IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz
Universität
Hannover

Masterarbeit

Konzeptionierung und Implementierung einer hybriden MAC-Layer-Architektur für Paket-basierte Powerline Kommunikation auf einem FPGA

Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Mikroelektronische Systeme
Fachgebiet Architekturen und Systeme

vorgelegt von

Niklas Rother, B.Sc.
Matrikelnummer 2942200
geb. am: 10.02.1994 in: Langenhagen

Erstprüfer:	Prof. Dr.-Ing Holger Blume
Zweitprüfer:	Jun.-Prof. Dr.-Ing. Guillermo Payá Vayá
Betreuer:	Tobias Stuckenberg, M.Sc.

Hannover, 18. Oktober 2017

Erklärung

Ich versichere hiermit, dass ich die vorstehende Arbeit selbständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und sowohl wörtliche, als auch sinngemäßentlehnte Stellen als solche kenntlich gemacht habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Hannover, den 18. Oktober 2017

Niklas Rother

IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz Universität Hannover

Hannover, 18. April 2017

MASTERARBEIT

Konzeptionierung und Implementierung einer hybriden MAC-Layer-Architektur für Paket-basierte Powerline Kommunikation auf einem FPGA

Am Fachgebiet „Architekturen und Systeme" des Instituts für Mikroelektronische Systeme werden VLSI-Architekturen für Algorithmen der digitalen Signalverarbeitung mit besonderen Anforderungen an Echtzeitfähigkeit und Verlustleistung konzipiert und implementiert. Ein Forschungsschwerpunkt ist dabei die digitale Signalverarbeitung in der Kommunikationstechnik.

Heutige Anwendungen elektronischer Systeme setzen vermehrt auf die drahtlose Kommunikation und den Austausch von immer größer werdenden Datenmengen untereinander. Am Institut für Mikroelektronische Systeme werden unter anderem OFDM-basierte Kommunikationssysteme für diese Anforderungen konzeptioniert und evaluiert. Ein besonders interessantes Verfahren ist hier die Powerline Kommunikation. Diese ist ein Paket-basiertes Verfahren, welches nach dem Prinzip des Ethernet-Stacks einen MAC- und einen PHY-Layer definiert.

Der MAC-Layer übernimmt hierbei eine steuernde Rolle und leitet die Pakete an die PHY-Layer weiter, bewertet die frequenzbezogenen Eigenschaften des Übertragungskanals und kontrolliert den Verbindungsstatus zur Gegenstelle. Die Über- und Weitergabe von Paketen erfolgt hierbei über so genannte FIFO-Interfaces, jeweils zur darüber liegenden Schicht und dem PHY-Layer. Außerdem regelt der MAC-Layer den Medienzugriff auf die Powerline, wobei dieser parallel den Status des Mediums abfragen und Daten für das Senden vorbereiten muss.

Herr Niklas Rother bekommt die Aufgabe eine hybride Architektur, bestehend aus Hardwareeinheiten und ein oder mehreren Softcore-Prozessoren zu konzeptionieren. Die Implementierung eines MAC-Layer nach dem HomePlug V1.0.1 Standard liegt hierbei bereits als VHDL-Hardware-Beschreibung und C-Referenzimplementierung vor. Zunächst soll eine Evaluation der bestehenden Hardware-Architektur hinsichtlich der Umsetzbarkeit einzelner Komponenten auf einem Softcore untersucht werden. Anschließend werden verschiedene Architekturen von Softcore-Prozessoren verglichen und bezüglich ihrer hardwarespezifischen Parameter bewertet. Die am besten geeigneten Softcore-Prozessor-Architekturen werden schließlich in einem Gesamtkonzept als hybride Architektur mit weiteren Hardware-Komponenten auf einem FPGA realisiert und gegen die Referenzimplementierung verifiziert.

Prof. Dr.-Ing. Holger Blume

Prof. Dr.-Ing Holger Blume
blume@ims.uni-hannover.de

Appelstraße 4 | 30167 Hannover
fon 0511 762-19640 | fax 0511 762-19601
www.ims.uni-hannover.de | info@ims.uni-hannover.de

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Abkürzungen

BPSK Binary Phase Shift Keying

CRC Cyclic Redundancy Check

CSMA/CA Carrier Sense Multiple Access/Collision Avoidance

DBPSK Differential Binary Phase Shift Keying

DES Data Encryption Standard

DMA Direct Memory Access

DQPSK Differential Quad Phase Shift Keying

DSP Digital Signal Processing

FPGA Field Programmable Gate Array

GCC GNU Compiler Collection

ISA Instruction Set Architecture

LFB Local Frame Buffer

LFSR Linear Feedback Shift Register

OFDM Orthogonal Frequency-Division Multiplexing

ReasMem Reassembly-Speicher

ROBO Robust OFDM

VCS Virtual Carrier Sense

1 Einleitung

Der Zugang zu einer schnellen und stabilen Kommunikationsinfrastruktur ist ein zentraler Bestandteil modernen Lebens und Arbeitens. Häufig wird dabei eine drahtlose Verbindung verwendet, um auf das Verlegen dedizierter Kommunikationsleitungen zu verzichten. Ist eine drahtlose Kommunikation nicht möglich, stellt die Datenübertragung über Stromversorgungsleitungen eine oft verwendete Alternative dar. Dies wird als Powerline-Kommunikation bezeichnet.

Die Powerline-Technologie ist im Heimanwenderbereich bereits etabliert. Für diesen Bereich stehen standardisierte Systeme verschiedener Hersteller zur Verfügung. Auch im Bereich SmartHome-Vernetzung, Internet of Things (IoT) und Maschine-to-Maschine (M2M)-Kommunikation gewinnt die Powerline-Technik an Bedeutung. Powerline-Endgeräte werden, analog zum OSI-Schichtenmodell, in MAC- und PHY-Layer eingeteilt. Die MAC-Schicht steuert dabei die PHY-Schicht an und stellt eine Paket-orientierte Datenübertragungsschnittstelle bereit.

In der vorliegenden Arbeit soll eine Powerline-MAC-Schicht nach dem HomePlug-1.0.1-Standard entwickelt werden. Das System ist für den Einsatz im industriellen Umfeld mit begrenztem Bauraum vorgesehen. Eine möglichst kompakte Realisierung ist daher wichtig. Im System werden aus diesem Grund Softcores eingesetzt um eine ressourceneffiziente und flexible Implementierung zu erreichen. Das so entwickelte Design wird dann mit einer, zum Zeitpunkt der Arbeit bereits bestehenden, reinen VHDL-Implementierung verglichen. Kernpunkte sind dabei die Verwendung von FPGA-Ressourcen und die Anpassbarkeit des Systems. Zudem wird die erreichbare Datentransferrate untersucht.

In Kapitel 2 werden die Grundlagen erläutert, die für das Verständnis dieser Arbeit notwendig sind und der HomePlug-MAC-Layer beschrieben. Die entwickelte Implementierung wird in Kapitel 3 vorgestellt. Eine Evaluation der Ergebnisse sowie der Vergleich mit der Referenzimplementierung sind in Kapitel 4 zu finden. Abschließend werden die Ergebnisse in Kapitel 5 zusammengefasst und ein Ausblick gegeben.

2 Grundlagen

In diesem Kapitel werden die Grundlagen dargestellt, welche für das Verständnis der vorliegende Arbeit wichtig sind. Es werden in Abschnitt 2.1 die Grundzüge von Softcore-CPUs erläutert sowie die beiden in dieser Arbeit verwendeten Softcores, *NEO430* und *PauloBlaze* beschrieben. Abschnitt 2.2 behandelt den Cyclic Redundancy Check (CRC) zur Fehlerkorrektur. Es wird dann in Abschnitt 2.3 auf die Powerline-Kommunikation und in Abschnitt 2.4 auf den *HomePlug*-Standard eingegangen.

2.1 Softcore-CPUs

Im Bereich des Schaltungsdesigns bezeichnen „Softcore-Blöcke“ bestehende Schaltungen mit einer bestimmten Funktion, welche in Form einer Hardware-Beschreibungssprache oder als Netzliste vorliegen. Solche Blöcke können mit wenig Aufwand in eigenen Schaltungen eingesetzt werden. Durch die Verwendung eines vorgefertigten Designs kann der Entwicklungsaufwand und die Wahrscheinlichkeit von fehlerhaften Designs deutlich reduziert werden. Ist kein Softcore-Block mit genau der gewünschten Funktion verfügbar, muss hier unter Umständen ein Kompromiss bezüglich der Implementierung eingegangen werden. Falls der Softcore-Block in einer Hardware-Beschreibungssprache vorliegt, ist es jedoch teilweise möglich, den Block in der gewünschten Form anzupassen. Bei Blöcken in Netzlisten-Form ist dies nicht möglich und auch oft lizenzrechtlich nicht zulässig. Beispiele für Softcore-Blöcke sind unter anderem eingebette CPUs, De-/Encoder oder Verschlüsselungsfunktionen. Insbesondere, wenn es sich bei den Softcore-Blöcken nicht um CPUs handelt, werden diese auch als „IP-Cores“ bezeichnet.

Im Gegensatz zu Softcore-Blöcken spricht man von „Hardcore-Blöcken“ wenn entsprechende Funktionseinheiten direkt in ein Field Programmable Gate Array (FPGA) integriert sind. Hardcore-Blöcke bieten in der Regel höhere Taktraten sowie Vorteile bei der Energieeffizienz. Eine über die Konfiguration des Blockes hinausgehende Anpassung ist jedoch nicht möglich. Zusätzlich ist eine Migration des Designs auf andere FPGAs schwieriger, da

nur solche in Frage kommen, die identische oder kompatible Hardcore-Blöcke bieten. Häufig in FPGAs vorzufindende Hardcore-Blöcke sind z. B. Speicher, Multiplizierer, Digital Signal Processing (DSP)-Blöcke, teilweise sind aber auch ganze CPUs vorhanden [1].

Eine besondere Klasse stellen Softcore-CPUs dar, also Softcore-Blöcke, welche ein vollständige CPU enthalten. Blume definiert entsprechend einen Softcore-Prozessor als „ein vollständig in einer HDL beschriebener Mikroprozessor, der sich für einen programmierbaren Baustein, wie einen FPGA, synthetisieren lässt“ [2, Folie 6.7]

In der vorliegenden Arbeit werden zwei Softcore-CPUs verwenden, der *NEO430* von Stephan Nolting [3] sowie der *PauloBlaze* von Paul Genßler [7]. Die beiden CPUs werden in den folgenden Abschnitten ausführlich vorgestellt.

2.1.1 NEO430

Der NEO430 ist eine 16-Bit-CPU die den Instruktionssatz der MSP430-Familie (ohne 20-Bit-Erweiterung) von Texas Instruments implementiert [3]. Das Design ist in VHDL beschrieben und steht unter der LGPL-3.0-Lizenz.

Die vollständige Kompabilität zur Instruction Set Architecture (ISA) des MSP430 ermöglicht die Verwendung eines bestehenden Compilers, da der entstehende Binärcode ohne weitere Umwandlung direkt auf dem NEO430 ausgeführt werden kann. Für den MSP430 stehend verschiedene Compiler für die Programmiersprache C zur Verfügung, unter anderem auch eine Portierung des GCC-Compilers[1]. Auf diese Weise ist eine Programmierung in der Programmiersprache C möglich. Die angesproche Kompabilität bezieht sich jedoch explizit nur auf die eigentliche CPU. Speicher-Layout, verfügbare interne Peripherie sowie das Timing der einzelnen Instruktionen unterscheiden sich zwischen NEO430 und MSP430 deutlich. Insbesondere ist der in einigen MSP430 vorhandene Hardware-Multiplizierer nicht implementiert.

Wie Abbildung 2.1 zeigt, besteht das Design des NEO430 aus einer CPU, welche über einen zentralen Bus mit weiteren Funktionseinheiten verbunden ist. Die Implementierung der Funktionseinheiten kann über VHDL-Generics zur Synthesezeit gesteuert werden, so dass nicht benötigte Funktionseinheiten keine Resourcen (LUTs, Register, Speicher) verbrauchen. Es stehend die folgenden Funktionseinheiten zur Verfügung:

IMEM Bis zu 32 kB Instruktionsspeicher

DMEM Bis zu 28 kB Datenspeicher

[1] vgl. http://www.ti.com/tool/msp430-gcc-opensource

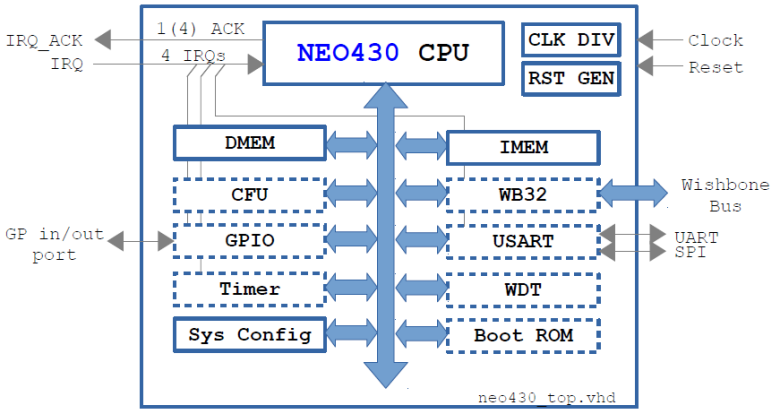


Abbildung 2.1: Blockdigramm des NEO430. Gestrichelte Linien markieren optionale Module [3].

GPIO 16 unabhängige binäre Ein- und Ausgabesignale

USART UART und SPI Sender und Empfänger

TIMER 16 bit Timer, inklusive Prescaler

WDT Watchdog zum Zurücksetzen des Prozessors bei einer Fehlfunktion

SYSCONFIG Informationen und Konfiguration der CPU

WB32 Adapter für das Wishbone-Protokoll[2]

CFU Vorlage für eigene Hardware-Erweiterungen mit CPU-Bus-Anbindung

Boot-ROM Bootloader mit UART/SPI Unterstützung

Neben den bestehenden Funktionseinheiten ist es möglich, eigene Erweiterungen mit der CPU zu verbinden. Hierzu stehen zwei Möglichkeiten zur Verfügung: Über das standardisierte Wishbone-Protokoll können Module angebunden werden, die ebenfalls dieses Protokoll unterstützen [4]. Dies ermöglicht eine einfache Verbindung sowie die Verwendung einer Vielzahl an schon bestehenden, Wishbone-kompatiblen Komponenten. Durch den Protokolloverhead sind Durchsatz und Latenz beschränkt. Eine weitere Option stellt die direkte Anbindung an den Bus der CPU dar. Diese Herangehensweise erfordert einen höheren Aufwand, da die Schnittstelle der Komponente an das Protokoll des CPU-Busses angepasst werden muss. Auf der anderen Seite treten so keine Einschränkungen bei Latenz und Durchsatz auf.

[2] vgl. [4]

	Altera Cyclone IV (LUT-4)		Xilinx Virtex 6 (LUT-6)	
	NEO430	openMSP	NEO430	openMSP
Logik	1188 LE	2024 LE	849 LUTs	1579 LUTs
FFs	557 Register	610 Register	652 FFs	594 FFs
Speicher	79 168 bit	–	4 BlockRAM	–
Max. Taktrate	116 MHz	51 MHz	173 MHz	102 MHz

Tabelle 2.1: Syntheseergebnisse verschiedener MSP430-kompatibler Softcores [3, 22]

Die ISA des MSP430 ist eine von-Neumann-Architektur, bei der Daten und Instruktionen in einem einheitlichen Adressraum liegen. Beim MSP430 sind auch alle Funktionseinheiten in diesen Adressraum eingeblendet (engl. memory-mapped). Durch diese Implementierung und den Adressraum von 64 kB stehen ausreichend Adressen zur Verfügung um Erweiterungsmodule anzubinden.

Die CPU unterstützt bis zu vier unabhängige Interruptquellen, die jeweils einen eigenen Interrupt-Handler haben können. Als Quellen kommen der Timer, die USART-Einheit, die GPIO-Einheit sowie eine externe Interupt-Quelle in Frage. Die Prioritäten sind dabei fest vergeben. Für externe Interuptquellen steht ein Acknowledge-Signal zu Verfügung, mit dem angezeigt wird, dass ein Interrupt von der CPU angenommen wurde.

Die NEO430-CPU wurde als Multi-Cycle-Architektur ohne Pipelining entworfen. Eine einzelne Instruktion benötigt also mehrere Takte zur Ausführung. Im vorliegenden Fall sind je nach Instruktion 6 bis 12 Takte notwendig um eine einzelne Instruktion auszuführen. Die Multi-Cycle-Architektur ermöglicht hohe Taktraten[3], limitiert aber bei einer feststehenden Taktrate den Instruktionsdurchsatz. Der originale MSP430 benötigt je nach Instruktion 1 bis 6 Takte zur Ausführung, der NEO430 benötigt also etwas 2 bis 3 Mal so viele Takte für die gleiche Instruktion. Mit dem openMSP430 steht eine weitere Implementierung der gleiche ISA zur Verfügung, welche in etwa das Original-Timing des MSP430 hat. Diese Variante benötigt allerdings etwa doppelt so viele FPGA-Ressourcen (siehe Tabelle 2.1) [5].

Laut Synthese-Ergebnissen des Entwicklers (siehe Tabelle 2.1) benötigt der NEO430 nur wenige Hardware-Ressourcen. Synthesen im Rahmen dieser Arbeit konnten dies bestätigen. Durch Auswahl der implementierten Komponenten an die Anforderungen des Design kann die Ressourcenausnutzung zudem weiter optimiert werden.

[3] laut Entwickler sind bis zu 175 MHz möglich

Änderungen der Architektur

Im Rahmen dieser Arbeit wurden einige Änderungen an der oben beschriebenen Grundversion des NEO430 vorgenommen, um ihn an die Anforderungen der Arbeit anzupassen. Einige dieser Änderungen wurden inzwischen in die Hauptversion übernommen.

In der Grundversion benötigt der NEO430 drei Takte um eine neue Instruktion abzurufen (Instruction Fetch). Durch das Einfügen eines Bypass zwischen Speicher und Instruktionsdekoder konnte diese Zeit um einen Takt reduziert werden. Da das Abrufen eines Befehls für jede Instruktion notwendig ist, konnte auf diese Weise die Anzahl der benötigten Takte pro Instruktion insgesamt um einen Takt reduziert werden[4], was einer Leistungsverbesserung von etwa 10 % entspricht. Diese Änderung wurde inzwischen in die Hauptversion des NEO430 übernommen.

Der NEO430 ermöglicht es, über die Software die Behandlung von Interrupts zu aktivieren und zu deaktivieren. Selbst wenn das Programm die Interrupts nie aktiviert, wird die entsprechenden Hardware dennoch immer synthetisiert. Um den Ressourcenverbrauch zu zu reduzieren, wurde ein neues VHDL-Generic `IRQ_USE` eingefügt, welches es ermöglicht die Logik für die Interruptbehandlung vollständig zu entfernen. Dies vereinfacht insbesondere die Control Unit des Prozessors. Durch diese Änderung können einige Ressourcen gespart werden, wenn die Software die Interrupt-Funktionen des Prozessors nicht benötigt.

Das Timer-Modul des NEO430 enthält einen einzigen 16 bit-Timer, was für diese Arbeit zu nicht ausreichend ist. Das Modul wurde daher so modifiziert, dass über VHDL-Generics bis zu 4 weitere Timer hinzugefügt werden können. Diese Timer verfügen nicht über alle Funktionen, wie der ursprünglich vorhandene. So können die zusätzlichen Timer keine Interrupts auslösen und können nicht angehalten werden. Wenn ein Timer mit einem Wert geladen wird, zählt er automatisch herunter, bis er den Wert Null erreicht und hält dann an. Die zusätzlichen Timer verwenden immer den gleichen Prescaler wie der Haupt-Timer.

2.1.2 PauloBlaze

Der PauloBlaze ist eine Nachimplementierung des PicoBlaze 6 („KCPSM6“) von Xilinx [6]. Anders als dieser steht er aber unter der Apache-2.0-Lizenz, wodurch eine Verwendung ausserhalb von Xilinx-FPGAs möglich ist. Der PauloBlaze bietet die gleiche externe Schnittstelle wie der PicoBlaze und verwendet die selbe ISA, wodurch er vollständig kom-

[4] außer für Sprungbefehle

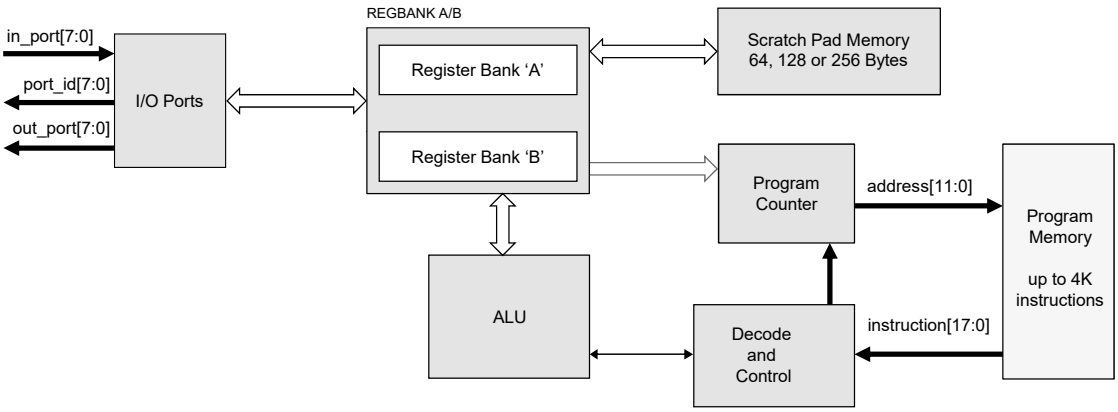


Abbildung 2.2: Architektur des Pico/PauloBlaze (nach [6])

patibel zu diesem ist. Eine Verwendung des von Xilinx entwickelten Assemblers ist daher möglich, ebenso wie die Verwendung eines beliebigen anderen, für den PicoBlaze entwickelten Assemblers. Der PicoBlaze liegt als Netzliste für verschiedene Xilinx-FPGAs vor, der PauloBlaze hingegen ist in VHDL beschrieben. Dies führt dazu, dass der PauloBlaze etwa 20 % bis 50 % mehr Ressourcen verbraucht als der PicoBlaze [7].

Pico- und PauloBlaze sind 8-Bit-CPUs, die mit besonderem Augenmerk auf einen geringen Ressourcenverbrauch entworfen wurden. Systemarchitektur (vgl. Abbildung 2.2) und Instruktionssatz spiegeln dies wieder. So sind neben den zwei Registerbänken mit jeweils sechzehn 8 bit-Registern nur maximal 256 Byte Datenspeicher (als „Scratchpad“ bezeichnet) vorgesehen. Alle Instruktionen verwenden zudem Register als Quelle und Ziel, sodass der Transfer von oder zum Speicher explizit kodiert werden muss. Zur Kommunikation mit anderen Teilen der Schaltung stehen 256 unabhängige, jeweils 8 bit breite Ein- und Ausgabeports zur Verfügung. Der Zugriff auf diese erfolgt ebenfalls über spezielle Instruktionen.

Alle Instruktionen des Prozessors sind 18 bit breit und werden in jeweils zwei Takten ausgeführt. Die ungewöhnliche Bitbreite von 18 bit entspricht genau der Breite der eingebetten Speicher-Blöcke in Xilinx-Prozessoren (1K×18). Ursprünglich ist pro Byte jeweils 1 Bit als Paritätsbit vorgesehen. Für den Pico- bzw. PauloBlaze werden die beiden Bits aber als reguläre Daten verwendet. Abbildung 2.3 zeigt eine typische Verbindung zwischen CPU und Speicher (BlockRAM). Beide Blöcke können direkt verbunden werden. Es sind maximal 4096 Instruktionen adressierbar.

Der Befehlssatz des Pico/PauloBlaze umfasst nur wenige Instruktionen. Es stehen die typischen arithmetischen und logischen Operationen, sowie Schiebe- und Programmfluss-

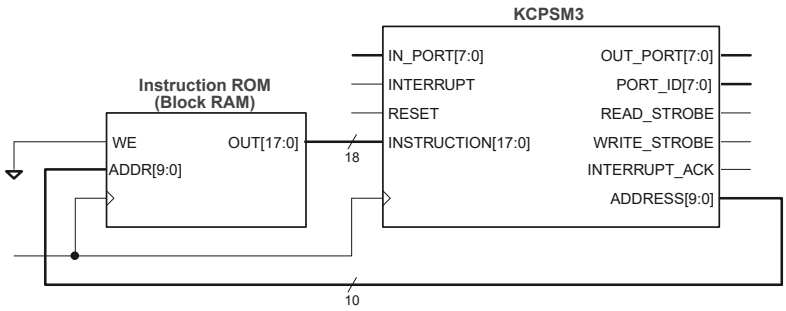


Abbildung 2.3: Verbindung zwischen PicoBlaze und Speicher. Es ist ein KCPSM3 mit maximal 1024 Instruktionen dargestellt [6].

operationen zur Verfügung, jedoch keine Operationen um relative Sprünge zu realisieren. Als Besonderheit existieren zwei unabhängige Registerbänke (A und B), zwischen denen über spezielle Befehle gewechselt werden kann. Die CPU unterstützt eine einzelne Interruptquelle.

Der PauloBlaze verfügt über keine interne Peripherie, wie etwa Timer oder GPIO-Funktionen. Solche Einheiten müssen über die Ein- und Ausgabeports verbunden werden.

Durch die oben beschriebene, sehr limitierte Architektur ist eine Programmierung in einer Hochsprache wie C nicht sinnvoll. Es steht aber eine Vielzahl an Assemblern zur Verfügung, neben dem von Xilinx selbst entwickelten, beispielsweise auch noch „opbasm“ des Entwicklers Kevin Thibedeau[5], der durch C-ähnliche Kontrollflussstrukturen die Entwicklung vereinfacht.

2.2 Cyclic Redundancy Check

Bei der Übertragung von Daten kann es zu kanalbedingten Fehlern kommen. Damit diese beim Empfänger erkannt werden können, müssen neben den Nutzdaten weitere, redundante Informationen übertragen werden. Diese werden häufig als *Paritätsbits* oder *Prüfsumme* bezeichnet. Ein oft verwendetes Verfahren ist der Cyclic Redundancy Check, kurz CRC. Dieses Verfahren wird im HomePlug-1.0.1-Standard an verschiedenen Stellen verwendet.

Zur Berechnung der CRC-Prüfsumme wird die Bitfolge, für welche die Prüfsumme berechnet werden soll, als Polynom $M(x)$ über dem Körper $\mathbb{Z}_2$ betrachtet. In diesem Körper sind Addition und Subtraktion identisch mit der XOR-Operation. CRC-Prüfsummen

[5] vgl. http://kevinpt.github.io/opbasm/

können in verschiedenen Längen verwendet werden. Es wird ein Generatorpolynom $G(x)$ benötigt, dessen Grad $deg(G) = l$ der gewünschten Länge der Prüfsumme entspricht. Das Generatorpolynom muss Sender und Empfänger bekannt sein. Häufige Werte für l sind 8, 16 oder 32.

Es wird im ersten Schritt $T(x) = x^l M(x)$ berechnet. Dies hat den gleichen Effekt wie l Nullen an die Bitfolge anzuhängen. Nun wird mit Hilfe der Polynomdivision $T(x)$ durch $G(x)$ geteilt. Der Rest der Division wird von $T(x)$ subtrahiert:

$$T'(x) = T(x) - T(x)/G(x)$$

$T'(x)$ ist nun ohne Rest durch $G(x)$ teilbar[6]: $T'(x)/G(x) = 0$. Die zu $T'(x)$ gehörende Bitfolge wird übertragen.

Der Empfänger teilt nun das zu der von ihm empfangenen Bitfolge gehörende Polynom $R(x)$ durch $G(x)$. Bleibt kein Rest, gilt höchstwahrscheinlich $R(x) = T'(x)$, und die Übertragung war fehlerfrei. Ein Übertragungsfehler bleibt nur dann unentdeckt, wenn das dem Fehler entsprechende Polynom ohne Rest durch $G(x)$ teilbar ist und daher den Divisionrest nicht beeinflusst. Es kann gezeigt werden, dass CRC-Prüfsummen alle Burstfehler der Länge $r \leq l$ erkennt. Die Wahrscheinlichkeit, dass ein längerer Burstfehler erkannt wird, liegt bei $1 - 1/2^r$ [16].

Werden an die übertragene Bitfolge vorne oder hinten Nullen angehängt kann dies durch die Überprüfung der CRC-Prüfsumme nicht erkannt werden. Dies kann passieren, wenn etwa ein Feld mit einer Längenangabe falsch übertragen wird. Um dieses Problem zu vermeiden wird üblicherweise als Startwert bei Polynomdivision ein Wert nur aus Einsen verwendet. Zusätzlich wird die Prüfsumme vor der Übertragung bitweise invertiert. Auf diese Weise lassen sich führende oder folgende Nullen erkennen. Es muss beachtet werden, dass nun bei der Überprüfung nicht mehr der Rest Null eine fehlerfreie Übertragung anzeigt, sondern ein einzigartiger, für jedes Generatorpolynom unterschiedlicher Wert [19].

2.2.1 Serielle Hardware-Implementierung

Die Berechnung bzw. Überprüfung einer CRC-Prüfsumme in Hardware ist sehr einfach, wenn die Daten als Bitstrom vorliegen. Der Bitstrom wird dabei durch ein Schieberegister der Länge $l = deg(G)$ geleitet. Wenn das vorderste (älteste) Bit des Schieberegisters den

[6] $T'(x)$ wurde ja gerade durch Subtraktion des verbleibenden Restes konstruiert.

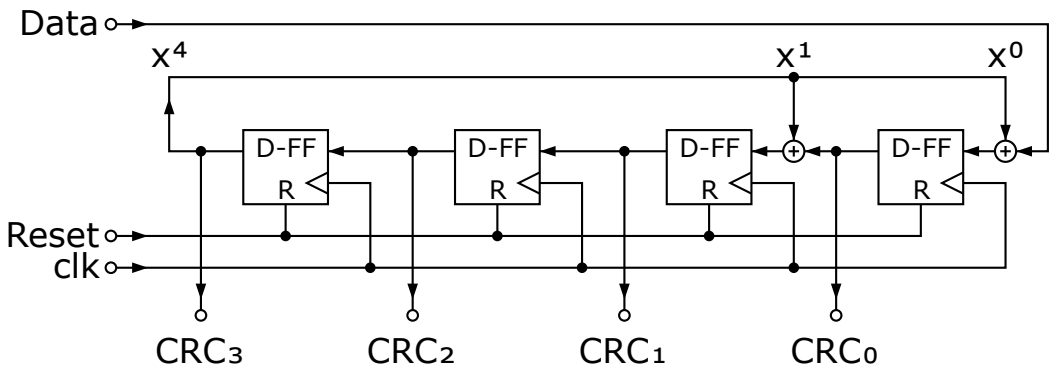


Abbildung 2.4: Beispielhafte serielle Hardware-Implementierung für das Generatorpolynom $x^4 + x^1 + x^0$

Wert 1 hat, muss das Generatorpolynom vom Inhalt des Schieberegisters subtrahiert werden. Es müssen also alle Stellen invertiert werden an denen das Generatorpolynom eine 1 enthält. Dies lässt sich sehr einfach durch XOR-Gatter realisieren.

Abbildung 2.4 zeigt eine beispielhafte Implementierung für das Generatorpolynom $G(x) = x^4 + x^1 + x^0$. Nachdem die Bitfolge des Polynoms $T(x)$, also die zu schützenden Nachricht gefolgt von vier Nullbits, in das Schieberegister geschoben wurde, enthalten die Flip-Flops die CRC-Prüfsumme.

Die berechnete Prüfsumme muss nun vom Eingabepolynom abgezogen werden, um die zu übertragende Bitfolge $T'(x)$ zu erhalten. Da die letzen l Bits des Eingabepolynoms per Definition Null sind und die Subtraktion durch eine XOR-Verknüpfung ausgeführt wird, kann auch einfach das Ende des Eingabepolynoms durch die berechnete Prüfsumme ersetzt werden, da eine XOR-Verknüpfung von 0 und x immer x liefert. In der Praxis bedeutet dies, dass zunächst die zu schützende Nachricht übertragen wird und dabei parallel in das Schieberegister eingespeist wird. Nun werden l Nullen in das Schieberegister gespeist und danach der Inhalt des Registers bitweise gesendet.

Auf Empfängerseite werden alle empfangenen Bits in das Schieberegister geschoben. Hat das CRC-Register am Ende den Wert Null (bzw. zu erwartenden Wert, wenn ein Startwert aus Einsen verwendet wurde, siehe oben) liegt kein Übertragungsfehler vor.

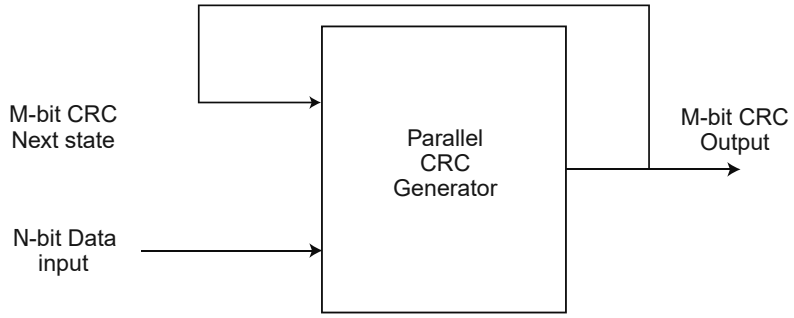


Abbildung 2.5: Funktion eines parallelen CRC-Blocks. Aus der aktuellen Prüfsumme und den nächsten Eingabedaten wird die neue Prüfsumme berechnet [20].

2.2.2 Parallele Hardware-Implementierung

Die oben beschriebene serielle CRC-Implementation ist sehr einfach und lässt sich mit wenig Hardware-Ressoucen realisieren. Der Durchsatz dieses Systems ist allerdings limitiert, da nur ein Bit pro Takt verarbeitet werden kann. Ein einfacher Ansatz zur Beschleunigung ist die Verwendung einer Look-Up-Tabelle. Wenn beispielsweise ein Byte pro Takt verarbeitet werden soll, werden in dieser Tabelle für alle 256 möglichen Eingabekombinationen die vorberechneten Divisionsreste hinterlegt. Durch diese Änderung kann dann ein Byte pro Takt verarbeitet werden, eine Steigerung um den Faktor acht. Für eine CRC16-Prüfsumme werden hierdurch jedoch $256 * 16\,\text{bit} = 4096\,\text{bit} = 512\,\text{B}$ an Speicherplatz benötigt.

Ein anderer Ansatz wird in [20] vorgestellt. Es wird dabei die Linearität der CRC-Berechnung ausgenutzt, um ein optimiertes kombinatorisches Netz zu erstellen. Dieses Netz bekommt den aktuellen Wert der Prüfsumme und die aktuellen Daten als Eingabe und gibt die neue Prüfsumme als Ausgabe aus, wie in Abbildung 2.5 gezeigt. Mit der in dem Artikel vorgestellten Methode ist es möglich, parallele CRC-Netze zu erstellen, welche die gewünschte Anzahl an Bits gleichzeitig verarbeiten. Die Größe des Netzes steigt dabei linear mit der Eingangsbitbreite. Pro Takt wird dabei ein Wort der Eingangsbitbreite verarbeitet.

Das kombinatorische Netz verwendet nur XOR-Operationen und lässt sich daher sehr gut auf die LUTs eines FPGA abbilden. In Abbildung 2.6 ist ein solches Netz für eine CRC8 Prüfsumme mit einen Eingangsbitbreite von siebzehn Bit dargestellt. In dieser Variante kann allerdings keine bestehende Prüfsumme zugeführt werden, es kann also nur von genau siebzehn Bit die Prüfsumme berechnet werden. Dies gescheit rein kombinatorisch.

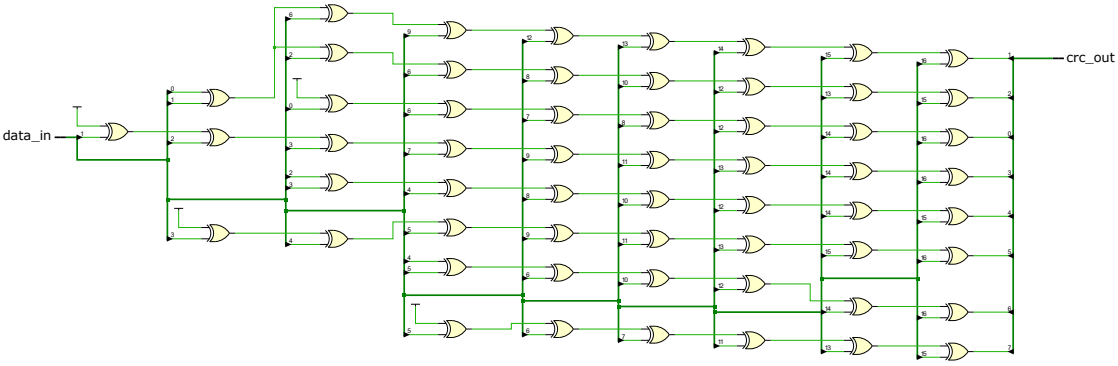


Abbildung 2.6: Beispielhaftes CRC-XOR-Netz mit Eingangsbitbreite von siebzehn Bit und Polynom $x^8 + x^2 + x + 1$. Es ist keine Rückführung des CRC-Wertes implementiert, es kann also nur von genau siebzehn Bit die Prüfsumme berechnet werden.

2.3 Powerline-Kommunikation

Als *Powerline communications* wird ein System bezeichnet, mit dem Daten über ein vorhandenes Stromversorgungskabel übertragen werden. Die Daten werden dabei auf einem oder mehreren Trägerfrequenzen auf die Leitung aufmoduliert. Ein solches System ermöglicht das Herstellen einer Datenverbindung unter Verwendung der bestehenden Infrastruktur.

Diese Technik ist als *PowerLAN* oder *dLan* im Heimanwendermarkt verbreitet. Dabei wird ein bestehenden LAN-Netzwerk über die üblicherweise bereits vorhanden Stromleitungen im Haus erweitert. Auf diese Weise entfällt der Aufwand für das Verlegen zusätzlicher LAN-Kabel. Neben dieser Anwendung wird Powerline-Kommunikation auch von Stromversorgungsunternehmen eingesetzt. Dabei wird zum einen die Kommunikation mit intelligenten Stromzählern ermöglicht um beispielsweise Zählerstände abzufragen, zum anderen aber auch die Verbindung zu Sensoren entlang von Hochspannungsleitungen hergestellt. In beiden Fällen wäre das Verlegen zusätzlicher Datenleitungen äußerst aufwendig. Schlussendlich wird die Technik auch zur Vernetzung von Maschinen im industriellen Umfeld eingesetzt [25].

Im Heimanwenderbereich ist eine hohe Datenrate, vor allem für Multimedia-Anwendungen, bei geringer Länge der Übertragungsstrecke gefragt, in den anderen Anwendungsbereichen ist vor allem eine hohe Störsicherheit aufgrund der großen Entfernung notwendig. Die Datenrate ist in diesen Bereichen von geringer Priorität [14].

Es existieren verschiedene Standards zur Powerline-Technik. In dieser Arbeit wird der in Europa und Amerika dominierende HomePlug-1.0.1-Standard verwendet.

2.4 HomePlug

Der HomePlug-1.0.1-Standard wurde 2001 von der HomePlug Powerline Alliance verabschiedet. Der Standard definiert ein Verfahren zur Powerline-Kommunikation welches vor allem für den Heimanwenderbereich gedacht ist. Es sind kompatible Geräte verschiedener Hersteller verfügbar. Das Verfahren erreicht eine Datenrate von bis zu 14 Mbit/s auf dem Kanal, davon bleiben bis zu 8,4 Mbit/s an Nutzdatenrate [23].

Es sind verschiedenen Weiterentwicklungen des Standards von der HomePlug Powerline Alliance verfügbar. 2007 wurde HomePlug AV verabschiedet, dieser Standard erhöht die Datenrate auf 200 Mbit/s. Das 2012 veröffentlichte HomePlug AV2 erreicht schließlich bis zu 1,5 Gbit/s. Neben dieser Hauptlinie existiert noch der HomePlug-GreenPHY-Standard der für Kommunikation zwischen intelligenten Geräten (Internet of Things, IoT) vorgesehen ist. In dieser Arbeit wird nur der HomePlug-1.0.1-Standard betrachtet.

Im HomePlug-1.0.1-Standard wird die Orthogonal Frequency-Division Multiplexing (OFDM)-Modulation zur Datenübertragung verwendet. Das Verfahren arbeitet mit 84 Trägern im Bereich von 4,49 MHz bis 20,7 MHz. Bei der Übertragung der Daten kommen verschiedene Modulationssverfahren zum Einsatz. Welches Verfahren benutzt wird, hängt von der Qualität des Kanals ab. Je nach Übertragungsqualität kann ROBO, DBPSK oder DQPSK zum Einsatz kommen. Die Kanalqualität wird dabei laufend überwacht und die Liste der verwendeten, störungsarmen Träger sowie das Modulationsverfahren angepasst.

Die Spezifikation des Standards sieht eine Aufteilung eines kompatiblen HomePlug-Gerätes in zwei Schichten vor, die MAC- und PHY-Schicht. Dies ist in Abbildung 2.7 dargestellt. Die Funktionen der beiden Schichten entsprechen dabei den gleichnamigen Schichten aus dem OSI-Referenzmodell: Die Bitübertragungsschicht (PHY) stellt die physische Verbindung zum Medium (in diesem Fall dem Stromkabel) her. Diese Schicht nimmt die Modulation der Daten gemäß OFDM vor. Die Medienzugriffsschicht (Media Access Control, MAC) steuert, welcher Teilnehmer wann senden darf und fügt den Daten Metainformationen hinzu. Auf diese Weise werden Prioritäten realisiert und Kollisionen vermieden (Carrier Sense Multiple Access/Collision Avoidance, CSMA/CA). Sie stellt

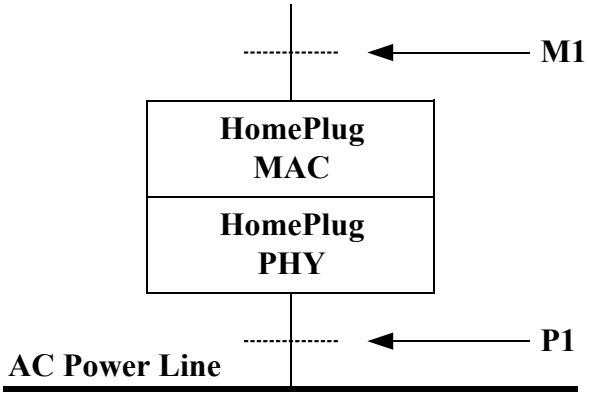


Abbildung 2.7: Schichtenmodell des HomePlug-1.0.1-Standards [9]

der darüberligendenden Schicht (Logical Link Control, LLC) eine Paket-orientierte Schnittstelle zur Verfügung. Die LLC-Schicht ist nicht Teil der Spezifikation.

Der HomePlug-Standard sieht vor, dass die übertragenen Daten optional mit DES verschlüsselt werden, um die Privatsphäre des Nutzer sicherzustellen. Das Stromnetz eines Hauses stellt einen gemeinsamen Kanal dar. Es ist daher schwierig, die Ausbreitung der Powerline-Signale etwa zum Nachbarn zu verhindern. Die Verschlüsselung ermöglicht es außerdem, mehrere logisch unabhängige Teilnetze im gleichen Stromnetz zu betreiben[7].

Der HomePlug-1.0.1-Standard definiert die Schnittstelle zwischen MAC- und LLC-Schicht, das sogenannte M1-Interface. Ebenso ist die Schnittstelle zwischen PHY-Layer und analogen Frontend definiert, das sogenannte P1-Interface. Die Verbindung zwischen MAC- und PHY-Schicht ist ebenfalls beschrieben, da diese beiden Schichten bei Powerline-Modems aber üblicherweise in einem Gerät vereint sind, ist an dieser Stelle eine strikte Standardisierung nicht notwendig. Die Spezifikation legt die Parameter von MAC- und PHY-Schicht fest. LLC-Schicht und analoges Frontend sind nicht Teil der Spezifikation.

In dieser Arbeit wird ausschließlich die MAC-Schicht betrachtet. Im Folgenden soll ihre Funktion nun näher beschrieben werden.

2.4.1 Homeplug 1.0.1 MAC-Schicht

Die MAC-Schicht stellt an der M1-Schnittstelle zur LLC-Schicht verschiedene Dienste bereit. Zu diesen Diensten gehört die Übertragung von Daten-Frames, aber auch die

[7] Da DES nach heutiger Einschätzung keine wesentliche Sicherheit mehr bietet, bleibt als hauptsächliche Funktion die Seperation der Netze. Zur Sicherstellung der Privatsphäre sollten die Daten auf einer höheren Protokollschicht zusätzlich verschlüsselt werden.

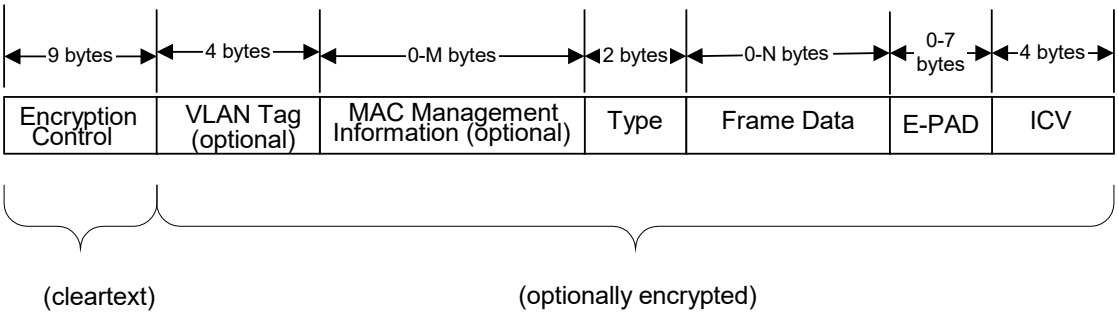


Abbildung 2.8: Aufbau eines HomePlug-MAC-Frames [9]

Konfiguration der DES-Schlüssel und die Abfrage von Statistiken. Auch eingehende Daten-Frames werden über diese Schnittstelle signalisiert.

Powerline-Übertragungen erfolgen üblicherweise über ungeschirmte Stromleitungen. Dadurch ist die Störanfälligkeit sehr hoch. Zudem gibt es durch Reflexionen oft mehr als einen Weg zwischen Sender und Empfänger, was zu Auslöschung und Mehrwegempfang von Signalen führen kann [26]. Um dennoch eine hohe Datenrate erreichen zu können, wird im HomePlug-Standard eine adaptive Modulation verwendet. Dabei wird laufend der Kanal zwischen zwei Stationen geschätzt. Hierbei werden Träger erkannt, die aufgrund von Störungen nicht verwendet werden können. Je nach Qualität von Anzahl der zur Verfügung stehenden Träger kommt bei der Übertragung dann ROBO-, DBPSK oder DQPSK-Modulation mit einer Fehlerkorrekturrate von 1/2 oder 3/4 zum Einsatz. Alle diese Informationen werden in einer sog. *Tonemap* zusammengefasst. Es werden 84 Bit benötigt, um die verfügbaren Träger zu speichern. Zusammen mit den Informationen über Fehlerkorrekturrate und Modulationsart umfasst eine Tonemap 104 Bit, also 13 Byte [9].

Es kommt eine eigene Tonemap für jedes Paar von Sender und Empfänger sowie für jede Richtung zum Einsatz. Da eine einzelne Station mit bis zu 15 Gegenstellen kommunizieren kann, muss sie bis zu 30 Tonemaps verwalten (jeweils 15 in Sende- und Empfangsrichtung). Die Tonemaps werden automatisch alle 30 s erneuert, um auf veränderliche Störungen zu reagieren.

Der HomePlug-1.0.1-Standard wurde konzipiert um die Kapselung einer 10 Mbit/s-Ethernet-Verbindung verlustfrei zu ermöglichen. Es ist daher vorgesehen, vollständige Ethernet-Frames mit einer Größe von bis zu 1500 Bytes zu übertragen.

Der MAC-Layer ergänzt die erhaltenen Nutzdaten – überlicherweise also ein Ethernet-Frame – um weitere Metadaten. In Abbildung 2.8 ist ein vollständiger MAC-Frame zu sehen. Der Frame beginnt mit einem 9 Byte großen *Encryption-Control*-Header.

Dieser enthält die Nummer des zu verwendenden gemeinsamen DES-Schlüssel und den 64 bit langen Initialisierungsvektor (IV). Alle folgenden Daten sind optional mit den angegebenen Parametern verschlüsselt [23].

Auf das Encryption-Control-Feld folgt der Ethernet-Frame. Dieser beginnt, wie im Ethernet-Standard vorgesehen, mit einem 2 Byte großen Typ-Feld. Wenn dieses Typ-Feld den Wert `0x8100` hat, folgt ein VLAN-Tag. Je nach Vorhandensein des VLAN-Tags ist ein weiteres Typ-Feld vorhanden. Hat dieses Feld den Wert `0x887B`, den für HomePlug reservierten IEEE Ethertype, dann sind die nachfolgenden Daten Mac-Managment-Informationen. Dies sind Steuerdaten, die vom MAC-Layer des Empfängers ausgewertet werden. Diese Mac-Managment-Informationen können bereits in dem auf der LLC-Schnittstelle erhaltenen Ethernet-Frame enthalten sein. Teilweise werden sie aber auch durch die MAC-Schicht des Senders hinzugefügt. Werden durch die MAC-Schicht des Senders Mac-Managment-Informationen hinzugefügt, müssen diese in einen bestehenden Mac-Managment-Block integriert werden, sofern ein solcher bereits vorhanden ist. Ein bestehender VLAN-Tag muss in jedem Fall unverändert weitergeleitet werden.

Das nun folgende Typ-Feld enthält den Ethertype des eigentlichen Ethernet-Frames. Dieser ist, wie VLAN-Tag und Mac-Managment-Informationen, optional. Es kann also auch ein Frame übertragen werden, welcher nur Mac-Managment-Informationen enthält. Es ist aber nicht möglich, einen Frame mit nur 1 Byte Nutzdaten zu übertragen, da das zwei Byte große Typ-Feld immer vollständig oder gar nicht vorhanden sein muss.

Nach den Nutzdaten folgen bis zu sieben Padding-Bytes. Diese sind notwendig, da der zur Verschlüsselung verwendete DES-Algorithmus nur mit 64 bit großen Blöcken arbeiten kann. Dieses Padding wird als EPAD (Encryption Padding) bezeichnet. Abgeschlossen wird der MAC-Frame von einer CRC32-Prüfsumme zur Fehlererkennung, die als *Integrity Check Value (ICV)* bezeichnet wird.

Ein MAC-Frame kann inklusive aller Metainformationen bis zu 1609 Byte groß sein. Die Übertragung dieser Datenmenge nimmt mit der ROBO-Modulation 14,45 ms in Anspruch. Die Wahrscheinlichkeit, dass während dieser Periode ein nicht-korrigierbarer Übertragungsfehler auftritt, ist hoch, da die Powerline-Übertragung generell sehr störanfällig ist [27]. Tritt ein Fehler auf, muss der komplette Frame erneut übertragen werden. Dies limitiert die Nutzdatenrate stark. Weiterhin kann eine laufende Übertragung nicht unterbrochen werden, der Kanal ist für die Dauer der Übertragung blockiert. Dies führt zu einer hohen Latenz, selbst für Frames mit der höchten Priorität (Prioritäteninversion).

Abbildung 2.9: Aufbau eines HomePlug-Segmentes [9]

Um diese beiden Probleme zu lösen, werden MAC-Frames im HomePlug-Standard nicht als Block übertragen, sondern segmentiert. Sollte ein Fehler auftreten, können einzelne Segmente wiederholt werden. Zusätzlich können Übertragungen mit höherer Priorität nach dem Ende jedes Segmentes durchgeführt werden. Die Segmente werden nicht nach einer festen Datenmenge gebildet, sondern für jedes Segment steht eine maximale Übertragungszeit von 1344 µs auf dem Kanal zur Verfügung. Je nach verwendeter Modulationsart können in dieser Zeit mehr oder weniger Daten übertragen werden. Diese zeitliche Limitierung führt dazu, dass es eine definierte maximale Verzögerung für Frames der höchsten Prioritätsstufe gibt, sofern nicht andere Stationen ebenfalls auf der höchsten Priorität senden möchten. Für die Segmentierung wird ein kompletter MAC-Frame als Bytefolge ohne weitere Struktur betrachtet.

Jedes Segment wird vor der Übertragung ebenfalls mit Metadaten versehen. Die Struktur eines Segmentes ist in Abbildung 2.9 gezeigt. Den Beginn eines Segmentes markiert ein *Segment-Control*-Header. Dieser enthält die Nummer und die Länge des aktuellen Segmentes. Danach folgen MAC-Adresse von Empfänger (DA) und Sender (SA) und die eigentlichen Daten [9].

Die Übertragung eines Segmentes erfolgt in einzelnen OFDM-Symbolen. Ein Segment kann dabei inklusive der Header 20 bis 160 Symbole lang sein, wobei die Länge nur in Zwanzigerschritten variiert werden kann. Falls nicht genügend Bytes zur Verfügung stehen, um eine Symbol komplett zu füllen, werden Padding-Bytes eingefügt. Dieses Padding wird als BPAD (Block Padding) bezeichnet. Den Abschluss bildet eine CRC16-Prüfsumme.

Um Kollisionen auf dem Kanal und die dadurch notwendigen Neuübertragungen möglichst zu verhindern, kann eine Station nicht zu beliebiger Zeit mit dem Senden eines Segmentes beginnen. Dem Kanal wird ein sogenannter „virtueller Kanalzustand“ zugeordnet. Alle Stationen hören ständig auf die Kommunikation auf dem Kanal und verfolgen so den Kanalzustand. In Abbildung 2.10 ist eine typische Folge von Kanalzuständen dargestellt.

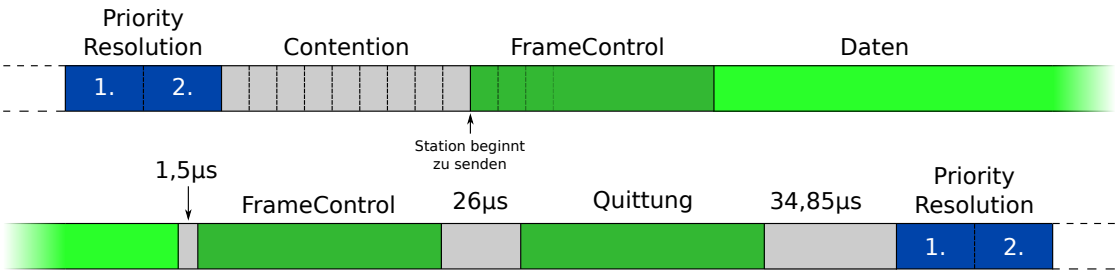


Abbildung 2.10: Abfolge von virtuellen Kanalzuständen

Den Beginn einer Übertragung stellt die *Priority-Resolution*-Phase dar. Diese Phase ist in zwei Hälften geteilt. Je nach Priorität des zu übertragenden Segmentes sendet eine Station in den beiden Hälften ein *Priority Resolution Symbol*. Die Zuordnung ist in Tabelle 2.2 gezeigt. Die vier Prioritätsebenen werden dabei binär kodiert. Stationen, die kein Symbol senden, hören auf gesendete Symbole. Wird erkannt, dass es eine Station mit einem Segment mit höherer Priorität gibt, wird das niederpriore Segment bis zum Ende der aktuellen Übertragung zurück gestellt (*defer*). Am Ende der Priority Resolution Phase sind nur noch Stationen aktiv, die ein Segment mit der aktuell höchsten Priorität senden möchten. Diese gehen nun in die *Contention*-Phase über [21].

Priorität	1. Hälfte	2. Hälfte
0	-	-
1	-	✓
2	✓	-
3	✓	✓

Tabelle 2.2: In der Priority-Resolsution-Phase gesendete Symbole, nach Priorität

Die Contention-Phase wird in einzelne *Slots* geteilt. Jede Station bestimmt beim Eintritt in die Contention-Phase zufällig, bei welchem Slot sie zu senden beginnt. Bis dieser Zeitpunkt erreicht wurde, hört die Station auf andere Übertragungen. Sollte eine Station zu senden beginnen, stellen alle anderen Stationen ihr Segment bis zum Ende der Übertragung zurück. Durch dieses als *Backoff* bezeichnete Verfahren werden Kollisionen reduziert. Sollte es dennoch zu einer Kollision kommen, weil zwei Stationen den gleichen Slot gewählt haben, wird bei der nächsten Übertragung das *Contention Window*, also die Menge der Slots aus denen gewählt wird, vergrößert. Dadurch wird eine Kollision unwahrscheinlicher.

Typischerweise gehören mehrere Semente zum gleichen Frame. Wenn zwei Stationen einen Frame mit der gleichen Priorität übertragen möchten, würden sich nach dem oben dargestellten Verfahren die Segmente der beiden Frames auf dem Kanal mischen, da beide Stationen immer wieder abwechselt die Contention-Phase „gewinnen“. Wenn ein Empfänger nicht genügend Speicher hat, um zwei Frames parallel zu empfangen, gehen möglicherweise Daten verloren. Zusätzlich reduziert die Wartezeit in der Contention-Phase die Nutzdatenrate. Um diese zu verhindern, kann eine Station während der Übertragung signalisieren, dass nach dem aktuellen Segment ein weiteres folgt. In diesem Fall folgt nach der Priority-Resolution-Phase keine Contention-Phase, sondern die Station beginnt direkt mit der Übertragung des nächsten Segmentes. Segmente eines Frames mit einer höheren Priorität haben dennoch Vorrang, da die Priority-Resolution-Phase weiterhin ausgeführt wird. Für Frames der Priorität 3 ist es sogar möglich, Semente von mehr als einem Frame ohne Contention-Phase zu senden, jedoch nie mehr als sieben am Stück [9].

Die Übertragung eines Segmentes beginnt mit der Übertragung eines *Frame-Control*-Headers[8] auf dem Kanal. Der Ablauf ist in Abbildung 2.10 zu sehen. Der Header enthält die Dauer des folgenden Segmentes in Form der Anzahl von OFDM-Symbolen. Er wird immer mit BPSK-Modulation gesendet und kann daher von allen Stationen empfangen werden. Danach folgt die angegebene Anzahl an OFDM-Symbolen mit dem eigentlichen Inhalt des Segmentes. Nach einer Pause von 1,5 µs wird ein weitere Frame-Control-Header als Endmarkierung gesendet. Dieser kann wieder von allen Stationen empfangen werden und dient die Synchronisation des virtuellen Kanalzustandes. Sollte eine Quittung erforderlich sein, wird diese 26 µs nach dem Ende des Frame-Control-Headers gesendet. Nach weiteren 34,85 µs beginnt dann die nächste Priority-Resolution-Phase.

Frame-Control-Header können immer von allen Stationen empfangen werden. Diese enthalten alle Informationen, die notwendig sind, um den virtuellen Kanalzustand zu synchronisieren. Durch diesen Mechanismus können Stationen erkennen, wie lange der Kanal belegt sein wird, auch wenn sie den Inhalt der Übertragung aufgrund einer falschen Tonemap nicht dekodieren können.

Auf Empfängerseite müssen die einzelnen Segment wieder zu Frames zusammengesetzt werden. Da, wie oben beschrieben, möglicherweise mehrere Frames ineinander verwoben gesendet werden, ist es vorteilhaft, mehr als einen Frame gleichzeitig zusammenfügen zu könen. In der Spezifikation ist dies jedoch nicht zwingend gefordert [9].

[8] trotz des Namens ist dies der Beginn eines *Segmentes*

Wurde ein Frame vollständig empfangen, werden die darin möglicherweise enthaltenen Mac-Managment-Informationen verarbeitet. Die Nutzdaten werden über das LLC-Interface weitergegeben. Für einige Mac-Managment-Einträge ist eine Antwort erforderlich. Diese wird, wenn notwendig, durch den MAC-Layer automatisch gesendet. Die LLC-Schicht ist daran nicht beteiligt.

3 Implementierung

Im Folgenden wird die Implementierung des HomePlug-1.0.1-kompatiblen MAC-Layers ausführlich vorgestellt. Es werden zunächst in Abschnitt 3.1 die Anforderungen an das System analysiert und dann in Abschnitt 3.2 ein Überblick über das Gesamtsystem gegeben. In Abschnitt 3.3, Abschnitt 3.4 und Abschnitt 3.5 wird dann auf die Implementierung der Softcores sowie der Speicheranbindung eingegangen.

3.1 Anforderungen

Ziel dieser Arbeit ist es, einen standardkonformen MAC-Layer des HomePlug-1.0.1-Standards zu implementieren. Die Zielplattform ist ein Kintex 7 410TIFFV900 FPGA von Xilinx. Die Kintex-7-Familie verwendet LUTs mit sechs Eingängen. Das verwendete FPGA verfügt über 406 720 Logikelemente, 1540 DSP-Blöcke und 795 BlockRAM-Elemente mit jeweils 36 kbit [15]. Das System soll mit einer einzigen Taktquelle von 50 MHz betrieben werden. Diese Anforderung ergibt sich durch das analoge Frontend, welches 50 MSample/s erfasst und ein Sample pro Takt verarbeitet.

Zum Zeitpunkt dieser Arbeit bestand bereits eine Implementierung des MAC-Layers als Hardwarebeschreibung für das oben genannte FPGA [14]. In dieser bestehenden Version wurde der MAC-Layer vollständig als Zustandsautomat in Hardware beschrieben. Es konnte dort ein Ergebnis erreicht werden, welches bei einem Takt von 50 MHz deutlich mehr als den benötigten Durchsatz von 14 Mbit/s erreicht. Diese Design benötigt 7845 LUTs, 5718 Flip-Flops, einen DSP-Block und 21 BlockRAMs. Es dient für diese Arbeit als Referenz.

Bei der in dieser Arbeit vorgestellten Implementierung werden Softcore-Prozessoren verwendet. Es soll dabei untersucht werden, ob sich durch den Einsatz dieser Vorteile im Bereich des Ressourcenverbrauchs und der Anpassbarkeit ergeben und welche Auswirkungen dies auf Durchsatz und Latenz des Systems hat. Es soll dabei vor allem der die Anzahl an Logikelementen (LUTs, Flip-Flops) optimiert werden. Hauptaugenmerk der

Implementierung ist daher ein möglichst geringer Bedarf an FPGA-Ressourcen (LUTs, FFs, BlockRAM).

Das in dieser Arbeit vorgestellt Hardware-Konzept ist von der bereits in [14] bestehenden Implementation unabhängig. Die HomePlug-Spezifikation schreibt nur die Beschaffenheit der Schnittstellen vor, der innere Hardware-Aufbau ist nicht spezifiziert und wurde für diese Arbeit neu entworfen.

3.2 Überblick

Wie in Abschnitt 2.4.1 dargestellt hat der MAC-Layer des HomePlug-Standards zwei Schnittstellen. Die „obere“ Schnittstelle führt zur LLC-Schicht, die „untere“ zur PHY-Schicht. Ausgehenden Daten nimmt der MAC-Layer von der LLC-Schnittstelle entgegen, verarbeitet sie und gibt sie auf der PHY-Schnittstelle aus. Eingehende Daten nehmen den umgekehrten Weg. Die PHY-Schicht entspricht dabei der Schicht 1 des OSI-Referenzmodells, MAC- und LLC-Schicht den Schichten 2a und 2b [16].

Ein- und ausgehende Daten des PHY-Layers werden dabei jeweils in einem Speicher innerhalb des MAC-Layers gepuffert. Wie in Abbildung 3.1 zu sehen ist, sind im System neben diesen beiden Speichern zwei Softcore-Blöcke enthalten. Diese werden als *Top Softcore* und *Bottom Softcore* bezeichnet. Diese stellen dabei die Kommunikationsschnittstelle zu LLC- bzw. PHY-Layer her. Diese Architektur ist durch das zweigeteilte Aufgabenfeld des MAC-Layers motiviert, das aus der direkten zeitkritischen Steuerung des PHY-Layers sowie der zeitlich unkritischeren Kommunikation mit der LLC-Schicht besteht. Die Kommunikation mit dem PHY-Layer erfordert definierte Antwortzeiten im µs-Bereich. Die Komplexität des Kontrollflusses in diesem Bereich ist gering. Die Kommunikation mit der LLC-Schicht beinhaltet einen komplexen Kontrollfluss. Die Antwortzeiten unterliegen aber keiner harten Beschränkung. Die Kommunikation zwischen den beiden CPUs erfolgt über einen geteilten Speicher sowie Interrupts.

Der Datenaustausch der Softcores mit den verschiedenen Speichern wird dabei über sogenannte „Streams“ realisiert. Diese abstrahieren den Speicher durch ein FIFO-Interface und können während der Datenübertragung noch weitere Funktionen, wie das Berechnen von Prüfsummen oder Multiplexing von verschiedenen Datenquellen, erledigen. Die Implementierung dieser Funktionen erfolgt in Hardware und ermöglicht einen hohen Datendurchsatz.

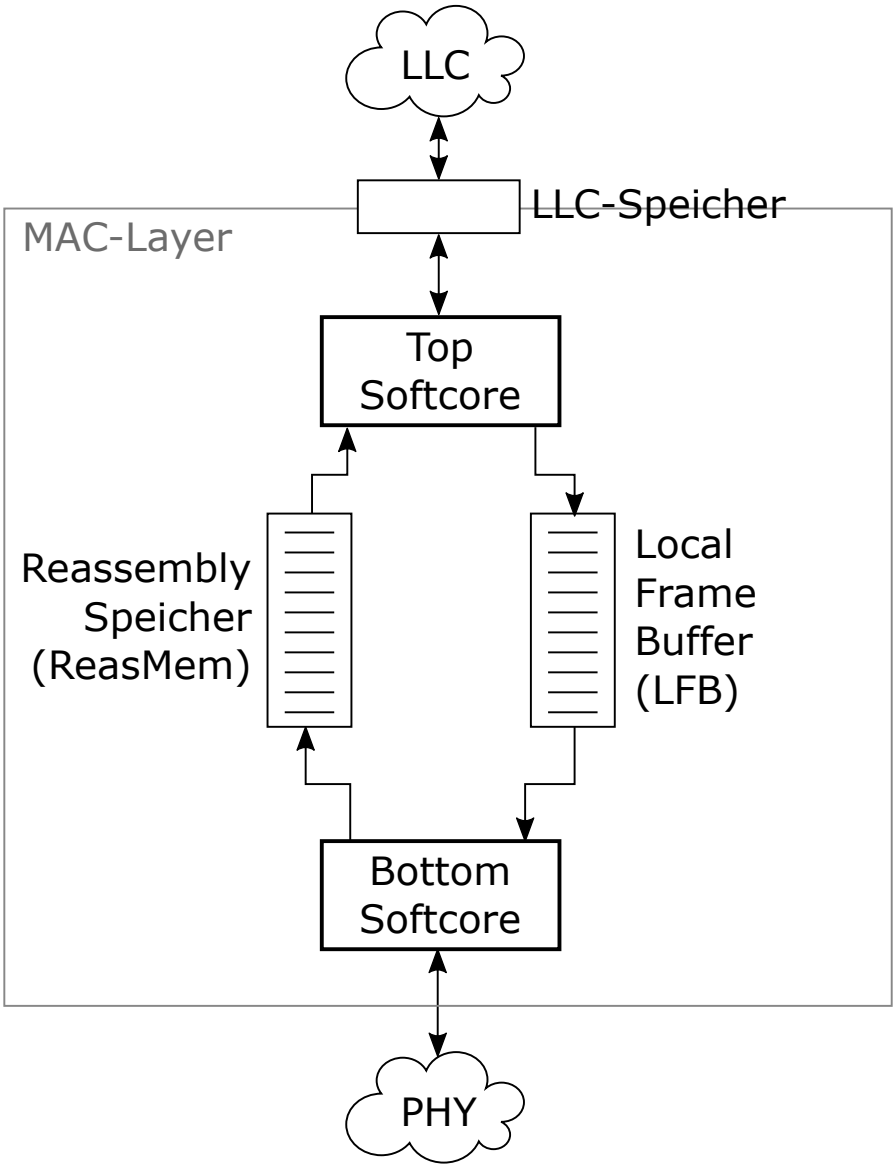


Abbildung 3.1: Überblick über das Gesamtsystem

Ausgehende Frames werden von der LLC-Schicht im gemeinsamen Speicher abgelegt. Der Top Softcore verarbeitet den Frame und reichert ihn mit den zugehörigen Metadaten an. Beispielsweise werden Anfragen zur Kanalschätzung in den Frame eingefügt, wenn keine aktuellen Informationen zum Kanalzustand (Tonemap) vorliegen. Der vollständige Frame wird dann in den Local Frame Buffer (LFB) kopiert. Eingehende Frames werden im Reassembly-Speicher (ReasMem) abgelegt. Von dort werden sie vom Top Softcore gelesen und verarbeitet. Bei der Verarbeitung eingehender Frames werden beispielsweise Anfragen zur Kanalschätzung erkannt und mit der entsprechenden Tonemap beantwortet. Sofern kein Übertragungsfehler vorliegt, werden die Nutzdaten des Frames zur weiteren Verarbeitung durch den LLC-Layer in den gemeinsamen LLC-Speicher kopiert.

Der Bottom Softcore beschäftigt sich mit der Übertragung einzelner Segmente. Dabei werden die notwendigen Metainformationen (FrameControl-Informationen) erzeugt. Für eingehende Segmenten werden automatisch die passenden Quittungen generiert und gesendet. Wenn notwendig, werden fehlerhafte Segmente automatisch wiederholt.

Es soll nun zunächst auf die Implementierung des Top Softcores eingegangen werden. Danach werden LFB und ReasMem sowie ihre Anbindung an die Softcores betrachtet. Anschließend wird die Implementierung des Bottom Softcores vorgestellt.

3.3 Der Top Softcore

Der Top Softcore bildet die Schnittstelle zur LLC-Schicht. Abbildung 3.2 zeigt diesen mit seiner Anbindung an Speicher und Streams. Die Kommunikation mit der LLC-Schicht erfolgt über einen 8 kB großen Speicher, der von LLC und MAC gemeinsam genutzt wird. Ein- und ausgehenden Daten-Frames werden in diesem Speicher abgelegt. Weiterhin hat der Top Softcore Zugriff auf den 2 kB großen LFB und den 8 kB großen ReasMem. Der Zugriff auf den LFB erfolgt dabei durch den CRC Stream TX, der Zugriff auf den ReasMem durch den CRC Stream RX. Auf diese wird in Abschnitt 3.4.1 bzw. Abschnitt 3.4.2 genauer eingegangen. Zu den weiteren Funktionen gehört die Verwaltung von Tonemaps und das Erzeugen von Segment-Headern. Dazu ist der Top Softcore mit dem Tonemap-Speicher und dem HeaderMem verbunden. Im Tonemap-Speicher werden die Kanalparameter für alle Gegenstellen gespeichert. Der HeaderMem enthält den Header für das nächste zu übertragende Segment. Zur Beschleunigung einiger Berechnung kann der Softcore auf eine Tabelle mit vorberechneten Segmentgrößen zugreifen. Dieser Capacity ROM wird in Abschnitt 3.3.2 beschrieben.

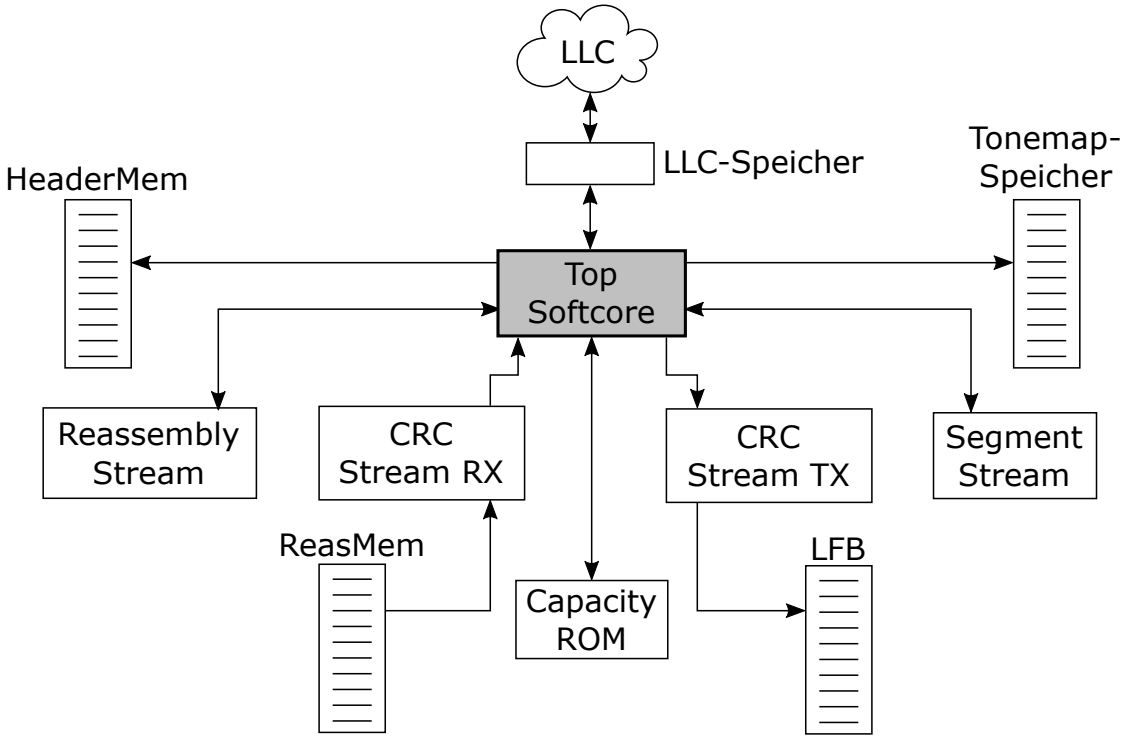


Abbildung 3.2: Die Verbindungen des Top Softcores

Die Übertragung einzelner Segmente ist die Aufgabe des Bottom Softcores. Die Steuerung von Segmentierung und Wiederzusammenstellung (Reassembly) sowie das Erzeugen von Segment-Headern im HeaderMem werden vom Top Softcore durchgeführt. Dazu ist er mit dem Segment Stream und dem Reassembly Stream verbunden. Der Top Softcore trifft hierbei nur die Entscheidung, welches Segment als Nächstes zu übertragen ist bzw. wo ein eingehendes Segment abgespeichert wird. Informationen zu Segment Stream und Reassebly Stream finden sich in Abschnitt 3.4.3 und Abschnitt 3.4.4. Der Bottom Softcore wird in Abschnitt 3.5 beschrieben.

Der Top Softcore steuert Übertragung, Segmentierung, Wiederzusammenstellung und Empfang von *Frames*. Er greift zur Übertragung einzelner *Segmente* auf die Funktionen des Bottom Softcores zu. Im Folgenden wird nun die Implementierung des Top Softcores betrachtet.

3.3.1 Implementierung

Für die Implementierung des Top Softcore wurde ein NEO430 gewählt. Es handelt sich um eine MSP430-kompatible 16-Bit-Architektur. Durch die Kompatibilität auf Instruktionssatzebene kann der vorhandene GCC-Compiler für die MSP430-Architektur verwendet werden (vgl. Abschnitt 2.1.1).

Wie in Tabelle 3.1 zu sehen ist, ist er durch seinen geringen Ressourcenverbrauch von nur rund 650 LUT6 und seine einfache Erweiterbarkeit gut geeignet (vgl. Abschnitt 2.1.1). Andere Softcores, wie etwa der achtbittige *AVR_Core* oder der ebenfalls MSP430-kompatible *openMSP* benötigen bei gleicher oder geringerer Funktionalität ähnlich viel oder mehr Ressourcen [10].

Name	#LUT6	Architektur	Erweiterbarkeit
AVR_Core	≈ 465[1]	AVR (8 bit)	gering
NEO430	≈ 650	MSP430 (16 bit)	hoch
openMSP	≈ 1650	MSP430 (16 bit)	hoch

Tabelle 3.1: Vergleich verschiedener Softcores

Für die Implementation des HomePlug-Standards ist eine 16-Bit-Architektur vorteilhaft, da die zu verbeitenden Frames eine maximale Größe von von 1609 Bytes haben (vgl. Abschnitt 2.4.1). Im weiteren Verlauf der Arbeit hat sich die geringe Compiler-Unterstützung der MSP430-Architektur und die Multi-Cycle-Architektur des NEO430 als Herausforderungen bei der Entwicklung herausgestellt (vgl. Abschnitt 3.5.2).

Für die Verschlüsselung der Frames und das Backoff-Verfahren des HomePlug-Standards ist ein Zufallszahlengenerator notwendig. Der NEO430 bietet keinen eingebauten Zufallszahlengenerator, daher wurde ein einfacher Linear Feedback Shift Register (LFSR)-Generator verwendet [18], da der HomePlug-Standard nicht zwingend die Verwendung eines kryptografisch sicheren Generators fordert („Values returned by Random() shall be statistically independent to the greatest degree possible.“, [9, 3.4.5]). Für den Start des Zufallszahlengenerators wird ein Startwert (Seed) benötigt. Dieser sollte eine möglichst hohe Entropie aufweisen. Es wurde an dieser Stelle das letzte Byte der MAC-Adresse verwendet, da keine echte Zufallsquelle zur Verfügung stand. In einer weiteren Überarbeitung wäre es möglich, einen Ringozillator oder eine andere Hardware-basierte Entropiequelle zu verwenden [17].

[1] geschätzt, 1860 LUT4

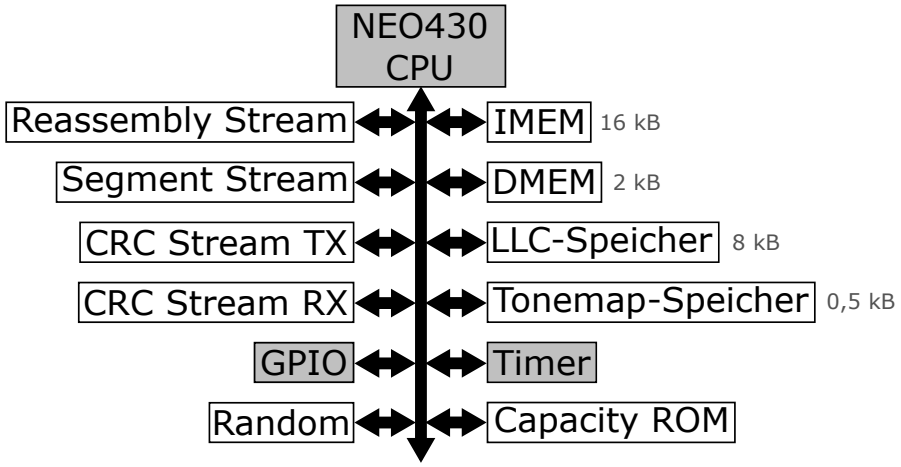


Abbildung 3.3: Struktur des NEO430 mit allen verwendeten Modulen. Grau hinterlegte Module sind Teil des Standardumfangs.

Der Softcore wurde mit 16 kB Instruktionsspeicher (IMEM) und 2 kB Datenspeicher (DMEM) konfiguriert. Die Interrupt-Funktion der CPU ist aktiv. Von den mitgelieferten Peripheriemodulen wurden nur GPIO- und Timer-Modul verwendet. Alle weiteren Module wurden direkt mit dem CPU-Bus verbunden, statt sie über den Wishbone-Bus anzubinden. Dies vereinfach den Zugriff auf die Module durch die Software und vermeidet den Overhead durch die Wishbone-Kommunikation. Die Struktur des NEO430 mit den Erweiterungsmodulen ist in Abbildung 3.3 zu sehen.

Die vom Top Softcore zu sendenden Frames können entweder von der LLC-Schicht oder vom Softcore selbst stammen[2]. Je nach Quelle sind die Nutzdaten im LLC-Speicher oder im Hauptspeicher des Softcores zu finden. Der LLC-Speicher ist direkt in der Adressraum der CPU eingeblendet. Auf diese Weise ist eine Unterscheidung in der weiteren Verarbeitung nicht notwendig. Eine Anbindung des LLC-Speichers über den Wishbone-Bus hätte eine Fallunterscheidung notwendig gemacht.

Die eigentlichen Nutzdaten eines Frames (Payload) werden bei der Übertragung in den LFB bzw. aus dem ReasMem kopiert. Da üblicherweise der größte Anteil eines Frames Nutzdaten sind, ist eine Optimierung hier besonders sinnvoll. Es wurden, wie in Tabelle 3.2 gezeigt, verschiedene Versionen implementiert. In der ersten Version der Software wurde das Kopieren durch eine einfache `while`-Schleife, wie in Listing 3.1 gezeigt, realisiert. Durch den Zugriff über ein FIFO-Interface auf LFB und ReasMem kann die übliche `memcpy`-Funktion nicht verwendet werden. Diese Version hat einen hohen Overhead,

[2] Beispielsweise werden Anfragen zur Kanalschätzung automatisch beantwortet.

Methode	Laufzeit (µs)	Speedup
`while`-Schleife	158	1,00×
Duff's-Device	122	1,30×
Inline-Assembler	47	3,36×

Tabelle 3.2: Laufzeiten verschiedener Varianten der Kopierschleife. Die Test wurden mit einer Datenmenge von 201 Bytes ausgeführt.

da pro Datenbyte der Schleifenzähler überprüft wird. Eine bekanntes Verfahren zur Optimierung ist das Abrollen einer Schleife. Hierbei wir der Schleifenkörper in jedem Durchlauf mehr als einmal ausgeführt bevor der Schleifenzähler überprüft wird. Ein Problem hierbei sind Schleifen deren Durchlaufzahl kein Vielfaches des Abrollfaktors ist [12]. Eine Lösung hierfür ist das „Duff's Device", bei dem in der ersten Iteration in die Mitte der Schleife gesprungen wird [11]. Dies ist in der Programmiersprache C möglich. Listing 3.2 zeigt eine entsprechende Implementierung. Dieser Code wird allerdings durch den gcc-Compiler für MSP430 in ein ineffizientes Programm umgesetzt, sodass sich nur eine geringe Verbesserung ergibt. Schlussendlich wurde eine auf dem Duff's Device basierende Variante in Inline-Assembler implementiert. Der zugehörige Code ist in im Anhang in Listing A.1 zu finden. Durch diese Variante konnte die Laufzeit der Schleife um mehr als Faktor drei reduziert werden.

```
while(count--) {
        CRC_TX_DATA = *from++;
}
```

Listing 3.1: Kopieren mit `while`-Schleife

Eine weitere Herausforderung bei der Erstellung der Software für den Top Softcore ist der FIFO-Zugriff auf den ReasMem durch den CRC Stream RX. Durch die Stream-Architektur ist das Lesen der Prüfsumme am Ende des Frames erst möglich, wenn alle anderen davor liegenden Daten verarbeitet wurden. Ein Zurückspringen zu den Inhalten davor ist nicht ohne Weiteres möglich. Eine Benachrichtigung an die LLC-Schicht darf nicht erfolgen, wenn die Prüfsumme inkorrekt ist, da in diesem Fall die empfangengen Daten möglicherweise beschädigt sind. Um diesem Problem zu lösen, werden alle Nachrichten an die LLC-Schicht, die während der Verarbeitung des Paketes entstehen, zwischengespeichert. Sollte sich nach dem Lesen aller Daten herausstellen, dass die Prüfsumme falsch ist, werden die Nachrichten verworfen, andernfalls werden sie alle nacheinander zugestellt.

```
int n = (count+7)/8;
switch (count%8) {
        case 0: do { CRC_TX_DATA = *from++;
        case 7:      CRC_TX_DATA = *from++;
        case 6:      CRC_TX_DATA = *from++;
        case 5:      CRC_TX_DATA = *from++;
        case 4:      CRC_TX_DATA = *from++;
        case 3:      CRC_TX_DATA = *from++;
        case 2:      CRC_TX_DATA = *from++;
        case 1:      CRC_TX_DATA = *from++;
        } while (--n>0);
}
```

Listing 3.2: Kopieren mit „Duff's Device“

Der Top Softcore benötigt verschiedene Timer. So muss etwa für jedes eingehende Segment ein Timeout überwacht werden oder eine ausgehende Übertragung abgebrochen werden, wenn zu viel Zeit vergangen ist. Wie in Tabelle 3.3 zu sehen ist, werden insgesamt acht Timer benötigt, der NEO430 verfügt aber nur über einen einzelnen Hardware-Timer. Es wäre möglich, weitere Timer in Hardware zu implementieren, dies würde den Verbrauch an Logikelementen erhöhen. Die verschiedenen Timer werden daher in Software simuliert. Der Hardware-Timer des NEO430 wird so konfiguriert, dass er alle 25 ms einen Interrupt auslöst. In diesem Interrupt werden dann Ereignisbehandlungsroutinen aller Module der Software aufgerufen. Dies zählen den Wert einer Variablen herunter und lösen eine Aktion aus, wenn sie den Wert Null erreicht. Auf diese Weise können nur Zeitspannen dargestellt werden, die Vielfache von 25 ms sind, was aber für den aktuellen Anwendungsfall kein Problem darstellt. Es können mit dieser Methode nahezu beliebig viele Timer realisiert werden, ohne den Verbrauch an FPGA-Ressourcen zu erhöhen.

Name	Anzahl	Maximale Zeit
Transmit Timeout	1	$2^{16} * 25\,\text{ms}$
Receive Timeout	5	1 s
Link Status Timer	1	5 s
Fail Wait Delay	1	25 ms

Tabelle 3.3: Im System verwendete Timer

Besondere Beachtung verdient der Fail-Wait-Timer: Für diesen ist in der Spezifikation ein Wert von 10 ms vorgesehen. Dieser Wert wurde auf 25 ms erhöht. Dies verringert

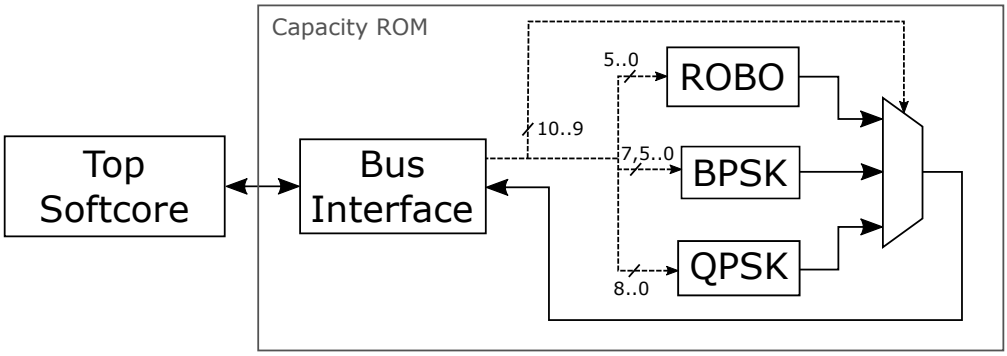


Abbildung 3.4: Capacity ROM

die Interruptbelastung des Softcores durch den Timer. Diese Änderung verursacht keine Probleme mit bestehenden Empfängern.

3.3.2 Capacity ROM

Zur Erzeugung ausgehender Segmente muss die Menge an Nutzdaten pro Segment bestimmt werden. Diese hängt von der verwendeten Modulationsart, der Fehlerkorrekturrate und der Anzahl der zur Verfügung stehenden Träger ab (vgl. Abschnitt 2.4.1). Die Berechnung der Nutzdatenmenge benötigt Divisionen und Multiplikationen[3]. Da die eingesetzte CPU über keinen Hardware-Multiplizierer verfügt, ist die Berechnung in Software sehr zeitaufwendig. Zur Beschleunigung wird daher ein an die CPU angebundener ROM-Speicher eingesetzt, in dem vorberechnete Werte für alle sinnvollen Kombinationen von Modulationsart, Fehlerkorrekturrate und Trägeranzahl vorhanden sind.

Abbildung 3.4 zeigt eine Übersicht über den Capacity ROM. Der ROM-Speicher besteht aus drei einzelnen Teilen (Seiten) mit unterschiedlichen Daten- und Adressbreiten. Der Zugriff erfolgt über ein einziges Register. Die CPU muss dabei die gewünschte Adresse in das Register schreiben. Die oberen beiden Bits der Adresse wählen dabei die Seite aus, die restlichen Bits das entsprechende Wort im ROM-Speicher. Danach kann das Register gelesen werden, um den angeforderten Wert aus dem ROM-Speicher zu erhalten. Je nach ausgewählter Seite werden mehr oder weniger Bits der Adresse verwendet.

Die Aufteilung in drei Seiten trägt dem Zustand Rechnung, dass nicht alle Kombinationen von Modulationsart, Fehlerkorrekturrate und Trägeranzahl gültig sind. Würde ein einzelner ROM-Speicher verwendet, müssten auch für die ungültigen Kombinationen Werte

[3] Die Berechnungs ist im Anhang in Abschnitt A.3 beschrieben.

Modulationsart	Blockgröße	Fehlerkorrekturrate	Anzahl Träger
ROBO	40	1/2	32 bis 84
BPSK	20	1/2	32 bis 84
	40	1/2	32 bis 84
QPSK	20	1/2	16 bis 84
		3/4	11 bis 84
	40	1/2	16 bis 84
		3/4	11 bis 84

Tabelle 3.4: Gültige Kombinationen von Kanalparametern. Nur 445 von 2048 sind möglich (21,7 %).

gespeichert werden, was unnötig viele Ressourcen verbrauchen würde. Alle gültigen Kombinationen sind in Tabelle 3.4 angegeben. Der indirekte Zugriff über das Registerinterface reduziert die Auslastung des Adressraums des Top Softcores. Würde der ROM-Speicher direkt in den Adressraum der CPU eingeblendet, würden 2048 Speicheradressen der CPU alleine für den Zugriff auf den ROM-Speicher benötigt, was ca. 3 % des gesamten Adressraums entspricht.

Der eingesetzte ROM-Speicher ist identisch mit dem im PHY-Layer verwendeten. Sollten PHY- und MAC-Layer auf einem einzelnen Chip implementiert werden, ist es es unter Umständen möglich, diesen Speicher gemeinsam zu nutzen und so weitere Ressourcen zu sparen.

Ein wesentliches Merkmal des MAC-Layer-Designs stellt die Verbindung zwischen zu den Speichern über sog. „Streams“ dar. Diese werden nun genauer beschrieben.

3.4 Anbindung der Speicher

Zentraler Bestandteil des Entwurfs sind die Speicher für ausgehende (LFB) und eingehende (ReasMem) Segmente. Der Zugriff auf diese Speicher wird durch FIFO-Schnittstellen abstrahiert. Dadurch ist es möglich, während der Übertragung eine zusätzliche Verarbeitung der Daten vorzunehmen, und beispielsweise die Prüfsumme zu berechnen. In Abbildung 3.5 sind die beiden Speicher und die Streams dargestellt. Der Top Softcore kann durch den CRC Stream TX in den LFB schreiben und durch den CRC Stream

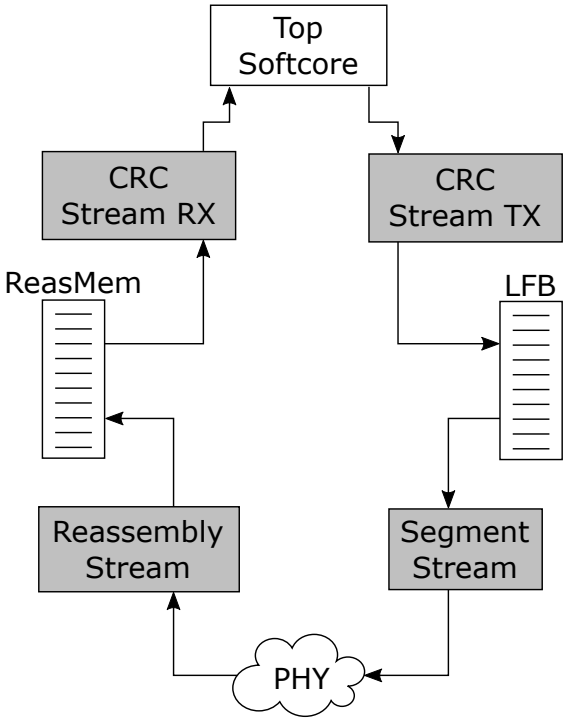


Abbildung 3.5: LFB und ReasMem mit Streams

RX aus dem ReasMem lesen. Der PYH-Layer ist über Segment Stream und Reassembly Stream mit diesen beiden Speichern verbunden.

Nachfolgend werden zunächst CRC Stream TX und RX betrachtet, danach wird auf Segment Stream und Reassembly Stream eingegangen.

3.4.1 CRC Stream TX

Aufgabe des CRC Stream TX ist es, für den Top Softcore ein FIFO-Interface zum LFB bereitzustellen. Eine Übersicht über den CRC Stream TX ist in Abbildung 3.6 dargestellt. Im LFB werden ausgehenden Frames zwischengespeichert. Während der Übertragung können die Daten optional verschlüsselt werden. Zusätzlich wird die Prüfsumme der übertragenen Daten berechnet.

Durch die Stream-Architektur erfolgen Verschlüsselung und Prüfsummenberechnung für die CPU transparent während der Übertragung der Daten. Es ist nicht notwendig, die Daten dafür erneut vollständig einzulesen, was sich positiv auf Datendurchsatz und Latenz auswirkt.

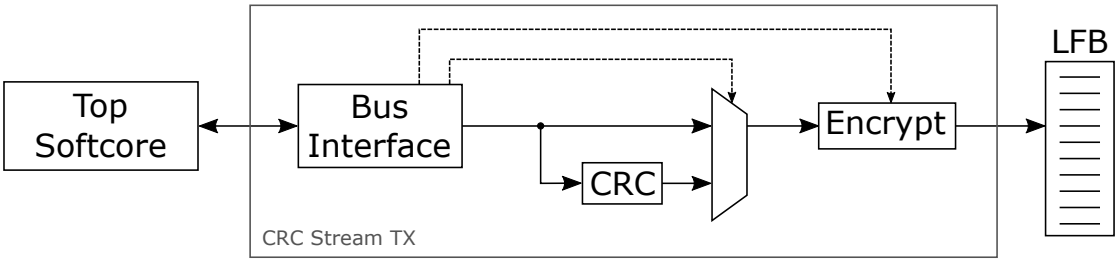


Abbildung 3.6: CRC Stream TX

Nach der Übertragung der Daten vom Top Softcore kann dieser den CRC Stream TX veranlassen, die berechnete Prüfsumme in den LFB zu schreiben, wie in Abbildung 3.6 durch den Multiplexer dargestellt. Diese Konstruktion vermeidet die doppelte Übertragung der Prüfsumme zwischen CRC Stream TX und CPU und vereinfacht zusätzlich die Schnittstelle zwischen den beiden.

Die im LFB gespeicherten Daten enthalten Frame-Header, Mac-Managment-Informationen, Payload, EPAD und FCS und sind ggf. verschlüsselt[4] (vgl. Abschnitt 2.4.1). Eine Aufteilung in Segmente erfolgt jedoch noch nicht.

Da die Verschlüsselung der Daten blockorientiert erfolgt, ist es notwendig jeweils acht Byte zu puffern und gesammelt zu verschlüsseln. Diese Aufgabe wurde in das Encryption-Modul verlagert. Im Rahmen dieser Arbeit wurde das Encryption-Modul nicht implementiert. Es ist jedoch eine entsprechende Schnittstelle vorhanden sodass sich diese Modul, falls notwendig, leicht ergänzen lässt. Das Konfiguration von Schlüssel und Initialisierungsvektor für die Verschlüsselung erfolgt ebenfalls über den CRC Stream TX. Um das Interface zwischen CPU und CRC Stream TX klein zu halten gibt es nur einen gemeinsamen Registersatz für IV und Schlüssel[5]. Die CPU legt den entsprechenden Wert in den Registern ab und veranlasst dann die Verschlüsselungseinheit den Inhalt als Schlüssel oder IV zu laden, indem ein dafür vorgesehenes Bit im Konfigurationsregister des CRC Stream TX gesetzt wird.

Die Prüfsumme ist eine CRC16-Prüfsumme (vgl. Abschnitt 2.4.1 und Abschnitt 2.2). Sie ist so ausgelegt, dass jeweils ein Byte gleichzeitig verarbeitet werden kann, da das Interface zur Softcore byte-orientiert ist. Zur Implementierung wird die in Abschnitt 2.2.2 beschriebene XOR-Technik verwendet.

[4] sie entsprechen damit dem TX Encrypted Service Block (TESB) aus der HomePlug-Spezifikation [9, C.4.2 ff]

[5] Da IV und Schlüssel 64 bit lang sind, werden vier 16 bit-Register benötigt.

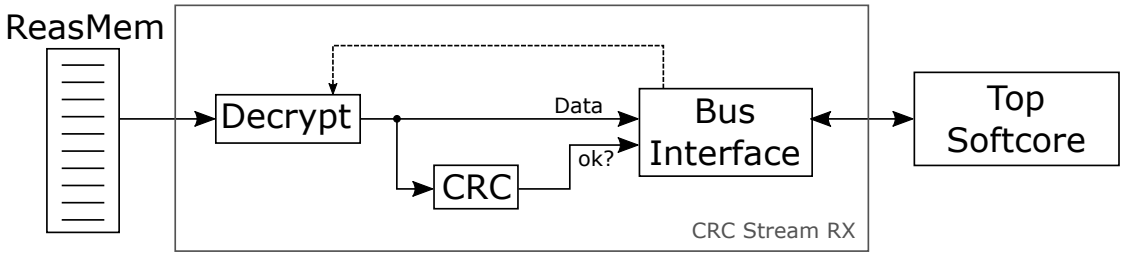


Abbildung 3.7: CRC Stream RX

Der LFB nimmt ausgehende Frames auf. Eingehende Frames werden aus dem ReasMem gelesen, der über den CRC Stream RX angebunden ist.

3.4.2 CRC Stream RX

Der CRC Stream RX bietet der CPU ein FIFO-Interface für den ReasMem an. Abbildung 3.7 zeigt eine Übersicht des CRC Stream RX. Der ReasMem nimmt eingehenden Segmente auf und speichert diese bis alle Segmente eines Frames empfangen wurden. Während der Übertragung der Daten zur CPU werden diese ggf. entschlüsselt und ihre Prüfsumme wird berechnet. Er stellt damit in gewissem Maße die umgekehrte Funktionalität des CRC Stream TX bereit.

Auch an dieser Stelle ermöglicht es die Stream-Architektur, das Lesen der Daten und die Berchnung der Prüfsumme zu parallelisieren. Wenn die CPU alle Daten gelesen hat muss sie nur ein Bit im Kontrollregister des CRC Stream RX lesen, um festzustellen, ob die Prüfsumme korrekt war. Es ist nicht möglich, die berechnete Prüfsumme auszulesen, über das Bit im Kontrollregister kann nur festgestellt werden, ob sie korrekt war. Dies ist für den vorliegenden Anwendungsfall ausreichend und vereinfacht die Schnittstelle zwischen CPU und CRC Stream RX.

Im ReasMem werden die Segmente eines Frames vorgehalten, bis alle Segmente eines Frames empfangen wurden. Es können Segmente von mehr als einem Frame ineinander verwoben gesendet werden, daher ist es zweckmäßig, mehrere Frames parallel wiederzusammestellen zu können. Jeder Frame kann maximal 1609 Byte groß sein (vgl. Abschnitt 2.3). Um fünf Frames parallel speichern zu können, sind daher $5 * 1609\,\mathrm{B} = 8045\,\mathrm{B} \approx 8\,\mathrm{kB}$ notwendig. Mehr oder weniger Reassemble-Stations führen zu einem ungünstigen Verhältnis von benötigtem und vorhandenem Speicher[6], daher wurde ein Speicher mit 8 kB Kapazität implementiert.

[6] Speichermenge die keine Potenz von 2 sind, sind schlecht zu realisieren.

Die Entschlüsselung der Daten erfolgt – genau wie die Verschlüsselung – blockweise. Das dafür notwendige Puffern von jeweils acht Byte ist Aufgabe des Entschlüsselungsmoduls. Dieses Modul wurde im Rahmen dieser Arbeit nicht implementiert, kann aber leicht ergänzt werden. Die Kombination von Entschlüsselung von Prüfsummenberechnung wurde auch schon in [14] angedacht.

CRC Stream RX und TX verbinden die Speicher mit dem Top Softcore. Die Verbindung zum PHY-Layer wird durch Segment Stream und Reassembly Stream hergestellt.

3.4.3 Segement Stream

Der Segment Stream verbindet LFB und HeaderMem mit dem PHY-Layer. Abbildung 3.8 zeigt eine Übersicht des Segment Streams. Es wird während der Übertragung der Daten automatisch zwischen den Speichern, der Konstanten Null und der aktuellen Prüfsumme umgeschaltet. Von allen übertragenen Daten wird automatisch die Prüfsumme berechnet.

Die Übertragung einzelner Frames erfolgt im HomePlug-Standard nicht am Stück, sondern aufgeteilt in einzelne Segmente (vgl. Abschnitt 2.3). Jedes Segment ist dabei mit einem eigenen Header versehen und wird von einer eignen Prüfsumme geschützt. Es wäre möglich, vor der Übertragung des ersten Segments alle Segmente vorzuberechnen und in einem Speicher abzulegen[7]. Diese Art der Implementierung würde viel Speicher benötigen, da beispielsweise die Paddingbytes zum Auffüllen des letzten Symbolblockes mit gespeichert werden müssten. Außerdem kann es passieren, dass während der Übertragung die Modulationsart auf ROBO reduziert wird, falls die Übertragung wiederholt fehlschlägt. Wenn dies passiert ändert sich auch die Segmentgröße, sodass eine Neusegmentierung notwendig wäre, die viel Zeit kosten würde. In der vorliegenden Architektur werden die einzelnen Segmente daher nicht gespeichert, sondern während der Übertragung on-the-fly erzeugt.

Während des Lesens führt der Segment Stream dazu automatisch ein Multiplexing durch: Die ersten 17 Byte werden aus dem HeaderMem gelesen. In diesem Speicher hat der Top Softcore den Header des aktuellen Segmentes abgelegt. Nachdem das siebzehnte Byte gelesen wurde, schaltet der Segment Stream automatisch um und liefert Daten aus dem LFB. Wenn die Nutzdaten des Segmentes (Payload) durch den PHY-Layer gelesen wurden, erfolgt eine weitere Umschaltung, sodass nun Nullen als Padding geliefert werden. Wenn die benötigte Menge an Padding übertragen wurde, folgt die Prüfsumme. Für den

[7] Diese Art der Implementierung wurde in der bereits bestehenden Variante verwendet.

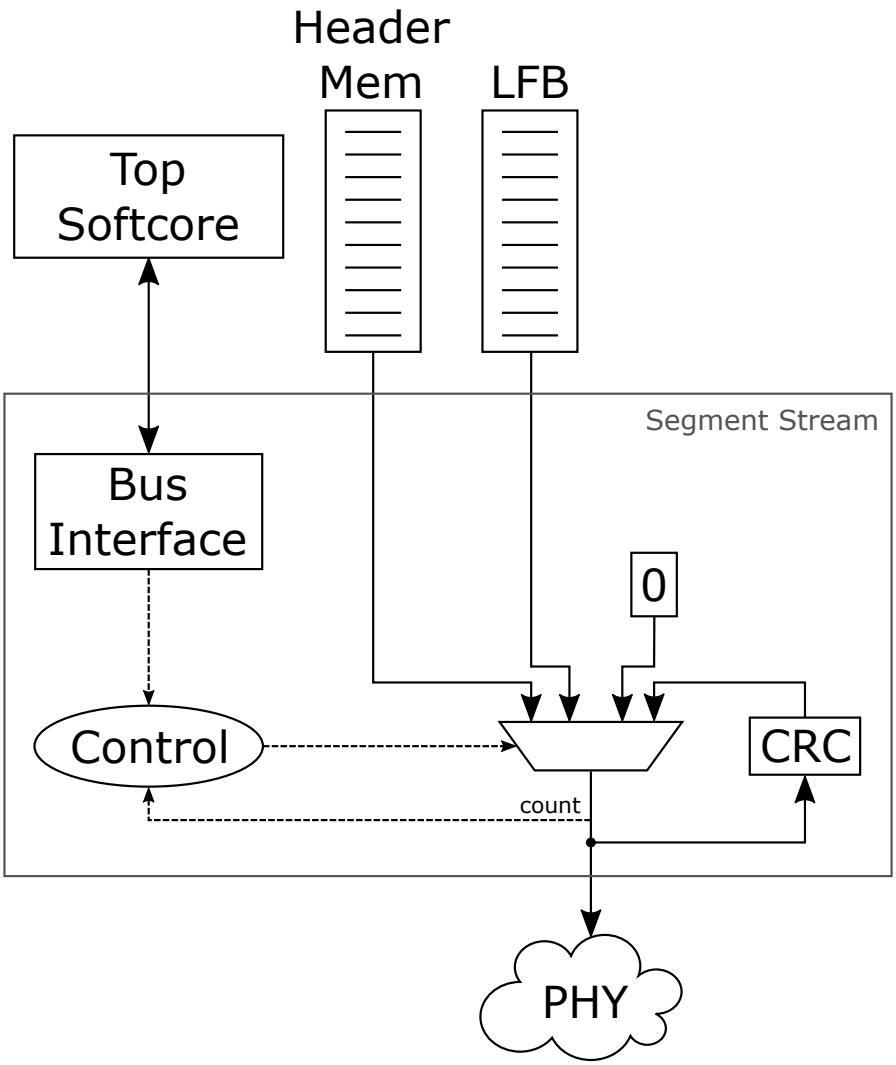


Abbildung 3.8: Segment Stream

PHY-Layer ist diese Umschaltungen unsichtbar. Dies wird als „virtuelle“ Segmentierung bezeichnet.

Die Steuerung der Segmentierung ist Aufgabe des *Top* Softcores, daher ist das Registerinterface des Segment Streams mit diesem verbunden. Der Ausgang des Segment Streams ist direkt mit dem PHY-Layer verbunden. Der Bottom Softcore ist bei der Segmentierung nicht involviert.

Der Datenausgang des Segment Streams ist über Handshake-Signale mit dem PHY-Layer verbunden. Die Lesegeschwindigkeit wird ausschließlich durch den PHY-Layer gesteuert. Da dieser bitseriell arbeitet, kann ein Byte alle acht Takte gelesen werden. Die Interruptlatenz des NEO430 beträgt bis zu 16 Takte, daher ist es erforderlich, dass die Umschaltung zwischen den Datenquellen in Hardware realisiert wird.

Länge und Position der Nutzdaten des Frames im LFB sowie die benötigte Menge an Paddingbytes werden durch den Top Softcore mit Hilfe des Registerinterfaces eingestellt. Sollte es notwendig sein ein Segment zu wiederholen, reicht es aus, den Lesezeiger im LFB entsprechend zurückzusetzen. Wenn ein Übergang in den ROBO-Modus notwendig ist und sich dadurch die Segmentelänge ändert kann der Top Softcore die Menge der zu lesenden Nutzdaten über das Registerinterface einfach anpassen. Durch diese „virtuelle“ Segmentierung ist kein Speicher notwendig, um die Segmente zu speichern und das Segmentieren erfordert keine Rechenzeit. Dies wirkt sich positiv auf den Verbrauch an FPGA-Speicherblöcken sowie Datendurchsatz des MAC-Layers aus. Als weiterer positiver Nebeneffekt ist das Umschalten in den ROBO-Modus problemlos möglich.

Neben der Segmentierung ausgehender Frames ist auch die Wiederzusammenstellung eingehender Frames notwendig. Dies ist die Aufgabe des Reassembly Streams.

3.4.4 Reassembly Stream

Der Reassembly Stream stellt die Verbindung zwischen PHY und ReasMem her. Abbildung 3.9 zeigt eine Übersicht über den Reassembly Stream. Die eingehenden Daten werden dabei zunächst in einem FIFO-Speicher gepuffert und dann an die passenden Stelle im ReasMem geschrieben. Alternativ können über das Registerinterface auch direkt Werte aus dem FIFO-Speicher gelesen werden. Der Reassembly Stream berechnet über alle eingehenden Daten automatisch eine Prüfsumme und kann einen Interrupt im Top Softcore auslösen.

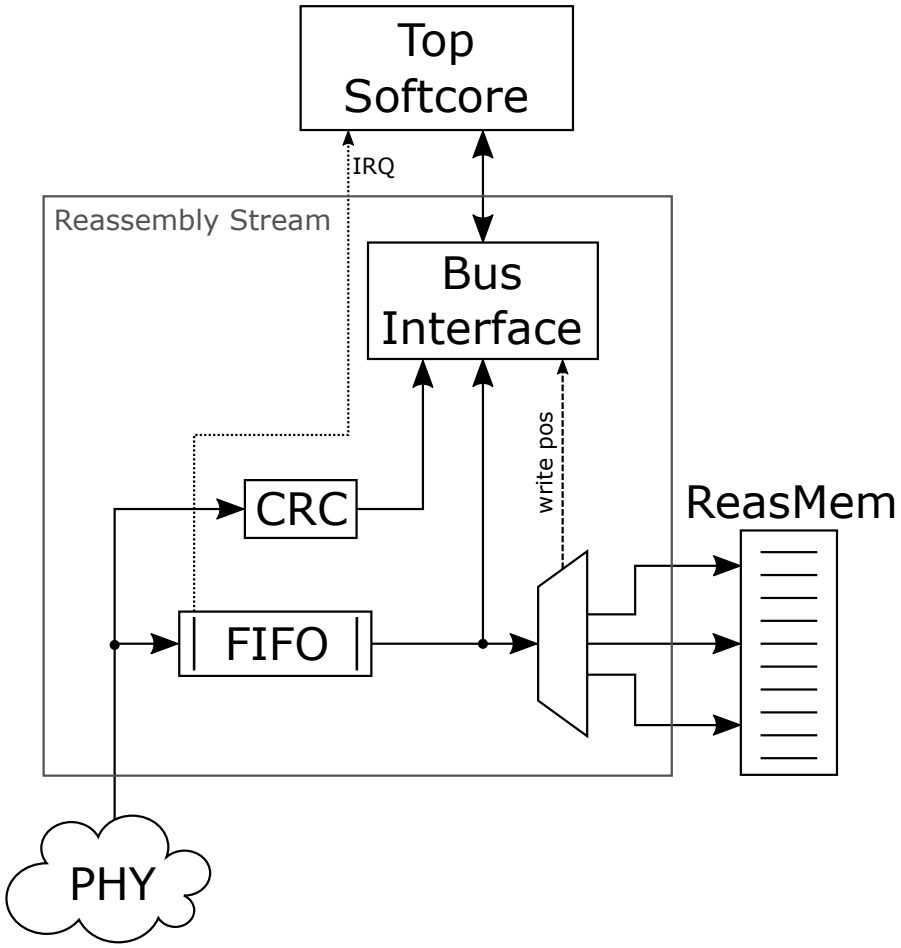


Abbildung 3.9: Reas Stream

Der Dateneingang des Reassembly Streams ist direkt mit dem PHY-Layer verbunden. Die eingehenden Daten landen zunächst in einem FIFO-Speicher. Wenn dieser Speicher 17 Byte enthält, also der Header des Segmentes vollständig empfangen wurde, wird ein Interrupt an den Top Softcore gesendet.

Der Top Softcore liest nun der Segmentheader aus dem FIFO-Speicher. Er legt dann fest, an welche Stelle im ReasMem das eingehende Segment gespeichert werden soll[8]. Diese Adresse wird in einem Register des Reassembly Stream abgelegt. Sobald diese Information vorliegt werden die Daten aus dem FIFO-Speicher in den ReasMem kopiert. Wenn der PHY-Layer signalisiert, dass die Übertragung abgeschlossen wurde, wird erneut ein Interrupt ausgelöst. Der Bottom Softcore ist bei der Reassemblierung der Segmente nicht involviert.

Die Schreibgeschwindigkeit der Daten in den FIFO-Speicher wird direkt vom PHY-Layer über Handshake-Signale gesteuert. Es kann aufgrund der bitseriellen Arbeitsweise des PHY-Layers maximal ein Byte alle acht Takte übertragen werden. Messungen haben ergeben, dass der Top Softcore 31 µs braucht, um zu entscheiden, an welche Stelle im ReasMem die Daten geschrieben werden sollen. Der Top Softcore beginnt mit der Entscheidung, wenn der 17 Byte große Segment-Control-Header empfangen wurden. Innerhalb der 31 µs können bis zu $31\,\mu s * 50\,MHz * \frac{1}{8}\,B = 188\,B$ übertragen werden. Zusammen mit den 17 Byte des Headers wird eine FIFO-Tiefe von $188\,B + 17\,B = 205\,B$ benötigt. Es wurde daher ein 256 Byte tiefes FIFO implementiert.

Sobald die Entscheidung über das Ziel der Daten vorliegt, beginnt der Reassembly Stream den FIFO-Speicher zu leeren und die Daten an ihr Ziel im ReasMem zu kopieren. Der PHY-Layer kann weiterhin parallel Daten in das FIFO schreiben. Da das Wegschreiben der Daten schneller ist, als die eingehenden Datenrate vom PHY-Layer[9] leert sich der FIFO-Speicher nach höchstens 165 Takten. Eingehenden Daten werden aber der Einfachheit halber auch dann weiter vom PHY-Layer im FIFO-Speicher abgelegt und dann von dort durch den Reassembly Stream in den ReasMem kopiert.

Der Header des Segmentes wird nicht in den ReasMem kopiert, da dieser von vom Top Softcore aus der FIFO gelesen wurde, bevor der Reassembly Stream beginnt, die Daten in den ReasMem zu kopieren. Wie in Abbildung 3.10 dargestellt, werden beim Kopieren der Daten auch die Padding-Bytes und die Prüfsumme, die auf die eigentlichen Nutzdaten

[8] Alternativ kann das Segment auch verworfen werden, wenn die Empfängeradress nicht passt oder der Header ungültig ist.

[9] Es kann ein Byte pro Takt weggeschrieben werden, der PHY-Layer kann aber nur ein Byte alle acht Takte produzieren.

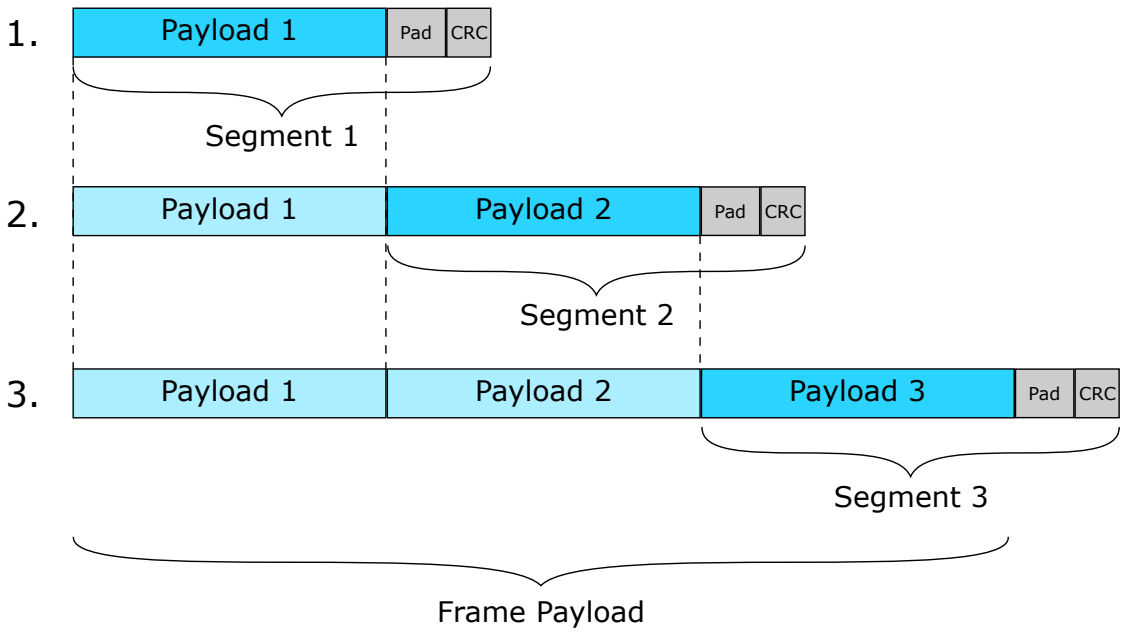


Abbildung 3.10: Grafische Darstellung der Speichervorgänge im ReasMem. Die nachfolgenden Segmente überschreiben Padding und Prüfsumme der vorhergehenden, sodass am Ende der Payload linear im Speicher liegt.

folgen, in den ReasMem kopiert. Der Top Softcore setzt den Schreibzeiger im ReasMem nur um die Länge der *Nutz*daten weiter. Daher werden diese Bytes vom nachfolgenden Segment überschrieben. Durch diese Konstruktion findet sich nach dem Empfang des letzten Segmentes der vollständige Payload des Frames linear im ReasMem.

Neben dem Top Softcore zur Kommunikation mit der LLC-Schicht sowie den Speichern und Streams enthält der MAC-Layer noch den Bottom Softcore. Auf diesen wird im folgenden Abschnitt eingegangen.

3.5 Der Bottom Softcore

Der Bottom Softcore stellt die Verbindung zur PHY-Schicht her. Zu seinen Aufgaben gehören das Nachverfolgen des virtuellen Kanalzustandes (Virtual Carrier Sense, VCS) und die zeitliche Ansteuerung des PHY-Layers. Abbildung 3.11 zeigt seine Verbindungen zu den Modulen des Systems. Der Bottom-Softcore ist direkt mit der PHY-Schicht verbunden. Weiterhin bestehen Verbindungen zum Tonemap-Controller und FrameControl-Speicher. Der Bottom Softcore ist nicht direkt mit dem LFB oder dem ReasMem verbunden. Diese

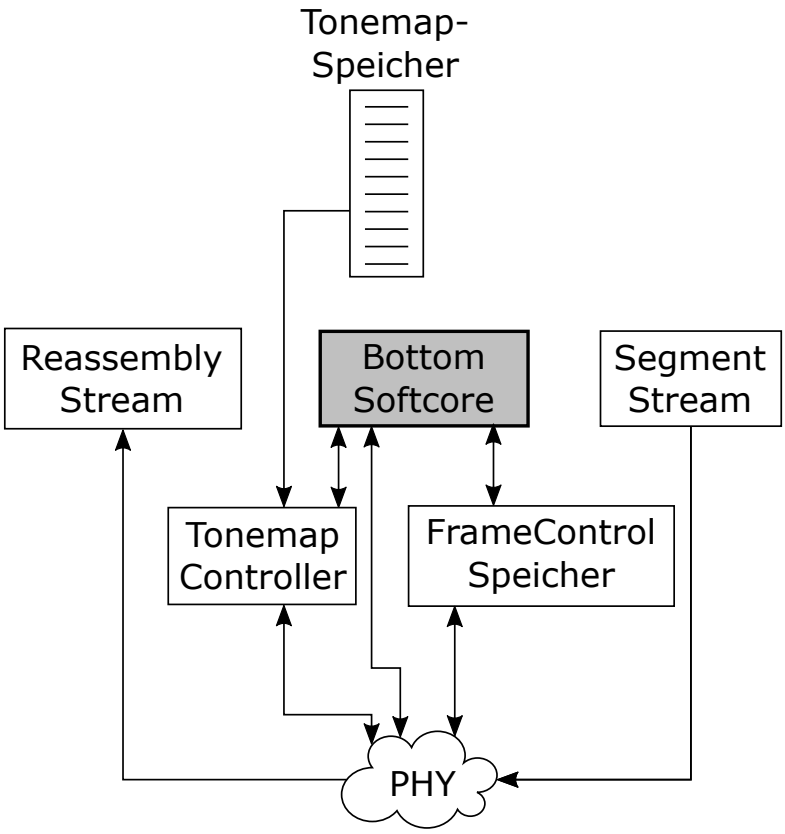


Abbildung 3.11: Die Verbindungen des Bottom Softcores

beiden Speicher sind über den Segment Stream bzw. den Reassemly Stream direkt mit dem PHY-Layer verbunden.

Der Bottom Softcore übernimmt die Übertragung einzelner Segmente. In welchem Zusammenhang diese zu einzelnen Frames stehen, wird im Bottom Softcore nicht beachtet. Zu seinen weiteren Aufgaben gehört die Erfassung von Statistiken.

Die Ansteuerung der PHY-Schicht erfordert eine präzise zeitliche Steuerung, da im HomePlug-Standard feste Reaktionszeiten im µs-Bereich vorgesehen sind (vgl. Abschnitt 2.3). Dies stellt eine wichtige Anforderung bei Entwicklung der Hard- und Software dar (vgl. Abschnitt 3.1).

Es wurden drei Varianten des Bottom Softcores entworfen, implementiert und evaluiert. Es ist dabei eine Variante mit zwei NEO430-Softcores („2xNEO“), eine Variante mit einem NEO430-Softcore („1xNEO“) und eine Variante mit einem PauloBlaze-Softcore

(„1xPauloBlaze“) untersucht worden. Die folgenden Abschnitte beschreiben die Implementierung der drei Varianten. Die Evaluierung ist in Kapitel 4 zu finden.

Es soll nun zunächst auf die Variante mit zwei NEO430-Softcores eingegangen werden.

3.5.1 Variante 2xNEO

In der HomePlug-Spezifikation wird die Ansteuerung der PHY-Schicht durch zwei parallele Prozesse modelliert [9, C.4.2.4, C.4.1]. Die beiden zugehörigen Zustandsautomaten sind im Anhang in Abbildung A.2 zu finden. In dieser Architekturvariante wurde daher ein Softcore für jeden der beiden Automaten vorgesehen. Die Ansteuerung der PHY-Schicht erfolgt dann durch beide Softcores gemeinsam. Die Idee hinter dieser Variante ist es, mit dem Software-Entwurf der Spezifikation so nah wie möglich zu folgen und so die Software einfach zu halten.

Insgesamt hat sich diese Architekturvariante als wenig geeignet herausgestellt: Die beiden Zustandsautomaten sind grob nach Sende- und Empfangsfunktion getrennt, allerdings gibt es einige Vermischungen. So wird beispielsweise die Priority-Resolution-Phase auch für ausgehenden Segmente immer durch den Zustandsautomaten des Empfangsteils abgehandelt. Es ist daher notwendig, beide Automaten zu synchronisieren, um eine korrekte Ansteuerung der PHY-Schicht zu erreichen. Diese Synchronisation der Programme auf den beiden Softcores erhöht die Komplexität der Software erheblich, da eine Vielzahl an Sperren (locks) notwendig ist. Weiterhin wird durch den zusätzlichen Aufwand die Performance und die Übersichtlichkeit der Software reduziert.

Eine korrekte Funktion der beiden Softcores hätte nur mit einer Vielzahl an Sperren erreicht werden können. Dies widerspricht der Idee, die Software zu vereinfachen, indem sie auf zwei Softcores aufgeteilt wird. Diese Variante benötigt außerdem durch die Verwendung von zwei Softcores mehr Ressourcen als alle anderen Varianten.

Es wurde eine Software-Version implementiert, mit der einzelne Frames übertragen werden konnten. Eine Weiterentwicklung erschien jedoch aufgrund der genanten Nachteile nicht sinnvoll.

Um die Synchronisationsprobleme zu lösen und gleichzeitig den Ressourcenverbrauch zu reduzieren, wurde eine Variante mit nur einem NEO430-Softcore entworfen.

3.5.2 Variante 1xNEO

In diese Variante wurden die beiden Prozesse für Sender und Empfänger aus der HomePlug-Spezifikation (vgl. Abschnitt 3.5.1) auf einem Softcore zusammengefasst. Dies vermeidet die explizite Synchronisation zwischen den beiden Kernen. Da nur einer statt zwei NEO430-Kernen verwendet wird, wird zudem der Verbrauch an FPGA-Ressourcen reduziert.

Für die Zusammenfassung der beiden Softcores ist es notwendig, die Zustandsautomaten von Sender- und Empfängerprozess zu vereinigen. Die Kombination der beiden Zustandsautomaten ist in Abbildung A.1 dargestellt. Grundsätzlich führt die Vereinigung von zwei parallel ablaufenden Zustandsautomaten zu eine sog. „Zustandsexplosion“, da potenziell jede mögliche Kombination aus Zuständen der beiden Automaten auftreten kann. Durch eine Analyse der *möglichen* Zustandsübergänge konnte die Anzahl der Zustände jedoch von 42 auf 8 reduziert werden. Hierbei kann ausgenutzt werden, dass es sich bei HomePlug um ein Halbduplex-Verfahren handelt, also nie gleichzeitig gesendet und empfangen werden kann [9].

Weiterhin wurden bei der Transformation leichte Veränderungen vorgenommen. Der `FAIL_WAIT`-Zustand, bei dem der Sender 10 ms vor einer erneuten Übertragung des Segments wartet, wurde entfernt. Diese Aufgabe wird nun vom Top Softcore übernommen.

Der verwendete NEO430-Softcore wurde für 8 kB Instruktionsspeicher und 2 kB Datenspeicher konfiguriert. Von den mitgelieferten Modulen werden nur Timer- und GPIO-Modul verwendet. Die Interrupthardware des Softcores wurde deaktiviert, da keine Interrupts verwendet werden. Das Timer-Modul wurde mit zwei zusätzlichen Timern konfiguriert (vgl. Abschnitt 2.1.1) die in der Software verwendet werden.

Die Kommunikation zwischen Top und Bottom Softcore erfolgt über Interrupts und einen geteilten Speicher. Da der geteilte Speicher nur 32 Bytes groß ist, wird er durch das Synthesetool nicht in BlockRAM sondern als Distributed RAM umgesetzt. Dabei werden einzelne Logikzellen des FPGAs als Speicher statt als Look-Up-Table verwendet. Distributed RAM kann, im Gegensatz zu BlockRAM, nicht über zwei Schreibports verfügen [13]. Ein solcher Speicher wird durch das Synthesetool daher in einzelne LUTs aufgelöst, wodurch sehr viele LUTs benötigt, da nur ein Bit pro LUT gespeichert werden kann. Bei der Verwendung von Distributed RAM sind 64 bit pro LUT möglich. Der gemeinsame Speicher wurde daher in zwei Hälften aufgeteilt. Der Top Softcore kann dabei nur die eine Hälfte schreiben, der Bottom Softcore die andere. Beide Softcores können den gesamten Speicher lesen. Durch die Änderung verfügt jede Hälfte des Speichers

nur noch über einen Schreibport und konnte durch das Synthesetool auf Distributed RAM abgebildet werden. Dies verbessert die Ressourcenausnutzung. Für die Software stellt der begrenzte Schreibzugriff kein Problem dar, da keine Stelle im gemeinsamen Speicher von beiden Softcores geschrieben werden muss.

Der vereinheitliche Zustandsautomat wurde in Software implementiert. Es wurde dabei die Implementierung stärker als bei der Variante 2xNEO vom Pseudocode der Spezifikation gelöst, was insgesamt zu einem kürzeren und übersichtlicheren und damit besser wartbaren Code führte.

Der verwendete 50 MHz-Takt teilt eine Mikrosekunde in 20 Takte. Der NEO430 benötigt durch seine Multi-Cycle-Architektur durchschnittlich 10 Takte, um eine Instruktion auszuführen [3]. Es können also in einer Mikrosekunde zwei Instruktionen ausgeführt werden. Um die zeitlichen Anforderungen zu erfüllen, ist daher die Verwendung von zeitlich effizientem Code wichtig.

Der verwendete GCC-Compiler erzeugt an vielen Stellen nicht-optimalen Assemblercode. Als besonders problematisch haben sich hier Bitshift-Operationen herausgestellt. Shifts um 8 Byte werden durch den Compiler in acht einzelne Shift-Befehle umgesetzt. Effizienter wäre jedoch die Verwendung des `swpb`-Befehls, mit dem die beiden Bytes eines Wortes getauscht werden. Bitshift-Operationen werden an vielen Stellen im Programmcode des Bottom Softcores verwendet, um einzelne Bits aus Wörtern zu extrahieren.

Weiterhin wurde Code der Art `val++` in eine sehr ineffiziente Folge von byteweise Load-, Store- und Additionsbefehlen umgesetzt, wenn die verwendete Variable als `volatile` deklariert wurde. Besser geeignet wäre an diese Stelle der `inc`-Befehl, der zudem direkt auf einer Speicherstelle arbeiten kann.

Außerdem wurde die Multiplikation einer Zahl mit der Konstanten 131 nicht erkannt und eine allgemeine Software-Multiplikationsroutine aufgerufen. Der in Listing 3.3 gezeigte Code löst diese Aufgabe effizienter. Dabei werden zunächst die beiden Bytes der Zahl getauscht. Da die Zahl immer kleiner als 256 ist wird der Wert so mit 256 multipliziert. Der Wert wird nun durch eine Rechtsverschiebung durch zwei geteilt. Auf das Ergebnis wird der ursprüngliche Wert noch drei Mal addiert, sodass sich schlussendlich der mit 131 multiplizierte Wert ergibt.

Der durch GCC erzeugte Assemblercode wurde manuell auf die oben genannten ineffiziente Konstrukte untersucht und im C-Code durch Inline-Assembler ersetzt. Mit diesen Optimierungen konnten die zeitlichen Vorgaben des HomePlug-Standards eingehalten werden.

```
int delay20;
asm(
        "mov %[fl], %[res]      \n\t" //res = fl
        "swpb %[res]            \n\t" //res *= 256
        "clrc                   \n\t"
        "rrc %[res]             \n\t" //res /= 2
        "add %[fl], %[res]      \n\t"
        "add %[fl], %[res]      \n\t"
        "add %[fl], %[res]      \n\t" //res += 3*fl -> res = 131*fl
: [res] "=r" (delay20)
: [fl] "r" (fl));
```

Listing 3.3: Multiplikation mit 131

Die Variante 1xNEO ist vollständig funktionsfähig. Der Bottom Softcore stelt im Wesentlichen einen einfachen Zustandsautomaten dar. Es wurde daher untersucht, ob der Einsatz eines einfacheren Softcores, des PauloBlaze möglich ist.

3.5.3 Variante 1xPauloBlaze

Als dritte Variante des Bottom Softcores wurde ein Entwurf mit einem PauloBlaze-Softcore implementiert. Diese Version nutzt den gleichen kombinierten Zustandsautomaten wie die Variante 1xNEO (vgl. Abschnitt 3.5.2). Mit dieser Variante soll untersucht werden, ob es möglich ist, FPGA-Ressourcen zu sparen, indem für die Ansteuerung der PHY-Schicht ein PauloBlaze verwendet wird.

Der PauloBlaze kann, wie in Abschnitt 2.1.2 dargestellt, nicht in der Programmiersprache C oder eine anderen Hochsprache programmiert werden. Daher muss stattdessen Assemblercode verwendet werden. Es wurde daher das in C geschriebene Programm der Variante mit einem NEO430 manuell in Assemblercode für den PauloBlaze übersetzt. Neben der Verwendung von Assemblercode mussten einige weitere Änderungen vorgenommen werden, da der PauloBlaze eine 8-Bit-Architektur und der NEO430 eine 16-Bit-Architektur verwendet.

Der PauloBlaze verfügt über keine integrierte Peripherie. Diese kann nur über die 256 zur Verfügung stehenden Ein- und Ausgabeports angebunden werden. Es wurde ein GPIO- und ein Timer-Modul implementiert und auf diese Weise angebunden. Auch die Verbindung zu FrameControl-Speicher und Tonemap-Controller erfolgt über Ein- bzw. Ausgabeports. Es muss dabei berücksichtigt werden, dass diese nur 8 Bit breit sind und

sich daher nicht direkt auf das 16 bit breite Registerinterface der NEO430-CPU abbilden lassen.

Der PauloBlaze wird, wie das gesamte Design, mit 50 MHz getaktet, er benötigt aber zur Ausführung sämtlicher Befehle nur 2 Takte [6]. Damit erreicht er einen etwa fünfmal höheren Instruktionsdurchsatz als der NEO430. Zusätzlich führt die Programmierung in Assembler zu einer besseren Kontrolle über das Zeitverhalten des Programms. Die zeitlichen Anforderungen der PHY-Schicht konnten mit dem PauloBlaze ohne spezielle Optimierungen erreicht werden.

Der als VHDL-Code vorliegende PauloBlaze ermöglicht es, die Tiefe des Hardware-Stacks und die Größe des Scratchpad-Speichers über ein VHDL-Generic anzupassen (vgl. Abschnitt 2.1.2). Im originalen PicoBlaze kann die Größe des Hardware-Stacks gar nicht verändert werden. Die Größe des Scratchpad-Speichers kann nur zwischen 64, 128 oder 256 Einträgen variiert werden. Der Scratchpad-Speicher des PauloBlaze wurde auf 64 Einträge eingestellt, da mehr nicht verwendet werden. Dies ist durch die manuelle Verwaltung der Speicheradressen sichergestellt. Der Hardware-Stack wurde auf vier Einträge begrenzt, da eine tiefere Verschachtlung von Funktionsaufrufen nicht vorkommt wird.

Der PauloBlaze verfügt, wie in Abschnitt 2.1.2 beschrieben, über zwei unabhängige Registerbänke A und B. Die entworfene Software benutzt nur Registerbank A, daher konnte Bank B entfernt werden. Dies ist nicht über ein VHDL-Generic möglich, es musste daher direkt der Quellcode des Prozessors geändert werden. Alle diese Maßnahmen reduzieren die Verwendung von Speicher auf dem FPGA.

Unabhängig von der verwendeten Softcore-Variante werden im Bereich des Bottom Softcores verschiedene Hardware-Module verwendet die mit diesem interagieren. Dazu gehört der Tonemap-Controller.

3.5.4 Tonemap-Controller

Der Tonemap-Controller ist dafür zuständig eine benötigte Tonemap aus dem Tonemap-Speicher zu laden und dem PHY-Layer bereitzustellen. Eine Übersicht über den Tonemap-Controller ist in Abbildung 3.12 zu finden. Eine passende Tonemap muss dem PHY-Layer bereitgestellt werden wenn dieser ein Segment senden oder empfangen soll. Die Tonemap wird parallel als 104 bit breites Signal an den PHY-Layer geliefert. Für ausgehende Segmente kann das Laden einer Tonemap über das Registerinterface von dem Bottom

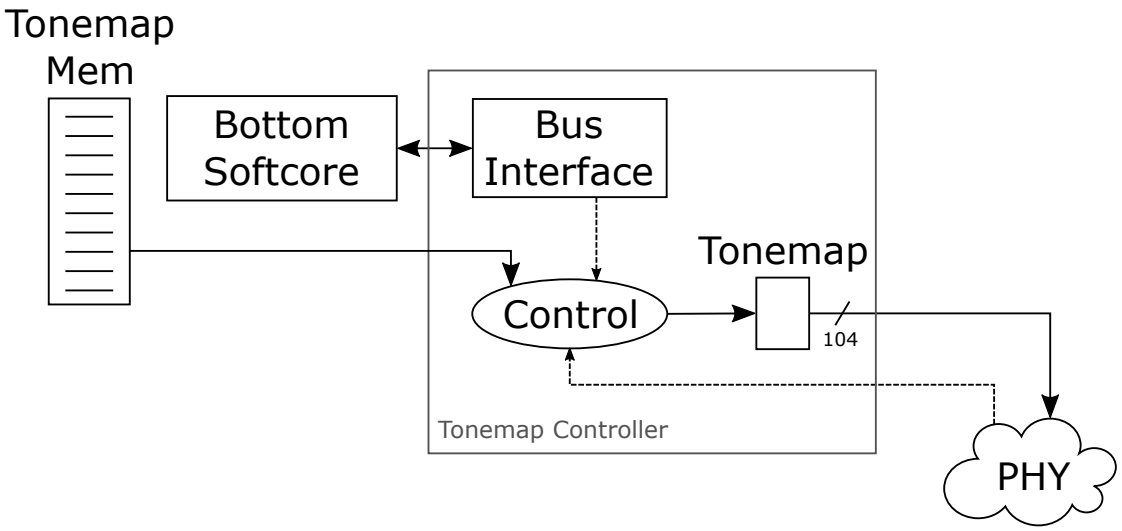


Abbildung 3.12: Tonemap-Controller

Softcore ausgelöst werden; bei eingehenden Segmenten über ein Trigger-Signal direkt vom PHY-Layer.

Die Verwaltung der Tonemaps obliegt dem Top Softcore. Es können jeweils bis zu 15 Tonemaps in Sende- und Empfangsrichtung bekannt sein, insgesamt also bis zu 30 (vgl. Abschnitt 2.4.1). Jede Tonemap ist 13 Byte groß, daher werden $30 * 13\,\mathrm{B} = 390\,\mathrm{B}$ benötigt. Der Tonemap-Speicher ist daher 512 Byte groß.

Da nur 390 Byte des 512 Byte großen Tonemap-Speichers benutzt werden, ist es möglich, den Anfang aller 30 Tonemaps an 16-Byte-Grenzen auszurichten (siehe Abbildung 3.13). Dies vereinfacht die Adressierung der Tonemaps im Tonemap-Controller, da so die Tonemapnummer gefolgt von vier Nullbits direkt als Adresse verwendet werden kann. Durch diese Maßnahme wird die Fragmentierung des Speichers erhöht, da nun viele kleine statt einem großen, zusammenhängen Speicherbereich frei bleiben. Der Tonemap-Speicher wird allerdings für keine anderen Daten verwendet, daher stellt dies keinen Nachteil dar.

Zur Übertragung eines Frames muss neben der passeden Tonemap auch der passenden FrameControl-Header bereitgestellt werden. Hierfür ist der Bottom Softcore mit dem FrameControl-Speicher verbunden.

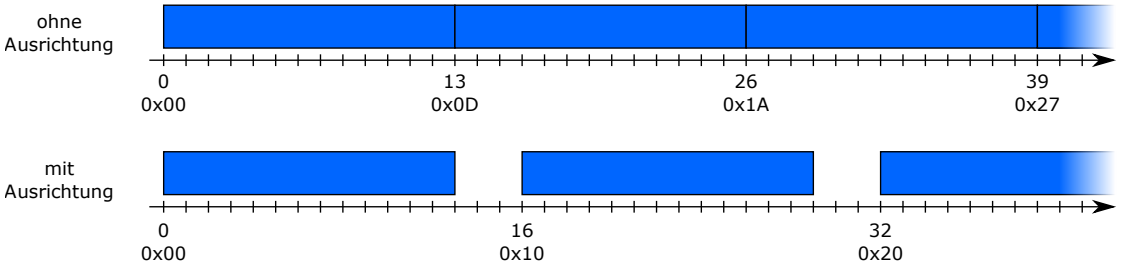


Abbildung 3.13: Tonemap-Speicher mit und ohne Speicherausrichtung (alignment). Die Speicherausrichtung reduziert die Menge des nutzbaren Speichers, vereinfacht aber die Adressierung.

3.5.5 FrameControl-Speicher

Der FrameControl-Speicher speichert die FrameControl-Header für ein- und ausgehende Frames. Eine Übersicht ist in Abbildung Abbildung 3.14 dargestellt. Jeder FrameControl-Header ist 25 bit groß. Der FrameControl-Speicher bietet Platz für drei FrameControl-Header. Dabei können zwei (Tx SFC und Tx EFC) von dem Bottom Softcore geschrieben werden. Sie enthalten die Informationen für den Start- und End-Delimiter eines ausgehenden Long Frames (vgl. Abschnitt 2.3). Der dritte Speicher (Rx FC) wird von der PHY-Schicht geschrieben. Er enthält die FrameControl-Informationen eines eingehenden Delimiters.

Ein FrameControl-Header enthält 17 bit Nutzdaten. Die restlichen acht Bit enthalten eine Prüfsumme. Im FrameControl-Speicher werden für SFC und EFC nur 17 bit gespeichert. Die CRC-Prüfsumme wird durch ein kombinatorisches XOR-Netz automatisch erzeugt. Dabei wird die in Abschnitt 2.2.2 beschriebene Methode angewandt.

Für eingehende FrameControl-Header wird die Prüfsumme mit einem ähnlichen CRC-Netz automatisch kontrolliert. Der Bottom Softcore kann über das Registerinterface abfragen, ob die Prüfsumme korrekt ist.

Durch die Berechnung bzw. Überprüfung der Prüfsumme in Hardware wird der Softcore entlastet. Durch den Verzicht auf das Speichern des berechneten Prüfsumme wird der Verbrauch an Registern reduziert.

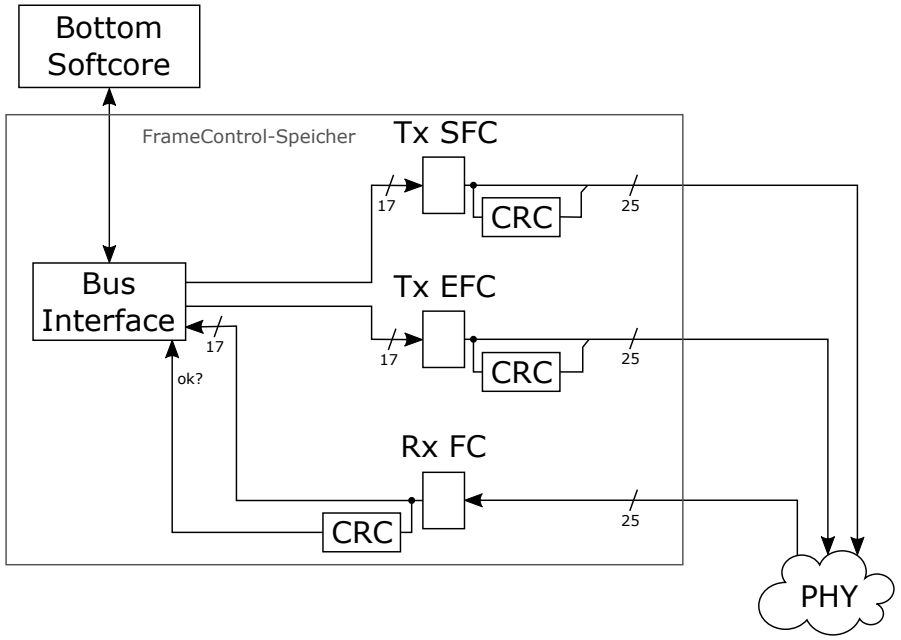


Abbildung 3.14: FrameControl-Speicher

3.6 Zusammenfassung

Der Top Softcore kommuniziert über einen gemeinsamen Speicher mit der LLC-Schicht. Soll ein Frame gesendet werden wird dieser, zusammen mit einem Header in den LFB kopiert. Während der Übertragung wird der Frame automatisch durch den CRC Stream TX verschlüsselt und seine Prüfsumme berechnet. Eingehende Frames liegen im ReasMem. Aus diesem Speicher werden sie vom Top Softcore durch den CRC Stream RX gelesen. Dabei werden sie automatisch entschlüsselt und ihre Prüfsumme wird kontrolliert. Der eingehende Frame wird vom Top Softcore verarbeitet und die Nutzdaten in den LLC-Speicher kopiert.

Zu den weiteren Aufgaben des Top Softcore gehören die Steuerung von Segmentierung und Wiederzusammenstellung sowie die Verwaltung von Tonemaps. Die Tonemaps werden im Tonemap-Speicher abgelegt. Für die Segmentierung ist die Berechnung der Segmentgröße notwendig. Dabei wird der Softcore durch vorberechente Werte im Capacity-ROM unterstützt. Wenn eine neues Segment übertragen werden soll erzeugt der Top Softcore den dafür notwendigen Header. Dieser wird im HeaderMem abgelegt.

Der Top Softcore wurde mit einem NEO430-Softcore implementiert und in C programmiert. Der NEO430 verfügt über eine 16-Bit-Architektur und zeichnet sich durch eine hohe Erweiterbarkeit und einen geringen Ressourcenbedarf aus.

Der Bottom Softcore nimmt die zeitliche Steuerung der PHY-Schicht vor. Seine Wesentliche Aufgabe ist dabei die Verfolgung des virtuellen Kanalzustandes (VCS). Soll ein Frame übertragen werden, wird zur passenden Zeit die Übertragung in der PHY-Schicht gestartet. Die zu übertragenden Daten werden dabei von der PHY-Schicht ohne Interaktion des Bottom Softcore über den Segment Stream aus dem Speicher gelesen und dabei virtuell segmentiert. Eingehende Segmente werden automatisch in den Reassembly Stream geschrieben. Der Bottom Sotcore versendet dann automatisch die passende positive oder negative Bestätigung.

Im Zuge der Arbeit wurden insgesamt drei Varianten des Bottom Softcores erstellt. Es wurde eine Variante mit zwei NEO430 Softcores, eine Variante mit nur einem NEO430, und eine Variante mit einem PauloBlaze entwickelt.

In der Variante 2xNEO werden Sender und Empfänger auf jeweils einen Softcore abgebildet. Dies folgt der Darstellung in der Spezifikation, welche diese als zwei parallele Zustandsautomaten abbildet. Trotz der logischen Trennung sind in die meisten Vorgänge beide Automaten involviert. Es ist daher nötig, die beiden Softcores zu synchronisieren, was eine große Anzahl an Sperren (locks) erfordert. Dies erhöht die Komplexität der Software. Weiterhin hat diese Variante durch den Einsatz von zwei Softcores dem größten Bedarf an FPGA-Ressourcen. Diese Version wurde daher nicht weiter entwickelt.

Für die Variante 1xNEO wurden die beiden Zustandsautomaten von Sender und Empfänger zusammengefasst. Der resultierende, vereinfachte Automat wurde dann auf dem Softcore implementiert. Da nur ein Prozessor verwendet wird sind keine Sperren nötig, was die Software vereinfacht. Diese Variante wurde vollständig entwickelt.

In der dritten Variante wurde der NEO430 durch den einfacheren PauloBlaze ausgetauscht. Dieser ist als 8-Bit-Architektur ausgelegt und lässt sich nur in Assembler programmieren, benötigt aber weniger FPGA-Ressourcen und hat einen höheren Instruktionsdurchsatz als der NEO430. Durch die höhere Ausführungsgeschwindigkeit wird die Ansteuerung des PHY-Layers zeitlich präziser. Die Implementation der Software in Assembler steigert jedoch die Komplexität der Software, was die Anpassbarkeit und Wartbarkeit reduziert. Die Variante mit dem PauloBlaze als Bottom Softcore ist ebenfalls vollständig funktionsfähig. Diese Variante benötigt die wenigsten Hardware-Ressourcen.

4 Evaluation

In diesem Kapitel werden die in Kapitel 3 vorgestellten Implementierungen unter verschiedenen Aspekten untersucht. Es werden dabei die drei vorgestellten Varianten mit zwei bzw. einem NEO430-Softcore (2xNEO/1xNEO) sowie mit einem PauloBlaze-Softcore (1xPauloBlaze) als Bottom Softcore evaluiert und verglichen. Als Referenz dient die in [14] beschriebene, bereits bestehende, VHDL-Implementierung.

In Abschnitt 4.1 wird der Bedarf an FPGA-Ressourcen untersucht. Es werden dann in Abschnitt 4.2 die drei Varianten bezüglich ihrer Software-Komplexität verglichen. Der Datendurchsatz und die Anpassbarkeit der Varianten wird in Abschnitt 4.3 und Abschnitt 4.4 betrachtet.

4.1 Verwendung von FPGA-Ressourcen

In diesem Abschnitt wird der Verbrauch an FPGA-Ressourcen, also LUTs, BlockRAM und DSP-Blöcken der verschiedenen Varianten untersucht. Zu diesem Zweck wurden alle Designs mit dem Synthesewerkzeug Vivado 2017.2 der Firma Xilinx auf das FPGA Kintex 7 410TIFFV900 abgebildet. Als Zieltaktfrequenz wurde dabei 50 MHz angegeben. Die aus [14] entnommenen Werte des bereits bestehenden Designs dienen als Referenz. Diese Werte wurden für das gleiche FPGA ermittelt, allerdings wurde die ältere Version 2016.4 von Vivado verwendet; eine Vergleichbarkeit der Werte ist dennoch gegeben.

In Tabelle 4.1 ist die Ressourcennutzung gezeigt. Angeben ist die Zahl der verwendeten LUTs, Register, DSP-Blöcke und BlockRAMs. Weiterhin ist die prozentuale Änderung gegenüber der Referenz gezeigt. Grüne Zahlen stellen Verbesserungen dar, rote Verschlechterungen. Alle drei Varianten benötigen deutlich weniger Ressourcen als die Referenz. Die maximale Taktfrequenz sinkt jedoch.

Bereits die Variante 2xNEO benötigt weniger als die Hälfte an LUTs wie die Referenz ($-64\,\%$). Dies ist auf den Einsatz der Softcores zurückzuführen: Ein Softcore benötigt eine feste Anzahl an LUTs und ermöglicht es dann, durch Programmierung einen beliebigen

Variante	LUTs	Register	DSPs	BlockRAM	Max. Taktfrequenz (MHz)
Referenz	7845	5718	1	21	160
2xNEO	2836 −64 %	1795 −69 %	0	14 −29 %	≈100
1xNEO	1941 −75 %	1401 −75 %	0	10,5 −50 %	130 −19 %
1xPauloBlaze	1730 −78 %	1198 −79 %	0	9,5 −55 %	104 −35 %

Tabelle 4.1: Verwendung von FPGA-Ressourcen der verschiedenen Varianten. Angeben ist die prozentuale Änderung gegenüber der Referenz. Grüne Werte stellen Verbesserungen dar, rote Verschlechterungen.

Modul	LUTs	Register	Softcore
PHY Frame Receive	771	375	✓
Reassemble	2548	1939	✓
Entschlüsselung	735	512	–
RX MAC Serviceblock	1116	1171	✓
TX MAC Serviceblock	788	407	✓
Verschlüsselung	506	377	–
Segmentierung	478	242	(✓)
PHY Frame Transmit	527	254	✓
Link Status	229	138	✓

Tabelle 4.2: Verwendung von FPGA-Ressourcen der Referenzvariante [14]. Dargestellt ist auch, welche Funktionen auf die Softcores abgebildet werden konnten.

Kontrollfluss zu realisieren, ohne dass sich der Bedarf an Hardware-Ressourcen ändert. In der Referenzversion ist der Kontrollfluss als Hardware in Zustandsautomaten realisiert. Diese benötigen für jeden möglichen Zustand Hardware-Ressourcen, auch wenn immer nur ein Zustand gleichzeitig aktiv ist. Dies wird auch in Tabelle 4.2 deutlich. Dort ist Anzahl an benötigten LUTs pro Modul für die Refenzvariante aufgetragen. Jedes Modul wird durch einen eignen Zustandsautomaten dargestellt. Ein besonders hoher Anteil an verwendeten LUTs ist im Reassemble- und RX-MAC-Serviceblock-Modul zu erkennen. Diese beiden Module haben den komplexesten Kontrollfluss. Alle Kontrollflüsse werden in der in dieser Arbeit entwickelten Variante auf Softcores ausgeführt. Nur Ver- und Entschlüsselung, die Berechnung von Prüfsummen und Teile der Segmentierung werden weiterhin in Hardware ausgeführt (vgl. Kapitel 3). Insgesamt werden durch die Softcore-Implementierung etwa 6000 LUTs eingespart.

Die Unterschiede in der Ressourcenverwendung zwischen den drei Varianten sind auf den Einsatz verschieden großer Softcores zurückzuführen. Der Ressourcenbedarf der restlichen Logik ändert sich nicht durch die Verwendung anderer Softcores, es werden immer für alle Varianten fest 540 LUTs und 820 Register benötigt.

In der Variante 1xNEO wird gegenüber der Variante 2xNEO nur ein NEO430 (ca. 700 LUTs) als Bottom Softcore verwendet. Dadurch reduziert sich die Anzahl der verwendeten LUTs um fast 900. Neben den 700 LUTs durch den NEO430 werden rund 200 LUTs eingespart, weil der gemeinsame Speicher reduzierter ausfällt. In der Version 2xNEO muss dieser über drei Leseports verfügen. In der Version 1xNEO wird weniger Speicher benötigt und zwei Leseports sind ausreichend.

Durch das Austauschen des NEO430 gegen einen PauloBlaze (ca. 500 LUTs inklusive nötiger Peripherie) werden noch einmal etwa 200 LUTs eingespart, sodass diese Variante insgesamt 78 % weniger LUTs benötigt als die Referenzvariante. Detaillierte Syntheseergebnisse aller Varianten sind im Anhang in Tabelle A.3, Tabelle A.4 und Tabelle A.5 zu finden.

Die Verwendung von Registern und BlockRAM konnte gegenüber der Referenzvariante ebenfalls deutlich reduziert werden. Dies ist auf die Streaming-Architektur (vgl. Abschnitt 3.4) zurückzuführen. Durch die on-the-fly-Erzeugung der benötigten Daten, müssen diese nicht zwischengespeichert werden. Es werden nur 4,5 BlockRAMs zum Speichern von Nutzdaten verwendet, der Rest wird als Instruktions- und Datenspeicher für die Softcores benutzt. Die Variante 2xNEO benötigt ingesamt 14 BlockRAMs. Dies stellt gegenüber der Referenz eine Verbesserung um 29 % dar. In der Variante 1xNEO fallen Instruktions und Datenspeicher für einen NEO430 weg, daher benötigt diese Variante insgesamt nur 10,5 BlockRAMs. Der PauloBlaze verwendet keinen BlockRAM als Datenspeicher und benötigt auch weniger Instruktionsspeicher als der NEO430, sodass in dieser Variante 1xPauloBlaze nochmals ein BlockRAM eingespart werden kann. Damit benötigt diese Variante insgesamt 9,5 BlockRAMs, 55 % weniger als die Referenz.

Alle drei Varianten erreichen die Zieltaktfrequenz von 50 MHz. Gegenüber der Referenzvariante sinkt die maximal erreichbare Taktfrequenz von 160 MHz auf 104 MHz bis 130 MHz. Auch dies ist durch den Einsatz von Softcores zurückzuführen. In den Zustansautomaten der Referenzvariante ist üblichweise einfachere Logik als in den Softcores zu finden, was zu einem kürzeren kritischen Pfad und damit einer höheren maximalen Taktfrequenz führt.

Softcore	Code-Zeilen (ca.)	Instruktionen
Top	2800	5829
Bottom (2xNEO)		
Sender	570	1070
Empfänger	550	1168
Bottom (1xNEO)	1180	1696
Bottom (1xPauloBlaze)	1490	1024

Tabelle 4.3: Anzahl Codezeilen und Instruktionen der Software-Varianten

Im Bereich der Ressourcennutzung ist die Variante 1xPauloBlaze das Optimum. Nachteilig ist jedoch die Programmierung des PauloBlaze in Assembler. Hier kann Ressourcennutzung gegen einfache Code-Wartbarkeit abgetauscht werden.

4.2 Software-Komplexität

Durch die Verwendung von Softcores wird nicht mehr die gesamte Funktionalität durch die Hardware beschrieben, sondern es ist Software für die Ansteuerung notwendig. Die drei Varianten unterscheiden sich hier in ihrer Komplexität.

Die Software der NEO430-Softcores wurde in der Programmiersprache C implementiert. Der Code für den PauloBlaze wurde mit dem Assembler *obpasm* entwickelt [24]. Der Umfang der Quelldateien sowie die Größe des resultierenden Programms ist in Tabelle 4.3 gezeigt. Der Top Softcore ist in allen Varianten unverändert enthalten. Sein Programm ist mit 5829 Instruktionen mit Abstand das Umfangreichste. Dies ist dadurch zu erklären, dass der Top Softcore die Programmteile mit dem komplexesten Kontrollfluss enthält, also die LLC-Kommunikation sowie die Steuerung von Segmentierung und Wiederzusammenstellung.

Das Programm für den Bottom Softcore in der Variante 1x NEO umfasst etwa so viele Zeilen wie der Code der beiden Bottom Softcores in der Variante 2xNEO zusammen. Das resultiere Programm ist dennoch etwas kleiner, da gemeinsammer Code, wie etwa Initialisierungsfunktionen und Teile der C-Laufzeitbibliothek, nur einmal enthalten sind.

Der Assembler-Quellcode für den PauloBlaze ist länger als der korrespondierende C-Code, da C oft eine kompaktere Schreibweise von Ausdrücken ermöglicht. Das resultierende Programm ist dennoch kürzer, was sich dadurch erklären lässt, dass handoptimierter

Assemblercode häufig kompakter ist als die Ausgabe eines Compilers, insbesondere, wenn dieser, wie im vorliegenden Fall, nur wenige Optimierungen vornimmt.

Der Top Softcore muss keine strikten Echtzeitanforderungen erfüllen, daher wurde das Programm nicht primär auf Geschwindigkeit optimiert, sondern die Wartbarkeit des Codes stand im Vordergrund. Für die Steuerung des PHY-Layers durch den Bottom Softcore sind jedoch enge Zeitschranken einzuhalten. Der C-Code des Bottom Softcores musste daher, wie in Abschnitt 3.5.2 dargestellt, stark optimiert werden. An einigen Stellen war die Verwendung von Inline-Assembler notwendig. Zudem muss bei jeder Änderung des Programms untersucht werden, ob die Zeitschranken weiterhin nicht verletzt werden, da der Zeitbedarf einer Anweisung nicht direkt aus dem C-Code abgeleitet werden kann. Der Assemblercode des PauloBlaze musste nicht speziell auf Geschwindigkeit optimiert werden. Dies ist auf den höheren Instruktionsdurchsatz des PauloBlaze zurückzuführen. Zusätzlich ist der Zeitbedarf eines Programms direkt aus dem Quellcode ersichtlich.

Zusammengenommen relativiert sich damit der Aspekt der Wartbarkeit: Der C-Code des NEO430 erscheint zunächst einfacher, jedoch ist seine Wartbarkeit aufgrund der notwendigen starken Optimierung nur wenig besser als die des Assemblercodes des PauloBlaze.

Das Zusammenspiel aus Soft- und Hardware bestimmt die Funktion des Gesamtdesigns und beeinflusst den schlussendlich erreichten Datendurchsatz. Dieser soll im folgenden Kapitel untersucht werden.

4.3 Datendurchsatz

Um den Datendurchsatz des Gesamtsystems zu messen, wurden zwei identische MAC-Layer verwendete. Diese wurden durch einen simulierten PHY-Layer verbunden. Dieser PHY-Simulator simuliert einen optimalen Kanal mit 78 verfügbaren Trägern, DQPSK-Modulation (vgl. Abschnitt 2.4.1) und ohne Übertragungsfehler. Es wurden acht Pakete mit jeweils 1,5 kB Nutzdaten übertragen und die dafür benötigte Zeit gemessen. Die Zeit beginnt mit dem Anliegen des ersten Paketes an der LLC-Schnittstelle des ersten MAC-Layers und endet, wenn das letzte Pakete auf der LLC-Schnittstelle des zweiten MAC-Layers empfangen wurde. Es wird dabei davon ausgegangen, dass der Kanal im Ruhezustand (idle) ist und keine andere Station gleichzeitig sendet. Zudem ist der sendenden Station die Tonemap des Empfängers bereits bekannt.

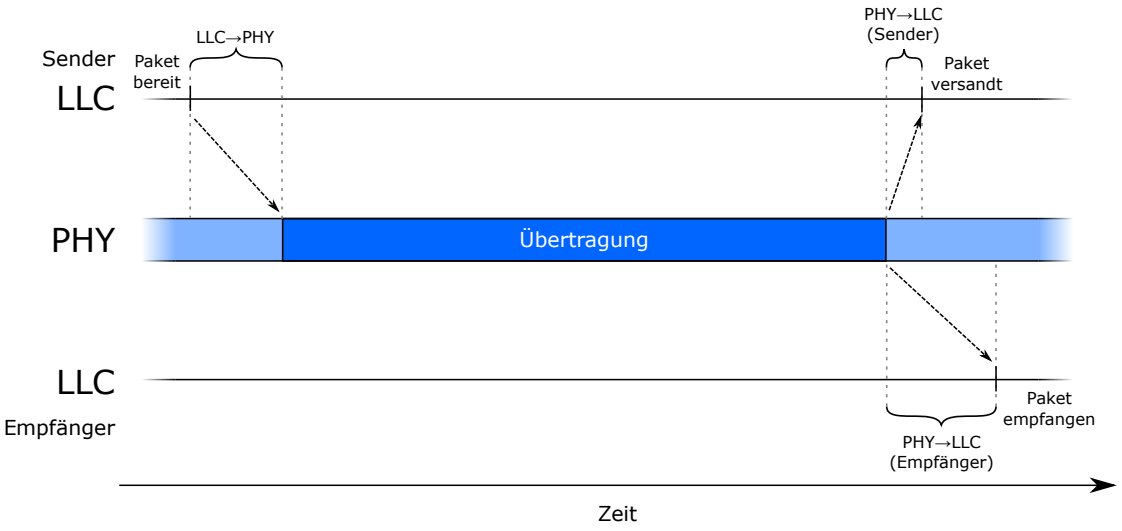


Abbildung 4.1: Die drei relevanten Latenzen des MAC-Layers

Unter diesen optimalen Bedingungen dauert die Übertragung der 12 kB 11,89 ms, was einer Datenrate von 8,07 Mbit/s entspricht. Dieser Wert ist vergleichbar mit dem von der HomePlug Alliance berechneten Maximum von 8,4 Mbit/s [23] und deckt sich mit Simulationen und Messungen anderer Untersuchungen, in denen eine Datenrate von 8,08 Mbit/s bestimmt wurde [21]. Der implementierte MAC-Layer liefert also eine mit kommerziellen Geräten vergleichbare Performance.

Die Übertragungsrate auf dem Kanal ist durch den Standard fest vorgegeben. Es ist daher die Latenz des MAC-Layers, die den Durchsatz beschränkt. Es lassen sich drei Arten von Latenz unterscheiden, die in Abbildung 4.1 zu sehen sind. Zunächst ist die Zeit wichtig, die vom Bereitstellen des Paketes auf der LLC-Schnittstelle bis zur Sendeanforderung an den PHY-Layer benötigt wird (LLC→PHY). Wurde das Paket vollständig übertragen und die Quittung empfangen, vergeht beim Sender nochmals Zeit, bis die erfolgreiche Übertragung auf der LLC-Schnittstelle anzeigt wird (PHY→LLC, Sender). Ebenso muss der Empfänger das Paket verarbeiten und es verstreicht Zeit, bis er die Nutzlast der LLC-Schicht bereitstellt (PHY→LLC, Empfänger).

Im Idealfall wären alle diese Latenzen Null, was die maximale Übertragungsrate von 8,4 Mbit/s ermöglichen würde. In der Realität vergeht immer eine feste Zeit, die benötigt wird, um die Metadaten der Frames und Segmente zu erzeugen bzw. zu prüfen. Dazu kommt ein variabler Anteil, der von der Menge der Nutzdaten abhängt. Dieser variable Anteil entsteht im Wesentlichen dadurch, dass die Nutzdaten aus bzw. in den LLC-Speicher kopiert werden müssen.

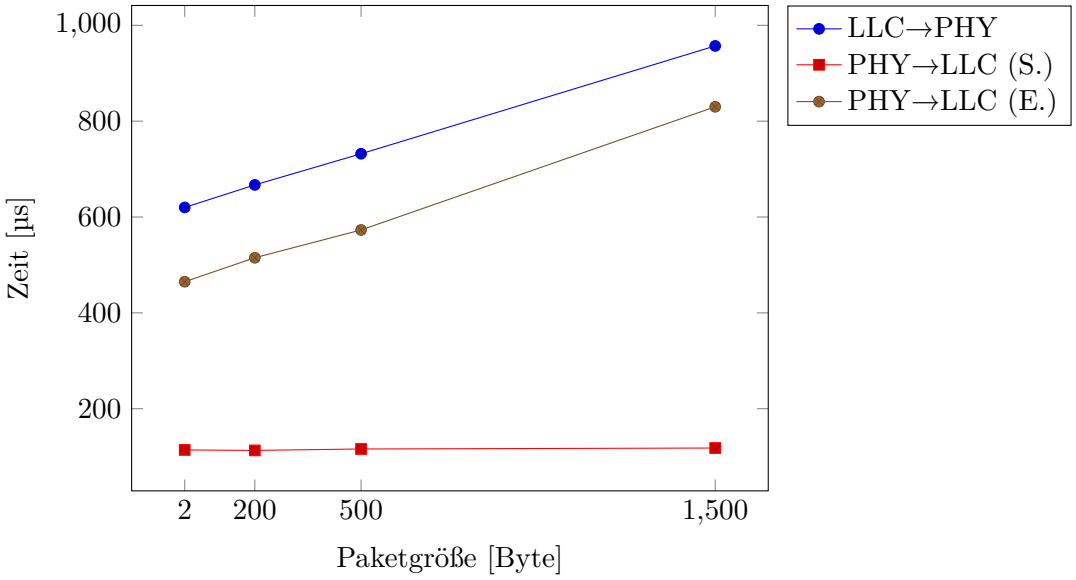
In Abbildung 4.2 sind die gemessenen Latenzen für verschiedene Paketgrößen dargestellt. Zusätzlich wurde eine lineare Regression über die Datenpunkte durchgeführt. Es wurden die Latenzen mit Nutzdaten von 2, 200, 500 und 1500 Byte gemessen. Für die Messung wurde davon ausgegangen, dass der Kanal im Ruhezustand (idle) ist, die Übertragung also ohne Wartezeit durch Contention oder andere laufende Übertragungen beginnen kann.

Die Latenz LLC→PHY beträgt bei einer Paktgröße von 2 Byte 620 µs. Mit steigender Paketgröße nimmt die Latenz linear um 0,23 µs/Byte zu. Dies ist exakt die Datentransferrate, die mit der optimierten Kopierschleife erreicht wird (vgl. Abschnitt 3.5.2). Die 620 µs entsprechen damit dem statischen Anteil der Latenz. Dieser ist durch die aufwendige Verarbeitung ausgehender Pakete zu erklären: Für jedes ausgehende Paket muss überprüft werden, ob die Tonemap des Empfängers vorliegt und andernfalls eine Kanalschätzungsanfrage in den Frame eingefügt werden. Bei der maximalen Paketgröße von 1500 Byte liegt die Latenz bis zum Beginn der Übertragung bei 957 µs.

Die Zeit, die zwischen dem Ende der Übertragung und der Signalisierung der LLC-Schicht vergeht (PHY→LLC, Sender) ist unabhängig von der Paketgröße. Sie beträgt immer rund 115 µs. In dieser Zeitspanne werden keine Daten kopiert, daher ist sie unabhängig von der übertragenen Datenmenge.

Auf der Seite des Empfängers (PHY→LLC, Empfänger) ist eine solche Abhängigkeit von der übertragenen Datenmenge erkennbar. Bei einer minimalen Paketgröße von 2 Byte vergehen 465 µs zwischen dem Ende der Übertragung und der Signalisierung der LLC-Schicht. Diese Verzögerung entsteht durch die notwendige Analyse eingehender Pakete. Diese können Mac-Managment-Informationen enthalten, die an dieser Stelle ausgewertet werden müssen. Mit steigender Paketgröße nimmt die Verzögerung linear zu. Die Änderungsrate von 0,23 µs/Byte entspricht dabei der Datentransferrate der Kopierschleife (siehe oben). Im Falle der maximalen Paketgröße von 1500 Byte beträgt die Latenz 830 µs.

Neben den Latenzen, die den Durchsatz des System verringern, gibt es auch harte Echtzeitbedingungen, die eingehalten werden müssen, um das HomePlug-Protokoll nicht zu verletzten. Die wichtigste ist die 26 µs lange RIFS-Zeit. Diese Zeitspanne liegt zwischen dem Ende der Übertragung eines Segmentes und dem Beginn der Quittung (vgl. Abschnitt 2.4.1). In dieser Zeit muss entschieden werden, ob eine positive oder negative Quittung gesendet wird. Dazu müssen die empfangenen Daten vollständig verarbeitet und ihre Prüfsumme kontrolliert worden sein. In Abbildung 4.3 ist aufgetragen, wie lange



(a) Grafische Darstellung der Latenzen

Paketgröße (Byte)	LLC→PHY Sender (µs)	PHY→LLC Sender (µs)	PHY→LLC Empfänger (µs)
2	620	114	465
200	667	113	515
500	732	116	573
1500	957	118	830
Lin. Regression ($R^2 > 0,99$)	$0,2243x + 620$	–	$0,2437x + 461$

(b) Gemessene Daten

Abbildung 4.2: Gemessene Latenzen während der Übertragung bei verschiedenen Paketgrößen

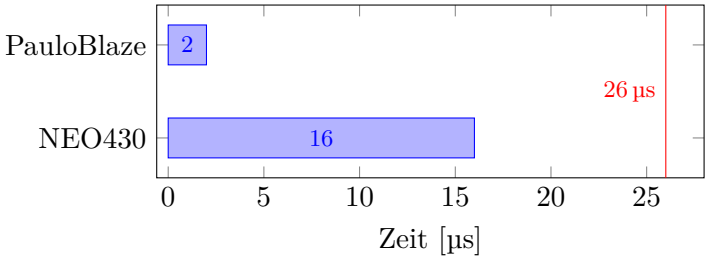


Abbildung 4.3: Aktive Zeit der Softcores in der RIFS-Zeit

die Varianten 1xNEO bzw. 1xPauloBlaze in der RIFS-Zeit aktiv sind. Beide Varianten verletzen die Zeitschranke nicht, es wird aber deutlich, dass der PauloBlaze, der 2 µs aktiv ist, mehr zeitliche Reserven hat als der NEO430, der 16 µs benötigt. Dies ist wieder auf den höheren Instruktionsdurchsatz des PauloBlaze zurückzuführen. Die NEO430-Variante konnte überhaupt erst durch die in Abschnitt 3.5.2 dargestellten Optimierungen die Echtzeitbedingung erfüllen.

In der Variante 1xPauloBlaze ist die MAC-Schicht in der RIFS-Zeit 24 µs inaktiv. Wenn eine Inkompatibilität mit bestehenden HomePlug-Geräten in Kauf genommen wird, kann die RIFS-Zeit entsprechend verkürzt werden. Dies erhöht die Auslastung des Kanals und damit die erreichbare Datentransferrate. In der Variante 1xNEO ist eine Verkürzung ebenfalls möglich, jedoch ist die MAC-Schicht hier nur in 10 µs inaktiv. Die Variante mit PauloBlaze bietet also mehr Optimierungspotenzial.

Der Datendurchsatz von 8,07 MHz des Softcore-MAC-Layers liegt nahe am theoretischen Maximum von 8,4 MHz. Die Referenzvariante erreicht durch die Implementierung in Hardware einen deutlich höheren Durchsatz [14]. Die Übertragungsrate ist jedoch durch den HomePlug-Standard und die Bandbreite des Kanals fest begrenzt. Die Hardware-Implementierung bringt hier daher keinen Vorteil, benötigt aber mehr als drei Mal so viele Hardware-Ressourcen (vgl. Abschnitt 4.1). Der Einsatz von Softcores ermöglicht es, den real notwendigen Durchsatz mit deutlich weniger Ressourcen zu erreichen. Weiterhin ermöglich die Softcore-Realisierung eine einfachere Anpassung des System.

4.4 Anpassbarkeit

Der im Rahmen dieser Arbeit entwickelte MAC-Layer erfüllt die Vorgaben des HomePlug-Standards. Durch den Einsatz von Softcore-Prozessoren für die Steuerung aller relevanten Vorgänge ist jedoch auch eine Anpassung des verwendeten Protokolls möglich. Sofern der

MAC-Layer in einem abgeschlossenen System eingesetzt wird und eine Kompatibilität zu bestehenden HomePlug-Geräten nicht notwendig ist, kann so das Protokoll auf die Anforderungen des Einsatzes optimiert werden. Mögliche Änderungen wären beispielsweise die Erhöhung der Segmentgröße, die Verwendung einer anderen Modulationsart oder von anderen oder einer größerern Anzahl an Trägerfrequenzen.

In der zum Zeitpunkt dieser Arbeit bereits bestehenden Implementierung wurde der komplette MAC-Layer in VHDL beschrieben (vgl. Abschnitt 3.1). Dies schränkt die Anpassbarkeit ein, da VHDL-Code eine aufwendige Beschreibung der Hardware erfordert. Zudem ist bereits bestehende Hardware zur Laufzeit nur schwierig änderbar.

In der in dieser Arbeit vorgestellten Implementierung sind nur wenige Teilmodule in VHDL implementiert. Alle Teile sind dabei durch die Softcores konfigurierbar, sodass eine Änderung daran nur bei weitreichenden Anpassungen notwendig ist.

Der Top Softcore und der/die Bottom Softcore(s) in den NEO-Varianten sind in C programmiert. Dadurch ist es auf einfache Weise möglich, Änderungen an dem System vorzunehmen. Sofern keine Veränderung der Hardware erforderlich ist, können Anpassungen so sogar noch an einem fest implementierten System (ASIC) vorgenommen werden.

In der Variante mit einem PauloBlaze als Bottom Softcore sind Änderungen aufwendiger, da der Assemblercode des PauloBlaze eine höhere Einarbeitungszeit erfordert als der C-Code des NEO430. Zusätzlich ist der Instruktionsspeicher von 1024 Instruktionen aktuell vollständig gefüllt, sodass Erweiterungen der Software auch eine Erweiterung des Speichers benötigten. Der Bottom Softcore übernimmt jedoch nur die Ansteuerung des PHY-Layers und ist durch den Top Softcore in gewissem Maße konfigurierbar. Eine Änderung ist daher nur bei größeren Protokolländerungen notwendig.

Wie Abbildung 4.3 zeigt, bieten beide Softcore-Varianten in der kritischen RIFS-Zeit noch einige zeitliche Reserven für Erweiterungen. Der Top Softcore unterliegt keiner harten zeitlichen Beschränkung, daher sind hier Anpassungen problemlos möglich. Werden zusätzliche Berechnungen durchgeführt und damit die Latenz des Systems erhöht, sinkt jedoch die maximale Transferrate.

4.5 Zusammenfassung

Alle implementierten Varianten benötigen deutlich weniger FPGA-Ressourcen als die Refrenzvariante und erreichen trotzdem eine Transferrate in der Nähe des theoretischen Maximums. Sowohl die Variante 1xNEO als auch die Variante 1xPauloBlaze bieten eine deutlich höhere Anpassbarkeit als die bestehende VHDL-Implementierung. Die Variante mit NEO430 ist dabei durch die einfachere Programmierbarkeit des NEO430 als Bottom Softcore noch etwas flexibler als die Variante mit PauloBlaze. Die Variante 1xNEO benötigt jedoch mehr Ressourcen.

Durch den höheren Instruktionsdurchsatz erreicht die Variante 1xPauloBlaze die zeitlichen Vorgaben einfacher und bietet an dieser Stelle mehr Reserven. Durch den direkten Zusammenhang zwischen Assemblerinstruktionen und benötigter Zeit ist das Einhalten der Zeitschranken leicht zu überprüfen. Die Variante 1xNEO ist durch den Einsatz von C leichter zu programmieren, es erfordert jedoch einigen Aufwand, die notwenigen Zeitvorgaben zu erreichen.

5 Zusammenfassung und Ausblick

Ziel der vorliegenden Arbeit war die Implementierung eines MAC-Layers nach dem HomePlug-1.0.1-Standard unter der Verwendung von Softcore-Prozessoren. Es wurde dabei untersucht, ob sich gegenüber der bereits bestehenden, reinen VHDL-Implementierung, Vorteile bei Ressourcennutzung und Anpassbarkeit ergeben.

In Kapitel 2 wurden zunächst die in dieser Arbeit verwendeten Softcore-Prozessoren, NEO430 und PauloBlaze, vorgestelllt und erläutert. Es wurde dann auf CRC-Prüfsummen und mögliche Hardware-Implementierungen dieser eingegangen. Abschließend wurde ein Überblick über den HomePlug-Standard und die Funktionen des MAC-Layers gegeben.

In Kapitel 3 wurde die Implementierung des Systems detailliert vorgestellt. Die Besonderheiten der Architektur stellen dabei die Aufteilung in Top und Bottom Softcore sowie die Anbindung der Speicher über FIFO-Interfaces, sogennante Streams, dar. Der Top Softcore kommuniziert mit der LLC-Schicht und steuert das Gesamtsystem. Der Bottom Softcore ist mit der PHY-Schicht verbunden und nimmt die zeitliche Ansteuerung dieser vor. Es wurden drei Varianten des Bottom Softcores implementiert um die Unterschiede verschiedener Softcores untersuchen zu könnnen. Eine Version mit zwei NEO430 folgt der in der Spezifikation vorgebenen Aufteilung zwischen Sender und Empfänger. Diese Variante hat sich aufgrund der notwendigen Synchronisation der beiden Softcores als schwierig zu implementieren herausgestellt. Durch die Kombination der Zustandsautomaten von Sender und Empfänger konnten diese auf einen einzelnen NEO430 zusammengelegt werden. Als dritte Variante wurde der NEO430 durch den einfacheren PauloBlaze ausgetauscht.

Die drei implementierten Varianten wurden dann in Kapitel 4 untersucht. Es konnte gezeigt werden, dass sowohl die Varianten mit NEO430 als auch die Variante mit PauloBlaze die Zieltaktfrequenz von 50 MHz erreicht. Der Datendurchsatz von 8,07 Mbit/s ist mit dem theoretischen Maximum von 8,4 MHz vergleichbar. Die Programmierung der Softcores in C ermöglicht eine einfachere Anpassung der Funktion. Das entwickelte Design benötigt 1730 LUTs, 1198 Register und 14 BlockRAMs. Dies entspricht einer Einsparung

von 78 % (LUTs, Register) bzw. 55 % (BlockRAMs) gegenüber der bestehenden Variante ohne Softcores.

Am Ende dieser Arbeit konnte gezeigt werden, dass die Implementierung eines HomePlug-MAC-Layers unter Zuhilfenahme von Softcores möglich ist. Dadurch ergeben sich klare Vorteile bei der Anpassbarkeit. Gegenüber einer reinen VHDL-Implementierung werden außerdem insbesondere viermal weniger FPGA-Ressourcen benötigt.

Aufbauend auf dieser Arbeit sind weitere Anpassungen denkbar. Die zeitlichen Reserven in der RIFS-Zeit ermöglichen es, diese Zeitspanne zu verkürzen. Mit dieser Abweichung vom Standard ließe sich der Datendurchsatz erhöhen. Alternativ könnte die verbleibende Zeit genutzt werden, um weitere Anwendungen oder Dienste auf MAC-Ebene zu implementieren.

Um die Ressourcennutzung weiter zu optimieren, könnten die verwendeten CRC-Blöcke durch Sende- und Empfangspfad gemeinsam genutzt werden. Aktuell sind diese doppelt vorhanden, werden jedoch nie gleichzeitig genutzt. Die Timing-Analyse zeigt, dass es problemlos möglich wäre, das Design mit einem Takt von 100 MHz zu betreiben. Hierdurch würde sich der Durchsatz nochmal steigern und die Latenzen verringern lassen.

A Appendix

A.1 Speicherkarten

Tabelle A.1: Speicherkarte des NEO430

Adresse	Funktion	Top	Bottom	Mode
0x0000-0x7FFF	IMEM (ROM)	✓	✓	
0x8000-0x9000	DMEM (RAM)	✓	✓	
0xA000-0xBFFF	LLC Memory (8kB)	✓		
0xC000-0xC1FF	Tonemap Memory (512 Bytes)	✓		
0xF000-0xF07F	Shared Memory (32 Bytes)	✓	✓	
0xF100-0xF112	Header Memory (18 Bytes)	✓		
0xFF30-0xFF38	Frame Control Memory		✓	
0xFF30	TX SFC Variant Field			R/W
0xFF32	TX EFC Variant Field			R/W
0xFF34	CC, DT and RX FCCS			R/W
0xFF36	RX FC Variant Field			R/-
0xFF40-0xFF4C	CRC Stream RX	✓		
0xFF40	Control Register			R/W
0xFF42	Data Register			R/-
0xFF44	Address Register			R/W
0xFF46	Key/IV 0			R/W
0xFF48	Key/IV 1			R/W
0xFF4A	Key/IV 2			R/W
0xFF4C	Key/IV 3			R/W
0xFF50-0xFF56	Reassembly Stream	✓		
0xFF50	Control Register			R/W
0xFF52	Address Register			R/W

0xFF54	Data Register			R/-
0xFF56	Frame Check Sequence			R/-
0xFF60-0xFF66	Segement Stream	✓		
0xFF60	Control Register			R/W
0xFF62	LFB Address Register			R/W
0xFF64	LFB Limit Register			R/W
0xFF66	BPAD Limit Register			R/W
0xFF70-0xFF7C	CRC Stream TX	✓		
0xFF70	Control Register			R/W
0xFF72	Data Register			-/W
0xFF74	Address Register			R/W
0xFF76	Key/IV 0			R/W
0xFF78	Key/IV 1			R/W
0xFF7A	Key/IV 2			R/W
0xFF7C	Key/IV 3			R/W
0xFF80	(Debug Port)	✓	✓	
0xFF82	Zufallsgenerator	✓	✓	
0xFF84-0xFF86	LLC Kommunikation	✓		
0xFF84	Low Word			R/W
0xFF86	High Word			R/W
0xFF88	Tonemap Index		✓	
0xFF8A	Capacity ROM	✓		
0xFF8C	Aktuelle FCS		✓	
0xFF8E	MAC Adresse	✓		
0xFFB0-0xFFB6	GPIO			
0xFFC0-0xFFCC	Timer			
0xFFF0-0xFFFE	Interruptvektoren			

Tabelle A.2: Speicherkarte des PauloBlaze

Adresse	Funktion	Mode
0x00-0x01	(Debug Port)	
0x20-0x27	FrameControl-Speicher	
0x20-0x21	TX SFC Variant Field	R/W
0x22-0x23	TX EFC Variant Field	R/W
0x24	CC, DT and RX FCCS (low)	R/W
0x25-0x26	RX FC Variant Field	R/-
0x27	CC, DT and RX FCCS (high)	R/-
0x30	GPIO	R/W
0x40	Zufallsgenerator	R/-
0x50-0x51	Aktuelle FCS	R/-
0x60	Tonemap-Index	-/W
0x70-0x72	Timer	
0x70-0x71	Timer0	R/W
0x72	Timer1	R/W
0x80-0x9F	Shared Memory	

A.2 Code-Listings

Listing A.1: Kopieren mit Inline-Assembler

```
uint16_t n = 0;
asm volatile (
        "mov %[c], %[n]          \n\t" //n = c
        "add #7, %[n]            \n\t" //n += 7
        "rra %[n]                \n\t"
        "rra %[n]                \n\t"
        "rra %[n]                \n\t" //n /= 8
        "and #7, %[c]            \n\t" //c %= 8
        "add %[c], %[c]          \n\t" //c *= 2 (word addressing)
        "add #9f, %[c]           \n\t" //add base address of jump table
        "br @%[c]                \n\t" //indirect jump through table
        "9: .word 0f, 1f, 2f, 3f, 4f, 5f, 6f, 7f \n\t" //jump table
        "0: mov.b @%[f]+, @%[t] \n\t" //*t = *f++
        "7: mov.b @%[f]+, @%[t] \n\t" //...
        "6: mov.b @%[f]+, @%[t] \n\t"
        "5: mov.b @%[f]+, @%[t] \n\t"
        "4: mov.b @%[f]+, @%[t] \n\t"
        "3: mov.b @%[f]+, @%[t] \n\t"
        "2: mov.b @%[f]+, @%[t] \n\t"
        "1: mov.b @%[f]+, @%[t] \n\t"
        "add #-1, %[n]           \n\t" //n--
        "jnz 0b                  \n\t" //goto 0, if n != 0
        : [n] "+r" (n), [c] "+r" (count), [f] "+r" (from)
        : [t] "r" (&CRC_TX_DATA)
        );
```

A.3 Berechnung der Datenmenge pro Segment

Zur Berechnung der Anzahl an Bytes in einem Segment werden folgenden Daten benötigt:

numCarr Anzahl der verfügbaren Träger (bis zu 84)

NumBitsPerCarr Anzahl des Bits pro Träger (ROBO: 0,25, DBPSK: 1, DQPSK: 2)

ConvCodeRate Die Fehlerkorrekturrate, 0,5 oder 0,75

PhyBlockSize Die Anzahl der ODFM-Symbole im Block (20 oder 40)

Es sind dabei nur die in Tabelle 3.4 angegeben Kombinationen gültig. Mit diesen Informationen kann nun die maximale Anzahl der Bytes $maxRSBytes$ in einem Reed-Solomon-Block bestimmt werden:

$$maxRSBytes = \left\lfloor \frac{NumCarr * NumBitsPerCarr * PhyBlockSize * ConvCodeRate - 6}{8} \right\rfloor$$

Mit diesem Wert kann nun die Anzahl $NumRSblocks$ und Größe $RSblockSize$ der benötigten Reed-Solomon-Blöcken berechnet werden:

$$NumRSblocks = \left\lfloor \frac{MaxRSbytes + 239}{255} \right\rfloor$$

$$RSblockSize = min\left(254, \left\lfloor \frac{MaxRSbytes}{NumRSblocks} \right\rfloor\right)$$

Wurden diese beiden Werte berechnen kann nun die gesuchte Anzahl an Bytes in einem 20er bzw. 40er-Block bestimmt werden. Die Berechnung hängt dabei von der Modulationsart ab:

$$MaxDataBytes_{ROBO} = (RSblockSize - 8) * NumRSblocks$$
$$MaxDataBytes_{DBPSK,DQPSK} = (RSblockSize - 16) * NumRSblocks$$

Der so berechnete Wert gibt die Anzahl an Bytes an, die mit 20 bzw. 40 OFDM-Symbolen übertragen werden können. Ein Segment kann maximal 160 Symbole enthalten [9, 2.2.2.2].

A.4 Zustandsautomaten

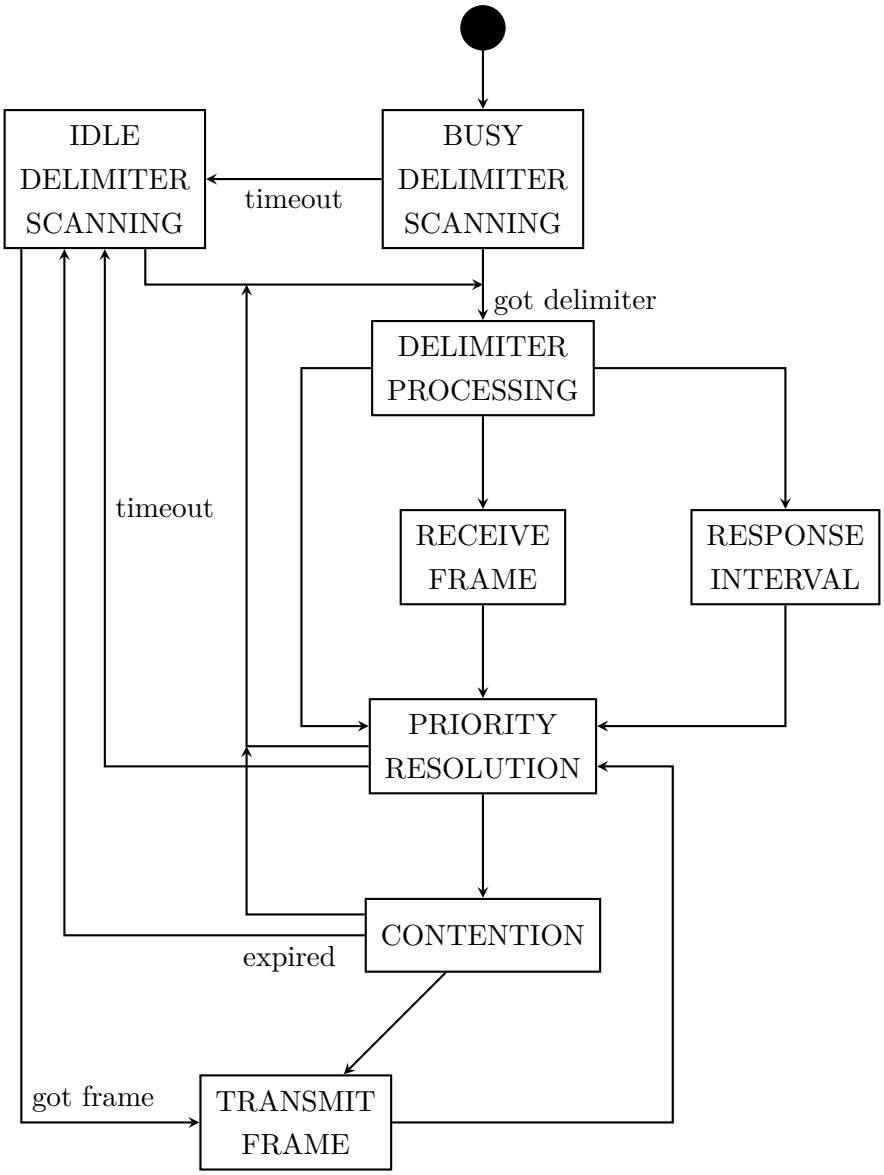
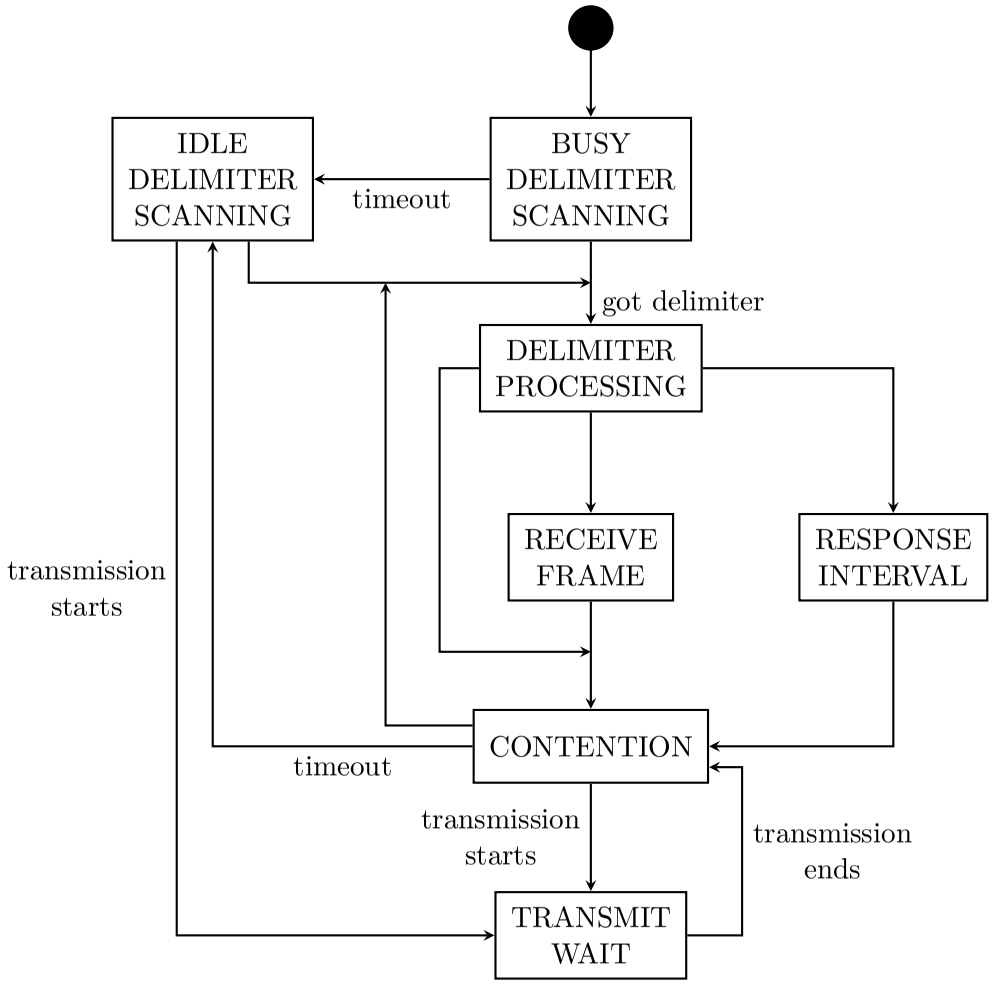
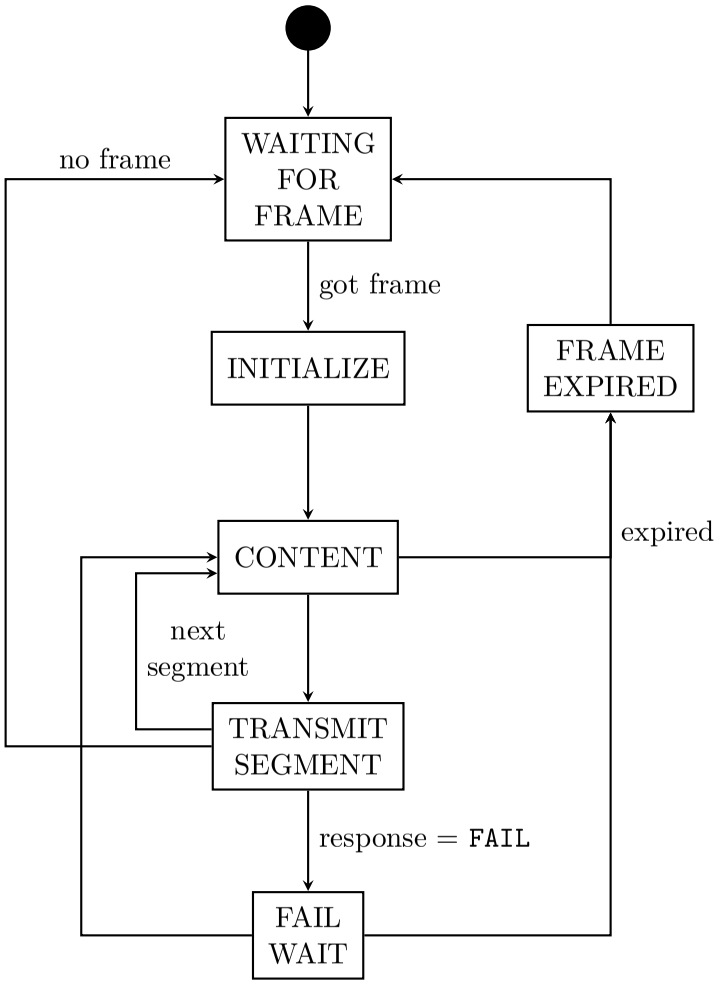


Abbildung A.1: Kombinierter Zustandsautomat wie er in der Software für 1xNEO und 1xPauloBlaze eingesetzt wird (vereinfacht).



(a) Zustandsautomat des Empfängers laut Spezifikation (vereinfacht) [9, C.4.1]



(b) Zustandsautomat des Senders laut Spezifikation (vereinfacht) [9, C.4.2.4]

Abbildung A.2: Zustandsautomaten der HomePlug-Spezifikation

A.5 Syntheseergebnisse

Tabelle A.3: Syntheseergebnisse Variante 2xNEO

Name	LUTs	Registers	BlockRAM
Top Softcore (NEO430)	777	295	4.5
Bottom Softcore (Sender, NEO430)	667	326	4.5
Bottom Softcore (Empfänger, NEO430)	657	315	0.5
Shared Memory	271	51	0
Segment Stream	102	82	0
Reassembly Stream	79	110	0.5
CRC Stream TX	77	141	0
CRC Stream RX	63	129	0
FrameControl Speicher	47	94	0
HeaderMem	30	17	0
Tonemap Controller	25	113	0
Tonemap Speicher	15	1	0.5
Capacity ROM	14	18	1
ReasMem	8	2	2
LLC-Interface	3	49	0
Random-Generator	2	16	0
LFB	1	3	0.5
Softcores	2101	936	9,5
Sonstiges	735	859	4,5
Gesamt	2836	1795	14

Tabelle A.4: Syntheseergebnisse Variante 1xNEO

Name	LUTs	Register	BlockRAM
Top Softcore (NEO430)	772	282	4,5
Bottom Softcore (NEO430)	626	290	1,5
Segment Stream	102	82	0
CRC Stream TX	80	141	0
Reassembly Stream	78	110	0,5
Shared Memory	74	34	0
CRC Stream RX	65	129	0
FrameControl Speicher	47	81	0
HeaderMem	30	17	0
TonemapController	27	113	0
Tonemap-Speicher	16	1	0,5
Capacity ROM	14	18	1
ReasMem	8	2	2
LLC-Interface	3	49	0
Random-Generator	1	16	0
LFB	1	3	0,5
Softcores	1398	572	6
Sonstiges	543	829	4,5
Gesamt	1941	1401	10,5

Tabelle A.5: Syntheseergebnisse Variante 1xPauloBlaze

Name	LUTs	Register	BlockRAM
Top Softcore (NEO430)	775	282	4,5
Bottom Softcore (PauloBlaze)	366	74	0
Segment Stream	102	82	0
CRC Stream TX	84	141	0
Reassembly Stream	81	110	0,5
CRC Stream RX	65	129	0
Shared Mem	63	25	0
FrameControl Speicher	43	76	0
PauloBlaze Timer	32	29	0
Tonemap Controller	31	113	0
HeaderMem	30	17	0
PauloBlaze ROM	17	0	0,5
Tonemap Speicher	16	1	0,5
Capacity ROM	14	18	1
ReasMem	8	2	2
LLC-Interface	3	49	0
Random-Generator	1	16	0
LFB	1	3	0,5
Softcores	1190	385	5
Sonstiges	540	813	4,5
Gesant	1730	1198	9,5

Literaturverzeichnis

[1] Xilinx, Hrsg., *Virtex-5 Family Overview*, 2015

[2] H. Blume, *Vorlesung FPGA-Entwurftechnik*, Leibniz Universität Hannover, 2016

[3] Dipl.-Ing. S. Nolting, Hrsg., *The NEO430 Processor*, 2017

[4] OpenCores, Hrsg., *Wishbone B4*, 2010

[5] O. Girad, Hrsg., *openMSP430*, Rev 1.16, 2016

[6] K. Chapman, Xilinx, Hrsg., *PicoBlaze for Spartan-6, Virtex-6 and 7-Series (KCPSM6)*, 2012

[7] P. R. Genßler, Technische Universität Dresden, Deparment of Computer Science, Institute for Computer Engineering, Chair for VLSI Design, Diagnostic and Architecture, *PauloBlaze*, 2015

[8] Xilinx, Hrsg., *PicoBlaze 8-bit Embedded Microcontroller User Guide*, 2011

[9] Homeplug Powerline Alliance, Hrsg., *HomePlug 1.0.1 Specification*, 2001

[10] FPGA Soft-Core – Microcontroller.net, https://www.mikrocontroller.net/articles/FPGA_Soft_Core

[11] Seite *Duff's Device*. In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 5. April 2016, 16:03 UTC. https://de.wikipedia.org/w/index.php?title=Duff%E2%80%99s_Device&oldid=153198489

[12] Seite *Loop unrolling*. In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 27. Juli 2017, 12:06 UTC. https://de.wikipedia.org/w/index.php?title=Loop_unrolling&oldid=167640790

[13] Xilinx, Hrsg., *7 Series FPGAs Configurable Logic Block User Guide*, 2016

[14] C. Uhlemann, *Konzeptionierung, Implementierung und Verifikation eines MAC-Layers für Paket-basierte Powerline Kommunikation*, 2017

[15] Xilinx, Hrsg., *7 Series FPGAs Data Sheet: Overview*, 2017

[16] A. S. Tanenbaum, D. J. Wetherall, *Computernetzwerke*, 5. Auflage, München: Pearson Deutschland, 2012

[17] D. Schellekens, B. Preneel, I. Verbauwhede, *FPGA Vendor Agnostic True Random Number Generator*, 2006 International Conference on Field Programmable Logic and Applications, Madrid, 2006

[18] Seite *Linear rückgekoppeltes Schieberegister*. In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 8. Februar 2017, 16:28 UTC. https://de.wikipedia.org/w/index.php?title=Linear_rückgekoppeltes_Schieberegister&oldid=162445061)

[19] Seite *Zyklische Redundanzprüfung*. In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 4. August 2017, 15:06 UTC. https://de.wikipedia.org/w/index.php?title=Zyklische_Redundanzprüfung&oldid=167868640

[20] E. Stavinov, *A Practical Parallel CRC Generation Method* in: Circuit Cellar, Ausgabe 234, 2010, S. 38–45

[21] M. K. Lee et al., *HomePlug 1.0 Powerline Communications LANs – Protocol Description and Performance Results* in International Journal of Commununication Systems, Jahrgang 16, Ausgabe 5, 2003, S. 447–473

[22] O. Girad, Hrsg., *openMSP430 Area and speed Analysis*, Rev. 1.0, 2012

[23] HomePlug Powerline Alliance, Hrsg., *HomePlug 1.0 Technology White Paper*, o.J.

[24] Open PicoBlaze Assembler – Optasm 1.3 Documentation, https://kevinpt.github.io/opbasm/

[25] Seite *Trägerfrequenzanlage*. In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 22. September 2017, 12:54 UTC. https://de.wikipedia.org/w/index.php?title=Tr%C3%A4gerfrequenzanlage&oldid=169332549

[26] M. Zimmermann, K. Dostert, *A Multipath Model for the Powerline Channel* in IEEE Transactions on Communications, Jahrgang 50, Ausgabe 4, 2002, S. 553–559

[27] Philipps H., *Performance measurements of powerline channels at high frequencies.* in Proceedings of International Symposium on Power-line Communications and its Applications, 1998, S. 229–237.

Alle Online-Quellen wurden zuletzt am 13. Oktober 2017 abgerufen.