Karlsruhe Institute of Technology

Institut für Technische Informatik
Chair for Embedded Systems

Beschleunigen einer post-quantum sicheren Hashfunktion auf einem rekonfigurierbaren Prozessor

Bachelorarbeit
von

Niklas Lorenz

am Karlsruher Institut für Technologie (KIT)
Fakultät für Informatik
Institut für Technische Informatik (ITEC)
Chair for Embedded Systems (CES)

Erstgutachter:	Prof. Dr. Jörg Henkel
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuer:	Hassan Nassar, Dr. Lars Bauer

Tag der Anmeldung:	01.06.2023
Tag der Abgabe:	02.10.2023

Erklärung

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Die verwendeten Quellen und Hilfsmittel sind im Literaturverzeichnis vollständig aufgeführt.

Karlsruhe, den 02.10.2023

Niklas Lorenz

Zusammenfassung

Eine kurze Zusammenfassung der Arbeit. Vielleicht ein Absatz; allerhöchstens eine Seite. Die deutschsprachige Version ist nur erforderlich, wenn die Ausarbeitung auf Deutsch geschrieben ist. Die englischsprachige Version (s.u.) ist immer(!) erforderlich. Könnte auf der gleichen Seite stehen oder auf der nächsten Seite.

Summary

A brief summary of the work. Maybe just a paragraph; at most one page. The German summary (see above) is only required if the thesis was written in German.

Contents

Chapter 1

Einleitung

Chapter 2

i-Core

Der icore ist ein rekonfigurierbarer Prozessor (Cite Missing), das bedeutet er besitzt neben den klassischen Komponenten eines Prozessors noch eine zur Laufzeit konfigurierbare Einheit aus Hardwarebeschleunigern. Aktuell ist er vollständig auf einer Xilinx VC-707 FPGA (siehe Abschnitt 2.1.1) implementiert. Als Grundlage dient ein modifizierter Leon3-Kern, ein 7-Stufen-Pipeline-Prozessor basierend auf der SPARC-V8-Architektur (Cite Missing). Er wird erweitert um eine sogenannte **Reconfigurable Fabric**, einen Container für fünf Beschleunigerblöcke mit Anbindung an Speicher und Registerdatei. Wie genau diese Fabric aufgebaut ist und wie sie funktioniert schauen wir uns in Abschnitt 2.2 an. Vorher wollen wir uns aber erstmal überlegen wofür rekonfiguruerbare Prozessoren eigentlich gut sind.

2.1 Rekonfigurierbare Prozessoren

Herkömmliche Prozessoren verfügen über einen mehr oder weniger komplexen Befehlssatz, der es ihnen erlaubt jede beliebige Berechnung durchzuführen, indem die gewünschte Rechenoperation in mehrere vom Befehlssatz unterstützte Operationen aufgeteilt wird. Dadurch sind sie extrem flexibel. Für rechenintensive Aufgaben, bei denen komplexere Operationen sehr oft ausgeführt werden müssen, ist dieser Ansatz jedoch nachteilig, da der Prozessor viel Zeit benötigt, um diese komplexen Operationen zu berechnen. Um dem Abhilfe zu verschaffen, wird spezializsierte Hardware wie zum Beispiel Grafikkarten, Netzwerkkarten oder FPGAs eingesetzt, die besonders effizient eine bestimmte Art von Aufgabe erfüllen. Durch die fortschreitende Digitalisierung und Vorstöße in Bereichen wie der Industrie 4.0 oder dem Internet der Dinge (IoT) wächst der Bedarf an Kleinstrechensystemen, die für den konkreten Anwendungszweck spezielle Aufgaben übernehmen können. Diese müssen vor allem energieeffizient, klein und günstig in der Produktion sein und müssen dabei trotzdem auch in der Lage sein komplexere Aufgaben teilweise sogar in Echtzeit erfüllen zu können. Ein rekonfigurierbarer Prozessor bietet hier die Vorteile der hohen Flexibilität eines normalen Prozessors und verbindet sie mit der hohen Spezialisierbarkeit von FPGAs, indem er mehrere kleine Blöcke an vom Entwickler konfigurierbare Logikblöcke bereitstellt, die mit speziellen Prozessorinstruktionen gesteuert werden können. Auf diese Weise können auch sehr spezielle Anwendungen von Off-The-Shelf Hardware erfüllt werden, was die Produktionskosten gegenüber Spezialanfertigungen, sowie den Energieverbrauch und den Bedarf an Rechenleistung gegenüber klassischen Prozessoren reduziert.

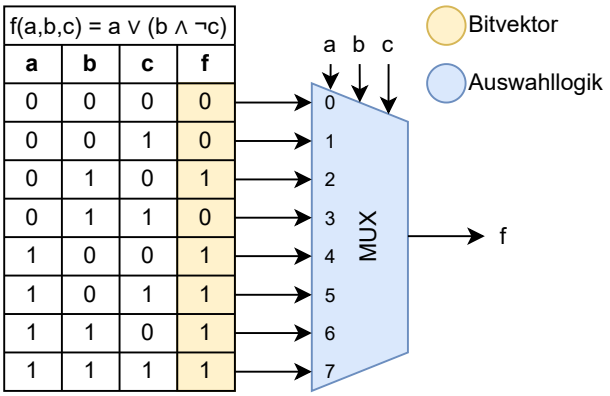

Figure 2.1: Implementierung einer dreistelligen Funktion durch einen Lookup Table

2.1.1 FPGA

Field Programmable Gate Arrays (FPGAs) sind integrierte Schaltkreise, die von sich aus keine genaue Funktion implementieren. Stattdessen muss erst eine Schaltung "geladen" werden. Dabei macht man sich zu Nutze, dass eine n-stellige boolsche Funktion durch einen 2^n-Bit Vektor codiert werden kann. In Abb. 2.1 ist zum Beispiel die Funktion $f = A \vee (B \wedge \neg C)$ dargestellt. Der 8-Bit Ergebnisvektor wird in SRAM-Speicherzellen gehalten und durch einen 8-zu-1-Multiplexer kann mit A, B und C das entsprechende Ergebnisbit ausgewählt werden. Diese Schaltung wird Lookup Table (LUT) genannt. Mehrere solcher Lookup Tables werden zusammen mit anderen Komponenten wie zum Beispiel Flip-Flops oder Recheneinheiten wie Full-Addern zu sogenannten **Configurable Logic Blocks** (CLBs) zusammengefasst, im Folgenden auch Logikblöcke genannt. Logikblöcke sind bereits sehr vielseitig, aufgrund der festen Anordnung ihrer Komponenten jedoch immer noch stark eingeschränkt. Um dem entgegen zu wirken sind die Logikblöcke untereinander mit einem flexiblen und ebenfalls konfigurierbaren Verbindungsnetz verbunden. Auf diese Weise kann jede beliebige Funktion realisiert werden, sofern genug Logikblöcke zur Verfügung stehen. Neben den normalen Logikblöcken stellen FPGAs typischerweise auch noch andere Komponenten bereit, die etwas speziellere Funktionen realisieren, die oft benötigt werden und die sehr viel Platz benötigen, wenn mit Logikblöcken implementert. Dazu gehören unter anderem **Digital Signal Processors** (DSPs), sie stellen Funktionen bereit, die zur Signalverarbeitung benötigt werden, und **Block RAM** (BRAMs), die ähnlich wie klassischer RAM in der Lage sind eine große Menge Daten zu speichern. FPGAs haben gegenüber Mikroprozessoren den großen Vorteil, dass alle Funktionen, die von den Logikblöcken realisiert werden, gleichzeitig und kontinuierlich berechnet werden. Auf diese Weise kann eine enorme Beschleunigung gegenüber einer reinen Software-Implementierung erzielt werden, da der Prozessor alle Berechnungen nacheinander durchführen muss.

2.2 icore-Architektur

Der icore ist ein Multicoreprozessor, der vollständig auf einer FPGA implementiert ist. Als Hardware wird ein Xilinx VC 707 Board verwendet. Es handelt sich um einen heterogenen Prozessor, seine Kerne sind also unterschiedlich gebaut. Während zwar alle Kerne auf dem Leon3 basieren, verfügt der Primärkern, wie in Abb. 2.2 über die *Reconfigurable Fabric*. Diese Fabric enthält fünf zur Laufzeit rekonfigurierbare Blöcke, die sogenannten

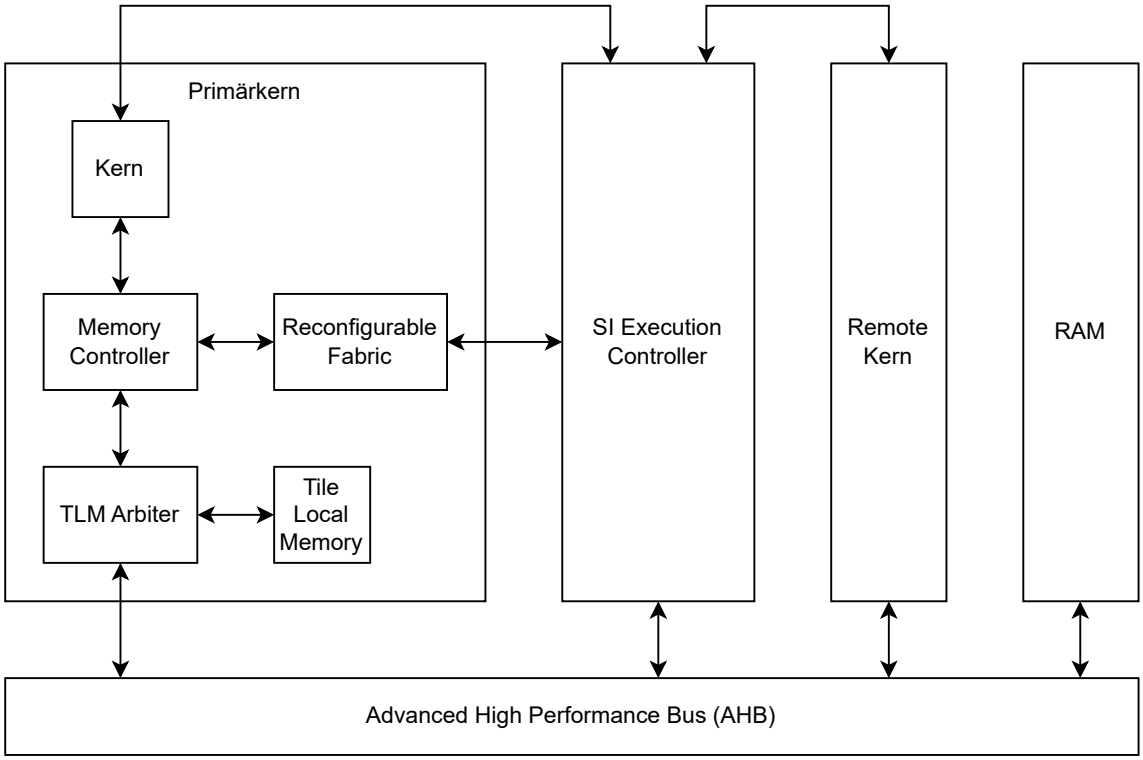

Figure 2.2: Aufbau des icore mit Ausführungskontrolle und Reconfigurable Fabric (Nachbildung aus (Cite missing, [3]))

Atome, die zur Hardwarebeschleunigung verwendet werden können. Der Primärkern verfügt außerdem noch über einen 1MB großen *Tile Local Memory* (TLM), der mit dem *Memory Controller* über ein 256 Bit breites Interface verbunden ist.

2.2.1 Reconfigurable Fabric

Die *Reconfigurable Fabric* enthält fünf Atomcontainer (AC0 bis AC4), die zur Laufzeit rekonfiguriert werden können und in die die Beschleuniger geladen werden. Jedes Atom hat eine Größe von 1600 LUTs, die den Beschleunigern zur Verfügung stehen. Über einen Bus-Connector sind die Atome über eine 2 Lane breite Schnittstelle an einen 4 Lane breiten Bus angebunden (eine Lane ist ein Vollduplexkanal mit einer Breite von 32 Bit). Welche Kanäle den Atomen als Ein- und Ausgabe dienen, wird von den VLCWs der Spezialinstruktion bestimmt (siehe Abschnitt 2.2.2). Der Bus ist segmentiert, das heißt es können auch mehrere Kommunikationen über den gleichen Kanal stattfinden, solange sich die Kommunikationspfade nicht physisch überlappen. Neben den Atomcontainern verfügt die *Reconfigurable Fabric* noch über weitere Komponenten wie *Load-Store-Units* (LSUs), mit denen Daten aus dem TLM oder dem RAM gelesen oder geschrieben werden können. Jede LSU besitzt eine 128 Bit Anbindung mit dem TLM, mit dem die LSU mit geringer Latenz eine große Menge Daten verarbeiten kann. Um die gelesenen Daten zu halten, stehen den LSUs jeweils vier Puffer it jeweils 128 Bit zur Verfügung. Mit Hilfe von *Address-Generation-Units* (AGUs) können verschiedene Zugriffsmuster für die LSUs generiert werden. Zuletzt gibt es noch die *Repack Units* mit denen Daten auf dem Bus kombiniert werden können. Wir werden sie nicht weiter benötigen, deshalb seien sie

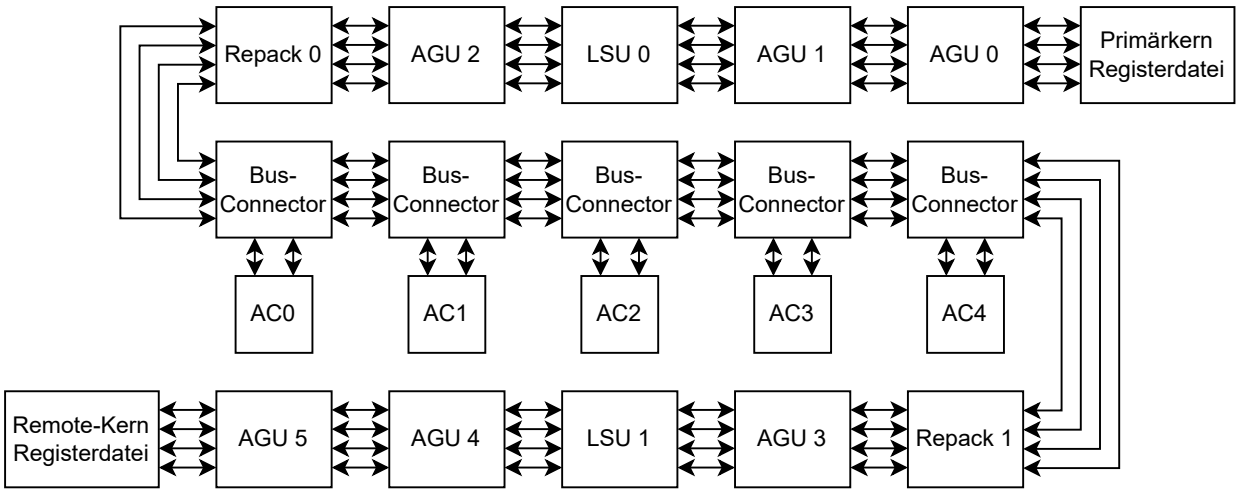

Figure 2.3: Aufbau der Reconfigurable Fabric

hier nur der Vollständigkeit halber kurz erwähnt. All diese Komponenten sind ebenfalls mit dem Bus verbunden. An den Enden des Busses werden die Register des Prozessors bereitgestellt, sodass die Fabric auf bis zu vier Argumente zugreifen kann. Die genaue Anordnung der Komponenten ist in Abbildung 2.3 dargestellt.

2.2.2 Spezialinstruktionen

Der Prozessor stellt Spezialinstruktionen (SIs) bereit, die von Programmen benutzt werden, um die Beschleuniger zu verwenden. Taucht eine solche Spezialinstruktion im Programmcode auf, wird die Pipeline des Prozessors angehalten und, falls notwendig, die für den Beschleuniger benötigten Atome konfiguriert. Über den *SI Execution Controller* (siehe Abschnitt 2.2.3) wird dann die Abarbeitung durch den Beschleuniger durchgeführt. Nachdem die Abarbeitung abgeschlossen ist, wird dann die Kontrolle wieder an den Prozessorkern übertragen und die Pipeline wird neugestartet. Dabei ist noch anzumerken, dass die *Reconfigurable Fabric* zwar nur im Primärkern implementiert ist, die anderen Kerne jedoch trotzdem die Beschleuniger verwenden können, indem sie über den AHB den *SI Execution Controller* ansprechen (siehe Abb. 2.2), der die *Reconfigurable Fabric* verwaltet.

2.2.3 SI Execution Controller

Die Spezialinstruktionen sind mit VLCWs mikroprogrammiert. Wird ein Beschleuniger geladen wird auch der Mikrocode für den Beschleuniger in den *Execution Controller* geladen. Dieser hat die volle Kontrolle über die *Reconfigurable Fabric* und kontrolliert mit Hilfe der VLCWs ihre Komponenten. Jedes VLCW definiert dabei einen Kontrollschritt, die der Reihe nach abgearbeitet werden. Für die Atome werden von einem VLCW die Anbindung der Eingabe- und Ausgabelanes an den Bus bestimmt sowie ein 6 Bit Kontrollvektor. Die Funktion des Kontrollvektors kann im Atom beliebig realisiert werden. Für die LSUs wird bestimmt welche AGU zur Adressgenerierung verwendet werden soll, welche Puffer für die Lese-/Schreiboperation verwendet werden sollen und welche Daten aus dem Puffer an den Bus angelegt werden sollen bzw. vom Bus in den Puffer übernommen werden sollen. Für die AGUs wird der Betriebsmodus über die VLCWs bestimmt. Es gibt sowohl Betriebsmodi für rein statische Zugriffsmuster, die vollständig über die VLCWs bestimmt werden, als auch

dynamische Modi, bei denen das Zugriffsmuster über Parameter direkt vom Datenbus bestimmt wird. Für jede Spezialinstruktionen stehen insgesamt 256 VLCWs zur Verfügung. Um auch Programme zu realisieren, die länger als 256 Kontrollschritte sind, sind auch rudimentäre

2.3 Erweiterung Dynamic Execution

Um auch einen dynamischen Kontrollfluss innerhalb des VLCW-Mikrocodes zu ermöglichen, wie zum Beispiel Schleifen, die von einem Laufzeitparameter abhängen, verwenden wir eine von Sascha Hering entwickelte Erweiterung der VLCWs (Cite missing, [3]). Die Atomcontainer werden mit zwei weiteren Ausgabebits ausgestattet. Diese können vom den Atomen des Beschleunigers beliebig implementiert werden. Um nun Sprünge realisieren zu können, werden die VLCWs um einen neuen Kontrollabschnitt erweitert. Dieser kann benutzt werden, um Zählervariablen für Schleifen zu setzen oder Sprünge anhand von diesen Zählern durchzuführen. Es sind jedoch auch Sprünge möglich, Die nur bei bestimmten Mustern der neuen Ausgabebits der Atome genommen werden. Hierzu lässt sich im VLCW mit einer Bitmaske codieren welche Atom Container betrachtet werden sollen, ob die Bedingung für alle oder nur für ein Atom erfüllt sein muss, welche Vergleichsoperation auf dem Kontrollvektor durchgeführt werden soll und zu welcher VLCW gesprungen werden soll. Diese Erweiterung ist für die Realisierung von Hashfunktionen besonders interessant, da die Eingabelänge für die Berechnung völlig variabel ist und so immer zur Laufzeit bestimmt werden muss, wie oft die zugrundeliegende Funktion berechnet werden muss. Ohne diese Erweiterung wäre es nicht möglich die vollständige Berechnung innerhalb einer einzigen Spezialinstruktion durchzuführen, sondern der dynamische Teil müsste in Software implementiert werden und es müsste nach jeder Iteration des beschleunigers das Zwischenergebnis gespeichert werden was, wie wir später bei der Implementierung sehen werden, zu starken Leistungseinbußen führen würde.

Chapter 3

SHA-3

SHA-3 ("*Secure Hash Function*"-3) ist eine vom US-amerikanischen "National Institute of Standards and Technology" definierte Familie von kryptographischen Hashfunktionen. Zu ihr gehören SHA3-224, SHA3-256, SHA3-384, SHA3-512, sowie noch zwei Zufallsfunktionen mit variabler Ausgabelänge SHAKE128 und SHAKE256, die wir hier aber nicht weiter betrachten wollen. In Tabelle 3.1 sind ein paar wichtige Eigenschaften der unterschiedlichen SHA3-Funktionen aufgelistet.

Bei einer kryptographischen Hashfunktion handelt es sich um eine Funktion, die zu einer Eingabe beliebiger Länge eine Art Prüfsumme mit fixer Länge berechnet, den sogenannten Hash. Es soll einfach sein so einen Hash für eine Eingabe zu berechnen, aber zu einem Hash eine Eingabe zu finden, die diesen Hash besiztz, oder eine Nachricht zu verändern, sodass der Hash sich nicht verändert, soll praktisch nicht möglich sein. Damit eine solche Hashfunktion H als kryptographisch sicher gilt, erwartet man folgende grundlegende Eigenschaften von ihr:

1. Einwegfunktion:
   Für einen gegebenen Hash h kann nicht effizient eine Nachricht M gefunden werden, sodass $H(M) = h$.

2. Schwache Kollisionsresistenz:
   Zu einer gegebenen Nachricht M soll es nicht möglich sein, effizient eine zweite Nachricht M' zu finden, die den gleichen Hash besitzt, sodass $H(M) = H(M')$ gilt.

3. Starke Kollisionsresistenz:
   Es soll nicht möglich sein, effizient zwei Nachrichten M und M' zu finden, die den selben Hashwert gesitzen, also $H(M) = H(M')$ [RS04]

Ob eine Hashfunktion nun diese Anforderungen erfüllt, hängt davon ab, welche Mächtigkeit man einem Angreifer erlaubt, der versucht diese Probleme zu lösen. Für klassische

Name	Kapazität c	Blocklänge r	Hash-Länge
SHA3-224	448 Bit	1152 Bit	224 Bit
SHA3-256	512 Bit	1088 Bit	256 Bit
SHA3-384	768 Bit	832 Bit	384 Bit
SHA3-512	1024 Bit	576 Bit	512 Bit

Table 3.1: Übersicht über die verschiedenen SHA-3 Hashfunktionen

Computer betrachtet man typischerweise Algorithmen aus der Komplexitätsklasse BPP (Bounded-Error Probabilistic Polynomial Time). Diese Algorithmen sind probabilistisch und müssen nur mit einer Wahrscheinlichkeit von $p > 2/3$ das richtige Ergebnis ausgeben. Es ist die mächtigste Klasse klassisch probabilistischer Algorithmen, weshalb man die Angreifer aus dieser Klasse wählt. Quantencomputer sind jedoch in einigen Aspekten deutlich mächtiger als klassische Computer und auch wenn sie bisher noch um mehrere Größenordnungen zu klein sind, um moderne Hashfunktionen zu brechen, untersucht man schon länger die Sicherheit von kryptographischen Systemen wie Hashfunktionen oder Verschlüsselungen gegen Quantenalgorithmen. Analog zur Komplexitätsklasse BPP gibt es daher für Quantenalgorithmen die Klasse BQP (Bounded Error Quantum Polynomial Time). In 3.4 werden wir uns die Sicherheitseigenschaften von SHA-3 genauer anschauen.

Um Anfragen beliebiger Länge bearbeiten zu können, bestehen Hashfunktionen typischerweise aus vier Teilen. Mit Hilfe einer *Padding*-Funktion wird die Eingabe so erweitert, dass die Länge ein ganzzahliges Vielfaches einer von der Hashfunktion benötigten Blocklänge ergibt. Die Eingabe kann so in mehrere gleich große Blöcke eingeteilt werden, die nacheinander weiterverarbeitet werden. Aus einer *Kompressionsfunktion* oder, wie im Fall von SHA-3, einer *Permutation* wird dann eine Funktion konstruiert, die nacheinander die Blöcke mit dem Ergebnis der Kompression/Permutation kombiniert und weiterverarbeitet. Während viele bekannte Hashfunktionen dazu auf die Merkle-Damgård-Konstruktion zurückgreifen, verwendet SHA-3 die sogenannte *Schwammkonstruktion*. Wie diese genau aussieht und wie sie funktioniert, werden wir später in Abschnitt 3.3 sehen.

3.1 Keccak-Permutation

Als Grundlage für alle SHA3-Funktionen dient eine Instanz der Keccak-Permutationsfamilie KECCAK-p. Für eine kryptographische Sicherheitsanalyse betrachtet man in der Regel das asymptotische Laufzeitverhalten eines Angreifers. Hierzu benötigt man eine Funktion mit variablem Sicherheitsparameter, damit diese Analyse durchgeführt werden kann. Wir interessieren uns hier allerdings nur für die konkrete Instanz mit einem festen Sicherheitsparameter, die vom SHA-3 Standard [Dwo15] für die in Tabelle 3.1 aufgeführten Hashfunktionen festgelegt wird. Die genaue Definition mit variablem Sicherheitsparameter ist zum Nachlesen auch im Standard [Dwo15] aufgeführt. Wir wollen uns nun zuerst einmal den Zustandsvektor ein wenig genauer anschauen, auf dem die Permutation arbeitet, bevor wir uns die fünf Unterfunktionen ansehen, aus denen die KECCAK-p-Funktion aufgebaut ist.

3.1.1 Zustandsvektor

Die KECCAK-p-Funktion arbeitet auf einem 1600 Bit breiten Zustandsvektor, der *State Array* genannt wird. Am besten lässt sich ein solches State Array $\mathbf{A}$ als einen dreidimensionalen Block aus 5x5x64 Bits vorstellen, siehe Abbildung 3.1. Wir können dabei jedes Bit von $\mathbf{A}$ über die drei Indizes x, y und z identifizieren. Genauer gilt

$$\mathbf{A}[x][y][z] \in \{0,1\} (\forall x, y \in \{0, ..., 4\};\ z \in \{0, ..., 63\})$$

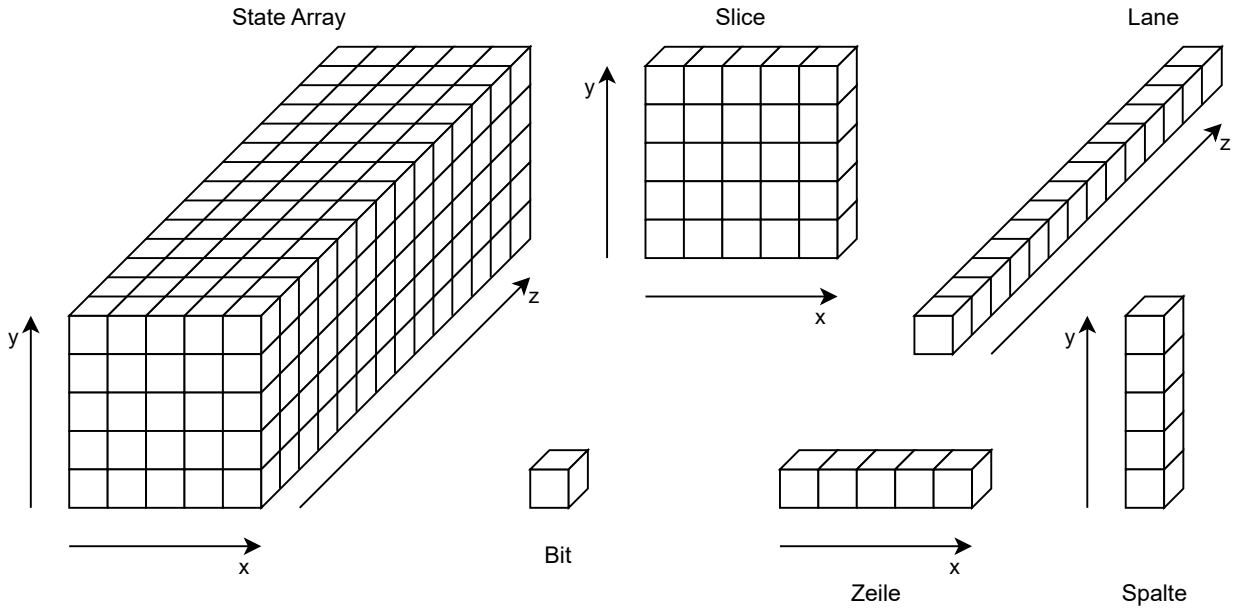

Figure 3.1: Blockrepräsentation des State Array

Außerdem nennen wir

$\mathbf{A}[x][y]$	$\coloneqq (\mathbf{A}[x][y][0], ..., \mathbf{A}[x][y][63])$	eine *Lane* von $\mathbf{A}$,
$\mathbf{A}[.][x][y]$	$\coloneqq (\mathbf{A}[0][y][z], ..., \mathbf{A}[4][y][z])$	eine *Zeile* von $\mathbf{A}$,
$\mathbf{A}[x][.][z]$	$\coloneqq (\mathbf{A}[x][0][z], ..., \mathbf{A}[x][4][z])$	eine *Spalte* von $\mathbf{A}$,
$\mathbf{A}[.][.][z]$	$\coloneqq (\mathbf{A}[.][0][z], ..., \mathbf{A}[.][4][z])$	einen *Slice* von $\mathbf{A}$

Über die folgende Konvention können ein eindimensionaler Bitvektor $\mathbf{V} \in \{0,1\}^{1600}$ und ein State Array $\mathbf{A}$ ineinander umgewandelt werden:

$$\mathbf{A}[x][y][z] \coloneqq \mathbf{V}[64(5y+x)+z] \forall x, y = 0, ..., 4; z = 0, ..., 63$$

Die Lanes werden also der Reihe nach erst in x-Richtung und dann in y-Richtung mit dem Inhalt von $\mathbf{V}$ gefüllt.

3.1.2 Unterfunktionen

Die KECCAK-p-Funktion ist aus einer Rundenfunktion *Rnd* aufgebaut, die mehrfach hintereinander ausgeführt wird. Diese Rundenfunktion ist wiederum aus fünf Unterfunktionen aufgebaut, die alle auf dem State Array arbeiten. Alle folgenden Funktionen basieren auf den Definitionen im SHA-3-Standard [Dwo15].

3.1.2.1 Theta-Unterfunktion

Die erste Unterfunktion der Rundenfunktion ist Theta (θ). Sie modifiziert jedes Bit eines State Arrays, indem sie, wie in Abbildung 3.2 skizziert, zwei benachbarte Spalten auf das Bit aufsummiert. Das Symbol $\oplus$ steht dabei für die XODER-Verknüpfung und $rotl(x,k)$ beschreibt eine Linksrotation einer Lane x um eine Distanz k, wobei das MSB mit $z = 63$

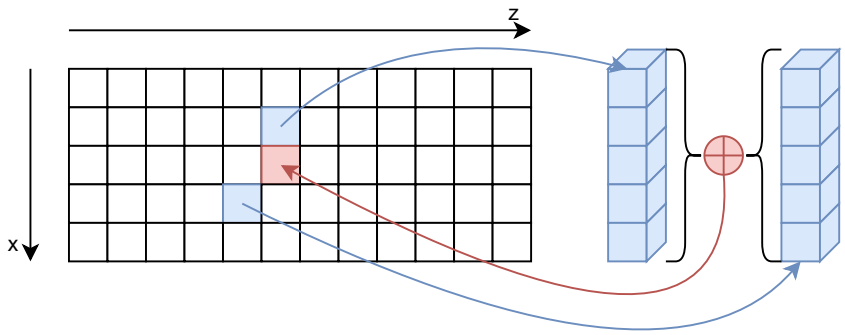

Figure 3.2: Spaltensummierung der θ-Funktion

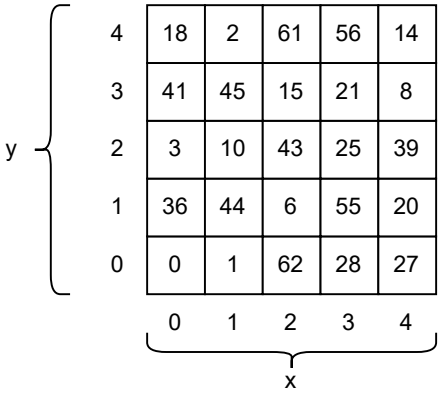

Figure 3.3: Rotationsdistanzen der Lanes für ρ

und das LSB mit $z = 0$ indiziert wird.

$$
\begin{aligned}
\theta(\mathbf{A}) &\coloneqq \mathbf{A}' \text{ mit} \\
\mathbf{C}[x] &\coloneqq \mathbf{A}[x][0] \oplus ... \oplus \mathbf{A}[x][4] && \forall x = 0, ..., 4 \\
\mathbf{D}[x] &\coloneqq \mathbf{C}[(x-1) mod\ 5] \oplus rotl(C[(x+1) mod\ 5], 1) && \forall x = 0, ..., 4 \\
\mathbf{A}'[x][y] &\coloneqq \mathbf{A}[x][y] \oplus \mathbf{D}[x] && \forall x = 0, ..., 4;\ y = 0, ..., 4
\end{aligned}
$$

Die Hilfsvariable $\mathbf{C}$ hält hierbei für alle x jeweils eine Lane, die an jeder Stelle die Summe aller Bits der entsprechenden Spalte enthält. In $\mathbf{D}$ werden für alle x immer zwei Lanes an Spaltensummen aufaddiert, wobei immer eine Lane um eine Stelle verschoben wird um das in 3.2 beschriebene Muster zu erhalten.

3.1.2.2 Rho-Unterfunktion

Bei der Rho-Unterfunktion (ρ) handelt es sich um eine einfache Bitrotation der einzelnen Lanes. Bis auf die Lane bei $x = 0$ und $y = 0$ werden alle Lanes um eine konstante Distanz nach links rotiert. Die genauen Distanzen sind in Abbildung 3.3 aufgeführt.

$$
\begin{aligned}
\rho(\mathbf{A}) &\coloneqq \mathbf{A}' \text{ mit} \\
\mathbf{A}'[x][y] &\coloneqq rotl(\mathbf{A}[x][y], \mathbf{D}[x][y])\ \forall x = 0, ..., 4;\ y = 0, ..., 4 \\
\mathbf{D}: &\ \text{Eine 5x5 Matrix an Konstanten, siehe Abb. 3.3}
\end{aligned}
$$

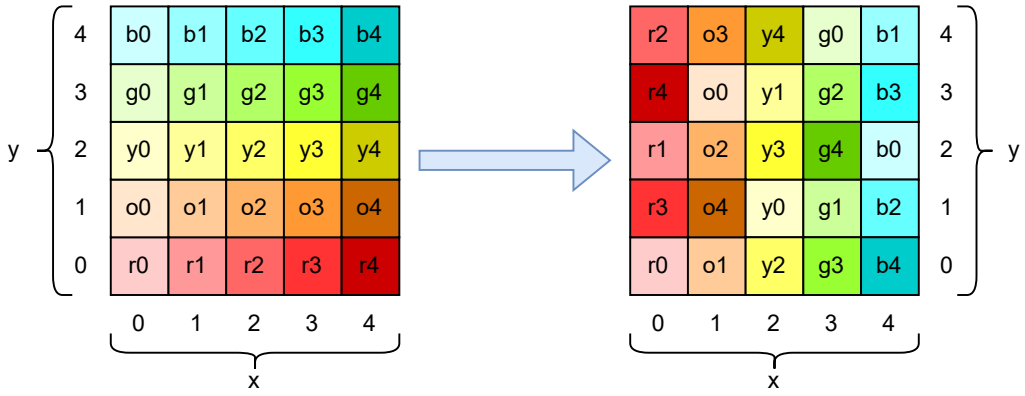

Figure 3.4: Visualisierung der *pi*-Permutation

3.1.2.3 Pi-Unterfunktion

Die Pi-Unterfunktion (π) vertauscht die Lanes eines State Arrays untereinander nach einer einfachen Vorschrift:

$$\pi(\mathbf{A}) \coloneqq \mathbf{A}' \text{ mit}$$
$$\mathbf{A}'[x][y] \coloneqq \mathbf{A}[(x+3y)\ mod\ 5][x]\ \forall x = 0, ..., 4;\ y = 0, ..., 4$$

Da der Ring $\mathbb{Z}/5\mathbb{Z}$ nullteilerfrei ist, ist die Indexberechnung $(x+3y)\ mod\ 5$ injektiv für ein festes x. Es sind also wirklich wieder alle Lanes im Ergebnis $\mathbf{A}'$ enthalten. In Abbildung 3.4 ist *pi* auch nochmal veranschaulicht.

3.1.2.4 Chi-Unterfunktion

Im Gegensatz zu allen anderen Unterfunktionen ist Chi (χ) nicht affin-linear. Interessanterweise ist sie trotzdem invertierbar, solange die Anzahl an Spalten ungerade ist [Dae95], in unserem Fall fünf. Das bedeutet, dass die Keccak-p Rundenfunktion sowie die Keccak-p Permutation invertierbar ist. Trotzdem eignet sie sich, wie wir sehen werden, um eine sichere Hashfunktion zu konstruieren, da die Art und Weise wie sie verwendet wird, die Nicht-Invertierbarkeit nicht benötigt.

$$\chi(\mathbf{A}) \coloneqq \mathbf{A}' \text{ mit}$$
$$\mathbf{A}'[x][y] \coloneqq \mathbf{A}[x][y] \oplus ((\sim \mathbf{A}[(x+1)\ mod\ 5][y]) * \mathbf{A}[(x+2)\ mod\ 5][y]) \quad \forall x = 0, ..., 4;\ y = 0, ..., 4$$

Die *chi* invertiert jedes Bit genau dann, wenn sein direkt benachbartes Bit in der Zeile 0 und der direkte Nachbar dessen 1 ist.

3.1.2.5 Iota-Unterfunktion

Die letzte Unterfunktion Iota (ι) modifiziert lediglich die Lane an Position (x,y) = (0,0), indem sie sie mit einer Rundenkonstante per bitweisem XODER kombiniert. Die genauen Werte der Rundenkonstanten C(r) sind in Tabelle 3.2 dargestellt. Damit wir nachher die Rundenfunktion besser darstellen können, definieren wir neben der zweistelligen Funktion

Rundenindex r	Rundenkonstante $C(r)$	Rundenindex r	Rundenkonstante $C(r)$
0	0x1	1	0x8082
2	0x800000000000808a	3	0x8000000080008000
4	0x808b	5	0x80000001
6	0x8000000080008081	7	0x8000000000008009
8	0x8a	9	0x88
10	0x80008009	11	0x8000000a
12	0x8000808b	13	0x800000000000008b
14	0x8000000000008089	15	0x8000000000008003
16	0x8000000000008002	17	0x8000000000000080
18	0x800a	19	0x800000008000000a
20	0x8000000080008081	21	0x8000000000008080
22	0x80000001	23	0x8000000080008008

Table 3.2: ι-Rundenkonstanten für die einzelnen Runden der Keccak-f Permutation

$\iota(\mathbf{A}, r)$ noch die über dem Rundenindex r parametrisierte Funktion $\iota_r(\mathbf{A})$.

$$\begin{aligned} \iota(\mathbf{A}, r) &\coloneqq \mathbf{A}' \text{ mit} \\ \mathbf{A}'[x][y] &\coloneqq \begin{cases} \mathbf{A}[x][y] \oplus C(r), & x = 0 \text{ und } y = 0 \\ \mathbf{A}[x][y], & \text{sonst} \end{cases} \\ \iota_r(\mathbf{A}) &\coloneqq \iota(\mathbf{A}, r) \end{aligned}$$

3.1.3 KECCAK-p

Die fünf Unterfunktionen werden zur Rundenfunktion Rnd_r kombiniert, mit der dann die Permutationsfunktion KECCAK-p definiert wird:

$$\begin{aligned} Rnd_r(\mathbf{A}) &\coloneqq (\iota_r \circ \chi \circ \pi \circ \rho \circ \theta)(\mathbf{A}) \; \forall r \in \{0, ..., 23\} \\ \text{KECCAK-p}(\mathbf{A}) &\coloneqq (\bigcirc_{i=0}^{23} Rnd_i)(\mathbf{A}) \end{aligned}$$

Der Index der Rundenfunktion Rnd_r beschriebt die aktuelle Runde von 0 bis 23. Das Symbol $\bigcirc$ soll analog zum Summenzeichen $\sum$ die Funktionenkomposition bezeichnen:

$$\bigcirc_{i=1}^{n} f_i \coloneqq f_n \circ ... \circ f_1$$

3.2 Padding-Funktion $pad10^*1$

Um eine Eingabe beliebiger Länge gescheit verarbeiten zu können, wird eine Padding-Funktion verwendet. Diese nimmt eine Eingabe beliebiger Länge entgegen und erzeugt ein einfaches dynamisches Datenmuster, sodass, wenn man es an die Eingabe anhängt, das Ganze eine länge hat, die ein Vielfaches der gewünschten Blocklänge ist. SHA-3 verwendet

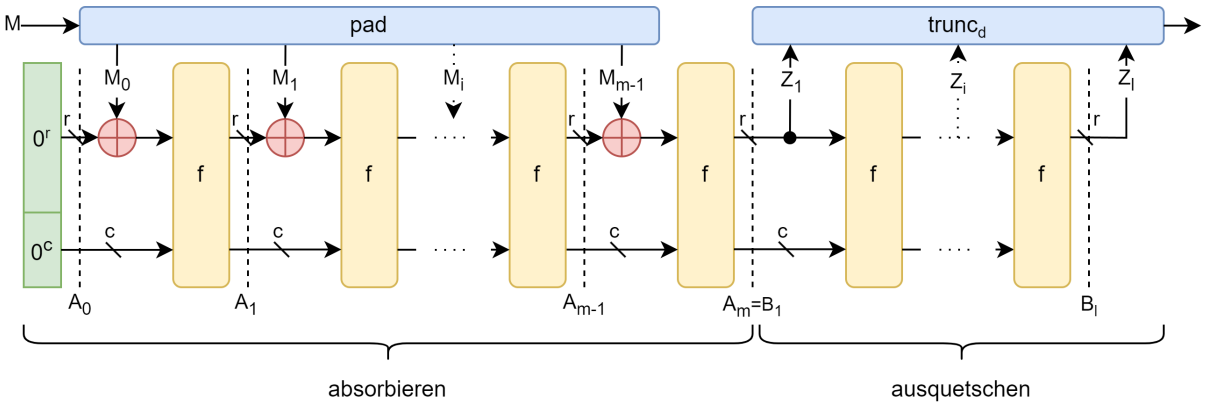

Figure 3.5: Aufbau der Schwammkonstruktion [Cite missing]

als Padding die Funktion $pad10^*1$. Diese erzeugt, wie der Name vermuten lässt, einen Bitvektor, der bis auf das erste und letzte Bit nur aus Nullen besteht.

$$\begin{aligned} &\text{Für eine Eingabe} && M \in \{0,1\}^*, \\ &\text{und eine verlangte Blocklänge} && r \in \mathbb{N} \\ &\text{ist } pad10^*1(r, M) \coloneqq 1 \parallel 0^{(-|M|-2) mod\ r} \parallel 1 \end{aligned}$$

$\parallel$ ist dabei die Konkatenation zweier Bitstrings.

3.3 Schwammkonstruktion

Zur Komprimierung der Eingabe verwendet SHA-3 die sogenannte Schwammkonstruktion. Sie erlaubt es eine Eingabe beliebiger Länge auf eine Ausgabe einer beliebigen anderen Länge d abzubilden. Dazu wird die Eingabe, wie in Abbildung 3.5 veranschaulicht, erst mit Hilfe einer Padding-Funktion in mehrere gleich große Blöcke einer festgelegten Länge aufgeteilt. Die Eingabeblöcke werden dann der Reihe nach vom Schwamm "absorbiert". Danach werden auf ähnliche Weise die Ausgabeblöcke aus dem Schwamm "ausgequetscht". Diese Ausgabeblöcke werden dann zur finalen Ausgabe zusammengesetzt. Wenn die Ausgabelänge kein Vielfaches der Blocklänge sein sollte, wird der Rest einfach abgeschnitten. Die genaue Definition der Schwammkonstruktion über einer Transformation f mit einer Paddingfunktion pad sieht folgendermaßen aus:

Seien $n \in \mathbb{N}$	die Transformationsbreite,
$c \in \{1, ..., n\}$	die Kapazität (Anzahl "versteckter" Bits),
$r \in \{1, ..., n\}$	die Blocklänge,
$M \in \{0,1\}^*$	die zu verarbeitende Nachricht,
$m \in \mathbb{N}$	die Anzahl an Blöcken, in die M eingeteilt wird,
$d \in \mathbb{N}$	die gewünschte Ausgabe,
$l \coloneqq \lceil \frac{d}{r} \rceil$	die benötigte Blockanzahl an Ausgabe,
$pad : \mathbb{N}x\{0,1\}^* \to (\{0,1\}^r)^+$	eine Padding-Funktion,
$f : \{0,1\}^n \to \{0,1\}^n$	eine Transformation.

Dann ist die Schwammkonstruktion $SPONGE[f, pad, c](M, d)$ definiert als:

$$
\begin{aligned}
SPONGE[f, pad, c](M, d) &\coloneqq \mathbf{Z}[0] \parallel ... \parallel \mathbf{Z}[d-1] \; mit \\
r &\coloneqq 1600 - c \\
M_1 ... M_m &\coloneqq M \parallel pad(r, M) \text{ wobei } |M_i| = r \; \forall i = 1, ..., m \\
\mathbf{A}_0 &\coloneqq 0^n \\
\mathbf{A}_i &\coloneqq f(\mathbf{A}_{i-1} \oplus (M_{i-1} \parallel 0^c)) && \forall i = 1, ..., m \\
\mathbf{B}_1 &\coloneqq \mathbf{A}_m \\
\mathbf{B}_i &\coloneqq f(\mathbf{B}_{i-1}) && \forall i = 2, ..., l \\
\mathbf{Z}_i &\coloneqq \mathbf{B}_i[0] \parallel ... \parallel \mathbf{B}_i[r-1] \\
\mathbf{Z} &\coloneqq \mathbf{Z}_1 \parallel ... \parallel \mathbf{Z}_l
\end{aligned}
$$

Die Kapazität c bestimmt dabei wie viele Bits des internen Zustands $\mathbf{A}$ der Schwammkonstruktion von neuen Eingabeblöcken nicht verändert werden dürfen.

3.3.1 SHA3-Hashfunktionen

Die in Tabelle 3.1 aufgelisteten Hashfunktionen sind nun Instanzen dieser Schwammkonstruktion:

$$
\begin{aligned}
\text{SHA3-224}(M) &\coloneqq SPONGE[\text{KECCAK-p}, pad10^*1, 448](M \parallel 01, 224), \\
\text{SHA3-256}(M) &\coloneqq SPONGE[\text{KECCAK-p}, pad10^*1, 512](M \parallel 01, 256), \\
\text{SHA3-384}(M) &\coloneqq SPONGE[\text{KECCAK-p}, pad10^*1, 768](M \parallel 01, 256), \\
\text{SHA3-512}(M) &\coloneqq SPONGE[\text{KECCAK-p}, pad10^*1, 1024](M \parallel 01, 512)
\end{aligned}
$$

Die zwei Extrabits "01", die an die Nachricht angefügt werden, dienen nur dazu die erzeugten Hashes von den Ergebnissen anderer Betriebsmodi der KECCAK-p Permutation zu unterscheiden, wie beispielsweise den beiden SHAKE-Funktionen.

3.4 Sicherheitseigenschaften

Nun ist erstmal noch überhaupt nicht klar, wieso es sich bei den 3.3.1 definierten Funktionen um eine Einwegfunktion handelt. Schließlich verwendet sie eine sehr leicht invertierbare

Permutation. Dazu wollen wir uns die Schwammkonstruktion noch einmal etwas genauer anschauen, in diesem Fall am Beispiel von SHA3-256, wobei wir nur einen Block hashen. Um die Permutation invertieren zu können, muss allerdings die gesamte Eingabe vorliegen. Da über den Hash h einer Nachricht M allerdings nur 256-Bit der Ausgabe der Permutation bekannt sind, kann die Funktion nicht direkt invertiert werden. Auch kann nicht einfach eine beliebige Belegung für den restlichen Teil gewählt werden, da am Ende nur ein Teil der Eingabe durch den Nachrichtenblock festgelegt werden kann. Die Schwierigkeit bei der Invertierung der Schwammfunktion entsteht also durch die Kombination aus Kapazität und dem Verwerfen eines Teils des Ergebnisses. Diese Überlegung reicht natürlich noch nicht als Beweis, aber sie gibt einen intuitiven Einblick das zugrunde liegende Problem. Bertoni et al. [BDPA08] bewiesen bereits, dass die Schwammkonstruktion selbst, wenn sie mit einer zufälligen Permutation initialisiert wird, für einen klassischen Algorithmus von einem Zufallsorakel undifferenzierbar ist, eine Eigenschaft, die von Maurer et al. [MRH04] eingeführt wurde als Generalisierung der Ununterscheidbarkeit. Czajkowski [Cza21] Zeigte außerdem, dass diese Eigenschaft auch für Quantenalgorithmen gilt. Undifferenzierbarkeit von einem Zufallsorakel ist eine sehr starke Eigenschaft für kryptographische Systeme, da aus ihr direkt viele Sicherheitseigenschaften folgen [BDPA08]. Damit diese Eigenschaft für SHA-3 gilt, muss allerdings die zugrundeliegende Permutation KECCAK-p ebenfalls undifferenzierbar von einer zufälligen Permutation sein. Einen solchen Beweis gibt es bisher nicht. Allerdings ist beweisbar, dass KECCAK-p gegen alle bekannten Angriffe auf andere Hashfunktionen besteht, die zum Beispiel Techniken wie die lineare oder differenzielle Kryptoanalyse verwenden [PA11].

Chapter 4

Implementierung

4.1 Vorgehensweise

Ziel ist es, einen Beschleuniger zu entwickeln, der die Berechnung von SHA-3 möglichst effizient durchführt. Da alle SHA3-Funktionen die gleiche Instanz von KECCAK-p verwenden, ist die Wahl der konkret zu implementierenden SHA3-Funktion praktisch egal, sie unterscheiden sich nur in der Größe der Eingabeblöcke, die mit dem internen State Array kombiniert werden. Daher legen wir uns hier auf SHA3-256 fest, die der Beschleuniger implementieren soll. Dabei müssen aber alle Voraussetzungen der gewählten Architektur eingehalten werden. Im Falle des icore sind das die Größenbeschränkung von maximal fünf Atomen mit jeweils 1600 LUTs, sowie das Implementieren der Kommunikationsschnittstelle der Atome, sodass sie sowohl untereinander, als auch mit dem icore, Daten austauschen können. Da quasi der gesamte Rechenaufwand von SHA-3 aus der wiederholten Berechnung der KECCAK-p-Funktion besteht, soll der hier entworfene Beschleuniger genau diese KECCAK-p-Funktion berechnen. Für den Entwurf verwenden wir hier ein iteratives Design-Verfahren. Angefangen wird mit einem Beschleuniger, der die Architekturbeschränkungen komplett ignorieren darf, um eine maximale Berechnungsgeschwindigkeit zu erzielen. Der Platz des Beschleunigers sollte dennoch sinnvoll genutzt werden. Auf diesem Entwurf aufbauend können dann Strategien entwickelt werden, um die Berechnung so aufzuteilen, dass der entstehende Beschleuniger nach und nach die Voraussetzungen der Architektur erfüllt.

4.2 Erster Entwurf

4.2.1 Entwurfsziele

Der triviale Ansatz, die KECCAK-p-Funktion direkt in einem riesigen kombinatorischen Netz zu berechnen ist zwar theoretisch am schnellsten, jedoch wenig platzeffizient. Da die KECCAK-p-Funktion aus 24 nahezu identischen Runden aufgebaut ist, müsste auch jede einzelne ihrer Berechnungen nebeneinander realisiert werden. Da Ein- und Ausgabedaten mit jeweils 1600 Bits in der gleichen Größenordnung liegen wie die Größenbeschränkung des icore, ist es deutlich sinnvoller, die Berechnung der Runden nacheinander über dieselbe Realisierung der Rundenfunktion durchzuführen. Ziel dieses Entwurfs ist daher die Implementierung der Rundenfunktion in einem rein kombinatorischen Netz, womit dann in jedem Takt jeweils eine Runde berechnet werden kann, bis schließlich die komplette

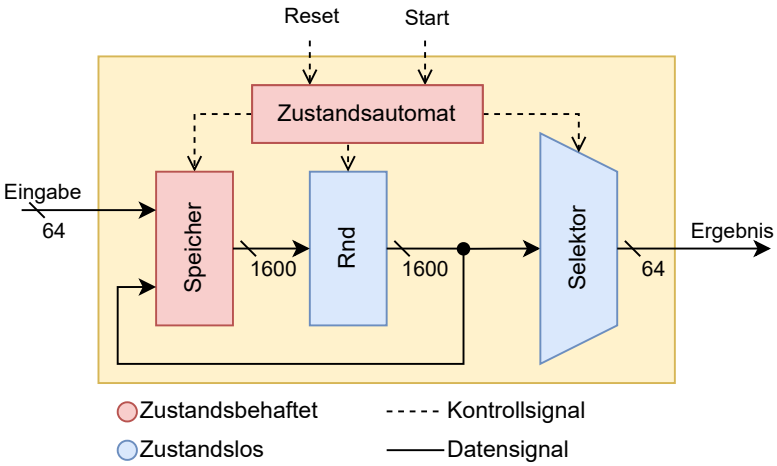

Figure 4.1: Aufbau des ersten Entwurfs

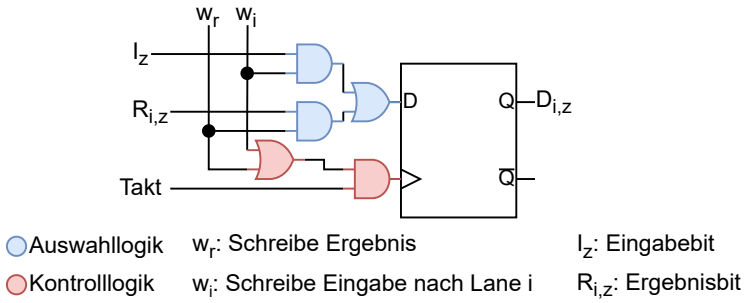

Figure 4.2: Speicherzelle des ersten Entwurfs

KECCAK-p-Funktion berechnet ist. Die Kommunikationsschnittstelle der Atome soll auch rudimentär implementiert werden, sodass das Einlesen und Ausgeben von Daten den dafür vorgesehenen 64-Bit Datenkanal nutzt und die Steuerung des Atoms über maximal 6 Kontrollbits durchgeführt wird.

4.2.2 Aufbau

In Abbildung 4.1 ist der grobe Aufbau des Beschleunigers zu sehen. Die Rnd-Komponente berechnet aus den im Speicher liegenden Daten genau die Rundenfunktion. Welche Runde sie berechnen soll, wird vom Zustandsautomaten bestimmt. Das Ergebnis kann nun in den Speicher zurückgeschrieben oder ausgegeben werden, wobei der Selektor immer jeweils eine Lane des Ergebnisses an die Ausgabe anlegt. Der Zustandsautomat ist bei der Ausgabe dafür verantwortlich, dass der Selektor alle Lanes nacheinander auswählt und somit das gesamte Ergebnis ausgibt. Der Speicher besteht aus 1600 Speicherzellen, sodass jedes Bit der Ein-/Ausgabe der Rundenfunktion darin gespeichert werden kann. Jenachdem, ob das Ergebnis der Rundenfunktion oder die Eingabe in den Beschleuniger übernommen werden soll, muss das entsprechende Bit ausgewählt werden. In Abbildung 4.2 ist eine Speicherzelle mit dieser Auswahllogik skizziert. Aus einem State Array $\mathbf{A}$ wird das Bit $\mathbf{A}[x][y][z]$ in der Speicherzelle $D_{i,z}$ mit $i = 5 * y + x$ gespeichert. Der Index i beschreibt also in welcher Lane, und der Index z an welcher Position innerhalb der Lane sich das Bit befindet. Die Kontrollsignale w_i und w_r bestimmen, ob ein Bit aus der Eingabe (I_z) oder aus dem Ergebnis von Rnd ($R_{i,z}$) übernommen werden soll. In der Auswahllogik (blau)

wird das entsprechende Bit ausgewählt und in der Kontrolllogik (rot) wird am nächsten Takt der Wert in die Speicherzelle geschrieben, falls eines der beiden Bits übernommen werden soll. Die Kontrolllogik muss dabei für jede Lane nur einmal realisiert werden, die Auswahllogik allerdings für jedes der 1600 Bits.

4.2.3 Bewertung

Der gesamte Entwurf besteht aus 3314 LUTs. Davon werden etwa 1600 LUTs für den Datenspeicher verwendet. Der Rest wird fast vollständig für die Berechnung der Rundenfunktion benötigt. Der Zustandsautomat und der Ergebnisselektor sind da verhältnismäßig klein. Insgesamt ist der Entwurf damit ungefähr doppelt so groß wie es die Architektur des icore vorgibt. Im Folgenden wollen wir uns daher ein paar Verbesserungen anschauen, mit denen die Größe des Beschleunigers angepasst werden kann.

4.2.4 Optimierungsansätze

Die naheliegendste Möglichkeit zur Verbesserung des Platzbedarfs besteht darin den Speicheraufwand zu reduzieren, da dieser zwar viel Platz einnimmt, aber an der Berechnung selbst nicht teilnimmt. Das Speichern der Daten außerhalb des Beschleunigers ist alleine betrachtet leider keine Option, da zur Berechnung der Rundenfunktion ja der Gesamte Datenblock im Atom wieder vorliegen muss. Um mit weniger Daten auf einmal arbeiten zu können, muss daher zuerst die Rundenfunktion in kleinere Teiloperationen aufgeteilt werden.

4.2.4.1 Aufspalten der Rundenfunktion

Um die Rundenfunktion in mehrere Operationen aufteilen zu können, ist eine genauere Untersuchung der Funktion selbst notwendig, um Teie ausfindig zu machen, die unabhängig voneinander berechnet werden können. Ziel ist es, möglichst große Abschnitte in der Berechnung zu finden, die die gleiche (oder zumindest sehr ähnliche) Operation unabhängig voneinander auf verschiedenen Daten berechnen. Diese Berechnungen können dann sequenziell statt parallel berechnet werden, wodurch der benötige Platz reduziert wird. Damit fällt die Aufteilung in die Teilfunktionen aus der Definition der Rundenfunktion θ, ρ, π, χ und ι_r weg, da diese Teile direkt voneinander abhängen und nicht gleichzeitig berechnet werden, sondern nacheinander. Allerdings lassen sich alle fünf Teilfunktionen in zwei Kategorien einteilen:

1. Slice-orientierte Funktionen Jeder Slice der Ausgabe hängt nur von einem oder zwei Slices der Eingabe ab, die anderen Slices werden ignoriert. Das sind genau die Funktionen, bei denen in der Definition (siehe 3.1.2) der z-Index nicht groß verändert wird, also θ, π, χ und ι_r.

2. Lane-orientierte Funktionen Analog handelt es sich hier um die Funktionen, bei denen jede Lane der Ausgabe nur von einer Lane der Eingabe abhängt. Für die Berechnung gemäß der Definition bedeutet das, dass die Indizes x und y nicht manipuliert werden werden. Zu den Lane-orientierten Funktionen zählen nur ρ und ι_r.

ι_r zählt hierbei in beide Kategorien, da es sich nur um die Aufaddierung einer Konstanten handelt und somit jedes Bit der Ausgabe nur von einem Bit und einer Konstanten abhängt.

Die Rundenfunktion kann so in verschiedene Abschnitte eingeteilt werden, die nur aus Slice-orientierten oder Lane-orientierten Teilfunktionen bestehen. Um eine möglichst gute Ausführungsgeschwindigkeit beizubehalten, ist es wichtig, die Anzahl dieser Abschnitte möglichst gering zu halten, da so in jedem Abschnitt ein möglichst großer Teil der Berechnung durchgeführt wird. Die Rundenfunktion Rnd_r besitzt drei dieser Abschnitte.

$$Rnd_r = \underbrace{\iota_r \circ \chi \circ \pi}_{\text{Slice-orientiert}} \circ \underbrace{\rho}_{\text{Lane-orientiert}} \circ \underbrace{\theta}_{\text{Slice-orientiert}}$$

Für die Berechnung von KECCAK-p lässt sich die Rundenfunktion leicht modifizieren, sodass sie nur noch zwei Abschnitte besitzt. Die Idee ist dabei, dass die Berechnung von θ jeweils an das Ende der vorherigen Runde verschoben wird. Durch Einsetzen der Definition von Rnd_r in KECCAK-p erhält man:

$$\begin{aligned}
\text{KECCAK-p} &= \bigcirc_{r=0}^{23} Rnd_r = \bigcirc_{r=0}^{23} \iota_r \circ \chi \circ \pi \circ \rho \circ \theta \\
&= \iota_{23} \circ \chi \circ \pi \circ \rho \circ \theta \circ (\bigcirc_{r=0}^{22} \iota_r \circ \chi \circ \pi \circ \rho \circ \theta) \\
&= \iota_{23} \circ \chi \circ \pi \circ \rho \circ (\bigcirc_{r=0}^{22} \theta \circ \iota_r \circ \chi \circ \pi \circ \rho) \circ \theta
\end{aligned}$$

KECCAK-p lässt sich dann mit der modifizierten Rundenfunktion ($RMod_r$) wieder kompakt zusammenfassen:

$$\begin{aligned}
\alpha_r &\coloneqq \iota_r \circ \chi \circ \pi \\
\beta_r &\coloneqq \theta \circ \alpha_r \\
\gamma_r &\coloneqq \begin{cases} \theta & , r = -1 \\ \alpha_{23} & , r = 23 \\ \beta_r & , \text{sonst} \end{cases} \\
RMod_r &\coloneqq \begin{cases} \gamma_{-1} & , r = -1 \\ \gamma_r \circ \rho & , sonst \end{cases} \\
\text{KECCAK-p} &= \iota_{23} \circ \chi \circ \pi \circ \rho \circ (\bigcirc_{r=0}^{22} \theta \circ \iota_r \circ \chi \circ \pi \circ \rho) \circ \theta \\
&= \alpha_{23} \circ \rho \circ (\bigcirc_{r=0}^{22} \beta_r \circ \rho) \circ \theta \\
&= \gamma_{23} \circ \rho \circ (\bigcirc_{r=0}^{22} \gamma_r \circ \rho) \circ \gamma_{-1} \\
&= (\bigcirc_{r=0}^{23} \gamma_r \circ \rho) \circ \gamma_{-1} \\
&= \bigcirc_{i=-1}^{23} RMod_r
\end{aligned}$$

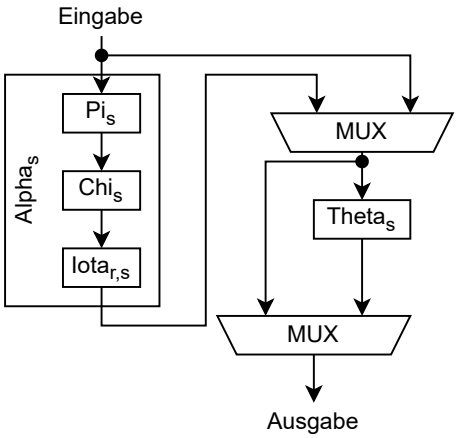

Figure 4.3: Implementierung der Gamma-Funktion
Die Funktionen π_s, χ_s und $\iota_{r,s}$ berechnen jeweils genau zwei Slices, sodass θ_s daraus einen Slice berechnen kann.

Der Vorteil von $RMod_r$ gegenüber Rnd_r ist wie anfangs motiviert, dass $RMod_r$ für jeden Index r nur zwei Abschnitte mit unterschiedlicher Orientierung besitzt (ρ und γ). Da γ eine Slice-orientierte Funktion ist, können alle Slices unabhängig voneinander nacheinander oder parallel berechnet. Auf diese Weise ist es möglich, den von γ benötigten Platz auf Kosten von mehr Berechnungszeit sehr stark zu reduzieren. In Abbildung 4.3 wird skizziert, wie die Berechnung von Gamma auf einzelnen Slices sehr platzeffizient implementiert werden kann, da nicht α, β und θ alle vollständig implementiert werden müssen. Über zwei Multiplexer können alle drei Teile berechnet werden.

4.2.4.2 BRAM als Datenspeicher

Um nun den Platzbedarf des Speichers zu reduzieren, gibt es mehrere Möglichkeiten. Eine davon besteht darin, die Daten nicht in einem Flip-Flop-Register zu speichern, sondern einen BRAM zur Speicherung von Daten zu verwenden. Leider lässt sich dieser Ansatz nicht direkt mit der Aufteilung der Rundenfunktion, wie oben beschrieben, kombinieren. Grund dafür ist die unterschiedliche Orientierung der Operationen, die auf den Daten durchgeführt werden. Aus einem Flip-Flop-Register können jederzeit beliebige Bits ausgelesen werden, sodass sowohl Slice- als auch Lane-orientierte Operationen direkt aus dem Speicher mit Daten versorgt werden können. Möchte man den BRAM als Datenspeicher verwenden, so muss für den Speicher eine Orientierung festgelegt werden. Um dann Operationen mit einer anderen Orientierung durchführen zu können, müsste jede Speicherstelle des BRAM nacheinander ausgelesen und daraus das gewünschte Datenobjekt zusammengesetzt werden. Daher wollen wir im zweiten Entwurf diesen Ansatz nicht verwenden, werden ihn aber im dritten Entwurf nochmal weiterverfolgen.

4.2.4.3 Aufspalten des Beschleunigers

Da ein Atom nicht ausreicht um den ganzen Datenblock zusammen mit der Rundenfunktion zu halten, kann der Beschleuniger auch in bis zu 5 Atome aufgeteilt werden, wobei jeder Block nur noch einen Teil des Datenblocks hält und auch nur für einen Teil der Daten die modifizierte Rundenfunktion aus 4.2.4.1 berechnet. Da das Ergebnis der Rundenfunktion im allgemeinen auch noch von den Daten anderer Blöcke abhängt, müssen diese Daten

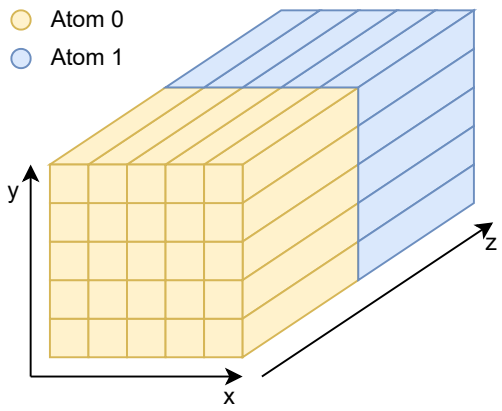

Figure 4.4: Lane-orthogonale Aufteilung des Datenblocks

über das Interface zwischen den Atomen ausgetauscht werden. Zwei Aspekte sind daher bei der Aufteilung wichtig:

1. Wie viele Blöcke sind sinnvoll? Mit höherer Anzahl an Blöcken nimmt die Datenmenge ab, die jeder Block speichern muss und da jeder Block über die Implementierung der Rundenfunktion verfügt, kann auch die Berechnung parallel auf den Atomen durchgeführt werden. Leider steigt mit der Anzahl der Atome auch die Menge an Datenabhängigkeiten zwischen den Atomen. Es gilt also herauszufinden an welchem Punkt die Kommunikationsschnittstelle zwischen den Atomen zum Bottleneck wird.

2. Wie werden die Daten am besten auf die Atome aufgeteilt? Die Daten müssen so auf die Atome verteilt werden, dass die Datenabhängigkeiten für die Rundenfunktion möglichst gering sind. Jedoch sollte das Muster auch nicht zu kompliziert sein. Für die Übertragung der Daten muss ein Kommunikationsprotokoll festgelegt werden, das bestimmt, welche Teile der Daten in welchem Takt ausgetauscht werden. Ist das Muster zu komplex, so benötigt die Implementierung des Protokolls zu viel Platz.

Weiterhin wäre es schön, wenn die verschiedenen Blöcke allesamt symmetrisch, also baugleich sind, es würden also alle Atome mit dem gleichen Beschleuniger konfiguriert und über ein Kontrollsignal wird am Anfang der Ausführung bestimmt für welchen Teil der Daten ein Atom verantwortlich ist. Dadurch ist der Beschleuniger leichter zu testen. Im Folgenden werden wir uns ein paar der naheliegendsten Aufteilungsmuster anschauen.

4.2.4.4 2-Block Lane-orthogonale Aufteilung

Spaltet man die Daten wie in Abbildung 4.4 gezeigt, sodass jeder Block jeweils 32 der insgesamt 64 Slices enthält, so kann jedes Atom sehr einfach die Gamma-Funktion für die von ihm gehaltenen Teil berechnen. Einzig die Slices 31 und 63 müssen ausgetauscht werden, da die Theta-Funktion für jeden Slice auch den benachbarten linken Slice benötigt. Die Berechnung der Rho-Funktion ist allerdings ein wenig komplizierter, da jede Lane durch Rho unterschiedlich weit rotiert wird. Es muss also im Kommunikationsprotokoll für jede Lane extra festgelegt welche Bits genau in welchem Takt ausgetauscht werden.

4.2.4.5 2-Block Spalten-orthogonale Aufteilung

Spaltet man die Daten entlang der Lanes wie in Abbildung 4.5, so muss für die Gamma-Funktion jeder Slice einmal zwischen den Atomen ausgetauscht werden. Die Berechnung

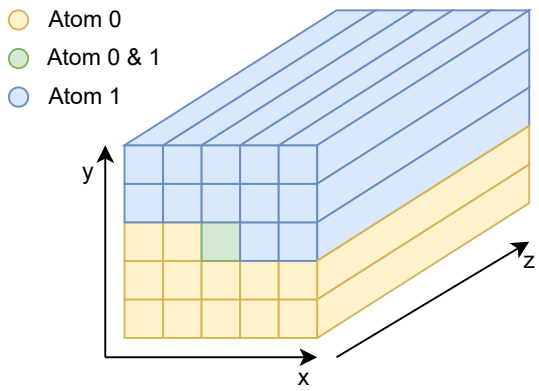

Figure 4.5: Spalten-orthogonale Aufteilung des Datenblocks

kann allerdings weiterhin parallel erfolgen. Auch die Rho-Funktion kann parallel berechnet werden und benötigt keinerlei Kommunikation.

4.2.4.6 Anmerkung zu Zeilen-orthogonalen Mustern

Die Gamma-Funktion benötigt ganze Slices für die Berechnung, wenn ein Slice in einem Schritt berechnet werden soll. Daher bestehen für Zeilen-orthogonale Aufteilungsmuster exakt die gleichen Vor- und Nachteile wie für Spalten-orthogonale Muster. Einzig für das Ergebnis ist ein Spalten-orthogonales Muster vorteilhaft, da das Endergebnis aus den ersten 4 Lanes besteht und nur dort alle 4 Lanes in einem Atom enthalten sind.

4.2.4.7 4-Block Muster

Für die Aufteilung in 4 Blöcke können so die vorherigen Muster mehrfach angewendet oder auch miteinander kombiniert werden. Der Speicheraufwand pro Atom sinkt hier zwar auf etwa 25% der gesammten Datenmenge, jedoch ist der zu erwartende Gewinn an LUTs nicht mehr so groß wie beim Schritt von 1 auf 2 Atome. Zudem steigt der Kommunikationsaufwand deutlich an, was nicht nur eine erhöhte Ausführungszeit mit sich bringt, sondern auch wieder mehr Platz im Atom benötigt.

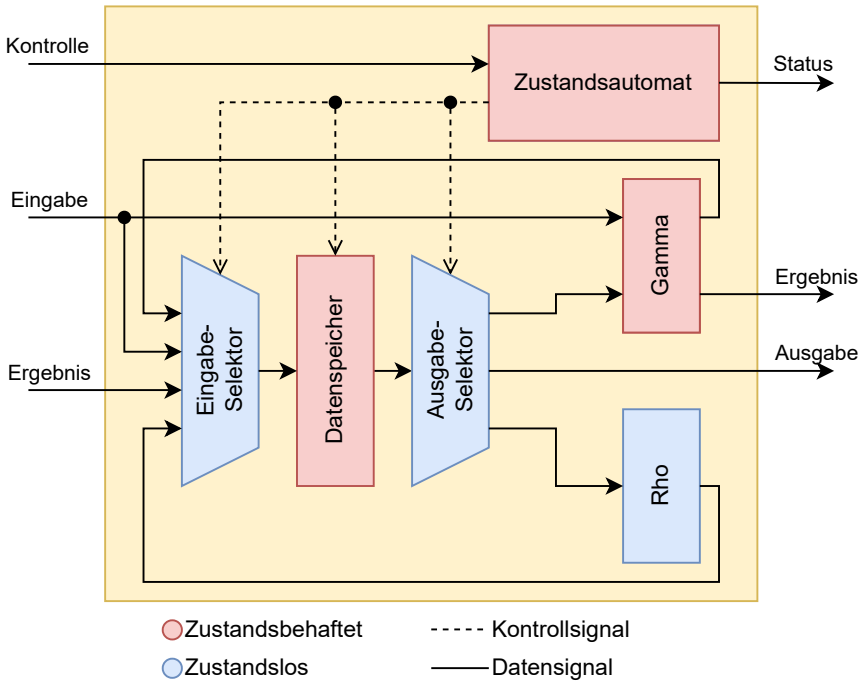

Figure 4.6: Atomaufbau des zweiten Entwurfs

4.3 Zweiter Entwurf

4.3.1 Entwurfsziele

Ziel des zweiten Entwurfs ist es, die Ideen aus 4.2.4 umzusetzuen und sicherzustellen, dass sie den gewünschten Effekt erzielen. Dazu wird der Beschleuniger in zwei Atome aufgeteilt, um die Datenmenge zu reduzieren, die jedes Atom halten muss. Für die Aufteilung wird das 2-Block Spalten-orthogonale Muster verwendet (siehe 4.2.4.5). Außerdem wird die Standard-Rundenfunktion durch die modifizierte Rundenfunktion aus 4.2.4.1 abgelöst. Damit die Komponenten miteinander arbeiten können und die Atome Daten miteinander austauschen können, braucht es zusätzlich noch einen Zustandsautomaten, der das Verhalten der Komponenten kontrolliert. Die Atome sollen so klein sein wie möglich und dürfen dabei ruhig ein wenig die Ausführungszeit erhöhen.

4.3.2 Aufbau

Abbildung 4.6 zeigt den Aufbau der beiden Atome *A0* und *A1*. Beide Atome teilen sich den gleichen Aufbau und bekommen ihre Rolle, den sogenannten *Atom-Index*, bei der Initialisierung per Kontrollsignal mitgeteilt. Durch die Aufteilung der Daten auf beide Atome speichert jetzt jedes Atom nur noch 13 der 25 Bits jedes Slices. Diesen Teil eines Slices nennen wir im Folgenden *Tile*. Die Speicherung der Tiles findet weiterhin in einem Flip-Flop-Register statt. Zusätzlich zu den Funktionen des Speichers aus dem ersten Entwurf bietet er noch die Möglichkeit, Eingabedaten mit dem aktuellen Speicherinhalt über ein XODER zu kombinieren. Dadurch können weitere Blöcke direkt eingelesen werden und es muss nicht das vollständige Ergebnis der KECCAK-p-Berechnung ausgegeben und danach die Kombination aus Ergebnis und neuen Daten wieder eingelesen werden. Die Funktionen ρ und γ sind in ihren eigenen Komponenten implementiert. Dabei kann

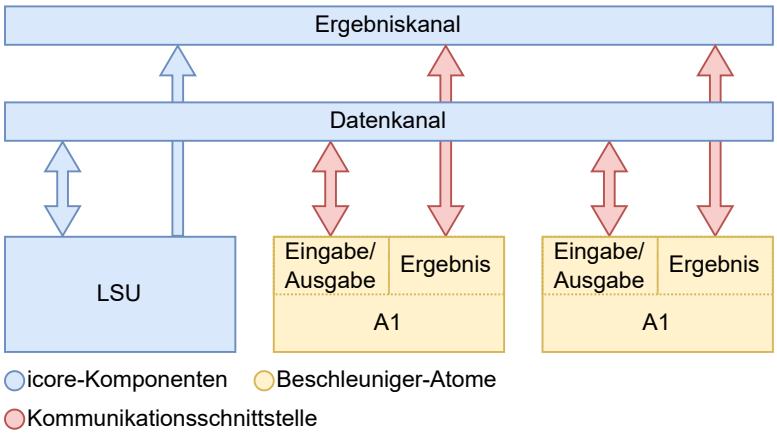

Figure 4.7: Integration des zweiten Beschleunigers in den icore
Unbenutzte Komponenten, Kontrollsignale, sowie die beiden nicht verwendeten Kanäle des Busses wurden weggelassen

der Rho-Block die komplette Funktion auf seinem Teil der Daten in einem Takt direkt berechnen. Der Gamma-Block berechnet pro Takt immer jeweils zwei Slices. Da für die Berechnung von Gamma auf den Slices immer nur die Hälfte jedes Slices im Atom gespeichert ist, muss die andere Hälfte der Eingabedaten über den Kommunikationskanals übertragen werden und auch die Hälfte des Ergebnisses muss wieder zurück übermittelt werden. Um das zu erlauben, wird der Kommunikationskanal in zwei 32-Bit breite Teile eingeteilt. Der Datenkanal übermittelt Daten direkt aus dem Speicher und der Ergebniskanal überträgt Ergebnisse der Gamma-Berechnung. Die Kanäle sind wie in Abbildung 4.7 dargestellt an den Bus der Fabric angebunden. Außerdem kann der Gamma-Block die Informationen zwischenhalten, die notwendig sind, um θ für den nächsten Slice zu berechnen. Auf diese Weise kann der Kommunikationsaufwand reduziert werden. Über den Ausgabeselektor werden aus dem Speicher die Bits ausgewählt, welche Bits aus dem Speicher an die Berechnungseinheiten Rho und Gamma angelegt werden und welche Bits über den Datenkanal der Kommunikationsschnittstelle ausgegeben werden. Der Eingabeselektor nimmt Daten aus dem Daten- und Ergebniskanal, sowie den Berechnungseinheiten Gamma und Rho entgegen und bestimmt, welche Teile an den Speicher weitergeleitet werden. Die Steuerung aller Komponenten übernimmt wieder der Zustandsautomat, der auch für die Einhaltung des Kommunikationsprotokolls zwischen den Atomen verantwortlich ist.

4.3.3 Ablauf einer Berechnung

4.3.3.1 Dateneingabe

Zum Einlesen der Daten können sowohl Daten- als auch Ergebnisbus gleichzeitig verwendet werden. So kann die LSU in jedem Kontrollschritt jeweils eine Lane an den Bus anlegen und die Atome greifen sich die Lanes ab, die sie speichern müssen. Die eingelesenen Lanes werden entweder direkt in den Speicher geschrieben oder über ein XODER mit dem aktuellen Speicherinhalt kombiniert, falls es sich nicht um den ersten Block der SHA-3-Berechnung handelt.

4.3.3.2 Rho

Für die Berechnung von ρ werden in jedem Atom alle Daten des Speichers gleichzeitig an den Rho-Block angelegt. Das Ergebnis wird dann über den Eingabeselektor wieder in den Speicher übernommen. Da es sich bei ρ um eine reine Permutation der Bits in den einzelnen Lanes handelt, existiert der Rho-Block in der Implementierung gar nicht wirklich, sondern die Bits der Ausgabe werden einfach permutiert wieder an den Selektor angelegt. Die ganze ρ-Funktion kann somit in einem einzigen Takt berechnet werden.

4.3.3.3 Gamma

Die Berechnung von Gamma wird in mehrere Teilschritte aufgeteilt. *A0* ist dabei für die Berechnung der Slices 0 bis 31 zuständig und textitA1 führt die Berechnung für die Slices 32 bis 63 durch. Damit ein neuer Slice berechnet werden kann, muss der vollständige Slice, sowie ein paar Informationen aus seinem Vorgänger im Atom vorliegen. Da für jeden Slice nur ein Tile im Speicher des zuständigen Atoms vorliegt, muss das andere Tile noch übertragen werden. Das geschieht über den Datenkanal zwischen den Atomen. Bevor die Berechnung aller Slices startet, werden die Slices 31 und 63 einmal ausgetauscht, da diese benötigt werden, um die Slices 0 und 32 zu berechnen. Für die restlichen Slices werden diese benötigten Daten immer im Takt vorher berechnet und können kurz in der Komponente gepuffert werden. Nach der Berechnung eines Slices muss das Ergebnis auch in beiden Atomen wieder übernommen werden. Dafür wird wieder ein Tile in den lokalen Speicher übernommen und das andere Tile wird über den Ergebniskanal an das andere Atom übertragen und dort gespeichert. In jedem Takt können somit insgesamt vier Slices gleichzeitig berechnet werden. Hier ist die genaue Abarbeitungsreihenfolge für einen Slice nochmal für textitA0 beschrieben. Jeder Punkt beschreibt dabei, was in einem Takt passiert:

1. Die Daten für textitA1 werden aus dem Datenspeicher gelesen und an den Datenkanal angelegt.
2. Die Daten für textitA1 befinden sich im Register des Datenkanals
3. Die Daten für textitA1 sind bei textitA1 eingetroffen. Gleichzeitig treffen auch die von textitA1 gesendeten Daten bei textitA0 ein. Die erhaltenen Daten werden mit den Daten aus dem Speicher von textitA0 zu vollständigen Slices kombiniert und der Berechnungseinheit bereitgestellt.
4. Die Berechnungseinheit berechnet das Ergebnis und gibt es aus.
5. Das Ergebnis wird in zwei Tiles aufgeteilt. Eines wird im Datenspeicher übernommen und das andere wird am Ergebniskanal angelegt.
6. Das Erbebnis im Ergebnisbus befinden sich im Register des Ergebniskanals
7. Das Ergebnis trifft bei textitA1 ein. Gleichzeitig trifft auch das Ergebnis von textitA1 bei textitA0 ein. Das erhaltene Ergebnis wird im Datenspeicher übernommen.

Die maximale Anzahl an Slices, die gleichzeitig in einem Atom berechnet werden kann, ergibt sich in diesem Fall aus der stark beschränkten Bandbreite des Kommunikationskanals. In dem 32 Bit breiten Daten-/Ergebniskanal können maximal zwei Tiles in

einem Takt übertragen werden. Um die gesamte γ-Funktion zu berechnen, muss die oben aufgeführte Berechnungsabfolge also 16 Mal mit jeweils 2 Slices durchgeführt werden. Die Berechnung der 7 Schritte wird in einer Pipeline durchgeführt, die zweite Berechnungsabfolge beginnt also nicht erst, wenn die erste Abfolge abgeschlossen ist, sondern direkt, nachdem der erste Schritt der vorherigen Abfolge abgeschlossen ist. Dadurch beträgt die Berechnungsdauer aller 16 Ausführungen $7 + (16 - 1) = 22$ Schritte. Dabei ist allerdings der Syncronisationsaufwand, wie zum Beispiel der Austausch der Slices 31 und 63, nicht enthalten.

4.3.4 Bewertung

Die Ausführungszeit für eine Iteration der modifizierten Rundenfunktion ist mit einem Faktor von 20 wie erwartet deutlich langsamer als die Implementierung des ersten Entwurfs. Dies ist wie bereits erklärt hauptsächlich der Aufteilung der Gamma-Funktion in 16 Teilschritte geschuldet, sowie der damit einhergehenden Verzögerung. Anders jedoch als erwartet, ist die Größe der Atome durch das Aufteilen der Berechnung und des Datenspeichers nicht wie gewünscht gesunken. Tatsächlich ist der Entwurf mit seinen 4643 LUTs nochmal um gut 40% größer. Dafür gibt es zwei wesentliche Gründe: den Zustandsautomaten sowie die Komplexität der Speicherzugriffsmuster, die in der Überlegung für das Design nicht bedacht wurden.

4.3.4.1 Zustandsautomat

Der Zustandsautomat besteht aus einem Iterator, der in jedem Takt hochgezählt wird und anhand dessen die Steuersignale für die anderen Komponenten genertiert werden. Entgegen der ursprünglichen Annahme, dass seine Größe aufgrund der Einfachheit der Aufgabe vernachlässigbar ist, nimmt er in diesem Design über 300 LUTs, also etwa 20% der Maximalgröße von 1600 LUTs ein. Auch wenn sich die konkrete Implementierung noch optimieren lässt, so ist klar geworden, dass die weitere Erhöhung der Berechnungskomplexität mit Bedacht durchgeführt werden muss, da der Zustandsautomat dadurch nur noch größer wird.

4.3.4.2 Schreib- und Lesemuster

Im ersten Entwurf wird der Wert jedes Bits im Register entweder von der Eingabe oder von der Ausgabe der Rundenfunktion bestimmt. Im zweiten Entwurf hingegen hängt dieser Wert ab von der Eingabe, des Ergebnisses des Rho-Blocks, des lokalen Ergebnisses des Gamma-Blocks, sowie einem Bit im Ergebnis-Kanal. Welches Bit aus dem Ergebniskanal für ein Bit im Datenspeicher bestimmt ist, legt der Zustandsautomat und auch der Atom-Index fest. Diese Auswahlschaltung benötigt schon mehr Platz als die Reduktion der Datenmenge einspart. Analog ist auch das Lesen der Daten komplizierter geworden. Für den Gamma-Block und den Datenkanal müssen anhand des aktuellen Zustands und des Atom-Indexes aus allen 800 gespeicherten Bits immer ein paar auswählen.

4.3.4.3 Gamma-Funktion

Der Gamma-Block übernimmt praktisch den gesamten Rechenaufwand der modifizierten Rundenfunktion. Da die Berechnung auf zwei Atome aufgeteilt ist und auch nicht alle Slices in einem Atom gleichzeitig berechnet werden, ist der Platzbedarf für die γ-Berechnung

sehr stark geschrumpft, sodass das aktuelle Design nur etwa 70 LUTs benötigt. Eine weitere Optimierung der Berechnungsweise von γ ist daher auch in weiteren Iterationen nicht mehr nötig.

4.3.5 Optimierungsansätze

Die starke Steigerung der Speicherkomplexität ist das Hauptproblem des Entwurfs und weitere Verbesserungen müssen hier ansetzen, um den Beschleuniger aus die erforderliche Größe reduzieren zu können. Um die Speicherverwaltung vollständig aus dem Design zu entfernen, hatten wir die Nutzung der BRAM-Blöcke in den Überlegungen des ersten Entwurfs schon einmal in Erwägung gezogen und uns letztendlich dagegen entschieden, weil die Verwendung des BRAM das Festlegen auf eine feste Orientierung der gespeicherten Daten bedeutet. Damit der Gamma-Block einfach wiederverwendet werden kann, ist es sinnvoll, die aktuelle Aufteilung der Daten zu berücksichtigen. Die einzig sinnvolle Orientierung des Speichers ist somit Lane-orthogonal, Also Daten werden immer als ein ganzzahliges Vielfaches an Tiles gespeichert. So können immer ganze Tiles dem Gamma-Block bereitgestellt werden. Da die Rho-Funktion, eigentlich auf Lanes arbeitet, muss dann aber in ihrer Implementierung so angepasst werden, dass sie mit Tiles arbeiten kann.

4.3.5.1 Transformation der Rho-Funktion

Das Berechnen der Rho-Funktion auf Tile orientierten Daten kann mit Hilfe mehrerer Schieberegister realisiert werden. Dazu wird die Berechnung in zwei Stufen aufgeteilt. Die erste Stufe berechnet Links-Rotationen um maximal 32 Bits und die zweite Stufe berechnet Rechts-Rotationen um maximal 32 Bits. Mit der aktuellen Aufteilung der Daten müssen in beiden Atomen maximal sieben Links- bzw. Rechts-Rotationen gleichzeitig durchgeführt werden. Da die Berechnungen der Lanes untereinander unabhängig sind, können wir einfach eine Lösung für eine einzelne Lane sieben mal nebeneinander implementieren. Sollte der Platz dafür nicht ausreichen, kann die Berechnung auch in noch mehr als zwei Stufen eingeteilt werden, was zwar nochmal mehr Berechnugnszeit, dafür aber weniger Platz benötigt. Der Pseudocode 4.8 skizziert, wie eine Links-Rotation einer Lane um eine fixe Distanz $k <= 32$ mit Hilfe eines 32-Bit Schieberegisters durchgeführt werden kann. Die Rechts-Rotation funktioniert genau analog, nur werden die Ein- und Ausgabebits in der anderen Reihenfolge eingelesen / geschrieben; Mit diesem Vorgehen kann jede beliebige Rotation um 32 Bits mit nur einem Puffer realisiert werden, wobei jede unterschiedliche Distanz ein anderes Bit aus dem Puffer auswählt. Außerdem können auch immer mehrere Bits gleichzeitig gelesen und geschrieben werden, indem der Inhaöt des Schieberegisters um mehr als eine Stelle pro Schritt bewegt wird. Maximal brauchen wir von diesen Registern sieben Stück. Je mehr sich in den Beschleuniger integrieren lassen, desto schneller ist die Berechnung der ρ-Funktion. Da die zu lesenden und zu schreibenden Bits jedes Schritts nicht von der Rotations-Distanz abhängen, befinden für mehrere Puffer alle benötigten / berechneten Bits immer in den gleichen Tiles. Somit funktioniert die ganze Berechnung auf Slice-orientierten Daten.

4.3.5.2 BRAM als Datenspeicher

Mit dem neuen Ansatz für die Berechnung der ρ-Funktion kann auch der Datenspeicher in den BRAM verschoben werden, da das Problem der unterschiedlichen Ausrichtung der benötigten Eingabedaten behoben ist. Ein BRAM Block unterstütz dabei bis zu

```
1  Lane shift_left(Lane x, Distance k)
2  {
3    lane result = 0
4    Bit[32] buffer           // Unser Schieberegister, das nur
5                             // mit << 1 geschoben werden darf.
6
7    for i in 32 to 63 {      // Die oberen 32 Bits der Lane
8      buffer = buffer << 1   // werden nach und nach in den
9      buffer[0] = x[i]       // Puffer geschrieben
10   }
11
12   for i in 0 to 63 {       // Aus dem Puffer kann dann
13     r[i] = buffer[k]       // immer an der gleichen Stelle
14     buffer = buffer << 1   // ein Ergebnisbit ausgelesen werden
15     buffer[0] = x[i]
16   }
17   return result
18 }
```

Figure 4.8: Pseudocode für die Berechnung einer Links-Rotation mit Hilfe eines Schieberegisters

zwei Lese- und Schreibports. Das ist essenziell für die Berechnung der γ-Funkion, da durch die Pipeline in jedem Takt sowohl Daten für die eigenen Berechnungen, als auch für die Berechnungen des jeweils anderen Atoms gelesen werden müssen und auch die Ergebnisse beider Atome gleichzeitig festgehalten werden müssen. Da der Gamma-Block immer zwei Slices gleichzeitig verarbeitet, bietet sich dieses Format auch für die Breite der Speicherports an. So können von jedem Port immer zwei Tiles gleichzeitig adressiert werden. Da jeder Atom-Contrainer über insgesammt 3 BRAM Einheiten verfügt, können die Ergebnisse der γ-Funktion auch in einer anderen Einheit gespeichert werden als die Eingabedaten. Die ρ-Funktion kann tatsächlich quasi inplace in einem BRAM-Block berechnet werden, da der BRAM read-before-write-Zugriffe unterstützt. Wird ein Tile k gelesen und liegt das Ergebnis n Takte später vor, so kann es an der Stelle k + n gespeichert werden, nachdem im gleichen Takt der alte Slice mit dem Index k + n gelesen wurde. Das bedeutet, dass beide Ports gleichzeitig Daten für die Berechnung bereitstellen können, sodass immer 4 Tiles gleichzeitig in den Puffer eingelesen werden können. Auf diese Weise benötigt die Berechnung einer vollständigen Rotation somit theoretisch etwa 8 Takte zum Füllen des Puffers mit den Initialwerten und 16 Takte zum Lesen Schreiben aller Tiles zuzüglich zu ein paar Verzögerungstakten aufgrund des BRAMs.

4.3.5.3 Datenbus

Da sowohl der Gamma-Block als auch der neue Rho-Block in Zusammenarbeit mit dem BRAM wie oben beschrieben niemals auf mehr als zwei Ports gleichzeitig lesen oder schreiben, können die Datenleitungen für immer jeweils zwei Ports auf unterschiedlichen BRAM-Einheiten zusammengelegt werden. Wie das genau aussieht, ist im nächsten Kapitel genauer erläutert.

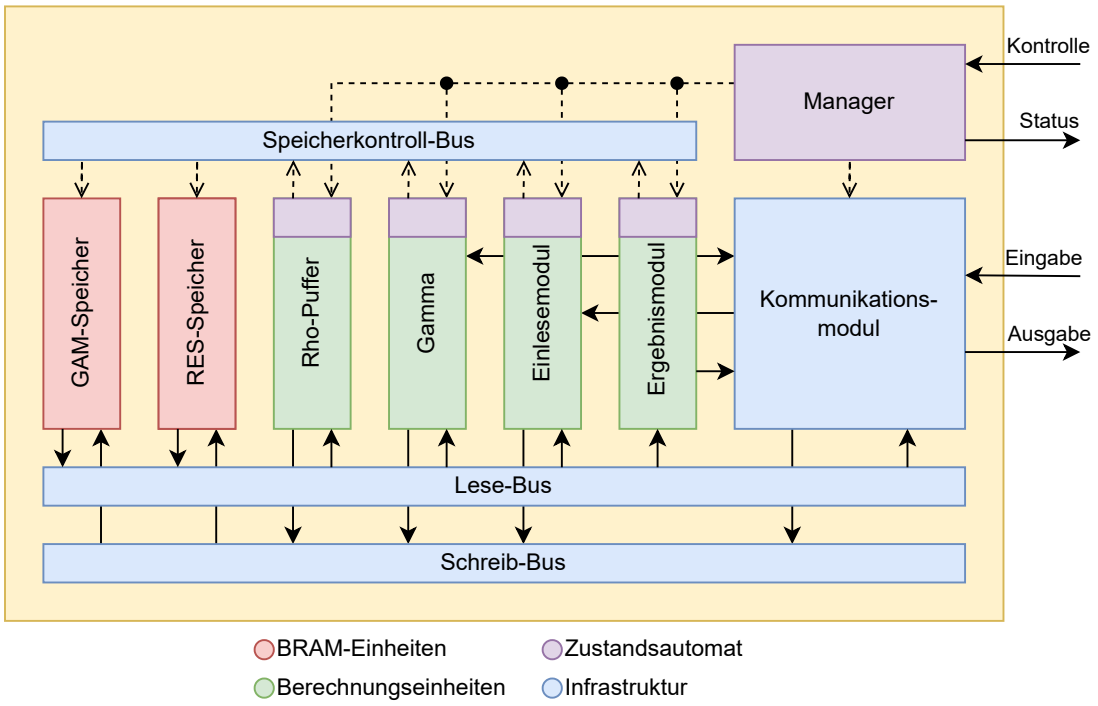

Figure 4.9: Aufbau der A-Atome

4.4 Finaler Entwurf

4.4.1 Entwurfsziele

Im finalen dritten Entwurf sollen nun diese Überlegungen realisiert werden, um den Beschleuniger endlich auf die erforderliche Größe zu bringen. Konkret soll der Datenspeicher in den BRAM verschoben werden. Dazu sollen zwei BRAM-Einheiten verwendet werden, auf die die Recheneinheiten über zwei Kanäle zugreifen können. Zusätzlick soll die Berechnung der ρ-Funktion über ein Schieberegister so realisiert werden, dass sie ihre Daten aus dem BRAM lesen und schreiben kann. Da der BRAM mit Tiles arbeitet, müssen außerdem das Einlesen der Datenblöcke, sowie das Ausgeben des Endergebnisses über eigenene Komponenten gesteuert werden, die den Datenaustausch zwischen der Kommunikationsschnittstelle und dem BRAM übernehmen. Durch den Einsatz des BRAM entsteht jedoch noch ein Problem: Die Eingabedaten liegen im Haupspeicher Lane-orientiert vor. Um sie in den BRAM schreiben zu können, müssen sie in Tiles umgeordnet werden. Dieser Prozess muss entweder vor der Berechnung mühsam in Software ausgeführt werden, oder die Konvertierung findet im Beschleuniger selber statt. Die Konvertierung in Software würde viel extra Rechenzeit in Anspruch nehmen. Deshalb werden wir sie im Beschleuniger durchführen. Da aber noch nicht klar ist, wie viel Platz in den Atomen nach den Änderungen vorhanden sein wird, wird die Konvertierung in zwei weiteren Atomen parallel zur Berechnung der KECCAK-p-Funktion durchgeführt.

4.4.2 Aufbau

Der Beschleuniger besteht diesmal aus insgesamt vier Atomen. Die beiden Atome *A0* und *A1* sind wie ihre Vorgänger symmetrisch und sind für die eigentliche Berechnung der

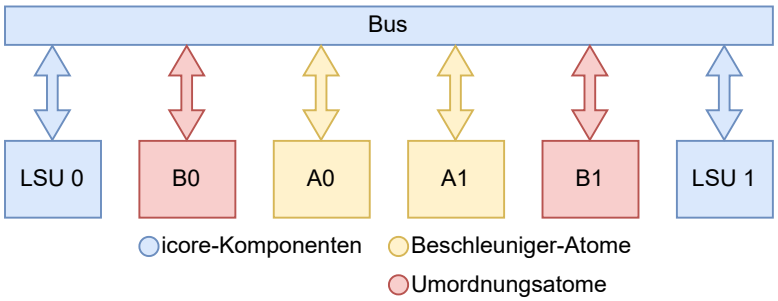

Figure 4.10: Integration der Atome im finalen Beschleuniger
Unbenutzte Komponenten und Kontrollsignale wurden weggelassen

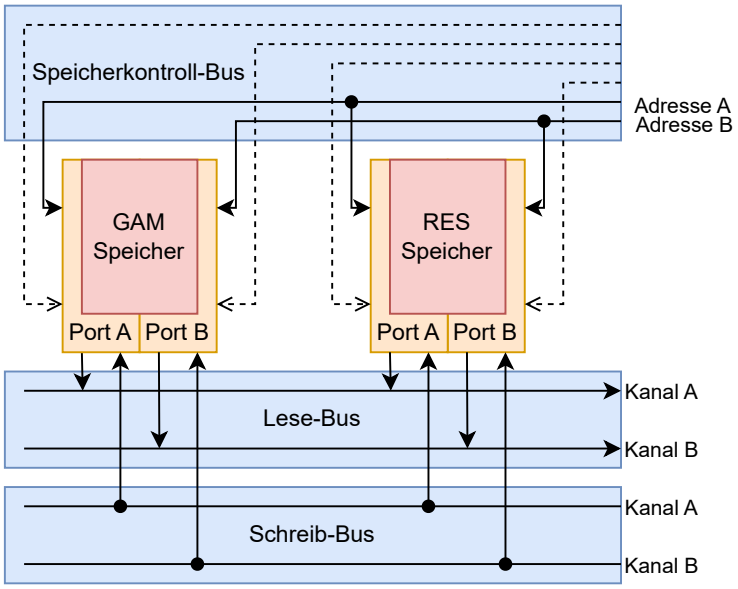

Figure 4.11: Speicheranbindung

Permutationsfunktion zuständig (siehe Abb. 4.9). Die Atome *B0* und *B1* lesen während der Berechnung schon den nächsten Datenblock aus dem externen Speicher ein und konvertieren ihn in Tiles, die dann von den A-Atomen entgegen genommen werden. Wie auch im vorherigen Entwurf wird den Atomen ihre Rolle über einen Atom-Index zugewiesen. In Abbildung 4.10 ist dargestellt, wie die Atome in der Fabric des icore angeordnet sind. Die Funktionalitäten der A-Atome "Daten lesen", "Ergebis schreiben", "ρ berechnen" und "γ berechnen" haben ihre eigenen Berechnungseinheiten erhalten. Verbunden sind alle Komponenten über eine Infrastruktur aus Bussen und einem Kommunikationsmodul für den Datenaustausch mit anderen Atomen. Verwaltet wird die gesamte Berechnung wieder von einem Zustandsautomaten.

4.4.2.1 BRAM

Die beiden verwendeten BRAM-Bänke GAM (für Gamma) und RES (für Result) besitzen jeweils zwei Lese-Schreib-Ports, mit denen jeweils zwei Tiles gelesen und auch gleichzeitig geschrieben werden kann (read before write). Je ein Port ist dabei an den A-Kanal und der Andere an den B-Kanal angeschlossen, sodass insgesamt bis zu vier Tiles gleichzeitig gelesen und geschrieben werden können.

4.4.2.2 Speicherbus

Der Speicherbus besteht aus drei Segmenten (Abb. 4.11). Auf dem Lese-Bus werden Daten aus dem BRAM ausgelesen und den anderen Modulen zur Verfügung gestellt. Er besteht aus zwei Kanälen, die beide zwei Tiles breit sind. Auf dem Schreib-Bus werden Daten von den Berechnungsmodulen und dem Kommunikationsmodul gesammelt und an den BRAM weitergegeben. Er ist auch zwei Tiles breit. Über den Speicherkontroll-Bus werden alle Steuersignale für den BRAM gesammelt. Er besteht aus zwei 7 Bit breiten Address-Vektoren für die beiden Datenkanäle, sowie einem Read-Enable-Signal und einem Write-Enable-Signal für jeden der insgesamt vier Ports.

4.4.2.3 Zustandsautomat

Der Zustandsautomat besteht nicht mehr aus einer Einheit, sondern besteht nun aus einer zentralen Kontrolle, dem Manager, sowie spezialisierten Kontrolleinheiten innerhalb der Berechnungseinheiten, angedeutet durch die kleinen lila Blöcke oberhalb der Berechnungseinheiten. Der Manager steuert dabei nur noch den Betriebsmodus des Kommunikationsmoduls und stößt die Abarbeitung in den Berechnungseinheiten an. Die Zustandskontrolle innerhalb einer Berechnungseinheit bekommt bei Aktivierung die Kontrolle über den Speicherkontroll-Bus und generiert die Steuersignale für die Berechnungseinheit und den Speicher. Ist die Abarbeitung einer Berechnungseinheit abgeschlossen, wird die Kontrolle über den Speicher und die weitere Abarbeitung wieder an den Manager übergeben.

4.4.2.4 Kommunikationsmodul

Das Kommunkationsmodul dient zum Datenaustausch zwischen den Atomen sowie zur Kommunikation mit dem externen Speicher, der die Eingabedaten bereitstellt und das Ergebnis entgegennimmt. Für jede Berechnungseinheit gibt es einen Betriebsmodus, der festlegt, welche Daten von der Berechnungseinheit und dem Speicherbus ausgegeben

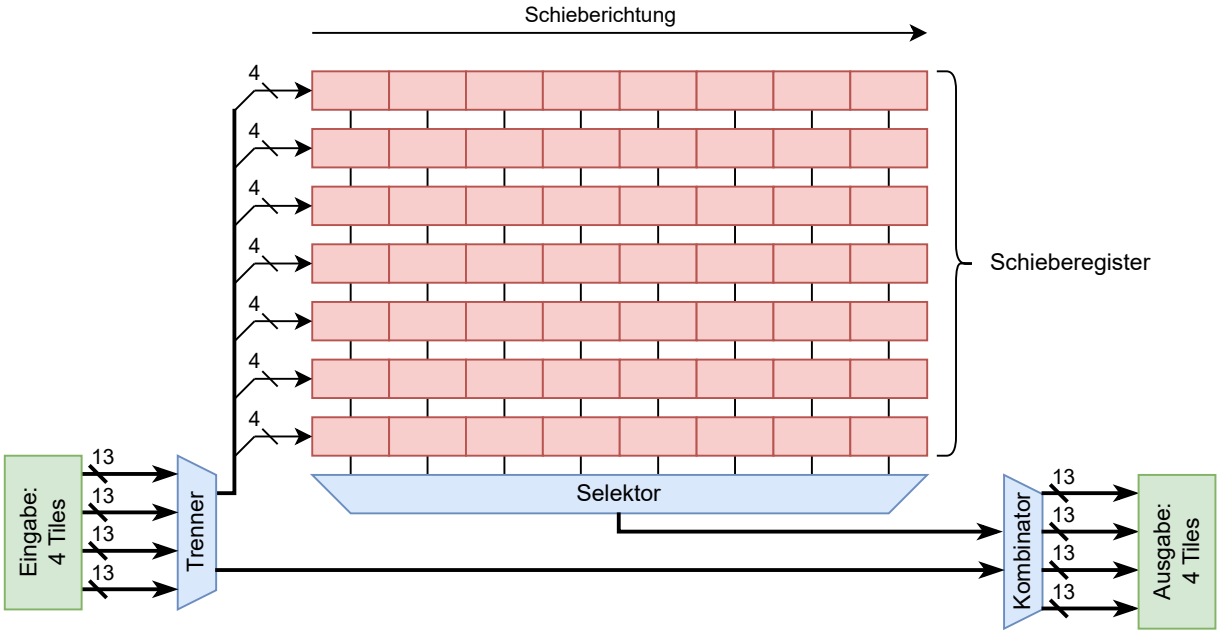

Figure 4.12: Aufbau des Rho-Puffers

werden und welche empfangenen Daten an den Speicherbus und die Berechnungseinheit weitergegeben werden. Dieser Betriebsmodus wird vom Manager bestimmt.

4.4.2.5 Gamma

Das Gamma-Modul berechnet analog zum Modul aus dem vorherigen Entwurf immer zwei Slices und gibt die entsprechenden Tiles an den Speicher und über das Kommunikationsmodul an das andere Atom aus. Bis auf die Einführung des Kontrollblocks und einer leichten Anpassung der Schnittstelle ist es identisch zum vorherigen Design. Es nimmt die Eingabedaten vom RES-Speicher und dem Kommunikationsmodul entgegen und schreibt das Ergebnis der Berechnung in den GAM-Speicher, bzw. übermittelt es die restlichen Tiles wieder über die Kommunikationseinheit an das andere Atom, wo sie gespeichert werden.

4.4.2.6 Rho-Puffer

Der Rho-Puffer besteht aus sieben 32-Bit-Schieberegistern. Diese berechnen, wie in 4.3.5.1 beschrieben, interativ die Rotationen der ρ-Funktion. Der Aufbau des Rho-Moduls ist in Abbildung 4.12 skizziert. Die Berechnung erfolgt wie folgt: Im ersten Schritt werden die Tiles aus dem GAM-Speicher gelesen und auf den betrffenden Lanes wird eine Linksrotation durchgeführt, während die anderen Lanes unverändert bleiben. Das Ergebnis wird im wieder im GAM-Speicher gespeichert. Im zweiten Schritt werden die Tiles aus dem GAM-Speicher erneut ausgelesen und diesmal wird auf den übrigen Lanes eine Rechtsrotation durchgeführt. Das finale Ergebnis der ρ-Berechnung wird dann im RES-Speicher wieder abgespeichert. Da es zur Berechnung keinerlei Kommunikation benötigt, ist dieses Modul auch als einziges nicht an die Kommunikationseinheit angeschlossen.

4.4.2.7 Einlesemodul

Um das Ergebnis einer Permutation mit einem neuen Datenblock gemäß der Schwammkonstruktion zu kombinieren, müssen die eingelesenen Daten mit den Daten im Ergebnisspeicher mit einem XOR kombiniert werden. Das Einlesemodul macht genau das und schreibt die kombinierten Daten wieder zurück in den Ergebnisspeicher, sodass erneut die Permutation berechnet werden kann. Während die neuen Daten eingelesen werden, kann jedoch die eigentliche Berechnung nicht durchgeführt werden. Daher ist es wichtig, dass dieser Schritt so schnell wie möglich abläuft. Dazu werden die Daten den A-Atomen beim Einlesen direkt als Tiles bereitgestellt, sodass das Einlesemodul nicht erst die Lanes in Tiles konvertieren muss. Diese Aufgabe übernehmen, wie bereits beschrieben, die B-Atome.

4.4.2.8 Ergebnismodul

Zur Ausgabe des 256-Bit Hashes wird das Ergebnismodul verwendet. Es besteht aus einem in Flip-Flops implementierten Puffer, der mit den Daten des Ergebnisspeichers gefüllt wird und die Daten so zurück in Lanes konvertiert. So kann das Ergebnis direkt im richtigen Format ausgegeben werden und im externen Speicher übernommen werden.

4.4.3 Berechnung der Permutationsfunktion

Da die Daten im BRAM nur als Tiles gespeichert werden und die Eingabedaten aber als Lanes vorliegen, müssen sie erst konvertiert werden. Diese Konvertierung findet während der vorherigen Berechnung in den B-Atomen statt. Dabei lesen die B-Atome mehrfach den gesammten Datenblock Lane für Lane ein und puffern immer einen Teil aller Lanes, bis sie sie als vollständige Tiles speichern können. Dass dabei jede Lane mehrfach eingelesen werden muss, hat keinen Einfluss auf die Ausführunszeit des Beschleunigers, da die Berechnungszeit der modifizierten Rundenfunktion größer ist, als die Berechnungszeit der Konvertierer und beide Systeme vollständig parallel arbeiten können. Nachdem ein Block konvertiert wurde und die A-Atome bereit sind, werden die Tiles der Reihe nach über das Kommunikationsmodul von den B-Atomen zum Einlesemodul der A-Atome übertragen und mit den Daten aus dem RES-Block über ein XOR kombiniert und anschließend wieder in den RES-Block übernommen. Für die Gamma-Berechnung werden die Slices über den Lesebus aus dem RES-Block gelesen und an die Recheneinheit sowie an die Kommunikationseiheit übergeben. Die Ergebnisse werden anschließend vom Kommunikationsmodul und der Recheneinheit über den Schreib-Bus in den GAM-Block geschrieben. Anschließend wird die Links-Rotation mit Hilfe des Rho-Puffers durchgeführt. Die Daten werden aus dem GAM-Block über den Datenbus an den Puffer übertragen. Zuerst wird der Puffer mit den Initialdaten gefüllt, dann werden, wie in 4.3.5.1 erläutert, die Tiles nach und nach in den Puffer eingeschoben und die Erbebnis-Tiles werden über den Schreib-Bus wieder in den GAM-Block übernommen. Die Rechts-Rotation wird genau so durchgeführt, nur werden dieses Mal die Ergebnisse in den RES-Block geschrieben anstatt in den GAM-Block. Hier wird auch ein Vorteil des Bus-Designs deutlich: Da beide Blöcke ihre Daten über den Bus empfangen, braucht für die Rechts-Rotation nur das Write-Enable-Signal verändert werden und die Ergebnisse müssen nicht auf eine andere Eingabe umgeleitet werden. Die Rho und Gamma Funktionen werden so oft abwechselnd berechnet, bis der Datenblock gemäß der Definition der KECCAK-p-Funktion verarbeitet wurde. Danach kann über den Kontrollvektor von außen bestimmt werden, ob entweder der nächste Block eingelesen werden, oder das Ergenis ausgegeben werden soll. Für die Ausgabe des Ergebnisses werden

alle 64 Tiles im Atom A0 vom Ergebnismodul ausgelesen und die ersten 4 Bits im Puffer des Ausgabemoduls gespeichert. Als letztes wird das Ergebnis als Lanes ausgegeben und im externen Speicher als Endergebnis gespeichert.

4.4.4 Bewertung

Die A-Atome erfüllen mit einer Größe von 1205 LUTs die Anforderungen des icore, was das Hauptziel dieses Entwurfs war. Leider wurde dafür sehr viel Berechnungsgeschwindigkeit geopfert. Mit weiteren Optimierungen lässt sich der noch verfügbare Platz bestimmt sinnvoll nutzen, um die Berechnung wieder etwas zu beschleunigen. Wie solche Ansätze aussehen könnten, werden wir uns später ansehen. Außerdem besteht der Beschleuniger jetzt aus vier Atomen. Die B-Atome sind dabei sehr unspektakulr, weshalb wir sie uns auch nicht weiter angeschaut haben. Sie sind mehr eine Bequemlichkeitslösung, um sich nicht mit der Integration des Konvertierungsprozesses in die A-Atome beschäftigen zu müssen, als dass sie sonderlich hilfreich bei der Berechnung sind. Weitere Optimierungsansätze sollten sich daher vor allem darauf fokussieren, diese B-Atome entweder loszuwerden, oder sinnvoll für die Berechnung zu nutzen. Was jedoch sehr schön funktioniert, ist die Berechnung der modifizierten Rundenfunktion in den einzelnen Modulen. Ohne die Berechnung der ρ-Funktion über die Schieberegister wäre dieses Verfahren nicht möglich gewesen. Die Frage, wie gut der Beschleuniger die Berechnung von KECCAK-p und damit auch SHA-3 denn jetzt tatsächlich beschleunigt, werden wir uns im nächsten Kapitel anschauen.

Chapter 5

Ergebnisse

Die drei vorgestellten Designs für den Beschleuniger bauen aufeinander auf mit dem Ziel immer mehr Leistung für weniger Platzbedarf einzutauschen. In diesem Kapitel werden alle Designs miteinander verglichen was sowohl ihre Größe angeht, als auch ihr Leistungspotential. Das konkrete Ziel dieser Arbeit bestand darin einen Beschleuniger zu entwerfen, der speziell die Anforderungen der icore-Architektur erfüllt. Daher wird in diesem Kapitel hauptsächlich der icore als Vergleich herangezogen für die Ausführung von Software-Implementierungen. Außerdem werden wir uns bei der Untersuchung der Ausführungszeiten auf die KECCAK-p-Funktion beschränken, da diese den Hauptaufwand darstellt und gerade bei größeren Datenmengen Schritte wie die Initialisierung und das Auslesen des Hashs am Ende vernachlässigbar sind. Bei Vergleichen zwischen Beschleunigern mit Software-Berechnungen dient immer die Implementierung aus Anhang 1 als Bezug.

5.1 Syntheseergebnisse

In Tabelle 5.1 sind die Größen der verschiedenen Beschleunigeriterationen nocheinmal zusammengefasst. Während die ersten beiden Iterationen alle Funktionalitäten in die A-Atomen integrieren, die die tatsächliche Berechnung der KECCAK-p-Funktion durchführen, benötigt die dritte Iteration noch zwei zusätzliche B-Atome, in denen die Eingabedaten so umgeordnet werden, dass sie in den BRAM der A-Atome geschrieben werden können. Da in dem dritten Entwurf die Obergrenze von 1600 LUTs für die A-Atome noch nicht erreicht ist, kann diese Funktionalität auch in die Beschleuniger intergriert werden. Je weniger Atome für den Beschleuniger benötigt werden, desto schneller kann der Beschleuniger geladen werden. Performance-technisch ist es jedoch sinnvoller diese Funktionen zu trennen. Da die A-Atome bei der Berechnung sowohl den BRAM als auch ihre Kommu-

	A-Atome	B-Atome	LUTs	FFs	kritische Pfadlänge	BRAM Bänke
1. Entwurf	1	0	3314	1845	?	0
2. Entwurf	2	0	4643	1865	?	0
3. Entwurf	2	2	1205	628	?	2

Table 5.1: Interessante Größen der synthetisierten Designs
Sämtliche Größen beziehen sich auf die A-Atome

	Berechnungsdauer	Einlesedauer	Gesamtdauer
1. Entwurf	24 Takte	34 Takte	58 Takte
2. Entwurf	509 Takte	34 Takte	543 Takte
3. Entwurf	1968 Takte	36 Takte	2004 Takte

Table 5.2: Ausführungszeiten der verschiedenen Beschleuniger-Entwürfe
Alle Zeitangaben beziehen sich auf die Abarbeitung eines Datenblocks. Initialisierung und Auslesen des Hashs sind nicht einberechnet.

nikationsschnittstelle voll auslasten, kann die Aufgabe der B-Atome nicht gleichzeitig in den A-Atomen durchgeführt werden. Daher müsste für die Zusammenführung der Atome die Einlesephase deutlich verlängert werden. Für das Verarbeiten großer Datenmengen ist es daher sinnvoll die leicht erhöhte Konfigurationszeit des Beschleunigers inkauf zu nehmen.

5.2 Ausführungszeit

Zur bestimmung der Ausführungszeit des Beschleunigers verwenden wir hier die Simulation. Das hat den Vorteil, dass auch die vorherigen Entwürfe mit verglichen werden können, die zu groß für den icore sind. Da alle Entwürfe statisch sind in ihrer Ausführungszeit, die Abarbeitung eines Blocks also immer die gleiche Zeit benötigt, liefert diese Herangehensweise genauere Werte als die Zeitmessung es auf dem icore tun würde. Da die kritischen Pfade aller Entwürfe die Operation auf dem 50MHz Takt des icores zulassen, können die Taktzahlen direkt miteinander verglichen werden, siehe Tabelle 5.2. Die Einlesedauer der Iterationen unterscheidet sich dabei fast gar nicht. In den ersten beiden Entwürfen liegt jede der 17 Lanes aus den Eingabedaten nacheinander jeweils einen Kontrollschritt, also zwei Takte, an beiden Atomen an und die Atome wählen jeweils die Lanes aus, die sie benötigen (im ersten Entwurf liest das einzige Atom alle Lanes ein). Im dritten Entwurf hingegen lesen beide A-Atome gleichzeitig verschiedene Slices, die ihnen von den B-Atomen bereitgestellt werden. Da in jedem Kontrollschritt jedoch nur jeweils vier Tiles zu je 13 Bits übertragen werden anstatt einer ganzen Lane, werden 16 statt der vorherigen 13 Kontrollschritte benötigt. Diese Effekte heben sich gegenseitig auf. Eine weitere Optimierung der Einlesezeiten ist auch nicht weiter nötig. Im ersten Entwurf ist sie nicht möglich, da die Kommunikationsschnittstelle vollständig ausgelastet ist und in den beiden letzten Entwürfen beträgt das Einlesen einen zu kleinen Teil an der gesamten Berechnung.

Die tatsächliche Berechnung hingegen nimmt mit jedem Entwurf deutlich mehr Zeit in Anspruch. Das ist alleine der Sequenzialisierung geschuldet. Statt wie vorher die gesamte Rundenfunktion in einem Takt abzuarbeiten, werden im zweiten Entwurf Rho in einem Schritt und Gamma/Theta in 16 Schritten für jeweils 2 Slices pro Atom berechnet. Hinzu kommen noch ein paar Takte durch die Verzögerung zwischen den Atomen, sodass die Berechnung der erweiterten Rundenfunktion im zweiten Entwurf insgesamt 21 Takte benötigt. Der dritte Entwurf verwendet die gleiche Vorgehensweise für die Berechnung von Gamma/Theta, braucht allerdings ein drei Takte länger durch die Verzögerung des BRAM und berechnet außerdem auch Rho sequenziell in 57 Takten. In Tabelle 5.2 sind die Ausführungszeiten für die verschiedenen Teilfunktionen im dritten Entwurf nocheinmal zusammengefasst. Die KECCAK-p-Funktion, für die wir uns eigentlich bei der Berechnung von SHA-3 interessieren, setzt sich zusammen aus dem Einlesen des Datenblocks, einer

Funktion	Zeit (in Takten)	Iterationen in KECCAK-p	Anteil an KECCAK-p
Initialisierung	18	-	-
Einlesen	36	1	1,8%
Theta	24	1	1,2%
Gamma	24	24	28,74%
Rho	57	24	68,26%
KECCAK-p	2004	1	100%
Auslesen	27	-	-

Table 5.3: Ausführungszeiten des finalen Beschleunigers

Berechnung von Theta, sowie 24 Berechnungen von Rho und Gamma.

5.3 Weitere Optimierungsansätze

Der dritte Entwurf des Beschleunigers erfüllt zwar alle geforderten Voraussetzungen, trotzdem lassen sich noch weitere Verbesserungen vornehmen, um die Leistungsfähigkeit zu erhöhen. Daher wollen wir uns hier noch ein paar dieser möglichen Optimierungen anschauen.

5.3.1 Auslagerung der Ergebniskonvertierung

Das Ergebnismodul nimmt unnötig viel Platz im Atom ein und ist eigentlich nur deshalb in den A-Atomen enthalten, weil der Platz nicht weiter benötigt wird. Es lässt sich aber auch in die B-Atome verschieben, die auch die Konvertierung für die Eingabe übernehmen. So kann noch ein wenig mehr Platz für andere Optimierungen geschaffen werden, die die Berechnungsgeschwindigkeit weiter verbessern.

5.3.2 Reduktion auf ein A-Atom

Die Berechnung wurde ursprünglich auf zwei Atome aufgeteilt, damit die Datenmenge reduziert werden kann, die ein Atom halten muss. Durch die Verwendung des BRAM für den Datenspeicher fällt dieser Aufwand weg, da der BRAM mehr als genug Platz bereitstellt. Ein Beschleuniger, der nur ein A-Atom verwendet, bietet vor allem den Vorteil, dass für die Konvertierung der Eingabedaten auch nur ein B-Atom benötigt wird. Der gesamte Beschleuniger benötigt also nur zwei der fünf Atome. Abhängig vom Anwendungsfall können so auch mehrere kleine Beschleuniger nebeneinander existieren, ohne dass die Atom-Container zwischendurch rekonfigueriert werden müssen. Außerdem ist die Berechnung selbst nicht mehr durch die Kommunikationsschnittstelle zwischen den Atomen beschränkt, wodurch weitere Optimierungen möglich werden. Die Reduktion auf ein A-Atom alleine bringt jedoch keine direkte Leistungsverbesserung. Im Gegenteil, da aktuell beide A-Atome parallel sowohl Rho als auch Gamma berechnen, würde die Reduktion alleine die Berechnungszeit etwa verdoppeln. Auch das Einlesen der Datenblöcke dauert länger, da der gesamte Block in ein einziges Atom übertragen werden muss, was wiederum durch die Kommunikationsschnittstelle beschränkt ist.

5.3.3 Erweiterung der BRAM-Schnittstelle

Aktuell können an einem Port des BRAM immer jeweils zwei Tiles gleichzeitig gelesen und geschrieben werden. Die Erweiterung der Speicherschnittstelle auf zum Beispiel 4 Tiles erlaubt es, eine größere Menge an Tiles gleichzeitig für die Berechnung von Rho bereit zu stellen. Die Rho-Funktion könnte somit doppelt so schnell berechnet werden. Da die Berechnung von Rho mit 57 Takten etwa 70% der 81 Takte für die Berechnung der erweiterten Rundenfunktion beansprucht (siehe 5.2), ist nochmal mit einer weiteren Beschleunigung von $S_{WideBRAM} = \frac{81\ Takte}{81\ Takte - (57\ Takte/2)} \approx 1,54$ zu rechnen. Die restlichen 30% der Berechnungszeit für die erweiterte Rundenfunktion werden für die Gamma-Funktion verwendet. Diese kann durch die Erweiterung der Speicherschnittstelle nicht weiter beschleunigt werden, da sie die Kommunikationsschnittstelle zwischen den Atomen bereits den limitierende Faktor darstellt. Unklar ist noch, ob der vorhandene Plat für diese Optimierung ausreicht.

5.3.4 Erweiterung des Rho-Puffers

Möchte man wie oben beschrieben den Beschleuniger auf ein Atom reduzieren, kann auch der Rho-Puffer, in dem die Bit-Rotationen blockweise auf mehreren Lanes gleichzeitig durchgeführt werden, erweitert werden. Im aktuellen Entwurf ist das nicht mehr möglich, da genau eine Links- und eine Rechtsrotation auf etwa jeweils der Hälfte der in den Atomen gespeicherten Lanes durchgeführt werden. Links- und Rechtsrotationen lassen sich nicht zusammenlegen, was damit zusammenhängt in welche Richtung die Lanes aus dem Speicher gelesen und geschrieben werden. Findet jedoch die gesamte Berechnung nur in einem Atom statt, sind für die vollständige Abarbeitung zwei Links- und zwei Rechts-Rotationen notwendig. Erweitert man den Puffer, sodass jeweils die beiden Links- und die beiden Rechts-Rotationen zusammengelegt werden können, kann dadurch verhindert werden, dass die Berechnung der Rho-Funktion länger dauert.

5.3.5 Auslagerung der Rho-Berechnung

Es ist zu erwarten, dass die Erweiterung des Rho-Puffers viel zusätzlichen Platz in Anspruch nehmen wird. Sollte das Atom dadurch die maximale Größe überschreiten, kann eventuell ein Teil der Berechnung von Rho in ein anderes Atom wieder ausgelagert werden. Aktuell werden in jedem Takt jeweils 4 Bits aus maximal 7 Lanes, also maximal 28 Bit in den Puffer geschrieben und gleichzeitig gelesen. Dieser Datenverkehr ist über die Kommunikationsschnittstelle durchaus gleichzeitig in beide Richtungen möglich, er kann sogar noch verdoppelt werden, wenn man auch die BRAM-Schnittstelle wie oben beschrieben erweitert. Jenachdem wie groß diese Konstruktion wird, kann sie auch in das B-Atom integriert werden, sodass der Beschleuniger trotzdem nur aus zwei Atomen besteht. Das B-Atom würde dann während der Berechnung von Rho mit dem A-Atom zusammenarbeiten und zwischendurch den nächsten Datenblock einlesen.

5.3.6 Erhöhung der Berechnungsfrequenz

Der Beschleuniger selbst erlaubt eine Taktfrequenz von 200MHz, was in diesem Fall das Maximum für das verwendete FPGA ist. Der icore selbst läuft jedoch nur mit einer Frequenz von 50MHz. Da der Beschleuniger seinen Takt mit dem icore teilt, damit die

Messung	1	2	3	4	5	Durchschnitt ($\tilde{t}$)
Zeit (s)	14,505455	14,505454	14,505463	14,505464	14,505452	14,5054576

Table 5.4: Ausführungszeit der Software-Berechnung

Synchronisierung am einfachsten ist, läuft er jedoch sehr viel langsamer, als er eigentlich könnte. Verwendet man einen Takt von 200MHz für die Berechnung der Rho-Funktion, die vollständig in den Atom erfolgt und keinerlei Kommunikation benötigt, erhält man eine Beschleunigung von $S_{200MHz} = \frac{81\ Takte/50MHz}{(81\ Takte - 57\ Takte)/50MHz + 57\ Takte/200MHz} \approx 2,12$. Für die Herkunft der genauen Taktzahlen siehe Tabelle 5.2. Auch hier kann wieder nur die Rho-Funktion beschleunigt werden, da die Kommunikation zwischen den Atomen durch den i-core wieder durch den 50MHz-Takt beschränkt ist.

5.4 Gemessene Beschleunigung

Wie bereits erwähnt, sind die entworfenen Beschleuniger strikt deterministisch in der Hinsicht, dass die Dauer der Berechnung immer die selbe Zeit benötigt. Daher kann aus den vom Beschleuniger benötigten Takten sowie der Taktfrequenz direkt die Ausführungszeit des Beschleunigers bestimmt werden. Bei der Software-Berechnung ist das nicht so einfach möglich, da zum Beispiel Speicherzugriffe aufgrund der Speicherstruktur unterschiedliche Ausführungszeiten benötigen. So kann die Anzahl der auftretenden Cache Misses einen großen Einfluss auf die Berechnungsdauer haben. Auch ist nicht klar, ob wirklich jede Operation des Befehlssatzes in genau einem Takt ausgeführt werden kann. Durch Datenabhängigkeiten kann es notwendig sein, dass die weitere Abarbeitung auf Ergebnisse vorheriger Instruktionen, die sich noch in der Pipeline befinden, warten muss. Die beste Möglichkeit zur Bestimmung der Laufzeit der Software-Berechnung ist daher die tatsächliche Zeitmessung. Diese Messung ist natürlich etwas ungenau, weshalb in 5 Durchläufen jeweils 10.000 Iterationen der KECCAK-p-Funktion durchgeführt werden, woraus dann die mittlere Berechnungsdauer bestimmt wird. Die Ergebnisse dieser Messungen sowie der berechnete Durchschnitt $\tilde{t}$ sind in Tabelle 5.4 aufgeführt. Aus dieser Messung ergibt sich für eine einzige Berechnung der KECCAK-p-Funktion eine Zeit von

$$t_{SW} = \tilde{t}/10000 = 14,5054576s/10000 = 1450,54576\mu s$$

Die Ausführungszeit der KECCAK-p-Funktion auf dem finalen Beschleuniger und der daraus resultierende Speedup kann mit Hilfe der Tabelle 5.2 aus dem 50MHz-Takt des icore wie folgt exakt berechnet werden:

$$t_{ACC} = \frac{2004\ Takte}{50 * 10^6 Hz} = 0,00004008s = 40,08\mu s$$
$$S_{ACC} = \frac{t_{SW}}{t_{ACC}} \frac{1450,54576\mu s}{40,08\mu s} \approx \mathbf{36,2}$$

5.5 Theoretische Beschleunigung

Neben der tatsächlichen Ausführungszeit auf dem icore wollen wir uns noch eine andere Metrik anschauen, um den Beschleuniger zu bewerten. Dazu betrachten wir die Anzahl

	XODER	UND	ODER	NEG	Kopie	ROL	Shift	Gesamt
Theta	50	0	0	0	0	5	0	55
Rho-Pi	0	0	24	0	49	0	48	121
Chi	25	25	0	25	25	0	0	100
Iota	1	0	0	0	0	0	0	1
Rnd	76	25	24	25	74	5	48	277
KECCAK-p	1824	600	576	600	1776	120	1152	6648

Table 5.5: Instruktionen der Software-Funktionen

	32-Bit-Speedup	64-Bit-Speedup
1. Entwurf	229,24	114,62
2. Entwurf	24,49	12,25
3. Entwurf	**6,63**	3,32

Table 5.6: Theoretischer Speedup der verschiedenen Entwürfe

an elementaren Operationen, die die Software auf einem virtuellen System benötigt, um die KECCAK-p-Funktion zu berechnen. Als elementare Operationen zählen dabei Instruktionen, von denen man erwarten kann, dass sie von jedem modernen Prozessor in jeweils einem Takt berechnet werden können. Als solche Instruktionen zählen in diesem Fall XODER, UND, ODER, NEGATION (NEG), das Kopieren, eine Rotation um ein Bit nach links (ROL) sowie ein Bitshift variabler Länge. Weiter unterschieden werden muss allerdings zwischen 32-Bit- und 64-Bit-Operationen. In der Tabelle 5.5 ist die Anzahl der jeweils benötigten 64-Bit-Operationen aufgeführt, die für die einzelnen Teilfunktionen von KECCAK-p benötigt werden, die sich aus insgesamt 24 Runden zusammensetzt. Die Anzahl an benötigten 32-Bit-Operationen ist dann doppelt so groß. Eine 64-Bit-Rotation variabler Länge, wie sie von Rho verwendet wird, ist dabei, wie auch in der Implementierung in Anhang 1 beschrieben, aus zwei Shifts und einem ODER zusammengesetzt. Außerdem werden sämtliche Operationen, die nicht direkt zur die für die Ergebnisberechnung dienen, wie zum Beispiel Zählvariablen oder Indexberechnungen für Felder, ignoriert. Da wir gefordert haben, dass jede dieser Operationen in jeweils einem Takt berechnet können werden soll, ergibt sich aus der Anzahl der Operatonen auch gleich die Anzahl der benötigten Takte. Auf diese Weise können wir wieder einen Speedup für den Beschleuniger berechnen, indem wir die benötigten Operationen der Software mit den benötigten Takten des Beschleunigers vergleichen. Hier können auch alle Entwürfe einbezogen werden, da wir für unsere virtuelle Umgebung keine Begrenzungen für die Größe der Beschleuniger vorgegeben haben und somit der Vergleich auch sinnvoll ist. In Tabelle 5.5 ist sowohl der theoretische Speedup sowohl bezüglich 32-Bit- als auch 64-Bit-Operationen aufgeführt. Dieser theoretische Speedup beschreibt den Faktor, wie viel der tatsächlichen Berechnung der Beschleuniger in jedem Schritt mehr erledigt, als die Software es im Optimalfall tut. Vergleicht man den theoretischen Speedup mit dem gemessenen Speedup, so stellt man fest, dass der gemessene Speedup des finalen Entwurfs mit 36,2 um einen Faktor 5,46 größer ist als der theoretische Speedup von 6,63 (wir verwenden hier den 32-Bit-Speedup, da der icore eine 32-Bit-Plattform ist). Dieser Faktor beschreibt die gewonnene Geschwindigkeit, die dadurch entsteht, dass der Beschleuniger keine Kontrollstrukturen wie Schleifen, Speicherzugriffszeiten, oder mehrere Takte für eine Instruktion benötigt.

Chapter 6

Fazit

Chapter 7

Ähnliche Arbeiten

Chapter 8

Glossar

Chapter 9

Symbolverzeichnis

Chapter 10

Anhang 1: Softwareimplementierung

Chapter 11

Template Stuff that is still here

11.1 Some Template Comments

- It is recommended to use one sentence per line of the latex source code. That is a good compromise between (i) 'diffs' when using repositories, and (ii) forward-/backward search between latex source code and pdf output.
- Note that you can have multiple refs in the same \cref block (e.g., **??**, Sections 11.1 to 11.3, and Figure 11.2), but there must not be spaces after the commas.
- Note that you should use \Cref (upper-case C) at the beginning of a sentence and \cref (lower-case c) in the middle of a sentence. They are defined differently, such that the upper-case C version does not use abbreviations (which is recommended for the beginning of a sentence), e.g., Eq. (11.2) vs. Equation (11.2).
- You can use the outline environment to collect itemized points before actually writing your text.
  - It helps structuring your ideas by simplifying indentation of items
    - Like here.

11.2 Problem Statement

Based on a partitioning $P \subset 2^V$, i.e., $p_i, p_j \in P, p_i \neq p_j \Rightarrow p_i \cap p_j = \emptyset, \bigcup_P = V$, an equivalence relation $\sim_P$ as well as the partition graph G_P are defined as follows:

$$\sim_P = \{(v_1, v_2) \in V | \exists p \in P : v_1 \in p \wedge v_2 \in p\} \tag{11.1}$$

$$G_P = (V_P, E_P) = (V/\sim_P, \{([v_1]_{\sim_P}, [v_2]_{\sim_P}) | \, (v_1, v_2) \in E\}) \tag{11.2}$$

11.3 Results

Following is the discussion of obtained results.

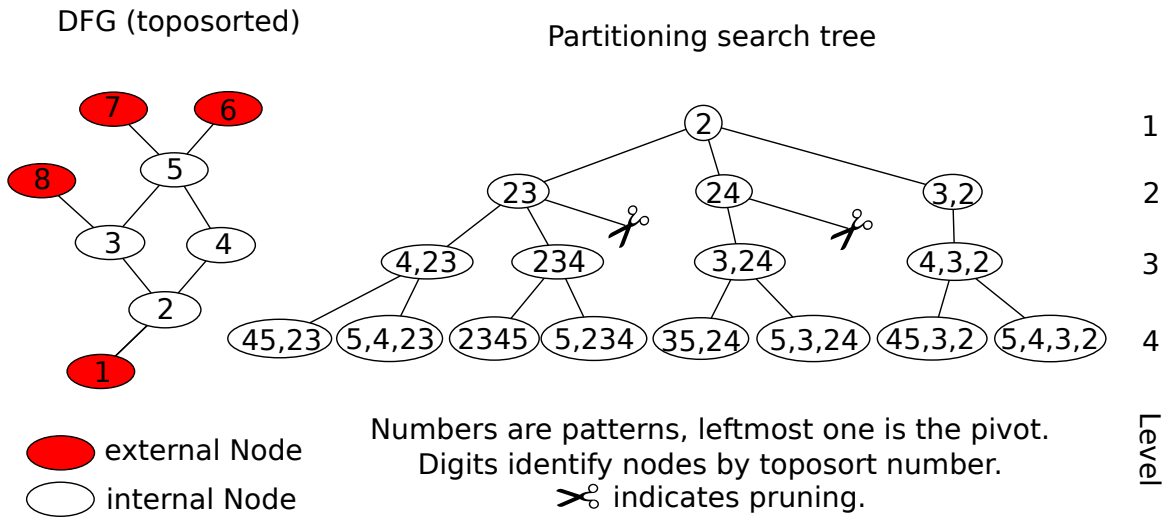

Figure 11.1: Topologically sorted DFG along with the complete search tree of the partition enumeration algorithm.

SI	types manual	types generated	atoms manual	atoms generated
htfour	1	4	8	81
satdfour	3	8	16	104
dctfour	2	9	12	90
sadsixteen	1	4	64	255

Table 11.1: Comparison of generated SI graphs vs. hand-crafted ones.

```
1 uint32_t popcount_a(uint32_t x)
2 {
3   x -= ((x >> 1) & 0x55555555);
4   x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
5   x = (x + (x >> 4)) & 0x0f0f0f0f;
6   x += x >> 8;
7   x += x >> 16;
8   return x & 0x3f;
9 }
```

Figure 11.2: C-Code from [?] to compute the number of set bits of a 32-bit value.

List of Tables

List of Figures

Bibliography

[BDPA08] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In *International Conference on the Theory and Application of Cryptographic Techniques*, 2008.

[Cza21] Jan Czajkowski. Quantum indifferentiability of sha-3. Cryptology ePrint Archive, Paper 2021/192, 2021. https://eprint.iacr.org/2021/192.

[Dae95] Joan Daemen. Cipher and hash function design strategies based on linear and differential cryptanalysis. 1995.

[Dwo15] Morris Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions, 2015-08-04 2015.

[MRH04] Ueli Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In Moni Naor, editor, *Theory of Cryptography Conference — TCC 2004*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39. Springer-Verlag, 2 2004.

[PA11] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche. The Keccak reference. Round 3 submission to NIST SHA-3, 2011.

[RS04] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In Bimal Roy and Willi Meier, editors, *Fast Software Encryption*, pages 371–388, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.