**Universität Stuttgart**
**Institut für Automatisierungs- und Softwaretechnik**
Prof. Dr.-Ing. Dr. h. c. P. Göhner

08.05.2008

# Diplomarbeit

**Thomas Ruschival 2188 DA**

# Entwicklung eines Kryptographieprozessors für eingebettete Feldgeräte

**Betreuer: Prof. Dr.-Ing. Dr. h. c. Peter Göhner**
**Dipl.-Ing. Stephan Pech**
**Dipl.-Ing. Felix Gutbrodt**
**Dipl.-Ing. Mathias Maurmaier, M.Sc.**

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| Adata | **A**ssociated **Data** – geprüfter aber unverschlüsselter Header einer Nachricht |
| AES | **A**dvanced **E**ncryption **S**tandard – von NIST erstellter Standard für symmetrische Blockchiffren, auch als Rijndael bekannt. |
| CBC | **C**ounter-**B**lock-**C**haining – Betriebsmodus für Blockchiffre |
| CBC-MAC | **C**ypher**b**lock-**C**haining-**M**essage **A**uthentication **C**ode |
| CT | **C**ypher **T**ext – Geheimtext einer Nachricht |
| CTR | **C**oun**te**r-Betriebsmodus für Blockchiffre |
| ECB | **E**lectronic **C**ode **B**ook – Betriebsmodus einer Blockchiffre |
| FIPS | **F**ederal **I**nformation **P**rocessing **S**tandard – US-amerikanische Standardisierungsbehörde |
| FPGA | **F**ield **P**rogrammable **G**ate **A**rray – rekonfigurierbare Hardwareschaltung |
| IEEE | **I**nstitute of **E**lectronic and **E**lectric **E**ngineers – Internationales Standardisierungskonsortium |
| IT | **I**nformations**t**echnik |
| IV | **I**nitialisierungs-**V**ektor |
| JTAG | **J**oint **T**est **A**ction **G**roup |
| LAB | **L**ogic **A**rray **B**lock, Basiselement von FPGAs (Altera Terminologie) |
| LC | **L**ogic **C**ell , Basiselement von FPGAs (Xilinx Terminologie) |
| LSB | **L**east **S**ignificant **B**it/**B**yte |
| LUT | **L**ook **U**p **T**able |
| MAC | **M**essage **A**uthentication **C**ode – kryptographische Prüfsumme, die unter Verwendung eines geheimen Schlüssels erstellt wird. |
| MD5 | **M**essage **D**igest Version **5** – Algorithmus zur Berechnung einer Prüfsumme |
| MISO | **M**aster **I**n **S**lave **O**ut |

| | |
|---|---|
| MOSI | **M**aster **O**ut **S**lave **I**n |
| MSB | **M**ost **S**ignificant **B**it/**B**yte |
| NIST | **N**ational **I**nstitute of **S**tandards and **T**echnology– US-Amerikanische Standardisierungsbehörde |
| PT | **P**lain **T**ext – Klartext einer Nachricht |
| RSA | **R**ivest **S**hamir **A**ldeman – Asymmetrischer Verschlüsselungsalgorithmus |
| SPI | **S**erial **P**eripheral **I**nterface – taktsynchrone serielle Schnittstelle |
| VHDL | **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage |

# Begriffsverzeichnis

| | |
|---|---|
| Architecture | Implementierung eines Interface in VHDL. |
| Associated Data | Zusätzliche Informationen einer Nachricht (Header, Absender….) |
| Authentizität | Die Eigenschaft, dass die Nachricht von der angegebenen Quelle stammt. |
| Bit | Binärer Wert, 0 oder 1 |
| Blockchiffre | Chiffre, die auf festen Blockgrößen von Daten arbeitet. |
| Carry | Übertrag bei Addition oder Subtraktion, hier als Bit zu verstehen. |
| Chiffre | Algorithmus zur Ver- und Entschlüsselung von Daten |
| Compiler | Übersetzungsprogramm, das Programme in höheren Programmiersprachen in Maschinencode übersetzt. |
| Exponiertheit | Hervorhebung durch eine besondere räumliche Lage. |
| Initialisierungsvektor | Startwert für Algorithmen |
| Integrität | Die Eigenschaft, dass die Nachricht unverändert ist. |
| Interface | Schnittstelle aus Funktionen, im Falle von Software oder Signalen in Hardware |
| IT-Sicherheit | Oberbegriff für alle Bereiche der Sicherung und Gewährleistung des Betriebs von Rechner- und Kommunikationssystemen gegenüber unbefugtem Zugriff. |
| Master | Moderator einer Kommunikation. Instanz, die die Kommunikation steuert. |
| Nonce | Kryptografischer Wert, der nu rein einziges Mal verwendet wird. |
| Oktett / Byte | String mit 8 Bits |
| Padding | Prozess des Auffüllens von Datenblocks mit 0x00 Bytes auf volle Blockgröße |
| Payload | Inhalt einer Nachricht |
| Slave | Kommunikationsteilnehmer, meist passiv, nimmt nur nach Aufforderung durch den Master an der Kommunikation aktiv teil. |
| Stromchiffre | Chiffre, die mit einem kontinuierlichen Datenstrom arbeitet. |

| | |
|---|---|
| Toolchain | Werkzeugkette, mehrere kleine Werkzeuge, die im häufig in einer festen Reihenfolge hintereinander auf Daten angewandt werden. Beispiel hierfür wäre: Präprozessor-Compiler-Linker |
| VHDL | Hardwarebeschreibungssprache nach IEEE 1164 Standard |

# Zusammenfassung

Mit der zunehmenden räumlichen Verteiltheit von Automatisierungssystemen wächst der Kommunikationsbedarf zwischen den einzelnen Elementen des Systems, den Feldgeräten. Diese tauschen auf Feldebene über eine exponierte und teilweise sogar öffentliche Netzinfrastruktur Nachrichten mit sensitivem Inhalt aus. Wird dieser Inhalt abgehört, bedroht dies die Informationssicherheit der Anlage. Manipulierte Nachrichten können die Funktionsweise und die funktionale Sicherheit des Systems gefährden.

Aus diesen Bedrohungen ergibt sich die Forderung nach der Sicherstellung einer vertraulichen und korrekten Nachrichtenübertragung. In der klassischen Informationstechnik wird dies durch kryptographische Algorithmen erreicht. Jedoch sind die bekannten Algorithmen sehr ressourcen- und rechenintensiv. Feldgeräte der Automatisierungstechnik sind aber auf hohe Wirtschaftlichkeit und geringe Stückkosten optimiert, woraus sich Ressourcenbeschränkugen im Feldgerät ergeben. Hinzu kommt oftmals die Forderung nach Echtzeitverhalten der Automatisierungssoftware. Diese Randbedingungen mit einer Softwareimplementierung der kryptographischen Algorithmen zu erfüllen, ist oft nicht möglich. Deshalb wird in der Arbeit eine Systemarchitektur entwickelt, die ressourcenintensive kryptographische Algorithmen auf einen speziellen Koprozessor auslagert. Dieser Kryptographieprozessor wird in das Feldgerät integriert und von dem Prozessor, der die Automatisierungsaufgaben erfüllt, angesteuert. Zusätzlich wird eine Treiberarchitektur aus Strukturierten Komponenten vorgestellt, die es ermöglicht den Kryptographieprozessor transparent in bestehende Software zu integrieren.

# Abstract

Automation systems are nowadays increasingly distributed, due to this the communication between the components of the system becomes crucial part. The embedded units of the system today still exchange sensitive data over unprotected buses and partially over public networks. Evesdropping the messages becomes a direct menace to the information security of a company, while manipulating data packets threaten the functional safety of a facility.

These threats result in a demand for means that guarantee confidentiality and integrity of the purported data. In classical computer science this is achieved by well proven cryptographic algorithms. However these algorithms all have a high computational complexity and are very resource intensive. Embedded Systems in automation technology are optimized for economic use and low unit costs, this leads to scarce resources in the single units. Additionally real-time behaviour of the software is often mandatory for a given situation. With use of software implemented cryptographic algorithms these conditions are hard to fulfill.

This work presents an architecture with a cryptographic coprocessor that implements the computationally demanding algorithms as dedicated hardware. This coprocessor will be integrated into the embedded systems and is controlled by a microcontroller that runs the software for the automation tasks. A driver architecture for the algorithms on the coprocessor is presented that allows the transparent integration of the cryptographic processor into existing systems.

# 1 Einleitung

Ein Großteil heutiger Automatisierungsanlagen besteht aus mehreren verteilten Geräten. Auf den höheren Ebenen der Automatisierungspyramide, wie der Prozess- und Leitebene, werden oft Standard-IT-Geräte eingesetzt. Auf der Feldebene, nahe am technischen Prozess, befinden sich hauptsächlich Mikrokontrollersysteme im Einsatz. Diese sogenannten Feldgeräte besitzen wenig Wissen über das Gesamtsystem und sind auf die effiziente Erfüllung einer Teilaufgabe des Automatisierungssystems spezialisiert. Die Feldgeräte kommunizieren über ein Netzwerk aus Feldbussen oder anderen Medien, wie Ethernet oder WLAN, miteinander und mit höheren Hierarchieebenen des Automatisierungssystems.

Diese Kommunikation ist somit ein kritischer Teil für die Funktionsfähigkeit des Gesamtsystems. Durch die Umgebungsbedingungen in Produktionsanlagen kann es leicht zu elektromagnetischen Störungen auf den Leitungen kommen und Nachrichten werden verfälscht. Es muss gewährleistet werden, dass die Integrität der Nachrichten nicht verletzt wird, da sonst das korrekte Verhalten des Systems nicht mehr garantiert werden kann.

Nicht nur zufällige Störungen sondern auch gezielte Fälschungen von Nachrichten sind in Produktionsanlagen ein Problem. Die Kommunikation ist ein möglicher Angriffspunkt für Sabotage. Eine Gefährdung für die Informationssicherheit des Unternehmens ist die Industriespionage in Produktionsanlagen. Für Konkurrenten kann es attraktiv sein, sich durch Abhören des Netzwerkverkehrs im Automatisierungssystem, unerlaubterweise Informationen über den Produktionsablauf und das technische Know-how des Unternehmens zu verschaffen. Im Gegensatz zu höheren Hierarchieebenen des Automatisierungssystems, wie der Leitwarte, die physisch vor unbefugtem Zutritt geschützt ist, kann auf der Feldebene kaum verhindert werden, dass sich ein Angreifer physischen Zugriff auf das Kommunikationsmedium verschafft.

Auf den höheren Hierarchieebenen werden bekannte Schutzmechanismen der Standard-IT verwendet. Das Security-Framework des IAS zeigt, dass es prinzipiell möglich ist, auch auf der Feldebene mit Mikrocontrollern kryptographische Algorithmen zu implementieren. Deren Berechnung ist jedoch sehr aufwändig und ressourcenintensiv. Die dafür benötigte Rechenzeit steht im Konflikt mit den Automatisierungsaufgaben des Feldgerätes und ist in Echtzeitanwendungen oft nicht vertretbar.

In dieser Arbeit wird eine Architektur vorgestellt, die rechenintensive kryptographische Algorithmen auf einen speziellen Koprozessor auslagert, um die Verarbeitung mit Hilfe optimierter Hardwareschaltungen zu beschleunigen.

# 2 Kryptographische Grundlagen

## 2.1 Definition

Kryptologie ist die Wissenschaft, deren Aufgabe die Entwicklung von Methoden zur Verschlüsselung, die Chiffrierung, von Informationen, die Kryptographie, und deren mathematische Absicherung gegen unberechtigte Entschlüsselung ist**.** Als interdisziplinärer Zweig der Informatik weist die Kryptologie enge Beziehungen zur Mathematik, vor allem zur Komplexitäts- und zur Zahlentheorie, auf.

Unter dem Aspekt der IT-Sicherheit hat das Interesse an der Kryptologie erheblich zugenommen. Grundprobleme bei der Entwicklung von Verschlüsselungsverfahren sind dabei: Vertraulichkeit (Schutz der Daten vor unberechtigter Kenntnisnahme), Authentifizierung und Identifikation (Schutz der Daten vor unerkannter Veränderung, eindeutige Zuordnung ihrer Urheberschaft). [Mey07]

In der Kryptographie wird von der zu verschlüsselnden Information als Klartext, oft auch „Plaintext“ *PT*, gesprochen. Der Geheimtext oder „Cyphertext“ *CT*, ist das Resultat der Verschlüsselungsoperation.

Die Authentizität einer Nachricht bezeichnet die Korrektheit des Absenders. Unter der Integrität einer Nachricht wird verstanden, dass die Daten nicht geändert wurden.

## 2.2 Kategorien kryptographischer Algorithmen

Kryptographische Algorithmen lassen sich nach verschiedenen Kriterien in Kategorien einteilen. Nach der Funktionsweise lassen sich Blockchiffren und Stromchiffren unterscheiden. Ein, für die Anwendung, sehr wichtiges Unterscheidungsmerkmal ist die Einteilung in symmetrische und asymmetrische Algorithmen.

Hashalgorithmen sind ein anderes Feld der Kryptographie, hierbei steht nicht die Geheimhaltung von Nachricht im Vordergrund sondern die Überprüfung der Integrität der Nachricht.

### 2.2.1 Blockchiffren

Blockchiffren sind blockbasierte Verschlüsselungsalgorithmen, englisch „block Cypher“. Die Nachricht wird in Blöcke gleicher Länge unterteilt, die dann einzeln durch den Algorithmus mit dem Schlüssel verarbeitet werden. Hat eine Nachricht eine Länge, die sich nicht ganzzahlig durch die Bockgröße dividieren lässt, so müssen Fülldaten angehängt werden. Weit verbreitete Blockchiffren sind DES, IDEA und AES aus Kapitel 3.

### 2.2.2 Stromchiffren

Stromchiffren sind strombasierten Verschlüsselungsalgorithmen, englisch „stream Cypher“. Dabei wird jedes Symbol oder Bit mit einem Schlüsselsymbol verschlüsselt. Meist ist dies eine XOR (exklusiv-oder), Verknüpfung. Die „One-Time-Pad“-Verschlüsselung ist die bekannteste

Stromchiffre. Hier können Nachrichten beliebiger Länge im Gegensatz zu Blockchiffren aus Kapitel 2.2.1 sofort verschlüsselt werden, ohne dass eine bestimmt Menge von Daten vorhanden sein muss oder Fülldaten angehängt werden müssen. Es ist für die Sicherheit unerlässlich, dass der Schlüsselstrom gleich lang ist, wie die zu verschlüsselnde Nachricht und nicht wiederverwendet wird, da sonst durch Kryptoanalyse Muster und schließlich der Klartext erkannt werden kann.

Zur Erzeugung der Schlüssel eignen sich rückgekoppelte Schieberegister mit besonders langer Periode. Der eigentliche geheime Schlüssel wird als Initialisierunsgsvektor (IV), geladen. Ein Problem tritt auf, wenn ein Angreifer einmal Zugriff auf Klartext und Geheimtext hat. Daraus lässt sich der Schlüssel errechnen, da die Schieberegisterstruktur meist bekannt ist, kann der Schlüsselstrom rekonstruiert und alle Nachrichten entschlüsselt werden. Mit dieser Methode kann z. B. im WLAN die WEP-Verschlüsselung gebrochen werden [ETRWAP2005]. Bekannte Stromchiffren sind RC4 oder der Bluetooth Standard E0.

### 2.2.3 Symmetrische Chiffren

Bei dieser Art der Verschlüsselung wird nur ein geheimer Schlüssel benutzt. Der Sender verschlüsselt die Nachricht mit demselben Schlüssel, den der Empfänger der Nachricht zum Entschlüsseln benutzt. Dieses Verfahren hat den entscheidenden Nachteil, dass Sender und Empfänger vorher über einen sicheren Kanal die Schlüssel austauschen müssen. Soll in einem System nicht nur ein einziger Schlüssel benutzt werden, so muss jeder Sender aller Schlüssel seiner potenziellen Empfänger kennen oder jeder Empfänger den Schlüssel jedes Senders. Bei symmetrischen Verfahren wird das Schlüsselmanagement, die Verteilung der geheimen Schlüssel, sehr schnell zu einem logistischen Problem. Die Komplexität des Problems ist $O(n^2)$. Ist die Sicherheit eines Schlüssels kompromittiert, so muss dieser Schlüssel im gesamten System ausgetauscht werden.

Der Vorteil der symmetrischen Verfahren ist jedoch die wesentlich einfachere Art der Algorithmen und die hohe Verarbeitungsgeschwindigkeit von Nachrichten. Der Durchsatz von Daten ist um Größenordnungen höher als bei asymmetrischen Verfahren.

Wird davon ausgegangen, dass die Sicherheit des Schlüssels nicht kompromittiert ist, d. h. kein Dritter verfügt über den gemeinsamen geheimen Schlüssel, so ist auch die Authentizität der Nachricht garantiert. Es kann davon ausgegangen werden, dass die Nachricht vom angegebenen Empfänger versendet wurde. Jedoch ist ohne Prüfsumme, vgl. Hashalgorithmen in Kapitel 2.2.5, die Integrität Nachricht noch nicht gewährleistet. Bekannte und in der Informatik weitverbreitete symmetrische Algorithmen sind DES, IDEA oder AES, siehe Kapitel 3.

### 2.2.4 Asymmetrische Chiffren

Bei asymmetrischer Verschlüsselung wird ein zusammenpassendes Paar von Schlüsseln benutzt. Jedes Mitglied einer Kommunikationsinfrastruktur besitzt ein solches Schlüsselpaar. Ein Schlüssel wird im System verteilt, alle potenziellen Sender kennen diesen Schlüssel, er ist nicht geheim und darf von allen gelesen werden, daher wird dieser Schlüssel als öffentlicher Schlüssel oder „public Key“ (PK) bezeichnet. Mit diesem Schlüssel können Nachrichten nur *verschlüsselt*

werden. Eine Entschlüsselung damit ist nicht möglich, d. h. nicht einmal der Sender kann die, von ihm verschlüsselte, Nachricht mehr entschlüsseln.

Lediglich der Empfänger verfügt über den dazugehörigen Schlüssel zum Dechiffrieren der Nachricht. Dieser Schlüssel ist geheim und darf nicht veröffentlicht werden. Er wird daher als „private Key“ oder „secret Key“ (SK) bezeichnet. Die Verfahren sind daher auch als Public-Key-Verfahren bekannt.

Asymmetrische Verfahren sind im Gegensatz zu symmetrischen Verfahren relativ neu, erst 1977 wurde von Ronald L. Rivest, Adi Shamir und Leonard Aldeman das in Kapitel 5 beschriebene RSA Verfahren entwickelt.

Da jeder Sender Zugriff auf die öffentlichen Schlüssel hat, ist es ohne Zusatzfunktionalität nicht möglich die Authentizität des Absenders zu gewährleisten. Ferner ist nicht gewährleistet, dass der Nachrichteninhalt unverändert ist. Asymmetrische Verfahren bieten jedoch die Zusatzfunktion, dass mit dem privaten Schlüssel eine Prüfsumme der Nachricht verschlüsselt werden kann, die mit dem öffentlichen Schlüssel entschlüsselt werden kann. Hierbei spricht man vom *Signieren* einer Nachricht. Zur Prüfung der Signatur verwendet der Empfänger den öffentlichen Schlüssel des Senders und entschlüsselt die Signatur der Nachricht. Er erhält eine Prüfsumme, diese Prüfsumme wird gegen die berechnete Prüfsumme der empfangenen Nachricht geprüft. Sind die Prüfsummen identisch, so ist gewährleistet, dass die Nachricht und der Absender unverändert und echt sind.

Der entscheidende Vorteil asymmetrischer Verfahren ist der Wegfall eines komplizierten Austauschverfahrens für Schlüssel, da öffentliche Schlüssel, ohne die Geheimhaltung zu gefährden, verteilt werden können. Es muss jedoch gewährleistet werden, dass der verwendete öffentliche Schlüssel auch tatsächlich der öffentliche Schlüssel des Adressaten ist. Sonst ist es einem Angreifer möglich, dem Absender einen falschen Public Key zu übergeben, der damit Nachrichten für den Empfänger verschlüsselt, diese Nachrichten können dann abgefangen und mit dem privaten Schlüssel des Angreifers entschlüsselt werden. Der eigentliche Empfänger kann sie jedoch nicht entschlüsseln.

Dieses Problem wird durch den Einsatz eines sogenannten „Web of Trust“ gelöst. Dabei werden die öffentlichen Schlüssel alle von einer zentralen vertrauenswürdigen Institution signiert, die die Identität der Schlüsselinhaber überprüft. Jeder Sender kann mit dem bekannten öffentlichen Schlüssel der Zertifizierungsstelle überprüfen ob die Signatur des jeweiligen öffentlichen Schlüssels des Empfängers echt ist und somit die Identität des Empfängers geprüft wurde. Dieses Prinzip wird z. B. bei SSL-Zertifikaten von Websites eingesetzt.

Der Datendurchsatz ist bei asymmetrischen Verfahren viel geringer als bei symmetrischen. Ein anderer nachteiliger Aspekt ist, dass wenn eine Nachricht an mehrere Empfänger versandt werden soll, die Nachricht für jeden Empfänger mit dessen public-key individuell verschlüsselt werden muss. Beide Nachteile werden in der Praxis dadurch umgangen, dass die Nachricht selbst mit einem Schlüssel symmetrisch verschlüsselt wird. Dieser symmetrische Schlüssel, der viel kürzer ist als die Nachricht selbst, wird dann mit den öffentlichen Schlüsseln individuell verschlüsselt und and die Nachricht angehängt. Der Empfänger wendet zuerst seinen privaten Schlüssel an, um den symmetrischen Schlüssel der Nachricht zu erhalten und dann mit einem

symmetrischen Algorithmus den Inhalt der Nachricht zu entschlüsseln. Asymmetrische Chiffren sind ElGamal und RSA aus Kapitel 5.

### 2.2.5 Hashalgorithmen

Unter Hashalgorithmen versteht man Algorithmen die aus einer Eingabe mit beliebiger Länge eine kurze und vor allem *eindeutige* Prüfsumme, den Hashwert, errechnen. Jede Quellnachricht hat einen *eindeutigen* Hashwert der reproduziert werden kann. Zwei unterschiedliche Quellen dürfen nicht denselben Hashwert besitzen, ist dies trotzdem der Fall spricht man von einer „Kollision“. Hashalgorithmen sind daraufhin entwickelt, solche Kollisionen zu vermeiden.

Das Ziel ist hier nicht die Geheimhaltung des Inhaltes einer Nachricht sondern die Überprüfung der Richtigkeit. Es ist möglich, dass die Nachricht bei der Übertragung absichtlich oder zufällig verändert wurde. Bei Informationen wie z. B. Text, die von Menschen geprüft werden, sind Bitfehler leicht zu erkennen. Werden jedoch nur Zahlen übertragen oder die Information von Rechnern ausgewertet, wird der Programmablauf mit falschen Informationen fortgesetzt.

Praktisch bietet ein, an die Nachricht angehängter, Hashwert ein gutes Mittel, um Übertragungsfehler zu erkennen. Absichtliche Manipulationen eines Angreifers sind aber nur bedingt erkennbar, da zum erzeugen des Hash keine geheimen Schlüssel benötigt werden. Der Angreifer kann also die Nachricht ändern, einen neuen Hashwert berechnen und den alten Hashwert durch den neuen ersetzen. Für den Empfänger ist dies nicht erkennbar. Um sich gegen solche Manipulation abzusichern, muss der Hashwert über einen anderen, sicheren Kanal gesendet werden.

Weitverbreitete Hashalgorithmen sind SHA, MD4 und MD5, der in Kapitel 6 beschrieben ist. Für die letzteren beiden wurden aber bereits Kollisionen gefunden und sie gelten, mathematisch, nicht mehr als sicher. Am 1. März 2005 wurden von drei Wissenschaftlern der Bell Laboratories zwei X.509 Zertifikate mit gleichen MD5 Hashes erstellt [LWW05].

# 3 Der AES Algorithmus - (Rijndael)

Der AES (**A**dvanced **E**ncryption **S**tandard), auch „Rijndael“ ist eine symmetrische Blockchiffre. Der Algorithmus wurde 1998 von den Belgiern Joan Daemen und Vincent Rijmen veröffentlicht und im Jahr 2000 vom National Institute of Standards and Technology als DES Nachfolger standardisiert [FIPS197]. AES wurde mit besonderem Augenmerk auf Durchsatz und einfache Implementierung in Hard- oder Software entwickelt. Übliche Blockgrößen sind 128, 192 oder 256 Bit. AES gilt bisher als sicher und ist ein offener Standard.

## 3.1 Elemente des AES Algorithmus

Der Algorithmus besteht aus mehreren Funktionen, die in einer Runde, nacheinander mehrmals auf den Datenblock, auch „State“ genannt, angewandt werden. Je nach Blockgröße durchläuft der Block unterschiedlich viele Runden.

| | **Schlüssellänge** ***Nk*** **[32-Bit Worte]** | **Blockgröße** ***Nb*** **[32-Bit Worte]** | **Anzahl der Runden *Nr*** |
|---|---|---|---|
| **AES-128** | 4 | 4 | 10 |
| **AES-192** | 6 | 4 | 12 |
| **AES-256** | 8 | 4 | 14 |

Tabelle 1: AES Runden [FIPS197]

### 3.1.1 Schlüsselexpansion

Aus dem eingegebenen Benutzerschlüssel *K* werden *Nr*+1 Rundenschlüssel erzeugt. Jeder Rundenschlüssel hat die Größe eines AES-Blocks *b*. Der Schlüssel *K* muss also auf eine länge von (*Nr*+1)**b* expandiert werden. Die Spalten des expandierten Schlüsselblocks bilden die Rundenschlüssel. In die erste Spalte des Blocks wird der Benutzerschlüssel K gespeichert. Die Bytes der weiteren Spalten werden rekursiv durch Permutation der vorhergehenden Spalte mittels der SBOX Kapitel 3.1.3 und anschließender XOR-Verknüpfung mit der vorhergehenden Spalte errechnet. Jedes zweite Byte wird zusätzlich mit einer Konstante für die entsprechende Position XOR-verknüpft.

### 3.1.2 Formatierung der Eingangsdaten

Die Eingangsnachricht wird wie bei anderen Blockchiffren in Blocks fester Länge unterteilt. In einer Spalte des Blocks befinden sich immer 32 Bits, bzw. vier Bytes. Die Anzahl der Spalten variiert, je nach Blockgröße enthält ein AES-Block vier, sechs oder acht Spalten.

### 3.1.3 SBOX

Die SBOX ist ein zweidimensionales Array der Größe 16x16 mit dem jedes Byte des Blocks monoalphabetisch substituiert wird. Der Inhalt ist sind die Elemente des endlichen Galoisfeld GF($2^8$).

Da das Galoisfeld konstant ist, kann die SBOX als Look-up-Table gespeichert werden. Die vollständige SBOX ist in [FIPS197] Abschnitt 5.11 zu finden.

Die oberen vier Bits des Bytes im Block geben die Reihe der SBOX an, die unteren vier Bits die Spalte. Das Byte im Block wird dann durch den Inhalt des ermittelten Feldes der SBOX ersetzt. Zur Entschlüsslung wird eine dazu inverse SBOX benutzt.

### 3.1.4 Shift-Row

Beim Shift-Row-Schritt einer AES-Runde werden die Zeilen des Blocks gegeneinander zyklisch nach links verschoben. Bei der Entschlüsslung wird in entgegen gesetzter Richtung rotiert. Das Byte, das aus der ersten Spalte herausgeschoben wird, wird in die vierte Spalte eingesetzt, siehe Abbildung 1. Um wie viele Bytes die jeweilige Zeile rotiert wird, hängt wiederum von der Blockgröße ab.

| **Blockgröße** | **128** | **192** | **256** |
|---|---|---|---|
| **Zeile 1** | 0 | 0 | 0 |
| **Zeile 2** | 1 | 1 | 1 |
| **Zeile 3** | 2 | 2 | 3 |
| **Zeile 4** | 3 | 3 | 4 |

Tabelle 2: Verschiebung bei AES Shift Row

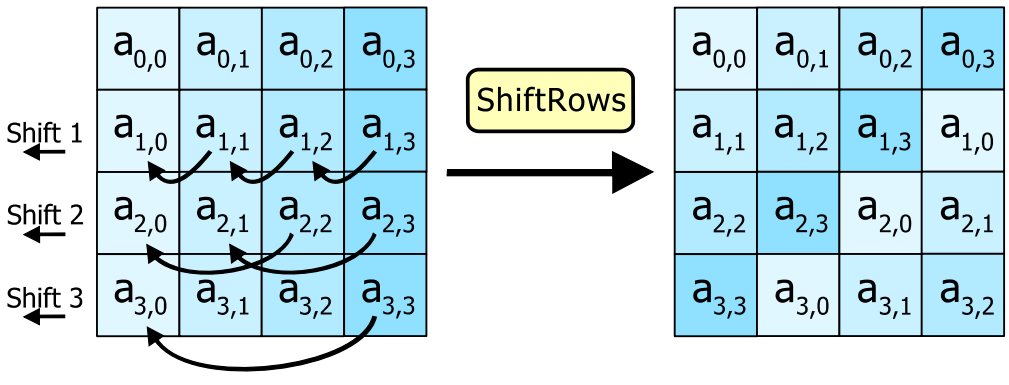


Abbildung 1: AES Shift-Row [Wiki08]

### 3.1.5 Mix-Columns

In diesem Bearbeitungsschritt werden die Spalten miteinander vermischt, nach links geschoben und XOR-verknüpft. Dies entspricht der Multiplikation der Spalte im Galoisfeld GF($2^8$) mit

einer Zeile der Matrix Abbildung 2. Eine Multiplikation mit 2 ist hier als Linksschiebeoperation, eine Addition als XOR definiert. Die Mathematischen Hintergründe dazu sind in [FIPS197] Abschnitt 5.1.3 zu finden. In Abbildung 4 ist der Mix-Column graphisch dargestellt. Zur Entschlüsslung wird die inverse Matrix Abbildung 3 auf der Multiplikation auf dem Galoisfeld GF($2^8$) eingesetzt.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

Abbildung 2: Mix-Column C

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix}$$

Abbildung 3: Inverse Mix-Column C’

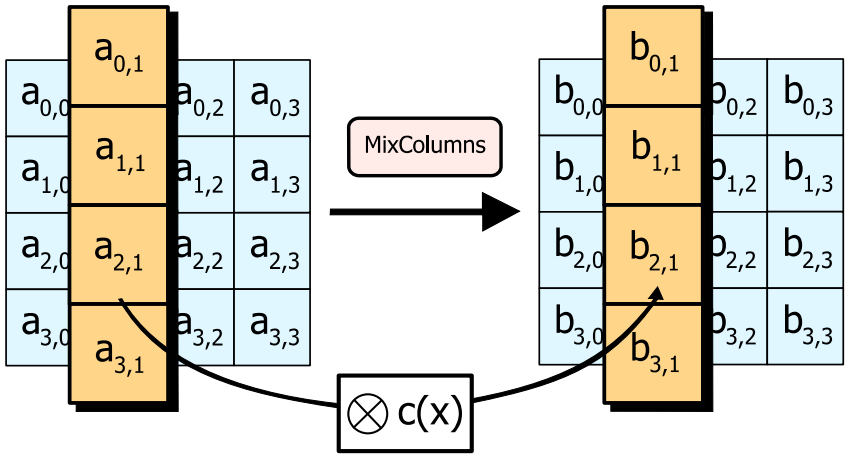


Abbildung 4: AES Mix Columns [Wiki08]

### 3.1.6 Key-Addition

In diesem Schritt werden der State mit dem jeweiligen Rundenschlüssel XOR-verknüpft. Die Verschlüsselung beginnt mit dem ersten Rundenschlüssel, dem Benutzerschlüssel, die Entschlüsslung beim letzten Rundenschlüssel.

## 3.2 Ablauf einer Verschlüsselung

Die Verschlüsselung eines Klartext-Blocks läuft nach dem Algorithmus ab, der in Abbildung 5 als Kontrollfluss dargestellt ist. Jeder Block durchläuft *Nr* mal die Schritte 3 bis 6:

1. Zuerst werden mit der die Rundenschlüssel nach der Schlüsselexpansion aus Abschnitt 3.1.1 erzeugt.
2. Auf den formatierten Klartext wird eine Key-Addition nach Kapitel 3.1.6 mit dem Benutzerschlüssel angewandt.
3. Das Ergebnis aus 2. wird mit Hilfe SBOX wie in Kapitel 3.1.3 verschlüsselt.

4. Die Zeilen des Ergebnisblocks aus 3. werden nach dem Shift-Row-Verfahren nach Abschnhitt 3.1.4 rotiert.
5. Die Spalten des Ergebnisblocks aus Punkt 4. werden durch Mix-Column wie in Kapitel 3.1.5 miteinander vertauscht Dieser Schritt wird in der letzten Runde übersprungen.
6. Der Block wird wieder über die Key-Addition Rundenschlüssel XOR-verknüpft.

Die Entschlüsselung läuft dazu analog in umgekehrter Reihenfolge ab und es werden die inversen Funktionen der jeweiligen Elemente benutzt. Die XOR-Verknüpfung des Addroundkey Elementes ist zu sich selbst invers, allerdings muss beachtet werden, dass die Rundenschlüssel in umgekehrter Reihenfolge angewandt werden.

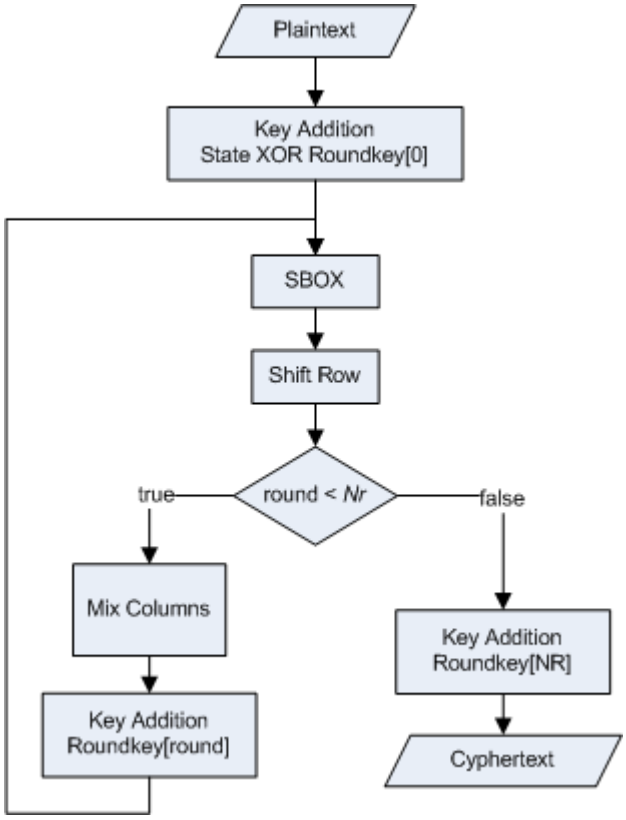


Abbildung 5: Ablauf der AES-Verschlüsselung

# 4 Betriebsmodi von Blockchiffren – „Block-Cypher-Modes“

Für symmetrische Blockchiffren wurden verschiedene Betriebsmodi entwickelt, die auf den zugrunde liegenden Algorithmus in unterschiedlicher Weise zurückgreifen, um entweder die Sicherheit zu erhöhen oder den Algorithmus für zusätzliche Funktionen wie z. B. Authentifizierung zu benutzen.

Derzeit existieren sieben Blockchiffremodi für die, vom [NIST] anerkannten, Verschlüsselungsalgorithmen [NIST-38a]. Fünf Modi dienen der Vertraulichkeit, hiervon ist der Counter-Mode in Kapitel [4.2] für das Projekt von Relevanz. Ein Modus für Authentifizierung, CMAC aus [NIST38b] und ein Modus für Vertraulichkeit und Authentifizierung, den CCM (**C**ounter-Mode with **C**BC-**M**AC) aus Kapitel 4.3 [NIST38c]

Alle spezifizierten Modi, bis auf den CBC (**C**ipher-**B**lock-**C**haining)-Mode, nutzen die Eigenschaft, dass eine XOR-Verknüpfung zu sich selbst invers ist. Es wird bei diesen Betriebsmodi nicht der Klartext, sondern ein Initialisierungsvektor oder daraus abgeleitete Vektoren verschlüsselt, welche dann mit dem Klartext oder Geheimtext XOR-verknüpft werden, um die Nachricht zu ver- oder zu entschlüsseln. Dadurch kommen diese Modi ohne Entschlüsslungsdatenpfad des zugrunde liegenden Algorithmus aus.

## 4.1 ECB – Electronic-Codebook-Mode

Der Electronic-Code-Book Mode einer Blockchiffre ist im engeren Sinne kein spezieller Betriebsmodus. Es ist die Anwendung des Algorithmus auf einen einzigen Block ohne weitere Randbedingungen und Abhängigkeiten von Initialisierungsvektoren oder früheren Aufrufen des Algorithmus. Eine Verschlüsselung eines Blocks im ECB-Mode erzeugt immer denselben Geheimtext. Diese Eigenschaft macht den an sich sicheren Algorithmus bei großen Mengen ähnlicher Daten anfällig für einen Angriff über Mustererkennung. In Abbildung 7ist das Original aus Abbildung 6 im ECB-Mode verschlüsselt dargestellt. Man erkennt deutlich Muster des Originals, da viele Klartext-Blocks einer Länge von 128 Bit identisch waren ergeben sich identische Geheimtext-Blocks. In Abbildung 8 ist dasselbe Original unter Verwendung eines anderen Betriebsmodus verschlüsselt dargestellt.

Abbildung 6: Original

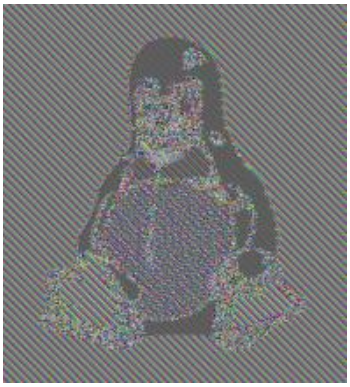

Abbildung 7: ECB-Mode

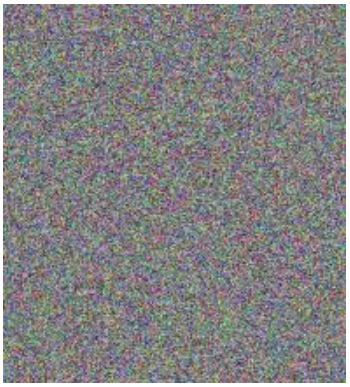

Abbildung 8: anderer Mode

## 4.2 CTR - Counter-Mode

Der Counter Mode ist ein Modus, der die Vertraulichkeit einer Nachricht absichert. Grundvoraussetzung hierfür ist ein eindeutiger sich nicht wiederholender Counter-Block für jeden zu verschlüsselnden Klartext-Block. Hierfür ist ein Initialisierungsvektor für den Zähler notwendig. Die nachfolgenden Counter-Blocks werden nach der Inkrementierungsfunktion aus [NIST38b] erzeugt. Sender und Empfänger brauchen denselben Initialisierungsvektor.

Die einzelnen Counter-Blocks werden mit der zugrunde liegenden Blockchiffre und dem Benutzerschlüssel verschlüsselt. Es wird eine Sequenz von verschlüsselten Counter-Blocks ähnlich dem Prinzip eines „One-Time-Pad“ erzeugt. Die Sequenz von verschlüsselten Counter-Blocks kann auf Vorrat erzeugt und in einem Speicher abgelegt werden, was die eigentliche Ver- und Entschlüsselung einer Nachricht auf eine einfache XOR-Operation reduziert. Dies kann vor bei Anwendungen, die eine kurze Latenz benötigen von Vorteil sein.

Die verschlüsselten Counter-Blocks werden zum Verschlüsseln mit dem Klartext-Block XOR-verknüpft siehe Abbildung 9.

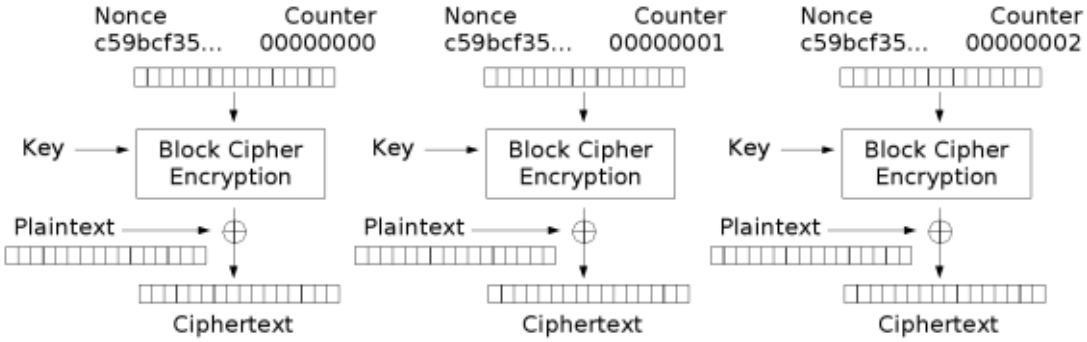


Abbildung 9: Verschlüsselung im CTR-Mode [Wiki08]

Zur Entschlüsselung einer Nachricht wird die XOR-Verknüpfung auf den Geheimtext-Block mit dem dazugehörigen verschlüsselten Counter-Block angewandt, um den Klartext-Block zu erhalten.

## 4.3 CBC-MAC Mode

Bei CBC-MAC (**C**ypher **B**lock **C**haining **M**essage **A**uthentication **C**ode) handelt es sich um einen Betriebsmodus der nicht die Geheimhaltung einer Nachricht gewährleisten soll, sondern die Authentizität und die Integrität sicherstellt. Ziel ist es, ähnlich einer Hashfunktion, vgl. Kapitel 2.2.5, einen eindeutigen Fingerabdruck, den MAC, einer Nachricht zu errechnen und den gegen den empfangenen Hashwert zu prüfen. Anders als bei einer Hashfunktion ist hier aber die Kenntnis des Schlüssels für die Blockchiffre nötig. Der Hashwert gibt somit, über die Integrität der Nachricht hinaus, auch Auskunft über die Authentizität des Absenders.

Eingangsdaten für den CBC-MAC sind die Nachricht, der Benutzerschlüssel und ein Initialisierungsvektor. Dieser Initialisierungsvektor darf bekannt sein und kann auch der Nullvektor sein. Als Nachricht kann der Klartext oder der Geheimtext dienen.

Der Ablauf des CBC-MAC-Modus unterscheidet sich vom, in [NIST38a] beschriebenen, CBC-Mode kaum:

1. Der erste Nachrichtenblock wird mit dem Initialisierungsvektor IV über XOR verknüpft.
2. Das Ergebnis wird mit der Blockchiffre verschlüsselt und mit dem nächsten Nachrichtenblock XOR-verknüpft. Dieser Schritt wird bis zum letzten Nachrichtenblock wiederholt.
3. Ein Teil des letzten Ergebnisblocks dient dann als MAC.

Es entsteht eine Sequenz von Blöcken in der jeder Block von der erfolgreichen Verschlüsselung des Vorgängers abhängig ist. Diese Abhängigkeit garantiert eine unvorhersagbare Veränderung des MAC wenn ein Block verändert wurde.

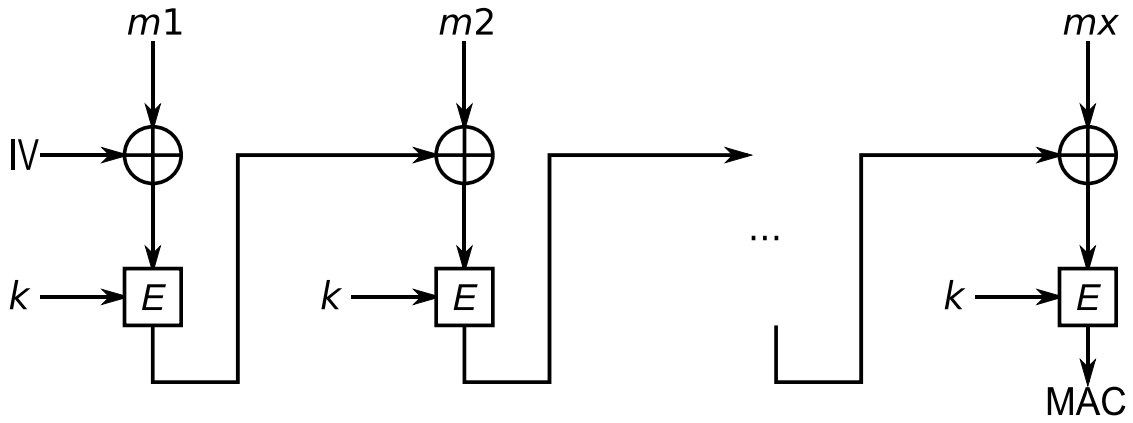


Abbildung 10: CBC-MAC

## 4.4 CCM - Counter-Mode with CBC-MAC

Der CCM-Mode ist eine Kombination aus CTR-Mode, vgl. Kapitel 4.2 zu Verschlüsselung und dem CBC-MAC-Modus 4.3 zur Authentifizierung der Nachricht. Eine bekannte Anwendung dieses Verfahrens ist der IEEE Standard 802.11i zur Verschlüsselung von Paketen in WLAN-Netzen. Dieser Mode ist in [RFC3610] und in [NIST38c] spezifiziert. Durch die Kombination beider Modi braucht der CCM-Mode zwei vollständige Durchläufe der zugrunde liegenden Blockchiffre pro Nachrichtenblock.

# 5 Der RSA Algorithmus

RSA ist eine asymmetrische Blockchiffre, die 1977 als erstes Public-Key-Verfahren von Ron Rivest, Adi Shamir und Leonard Aldeman am MIT in Boston entwickelt wurde und in vielen Anwendungen benutzt wird. Der Algorithmus wurde nach den Anfangsbuchstaben der Nachnamen benannt. Die Sicherheit von RSA beruht auf der Tatsache, dass bisher noch kein effizienter Algorithmus gefunden wurde, um sehr große Zahlen in Ihre Primfaktoren zu zerlegen oder das RSA-Problem zu lösen. Das RSA Problem ist die Suche nach einer Zahl *PT* unter der Kenntnis von *CT*, *M* und *e* für die gilt: $PT^e \equiv CT \bmod M$ .

Prinzipiell lässt sich mit ausreichend Zeit jede RSA-Verschlüsselung entschlüsseln. 2005 Wurde an der Universität Bonn im Rahmen des RSA-Factoring-Challenge innerhalb von zwei Jahren eine 640-Bit Zahl faktorisiert [RSAFC], übliche Schlüsselgrößen liegen mit 1024 oder 2048 Bit aber weit darüber und gelten als sicher.

Neben der Verschlüsselung von Daten bietet RSA außerdem die Möglichkeit, ähnlich dem CCM-Mode aus Kapitel 4.4, eine Nachricht mit dem privaten Schlüssel digital zu signieren. Die Gültigkeit der Signatur kann mit dem öffentlichen Schlüssel überprüft werden.

## 5.1 Erzeugung des Schlüsselpaares

Die Nichtvorhersagbarkeit der Primzahlen aus denen das Schlüsselpaar generiert wird, ist von entscheidender Bedeutung für die Sicherheit. Es sollten hierbei nicht ausschließlich Pseudozufallszahlengeneratoren eingesetzt werden, da deren Zahlensequenz mit Kenntnis der Struktur vorhergesagt werden kann.

Die Schlüssel bestehen aus je einem Zahlenpaar. Das Zahlenpaar des öffentlichen Schlüssels sei (*e*, *N*), das des privaten Schlüssels (*d*, *N*). *N* wird auch „RSA-Modul“ genannt. Die Berechnung der Zahlenpaare ist nicht besonders aufwändig. Dazu sind zunächst zwei voneinander verschiedene zufällige Primzahlen *p* und *q* nötig. Daraus berechnet sich das RSA-Modul als Produkt. Außerdem wird die Eulersche Phi-Funktion $\varphi(N) = (p-1)\cdot(n-q)$ dazu berechnet. Die Zahl *e* des öffentlichen Schlüssels ist eine beliebige zu $\varphi(N)$ teilerfremde natürliche Zahl. Die Zahl *d* des privaten Schlüssels berechnet sich als aus der Vorschrift: $e \cdot d \equiv 1 \bmod \varphi(N)$. Sie ist das multiplikativ inverse Element zu e bezüglich der Modulo $\varphi(N)$-Operation. [BSM+01]

## 5.2 RSA-Operation

Um einen Nachrichtenklartext *PT* zu verschlüsseln, muss diese zunächst in Blöcke $X_0$ bis $X_m$ zerlegt werden. Die Größe eines Blocks bestimmt sich dadurch wie groß die größte, durch den Block darstellbare Zahl ist, diese muss kleiner als *M* sein. Wird zum Beispiel eine Blockgröße von vier Byte gewählt, so muss *M* größer als $255^4$ gewählt sein.

Der Geheimtextblock $CT_i$ wird durch Potenzierung Modulo $M$ berechnet.

$$CT = PT^E \bmod M$$

Formel 1: RSA Verschlüsselung

Das technische Problem der Rechnung besteht darin, dass die Zahlen schnell sehr groß werden und den darstellbaren Bereich des Rechners überschreiten. Daher wird die Rechenvorschrift nach Peter L. Montgomery [Mo95] oder dem Algorithmus „Square & Multiply“ gelöst.

Da $D$ das multiplikativ inverse Element des öffentlichen Schlüssels $E$ ist, kann der Empfänger die ursprünglichen Nachrichtenblocks $PT_0$ bis $PT_m$ aus den Geheimtextblocks $CT_0$ bis $CT_m$ mit der gleichen Modularen Exponentation erhalten.

$$PT_i = CT_i^d \bmod M$$

Formel 2: RSA Entschlüsselung

## 5.3 Effiziente Realisierung des RSA-Algorithmus

### 5.3.1 Modulo-Potenzierung

Die Modulo-Potenzierung des RSA-Algorithmus aus Formel 1 wird durch die „Square-Multiply“-Methode nach [Formel 3] durchgeführt. Dies ist ein übliches Verfahren zur Potenzierung von Ganzzahlen in Hardware. Hierbei wird der Exponent *E,* mit n-Bit, in *n* Iterationen bearbeitet. Die Modulo-Multiplikationen in Zeile 5 und Zeile 6 sind voneinander unabhängig, verwenden aber einen gemeinsamen Faktor $Z_i$, sie können parallel durchgeführt werden [Bl99].

1) $E = \sum_{i=0}^{n-1} e_i \cdot 2^i \quad , \quad e_i \in \{0,1\}$
2) $P_0 := 1$
3) $Z_0 := PT$
4) *for* $i = 0$ *to* $n-1$ *loop*
5) $tmp := CT_i \cdot Z_i \bmod M$
6) $Z_{i+1} := Z_i^2 \bmod M$
7) *if* $e_i = 1$
8) $CT_{i+1} := tmp$
9) *else*
10) $CT_{i+1} := CT_i$
11) *endif*
12) *endloop*

Formel 3: Square-Multiply-Algorithmus

Die Geschwindigkeit der Modulo-Potenzierung ist maßgeblich von der zugrunde liegenden Modulo-Multiplikation abhängig.

### 5.3.2 Algorithmus nach Montgomery

Die Modulo-Multiplikation ist allgemeine und in Hardware im besonders, eine komplexe arithmetische Funktion, da sie eine Multiplikation und eine Ganzzahldivision beinhaltet. Zwei Ansätze zur Durchführung sind denkbar. Die Modulo-Operation wird nach oder während der Multiplikation durchgeführt. Im ersten Ansatz wird ein n-Bit Multiplizierwerk, ein 2-n-Bit Register für das Zwischenergebnis und ein 2-n-Bit Dividierwerk benötigt [DaMa02]. Beim zweiten Ansatz ist der Hardwareaufwand nicht so groß, es werden aber in jedem Schleifendurchlauf der Multiplikation Vergleichs- und Subtraktionsoperationen eingefügt. Immer wenn das Zwischenergebnis größer als der Modulus ist, wird der Modulus davon subtrahiert.

Peter L. Montgomery stellt in [Mo85] einen effizienten Algorithmus zur Berechnung der Modulo-Multiplikation $A \cdot B \bmod M$ ohne Division durch *M* vor. Der Ansatz des Algorithmus von Montgomery ist es, die Zahlen in sogenannte *m*-Residuen zu transformieren, damit die Multiplikation durchzuführen und das Ergebnis zurückzutransformieren. Die m-Residue X´ einer Zahl X, ist $X' = X \cdot R \bmod M$ . Mit den angegebenen Algorithmen lassen sich Zahlen wie folgt in ihre m-Residuen überführen:

$$\begin{aligned} montgomery(X, R^2, M) &= X \cdot R^2 \cdot R^{-1} \bmod M \\ &= X \cdot R \bmod M \\ &= X' \end{aligned}$$

Formel 4: Transformation in m-Residuen

Der Algorithmus nach Montgomery und die vorgestellten Variationen berechnen anstelle des gewünschten Ergebnisses $A \cdot B \bmod M$ nur das Ergebnis: $A \cdot B \cdot R^{-1} \bmod M$ . Da ein zusätzlicher Faktor $R^{-1}$ eingeführt wird, sind Korrekturberechnungen nötig. $R^{-1}$ ist das multiplikativ-inverse Element von *R* bezüglich Modulo M, also $R^{-1}$ R = 1 mod *M*.

Um die Montgomery-Multiplikation durchzuführen, ist eine Zahl *R* mit $R>M$ notwendig, ferner müssen R und M teilerfremd sein, da sonst kein multiplikativ-inverses Element $R^{-1}$ existiert. Die Bedingung der Teilerfremdheit ist in RSA immer erfüllt, da *M* als Produkt zweier Primzahlen immer ungerade ist und wenn $R=2^n$ gewählt wird *R* immer gerade ist. Ein weiterer positiver Aspekt der Wahl von R als Potenz zur Basis 2 ist, dass sowohl die Division als auch die Modulo-Operation trivial werden und als Rechtsschieben bzw. Maskierung der höherwertigen Bits des Zwischenergebnisses realisiert werden können. Daher ist die Berechnung der Residue $S' = montgomery(A, B, M) = A \cdot B \cdot R^{-1} \bmod M = S \cdot R \bmod M$ wesentlich schneller $A \cdot B \bmod M$ , als auf konventionelle Art zu berechnen.

Werden die Operanden vorher nach Formel 4 in ihre m-Residuen überführt, so lässt sich das Ergebnis durch Montgomery-Multiplikation mit 1 auf einfache Weise wieder rücktransformieren.

$$A' = montgomery(A, R^2 \bmod M, M) = A \cdot R \bmod M$$
$$B' = montgomery(B, R^2 \bmod M, M) = B \cdot R \bmod M$$
$$S' = montgomery(A', B', M) = A' \cdot B' \cdot R^{-1} \bmod M = A \cdot R \bmod M \cdot B \cdot R \bmod M \cdot R^{-1} \bmod M$$
$$S' = A \cdot B \cdot R \bmod M$$
$$S = montgomery(S, 1, M) = A \cdot B \cdot R \cdot R^{-1} \bmod M = A \cdot B \bmod M$$

Formel 5: Rücktransformation der m-Residuen

Der Faktor $R^2 = 2^{2n}$ ist allerdings außerhalb des mit n Bit darstellbaren Bereichs, also wird wiederum der Modulo des Faktor berechnet. Der Faktor $R^2$ mod $M$ muss vorher berechnet werden und ist für eine RSA-Operation mit einem Schlüsselpaar immer konstant.

In [DaMa02] werden drei Varianten des Algorithmus von Montgomery für die Realisierung des RSA-Algorithmus in Hardware vorgestellt. Die Eingangsparameter der Algorithmen sind wie folgt definiert:

$$A = \sum_{i=0}^{k-1} a_i \cdot 2^i \quad , \quad B = \sum_{i=0}^{k-1} b_i \cdot 2^i \quad , \quad M = \sum_{i=0}^{k-1} m_{i\cdot} \cdot 2^i \qquad a_i, b_i, m_i \in \{0, 1\}$$

Formel 6: Montomery-Parameter

Es gilt, dass $M$ maximal $k$-Bit Lang ist und $A$, $B < M$, dann ist die Länge des Multiplizierwerks, $n$, $n=k+2$. Es ist notwendig die Länge des Multiplizierwerks k+2 zu definieren, denn dadurch kann das Zwischenergebnis in der nächsten Iteration weiterverwendet werden.

Die drei von [DaMa02] vorgestellten Algorithmen führen alle zu einer leicht unterschiedlichen Hardwarearchitektur. Hier werden lediglich zwei der Algorithmen betrachtet:

$montgomery2(A, B, M)\{$
$\quad S_{-1} = 0$
$\quad for(i = 0\ to\ n-1)\{$
$\quad\quad q_i = (S_{i-1} + b_i A) \bmod 2$
$\quad\quad S_i = \frac{(S_{i-1} + q_i M + b_i A)}{2}$
$\quad \}$
$\quad return\ S_{n-1}$
$\}$

Formel 7: Montgomery2

$montgomery3(A, B, M)\{$
$\quad S_{-1} = 0$
$\quad A = 2 \cdot A$
$\quad for(i = 0\ to\ n)\{$
$\quad\quad q_i = S_{i-1} \bmod 2$
$\quad\quad S_i = \frac{(S_{i-1} + q_i M + b_i A)}{2}$
$\quad \}$
$\quad return\ S_n$
$\}$

Formel 8: Montgomery3

Da in Montgomery3 $A$ um Eins nach links geschoben wird, ist der Beitrag von $A$ zum LSB $q_i$ der Summe $S_{i-1}$ immer 0. Die Modulo Operation für das Letzte Bit hängt also nur noch von der, in der vorherigen Iteration errechneten Summe, $S_{i-1}$ ab. Das Ergebnis wird aber dadurch auch um den Faktor zwei größer was durch eine zusätzliche Iteration korrigiert wird.

Wie gezeigt arbeitet der Algorithmus nach Montgomery die Bits in umgekehrter Reihenfolge ab. Die Operationen nachfolgender Iterationen sind lediglich vom niederwertigsten Bit der vorherigen Iteration abhängig. Da sich dieses Bit aber durch Überträge oder ähnliches nicht ändert, können weiter Iterationen begonnen werden bevor das vollständige Zwischenergebnis des ersten Schleifendurchlaufs vorliegt.

### 5.3.3 Modulo-Potenzierung für RSA mit Montgomery Multiplikation

Um nun die RSA-Operation aus Formel 1 mit Montgomery-Multiplikation durchzuführen, sind zusätzlich Transformation in m-Residuen und die Rücktransformation des Ergebnisses nötig. Daher ergibt sich folgender Algorithmus:

$X^E \bmod M = RSA(X, E, M)\{$
$\quad P = montgomery(1, R^2 \bmod M, M)$
$\quad Z = montgomery(X, R^2 \bmod M, M)$
$\quad for(i = 0\ to\ n-1)\{$
$\quad\quad Z_{i+1} = montgomery(Z_i, Z_i, M)$
$\quad\quad if\ (e_i = 1)$
$\quad\quad\quad P_{i+1} = montgomery(P_i, Z_i, M)$
$\quad\quad else$
$\quad\quad\quad P_{i+1} = P_i$
$\quad\quad endif$
$\quad endloop$
$\quad return\ montgomery(1, P_n, M)$
$\}$

Formel 9: RSA-Montgomery-Algorithmus

### 5.3.4 Multiplikation mit Systolischem Array

Da der Algorithmus nach Montgomery nur auf Addition mit Carry-Bit und Rechtsschieben basiert und nur das letzte Bit des vorherigen Ergebnisses den Ablauf beeinflusst, lässt sich das Ergebnis in Blöcken mit konstanter Breite berechnen. Jeder Block wird von einer Zelle berechnet. Aus diesen Zellen lässt sich ein sogenanntes Systolisches Array aufbauen, wie es von C.D. Walter erstmals in [Wa93] vorgestellt wurde. Da aber ein komplettes Systolisches Array nach Walter n+2 Zellen in n+3 Reihen benötigt und RSA mit n>512 arbeitet, ist dieser Ansatz mit vertretbarem Hardwareaufwand nicht zu realisieren. Bei Walter berechnet jede Zeile des Array eine Iteration der Schleife aus dem Algorithmus in Formel 9. Durch Einführen eines Taktes in den Array kann das Ergebnis sequenziell in 2(n+3) Takten mit nur einer Zeile

berechnet werden. Wenn mehr Zeilen implementiert sind, ist die Latenz zwar immer noch n+3, der Durchsatz erhöht sich aber entsprechend.

# 6 Der MD5 Algorithmus

Der MD5 Algorithmus ist eine weitverbreitete Hashfunktion, um die Integrität von Daten zu überprüfen, er erzeugt aus einer beliebig langen Nachricht einen 128 Bit langen Hashwert, der aus vier 32-Bit Worten besteht. Der Algorithmus wurde als von Ronald Rivest vom MIT 1991 entwickelt und ist in [RFC1321] veröffentlicht. Ziel der Entwicklung war es einen sicheren Nachfolgekandidaten zu dem, damals bereits als unsicher vermuteten, MD4 zu finden.

1996 wurde jedoch bereits die erste Kollision in Teilen des MD5 Algorithmus gefunden. 2004 gelang es Chinesischen Wissenschaftlern Kollisionen in MD5 herbeizuführen [LWW05]. Der somit als nicht mehr sicher geltende MD5-Algorithmus ist dennoch weiter in vielen Bereichen eingesetzt. Der dafür Grund liegt in der weiten Verbreitung des Algorithmus und darin, dass der Angriff sich lediglich auf die Erzeugung zweier neuer, unterschiedlicher Dokumente mit demselben Hashwert beschränkt. Bestehende Dokumente können noch nicht verändert werden, ohne dass sich der Hashwert ändert.

### 6.1.1 Formatierung der Nachricht

Da die Nachricht, *M*, eine beliebige Länge, *L*(*M*) in Bits, aufweisen kann, der Algorithmus aber nur auf Blocks *X* einer festen Länge von 512 Bits arbeitet, muss die Nachricht zuerst formatiert werden. Es sei $X_j$ der j-te 512-Bit-Block der Nachricht *M*, in $X_0$ sind also die ersten 512 Bits der Nachricht abgelegt. Das Format eines Blocks $X_j$ gliedert sich in k=16 Spalten, $X_j[k]$ einem 32-Bit Wort im Little-Endian-Format. Da der Algorithmus nur mit ganzen Blocks arbeitet, muss meist mit Füllbits aufgefüllt werden, hierbei spricht Rivest von „*Padding*“.

Dazu wird ein Bit mit dem Wert „1“ angehängt und danach n „0-Bits“ bis die Länge der Gesamtnachricht ($L(M)+1+n$) mod 512 = 448 ist. Danach wird in den letzten Block die Länge der Nachricht *L*(*M*) in Bits als 64-Bit Little-Endian Integer angehängt. Daraus ergibt sich für den Fall, dass das angehängte 1-Bit auf das 448. Bit des letzten Blocks kommt insgesamt 511 0-Bits angehängt werden.

### 6.1.2 Initialwerte und Konstanten

Vor dem Verarbeiten des ersten Blocks werden die 32-Bit Worte, *A*, *B*, *C* und *D* des Hashwerts wie in Formel 10 gezeigt initialisiert.

$$A = 0x01234567$$
$$B = 0x89ABCEF$$
$$C = 0xFEDCBA98$$
$$D = 0x76543210$$

Formel 10: MD5 Initialwerte

Für die Berechnung des Hashwerts ist eine Liste von 64 32-Bit-Rundenkonstanten, *T*, nötig. Diese Berechnen sich über die, in Formel 11, dargestellte Vorschrift. Die 64 Werte müssen nicht neu berechnet werden, sie können im Programmcode oder einer ROM-Tabelle abgelegt werden.

$$T[i] = \left\lfloor \left|\sin(i+1)\right| * 2^{32} \right\rfloor \qquad 0 \le i < 64$$

Formel 11: MD5 Rundenkonstanten

Weiter ist sind insgesamt 64 Verschiebekonstanten S nötig. Jede wird in vier Runden verwendet. Diese können ebenfalls im Programmcode integriert werden.

### 6.1.3 Ablauf der Berechnung eines Blocks

Jede Spalte $X_j[k]$ des MD5 Blocks $X_j$ durchläuft die vier, in Formel 12 dargestellten, nichtlinearen Funktionen, insgesamt werden also $i$=64 Funktionsaufrufe während der Berechnung eines Blocks ausgeführt.

$$0 \le i \le 15 : F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$$
$$16 \le i \le 31 : G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$$
$$32 \le i \le 47 : H(X,Y,Z) = X \oplus Y \oplus Z$$
$$48 \le i \le 63 : I(X,Y,Z) = Y \oplus (X \vee \neg Z)$$

$\oplus$ bezeichnet bitweise XOR - Operation
$\neg$ bezeichnet bitweise NOT - Operation
$\wedge$ bezeichnet bitweise AND - Operation
$\vee$ bezeichnet bitweise OR - Operation

Formel 12:Md5-Hashfunktionen

Vor der Verarbeitung des Blocks wird der vorherige Wert des Hash gespeichert. Abhängig vom Rundenindex *i* werden die Worte *B*, *C*, und *D* des Hash mit einer der vier Funktionen miteinander verarbeitet. Das Ergebnis der Funktion wird mit dem Wort *A*, mit dem Rundenindex *i*, einer Konstante $K_i$ und dem Nachrichtenwort $M_i$ verknüpft, um einen für die Runde konstanten Offset *S* nach links verschoben und zu *B* addiert. Der Wert von *B* wird durch diesen Wert ersetzt und die Positionen der Worte des Hash werden vertauscht.

$$A = B + ((A + Hashfunktion(B, C, D) + X_j[k] + T_i) << S[i])$$
$$D \leftarrow C, C \leftarrow B, B \leftarrow A, A \leftarrow D$$

Formel 13: MD5 Runde

Am Ende der Berechnung eines Blocks werden die Worte des neuen Hashwerts zu dem alten Hashwert, beim ersten Block zum Initialwert, addiert.

Alle Additionen sind als Additionen Modulo $2^{32}$ zu verstehen, alle Zahlen sind natürliche Zahlen.

# 7 Architektur des Security-Chips

## 7.1 Architekturschichten

Die Architektur des Security Chips wurde strikt Modular im Bottom-Up-Verfahren entwickelt. Das System des Security Chips auf dem FPGA gliedert sich in vier Hierarchieebenen, die in Abbildung 11 dargestellt sind.

- Das Interface-Level stellt den Komponenten eine byteorientierte Kommunikationsschnittstelle bereit. Im vorliegenden Fall wurde ein SPI-Slave-Controller vgl. Kapitel 8 implementiert. Auf dieser Ebene des Systems werden keine höheren Protokolle ausgewertet. Das Interface-Level des Systems entspricht der Schicht 2, der MAC-Layer, des OSI-Referenzmodells.

- Das System-Level implementiert einen Zustandsautomaten, den `System-Controller`, vgl. Kapitel 9, der Steuerbytes auswertet und Komponenten des Corecontroller-Level aktiviert. Ist eine Komponente ausgewählt, wird ihr der Kontrollfluss übergeben. Das System-Level ist mit Schicht 3, der Vermittlungsschicht, des OSI-Referenzmodells vergleichbar.

- Im Corecontroller-Level wertet der, vom SystemController ausgewählte, Corecontroller, die empfangenen Bytes nach Daten- und Kontrollbytes aus. Der Corecontroller bereitet die Daten für das Interface des jeweiligen Algorithmus auf und steuert diesen. Auf dieser Hierarchieebene wird das Protokoll des Algorithmus implementiert. Ist ein Corecontroller ausgewählt, kommuniziert er mittels Systembus, siehe Kapitel 9.2 mit der SPI-Schnittstelle solange, bis er den Kontrollfluss von sich aus an den SystemController zurückgibt. Diese Schicht vereint die Sitzungsschicht und die Darstellungsschicht des OSI-Schichtenmodells.

- Das Core-Level entspricht der Anwendungsschicht des OSI-Schichtenmodells. Hier sind die kryptographischen Algorithmen mit ihrer spezifischen Schnittstelle implementiert. Die Algorithmen werten keine Steuerbytes des Protokolle aus sondern reagieren lediglich auf Signale des jeweiligen Corecontrollers.

Da die Komponenten einer Hierarchieebene untereinander keine Abhängigkeiten aufweisen und die Abhängigkeiten über Ebenen hinweg nur unidirektional von einer höheren Eben zur nächst niedrigeren bestehen, können problemlos Komponenten ausgetauscht, weggelassen oder hinzugefügt werden. Damit ist das System sehr leicht an geänderte Anforderungen anpassbar. Darüber hinaus erleichtert der modulare Aufbau das Testen einzelner Komponenten und reduziert die Komplexität des Systemtests.

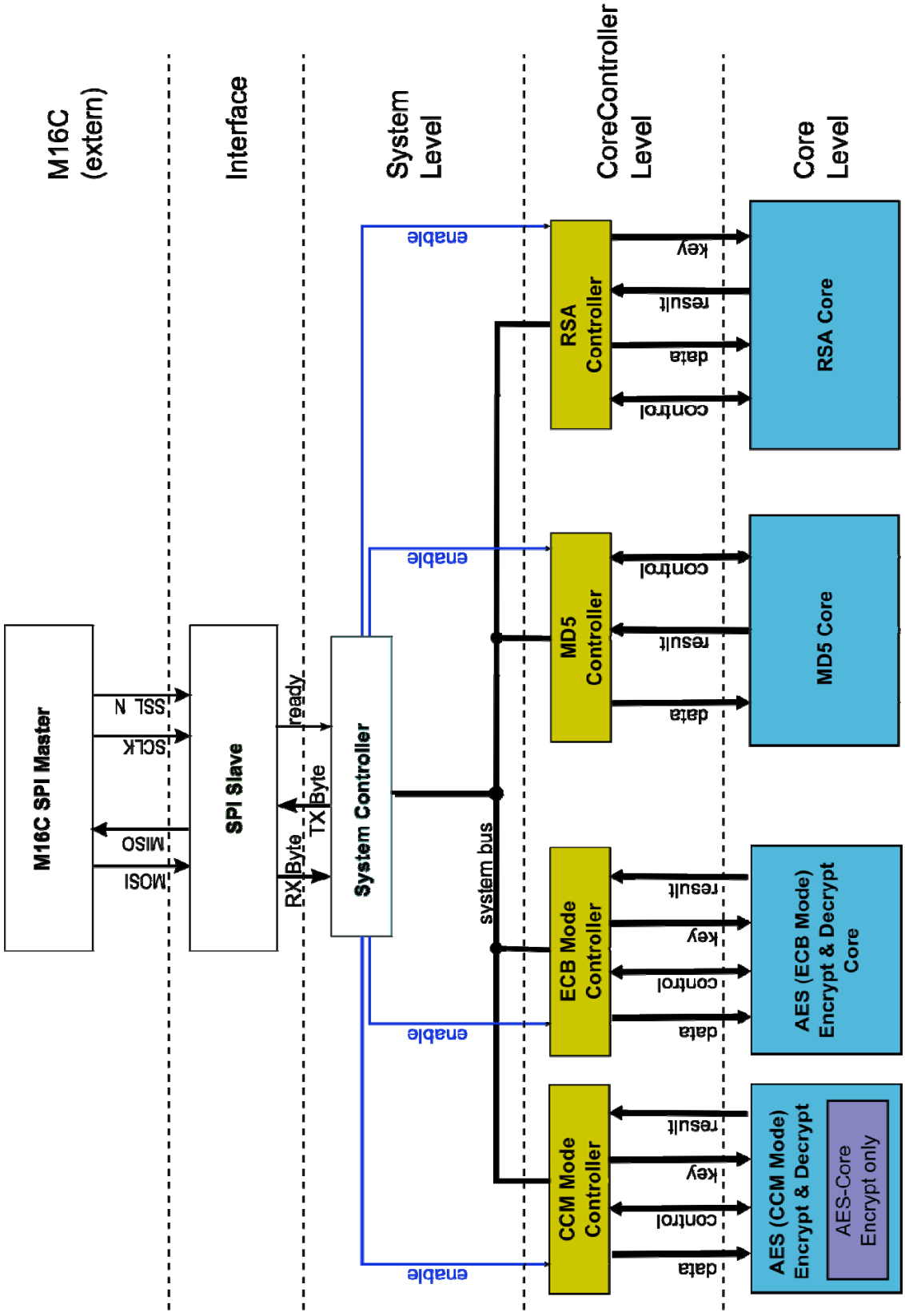


Abbildung 11: Architektur des Security Chips

## 7.2 Hardwareressourcen

Der Ressourcenverbrauch und die maximale Betriebsfrequenz des gesamten Security-Chips wurden mit „Mentor Graphics Precision RTL 2007“ ermittelt und sind lediglich eine Abschätzung.

| FPGA | LUTs/LCs | Register | Memory |
|---|---|---|---|
| Cyclone II EP2C70F896C | 23581 (34,5%) | 14064 (20,5%) | 150528 (13,7%) |

Tabelle 3: FPGA Ressourcen (Security-Chip)

Die maximale Betriebsfrequenz liegt bei 57,6 MHz, 17,35 ns Register-Register-Delay. Der kritische Pfad liegt in der MD5-Core-Komponente aus Kapitel 13.1.

# 8 SPI-Slave-Komponente

Die SPI-Slave-Komponente implementiert die SPI-Schnittstelle des Security-Chips. Dabei wurde die Konfiguration an den synchronen Modus der seriellen Schnittstelle des M16C-Mikrokontrollers angepasst. Alle Signale sind über Dual-Rank-Synchronizer mit zwei Flip-Flops synchronisiert. Die Frequenz der SPI-Clock `SCLK` darf daher nicht mehr als ein Viertel der Systemfrequenz sein.

$$f(sclk) <= 0{,}25\, f(clk)$$

Formel 14: Spi-Clockfrequenz

## 8.1 SPI Schnittstelle

Um den Kryptographieprozessor an den Mikrocontroller anzubinden soll SPI (**S**erial **P**eripheral **I**nterface) als Schnittstelle verwendet werden. SPI ist ein relativ einfaches Interface, das zur Kommunikation zwischen beliebigen digitalen Bausteinen entworfen wurde. Es ist kein hardwareseitiges Kommunikationsprotokoll für diesen Bus spezifiziert, es wird keine Prüfsumme zur Verifikation der Daten erzeugt, ferner existiert keine Flusskontrolle des Datenstroms [Kal02]. Wird über SPI kommuniziert so müssen die Daten mit Protokollen höherer Schichten kontrolliert werden.

Zur Kommunikation sind lediglich drei Leitungen erforderlich: `MISO` (**M**aster **I**n **S**lave **O**ut), `MOSI` (**M**aster **O**ut **S**lave **I**n) und die Taktleitung `SCLK` (**S**erial **C**lock), die den Takt zur seriellen Kommunikation vorgibt, dieser Takt wird immer vom Master vorgegeben. Da der Takt zusammen mit den Daten übertragen wird, ist im Gegensatz zur asynchronen Übertragung bei einer UART-Schnittstelle keine Synchronisation zwischen Sender und Empfänger über Start- oder Stopp-Bits notwendig. Zusätzlich kann bei mehreren Slaves im System über eine `SSL_N` (**S**lave-**S**elect) Leitung ein Gerät ausgewählt werden.

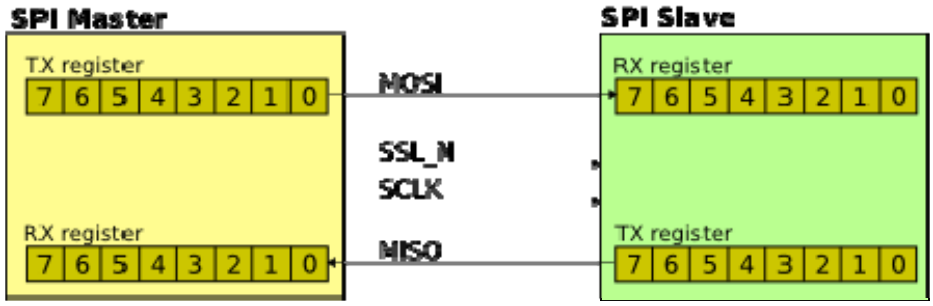


Abbildung 12: SPI Architektur

| Signal | Treiber | Funktion |
|---|---|---|
| SCLK | Master | Takt |
| SSL_N | Master | Slave Select (active low) |
| MOSI | Master | Master Out Slave In |
| MISO | Slave | Master In Slave Out |

Tabelle 4: SPI Signale

Das Interface arbeitet im Full-Duplex-Mode, das heißt, es werden gleichzeitig Daten empfangen und gesendet. Wenn nur ein Gerät Daten zu senden hat, werden vom Kommunikationspartner Füllbytes übertragen. Der Empfänger muss über höhere Protokollschichten entscheiden, ob dies gültige Daten sind, oder lediglich Fülldaten.

Ein Slave-Gerät kann nicht aktiv eine Übertragung beginnen, wenn z. B. Daten bereit sind, da SPI keine Interruptleitungen oder Handshakes vorsieht. Der Protokollmaster muss mittels „Polling“ regelmäßig Daten von den Slave-Geräten anfordern. Der Slave muss mit Längeninformationen antworten, wie viele Datenbytes gesendet werden sollen. Der Master sendet dann so viele (Füll-)Bytes, wie der Slave Datenbytes senden soll.

### 8.1.1 Ablauf der Datenübertragung

1. Master treibt `SSL_N = `0``
2. Bit 1 des TX-Registers an `MOSI` (Master) bzw. `MISO` (Slave)
3. Master treibt `SCLK` Takt.
4. Bei steigender Flanke wird `MOSI` bzw. `MISO` in `RX`-Register geschrieben und das nächste Bit an `MOSI` bzw. `MISO` angelegt.

Sind alle Bits übertragen, kann `SSL_N` wieder auf `` `1` `` getrieben werden oder das nächste Wort wird nach einer vorgeschriebenen Wartezeit übertragen.

## 8.2 Interface

| Signal | Richtung | Typ |
|---|---|---|
| clk | IN | STD_LOGIC |
| res_n | IN | STD_LOGIC |
| MOSI | IN | STD_LOGIC |
| SCLK | IN | STD_LOGIC |
| SSL_N | IN | STD_LOGIC |
| MISO | OUT | STD_LOGIC |
| io_ack | OUT | STD_LOGIC |
| rx_data | OUT | BYTE |
| tx_data | IN | BYTE |
| Wreq | IN | STD_LOGIC |

Tabelle 5: Interface SPI-Slave-Controller

- **clk**: Systemtakt des Security-Chip
- **res_n**: Reset Port, des Chip ist active-low, alle Datenregister werden bei res_n='0 gelöscht und die Zustandsautomaten auf den Initialzustand gesetzt.
- **MOSI**: Master Out Slave In. Datenleitung vom SPI-Master, taktsynchron mit SCLK. Daten werden mit der steigenden Flanke von SCLK in rx_data übernommen.
- **SCLK**: Taktleitung für SPI Datenübertragung, wird vom SPI-Master generiert.
- **SSL_N**: Wählt Security Chip als SPI-Slave für Datenübertragung. Nur wenn ssl_n='0' ist das SPI-Interface aktiv und an MISO liegen Daten an, bei ssl_n='1' wird MISO hochohmig.
- **MISO**: Master In Slave Out. Datenleitung vom SPI-Slave zum Master. Wird nur bei ssl_n='0' getrieben hierfür muss der FPGA über Tri-State-IO-Pins verfügen.
- **Rx_data**: Empfangenes Datenbyte. Rx_data wird von Bit #7 bis Bit#0 gefüllt.
- **Tx_data**: Wird bei der nächsten SPI-Übertragung bitweise über MISO gesendet. Bit#7 wird als Erstes gesendet.
- **Wreq**: Nur bei wreq='1' werden die Daten an tx_data in den Sendepuffer übernommen, sonst wird ,0' übertragen.
- **Io_ack**: Datenbyte an rx_data ist gültig und Datenbyte von tx_data wird bei wreq='1' in den internen Sendepuffer übernommen.

## 8.3 Protokoll

Im Initialzustand ist der Zähler `bitcnt=7`, mit jeder steigenden Flanke von `SCLK` wird das Bit an `MOSI` in `rx_data[bitcnt]` übernommen. `MISO` wird vom M16C Mikrokontroller an der fallenden Flanke von `SCLK` ausgewertet. Der Zähler `bitcnt` wird an der fallenden Flanke dekrementiert. Sind acht Bit übertragen, d. h. `bitcnt=0` und es liegt eine fallende Flanke von `SCLK` geht der SPI-Slave für eine Periode von `clk` in den Zustand `STORE.` Im Zustand `STORE` ist `io_ack='1'`, `rx_data` kann ausgewertet werden und mit wreq='1' wird tx_data in den Sendepuffer übernommen. Im `STORE`-Zustand liegt an `MISO` Bit #0 des Sendepuffers an.

Vor einer SPI-Übertragung liegt an `MISO` bereits Bit#7 des Sendepuffers an.

## 8.4 Timing Diagramm

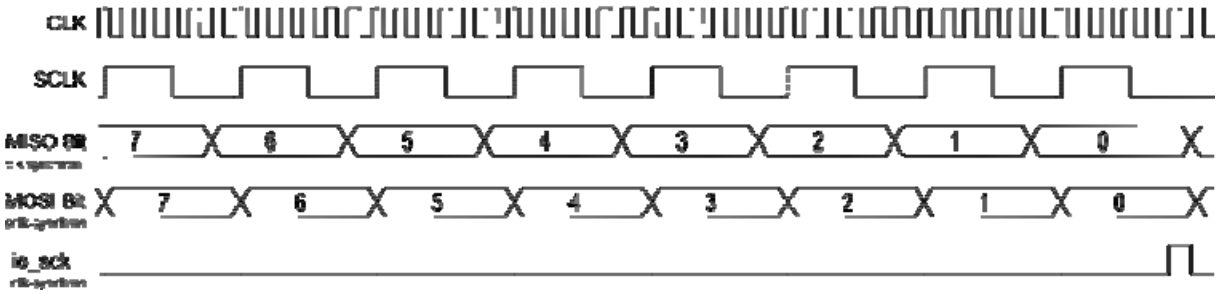


Abbildung 13: SPI-Timing

Die Verschiebung von MISO gegenüber SCLK rührt daher, dass aufgrund der Dual-Rank-Synchronisation die fallende Flanke erst zweit clk-Perioden später ausgewertet wird.

# 9 Systemcontroller-Komponente

Die Systemcontroller-Komponente dient der Auswahl der Funktionen des Security-Chips. Sie ist nur aktiv, wenn sich alle Corecontroller im Initialzustand befinden. Da auf den meisten FPGAs keine internen Tri-State-Signale realisiert werden können, schaltet der SystemController Multiplexer, die es den Corecontroller ermöglichen, der SPI-Slave-Komponente Daten zu senden.

## 9.1 Interface

Der SystemController bietet eine byteorientierte Schnittstelle zur SPI-Slave-Komponente und vier Systembus-Signalleitungen vom Typ `System_IF`, vgl. Kapitel 9.2 zur Kommunikation mit den Corecontroller.

| Signal | Richtung | Typ |
|---|---|---|
| clk | IN | STD_LOGIC |
| res_n | IN | STD_LOGIC |
| Spi_rx_data | IN | BYTE |
| Spi_tx_data | Out | BYTE |
| Spi_ready | IN | STD_LOGIC |
| Spi_wreq | OUT | STD_LOGIC |
| Comp_ecb | INOUT | System_IF |
| Comp_ccm | INOUT | System_IF |
| Comp_rsa | INOUT | System_IF |
| Comp_md5 | INOUT | System_IF |

Tabelle 6: Interface SystemController

- **`clk`**: Systemtakt des Security-Chip
- **`res_n`**: Reset Port, des Chip ist active-low, alle Datenregister werden bei `res_n='0` gelöscht und alle Zustandsautomaten auf den Initialzustand gesetzt.
- **`Spi_rx_data`**: empfangenes Datenbyte, `rx_data`, der SPI-Slave-Komponente aus Kapitel 8.2. Dieses Signal wird, da es nur gelesen wird, nicht mit Multiplexern umgeschaltet sondern an alle Corecontroller weitergeleitet.
- **`Spi_tx_data`**: Zu sendendes Datenbyte, `tx_data`, der SPI-Slave-Komponente aus Kapitel 8.2. Dieses Signal wird mittels Multiplexern geschaltet werden da alle Corecontroller dieses Signal treiben.

- **Spi_ready**: Daten an spi_rx_data sind gültig und Daten an spi_tx_data werden, wenn spi_wreq='1' gesendet. Vergleiche hierzu io_ack der SPI-Slave-Komponente aus Kapitel 8.2.

- **Spi_wreq**: Bei spi_wreq='1' sind die Daten an spi_tx_data gültig und sollen gesendet werden. Dieses Signal wird mittels Multiplexern geschaltet werden da alle Corecontroller dieses Signal treiben. Vergleiche hierzu wreq der SPI-Slave-Komponente aus Kapitel 8.2.

Bei folgenden Signalen handelt es sich um die Kommunikationsschnittstelle mit den Corecontrollern, die als Signal vom Typ System_IF vgl. Kapitel 9.2 realisiert ist.

- **Comp_ecb**: Systembus-Leitung der AES-ECB-Komponente. Ist aktiv, wenn vorher die Konstante MB_ECB empfangen wurde. Und die Komponente den Kontrollfluss noch nicht zurückgegeben hat.

- **Comp_ccm**: Systembus-Leitung der AES-CCM-Komponente. Ist aktiv, wenn vorher die Konstante MB_CCM empfangen wurde. Und die Komponente den Kontrollfluss noch nicht zurückgegeben hat.

- **Comp_rsa**: Systembus-Leitung der RSA-Komponente. Ist aktiv, wenn vorher die Konstante MB_RSA empfangen wurde. Und die Komponente den Kontrollfluss noch nicht zurückgegeben hat.

- **Comp_md5**: Systembus-Leitung der MD5-Komponente. Ist aktiv, wenn vorher die Konstante MB_MD5 empfangen wurde. Und die Komponente den Kontrollfluss noch nicht zurückgegeben hat.

## 9.2 System_IF als Systembus

Der Systembus des Security-Chips ist als Single-Master Busarchitektur entworfen. Der SystemController dient als Busmaster, der Sendeberechtigung an die Slaves, die Corecontroller verteilt. Nur ein Corecontroller darf zu einem Zeitpunkt aktiv sein, dies wird mit dem Signal ena='1' der Systembusleitung des jeweiligen Controllers angezeigt.

Der Signaltyp System_IF aus dem VHDL-Package system_pkg ist ein Record aus sechs Signalen. Der Systembus dient als einheitliche Schnittstelle aller CoreController zum SystemController.

| Signal | Typ | Treiber |
|---|---|---|
| Rx_data | BYTE | SystemController |
| Io_ack | STD_logic | SystemController |
| Tx_data | Byte | CoreController |
| Wreq | STD_logic | CoreController |
| Finished | STD_logic | CoreController |
| Ena | STD_logic | SystemController |

Tabelle 7: System_IF Signale

- **rx_data**: empfangenes Datenbyte, entspricht. spi_rx_data des SystemController. Dieses Signal wird an alle Systembusteilnehmer weitergeleitet.
- **io_ack**: rx_data ist gültig und tx_data kann geschrieben werden wenn io_ack='1', vgl. spi_ready des Systemcontrollers. Dieses Signal wird an alle Systembusteilnehmer weitergeleitet.
- **tx_data**: Sendedaten des Corecontrollers, vgl. spi_tx_data des SystemControllers.
- **wreq**: Sendedaten tx_data des Corecontrollers sind gültig wenn wreq='1', vgl. spi_wreq des Systemcontrollers.
- **finished**: Der Corecontroller gibt mit finished='1' den Kontrollfluss an den SystemController zurück.
- **ena**: Der CoreController ist ausgewählt, hat Sendeberechtigung auf tx_data, wreq und finished. Nur wenn ena='1' ist soll der CoreController die Daten an rx_data auswerten.

## 9.3 Protokoll

Im Initialzustand wertet der SystemController das von der SPI-Slave-Komponente empfangene rx_byte aus. Unbekannte Steuersignale werden verworfen. Wird ein bekanntes Steuersignal empfangen, so gibt der SystemController den Kontrollfluss an einen der CoreController weiter. Die Multiplexer für den Systembus werden umgeschaltet und die gewählte Komponente wird aktiviert. Danach werden keine Steuerbytes vom SystemController mehr ausgewertet bis die gewählte Komponente, über finished='1'des Systembus aus Kapitel 9.2 den Kontrollfluss an den SystemController zurückgibt.

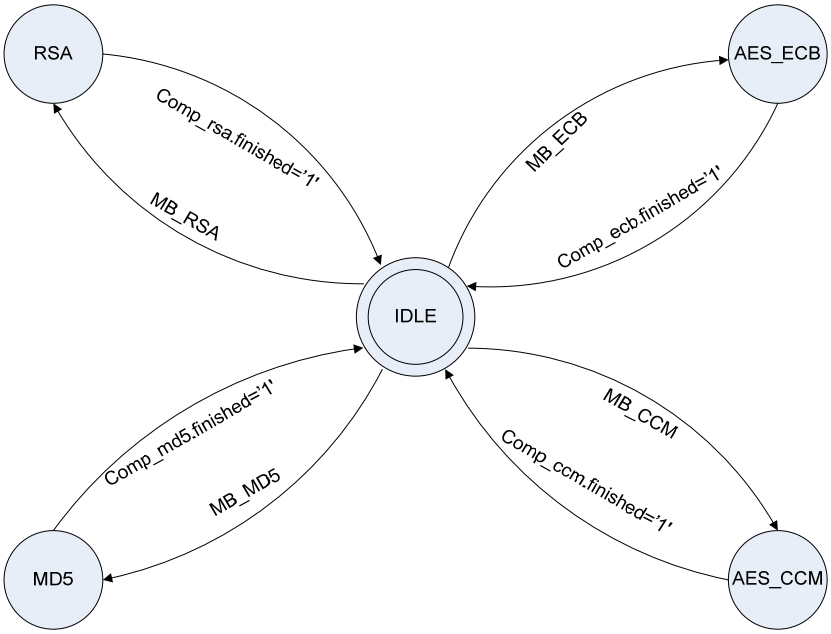


Abbildung 14: Zustandsautomat des SystemController

# 10 AES-ECB-Komponente

Die AES-ECB-Komponente besteht aus dem AES-ECB-Core und dem dazugehörigen ECBCoreController. Sie verschlüsselt Blocks mit 128 Bit Länge mit dem AES (Rijndael) Algorithmus, vgl. [FIPS197] im Electronic-Codebook-Mode.

## 10.1 AES-ECB-Core Komponente

Der AES-ECB-Core ist die Hardwareimplementierung des AES Algorithmus. Alle Module dieser Komponente befinden sich im Ordner `./hdl/aes_ecb/` die Module werden in die VHDL-Library `AES_ECB_lib` kompiliert. Zusätzlich sind Module der VHDL-Library `util_lib` nötig. Alle Typen sind im VHDL-Package `aes_ecb_pkg` definiert.

Der AES-ECB-Core unterteilt sich in Verschlüsselungs- und Entschlüsselungsdatenpfad, die in der Strukturbeschreibung `AES_core_ecb_struct.vhd` zusammengefasst sind, und von einem gemeinsamen Zustandsautomat aus der Datei `AES_ECB_FSM.vhd` gesteuert werden. Allerdings ist immer nur ein Datenpfad aktiv, da die Teile den `Keygenerator` und die IO/Ports gemeinsam nutzen. Zur Einsparung von Ressourcen wird in dieser Implementierung des SBOX-Schrittes einer AES-Runde als Multicycle-Operation ausgeführt. In vier Taktzyklen werden nacheinander die vier Spalten des STATE mit der SBOX substituiert.

### 10.1.1 Interface

| Signal | Richtung | Typ |
|---|---|---|
| clk | IN | STD_LOGIC |
| res_n | IN | STD_LOGIC |
| data_in | IN | STATE |
| data_stable | IN | STD_LOGIC |
| direction | IN | CRYPTODIRECTION |
| initialkey | IN | KEYBLOCK |
| key_stable | IN | STD_LOGIC |
| run | IN | STD_LOGIC |
| cyphertext | OUT | STATE |
| finished | OUT | STD_LOGIC |

Tabelle 8: Interface AES-ECB-Core

- **`clk`**: Systemtakt des Security-Chip
- **`res_n`**: Reset Port des Chip ist active low, alle Datenregister werden bei `res_n='0` gelöscht und alle Zustandsautomaten auf den Initialzustand gesetzt.

- **data_in**: Port für Datenblock zur Ver- oder Entschlüsselung.
- **data_stable**: signalisiert Gültigkeit der Daten an data_in. data_in wird in das Stateregister des AES-Core übernommen, wenn data_stable='1' ist.
- **direction**: wählt mit Hilfe der Protokollkonstanten MB_ENCRYPT und MB_DECRYPT den Ver- oder Entschlüsselungsmodus aus.
- **initialkey**: Benutzerschlüssel für Ver- und Entschlüsselung.
- **key_stable**: signalisiert Gültigkeit des Benutzerschlüssels. Ist key_stable='1' werden aus dem Benutzerschlüssel an initialkey die Rundenschlüssel erzeugt. Nach elf Takten sind alle Schlüssel bereit. Solange bis ein neuer Schlüssel geladen werden soll muss key_stable='1' sein. Das Signal darf während einer Ver- oder Entschlüsselungsoperation nicht auf key_stable='0' wechseln. Key_stable='0' löscht alle Rundenschlüssel.
- **run**: Startet die Ver- oder Entschlüsselungsoperation nachdem der Benutzerschlüssel geladen, die Rundenschlüssel erzeugt und data_in mit data_stable='1' übernommen wurde. Ist run='0' wird die Operation abgebrochen. Run ist somit auch ein synchroner Reseteingang.
- **cyphertext**: Ergebnis der Ver- oder Entschlüsselungsoperation.
- **finished**: wenn finished='1' sind die Daten an cyphertext gültig und die Operation ist beendet.

### 10.1.2 Protokoll

Beim ersten Aufruf wird zuerst der Benutzerschlüssel geladen und die Rundenschlüssel erzeugt werden. Initialkey in den Keygenerator geladen wenn key_stable='1'. Key_stable='1' muss während der gesamten Ver- oder Entschlüsselungsoperation gehalten werden, sobald key_stable='0' ist werden die Rundenschlüssel gelöscht und ungültig und es kann ein anderer Schlüssel geladen werden. Die Generierung der Rundenschlüssel dauert elf Takte. Der STATE an data_in wird mit data_stable='1' in das Stateregister des AES-Core übernommen. Mit run='1' wird der Algorithmus gestartet. Der Port cyphertext ändert bei jeder Runde seinen Wert auf das Zwischenergebnis. Ist finished='1', ist cyphertext gültig.

### 10.1.3 Timing Diagramm

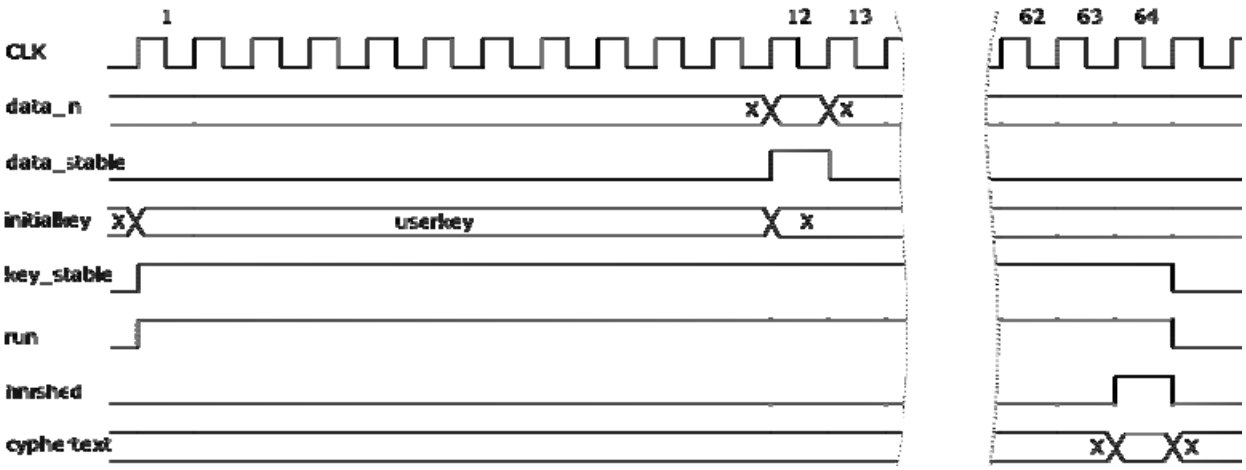


Abbildung 15: AES-ECB-Core Timing

### 10.1.4 Blockschaltbilder

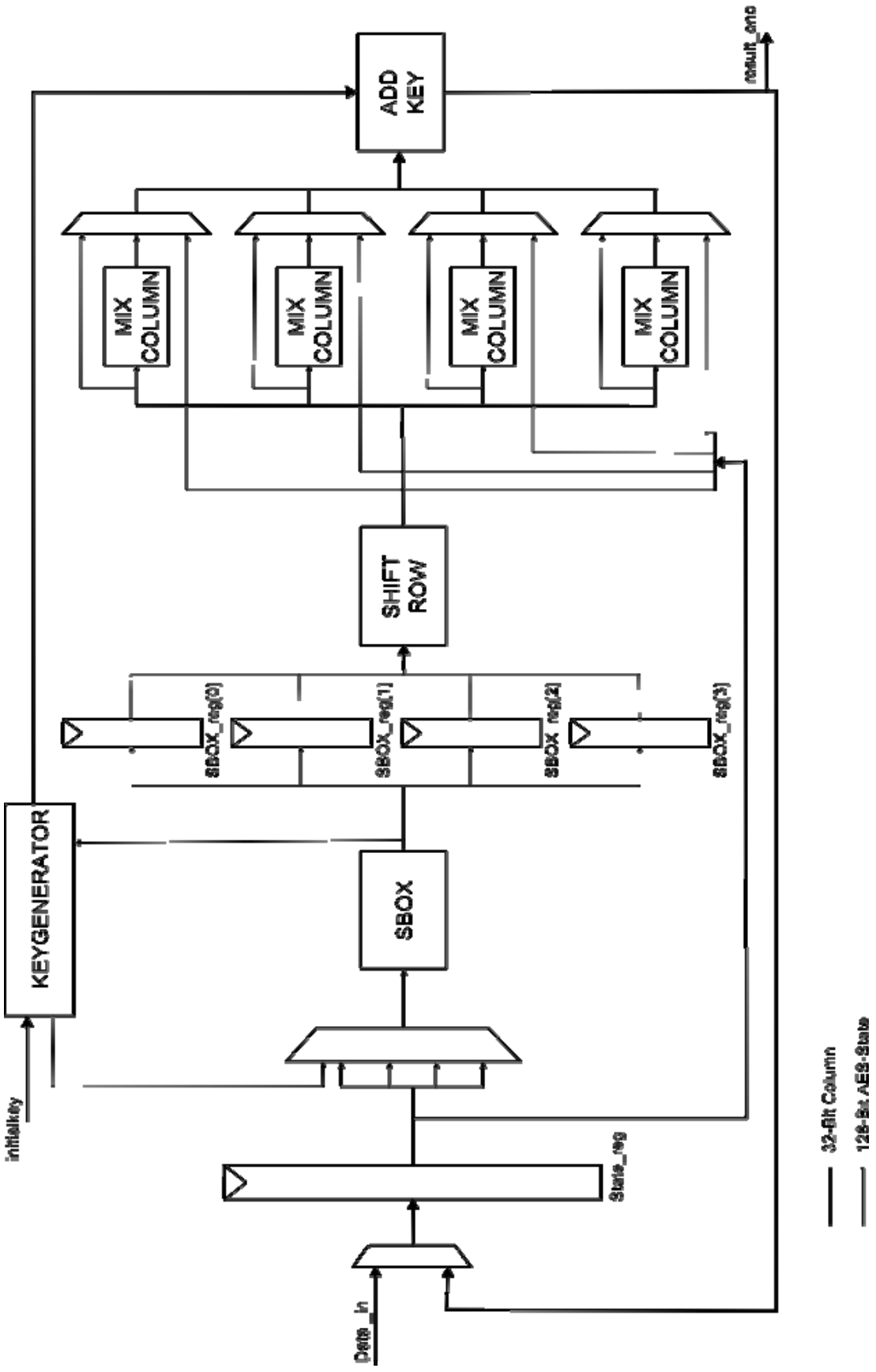


Abbildung 16: Verschlüsselungsdatenpfad

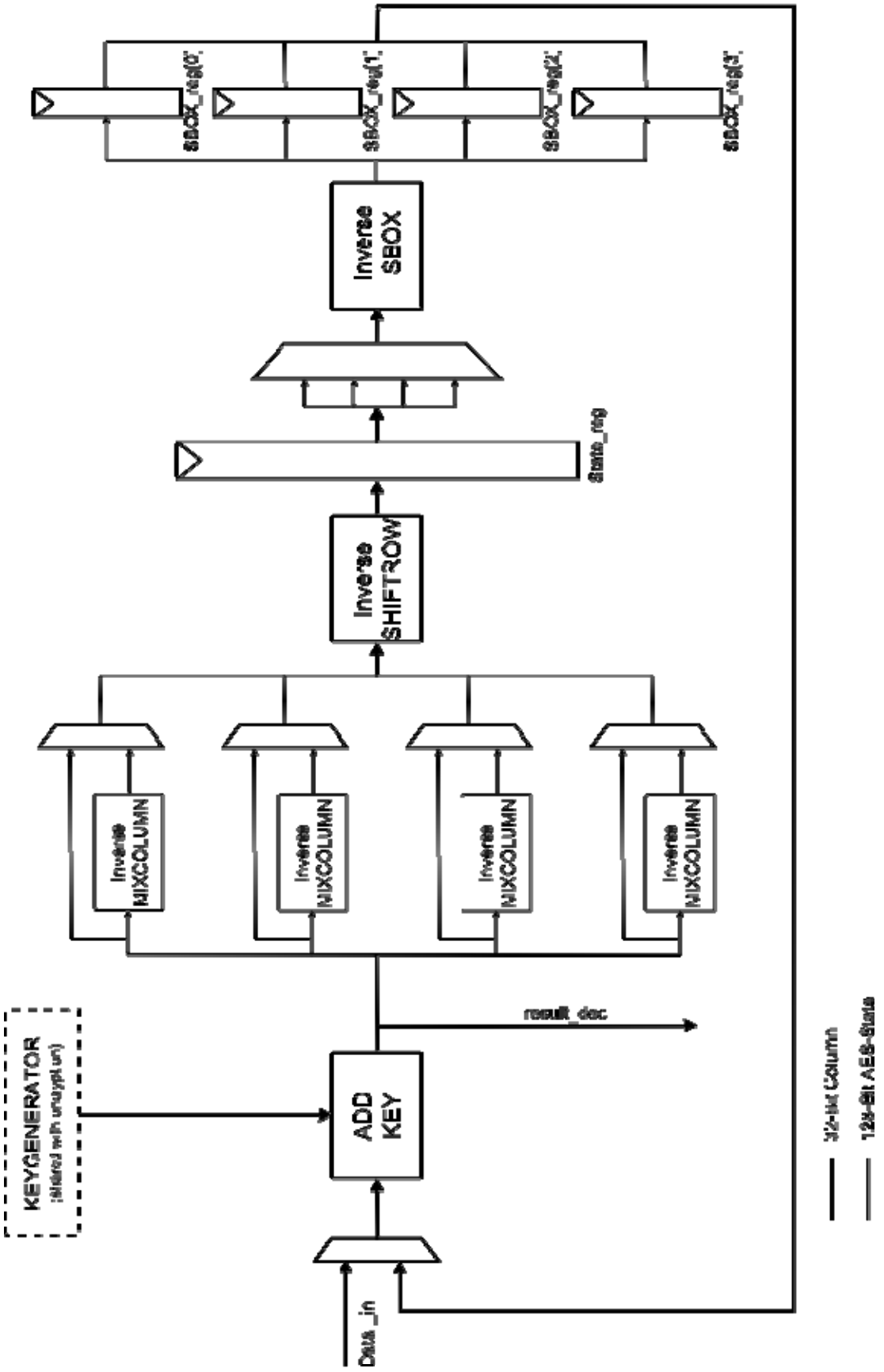


Abbildung 17: Entschlüsselungsdatenpfad

### 10.1.5 FPGA-Ressourcen

Die benötigten Ressourcen auf Altera FPGAs wurden mit „Mentor Graphics Precision RTL 2007“ ermittelt und sind lediglich eine Abschätzung.

| FPGA | LUTs/LCs | Register |
|---|---|---|
| APEX 20KC - EP20K400CB652C | 4588 (27,57%) | -- |
| Stratix2 - EP2S60F1020I | 3053 (6,31%) | 2735 (5,21%) |
| Cyclone2 - EP2C35F484I | 6395 (19,25%) | 2729 (8,22%) |

Tabelle 9: FPGA Ressourcen (AES-ECB)

## 10.2 ECBCoreController

Der AES-ECB-CoreController bindet den AES-ECB-Core über den Systembus an den SystemController, vgl. Kapitel 9.2 und das SPI-Slave-Module, vgl. Kapitel 8 an. In diesem Modul werden die Protokollinformationen ausgewertet und die Datenbytes der SPI-Schnittstelle in das entsprechende Datenformat des Interface des AES-ECB-Core umgesetzt.

### 10.2.1 Interface

Die Schnittstelle des `ECBCoreControllers` setzt sich aus dem Systembus „`sysconnect`“ vom Typ `System_IF` aus Kapitel [9.2] zum SystemController, vgl. Kapitel 9 und der Schnittstelle des AES-ECB-Core [10.1.1] zusammen.

### 10.2.2 Protokoll

Nur wenn `sysconnect.ena='1'` ist, arbeitet die Komponente, sonst sind die Speicher der Komponente angehalten und sie verharrt in ihrem Zustand. Zu Beginn befindet sich die Komponente im Verschlüsselungsmodus.

Befindet sich der Zustandsautomat der Komponente im Zustand `IDLE`, kann mit der Konstanten `MB_LOADKEY` ein neuer Schlüssel geladen werden. Im Zustand `LOADKEY` werden die nächsten 16 Bytes als Schlüsseldaten interpretiert. Ist der Schlüssel vollständig empfangen, wird dies gespeichert, und der Automat kehrt in den `IDLE`-Zustand zurück. Dieser Schlüssel bleibt über weitere Aufrufe der Komponente hinweg gültig, bis wieder das Steuerbyte `MB_LOADKEY` empfangen wird. Das Signal `key_stable` des AES-ECB-Core bleibt logisch „1“ und die Rundenschlüssel bleiben erhalten. Mit der Konstanten `MB_LOADDATA` geht der Automat in den Zustand `LOADDATA` über und es wird ein neuer Datenblock für `data_in` geladen. Befindet sich der Automat im Zustand `LOADDATA` werden die nächsten 16 Bytes als Daten interpretiert. Sind die Daten vollständig empfangen, wird dies gespeichert, das Signal `data_stable='1'` gesetzt und der Automat kehrt in den `IDLE`-Zustand zurück. Schlüssel und Daten können in beliebiger Reihenfolge geladen werden, es ist außerdem jederzeit möglich die geladenen Daten oder Schlüssel erneut zu überschreiben.

Im IDLE-Zustand kann mit MB_DECRYPT der Algorithmus in den Entschlüsselungsmodus gewechselt werden. Mit MB_ENCRYPT wird wieder in den Verschlüsselungsmodus zurückgewechselt.

Mit dem Steuerbyte MB_START wird die Verarbeitung der geladenen Daten gestartet. Ob ein Ergebnisdatenblock vorliegt kann mit MB_POLL abgefragt werden. Auf MB_POLL sendet die Komponente mit dem nächsten übertragenen Byte der SPI-Schnittstelle die Länge der Ergebnisdaten zurück. In diesem Fall ist es entweder 0, keine Daten sind vorhanden, oder 16 ein Datenblock mit 16 Bytes Länge ist bereit. Ist ein Ergebnis vorhanden, kann die Übertragung des Ergebnisses mit der Konstanten MB_RESULT gestartet werden. Werden aber neue Daten, Schlüssel geladen oder der Algorithmus gestartet wird das Ergebnis überschrieben.

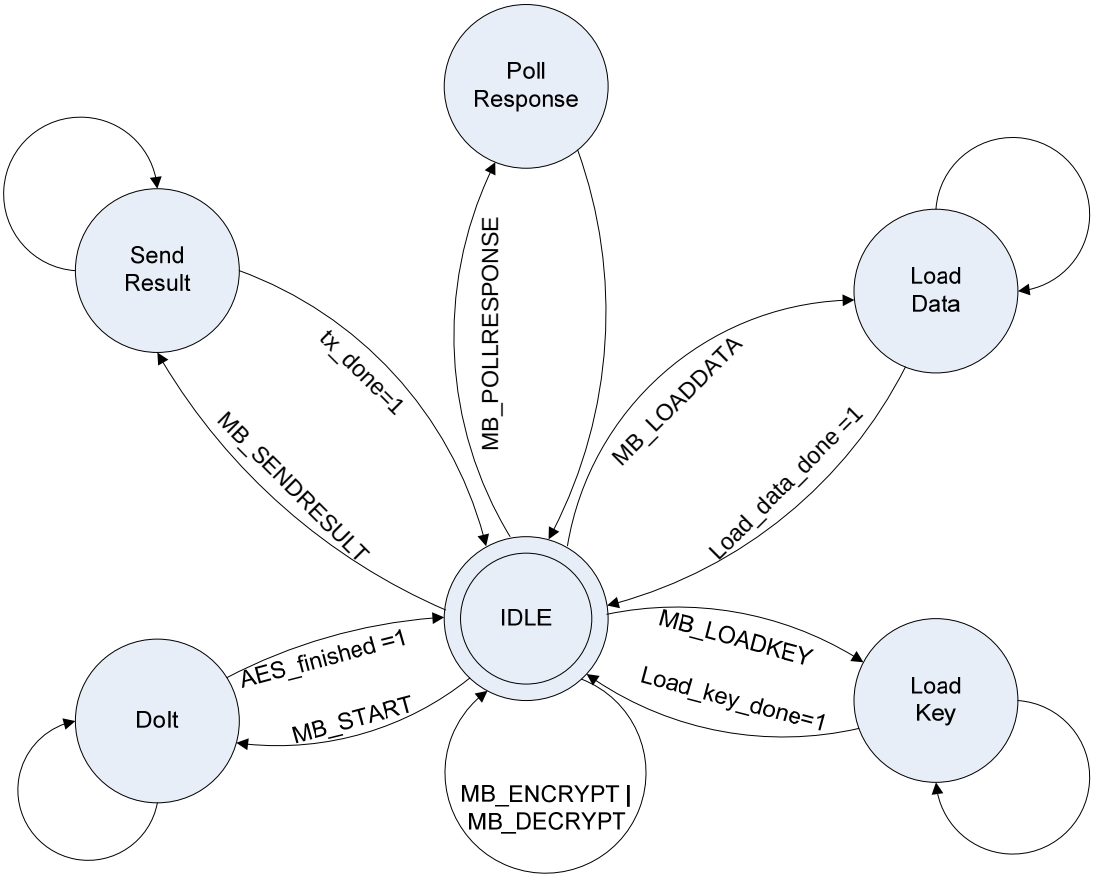


Abbildung 18: Zustandsautomat ECBCoreController

# 11 AES-CCM-Komponente

Die AES-CCM-Komponente besteht aus dem AES-CCM-Core und dem dazugehörigen CCMCoreController. Sie verschlüsselt Blocks mit 128 Bit Länge mit dem AES (Rijndael) Algorithmus, vgl. [RFC3610] im Counterblock-Chaining-Mode und erstellt einen MAC, (**M**essage **A**uthentication **C**ode) im Cypherblock-Chaining-Verfahren. Ein MAC ist eine kryptographische Prüfsumme, die nur mit einem geheimen Schlüssel berechnet werden kann. Sie schützt Manipulation der Nachricht. Eine vollständige, im CCM-Verfahren geschützte Nachricht lässt sich in folgende Teile untergliedern.

- ein Header der nicht verschlüsselt aber authentifiziert wird, die sogenannte Associated Data oder „*adata*“.
- die *Payload*, Daten die sowohl authentifiziert ,als auch verschlüsselt werden.
- der MAC, die kryptographische Prüfsumme

Der Algorithmus besteht aus zwei Teilen, *encryption-generation* zur Verschlüsselung und Erstellung des MAC und *decryption-verification* zum Entschlüsseln der Nachricht und Prüfen des berechneten MAC gegen den empfangenen MAC.

Die CCM-Komponente benötigt lediglich den Verschlüsselungs-Datenpfad des zugrunde liegenden AES-Algorithmus.

## 11.1 Parameter für das CCM-Verfahren

Für die Berechnung des MAC und die Verschlüsselung der Payload der Nachricht muss die gesamte Nachricht im AES-CCM-Core gespeichert sein. Es ist nicht möglich, inkrementell die Nachricht kryptographisch zu bearbeiten [NIST38c]. Vor der Verarbeitung muss ein geheimer symmetrischer Schlüssel K in den AES-Core geladen werden. Unter Verwendung dieses Schlüssels werden sowohl der MAC als auch die verschlüsselte Payload berechnet. Die hier vorgestellte Implementierung beschränkt die Länge der Associated Data und der Payload auf je $2^{16}$ Bytes, da dies für die Verwendung in eingebetteten Feldgeräten aussreichend ist und auf dem FPGA der Speicher der FIFOs begrenzt ist.

Im Folgenden werden die Teile einer Nachricht und die Parameter des Algorithmus weiter beschrieben.

Eine Nachricht ist ein vier Teile zu unterteilen:

- Block0 der Parameter und Längen der Parameter beinhaltete, Der erste übertragene 128-Bit große Block der Nachricht Block0, $B_0$, ist wie in Tabelle 10 dargestellt definiert. Die Flags in Byte 0 sind in Tabelle 11 beschrieben.

- Nach Block0 folgen 0-10 Bytes *adata_len* für die Codierung der Länge der Associated Data, *adata*, mit entsprechenden Codierungsbytes. Die Codierung der *adata_len* ist in Tabelle 12 dargestellt. Dieser Teil wird nur ausgewertet wenn im Flags-Byte das sechste Bit *has_adata* gesetzt ist.

- Der Teil der, die Associated Data, *adata*, enthält folgt unmittelbar auf den *adata_len* Teil und beinhaltet die Headerinformationen der Nachricht, die zwar authentifiziert, nicht aber verschlüsselt werden. Die Länge dieses Teils wird durch *adata_len* definiert. Der Algorithmus erwartet genau soviele Bytes wie in *adata_len* angegeben wurden.

- Als letzter Teil der Nachricht folgt unmittelbar auf die Associated Data die sogenannte Payload, der Teil der Nachricht der sowohl verschlüsselt als auch authentifiziert wird.

| **Byte 0** | **Byte 1 bis Byte(15-q)** | **Byte(15-q) bis Byte 15** |
|---|---|---|
| Flags | Nonce | Q |

Tabelle 10: CCM Block0

Der Nonce ist ein kryptographischer Wert, der nur einmal unter einem bestimmten Kontext verwendet werden darf. Der Nonce muss nicht zufällig sein.[NIST38c]

Der Wert Q aus den letzten Bytes des Blocks, codiert die Länge der Payload in Bytes.

Das Flags-Byte ist unten beschrieben und ist maßgeblich dafür verantwortlich, dass der Bytestrom der Nachricht korrekt ausgewertet wird.

| **Bit 7** | **Bit 6** | **Bit 5** | **Bit 4** | **Bit 3** | **Bit 2** | **Bit 1** | **Bit 0** |
|---|---|---|---|---|---|---|---|
| X | has_adata | t_enc[2] | t_enc[1] | t_enc[0] | q_enc[2] | q_enc[1] | q_enc[0] |

Tabelle 11: CCM Flags-Byte

Bit 7 des Flags-Byte ist für zukünftige Verwendung reserviert und sollte immer 0 sein. Dieses Bit wird von der CCM-Komponente nicht ausgewertet. Bit 6 gibt an, ob die Nachricht Associated Data beinhaltet. Ist dieses Bit=1 gesetzt, so werden die Bytes nach Block0 als Länge der Associated Data interpretiert. Ist Bit6=0 so enthält die Nachricht keine Header und die Bytes nach Block 0 werden als Payload behandelt. Die Felder *Q_enc* und *T_enc* dürfen nicht 0 sein. *T_enc* codiert die Länge des MAC, *t*; in Bytes nach der Vorschrift:

$$t = \frac{t\_enc - 2}{2}.$$

Daraus folgt, dass der MAC die Länge $t \in \{4, 6, 8, 10, 12, 14, 16\}$ Bytes hat.

*Q_enc* codiert die Länge des Feldes Q von Block 0 nach der Vorschrift:

$$len(Q) = q = q\_enc - 1$$

Da $q \in \{2,3,4,5,6,7\}$ Bytes ist und die Länge des Nonce, *n*, als 15-q definiert ist ergibt sich für die Noncelänge n: $n \in \{8,9,10,11,12,13\}$.

Die Grösse der Associated Data ist auf $adata\_len < 2^{64}$ begrenzt. *Adata_len* wird mit Präfixes wie folgt codiert:

| **Codierung (adata_len)** | **Länge adata** | **Bytes für adata_len Feld** |
|---|---|---|
| Length in 2 Bytes | $0 < adata\_len < 2^{16} - 2^8$ | 2 |
| 0xFF, 0xFE, Length (4Bytes) | $2^{16} - 2^8 < adata\_len < 2^{32}$ | 6 |
| 0xFF,0xFF, Length (8Bytes) | $2^{32} < adata\_len < 2^{64}$ | 10 |

Tabelle 12: Längencodierung für Associated Data

### 11.1.1 Formatierung der Blocks

Die zugrunde liegende AES-Blockchiffre kann nur Blocks ganzer Länge verarbeiten. Daher werden Daten, die getrennt verarbeitet werden müssen und eine Länge ungleich eines vielfachen von 16 Bytes besitzen mit 0x00-Bytes aufgefüllt. Das sogenannte Padding erfolgt im Falle der Associated Data und der Payload.

Block 0 hat immer eine Länge von 16 Bytes und muss nicht aufgefüllt werden. Dieser Block dient als Initialwert für die Berechnung des MAC im *encryption-generation* Prozess. Dabei wird Block 0 mit dem angegebenen Schlüssel im ECB-Modus verschlüsselt.

An den Datenblock *adata_len*, mit Präfix-Bytes wird direkt die Associated Data angehängt, und gegebenenfalls mit 0x00-Bytes aufgefüllt.

Die auf die Associated Data folgenden Payloadbytes, werden ebenfalls als Bytestrom in Blöcke mit einer Länge von16 Bytes formatiert und mit 0x00 Bytes auf eine ganzzahlig durch 16 teilbare Länge aufgefüllt.

### 11.1.2 Formatierung und Generation der Counterblocks

Für den CTR-Modus zur Ver- und Entschlüsselung der Payload sind Zählerblöcke mit 16-Byte Länge notwendig. Diese werden wie folgt aus Flags, dem kryptographischen Nonce und einem fortlaufenden Zähler i erstellt:

| **Byte 0** | **Byte 1 bis 15-q** | **Byte 16-q bis 16** |
|---|---|---|
| Flags | Nonce | Zählerstand i |

Tabelle 13: Formatierung des i-ten Counterblocks

Das erste Byte des Counterblocks definiert die Flags die im Wesentlichen nur aus dem Feld Q_enc des Flag-bytes des Block0 bestehen.

| **Bit 7** | **Bit 6** | **Bit 5** | **Bit 4** | **Bit 3** | **Bit 2** | **Bit 1** | **Bit 0** |
|---|---|---|---|---|---|---|---|
| X | X | 0 | 0 | 0 | Q_enc[2] | Q_enc[1] | Q_enc[0] |

Tabelle 14: Flag-Byte für Counterblocks

Um Sicherheit zu gewährleisten besteht die Forderung, dass ein Counterblock nur ein einziges Mal zur Verschlüsselung einer Nachricht unter Verwendung eines bestimmten Schlüssels verwendet wird. Daraus ergibt sich eine Frage der Abwägung zwischen Länge des Nonce, wie oft eine CCM-Operation mit einem bestimmten Schlüssel benutzt werden kann und der Länge der einzelnen Nachrichten [NIST38c].

## 11.2 CCM-Core

Der AES-CCM-Core ist die Hardwareimplementierung des CCM-Algorithmus. Alle Module dieser Komponente befinden sich im Ordner `./hdl/aes_ccm/.` Die Module werden in die VHDL-Library `AES_CCM_lib` kompiliert. Zusätzlich sind Module der VHDL-Library `util_lib` und die Komponente `ECB_encrypt_only` aus der Bibliothek `aes_ecb_lib` nötig. Alle Typen sind im VHDL-Package `aes_ccm_pkg` definiert.

### 11.2.1 Interface

| Signal | Richtung | Typ |
|---|---|---|
| clk | IN | STD_LOGIC |
| res_n | IN | STD_LOGIC |
| data_in | IN | BYTE |
| data_stable | IN | STD_LOGIC |
| direction | IN | CRYPTODIRECTION |
| initialkey | IN | KEYBLOCK |
| key_stable | IN | STD_LOGIC |
| Ena_core | IN | STD_LOGIC |
| Want_data | OUT | STD_LOGIC |
| terminated | OUT | STD_LOGIC |
| Payload_len | OUT | Unsigned(2 downto 0) |
| Payload_result | Out | STATE |
| Rreq_payload_result | In | STD_LOGIC |
| No_payload_result | Out | STD_LOGIC |
| MAC_out | OUT | STATE |
| MAC_verified | OUT | STD_LOGIC |
| Mac_len | OUT | UNSIGNED(4 downto 0) |
| MAC_in | In | STATE |
| Mac_in_stable | IN | STD_LOGIC |

Tabelle 15: Interface AES-CCM-Core

- **clk**: Systemtakt des Security-Chip
- **res_n**: Reset Port des Chip ist active low, alle Datenregister werden bei res_n='0 gelöscht und alle Zustandsautomaten auf den Initialzustand gesetzt.
- **data_in**: Port für Datenbytes der Nachricht.
- **data_stable**: Signalisiert Gültigkeit der Daten an data_in. data_in wird in die jeweiligen Speicher CCM-Core übernommen, wenn data_stable='1' ist.
- **direction**: Wählt mit Hilfe der Protokollkonstanten MB_ENCRYPT und MB_DECRYPT den Ver- oder Entschlüsselungsmodus aus.
- **initialkey**: Benutzerschlüssel für Ver- und Entschlüsselung.
- **key_stable**: Signalisiert Gültigkeit des Benutzerschlüssels. Ist key_stable='1' werden aus dem Benutzerschlüssel an initialkey die Rundenschlüssel erzeugt. Nach elf Takten sind alle Schlüssel bereit. Solange bis ein neuer Schlüssel geladen werden

soll, muss `key_stable='1'` sein. Das Signal darf während einer Ver- oder Entschlüsselungsoperation nicht auf `key_stable='0'` wechseln. `Key_stable='0'` löscht alle Rundenschlüssel.

- **`Ena_core`**: Signalisiert dem AES-CCM-Core dass die Daten an data_in gelesen und verarbeitet werden sollen.
- **`Result_out`**: Ergebnis der Ver- oder Entschlüsselungsoperation für Payload. Ausgang des Result-FIFO.
- **`terminated`**: Wenn `terminated='1'` sind die Daten an `result_out` gültig und die Operation ist beendet.
- **`Want_data`**: Signalisiert mit `want_data='1',` dass die Nachricht noch nicht vollständig eingelesen wurde und weiter Bytes übertragen werden müssen.
- **`Mac_in`**: Der zu prüfende MAC der Nachricht – wird nur im Modus *decrypt-verify* benötigt. Es wird nur die im Block0 angegebene Anzahl von Bytes des MAC (t) geprüft. Der MAC wird am Ende der CCM-Operation geprüft.
- **`MAC_IN_Stable`**: Die Daten an MAC_in sind gültig und können in das MAC-Register geschrieben werden.
- **`Mac_verified`**: Gibt an, ob der MAC und die Nachricht im Falle von *decrypt-verify* gültig ist und die Nachricht somit unverändert ist.
- **`MAC_out`**: berechneter MAC der Nachricht. Lediglich `mac_len` des 16-Byte Mac sind gültig.
- **`Mac_len`**: Länge des MAC (*t*) in Bytes. Wird aus dem Feld `T_enc` des ersten Block berechnet.
- **`Payload_len`**: Länge der Payload in Bytes. Entspricht dem Feld Q des ersten Blocks.
- **`Rreq_payload`**: Leseanforderung für das Result-FIFO. Liest den nächsten 16-Byteblock der ver- bzw. entschlüsselten Payload aus dem FIFO.
- **`No_payload_result`**: Alle Blocks der ver- bzw. entschlüsselten Payload wurden aus dem Result-FIFO gelesen.
- **`Payload_result`**: 16-Byte Block der Payload, die länge der gültigen Bytes im Block muss über `Payload_len` berechnet werden.

### 11.2.2 Protokoll

Beim ersten Aufruf wird zuerst der Benutzerschlüssel geladen und die Rundenschlüssel werden erzeugt. `Initialkey` in den Keygenerator geladen, wenn `key_stable='1'.` `Key_stable='1'` muss während der gesamten Ver- oder Entschlüsselungsoperation gehalten werden. Sobald `key_stable='0'` ist werden die Rundenschlüssel gelöscht und ungültig und es kann ein anderer Schlüssel geladen werden. Die Generierung der Rundenschlüssel dauert elf Takte.

Danach wird die Nachricht in einem Bytestrom über `data_in` mit `data_in_stabe='1'` in den CCM-Core geschrieben. Der MAC kann im Falle des *decrypt-verification* Prozesses parallel dazu geschrieben werden. Auch das Lesen von Daten über `payload_result` ist davon unabhängig, da es sich um einen getrennten FIFO-Speicher handelt.

Mit `terminated='1'` wird das Ende der CCM-Operation angezeigt und die Daten können über den Port `payload_result` ausgelesen werden.

### 11.2.3 Blockschaltbild

Das Blockschaltbild

Abbildung 19 wurde vereinfacht dargestellt. Es sind nicht alle Steuerleitungen eingezeichnet.

Die Komponenten haben folgende Aufgaben:

- `Msg_header_analyzer` wertet die Informationen des Block0 aus und liefert die Daten an die einzelnen Komponenten. Hier wird der Nonce, die Payload Länge, die Länge des MAC und das Flag-Byte aus Tabelle 11 ausgewertet.

- `Adata_len_decoder` interpretiert die Länge der Associated Data und liefert die Länge and den Zustandsautomat `CCM-FSM`.

- `CCM-FSM` dient der Steuerung des Ablaufs des CCM-Algorithmus.

- `BuildBlock` liest Bytes über `data_in` ein und fügt sie zu 16-Byte-Blocks zusammen. Hier erfolgt gegebenenfalls Padding. Wenn es sich um Associated Data handelt, so muss lediglich der MAC daraus berechnet werden. Die Datenblocks werden dann in das `CBC-FIFO` geschrieben. Handelt es sich um Payload-Daten so müssen die Blocks ver- bzw entschlüsselt und der MAC berechnet werden. Die Daten werden dann in das Payload-FIFO geschrieben.

- Die `CTR-Mode-Komponente` ver- und entschlüsselt Payload-Datenblocks im CTR-Mode und gibt Plaintext-Datenblocks an das `CBC-FIFO` zur Berechnung des MAC.

- Die `CBC-Mode-Komponente` liest Datenblocks aus dem `CBC-FIFO` und berechnet inkrementell den MAC im CBC-Mode.

- Im Funktionsblock `BlockCompare` wird im Fall von *decrypt-verify* der errechnete MAC mit dem geladenen MAC verglichen.

Die Komponenten `Adata_len_decoder`, `Msg_header_analyzer` und `BuildBlock` arbeiten parallel und lesen bei `data_in_stable='1'` das Byte an `data_in` ein.

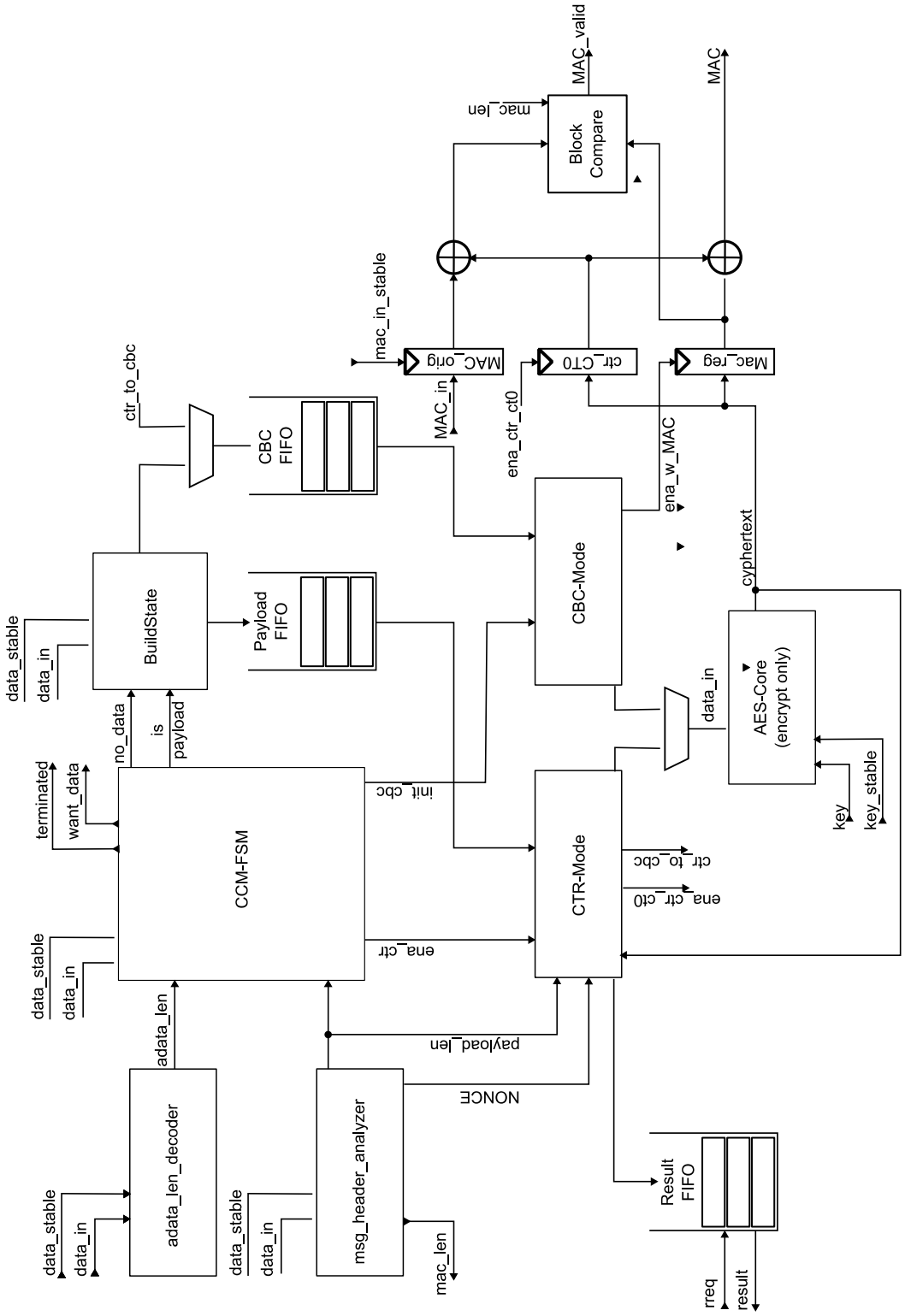


Abbildung 19: CCM-Core Blockschaltbild

### 11.2.4 FPGA-Ressourcen

Die Ressourcen und Timingangaben der Altera FPGAs wurden mit „Mentor Graphics Precision RTL 2007“ ermittelt und sind lediglich eine Abschätzung.

| FPGA | LUTs/LCs | Register | Memory |
|---|---|---|---|
| Cyclone II EP2C70F896C | 5010 (7,3%) | 2023 (2,9%) | 135168 (11%) |

Tabelle 16: FPGA Ressourcen (AES-CCM)

Die maximale Betriebsfrequenz liegt bei 71.6 MHz, bzw.14 ns Register-Register-Delay.

## 11.3 CCMCoreController

Der AES-CCM-CoreController bindet den AES-CCM-Core über den Systembus an den SystemController, vgl. Kapitel 9 und das SPI-Slave-Module, vgl. Kapitel 8 an. In diesem Modul werden die Protokollinformationen ausgewertet und die Signale der SPI-Schnittstelle in das entsprechende Datenformat des Interface des AES-CCM-Core umgesetzt.

### 11.3.1 Interface

Die Schnittstelle des `CCMCoreControllers` setzt sich aus dem Systembus „`sysconnect`“ vom Typ `System_IF` aus Kapitel [9.2] zum SystemController, vgl. Kapitel 9 und der Schnittstelle des AES-CCM-Core [10.1.1] zusammen.

### 11.3.2 Protokoll

Zu Beginn befindet sich der Protokollzustandsautomat im Zustand Setup. Mit den Schlüsselsymbolen `MB_DECRYPT` kann der *Decrypt-Verification* Prozess, bzw. mit `MB_ENCRYPT` der *encrypt-generate* Prozess ausgewählt werden. Ist noch kein Schlüssel im AES-Core der CCM-Core-Komponente geladen, muss mit `MB_LOADKEY` der Automat in den Zustand `LOADKEY` überführt werden. Die darauf folgenden 16 Bytes werden dann als Schlüssel interpretiert und die Rundenschlüssel werden daraus generiert.

Im Falle von *Decrypt-Verification* muss der MAC, der zu prüfenden Nachricht geladen werden. Nach dem Empfang von MB_LOADMAC geht der Zustandsautomat in den Zustand LOADMAC über. Es müssen 16 Bytes empfangen werden, da der CCMCoreController zu diesem Zeitpunkt noch nicht entscheiden kann, welche Länge der MAC der Nachricht tatsächlich hat. Zur Vereinfachung des Protokolls wurde auf ein zusätzliches Senden dieser Information verzichtet. Von den empfangenen 16 Bytes des MAC werden jedoch später nur so viele Bytes geprüft wie in Byte0 der Nachricht angegeben sind. Die übrigen Bytes werden ignoriert.

Nach dem Empfang des Schlüsselsymbols `MB_LOADDATA` verweilt der Automat solange im Zustand `LOADMSG` bis alle Bytes der Nachricht empfangen wurden. Dies wird von der CCM-Core-Komponente über `want_data='0'` signalisiert. Schon während dem Empfang wird in

der CCM-Core-Komponente mit der Verarbeitung begonnen. Dies ist möglich, da genügend Speicher zur Verfügung steht und der MAC inkrementell berechnet werden kann.

Sind alle Daten empfangen geht der Protokollzustandsautomat in den Zustand `WORKING` über. Hier wird auf das Ende der Verarbeitung im Core gewartet.

Nachdem die CCM-Core-Komponente mit `terminated='1'` signalisiert, dass die kryptographische Verarbeitung abgeschlossen ist, befindet sich der Automat im Zustand `POLLSTATE`, in dem wieder Schlüsselsymbole ausgewertet werden.

Aus dem Zustand `POLLSTATE` kann mit `MB_POLL` die tatsächliche Payload-Länge abgefragt werden. Es werden zwei Bytes zurückgegeben, da eine maximale Länge von $2^{16}$ für Payload implementiert wurde. Das heißt der Automat befindet sich für die Dauer von 2 SPI-Zyklen im Zustand `POLL_PAYLOAD_LEN`.

Das Symbol MB_POLLMAC führt im Zustand POLLSTATE den Automaten für einen SPI-Zyklus in den Zustand POLL_MACLEN über, in dem die tatsächliche Länge des MAC ausgegeben wird.

Mit dem Symbol `MB_POLLMACVALID` kann im Fall des *decrypt-verify* Prozesses abgefragt werden, ob der MAC und somit auch die Nachricht gültig ist. Die Komponente sendet entweder `MB_MACINVALID` oder `MB_MACVALID` zurück. Ist der MAC ungültig, so darf laut Spezifikation weder der ungültige MAC noch die Entschlüsselte Payload zurückgegeben werden. Um den Automat in den Setup-Zustand zu überführen, muss dennoch die Payload und der MAC ausgelesen werden. In diesem Fall bestehen die zu zurückgegebene Payload und der MAC aus `MB_MACINVALID` Symbolen.

Nach Empfang von `MB_SENDMAC` geht der Zustandsautomat in den Zustand `SENDMAC` über hier werden die einzelnen Bytes des MAC oder `MB_MACINVALID` gesendet.

Wenn alle Bytes des MAC übertragen sind, wird im Zustand `WAIT_SENDPAYLOAD` auf das Symbol `MB_RESULT` gewartet, welches die Übertragung der ver- bzw. entschlüsselten Payload startet. Der Automat verweilt solange im Zustand `SENDRESULT` bis *Payload_len* Bytes gesendet wurden.

Abschließend geht der Automat über den Zustand `DONE`, in dem er einen Takt lang verweilt und über `sysconnect.finished='1'` den Kontrollfluss an den SystemController zurückgibt, in den Initialzustand `SETUP` über.

Nachfolgend ist der Protokollzustandsautomat dargestellt. Alle Transitionen in denen Daten empfangen werden sind synchron mit dem Signal `sysconnect.io_ack='1'`.

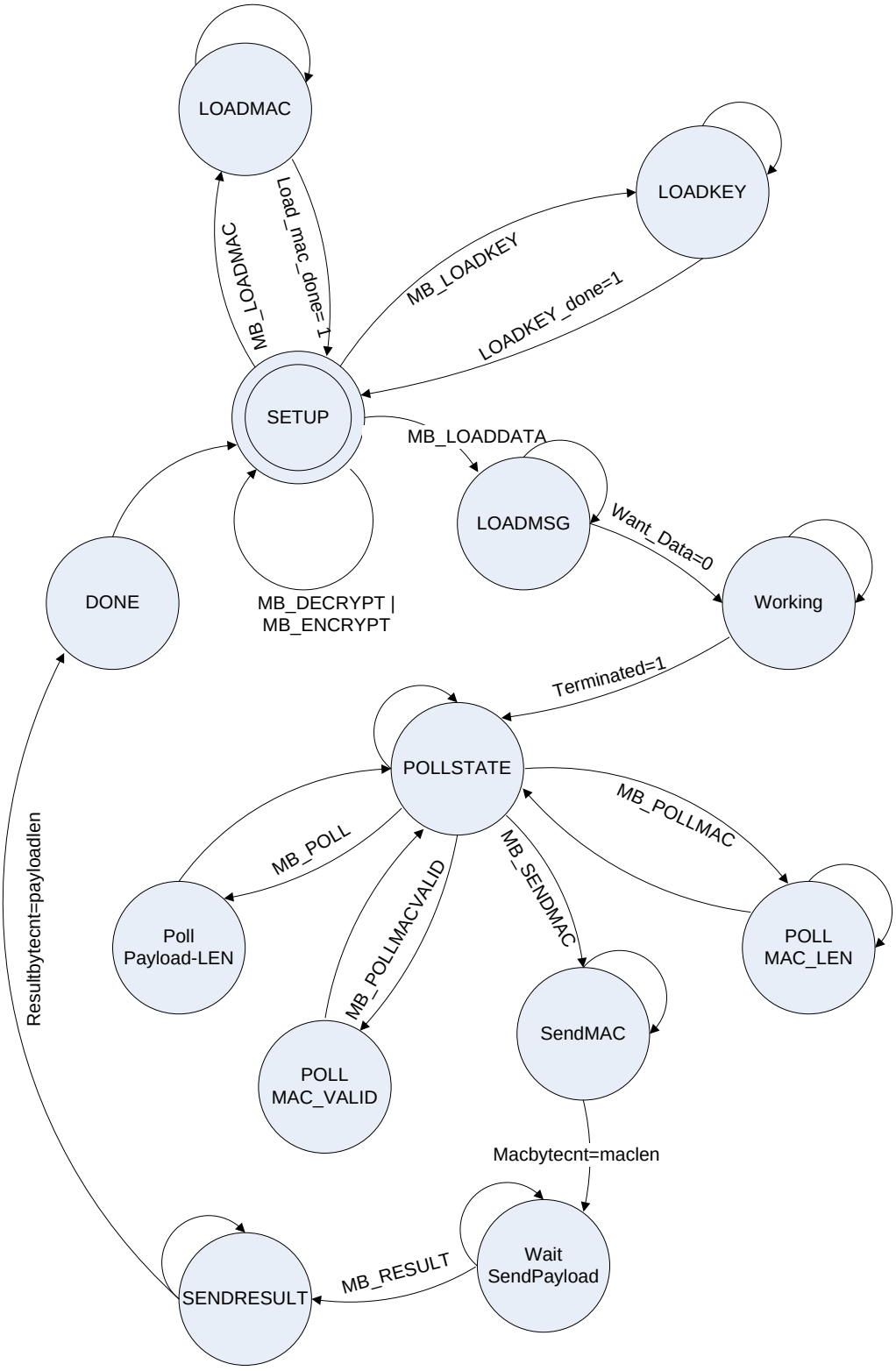


Abbildung 20: Protokollzustandsautomat CCMCoreController

# 12 RSA-Komponente

Die RSA-Komponente besteht aus dem RSA-Core und dem dazugehörigen RSACoreController. Im RSA-Core wird die dem RSA-Algorithmus zugrundeliegende Modulo-Potenzierung durchgeführt.Die beiden Funktionen zur Ver-, bzw. Entschlüsselung unterscheiden sich nur durch den Exponenten. Daher braucht nur ein Datenpfad realisiert werden, der sowohl die Ver- als auch Entschlüsselung berechnen kann.

## 12.1 RSA-Core Komponente

Der RSA-Core ist die Hardwareimplementierung des RSA Algorithmus mit einer Modulolänge von 1024 Bit. Alle Module dieser Komponente befinden sich im Ordner `./hdl/rsa/` die Module werden in die VHDL-Library `rsa_lib` kompiliert. Zusätzlich sind Module der VHDL-Library `util_lib` nötig. Alle Typen sind im VHDL-Package `rsa_pkg` definiert.

Das zentrale Element des RSA-Core ist ein Zustandsautomat der die Modulo-Potenzierung über den Algorithmus von Montgomery steuert. Die Montgomery-Multiplikation wird in einem Systolischen Array mit einer Zeile bestehend aus 129 Zellen berechnet.

### 12.1.1 Interface

| Signal | Richtung | Typ |
|---|---|---|
| clk | IN | STD_LOGIC |
| res_n | IN | STD_LOGIC |
| Start | IN | STD_LOGIC |
| Terminated | Out | STD_LOGIC |
| Waddr_res_modulo | IN | STD_LOGIC |
| Waddr_res_factor | IN | STD_LOGIC |
| Waddr_res_exp | IN | STD_LOGIC |
| Result_reset | IN | STD_LOGIC |
| M_byte | IN | BYTE |
| Wreq_m | IN | STD_LOGIC |
| Factor_byte | IN | BYTE |
| Wreq_factor | IN | STD_LOGIC |
| Msg_byte | IN | BYTE |
| Wreq_msg | IN | STD_LOGIC |
| Exp_byte | IN | BYTE |
| Wreq_exp | IN | STD_LOGIC |
| Rreq_result | IN | STD_LOGIC |
| Result_byte | Out | BYTE |
| No_result | Out | STD_LOGIC |

Tabelle 17: Interface RSA-Core

- **clk**: Systemtakt des Security-Chip
- **res_n**: Reset Port des Chip ist active low, alle Datenregister werden bei `res_n='0'` gelöscht und alle Zustandsautomaten auf den Initialzustand gesetzt.
- **start**: active high, startet die Verarbeitung im RSA-Core
- **terminated**: active high, RSA-Operation ist abgeschlossen und Ergebnis liegt im Result-FIFO, kann durch `rreq_result` ausgelesen werden bis `no_result='1'.`
- **waddr_res_modulo:** active high, Setzt die Schreibaddresse für den Modulo auf 0 zurück. Nötig vor dem Laden eines neuen Modulus.
- **waddr_res_factor:** active high, Setzt die Schreibadresse für den Faktor $R^2$ mod $M$ auf 0 zurück. Nötig vor dem Laden eines neuen Faktors.
- **waddr_res_exp:** active high, Setzt die Schreibadresse für den Exponenten auf 0 zurück. Nötig vor dem Laden eines neuen Exponenten.
- **result_reset:** Setzt die Leseadresse des Result-FIFO auf 0 zurück.
- **M_byte:** Byte des Modulo wird bei `wreq_m='1'` in den Speicher für den Modulus geschrieben. **Der Modulus muß Least-Significant-Byte-first geschrieben werden,** wobei die Bits in einem Byte MSB-LSB angeordnet sind. Die Zahl 0xA1B2C3 wird als 0xC3, 0xB2, 0xA1 in den Speicher geschrieben.
- **wreq_m:** Schreiben des Byte an M_byte in den Speicher.
- **factor_byte:** Byte des Faktors $R^2$ mod $M$ wird bei `wreq_factor ='1'` in den Speicher für den Faktor geschrieben. **Der Faktor muß Least-Significant-Byte-first geschrieben werden,** wobei die Bits in einem Byte MSB-LSB angeordnet sind. Die Zahl 0xA1B2C3 wird als 0xC3, 0xB2, 0xA1 in den Speicher geschrieben.
- **wreq_factor:** Schreiben des Byte an factor_byte in den Speicher.
- **exp_byte:** Byte des Exponenten wird bei `wreq_exp ='1'` in den Speicher für den Faktor geschrieben. **Der Exponent muß Least-Significant-Byte-first geschrieben werden,** wobei die Bits in einem Byte MSB-LSB angeordnet sind. Die Zahl 0xA1B2C3 wird als 0xC3, 0xB2, 0xA1 in den Speicher geschrieben.
- **wreq_exp:** Schreiben des Byte an `exp_byte` in den Speicher.
- **Result_byte:** Byte des Ergebnisses, das Ergebnis wird bei `rreq_result='1'` in Bytes zurückgegeben, beginnend mit dem niederwertigsten Byte.

- **`rreq_result:`** Nächstes Byte des Ergebnisses aus dem FIFO lesen.
- **`no_result:`** bei `no_result='1'` stehen keine Bytes des Ergebnisses im FIFO und `result_byte` ist ungültig.

### 12.1.2 Protokoll

Bei jedem Laden eines neuen Schlüssels müssen die Schreibadressen reinitialisiert werden. Zunächst sind `waddr_res_factor`, `waddr_res_exponent`, und `waddr_res_modulo` für einen Takt `,1'` sein. Danach können die Parameter in beliebiger Reihenfolge in die FIFOs geladen werden. Die Parameter müssen allerdings *Least-Significant-Byte-first* übertragen werden.

Sind die Schlüsselparameter geladen kann die Nachricht über `msg_byte` mit `wreq_msg='1'` in den Nachrichtenspeicher geschrieben. Mit `start='1'` wird die Verarbeitung gestartet. Bei `terminated='1'` ist die RSA-Operation abgeschlossen und das Ergebnis kann an `result_byte` mit `rreq_result='1'` ausgelesen werden. Ist `no_result='1'` sind alle Bytes ausgelesen.

### 12.1.3 Aufbau des RSA-Core

#### 12.1.3.1 Zellen des Array

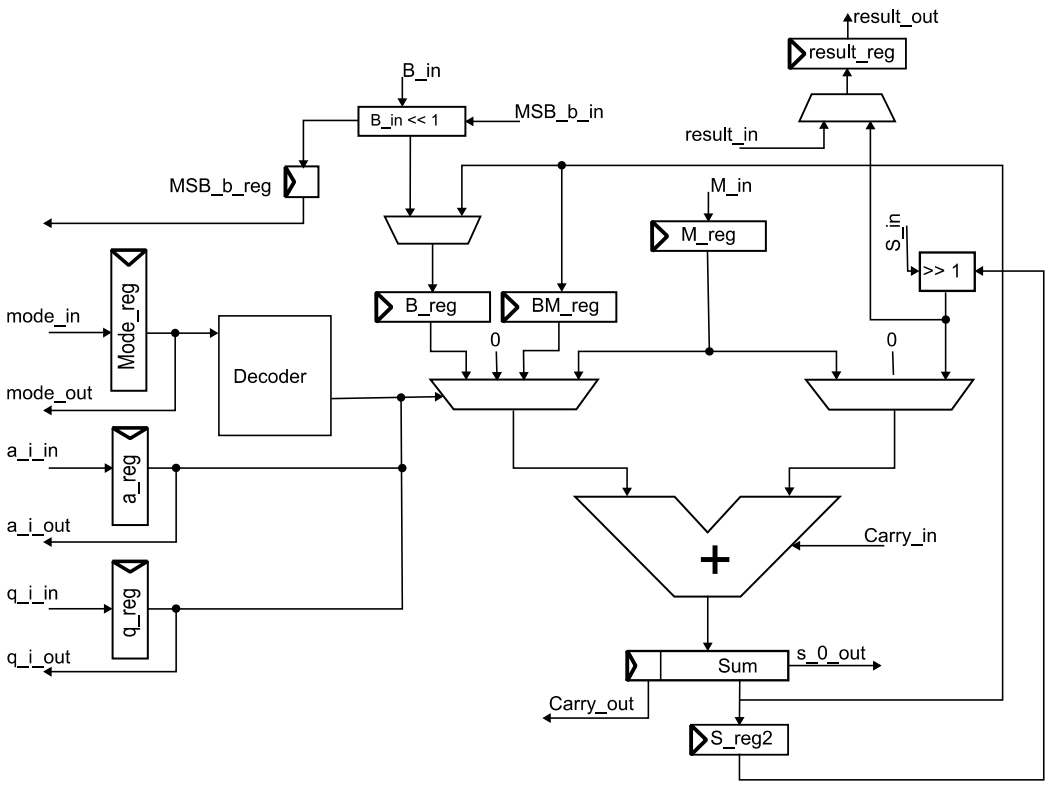


Abbildung 21: Zelle des Systolischen Array

Das Systolische Array wird aus gleichartigen Zellen aufgebaut. Die hier vorgestellte Implementierung ist ähnlich mit der aus [Bl99] und [HSH00]. Dieses Zellelement führt die dem Algorithmus zugrundeliegende Addition und Rechtsschiebeoperation auf eine feste Anzahl von Bits durch. In dieser Implementierung wurden blockbreiten von 8 Bit gewählt. Es sind auch andere Blockgrössen denkbar, in [Bl99] wird die Effizienz in Abhängigkeit der Blockgrößen verglichen. Jede Zelle ist für sich eine abgeschlossene Einheit, alle Steuersignale und Daten werden durch ein Register geführt, um die Länge der kombinatorischen Pfade gering und die Synthese-Frequenz entsprechend hoch zu halten. Jede Zelle berechnet einen Block von acht Bits des Ergebnisses, dafür wird folgende Operation durchgeführt.

$$S_{i,j} = \frac{Carry_{i,j-1} + S_{i-1,j+1}[0], S_{i-1,j}[7:1] + a_i B_j + q_i M_j}{2}$$

Formel 15: Summation in Systolic_cell

*I* gibt den Index der Iteration und *j* den Zellindex angibt. Die erste Zelle berechnet zusätzlich das niederwertigste Bit des Zwischenergebnisses $q_i$. Um den Addierer möglichst einfach zu halten wird ein Volladdierer verwendet, der lediglich das letzte Zwischenergebnis, den Übertrag, Carry, der vorherigen Zelle *j-1* und einen zweiten Summanden addiert. Bevor die eigentliche Montgomery-Multiplikation beginnt, muss daher der zweite Summand berechnet werden. Er ist für eine Montgomery-Operation konstant. Zunächst werden dazu 8-Bit Blöcke des Faktors *B*, der durch linksschieben mit 2 multipliziert wird, in das Register *B_reg* geladen und das höchstwertige Bit wird an die nächste Zelle *j+1* nach links weitergereicht. Zu *B* wird ein 8-Bit Block des Modulo *M* aus `M_reg` und gegebenenfalls der Carry der *B*+*M*-Operation der linken Zelle addiert. Das Ergebnis *B*+*M* wird in `BM_reg` abgelegt. Alle für die Addition nötigen Parameter sind nun in der Zelle gespeichert.

Wie aus Formel 15 abzulesen, ist der Summand, der zu $S_{i-1}$ addiert wird, abhängig von $q_i$ und $a_i$. Der Summand ist entweder 0, *B*, *M* oder *B*+*M*. Der erste Multiplexer wählt daher mit $q_i$ und $a_i$ den Wert des Summanden.

Der Decoder hat die Aufgabe über das Kontrollwort die Zelle zu initialisieren, beispielsweise den Multiplexer so zu schalten, dass *B*+*M* berechnet und gespeichert wird oder das Ergebnis des Addierers auf den Ausgang `result_out` geschaltet wird.

Die Rückführung der Summe `sreg` auf das Register `b_reg` ist eine Optimierung des Ablaufes, da in Formel 9 $Z_{i+1}$ als Faktor für die nachfolgende Iteration benötig wird.

#### 12.1.3.2 Aufbau des Systolischen Array

Der Array besteht aus 129 Zellen. Die Zellen sind so miteinander verbunden, dass das Carry-Bit zusammen mit den Steuerinformationen aus dem `mode` und den Bits $a_i$ und $q_i$ nach links von Zelle 0 zu Zelle 129 wandert. Das LSB der Summe wandert, wie auch das Endergebnis von rechts nach links. Zelle 129 rechnet nur mit den Überträgen, da alle Daten des Systems maximal 128 Byte lang sein. Am Eingang `B_in` liegt 0 an, ebenso am Eingang `M_in` für den Modulus.

Das Register B_REG der 129. Zelle wird mit dem 128. Bit von B über `msb_b_in` gefüllt. In `BM_REG` der Zelle 128 kann also maximal 2 stehen (`B_REG + CARRY`), mit einer Zellgröße von 8-Bit wird in der letzten Zelle daher nie ein Übertrag generiert. Bei allen übrigen Zellen wird das jeweilige Byte von M und B gespeichert und die Summe daraus gebildet.

Für die Berechnung eines 8-Bit Blocks in einer Zelle j im Schleifenschritt i ist das LSB des höherwertigeren Blocks aus Zelle *j*+1 der vorherigen Iteration *i*-1nötig. Dieses Bit wird in aber erst in Zelle *j*+1 im Takt *i* berechnet, die den Übertrag aus Zelle j des Taktes i-1 verarbeitet, dadurch entsteht ein Takt Wartezeit in Zelle j. In diesem Takt kann jedoch bereits das zweite Teilprodukt *P* aus dem Algorithmus in Formel 9 berechnet werden.

Nach 2n+1 Takten liegt das Ergebnis der ersten Operation am Ausgang result_out der Zelle 0 an. Dafür muss ein Kontrollwort im 2n Takt in mode_in des Array geschrieben werden welches den Multiplexer an `result_out` umschaltet. Gleichzeitig wird dieses Teilprodukt ($Z_i$) in B_Reg gespeichert. Einen Takt später, liegt das erste Byte des zweiten Teilproduktes ($P_i$) an `result_out` an. Nach 128 Takten sind alle Datenbytes aus dem Systolischen Array ausgelesen und die nächste Multiplikation kann beginnen.

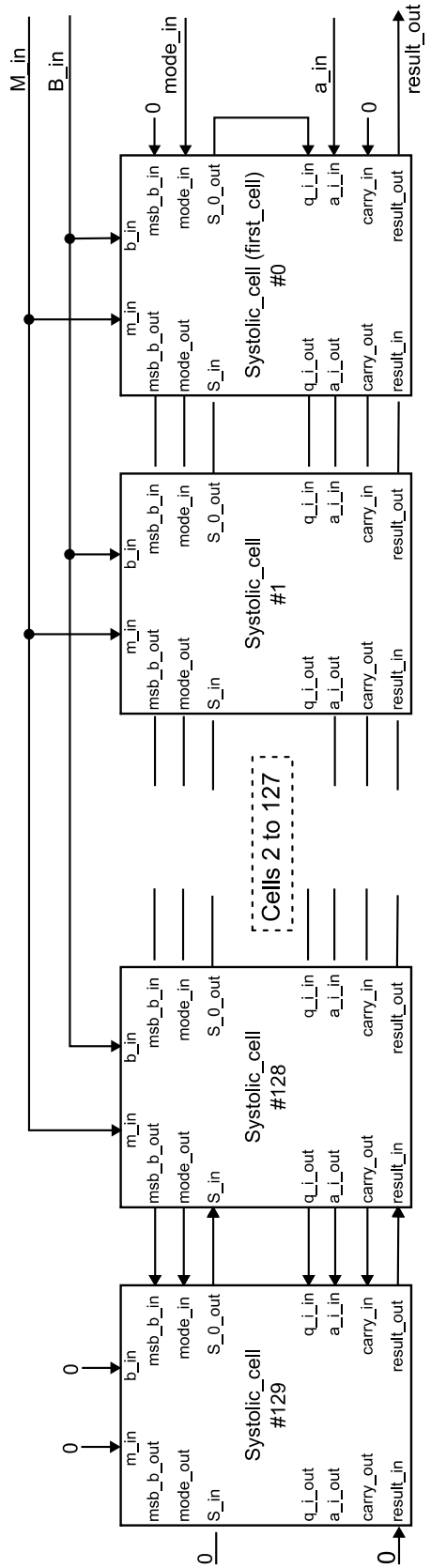


Abbildung 22: Aufbau des Systolischen Arrays

### 12.1.4 RSA-Core Funktionsbeschreibung

Nachdem die RSA-Operation mit `start='1'` aktiviert wurde wird zuerst ein Kontrollwort über `mode_in` an das Systolischen Array übergeben welches Zelle 0 auswertet, wo dann Bytes des Faktor R2modM über `B_in` und Bytes des Modulo über `M_in` geladen werden. Der Zustandsautomat setzt `rreq='1'`, so dass im nächsten Takt das zweite Byte aus dem Modulo FIFO und dem Factor-FIFO an die Leitungen `B_in` und `M_in` anliegen, solange bis alle Bytes des Modulo und des Faktors in das Systolische Array übertragen wurden. Im zweiten Takt ist das Ladekontrollwort an Zelle 1. Der Zelle 0 wird vom Zustandsautomaten ein Kontrollwort gegeben, um die Summe B+M zu errechnen. In Takt 3 lädt Zelle 2 die Daten über `B_in` und `M_in`, Zelle 1 addiert und Zelle 0 speichert das Ergebnis in `BM_REG.` In den folgenden Takten werden dann abwechslungsweise Bits der Nachricht aus dem Z-Ram bzw. $P_0 = 1$ in das systolische Array übertragen. Diese Parallelisierung der beiden Multiplikationen ist möglich, da mit der Berechnung des nächsten Iterationsschrittes des ersten Produktes einen Takt auf die Berechnung in der nachfolgenden Zelle gewartet werden muss, bis die Zelle j+1 das LSB liefert und beide Produkte einen gemeinsamen Faktor aus `B_REG` besitzen. Nach 2n+1 Takten sind die Werte $Z_1$ bzw. $P_1$ berechnet und werden mit einem Kontrollwort auf den Ausgang `result_out` der jeweiligen Zelle gelegt. Diese Werte werden dann Byteweise in das Z-Ram geschrieben.

In Tabelle 18 sind beispielhaft die ersten sechs Takte der Berechnung von $Z_i$ und $P_i$ dargestellt. Die Indizes $Z_{k,l}$ sind folgendermaßen zu verstehen: k entspricht der Schleifenvariablen i des Algorithmus aus Formel 8, während „l“ den l-ten 8-Bit Block des Zwischenergebnisses beschreibt.

| **Takt** | **Zelle 0** | **Zelle 1** | **Zelle 2** |
|---|---|---|---|
| 1 | Lade B, M | - | - |
| 2 | B+M | Lade B, M | - |
| 3 | Schreibe BM_reg | B+M | Lade B, M |
| 4 | $Z_{1,0}$ | Schreibe BM_reg | B+M |
| 5 | $P_{1,0}$ | $Z_{1,1}$ | Schreibe BM_reg |
| 6 | $Z_{2,0}$ | $P_{1,0}$ | $Z_{1,2}$ |

Tabelle 18: Beispiel Paralleler Ablauf in Systolischem Array

In den Folgeiterationen des Algorithmus Formel 9 werden die Werte aus P-RAM und Z-RAM wieder in das Systolische Array geschrieben. Je nach Wertigkeit des Exponenten-Bits wird der Wert $P_i$ in P-RAM geschrieben oder verworfen. Der Wert von $Z_i$ wird zusätzlich durch ein Kontrollwort für die nächste Iteration in den Zellen in `B_reg` gespeichert. Dies ist möglich, da $Z_i$ sowohl für die Berechnung von $Z_{i+1}$ als auch $P_{i+1}$ nötig ist.

Der Zustandsautomat wiederholt diesen Vorgang solange, bis alle Bits des Exponenten abgearbeitet sind. Als abschließende Korrekturberechnung muss das vorläufige Ergebnis im P-RAM noch mit „1“ multipliziert werden. Dafür muss zunächst „1“ über die Leitung `B_in` in die Zellen gespeichert werden, danach wird P aus dem P-RAM in das systolische Array geschrieben. Das Endergebnis wird im Result-FIFO abgelegt. Da hier kein anderer Wert parallel berechnet wird ist nur jedes zweite Byte aus dem Array gültig.

In Abbildung 23 ist der Aufbau des RSA-Cores dargestellt. Zur Vereinfachung wurden die Steuerleitungen für die FIFOs vernachlässigt.

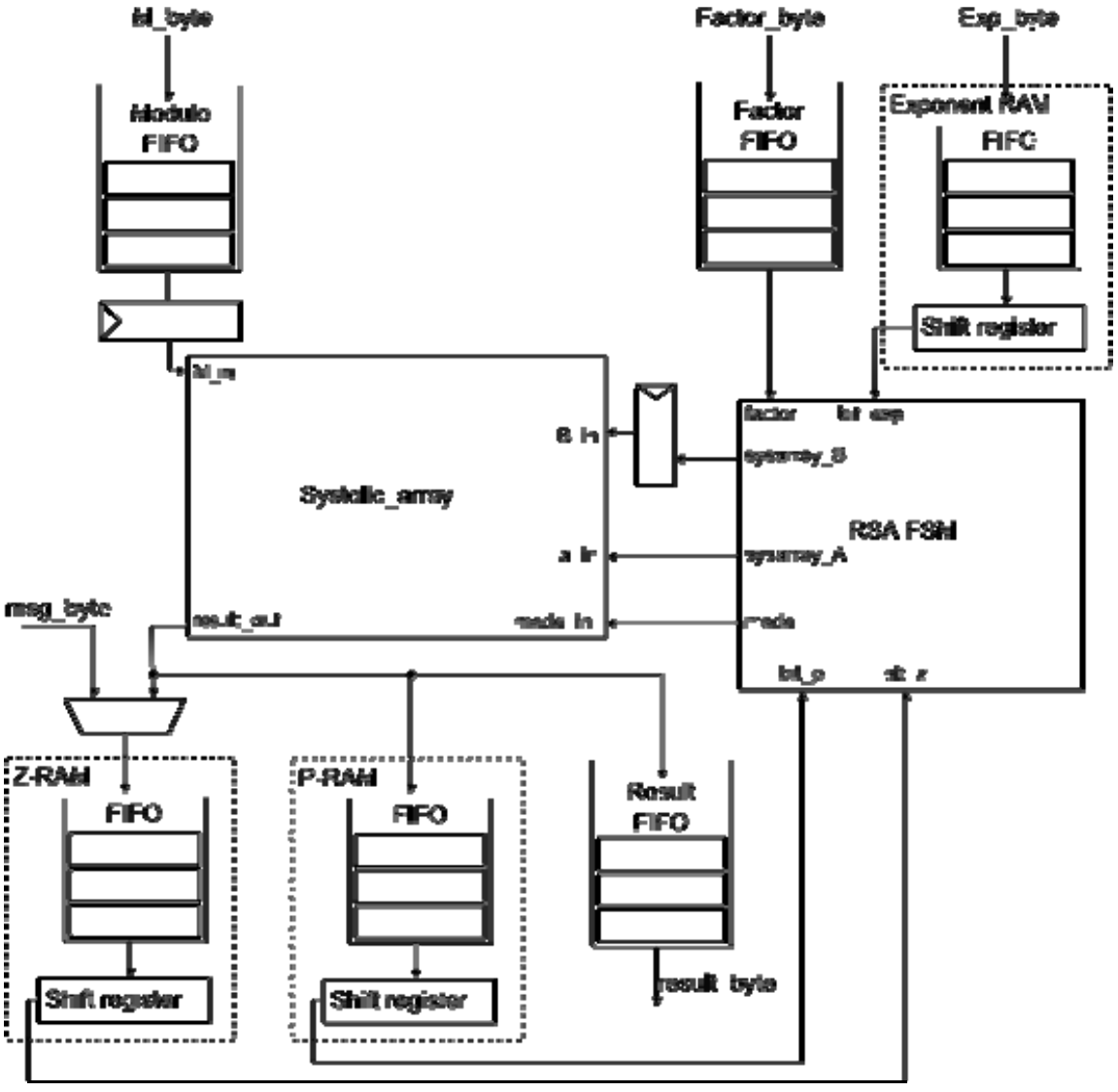


Abbildung 23: RSA-Core

### 12.1.5 FPGA-Ressourcen

Die Ressourcen und Timingangaben der Altera FPGAs wurden mit „Mentor Graphics Precision RTL 2007“ ermittelt und sind lediglich eine Abschätzung.

| FPGA | LUTs/LCs | Register | Memory |
|---|---|---|---|
| Cyclone II EP2C70F896C | 9264(13,5%) | 7394(10,8%) | 12288 (1%) |

Tabelle 19: FPGA Ressourcen (RSA)

Die maximale Betriebsfrequenz liegt bei 95,4 MHz, 10,47 ns Register-Register Delay.

## 12.2 RSACoreController

Der RSACoreController bindet den RSA-Core über den Systembus an den SystemController, vgl. Kapitel 9 und das SPI-Slave-Module, vgl. Kapitel 8 an. In diesem Modul werden die Protokollinformationen ausgewertet und die Datenbytes der SPI-Schnittstelle in das entsprechende Datenformat des Interface des RSA-Core umgesetzt.

### 12.2.1 Interface

Die Schnittstelle des `RSACoreControllers` setzt sich aus dem Systembus „`sysconnect`“ vom Typ `System_IF` aus Kapitel [9.2] zum SystemController und der Schnittstelle des RSA-Core asu Kapitel [12.1] zusammen.

### 12.2.2 Protokoll

Nur wenn `sysconnect.ena='1'` ist, arbeitet die Komponente, sonst sind die Speicher der Komponente angehalten und sie verharrt in ihrem Zustand. Zu beginn und nach jeder Operation befindet sich der Zustandsautomat im Zustand `SETUP` Vor einer RSA-Operation müssen zunächst die Parameter `Modulo, Key` und `Factor (R2ModM)`geladen werden. Dies geschieht immer nach dem gleichen Ablauf. mit dem entsprechenden Schlüsselsymbol (`MB_MODULO, MB_LOADKEY, MB_FACTOR`) des Protokolls geht der Automat in einen Zustand (`MODLEN, KEYLEN, FACTORLEN)` über, in dem ein Byte erwartet wird das die Länge der nachfolgenden Daten codiert. Ferner wird mit dem Empfang eines solchen Schlüsselsymbols, mit Hilfe des Signals `waddr_res{exp, modulo, factor},` die Schreibadresse des jeweiligen Speichers reinitialisiert. Wurde das Längenbyte empfangen erwartet der Zustandsautomat entsprechend viele nachfolgende Datenbytes, die auf die jeweiligen Ports des RSA-Core geschrieben werden. Ist die Transaktion eines Datentyps abgeschlossen, kehrt der Zustandsautomat in den `SETUP`-Zustand zurück.

Alle Daten, die Schlüsselparameter *R2ModM*, Exponent und Modulo, sowie die Nachricht müssen *Least-Significant-Byte-first* übertragen werden. Das niederwertigste Byte wird als erstes übertragen wobei die einzelnen Bytes im Little-Endian-Format interpretiert werden.

Sind die Schlüsselparameter geladen, können Nachrichtendaten geladen werden. Dazu wird der Automat mit MB_LOADDATA in den Zustand MSGLEN gebracht, in welchem die Länge der nachfolgenden Nachricht in Bytes erwartet wird. Nach Empfang des Längenbytes msgsize werden die nachfolgenden Bytes als Nachricht behandelt und auf den Port msg_byte des RSA-Core geschrieben.

Mit dem Schlüsselsymbol MB_START geht der Zustandsautomat in den Zustand WORKING über, in dem er verweilt bis die Komponente RSA-Core über den Port terminated='1' signalisiert, dass die RSA-Operation abgeschlossen ist und die Daten ausgelesen werden können. Im Zustand WORKING werden alle empfangenen Bytes ignoriert. Danach befindet sich der Automat wieder im Ausgangszustand SETUP. Mit dem Schlüsselsymbol MB_POLL kann jederzeit abgefragt werden, ob Ergebnisdaten vorliegen. Befindet sich der Automat im Zustand WORKING wird automatisch 0 zurückgegeben sonst die Modulolänge modsize.

Das Auslesen des Ergebnisses aus dem Result-FIFO des RSA-Core erfolgt mit dem Schlüsselsymbol MB_RESULT. Der Automat wird dadurch in den Zustand SENDRESULT versetzt in dem er verweilt bis die Komponente RSA-Core über no_result='1' signalisiert dass das FIFO leer ist. Die Nachrichtenlänge in Bytes ist durch die Länge des Modulus bestimmt. Maximal werden jedoch 128 Bytes übertragen. Sind alle Bytes übertragen wird im Zustand DONE dem SystemController aus Kapitel 9 signalisiert, dass die RSA-Operation abgeschlossen ist. Danach muss mit MB_RSA die RSA-Komponente wieder aktiviert werden. Die Schlüsselparameter bleiben jedoch gespeichert, das heißt es können sofort wieder neue Nachrichtendaten geladen werden, oder aber es werden, wie oben beschrieben, neue RSA-Schlüsselparameter geladen.

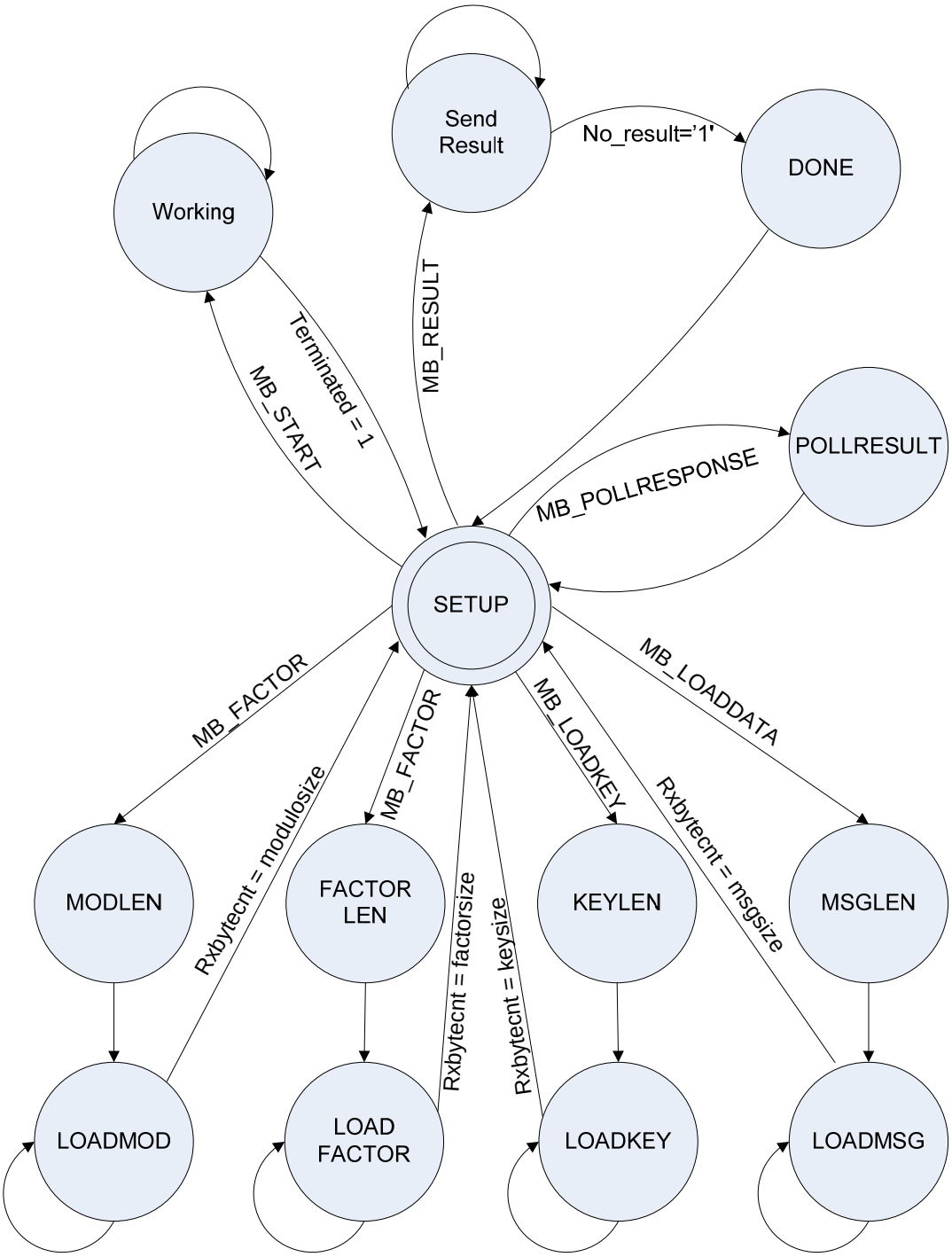


Abbildung 24: Zustandsautomat RSACoreController

Alle Übergänge im Zustandsautomat werden durch `io_ack` des `System_IF` aus Kapitel 9.2 ausgelöst, unbeschriftete Übergänge bedeuten `sysconnect.io_ack='1'`.

# 13 MD5-Komponente

Die MD5-Komponente besteht aus dem MD5-Core und dem dazugehörigen MD5CoreController. In dieser Komponente wird eine MD5-Prüfsumme über die empfangenen Datenbytes berechnet. Die Komponente arbeitet mit Bytes als Eingabegröße und gibt die 128 Bit große Prüfsumme ebenfalls byteweise zurück. Da im Falle der SPI-Kommunikation immer mit *In-Band-Signalling* gearbeitet wird, existiert keine Möglichkeit während der Nachrichtenübertragung das Ende der Nachricht anzuzeigen. Die Länge der nachfolgenden Nachricht in Bytes muss der Komponente also vor der Nachricht übermittelt werden. Da der Security-Chip für die Verwendung in eingebetteten Feldgeräten entwickelt wurde, ist entgegen der Spezifikation in [RFC1321], die Nachrichtenlänge hier auf $2^{32}$ Bytes begrenzt, da dies für die Verwendung auf der Feldebene ausreichend betrachtet wird.

## 13.1 MD5-Core

Der MD5-Core ist die Hardwareimplementierung des MD5-Algorithmus. Alle Module dieser Komponente befinden sich im Ordner./`hdl/md5/` die Module werden in die VHDL-Library `md5_lib` kompiliert. Zusätzlich sind Module der VHDL-Library `util_lib` nötig. Alle Typen sind im VHDL-Package `md5_pkg` definiert.

### 13.1.1 Interface

| Signal | Richtung | Typ |
|---|---|---|
| clk | IN | STD_LOGIC |
| res_n | IN | STD_LOGIC |
| msg_byte | IN | BYTE |
| msg_byte_stable | IN | STD_LOGIC |
| msg_len | IN | ByteArray(3 downto 0) |
| lastbyte | IN | STD_LOGIC |
| ena | IN | STD_LOGIC |
| init | IN | STD_LOGIC |
| Digest | OUT | MD5DIGEST |
| done | OUT | STD_LOGIC |

Tabelle 20: Interface MD5-Core

- **`clk`**: Systemtakt des Security-Chip
- **`res_n`**: Reset Port des Chip ist active low, alle Datenregister werden bei `res_n='0` gelöscht und alle Zustandsautomaten auf den Initialzustand gesetzt.
- **`msg_byte`**: Port für Bytes der Nachricht.

- **msg_byte_stable**: signalisiert Gültigkeit der Daten an msg_byte.
- **msg_len:** Länge der Nachricht in Bytes. 32-Bit-Signal, d. h. die maximale Nachrichtenlänge, über die eine Prüfsumme berechnet werden kann ist auf $2^{32}$ Bytes begrenzt.
- **last_byte:** Mit last_byte='1' wird angegeben, dass es sich um das letzte Byte der Nachricht handelt und die Verarbeitung der Nachricht abgeschlossen. Die Komponente selbst beinhaltet keinen Zähler für die Anzahl der verarbeiteten Bytes.
- **ena:** Aktiviert den MD5-Core mit ena='0' wird die Verarbeitung angehalten.
- **init:** Initialisiert den MD5-Core für die Verarbeitung einer neuen Nachricht und startet die Verarbeitung, wenn ena='1' ist.
- **digest:** Prüfsumme der Nachricht, wird nach der Verarbeitung von jeweils 64-Bytes auf das Zwischenergebnis aktualisiert.
- **done:** Mit done='1' signalisiert die MD5-Core-Komponente das Ende der Verarbeitung einer Nachricht. Die Daten an digest entsprechen nun dem endgültigen MD5-Hash der Nachricht.

### 13.1.2 Protokoll

Bei init='1' wird die Verarbeitung einer Nachricht gestartet, mit ena='0' kann der MD5-Core jederzeit angehalten werden. Zu Beginn muss die Nachrichtenlänge an msg_len anliegen. Die Komponente liest Bytes über msg_byte in jedem Takt mit msg_byte_stable='1' ein.

Wird mit last_byte='1' signalisiert, dass alle Bytes der Nachricht eingelesen wurden, so schließt die Komponente die Formatierung des aktuellen Blocks mit Anhängen der Länge in Bits als 64-Bit-String und gegebenenfalls Padding ab. Daraufhin wird die Prüfsumme des letzten Blocks berechnet und zum bisherigen Zwischenergebnis der Prüfsumme vorheriger Blocks addiert.

Ist die Verarbeitung abgeschlossen, wird dies mit done='1' angezeigt. Sollen weiter Nachrichten verarbeitet werden kann die Komponente mit init='1' neu gestartet werden und es wird wie beschreiben eine neue Prüfsumme berechnet.

### 13.1.3 Blockschaltbild

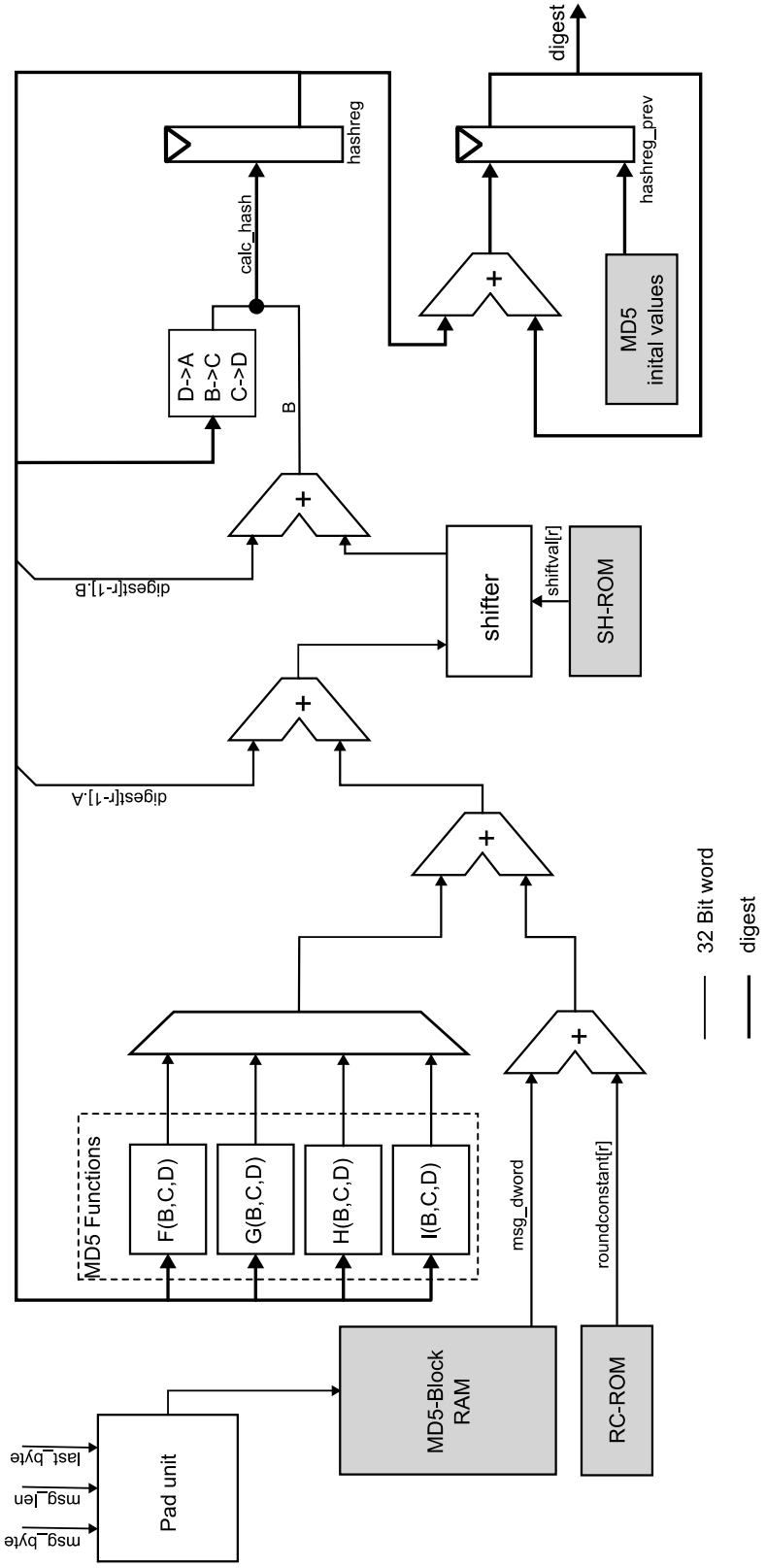


Abbildung 25: MD5-Core Blockschaltbild

### 13.1.4 FPGA-Ressourcen

Die Ressourcen und Timing-Angaben auf Altera FPGAs wurden mit „Mentor Graphics Precision RTL 2007“ ermittelt und sind lediglich eine Abschätzung.

| FPGA | LUTs/LCs/ | Register | Memory |
|---|---|---|---|
| Cyclone II EP2C70F896C | 2691 (3,9%) | 1445 (2,1%) | 3072 (0,27%) |

Tabelle 21: FPGA Ressourcen (MD5)

Die maximale Betriebsfrequenz liegt bei 57,6 MHz, 17,35 ns Register-Register-Delay.

## 13.2 MD5CoreController

Der MD5CoreController bindet den MD5-Core über den Systembus an den SystemController, vgl. Kapitel 9 und das SPI-Slave-Modul, vgl. Kapitel 8 an. In diesem Modul werden die Protokollinformationen ausgewertet und die Datenbytes der SPI-Schnittstelle in das entsprechende Datenformat des Interface des MD5-Core umgesetzt.

### 13.2.1 Interface

Die Schnittstelle des `MD5CoreControllers` setzt sich aus dem Systembus „`sysconnect`“ vom Typ `System_IF` aus Kapitel [9.2] zum SystemController, vgl. Kapitel 9 und der Schnittstelle des MD5-Core [12.1] zusammen.

### 13.2.2 Protokoll

Nach der Aktivierung über den SystemController befindet sich der Zustandsautomat des MD5CoreController im Initialzustand `IDLE`. Das Verarbeiten einer Nachricht wird mit dem Symbol `MB_LENGTH` gestartet. Daraufhin werden vier Bytes Längeninformation des Nachrichtenbytestroms erwartet. Diese müssen *least-significant-byte-first* gesendet werden. Wurde die Längeninformation empfangen, geht der Automat in den Zustand `ProcessMD5` über, in dem die Nachricht in den MD5-Core geschrieben wird.

Die Komponente MDCoreController beinhaltet den Zähler, der das Signal `last_byte='1'` setzt.

Nach der Verarbeitung im Zustand `ProcessMD5` geht Zustandsautomat in den Initialzustand über. Mit dem Symbol `MB_RESULT` wird der Automat in den Zustand `SENDRESULT` übergeführt. Daraufhin werden 16 Füllbytes zur Übertragung der 128-Bit großen Prüfsumme erwartet. Nach 16 Bytes wird dem SystemController mit `sysconnect.done='1'` der Kontrollfluss zurückgegeben.

In jedem Zustand kann mit dem Symbol MB_POLL abgefragt werden, ob eine Prüfsumme vorhanden ist. Im Zustand POLL sendet die Komponente entweder 16, falls eine Prüfsumme vorliegt, die noch nicht gesendet wurde, oder 0, falls die Verarbeitung noch nicht abgeschlossen ist.

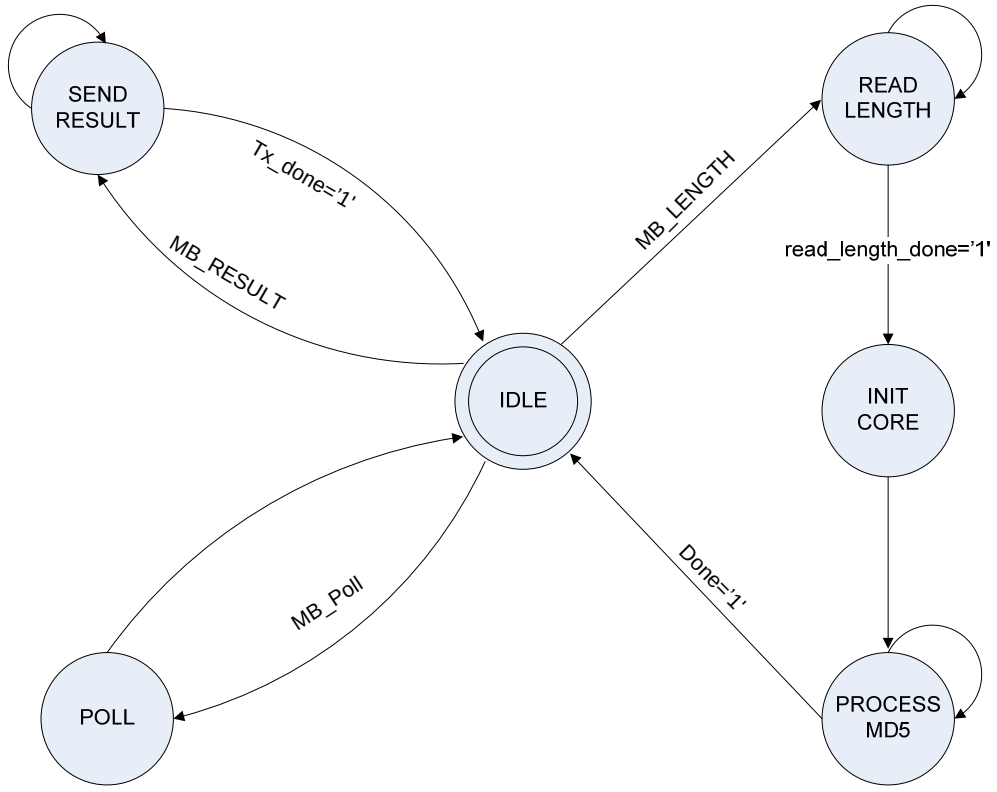


Abbildung 26: Protokollzustandsautomat MD5CoreController

# 14 Softwarekomponenten

Um die Hardwarekomponenten in Automatisierungsprogrammen zu nutzen wurden Softwaretreiber als Strukturierte Komponenten [GÖSE04] implementiert, die mit dem Security-Chip kommunizieren. Im Folgenden sind diese Treiber als Proxy-Komponenten bezeichnet. Proxy-Komponenten treten an die Stelle der Softwareimplementierung der kryptographischen Algorithmen und implementieren dieselbe Schnittstelle. Dadurch ist die Verwendung der Hardware für die Applikation transparent.

Die Proxy-Komponenten nutzen die SPI-Schnittstelle des Mikrokontroller-System als gemeinsame Ressource. Die Funktionalität der SPI-Schnittstelle wird durch die Strukturierte Komponente SPI-Proxy bereitgestellt. Es wurde in dieser Arbeit jedoch keine Synchronisation auf die Ressource mit Semaphoren implementiert. Dies war nicht Teil der Arbeit, da die Testapplikation keine parallelen Ausführungsablauf aufweist.

## 14.1 Strukturierte Komponente AesProxy

Die strukturierte Komponente `AesProxy` implementiert das Interface `ISymAlgo` für die Nutzung des Security-Chip in bestehenden Anwendungen,die AES-Funktionen nutzen. In der strukturierten Komponente ist der Zustandsautomat des Security-Chip implementiert. Der `AesProxy` nimmt Funktionsaufrufe des ISymAlgo Interface entgegen und übersetzt die Anfrage in das Protokoll des AES-CoreControllers auf dem Security-Chip.

Alle Funktionsaufrufe sind „blocking“, das bedeutet, die Proxykomponente kehrt erst aus der Funktion zurück wenn die Transaktion mit dem Security-Chip abgeschlossen ist.

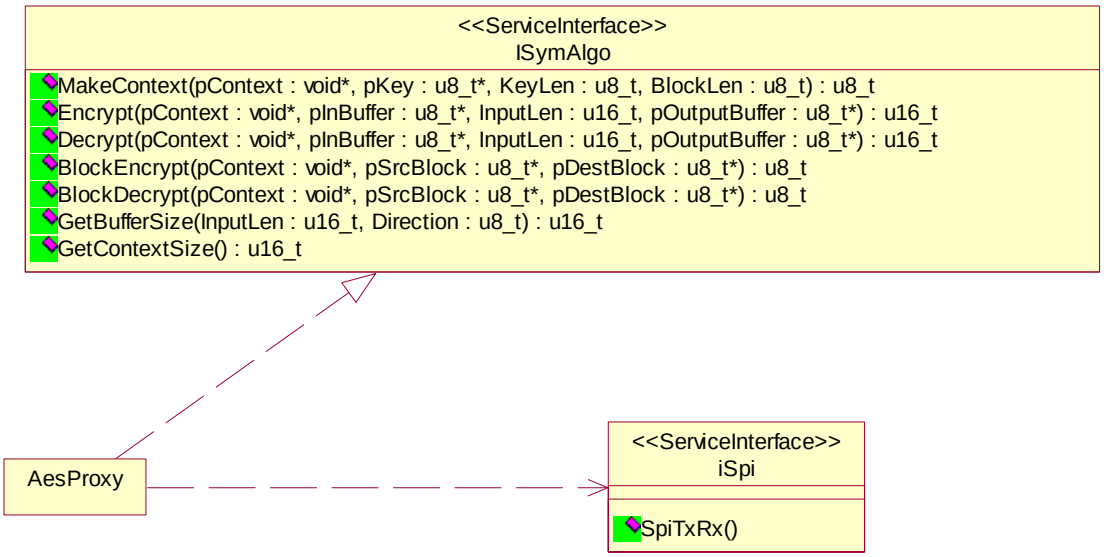


Abbildung 27: AesProxy UML Diagramm

### 14.1.1 Funktionen

`MakeContext` lädt den Userkey `pKey` aus dem auf den Security-Chip. Der Zeiger pContext wird in dieser Implementierung des Interface `ISymAlgo` ignoriert, da der Context auf dem Security-Chip gespeichert ist. Ebenso werden die Parameter `KeyLen` und `BlockLen` ignoriert, da diese Implementierung nur eine Schlüssel und Blocklänge von 128 Bit oder 16 Byte vorsieht.

```
void MakeContext(void* pContext, u8_t* pKey, u8_t KeyLen, u8_t BlockLen)
```

| **Parameter** | **Bedeutung** |
|---|---|
| pContext | Instanz einer AES-Operation (ignoriert) |
| pKey | Zeiger auf Schlüssel |
| KeyLen | Länge des Schlüssels in Bytes (ignoriert) |
| BlockLen | Länge eines DatenBlocks (ignoriert) |

`BlockEncrypt` verschlüsselt einen den Datenblock `pInBuffer` und legt das Ergebnis in `pOutBuffer` ab.

```
void BlockEncrypt(void* pContext, u8_t* pInBuffer, u8_t* pOutBuffer)
```

| **Parameter** | **Bedeutung** |
|---|---|
| pContext | Instanz einer AES-Operation (ignoriert) |
| pInBuffer | Zeiger auf einen Klartextdatenblock |
| pOutBuffer | Zeiger auf Speicherbereich für Geheimtext |

`BlockDecrypt` entschlüsselt den Datenblock `pInBuffer` und legt das Ergebnis in `pOutBuffer` ab.

```
void BlockDecrypt(void* pContext, u8_t* pInBuffer, u8_t* pOutBuffer)
```

| **Parameter** | **Bedeutung** |
|---|---|
| pContext | Instanz einer AES-Operation (ignoriert) |
| pInBuffer | Zeiger auf einen Geheimtextdatenblock |
| pOutBuffer | Zeiger auf Speicherbereich für Klartext |

Decrypt entschlüsselt den Datenblock pInBuffer der Länge InputLen und legt das Ergebnis in pOutBuffer ab.

```
void Decrypt(void* pContext, u8_t* pInBuffer, u8_t InputLen, u8_t*
pOutBuffer)
```

| Parameter | Bedeutung |
|---|---|
| pContext | Instanz einer AES-Operation (ignoriert) |
| InputLen | Länge des Geheimtextblocks |
| pInBuffer | Zeiger auf einen Geheimtextdatenblock |
| pOutBuffer | Zeiger auf Speicherbereich für Klartext |

Encrypt entschlüsselt den Datenblock pInBuffer der Länge InputLen und legt das Ergebnis in pOutBuffer ab.

```
void Encrypt(void* pContext, u8_t* pInBuffer, u8_t InputLen u8_t* pOutBuffer)
```

| Parameter | Bedeutung |
|---|---|
| pContext | Instanz einer AES-Operation (ignoriert) |
| InputLen | Länge des Klartextblocks |
| pInBuffer | Zeiger auf einen Klartextdatenblock |
| pOutBuffer | Zeiger auf Speicherbereich für Geheimtext |

GetContextSize gibt die benötigte Grösse für die Speicherung eines Context zurück. In dieser Implementierung immer 0.

```
u16_t GetContextSize(void)
```

GetBufferSize gibt die Blockgröße zurück. In dieser Implementierung immer 16 Bytes.

```
u16_t GetBufferSize(u16_t InputLen, u8_t direction)
```

## 14.2 Strukturierte Komponente CcmProxy

Die Strukturierte Komponente CcmProxy implementiert das Interface ICcmAlgo für die Nutzung des Security-Chips in bestehenden Anwendungen, die CCM-Funktionen nutzen. In der Strukturierten Komponente ist der Zustandsautomat des Security-Chips implementiert. Der CcmProxy nimmt Funktionsaufrufe des ICcmAlgo-Interface entgegen und übersetzt die Anfrage in das Protokoll des CCMCoreControllers auf dem Security-Chip.

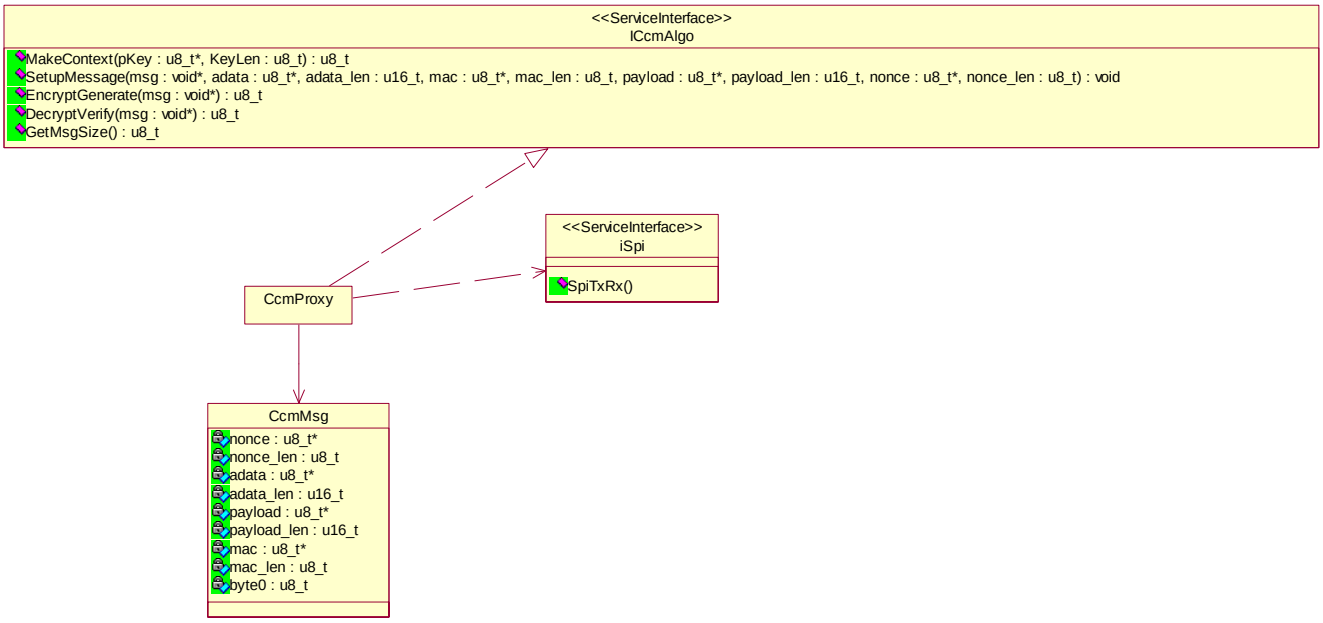


Abbildung 28: UML-Diagramm des CCMProxy

### 14.2.1 Die Struktur CcmMsg

Die Struktur `CcmMsg` speichert alle für eine Nachricht relevanten Daten. Zur Verwendung muss bei `SetupMessage()` ein Zeiger auf einen leeren Speicherbereich übergeben werde, dort werden die Zeiger für die einzelnen Nachrichteneile abgelegt.

Die Funktionen des CcmProxy arbeiten mit Zeigern auf diese Struktur. Das Ergebnis der Operationen wird an die jeweiligen Felder der Struktur gespeichert. So überschreibt `DecryptVerify()` `msg.payload` mit dem Klartext der Nachricht, `EncryptGenerate()` liest den Klartext aus `msg.payload` und ersetzt ihn durch den Geheimtext.

```
typedef struct{
      u8_t* nonce,
      u8_t  nonce_len,
      u8_t* adata;
      u16_t adata_len;
      u8_t* mac;
      u8_t  mac_len;
      u8_t* payload;
      u16_t payload_len
      u8_t byte0;
} CcmMsg;
```

Abbildung 29: Struktur CcmMsg

| Feld | Bedeutung |
|---|---|
| adata | Associated-Data der Nachricht, die für die Erstellung des MAC benutzt, selbst aber nicht verschlüsselt werden. |
| adata_len | Länge des Associated-Data Blocks (in Bytes) |
| mac | Zeiger auf Speicher mit dem MAC der zu prüfenden Nachricht (im Falle von DecryptVerify) oder einen leeren Bereich zum Speichern des erzeugten MAC (im Falle von EncryptGenerate) |
| mac_len | Länge des Message Authentication Codes.(in Bytes) |
| payload | Zeiger auf den zu verschlüsselnden Teil der Nachricht. |
| payload_len | Länge der zu verschlüsselnden Daten(in Bytes) |
| nonce | Zeiger auf den kryptographischen Nonce für die Erzeugung der Counterblocks und des MAC. |
| nonce_len | Länge des kryptographischen Nonce. Da die maximale Payload_len mit 16 Bit kodiert ist, ist der nonce_len konstant 13 (in Bytes) |
| Byte0 | Header für Block0, wird automatisch generiert |

### 14.2.2 Funktionen

`MakeContext` aktiviert den CCM-Core des Security-Chips und lädt den symmetrischen Benutzerschlüssel in den AES-Core.

```
void MakeContext(u8_t* pKey, u8_t KeyLen)
```

| Parameter | Bedeutung |
|---|---|
| pKey | Pointer auf den Speicherbereich des Userkey. |
| KeyLen | Länge des Schlüssels – In der Implementierung ignoriert, da nur ein 128-Bit AES-Core implementiert wurde. Immer 16 Bytes |

`SetupMessage` aktiviert den CCM-Core des Security-Chips und lädt den symmetrischen Benutzerschlüssel in den AES-Core.

```
void SetupMessage(void* msg, u8_t* adata, u16_t adata_len, u8_t* mac, u8_t
mac_len,u8_t* Payload, u8_t Payload_len, u8_t* Nonce, u8_t nonce_len)
```

| Parameter | Bedeutung |
|---|---|
| msg | Pointer auf den Speicherbereich für eine Nachricht. Hier werden die Parameter in die Struktur CcmMsg gespeichert. |

| | |
|---|---|
| adata | Associated-Data der Nachricht, die für die Erstellung des MAC benutzt, selbst aber nicht verschlüsselt werden. |
| adata_len | Länge des Associated-Data Blocks |
| mac | Zeiger auf Speicher mit dem MAC der zu prüfenden Nachricht (im Falle von Decrypt-verify) oder einen leeren Bereich zum Speichern des erzeugten MAC (im Falle von Encrypt-generate) |
| mac_len | Länge des Message Authentication Codes. |
| Payload | Zeiger auf den zu verschlüsselnden Teil der Nachricht. |
| Payload_len | Länge der zu verschlüsselnden Daten |
| Nonce | Zeiger auf den kryptographischen Nonce für die Erzeugung der Counterblocks und des MAC. |
| nonce_len | Länge des kryptographischen Nonce. Da die maximale Payload_len mit 16 Bit kodiert ist, ist der nonce_len konstant 13. |

`EncryptGenerate` verschlüsselt die Daten aus `msg.payload` und erstellt den dazugehörigen MAC. Dieser wird dann in `msg.mac` gespeichert. Die Payload der Nachricht wird durch den Geheimtext der Payload ersetzt.

```
void EncryptGenerate(void* msg)
```

| **Parameter** | **Bedeutung** |
|---|---|
| msg | Pointer auf eine Struktur CcmMsg, die mit SetupMessage() initialisiert wurde. |

`DecryptVerify` entschlüsselt die Daten aus `msg.payload` und überschreibt den Geheimtext in `msg.payload` mit dem Klartext Der Mac aus msg.mac wird übertragen und mit dem berechneten Mac verglichen. Sind die beiden Werte ungleich, so gibt der CCM-Core für die entschlüsselte Payload und den MAC nur 0x0F zurück.

```
void DecryptVerify(void* msg)
```

| **Parameter** | **Bedeutung** |
|---|---|
| msg | Pointer auf eine Struktur CcmMsg, die mit SetupMessage() initialisiert wurde. |

`GetMsgSize` liefert die Grösse der Struktur CcmMsg zurück.

```
u8_t GetMsgSize()
```

## 14.3 Strukturierte Komponente RsaProxy

Die Strukturierte Komponente `RsaProxy` implementiert das Interface `IRsa` für die Nutzung des Security-Chips in bestehenden Anwendungen, welche RSA-Funktionen nutzen. In der strukturierten Komponente ist der Zustandsautomat des Security-Chips implementiert. Der `RsaProxy` nimmt Funktionsaufrufe des IRsa Interface entgegen und übersetzt die Anfrage in das Protokoll des RSA-CoreController auf dem Security-Chip.

Alle Funktionsaufrufe sind „blocking“, das bedeutet, die Proxykomponente kehrt erst aus der Funktion zurück, wenn die Transaktion mit dem Security-Chip abgeschlossen ist.

Die Implementierung des RSA Algorithmus mit einem Montgomery Multiplizierers in Hardware erfordert eine vorher mit den Schlüsseln berechnete Konstante `R2modM`. Der RSA-Algorithmus ist bis zu einer Schlüssellänge von 1024 Bit ausgelegt und arbeitet mit einer Radix-2 Montgomery-Implementierung.

m=1024, r=2, M ist der Modulus des RSA-Schlüsselpaares mit der Länge `0<m<1024` Bit

$$R2 \bmod M = r^{2(m+2)} \bmod M$$

Formel 16: Berechnung des Faktors R2modM

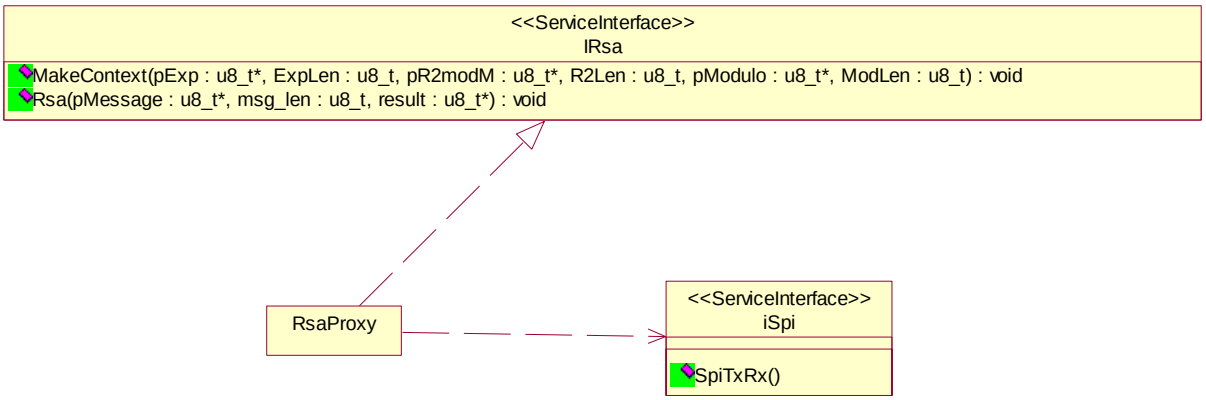


Abbildung 30: UML-Diagramm des RSAProxy

### 14.3.1 Funktionen

`MakeContext` lädt den Schlüssel bestehend aus Modulus `pModulo` und `pExponent` auf den Security-Chip. Dabei ist es für den Algorithmus irrelevant ob es sich um den privaten oder öffentlichen Schlüssel handelt. Zusätzlich wird der Faktor R2modM, der wie in Formel 16 vorher berechnet werden muss übertragen.

```
void MakeContext(u8_t* pExp, u8_t ExpLen, u8_t* pR2modM, u8_t R2Len, u8_t*
pModulo, u8_t ModLen)
```

| Parameter | Bedeutung |
|---|---|
| pExp | Zeiger auf den Exponenten der RSA Operation. Der Exponent muss LSB-first an der niederen Speicheradresse aufsteigend zu MSB abgelegt sein. pExp zeigt auf das LSB. |
| ExpLen | Länge des Exponenten (in Bytes) |
| pR2ModM | Zeiger auf Faktor für RSA-Operation, berechnet nach Formel 1, muss LSB-first an der niederen Speicheradresse aufsteigend zu MSB abgelegt sein. pR2modM zeigt auf das LSB. |
| R2len | Länge des Faktor R2modM(in Bytes) |
| pModulo | Zeiger auf Modulus, muss LSB-first an der niederen Speicheradresse aufsteigend zu MSB abgelegt sein. pModulo zeigt auf das LSB. |
| ModLen | Länge des Modulus (in Bytes) |

Die Operation unterscheidet nicht zwischen Ver- und Entschlüsselung. Daher bestimmt lediglich der mit `MakeContext` geladene Exponent darüber ob eine Ver- oder Entschlüsselung vorliegt.

```
void Rsa(u8_t* pMsg, u8_t MsgLen, u8_t* result)
```

| Parameter | Bedeutung |
|---|---|
| pMsg | Zeiger auf die zu bearbeitende Nachricht, muss LSB-first an der niederen Speicheradresse aufsteigend zu MSB abgelegt sein. pExp zeigt auf das LSB |
| MsgLen | Länge der Nachricht (in Bytes) |
| result | Zeiger auf eine 128 Byte großen Speicherbereich. Zur Speicherung des Ergebnis |

## 14.4 Strukturierte Komponente MD5Proxy

Die strukturierte Komponente MD5Proxy implementiert das Interface `IHashAlgo2` für die Nutzung des Security-Chip in bestehenden Anwendungen, die MD5-Funktionen nutzen. In der strukturierten Komponente ist der Zustandsautomat des Security-Chip implementiert. Der `Md5Proxy` nimmt Funktionsaufrufe des IHashAlgo2 Interface entgegen und übersetzt die Anfrage in das Protokoll des MD5-CoreControllers auf dem Security-Chip.

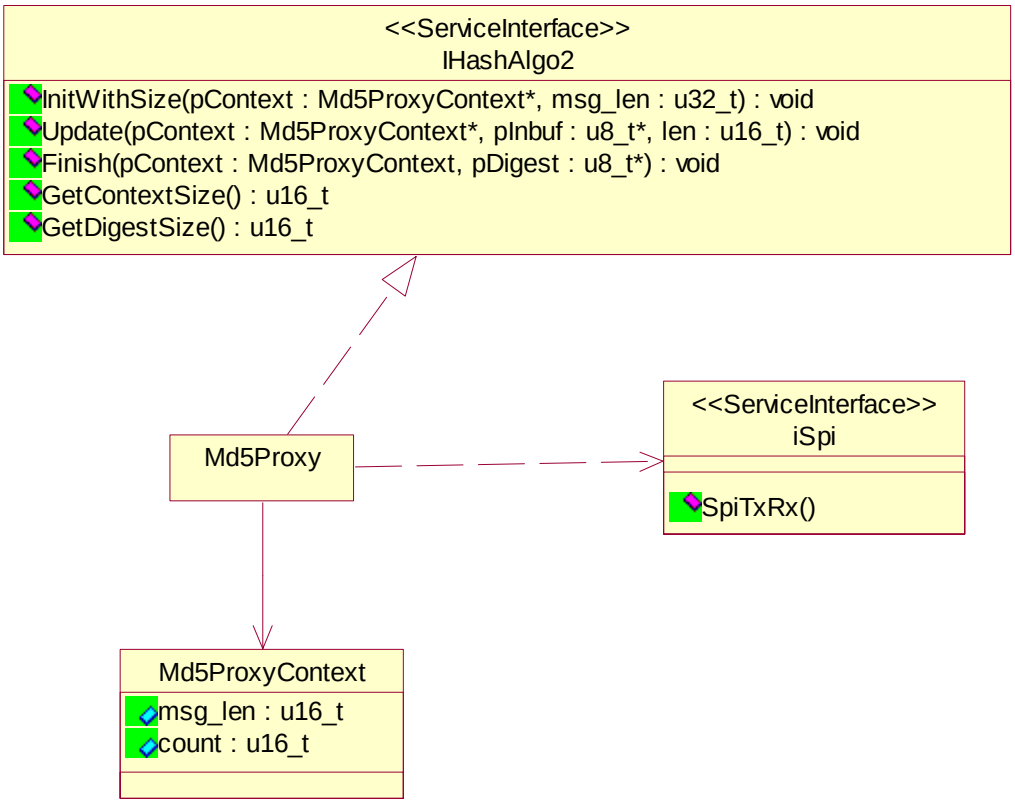


Abbildung 31: UML Diagramm MD5Proxy

### 14.4.1 Funktionen

`InitWithSize` initialisiert eine neue Hashoperation auf dem Security-Chip. Da Kontrollflussinformationen nicht getrennt übertragen werden können muss vorher bekannt sein wie groß die Nachricht ist über die, die Prüfsumme berechnet wird. Die Größe der Nachricht geht in die Berechnung mit ein.

```
void InitWithSize(Md5ProxyContext* pContext, u32_t msg_len)
```

| **Parameter** | **Bedeutung** |
|---|---|
| `pContext` | `Speichert die Anzahl der übertragenen Bytes und die Länge der Nachricht` |
| `msg_len` | `Länge der gesamten Nachricht in Bytes über die, die Prüfsummer berechnet werden soll. Das Datenformat ist Little-Endian. Innerhalb der Funktion wird in Big Endian gewandelt.` |

`Update` überträgt einen Teil der Größe `len` zum MD5Core des Security-Chips, um die Prüfsummer zu berechnen. Die in pContext gespeicherte Anzahl der übertragenen Daten wird aktualisiert.

```
void update(Md5ProxyContext* pContext, u8_t* pInbuf, u16_t len)
```

| **Parameter** | **Bedeutung** |
|---|---|
| pContext | Speichert die Anzahl der übertragenen Bytes und die Länge der Nachricht die Länge len wird zu den bisher der übertragenen Daten addiert. |
| pInBuf | Zeiger auf Speicherbereich für die hinzugefügten Daten |
| len | Länge des Blocks pInbuf |

`Finish` schließt die Berechnung der Prüfsumme ab. Dies kann nur geschehen wenn ebenso viele Datenbytes mit `update()` übertragen wurden, wie mit `InitWithSize()` angegeben wurde Der Messagedigest wird in den Speicherbereich `pDigest` kopiert.

```
void Update(Md5ProxyContext* pContext, u8_t* pInbuf, u16_t len)
```

| **Parameter** | **Bedeutung** |
|---|---|
| pContext | Speichert die Anzahl der übertragenen Bytes und die Länge der Nachricht die Länge len wird zu den bisher der übertragenen Daten addiert. |
| pDigest | Zeiger auf Speicherbereich für den Messagedigest (16 Bytes) |

`GetDigestSize` wurde aus Kompatibilitätsgründen mit anderen Hashalgorithmen implementiert. Die Länge des Digest ist hier konstant 16 Bytes.

```
u16_t GetDigestSize()
```

`GetContextSize` liefert die Größe des Speicherbereiches für einen MD5ProxyContext zurück, sie ist hier vier Bytes, zwei Bytes für die Soll-Nachrichten Länge und zwei Bytes für die Anzahl der bereits übertragenen Daten.

u16_t GetContextSize()

# 15 Evaluierung und Test

## 15.1 Testmethodik

### 15.1.1 Testprinzip

Alle in diesem Dokument beschriebenen Tests basieren auf Testvektoren für die zu testenden Algorithmen, d. h. einer Menge von Informationen, bestehend aus unverschlüsselten Daten (Plaintext), Schlüssel (Key) und verschlüsselten Daten (Ciphertext) bei den Verschlüsselungsalgorithmen, bzw. aus Daten (Data) und Prüfsumme (Hash Value) bei der Hash-Funktion. Zum Test werden solche Testvektoren aus Spezifikationen sowie durch existierende Implementierungen der Algorithmen in der Java Class Library von Sun Microsystems (JDK 6.0 Update 4) herangezogen. Zur Überprüfung der korrekten Funktionsweise werden die unverschlüsselten Daten der Testvektoren mit dem zu testenden Algorithmus verschlüsselt und die Ergebnisse mit den verschlüsselten Daten des Testvektors verglichen. Die Funktionsweise der Hash-Funktion wird analog geprüft, indem die Prüfsumme von Daten des Testvektors mit der Prüfsumme verglichen wird, welche von der zu prüfenden Hash-Funktion berechnet wurde.

Bei Performance-Tests wird die Zeit für die Ver-/Entschlüsselung, bzw. den Hash-Vorgang gemessen, aber keine Überprüfung auf korrekte Funktion vorgenommen. Die Testvektoren, die bei Performance Tests zum Einsatz kommen, sind mit denen der Funktionstests identisch, so dass von einer korrekten Funktion im Rahmen der Performancetests ausgegangen werden kann, wenn dies auch bei den Funktionstests der Fall ist.

### 15.1.2 Testumgebung

Alle Tests werden auf dem eingebetteten System, dem IAS-WebBoard durchgeführt. Dazu sind die Testfälle als strukturierte Komponenten auf dem IAS-WebBoard abgelegt und rufen die zu testende Funktionalität ebenso auf, wie dies auch eine „reale“ Applikation tun würde. Zu diesem Zweck wurde das Structured Component Test Framework (SCTF) herangezogen, welches wie folgt aufgebaut ist.

### 15.1.3 Architektur des SCTF

Das Structured Component Test Framework besteht aus zwei Teilen: einem Teil, der auf dem eingebetteten System ausgeführt wird und einem Teil, der auf einem PC ausgeführt wird.

Der Teil, der auf dem eingebetteten System ausgeführt wird, besteht aus der Strukturierten Komponente *TestManager* sowie einer oder mehreren *TestSuite*-Komponenten. Zusätzlich werden noch Kommunikationsfunktionalitäten benötigt, mit welchem die vorhandenen Testfälle

vom PC aus abgefragt und gestartet werden können. Im vorliegenden Fall wird hierzu der IAS-WebStack herangezogen.

Auf dem PC existiert eine Benutzungsschnittstelle, mit welcher die vorhandenen Testfälle abgefragt und ausgeführt werden können. Zudem können Testergebnisse als Testprotokoll gespeichert werden. Die Parametrierung von Testfällen ist auf dem PC ebenfalls möglich. Abbildung 32 stellt diese Architektur dar.

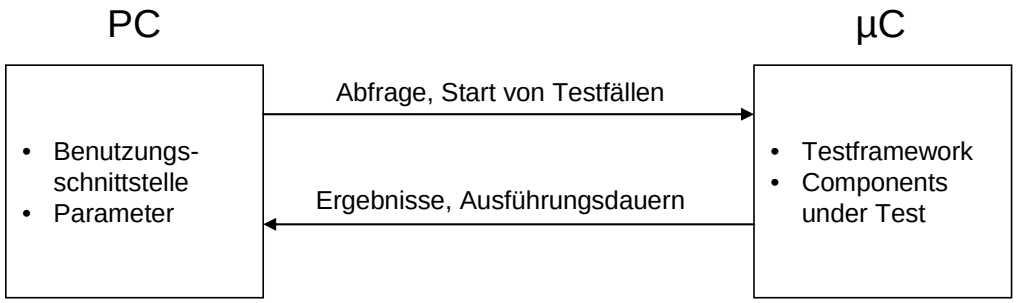


Abbildung 32: Architektur des Frameworks

### 15.1.4 Aufbau von Testfällen auf dem eingebetteten System

Der Aufbau von Testfällen stellt sich so dar, wie in der Abbildung 33 dargestellt. Die Komponente TestManager stellt die Schnittstelle zum PC dar, mit dem sie über den IAS-WebStack kommuniziert. TestSuite-Komponenten beinhalten Funktionen, welche die Testfälle realisieren und ein Testresultat an den TestManager zurückliefern, welches dann an den PC weitergeleitet und von diesem ausgewertet wird. Die TestSuite-Komponenten greifen auf die zu testenden Komponenten zu.

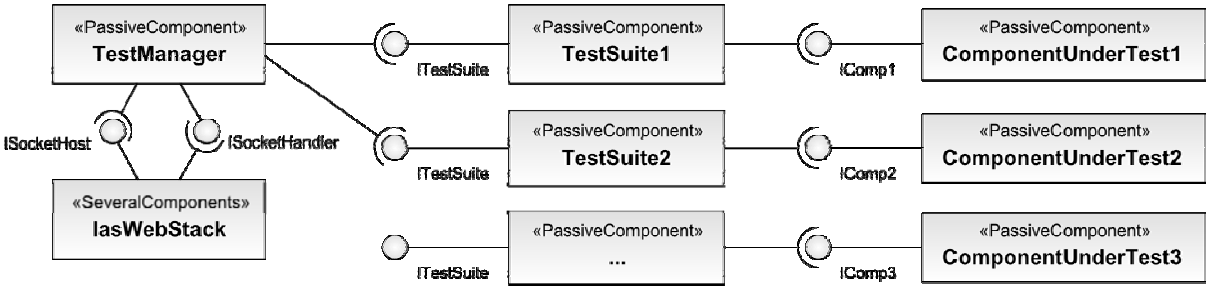


Abbildung 33: Aufbau von Testfällen

## 15.2 Ergebnisse

Im Folgenden sind Performancetests der einzelnen Komponenten dargestellt. Vor Durchführung der Leistungsmessung wurde die korrekte Funktion der Komponente getestet, in dem das

Ergebnis des Funktionsaufrufes mit einem gespeicherten Wert aus einer Referenzimplementierung verglichen wurde.

### 15.2.1 AES-Tests

In Abbildung 34 ist die Dauer der AES-Verschlüsselung im ECB-Modus in ms abhängig von der Nachrichtenlänge in Bytes dargestellt. Die rote Kurve ist beschreibt die Geschwindigkeit der Verschlüsselung der Softwareimplementierung auf dem M16C Mikrocontroller. Die übrigen Kurven geben die Geschwindigkeit der Hardwareimplementierung bei unterschiedlichen Frequenzen der SPI-Datenübertragung an.

Die Geschwindigkeitsmessung des CCM-Betriebsmodus der Blockchiffre ist nicht dargestellt, da sie im Wesentlichen der, des ECB-Modus entspricht.

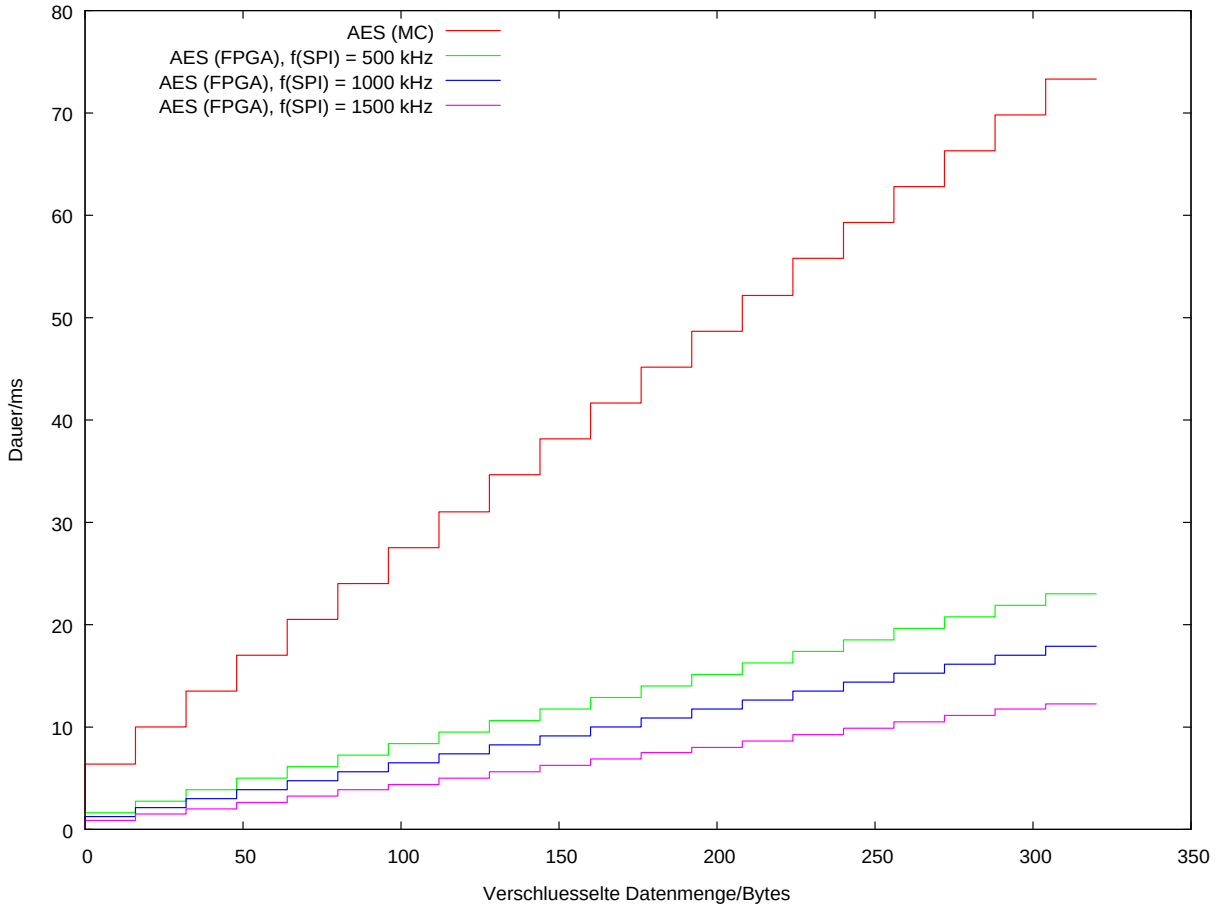


Abbildung 34: Messergebnisse AES

### 15.2.2 MD5-Tests

Die Hardwareimplementierung wurde in diesem Test wieder mit drei unterschiedlichen Frequenzen der SPI-Datenübertragung getestet, um die Abhängigkeit von der Übertragungsgeschwindigkeit darzustellen. Als Vergleich dienen zwei verschiedene

Softwareimplementierungen von Devine und Lauser, die auf dem M16C Mikrokontroller des Webboard ausgeführt wurden. Hierbei wird der aus der Informatik bekannte Time-Memory-Tradeoff sehr deutlich, während die Implementierung von Lauser wesentlich langsamer arbeitet, als die Implementierung von Devine, ist ihr Speicherbedarf entsprechend geringer.

Die Implementierung von Devine ist sogar effizienter als die Hardwareimplementierung, wobei das Ergebnis eindeutig von der Übertragungsgeschwindigkeit abhängig ist. Der Security-Chip ist in der Lage einen MD5-Datenblock mit 64 Byte in 65 Takten zu verarbeiten. Bei einem Systemtakt von 28 MHz entspricht dies 2,3 µs, während die SPI-Schnittstelle bei 1500 kHz für die Übertragung eines Bytes rund 5,3 µs benötigt.

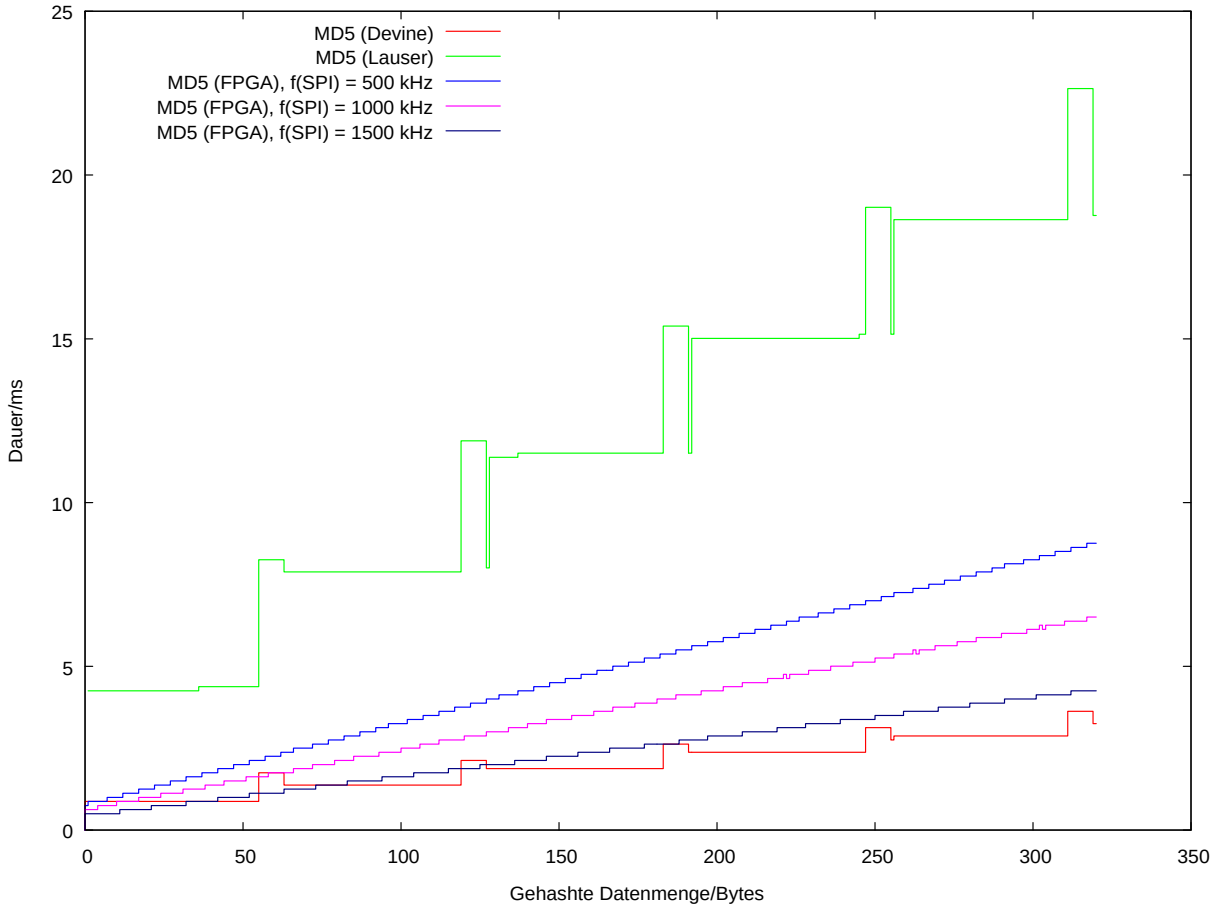


Abbildung 35: Messergebnisse MD5

### 15.2.3 Rsa-Tests

Abbildung 36 zeigt die Geschwindigkeiten für die Verschlüsselungen (E) und Entschlüsselungen (D) der RSA-Komponente bei unterschiedlichen Schlüssellängen (512 bit, 768 bit und 1024 bit). Bei den Ver- und Entschlüsselungen mit 1024 bit Schlüssellänge wurden zudem unterschiedliche Nachrichtenlängen verwendet. Es wird ersichtlich, dass der Haupteinfluss auf die Zeitdauern durch die Schlüssellänge bestimmt wird. Einen kleineren Einfluss übt die SPI-Frequenz aus. Die Nachrichtenlänge wirkt sich dagegen nur

vernachlässigbar auf die Verschlüsselungsdauer aus (unter der Annahme, dass die Nachricht durch eine einzige RSA-Operation verarbeitet werden kann).

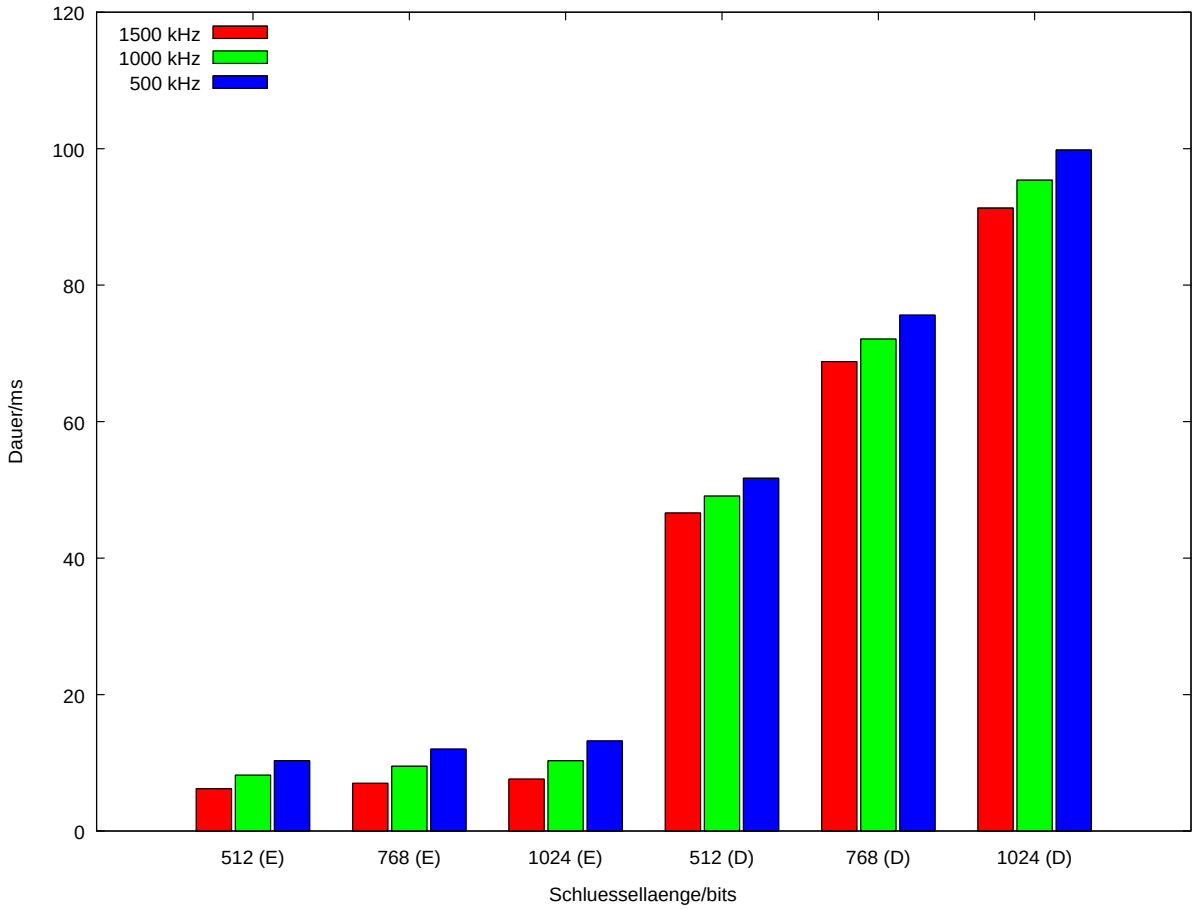


Abbildung 36: Messergebnisse RSA

Zum Vergleich werden hier noch die Zeitdauern angegeben, welche für die RSA-Verschlüsselung, bzw. RSA-Entschlüsselung gelten, wenn diese in Software auf dem M16C-Mikrocontroller durchgeführt werden. Die Verschlüsselung mit 1024 bit Schlüssellänge benötigt 1,470 s, die Entschlüsselung benötigt 156,115 s (ca. 2,6 min). Aus Maßstabsgründen sind diese Werte im oben stehenden Diagramm nicht eingezeichnet.

# 16 Ergebnis und Ausblick

In dieser Diplomarbeit wurden die kryptographische Algorithmen AES-ECB, AES-CCM, RSA und MD5 in VHDL beschrieben und auf einem FPGA realisiert. Zur Integration des Kryptographieprozessors in Automatisierungsanwendungen wurden Softwaretreiber als Strukturierte Komponenten implementiert, die die Funktionen der Hardware über Interfaces bereitstellen. Die Treiber besitzen dieselben Schnittstellen wie bereits bestehende Software-Implementierungen der Algorithmen. Dadurch lassen sich Hard- und Softwareimplementierung beliebig gegeneinander austauschen, ohne dass bestehende Anwendungen geändert werden müssen.

Die Architektur der Hardware ist modular, so daß der Kryptographieprozessor um Implementierungen anderer Algorithmen erweitert werden kann. Bestehende Algorithmenblöcke können entfernt werden, wenn es zur Einsparung von Chipfläche nötig ist.

Die Funktion des Gesamtsystems aus Hard- und Software wurde ausführlich auf Korrektheit gegen Softwarereferenzimplementierungen getestet. Es konnte gezeigt werden, dass die Ausführung der Algorithmen in Hardware bis zu einem Faktor 1500 effizienter ist, als ihre Softwareimplementierung. Als begrenzender Faktor der Verarbeitungsgeschwindigkeit wurde die verwendete SPI-Schnittstelle identifiziert. Eine beträchtliche Verbesserung der Performance ist durch die Verwendung anderer Hardwareschnittstellen zu erwarten.

Der Fokus dieser Arbeit lag auf der korrekten Implementierung der Algorithmen in Hardware und die Integration in das IAS-Security-Framework. Ansätze zur Optimierung in zukünftigen Arbeiten finden sich in den verwendeten Kommunikationsprotokollen der Proxies mit der Hardware.

# Literaturverzeichnis


[IEEE1076] **IEEE Std 1076.3-1997** IEEE Standard VHDL Synthesis Packages –Description

[Mey07] Meyers Lexikon Ausgabe 2007

[FIPS197] **Federal information Processing Standards Publication 197 "**Announcing the Advanced Encryption Standard (AES). – 26/11/2001 http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[RFC1321] **Request for Comments #1321** Ronald Rivest MIT Laboratory for Computer Science.: "The MD5 Message-Digest Algorithm" – 4/1992 http://tools.ietf.org/html/rfc1321

[BSM+01] **Bronstein, Semendjajew et al.-"**Taschenbuch der Mathematik" - 2001

[MOV96] Alfred Menezes, Paul Oorschot and Scott Vanstone: "Handbook of Applied Cryptography" –CRC Press 1996

[NIST38a] **NIST – National Institute of Standards and Technology** Special Publications 800-38a "Recommendation for Block Cypher Modes of Operation" 12/2001 http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

[NIST38b] **NIST – National Institute of Standards and Technology** Special Publications 800-38b "Recommendation for Block Cypher Modes of Operation The CMAC Mode for Authentication" 5/2005

http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf

[NIST38c] **NIST – National Institute of Standards and Technology** Special Publications 800-38c "Recommendation for Block Cypher Modes of Operation - The CCM Mode for Authentication and Confidentiality" 5/2004 http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C.pdf

[RFC3610] **Request for Comments #3610** Network Working Group: D. Witing, R.Housley et al.: Counter with CBC-MAC - 9/2003 http://tools.ietf.org/html/rfc3610#ref-AES

[RSAFC] **RSA Laboratories** The RSA Factoring Challenge 2005 http://www.rsa.com/rsalabs/node.asp?id=2964

[RSA2002] **RSA Laboratories.** PKCS#1 v2.1 – 6/2002

http://www.rsa.com/rsalabs/node.asp?id=2125

[TWP05] **Erik Tews, Ralf-Phillip Weinmann und Andrei Pyshkin: "**Breaking 104Bit WEP key in less than 60 seconds". TU Darmstadt 2005 http://eprint.iacr.org/2007/120.pdf

[LWW05] **Arjen Lenstra, Xiaoyun Wang, Benne Wegner:** Colliding X.509 Certificates März 2005 http://eprint.iacr.org/2005/067.pdf

[Kal02] **Kalinsky, David:** Introduction to Serial Peripheral Interface – Embedded Systems Design

[Mo95] **Peter L. Montgomery –** "Modular Multiplikation without Trial Division" Mathematics of Computation Vol. 44 – 1985

[Wa93] **C. D. Walter –** "Systolic Modular Multiplication" – Electronic Letters 1999

[Bl99] **Thomas Blum –** "Modular Exponentiation on Reconfigurable Hardware" – Master Thesis 1999

[DaMa02] **Alan Daly William Marnane – "**Efficient Architectures for Implementing Montgomery Modular Multiplication and RSA Modular Exponetation on Reconfigurable Logic" FPGA'02 2002

[HSH00] **Mohamed Khalil Hani, Tan Siang, Nasir Shaik Husin –** FPGA Implementation of RSA Public Key Cryptographic Coprocessor" – IEEE 2000

[FoKo05] **A.P. Fournaris, O. Koufopavlou –** "A new RSA Encryption Architecture and Hardware Implementation based on Montgomery Multiplication" – IEEE 2005

[GÖSE04] **Peter Göhner, Stephan Eberle –** Software Entwicklung für eingebettete Systeme mit Strukturierten Komponenten – Universität Stuttgart 2004

[Wiki08] **Wikipedia.org** – Freie Enzyklopädie 2008-05-05

http://www.wikipedia.org

Erklärung

Ich erkläre, die Arbeit selbständig verfasst und bei der Erstellung dieser Arbeit die einschlägigen Bestimmungen, insbesondere zum Urheberrechtsschutz fremder Beiträge, eingehalten zu haben. Soweit meine Arbeit fremde Beiträge (z. B. Bilder, Zeichnungen, Textpassagen) enthält, erkläre ich, dass diese Beiträge als solche gekennzeichnet sind (z. B. Zitat, Quellenangabe) und ich eventuell erforderlich gewordene Zustimmungen der Urheber zur Nutzung dieser Beiträge in meiner Arbeit eingeholt habe.

Unterschrift:

Stuttgart, den 08.05.08