FPGA-based Evaluation of Cryptographic Algorithms

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt beim Fachbereich Informatik und Mathematik
der Johann Wolfgang Goethe-Universität
in Frankfurt am Main

von

Herr Bernhard Jungk
aus Ulm

Frankfurt 2015
(D 30)

vom Fachbereich Informatik und Mathematik der

Johann Wolfgang Goethe-Universität als Dissertation angenommen

Dekan: Prof. Dr. Uwe Brinkschulte

Gutachter:
Prof. Dr. Steffen Reith
Prof. Dr. Uwe Brinkschulte
Dr. Guido Bertoni

Datum der Disputation: 24.01.2016

Für Maike

Acknowledgements

I would like to thank all the people, without whom, I might not have finished this thesis after all. Foremost, I would like to thank Steffen Reith for his support during all the years since I started this project and for not giving up hope, that I would finish eventually.

Furthermore, I would like to thank the people, with whom I had long or short discussions on the technical or academical aspects of my thesis. This long list of people includes Steffen Reith, Marc Stöttinger, Guido Bertoni, Joan Daemen, Gilles van Assche, Jürgen Apfelbeck, Kris Gaj, Jens-Peter Kaps, Marcin Rogawski, Ekawat 'Ice' Homsirikamol, Christian Wenzel-Benner and Axel Poschmann. I would also thank all the people, which I surely forgot – this is due to my bad memory, not intentional.

A special thanks goes to Steffen Reith, Uwe Brinkschulte, and Guido Bertoni for agreeing to review my thesis, which is – of course – a very important and vital part of the process.

Last but not least, I have to thank all my friends and my family, who supported me throughout all the years, but for whom I had too little time.

Zusammenfassung

Effiziente kryptographische Algorithmen sind ein wichtiger Grundstein für viele neue Anwendungen, wie zum Beispiel das Internet der Dinge (IoT), kooperative intelligente Transportsysteme (C-ITS) oder kontaktlose Zahlungssysteme. Daher ist es wichtig, dass neue Algorithmen mit verbesserten Sicherheitseigenschaften oder speziellen Leistungseigenschaften entwickelt und auch gründlich analysiert werden. Ein Beispiel ist der aktuelle Trend in der Krypto-Forschung zu leichtgewichtigen Algorithmen. Die neuen Entwicklungen erleichtern die Implementierung neuartiger Systeme und ermöglichen auch einen effektiven Schutz von bestehenden Systemen durch eine Anpassung auf den neuesten Stand der Technik. Neben der kryptologischen Analyse, ist die Bewertung von Implementierungs-Aspekten sehr wichtig, damit eine realistische Einschätzung der erzielbaren Leistung möglich ist.

Daher müssen für jeden neuen Algorithmus unterschiedliche Software- und Hardwarearchitekturen entwickelt, implementiert und evaluiert werden. Die systematische Bewertung von Software-Implementierungen von kryptographischen Algorithmen für unterschiedliche Hardware-Architekturen hat in den letzten Jahren große Fortschritte gemacht, zum Beispiel durch den SHA-3 Wettbewerb. Im Vergleich dazu ist die Evaluation für Hardware-Plattformen wie z.B. FPGAs weiterhin sehr zeitaufwendig und fehleranfällig. Dies liegt an vielen Faktoren. Beispielsweise benötigt man vergleichsweise viel Zeit, um die verschiedenen Möglichkeiten auf zahlreichen Zieltechnologien umzusetzen. Auch aufgrund einer Vielzahl an Optionen für die Synthese-Software der FPGA-Anbieter erhöht sich der Aufwand für eine faire Analyse. Ein möglicher Verbesserungsansatz besteht darin, die Bewertung mit einem abstrakteren und theoretischeren Ansatz zu beginnen, um interessante Architekturen und Implementierungen anhand von theoretischen Eigenschaften auszuwählen.

Der erste Hauptbeitrag dieser Arbeit ist die Entwicklung einer neuen abstrakten Bewertungsmethodik, die auf einem theoretischen Modell von getakteten integrierten Schaltungen basiert. Das Modell verbessert das Verständnis von Grundeigenschaften dieser Schaltungen und erleichtert auch die abstrakte Modellierung von Architekturen für einen spezifischen Algorithmus. Wenn mehrere verschiedene Architekturen für den gleichen Algorithmus ausgewertet werden,

ist es auch möglich zu bestimmen, ob ein Algorithmus gut skaliert. Beispielsweise können Auswirkungen einer Verkleinerung des Datenpfades auf die Größe des Speicherverbrauchs analysiert werden. Basierend auf der entwickelten Methodik können einige wichtige Eigenschaften, wie der minimal Speicherbedarf, die minimale Anzahl an Taktzyklen, die Pipeline-Tiefe und ein theoretischer Durchsatz einer spezifischen Architektur systematisch bewertet werden. Damit kann eine grobe Schätzung für die Effektivtät einer Implementierung einer Architektur abgeleitet werden.

Die Performance-Schätzung wird auch durch ein theoretisches Konzept der Optimalität der Anzahl an Taktzyklen theoretisch untermauert. Das Optimalitäts-Kriterium wird in der vorliegenden Arbeit für eine Architektur, die eine Kompressionsfunktion einer Hash-Funktion umsetzt, wie folgt definiert:

Definition 1 *Es sei f eine Kompressionsfunktion $f = r_n \circ \cdots \circ r_2 \circ r_1$. Dann ist eine Architektur für r_i zur Implementierung von f* optimal*, wenn es* keine *Pipeline-Stalls bei der Berechnung von f gibt.*

Dieses Konzept kann man für runden-basierte Algorithmen weiter verfeinern, wenn für die Berechnung von f mehrere quasi identische Runden r_i verwendet werden. Für eine solche Hash-Funktion ist die Anzahl an Taktzyklen einer optimalen Architektur dann wie folgt beschränkt:

Lemma 2 *Es sei b die Zustandsgröße, d die Datenpfadbreite, n die Anzahl der Runden, s die Serialisierungs-Metrik einer Architektur und c eine Konstante. Dann ist die Anzahl von Taktzyklen einer optimalen Architektur für $f = r_n \circ \cdots \circ r_2 \circ r_1$ wie folgt beschränkt:*

$$\frac{s \cdot b \cdot n}{d} \leq \mathrm{cyc}_f(s, b, n, d) \leq \frac{s \cdot b \cdot n}{d} + c,$$

wenn die Anzahl an Taktzyklen für eine Runde r_i wie folgt beschränkt ist:

$$\frac{s \cdot b}{d} \leq \mathrm{cyc}_{r_i}(s, b, d) \leq \frac{s \cdot b}{d} + c.$$

Darüber hinaus wird eine obere Schranke für die Konstanten c in einem Korollar bewiesen. Die Beweise für das Lemma und das Korollar basieren auf der Beobachtung, dass die Datenabhängigkeiten zwischen den Runden eine maximale

Anzahl an Taktzyklen pro Runde erlauben, da ansonsten die Berechnung der Runden-Funktion nicht alle Ausgangs-Bits produziert hat, wenn diese für die nächste Runde benötigt werden und somit würden Pipeline-Stalls entstehen.

Der zweite Beitrag der Dissertation nutzt die Analysemethodik für mehrere Hash-Funktion. Es werden sechs Hash-Funktionen bewertet: BLAKE, Grøstl, Keccak, JH, Skein und Photon. Die ersten fünf Hash-Funktionen sind die Finalisten des SHA-3 Wettbewerb. Das Ziel des SHA-3 Wettbewerbs war es, eine Hash-Funktion mit mindestens der gleichen Sicherheit wie SHA-2 zu finden, die dazuhin eine ähnliche oder bessere Leistung aufweist. Daher haben diese Hash-Funktionen eine hohe Sicherheit als oberstes Design-Ziel und in zweiter Linie eine hohe Performance. Im Gegensatz dazu wurde Photon für leichtgewichtige Anwendungen konzipiert, z.B. RFID-Tags. Dazu wurde auch die Sicherheit reduziert. Photon ist in dieser Arbeit mit einbezogen, um zu analysieren, ob kleinere Varianten des Gewinners des SHA-3 Wettbewerbs – Keccak – auch mögliche Kandidaten für leichtgewichtige Anwendungen sind, wenn die Sicherheit an die von Photon angeglichen wird.

Für jeden Algorithmus ist als erstes die Definition angegeben. Diese wird im nächsten Schritt genutzt, um eine generische Schätzung für den minimalen Speicherplatz abzuleiten, und auch um mögliche Organisationensformen des Speichers zu entwickeln. Die Speicher-Organisation basiert vor allem auf der Breite des Datenpfads, sowie auf einem zusätzlichen möglichen Serialisierung-Faktor. Als nächstes wird die Anzahl von Taktzyklen auf der Grundlage der generischen Speicherorganisation und des Serialisierung-Faktors ermittelt. Das generelle Ziel dabei ist die Entwicklung von Architekturen mit einer optimalen Anzahl von Taktzyklen. Dadurch ist die Ermittelung der nötigen Taktzyklen zumeist nur eine weitere Einschränkung der generischen Grenzen für die Optimalität. Die Diskussion konzentriert sich als nächstes auf verschiedene Möglichkeiten die Runden-Funktion der Hash-Funktion oder von Teilen davon optimal umzusetzen. Auf Basis dieser Ergebnisse werden anschließend einige interessante Architekturen ausgewählt und weiter untersucht.

Die Analyse der einzelnen Architekturen fängt mit einer detaillierten, aber abstrakten Beschreibung einer möglichen Implementierung an. Es wird aber keine Implementierung auf der Registertransferebene entwickelt. Zusätzlich führt die detaillierte Analyse manchmal zu einer Verfeinerung der unteren Grenze der

Speicherabschätzung. Für gewöhnlich bedeutet dies eine Verringerung der Leistung für eine bestimmte Speicherorganisation. Die Ergebnisse der detaillierten Analyse sind am Ende für jeden Algorithmus zusammengefaßt. Dies umfasst mindestens die Schätzung der minimalen Speicheranforderung, die analysierte Pipeline-Tiefe und den theoretischen Durchsatz für lange Nachrichten mit einer festgelegten Taktfrequenz. Diese Ergebnisse lassen eine Einschätzung über die mögliche Leistung der jeweiligen Architekturen zu.

Die Analyse der bewerteten Algorithmen kann wie folgt zusammengefasst werden. Für *BLAKE* basieren die Architekturen auf zwei großen Design-Paradigmen, Serialisierung von BLAKEs Rundenfunktion, sowie Serialisierung von BLAKEs G_i Funktion. Der theoretische Durchsatz bei 100 MHz reicht von 57 bis zu 3657 $^{\text{MBits}}/_{\text{s}}$, mit einer Datenpfadsbreite von 32 bis zu 512 Bit. Der minimale Speicherbedarf erhöht sich leicht für die 32 Bit und 64 Bit Varianten um 64 Bit vom Ausgangswert 1344 Bit auf 1408 Bit. Für einige Varianten ist zusätzlich Pipelining möglich. Dies erhöht den Speicherbedarf weiter, proportional zur Anzahl der zusätzlichen Pipelinestufen.

Für *Grøstl* nutzen die Architekturen zwei Ansätze. Die beiden Permutationen P und Q kann man entweder parallel oder serialisiert berechnen. Zusätzlich sind die Permutationen P und Q leicht serialisierbar. Dabei kann die der Datenpfad um Faktor 8 oder 64 reduziert werden. Dies führt zu einem theoretischen Durchsatz bei 100 MHz zwischen 20 und 2560 $^{\text{MBits}}/_{\text{s}}$, wenn der Datenpfad zwischen 8 und 1024 Bit breit ist. Eine minimale Anzahl von 1536 Bit werden als Speicher benötigt. Der Bedarf steigt auf 1600 Bit für die Architektur mit einer Datenpfadbreite von 8 Bit und 1648 Bits für die 16 Bit breite Variante. Ähnlich wie bei BLAKE ist Pipelining für einige Architekturen möglich. Dadurch steigt der Speicherplatzbedarf ebenfalls proportional zur Anzahl von zusätzlichen Pipeline-Stufen.

JH hat eine einfache und flexible Möglichkeit zur Serialisierung der Runden-Funktion. Diese ist das wichtigste Instrument zur Modellierung der analysierten Architekturen. Jedoch können die Runden-Konstanten leicht berechnet werden, wenn sie benötigt werden, anstatt sie direkt zu speichern. Da dies in der Regel platzsparender ist, muss diese Berechnung ebenfalls berücksichtigt werden. Die Runden-Konstanten können entweder parallel berechnet werden, oder es kann die gleiche Logik verwendet werden wie für die normale Runden-Funktion. Die zweite Option verringert die Größe auf Kosten einer erhöhten Latenz. Der theoretische Durchsatz bei 100 MHz ist 8 $^{\text{MBits}}/_{\text{s}}$ für die kleinste Variante mit Hilfe eines 8 Bit Datenpfads und erreicht 1280 für die parallele Version, mit einer Datenpfadbreite von 1280 Bit. Die minimale Speicher-Anforderungen sind 1792 Bit. Diese untere Schranke gilt für alle Varianten, welche die Runden-Konstanten parallel berechnen. Für alle Varianten, welche die Logik teilen benötigen mehr Speicher. Genauer gesagt, erhöht sich der Speicherverbrauch genau um die Anzahl der Bits der Datenpfadbreite, z.B. werden für einen 8 Bit Datenpfad 1800 Bit benötigt. Pipelining ist auch bei JH für einige Architekturen möglich, diese erhöhen die Speicheranforderungen weiter.

Für *Keccak* werden mehrere unterschiedliche Architekturen analysiert. Die Analyse wird unabhängig von der tatsächlichen Parametrisierung von KECCAK durchgeführt, d.h. alle Ergebnisse hängen vor allem vom Parameter l ab. Dieser Parameter beschreibt die Zustandsgröße $b = 25 \times 2^l$. Die Sicherheitsparameter können auch angepasst werden, d.h. die Rate r, die Kapazität f, sowie die Größe des Hash-Digests n. Vor allem die beiden Parameter r und n haben einen Einfluss auf den Durchsatz. Die analysierten Architekturen nutzen eine dreidimensionale Interpretation des KECCAK-Zustands. Dies ermöglicht viele Serialisierungs-Möglichkeiten. Die erste Option ist eine parallele Architektur. Alternativen sind Bit-, Lane-, Column-, Row-, Sheet-, Plane- und Slice-weise Implementierungen. Für die (bald) standardisierte Version von KECCAK mit $n = 256$, sind die Parameter $b = 25 \times 2^6 = 1600$ und $r = 1088$ festgelegt. In dieser Einstellung befindet sich der theoretische Durchsatz im Bereich von $2,72$ bis zu 4352 $^{\text{MBits}}/_{\text{s}}$ bei 100 MHz und der minimale Speicherbedarf ist im Bereich von 1600 bis zu 4800 Bit. Die vielseitigste Architektur ist die Slice-weise Variante. Sie kann entweder nur ein Slice pro Taktzyklus verarbeiten oder eine in Zweierpotenzen steigende Anzahl von Slices, d.h. maximmal können 2^l Slices parallel verarbeitet werden. Die größte Variante ist jedeoch eine parallele Architektur, die leicht durch bessere parallele Implementierungen geschlagen wird.

Für *Skein* werden drei verschiedene Ansätze untersucht. Der erste ist eine einfache parallele Implementierung. Diese Architektur kann zu einer ausgerollten Architektur verändert werden, welche möglicherweise den Durchsatz verbessern kann, indem die Komplexität der Implementierung der Rundenfunktion reduziert wird. Als drittes wurden Möglichkeiten zur Serialisierung von Skein untersucht. Der theoretische Durchsatz bei 100 MHz reicht von 89 bis zu 2844 MBits/s. Die Datenpfadbreite wurde von 64 bis zu 512 Bit untersucht. Die minimale Speicher-Voraussetzungen sind 1280 Bit, aber erhöhen sich leicht für die kleinste Variante zu 1344 Bit. Weiterhin erhöht sich der Speicherbedarf für Pipeline-Varianten, wobei der Speicher wie bei den anderen untersuchten Hash-Funktionen proportional zur Anzahl der Pipeline-Stufen ansteigt.

Die *Photon* Hash-Funktion ist eine Familie von fünf ähnlichen Funktionen mit unterschiedlichen Sicherheitsstufen. Allerdings kann die Architekturdiskussion unabhängig von der exakten Funktion durchgeführt werden, ähnlich zu der Analyse von KECCAK. Photon kann auf zwei unterschiedlichen Wege serialisiert werden. Bei der ersten wird die Berechnung der Rundenfunktion serialisiert, indem die Datenpfadbreite reduziert wird. Außerdem kann die sogenannte MixColumnsSerial Funktion auf eine spezielle Art und Weise serialisiert werden, da die Funktion eine spezielle Struktur aufweist. Für die größte Variante von Photon mit der höchsten Sicherheit ergibt die Analyse der vorgeschlagenen Architekturen einen theoretischen Durchsatz der bei 100 MHz von $1,23$ bis zu 266 MBits/s gefächert ist, wobei die Datenpfadbreite von 8 bis zu 288 Bit reicht. Die minimalen Speicheranforderungen sind 288 Bit und erhöhen sich für die kleineren Implementierungen auf Grund der Datenabhängigkeiten in der Rundenfunktion.

Der dritte Beitrag der Arbeit besteht aus mehreren kleinen und mittleren Implementierungs-Ergebnissen. Zunächst werden Ergebnisse für Architekturen von den SHA-3 Finalisten BLAKE, Grøstl, JH, KECCAK und Skein gezeigt. Sie ergänzen den aktuellen Forschungsstand zur Implementierung dieser Algorithmen. Von den fünf Algorithmen haben alle außer Skein eine relativ hohe Performanz, während Skein weit abgeschlagen ist. Eine weitere Untersuchung konzentriert sich auf kleine und mittlere Implementierungen des SHA-3 Siegers KECCAK. Dazu gehören auch nicht standardisierte Varianten mit einem kleineren Zustand. Dies ermöglicht auch kleinere Implementierungen mit einer reduzierten Sicherheitsstufe. Diese kleineren Versionen werden mit ersten FPGA-Ergebnissen für die Photon Hash-Funktion verglichen. Eine wesentliche Erkenntnis davon ist, dass KECCAK auch für FPGA-Anwendungen mit beschränktem Ressourcen-Bedarf prinzipiell sehr wettbewerbsfähig ist.

Schlüsselwörter. Kryptographie, Hash Funktionen, SHA-3, FPGAs, Leichtgewichtige Kryptographie

Contents

List of Figures

List of Tables

List of Algorithms

Part I

Introduction and Motivation

Chapter 1

Introduction

1.1 Motivation and Background

In a world that depends more and more on electronic infrastructure and computer networks, the importance of the security of these systems increases almost every day. For example, in the near future, a growing number of vehicles will have a wireless network connection, which is not just an integrated Internet connection for cars. Many of such new wireless applications will require access to the on-board network and thus, the safety of passengers will be at risk, if a malicious attacker can get access through these external interfaces, e.g. the car-2-car communication system [Con07].

Another prominent example is the advent of the so called Internet of Things (IoT), which is a label for the continuing development, that an increasing number of everyday devices is connected to the Internet [Ash09]. This idea goes far beyond the usual Internet devices such as computers, tablets or smart phones and a large part of of these devices will use wireless network technologies. Almost all of these applications need to be protected by effective security mechanisms, because many will have an influence of the privacy or the safety of people [AIM10]. Many more such modern applications of networking technology exist and a lot of them are in need of tailored security solutions.

1.1.1 Trade-Offs in Cryptography

Such security mechanisms are a major cost factor for many applications and thus, a lot of the current research is focused on several areas in this field.

Of course, the top priority of a security protocol is a proper protection and therefore, the correctness of such architectures and also of implementations has to be verified. However, the security level has to be balanced with other factors such as interoperability, power consumption and costs, and hence, the research has a very broad focus. Interoperability for example is usually a problem if new systems have to cooperate with an existing infrastructure. In this case, older and sometimes less efficient protocols have to be implemented. This can be in contradiction to the desirable security, cost or performance goals.

In this thesis the current state of the art is evolved by investigating several efficient architectures of modern hash functions for midrange and lightweight applications. Such implementations are often necessary, because there is a great pressure on the cost and thus, it is necessary to find a compromise between the performance and the costs and therefore, architectures which have a reasonable throughput with a reduced area are also important beside the extreme cases of high-throughput and ultra-lightweight architectures.

Many other research efforts also try to improve the state of the art for new applications on a more general level. For example Bernstein developed a more efficient elliptic curve representation and thus, the cost of asymmetric cryptography can now be reduced for new systems [Ber06]. Further research pushes this limit further. In [BKL+07] PRESENT, a new ultra-lightweight block cipher has been proposed, or the Photon hash function has been developed [GPP11]. The two latter algorithms are aiming at applications that require a very low area footprint, such as RFID tags. Many more new algorithms and architectures of existing algorithms were proposed in the last years and the cryptographic research community is busy developing new ideas to further push the limits.

1.1.2 SHA-3 Competition

As already mentioned, one of the top priorities is to achieve a sufficient level of protection at minimum cost. Therefore, new systems and applications not only drive the development of new cryptographic algorithms, but also the advances in cryptanalysis. For example, recent attacks produced collisions in MD5 [WFLY04], and also SHA-1 was broken in the sense, that there are known attacks that are theoretically faster than brute force [WYY05]. These attacks and concerns about possible weaknesses in SHA-2 triggered the start of the SHA-3 competition [Kay07]. In parallel, some new improved attacks on reduced versions of SHA-2 were published [SS08, IS09]. Yet, no substantial breakthrough in the cryptanalysis of SHA-2 was made. The SHA-3 competition was concluded at the end of 2012 with the announcement that Keccak will become the next standardized hash function SHA-3.

The SHA-3 competition acted as a catalyst for the research in the area of hash functions, most notably, the five finalists of the competition. An overview of the research which was conducted during the competition can be found in [CPB+12]. The theoretical security was the most important aspect, closely followed by the importance of the performance of the candidates. Therefore, a lot of effort was invested into the performance evaluation for many target platforms from low-end processors [WG10] to high-end hardware implementations [GSH+12]. For the midrange segment the performance is also very important, because reducing the hardware area or the memory usage of software implementations improves the cost effectiveness. The throughput-area trade-off for FPGAs was for example investigated for the SHA-3 winner Keccak by the author of this thesis in [JS13]. It is also one of the important aspects of the present thesis.

1.1.3 Field-Programmable Gate Arrays

The evaluation in this thesis is based on Field-Programmable Gate Arrays (FPGAs), which are based on a somewhat hybrid technology. An FPGA can be seen as an integrated circuit (IC), because such an FPGA works in many ways like a classical IC. However, FPGAs also have aspects similar to software, because they can be (re-)programmed in the field and thus, it is usually possible to update and change the circuit without replacing the hardware, which is usually impossible with application specific ICs (ASIC).

This has several advantages. First of all, FPGAs are a cheap development and prototyping platform for the development of ICs. Hence, for hardware implementations, one does not have to fabricate an ASIC right away, and yet it is possible to test a prototype in a form that is closer to the final hardware than many simulations. Furthermore, if the implementation is only used for a low volume production, an FPGA may be cheaper than a similar ASIC implementation, because of the high fixed cost to fabricate ASICs.

One other important advantage is, that it is possible to fix problems with the implementation or to update the FPGA implementation to include new functionality without actually replacing the hardware. For example, this has important applications in the aerospace and avionics markets, where it is sometimes expensive or impossible to replace the hardware and much cheaper to do a software update.

1.1.4 Evaluation Methodologies

As it has been shown in the last sections, it is important to evaluate efficient hardware architectures for cryptographic algorithms. Unfortunately, a fair performance evaluation is difficult, because there are many possible pitfalls. For example, there are a lot of implementation variants and an evaluation between different implementations is usually multi-dimensional. For software implementations, these dimensions are for example throughput, RAM and ROM usage or the power consumption. However, using a fixed software interface and platform, it possible to get a good and meaningful evaluation of different implementations, using a systematic methodology [BL12].

This changes for hardware implementations considerably, because even more variations are possible. Hence, an evaluation is tedious and error prone, because of many factors [Dri09]. Among these are:

- The target technology varies. For FPGAs, it does not only include the type of the FPGA, but evaluation results also change for the exact target FPGA (e.g. two different Xilinx Virtex-5 FPGAs). It is also a huge difference, if DSPs, BRAMs or other special hardware primitives are used. All of these factors are often very different between two evaluations.
- The parameters used in synthesis, map and place and route have a considerable impact on the results.
- The I/O interface has a high impact for lightweight implementations.

Another big difference between software and hardware implementations is, that the evaluation times for hardware developments are usually much longer and hence, it is more difficult to setup a common benchmarking system which would standardize the evaluation. Furthermore, there are usually many possible optimization strategies and optimization goals. Thus, even if an evaluation is sound in itself, comparing results with another evaluation is almost always slightly wrong. Therefore, a fair comparison between more than one architecture of an algorithm should be based on a more abstract design approach to select good candidate architectures before implementation.

A methodology which takes a step towards such a more abstract design process was proposed by the author of this thesis in [JS13]. It analyzes theoretical properties of a proposed architecture before the actual implementation. In this thesis, this last methodology is used as a basis and an extended and revised version is discussed in more detail and later applied to several cryptographic hash algorithms. The extended version is based on an abstract circuit model, and thus has stronger theoretical foundations than many other methodologies.

1.2 Thesis Organization

The thesis is organized in three parts. The first introductory part (Part I) gives a motivation and background information on the topic of the thesis. In the second part, theoretical foundations are provided. The third part contains the bulk of the evaluation of the algorithms, including the theoretical and the practical evaluation and a thorough evaluation of each.

The second part (Part. II) contains a treatment of a tailored Boolean circuits model based on the standard model, commonly used in complexity theory (Ch. 2). The main change to the model is the inclusion of memories, which for example facilitates a formal reasoning about the memory consumption of circuits. It is later used as the basis of the systematic methodology (Ch. 5).

The introduction to finite fields is followed by an introductory chapter about hash functions (Ch. 3). It defines the basic notion of a hash function, security properties and also some important iteration modes. The iteration modes play a significant role, because the different constructions usually have a major influence on the area consumption, e.g. the Davies-Meyer mode requires a larger memory than a plain Merkle-Damgård design.

The last chapter in Part II discusses several aspects of hardware design (Ch. 4). In particular, the RTL design methodology is described and later low- and high-level optimization strategies are introduced. Additionally, some aspects of FPGAs are described, which are important as several design decisions in the evaluation are based on them.

In the third part, the main evaluation is described (Part. III). The first chapter in this part describes the systematic evaluation methodology (Ch. 5), which is used in the next chapter to evaluate six different hash functions based on a theoretical treatment (Ch. 6). For each algorithm, several architectures are investigated. For each architecture properties such as minimum memory consumption, number of clock cycles and a theoretical throughput are shown. Ch. 6 is concluded with a discussion of the theoretical results.

The last chapter in this part gives an evaluation of the concrete implementations, a comparison with third-party results and a discussion of the results (Ch. 7).

The appendix contains additional material. The first appendix (Appx. A) introduces the theory of finite fields and the composite fields approach. This theory is used in the optimization of the area consumption of the AES S-box used by both Grøstl and the largest variant of the Photon hash function. It is in the later chapters investigated, if choosing a different representation compared to the previous publications is more efficient for FPGAs, than the published variants, which were focused on ASICs. Furthermore, the appendix contains additional post place and route results of the implementations (Appx. B).

1.3 Published Material

Parts of this thesis are extension or refined versions of previous published material of the author. In particular, the thesis contains results of the following peer-reviewed papers:

- The revisited optimization of the AES S-box for was published in [JR10b] together with an implementation of the second round version of Grøstl.

- The implementations of the SHA-3 finalists were gradually improved. The first results were presented at the Ecrypt II Hash Workshop [Jun11], followed by the paper [JA11], which presented implementation results for all finalist. Furthermore, a refined version was published at the final SHA-3 conference [Jun12].

- After the announcement of the winning algorithm KECCAK, the implementations of this algorithm have been further optimized and the results published in [JS13]. Additional improvements and also results for ASICs have been published at the SHA-3 2014 workshop [JSH14].
- A comparison between KECCAK, Photon and the hash function Spongent has been presented at the ReConFig '14 conference [JLH14]. However, Spongent is not analyzed in the present thesis.

1.4 Notations

The following notations are used throughout the thesis. Some other notations, that are only important in a local context are introduced in the text.

Table 1.1: Mathematical notations.

Notation	Explanation
$\mathbb{Z}_2$	Representation of a bit, i.e. $\mathbb{Z}_2 =_{\text{def}} \{0,1\}$ or a finite field $\mathbb{Z}_2 =_{\text{def}} \mathbb{F}_2$.
$\mathbb{Z}_2^m$	$\mathbb{Z}_2^m$ is the set of bit strings of length m, i.e. $\mathbb{Z}_2^m =_{\text{def}} \{0,1\}^m$.
$\mathbb{Z}_2^{\geq 0}$	$\mathbb{Z}_2^{\geq 0}$ is the set of bit strings of arbitrary length, i.e. $\mathbb{Z}_2^{\geq 0} =_{\text{def}} \bigcup_{i\geq 0} \mathbb{Z}_2^i$.
$\lfloor x \rfloor_n$	If $x \in \mathbb{Z}_2^m$ and $m > n$, then the value of $\lfloor x \rfloor_n$ consists of the n least significant bits of x, i.e. $\lfloor x \rfloor_n \in \mathbb{Z}_2^n$.
$\lvert x \rvert$	The length of $x \in \mathbb{Z}$, i.e. if $x \in \mathbb{Z}_2^{\geq 0}$, then $x \in \mathbb{Z}_2^i$ for some i and $\lvert x \rvert = i$.
$\lvert x \rvert_n$	The length of x encoded in a binary encoding using n bits, i.e. $\lvert x \rvert_n \in \mathbb{Z}_2^n$.
$a \vert\vert b$	Concatenation of two elements. If $a \in M$ and $b \in N$, then $a\vert\vert b \in M \times N$ and if $M = \mathbb{Z}_2^m$ and $N = \mathbb{Z}_2^n$, then $a\vert\vert b \in \mathbb{Z}_2^{(m+n)}$ and $a\vert\vert b = \{a_1, \ldots, a_m, b_1, \ldots, b_n\}$. Furthermore, if the binary representations of a and b are interpreted as $a, b \in \mathbb{N}$, then $a\vert\vert b =_{\text{def}} a \times 2^n + b$.

x_i	If x consists of several concatenated elements, then x_i refers to the ith element. For example, if $x = x_1 \|\| x_2$ then x_1 refers to the first part of x and x_2 to the second.
$x[i]$	Equivalent to x_i. The notation is sometimes used, when it is more convenient, e.g. for indexing cells in a matrix.
$\vec{a}_1^n$	The notation $\vec{a}_1^n$ is used to abbreviate enumerations, i.e. $\vec{a}_1^n =_{\text{def}} a_1, \ldots, a_n$.

Part II

Foundations

Chapter 2

Boolean Circuits with Memory

2.1 Introduction

Throughout the following chapters, the argument will refer to the Boolean circuits model, which is a general abstract model for integrated circuits. In the literature, this model is mainly used in the field of complexity theory, where it is for example applied to characterize the **NC** hierarchy [Coo79], which is believed to capture the notion of efficient parallelizable algorithms. Additionally, many complexity theorist believe, that no efficient parallel algorithms for **P**-complete problems exist. This would be the case, if the still unproven assumption $\mathbf{NC} \neq \mathbf{P}$ is true [GHR95].

The usefulness of the model comes from its abstract view on circuits, which – in complexity theory – is readily exploited to get a better understanding of algorithms and complexity classes [Vol99]. The model is also interesting to analyze how to efficiently implement algorithms. Hence, this model is a good starting point for a more formal analysis of cryptographic algorithms towards efficient hardware implementations.

The usual model of Boolean circuits defines a circuit as an acyclic graph without any memory elements, because simple models are usually easier to analyze. In complexity theory this is particularly useful, because it is much easier to reason about and thus to prove certain properties. However, for the analysis in the following chapters this is inadequate, because the evaluation goal is an estimate on the performance of concrete implementations for several cryptographic algorithms. Therefore, the notion of *Boolean Circuits with*

Memory is introduced and tailored to the current use case. This new model is defined in a two step approach. First, an adapted version of the original model of Boolean circuits is introduced, and then it is extended in a second step to use clocked memory primitives.

The remainder of this chapter is organized as follows. The new model is introduced in several parts. First, a restricted version of the standard Boolean circuits model will be defined in Sec. 2.2. Afterwards, the model will be extended to Boolean circuits with memory (Sec. 2.3), followed by a possible modeling of random access memories in Sec. 2.4. Finally several generally useful complexity measures are defined (Sec. 2.5).

2.2 Boolean Circuits

The definition of Boolean circuits (Def. 2.3, Def. 2.5) used in the further sections and chapters is based on the definition by Vollmer [Vol99]. In the literature, several slightly different and sometimes less formal definitions may be found, e.g. in [Sav97, AB09, Sip96]. The main difference from the herein defined model to the reference definition is, that families of Boolean functions are not included in the basis of a circuit, because these infinite objects are only needed to model gates with unbounded fan-in [Vol99]. For a theoretical treatment, these gate types are an important abstraction. However, in reality no such gates exist and thus, for an abstract, yet practical evaluation of algorithms, they should not be used in a more realistic setting.

An example of such a circuit is depicted in Fig. 2.1. The figure shows a small adder for three natural numbers which have two bits each, i.e. $x, y, z \in \mathbb{Z}_4$ and

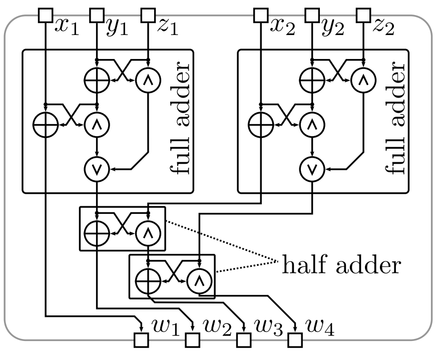


Figure 2.1: Adder for three two-bit natural numbers.

$0 \leq w \leq 9$. This adder is a variant of a carry save adder, which is an efficient circuit for adding multiple values. Other efficient adders, which are suitable to add only two natural numbers usually use a variation of a prefix adder circuit [Kor01, Sav97].

One of the key ingredients for a Boolean circuit is a *Boolean function.* Such functions are the nodes in an acyclic graph, which describes a circuit. Well known Boolean functions are for example $\wedge$ (AND), $\vee$ (OR), or $\oplus$ (XOR).

Definition 2.1 *A* Boolean function *is a function $f : \mathbb{Z}_2^n \to \mathbb{Z}_2$ for some $n \in \mathbb{N}$.*

Sometimes a set of Boolean functions is combined to form a vectorial Boolean function:

Definition 2.2 *A* vectorial Boolean function *is a function $f : \mathbb{Z}_2^n \to \mathbb{Z}_2^m$ for some $n, m \in \mathbb{N}$.*

In complexity theory, it is interesting to limit the set of basic Boolean functions which can be used to implement an algorithm. Then the influence of this limited set of operations on various parts of complexity theory is examined. This investigation leads to a lot of interesting theoretical results, e.g. to a separation of the complexity classes $\mathbf{AC}^0$ and $\mathbf{NC}^1$ by proving that **Parity** is not in $\mathbf{AC}^0$ but in $\mathbf{NC}^1$ [Sav97].

Additionally, Boolean functions may be used to model technology specific gate types directly. Two examples are lookup-tables of modern FPGAs or ASIC libraries. The lookup-tables of FPGAs usually may be used to implement any Boolean function with a fixed number of input variables [Xil12c]. ASIC libraries are similar but more restricted and usually support at least several two-input Boolean functions as gate types efficiently [DNRH07]. Note, that the technology always limits the number of possible Boolean functions to a finite set. The general idea of such a finite set of Boolean functions is captured in the notion of a basis which is defined in Def. 2.3.

Definition 2.3 (Def. 1.5 [Vol99]) *A* basis *$\mathcal{B}$ is a finite set of Boolean functions.*

With such a basis, the syntactic structure of a Boolean circuit C is defined inductively as an acyclic graph as follows:

Definition 2.4 (Def. 1.6 [Vol99]) *Let $\mathcal{B}$ be a basis, then a Boolean circuit with inputs $\vec{x}_1^n$ and outputs $\vec{y}_1^m$ is a tuple:*

$$C = (\mathcal{B}, V, E, \alpha, \beta, \gamma),$$

where (V, E) is a finite directed acyclic graph, $\alpha : E \to \mathbb{N}$ is an injective function, $\beta : V \to \mathcal{B} \cup \{\vec{x}_1^n\}$, and $\gamma : V \to \{\vec{y}_1^m\} \cup \{*\}$ such that the following conditions hold:

- *If $v \in V$ has in-degree 0, then $\beta(v) \in \{\vec{x}_1^n\}$ or $\beta(v)$ is a 0-ary function (constant) from $\mathcal{B}$.*
- *If $v \in V$ has in-degree $k > 0$, then $\beta(v) \in \mathcal{B}$ and $\beta(v) : \mathbb{Z}_2^k \to \mathbb{Z}_2$. These vertices are called* computational gates*.*
- *For every i, $1 \leq i \leq n$, there is at most one node $v \in V$ such that $\beta(v) = x_i$. Hence, each x_i is an* input gate*.*
- *For every j, $1 \leq j \leq m$, there is exactly one node $v \in V$ such that $\gamma(v) = y_j$. Hence, each y_j is an* output gate*. For every gate $v \in V$ which is not an output gate, i.e. $\gamma(v) \notin \{\vec{y}_1^m\}$, $\gamma(v) = *$ holds.*

The function α enumerates the edges of the acyclic circuit graph. This enumeration is used in the definition of the circuit evaluation to define the order in which the outputs of predecessor gates are mapped to inputs of a successor gate. Of course, this ordering is only relevant, if the Boolean function $f(v) \in \mathcal{B}$ implemented by the respective gate $v \in V$ is not commutative.

After the definition of the structure (syntax) of the circuit, the computation (semantic) of the circuit has to be defined. In general, a circuit C computes a function $f_C : \mathbb{Z}_2^n \to \mathbb{Z}_2^m$ as defined in the following two definitions:

Definition 2.5 (Def. 1.7 [Vol99]) *Let $C = (\mathcal{B}, V, E, \alpha, \beta, \gamma)$ be a Boolean circuit. Then for every $v \in V$ a function $\mathrm{val}_v : \mathbb{Z}_2^n \to \mathbb{Z}_2$ is defined, where n is the number of arguments to the circuit. Let $\vec{a}_1^n$ be the input value functions to the circuit, then:*

Induction Basis:

- *If $v \in V$ has fan-in 0 and $\beta(v) = x_i$ for some i, $1 \leq i \leq n$, then $\mathrm{val}_v(\vec{a}_1^n) =_{\mathrm{def}} a_i$.*

- *If* $v \in V$ *has fan-in* 0 *and* $\beta(v) = b$*, for some* $b \in \mathcal{B}$*, then* b *must be a* 0*-ary function (constant) and hence* $\mathrm{val}_v(\vec{a}_1^n) =_{\mathrm{def}} b$*.*

Induction Step:
Let $v \in V$ *have fan-in* $k > 0$ *and let* $\vec{v}_1^k$ *be the predecessor gates and the order of the predecessors be* $\alpha((v_1, v)) < \cdots < \alpha((v_k, v))$*, then*

$$\mathrm{val}_v(\vec{a}_1^n) =_{\mathrm{def}} \beta(v)(\mathrm{val}_{v_1}(\vec{a}_1^n), \ldots, \mathrm{val}_{v_k}(\vec{a}_1^n))$$

Definition 2.6 *For* $1 \leq i \leq m$*, let* $y_i \in V$ *be the output gates, then the function computed by the circuit* $C = (\mathcal{B}, V, E, \alpha, \beta, \gamma)$

$$f_C : \mathbb{Z}_2^n \to \mathbb{Z}_2^m$$

is for all $\vec{a}_1^n \in \mathbb{Z}_2$ *given by*

$$f_C(\vec{a}_1^n) =_{\mathrm{def}} (\mathrm{val}_{y_1}(\vec{a}_1^n), \ldots, \mathrm{val}_{y_m}(\vec{a}_1^n)).$$

2.3 Memory and Timing

Adding the notion of memories to a Boolean circuit makes it necessary to introduce a concept of time, because storage is obviously only useful, if the computation advances with some notion of progress. There are basically two ways to model this concept – synchronous and asynchronous circuits. However, the theoretical simpler model is the synchronous approach, because the depth of a combinational circuit is only a very poor approximation of the timing behavior of an asynchronous circuit. Modeling the propagation delay in a more realistic setting requires a sophisticated model and in turn, such a model is harder to analyze. For example, glitches may pose a major problem for the correctness of asynchronous designs [JNB99]. Glitches are a phenomenon were the output of a gate toggles multiple times before reaching a stable and correct state and usually only happens if the input signals to a gate do not arrive synchronously. Therefore, the discussion of asynchronous circuits is not considered in this thesis.

An example for a typical Boolean circuit with memory is depicted in Fig. 2.2. The adder circuit in this figure is similar to the previous adder (Fig. 2.1). However, there are two main differences. When implementing the first adder in

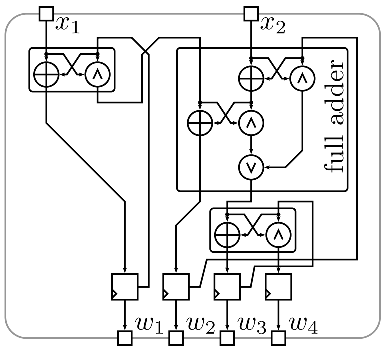


Figure 2.2: Adder for multiple two-bit natural numbers.

a real world circuit, all three values are added in one clock cycle. In the latter case, adding three values takes three clock cycles and it is additionally possible to add an infinite number of two bit numbers (addition is modulo 16 in this case). Other important differences are the size and the longest critical path (or depth) of both circuits.

Real world synchronous circuits usually use a clock signal to control the storage of the current output signals of combinational circuits in the registers. That means, that all registers operate time synchronously, i.e. on the rising or falling edge of the clock signal. However, modern sub micron VLSI circuits suffer from the propagation delay of the clock signal itself, i.e. the registers are often not triggered strictly time synchronous. Such timing issues may have a high influence on the performance. The overall impact depends on the routing and distribution of the clock signal, because a synchronous circuit must still synchronize on a global clock signal and therefore, the distribution of the clock signal is one of the important major problems in modern VLSI design [Fri01].

A high level view which captures this idea of synchronous circuits is the programming of circuits at the register transfer level (RTL), i.e. the programmer specifies registers and the combinational circuits between these registers. This model is very different to a classical sequential programming style, because the computation of the combinational circuits is inherently fully parallel. The *Boolean circuits with Memory* model used in this thesis is based on the RTL methodology. Hardware design using this methodology is described in more detail in Sec. 4.2.

The main reason to introduce the formal model of synchronous circuits in this thesis is a step towards a more formal evaluation approach for hardware implementations of cryptographic algorithms. FPGAs are the main target platform and thus, the new theoretical model is defined to support features, which are important to this technology. In particular, the memory architecture of FPGAs is interesting, because there are two different kinds of synchronous memories, which differ considerably in the area consumption on the one hand and the usage on the other hand:

- *Registers* are usually implemented as flip-flops. Values stored in these memories may be used as an input for any number of computational gates in parallel. However, only the output of a single gate may be written into each individual bit of a register in each clock cycle.
- *RAMs* are random access memories, which are divided into memory cells of fixed depth and width. They are in principle similar to registers, however, only a fixed number of these memory cells can be written or read in each clock cycle. The implementation of a random access memory cell usually differs considerably from a flip-flop implementing a register, which saves a lot of hardware area for big memories.

The following definition captures the notion of a simple memory element, storing exactly one bit. In the following, the term register and memory element are often used interchangeable. From a more formal point of view, a memory element is defined to be the storage for exactly one bit, whereas a register may have a width of one or more bits.

RAMs are not included in the definition. Yet, they can be modeled using some Boolean circuitry and several registers as will be shown in Sec. 2.4. Informally, the value of a memory element at a point in time t is identical to the input which was stored at the previous point in time $t-1$ or the initial value 0 at $t = 0$ or more formally:

Definition 2.7 *If* $a : \mathbb{N} \leftarrow \mathbb{Z}_2$ *is the input function of the* memory element r*. The initial output of the register* r *for* $t = 0$ *is defined as* $\mathrm{val}_r(0, a) =_{\mathrm{def}} 0$ *and the other outputs for* $t \geq 1$ *as* $\mathrm{val}_r(t, a) =_{\mathrm{def}} a(t-1)$.

This definition of a memory element is in a theoretical sense sufficient, because building larger memories, i.e. registers with more than one bit storage capacity, does not impact any basic complexity measures. In reality, grouping several registers or RAMs is often more efficient, because it is computationally easier to place and route several identical primitives in close proximity in an efficient way using a regular structure. Additionally, special RAM resources usually use a different technology then registers, which is more efficient than registers for storing many bits, with the disadvantage of removing the possibility of parallel read and write accesses. For example, many FPGAs provide RAM block resources, which may be more efficient for use cases, where a lot more bits have to be stored compared to a typical register and the bits are accessed sequentially.

However, for an abstract analysis, narrowing the definition of a memory element to a single bit simplifies the model and thus, it is later easier to analyze without restricting the computational power, because a user of the model may create concrete instances of several elementary elements and then group these logically into larger memories.

The circuit definition is very similar to the case without memory. The main differences are the addition of memory and that the acyclic property of the circuit graph has to be dropped, because otherwise the memories do not add a lot of interesting theoretical properties to the model. However, from a more practical point of view, memories are also interesting for clocked circuits in an acyclic setting, because of possible higher clock frequencies using pipelining.

In the model such cycles are only allowed, if each cycle contains at least one memory element. Again, this is an abstract and restricted model compared to real world circuits, where arbitrary cycles are possible [Mal93]. Yet, the model strives for simplicity and therefore disallows these kinds of cycles.

Definition 2.8 *Let $\mathcal{B}$ be a basis, and r the gate type designating a memory element, then a* Boolean circuit with Memory *(BCM) with inputs $\vec{x}_1^n$ and outputs $\vec{y}_1^m$ is a tuple:*

$$C = (\mathcal{B}, r, V, E, \alpha, \beta, \gamma),$$

where (V, E) is a finite directed graph, $\alpha : E \to \mathbb{N}$ is an injective function, $\beta : V \to \mathcal{B} \cup \{r\} \cup \{\vec{x}_1^n\}$, and $\gamma : V \to \{\vec{y}_1^m\} \cup \{*\}$ such that the following conditions hold:

- *If $v \in V$ has in-degree 0, then $\beta(v) \in \{\vec{x}_1^n\}$ or $\beta(v)$ is a 0-ary function (constant) from $\mathcal{B}$.*

- *If $v \in V$ has in-degree $k > 0$, then $\beta(v) \in \mathcal{B}$ or $\beta(v) = r$. These vertices are called* computational *or* memory elements*, respectively.*

- *For every i, $1 \leq i \leq n$, there is at most one node $v \in V$ such that $\beta(v) = x_i$. Hence, each x_i is an* input gate.

- *For every j, $1 \leq j \leq m$, there is exactly one node $v \in V$ such that $\gamma(v) = y_j$. Hence, each y_j is an* output gate*. For every gate $v \in V$ which is not an output gate, i.e. $\gamma(v) \notin \{\vec{y}_1^m\}$, $\gamma(v) = *$ holds.*

- *For all subsets of nodes $V_{cyc} \subseteq V$ with $v_1, \dots, v_k \in V_{cyc}$ forming a cycle $(v_1, \dots, v_k, v_1)$, at least for one node $v \in V_{cyc}$, $\beta(v) = r$ holds.*

The evaluation of a Boolean circuit with memories is also very similar to the simpler case without memories. However, because of the concept of time necessary to describe the step-wise advancing computation and the cyclic nature of a circuit with memory, the evaluation is slightly different, i.e. the inductive definition is over the graph and the time in parallel.

Definition 2.9 *Let $C = (\mathcal{B}, r, V, E, \alpha, \beta, \gamma)$ be a Boolean circuit with memory. Then for every $v \in V$ a function $\mathrm{val}_v : \mathbb{N} \times \{\vec{a}_1^n\}$ is defined, where n is the number of arguments to the circuit. Let $\vec{a}_1^n$ be the n input value functions $a_i : \mathbb{N} \to \mathbb{Z}_2$, where $a_i(t)$ is an input to the circuit at time $t \in \mathbb{N}$, then:*

Induction Basis:

- *If $v \in V$ has fan-in 0 and $\beta(v) = x_i$ for some i, $1 \leq i \leq n$, then for all points in time $t \in \mathbb{N}$, $\mathrm{val}_v(t, \vec{a}_1^n) =_{\mathrm{def}} a_i(t)$, i.e. in every clock cycle a new set of inputs is applied to the input gates.*

- *If $v \in V$ has fan-in 0 and $\beta(v) = b$, for some $b \in \mathcal{B}$, then b must be a 0-ary function (constant) and hence, $\mathrm{val}_v(t, \vec{a}_1^n) =_{\mathrm{def}} b$ for all $t \in \mathbb{N}$. The value of such a gate is always constant in all clock cycles.*

- *If $v \in V$, $\beta(v) = r$ and $t = 0$, then $\mathrm{val}_v(t, \vec{a}_1^n) =_{\mathrm{def}} 0$. This is equivalent to initializing memory elements to 0 at the beginning of the computation.*

Induction Step:

Let $v \in V$ have fan-in $k > 0$ and let $\vec{v}_1^k$ be the predecessor gates and the order of the predecessors be $\alpha((v_1, v)) < \cdots < \alpha((v_k, v))$, then for the all points in time $t \in \mathbb{N}$:

- *If $v \in V$ and $\beta(v) \in \mathcal{B}$, then*

$$\mathrm{val}_v(t, \vec{a}_1^n) =_{\mathrm{def}} \beta(v)(\mathrm{val}_{v_1}(t, \vec{a}_1^n), \ldots, \mathrm{val}_{v_k}(t, \vec{a}_1^n))$$

  This means, that the evaluation is completed in the same clock cycle.

- *If $v \in V$ and $\beta(v) = r$, then*

$$\mathrm{val}_v(t, \vec{a}_1^n) =_{\mathrm{def}} \beta(v)(\mathrm{val}_{v_1}(t-1, \vec{a}_1^n, \ldots, \mathrm{val}_{v_k}(t-1, \vec{a}_1^n))$$

  Hence, the output value of a memory element is the value that was the input to the memory element in the previous clock cycle.

In each clock cycle t the circuit computes a tuple of outputs $(y_1(t), \ldots, y_m(t))$ from previous inputs as follows:

Definition 2.10 *For $1 \leq i \leq m$, let $y_i \in V$ be the output gates, then the function computed by the circuit $C = (\mathcal{B}, r, V, E, \alpha, \beta, \gamma)$ at time $t \in \mathbb{N}$*

$$f_C : N \times \{\vec{a}_1^n\} \to \mathbb{Z}_2^m$$

is for all input value functions $a_i : \mathbb{N} \leftarrow \mathbb{Z}_2$ given by

$$f_C(t, \vec{a}_1^n) =_{\mathrm{def}} (\mathrm{val}_{y_1}(t, \vec{a}_1^n), \ldots, \mathrm{val}_{y_m}(t, \vec{a}_1^n)).$$

Usually some vectorial Boolean function $g : \mathbb{Z}_2^{\geq 0} \to \mathbb{Z}_2^{\geq 0}$ is modeled as a circuit. However, it is not obvious how to map the inputs of g to the inputs of f_C. The same applies for the outputs. This is important to be able to measure the number of clock cycles a circuit needs to finish a computation. Therefore, the function that a BCM computes is defined as follows:

Definition 2.11 *Let C be a BCM with n input gates $\vec{x}_1^n$, m output gates $\vec{y}_1^m$ and $g : \mathbb{Z}_2^{\geq 0} \to \mathbb{Z}_2^{\geq 0}$ the function computed by C, then:*

- *Let $b \in \mathbb{Z}_2^{\geq 0}$ be the input to the function g. Then the i-th input function a_i for the circuit C, with $1 \leq i \leq n$ is defined as $a_i(t) =_{\mathrm{def}} b_{i+t\times n}$, if $|b| \geq i + t \times n$, and otherwise $a_i(t) =_{\mathrm{def}} 0$.*

- *The output of the function g is defined as $(\vec{d}_1^l) =_{\mathrm{def}} (f_C(1, \vec{a}_1^n), \ldots, f_C({}^l/_m, \vec{a}_1^n))$, where $m|l$ and l is the number of output bits that are generated for the input b.*

2.4 Modeling Random Access Memories

One key element of the discussion in the following Part III is the usage of random access memory primitives provided by most FPGAs, because they save a lot of area on these platforms. Yet, the previously defined model deliberately skipped RAMs to simplify the definitions and thus to facilitate the later analysis.

However, it is easy to model a RAM of arbitrary depth and width. For example a small 4×1 bit RAM may be modeled according to Fig. 2.3. In general, a simple dual-port write first RAM as this one can be modeled in three parts:

- A decoder of the write address.
- The memory itself including one two to one multiplexer per memory element.
- A multiplexer to select the output bits according to the read address.

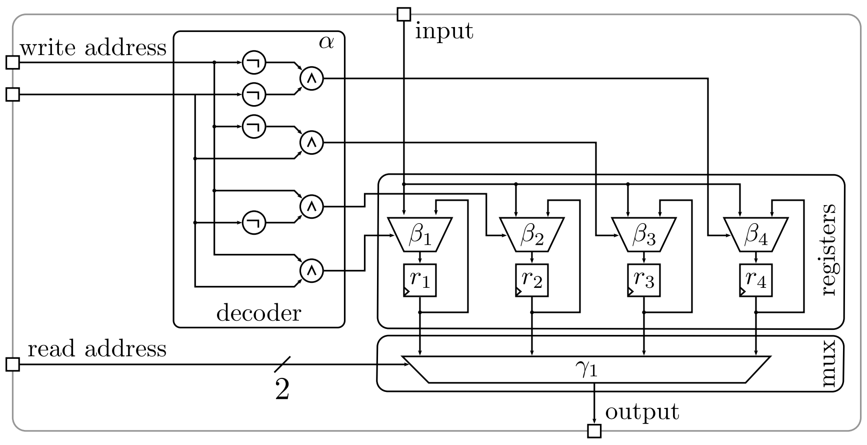


Figure 2.3: Modeling of a 4×1 bit RAM from registers and Boolean circuits.

A more formal definition of the components is as follows, beginning with the notion of a decoder. It is possible to build a decoder of any size which is a power of two, i.e. for $n = 2^k, k \in \mathbb{N}$.

Definition 2.12 *A* n-decoder *for some $n = 2^k, k \in \mathbb{N}$ is defined as an injective vectorial Boolean function $f_n : \mathbb{Z}_2^k \to \mathbb{Z}_2^n$. Each function f_n has the inputs $(\vec{x}_1^k)$ and is evaluated in two steps as follows:*

1. *A bijective function* $\delta : \mathbb{Z}_2^k \to \{1, \ldots, n\}$ *assigns each input vector a natural number. This number then selects in the next step, which bit of the output is set to* 1.
2. *Then* $f_n(\vec{x}_1^k) = (\vec{y}_1^n)$, *where* $y_{\delta(\vec{x}_1^k)} = 1$ *and all other* $y_i = 0$.

Similarly, n-to-one multiplexers are defined as follows:

Definition 2.13 *A* n-to-one multiplexer *for some* $n = 2^k, k \in \mathbb{N}$, *is a Boolean function* $f_n : \mathbb{Z}_2^n \times \mathbb{Z}_2^k \to \mathbb{Z}_2$. *Each function* f_n *has the inputs* $(\vec{x}_1^n, \vec{x}_{n+1}^{n+k})$ *and is evaluated as follows:*

- *A bijective function* $\delta : \mathbb{Z}_2^k \to \{1, \ldots, n\}$ *assigns a natural number to the last* k *inputs* $\vec{x}_{n+1}^{n+k}$.
- *Then* $f_n(\vec{x}_1^{n+k}) = x_{\delta(\vec{x}_{n+1}^{n+k})}$.

With the two Definitions 2.12 and 2.13, the components of a more generic version of the previously mentioned simple dual-port write first RAM (Fig. 2.3) can be defined formally:

Definition 2.14 *Let a* $d \times w$ *bit RAM be a tuple* $\mathcal{R} = (d, w, \alpha, \vec{\beta}_1^{d\times w}, \vec{\gamma}_1^w, \vec{r}_1^{d\times w})$, *where the depth* $d = 2^k, k \in \mathbb{N}$, *the width* $w \in \mathbb{N}$ *and*

- α *is a decoder function* $\alpha : \mathbb{Z}_2^k \to \mathbb{Z}_2^d$, *which decodes the write address.*
- *Each* β_i *is a two-to-one multiplexers* $\beta_i : \mathbb{Z}_2^2 \times \mathbb{Z}_2 \to \mathbb{Z}_2$. *For each of the* $d \times w$ *registers one such multiplexer exists to control in which the new input should be written.*
- *Each* γ_j *is a d-to-one multiplexer* $\gamma_j : \mathbb{Z}_2^d \times \mathbb{Z}_2^k \to \mathbb{Z}_2$ *to control the output of the RAM. There are* w *such multiplexers, one multiplexer for each output bit.*
- r_k *are the functions modeling* $d \times w$ *memory elements.*

The evaluation of the $d \times w$ RAM is defined as follows:

Definition 2.15 *Let* $\mathcal{R} = (d, w, \vec{r}_1^{d\times w}, \alpha, \vec{\beta}_1^{d\times w}, \vec{\gamma}_1^w)$ *be a RAM with the input functions* $(\vec{a}_1^w, \vec{b}_1^k, \vec{c}_1^k$, *where each* $a_i(t)$ *is a RAM input bit at time* t, $(\vec{b}_1^k(t))$ *encodes a write address at time* t *and* $(\vec{c}_1^k(t))$ *encodes a read address at the same point in time. The evaluation of the circuit proceeds as follows:*

- *The decoder function α decodes the write address which uses a binary encoding ($\vec{b}_1^k(t)$) to a one hot encoding. More formally, it evaluates the following function in each clock cycle t:*

$$\mathrm{val}_\alpha(t, \vec{a}_1^w, \vec{b}_1^k, \vec{c}_1^k) = \alpha(\vec{b}_1^k(t))$$

- *Each multiplexer β_i switches between updating the register r_i with the new input or keeping the value from the previous clock cycle, depending on the result of the decoding of the write address. Formally, for all $i \in \{1, \dots, d \times w\}$, the multiplexers β_i evaluate the following function in each clock cycle t:*

$$\begin{aligned}\mathrm{val}_{\beta_i}(t, \vec{a}_1^w, \vec{b}_1^k, \vec{c}_1^k) = \beta_i(a_{\lceil i/d \rceil}(t), &\mathrm{val}_{r_i}(t, \vec{a}_1^w, \vec{b}_1^k, \vec{c}_1^k), \\ &\mathrm{val}_\alpha(t, \vec{a}_1^w, \vec{b}_1^k, \vec{c}_1^k)[\lceil i/w \rceil])\end{aligned}$$

- *The last step is to produce the output of the RAM, i.e. to multiplex between the different register outputs. More formally, for all $j \in \{1, \dots, w\}$, the multiplexers γ_j evaluate in each clock cycle t the following function:*

$$\begin{aligned}\mathrm{val}_{\gamma_j}(t, \vec{a}_1^w, \vec{b}_1^k, \vec{c}_1^k) = \gamma_j(&\mathrm{val}_{r_{(1+((j-1)\times d))}}(t, \vec{a}_1^w, \vec{b}_1^k, \vec{c}_1^k), \dots \\ &\mathrm{val}_{r_{(d+((j-1)\times d))}}(t, \vec{a}_1^w, \vec{b}_1^k, \vec{c}_1^k), \\ &\vec{c}_1^k(t))\end{aligned}$$

- *Also the registers have to be defined, i.e. for all $k \in \{1, \dots, d \times w\}$, the register functions r_k evaluate in each clock cycle $t > 0$:*

$$\mathrm{val}_{r_k}(t, \vec{a}_1^w, \vec{b}_1^k, \vec{c}_1^k) = \mathrm{val}_{\beta_k}(t-1, \vec{a}_1^w, \vec{b}_1^k, \vec{c}_1^k)$$

  and for $t = 0$:

$$\mathrm{val}_{r_k}(t, \vec{a}_1^w, \vec{b}_1^k, \vec{c}_1^k) = 0$$

The behavioral model of this generic RAM and of the RAM depicted in Fig. 2.3 is dual-port write-first. However, there are several other behavioral models available for memory primitives on most FPGAs. These are usually variations or extensions of the already introduced RAM and not based on totally different concepts. Note that DRAM, which is typically used as main memory in modern computer systems has a quite different and in general more complicated (timing) behavior, which is out of scope in this thesis. The following options can usually be used:

- Single-port or dual-port RAMs, with one read/write port and optionally one additional read port.
- Read-first or write-first RAMs.
- A write enable function may be used.
- A read enable function may be used.

The exact area consumption of these behavior models changes, depending on the technology. However, the differences for FPGAs are usually quite small, except for dual- or multi-port RAMs with two or more read ports. Therefore, the model used in this thesis is limited to RAMs with up to one read and one write port exclusively and disregards RAMs with more than one write or read port. An abstraction on the area consumption of RAMs as a complexity measure is discussed in the next section.

2.5 Complexity Measures

In complexity theory, there are two important complexity measures, time and space [AB09, Sip96, Sav97]. For Boolean circuits, the space complexity measure can be translated to the circuit size, i.e. the number of gates, whereas the time measure can be roughly seen as the depth of a Boolean circuit [Vol99]. For the modified model, which includes memory, these complexity measures, circuit size and depth have to be reevaluated.

The case of size is rather easy, because the additional registers can be interpreted as a special kind of gate. Therefore, the definition of the general circuit size can be the same. However, the usual depth measure for Boolean circuits is not usable anymore. First, it reflects the computation time very poorly, because the computation takes multiple clock cycles, possibly using the same paths multiple times and second, the circuit graph may contain cycles and thus, the depth is not well defined anymore.

Therefore, the depth measure has to be replaced with the length of the longest path between two registers,an input and a register, a register and an output or an input and an output. Furthermore, a meaningful measure of computation time can only be achieved, if the number of clock cycles to calculate the output of an algorithm is considered in the complexity measure. Thus, the total computation time of an implementation can be roughly approximated by the longest path multiplied by the number of clock cycles. The number of clock cycles determines the computation time to a large degree and the longest path in a circuit limits the clock frequency in a real world implementation. Hence, the combination of both ideas leads to a theoretically reasonable approximation of the total computation time.

Definition 2.16 *Let* $C = (\mathcal{B}, r, V, E, \alpha, \beta, \gamma)$ *be a BCM. Then* $\mathrm{SIZE}(C) =_{\mathrm{def}} |\{v \in V : \beta(v) \in B \cup r\}|$ *is the number of computational and memory elements of the circuit.*

The longest path $\mathrm{MAXPATH}(C)$ *is the length of the longest directed path in the graph* (V, E) *from a memory element to another memory element, from an input gate to a memory element, from a memory element to an output gate or from an input gate to an output gate.*

The computation time *is defined as* $\mathrm{TIME}(C) =_{\mathrm{def}} \mathrm{MAXPATH}(C) \times {}^{l}/_{m}$. *In other words, it is defined as the multiple of the longest path and the number of clock cycles, until the* l *output bits of the function* g *computed by the circuit* C *are generated according to Def. 2.11.*

In addition to the general complexity measures, it is helpful to define more specialized measures to get a better idea which part of an implementation is dominant in terms of area consumption or performance. For this goal, a separation between the memory elements and the logic is useful. Additionally, a partition between registers and RAMs for FPGA implementations, because an algorithm which requires a lot of memory can be often implemented more efficiently as a RAM-based version than a functionally equivalent solution using registers.

The complexity measure for RAMs grows proportional to the depth and the width of the RAM, i.e. if $d \geq 2$ is the depth and w is the width of the RAM, then the area grows with $O(d \times w)$. The restriction to $d \geq 2$ is not strictly necessary. However, if $d = 1$, then it is a simple group of registers and thus it is equivalent to the number of registers. The rationale for this approximation follows the modeling of the RAM according to Sec. 2.4. There are three main parts, the decoder, which only grows in terms of the depth, the input multiplexers and the registers which both grow proportionally of depth d and width w. The last part, the output multiplexer also grows in terms of d and w. In particular, in a concrete implementation $(d-1) \times w$. Therefore, an abstract metric which grows proportionally to $d \times w$ describes the area consumption of a RAM with a reasonable approximation.

Definition 2.17 *Let* $C = (\mathcal{B}, r, V, E, \alpha, \beta, \gamma)$ *be a BCM. Then the* memory size *is defined as the number of memory elements of the circuit* C*, i.e.* $\mathrm{SIZE}_{\mathrm{mem}}(C) =_{\mathrm{def}} |\{v \in V : \beta(v) = r\}|$

If the circuit C *has a set of RAM instances* $\mathcal{R}$ *with depth* $d \geq 2$*. Then the* RAM size *is defined as* $\mathrm{SIZE}_{\mathrm{RAM}}(C) = \sum_{i \in \mathcal{R}} d_i \times w_i$*, where* d_i *is the depth and* w_i *is the width of the i-th RAM.*

Another valuable complexity measure is the fan-out of a BCM or a Boolean circuit. An interesting property is, that the fan-out of a Boolean circuit without memory has a close connection to the memory requirements of a BCM.

Definition 2.18 *Let* C *be a BCM or a Boolean circuit. Then the* fan-out *of the circuit* C *is defined as* $\mathrm{FANOUT}(C) = |\{v \in V : \gamma(v) \neq *\}|$*, i.e. the number of output gates of the circuit.*

A BCM has always a simple corresponding Boolean circuit, that only computes one step of the computation. An example of this concept is shown in Fig. 2.4. It is the corresponding Boolean circuit of the RAM depicted in Fig. 2.3. This sub circuit may be constructed in general according to the following definition.

Definition 2.19 *Let* $C = (\mathcal{B}, r, V, E, \alpha, \beta, \gamma)$ *be a BCM. Then a corresponding Boolean circuit* $C' = (\mathcal{B}, V, E, \alpha, \beta, \gamma)$ *may be constructed by replacing each register, i.e.* $\beta(v) = r$ *with a pair of input and output gates in* C'*.*

The following lemma can be derived from the definition, which introduces a connection between the fanout of C' and the memory requirements of C.

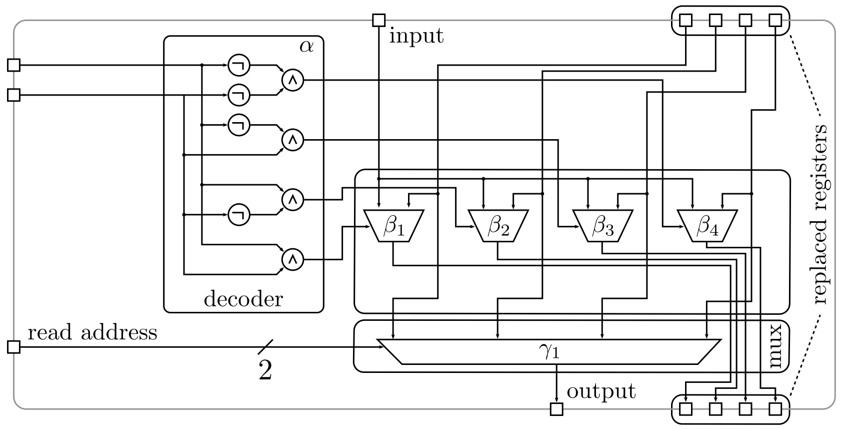


Figure 2.4: Boolean sub-circuit of the BCM depicted in Fig. 2.3.

Lemma 2.20 *Let C be a BCM and C' the corresponding Boolean circuit, then* $\text{SIZE}_{\text{mem}}(C) \leq \text{FANOUT}(C') \leq \text{SIZE}_{\text{mem}}(C) + \text{FANOUT}(C)$*.*

Proof Every memory element of C is replaced by an input and an output gate in C' according to Def. 2.19 and hence, $\text{SIZE}_{\text{mem}}(C) \leq \text{FANOUT}(C')$. Additionally, $\text{FANOUT}(C')$ is bounded by the sum of the number of memory elements and the number of output gates in C, because only memory elements are replaced, and hence $\text{FANOUT}(C') \leq \text{SIZE}_{\text{mem}}(C) + \text{FANOUT}(C)$. □

For some special cases of circuits, Lemma 2.20 may be refined. The first special case consists of circuits, where all output gates are implemented as memory elements. This implementation strategy using so-called registered outputs has a real world application, because in many applications, it helps to solve timing problems with external components [Kil07]. In particular, most FPGAs have special outputs for this purpose. For example, Xilinx devices usually have so-called Input/Output Blocks (IOBs), which can be configured to use a register in exactly this way [Xil12c].

Corollary 2.21 *Let C be a BCM and C' the corresponding Boolean circuit. Then* $\text{FANOUT}(C') = \text{SIZE}_{\text{mem}}(C)$*, iff for all $v \in V_C \beta(v) = r$, if $\gamma(v) \neq *$.*

Proof "⇒": The number of output gates of C' is equal to the number of memory elements of C. Therefore by Def. 2.19, all output gates of C have to be memory elements.

"⇐": Similarly, if all output gates are memory elements, then by Def. 2.19

the number of output gates of the circuit C' is equal to the number of memory elements of C. □

It is similarly possible to derive another corollary in the other direction, if none of the outputs are registers:

Corollary 2.22 *Let C be BCM and C' the corresponding Boolean circuit. Then* $\mathrm{FANOUT}(C') = \mathrm{SIZE}_{\mathrm{mem}}(C) + \mathrm{FANOUT}(C)$*, iff for all* $v \in V_C \beta(v) \neq r$*, if* $\gamma(v) \neq *$.

Proof "⇒": The number of output gates of C' is equal to the sum of the number of memory gates of C and the number of output gates of C. Therefore by Def. 2.19, no output gate of C is a memory element.

"⇐": Similarly, if none of the output gates are memory elements, then by Def. 2.19 the number of output gates of the circuit C' is equal to the sum of the number of memory elements of C and the number of output gates of C. □

In addition to the two previous results, there is a connection between the maximum path length $MAXPATH(C)$ of a BCM C and the depth of a corresponding Boolean circuit C'. For this connection, the DEPTH measure for Boolean circuits has to be formally introduced:

Definition 2.23 *Let C be a Boolean circuit, then the depth of this circuit* $\mathrm{DEPTH}(C)$ *is the length of the longest directed path in the circuit graph* (V, E).

With this definition it is easy to see, that the following lemma is true:

Lemma 2.24 *Let C be a BCM and C' the corresponding Boolean circuit. Then* $\mathrm{MAXPATH}(C) = \mathrm{DEPTH}(C')$.

Proof The MAXPATH complexity measure can also be used for Boolean circuits as a special case of BCMs. Thus, because of the definitions of MAXPATH and DEPTH, $\mathrm{MAXPATH}(C') = \mathrm{DEPTH}(C')$. Furthermore, the procedure to generate the corresponding Boolean circuit C' preserves the $\mathrm{MAXPATH}(C)$ measure for C' and thus $\mathrm{MAXPATH}(C) = \mathrm{MAXPATH}(C') = \mathrm{DEPTH}(C')$. □

Chapter 3

Hash Functions

3.1 Introduction

In this chapter, the notion of cryptographic hash functions will be introduced and discussed. A hash function calculates a digest of fixed length from a message of arbitrary length. This idea is readily exploited in many sub-disciplines of computer science. For example, hash tables [Knu73] or cyclic redundancy checks [PB61] are widely used and very important concepts built upon hash functions.

However, most of these concepts only work successfully in their particular domain. Even then there may be problems. For example, an adversary may have the ability to attack a system, because the algorithms process user-supplied data, e.g. in a web application. A denial of service attack on hash tables is a prominent case, because the performance may degrade significantly (from $O(1)$ to $O(\log\, n)$), if the adversary may easily manipulate the input in such a way, that the hash function output collides, i.e. the same digest is generated for two or more input messages [CW03]. For example, this is a serious problem for large-scale web applications. Cryptographic applications, e.g. message authentication codes (MAC) or digital signatures are even more sensitive to this property.

Some groundbreaking work on the security of cryptographic hash functions was published by Merkle [Mer79, Mer89] and Damgård [Dam89]. They both independently proved the collision resistance property of the so-called Merkle-Damgård construction, under the condition, that the underlying compression function is also collision resistant. In the following years more important results in the field of hash and one-way functions were published. Especially noteworthy, in the context of the present thesis, are other hash function constructions, for example the Davies-Meyer construction [DP84, Win84b, Win84a] and the recently invented sponge functions [BDPA07, BDPA08]. These constructions have a major influence on the area consumption of lightweight implementations.

Many other important advances on the theory of hash functions were published, e.g. the PhD thesis of Preneel [Pre93]. Preneel systematically analyzed the security of hash functions using two approaches based on information and complexity theory, respectively, hence improving the state of the art at that time. This theoretical approach was further improved, e.g. by Rogaway et al. [RS04]. The present introduction to cryptographic hash functions will, however, only provide a rough overview over the basic concepts and does not claim to cover the details on the latest research on the theory of hash functions. Thus, mostly topics relevant for this thesis are described and other references are only provided as necessary.

The core of this thesis is a thorough performance evaluation of several algorithms with the focus on lightweight and midrange applications (Part III). Most of the evaluated algorithms were published during the SHA-3 competition, which was initiated in 2007 by the National Institute of Standards and Technology (NIST) [Kay07]. The competition followed attacks on the well-known hash functions MD5, RIPEMD [WFLY04] and SHA-1 [WYY05] and also slight improvements of the cryptanalysis of the SHA-2 family [IS09, SS08]. The competition inspired a lot of further research into hash functions, in particular the five finalist algorithms of the competition [AHMP10, BDPA11b, Wu11, FLS+10, GKM+10]. An overview over most important research and results regarding the SHA-3 finalists that were published during the competition is given in NIST's third round SHA-3 report [CPB+12]. The algorithms themselves are described in Ch. 6.

The remainder of this chapter is organized as follows. First, the notion of hash functions will be defined together with the general security properties that a cryptographic hash function should fulfill to ensure the security of many protocols (Sec. 3.2, Sec. 3.3). The definition about the security properties is accompanied by a brief discussion about generic attacks on hash functions (Sec. 3.3.2) and implementation attacks in the form of side channel analysis (Sec. 3.3.3). Then the introduction is concluded by a description of commonly employed iteration modes (Sec. 3.5), a short overview of tree modes (Sec. 3.6) and a list of important applications of cryptographic hash functions (Sec. 3.7).

3.2 Hash Functions

Definition 3.1 *A* hash function h_n *is a function* $h_n : \mathbb{Z}_2^{\geq 0} \rightarrow \mathbb{Z}_2^n$*.*

A hash function can be seen as a function that generates fixed-length fingerprints $y \in \mathbb{Z}_2^n$ of messages $x \in \mathbb{Z}_2^{\geq 0}$. This fingerprint is usually called a hash value or message digest of the message x. All of these message digests have the same fixed-length n, and hence, there are security related implications, because infinitely many messages are mapped to the same fixed-length message digest. This phenomenon, called a hash collision, may be defined as:

Definition 3.2 *The hashes of two messages* $x, x' \in \mathbb{Z}_2^{\geq 0}$ *and* $x \neq x'$ *collide, if* $h_n(x) = h_n(x')$*, i.e. the message digests for both messages are equal. This is said to be a* hash collision*.*

In the following chapters and sections, the subscript of h_n will be skipped, if the value of n is not important or clear from the context.

3.3 Security of Hash Functions

Obviously, there are always collisions because of the mapping of an infinite set of messages to a finite set of digests. Thus, a hash function has to be designed carefully in such a way, that it ensures practical security [Mer79]. The theoretical properties, which a cryptographic hash function has to fulfill are defined in this section [Mer79, Mer89, Pre93]. Furthermore, generic attacks will be outlined to understand the theoretical upper bounds.

Besides the theoretical generic attacks on hash functions, there are several implementation attacks. These attacks do not break any theoretical security claims of a hash algorithm. Instead, they attack weaknesses of the implementations to extract secret data. An attack of this kind is obviously only threatening for hash functions, if the algorithm processes sensitive data, such as passwords or secret keys, e.g. the HMAC algorithm [BCK96]. One of the most prominent class of attacks of this kind are so-called side-channel attacks, which are briefly discussed later in form of power analysis [KJJ99].

3.3.1 Security Properties

A hash function for cryptographic applications has to fulfill some important properties. These are collision, preimage and second preimage resistance. However, depending on the application where a hash function is used, only one or several of these properties must hold. This can be justified with the security proofs of a particular application or protocol, which is most often a proof by reduction and uses one or several properties of the underlying hash function. This principle has been often reiterated and a classification of hash functions can be developed, e.g. collision resistant hash functions or (weak) one-way hash functions [Pre93]. It is also possible to define several more precise versions of the following properties, which have subtle, but important differences [RS04]. However, for the current thesis, the simple general definitions are sufficient.

Collision Resistance All hash functions have collisions, which is a general problem that has to be solved practically to make hash functions useful in a cryptographic application. Therefore, the most basic security property is collision resistance against random collisions of two messages [Mer89, Rog06, Dam88]. The collision resistance of a hash function may be defined as follows, similar to the definition by Preneel [Pre99]; however, the following definition is less formal.

Definition 3.3 *A hash function h is said to be* collision-resistant*, if the probability that a (computationally uniform) polynomial time adversary can find two messages $x, x' \in \mathbb{Z}_2^{\geq 0}$ and $x \neq x'$, such that $h(x) = h(x')$ is negligible.*

It is important to note, that the adversary is set in a uniform computational model. Otherwise, there would be always a trivial algorithm to find a collision. This constant time algorithm would just print two messages x and x' with $h(x) = h(x')$. However, it should be infeasible to implement this algorithm in polynomial time. This fundamental difference between the uniform and non-uniform setting was pointed out by Rogaway [Rog06]. For the non-uniform setting, an infinite family of hash functions has to be used in the previous definition to get a meaningful definition of collision resistance.

However, while the definition states, that it should be next to impossible to find a collision, it is still theoretically possible. Therefore, the question remains, how long an adversary would need to find a collision in the generic case. The upper bound may be derived from the birthday paradox. It leads to a generic attack with $O(2^{n/2})$ operations to have a probability which is higher than $1/2$ to find a collision [Yuv79].

Preimage Resistance The second important security property is the preimage resistance, i.e. it has to be difficult to find a message for a known hash digest. This property is also called one-wayness [Mer79]. The definition is again similar to, but less formal then the definition of preimage resistance by Preneel [Pre99].

Definition 3.4 *A hash function h is said to be* preimage resistant*, if the probability that a (computationally uniform) polynomial time adversary A can find a message $x' = A(h(x))$ with $x \neq x'$, such that $h(x') = h(x)$ is negligible.*

Again, the computational upper bound is particular interesting in a practical setting. It turns out, that $O(2^n)$ is the best known generic upper bound, if no quantum computer is assumed [Gro96, BHT98].

2nd Preimage Resistance The third important security property is the second preimage resistance, i.e. it has to be difficult to find a second message with the same hash digest than the original message [Mer79]. Similar to the definition of preimage resistance, the definition is a less formal variant of the definition by Preneel [Pre99].

Definition 3.5 *A hash function h is said to be* second preimage resistant*, if the probability that a (computationally uniform) polynomial time adversary A can find a message $x' = A(x)$, such that $h(x') = h(x)$ and $x \neq x'$ is negligible.*

From a practical perspective a brute force attack does not require significant rethinking, therefore the same $O(2^n)$ bound does apply for the second preimage resistance.

3.3.2 Generic Attacks

The security bounds for the definitions in the previous subsection may be derived from the best known generic attacks on hash functions without using quantum computation models. These attacks have been discovered quite early at the beginning of the systematic research of hash functions [Yuv79, Mer89, Mer79]. However, research on quantum computer improved some of the upper bounds for the quantum computer setting [Gro96, BHT98].

Random Collisions For the case of random collisions, it is obvious that the probability to find a collision is 1, if the attacker tests more than 2^n messages on collisions. However, the probability to find a message is already very high with a lot less messages, to be more precise $O(2^{n/2})$. This phenomenon has been described quite early [Yuv79, Mer89].

The attack is based on a simple combinatorial argument, typically called the birthday paradox. Checking if the message digests for two distinct messages collide, involves the calculation of $h(x)$ and $h(x')$ and an equality test $h(x) = h(x')$. However, for k messages there are $\frac{k\times(k-1)}{2}$ pairs of possible collisions and not only $k/2$. Although these pairs are not all statistically independent, it is plausible, that the probability to find a random collision is much higher.

The exact probability can be calculated as follows, where **coll** is a function which tests k random messages out of 2^n possible messages and outputs 1, if one of the k messages collides with one of the other $k-1$ messages [TW02]:

$$\Pr[\mathbf{coll}(k, 2^n) = 1] = 1 - \frac{2^n}{2^n} \times \frac{2^n - 1}{2^n} \times \cdots \times \frac{2^n - k + 1}{2^n} = 1 - \frac{k!\binom{2^n}{k}}{2^{nk}}$$

Finding (2nd) Preimages The best known generic (second) preimage attack on ideal cryptographic hash functions is a brute-force attack taking $O(2^n)$ operations. This bound follows from the following argument. If the outputs of the hash function are uniformly distributed, then the probability to find a preimage for $h(x)$ is $\Pr[h(A(h(x))) = h(x)] = 1 - [\frac{2^n-1}{2^n}]^k$, where k is the number of messages that are tested by the adversary. Hence, the probability, that the adversary finds a message is significantly higher then $1/2$ for $k = 2^n$. And if the adversary tests only half of that number, i.e. $k' = 2^{n-1}$ messages, the probability of a successful attack is significantly below $1/2$.

Quantum computing may speed up a preimage attack considerably to $O(2^{n/2})$ using Grover's algorithm [Gro96, BHT98]. However, even for classical computers, there are sometimes possibilities to speed up the exhaustive search, e.g. by using rainbow tables, where the attacker trades memory for a speed up in attack time [Oec03].

3.3.3 Side-Channel Attacks

Every computation leaks a certain amount of information about the data it currently processes. For example, the runtime may be different, the power consumption varies, or the electromagnetic dissipation changes dependent on the input. This type of attack is only relevant, if secrets are processed, e.g. passwords. Otherwise, no private information and thus nothing of value to an adversary is leaked, if the algorithm and the data to be hashed are both publicly known. For succinctness, only simple power analysis (SPA) and differential power analysis (DPA) will be discussed, because after their introduction, they became quickly two of the most common attack methods in the field of side-channel analysis [KJJ99]. A standard reference for power analysis has been written by Mangard et al. [MOP07].

Simple Power Analysis For a successful power analysis, the attack has to measure the power consumption while the computation is performed. Afterwards, the attacker interprets the power trace directly. For naïve implementations only one or very few power traces are necessary to extract the secret. It is even possible that the attacker may be able to visually interpret the results herself [MOP07]. More sophisticated attacks build templates of known operations with known keys in advance. Then in the attack phase, the adversary may match parts of the power trace to the operations and in turn directly to (parts of) the key. The main drawback of template attacks is, that the attacker has to have access to an equivalent device before the attack and that the template building, depending on the granularity of the templates, needs a lot of memory and time.

Differential Power Analysis Differential power analysis is more powerful than SPA. However, the attacker usually has to measure many more traces.

According to Mangard et al. [MOP07], the basic attack consists of five steps:

1. Choose a suitable intermediate result of the algorithm. This result should have two properties. It should be computed by a function $f(d, k)$, where f and d are known to the attacker and k is a part of the key or derived from the key.

2. Measure and record the power consumption of the algorithm several times. Most often each measurement is performed with a different set of known inputs.

3. Calculate hypothetical intermediate values according to some idealized model, e.g. the Hamming weight or Hamming distance model [BCO04].

4. Map the measured intermediate values to the modeled hypothetical power consumption values.

5. Compare the hypothetical power consumption values to the power traces. This last step should reveal the correct key.

Many variants of this basic DPA scheme were developed. One of the most common variant uses the correlation coefficient to determine the correlation between the hypothetical and the measured power traces in step five [BCO04].

3.3.4 Side-Channel Countermeasures

Countermeasures against side-channel analysis usually fall in one of two categories, hiding and masking [MOP07]. The goal of both is to reduce or break the statistical connection between the secret values and the power consumption (or the electromagnetic emission, computation time, and so on). For the leakage reduction, the cause of the leakage has to be found first. An intuitively accessible example is, that the registers in most integrated circuits have a different power consumption, if they switch from a 0 to 1 or the other way, in contrast to maintaining the same value between two clock cycles. These kinds of leakage have to be removed or at least reduced, if an attack is to be inhibited.

In real-world implementations, several countermeasures are usually combined to achieve a high resistance against different types of side-channel attacks. However, the state of the art never completely removes the leakage and new attacks are constantly developed. Thus, a permanent re-evaluation of attacks and countermeasures is necessary.

Hiding Hiding tries to remove the dependency by breaking the connection between processed values and the leakage. The algorithm itself is kept the same, i.e. the intermediate values are identical to the unprotected version. However, the implementer tries to reduce the leakage.

A lot of different hiding countermeasures were developed in the last years, for example dual-rail precharge logic which tries to remove the leakage altogether by changing the physical implementation [TV04], introducing dummy operations, which makes it more difficult to correlate the power traces to the hypothetical values [MOP07], or mutating data paths which reduce the link by shuffling the operations of the algorithm in the time domain [Stö13].

Masking Masking uses a different approach to break the link between the secret input values and the leakage by internally changing the processed data in a (pseudo-)random fashion. Now, because a different randomized set of data is used, the attacker should be unable to correlate the leakage and the intermediate values.

Many possible masking schemes were proposed. For example Boolean masking adds a random mask value to the intermediate values, which later has to be removed to recover the correct value [CB08]. Another example are threshold implementations, which use two random masks, similar to the Boolean masking scheme [NRR06]. The countermeasure is based on Shamir's secret sharing scheme [Sha79] and it can be proven that much less random data is necessary compared to the Boolean masking scheme.

3.4 Domain Extender

According to Definition 3.1, the input of a hash function can be of arbitrary size. It is easy to see, how to construct such an algorithm in principle. However, it is very difficult to proof security properties of an algorithm, such as collision or preimage resistance in general. For example, it is still unknown if one-way functions exist at all [TW06, AB09, Ko85] and therefore, it is also unknown if hash functions exist.

Thankfully, security proofs based on some (reasonable) assumptions are still possible. One of the most useful technique is to base a hash function on a small component, often called the compression function. The compression function is usually a function $f : \mathbb{Z}_2^n \times \mathbb{Z}_2^m \rightarrow \mathbb{Z}_2^n$, which takes a n bit initialization value (or chaining value), a m bit message block, and produces a compressed n bit output. Computations of this compression function may be repeated or chained, therefore enabling the hash function designer to build a hash function which processes arbitrary sized inputs. These constructions are often called *domain extenders*, because they extend the fixed domain of a compression function to a much larger, in general infinite domain [CDMP05].

The designer of a hash function may leverage this concept to proof that the domain extender has a specific property (e.g. collision resistance), if the compression function fulfills some assumptions. Usually these assumptions are based on an idealized function such as a random oracle, a random block cipher or a random permutation. In the next section, some important domain extenders will be discussed together with a sketch on the security claims and proofs.

3.5 Iterated Hash Functions

One of the earliest construction of cryptographic hash functions is the Merkle-Damgård construction [Mer79, Mer89, Dam89]. It pioneered the concept of so called iterated hash functions, which virtually all currently deployed hash functions are based upon (e.g. SHA-1 or SHA-2). Iterated hash functions process the message in several iterations. In each iteration a fixed number of message bits are processed. The hash digest is the output of the last call to the compression function, which is computed for the processing of the last message bits. Optionally, this output is post-processed before generating the final hash digest.

3.5.1 Merkle-Damgård

The Merkle-Damgård design was first proposed by Merkle [Mer79]. It has been proved independently by Merkle [Mer89] and Damgård [Dam89], that if the compression function is collision resistant, then the hash function is also collision resistant.

Design The design of a Merkle-Damgård hash function is quite simple. Informally it can be illustrated as depicted in Fig. 3.1. More formally, the hash function construction may be defined as follows [Mer79]:

Definition 3.6 *Let $n, m \in \mathbb{N}$ be constant and $k \in \mathbb{N}$ variable, then a Merkle-Damgård hash function h_n consists of a compression function $f : \mathbb{Z}_2^n \times \mathbb{Z}_2^m \to \mathbb{Z}_2^n$, a bijective padding function $\mathrm{pad}_m : \mathbb{Z}_2^{\geq 0} \to \bigcup_{k \geq 1} \mathbb{Z}_2^{k \times m}$ and the initial value $\mathrm{iv} \in \mathbb{Z}_2^n$. The hash function h_n is evaluated according to Algorithm 3.1.*

Algorithm 3.1 Merkle-Damgård construction [Mer79]

```
Require: $f : \mathbb{Z}_2^n \times \mathbb{Z}_2^m \to \mathbb{Z}_2^n$, $\mathrm{pad}_m : \mathbb{Z}_2^{\geq 0} \to \bigcup_{k \geq 1} \mathbb{Z}_2^{k \times m}$, $\mathrm{iv} \in \mathbb{Z}_2^n$ and $x \in \mathbb{Z}_2^{\geq 0}$
Ensure: $y \leftarrow h_n(x)$
  $p \leftarrow \mathrm{pad}_m(x)$
  $y \leftarrow \mathrm{iv}$
  for $i = 1$ to $|p|/m$ do
    $y \leftarrow f(y, p_i)$
  end for
  return $y$
```

The padding function has several important purposes. The first is to ensure, that the length of the processed message is a multiple of the message block length m. Another more important purpose is to pad the message in a way, that a simple length extension attack is impossible. Such an attack generates – in its simplest instance – two messages x_1, x_2, where x_1 is a prefix of x_2, $h(x_1) = h(x_2)$ and $|x_1| \neq |x_2|$. Many hash functions using a Merkle-Damgård construction append at least a number of zeros and the length of the original message, such that the total length is a multiple of the message block length.

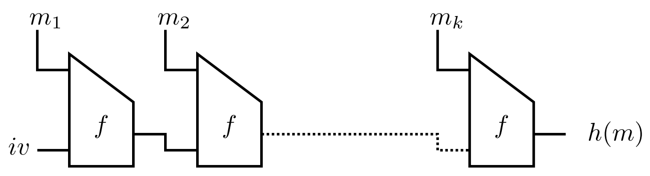


Figure 3.1: Merkle-Damgård domain extender.

Security Claims As already mentioned, the main security claim is, that if the compression function f is collision resistant, then the hash function h is also collision resistant. The proofs from Merkle [Mer89] and Damgård [Dam89] may be easily adapted to the notation used in this thesis. First, it is necessary to see, that the hash computation may be defined inductively:

Definition 3.7 *Let $p \in \bigcup_{k \geq 1} \mathbb{Z}_2^{k \times m}$ be an already padded message and let p_1, $\ldots, p_k$ be k parts of the message with at least m bits each. Then an alternative definition for the computation of the Merkle-Damgård construction is:*
Induction Basis:
The first message block is processed using $y_1 = f(iv, p_1)$.
Induction Step:
Each further message block p_{n+1}, with $1 < n < k$, is processed by computing $y_{n+1} = f(y_n, p_{n+1})$.

The hash digest for a message x is defined to be $h(x) = y_k$.

Then the security claim may be expressed in the following theorem. The proof sketch follows the proof developed by Damgård; the slightly different proof proposed by Merkle is by induction but uses in general the same argument.

Theorem 3.8 *Let f be a compression function and let h be a hash function built using the Merkle-Damgård construction. Then h is a collision resistant hash function, if f is collision resistant.*

Proof sketch For two different already padded messages x and x', there are two cases, $|x| \neq |x'|$ and $|x| = |x'|$.

For the first case, it is easy to see that the last message blocks of x and x' differ, because of the padding function which appends the message length to the message and thus, there must also be a collision in f for the last message blocks, if $h(x) = h(x')$.

For the second case, consider two message blocks $x_k \neq x'_k$. Since both messages are of equal length, the last message block does not necessarily collide in f. However, there must be at least one earlier collision $f(y_{k-1}, x_k) = f(y'_{k-1}, x'_k)$, which is then obviously a collision in f. □

Modifications Despite the main security proof of collision resistance, the basic design has several general weaknesses, which are demonstrated by the following short and by no means comprehensive list:

- It is impossible to construct a simple secure message authentication code (MAC) by prefixing the key k to the message x, i.e. $MAC(k, x) = h(k||x)$ is vulnerable to easy length extension attacks. In particular, $h(MAC(k, x), y) = MAC(pad(x)||y)$ [CDMP05].
- *2nd collisions* are very easy to find, i.e. if a collision $h(x) = h(x')$ is found, then all $h(x||s) = h(x'||s)$ are also collisions for all $s \in \mathbb{Z}_2^{\geq 0}$ [Luc05].
- It is easier to find second preimages than the theoretical $O(2^n)$ bound for (very) long messages. [KS05].

Therefore, several enhanced designs were proposed in the literature, which fix a lot of these problem. Among the most important improvements is the *wide-pipe* Merkle-Damgård construction, which uses a larger internal state. Lucks proposed to choose an internal state with $w > n$ bits in general and $w = 2n$ in particular, while the output of the hash function is still only n bit wide [Luc05]. In the latter case, the compression function can be replicated twice. A similar idea was proposed by Coron et al. with *chop-MD*, where a number of bits are removed from the last compression function output [CDMP05].

3.5.2 Davies-Meyer

The Davies-Meyer construction was first proposed by Davies et al. [DP84] and was further investigated by Winternitz [Win84b, Win84a]. It is a slight improvement to the earlier design proposed by Merkle, but is in several aspects very similar.

Design The design of the Davies-Meyer hash function construction is very similar to the Merkle-Damgård design (Fig. 3.2). Thus, it can be defined as follows:

Definition 3.9 *Let $n, m \in \mathbb{N}$ be constant and $k \in \mathbb{N}$ variable. Then a Davies-Meyer hash function h_n consists of a compression function $f : \mathbb{Z}_2^n \times \mathbb{Z}_2^m \to \mathbb{Z}_2^n$, a bijective padding function $\mathrm{pad}_m : \mathbb{Z}_2^{\geq 0} \to \bigcup_{k \geq 1} \mathbb{Z}_2^{k \times m}$ and the initial value $\mathrm{iv} \in \mathbb{Z}_2^n$. The hash function h_n is evaluated according to Alg. 3.2.*

Algorithm 3.2 Davies-Meyer construction [DP84]

```
Require: $f : \mathbb{Z}_2^n \times \mathbb{Z}_2^m \to \mathbb{Z}_2^n$, $\text{pad}_m : \mathbb{Z}_2^{\geq 0} \to \bigcup_{k \geq 1} \mathbb{Z}_2^{k \times m}$, iv $\in \mathbb{Z}_2^n$ and $x \in \mathbb{Z}_2^{\geq 0}$
Ensure: $y \leftarrow h_n(x)$ with $n \in \mathbb{N}$ and $y \in \mathbb{Z}_2^n$
  $p \leftarrow \text{pad}_m(x)$
  $y \leftarrow \text{iv}$
  for $i = 1$ to $|p|/m$ do
    $y \leftarrow f(y, p_i) \oplus y$
  end for
  return $y$
```

The only difference between the Merkle-Damgård design and the Davies-Meyer construction is the XOR operation after the call to the compression function which combines the previous state with the new output of the compression function. This additional operation makes it possible to prove the preimage resistance. However, especially for hardware implementations, it adds an additional cost factor. because the XOR operation forces the developer to allocate more memory resources to store the previous state while computing the compression function.

Security Claims For the Davies-Meyer construction, the previous result of the Merkle-Damgård hash design also holds. This can be easily seen, by integrating the XOR operation in f. Then the structure is the same as the Merkle-Damgård design. However, there is an additional provable security property; the improved construction is provably preimage resistant, if the compression function is ideal [Win84a]. An ideal compression function is usually considered to be a random oracle or a random block cipher[CDMP05]. However, these primitives do not exist in practice [MRH04, CGH04]. This is a severe limitation, but it is usually sufficient, if there are no known cryptanalytic results on the underlying primitive in a concrete hash function, which are better than

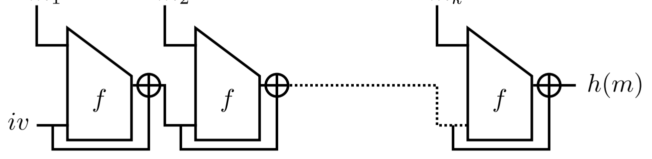


Figure 3.2: Davies-Meyer domain extender.

the generic attacks.

Lemma 3.10 (Lemma 1 [Win84a]) *Let $d \in \mathbb{Z}_2^n$ be an internal state of the hash function and f be an ideal compression function. Then the runtime to find a pair (c, x_i), such that $f(c, x_i) \oplus c = d$ is $O(2^n)$.*

Proof sketch If the compression function f is ideal, the output distribution of f is uniform. Then the success probability of finding such a pair by computing j-times the compression function f is bounded by $j/2^n$. This success probability translates into the expected runtime of $O(2^n)$. $\square$

This lemma can be used to prove the following theorem:

Theorem 3.11 (Theorem 2 [Win84a]) *Given a hash digest $y \in \mathbb{Z}_2^n$, finding a message $x \in \mathbb{Z}_2^{\geq 0}$ with $h(x) = y$ requires $O(2^n)$ computation steps.*

Modifications Again, as it is the case for the Merkle-Damgård design, there are some weaknesses for this design strategy, e.g. the attack from Kelsey et al. holds also for the Davies-Meyer design [KS05]. Therefore, other variants were proposed, e.g. the Matyas-Meyer-Oseas design, which is very similar to the Davies-Meyer mode [MMO85] or the Miyaguchi-Preneel construction, which was proposed independently by Preneel [Pre93] and Miyaguchi et al. [MOI90].

3.5.3 Sponge Functions

Sponge functions were introduced by Bertoni et al. [BDPA07, BDPA11c]. They are modeled to resemble a random oracle [BDPA11c]. Random oracles are (idealized) functions, which produce a random bitstring of infinite length from an input bitstring of arbitrary but finite length. This concept does not exist in the real world and therefore, random sponge functions differ from random oracles in the effects of the finite internal state required to implement a sponge function [BDPA11c].

Design Similar to the previously described Merkle-Damgård and Davies-Meyer designs (Fig. 3.3), the sponge construction may be formally defined as follows. Note, that the random permutation can also be replaced by a random transformation.

Definition 3.12 *Let $r, c, b, n \in \mathbb{N}$ be the rate r, the capacity c, the state size b, and the message digest size n. Then a sponge function consists of a random permutation $f : \mathbb{Z}_2^b \to \mathbb{Z}_2^b$, and a sponge-compliant padding function* $\mathrm{pad}_r : \mathbb{Z}_2^{\geq 0} \to \bigcup_{k \geq 1} \mathbb{Z}_2^{k \times r}$. *A sponge function is evaluated according to Algorithm 3.3.*

Algorithm 3.3 Sponge construction [BDPA11b]

```
Require: $r < b$, $f : \mathbb{Z}_2^b \to \mathbb{Z}_2^b$, $\mathrm{pad}_r : \mathbb{Z}_2^{\geq 0} \to \bigcup_{k \geq 1} \mathbb{Z}_2^{k \times r}$
Ensure: $z \leftarrow \mathrm{sponge}(x, n)$ with $x \in \mathbb{Z}_2^{\geq 0}$, $n \in \mathbb{N}$ and $z \in \mathbb{Z}_2^n$
  $p \leftarrow \mathrm{pad}_r(m)$
  $s \leftarrow 0^b$
  for $i = 1$ to $|p|/m$ do
    $s \leftarrow s \oplus (p_i || 0^c)$
    $s \leftarrow f(s)$
  end for
  $z \leftarrow \lfloor s \rfloor_r$
  for $i = 1$ to $\lceil n/r \rceil$ do
    $s \leftarrow f(s)$
    $z \leftarrow z || \lfloor s \rfloor_r$
  end for
  return $\lfloor z \rfloor_n$
```

There are multiple possible sponge-compliant padding schemes. In [BDPA11c], the multi-rate padding is proposed. This padding scheme appends a number of bit to a message m according to the following rule. If $l =_{\text{def}} |m|$ is the length of the message, then a string 10^k1 is appended, where $k = (r - l - 2) \bmod r)$.

The sponge construction is considerably different to both the Merkle-Damgård and the Davies-Meyer designs. From a practical point of view, the most important features of the sponge construction are the highly configurable

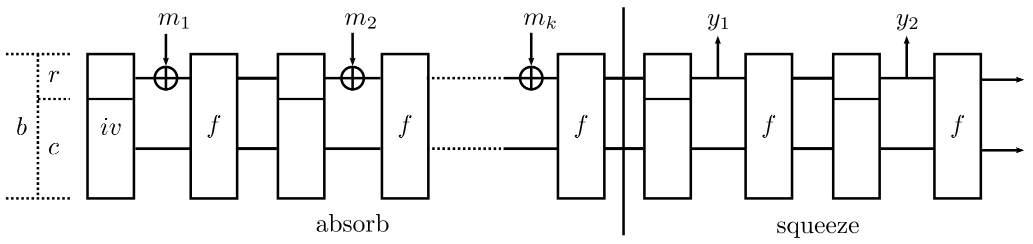


Figure 3.3: Design of sponge functions.

performance and security parameters r, c, b, and n. As usual n is the size of the message digest. The other parameters describe the performance and the security with $b = r + c$. The rate r is comparable with the message block size m in the two other constructions and thus determines a large part of the performance. However, balancing performance and security is a classical trade-off between both parameters, because increasing r decreases the security parameter c. On the other hand b may be increased and in turn increase both the security and the throughput. This would influence another performance parameter, the memory requirements, which are increased proportionally to b.

Security Claims Bertoni et al. proved several properties of the sponge construction. One of the central theorems states, that so-called inner collisions are the only source of non-uniformity [BDPA11c]. In particular, the output of a sponge function is uniformly distributed, if a sequence of queries to the sponge function does not have inner collisions.Inner collisions are collisions of the inner state. This inner state consists of c bits, which are only changed by the permutation and not by the message input of size r. This can be formalized in the following theorem:

Theorem 3.13 (Theorem 5 [BDPA11c]) *Let f be a random permutation and* pad *a sponge-compliant padding function. Then the output bits of the sponge function are distributed uniformly and independent for a sequence of queries, if no inner collisions happen during the queries.*

Proof sketch Let $\mathcal{S} \subseteq \mathbb{Z}_2^b$ be the set of all states and $\hat{\mathcal{S}} \subseteq \mathbb{Z}_2^c$ the set of all inner states which are traversed by all queries of the query sequence. If f is a random permutation, then by construction of f and the sponge function, all states in $\mathcal{S}$ are traversed exactly once (except the initial state $0^r||0^c$), if all inner states of $\hat{\mathcal{S}}$ are traversed exactly once. Then for all $s \in \mathcal{S}$ and inputs $x \in \mathbb{Z}_2^r$, the first r output bits of $f(s \oplus (x||0^c))$ are independent of all previously traversed states, because each possible state is traversed at most once. Hence, all possible values (and all individual bits) are equiprobable. $\square$

Additional results were proved, e.g. based on the indifferentiability framework of Maurer et al. [BDPA11c, MRH04], which was first applied to hash functions by Coron et al. [CDMP05]. These results translate to security properties which are equal to the computational complexity of the generic attacks described in Sec. 3.3.2, if $c = 2n$ is chosen. The expected number of operations to find collisions, preimages or second preimages is as follows:

- Collision resistance: $O(min(2^{n/2}, 2^{c/2}))$
- Preimage resistance: $O(min(2^{n}, 2^{c/2}))$
- Second preimage resistance: $O(min(2^{n}, 2^{c/2}))$

Modifications The security proofs, especially the proof about indifferentiability from a random oracle, exclude virtually all of the weaknesses that the Merkle-Damgård and the Davies-Meyer constructions have [BDPA11c]. Yet, some modifications were developed. For example, Bertoni et al. proposed the duplex construction which is similar to the sponge construction [BDPA12]. It differs from the sponge construction in that it combines absorption and squeezing phases, i.e. it outputs r bits after each absorption of r bits and thus enables interesting additional applications, such as authenticated encryption. The security was proven to be fully inherited from the sponge construction [BDPA12].

Furthermore, Guo et al. modified the sponge function to improve the preimage resistance by using different rates for the absorption and the squeezing phases. This is used in the construction of the Photon hash function, which has a low capacity and thus would have a lower preimage resistance, or a lower throughput than with the modification [GPP11].

3.6 Tree-based Hash Functions

Iterated hash functions are impossible to implement in a highly parallel fashion, because of their inherent iterative nature, i.e. the message block x_{n+1} has to be processed by the compression function after message block x_n. Therefore, increasing the throughput of an implementation is usually only possible by improving the implementation of the compression function (or the internal permutation).

Larger speedups can be achieved, if several calls to an underlying primitive can be computed in parallel. This is were hash functions based on tree modes have their merit, because several nodes of the tree may be processed in parallel. However, designing a sufficiently performant and secure tree mode is non-trivial. Current work on tree modes is reflected in articles recently published by Bertoni et al. [BDPA09, BDPA13] and similar work by Dodis et al. [DRRS09].

The basic idea of tree modes is to change the iterative nature of conventional hash functions to a tree-based approach. Then the message bits are distributed over several nodes of the tree. The output of each node is connected to other nodes in the tree closer to the final node. Hence, all nodes which do not depend on the output of other nodes can be independently computed. While the potential speedup is quite high, the theoretical speedup cannot be linear in the number of parallel processing units, because of the need to consolidate the outputs of all nodes into the final node. The major drawback is the much higher memory demand, which make a tree-mode unattractive for most resource constraint environments, such as RFID tags or other low-budget embedded devices. Therefore, such hash functions are not investigated in the present thesis.

3.7 Applications

There are many applications of hash functions. The first applications were digital signatures [Mer79, DP84]. One important advantage for this application is that a signature is only calculated over the relatively short digest instead of calculating it over the whole message. This fundamental improvement made the RSA signature scheme feasible in relatively short time. Consider that at the time of the invention of RSA, one single RSA calculation with only a few bits took a very long time. Even much later it still took up to several seconds on a then modern Intel 80286 processor and thus, calculating RSA for a long message was considered impractical [BR89]. Unfortunately, this very high computation time is still true for many slow low-end microprocessors, without special hardware support.

However, beside the improvements for digital signatures, there is a large number of other important applications. Among these are the following applications. They are not necessarily tied to hash functions and may also often be constructed without hash functions, e.g. using block ciphers or other algorithms.

- *Message authentication codes (MAC)* are similar to digital signatures, e.g. the Keyed-Hash MAC (HMAC) construction [BCK96]. However, they are used with a symmetric key (keyed hash function) and therefore, it is impossible to tell, who actually generated the MAC, if the key was distributed to more than two parties.

- *Pseudo-random number generators (PRNG)* are a very important building block in modern cryptography. One of the most important properties of a PRNG is the uniformity of the output and hence, a hash function may be suitable to construct such a PRNG together with a true random number generator (TRNG) to seed the PRNG, if its output is not biased. One of many PRNGs based on this idea is described by Wang et al. [WZ10].

- *Challenge-response protocols* are used for authentication. If a proofer and a verifier previously exchanged a secret, it is easy to construct such a scheme using a hash function and a random number generator. The gist is, that the verifier sends the proofer a random challenge. The proofer combines this challenge with the previously exchanged secret, hashes this combination and sends the verifier the digest. The verifier can now check this result, without transferring the secret over the communication channel. This authentication scheme is for example used in the CHAP protocol [Sim96].

- *Key derivation functions (KDF)* are important for several cryptographic infrastructures, especially for symmetric crypto systems. The key functionality is to derive one or more (symmetric) keys from a common secret. A good KDF ensures, that it is impossible or at least very unlikely, that an attacker gets information about the common secret after she acquired successfully one of the derived keys. For example, such a hash-based KDF is described in [KE10].

Chapter 4

Hardware Design Aspects

4.1 Introduction

In this chapter, several aspects of hardware design and optimization will be introduced with a special focus on Field Programmable Gate Arrays (FPGA). Hardware and software design are significantly different from each other. While common hardware description languages such as VHDL [VHDL08] or Verilog [Ver05] offer the same computational power than for example the C programming language [Tur36], there are several important distinguishing characteristics. The model of Boolean circuits with memory defined in Ch. 2 already hints at the most important differences. Instead of a strictly sequential computation model, many operations can be computed in parallel in one clock cycle. This is naturally expressed in hardware description languages using an approach at the register transfer level (RTL) and can be only roughly approximated in procedural languages. A short introduction to the RTL design will be given, because all algorithm designs evaluated in this thesis were realized with the RTL methodology.

Another topic are several important optimization problems, which the hardware designer has to deal with. Some of these optimization problems can be approached in terms of the Boolean circuits model introduced in Sec. 2.2. Compared to software programming, there are again several differences, which also lead to diverging approaches for the optimizations. Software programmers usually try to minimize several important key characteristics of their programs, e.g. the code size, the memory usage or the runtime of the software. Hardware developers follow different best practices for the optimization of the circuits they design. While some are comparable to software optimization strategies, others are not accessible to software programmers. Usually, these techniques are specifically tailored to a certain optimization goal, e.g. low-latency, high-throughput, low-area or a combination of these. Furthermore, the target technology plays an important role. For example, sometimes the mapping to an FPGA makes it possible to use design strategies, which are not suitable to Application-Specific Integrated Circuits (ASIC) and vice versa.

In the following sections, all of these topics are covered in more detail. First, the register transfer level synthesis methodology is described in Sec. 4.2. This is then used to explain the different optimization techniques in Sec. 4.3 and Sec. 4.4. The last section in this chapter will cover the FPGA details (Sec. 4.5). the architecture of a contemporary FPGA will be described, enhanced with details for the Xilinx Virtex-5 architecture.

4.2 Register Transfer Level Synthesis

There are several different competing approaches for hardware developers to implement algorithms, each on a different abstraction level. High-level synthesis (HLS) starts with an abstract behavioral description of the algorithms in a high level language, such as C or Matlab and then works in several steps towards an implementation of the integrated circuit (Fig. 4.1) [MS09, CGMT09, Xil13b,

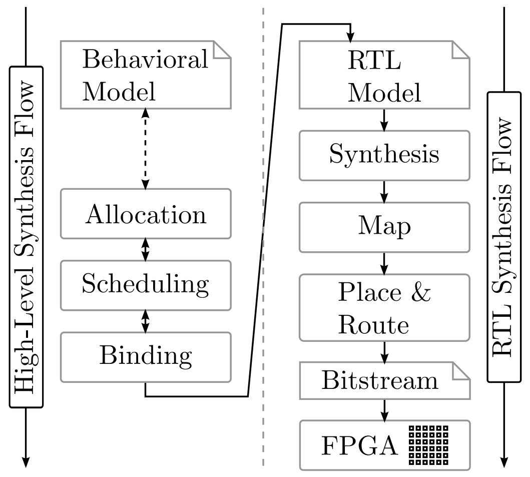


Figure 4.1: Hardware Design Flows.

Xil13a].

The high-level approach (left side in Fig. 4.1) adds several additional steps to the RTL methodology (right side in Fig. 4.1), foremost the automation of the allocation, scheduling, and binding tasks, i.e. the following tasks are automated [Xil13b]:

- The necessary hardware resources are allocated.
- The operations are scheduled, i.e. the execution time of each operation is fixed.
- The scheduled operations and the variables are bound to the allocated resources.

The output of this automated process is usually an architecture on the RT-level, which was already discussed, when the definition of Boolean circuits with memory was developed in Sec. 2.3. This connection to the theoretical model described in Ch. 2 is an important corner stone to the following evaluation part of this thesis.

The RTL synthesis starts with such a description (right side in Fig. 4.1). An engineer which develops at this level, has to solve the allocation, scheduling and the binding manually. This often results in a smaller or faster implementation at the expense of a usually longer development time compared to high-level synthesis [MS09]. Thus, this methodology is particularly interesting for high-volume products, because the unit cost will be lower. An RTL architecture consists of *registers* and *combinational circuits* between these registers [TLW+90].

The combinational circuits are divided into two parts, the *control logic* and the *data path*. Commonly, the control logic is built using a finite state machine (FSM). This controller takes care of the scheduling of I/O interfaces and also the data processing using counters and control signals, which are derived from the current state of the FSM and the input signals. The data path implements the processing of the input and intermediate data. Hence, the developer has to design three parts: a finite state machine, a memory layout, and a data path.

From this point on, the RTL architecture is transformed into lower-level descriptions. First, a netlist is synthesized from the RTL model which is then optimized using several low-level optimizations, e.g. logic optimization or register balancing. In the case of logic optimization, depth or area minimization are typical goals. Register balancing moves the registers in the circuit to reduce the latency between the register stages. The next step is the mapping to the physical instances followed by the place and route process, which finally maps the intermediate model to the physical implementation. The resulting layout for FPGAs is then stored in a bit stream file, which may be used to configure an FPGA.

While the design is gradually transformed, refined and optimized towards a physical implementation, there is also the need to verify the correctness of the results of each transformation and optimization. This may be either proven formally [Coh89] or the developer uses simulations [HP75, BR02]. The formal approach is the more rigorous process, but as the circuit grows, the computational overhead gets unmanageable. Therefore, the simulation based approach is commonly used, if a formal verification is not required.

4.3 Low-Level Optimizations

The low-level optimizations discussed in this section are targeted at the lower levels of the general methodology, foremost the logic optimization taking place during the RTL synthesis flow and at the place and route level (Fig. 4.1).

The low-level optimization of Boolean circuits is in general a computational hard problem, even if only combinational circuits without memory are considered. There are several problems involved, which are believed to be difficult or even proved to be **NP**-hard. Hence, it is often impossible to solve them efficiently, if the assumption $\mathbf{P} \neq \mathbf{NP}$ holds [GJ90]. Among these, the following two are probably the most important problems, which were already studied in classical complexity theory. Both problems are combinatorial optimization problems and several variants were proved to be **NP**-complete or **NP**-hard [Cou94].

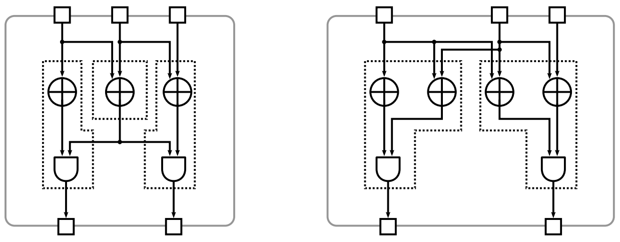
- *Minimization of the circuit size* is a hard problem with many variations, which was studied quite in depth [KC99, BU08, HW02]. For the practical usage, several minimization algorithms were developed, e.g. [DAR85, TNW96, MSBS93, Sas13].

- *Minimization of the circuit depth* is a problem which is not difficult in the classical sense of **NP**-hardness without also bounding the circuit size, because it is long known, that all Boolean formula can be represented in several normal forms (e.g. CNF, DNF) and then translated to depth two circuits using unbounded fan-in gates. These circuits can be realized with circuits of depth $O(\log_n)$, when no unbounded fan-in gates are allowed. However, these circuits have exponential size in the worst case. For example **Parity** $\notin$ $\mathbf{AC}^0$, because **Parity** has exponential size, if implemented with bounded depth [Sav97]. Hence, the problem has to be treated as size-depth minimization. Unbounded fan-in is also an unrealistic assumption, because unbounded fan-in gates obviously do not exist in practice as already mentioned in Ch. 2.

In the development of a real-world FPGA implementation, there are a lot of additional problems which are hard and have to be approximated. However, for many of these problems it is unknown, if there are approximation algorithms for these problems, which would allow the developer of the optimization algorithms to guarantee a minimum quality of the solution [ACK+02]. If such an approach is impossible, there are still generic solutions (e.g. greedy algorithms) which in many cases achieve results that are good enough for practical usages. Among other problems the following challenges are encountered:

- The circuit depth describes the delay of an electrical signal along the routing lines only on a very abstract level and in reality this model is inadequate because of these routing delays. Minimization of routing delays is likely to be an **NP**-hard problem, because it is a variation of the minimum edge-cost flow problem ([ND32] [GJ90]).

- The technology mapping from an abstract Boolean circuit model to FPGA primitives is **NP**-complete, if the optimization goal is area minimization. There exists a polynomial-time algorithm to optimally map a Boolean circuit to a LUT-based FPGA, if no parts of the circuit may be duplicated [CD94, MBV06]. However, it is possible to generate a smaller FPGA circuit by duplication of sub-circuits (Fig. 4.2). This problem is then **NP**-complete [FS94].



(a) Area minimization without duplication (b) Area minimization with duplication

Figure 4.2: Area minimized LUT-mapping.

- The maximum latency that a signal may take between two registers implies an upper bound on the clock frequency. This latency may be reduced by moving the registers forward or backwards in the circuit. However, moving a register reduces the latency of one sub-circuit, while increasing the latency of a different sub-circuit at the same time. This problem is also a variation of the minimum edge-cost flow problem [Gol97] ([ND32] [GJ90]).

- Register minimization is also an important problem. It is difficult because the number of theoretically possible states grows exponentially with the number of registers. A global minimization of registers is equivalent to the minimization of a deterministic finite automaton, which is solvable in polynomial time [HMU06]. However, if an implementation of an algorithm uses for example 100 registers, there are 2^{100} theoretically possible states. In most implementations only a subset of all states can be reached, but it is often difficult to determine which states can actually be reached and thus, the runtime of the polynomial time minimization algorithm is still very high.

Fortunately, the circuit designer does not have to solve these problems manually or to re-invent appropriate algorithms for every new design, because the tool chains of the FPGA vendors already cover a large part of the optimization process, e.g. the tool chain from Xilinx [Xil13c, Xil13a]. However, it is important that the hardware developer is aware of the different optimization problems, because a careful design may simplify the optimization tasks considerably and hence, improve the quality of the resulting FPGA design [Kil07]. It is also possible to manually improve a design by directly instantiating FPGA primitives, if the developer is able to identify specific weaknesses of the tool chains and the algorithms used [Ehl10].

4.4 High-Level Optimizations

Apart from the low-level techniques, there are other possible optimizations on the algorithmic level. These are partially connected to the three high-level steps allocation, scheduling and binding. If the developer writes code on the RT-level and not in a more abstract language, this is usually a manual task. Optimizations at this level often lead to better results than the automatic counterparts which use a higher level representation of the algorithm [PAFL98, PR09].

There are several types of optimizations. Which kind is used depends heavily on the optimization target, such as high performance, low area or a compromise between both. It is also important to distinguish between a more scientific exploration and a requirements driven approach typical for industrial products. The first usually tries to find the extreme cases and to classify these systematically, whereas the latter is more rigid and the developer has to meet the previously specified timing and other requirements, which often are derived from a previous research effort. In this thesis, the first approach is used to explore parts of the design space of each studied hash function.

The algorithmic high-level optimizations which are discussed in this thesis are *parallelization*, *pipelining*, *unrolling*, *serialization*, and *algorithmic specific optimizations*. Most of the general optimization techniques are already studied in detail in the literature, e.g. [HRG11, Kil07]. Yet, it is important to introduce these techniques as a basis for the following systematic evaluation.

4.4.1 Parallelization

As the name of the optimization technique already suggests, parallelization is about implementing an algorithm in parallel. However, there are several general possibilities and thus, the right choice of parallelization is non-trivial. The developer usually may choose one of the following possibilities or also a combination of them:

- Algorithmic-level parallelism
- Hardware duplication
- Pipeline-level parallelism

Parallelization on the algorithmic level is a hard problem, because it is non-trivial to optimally parallelize an algorithm which is described as a sequential program. Furthermore, as described in Ch. 2, it is unknown, if all problems in the complexity class **P** are efficiently parallelizable at all [GHR95]. Therefore, a thorough investigation of the algorithm is necessary to find parallelization possibilities. In particular, all functions that do not have data dependencies on the output of other functions may be easily parallelized (Fig. 4.4). Fortunately, for the case of cryptographic hash functions, it is usually possible to find the most important variants in a reasonable amount of time, because most such functions are based on a round function and this function may be usually computed in a parallel way.

Another possible approach to parallelization is to implement the same hardware multiple times, which is simple and effective to improve the throughput (Fig. 4.4). However, this approach is often expensive and does not decrease the latency of an implementation. The latency in this context is a measure to define the time between arrival of the input until the completion of the computation of the algorithm.

The last approach – pipeline parallelism – uses a pipeline, which can process several data streams in parallel (Fig. 4.5c). This is a special case of the pipelining which is described in the next section. Interestingly, many modern

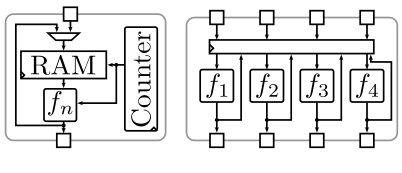


Figure 4.3: Algorithmic-level serialization and parallelism.

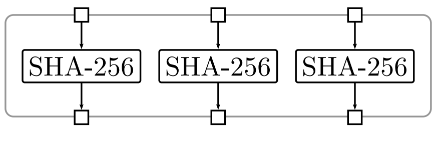


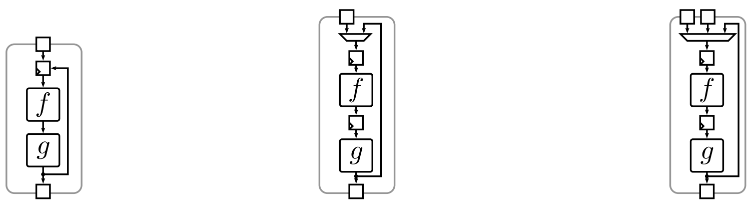
Figure 4.4: Hardware duplication of a SHA-256 implementation.

microprocessor architectures combine both pipeline based parallelism, e.g. using SMT [KST04] or Hyper Threading [KM03] and parallelism based on physical replication of the same processing modules, which is also known as multi-core architecture.

4.4.2 Pipelining

Pipelining and the timing of a circuit have a close relation. The general concept is based on the observation that a new input can be supplied to a circuit not only after the circuit evaluation is completed for the previous input, but much earlier, if the second input does not depend on the output of the computation with the first input. Keeping to the RTL methodology, the concept is translated to introduce additional registers into the circuit to shorten the worst case delay between two register stages (Fig. 4.5) [Kil07]. Then the second input may be supplied to the circuit as soon as the now shorter worst case latency has elapsed. However, the purely RTL-based approach is not the only possibility, because the routing delays themselves may serve as storage elements in an abstract sense [HE96].

As can be easily observed, the pipelining approach improves the clock frequency. However, there is also a drawback, because the total latency for the circuit evaluation does not decrease automatically, because data dependencies may introduce pipeline stalls, when no new input can be supplied, e.g. because the previous computation is not yet finished. This has to be carefully balanced, because the area overhead for the additional registers may easily counter the improved clock frequency, if there are too many pipeline stalls. The best case is achieved, if the clock frequency increases without introducing pipeline stalls, the performance scales linearly with the clock frequency.



(a) Standard circuit (b) Pipelined circuit (c) Pipeline parallelism

Figure 4.5: Pipelining a circuit.

Pipelining may also be used as an alternative method for parallelization as already mentioned in Sec. 4.4.1. The basic principle is to increase the number of pipeline stages by the number of parallel executable data sets. Then several independent data sets are processed in an interleaved but parallel fashion, i.e. in the first clock cycle the inputs to the circuit are taken from the first set, in the second cycle from the second set and so on (Fig. 4.5c). For this technique, the implementation is only extended by additional registers and a somewhat more complicated state machine and the duplication of a whole processing unit is avoided. This approach to parallelization is more complicated, but results in a better area-throughput ratio, if the area required for the additional registers and the more complicated state machine is smaller than using multiple identical processing units [Vad04].

4.4.3 Unrolling

Unrolling an algorithm means that several iterations of a loop are calculated in one clock cycle instead of iterating the same loop body over several clock cycles (Fig. 4.6). This increases the critical path and the area consumption of an implementation. For most algorithms, this leads to a reduction of the throughput-area ratio.

However, it also may be advantageous, if the loop body is not identical for each iteration and thus, a lot of multiplexers have to be used in the iterated variant. In a standard parallel implementation, these multiplexers lead to a long critical path. Unrolling potentially decreases the number of multiplexers and therefore, in some cases the total delay in the unrolled version is shorter than the summed up delay in the original version for the same number of loop iterations. Consequently the throughput increases, while the area grows less. Additionally, it is sometimes possible to efficiently use pipeline parallelism, and

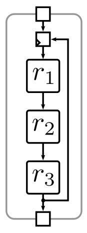


Figure 4.6: Unrolling a loop with three iterations.

thus to reduce the delay of the critical path again.

4.4.4 Serialization

Efficient serialization of an algorithm is often more complicated than a straightforward high-throughput implementation. Although a developer could implement every algorithm with only a state machine encoding a sequential program, one NAND (or NOR) gate, and a sufficient amount of registers[1], this approach would be very slow. Furthermore, it would also lead to the minimization problem of the sequential program, which is most likely a hard problem similar to the minimization of Boolean formula [BU08].

The approach to serialization followed in this thesis is a different, more algorithm driven approach. Most cryptographic algorithms, and especially all of the investigated hash functions consist of two core components, a state and a round function. The state translates directly into memory primitives, whereas the round function is split into a finite state machine, which controls the computation and the data path, which implements the internal operations of the round function. The design space exploration for serialization follows the following principle:

1. Identify parts of the round function which are similar enough to each other, such that they can be implemented by a single component.

2. Split and implement the round function in one or several sub-functions according to the analysis of (1).

3. Organize the state representation to satisfy the data dependencies of the sub-functions in the appropriate clock cycles.

This approach may be applied repeatedly to evaluate different architectures and different trade-offs. Of course, the first step is the probably hardest part and the other actions follow from the choices made at the beginning of the design phase. These choices lead to area savings in both implementation steps (2) and (3). Step (2) decreases the area required for the implementation of the combinational circuit and step (3) the size of the state memory. The latter

[1]Any Boolean formula can be represented using only NAND or NOR gates as has been proved by Sheffer [She13] and with more details by many others [Pos41].

savings are unique to FPGAs and usually can be observed less for ASICs, because all FPGA architectures under investigation have optimized hardware primitives for RAMs [Xil12c].

The approach is similar to the folding proposed by Homsirikamol et al. [HRG11]. In fact, all of the folding possibilities are also covered by the slightly different approach used here. The terms vertical and horizontal folding can be seen as instances of the proposed structured approach. Horizontal folding splits the round function into two parts, but the data path width stays the same, thus it mostly saves area in step (2). Vertical folding splits the round function into two parts and additionally, the data path is narrowed. In the terms of this thesis, the savings occur in step (2) and also step (3), because it reduces the data path width and thus, the RAM implementing the state can be smaller.

4.4.5 Algorithmic Specific Optimizations

The previously discussed optimizations are all generally applicable to most algorithms and implementations thereof. However, it is often possible to optimize specific parts of an algorithm. For example, based on mathematical properties of an algorithm, equivalent but smaller or faster circuits may be constructed, e.g. [Can05a, Can05b, CO09, Mon85]. Another possibility is to instantiate FPGA primitives for a part of an algorithm directly, e.g. [GGE09, JA11].

Such algorithm specific optimizations often lead to significant improvements, because only a small subset of a more general optimization problem is considered and furthermore, the problem is approached with different, more specific optimizations. The algorithms implemented by vendor tools usually also try to divide the general problem into smaller subproblems. However, an optimal division into subproblems is itself very difficult and thus, a manual intervention with domain-specific knowledge may improve the results.

The most important example in this thesis for such algorithmic optimizations is the AES S-box optimization using composite field representations, which was previously investigated by many authors [Rij00, Can05a, Can05b, MS03]. It is worth to reevaluate that approach for FPGAs, because the previous evaluations aim at ASICs and the technological dependence of the optimization may yield a different optimum for FPGAs [JR10a].

4.5 FPGA Hardware Aspects

FPGAs are significantly different from ASICs on the technological level. While in principle they can both be used to implement RTL designs, the actual technological mapping is quite different between both technologies. For ASICs the RTL description is mapped to a gate level netlist first. Then in the further design phases, this model is refined step by step to allow for fabrication at the end. This approach has many degrees of freedom and neither the placement, the routing nor the clock network is fixed in advance.

For FPGAs, this is considerably different. The layout of an FPGA is fixed, i.e. the hardware primitives, the possible interconnects between these primitives and the clock distribution network are fabricated in advanced. Hence, there are less degrees of freedom. In the final stages of the development of an FPGA implementation these primitives and interconnects are only configured appropriately. Using this configuration capability, it is possible to implement any algorithm in principle, if the capacity of the FPGA is sufficient.

Additionally, most FPGAs can be reconfigured either at startup or even partially at runtime. Thus, they position themselves between the fixed, and thus very efficient ASICs and flexible general purpose CPUs. However, in this thesis, the reconfiguration possibility is not used, and hence, it will not be further discussed.

Summarized, using FPGAs has several important advantages over ASICs or software implementations [KR07, Sem12]:

- Fast prototyping of hardware implementations.
- Cheaper than ASICs for low volume products.
- Changeable hardware implementations for new or updated features.
- Inexpensive bug fixing in hardware, due to reconfiguration.
- Usually faster than software implementations.

However, there are also several disadvantages:

- Usually slower than ASIC implementations.
- More expensive than ASICs for high volume products.

- Higher development costs than software implementations.
- Less flexibility than software implementations.
- Can easily be manipulated without special cryptographic protection.

4.5.1 General FPGA Architecture

The architecture of a modern FPGA consists of fixed blocks and interconnects between them (Fig. 4.7) [KTR08]. These blocks usually are one of the following types: I/O ports, generic logic primitives or more specialized hardware, such as larger blocks of RAMs or DSP units. Additionally, most FPGAs have specialized hardware for the efficient clock generation and distribution.

The logic primitives can be programmed to compute a small fixed subset of Boolean functions. For this functionality, each block consists of several smaller primitives. In most modern FPGAs, these include at least look-up tables (LUT), multiplexers (MUX) and flip-flops (FF). The routing channels provide the capability to connect blocks with each other. Hence, using the logic blocks and the interconnects, it is possible to build implementations of arbitrary algorithms, as long as the capacity of the FPGA is sufficient.

Larger RAMs can be implemented with the specialized RAM blocks. The major advantage is, that they are more efficient than the same implementation using the FFs embedded in the logic blocks. Similarly, DSP blocks contain fast

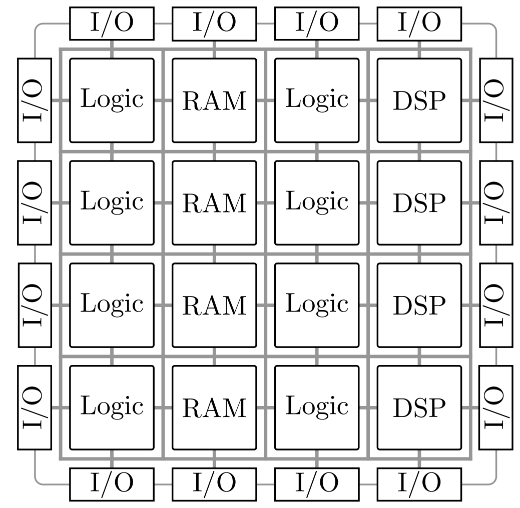


Figure 4.7: Modern FPGA architecture.

prefabricated multiply and add components, which can be used to optimize typical signal processing algorithms or other applications, which use a lot of multiplications and additions (e.g. the RSA signature scheme [RSA78]).

4.5.2 Xilinx FPGAs

The Xilinx Virtex-5 is a modern FPGA architecture, which is used as an example of a realization of the previously introduced ideas [Xil12c, Xil12d]. Most Xilinx architectures – and also FPGAs of other hardware vendors – are similar to the Xilinx Virtex-5 FPGA on an abstract level. Younger generations, such as the Xilinx Virtex-6/-7, provide the user with more flexibility and a better routing architecture. Additionally, they have a considerably higher maximum capacity due to the shrinking of the fabrication technology. Therefore, the maximum performance increases with each generation.

Another possibility is to look at older architectures such as the Xilinx Spartan-3. These FPGAs are smaller and provide less flexibility for the interconnect network. Hence, the performance is usually worse. Yet, the Xilinx Spartan-3 is still a frequently used FPGA, because of its competitive pricing. All those mentioned Xilinx FPGAs share the same abstract design principles and thus, it is valid to provide only the details of one architecture.

As already mentioned in the previous subsection, an FPGA consists of several different components. The parts of a Virtex-5 which are most significant to the following evaluation of cryptographic algorithms will be described in detail. However, a Virtex-5 FPGA contains additional resources, such as Phased Locked Loops (`PLL`) and Digital Clock Managers (`DCM`) for clock generation, specialized clock buffers (`BUFG`) to improve the clock distribution or I/O logic blocks for fast I/O operations with the outside world (e.g. `SERDES`) [Xil12c].

Configurable Logic Blocks All current Xilinx FPGAs contain a large number of so called Configurable Logic Block (CLB) slices. These slices make up the bulk of a Xilinx FPGA together with the interconnects between the slices. Each CLB slice contains several primitives, such as lookup-tables (`LUT`), multiplexers (`MUX`), registers, dedicated `XOR` primitives and fast carry chains. All of these primitives can be instantiated by the developer in a predefined way.

A simplified logical structure of a slice is presented in Fig. 4.8. Several

LUTs are contained in one slice, each followed by two registers, which may optionally store the outputs of the LUT. For Xilinx FPGAs the LUT is based on Static Random Access Memory (SRAM) and can be programmed to compute any Boolean function with a fixed number of inputs. For the Xilinx Virtex-5 FPGAs, this number is fixed to six inputs and two outputs. This so called LUT6_2 primitive consists of two separate LUTs with five shared inputs and one output, and a multiplexer which switches one of the two outputs based on the sixth input. Hence, it is possible to either implement a Boolean function with six inputs or two Boolean functions, sharing the same five inputs.

Additionally, a CLB slice contains a fast carry chain, which makes it possible to implement a fast ripple-carry adder. Actually, a ripple-carry adder implemented using these specifically design carry chains is faster for most FPGAs and common bit sizes than the theoretically better prefix adders, such as the carry-lookahead adder [XY98, VK07].

Distributed RAM All modern Xilinx FPGAs contain two types of CLB slices SLICEL and SLICEM [Xil05, Xil12c]. SLICEM slices extend the functionality of the SLICEL slices, such that they are also usable as memory primitives. This memory is called *distributed RAM*. This is realized by exporting the possibility

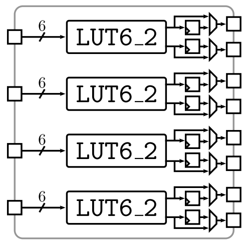


Figure 4.8: A simplified CLB architecture.

Table 4.1: Detailed area of distributed RAM in LUT-FF pairs with 256 Bit.

Dimensions	256×1	128×2	64×4	32×8	16×16
LUT6_2-FF pair	16	15	11	12	16
Dimensions	8×32	4×64	2×128	1×256	
LUT6_2-FF pair	24	46	89	256	

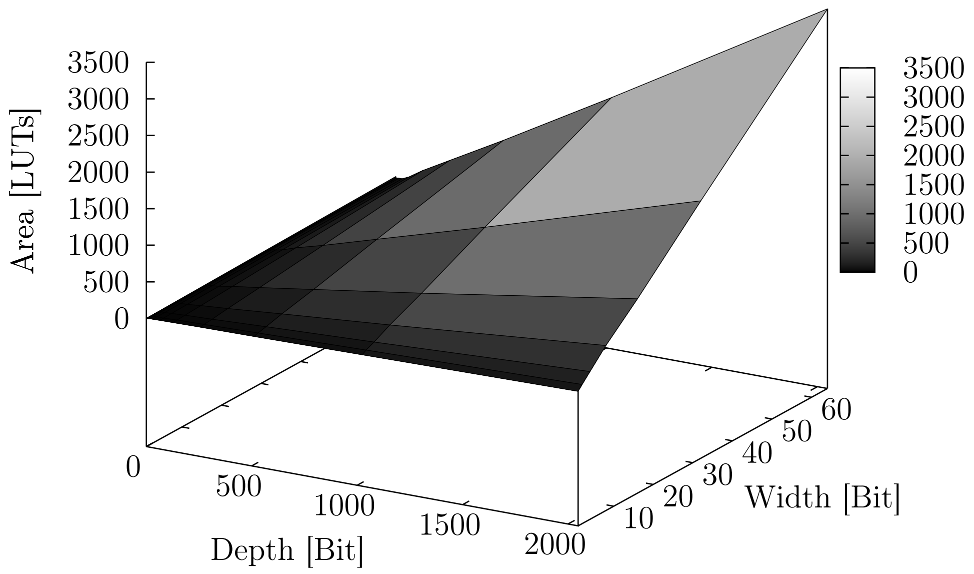


Figure 4.9: Area of distributed RAM in LUTs with different values for depth and width, synthesized for Xilinx Virtex-5 FPGAs.

to reprogram the `LUT6_2` as a user function. Hence, each `LUT6_2` in a `SLICEM` may be used as a 32×2 bit or 64×1 bit single port RAM [Xil12c].

Larger distributed RAMs may be constructed out of several `LUT6_2`/`SLICEM` slices. The total area depends on the width and the depth of the RAM, roughly according to the complexity measure SIZE_{RAM} (Def. 2.17). However, the real consumption does not grow strictly linearly with the depth and the width, because of the mapping to the LUT based structure of FPGAs. For example, implementing a 16×4 bit RAM needs two instead of one `LUT6_2`, because the hardware primitive supports only two outputs. In general, the number of `LUT6_2` primitives grows slightly over-proportionately, because of this effect (Tab. 4.1). Note that the variant 1×256 bit is implemented using slice registers instead of the `LUT6_2` instances.

Another similar effect can be observed, if the depth increases above 64 bit. Up to 64 bit, the output multiplexer is included in the `LUT6_2` instance. However, for larger depths, a cascade of additional multiplexers has to be instantiated and thus, the area increases again (Tab. 4.1).

Therefore, it can be concluded, that the optimal area consumption for a distributed RAM is achieved, if the depth d and width w do not diverge too much. This can be experimentally tested using the Xilinx tool chain for a Virtex-5 FPGA (Fig. 4.9).

Shift Registers Another possibility to store data inside an FPGA is the usage of `SLICEM` slices to construct shift registers. The shift registers have the same capacity as distributed RAM with the same number of SLICEM instances. However, they reduce the area consumption, because the multiplexer tree for the output can be much smaller in most cases. Shift registers do not provide a random access possibility and thus, they can only be efficiently used for straightforward linear access patterns. If random access is needed, then either a random access logic has to be added, which leads to an area overhead similar to the multiplexer trees for distributed RAM. Alternatively the shift register can be cycled to the correct output. This leads to additional clock cycles to access a specific memory cell.

Block RAM An alternative to distributed RAM is the Block RAM (BRAM) [Xil12c]. In contrast to distributed RAM, the granularity in which memory blocks can be configured is coarse. Each BRAM contains a maximum of $36,864$ bits configurable as 32768×1 bit, 16384×2 bit, 8192×4 bit, 4096×9 bit, 2048×18 bit, 1024×36 bit, 512×72 bit. It can be also configured as two independent RAM blocks with half of the bits each, with the exception of the 512×36 bit option, which does not exist.

If BRAMs are used, a fair comparison of different implementations is difficult, because they can be used to implement many different state sizes at the same cost, and worse for evaluations, BRAM can also implement parts of the algorithmic logic, or the state machine. Hence, designs using BRAM are not comparable to any other implementations, unless very strict evaluation guidelines are used. Otherwise, a design with a high throughput-area ratio, which typically only considers the number of slices as area, may seem to be very efficient, while in reality it is not. Therefore, *BRAM is not used* for evaluation purposes in this thesis.

Digital Signal Processing Blocks Similar to the BRAMs, there are hard blocks for the computation of important operations for digital signal processing (`DSP48E`) [Xil12d]. The most important functionality is the possibility to combine a multiplication and addition in one fast operation, which enhances the performance of many digital signal processing algorithms. In cryptography the DSP slices may be used to improve the performance of algorithms such as RSA [SKNI10]. Similar to the case of BRAM, the usage of DSP slices makes comparisons between two implementations very difficult and thus *DSP slices are also not used* in this thesis.

Part III

FPGA-based Evaluation

Chapter 5

A Systematic Design Approach

5.1 Introduction

In this chapter, a systematic design methodology is introduced, which is used in Ch. 6 to evaluate the six hash functions BLAKE, Grøstl, JH, Keccak, Skein and Photon. The core idea of this methodology is to analyze the algorithms in an abstract fashion, such that it is possible to compare different design approaches without implementing them all. Thus, in a sense, it is possible to weed out (some of) the bad candidate architectures. It is also possible to roughly estimate the throughput of an implementation, provided that the target frequency is reachable.

To achieve these possibilities, several estimates on key performance indicators are gathered, for example:

- The number of clock cycles to compute the compression function for one message block.
- The number and size of registers and the size of RAMs
- The data path width, and the organization of the state memory.

Note, that these are only indicators and a lot of the actual implementation performance depends on further implementation details, for example the maximum reachable clock frequency, or the size of the combinational parts of the algorithm. Furthermore, especially for lightweight implementations the external interface and the control logic plays an important role.

The remainder of this chapter is organized as follows. In Sec. 5.2 the motivation for the new methodology is provided and substantiated with previous work on the evaluation of FPGA implementations. In the next section the generic methodology is described in detail (Sec. 5.3). Based on this methodology, general properties of hardware architectures are derived, which follow from data dependencies inherent to algorithms. First, it is shown that a scheduling of operations without pipeline stalls implies a lower and upper bound on the number of clock cycles for a fixed round function implementation (Sec. 5.4), followed by a simple approach to design stall-free pipelined designs (Sec. 5.5). The last section describes how to formally determine the size of additional memories needed for the successful implementation of a round function (Sec. 5.6).

5.2 Motivation and Previous Work

The relevance of hardware implementations of cryptographic algorithms is high since the advent of modern cryptography. First, specialized hardware was necessary, because the general purpose hardware was too slow, e.g. the Data Encryption Standard (DES) was very slow on older processors and thus, hardware implementations were developed [HGD85]. Nowadays, microprocessors are much faster, and hence, software implementations of cryptographic algorithms are reasonable fast for most applications. Still, if the performance or the cost of software implementations is a major factor, then hardware implementations are needed, e.g. for network appliances [FPO05], or for the car-2-car communication [Sch11]. Therefore, it is important to evaluate the hardware performance of new algorithms.

Unfortunately, the evaluation of hardware implementations is tedious and error prone, because of many factors [Dri09]. Among these are:

- The target technology varies, e.g. the exact target FPGA, or the usage of DSPs, BRAMs and other special hardware primitives almost always differs between two evaluations.
- The parameters used in synthesis, map and place and route have a considerable impact on the results.
- The I/O interface has a high impact for lightweight implementations.

Another major problems is, that most evaluations are reporting results of standalone implementations. In contrast to these reports, it is unlikely, that a cryptographic algorithm is used in such a fashion and hence, it may be problematic to transfer the evaluation results into real world applications. For example, if the design relies on DSP slices, but other components integrated on the same FPGA are also using DSP slices, there may be a resource conflict and thus, a severe performance drop may be the result.

A typical evaluation based on implementation results can be enhanced and therefore become more meaningful, if it is based on an abstract and systematic methodology before the actual implementation. This can be very useful for selecting a candidate architecture, because it is possible to roughly estimate the area and the throughput of an architecture prior to implementation.

A step in this direction was made by Gaj et al. They identified different architectures to implement cryptographic hash functions, such as folding, unrolling, pipelining and circuit replication [GHR+12a]. Their methodology first finds possible architectures and then proceeds to implement those, and finally evaluates the implementations based on the post place and route results of the tool chain. While the approach is in principle sound, it has the drawback, that it is very time consuming to implement all of the possible architectures for a thorough evaluation.

Another methodology proposed by Jungk et al. takes this idea one step further in a more abstract direction by analyzing theoretical properties of the proposed architecture before starting to implement an architecture [JS13]. The design strategy was used to evaluate different architectures for the Keccak algorithm. It is particularly suitable to compare different architectures of the same algorithm and is also applicable to compare several algorithms in a limited fashion, because it evaluates the needed minimum memory size to implement an architecture, which has a large impact on the area of an implementation. In addition, the number of clock cycles is calculated explicitly up front and thus, the throughput of an implementation is roughly known before implementation.

In this thesis, the basis is the last methodology. An extended and revised version is discussed next in more detail and later applied to the cryptographic hash algorithms under investigation in Ch. 6.

5.3 Methodology Overview

All cryptographic functions investigated in this thesis have a common structure. From a high level engineering point of view, all designs can be seen as variations of the Merkle-Damgård construction (Sec. 3.5.1). Hence, they all include the concept of a compression function or a permutation. These internal functions are all based on the concept of a round function, which is iterated for a fixed number of rounds. Together with some pre- and post-processing of the input and the output, respectively, this computation of the round function forms the compression function and is thus central to the implementation of all algorithms.

For such algorithms, the methodology can be used to evaluate different hardware architectures. It consists of the following steps, which are also illustrated in Fig. 5.1:

1. Design an appropriate high-level organization for the internal state.
2. Estimate the number of clock cycles per round and for the complete compression function.
3. Develop an architecture for the round function to match the estimated number of clock cycles with the chosen state organization.
4. Determine the size of the memory needed for the computation of the round function.

As indicated in Fig. 5.1, this methodology can be repeated to analyze as many possibilities as desirable. However, finding the best approach automatically

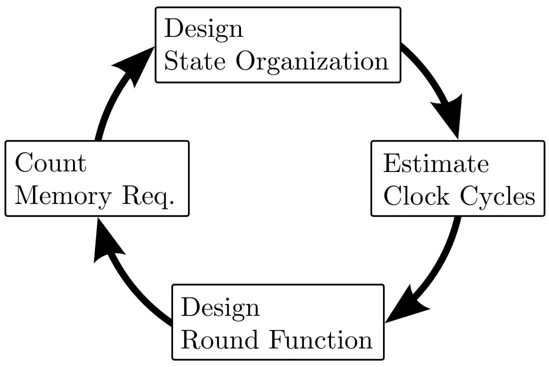


Figure 5.1: Systematic Evaluation Methodology.

this way is probably impossible. Yet, it can be of great help to identify an architecture out of several candidates, which fits the requirements of a later product development or the research goal, such as lightweight implementations. Therefore, the main task of the developer shifts to finding good candidate architectures. A step in this direction is to understand which parts of an implementation influence the area or the throughput in what way.

5.3.1 Area Impact

The area consumption is mainly influenced by the following components:

- State size
- Round function
- Control logic
- External interface

The state size (in bits) of most cryptographic functions cannot be reduced, because it is fixed by the specification of the algorithm. However, as discussed in Sec. 4.5.2, it is possible to reduce the area consumption when using FPGAs with distributed RAM.

The round function has also a high area impact, which cannot be disregarded. In contrast to the state, the round function can be tuned to the performance requirements, e.g. low-area or high-throughput using the optimizations discussed in Sec. 4.4. However, there are limits how much the area can be reduced and how much performance can be gained by using additional area. These limits have to be explored.

For lightweight implementations, the area reductions are limited by two factors. The first are data dependencies, which may force the developer to add either wait cycles or additional memory resources (registers or distributed RAM instances) and thus, further reducing the area becomes more and more difficult. The second limiting factor is, if the round function consists of several sub-functions which are hard to implement by shared logic. Then, it is next to impossible to save area by further serialization.

For high-throughput implementations, there are other issues. Since most algorithms have an inherent limit on the possible parallelism, using more area does not necessarily lead to a higher throughput. For example, unrolling the rounds does not always increase the throughput, because the depth of the circuit increases and hence, the maximum clock frequency decreases (Sec. 4.4.3). One possibility to counter this effect is pipeline parallelism (Sec. 4.4.2). However, if pipeline parallelism is used with only one data stream, it is again necessary to analyze the data dependencies between two or more rounds to verify that no pipeline stalls are introduced.

The control logic also influences the area requirements. However, since all implementations of all algorithms typically need a very similar control logic, this influence can be neglected for an abstract high-level evaluation. Similar, the external interface has a high impact on the area consumption, especially for lightweight implementations. Again, this is similar for all implementations and has a close connection to the control logic and hence, it can also be disregarded in the theoretical analysis for the same reasons.

5.3.2 Throughput Impact

The throughput of an architecture is both determined by the number of clock cycles to compute an algorithm and the maximum reachable clock frequency. From both measures, the practical relevance of the number of clock cycles is higher, because in many projects, the clock frequency is fixed by external requirements and also depends heavily on the target platform. Therefore, an abstract analysis should first focus on the number of clock cycles and only afterwards examine the clock frequency.

The minimum number of clock cycles is usually fixed by the width of the internal data path, because it defines how many bits can be read from the state in each clock cycle. Additionally, depending on the structure of the round function, there may be data dependencies, which force the developer to introduce wait cycles or to repeatedly read parts of the state and thus, the performance decreases. This can be observed when using a pipeline with too many stages (Sec. 4.4.2); in this special case, the additional wait cycles are the pipeline stalls.

Additional clock cycles are also necessary, if not enough temporary memory is provided and the same data has to be loaded from the state RAM in multiple clock cycles. This may for example happen, for a serialized implementation, which requires to store intermediate values between parts of a single round.

5.3.3 Performance Indicators

Based on the previous discussion, several performance indicators can be identified, which are important to rate the quality of an architecture for a particular algorithm:

- The number of clock cycles.
- The minimum memory size including its internal organization.
- The degree of serialization or unrolling.

The clock cycle count can be completely determined without transforming the architecture to an RTL model. As will be discussed in Sec. 5.4, it is also possible to evaluate, if an architecture is clock cycle optimal in some special sense.

The minimum memory size has a high influence on the area consumption. Hence, it is necessary to evaluate it and the memory organization. This may happen in an abstract fashion according to the complexity measures for memory introduced in Sec. 2.5. For the basic architecture, both the memory for the state as well as memories introduced in the round computation have to be evaluated to get to a meaningful performance indicator. In parts, this can be reduced to analyzing data dependencies, as will be shown in Sec. 5.6.

The last indicator is based on the round function architecture. This includes several slightly different things. For a serialized architecture, the first attribute is the width of the data path, because a smaller data path usually implies an area reduction in the implementation of the round function.

It is sometimes also possible to reduce the area of the round function without reducing the data path width. This usually has the additional benefit, that the maximum clock frequency increases, because the critical path delay in the round function usually reduces. For both techniques, the area reduction achieved for the round function is roughly proportional to the degree of serialization. However, the total area reduction for the complete algorithm cannot be proportional to this reduction, because the size of the state memory may only be reduced to a small degree in an FPGA implementation using distributed

RAM instead of registers and secondly, because the control logic may increase with a higher degree of serialization, e.g. a higher number of clock cycles leads to larger counters.

In the other direction a round function may be unrolled (Sec. 4.4.3 and the area consumption increases accordingly. However, as already mentioned it does not directly imply a higher throughput, as the critical path delay usually increases.

The degree of serialization which is independent of the data path width reduction, can be expressed in one metric and will be called the serialization metric s.[1]

Definition 5.1 *Let b be the internal state size of an algorithm, d the data path width of the architecture and $s \in \mathbb{N}$. Then s is called the* serialization metric*, if the number of clock cycles to compute the round function is proportional to $^{s \times b}/_{d}$.*

It is important to point out, that a reduction of the data path width is explicitly excluded from this serialization metric, based on the discussion in Sec. 4.4.4. A reduction of the data path width directly forces a serialized processing of the state. However, it is not necessary to reduce the data path width to serialize the processing. A good example is the round based processing of a compression function itself. It is possible to unroll the complete compression function and thus to reduce the number of clock cycles, but it is also possible to serialize the processing in such a way, that only one round is computed per clock cycle. The data path width stays the same in both cases. Therefore, it is convenient to analyze the serialization phenomenons separately.

This means, if $s = 1$, then the computation of the round function takes exactly $^{b}/_{d}$ clock cycles. If $s > 1$, then the algorithm was serialized and the round function is computed in $^{s \times b}/_{d}$ cycles and if $0 < s < 1$, then the architecture is an unrolled implementation. Hence, the metric s is proportional to the number of clock cycles that an architecture needs to compute an algorithm.

Based on these indicators, several candidate architectures can be partially compared. For example, if the goal is a high throughput implementation, then the cycle count is the most important indicator. Conversely, if area minimization

[1]Unrolling can be considered the inverse of serialization for this purpose.

is the optimization target, then the memory size and the serialization play an important role. Unfortunately, it is difficult to directly compare two different algorithms without knowing the approximate size of the round function and thus, the comparison easily leads to misinterpretations.

5.3.4 Data Dependencies

As noted in Sec. 5.3 for both the area and the throughput impact, data dependencies play an important role in the design of hardware architectures for both the area and the throughput. Hence, the evaluation should be focused on the data dependencies between the operations of the algorithm.

The main question behind the analysis of these dependencies is to find a feasible scheduling of the operations of an algorithm which avoids (pipeline) stalls. In general, this is a computationally hard problem, because it is a variation of the no-wait flow-shop scheduling problem ([SS16] [GJ90]). However, since the problem instances are small, it is usually possible to find an optimal solution manually using exhaustive search.

Data dependencies are inherent to the design of an algorithm. In particular, since all of the hash functions under investigation are based on a round function, the analysis is based on the notions of intra-round and inter-round dependencies as introduced in [JS13].

- If a round function r may be expressed in terms of sub-functions g_1, $\ldots, g_m$, i.e. $r = g_m \circ \cdots g_2 \circ g_1$ of m. Then *intra-round dependencies* are dependencies between the different g_i sub-functions. This may be a carry bit of an addition operation, or the round function consists of several different sub-functions which in parts depend on the output of other sub-functions.

- The *inter-round dependencies* are similar, but on the next functional level. If a compression function f can be expressed in an abstract fashion as $f = r_n \circ \cdots \circ r_2 \circ r_1$, where r_i are the individual rounds. Then inter-round dependencies are dependencies between two rounds r_i and r_j.

5.4 Cycle Optimal Architectures

If the operations are optimally scheduled, there are no pipeline stalls. This may be expressed more formally as follows. Let b be the internal state size of an algorithm, d the data path width of the architecture, n the number of rounds of the algorithm, and s a constant which describes the serialization metric according to Def. 5.1. Then an architecture implementing a compression function f may take about

$$\mathrm{cyc}_f(s,b,n,d) \approx \frac{s \cdot b \cdot n}{d}$$

clock cycles. In particular, the total number of clock cycles is proportional to the state size, the number of rounds and the degree of serialization and inverse proportional to the data path width.

Many architectures need some additional clock cycles because of the data dependencies and thus, they do not exactly meet the above estimated number of clock cycles. For example, this is always the case for pipelined implementations. This (small) overhead is considered optimal, if it is constant in terms of the number of rounds, because then no pipeline stalls are occurring between each round. This idea is formalized in the following definition:

Definition 5.2 *Let f be a compression function $f = r_n \circ \cdots \circ r_2 \circ r_1$. Then an architecture for r_i used to implement f is* optimal*, if there are* no pipeline stalls *when f is computed.*

An architecture fulfills this definition, if the following lemma holds:

Lemma 5.3 *Let b be the state size, d the data path width, n the number of rounds, s the serialization metric of an architecture, and c a constant. Then the number of clock cycles of an optimal architecture for $f = r_n \circ \cdots \circ r_2 \circ r_1$ is bounded by*

$$\frac{s \cdot b \cdot n}{d} \leq \mathrm{cyc}_f(s,b,n,d) \leq \frac{s \cdot b \cdot n}{d} + c,$$

if the number of clock cycles to compute one round r_i is bounded by

$$\frac{s \cdot b}{d} \leq \mathrm{cyc}_{r_i}(s,b,d) \leq \frac{s \cdot b}{d} + c.$$

Proof By assumption, every round costs at most $\frac{s \cdot b}{d} + c$ clock cycles. Hence, a first upper bound is $\text{cyc}_f(s, b, n, d) \leq (\frac{s \cdot b}{d} + c) \cdot n = \frac{s \cdot b \cdot n}{d} + c \cdot n$.

However, after $\frac{s \cdot b}{d}$ clock cycles, all inputs of round r_i have been read (because it is an optimal architecture) and now the inputs for round r_{i+1} are to be loaded next, otherwise there would be a pipeline stall. Therefore, it is necessary to interleave the computation of the rounds n and $n+1$ by c clock cycles, such that the outputs of round n are available as inputs for round $n+1$. Because of this interleaving, each round except one takes only $\frac{s \cdot b}{d}$ cycles and one round (e.g. the first or the last) needs $\frac{s \cdot b}{d} + c$ clock cycles. This proves the upper bound.

The lower bound for the compression function f follows from the lower bound $\frac{s \cdot b}{d} \leq \text{cyc}_{r_i}(s, b, d)$. □

Furthermore, the constant c cannot exceed a certain amount of clock cycles, because otherwise automatically inter-round dependencies will be violated and thus, pipeline stalls will occur. In Fig. 5.2 a scheduling of the operations g_0, g_1, g_2 with a maximum offset is shown, where $r_i = g_2 \circ g_1 \circ g_0$. The red line marks the last rising edge of the clock signal after the fifth clock cycle ($c = 2$) in which the output 3 is calculated. This idea can be formalized in the following corollary:

Corollary 5.4 *Let b be the state size, d the data path width and s the serialization metric of an optimal architecture. Then the constant overhead c of an optimal architecture is bounded by*

$$c < \frac{s \cdot b}{d},$$

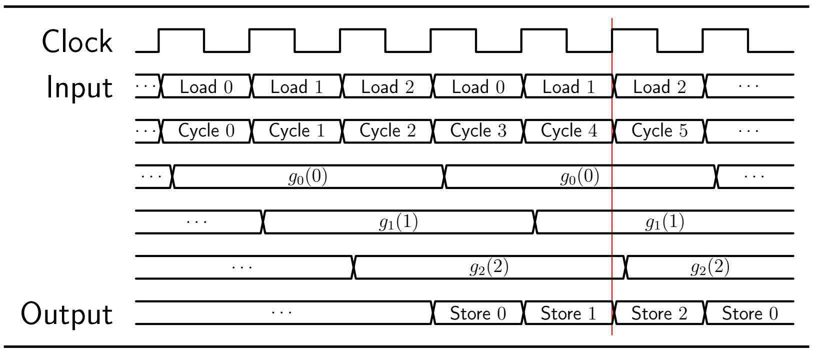


Figure 5.2: Example of a scheduling with maximum value for c.

if the number of clock cycles to compute one round r_i is bounded by

$$\mathrm{cyc}_{r_i}(s,b,d) \leq \frac{s \cdot b}{d} + c.$$

Proof Each computation of a round r_i takes at most ${}^{s\cdot b}/_{d} + c$ clock cycles. For each round r_i, ${}^{s\cdot b}/_{d}$ inputs are scheduled. Hence, the last output of r_{i-1} has to be stored at the latest after clock cycle ${}^{2s\cdot b}/_{d} - 1$ to be usable as an input for the next round, i.e. the worst case for c is $c \leq {}^{s\cdot b}/_{d} - 1$.

Now, suppose that $c \geq \frac{s\cdot b}{d}$, and that the last output of r_{i-1} is stored after clock cycle $\frac{s\cdot b}{d} + c$. Then $\frac{s\cdot b}{d} + c \geq \frac{2s\cdot b}{d}$ and at least one output bit of r_{i-1} is computed one clock cycle too late, because the result for this bit is not known$g_0(0)$, when it is needed to be loaded as an input bit for r_i. Hence, the pipeline stalls. This is clearly not optimal and thus $c < \frac{s\cdot b}{d}$. □

Note, that it is still possible to design a hardware architecture for a compression function without pipeline stalls, which violates this bound for the constant c. Such a design for the round function takes a varying number of clock cycles for the round function implementation and thus violates the assumption of Lemma 5.3 and Corollary 5.4. This leads to more conditional branching in the algorithm, which might increase the resource consumption for FPGA implementations because of the additional multiplexers. Furthermore, such an architecture would imply pipeline stalls between message blocks for designs which are able to process long messages as streams. Therefore, such design decisions should be avoided, if possible.

5.5 Stall-Free Pipelining

One side product of the methodology is an easy way to determine the maximum pipeline depth which can be achieved without introducing pipeline stalls and

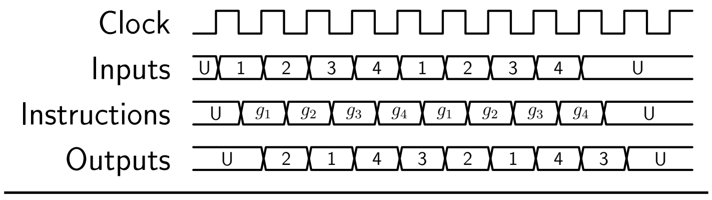


Figure 5.3: Example scheduling without pipelining.

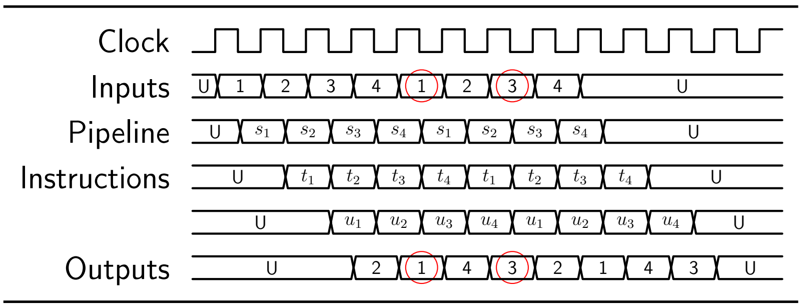


Figure 5.4: Example scheduling with a depth 3 pipeline.

without further changing the general architecture. For this goal, all outputs of the round function $r = g_m \circ \cdots \circ g_1$ have to be considered in the same order. In particular, if $y_1, y_2, \ldots, y_m$ are the outputs of $g_1, \ldots, g_m$, then for each output y_i a delta of clock cycles $\Delta_{\text{cyc}_{y_i}}$ can be calculated. This $\Delta_{\text{cyc}_{y_i}}$ is defined to be the number of clock cycle between the cycle in which an output y_i becomes available and the cycle when it is again an input for some function g_i. Then $\min(\Delta_{\text{cyc}_{y_1}}, \ldots, \Delta_{\text{cyc}_{y_m}}) - 1$ is the maximum number of additional pipeline stages that can be added to an architecture, such that the pipelined architecture has no stalls.

An example of this mechanism is given in Fig. 5.3 and Fig. 5.4. The first figure (Fig. 5.3) represents the scheduling of the operations $g_1, \ldots, g_4$ without pipelining. The second figure (Fig. 5.4) shows the pipelined version of this architecture, where $g_i = u_i \circ t_i \circ s_i$. The critical dependencies which prevent an even deeper optimal pipeline are marked in Fig. 5.4 with circles.

Despite that such a pipelined architecture is optimal in the sense of Def. 5.2, it is always possible, that a different scheduling of the operations computing the round r leads to a maximum pipeline depth of $^b/_d$ pipeline stages with a (possibly) non-regular structure. This strategy always works, because it is possible to schedule the outputs of the round function in such a way, that all outputs are exactly computed one clock cycle before they are again needed as an input. However, for this construction, at least the complete state has to be stored in the pipeline and furthermore, the architecture does not necessarily have a regular structure anymore. Hence, the implementation efficiency is often reduced.

5.6 Round Function Memory Estimation

For many architectures, it is necessary to include memory components in the computation of the round function to store intermediate results. These intermediate results have usually a close connection to the intra-round data dependencies. In particular, if a round function is split into several sub-functions, then it may happen, that one or more output bits of a sub-function are needed as inputs for another operation, which is scheduled in another clock cycle. If this is the case, than some additional memory primitive has to be included in the circuit.

More formally, let g_i be a sub-function that computes one part of the round function $r = g_m \circ \cdots g_2 \circ g_1$ in one clock cycle, implemented as purely combinational logic. Then this sub-function g_i may be modeled as a Boolean circuit C_{g_i}. Note, that this is exactly a sub-circuit of the complete BCM for the round function according to Def. 2.19, if each sub-function g_i is implemented using the same circuit. In the following, it is assumed for simplicity that all g_i sub-functions are identical and thus $g_i = g$.

If this circuit C_g is treated as a sub-circuit of a larger BCM C'_g according to Def. 2.19, then all outputs of the original circuit have to be either output gates or registers in C'_g. However, all outputs of this circuit are either stored in the state RAM – which is in the context of the round function a registered output – or used as inputs of another computation of g. Therefore, all of these output gates are registered outputs in C'_g and hence, all outputs of C_g are registers according to Corollary 2.21.

Furthermore, the state RAM has a fixed width w and thus, cannot store more than w bits per clock cycle. Hence, if $\text{FANOUT}(C_g) > w$, then at least $k = \text{FANOUT}(C_g) - w$ bits additional memory have to be provisioned in the circuit C'_g implementing the sub-round function. This memory may of course be modeled as a RAM according to Sec. 2.4 and can also be implemented as distributed RAM or BRAM on the FPGA to be more efficient.

Note, that these k bits correspond exactly to the intra-round dependencies between two or more (sub-)rounds, because only w new input bits will be read from the state RAM and thus, the additional intra-round dependencies cannot be fulfilled by the input bits supplied by the state RAM.

The situation does not change considerably for pipelined implementations. For each pipeline stage at least d registers have to provided, where d is the data path width. However, these registers are also covered by the approach, because the number of outputs of the combinational part of the circuit increases by at least d output gates for each pipeline stage and thus, the argument is still valid.

The same argument may be used to show, that for a maximum pipeline depth implementation as described in the previous section (Sec. 5.5), it is possible to reduce the memory requirements in abstract terms by b bits again. The first $^b/_d$ reads are performed without writing to the state RAM. Afterwards, all outputs are directly used as inputs for the round function again. Therefore, it is not necessary to write to the state RAM afterwards and hence, at least $b - d$ bits are stored in the pipeline. Since the output of the pipeline is directly fed back into the pipeline, it is enough to reduce the minimum memory requirements to b bits in total, instead of $2b - d$ bits. This means, that the state is always completely stored in the pipeline registers.

Chapter 6

Hash Function Evaluation

6.1 Introduction

In this chapter, the main evaluation of the six hash functions will be developed. For every analyzed hash function, the definition of the hash function will be provided, followed by a theoretical analysis based on the systematic approach introduced in Ch. 5. The main focus of the present evaluation are lightweight and midrange implementations. Therefore, the architectural serialization of the algorithms as discussed in Sec. 4.4.4 are analyzed in detail and high-performance approaches are only briefly discussed (Sec. 4.4), mainly because other researchers already conducted extensive evaluation of such high-performance implementations, e.g. Gaj et al. [GHR+12a, GHR+12b] investigated many variants of the SHA-3 finalists.

For some of the evaluated algorithms, it is also possible to apply one or more additional optimization techniques, such as the optimization based on mathematical observations for the AES S-box used by Grøstl or the optimizations based on direct instantiation of LUT primitives for JH and Keccak. The principles of both optimization techniques were previously described in Sec. A.5 and Sec. 4.4.5.

Five of the six evaluated algorithms were submitted to the SHA-3 competition. These algorithms were all part of the final round and hence, they received a lot of previous analysis. Therefore, the present analysis bundles already published ideas with several new ideas. The only algorithm under investigation that was not part of the competition is Photon. This hash function has been designed for lightweight applications such as RFID tags. The analysis is one of the first for of FPGA-based implementations.

The analysis of Keccak is also special, because after the announcement that Keccak was chosen as the new SHA-3 algorithm, the design has received and will continue to receive increased attention. This is also reflected in the evaluation of hardware implementations of the Keccak algorithm. Another reason for the extended study of Keccak is its flexibility, which allows the developer to easily tune the algorithm for lightweight implementations. In particular, it will be shown, that a lightweight Keccak implementation with a smaller state is also a good candidate for applications with a need for area-reduced hash functions, where it competes with algorithms like Photon, which are specially designed for lightweight applications. For the other algorithms the target is a midrange performance, i.e. the designs were developed to be small while still being efficient for a lot of applications. This is a reasonable compromise, because all SHA-3 finalists are too area consuming for extreme lightweight applications, since the state of all algorithms is quite large.

The remainder of this chapter is organized as follows. In Sec. 6.2, the bit string convention is defined, which is used for the external interfaces for the implementations. This definition is followed by a description of the external interfaces used by the implementations (Sec. 6.3). In the following sections (Sec. 6.5-6.10), the algorithms BLAKE, Grøstl, JH, Keccak, Skein and Photon are systematically analyzed according to the systematic approach developed in Ch. 5. This evaluation is organized for each algorithm as follows. First, the definition of the algorithm is provided. Then the theoretical analysis is developed, which is concluded with an evaluation. Afterwards, a description of one or more concrete implementation(s) round up each study.

6.2 Input and Output Bitstring Convention

Correct implementations of a hash function have to follow some conventions. Foremost the interpretation of input and output bit strings are to be defined. If an algorithm defines a different interpretation, a mapping between the two interpretations has to be provided. Throughout this thesis, the interpretation of bit and byte ordering defined by the NIST is used [Kay07], if not otherwise explicitly stated together with a mapping between the interpretations.

Definition 6.1 *A bitstring $x \in \mathbb{Z}_2^{\geq 0}$ is interpreted in big-endian byte order, i.e. if $x_1, \dots, x_n \in \mathbb{Z}_2^8$ are the $n = |x|/8$ bytes of the bitstring x, where x_1 is the most significant byte and x_n the least significant byte, then*

$$x =_{\text{def}} \sum_{i=1}^{n} x_i \times 256^{n-i} = x_1 || \cdots || x_n$$

Similarly, the bits $x_{i,1}, \dots, x_{i,8}$ in each byte x_i are ordered, such that

$$x_i = \sum_{j=1}^{8} x_{i,j} \times 2^{8-j}.$$

Intuitively, using a big-endian byte order means, that the most significant bits and bytes of a bitstring in a binary or hexadecimal notation are the left-most bits or bytes of the bitstring.

6.3 External Interfaces

The different hardware designs use three different, yet very similar I/O interfaces. The first interface utilizes the Fast Simplex Link (FSL) which is used by the MicroBlaze microprocessor from Xilinx [Xil12a, Xil12b]. The second one is based on an interface designed at the George Mason University (GMU). A slightly different variant of this interface was first used by Gaj et al. and then updated to the version used in this thesis [GHR10]. The third variant is an adapted version of the same interface to support the lightweight variants of the Photon hash function with a 4 bit wide interface.

6.3.1 FSL-based Interface

This interface is compliant to the FSL specification [Xil12a]. The FSL is a popular method to connect IP cores to the Xilinx Microblaze softcore processor [Xil12b]. The FSL is a generic 32 bit wide unidirectional link with an optional FIFO. Two synchronous links form the bidirectional interface. The subsets of signals used by the interface are listed in Tab. 6.1 and also displayed in Fig. 6.1.

The FSL works with a hand-shake protocol between master and slave as illustrated in Fig. 6.2 for the communication from master to slave [Xil12a]. For

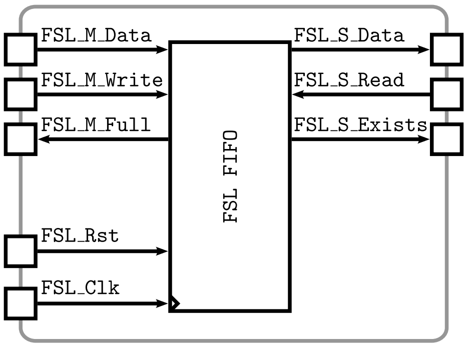


Figure 6.1: The Fast Simplex Link Interface.

transferring data from the microprocessor to the hash function, the microprocessor is the master. For the other direction it is the hash function implementation. In Fig. 6.2, the rising clock edges marked with a red line are writes to the FIFO and blue lines mark the reads from the FIFO.

The data format for the message input is depicted in Tab. 6.2a. The bit and byte order is implemented according to Def. 6.1. Each message block is transfered individually. First, the length information for a block is transfered. This length information is either the maximum block length, if the block has the maximum length, or otherwise the remaining number of message bits. Additionally, it is necessary for some algorithms to explicitly encode the end of the data stream in the length information. This end of data information is realized as a one bit prefix to the length, and since each algorithm has a fixed

Table 6.1: Implemented I/Os of the FSL Interface.

Signal Name	**I/O**	**Description**
`FSL_Clk`	`I`	FSL clock for synchronous FIFO mode
`FSL_Rst`	`I`	Peripheral reset
`FSL_M_Data`	`I`	Master input data (32 bits)
`FSL_M_Write`	`I`	Master writes data to the FIFO
`FSL_M_Full`	`O`	Master FIFO is full
`FSL_S_Data`	`O`	Slave output data (32 bits)
`FSL_S_Read`	`I`	Slave reads data from the FIFO
`FSL_S_Exists`	`O`	Data exists in the slave FIFO

width of the length information for one block, the position of this bit is also fix. All bits are right aligned in the first 32 bit wide transfer, i.e. the information is prefixed with zeros.

After the length information, the message block is transfered over the link in 32 bit chunks. If the message block is shorter than the maximum block length, it is padded with as many 0s as necessary to fill the message block. The actual padding according to the hash function specification happens later in the hardware implementation of the hash function. This principle simplifies the design of the state machine, because it does not have to cope with a differing number of transfers for message blocks of different length. As soon as a complete message block has been transfered, all implementations start the computation of the compression function automatically.

When the last block has been transfered, the hash function finishes the computation and then transfers the digest back over another FSL instance. This protocol is simpler, because the length of the digest is fixed. Thus, only the necessary amount of 32 bit blocks is transfered. For the 256 bit variants, this corresponds to eight transfers.

The implementations of the hash function using this interface and hence the area and throughput results only include the control logic for the FSL. The FIFO of the FSL implementation is not included, because it is configurable (e.g. the size and implementation style of the FIFO) and thus, the implementation details of the FSL link varies depending on the requirements of the application.

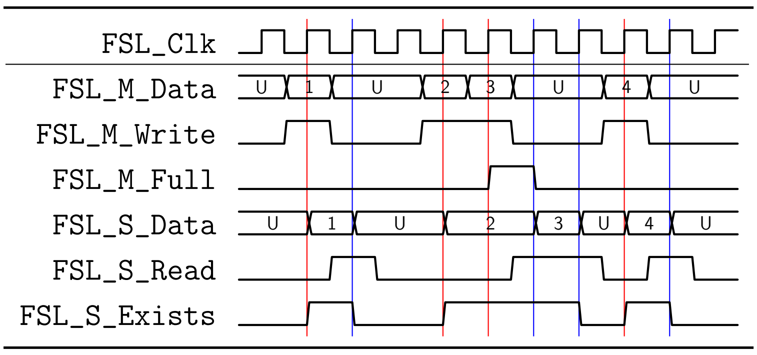


Figure 6.2: Timing diagram for the Fast Simplex Link.

Table 6.2: Protocol Messages for the FSL Interface.

(a) Message Input.

31	$0 \cdots 0 \| \| \mathrm{eof} \| \| \mathrm{length}$	0
MSB	message block	
31		0

(b) Digest Output.

MSB	hash digest	
31		0

6.3.2 GMU Interface

The second interface is based on the interface defined by Gaj et al. [GHR10]. The basic principle is similar to the FSL interface, i.e. it uses a hand-shake protocol between the two communication partners in a similar way (Fig. 6.3). One difference is, that it uses an active low signal to notify that data can be read from the pipeline. Furthermore, it supports a configurable link width.

The protocol part for transferring the message digest from the implementation to its user is identical to the FSL interface. However, the protocol to transfer data to the hash function is considerably different. In principle, it supports the transfer of already padded messages or unpadded messages to the hash function. For the present evaluation, only the first version for externally padded messages is used.

Two different ways of transferring parts of a message are supported (Tab. 6.3). The first possibility sends first a 1 followed by the length after padding (Tab. 6.3a). Then the length of the message before the padding is transfered

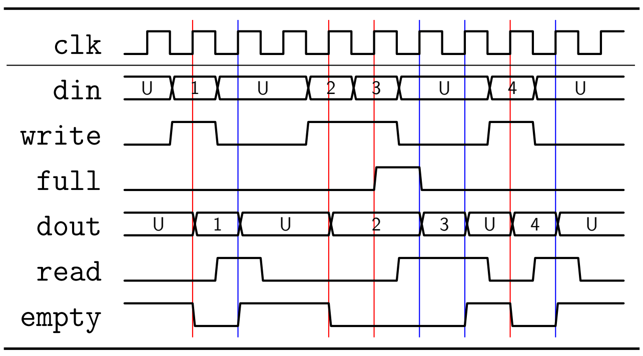


Figure 6.3: Timing diagram for the FIFO buffer supporting the GMU interface.

Table 6.3: Protocol Messages for the GMU-Based Interface.

(a) Message Input Without Splitting.

$w-1$	1||padded length	0
$w-1$	unpadded length	0
MSB	message	
$w-1$		0

(b) Message Input With Splitting.

$w-1$	0||segment length	0
MSB	message segment	
$w-1$		0

and finally the message itself. This may be used to transfer a message up to $2^{w-1}-1$ bits over the link.

The second possibility is to split the transfer into several segments, each prefixed with the length of the segment that is to be transfered next (Tab. 6.3b). To distinguish between the first and the second option a 0 is transferred first, followed by the segment length. Then the data of the segment follows.

The last message block always has to be transfered with the first variant. Otherwise, the hash function would not know the padded length and hence, produce false hash digests. Therefore, the second option can be either used as a way to hash messages that are larger than $2^{w-1}-1$ bits or when the exact message length is not known it advance.

This interface can be scaled for many different applications. For example,

Table 6.4: Protocol Messages for the Modified Lightweight Interface.

(a) Message Input Without Splitting.

$w-1$	1||high p. length	0
$w-1$	low p. length	0
$w-1$	high unp. length	0
$w-1$	low unp. length	0
MSB	message	
$w-1$		0

(b) Message Input With Splitting.

$w-1$	0||high seg. length	0
$w-1$	low seg. length	0
MSB	message segment	
$w-1$		0

for high-throughput implementations, a wide interface with 32 or even 64 bit is proposed by Gaj et al. [GHR10]. In contrast, for lightweight applications, it is better to have a narrow interface [KYS+11]. For the message block sizes of the five SHA-3 finalists, the smallest possible width is 16 bit. Using a smaller interface with an eight bit wide link a message block implies a maximum length of $2^7 - 1 = 127$ bits. Yet, the message blocks have 512 bits, which makes it impossible to transfer a message block over this link using this protocol. Only for some lightweight variants of KECCAK, an eight bit wide communication link is possible, because the rate (message block size) is reduced below 127 bits.

Further shrinking the link width is impossible without changing the protocol, because a four bit wide link only has three bit available to describe the message length in the last transfer. Hence, it may only describe a seven bit long message block. Therefore, the protocol was adapted for lightweight hash functions. The only difference is, that the length fields use two consecutive transfers (Tab. 6.4).

6.4 General Assumptions and Design Goals

Beside the byte order and the interface, there are several additional assumptions about the architectures and implementations:

- FPGA implementations are often used as application accelerators to offload computational intensive tasks to a specially designed hardware, e.g. [HV04]. Therefore, it is typically required for implementations to reach a throughput target performance. Hence, while reducing the area reduces the cost, a too narrow focus on the area consumption is an unrealistic assumption for such accelerators, if the throughput suffers too much. Therefore, none of the developed implementations is pushing the limits of the area consumption.

- Each RAM has at most one read and one write port. This allows for two architecture variants. Either one port which is both a read and write port or two ports, where one port is exclusively a read port and the other exclusively a write port. Other configurations are in principle possible, such as using two read ports. However, the analysis would become more complex.

- If a RAM is configured in a $t \times d$ fashion, then our optimal architectures read d bits per clock cycle when processing the round function. The last input is loaded for the clock cycle ${}^{s \times b \times (n_r+1)}/_{d} - 1$, where s is the serialization metric of the architecture, b is the state size, n_r is the round number and d is the data path width. Furthermore, after c clock cycles d bits are written to the RAM until clock cycle ${}^{s \times b \times (n_r+1)}/_{d} + c$.

6.5 BLAKE

The BLAKE hash function was a finalist in the SHA-3 competition, designed by Aumasson et al. The latest specification of the algorithm was submitted for the third round of the SHA-3 competition [AHMP10]. Compared to the previous second round version, the number of rounds was increased from 10 to 14 for the smaller option and from 14 to 16 for the larger variant.

BLAKE is based on the stream cipher ChaCha [Ber08a] and the HAIFA iteration mode [BD07]. BLAKE also uses a variation of the Davies-Meyer construction in its compression function (Sec. 3.5.2). ChaCha itself is a strengthened variant of the Salsa20 stream cipher [Ber08b], which is a profile 1 stream cipher from the eSTREAM portfolio [BBV12]. Profile 1 ciphers are recommended for software applications.

6.5.1 Definition

The following definition is a slightly rearranged version of the original definition [AHMP10]. The original specification defines four variants BLAKE-224, BLAKE-256, BLAKE-384, and BLAKE-512. BLAKE-224 and BLAKE-256 differ only in the initial values, the padding rule, and the truncation of the output to 224 instead of 256 bit.

The larger options BLAKE-384 and BLAKE-512 use a larger state and a compression function adapted to process the larger state. The differences between these two are similar to those between BLAKE-224 and BLAKE-256. For the sake of clarity and because it is the variant analyzed in detail, only the BLAKE-256 version will be defined explicitly.

Input and output mapping The byte order of the input messages and the output digests are organized according to the NIST specification (Sec. 6.2). The padded input for BLAKE-256 is split into 512 bit message blocks and further into 32 bit words. According to the BLAKE specification, a message block m in big-endian order consists of 16 words, i.e. $m = m_0||m_1||\cdots||m_{14}||m_{15}$, where each $m_i \in \mathbb{Z}_2^{32}$. The digest is organized as $h = h_0||\cdots||h_7$, where each $h_i \in \mathbb{Z}_2^{32}$.

Padding For BLAKE-256 the padding function $\text{pad}_{256}(M)$ is defined as follows. For $M \in \mathbb{Z}_2^{\geq 0}$ and $k = (-|M| - 66) \bmod 512$, let

$$\text{pad}_{256}(M) =_{\text{def}} M||1||0^k||1||(\lfloor M \rfloor_{64})$$

Initialization Values and Constants BLAKE uses several initialization values, which differ for each variant. The BLAKE-256 initialization values are the same as used by SHA-256 [FIPS 180-4]:

$\text{IV}_0 =_{\text{def}}$ `6A09E667`	$\text{IV}_1 =_{\text{def}}$ `BB67AE85`
$\text{IV}_2 =_{\text{def}}$ `3C6EF372`	$\text{IV}_3 =_{\text{def}}$ `A54FF53A`
$\text{IV}_4 =_{\text{def}}$ `510E527F`	$\text{IV}_5 =_{\text{def}}$ `9B05688C`
$\text{IV}_6 =_{\text{def}}$ `1F83D9AB`	$\text{IV}_7 =_{\text{def}}$ `5BE0CD19`

The constants for BLAKE-256 (and BLAKE-224) are the first digits of π:

$\text{c}_0 =_{\text{def}}$ `243F6A88`	$\text{c}_1 =_{\text{def}}$ `85A308D3`	$\text{c}_2 =_{\text{def}}$ `13198A2E`	$\text{c}_3 =_{\text{def}}$ `03707344`
$\text{c}_4 =_{\text{def}}$ `A4093822`	$\text{c}_5 =_{\text{def}}$ `299F31D0`	$\text{c}_6 =_{\text{def}}$ `082EFA98`	$\text{c}_7 =_{\text{def}}$ `EC4E6C89`
$\text{c}_8 =_{\text{def}}$ `452821E6`	$\text{c}_9 =_{\text{def}}$ `38D01377`	$\text{c}_{10} =_{\text{def}}$ `BE5466CF`	$\text{c}_{11} =_{\text{def}}$ `34E90C6C`
$\text{c}_{12} =_{\text{def}}$ `C0AC29B7`	$\text{c}_{13} =_{\text{def}}$ `C97C50DD`	$\text{c}_{14} =_{\text{def}}$ `3F84D5B5`	$\text{c}_{15} =_{\text{def}}$ `B5470917`

Round Function The round function used by the compression function of BLAKE is based on a family of G_i functions as defined in Alg. 6.1. In this algorithm a, b, c, and d are four words of the internal BLAKE state, r is the current round, σ_r is a permutation, which is different for each round (Tab. 6.5), c_i are the constants defined above and m is a message block.

Table 6.5: BLAKE's σ_r permutations.

σ_0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
σ_1	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
σ_2	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
σ_3	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
σ_4	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
σ_5	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
σ_6	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11
σ_7	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
σ_8	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
σ_9	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	0

Algorithm 6.1 BLAKE G_i function family [AHMP10]

```
Require: $a, b, c, d \in \mathbb{Z}_2^{32}$, $r \in \mathbb{Z}_{10}$, $\sigma_r : \mathbb{Z}_{16} \to \mathbb{Z}_{16}$ and $m = m_0||\cdots||m_{15}$, with $m_i \in \mathbb{Z}_2^{32}$
Ensure: $(a, b, c, d) \leftarrow G_i(a, b, c, d, r, m)$
    $a \leftarrow a + b + (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)})$
    $d \leftarrow (d \oplus a) \ggg 16$
    $c \leftarrow (c + d)$
    $b \leftarrow (b \oplus c) \ggg 12$
    $a \leftarrow a + b + (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)})$
    $d \leftarrow (d \oplus a) \ggg 8$
    $c \leftarrow (c + d)$
    $b \leftarrow (b \oplus c) \ggg 7$
    return $(a, b, c, d)$
```

Each of the eight G_i functions differs slightly from the others, depending on the σ_r permutations and the index i. The σ_r permutations control the injection of message bits into the computation and the usage of constants. Another interesting feature of the G_i functions is that each G_i function is a repetition of two almost identical sub transformations, where the parameters to σ_r and the rotation constants are different between the first and the second transformation.

The round function is computed according to Alg. 6.2. The input to the

Algorithm 6.2 BLAKE round function [AHMP10]

```
Require: $m = m_0||\cdots||m_{15}$, $h = v_0||\cdots||v_{15}$, with $m_i, v_i \in \mathbb{Z}_2^{32}$, and $r \in \mathbb{Z}_{10}$.
Ensure: $h \leftarrow \text{round}(h, r, m)$
    $(v_0, v_4, v_8, v_{12}) \leftarrow G_0(v_0, v_4, v_8, v_{12}, r, m)$
    $(v_1, v_5, v_9, v_{13}) \leftarrow G_1(v_1, v_5, v_9, v_{13}, r, m)$
    $(v_2, v_6, v_{10}, v_{14}) \leftarrow G_2(v_2, v_6, v_{10}, v_{14}, r, m)$
    $(v_3, v_7, v_{11}, v_{15}) \leftarrow G_3(v_3, v_7, v_{11}, v_{15}, r, m)$
    $(v_0, v_5, v_{10}, v_{15}) \leftarrow G_4(v_0, v_5, v_{10}, v_{15}, r, m)$
    $(v_1, v_6, v_{11}, v_{12}) \leftarrow G_5(v_1, v_6, v_{11}, v_{12}, r, m)$
    $(v_2, v_7, v_8, v_{13}) \leftarrow G_6(v_2, v_7, v_8, v_{13}, r, m)$
    $(v_3, v_4, v_9, v_{14}) \leftarrow G_7(v_3, v_4, v_9, v_{14}, r, m)$
    return $v_0||\cdots||v_{15}$
```

round function consists of the message block m, the internal state h and the current round $r \bmod 10$. A property of the round function, which will be used in the further analysis, is, that the inputs of $G_0, \dots, G_3$ are independent from each other and thus, they can be computed in any order, or also in parallel. The same applies to $G_4, \dots, G_7$. This is helpful for both lightweight and high-throughput implementations of BLAKE.

Compression Function The compression function is defined according to Alg. 6.3, where h is the previous chaining value, m is a message block, s is a salt, t is a counter of the processed message length and round(h, r, m) is computed as specified in Alg. 6.2. The compression function consists of three different parts, the initialization, the computation of the 14 rounds, and the finalization.

Algorithm 6.3 BLAKE compression function [AHMP10]

```
Require: $h = h_0 || \cdots || h_7$ with $h_i \in \mathbb{Z}_2^{32}$, $m \in \mathbb{Z}_2^{512}$, $s = s_0||s_1||s_2||s_3$, with $s_i \in \mathbb{Z}_2^{32}$,
    and $t = t_0||t_1$, with $t_i \in \mathbb{Z}_2^{32}$
Ensure: $h \leftarrow \text{compress}(h, m, s, t)$
    -- Initialization
    $v_0 \leftarrow h_0,$    $v_1 \leftarrow h_1,$    $v_2 \leftarrow h_2,$    $v_3 \leftarrow h_3$
    $v_4 \leftarrow h_4,$    $v_5 \leftarrow h_5,$    $v_6 \leftarrow h_6,$    $v_7 \leftarrow h_7$
    $v_8 \leftarrow s_0 \oplus c_0,$    $v_9 \leftarrow s_1 \oplus c_1,$    $v_{10} \leftarrow s_2 \oplus c_2,$    $v_{11} \leftarrow s_3 \oplus c_3$
    $v_{12} = t_0 \oplus c_4,$    $v_{13} \leftarrow t_0 \oplus c_5,$    $v_{14} \leftarrow t_1 \oplus c_6,$    $v_{15} \leftarrow t_1 \oplus c_7$
    -- Round Computation
    for $r = 0$ to 13 do
        $v = \text{round}(v, (r \bmod 10), m)$
    end for
    -- Finalization
    $h_0 \leftarrow h_0 \oplus s_0 \oplus v_0 \oplus v_8,$    $h_1 \leftarrow h_1 \oplus s_1 \oplus v_1 \oplus v_9$
    $h_2 \leftarrow h_2 \oplus s_2 \oplus v_2 \oplus v_{10},$    $h_3 \leftarrow h_3 \oplus s_3 \oplus v_3 \oplus v_{11}$
    $h_4 \leftarrow h_4 \oplus s_0 \oplus v_4 \oplus v_{12},$    $h_5 \leftarrow h_5 \oplus s_1 \oplus v_5 \oplus v_{13}$
    $h_6 \leftarrow h_6 \oplus s_2 \oplus v_6 \oplus v_{14},$    $h_7 \leftarrow h_7 \oplus s_3 \oplus v_7 \oplus v_{15}$
    return $h_0 || \cdots || h_7$
```

Algorithm 6.4 HAIFA iteration mode used by BLAKE [AHMP10]

```
Require: $M \in \mathbb{Z}_2^{\geq 0}$, $\mathrm{IV}_0, \ldots, \mathrm{IV}_7 \in \mathbb{Z}_2^{32}$, and $s = s_0||s_1||s_2||s_3$, with $s_0, \ldots, s_3 \in \mathbb{Z}_2^{32}$
Ensure: digest $\leftarrow$ BLAKE-256$(M, s)$
  $p \leftarrow \mathrm{pad}_{256}(M)$
  $\bar{t} \leftarrow |p|/512$
  $t \leftarrow 0$
  $h \leftarrow \mathrm{IV}_0|| \cdots ||\mathrm{IV}_7$
  for $i = 0$ to $\bar{t} - 1$ do
    if $i = \bar{t} - 1$ then
      $t \leftarrow t + 512$
    else
      $t \leftarrow t + |M| \bmod 512$
    end if
    $h \leftarrow \mathrm{compress}(h, p_i, s, t)$
  end for
  return $h$
```

Iteration Mode The HAIFA iteration mode used by BLAKE is defined according to Alg. 6.4. The iteration mode has a message M and a salt s as inputs. Furthermore, the eight initialization values $\mathrm{IV}_0, \ldots, \mathrm{IV}_7$ are necessary to initialize the internal state of BLAKE. The official submitted SHA-3 candidate with hash size $n = 256$ does not use a salt, i.e. it is defined to be BLAKE-256$(M, 0)$.

6.5.2 Systematic Evaluation Overview

The systematic evaluation of BLAKE-256 has been conducted according to the methodology introduced in Sec. 5.3. According to this methodology, the first step is to choose an appropriate state representation. As will be shown, this is possible in a generic way for many practicable architectures.

State and Memory Organization The state of BLAKE-256 consists of 512 bits organized in 16 words. Each word consists of 32 bits. This organization leads to several architectures for the state memory of the form $2^k \times (512/2^k)$ bits with the data path width $d = 512/2^k = 2^{9-k}$ bit. However, only the values $0 \leq k \leq 4$ are investigated here. It is more difficult for larger k to analyze the round function, because of the additional intra-round dependencies of the addition operation and the rotations and the benefits seem to be small.

Unfortunately, it is impossible to use a straightforward $2^k \times (^{512}/_{2^k})$ bit RAM in most cases to achieve an optimal architecture according to Def. 5.2, because the scheduling of the input variables to the G_i functions differs between the first $(0 \leq i \leq 3)$ and the second half $(4 \leq i \leq 7)$ of the round function. Hence, it would require a significant number of additional clock cycles or memory resources to reorder the RAM content appropriately. For the discussed variants, this argument leads to the following state organizations:

- For $k = 0$, the state organization $(1 \times 512$ bits) is ideal for high-throughput architectures and since the state is accessed fully in parallel, there is no need to logically divide the memory into more than one part. Such designs are called *parallel* architectures in the following discussion.

- For $k = 1$, two RAMs may be used to implement the 512 bit memory. Each RAM is organized as a 2×128 bits RAM. In this example architecture, the first RAM contains the tuples $(0, 1, 5, 12)$ in one memory location and $(2, 3, 7, 14)$ in the second. The remaining words are stored in the second RAM, i.e. the tuples $(4, 8, 9, 13)$ and $(6, 10, 11, 15)$. This design is called *intermediate serialized* architecture.

- For $k \in \{2, 3\}$, the $2^k \times (^{512}/_{2^k})$ bit memory can be organized with at most $^{16}/_{2^k}$ different $2^k \times 32$ bit RAMs. Such designs are also called *intermediate serialized* architectures.

  It is possible to group some of the RAMs in wider instances for some special architectures and thus, to save some area. This process of merging RAMs can also be achieved by the synthesis tool, if it can prove, that the read and write addresses are identical in all cases.

  However, one single RAM is in most cases insufficient, because of the different read and write patterns of the first and the second half of the round function. For example, Fig. 6.4 shows the different reading patterns for a non-pipelined architecture with $k = 3$. In this figure, the words with the grey background are loaded in the odd clock cycles and the words with the white background in the even clock cycles.

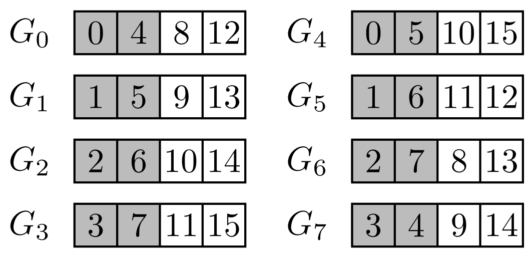


Figure 6.4: Example reading pattern for BLAKE-256 and $k = 3$.

Similar, for $k = 2$, the different read and write patterns for the first half of the round function and the second half remove the possibility to merge any of the four individual words needed to compute each G_i function into a RAM that is wider than a 32 bit word. Therefore, the words $0, 1, 2, 3$ can only be stored in one RAM together, $4, 5, 6, 7$ in the second RAM and so on.

- For $k = 4$, the RAM organization is simply $16 \times (512/16) = 16 \times 32$ and thus, one single RAM can be used, because each word of the state can be addressed individually. Architectures using this memory organization are called *fully serialized.*

In addition to the state memory, every BLAKE implementation has to provide memory to store the chaining value h, which is needed to correctly compute BLAKE's compression function. This amounts to additional 256 bits. Similar to the previous three cases for the internal state, the memory for the chaining value has several possible implementations:

- For $k \in \{0, 1\}$, a 1×256 bit memory is necessary, because it is impossible to find a scheduling of read and write operations for single-port RAMs for $k = 1$ that achieves an optimal architecture according to Def. 5.2.
- For $k = 2$, two 2×64 bit RAMs can be used. An example organization groups the two pairs $(h_0, h_2), (h_1, h_3)$ in two memory locations of one

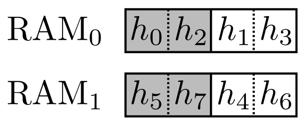

RAM_0 h_0 h_2 h_1 h_3
RAM_1 h_5 h_7 h_4 h_6

Figure 6.5: Grouping of the words h_i for $k = 2$.

RAM, and $(h_5, h_7), (h_4, h_6)$ in the other (Fig. 6.5). This facilitates the parallel computation of the new intermediate state h, i.e. in one clock cycles the words h_0, h_2, h_5, and h_7 are updated and in another clock cycle the other four words. With this organization, it is also possible to read the input to the first round in two consecutive clock cycles, i.e. the pairs (h_0, h_4), (h_1, h_5), and the other two pairs in the two following clock cycles.

- For $k = 3$, a similar approach as for $k = 2$ may be adapted with two 4×32 bit RAMs. This is for example possible, if h_0, h_1, h_2, and h_3 are stored in the first RAM and h_4, h_5, h_6, and h_7 in the second.
- For $k = 4$, the memory may be organized as a 8×32 bit RAM, because at most one 32 bit word is accessed per clock cycle.

A third important memory consumer is the input message block of 512 bit. This message block has to be stored until the computation of the compression function is completed, because the message injection happens during the round computation. In principle, the message may be stored in a RAM and accessed depending on the round function architecture. Unfortunately, multiple read ports are needed to use RAM resources, because the reading patterns for the message injection are not regular. The usage of such RAMs is ruled out by the earlier general design assumptions (Sec. 6.4), and hence, all architectures except the fully serialized ones use registers to store the message.

Another memory consumer is the counter t, which needs 64 additional bits and may be implemented either as a 64 bit register, or a 2×32 RAM. Hence, for the complete BCM there is a lower bound for the memory size of $\text{SIZE}_{\text{mem}}(C) \geq 1344$.

A practical implementation needs additional read-only memories (ROMs), which store the constants (256 bit) and the initialization values (512 bit). According to the definition of the $\text{SIZE}_{\text{mem}}(C)$ complexity measure, constants are not part of the memory estimate. However, they would be of course part of the total size complexity measure.

Clock Cycles Estimation Similar to the memory architectures, all investigated architectures for the round function are related to each other. Two different aspects are important for serialized architectures:

- Serialization of the G_i function. The degree of serialization by this method is expressed in the parameter α.
- Serialization of the round function, i.e. the scheduling of the different G_i instances of the function family. The degree of serialization achieved this way is expressed in the parameter β.

Both architectural aspects have cross-dependencies to the state organization, especially to the total width of the state RAM. In particular, some combinations are not efficiently implementable, i.e. the total width of the RAM has to correspond to the data path width of the round function architecture to be efficient. Hence, the serialization metric γ[1] according to Def. 5.1, can be calculated as follows for all discussed architectures, where d is the data path width, b is the state size and α and β describe the serialization options from above:

$$\gamma = \frac{d}{b}\alpha\beta$$

Using this serialization metric, the number of clock cycles to compute the round function fulfills the bounds from Lemma 5.3 and Corollary 5.4 for all discussed architectures. Hence, the number of clock cycles to compute the compression function is calculated as follows, where γ is the serialization metric, and d is the data path width. The state size is assumed to be 512 and the number of rounds to be 14.

$$\begin{aligned} \mathrm{cyc}_{\mathrm{compress}}(s,d) &\geq \frac{s \times 512 \times 14}{d}, \\ \mathrm{cyc}_{\mathrm{compress}}(s,d) &< \frac{s \times 512 \times (14+1)}{d}. \end{aligned}$$

Architectures for G_i The G_i function can be implemented in one clock cycle (Fig. 6.6) or serialized in several steps. One or several parts of G_i are then computed in a different clock cycle. The first option is to divide the computation of the G_i function in two parts. The computation of the shared implementation of the first and second part according to Alg. 6.5 uses the additional parameter

[1]Renamed from s to γ to avoid name clashes

Algorithm 6.5 BLAKE G_i function family with $\alpha = 2$

```
Require: $a, b, c, d \in \mathbb{Z}_2^{32}$, $r \in \mathbb{Z}_{10}$, $\sigma_r : \mathbb{Z}_{16} \to \mathbb{Z}_{16}$, $m = m_0 || \cdots || m_{15}$, with $m_i \in \mathbb{Z}_2^{32}$,
  and $j \in \mathbb{Z}_2$
Ensure: $(a, b, c, d) \leftarrow G_i(a, b, c, d, r, m, j)$
  if $j = 0$ then
    $\mathrm{sel}_0 \leftarrow 2i$, $\mathrm{sel}_1 \leftarrow 2i + 1$
    $\mathrm{rot}_0 \leftarrow 16$, $\mathrm{rot}_1 \leftarrow 12$
  else
    $\mathrm{sel}_0 \leftarrow 2i + 1$, $\mathrm{sel}_1 \leftarrow 2i$
    $\mathrm{rot}_0 \leftarrow 8$, $\mathrm{rot}_1 \leftarrow 7$
  end if
  $a \leftarrow a + b + (m_{\sigma_r(\mathrm{sel}_0)} \oplus c_{\sigma_r(\mathrm{sel}_1)})$
  $d \leftarrow (d \oplus a) \ggg \mathrm{rot}_0$
  $c \leftarrow (c + d)$
  $b \leftarrow (b \oplus c) \ggg \mathrm{rot}_1$
  return $(a, b, c, d)$
```

j, such that it is possible to select between the rotation constants, and also the inputs to the permutations σ_r.

Further serialization of the round function is also possible. In particular, the serialization factors $\alpha \in \{1, 2, 4, 8\}$ can be implemented, i.e. the G_i function is computed in one clock cycle ($\alpha = 1$) or serialized in 2, 4, or 8 clock cycles.

The scheduling of the individual operations of each step in a G_i function is more complicated for $\alpha \in \{4, 8\}$ compared to $\alpha \in \{1, 2\}$. For example, a possible, but simplified scheduling for $\alpha = 4$ is depicted in Fig. 6.7. In this architecture, the intra-round dependencies of the variable b force the developer to add a minimum overhead of one clock cycle. Furthermore, at least two additional 32 bit registers are needed to store the intermediate values of a in

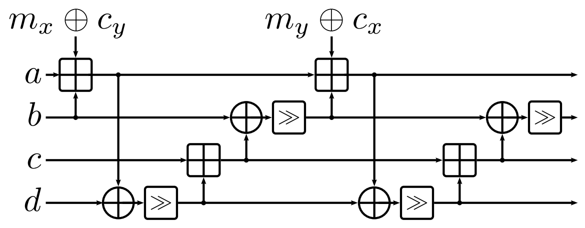


Figure 6.6: Illustration of the BLAKE G_i function.

the first clock cycle and b and d in the second clock cycle (64 bit in total). The register storing a in the first clock cycle may be reused in the second clock cycle to store either b or d.

For $\alpha = 8$, there is at least an overhead of one clock cycle, because of the intra-round dependencies of b. Furthermore, the value of b has to be stored (32 bit memory) for three clock cycles and since the first addition operation may not be computed before loading both a and b, one additional 32 bit register is necessary to temporarily store a. This amounts to 64 extra memory bits (Fig. 6.8).

Interestingly, it is possible for $\alpha = 8$ to create a shared implementation of the addition and the XOR operation in FPGA implementations for Xilinx devices, by explicitly instantiating the multiplexers in the carry chains. This makes it possible to deactivate the carry chain and thus to implement the XOR operation with the same hardware resources as the addition [BOY10].

As already mentioned, the serialization of the G_i function has dependencies with the state RAM width and the data path width, i.e. since not all inputs of G_i are processed in parallel for $\alpha \in \{4, 8\}$, the data path width shrinks and thus, also the state RAM width has to be reduced for an efficient architecture by a factor of two or four, respectively.

Round Function Architectures It is also possible to serialize BLAKE's round function by scheduling the individual G_i functions in different ways, e.g. to compute all in one clock cycle or to divide the computation in several clock cycles. This general principle can be repeatedly applied. Overall, there are four options $\beta \in \{1, 2, 4, 8\}$, i.e. either all eight G_i functions are processed in one clock cycle, or alternatively in two, four or eight clock cycles. Furthermore, it is

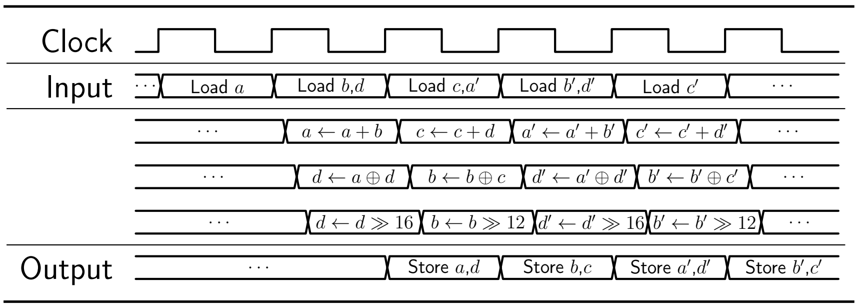


Figure 6.7: Simplified timing for the G_i function ($\alpha = 4$).

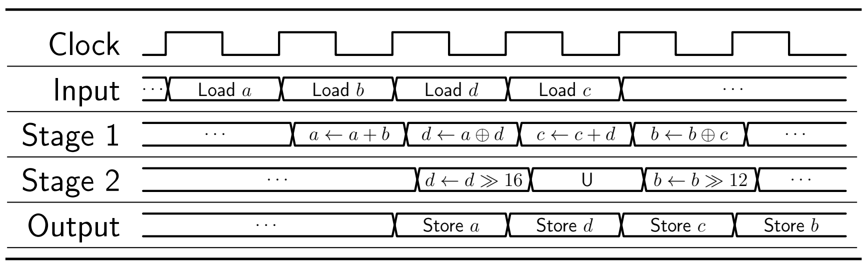


Figure 6.8: Simplified timing for the G_i function ($\alpha = 8$).

sometimes necessary to change the order of the computation of the G_i functions to facilitate more efficient pipeline architectures, e.g. for implementations with a 32 bits wide data path [BOY10].

Similar to the serialization of the G_i function family, the scheduling of the individual G_i instances has dependencies to the data path width and the state RAM width. For example, it is impossible to compute all G_i functions in one clock cycle, if the width of the state RAM is too narrow. If such a non-optimal architecture is implemented, it is impossible to achieve a clock cycle optimal implementation and thus, the resulting implementation would be inefficient.

6.5.3 Detailed Analysis

The following detailed analysis fits the previous analysis together for each investigated variant. In particular, many variants are described, where the memory organization works well together with the two serialization options described in the previous subsection. However, the details may differ slightly due to the analyzed pipelining depth.

Parallel Architecture ($k = 0$) An architecture with $k = 0$ reads the complete state in parallel, i.e. $d = 2^{9-0}$. This leads to a computation where at least the functions $G_0, \ldots, G_3$ are computed in parallel. Furthermore, it is possible to compute $G_4, \ldots, G_7$ in the same clock cycle ($\beta = 1$) or in the next clock cycle ($\beta = 2$).

The first option schedules all G_i functions in the same clock cycle ($\beta = 1$). This design option may only be combined with $\alpha = 1$, i.e. a fully parallel implementation of each G_i function, because a serialization of G_i introduces additional intra-round dependencies between the first four $G_0, \ldots, G_3$ functions

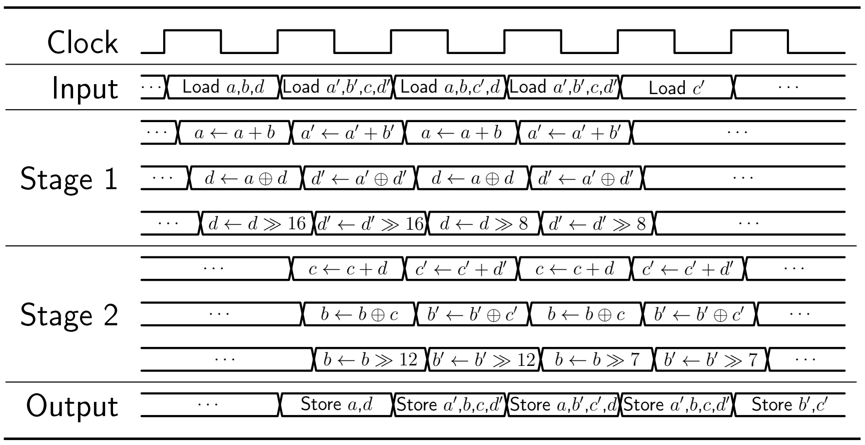

Clock
Input
··· Load a,b,d Load a',b',c,d' Load a,b,c',d Load a',b',c,d' Load c' ···
Stage 1
··· $a \leftarrow a + b$ $a' \leftarrow a' + b'$ $a \leftarrow a + b$ $a' \leftarrow a' + b'$ ···
··· $d \leftarrow a \oplus d$ $d' \leftarrow a' \oplus d'$ $d \leftarrow a \oplus d$ $d' \leftarrow a' \oplus d'$ ···
··· $d \leftarrow d \gg 16$ $d' \leftarrow d' \gg 16$ $d \leftarrow d \gg 8$ $d' \leftarrow d' \gg 8$ ···
Stage 2
··· $c \leftarrow c + d$ $c' \leftarrow c' + d'$ $c \leftarrow c + d$ $c' \leftarrow c' + d'$ ···
··· $b \leftarrow b \oplus c$ $b' \leftarrow b' \oplus c'$ $b \leftarrow b \oplus c$ $b' \leftarrow b' \oplus c'$ ···
··· $b \leftarrow b \gg 12$ $b' \leftarrow b' \gg 12$ $b \leftarrow b \gg 7$ $b' \leftarrow b' \gg 7$ ···
Output
··· Store a,d Store a',b,c,d' Store a,b',c',d Store a',b,c,d' Store b',c'

Figure 6.9: Simplified timing for the pipelined G_i function ($\alpha = 2, \beta = 4$).

and $G_4, \ldots, G_7$. These dependencies cannot be fulfilled in the same clock cycle with $\beta = 1$ and hence, the architecture cannot work.

For $\beta = 2$, the useful G_i serialization options are $\alpha \in \{1, 2\}$, i.e. either computing each G_i function in one clock cycle or splitting it in two clock cycles. Both options together lead to a total serialization factor of $s \in \{1, 2, 4\}$, because the data path is always 512 bit wide.

The minimum memory requirements apply for all BCMs C implementing one of the discussed architectures. Therefore $\text{SIZE}_{\text{mem}}(C) \geq 1344$.

Intermediate Serialized Architecture 1 ($k = 1$) For $k = 1$, the data path width is reduced from 512 bit to $d = 2^{9-1} = 256$ bits. Therefore, only one quarter of the round function can be computed in one clock cycle and at least four clock cycles are necessary to compute the complete round function ($\beta = 4, \alpha = 1$). Other architectures need at least the double amount of clock

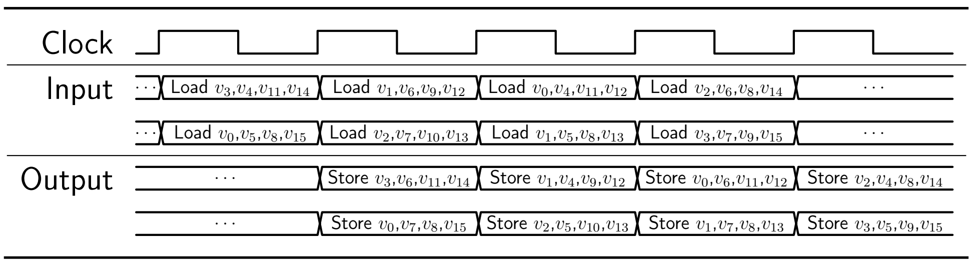

Clock
Input
··· Load v_3,v_4,v_{11},v_{14} Load v_1,v_6,v_9,v_{12} Load v_0,v_4,v_{11},v_{12} Load v_2,v_6,v_8,v_{14} ···
··· Load v_0,v_5,v_8,v_{15} Load v_2,v_7,v_{10},v_{13} Load v_1,v_5,v_8,v_{13} Load v_3,v_7,v_9,v_{15} ···
Output
··· Store v_3,v_6,v_{11},v_{14} Store v_1,v_4,v_9,v_{12} Store v_0,v_6,v_{11},v_{12} Store v_2,v_4,v_8,v_{14}
··· Store v_0,v_7,v_8,v_{15} Store v_2,v_5,v_{10},v_{13} Store v_1,v_7,v_8,v_{13} Store v_3,v_5,v_9,v_{15}

Figure 6.10: Reordered scheduling of the G_i functions ($\alpha = 2, \beta = 4$). Transition from (G_5, G_6) to (G_0, G_1)

cycles, because an architecture with $\beta = 2$ may only implement a G_i architecture with $\alpha = 4$ and the second possibility for $\beta = 4$ is $\alpha = 2$. There are several advantages of $(\beta = 4, \alpha = 2)$ over $(\beta = 2, \alpha = 4)$, therefore only the first of the two options is investigated further. For example, it is easy to add a pipeline step to the first architecture.

The best way to approach an efficient implementation is to implement two G_i functions, i.e. $\beta = 4$. This architectural decision can be combined with an implementation of the G_i function which takes one ($\alpha = 1$) or two clock cycles ($\alpha = 2$). For $\alpha = 2$, it is additionally possible to add a pipelining stage (Fig. 6.9) and thus, this architecture is in most cases superior to its larger sibling.

Two minor change is required for the pipelined version. It needs a reordering of the G_i functions to work without pipeline stalls. An example schedule is (G_0,G_1),(G_2,G_3),(G_7,G_4), and (G_5,G_6), with interleaved computation of (G_0,G_1),(G_2,G_3) and (G_7,G_4),(G_5,G_6), respectively. Fig. 6.10 exemplary depicts the transition from (G_5,G_6) to (G_0,G_1). The other transition may be easily checked in a similar way. Also the memory organization changes slightly for the pipelined version compared to the one presented above.

The lower bound estimate for a BCM $C_{non-pipelined}$ on the memory size of both non-pipelined versions is the same as for $k = 0$, i.e. $\text{SIZE}_{\text{mem}}(C) \geq 1344$. The pipelined version $C_{pipelined}$ adds the pipeline registers and thus $\text{SIZE}_{\text{mem}}(C_{pipelined}) \geq 1600$.

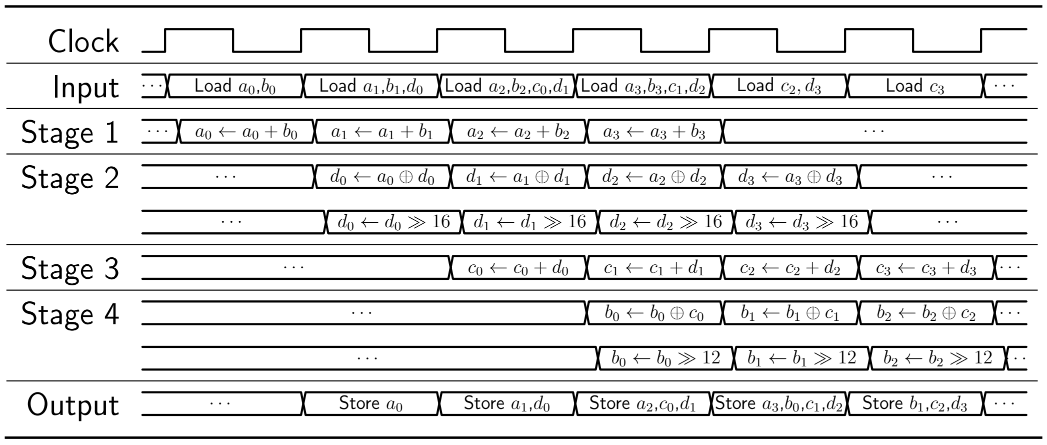

Clock
Input
Load a_0,b_0
Load a_1,b_1,d_0
Load a_2,b_2,c_0,d_1
Load a_3,b_3,c_1,d_2
Load c_2, d_3
Load c_3
Stage 1
$a_0 \leftarrow a_0 + b_0$
$a_1 \leftarrow a_1 + b_1$
$a_2 \leftarrow a_2 + b_2$
$a_3 \leftarrow a_3 + b_3$
Stage 2
$d_0 \leftarrow a_0 \oplus d_0$
$d_1 \leftarrow a_1 \oplus d_1$
$d_2 \leftarrow a_2 \oplus d_2$
$d_3 \leftarrow a_3 \oplus d_3$
$d_0 \leftarrow d_0 \gg 16$
$d_1 \leftarrow d_1 \gg 16$
$d_2 \leftarrow d_2 \gg 16$
$d_3 \leftarrow d_3 \gg 16$
Stage 3
$c_0 \leftarrow c_0 + d_0$
$c_1 \leftarrow c_1 + d_1$
$c_2 \leftarrow c_2 + d_2$
$c_3 \leftarrow c_3 + d_3$
Stage 4
$b_0 \leftarrow b_0 \oplus c_0$
$b_1 \leftarrow b_1 \oplus c_1$
$b_2 \leftarrow b_2 \oplus c_2$
$b_0 \leftarrow b_0 \gg 12$
$b_1 \leftarrow b_1 \gg 12$
$b_2 \leftarrow b_2 \gg 12$
Output
Store a_0
Store a_1,d_0
Store a_2,c_0,d_1
Store a_3,b_0,c_1,d_2
Store b_1,c_2,d_3

Figure 6.11: Simplified timing for the pipelined G_i function ($\alpha = 2, \beta = 8$).

Intermediate Serialized Architecture 2 ($k = 2$) The next smaller architecture narrows the data path down to 128 bits. This leads in principle to three general round function architectures, i.e. $\beta \in \{2, 4, 8\}$. Each architecture corresponds to a limited number of options for the implementation of G_i. Using $\beta = 2$ and $\alpha = 8$ four identical serialized implementations of G_i are used in parallel. In a similar fashion $\beta = 4$ and $\alpha = 4$ leads to two identical instances of G_i.

For the third choice ($\beta = 8$), there are two possibilities for the G_i implementation ($\alpha = 1$ or $\alpha = 2$). Both options are the most interesting choices and the only ones that are further analyzed, because the other options have similar or worse trade-offs compared to this case.

For both choices of α, the second option ($\alpha = 2$) is often the better variant. First, the clock frequency is higher due to the serialized G_i function and second, an implementation is smaller. In addition a shallow pipeline may improve the clock frequency further and hence, the throughput may be higher than for $\alpha = 1$ with less area consumption (Fig. 6.11). The pipelined architecture loads four 32 bit inputs for three different G_i instances per clock cycle and produces four outputs for four different G_i instances. Hence, the processing of a complete G_i function stretches over four clock cycles.

Similar to the pipelining for $k = 1$, the G_i functions have to be reordered for a stall free implementation (G_0,G_1,G_2,G_3,G_7,G_4,G_5,G_6). This change may be checked to yield a scheduling without stalls similar to the previous case.

The memory requirement of the non-pipelined architecture is identical to the previous cases, i.e. $\text{SIZE}_{\text{mem}}(C_{non-pipelined}) \geq 1344$. The pipelined version needs more memory. In particular, each additional pipeline stage adds 128 bits of registers. This amounts to $\text{SIZE}_{\text{mem}}(C_{pipelined}) \geq 1600$ for the described pipelined implementation.

Intermediate Serialized Architecture 3 ($k = 3$) For $k = 3$, the data path width is reduced to 64 bit. This data path width leads to only two possible architectures, i.e. $\beta = 4, \alpha = 8$ (two G_i function instances) and $\beta = 8, \alpha = 4$ (one G_i function). Only the option with $\beta = 8$ and $\alpha = 4$ is analyzed, because it is possible to easily pipeline this architecture, which is in many cases a big advantage. The general architecture for the G_i function follows Fig. 6.7.

One pipeline stage may be easily added, if the G_i functions are computed in an interleaved fashion and are reordered. This means in this case swapping the computation of G_6 and G_7. It is possible to check the correctness similar to the reordering for the other architectures.

For the non-pipelined architecture, at least two 32 bit registers have to be used in the round function according to the analysis in addition to the minimum memory requirements. The first 32 bit register is used to buffer the value of the input value for a and the second one to buffer the input value for b (Fig. 6.7). In total, the BCM $C_{non-pipelined}$ to implement the non-pipelined architecture has a memory lower bound of $\text{SIZE}_{\text{mem}}(C_{non-pipelined}) \geq 1408$. Each pipeline stage adds 64 bits, i.e. the described pipelined version takes $\text{SIZE}_{\text{mem}}(C_{pipelined}) \geq 1472$.

Fully Serialized Architecture ($k = 4$) For $k = 4$, the only possible choice is $\beta = 8$ and $\alpha = 8$, because it is impossible to implement other variants without pipeline stalls. The architecture follows Fig. 6.8 and thus, has a pipeline depth of at least 2.

For this approach, one 32 bit register is necessary to cache the second operand of each operation and a register to store the variable b temporarily for 3 clock cycles. This amounts to 64 bits of memory in addition to the minimum requirements for all architectures. Therefore, the amount of memory for a BCM C implementing this architecture has the lower bound $\text{SIZE}_{\text{mem}}(C) \geq 1408$.

6.5.4 Evaluation Summary

The analysis is summed up in Tab. 6.6. First the data path width is described with the parameter $d = 2^{9-k}$, the next two columns present the interesting architectures from the detailed discussion above in terms of α and β. The serialization metric γ is in turn the metric measuring the serialization, which is independent of the data path width reduction. This is especially interesting, because it determines together with the data path width the number of clock cycles necessary to compute the complete compression function.

The next column describes the minimum memory requirements for the particular variant. Then the pipeline depth which was analyzed is shown. Note that in the minimum memory requirements, pipeline registers are always included. The following three columns describe key performance indicators for the throughput, i.e. the number of clock cycles, the minimum overhead, which is in all cases optimal according to Def. 5.2, and the theoretical throughput for 100 MHz. The last information is particular relevant for architectures that have to achieve a certain throughput at a fixed clock frequency. However, this last indicator gives only a very rough estimate on the maximum throughput, because usually serialized and pipelined architectures reach a higher clock frequency than their parallel counterparts and thus, this comparison is only valid, if the clock frequency is fixed. For BLAKE, the theoretical throughput of the compression function does not differ considerably between short and long messages.

There are a few more options, for example it would be possible to relax the requirement, that in each clock cycle new data from the state RAM is read. Then other configurations are possible, because one can read an input value twice, e.g. the variable b in the G_i function or the message and constant injection can be scheduled in a separate clock cycle. The latter idea leads to architectures similar to the one proposed by Beuchat et al. which needs 80 clock cycles for the round function and thus 1120 clock cycles for the complete

Table 6.6: Summary of BLAKE-256 analysis

Data Path $d = 2^{9-k}$ [Bits]	α	β	γ	Memory [Bits]	ROM [Bits]	Analyzed Pipeline Depth	Clock Cycles	Overhead	Long Message Throughput @ 100 MHz [MBits/s]
512	1	1	1	1344	768	1	14	0	3657
512	1	2	2	1344	768	1	28	0	1828
512	2	2	4	1344	768	1	56	0	914
256	1	4	2	1344	768	1	56	0	914
256	2	4	4	1344	768	1	112	0	457
256	2	4	4	1600	768	2	112	1	457
128	1	8	2	1344	768	1	112	0	457
128	2	8	4	1344	768	1	224	0	228
128	2	8	4	1600	768	4	224	3	228
64	4	8	4	1408	768	2	448	1	114
64	4	8	4	1472	768	3	448	2	114
32	8	8	4	1408	768	3	896	1	57

compression function plus some overhead [BOY10]. Note, that the original architecture proposed by Beuchat et al. is based on an older BLAKE-256 specification with only 10 rounds and hence, takes only 800 clock cycles for the compression function.

6.5.5 Implementation

Two architectures for BLAKE-256 were implemented and evaluated [Jun12]. From the previous discussion these are the architectures with the following parameters:

- $d = 256$, $\alpha = 2$, $\beta = 4$
- $d = 128$, $\alpha = 2$, $\beta = 8$

Both implementations use the FSL-based interface described in Sec. 6.3.1. Hence, a padding unit is included as specified.

As already mentioned, the first architecture uses two G_i instances, whereas the latter uses only one copy of G_i. Each G_i function operates on 128 bit of BLAKE's 512 bit state. Both architectures are quite similar to each other, therefore only the second architecture is described in detail.

The BLAKE-256 design uses some of the properties that were already analyzed in the previous section to achieve an area-efficient design with reasonable throughput (Fig. 6.12):

- Implementing one half of a G_i-function as described above (The gray dashed box).
- Pipelining of the G_i function, therefore adding the pipeline registers.
- Rescheduled order of the G_i functions to ensure, that the pipeline never stalls.

All signals in Fig. 6.12 are 32 bit wide, except the signal from *pad* to t, which is only 9 bit wide.

For most LUT-based FPGAs the additional registers do not require a lot more area, because they can often be mapped together with the logic in the same slice (Sec. 4.5.2). At the same time, the pipelined computation of the

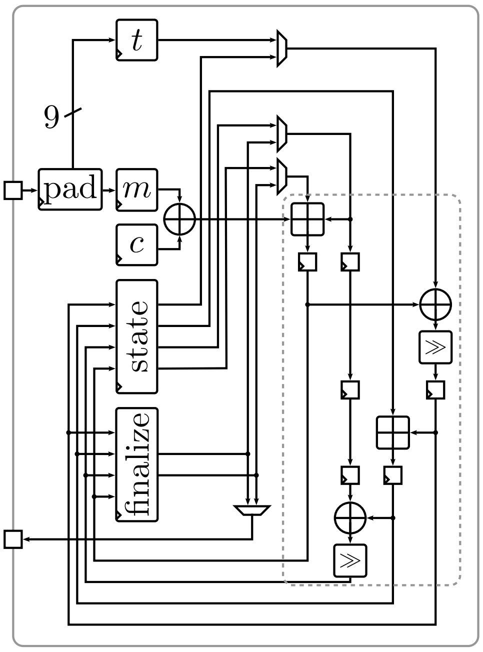


Figure 6.12: BLAKE-256 architecture with $\alpha = 2, \beta = 8$.

complete compression function only needs 4 additional clock cycles, while the clock frequency will be higher. Thus, the throughput-area ratio increases.

Another important feature of the design is its usage of distributed RAMs for the input message including double-buffering (m, 32×32 bit), the round constants (c, 16×32 bit), the state (four 4×32 bit RAMs) and the chaining value used in by the finalization (8×32 bit). The message length counter (t, 64 bit) is implemented with a 64 bit register.

The double-buffering of the message block is a possibility to improve the throughput, by loading the next message block, while the current block is still processed. Otherwise, the compression function would stall after the computation of one message block has finished, until the new message block has been loaded.

The compression function in this implementation needs a moderate number of clock cycles:

- In average, every G_i function evaluation takes 2 clock cycles and thus, to compute a whole round 16 clock cycles are needed.
- The round function is executed 14 times.
- Continuing with the next execution of the compression function is only

possible 4 cycles later, due to the finalization after each compression function invocation.

Thus, each computation of the compression function takes 228 clock cycles. Despite the 4 cycles overhead for the finalization the implementation is still within the theoretical optimal bounds for the compression function computation.

The second architecture ($\alpha = 2, \beta = 4$) differs from the described implementation only in details:

- Two half G_i functions are used in parallel.
- The state and finalize RAMs have the duplicate number of outputs and inputs. Therefore, the finalize RAM is implemented as a wide register as discussed above.
- The finalization needs 3 clock cycles. Therefore, the overall amount of clock cycles is 115.

6.6 Grøstl

The Grøstl hash function was developed by Gauravaram et al. and was one of the five finalists of the SHA-3 competition [GKM+10]. For the third round of the competition, there were some major changes to the algorithm, foremost, the permutation Q was changed to differ substantially from P. Furthermore, the constants added to the state in the AddRoundConstant function were changed.

The compression function of Grøstl is built from building blocks that are very similar to the AES block cipher [DR99]. Furthermore, the hash function uses a wide-pipe Merkle-Damgård construction as domain extender (Sec. 3.5.1). The submission specifies two variants. The smaller one can be parameterized to produce 224 or 256 bit hash digests, whereas the larger version may produce 384 or 512 bit digests. In the following sections, only the 256 bit option is discussed. However, the differences are rather small beside the larger state.

6.6.1 Definition

The definite reference for Grøstl is its specification [GKM+10]. The presentation of the definition in this thesis is slightly different to unify the style for all discussed algorithms. It is also limited to the description of Grøstl-256, because it is the only version analyzed in detail. However, the core ideas also apply to the other versions.

Input and output mapping The byte order of the input messages and the output digests are organized according to the NIST specification (Sec. 6.2). A message M with $|M|$ bits is thus defined as $M = \sum_{i=1}^{|M|/8} M_i \times 256^{|M|-i}$, where each $M_i \in \mathbb{Z}_2^8$ is a byte. After the padding, the message is split into message blocks of 64 bytes and since Grøstl's state is usually represented as a 8×8 bytes matrix, the 64 bytes of a message block $m \in (\mathbb{Z}_2^8)^{64}$ are mapped to the state h as follows:

$$h[i][j] \leftarrow m[8i + j]$$

The entries of h are defined to be interpreted as follows:

$$h =_{\text{def}} \begin{pmatrix} h[0][0] & h[1][1] & \cdots & h[6][1] & h[7][1] \\ h[0][1] & h[1][0] & \cdots & h[6][0] & h[7][0] \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ h[0][6] & h[1][6] & \cdots & h[6][6] & h[7][6] \\ h[0][7] & h[1][7] & \cdots & h[6][7] & h[7][7] \end{pmatrix}$$

Padding The input to the padding function pad is a message $M \in \mathbb{Z}_2^{\geq 0}$. The output is a padded message p with $l|(|p|)$, i.e. the size $|p|$ in bits is a multiple of l bits. The submission of Grøstl-256 fixes l for the Grøstl-256 variant to $l = 512$. Let $k = (-|M| - 65) \bmod 512$, then the padding function is defined as follows:

$$\text{pad}_{256}(M) =_{\text{def}} M||1||0^k|||M|_{64}$$

Initialization Values For each variant, the specification fixes a set of initial values. For the 256 bits variant, the initialization value is set to $\text{IV}_{256} = 0^{503}||1||0^8$. The state is initialized with this value before the first injection of a message block.

Round Function The round function consists of two permutations P and Q, which are computed independently. Each function consists of AddRoundConstant, SubBytes, ShiftBytes and MixBytes. However, the constants added in AddRoundConstant and the ShiftBytes permutation differ between P and Q. For AddRoundConstant the constants for P are defined as follows, where i is the current round number

$$C_P(i) =_{\text{def}} \begin{pmatrix} \mathsf{00} \oplus i & \mathsf{10} \oplus i & \mathsf{20} \oplus i & \mathsf{30} \oplus i & \mathsf{40} \oplus i & \mathsf{50} \oplus i & \mathsf{60} \oplus i & \mathsf{70} \oplus i \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} \\ \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} & \mathsf{00} \end{pmatrix}$$

And for Q the constants are defined as follows:

$$C_Q(i) =_{\text{def}} \begin{pmatrix} \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} \\ \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} \\ \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} \\ \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} \\ \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} \\ \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} \\ \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} & \texttt{FF} \\ \texttt{FF} \oplus i & \texttt{EF} \oplus i & \texttt{DF} \oplus i & \texttt{CF} \oplus i & \texttt{BF} \oplus i & \texttt{AF} \oplus i & \texttt{9F} \oplus i & \texttt{8F} \oplus i \end{pmatrix}$$

Beside the different constants, the AddRoundConstant transformation is just a binary XOR of the constants $C_P(i)$, respectively $C_Q(i)$ with the state $h \in \mathbb{Z}_2^{512}$:

$$\text{AddRoundConstant}_P(h, i) : h \leftarrow h \oplus C_P(i)$$
$$\text{AddRoundConstant}_Q(h, i) : h \leftarrow h \oplus C_Q(i)$$

The next transformation is SubBytes. It is based on the AES S-box, which is defined as follows [DR99, RD02]. Let $a \in \mathbb{F}_{2^8}$ be an entry of the matrix representation of the state h_P or h_Q, then the SubBytes transformation for a is defined as follows:

$$\text{SubBytes}(a) : \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} \leftarrow \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix}^{-1} \oplus \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

The subsequent ShiftBytes transformation rotates each row of the state matrix by a different number of bytes. These rotation values differ between P and Q. For P, ShiftBytes is defined as follows. Let h be a state matrix, i.e. $h \in ((\mathbb{Z}_2^8)^8)^8$. Then for all $0 \leq i \leq 7$, and $0 \leq j \leq 7$

$$\text{ShiftBytes}_P(h) : h[i][j] \leftarrow h[(i - j) \bmod 8][j]$$

For Q the transformation is defined for all $0 \leq i \leq 7$, and $0 \leq j \leq 3$:

$$\text{ShiftBytes}_Q(h) : \begin{cases} h[i][j] & \leftarrow h[(i-2j-1) \bmod 8][j] \\ h[i][j+4] & \leftarrow h[(i-2j) \bmod 8][j] \end{cases}$$

The last transformation is MixBytes. This transformation performs a matrix multiplication over the finite field $\mathbb{F}_{256}$ with the complete state.

$$\text{MixBytes}(h) : h \leftarrow \begin{pmatrix} 02 & 02 & 03 & 04 & 05 & 03 & 05 & 07 \\ 07 & 02 & 02 & 03 & 04 & 05 & 03 & 05 \\ 05 & 07 & 02 & 02 & 03 & 04 & 05 & 03 \\ 03 & 05 & 07 & 02 & 02 & 03 & 04 & 05 \\ 05 & 03 & 05 & 07 & 02 & 02 & 03 & 04 \\ 04 & 05 & 03 & 05 & 07 & 02 & 02 & 03 \\ 03 & 04 & 05 & 03 & 05 & 07 & 02 & 02 \\ 02 & 03 & 04 & 05 & 03 & 05 & 07 & 02 \end{pmatrix} \times h.$$

The round permutations for P and Q are defined based on these four permutations. One round of P is computed according to Alg. 6.6. The round function for Q is identical to the round function for P, except the usage of the Q variants of AddRoundConstant$_Q$ and ShiftBytes$_Q$.

Algorithm 6.6 Grøstl-256 P round function [GKM+10]

```
Require: $h \in \mathbb{Z}_2^{512}$, $i \in \mathbb{Z}_10$
Ensure: $h \leftarrow \text{round}_P(h, i)$
  $h \leftarrow \text{AddRoundConstant}_P(h, i)$
  $h \leftarrow \text{SubBytes}(h)$
  $h \leftarrow \text{ShiftBytes}_P(h)$
  $h \leftarrow \text{MixBytes}(h)$
  return $h$
```

Compression Function The compression function uses both permutations P and Q in parallel. However, the inputs to both permutations is different. After processing ten rounds, the outputs of P and Q are combined. Let h be the internal state m a message block, then the compression function is defined according to Alg. 6.7.

Algorithm 6.7 Grøstl-256 compression function [GKM+10]

```
**Require:** $h \in \mathbb{Z}_2^{512}$, $m \in \mathbb{Z}_2^{512}$
**Ensure:** $h \leftarrow \text{compress}(h, m)$
  -- Permutation $P$
  $h_P \leftarrow h \oplus m$
  for $i = 0$ to $9$ do
    $h_P \leftarrow \text{round}_P(h_P, i)$
  end for
  -- Permutation $Q$
  $h_Q \leftarrow h$
  for $i = 0$ to $9$ do
    $h_Q \leftarrow \text{round}_Q(h_Q, i)$
  end for
  return $h_P \oplus h_Q \oplus h$
```

Iteration Mode The iteration mode is a modified wide-pipe Merkle-Damgård design (Sec. 3.5.1). The final output is truncated from 512 to 256 bits for Grøstl-256. Before the truncation, an additional output transformation Ω has to be computed to thwart some length extension attacks. Thus, the iteration mode of Grøstl is slightly different then the plain Merkle-Damgård design and hence, it is separately described in Alg. 6.8.

Algorithm 6.8 Grøstl-256 iteration mode [GKM+10]

```
**Require:** $M \in \mathbb{Z}_2^{\geq 0}$
**Ensure:** $h \leftarrow \text{Grøstl-256}(M)$
  $p \leftarrow \text{pad}_{256}(M)$
  $h \leftarrow IV_{256}$
  for $i = 0$ to $|p|/512 - 1$ do
    $h \leftarrow \text{compress}(h, p_i)$
  end for
  -- Output transformation $\Omega$
  $h' \leftarrow h$
  for $i = 0$ to $9$ do
    $h \leftarrow \text{round}_P(h, i)$
  end for
  return $\lfloor h' \oplus h \rfloor_{256}$
```

6.6.2 Systematic Evaluation Overview

State and Memory Organization The total state of Grøstl-256 amounts to 1024 bits, equally shared between P and Q. There are several possible memory organizations. The following three variants may be used, if the P and Q are computed in parallel:

- A parallel implementation needs a 1×1024 bit memory organization.
- A 8×128 bit organization is suitable for a serialized implementation, where two columns are loaded per clock cycle, one column from h_P and the other from h_Q. However, because of the ShiftBytes permutation, the RAM has to be split into sixteen smaller 8×8 bit RAMs, i.e. one RAM for each row of the both states h_P and h_Q.
- For a byte-wise implementation a 64×16 bit RAM is sufficient, i.e. one byte from h_P and one byte from h_Q is loaded every clock cycle. However, this basic organization has also to be split into two 64×8 RAMs, because of the different ShiftBytes rotations.

These three variants may be adapted for architectures that interleave the computation of P and Q. Such implementation use a shared implementation of the round function to compute P and Q in alternating clock cycles. The general memory organizations differ from the parallel version insofar that the memory depth is doubled and the width is halved. The details also change for the serialized versions to eight 16×8 bit RAMs and one 128×8 bit RAM, respectively.

Additionally a memory for the chaining value h has to provisioned, which is organized either as a 512 bit wide register, a RAM with 8×64 bits, or 64×8 bits depending on the architecture. Hence, the total memory size of a BCM C implementing Grøstl-256 in this fashion is at least $\mathrm{SIZE}_{\mathrm{mem}}(C) \geq 1536$.

In contrast to many of the other analyzed algorithms, the Grøstl-256 round constants do not need any additional ROM, because they can be easily derived from the current round number and a couple of NOT gates.

State Read and Write Schedule The x and y coordinates of the data that has to be read from the states h_P and h_Q are used to derive the read addresses.

The write addresses are also derived in this way. However, it is not always possible to directly use the coordinates, because unprocessed data could be overwritten.

- For the two parallel versions, the scheduling is trivial. For the permutation P the full state h_P is read and written. The same applies for Q and its state h_Q.
- For the two column-wise serialized versions, the scheduling has to be carefully designed, such that no data is overwritten.

  For non-pipelined architectures, the write addresses in clock cycle $n+1$ are always identical to the read addresses in the previous clock cycle n. However, a stall-free architecture has to implement the ShiftBytes permutation by adjusting the read or the write addresses. As discussed, the write addresses are the same as the read addresses one clock cycle earlier, and thus, a round counter is required, which is used to derive different read addresses in each round according to the ShiftBytes permutation.

  If a pipeline with ideal depth is used, then the complete state is read before the first write. Therefore, the ShiftBytes permutation may be implemented by calculating appropriate write addresses and no further adjustment is necessary. However, the inter-round data dependencies only allow a stall-free pipeline for the P/Q-interleaved versions.
- The byte-wise serialized architectures solves the scheduling in a very similar way then their larger column-wise siblings.

Clock Cycles Estimation The number of clock cycles for Grøstl-256 is completely determined by the data path width, if no unrolling is used. All six variants lead to a clock cycle optimal implementation according to Lemma 5.3 and Corollary 5.4. Hence, the number of clock cycles to compute the compression function is bounded as follows, where d is the data path width, the state size is assumed to be 1024 and the number of rounds is assumed to be 10:

$$\mathrm{cyc}_{\mathrm{compress}}(d) \geq \frac{1024 \times 10}{d},$$
$$\mathrm{cyc}_{\mathrm{compress}}(d) < \frac{1024 \times (10+1)}{d}.$$

Basic Round Function Architectures The round functions of P and Q may be either computed in parallel, or in several serialized variants. The first serialization option splits the computation per column, leading to the computation of four, two or one columns in parallel. Since the basic architecture is very similar for all variants, only the version, which processes one column at a time is further analyzed. The second serialization possibility further splits the column into bytes.

The three basic architectures correspond to the choices for the state RAM organization described above. Together with the interleaved computation of P and Q, this leads to a total of six architectures for Grøstl-256 that are further analyzed.

6.6.3 Detailed Analysis

Parallel Architecture A parallel architecture for Grøstl-256 is a straightforward implementation of the specification. This architecture needs exactly 10 clock cycles to compute the compression function and after the last message block is absorbed, additional 10 clock cycles for the post processing step.

Algorithm 6.9 Shared Grøstl-256 round function [GKM+10]

```
Require: $h \in \mathbb{Z}_2^{512}$, $0 \le i \le 9$, $j \in \mathbb{Z}_2$
Ensure: $h \leftarrow \text{round}_P(h, i, j)$
  if $j = 0$ then
    $h \leftarrow \text{AddRoundConstant}_P(h, i)$
  else
    $h \leftarrow \text{AddRoundConstant}_Q(h, i)$
  end if
  $h \leftarrow \text{SubBytes}(h)$
  if $j = 0$ then
    $h \leftarrow \text{ShiftBytes}_P(h)$
  else
    $h \leftarrow \text{ShiftBytes}_Q(h)$
  end if
  $h \leftarrow \text{MixBytes}(h)$
  return $h$
```

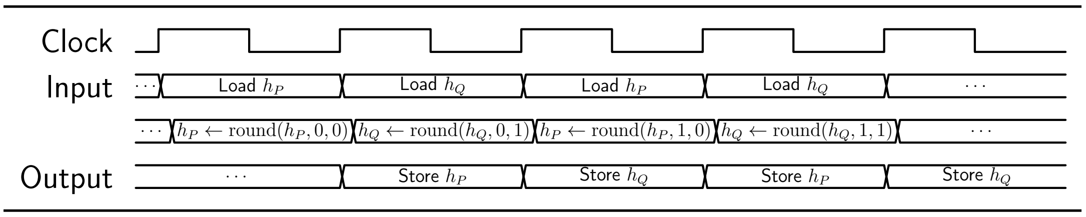

Clock
Input
Load h_P
Load h_Q
Load h_P
Load h_Q
$h_P \leftarrow \text{round}(h_P, 0, 0)$
$h_Q \leftarrow \text{round}(h_Q, 0, 1)$
$h_P \leftarrow \text{round}(h_P, 1, 0)$
$h_Q \leftarrow \text{round}(h_Q, 1, 1)$
Output
Store h_P
Store h_Q
Store h_P
Store h_Q

Figure 6.13: Simplified timing for the parallel interleaved implementation of the Grøstl-256 compression function.

A variation of this architecture is to interleave the computation of P and Q, such that the implementation may be shared (Alg. 6.9, Fig. 6.13). However, the differences of the AddRoundConstant and ShiftBytes permutations make it necessary to add several multiplexers to switch between the P and the Q permutation. Hence, the area consumption for the round function does not decrease proportionally.

The interleaved version can be improved by a shallow pipeline. Since there are no data dependencies between P and Q, it is possible to apply the abstract stall-free pipelining method described in Sec. 5.5 to detect the number of possible pipeline stages. In particular, analyzing the schedule depicted in Fig. 6.13, leads to one additional pipeline stage. Hence, the clock frequency can be increased significantly and the throughput-area ratio of the interleaved architecture may be better than the P/Q-parallel version. It is possible to hide the implementation cost of the pipeline stage by using distributed RAM for the SubBytes permutation. Then the pipeline stage is hidden in the read address registers of the RAMs.

The minimum memory requirements apply for any BCM $C_{non-pipelined}$ implementing one of the discussed non-pipelined architectures. Therefore $\text{SIZE}_{\text{mem}}(C_{non-pipelined}) \geq 1536$. The pipelined P/Q-interleaved architecture adds one pipeline stage and hence, a BCM $C_{pipelined}$ implementing it requires $\text{SIZE}_{\text{mem}}(C_{pipelined}) \geq 2048$.

Column-wise Architecture From the different serialization options, two are investigated more thoroughly. The first is the column-wise serialization, which computes only one column per clock cycle. The second is a byte-wise serialization which is discussed afterwards.

The column-wise serialization can be implemented by cutting the parallel

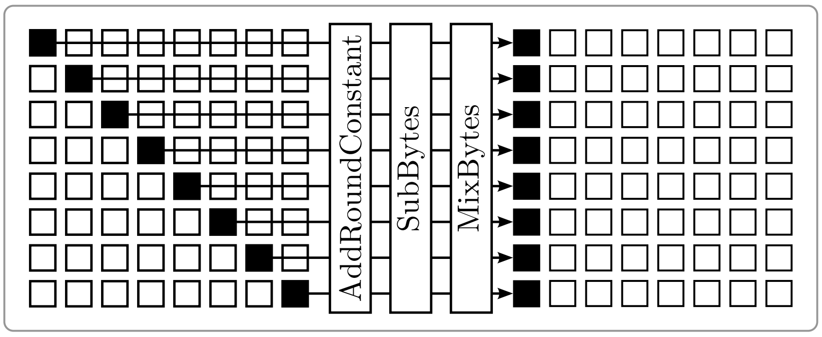


Figure 6.14: Grøstl-256 serialized architecture with implicit ShiftBytes for P.

implementation in eight parts and by scheduling each of these parts in a different clock cycle. It is important to take the ShiftBytes permutation implicitly into account by reading the eight bytes according to the shuffling by this permutation (Fig. 6.14). Then each processed column has no intra-round dependencies with any other column and thus, may be processed in one clock cycle.

For the column-wise serialization, interleaving P and Q works similar to the parallel architecture and produces the same benefit. Since there are no dependencies between P and Q, the method from Sec. 5.5 can be easily applied again. In this particular instance, the result of the analysis is, that a maximum of eight pipeline stages are possible. Hence, the maximum throughput of the architecture is greatly increased.

A BCM $C_{non-pipelined}$ implementing the column-wise architecture can also be implemented using the minimum memory requirements, i.e. $\text{SIZE}_{\text{mem}}(C_{non-pipelined}) \geq 1536$. The maximum pipelined version needs 64 memory bits for each pipeline stage, i.e. $\text{SIZE}_{\text{mem}}(C_{pipelined}) \geq 1984$.

Byte-Wise Architecture The byte-wise serialization option takes the approach one step further by serializing the computation on a byte by byte level. However, the MixBytes permutation now exhibits intra-round dependencies between the individual bytes of a column, which have to be resolved. Therefore, an overhead of seven additional clock cycles is always necessary to compute the round function (Fig. 6.15). Furthermore, several additional memories are needed to store the intermediate values. This amounts to 128 bits to store the values for all eight bytes for both permutations. A fast and small stall-free implementation additionally needs 128 bits, otherwise the MixBytes permutation can only be computed when a complete column is loaded. However, the 128 bits are not required.

The pipelining options for the byte-wise architecture are similar to the previously described architectures. The default architecture for a parallel implementation of P and Q already has the maximum pipeline depth, because the last output $h_P[7][7]$ is stored in the same clock cycle, when it has to be loaded again into an input register.

This restriction is relaxed and a deeply pipelined architecture is possible for the variant with interleaved P and Q, because the computation of Q delays the scheduling of the inputs for P for 64 clock cycles and vice versa. Therefore, it is easily possible to implement a pipeline with up to 64 pipeline stages.

This architecture cannot be implemented within the minimum requirements, because some temporary memory is always necessary. Hence, a BCM C implementing the P/Q-parallel version has $\text{SIZE}_{\text{mem}}(C) \geq 1648$, for the P/Q-interleaved version it is $\text{SIZE}_{\text{mem}}(C) \geq 1600$, and for the deeply pipelined version $\text{SIZE}_{\text{mem}}(C) \geq 2040$.

6.6.4 Evaluation Summary

The theoretical estimates are summed up in Tab. 6.7. The first column describes the data path width d, followed by two columns showing the minimum RAM and ROM requirements. The next attributes in the table describe the analyzed pipeline depth, the minimum number of clock cycle and the overhead in clock cycles. The second last column estimates the theoretical throughput for long messages at 100 MHz and the last column shows roughly the theoretical

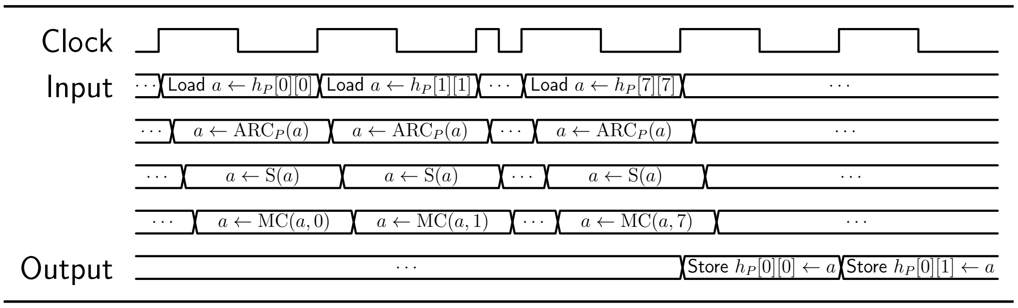

Clock
Input
Load $a \leftarrow h_P[0][0]$
Load $a \leftarrow h_P[1][1]$
Load $a \leftarrow h_P[7][7]$
$a \leftarrow \text{ARC}_P(a)$
$a \leftarrow \text{S}(a)$
$a \leftarrow \text{MC}(a,0)$
$a \leftarrow \text{MC}(a,1)$
$a \leftarrow \text{MC}(a,7)$
Output
Store $h_P[0][0] \leftarrow a$
Store $h_P[0][1] \leftarrow a$

Figure 6.15: Simplified timing for the byte-wise serialized implementation of the Grøstl-256 compression function.

throughput for short messages. This throughput is exactly one half of the long messages throughput because of the finalization, which includes a complete computation of the permutation P and thus is a significant burden for hashing short messages. Both estimates are only useful, if the clock frequency is fixed by external requirements, because similar to the other hash functions, the maximum clock frequency typically increases with the serialization.

Table 6.7: Summary of Grøstl-256 analysis

Data Path d [Bits]	Memory [Bits]	ROM [Bits]	Pipeline Depth	Clock Cycles	Overhead [Clock Cycles]	Long Message Throughput @ 100 MHz [MBits/s]	Short Message Throughput @ 100 MHz [MBits/s]
1024	1536	0	1	10	0	5120	2560
512	1536	0	1	20	0	2560	1780
512	2048	0	2	20	1	2560	1780
128	1536	0	1	80	0	640	320
64	1536	0	1	160	0	320	160
64	1984	0	8	160	7	320	160
16	1648	0	8	648	7	80	40
8	1600	0	8	1288	7	40	20
8	2040	0	64	1288	63	40	20

6.6.5 S-box Optimization

The optimization of the S-box uses the composite field approach, which is described in detail in Sec. A.5. It would be possible to use the implementation of Canright directly [Can05a]. However, because of the technological differences between ASICs and FPGAs and the fact that Canright only investigated ASIC implementations, it is possible that different representation are better for some FPGAs.

Based on the reasoning of Canright, the same 432 different representations were analyzed. A generic VHDL template for the formulas for squaring, multiplication and inversion for polynomial and normal bases was developed and then later instantiated for the selected polynomials and bases. For the concrete S-box not only the inverse in $\mathbb{F}_{256}$ has to be calculated, but additionally an affine transformation. This can be easily solved by multiplying the matrix from the affine transformation with the conversion matrix and by adding a few NOT gates for the additive part of the affine transformation. Exemplary implementation results for the Spartan-3 FPGAs are listed in Appendix A.7.

6.6.6 Implementation

Only one implementation of the latest specification of Grøstl-256 was realized [Jun12], because the midrange performance was targeted in this thesis for all algorithms except Keccak and PHOTON. In particular, the implementation follows the column-wise architecture, with a data path width $d = 64$ and eight pipeline steps [Jun12]. This implementation uses the FSL-based interface described in Sec. 6.3.1. Hence, a padding unit is included as specified in the Grøstl specification.

In this implementation, the computation of the permutations P and Q are serialized according to the previous analysis and hence, only one eighth of the original round function is implemented for the computation of the compression function (Fig. 6.16).

The implementation is based on several core ideas. Some of them have already been addressed in the previous analysis:

- Usage of distributed RAM
- An implicit ShiftBytes permutation
- Pipelining of the round permutation
- S-box based on composite fields
- Reusing the implementation of P for the output transformation

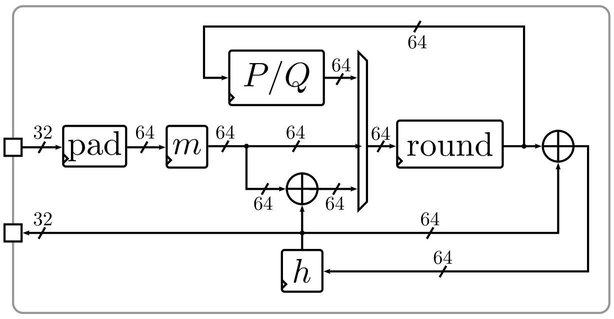


Figure 6.16: Grøstl-256 serialized implementation.

- Input streaming to increase the throughput

A 16×64 bit distributed RAM stores the state of both permutations P and Q. The RAM is organized in eight individual 16×8 bit RAMs, each one representation a row of h_P or h_Q. This organization facilitates an implicit implementation of the ShiftBytes transformation, by calculating appropriate read addresses.

Both states can be integrated into a single RAM, because the reads and writes alternate between P and Q. In each clock cycle a column from either h_P or from h_Q is read, i.e. one byte from each row. Furthermore, a 8×64 bit RAM is needed to store the chaining value h of the iteration mode, which is also organized in a row-wise fashion.

The next important concept is the pipelining of Grøstl's round transformation, to achieve a higher maximum clock frequency. The maximum depth of eight pipeline stages is implemented, according to the previous analysis. Since the pipeline registers are mostly integrated in the same slices as the logic, the area does not significantly increase for FPGA implementations.

Using the maximum pipeline depth has another benefit beside the improvement of the clock frequency. The eight pipeline stages store the complete state in the pipeline while processing. Hence. it is possible to overwrite RAM locations, without taking care of data that is still to be read, because the last input is read in the clock cycle before the first output is written to the memory. A shorter pipeline would require, that the architecture stores the output at the same memory locations where they were originally read. This would require additional bookkeeping in the form of a counter, which would be used to derive read and write addresses.

The S-box implementation is based on the composite field approach discussed in Sec. 6.6.5 and described in detail in Sec. A.5. It is used to calculate the output of the SubBytes permutation on-the-fly instead of using a lookup table. In addition to the area saved by this implementation style, it is possible to insert the pipeline registers in this S-box implementation more easily than in a design based on lookup tables (Fig. 6.17).

Another question is how to implement the output transformation. The easiest way is to reuse the implementation of the compression function. Since the compression function is defined as $\text{compress}(h, m) = P(h \oplus m) \oplus Q(m) \oplus h$,

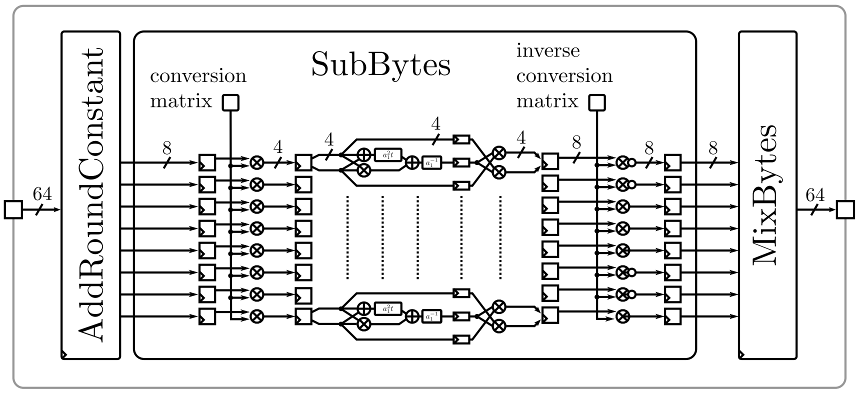


Figure 6.17: Grøstl-256 pipelined serialized round function.

it is possible to set $m = 0^{512}$ and to ignore the output of Q. The result is then $\Omega(h, m) = P(h) \oplus h$.

The implementation uses an streaming approach to load new message blocks to achieve a higher throughput, i.e. each new message block is loaded asynchronously using the FSL-based interface. This may also be done while the computation of the previous block is still ongoing. Hence, if the user of the hash function is able to transfer the data fast enough, then the absorption of new message blocks incurs no additional overhead.

The performance of this implementation is optimal in the sense of Lemma 5.3 and Corollary 5.4, because only 160 clock cycles are needed for a complete computation of the compression function. In particular, eight clock cycles are needed for each round of P and Q and an overall of ten rounds are calculated. Therefore, the processing of the compression function is finished after 160 clock cycles.

6.7 JH

The JH hash function was also one of the five finalist of the SHA-3 competition [Wu11]. The version for the final round was slightly modified. The main change increased the number of rounds from 35.5 to 42 rounds to increase the security margin and to slightly improve the efficiency of the last round for hardware implementations. In the original specification the last round was not completely computed, because the linear permutation, as well as the shuffling of bits does not add security in the last round.

The JH algorithm is based on a compression function which has a structure that is similar to the AES block cipher. However, the compression function itself is a new design independent of AES. The iteration mode of JH is a wide-pipe Merkle-Damgård construction (Sec. 3.5.1), but JH can be also seen as a variant of the Matyas-Meyer-Oseas mode [MMO85].

6.7.1 Definition

The original definition of JH is provided in its specification [Wu11]. The variant provided here is changed to achieve a common style for all analyzed algorithms. As the other SHA-3 candidates, the JH specification provides variations of the basic algorithm for the four different hash sizes $224, 256, 384$ and 512. The algorithm of the compression function is identical for all four variants, only the final output is truncated to the desired bit length. The domain separation between these algorithms is achieved by different initial values for this variant will be presented.

Input and output mapping The byte order of the input messages and the output digests are organized according to the NIST specification (Sec. 6.2). Hence, a message M with length $|M|$ is organized as follows:

$$M =_{\text{def}} \sum_{i=1}^{|M|/256} M_i \times 256^{|M|-i-1}.$$

The message digest is also interpreted using this byte order.

However, this general interpretation of a message is reorganized internally, because JH processes the bits in a different order after the padding. This so called grouping is defined as follows. A message $m \in \mathbb{Z}_2^{1024}$ is grouped as $m' \in \mathbb{Z}_2^{1024}$, such that for all $i \in \mathbb{Z}_2^{128}$:

$$\begin{aligned}
m'_{0+8i} &\leftarrow m_i & m'_{1+8i} &\leftarrow m_{i+2^d} \\
m'_{2+8i} &\leftarrow m_{i+2\times 2^d} & m'_{3+8i} &\leftarrow m_{i+3\times 2^d} \\
m'_{4+8i} &\leftarrow m_{i+2^{d-1}} & m'_{5+8i} &\leftarrow m_{i+2^{d-1}+2^d} \\
m'_{6+8i} &\leftarrow m_{i+2^{d-1}+2\times 2^d} & m'_{7+8i} &\leftarrow m_{i+2^{d-1}+3\times 2^d}
\end{aligned}$$

The degrouping is defined to be the inverse mapping. In a strict sense, the degrouping should be performed after each execution of the compression function, i.e. to compute the chaining value and the final hash value. However, the internal state and thus also the chaining value can be always stored in the grouped representation.

Padding The input to the padding function pad is a message $M \in \mathbb{Z}_2^{\geq 0}$. The output is a padded message p, such that $|p| \equiv 0 \bmod 512$, i.e. the size $|p|$ in bits is a multiple of 512 bits. For JH, the padding function is defined as follows, where $k = 384 - 1 + (-|M| \bmod 512)$.

$$\text{pad}(M) =_{\text{def}} M||1||0^k||(|M|_{128})$$

The padding is important for the security of JH, because it eliminates length extension attacks for some applications like MACs. Unfortunately, the padding has a significant negative impact on the performance of JH for short messages, because at least one additional message block is appended. Therefore, for short messages, the theoretical throughput is halved.

Initialization Values and Constants The initialization value h_0 for the state of JH-256 is calculated by setting the 16 most significants bit of h_{-1} to the hexadecimal value `0100` and then calculating $h_0 = \text{compress}(h_{-1})$.

Furthermore, JH uses round constants which are additional inputs to the round function and are used as the fifth input to the 5-to-4 S-box. The round constants are generated using a smaller version of the JH round function. In this smaller version the round constants are all set to 0.

The initial constant c_0 for round 0 and the following constants c_i for all other rounds $1 \leq i \leq 41$ are set as follows:

$$c_0 =_{\text{def}} \texttt{6a09e667f3bcc908b2fb1366ea957d3e}$$

```
3adec17512775099da2f590b0667322a
```

$$c_r =_{\text{def}} \quad \text{round}_6(c_{r-1}, 0)$$

Round Function The function round_d consists of three different permutations S_d, L_d, and P_d. S_d consists of 2^d parallel 5-to-4 bit S-boxes, L_d is an eight bit linear permutation and P_d is a permutation that shuffles 4 bit blocks of the state. The parameter d determines the size of the state and the round function and thus also of the individual permutations. The size of the state h_d is calculated as $|h_d| = 4 \times 2^d$. According to the JH specification, the parameter d is set to $d = 8$ for all instances that have been submitted to the SHA-3 competition. Setting $d = 8$ for JH-256 implies that a second round function with $d = 6$ is used to generate the round constants.

The S-box is defined according to Alg. 6.10 and 2^d of these 5-to-4 S-boxes form the S-box layer S_d of JH. Each of these identical S-boxes has 5 inputs and 4 outputs. However, only four of the five inputs are from the state h. The fifth input is part of the round constant. Therefore, the S-box can also be viewed as two independent S-boxes and that the fifth bit selects between both S-boxes. Then both S-boxes are permutations.

The linear permutation L_d implements a maximum distance separable (MDS)

Algorithm 6.10 JH S-box [Wu11]

```
Require: $h = \mathbb{Z}_2^4$, and $c_r \in \mathbb{Z}_2$, $0 \leq r \leq 13$
Ensure: $h \leftarrow S(h, c)$
  if $c = 0$ then
    $h_3 \leftarrow 1 \oplus h_3 \oplus h_2 \oplus h_2h_1 \oplus h_3h_2h_1 \oplus h_3h_2h_0 \oplus h_3h_1h_0 \oplus h_2h_1h_0$
    $h_2 \leftarrow h_3h_2 \oplus h_2h_1 \oplus h_3h_0 \oplus h_2h_0 \oplus h_1h_0 \oplus h_3h_2h_1 \oplus h_2h_1h_0$
    $h_1 \leftarrow h_2 \oplus h_3h_2 \oplus h_1 \oplus h_2h_0 \oplus h_2h_1h_0$
    $h_0 \leftarrow 1 \oplus h_3 \oplus h_2 \oplus h_0 \oplus h_3h_1 \oplus h_2h_0 \oplus h_3h_2h_1 \oplus h_2h_1h_0$
  else
    $h_3 \leftarrow 1 \oplus h_3 \oplus h_2 \oplus h_3h_1 \oplus h_2h_1 \oplus h_3h_2h_0 \oplus h_3h_1h_0 \oplus h_2h_1h_0$
    $h_2 \leftarrow 1 \oplus h_3 \oplus h_1 \oplus h_3h_0 \oplus h_2h_0 \oplus h_1h_0 \oplus h_3h_2h_1 \oplus h_2h_1h_0$
    $h_1 \leftarrow h_3 \oplus h_2 \oplus h_1 \oplus h_0 \oplus h_3h_2 \oplus h_3h_1 \oplus h_3h_2h_1 \oplus h_3h_2h_0$
    $h_0 \leftarrow h_3 \oplus h_0 \oplus h_3h_1 \oplus h_2h_0 \oplus h_3h_2h_1 \oplus h_3h_2h_0 \oplus h_2h_1h_0$
  end if
  return $h$
```

Algorithm 6.11 JH linear transformation [Wu11]

```
Require: $h \in \mathbb{Z}_2^{4\times 2^d}$
Ensure: $h \leftarrow L_d(h)$
  for $i = 0$ to $2^{d-1}$ do
    $a_0 \leftarrow h[8i]$, $a_1 \leftarrow h[8i+1]$, $a_2 \leftarrow h[8i+2]$, $a_3 \leftarrow h[8i+3]$
    $b_0 \leftarrow h[8i+4]$, $b_1 \leftarrow h[8i+5]$ $b_2 \leftarrow h[8i+6]$, $b_3 \leftarrow h[8i+7]$

    $d_0 \leftarrow b_0 \oplus a_1$, $d_1 \leftarrow b_1 \oplus a_2$, $d_2 \leftarrow b_2 \oplus a_3 \oplus a_0$, $d_3 \leftarrow b_3 \oplus a_0$
    $c_0 \leftarrow a_0 \oplus d_1$, $c_1 \leftarrow a_1 \oplus d_2$, $c_2 \leftarrow a_2 \oplus d_3 \oplus d_0$, $c_3 \leftarrow a_3 \oplus d_0$

    $h[8i] \leftarrow c_0$, $h[8i+1] \leftarrow c_1$, $h[8i+2] \leftarrow c_2$, $h[8i+3] \leftarrow c_3$
    $h[8i+4] \leftarrow d_0$, $h[8i+5] \leftarrow d_1$ $h[8i+6] \leftarrow d_2$, $h[8i+7] \leftarrow d_3$
  end for
  return $h$
```

Algorithm 6.12 JH permutation layer [Wu11]

```
Require: $h_0, \dots, h_{2^d-1} \in \mathbb{Z}_2^4$, where $h = h_0 || \cdots || h_{2^d-1}$
Ensure: $h \leftarrow P_d(h)$
  for $i = 0$ to $2^{(d-2)} - 1$ do
    $h_{4i+0} \leftarrow h_{4i+0}$, $h_{4i+1} \leftarrow h_{4i+1}$, $h_{4i+2} \leftarrow h_{4i+3}$, $h_{4i+3} \leftarrow h_{4i+2}$
  end for
  for $i = 0$ to $2^{(d-1)} - 1$ do
    $h_i \leftarrow h_{2i}$, $h_{i+2^{d-1}} \leftarrow h_{2i+1}$
  end for
  return $h$
```

code. The specified algorithm works with the state $h \in \mathbb{Z}_2^{4\times 2^d}$, which is split into 2^{d-1} pairs (a, b), where $a, b \in \mathbb{Z}_2^4$. Each element can be interpreted as a polynomial of degree 3, e.g. $a = a_0x^3 + a_1x^2 + a_2x + a_3$, where $a_0, \dots, a_3$ are bits. Then the algorithm to compute L_d is defined by Alg. 6.11.

The permutation P_d is defined according to Alg. 6.12. Its original definition is split into two parts π_d and P'_d. Both steps are fused in Alg. 6.12.

Compression Function The function compress used by the JH variants submitted to the SHA-3 competition sets the parameter to $d = 8$ and thus, the round function round_8 is used. In total round_8 is computed 14 times. The compression function additionally contains the message absorption, which happens in two steps. First, a message block m of size 2^{1024} bits is XORed to the first half of the JH state before the 14 round computations and second the message is also XORed to the second half. The complete compression function is specified in Alg. 6.13.

Algorithm 6.13 JH compression function [Wu11]

```
Require: $m \in \mathbb{Z}_2^{512}$, $h'_0, h'_1 \in \mathbb{Z}_2^{512}$, such that $h' = h'_0||h'_1$
Ensure: $h \leftarrow f(m, h')$
  $h \leftarrow (h'_0 \oplus m)||h'_1$
  for $i = 0$ to 41 do
    $h \leftarrow \text{round}_8(h)$
  end for
  $h \leftarrow h_0||(h_1 \oplus m)$
  return $h$
```

Iteration Mode The iteration mode is a standard wide-pipe Merkle-Damgård design (Sec. 3.5.1). Thus, the output of the compression function is truncated after the absorption of the last block of a padded message. However, due to the special message absorption in the compression function and the padding rule which appends at least one additional message block, JH provides additional security properties beyond the basic collision resistance [Wu11].

6.7.2 Systematic Evaluation Overview

State and Memory Organization The memory storing the 1024 bit state can be organized in eight different RAM patterns in the general form of $2^k \times ({}^{1024}/_{2^k})$ bits, where $0 \leq k \leq 7$ However, because of the permutation layer of JH, the RAM has to be split as follows:

- For $k = 0$, the implementation is a 1024 bit wide register, which can be accessed in parallel.
- For $1 \leq k \leq 7$, the RAM has to be split into two parts each with half of the width, i.e. two $2^k \times ({}^{512}/_{2^k})$ RAMs. This splitting to achieve a stall free architecture for the permutation layer P_8. The state is distributed in an alternating pattern over the RAMs, as shown in Fig. 6.18 for $k = 2$.

In addition to the state, it is necessary to add a smaller RAM instance (256 bit) to the architecture, which stores the round constants. This RAM is just a smaller version of the state memory and thus, the same properties apply with the only difference, that k' is limited to $0 \leq k' \leq 5$. In particular $k' = 0$ for $k \in \{0, 1\}$ and $k' = k - 2$ for $k \geq 2$.

Furthermore a 512 bit memory is needed to store the message, based on the structure of the JH compression function. Therefore, the memory size of a BCM C implementing JH is estimated to be at least $\text{SIZE}_{\text{mem}}(C) \geq 1792$.

An implementation has also to store the same amount of bits for the initialization values of the state and of the round constants in a ROM. The organization is similar but slightly easier then the RAM organization.

State Read and Write Schedule The read and write schedule is dependent on the permutation layer P_d of JH. An efficient way to successfully schedule the read and write operations for $1 \leq k \leq 6$ follows the simplified method depicted in Fig. 6.18. Two RAMs store blocks of $^{512}/_{2^k}$ bits in an alternating fashion.

However, because of the permutation layer, this pattern is reversed for the second half, i.e. the RAMs are used in the inverted order. Otherwise, it would be impossible to store the second part of the output and still get an optimal architecture, because two blocks with different addresses would have to be written to the same RAM. In the example in Fig. 6.18 this would be equal to

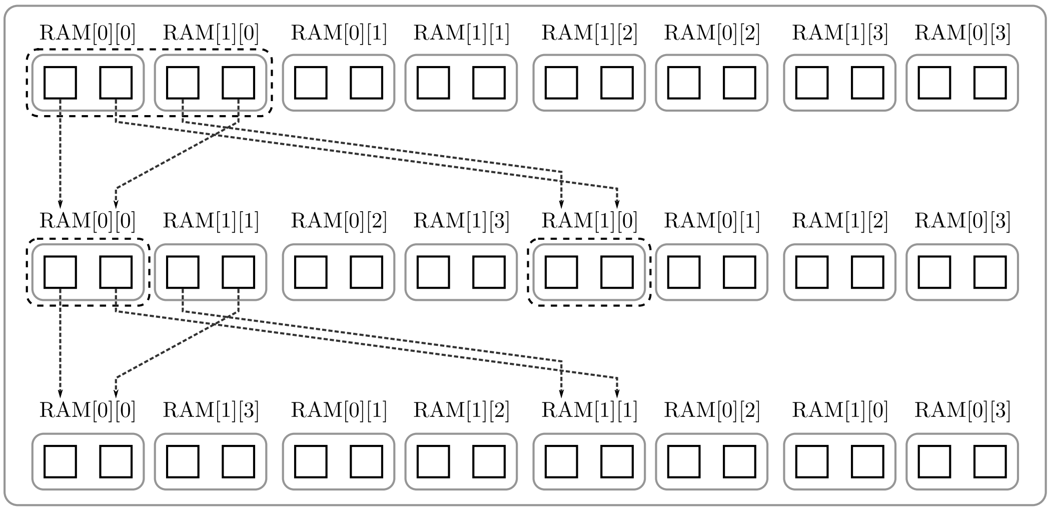


Figure 6.18: Example organization of the state and read and write operations for the JH state and $k = 2$.

try to store new values to RAM[0][0] and RAM[0][2] in one clock cycle, which is impossible, if the memory is not implemented as a register or as a RAM with two write ports. The main drawback of this scheduling is, that a number of multiplexers is necessary to switch the outputs of the RAMs for the second half of the round function.

Furthermore, the read and write addresses change for each round as also depicted in Fig. 6.18, such that no data is overwritten, before it was read. This means, that the read addresses are used as write addresses one clock cycle later and therefore, the read addresses for the next round have to be adjusted. In the example of Fig. 6.18, the first values to be read are RAM[0][0] and RAM[1][0], whereas in the second round the values would be RAM[0][0] and RAM[1][1].

An easier architecture may be implemented for the smallest variant with $k = 7$, because the state memory can be split into two RAMs with 4 bit width, were the first RAM stores the 4 bit parts of the state with even addresses, and the other with odd addresses. Since only 4 bits are stored in each memory cell, the scheduling of read and write operations becomes easier.

Clock Cycles Estimation The number of clock cycles is determined by the data path width. However, since it is cheaper to compute the round constants on the fly compared to storing all round constants, it is convenient to view both RAMs as the state of JH for the clock cycle estimation. Hence, the data path width also includes the generation of the constants.

If this assumption is made for the analysis, the estimates are optimal according to Lemma 5.3 and Corollary 5.4. Therefore, the compression function of JH may be computed in $\mathrm{cyc}_{\mathrm{compress}}(d)$ clock cycles, which is defined as follows. The data path width d is the only variable, the state size is assumed to be 1280 bit, including the RAM for the round constants and the number of rounds is assumed to be 42:

$$\begin{aligned}\mathrm{cyc}_{\mathrm{compress}}(d) &\geq \frac{1280 \times 42}{d},\\ \mathrm{cyc}_{\mathrm{compress}}(d) &< \frac{1280 \times (42+1)}{d}.\end{aligned}$$

Round Function Architectures The round function architecture consists of only two S-boxes and the linear permutation layer, which is replicated $d/8$

times. The simple permutation layer is conveniently handled by the state RAM read and write addresses.

The input scheduling for the round Function has two different possibilities. The first is a parallel computation of both the round function for the state and the computation. The other one is an interleaved processing of state and constants. For the interleaved variant, in the first four clock cycles the state is processed and in the fifth the round constants are updated.

6.7.3 Detailed Analysis

Architecture 1 The first architecture processes JH's internal state and the round constants in parallel. There are in total six different options, processing $4 \times 2^{8-k} + 2^{8-k}$ bits per clock cycle, with $0 \leq k \leq 5$. Smaller data path widths cannot be easily used for a parallel processing approach, because the input for processing the round constants would be less than eight bits. Hence, intra-round dependencies of the linear permutation L_8 would imply a more complicated implementation with additional registers.

- Using $k = 0$ leads to a fully parallel implementation of the JH round function.
- All other versions use a variant of the read and write scheduling already discussed in Sec. 6.7.2. In relation to the parallel version with $k = 0$, the number of clock cycles increases by a factor of 2^k.

It is also possible to add several pipelining steps for $k \geq 2$. For example, Fig. 6.19 shows an abstraction of the possible pipeline stages for $k = 2$. The corresponding loads and stores for the constants are omitted in this figure. The permutation layer of JH shuffles the bits in such a way, that for these variants, exactly 2^{k-1} pipeline stages are possible. However, since the logic depth of the round function is already shallow, a deep pipelining does not improve the clock frequency significantly.

The basic memory estimate for a BCM C implementation on of the discussed non-pipelined holds, i.e. $\text{SIZE}_{\text{mem}}(C) \geq 1792$. The pipelining adds 2^k bits for every additional pipeline stage.

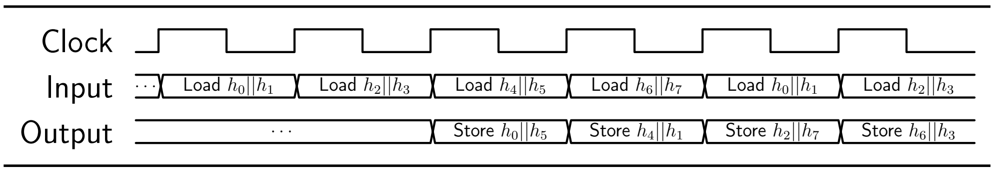


Figure 6.19: Scheduling of the I/O for a pipelined implementation of JH with $k = 2$.

Architecture 2 The second architecture implements an interleaved processing of the state and the round constants. Assuming the same state organization as for the first architecture in principle, it is possible to compute $4 \times 2^{8-k}$ bits per clock cycle for $2 \leq k \leq 7$. The same architecture does not work efficiently for $k = 0$ and $k = 1$, because the width of the round constants would be smaller than the data path width. Hence, the round function would be under-utilized when processing the round constants.

The interleaved scheduling of the round constants and the state works as shown in Fig. 6.20 for $k = 2$. Since the round constants are used in the processing of the state, it is necessary to first load d bits from the constants and to store them into registers. The first computation of the round function computes d bits of the round constants for the next rounds. Afterwards, these d bits are stored to the constants RAM and the four corresponding d bits parts of the state are processed next, before the next part of the constants RAM is loaded and processed again. The original round constants have to be saved in a register. Therefore, the memory consumption of a BCM C implementing one of the variants is estimated to be at least $\text{SIZE}_{\text{mem}}(C) \geq 1792 + d$ memory bits.

As can be seen from Fig. 6.20, the scheduling of the I/O of the state RAM is almost identical to the first architecture with the same k, with the exception of loading and storing the round constants c. This additional operation facilitates

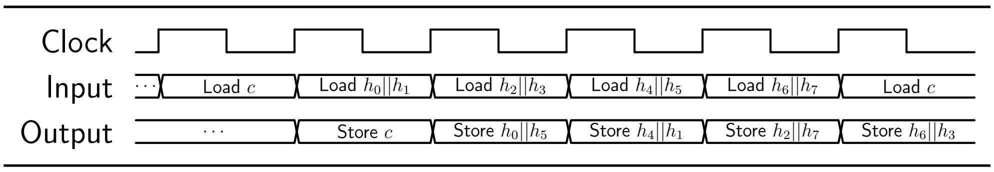


Figure 6.20: Scheduling of the I/O for an interleaved non-pipelined implementation of JH with $k = 2$.

a pipeline depth that is slightly deeper than for the first architecture. For $k = 2$ this leads to three pipeline stages instead of only two. For $k \geq 3$, the number of pipeline stages can be generically expressed with the formula $2^{k-1} + 2^{k-3}$. Each pipeline stage adds d bits to the memory estimate above.

6.7.4 Evaluation Summary

The theoretical results of the analysis are shown in Tab. 6.8. The table shows the data path width d, the number of RAM and ROM bits necessary to implement JH. Furthermore, the information regarding the throughput performance is comprised of the number of maximum pipeline steps, the number of clock cycles for one round, the offset for an implementation, the throughput for long and short messages.

Table 6.8: Summary of JH-256 analysis

Data Path d [Bits]	Memory [Bits]	ROM [Bits]	Analyzed Pipeline Depth	Clock Cycles	Overhead	Long Message Throughput @ 100 MHz [MBits/s]	Short Message Throughput @ 100 MHz [MBits/s]
1280	1792	1280	1	42	0	1219	609.5
640	1792	1280	1	84	0	609.5	304.7
320	1792	1280	1	168	0	304.7	152.3
320	2112	1280	2	168	1	152.3	76.19
256	2048	1280	1	210	0	243.8	121.9
256	2560	1280	3	210	2	243.8	121.9
160	1792	1280	1	336	0	152.3	76.19
160	2272	1280	4	336	3	152.3	76.19
128	1920	1280	1	420	0	121.9	60.95
128	2432	1280	5	420	4	121.9	60.95
80	1792	1280	1	672	0	76.19	38.09
80	2352	1280	8	672	7	76.19	38.09
64	1856	1280	1	840	0	60.95	30.47
64	2432	1280	10	840	9	60.95	30.47
40	1792	1280	1	1344	0	38.09	19.04
40	2392	1280	16	1344	15	38.09	19.04
32	1824	1280	1	1680	0	30.47	15.23
32	2432	1280	20	1680	19	30.47	15.23
16	1808	1280	1	3360	0	15.23	7.619
16	2432	1280	40	3360	39	15.23	7.619
8	1800	1280	1	6720	0	7.619	3.809
8	2432	1280	80	6720	79	7.619	3.809

One variant which was not discussed is the possibility of unrolling the round function. This could prove to be beneficial, because the round function itself is small compared to other algorithms and the depth of it is rather shallow [GHR+12a, GHR+12b].

6.7.5 Manual `LUT6_2` Instantiation

All architectures can be improved by manually instantiating `LUT6_2` macros to efficiently implement the inner transformation for Xilinx devices that support these primitives. Overall, this leads to eight `LUT6_2` instances eight input bits. Two S-boxes are needed for eight bits. Each S-box has five inputs and four outputs. A `LUT6_2` can be configured using five inputs and two outputs. Hence, four `LUT6_2`s are needed to implement the two S-boxes.

For the linear transformation layer also four `LUT6_2` are needed. This transformation layer can be reorganized as follows:

$$
\begin{aligned}
&d_0 \leftarrow b_0 \oplus a_1 \\
&c_3 \leftarrow a_3 \oplus b_0 \oplus a_1 \\
&d_1 \leftarrow b_1 \oplus a_2 \\
&c_0 \leftarrow a_0 \oplus b_1 \oplus a_2 \\
&d_2 \leftarrow b_2 \oplus a_3 \oplus a_0 \\
&c_1 \leftarrow a_1 \oplus b_2 \oplus a_3 \oplus a_0 \\
&d_3 \leftarrow b_3 \oplus a_0 \\
&c_2 \leftarrow a_2 \oplus b_3 \oplus a_0 \oplus b_0 \oplus a_1
\end{aligned}
$$

Table 6.9: Instantiation of `LUT6_2`s for JH

`LUT6_2` function name	`INIT` values
S-box 1	`11f931d973259ec8`
S-box 2	`493eb8b4f18ac2b9`
Linear transformation 1	`000000960000003C`
Linear transformation 2	`966969963C3C3C3C`
Linear transformation 3	`000069960000C33C`
Linear transformation 4	`000000960000003C`

Now, it is easy to see, that by duplicating the logic for $d_0, \ldots, d_3$ the operations of the linear transformation can be also fitted into four `LUT6_2` instances. The concrete values for the generic parameter `INIT` of the `LUT6_2` macros are provided in Tab. 6.9.

6.7.6 Implementation

Two implementations were developed [Jun12]. The first implementation is a very compact implementation, which explores the easy serialization possibility of JH. The second version is aiming at the midrange target just like most of the other implementations of the SHA-3 finalists. Both variants use the FSL-based I/O interface.

The first design uses the smallest analyzed data path width with only eight bits (Fig. 6.21). The logic in JH's round function is very small and shallow, thus pipelining does not increase the clock frequency very much and was not further pursued. Unfortunately, the high number of rounds of JH is the bottleneck of this implementation, which reaches a very low throughput.

The design uses distributed RAM for the input, the internal state and the round constants. An additional RAM stores the message after the message injection for the second XOR after the completion of the round function. Therefore, a new message block may be loaded, while the old message block is still processed.

Following the earlier theoretical analysis, the round constants are computed using the same implementation of the round function as the normal processing

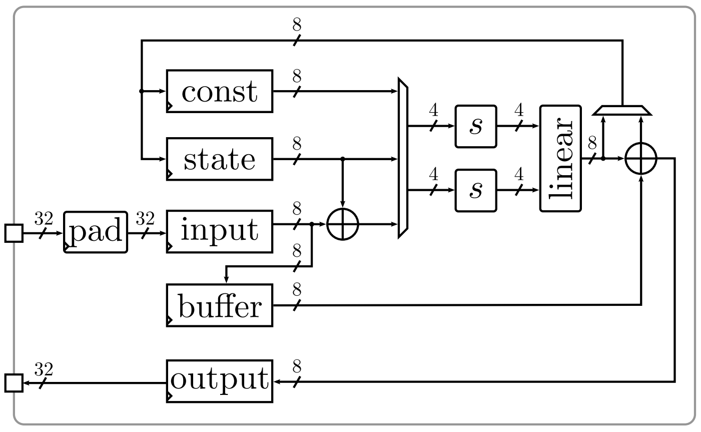


Figure 6.21: JH-256 serialized implementation with $d = 8$.

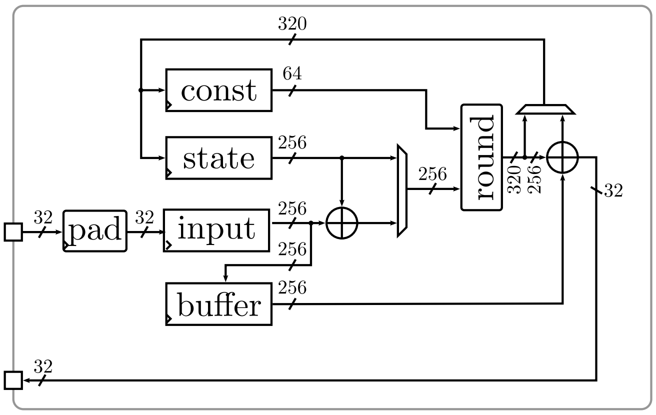


Figure 6.22: JH-256 serialized implementation with $d = 320$.

of the state. This shared core of the JH architecture consists of two S-boxes and one linear transformation. The JH permutation layer is easily realized by writing to the state RAM according to the specification of the permutation as discussed earlier in Sec. 6.7.2.

Beside the implementation of the round function and the compression function, the grouping and de-grouping is problematic. The bits of the input and output have to be reordered and the two representations cannot be implemented efficiently without many additional clock cycles and a distributed RAM organization, which is less than optimal. Both functions are covered by the input and the output RAMs, which therefore are larger than necessary for the required capacity.

The first design needs at least 6720 clock cycles to compute the compression function. The 128 bytes of the state and the 32 bytes constants lead to 160 clock cycles per round. Hence, the computation of all 42 rounds needs 6720 clock cycles.

The second design uses the option from the theoretical analysis with a data path of 320 bit (Fig. 6.22). Hence, the number of clock cycles per round is reduced by a factor of 40. Yet, the design stays reasonably small. Compared to the previous version, the following changes were applied:

- The round function has to implement more S-boxes and linear transformations.
- The grouping of the input and the buffering stays practically the same. However, they have to be adapted to use the wider data path.

- The degrouping of the final hash value is no longer required, because in each clock cycle of the last round the 32 bit which are part of the hash digest are included.
- The S-Boxes and linear transformations are implemented by manually instantiating `LUT6_2` instances to achieve a reasonable area consumption.

6.8 Keccak

The Keccak hash function is the winner of the SHA-3 competition [BDPA11b, BDPA11a, CPB+12]. For the last round it was slightly modified, i.e. the padding function was simplified and a so called diversifier parameter has been removed [BDPA11b, BDPA11a]. Keccak is based on the sponge construction which is a new way to construct hash functions [BDPA11c] (Sec. 3.5.3). The permutation used by Keccak is also a new design.

6.8.1 Definition

The following definition only describes the permutation used by Keccak, and references the earlier description of the sponge construction in Sec. 3.5.3. Since Keccak is the only SHA-3 finalist under investigation that is later implemented and evaluated in non-standard modes for lightweight and midrange applications, the generic specification is presented here. The original specification can be found in [BDPA11b] and the instances which are subject to standardization are specified in [BDPA11a].

State Size and Representation The state of Keccak has a size of $b = 25 \times 2^l$. It can be interpreted in a three-dimensional representation, i.e. with coordinates (x, y, z), where $x, y \in \mathbb{Z}_5$ and $z \in \mathbb{Z}_{2^l}$ (Fig. 6.24).

Input and Output Mapping The interpretation of input and output bit strings used by Keccak differs from the ordering defined in Def. 6.1. In particular the bits of a byte are interpreted in the opposite direction. For hardware implementations this reordering is practically free of any cost, however if the padding function is also implemented in hardware, this at least results in one additional bit shift, if the last byte contains less than eight bits.

The following definition maps a message in the generic format to the internal representation of Keccak. Let $M \in \mathbb{Z}_2^{\geq 0}$ be a message in the representation according to Def. 6.1, then the message may be converted to the message M' as follows:

$$M' \leftarrow \sum_{i=0}^{\left\lfloor \frac{|M|}{8} \right\rfloor - 1} \sum_{j=0}^{7} M[8i + (7 - j)] \times 2^{|M| - (8i+j) - 1}$$

$$+ \sum_{k=0}^{|M| \bmod 8} M[(|M| \bmod 8) - k] \times 2^k$$

Furthermore, a message block $m \in \mathbb{Z}_2^b$ is mapped to the internal three-dimensional representation a as follows:

$$a[x][y][z] =_{\text{def}} m[2^l(x + 5y) + z]$$

The mapping of the state to the output is performed by the inverse mapping.

Padding The multi-rate padding used by KECCAK appends a string 10*1, such that the length of the padded message is a multiple of r. Formally, for a message $M \in \mathbb{Z}_2^{\geq 0}$ and $k = (-|M| - 2) \bmod r$, the padding function is defined as:

$$\text{pad}_{256}(M) =_{\text{def}} M||1||0^k||1$$

Initialization Values and Constants The initial state s of KECCAK is initialized to 0, i.e. $s \leftarrow 0^b$, where b is the state size.

The round constants are defined as follows:

$$\text{RC}[i_r][0][0][2^j - 1] =_{\text{def}} \text{rc}[j + 7i_r]$$

where the individual bits of the round constant are calculated with a linear feedback shift register (LFSR):

$$\text{rc}[t] = (x^t \bmod x^8 + x^6 + x^5 + x^4 + 1) \bmod x$$

Hence, the constants (64 bits) for each round are defined to be as follows in the representation defined above. To simplify the notation the x and y coordinates are omitted:

$\text{RC}[0] =_{\text{def}}$ `8000000000000000`	$\text{RC}[1] =_{\text{def}}$ `4101000000000000`
$\text{RC}[2] =_{\text{def}}$ `5101000000000001`	$\text{RC}[3] =_{\text{def}}$ `0001000100000001`
$\text{RC}[4] =_{\text{def}}$ `D101000000000000`	$\text{RC}[5] =_{\text{def}}$ `8000000100000000`
$\text{RC}[6] =_{\text{def}}$ `8101000100000001`	$\text{RC}[7] =_{\text{def}}$ `9001000000000001`
$\text{RC}[8] =_{\text{def}}$ `4100000000000000`	$\text{RC}[9] =_{\text{def}}$ `1100000000000000`
$\text{RC}[10] =_{\text{def}}$ `9001000100000000`	$\text{RC}[11] =_{\text{def}}$ `4000000100000000`

RC[12] $=_{\text{def}}$ `D101000100000000`	RC[13] $=_{\text{def}}$ `D100000000000001`
RC[14] $=_{\text{def}}$ `9101000000000001`	RC[15] $=_{\text{def}}$ `C001000000000001`
RC[16] $=_{\text{def}}$ `4001000000000001`	RC[17] $=_{\text{def}}$ `0100000000000001`
RC[18] $=_{\text{def}}$ `5001000000000000`	RC[19] $=_{\text{def}}$ `5000000100000001`
RC[20] $=_{\text{def}}$ `8101000100000001`	RC[21] $=_{\text{def}}$ `0101000000000001`
RC[22] $=_{\text{def}}$ `8000000100000000`	RC[23] $=_{\text{def}}$ `1001000100000001`

For $l < 6$, the round constants are truncated to 2^l bits, i.e. only the 2^l least significant bits (left most bits) are used.

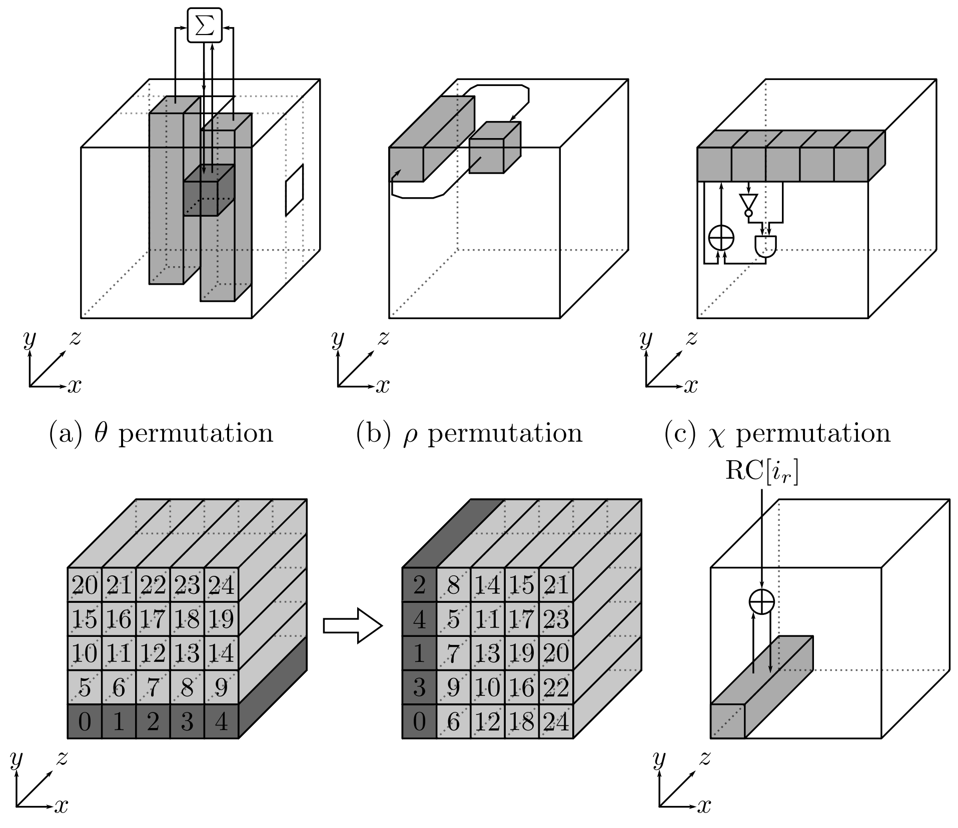
Algorithm 6.14 Keccak round function [BDPA11b].

```
Require: $a \in \mathbb{Z}_2^{5\times5\times2^l}, 0 \le i_r \le 12 + 2l$
Ensure: $a \leftarrow \text{round}(a, i_r)$
  for $x = 0$ to 4 and $y = 0$ to 4 and $z = 0$ to $2^l - 1$ do
    $\theta : a[x][y][z] \leftarrow a[x][y][z] + \sum_{y'=0}^{4} a[x-1][y'][z] + \sum_{y'=0}^{4} a[x+1][y'][z-1]$
  end for
  for $x = 0$ to 4 and $y = 0$ to 4 and $z = 0$ to $2^l - 1$ do
    $\rho : a[x][y][z] \leftarrow a[x][y][z - (t+1)(t+2)/2]$, with $t$ satisfying
        $0 \le t < 24$ and $\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$ in $\mathbb{F}_5^{2\times2}$,
        or $t = -1$ if $x = y = 0$
  end for
  for $x = 0$ to 4 and $y = 0$ to 4 do
    $\pi : a[x][y] \leftarrow a[x'][y']$, with $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}$
  end for
  for $x = 0$ to 4 do
    $\chi : a[x] \leftarrow a[x] + (a[x+1] + 1)a[x+2]$
  end for
  $\iota : a[0][0] \leftarrow a[0][0] + \text{RC}[i_r]$
  return $a$
```

Round Function and KECCAK Permutation The round function of KECCAK consists of five different functions, θ, ρ, π, χ, and ι, which are computed consecutively in each round according to Alg. 6.14.

The five permutations have significantly different properties. The first transformation θ is a linear function (Fig. 6.23a). Each output bit depends on 11 input bits. The next two functions ρ (Fig. 6.23b) and π (Fig. 6.23d) only shuffle the bits, therefore each output bit depends only on exactly one input bit. The χ permutation is the only non-linear function (Fig. 6.23c). To generate one output bit, at least three input bits are needed. The last function ι only adds the round constants to the lane with $x = y = 0$ and because the round constants are realized with a simple $\oplus$-operation, each output bit has only a dependency on one round constant bit and a state bit (Fig. 6.23e).

This round function is computed for a number of rounds n_r, defined in terms of the parameter l, i.e. n_r depends on the state size, in particular $n_r = 12 + 2l$.



(a) θ permutation (b) ρ permutation (c) χ permutation

(d) π permutation (e) ι permutation

Figure 6.23: The different KECCAK sub-permutations.

The iterative computation of all n_r rounds forms the permutation that is used to instantiate the sponge function KECCAK-$f[b]$.

Iteration Mode The iteration mode used by KECCAK is the sponge construction which was already discussed in detail in Sec. 3.5.3. In particular, Alg. 3.3 applies. In general, a sponge function is split into the absorption and the squeezing phases and therefore, allows very interesting options for both high throughput and lightweight design goals. The absorption phase processes the input message, whereas the squeezing phase produces the output of the function of possibly infinite length.

As mentioned previously in Sec. 3.5.3, a sponge function can be parameterized by choosing b, c and r, with $b = c + r$. The parameter b describes the size of the internal state and r is the so-called rate, i.e. the size of a single message block. Then c determines the security of the instance of the sponge function. In case of KECCAK, the sponge function KECCAK-$f[b]$ is parameterized with the values in Tab. 6.10 for the implementations. However, the analysis is performed in a generic way, independent of the chosen parameters.

Table 6.10: Sponge parameters for KECCAK-$f[b]$

Message digest n [bit]	State Size b [bit]	Capacity c [bit]	Rate r [bit]	Rounds
128	200	128	72	18
160	200	160	40	18
128	400	128	272	20
128	400	256	144	20
160	400	160	240	20
160	400	320	80	20
224	400	224	176	20
256	400	256	144	20
256	800	256	544	22
256	800	512	288	22
256	1600	512	1088	24

6.8.2 Systematic Evaluation Overview

State and Memory Organization According to the systematic methodology from Ch. 5, the state organization is investigated first. The state memory and the computation may be split according to the different parts of the state shown in Fig. 6.24, i.e. a bit-, column-, row-, lane-, plane-, sheet-, or slice-wise processing is possible in addition to a parallel design. Beside the state RAM, no additional memory is needed in general to implement Keccak. Therefore the memory size of a BCM C implementing Keccak is estimated to be at least $\mathrm{SIZE}_{\mathrm{mem}}(C) \geq 25 \times 2^l = b$.

For each case, the state memory has to be organized in a different fashion:

- A *parallel* implementation uses a wide register with b bits and is the easiest design for the state memory. The data path width is thus $d = b$.
- A *bit-wise* design may be implemented using a $b \times 1$ bit RAM organization. The RAM can be used directly in this organization and no further splitting or other reorganization has to be made, provided that appropriate read and write addresses are calculated. The data path has the minimum possible width $d = 1$.
- The *lane-wise* architecture can be realized with a 25×2^l bit RAM, and hence $d = 2^l$. Similar to the bit-wise organization, only the correct read and write addresses have to be computed.
- The *row-* and *column-wise* organizations need five $5(2^l) \times 1$ RAMs, i.e. for the row-wise design, each sheet is stored in a single RAM and for the column-wise design, each plane. For both designs the data path width is $d = 5$.
- The *sheet-* and the *plane-wise* organizations have to use five 5×2^l memories, because of the π permutation and hence $d = 5(2^l)$.

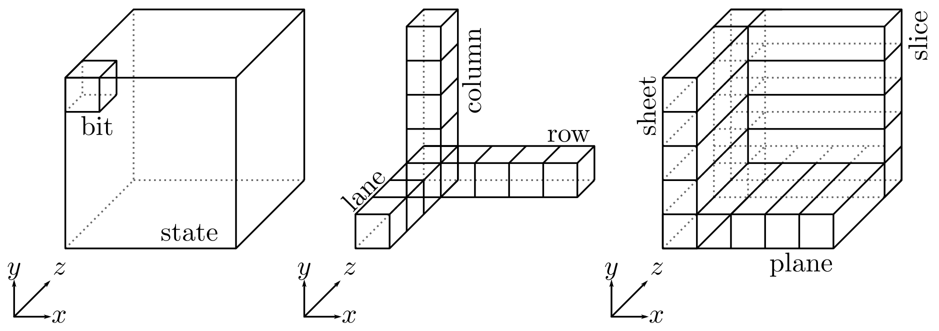


Figure 6.24: Parts of Keccak's state.

- The *slice-wise* design has several variants, starting with the computation of only one slice per clock cycle up to 2^l slices. Hence, depending on the number of slices in the state, the organization changes slightly. Let p be the number of slices processed in parallel, then the RAM needs to be organized in 25 smaller RAMs with ${}^{2^l}/_{p} \times p$ bits. However, this still does not work, because of the ρ permutation. Therefore, each RAM has again to be split to facilitate the lane rotations. The width of these RAMs is dependent on p and the rotation constants of ρ (Tab. 6.11). For example, for $x = y = 2$ and $p = 8$, the RAM is split into a $2^l \times 3$ bit and a $2^l \times 5$ bit RAM. For the case $p = 1$, this generic split leads to the a special variant, where one of the two RAMs disappears with zero bits to store.

Table 6.11: Rotation offsets for the ρ permutation of Keccak-[1600].

y \ x	[0]	[1]	[2]	[3]	[4]
[0]	0	1	62	28	27
[1]	36	44	6	55	20
[2]	3	10	43	25	39
[3]	41	45	15	21	8
[4]	18	2	61	56	14

In addition to the state RAM, the round constants are usually stored in an extra ROM with $(1 + l) \times n_r$ bits. It is in general possible to compute these constants on the fly using an LFSR. However, since only seven bits per 64 bit lane have to be generated, the book-keeping for an on demand computation of the LFSR output bits is in most cases not worth the effort.

State Read and Write Schedule The scheduling of read and write addresses differs considerably between the variants. For the parallel version it is trivial. For all other architectures it is assumed that the round function is rescheduled. Otherwise, it is either necessary to add a lot more memory to the round function implementation or to store intermediate results in the state RAM and hence, to use many more clock cycles to compute the round function. Both consequences are due to the data dependencies of the π and the θ permutations.

The original round function r_i is defined for all rounds $0 \leq i \leq 23$ as follows:

$$r_i =_{\text{def}} \iota \circ \chi \circ \pi \circ \rho \circ \theta.$$

The rescheduled version can be expressed as follows, where i is the current round number:

$$r_i =_{\text{def}} \begin{cases} \pi \circ \rho \circ \theta & \text{if } i = 0 \\ \pi \circ \rho \circ \theta \circ \iota \circ \chi & \text{if } 1 \leq i \leq 23 \\ \iota \circ \chi & \text{if } i = 24 \end{cases}$$

The important property of the rescheduled version is, that the two permutations π and ρ may now directly be used to derive the read and write addresses for the state RAM. For comparison, the original version shuffles the bits internally in the middle of the round function and thus, the complete state has to be reordered before computing χ.

The read and write addresses can be calculated for each variant beside the parallel version as follows:

- For the *bit-wise* architecture, it is possible to use the x and y components according to the π permutation as write addresses to implement π. The z component of the bit coordinate in the three-dimensional state has to be solved differently, because otherwise parts of the state would be overwritten before they have been processed. Therefore, the old z component is used, because then it is guaranteed that the old bit has already been processed. Then when reading the inputs for the next round, the read address is manipulated to correct the offset implied by ρ.

- The *lane-wise* architecture can be seen as a z-parallel version of the bit-wise variant. Therefore, the identical analysis applies for the π permutation. The handling of ρ is different, because complete lanes are processed per clock cycle. Hence, the rotation can be implemented as a barrel shifter before storing the lane and no data dependencies must be handled between clock cycles.

- The *row-wise* organization requires a more complicated read and write pattern, because the π permutation reorders rows to columns (Fig. 6.23d).

Furthermore, it has to be ensured, that no unprocessed data is overwritten. Therefore, the old RAM locations are reused to write to the RAM and the read addresses are corrected for the next read. To do that, a round counter has to keep track of the current round. A feasible schedule for the read addresses in terms of the x and y components of the first row would be:

- $(0,0),(1,0),(2,0),(3,0),(4,0)$ for the first round,
- $(0,0),(1,1),(2,2),(3,3),(4,4)$ for the second round,
- $(0,0),(1,2),(2,4),(3,1),(4,3)$ for the third round and so on.

The z coordinate may be handled the same way as it was for the bit-wise implementation.

- A *column-wise* design is very similar to the row-wise version regarding the state. Therefore, the feasible scheduling of the read addresses in terms of the x and y components for the first column is also similar:

  - $(0,0),(0,1),(0,2),(0,3),(0,4)$ for the first round,
  - $(0,0),(3,1),(1,2),(4,3),(2,4)$ for the second round,
  - $(0,0),(1,1),(2,2),(3,3),(4,4)$ for the third round and so on.

  The z coordinate is handled in the same way as for the bit- and row-wise variants.

- The *plane-wise* architecture can be interpreted as a parallel version of the row-wise architecture. Thus, the address scheduling which facilitates the π permutation may be used exactly as for the column-wise version and because complete lanes are processed, no special treatment of the ρ permutation is needed.

- The *sheet-wise* design can be seen as a parallel extension of the column-wise architecture. Hence, the address changes which facilitate the π permutation have to be used exactly as for the column-wise version. However, since complete lanes are processed, the rotations for the ρ permutation can be implemented before storing a lane.

- The *slice-wise* processing has a trivial solution for the π permutation, because π shuffles the bits in one slice, it is just a fix reordering of the bits.

  In contrast, the ρ permutation needs special consideration. For example, the state organization may be split like described above. Then, the first memory contains the most significant parts of the slices computed in parallel and in the second part the least significant bits are stored.

  The rotation for a lane i is then achieved by reading from address j of the two RAMs $\mathrm{RAM}_i[0][j], \mathrm{RAM}_i[1][j]$ in the first clock cycle, but afterwards writing the results to $\mathrm{RAM}_i[1][j], \mathrm{RAM}_i[0][j+1]$. For the rotation constants which are greater than p, an additional round counter is necessary, which adjusts the read addresses for further rounds.

Clock Cycles Estimation The second step of the methodology is to estimate the number of clock cycles to compute the round and the compression function. For Keccak, it is possible for each state organization to design it in a way, that the number of clock cycles to compute the round function depends only on the data path width for each variant of Keccak. However, since different instances of Keccak have a varying number of rounds, the measure varies for the compression function in terms of the number of rounds.

Another detail which has to be kept in mind is the rescheduled version of the compression function. This different schedule leads to the computation of one additional round and hence at least $^{b}/_{d}$ additional clock cycles. Hence, the number of clock cycles to compute the compression function is bounded as follows, where n_r is the number of rounds, d is the data path width, and b is the state size:

$$\mathrm{cyc}_{\mathrm{compress}}(b, d, n_r) \geq \frac{b \times n_r}{d},$$
$$\mathrm{cyc}_{\mathrm{compress}}(b, d, n_r) < \frac{b \times (n_r + 2)}{d}.$$

The upper bound is not strictly adhering to Lemma 5.3 and Corollary 5.4, if the original number of rounds is the reference. However, it can be reasoned, that the rescheduled version is a version of Keccak, which takes one more round than the standard variant and then, the new version is also optimal in a slightly wider sense.

6.8.3 Detailed Analysis

In general, only the χ (Fig. 6.23c) and the θ (Fig. 6.23a) permutations have significant intra-round dependencies. Hence, they influence the detailed architectures significantly. The ι permutation is in all cases trivial to implement, because it only adds a fixed constant to some bits with XOR (Fig. 6.23e). The other two permutations π and ρ are easily addressed for most architectures as read and write addresses to the state memory as shown in the previous analysis or even simpler by a fixed routing or a barrel shifter.

Parallel Architecture The parallel approach is easy to implement. In each clock cycle, the complete state is loaded and the round function is processed in one step. Therefore, the number of clock cycles is optimal, i.e. the number of clock cycles is

$$\mathrm{cyc}_{\mathrm{compress}}(b, d, n_r) = n_r.$$

Thus the total memory required is only the state, i.e. the memory requirements of a BCM C implementing the architecture are estimated to be $\mathrm{SIZE}_{\mathrm{mem}}(C) \geq b$.

Bit-wise Architecture The bit-wise approach can be designed either using the standard round function or with the rescheduled variant. However, the first version needs a very high amount of additional memory (about b bits) because of the ρ permutation to be clock cycle optimal. Therefore, only the latter version with the rescheduled round and compression function is analyzed in detail. For the rescheduled variant, it is enough to analyze the round functions $r_1, \ldots, r_{23}$, where all five sub-permutations are computed, because the derivation of the first and the last round can be easily implemented using additional multiplexers that deactivate the parts not needed for these rounds.

The first step of the proposed architecture loads the bits in a row-first fashion. With this approach, the first output of χ may be computed after the first three input bits have been loaded. However, four inputs bits of each row have to be buffered, because each output bit with $x \in \mathbb{Z}_5$ depends on two bits with $x' = (x + 1) \bmod 5$ and $x'' = (x + 2) \bmod 5$ (Fig. 6.23c). The fifth input may replace the third input in the clock cycle, when the third output is generated. Therefore, not five but only four registers are strictly necessary.

It is somewhat more difficult to find a good design for θ, because at least eleven input bits from two different slices are necessary to produce one output bit (Fig. 6.23a). Therefore, because of the row-first strategy used to compute χ, at least the complete output for one slice has to be stored and additional 21 bits of the consecutive slice. Then the dependencies for the first four bits are met with $x = 1$, $0 \leq y \leq 4$. These four bits are the white bits of the slice with $z = 1$ in Fig. 6.25. After the 46th output bit of χ is computed, one output bit of θ can be generated in each clock cycle.

The last two permutations π and ρ are implemented using the read and write addresses calculated according to the previous discussion about the I/O scheduling of the RAM (Fig. 6.23d, Fig. 6.23b).

The resulting straightforward implementation takes $2 + 25 + 21 = 48$ clock cycles to produce the first output with $x = 1, y = 0$, and $z = 1$. Two clock cycles are due to the χ function, in the first 25 clock cycles parts of θ for the first slice can be computed and then additional 21 clock cycles are needed until the first output of θ is available.

One problem remains, which is the case $z = 0$. The output of θ for $z = 0$ cannot be computed before the computation of χ for $z = 2^l$. Therefore, either 25 additional clock cycles per round are needed, or a shortcut to store two bits per clock cycle in the last round, i.e. an extra memory for 25 bits is needed. The latter option removes 25 clock cycles of the overhead again, thus the overhead reduces from 48 to 23 clock cycles.

Another alternative is to start the second round with a different value for z instead of $z = 0$, for example $z = 46 \bmod 2^l$, the third with $z = (2 \times 46) \bmod 2^l$ and so on. It is possible to use several different values for z in this case, as

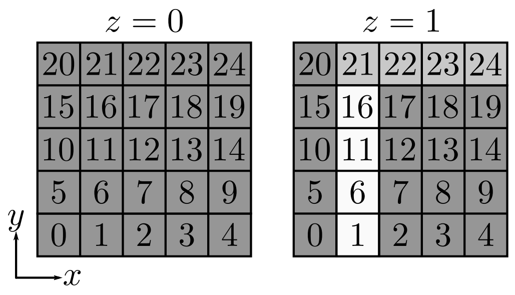


Figure 6.25: Partial fulfillment of the dependencies of KECCAK's θ in the bit-wise architecture for $x = z = 1$.

long as inter-round dependencies introduced by the ρ permutation are fulfilled. Since ρ rotates lanes, this is the case for all slices that do not depend on an output bit of $z = 0$.

The storage requirements for this strategy are as follows. For each row, at least four bits have to be provisioned to store the inputs for χ. Additionally, for a slice $z = i$, at least five bits are necessary to store the intermediate results of the slice with $z = i - 1 \bmod 2^l$ to compute θ. Furthermore, at least 20 bits are needed, because they have to be buffered until the first result may be computed in the 21th clock cycle, and for $z = 0$, 25 bits have to be stored, because of the inter-round dependencies between the slices with $z = 0$ and $z = 2^l$. Together with the state, this amounts to $\text{SIZE}_{\text{mem}}(C) \geq b + 4 + 5 + 20 + 25 = b + 54$ bits for a BCM C implementing this architecture.

To meet the estimate on the number of clock cycles from above, the shortcut or the moving starting slice per round are used. The last approach has the additional benefit, that it is possible to implement a very high number of pipelining steps. However, the exact value depends heavily on the parameter l and the chosen value for z and thus, the pipelining is not analyzed in detail.

Lane-wise Architecture The lane-wise architecture can be in parts seen as an extension of the bit-wise architecture, processing the bits of a lane in parallel instead of individual bits.

Therefore, similar to the bit-wise architecture a row-first strategy for χ is used. That means, if a clock cycle of the processing of the compression function is denoted as t, then in each clock cycle $t \in \mathbb{N}$ a lane is read with the coordinates $x \in \mathbb{Z}_5, y \in \mathbb{Z}_5$, fulfilling the equation $x + 5y = t - 25\lfloor t/25 \rfloor$. In the first two clock cycles χ may not produce an output. However, afterwards in each cycle the output for one lane may be generated. This strategy for computing χ only works, if four lanes $(4 \times 2^l bits)$ are buffered in registers, identical to the bit-wise case.

A schedule for the operations of θ is more difficult, because after 25 clock cycles, the output for the complete lane with $x = y = 0$ has to be available. An elegant solution calculates the correct output lane on demand in the clock cycle before it is used as an input value. For which lane the output has to be computed is determined by the π permutation, because π shuffles the lanes and creates the only inter-round dependencies for this architecture. The following steps help to understand the general idea:

1. Most outputs of χ are computed and stored before the computation of θ may produce the first output lane.

2. Evaluate the θ function iteratively according to the addresses calculated to implement the π permutation, to fulfill the inter-round dependencies.

Analyzing a naïve implementation of this strategy shows that, only in the 27th clock cycle, the dependencies to compute θ for the lane $x = y = 0$ are fulfilled. Hence, this strategy does not work in an optimal way.

However, it is possible to reorder the computation of the outputs of χ in such a way that the output for this lane can be computed two clock cycles earlier. In particular, computing the outputs of χ with the x component in the order $0, 1, 4, 2, 3$ instead of $0, 1, 2, 3, 4$ facilitates the computation for the lane with $x = y = 0$ two clock cycles earlier and thus in the 25th clock cycle, when the last input of a round i is supplied to χ, the output for $x = y = 0$ is computed (Fig. 6.26). Furthermore, it guarantees, that it is possible to generate the output in the order $0, 6, 12, 18, 24$ in terms of the inputs of the round i to fulfill all dependencies of π (Fig. 6.23d). And hence, the total overhead of 24 additional clock cycles for the computation of the complete KECCAK compression function is within the theoretical optimal bounds. The downside is, that instead of only four input lanes, all five lanes of a sheet have to be stored, increasing the memory footprint slightly.

The first and the last round of the rescheduled version are somewhat special. For the first round, the input is not directly supplied by the computation of χ and ι, but loaded from the state RAM. And in the last round instead of storing

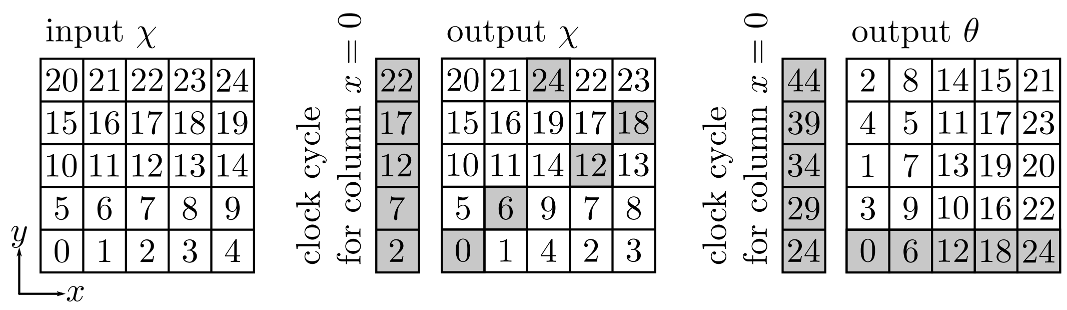


Figure 6.26: Reordering for the lane-wise KECCAK architecture at work.

the intermediate results to compute θ, the results are stored directly to the state RAM.

The memory overhead for this strategy is higher than for the bit-wise version. For the basic implementation of χ, a memory to store five lanes is necessary, i.e. 5×2^l bits. The results of χ then have to be stored in a memory of b bits for the on-the-fly calculation of the θ outputs. For a BCM C implementing the strategy, this means $\text{SIZE}_{\text{mem}}(C) \geq 2b + 5 \times 2^l = 55 \times 2^l$ bits.

Column-wise Architecture The lane-wise architecture, is also a parallelized extension of the bit-wise version. This time, instead of lanes, columns are loaded as inputs. Hence, the only option to compute χ is to load the columns with the row-first strategy. This strategy has an overhead of two clock cycles for the computation of χ and it is necessary to store four columns in an intermediate memory. A correct implementation must additionally buffer the output for the first column of the next slice. Additionally, for $z = 0$ the output of χ for the slice has to be stored completely, because of the intra-round dependency with $z = 2^l$.

The θ permutation is computed next. It is split in two parts. First, the five columns are summed up and each sum is stored. For the next slice, the same process happens and additionally, the outputs for θ are processed in parallel. If the x coordinate is traversed in an ascending order, the first output column has the coordinates $(z = 1, x = 1)$, followed by $(z = 1, x = 2)$, $(z = 1, x = 3)$ and so on. For θ, only 6 additional bits for the column sums have to be stored, because if the outputs for $z = i$ are computed, each intermediate bit of $z' = i - 1$ is only used once (Fig. 6.23a). The outputs of θ are stored according to the previous discussion.

The special case for $z = 0$ can be solved in the same way as for the bit-wise implementation. Again a shortcut or the version with an increasing starting value for z are two elegant solutions. The latter one allows a deep pipeline. However, the exact pipeline depth is again not easily determined in a general way. Another disadvantage of the solution using the shortcut is, that it needs additional 25 bits to store a copy of the $z = 0$ slice, because otherwise parts of this memory would be overwritten. In any case, the overhead to compute the round function for this design consists of two clock cycles for χ and six clock cycles until the first output of θ is available and thus in total is eight clock cycles.

Overall, the column-wise design needs 25 bits for the χ permutation buffer and additional 25 bits to store the intermediate values for $z = 0$. Furthermore, 6 bits for the intermediate column-sums to compute θ are necessary. Summed up, a BCM C implementing KECCAK using this strategy has memory requirements of at least $\text{SIZE}_{\text{mem}}(C) \geq 25 \times 2^l + 56$.

Row-wise Architecture The row-wise design is very similar to the column-wise architecture. The most interesting difference is, that the χ function can now be computed without any offset and without any temporary memory requirements.

However, because of the very same feature, the θ function needs ten clock cycles to compute the first output for $y = 0$ and $z = 1$. This means, the architecture has an overhead of nine clock cycles.

Therefore, for one slice with $z = i$, five bits are necessary to buffer the intermediate results for θ. Five bits are also needed for each consecutive slice $z = i + 1$ and additionally this same slice, 20 memory bits have to be used as a buffer for four rows of the slice. The slice with $z = 0$ has to be also saved. Overall, this amounts to $5+5+20+25 = 55$ bits of intermediate memory and in total a BCM C needs at least $\text{SIZE}_{\text{mem}}(C) \geq 25 \times 2^l + 55$. The possible solutions for the $z = 0$ case are practically the same as for the column-wise version, including a very similar pipelining option. The shortcut needs additional 25 bits to store a intermediate values of the $z = 0$ slice.

Sheet-wise Architecture The sheet-wise organization can be interpreted as a parallelization of the lane-wise design. Since complete lanes and complete columns are loaded, the χ permutation has to store the complete state in an intermediate RAM. Therefore, the memory requirements are high.

The θ permutation works similar to the lane-wise version. After the first five clock cycles, the first sheet is already scheduled as an input again. Hence, the output has to be available after the fifth clock cycle. Therefore, the corresponding output lanes have to be computed on the fly corresponding to the shuffling by the π permutation using another intermediate copy of the complete state. For a working implementation, the computation of χ has to be reordered in the same way as the lane-wise architecture, hence the outputs are computed in the order $0, 1, 4, 2, 3$. The overhead is four clock cycles, and the memory requirements for the intermediate storage are very high with $2 \times 25 \times 2^l$ bits and in total $\mathrm{SIZE}_{\mathrm{mem}}(C) \geq 75 \times 2^l$.

It is also possible to implement a trade-off between the serialization of χ and memory requirements, by computing more outputs of χ in parallel on demand. Then, the memory requirements reduce by 25×2^l at the cost of a parallelization of χ and a deeper circuit. However, this architecture is basically transforming the sheet-wise architecture in a plane-wise variant, which is clearly inferior. Therefore, this variant is not further discussed.

Plane-wise Architecture The plane-wise organization is almost identical to the sheet-wise architecture. However, it is possible to compute the χ permutation for each plane without storing a temporary copy of the state and thus, the memory requirements are lower with only $\mathrm{SIZE}_{\mathrm{mem}}(C) \geq 50 \times 2^l$, however the offset of four clock cycles is identical.

Slice-wise Architecture The slice-wise design is the most flexible approach. The smallest possible variant uses exactly one slice. For this variant, it is easy to see, that the computation of χ is trivially implemented, because the data dependencies of χ are always fulfilled. Also, the calculation of θ is straightforward. One possible strategy first computes the column-sums in a clock cycle t for a slice z, i.e. for all $x \in \mathbb{Z}_5$, $\sum_{i=0}^{4} a[x][i][z]$ is evaluated and stored. In the next clock cycle $t + 1$, the process is repeated for $z' = z + 1$ and furthermore, the output for z' is calculated in the same clock cycle. Therefore, the overhead is exactly 1 clock cycle, because no output is available after the first clock cycle.

Again, the special case $z = 0$ has to be solved, by either using an exception, i.e. the results for the slice with $z = 0$ are computed in parallel to the computation of $z = 2^l - 1$, or with the moving starting slice per round. If the special case is correctly solved, all other inter-round dependencies are fulfilled. However, the second option becomes more difficult for the variants processing multiple slices in parallel, therefore the version using the shortcut is to be preferred for a generic implementation.

The previously described basic strategy is called Slice-25, because it has a data path width of 25 bits. This can be extended to Slice-d, with $d \in \{25 \cdot 2^k | 1 \leq k \leq l\}$. The idea is to parallelize the computation of several slices in one clock cycle. This is straightforward, because the basic scheme stays the same.

The number of intermediate values to be stored in the implementation of the round function is the same for all Slice-d variants. This includes 5 bits for the column-sums which are carried over to the next sub-round and 25 bits for the results of χ for the $z = 0$ special case. Altogether 30 additional bits have to be stored. Additional 25 bits are necessary for the version that uses the shortcut to process the slice with $z = 0$, because some bits of the output for this slice are read in several clock cycles, but the output for the next round would overwrite the registers after the first clock cycle, except the bit with $x = y = z = 0$. In total a BCM C implementing the architecture with the shortcut needs at least $\text{SIZE}_{\text{mem}}(C) \geq b + 55$ bits.

6.8.4 Evaluation Summary

The results of the detailed analysis (Tab. 6.12) show, that the slice-wise approach is a very competitive and scalable architecture and the other serialization options are all worse in one or more aspects. The sheet- and plane-wise variants need a lot of memory and thus, it can be conjectured, that they do not scale well in terms of the area consumption.

The bit-, column- and row-wise versions probably allow for smaller implementations than the slice-wise variant, because the data path width is reduced and the memory requirements are similar. However, all three implementation styles need slightly more intermediate memory and a lot more clock cycles per round. Since the general design goal is not to push the area consumption to its limit (Sec. 6.4), the slice-wise approach meets this goal better than the other architectures.

The remaining option, the lane-wise implementation may have a better trade-off between area consumption and throughput than the other six variants, mainly because the number of clock cycles is less than for the other smaller architectures and because the memory consumption is less than for the sheet- and plane-wise variants. However, compared to the slice-wise design the memory consumption is high and the general approach is also less scalable, because

Table 6.12: Summary of the theoretical analysis of Keccak

Variant	Data Path d [Bits]	Memory [Bits]	ROM [Bits]	Analyzed Pipeline Depth	Clock Cycles	Overhead	Long Message Throughput @ 100 MHz [MBits/s]	Short Message Throughput @ 100 MHz [MBits/s]
Parallel	b	b	$(1+l)\times n_r$	1	n_r	0	$\frac{100\times r}{n_r}$	$\frac{100\times r}{n_r}/\lceil\frac{n}{r}\rceil$
Bit	1	$b+54$	$(1+l)\times n_r$	23	$(n_r+1)\times b$	22	$\frac{100\times r}{(n_r+1)\times b}$	$\frac{100\times r}{(n_r+1)\times b}/\lceil\frac{n}{r}\rceil$
Lane	2^l	55×2^l	$(1+l)\times n_r$	25	$(n_r+1)\times 25$	24	$\frac{100\times r}{(n_r+1)\times 25}$	$\frac{100\times r}{(n_r+1)\times 25}/\lceil\frac{n}{r}\rceil$
Column	5	$b+56$	$(1+l)\times n_r$	9	$(n_r+1)\times\frac{b}{5}$	8	$\frac{100\times r\times 5}{(n_r+1)\times b}$	$\frac{100\times r\times 5}{(n_r+1)\times b}/\lceil\frac{n}{r}\rceil$
Row	5	$b+55$	$(1+l)\times n_r$	10	$(n_r+1)\times\frac{b}{5}$	9	$\frac{100\times r\times 5}{(n_r+1)\times b}$	$\frac{100\times r\times 5}{(n_r+1)\times b}/\lceil\frac{n}{r}\rceil$
Sheet	5×2^l	$3b$	$(1+l)\times n_r$	5	$(n_r+1)\times 5$	4	$\frac{100\times r}{(n_r+1)\times 5}$	$\frac{100\times r}{(n_r+1)\times 5}/\lceil\frac{n}{r}\rceil$
Plane	5×2^l	$2b$	$(1+l)\times n_r$	5	$(n_r+1)\times 5$	4	$\frac{100\times r}{(n_r+1)\times 5}$	$\frac{100\times r}{(n_r+1)\times 5}/\lceil\frac{n}{r}\rceil$
Slice-d	d	$b+55$	$(1+l)\times n_r$	1	$(n_r+1)\times\frac{b}{d}$	0	$\frac{100\times r\times d}{(n_r+1)\times b}$	$\frac{100\times r\times d}{(n_r+1)\times b}/\lceil\frac{n}{r}\rceil$

processing for example two lanes in parallel results in a more complicated architecture, because the state is not organized with a number of lanes that is divisible by two.

6.8.5 Implementation

The analysis of the different Keccak architectures shows, that the slice-wise implementation has a very good trade-off between the additional memory requirements and the number of clock cycles. Therefore, two slice-based implementations were developed [Jun12, JS13]. The first uses the FSL-based interface and is fixed to process 8 slices in parallel and uses the 1600 bit state of the SHA-3 candidate Keccak-f[1600] (Fig. 6.27). With this configuration, the implementation is comparable to the other SHA-3 candidates. The number of clock cycles per message block corresponds exactly to the analysis above, if the FSL interface is saturated with enough user data.

The second version uses the GMU interface and can be parameterized to

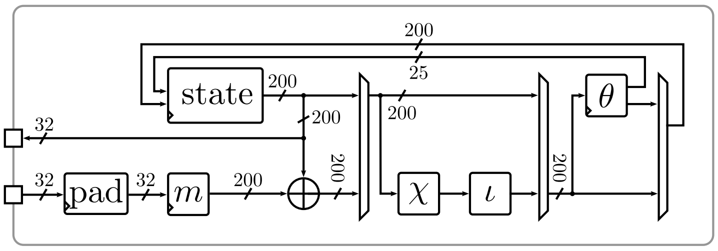


Figure 6.27: First Keccak fixed architecture.

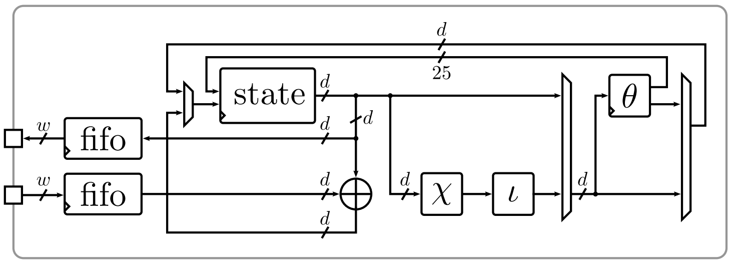


Figure 6.28: Second generic KECCAK architecture.

change the data path width, the state size, the rate and the size of the message digest. As depicted in Fig. 6.28, the proposed architecture stays the same for all variants and only the data path width varies. For this version of the implementation, the message absorption is separated and thus the number of clock cycles per message block is higher for the same number of slices.

The main difference between the two implementations is the handling of the message absorption phase, and the organization of the state RAM. The first implementation uses a RAM to buffer message blocks and to change the order of the input bits from the standard lane-wise to a slice-wise organization. Additionally, the state RAM is using the double amount of the state size, because then it is simpler to organize the read and write addresses than the concept described in Sec. 6.8.2.

The state RAM is also organized in individual lanes. However, the permutation of ρ is achieved by using temporary registers. These registers are used to store the part of the output, that cannot be written to the RAM just as yet, because the rotation constant is not divisible by the number of slices processed in parallel. Then, in the next cycle when the next output has to be written to

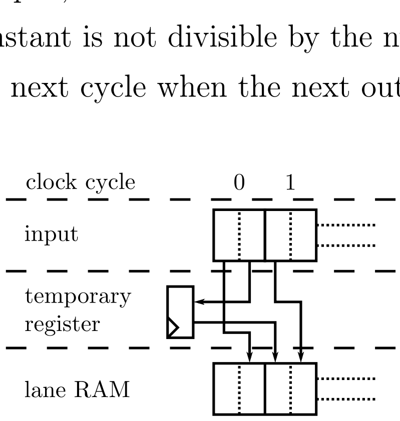


Figure 6.29: Illustration of an alternative implementation of ρ.

the RAM, this new output is again split into two parts, and the first part of this output is concatenated with the register content and the second part is again written to this register (Fig. 6.29).

The second implementation uses the state organization for slice-wise implementations as described in Sec. 6.8.2. Furthermore, it removes the buffer for the input message to save area. Therefore, the message absorption phase has to be implemented differently. The input data is split up into $d/25$ bit wide chunks and each such part is absorbed in one clock cycle at its appropriate place in the state RAM. For example if $d = 25$ and the interface is $w = 16$ bit wide, then 16 clock cycles are needed for the absorption of these 16 bits, because every bit is absorbed in a separate clock cycle.

The second architecture is also optimized by manually instantiating `LUT6_2` primitives. In particular the χ and ι functions may be combined with the multiplexer before θ. This idea works nicely, because for each output bit of χ only three input bits are necessary, ι only influences one or zero bits per row and the multiplexer selects only between the computation of χ and ι or route-through. Therefore, the computation of four output bits can be packed into two `LUT6_2` instances with five input bits, i.e. four inputs and the multiplexer bit. A third `LUT6_2` instance is needed for the bit with $x = 0$ which has three data inputs, the multiplexer bit and additionally the bit for ι. Overall, only three `LUT6_2` instances are needed per row.

6.9 Skein

The Skein hash function was originally submitted to the SHA-3 competition by Ferguson et al. and was modified for the last round of the competition [FLS+10]. However, only the key schedule constants were changed. The Skein hash function is based on the block cipher Threefish and the iteration mode UBI. Threefish is a ARX-based design, i.e. it uses addition, rotation and XOR and is thus a relatively conservative design, because many older algorithms are based on these operations [Riv92].

6.9.1 Definition

The original specification of Skein defines four variants for $224, 256, 384$ and 512 bit hash digests [FLS+10]. As for most of the other SHA-3 finalists, only the 256 bit variant is defined here.

Input and output mapping Skein uses a little endian byte order, which is different from the big endian order defined in Sec. 6.2. Hence, the byte order is exactly the opposite. A byte string $x \in \mathbb{Z}_{256}^{\geq 0}$ with $|x|/8$ bytes is ordered as follows:

$$x =_{\text{def}} \sum_{i=0}^{|x|/8} x_i \times 256^i.$$

Padding The padding for function of Skein is defined as follows. For a message $M \in \mathbb{Z}_2^{\geq 0}$ and $K = (-|M|) \bmod 512$:

$$\text{pad}_{256}(M) = \begin{cases} 0^{512}, & \text{if } |M| = 0 \\ M||0^K, & \text{if } |M| > 0 \end{cases}$$

Initialization Values and Constants The state h is initialized with the following initialization values.

$$\begin{aligned} \text{IV}_0 &= \texttt{CCD044A12FDB3E13}, & \text{IV}_1 &= \texttt{E83590301A79A9EB}, \\ \text{IV}_2 &= \texttt{EC06025E74DD7683}, & \text{IV}_3 &= \texttt{E7A436CDC4746251}, \\ \text{IV}_4 &= \texttt{55AEA0614F816E6F}, & \text{IV}_5 &= \texttt{C36FBAF9393AD185}, \\ \text{IV}_6 &= \texttt{2A2767A4AE9B94DB}, & \text{IV}_7 &= \texttt{3EEDBA1833EDFC13} \end{aligned}$$

These values are the output of a computation of the Threefish block cipher, which has an all-zero key and a configuration string as input.

Key Schedule Since Threefish is a typical block cipher, it has a key schedule. In the hashing mode, the key $k = (k_0, \dots, k_7)$ is the output of the previous computation of the Threefish block cipher, or the initialization values. Another input is the so-called tweak, consisting of two 64 bit words t_0 and t_1.

The key schedule for the currently discussed version uses nine 64 bit words, where the ninth is generated as follows, where is defined to be $C_{240} =_{\text{def}}$ `1BD11BDAA9FC1A22`. This constant has been chosen by Skein's designers to defend against some attacks on the key schedule [FLS+10].

$$k_8 = C_{240} \oplus \bigoplus_{i=0}^{7} k_i$$

Furthermore, another 64 bit word is used:

$$t_2 = t_0 \oplus t_1$$

The key schedule key uses these words as follows, where s is the current sub-round and i is the index of the state word:

$$\text{key}(k, t, s, i) = \begin{cases} k_{(s+i) \bmod 9}, & \text{for } i = 0, \dots, 4 \\ k_{(s+i) \bmod 9} + t_{s \bmod 3}, & \text{for } i = 5 \\ k_{(s+i) \bmod 9} + t_{(s+1) \bmod 3}, & \text{for } i = 6 \\ k_{(s+i) \bmod 9} + s, & \text{for } i = 7 \end{cases}$$

Round Function The round function consists of key injection, MIX function (Alg. 6.15) and permutation layer according to Alg. 6.16. This function is iterated for 72 rounds. The values for the word permutation π are defined as follows:

$$\pi = (2, 1, 4, 7, 6, 5, 0, 3).$$

The $\text{MIX}_{j,i}(x_0, x_1)$ function is defined according to Alg. 6.15, where $i \in \mathbb{Z}_4$ is an index to lookup the rotation constant and $j \in \mathbb{Z}_{72}$ is the current round. The rotation constants $R_{j,i}$ can be found in the Skein specification [FLS+10].

Iteration Mode The iteration mode UBI can be parameterized to yield several different modes, e.g. a MAC, tree hashing or a simple iterated hash. The last option is used in case of the SHA-3 candidate. The parameterization works by specifying appropriate values for the tweak $t = (t_0, t_1)$, where the length is encoded together with several other bits that signal the first and the last message block and if the currently processed message block contains padding bits or not. The iteration mode itself is a variation of the Matyas-Meyer-Oseas construction [MMO85].

6.9.2 Systematic Evaluation Overview

State and Memory Organization The state of the investigated Skein variant is organized in eight 64 bit words. Therefore, it is natural to split the state in the following ways:

- A parallel architecture reads and writes the complete state in one clock cycle and uses a 1×512 bits memory.
- The serialized versions use $2^z \times ({}^{512}\!/_{2^z})$ organizations, where $1 \leq z \leq 3$.

The key schedule has a 576 bit state, which may be organized almost identical to the state memory. However, the ninth 64 bit part of the state may be stored in an additional register, such that no pipeline stalls are necessary to reorder the key schedule (Fig. 6.30). Furthermore, 192 bits are necessary to store the tweak and its generated third part and 512 bits for the message. Overall the memory size of a BCM C takes at least $\text{SIZE}_{\text{mem}}(C) \geq 1280$.

In addition to the memory requirements, the initialization value has to be stored in a ROM. Therefore, 512 bits are necessary as ROM.

Algorithm 6.15 Threefish MIX function [FLS+10]

```
Require: $h_0, h_1 \in \mathbb{Z}_{64}$, $j \in \mathbb{Z}_{72}$, $i \in \mathbb{Z}_4$
Ensure: $h \leftarrow \text{MIX}_{j,i}(h_0, h_1)$
  $h_0 \leftarrow (h_0 + h_1) \bmod 2^{64}$
  $h_1 \leftarrow (h_1 \lll R_{(j \bmod 8),i}) \oplus h_0$
  return $(h_0, h_1)$
```

Algorithm 6.16 Threefish round function [FLS+10]

```
Require: $h \in \mathbb{Z}_{64}^8$, $j \in \mathbb{Z}_{72}$, $t \in \mathbb{Z}_{64}^2$, $k \in \mathbb{Z}_{64}^8$
Ensure: $h \leftarrow \mathrm{round}(k, t, h, j)$
  for $i = 0$ to 7 do
    $h_i \leftarrow \begin{cases} h_i + \mathrm{key}(k, t, j/4, i) & \text{if } j \bmod 4 = 0 \\ h_i & \text{otherwise} \end{cases}$
  end for
  for $i = 0$ to 3 do
    $(h_{2i}, h_{2i+1}) \leftarrow \mathrm{MIX}_{j,i}(h_{2i}, h_{2i+1})$
  end for
  for $i = 0$ to 7 do
    $h_i \leftarrow h_{\pi[i]}$
  end for
  return $h$
```

Algorithm 6.17 Threefish encryption [FLS+10]

```
Require: $h \in \mathbb{Z}_{64}^8$, $t \in \mathbb{Z}_{64}^2$, $k \in \mathbb{Z}_{64}^8$
Ensure: $h \leftarrow \mathrm{E}(h, t, k)$
  for $i = 0$ to 71 do
    $h \leftarrow \mathrm{round}(k, t, h, i)$
  end for
  for $i = 0$ to 7 do
    $h_i \leftarrow h_i + \mathrm{key}(k, t, 72/4, i)$
  end for
  return $h$
```

State Read and Write Schedule The state read and write schedule has to be implemented according to the round permutation. For a none-pipelined serialized version, the write addresses are identical to the previous read addresses.

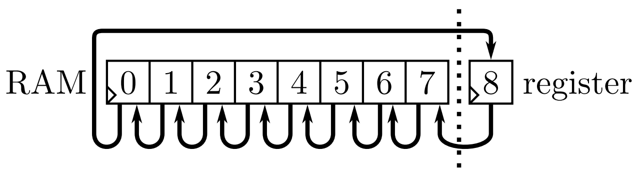


Figure 6.30: The Skein key schedule memory architecture.

Therefore, the read addresses for the next round have to be adjusted using a round counter. This procedure can be used for all serialized variants in a similar fashion, and is unnecessary for the parallel version.

Clock Cycles Estimation The number of clock cycles is directly proportional to the data path width d and the unrolling which is expressed in the serialization metric s. However, it is not in the theoretical bounds described in Sec. 5.4, because of the additional key injection after the 72nd round. Yet, it is similar to the case of the rescheduled KECCAK versions and thus, the last key injection can be viewed as a modified 73rd round of Skein. In total the number of clock cycles is bounded as follows, where the state size is assumed to be 512 bit, the number of rounds is 73, s is the serialization metric and d the data path width.

$$\begin{aligned}\mathrm{cyc}_{\mathrm{compress}}(d) &\geq \frac{512 \times 73 \times s}{d},\\ \mathrm{cyc}_{\mathrm{compress}}(d) &< \frac{512 \times (73+1) \times s}{d}.\end{aligned}$$

6.9.3 Detailed Analysis

Parallel and Unrolled Architectures A parallel architecture of Skein is based on a straightforward implementation of the key schedule and the round function of the Threefish block cipher. Therefore, not a single clock cycle overhead is needed and the memory corresponds exactly to the state memory, the key schedule and the tweak.

It is also possible to unroll Skein to increase the throughput. This improvement is possible, because the logic depth for the different rotations can be decreased and hence, the length of the critical path does not grow proportional to the unrolling factor. In the literature, unrolling of $2, 4$ and 8 rounds is reported [GHR10, ATM+13]. All of these basic architectures have the same properties regarding the RAM consumption and the clock cycle overhead.

A BCM C implementing the architecture does not need any additional memory and hence, $\mathrm{SIZE}_{\mathrm{mem}}(C) \geq 1280$.

Serialized Architectures Serialization of Skein works best according to the data path width $^{512}/_{2^z}$ bits according to the analysis of the state organization above. Therefore, three basic variants can be implemented this way. The basic architecture loads one or several locations of the memory and then writes them back to the state RAM after one clock cycle. The read and write addresses are calculated such that the inter-round dependencies from the the permutation layer are fulfilled.

For the variant with a data path width of 64 bits, one clock cycle overhead is produced, because no output can be computed after the first clock cycle. Therefore, it is also necessary to have an additional 64 bit register for this variant to store this intermediate value. A pipelined architecture is also possible, with a combined depth of 6 for this variant [Jun12].

A BCM C implementing the architectures with a data path width greater than 64 does not need additional memory and hence, for these variants the same bound is valid $\mathrm{SIZE}_{\mathrm{mem}}(C) \geq 1280$. For the version with a data path width of 64 at least one additional 64 bit register is necessary and hence, the C_{64} implementing that architecture takes $\mathrm{SIZE}_{\mathrm{mem}}(C_{64}) \geq 1280$. The pipelined version then adds at least another 64 bits per pipeline stage.

6.9.4 Evaluation Summary

The evaluation of the Skein hash function shows, that it scales pretty well in theory. In the one direction, unrolling improves the theoretical throughput

Table 6.13: Summary of the theoretical analysis of Skein

Data Path d [Bits]	Serialization Metric	Memory [Bits]	ROM [Bits]	Analyzed Pipeline Depth	Clock Cycles	Minimum Overhead	Long Message Throughput @ 100 MHz [$^{\mathrm{MBits}}/_{\mathrm{s}}$]
512	0.25	1280	576	1	19	0	2694
512	0.5	1280	576	1	37	0	1383
512	1	1280	576	1	73	0	701
256	1	1280	576	1	146	0	350
128	1	1280	576	1	292	0	175
64	1	1344	576	2	584	1	87
64	1	1664	576	6	584	5	87

linearly. However, this only helps, if the clock frequency does not significantly decrease. Therefore, the throughput estimate is probably not that meaningful.

In the other direction towards lightweight implementations, the high area consumption of fast 64 bit adders and the high number of rounds is a drawback, that either leads to a pretty high number of clock cycles or a higher area consumption.

For the implementation, an area reduced version with a good overall throughput was the main evaluation goal. Therefore, the version with a 64 bit data path width has been selected to be implemented. The decision was mainly based on the reduction of the number of adders.

6.9.5 Implementation

The presented implementation was published in [Jun12]. The state for the implementations can be stored in distributed RAM with 512 bits. However, the implementation uses a 1024 bit state, where a single part is used as the input to the round and the other part to save the output to simplify the read and write address calculation. Furthermore, the input message block is buffered in a wide RAM with also 1024 bit, which allows loading a new message block while the computation of the current message block is in process. Additionally, the key schedule uses a 9×64 bits distributed RAM to store the different subkeys. The architecture is depicted in Fig. 6.31.

Efficient implementations of Skein for FPGAs are quite a challenge, which is mainly caused by the 64 bit adders and further complicated by the rotations which impact the routing on an FPGA device and furthermore the necessary

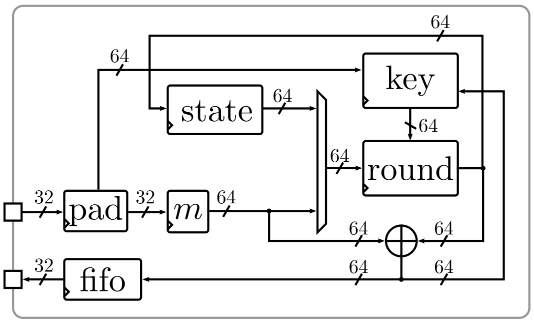


Figure 6.31: Skein implementation.

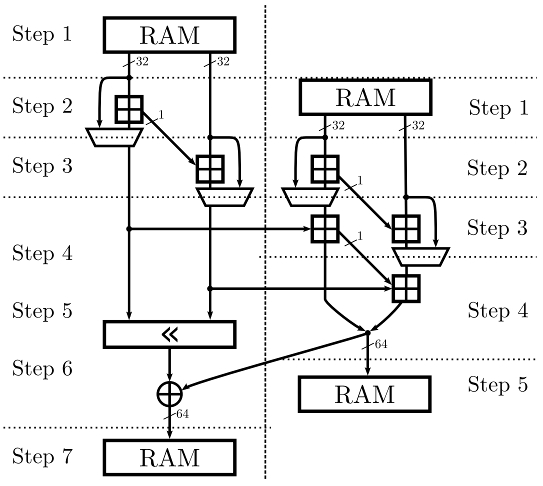


Figure 6.32: Implemented Skein pipeline.

barrel shifter adds a lot of area. Together both features have a significant impact on the maximum achievable clock frequency. Pipelining the round function is the obvious countermeasure, but this is not as easy as for some other hash functions.

The pipeline itself consists of two distinct parts, each using 64 bit as input (Fig. 6.32). The three necessary 64 bit adders are split into 32 bit adders and the computation is scheduled in such a way, that only three of them are needed. The rotation and the last XOR are distributed over three clock cycles, which makes it possible to reduce the cost of the rotations. The two parts have two different depths, such that the pipeline never stalls. The reason for this pipeline design is the permutation layer of Skein.

The performance of this design is dominated by the large number of rounds required. Overall the architecture requires 584 clock cycles for one execution of the compression function (72 rounds + 1 extra round for the last key injection and 8 clock cycles for each round, thus $73 \times 8 = 584$).

6.10 Photon

Photon is the only hash algorithm investigated in this thesis, that was not part of the SHA-3 competition [GPP11]. The design goal of Photon is a low area implementation, and hence, in this thesis it competes only with the low-area variants of Keccak. The algorithm uses the sponge construction developed by Bertoni et al. [BDPA07] and borrows the structure of its internal round permutation from AES. However, the four sub-permutations are slightly different than the AES ones to allow for a less area consuming implementation.

6.10.1 Definition

The Photon hash function is specified in [GPP11] and is adapted here to the common structure. The specification defines five variants with different area consumption and security levels. It furthermore extends the sponge construction to use different input and output rates to improve the preimage resistance. Five different configurations are defined, which are shown in Tab. 6.14.

State Size and Representation The state of Photon can be interpreted in a two-dimensional way, similar to Grøstl. The size t of Photon's state depends on the parameters $\mathfrak{d}$ and $\mathfrak{s}$. The parameter $\mathfrak{d}$ defines the number of cells in the two-dimensional representation ($\mathfrak{d}^2$ cells), and $\mathfrak{s} \in \{4, 8\}$ defines the number of bits per cell and thus $t = \mathfrak{s}\mathfrak{d}^2$.

Input and output mapping The output mapping corresponds to the bit-string convention defined in Sec. 6.2. A message block m is mapped to the

Table 6.14: The different Photon variants

Variant	Hash digest n	State size b	Input rate r	Output rate r'	Capacity c
	[Bits]	[Bits]	[Bits]	[Bits]	[Bits]
Photon-80/20/16	80	100	20	16	80
Photon-128/16/16	128	144	16	16	128
Photon-160/36/36	160	196	36	36	160
Photon-224/32/32	224	256	32	32	224
Photon-256/32/32	256	288	32	32	256

$\mathfrak{d} \times \mathfrak{d} \times \mathfrak{s}$ matrix representation of the state h as follows, where m is interpreted bit-wise.

$$h[i][j][k] \leftarrow m[\mathfrak{s}\mathfrak{d}i + \mathfrak{s}j + k]$$

Padding The padding used by PHOTON appends a string 10^*, such that the length of the padded message is a multiple of r. Formally, for a message $M \in \mathbb{Z}_2^{\geq 0}$ and $k = (-|M| - 1) \bmod r$, the padding function is defined as:

$$\text{pad}_{256}(M) =_{\text{def}} M||1||0^k$$

Initialization Values and Constants The state is initialized to the following value, where t is the state size, n is the size of the hash digest, r is the input rate and r' is the output rate:

$$\text{IV} =_{\text{def}} 0^{t-24}||n/4||r||r'$$

Furthermore, Photon uses a number of round constants, which are defined as follows:

$$\text{RC} =_{\text{def}} (\texttt{1}, \texttt{3}, \texttt{7}, \texttt{E}, \texttt{D}, \texttt{B}, \texttt{6}, \texttt{C}, \texttt{9}, \texttt{2}, \texttt{5}, \texttt{A})$$

Additionally, some so called internal constants IC_d are defined for each variant, dependent on the parameter $\mathfrak{d}$:

$$\begin{aligned}
\text{IC}_5 &=_{\text{def}} (\texttt{0}, \texttt{1}, \texttt{3}, \texttt{6}, \texttt{4}) \\
\text{IC}_6 &=_{\text{def}} (\texttt{0}, \texttt{1}, \texttt{3}, \texttt{7}, \texttt{6}, \texttt{4}) \\
\text{IC}_7 &=_{\text{def}} (\texttt{0}, \texttt{1}, \texttt{2}, \texttt{5}, \texttt{3}, \texttt{6}, \texttt{4}) \\
\text{IC}_8 &=_{\text{def}} (\texttt{0}, \texttt{1}, \texttt{3}, \texttt{7}, \texttt{F}, \texttt{E}, \texttt{C}, \texttt{8})
\end{aligned}$$

Round Function and Photon Permutation The round function of Photon has a structure very similar to AES (Alg. 6.18). However, the internal details are changed. The first function is AddConstants, which adds a constant to the state as follows, where $i \in \mathbb{Z}_1 2$ and $j, k \in \mathbb{Z}_{\mathfrak{d}}$:

$$\text{AddConstants}_{\mathfrak{d}} : h[j][k] \leftarrow \begin{cases} h[j][k] \oplus \text{RC}[i] \oplus \text{IC}_{\mathfrak{d}}[k] & \text{if } j = 0 \\ h[j][k] & \text{if } j \geq 1 \end{cases}$$

Table 6.15: The PRESENT S-box

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S_{\text{PRE}}(x)$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

The next function is SubCells. For the Photon variants using $\mathfrak{s} = 4$ bit cells, than the S-box S_{PRE} defined by PRESENT is used [ISO 29192-2, BKL+07]. For the other variant with $\mathfrak{s} = 8$, the AES S-box S_{AES} is used as already defined in Sec. 6.6.1 [DR99, RD02]. The PRESENT S-box S_{PRE} is shown in Tab. 6.15.

Then, depending on the parameter $\mathfrak{s}$, SubCells is defined as follows for $i, j \in \mathbb{Z}_{\mathfrak{d}}$:

$$\text{SubCells}_{\mathfrak{s}} : h[i][j] \leftarrow \begin{cases} S_{\text{PRE}}(h[i][j]), & \text{if } \mathfrak{s} = 4 \\ S_{\text{AES}}(h[i][j]), & \text{if } \mathfrak{s} = 8 \end{cases}$$

The ShiftRows function is mostly identical to the AES ShiftRows transformation and can be defined formally as follows:

$$\text{ShiftRow}_{\mathfrak{d}} : h[i][j] \leftarrow h[(i-j) \bmod \mathfrak{d}][j]$$

The MixColumnsSerial is a matrix multiplication using a maximum distance separable (MDS) matrix, i.e. the diffusion property of the linear permutation is maximized. This property is shared with the AES MixColumns permutation. However, for Photon the design is focused on minimum area consumption. In particular, the MixColumnsSerial permutation is designed to be efficiently computable by a serialized implementation computing 4 or 8 bits per clock cycle without intermediate memories. It is in general defined as follows:

$$\text{MixColumnsSerial}_{t,\mathfrak{d}}(h) : h \leftarrow A_t^{\mathfrak{d}} \times h$$

The field used for the matrix multiplication is $\mathbb{F}_{\mathfrak{s}}$. For $\mathfrak{s} = 4$, the irreducible polynomial x^4+x+1 is used and for $\mathfrak{s} = 8$, the AES polynomial $x^8+x^4+x^3+x+1$.

The matrix A_t is defined depending on the state size t, as follows:

$$A_t =_{\text{def}} \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ & \vdots & & & & \vdots & \\ 0 & 0 & 0 & \cdots & 0 & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 1 \\ Z_1 & Z_2 & Z_3 & \cdots & Z_{\mathfrak{d}-3} & Z_{\mathfrak{d}-2} & Z_{\mathfrak{d}-1} \end{pmatrix}$$

And the coefficients Z are defined as follows:

$$Z =_{\text{def}} \begin{cases} (\texttt{1},\texttt{2},\texttt{9},\texttt{9},\texttt{2}), & \text{for } t = 100 \\ (\texttt{1},\texttt{2},\texttt{8},\texttt{5},\texttt{8},\texttt{2}), & \text{for } t = 144 \\ (\texttt{1},\texttt{4},\texttt{6},\texttt{1},\texttt{1},\texttt{6},\texttt{1}), & \text{for } t = 196 \\ (\texttt{2},\texttt{4},\texttt{2},\texttt{B},\texttt{2},\texttt{8},\texttt{5},\texttt{6}), & \text{for } t = 256 \\ (\texttt{2},\texttt{3},\texttt{1},\texttt{2},\texttt{1},\texttt{4}), & \text{for } t = 288 \end{cases}$$

Algorithm 6.18 Photon round function [GPP11]

```
**Require:** $h \in \mathbb{Z}_2^t$, $i \in \mathbb{Z}_{10}$
**Ensure:** $h \leftarrow \text{round}(h, i)$
  $h \leftarrow \text{AddConstants}_{\mathfrak{d}}(h, i)$
  $h \leftarrow \text{SubCells}_{\mathfrak{s}}(h)$
  $h \leftarrow \text{ShiftRows}_{\mathfrak{d}}(h)$
  $h \leftarrow \text{MixColumnsSerial}_{t,\mathfrak{d}}(h)$
  **return** $h$
```

The permutation used to instantiate the sponge function for Photon is a simple repeated execution of the round function (Alg. 6.18) for $n_r = 12$ times.

Iteration Mode The iteration mode is the sponge function as defined in Sec. 3.5.3. However, for Photon-80/20/16, the sponge function is slightly adapted to use a different output rate r', i.e. the squeezing phase uses a rate that is different from the input rate used by the absorption phase. This improves the preimage resistance [GPP11].

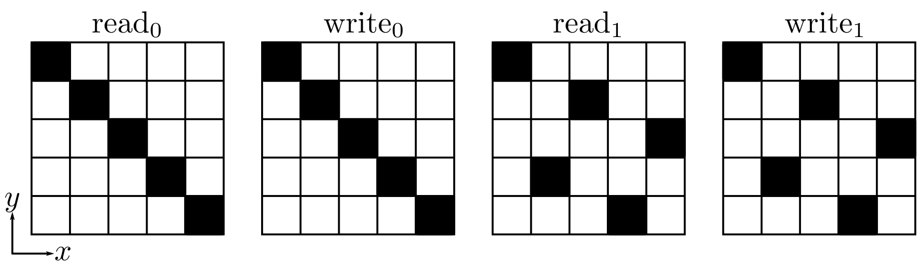


Figure 6.33: Read and write locations for the column-wise Photon architecture.

6.10.2 Systematic Evaluation Overview

State and Memory Organization Similar to Grøstl, Photon's state can be organized in three different patterns:

- The parallel design reads and writes the complete data in each clock cycle. Hence, a $1 \times t$ bit implementation is used. Thus, the data path width d is t bits.
- A column-wise architecture reads and writes one or more column per clock cycle. The simplest one reads exactly one column per clock cycle. Therefore, the memory organization is $\mathfrak{d} \times \mathfrak{d}\mathfrak{s}$ bits. Hence, the data path width is $d = \mathfrak{d}\mathfrak{s}$ bits. However, the ShiftRows transformation makes it necessary to split the memory into $\mathfrak{d}$ smaller RAMs, each $\mathfrak{d} \times \mathfrak{s}$ bits, such that a stall free implementation can read the reordered column in exactly one clock cycle.
- A cell-wise organization uses a $\mathfrak{d}^2 \times \mathfrak{s}$ bits memory. This means reading and writing of a single cell per clock cycle and hence, $d = \mathfrak{s}$.

Since Photon uses a sponge construction as iteration mode, no additional memory is needed and thus a BCM C implementing Photon has $\mathrm{SIZE}_{\mathrm{mem}}(C) \geq \mathfrak{d}^2\mathfrak{s} = t$.

In addition to the RAM, the constants have to be saved in a ROM, in total these need $\mathfrak{d} \times 4$ bits for the internal constants and 12×4 bits for the round constants, i.e. overall $(12 + \mathfrak{d}) \times 4$ bits.

State Read and Write Schedule The scheduling of read and write operations is almost identical to the case of Grøstl. However, there are some differences compared to Grøstl. First, it is not possible to add the stall free

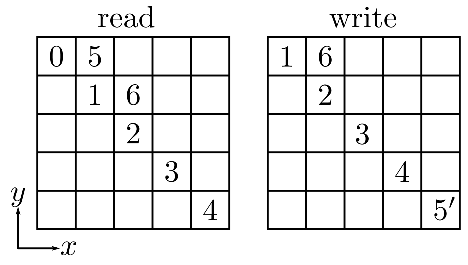


Figure 6.34: Design for the MixColumnsSerial serialization for Photon.

pipelining that enabled the Grøstl architecture to use the x and y coordinates directly. Second, if the MixColumnsSerial permutation is implemented in a serialized fashion, the adjustment for the read addresses has to be changed in the clock cycles when only the MixColumnsSerial permutation is computed.

Beside these changes, the computation of the read and write addresses are very similar. For the parallel version it is trivial for all cases. For the column-wise and cell-wise architectures, the read addresses have to be adapted to the ShiftRows permutation, for all clock cycles when ShiftRows is computed. This means, that the first read is adjusted as shown in Fig. 6.33. However, since no effective pipelining can be used to hide the inter-round data dependencies, a round counter is necessary to adjust the read addresses for the further rounds. For the cell-wise implementation, the same approach has to be used in principle.

For a cell-wise implementation, it may be worthwhile to investigate the serialization possibility for MixColumnsSerial. Because of the special structure of A_t, only one cell changes per column per matrix multiplication with A_t. Therefore, after reading $\mathfrak{d}$ cells from the memory, only one cell has to be changed and the other entries are shifted one row up. This process may be implemented as depicted in Fig. 6.34, i.e. the entry read in the first clock cycle (0) is processed according to the matrix multiplication with A_t and this result written to a temporary register. The next clock cycle loads the value in the cell marked with (1). This value is also processed according to A_t, added to the register and written back to the previous location of (0) and so on. After the value (4) has been loaded, the intermediate value stored in the register is written to the location marked with (5′) and the process continues with the second column.

Note that this procedure is a lot less efficient in terms of clock cycles than the one proposed in the original Photon publication [GPP11]. However, using distributed RAM in FPGAs makes it very difficult to use the proposed architecture, which is much better suited for ASICs.

Clock Cycle Estimation The number of clock cycles can be estimated roughly in the following way. First, the state organization and thus the data path width d has a major influence on the number of clock cycles. However, the Photon design also allows an implementation of the MixColumnsSerial permutation in a serialized way, that is independent of the data path width. For this, the most efficient serialization metric $s = \mathfrak{d}$. Thus the number of clock cycles can be bounded according to the generalized formula defined in 5.4:

$$\mathrm{cyc}_{\mathrm{compress}}(s, t, n_r, d) \geq \frac{s \cdot t \cdot n_r}{d}$$
$$\mathrm{cyc}_{\mathrm{compress}}(s, t, n_r, d) < \frac{s \cdot t \cdot (n_r + 1)}{d}$$

Basic Round Function Architectures There are three basic round function architectures which are dictated by the data path width. The basic architectures are as follows:

- The parallel computation uses a straightforward implementation of the round function.
- The column-wise approach splits the computation into $\mathfrak{d}$ parts. The ShiftRows architecture is implemented by adjusting the read and write addresses as depicted in Fig. 6.33 and described above.
- The cell-wise procedure further splits the computation into $\mathfrak{d}^2$ steps. However, a straightforward implementation of the MixColumnsSerial permutation causes an constant overhead in terms of clock cycles, because of the intra-round dependencies between different cells.

In addition to the described designs, it is also possible to serialize the computation of MixColumnsSerial independent of the reduction of the data path width. However, this approach is only worthwhile for the cell-wise implementation, because reducing the data path increases the number of clock cycles by the same number, but the state RAM may be implemented more efficiently. Thus, the overall performance of the basic versions are probably better.

6.10.3 Detailed Analysis

Parallel Architecture The parallel architecture is very simple. It computes one round in one clock cycle. Therefore only the t bits of memory are needed, no stall-free pipelining is possible, but also no overhead in terms of clock cycles is produced. Also no memory overhead is needed. Hence, for a BCM C, $\mathrm{SIZE}_{\mathrm{mem}}(C) \geq t$.

Serialized Architecture 1 The second architecture analyzed in detail is the column-wise architecture. This design is also straightforward to implement. If the ShiftRows permutation is implemented using the adjusted read and write addresses, smaller versions of the round computation can be re-used from the parallel version, i.e. the AddConstants, SubCells and MixColumnsSerial implementations are implemented only for one column instead of $\mathfrak{d}$ columns. Therefore, the properties memory consumption, pipelining and overhead in terms of clock cycles are identical to the parallel architecture, i.e., for a BCM C, $\mathrm{SIZE}_{\mathrm{mem}}(C) \geq t$.

Serialized Architecture 2 Further serialization leads to a cell-wise architecture. This variant processes only single cells and hence, the AddConstants and SubCells permutations can be again reduced to process only one cell per clock cycle. However, the MixColumnsSerial implementation has to be changed, because it is not possible to further split the computation in the same straightforward way than from the parallel to the column-wise architecture.

In particular, $\mathfrak{d}$ registers of $\mathfrak{s}$ bits are necessary to store the intermediate results of the matrix multiplication for one column. In each clock cycle the intermediate results are stored in these registers. After $\mathfrak{d}$ clock cycles, the first result is produced. This phenomenon also requires another $\mathfrak{d}$ registers, because only $\mathfrak{s}$ bits at a time are written to the state RAM. Hence, the new intermediate values have to be written to another memory.

Thus, because of the MixColumnsSerial architecture, $2\mathfrak{d}\mathfrak{s}$ bits of intermediate memory are needed and $\mathfrak{d} - 1$ additional clock cycles are needed as overhead. Hence, a BCM C takes at least $\mathrm{SIZE}_{\mathrm{mem}}(C) \geq t + 2\mathfrak{d}\mathfrak{s}$.

Serialized Architecture 3 As mentioned above, the cell-wise implementation strategy may be further serialized independent of the data path width reduction. This is possible, because of the specially constructed matrices A_t. The serialization metric is now $s = \mathfrak{d}$. The main benefit of this approach is the reduction of the intermediate memory from $2\mathfrak{d}\mathfrak{s}$ bits to only $\mathfrak{s}$ bits. The number of clock cycles per round increases by an additional factor of $\mathfrak{d}$, but the total overhead stays the same.

A pipeline depth of $\mathfrak{d}^2$ can be implemented, because the serialization of MixColumnsSerial stretches the inter-round dependencies over $\mathfrak{d}^2$ clock cycles, i.e. the input with $x = y = 0$ is only needed $\mathfrak{d}^2$ clock cycles after the same cell was an input to the previous sub-round.

6.10.4 Evaluation Summary

The evaluation summary shows the same values as for the other algorithms. A few interesting observations can be made. Compared to all other algorithms, the throughput for a midrange implementation is bad, even compared to the smallest Keccak-200 implementation. This follows from the low input rate for all Photon variants. For short messages, it is even worse, because the Photon permutation has to be performed multiple times in the squeezing phase. For example, a short message which fits into the r bits after padding leads to $\left\lceil \frac{n}{r'} \right\rceil$ computations of the permutation.

Table 6.16: Summary of Photon analysis

Data Path d [Bits]	Serial. s	Memory [Bits]	ROM [Bits]	Analyzed Pipeline Depth	Clock Cycles	Minimum Overhead	Long Message Throughput @ 100 MHz [MBits/s]	Short Message Throughput @ 100 MHz [MBits/s]
t	1	t	$(12+\mathfrak{d}) \times 4$	1	12	0	$\frac{100\times r}{12}$	$\frac{100\times r}{12} / \left\lceil \frac{n}{r'} \right\rceil$
$t/\mathfrak{d}$	1	t	$(12+\mathfrak{d}) \times 4$	1	$12 \times \mathfrak{d}$	0	$\frac{100\times r}{12\mathfrak{d}}$	$\frac{100\times r}{12\mathfrak{d}} / \left\lceil \frac{n}{r'} \right\rceil$
$t/\mathfrak{d}^2$	1	$t + 2\mathfrak{d}\mathfrak{s}$	$(12+\mathfrak{d}) \times 4$	$\mathfrak{d}$	$12 \times \mathfrak{d}^2$	$\mathfrak{d} - 1$	$\frac{100\times r}{12\mathfrak{d}^2}$	$\frac{100\times r}{12\mathfrak{d}^2} / \left\lceil \frac{n}{r'} \right\rceil$
$t/\mathfrak{d}^2$	$\mathfrak{d}$	$t + \mathfrak{s}$	$(12+\mathfrak{d}) \times 4$	$\mathfrak{d}$	$12 \times \mathfrak{d}^3$	$\mathfrak{d} - 1$	$\frac{100\times r}{12\mathfrak{d}^3}$	$\frac{100\times r}{12\mathfrak{d}^3} / \left\lceil \frac{n}{r'} \right\rceil$
$t/\mathfrak{d}^2$	$\mathfrak{d}$	$2t + \mathfrak{s}$	$(12+\mathfrak{d}) \times 4$	$\mathfrak{d}^2$	$12 \times \mathfrak{d}^3$	$\mathfrak{d}^2 - 1$	$\frac{100\times r}{12\mathfrak{d}^3}$	$\frac{100\times r}{12\mathfrak{d}^3} / \left\lceil \frac{n}{r'} \right\rceil$

6.10.5 Implementation

This new implementation of Photon is pretty simple and straightforward. The design closely follows the analysis of the cell-wise architecture. It consists of the state RAM and the round function (Fig. 6.35). The control logic switches between three modes, the first is the absorption mode, which is used for the absorption of a new message block. The second is the computation of the round function and the third is the generation of the output message.

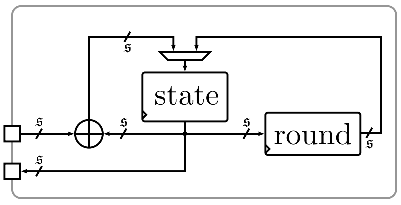


Figure 6.35: Architecture of the Photon implementation.

6.11 Discussion and Further Work

The theoretical results between the different hash functions can also be compared. Interesting is in particular the comparison of parallel implementations as depicted in Tab. 6.17. The table shows the long message throughput of all evaluated hash functions. As can be seen, Grøstl and KECCAK have a significant performance advantage for the straight forward implementation strategy, if only 100 MHz are assumed.

Table 6.17: Long message throughput for parallel implementations.

Algorithm	Long Message Throughput @ 100 MHz [MBits/s]
BLAKE-256	3657
Grøstl-256	5120
JH-256	1219
KECCAK-256	4533
Skein-256	701
Photon-256	266

These results are also roughly reflected in several implementation evaluation results, e.g. [GHR+12b]. For most FPGAs, this is natural, because it is usually possible to reach a frequency above 100 MHz, but because of the limited maximum frequency of FPGAs, it is difficult to reach a clock frequency significantly over $350 - 400$ MHz. Therefore, other techniques such as pipelining combined with parallelization have to be used to increase the throughput, which also increases the area footprint of implementations.

For lightweight implementations, a similar comparison can be made. For these implementations, the data path width plays an important role. However, comparing several architectures from different algorithms with a similar data path width is not meaningful, because the size of the round function implementation becomes the dominant factor. Instead, an evaluator should select architectures with roughly the same throughput.

In the current evaluation one important factor was not analyzed, which additionally influence the maximum throughput of a design in a major way. These include the maximum delay of the implementation strategies, which could be expressed using the theoretical MAXPATH complexity measure. However, this complexity measure is in reality influenced by a lot of factors. First, it is difficult to find a circuit of minimal depth. Second, the real world clock frequency depends a lot more on the actual technology, because the basis of a corresponding theoretical Boolean circuit model changes. This could be abstracted using the big-O notation. However, this notation is too coarse to be of much use in this application domain. Therefore, another alternative notation should be developed, which is less abstract than the big-O notation.

Chapter 7

Implementation Evaluation

7.1 Introduction

In the evaluation, key performance metrics of the implemented and compiled designs are compared. For this goal, all earlier described implementations were optimized using the Xilinx ISE tool chain. This consists of running the synthesis (XST), mapping (MAP) and place-and-route (PAR) tools. Since the implementations use no special primitives, such as BRAM or DSP slices, the results can be compared in a relatively fair manner.

The core metrics used in this evaluation are the area, the throughput and the throughput-area ratio. All three metrics are important for different application areas. The area aspect is most important for lightweight implementations. However, the smallest FPGAs are rather large and expensive compared to many ASIC such as RFID tags, and hence the area plays a minor role compared to other measures, especially the throughput-area ratio.

Reaching the throughput required for an application is one of the key requirements. For example, a high-speed router has to process data in the order of 1 GiB/s [FPO05], other applications like Car-2-X only need the capability to process little over 1 MiB/s [SGI+11]. Nevertheless, if an architecture is fast enough for a particular application, shrinking the area improves the cost effectiveness of the total implementation, if it is possible to use a smaller FPGA for the same use case. Therefore, again the throughput-area ratio is very important.

Beside the comparison of the metrics between two or more implementations, it is interesting to see how scalable an architecture is. For example, it is possible, that a parallel implementation of one algorithm has a higher throughput and throughput-area ratio compared to another algorithm and that the situation changes for serialized implementations. However, only for Keccak more than one implementation was developed, the scalability argument is not investigated in detail.

The remainder of the evaluation chapter is organized as follows. First the criteria of the evaluation are described in more detail in Sec. 7.2. Then the automated optimization approach is introduced in Sec. 7.3. Afterwards, the results for the implementations presented in this thesis are discussed for Virtex-5 FPGAs (Sec. 7.4) followed by currently known third party results for the same FPGA (Sec. 7.5). Further results for other Xilinx FPGAs can be found in Appendix B.

7.2 Criteria

Several criteria are interesting for the evaluation of FPGA implementations, foremost the area, the throughput and the throughput-are ratio. Another interesting criteria is the power consumption. However, this metric is not evaluated in this thesis. In this section, the measures that are used to score the evaluations are detailed and a short rationale is provided for each choice.

- The *area* of FPGA implementations is usually measured in the total number of slices required for an implementation. This criterion is sometimes not that useful, e.g. because of the usage of BRAM or DSP slices. The main problem with the usage of special circuitry of an FPGA is, that they hide a lot of the complexity of an algorithm. For example, BRAM may not only implement memory, but also parts of the logic.

  Therefore, the results generated for the developed algorithms these special primitives are not used and only slices of the Xilinx FPGAs are used.

- The *throughput* is measured in MBit/s. For hash functions, the throughput changes with the message length. For most hash functions longer messages lead to a higher throughput and short messages to a lower throughput.

  This has two root causes. The first is the padding, which fills at least one message block completely. Hence, if the message is shorter, a lot of padding bits are processed and less message bits. Therefore, the throughput is reduced. Furthermore, many hash functions have a post processing transformation. For example, the sponge construction requires additional computations of the permutation in the squeezing phase, if the rate r is smaller than the message digest length n.

Thus, for the current evaluation, the throughput is evaluated in two parts. The first part evaluates the throughput for very long messages. There, the number of clock cycles to process the compression function once for each message block is the dominating influence on the throughput. The second part evaluates the throughput for short messages that have a length of exactly one message block.

- The throughput-area is a measure derived from the area and the throughput. Therefore, this measure is also split into two values. One for long messages and another one for short messages.

7.3 Automated Optimization

Depending on the exact parameters used with the tool chain, the synthesis results vary in throughput, and area consumption. Therefore, for a fair evaluation, it is necessary to try a lot of different options to achieve the best result. The (simplified) process of the Xilinx ISE 14.5 tool chain is as follows:

1. Synthesis with XST.
2. Mapping with MAP.
3. Place and Route with PAR.

The first two steps have a lot of parameters and because many different choices may be combined, the set of possible options grows exponentially. Fortunately, there are only a few options that have a significant influence on the results. However, for each FPGA architecture this changes slightly.

The optimization follows a similar concept to the one developed in ATHENa developed by the GMU [GKA+10]. However, it uses a slightly different strategy. First a set of parameters to iterate over is selected. After this selection for a specific FPGA architecture, the optimization commences as follows:

1. The implementation process is run for the all possible combinations from the chosen set, e.g. for the Virtex-5 platform 384 possibilities are used.
2. The best eight candidates for each category area, throughput and throughput-area are selected.
3. For each selected parameter choice the automatic timing result is the starting point for a timing exploration, i.e. the timing of the performance evaluation mode is used as a timing constraint for a new synthesis run. This results in a new timing estimation, which can be used again as a new constraint. This process continues until the timing constraint is too tight and PAR is unable to route a design.
4. The next step is to try all 100 possible seeds for the best candidates for the same categories area, throughput and throughput-area, because the probability is high, that PAR may be able to route the design using the tight timing constraint with a different seed.
5. After completion of the run for the final seed, the best parameter sets for the three categories are determined.

7.4 Implementation Results

The implementation results are split in three parts, depending on the parameters and the hardware interface.

- The first presents the results for the SHA-3 finalists only by comparing the implementations that use the FSL-based interface (Sec. 6.3.1) and include the padding unit.
- The second part presents alternative implementations of Keccak-f[1600] and also Keccak-f[800]. These implementations use the GMU-based interface (Sec. 6.3.2) and do not include the padding in hardware.
- The last part displays the results for the lightweight variants of Keccak, i.e. Keccak-f[400] and Keccak-f[200] and also of Photon. The Keccak variants use the GMU interface (Sec. 6.3.2) and the Photon implementations the adapted version for lightweight hashing (Tab. 6.4).

Table 7.1: Results for the 256 bits versions of the SHA-3 finalists (Virtex-5).

Name	Area [Slices]	Frequency [MHz]	Clock Cycles	Input size [Bits]	Long TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	Short TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
BLAKE	261	205	228	512	460	1.76	460	1.76
BLAKE-2	388	154	115	512	684	1.76	684	1.76
Grøstl	352	298	160	512	954	2.71	477	1.36
JH	190	315	6720	512	24	0.13	12	0.07
JH-2	388	270	168	512	823	2.12	411	1.06
KECCAK	380	168	200	1088	916	2.41	916	2.41
Skein	502	283	584	512	248	0.49	124	0.25

7.4.1 SHA-3 Finalists with Padding

The implementations of the SHA-3 finalists that were implemented, can be grouped into four sets (Fig. 7.1,Tab. 7.1).

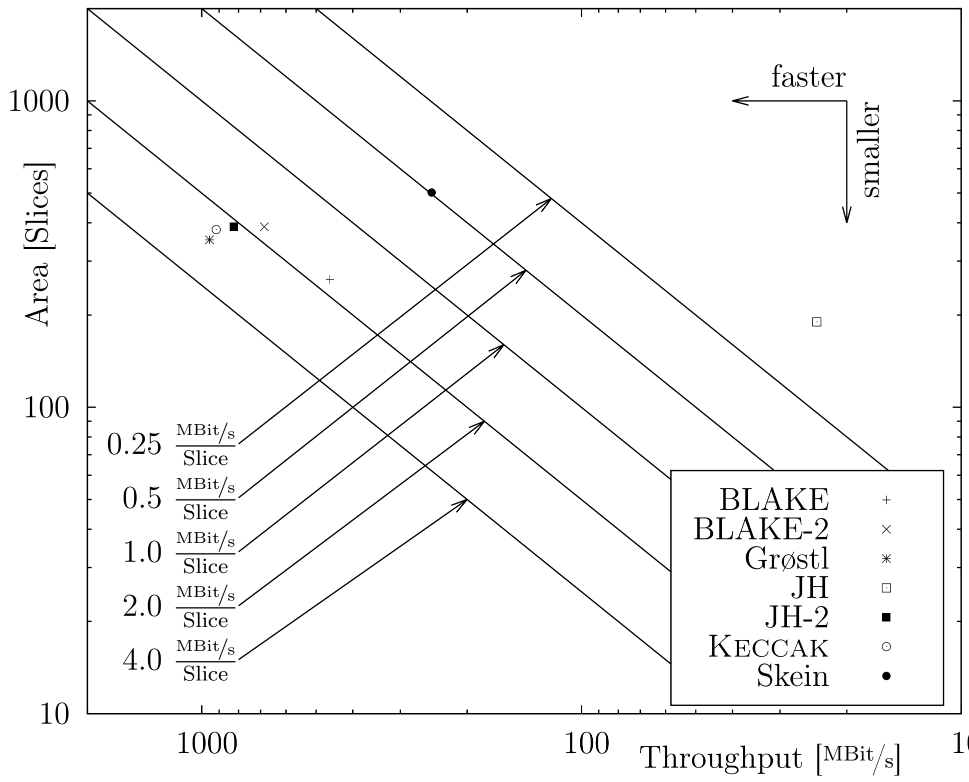


Figure 7.1: Area and throughput of the SHA-3 finalist implementations (Virtex-5).

- For all algorithms, except Skein, there is an implementation, with very similar throughput and area properties. The best algorithm in this cluster in terms of throughput, area and throughput area ratio is Grøstl. However, all other algorithms are close.
- The first BLAKE implementation has practically the same throughput-area ratio compared to its larger sibling, while consuming only approximately $^2/_3$ of its area.
- Skein is the candidate which trails behind the other candidates. Neither the throughput, nor the area is competitive.
- The smallest JH implementation is considerably smaller than all other implementations and serves as an extreme reference point. It consumes only about 50% of the larger JH implementation. However, the implementation is also the slowest.

Skein is slow, because of the very high number of rounds compared to all other algorithms. To keep up with the other algorithms, a complicated pipelined design is used and the serialization is not very effective. Therefore, Skein is also quite large.

The situation changes slightly for short messages (Tab. 7.1), because the number of clock cycles doubles per message block for some algorithms, if only one message block is processed. Therefore, the throughput is halved for Grøstl, JH and Skein, whereas for BLAKE and Keccak the throughput does not drop for small messages in such a significant way. Hence, the best algorithm for small messages is now Keccak and Grøstl falls back on place three.

7.4.2 Heavyweight Keccak without Padding

All of the analyzed heavyweight versions of Keccak produce a hash digest with 256 bits. However, for Keccak-$f[800]$ two variants with different capacities and hence different security levels are analyzed. Note additionally, that for these Keccak variants, the throughput for short messages is not significantly lower than for long messages, because the rate is always greater than the digest size. Therefore, the additional discussion of this aspect is dropped here.

The implementations of the fast and heavyweight versions scale practically in the same way for the different variants and thus show, that the slice-oriented architecture for Keccak is a very flexible and scalable architecture for mid-range and small implementation (Fig. 7.2, Tab. 7.2). For example, for the official SHA-3 variant of Keccak, the smallest design needs only 11% of the area compared to the fully parallel version with the same basic strategy.

The scalability can be analyzed further:

- The architecture does not scale in a linear fashion in terms of area, because at least the state has to be stored and the control logic is roughly the same for all implementations with the same state size and capacity.
- The clock frequency is higher for the largest version than for most of the serialized variants. This phenomenon can be explained by the different implementation strategy for the state RAM. In the serialized implementa-

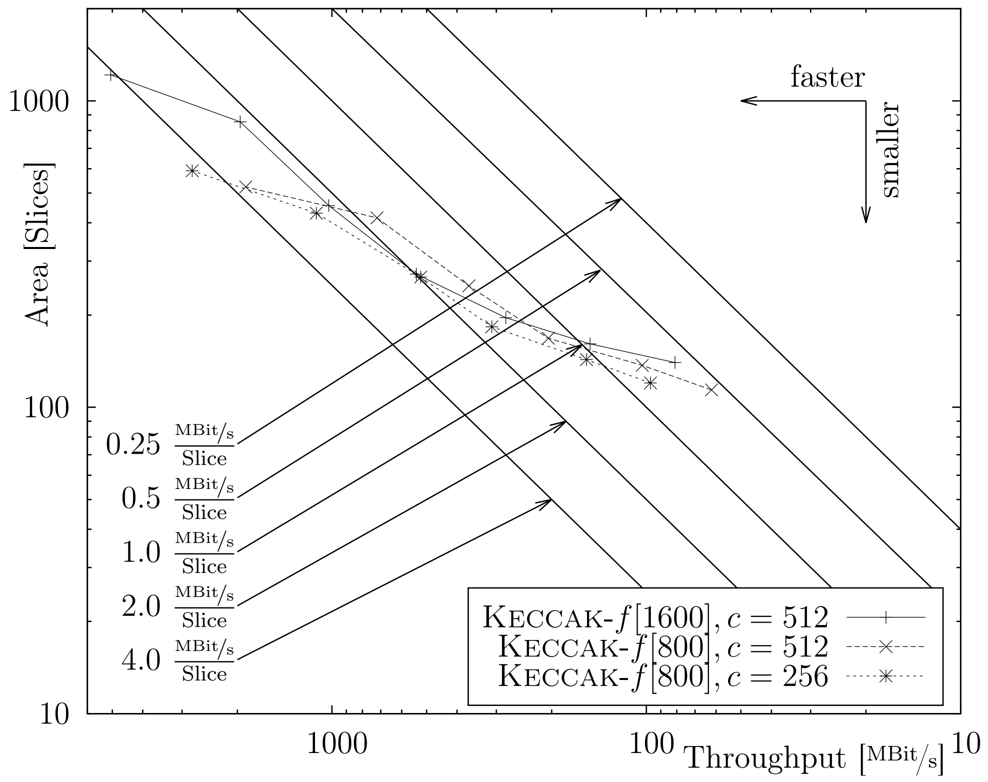


Figure 7.2: Area and throughput of Keccak with $b \in \{1600, 800\}$ and $c = 512$ for $b = 1600$ and $c \in \{512, 256\}$ for $b = 800$ (Virtex-5).

Table 7.2: Results for the heavyweight versions of Keccak (Virtex-5).

Name	Capacity [Bits]	Parallel Slices	Area [Slices]	Frequency [MHz]	Clock Cycles	Input size [Bits]	Long TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
Keccak-f[1600]	512	1	140	200	2688	1088	81	0.58
		2	161	186	1344	1088	151	0.93
		4	196	173	672	1088	280	1.43
		8	272	166	336	1088	539	1.98
		16	455	158	168	1088	1024	2.25
		32	854	151	84	1088	1959	2.29
		64	1215	195	42	1088	5055	4.16
Keccak-f[800]	512	1	114	220	1024	288	62	0.54
		2	137	184	512	288	103	0.75
		4	168	182	256	288	205	1.22
		8	249	163	128	288	367	1.47
		16	415	160	64	288	719	1.73
		32	524	209	32	288	1880	3.59
	256	1	120	228	1280	544	97	0.81
		2	143	182	640	544	155	1.08
		4	183	182	320	544	310	1.69
		8	266	153	160	544	522	1.96
		16	430	165	80	544	1121	2.61
		32	591	205	40	544	2785	4.71

tions, the state is always stored in distributed RAM, whereas the parallel version uses registers. The access to registers has a shorter critical path, because less logic is involved and hence, the clock frequency can be higher (Sec. 2.4).

- The maximum clock frequencies increase for the versions processing less slices in parallel, because the overall area consumption is less and therefore, the Xilinx tool chain can better optimize the design. Hence, the critical path typically becomes shorter.

7.4.3 Lightweight Hash Functions without Padding

In this thesis, several lightweight variants of Keccak are compared with an implementation of the Photon hash function. For both algorithms not the most compact implementation possibilities are taken into consideration, because typical usages of FPGAs are not area limited such as ASIC applications such as

RFID tags. Therefore, FPGA implementations may often trade area savings for additional throughput, if the area does not increase significantly. Furthermore, many optimizations that are valid for ASICs may also be counter-productive for FPGAs because it is well known, that replicating parts of a circuit may even reduce the area for FPGAs [FS94, MBV06] (Sec. 4.3).

To facilitate a roughly fair comparisons between different variants of Keccak-$f[400]$ and Photon, similar security parameters are used. In particular, Photon specifies five variants with $256, 224, 160, 128$, and 80 bit message digests. The security parameter c is equal to the size of the message digest for each variant. Except for $n = c = 80$, the same values for the parameters n, and c are

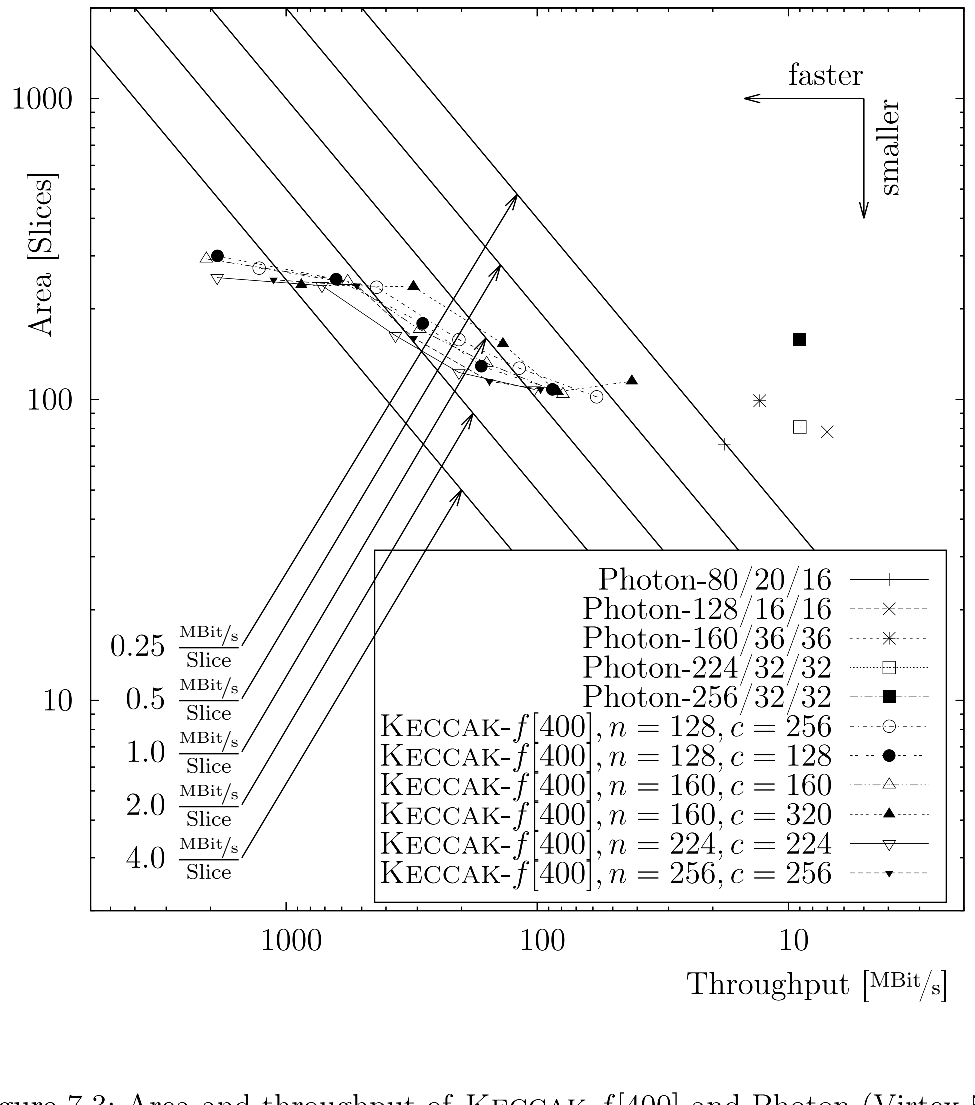


Figure 7.3: Area and throughput of Keccak-$f[400]$ and Photon (Virtex-5).

used for KECCAK-$f[400]$ in addition to variants with higher pre-image security for $n \in \{160, 128\}$. For KECCAK-$f[200]$, only the variants with $n = c = 128$ and $n = c = 160$ can be implemented.

The results in Tab. 7.3, and Tab. 7.4 as well as Fig. 7.3 and Fig. 7.4 show, that except for the variant with 256 bit message digest, all Photon variants are smaller compared to any of the KECCAK-$f[400]$ implementations. This changes for KECCAK-$f[200]$, where the smallest variants are direct competitors to the Photon variants with the same security parameters. In contrast to the area results, the throughput of KECCAK is much higher than for Photon, which is extremely slow. This can be explained with the very low input rate of Photon

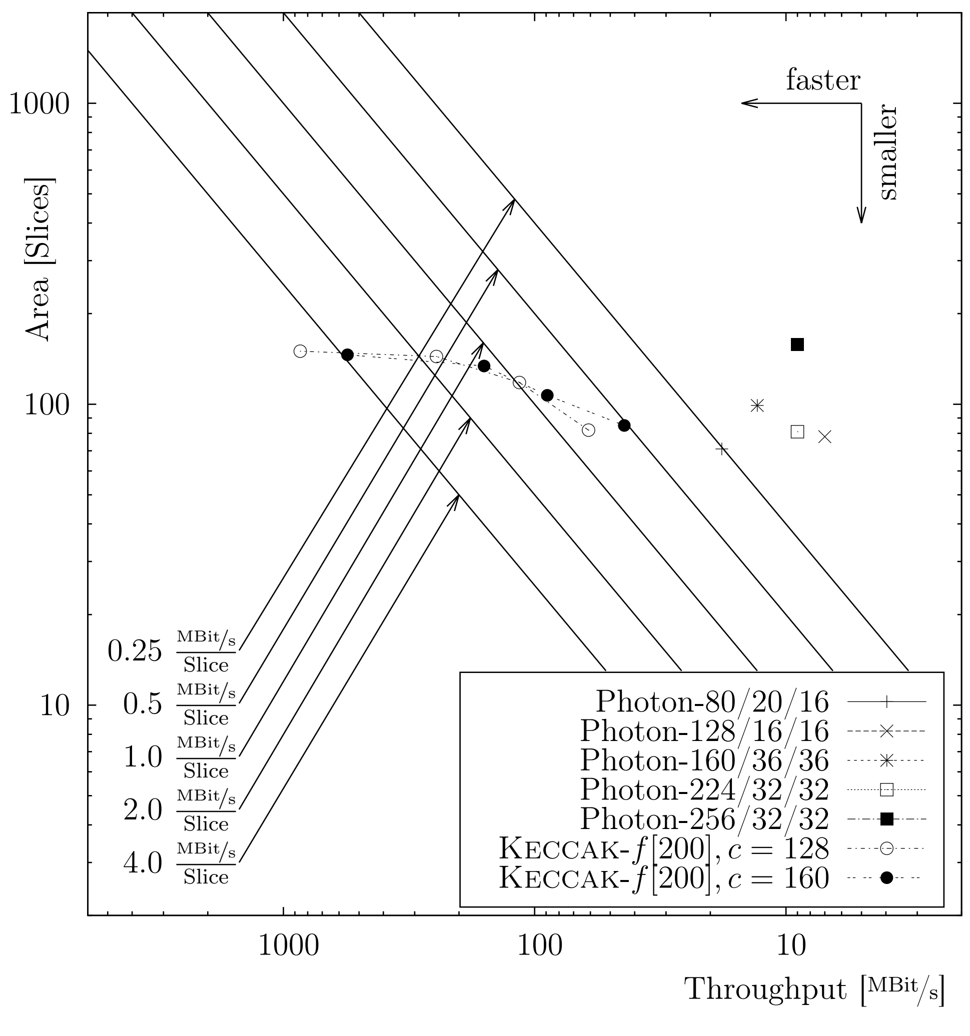


Figure 7.4: Area and throughput of KECCAK-$f[200]$ and Photon (Virtex-5).

Table 7.3: Results for the lightweight versions of Keccak (Virtex-5).

Name	Digest	Capacity	Parallel	Area	Frequency	Clock	Input size	Long TP	TP-area	Short TP	TP-area
	[Bits]	[Bits]	Slices	[Slices]	[MHz]	Cycles	[Bits]	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
Keccak-f[400]	128	256	1	102	194	480	144	58	0.57	58	0.57
			2	127	197	240	144	118	0.93	118	0.93
			4	158	171	120	144	205	1.30	205	1.30
			8	236	181	60	144	436	1.85	436	1.85
			16	273	267	30	144	1280	4.69	1280	4.69
	128	128	1	108	195	608	272	87	0.81	87	0.81
			2	129	187	304	272	167	1.30	167	1.30
			4	179	160	152	272	286	1.60	286	1.60
			8	251	177	76	272	632	2.52	632	2.52
			16	300	262	38	272	1877	6.26	1877	6.26
	160	160	1	104	189	576	240	79	0.76	79	0.76
			2	132	191	288	240	159	1.21	159	1.21
			4	171	176	144	240	294	1.72	294	1.72
			8	247	170	72	240	568	2.30	568	2.30
			16	293	312	36	240	2077	7.09	2077	7.09
	160	320	1	115	218	416	80	42	0.36	21	0.18
			2	106	216	208	80	83	0.79	42	0.39
			4	153	179	104	80	137	0.90	69	0.45
			8	237	202	52	80	311	1.31	155	0.66
			16	240	283	26	80	869	3.62	435	1.81
	224	224	1	108	184	512	176	63	0.58	32	0.29
			2	123	182	256	176	125	1.02	63	0.51
			4	163	163	128	176	224	1.38	112	0.69
			8	239	160	64	176	439	1.84	220	0.92
			16	254	209	32	176	1149	4.52	574	2.26
	256	256	1	108	228	480	144	68	0.63	34	0.32
			2	115	182	240	144	109	0.95	55	0.48
			4	160	182	120	144	219	1.37	109	0.68
			8	239	153	60	144	368	1.54	184	0.77
			16	250	165	30	144	791	3.17	396	1.58
Keccak-f[200]	128	128	1	82	190	224	72	61	0.74	31	0.37
			2	118	179	112	72	115	0.98	58	0.49
			4	144	191	56	72	246	1.71	123	0.85
			8	150	333	28	72	857	5.71	429	2.86
	160	160	1	85	211	192	40	44	0.52	11	0.13
			2	107	214	96	40	89	0.83	22	0.21
			4	134	190	48	40	159	1.18	40	0.30
			8	146	333	24	40	556	3.81	139	0.95

and furthermore, the high number of clock cycles.

The Keccak-f[400] and Keccak-f[200] results scale very well, again, similar to the previous heavyweight evaluation. However, since the control logic and the state dominates the area for almost all designs, the area reduction from the fastest to the smallest version is less significant than for the SHA-3 candidate version.

The Photon results are almost as expected. However for Photon-224/32/32,

Table 7.4: Results for the Photon hash function family (Virtex-5).

Name	Area [Slices]	Frequency [MHz]	Clock Cycles	Input size [Bits]	Long TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	Short TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
Photon-80/20/16	71	220	245	20	18	0.25	3.60	0.051
Photon-128/10/16	78	196	436	16	7	0.09	0.90	0.012
Photon-160/36/36	99	219	597	36	13	0.13	2.65	0.027
Photon-224/32/32	81	214	776	32	9	0.11	1.26	0.016
Photon-256/32/32	158	126	436	32	9	0.06	1.15	0.007

the area usage is significantly below Photon-160/36/36. This can be explained with the parameter d of Photon, because for Photon-224/32/32 $d = 8$ is used and thus, the management of counters becomes much easier, because it is a power of 2. Hence, the Xilinx tool chain is able to better optimize some parts of the design and the area decreases.

7.5 Third-Party Implementations

For the comparisons with third-party results, only results for implementations of the third round specifications are used. This is fair, because for some algorithms, changes were made that influence the performance in a significant way. For example the throughput of BLAKE reduces, because of an increased number of rounds. The third round version of JH suffers the same fate. Furthermore, Grøstl tweaked the round function considerably and therefore, the performance also changes. The decision to exclude all other results is sometimes unfortunate, because otherwise excellent compact implementations like the one from Beuchat et al. [BOY10] are also not shown.

From the many different possibilities, three comparative studies highlight the relation of the results of this thesis and the results in the literature [KDV+11, GHR+12b, KYS+11] (Tab. 7.5). Note that the results of [KDV+11] are only provided as a reference point in this table, because they only presented results for Virtex-6 FPGAs. The chosen results highlight the high end implementations in terms of the throughput-area ratio and the lightweight end with small implementations.

The comparison shows mixed results. For the high-throughput variants Keccak excels all other candidates, followed by JH and Grøstl and then Skein.

Table 7.5: Third-Party Results for the 256 bits versions of the SHA-3 finalists.

Name	Reference	FPGA	Area [Slices]	Frequency [MHz]	Clock Cycles	Input size [Bits]	Long TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
BLAKE	This work	Virtex-5	261	205	228	512	460	1.76
	[KDV+11]	Virtex-6	117	274	1182	512	105	0.90
		Virtex-6	175	347	1182	512	132	0.75
	[GHR+12b]	Virtex-5	231	-	-	512	389	1.69
		Virtex-5	3495	-	-	512	7547	2.16
	[KYS+11]	Virtex-5	271	253	290	512	448	1.65
Grøstl	This work	Virtex-5	352	298	160	512	954	2.71
	[KDV+11]	Virtex-6	260	280	176	512	815	3.13
		Virtex-6	293	330	176	512	960	3.27
	[GHR+12b]	Virtex-5	981	-	-	512	951	0.97
		Virtex-5	4177	-	-	512	16353	3.91
	[KYS+11]	Virtex-5	313	317	547	512	417	1.33
JH	This work	Virtex-5	388	270	168	512	823	2.12
	[KDV+11]	Virtex-6	240	288	688	512	214	0.89
		Virtex-6	304	299	688	512	222	0.73
	[GHR+12b]	Virtex-5	306	-	-	512	138	0.45
		Virtex-5	982	-	-	512	4955	5.05
	[KYS+11]	Virtex-5	271	250	800	512	160	0.88
KECCAK	This work	Virtex-5	380	168	200	1088	916	2.41
	This work	Virtex-5	272	166	336	1088	539	1.98
	This work	Virtex-5	161	186	1344	1088	151	0.93
	This work	Virtex-5	140	200	2688	1088	81	0.58
	[KDV+11]	Virtex-6	144	250	2137	1088	128	0.89
		Virtex-6	188	285	2137	1088	145	0.77
	[GHR+12b]	Virtex-5	354	-	-	1088	855	2.41
		Virtex-5	1369	-	-	1088	13337	9.74
	[KYS+11]	Virtex-5	275	259	2396	1088	118	0.43
Skein	This work	Virtex-5	502	283	584	512	248	0.49
	[KDV+11]	Virtex-6	240	160	230	256	179	0.75
		Virtex-6	291	200	230	256	223	0.77
	[GHR+12b]	Virtex-5	1025	-	-	512	1179	1.15
		Virtex-5	1858	-	-	512	5338	3.87
	[KYS+11]	Virtex-5	246	176	2398	512	37	0.15

However, for midrange and lightweight implementations this picture changes considerably. The winner is now Grøstl, which scales very good. The overall area consumption is still pretty high. The next three candidates are KECCAK, JH, and BLAKE, for a similar area consumption and Skein is the loser, which trails far behind.

The results presented in this thesis are based on the studies presented in [JA11, Jun12] and the recent paper [JS13]. At that time, the results on KECCAK and JH were improving the state of the art significantly by providing reasonably small implementations with much better throughput for both algorithms. Especially for KECCAK it was believed, that it is impossible to implement a midrange implementation with a high throughput for FPGAs. Another improvement for lightweight to midrange implementations was developed in [JS13], scaling the previously developed architecture in both directions lightweight and high-throughput.

7.6 Discussion and Further Work

From the practical as well as the previous theoretical evaluation, it can be seen, that the specification of the KECCAK hash function is a very versatile hash function for hardware implementations. It is possible to choose parameters for high-throughput applications as well as for very lightweight applications. Extending the specifications of the other SHA-3 finalists to be more scalable towards lightweight applications seems to be in theory feasible. However, the security of such variants would have to be analyzed in detail first.

The Photon hash function is specially designed for low-area implementations. Thus, the designers did not emphasize the possibility of high-throughput implementations. This can be seen in the theoretical evaluation in the previous chapter as well as in the implementation results. It would be possible to extend the Photon hash function to use a larger state and thus, to be able to use a higher rate. Again, the security of such constructions would have to be analyzed in detail, before they are used.

In further work, a stronger connection between the theoretical and the practical evaluations could be developed. A first step would be to study the effects of the serialization and the static resource consumption of the state and the control logic, e.g. the correlation between the theoretical and the practical results can be analyzed. Furthermore, the theoretical approach could be extended to estimate the clock frequency, as already mentioned in the discussion in Ch. 6 and additionally, a better estimate on the area consumption could be developed. A starting point for this extension and further improvement of the theoretical tools for the performance modeling would be to implement more of the discussed architectures and then to analyze the results and to correlate them to the theoretical evaluation results.

From a practical point of view, it is also possible to extend the evaluation methodology for other cryptographic functions. For example, the approach could help to better understand the architectural properties of the authenticated encryption algorithms proposed for the CAESAR competition [Ber14].

Part IV

Appendix

Appendix A

Finite Fields

A.1 Introduction

In this chapter, an introduction to the theory of finite fields is provided. It is limited to the essentials which are relevant for the discussion of the optimizations of the AES S-box in Sec. 6.6.5. The theory is also helpful to better understand several aspects of the evaluated hash algorithms, because they are in parts defined using finite fields. The standard reference for finite fields used in this thesis is [LN96]. Additional references for the theory of finite fields are for example [KM09, How06]. References to the individual definitions, lemmas, theorems and corollaries are given, if applicable. References to publications describing algorithms are also provided as necessary.

This introduction covers the necessary theoretical tools to build so called composite field representations [Paa94, Can05a, Can05b]. These composite field representations are based upon the concept of extension fields and are used to construct efficient implementations for expensive computations from several smaller and simpler parts. This different representation often leads to hardware implementations with less area consumption. The decomposition into smaller subproblems may also be accompanied by changing the basis of the finite field representation. This can be useful, because for finite fields different bases may lead to different trade-offs in terms of implementation efficiency. For example, squaring using a normal basis representation is almost free of any cost, because it is just a bit-wise rotation. Hence, in hardware it is very efficient to implement.

The chapter is organized as follows. First, the definitions of rings, commutative rings and finally finite fields are introduced (Sec. A.2). These definitions are then used to present core results of finite field theory (Sec. A.3), followed by the definition of polynomial and normal bases and the corresponding arithmetic operations addition and multiplication (Sec. A.4). Afterwards, the composite field approach is explained in Sec. A.5. It is accompanied by the definitions of all algorithms needed to construct the evaluated representations and to convert elements from one representation to another (Sec. A.6).

A.2 Basic Definitions

Definition A.1 (Def. 1.28 [LN96]) *A* ring *with identity is a triple* $\mathbb{R} = (R, \oplus, \otimes)$*, where* R *is a non-empty set of elements,* $\oplus$ *and* $\otimes$ *are the operations, usually called addition and multiplication. The following properties hold, for* $a, b, c, 0, 1 \in R$*:*

1. *Associativity for addition:*

$$(a \oplus b) \oplus c = a \oplus (b \oplus c)$$

2. *Commutativity for addition:*

$$a \oplus b = b \oplus a$$

3. *Existence of* 0*:*

$$a \oplus 0 = a$$

4. *Existence of additive inverse (negative):*

$$a \oplus (-a) = 0$$

5. *Associativity for multiplication:*

$$(a \otimes b) \otimes c = a \otimes (b \otimes c)$$

6. *Distributivity:*

$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$

$$(a \otimes b) \oplus c = (a \oplus c) \otimes (b \oplus c)$$

7. *Existence of* 1*:*

$$(a \otimes 1) = (1 \otimes a) = a$$

Note that in the previous definition the last property is not mandatory for the definition of a ring. However, since it is not important for the present work to distinguish between the two cases of a ring with identity and without, only the first case is defined.

Definition A.2 (Def. 1.29 [LN96]) *A* commutative ring *is a ring* $\mathbb{R} = (R, \oplus, \otimes)$ *with the following additional property, for* $a, b \in R$*:*

8. *Commutativity for multiplication:*

$$a \otimes b = b \otimes a$$

Definition A.3 (Def. 1.29 [LN96]) *A* (finite) field *is a commutative ring* $\mathbb{F} = (F, \oplus, \otimes)$*, where* F *is a (finite) non-empty set and the following additional property holds, for* $a \in F \setminus \{0\}$*.*

9. *Existence of multiplicative inverse:*

$$(a \otimes a^{-1}) = 1$$

A.3 Basic Results

Examples of commutative rings are the well known algebras $\mathbb{Z}, \mathbb{Q}$, $\mathbb{R}$ and $\mathbb{Q}$. They are obviously not finite fields. However, any number of finite fields can be easily constructed. A first step to these finite fields are residue class rings over $\mathbb{Z}$. In these rings, denoted as $\mathbb{Z}_n$[1], the arithmetic is performed modulo some $n \in \mathbb{N}$.

Definition A.4 (Def. 1.36 [LN96]) *An algebra* $(\mathbb{Z}_n, \oplus, \otimes)$ *with* $\mathbb{Z}_n =_{\text{def}} \{[a]_{\equiv_n} : 0 \leq a < n\}$ *is called* residue class ring. *Its elements are residue classes, which are defined as equivalence classes modulo* n*, i.e. the residue class* $[a]_{\equiv_n}$ *is defined as* $[a]_{\equiv_n} =_{\text{def}} \{x \in \mathbb{Z} : a \equiv x \mod n\}$*.*

[1]In the present work the $\mathbb{Z}_n$ notation is preferred over the alternative more verbose $\mathbb{Z}/n\mathbb{Z}$. Both notations are otherwise describing the same algebraic structure.

An algebra $\mathbb{Z}_n$ inherits some of its basic properties from $\mathbb{Z}$, which is useful to prove that for all $n > 1$, $\mathbb{Z}_n$ is indeed a commutative ring:

Theorem A.5 (Ex. 1.37 [LN96]) *Let $n \in \mathbb{N}, n > 1$, then the algebra $(\mathbb{Z}_n, \oplus, \otimes)$ is a commutative ring.*

Proof For two residue classes $[a]_{\equiv_n}$ and $[b]_{\equiv_n}$, addition and multiplication can be defined by the homomorphisms $[a+b]_{\equiv_n} = [a]_{\equiv_n} \oplus [b]_{\equiv_n}$ and $[ab]_{\equiv_n} = [a]_{\equiv_n} \otimes [b]_{\equiv_n}$. Thus, the above defined properties 1 to 8 are inherited from $\mathbb{Z}$ for all $n \in \mathbb{Z}$. $\square$

A special case is a residue class ring $\mathbb{Z}_p$, when p is prime. This is formalized in the following theorem.

Theorem A.6 (Thm. 1.38 [LN96]) *A residue class ring $\mathbb{Z}_p$ is a finite field iff p is prime.*

Proof By Thm. A.5, it is sufficient to show, that property 9 from Def. A.3 holds (Existence of a multiplicative inverse).

Let a, b be two arbitrary elements $a, b \in \mathbb{Z}_p \setminus \{[0]_{\equiv_p}\}$. Then $ab = [0]_{\equiv_p}$ if and only if $p|ab$. However, since p is prime $p|ab$ if and only if $p|a$ or $p|b$ and thus, $a = [0]_{\equiv_p}$ or $b = [0]_{\equiv_p}$ which is a contradiction to the assumption $a, b \in \mathbb{Z}_p \setminus \{[0]_{\equiv_p}\}$. Hence, $\mathbb{Z}_p$ has no zero divisors, i.e. for any $a \in \mathbb{Z}_p \setminus \{[0]_{\equiv_p}\}$, there is no $b \in \mathbb{Z}_p \setminus \{[0]_{\equiv_p}\}$ such that $ab = [0]_{\equiv_p}$.

Furthermore, let $a_0, \ldots, a_{p-1} \in \mathbb{Z}_p$ be the elements of $\mathbb{Z}_p$ and $a \in \mathbb{Z}_p \setminus \{[0]_{\equiv_p}\}$. Then, for two elements $a_i, a_j \in \mathbb{Z}_p$, one has $aa_i = aa_j$, if $a(a_i - a_j) = [0]_{\equiv_p}$ and because $\mathbb{Z}_p$ contains no zero divisors, one has $a_i = a_j$. Therefore, all elements of the form $aa_i \in \mathbb{Z}_p$ are distinct.

In particular, for all $a \in \mathbb{Z}_p \setminus \{[0]_{\equiv_p}\}$, there exists an element $b \in \mathbb{Z}_p \setminus \{[0]_{\equiv_p}\}$, such that $ab = [1]_{\equiv_p}$, i.e. b is the multiplicative inverse of a. $\square$

The finite field $\mathbb{Z}_2$ has a special importance, because many cryptographic algorithms are defined over this field for efficiency reasons. In particular $\mathbb{Z}_2$ is interesting because the field operations correspond directly to binary Boolean gates ($\oplus = XOR$, $\otimes = AND$) and thus, the operations can be efficiently implemented in hardware. It is isomorphic to the finite field $\mathbb{F}_2$, which is often also denoted as $GF(2)$, where GF is an abbreviation for Galois field, named after Évariste Galois, who pioneered the field of Galois theory. In the following chapters the $\mathbb{F}_2$ notation will be used in cases, where properties of the finite fields are important.

The importance of $\mathbb{F}_2$ is generalized in finite fields of characteristic 2. The characteristic of a ring in the general case is defined as follows:

Definition A.7 (Def. 1.43 [LN96]) *Let R be a ring and $c > 0$ such that for every $r \in R$, $cr = 0$ is true. Then, the least such c is called the* characteristic *of the ring R. If no such $c > 0$ exists, then $c = 0$.*

For finite fields, this definition results in the following theorem:

Theorem A.8 (Cor. 1.45 [LN96]) *The characteristic of a finite field $\mathbb{F}$ is always prime.*

Proof Let $0, 1 \in \mathbb{F}$. Then the finite field $\mathbb{F}$ has a positive characteristic $c \geq 0$, because it has only finitely many elements and thus, there are always two natural numbers $1 \leq k < l$ such that $k \cdot 1 = l \cdot 1$ or $(l - k) \cdot 1 = 0$. Thus, because $k \neq l$ one has $c = l - k > 0$.

Furthermore, the characteristic c is prime, because if c is not prime, then for two natural numbers $m, n < c$, it is possible to write $c \cdot 1 = mn \cdot 1 = (m \cdot 1)(n \cdot 1) = 0$ and thus, one has also $(m \cdot 1) = 0$ or $(n \cdot 1) = 0$. This is a contradiction to Def. A.7 and hence, c is prime. □

Example A.9 $\mathbb{F}_2$ *is easily seen to be of characteristic* 2*. If* $0, 1 \in \mathbb{F}_2$*, then* $2 \cdot 1 = 1 \oplus 1 = 0$.

An extension to the concept of rings of the form $\mathbb{Z}_n$ (or finite fields $\mathbb{Z}_p$) are polynomial rings, which lead to the concept of extension fields. These extension fields are later used to construct the composite field representations and corresponding algorithms using operations in subfields of the finite field.

Definition A.10 (Def. 1.48 [LN96]) *Let $\mathbb{F}$ be a finite field. Then $\mathbb{F}[x]$ is a polynomial ring with elements of the form $a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$, with coefficients $a_i \in \mathbb{F}$. $\mathbb{F}$ is called the* ground field *of* $\mathbb{F}[x]$[2].

The arithmetic in polynomial rings is defined as follows. Addition is computed by adding the coefficients a_i from both summands using the addition operator

[2]The ground field $\mathbb{F}$ is also often called coefficient field.

of the ground field $\mathbb{F}$. Multiplication is more complicated. In particular, the multiplication of two polynomials $a = \sum_{i=0}^{n} a_i x^i$ and $b = \sum_{j=0}^{m} b_j x^j$ is defined as $c = \sum_{i=0}^{n} \sum_{j=0}^{m} a_i b_j x^{i+j}$, e.g. $(a_2x^2 + a_3x^3) \otimes (b_1x + b_3x^3) = a_2b_1x^3 + a_3b_1x^4 + a_2b_3x^5 + a_3b_3x^6$. Analog to addition, the coefficients are multiplied using the multiplication from the ground field $\mathbb{F}$.

The concept of residue class rings over $\mathbb{Z}$can be extended for polynomial rings. Instead of the arithmetic modulo a natural number, a polynomial $f \in \mathbb{F}[x]$ is used as modulus. The following theorem follows directly from Thm. A.5 and Def. A.10.

Theorem A.11 *Let $\mathbb{F}[x]$ be a polynomial ring and $f \in \mathbb{F}[x]$. Then $\mathbb{F}[x]/\langle f \rangle$ is a residue class ring modulo f.*

This concept is further studied in the notion of a finite extension, which is basically a polynomial residue class ring modulo a polynomial f. In the notation $\mathbb{F}[x]/\langle f \rangle$, $\langle f \rangle$ is called an ideal. The notation is usually used to denote such extensions, however, because ideals are not further needed in the present thesis, they are not discussed here.

If the polynomial f is irreducible, the residue class ring modulo f $\mathbb{F}[x]/\langle f \rangle$ is a finite field extension or simply a finite field. This is an extension of Thm. A.6 and Thm. A.11, because irreducible polynomials are similar to prime numbers in $\mathbb{N}$, i.e. a polynomial with degree k cannot be factored into polynomials with a smaller degree $0 < l < k$.

Theorem A.12 (Thm. 1.61 [LN96]) *Let $\mathbb{F}$ be a finite field. Then the ring $\mathbb{F}[x]/\langle f \rangle$ is a finite field iff f is an irreducible polynomial over $\mathbb{F}$.*

Definition A.13 *Let $\mathbb{F}$ be a finite field and $f(x)$ an irreducible polynomial in $\mathbb{F}[x]$, then $\mathbb{K} = \mathbb{F}[x]/\langle f \rangle$ is called a* finite field extension $\mathbb{K}/\mathbb{F}$.

It is obvious from the definition of the addition in polynomial rings, that the characteristic of the newly constructed extension fields is also prime and thus, the following corollary holds.

Corollary A.14 *Let $\mathbb{F}_p$ be a finite field with characteristic p. Then all finite extensions $\mathbb{F}_{p^n}/\mathbb{F}_p$ also have characteristic p.*

An important concept is the degree of an extension. The degree is easily described using the observation that every finite extension can be interpreted as a vector space L over the field K. The dimension of that vector space is the degree of the extension L/K, which is equal to the degree of the irreducible polynomial f.

Definition A.15 (Def. 1.83 [LN96]) *Let K/M be a finite extension. Then the degree of the extension K/M is the dimension of the vector space K over M. The notation is $[K : M]$.*

A finite field extension of a finite field is itself a finite field. Thus, it is possible to further extend such a finite field extension. The degree of these repeated extensions can be calculated according to Thm. A.16.

Theorem A.16 (Thm. 1.84 [LN96]) *Let K/L and L/M be two finite field extensions. Then K/M is a finite field extension with the degree $[K : M] = [K : L][L : M]$.*

To show that the theorem is true, one has to prove that the combination of a basis of K/L and a basis of L/M is linearly independent [LN96].

Furthermore, the number of elements contained in a finite field is connected to the degree of a finite extension as follows:

Lemma A.17 (Lemma 2.1 [LN96]) *Let $M = \mathbb{F}_q$ be a finite field and K/M a finite field extension. Then, the field K/M has $|K/M| = q^{[K:M]}$ elements.*

Proof First, suppose that q is prime, then because of Thm. A.6 $|M| = q$. Furthermore, every element of K can be represented using a basis $\{\alpha_1, \dots, \alpha_{[K:M]}\} \subseteq \mathbb{F}_q$. The elements of the basis have to be linearly independent.

An element $b \in K$ is thus represented as $b_1\alpha_1 + \cdots + b_{[K:M]}\alpha_{[K:M]}$. Each b_i can have q values and thus, there are $q^{[K:M]}$ possible values for b, because the basis elements are linearly independent. The same reasoning can be applied inductively for the case when q is not prime. $\square$

The most important result for the further work is the uniqueness of finite fields, i.e. there is exactly one finite field up to isomorphism with $q = p^n$ elements, p prime. However, to prove this theorem, a little bit more finite field theory has to be introduced:

Lemma A.18 (Lemma 2.3 [LN96]) *Let $\mathbb{F}_q$ be a finite field with $|\mathbb{F}_q| = q$. Then $a^q = a$ for all $a \in \mathbb{F}$.*

Proof $\mathbb{F}_q \setminus \{0\}$ forms a group under multiplication. Hence, by Lagrange's theorem $a^{q-1} = 1$ and therefore $a^q = a$. □

The following lemma is usually used in a more general form for commutative rings (the Frobenius endomorphism), but the more restricted version for finite fields is sufficient for the following purposes.

Lemma A.19 (Thm 1.46 [LN96]) *Let $\mathbb{F}_q$ be a finite field with $q = p^n$. Then for two elements $a, b \in \mathbb{F}_q$, $(a+b)^q = a^q + b^q$ and $(a-b)^q = a^q - b^q$.*

Proof The following property is $\binom{p}{i} \equiv 0 \bmod p$ true for all $0 < i < p$ and thus $(a+b)^p = a^p + \binom{p}{1}a^{p-1}b + \cdots + \binom{p}{p-1}ab^{p-1} + b^p = a^p + b^p$. This result can be extended to $q = p^n$ by induction, because $(a+b)^{(p^n)} = (a+b)^{p(p^{n-1})} = (a^p + b^p)^{(p^{n-1})}$. A similar argument with alternating signs of the summands holds for the second case. □

An extension can be alternatively defined by adjoining the elements of a set $M \subseteq F$ to a subfield $K \subseteq F$, denoted as $K(M)$, or if $M = \{\alpha_1, \ldots, \alpha_n\}$, simply $K(\alpha_1, \ldots, \alpha_n)$ (Def. 1.79 [LN96]). The extension is then the smallest subfield of F which contains both K and M. This is used in the next definition.

Definition A.20 (Def. 1.89 [LN96]) *Let K be a finite field, F be an extension field of K and $f \in K[x]$. Then F splits f, if f can be factored into linear factors in $F[x]$, i.e. there are elements $\alpha_1, \ldots, \alpha_n \in F$, such that*

$$f(x) = a(x - \alpha_1)(x - \alpha_2)\cdots(x - \alpha_n),$$

where a is the leading coefficient of f. The field F is said to be the splitting field *of f over K if f splits in F and if $F = K(\alpha_1, \ldots, \alpha_n)$.*

Because of the last condition $F = K(\alpha_1, \ldots, \alpha_n)$, the splitting field is the smallest field which contains all the roots $\alpha_1, \ldots, \alpha_n$. This idea leads to the result that a splitting field f over K always exists and is unique up to isomorphism (Thm. 1.91 [LN96]).

The next theorem states the central result, that all finite fields with the same cardinality (i.e. the same number of elements) are isomorphic. This result makes it possible for a developer to choose a representation of a finite field to improve the efficiency of an implementation in terms of circuit size, depth or another metric such as power consumption.

Theorem A.21 (Thm. 2.5 [LN96]) *There exists exactly one finite field $\mathbb{F}_q$ up to isomorphism, with $q = p^n$, p prime and $n \geq 1$. This finite field has exactly $|\mathbb{F}_q| = q = p^n$ elements and is isomorphic to the splitting field of $x^q - x$ over $\mathbb{F}_p$.*

Proof (Existence) Let $f(x) = x^q - x$ be a polynomial in $\mathbb{F}_p[x]$ and let F be the splitting field of f over $\mathbb{F}_p$. Then f has q distinct roots, because it has no common roots with the derivative $f'(x) = qx^{q-1} - 1 = q - 1 = -1$ and by Thm. 1.68. [LN96].

Let $S = \{a \in F : a^q = a\}$ be the subfield of F that contains only the roots of $x^q - x$, then S is a subfield of F, because for all elements $a, b \in F$ (by applying Lem. A.18 and Lem. A.19):

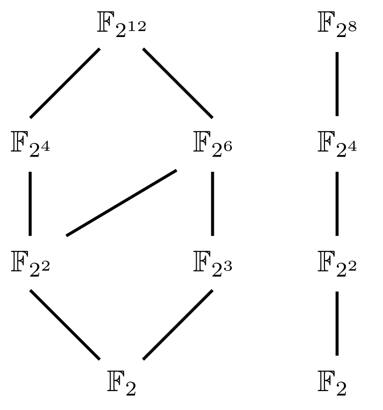
- $0, 1 \in S$,
- $(a - b)^q = a^q - b^q = a - b \in S$.
- and for $b \neq 0$, $(ab^{-1})^q = a^q b^{-q} = ab^{-1} \in S$

Furthermore, S must split $x^q - x$, because S contains all its roots, and therefore $S = F$, because the splitting field F is the smallest field that splits $x^q - x$.

(Uniqueness) Let F be a finite field with $q = p^n$ elements, then F contains $\mathbb{F}_p$ and furthermore it splits $x^q - x \in \mathbb{F}_p[x]$, because $a^q - a = 0$ for all q elements $a \in F$ (Lemma A.18). Therefore, because of the uniqueness of splitting fields, the result follows. $\square$

Each finite field contains a number of subfields. For a finite field $\mathbb{F}_{p^n}$ the subfields are defined by the divisibility relation. In particular, for an extension $\mathbb{F}_{p^n}/\mathbb{F}_{p^m}$ with degree n, each subfield has a degree m, such that $m|n$.

This may also be expressed by the lattice of subfields of finite fields. Two examples are depicted in Fig. A.1. The lattice for the subfields of the finite field $\mathbb{F}_{2^8}$ is very simple (Fig. A.1b). This is important, because it plays a major role in the further optimization approach, especially in the optimization of the AES S-box used by the Grøstl and the Photon hash functions.



(a) $\mathbb{F}_{2^{12}}$ (b) $\mathbb{F}_{2^8}$

Figure A.1: Lattices of the subfields of a finite field.

A.4 Representations

A finite field element has many possible representations. First of all, many isomorphic finite fields $\mathbb{F}_{p^n}$ can be constructed using repeated finite extensions, if there exists one or more m with $2 \leq m < n$ and $m|n$. For example, the finite field $\mathbb{F}_{p^6}$ may be constructed using the extension $\mathbb{F}_{p^6}/\mathbb{F}_p$ or in two steps – first $\mathbb{F}_{p^2}/\mathbb{F}_p$ and then $\mathbb{F}_{p^6}/\mathbb{F}_{p^2}$. Because of Thm. A.21 both finite fields are isomorphic. However, the representation of the elements differs for the first and the second method to construct the same finite field.

Furthermore, even using the same method to construct a finite field may lead to different representations of field elements, because there are many possible bases and for most extension fields, there are several irreducible polynomials. Both choices have a major impact on the number of ground field operations. Therefore, it is necessary to understand the different representations. The number of possible bases is rather large, i.e. if $\mathbb{F}_{q^m}$ is an extension field over $\mathbb{F}_q$, then the number of unique bases is $(q^m-1)(q^m-q)(q^m-q^2)\cdots(q^m-q^{m-1})$ (Ex. 2.37 [LN96]). Fortunately, there are only several types of bases that are interesting for the composite field approach.

The probably easiest to grasp basis is the polynomial basis. Furthermore it is relatively easy to implement the multiplication algorithm for a polynomial basis representation in software implementations.

Definition A.22 *A* polynomial basis *of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$ is a basis of the form*

$\{1, \alpha, \alpha^2, \dots, \alpha^{n-1}\}$ *for some primitive root* $\alpha \in \mathbb{F}_{q^n}$.

Example A.23 *An element in a polynomial basis is represented as* $a_0 + a_1\alpha + a_2\alpha^2 + \cdots + a_{n-1}\alpha^{n-1}$, *where all* $a_i \in \mathbb{F}_q$.

For a thorough evaluation of all possibilities, the number of different polynomial bases is interesting. This number has a close connection to the number of irreducible polynomials.

Definition A.24 *([Möb32, Mer74]) Let* $m \in N$, *then the* Möbius function $\mu(m)$ *is defined as follows:*

$$\mu(m) = \begin{cases} 1, & \text{if } m = 1 \\ 0, & \text{if } m \text{ is divisible by the square of a prime} \\ (-1)^k, & \text{if } m \text{ is the product of } k \text{ distinct primes} \end{cases}$$

Theorem A.25 (Thm. 3.25 [LN96]) *Let* $\mathbb{F}_q[x]$ *be a polynomial ring over* $\mathbb{F}_q$, *then the number of monic irreducible polynomials of degree* n *is exactly:*

$$N_q(n) = \frac{1}{n}\sum_{d|n} \mu(\frac{n}{d})q^d = \frac{1}{n}\sum_{d|n} \mu(d)q^{n/d}.$$

Since every irreducible polynomial of degree n has n distinct primitive roots, the number of polynomial bases is exactly $\sum_{d|n} \mu(d)q^{n/d}$. For example, the number of monic irreducible polynomials of degree 2 over $\mathbb{F}_2$, $\mathbb{F}_{2^2}$ and $\mathbb{F}_{4^2}$ are displayed in Tab. A.1. The corresponding number of polynomial bases is shown in Tab. A.2.

Table A.1: Number of monic irreducible polynomials of degree 2.

polynomial ring	$\mathbb{F}_2[x]$	$\mathbb{F}_{2^2}[x]$	$\mathbb{F}_{4^2}[x]$
number of polynomials	1	2	8

Table A.2: Number of different polynomial bases.

finite field	$\mathbb{F}_{2^2}/\mathbb{F}_2$	$\mathbb{F}_{4^2}/\mathbb{F}_4$	$\mathbb{F}_{16^2}/\mathbb{F}_{16}$
number of polynomial bases	2	4	16

For hardware implementations, using a normal basis sometimes results in a better resource utilisation [OM86].

Definition A.26 (Def. 2.32 [LN96]) *A* normal basis *of* $\mathbb{F}_{q^n}$ *over* $\mathbb{F}_q$ *is a basis of the form* $\{\alpha, \alpha^q, \ldots, \alpha^{q^{n-1}}\}$ *for some primitive root* $\alpha \in \mathbb{F}_{q^n}$*.*

Compared to the case of polynomial bases, there are less possible normal bases. The reasoning will become apparent with the following theorem:

Theorem A.27 (Thm. 2.14 [LN96]) *Let* f *be an irreducible polynomial of degree* n *in* $\mathbb{F}_q[x]$*. Then all roots are in* $\mathbb{F}_{q^n}$ *and for one root* $\alpha \in \mathbb{F}_{q^m}$*, all roots are* $\alpha, \alpha^q, \alpha^{q^2}, \ldots, \alpha^{q^{n-1}}$*.*

The key observation is, that the roots of an irreducible polynomial $f(x)$ correspond exactly to the elements of a normal basis. Thus, the number of normal bases and the number of irreducible polynomials coincides. This is also the reason for the efficient squaring operation over an extension field of $\mathbb{F}_2$, which is in fact a simple bit rotate operation.

The most important difference between these representations for the following optimizations is the complexity of arithmetic operations, i.e. the number of operations over the ground field $\mathbb{F}_q$. The most fundamental operations are addition and multiplication. The *addition* of two polynomials $a, b \in \mathbb{F}_q[x]$ each of degree n is very easy and never changes depending on the representation:

$$s = a \oplus b = \sum_{i=0}^{n} (a_i \oplus b_i) x^i$$

In contrast, the multiplication differs considerably between polynomial and normal bases. The *multiplication* can be generally expressed as a multiplication table independent of the basis. For example, Fig. A.2 presents the multiplication table for the finite field $\mathbb{F}_{2^2}$ using a polynomial basis (Fig. A.2a) and a normal basis (Fig. A.2b).

According to Thm. A.21, all finite field representations of the same order are isomorphic to each other. Therefore, the general structure of such multiplication tables is the same, only the names (or symbols) are changed. The main drawback of the multiplication tables is the exponential growth with the number of bits of the binary representation of finite field elements. Hence, for practical

Algorithm A.1 Multiplication in finite fields

```
Require: $a, b \in \mathbb{F}_q[x]/\langle f \rangle$ in polynomial basis representation
Ensure: $c \leftarrow a \otimes b, c \in \mathbb{F}_q[x]/\langle f \rangle$
  $c \leftarrow \sum_{i=0}^{n} \sum_{j=0}^{m} a_i b_j x^{i+j}$
  $c \leftarrow c \mod f(x)$
  return $c$
```

$\otimes$	00	01	10	11
00	00	00	00	00
01	00	01	10	11
10	00	10	11	01
11	00	11	01	10

(a) Polynomial Basis

$\otimes$	00	01	10	11
00	00	00	00	00
01	00	10	11	01
10	00	11	01	10
11	00	01	10	11

(b) Normal Basis

Figure A.2: Multiplication table for $\mathbb{F}_{2^2}$.

implementations, often a polynomial multiplication algorithm is used with a polynomial basis representations, followed by modular reduction (Alg. A.1).

Depending on the finite field, there exists a variety of possible algorithms. For small fields, the naïve textbook implementations of polynomial multiplication and division is often sufficient and very easy to implement. For larger finite fields, especially for fields with a high characteristic, it is possible to use the Karatsuba multiplication [WP06, KO62] and some variation on the Barrett reduction [Bar87] or the Montgomery multiplication [Mon85].

The multiplication algorithm for normal basis representations is in its general form more complicated and only the special case of squaring is very efficient. There are basically two possibilities. The first is to convert the normal basis representation to a polynomial basis and then use the multiplication algorithm for polynomial bases and later convert it back. The second is to use a specialized normal basis multiplier, which usually implies the precomputation of a multiplication matrix M. This matrix for finite fields over $\mathbb{F}_2$ can be calculated for example using Alg. A.6.3 described in IEEE Std. 1363-2000 [Yin00]. The multiplication is then computed according to Alg. A.2. In this work the first approach was used, because all the internal calculations are implemented using a polynomial basis and thus, it is easier to do the conversion to a normal basis representation once as a post processing operation instead of implementing all

Algorithm A.2 Multiplication in finite fields over $\mathbb{F}_2$ using a normal basis.

```
Require: $M \in \mathbb{F}_2^{m \times m}$, $a, b \in \mathbb{F}_{2^m}$ in normal basis representation
Ensure: $c \leftarrow ab$
  $x \leftarrow a$
  $y \leftarrow b$
  for $i = 0$ to $m - 1$ do
    $c_i \leftarrow aMb^T$
    $x \leftarrow \mathrm{RotateLeft}(x, 1)$
    $y \leftarrow \mathrm{RotateLeft}(y, 1)$
  end for
  return $c$
```

necessary algorithms for a normal basis representation.

Note, that even though polynomial basis multiplication is usually faster in software, both described algorithms have a roughly quadratic runtime in terms of the number of bits per finite field element. However, if the parameters of the finite field are fixed for an application, such as the basis and the irreducible polynomial, it is often possible to optimize the implementation and hence, to achieve a significant speedup.

A.5 Composite Fields

One of the more expensive problems over finite fields is the computation of the inverse of an element a in a finite field. Several standard algorithms can be used, e.g. the extended Euclidean algorithm [Knu69]. Another possibility is to use a square and multiply algorithm, because according to Lem. A.18 the inverse of a field element $a \in \mathbb{F}_q$ can be calculated as $a^{-1} = a^{q-2}$. For example, let $a \in \mathbb{F}_{2^8}$, then the inverse of a is $a^{-1} = a^{254}$. The main advantage of the latter approach is, that no general algorithm for modular computation has to be implemented; a modular reduction with the irreducible polynomial is still necessary, but this modular reduction can be implemented more efficiently than a general algorithm.

Both ideas are quite inefficient for hardware implementations – the extended Euclidean algorithm requires a polynomial division algorithm, the square and multiply approach takes a lot of time to compute. A different observation leads to a more area-efficient implementation. Every algorithm over a finite field may also be expressed in terms of simpler operations over a subfield of the original finite field. For example, the inversion in $\mathbb{F}_{2^8}$ can be expressed in terms of simpler operations in $\mathbb{F}_{2^4}$ and these operations may in turn be implemented using operations in $\mathbb{F}_{2^2}$ and $\mathbb{F}_2$.

In this section, this decomposition approach is demonstrated exemplary using the finite field $\mathbb{F}_{2^8}$. However, the methodology can be easily adapted to larger finite fields, if needed.

Each element of the finite field $\mathbb{F}_{2^8}$ can be expressed as polynomial a of degree 7 of the following form:

$$a = a_0 + a_1x^1 + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7,$$

where each coefficient $a_i \in \mathbb{F}_2$. Instead of this polynomial representation, we can express the same element in an isomorphic finite field $\mathbb{F}_{16^2}/\mathbb{F}_{16}$, by a polynomial

$$a' = a'_0 + a'_1x'$$

with coefficients a'_0, a'_1 in $\mathbb{F}_{16} = \mathbb{F}_{2^4}$. This isomorphic conversion can be continued with $\mathbb{F}_{2^4}$, at the end yielding a nested representation using the subfields $\mathbb{F}_2$,$\mathbb{F}_{2^2}/\mathbb{F}_2$, $\mathbb{F}_{(2^2)^2}/\mathbb{F}_{2^2}$ and $\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}$.

For this conversion, three irreducible polynomials of degree two are necessary:

$$\begin{aligned} f(x) &= t + sx + x^2 \\ g(y) &= v + uy + y^2 \\ h(z) &= 1 + z + z^2, \end{aligned}$$

where $f(x)$ is an irreducible polynomial over $\mathbb{F}_{16}$, i.e. $f(x) \in \mathbb{F}_{16}[x]$, $g(y) \in \mathbb{F}_4[y]$ and $h(z) \in \mathbb{F}_2[z]$. There is only one irreducible polynomial $h(z)$ of degree 2, therefore no search for such a polynomial is necessary.

The exact values for s,t,u and v may have several different values, and thus, different polynomials can be constructed. Several irreducible polynomials have to be investigated, because for each irreducible polynomial, the area consumption of the hardware implementation differs. The choice of basis for each subfield adds additional variety, because it also influences the gate count.

Thus, at first only a general formula for both inversion and multiplication in each subfield is developed, independent of the exact irreducible polynomial. This decomposition was originally developed in the papers by Canright [Can05a, Can05b]). Let $a, b \in \mathbb{F}_{16^2}$ and b be the inverse element of a. Then the inverse may be calculated as follows.

$$\begin{aligned} ab &= (a_0 + a_1x)(b_0 + b_1x) \mod (t + sx + x^2) \\ &= (a_0b_0) + (a_0b_1 + a_1b_0)x + (a_1b_1)x^2 \mod (t + sx + x^2) \\ &= (a_0b_0) + (a_0b_1 + a_1b_0)x + (a_1b_1)x^2 + a_1b_1(t + sx + x^2) \\ &= (a_0b_0 + a_1b_1t) + (a_0b_1 + a_1b_0 + a_1b_1s)x \\ 1 &= 1 + 0x \end{aligned}$$

The next step is to solve the two equations:

$$\begin{aligned} 1 &= a_0b_0 + a_1b_1t \\ 0 &= a_0b_1 + a_1b_0 + a_1b_1s \end{aligned}$$

$$\begin{aligned} a_1 &= a_0a_1b_0 + a_1^2b_1t \\ 0 &= a_0^2b_1 + a_0a_1b_0 + a_0a_1b_1s \end{aligned}$$

$$\begin{aligned} a_1 &= b_1(a_1^2t + a_0^2 + a_0a_1s) \\ a_1b_0 &= (a_0 + a_1s)b_1 \end{aligned}$$

$$\begin{aligned} b_1 &= (a_1^2t + a_0^2 + a_0a_1s)^{-1}a_1 \\ b_0 &= (a_1^2t + a_0^2 + a_0a_1s)^{-1}(a_0 + a_1s) \end{aligned}$$

The coefficients a_0, a_1, b_0, b_1, s and t are all elements in $\mathbb{F}_{16}$. Hence, the calculation of the inverse uses several operations in $\mathbb{F}_{16}$, in particular several multiplications, additions and one inversion.

The inversion in $\mathbb{F}_{16} = \mathbb{F}_{4^2}/\mathbb{F}_4$ is very similar. For the elements $c, d \in \mathbb{F}_{4^2}$, where d is the inverse of c, the following equation describes the process of inversion:

$$cd = (c_0d_0 + c_1d_1v) + (c_0d_1 + c_1d_0 + c_1d_1u)y$$

Solving the two equations similar to the previous case yields:

$$\begin{aligned} d_1 &= (c_1^2 v + c_0^2 + c_0 c_1 u)^{-1} c_1 \\ d_0 &= (c_1^2 v + c_0^2 + c_0 c_1 u)^{-1} (c_0 + c_1 u) \end{aligned}$$

The computation of the inverse in $\mathbb{F}_{2^2}$ is simpler, because the coefficients of the irreducible polynomial are fixed to one. For two elements $e, f \in \mathbb{F}_{2^2}$, where f is the inverse of e, solving the similar equations leads to:

$$\begin{aligned} f_1 &= (e_1^2 + e_0^2 + e_0 e_1)^{-1} e_1 \\ f_0 &= (e_1^2 + e_0^2 + e_0 e_1)^{-1} (e_0 + e_1) \end{aligned}$$

The coefficients e_0, e_1, f_0 and f_1 are all elements of $\mathbb{F}_2$. In $\mathbb{F}_2$, squaring and inversion is the same as identity, therefore these equations can be simplified further:

$$\begin{aligned} f_1 &= (e_1^2 + e_0^2 + e_0 e_1)^{-1} e_1 \\ &= (e_1 + e_0 e_1 + e_0 e_1) \\ &= e_1 \end{aligned}$$

$$\begin{aligned} f_0 &= (e_1^2 + e_0^2 + e_0 e_1)^{-1} (e_0 + e_1) \\ &= (e_1 + e_0 + e_0 e_1)(e_0 + e_1) \\ &= (e_0 e_1 + e_0 + e_0 e_1 + e_1 + e_0 e_1 + e_0 e_1) \\ &= e_0 + e_1 \end{aligned}$$

A general formulation for multiplication in $\mathbb{F}_{16^2}/\mathbb{F}_{16}$ is calculated as follows:

$$\begin{aligned} ab &= (a_0 + a_1 x)(b_0 + b_1 x) \mod (t + sx + x^2) \\ &= (a_0 b_0) + (a_0 b_1 + a_1 b_0)x + (a_1 b_1)x^2 \mod (t + sx + x^2) \\ &= (a_0 b_0) + (a_0 b_1 + a_1 b_0)x + (a_1 b_1)x^2 + a_1 b_1 (t + sx + x^2) \\ &= (a_0 b_0 + a_1 b_1 t) + (a_0 b_1 + a_1 b_0 + a_1 b_1 s)x \end{aligned}$$

Essentially the same result holds for the multiplication in $\mathbb{F}_{4^2}/\mathbb{F}_4$:

$$cd = (c_0 d_0 + c_1 d_1 v) + (c_0 d_1 + c_1 d_0 + c_1 d_1 u)y$$

In contrast to inversion in $\mathbb{F}_{2^2}$, multiplication is not much easier than in $\mathbb{F}_{16^2}$ and $\mathbb{F}_{4^2}$:

$$ef = (e_0 f_0 + e_1 f_1) + (e_0 f_1 + e_1 f_0 + e_1 f_1)z$$

Very similar calculations can be done for normal basis representations. The results were also presented by Canright in [Can05a].

The representation of the newly constructed finite field is different to the original one. Therefore, it is necessary to convert between the two isomorphic representations. A relevant result of Sunar [Sun05] is the following theorem. However, Sunar did only provide an informal proof sketch why this works.

Theorem A.28 *Let $\mathbb{F}_q[x]/\langle f\rangle$ be a finite field with the normal or polynomial basis $B = \{\alpha_1, \ldots, \alpha_n\}$ and let $\mathbb{G}_r[y]/\langle g\rangle$ be a finite field isomorphic to $\mathbb{F}_q[x]/\langle f\rangle$ with the normal or polynomial basis $B' = \{\beta_1, \ldots, \beta_m\}$, then*

1. *Finding an isomorphism between $\mathbb{F}_q[x]/\langle f\rangle$ and $\mathbb{G}_r[y]/\langle g\rangle$ is reducible to finding a primitive root of the defining irreducible polynomial $f(x)$ in terms of $\mathbb{G}_r[y]/\langle g\rangle$.*

2. *Finding a primitive root of an irreducible polynomial is reducible to factoring the polynomial $f(x)$ over the splitting field $\mathbb{G}_r[y]/\langle g\rangle$.*

Proof It is well known from linear algebra, that the change of basis matrix M, to convert elements in terms of the basis B to elements in terms of the basis B', may be expressed in terms of the basis vectors $\alpha_1, \ldots, \alpha_m$, if they are converted to representations in B' (Ch. 2 Thm. 7 [HK71]). The columns of M are the vectors $\alpha_1, \ldots, \alpha_n$ converted to the representation in B'.

Since each polynomial and normal basis is defined using a single primitive root of the irreducible polynomial $f(x)$, it is enough to find the representation of one primitive root in terms of the basis B'. This is equivalent to finding a root α of the irreducible polynomial $f(x)$ over the splitting field of $\mathbb{G}_r[y]/\langle g\rangle$, because it is isomorphic to the splitting field over $\mathbb{F}_q[x]/\langle f\rangle$ (Thm. A.21).

Finding a primitive root is reducible to factoring the irreducible polynomial in its linear factors, because a primitive root γ of $f(x)$ results in $f(\gamma) = 0$. Since all factorizations are of the form $f(x) = (x - \gamma_1)(x - \gamma_2)\cdots(x - \gamma_n)$, factoring the polynomial $f(x)$ in its linear polynomials yields all primitive roots of the polynomial. $\square$

Example A.29 *A root of the irreducible polynomial $f(\mathcal{X})$ of degree 8 in $\mathbb{F}_2[x]$ is an element $\alpha \in \mathbb{F}_{2^8}$. This element can be alternatively interpreted as $q_i \in \mathbb{F}_{((2^2)^2)^2}$ with $1 \leq i \leq 8$, such that:*

$$f(q_i(x, y, z)) = 0$$

This is identical to factoring $f(\mathcal{X})$ into its linear factors over the splitting field $\mathbb{F}_{((2^2)^2)^2}$:

$$f(\mathcal{X}) = (\mathcal{X} - q_1(x, y, z))(\mathcal{X} - q_2(x, y, z)) \cdots (\mathcal{X} - q_8(x, y, z))$$

Each $q_i(x, y, z)$ is a root of $f(\mathcal{X})$ in terms of the composite field $\mathbb{F}_{((2^2)^2)^2}$.

A.6 Algorithms

In this thesis, several straightforward finite field algorithms were used in software to generate the irreducible polynomials, the polynomial basis and normal basis representations and also the matrices to convert to and from each representation. In this section, the following algorithms implemented in software are described:

- Addition, multiplication, division, exponentiation, and extended Euclidean algorithm
- Irreducibility test
- Root finding
- Basis transformation

As most of these algorithms are well known [MOV97] and the implementations were not specifically optimized, the algorithms will not be described in depth.

A.6.1 Basic Operations over $\mathbb{F}_{2^n}$

Addition Addition in finite fields with characteristic 2 is very easy, because it is just a bitwise exclusive-or. Therefore addition is easier than in many other algebras, e.g. the ring $\mathbb{Z}$.

Algorithm A.3 Polynomial Division

```
Require: $a, m \in \mathbb{F}_{2^n}[x],\ m \neq 0$
Ensure: $a \leftarrow qm \oplus r$, such that $r < m$.
  msc $\leftarrow m_{deg(m)}$ -- Save the most significant coefficient of $m$.
  $m \leftarrow \text{msc}^{-1} \otimes m$ -- Normalize $m$, i.e. make $m$ a monic polynomial.
  $a \leftarrow \text{msc}^{-1} \otimes a$ -- Adjust $a$ to be able to revert the normalization of $m$ after division.
  for $i = \deg(a)$ downto $\deg(m)$ do
    if $a_i \neq 0$ then
      $q_i \leftarrow a_i$
      for $j = \deg(m) - 1$ downto 1 do
        $a_{i+j-\deg(m)} \leftarrow a_{i+j-\deg(m)} \oplus (a_i \otimes m_j)$
      end for
      $a_i \leftarrow 0$
    end if
  end for
  $r \leftarrow \text{msc} \otimes a$ -- Revert the normalization.
  return $(q, r)$
```

Polynomial Multiplication Multiplication is a little bit more complex but is also very easy to implement. The implementation uses a variant from a textbook algorithm according to the already presented Alg. A.1.

Polynomial Division The algorithm for polynomial division is more complicated than addition and multiplication. The easiest case of division is for extension fields over $\mathbb{F}_2$, because all polynomials are always monic.

For the general polynomial division algorithm (Alg. A.3), first, the divisor and the dividend have to be normalized before performing the division. Thus, before returning the result, the remainder has to be denormalized again. For the normalization, the multiplicative inverse of the most significant coefficient of the divisor has to be calculated. Furthermore, the original most significant coefficient of the divisor has to be stored, to be able to denormalize the result.

Polynomial Exponentiation The polynomial exponentiation is implemented using a repeated square-and-multiply algorithm (Alg. A.4).

Note, that in the general case of polynomial rings, the result grows exponentially. However, for the case of finite fields, this problem does not exist, because the temporary result of each multiplication is performed modulo the irreducible polynomial.

Algorithm A.4 Polynomial exponentiation

```
Require: $a \in \mathbb{F}_{2^n}[x], e \in \mathbb{N}$, with the binary representation $e = \sum_{i=1}^{k} e_i 2^i$
Ensure: $b \leftarrow a^e$
  $b \leftarrow 1$
  for $i = k$ downto 1 do
    if $e_i = 1$ then
      $b \leftarrow b^2 \otimes a$
    else
      $b \leftarrow b^2$
    end if
  end for
  return $b$
```

Extended Euclidean Algorithm The greatest common divisor and the multiplicative inverse of a finite field element are both calculated using the extended Euclidean algorithm (Alg. A.5). In the description of the algorithm, d is the greatest common divisor, s is the multiplicative inverse of a modulo b, and t is the multiplicative inverse of b modulo a.

For the computation of the multiplicative inverse, it is necessary to normalize the result from the extended Euclidean algorithm for the polynomial version, i.e. if $d \neq 1$, it is necessary to calculate $s' = s \times d^{-1}$ and $t' = t \times d^{-1}$. This follows from the definition of co-prime polynomials, i.e. two polynomials are co-prime, if they do not have a common divisor with degree ≥ 1 and thus, $d \neq 1$ is possible. The definition of the multiplicative inverse does not have this relaxed property (Def. A.3), but requires that $aa^{-1} = 1$, which would not be the case without the additional normalization.

Algorithm A.5 Polynomial extended Euclidean algorithm ($extgcd(a, b)$)

```
Require: $a, b \in \mathbb{F}_{2^n}[x]$
Ensure: $d \leftarrow gcd(a, b)$ and $as \oplus bt = d$, such that $as = d \bmod b$ and $bt = d \bmod a$.
  $s_2 \leftarrow 1,\ s_1 \leftarrow 0,\ t_2 \leftarrow 0,\ t_1 \leftarrow 1$
  while $h \neq 0$ do
    $q \leftarrow g$ div $h$
    $r \leftarrow g$ mod $h$
    $s \leftarrow s_2 - qs_1$
    $t \leftarrow t_2 - qt_1$
    $g \leftarrow h$
    $h \leftarrow r$
    $s_2 \leftarrow s_1,\ s_1 \leftarrow s$
    $t_2 \leftarrow t_1,\ t_1 \leftarrow t$
  end while
  $d \leftarrow g,\ s \leftarrow s_2,\ t \leftarrow t_2$
  return $(d, s, t)$
```

A.6.2 Finding Irreducible Polynomials

As already mentioned, different irreducible polynomials result in different representations of the same finite field elements. Each representation may be more or less efficient. Thus, it is important to find and to evaluate all irreducible polynomials for each subfield.

The irreducibility test implemented was first published by Ben-Or [Ben81]. Instead of finding only one monic irreducible polynomial, all such polynomials have to be found for the evaluation. In the present case performance was not the most important aspect, therefore a brute force search over all polynomials of interest was implemented. For example, testing all polynomials of degree at most 2 over $\mathbb{F}_{16}$ results in a feasible number of 4096 different tests.

Nonetheless, the search space may be pruned easily, with two observations:

1. Only monic polynomials are interesting, therefore less polynomials have to be tested.

2. Every polynomial with lesser degree than desired may be omitted.

If non-monic polynomials would be used to construct the composite fields, testing for these polynomials would be completely unnecessary anyway. This is a result of the fact, that two irreducible polynomials f, g do not require to have $gcd(f, g) = 1$. Instead $gcd(f, g) = c$ for some $c \in \mathbb{F}$ is sufficient, where $\mathbb{F}$ is the ground field. This means that non-monic irreducible polynomials can be easily constructed by multiplying a monic polynomial with some constant c. Another observation reduces the number of irreducibility tests, because the search may be aborted, if all monic irreducible polynomials were found (Sec. A.4, Thm. A.25).

The test for irreducibility of a polynomial itself is relatively straightforward [Ben81, GP97] (Alg. A.6). The following Thm. A.30 is central to the idea of the algorithm, which is given here without proof.

Theorem A.30 (Thm 3.20 [LN96]) *Let $\mathbb{F}_q[x]$ be a polynomial ring, then for every $n \in \mathbb{N}$, $x^{q^n} - x$ is the product of all monic irreducible polynomials over $\mathbb{F}_q$, whose degrees divide n.*

The algorithm tests for all possible monic irreducible factors of $f(x)$ with low degree using the greatest common divisor algorithm. In fact, if and only if $f(x)$ has no common factor with $x^{q^k} - x$, where $k < n$, then $f(x)$ is irreducible. However, it is enough for the algorithm to test for $k \leq n/2$, because if $f(x)$ has a factor with a degree greater than $n/2$, the polynomial will always also have a factor with a degree less than $n/2$.

Algorithm A.6 Ben-Or irreducibility test [Ben81]

```
Require: $f(x) \in \mathbb{F}_{2^m}[x]$, $n = \deg(f)$
Ensure: true iff $f(x)$ is irreducible
  for $i = 1$ to $\lfloor \frac{n}{2} \rfloor$ do
    if $\gcd(f(x), x^{q^i} - x \bmod f(x)) \neq 1$ then
      return false
    end if
  end for
  return true
```

A.6.3 Root Finding Algorithm

For the present use case it is enough to investigate the problem to find the roots of an irreducible polynomial $f(x) \in \mathbb{F}_2[x]$, because the AES polynomial is $f(x) = x^8 + x^4 + x^3 + x + 1$. However, the principle can be easily adapted to other irreducible polynomials over field with binary characteristic. The idea to find the roots of such an irreducible polynomial works as follows. Since f is irreducible in $\mathbb{F}_2[x]$, the same polynomial is reinterpreted as a polynomial over another field, where f is guaranteed to have only linear factors. This happens, if each linear factor $(z - \alpha)$ leads to a root α of $f(z)$, where $\alpha \in \mathbb{F}_q$ and $q = 2^{\deg(f)}$. Therefore, the polynomial $f(z) \in \mathbb{F}_q[z]$ can be factored into linear factors using any factorization algorithm.

The factorization algorithm implemented is an adapted version of an equal degree factorization algorithm proposed for this purpose by Sunar [Sun05]. The original version of the algorithm was published by Ben-Or [Ben81], which is itself a modification of an algorithm developed by Rabin [Rab80]. However, it is also possible to use other factorization algorithms to find a root of an irreducible polynomial, e.g. [Ber67, Sho91].

Algorithm A.7 Sunar's factorization algorithm [Sun05].

```
**Require:** $f(z) \in \mathbb{F}_q[z]$, $n = \deg(f)$
**Ensure:** $g(z) \in \mathbb{F}_q[z]$ with degree 1.
  $g(z) \leftarrow f(z)$
  **while** $\deg(g(z)) > 1$ **do**
    $\alpha(z) \leftarrow \text{random}(\mathbb{F}_q[z]/\langle f \rangle)$
    $\beta(z) \leftarrow \alpha + \alpha^2 + \alpha^{2^2} + \cdots + \alpha^{2^n - 1}$
    $h(z) \leftarrow \gcd(g(z), \beta(z))$
    **if** $\deg(h(z)) > 0$ **then**
      $g(z) \leftarrow h(z)$
    **end if**
  **end while**
  **return** $g(z)$
```

This algorithm works in probabilistic polynomial time, because Ben-Or proved in his paper that the degree of f decreases with each iteration with the probability $1/2^{n-1}$. Therefore, after a polynomially bounded number of computation steps, the algorithm returns with a linear polynomial [Sun05, Ben81]. After normalization of $g(z)$, the constant part of this linear polynomial is a root of the original irreducible polynomial $f(x)$. Note, that it is enough to find one root α, because according to Thm. A.27, the roots of an irreducible polynomial of degree n are exactly $\alpha, \alpha^2, \ldots, \alpha^{2^{n-1}}$. Furthermore, if the representation of $\mathbb{F}_q$ is chosen appropriately, the root will already be in the target representation for constructing the basis conversion matrix.

A.6.4 Basis Conversion Matrix

The basis conversion matrix M_{poly} may be constructed with a root computed using Alg. A.7. Let B be a polynomial basis of a finite field and B' another polynomial basis of the same field. Then the conversion matrix M_{poly} to convert from basis B to B' is defined as $M_{\text{poly}} =_{\text{def}} (\alpha^0 \ \alpha^1 \ \ldots \ \alpha^{n-1})$, if α is a root of the irreducible polynomial f in terms of the basis B'. This means the columns of the matrix M_{poly} correspond exactly to the polynomial basis in a representation using basis B' [HK71]. The inverse matrix M_{poly}^{-1} may be easily computed using standard linear algebra techniques.

The conversion matrix M'_{normal} to convert from the polynomial basis to a normal basis may be constructed in a similar way as $M' =_{\text{def}} (\alpha, \alpha^q, \ldots, \alpha^{q^{n-1}})$.

A.7 Implementation Results AES S-box

As mentioned in Sec. 6.6.5, 432 different representations are evaluated. In theory there are more irreducible polynomials $f(x)$ and $g(y)$ which can be used to construct many more isomorphic representations in a similar way, but choosing $s = u = 1$ simplifies the arithmetic as previously discussed by Canright in [Can05a]. In Tab. A.3 all post place and route results for Spartan-3 FPGAs are presented.

In Tab. A.3, the values for u and v to form the irreducible polynomial $g(y)$ are displayed in a hexadecimal representation. Similar s and t for $f(x)$. Furthermore, the basis types for $\mathbb{F}_4$, $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ are displayed. The computed roots are not shown, because for each extension, only two roots α and α^2 are relevant to form the three bases $\{1, \alpha\}$ and $\{1, \alpha^2\}$, and $\{\alpha, \alpha^2\}$. The smallest results are displayed in boldface. Interestingly, the representation from [Can05a] is not among the smallest ones.

Table A.3: Results for the AES S-box composite field implementation (Spartan-3).

u	v	s	t	$\mathbb{F}_4$ basis	$\mathbb{F}_{16}$ basis	$\mathbb{F}_{256}$ basis	Spartan-3 [Slices]
1	2	1	8	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma\}$	38
1	2	1	8	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	33
1	2	1	8	$\{1,\alpha\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	36
1	2	1	8	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	39
1	2	1	8	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	37
1	2	1	8	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	39
1	2	1	8	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	43
1	2	1	8	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	34
1	2	1	8	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	36
1	2	1	8	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	35
1	2	1	8	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	43
1	2	1	8	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	45
1	2	1	8	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	35
1	2	1	8	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	38
1	2	1	8	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	42
1	2	1	8	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	39
1	2	1	8	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	38
1	2	1	8	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	40
1	2	1	8	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	35
1	2	1	8	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	41
1	2	1	8	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	44
1	2	1	8	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	35
1	2	1	8	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	39
1	2	1	8	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	40
1	2	1	8	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	36
1	2	1	8	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	36
1	2	1	8	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	40

1	2	1	9	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma\}$	33
1	2	1	9	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	41
1	2	1	9	$\{1,\alpha\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	39
1	2	1	9	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	36
1	2	1	9	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	37
1	2	1	9	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	40
1	2	1	9	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	35
1	2	1	9	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	42
1	2	1	9	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	38
1	2	1	9	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	40
1	2	1	9	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	43
1	2	1	9	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	46
1	2	1	9	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	39
1	2	1	9	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	35
1	2	1	9	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	40
1	2	1	9	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	36
1	2	1	9	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	39
1	2	1	9	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	43
1	2	1	9	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	35
1	2	1	9	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	42
1	2	1	9	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	44
1	2	1	9	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	35
1	2	1	9	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	36
1	2	1	9	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	38
1	2	1	9	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	37
1	2	1	9	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	42
1	2	1	9	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	40
1	2	1	A	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma\}$	36
1	2	1	A	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	35
1	2	1	A	$\{1,\alpha\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	37
1	2	1	A	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	43
1	2	1	A	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	39
1	2	1	A	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	35
1	2	1	A	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	42

1	2	1	A	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	36
1	2	1	A	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	38
1	2	1	A	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	37
1	2	1	A	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	47
1	2	1	A	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	51
1	2	1	A	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	34
1	2	1	A	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	45
1	2	1	A	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	38
1	2	1	A	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	33
1	2	1	A	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	38
1	2	1	A	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	41
1	2	1	A	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	35
1	2	1	A	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	38
1	2	1	A	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	43
1	2	1	A	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	34
1	2	1	A	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	38
1	2	1	A	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	39
1	2	1	A	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	37
1	2	1	A	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	40
1	2	1	A	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	42
1	2	1	B	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma\}$	38
1	2	1	B	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	40
1	2	1	B	$\{1,\alpha\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	37
1	2	1	B	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	37
1	2	1	B	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	40
1	2	1	B	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	40
1	2	1	B	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	30
1	2	1	B	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	40
1	2	1	B	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	44
1	2	1	B	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	35
1	2	1	B	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	45
1	2	1	B	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	47
1	2	1	B	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	36
1	2	1	B	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	41

1	2	1	B	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	41
1	2	1	B	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	38
1	2	1	B	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	36
1	2	1	B	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	36
1	2	1	B	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	35
1	2	1	B	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	40
1	2	1	B	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	40
1	2	1	B	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	37
1	2	1	B	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	39
1	2	1	B	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	38
1	2	1	B	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	39
1	2	1	B	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	38
1	2	1	B	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	35
1	2	1	C	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma\}$	33
1	2	1	C	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	40
1	2	1	C	$\{1,\alpha\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	37
1	2	1	C	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	42
1	2	1	C	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	39
1	2	1	C	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	36
1	2	1	C	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	31
1	2	1	C	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	36
1	2	1	C	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	38
1	2	1	C	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	36
1	2	1	C	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	46
1	2	1	C	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	40
1	2	1	C	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	36
1	2	1	C	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	42
1	2	1	C	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	41
1	2	1	C	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	32
1	2	1	C	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	38
1	2	1	C	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	39
1	2	1	C	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	34
1	2	1	C	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	38
1	2	1	C	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	41

1	2	1	C	$\{\alpha, \alpha^2\}$	$\{1, \beta^2\}$	$\{1, \gamma\}$	32
1	2	1	C	$\{\alpha, \alpha^2\}$	$\{1, \beta^2\}$	$\{1, \gamma^2\}$	36
1	2	1	C	$\{\alpha, \alpha^2\}$	$\{1, \beta^2\}$	$\{\gamma, \gamma^2\}$	40
1	2	1	C	$\{\alpha, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{1, \gamma\}$	33
1	2	1	C	$\{\alpha, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{1, \gamma^2\}$	38
1	2	1	C	$\{\alpha, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{\gamma, \gamma^2\}$	38
1	2	1	D	$\{1, \alpha\}$	$\{1, \beta\}$	$\{1, \gamma\}$	40
1	2	1	D	$\{1, \alpha\}$	$\{1, \beta\}$	$\{1, \gamma^2\}$	34
1	2	1	D	$\{1, \alpha\}$	$\{1, \beta\}$	$\{\gamma, \gamma^2\}$	38
1	2	1	D	$\{1, \alpha\}$	$\{1, \beta^2\}$	$\{1, \gamma\}$	37
1	2	1	D	$\{1, \alpha\}$	$\{1, \beta^2\}$	$\{1, \gamma^2\}$	36
1	2	1	D	$\{1, \alpha\}$	$\{1, \beta^2\}$	$\{\gamma, \gamma^2\}$	37
1	2	1	D	$\{1, \alpha\}$	$\{\beta, \beta^2\}$	$\{1, \gamma\}$	43
1	2	1	D	$\{1, \alpha\}$	$\{\beta, \beta^2\}$	$\{1, \gamma^2\}$	41
1	2	1	D	$\{1, \alpha\}$	$\{\beta, \beta^2\}$	$\{\gamma, \gamma^2\}$	42
1	2	1	D	$\{1, \alpha^2\}$	$\{1, \beta\}$	$\{1, \gamma\}$	38
1	2	1	D	$\{1, \alpha^2\}$	$\{1, \beta\}$	$\{1, \gamma^2\}$	43
1	2	1	D	$\{1, \alpha^2\}$	$\{1, \beta\}$	$\{\gamma, \gamma^2\}$	45
1	2	1	D	$\{1, \alpha^2\}$	$\{1, \beta^2\}$	$\{1, \gamma\}$	32
1	2	1	D	$\{1, \alpha^2\}$	$\{1, \beta^2\}$	$\{1, \gamma^2\}$	43
1	2	1	D	$\{1, \alpha^2\}$	$\{1, \beta^2\}$	$\{\gamma, \gamma^2\}$	36
1	2	1	D	$\{1, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{1, \gamma\}$	37
1	2	1	D	$\{1, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{1, \gamma^2\}$	40
1	2	1	D	$\{1, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{\gamma, \gamma^2\}$	42
1	2	1	D	$\{\alpha, \alpha^2\}$	$\{1, \beta\}$	$\{1, \gamma\}$	34
1	2	1	D	$\{\alpha, \alpha^2\}$	$\{1, \beta\}$	$\{1, \gamma^2\}$	41
1	2	1	D	$\{\alpha, \alpha^2\}$	$\{1, \beta\}$	$\{\gamma, \gamma^2\}$	41
1	2	1	D	$\{\alpha, \alpha^2\}$	$\{1, \beta^2\}$	$\{1, \gamma\}$	39
1	2	1	D	$\{\alpha, \alpha^2\}$	$\{1, \beta^2\}$	$\{1, \gamma^2\}$	40
1	2	1	D	$\{\alpha, \alpha^2\}$	$\{1, \beta^2\}$	$\{\gamma, \gamma^2\}$	40
1	2	1	D	$\{\alpha, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{1, \gamma\}$	35
1	2	1	D	$\{\alpha, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{1, \gamma^2\}$	37
1	2	1	D	$\{\alpha, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{\gamma, \gamma^2\}$	35
1	2	1	E	$\{1, \alpha\}$	$\{1, \beta\}$	$\{1, \gamma\}$	34

1	2	1	E	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	37
1	2	1	E	$\{1,\alpha\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	37
1	2	1	E	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	41
1	2	1	E	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	35
1	2	1	E	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	42
1	2	1	E	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	39
1	2	1	E	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	36
1	2	1	E	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	40
1	2	1	E	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	32
1	2	1	E	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	42
1	2	1	E	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	43
1	2	1	E	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	33
1	2	1	E	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	37
1	2	1	E	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	42
1	2	1	E	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	35
1	2	1	E	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	38
1	2	1	E	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	41
1	2	1	E	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	33
1	2	1	E	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	45
1	2	1	E	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	41
1	2	1	E	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	33
1	2	1	E	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	36
1	2	1	E	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	35
1	2	1	E	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	38
1	2	1	E	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	41
1	2	1	E	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	42
1	2	1	F	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma\}$	32
1	2	1	F	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	38
1	2	1	F	$\{1,\alpha\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	37
1	2	1	F	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	30
1	2	1	F	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	35
1	2	1	F	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	37
1	2	1	F	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	39
1	2	1	F	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	37

1	2	1	F	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	35
1	2	1	F	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	37
1	2	1	F	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	46
1	2	1	F	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	44
1	2	1	F	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	36
1	2	1	F	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	40
1	2	1	F	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	38
1	2	1	F	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	31
1	2	1	F	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	41
1	2	1	F	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	38
1	2	1	F	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	32
1	2	1	F	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	42
1	2	1	F	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	38
1	2	1	F	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	37
1	2	1	F	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	38
1	2	1	F	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	35
1	2	1	F	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	32
1	2	1	F	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	39
1	2	1	F	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	37
1	3	1	8	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma\}$	34
1	3	1	8	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	35
1	3	1	8	$\{1,\alpha\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	40
1	3	1	8	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	35
1	3	1	8	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	37
1	3	1	8	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	36
1	3	1	8	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	36
1	3	1	8	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	39
1	3	1	8	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	36
1	3	1	8	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	35
1	3	1	8	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	41
1	3	1	8	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	38
1	3	1	8	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	33
1	3	1	8	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	40
1	3	1	8	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	36

1	3	1	8	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	31
1	3	1	8	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	38
1	3	1	8	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	38
1	3	1	8	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	40
1	3	1	8	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	42
1	3	1	8	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	46
1	3	1	8	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	35
1	3	1	8	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	41
1	3	1	8	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	42
1	3	1	8	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	33
1	3	1	8	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	39
1	3	1	8	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	39
1	3	1	9	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma\}$	38
1	3	1	9	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	34
1	3	1	9	$\{1,\alpha\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	35
1	3	1	9	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	36
1	3	1	9	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	40
1	3	1	9	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	39
1	3	1	9	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	42
1	3	1	9	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	35
1	3	1	9	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	36
1	3	1	9	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	34
1	3	1	9	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	41
1	3	1	9	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	41
1	3	1	9	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	39
1	3	1	9	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	40
1	3	1	9	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	40
1	3	1	9	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	35
1	3	1	9	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	37
1	3	1	9	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	35
1	3	1	9	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	38
1	3	1	9	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	43
1	3	1	9	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	45
1	3	1	9	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	32

1	3	1	9	$\{\alpha, \alpha^2\}$	$\{1, \beta^2\}$	$\{1, \gamma^2\}$	43
1	3	1	9	$\{\alpha, \alpha^2\}$	$\{1, \beta^2\}$	$\{\gamma, \gamma^2\}$	36
1	3	1	9	$\{\alpha, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{1, \gamma\}$	37
1	3	1	9	$\{\alpha, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{1, \gamma^2\}$	40
1	3	1	9	$\{\alpha, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{\gamma, \gamma^2\}$	42
1	3	1	A	$\{1, \alpha\}$	$\{1, \beta\}$	$\{1, \gamma\}$	30
1	3	1	A	$\{1, \alpha\}$	$\{1, \beta\}$	$\{1, \gamma^2\}$	35
1	3	1	A	$\{1, \alpha\}$	$\{1, \beta\}$	$\{\gamma, \gamma^2\}$	35
1	3	1	A	$\{1, \alpha\}$	$\{1, \beta^2\}$	$\{1, \gamma\}$	29
1	3	1	A	$\{1, \alpha\}$	$\{1, \beta^2\}$	$\{1, \gamma^2\}$	36
1	3	1	A	$\{1, \alpha\}$	$\{1, \beta^2\}$	$\{\gamma, \gamma^2\}$	35
1	3	1	A	$\{1, \alpha\}$	$\{\beta, \beta^2\}$	$\{1, \gamma\}$	34
1	3	1	A	$\{1, \alpha\}$	$\{\beta, \beta^2\}$	$\{1, \gamma^2\}$	35
1	3	1	A	$\{1, \alpha\}$	$\{\beta, \beta^2\}$	$\{\gamma, \gamma^2\}$	33
1	3	1	A	$\{1, \alpha^2\}$	$\{1, \beta\}$	$\{1, \gamma\}$	34
1	3	1	A	$\{1, \alpha^2\}$	$\{1, \beta\}$	$\{1, \gamma^2\}$	38
1	3	1	A	$\{1, \alpha^2\}$	$\{1, \beta\}$	$\{\gamma, \gamma^2\}$	42
1	3	1	A	$\{1, \alpha^2\}$	$\{1, \beta^2\}$	$\{1, \gamma\}$	36
1	3	1	A	$\{1, \alpha^2\}$	$\{1, \beta^2\}$	$\{1, \gamma^2\}$	35
1	3	1	A	$\{1, \alpha^2\}$	$\{1, \beta^2\}$	$\{\gamma, \gamma^2\}$	38
1	3	1	A	$\{1, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{1, \gamma\}$	29
1	3	1	A	$\{1, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{1, \gamma^2\}$	37
1	3	1	A	$\{1, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{\gamma, \gamma^2\}$	39
1	3	1	A	$\{\alpha, \alpha^2\}$	$\{1, \beta\}$	$\{1, \gamma\}$	39
1	3	1	A	$\{\alpha, \alpha^2\}$	$\{1, \beta\}$	$\{1, \gamma^2\}$	44
1	3	1	A	$\{\alpha, \alpha^2\}$	$\{1, \beta\}$	$\{\gamma, \gamma^2\}$	46
1	3	1	A	$\{\alpha, \alpha^2\}$	$\{1, \beta^2\}$	$\{1, \gamma\}$	34
1	3	1	A	$\{\alpha, \alpha^2\}$	$\{1, \beta^2\}$	$\{1, \gamma^2\}$	38
1	3	1	A	$\{\alpha, \alpha^2\}$	$\{1, \beta^2\}$	$\{\gamma, \gamma^2\}$	40
1	3	1	A	$\{\alpha, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{1, \gamma\}$	32
1	3	1	A	$\{\alpha, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{1, \gamma^2\}$	38
1	3	1	A	$\{\alpha, \alpha^2\}$	$\{\beta, \beta^2\}$	$\{\gamma, \gamma^2\}$	41
1	3	1	B	$\{1, \alpha\}$	$\{1, \beta\}$	$\{1, \gamma\}$	41
1	3	1	B	$\{1, \alpha\}$	$\{1, \beta\}$	$\{1, \gamma^2\}$	35

1	3	1	B	$\{1,\alpha\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	36
1	3	1	B	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	38
1	3	1	B	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	38
1	3	1	B	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	37
1	3	1	B	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	40
1	3	1	B	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	37
1	3	1	B	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	37
1	3	1	B	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	33
1	3	1	B	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	45
1	3	1	B	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	41
1	3	1	B	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	33
1	3	1	B	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	35
1	3	1	B	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	37
1	3	1	B	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	38
1	3	1	B	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	41
1	3	1	B	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	42
1	3	1	B	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	32
1	3	1	B	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	43
1	3	1	B	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	43
1	3	1	B	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	33
1	3	1	B	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	37
1	3	1	B	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	42
1	3	1	B	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	35
1	3	1	B	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	38
1	3	1	B	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	41
1	3	1	C	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma\}$	41
1	3	1	C	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	35
1	3	1	C	$\{1,\alpha\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	34
1	3	1	C	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	34
1	3	1	C	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	35
1	3	1	C	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	35
1	3	1	C	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	32
1	3	1	C	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	39
1	3	1	C	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	37

1	3	1	C	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	36
1	3	1	C	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	44
1	3	1	C	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	41
1	3	1	C	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	33
1	3	1	C	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	40
1	3	1	C	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	39
1	3	1	C	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	36
1	3	1	C	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	36
1	3	1	C	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	36
1	3	1	C	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	38
1	3	1	C	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	45
1	3	1	C	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	43
1	3	1	C	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	34
1	3	1	C	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	42
1	3	1	C	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	38
1	3	1	C	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	33
1	3	1	C	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	40
1	3	1	C	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	38
1	3	1	D	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma\}$	35
1	3	1	D	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	38
1	3	1	D	$\{1,\alpha\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	34
1	3	1	D	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	31
1	3	1	D	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	40
1	3	1	D	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	36
1	3	1	D	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	35
1	3	1	D	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	41
1	3	1	D	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	37
1	3	1	D	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	35
1	3	1	D	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	42
1	3	1	D	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	44
1	3	1	D	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	35
1	3	1	D	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	36
1	3	1	D	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	38
1	3	1	D	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	37

1	3	1	D	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	37
1	3	1	D	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	40
1	3	1	D	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	40
1	3	1	D	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	43
1	3	1	D	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	46
1	3	1	D	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	39
1	3	1	D	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	35
1	3	1	D	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	40
1	3	1	D	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	36
1	3	1	D	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	39
1	3	1	D	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	43
1	3	1	E	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma\}$	34
1	3	1	E	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	39
1	3	1	E	$\{1,\alpha\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	36
1	3	1	E	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	31
1	3	1	E	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	41
1	3	1	E	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	35
1	3	1	E	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	36
1	3	1	E	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	37
1	3	1	E	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	37
1	3	1	E	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	34
1	3	1	E	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	40
1	3	1	E	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	40
1	3	1	E	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	40
1	3	1	E	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	38
1	3	1	E	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	39
1	3	1	E	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	37
1	3	1	E	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	35
1	3	1	E	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	38
1	3	1	E	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	41
1	3	1	E	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	47
1	3	1	E	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	44
1	3	1	E	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	37
1	3	1	E	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	41

1	3	1	E	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	41
1	3	1	E	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	45
1	3	1	E	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	36
1	3	1	E	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	36
1	3	1	F	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma\}$	41
1	3	1	F	$\{1,\alpha\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	39
1	3	1	F	$\{1,\alpha\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	34
1	3	1	F	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	33
1	3	1	F	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	37
1	3	1	F	$\{1,\alpha\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	34
1	3	1	F	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	31
1	3	1	F	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	39
1	3	1	F	$\{1,\alpha\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	35
1	3	1	F	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	35
1	3	1	F	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	38
1	3	1	F	$\{1,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	43
1	3	1	F	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	34
1	3	1	F	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	38
1	3	1	F	$\{1,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	39
1	3	1	F	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	37
1	3	1	F	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	40
1	3	1	F	$\{1,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	42
1	3	1	F	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma\}$	37
1	3	1	F	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{1,\gamma^2\}$	47
1	3	1	F	$\{\alpha,\alpha^2\}$	$\{1,\beta\}$	$\{\gamma,\gamma^2\}$	51
1	3	1	F	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma\}$	34
1	3	1	F	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{1,\gamma^2\}$	45
1	3	1	F	$\{\alpha,\alpha^2\}$	$\{1,\beta^2\}$	$\{\gamma,\gamma^2\}$	38
1	3	1	F	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma\}$	33
1	3	1	F	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{1,\gamma^2\}$	37
1	3	1	F	$\{\alpha,\alpha^2\}$	$\{\beta,\beta^2\}$	$\{\gamma,\gamma^2\}$	41

Appendix B

Further Implementation Results

B.1 Introduction

In the following sections, further implementation results are provided for reference. For some all evaluated Xilinx devices, the results are similar and do not significantly change the overall picture. As expected, the low-end devices (Spartan-3, Spartan-6, and Artix-7) have a significantly lower maximum throughput and thus, the throughput-area ratio is also reduced. Furthermore, the large technological difference between the Spartan-3 and all other devices leads to a higher slice count for this series.

B.2 Spartan-3

Table B.1: Results for the 256 bits versions of the SHA-3 finalists (Spartan-3).

Name	Area	Frequency	Clock	Input size	Long TP	TP-area	Short TP	TP-area
	[Slices]	[MHz]	Cycles	[Bits]	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
BLAKE	1753	78	115	512	347	0.20	347	0.20
BLAKE-2	974	94	228	512	212	0.22	212	0.22
Grøstl	1240	167	160	512	534	0.43	267	0.22
JH	2111	116	168	512	354	0.17	177	0.08
JH-2	829	140	6720	512	11	0.01	5	0.01
Keccak	1696	76	200	1088	413	0.24	413	0.24
Skein	1407	128	584	512	112	0.08	56	0.04

Table B.2: Results for the heavyweight versions of Keccak (Spartan-3).

Name	Capacity [Bits]	Parallel Slices	Area [Slices]	Frequency [MHz]	Clock Cycles	Input size [Bits]	Long TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
Keccak-f[1600]	512	1	583	78	2688	1088	31	0.05
		2	633	77	1344	1088	62	0.10
		4	638	76	672	1088	123	0.19
		8	990	72	336	1088	234	0.24
		16	1608	67	168	1088	436	0.27
		32	2967	59	84	1088	770	0.26
		64	4226	74	42	1088	1916	0.45
Keccak-f[800]	512	1	439	79	1024	288	22	0.05
		2	463	80	512	288	45	0.10
		4	583	76	256	288	85	0.15
		8	916	69	128	288	155	0.17
		16	1518	64	64	288	289	0.19
		32	2180	87	32	288	779	0.36
	256	1	460	84	1280	544	36	0.08
		2	466	82	640	544	70	0.15
		4	569	78	320	544	133	0.23
		8	930	70	160	544	238	0.26
		16	1584	66	80	544	451	0.28
		32	2139	84	40	544	1147	0.54

Table B.3: Results for the lightweight versions of KECCAK (Spartan-3).

Name	Digest [Bits]	Capacity [Bits]	Parallel Slices	Area [Slices]	Frequency [MHz]	Clock Cycles	Input size [Bits]	Long TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	Short TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
KECCAK-f[400]	128	256	1	321	86	480	144	26	0.08	26	0.08
			2	394	81	240	144	49	0.12	49	0.12
			4	567	76	120	144	91	0.16	91	0.16
			8	881	71	60	144	169	0.19	169	0.19
			16	1051	91	30	144	437	0.42	437	0.42
	128	128	1	375	99	608	272	44	0.12	44	0.12
			2	386	84	304	272	75	0.19	75	0.19
			4	553	76	152	272	136	0.25	136	0.25
			8	907	71	76	272	256	0.28	256	0.28
			16	1107	100	38	272	715	0.65	715	0.65
	160	160	1	310	84	576	240	35	0.11	35	0.11
			2	381	82	288	240	68	0.18	68	0.18
			4	541	74	144	240	123	0.23	123	0.23
			8	864	69	72	240	230	0.27	230	0.27
			16	1045	99	36	240	663	0.63	663	0.63
	160	320	1	327	88	416	80	17	0.052	8	0.026
			2	373	75	208	80	29	0.078	14	0.039
			4	555	73	104	80	56	0.101	28	0.051
			8	889	70	52	80	108	0.122	54	0.061
			16	1001	96	26	80	296	0.295	148	0.148
	224	224	1	335	80	512	176	27	0.097	14	0.041
			2	408	76	256	176	52	0.146	26	0.064
			4	574	69	128	176	95	0.169	47	0.083
			8	889	64	64	176	177	0.190	88	0.099
			16	1141	87	32	176	476	0.357	238	0.209
	256	256	1	327	84	480	144	25	0.077	13	0.038
			2	407	82	240	144	49	0.149	25	0.060
			4	560	78	120	144	94	0.234	47	0.084
			8	889	70	60	144	168	0.256	84	0.094
			16	1132	66	30	144	318	0.285	159	0.141
KECCAK-f[200]	128	128	1	255	83	224	72	27	0.105	13	0.052
			2	375	80	112	72	51	0.137	26	0.068
			4	499	74	56	72	95	0.191	48	0.096
			8	534	92	28	72	237	0.444	118	0.222
	160	160	1	251	85	192	40	18	0.071	4	0.018
			2	353	77	96	40	32	0.091	8	0.023
			4	492	76	48	40	63	0.129	16	0.032
			8	533	91	24	40	152	0.285	38	0.071

Table B.4: Results for the Photon hash function family (Spartan-3).

Name	Area [Slices]	Frequency [MHz]	Clock Cycles	Input size [Bits]	Long TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	Short TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
Photon-80/20/16	295	80	245	20	6.57	0.0223	1.31	0.0045
Photon-128/10/16	343	84	436	16	3.10	0.0090	0.39	0.0011
Photon-160/36/36	331	66	597	36	3.97	0.0120	0.89	0.0027
Photon-224/32/32	308	90	776	32	3.71	0.0120	0.53	0.0017
Photon-256/32/32	524	49	436	32	3.57	0.0068	0.45	0.0009

B.3 Spartan-6

Table B.5: Results for the 256 bits versions of the SHA-3 finalists (Spartan-6).

Name	Area	Frequency	Clock	Input size	Long TP	TP-area	Short TP	TP-area
	[Slices]	[MHz]	Cycles	[Bits]	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
BLAKE	412	115	115	512	511	1.24	511	1.24
BLAKE-2	239	145	228	512	326	1.36	326	1.36
Grøstl	333	238	160	512	763	2.29	381	1.15
JH	478	161	168	512	489	1.02	245	0.51
JH-2	170	240	6720	512	18	0.11	9	0.05
KECCAK	417	117	200	1088	636	1.52	636	1.52
Skein	503	226	584	512	198	0.39	99	0.20

Table B.6: Results for the heavyweight versions of KECCAK (Spartan-6).

Name	Capacity	Parallel	Area	Frequency	Clock	Input size	Long TP	TP-area
	[Bits]	Slices	[Slices]	[MHz]	Cycles	[Bits]	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
KECCAK-f[1600]	512	1	160	144	2688	1088	58	0.36
		2	194	131	1344	1088	106	0.55
		4	221	129	672	1088	209	0.94
		8	288	120	336	1088	390	1.35
		16	526	134	168	1088	870	1.65
		32	909	124	84	1088	1610	1.77
		64	1378	122	42	1088	3155	2.29
KECCAK-f[800]	512	1	150	141	1024	288	40	0.26
		2	150	132	512	288	74	0.50
		4	183	126	256	288	142	0.78
		8	263	117	128	288	264	1.00
		16	426	113	64	288	509	1.19
		32	743	184	32	288	1658	2.23
	256	1	153	153	1280	544	65	0.42
		2	165	148	640	544	125	0.76
		4	194	128	320	544	218	1.12
		8	327	142	160	544	482	1.47
		16	462	122	80	544	826	1.79
		32	727	169	40	544	2292	3.15

Table B.7: Results for the lightweight versions of Keccak (Spartan-6).

Name	Digest	Capacity	Parallel	Area	Frequency	Clock	Input size	Long TP	TP-area	Short TP	TP-area
	[Bits]	[Bits]	Slices	[Slices]	[MHz]	Cycles	[Bits]	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
Keccak-f[400]	128	256	1	118	159	480	144	48	0.41	48	0.41
			2	159	164	240	144	98	0.62	98	0.62
			4	190	128	120	144	154	0.81	154	0.81
			8	251	125	60	144	301	1.20	301	1.20
			16	311	169	30	144	809	2.60	809	2.60
	128	128	1	124	158	608	272	71	0.57	71	0.57
			2	150	138	304	272	123	0.82	123	0.82
			4	205	136	152	272	244	1.19	244	1.19
			8	264	130	76	272	467	1.77	467	1.77
			16	345	176	38	272	1258	3.65	1258	3.65
	160	160	1	124	159	576	240	66	0.54	66	0.54
			2	148	142	288	240	118	0.80	118	0.80
			4	198	136	144	240	227	1.14	227	1.14
			8	263	131	72	240	437	1.66	437	1.66
			16	347	213	36	240	1420	4.09	1420	4.09
	160	320	1	115	158	416	80	30	0.265	15	0.132
			2	130	135	208	80	52	0.399	26	0.200
			4	188	125	104	80	96	0.512	48	0.256
			8	245	124	52	80	191	0.778	95	0.389
			16	348	226	26	80	696	1.999	348	1.000
	224	224	1	124	132	512	176	45	0.496	23	0.183
			2	141	126	256	176	87	0.777	43	0.308
			4	201	117	128	176	161	1.005	81	0.402
			8	256	113	64	176	311	1.194	155	0.607
			16	329	184	32	176	1013	2.231	507	1.540
	256	256	1	123	153	480	144	46	0.425	23	0.186
			2	135	148	240	144	89	0.760	44	0.328
			4	196	128	120	144	154	1.121	77	0.392
			8	252	142	60	144	340	1.475	170	0.675
			16	328	122	30	144	583	1.788	292	0.889
Keccak-f[200]	128	128	1	103	161	224	72	52	0.502	26	0.251
			2	137	140	112	72	90	0.657	45	0.329
			4	166	131	56	72	168	1.011	84	0.505
			8	173	176	28	72	452	2.613	226	1.307
	160	160	1	91	152	192	40	32	0.348	8	0.087
			2	120	130	96	40	54	0.452	14	0.113
			4	164	128	48	40	107	0.651	27	0.163
			8	171	194	24	40	323	1.886	81	0.472

Table B.8: Results for the Photon hash function family (Spartan-6).

Name	Area	Frequency	Clock	Input size	Long TP	TP-area	Short TP	TP-area
	[Slices]	[MHz]	Cycles	[Bits]	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
Photon-80/20/16	73	110	245	20	8.98	0.1230	1.80	0.0246
Photon-128/10/16	89	146	436	16	5.34	0.0601	0.67	0.0075
Photon-160/36/36	102	122	597	36	7.37	0.0722	1.47	0.0144
Photon-224/32/32	96	146	776	32	6.03	0.0628	0.86	0.0090
Photon-256/32/32	159	78	436	32	5.71	0.0359	0.71	0.0045

B.4 Virtex-6

Table B.9: Results for the 256 bits versions of the SHA-3 finalists (Virtex-6).

Name	Area [Slices]	Frequency [MHz]	Clock Cycles	Input size [Bits]	Long TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	Short TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
BLAKE	414	210	115	512	749	1.81	749	1.81
BLAKE-2	278	245	228	512	609	2.19	609	2.19
Grøstl	355	383	160	512	1211	3.41	606	1.71
JH	498	354	168	512	898	1.80	449	0.90
JH-2	220	419	6720	512	32	0.15	16	0.07
KECCAK	435	252	200	1088	1004	2.31	1004	2.31
Skein	464	356	584	512	288	0.62	144	0.31

Table B.10: Results for the heavyweight versions of KECCAK (Virtex-6).

Name	Capacity [Bits]	Parallel Slices	Area [Slices]	Frequency [MHz]	Clock Cycles	Input size [Bits]	Long TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
KECCAK-f[1600]	512	1	164	257	2688	1088	104	0.64
		2	193	206	1344	1088	167	0.87
		4	194	190	672	1088	308	1.59
		8	263	187	336	1088	606	2.30
		16	457	165	168	1088	1068	2.34
		32	961	185	84	1088	2390	2.49
		64	1360	201	42	1088	5195	3.82
KECCAK-f[800]	512	1	156	251	1024	288	71	0.45
		2	159	252	512	288	142	0.89
		4	164	202	256	288	227	1.38
		8	308	233	128	288	524	1.70
		16	496	227	64	288	1023	2.06
		32	641	221	32	288	1992	3.11
	256	1	148	255	1280	544	108	0.73
		2	204	271	640	544	230	1.13
		4	178	192	320	544	326	1.83
		8	290	228	160	544	776	2.68
		16	421	187	80	544	1274	3.03
		32	664	210	40	544	2853	4.30

Table B.11: Results for the lightweight versions of KECCAK (Virtex-6).

Name	Digest	Capacity	Parallel	Area	Frequency	Clock	Input size	Long TP	TP-area	Short TP	TP-area
	[Bits]	[Bits]	Slices	[Slices]	[MHz]	Cycles	[Bits]	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
KECCAK-f[400]	128	256	1	110	216	480	144	65	0.59	65	0.59
			2	123	213	240	144	128	1.04	128	1.04
			4	207	253	120	144	303	1.47	303	1.47
			8	261	242	60	144	582	2.23	582	2.23
			16	324	243	30	144	1166	3.60	1166	3.60
	128	128	1	120	258	608	272	115	0.96	115	0.96
			2	156	260	304	272	233	1.49	233	1.49
			4	195	222	152	272	398	2.04	398	2.04
			8	277	244	76	272	875	3.16	875	3.16
			16	343	247	38	272	1770	5.16	1770	5.16
	160	160	1	112	212	576	240	89	0.79	89	0.79
			2	152	262	288	240	218	1.43	218	1.43
			4	200	252	144	240	420	2.10	420	2.10
			8	244	198	72	240	661	2.71	661	2.71
			16	322	237	36	240	1581	4.91	1581	4.91
	160	320	1	317	264	416	80	51	0.16	25	0.08
			2	128	252	208	80	97	0.76	48	0.38
			4	137	256	104	80	197	1.44	99	0.72
			8	202	255	52	80	392	1.94	196	0.97
			16	227	206	26	80	635	2.80	317	1.40
	224	224	1	126	252	512	176	87	0.69	43	0.34
			2	143	202	256	176	139	0.97	69	0.48
			4	203	233	128	176	320	1.58	160	0.79
			8	316	227	64	176	625	1.98	313	0.99
			16	369	221	32	176	1217	3.30	609	1.65
	256	256	1	123	255	480	144	76	0.62	38	0.31
			2	159	271	240	144	162	1.02	81	0.51
			4	196	192	120	144	230	1.18	115	0.59
			8	227	228	60	144	548	2.41	274	1.21
			16	385	187	30	144	899	2.34	450	1.17
KECCAK-f[200]	128	128	1	98	260	224	72	84	0.85	42	0.43
			2	138	265	112	72	170	1.23	85	0.62
			4	162	224	56	72	288	1.78	144	0.89
			8	172	343	28	72	883	5.13	441	2.57
	160	160	1	93	263	192	40	55	0.59	14	0.15
			2	125	251	96	40	105	0.84	26	0.21
			4	171	266	48	40	222	1.30	55	0.32
			8	187	357	24	40	596	3.19	149	0.80

Table B.12: Results for the Photon hash function family (Virtex-6).

Name	Area	Frequency	Clock	Input size	Long TP	TP-area	Short TP	TP-area
	[Slices]	[MHz]	Cycles	[Bits]	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
Photon-80/20/16	89	250	245	20	20	0.23	4.08	0.046
Photon-128/10/16	86	230	436	16	8	0.10	1.06	0.012
Photon-160/36/36	122	233	597	36	14	0.12	2.82	0.023
Photon-224/32/32	101	255	776	32	11	0.10	1.50	0.015
Photon-256/32/32	179	150	436	32	11	0.06	1.37	0.008

B.5 Artix-7

Table B.13: Results for the 256 bits versions of the SHA-3 finalists (Artix-7).

Name	Area	Frequency	Clock	Input size	Long TP	TP-area	Short TP	TP-area
	[Slices]	[MHz]	Cycles	[Bits]	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
BLAKE	443	143	115	512	637	1.44	637	1.44
BLAKE-2	261	174	228	512	390	1.50	390	1.50
Grøstl	395	282	160	512	901	2.28	451	1.14
JH	514	234	168	512	713	1.39	356	0.69
JH-2	186	311	6720	512	24	0.13	12	0.06
Keccak	482	180	200	1088	980	2.03	980	2.03
Skein	469	226	584	512	198	0.42	99	0.21

Table B.14: Results for the heavyweight versions of Keccak (Artix-7).

Name	Capacity	Parallel	Area	Frequency	Clock	Input size	Long TP	TP-area
	[Bits]	Slices	[Slices]	[MHz]	Cycles	[Bits]	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
Keccak-f[1600]	512	1	172	179	2688	1088	73	0.42
		2	207	159	1344	1088	129	0.62
		4	247	182	672	1088	294	1.19
		8	293	145	336	1088	471	1.61
		16	463	150	168	1088	970	2.09
		32	900	138	84	1088	1792	1.99
		64	1359	143	42	1088	3694	2.72
Keccak-f[800]	512	1	159	187	1024	288	53	0.33
		2	158	163	512	288	92	0.58
		4	220	191	256	288	215	0.98
		8	259	148	128	288	334	1.29
		16	409	146	64	288	656	1.60
		32	633	191	32	288	1717	2.71
	256	1	165	191	1280	544	81	0.49
		2	188	197	640	544	167	0.89
		4	192	154	320	544	263	1.37
		8	276	155	160	544	526	1.91
		16	443	143	80	544	970	2.19
		32	699	216	40	544	2935	4.20

Table B.15: Results for the lightweight versions of KECCAK (Artix-7).

Name	Digest [Bits]	Capacity [Bits]	Parallel Slices	Area [Slices]	Frequency [MHz]	Clock Cycles	Input size [Bits]	Long TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	Short TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
KECCAK-f[400]	128	256	1	129	189	480	144	57	0.44	57	0.44
			2	165	204	240	144	122	0.74	122	0.74
			4	185	161	120	144	193	1.04	193	1.04
			8	243	152	60	144	365	1.50	365	1.50
			16	322	220	30	144	1056	3.28	1056	3.28
	128	128	1	141	193	608	272	86	0.61	86	0.61
			2	144	160	304	272	143	1.00	143	1.00
			4	201	156	152	272	279	1.39	279	1.39
			8	264	156	76	272	558	2.11	558	2.11
			16	343	218	38	272	1561	4.55	1561	4.55
	160	160	1	129	193	576	240	80	0.62	80	0.62
			2	141	157	288	240	131	0.93	131	0.93
			4	189	162	144	240	270	1.43	270	1.43
			8	249	150	72	240	500	2.01	500	2.01
			16	326	217	36	240	1447	4.44	1447	4.44
	160	320	1	139	199	416	80	38	0.27	19	0.14
			2	159	199	208	80	77	0.48	38	0.24
			4	182	164	104	80	126	0.69	63	0.35
			8	239	157	52	80	241	1.01	121	0.50
			16	320	219	26	80	675	2.11	337	1.05
	224	224	1	130	163	512	176	56	0.58	28	0.22
			2	168	191	256	176	131	0.98	66	0.39
			4	192	148	128	176	204	1.29	102	0.53
			8	252	146	64	176	401	1.60	200	0.80
			16	328	191	32	176	1049	2.71	525	1.60
	256	256	1	131	191	480	144	57	0.49	29	0.22
			2	132	197	240	144	118	0.89	59	0.45
			4	184	154	120	144	185	1.37	93	0.50
			8	248	155	60	144	372	1.91	186	0.75
			16	349	143	30	144	685	2.19	343 302	0.98
KECCAK-f[200]	128	128	1	99	173	224	72	56	0.56	28	0.28
			2	127	177	112	72	114	0.90	57	0.45
			4	164	171	56	72	220	1.34	110	0.67
			8	173	272	28	72	700	4.04	350	2.02
	160	160	1	103	193	192	40	40	0.39	10	0.10
			2	119	177	96	40	74	0.62	18	0.16
			4	158	172	48	40	143	0.91	36	0.23
			8	167	240	24	40	400	2.40	100	0.60

Table B.16: Results for the Photon hash function family (Artix-7).

Name	Area [Slices]	Frequency [MHz]	Clock Cycles	Input size [Bits]	Long TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	Short TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
Photon-80/20/16	98	184	245	20	15.04	0.15	3.01	0.031
Photon-128/16/16	96	173	436	16	6.35	0.07	0.79	0.008
Photon-160/36/36	123	160	597	36	9.65	0.08	1.93	0.016
Photon-224/32/32	111	193	776	32	7.98	0.07	1.14	0.010
Photon-256/32/32	187	90	436	32	6.59	0.04	0.82	0.004

B.6 Kintex-7

Table B.17: Results for the 256 bits versions of the SHA-3 finalists (Kintex-7).

Name	Area	Frequency	Clock	Input size	Long TP	TP-area	Short TP	TP-area
	[Slices]	[MHz]	Cycles	[Bits]	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
BLAKE	444	201	115	512	897	2.02	897	2.02
BLAKE-2	267	248	228	512	557	2.09	557	2.09
Grøstl	400	417	160	512	1335	3.34	668	1.67
JH	536	336	168	512	1025	1.91	513	0.96
JH-2	191	429	6720	512	33	0.17	16	0.09
KECCAK	475	216	200	1088	1177	2.48	1177	2.48
Skein	467	362	584	512	317	0.68	159	0.34

Table B.18: Results for the heavyweight versions of KECCAK (Kintex-7).

Name	Capacity	Parallel	Area	Frequency	Clock	Input size	Long TP	TP-area
	[Bits]	Slices	[Slices]	[MHz]	Cycles	[Bits]	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
KECCAK-f[1600]	512	1	171	248	2688	1088	100	0.59
		2	211	257	1344	1088	208	0.99
		4	221	207	672	1088	335	1.51
		8	294	193	336	1088	625	2.13
		16	568	227	168	1088	1471	2.59
		32	896	179	84	1088	2320	2.59
		64	1433	262	42	1088	6783	4.73
KECCAK-f[800]	512	1	147	253	1024	288	71	0.48
		2	176	284	512	288	160	0.91
		4	228	283	256	288	318	1.40
		8	262	209	128	288	471	1.80
		16	410	185	64	288	835	2.04
		32	637	295	32	288	2652	4.16
	256	1	152	259	1280	544	110	0.72
		2	192	283	640	544	241	1.25
		4	244	275	320	544	467	1.92
		8	277	213	160	544	726	2.62
		16	443	178	80	544	1209	2.73
		32	668	292	40	544	3973	5.95

Table B.19: Results for the lightweight versions of KECCAK (Kintex-7).

Name	Digest [Bits]	Capacity [Bits]	Parallel Slices	Area [Slices]	Frequency [MHz]	Clock Cycles	Input size [Bits]	Long TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	Short TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
KECCAK-f[400]	128	256	1	130	230	480	144	69	0.53	69	0.53
			2	161	291	240	144	175	1.08	175	1.08
			4	231	284	120	144	340	1.47	340	1.47
			8	246	206	60	144	494	2.01	494	2.01
			16	322	329	30	144	1579	4.91	1579	4.91
	128	128	1	134	284	608	272	127	0.95	127	0.95
			2	178	284	304	272	254	1.43	254	1.43
			4	246	282	152	272	505	2.05	505	2.05
			8	264	213	76	272	761	2.88	761	2.88
			16	344	334	38	272	2388	6.94	2388	6.94
	160	160	1	130	284	576	240	118	0.91	118	0.91
			2	172	289	288	240	241	1.40	241	1.40
			4	190	235	144	240	391	2.06	391	2.06
			8	252	209	72	240	695	2.76	695	2.76
			16	325	323	36	240	2153	6.63	2153	6.63
	160	320	1	129	233	416	80	45	0.35	22	0.17
			2	163	292	208	80	112	0.69	56	0.34
			4	187	235	104	80	181	0.97	90	0.48
			8	240	214	52	80	329	1.37	164	0.69
			16	317	320	26	80	986	3.11	493	1.55
	224	224	1	129	284	512	176	98	0.76	49	0.38
			2	141	283	256	176	194	1.38	97	0.69
			4	235	209	128	176	288	1.23	144	0.61
			8	253	185	64	176	510	2.02	255	1.01
			16	334	295	32	176	1621	4.85	810	2.43
	256	256	1	130	259	480	144	78	0.60	39	0.30
			2	168	283	240	144	170	1.01	85	0.51
			4	233	275	120	144	330	1.42	165	0.71
			8	253	213	60	144	512	2.02	256	1.01
			16	332	178	30	144	854	2.57	427	1.29
KECCAK-f[200]	128	128	1	105	276	224	72	89	0.85	44	0.42
			2	128	266	112	72	171	1.34	85	0.67
			4	162	241	56	72	310	1.91	155	0.96
			8	175	393	28	72	1011	5.78	505	2.89
	160	160	1	101	271	192	40	56	0.56	14	0.14
			2	126	269	96	40	112	0.89	28	0.22
			4	158	243	48	40	203	1.28	51	0.32
			8	169	357	24	40	595	3.52	149	0.88

Table B.20: Results for the Photon hash function family (Kintex-7).

Name	Area [Slices]	Frequency [MHz]	Clock Cycles	Input size [Bits]	Long TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	Short TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
Photon-80/20/16	89	248	245	20	20	0.23	4.05	0.045
Photon-128/16/16	95	245	436	16	9	0.09	1.13	0.012
Photon-160/36/36	125	241	597	36	15	0.12	2.90	0.023
Photon-224/32/32	104	281	776	32	12	0.11	1.66	0.016
Photon-256/32/32	243	171	436	32	13	0.05	1.57	0.006

B.7 Virtex-7

Table B.21: Results for the 256 bits versions of the SHA-3 finalists (Virtex-7).

Name	Area [Slices]	Frequency [MHz]	Clock Cycles	Input size [Bits]	Long TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	Short TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
BLAKE	448	210	115	512	935	2.09	935	2.09
BLAKE-2	261	245	228	512	551	2.11	551	2.11
Grøstl	390	383	160	512	1227	3.14	613	1.57
JH	539	354	168	512	1079	2.00	540	1.00
JH-2	186	419	6720	512	32	0.17	16	0.09
KECCAK	549	252	200	1088	1370	2.50	1370	2.50
Skein	474	356	584	512	312	0.66	156	0.33

Table B.22: Results for the heavyweight versions of KECCAK (Virtex-7).

Name	Capacity [Bits]	Parallel Slices	Area [Slices]	Frequency [MHz]	Clock Cycles	Input size [Bits]	Long TP [MBits/s]	TP-area $\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
KECCAK-f[1600]	512	1	170	254	2688	1088	103	0.60
		2	224	279	1344	1088	226	1.01
		4	226	215	672	1088	349	1.54
		8	296	206	336	1088	667	2.25
		16	468	198	168	1088	1282	2.74
		32	903	189	84	1088	2452	2.72
		64	1444	276	42	1088	7150	4.95
KECCAK-f[800]	512	1	163	274	1024	288	77	0.47
		2	177	294	512	288	165	0.93
		4	188	213	256	288	240	1.27
		8	339	267	128	288	601	1.77
		16	411	193	64	288	871	2.12
		32	635	290	32	288	2612	4.11
	256	1	164	270	1280	544	115	0.70
		2	193	290	640	544	246	1.28
		4	194	215	320	544	366	1.89
		8	280	207	160	544	703	2.51
		16	451	198	80	544	1346	2.98
		32	692	310	40	544	4220	6.10

Table B.23: Results for the lightweight versions of KECCAK (Virtex-7).

Name	Digest	Capacity	Parallel	Area	Frequency	Clock	Input size	Long TP	TP-area	Short TP	TP-area
	[Bits]	[Bits]	Slices	[Slices]	[MHz]	Cycles	[Bits]	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
KECCAK-f[400]	128	256	1	134	295	480	144	89	0.66	89	0.66
			2	164	295	240	144	177	1.08	177	1.08
			4	187	230	120	144	276	1.47	276	1.47
			8	250	212	60	144	509	2.03	509	2.03
			16	326	333	30	144	1598	4.90	1598	4.90
	128	128	1	134	283	608	272	127	0.94	127	0.94
			2	155	220	304	272	197	1.27	197	1.27
			4	203	226	152	272	404	1.99	404	1.99
			8	270	213	76	272	763	2.82	763	2.82
			16	344	341	38	272	2440	7.09	2440	7.09
	160	160	1	130	234	576	240	98	0.75	98	0.75
			2	175	288	288	240	240	1.37	240	1.37
			4	196	225	144	240	375	1.91	375	1.91
			8	260	214	72	240	714	2.75	714	2.75
			16	326	326	36	240	2177	6.68	2177	6.68
	160	320	1	130	278	416	80	53	0.41	27	0.21
			2	164	291	208	80	112	0.68	56	0.34
			4	188	232	104	80	179	0.95	89	0.48
			8	238	215	52	80	330	1.39	165	0.69
			16	320	332	26	80	1022	3.19	511	1.60
	224	224	1	128	294	512	176	101	0.93	51	0.40
			2	171	213	256	176	146	1.27	73	0.43
			4	197	267	128	176	367	1.77	184	0.93
			8	258	193	64	176	532	2.12	266	1.03
			16	333	290	32	176	1597	4.11	798	2.40
	256	256	1	129	270	480	144	81	0.70	40	0.31
			2	140	290	240	144	174	1.28	87	0.62
			4	189	215	120	144	258	1.89	129	0.68
			8	253	207	60	144	496	2.51	248	0.98
			16	335	198	30	144	950	2.98	475	1.42
KECCAK-f[200]	128	128	1	108	285	224	72	92	0.85	46	0.42
			2	132	271	112	72	174	1.32	87	0.66
			4	161	234	56	72	301	1.87	151	0.94
			8	175	397	28	72	1020	5.83	510	2.92
	160	160	1	102	282	192	40	59	0.58	15	0.14
			2	127	270	96	40	112	0.88	28	0.22
			4	194	301	48	40	251	1.30	63	0.32
			8	169	357	24	40	595	3.52	149	0.88

Table B.24: Results for the Photon hash function family (Virtex-7).

Name	Area	Frequency	Clock	Input size	Long TP	TP-area	Short TP	TP-area
	[Slices]	[MHz]	Cycles	[Bits]	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$	[MBits/s]	$\left[\frac{\text{MBits/s}}{\text{Slice}}\right]$
Photon-80/20/16	89	249	245	20	20	0.23	4.07	0.046
Photon-128/10/16	95	253	436	16	9	0.10	1.16	0.012
Photon-160/36/36	129	250	597	36	15	0.12	3.01	0.023
Photon-224/32/32	107	286	776	32	12	0.11	1.69	0.016
Photon-256/32/32	198	131	436	32	10	0.05	1.20	0.006

Bibliography

[AB09] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2009.

[ACK+02] G. Ausiello, P. Crescenzi, V. Kann, A. Marchetti-Spaccalmela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties.* Springer-Verlag, 2002.

[AHMP10] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan. *SHA-3 proposal BLAKE.* Submission to NIST. 2010. URL: `http://www.131002.net/blake/blake.pdf`.

[AIM10] L. Atzori, A. Iera, and G. Morabito. „The Internet of Things: A Survey". In: vol. 54. 15. Elsevier, 2010, pp. 2787–2805.

[Ash09] K. Ashton. „That 'Internet of Things' Thing". In: *RFiD Journal* 22 (2009), pp. 97–114.

[ATM+13] G. S. Athanasiou, E. Tsingkas, H. E. Michail, G. Theodoridis, and C. E. Goutis. „Throughput/Area Trade-Offs of Loop-Unrolling, Functional, and Structural Pipeline for Skein Hash Function". In: *Computer Science & Engineering* 3.1 (2013).

[Bar87] P. Barrett. „Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor". In: *Advances in Cryptology - CRYPTO '86.* Springer-Verlag, 1987, pp. 311–323.

[BBV12] S. Babbage, J. Borghoff, and V. Velichkov. *The eSTREAM Portfolio in 2012.* Tech. rep. ECRYPT II, 2012. URL: `http://www.ecrypt.eu.org/documents/D.SYM.10-v1.pdf`.

[BCK96] M. Bellare, R. Canetti, and H. Krawczyk. „Message Authentication using Hash Functions – the HMAC Construction“. In: RSA Laboratories, 1996.

[BCO04] E. Brier, C. Clavier, and F. Olivier. „Correlation Power Analysis with a Leakage Model“. In: *Cryptographic Hardware and Embedded Systems*. Springer-Verlag, 2004, pp. 16–29.

[BD07] E. Biham and O. Dunkelman. *A Framework for Iterative Hash Functions - HAIFA*. Cryptology ePrint Archive, Report 2007/278. `http://eprint.iacr.org/2007/278`. 2007.

[BDPA07] G. Bertoni, J. Daemen, M. Peeters, and G. van Assche. „Sponge functions“. In: *Ecrypt Hash Workshop*. 2007.

[BDPA08] G. Bertoni, J. Daemen, M. Peeters, and G. van Assche. „On the Indifferentiability of the Sponge Construction“. In: *Advances in Cryptology - EUROCRYPT '08*. Vol. 4965. Lecture Notes in Computer Science. Springer-Verlag, 2008, pp. 181–197.

[BDPA09] G. Bertoni, J. Daemen, M. Peeters, and G. van Assche. *Sufficient conditions for sound tree and sequential hashing modes*. Cryptology ePrint Archive, Report 2009/210. `http://eprint.iacr.org/2009/210`. 2009.

[BDPA11a] G. Bertoni, J. Daemen, M. Peeters, and G. van Assche. *The Keccak reference*. Submission to NIST. 2011. URL: `http://keccak.noekeon.org/Keccak-reference-3.0.pdf`.

[BDPA11b] G. Bertoni, J. Daemen, M. Peeters, and G. van Assche. *The Keccak SHA-3 submission*. Submission to NIST. 2011. URL: `http://keccak.noekeon.org/Keccak-submission-3.pdf`.

[BDPA11c] G. Bertoni, J. Daemen, M. Peeters, and G. van Assche. *Cryptographic Sponge Functions*. 2011. URL: `http://sponge.noekeon.org/CSF-0.1.pdf`.

[BDPA12] G. Bertoni, J. Daemen, M. Peeters, and G. van Assche. „Duplexing the sponge: single-pass authenticated encryption and other applications“. In: *Selected Areas in Cryptography*. Springer-Verlag. 2012, pp. 320–337.

[BDPA13] G. Bertoni, J. Daemen, M. Peeters, and G. van Assche. *Sakura: a flexible coding for tree hashing*. Cryptology ePrint Archive, Report 2013/231. `http://eprint.iacr.org/2013/231`. 2013.

[Ben81] M. Ben-Or. „Probabilistic Algorithms in Finite Fields“. In: *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*. IEEE, 1981, pp. 394–398.

[Ber06] D. J. Bernstein. „Curve25519: new Diffie-Hellman Speed Records“. In: *Public Key Cryptography - PKC ’06*. Springer-Verlag, 2006, pp. 207–228.

[Ber08a] D. J. Bernstein. „ChaCha, a Variant of Salsa20“. In: *Workshop Record of SASC*. 2008.

[Ber08b] D. J. Bernstein. „The Salsa20 Family of Stream Ciphers“. In: *New Stream Cipher Designs*. Vol. 4986. Lecture Notes in Computer Science. Springer-Verlag, 2008, pp. 84–97.

[Ber14] D. J. Bernstein. *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness*. `http://competitions.cr.yp.to`. 2014.

[Ber67] E. R. Berlekamp. „Factoring Polynomials Over Finite Fields“. In: *Bell System Technical Journal* 46 (1967), pp. 1853–1859.

[BHT98] G. Brassard, P. Høyer, and A. Tapp. „Quantum Cryptanalysis of Hash and Claw-Free Functions“. In: *LATIN’98: Theoretical Informatics*. Vol. 1380. Lecture Notes in Computer Science. Springer-Verlag, 1998, pp. 163–169.

[BKL+07] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. „PRESENT: An Ultra-Lightweight Block Cipher“. In: *Cryptographic Hardware and Embedded Systems - CHES ’07*. Vol. 4727. Lecture Notes in Computer Science. Springer-Verlag, 2007, pp. 450–466.

[BL12] D. J. Bernstein and T. Lange. *The new SHA-3 software shootout*. Cryptology ePrint Archive, Report 2012/004. `http://eprint.iacr.org/2012/004`. 2012.

[BOY10] J.-L. Beuchat, E. Okamoto, and T. Yamazaki. „Compact Implementations of BLAKE-32 and BLAKE-64 on FPGA“. In: *International Conference on Field-Programmable Technology.* 2010, pp. 170–177.

[BR02] L. M. Burgun and A. Raynaud. „Method and Apparatus for Gate-Level Simulation of Synthesized Register Transfer Level Design with Source-Level Debugging“. US Patent 6,336,087. 2002.

[BR89] D. Bong and C. Ruland. „Optimized Software Implementations of the Modular Exponentiation on General Purpose Microprocessors“. In: *Computers & Security* 8.7 (1989), pp. 621–630.

[BU08] D. Buchfuhrer and C. Umans. „The Complexity of Boolean Formula Minimization“. In: *Automata, Languages and Programming.* Vol. 5125. Lecture Notes in Computer Science. Springer-Verlag, 2008, pp. 24–35.

[Can05a] D. Canright. *A Very Compact Rijndael S-Box.* 2005.

[Can05b] D. Canright. „A Very Compact S-Box for AES“. In: *Cryptographic Hardware and Embedded Systems - CHES ’05.* Vol. 3659. Lecture Notes in Computer Science. Springer-Verlag, 2005, pp. 441–455.

[CB08] D. Canright and L. Batina. „A Cery Compact "Perfectly Masked" S-box for AES“. In: *Proceedings of the 6th International Conference on Applied Cryptography and Network Security.* ACNS’08. Springer-Verlag, 2008, pp. 446–459.

[CD94] J. Cong and Y. Ding. „On Area/Depth Trade-Off in LUT-based FPGA Technology Mapping“. In: vol. 2. 2. IEEE, 1994, pp. 137–148.

[CDMP05] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. „Merkle-Damgård Revisited: How to Construct a Hash Function“. In: *Advances in Cryptology - CRYPTO ’05.* Vol. 3621. Lecture Notes in Computer Science. Springer-Verlag. 2005, pp. 430–448.

[CGH04] R. Canetti, O. Goldreich, and S. Halevi. „The Random Oracle Methodology, Revisited“. In: vol. 51. 4. ACM, 2004, pp. 557–594.

[CGMT09] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. „An Introduction to High-Level Synthesis“. In: vol. 26. 4. IEEE, 2009, pp. 8–17.

[CO09] D. Canright and D. A. Osvik. „A More Compact AES“. In: *Selected Areas in Cryptography*. Vol. 5867. Lecture Notes in Computer Science. Springer-Verlag, 2009, pp. 157–169.

[Coh89] A. Cohn. „The Notion of Proof in Hardware Verification“. In: vol. 5. 2. Kluwer Academic Publishers, 1989, pp. 127–139.

[Con07] C.-C. C. Consortium. *C2C-CC Manifesto, Version 1.1.* Aug. 2007.

[Coo79] S. A. Cook. „Deterministic CFL's are accepted simultaneously in polynomial time and log squared space“. In: *Proceedings of the 11th Annual ACM Symposium on Theory of Computing*. STOC '79. ACM, 1979, pp. 338–345.

[Cou94] O. Coudert. „Two-level logic minimization: an overview“. In: *Integration, the {VLSI} Journal* 17.2 (1994), pp. 97–140.

[CPB+12] S. Chang, R. Perlner, W. E. Burr, M. S. Turan, J. M. Kelsey, S. Paul, and L. E. Bassham. *Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition*. National Institute of Standards and Technology (NIST). 2012. URL: `http://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf`.

[CW03] S. A. Crosby and D. S. Wallach. „Denial of Service via Algorithmic Complexity Attacks“. In: *Proceedings of the 12th USENIX Security Symposium*. 2003, pp. 29–44.

[Dam88] I. B. Damgård. „Collision Free Hash Functions and Public Key Signature Schemes“. In: *Advances in Cryptology - EUROCRYPT '87*. Springer-Verlag. 1988, pp. 203–216.

[Dam89] I. B. Damgård. „A Design Principle for Hash Functions“. In: *Advances in Cryptology - CRYPTO ’89*. Springer-Verlag, 1989, pp. 416–427.

[DAR85] M. R. Dagenais, V. K. Agarwal, and N. C. Rumin. „The McBOOLE Logic Minimizer“. In: *22nd Design Automation Conference*. IEEE, 1985, pp. 667–673.

[DNRH07] J. D. Djigbenou, T. V. Nguyen, C. W. Ren, and D. S. Ha. „Development of TSMC 0.25μm Standard Cell Library“. In: *Proceedings of SoutheastCon*. IEEE. 2007, pp. 566–568.

[DP84] D. W. Davies and W. L. Price. „Digital signatures: An update“. In: *5th International Confercerence on Computer Communication*. 1984, pp. 845–849.

[DR99] J. Daemen and V. Rijmen. *AES Proposal: Rijndael*. Submission to NIST. 1999. URL: `http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf`.

[Dri09] S. Drimer. „Security for volatile FPGAs“. In: University of Cambridge, 2009.

[DRRS09] Y. Dodis, L. Reyzin, R. L. Rivest, and E. Shen. „Indifferentiability of Permutation-Based Compression Functions and Tree-Based Modes of Operation, with Applications to MD6“. In: *Fast Software Encryption*. Vol. 5665. Lecture Notes in Computer Science. Springer-Verlag, 2009, pp. 104–121.

[Ehl10] A. Ehliar. „Optimizing Xilinx designs through primitive instantiation“. In: *Proceedings of the 7th FPGAworld Conference*. ACM, 2010, pp. 20–27.

[FIPS 180-4] *Secure Hash Standard (SHS)*. National Institute of Standards and Technology (NIST), 2012.

[FLS+10] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. *The Skein Hash Function Family*. Submission to NIST. 2010. URL: `http://www.skein-hash.info/sites/default/files/skein1.3.pdf`.

[FPO05] A. Ferrante, V. Piuri, and J. Owen. „IPSec Hardware Resource Requirements Evaluation“. In: *Next Generation Internet Networks*. 2005, pp. 240–246.

[Fri01] E. G. Friedman. „Clock Distribution Networks in Synchronous Digital Integrated Circuits“. In: vol. 89. 5. IEEE, 2001, pp. 665–692.

[FS94] A. H. Farrahi and M. Sarrafzadeh. „Complexity of the Lookup-Table Minimization Problem for FPGA Technology Mapping“. In: vol. 13. 11. IEEE, 1994, pp. 1319–1332.

[GGE09] S. Ghaznavi, C. Gebotys, and R. Elbaz. „Efficient Technique for the FPGA Implementation of the AES MixColumns Transformation“. In: *International Conference on Reconfigurable Computing and FPGAs - ReConFig '09*. 2009, pp. 219–224.

[GHR+12a] K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif. *Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs.* Third SHA-3 Candidate Conference. 2012. URL: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/March2012/documents/papers/GAJ_paper.pdf`.

[GHR+12b] K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif. *Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs.* Cryptology ePrint Archive, Report 2012/368. `http://eprint.iacr.org/2012/368`. 2012.

[GHR10] K. Gaj, E. Homsirikamol, and M. Rogawski. „Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs“. In: *Cryptographic Hardware and Embedded Systems - CHES '10*. Vol. 6225. Lecture Notes in Computer Science. Springer-Verlag, 2010, pp. 264–278.

[GHR95] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, 1995.

[GJ90] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1990.

[GKA+10] K. Gaj, J.-P. Kaps, V. Amirineni, M. Rogawski, E. Homsirikamol, and B. Y. Brewster. „ATHENa - Automated Tool for Hardware EvaluatioN: Toward Fair and Comprehensive Benchmarking of Cryptographic Hardware Using FPGAs“. In: *20th International Conference on Field Programmable Logic and Applications.* IEEE, 2010, pp. 414–421.

[GKM+10] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen. *Grøstl – a SHA-3 candidate.* Submission to NIST. 2010. URL: `http://groestl.info/Groestl.pdf`.

[Gol97] A. V. Goldberg. „An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm“. In: *Journal of Algorithms* 22.1 (1997), pp. 1–29.

[GP97] S. Gao and D. Panario. „Tests and Constructions of Irreducible Polynomials over Finite Fields“. In: *In Foundations of Computational Mathematics.* Springer-Verlag, 1997, pp. 346–361.

[GPP11] J. Guo, T. Peyrin, and A. Poschmann. „The PHOTON Family of Lightweight Hash Functions“. In: *Advances in Cryptology - CRYPTO '11.* Vol. 6841. Lecture Notes in Computer Science. Springer-Verlag, 2011, pp. 222–239.

[Gro96] L. K. Grover. „A fast quantum mechanical algorithm for database search“. In: *Proceedings of the 28th Annual ACM Symposium on Theory of Computing.* ACM. 1996, pp. 212–219.

[GSH+12] X. Guo, M. Srivastav, S. Huang, D. Ganta, M. B. Henry, L. Nazhandali, and P. Schaumont. „ASIC implementations of five SHA-3 finalists“. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE. 2012, pp. 1006–1011.

[HE96] S. Hassoun and C. Ebeling. „Architectural Retiming: Pipelining Latency-Constrained Circuits". In: *33rd Design Automation Conference*. ACM, 1996, pp. 708–713.

[HGD85] F. Hoornaert, J. Goubert, and Y. Desmedt. „Efficient Hardware Implementation of the DES". In: *Advances in Cryptology - CRYPTO '85*. Vol. 196. Lecture Notes in Computer Science. Springer-Verlag, 1985, pp. 147–173.

[HK71] K. Hoffman and R. Kunze. *Linear Algebra*. Prentice-Hall, 1971.

[HMU06] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2006.

[How06] J. M. Howie. *Fields and Galois Theory*. Springer-Verlag, 2006.

[HP75] H. Hoehne and R. Piloty. „Design Verification at the Register Transfer Language Level". In: vol. 24. 9. IEEE, 1975, pp. 861–867.

[HRG11] E. Homsirikamol, M. Rogawski, and K. Gaj. „Throughput vs. Area Trade-offs in High-Speed Architectures of Five Round 3 SHA-3 Candidates Implemented Using Xilinx and Altera FPGAs". In: *Workshop on Cryptographic Hardward and Embedded Systems*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 491–506.

[HV04] A. Hodjat and I. Verbauwhede. „Interfacing a High Speed Crypto Accelerator to an Embedded CPU". In: *Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers*. Vol. 1. IEEE. 2004, pp. 488–492.

[HW02] E. Hemaspaandra and G. Wechsung. „The Mminimization Problem for Boolean Formulas". In: vol. 31. 6. SIAM, 2002, pp. 1948–1958.

[IS09] T. Isobe and K. Shibutani. „Preimage attacks on reduced Tiger and SHA-2". In: *Fast Software Encryption*. Vol. 5665. Lecture Notes in Computer Science. Springer-Verlag, 2009.

[ISO 29192-2] *ISO/IEC 29192-2:2012 Information technology – Security techniques – Lightweight cryptography – Part 2: Block ciphers*. International Organization for Standardization, 2012.

[JA11] B. Jungk and J. Apfelbeck. „Area-Efficient FPGA Implementations of the SHA-3 Finalists“. In: *International Conference on Reconfigurable Computing and FPGAs - ReConFig '11*. IEEE, 2011, pp. 235–241.

[JLH14] B. Jungk, L. R. Lima, and M. Hiller. „A Systematic Study of Lightweight Hash Functions on FPGAs“. In: *International Conference on Reconfigurable Computing and FPGAs - ReConFig '14*. IEEE, 2014.

[JNB99] M. B. Josephs, S. M. Nowick, and C. H. V. Berkel. „Modeling and Design of Asynchronous Circuits“. In: vol. 87. 2. IEEE, 1999, pp. 234–242.

[JR10a] B. Jungk and S. Reith. *On FPGA-based implementations of Grøstl*. Cryptology ePrint Archive, Report 2010/260. `http://eprint.iacr.org/2010/260`. 2010.

[JR10b] B. Jungk and S. Reith. „On FPGA-Based Implementations of the SHA-3 Candidate Grøstl“. In: *International Conference on Reconfigurable Computing and FPGAs - ReConFig '10*. IEEE, 2010, pp. 316–321.

[JS13] B. Jungk and M. Stöttinger. „Among Slow Dwarfs and Fast Giants: A Systematic Design Space Exploration of Keccak“. In: *International Workshop on Reconfigurable Communication-centric Systems-on-Chip - ReCoSoC '13*. IEEE, 2013.

[JSH14] B. Jungk, M. Stöttinger, and M. Harter. *Shrinking KECCAK Hardware Implementations*. SHA-3 2014 Workshop. 2014.

[Jun11] B. Jungk. *Compact Implementations of Grøstl, JH and Skein for FPGAs*. Ecrypt II Hash Workshop. 2011.

[Jun12] B. Jungk. *Evaluation Of Compact FPGA Implementations For All SHA-3 Finalists.* Third SHA-3 Candidate Conference. 2012. URL: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/March2012/documents/papers/JUNGK_paper.pdf`.

[Kay07] R. F. Kayser. „Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family“. In: *Federal Register.* Vol. 72. National Institute of Standards and Technology (NIST), 2007, pp. 62212–62220.

[KC99] V. Kabanets and J.-Y. Cai. *Circuit Minimization Problem.* Tech. rep. Electronic Colloquium on Computational Complexity, 1999.

[KDV+11] S. Kerckhof, F. Durvaux, N. Veyrat-Charvillon, F. Regazzoni, G. M. de Dormale, and F.-X. Standaert. „Compact FPGA Implementations of the Five SHA-3 Finalists“. In: *10th Smart Card Research and Advanced Application Conference.* Vol. 7079. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 217–233.

[KE10] H. Krawczyk and P. Eronen. „HMAC-based Extract-and-Expand Key Derivation Function (HKDF)“. In: IETF, 2010.

[Kil07] S. Kilts. *Advanced FPGA Design: Architecture, Implementation, and Optimization.* John Wiley & Sons, 2007.

[KJJ99] P. Kocher, J. Jaffe, and B. Jun. „Differential Power Analysis“. In: *Advances in Cryptology - CRYPTO '99.* Vol. 1666. Lecture Notes in Computer Science. Springer-Verlag, 1999, pp. 388–397.

[KM03] D. Koufaty and D. T. Marr. „Hyperthreading Technology in the Netburst Microarchitecture“. In: *IEEE Micro* 23.2 (2003), pp. 56–65.

[KM09] C. Karpfinger and K. Meyberg. *Algebra Gruppen - Ringe - Körper.* Spektrum Akademischer Verlag, 2009.

[Knu69] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms.* Addison-Wesley, 1969.

[Knu73] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.

[KO62] A. Karatsuba and Y. Ofman. „Multiplication of Many-Digital Numbers by Automatic Computers“. In: *Proceedings of the USSR Academy of Sciences.* Vol. 145. 1962, pp. 293–294.

[Ko85] K.-I. Ko. „On some Natural Complete Operators“. In: *Theoretical Computer Science* 37 (1985), pp. 1–30.

[Kor01] I. Koren. *Computer Arithmetic Algorithms.* 2nd. A. K. Peters, Ltd., 2001.

[KR07] I. Kuon and J. Rose. „Measuring the Gap Between FPGAs and ASICs“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (2007), pp. 203–215.

[KS05] J. Kelsey and B. Schneier. „Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work“. In: *Advances in Cryptology - EUROCRYPT '05.* Vol. 3494. Lecture Notes in Computer Science. Springer-Verlag, 2005, pp. 474–490.

[KST04] R. Kalla, B. Sinharoy, and J. M. Tendler. „IBM POWER5 Chip: a Dual-Core Multithreaded Processor“. In: *IEEE Micro* 24.2 (2004), pp. 40–47.

[KTR08] I. Kuon, R. Tessier, and J. Rose. „FPGA Architecture: Survey and Challenges“. In: vol. 2. 2. Now Publishers Inc., 2008, pp. 135–253.

[KYS+11] J.-P. Kaps, P. Yalla, K. K. Surapathi, B. Habib, S. Vadlamudi, S. Gurung, and J. Pham. *Lightweight Implementations of SHA-3 Finalists on FPGAs.* Submission to NIST (Round 3). 2011. URL: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/March2012/documents/papers/KAPS_paper.pdf`.

[LN96] R. Lidl and H. Niederreiter. *Finite Fields (Encyclopedia of Mathematics and its Applications).* Cambridge University Press, 1996.

[Luc05] S. Lucks. „A Failure-Friendly Design Principle for Hash Functions“. In: *Advances in Cryptology - ASIACRYPT '05*. Vol. 3788. Lecture Notes in Computer Science. Springer-Verlag, 2005, pp. 474–494.

[Mal93] S. Malik. „Analysis of Cyclic Combinational Circuits“. In: *1993 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers.* 1993, pp. 618–625.

[MBV06] V. Manohararajah, S. D. Brown, and Z. G. Vranesic. „Heuristics for Area Minimization in LUT-Based FPGA Technology Mapping“. In: vol. 25. 11. IEEE, 2006, pp. 2331–2340.

[Mer74] F. Mertens. „Über einige asymptotische Gesetze der Zahlentheorie“. In: *Journal für die reine und angewandte Mathematik* 77 (1874), pp. 289–338.

[Mer79] R. C. Merkle. „Secrecy, Authentication, and Public Key Systems“. PhD thesis. Stanford University, 1979.

[Mer89] R. C. Merkle. „A Certified Digital Signature“. In: *Advances in Cryptology - CRYPTO '89*. Springer-Verlag, 1989, pp. 218–238.

[MMO85] S. M. Matyas, C. H. Meyer, and J. Oseas. „Generating strong one-way functions with cryptographic algorithm“. In: IBM, 1985.

[Möb32] A. F. Möbius. „Über eine besondere Art von Umkehrung der Reihen“. In: *Journal für die reine und angewandte Mathematik* 9 (1832), pp. 105–123.

[MOI90] S. Miyaguchi, K. Ohta, and M. Iwata. „128-bit hash function (N-hash)“. In: *NTT review* 2.6 (1990), pp. 128–132.

[Mon85] P. L. Montgomery. „Modular Multiplication Without Trial Division“. In: *Mathematics of Computation* 44.170 (1985), pp. 519–521.

[MOP07] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer-Verlag, 2007.

[MOV97] A. Menezes, P. van Ooorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[MRH04] U. Maurer, R. Renner, and C. Holenstein. „Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology“. In: *Theory of Cryptography*. Vol. 2951. Lecture Notes in Computer Science. Springer-Verlag, 2004, pp. 21–39.

[MS03] S. Morioka and A. Satoh. „An Optimized S-Box Circuit Architecture for Low Power AES Design“. In: *Cryptographic Hardware and Embedded Systems - CHES '02*. Vol. 2523. Lecture Notes in Computer Science. Springer-Verlag, 2003, pp. 172–186.

[MS09] G. Martin and G. Smith. „High-Level Synthesis: Past, Present, and Future“. In: vol. 26. 4. IEEE, 2009, pp. 18–25.

[MSBS93] P. C. McGeer, J. V. Sanghavi, R. K. Brayton, and A. L. Sangiovanni-Vicentelli. „ESPRESSO-SIGNATURE: A New Exact Minimizer for Logic Functions“. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 1.4 (1993), pp. 432–440.

[NRR06] S. Nikova, C. Rechberger, and V. Rijmen. „Threshold Implementations against Side-Channel Attacks and Glitches“. In: *Information and Communications Security*. Springer-Verlag, 2006, pp. 529–545.

[Oec03] P. Oechslin. „Making a Faster Cryptanalytic Time-Memory Trade-Off“. In: *Advances in Cryptology - CRYPTO '03*. Springer-Verlag, 2003, pp. 617–630.

[OM86] J. K. Omura and J. L. Massey. *Computational method and apparatus for finite field arithmetic*. U.S. patent #4,587,627. 1986.

[Paa94] C. Paar. „Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields“. PhD thesis. Rhur-Universität Bochum, 1994.

[PAFL98] A. Postula, D. Abramson, Z. Fang, and P. Logathetis. „A Comparison of High Level Synthesis and Register Transfer Level Design Techniques for Custom Computing Machines“.

In: *Proceedings of the 31st Hawaii International Conference on System Sciences*. Vol. 7. IEEE. 1998, pp. 207–214.

[PB61] W. W. Peterson and D. T. Brown. „Cyclic Codes for Error Detection“. In: *Proceedings of the IRE* 49.1 (1961), pp. 228–235.

[Pos41] E. L. Post. *The Two-Valued Iterative Systems Of Mathematical Logic*. 5. Princeton University Press, 1941.

[PR09] L. Piga and S. Rigo. „Comparing RTL and High-Level Synthesis Methodologies in the Design of a Theora Video Decoder IP Core“. In: *5th Southern Conference on Programmable Logic*. IEEE. 2009, pp. 135–140.

[Pre93] B. Preneel. „Analysis and Design of Cryptographic Hash Functions“. PhD thesis. Katholieke Universiteit te Leuven, 1993.

[Pre99] B. Preneel. „The State of Cryptographic Hash Functions“. In: *Lectures on Data Security*. Springer-Verlage, 1999, pp. 158–182.

[Rab80] M. O. Rabin. „Probabilistic Algorithms in Finite Fields“. In: vol. 9. 2. SIAM, 1980, pp. 273–280.

[RD02] V. Rijmen and J. Daemen. *The Design of Rijndael*. Springer-Verlag, 2002.

[Rij00] V. Rijmen. *Efficient implementation of the Rijndael S-Box*. Tech. rep. Katholieke Universiteit Leuven, 2000.

[Riv92] R. Rivest. „The MD4 Message-Digest Algorithm“. In: IETF, 1992.

[Rog06] P. Rogaway. „Formalizing Human Ignorance: Collision-Resistant Hashung without the Keys“. In: *Progress in Cryptology - VIETCRYPT '06*. Springer-Verlag, 2006, pp. 211–228.

[RS04] P. Rogaway and T. Shrimpton. „Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance“. In: *Fast Software Encryption*. Vol. 3017. Lecture Notes in Computer Science. Springer-Verlag, 2004, pp. 371–388.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. „A Method for Obtaining Digital Signatures and Public-Key Cryptosystems“. In: vol. 21. 2. ACM, 1978, pp. 120–126.

[Sas13] T. Sasao. „Four Decades of Multi-Valued Logic: Lists of Highly Cited Papers“. In: *43rd International Symposium on Multiple-Valued Logic*. 2013, pp. 198–202.

[Sav97] J. E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, 1997.

[Sch11] T. Schütze. „Automotive Security: Cryptography for Car2X Communication“. In: *Embedded World Conference*. 2011.

[Sem12] *How an FPGA Approach to Complex System Design Can Improve Profitability: Real Case Studies*. SEMICO Research Corporation, 2012.

[SGI+11] H. Schweppe, T. Gendrullis, M.-S. Idrees, Y. Roudier, B. Weyl, and M. Wolf. „Securing Car2X Applications with effective Hardware-Software Co-Design for Vehicular On-Board Networks“. In: *VDI Automotive Security* 27 (2011).

[Sha79] A. Shamir. „How to Share a Secret“. In: vol. 22. 11. ACM, 1979, pp. 612–613.

[She13] H. M. Sheffer. „A set of five independent postulates for Boolean algebras, with application to logical constants“. In: vol. 14. 4. JSTOR, 1913, pp. 481–488.

[Sho91] V. Shoup. „A Fast Deterministic Algorithm for Factoring Polynomials over Finite Fields of Small Characteristic“. In: *Proceedings of the International Symposium on Symbolic and Algebraic Computation '91*. ACM. 1991, pp. 14–21.

[Sim96] W. A. Simpson. „PPP Challenge Handshake Authentication Protocol (CHAP)“. In: IETF, 1996.

[Sip96] M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[SKNI10] B. Song, K. Kawakami, K. Nakano, and Y. Ito. „An RSA Encryption Hardware Algorithm Using a Single DSP Block and a Single Block RAM on the FPGA“. In: *1st International Conference on Networking and Computing (ICNC)*. IEEE. 2010, pp. 140–147.

[SS08] S. K. Sanadhya and P. Sarkar. „New collision attacks against up to 24-step SHA-2“. In: *Progress in Cryptology - INDOCRYPT '08*. Vol. 5365. Lecture Notes in Computer Science. Springer-Verlag, 2008.

[Stö13] M. S. P. Stöttinger. „Mutating Runtime Architectures as a Countermeasure Against Power Analysis Attacks“. PhD thesis. TU Darmstadt, 2013.

[Sun05] B. Sunar. „An Efficient Basis Conversion Algorithm for Composite Fields with Given Representations“. In: vol. 54. 8. IEEE, 2005, pp. 992–997.

[TLW+90] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, and R. L. Blackburn. *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*. The Springer International Series in Engineering and Computer Science. Springer-Verlag, 1990.

[TNW96] M. Theobald, S. M. Nowick, and T. Wu. „Espresso-hf: A heuristic hazard-free minimizer for two-level logic“. In: *33rd Design Automation Conference*. ACM, 1996, pp. 71–76.

[Tur36] A. M. Turing. „On Computable Numbers, with an Application to the Entscheidungsproblem“. In: *Proceedings of the London Mathematical Society* 42 (1936), pp. 230–265.

[TV04] K. Tiri and I. Verbauwhede. „A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation“. In: *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*. Vol. 1. IEEE, 2004, pp. 246–251.

[TW02] W. Trappe and L. C. Washington. *Introduction to Cryptography: with Coding Theory*. Prentice Hall, 2002.

[TW06] J. Talbot and D. Welsh. *Complexity and Cryptography – An Introduction.* Cambridge University Press, 2006.

[Vad04] S. Vadlamani. „The Synchronized Pipelined Parallelism Model“. MA thesis. University of California, Irvine, 2004.

[Ver05] *IEEE Std 1364-2005. IEEE Standard for Verilog Hardware Description Language.*

[VHDL08] *IEEE Std 1076-2008. IEEE Standard VHDL Language Reference Manual.*

[VK07] K. Vitoroulis and A. J. Al-Khalili. „Performance of Parallel Prefix Adders Implemented with FPGA Technology“. In: *IEEE Northeast Workshop on Circuits and Systems – '07.* 2007, pp. 498–501.

[Vol99] H. Vollmer. *Introduction to Circuit Complexity: A Uniform Approach.* Springer-Verlag, 1999.

[WFLY04] X. Wang, D. Feng, X. Lai, and H. Yu. *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD.* Cryptology ePrint Archive, Report 2004/199. http://eprint.iacr.org/2004/199. 2004.

[WG10] C. Wenzel-Benner and J. Gräf. „XBX: eXternal Benchmarking eXtension for the SUPERCOP Crypto Benchmarking Framework“. In: *Cryptographic Hardware and Embedded Systems - CHES '10.* Vol. 6225. Lecture Notes in Computer Science. Springer-Verlag, 2010, pp. 294–305.

[Win84a] R. S. Winternitz. „A Secure One-Way Hash Function Built from DES.“ In: *IEEE Symposium on Security and Privacy.* 1984, pp. 88–90.

[Win84b] R. S. Winternitz. „Producing a One-Way Hash Function from DES“. In: *Advances in Cryptology - CRYPTO '83.* Vol. 1440. Springer-Verlag, 1984, pp. 203–207.

[WP06] A. Weimerskirch and C. Paar. *Generalizations of the Karatsuba Algorithm for Efficient Implementations.* Cryptology ePrint Archive, Report 2006/224. `http://eprint.iacr.org/2006/224`. 2006.

[Wu11] H. Wu. *The Hash Function JH.* Submission to NIST. 2011. URL: `http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf`.

[WYY05] X. Wang, Y. L. Yin, and H. Yu. „Finding Collisions in the Full SHA-1“. In: *Advances in Cryptology - CRYPTO '05.* Vol. 3621. Lecture Notes in Computer Science. Springer-Verlag, 2005, pp. 17–36.

[WZ10] H. Wang and H. Zhang. „A fast pseudorandom number generator with BLAKE hash function“. In: vol. 15. 5. Wuhan University, 2010, pp. 393–397.

[Xil05] Xilinx. *Using Look-Up Tables as Distributed RAM in Spartan-3 Generation FPGAs.* 2005. URL: `http://www.xilinx.com/support/documentation/application_notes/xapp464.pdf`.

[Xil12a] Xilinx. *LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11f).* 2012. URL: `http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20/v2_11_f/fsl_v20.pdf`.

[Xil12b] Xilinx. *LogiCORE IP MicroBlaze Micro Controller System (v1.1).* 2012. URL: `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ds865_microblaze_mcs.pdf`.

[Xil12c] Xilinx. *Virtex-5 FPGA User Guide.* 2012. URL: `http://www.xilinx.com/support/documentation/user_guides/ug190.pdf`.

[Xil12d] Xilinx. *Virtex-5 FPGA XtremeDSP Design Considerations.* 2012. URL: `http://www.xilinx.com/support/documentation/user_guides/ug193.pdf`.

[Xil13a] Xilinx. *Command Line Tools User Guide*. 2013. URL: `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_5/devref.pdf`.

[Xil13b] Xilinx. *Vivado Design Suite User Guide – High-Level Synthesis*. 2013. URL: `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_4/ug902-vivado-high-level-synthesis.pdf`.

[Xil13c] Xilinx. *XST User Guide*. 2013. URL: `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_5/xst.pdf`.

[XY98] S. Xing and W. W. h. Yu. „FPGA Adders: Performance Evaluation and Optimal Design“. In: vol. 15. IEEE, 1998, pp. 24–29.

[Yin00] „IEEE Standard Specifications for Public-Key Cryptography“. In: *IEEE Std 1363-2000* (2000). Ed. by Y. L. Yin.

[Yuv79] G. Yuval. „How to swindle Rabin“. In: *Cryptologia* 3.3 (1979), pp. 187–191.

Author’s Previous Publications

[HJR13] A. Himmighofen, B. Jungk, and S. Reith. „On a FPGA-based Method for Authentication using Edwards Curves“. In: *International Workshop on Reconfigurable Communication-centric Systems-on-Chip - ReCoSoC ’13*. IEEE, 2013.

[JA11] B. Jungk and J. Apfelbeck. „Area-Efficient FPGA Implementations of the SHA-3 Finalists“. In: *International Conference on Reconfigurable Computing and FPGAs - ReConFig ’11*. IEEE, 2011, pp. 235–241.

[JLH14] B. Jungk, L. R. Lima, and M. Hiller. „A Systematic Study of Lightweight Hash Functions on FPGAs“. In: *International Conference on Reconfigurable Computing and FPGAs - ReConFig ’14*. IEEE, 2014.

[JR10a] B. Jungk and S. Reith. *On FPGA-based implementations of Grøstl*. Cryptology ePrint Archive, Report 2010/260. `http://eprint.iacr.org/2010/260`. 2010.

[JR10b] B. Jungk and S. Reith. „On FPGA-Based Implementations of the SHA-3 Candidate Grøstl“. In: *International Conference on Reconfigurable Computing and FPGAs - ReConFig ’10*. IEEE, 2010, pp. 316–321.

[JRA09] B. Jungk, S. Reith, and J. Apfelbeck. *On Optimized FPGA Implementations of the SHA-3 Candidate Grøstl*. Cryptology ePrint Archive, Report 2009/206. `http://eprint.iacr.org/2009/206`. 2009.

[JS13] B. Jungk and M. Stöttinger. „Among Slow Dwarfs and Fast Giants: A Systematic Design Space Exploration of Keccak“. In: *International Workshop on Reconfigurable Communication-centric Systems-on-Chip - ReCoSoC '13*. IEEE, 2013.

[JSG+12] B. Jungk, M. Stöttinger, J. Gampe, S. Reith, and S. A. Huss. „Side-channel Resistant AES Architecture Utilizing Randomized Composite Field Representations“. In: *International Conference on Field Programmable Technology*. IEEE, 2012, pp. 125–128.

[JSH14] B. Jungk, M. Stöttinger, and M. Harter. *Shrinking KECCAK Hardware Implementations*. SHA-3 2014 Workshop. 2014.

[Jun09] B. Jungk. „Selbstorganisierendes Service Level Management basierend auf Mechanismus-Design“. In: *ECEASST* 17 (2009).

[Jun11] B. Jungk. *Compact Implementations of Grøstl, JH and Skein for FPGAs*. Ecrypt II Hash Workshop. 2011.

[Jun12] B. Jungk. *Evaluation Of Compact FPGA Implementations For All SHA-3 Finalists*. Third SHA-3 Candidate Conference. 2012. URL: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/March2012/documents/papers/JUNGK_paper.pdf`.