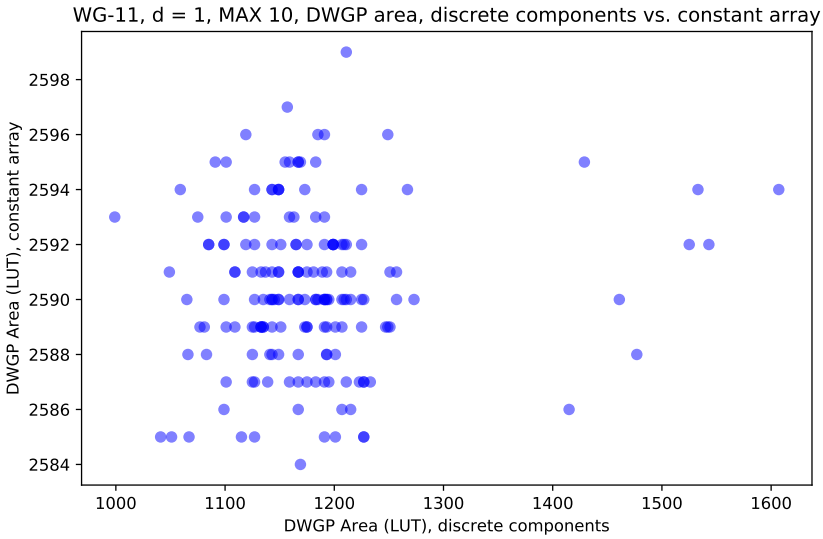


Figure 4.37: DWGP area for WG-11, MAX 10 (discrete componentsvs. constant array)

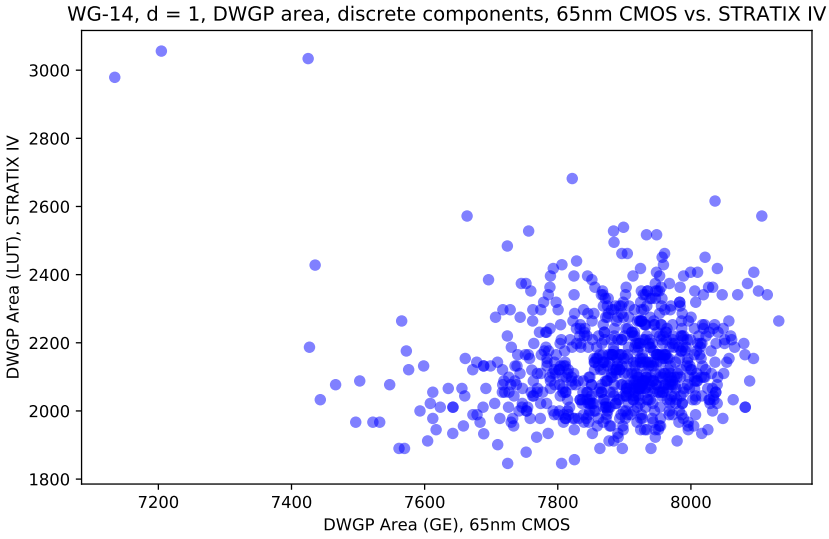


Figure 4.38: DWGP area for WG-14, discrete comp. (65 nm CMOS vs. STRATIX IV)

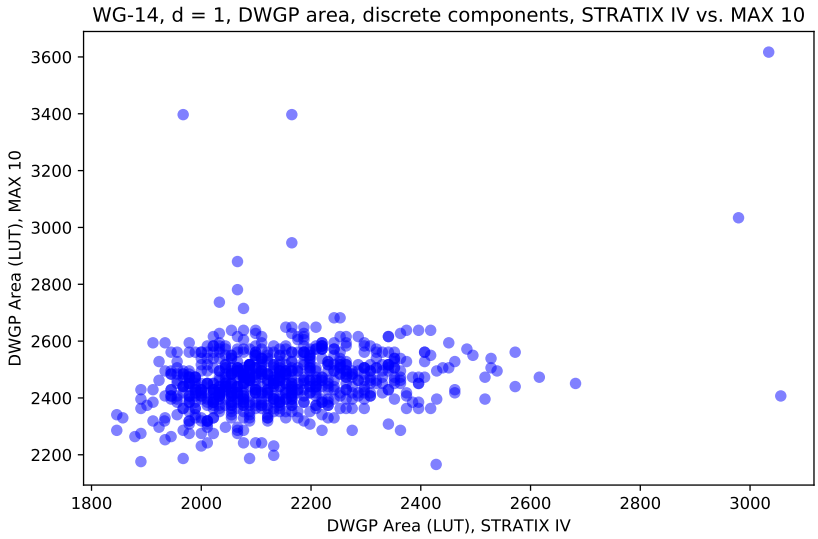


Figure 4.39: DWGP area for WG-14, discrete components (STRATIX IV vs. MAX 10)

Another question is whether the best field defining polynomial for 65 nm CMOS is also the best for STRATIX IV when using discrete components implementation. Figure 4.38 clearly shows that, unfortunately, it is not. Moreover, two dots in top left quadrant mean that the best two polynomials for 65 nm CMOS are in fact among 3 absolute worst ones for STRATIX IV. It is necessary to perform the search for best field defining polynomial separately for 65 nm CMOS library and STRATIX IV FPGA. However, WG ciphers must use the same field defining polynomial to communicate.

Figure 4.39 shows correlation between DWGP area on STRATIX IV and MAX 10 FPGAs using discrete components implementation. Unlike all previous cases, the majority of the polynomials are located in bottom left quadrant. That means there are many polynomials that results in small DWGP area for both FPGAs. However, the best polynomial for MAX 10 is clearly not the best for STRATIX IV. So despite higher correlation, it is recommended to search for best polynomial separately for each FPGA board.

# Chapter 5

# WG Cipher Implementation

In this chapter we focus on complete WG ciphers and provide the final choice of the following parameters:

- Preferred implementation technique (discrete components or constant array) for each field size
- Field defining polynomial $f(x)$ for each field size and chosen implementation technique
- $\gamma$ for multiplication by a constant
- LFSR feedback polynomial $l(x)$ for each field defining polynomial and chosen $\gamma$

We also present implementation results for different WG ciphers post place-and-route for the 65 nm CMOS library for 80-bit key and 80-bit IV, with metrics that include:

- Area in Gate Equivalents (GE)
- Maximum frequency in MHz
- Maximum throughput (also denoted as *tput*) in Mpbs.
- Maximum throughput over area in Mbps/GE

- Total power consumption in $\mu$W at 2 MHz frequency, calculated for a complete key + IV loading phase, initialization phase and generation of 1024 bits of keystream. It consists of dynamic power and leakage power.
- Energy per single keystream bit at 2 MHz frequency in pJ/bit

The choice of the 80-bit key is justified by smaller fields and smaller area implementations being the priority of this thesis. A 256-bit key version of WG-16 is also presented for a direct comparison with other implementations.

## 5.1 Parameters for the WG Ciphers

The standard WG cipher has a single DWGP and a separate *trace* unit connected to the last LFSR stage as shown in Figure 5.1. For each field size, we chose the implementation technique and field defining polynomial $f(x)$ that gives the smallest area based on the results in Section 4.3.

The value of $\gamma$ was chosen based on the smallest multiplication matrix Hamming weight [28], and LFSR feedback polynomial $l(x)$ was chosen arbitrarily from multiple polynomials found with GAP search for a given LFSR size and gamma.

There is a potential to further optimize the hardware by evaluating different LFSR feedback polynomials. The parameters are compiled in Table 5.1.

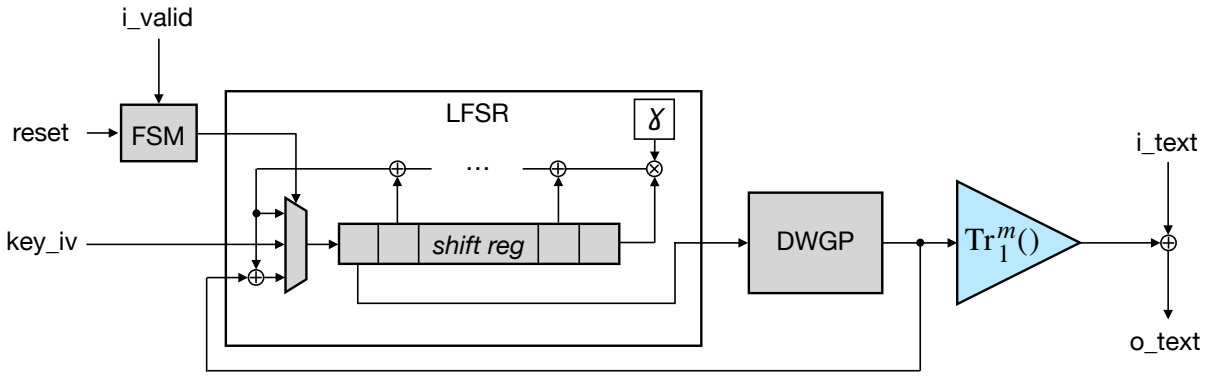


Figure 5.1: WG cipher structure

| | Field defining polynomial, $f(x)$<br>LFSR feedback polynomial, $l(x)$ | LFSR size | Key (bits) | State (bits) | Implementation | |
|---|---|---|---|---|---|---|
| | | | | | DWGP | DWGT |
| **WG-5** | $f(x) = x^5 + x^4 + x^3 + x + 1$<br>$l(x) = x^{32} + x^{14} + x^{13} + x^{11} + x^{10} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + x^2 + x + \omega$ | 32 | 80 | 160 | const | const |
| **WG-7** | $f(x) = x^7 + x^6 + x^5 + x^3 + x^2 + x + 1$<br>$l(x) = x^{23} + x^{12} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^3 + x^2 + x + \omega$ | 23 | | 161 | | |
| **WG-8** | $f(x) = x^8 + x^6 + x^5 + x^2 + 1$<br>$l(x) = x^{20} + x^8 + x^7 + x^5 + x^4 + x^3 + x^2 + x + \omega$ | 20 | | 160 | | |
| **WG-10** | $f(x) = x^{10} + x^5 + x^3 + x^2 + 1$<br>$l(x) = x^{16} + x^9 + x^8 + x^6 + x^5 + x^4 + x + \omega$ | 16 | | 160 | | |
| **WG-11** | $f(x) = x^{11} + x^2 + 1$<br>$l(x) = x^{15} + x^9 + x^6 + x^5 + x^4 + x^2 + \omega$ | 15 | | 165 | comp | |
| **WG-13** | $f(x) = x^{13} + x^{12} + x^{11} + x^9 + x^6 + x^5 + x^4 + x^2 + 1$<br>$l(x) = x^{13} + x^7 + x^4 + x^3 + x + \omega$ | 13 | | 169 | | |
| **WG-14** | $f(x) = x^{14} + x^{13} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^3 + x + 1$<br>$l(x) = x^{12} + x^7 + x^5 + x^4 + x^3 + \omega$ | 12 | | 168 | | |
| **WG-16** | $f(x) = x^{16} + x^{14} + x^{12} + x^{10} + x^8 + x^7 + x^6 + x^5 + x^4 + x^2 + 1$<br>$l(x) = x^{10} + x^7 + x^6 + x^2 + \omega$ | 10 | | 160 | | |
| **WG-16** | $f(x) = x^{16} + x^{14} + x^{12} + x^{10} + x^8 + x^7 + x^6 + x^5 + x^4 + x^2 + 1$<br>$l(x) = x^{32} + x^8 + x^6 + x^5 + x^3 + x^2 + x + \omega$ | 32 | 256 | 512 | | |

**comp** = discrete components; **const** = constant array

Table 5.1: Parameters for the WG ciphers

In the $l(x)$ polynomial for LFSR feedback the $\gamma$ is shown as a power of $\omega$, a root of field defining polynomial $f(x)$. For our polynomial bases the powers happen to be 1. LFSR size is chosen in order to fit 80-bit key and 80-bit IV (or 256-bit key and 256-bit IV for the second WG-16 instance). The actual internal state size is determined as field dimension $m$ times the *number of LFSR stages* and is padded with zeroes after loading if necessary.

As can be inferred from Table 5.1, WG-5, 7, 8 and 10 are implemented using constant array for both DWGP and DWGT (when applicable, see Section 5.3). WG-11, 13, 14 and 16, on the other hand, are implemented using discrete components for DWGP and constant array for DWGT. These choices are made based on the area profiling results summarized in Table 4.1 on page 56.

## 5.2 1 Bit / Cycle WG Instances

Figure 5.2 shows a schematic example of the WG cipher in the initialization phase (left) and the running phase (right). The example LFSR has *32* stages. A square with number *33* denotes a "future" value that was calculated through feedback and will be shifted into the LFSR in the next cycle. The DWGP output is fed back to the LFSR to

create non-linear feedback during the initialization phase for security reasons [5, 7]. That is the reason why we need DWGP to be a separate block and can't use the much smaller DWGT instead. A possible option would be to find and use a technique similar to the one discovered in [13, 14], which was briefly discussed in Section 2.4.1, but is not considered in this thesis.

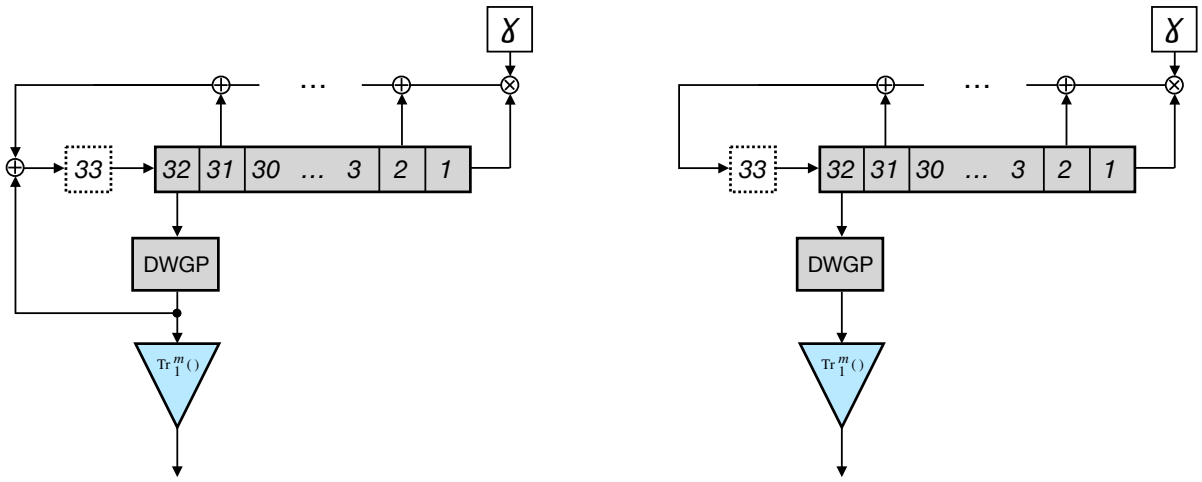


Figure 5.2: WG cipher: 1 round / cycle in init. phase, 1 bit / cycle in running phase

### 5.2.1 Results

Post place-and-route results for 65 nm CMOS ASIC library for different fields are compiled in Table 5.2. Decimation does not significantly increase the area for constant array implementations, but it noticeably affects both area and delay for discrete components implementations. There is also an abrupt drop in maximum frequency when going from constant array to discrete components. Maximum throughput does not have a separate column, but it has the same value in *Mbps* as maximum frequency in *MHz* because all ciphers in the table output 1 bit / clock cycle. As this thesis is considering low power and low frequency applications, it was chosen to benchmark total power consumption at frequency of 2 *MHz*. The total power was calculated for key/IV loading phase followed by initialization phase and generation of 1024 keystream bits in running phase. This value should scale closer to linear with frequency for larger designs like WG-11 to WG-16, because frequency-independent leakage power for them is typically within 1 to 5% of total power. For smaller constant array implementations, leakage power is between 14 and 17%

of total power. Energy per bit, however, is a different metric and should stay relatively constant for different frequencies. It is still a subject of higher variation based on the percentage of leakage power, with it getting lower for higher frequencies.

| | **d** | **Area (GE)** | **Max freq. (MHz)** | **Max T/A (Mbps/GE)** | **Power ($\mu$W) @ 2 MHz** | **Energy (pJ/bit) @ 2 MHz** |
|---|---|---|---|---|---|---|
| constant array implementations: | | | | | | |
| **WG-5** | 1 | 1160 | 792.4 | 0.6834 | 9.64 | 4.82 |
| | 11 | 1168 | 809.7 | 0.6934 | 9.99 | 5.00 |
| **WG-7** | 1 | 1396 | 575.0 | 0.4119 | 11.20 | 5.60 |
| | 63 | 1411 | 603.5 | 0.4279 | 11.18 | 5.59 |
| **WG-8** | 1 | 1644 | 489.7 | 0.2980 | 14.54 | 7.27 |
| | 19 | 1654 | 441.5 | 0.2670 | 12.77 | 6.39 |
| **WG-10** | 1 | 3412 | 310.6 | 0.0910 | 28.39 | 14.19 |
| | 73 | 3458 | 286.0 | 0.0827 | 28.60 | 14.30 |
| discrete components implementations: | | | | | | |
| **WG-11** | 1 | 4462 | 131.6 | 0.0295 | 100.30 | 50.15 |
| | 203 | 6006 | 81.4 | 0.0135 | 261.99 | 130.99 |
| **WG-13 *** | 1 | 7143 | 71.1 | 0.0100 | 276.44 | 138.22 |
| | 195 | 8847 | 51.4 | 0.0058 | 575.49 | 287.74 |
| **WG-14 *** | 1 | 8745 | 63.2 | 0.0072 | 404.73 | 202.36 |
| | 47 | 11355 | 42.7 | 0.0038 | 918.08 | 459.04 |
| **WG-16 *** | 1 | 12864 | 43.5 | 0.0034 | 945.03 | 472.51 |
| | 1057 | 14786 | 33.0 | 0.0022 | 1462.11 | 731.06 |
| **WG-16 * (256-bit key)** | 1 | 14841 | 42.9 | 0.0029 | 975.20 | 487.60 |
| | 1057 | 16765 | 35.0 | 0.0021 | 1457.38 | 728.69 |

**d** = decimation exponent; **T/A** = throughput over area

* trace module done via explicit equation derived from GAP

Table 5.2: WG ciphers, post place-and-route (80-bit key, 1 bit / cycle, 65 nm CMOS)

An effect of decimation on power and delay in WG-11 is demonstrated in Figure 5.3, which is a screenshot from *ModelSim* during post place-and-route simulation with timing data. The waveforms correspond to signals from similar parts of the design. After the rising edge of the clock signal (not shown) happened at 200250 *ns*, we can see signal glitches

until the values stabilize. Every transition consumes energy, and the higher number of such transitions results in an increase of power consumption per transistor, i.e. per unit of area. Another major part of the increased total power is the larger circuit area itself.

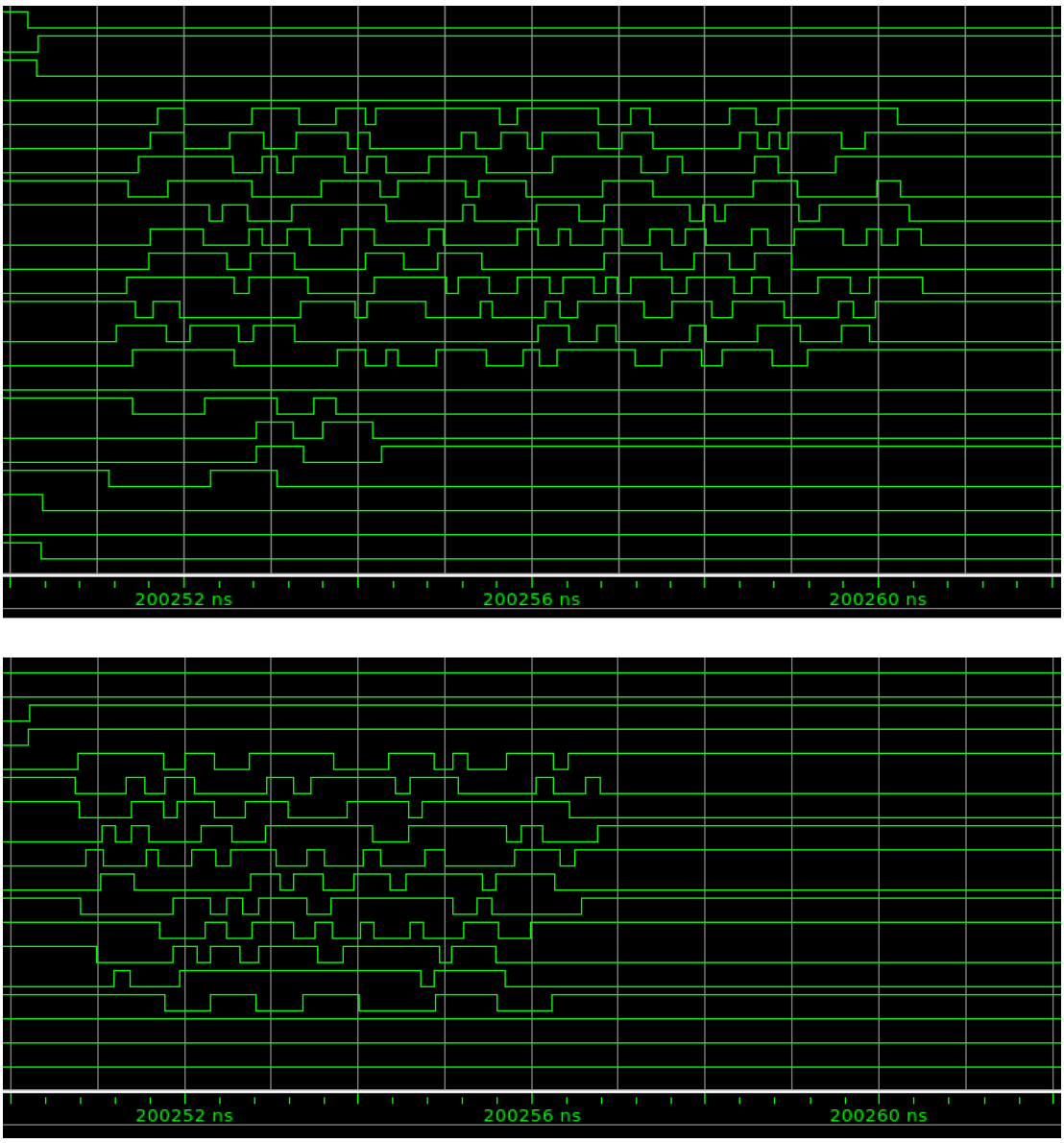


Figure 5.3: Signal glitches in WG-11 (top: d = 203, bottom: d = 1)

As mentioned in Section 3.4, trace implementation using squarers provided less than

optimal hardware for WG-13, 14 and 16 after synthesis compared to a simple trace equation obtained through GAP for a specific field size and field defining polynomial $f(x)$ - likely, due to a high complexity of the circuits. Area penalty post place-and-route was 46 GE for WG-13, 531 GE for WG-14 and 223 GE for WG-16. It also affected maximum frequency and power/energy, the worst case being WG-14 with 13% maximum frequency drop and 63 pJ more energy per 1 bit of keystream at 2 $MHz$. For every other instance from WG-5 to WG-11, there was absolutely no difference between two approaches. Using GAP-derived equation makes the code less generic, but currently is the best solution.

| | **d** | **Area** | **Max freq.** | **Max T/A** | **Power @ 2 MHz** |
|---|---|---|---|---|---|
| discrete components implementations: | | | | | |
| **WG-11** | 1 | −0.5% | −9.0% | −8.6% | +3.1% |
| | 203 | +0.4% | −7.8% | −8.2% | +5.4% |
| **WG-13 *** | 1 | −1.1% | −3.5% | −2.5% | +0.8% |
| | 195 | −0.4% | −4.2% | −3.8% | +0.9% |
| **WG-14 *** | 1 | −4.1% | −1.2% | +3.0% | −2.4% |
| | 47 | −4.7% | +3.7% | +8.8% | −3.2% |
| **WG-16 *** | 1 | −1.1% | −0.5% | +0.6% | +13.0% |
| | 1057 | +0.8% | +9.9% | +9.1% | +8.4% |
| **WG-16 *** | 1 | −0.9% | +3.6% | +4.6% | +9.1% |
| **(256-bit key)** | 1057 | +0.7% | +3.2% | +2.5% | +10.3% |

**d** = decimation exponent; **T/A** = throughput over area

* trace module done via explicit equation derived from GAP

Table 5.3: Difference from using Karatsuba mult. (80-bit key, 1 bit / cycle, 65 nm CMOS)

Table 5.3 shows the difference in area, maximum frequency, throughput over area and power between WG ciphers using Karatsuba multiplier and the ones using classic multiplier. The results are unexpected – unlike for standalone multipliers (see Table 3.2 on page 19), where Karatsuba architecture provided consistent and tangible decrease in area up to -16.6%, the effect on the complete WG ciphers is much smaller. In fact, only WG-14 shows noticeable decrease in area (-4.7% for decimation = 47) and quite often using Karatsuba multiplier has overall negative effect. It might be another example of optimization inconsistency, where a simpler single-entity classic multiplier design allows for more efficient implementations during synthesis.

## 5.3 Multiple Bits / Cycle WG Instances

In order to generate more than 1 bit of keystream per cycle, two approaches were considered - with fast and normal initialization speeds. The common idea is to have multiple copies of the LFSR feedback, each offset by 1 stage. During each cycle there are multiple future values ready, so it becomes possible to shift the LFSR by more than 1 stage at a time. The first option retains shifting LFSR by multiple stages in initialization phase, while the second option switches LFSR into standard 1 round / cycle shifting mode for initialization. This minor detail resulted in significant differences, which are discussed in the following sections. The last step is to add more copies of DWGP + trace.

These instances will be also referred as WG ciphers with *parallel output* and the number of bits / cycle will be also referred to as *degree of parallelism*. The current implementation does not support the number of output bits / cycle higher than LFSR size. Additionally, odd number of bits / cycle is not supported by our testbench environment. These limitations will be removed in the future.

### 5.3.1 Fast Initialization Phase Option

Figure 5.4 and Figure 5.5 show diagrams of WG ciphers that output 2 bits / cycle and 3 bits / cycle respectively. On the left side of each figure a configuration during initialization phase is given. The red line denotes critical path, which goes though all DWGP instances. Since DWGP is a major contributor to the overall cipher delay, doubling or tripling the number of the permutations will almost double or triple the delay. For an instance with 16 or 32 bits / cycle this limitation could have a very significant impact on maximum frequency.

This approach has one advantage:

- Faster initialization time

and three major disadvantages:

- Using separate DWGP + trace for each additional output bit result in large area
- Each additional output bit increases critical path delay
- Total number of initialization rounds can only be a multiple of the number of rounds / cycle, which limits design options

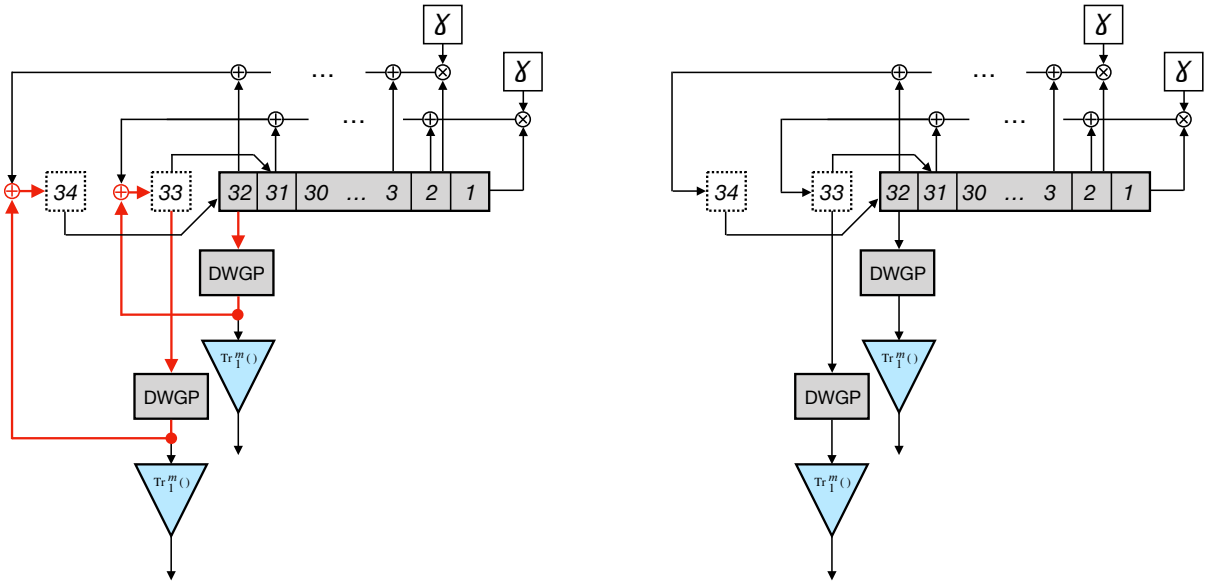


Figure 5.4: WG cipher: 2 rounds / cycle in init. phase, 2 bits / cycle in running phase

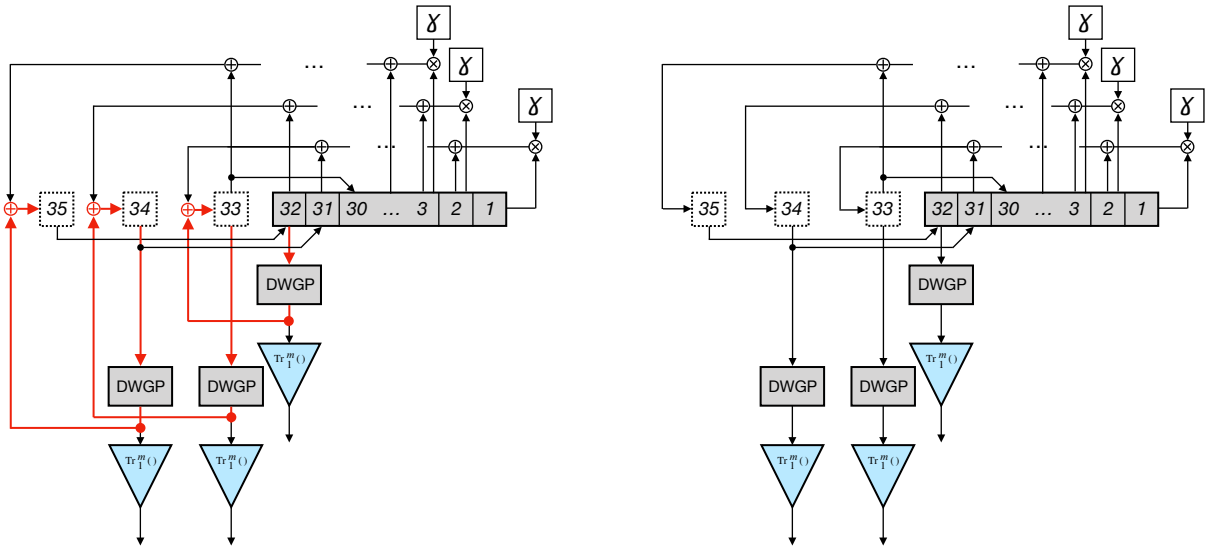


Figure 5.5: WG cipher: 3 rounds / cycle in init. phase, 3 bits / cycle in running phase

For example, WG-8 with 80-bit key and 80-bit IV uses 20 stage LFSR and requires 40

initialization rounds. If we have an instance with 8 rounds / cycle, we can achieve exactly 40 rounds in 5 cycles. However, if we increase the number of rounds to 16 / cycle, we can either get 32 rounds in 2 cycles or 48 rounds in 3 cycles. The former option is insufficient in terms of security [5, 7], and neither match the standard value. If standard for WG-8 with 80-bit key is changed to 48 initialization rounds, it will provide compatibility with 2, 4, 6, 8, 12 and 16 rounds / cycle in initialization phase. However, options like 10, 14 or odd number of rounds will remain incompatible.

### 5.3.2 Normal Initialization Phase Option

An alternative to fast initialization that we call normal initialization is shown in Figures 5.6 and 5.7. The LFSR is only shifting by one stage each cycle in initialization phase. This approach was initially proposed by Yang in [17] and has three major advantages:

- Regardless of the number of bits / cycle, there is only 1 DWGP in critical path
- It is possible to replace other DWGP + trace copies by DWGT to reduce area
- Any number of bits / cycle is possible because initialization phase does not impose any constraints

and two disadvantages:

- No decrease in initialization time compared to regular 1 bit / cycle instances
- Additional multiplexors for each LFSR stage to support two shifting modes result in area overhead noticeable for small field sizes (see Section 5.3.3)

According to Table 4.1 on page 56, DWGT as a single block using constant array implementation has noticeably smaller area than discrete component DWGT and DWGP of any implementation for each field size evaluated.

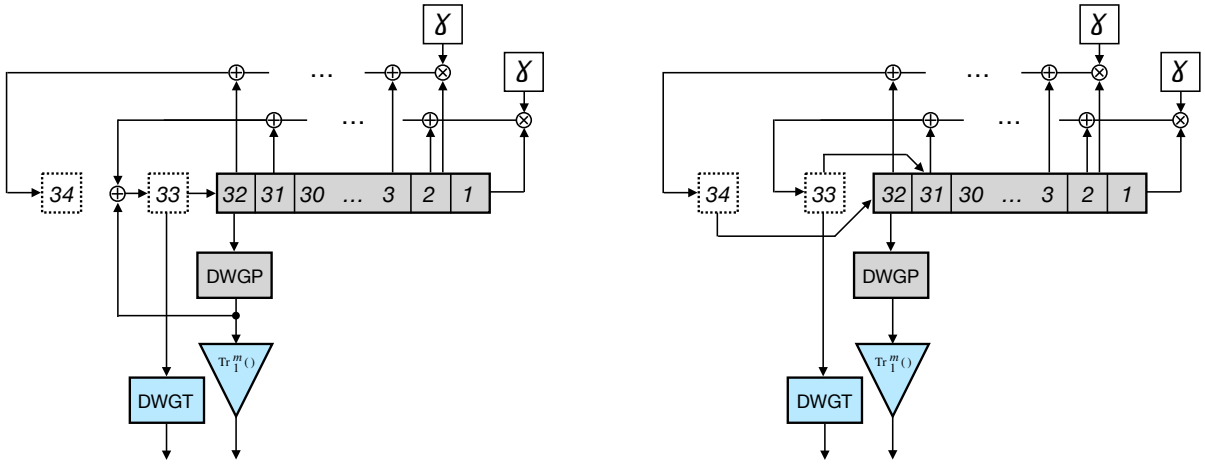


Figure 5.6: WG cipher: 1 round / cycle in init. phase, 2 bits / cycle in running phase

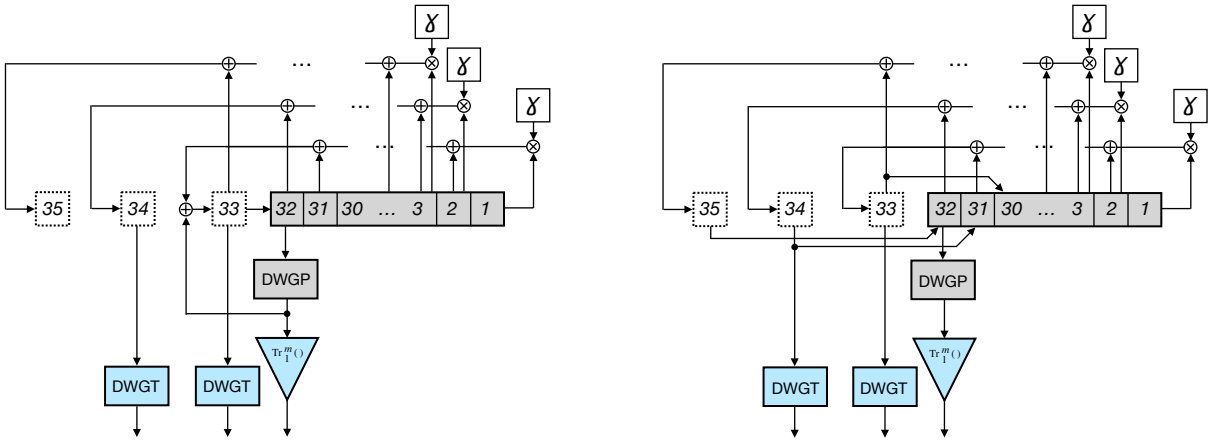


Figure 5.7: WG cipher: 1 round / cycle in init. phase, 3 bits / cycle in running phase

### 5.3.3 Results

In this section we present post place-and-route results for WG-5, WG-8 and WG-11 with various degree of parallelism. Results are presented in Tables 5.4, 5.5 and 5.6.

It is possible to isolate common trends:

- The higher is the degree of parallelization, the higher is area and maximum frequency advantage of the standard initialization phase option

| Bits / cycle | Area (GE) | Max freq. (MHz) | Max tput (Mbps) | Max T/A (Mbps/GE) | Power ($\mu$W) @ 2 MHz | Energy (pJ/bit) @ 2 MHz |
|---|---|---|---|---|---|---|
| 1 | 1168 | 809.7 | 809.7 | 0.6934 | 9.99 | 5.00 |
| with fast initialization phase: | | | | | | |
| 2 | 1295 | 502.3 | 1004.5 | 0.7758 | 11.30 | 2.82 |
| 4 | 1591 | 275.0 | 1099.8 | 0.6911 | 14.44 | 1.81 |
| 8 | 2245 | 137.7 | 1101.8 | 0.4908 | 18.51 | 1.16 |
| 16 | 3510 | 73.8 | 1180.6 | 0.3364 | 32.75 | 1.02 |
| 32 | 6148 | 39.2 | 1254.6 | 0.2041 | 81.80 | 1.28 |
| with standard initialization phase: | | | | | | |
| 2 | 1521 | 517.3 | 1034.7 | 0.6801 | 11.73 | 2.93 |
| 4 | 1698 | 585.8 | 2343.3 | 1.3798 | 13.35 | 1.67 |
| 8 | 2121 | 552.2 | 4417.4 | 2.0826 | 16.79 | 1.05 |
| 16 | 2833 | 532.2 | 8515.2 | 3.0059 | 24.71 | 0.77 |
| 32 | 4293 | 172.4 | 5515.3 | 1.2847 | 49.06 | 0.77 |

**tput** = throughput; **T/A** = throughput over area

Table 5.4: WG-5 ciphers, d = 11, post place-and-route (80-bit key, 65 nm CMOS)

- Fast initialization phase option tends to keep maximum throughput relatively constant for different value of bits / cycle - maximum frequency halves as we double the number of output bits
- Standard initialization phase option has consistent maximum frequency and linear increase in maximum throughput with the number of output bits
- Standard initialization phase option provides noticeable decrease in energy / bit of keystream for higher degree of parallelization
- Standard initialization phase option has higher area than fast initialization phase option for small fields like WG-5 with low number of output bits (2 or 4) due to the area overhead from multiplexors needed to support two LFSR shifting modes.

The drop in maximum frequency for WG-5 in Table 5.4 with standard initialization phase and 32 bits / cycle option might be attributed to a high number of LFSR feedback copies required and the corresponding increase in critical path delay. It formally decreases

our maximum T/A performance metric, but we chose to keep high density target to minimize area. It makes sense given the potential use case of these WG-5 cipher instances in very constraint environments like RFID tags of IoT devices at much lower frequency.

The same effect can be observed for WG-8 in Table 5.5 with 16 bit / cycle option, but to a lower extent. Also, due to higher DWGP area for WG-8, fast initialization phase no longer has area advantage for options with any number of bits / cycle.

| **Bits / cycle** | **Area (GE)** | **Max freq. (MHz)** | **Max tput (Mbps)** | **Max T/A (Mbps/GE)** | **Power ($\mu$W) @ 2 MHz** | **Energy (pJ/bit) @ 2 MHz** |
|---|---|---|---|---|---|---|
| 1 | 1654 | 441.5 | 441.5 | 0.2670 | 12.77 | 6.39 |
| with fast initialization phase: | | | | | | |
| 2 | 2373 | 241.8 | 483.6 | 0.2038 | 18.94 | 4.74 |
| 4 | 3725 | 113.8 | 455.0 | 0.1221 | 30.79 | 3.85 |
| 8 | 6312 | 49.5 | 396.3 | 0.0628 | 56.66 | 3.54 |
| 16 | 11741 | 26.7 | 426.7 | 0.0363 | 137.18 | 4.29 |
| with standard initialization phase: | | | | | | |
| 2 | 2153 | 439.9 | 879.9 | 0.4086 | 15.55 | 3.89 |
| 4 | 2563 | 453.9 | 1815.7 | 0.7083 | 19.46 | 2.43 |
| 8 | 3338 | 450.2 | 3602.0 | 1.0792 | 26.51 | 1.66 |
| 16 | 4914 | 270.0 | 4319.7 | 0.8790 | 44.37 | 1.39 |

**tput** = throughput; **T/A** = throughput over area

Table 5.5: WG-8 ciphers, d = 19, post place-and-route (80-bit key, 65 nm CMOS)

Since our implementation currently does not support the output of more bits / cycle than LFSR size, we had to limit our WG-11 with 14 bits / cycle in Table 5.6. While technically we could have used 15 bits / cycle (the LFSR size is 15 stages for 80-bit key / IV option), our testbench environment does not work with odd values yet. Overall, the trend of standard initialization phase having significant area and frequency advantages remains and is even exaggerated due to the highest relative area savings from using DWGT instead of DWGP.

Since all WG-5, WG-8 and WG-11 instances shown in this chapter are for 80-bit key, their internal states are similar in size (160 bits for WG-5, 160 bits for WG-8 and 165 bits for WG-11). The area overhead associated with multiplexors for standard initialization phase is therefore constant and becomes less noticeable with field size increase.

| Bits / cycle | Area (GE) | Max freq. (MHz) | Max tput (Mbps) | Max T/A (Mbps/GE) | Power ($\mu$W) @ 2 MHz | Energy (pJ/bit) @ 2 MHz |
|---|---|---|---|---|---|---|
| 1 | 6006 | 81.4 | 81.4 | 0.0135 | 261.99 | 130.99 |
| with fast initialization phase: | | | | | | |
| 2 | 10992 | 39.0 | 78.0 | 0.0071 | 506.85 | 126.71 |
| 4 | 21028 | 19.5 | 78.0 | 0.0037 | 1057.65 | 132.21 |
| 8 | 41106 | 9.6 | 76.8 | 0.0019 | 2171.68 | 145.29 |
| 14 | 71211 | 5.1 | 71.3 | 0.0010 | 5140.73 | 183.60 |
| with standard initialization phase: | | | | | | |
| 2 | 6967 | 80.7 | 161.3 | 0.0232 | 240.51 | 60.13 |
| 4 | 8271 | 79.6 | 318.2 | 0.0385 | 263.03 | 32.88 |
| 8 | 10901 | 81.8 | 654.2 | 0.0600 | 277.15 | 17.32 |
| 14 | 14775 | 78.5 | 1098.8 | 0.0744 | 322.89 | 11.53 |

**tput** = throughput; **T/A** = throughput over area

Table 5.6: WG-11 ciphers, d = 203, post place-and-route (80-bit key, 65 nm CMOS)

## 5.4 Comparison with Existing Implementations

The implementation results from Table 5.4 for WG-5 are compared to the ones from [6] in Table 5.7. However different ASIC library was used, which has to be taken into account. For 1 and 2 bits per clock cycle, WG-5 implementations from this thesis provide slightly lower area in GE (which can be caused by a different library) and lower energy per bit, which can be partially attributed to 20 times higher frequency that we used for power analysis because it reduces the contribution of leakage power.

WG-8 was implemented using constant array in [17]. Table 5.8 shows a comparison with our normal initialization mode. For 1 bit / cycle the results from this thesis provide 7.4% smaller area, but lower maximum frequency and slightly lower throughput over area. If we are not concerned with frequency, our implementation is superior. While we do not have 11 bits / cycle implementation, we have 8 and 16 bits / cycle - but both with lower throughput over area score.

While large fields like WG-16 were not the main focus of the thesis, it is still interesting to see how parametrized code fares against optimized implementations. When compared to the compiled results from various sources (see Table 5.9) it is clear that our imple-

| State / Key (bits) | Bits / cycle | d | Area (GE) | Power ($\mu$W) | Energy (pJ/bit) | Source |
|---|---|---|---|---|---|---|
| 130 nm CMOS ASIC library: | | | | | | |
| 160/80 | 1 | 1 | 1229 | 0.78 (0.1 MHz) | 7.80 | [6] * |
| | | 11 | 1235 | 0.79 (0.1 MHz) | 7.80 | |
| | 2 | 1 | 1350 | 0.84 (0.1 MHz) | 4.20 | |
| | | 11 | 1360 | 0.85 (0.1 MHz) | 4.20 | |
| 180 nm CMOS ASIC library: | | | | | | |
| 160/80 | 1 | 1 | 1361 | - | - | [6] * |
| | | 11 | 1373 | | | |
| | 2 | 1 | 1508 | | | |
| | | 11 | 1521 | | | |
| 65 nm CMOS ASIC library: | | | | | | |
| 160/80 | 1 | 1 | 1160 | 9.64 (2 MHz) | 4.82 (2 MHz) | this thesis |
| | | 11 | 1168 | 9.99 (2 MHz) | 5.00 (2 MHz) | |
| | 2 ** | 11 | 1295 | 11.30 (2 MHz) | 2.82 (2 MHz) | |
| | 4 ** | | 1591 | 14.44 (2 MHz) | 1.81 (2 MHz) | |
| | 2 † | 11 | 1521 | 11.73 (2 MHz) | 2.93 (2 MHz) | |
| | 4 † | | 1698 | 13.35 (2 MHz) | 1.67 (2 MHz) | |

**d** = decimation exponent
* implemented using GAP-derived equation
** with fast initialization phase
† with standard initialization phase

Table 5.7: WG-5 implementations comparison (polynomial bases)

mentation has higher area and lower maximum frequency even compared to non-pipelined implementation. One of the reasons of such a big difference in area is that [15, 16] used efficient tower field construction and [14] greatly reduced the number of multipliers from 12 to 6 after discovering new properties of the trace function. Such optimizations can be done for a specific set of parameters (field size, field defining polynomial) but not for a generic or universal version that was developed in this thesis. In addition to that, all pipelined architectures show significant intrinsic advantage in maximum frequency. When compared to [16], their pipelined implementation requires 15.8 times less energy per bit. This can be explained by much higher evaluation frequency, which reduces contribution of leakage power and 7-stage pipeline, which likely results in much less signal glitching due to reduced

| State / Key (bits) | Bits / cycle | d | Area (GE) | Max freq. (MHz) | Max tput (Mbps) | Max T/A (Mbps/GE) | Source |
|---|---|---|---|---|---|---|---|
| 160/80 | 1 | 19 | 1786 | 500 | 500 | 0.27996 | [17] * |
| | 11 | | 3942 | 610 | 6710 | 1.70218 | |
| 160/80 | 1 | 19 | 1654 | 441.5 | 441.5 | 0.2670 | this thesis |
| | 8 | | 3338 | 450.2 | 3602.0 | 1.0792 | |
| | 16 | | 4914 | 270.0 | 4319.7 | 0.8790 | |

**d** = decimation exponent; **tput** = throughput; **T/A** = throughput over area
* implemented using constant array

Table 5.8: WG-8 implementations comparison (polynomial bases, 65 nm CMOS)

complexity of combinational paths in-between stage boundaries.

| | Basis | State / Key (bits) | Bits / cycle | d | Area (GE) | Max freq. (MHz) | Max tput (Mbps) | Max T/A (Mbps/GE) | Power ($\mu$W) | Energy (pJ/bit) | Source |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 1 | | 9103 | 189 | 189 | 0.02076 | | | |
| † | PB | 512/256 | 1 | 1057 | 11795 | 1149 | 1149 | 0.09741 | - | - | [14] ** |
| †† | | | 1/6 | | 5267 | 680 | 113 | 0.02145 | | | |
| | | | 1 | | 8060 | 193 | 193 | 0.02395 | | | |
| † | PB * | 512/256 | 1 | 1057 | 10681 | 1370 | 1370 | 0.12827 | - | - | [14] ** |
| †† | | | 1/6 | | 5026 | 714 | 119 | 0.02368 | | | |
| | | | | | 26300 | 2440 | 2440 | 0.09278 | | | |
| † | TFB | 512/256 | 1 | 1057 | 10900 | 880 | 880 | 0.08073 | - | - | [15] |
| | | | | | 9900 | 330 | 330 | 0.03333 | | | |
| † | TFB | 512/256 | 1 | 1057 | 12031 | 552 | 552 | 0.04588 | 25500 (552 MHz) | 46.12 (552 MHz) | [16] |
| | PB | 512/256 | 1 | 1 | 14841 | 42.9 | 42.9 | 0.0029 | 975.20 (2 MHz) | 487.60 (2 MHz) | this thesis |
| | PB | 512/256 | 1 | 1057 | 16765 | 35.0 | 35.0 | 0.0021 | 1457.38 (2 MHz) | 728.69 (2 MHz) | this thesis |

**d** = decimation exponent; **tput** = throughput; **T/A** = throughput over area
† pipelined implementation
†† serialized implementation
* Karatsuba multiplier was used
** synthesis results only pre place-and-route

Table 5.9: WG-16 implementations comparison (65 nm CMOS)

A comparison of our smallest instance, WG-5, with Grain and Trivium is shown in Table 5.10. While it is not fair to compare area implementations results for different ASIC libraries even using GE metric, the comparison data is still useful. WG-5 (0.065 $\mu m$) with standard initialization phase shows advantage over Trivium (0.13 $\mu m$) in area, maximum frequency and T/A for implementations with 1 to 16 bits / cycle output. Grain (0.13 $\mu m$) seems to provide lower area than Trivium for 1 to 8 bits / cycle options and is similar to WG-5 in this range. The 0.09 $\mu m$ implementations results for Grain and Trivium are

| ASIC library | Cipher | Area | Max freq. (MHz) | Bits / cycle | Max tput (Mbps) | Max T/A (Mbps/$\mu$m$^2$) | Max T/A (Mbps/GE) | Source |
|---|---|---|---|---|---|---|---|---|
| 250 nm | Grain | 119821 $\mu m^2$ | 300 | 16 | 4475 | 0.0373 | - | [20] |
| | Trivium | 144128 $\mu m^2$ | 312 | 64 | 18568 | 0.1288 | - | |
| 130 nm | Grain | 1294 GE | 724.6 | 1 | 724.6 | - | 0.5600 | [21] |
| | | 1678 GE | 694.4 | 4 | 2777.6 | | 2.1465 | |
| | | 2191 GE | 632.9 | 8 | 5063.2 | | 2.3109 | |
| | | 3239 GE | 617.3 | 16 | 9876.8 | | 3.0493 | |
| | Trivium | 2580 GE | 327.9 | 1 | 327.9 | - | 0.1271 | |
| | | 2627 GE | 574.7 | 2 | 1149.4 | | 0.4375 | |
| | | 2705 GE | 473.9 | 4 | 1895.6 | | 0.7008 | |
| | | 2952 GE | 471.7 | 8 | 3773.6 | | 1.2783 | |
| | | 3166 GE | 467.3 | 16 | 7476.8 | | 2.3616 | |
| | | 3787 GE | 350.9 | 32 | 11288.8 | | 2.9809 | |
| | | 4921 GE | 348.4 | 64 | 22297.6 | | 4.5311 | |
| 90 nm | Grain | 4911 $\mu m^2$ | 565 | 1 | 565 | 0.1150 | - | [22] * |
| | | 10548 $\mu m^2$ | 495 | 16 | 7920 | 0.7508 | | |
| | Trivium | 7428 $\mu m^2$ | 840 | 1 | 840 | 0.1131 | - | |
| | | 13440 $\mu m^2$ | 800 | 64 | 51200 | 3.8095 | | |
| 65 nm | WG-5 | 1168 GE | 809.7 | 1 | 809.7 | 0.3334 | 0.6934 | this thesis |
| | WG-5 ** | 1295 GE | 502.3 | 2 | 1004.5 | 0.3730 | 0.7758 | |
| | | 1591 GE | 275.0 | 4 | 1099.8 | 0.3323 | 0.6911 | |
| | | 2245 GE | 137.7 | 8 | 1101.8 | 0.2360 | 0.4908 | |
| | | 3510 GE | 73.8 | 16 | 1180.6 | 0.1617 | 0.3364 | |
| | | 6148 GE | 39.2 | 32 | 1254.6 | 0.0981 | 0.2041 | |
| | WG-5 † | 1521 GE | 517.3 | 2 | 1034.7 | 0.3270 | 0.6801 | |
| | | 1698 GE | 585.8 | 4 | 2343.3 | 0.6634 | 1.3798 | |
| | | 2121 GE | 552.2 | 8 | 4417.4 | 1.0013 | 2.0826 | |
| | | 2833 GE | 532.2 | 16 | 8515.2 | 1.4451 | 3.0059 | |
| | | 4293 GE | 172.4 | 32 | 5515.3 | 0.6176 | 1.2847 | |

**tput** = throughput; **T/A** = throughput over area; 1 GE = 2.08 $\mu m^2$ for 65 nm ASIC

* synthesis results only pre place-and-route

** with fast initialization phase

† with standard initialization phase

Table 5.10: Comparison between WG-5, Grain and Trivium (80-bit key)

given for pre place-and-route. We might assume that their design is synthesizeable at 0.95 density target we used, which would give us good area estimates. However, area in $\mu m^2$ is not that useful for a direct comparison because different ASIC libraries were used.

# Chapter 6

# Conclusion

In this thesis, a parametrized implementation of the WG cipher family operating over polynomial bases was developed and analyzed. Having consistent implementation for different field sizes, field defining polynomials, implementation approaches and technologies allowed us to study their impact on key cipher performance parameters, such as area, maximum frequency, maximum throughput, power consumption and derivative performance metrics like maximum throughput over area and energy per bit of keystream. The results are also compared to existing implementations of WG ciphers.

Two implementation approaches (constant array and discrete components) and three implementation technologies (65 nm CMOS ASIC library, STRATIX IV FPGA and MAX 10 FPGA) were considered in this thesis. Definitions of constant array and discrete components are given in Section 2.3.3. A brief summary of implementation technologies and fundamental differences between them can be found in Section 2.3.1.

Discrete components implementations were studied in Chapter 3. Individual building blocks like multipliers, squarers, exponentiations and trace were developed. The chapter is concluded with the hardware design of DWGP and DWGT, which is a core of WG cipher. Constant array implementations of DWGP and DWGT were provided by Zidaric [9] and were incorporated in the cipher design.

Area profiling of DWGP is the focus of Chapter 4. More than 26000 instances were synthesized and their area pre place-and-route was analyzed. We found that the Hamming weight of the reduction matrix (see Section 4.1.3) can be used effectively to reduce the search space for a field defining polynomial that provides the smallest DWGP area when discrete components implementations are used. It was also discovered that the importance of finding such polynomial for 65 nm CMOS is higher than for FPGAs due to the differences

between the absolute minimum and mean values. Thankfully, using reduction matrix Hamming weight also works well for 65 nm CMOS implementations. It was also shown that this method does not work for constant array implementations on 65 nm CMOS. However, it is not a big problem, because the constant array approach itself is only practical for smaller fields (from WG-5 to WG-10), where exhaustive search is possible. Moreover, constant array implementations on FPGA either have negligible variation in area (MAX 10) or do not have any variation at all (STRATIX IV). In Section 4.3 we summarize the area profiling results and choose field defining polynomials and implementation techniques for complete WG ciphers. It is important to notice exponential increase in DWGP area with respect to field dimension for constant array implementations and quadratic increase in DWGP areawith respect to field dimension for discrete components implementation. It was found that constant array DWGP is smaller for WG-5, WG-7, WG-8 and WG-10, while discrete components DWGP is smaller for WG-11, WG-13, WG-14, WG-16 and above. Also, constant array implementations of DWGT are preferred for all fields that were studied (from WG-5 to WG-16) and extrapolation of the trend allowed us to predict that it will not be the case for DWGT for 17 and above.

Complete WG cipher implementations for all field sizes using 65 nm CMOS ASIC library post place-and-route are shown in Chapter 5. Instances with 1 output bit / cycle as well as multiple output bits per cycle are shown. For the latter, two options with fast and standard initialization phase were given. WG ciphers with normal initialization phase provide lower area, lower power and higher maximum frequency compared to the ones fast initialization. There are exceptions from this rule, for example, WG-5 with 2 or 4 bits / cycle options. Our WG ciphers are comparable with the existing WG-5 and WG-8 implementations. However, when it comes to larger field sizes, like WG-16, highly optimized hardware provides smaller area, lower power and faster maximum frequency than our designs. This is the price we pay for a fully parametrized generic implementation, because such optimization are often specific to a particular field size and field defining polynomial. Also, since pipelining was not considered in this thesis (our datapath is fully combinational), we see a significant advantage of pipelines designs in throughput, maximum frequency and even energy per output bit.

It is also important to talk about synthesis tools limitations, *Synopsys Design Compiler* in particular, that we faced. While it is very likely that all of these issues can be solved with certain flags and settings passed to the tool, we were unable to find the solution at this point. The problems include inability to simplify chains of squarers with Karatsuba multipliers (see Figure 3.6 on page 23), area penalty for constructing trace block with squarers instead of straightforward equation (see Table 3.5 on page 31), area penalty for using constant array DWGP + trace instead of constant array DWGT (see Table 4.3 on

page 59) and loss of area advantage of Karatsuba multipliers in WG ciphers (see Table 5.3 on page 71). Additionally, *Design Compiler* had issues with the Karatsuba multiplier that used recursive instantiation. Before an update to a new version of *Design Compiler*, it worked with only one problem – if a common generic $m$ was used to calculate signal ranges in a port map of a recursive instantiation, the new value of $m$ (that was passed to the child instance) was taken instead of the value from the parent entity. This was not intuitive, because $m$ was used in the range of a signal that belonged to the parent instance. After an update, *Design Compiler* started assigning a single value of the generic $m$ to the parent component and all of its recursive instantiations. It forced us to create a duplicate entity of the multiplier so that two identical entities with different names would instantiate each other.

Future work might include finding optimal LFSR feedback polynomials for each field size, exploring opportunities for hardware optimizations for discrete components implementations and finding a way to increase efficiency and consistency of synthesis tools in minimizing area of larger design instances. Another options is to develop a generic pipelined and a generic serialized implementation with similar degree of parametrization, as well as an extension of the current design to normal basis and tower field bases.

# References


[1] F. Rodríguez-Henríquez, N. Saqib, A. Díaz-Pèrez, and Ç. K. Koç, *Cryptographic Algorithms on Reconfigurable Hardware*. Springer Science + Business Media, LLC, pages 70-72, 2006.

[2] M. A. Hasan, "Lecture Slides: Set 1." ECE 720 (Topic 2) Selected Topics in Cryptographic Computations, University of Waterloo, 2017.

[3] Ç. K. Koç, *Cryptographic Engineering*. Springer Science + Business Media, LLC, pages 105-106, 2009.

[4] R. Lidl and H. Niederreiter, *Encyclopedia of Mathematics and its Applications*, vol. 20, ch. Finite Fields. Cambridge University Press, 1997.

[5] Y. Nawaz and G. Gong, "WG: A Family of Stream Ciphers with Designed Randomness Properties," *Inf. Sci.*, vol. 178, no. 7, pp. 1903 – 1916, 2008.

[6] M. D. Aagaard, G. Gong, and R. K. Mota, "Hardware Implementation of the WG-5 Cipher for Passive RFID Tags," *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013.

[7] K. Mandal, G. Gong, X. Fan, and M. Aagaard, "Optimal Parameters for the WG Stream Cipher Family," *CACR*, vol. 15, 2013.

[8] Adaptive Logic Module (ALM), https://www.intel.com/content/www/us/en/programmable/products/fpga/features/stxiv-alm-logic-structure.html.

[9] N. Zidaric, M. Aagaard, and G. Gong, "Rapid Hardware Design for Cryptographic Modules with Filtering Structures over Small Finite Fields," *International Workshop on the Arithmetic of Finite Fields (WAIFI)*, 2018.

[10] M. Robshaw, *New Stream Cipher Designs - The eSTREAM Finalists*, ch. The eSTREAM Project. Springer-Verlag, Berlin Heidelberg, 2008.

[11] Y. Nawaz and G. Gong, "The WG Stream Cipher," *eSTREAM PHASE 2 Archive*, 2005.

[12] C. Lam, M. Aagaard, and G. Gong, "Hardware Implementations of Multi-output Welch-Gong Ciphers," *CACR*, vol. 01, 2011.

[13] H. El-Razouk, A. Reyhani-Masoleh, and G. Gong, "New Implementations of the WG Stream Cipher," *CACR*, vol. 31, 2012.

[14] H. El-Razouk, A. Reyhani-Masoleh, and G. Gong, "New Hardware Implementations of WG(29, 11) and WG-16 Stream Ciphers Using Polynomial Basis," *CACR*, vol. 2, 2014.

[15] N. Zidaric, M. Aagaard, and G. Gong, "Hardware Optimizations and Analysis for the WG-16 Cipher with Tower Field Arithmetic," - *manuscript submitted for publication.*

[16] X. Fan, N. Zidaric, M. Aagaard, and G. Gong, "Efficient Hardware Implementation of the Stream Cipher WG-16 with Composite Field Arithmetic," *TrustED'13*, pp. 21–34, Nov. 2013.

[17] G. Yang, X. Fan, M. Aagaard, and G. Gong, "Design Space Exploration of the Lightweight Stream Cipher WG-8 for FPGAs and ASICs," in *The 8th Workshop on Embedded Systems Security (WESS'13)*, ACM Press, Article No. 8, September 29, 2013.

[18] M. Hell, T. Johansson, and W. Meier, "Grain - A Stream Cipher for Constrained Environments." http://www.ecrypt.eu.org/stream/ciphers/grain/grain.pdf.

[19] M. Hell, T. Johansson, A. Maximov, and W. Meier, *New Stream Cipher Designs - The eSTREAM Finalists*, ch. The Grain Family of Stream Ciphers. Springer-Verlag, Berlin Heidelberg, 2008.

[20] F. K. Galrkaynak, P. Luethi, N. Bernold, R. Blattmann, V. Goode, M. Marghitola, H. Kaeslin, N. Felber, and W. Fichtner, "Hardware Evaluation of eSTREAM Candidates: Achterbahn, Grain, MICKEY, MOSQUITO, SFINKS, Trivium, VEST, ZKCrypt." http://www.ecrypt.eu.org/stream/papersdir/2006/015.pdf.

[21] T. Good and M. Benaissa, *New Stream Cipher Designs - The eSTREAM Finalists*, ch. ASIC Hardware Performance. Springer-Verlag, Berlin Heidelberg, 2008.

[22] K. Gaj, G. Southern, and R. Bachimanchi, "Comparison of Hardware Performance of Selected Phase II eSTREAM Candidates." http://www.ecrypt.eu.org/stream/papersdir/2007/026.pdf.

[23] C. D. Canniere and B. Preneel, *New Stream Cipher Designs - The eSTREAM Finalists*, ch. Trivium. Springer-Verlag, Berlin Heidelberg, 2008.

[24] M. Aagaard, G. Gong, K. Mandal, M. Sattarov, and N. Zidaric, "WG-lite Design Space Exploration," - *work in progress*.

[25] A. Karatsuba and Y. Ofman, "Multiplication of Multidigit Numbers on Automata," *Soviet Physics-Doklady*, vol. 7, pp. 595–596, 1963.

[26] M. A. Hasan, "Lecture Slides: Set 9." ECE 720 (Topic 2) Selected Topics in Cryptographic Computations, University of Waterloo, 2017.

[27] M. A. Hasan, "Lecture Slides: Set 7." ECE 720 (Topic 2) Selected Topics in Cryptographic Computations, University of Waterloo, 2017.

[28] M. Aagaard, G. Gong, and N. Zidaric, "Hardware Design Automation," - *work in progress*.

# Glossary

**65 nm CMOS** STMicroelectronics 65 nm CMOS ASIC library. 1 gate equivalent (GE) = $2.08 \mu m^2$ iii, 7, 11, 13, 20, 22–24, 31, 35–37, 39, 40, 43, 48–52, 56–61, 63–66, 69, 70, 72, 77–81, 83, 84

**MAX 10** Intel / Altera MAX 10 FPGA, sometimes also referred to as *lstep* board 7, 8, 35, 44–49, 53–58, 61–65, 83, 84

**STRATIX IV** Intel / Altera STRATIX IV FPGA 7, 8, 35, 40–44, 48, 49, 53, 56–58, 61, 62, 64, 65, 83, 84