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.

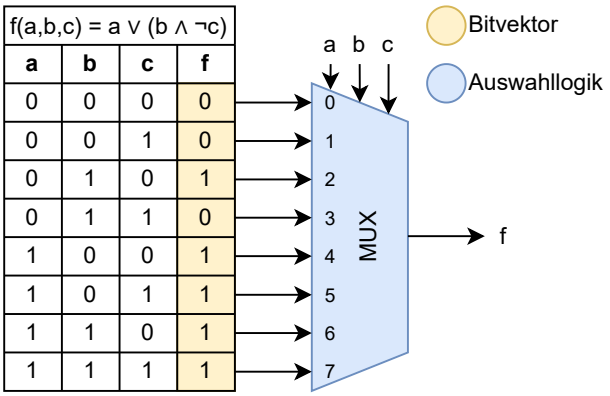

Figure 2.1: Implementierung einer dreistelligen Funktion durch einen Lookup Table

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

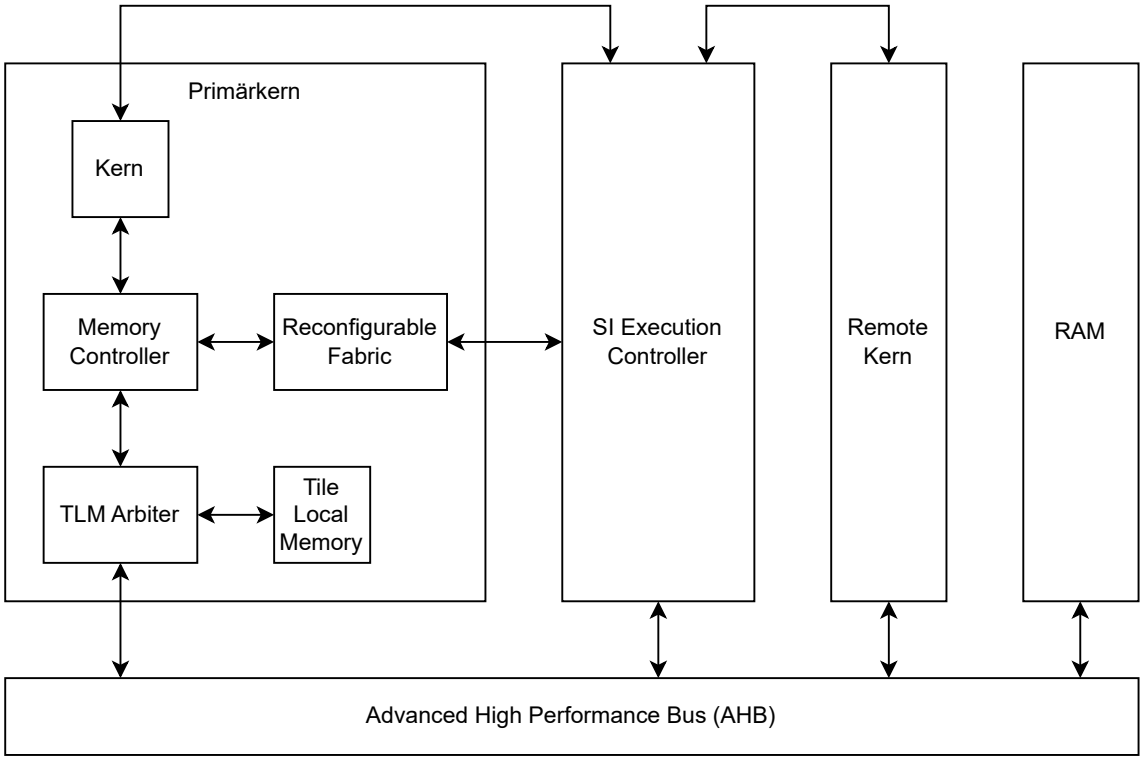

Figure 2.2: Aufbau des icore mit Ausführungskontrolle und Reconfigurable Fabric (Nachbildung aus (Cite missing, [3]))

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

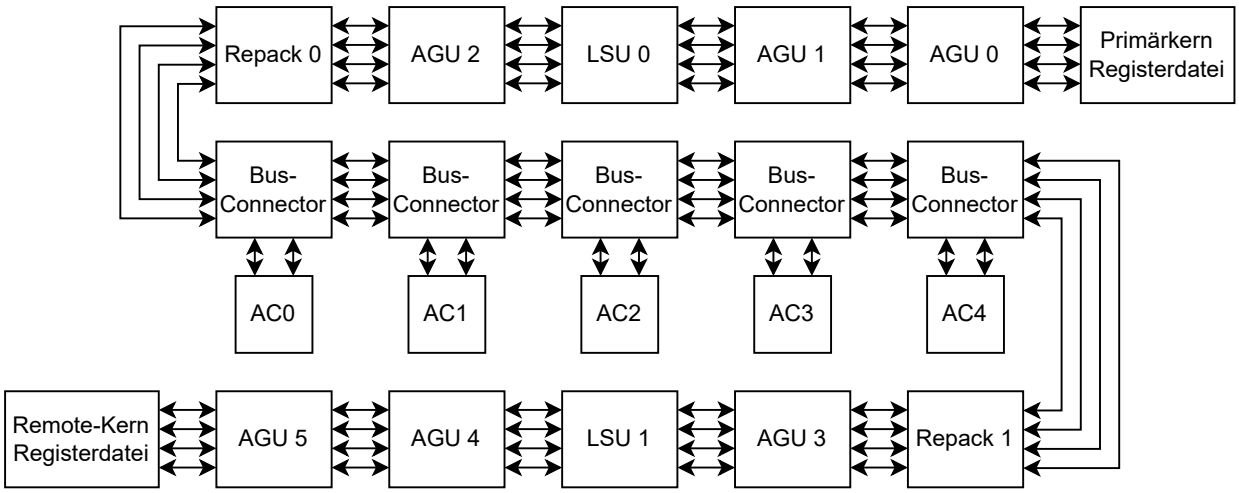

Figure 2.3: Aufbau der Reconfigurable Fabric

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 ist eine vom US-amerikanischen "National Institute of Standards and Technology" definierte Familie von kryptographischen Hashfunktion. Bei einer kryptographischen Hashfunktion handelt es sich um eine Funktion, die zu einer Eingabe beliebiger Länge eine Art Prüfsumme fixer Länge berechnet, den sogenannten Hash. Dieser Hash dient als eine Art Fingerabdruck. Damit eine solche Hashfunktion als kryptographisch sicher gilt, soll es nicht effizient möglich die Funktion umzukehren. Generell erwartet man folgende grundlegende Eigenschaften von solchen Hashfunktionen H:

1. Schwache Kollisionsresistenz: Zu einer gegebenen Nachricht M soll es praktisch unmöglich sein eine zweite Nachricht M' zu finden, die den gleichen Hash besitzt, also $H(M) = H(M')$

2. Starke Kollisionsresistenz: Es soll praktisch unmöglich sein zwei Nachrichten M und M' zu finden, die den selben Hashwert gesitzen, also $H(M) = H(M')$

Bei der Frage, was als "praktisch unmöglich" zählt, betrachtet man typischerweise eine ganze Familie an Hashfunktionen $\mathbb{H}$ mit einem auswählbaren Sicherheitsparameter n und verlangt, dass es keine probabilistische Turingmaschine gibt, die das entsprechende Problem mit einer Wahrscheinlichkeit $p > \frac{2}{3}$ in einer Laufzeit polynomiell in n löst. Das Problem liegt also außerhalb der Komplexitätsklasse BPP. Analog definiert man auch die Resistenz gegen Quanten-Angriffe, indem man verlangt, dass das Problem auch außerhalb der Komplexitätsklasse BQP liegt.

Um Anfragen beliebiger Länge bearbeiten zu können, bestehen Hashfunktionen typischerweise aus drei 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. Aus einer **Kompressionsfuntion** oder wie im Fall von SHA-3 einer **Permutation** wird dann eine Funktion konstruiert, die nacheinander die Blöcke mit dem Ergebnis der Permutation kombiniert und weiterverarbeitet. Während viele bekannte Hashfunktionen dazu die Merkle-Damgård-Konstruktion, verwendet SHA-3 die sogenannte **Schwammkonstruktion**. Wie diese genau aussieht und wie sie funktioniert, werden wir später in Abschnitt 3.3 sehen.

Dazu wird die Eingabe mithilfe eines **Paddings** in mehere gleich große Blöcke aufgeteilt. Die Blöcke werden nacheinander über eine **Schwammkonstruktion** zu einem 1600 Bit breiten Bitvektor kombiniert, woraus dann der endgültige Hash ausgelesen wird. Neben den vier Kandidaten, die in Tabelle 3.1 dargestellt sind, gibt es außerdem noch die beiden

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

Funktionen $SHAKE128$ und $SHAKE256$, bei denen sich die Ausgabegröße beliebig festlegen lässt. Sie unterscheiden sich in ihrer Funktionsweise nicht wesentlich von den anderen Funktionen, weshalb wir sie hier nicht weiter beachten werden.

3.1 Keccak-Permutation

Als Grundlage für alle SHA3-Funktionen dient eine Instanz der Keccak-Permutationsfamilie KECCAK-f. Für eine kryptographische Sicherheitsanalyse betrachtet man in der Regel das asymptotische Verhalten der erwarteten Laufzeit 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 (Cite missing, [2]) festgelegt wird. Die genaue Definition mit variablem Sicherheitsparameter ist zum Nachlesen auch in (Cite missing, [2]) angegeben. Wir wollen uns nun zuerst einmal den Zustandsvektor, auf dem die Permutation arbeitet, ein wenig genauer anschauen, bevor wir uns die fünf Unterfunktionen ansehen, aus denen die KECCAK-f Permutation aufgebaut ist.

3.1.1 Zustandsvektor

Die KECCAK-f Permutation arbeitet auf einem 1600 Bit breiten Zustandsvektor, auch **State Array** genannt. Am besten lässt er sich als dreidimensionale Struktur der Form 5x5x64 Bit vorstellen, siehe Abb. 3.1. Wir indizieren ein solches State Array **A**[x][y][z] über $x, y \in \{0, ..., 4\};\ z \in \{0, ..., 63\}$. Außerdem nennen wird

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

Die Konvertierung eines eindimensionalen Bitvektors **V** in dieses dreidimensionale State Array **A** funktioniert wie folgt:

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

Die Lanes werden also der Reihe nach erst in x-Richtung und dann in y-Richtung mit dem Inhalt von **V** gefüllt.

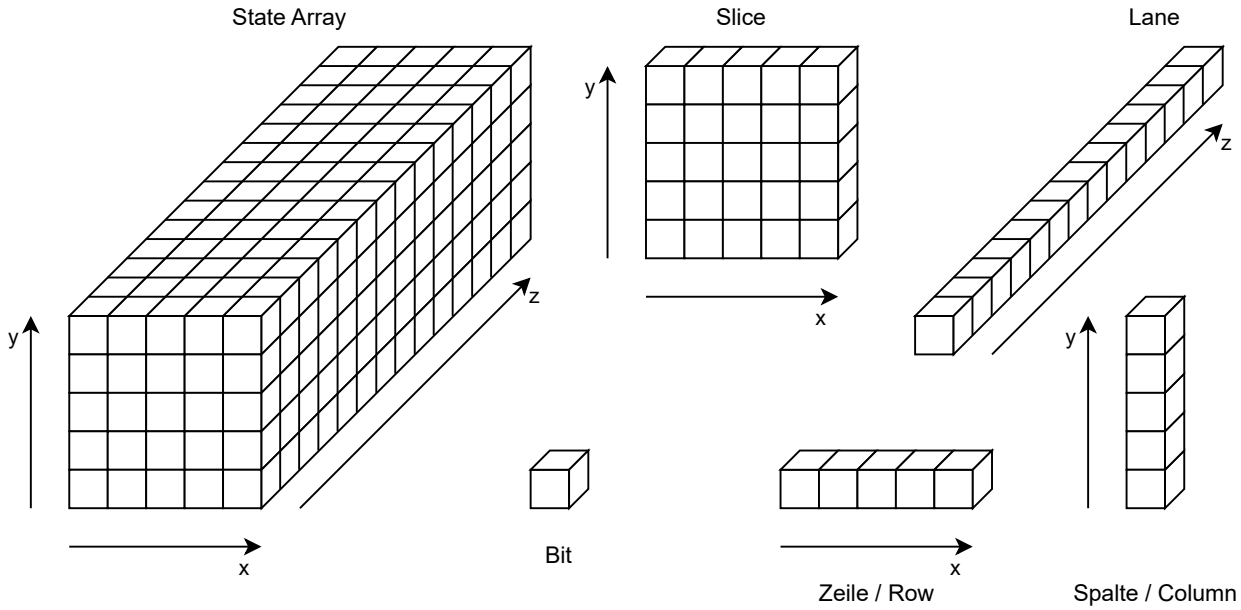

Figure 3.1: Blockrepräsentation des State Array

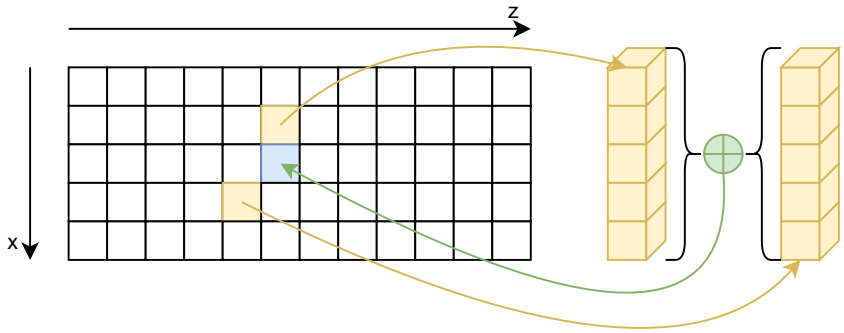

Figure 3.2: Spaltensummierung der θ-Funktion

3.1.2 Unterfunktionen

3.1.2.1 Theta-Unterfunktion

Die erste Unterfunktion der Keccak-Permutation ist Theta (θ). Sie modifiziert jedes Bit eines State Arrays, indem sie zwei benachbarte Spalten auf das Bit aufsummiert, siehe Abb. 3.2.

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

Man beachte, dass hier die State Arrays nur mit x und y indiziert werden, alle Operationen finden also immer auf ganzen Lanes gleichzeitig statt. Ein 64-Bit Prozessor kann also Theta mit Hilfe von ein paar bitweisen XOR-Operationen auf 64-Bit Operanden berechnen. $\mathbf{C}$ dient hierzu als Zwischenspeicher für Spaltensummen und $\mathbf{D}$ wird verwendet, um jeweils zwei Spaltensummen aufzuaddieren.

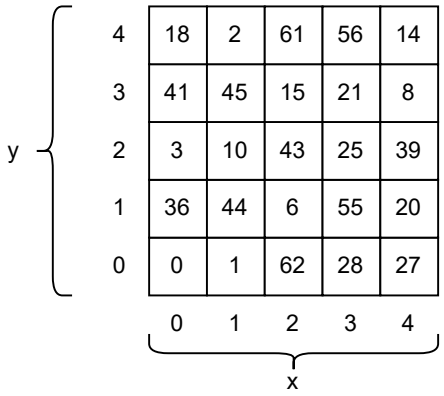

Figure 3.3: Rotationsdistanzen der Lanes für ρ

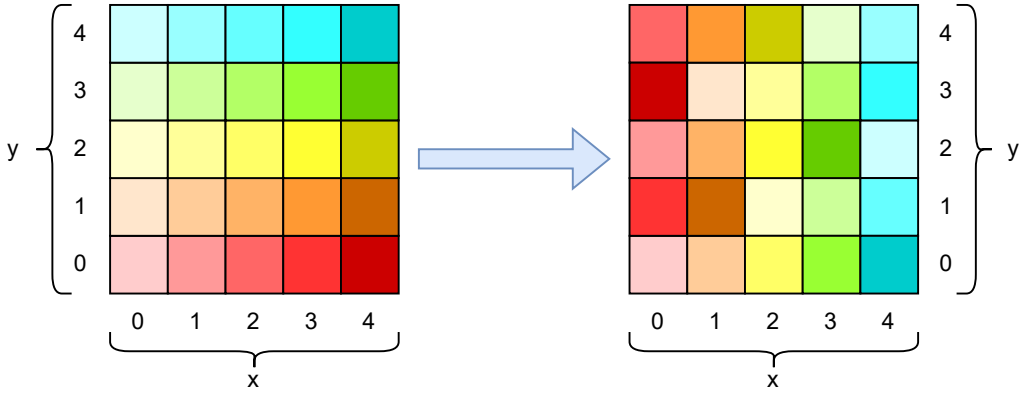

Figure 3.4: Visualisierung der *pi*-Permutation

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 Abb. 3.3 aufgeführt.

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

3.1.2.3 Pi-Unterfunktion

Die Pi-Unterfunktion (π) permutiert die Lanes eines State Arrays untereinander nach einer einfachen Vorschrift. Dass diese Abbilung injektiv ist, folgt direkt aus der Nullteilerfreiheit des Rings $\mathbb{Z}/5\mathbb{Z}$, oder man akzeptiert einfach die Veranschaulichung in Abb. 3.4.

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

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

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 (Cite Missing, [1]), 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.

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

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 XOR 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 $\iota(\mathbf{A}, r)$ 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:

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

3.1.4 KECCAK-f

Für die letztendliche Permutation wird die Rundenfunktion KECCAK-p mehrfach hintereinander angewendet. Wir definieren hier allerdings nur die eine Instanz der gesamten KECCAK-f Familie, die wir am Ende für die SHA-3 Hashfunktionen benötigen. Die vollständige Definition der Familie ist in (Cite Missing, [2])

$$\text{KECCAK-f}(\mathbf{A}) \coloneqq (\bigcirc_{i=0}^{23} \text{KECCAK-p}_i)(\mathbf{A})$$

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

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

3.3 Schwammkonstruktion

Um beliebig lange Eingaben zu einem kurzten Hashwert komprimieren zu können, 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 Abb 3.5 veranschaulicht, erst mit Hilfe einer Padding-Funktion in mehrere gleich große Blöcke einer festgelegten Länge abgebildet. 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. Der genaue Definition der Schwammkonstruktion über einer Transformation f mit einer Paddingfunktion pad sieht folgendermaßen aus:

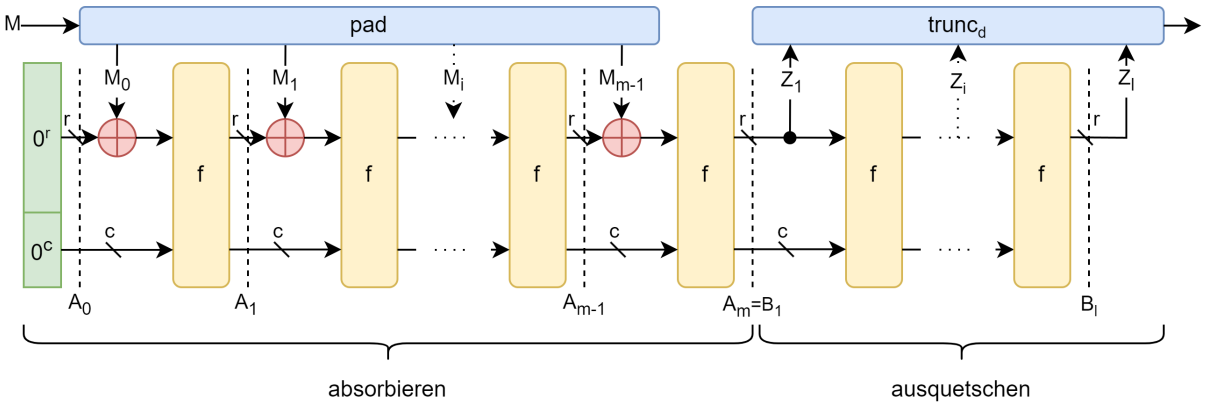

Figure 3.5: Aufbau der Schwammkonstruktion [Cite missing]

Seien $n \in \mathbb{N}$	die Transformationsbreite,
$c \in \{1, ..., n\}$	die Kapazität (Anzahl nicht von Blöcken veränderbarer 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:

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

3.3.1 SHA3-Hashfunktionen

Die in 3.1 genannten Hashfunktionen sind nun Instanzen dieser Schwammkonstruktion:

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

Die zwei Extrabits "01", die an die Nachricht angefügt werden, dienen nur dazu die erzeugten Werte von denen 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 oben 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 (Abb.). Da für eine gehashte Nachricht M nicht die ganze Ausgabe $Hash \parallel Trunc$ bekannt ist, sondern nur $Hash$, muss um aus $Hash$ eine Nachricht M' berechnen zu können, für die SHA3-256$(M') = Hash$ gilt, Eine Belegung für Trunc gefunden werden, sodass die letzten 512 Bits der Eingabe in die KECCAK-p Funktion alle 0 sind. Selbst wenn die verwendete Permutation einfach invertierbar ist, folgt daraus also nicht gleich, dass auch die Schwammkonstruktion über der Permutation einfach umkehrbar ist. Diese Überlegung reicht natürlich noch nicht als Beweis, aber gibt einen intuitiven Einblick in die Schwierigkeit, die dem Problem zugrunde liegt.

3.4.1 Sicherheit der Schwammkonstruktion

3.4.2 Kollisionsresistenz

3.4.3 Post-Quantum Sicherheit der Schwammkonstruktion

Chapter 4

Implementierung

4.1 Erste Iteration

4.1.1 Entwurfsziele

Die Architektur gibt eine maximale Größe von 1600 LUTs für seine Beschleunigerblöcke vor und ein Beschleuniger kann aus maximal 5 solcher Blöcke bestehen. Weiterhin permutiert die Keccak-Funktion ein 1600 Bit breites Feld, wobei jedes Bit von vielen anderen Bits an verschiedenen Stellen im Feld abhängt. Da die verschiedenen Blöcke nicht beliebig miteinander kommunizieren können, sondern nur über eine taktsynchrone Schnittstelle mit sehr begrenzter Bandbreite, folgt aus den beiden Beobachtungen die Vermutung, dass ein guter Beschleuniger aus so wenig Blöcken wie möglich besteht. Ziel des ersten Entwurfs war daher einen Beschleuniger zu entwickeln, der die Keccak-Permutation so schnel wie möglich in einem Block berechnet. Dabei war es erlaubt die Größenbeschränkung von 1600 LUTs in einem sinnvollen Maß zu überschreiten um daraufhin das Optimierungspotential des Entwurfs zu untersuchen und nach und nach die Größe zu reduzieren, bis schließlich das Größenlimit eingehalten wird. So sollten zum Beispiel nicht einfach alle 24 Runden in einem Takt über eine riesige kombinatorische Schaltung berechnet werden, da selbst für eine einzelne Runde die 1600 LUTs für die 1600 Bit Ausgabe schon kritisch sind. Weil sich die einzelnen Runden voneinander nur in einem Rundenindex unterscheiden, der sich in einer 64-Bit Konstanten in der Iota-Funktion manifestiert, hätte man eine um Größenordnungen zu große Schaltung mit sehr viel repetetiver Logik.

4.1.2 Aufbau

Daher Bestand der erste Entwurf direkt aus einer kombinatorischen Schaltung, die eine komplette Runde berechnet sowie einem Register, das die Ausgabe dieser Schaltung speichert und wieder als Eingabe bereit stellt. [Bild hier + Beschreibung]

4.1.3 Bewertung

Letztendlich besteht der erste Entwurf aus [circa 3500 LUTs, Zahl muss noch präzisiert werden] ohne Kommunikationslogik. Die Zahl setzt sich aus ungefähr 1600 LUTS für das Speichern der Daten zusammen und der Rest wird von der kombinatorischen Rundenfunktion benötigt. Das ist zwar noch etwa doppelt so groß wie das finale Design am Ende sein soll, aber mit ein paar Anpassungen kann die größe potentiell weiter reduziert werden.

4.1.4 Optimierungsansätze

Der wohl naheliegendste Gedanke ist den Datenspeicher zu reduzieren, da er mit seinen 1600 LUTs bereits das gesammte Budget alleine benötigt.

4.1.4.1 Aufspalten der Rundenfunktion

Um die Rundenfunktion in mehrere Teile aufteilen zu können, ist eine genauere Untersuchung der Funktion selbst notwendig um Teie ausfindig zu machen, die unabhängig voneinander berechnet werden können. Das Aufteilen in die gleichen Teilfunktionen, aus denen sie zusammengesetzt ist, ist nicht sinnvoll. Jede einzelne der Funktionen Theta, Rho,

Pi und Chi hat genau die gleiche Eingabelänge. Es müssen also auch die Teilfunktionen selbst aufgeteilt werden.

Folgendes lässt sich aber über die einzelnen Teilfunktionen festhalten: Theta ist eine Slice-orientierte Funktion. Jeder Slice S(i) der Ausgabe hängt nur von zwei Slices S(i) und S(i - 1) der Eingabe ab. Rho hingegen ist eine Lane-orientierte Funktion. Genauer genommen einfach ein Bit-Rotate jeder Lane, wobei die Weite der Rotation vom Index der Lane abhängt. Pi und Chi sind auch Slice-orientiert, anders als Theta hängt jedoch ein Ausgabe-Slice S(i) nur von einem Einzigen Eingabe-Slice S(i) ab. Iota ist ein Spezialfall; da sie nur eine Rundenkonstante auf eine einzige Lane aufaddiert. Dieses Operation kann aber auch als Slice-Operation Iota(S(i), i, r) dargestellt werden, die zusätzlich von dem Slice-Index i abhängt. Jede Teilfunktion lässt sich also Schrittweise mit einem Teil des Datenblocks berechnen, wobei Rho sich in der Hinsicht von den anderen Funktionen unterscheidet, dass sie Lane-orientiert arbeitet. Die Rundenfunktion Rnd(r) = Iota(r) . Chi . Pi . Rho . Theta kann also in mehrere Schritte unterteilt werden, die den Datenblock wiederum in jeweils mehreren Schritten verarbeiten. Weiterhin sollen allerdings so viele Teilfunktionen wie möglich zusammengefasst werden können, um die benötigte Anzahl an Schritten zu minimieren. Durch Abrollen der Permutationsfunktion lässt sich diese Anzahl auf zwei reduzieren. Rollt man die Permutationsfunktion KECCAK-P = .[r = 0 ... 23] Rnd(r) einmal ab, so erhält man nach Einsetzen der Definition von Rnd(r): KECCAK-P = Iota(23) . Chi . Pi . Rho . Theta . (.[r = 0 ... 22] Iota(r) . Chi . Pi . Rho . Theta) Dies erlaubt folgende äquivalente Schreibweise: KECCAK-P = Iota(23) . Chi . Pi . Rho . (.[r = 0 ... 22] Theta . Iota(r) . Chi . Pi . Rho) . Theta Anmerkung: Diese Schreibweise ist deshalb äquivalent, weil Theta nicht vom Rundenindex r abhängt. Mit Definition des Slice-orientierten Schlusses Alpha(r) = Iota(r) . Chi . Pi . Rho sowie dem Slice-orientierten Schleifenteil Beta(r) = Theta . Alpha(r) lässt sich die Permutationsfunktion schreiben als KECCAK-P = Alpha(23) . (.[r = 0 ... 22] Beta(r)) . Theta Der folgende Schritt ändert zwar rein semantisch nichts an den Teilfunktionen, erlaubt aber eine Implementierung, bei der jede Teilfunktion genau einmal in Logik umgesetzt werden muss wie Schaubild [Schaubild Nr] zeigt. Mit Gamma(r) = Theta — r = -1 Alpha — r = 23 Beta(r) — sonst und Delta(r) = Gamma(r) — r = -1 Gamma(r) . Rho — sonst fällt die Permutationsfunktion zusammen auf KECCAK-P = .[r = -1 ... 23] Delta(r)

[Schaubild zur Implementierung von Gamma und Delta]

Um Delta zu berechnen genügt es, zuerst schrittweise Rho zu berechnen (wenn r /= -1), wozu nur immer eine Lane benötigt wird und danach kann Gamma schrittweise aus den Slices des Zwischenergebnisses berechnet werden. Auf diese Weise kann die Rundenfunktion in möglichst große Teilschritte zerlegt werden, in denen die Datenabhängigkeiten der Ausgabebits möglichst gering sind.

4.1.4.2 BRAM als Datenspeicher

Ersetzt man das Flip-Flop-Register durch einen BRAM-Block, so spart man potentiell den gesammten Platz, den das Register einnimmt. Für jedes Bit, das gleichzeitig gelesen oder geschrieben wird, sollte man jedoch mindestens ein LUT einkalkulieren, damit nicht nur Daten von der kombinatorischen Schaltung, sondern auch von außerhalb geschrieben werden können. Diese Idee ist daher nur dann sinnvoll, wenn auch die Rundenfunktion weiter aufgeteilt wird, sodass nicht der ganze Block auf einmal als Eingabe vorliegen muss. Leider lässt sich dieser Ansatz nicht gut mit der vorherigen Idee verbinden, da der BRAM im Gegensatz zum Flip-Flop-Register es nicht erlaubt sowohl Slices als auch Lanes in nur

einem Takt auszulesen. [Schaubild mit Erklärung]

In Iteration 3 werden wir nochmal über diesen Ansatz weiterverfolgen, aber vorerst soll der Datenspeicher weiterhin als Flip-Flop Register implementiert werden.

4.1.4.3 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 Blöcke aufgeteilt werden, wobei jeder Block nur noch einen Teil des Datenblocks hält und nur für diesen ihm zugeteilten Teil die Rundenfunktion berechnet. Da das Ergebnis der Rundenfunktion auch noch von den Daten anderer Blöcke abhängt, müssen diese Daten über das Interface zwischen den Atomen ausgetauscht werden. Zwei Aspekte sind bei der Aufteilung zu beachten:

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 in der konkreten Architektur das Interface zum Bottleneck wird. Darüber hinaus führt die Erhöhung der Atom-Anzahl zu Leistungseinbußen.

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 baugleich sind, es würden also alle Atome mit dem gleichen Beschleuniger beladen und für welchen Teil der Daten ein Atom verantwortlich ist, wird ihm über einen Initialisierungsparameter mitgeteilt. Im Folgenden werden nun ein par der naheliegendsten Aufteilungsmuster untersucht:

4.1.4.4 2 Block Zeilen-orthogonal

[Schaubild] Spaltet man die Daten wie in Schaubild [Bild Nr] gezeigt, sodass jeder Block jeweils 32 der insgesamt 64 Slices enthält, so kann jeder Block sehr einfach die Gamma-Funktion für sein Block-Tile 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 ein wenig komplizierter.

4.1.4.5 2 Block Spalten-orthogonal

[Schaubild] Spaltet man die Daten entlang der Lanes, so muss für die Gamma-Funktion jeder Slice einmal zwischen den Atomen ausgetauscht werden. Die Berechnung kann allerdings weiterhin parallel erfolgen. Auch die Rho-Funktion kann parallel berechnet werden und benötigt keinerlei Kommunikation. Anmerkung zu Reihen-orthogonalen Mustern: Die Gamma-Funktion benötigt ganze Slices für die Berechnung, wenn ein Slice in einem Schritt berechnet werden soll. Daher bestehen für Reihen-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.

4.1.4.6 4 Block Muster

[Schaubild] Für die Aufteilung in 4 Blöcke können so die vorherigen Muster mehrfach angewendet oder auch miteinander kombiniert werden. Der Speicheraufwand sinkt hier zwar auf etwa 25Zudem steigt der Kommunikationsaufwand deutlich an, was nicht nur eine erhöhte Ausführungszeit mit sich bringt, sondern auch wieder mehr Platz im Atom benötigt.

Für den zweiten Entwurf soll daher das 2 Block Spalten-orthogonale Muster verwendet werden, da das Kommunikationsprotokoll am wenigsten komplex ist. Dadurch soll der Platz des Entwurfs möglichst klein gehalten werden auf Kosten der leicht höheren ausgetauschten Datenmenge und der damit verbundenen Ausführunszeit.

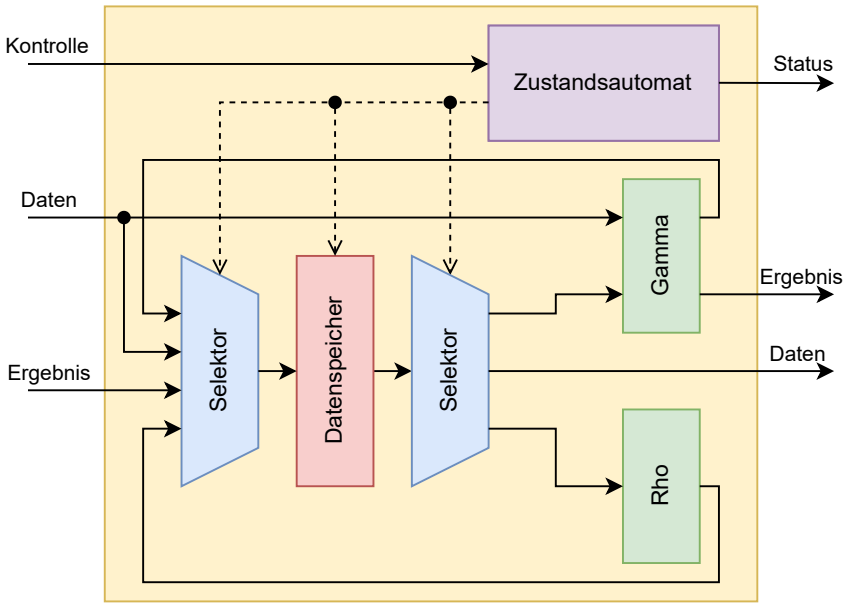

Figure 4.1: Atomaufbau des zweiten Entwurfs

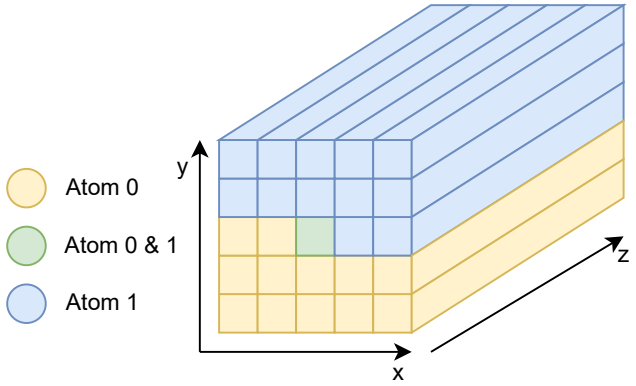

Figure 4.2: Aufteilung des Datenblocks

4.2 Zweiter Entwurf

4.2.1 Entwurfsziele

Im zweiten Entwurf sollen die Ideen aus dem vorherigen Abschnitt möglichst effizient umgesetzt werden. Das heißt der Beschleuniger wird in zwei Atome aufgeteilt, um die Datenmenge in einem Atom zu reduzieren und statt der Standard-Rundenfunktion wird die erweiterte Rundenfunktion in den zwei Teilschritten Gamma und Rho implementiert. 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. Ziel des Entwurfs ist es die bisherigen Überlegungen zu testen und sicher zu stellen, dass sie die gewünschte Auswirkung zeigen.

4.2.2 Aufbau

Der Beschleuniger ist in zwei Atome aufgeteilt (Abb. 4.1), die jeweils einen Teil des Datenblocks speichern. Beide Atome sind gleich aufgebaut, erst über ein Bit im Kontrollvektor, den sogenannten **Atom-Index**, wird festgelegt als welcher Teil des Beschleunigers ein Atom arbeitet. Die Daten sind dabei spaltenorthogonal aufgeteilt, sodass die Lanes 0 bis 12 in Atom 0, und die Lanes 12 bis 24 in Atom 1 gespeichert werden (Abb. 4.2). Die Lane 12 wird dabei absichtlich doppelt gespeichert, damit beide Atome 13 Lanes speichern, wodurch der gleiche Aufgebaut ermöglicht wird. Jedes Atom speichert somit 13 Lanes aus jeweils 64 Bits, also insgesamt 832 Bits. Diese Daten werden in einem Block aus FFs gehalten. Über einen Selektor werden aus diesem Datenspeicher die Daten für die Berechnungsblöcke Rho und Gamma sowie für die Kommunikation ausgewählt. Die Ergebnisse der Berechnungsblöcke sowie die empfangenen Daten werden über einen weiteren Selektor zusammengefügt und wieder im Datenspeicher gespeichert. Welche Daten von den Selektoren ausgewählt werden und welche im Speicher übernommen werden, bestimmt ein Zustandsautomat. Dieser wird über einen Kontrollvektor von außen gesteuert und koordiniert die einzelnen Berechnungsschritte und sorgt dafür, dass das Kommunikationsprotokoll befolgt wird.

4.2.3 Ablauf einer Berechnung

4.2.3.1 Dateneingabe

Über den Datenbus werden die Atome mit den Eingabedaten versorgt. Die Eingabe wird entsprechend mit den bereits gespeicherten über ein XOR kombiniert. Auf diese Weise können der Schwammkonstruktion entsprechend neue Datenblöcke direkt in den Atomen aufgenommen werden ohne, dass das Ergebis der vorherigen Berechnung erst gelesen werden muss.

4.2.3.2 Rho

Für die Berechnung von Rho werden alle Lanes in einem Atom gleichzeitig in einem Takt wie in einem Schieberegister entsprechend rotiert.

4.2.3.3 Gamma

Die Berechnung von Gamma wird in mehrere Teilschritte aufgeteilt. Atom 0 ist dabei für die Berechnung der Slices 0 bis 31 zuständig und Atom 1 führt die Berechnung für die Slices 32 bis 63 durch. Damit ein neuer Slice berechnet werden kann, muss der vollständige Slice im Atom vorliegen. Da jeweils 13 Bit schon im eigenen Datenspeicher vorhanden sind, müssen noch 12 Bits aus dem Speicher des anderen Atoms übertragen werden. Nach der Berechnung muss das Ergebnis in beiden Atomen übernommen werden. Dafür müssen wieder 13 Bits pro Slice übertragen werden. Um das Kommunikationsprotokoll so einfach wie möglich zu halten und die Ausführungszeit zu minimieren, wird die Berechnung ge-pipelined. Der 64 Bit breite Voll-Duplex-Kanal zwischen den Atomen wird aufgeteilt in einen 32 Bit breiten Datenkanal und einen 32 Bit breiten Ergebniskanal. Der genaue Berechnungsverlauf ist noch einmal im Schaubild [Bild Nr] erklärt. [Schaubild]

1. Die Daten für Atom 1 werden aus dem Datenspeicher gelesen und an den Datenkanal angelegt.

2. Die Daten für Atom 1 befinden sich im Register des Datenkanals

3. Die Daten für Atom 1 sind am Atom eingetroffen. Gleichzeitig treffen auch die von Atom 1 gesendeten Daten an Atom 0 ein. Die erhaltenen Daten werden mit den Daten aus dem Speicher von Atom 0 zu vollständigen Slices kombiniert und der Berechnungseinheit bereitgestellt.

4. Die Berechnungseinheit berechnet das Ergebnis und gibt es zurück.

5. Das Ergebnis wird im Datenspeicher übernommen und die Hälfte, die in Atom 1 gespeichert werden soll, wird am Ergebniskanal angelegt.

6. Das Erbebnis im Ergebnisbus befinden sich im Register des Ergebniskanals

7. Das Ergebnis trifft in Atom 1 ein. Gleichzeitig trifft auch das Ergebnis von Atom 1 in Atom 0 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 Kanal können maximal zwei 12 Bit bzw 13 Bit Einträge in einem Takt übertragen werden. Um den gesammten Blockteil zu berechnen, muss die oben aufgeführte Berechnungsabfolge also 16 Mal mit jeweils 2 Slices durchgeführt werden. Da die Berechnung der 7 Schritte in einer Pipeline durchgeführt wird, beträgt die Berechnungsdauer nicht 16 * 7 = 112 Schritte, sondern nur 7 + (16 - 1) = 22 Schritte.

4.2.4 Bewertung

Die Ausführungszeit für eine Iteration der modifizierten Rundenfunktion ist wie erwartet etwa um einen Faktor 30 langsamer als die Implementierung der ersten Iteration. 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 etwa 4600 LUTs nochmal um gut 37% größer. Dafür gibt es zwei wesentliche Gründe: den Zustandsautomaten sowie die Speicherkomplexität, die in der Überlegung für das Design nicht bedacht wurden.

4.2.4.1 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 etwa 300 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.

4.2.4.2 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 der Rho-Rotation 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 sowie die Entscheidung, wann genau das Bit im Register überschrieben werden muss (im ersten Entwurf wurden einfach alle Bits in jedem Takt überschrieben, wenn das Kontrollsignal aktiv war), benötigen schon mehr Platz als die Reduktion der Datenmenge einspart. Analog ist auch das Lesen der Daten komplizierter geworden. Die Gamma-Funktion sowie der Datenkanal müssen anhand des Zustandsautomaten und des Atom-Indexes aus allen Bits nur ein par auswählen.

4.2.4.3 Gamma-Funktion

Die Gamma-Funktion übernimmt bis auf Rho alle Subfunktionen der Standard-Rundenfunktion. Da die Berechnung auf zwei Atome aufgeteilt ist und auch nicht alle Slices in einem Atom gleichzeitig berechnet werden, ist die Komplexität der Gamma-Funktion sehr stark geschrumpft, sodass das aktuelle Design nur etwa 70 LUTs benötigt. Eine weitere Optimierung der Gamma-Funktion ist daher auch in weiteren Iterationen nicht mehr nötig.

4.2.5 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 erwähnt und uns letztendlich dagegen entschieden, weil das Festlegen auf einen Tile-orientierten Speicher bedeutet, dass die Rho-Funktion, die eigentlich auf Lanes arbeitet, so implementiert werden muss, dass sie mit Tiles arbeiten kann.

4.2.5.1 Transformation der Rho-Funktion

4.2.5.2 BRAM als Datenspeicher

4.2.5.3 Rho-Transformation

Das Berechnen der Rho-Funktion auf Tile orientierten Daten kann mit Hilfe eines Schieberegisters als Puffer realisiert werden. Schreibt man alle Rotationen, die größer als 32 Stellen sind, als Rechts-Rotationen um, so müssen in jedem Atom maximal 7 Links- und Rechts-Rotationen durchgeführt werden. Hier bietet es sich wieder an, die Berechnung in zwei Schritte aufzuteilen. Im ersten Schritt werden die Links-Rotationen berechnet, während die anderen Lanes nicht verändert werden. Im zweiten Schritt werden dann analog die Rechts-Rotationen durchgeführt. Um das Prinzip zu verdeutlichen, schauen wir uns zuerst die Verarbeitung einer einzelnen Lane an. Für die Rotation um k Bits einer Lane l mit k ¡ 32, kann die obere Hälfte von l rechts an l angefügt werden. w = l[63 downto 0] —— l[63 downto 32] rotl(l, k) = w[95 - k downto 31 - k]

Interessant ist, dass diese Berechnung über ein 32 Bit Schieberegister realisiert werden kann.

b[31 downto 0] := l[63 downto 32] for i in 0 to 63 loop r[i] := b[32 - k] b := (b ¿¿ 1) — (l[i] ¡¡ 63) end loop return r

Mit diesem Vorgehen können die Rotationen berechnet werden, wobei nicht die ganze Lane, sondern immer nur die Hälfte in einem Puffer vorliegen muss. Da die Berechnung

der Rechts-Rotationen analog funktioniert, kann ein Puffer für beide Arten von Rotation verwendet werden. Des weiteren lässt sich die schleife auch abrollen, womit mehrere Bits auf einmal verarbeitet werden können und natürlich können mit mehreren Puffern auch mehrere Lanes gleichzeitig rotiert werden. Dabei unterscheiden sich die einzelnen Lanes nur im Index der Auswahl ihrer Ergebnisbits. Für den konkreten Fall reichen insgesamt 7 Puffer aus. So können im ersten Schritt alle Links- und im zweiten Schritt alle Rechts-Rotationen berechnet werden. [Schaubild]

4.2.5.4 BRAM als Datenspeicher

Mit dem neuen Ansatz für die Berechnung der Rho-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 zwei Lese- und Schreibports. Das ist essenziell für die Berechnung der Gamma-Funkion, da durch die Pipeline in jedem Takt sowohl Daten für die eigenen Berechnungen als auch für die Berechnungen des anderen Atoms gelesen werden müssen und auch die Ergebnisse beider Atome gleichzeitig festgehalten werden. Da die Gamma-Funktion immer zwei Slices gleichzeitig verarbeitet, bietet sich dieses Format auch für die Speicherstruktur an. So können von jedem Port immer zwei Tiles gleichzeitig adressiert werden. Da jeder Atom-Contrainer über insgesammt 3 BRAM Blöcke verfügt, können die Ergebnisse der Gamma-Funktion auch in einem anderen BRAM-Block gespeichert werden. Die Rho-Funktion kann tatsächlich quasi inplace in einem BRAM-Block berechnet werden, da der BRAM read-before-write 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.

4.2.5.5 Datenbus

Da sowohl Gamma als auch Rho in der Zusammenarbeit mit dem BRAM wie es oben beschrieben ist, niemals auf zwei unterschiedlichen BRAM-Blöcken gleichzeitig schreiben oder gleichzeitig lesen, können die Datenleitungen für die beiden Speicherblöcke zusammengelegt werden. [Schaubild + Erklärung]

Figure 4.3: Atomlayout des Beschleunigers

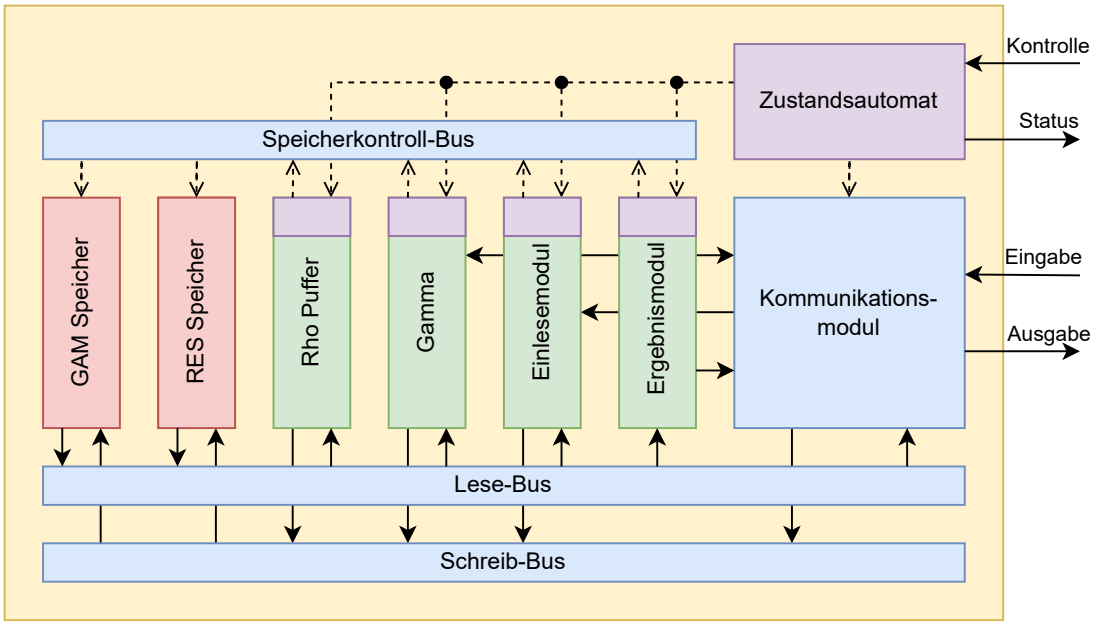

Figure 4.4: Aufbau der A-Atome

4.3 Finaler Entwurf

4.3.1 Entwurfsziele

Mit den vorgestellten Überlegungen soll versucht werden das Design endlich auf die erforderliche Größe zu schrumpfen. Die Ausführungszeit sollte dabei nicht mehr als bereits in den Überlegungen beschrieben.

4.3.2 Aufbau

Der Beschleuniger besteht diesmal aus insgesamt vier Atomen. Die beiden Atome A0 und A1 haben denselben Aufbau und sind für die eigentliche Berechnung der Permutationsfunktion zuständig. 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. Der Beschleuniger besteht wie auch im vorherigen Entwurf aus zwei identischen Atomen, deren Funktionsweise über den Atom-Index definiert wird. Da der Datenspeicher in den BRAM verschoben wird, müssen die anderen Komponenten so umgebaut werden, dass sie mit dem Interface des BRAM funktionieren. Dafür werden alle Komponenten sowie der BRAM über einen Speicherbus miteinander verbunden (Abb. 4.4).

4.3.2.1 BRAM

Die beiden verwendeten BRAM-Bänke GAM (für Gamma) und RES (für Result) besitzen jeweils zwei Lese-Schreib-Ports, mit denen jeweils ein Tile-Block (bestehend aus zwei benachbarten Tiles) gelesen und auch gleichzeitig geschrieben werden kann (read before

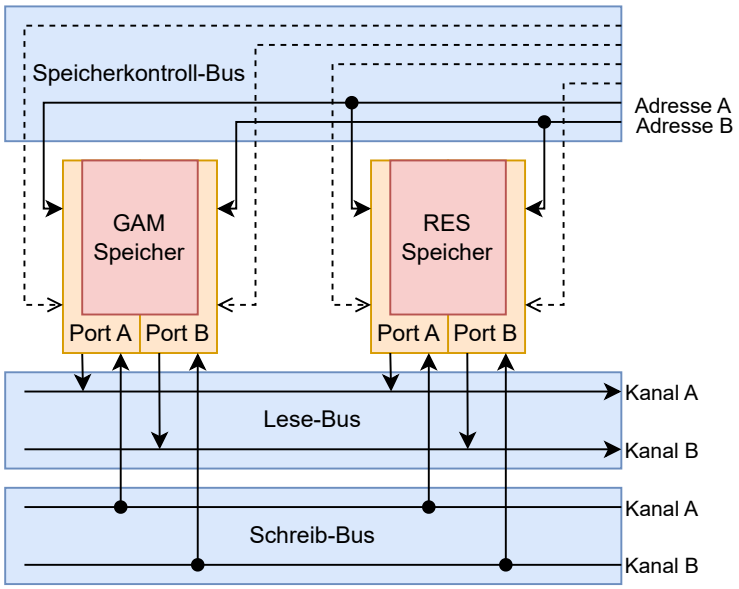

Figure 4.5: Speicheranbindung

write). Je ein Port ist dabei an den A-Kanal und der Andere an den B-Kanal angeschlossen.

4.3.2.2 Speicherbus

Der Speicherbus besteht aus drei Segmenten (Abb. 4.5). Auf dem Lese-Bus werden Daten aus dem BRAM ausgelesen und den anderen Modulen zur Verfügung gestellt. Er besteht aus zwei Kanälen, die beide einen Tile-Block breit sind. Auf dem Schreib-Bus werden Daten von den Berechnungsmodulen und dem Kommunikationsmodul gesammelt und an den BRAM weitergegeben. Auch dieser besteht aus zwei ein-Tile-Block breiten Kanälen. 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.

4.3.2.3 Zustandsautomat

Der Zustandsautomat besteht nicht mehr aus einer Einheit, sondern besteht nun aus einer zentralen Kontrolle sowie spezialisierten Kontrolleinheiten innerhalb der Berechnungseinheiten, angedeutet durch die lila Blöcke. Die zentrale Kontrolleinheit steuert dabei nur noch den Betriebsmodus des Kommunikationsmoduls und stößt die Abarbeitung in den Berechnungseinheiten an. Die Kontrolle innerhalb der Berechnungseinheiten 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 wieder an die zentrale Kontrolleinheit übergeben.

4.3.2.4 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

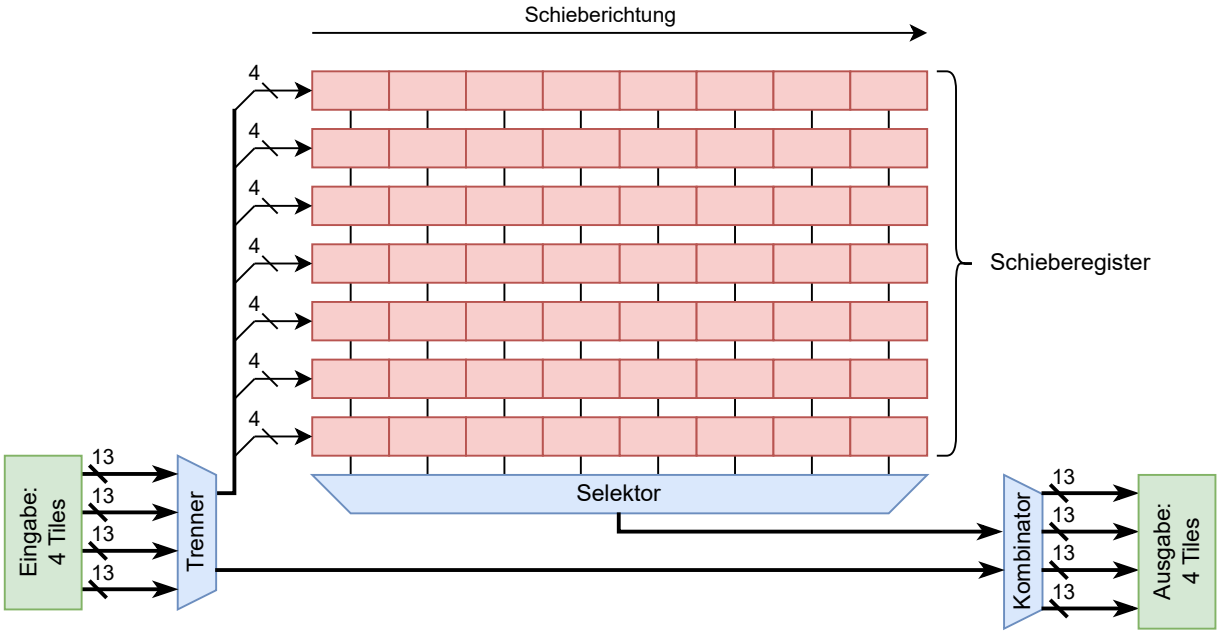

Figure 4.6: Aufbau des Rho-Puffers

werden und welche empfangenen Daten an den Speicherbus und die Berechnungseinheit weitergegeben werden. Dieser Betriebsmodus wird von der zentralen Kontrolleinheit bestimmt.

4.3.2.5 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 aus dem RES-Speicher und schreibt das Ergebnis in den GAM-Speicher

4.3.2.6 Rho-Puffer

Der Rho-Puffer besteht aus sieben 32-Bit-Schieberegistern, die wie in [Ref] verwendet werden, um interativ die Bit-Rotationen zu berechnen (Abb. 4.6). Dazu wird die Berechnung in zwei Schritte unterteilt. Im ersten Schritt werden die Daten 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 Daten aus dem GAM-Speicher erneut ausgelesen und diesmal wird auf den übrigen Lanes eine Rechtsrotation durchgeführt. Das finale Ergebnis der Rho-Berechnung wird dann im RES-Speicher wieder abgespeichert. Es ist die einzige Berechnungseinheit, die keinerlei Kommunikation benötigt und ist daher auch nicht an das Kommunikationsmodul angeschlossen.

4.3.2.7 Einlesemodul

Um das Ergebnis einer Permutation mit einem neuen Datenblock gemäß der Schwammkonstruktion zu kombinieren, müssen die eingelesenen Daten mit den Daten im Ergebnis-

speicher 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 Atomen direkt als Tiles bereitgestellt, sodass die Einleseeinheit nicht erst die Lanes in Tiles konvertieren muss. Da der Datenblock jedoch in Lanes gespeichert ist, muss dieser Konvertierungsprozess bereits im Vorhinein irgendwo ausgeführt werden. Diese Aufgabe übernehmen die Atome B0 und B1. Da die Berechnung der Permutation länger dauert als das Einlesen und die Konvertierung des Datenblocks aus dem externen Speicher, hat dieser zusätzliche Schritt keinerlei Auswirkung auf die Berechnungsgeschwindigkeit.

4.3.2.8 Ergebnismodul

Zur Ausgabe des 256-Bit Hashes wird das Ergebnismodul verwendet. Es besteht aus einem in FFs 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.

4.3.3 Berechnung der Permutationsfunktion

Da die Daten im BRAM nur Slice orientiert 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 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 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 im Abschnitt [Ref] 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 Block gemäß der KECCAK-P Funktion verarbeitet wurde. Danach kann über den Kontrollvektor 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 in einem FF-Puffer gespeichert. Als letztes werden diese Ergebnisbits als Lanes ausgegeben und im externen Speicher als Endergebnis gespeichert.

4.3.4 Bewertung

4.3.5 Weitere Optimierungsansätze

4.3.5.1 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 auf Kosten eines weiteren Kommunikationsschrittes. So kann noch ein wenig Platz für weitere Optimierungen geschaffen werden, die mit ein wenig mehr Platz die Berechnungsgeschwindigkeit wieder verbessern.

4.3.5.2 Reduktion auf 1 Atom

Die Berechnung wurde ursprünglich auf zwei Atome aufgeteilt, damit die Datenmenge reduziert werden kann, die ein Atom halten muss. Durch die Einführung des BRAM fällt diese Optimierung weg, da der BRAM mehr als genug Platz bereitstellt.

4.3.5.3 Erweiterung des Speicherinterfaces auf mehr Tiles

Die Aufteilung des Speichers in zum Beispiel 4 Tiles erlaubt es eine größere Menge an Tiles gleichzeitig für die Berechnung von Rho bereit zu stellen. Da in einem zwei Atom Ansatz die Gamma-Funktion durch die Bandbreite des Datenkanals begrenzt ist, würde dieser Teil keine Beschleunigung erfahren. In einem Ein-Atom-Ansatz hingegen würden allerdings auch hier noch mehr Slices gleichzeitig berechnet werden können.

4.3.5.4 Erweiterung des Rho-Puffers

Durch 1 Atom Ansatz müssen alle Lanes in einem Atom berechnet werden, kann der Puffer erweitert werden oder braucht das zu viel Platz? Müssen mehr als 2 Rotationen durchgeführt werden?

4.3.5.5 Auslagerung der Rho-Berechnung

Auch wenn alle Daten in einem Atom gespeichert werden, könnte es sinnvoll sein, die Berechnung von Rho auf verschiedene Atome zu verteilen. [Schaubild]. So können beim Filtern der Lanes die restlichen Lanes an ein anderes Atom übertragen werden, das die Rotation durchführt. So kann die Größenbeschränkung des Puffers umgangen werden. Wird auch dieser Prozess durch eine Pipeline realisiert, beträgt die maximale Bandbreite hierfür max k s.t. k * 7 ¡= 32 =¿ k = 4 Slices. Wird der Puffer im Hauptatom noch ein wenig erweitert, sodass maximal 6 Lanes gleichzeitig ausgelagert werden müssen, so steigt die Bandbreite sogar auf 5 Slices. Da allerdings immer zwei Ports für die Zusammensetzung der Daten verantwortlich sind, müssten immer 6 Slices gleichzeitig gelesen werden und die Restlichen in einem weiteren Puffer zwischengehalten werden was die Komplexität weiter

erhöht. Außerdem ist ein solcher Ansatz nicht mehr symmetisch, heißt es kann nicht der gleiche Beschleuniger auf beide Atome geladen werden.

4.3.5.6 Erhöhung der Berechnungsfrequenz (Nutzung der fallenden Taktflanken)

Chapter 5

Ergebnisse

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.

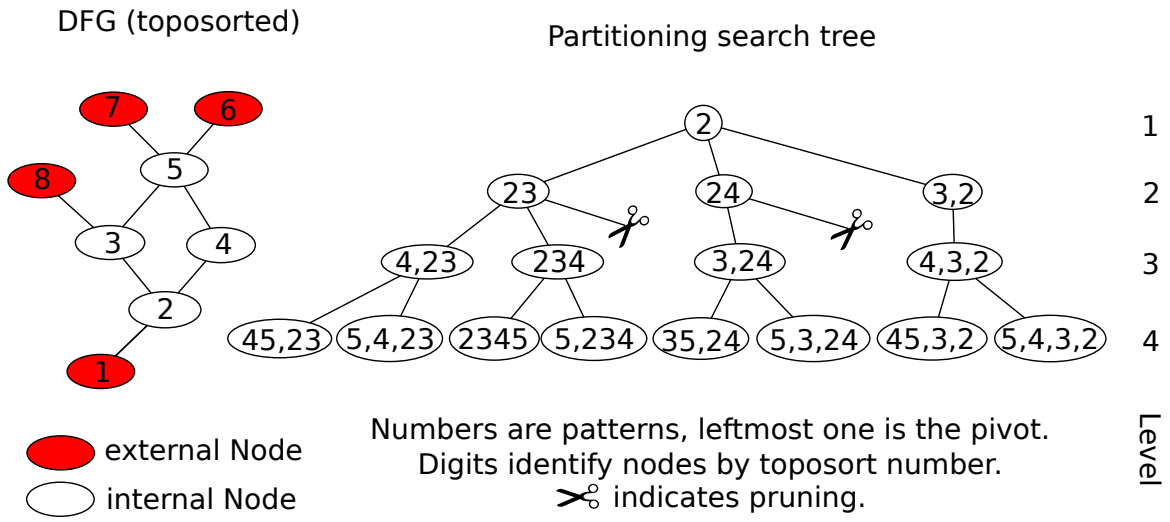

Figure 11.1: Topologically sorted DFG along with the complete search tree of the partition enumeration algorithm.

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 [War03] to compute the number of set bits of a 32-bit value.

List of Tables

List of Figures

Bibliography

[War03] H.S. Warren. *Hacker's Delight*. Addison-Wesley, 2003.