Untersuchungen zur hardwareoptimalen Implementierung digitaler Signalverarbeitungskomponenten

Von der Fakultät für Elektrotechnik und Informationstechnik
der Universität Dortmund
genehmigte Dissertation zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften

von

Mark Jung

aus

Bergisch Gladbach

Hauptreferent:	Prof. Dr.-Ing. H.-L. Fiedler
Korreferent:	Prof. Dr.-Ing. C. Paar
Einreichung:	Dortmund, 01. Dezember 2004
Mündliche Prüfung:	Dortmund, 28. Juni 2005

Danksagung

Die vorliegende Arbeit entstand während meiner 2002 beginnenden Tätigkeit als wissenschaftlicher Angestellter am Fraunhofer Institut für Mikroelektronische Schaltungen und Systeme (IMS).

An dieser Stelle möchte ich mich bei Herrn Prof. Dr. H.-L. Fiedler für das Betreuen und fortlaufende Begleiten meiner Arbeit bedanken. Er hat durch viele wertvolle Anregungen sehr zum Gelingen der Arbeit beigetragen.

Desweiteren danke ich Herrn Prof. Dr. C. Paar für die freundliche Übernahme des Korreferats und die Förderung meiner Arbeit durch sehr hilfreiche Diskussionen und Hinweise.

Zudem spreche ich allen Mitarbeitern der Abteilung ADS des Fraunhofer IMS meinen Dank aus. Hier möchte ich insbesondere Herrn R. Lerch, Herrn Dr. K. Gorontzi, Herrn H. Kappert und Herrn M. Engeln für ihre Unterstützung, die zahlreichen Diskussionen und ihr stetes Interesse am Fortgang meiner Arbeit danken.

Ganz besonders danke ich L. Obalski und meinen Eltern, die mich zu jeder Zeit geduldig und verständnisvoll unterstützt haben.

Duisburg, im Juli 2005

Mark Jung

Inhaltsverzeichnis

1 Einleitung und Überblick

In Forschung und Industrie gewinnen hochintegrierte Schaltungen für die digitale Signalverarbeitung zunehmend an Bedeutung. Gleichzeitig werden immer neue Einsatzgebiete für signalverarbeitende Hardware erschlossen. Neben der steigenden Anzahl der Einsatzdomänen unterliegen die unterschiedlichen Umsetzungsmöglichkeiten in Hardware stetiger Weiterentwicklung.

Gemeinhin werden für Aufgaben der Signalverarbeitung digitale Signalprozessoren (DSP) eingesetzt. Immer öfter findet jedoch eine Erweiterung von Standard-Mikrocontrollern mittels Prozessoreinheiten oder Coprozessoren um die Fähigkeiten eines DSP statt, sodass sich die Lücke zwischen DSP und Mikrocontroller zunehmend schließt.

Gegenstand der Arbeit ist die Entwicklung eines Systems zur gezielten Beschleunigung ausgewählter Signalverarbeitungsaufgaben, das mit minimalem Kostenaufwand die maximal mögliche Leistung erbringt. Dafür wird eine besonders zielgerichtete Anpassung des Systems an das Einsatzgebiet erforderlich. Idee ist es, im Gegensatz zu einem komplett anwendungsspezifischen Chip einen Standardcore lediglich modular um maßgeschneiderte Beschleunigungseinheiten zu erweitern, um eine erhöhte Flexibilität des Systems zu erreichen.

Die Implementierung solcher Erweiterungen basiert auf einer einheitlichen Grundstruktur für Coprozessoren zur Unterstützung digitaler Signalverarbeitungsaufgaben. Dieses Architekturgerüst wird ausgefüllt durch die in der Arbeit entwickelten Coprozessoren für den kryptographischen Algorithmus AES (Advanced Encryption Standard), namentlich den CpCiph, und zur Beschleunigung von Vektormultiplikationen und digitalen Filtern, den CpMac. Durch die großen algorithmischen Unterschiede dieser Operationen wird die Einsetzbarkeit des Architekturgerüsts für verschiedene Anforderungen verdeutlicht.

Besonderes Augenmerk gilt der Entwicklung einer leistungsfähigen und dennoch schlanken Schnittstelle zwischen Core und Erweiterung. Mehrere Entwürfe des CpCiph und CpMac werden implementiert und anhand des gewählten Optimierungskriteriums verglichen. Dabei fokussieren alle Entwicklungsstufen das ausgewählte Optimierungsziel: Für die Implementierung des AES wird die zugrundeliegende Galois-Feld-Arithmetik, und für die Vektormultiplikationen werden Addition sowie Multiplikation analysiert.

Kapitel 2 begründet die Auswahl der selektierten Algorithmen. Unter Berücksichtigung des technischen Standes erfolgt die Definition der Optimierungskriterien und der Ziele dieser Arbeit.
Die relevanten mathematischen Grundlagen der Galois-Feld-Arithmetik und die strukturelle Basis des AES werden in *Kapitel 3* eingeführt.
Darauffolgend dient *Kapitel 4* der Erläuterung des gewählten Controllers und der Grundstruktur der Beschleunigungseinheiten. Desweiteren wird ein Anforderungsprofil für die Einheiten thematisiert.
Kapitel 5 stellt mögliche Schnittstellen zwischen Beschleunigungseinheit und Core vor und diskutiert deren Vorteile und Beschränkungen. Den Schwerpunkt bilden die im Zuge der Arbeit entwickelten Interfacelösungen: schlicht, sequentiell, parallel und hybrid.
In *Kapitel 6* folgt die Partitionierung der auf das Architekturgerüst abgebildeten Algorithmen in logische Komponenten, die von Ablaufsteuerungen in eine zeitliche Sequenz gebracht werden. Die Partitionierung stellt das Optimierungskriterium der Logikminimierung in den Vordergrund, während die Ablaufsteuerung maximalen Durchsatz der zeitlich verknüpften Komponenten anstrebt.
Nach ihrer Selektion werden die Komponenten in *Kapitel 7* unter Abwägung unterschiedlicher Implementierungsmöglichkeiten hinsichtlich eines minimalen Flächenbedarfs optimiert. Dabei liegt der Schwerpunkt auf dem Vergleich einer ROM-Implementierung mit einer festverdrahteten Lösung für den AES Byte-Substitutionsschritt unter Zuhilfenahme der Transformation vom $\mathcal{GF}(2^8)$ in das $\mathcal{GF}((2^4)^2)$.
Kapitel 8 präsentiert die Ergebnisse der vorangegangenen Implementierungen und vergleicht diese mit Resultaten anderer Umsetzungen. Weiterhin wird die Simulations- und Testumgebung der entworfenen Architekturen thematisiert.
Abschließend fasst *Kapitel 9* die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick auf zukünftige Arbeiten.

2 Motivation und Stand der Technik

Digitale Signalverarbeitung beschäftigt sich mit der Manipulation zeit- und amplitudendiskreter Signale. In den letzten Jahren hat sich die Anzahl der signalverarbeitenden Aufgaben zunehmend erhöht. Mögliche Anwendungen für Algorithmen der digitalen Signalverarbeitung findet man in verschiedensten Bereichen wie z.B.

- Mobilfunk (GSM, UMTS),
- kabelgebundener Datenübertragung (DSL),
- Medizintechnik (Chipimplantate),
- Unterhaltungsmedien (MP3-Player) oder
- Chipkarten (Smart Cards).

Obengenannte Einsatzgebiete stellen hohe Anforderungen an signalverarbeitende und datenbearbeitende Operationen und Algorithmen. Zu diesen zählen unter anderem

- Datenverschlüsselung (AES, DES, RSA, ECC),
- Matrix- und Vektoroperationen,
- digitale Filterung (FIR[1], IIR[2]) und DFT,
- Quellencodierung und -decodierung (Huffmann, Fano) sowie
- Kanalcodierung und -decodierung (Hammingcodes).

Abhängig vom ausgewählten Algorithmus, den Anforderungen und dem Einsatzgebiet gibt es verschiedene Umsetzungsmöglichkeiten. Dabei sind sowohl Software- als auch Hardwarelösungen verbreitet.
In den folgenden Abschnitten werden mögliche Architekturen vorgestellt, welche die Implementierung digitaler Signalverarbeitungsanforderungen unterstützen.

[1] Finite Impulse Response

[2] Infinite Impulse Response

Desweiteren wird der Stand der Technik anhand von gängigen Architekturkonzepten aus Forschung und Entwicklung erörtert.

Im Zuge einer Definition der Begrifflichkeit der "hardwareoptimalen" Implementierung erfolgt die Vorstellung möglicher Optimierungskriterien und ihrer gegenseitigen Korrelation. Anhand zweier unterschiedlicher Zielanwendungen werden die Optimierungskriterien priorisiert und Implementierungsziele definiert.

2.1 Auswahl der Algorithmen

Für die folgenden Ausführungen seien zwei Anwendungen ausgewählt, anhand derer die Optimierung in der Umsetzung verdeutlicht wird. Die für diese Arbeit entwickelten Architekturen sollen Anwendung finden in

- Datenverschlüsselung und
- Vektormultiplikation.

Die Auswahl dieser Anforderungen begründet sich durch die zu erwartenden Unterschiede in der Hardwareumsetzung, um zu demonstrieren, dass ein breites algorithmisches Spektrum auf das im Verlauf der Arbeit entwickelte Architekturgerüst abbildbar ist. Die Optimierung der dedizierten Datenverschlüsselungseinheit wird vorrangig betrachtet, die Vektormultiplikationseinheit hingegen immer dann, wenn ihre Umsetzung in Hardware deutlich von jener der Chiffriereinheit abweicht.

2.1.1 Datenverschlüsselung mit dem AES

Da digitale Systeme in immer sensiblere Bereiche unseres Alltags vordringen, ist die Sicherstellung der Geheimhaltung privater Daten durch Verschlüsselung unabdingbar. Die Kryptographie kann als ein Teilgebiet der Codierungstheorie betrachtet werden,[3] die wiederum ein Teilgebiet der Signalübertragung ist. Kryptographische Hardware- und Softwarelösungen werden in unterschiedlichsten Bereichen eingesetzt. Mögliche Einsatzgebiete sind z.B.

[3] vgl. [Hen74], S.42

- Smart Cards [Dhe01],
- Automobilelektronik [Pa04],
- Netzwerkrechner,
- Mobilfunkempfänger oder
- Chipanwendungen in der Medizintechnik.

In der Kryptographie unterscheidet man hauptsächlich zwei Verschlüsselungsmethoden: das asymmetrische und das symmetrische Verfahren. Während bei asymmetrischen Verfahren Sender und Empfänger keine Rückschlüsse über den jeweils anderen Schlüssel ziehen können, benutzen in symmetrischen Verfahren Sender und Empfänger voneinander ableitbare Schlüssel. Diese Arbeit behandelt ausschließlich einen symmetrischen Algorithmus, namentlich den Advanced Encryption Standard (AES). Weitergehende Informationen über asymmetrische Kryptographie sind in [PKC00] und [Schn96] beschrieben.

Der AES wurde im Oktober 2000 von der NIST (US National Institute of Standards and Technology) zum neuen Datenverschlüsselungsstandard ernannt [Fips01] und ist vom Rijndael Algorithmus von Vincent Rijmen und Joan Daemen [Dae99] abgeleitet. Er ersetzt den Data Encryption Standard (DES) [Fips77] bzw. den 3-DES[4] und ist Federal Information Processing Standard (FIPS) [Fips01].

Die Charakteristika des AES sind

- 128 Bit Blocklänge,
- wahlweise 128, 192 oder 256 Bit Schlüssellänge,
- 10, 12 oder 14 nahezu identische Rundentransformationen,
- gute Implementierbarkeit (auch auf 8-Bit-Prozessoren),
- hohe Datenraten und vor allem

[4] Der 3-DES war lediglich eine Erweiterung, um den längst unsicher gewordenen DES weiter als Standard nutzen zu können. Die zu verschlüsselnde Nachricht wird mit verschiedenen Schlüsseln zuerst verschlüsselt, danach entschlüsselt und zuletzt nochmals chiffriert.

- hohe Sicherheit gegen kryptoanalytische Attacken[5].

Erläuterungen zu den o.g. Charakteristika und algorithmische Details werden in Kapitel 3 vorgestellt.

2.1.2 Vektormultiplikationen

Zum Vergleich wird mit der Beschleunigung von Vektoroperationen eine zusätzliche Anforderung ausgewählt, deren optimale hardwaretechnische Umsetzung Erörterung findet. Dies ist u.a. für folgende Anwendungen der Signalverarbeitung interessant:

- Matrixoperationen (z.B. DFT) oder
- Filter (z.B. FIR oder IIR)[6].

Die Basisoperation für Vektoroperationen stellt die Multiplikation zweier Faktoren und die anschließende Addition zu einem bereits berechneten Wert dar. Diese Bildung der Summe von Produkten wird auch Multiply-Accumulate- (MAC) Operation genannt. Desweiteren sollen Operationen zur Normalisierung des Ergebnisses und zur Unterstützung von einfachen Multiplikationen bzw. Additionen vorgesehen werden.

2.1.3 Vergleich der Signalverarbeitungsaufgaben

Die beiden Signalverarbeitungsaufgaben wurden aufgrund ihrer großen Verschiedenheit ausgewählt, um ein möglichst breites Anforderungsspektrum abzudecken. Zu den Unterschieden zählen:

[5] Diese Arbeit beschäftigt sich nicht mit der Sicherheit des AES oder seiner Umsetzung. Details zu diesem Thema behandeln u.a. [Dae00], [Gol03], [Luc00] oder [Tri03].

[6] vgl. [Che79], S.191ff

AES	Vektormultiplikationen
• Datenverschlüsselung	• digitale Filterung
• basiert auf Galois-Feld-Arithmetik ($\mathcal{GF}(2^8)$)	• basiert auf Arithmetik der rationalen Zahlenmenge ($\mathbb{Q}$)
• große Eingangsdatenblöcke (128-Bit Nachricht oder Chiffre und 128-Bit Schlüssel)	• kleine Eingangsdatenblöcke (hier zwei 16-Bit-Faktoren)
• große Ausgangsdatenblöcke (128-Bit Chiffre oder Nachricht)	• kleine Ausgangsdatenblöcke (Akkumulationsergebnis)
• komplexe Abfolge einfacher Schritte	• aufwendige Operationen in einfacher Abfolge
• wenige Operationen, die spezielle Anforderung vollständig durchführen (Encrypt, Decrypt, Schlüsselexpansion).	• viele atomare Teiloperationen, die nur einen Teil des Gesamtproblems bearbeiten (z.B. MAC oder Multiplikation)
• nur für spezielle Anforderung einsetzbar (128-Bit-Ver- und Entschlüsselung mit AES)	• vielfältig einsetzbar (z.B. Multiplikation von Matrizen beliebiger Größe)

Die tabellarische Übersicht zeigt, dass gravierende Unterschiede zwischen den Algorithmen bestehen. Dennoch sollen diese hardwareoptimal auf dieselbe architektonische Struktur projiziert werden. Welche möglichen Architekturen sich anbieten, zeigt der folgende Unterpunkt.

2.2 Architekturen

Mittels einer kurzen Beschreibung der Architekturen und ihrer Vertreter werden ihre Vorzüge und Nachteile für den Einsatz in der Signalverarbeitung erläutert. Im Mittelpunkt des Interesses steht die Implementierbarkeit von MAC-Operationen und des AES-Verschlüsselungsalgorithmus.

2.2.1 DSP und Secure Core

Durch die stetig wachsenden Ansprüche an Mikroprozessoren in der Sprach- und Bildsignalverarbeitung hat der digitale Signalprozessor (DSP), der Anfang der 80er Jahre mit der Entwicklung der MOS[7]-Technologie seinen Durchbruch schaffte, zunehmend an Bedeutung gewonnen [Dob00]. DSP zeichnen sich insbesondere durch die Fokussierung signalverarbeitender Operationen und deren Verarbeitung mit sehr hohem Durchsatz aus.

Ihre Architekturen basieren sehr häufig auf dem CISC[8]-Konzept,[9] um möglichst viele Signalverarbeitungsaufgaben mit wenigen Instruktionen abzuarbeiten. Gemeinhin wirkt sich dieses negativ auf die Orthogonalität[10] des Instruktionssatzes aus.

Häufig steht die Minimierung der Fläche und somit der Kosten sowie die Verlustleistungsoptimierung weniger im Vordergrund als ein maximaler Datendurchsatz, auch wenn diese Aspekte durch den Zuwachs an eingebetteten Systemen in den letzten Jahren kontinuierlich zugenommen hat. Eine Erhöhung des Instruktionsdurchsatzes wird häufig durch VLIW[11]-Prozessoren wie den C62x von Texas Instruments (TI) erzielt, welcher acht parallele Probleme (Issues) pro Taktzyklus bearbeitet [HP03].

Zur Verbesserung des Datendurchsatzes sind DSP üblicherweise als Harvardarchitektur konzipiert. Das bedeutet, dass sie über einen separaten Daten- und Programmbus sowie über separate Daten- und Programmspeicher verfügen. Dadurch können sie im Gegensatz zu einer von-Neumann-Architektur in einem Taktzyklus neue Instruktionen laden und lesend oder schreibend auf den Datenbereich des Hauptspeichers zugreifen.
Weitere Beschleunigung erreicht man bisweilen durch eine modifizierte Harvardarchitektur, bei der durch Einbau eines Multiplexers zwischen Daten- und Programmbus Operanden auch mit dem Programmspeicher ausgetauscht wer-

[7] Metal Oxide Semiconductor

[8] Complex Instruction Set Computer

[9] vgl. [HP03], S.155

[10] Unter Orthogonalität versteht man die Regelmäßigkeit des Instruktionssatzes. Bei einem Prozessor mit einem orthogonalen Instruktionssatz kann man die meisten Instruktionen mit allen Registern, Adressierungsarten und teilweise auch allen Sprungarten ausführen.

[11] Very Long Instruction Word

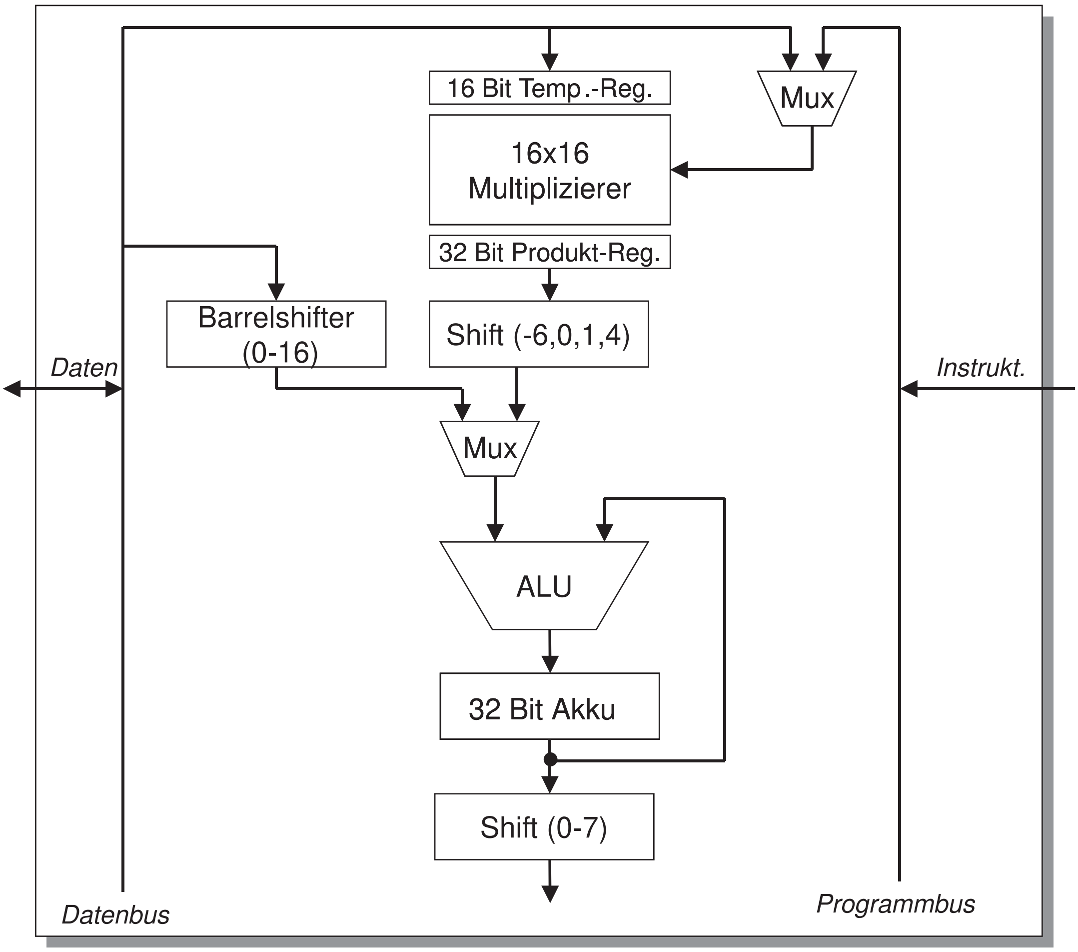


Abbildung 1: Prozessorkern des TMS320C25 von Texas Instruments.

den. Somit besteht die Möglichkeit, gleichzeitig zwei Operanden zu lesen oder zu schreiben. Die Auslastung beider Busse kann hierdurch weiter erhöht werden.

Ein Beispiel für eine solche modifizierte Harvardarchitektur ist der TMS320C25 von TI [TI02]. Dieser wird in Abb. 1 skizziert. Der Multiplexer zwischen Daten- und Programmbus, hier oben rechts abgebildet, ermöglicht es, auch aus dem Programmspeicher Operanden in die Multipliziererеinheit zu laden.

Der C25 verfügt über einen 16 Bit breiten Datenbus und einen ebenso breiten Programmbus. Kernstück der Architektur ist die ALU[12] (Abb. 1), welche nahezu alle Operationen in nur einem Taktzyklus durchführt. Trotz der nur 16 Bit

[12] Arithmetic Logical Unit

breiten Busse kann die ALU bereits einige 32-Bit-Operationen durchführen.

Speziell für signalverarbeitende Anforderungen hat der C25 einen 16x16 Bit festverdrahteten Multiplizierer und einen Barrelshifter, der das Verschieben der Eingangsdaten um eine beliebige Anzahl Stellen ermöglicht. Das 32 Bit breite Produkt wird in einem Produktregister abgelegt. Bevor das Resultat von der ALU bearbeitet wird, ist noch eine Normalisierung um einige ausgewählte Stellen (-6, 0, 1 und 4) möglich. Mit der ALU wird u.a. eine 32-Bit-Addition zum aktuellen Akkumulatorinhalt ermöglicht, sodass der C25 eine MAC-Operation zur Verfügung stellt.

Das Gesamtresultat der Instruktion wird im Akkumulator abgelegt. Zur Beschleunigung von Fließkomma-Rechenoperationen mit Hilfe dieser Festkomma-Architektur können alternativ zur Multiplikation Schiebeoperationen mit dem Barrelshifter vollzogen werden.

Durch Nutzung von Repeat-Instruktionen und das dadurch bewirkte mehrfache Wiederholen einer gewünschten Operation (z.B. der MAC-Operation) werden abermals Taktzyklen eingespart. Die Vorteile dieser signalverarbeitungsoptimierten Architektur haben allerdings einen unübersichtlichen Instruktionssatz mit Befehlen wie XORK (Exclusive-OR immediate with accumulator with shift) und einer geringen Orthogonolatität des Instruktionssatzes zur Folge.

In naher Vergangenheit sind Spezialprozessoren zur Gewährleistung von Datensicherheit speziell für Anforderungen der Datenverschlüsselung und der Datenintegrität konzipiert worden. Im folgenden werden sie als Secure Cores (SCs) bezeichnet.[13] Ein Vertreter dieser Prozessoren ist der MIPS32 4KSd [Mips02], der basierend auf der MIPS32 Architektur folgende Vorzüge bietet:

- benutzerdefinierte Instruktionssatzerweiterungen,
- kryptographische Erweiterungen für schnellere Chiffrierung,
- Sicherheit gegen Seitenkanalattacken[14] und
- geschützte Speicher.

[13]Der Trend ist noch recht jung, sodass sich noch keine eindeutige Bezeichnung für diesen Prozessortypen herausgebildet hat.

[14]Seitenkanalattacken greifen nicht den Algorithmus selber an, sondern versuchen anhand einer Analyse der Implementierung, die Daten zu entschlüsseln. Mögliche Angriffspunkte sind z.B. der Stromverbrauch oder die Dauer der Verschlüsselungsoperation.

MIPS verspricht dabei eine schnellere Bearbeitung kryptographischer Algorithmen als bei Standard-Prozessoren gleicher Fläche. Der MIPS32 4KSd wurde u.a. für die Anwendung in Smart Cards entwickelt, auch wenn er, bedingt durch seine 32-Bit-Architektur, sehr viele Gatter benötigt.

Auch ARM bietet einen SC an. Dieser basiert auf dem Instruktionssatz des in Punkt 2.2.2 erläuterten Prozessorkerns. Im Gegensatz zu den Standard-Prozessoren sind die SCs ähnlich wie die DSP auf eine Spezialfunktionalität zugeschneiderte Controller.

2.2.2 Mikrocontroller

Eine weitere Möglichkeit, signalverarbeitende Aufgaben durchzuführen, besteht in der Nutzung von General-Purpose-Mikrocontrollern. Seit geraumer Zeit zeichnet sich der Trend ab, Controller durch Spezialeinheiten für signalverarbeitende Aufgaben zu optimieren. Damit wird die ehemals strikte Trennung zwischen Controller und DSP aufgehoben und der Einsatzbereich von Mikrocontrollern weitet sich aus. Die Erweiterung um signalverarbeitende Einheiten ist jedoch hauptsächlich bei 16- oder 32-Bit-Architekturen zu beobachten.

Eine sehr häufig verwendete Architektur für signalverarbeitende Aufgaben ist der ARM7TDMI [Arm01]. Er wird in verschiedensten Bereichen der digitalen Signalverarbeitung verwendet, so z.B. als Audio-Decoder (Cirrus, EP7309), für Kryptographie (Rainbow Microtronx, MKY-85) oder als CDMA-Basisbandprozessor (LSI Logic, CBP3.0). Weitere Anwendungsgebiete werden in [Arm02] erläutert.

Die Hauptprozessoreinheit des ARM7 ist in Abb. 2 dargestellt. Seinen Kern bildet die 32-Bit-ALU. Insgesamt 31 Datenregister mit einer Wortbreite von 32 Bit und 6 Statusregister bilden die Registerbank des Prozessors. Maximal 16 der 31 Register sind abhängig vom aktuellen Modus (Benutzermodus, Supervisor-Modus, IRQ-Modus etc.) gleichzeitig adressierbar, die restlichen sind für die aktuelle Instruktion nicht nutzbar. Der ARM7 verfügt über einen 32 Bit breiten Adress- sowie Datenbus.

Der Instruktionssatz basiert auf den Prinzipien einer RISC[15]-Architektur, ist dieser Definition aber insbesondere aufgrund zusätzlicher signalverarbeitender

[15] Reduced Instruction Set Computer

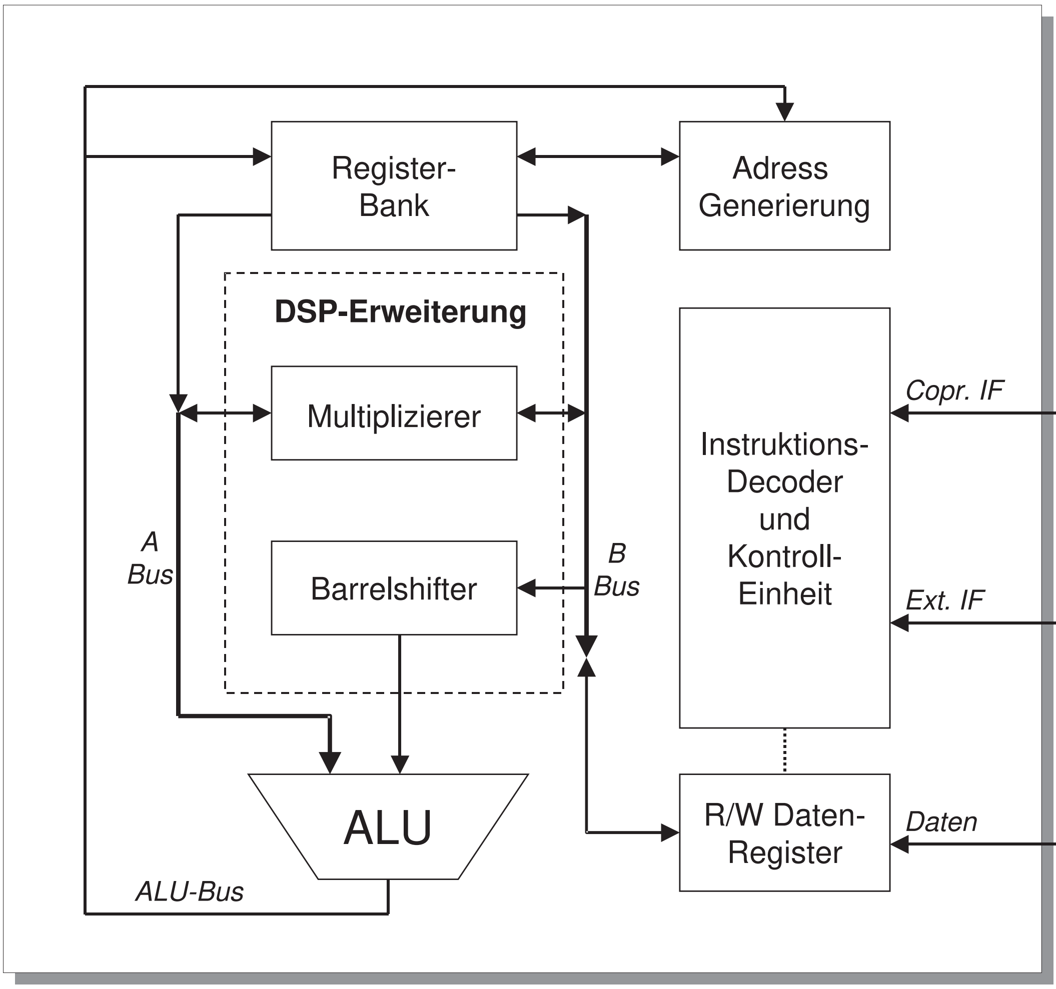


Abbildung 2: Der ARM7 Hauptprozessor.

Aufgaben entwachsen [Arm03].

Instruktionen werden beim ARM7 in drei Pipelinestufen (Fetch, Decode, Execute) abgearbeitet, was einen erhöhten Datendurchsatz gegenüber einer ungepipelineten Architektur bewirkt. Das Zurückschreiben der Daten in die Register wird hierbei noch in der Execute-Stufe veranlasst. Im Gegensatz zum C25 handelt es sich beim ARM7 um eine von-Neumann-Architektur, sodass der Datenbus sowohl Daten als auch Instruktionen führt (vgl. Abb. 2).

Aufgrund seiner Multipliziererenheit und der kombinatorischen Schiebelogik, die im Mittelteil von Abb. 2 dargestellt sind, eignet sich der Controller besonders für die Anforderungen der digitalen Signalverarbeitung. Die Multiplizierereinheit vermag es, 32x8 Multiplikationen in einem Execute-Zyklus zu berechnen. Sollte der Multiplikant größer als 8 Bit sein, so dauert die Operation entsprechend länger. Das Ergebnis der Multiplikation ist wahlweise 32 oder 64 Bit lang. Besonders wertvoll für digitale Filter oder Vektormultiplikationen sind die integrierten Multiply-Accumulate- (MAC) Instruktionen.

Desweiteren verleiht der interne Barrelshifter dem Controller zusätzliche Möglichkeiten, da beliebige Rechts- und Linksshifts erlaubt sind und innerhalb weniger Taktzyklen die Normalisierung einer Fließkommazahl vorgenommen werden kann.

Der ARM7 stellt eine Coprozessorschnittstelle zur Verfügung, mit der beliebige Beschleunigereinheiten mit dem Prozessor verknüpft werden können. Dadurch lässt sich der Datendurchsatz unterschiedlichster Signalverarbeitungsaufgaben gezielt erhöhen. Kapitel 5.1.1 geht genauer auf die Schnittstelle zwischen ARM7-Prozessor und -Coprozessor ein.

Der M68HC16 von Motorola besitzt ebenfalls eine MAC-Erweiterung [Mot97]. Im Gegensatz zum ARM verfügt er lediglich über einen 16 Bit breiten Datenbus und einen 20 Bit breiten Adressbus. Zusätzlich zu den beiden 16-Bit-Standardakkus, welche die Registerbasis der Akkumulatorarchitektur darstellen, zeichnet sich der 68HC16 durch zwei 16-Bit-Faktorregister und einen 36 Bit breiten Akkumulator aus. Dieser sichert u.a. das Gesamtergebnis einer Multiplikation oder eines MAC-Befehls.
Aufgrund der Möglichkeit der MAC-Einheit, über die Indexregister des Controllers Einfluss auf die Speicheradressierung zu nehmen, können mit der RMAC- (Repeated MAC) Instruktion nacheinander mehrere MAC-Operationen mit unterschiedlichen Faktoren durchgeführt werden. Dabei lädt der Prozessor die Faktoren mit Hilfe der Indexregister automatisch nacheinander aus dem Speicher.

2.2.3 Coprozessor

Eine weitere gängige Möglichkeit zur Beschleunigung dedizierter Tasks ist die Aufrüstung eines Controllers mit einem Coprozessor. Der Vorteil von Coprozessoren besteht darin, dass sie Standardcontroller um die erforderliche Zusatzfunktionalität erweitern können. Derselbe Controller kann durch Coprozessor-Erweiterungen für verschiedenste Aufgaben der Signalverarbeitung optimiert werden.

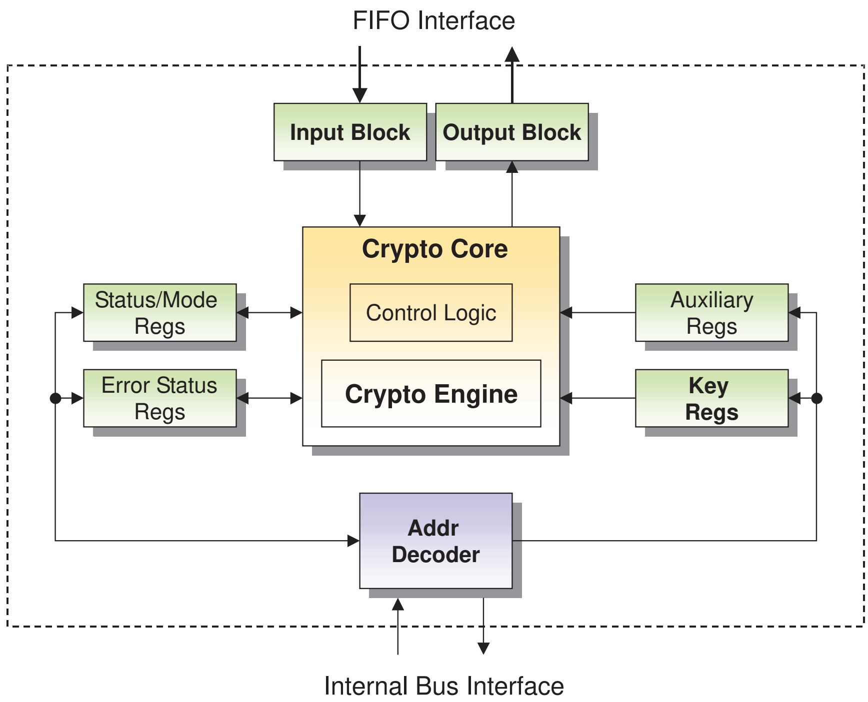


Abbildung 3: Blockschaltbild des Coprozessors für symmetrische Verschlüsselung des Motorola Coldfire.

Ein Beispiel für eine integrierte Prozessoreinheit sowie für einen Coprozessor wurde bereits in dem letzten Unterpunkt behandelt: Der ARM7 verfügt sowohl über eine erweiternde Prozessoreinheit, nämlich den Multiplizierer und Barrelshifter (s. Abb. 2), als auch über eine Coprozessorschnittstelle. Mit speziellen Instruktionen können MAC- oder Coprozessoroperationen durchgeführt werden. Wegen seiner übersichtlichen und flexiblen Schnittstelle existieren bereits mehrere Coprozessorimplementierungen für den ARM7 (z.B. die kryptographischen Coprozessoren von Mykotronx und Securealink).

Desweiteren ist der ColdFire Prozessor MCF5235 von Motorola mit einer Memory-Mapped-Coprozessoreinheit erschienen, die zur Beschleunigung der Datenverschlüsselung beiträgt [Mot04]. Dieser ist u.a. mit einer AES-Einheit ausgestattet, die das Ver- und Entschlüsseln von Nachrichten mit einem 128-Bit-Schlüssel zulässt.

Abb. 3 gibt das Blockschaltbild des Coprozessors wieder. Auf die Register kann über Speicheradressen zugegriffen werden. Vor der Verschlüsselung lädt der Coprozessor die Daten über ein im Adressraum liegendes FIFO[16]-Interface. Nach der Chiffrierung können die Daten nacheinander über die FIFO-Schnittstelle aus dem Coprozessor heraustransportiert werden. Sämtliche Kontroll- und Statusregister liegen auf Speicheradressen. Der Adressdecoder berechnet, auf welche Register zugegriffen wird. Der MCF5235 ist durch zahlreiche Zusatzfunktionen recht unübersichtlich und flächenaufwendig. Unterpunkt 5.2 schlägt eine deutlich effizientere Memory-Mapped-Schnittstelle vor.

Weitere AES-Coprozessoren werden in [Amp04], [Cast04], [Hel01] und [Tri01] erläutert. Eine genauere Untersuchung findet in Unterpunkt 8.3.1 im Rahmen des Ergebnisvergleichs statt.

2.3 Optimierungsmöglichkeiten

Eine optimale Hardwareumsetzung und die Auswahl einer Architektur definieren sich über das selektierte Optimierungskriterium. Folgende Möglichkeiten zur Optimierung digitaler Schaltungen spielen eine wichtige Rolle:

- Kosten,
- Gatterminimierung,
- Minimierung der Chipfläche,
- Entwicklungszeit,
- Leistungsverbrauch,
- Taktfrequenz,
- Durchsatz und
- Wiederverwertbarkeit.

Einige Kriterien sind widersprüchlich zueinander, sodass nicht alle in gleichem Maße verfolgt werden können.

Während das Kostenkriterium sowohl die Gatter- bzw. Flächenminimierung als auch eine kurze Entwicklungszeit beinhaltet, steht ein geringer Flächenbedarf

[16] First-In-First-Out

häufig konträr zu einer kurzen Entwicklungszeit: Gatterminimierte Umsetzungen benötigen eine detaillierte Auseinandersetzung mit dem zu realisierenden Algorithmus. Desweiteren sollte der Algorithmus auf einer recht niedrigen Abstraktionsebene implementiert werden, was gemeinhin für eine längere Entwicklungszeit verantwortlich ist. Allerdings lässt sich hiermit sehr viel gezielter Einfluss auf die später entstehende Gatternetzliste und somit die digitale Schaltung nehmen.
Teilweise werden stark flächenoptimierte Schaltungen auf Transistorebene mit Full-Custom-Designs[17] gelöst. Der hierbei entstehende Arbeitsaufwand rechtfertigt sich jedoch nur bei einer sehr hohen Stückzahl des Chips, um die Fixkosten des Entwicklungsaufwands zu rechtfertigen.

Eine hohe Taktfrequenz ist einerseits sehr gut vereinbar mit einem effektiven Datendurchsatz, doch die gesteigerte Geschwindigkeit hat häufig den Nachteil, dass sie größere Gatteranzahl und damit mehr Siliziumfläche sowie einen erhöhten Leistungsverbrauch zur Folge hat.

Geringerer Leistungsverbrauch lässt sich in erster Näherung mit der Verminderung der Gatterzahl erreichen. Er ist jedoch auch sehr stark von weiteren Parametern wie z.B. der Versorgungsspannung des Chips abhängig. Desweiteren kann durch gezielte designtechnische Maßnahmen, z.B. die Abschaltung nicht genutzter Schaltungsbereiche oder die Reduktion von Schaltungsaktivität, die Leistungsaufnahme eines Chips verringert werden [Ste03]. Letztere Maßnahmen können jedoch eine höhere Gatterzahl bewirken.

Es wird klar, dass die einzelnen Optimierungskriterien stark miteinander korrelieren und dabei häufig kollidieren. Aus diesem Grund sollte die Auswahl des Kriteriums immer auf die Anforderungen an die zu entwickelnden Schaltungen abgestimmt sein und die verschiedenen Optimierungskriterien mit unterschiedlicher Priorität verfolgt werden.

2.4 Eingrenzung und Zielsetzungen

Wie zu Beginn des Kapitels erläutert, wird die hardwareoptimale Implementierung des Datenverschlüsselungs-Algorithmus AES und einer MAC-Einheit aufgrund ihrer großen algorithmischen Unterschiede untersucht. Um aus den in

[17] voll-kundenspezifischer Entwurf

2.2 vorgestellten Architekturen eine passende zu selektieren, sei jedoch zuerst ein Anwendungsfeld definiert, um daraus folgend die Priorisierung der Kriterien vornehmen zu können.

2.4.1 Auswahl eines Kriteriums

In der Literatur werden zunehmend Architekturen vorgestellt, die einen sehr hohen Datendurchsatz erzielen. Dadurch ist jedoch auch die Gatterzahl solcher Architekturen sehr groß.

Betrachtet man den Bereich der Datenverschlüsselung, so ist ein hoher Datendurchsatz in vielen Bereichen sicherlich erwünscht. Wird z.B. der gesamte Datenverkehr eines Servers verschlüsselt, ist es sinnvoll, wenn der Verschlüsselungsalgorithmus mehrere 100 MBit/s chiffrieren kann.

In vielen Bereichen ist jedoch ein weitaus geringerer Datendurchsatz erforderlich. Smart Cards arbeiten z.B. häufig mit den Schnittstellenformaten ISO 14443 Typ A/B oder FeliCa, deren maximale Übertragungsraten bei 424 kBaud bzw. 211 kBaud liegen [Vol03]. Da der Bottleneck hier bereits im Bereich der Datenübertragung liegt, ist es wenig sinnvoll, den Chiffrieralgorithmus hinsichtlich des Durchsatzes zu optimieren.

Smart Cards erreichen jedoch sehr hohe Stückzahlen,[18] daher ist an dieser Stelle eine Flächenoptimierung des Chips und damit eine Kostenminimierung unerlässlich. Auch in vielen anderen Bereichen ist die Optimierung der Fläche jener der Geschwindigkeit vorzuziehen (Datenübertragung im Automobil, Debuginterface mit unsicherem Datenkanal, etc.).

Die vorliegende Arbeit stellt die Gatterminimierung eines Chips und daraus resultierend die Kosteneffizienz in den Vordergrund. Es sollen Hardwaresysteme erörtert werden, die minimale Gatterzahl und mittleren Datendurchsatz für signalverarbeitende Spezialanforderungen verbinden und dabei die Möglichkeit zulassen, mit geringer Geschwindigkeit zeitunkritische Aufgaben zu bearbeiten.

[18] Smart-Card-Hersteller Gemplus verkaufte in der ersten Jahreshälfte 2003 alleine 77,7 Mio. SIMs für Mobiltelefone [Card03].

2.4.2 Auswahl einer Architektur

Da eine geringe Fläche als priorisiertes Kriterium ausgewählt wurde, stellt ein DSP aufgrund seiner üblicherweise großen Busbreite keine optimale Lösung dar. Desweiteren arbeiten DSP wie oben beschrieben vorwiegend mit Harvard-Architekturen, was den Flächenbedarf des Chips erhöht.

Eine Softwarelösung auf einem schnellen Mikrocontroller könnte die Anforderungen der Signalverarbeitung hinreichend erfüllen. Allerdings wäre ein solcher Controller für die übrigen anfallenden Aufgaben wie Messdatenauswertung oder die Erzeugung von Kontrollsequenzen überdimensioniert: Ein Controller, der die signalverarbeitenden Aufgaben mit der erforderlichen Datenrate durchführen kann, muss oft größer und schneller sein, als dies die restlichen nicht-signalverarbeitenden Aufgaben rechtfertigen würden. Ein kleiner und langsamer Prozessor hingegen erfüllt seine regulären Kontrollaufgaben mit einer angemessenen Geschwindigkeit, ist jedoch zu langsam für die Signalverarbeitungsanwendungen.

Eine optimale Lösung für das beschriebene Problem kann ein kompakter 8-Bit-Controllerkern mit austauschbaren signalverarbeitenden Coprozessoren sein. Durch die resultierende geringe Fläche und ein modulares Coprozessorkonzept kann ein solcher Controller hervorragend in eingebettete Systeme integriert werden. Vor allem wegen ihrer geringen Kosten haben 8-Bit-Cores mit 43,8 % immer noch den größten Anteil auf dem Controllermarkt [Stl04]. Einige der Controller sind mitunter auch für den Einsatz unter extremen Randbedingungen (Vibration, elektromagnetische Störungen, etc.) entwickelt und können somit neue Einsatzgebiete erschließen [Ler90].

Als Controllerkern wird die 8-Bit-Architektur des IMS3311C ausgewählt, ein zum Motorola MC68HC11 instruktionssatzkompatibler Prozessor. Dieser sei im Rahmen dieser Arbeit um das kryptographische Coprozessormodul CPCIPH[19] und die MAC-Einheit CPMAC[20] erweitert. Kapitel 4 thematisiert die Architektur des 3311C und seiner Coprozessoren.

[19] CoProcessor for AES Ciphering

[20] CoProcessor for Multiply-Accumulate Operations

2.4.3 Definition der Ziele

Es werden Untersuchungen hinsichtlich der hardwareoptimalen Umsetzung eines kryptographischen Algorithmus und einer MAC-Einheit durchgeführt. Im Blickpunkt steht dabei die Umsetzung als Coprozessor für einen kompakten Mikrocontroller. Die Ziele der Arbeit lassen sich wie folgt zusammenfassen:

- Definition einer einheitlichen Coprozessorarchitektur für verschiedenste Algorithmen,
- Integration in den Controllerkern IMS3311C durch Vergleich unterschiedlicher Schnittstellen,
- Analyse der Flächenoptimierung für dedizierte Coprozessoren,
- Entwicklung und Vergleich verschiedener Umsetzungen der AES-Einheit CPCIPH durch Nutzung der vorgestellten Interfaces und Optimierungsmethoden,
- Verdeutlichung der universellen Nutzbarkeit der Schnittstellen und der Architektur durch Implementierung der MAC-Einheit CPMAC,
- Optimierung der Einheiten für einen ASIC[21],
- Verifikation und Test der Systeme durch Simulation und Emulation auf einem FPGA[22]-Evaluationsboard und
- Vergleich und Auswertung der Syntheseergebnisse.

Insbesondere sei zu beachten, dass die Kernzellfläche eines Coprozessors die des Controllers nicht überschreiten soll.

[21] Application Specific Integrated Circuit

[22] Field Programmable Gate Array

3 Der AES Kryptoalgorithmus

Nachdem im letzten Kapitel der kryptographische Algorithmus AES zur Demonstration der hier entwickelten Designmethodik ausgewählt wurde, befasst sich dieses Kapitel mit dessen mathematischen und strukturellen Grundlagen. Zu Beginn werden notwendige Begriffe und mathematische Voraussetzungen erläutert.[23]

Dabei bildet die Einführung in die Theorie der endlichen Körper (hier auch Galois-Felder) eine Grundlage für das Verständnis der AES-Operationen. Nach der Strukturanalyse des AES werden die Einzelschritte des Algorithmus eingeführt, die zum Teil auf Galois-Feld-Arithmetik basieren.

3.1 Grundlegende Begriffe und Abgrenzung

Bei dem symmetrischen Verschlüsselungsverfahren AES spricht man von einem *Blockchiffre*. Dieser unterscheidet sich von einem *Stromchiffre* dadurch, dass Blöcke mit einer festen Größe n_b ver- bzw. entschlüsselt werden (s. Abb. 4).

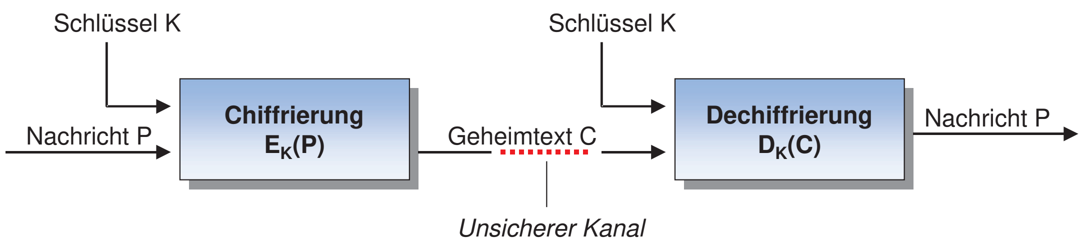


Abbildung 4: Blockchiffrierung mit dem AES.

Der zu verschlüsselnde Text ist mit *Nachricht* oder *Klartext P* (engl. plaintext) bezeichnet. Sie wird mit der AES-*Chiffrierung* $E_K(P)$ (engl. encryption) verschlüsselt, wobei der Index K den symmetrischen Schlüssel darstellt. Die dabei entstehende verschlüsselte Nachricht ist der *Chiffre-* oder *Geheimtext* C (ciphertext).

Die Übertragung des Geheimtextes durch den Sender geschieht über einen unsicheren Kanal, bevor er vom Empfänger entgegengenommen werden kann. Der

[23] Auf mathematische Beweise und Herleitungen wird hierbei verzichtet, diese können bei Bedarf in der entsprechenden Literatur nachvollzogen werden ([Lid83], [McE87]).

Kanal wird als unsicher bezeichnet, da er durch Dritte abgehört werden kann. Nach der *Dechiffrierung* $D_K(C)$ (decryption) wird die ursprüngliche Nachricht

$$P = D_K(E_K(P))$$

wiederhergestellt.

Der Standard gestattet die Umsetzung des Algorithmus mit den Schlüssellängen 128, 192 und 256 Bit. Zur Erläuterung der Methodik werden in dieser Arbeit lediglich Schlüssel mit einer Länge von 128 Bit betrachtet. Dies ist nach dem FIPS Standard [Fips01] zulässig. Nach dem heutigem Stand der Technik ist die symmetrische Verschlüsselung mit einer solchen Schlüssellänge weitestgehend sicher gegen kryptoanalytische Attacken. Sollte sich an dieser Einschätzung etwas ändern, so ist die Ausweitung einer Hardwareumsetzung auf Schlüssellängen von 192 oder 256 Bit mit geringem Aufwand durchführbar.

3.2 Mathematische Grundlagen

Die mathematische Grundlage des AES bildet der endliche Körper $\mathcal{GF}(2^8)$. Das folgende Kapitel führt über die Definition einer Gruppe und eines Rings an die des Körpers und schließlich an endliche Körper heran und thematisiert die grundlegenden Rechenregeln wie Addition oder Multiplikation im $\mathcal{GF}(2^8)$. Schließlich werden komplexe Operationen wie die multiplikative Inversion im $\mathcal{GF}(2^8)$ erläutert. Ein Basisverständnis von Galois-Feldern und der Grund ihres Einsatzes sind für das Verständnis des AES und seiner hardwareoptimalen Umsetzung in den späteren Kapiteln relevant.

3.2.1 Körper

Der Begriff des endlichen Körpers leitet sich aus der Definition der kommutativen Gruppe und des Rings her. Die in diesem Unterpunkt folgenden Definitionen orientieren sich an [Dae02] und [Lid83] .

Definition der kommutativen Gruppe. Eine kommutative (Abelsche) Gruppe $(\mathbb{M}_G, +)$ besteht aus einer Menge $\mathbb{M}_G$ und *einer* auf deren Elementen definierten Operation, von hier an mit '$+$'bezeichnet (vgl. [Dae02]).

Dabei sind im folgenden a, b und c Elemente von $\mathbb{M}_G$, sodass gilt: $a, b, c \in \mathbb{M}_G$. Mit e_+ oder 0 wird das neutrale Element bezeichnet. Die Abelsche Gruppe zeichnet sich durch folgende Eigenschaften aus:

- Kommutativität: $a + b = b + a$,
- Assoziativität: $(a + b) + c = a + (b + c)$,
- Geschlossenheit: $a + b \in \mathbb{M}_G$,
- Inversion: $\forall\, a \;\exists\, b,$ sodass $a + b = a + (-a) = e_+ = 0$,
- Neutrales Element: $\exists\, e_+ \in \mathbb{M}_G,$ sodass $a + e_+ = a + 0 = a$.

Ein einfaches Beispiel für eine Abelsche Gruppe ist die Menge der ganzen Zahlen $\mathbb{Z}$ in Kombination mit dem Additions-Operator '$+$'. $(\mathbb{Z}, +)$ ist kommutativ, assoziativ, geschlossen, besitzt zu jedem Element ein inverses Element in der Addition und verfügt über das neutrale Element 0.

Definition des Rings. Ein Ring $(\mathbb{M}_R, +, \circ)$ besteht aus einer Menge $\mathbb{M}_R$, für die *zwei* Operationen definiert sind. Diese werden hier als '$+$' und '$\circ$' bezeichnet (vgl. [Dae02]).

Zusätzlich zu der Abelschen Gruppe $(\mathbb{M}_R, +)$ hat der Ring somit noch eine zweite Operation '$\circ$'. Hier wird mit $e_\circ$ oder 1 das neutrale Element bezeichnet. Für die neue Operation gilt mit $a, b, c \in \mathbb{M}_R$:

- Geschlossenheit: $a \circ b \in \mathbb{M}_R$,
- Neutrales Element: $\exists\, e_\circ \in \mathbb{M}_R,$ sodass $a \circ e_\circ = a \circ 1 = a$,
- Distributivität: $(a + b) \circ c = (a \circ c) + (b \circ c)$.

Zudem ist '$\circ$' assoziativ über $\mathbb{M}_R$. Die Menge $\mathbb{Z}$ ist dementsprechend auch ein Ring, da für die Operation '$\circ$' das Gesetz der Geschlossenheit, der Assoziativität, der Distributivität und des neutralen Elements 1 gilt.

Definition eines Körpers. Ein Ring nennt sich Körper $(M_K, +, \circ)$, falls die Operation '$\circ$' kommutativ ist und zu jedem Element a (ausser '0') ein inverses Element a^{-1} bezüglich '$\circ$' existiert.
Die Menge der ganzen Zahlen $\mathbb{Z}$ ist demnach kein Körper, da es nicht zu jedem Element ein inverses gibt, sodass $a \circ a^{-1} = 1$. Hingegen ist die Menge der rationalen Zahlen $\mathbb{Q}$ ein Körper, da es zu jedem Element von $\mathbb{Q}$ ein inverses Element gibt. Jeder Körper ist *nullteilerfrei*. Somit gilt $a \circ b \neq 0$, falls weder a noch b gleich 0 sind.

3.2.2 Endliche Körper

Unter einem *endlichen Körper* versteht man einen Körper, der eine endliche Anzahl Elemente besitzt. Somit ist die Menge $\mathbb{Q}$ kein endlicher Körper.

Unter der *Ordnung* σ eines Körpers versteht man die Anzahl der Elemente im Körper. Ein Theorem [McE87] setzt für die Existenz eines Körpers voraus, dass seine Ordnung gleich einer Primzahl p in der n-ten Potenz ist, also $\sigma = p^n$. Dabei ist die Potenz n eine beliebige ganze Zahl, während p die *Charakteristik* des Körpers bezeichnet.

Für den AES-Algorithmus werden Körper der Ordnung $\sigma = 2^8 = 256$ verwendet. Also ist $p = 2$ die Charakteristik des Körpers und $n = 8$ die Potenz. Man nennt diesen Körper Galois-Feld mit der Ordnung 256 oder kurz $\mathcal{GF}(2^8)$. Das Galois-Feld $\mathcal{GF}(2^8)$ ist eindeutig, allerdings gibt es homomorphe und isomorphe Körper derselben Ordnung, sodass die Elemente des einen Körpers eindeutig auf die des anderen abgebildet werden können. In solchen Körpern unterscheidet sich lediglich die Darstellungform der Elemente. Dies wird in einem späteren Unterpunkt wichtig, wenn zur hardwareoptimalen Berechnung der Inversionsoperation des AES kurzzeitig die Darstellungsform vom $\mathcal{GF}(2^8)$ in den Körper $\mathcal{GF}((2^4)^2)$ gewechselt wird.

Die Auswahl des $\mathcal{GF}(2^8)$ basiert auf der besonders effizienten Umsetzbarkeit für digitale Schaltungen. Vorteil der Verwendung eines Galois-Felds mit der Charakteristik 2 und der Potenz 8 beim AES ist die dadurch ermöglichte parallele Bearbeitung eines Bytes. Da die einzelnen Operationen invertierbar sein müssen, wird die Inversionseigenschaft des Körpers benötigt. Die multiplikative Inversion ist von Bedeutung, da sie durch ihre Nichtlinearität die Sicherheit des Algorithmus entscheidend mitbegründet.

Im folgenden werden die einzelnen Operationen erläutert, da diese von den gewöhnlichen Operationen in der Menge $\mathbb{Z}$ oder $\mathbb{Q}$ abweichen.

3.2.3 Darstellungsform und Grundoperationen im $\mathcal{GF}(2^8)$

Das $\mathcal{GF}(2^8)$ wird in der Literatur zumeist in Polynomdarstellung [Dae02] repräsentiert. Damit erhält man das Polynom:

$$a(x) = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0, \tag{3.1}$$

Es gibt also acht Koeffizienten a_i, bei denen jeder den Wert 0 oder 1 annehmen kann (da $p = 2$). Somit existieren 256 verschiedene Polynome im $\mathcal{GF}(2^8)$.

An dieser Stelle erfolgt der Brückenschlag zur Hardware: Die 256 verschiedenen Polynome lassen sich mit acht Bit darstellen. Jeder Koeffizient stellt ein Bit dar, wobei a_7 das MSBit[24] und a_0 das LSBit[25] sind. Die Wertigkeit des Bits a_i wird durch x angegeben. Somit entspricht z.B. das Polynom

$$\begin{aligned} &0 \cdot x^7 + 1 \cdot x^6 + 1 \cdot x^5 + 0 \cdot x^4 + 1 \cdot x^3 + 0 \cdot x^2 + 1 \cdot x + 0 \\ &= x^6 + x^5 + x^3 + x \end{aligned}$$

dem Byte `%01101010` [26] bzw. `$6A`. Die *Addition* zweier Bytes bzw. zweier Polynome lässt sich darstellen als die Addition der Koeffizienten gleicher x^i *modulo* 2. So ergibt die Additon der Polynome

$$\begin{aligned} &(x^6 + x^5 + x^3 + x) + (x^7 + x^5 + x^2 + x) \\ = \;\; &x^7 + x^6 + (1 \oplus 1)x^5 + x^3 + x^2 + x(1 \oplus 1) \\ = \;\; &x^7 + x^6 + x^3 + x^2, \end{aligned}$$

oder auch `$6A` + `$A6` = `$CC`. Diese Operation stellt eine bitweise XOR-Verknüpfung[27] dar. Die Addition im $\mathcal{GF}(2^8)$ wird in mehreren Einzelschritten des AES benötigt.

---

[24] Most Significant Bit

[25] Least Significant Bit

[26] Falls nicht eindeutig aus dem Kontext hervorgehend, werden Binärzahlen durch das Präfix '`%`' und Hexadezimalzahlen durch das Präfix '`$`' bezeichnet.

[27] Das Symbol $\oplus$ wird hier für eine XOR-Verknüpfung von zwei einzelnen Bits genutzt. Dies ist nicht zu verwechseln mit dem $+$ innerhalb der Polynome. Diese Summenzeichen werden zur Elementrepräsentation oder bei einer $\mathcal{GF}$-Addition von Vektoren benutzt.

Die *Multiplikation* zweier Polynome könnte zu einem Ergebnispolynom mit einem Grad größer als 7 führen. Aufgrund der oben beschriebenen Eigenschaften eines endlichen Körpers ist dies nicht erlaubt. Aus diesem Grund muss ein Reduktionspolynom eingeführt werden, das eine Modulo-Operation auf das Multiplikationsergebnis durchführt. Die Forderung an dieses Polynom ist, dass es *irreduzibel* ist. Das bedeutet, dass es über $\mathcal{GF}(2^8)$ nur durch 1 und sich selber teilbar sein darf. Für den AES-Algorithmus wird das Polynom

$$R(x) = x^8 + x^4 + x^3 + x + 1$$

spezifiziert. Ist also nach der Multiplikation der größte Exponent von x größer als 7, so muss eine Polynomdivision durch $R(x)$ vorgenommen werden. Der verbleibende Rest nach der Division ist das Ergebnis der Multiplikation. Als Beispiel soll hier die Multiplikation von `$6A` mit `$18` dienen:

$$\begin{aligned}
&(x^6 + x^5 + x^3 + x) \cdot (x^4 + x^3) \\
=\ & x^{10} + (x^9 \oplus x^9) + x^8 + x^7 + x^6 + x^5 + x^4 \\
=\ & x^{10} + x^8 + x^7 + x^6 + x^5 + x^4 \\
=\ & (x^2 + 1) \cdot (x^8 + x^4 + x^3 + x + 1) \ + \ (x^7 + x^2 + x + 1) \\
\equiv\ & (x^7 + x^2 + x + 1) \ \ (mod(x^8 + x^4 + x^3 + x + 1))
\end{aligned}$$

Der AES-Algorithmus verwendet die Multiplikation im $\mathcal{GF}(2^8)$ bei der `MixColumns`- und der `InvMixColumns`-Operation (s. Unterpunkt 3.3.2).

3.2.4 Inversion im $\mathcal{GF}((2^4)^2)$

In Sektion 3.2.1 definiert sich ein Körper u.a. über seine Invertierbarkeit sowohl in Addition als auch in Multiplikation.

Die Ermittlung der *additiven Inversen* ist dabei trivial: Die Addition zweier Polynome ist eine einfache XOR-Verknüpfung ihrer Koeffizienten mit gleicher Wertigkeit. Damit ergibt die Addition eines Polynoms mit sich selbst Null.

Die *multiplikative Inversion* im $\mathcal{GF}(2^8)$ gestaltet sich komplexer. Sie wird oftmals mittels des Erweiterten Euklidschen Algorithmus berechnet.[28] Für die Hardwareimplementierung bietet sich dieser Algorithmus allerdings nicht an, da

[28] vgl. [Schn96], S.289 sowie [Ert01]

die von ihm benötigte Gatter- und Taktzyklenzahl eine effiziente Hardwareumsetzung ausschließt. Zumeist wird sie durch eine 256 Byte große Lookup-Table (LUT) realisiert, welche zu jedem Polynom das inverse bereithält.

Führt man sich vor Augen, dass zu dem Körper des $\mathcal{GF}(2^8)$ homomorphe bzw. isomorphe Körper existieren, so ist es denkbar, Operationen wie die multiplikative Inversion in einem zum $\mathcal{GF}(2^8)$ arithmetisch äquivalenten Körper $\mathcal{GF}((2^4)^2)$ durchzuführen. Es ist möglich, dass der Hardwareaufwand der Inversion in einem solchen zusammengesetzten Körper (engl. Composite Field) sinkt.[29]

Zur Erklärung des Aufbaus des Composite Fields $\mathcal{GF}((2^4)^2)$ sei zunächst dessen *Unterkörper* $\mathcal{GF}(2^4)$ betrachtet. Ein endlicher Körper im $\mathcal{GF}(2^4)$ hat folgendes Erscheinungsbild:

$$b_3 \cdot y^3 + b_2 \cdot y^2 + b_1 \cdot y + b_0,\ b_i \in \mathcal{GF}(2). \tag{3.2}$$

Die Koeffizienten b_i können dementsprechend die Werte 0 oder 1 annehmen. Der Körper $\mathcal{GF}(2^4)$ braucht nun ein Reduktionspolynom, welches zur Erfüllung der Endlichkeitsbedingung benutzt wird. Das hier verwendete Polynom ist

$$S(y) = y^4 + y + 1. \tag{3.3}$$

Multiplikationen im $\mathcal{GF}(2^4)$ werden folglich modulo $S(y)$ durchgeführt.

Dem Unterkörper $\mathcal{GF}(2^4)$ ist das Composite Field $\mathcal{GF}((2^4)^2)$ übergeordnet:

$$c_1 \cdot z + c_0,\ c_i \in \mathcal{GF}(2^4), \tag{3.4}$$

Für sein Reduktionspolynom wird

$$T(z) = z^2 + z + \omega^{14},\ \omega^{14} \in \mathcal{GF}(2^4), \tag{3.5}$$

selektiert [Pa94]. Der konstante Anteil ω^{14} ist eine verkürzte Darstellung für ein Polynom im $\mathcal{GF}(2^4)$, die durchgehend in der Literatur genutzt wird. Dieses entspricht hier $y^{14}\ mod\ S(y) = y^3 + 1$. Solange damit T(z) irreduzibel bleibt, darf ω^{14} beliebig ersetzt werden.

Für die Umwandlung vom $\mathcal{GF}(2^8)$ in den Körper $\mathcal{GF}((2^4)^2)$ muss eine Transformation zwischen den Darstellungsformen durchgeführt werden. 8x8 Matrizen

[29] Mit der Umwandlung in Composite Fields setzt sich insbesondere [Pa94] auseinander.

überführen ein Element eindeutig aus dem einen Körper in das andere und zurück.

Bestimmen der Transformationsmatrizen. Es wird nun ein Element $\alpha \in \mathcal{GF}(2^8)$ bestimmt, das die Bedingung erfüllt:

$$R(\alpha) = 0 \; (mod \; R(x)). \tag{3.6}$$

Um zu garantieren, dass die Körper $\mathcal{GF}(2^8)$ und $\mathcal{GF}((2^4)^2)$ zumindest homomorph bezüglich der Multiplikation $mod \; R(x)$ sind, muss die Bedingung [Pa94]

$$R(\beta^e) = 0 \; (mod \; S(y), T(z)), \;\; \beta^e \in \mathcal{GF}((2^4)^2), \tag{3.7}$$

gelten. Es wird somit ein Element β^e gesucht, das eine direkte Abbildung von α darstellt. Der Exponent e ist dabei eine ganzwertige Zahl zwischen 0 und 254, sodass mit β^e Elemente des $\mathcal{GF}((2^4)^2)$ dargestellt werden können. Gleichzeitig muss gelten, dass $S(\omega) = 0$ und $T(\beta) = 0$.

Ist eine gültige Abbildung von α auf β^e gefunden, so ist es trivial, α^2 auf β^{2e}, α^3 auf β^{3e}, ..., α^7 auf β^{7e} abzubilden.[30]

Die Aufgabe zur Bestimmung der Transformationsmatrix besteht nun darin, nacheinander alle β^e dahingehend zu untersuchen, ob das gefundene β^e die Bedingung $R(\beta^e) = 0$ erfüllt. Dies geschieht durch Polynomdivision von $R(\beta^e)$ mit $S(y)$ und $T(z)$. In [Pa94] lässt sich ein optimierter Algorithmus für diese Berechnung finden, der mit wenigen Schritten die möglichen Abbildungen für α auf β^e berechnet.

Anhand eines Beispiels soll nun der erste Schritt der Berechnung veranschaulicht werden:
Gesetzt wird $e = 1$ und somit $\alpha = \beta^1$. Daraus folgt:

$$\begin{aligned}
&R(\beta^1) = \beta^8 + \beta^4 + \beta^3 + \beta^1 + 0 \\
&= \omega^{14} \cdot \beta + \omega^5 \\
&= (y^3 + 1) \cdot \beta + (y^2 + y) \\
&\neq 0
\end{aligned}$$

[30] Der Exponent e wird *modulo* 255 berechnet, da $\beta^1 = \beta^{256}$, $\beta^2 = \beta^{257}$, usw.

β^1 entspricht also nicht den Anforderungen. Für die Exponenten 2, 3 und 4 gilt dasselbe. Erst β^5 erfüllt die Bedingungen, da für dieses Element gilt:

$$R(\beta^5) = \ 0 \ (mod \ S(y), T(z)).$$

Nun wird eine 8x8 Matrix $\mathbf{M}$ aufgestellt, die Polynome aus dem $\mathcal{GF}(2^8)$ in das $\mathcal{GF}((2^4)^2)$ überführt. Dabei beinhaltet die erste Spalte von $\mathbf{M}$ die entsprechende Darstellung von α^7 im $\mathcal{GF}((2^4)^2)$, die zweite Spalte für α^6, etc. Für die Lösung $\alpha = \beta^5$ bedeutet dies:

$$\begin{aligned}
&\alpha^0 = \beta^0 = 0 \cdot \beta + 1, \\
&\alpha^1 = \beta^5 = \omega^8 \cdot \beta + \omega^{14} = (y^2 + 1) \cdot \beta + (y^3 + 1), \\
&\alpha^2 = \beta^{10} = \omega^2 \cdot \beta + \omega^6 = y \cdot \beta + (y^3 + y^2), \\
&\ldots \\
&\alpha^7 = \beta^{35} = \omega^{13} \cdot \beta = (y^3 + y^2 + 1) \cdot \beta + 0,
\end{aligned}$$

und übertragen auf die Matrix $\mathbf{M}$

$$\mathbf{M} = \begin{bmatrix}
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 1 & 1
\end{bmatrix}$$

Mit der inversen Matrix $\mathbf{M_R}$ soll eine eindeutige Rücktransformation durchgeführt werden. Diese kann z.B. durch das Gaußsche Eliminationsverfahren bestimmt werden [Bro00] und muss bezüglich der Erhaltung der Homomorphiebedingung überprüft werden.

Es existieren insgesamt acht mögliche Transformationen von α auf β^e. Dies sind β^5 und die sogenannten Konjugierten zu β^5. Gültige Abbildungen gibt es für e = 5, 10, 20, 40, 65, 80, 130 und 160.[31] Diese werden später zum Zweck der

[31] Die hier berechnete Matrix für $e = 160$ ist identisch mit der von Rudra et al. in [Rud01] veröffentlichten Matrix, die übrigen in Anhang A aufgeführten Matrizen konnten in Veröffentlichungen noch nicht gesichtet werden.

Flächenoptimierung verglichen. Alle in dieser Arbeit berechneten Matrizen listet Anhang A auf.

An dieser Stelle sei darauf hingewiesen, dass Paar [Pa94] zur Berechnung der Trafomatrizen von der Primitivität der körpererzeugenden Polynome $R(x)$, $S(y)$ und $T(z)$ ausgeht. Primitiv bedeutet hier, dass die Polynome maximale Ordnung besitzen. Für das Polynom $R(x)$ meint maximale Ordnung, dass $x^s - 1$ ($s \in \mathbb{N}$) erst für $s = 2^8 - 1 = 255$ durch $R(x)$ teilbar sein dürfte. Dies trifft auf $R(x)$ *nicht* zu, sodass dieses nicht primitiv, sondern lediglich irreduzibel ist.
Rudra et al. [Rud01] zeigen auf, dass die Operationen im $\mathcal{GF}(2^8)$ und $\mathcal{GF}((2^4)^2)$ arithmetisch äquivalent sind. Für die hier vorliegende Arbeit werden hingegen für alle 256 möglichen Polynome im $\mathcal{GF}(2^8)$ deren multiplikative Inverse durch Inversion im $\mathcal{GF}((2^4)^2)$ bestimmt und mit den in [Dae02] veröffentlichten Polynomen verglichen. Diese lassen sich leicht auf die Polynome aus dem Standard [Fips01] zurückrechnen. Durch die hierdurch belegte Übereinstimmung ist ein Beweis der Isomorphie nicht notwendig.

Inversionsalgorithmus. Nachdem eine Transformation vom $\mathcal{GF}(2^8)$ in das $\mathcal{GF}((2^4)^2)$ ermöglicht wurde, soll in diesem Unterpunkt die Inverse im dortigen Körper berechnet werden.[32]

Es werden nun die Polynome im $\mathcal{GF}((2^4)^2)$

$$C(z) = c_1 \cdot z + c_0;\ c_i \in \mathcal{GF}(2^4),\ i = 0,1 \text{ und}$$
$$D(z) = d_1 \cdot z + d_0;\ d_i \in \mathcal{GF}(2^4),\ i = 0,1$$

definiert. Es gelten weiterhin die Reduktionspolynome der vorangegangen Unterpunkte $S(y)$ und $T(z)$. Ist nun $C(z) = D(z)^{-1}$, dann muss gleichzeitig $C(z) \cdot D(z) = 1$ gelten. Multipliziert man C und D miteinander, so erhält man

$$\begin{aligned} 1 &= (c_1 z + c_0) \cdot (d_1 z + d_0)\ mod\ T(z) \\ &= (c_1 d_1 z^2 + (c_0 d_1 + c_1 d_0) \cdot z + c_0 d_0)\ mod\ T(z) \\ &= (c_1 d_1 + c_0 d_1 + c_1 d_0) \cdot z + (c_0 d_0 + c_1 d_1 \omega^{14}). \end{aligned} \tag{3.8}$$

Daraus lässt sich ableiten, dass der linke Summand in Gl. 3.8 gleich 0 und der rechte Summand gleich 1 sein muss, damit aus der gesamten Gleichung der Wert

[32] Der hier vorgeschlagene Algorithmus stammt von Morii und Kasahara [Mor89]. Desweiteren beschäftigen sich [Ito88], [Pa95], [Rijm] und [PaR97] mit dem Thema.

1 resultiert. Löst man nach d_0 und d_1 auf, dann ergibt sich:

$$\begin{aligned} d_1 &= \quad c_1 \quad \cdot (\omega^{14} \cdot c_1^2 + c_0 \cdot (c_0 + c_1))^{-1} \\ d_0 &= (c_0 + c_1) \cdot (\omega^{14} \cdot c_1^2 + c_0 \cdot (c_0 + c_1))^{-1} \end{aligned} \tag{3.9}$$

Der rechte Faktor beider Gleichungen ist identisch. Bei einer Hardwareimplementierung muss man diesen Teil dementsprechend nur einmal umsetzen.

Die gesamte Inversion im $\mathcal{GF}((2^4)^2)$ reduziert sich auf eine Inversion, eine Konstantenmultiplikation, drei Multiplikationen und zwei Additionen im $\mathcal{GF}(2^4)$. Da das $\mathcal{GF}(2^4)$ nur 16 verschiedene Polynome mit maximalem Grad 3 beinhaltet, lässt sich annehmen, dass sich hier eine Inversion mit sehr viel geringerem Aufwand als im $\mathcal{GF}(2^8)$ ausführen lässt, welches über 256 Polynome mit maximalem Grad 7 verfügt.

Die in dieser Arbeit durchgeführten *Multiplikationen* im $\mathcal{GF}(2^4)$ werden allesamt mit dem Multiplizierer von Mastrovito [Mas91] durchgeführt, da dieser sich für Hardwareumsetzungen durch seine geringe Anzahl an benötigten XOR-Gattern und die unbedingte Berücksichtigung der Modulo-Operation mit T(z) besonders gut eignet.
Multiplikationen lassen sich dementsprechend darstellen durch:

$$\begin{aligned} H(y) &= G(y) \cdot F(y) \; mod \; S(y) \\ &= \begin{bmatrix} h_3 \\ h_2 \\ h_1 \\ h_0 \end{bmatrix} = \begin{bmatrix} (g_3 + g_0) & g_1 & g_2 & g_3 \\ (g_2 + g_3) & (g_3 + g_0) & g_1 & g_2 \\ (g_1 + g_2) & (g_2 + g_3) & (g_3 + g_0) & g_1 \\ g_1 & g_2 & g_3 & g_0 \end{bmatrix} \cdot \begin{bmatrix} f_3 \\ f_2 \\ f_1 \\ f_0 \end{bmatrix}, \end{aligned} \tag{3.10}$$

wobei $f_i, g_i, h_i \in \mathcal{GF}(2)$. Die *Konstantenmultiplikation* mit ω^{14} kann direkt ausgewertet werden. Genau wie die Addition im $\mathcal{GF}(2^8)$ ist die *Addition* im $\mathcal{GF}(2^4)$ durch einfache XOR-Verknüpfung der Koeffizienten gleicher Wertigkeit durchzuführen. Die *Quadratur* kann ebenfalls bereits abstrakt berechnet werden:

$$\begin{aligned} c_1^2(y) &= (c_1(y) \cdot c_1(y)) \; mod \; Q(y) \\ &= c_{13} \cdot y^3 + (c_{13} \oplus c_{11}) \cdot y^2 + c_{12} \cdot y + (c_{12} \oplus c_{10}) \;, \; c_{1i} \in \mathcal{GF}(2). \end{aligned} \tag{3.11}$$

Hiermit sind die mathematischen Grundlagen für eine hardwareoptimierte Implementierung des AES erläutert. Ein Vergleich der acht Transformationsmatrizen unter dem Gesichtspunkt der Flächenoptimierung unter Berücksichtigung

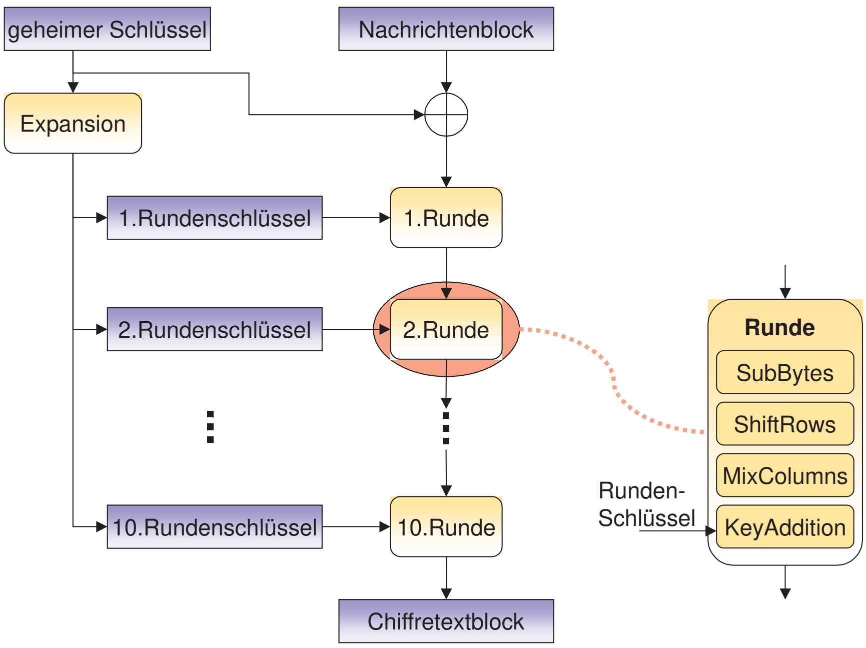


Abbildung 5: Struktur des AES-Verschlüsselungsalgorithmus.

des kritischen Pfads konnte in der Literatur noch nicht gesichtet werden. Die Vorgehensweise hierfür wird in Unterpunkt 7.3.3 entwickelt.

3.3 Grundlagen des AES

Dieser Unterpunkt baut auf den vorgestellten Basisoperationen der vorangegangenen Abschnitte auf und erörtert im folgenden die Funktionsweise des AES. Nach Erläuterung der Grundstruktur werden die Einzelschritte des Algorithmus erklärt. Details zum AES werden im Standard [Fips01] beschrieben.

3.3.1 Struktur

Der AES-Algorithmus wird in mehrere Runden unterteilt, die sich aus verschiedenen Operationen zusammensetzen.

Abhängig von der Länge des Schlüsselwortes und der Größe eines Eingangsblocks besteht das Codierverfahren aus 10, 12 oder 14 Runden. In dieser Arbeit wird lediglich eine Datenblockgröße und Schlüssellänge von 128 Bit betrachtet, sodass sich die Anzahl der Runden auf 10 beläuft.

```
//++++++++++++ ENCRYPTION ++++++++++++//
AddRoundKey(Input, Key);
// 9 Runden.
for(i=1; i<10; i=i+1)
{
   SubBytes(State);
   ShiftRows(State);
   MixColumns(State);
   AddRoundKey(State, ExpKey[i]);
 }
// 10. Runde ohne MixColumns.
SubBytes(State);
ShiftRows(State);
AddRoundKey(State, ExpKey[10]);

//++++++++++++ DECRYPTION ++++++++++++//
AddRoundKey(Input, InvExpKey[0]);
InvShiftRows(State);
InvSubBytes(State);
for(i=1; i<10; i=i+1)
{
   AddRoundKey(State, InvExpKey[i]);
   InvMixColumns(State);
   InvShiftRows(State);
   InvSubBytes(State);
}
AddRoundKey(State, Key);
```

Beispiel 1: Chiffrier- und Dechiffrierstruktur AES.

Abbildung 5 stellt den groben Aufbau der AES-Chiffrierung dar. Jede Runde erhält einen eigenen Teilschlüssel, der aus dem geheimen Schlüsselblock generiert wird. Den gesamten Ablauf des Ver- und Entschlüsselungsvorgangs gibt der Pseudocode in Beispiel 1 wieder. `State` meint dabei einen 128-Bit-Zustandsvektor, während `ExpKey` und `InvExpKey` einen 128-Bit-Rundenschlüssel bezeichnen. Der rechte Kasten in Abb. 5 und die Funktionen in Bsp. 1 führen die Einzelschritte des Algorithmus ein.

Von zehn Runden werden jeweils neun reguläre und eine modifizierte durchgeführt. Die einzige Unregelmäßigkeit im Verschlüsselungsalgorithmus tritt in

der letzten Runde auf, dort wird auf den `MixColumns`-Schritt verzichtet.
Der Dechiffriervorgang verläuft entgegengesetzt. Lediglich die inversen Funktionen der einzelnen Schritte werden in umgekehrter Reihenfolge ausgeführt. Hier entfällt in der ersten Runde die `InvMixColumns`-Operation.
Bei der Verschlüsselung wird vor Beginn der ersten und bei der Entschlüsselung nach der letzten Runde eine Schlüsseladdition mit dem geheimen Schlüssel durchgeführt.

3.3.2 Operationsschritte

Im folgenden werden die einzelnen Schritte einer Runde erklärt und anhand von Abbildungen veranschaulicht. In dieser Arbeit repräsentieren stets Tabellen mit vier Zeilen und vier Spalten die Datenblöcke, wobei ein Eintrag ein Byte des Datenblocks wiederspiegelt.[33]

SubBytes und InvSubBytes. Beim `SubBytes`-Schritt wird eine sogenannte $\mathcal{S}$-Box $\mathcal{S}(a)$ auf einen Eingang angewendet. Die $\mathcal{S}$-Box hat die Aufgabe, einen 8-Bit-Eingang eindeutig auf einen 8-Bit-Ausgang abzubilden (s. Abb. 6). Der `SubBytes`-Schritt ist der einzige nichtlineare Schritt des AES und somit hauptsächlich für die Sicherheit des Algorithmus verantwortlich.

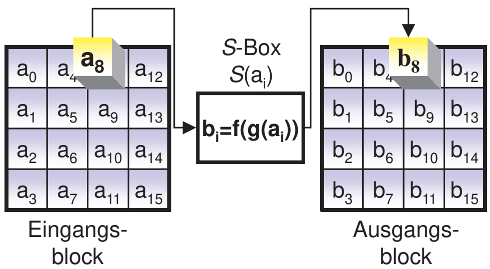


Abbildung 6: Der `SubBytes` Schritt.

Die Funktion zur Berechnung der $\mathcal{S}$-Box kann in zwei Teilfunktion $t = g(a)$ und $\mathcal{S}(a) = b = f(g(a))$ unterteilt werden. Hierbei ist die Funktion $g(a)$ der nichtlineare Teil, welcher für den AES-Algorithmus durch die multiplikative Inverse

$$t = g(a) = a^{-1} \tag{3.12}$$

gewählt wurde. Die Funktion $f(t)$ ist eine affine Transformation des Vektors t

[33] vgl. [Dae02], S.33

und berechnet sich durch

$$b = \mathbf{F}_0 \cdot t + f_0 \tag{3.13}$$

Im Detail ist die affine Transformation wie folgt festgelegt:[34]

$$b = \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} t_7 \\ t_6 \\ t_5 \\ t_4 \\ t_3 \\ t_2 \\ t_1 \\ t_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}. \tag{3.14}$$

Für den Entschlüsselungsvorgang findet die inverse Operation der $\mathcal{S}$-Box $\mathcal{S}_{inv}$ Verwendung. Diese lautet:

$$u = \begin{bmatrix} u_7 \\ u_6 \\ u_5 \\ u_4 \\ u_3 \\ u_2 \\ u_1 \\ u_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} p_7 \\ p_6 \\ p_5 \\ p_4 \\ p_3 \\ p_2 \\ p_1 \\ p_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}. \tag{3.15}$$

Die komplette Umkehrfunktion bestimmt sich zu[35]

$$\mathcal{S}_{inv} = q = g^{-1}(u) = g^{-1}(f^{-1}(p)) = g(f^{-1}(p)). \tag{3.16}$$

`SubBytes` und `InvSubBytes` werden im weiteren Verlauf durch den Begriff *Byte-Substitution* zusammengefasst.

[34] Der Index 7 bezeichnet fortan das MSBit eines Vektors, während der Index 0 das LSBit kennzeichnet.

[35] $g^{-1}(u) = g(u)$, da die Umkehrfunktion der multiplikativen Inversion gleich der Funktion ist.

ShiftRows und InvShiftRows. Während des `ShiftRows`-Schritts werden die Bytes einer Zeile eines Zustandsblocks zyklisch nach links verschoben. Die Anzahl der zu verschiebenden Stellen ist abhängig von der Zeilennummer. Wie in Abbildung 7 dargestellt, verschiebt sich die oberste Zeile nicht, die zweite um ein, die dritte um zwei und die vierte um drei Bytes zyklisch nach links.

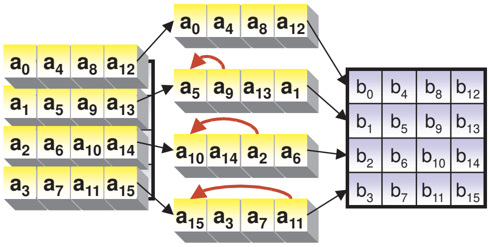


Abbildung 7: Der `ShiftRows` Schritt.

Beim Entschlüsseln kehrt sich die Schieberichtung um. `ShiftRows` hat beim AES die Funktion, die Diffusion zu erhöhen. Aus diesem Grunde wurden die Offsets für jede Zeile unterschiedlich ausgewählt. Ein Ziel dieser Byte-Transposition besteht darin, dass alle 128 Bit des Eingangsworts mit allen 128 Bit des Ausgangsworts korrelieren.

MixColumns und InvMixColumns. Beim `MixColumns`-Schritt werden die Elemente der einzelnen Spalten verändert. Dieser Schritt ist eine sogenannte Bricklayer- (Maurer-) Permutation, d.h. dass sich die Gesamttransformation in mehrere Teile zerlegen lässt.

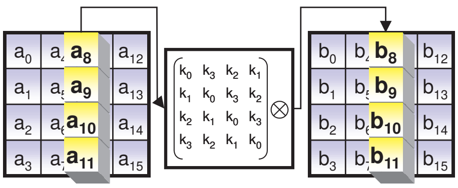


Abbildung 8: Der `MixColumns` Schritt.

Der Spaltenvektor wird als Polynom dritten Grades angesehen. Daraus resultiert für die erste Spalte:

$$a(v) = a_0 + a_1 \cdot v + a_2 \cdot v^2 + a_3 \cdot v^3,\ a_i \in \mathcal{GF}(2^8). \tag{3.17}$$

Zum Erlangen größerer Diffusion erfolgt die Multiplikation dieses Spaltenvektors mit einem konstanten Polynom $k(v)$. Das Ergebnis wird *modulo* $(v^4 + 1)$ berechnet:

$$\begin{aligned} & b = k \cdot a \\ & \Leftrightarrow (b_0 + b_1 v + b_2 v^2 + b_3 v^3) \\ & = (k_0 + k_1 v + k_2 v^2 + k_3 v^3) \times (a_0 + a_1 v + a_2 v^2 + a_3 v^3)(mod(v^4 + 1)) \end{aligned} \tag{3.18}$$

Diese Gleichung kann ausmultipliziert und nach Exponenten von v sortiert werden. Danach wird sie in die folgende Matrixdarstellung für eine Spalte transformiert:

$$b = \begin{bmatrix} b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} k_0 & k_3 & k_2 & k_1 \\ k_1 & k_0 & k_3 & k_2 \\ k_2 & k_1 & k_0 & k_3 \\ k_3 & k_2 & k_1 & k_0 \end{bmatrix} \times \begin{bmatrix} a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} \tag{3.19}$$

Die konstanten Koeffizienten k_i wurden für eine schnelle Hardwareumsetzung möglichst einfach gewählt. Um eine große Diffusion der Eingangswerte zu erzielen, werden drei unterschiedliche Koeffizienten selektiert: `$01`, `$02` und `$03`.

Daraus folgt für Gleichung 3.19:

$$b = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} \texttt{02} & \texttt{03} & \texttt{01} & \texttt{01} \\ \texttt{01} & \texttt{02} & \texttt{03} & \texttt{01} \\ \texttt{01} & \texttt{01} & \texttt{02} & \texttt{03} \\ \texttt{03} & \texttt{01} & \texttt{01} & \texttt{02} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{3.20}$$

Für den `InvMixColumns` Schritt gilt die Umkehrfunktion

$$q = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} \texttt{0E} & \texttt{0B} & \texttt{0D} & \texttt{09} \\ \texttt{09} & \texttt{0E} & \texttt{0B} & \texttt{0D} \\ \texttt{0D} & \texttt{09} & \texttt{0E} & \texttt{0B} \\ \texttt{0B} & \texttt{0D} & \texttt{09} & \texttt{0E} \end{bmatrix} \times \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} \tag{3.21}$$

AddRoundKey. Der letzte Schritt einer Verschlüsselungsrunde ist die Schlüsseladdition und wird mit `AddRoundKey` bezeichnet. Sie addiert einen Rundenschlüssel zum aktuellen Zustandsblock. Diese Addition wird byteweise im $\mathcal{GF}(2^8)$ umgesetzt, sodass sie durch die Gleichung

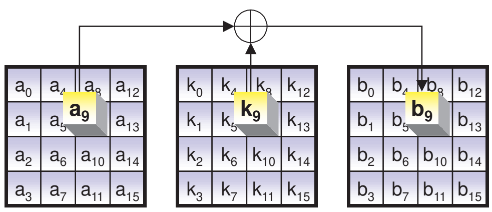


Abbildung 9: Der `AddRoundKey` Schritt.

$$b = a \oplus k \tag{3.22}$$

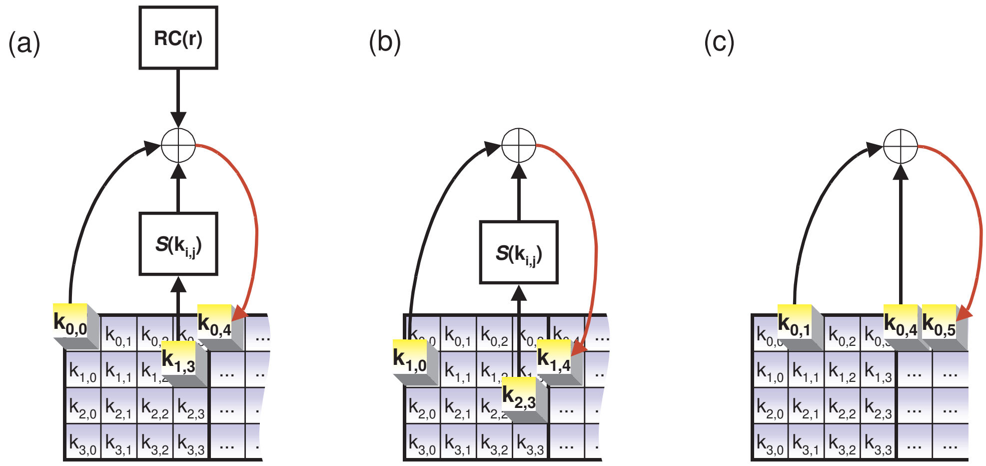


Abbildung 10: Der Schlüsselexpansions-Algorithmus.

wiedergegeben werden kann. Dabei ist b ein Byte des Ausgangszustands, k ein Rundenschlüssel-Byte und a ein Byte des Eingangszustands. `AddRoundKey` ist identisch für Ver- und Entschlüsselung.

Schlüsselexpansion. Beim AES soll für jede Runde ein eigener 16-Byte-Rundenschlüssel erzeugt werden. Der `KeyExpansion`-Algorithmus generiert jene Rundenschlüssel aus dem geheimen Schlüssel. Die Teilschlüssel werden innerhalb einer Runde vom `AddRoundKey`-Schritt verwendet. Der Algorithmus zur Erzeugung von Rundenschlüsseln lautet folgendermaßen:

$$\begin{array}{lll} k_{0,j} = k_{0,j-4} \ + \ \mathcal{S}(k_{1,j-1}) \ + \ \mathcal{RC}[j/4] & \text{für } \ i = 0 & \text{und } (j \ mod \ 4) = 0 \\ k_{i,j} = k_{i,j-4} \ + \ \mathcal{S}(k_{((i+1) \ mod \ 4),(j-1)}) & \text{für } i > 0 & \text{und } (j \ mod \ 4) = 0 \\ k_{i,j} = k_{i,j-4} \ + \ k_{i,j-1} & & \text{für } (j \ mod \ 4) \neq 0 \end{array} \qquad (3.22)$$

Dabei kennzeichnet i die Zeilennummer, j die Spaltennummer, k ein Eingangsbyte, $\mathcal{S}()$ die bereits erwähnte $\mathcal{S}$-Box und $\mathcal{RC}$ die aktuelle Rundenkonstante (s.u.). Bei einem 192 oder 256 Bit breiten Schlüssel wird dieser Algorithmus leicht modifiziert.
Abb. 10 verdeutlicht den oben angegebenen Algorithmus. Figur (a) veranschaulicht die erste Zeile, (b) die zweite und (c) die dritte.
Man bestimmt die Rundenkonstante $\mathcal{RC}$ der Runde r durch

$$\mathcal{RC}[r] = x^{r-1} \qquad \text{für } r \geq 1 \,. \qquad (3.23)$$

Die Rundenschlüssel sind für Chiffrierung und Dechiffrierung identisch.

4 Entwicklung einer Architektur

Nach der Auswahl geeigneter Algorithmen und Anwendungen in Kapitel 2 und der Erläuterung ihrer Grundlagen in Kapitel 3, liegt ein Schwerpunkt dieses Kapitels in der Entwicklung einer Grundstruktur für die Coprozessorarchitektur. Im Fokus steht desweiteren der Controller IMS3311C und die für die Coprozessoren benötigte prozessorseitige Schnittstelle.

Nachfolgend werden eine einheitliche Coprozessorarchitektur, ihre einzelnen funktionalen Einheiten und deren Aufgaben sowie ihre Vernetzung untereinander vorgestellt. Den Kapitelabschluss bildet die Erörterung der Anforderungen an die Hardwareumsetzung des kryptographischen Coprozessors CPCIPH und der MAC-Einheit CPMAC, die in dieser Arbeit entwickelt wurden.

4.1 Der IMS3311C

Der IMS3311C ist ein am Institut für Mikroelektronische Schaltungen und Systeme (IMS) entstandener Mikrocontroller [Ims01a]. Er wurde entwickelt, um einen generell verfügbaren Controllerkern für kundenspezifische integrierte Schaltungen zur Verfügung zu stellen. Der 3311C ist eine Weiterentwicklung des IMS3311 und Instruktionssatz-kompatibel zum Motorola 68HC11 Mikrocontroller [Mot88]. Aufgrund dieser Kompatibilität kann er sämtliche Entwicklungswerkzeuge des M68HC11 mitbenutzen.[36]

Insbesondere aufgrund seiner vielfältigen analogen wie digitalen Erweiterungsmöglichkeiten deckt er einen großen Einsatzbereich ab. Mögliche Coreerweiterungen sind diverse Interfaces (PIO, SCI, SPI, CAN), Analogkomponenten (ADC) oder ein Timer.[37] Abb. 11 zeigt eine mögliche Umsetzung eines 3311C Mikrocontrollersystems. Die Philosophie des IMS3311 beruht auf der Modularisierung des Cores und seiner Peripherieelemente. Dadurch können die einzelnen Module projektabhängig zusammengestellt werden, um auf kundenspezifische Wünsche mit minimalem Aufwand einzugehen.

Für die Nutzung eines Coprozessors ist der IMS3311 um ein Interface erweitert worden, welches das Einführen eigener Coprozessorinstruktionen ermöglicht.

[36] Weiterführende Literatur zum M68HC11 findet man u.a. in [Ros94].

[37] Für die Abkürzungen vgl. Anhang E.

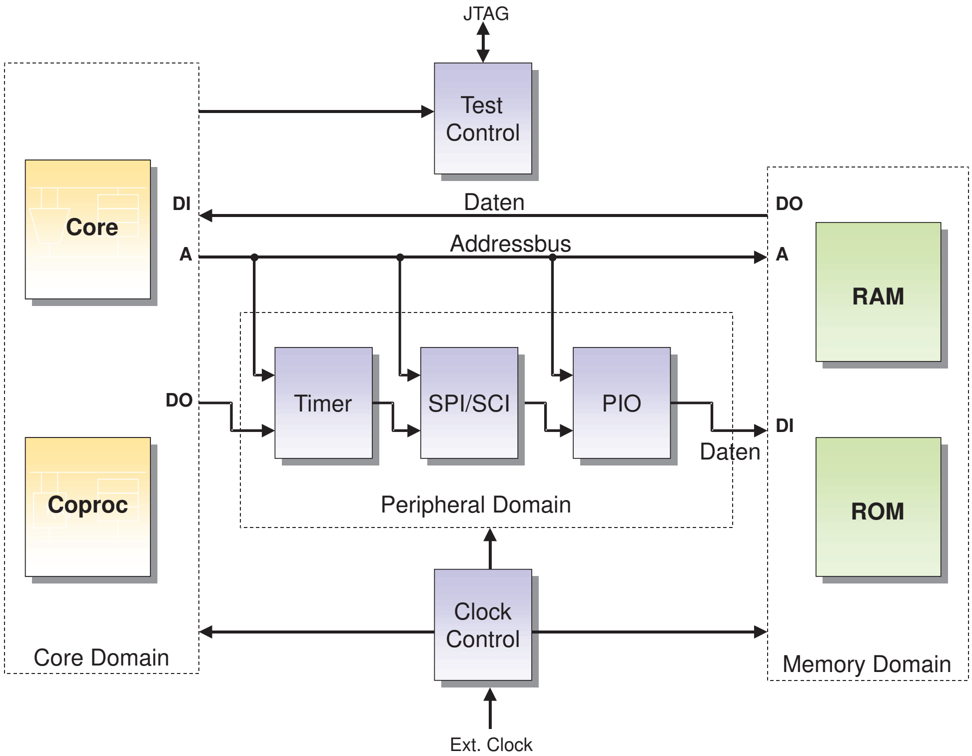


Abbildung 11: Ein IMS3311C Beispielsystem.

4.1.1 Charakteristika

Beim IMS3311C handelt es sich um eine *Akkumulator-Architektur*. Hierbei ist der Akkumlator sowohl ein impliziter Operand als auch ein Ergebnisregister.[38] Der 3311C beinhaltet zwei 8-Bit-Akkumulatoren `ACCA` und `ACCB`. Die Verkettung der beiden Register {`ACCA`, `ACCB`} zu einem 16-Bit-Register wird zu `ACCD` zusammengefasst. Zwei 16-Bit-Indexregister `IX` und `IY` und ein 16-Bit-Programmzähler werden vorwiegend für die Speicheradressierung bereitgestellt. Desweiteren ist der 3311C mit einem 16-Bit-Stackpointer ausgestattet.

Der Instruktionssatz des 3311C ermöglicht zusätzlich zu den üblichen 8-Bit-Instruktionen bereits einige Befehle, die auf 16-Bit-Operanden arbeiten [Ims01b]. Der Befehl `ADDD` ermöglicht z.B. die Addition eines 16-Bit-Speicherwertes zum Akku `ACCD`, was die Assemblerprogrammierung des Cores vereinfachen und zur Verringerung der Codegröße beitragen kann.

[38] vgl. [HP03], S.113ff

Der Controller verfügt über ein 2 kB großes Mikrocode-ROM, das die internen Kontrollsequenzen erzeugt. Die Datentransfers im 3311C finden über einen 8 Bit breiten *Datenbus* statt, während der 16 Bit breite *Adressbus* einen max. 64 kB großen Speicher anspricht. Der Datenbus, der den Datenaustausch der Core Domain[39] mit der Memory Domain und der Peripherie ermöglicht, ist als Ringbus[40] organisiert, wie sich in Abb. 11 erkennen lässt. Das Ringbuskonzept hat den Vorteil, dass es die Last im Gegensatz zu anderen Konzepten auf viele einzelne Treiber verteilt. Der Bus besteht aus kombinatorischen Logikzellen und lässt somit keine undefinierten Zustände zu.[41] Desweiteren ist eine Ringbusimplementierung einfach synthetisierbar.

4.1.2 Coprozessorschnittstelle

Von besonderem Interesse ist die Coprozessorschnittstelle des IMS3311C, da diese für die Umsetzung der instruktionsbasierten Schnittstellenkonzepte im Rahmen dieser Arbeit erforderlich wurde (s. Unterpunkt 5.1).

Die Schnittstelle stellt zwei Opcodes zur Verfügung, die als Prebytes für die coprozessoreigenen Instruktionen Verwendung finden. Der Core wertet die Prebytes folgendermaßen aus:

1. *CopShort* = `$6B`: Es folgt ein Opcode-Postbyte.
2. *CopLong* = `$87`: Es folgen zwei Opcode-Postbytes.

Die Opcode-Postbytes spannen also entweder einen oder zwei neue Opcode-Map-Pages auf. Somit lassen sich mit dem *CopShort*-Befehl 256 neue Instruktionen, mit *CopLong* bis zu 65536 neue Instruktionen kreieren. An die Coprozessoren wird das Prebyte nicht weitergeleitet, ihre Auswertung findet vollständig innerhalb des Cores statt.

---

[39] Zur Core Domain zählen in dieser Arbeit der Controllerkern und die Coprozessoren, die Memory Domain besteht aus RAM und ROM, alle übrigen Einheiten werden der Peripheral Domain zugerechnet (s. Abb. 11). Von hier an ist mit Core grundsätzlich der Controllerkern gemeint.

[40] Beim Ringbus werden die Daten nacheinander durch die teilnehmenden Einheiten geführt. Die Einheit, die der zentrale Adressdecoder im aktuellen Taktzyklus selektiert, hat Lese- oder Schreibrechte, die anderen Einheiten reichen die Daten lediglich weiter.

[41] vgl. [Fär87] und [Ims01a]

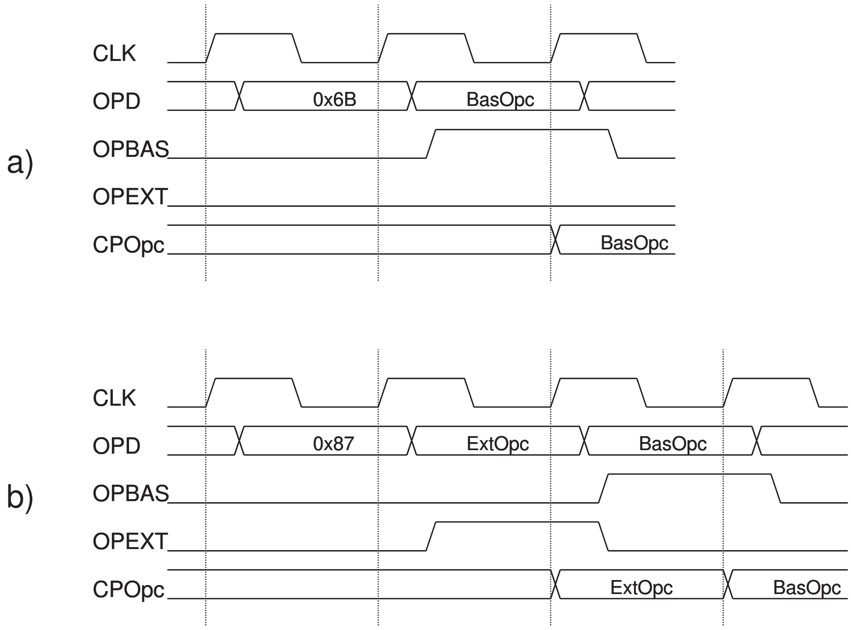


Abbildung 12: Timing Diagramm der (a) *CopShort*- und (b) *CopLong*- Instruktion.

Bei beiden Instruktionstypen folgt als Postbyte das sogenannte "Basic Opcode"-Byte, während nur bei *CopLong*-Instruktionen zusätzlich das "Extended Opcode"-Byte ausgewertet werden muss.

Während Abb. 12 die Timing-Diagramme beider Befehlstypen zeigt, visualisiert Abb. 13 die Signale des Coprozessorinterfaces. Die beiden Postbytes werden über den `OPD`-Bus (Opcode and Data Bus) übertragen, der den Datentransfer zwischen Controller und Coprozessor übernimmt. Dies geschieht genau in den Taktzyklen, in denen das `OPBAS`- (Opcode Basic Strobe) oder das `OPEXT`- (Opcode Extended Strobe) Signal gesetzt ist. Das `OPBAS`-Signal zeigt dabei die Übertragung eines "Basic Opcodes" an, während das `OPEXT`-Signal die Übermittlung des "Extended Opcodes" kennzeichnet.

Mit steigender Taktflanke müssen die Opcode-Bytes vom Coprozessor übernommen und in einem `CPOpc`- (Coprozessor Opcode) Register gesichert werden.
Desweiteren verdeutlicht Abb. 12 (b), dass der "Extended Opcode" vor dem "Basic Opcode" übertragen wird.

Letzterer wird in zwei Nibbles[42] unterteilt: Sowohl Core als auch Coprozessor werten das Nibble mit der geringeren Wertigkeit (LSN) aus, während das höher-

[42] Ein Nibble entspricht 4 Bit.

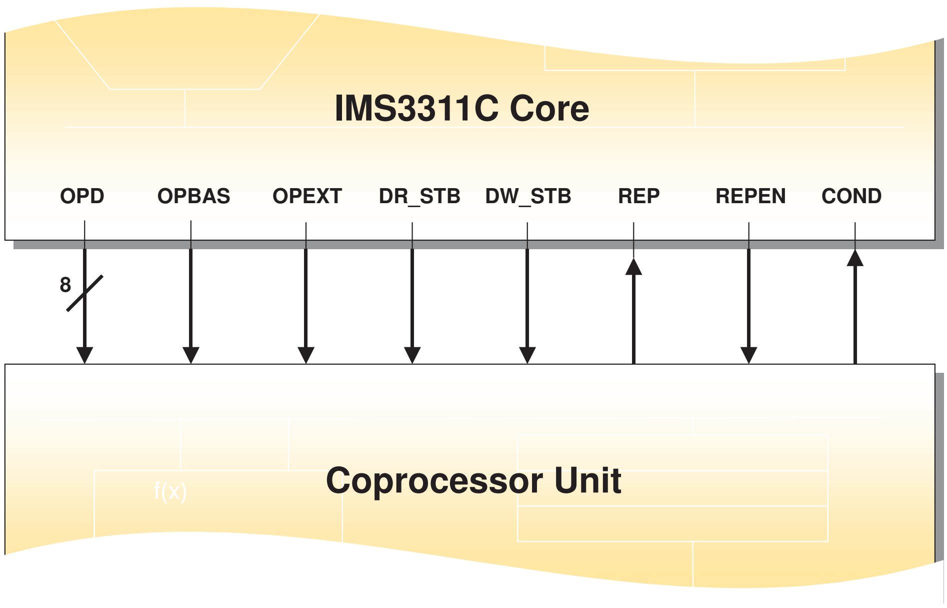


Abbildung 13: IMS3311C Coprozessorschnittstelle.

wertige Nibble (MSN) ausschließlich für den Coprozessor bestimmt ist und vom Core unbearbeitet an diesen weiterpropagiert wird. Das vom Core ausgewertete Nibble klassifiziert Instruktionen in:

- Load/Store (LD/ST)-Befehle mit verschiedenen Adressierungsarten,
- Branch-Befehle,
- Push/Pop-Befehle und
- inhärente[43] benutzerspezifische Befehle.

Der Coprozessor kann bei einem LD/ST-Befehl die Adressierungsarten *direct*, *extended*, *indexed* und *immediate* des Prozessors mitbenutzen.[44] Bei einem LD-

[43] Die inhärente Adressierung benötigt keine explizite Angabe der Operanden. Mögliche Operanden gehen direkt aus dem Befehlswort hervor. Bsp: INX (inkrementiere Indexregister X).

[44] vgl. [Ims04]

Befehl und gesetztem `DR_STB`- (Data Read Strobe) Signal empfängt der Coprozessor die Daten vom `OPD`-Bus, bei einem ST-Befehl werden Daten bei gesetztem `DW_STB` (Data Write Strobe) auf dem Datenbus erwartet. Die zu empfangenden Daten kommen also direkt von dem `OPD`-Prozessorbus, während die zu schreibenden Daten auf den gemeinsamen Datenbus gelegt werden.

Falls mehrere Datenbytes empfangen werden sollen, kann man dies coprozessorseitig durch Setzen des `REP`- (Repeat) Flags erreichen, sobald der Prozessorkern dies durch Setzen des `REPEN`- (Repeat Enable) Flags zulässt. Das Interface erlaubt somit blockweises Lesen und Schreiben des Speichers. Der Prozessor führt bei Erhalten des `REP`-Signals einen Autoinkrement auf das Adressregister durch und ermöglicht so im nächsten Taktzyklus den Zugriff auf die folgende Adresse in RAM oder ROM.

Der `COND`- (Condition) Eingang des Prozessors gibt an, ob bei einem Branch-Befehl die Bedingung wahr oder falsch ist und führt abhängig vom logischen Pegel einen relativen Sprung durch. Dieses Signal ermöglicht dem Coprozessor bedingte Sprünge und erhöht somit die Flexibilität der Schnittstelle.

4.2 Entwicklung der Coprozessorarchitektur

Nach Erläuterung des Controllers IMS3311C und seiner Coprozessorschnittstelle soll nun auf das Konzept der Coprozessorarchitektur eingegangen werden. Im Rahmen dieser Arbeit wurde ein Architekturgerüst entwickelt, das es ermöglicht, dass sämtliche Coprozessoreinheiten für den 3311C nach derselben modularen Grundstruktur aufgebaut werden. Abb. 14 veranschaulicht diese. Die Coprozessoren setzen sich aus den Einheiten

- Decoder und Interface (IF),
- Control Unit (CU),
- Address Generation Unit (AGU),
- Register Unit und
- Execute Unit (EXEC)

zusammen.

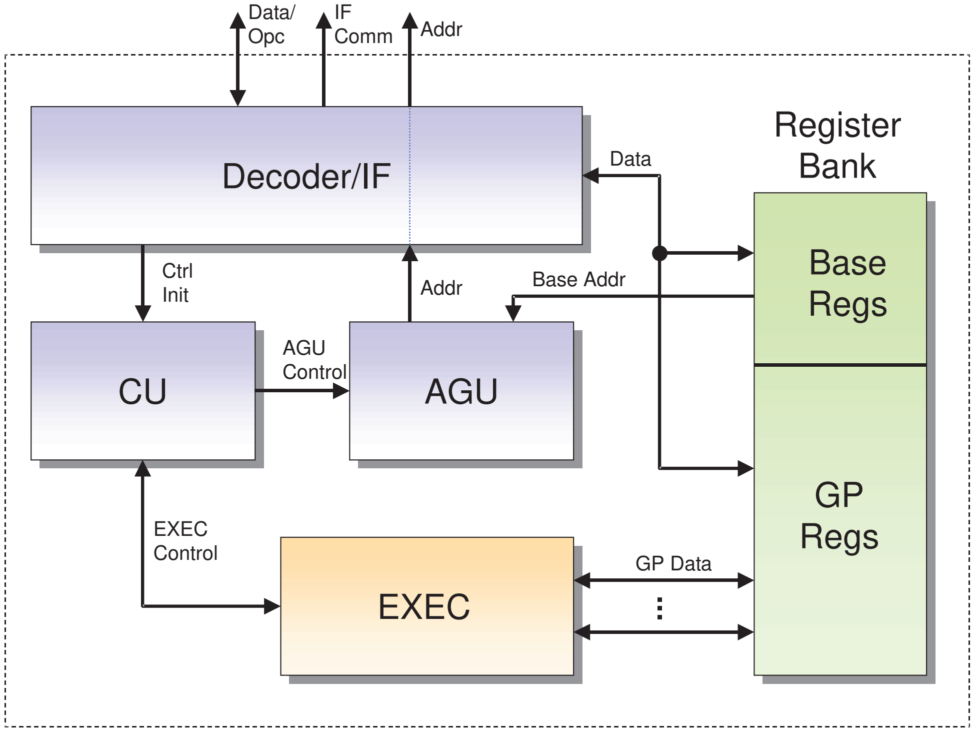


Abbildung 14: Modulare Grundstruktur der Coprozessorbausteine.

Der Decoder erhält die Befehle vom 3311C und entschlüsselt sie für den Coprozessor. Die Ausgänge der Dekodiereinheit gelangen zur Kontrolleinheit.

Die Kontrolleinheit CU erzeugt für die aktuelle Instruktion adäquate Kontrollsequenzen mittels festverdrahteter Logik, welche an die ausführende Einheit EXEC und die Adress-Generierungseinheit AGU weitergegeben werden. Der Kern der CU ist in der Regel ein globaler endlicher Automat (FSM[45]), der die Abfolge sämtlicher Einheiten steuert.

Die AGU wird abhängig vom aktuellen Interface benötigt. Sie ist erforderlich, falls der Coprozessor eigene Speicherzugriffe durchführt. In diesem Fall generiert die AGU je nach Befehl eine Adressequenz, die das Lesen und Schreiben von Daten über den Datenbus ermöglicht. Pro Takt kann somit ein 8 Bit breites Datum zwischen Coprozessor und Speicher übertragen werden. Entsprechend dem 64 kB großen Adressraum des 3311C ist das Adressregister der Coprozessoreinheiten bis zu 16 Bit breit.

[45] Finite State Machine

Kapitel 5 zeigt, dass sich für die flächenoptimale Implementierung des AES die Nutzung einer AGU aufdrängt, während sie für die Umsetzung einer MAC-Unit nicht sinnvoll ist.

Die in Abb. 14 dargestellte Register Unit beinhaltet die Registerbank der Coprozessoreinheit. Dazu gehören zum einen die GP- (General Purpose) Register und zum anderen die BA- (Basisadress) Register. Die GP-Register sind der Datenbusbreite entsprechend 8 Bit breit und übernehmen folgende Aufgaben:

- Ablegen von Zwischenergebnissen bei Multi-Cycle-Operationen,
- Speichern der Operations-Endergebnisse,
- Sichern der aus dem Speicher ausgelesenen Daten,
- Bereitstellen der in den Speicher zu schreibenden Daten.

Basisadressen geben in dieser Arbeit die Position der zu bearbeitenden Datenblöcke im Speicher an. Die Datenblöcke können entweder Ein- oder Ausgangsdaten für den Coprozessor sein bzw. Konstanten, die als Lookup-Tables (LUTs) im Speicher angeordnet sind (vgl. Abb. 15). Bezogen auf das Beispiel des CPCIPH lässt sich mit den BA-Registern z.B. die Speicherposition der Nachricht, des Chiffre, des geheimen oder expandierten Schlüssels oder der Konstanten der Byte-Substitution angeben.

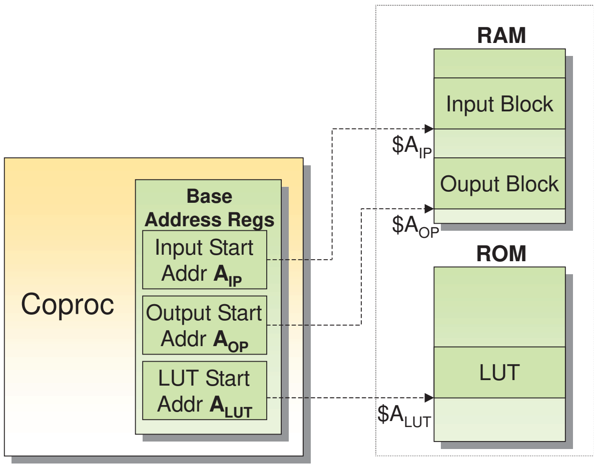


Abbildung 15: Basisadressregister des CPCIPH.

Die BA-Register sind zwischen 8 und 16 Bit breit. Damit beginnt ein Datenblock bei einem 8 Bit breiten BA-Register stets auf einer 256-Byte-Grenze im Speicher, da die letzten 8 Bit der Anfangsadresse als Null angenommen werden.

Sämtliche logische und arithmetische Funktionalität des Coprozessors wird bei der hier vorgestellten Grundstruktur innerhalb der Execute Unit EXEC umgesetzt. Im Beispiel des CPCIPH werden in dieser Einheit Berechnungen wie die Schlüsseladdition oder die Multiplikationen im $\mathcal{GF}(2^8)$ für den `MixColumns`

Schritt vollzogen. Der Anteil der kombinatorischen Logikbausteine an der Gesamtfläche dieser Einheit ist sehr groß im Vergleich zum Anteil der Speicherelemente. Sollte sich ein Zustand der globalen FSM in mehrere lokale Zustände unterteilen, so werden diese Subzustandsmaschinen innerhalb der EXEC-Einheit berechnet (s. Unterpunkt 6.2.2).

Die hier vorgestellte Architektur bildet eine Basis für unterschiedliche Coprozessorimplementierungen. Sie gibt eine Struktur für die Abbildung verschiedenartiger Algorithmen der digitalen Signalverarbeitung auf eine beschleunigende Spezialeinheit vor.

Nach der Bestimmung des Architekturgerüsts behandeln die beiden folgenden Unterkapitel die Hauptfunktionalität der auf dieser Grundstruktur basierenden Einheiten CPCIPH und CPMAC. Das dargestellte Anforderungsprofil bildet die Grundlage für den Instruktionssatz der Beschleunigungseinheiten.

4.3 Anforderungsprofil des CPCIPH

Die Funktionalität des CPCIPH beschränkt sich auf wenige Anforderungen. Allerdings ist jede einzelne Anforderung umfangreich und wird später in mehrere Teiloperationen zerlegt. Die geforderten Basisfunktionalitäten des Coprozessors bestehen in:

1. Chiffrierung eines 128-Bit-Datenblocks mit einem 128-Bit-Schlüssel,
2. Dechiffrierung eines 128-Bit-Datenblocks mit einem 128-Bit-Schlüssel und
3. Erzeugung der Rundenschlüssel.

Ergänzend zu diesen Basisfunktionalitäten erhält der CPCIPH optional Zusatzinstruktionen, die entweder das Programmieren vereinfachen oder den Datenverkehr zum Prozessor oder zum Speicher regeln. Unterstützt wird die

1. freie Platzierung von Nachricht und Geheimtext im Speicher,
2. freie Platzierung des Schlüssels im Speicher und
3. Chiffrierung und Dechiffrierung großer zusammenhängender Datenblöcke.

Der erste und zweite Aspekt bewirken, dass Nachricht, Geheimtext und Schlüssel nicht zwangsläufig an derselben Stelle im Speicher platziert werden müssen und somit zyklenaufwendiges Umkopieren von Daten vermieden wird.

Das Verarbeiten großer Datenblöcke senkt die Codegröße des Assemblerprogramms im Programmspeicher, da weniger Befehle für die gleiche Funktionalität erforderlich sind. Der Fall der Ver- oder Entschlüsselung eines großen Datenblocks ist wahrscheinlich, daher hat die Implementierung dieser Funktion auch für eine flächenoptimierte Umsetzung Berechtigung. Nebenbei verkürzt sie den Chiffrier- und Dechiffriervorgang um einige Taktzyklen und erhöht den Programmierkomfort.

4.4 Anforderungsprofil des CPMAC

Im Gegensatz zum CPCIPH berechnet der CPMAC keine vollständigen Algorithmen mit einer Instruktion. So wird z.B. ein digitales Filter durch Aneinanderreihung mehrerer MAC-Operationen und eventueller Normalisierung der Teilergebnisse berechnet. Eine MAC-Operation bearbeitet also keinen ganzen Algorithmus, sondern lediglich einen Teilschritt. Folglich sollte der CPMAC über viele Instruktionen verfügen, die jedoch nur atomare Bestandteile eines vollständigen Algorithmus darstellen.

Der CPMAC sollte dem 8-Bit-Mikrocontroller IMS3311C die Funktionalität eines 16-Bit-Prozessors im Bereich der digitalen Signalverarbeitung geben, um ihn in diesem Bereich gezielt zu beschleunigen. Die Architektur des CPMAC erhält somit zwei 16-Bit-Faktorregister FA und FB für die Operanden und einen 32-Bit-Akkumulator MA zum Speichern der Gesamtergebnisse. Als Basisoperationen des CPMAC werden gefordert:

- 32-Bit-Addition zum Akkumulator: MA = MA + {FA, FB}[46],
- 32-Bit-Subtraktion vom Akkumulator: MA = MA - {FA, FB},
- Löschen des Akkumulators: MA = 0,
- arithmetischer Linksshift: MA = MA ≪ <pos>,

[46] {FA, FB}: Diese Darstellung steht für die Verkettung der zwei 16-Bit-Register FA und FB zu einem 32-Bit-Wort, wobei FA der höherwertige 16-Bit-Anteil ist.

- arithmetischer Rechtsshift: $\texttt{MA} = \texttt{MA} \gg \texttt{<pos>}$,
- Multiplikation: $\texttt{MA} = \texttt{FA} \cdot \texttt{FB}$, und natürlich
- Multiply-Accumulate: $\texttt{MA} = \texttt{MA} + \texttt{FA} \cdot \texttt{FB}$.

Die Berechnung sämtlicher Operationen erfolgt im Zweierkomplement. Zudem soll der Coprozessor einen Überlauf im Akkumulator erkennen (Overflow Flag) und automatisch mittels Sättigung (Saturation) auf diesen reagieren. Sättigung und Vorzeichen des Akkumulators MA sollen in einer erweiterten Version des CPMAC mittels Branch-Instruktionen auswertbar sein.

Zusätzlich zu dieser Funktionalität sollte der CPMAC Speicherzugriffs-Instruktionen unterstützen. Faktorregister und Akkumulator können somit aus RAM oder ROM geladen oder in den Arbeitsspeicher zurückgeschrieben werden.

5 Implementierung der Schnittstelle

Während im letzten Kapitel die Architekturen des Controllers IMS3311C und der hier entwickelten Coprozessoren erläutert wurden, werden in diesem Kapitel Schnittstellenimplementierungen zwischen dem 3311C und der Coprozessoreinheit erarbeitet. Diese werden in verschiedenen Entwicklungsstufen aufgezeigt, die sich bezüglich Intelligenz, Datendurchsatz und Flächenaufwand unterscheiden.
Die Schnittstelle strebt folgende Ziele an:

- Unterstützung verschiedener Coprozessortypen,
- Unterstützung der Flächenoptimierung der Core Domain,
- einfache Erweiterbarkeit der Schnittstelle für mehrere Coprozessoren und
- Minimierung der Verlustzyklen hervorgerufen durch Schnittstellenkommunikation.

In den folgenden Abschnitten werden mögliche Schnittstellenimplementierungen vorgestellt sowie Vor- und Nachteile diskutiert. Die Schnittstellentypen werden unterteilt in

- befehlsbasierte Schnittstellen und
- Memory-Mapped-Schnittstellen.

Der Core lädt bei der instruktionsgestützten Schnittstelle coprozessoreigene Befehle aus dem Programmspeicher und gibt sie zum Teil undecodiert an den Coprozessor weiter, der diese auswertet und ausführt.

Bei der Memory-Mapped-Implementierung liegen die Kontroll-, Status- und BA-Register im Adressraum. Dadurch kann der Controllerkern mittels LD/ST-Befehlen mit dem Coprozessor kommunizieren. Aus Sicht des Prozessors verhält sich der Coprozessor dementsprechend ähnlich wie eine Peripherieeinheit.

Befehlsbasierte Implementierungen nutzen die für diese Arbeit erstellte Schnittstellenerweiterung des 3311 (s.a. 4.1.2), während die Memory-Mapped-Implementierung bereits mit dem nicht erweiterten Interface des 3311 arbeiten kann.

Desweiteren wird für die wichtigsten der vorgestellten Schnittstellen eine Beispielimplementierung für den CPMAC oder den CPCIPH erläutert. Die angegebenen Beispiele der instruktionsbasierten Schnittstelle sind in Hardware implementiert, simuliert und sowohl für eine FPGA als auch für einen ASIC synthetisiert worden.

5.1 Instruktionsbasiertes Interface

Dieser Unterpunkt thematisiert mehrere Schnittstellenimplementierungen, bei denen der Coprozessor mittels dedizierter Instruktionen angesprochen wird. Nach Vorstellung der Schnittstellenkonzepte des ARM7TDMI und Motorola 68020 folgt die Untersuchung möglicher Schnittstellen für den 3311C.

5.1.1 ARM7 und M68000

Schnittstelle des ARM7. Wie bereits in Abschnitt 2.2.2 erwähnt, bietet der ARM7 ein instruktionsbasiertes Coprozessorinterface.[47] Diese Schnittstelle sieht es vor, dass der Coprozessor den Datenbus überwacht und nur die an ihn gerichteten Instruktionen übernimmt. Dabei fügt sich der Coprozessor in die Prozessor-Pipeline ein. Die Schnittstelle des ARM7 ermöglicht die Integration von 16 parallelen Coprozessoreinheiten.

Die Umsetzung der Schnittstelle geschieht wie folgt:
Der ARM7 untersucht, ob es sich bei der aktuell auf dem 32 Bit breiten Datenbus befindlichen Instruktion um einen Coprozessorbefehl handelt. Ist dies der Fall, so gibt der ARM-Controller dies mittels des `nCPI`-Signals an alle Coprozessoren weiter.

Der durch ein vier Bit breites Opcodefeld identifizierte Coprozessor antwortet auf zwei Handshaking-Leitungen[48], die über den Zustand des Coprozessors Auskunft geben. Mögliche Zustände sind

- ”present”: Der Coprozessor wird angesprochen und kann die Instruktion annehmen.

[47] vgl. [Arm01], Kapitel 4

[48] Bei Handshaking Protokollen kommunizieren zwei Teilnehmer miteinander, wobei der eine Teilnehmer nach einer Aktion explizit auf die Reaktion des anderen Teilnehmers wartet, bevor er die nächste Aktion auslöst.

- "busy": Der Coprozessor wird angesprochen, ist aber noch aktiv.
- "absent": Die aktuelle Instruktion ist ungültig für diesen Coprozessor.

Falls der Coprozessor im "busy"-Zustand ist, führt der Prozessor einen "busy-wait" durch. Er verharrt also so lange in der Pipeline-Stufe Execute, bis der Coprozessor wieder frei ist.

Der ARM7 stellt für das Coprozessorinterface drei verschiedene Instruktionstypen bereit:

- Datentransfer zwischen coprozessorinternen Registern,
- Datentransfer zwischen Prozessor und Coprozessor und
- Datentransfer zwischen Speicher und Coprozessor.

Wie auch beim IMS3311C besteht beim ARM7 die Möglichkeit, Daten direkt aus dem Speicher in den Coprozessor zu laden. Dabei berechnet stets der Prozessor die Adressen. Der ARM7 erlaubt mittels seiner LDC- (Load Data in Coprocessor) und STC- (Store Coprocessor Data) Befehle das Lesen und Schreiben von mehr als einem Datenwort.

Es ist nicht möglich, mit *einer* Instruktion sowohl lesend als auch schreibend auf den Speicher zuzugreifen, so wie dies für die hardwareoptimale Umsetzung des AES im CPCIPH beabsichtigt ist (s. Unterpunkt 5.1.3). Somit unterstützt er den direkten Speicherzugriff durch den Coprozessor nicht, der für die gatteroptimale Umsetzung komplexer Algorithmen benötigt wird.

Schnittstelle des Motorola MC68020. Motorola stellt für den M68020 und M68030 ein Coprozessorinterface zur Verfügung, an das bis zu 8 externe Coprozessoren angeschlossen werden können. Die Fließkommaerweiterungen MC68881 und MC68882 nutzen das Interface zur Beschleunigung der Berechnung mathematischer Fließkommaoperationen (Addition, Multiplikation, Logarithmus, etc.).

Im Gegensatz zur Schnittstelle des ARM7 ist das Interface des MC68020 recht komplex und keine rein instruktionsbasierte Schnittstelle. Obwohl es dedizierte Coprozessorinstruktionen gibt, findet die Kommunikation zwischen Prozessor und Coprozessor über den Speicher statt. Dennoch wird die Schnittstelle hier als instruktionsbasiertes Interface geführt, da für den Assemblerprogrammierer die auf dem Speicher durchgeführte Kommunikation zwischen Controller und Coprozessor nicht sichtbar ist.

Angesprochen werden die angeschlossenen Coprozessoren über die sogenannte "F-Line"-Instruktion, bei welcher die ersten vier Bits gesetzt sind (also hexadezimal `$F`). Die übrigen Bits bestimmen den angesprochenen Coprozessor über ein 3 Bit großes Identifikationsfeld und den Typ der Instruktion. Je nach Art der Instruktion folgt dem ersten Instruktionswort eine unterschiedliche Anzahl weiterer Wörter.

Das erste Wort wird vom Prozessor dekodiert, dieser selektiert den angesprochenen Coprozessor und die Kommunikation zwischen Prozessor und Coprozessor wird initiiert. Über die Adressausgänge des Prozessors und den bidirektionalen Datenbus werden nun die Daten und Befehle ausgetauscht. Dabei liegen immer an derselben Stelle im Adressraum die "Coprocessor Instruction Register"- (CIR) Sets. Motorola schreibt vor, an welcher Stelle welche Register im CIR-Set lokalisiert sind. In die verschiedenen Memory-Mapped-Register werden dann im Verlaufe einer Instruktion Informationen wie

- Operanden,
- Operandenadressen,
- weitere Instruktionswörter oder
- die Coprozessorantwort (Response)

geschrieben.

Ein angesprochener Coprozessor kann über das Response-Register Dienste vom Prozessor anfordern. Dafür muss er eins von zahlreichen sogenannten Primitiven, also vom Prozessor dekodierbare Bitfolgen, im Response-Register ablegen. Er kann u.a. Operanden für das Operandenregister oder ein zweites Coprozessor-Instruktionswort anfordern.

Ein großer Nachteil dieses Schnittstellendesigns ist seine Umständlichkeit, sodass der Entwurf einer einfachen und intuitiven Schnittstelle für einen Coprozessor des MC68020 schwer umsetzbar ist.
Die nächsten Unterpunkte stellen Schnittstellen zwischen dem 3311C Core und seinen Coprozessoren vor.

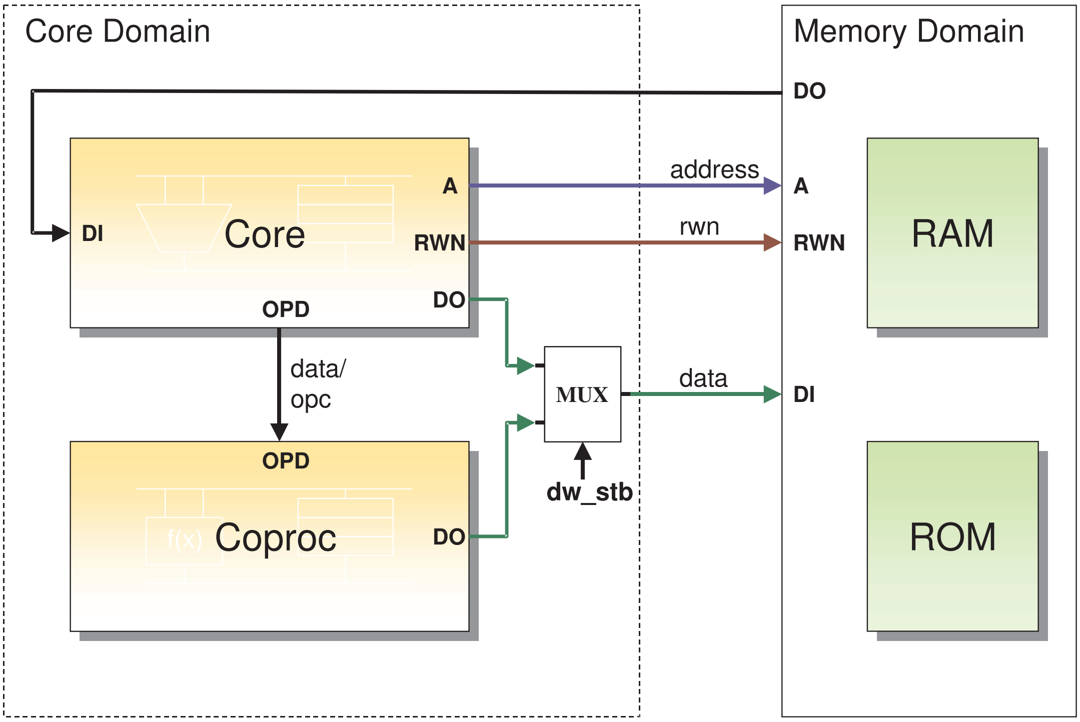


Abbildung 16: Coprozessor als interne Prozessoreinheit.

5.1.2 Schlichte Schnittstelle für IMS3311C

Bei der schlichten Lösung erscheint der Coprozessor wie eine interne Prozessoreinheit, auch wenn er prozessoreigene Ressourcen wie den Akkumulator oder Kontroll- und Statusbits nicht mitnutzen kann. Der aktuelle Bearbeitungszustand der Coprozessorinstruktionen muss jedoch nicht überprüft werden, weil diese in einem sequentiellen Bearbeitungsstrom mit den Prozessorinstruktionen liegen. In diesem Modell führt der Coprozessor die an ihn gerichteten Instruktionen direkt aus. Der Prozessor muss dabei nicht wie beim Coprozessorinterface des ARM7 auf Statusinformationen des Coprozessors warten, sondern kann nacheinander Instruktionen an diesen senden. Dies hat zur Folge, dass der Coprozessor innerhalb weniger Taktzyklen eine Instruktion bearbeiten muss, um für eine unmittelbar folgende Instruktion sämtliche erforderliche Ressourcen wie die Recheneinheit, die Flags oder die internen Coprozessorregister zur Verfügung stellen zu können.

Ein stark vereinfachtes Blockschaltbild dieses Schnittstellentypus veranschaulicht Abb. 16.[49] Das bereits in 4.1.2 eingeführte Data-Write-Strobe-Signal `DW_STB` ist das Steuersignal für einen Multiplexer, der über den Ausgang des Datenbusses aus der Core Domain entscheidet. Das Signal wird vom Prozessor getrieben, welcher entscheidet, ob der Coprozessor Buszugriff im aktuellen Taktzyklus erhält. Der Coprozessor kann Datentransfers erst initiieren, wenn dies vom Core veranlasst wird. Die benötigten Eingangsdaten erhält der Coprozessor direkt über den `OPD`-Bus.

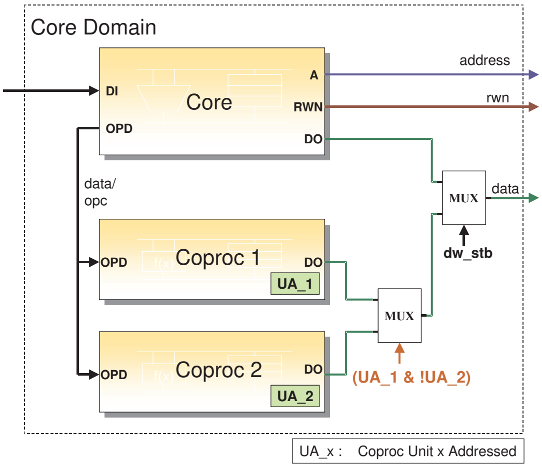


Abbildung 17: Erweiterung des schlichten Interfaces für zwei Coprozessoren.

Der Schnittstellentypus ist leicht auch für mehrere Coprozessoren durch paralleles Anschließen an den Core verwendbar. Abb. 17 gibt die Erweiterung des Interfaces für zwei Coprozessoren wieder. Beide Coprozessoren erhalten am Eingang die Daten vom `OPD` Bus und entscheiden anhand der übertragenen Instruktionen, ob sie angesprochen sind, was die Signale `UA_1` und `UA_2` zum Ausdruck bringen.[50] Die zurückzuschreibenden Daten werden gemultiplext, sodass nur der selektierte Coprozessor schreibenden Speicherzugriff erhalten kann.

Als gatteroptimierte AES-Beschleunigungseinheit ist der Typus ungeeignet, da dieser Algorithmus nur mit hohem Flächenaufwand in wenigen Taktzyklen durchführbar ist (vgl. Kap. 6). Alternativ wurde im Zuge dieser Arbeit die Unterteilung des AES in mehrere Einzelbefehle untersucht, die nur bestimmte rechenaufwendige Teile des Algorithmus durchführen. Diese Lösung ist softwarenah, bringt jedoch nur einen sehr geringen Zeitvorteil gegenüber einer reinen Softwarelösung ohne Coprozessor. Sie hat einen ähnlich hohen Flächenaufwand wie die Umsetzung mit *einer* Instruktion, da lediglich die ohnehin sehr kleine CU durch ein Assemblerprogramm ersetzt würde. Aus diesem Grund wird auf diese Art der Umsetzung des AES in Hardware an dieser Stelle verzichtet.

[49] Die Peripheral Domain wird aus Gründen der Anschaulichkeit nicht abgebildet, da diese das vorgestellte Schnittstellenkonzept nicht beeinflusst.

[50] UA: Unit Addressed

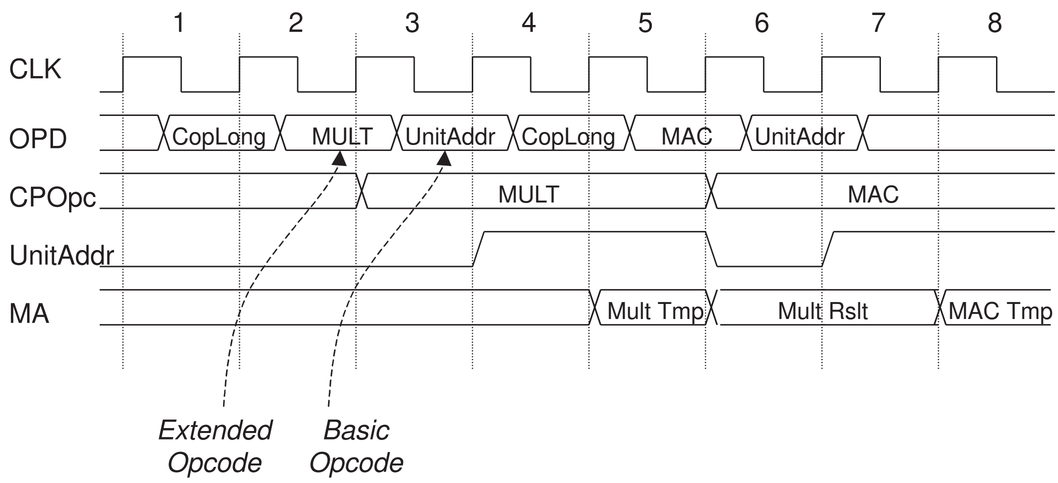


Abbildung 18: Timing Diagramm des CPMAC für die zwei aufeinanderfolgenden Instruktionen Multiplikation und MAC.

Stattdessen bietet sich der schlichte Schnittstellentypus für die CPMAC Einheit an, da diese zur Ausführung einer Instruktion nur besonders wenige Taktzyklen benötigt.

Der CPMAC mit schlichter Schnittstelle. Der CPMAC nutzt für seine Instruktionen den in Unterpunkt 4.1.2 mit *CopLong* bezeichneten Schnittstellentyp. Dieser stellt neben dem allein vom Prozessor ausgewerteten Prebyte zwei benutzerspezifische Postbytes zur Verfügung. Das frei zur Verfügung stehende MSN des "Basic Opcode"-Bytes wird dazu verwendet, einen Coprozessor zu selektieren. Der Controller addressiert somit bis zu 16 verschiedene Coprozessoren. Diese können sequentiell durch ein Assemblerprogramm angesprochen werden, ohne überschneidend Speicherzugriffe anzufordern, da sämtliche Datentransfers vom Controller gesteuert werden. Abbildung 18 veranschaulicht dies für die Ausführung der direkt hintereinander folgenden Befehle Multiplikation und MAC.

Das Diagramm zeigt die Übertragung des Opcodes über den `OPD` Bus (vgl. auch Abb. 16). Mit steigender Taktflanke wird das "Extended Opcode"-Byte *MULT* in das Coprozessor-Opcode-Register `CPOpc` übernommen.

Im nächsten Taktzyklus wird der "Basic Opcode" *UnitAddr* über den `OPD`-Bus übertragen. Dieser trägt im MSN die Information über den angesprochenen Coprozessor und im LSN die Adressierungsart der Operanden. Jeder der Coprozessoren speichert mit steigender Taktflanke im `UnitAddr`-Register, ob er durch die Instruktion adressiert wird. Falls sich mehrere Coprozessoren im System befinden, entscheidet sich demnach erst in diesem Taktzyklus, welcher von ihnen angesprochen ist.

Da *MULT* in den `CPOpc` Registern aller angeschlossenen Coprozessoren gespeichert wird, also auch in denen der nicht adressierten Einheiten, beginnt die Instruktionsbearbeitung erst nach der durch das `UnitAddr`-Flag erteilten Freigabe.

Desweiteren muss der alte Befehl komplett abgearbeitet sein, bevor ein neuer "Extended Opcode" vom Coprozessor angenommen wird, unabhängig davon, ob er tatsächlich für diesen Coprozessor bestimmt ist. Da der "Extended Opcode" bedingungslos übernommen wird, würden ansonsten die alten Instruktionen überschrieben und so Fehler bei dem in Bearbeitung befindlichen Befehl verursacht. Beim CPMAC wird dies dadurch vermieden, dass die Instruktionsbearbeitung maximal zwei Taktzyklen einnimmt. Somit ist der alte Befehl durchgeführt, bevor der neue "Extended Opcode" gespeichert wird. Sämtliche Befehle mit den dazugehörigen Opcodes lassen sich Anhang D entnehmen.

Die vollständige Verzahnbarkeit des Prozessorkerns und der maximal 16 Coprozessoren hat nicht nur für die arithmetisch-logischen Operationen, sondern auch für die LD/ST-Befehle Gültigkeit: Es ist nicht möglich, dass zwei Coprozessoreinheiten gleichzeitig auf den Speicher zu schreiben versuchen, da die Einheiten nacheinander vom Prozessor initiiert und ihre Schreib- und Lesezugriffe zentral und sequentiell vom Core ausgelöst werden. Es ist Aufgabe des Coprozessors im allgemeinen und des CPMAC im speziellen, die Daten für einen Schreibzugriff auf den Speicher rechtzeitig bereitzuhalten.

Der CPMAC nutzt die Option des blockweisen Speicherzugriffs mit Hilfe der `REP`- und `REPEN`- Signale zum Laden und Zurückschreiben des 32-Bit-Akkumulators `MA`. Somit können Datentransfers zwischen Speicher und Coprozessor zyklensparend durchgeführt werden, ohne dass der Prozessorkern die Kontrolle über den Speicherbus abgeben muss.

Die schlichte Schnittstelle erfüllt das Ziel der einfachen Erweiterbarkeit für mehrere Coprozessoren. Die einzelnen Einheiten werden ohne gegenseitige Be-

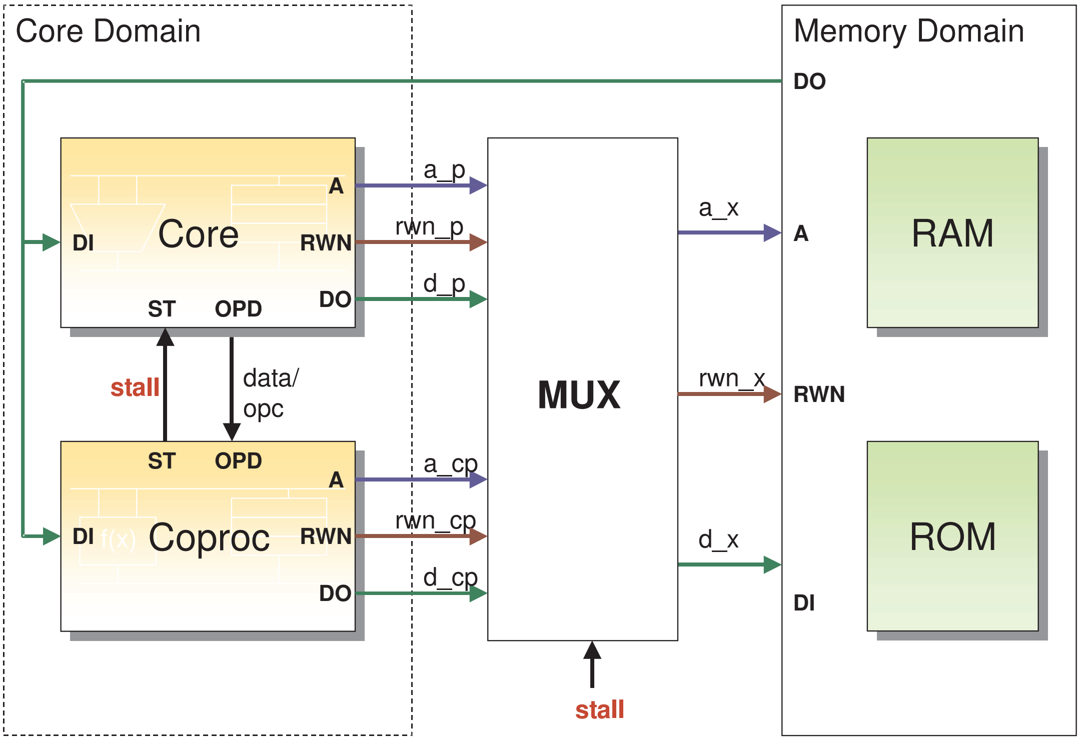


Abbildung 19: Sequentielle Instruktionsbearbeitung durch Anhalten des Prozessorkerns.

einflussung mittels des "Basic Opcodes" ausgewählt.
Allerdings ist die Schnittstelle nicht für sämtliche Anforderungen adäquat, da die Instruktionen innerhalb weniger Takte verarbeitet werden müssen und der Speicherzugriff ausschließlich vom Prozessorkern gesteuert wird. Falls *eine* Instruktion des Coprozessors eigene Lese- und Schreib-Speicherzugriffe erfordert, so ist dies mit dem schlichten Interface nicht möglich.

5.1.3 Sequentielle Instruktionsbearbeitung

Da das Interface aus 5.1.2 nicht für Coprozessoren geeignet ist, die mehr als drei Execute-Zyklen für eine Instruktion benötigen, wird es weiterentwickelt, um auch Algorithmen mit komplexeren Instruktionen durchführen zu können. Eine Möglichkeit dafür ist das Anhalten des Prozessorkerns während der Ausführung einer Coprozessorinstruktion. Die Umsetzung dieses Ansatzes wird in Abb. 19 veranschaulicht. Bei Coprozessorinstruktionen, die das Anhalten des Prozessors bewirken, arbeiten Core und Beschleunigungseinheit *sequentiell*.

STALL ist in diesem Interface ein vom Coprozessor getriebenes Signal, welches das Anhalten des Prozessors noch in demselben Taktzyklus verursacht. Während der Prozessor angehalten ist, führt er keine Speicherzugriffe durch. Dadurch wird erreicht, dass keine neuen Instruktionen bearbeitet werden. Dies verhindert das Senden einer neuen Instruktion an den Coprozessor, bevor der alte Befehl vollständig ausgeführt wurde. Desweiteren wird verhindert, dass der Prozessor auf ein Resultat aus dem Coprozessor zugreift, bevor dieses zur Verfügung steht. Die Schnittstelle wäre sonst nicht abgesichert gegen RAW-Fehler[51] [HP03].

Während der Core angehalten ist, setzt er seinen Adressbusausgang konstant auf $FFFF und den VMA-Ausgang[52] zum Speicher zurück, damit in der Stall-Phase nicht auf undefinierte Speicheradressen zugegriffen wird.

Diese Eigenschaft kann vom Coprozessor ähnlich wie bei einer DMA-Einheit[53] genutzt werden, um während einer Stall-Phase eigene Speicherzugriffe durchzuführen. Erreichen kann er das durch Multiplexen der Signale, welche die Kommunikation zwischen Core Domain und Memory Domain vollziehen: RWN[54], A (Address), DO (Data Out) und VMA. Dies wird durch den Multiplexer in Abb. 19 dargestellt. Der Core oder der Coprozessor übernehmen die Daten am Eingang DI ausschließlich dann, wenn sie neue erwarten.

Sobald der Coprozessor den Prozessor durch Setzen des STALL-Signals anhält, wird er zum Bus Master und erhält lesenden und schreibenden Speicherzugriff. Der Coprozessor aktiviert den Core durch Aufheben des STALL-Signals erst dann wieder, wenn er sämtliche Datentransfers mit der Memory Domain abgeschlossen hat und neue Instruktionen empfangen werden können.

Auch bei dieser Schnittstelle können die vom Prozessor angebotenen LD/ST-Speicherzugriffsoperationen inklusive sämtlicher Adressmodi genutzt werden. Für das Schreiben auf den Speicher ist dann zusätzlich zum STALL-Signal das DW_STB-Signal ein Kontrolleingang für den Datenmultiplexer, ähnlich dem in Abb. 16.

---

[51] RAW (Read After Write) Fehler: Eine Instruktion *j* versucht eine Quelle zu lesen, bevor die vorherige Instruktion *i* sie beschrieben hat.

[52] VMA (Valid Memory Access): Das Signal zeigt an, ob der aktuelle Buszyklus gültig ist.

[53] vgl. [Schm78], S.58ff

[54] RWN (Read Write Negative): Signalisiert, ob ein Lese- oder Schreibzugriff auf den Speicher vorliegt.

Bei der Berechnung der Latenz eines asynchronen Interrupts ist zu berücksichtigen, dass gegenwärtig eine zyklenreiche Coprozessorinstruktion in Bearbeitung sein könnte. Die Interrupt-Service-Routine kann somit erst dann durchgeführt werden, wenn die Coprozessorinstruktion vollständig beendet ist.

Das sequentielle Interface ermöglicht das Anschließen mehrerer Coprozessoren und unterstützt bereits mehr Algorithmen der digitalen Signalverarbeitung als die in Unterpunkt 5.1.2 vorgestellte Lösung. Inwiefern sie auch zu einer Flächenoptimierung beitragen kann, wird am Beispiel des CPCIPH im folgenden Abschnitt erörtert.

Der CPCIPH mit sequentieller Instruktionsbearbeitung. Die Literatur führt einige Coprozessoren auf, die in wenigen Taktzyklen eine Verschlüsselung mit dem AES durchführen, u. a. die in [Hel01] oder [Kuo01]. Ruft man sich jedoch eines der Hauptziele aus Unterpunkt 2.4.3 in Erinnerung, so sollte die Gatterzahl des Coprozessors nicht die des Controllerkerns überschreiten.

Einer der wichtigsten Einflussfaktoren für die Designgröße ist üblicherweise die Anzahl der verwendeten Flipflops. Bei der im IMS verwendeten Standardzellenbibliothek C0512 mit 0,5 µm Strukturbreite ist die Fläche eines Flipflops etwa 6-8-mal größer als die eines NAND-Gatters mit zwei Eingängen. Folglich sollte insbesondere die Anzahl der Flipflops so gering wie möglich gehalten werden.[55] Doch alleine die zum Laden des 128-Bit-Datenblocks und des 128-Bit-Schlüssels benötigten Flipflops übersteigen die Anzahl der Controller-Flipflops (diese beträgt 174 Stück beim 3311C). Desweiteren wären Flipflops zum Speichern der Zwischenergebnisse vonnöten.

Die Idee ist nun, Flipflops durch die verstärkte Nutzung des Hauptspeichers zu ersetzen. An dieser Stelle hat das Interface mit sequentieller Instruktionsbearbeitung Vorzüge, da es dem Coprozessor Bus-Master-Fähigkeit verleiht. Zwischenergebnisse werden in einem Scratchpad-Bereich des Arbeitsspeichers anstatt in Coprozessorregistern abgelegt. Somit kann der Algorithmus in kleinen Blöcken sequentiell verarbeitet werden und die Datenblöcke und der Schlüssel müssen nicht mehr vollständig in den Coprozessor geladen werden. Einen etwas

[55] Bei im Rahmen dieser Arbeit entworfenen Designs hat sich die Näherung bestätigt, dass der Logikflächenbedarf ca. 2-4-mal so groß ist wie der durch Flipflops verursachte Flächenbedarf. Somit kann man schon anhand der Anzahl der verwendeten Flipflops eine erste Approximation für den Gesamtflächenbedarf einer Schaltung aussprechen.

detaillierteren Einblick in die Partitionierung der Verschlüsselungsoperation gibt Kapitel 6.

Die Instruktionen des CPCIPH lassen sich in zwei Gruppen unterteilen: Befehle mit dem Core und Befehle mit dem Coprozessor als Bus Master. Unter letztere Gruppe fallen die Instruktionen

- Verschlüsselung eines Blocks (ECR und ECRI[56]),
- Entschlüsselung eines Blocks (DCR und DCRI) und
- Schlüsselexpansion (KE).

Alle diese Instruktionen sind inhärent, erwarten also keine weiteren Operanden. Daten werden über den DI-Bus vom Coprozessor empfangen. Die zur Durchführung der Operationen benötigten Informationen, z.B. die Basisadresse des Eingangsdatenblocks, müssen bereits vorher in den Coprozessor geladen werden.

Dies geschieht ähnlich den LD/ST-Befehlen des CPMAC über das vom Core kontrollierte Interface zum Speicher. Bei diesen werden die eingehenden Daten aus der Memory Domain nicht über den DI-Eingang des Coprozessors (vgl. Abb. 19), sondern über den OPD-Bus übermittelt.

5.1.4 Parallele Instruktionsbearbeitung

Die im letzten Unterpunkt beschriebene Schnittstelle ermöglicht bereits die Umsetzung verschiedenster Algorithmen der digitalen Signalverarbeitung in Hardware. Benötigt der Coprozessor jedoch sehr viele Taktzyklen zur Ausführung einer Instruktion, wie dies z.B. bei der Verschlüsselung von Daten mit dem CPCIPH der Fall ist, so kann der Controller während dieser Zeit keine weiteren Instruktionen ausführen. Auch eventuell auftretende Interrupts werden erst nach Beendigung der Coprozessorinstruktion bearbeitet.
Dieser Unterpunkt diskutiert, wie der Core *parallel* und unabhängig zu einem aktiven Coprozessor die nachfolgenden Instruktionen aus dem Speicher laden kann.

[56]ECRI (Encrypt and Auto-Increment BA Registers): Falls große zusammenhängende Datenblöcke zu verschlüsseln sind, bietet es sich an, dass nach Ausführen der Instruktion die BA-Register automatisch um die Größe eines Blocks weitergesetzt werden.

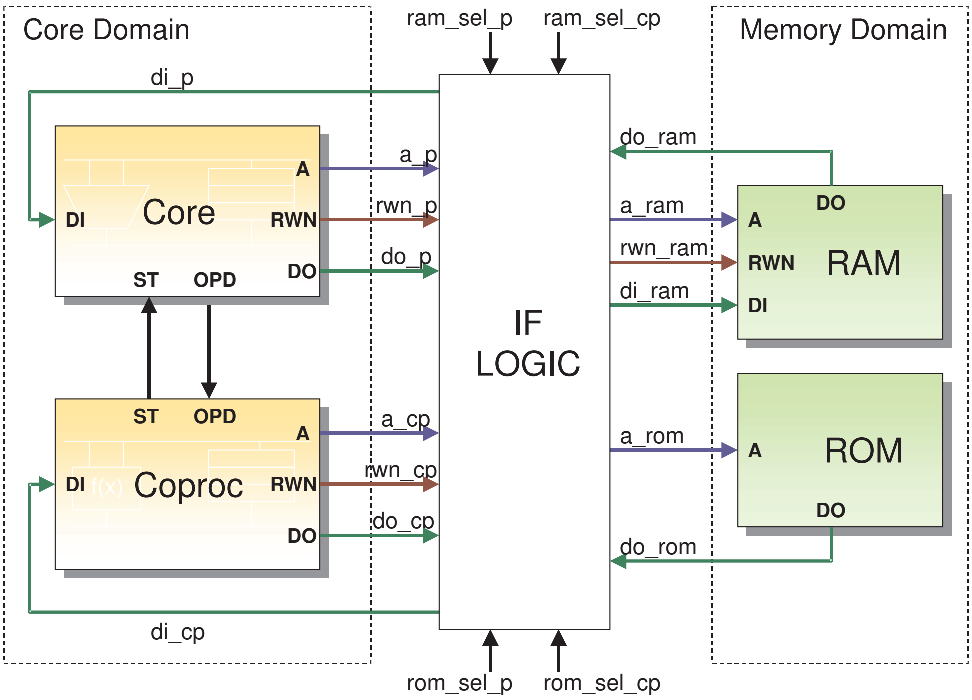


Abbildung 20: Parallele Instruktionsbearbeitung mit getrenntem RAM- und ROM-Bus.

Eine mögliche Umsetzung des parallelen Interface wird in Abb. 20 dargestellt. Die Memory Domain unterteilt sich in den RAM- und ROM-Bereich (bei den Signalen erkennbar an der Endung `_ram` bzw. `_rom`), die beide getrennt voneinander sowohl vom Core als auch vom Coprozessor angesteuert werden können. Erreicht den Coprozessor eine an ihn gerichtete Instruktion, so hält er den Core *nicht* an. Stattdessen kann er parallel zum Prozessor Datentransfers mit RAM oder ROM durchführen.

Wollen sowohl Core als auch Coprozessor gleichzeitig auf denselben Speicher zugreifen, so tritt ein Konflikt innerhalb der Core Domain auf. Bei Konflikten hat stets der Prozessorkern die höchste Priorität, der Coprozessor stellt für diesen Takt seinen eigenen Anspruch auf den Bus zurück. Im kommenden Takt versucht er den Speicherzugriff erneut. Sollte ein System mehrere Coprozessoren besitzen, die selbständig Speicherzugriffe ausführen, so muss eine Priorisierung der einzelnen Coprozessoren vorgesehen werden.

Falls der Coprozessor zwischenzeitlich einen Memory Zugriff des Cores abwarten muss, behält er sämtliche Registerinhalte bei.

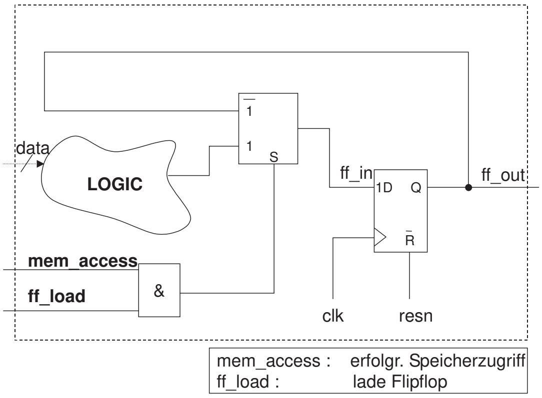


Abbildung 21: Flipflop mit bedingtem Lasteingang.

Erst bei freigegebenem Speicherzugriff werden die Register mit den neuen Werten geladen. Sämtlichen Flipflops sind deswegen AND-Gatter vorgeschaltet, um die Sicherung der Logikausgänge nur dann zu bewirken, wenn im aktuellen Taktzyklus vom Coprozessor angeforderte Daten durch den Speicher bereitgestellt oder zu schreibende Daten vom Speicher übernommen werden können. Die modifizierte Schaltung am Eingang des Flipflops wird in Abb. 21 dargestellt. Durch die Absicherung durch das `mem_access` Signal wird vermieden, dass der Coprozessor weiterrechnet, obwohl die für ihn bestimmten Daten aus dem Speicher noch nicht vorhanden sind.

Tritt während der Ausführung einer Coprozessorinstruktion ein Interrupt auf, so kann dieser vom Core behandelt werden, während der Coprozessor parallel zur ausgeführten Interruptroutine weiterarbeitet. Speicherzugriffe des Controllers und damit auch von Interruptroutinen haben immer Vorrang vor denen des Coprozessors.

Der Beendigungszeitpunkt einer Coprozessorinstruktion ist im Vergleich zur sequentiellen Instruktionsbearbeitung nicht eindeutig vorhersagbar, insbesondere wegen einer möglichen Unterbrechung des normalen Programmflusses durch asynchrone Interrupts. Aus diesem Grund werden zwei neue Instruktionen eingeführt:

- BBY (Branch On Busy) und
- BNB (Branch on Not Busy).

Diese ermöglichen einen konditionalen relativen Sprung nach einer Abfrage des Coprozessorstatus.

Coprozessor und Prozessor sollten im parallelen Betrieb nicht auf demselben RAM-Bereich arbeiten, falls einer der beiden schreibend auf diesen Speicherbereich zugreift. Für den Prozessor ist das Schreiben in den vom Coprozessor genutzten Speicherbereich erst dann wieder zulässig, wenn eindeutig ist, dass dieser seine letzte Instruktion beendet hat, z.B. durch Überprüfen von dessen Bearbeitungsstatus mit einer BNB- oder BBY- Instruktion.

Es muss berücksichtigt werden, dass auch der Sprungbefehl Speicherzugriffe durchführt. Steht das Programm im ROM, so könnte eine Sprunginstruktion ausschließlich ROM-Zugriffe bewirken. Sollte der Coprozessor genau auf einen ROM-Zugriff warten, so steckt das System in einer Endlosschleife, aus der es lediglich durch einen Interrupt befreit werden kann. Es muss bei der parallelen Lösung vom Assemblerprogrammierer beachtet werden, dass der Prozessor nicht durchgehend Zugriffe auf nur einen der beiden Speicher durchführt. Eine Umgehungsmöglichkeit dieses Problems wird in Unterpunkt 5.1.5 erläutert.

Der Einsatz der parallelen Coprozessorschnittstelle ist vornehmlich bei folgenden Coprozessortypen vorteilhaft:

- Logiklastige Coprozessoren mit großer Zyklenzahl für die Spezialinstruktionen und wenigen Speicherzugriffen,
- Coprozessoren mit vielen RAM-Zugriffen, falls das Programm des Cores im ROM liegt (oder umgekehrt),
- Coprozessoren, deren Instruktionen geringe Priorität haben gegenüber den Core-Befehlen oder selten aufgerufen werden.

5.1.5 Hybride Lösung

Möchte man sowohl die Vorzüge der sequentiellen als auch die der parallelen Lösung nutzen, so kann man beide Schnittstellen zu einer Lösung, namentlich der hybriden, weiterentwickeln. Um zwischen den beiden Schnittstellentypen zu wechseln, werden zwei neue inhärente Instruktionen eingeführt:

- SSI (Set Sequential Interface) und
- SPI (Set Parallel Interface).

Diese verändern das Register `SPF` (Sequential/Parallel Flag) des Coprozessors, welches die Information trägt, ob das Interface im sequentiellen oder im parallelen Modus arbeitet. Der zusätzliche Hardwareaufwand ist minimal, da hauptsächlich nur ein Flipflop für das SPF und Logik zum Dekodieren der beiden Instruktionen benötigt wird. Sollen dem Programmierer sämtliche Freiheiten für die Gestaltung der Schnittstelle überlassen werden, so ist dies mit Einführung der zwei neuen Instruktionen sehr kostengünstig möglich.

Es ist hiermit auch möglich, bereits sehr lange vom Coprozessor bearbeitete Instruktionen durch die SSI-Instruktion mit höherer Priorität zuende zu bringen, indem der Core unmittelbar angehalten wird. Eine sequentiell ausgeführte Instruktion kann selbstverständlich nicht unterbrochen werden, auch nicht durch den SPI-Befehl.

Dieses Interface ist optimal im Hinblick auf seine universelle Einsetzbarkeit für verschiedenste Algorithmen der digitalen Signalverarbeitung.

Der CPCIPH mit hybrider Schnittstelle. Zusätzlich zum CPCIPH mit sequentieller Instruktionsbearbeitung wurde im Rahmen dieser Arbeit auch ein CPCIPH mit hybrider Schnittstelle implementiert. Dieser ist mit den zusätzlichen Instruktionen SSI, SPI, BNB und BBY ausgestattet.

Der Assemblercode in Beispiel 2 soll den Einsatz des Interfaces veranschaulichen. Das Programm führt unter dem Label `_ecr_example` die Verschlüsselung eines Datenblocks parallel zum Core durch. Der Core bearbeitet in der Zwischenzeit unterhalb des Labels `_do_sth_else` beliebige andere Aufgaben. Falls diese Subroutine zehnmal aufgerufen wurde, wird die Warteschleife abgebrochen und eine Umschaltung in den sequentiellen Modus erfolgt. D.h. die Verschlüsselungsoperation wird priorisiert beendet, bevor der Prozessor mit der nächsten Chiffrierung fortschreiten kann.

Im Vergleich zur sequentiellen Schnittstelle benötigt die hybride Lösung 208 Gatteräquivalente (GÄ[57]) mehr. Davon werden 100 GÄ für die aufwendigere Interfacelogik zur Steuerung der getrennten RAM- und ROM-Busse verwendet. Die übrigen 108 GÄ beansprucht der Coprozessor intern vornehmlich für die Blöcke Decoder/IF und CU. Insgesamt wächst der Aufwand für den CPCIPH und sein Interface bei der Hybridschnittstelle um ca. 10% gegenüber der sequentiellen Lösung an.

[57] Die Fläche eines Chips wird üblicherweise in GÄ umgerechnet. Dabei entspricht ein GÄ einem Nand-Gatter mit zwei Eingängen und einem Ausgang.

```
_ecr_example:
   ...
   CLRA                 ; Lösche Inhalt von Akku ACCA.
   SPI                  ; Core und Coproz. arbeiten parallel.
   ECRI                 ; Verschlüssel einen Datenblock.

_loop_wait:
   JSR  _do_sth_else    ; Führe andere Berechnungen durch.
   INCA                 ; ACCA = ACCA + 1
   CMPA #10             ; Schon 10X _loop_wait?
   BEQ _finish_loop     ; Dann beende die Schleife.
   BBY _loop_wait       ; ECRI noch aktiv: zurück zu _loop_wait.
   JMP _ecr_example     ; Starte nächste ECRI Instruktion.

_finish_loop:
   SSI                  ; Umschalten auf sequentiellen Modus.
   JMP _ecr_example     ; Springe an den Anfang.

_do_sth_else:
   ...
   RTS                  ; Beende das Unterprogramm.
```

Beispiel 2: Beispielcode für das Hybrid-Interface.

5.2 Memory-Mapped-Interface

Eine weitere Möglichkeit der Schnittstellenumsetzung ist die Abbildung der Kontroll-, Status- und BA-Register des Coprozessors in den vom Prozessor adressierbaren Speicherbereich. Somit ähnelt der Coprozessor einer Prozessorperipherie, da er über Adressen angesprochen wird. Nach der kurzen Vorstellung anderer Architekturen, die ihre Coprozessoren über eine Memory-Mapped-Schnittstelle adressieren, wird eine diesbezügliche Beispielimplementierung des CPCIPH erläutert.

Die Memory-Mapped-Schnittstelle des kryptographischen Coprozessors des MCF5235 von Motorola hat bereits Unterpunkt 2.2.3 erklärt.

Der *MSP430* Mikrocontroller von TI ist ein 16-Bit-Controller auf der Basis einer von-Neumann-Architektur. Er verfügt über einen Hardware-Multiplizierer, dessen Register im Adressraum des Controllers liegen. Der MSP430 unterstützt so-

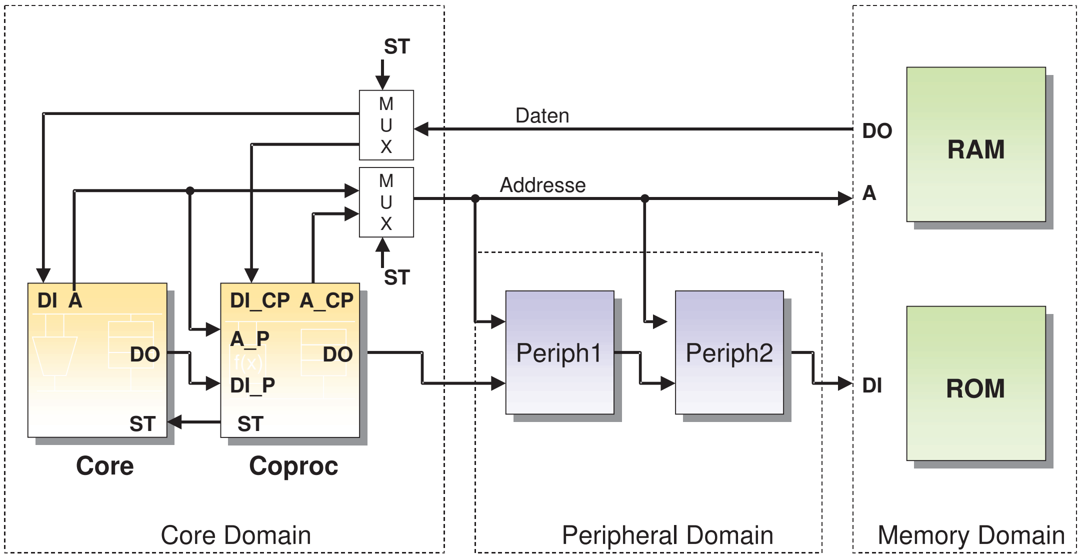


Abbildung 22: Memory-Mapped Coprozessorschnittstelle.

wohl vorzeichenlose also auch vorzeichenbehaftete Multiplikationen sowie MAC-Operationen, wahlweise mit 8- oder 16-Bit-Operanden.

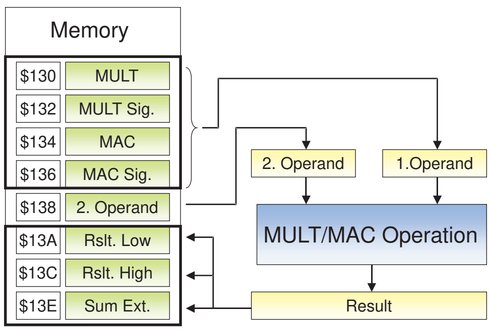


Abbildung 23: Schnittstelle des MSP430.

Abb. 23 zeigt die Umsetzung der Schnittstelle des MSP430. Die Operation wird automatisch ausgelöst, wenn nacheinander der erste und der zweite Faktor in die Operandenregister geschrieben wurden. Dabei entscheidet die Adresse des ersten Operanden darüber, welche Operation durchgeführt wird. Die Adresse des zweiten Faktors ist konstant.

Memory-Mapped-Schnittstellen sind immer dann von Vorteil, wenn der Prozessorkern keine eigene Coprozessorschnittstelle mit dedizierten Coprozessorinstruktionen anbietet.

In Figur 22 ist ein solches Memory-Mapped-Interface für ein IMS3311-System dargestellt. Der Coprozessor erhält über `A_P` (Address from Processor) und `DI_P` (Data In from Processor) wie die übrigen Peripherien im Ringbus Adressen und Daten. Liegt die Adresse bei `A_P` im für ihn bestimmten Adressbereich und ist der aktuelle Taktzyklus ein Schreibzyklus, dann übernimmt der Coprozessor die

Daten. Bei einem Lesezyklus gibt der Coprozessor die Daten aus der angesprochenen Adresse an die nächste Peripherie im Ringbus weiter, sofern diese in seinem Adressbereich liegt. Andernfalls werden die eingehenden Daten bei DO weitergegeben.

Für Coprozessoren wie den CPMAC, deren Speicherzugriffe vom Controllerkern durchgeführt werden, ist dieses Interface bereits ausreichend. Würde ein Coprozessor ähnlich dem CPCIPH Speicherzugriffe veranlassen, so müssten Daten- und Adressregister gemultiplext werden, abermals wie bei der sequentiellen Instruktionsbearbeitung mit der STALL-Leitung als Steuersignal. Dies wird in Abbildung 22 innerhalb der Core Domain veranschaulicht.
Der Coprozessor hat also zwei Dateneingänge, einen vom Prozessor (DI_P) und einen direkt vom Speicher (DI_CP). Das folgende Beispiel stellt eine Memory-Mapped-Schnittstelle mit sequentieller Instruktionsbearbeitung vor.

Eine Memory-Mapped CPCIPH-Einheit. Der CPCIPH belegt für dieses Beispiel den Adressbereich $20 – $24 im Speicher, kann aber an beliebiger Stelle im Speicher liegen. Dabei sind vier Bytes im Adressraum durch die Basisadressen belegt:

Addr/	Bit	7	6	5	4	3	2	1	0
$20:	IOH	Bit15	Bit14	...	...	...	...	Bit9	Bit8
$21:	IOL	Bit7	...	...	...	...	...	...	Bit0
$22:	KEYH	Bit15	...	...	...	...	...	...	Bit8
$23:	KEYL	Bit7	...	...	...	...	...	...	Bit0

Die beiden IO-Bytes geben den Adressoffset des Eingangs- und Ergebnisblocks einer Ver- oder Entschlüsselung wieder, während die KEY-Bytes in Adresse ($22 – $23) auf die Position des symmetrischen Schlüssels im Speicher zeigen. Dies unterscheidet die Memory-Mapped-Schnittstelle des CPCIPH von der Umsetzung beim Motorola ColdFire, der die Daten über eine FIFO-Schnittstelle in den Coprozessor lädt.

Die dereferenzierende Lösung für den CPCIPH ist flexibler, da die Daten nicht im Speicher verschoben werden müssen, sondern stattdessen nur der Zeiger auf die Daten umgesetzt wird.

Die Bytes sind im 3311C typischen Big-Endian-Format [HP03] im Speicher abgelegt, in dem auf der niedrigeren Adresse das MSB liegt.[58]

Zusätzlich zu den BA-Registern verfügt der Coprozessor über ein weiteres auf die Adresse $24 projiziertes Byte, das die Kontrollbits enthält. Sie sind in folgender Tabelle abgebildet:

Addr/	Bit	7	6	5	4	3	2	1	0
$24:	AES_CTRL	-	-	-	-	DCR	ECR	INC	KE

Eine Schlüsselexpansion erfolgt z.B. genau dann, wenn man mittels eines Store-Befehls das KE-Bit setzt. Bei gesetztem DCR wird dementsprechend ein Datenblock dechiffriert, bei ECR ein Block encodiert und bei INC wird ein BA-Registerinkrement nach vollzogener (De-) Chiffrierung durchgeführt.

Der Coprozessor kann gleichzeitig gesetzte DCR-, ECR- und KE-Kontrollbits nicht parallel bearbeiten. Eine Priorisierung der Bits gewährleistet, dass immer nur eine Operation pro Store-Befehl ausgeführt wird. Nach der Operation findet das Zurücksetzen der Register auf Null statt, sodass der Controller bei einem Load-Befehl auf Adresse $24 stets $00 zurückliest. Adresse $24 zeigt somit auf ein Write-Only-Byte im Speicher.

[58] Die Endung H steht dabei für High Byte, L steht für Low Byte.

6 Optimierung durch Algorithmische Analyse

Die vorangegangenen Kapitel beschäftigten sich mit der Bestimmung einer Architektur für die Umsetzung von Signalverarbeitungskomponenten und der Vorstellung möglicher Schnittstellenimplementierungen zwischen Controller und Coprozessor. In diesem Kapitel wird die Partitionierung der ausgewählten Algorithmen untersucht. Diese ermöglicht eine gatterminimierte Abbildung der Algorithmen auf die erarbeitete Architektur.

Der erste Unterpunkt erläutert die Vorgehensweise der algorithmischen Analyse, während nachfolgend die Umsetzung für die ausgewählten Beispielalgorithmen erklärt wird.

6.1 Prinzip der Partitionierung

Die algorithmische Analyse verfolgt hauptsächlich die Idee, eine logische sowie zeitliche Struktur in komplexe Abläufe zu bringen. Sie erfolgt in drei Schritten:

1. Unterteilung des Algorithmus in logische *Operationen*,
2. Abbildung der Operationen auf Hardware-*Komponenten* und
3. Festlegen einer *Ablaufsteuerung* für die extrahierten Komponenten.

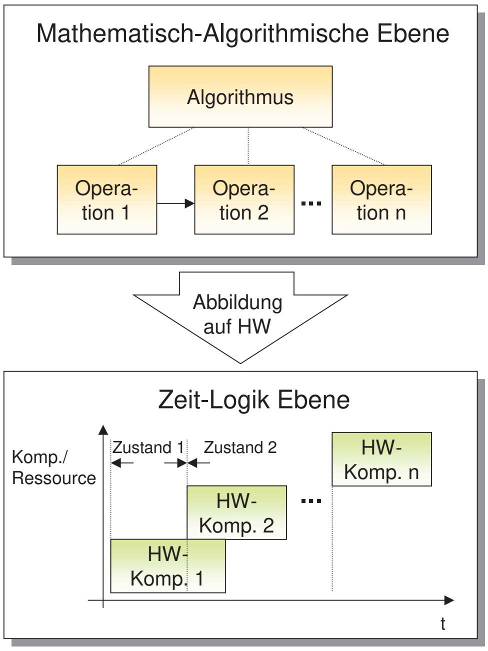


Abbildung 24: Abbildung eines Algorithmus auf Hardware.

Eine Komponente bezeichnet in den folgenden zwei Kapiteln die Realisierung einer Operation in Hardware. Abb. 24 veranschaulicht die hier eingeführten Schritte und Begriffe.

Zur Optimierung des Durchsatzes der Hardware wird der Algorithmus derart partitioniert, dass die Operationen in hohem Maße parallel oder gepipelinet bearbeitet werden können.

Ist hingegen die Flächenminimierung des Chips das favorisierte Designziel, so wird versucht, eine möglichst hohe Wiederverwertbarkeit der in Hardware umgesetzten Operationen zu erreichen. Die Komponenten können in verschiedenen Taktzyklen mehrfach benutzt werden.

Bereits zu Beginn dieser Partitionierung muss die Entscheidung getroffen werden, ob man den *gesamten* Algorithmus durch dedizierte Hardware (HW) beschleunigt oder nur dessen besonders rechenaufwendige Bestandteile auf Einzelinstruktionen abbildet. Tabelle 1 führt die Vor- und Nachteile der beiden Strategien auf. Für den AES wird eine vollständige Hardwarelösung gewählt.

Kriterium	HW-Lösung	HW/SW-Lösung
Anwendungsbeispiel	AES	Vektormult.
Beispielcoprozessor	CpCiph	CpMac
Kontrollstruktur	in HW	in SW
Instruktionen pro Algorithmus	wenige	viele
Taktzyklen pro Instruktion	viele	wenige
Flexibilität bzw. Retargierbarkeit	gering	groß
Programmierkomfort	hoch	mäßig

Tabelle 1: Vergleich der Algorithmenumsetzung.

Der Algorithmus besteht aus vielen sehr unterschiedlichen Einzeloperationen, die aneinandergereiht durchgeführt werden. Eine sequentielle Ausführung in Software (SW) ist außer für die Schritte `MixColumns` und `InvMixColumns` in wenigen Taktzyklen möglich. Um dennoch eine signifikante Beschleunigung zu erreichen, werden die Einzelschritte parallelisiert und durch eine effiziente festverdrahtete Steuerlogik kontrolliert.
Lee [Lee01] stellt ein Konzept vor, besonders aufwendige Operationen vieler kryptographischer Algorithmen (z.B. die Permutation auf Bitebene), auf dedizierte Instruktionen abzubilden. Somit lassen sich unterschiedliche Algorithmen durch Nutzung derselben Spezialinstruktionen beschleunigen. Es sei jedoch hingewiesen, dass die von Lee vorgestellten Befehle beim AES keine signifikante zeitliche Einsparung bewirken würden.

Da Vektormultiplikationen und digitale Filter ähnliche Operationen nutzen, spielt hier eher die Flexibilität und Retargierbarkeit, also die Anwendbarkeit auf verschiedene Zielstrukturen, eine bedeutende Rolle. Es wird versucht, mit *einer*

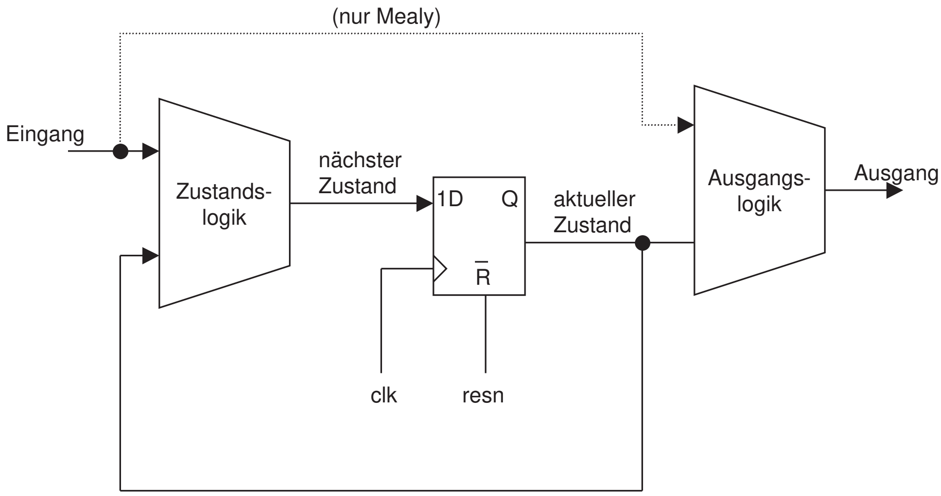


Abbildung 25: Mealy- und Moore-Automat.

Architektur unterschiedliche Filterstrukturen zu unterstützen. Mit Hilfe eines Assemblerprogramms erfolgt die zeitliche Steuerung der Komponenten und setzt somit die extrahierten Operationsrealisierungen in eine zeitliche Abhängigkeit.

Die Abbildung einer Ablaufsteuerung auf *Hardware* geschieht durch einen endlichen Automaten, auch endliche Zustandsmaschine oder Finite State Machine (FSM) genannt.[59] Die einzelnen Zustände der FSM initiieren dabei die Komponenten, welche die Operationen durchführen. In dieser Arbeit werden sowohl

- Mealy-Automaten als auch
- Moore-Automaten

verwendet. Diese unterscheiden sich darin, dass die Ausgangsfunktion des Mealy-Automaten vom aktuellen Zustand *und* vom Eingang (vgl. gepunktete Linie in Abb. 25), beim Moore-Automaten ausschließlich vom aktuellen Zustand abhängt.

Falls einzelne Operationen eine hohe Komplexität aufweisen, werden sie in logische bzw. zeitliche Suboperationen zerlegt. Gerade unter dem Aspekt der Wiederverwertbarkeit, die für die Flächenoptimierung eine entscheidende Rolle spielt, ist diese Subpartitionierung interessant. Die einzelnen Subkomponenten können dann für verschiedene Operationen eingesetzt werden.

[59] vgl. [Wen74], S.4ff und [Schi01], S.258ff

6.2 Partitionierung des AES

Der folgende Unterpunkt thematisiert die Aufteilung des AES-Algorithmus hinsichtlich einer flächenoptimierten Umsetzung des CPCIPH-Coprozessors. Anschließend werden die dabei extrahierten Komponenten in eine zeitliche Reihenfolge gesetzt.

6.2.1 Unterteilung in Komponenten

Die Unterteilung des AES in Komponenten geschieht in Anlehnung an die in Unterpunkt 3.3.2 aufgeführten Einzelschritte. Diese fasst Tabelle 2 zusammen.

Chiffrieren	Dechiffrieren	Schlüsselexpansion
`AddRoundKey`	`AddRoundKey`	`Byte Rotation`
`SubBytes`	`InvSubBytes`	`SubBytes`
`ShiftRows`	`InvShiftRows`	`Rundenk. Add.` $\mathcal{RC}$
`MixColumns`	`InvMixColumns`	`Schlüsselbyteadd.`

Tabelle 2: Zusammenfassung der Einzelschritte des AES.

Bei der Abbildung des Algorithmus auf Hardware werden zusätzlich `DataFetch` zum Lesen von Daten aus RAM und ROM und entsprechend `WriteBack` zum Zurückschreiben in das RAM benötigt. Ihre Funktionalität beschränkt sich auf Datentransfers mit dem Speicher, die mit den in den Unterpunkten 5.1 und 5.2 beschriebenen Schnittstellen realisiert werden können.

Die Auseinandersetzung mit dem AES hat ergeben, dass die ursprünglich vorgesehene parallele Verarbeitung von 128 Bit auf 32 Bit reduzierbar ist und damit signifikant Gatter eingespart werden können. Dies wird u.a. auch in einer FPGA-Umsetzung von Chodowiec genutzt [Cho03]. Nachfolgend seien die Bezeichnungen der bitreduzierten Komponenten durch das Postfix `32` markiert. Die Vorgehensweise zur Reduzierung der Komponentengröße wird in den kommenden Abschnitten erläutert.

Es sei nochmals darauf verwiesen, dass die 32 Datenbits vor einer Rundentransformation aus dem Speicher geladen werden und nach einer Runde in einen Scratchpad-Bereich des Speichers zurückgeschrieben werden. Durch diese Auslagerung wird eine wesentliche Reduzierung der Flipflop-Anzahl erreicht.

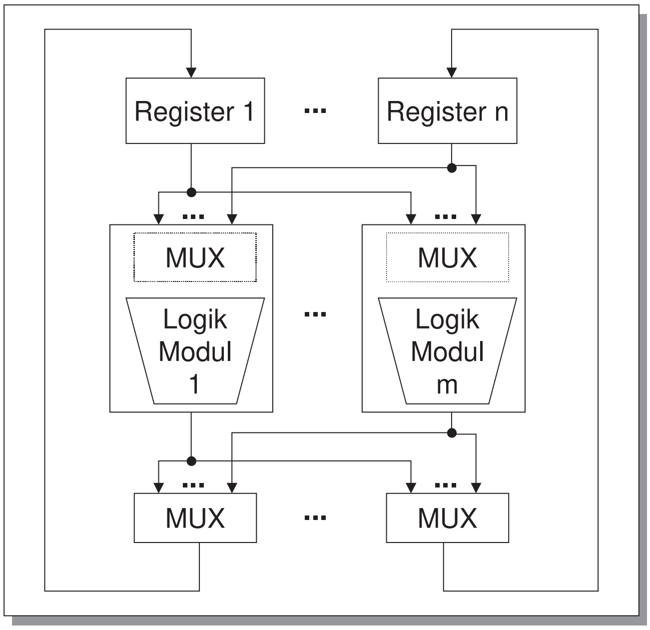


Abbildung 26: Wiederverwertung von Komponenten und Flipflops.

Die Minimierung der Gatter geschieht beim CPCIPH hauptsächlich über eine Verringerung der Flipflopzahl. Nach jedem Takt findet eine Zwischenspeicherung der Ergebnisse aus den Komponentenberechnungen in den gemeinsamen GP-Registern statt. Dadurch erreicht man eine gezielte Wiederverwertung der Register-Flipflops. Diese Methodik veranschaulicht Abb. 26.

Um die Anzahl GP-Register zu minimieren, werden die einzelnen Komponenten hinsichtlich der minimal benötigten Anzahl Ein- und Ausgangsbytes untersucht, welche die Register zwischenspeichern müssen.

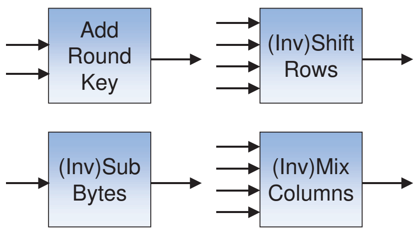


Abbildung 27: Komponenten-IOs für En- und Decryption.

Abb. 27 stellt dar, wieviele Eingangsbytes jede der Operationen benötigt, um ein Ausgangsbyte zu berechnen. Anhand der vereinfachenden Darstellung erkennt man, dass die Komponenten `MixColumns32` und `InvMixColumns32` sowie `ShiftRows32` und `InvShiftRows32` vier Eingangsbytes benötigen, um *genau* ein Ausgangsbyte zu erzeugen. Insbesondere erstere sind rein kombinatorisch durchführbar und damit nicht auf Speicherzu-

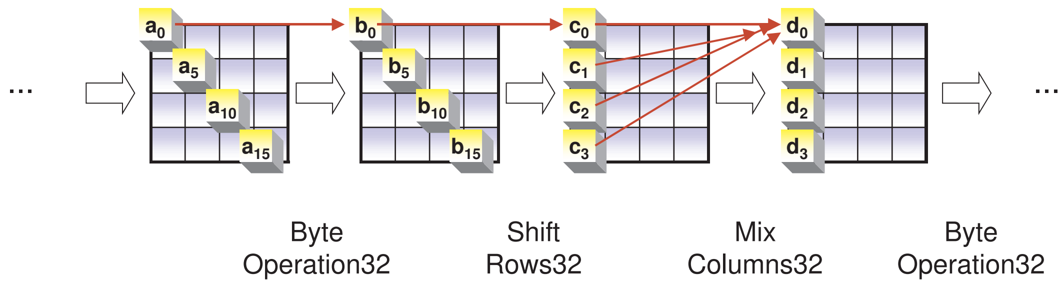


Abbildung 28: Ausschnitt einer Rundenberechnung auf 32-Bit-Wortebene.

griffe angewiesen. Um neben den Datenbuszugriffen nicht einen weiteren zeitlichen Engpass im eigentlichen Logikpfad hervorzurufen, sollten diese Komponenten in möglichst wenigen Taktzyklen berechnet werden.

Auffällig ist, dass dieselben vier Eingänge bei `ShiftRows32` und `MixColumns32` genutzt werden können, um drei weitere Ausgangsbytes zu berechnen: Während die erstgenannte Komponente die vier Bytes einer Zustandszeile eines AES-Datenblocks verwendet, benötigt letztere jeweils vier Spaltenbytes.

Im Gegensatz zu `MixColumns32` verändert `ShiftRows32` die Werte der Bytes nicht, sondern vertauscht lediglich die einzelnen Bytes untereinander. Falls die `DataFetch32`-Komponente die Daten gezielt herunterlädt, verschiebt `ShiftRows32` diese so, dass `MixColumns32` die benötigten Eingangsdaten zur Verfügung hat.
Abb. 28 veranschaulicht, dass die Bytes an den Stellen 0, 5, 10 und 15 nach dem `ShiftRows32`-Schritt an den Positionen 0, 1, 2 und 3 liegen, die damit die Eingänge zur Berechnung der Ausgangsbytes d_0, d_1, d_2 und d_3 darstellen. Der `ShiftRows32`-Schritt wird vorgezogen, indem `DataFetch32` direkt die vier Bytes aus dem Speicher lädt, die später als Eingänge der Spaltenoperation dienen. Auch für die übrigen drei `MixColumns`-Spalten kann `ShiftRows32` vorgezogen werden (vgl. [Cho03]).

Bei der Dechiffrierung kehrt sich die Reihenfolge der Operationen um. Damit wird die `InvShiftRows32`-Komponente in die Adressberechnung von `WriteBack32` integriert. Jede Runde wird also in vier 32-Bit-Pakete[60] unterteilt, die sequentiell bearbeitet werden können. Abb. 29 zeigt alle Komponenten für die Encrypt- und Decrypt-Operationen des CPCIPH. Die vier bisher ermit-

[60]Die Pakete werden von hier an wegen der `MixColumns` und `InvMixColumns`-Operation Spalte genannt.

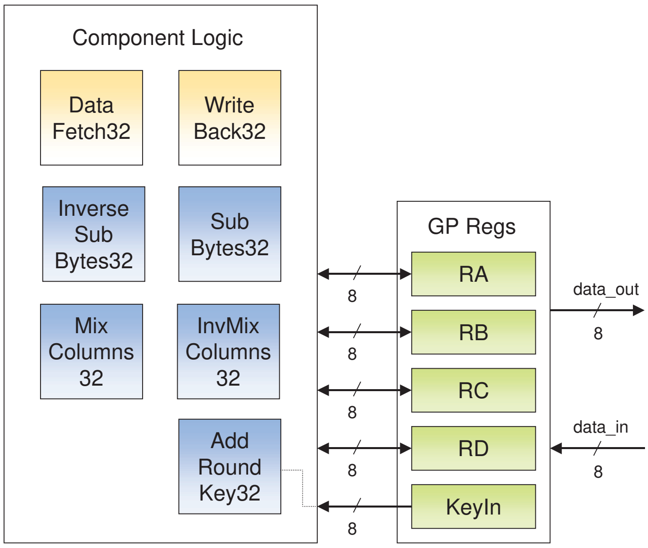


Abbildung 29: Logikkomponenten und Register des CPCIPH für En- und Decryption.

telten Register werden um ein Register zum Speichern des Rundenschlüssels ergänzt. Der CPCIPH benötigt folglich nur fünf 8-Bit breite GP-Register.

6.2.2 Ablaufsteuerung des CPCIPH

Nach der Partitionierung des AES im letzten Abschnitt, setzt sich dieser Unterpunkt nun mit der Ablaufsteuerung der Verschlüsselung, Entschlüsselung und Schlüsselexpansion auseinander.

Vorab sei jedoch die Zusammenstellung einer Runde neu definiert. Im Gegensatz zu der üblichen Darstellung in der Literatur ([Dae99], [Dae02], [Fips01]) und in Unterpunkt 3.3.1 wird der Algorithmus in Bsp. 3 äquivalent dargestellt. Damit rückt bei der Verschlüsselung die erste Schlüsseladdition in die erste Runde. Durch die Umstellung unterscheidet sich bei der Chiffrierung die letzte Runde lediglich durch die letzte Operation (`AddRoundKey` statt `MixColumns`) von den übrigen Runden. Bei der Dechiffrierung muss dementsprechend zwischen der ersten und den anderen Runden differenziert werden. Die Umstellung bewirkt eine größere Regularität.

Eine Runde setzt sich aus vier Spalten zusammen, die in Hardware sequentiell bearbeitet werden. Die Speicherzugriffs-Komponenten `DataFetch32` und `WriteBack32` rahmen eine Spaltenberechnung ein.

```
//++++++++++++++++ ENCRYPTION ++++++++++++++++//
State = Plaintext;
for(i=0; i<9; i=i+1)
{
   AddRoundKey(State, ExpKey[i]);
   SubBytes(State);
   ShiftRows(State);
   MixColumns(State);
 }
AddRoundKey(State, ExpKey[9]);
SubBytes(State);
ShiftRows(State);
AddRoundKey(State, ExpKey[10], Ciphertext);

//++++++++++++++++ DECRYPTION ++++++++++++++++//
AddRoundKey(Ciphertext, InvExpKey[0], State);
InvShiftRows(State);
InvSubBytes(State);
AddRoundKey(State, InvExpKey[1]);
for(i=1; i<10; i=i+1)
{
   InvMixColumns(State);
   InvShiftRows(State);
   InvSubBytes(State);
   AddRoundKey(State, InvExpKey[i+1]);
}
Plaintext = State;
```

Beispiel 3: Äquivalente Umformung des AES Algorithmus.

Zustände des globalen Automaten. Der globale Automat symbolisiert die übergeordnete Zustandsmaschine, welche die Einzelkomponenten initiiert. Diese erhalten bei Bedarf weitere lokale Ablaufsteuerungen.

Die Komponenten des CPCIPH werden sukzessive durch zugehörige Zustände initiiert, d.h. `AddRoundKey32` bearbeitet den `DoAddRoundKey`-Zustand, etc. Abb. 30 stellt dieses getrennt für Verschlüsselung, Entschlüsselung und Schlüsselexpansion auf der linken Seite dar. Die Pfeile entsprechen den Zustandsübergängen. Ziel ist es, diese drei Zustandsmaschinen zu einer zusam-

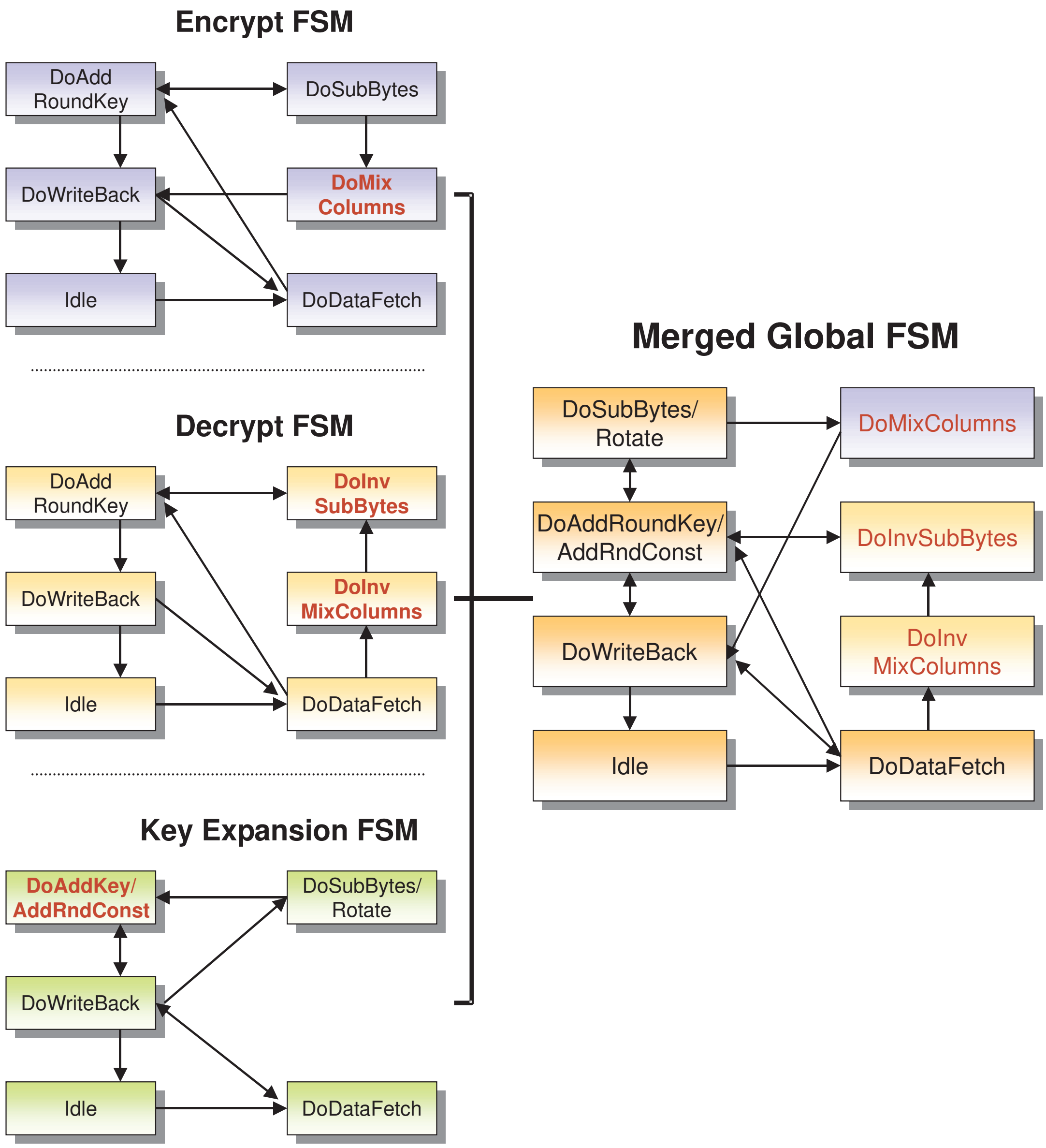


Abbildung 30: Der Globale Endliche Automat des CPCIPH.

menzufassen. Aus Hardwaresicht sind dabei Automaten mit der Eigenschaft

$$n_s = 2^{n_{ff}} \tag{6.1}$$

besonders günstig, wobei $n_s \in \mathbb{N}$ die Anzahl der Zustände und $n_{ff} \in \mathbb{N}$ die Anzahl der benötigten Flipflops meint. Bei diesen FSMs gibt es keine ungenutzten Zustände.

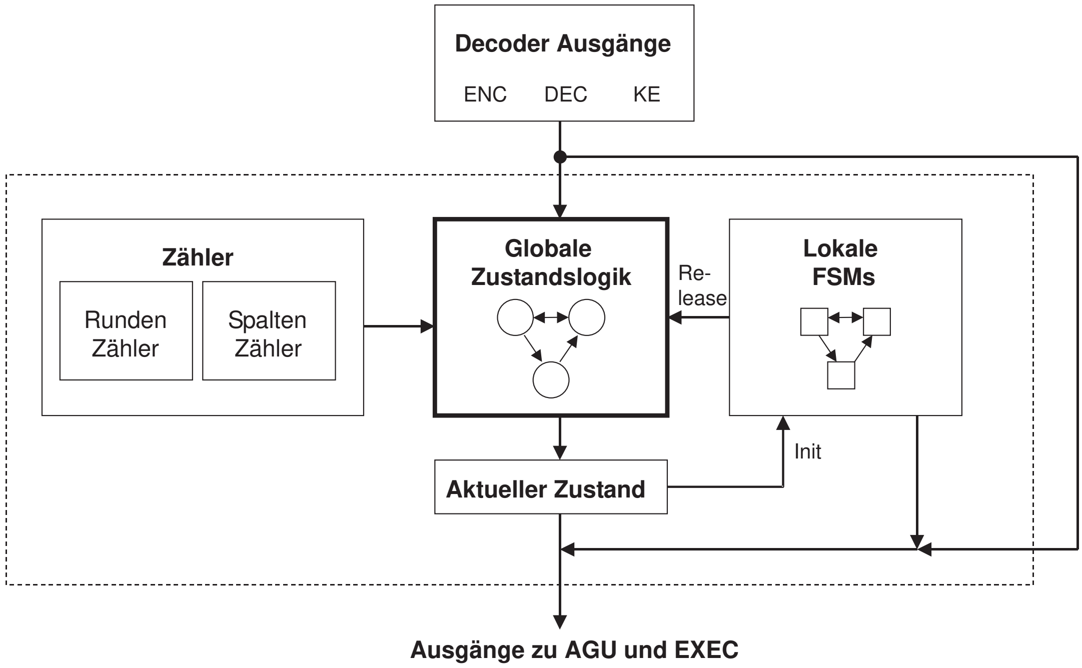


Abbildung 31: Eingänge und Ausgänge der globalen FSM.

Die rechte Seite von Abb. 30 zeigt die finale globale Ablaufsteuerung. Durch die Schlüsselexpansion kommen zu den bisher bestimmten Zuständen die Addition der Rundenkonstanten und die Byterotation hinzu. Um 8 Zustände zu erlangen und damit der Forderung aus Glg. 6.1 nachzukommen, erfolgt die Verknüpfung der Zustände `DoSubBytes` und `DoRotate`. Dies ist möglich, da letzterer Zustand nur durch die Schlüsselexpansion verwendet wird und stets direkt vor der Byte-Substitution durchgeführt werden muss. Desweiteren bilden `DoAddRoundKey` und die Addition von `DoAddRndConst` einen Zustand. Die globale FSM des CPCIPH beinhaltet genau 8 Zustände und obige Forderung ist erfüllt.

Zustandsübergänge des globalen Automaten. Dieser Abschnitt beschäftigt sich mit den Zustandsübergängen des globalen Automaten, deren Einflussfaktoren Abb. 31 visualisiert. Eingänge der Zustandsmaschine sind die

- aktuelle Runde und Spalte,
- Ausgänge aus dem Decoder und

- Release-Signale aus den lokalen Zustandsmaschinen.

Die lokalen Automaten arbeiten die Tasks einer Komponente komplett ab, z.B. `SubBytes32` eine Byte-Substitution mit 4 Byte Eingang und 4 Byte Ausgang. Der Einsatz solcher Zustandsmaschinen ist dann sinnvoll, wenn die Durchführung einer Operation mehrere Taktzyklen benötigt. Die lokalen FSMs senden die oben erwähnten Release-Signale aus, um die globale Zustandsmaschine wieder für eine Zustandsänderung freizugeben.

Die Ausgänge der Zustandsmaschine sind neben den aktuellen Zuständen die Init-Signale. Sie sind für genau einen Takt aktiv und erwecken die lokalen Zustandsmaschinen aus dem Idle-Zustand. Hiermit entsteht eine gegenseitige Abhängigkeit von globaler und lokaler FSM.

Das Release-Signal wird gesendet, wenn sichergestellt ist, dass die darauffolgenden Komponenten nicht zur gleichen Zeit auf die gleichen Ressourcen zugreifen wie die aktive. Belegte Ressourcen können z.B. der Datenbus oder GP-Register sein. Dies ermöglicht eine überlappende Bearbeitung der Operationen, sodass eine pipelineähnliche Struktur entsteht. Die nachfolgende lokale Zustandsmaschine wird also bereits aktiviert, bevor der vorherige lokale Automat wieder im Idle-Zustand ist (vgl. Abb. 24 unten). Diese Parallelisierung erhöht den Datendurchsatz, ohne verstärkt Einfluss auf die Gatterzahl zu nehmen.

Es handelt sich bei der globalen FSM um eine Mealy-Maschine, da die Ausgänge zur AGU und EXEC-Einheit nicht nur vom aktuellen Zustand, sondern auch von den Eingängen aus dem Decoder abhängen (vgl. Abb. 31).

Zeitliche Integration der lokalen Automaten. Der folgende Absatz erörtert nun die Verbindung der lokalen Zustandsmaschinen zu den globalen Automaten. Wie bereits angemerkt, erfolgt die Bearbeitung der Komponenten nicht sequentiell, sondern überlappend.

Das Zeitdiagramm in Abb. 32 veranschaulicht die zeitliche Abfolge bei der Berechnung der erste Spalte einer regulären AES-Runde. Die Ausführung von `AddRoundKey32` überschneidet sich z.B. für zwei Takte mit der Prozessierung von `SubBytes32`. Dies ist möglich, da die beiden Komponenten in diesen Takten nicht auf dieselben Ressourcen zugreifen. Verschiedene lokale Automaten steuern separat den Ablauf der Komponenten. Die roten Pfeile in Abb. 32 symbolisieren beispielhaft für das Register `RA`, von welchen Ressourcen dessen Inhalt

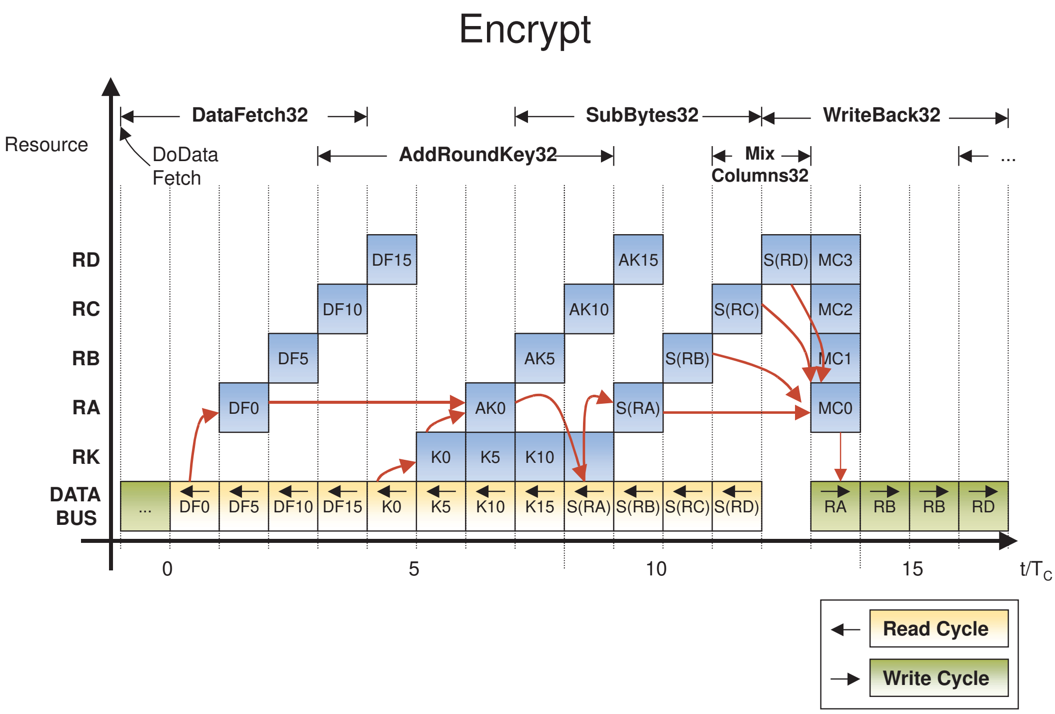


Abbildung 32: Zeitdiagramm der 1. Spalte einer regulären Verschlüsselungsrunde.

jeweils abhängt.

Auffallend ist, dass die Initialisierung der Komponenten schon einen Taktzyklus vor dem ersten Zugriff auf den Datenbus stattfindet. Der erste Zyklus des Zustands ist dabei zur Berechnung der Adresse für den Speicherzugriff notwendig.

Insgesamt benötigt die Rundenoperation für eine Spalte 17 Taktzyklen. Auf den ersten Blick erweckt das Zeitdiagramm den Eindruck, dass die Komponenten zeitlich noch stärker überlappen könnten, da die Register mit 23,5%[61] eine geringe Auslastung aufweisen. Der Flaschenhals liegt jedoch beim Datenbus, der in 16 von 17 Takten Daten befördert (Auslastung 94,1%). Die Modellierung der Ablaufsteuerung des CPCIPH strebt deshalb eine effektive Auslastung des Datenbusses zur Maximierung des Durchsatzes an.

Die auf dem Datenbus beförderten Daten sind:

- vier Eingangsbytes in `DataFetch32`,

[61] Ihre Inhalte werden in 4 von 17 Takten verändert.

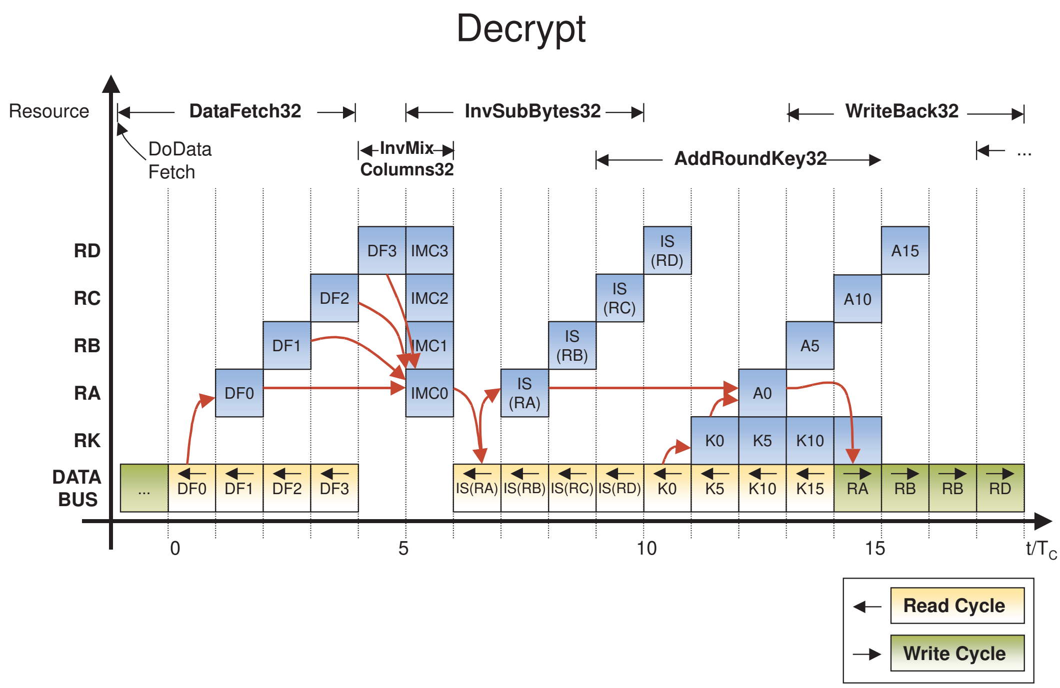


Abbildung 33: Zeitdiagramm der 1. Spalte einer regulären Entschlüsselungsrunde.

- vier Rundenschlüsselbytes in `AddRoundKey32`,
- vier Substitutionsbytes aus einer LUT in `SubBytes32` und
- vier Ausgangsbytes in `WriteBack32`.

Diese Arbeit untersucht insbesondere die Ersetzung der LUT durch festverdrahtete Logik. Dadurch lässt sich eine Entlastung des Datenbusses, eine Erhöhung der Parallelität und eine Verringerung der benötigten Taktzyklen erreichen. Diesen Aspekt untersuchen die Unterpunkte 7.3.2 und 7.3.3 im Detail.

Das Zeitdiagramm für den Entschlüsselungsvorgang in Abb. 33 zeigt die Ressourcenbelegung und den zeitlichen Verlauf der 1. Spalte einer regulären Dechiffrierrunde. Die Datenbusauslastung beträgt hier 88,9% (16/18), die Registerauslastung 22,2% (4/18). Auch bei der Entschlüsselung wird eine pipelineähnliche Struktur durch zeitlich versetzte Ausführung der Komponenten erreicht.

6.2.3 Partitionierung der Komponenten

Sowohl `SubBytes32` als auch `InvSubBytes32` können weiter zerlegt werden. Die Substitution *eines* Bytes benötigt dann mehrere Taktzyklen und wird somit zeitlich partitioniert. Dies ist eng verzahnt mit einer modulinternen Optimierung und wird in den Unterpunkten 7.3.2 und 7.3.3 erläutert. Zudem kann `InvMixColumns32` zerlegt werden, um Logik mit `MixColumns32` zu teilen (vgl. 7.3.1).

6.3 Algorithmische Analyse von Vektormultiplikationen

Die Aufteilung eines Algorithmus in Komponenten findet bei dem CPMAC schon zu einem früheren Zeitpunkt statt: Die MAC-Operation zerlegt Filteralgorithmen oder Matrixmultiplikationen bereits in ihre Bestandteile, nämlich in die Multiplikation zweier Werte und die anschließende Addition zu einem akkumulierten Wert.

Der eigentliche Algorithmus wird in Software in Form eines Assemblerprogramms ausgeführt. Damit verschiebt sich ein Großteil der Ablaufsteuerung des Algorithmus auf die Softwareseite und die Zustandsmaschine des CPMAC fällt deutlich simpler aus, als dies der Fall für den CPCIPH ist.

Die Komponenten entsprechen größtenteils den in den Anforderungen spezifizierten Operationen:

- Addition und Subtraktion,
- Linksshift und Rechtsshift sowie
- Multiplikation und MAC.

Pro Instruktion wird nur eine dieser Komponenten angesprochen. Eine weitere Aufteilung ist auch hier sinnvoll, um ähnliche Operationen mit gemeinsamer Logik auszustatten. Hiermit setzt sich Unterpunkt 7.4 auseinander.

7 Optimierung der Komponenten

Das vorige Kapitel hat sich mit der Partitionierung der gewählten Algorithmen auseinandergesetzt. Die sinnvolle Wahl der Komponenten und die Optimierung der zeitlichen Ablaufsteuerung bildet das Grundgerüst für die Umsetzung der Algorithmen in Hardware. In diesem Kapitel wird die flächenoptimierte Realisierung der im vorangegangen Kapitel bestimmten Komponenten erarbeitet. Nach einer einleitenden Vorstellung der Möglichkeiten zur Gatterminimierung wird der im Rahmen der Arbeit weiterentwickelte Greedy-Algorithmus vorgestellt. Den Hauptteil des Kapitels bildet die Gatterminimierung des CPCIPH und CPMAC.

7.1 Optimierungsmöglichkeiten

Die Optimierungsmöglichkeiten innerhalb der Komponenten hängen sehr stark von ihrer Funktionalität ab. Verschiedenartige Algorithmen verwenden unterschiedliche Operationen und lassen sich somit nicht auf dieselbe Art und Weise optimieren: Während die Komponenten des CPCIPH hauptsächlich Galois-Feld Operationen durchführen, welche Algorithmen zur Minimierung des XOR-Gatteraufwands erfordern, beschäftigt sich die Optimierung des CPMAC vornehmlich mit der Gatterminimierung der Multiplikations- und Additionsoperation und wägt zwischen den dafür möglichen Strukturen ab.

Das Hauptaugenmerk liegt auf folgenden Optimierungsformen:

- Anwendung von Optimierungsalgorithmen zur Verringerung der XOR-Gatterzahl: Hierfür wird in Unterpunkt 7.2 der Enhanced-Greedy-Algorithmus vorgestellt.
- Anwendung von Optimierungsalgorithmen zur Erstellung einer gatterminimierten disjunktiven Normalform (DNF)[62]: Die bekanntesten Algorithmen hierfür sind das Quine/McClusky-Verfahren und die Karnaugh-Diagramme. Derartige Methoden werden häufig von Synthesetools durchgeführt, die eine manuelle Optimierung überflüssig machen.
- Minimierung der Flipflopzahl: Diese kann durch Mehrfachnutzung der Register erreicht werden.

[62] vgl. [Tie91], S.196

- Vergleich des Bedarfs an Gatteräquivalenten (GÄ) eines ROMs gegenüber festverdrahteter Logik, z.B. für eine LUT. Es wird entschieden, für welche Einsatzgebiete festverdrahtete Logik einem ROM vorzuziehen ist.
- Minimierung des RAM- und ROM-Speicherbedarfs.
- Minimierung durch Mehrfachnutzung ähnlicher Logikstrukturen, wie z.B. Nutzung desselben Addierers durch zwei Komponenten.

Nach der Erläuterung des verbesserten Greedy-Algorithmus werden die vorgestellten Optimierungsmöglichkeiten bezüglich ihres Einsatzes für den CPCIPH und den CPMAC erörtert.

7.2 Enhanced-Greedy-Algorithmus

Insbesondere kryptographische Algorithmen wie der AES beinhalten Multiplikationen einer Matrix mit konstanten Elementen im $\mathcal{GF}(2)$ mit einem variablen Vektor. Die Ergebnisbits berechnen sich dann aus der Galois-Feld-Addition jeweils unterschiedlicher Eingänge.

Anhand der in 3.3.2 eingeführten Matrixmultiplikation der affinen Transformation wird erläutert, wie man die Gatterzahl in solch einem Fall minimieren kann. Es ist möglich, dass bei dem im folgenden vorgestellten Algorithmus mehrere Lösungen mit minimaler Gatterzahl berechnet werden. Die Auswahl einer dieser Lösungen fällt auf diejenige, welche den *minimalen kritischen Pfad* besitzt. Betrachtet man ausschließlich die Matrixmultiplikation der affinen Transformation, so wird der Vektor b durch Multiplikation einer konstanten Matrix mit dem variablen Vektor t berechnet:

$$b = \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1\,1\,1\,1\,1\,0\,0\,0 \\ 0\,1\,1\,1\,1\,1\,0\,0 \\ 0\,0\,1\,1\,1\,1\,1\,0 \\ 0\,0\,0\,1\,1\,1\,1\,1 \\ 1\,0\,0\,0\,1\,1\,1\,1 \\ 1\,1\,0\,0\,0\,1\,1\,1 \\ 1\,1\,1\,0\,0\,0\,1\,1 \\ 1\,1\,1\,1\,0\,0\,0\,1 \end{bmatrix} \times \begin{bmatrix} t_7 \\ t_6 \\ t_5 \\ t_4 \\ t_3 \\ t_2 \\ t_1 \\ t_0 \end{bmatrix} \tag{7.1}$$

Ausmultipliziert kann das resultierende Byte b dargestellt werden durch:

$$\begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} t_7 \oplus t_6 \oplus t_5 \oplus t_4 \oplus t_3 \\ t_6 \oplus t_5 \oplus t_4 \oplus t_3 \oplus t_2 \\ t_5 \oplus t_4 \oplus t_3 \oplus t_2 \oplus t_1 \\ t_4 \oplus t_3 \oplus t_2 \oplus t_1 \oplus t_0 \\ t_7 \oplus t_3 \oplus t_2 \oplus t_1 \oplus t_0 \\ t_7 \oplus t_6 \oplus t_2 \oplus t_1 \oplus t_0 \\ t_7 \oplus t_6 \oplus t_5 \oplus t_1 \oplus t_0 \\ t_7 \oplus t_6 \oplus t_5 \oplus t_4 \oplus t_0 \end{bmatrix} \tag{7.2}$$

Die Hardwarerealisierung von Glg. 7.2 benötigt 32 XOR-Gatter. Einige XOR-Verknüpfungen kommen jedoch in mehreren Zeilen vor und bräuchten zusammen nur ein Gatter. So kommt die Operation $t_6 \oplus t_5$ sowohl in der ersten als auch in der zweiten Zeile vor und kann von demselben Gatter berechnet werden.

Es existieren mehrere Möglichkeiten, die Logik zusammenzufassen: Anstatt des Terms $t_6 \oplus t_5$ hätte auch $t_6 \oplus t_4$ selektiert werden können. Jede dieser Entscheidungen hat Auswirkungen auf den weiteren Verlauf der Optimierung und wird sich schließlich bezüglich der insgesamt benötigten XOR-Gatter als vorteilhaft oder nachteilhaft herausstellen. Haben die Gleichungen wenige äquivalente Operationen, können alle Möglichkeiten überprüft werden. Steigt die Anzahl der Überschneidungen jedoch an, so nimmt auch die Zahl der möglichen Lösungswege exponentiell zu.

Aus diesem Grund wird ein Greedy-Algorithmus verwendet und weiterentwickelt, der gezielt Entscheidungen über das Zusammenfassen der Terme trifft. Glg. 7.2 wird nun folgendermaßen dargestellt:

t_7	t_6	t_5	t_4	t_3	t_2	t_1	t_0
1	1	1	1	1	0	0	0
0	1	1	1	1	1	0	0
0	0	1	1	1	1	1	0
0	0	0	1	1	1	1	1
1	0	0	0	1	1	1	1
1	1	0	0	0	1	1	1
1	1	1	0	0	0	1	1
1	1	1	1	0	0	0	1

(7.3)

```
greedy_step(M, noL, noC){
   max_matches = 0;
   noSaves = 0;

   // 1. Suchen aller "max_matches" Spalten.
   for (cl = 0; cl < noC-1; cl++){
      for (cr = cl+1; cr < noC; cr++){
         matches = get_matches(M, noL, cl, cr);
         if (matches >  max_matches){
            save_l[0]       = cl;
            save_r[0]       = cr;
            noSaves         = 1;
            max_matches     = matches;}
         else if (matches == max_matches){
            save_l[noSaves] = cl;
            save_r[noSaves] = cr;
            noSaves++; } } }

   // 2. Bearbeiten aller "max_matches" Spalten, Rekursion.
   if (max_matches > 1){
      for (i = 0; i < noSaves; i++){
         newM = merge_two_cols(M, save_l[i], save_r[i],
                               noC, noL);
         greedy_step(newM, noL, noC+1); } }

   // 3. Auswerten Ergebnismatrix.
   else {
      len_critpath = get_len_critpath(M, noL, noC);
      gate_cnt     = get_gate_cnt(M, noL, noC);
      if ((gate_cnt <  MIN_GATES) ||
          (gate_cnt == MIN_GATES  && len_critpath < MIN_PATH){
         MIN_GATES = gate_cnt;
         MIN_PATH  = len_critpath;
         write("TEMPORARY MINIMUM FOUND %M", M);} }
}
```

Beispiel 4: Der Greedy-Algorithmus mit Analyse des kritischen Pfads.

Der Pseudocode in Beispiel 4 verdeutlicht die *rekursive* Implementierung des Enhanced-Greedy-Algorithmus. Eine *iterative* Lösung des Problems stellt [Pa94] vor. Der in dieser Arbeit entwickelte Algorithmus zieht als zusätzliches Kriteri-

um für ein optimiertes Ergebnis die Länge des kritischen Pfads heran.[63]
Die Funktion `greedy_step` in Beispiel 4 fasst jeweils zwei Spalten zusammen. Als Parameter werden der Funktion die Anzahl der Zeilen `noL` und Spalten `noC` der Matrix sowie die zu optimierende Matrix `M` übergeben. Im 1. Abschnitt werden in allen möglichen Kombinationen immer zwei Spalten verglichen. Geprüft wird, in wievielen Zeilen für beide Spalten eine '1' geführt wird (sog. `matches`). Diese `matches` zeigen an, wie häufig ein Gatter wiederverwertet würde. Die Spaltenkombination mit den meisten Übereinstimmungen (`max_matches`) wird gesichert. Falls es mehrere Spaltenkombinationen mit maximaler Anzahl an Übereinstimmungen gibt, so werden alle gesichert.

Falls die aktuelle Matrix weiterhin optimierbar ist (`max_matches > 1`), so wird mit dem 2. Abschnitt fortgefahren. Die Funktion `merge_two_columns` erweitert die Matrix um eine Spalte. Diese enthält '1'-Werte in den Zeilen der Überschneidungen jeweils einer optimalen Spaltenkombination. In den beiden betrachteten Spalten werden in eben diesen Zeilen die Werte auf '0' zurückgesetzt. Danach wird die neue und um eine Spalte größere Matrix mit einem Rekursionsaufruf der `greedy_step` Funktion übergeben, die diese weiteroptimiert. Gibt es mehrere gleichwertige Spaltenkombinationen, dann werden alle diese innerhalb der Schleife im 2. Abschnitt bearbeitet.

Sind keine Optimierungen mehr durchführbar, erfolgt das Auswerten der Ergebnismatrix im 3. Abschnitt, indem die benötigte Anzahl der XOR-Gatter (`gate_cnt`) berechnet wird. Ein Ergebnis wird schließlich ausgegeben, wenn diese Gatterzahl niedriger ist als bei allen zuvor berechneten Lösungen. Die Gesamtzahl N_{gate} der benötigten XOR-Gatter einer Ergebnismatrix ergibt sich zu

$$N_{gate} = N_{pre} + N_{set} - N_{line}, \tag{7.4}$$

wobei N_{pre} die Anzahl der neuen Spalten, also der vorberechneten Terme, N_{set} die Anzahl der verbliebenen gesetzten Bits ('1') und N_{line} die Anzahl der Zeilen darstellt (vgl. [Pa94]).

Zusätzlich wird der kritische Pfad der Lösung bestimmt (`len_critpath`). Falls eine Lösung der Optimierung die gleiche Anzahl an Gattern wie das bisher beste

[63] Als kritischer Pfad einer rein kombinatorischen Schaltung wird hier vereinfachend der Pfad gewertet, der zwischen einem beliebigen Eingang und Ausgang der Schaltung die maximale Anzahl an zu durchlaufenden Gattern aufweist. Seine Länge wird durch die Anzahl durchlaufener Gatter wiedergegeben.

Resultat benötigt, aber einen kürzeren kritischen Pfad besitzt, so wird diese als neues Optimum übernommen. Den Algorithmus zur Bestimmung des kritischen Pfads erläutert Anhang B.

Optimiert man den Inhalt von Tabelle 7.3, so werden in vier nachfolgend durchgeführten Schritten, also vier rekursiven Aufrufen von `greedy_step`, die neuen Spalten r_0, r_1, r_2 und r_3 berechnet. Das Zwischenergebnis nach diesen vier Schritten zeigt untenstehende Tabelle:

t_7	t_6	t_5	t_4	t_3	t_2	t_1	t_0	$r_0 = t_7 \oplus t_6$	$r_1 = t_5 \oplus t_4$	$r_2 = t_3 \oplus t_2$	$r_3 = t_1 \oplus t_0$
0	0	0	0	1	0	0	0	1	1	0	0
0	1	0	0	0	0	0	0	0	1	1	0
0	0	0	0	0	0	1	0	0	1	1	0
0	0	0	1	0	0	0	0	0	0	1	1
1	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	1	0	0	1	0	0	1
0	0	1	0	0	0	0	0	1	0	0	1
0	0	0	0	0	0	0	1	1	1	0	0

Nach weiteren vier Schritten resultiert die erste Optimierungsmatrix:

t_7	t_6	t_5	t_4	t_3	t_2	t_1	t_0	r_0	r_1	r_2	r_3	$r_4 = r_0 \oplus r_1$	$r_5 = r_0 \oplus r_3$	$r_6 = r_1 \oplus r_2$	$r_7 = r_2 \oplus r_3$
0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0

Setzt man das Ergebnis in Gleichung 7.4 ein, so erhält man als Gesamtzahl

$$N_{gate} = 8 + 16 - 8 = 16$$

XOR-Gatter. Der kritische Pfad dieser Lösung enthält 3 Gatter. Der Algorithmus berechnet noch 1823 weitere Matrizen, doch in diesem Fall liefert schon das erste Resultat die kleinste Gatterzahl und Pfadlänge. Nach Bearbeitung mit dem Greedy-Algorithmus werden nur noch 16 statt 32 XOR-Gatter verwendet.[64]

[64]Es sei darauf hingewiesen, dass der Algorithmus lediglich lokale Minima errechnet und somit nicht den Anspruch auf ein globales Optimum erhebt.

7.3 Optimierung der CPCIPH Komponenten

Gegenstand der Ausführungen ist die Optimierung der in Sektion 6.2 bestimmten Komponenten des CPCIPH hinsichtlich der Reduzierung der Gatterzahl.

7.3.1 MixColumns und InvMixColumns

Betrachtet man die konstanten Matrizen der `MixColumns`- und `InvMixColumns`-Berechnungen in den Gleichungen 3.20 und 3.21, so scheinen diese auf den ersten Blick keine Gemeinsamkeiten aufzuweisen. Barreto (vgl. [Dae02]) zeigt jedoch auf, dass untenstehender Zusammenhang gilt:

$$\begin{bmatrix} \texttt{0E} & \texttt{0B} & \texttt{0D} & \texttt{09} \\ \texttt{09} & \texttt{0E} & \texttt{0B} & \texttt{0D} \\ \texttt{0D} & \texttt{09} & \texttt{0E} & \texttt{0B} \\ \texttt{0B} & \texttt{0D} & \texttt{09} & \texttt{0E} \end{bmatrix} = \begin{bmatrix} \texttt{02} & \texttt{03} & \texttt{01} & \texttt{01} \\ \texttt{01} & \texttt{02} & \texttt{03} & \texttt{01} \\ \texttt{01} & \texttt{01} & \texttt{02} & \texttt{03} \\ \texttt{03} & \texttt{01} & \texttt{01} & \texttt{02} \end{bmatrix} \times \begin{bmatrix} \texttt{05} & \texttt{00} & \texttt{04} & \texttt{00} \\ \texttt{00} & \texttt{05} & \texttt{00} & \texttt{04} \\ \texttt{04} & \texttt{00} & \texttt{05} & \texttt{00} \\ \texttt{00} & \texttt{04} & \texttt{00} & \texttt{05} \end{bmatrix} \tag{7.5}$$

Der linke Faktor entspricht demnach der Matrix aus dem `MixColumns`-Schritt. Der `InvMixColumns`-Schritt besteht nun aus der Multiplikation des rechten Faktors (von hier an als *Erweiterungsmatrix* bezeichnet) mit der Eingangsspalte und der anschließenden Durchführung des `MixColumns`-Schritts. Bezüglich der Hardwareimplementierung der 32 Bit breiten Komponenten bedeutet dies, dass `InvMixColumns32` in zwei Logik-Einheiten unterteilt wird, von denen die eine äquivalent zu `MixColumns32` ist.

Der besondere Vorteil der Erweiterungsmatrix besteht darin, dass statt der komplexen Elemente der `InvMixColumns`-Matrix lediglich die Elemente `$04` und `$05` enthalten sind. Die Multiplikation eines Bytes mit `$04` berechnet sich unmittelbar aus

$$\begin{aligned} & a \cdot \texttt{04} \\ = \; & ((a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0) \cdot x^2) \;\; (mod R(x)) \\ = \; & a_5x^7 + a_4x^6 + (a_3 \oplus a_7)x^5 + (a_7 \oplus a_6 \oplus a_2)x^4 \\ & +(a_6 \oplus a_1)x^3 + (a_7 \oplus a_0)x^2 + (a_7 \oplus a_6)x + a_6 \end{aligned} \tag{7.6}$$

Die Implementierung ist in Abb. 7.6 dargestellt und benötigt fünf XOR-Gatter. Im kritischen Pfad der Operation liegen zwei XOR-Gatter. Die Multplikation mit

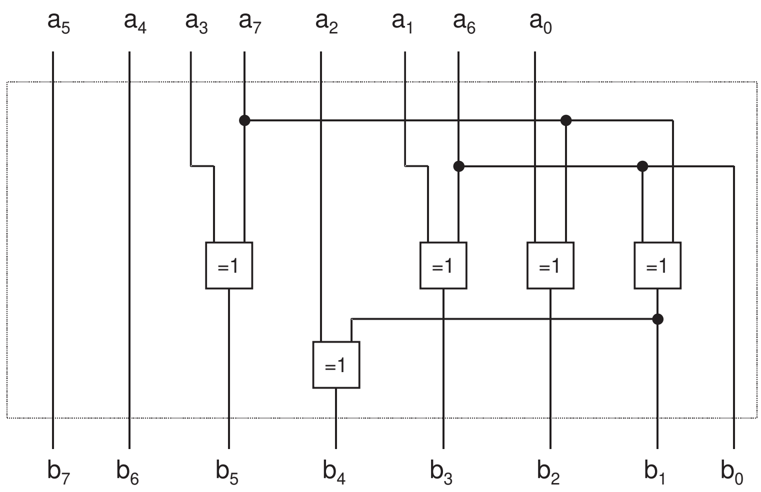


Abbildung 34: Multiplikation mit `$04`.

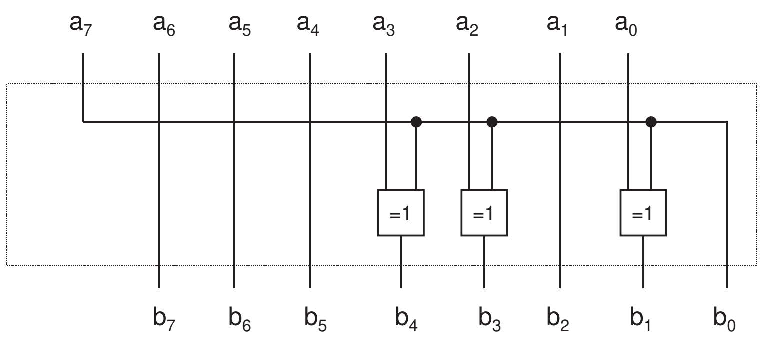


Abbildung 35: Multiplikation mit `$02`.

`$05` kann im $\mathcal{GF}(2^8)$ als Multiplikation mit `$04` und anschließender Addition des Eingangsvektors umgesetzt werden, also $\texttt{05} \cdot a = \texttt{04} \cdot a + a$. Eine ähnliche Berechnung kann für die Multiplikation mit `$02` vollzogen werden, die nach der algorithmischen Umstellung sowohl für den `MixColumns`- als auch für den `InvMixColumns`-Schritt benötigt wird. Abb. 35 veranschaulicht das Ergebnis der Berechnung von

$$
\begin{aligned}
& a \cdot \texttt{02} \\
= \;& a_6 x^7 + a_5 x^6 + a_4 x^5 + (a_7 \oplus a_3) x^4 \\
& +(a_7 \oplus a_2) x^3 + a_1 x^2 + (a_7 \oplus a_0) x + a_7
\end{aligned}
$$

Für diese Operation werden insgesamt 3 XOR-Gatter benötigt. Für die Multiplikation mit `$03` gilt die Gleichung $03 \cdot a = 02 \cdot a + a$.

Würde die gesamte `InvMixColumns32`-Komponente in einem einzigen Taktzyklus das Ergebnis berechnen, so ergäbe das einen langen kritischen Pfad. Um diesen zu vermeiden, soll `InvMixColumns32` in mehreren Taktschritten arbeiten, wobei im letzten Zyklus `MixColumns32` zum Einsatz kommt. Abb. 36 gibt das veränderte Zeitdiagramm der Komponente wieder.

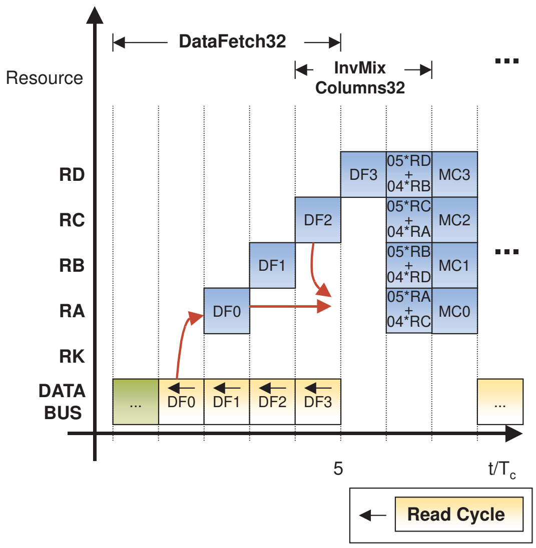


Abbildung 36: Ausschnitt aus verändertem Zeitdiagramm für Dechiffrierrunde.

Darauf aufbauend wird versucht, die `MixColumns32`- und `InvMixColumns32`-Komponenten zu verschmelzen, sodass sie Teile ihrer Logik gemeinsam nutzen. Für die `InvMixColumns32`-Komponente wird die Multiplikation der Eingangsspalte p mit den Bytes p_0, p_1, p_2 und p_3 mit der Erweiterungsmatrix wie folgt zerlegt und dem temporären Byte t mit den Bytes t_0, t_1, t_2 und t_3 zugewiesen:

$$
\begin{aligned}
t_0 &= 04 \cdot (p_0 + p_2) + p_0 \\
t_1 &= 04 \cdot (p_1 + p_3) + p_1 \\
t_2 &= 04 \cdot (p_0 + p_2) + p_2 \\
t_3 &= 04 \cdot (p_1 + p_3) + p_3 \qquad (7.7)
\end{aligned}
$$

Die erste und dritte bzw. zweite und vierte Zeile enthalten dieselben Klammerausdrücke, sodass sie jeweils nur einmal umgesetzt werden müssen.

Durch die Multiplikation des Ergebnisses in t mit der `MixColumns` Matrix ergibt sich die Spalte q, die die Bytes q_0, q_1, q_2 und q_3 enthält. Für die Hardwarerealisierung wird folgendermaßen umgestellt:

$$
\begin{aligned}
q_0 &= 02 \cdot (t_0 + t_1) + (t_1 + t_3) + t_2 \\
q_1 &= 02 \cdot (t_1 + t_2) + (t_0 + t_2) + t_3
\end{aligned}
$$

$$q_2 = \mathtt{02} \cdot (t_2 + t_3) + (t_1 + t_3) + t_0$$
$$q_3 = \mathtt{02} \cdot (t_3 + t_0) + (t_0 + t_2) + t_1 \tag{7.8}$$

Die Multiplikation mit `$02` kann ausgeklammert werden. Zusätzlich erkennt man, dass der zweite Klammerausdruck der 1. und 3. sowie der 2. und 4. Zeile identisch sind und nur einmal umgesetzt werden müssen.

Berücksichtigt man, dass die Multiplikation mit der Erweiterungs- und der `MixColumns`-Matrix in zwei verschiedenen Taktzyklen durchgeführt und die Bytes q_i und t_i in denselben GP-Registern `RA`, `RB`, `RC` und `RD` abgelegt werden (natürlich um einen Takt zeitversetzt), genügt die einmalige Bereitstellung der Logik für die Operationen $(t_1 + t_3)$ und $(p_1 + p_3)$ sowie für $(t_0 + t_2)$ und $(p_0 + p_2)$. Die Zuordnung der Variablen zu den Registern geschieht folgendermaßen:

- $p_0, t_0, q_0 \rightarrow$ `RA`,
- $p_1, t_1, q_1 \rightarrow$ `RB`,
- $p_2, t_2, q_2 \rightarrow$ `RC` und
- $p_3, t_3, q_3 \rightarrow$ `RD`.

Abb. 37 stellt die Umsetzung der verschmolzenen Komponenten dar. Die Signale sind byteweise aufgetragen, sodass die Gatter jeweils 8 Ausgangsbits erzeugen. Die grau gekennzeichneten Gatter verwendet ausschließlich die `InvMixColumns32`-Komponente, alle anderen Gatter werden von `MixColumns32` oder von beiden benötigt. Insgesamt setzen die verschmolzenen Komponenten

$$\begin{aligned} N_{gates} &= 18 \cdot N_{\oplus} + 4 \cdot N_{\mathtt{02}} + 2 \cdot N_{\mathtt{04}} \\ &= 18 \cdot 8 + 4 \cdot 3 + 2 \cdot 5 \\ &= 166 \end{aligned} \tag{7.9}$$

XOR-Gatter ein. Dabei meint $N_{\oplus}$ die Anzahl der 8 Bit breiten Exklusiv-Oder-Verknüpfungen und $N_{\mathtt{02}}$ sowie $N_{\mathtt{04}}$ die Zahl der XOR-Gatter der jeweiligen Konstantenmultiplikation. Die flächenoptimierte Implementierung von Zhang und Parhi [Zha02] benötigt im Vergleich hierzu 264 XOR-Gatter.

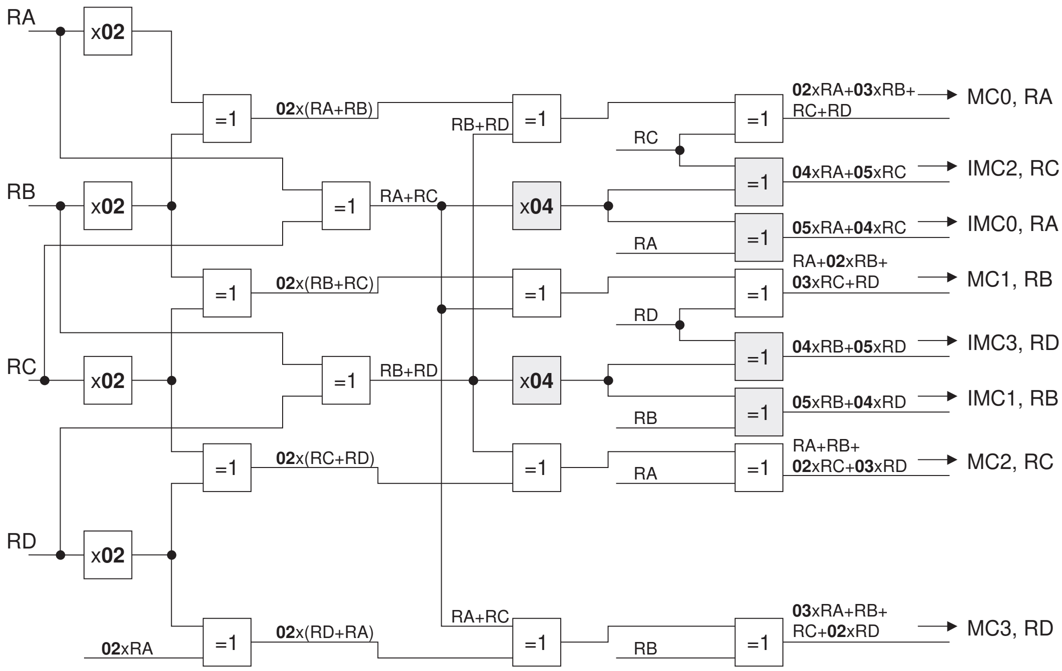


Abbildung 37: EXEC-Einheit der `MixColumns32`- und `InvMixColumns32`-Komponente.

7.3.2 Affine Transformation vs. ROM

Der `SubBytes`- und `InvSubBytes`-Schritt wird gemeinhin durch jeweils eine 256 Byte große LUT dargestellt. In Hardware erfolgt diese durch den Einsatz eines 512 Byte großen ROMs. 256 Byte beinhalten folglich die Werte für den `SubBytes`-, die anderen 256 Byte die Substitutionskonstanten für den `InvSubBytes`-Schritt. Aus Kostengründen wird die LUT im Programmspeicher abgelegt, sodass in vier Taktzyklen Daten aus dem ROM geladen werden. Ein separates ROM würde zwar den Datenbus entlasten, bräuchte aber eine eigene Adresslogik. Dadurch wäre es größer und kostenaufwendiger.

Bisher wurde davon ausgegangen, dass für beide Schritte eine eigene LUT belegt werden soll. Nach Unterpunkt 3.3.2 setzt sich eine Byte-Operation aus

- der multiplikativen Inversion eines Polynoms im $\mathcal{GF}(2^8)$ und
- einer affinen Transformation

zusammen. Bei der Inversion wird ein Eingangsbyte, welches das Polynom im $\mathcal{GF}(2^8)$ darstellt, auf ein Ausgangsbyte, welches das inverse Polynom im $\mathcal{GF}(2^8)$

wiederspiegelt, abgebildet. Diese Operation ist für beide Substitutionsschritte identisch. Die beiden Schritte unterscheiden sich also lediglich durch die Berechnungen der affinen und invers affinen Transformation.

Die Idee ist es nun, die Transformationen auf festverdrahtete Logik abzubilden und nur die Inversionsoperation durch eine LUT in einem 256 Byte großen ROM abzulegen. Diese LUT enthält dann für sämtliche $(2^8) = 256$ möglichen Eingangspolynome das Lösungspolynom der Inversionsoperation. Statt eines 512 Byte großen ROMs würde damit nur noch ein ROM mit 256 Byte benötigt.

Gesucht wird sowohl für die affine als auch für die invers affine Transformation eine gatterminimierte Hardwareumsetzung. Betrachtet man die affine Transformation aus Unterpunkt 3.3.2,

$$b = \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1\,1\,1\,1\,1\,0\,0\,0 \\ 0\,1\,1\,1\,1\,1\,0\,0 \\ 0\,0\,1\,1\,1\,1\,1\,0 \\ 0\,0\,0\,1\,1\,1\,1\,1 \\ 1\,0\,0\,0\,1\,1\,1\,1 \\ 1\,1\,0\,0\,0\,1\,1\,1 \\ 1\,1\,1\,0\,0\,0\,1\,1 \\ 1\,1\,1\,1\,0\,0\,0\,1 \end{bmatrix} \times \begin{bmatrix} t_7 \\ t_6 \\ t_5 \\ t_4 \\ t_3 \\ t_2 \\ t_1 \\ t_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \tag{7.10}$$

so wird ein Lösungsbit b_i durch Multiplikation einer Matrixzeile mit dem Eingangsbyte t und anschließender Addition eines konstanten Bits berechnet. Das Lösungsbit b_6 berechnet sich z.B. aus

$$\begin{aligned} b_6 &= 0 \cdot t_7 \oplus 1 \cdot t_6 \oplus 1 \cdot t_5 \oplus 1 \cdot t_4 \oplus 1 \cdot t_3 \oplus 1 \cdot t_2 \oplus 0 \cdot t_1 \oplus 0 \cdot t_0 \oplus 1 \\ &= t_6 \oplus t_5 \oplus t_4 \oplus t_3 \oplus t_2 \oplus 1 \\ &= \neg(t_6 \oplus t_5 \oplus t_4 \oplus t_3 \oplus t_2) \end{aligned} \tag{7.11}$$

Interessant ist die Behandlung der Konstantenaddition im $\mathcal{GF}(2^8)$. Die bitweise Addition einer '1' entspricht der Inversion des Ergebniswertes, die einer '0' der Identität des Resultats.

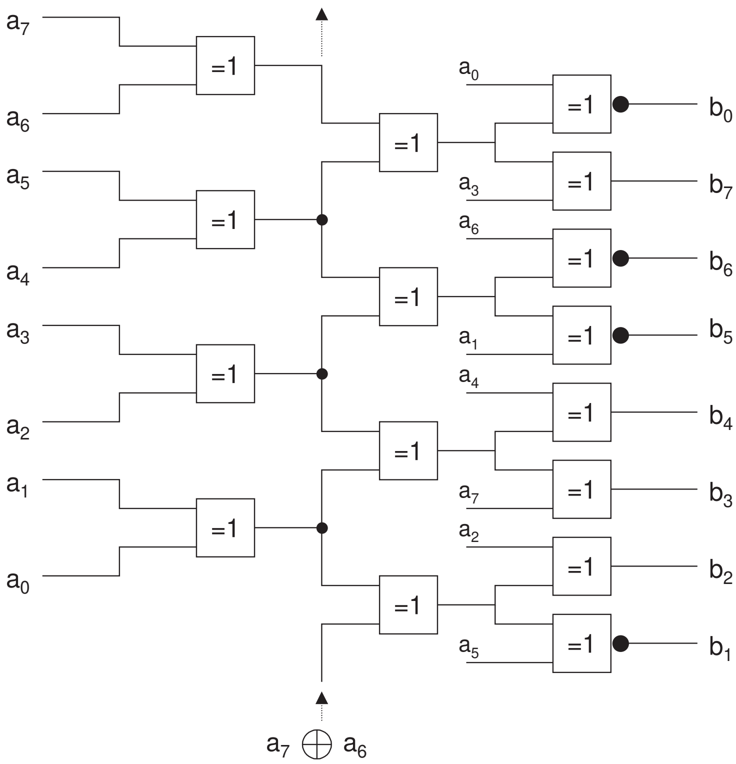


Abbildung 38: Schaltung der affinen Transformation.

Das Ausgangsbyte der affinen Transformation berechnet sich somit zu:

$$\begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} t_7 \oplus t_6 \oplus t_5 \oplus t_4 \oplus t_3 \\ \neg(t_6 \oplus t_5 \oplus t_4 \oplus t_3 \oplus t_2) \\ \neg(t_5 \oplus t_4 \oplus t_3 \oplus t_2 \oplus t_1) \\ t_4 \oplus t_3 \oplus t_2 \oplus t_1 \oplus t_0 \\ t_7 \oplus t_3 \oplus t_2 \oplus t_1 \oplus t_0 \\ t_7 \oplus t_6 \oplus t_2 \oplus t_1 \oplus t_0 \\ \neg(t_7 \oplus t_6 \oplus t_5 \oplus t_1 \oplus t_0) \\ \neg(t_7 \oplus t_6 \oplus t_5 \oplus t_4 \oplus t_0) \end{bmatrix} \tag{7.12}$$

Wie in Unterpunkt 7.2 erläutert, verwendet die Matrix nach Bearbeitung mit dem Enhanced-Greedy-Algorithmus nur noch 16 statt 32 XOR-Verknüpfungen Dies kommt einer Ersparnis von 50% der Gatter gleich. Die optimierte Schaltung wird in Abb. 38 veranschaulicht.

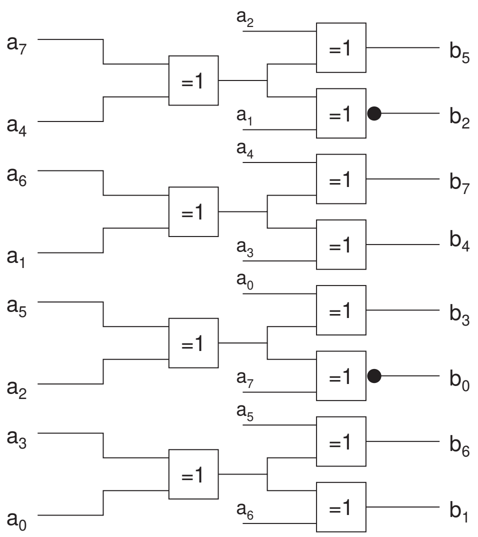


Abbildung 39: Schaltungsdarstellung der invers affinen Transformation.

Auch die Matrixmultiplikation der invers affinen Transformation kann mit Hilfe des Enhanced-Greedy-Algorithmus optimiert werden. Die resultierende Schaltung braucht insgesamt 12 anstatt 16 Gatter. Dies hat eine Ersparnis von 25% gegenüber einer nicht optimierten Schaltung zur Folge. Abb. 39 stellt die verbesserte Schaltung dar.

Insgesamt werden also 28 XOR-Gatter für die affine und die invers affine Transformation gemeinsam benötigt.

Es ist denkbar, beide Logikeinheiten zusammenzufassen und gemeinsam mit dem Enhanced-Greedy-Algorithmus zu optimieren. Die Untersuchung einer kombinierten Logikeinheit durch den Enhanced-Greedy-Algorithmus ergibt jedoch keine weitere Gatterersparnis. Das Minimum von 28 XOR-Gattern bleibt demnach auch für eine gemeinsame Einheit bestehen.

Zusätzlich kommt der Aufwand für Multiplexer und Routing, also die Verdrahtung der Zellen, hinzu. Ein ROM mit einer Größe von 256 Byte ohne eigene Zeilenlogik belegt im Vergleich dazu die Fläche von ca. 120 XOR-Gattern.[65] Desweiteren müssen für die Adressberechnung für die ROM-Zugriffe zusätzliche Gatter einkalkuliert werden.

Die Lösung mit festverdrahteter Logik wird im folgenden Unterpunkt weiterverfolgt, da sie zusätzlich zu ihrem geringen Flächenbedarf die Möglichkeit eröffnet, durch Nutzung der multiplikativen Inversion im $\mathcal{GF}((2^4)^2)$ gänzlich auf ein internes ROM zu verzichten.

[65]Die Berechnung erfolgte anhand der im IMS verwendeten Technologie mit 0,5 µm Strukturbreite. Die Fläche des LUTs im ROM wurde durch die Größe eines XOR-Gatters dividiert, um einen Vergleichswert in Form von XOR-Äquivalenten zu erhalten.

7.3.3 Multiplikative Inversion im $\mathcal{GF}(2^8)$

Im vorangegangenen Unterpunkt wurde die vom CPCIPH benötigte ROM-Größe von 512 auf 256 Byte reduziert. Die nachfolgenden Untersuchungen ersetzen die 256 Byte große LUT durch eine festverdrahtete Lösung der multiplikativen Inversion im $\mathcal{GF}(2^8)$. [66] Insbesondere wird die von der Logik benötigte Fläche mit der eines ROMs verglichen.

Algorithmische Umformung. Wie in Unterpunkt 3.3.2 gezeigt, kann man den `SubBytes`-Schritt durch die Funktion

$$\mathcal{S}(a) = f(g(a))$$

darstellen. Hierbei ist $g(a)$ die Inversion des Vektors a im $\mathcal{GF}(2^8)$ und f die affine Transformation. Möchte man die Inversion im isomorphen Körper $\mathcal{GF}((2^4)^2)$ durchführen, kann die Gleichung umgeschrieben werden in

$$\begin{aligned} \mathcal{S}(a) &= f(g(a)) \\ &= f(\mathbf{M}^{-1} \cdot h(\mathbf{M} \cdot a)) \end{aligned} \tag{7.13}$$

Dabei gibt h die Inversionsoperation im $\mathcal{GF}((2^4)^2)$, $\mathbf{M}$ die Transformationsmatrix und $\mathbf{M}^{-1}$ ihre Inverse wieder. Der Vektor a wird mit $\mathbf{M}$ in den Körper $\mathcal{GF}((2^4)^2)$ transformiert, danach wird die Inversion durchgeführt und das Resultat wird mit $\mathbf{M}^{-1}$ zurücktransformiert in den Körper $\mathcal{GF}(2^8)$. Abschließend folgt die affine Transformation. Sie lässt sich darstellen als Multiplikation des Eingangsvektors mit der konstanten Matrix $\mathbf{F}_0$ und anschließender Addition des konstanten Vektors f_0 oder

$$f(a) = \mathbf{F}_0 \cdot g(a) + f_0$$

Eingesetzt in Gleichung 7.13 ergibt dies

$$\begin{aligned} \mathcal{S}(a) &= \mathbf{F}_0 \cdot (\mathbf{M}^{-1} \cdot h(\ \mathbf{M} \cdot a\ )) + f_0 \\ &= (\mathbf{F}_0 \cdot \mathbf{M}^{-1}) \cdot h(\ \mathbf{M} \cdot a\ )) + f_0 \\ &= \mathbf{K} \cdot h(\ \mathbf{M} \cdot a\ ) + f_0 \end{aligned} \tag{7.14}$$

Die konstante Matrix $\mathbf{K}$ ersetzt die Matrixmultiplikation $(\mathbf{F}_0 \cdot \mathbf{M}^{-1})$.

[66] Hinsichtlich Verlustleistung optimierte Umsetzungen der Operationen stellt [MoS03] vor.

Auch für die Berechnung des `InvSubBytes`-Schritts ist das Zusammenfassen der Matrizen möglich. Die Multiplikation von $\mathbf{F}_{inv}$ mit dem Eingangsvektor p und die anschließende Addition des konstanten Vektors f_{inv0} ermitteln die invers affine Transformation f^{-1}. Nach der Umformung in

$$\begin{aligned} \mathcal{S}_{inv}(p) &= \mathbf{M}^{-1} \cdot h(\ \mathbf{M} \cdot f^{-1}(p))\ ) \\ &= \mathbf{M}^{-1} \cdot h(\ \mathbf{M} \cdot (\mathbf{F}_{inv} \cdot p + f_{inv0})\ ) \\ &= \mathbf{M}^{-1} \cdot h(\ (\mathbf{M} \cdot \mathbf{F}_{inv}) \cdot p + (\mathbf{M} \cdot f_{inv0})\ ) \\ &= \mathbf{M}^{-1} \cdot h(\ \mathbf{K}_{inv} \cdot p + k_{inv0}\ ) \end{aligned} \tag{7.15}$$

verbleibt eine Multiplikation mit der konstanten Matrix $\mathbf{K}_{inv}$, eine Addition mit dem konstanten Vektor k_{inv0}, die multiplikative Inversion im $\mathcal{GF}((2^4)^2)$ und die abschließende Multiplikation mit der Matrix $\mathbf{M}^{-1}$.

Multiplikative Inverse im $\mathcal{GF}(2^4)$. Die multiplikative Inverse im $\mathcal{GF}(2^4)$, die einen Teil der Inversion im $\mathcal{GF}((2^4)^2)$ ausmacht,[67] wird mittels des erweiterten Euklidschen Algorithmus berechnet (s. hierzu auch [Ert01]). Das Ergebnis wird in untenstehender LUT zusammengefasst, die zu den 16 möglichen Eingangsvektoren die dazugehörigen Ausgangsvektoren bereitstellt.

a	$b = a^{-1}$	a	$b = a^{-1}$
0	0	8	F
1	1	9	2
2	9	A	C
3	E	B	5
4	D	C	A
5	B	D	4
6	7	E	3
7	6	F	8

Die Werte sind hexadezimal angegeben und repräsentieren ein Polynom im $\mathcal{GF}(2^4)$. So steht z.B. der Wert `$D` für das Polynom $y^3 + y^2 + 1$. Multipliziert mit

[67] vgl. Unterpunkt 3.2.4

seinem inversen Wert $4 respektive dem Polynom y^2 und modulo $S(y) = y^4+y+1$ folgt:

$$\begin{aligned}
\texttt{\$D} \cdot \texttt{\$4} &= (y^3+y^2+1)\cdot y^2 \\
&= y^5+y^4+y^2 \\
&= (y+1)(y^4+y+1)+1 \\
&= 1\ mod\ S(y) \qquad (7.16)
\end{aligned}$$

Eine optimierte Schaltung zu dieser LUT wird mit Hilfe von Karnaugh-Veigh-Diagrammen oder dem Quine/McCluskey-Verfahren ermittelt: [68]

$$\begin{aligned}
b_3 &= (a_1 \oplus a_2)\,\overline{a_3} \;+\; \overline{a_0}\,\overline{a_1}\,a_3 \;+\; a_1\,a_3\,(\overline{a_0} \oplus \overline{a_2}) \\
b_2 &= a_0\,a_1\,\overline{a_3} \;+\; \overline{a_0}\,a_2\,\overline{a_3} \;+\; a_1\,\overline{a_2}\,a_3 \;+\; \overline{a_1}\,a_3\,\overline{(a_0 \oplus a_2)} \\
b_1 &= \overline{a_0}\,a_1\,a_2 \;+\; a_0\,a_1\,\overline{a_3} \;+\; a_0\,a_2\,\overline{a_3} \;+\; \overline{a_0}\,\overline{a_1}\,a_3 \;+\; \overline{a_1}\,\overline{a_2}\,a_3 \\
b_0 &= (a_0 \oplus a_1)\,\overline{a_3} \;+\; a_1\,a_3(a_0 \oplus a_2) \;+\; \overline{a_0}\,\overline{a_1}\,\overline{a_2}\,a_3 \;+\; \overline{a_0}\,a_2\,\overline{a_3} \qquad (7.17)
\end{aligned}$$

Zeitliche Partitionierung der Byte-Substitutions-Komponenten. Der nächste Abschnitt untersucht den kritischen Pfad bei der Realisierung der multiplikativen Inversion im $\mathcal{GF}(2^8)$. Dabei stellt sich die Frage, inwiefern die Operation partitioniert und damit in mehreren Taktzyklen durchgeführt werden kann. Mit der Partitionierung geht eine Verkürzung des kritischen Pfades einher, der nunmehr auf mehrere sequentiell durchgeführte Taktzyklen aufgeteilt ist.

Die multiplikative Inverse eines Polynoms $C(z) \in \mathcal{GF}((2^4)^2)$ wird nach Unterpunkt 3.2.4 durch das Polynom

$$D(z) \;=\; d_1 \cdot z + d_0; \quad D(z) \in \mathcal{GF}((2^4)^2); \quad d_i \in \mathcal{GF}(2^4), \quad i = 0,1$$

repräsentiert. Die beiden Koeffizienten d_0 und d_1 bestimmen sich nach Gleichung 3.9 zu

$$\begin{aligned}
d_1 &= \quad c_1 \quad \cdot (\omega^{14}\cdot {c_1}^2 + c_0\cdot(c_0+c_1))^{-1} \\
d_0 &= (c_0+c_1)\cdot(\omega^{14}\cdot {c_1}^2 + c_0\cdot(c_0+c_1))^{-1}
\end{aligned}$$

Die Berechnung der multiplikativen Inversen im $\mathcal{GF}(2^4)$ ist für d_1 und d_0 gleich. Folglich ist auch deren kritischer Pfad identisch. Lediglich die Faktoren c_1 bzw.

[68] In Gleichung 7.17 steht das '+' Zeichen ausnahmsweise für eine OR-Verknüpfung.

Op.-Nr.	**Operation**	XOR	AND	OR	in krit. Pfad
1	$\{c_1, c_0\} = \mathbf{M} \cdot a$ oder $\{c_1, c_0\} = \mathbf{K}_{inv} \cdot p + k_{inv0}$	≈ 3	0	0	+
2	$c_{01} = c_0 + c_1$	1	0	0	+
3	$c_{m0} = c_0 \cdot c_{01}$	2	1	0	+
4	$c_{m1} = \omega^{14} \cdot {c_1}^2$	(1)	(0)	(0)	-
5	$c_{m01} = c_{m0} + c_{m1}$	1	0	0	+
6	$c_{inv} = {c_{m01}}^{-1}$	0	2	3	+
7	$d_1 = c_1 \cdot c_{inv}$ oder $d_0 = c_{01} \cdot c_{inv}$	2	1	0	+
8	$b = \mathbf{K} \cdot \{d_1, d_0\} + f_0$ oder $q = \mathbf{M}^{-1} \cdot \{d_1, d_0\}$	≈ 3	0	0	+
	Gesamt	≈ 12	4	3	

Tabelle 3: Der kritische Pfad der `SubBytes32`- und `InvSubBytes32`-Komponenten.

$(c_0 + c_1)$ sind unterschiedlich und können dadurch abweichende kritische Pfade verursachen. Tabelle 3 listet die kritischen Pfade der einzelnen Terme zur Berechnung der $\mathcal{S}$-Box und der $\mathcal{S}_{inv}$-Box in der Reihenfolge ihrer Berechnung auf. Zwei Elemente in geschweiften Klammern wie $\{c_1, c_0\}$ stellen eine Verkettung der beiden Elemente dar. Dabei repräsentiert das linke Element stets das höherwertige Nibbel. Die rechte Spalte der Tabelle legt offen, ob eine Operation im kritischen Pfad liegt oder parallel zu anderen Operationen bearbeitet werden kann.

Operationen 1 und 8 verfügen über zwei unterschiedliche Zeilen, welche jeweils die entsprechenden Operationen für `SubBytes32` und `InvSubBytes32` darstellen. An dieser Stelle wird der durch sie entstehende Pfad einheitlich mit 3 XOR-Gattern angenommen.[69] Operation 4 kann parallel zu den vorherigen Operationen durchgeführt werden und muss deswegen für die Berechnung des kritischen Pfads nicht berücksichtigt werden.

Der kritische Pfad einer $\mathcal{S}$- bzw. $\mathcal{S}_{inv}$-Box-Implementierung besteht aus 12 XOR-, 4 AND- und 3 OR-Gattern. Er bildet gleichzeitig den längsten kombinatorischen

[69] Eine Untersuchung des tatsächlich entstehenden Pfades für eine gatteroptimierte Implementierung der Matrixmultiplikationen findet zu einem späteren Zeitpunkt statt.

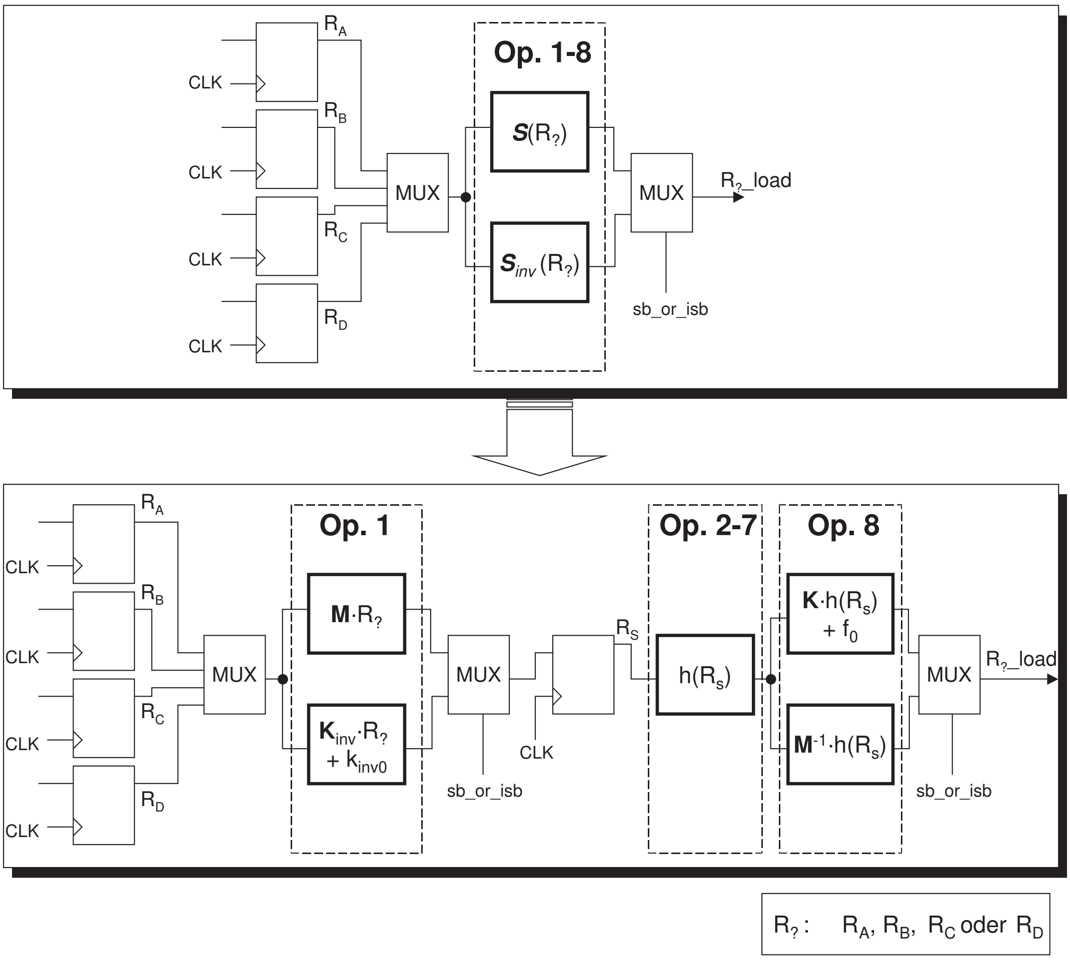


Abbildung 40: Aufteilung der Byte-Substitutions-Schaltung auf zwei Taktzyklen.

Logikpfad der Coprozessorschaltung und setzt ein unteres Limit für die Periodendauer eines Taktzyklus. Um den kritischen Pfad zu verkürzen, wird die Berechnung der Byte-Substitutions-Operation auf zwei Taktzyklen aufgeteilt.

Lediglich nach Operation 1 oder vor Operation 8 ist eine Auftrennung sinnvoll, da ansonsten deutlich mehr Zwischenergebnisse gespeichert werden müssten. Für diese Arbeit werden im ersten Taktzyklus Operation 1 und im zweiten Zyklus die Operationen 2 bis 8 berechnet. Die Aufteilung der Byte-Substitutions-Operation auf zwei Takte veranschaulicht Abb. 40.

Aus der Abbildung geht hervor, dass das neue 8 Bit breite Register `RS` eingeführt wurde. In diesem werden die temporären Ergebnisse nach Operation 1 abgelegt. Die Einführung von `RS` hat den Vorteil, dass die Eingangsmultiplexer vor

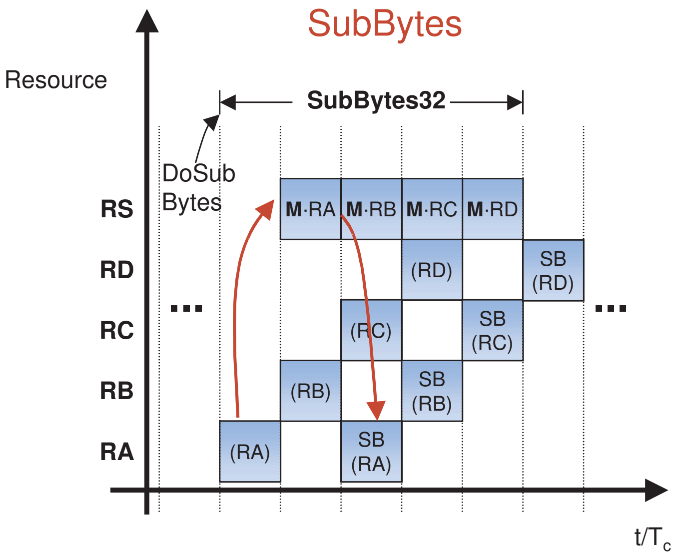


Abbildung 41: Gepipelinete `SubBytes32`-Komponente.

der Logik zur Berechnung von Operation 2 bis 7 wegfallen. Damit können deren zusätzliche Gatterlaufzeiten eingespart werden. Der Zeitpfad im 1. Takt der unterteilten Schaltung ist im Gegensatz zu dem im 2. Takt unproblematisch, da er lediglich aus den beiden Multiplexern und den 3 XOR-Gattern besteht.

Die neue Schaltung kann sehr gut gepipelinet werden, sodass Operation 1 parallel zu Operation 2 bis 8 durchführbar ist. Beide schreiben ihr Ergebnis stets in unterschiedliche Register (s. Abb. 41).

Vergleich der Trafomatrizen. Wie bereits in Unterpunkt 3.2.4 beschrieben, wurden acht Trafomatrizen bestimmt, die jeweils eine Abbildung vom $\mathcal{GF}(2^8)$ auf das $\mathcal{GF}((2^4)^2)$ ermöglichen. Die folgenden Abschnitte beschäftigen sich damit, welche der Matrizen für eine flächenoptimierte Umsetzung der Byte-Substitutions-Komponenten besonders gut geeignet sind.

Auch hier findet der in Sektion 7.2 vorgestellte Greedy-Algorithmus Anwendung. Er untersucht alle Lösungen hinsichtlich

- der minimalen Gatterzahl und
- des kürzesten kritischen Pfads.

Bisher wurde angenommen, dass im kritischen Pfad der Operationen 1 und 8

		zwei 16x8 Matrizen		vier 8x8 Matrizen	
Matrix Nr.	Element f. Transformation	Gatter #XOR	krit. Pfad Op.1/Op.8	Gatter #XOR	krit. Pfad Op.1/Op.8
1	$\beta = \alpha^5$	52	3/3	62	3/3
2	$\beta = \alpha^{10}$	48	4/4	57	4/4
3	$\beta = \alpha^{20}$	50	3/4	54	4/3
4	$\beta = \alpha^{40}$	52	3/4	55	3/5
5	$\beta = \alpha^{65}$	**44**	4/3	53	4/3
6	$\beta = \alpha^{80}$	51	4/3	58	4/4
7	$\beta = \alpha^{130}$	50	4/4	60	3/4
8	$\beta = \alpha^{160}$	**44**	4/3	**52**	4/3

Tabelle 4: Vergleich der Gatterzahl und der kritischen Pfade der acht Trafomatrizen.

drei XOR-Gatter liegen. Bei Einsatz des Greedy-Algorithmus kann seine Länge jedoch sowohl nach oben als auch nach unten abweichen. Dies ist abhängig davon, welche Terme in welcher Reihenfolge zusammengefasst werden. Die Qualität der optimierten Hardwareumsetzung dieser Operationen hängt also auch von der berechneten Länge des kritischen Pfads ab.

Es ist möglich, mehrere Matrixberechnungen mit einer Logikeinheit darzustellen. Die Logikeinheiten von $(\mathbf{K}_{inv0} \cdot p + k_{inv0})$ und $(\mathbf{M} \cdot a)$ benutzen dieselben Eingänge, nämlich nacheinander die Register RA, RB, RC und RD. Folglich können sie zu einer kleineren Einheit verschmolzen werden, in der identische logische Verknüpfungen mit denselben Gattern umgesetzt werden. Ebendies gilt für die Logikeinheiten der Operationen $(\mathbf{K}\cdot\{d_1, d_0\}+f_0)$ und $(\mathbf{M}^{-1}\cdot\{d_1, d_0\})$ (Operation 8 aus Tab. 3).

Der Greedy-Algorithmus optimiert demnach nicht die vier 8x8 Matrizen $\mathbf{M}$, $\mathbf{M}^{-1}$, $\mathbf{K}$ und $\mathbf{K}_{inv}$ getrennt, sondern nur zwei 16x8 Matrizen. So fasst man $\mathbf{M}$ und $\mathbf{K}_{inv}$ bzw. $\mathbf{M}^{-1}$ und $\mathbf{K}$ zusammen. Tabelle 4 stellt die resultierende Gatterzahl und die dazugehörigen kritischen Pfade dar.[70] Zum Vergleich werden in den beiden rechten Spalten die Ergebnisse für die separate Optimierung der vier Matrizen

[70]Die Einheit der kritischen Pfade der Operationen 1 und 8 ist die Anzahl maximal hintereinandergeschalteter XOR-Gatter. Da in der rechten Spalte jede der Operationen zwei Matrizen separat optimiert, wird jeweils das Maximum der zwei Ergebnisse als kritischer Pfad eingetragen.

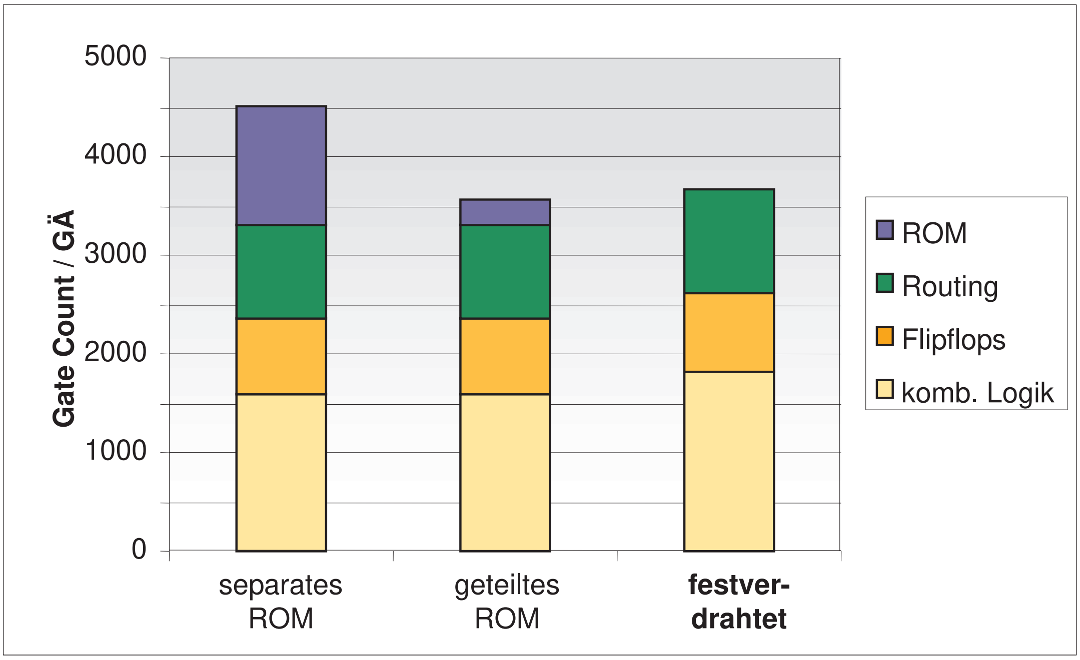


Abbildung 42: Vergleich der CpCiph-Implementierungen mit ROM oder mit festverdrahteter Logik für die Byte-Substitutions-Komponenten.

aufgeführt. Die fettgedruckten Werte markieren die besten Lösungen.

Im Rahmen dieser Arbeit wurden die Logikeinheiten für Lösungsmatrix Nr. 8 implementiert. Das Zusammenfassen der jeweils zwei Matrizen für die Operationen 1 und 8 führt zu einem geringeren Gatteraufwand von 44 statt 52 Gattern und zu einer zusätzlichen Verbesserung um 15,3% gegenüber der Lösung mit den bereits optimierten vier 8x8 Matrizen. Beide Ergebnisse sind deutlich besser als die Resultate für ein komplett ohne Enhanced-Greedy optimiertes System, bei dem Gatter mit gleichen Eingängen nicht doppelt genutzt werden. Ein solches System benötigt 88 XOR-Verknüpfungen.

ROM vs. festverdrahtete Logik. Nach der Bestimmung der minimalen Gatterzahl für eine komplett festverdrahtete Lösung im letzten Unterpunkt, vergleicht der kommende Abschnitt die benötigte Fläche der kombinierten Byte-Substitutions-Komponenten für folgende Implementierungsvarianten:

- ausschließlich festverdrahtete Logik und
- festverdrahtete Logik für die (invers) affine Transformation und ein 256

Byte großes ROM für die Inversion im $\mathcal{GF}(2^8)$.

Es wird somit hinterfragt, ob eine Lösung ganz ohne ROM Vorteile einbringt. Abbildung 42 fasst die Ergebnisse für den CPCIPH zusammen,[71] die Anhang C in tabellarischer Form vorstellt. Die Balken präsentieren die resultierende Fläche der drei Implementierungsvarianten. Jeder der Balken unterteilt sich dabei in den Flächenanteil der kombinatorischen Logik, der Flipflops, des Routingaufwands und der LUT im ROM.

Abbildung 43: Gemeinsame Nutzung eines ROMs durch Programmspeicher und LUT.

Der linke Balken schätzt das Ergebnis für ein separates ROM für die 256 Byte große LUT ab. Hingegen gibt der mittlere die Resultate für die Implementierung wieder, bei der Programmspeicher und LUT in einem gemeinsamen ROM liegen (vgl. Abb. 43). Bei dieser Lösung nutzt der Coprozessor den gemeinsamen Datenbus für Zugriffe auf das ROM. Da die Zeilenlogik des Programmspeichers bereits zur Verfügung steht, ist lediglich die Größe der zusätzlichen Speicherzellen und Spaltenlogik zu berücksichtigen.

Der rechte Balken wertet die Syntheseergebnisse des CPCIPH mit ausschließlich festverdrahteter Logik aus.

Die Gesamtfläche des Digitalteils setzt sich aus der Kernzellfläche, also der reinen Gatterfläche für kombinatorische Logik und Flipflops, und der Fläche für Routing zusammen. Dies nähert die Abschätzung

$$A_{digit} \approx 1.4 \cdot A_{kern}$$

an.[72] Die Gesamtfläche des CPCIPH entspricht der Summe der Digitalteil- und der ROM-Fläche.

Aus den vorliegenden Ergebnissen geht hervor, dass die Lösung mit einem *separaten* ROM im Widerspruch zum Ziel der Kostenminimierung steht. Aus diesem

[71] Es handelt sich um eine CPCIPH-Realisierung mit sequentiellem Interface.

[72] Die Näherung wurde aus dem Verhältnis festverdrahtete Logik des Digitalteils zu Kernzellfläche des Layouts eines Gesamtsystems mit CPCIPH berechnet.

Grund wird sie fortan nicht weiter betrachtet.

Die festverdrahtete Lösung verfügt über eine ca. 11% größere Kernzellfläche als die Version mit einem ROM. Berücksichtigt man jedoch auch die Größe des LUTs im geteilten ROM, dann fallen die Unterschiede deutlich geringer aus: Die Gesamtfläche der festverdrahteten Lösung ist lediglich um ca. 3% größer.

Dafür weist sie gegenüber der Prozessor-ROM-Lösung den Vorteil auf, weniger auf den Datenbus zuzugreifen. Da der Datenbus zeitlich den Engpass des CPCIPH bildet, werden bei der festverdrahteten Umsetzung weniger Taktzyklen zur Durchführung einer Ver- oder Entschlüsselung benötigt.
Bei der Implementierung mit dem parallelen bzw. hybriden Interface entstehen für diese Coprozessorrealisierung keine Wartezyklen durch Zugriffsversuche auf einen belegten ROM-Bus. Zudem lassen sich durch Erweiterung um eine zweite festverdrahtete Logikeinheit parallel zwei Bytes substituieren. Die Programmspeicherlösung bietet diese Möglichkeit nicht. Aus diesem Grund wird in den weiteren Ausführungen stets der CPCIPH mit festverdrahteter Byte-Substitutionslogik verwendet.

7.3.4 Lokale Zustandsmaschinen

Die lokalen Zustandsmaschinen werden in Unterpunkt 6.2.2 eingeführt, um zeitgleich mehrere Komponenten bearbeiten zu können. In einem ersten Schritt erhält dabei jede der sieben Operationsrealisierungen eine eigene lokale Ablaufsteuerung.

Tabelle 5 führt die Anzahl der benötigten Zustände[73] eines lokalen Automaten auf und ermittelt daraus die Anzahl der für die FSMs benötigten Flipflops. Bei einem Zustandswechsel des globalen endlichen Automaten wird die lokale FSM initiiert und wechselt ihren Zustand ab dann in jedem Taktzyklus.[74]

Die Idee für die flächenoptimierte Umsetzung dieser FSMs besteht darin, einen lokalen Automaten für mehrere Komponenten nutzbar zu machen. Komponenten, die sich eine FSM teilen, müssen eine der folgenden Eigenschaften erfüllen:

[73]Die Anzahl der Zustände entspricht bei den lokalen FSMs der Zyklenzahl und wird hier deshalb äquivalent genutzt.

[74]Bei den lokalen FSMs handelt es sich um Dualzähler [Tie91], die nach Initiierung in jedem Taktzyklus inkrementieren.

Komponente	Zustände	Flipflops
`DataFetch32`	5	3
`AddRoundKey32`	6	3
`SubBytes32/Rotate32`	5	3
`InvSubBytes32`	5	3
`MixColumns32`	2	1
`InvMixColumns32`	3	2
`WriteBack32`	4	2
Gesamt		17

Tabelle 5: Zustände der lokalen endlichen Automaten.

- Sie gehören zu unterschiedlichen Instruktionen. Beispiel: Die `MixColumns32`-Komponente wird nur von der Chiffrierinstruktion genutzt, `InvMixColumns32` nur von der Dechiffrierinstruktion.

- Sie werden von derselben Instruktion aufgerufen, arbeiten aber zu keinem Zeitpunkt parallel. Beispiel: Die `MixColumns32`- und `AddRoundKey32`-Komponenten werden beide von der Chiffrierinstruktion genutzt, arbeiten aber niemals in demselben Takt.

Nachfolgende Operationsrealisierungen können sich entsprechend eine FSM teilen:

- `AddRoundKey32`, `MixColumns32` und `InvMixColumns32`,

- `SubBytes32` und `InvSubBytes32`,

- `DataFetch32` und `WriteBack32`.

Die Anzahl der erforderlichen Flipflops bestimmt jeweils die Komponente, welche die meisten Zustände benötigt. Diese Zusammenfassung bewirkt, dass nur noch 3 der 7 FSMs mit insgesamt 9 statt 17 Flipflops eingesetzt werden. Abb. 44 zeigt anhand der `AddRoundKey32`-, `MixColumns32`- und `InvMixColumns32`-Komponenten die Reduzierung von drei FSMs auf eine.

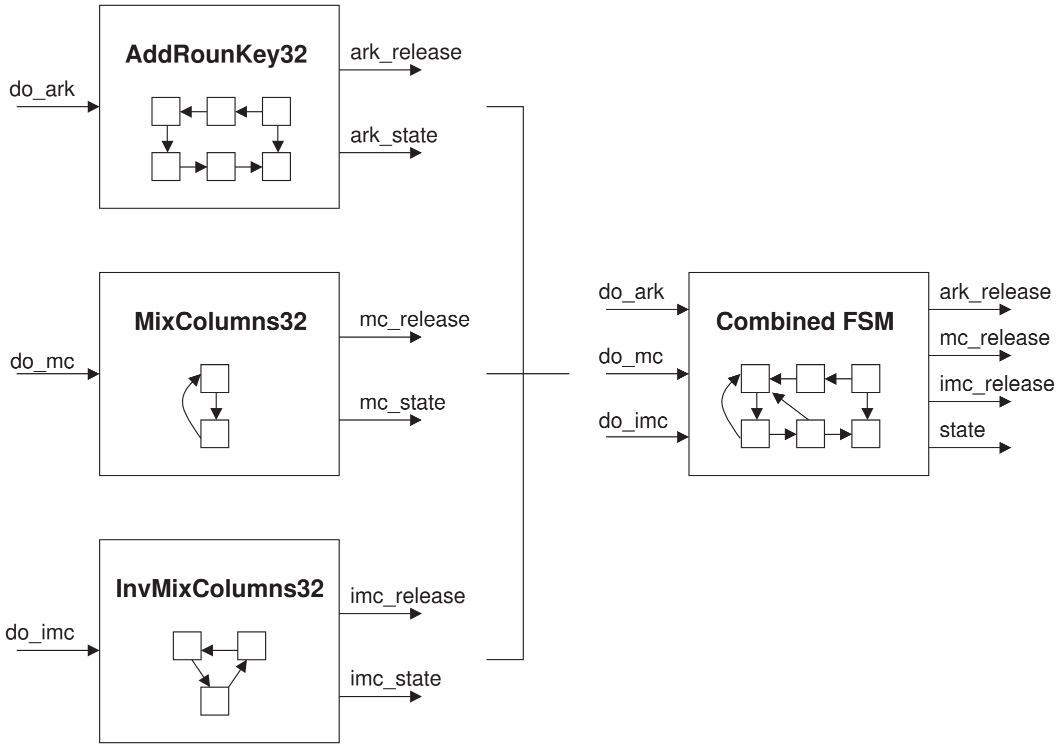


Abbildung 44: Ersetzen von drei lokalen FSMs durch eine mehrfachgenutzte lokale FSM.

7.3.5 Adressgenerierung

Der CPCIPH verfügt über eine eigene Address Generation Unit (AGU), welche die Adressberechnung all der Komponenten übernimmt, die Zugriff auf den gemeinsamen Speicher anfordern:

- `DataFetch32` liest einen Eingangsvektor und Zwischenergebnisse aus dem Speicher,
- `WriteBack32` schreibt Zwischenergebnisse und den Ausgangsvektor in den Speicher zurück,
- die Byte-Substitutions-Komponenten lesen bei Implementierung mit einem ROM die Werte aus einer LUT aus und
- `AddRoundKey32` benötigt die aktuellen Rundenschlüssel.

Die AGU wurde eingeführt, um Daten im gemeinsamen Speicher anstatt in flächenaufwendigen coprozessorinternen Registern abzulegen. Daher ist es besonders wichtig, die Fläche und damit auch die Anzahl der Flipflops der AGU minimal zu halten.

Die Adressberechnung aller Komponenten basiert auf den sogenannten Basisadressen. Diese können als Zeiger auf die Startadresse eines Datenblocks im Adressraum interpretiert werden. Der CPCIPH benötigt Basisadressen für

- den Eingangsdatenblock (Encrypt: Nachricht, Decrypt: Chiffre),
- den Ausgangsdatenblock (Encrypt: Chiffre, Decrypt: Nachricht),
- ein Scratchpad, in dem ein Zustandsblock nach einer Runde abgelegt wird,
- den geheimen Schlüssel,
- den expandierten Schlüssel und
- evt. für die LUT der Byte-Substitution.

Die Basisadressen können entweder fest vorgegeben oder in BA-Registern gespeichert werden. Feste Basisadressen erfordern einen geringen Hardwareaufwand, sind allerdings unflexibel: Vor der Synthese des Designs müssen sie festgelegt werden und sind ab dann nicht mehr änderbar. Insbesondere für Datenblöcke wie die zu verschlüsselnde Nachricht ist diese Unflexibilität unerwünscht, da diese sonst vor der Verschlüsselung mit hohem zeitlichem Aufwand an die geforderte Speicherstelle kopiert werden müssen.

Legt man die Basisadressen in Registern ab, so kann deren Inhalt mittels einer Load-Instruktion beliebig geändert werden. Die Datenblöcke können somit verstreut im Speicher liegen. Der CPCIPH muss sie vor Bearbeitung nicht umkopieren, sondern lediglich den Zeiger auf die Startadresse neu laden. Die LUT und das Scratchpad werden im Rahmen dieser Arbeit stets auf feste Adressen gelegt.

Aufgrund des hohen Komforts wurden für alle bisher beschriebenen Syntheseergebnisse 16-Bit BA-Register für Ein- und Ausgangsdatenblock sowie den geheimen Schlüssel bereitgestellt. Die Schlüsselexpansion schreibt an eine feste Speicherstelle. Für die BA-Register dieser Version werden insgesamt 48 D-Flipflops benötigt.

Für eine flächenreduzierte Version kann die Anzahl der Flipflops gesenkt werden. Bei dieser existiert lediglich ein gemeinsames BA-Register für Ein- und

Ausgangsdaten. Das bedeutet, dass die Eingangsdaten bei Ausführung der Instruktion geladen und durch die Ausgangsdaten überschrieben werden. Das BA-Register besteht aus nur noch 12 statt 16 Flipflops. Damit liegen die 16-Byte-Blöcke auf durch 16 teilbaren Adressen und sind nicht mehr frei im Speicher platzierbar. Es werden insgesamt 36 Flipflops eingespart.

Nach einer ASIC-Synthese unterscheiden sich die beiden Versionen um 459 GÄ (2168 GÄ statt 2627 GÄ), also insgesamt um etwa 20% der gesamten Coprozessorfläche.

7.4 Optimierung der CPMAC Komponenten

Da der CPMAC das schlichte Interface nutzt (vgl. 5.1.2), gibt es die zeitliche Restriktion, dass sämtliche arithmetische und logische Instruktionen in höchstens zwei Taktzyklen durchgeführt werden sollen. Damit stehen für eventuell nachfolgende Befehle sämtliche Ressourcen zur Verfügung. Der folgende Unterpunkt erörtert ausschließlich die Maßnahmen, die es ermöglichen, sowohl diese Beschränkungen einzuhalten als auch eine geringere Gatterzahl als der Core 3311C zu benötigen.

7.4.1 Multiplikation und MAC-Operation

Die Multiplikation $\texttt{MA} = \texttt{FA} \cdot \texttt{FB}$ und die MAC-Operation $\texttt{MA} = \texttt{MA} + \texttt{FA} \cdot \texttt{FB}$, im Zweierkomplement berechnet, sind die beiden zeitkritischsten Operationen des CPMAC. Die wichtigsten Maßnahmen zur Reduzierung des Pfads unter Berücksichtigung der Gatterminimierung werden in diesem Unterpunkt erläutert.

Eliminierung der Hilfsregister. Die vorzeichenlose Multiplikation kann in binärer Logik durch Addition der Partialprodukte (PP) durchgeführt werden.[75] Die 3-Bit-Multiplikation $\texttt{\%111} \cdot \texttt{\%101}$ berechnet sich z.B. folgendermaßen:

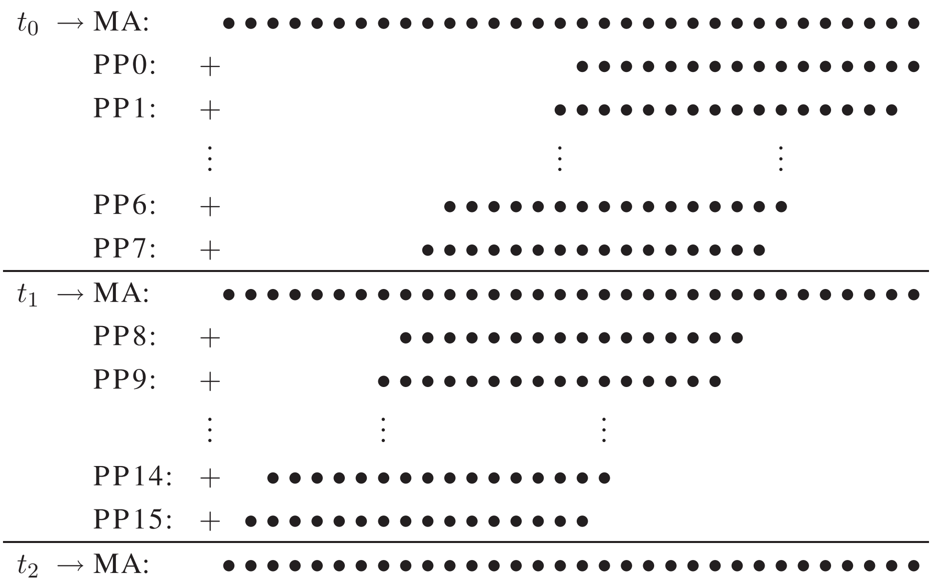
[75] vgl. [Was82], S.131

(1)	1	1	1	·	1	1	0
(2)					0	0	0
(3)	+			1	1	1	
(4)	+		1	1	1		
(5)		1	0	1	0	1	0

Dabei stellen die Zeilen (2), (3) und (4) die PP dar, die zusammengerechnet das Gesamtergebnis in Zeile (5) ergeben.

Multipliziert man wie im Fall des CPMAC die beiden 16-Bit-Faktorregister `FA` und `FB` miteinander, so entstehen 16 PP, die aufaddiert werden müssen. Unterteilt man die sechzehn PP auf zwei Taktzyklen, so verbleiben pro Taktzyklus acht PP. Nach der Addition der ersten 8 PP muss das Ergebnis zwischengespeichert werden. Da die PP steigende Wertigkeit haben, werden nicht 16, sondern mindestens 24 Flipflops zur Zwischenspeicherung nach dem ersten Taktzyklus benötigt.

Auf diese Flipflops kann verzichtet werden, wenn die 8 PP für die MAC-Operation direkt zum Akku `MA` addiert werden und das Zwischenergebnis danach wieder in MA abgelegt wird. Die gesamte Multiplikation wird also wie dargestellt durchgeführt:



Jede Binärstelle wird dabei durch einen Punkt repräsentiert. Die obenstehende Berechnung geht von dem einfachen Fall einer vorzeichenlosen Multiplikation der beiden Faktorregister aus, deren Ergebnis zum unsigned Wert im Akkumulator zu addieren ist. Da die Werte dieser Register beim CPMAC jedoch als Zweierkomplementzahlen zu interpretieren sind, sollten die PP gleich vorzeichenkorrekt aufgetragen werden, um zu einem korrekten Ergebnis bei Addition zum vorzeichenbehafteten Akkumulator zu führen.
Zum Zeitpunkt t_0 werden die ersten acht PP zu dem Inhalt von `MA` addiert. `MA` übernimmt diese zum Zeitpunkt t_1, also nach dem ersten Taktzyklus. Nach zwei Taktzyklen beinhaltet `MA` das Gesamtergebnis.

Die Addition des Akkus `MA` im ersten Taktzyklus ist nur dann vonnöten, wenn die MAC-Operation vorliegt. Bei der Multiplikation entfällt die Addition von `MA` im ersten Taktzyklus und es wird lediglich ein 32 Bit breiter Nullwert zu den 8 PP addiert.

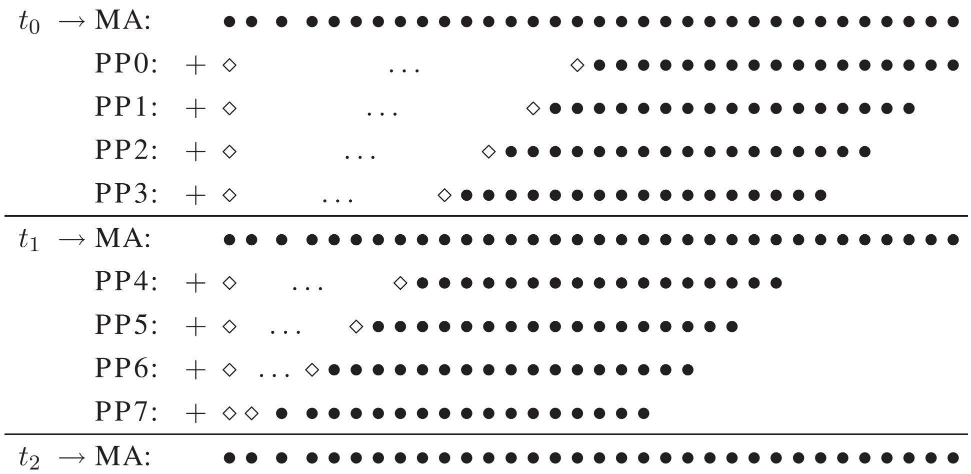
Reduktion der Partialprodukte. Pro Taktzyklus muss eine Addition von acht PP und dem Akku berechnet werden. Der Hardwareaufwand und der kritische Pfad hierfür sind sehr groß. Verwendet man die Multiplikation mit dem modifizierten Booth-Algorithmus, eine Weiterentwicklung des Booth-Algorithmus in [Boo51], so benötigt man nur noch die Hälfte der PP.

Die Reduzierung der PP mit dem Booth-Algorithmus wird erreicht, indem jedes zweite PP durch die anderen ersetzt wird. Der Multiplikand wird zur Berechnung eines PP unterschiedlich gewichtet: Die zur Gewichtung gültigen Faktoren -2, -1, 0, 1 und 2 werden vom Booth-Encoder berechnet. Wird z.B. die Multiplikation $(X \cdot \texttt{\%0111})$ üblicherweise durch $(2^2 \cdot X + 2^1 \cdot X + 2^0 \cdot X)$ vollzogen, so wird mit dem modifizierten Booth-Algorithmus $(2 \cdot 2^2 \cdot X + (-1) \cdot 2^0 \cdot X)$ berechnet. Das PP der Wertigkeit (2^1) wird durch die unterschiedlich gewichteten PP für (2^0) und (2^2) ersetzt.[76]

Insgesamt müssen nur noch 8 statt 16 PP addiert werden, also 4 pro Taktzyklus. Unter Berücksichtigung der Zweierkomplementdarstellung kann die MAC-

[76] Für eine genaue Beschreibung des Booth-Encoders sowie der Partialproduktberechnung für den modifizierten Booth-Algorithmus wird auf [Was82] verwiesen.

Operation folgendermaßen wiedergegeben werden:[77]



Die Punkte stellen abermals die einzelnen Binärstellen dar, während die Rauten die Sign Extension[78] repräsentieren. Sowohl die Zahl der benötigten Gatter als auch der kritische Pfad der MAC-Operation und der Multiplikation werden durch Verwendung des erweiterten Booth-Algorithmus signifikant gesenkt.

Mehrfachnutzung von Hardware. Die Berechnung der PP mit Hilfe der Booth-Encoder ist für beide Taktzyklen identisch, lediglich die *Eingänge* der vier Booth-Encoder sind unterschiedlich: So kann z.B. der Encoder, der im ersten Takt die Gewichtung für PP0 berechnet, im zweiten Takt jene für PP4 bestimmen. Dafür müssen abhängig vom Takt die Eingänge des Booth-Encoders gemultiplext werden.

---

[77] Bei der Berechnung der PP mit dem Booth-Algorithmus ist es möglich, dass der Multiplikand mit einem negativen Faktor multipliziert wird. Im Zweierkomplement erfordert die Multiplikation mit (-1) die Inversion der Bits mit anschließender Addition von 1. Dieses Inkrement, welches für jedes PP berücksichtigt werden muss, wird hier aus Gründen der Übersichtlichkeit nicht dargestellt.

[78] Die Sign Extension ist für die Addition im Zweierkomplement vonnöten. Eine negative Zahl wird im Zweierkomplement durch eine führende 1, eine positive Zahl durch eine führende 0 repräsentiert. Das MSBit entspricht also einem Vorzeichenbit (Sign Bit). Soll die Bitbreite erhöht werden, ohne den Wert zu verändern, so müssen alle links vom bisherigen MSBit stehenden Bits den Wert des Sign Bits annehmen (Sign Extension). Bsp.: `%1001` (bei vier Bit) entspricht genau wie `%11001` (bei fünf Bit) oder `%111001` (bei sechs Bit) dem Dezimalwert (-7).

Betrachtet man die Punktdarstellung der Multiplikation mit dem Booth-Algorithmus genauer, so wird deutlich, dass auch die Addition der PP zum Akku `MA` in beiden Taktzyklen annähernd identisch ist: Die PP werden im ersten Takt zu `MA[31:0]` addiert. Das Ergebnis wird in `MA[31:0]` gesichert.

Die im zweiten Taktzyklus auf den Eingang der Addierer gelegten Bits `MA[31:8]`, nutzen dieselben Hardware-Addierer wie im ersten Takt. Die dabei berechnete Summe wird in `MA[31:8]` gesichert, an `MA[7:0]` ändert sich im zweiten Taktzyklus nichts.

Die Addition der 4 PP und des Akkus `MA` erfolgt mit Carry-Save-Addern (CSA) durch Nutzung einer dem Wallace-Tree ähnlichen Struktur, da hierbei nur eine geringe Anzahl Volladdierer benötigt wird. CSA finden Anwendung, wenn mehr als zwei Summanden zu addieren sind. Nach CS-Addition verbleibt lediglich eine Additionsstufe mit zwei Summanden.
Diese letzte Additionsstufe wird mit einem dem Carry-Look-Ahead- (CLA) Adder ähnlichen Addierer berechnet, der zwei 32 Bit breite Eingänge addiert und ein 32 Bit breites Ergebnis zuzüglich eines Overflow-Bits erhält.
Der CLA-Adder ist im Vergleich z.B. zu einem einfachen Ripple-Addierer nicht kostenoptimal, verfügt aber über einen deutlich kürzeren kritischen Pfad und ermöglicht somit die Durchführung einer MAC-Operation in zwei Taktzyklen. Die verwendeten Addiererstrukturen werden in der Literatur eingehend behandelt und können detailliert in [Rab02], [Tka00], [Was82] oder [Won85] nachvollzogen werden.

Nach seiner Optimierung benötigt der CpMac insgesamt lediglich 66 Volladdierer, wobei drei Volladdierer im kritischen Pfad liegen.

7.4.2 Addition und Subtraktion

Die Komponente für die 32-Bit Addition `MA` = `MA` + {`FA`, `FB`} und Subtraktion `MA` = `MA` - {`FA`, `FB`} nutzt die letzte Additionsstufe der Multiplikation und der MAC-Operation, die mit dem CLA-Adder durchgeführt wird. Aus diesem Grund können diese beiden Operationen auf einen eigenen Addierer verzichten. Dies ist relevant, da der CLA-Adder eine sehr hohe Gatterzahl benötigt. Addition und Subtraktion werden in einem Taktzyklus berechnet.

Einen Überblick über die verwendete Addiererstruktur gibt Abb. 45. Der obere Teil mit dem Booth-Encoder und den CSA wird ausschließlich für Multiplika-

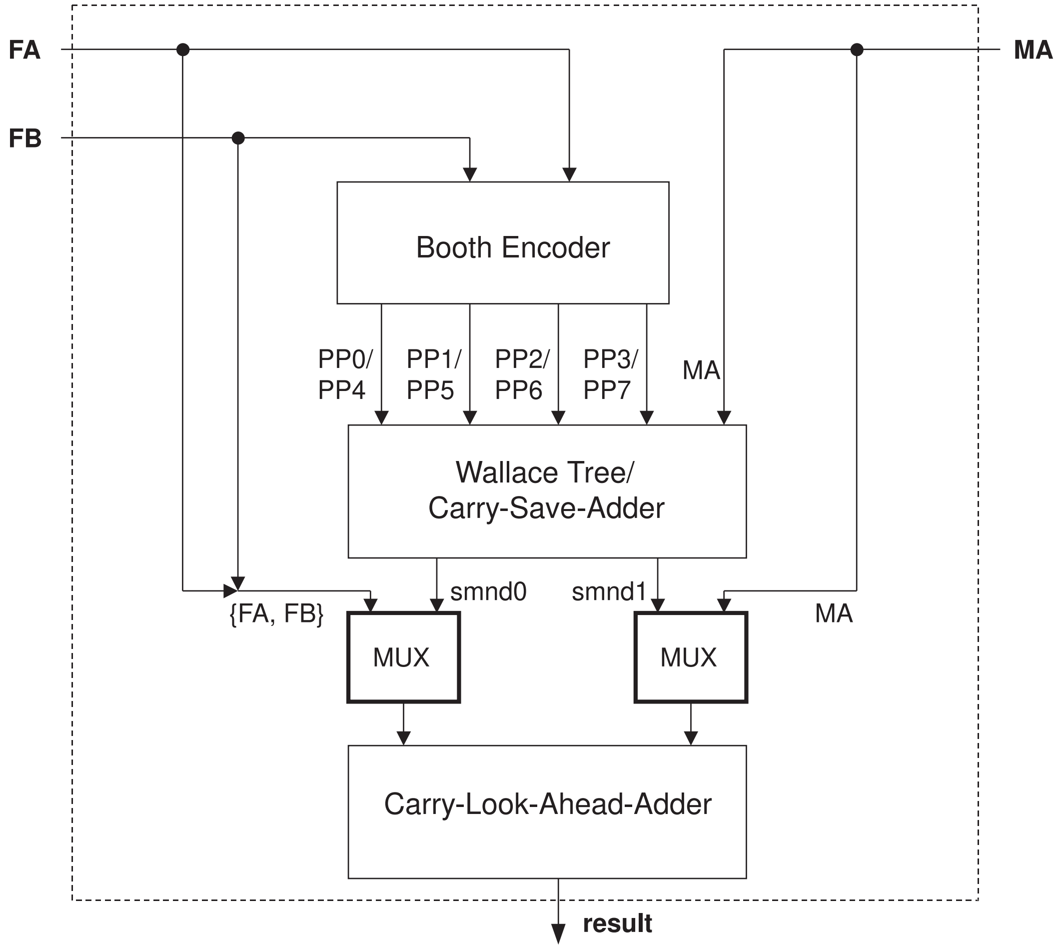


Abbildung 45: Gemeinsame Nutzung der Addiererstruktur von Addition, Subtraktion, Multiplikation und MAC-Operation.

tion und MAC-Operation verwendet. Dagegen wird abhängig von der aktuellen Instruktion entschieden, ob der CLA-Adder das Ergebnis für eine MULT- bzw. MAC- oder eine ADD- bzw. SUB-Operation berechnet. Das Resultat übernimmt der Akkumulator `MA` mit steigender Taktflanke.

7.4.3 Barrelshifter

Eine erweiterte Variante des CPMAC ermöglicht Links- und Rechtsshifts des `MA`-Wertes mit einem sogenannten Barrelshifter. Die einfachste Implementierungsmöglichkeit führt einen Shift zwischen z.B. 0 und 15 Stellen in einem Taktzyklus durch. Der Barrelshifter besteht dann aus vier Stufen. Mittels ei-

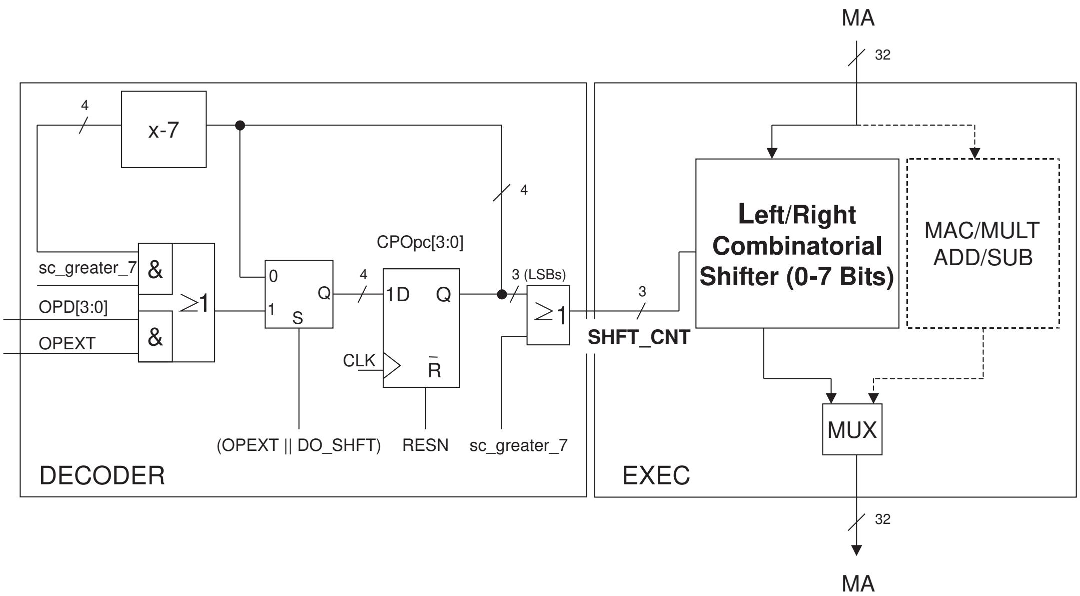


Abbildung 46: Barrelshifter für maximal 14-stelliges Links- und Rechtsshiften in zwei Takten.

ner baumartigen Struktur kann in der ersten Stufe um 1 Bit, in der zweiten um 2, in der dritten um 4 und in der letzten um 8 Bit entweder nach links oder nach rechts geschoben werden. Desweiteren benötigt der Shifter eine Dekodierlogik, welche entscheidet, ob in der jeweiligen Stufe nach links, nach rechts oder überhaupt nicht geschoben wird (vgl. [Was82]).

Da der in dieser Arbeit beschriebenen MAC-Unit aufgrund der gewählten Schnittstelle ohnehin zwei Taktzyklen zur Berechnung aller arithmetischer Operationen zur Verfügung stehen, lässt sich eine der vier Shiftstufen einsparen, wenn pro Taktzyklus jeweils zwischen 0 und 7 Stellen verschoben wird. Abb. 46 veranschaulicht dies. Die Shift-Instruktionen ermöglichen somit das Verschieben zwischen 0 und 14 Stellen. Das Zwischenergebnis nach dem ersten Takt und das Endergebnis hält der Akkumulator `MA`.

Die Anzahl der zu verschiebenden Bits sind ein vier Bit großer Bestandteil des Opcodes `OPD` bei den ASLM-[79] und ASRM-[80] Instruktionen. Dieser wird bei gesetztem `OPEXT` in das Register `CpOpc` übernommen. Soll um mehr als sieben Stellen verschoben werden (`sc_greater_7`), so ist die Shift-Operation folgender-

[79] Arithmetic Shift Left `MA`

[80] Arithmetic Shift Right `MA`

maßen aufgeteilt:

- 1. Taktzyklus: Unabhängig von der Anzahl der zu verschiebenden Stellen wird ein Shift um 7 Stellen durchgeführt.
- 2. Taktzyklus: Ein Shift um die verbleibende Anzahl an Stellen findet statt.

Somit muss nach dem ersten Taktzyklus die Anzahl der zu verschiebenden Stellen um 7 verringert werden und im `CpOpc` Register abgelegt werden. Dies kann durch Subtraktion von 8, also Löschen des Bits der Wertigkeit 2^3, und das Inkrement der drei LSBs erreicht werden. Nach vollzogenem Schiebevorgang wird das Register `CpOpc` gelöscht. Bei einer anderen als einer Schiebe-Instruktion behält das Register seinen Inhalt bis nach Abarbeitung dieser Instruktion bei (s. linker Multiplexer in Abb. 46).

Auch Links- und Rechtsshifts führen bei Überschreiten der positiven oder negativen Maxima den Inhalt von `MA` in die Sättigung. Dies ist mittels Branch-Instruktionen abfragbar. Das die Sättigung anzeigende Saturation Flag wird mit der nächsten auf den Akkumulator schreibenden Operation zurückgesetzt.

Aufgrund des beachtenswerten Flächenzuwachses durch die kombinatorische Schiebeeinheit um etwa 15% gibt es CPMAC-Versionen mit und ohne Barrelshifter. Letztere Version verfügt ersatzweise über arithmetische Schiebeoperationen um eine Stelle.

8 Umsetzung und Bewertung

Dieses Kapitel setzt sich mit der Implementierung des CPCIPH und CPMAC auseinander und überprüft, ob die an sie gestellten Erwartungen erfüllt werden. Abschließend findet eine Bewertung des CPCIPH insbesondere unter Berücksichtigung anderer Hardware- und Softwarelösungen statt. Zunächst sei jedoch kurz auf den Test der Architekturen und die Testbarkeit des Chips eingegangen.

8.1 Verifikation und Test

Es bedarf insbesondere einer konsequenten und auf allen Implementierungsstufen berücksichtigten Testumgebung, um einen Chip *hardwareoptimal* implementieren zu können. Dies ermöglicht das Prüfen des Designs sowohl in der Entwicklungsphase als auch am gefertigten Chip.

Aus diesem Grund ist die Simulation der Coprozessoren auf mehreren Hierarchieebenen sinnvoll, um die Funktionalität der Designs zu gewährleisten. Desweiteren können die Coprozessoren mit Hilfe des etablierten Standards IEEE 1149.1 (auch als JTAG[81] bekannt) mit Scanpfaden ausgestattet werden, die für das System einen Testzugang sowohl für das Rapid Prototyping mit FPGAs als auch für den gefertigten Chip zur Verfügung stellen.

8.1.1 Verilog-Simulation der Coprozessoren

Zur Durchführung sämtlicher Simulationen wird in der vorliegenden Arbeit der Verilog-XL Design Simulator von Cadence benutzt.

Durch die für die Testumgebung gewählte hierarchische Teststruktur besteht die Möglichkeit, Fehler bei der Simulation schnell und präzise zu lokalisieren. Auf unterster Hierarchieebene stellt eine automatisierte Testumgebung Tests für die separate Simulation von Decoder, Control Unit, Address Generation Unit, Register Unit und Execution Unit zur Verfügung.

Desweiteren wird die Funktionalität der *gesamten* Beschleunigungseinheiten durch umfangreiche Testbenches verifiziert. In einem letzten Simulationsschritt

[81] JTAG: Joint Test Action Group [Jtag90]

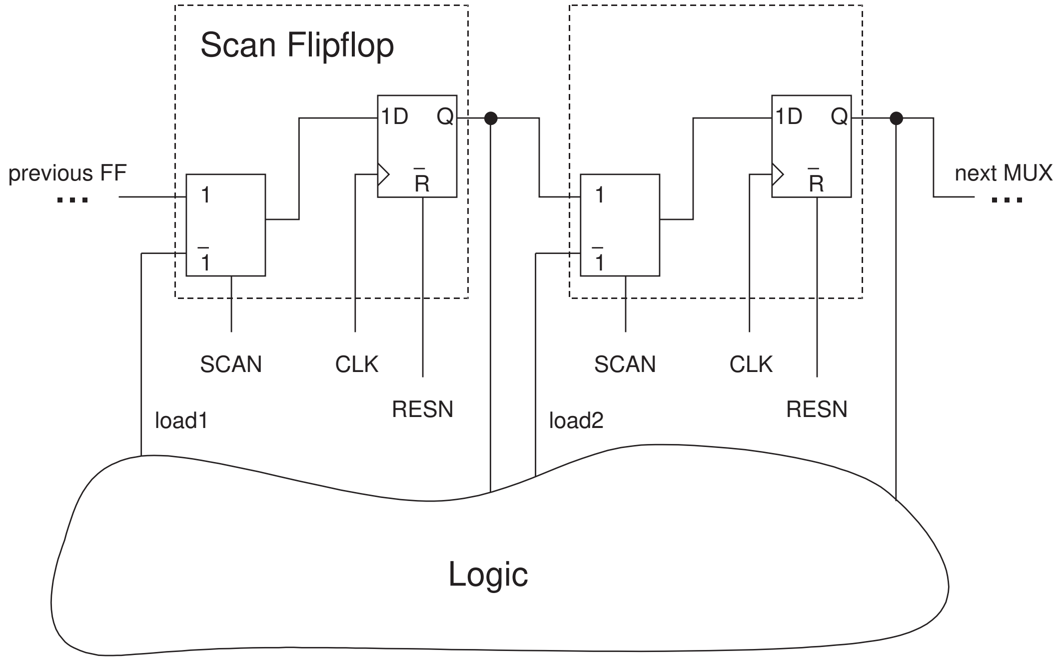


Abbildung 47: Serielle Verschaltung von Scan-Flipflops zu einem Scanpfad.

werden die Coprozessoren in ein IMS3311C Beispielsystem integriert und mit Assembler-Testprogrammen überprüft.

Die oben erwähnten Hierarchieebenen des Tests sind für verschiedene Versionen des CPCIPH implementiert, beim CPMAC wurden bisweilen die niedrigeren Stufen eingespart. Die verwendeten Testvektoren für den CPCIPH stammen von der AES-Homepage und wurden von der NIST bereitgestellt [NIST04]. Weitere Testvektoren liefert ein im Rahmen dieser Arbeit erstelltes C-Modell des AES, das für beliebige Eingangsblöcke die dazugehörigen ver- oder entschlüsselten Ausgangsblöcke berechnet.

8.1.2 Integration einer IEEE 1149.1 Testschnittstelle

Sämtliche erstellte Coprozessormodelle sind optional auch mit JTAG-Scanpfaden ausrüstbar. Der Standard IEEE 1149.1 wird in dieser Arbeit zum seriellen Ein- und Auslesen von Flipflop-Inhalten genutzt, die einen Test des Chips auch nach der Produktion ermöglichen. Die JTAG-Fähigkeit des Systems ist generierbar und kann je nach Bedarf hinzugefügt werden.

Die Idee der Testschnittstelle ist es, dass jedes D-Flipflop durch ein Scan-Flipflop

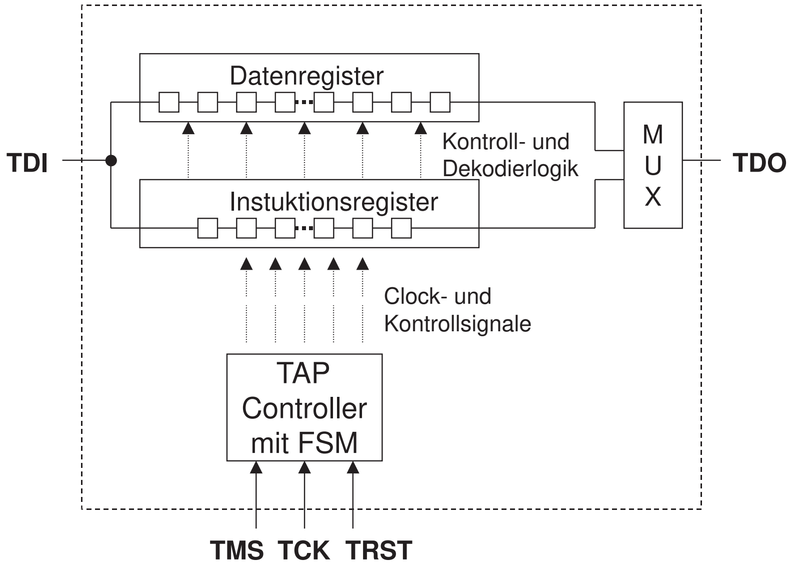


Abbildung 48: Schaltbild der IEEE 1149.1 Testlogik.

ersetzt wird, d.h. vereinfachend durch ein D-Flipflop mit vorgeschaltetem Multiplexer. Abb. 47 veranschaulicht dies.

Die Flipflops sind in zwei verschiedenen Modi betreibbar:

1. im Run-Modus und
2. im Scan-Modus.

Während die Scan-Flipflops im Run-Modus mit steigender Flanke den aktuellen Wert der Eingangslogik übernehmen (`load1` und `load2`), werden sie im Scan-Modus über die Multiplexer zu einem Scanpfad verbunden, um die aktuellen Flipflop-Inhalte seriell aus dem Chip herauszuschieben.

Das Schema der JTAG-Testlogik präsentiert Abb. 48. Die Schnittstelle verfügt über einen TAP (Test Access Port), der aus den Leitungen "Test Data Input" (TDI), "Test Data Output" (TDO), "Test Mode Select" (TMS), "Test Clock" (TCK) und optional "Test Reset" (TRST) besteht (vgl. [Jtag90]). TDI und TDO stellen dabei die seriellen Ein- und Ausgänge des zu testenden Systems dar, während TCK das Taktsignal im Scan-Modus führt. Das TMS-Signal ist das Steuersignal des TAP Controllers. Dieser verfügt über einen endlichen Automaten mit 16 Zuständen, der die Teststeuerung übernimmt. Er ermöglicht die Steuerung mehrerer schaltungsinterner oder -externer Scanpfade.

Systeme mit JTAG-Scanpfaden werden im Rahmen dieser Arbeit für die Tests und das Debugging auf der Basis FPGA-synthetisierter Modelle genutzt. Hiermit lässt sich das Gesamtsystem in einer Echtzeitumgebung testen und kann sich für die Produktion als ASIC qualifizieren. Der Test wird mit Hilfe eines am IMS entwickelten Emulators durchgeführt, der den TAP steuert und über diesen auf den internen Zustand der Schaltung zugreift.

Der Einsatz einer JTAG-Schnittstelle in einem ASIC ist jedoch nicht in jedem Fall vorteilhaft. Insbesondere bei den sicherheitskritischen Schaltungen mit kryptographischem Coprozessor erscheint es nicht ratsam, diese mit internen Scanpfaden auszustatten, da hiermit alle in einem Scanpfad liegenden Flipflops beliebig beschrieben und gelesen werden können.

Zudem ist der Flächenbedarf der Scan-Flipflops nicht unerheblich. Dieser skaliert mit der Anzahl der Flipflops, sodass sich eine Erhöhung des Flächenbedarfs je nach Version beim CPCIPH um ca. $8 - 10\%$ und beim CPMAC um ca. $5 - 7\%$ ergibt. Zudem muss die zusätzliche Fläche des TAP-Controllers berücksichtigt werden.

8.2 Modellsynthese

Die vorliegende Arbeit fokussiert die ASIC-Implementierung der Coprozessoren CPCIPH und CPMAC. Dennoch sind alle implementierten Designs auch für FPGAs synthetisierbar. Auf einem Emulationssystem mit FPGAs von XILINX[82] wurden die Gesamtsysteme in einer Echtzeitumgebung getestet. Damit ist sichergestellt, dass die Systeme auch außerhalb der Simulationsumgebung funktionsfähig sind.

Es sei jedoch darauf hingewiesen, dass sich die Flächenoptimierung für FPGAs und für ASICs unterscheidet: Die FPGA-Synthese bildet Logikfunktionen häufig auf LUTs ab, während eine ASIC-Synthese die beschriebene Logik mit Standardzellen implementiert. So liegt die bei der FPGA-Synthese geschätzte Gatterzahl deutlich über der mit einer ASIC-Synthese tatsächlich erzielten.

[82] Es handelt sich hierbei um die FPGAs XCV600E und XC2V2000 von XILINX.

8.2.1 Beispielsystem

Um eine reale Testumgebung zu schaffen, werden der CPCIPH und der CPMAC in eine Beispielumgebung eingebettet. Dabei wird das in Unterpunkt 4.1 (Abb. 11) dargestellte System verwendet. Die Core Domain enthält dementsprechend

- den Mikrocontroller IMS3311C mit Coprozessorinterface inklusive eines 2 kB großen Mikrocode-ROMs und
- die Coprozessoren CPCIPH oder CPMAC mit unterschiedlichen Interfaces.

Für den CPCIPH-Coprozessor findet die Version mit festverdrahteter Logik für die Byte-Substitutions-Komponente Verwendung. Falls nicht anders angegeben, basieren die Kurven für den CPMAC auf der flächenminimalen Version ohne Barrelshifter. Die Memory Domain setzt sich aus einem ROM und einem RAM zusammen:

- einem 4 kB großen Programmspeicher für die Testprogramme und
- einem 512 B großen Arbeitsspeicher.

Desweiteren verfügt das Beispielsystem über diverse Peripheriebausteine:

- ein Timer/Watchdog-Modul,
- die seriellen Schnittstellen SCI (Serial Communication Interface) und SPI (Synchronous Peripheral Interface) und
- die parallele Schnittstelle PIO (Parallel Input/Output) mit vier 8 Bit breiten Ports.

Schließlich beinhaltet das System einen Testcontroller zur Bereitstellung des JTAG-TAPs und einen Clockcontroller zur Erzeugung unterschiedlicher Systemtakte.[83]

[83] Detaillierte Informationen zu den Peripherieeinheiten enthält [Ims03].

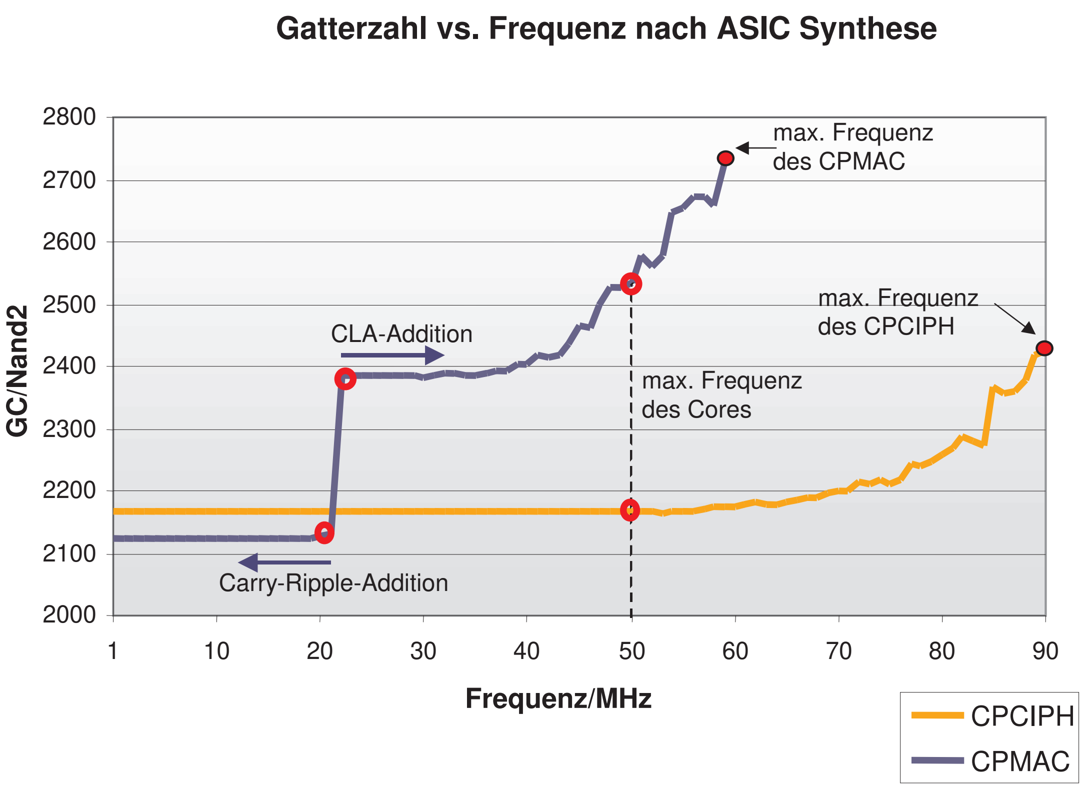


Abbildung 49: Entwicklung der Gatterzahlen bei steigenden Frequenzanforderungen an das CPCIPH- und das CPMAC-Design.

8.2.2 ASIC-Synthese

Die ASIC-Synthese wandelt die Register-Transfer-Level- (RTL) Beschreibung in eine Gate-Level-Beschreibung[84], sodass schließlich eine Gatternetzliste des beschriebenen Systems vorliegt.[85]

Abb. 49 gibt die Gatterzahl der Coprozessordesigns für den Fall wieder, dass diese mindestens mit der auf der Abszisse geforderten Frequenz lauffähig sind. Wie nicht anders erwartet, steigt die Gatterzahl mit der geforderten Mindestfrequenz. Beim CPCIPH wurde bereits in der Entwicklungsphase darauf geachtet, dass dieser einen minimalen Flächenbedarf auch bei höheren Maximalfrequenzen fordert. Augenfällig weist er bei 50 MHz, also der maximal möglichen Fre-

[84] vgl. [Pel99], S.13, sowie [Ga83]

[85] Die ASIC-Synthese der Beispielsysteme wurde mit dem Design Analyzer (Version 2000.11) der Firma Synopsys durchgeführt.

quenz des IMS3311C, noch ein Minimum an Gattern auf. Erst danach wächst die Gatterzahl des CPCIPH langsam an. Selbst bei hohen Frequenzen bis zur höchst möglichen von 90 MHz steigt die Kernzellfläche nur um ca. 12% gegenüber der Minimalfläche an.

Besonders interessant ist der Verlauf der Kurve für den CPMAC. Bei den in Abb. 49 präsentierten Ergebnissen wurde dem Synthesetool die Auswahl eines adäquaten Addierers für die letzte Additionsstufe, die Carry-Propagate-Addition, überlassen und damit der manuell implementierte CLA-Addierer ersetzt.
Bis zu einer Frequenz von 21 MHz wird ein einfacher Carry-Ripple-Addierer (CRA) verwendet. Charakteristisch ist für diesen ein sehr langer kritischer Pfad, jedoch auch ein äußerst geringer Gatterbedarf. Ab einer Frequenz von 22 MHz muss der langsame CRA durch einen schnelleren, aber auch deutlich größeren Addierer ersetzt werden, der eine dem CLA-Adder ähnliche Struktur besitzt. Ein weiterer Anstieg der Gatterzahl setzt erst wieder bei ca. 40 MHz ein.
Die Maximalfrequenz des CPMAC liegt mit ca. 60 MHz nur knapp über der des Controllerkerns. Dies hat vor allem den Grund, dass alle Operationen des CPMAC in nur zwei Taktzyklen durchführbar sein sollen und damit aufwendige Operationen nicht auf ähnlich viele Takte verteilt werden können wie beim CPCIPH.
Desweiteren hängt die Kernzellfläche des CPMAC durch die austauschbaren Addiererstrukturen deutlich stärker von der geforderten Frequenz ab als die des CPCIPH. Die Kurve für die erweiterte Version des CPMAC mit Barrelshifter, Branch-Befehlen und zusätzlichem Akku-Lade-Befehl ähnelt der in Abb. 49. Sie hat jedoch aufgrund der größeren EXEC und Decoder-Einheit einen um etwa 400 GÄ nach oben verschobenen Verlauf.

Abb. 50 untersucht die Gatterzahlen der einzelnen Coprozessoreinheiten, jeweils unterteilt in kombinatorische und nicht-kombinatorische Logik. Den drei linken Balken lässt sich entnehmen, dass die Gatterzahl sowohl des CPCIPH als auch die des CPMAC geringer ist als die des 3311C ohne Mikrocode-ROM. Damit ist eins der primären Designziele erreicht: Die Coprozessoren besitzen eine geringere Kernzellfläche als der Controllerkern.

Die übrigen Balken des Diagramms vergleichen die Coprozessorblöcke des CPCIPH und CPMAC. Es zeigt sich, dass die Execute-Einheit des CPMAC mehr Gatter benötigt als die des CPCIPH. Dies liegt insbesondere an seiner aufwendi-

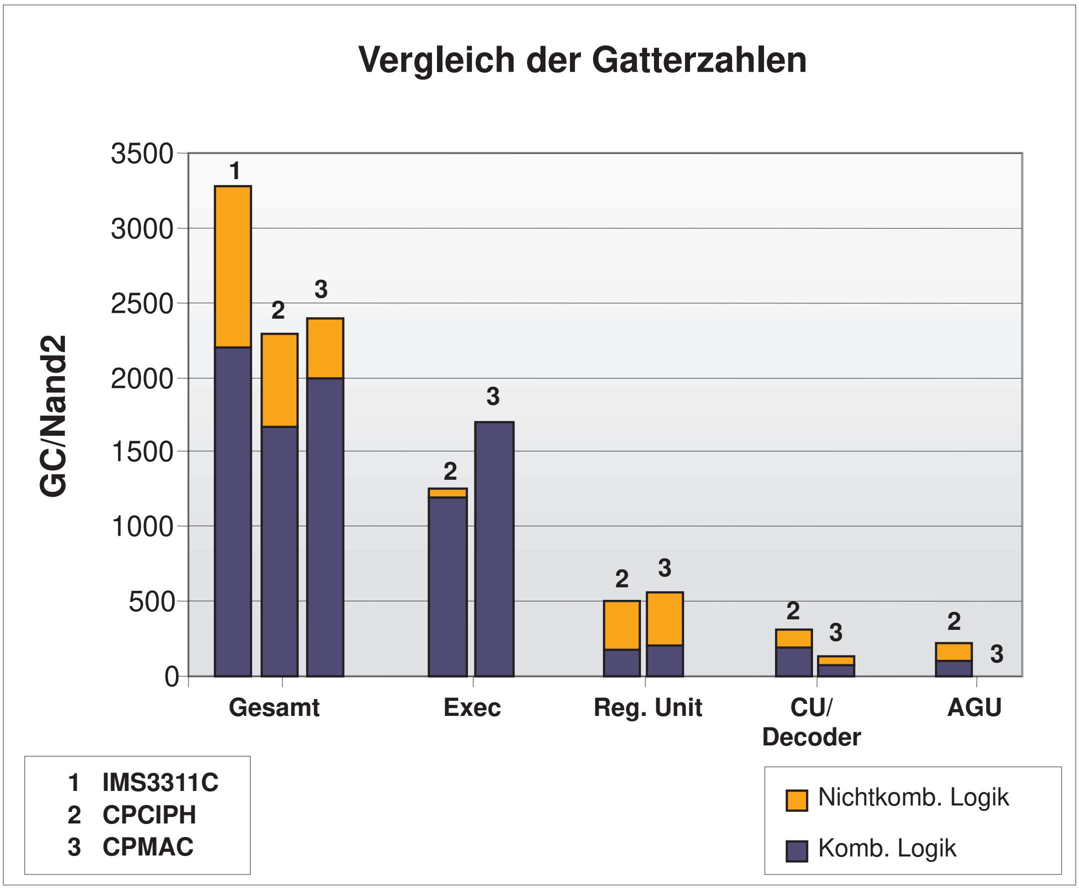


Abbildung 50: Vergleich der Gatterzahlen der Coprozessorblöcke nach einer Synthese für eine Frequenzanforderung von 30MHz.

gen Multipliziererstruktur, die deutlich mehr Fläche nutzt als die logiklastigste Operation des CPCIPH, die multiplikative Inversion im $\mathcal{GF}(2^8)$.

Die Control Unit des CPCIPH ist im Vergleich zum CPMAC recht groß. Sie wurde in der Darstellung in Abb. 50 mit dem Decoder zusammengefasst. Insbesondere die Zustandssteuerung des CPCIPH erfordert einen hohen Gatteraufwand, da diese bei einigen Instruktionen (z.B. Encrypt oder Key Expansion) die Abläufe für mehr als 600 Taktzyklen steuern muss.

Da der CPMAC nicht auf den Prozessorspeicher zugreift, entfällt hier die Address Generation Unit. Diese ist beim CPCIPH für die Adressberechnung zum Lesen und Schreiben des expandierten Schlüssels, der Eingangs- und Ausgangsdaten sowie der Temporärdaten im Scratchpad erforderlich.

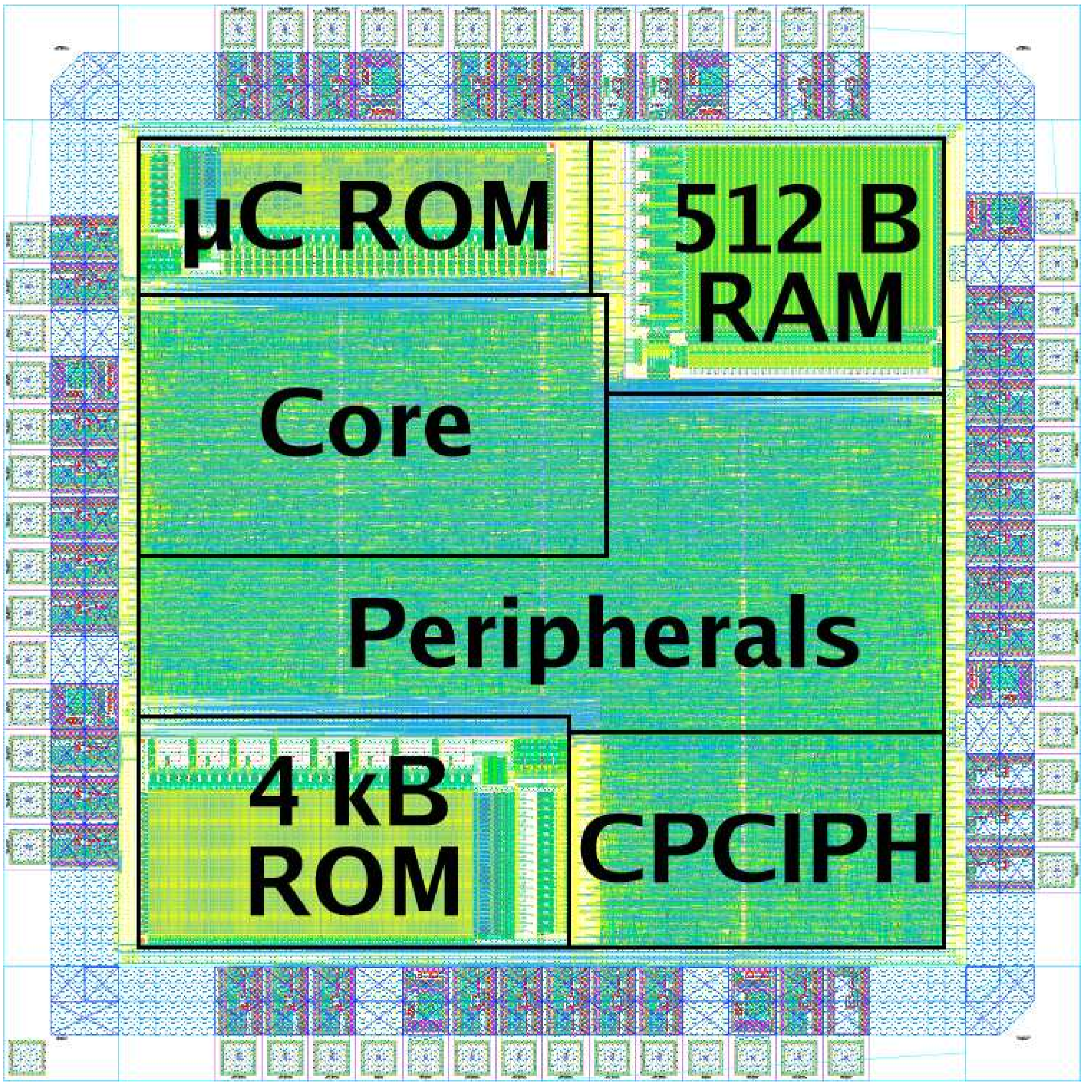


Abbildung 51: Layout eines IMS3311C Systems mit CPCIPH Coprozessor und Peripherie.

Das im Rahmen dieser Arbeit erstellte Layout eines Beispielsystem mit dem CPCIPH Coprozessor ist in Abb. 51 dargestellt. Zusätzlich zu den in 8.2.1 erläuterten Speicherbausteinen sind qualitativ die Flächenanteile des Controllerkerns, der Peripherie und des Coprozessors eingetragen.[86]

Die Anzahl der Taktzyklen des CPCIPH für eine Verschlüsselung oder Ent-

[86] Die jeweils eingerahmten Regionen des Digitalteils geben nicht die tatsächlichen Gebiete der Einheiten wieder, sondern veranschaulichen lediglich den Flächenanteil der jeweiligen Einheit an der Gesamtlogikfläche.

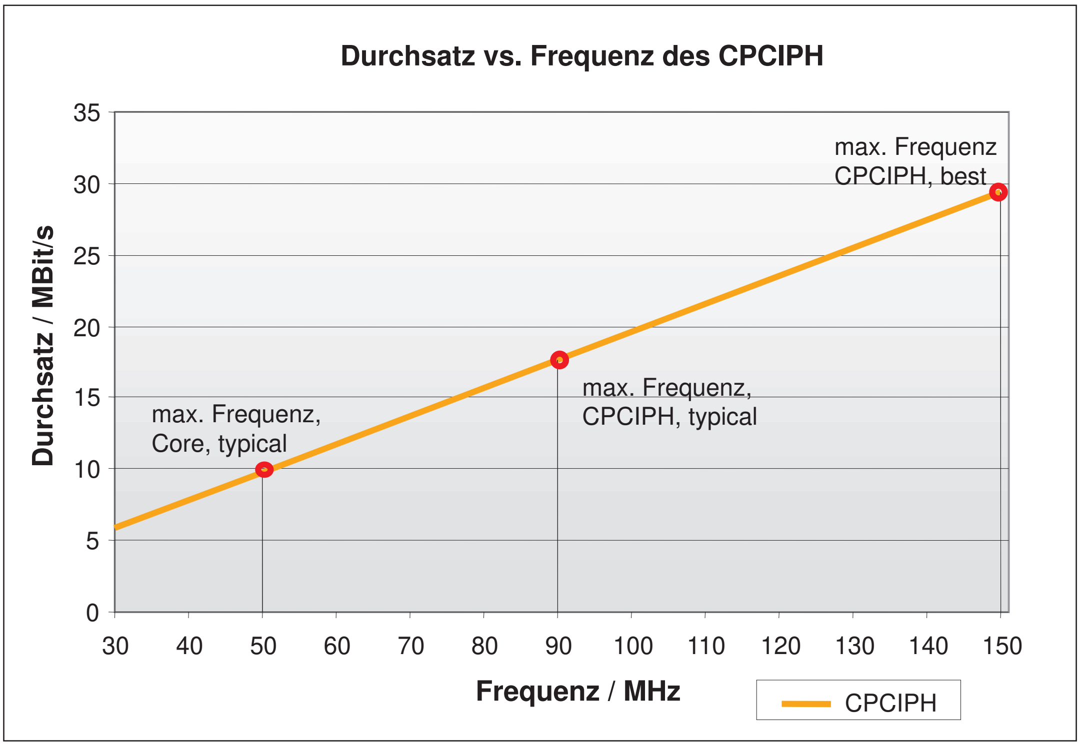


Abbildung 52: Datendurchsatz des CPCIPH abhängig von der Coprozessor-Frequenz.

schlüsselung liegen je nach Version zwischen 606 und 686, wobei innerhalb dieser Zeit ein 128-Bit-Datenblock bearbeitet wird. Den abhängig von der Frequenz ermittelten Datendurchsatz gibt Abb. 52 wieder.[87]
In dem Graphen sind Datenraten für signifikante Frequenzen markiert.[88] Unter optimalen Bedingungen kann der CPCIPH bei der hier gewählten Technologie eine Datenrate von ca. 30 MBit/s bei einer Frequenz von 150 MHz erreichen. Läuft der Controller mit 30 MHz, so erreicht man eine Datenrate von ca. 6 MBit/s.

[87]Der Graph wird für den Mittelwert, also für 646 Taktzyklen, angegeben.

[88]Die Angaben ”typical” oder ”best” spezifizieren die gewählten Prozessparameter.

8.3 Implementierungsergebnisse

Der kommende Unterpunkt vergleicht den CPCIPH mit unterschiedlichen Lösungen: Zuerst wird er relevanten Crypto-Cores[89] gegenübergestellt, anschließend werden die Ergebnisse des CPCIPH mit denen einer Assemblerlösung auf einem Standard-Mikrocontroller in Relation gebracht.

8.3.1 Vergleich mit Hardwarelösungen

In der Literatur werden zahlreiche auf Durchsatz optimierte ASIC- oder FPGA-Implementierungen behandelt, die wenigsten fokussieren jedoch die Gatterminimierung. Als eines der Hauptziele dieser Arbeit wurde jedoch gefordert, dass der Coprozessor eine geringere Fläche aufweist als der 3311C-Controller.

Auch wenn die Tests des 3311C-Systems mit CPCIPH-Coprozessor auf einer FPGA durchgeführt wurden, so fand die Optimierung doch für einen ASIC statt. Daher wird an dieser Stelle lediglich mit anderen ASIC-Implementierungen verglichen, obwohl FPGA-optimierte Realisierungen in der Literatur wesentlich häufiger zu finden sind, u. a. in [Bac03], [Chi02], [Cho03], [Elb00], [Fi01], [Sta03] und [Wea02].

Der Vergleich geschieht anhand vier flächenoptimierter Coprozessoren:

1. der CS5265TK und der CS5275TK von AMPHION [Amp04],
2. der AES-Core von CAST [Cast04],
3. der Standard Encryption Core von Helion [Hel01] und
4. der AES-Coprozessor Randaes von Trichina et al. [Tri01].

Im Gegensatz zum CPCIPH laden die AMPHION-Cores unmittelbar vor einer Ver- oder Entschlüsselung einen 128-Bit-Datenblock. Danach wird der Chiffrier- oder Dechiffriervorgang automatisch ausgelöst. Je nach Version benötigen die Cores für eine Verschlüsselung 11 (CS5275) bzw. 44 (CS5265) Taktzyklen. Der

[89] Es sei darauf hingewiesen, dass in der Literatur häufig von Verschlüsselungs-"Cores" gesprochen wird. Die hier vorgestellten Einheiten sind ausschließlich für das Chiffrieren und Dechiffrieren von Daten zuständig und beinhalten keinen Mikrocontroller. Dies weicht ab von der bisherigen Nutzung des Begriffs, in der mit Core der Controllerkern des IMS3311C ohne Peripherie bezeichnet wird.

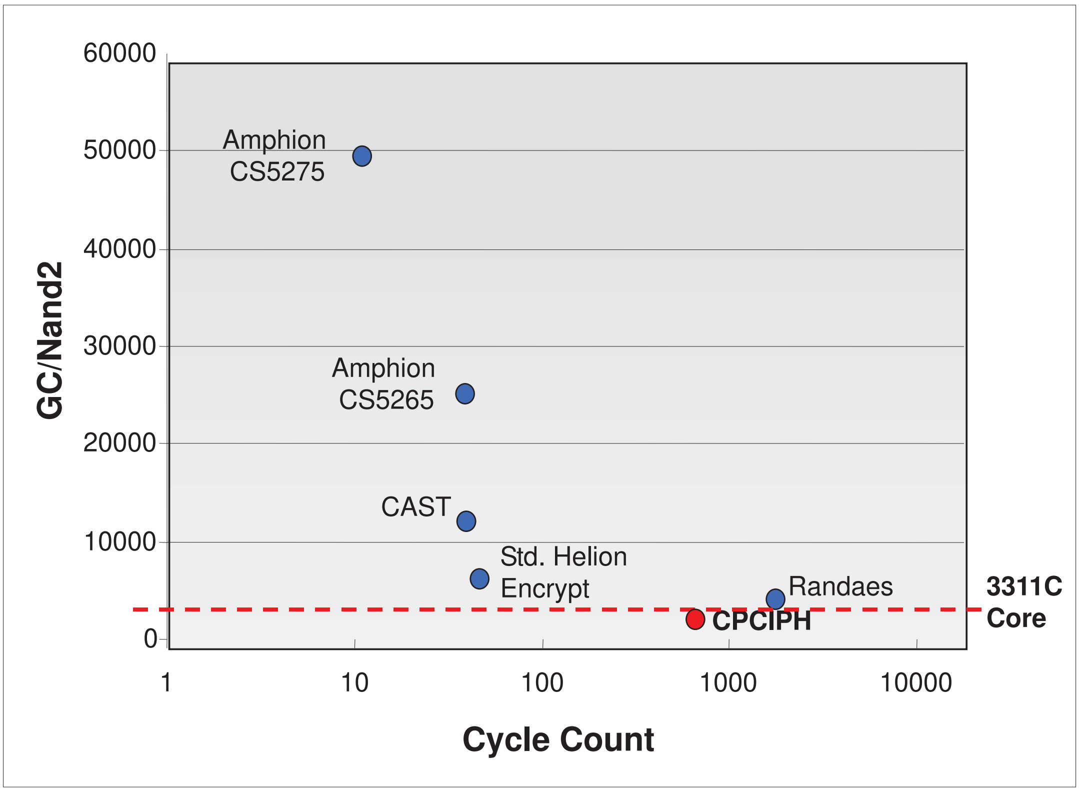


Abbildung 53: Vergleich des Gatterbedarfs aufgetragen über der Zyklenzahl unterschiedlicher ASIC-Realisierungen des AES.

Gewinn an Taktzyklen des CS5275 wird durch eine ca. doppelt so große Fläche im Vergleich zum CS5265 bezahlt (siehe Abb. 53). Die Gatterzahl des kleineren der beiden Cores ist aber immer noch ca. zehn Mal so hoch wie die des CPCIPH. Sie erfüllen damit nicht die Anforderung, weniger Gatter als der 3311C-Controllerkern zu benötigen, da sie weit oberhalb der gestrichelten Linie in Abb. 53 liegen.

Der für eine 0,18 µm Technologie entworfene CAST AES-Core unterscheidet sich hauptsächlich dadurch von den anderen vorgestellten Coprozessoren, dass die Schlüsselexpansionseinheit nicht im AES-Kern integriert ist. Wird die automatische Schlüsselexpansion gewünscht, so kann dies durch Erweiterung des Gesamtsystems um einen zusätzlichen Core (KEXP), der ausschließlich für die Schlüsselexpansion zuständig ist, erreicht werden. Alternativ kann man den bereits expandierten Schlüssel durch ein speziell hierfür vorgesehenes RAM bereitstellen. Der Eintrag in Abb. 53 stellt die gemeinsame Fläche des AES- und des

KEXP-Kerns dar, doch schon der AES-Kern allein besitzt eine größere Fläche als der 3311C.

Bei dem in der Abbildung präsentierten Helion Core handelt es sich um die Standardversion des AES-Verschlüsselungsprozessors. Dieser benötigt ähnlich dem CAST Crypto-Core eine zusätzliche Schlüsselexpansions-Einheit oder ein RAM. Der hier dargebotene Core kann lediglich enkodieren und nicht dekodieren, wobei Helion auch kombinierte Einheiten anbietet. Die Schlüsselexpansion übernimmt er ebenfalls nicht. Allein der Verschlüsselungscore bedarf fast dreimal sovieler Gatter wie der CPCIPH, erzielt jedoch einen Durchsatz von ca. 500 MBit/s bei einer Frequenz von 200 MHz.

Die wohl am meisten auf Flächenoptimierung ausgelegte Coprozessoreinheit der hier zum Vergleich vorgestellten Arbeiten ist der Randaes von Trichina et al. Ähnlich dem CPCIPH verarbeitet der Coprozessor die Eingangsdaten nicht wort- (32 Bit) sondern byteweise und benötigt somit weniger Logikgatter. Im Gegensatz zum CPCIPH lädt der Randaes den gesamten geheimen Schlüssel und den Eingangsdatenblock in lokale Register, was hauptsächlich verantwortlich für seine deutlich größere Gatterzahl ist.[90] Zudem benötigt der Randaes ca. 1800 Taktzyklen für eine einfache Verschlüsselung, was etwa zweimal soviel ist wie beim CPCIPH. Dies ist auf die höhere Parallelisierung der Komponenten des letztgenannten zurückzuführen.

Abb. 53 führt ausschließlich flächenoptimierte AES-Cores auf. Kuo und Verbauwhede stellen in ihrer Arbeit [Kuo01] hingegen eine Implementierung vor, die vorrangig auf Durchsatz optimiert ist und in jedem Taktzyklus eine komplette AES-Runde bearbeitet. Eine solche geschwindigkeitsoptimierte Realisierung benötigt ca. 173.000 Gatter und ist somit ungefähr 80 mal größer als der CPCIPH.[91]

Zusammenfassend ist festzustellen, dass außer dem CPCIPH alle vorgestellten Crypto-Cores oberhalb der gestrichelten Linie in Abb. 53 liegen und deswegen das für diese Arbeit gesteckte Ziel nicht erfüllen.

[90] Leider machen Trichina et al. keine Angaben zur Gatterzahl, sondern lediglich zur Gesamtfläche und zur verwendeten Technologie (0,18 µm). Aus diesem Grunde kann die tatsächliche Gatterzahl lediglich abgeschätzt werden. Angenommen wird der übliche Wert von 45.000 Gattern/mm^2 für eine 0,18 µm Standard-Technologie.

[91] Eine weitere durchsatzoptimierte Lösung von Lutz et al. beansprucht ca. 300.000 Transistoren [Lut03].

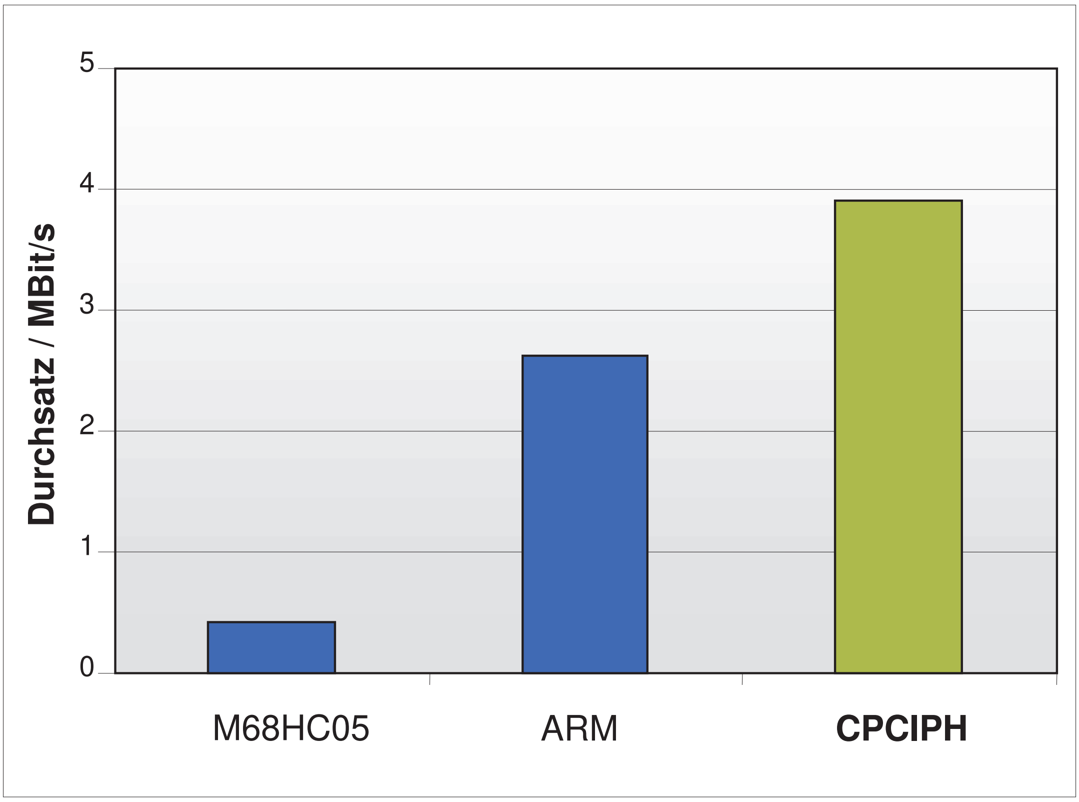


Abbildung 54: Durchsatz des CPCIPH inkl. Schlüsselexpansion im Vergleich zu einer Softwarelösung des ARM und des Motorola 6805 bei einer Frequenz von 30 MHz.

8.3.2 Vergleich mit Softwarelösungen

Nachdem die herausragende Stellung des CPCIPH bezüglich seiner geringen Fläche aufgezeigt wurde, kann nun bewertet werden, inwiefern der CPCIPH für eine Beschleunigung des Cores gegenüber einer Softwarelösung in Form eines Assemblerprogramms sorgt.

Verglichen wird hierfür die Hardwarelösung des CPCIPH mit einer Softwarelösung auf dem Controller M68HC05 von Motorola. Dieser eignet sich besonders gut für einen Vergleich, da er instruktionssatzkompatibel zum IMS2205, dem Vorgänger des 3311C, ist. Ein effektives Assemblerprogramm für den 6805 wird in der Arbeit von Keating [Kea99] beschrieben.

Desweiteren wird überprüft, ob der um die hier erstellte Beschleunigungseinheit erweiterte 8-Bit-Controller 3311C kryptographische Aufgaben mit einem ähnlichen Durchsatz durchführen kann wie ein schneller, gepipelineter 32-Bit-

Controller ohne Coprozessor: Die Chiffriererweiterung für den 3311C wurde eingangs insbesondere gefordert, um auf einen flächenaufwendigen Controller verzichten zu können. Lediglich die zeitkritischen Operationen der Signalverarbeitung erfahren eine gezielte Beschleunigung durch Zusatzhardware.

Hierfür wird der Durchsatz des CPCIPH mit einer Softwarelösung auf dem in Unterpunkt 2.2.2 beschriebenen ARM7 verglichen. Die Informationen über die Zyklenzahl der ARM-Realisierung ist der Arbeit von Hachez et al. [Hac99] entnommen.[92]

Das Diagramm in Abb. 54 stellt den Durchsatz der drei ausgewählten Architekturen für die Verschlüsselung eines Blocks[93] graphisch gegenüber. Für die Lösungen wird die Taktfrequenz auf 30 MHz normiert, da nicht die verwendeten Technologien, sondern die zeitliche Effizienz der Architekturen von Interesse sind.

Die Datenver- und -entschlüsselung mit dem CPCIPH ist etwa um den Faktor 10 bis 15 effektiver als eine Softwarelösung mit dem 6805. Insbesondere bei der in einem Taktzyklus ausgeführten `MixColumns`-Operation spart er viel Zeit gegenüber dem Controller. Die Effizienz eines Assemblerprogramms bei der Dechiffrieroperation ist niedriger, da eine Umsetzung des `InvMixColumns`-Schritts weit aufwendiger ist.

Sogar die Realisierung mit dem 32-Bit-Controller von ARM erreicht einen spürbar geringeren Datendurchsatz als der CPCIPH, obwohl der ARM sämtliche Operationen in seinen 16 Registern durchführen kann und somit zeitaufwendige LD/ST-Befehle vermeidet. Zudem benötigt der ARM eine recht große LUT (1 kB) im ROM, bei geringerer Größe der LUT würde sich der Durchsatz nach Angaben von Hachez et al. in etwa halbieren.

---

[92] Takenaka liefert Ergebnisse für High-End Prozessoren wie den Itanium von Intel [Tke00].

[93] Die Chiffrierung wird inklusive Schlüsselexpansion für einen 128-Bit-Schlüssel durchgeführt.

9 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurden zwei neuartige, flächenoptimierte und effiziente Hardwareumsetzungen für unterschiedliche Signalverarbeitungsalgorithmen realisiert.

Ausgehend von der Fragestellung nach einer optimalen Signalverarbeitungsarchitektur erfolgte die gezielte Eingrenzung des Themengebiets. Ausgewählt wurden der kryptographische Algorithmus AES und die Vektormultiplikationen aufgrund ihrer wachsenden Bedeutung in fast allen technischen Anwendungsbereichen wie der Automobilindustrie, den Unterhaltungsmedien oder der mobilen Datenübertragung. Desweiteren decken gerade diese beiden Algorithmen ein sehr breites Spektrum der Signalverarbeitung ab.
Im Mittelpunkt der Betrachtung stand die Gatterminimierung für eine ASIC-Realisierung unter Berücksichtigung der von den Anwendungen geforderten zeitlichen Minima.

Der 8-Bit-Controller IMS3311C hat sich als besonders geeignete Basisarchitektur herausgestellt. Dieser wurde modular um den kryptographischen Coprozessor CPCIPH und die MAC-Unit CPMAC erweitert, um in signalverarbeitenden Bereichen bezüglich des Durchsatzes mit 16- und 32-Bit Prozessoren konkurrieren zu können. Die beiden Einheiten bilden zwei Anwendungsbeispiele der entworfenen Grundstruktur für signalverarbeitende Coprozessoren.

Desweiteren wurden durch effiziente Möglichkeiten der Schnittstellenrealisierung Prozessor und Coprozessor verknüpft: Unterschieden wurden das schlichte, sequentielle, parallele und hybride instruktionsbasierte Interface sowie eine Memory-Mapped-Schnittstelle.

Durch Partitionierung in logische Komponenten und anschließende zeitliche Parallelisierung der Abläufe konnte bei minimalem Gatteraufwand größtmöglicher Durchsatz erreicht werden.
Unter Zuhilfenahme der mathematischen Analyse der Algorithmen wurde eine ganzheitliche Optimierung der wesentlichen Komponenten ermöglicht. Den signifikantesten Aspekt stellte dabei die Untersuchung einer festverdrahteten Einheit für die Inversion im $\mathcal{GF}(2^8)$ dar, die durch Transformation in den Körper $\mathcal{GF}((2^4)^2)$ mit geringer Gatterzahl implementierbar ist. Insbesondere der hier weiterentwickelte Greedy-Algorithmus ermöglichte das Ersetzen der LUT im ROM durch festverdrahtete Logik bei nahezu gleichbleibender Fläche.

Die beiden realisierten Coprozessoren sind deutlich kleiner als der durch sie erweiterte Mikrocontroller. Insbesondere der CPCIPH nimmt im Vergleich zu anderen Coprozessoren durch seine hohe Integration in den Standardcore und eine Gatterzahl von 2168 GÄ eine herausragende Stellung ein. Er ist ca. 10 bis 15 mal schneller als eine Softwarelösung auf einem M68HC05.

In Fortsetzung dieser Arbeit könnten weitergehende Untersuchungen sowohl hinsichtlich der Leistungsaufnahme des Systems als auch in Bezug auf die Schnittstelle zwischen Prozessorkern und Coprozessor durchgeführt werden.

Werden die Coprozessoren in einem batteriebetriebenen System genutzt, so sollten sie zusätzlich hinsichtlich des Leistungsverbrauchs optimiert werden. Insbesondere die in Galois-Feld-Arithmetik oft auftretenden hintereinandergeschalteten Exklusiv-Oder-Verknüpfungen können zu häufigen Zustandswechseln und hohem Leistungsverbrauch führen [MoS03]. Durch Minimierung der Leistungsaufnahme qualifizieren sich die Systeme für zusätzliche Einsatzgebiete, wie z.B. Chipimplantate in der Medizintechnik. Verlustleistungsoptimierte Umsetzungen bspw. eines AES-Coprozessors werden in der Literatur bisher nur sehr selten behandelt.

Bezüglich der vorgestellten parallelen und hybriden Coprozessorschnittstelle ist eine vertiefende Analyse eines einfachen, gatteroptimierten Priorisierungsschemas interessant, das den Einsatz mehrerer parallel arbeitender Coprozessoren und deren Zugriffspriorität auf die Speicher steuert. Dabei sollte untersucht werden, ob die Priorität mittels dedizierter Instruktionen auf Softwareebene gesetzt werden kann.

Anhang

A Transformationsmatrizen

An dieser Stelle werden alle ermittelten Trafomatrizen aufgeführt, auf deren Basis der Vergleich für die Flächenminimierung in Kapitel 7.3.3 stattfindet. Der im Index der Matrizen erscheinende Wert e gibt den Exponenten des Elements $\beta^e \in \mathcal{GF}((2^4)^2)$ an, das eine Abbildung des Elements $\alpha \in \mathcal{GF}(2^8)$ darstellt.

$$\mathbf{M_{e=5}} = \begin{bmatrix} 1&0&1&0&0&0&0&0 \\ 1&1&0&1&0&0&1&0 \\ 0&0&0&0&1&1&0&0 \\ 1&0&1&0&0&0&1&0 \\ 0&0&0&1&0&1&1&0 \\ 0&1&1&1&0&1&0&0 \\ 0&1&0&0&1&0&0&0 \\ 0&1&1&1&1&0&1&1 \end{bmatrix} ; \quad \mathbf{M_{e=10}} = \begin{bmatrix} 1&0&1&0&0&0&0&0 \\ 1&0&1&0&1&1&0&0 \\ 1&1&0&1&0&0&1&0 \\ 0&1&1&1&0&0&0&0 \\ 0&1&1&0&0&1&1&0 \\ 1&1&1&1&1&1&1&0 \\ 1&1&0&1&1&0&0&0 \\ 1&0&1&0&1&1&0&1 \end{bmatrix}$$

$$\mathbf{M_{e=20}} = \begin{bmatrix} 1&0&1&0&0&0&0&0 \\ 0&1&1&1&0&0&1&0 \\ 1&0&1&0&1&1&0&0 \\ 1&1&0&1&1&1&0&0 \\ 1&0&1&1&1&0&1&0 \\ 0&0&0&1&1&1&1&0 \\ 1&0&0&0&1&1&0&0 \\ 0&0&1&0&0&0&1&1 \end{bmatrix} ; \quad \mathbf{M_{e=40}} = \begin{bmatrix} 1&0&1&0&0&0&0&0 \\ 0&0&0&0&1&1&0&0 \\ 0&1&1&1&0&0&1&0 \\ 1&0&1&0&1&1&1&0 \\ 0&0&0&1&0&1&0&0 \\ 1&0&0&1&0&1&1&0 \\ 0&0&0&1&0&0&1&0 \\ 1&1&1&0&0&0&0&1 \end{bmatrix}$$

$$\mathbf{M_{e=65}} = \begin{bmatrix} 1&0&1&0&0&0&0&0 \\ 0&1&1&1&0&0&1&0 \\ 1&0&1&0&1&1&0&0 \\ 1&1&0&1&1&1&0&0 \\ 0&0&0&1&1&0&1&0 \\ 0&1&1&0&1&1&0&0 \\ 0&0&1&0&0&0&0&0 \\ 1&1&1&1&1&1&1&1 \end{bmatrix} ; \quad \mathbf{M_{e=80}} = \begin{bmatrix} 1&0&1&0&0&0&0&0 \\ 1&1&0&1&0&0&1&0 \\ 0&0&0&0&1&1&0&0 \\ 1&0&1&0&0&0&1&0 \\ 1&0&1&1&0&1&1&0 \\ 1&0&1&0&0&1&1&0 \\ 0&1&0&0&0&1&0&0 \\ 1&1&0&1&1&0&0&1 \end{bmatrix}$$

$$\mathbf{M_{e=130}} = \begin{bmatrix} 1&0&1&0&0&0&0&0 \\ 0&0&0&0&1&1&0&0 \\ 0&1&1&1&0&0&1&0 \\ 1&0&1&0&1&1&1&0 \\ 1&0&1&1&0&1&0&0 \\ 1&0&0&1&1&0&1&0 \\ 0&1&1&0&0&0&0&0 \\ 0&1&0&0&1&1&1&1 \end{bmatrix} ; \quad \mathbf{M_{e=160}} = \begin{bmatrix} 1&0&1&0&0&0&0&0 \\ 1&0&1&0&1&1&0&0 \\ 1&1&0&1&0&0&1&0 \\ 0&1&1&1&0&0&0&0 \\ 1&1&0&0&0&1&1&0 \\ 0&1&0&1&0&0&1&0 \\ 0&0&0&0&1&0&1&0 \\ 1&1&0&1&1&1&0&1 \end{bmatrix}$$

B Berechnung des Kritischen Pfads

Unterpunkt 7.2 stellt den Enhanced-Greedy-Algorithmus vor, der den Hardwareaufwand bei der Berechnung von Matrixmultiplikationen optimiert, indem er die Anzahl der XOR-Gatter minimiert. Als Resultat entsteht eine finale Matrix, bei der mehrfach wiederkehrende Teiloperationen zusammengefasst werden können.

Anstatt ausschließlich die Anzahl der verbleibenden XOR-Operationen zu bewerten, wird ein weiteres Bewertungskriterium herangezogen: die Länge des kritischen Pfads, die entsprechend der Anzahl durchlaufener kombinatorischer Gatter gemessen wird. Im folgenden sei der im Rahmen dieser Arbeit entwickelte Algorithmus erläutert, mit dem der kritische Pfad einer solchen Matrix berechnet und gleichzeitig *eine* mögliche Schaltung mit minimalem kritischen Pfad ermittelt wird.

Jedes Matrixelement erhält bereits bei der Berechnung der finalen Matrix einen Eintrag, der seine Pfadlänge in XOR-Gattern beschreibt. Bei der noch nicht optimierten Matrix sind alle diese Einträge selbstverständlich gleich 0, da noch keine Additionen im $\mathcal{GF}(2^8)$ zusammengefasst wurden. Alle zusammengefassten Elemente haben dementsprechend eine Pfadlänge, die größer als Null ist. Eine einzelne Matrixzeile könnte somit folgendermaßen aussehen:

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	r_0	r_1	r_2	r_3
0	1(0)	0	0	1(0)	0	0	1(0)	0	1(2)	0	1(1)

Dabei repräsentieren die t_x-Elemente Werte aus der ursprünglichen Matrix, r_x vom Greedy-Algorithmus zusammengefasste Elemente. Die Werte in Klammern geben die Pfadlänge der Elemente an.

Die Bestimmung des kritischen Pfads einer solchen Zeile geschieht folgendermaßen:

1. Es wird die Anzahl der '1'-Elemente in der Zeile gezählt. Falls weniger als zwei Einträge vorliegen, folgt (6.).

2. Die Zeile wird durchlaufen und das '1'-Element mit der kleinsten Pfadlänge

als `minPath1` gesichert, bevor der Eintrag aus der Zeile entfernt wird.

3. Erneut wird die Zeile durchlaufen und das '1'-Element mit der kleinsten Pfadlänge als `minPath2` gesichert, bevor auch dieser Eintrag aus der Zeile entfernt wird.

4. Es erfolgt die Erschaffung einer neuen Spalte mit einem neuen '1'-Element. Die Pfadlänge dieses Elements beträgt `newPath = minPath2 + 1`.

5. Es wird mit (1.) fortgefahren.

6. Die Zeile enthält nur noch einen Eintrag. Der Pfad dieses Elements entspricht der Pfadlänge der Schaltung dieser Zeile.

Der kritische Pfad der Schaltung, die durch die gesamte Matrix dargestellt wird, entspricht dem Maximalwert der Pfadlängen aller Zeilen. Anhand des Beispiels der oben dargestellten Zeile sei die Vorgehensweise verdeutlicht:

step	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	r_0	r_1	r_2	r_3	c_0	c_1	c_2	c_3
1)	0	$\boxed{1(0)}$	0	0	$\boxed{1(0)}$	0	0	1(0)	0	1(2)	0	1(1)				
2)	0	0	0	0	0	0	0	$\boxed{1(0)}$	0	1(2)	0	$\boxed{1(1)}$	**1(1)**			
3)	0	0	0	0	0	0	0	0	0	$\boxed{1(2)}$	0	0	$\boxed{1(1)}$	**1(2)**		
4)	0	0	0	0	0	0	0	0	0	0	0	0	0	$\boxed{1(2)}$	$\boxed{\mathbf{1(3)}}$	
5)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1(4)**

Die umrahmten Elemente sind die im jeweiligen Schritt selektierten Werte `minPath1` und `minPath2`, während die bei der Analyse hinzugefügten neuen Spalten mit c_x bezeichnet werden. Nach fünf Berechnungsschritten ergibt sich die Pfadlänge der untersuchten Zeile zu 4 XOR-Gattern (vgl. Abb. 55).

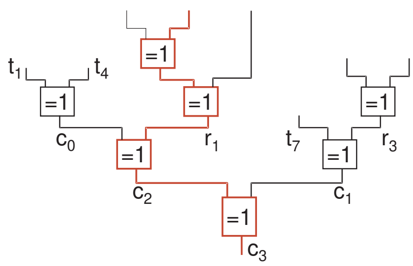


Abbildung 55: Schaltung der Bsp.-Zeile mit Pfadlänge 4.

C Tabellen

Umsetzungen der $\mathcal{GF}(2^8)$ Inversion.

Tabelle 6 bildet die Grundlage für den Vergleich zwischen einer CPCIPH-Implementierung mit LUT im Prozessor-ROM und einer Realisierung mit festverdrahteter Logik. Die Werte in Klammern stellen die geschätzte Gatterzahl einer Umsetzung mit separatem ROM vor.

Kriterium	256 B LUT im Progr.-ROM (in sep. ROM)	festverdr. Logik mit $\mathcal{GF}((2^4)^2)$ Inv.
komb. Logik (GÄ)	1599	1820
Flipflops (GÄ)	761	806
Kernzellfläche (GÄ)	2360	2626
Routingfläche (GÄ)	≈ 944	≈ 1050
Digitalteil gesamt (GÄ)	≈ 3304	≈ 3676
ROM-Fläche (GÄ)	≈ 250 (≈ 1200)	-
Gesamtfläche (GÄ)	≈ 3554 (≈ 4504)	≈ 3676

Tabelle 6: Tabellarischer Vergleich der CPCIPH-Implementierungen mit ROM oder mit festverdrahteter Logik für die Byte-Substitutions-Komponenten.

D Instruktionssatzerweiterungen

Die folgenden Unterpunkte stellen die für die Coprozessoren erstellten Zusatzinstruktionen des IMS3311C vor. Die innerhalb der Tabellen genutzten Adressierungsarten sind inherent (INH), direct (DIR), extended (EXT), indexed (IDX und IDY) sowie relative (REL). Desweiteren können Daten mit dem Stack ausgetauscht werden (PUSH und PULL) [Ims04].

D.1 Instruktionen des CPCIPH

Die folgende Tabelle führt die Instruktionen eines implementierten Beispielmodells des CPCIPH auf. Der Instruktionssatz differiert abhängig von dem gewählten Interface und der Anzahl der Basisregister: Unterschieden werden das Modell mit sequentieller (S) und hybrider (H) Schnittstelle. Die angegebenen Instruktionen setzen voraus, dass ein Modell mit drei BA-Registern vorliegt. Das Modell, welches lediglich über ein gemeinsames BA-Register für Ein- und Ausgangsdatenblock verfügt, lädt dieses mit dem LBI-Befehl und LBO und LBK entfallen.
Dem Modell liegt der *CopShort*-Instruktionstyp zugrunde. Das sämtlichen Befehlen gemeinsame Prebyte `$6B` wird in der Spalte der Opcode-Bytes nicht angeführt. Der hier aufgeführte Befehlssatz ist vorwiegend für Prozessoren mit nur einem Coprozessor geeignet, bei Einsatz mehrerer Coprozessoren sei auf den *CopLong*-Instruktionstyp verwiesen.

Mnemonic	Addr. Mode	Opcode Bytes (hex)	# Mach. Code Bytes	Modell	Bedeutung
ECR	INH	9C	2	S, H	Encrypt (128 Bit)
ECRI	INH	DC	2	S, H	Encr. Incr. (128 Bit)
DCR	INH	1C	2	S, H	Decrypt (128 Bit)
DCRI	INH	5C	2	S, H	Decr. Incr. (128 Bit)
KEXP	INH	3C	2	S, H	Key Expansion (128 Bit)
LBI	DIR EXT IDX IDY IMM PULL	80 81 82 83 88 8B	3 4 3 3 4 2	S, H	Load Base Address for Input: `BAI` $\Leftarrow$ {`(Mem), (Mem+1)`}
LBO	DIR	40	3	S, H	Load Base Address

	EXT	41	4		for Output:
	IDX	42	3		BAO $\Leftarrow$ {(Mem), (Mem+1)}
	IDY	43	3		
	IMM	48	4		
	PULL	4B	2		
LBK	DIR	20	3	S, H	Load Base Address
	EXT	21	4		for Key:
	IDX	22	3		BAK $\Leftarrow$ {(Mem), (Mem+1)}
	IDY	23	3		
	IMM	28	4		
	PULL	2B	2		
SSI	INH	6C	2	H	Set Sequent. IF (SPF=1)
SPI	INH	AC	2	H	Set Parallel IF (SPF=0)
BBY	REL	79	3	H	Branch On Busy
BNB	REL	B9	3	H	Branch On Not Busy

D.2 Instruktionen des CPMAC

Die nachfolgenden Befehle geben alle gültigen Instruktionen des CPMAC mit schlichtem Interface an. Die Auswahl der Opcodes geschah unter dem Aspekt der Gatterminimierung, sodass im Idealfall einzelne Opcode Bits direkt Aufschluss über die Art der durchzuführenden Operation zulassen. Das '?'-Symbol im Basic Opcode repräsentiert die vier Bit breite Coprozessor-ID, die das parallele Anschließen von 16 CPMAC Einheiten ermöglicht.
Die verwendeten Opcodes beruhen auf dem Prinzip der *CopLong*-Instruktion, wobei das Prebyte $87 in der dritten Spalte der Opcode-Tabelle nicht aufgeführt wird. Die Basisversion des CPMAC lässt ausschließlich arithmetische Links- und Rechtsshift-Operationen um eine Position zu.
Nur die erweiterte Version verfügt über einen Barrelshifter sowie Branch- und Akku-Ladebefehle. Die neuen oder veränderten Instruktionen des erweiterten CPMAC sind mit dem '$*$'-Symbol gekennzeichnet.

Mnemonic	Addr. Mode	Opcode Bytes (hex)	# Mach. Code Bytes	Operation/Beschreibung
MAC	INH	50 ?C	3	MA $\Leftarrow$ MA + FA $\cdot$ FB
MULM	INH	40 ?C	3	MA $\Leftarrow$ FA $\cdot$ FB
ADDM	INH	28 ?C	3	MA $\Leftarrow$ MA + {FA, FB}

SUBM	INH	20 ?C	3	MA ⇐ MA − {FA, FB}
ASLM*	INH	6x ?C	3	MA ⇐ MA ≪ <x>
ASRM*	INH	7x ?C	3	MA ⇐ MA ≫ <x>
CLRM	INH	08 ?C	3	MA ⇐ 0
LDFA	DIR	30 ?0	4	FA ⇐ {(Mem), (Mem+1)}
	EXT	30 ?1	5	
	IDX	30 ?2	4	
	IDY	30 ?3	4	
	IMM	30 ?8	5	
	PULL	30 ?B	3	
LDFB	DIR	34 ?0	4	FB ⇐ {(Mem), (Mem+1)}
	EXT	34 ?1	5	
	IDX	34 ?2	4	
	IDY	34 ?3	4	
	IMM	34 ?8	5	
	PULL	34 ?B	3	
LDM*	DIR	3C ?0	4	MA ⇐ {(Mem), (Mem+1), (Mem+2), (Mem+3)}
	EXT	3C ?1	5	
	IDX	3C ?2	4	
	IDY	3C ?3	4	
	PULL	3C ?B	3	
STFA	DIR	10 ?4	4	{(Mem), (Mem+1)} ⇐ FA
	EXT	10 ?5	5	
	IDX	10 ?6	4	
	IDY	10 ?7	4	
	PUSH	10 ?A	3	
STFB	DIR	14 ?4	4	{(Mem), (Mem+1)} ⇐ FB
	EXT	14 ?5	5	
	IDX	14 ?6	4	
	IDY	14 ?7	4	
	PUSH	14 ?A	3	
STM	DIR	1C ?4	4	{(Mem), (Mem+1), (Mem+2), (Mem+3)} ⇐ MA
	EXT	1C ?5	5	
	IDX	1C ?6	4	
	IDY	1C ?7	4	
	PUSH	1C ?A	3	
BNM*	REL	04 ?9	4	Branch on Negative MA
BPM*	REL	05 ?9	4	Branch on Positive MA
BSM*	REL	06 ?9	4	Branch on Saturated MA
BNS*	REL	07 ?9	4	Branch on Not Saturated MA

E Glossar

ADC	Analog Digital Converter
AES	Advanced Encryption Standard
AGU	Address Generation Unit
ALU	Arithmetic Logical Unit
ASIC	Application Specific Integrated Circuit
BA	Basisadresse
CAN	Controller Area Network
CISC	Complex Instruction Set Computer
CLA-Adder	Carry-Look-Ahead-Adder
CRA	Carry-Ripple-Adder
CSA	Carry-Save-Adder
CopShort	2-Byte-Coprozessorinstruktion
CopLong	3-Byte-Coprozessorinstruktion
CPCIPH	Coprocessor for AES Ciphering
CPMAC	Coprocessor for Multiply-Accumulate Operations
CU	Control Unit (Steuerwerk)
DES	Data Encryption Standard
DMA	Direct Memory Access
DSP	Digital Signal Processor
EXEC	Execute Unit (ausführende Recheneinheit)
FA, FB	Faktorregister des CPMAC
FIFO	First-In First-Out
FIPS	Federal Information Processing Standards
FIR	Finite Impulse Response (Filter)
FPGA	Field Programmable Gate Array
FSM	Finite State Machine (endlicher Automat)
GÄ	Gatteräquivalente basierend auf einem NAND2-Gatter
$\mathcal{GF}$	Galois-Feld
GP	General Purpose
IIR	Infinite Impulse Response (Filter)
JTAG	Joint Test Action Group
LD/ST	Load/Store

LSB, LSN	Least Significant Byte/Nibble
LUT	Lookup-Table
MA	Ergebnis-Akkumulator des CpMac
MAC	Multiply-Accumulate
MSB, MSN	Most Significant Byte/Nibble
MUX	Multiplexer
NIST	National Institute of Standards and Technology
PIO	Parallel Input/Output
PP	Partialprodukt
$\mathcal{RC}$	Rundenkonstante des AES-Algorithmus
RA, RB, RC, RD	General-Purpose-Register des CpCiph
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
$\mathcal{S}$-Box	Substitutionstabelle des AES-Algorithmus
$\mathcal{S}_{inv}$-Box	Inverse Substitutionstabelle des AES-Algorithmus
SC	Secure Core
SCI	Serial Communication Interface
SPI	Synchronous Peripheral Interface
TAP	Test Access Port (JTAG)
VLIW	Very Long Instruction Word

F Wissenschaftlicher Werdegang

Mark Jung wurde am 11. Oktober 1975 in Bergisch Gladbach geboren. Nach dem Abitur am Gymnasium Herkenrath, Bergisch Gladbach, studierte er von 1995 bis 2001 Elektrotechnik an der RWTH Aachen. Während des Studiums absolvierte er ein Praktikum bei Infineon, Santa Cruz (CA), und war als studentische Hilfskraft am Lehrstuhl für Allgemeine Elektrotechnik und Datenverarbeitungssysteme (EECS), RWTH Aachen, tätig.

Den Diplomabschluss erreichte er 2001 mit einer Arbeit über intelligente Testfallerzeugung für eine abstrakte Hardware-Beschreibungssprache am Lehrstuhl für Integrierte Systeme der Signalverarbeitung (ISS), RWTH Aachen.
Seit 2002 ist er beim Fraunhofer Institut für Mikroelektronische Schaltungen und Systeme (IMS) in Duisburg angestellt. Hier beschäftigt er sich vorrangig mit der Entwicklung effizienter Digitalschaltungen und FPGA-basierter Entwicklungssysteme. Im Rahmen seiner Anstellung als wissenschaftlicher Mitarbeiter am IMS fertigte er die vorliegende Dissertation an.

Abbildungsverzeichnis

Literatur

[Amp04] AMPHION: *CS5265/75 AES Simplex Encryption/Decryption Cores*, Datenblatt, URL: http://www.amphion.com, 2004

[Arm01] ADVANCED RISC MACHINES: *ARM7TDMI, Technical Reference Manual*, Spezifikation, ARM DDI 0210B, URL: http://www.arm.com, 2001

[Arm02] ADVANCED RISC MACHINES: *Advanced RISC Machines Standard Products, Microprocessors, Microcontrollers and ASSPs*, Application Notes, ARM, URL: http://www.arm.com, 2002

[Arm03] ADVANCED RISC MACHINES: *ARM7TDMI, Instruction Set*, Spezifikation, ARM QRC 0001H, URL: http://www.arm.com, 2003

[Bac03] E. BACKHAUS ET. AL.: *AES in FPGAs, Implementierung des Advanced Encryption Standards*, Elektronik, 08/2003

[Boo51] A.D. BOOTH: *A Signed Binary Multiplication*, Quarterly Journal of Mechanics and Applied Mathematics, 1951

[Bro00] I.N. BRONSTEIN, K.A. SEMENDJAJEW, G. MUSIOL: *Taschenbuch der Mathematik*, ISBN 3-81-712005-2, Harri-Deutsch-Verlag, 2000

[Card03] CARD TECHNOLOGY: *SIM rebound Continues*, Card Technology, August 2003

[Cast04] CAST: *AES - Advanced Encryption Standard Core*, Datenblatt, URL: http://www.cast-inc.com, 2004

[Che79] C.T. CHEN: *One-Dimensional Digital Signal Processing*, ISBN 0-8247-6877-9, Marcel Dekker, 1979

[Chi02] C. CHITU ET AL.: *A Hardware Implementation in FPGA of the Rijndael Algorithm*, 45th Midwest Symposium on Circuits and Systems, 2002, S.507-510

[Cho03] P. CHODOWIEC, K. GAJ: *Very Compact FPGA Implementation of the AES Algorithm*, Conference for Cryptographic Hardware and Embedded Systems (CHES), Springer Verlag, 2003, S.319-333

[Dae99] J.DAEMEN, V.RIJMEN: *The Rijndael Block Cipher*,
AES Proposal for Federal Information Processing Standards Publication, 1999

[Dae00] J. DAEMEN, V. RIJMEN: *Resistance against Implementation Aspects: A Comparative Study of the AES Proposals*,
Proceedings of the 2nd AES Candidate Conference, 1999, S.122-132

[Dae02] J. DAEMEN, V. RIJMEN: *The design of Rijndael*,
ISBN 3-540-42580-2, Springer Verlag, 2002

[Dhe01] J.F. DHEM, N. FEYT: *Hardware and Software Symbiosis Helps Smart Card Evolution*,
IEEE Micro, November 2001

[Dob00] G. DOBLINGER: *Signalprozessoren*,
ISBN 3-935340-01-X, Schlembach, 2000

[Elb00] A.J. ELBIRT ET AL.: *An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists*,
The 3rd Advance Encryption Standard Candidate Conference, 2000, S.13-27

[Ert01] W. ERTEL: *Angewandte Kryptographie*,
ISBN 3-446-21549-2, Carl Hanser Verlag, 2001

[Fär87] G. FÄRBER: *Bussysteme*,
ISBN 3-486-20120-4, R. Ouldenbourg Verlag, 1987

[Fips77] FEDERAL INFORMATION PROCESSING STANDARD (FIPS): *Data Encryption Standard (DES)*,
Publication 46, National Bureau of Standards, U.S. Department of Commerce, 1977

[Fips01] FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION (FIPS): *Announcing the Advanced Encryption Standard (AES)*,
Publication 197, National Bureau of Standards, U.S. Department of Commerce, 2001

[Fi01] V. FISCHER, M. DRUTAROVSKÝ: *Two Methods of Rijndael Implementation in Reconfigurable Hardware*, Conference for Cryptographic Hardware and Embedded Systems (CHES) 2001, Springer Verlag, 2001, S.77-92

[Ga83] D.D. GAJSKI, R. KUHN: *New VLSI Tools*, IEEE Computer, 1983, S.11-14

[Gol03] J.D. GOLIĆ, C. TYMEN: *Multiplicative Masking and Power Analysis of AES*, Conference for Cryptographic Hardware and Embedded Systems (CHES) 2002, Springer Verlag, 2003, S.198-212

[Hac99] G. HACHEZ, F. KOEUNE, J.-J. QUISQUATER: *cAESar results: Implementation of Four AES Candidates on Two Smart Cards*, Proceedings of the Second AES Candidate Conference, 1999, S.95-108

[Hel01] HELION TECHNOLOGY: *High Performance AES (Rijndael) Cores for ASIC*, Datenblatt, URL: http://www.heliontech.com, 2001

[Hen74] H. HENZE, H.H. HOMUTH: *Einführung in die Codierungstheorie*, ISBN 3-528-03024-0, Vieweg, 1976

[HP03] J.L. HENNESSY, D.A.PATTERSON: *Computer Architecture - A Quantitative Approach*, ISBN 1-55860-724-2, Morgan Kaufmann Publishers, 3. Aufl., 2003

[Ims01a] R. LERCH: *IMS3311, System Integration Manual*, Spezifikation, Fraunhofer Institut IMS, 2001

[Ims01b] R. LERCH: *IMS3311, Instruction Set Manual*, Spezifikation, Fraunhofer Institut IMS, 2001

[Ims03] H. KAPPERT, R. LERCH: *IMS3311 Design Example*, Spezifikation, Fraunhofer Institut IMS, 2003

[Ims04] FRAUNHOFER INSTITUT IMS: *IMS2205/IMS3311 Macro Cross Assembler*, Spezifikation, Fraunhofer Institut IMS, 2004

[Ito88] T. ITOH, S. TSUJII: *A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Bases*,
Information and Computation, 1988, S.171-177

[Jtag90] JTAG (JOINT TEST ACTION GROUP): *IEEE Standard Test Access Port and Boundary-Scan Architecture*,
IEEE Standard 1149.1, 1990

[Kat03] D. KATZ, R.RIVIN: *Meeting Media Processing Demands*,
Electronic Design Europe, Mai 2003

[Kea99] G. KEATING: *Performance Analysis of AES Candidates on the 6805 CPU Core*,
Proceedings of the Second AES Candidate Conference, 1999, S.109-114

[Kuo01] H. KUO, I. VERBAUWHEDE: *Architectural Optimization for a 1.82 Gbit/sec VLSI Implementation of the AES Rijndael Algorithm*,
Conference for Cryptographic Hardware and Embedded Systems (CHES), Springer Verlag, 2001, S.51-64

[Lee01] R.B. LEE, Z. SHI, X. YANG: *Efficient Permutation Instructions for Fast Software Cryptography*,
IEEE Micro, November 2001

[Ler90] R. LERCH ET AL.: *A Flexible Embedded Processor System for Automotive Applications*,
Proceedings of the 22nd International Symposium on Automotive Technology & Automation, Florence, 1990

[Lid83] R. LIDL, H. NIEDERREITER: *Finite Fields*,
Encyclopedia of Mathematics and its Applications 20, Addison-Wesley, 1983

[Luc00] S. LUCKS: *Attacking 7 Rounds of Rijndael under 192 Bit and 256 Bit Keys*,
Proceedings of the 3rd AES Candidate Conference, 2000, S.215-229

[Lut03] A.K. LUTZ ET AL.: *2 Gbit/s Hardware Realizations of Rijndael and Serpent: A Comparative Analysis*,
Conference for Cryptographic Hardware and Embedded Systems (CHES) 2002, Springer Verlag, 2003, S.144-158

[Mas91] E.D.MASTROVITO: *VLSI Architectures for Computation in Galois Fields*,
Dissertationsschrift, Universität Linköping, Schweden, 1991

[McE87] R.J. MCELIECE: *Finite Fields for Computer Scientists and Engineers*,
ISBN 0-89838-191-6, Kluwer Academic Publishers, 1987

[Mips02] MIPS: *MIPS32 4KS Secure Data Core*,
Datenblatt, URL: http://www.mips.com, 2002

[Mor89] M. MORII, M. KASAHARA: *Efficient Construction of Gate Circuit for Computing Multiplicative Inverses over $GF(2^m)$*,
Transactions of the IEICE, Januar 1989, S.37-42

[MoS03] S. MORIOKA, A. SATOH: *An Optimized S-Box Circuit Architecture for Low Power AES Design*,
Conference for Cryptographic Hardware and Embedded Systems (CHES) 2002, Springer Verlag, 2003, S.172-186

[Mot88] MOTOROLA: *Motorola M68HC11, Reference Manual*,
Spezifikation, Motorola, 1988

[Mot97] MOTOROLA: *Motorola M68HC16, Reference Manual*,
Spezifikation, Motorola, 1997

[Mot04] MOTOROLA: *Motorola Coldfire MCF5235 Reference Manual*,
Spezifikation, 2004

[NIST04] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST): *Test Values*,
AES Algorithm (Rijndael) Information, URL: http://csrc.nist.gov

[Pa94] C. PAAR: *Efficient VLSI Architectures for Bit Parallel Computation in Galois Fields*,
Dissertationsschrift, ISBN 3-18-332810-0, VDI-Verlag, 1994

[Pa95] C. PAAR: *Some Remarks on Efficient Inversion in Finite Fields*,
IEEE International Symposium on Information Theory, 1995

[Pa04] C. PAAR: *Embedded Security in Automobilanwendungen*,
Elektronik Automotive, 01/2004

[PaR97] C. PAAR, M. ROSNER: *Comparison of Arithmetic Architectures for Reed-Solomon Decoders in Reconfigurable Hardware*,
IEEE Symposium on Field-Programmable Custom Computing Machines, 1997

[Pel99] G. PELZ, G. ZIMMER: *Methoden und Werkzeuge zum Entwurf von CMOS VLSI Schaltungen*,
Skript zur Vorlesung "Rechnergestützter Entwurf", GM-Universität Duisburg, 1999

[Pir96] P. PIRSCH: *Architekturen der Digitalen Signalverarbeitung*,
ISBN 3-519-06517-0, Teubner, 1996

[PKC00] PUBLIC KEY CRYPTOGRAPHY: *IEEE Standard Specification for Public Key Cryptography*,
IEEE Standard 1363, 2000

[Rab02] J.M. RABAEY ET AL.: *Digital Integrated Circuits*,
ISBN 0-13-090996-3, 2.Aufl., Prentice Hall, 2002

[Rijm] V. RIJMEN: *Efficient Implementation of the Rijndael S-Box*,
URL: http://www.esat.kuleuven.ac.be/

[Ros94] M. ROSE: *Mikroprozessor 68HC11 - Architektur und Applikation*,
ISBN 3-7785-2277-9, 2.Aufl., Hüthig, 1994

[Rud01] A. RUDRA ET AL.: *Efficient Implementation of Rijndael Encryption with Composite Field Arithmetic*,
Conference for Cryptographic Hardware and Embedded Systems (CHES) 2001, Springer Verlag, 2003, S.171-184

[Schi01] W. SCHIFFMANN, R. SCHMITZ: *Technische Informatik*,
ISBN 3-540-60710-2, Springer Verlag Berlin, 2001

[Schm78] V. SCHMIDT: *Digitalschaltungen mit Mikroprozessoren*,
ISBN 3-519-02056-4, B.G. Teubner Stuttgart, 1978

[Schm95] H. SCHMECK: *Analyse von VLSI-Algorithmen*,
ISBN 3-86025-682-3, Spektrum Akademischer Verlag, 1995

[Schn96] B. SCHNEIER: *Applied Cryptography*,
ISBN 0-471-12845-7, 2. Aufl., Wiley, 1996

[Sta03] F.X. STANDAERT: *Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs*, Conference for Cryptographic Hardware and Embedded Systems (CHES) 2003, Springer Verlag, 2003, S.334-350

[Ste03] M. STEFFEN: *Entwurf eines RISC-Prozessorkerns für einfache Transponder und Messwertaufnahmeanwendungen*, Fraunhofer Institut IMS, Diplomarbeit, 2003

[Stl04] G. STELZER: *Der 8-bit Mikrocontroller-Report*, Elektronik 21, 2004

[TI02] TEXAS INSTRUMENTS: *SMJ320C25 Digital Signal Processor*, Datenblatt, URL: http://www.ti.com, 2002

[Tie91] U. TIETZE, C. SCHENK: *Halbleiter-Schaltungstechnik*, ISBN 3-540-19475-4, 9. Aufl., Springer, 1991

[Tka00] TAKAGI ET AL.: *Adders (in VLSI Handbook)*, ISBN 0-8493-8593-8, CRC Press Inc., 2000

[Tke00] M. TAKENAKA ET AL.: *Performance Comparison of 5 AES Candidates with New Performance Evaluation Tool*, Proceedings of the Third AES Candidate Conference, 2000

[Tri01] E. TRICHINA ET AL.: *Supplemental Cryptographic Hardware for Smart Cards*, IEEE Micro, November 2001

[Tri03] E. TRICHINA ET AL.: *Simplified Adaptive Multiplicative Masking for AES*, Conference for Cryptographic Hardware and Embedded Systems (CHES) 2002, Springer Verlag, 2003, S.187-197

[Vol03] A. VOLLMER: *Das Europäische Sicherheitsforum*, Elektronik Industrie, 2003

[Was82] S. WASER, M.J. FLYNN: *Arithmetic for Digital Systems Designers*, ISBN 0-03-060571-7, Holt, Rinehart and Winston, 1982

[Wea02] N. WEAVER, J. WAWRZYNEK: *High Performance, Compact AES Implementations in XILINX FPGAs*, Publikation, 2002

[Wen74] S. WENDT: *Entwurf Komplexer Schaltwerke*, ISBN 3-540-06178-9, Springer Verlag Berlin, 1974

[Won85] D.G. WONG: *Digital Systems Design*, ISBN 0-7131-3539-5, Edward Arnold, 1985

[Zha02] X. ZHANG, K.K. PARHI: *Implementation Approaches for the Advanced Encryption Standard Algorithm*, IEEE Circuits and Systems, 2002