RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Jürgen Drexler

Entwurf und Implementierung einer parallelen Netzwerkschnittstelle zum Betrieb Künstlicher Neuronaler Netze

Diplomarbeit

HD-KIP-09-05

KIRCHHOFF-INSTITUT FÜR PHYSIK

Fakultät für Physik und Astronomie
Ruprecht-Karls-Universität Heidelberg

Diplomarbeit
im Studiengang Physik

vorgelegt von
Jürgen Drexler
aus Neuburg an der Donau
2008

Entwurf und Implementierung einer parallelen Netzwerkschnittstelle zum Betrieb Künstlicher Neuronaler Netze

Die Diplomarbeit wurde von Jürgen Drexler ausgeführt am
Kirchhoff-Institut für Physik
unter der Betreuung von
Herrn Prof. Dr. Karlheinz Meier

Entwurf und Implementierung einer parallelen Netzwerkschnittstelle zum Betrieb Künstlicher Neuronaler Netze

Diese Arbeit beschreibt die Parallelisierung einer bestehenden, Token Ring-basierten Netzwerkschnittstelle zur Ansteuerung neuronaler Netzwerk-Module. Mit der entworfenen Schnittstelle können Datenraten im Bereich von 1 GBit/s realisiert werden. Die hierbei implementierte Logik wird im Rahmen des FACETS-Projekts eingesetzt um großskalige künstliche neuronale Netze zu steuern.
Die zu diesem Zweck entworfenen Module wurden in der Hardwarebeschreibungssprache VHDL entwickelt. Als Zieltechnologie stand ein FPGA auf einer Trägerplatine zur Verfügung. Vorgabe bei der Entwicklung war es die Bandbreite von 1 GBit/s, die durch eine Gigabit-Ethernet-Schnittstelle zur Verfügung gestellt wird, optimal auszunutzen und zugleich die Komplexität der Logik gering zu halten.
Die Arbeit beschreibt den Aufbau und das Zusammenwirken der entwickelten Logik. Das korrekte Verhalten der Module wird mit Hilfe funktionaler Simulationen nachgewiesen. Es wird gezeigt, dass die erreichbare Performance die Anforderungen erfüllt. Da das Ziel der Arbeit ein synthesefähiges Design war, wurden Tests in der realen Hardware durchgeführt, welche die korrekte Funktionalität bestätigen.

Design and Implementation of a parallel Network Interface to control Artificial Neural Networks

This thesis describes the parallelization of an existing, token ring-based network interface to control artificial neural network modules. The designed interface allows to realize data rates in the range of 1 GBit/s. The implemented logic will be used within the FACETS project to control large-scale artificial neural networks.
The designed modules were written in the hardware description language VHDL. The target technology was an FPGA on a backplane. The requirement for the developement was, to exploit the bandwidth of 1 GBit/s provided through a gigabit ethernet interface, while keeping the complexity low.
This thesis describes the composition and interaction of the developed logic. The correct behavior of the modules is proven with functional simulations. It is shown, that the achievable performance meets the requirements. Since the goal of this thesis was a synthesizable design, tests with the real hardware system prove correct behavior.

Inhaltsverzeichnis

Abbildungsverzeichnis

Einleitung und Motivation

Die grundlegenden Vorgänge und Mechanismen der Informationsverarbeitung und -speicherung innerhalb des Gehirns zu verstehen ist wohl eine der interessantesten Forschungsaufgaben unserer Zeit. Das menschliche Gehirn besteht aus etwa 100 Milliarden Neuronen, welche die kleinste informationsverarbeitende und -transportierende Einheit darstellen sowie ca. 100 Billionen Synapsen, welche für die Weiterleitung von Reizen zwischen den Neuronen verantwortlich sind [28]. Es stellt somit ein außerordentlich komplexes und darüber hinaus äußerst leistungsfähiges System dar, das in manchen Bereichen, wie z.B. der Mustererkennung, gewöhnlichen Computersystemen weit überlegen ist.

Bei der Betrachtung der Struktur eines neuronalen Netzes stellt man fest, dass es sich hierbei um ein relativ homogenes Gebilde handelt. Dies stellt einen entscheidenden Unterschied zu einem herkömmlichen Rechnersystem dar, welches auf der von Neumann- oder Harvard-Architektur aufbaut und über mehrere, in ihrer Aufgabe und Funktion hochgradig spezialisierte und lokal getrennte Einheiten – wie z.B. Speicher, Arithmetisch-logische Einheit etc. – verfügt. In einem neuronalen Netz sind logische Funktionen und Speicher über das gesamte Gebilde verteilt gelagert, während bei herkömmlichen Computern Daten an bestimmten Adressen abgelegt werden und somit lokalisiert sind.
Die Aufnahme und Verarbeitung von Information verläuft in einem neuronalen Netz vollkommen parallel, wohingegen beim Ablauf eines Programms auf einem PC[1] eine sequentielle Abarbeitung von Operationen stattfindet. Man spricht daher bei neuronalen Netzen von „Parallel Distributed Processing“-Systemen [24], während dem klassischen von Neumann-Rechner ein sequentieller „Single Instruction Stream, Single Data Stream“-Ansatz zugrunde liegt [20].
Ein weiterer zentraler Unterschied zu gewöhnlichen Rechnern liegt in der Lernfähigkeit neuronaler Netze. Sie stellen im Allgemeinen keine fest verdrahteten oder programmierten Gebilde dar, sondern lernen ihr Verhalten aufgrund von bestimmten, vorgegebenen Eingabereizen in einer Lernphase. Daraus folgt eine sehr hohe Flexibilität des Netzes um verschiedenste Aufgaben zu übernehmen.
Eine weitere wichtige Eigenschaft neuronaler Netze ist ihre Fehlertoleranz. Diese äußert sich zum einen in der Toleranz gegenüber internen Defekten, zum anderen in der Fähigkeit eine 'Unschärfe' bzw. ein 'Rauschen' in den Eingangsreizen tolerieren zu können.
Darüber hinaus weisen biologische neuronale Netze eine vergleichsweise geringe Leistungsaufnahme auf. So nimmt beispielsweise ein menschliches Gehirn nur ca. 20 Watt auf, und ist dennoch in der Lage sehr komplexe Aufgaben zu bewältigen.

All diese Eigenschaften haben zu dem Wunsch geführt, einen tieferen Einblick in die ablaufenden Prozesse in einem neuronalen Netz zu bekommen. Das EU-Projekt FACETS[2], in dessen Rahmen die vorliegende Arbeit angefertigt wurde, hat sich auf die Funktionsweise des visuellen Cortex spezialisiert und es sich zum Ziel gesetzt,

[1] Personal Computer
[2] Fast Analog Computing with Emergent Transient States

sowohl theoretische als auch experimentelle Grundlagenforschung zu betreiben, die ein besseres Verständnis für die Vorgänge im Gehirn ermöglichen soll [7]. Letztendlich sollen dadurch vollkommen neuartige Computer-Paradigmen abgeleitet werden, die die oben aufgeführten Vorteile eines neuronalen Netzwerks umsetzen.

Gegenstand der Arbeit In der Electronic Vision(s) Gruppe der Universität Heidelberg wurde eine Hardware-Plattform entwickelt, die es ermöglicht ein Netzwerk aus bis zu 6.144 künstlichen Neuronen und fast 1.600.000 Synapsen zu betreiben. Die Neuronen und Synapsen sind dabei auf bis zu 16 ASICs[3] untergebracht, die jeweils auf einer Trägerplatine (*Nathan-Modul*) montiert werden. Die Nathan-Module können auf einer Backplane platziert und parallel betrieben werden [9]. Da dieses System die zeitlichen Verläufe der modellierten biologischen Funktionen um einen Faktor von bis zu 10^5 beschleunigt, fallen sehr hohe Datenraten an: das Netzwerk muss von außen mit Eingangsreizen versorgt und die Reaktion des Systems darauf muss anschließend ausgelesen werden. Eine schnelle Anbindung der Backplane an die Außenwelt ist also von entscheidender Bedeutung. Daher wurde eine Gigabit-Ethernet-Schnittstelle implementiert, die eine schnelle Kommunikation mit einem PC ermöglicht [10]. Zudem wird hierdurch das Systems über eine genormte Schnittstelle an die Außenwelt angebunden. Ziel der vorliegenden Arbeit ist es die Datenpakete, die über die Ethernet-Anbindung geliefert werden, weiterzuverarbeiten und zu den Nathan-Modulen zu transportieren sowie die von den Nathan-Modulen kommenden Daten über das Ethernet zum PC zu transportieren. Um den angestrebten Übertragungsraten gerecht zu werden, wurde die Schnittstelle so konzipiert, dass eine parallele Kommunikation mit allen Nathan-Modulen ermöglicht wird. Die dazu erforderliche Logik wurde in einem FPGA[4], der sich auf der Backplane befindet, implementiert. Die Arbeit baut dabei auf Teilen eines bereits bestehende Cores auf [16].

Gliederung der Arbeit Das folgende Kapitel bietet einen Überblick über die zum Verständnis der Arbeit notwendigen Grundlagen. Dabei werden zunächst die Grundzüge des digitalen Systementwurfs dargestellt. Danach wird auf Netzwerkarchitekturen eingegangen, wobei auch die beiden Normen Ethernet und Token Ring diskutiert werden. Das anschließende Kapitel befasst sich mit der in der Electronic Vision(s) Gruppe entwickelten und verwendeten Hardware. Es wird auf die eigentliche Problemstellung dieser Arbeit hingeführt und der FPGA-Core beschrieben, auf dem die Arbeit aufbaut. In den Kapiteln 3 und 4 wird auf den Kern der Arbeit – also die implementierten Module und deren genaue Funktionsweise – eingegangen. Kapitel 5 stellt schließlich dar, wie die Funktionalität der entworfenen Logik in Simulationen und Hardware-Experimenten verifiziert wurde. Hierbei wird auch die erreichbare Performance des Gesamtsystems diskutiert.

[3] Application Specific Integrated Circuit, anwendungsspezifische integrierte Schaltung
[4] Field Programmable Gate Array

1 Grundlagen

1.1 Digitale Systeme

Dieses Kapitel beschäftigt sich mit den grundlegenden Prinzipien der digitalen Schaltungstechnik. Es soll zunächst erläutert werden, was unter einem digitalem System zu verstehen ist und welche Möglichkeiten der Darstellung eines solchen Systems es gibt. Anschließend soll die Theorie endlicher Automaten sowie die Verwendung von Pipelines zur Verbesserung der Performance diskutiert werden. Am Ende dieses Kapitels wird schließlich auf FPGAs eingegangen, die eine Zieltechnologie zur Realisierung verschiedenartigster, digitaler Schaltungen darstellen.

Beim Entwurf eines digitalen Systems gibt es drei verschiedene *Sichten* sowie mehrere Abstraktions-*Ebenen* (siehe Abb. 1.1) [29]. Bei den Sichten wird unterschieden zwischen der Verhaltens-Sicht, die beschreibt welche Funktionen ein Hardware-Bauteil zu erfüllen hat und wie diese implementiert sind. In der Struktursicht werden Blöcke wie Speicher, Prozessoren und Interface-Einheiten verwendet um das System zu beschreiben. Die Geometriesicht liefert letztendlich Aussagen über das Layout aller Komponenten und Leitungen auf dem Substrat.

Die Ebenen stellen von außen nach innen immer feinere Implementationsdetails dar. Während in der Systemebene noch das Verhalten des Gesamtsystems bzw. komplexe Subkomponenten wie CPUs etc. die betrachteten Objekt sind, befasst sich die unterste Schicht mit einzelnen Transistoren und Leitungsbahnen. Der Entwurf eines Systems erfolgt meist von außen nach innen. Dabei ist der Entwickler nicht auf eine einzelne Sicht beschränkt; vielmehr kann beliebig oft zwischen den einzelnen Sichten gewechselt werden.

Die vorliegende Arbeit setzt sich mit der Implementierung eines System auf einem FPGA auseinander. Bei einem derartigen Systementwurf nimmt die Register-Transfer-Ebene eine besondere Position ein; daher soll an dieser Stelle näher darauf eingegangen werden.

1.1.1 Register-Transfer-Logik

In einem digitalen System werden Daten- und – sofern es sich um ein synchrones Systems handelt – auch Zeitwerte in einer diskretisierten Form dargestellt. Die Datensignale nehmen dabei die beiden Werte `'0'` (logisch Null, *low*) oder `'1'` (logisch Eins, *high*) an, die wiederum durch (analoge) Spannungspegel repräsentiert werden. Die Diskretisierung der Zeit erfolgt durch die Einführung eines Taktsignals (*Clock*).

Das grundlegende Bauteil der digitalen Hardware ist das *FlipFlop*. Es stellt ein bi-stabiles System dar, d.h. es kann genau zwei stabile Zustände annehmen und speichern. FlipFlops lassen sich im Wesentlichen in zwei Gruppen einteilen: zum einen gibt es *pegelgesteuerte* FlipFlops (*Latches*), bei denen eine Änderung des intern gespeicherten Wertes vom Signalpegel eines oder mehrerer Eingangssignale abhängt; dies können z.B. ein enable-Signal oder auch ein set/reset-Signal sein. Zum anderen existieren *flankengesteuerte* FlipFlops, die synchron zu einem Taktsignal arbeiten und nur bei einem Flankenwechsel ihren Zustand ändern können. Üblicherweise ver-

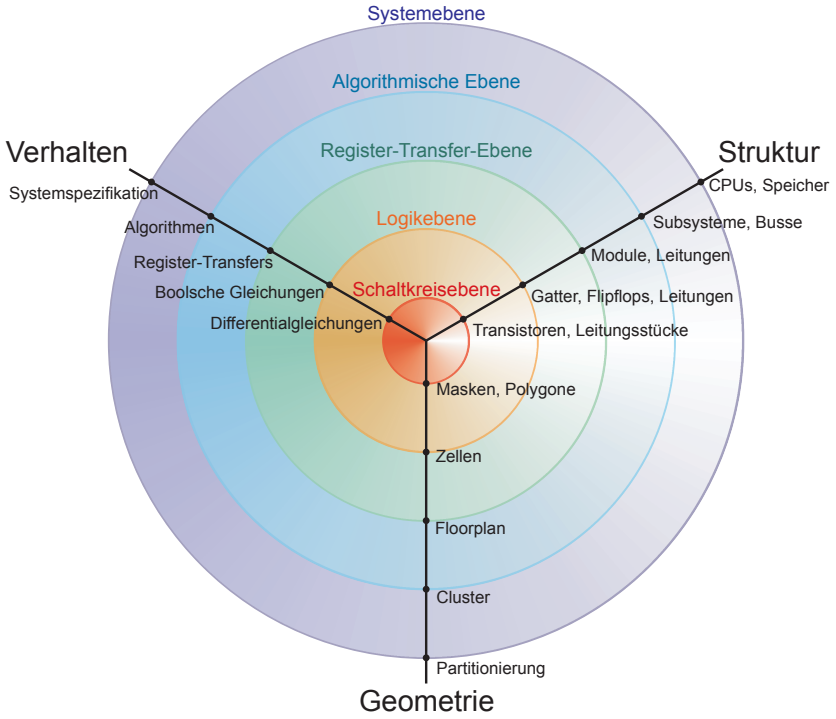


Abbildung 1.1: Y-Diagramm nach D. Gajski und R. Walker. Es existieren drei Sichten und fünf Ebenen zur Beschreibung digitaler Schaltungen [29].

wendet man hierfür die steigende Flanke. Flankengesteuerte FlipFlops werden häufig als *Register* bezeichnet und spielen die zentrale Rolle in synchronen Digitalsystemen.

Sehr häufig wird eine digitale Schaltung auf der Register-Transfer-Ebene beschrieben. Dies ist insbesondere auch beim Entwurf eines Systems für einen FPGA als Zieltechnologie häufig der Fall. In der Register-Transfer-Logik wird eine Schaltung dargestellt durch eine Reihe von Registern sowie kombinatorischer Logik, die sich jeweils zwischen den Registern befindet (siehe Abb. 1.2). Aus der Verwendung von Registern ergibt sich, dass die logischen Werte der Ausgangssignale sich nur bei einem Flankenwechsel des Taktsignals ändern können. Andererseits werden auch die Eingangssignale des Systems nur bei einem Flankenwechsel abgetastet (*gesampelt*). Die kombinatorische Logik zwischen den Registern besteht dabei nicht nur aus einzelnen Logikgattern[1], sondern kann auch einfache Module, wie z.B. Addierer, Komparatoren oder Multiplexer enthalten. Da sich alle Register synchron zu einem Taktsignal verhalten, muss die Periode dieses Taktes so gewählt werden, dass die Signale, die von der kombinatorischen Logik erzeugt werden, sich innerhalb einer Taktperiode stabilisieren und somit vom nächstem Register korrekt gesampelt werden können. Der Signalpfad mit der längsten Laufzeit begrenzt somit die maximal erreichbare Taktfrequenz. Dieser Pfad wird auch als *kritischer Pfad* bezeichnet. Durch ein synchrones Systemdesign muss nicht auf unterschiedliche Signallaufzeiten geachtet werden (solange diese nicht größer sind als eine Taktperiode) und die Gefahr eines falschen Systemverhaltens aufgrund von Glitches[2] kann vernachlässigt werden.

[1] ein Logikgatter dient der logischen Auswertung von Eingangssignalen in einer Schaltung; dabei werden einem oder mehreren Eingängen ein einziges logisches Ergebnis zugeordent (z.B. UND-Verknüpfung)

[2] als Glitch bezeichnet man ein kurzzeitig falsches Signal, das nicht der implementierten logischen Funktion entspricht und durch Laufzeitunterschiede auf dem Chip hervorgerufen wird

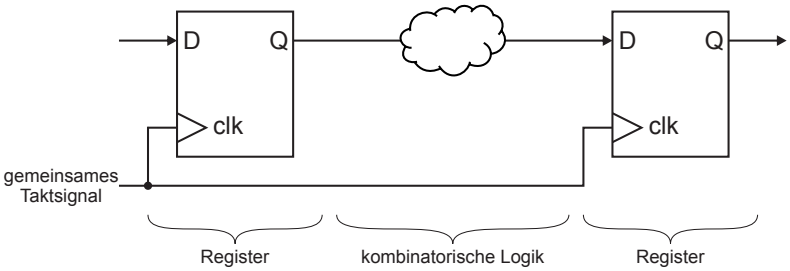


Abbildung 1.2: Synchrone Logik auf der Register-Transfer-Ebene.

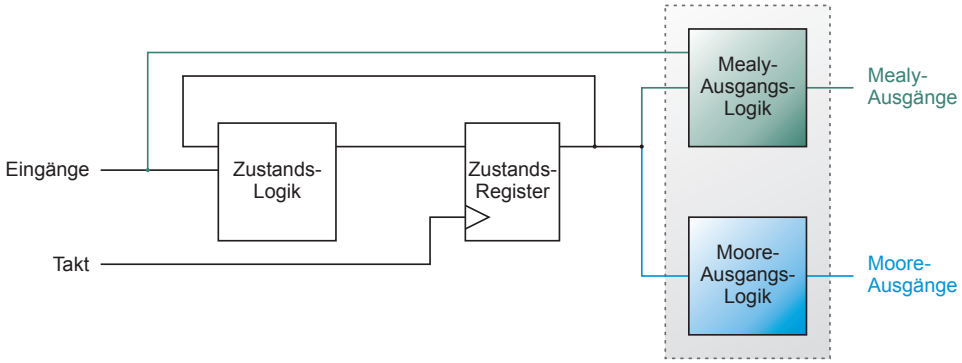


Abbildung 1.3: Schema eines endlichen Automaten mit Mealy- und Moore-Ausgängen.

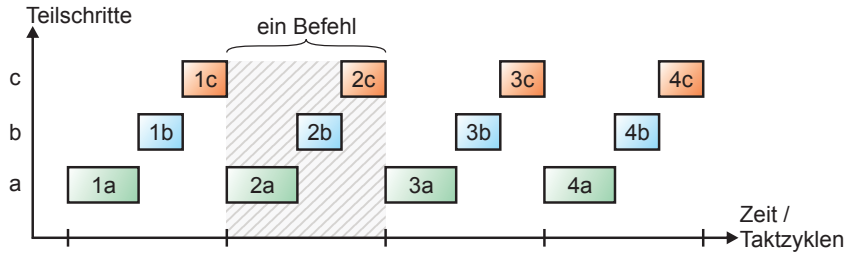
1.1.2 Endliche Automaten

Ein endlicher Automat (FSM[3]) stellt ein Verhaltensmodell eines Systems dar, welches aus diskreten Zuständen besteht, Übergänge zwischen diesen Zuständen beschreibt sowie Ausgangssignale festlegt [11]. Die Übergänge zwischen den Zuständen hängen im Allgemeinen von Eingangssignalen ab. Ein Automat wird als *endlich* bezeichnet wenn sein Verhalten durch eine endliche Anzahl an Zuständen beschrieben werden kann. Aus der Sicht des Hardware-Entwicklers besteht eine FSM aus drei Bestandteilen: der Zustandslogik (Zustandsübergangsfunktion), dem Zustandsregister und der Ausgangslogik (siehe Abb. 1.3) [4].

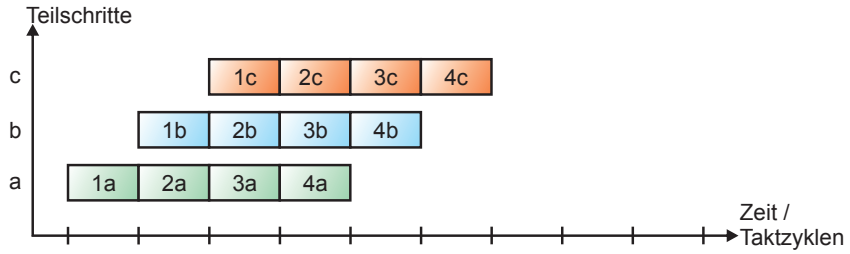
Aufgabe der Zustandslogik ist es, ausgehend vom aktuellen Zustand und den Eingangssignalen den neuen Zustand zu ermitteln. Dieser Zustand wird dann binär kodiert an den Dateneingang des Zustandsregisters gelegt, wo er bei der nächsten steigenden Taktflanke übernommen wird. Die Ausgangslogik legt die Werte für die Ausgangssignale fest. Bei der Ausgangslogik werden drei Typen unterschieden. Während beim *Moore*-Automaten die Ausgangssignale einzig und allein durch den aktuellen Zustand definiert sind, müssen beim *Mealy*-Automaten auch die jeweiligen Eingangssignale in Betracht gezogen werden. Eine Sonderform des Moore-Automaten stellt der *Medvedev*-Automat dar, bei dem die Ausgangssignale den binäre kodierten Zuständen entsprechen. Auf den Medvedev-Automaten soll im Folgenden nicht näher eingegangen werden.

Moore- und Mealy-Automaten sind prinzipiell ineinander überführbar. Der Vorteil des Mealy-Automaten liegt darin, dass er im Allgemeinen mit einer geringeren Anzahl an Zuständen auskommt. Lässt man jedoch zwei FSMs miteinander interagieren, so besteht bei Mealy-Typen die Gefahr einer unendlichen kombinatorischen

[3] Finite State Machine



(a) Sequentielles System



(b) Pipeline-System

Abbildung 1.4: Prinzip eines Pipeline-Systems. Die Befehle 1–4 können jeweils in drei Teilschritte (a–c) aufgeteilt werden. Jeder Teilschritt wird von einer speziellen, einmalig vorhandenen Hardware-Komponente durchgeführt. Das Prinzip des Pipelinings ist es, die einzelnen Teilschritte für aufeinanderfolgende Befehle überlappend auszuführen, d.h. Hardwarekomponente a beginnt mit der Bearbeitung des zweiten Befehls, bevor der erste Befehl vollständig abgearbeitet ist.

Schleife, was zu unvorhersehbarem Verhalten führen kann. Dies lässt sich vermeiden indem die Ausgangssignale der Mealy-FSM zusätzlich registriert werden. In der Praxis werden häufig Mischtypen verwendet, bei denen manche Ausgangssignal vom Mealy-Typ und andere vom Moore-Typ sind.

Um den Funktionsablauf einer FSM anschaulich darzustellen bedient man sich häufig eines Automatengraphen. In ihm werden die einzelnen Zustände als Zustandsblasen und die Übergänge zwischen den Zuständen als Pfeile dargestellt.

1.1.3 Pipelining

Dieser Abschnitt befasst sich mit der Technik des Pipelinings, die in heutigen Hardwaresystemen weit verbreitet ist und teilweise enorm zur Beschleunigung des Systems beiträgt.

Unter Pipelining versteht man die Unterteilung eines Befehls in mehrere Teilschritte (*Phasen*/*Stufen*), die in einer festen Reihenfolge durchgeführt werden sollen. Jeder dieser Teilschritte wird dabei von einer speziellen, einmalig vorhandenen Hardware-Komponente durchgeführt. Dies ist in Abbildung 1.4 im Vergleich zu einem System ohne Pipeline (sequentielles System) dargestellt. Der Vorteil eines Pipeline-Systems liegt in einer höheren maximalen Taktfrequenz gegenüber einem sequentiellen System. Dies lässt sich folgendermaßen verstehen: Beim sequentiellen System muss die Taktfrequenz so gewählt werden, dass *alle* Teilschritte innerhalb eines Taktzyklus ausgeführt werden können. Bei komplexeren Abläufen kann dies eine extreme Einschränkung darstellen. Bei einem System mit Pipeline wird jeder Teilschritt in

einem Taktzyklus ausgeführt; die maximal mögliche Frequenz richtet sich somit nach dem *langsamsten* Teilschritt [20]. Man beachte, dass bei dem in Abbildung 1.4 dargestellten Beispiel für vier aufeinanderfolgende Befehle ein Geschwindigkeitsvorteil erzielt wird, obwohl die Ausführung eines einzelnen Befehls länger dauert als ohne Pipeline. Der Vorteil kommt also erst dadurch zustande, dass ein und dieselbe Abfolge von Teilschritten mehrmals wiederholt wird. Pipelining erhöht lediglich den *Durchsatz*[4] des Systems, nicht jedoch die Ausführungszeit eines einzelnen Befehls (*Latenz*).

Um eine Pipeline in Hardware zu realisieren, müssen zwischen den einzelnen Stufen Register eingeführt werden. Diese sorgen dafür, dass sich die Eingänge zur jeweils nächsten Stufe synchron ändern und speichern die Zwischenergebnisse der vorherigen Stufe.

1.1.4 FPGAs

Ein FPGA (*Field Programmable Gate Array*) bezeichnet einen frei konfigurierbaren Chip, der vom Endanwender beliebig oft neu programmiert werden kann. Aufgrund ihrer Flexibilität und Leistungsfähigkeit finden FPGAs vor allem in der Forschung ihren Einsatz, aber auch in der industriellen Fertigung. Durch die hohen Fixkosten bei der Fertigung von ASICs sind FPGAs vor allem bei kleinen Stückzahlen und für Prototyp-Entwicklungen rentabel. Bei großen Stückzahlen hingegen ist der Einsatz von ASICs häufig günstiger, da die Stückkosten für FPGAs relativ hoch sind. In diesem Kapitel wird darauf eingegangen, wie ein solcher FPGA aufgebaut ist, wie man die gewünschten logischen Funktionen für derartige Chips implementiert (*Design Flow*) und wie ein Konfigurationsprozess vonstatten geht. Letzteres wird für das Verständnis von Kapitel 4 vorausgesetzt.

1.1.4.1 Aufbau und Funktionsprinzip

Der grundlegende Aufbau eines FPGAs ist in Abbildung 1.5 dargestellt. Die einzelnen *Logikblöcke* können beliebige binäre Funktionen mit einer bestimmten Anzahl an Eingangs- und Ausgangssignalen darstellen. Die innere Struktur eines solchen Blocks ist abhängig vom Hersteller des FPGAs sowie von dessen Typ. Die Verbindung der Logikblöcke mit den I/O[5]-Zellen, als auch ihre Vernetzung untereinander geschieht mit Hilfe von fest installierten Leitungen, die jedoch über *programmierbare Switch-Matrizen* (*PSM*) untereinander verbunden werden können. Sowohl die Logikblöcke als auch die PSMs werden bei der Konfiguration des FPGAs programmiert, so dass hierbei die Vernetzung im FPGA erfolgt. Zusätzlich zu diesen Ressourcen, die in jedem FPGA vorhanden sind, können manche Typen noch z.B. PowerPC[6] Prozessoren oder SRAM[7] enthalten. Darauf und auf den genauen Aufbau des im Rahmen dieser Arbeit verwendeten Chips wird in Abschnitt 2.1.4 eingegangen.

[4] Durchsatz bezeichnet die Anzahl an Befehlen, die in einer Zeiteinheit fertiggestellt werden können
[5] Input/Output
[6] Power: Performance Optimization With Enhanced RISC, PC: Performance Chip; PowerPC bezeichnet eine durch ein Konsortium aus Apple, IBM und Motorola spezifizierte CPU-Architektur
[7] Static Random Access Memory

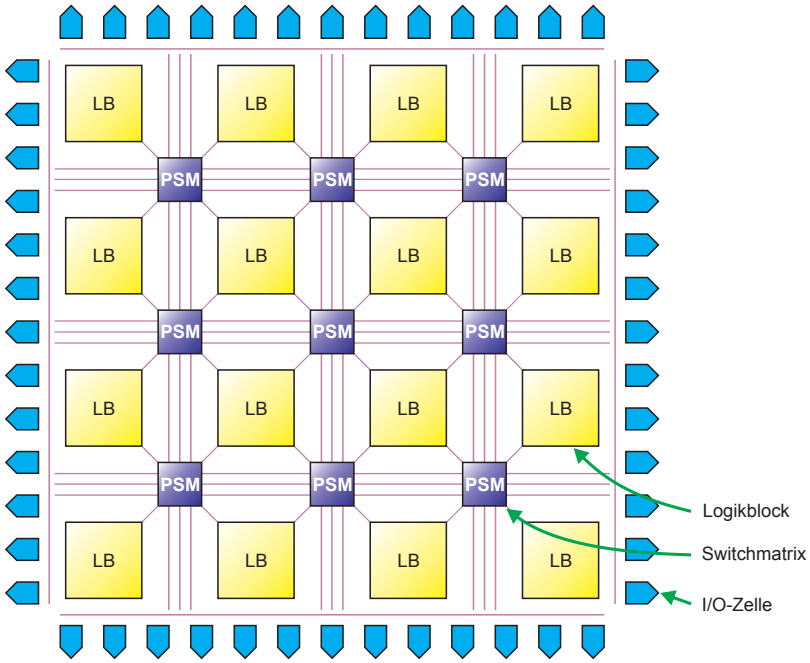


Abbildung 1.5: Prinzipieller Aufbau eines FPGAs aus Logikblöcken und Switch-Matrizen.

1.1.4.2 Design Flow

Dieser Abschnitt erklärt welche Schritte notwendig sind um die gewünschte logische Schaltung auf einem FPGA zu implementieren. Dieser sogenannte Design Flow ist in Abbildung 1.6 dargestellt.

Beschreibung des Designs Zunächst muss das gewünschte Design beschrieben werden. Dies kann mittels schematischer Schaltpläne erfolgen, wird jedoch in zunehmendem Maße in *Hardwarebeschreibungssprachen* (*HDL*[8]) realisiert. Diese erlauben im Gegensatz zu Software-Programmiersprachen die Beschreibung von zeitlichen Abfolgen und parallelen Abläufen, und sind somit auf die Bedürfnisse eines elektronischen Systems zugeschnitten. Die bekanntesten Vertreter solcher Hochsprachen sind VHDL[9] und Verilog.

Beim Entwurf mittels VHDL wird sowohl eine strukturelle Beschreibung unterstützt, die eine Komposition des Gesamtsystems aus mehreren Submodulen erlaubt, wie auch eine Verhaltensbeschreibung und in einem gewissen Rahmen auch die Beschreibung aus der Geometrie-Sicht. Die Verhaltensbeschreibung wiederum kann auf verschiedenen Ebenen (siehe Y-Diagramm auf S. 4) stattfinden. Auf der algorithmischen Ebene stehen abstrakte Konstrukte zur Verfügung, mit denen sich der algorithmische Ablauf des Systems beschreiben lässt; Designs auf dieser Ebene sind im Allgemeinen nicht synthetisierbar, können also nicht auf FPGAs oder ASICs abgebildet werden, sondern eignen sich insbesondere für den Aufbau von Testbenches (siehe unten). Die Logik-Ebene beschreibt ein System auf der Grundlage einzelner Gatter. Dies ist prinzipiell synthetisierbar, jedoch schon bei mittelgroßen Systemen viel zu aufwändig. Daher bedient man sich bei der Entwicklung eines synthetisierbaren Systems häufig der Register-Transfer-Ebene. Hier werden, wie bereits in Abschnitt 1.1.1 erwähnt, Register und Verbindungen sowie kombinatorische Logik zwischen ihnen

[8] Hardware Description Language
[9] Very High Speed Integrated Circuit HDL

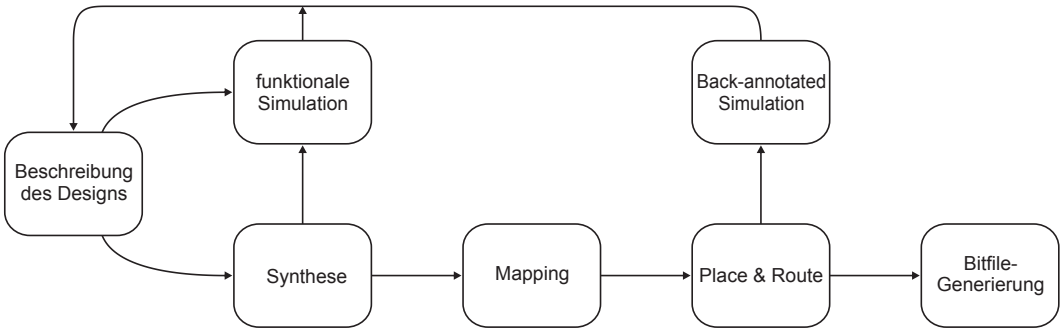


Abbildung 1.6: Design Flow beim Systementwurf für einen FPGA.

festgelegt. Von den FPGA-Herstellern werden auch parametrisierbare Module, wie z.B. FIFOs[10] oder RAM-Zellen bereitgestellt, die als Submodule in den Quellcode eingebettet werden können.

Funktionale Simulation Die funktionale Simulation erlaubt es, das Verhalten eines Systems zu Verifizieren und Implementierungsfehler zu erkennen. Dabei werden jedoch wesentliche zeitliche Parameter der Schaltung ignoriert, d.h. Register ändern ihren Ausgang exakt mit der Taktflanke, Signallaufzeiten sowie Anstiegs- und Abfallzeiten werden außer Acht gelassen. Um eine funktionale Simulation durchführen zu können, schreibt man eine *Testbench*, in die das zu simulierende Modul (*DUT*[11]) eingebettet wird. Die Testbench generiert dabei ggf. Takt und Reset-Signale sowie Stimuli, mit denen das DUT angeregt wird. Neben der Beschreibung in VHDL besteht auch die Möglichkeit Teile der Testbench in höheren Programmiersprachen darzustellen. Dazu bietet sich insbesondere SystemC an, welches eine Klassen-Bibliothek für C++ darstellt und dies mittels Makros und Funktionen um die notwendigen Mittel erweitert, um zusätzlich hardwaretypische Eigenschaften wie Synchronisation, Parallelität und Kommunikation zwischen Prozessen modellieren zu können [18]. Der Entwickler hat nun verschiedene Möglichkeiten mit Hilfe spezieller Simulations-Software die Reaktion des Systems auf die Stimuli zu untersuchen: Beispielsweise lässt sich der zeitliche Ablauf ausgewählter Signale in einem Signalverlaufs-Diagramm (*Waveform-Chart*) darstellen. Außerdem besteht die Möglichkeit, die Ausgangssignale des DUTs von der Testbench in Dateien schreiben zu lassen oder auch automatisiert auf ihre Richtigkeit zu überprüfen (*self-checking Testbench*). Diese Methode bietet sich vor allem bei komplexeren Systemen an.

Synthese Bei der Synthese untersucht die Synthese-Software den Code nach Standardkomponenten, wie Registern, Multiplexern, Komparatoren etc. Es wird auch erkannt, ob FSMs implementiert werden sollen, und es wird untersucht wie die Zustände dieser FSMs am besten binär repräsentiert werden können. Durch Syntheseoptionen lässt sich auch festlegen ob man eher eine Implementierung mit geringem Platzbedarf im Chip, oder eine möglichst hohe Taktfrequenz anstrebt. In diesem Schritt können auch vorgefertigte Cores in das System eingebunden werden. Das Ergebnis der Synthese ist eine Netzliste, die als Grundlage für die nachfolgenden Schritte dient.

[10] First In, First Out
[11] Device Under Test

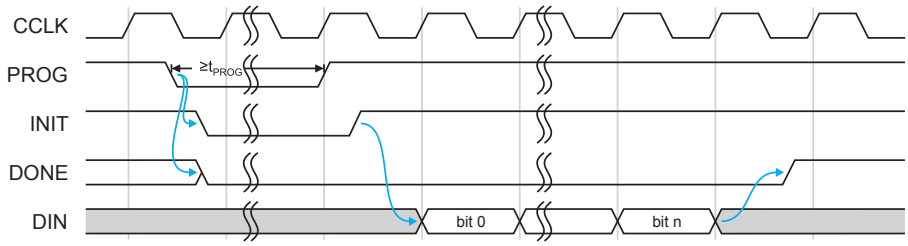


Abbildung 1.7: Konfigurationsablauf im Slave Serial Programming Mode (Virtex-II Pro).

Mapping und Place & Route Beim Mapping wird die Netzliste auf die FPGA-spezifischen Hardwareressourcen abgebildet. Dieser Vorgang ist somit abhängig vom verwendeten FPGA.

Das Place & Route (Platzieren und Verdrahten) bildet diese Ressourcen dann auf konkrete Bereiche im Chip ab und verbindet sie entsprechend miteinander. Die dafür verantwortliche Software muss dabei darauf achten, dass vom Entwickler vorgegebene Randbedingungen (*Constraints*) eingehalten werden. Dies betrifft häufig die Vorgabe einer gewünschten Taktfrequenz; die Software muss dann versuchen die Logik so zu platzieren, dass die Signallaufzeiten optimiert werden um diese Frequenz zu erreichen. Falls es der Software nicht gelingt die Constraints einzuhalten, muss der Entwickler das Design modifizieren. Dazu kann mit Hilfe einer Timing-Analyse herausgefunden werden welche Signalpfade das Constraint verletzen; diese Pfade können anschließend in der Hardwarebeschreibung modifiziert werden, z.B. durch das Einfügen einer zusätzlichen Pipelinestufe. Das Ergebnis des Platzierens und Verdrahtens ist eine vollständige Schaltungsbeschreibung (*Native Circuit Description*).

Back-annotated Simulation Bei der back-annotated Simulation wird das vollständig platzierte und verdrahtete FPGA-Design simuliert. Dabei werden nun auch Laufzeiten sowie das zeitliche Verhalten der einzelnen Komponenten des Systems berücksichtigt.

Bitfile-Generierung Nach erfolgtem Place & Route kann nun aus der Schaltungsbescheibung das *Bitfile* erzeugt werden. Dieses Bitfile wird bei der Konfiguration in den FPGA übertragen (siehe Abschn. 1.1.4.3) und enthält sämtliche Informationen bezüglich der verwendeten Hardwareressourcen, des Inhalts der Logikblöcke sowie deren Verschaltung durch die PSMs.

1.1.4.3 Konfigurationsablauf

In diesem Abschnitt soll der grundlegende Ablauf der Konfigurierung eines FPGAs erläutert werden. Das hier vorgestellte Schema entspricht jenem für den ihm Rahmen dieser Arbeit verwendetem Virtex-II Pro FPGA der Firma Xilinx [30]. Andere Hersteller verwenden unter Umständen geringfügig andere Schemata bzw. Signalbezeichnungen. Das Prinzip ist jedoch äquivalent.

Für den Virtex-II Pro stehen verschiedene Programmiermodi zur Verfügung, von denen hier der *Slave Serial Programming Mode* [34] beschrieben wird, da nur er im vorliegenden System verwendet wird. In diesem Modus wird der Takt nicht vom zu programmierenden FPGA selbst erzeugt, sondern von einer externen Quelle bereitgestellt (daher die Bezeichnung *Slave*).

Zur Steuerung bzw. Überwachung des Konfigurationsvorgangs existieren drei Steuersignale (`PROG`, `INIT`, `DONE`) sowie je eine Daten- bzw. Taktleitung. Zu Beginn

des Konfigurationsvorgangs sind `PROG` und `INIT` jeweils high; `DONE` ist high falls der FPGA schon programmiert ist, ansonsten low (siehe Abb. 1.7). Der Programmierzyklus beginnt, indem die `PROG`-Leitung von außen auf `'0'` gesetzt wird. Der FPGA wird daraufhin seine `INIT`-Leitung ebenfalls auf `'0'` setzen und damit beginnen eine evtl. bereits gespeicherte Konfiguration zu löschen. `PROG` muss mindestens 300 ns low gehalten werden. Danach wird es auf `'1'` gesetzt und der FPGA signalisiert die Bereitschaft Daten aufzunehmen indem er die `INIT`-Leitung auf `'1'` setzt. Nun können die Konfigurationsdaten (*Bitstrom*) als serieller Datenstrom synchron zum Konfigurationstakt (`CCLK`) zum FPGA gesendet werden. Während dieser Phase führt der FPGA intern CRC[12]-Überprüfungen durch um die Integrität der empfangenen Daten sicherzustellen. Sollte ein CRC-Fehler auftreten wird die `INIT`-Leitung vom FPGA auf `'0'` gesetzt. Falls bis zum Ende der Bitstrom-Übertragung keine Fehler auftreten, bestätigt der FPGA durch `DONE='1'` den korrekten Empfang der Konfigurationsdaten und seine Betriebsbereitschaft. Der Konfigurationsvorgang ist damit abgeschlossen.

1.2 Netzwerke

Computer-Netzwerke stellen heute ein alltägliches und unverzichtbares Mittel der Kommunikation und des Datenaustauschs dar. Dieser Abschnitt soll die wesentlichen Grundbegriffe von Netzwerken erläutern sowie ein sehr häufig verwendetes Netzwerkmodell darstellen.

1.2.1 Grundlegende Begriffe

Die Maschinen, welche dem Endanwender Zugriff auf das Netzwerk bieten werden als *Hosts* bezeichnet. Jeder Host ist mit einem *Router* verbunden, der die Weiterleitung der Daten von einem Sender zu einem Empfänger regelt. Die einzelnen Router sind durch das *Subnet* miteinander verbunden [27]. Dabei können unterschiedlichste Topologien realisiert werden, wie z.B. Ring-, Stern- oder beliebige unregelmäßige Topologien.

Netzwerke sind im Allgemeinen aus mehreren *Schichten* aufgebaut, wobei jede Schicht eine dedizierte Aufgabe zu erfüllen hat. Sie leistet der jeweils darüberliegenden Schicht *Dienste*, über die die Kommunikation zwischen den Schichten definiert ist. Die Kommunikation zwischen zwei Hosts ist in *Protokollen* geregelt. Abbildung 1.8 zeigt ein Beispiel für ein Schichtsystem; dort sind die Dienste als vertikale, die Protokolle als horizontale Pfeile dargestellt. Die Gesamtheit der Schichten und Protokolle eines Netzwerks wird als *Netzwerk-Architektur* bezeichnet.

Die Kommunikation zwischen äquivalenten Schichten zweier Hosts verläuft in der Realität zunächst vertikal nach unten, wobei nach und nach jede Schicht einen Kopf (*Header*) und evtl. einen Schwanz (*Trailer*) an die Nutzdaten anhängt, bis die Daten als Bitstrom über das physikalische Medium transportiert werden. Beim Empfänger wandern die Daten dann wieder vertikal nach oben, wobei Header und Trailer sukzessive wieder abgestreift werden. Die scheinbare Kommunikation erfolgt jedoch gemäß den definierten Protokollen horizontal, d.h. ein Prozess in Schicht n auf der Seite des Sendes verhält sich so, als ob er seine Daten direkt an Schicht n des Empfängers senden würde.

[12] Cyclic Redundancy Check

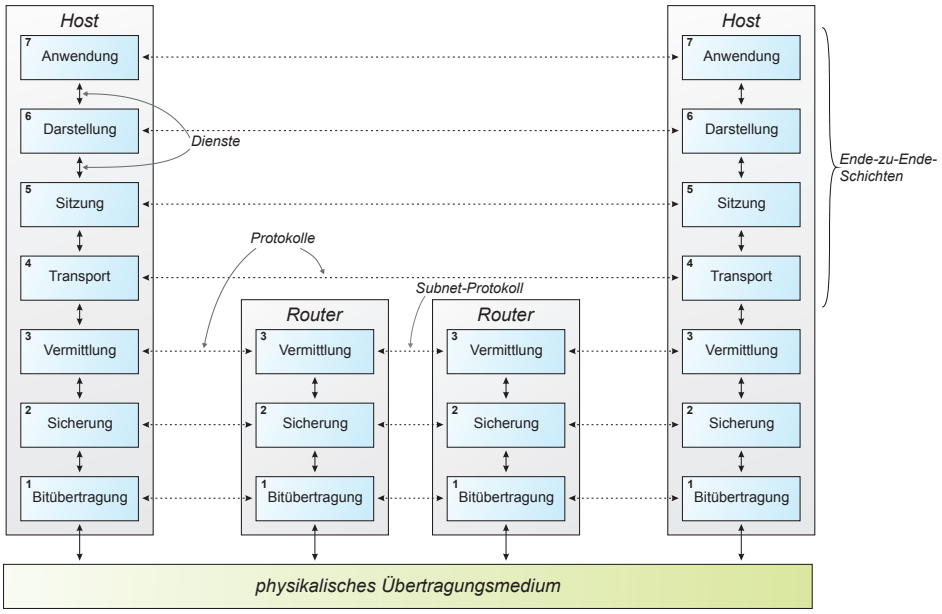


Abbildung 1.8: Schichten, Dienste und Protokolle im OSI-Referenzmodell.

Prinzipiell ist es die Aufgabe des Netzwerkentwicklers die Aufgaben und Dienste jeder einzelnen Schicht festzulegen. Jedoch hat sich die Unterteilung die von der International Organization for Standardization (ISO) vorgeschlagen wurde und als *OSI*[13]*-Referenzmodell* bekannt ist, bewährt.

1.2.2 OSI-Referenzmodell

Im OSI-Modell wird das Netzwerk in sieben Schichten aufgeteilt (siehe Abb. 1.8). Dabei beginnt eine neue Schicht immer dort, wo ein höherer Abstraktionsgrad benötigt wird. Das Abstraktionsniveau nimmt somit von Schicht 1 zu Schicht 7 stetig zu [27].

Schicht 1: Bitübertragung Die Bitübertragungsschicht (physical layer) befasst sich mit der Übertragung von Bits über das physikalische Medium. Sie stellt Hilfsmittel zur Verfügung um die Verbindung zum Medium herstellen und trennen zu können. Sie ist die einzige Schicht die direkt auf das Medium – dies kann beispielsweise ein Kupfer-Kabel oder eine Glasfaser sein – Zugriff hat. Die Bitübertragungsschicht muss auch spezifizieren, wie die binären Werte `'0'` und `'1'` kodiert werden; dies kann z.B. bedeuten, dass entsprechende Spannungspegel definiert werden. Die Hardware-Realisierung dieser Schicht wird häufig als PHY bezeichnet.

Schicht 2: Sicherung Die Aufgabe der Sicherungsschicht (data link layer) besteht darin eine möglichst fehlerarme Datenübertragung zu gewährleisten. Dazu werden die Rohdaten in Datenrahmen, sog. *Frames*, gepackt und evtl. mit Prüfsummen versehen. Die Sicherungsschicht befasst sich also mit strukturierten Daten, während die Bitübertragungsschicht nur einzelne Bits übertragt, ohne deren Bedeutung zu kennen.

[13] Open Systems Interconnection

Nach dem IEEE[14] Standard for Local and Metropolitan Area Networks [13] wird diese Schicht in zwei Subschichten unterteilt. Die MAC[15]-Subschicht regelt den Zugriff auf das physikalisch Medium, während die LLC[16]-Subschicht für die Datensicherung verantwortlich ist.

Schicht 3: Vermittlung Die Vermittlungsschicht (network layer) steuert den Betrieb des Subnets. Dazu gehört z.B. die Festlegung von Paketleitwegen zwischen Sender und Empfänger (*Routing*) sowie die Vermeidung von Datenstauungen im Subnet.

Schicht 4: Transport Zu den Aufgaben der Transportschicht zählen die Segmentierung von Datenpaketen, wie auch die Stauvermeidung. Sie bietet den darüberliegenden Schichten einen einheitlichen Zugriff, sodass diese die Eigenschaften des Kommunikationsnetzes nicht zu berücksichtigen brauchen [35].
Diese und alle höherliegenden Schichten sind nur an den Verbindungsendpunkten vorhanden. Daher findet die Kommunikation scheinbar nur zwischen Sender und Empfänger statt, und man spricht von *Ende-zu-Ende-Schichten.*

Schicht 5: Sitzung Diese Schicht befasst sich mit Problemen, die durch Verbindungsabbrüche entstehen. Dazu stellt sie Dienste für einen organisierten und synchronisierten Datenaustausch zur Verfügung.

Schicht 6: Darstellung Die Darstellungsschicht setzt die systemabhängige Darstellung der Daten in eine unabhängige Form um und ermöglicht somit den syntaktisch korrekten Datenaustausch zwischen unterschiedlichen Systemen. Sie bewältigt auch Aufgaben wie Datenkompression und Verschlüsselung.

Schicht 7: Anwendung Diese Schicht stellt den Endanwendungen Dienste zur Verfügung, die diese z.B. für den Dateitransfer nutzen können.

1.2.3 Sliding-Window-Algorithmus

Durch elektrische Störeinflüsse auf dem Weg vom Sender zum Empfänger ist es möglich, dass ein Frame beschädigt beim Empfänger ankommt. Um eine robuste, zuverlässige Kommunikation zwischen den Stationen eines Netzwerks zu erlauben ist es somit notwendig den Empfang der Frames bestätigen zu lassen (*Acknowledgement*). Die einfachste Art dies zu realisieren ist der *Stop-and-Wait*-Algorithmus, bei dem der Sender nach der Übermittlung eines Frames auf die Quittierung warten muss bevor er ein neues Frame aussendet. Dies ist sehr einfach zu implementieren, hat aber den Nachteil, dass der Sender relativ viel Zeit in Wartezuständen verbringt und somit die zur Verfügung stehende Bandbreite nicht optimal ausgenutzt werden kann. Wenn auf eine bessere Ausnutzung der Bandbreite Wert gelegt wird, so bietet sich die Verwendung von sogenannten *Sliding-Window-Algorithmen* an, deren Grundzüge hier dargelegt werden sollen.

Das Ziel derartiger Algorithmen ist es die Wartepause zu reduzieren und somit mehr Zeit für den effektiven Transfer von Datenrahmen aufzuwenden. Es existieren mehrere verschiedene Variationen des Sliding-Window-Prinzips, von denen hier nur auf die Grundform eingegangen werden soll. Allen Varianten gemeinsam ist die Markierung aller Frames mit einer *Sequenz-Nummer.* Hierbei handelt es sich um eine fortlaufende n Bit breite Zahl, die die Reihenfolge des Frames festlegt. Auf der Seite

[14] Institute of Electrical and Electronics Engineers
[15] Medium Access Control
[16] Logical Link Control

des Senders als auch des Empfängers stehen Puffer (*Fenster / Windows*) mit einer Tiefe von bis zu 2^{n-1} zur Verfügung um Frames abzulegen.

Der Ablauf der Transaktionen stellt sich nun wie folgt dar: Der Sender sendet nacheinander alle Frames, die sich in seinem Sendefenster befinden, ohne auf eine Quittung zu warten. Der Empfänger sendet für jedes Frame, welches er erhält eine Bestätigung. Sobald der Sender die Bestätigung für das erste Frame im aktuellen Fenster erhält verschiebt er sein Sendefenster um eine Position zu höheren Sequenznummern. Wird innerhalb eines definierten Zeitintervalls keine Quittung erhalten, so wird das entsprechende Frame erneut gesendet. Das Fenster auf der Empfängerseite wird benötigt, da aufgrund von Übertragungsfehlern neue Frames vor älteren ankommen können. Außerdem wird dadurch vermieden, dass im Falle einer Fehlübertragung bereits korrekt übermittelte Frames erneut gesendet werden müssen. Die Rahmen werden im jedem Fall in der korrekten Reihenfolge an die höhere Netzwerkschicht weitergereicht [27].

1.2.4 Überblick über den IEEE Standard 802

Der IEEE Standard 802 enthält verschiedene international etablierte Normen für die Bitübertragungs- und Sicherungsschicht lokaler Netzwerke. Zwei wichtige Vertreter hiervon sind *Ethernet* und *Token Ring*. In diesem Abschnitt soll auf die wesentlichen Festlegungen innerhalb dieser Normen eingegangen werden.

1.2.4.1 Ethernet (IEEE 802.3)

Der heute weit verbreitete Netzwerkstandard Ethernet ist in IEEE 802.3 [14] spezifiziert. Es existieren verschiedene Varianten dieser Norm, ja nach zugrundeliegender Topologie, Bandbreite und Art des physikalischen Mediums. Hier wird nur auf das grundlegende Funktionsprinzip eingegangen.

Bei Ethernet greifen alle angeschlossenen Stationen auf ein und dasselbe Übertragungsmedium zu. Die Arbitrierung des Mediums baut dabei auf dem *CSMA/CD*[17]-Algorithmus auf, der im Folgenden beschrieben wird. Falls eine Station Daten senden will prüft sie, ob das Medium gerade frei ist und, falls dies der Fall ist, beginnt mit den Versenden eines Frames. Ein Problem welches dabei auftreten kann ist, dass mehrere Stationen ein freies Medium detektieren und damit beginnen Frames zu versenden, so dass es zu Kollisionen auf dem Übertragungsmedium kommt. Die Norm legt für diesen Fall fest, dass alle sendenden Stationen ihre Übertragung abbrechen und eine zufällige Zeitspanne warten. Danach versuchen sie erneut zu senden. Sollte es erneut zu einer Kollision kommen müssen alle beteiligten Stationen erneut die Übertragung abbrechen und eine zufällige Zeit warten. Die maximale Wartepause nimmt dabei mit der Anzahl der Kollisionen exponentiell zu. Dadurch soll es dem Netzwerk ermöglicht werden, sich dynamisch an die Anzahl aller beteiligten Stationen anzupassen: bei wenigen Stationen ist häufig eine sehr kurze Pause ausreichend und wünschenswert, da man möglichst hohe Datenübertragungsraten erzielen will. Bei vielen Stationen würde jedoch eine zu kurze Wartezeit häufig zu wiederholten Kollisionen führen. Die im IEEE Standard festgelegte Zunahme der Wartezeiten garantiert einerseits kurze Verzögerungen bei wenigen Stationen und andererseits eine einigermaßen rasche Auflösung, falls viele Stationen beteiligt sind [27]. Eine wesentliche Folge dieser Art der Arbitrierung ist, dass nicht a priori feststeht, wann welche Station erfolgreich ein Paket versenden kann.

[17] Carrier Sense Multiple Access with Collision Detection

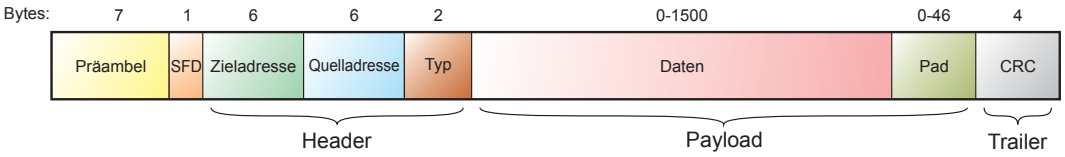


Abbildung 1.9: Basic MAC Ethernet-Frame mit Angabe der Länge der einzelnen Felder [14].

Das Format eines Basic MAC Ethernet-Frames ist in Abbildung 1.9 dargestellt. Daneben existiert noch das Tagged MAC Frame auf das hier nicht eingegangen wird. Das Basic MAC Frame beginnt mit einer *Präambel*, die eine alternierende Bitfolge mit einer Länge von sieben Bytes darstellt. Dadurch wird die Synchronisation auf den Bitabstand zwischen Sender und Empfänger ermöglicht. An die Präambel schließt sich der *start frame delimiter* (*SFD*) an, der den Beginn des eigentlichen Datenrahmens signalisiert.
Danach werden die jeweils sechs Byte langen *Adressen von Sende- und Empfangsstation* übermittelt. Dabei sieht die Norm auch Multicast-Adressen vor, d.h. ein Frame kann auch an eine definierte Gruppe von Adressaten verschickt werden.
Daran schließt sich das *Typ/Länge*-Feld an. Dieses Feld kann auf zwei Arten interpretiert werden, wobei durch das höherwertige Byte die jeweils gültige Interpretation festlegt. Ist es größer als `0x08`, so wird das Feld als Typ-Feld interpretiert und gibt Auskunft über das verwendete Protokoll der nächsthöheren Schicht innerhalb der Nutzdaten. Andernfalls gibt es die Länge der Nutzdaten im Datenrahmen an [15].
Anschließend folgen die eigentlichen *Nutzdaten* (*Payload*), die mit dem Frame übertragen werden sollen. Die Länge dieses Feldes darf 1500 Byte nicht überschreiten. Da die Spezifikation des Ethernet-Frames eine Mindestlänge von 64 Byte (vom SFD bis zum Ende des Rahmens) vorsieht, befindet sich nach den Nutzdaten noch ein *Pad*-Feld, dass den Rahmen auffüllt, falls nicht genug Daten vorhanden sind.
Das Frame wird beendet durch das *Prüfsummenfeld* (*CRC*), das eine Fehlererkennung beim Empfänger ermöglicht.

1.2.4.2 Token Ring (IEEE 802.5)

Für die Hardware, die in der Electronic Vision(s) Gruppe entwickelt wurde existiert ein Kommunikationsprotokoll (*Slow Control*, siehe Abschn. 2.1.5), dessen Funktionsprinzip an jenes der Token Ring-Norm angelehnt ist. Es soll daher hier auf das Prinzip von Token Ring, insbesondere auf die Arbitrierung, also die Zugriffssteuerung auf das physikalische Medium, eingegangen werden. In Abschnitt 2.1.5 wird dann auf die Slow Control im Speziellen eingegangen.

In einer Token Ring-Topologie sind alle Stationen in einen Ring implementiert, in dem die Frames unidirektional kreisen (siehe Abb. 1.10). Jede Station kann die Daten an ihrem Eingang aufnehmen, weiterreichen oder aktiv ein Frame versenden. Dabei ist zunächst keine Station ausgezeichnet. Durch das Protokoll muss sichergestellt werden, dass zwei Frames nicht miteinander kollidieren können und sich somit gegenseitig zerstören. Dies wird folgendermaßen realisiert: Falls alle Stationen untätig sind, kreist im Ring ein spezielles Frame, das *Token*. Immer wenn eine Station senden möchte, muss sie warten bis das Token bei ihr ankommt. Sie entfernt es dann vom Ring und sendet stattdessen ein Frame aus [12]. Da zu jedem Zeitpunkt immer nur maximal ein Token im Ring kreist kann somit nur jeweils eine Station senden. Die Stationen die ein Frame erhalten, müssen nun prüfen, ob das Frame an sie adres-

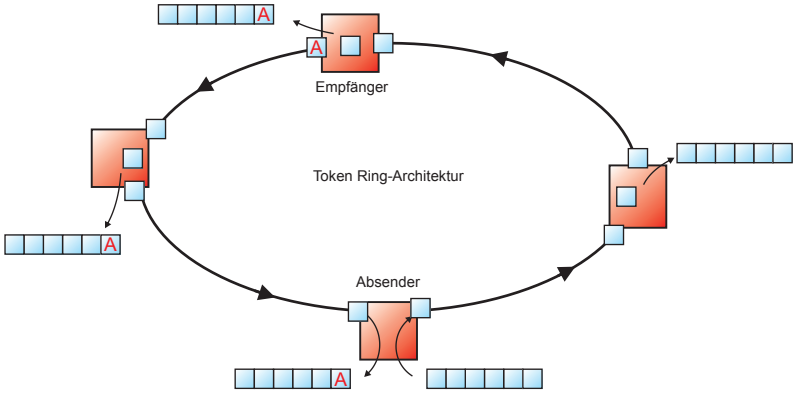


Abbildung 1.10: Schematische Darstellung eines Token Ring-Netzwerks. Der Absender hat ein Token aufgenommen und es durch ein gültiges Datenframe ersetzt. Die nachfolgenden Stationen reichen das Frame weiter, bis es am Empfänger ankommt, von diesem bestätigt wird und schließlich wieder beim Absender ankommt [21].

siert ist. Falls dies so ist, werden die Nutzdaten in einen internen Speicher kopiert und ein Bestätigungsbit im weitergereichten Frame gesetzt. Andernfalls wird das Frame unverändert weitergereicht. Nachdem die Sendestation das bestätigte Frame erhalten hat, sendet sie ein neues Token aus. Durch diese Art der Arbitrierung wird sichergestellt, dass das Schreibrecht von einer Station zur nächsten wandert, auch wenn alle Stationen senden wollen. Somit kann garantiert werden, dass keine Station über einen längeren Zeitraum von der Kommunikation ausgeschlossen wird und es kann eine maximale Wartezeit für die Zugriffsberechtigung berechnet werden, im Gegensatz zu Ethernet [26].

Jedes Token Ring-basierte Netzwerk muss über eine Überwachungsstation (*Monitor*) verfügen. Die Aufgaben des Monitors umfassen das Ersetzen verlorener Tokens und die Entfernung von Rahmenbruchstücken wie auch von 'verwaisten' Frames vom Ring. Derartige verwaiste Frames durchlaufen den Ring wiederholt, was z.B. passieren kann wenn der Rahmen an einen Adressaten gerichtet ist, der nicht existiert und der Sender nicht in der Lage ist das Frame vom Ring zu entfernen. Prinzipiell kann jede Station der Monitor sein. Es ist in der Tat auch so, dass im Falle eines Ausfalls des Monitors eine andere Station dessen Aufgaben übernimmt.

2 Beschreibung des vorhandenen Systems

In diesem Kapitel wird das System dargestellt, das in der Electronic Vision(s) Gruppe zur Modellierung großskaliger neuronaler Netze eingesetzt wird. Dabei wird zunächst auf die Hardware und die verwendeten Kommunikationsprotokolle eingegangen. Danach wird die Problemstellung thematisiert mit der sich die vorliegende Arbeit befasst. Der letzte Abschnitt des Kapitels behandelt den Entwicklungsstand der programmierbaren Logik zu Beginn der Arbeit.

2.1 Überblick über die vorhandene Hardware

In Abbildung 2.1 ist das Gesamtsystem dargestellt. Die ANN[1]-ASICs, welche die Funktion biologischer Neuronen und Synapsen nachbilden, werden auf der Trägerplatine *Nathan* montiert. Bis zu 16 Nathan-Module können auf einer *Backplane* platziert und parallel betrieben werden. Die Steuerung des Systems ist über eine SCSI[2]-Verbindung zu der PCI[3]-Karte *Darkwing* möglich. Des Weiteren besteht die Möglichkeit die Backplane über eine Gigabit-Ethernet-Schnittstelle mir einem PC zu verbinden.

2.1.1 Darkwing PCI-Karte

Die Verbindung zwischen der Backplane und einem PC kann vermittels der PCI-Karte Darkwing hergestellt werden. Als Übertragungsmedium dient dabei ein SCSI-Kabel, über welches sowohl Daten, als auch Steuersignale und ein Taktsignal übertragen werden. Auf der Darkwing-Platine befindet sich ein PLX-Chip, der die Anbindung an das PC-interne PCI-Interface regelt, ein FPGA der Firma Xilinx, der die Kommunikation mit der SCSI-Schnittstelle steuert sowie ein RAM-Baustein [2]. Die Darkwing-Karte steuert auch die Konfiguration der FPGAs auf den Nathan-Modulen (siehe Abschn. 2.1.3). Er nimmt bei der Steuerung und Kontrolle des Slow Control-Netzwerks eine ausgezeichnete Rolle ein, da er als Slow Control-Master konfiguriert ist (siehe Abschn. 2.1.5).

Mit dem Einsatz der Gigabit-Ethernet-Anbindung soll die Darkwing-Karte außer Dienst gestellt werden, da die erreichbaren Datenraten für die Versorgung des neuronalen Netzes mit Stimuli-Daten und die Auswertung der Reaktion des Systems nicht ausreichend sind (siehe Abschn. 2.2.3).

2.1.2 Backplane

Die Backplane stellt die Trägerplattform für die Nathan-Module dar. Bis zu 16 Nathan-Platinen können auf impedanzkontrollierten Steckplätzen platziert werden.

[1] Artificial Neural Network
[2] Small Computer System Interface
[3] Peripheral Component Interconnect

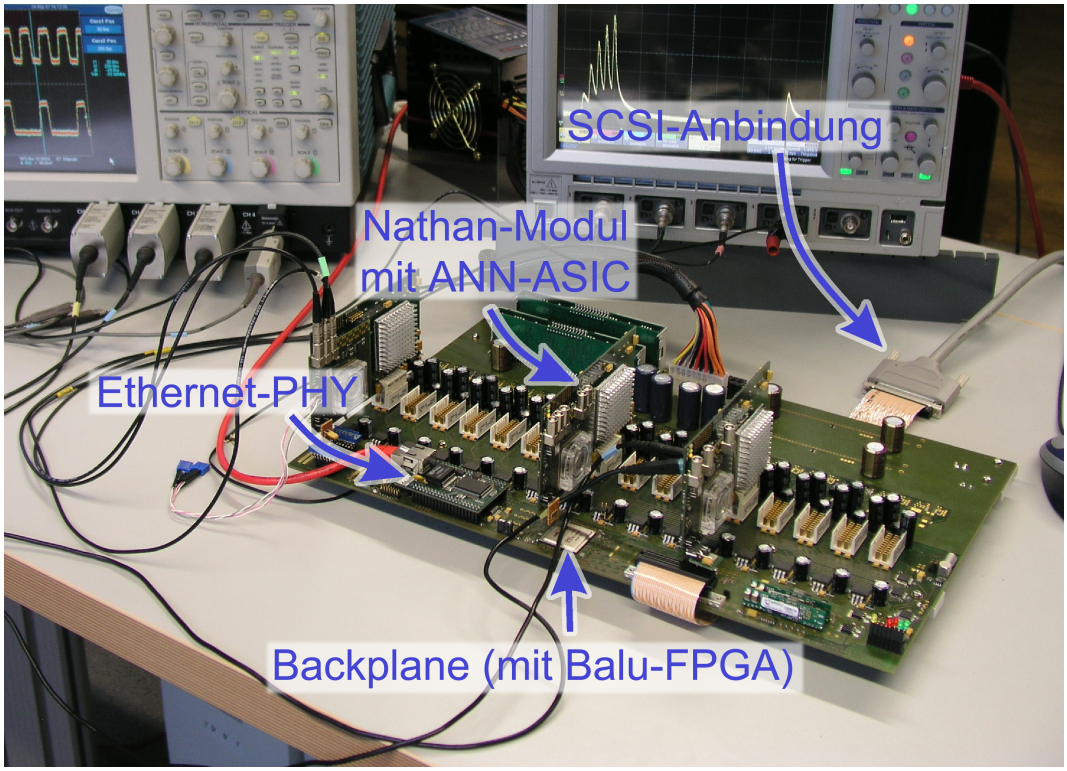
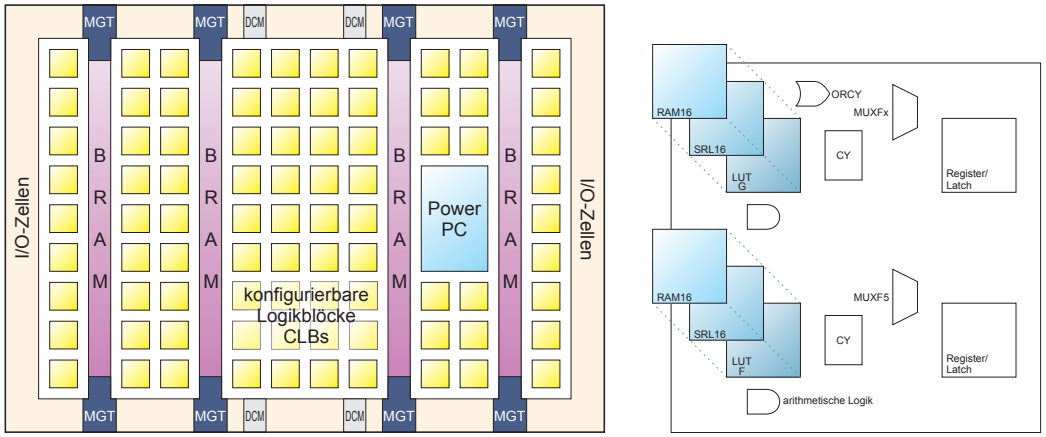


Abbildung 2.1: Gesamtsystem zum Betrieb künstlicher neuronaler Netze. Auf dem Bild zu sehen ist die Backplane mit dem darauf befindlichen FPGA *Balu*. Die Backplane ist hier mit drei Nathan-Modulen bestückt und verfügt über Außenanbindungen über ein SCSI-Kabel sowie eine Ethernet-Schnittstelle.

Die Aufgaben der Backplane umfassen dabei die physikalische Verdrahtung der Steckplätze sowie die Bereitstellung der Versorgungsspannung für die Nathan-Module. Außerdem befindet sich auf der Backplane ein Quarz-Oszillator, der einen Takt von 156,25 MHz, zusätzlich zu dem von der Darkwing-Karte gelieferten Taktsignal bereitstellt. Die einzelnen Steckplätze sind einerseits untereinander in Form eines 2-dimensionalen Torus verbunden [8]. Hierzu dienen serielle Hochgeschwindigkeitsverbindungen der FPGAs auf den Nathan-Platinen. Andererseits bestehen auch Punkt-zu-Punkt-Verbindung mit einem auf der Backplane befindlichen Virtex-II Pro FPGA namens *Balu*[4], die zur Anbindung der Nathan-Module an das Slow Control-Netzwerk benutzt werden (siehe Abschn. 2.1.5). Durch diese Punkt-zu-Punkt-Verbindungen lässt sich die gewünschte Topologie im FPGA-Design beschreiben und den jeweiligen Bedürfnisse anpassen. Es existiert jedoch auch eine ältere Version der Backplane, die über keinen FPGA verfügt und bei der die Steckplätze fest zu einer Ringtopologie vernetzt sind [8]. Die Anbindung der Backplane an die Außenwelt erfolgt zum einen über einen SCSI-Sockel mit der Darkwing-Karte, zum anderen befindet sich ein 80-poliger Santa-Cruz-Sockel auf der Backplane, auf den beliebige Module aufgesteckt werden können. Insbesondere eignet sich dieser Sockel für den Einsatz eines Gigabit-Ethernet-PHYs, womit sich eine Ethernet-Anbindung der Backplane an jeden handelsüblichen PC realisieren lässt [10].

[4] Backplane Control Logic Unit



(a) Schema des verwendeten FPGAs (b) Aufbau eines FPGA-Slices

Abbildung 2.2: Schematischer Aufbau eines Virtex-II Pro FPGAs [30].

2.1.3 Nathan-Modul

Auf jedem Nathan-Modul befindet sich ein Virtex-II Pro FPGA, der über acht serielle Multi-Gigabit-Transceiver verfügt [30]. Vier von diesen Transceivern werden benötigt um über auf der Backplane fest verdrahtete Leiterbahnen den 2-dimensionalen Torus zu realisieren; die restlichen vier sind mit Buchsen an der Oberseite des Nathan-Moduls verbunden und erlauben dadurch eine freie, kabelgebundene Vernetzung der Module untereinander. Zusätzlich befindet sich auf den Modulen jeweils ein DDR-SDRAM[5]-Sockel, der Speichermodule mit einer Größe bis zu 2 GB aufnehmen kann sowie zwei SRAM-Chips mit einer Gesamtkapazität von 512 kB. Diese Speicherressourcen werden benötigt um Daten für Experimente zu speichern. Dies sind zum einen die Eingabe-Stimuli (spike trains), zum anderen die Reaktion des ANN-Chips auf diese Stimuli. Schließlich stellt jede Nathan-Platine noch einen Sockel zur Verfügung, auf dem die entwickelten neuronalen ASICs (Spikey bzw. HAGEN[6]) platziert werden können.

2.1.4 Virtex-II Pro FPGA

In Abbildung 2.2(a) ist der prinzipielle Aufbau des Virtex-II Pro FPGAs schematisch dargestellt. Dieser FPGA kommt in der Ausführung XC2VP7 sowohl auf den Nathan-Platinen als auch auf der Backplane (Balu-FPGA) zum Einsatz. Tabelle 2.1 bietet einen Überblick über die Hardwareressourcen dieses FPGAs.

Die Logikblöcke, die die Grundlage für die kombinatorische und synchrone Logik legen, werden bei XILINX-FPGAs als CLBs[7] bezeichnet. Jeder dieser CLBs besteht aus vier Slices, die jeweils mit programmierbaren Switch-Matrizen (PSMs) sowie innerhalb eines CLBs auch untereinander über schnelle Datenpfade verbunden sind. Der Aufbau eines Slices ist in Abbildung 2.2(b) dargestellt. Die Grundlage für die Funktionsweise des FPGAs bilden Funktionsgeneratoren mit vier Eingängen und einem Ausgang. Bei diesen handelt es sich um SRAM-Zellen, die je nach Implementierung verschiedenen Zwecken dienen können: zum einen können sie als Look-Up Table (LUT) benutzt werden, wobei sie jede beliebige boolsche (logische) Funktion

[5] Double Data Rate Synchronous Dynamic RAM
[6] Heidelberg Analog Evolvable Neural Network
[7] Configurable Logic Block

Logikressourcen	
CLBs (1 CLB $\widehat{=}$ 4 Slices)	1.232
Slices	4.928
Logikzellen[9]	11.088
Speicher	
BlockRAM	$44 \cdot 18$ kBit $= 792$ kBit
max. Distributed RAM	154 kBit
sonstige Ressourcen	
PowerPC Prozessoren	1
RocketIO Transceiver	8
DCMs	4
I/O Pads	396

Tabelle 2.1: Übersicht über die Ressourcen des Virtex-II Pro FPGAs (Typ: XC2VP7) [30].

mit vier Eingängen implementieren können. Dies stellt ihre häufigste und wichtigste Verwendung in einem FPGA-Design dar. Intern werden dabei die Eingangssignale als Adress-Vektor interpretiert und der 1 Bit breite Inhalt der entsprechenden RAM-Zeile an den Ausgang gelegt. Aus diesem Schema folgt, dass die Verzögerung zur Auswertung jeder beliebigen boolsche Funktion – sofern sie in einer einzigen LUT untergebracht werden kann – identisch ist. Dies stellt einen wesentlichen Unterschied zu ASICs dar, bei denen im Allgemeinen jeder Gattertyp unterschiedliche Verzögerungen hervorruft. Eine andere Verwendung der Funktionsgeneratoren ist die Einbindung in das System als 1 Bit breites Schieberegister mit einer im laufenden Betrieb veränderbaren Tiefe bis zu 16 Bit (SRL16). Der Lesezugriff kann dabei wahlweise asynchron oder synchron erfolgen, während der Schreibzugriff immer synchron ist. Die Realisierung der Funktionsgeneratoren als SRAM ermöglicht auch eine direkte Nutzung als RAM. Jeder Funktionsgenerator kann einen 16 Worte[8] tiefen, 1 Bit breiten RAM-Block darstellen. Dies wird von Xilinx als Distributed SelectRAM+ bezeichnet [30].

Neben den Funktionsgeneratoren befindet sich in jedem Slice auch noch arithmetische Logik, welche die Realisierung eines 2-bit Volladdierers in einem Slice sowie die Implementierung effizienter Multiplizierer ermöglicht. Am Datenausgang des Slices befinden sich Speicherzellen, die als Register oder als Latch benutzt werden können. Die Multiplexer dienen unter Anderem der Verschaltung mehrerer LUTs bzw. Slices und ermöglichen so logische Funktionen mit mehr als vier Eingangssignalen.

Zusätzlich zu der Möglichkeit der Verwendung des Distributed RAM bietet der Virtex-II Pro FPGA noch Block SelectRAM (BRAM) Speicher. Dieser Speicher ist auf dem FPGA in 44 Blöcken zu je 18 kBit angeordnet, so dass er eine Gesamtspeicherkapazität von 792 kBit zur Verfügung stellt. Diese Speicherblöcke können mit Hilfe des Xilinx LogiCORE Core-Generators zu verschieden dimensionierten Speicherarealen verschaltet werden. Sie stehen auch für die Verwendung als FIFOs zur Verfügung.

[8] als (Daten-)Wort oder word wird der Inhalt einer Adresse in einem Speicher bezeichnet
[9] Logikzelle: (1) 4-input LUT + (1) FlipFlop + Übertragslogik

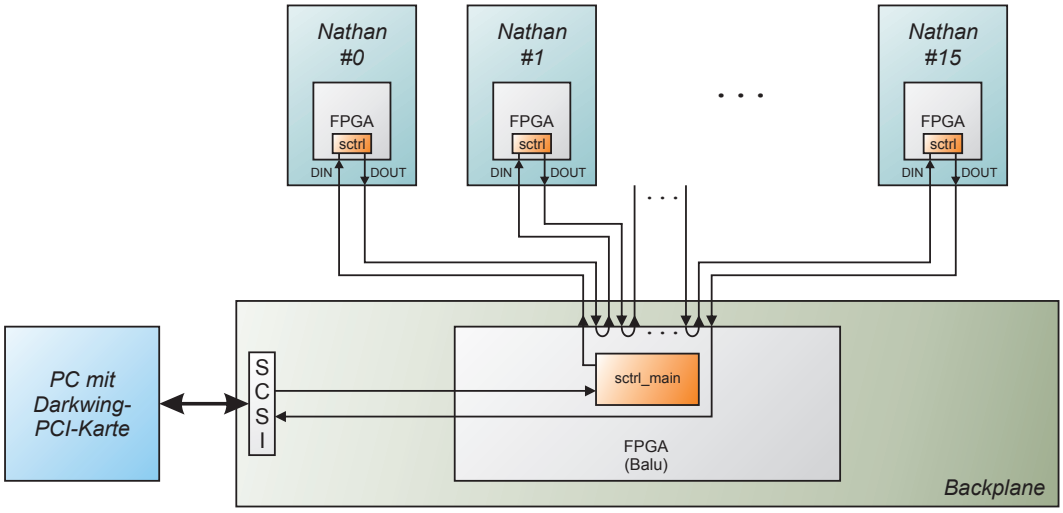


Abbildung 2.3: Schematische Darstellung des Slow Control-Rings mit 17 Stationen. Der Datenausgang eines Nathan-Moduls wird vom Balu-FPGA direkt auf den Dateneingang des nächsten geschaltet. In der Skizze nicht gezeigt sind die Takt- und Reset-Leitungen, die sternförmig vom Balu-FPGA zu jedem Nathan-Sockel verlaufen.

Des Weiteren befinden sich auf dem verwendeten FPGA acht DCMs[10], die aus einem Quelltakt mehrere Taktsignale zur Verwendung innerhalb des FPGA-Designs zur Verfügung stellen. Der Quelltakt kann von den DCMs in Taktfrequenz und Phasenlage modifiziert werden.

Außerdem wird noch ein eingebetteter PowerPC bereitgestellt, auf den z.B. ein Linux-System portiert und entsprechende Software implementiert werden kann [25].

Die Kommunikation des FPGAs mit externen Hardware-Komponenten findet über die I/O-Zellen statt. Diese unterstützen eine Vielzahl gängiger Signalstandards und können somit an die jeweiligen Erfordernisse angepasst werden. Zudem lassen sich über die RocketIO Multi-Gigabit-Transceiver (MGTs) serielle Hochgeschwindigkeits-Verbindungen mit externen Hardware-Komponenten herstellen. Die MGTs der FPGAs auf den Nathan-Platinen werden benutzt um den 2-dimensionalen Torus zu realisieren.

2.1.5 Slow Control

Mit Hilfe der vorgestellten Hardware ist es nun prinzipiell möglich bis zu 16 Nathan-Module und darauf montierte neuronale Netzwerk-ASICs parallel zu betreiben. Um Experimente auf dieser Hardware durchführen zu können, muss jedoch auch noch eine Möglichkeit bestehen, mit der Hardware zu kommunizieren. Es müssen Experimentdaten zu den Nathan-Modulen transferiert werden und die Antwort des neuronalen Netzes darauf muss ausgelesen werden. Außerdem müssen unter Umständen bestimmte Systemparameter im laufenden Betrieb verändert werden. Um dies zu bewerkstelligen wurde das Slow Control-Netzwerk entwickelt [21], über das die FPGAs auf der Darkwing-Karte, der Backplane und allen Nathan-Platinen miteinander verbunden sind. Die physikalische Vernetzung erfolgt dabei durch eine 1 Bit breite Datenleitung, über die Frames theoretisch mit Frequenzen bis zu 100 MHz übertragen werden können. Die in der Praxis üblicherweise stabilen Frequenzen lie-

[10] Digital Clock Manager

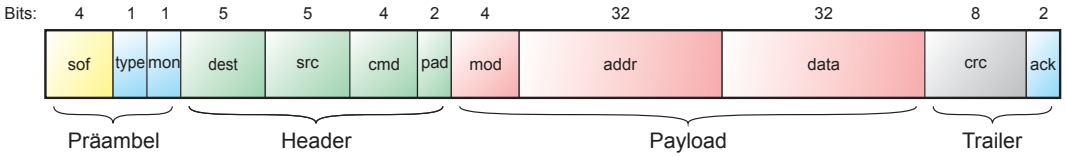


Abbildung 2.4: Frameformat des Slow Control-Netzwerks. Für ein Token ist das `typ`-Bit `'1'` und das Frame ist danach zu Ende. Bei allen anderen gültigen Frames ist das `typ`-Bit `'0'`. Die Felder `addr`/`data` sowie `mod` sind nicht bei allen Datenframes vorhanden.

gen jedoch eher im Bereich von etwa 50–60 MHz. Die Topologie entspricht einem Ring. Der Grund hierfür ist im Wesentlichen, dass auf der alten Version der Backplane alle Nathan-Steckplätze fest in einer Ringstruktur verdrahtet waren (siehe Abschn. 2.1.2); die aktuelle Version der Backplane bietet zwar durch die bestehenden Punkt-zu-Punkt-Verbindungen zwischen dem Balu-FPGA und jedem einzelnen Nathan-Sockel die Möglichkeit andere Topologien zu implementieren, jedoch wurde aus Kompatiblitätsgründen die Ringstruktur beibehalten. Die ringförmige Vernetzung ist schematisch in Abbildung 2.3 dargestellt.

Konzeptionell ist die Slow Control an den Token Ring-Standard (siehe Abschn. 1.2.4.2) angelehnt, auch wenn es signifikante Unterschiede gibt. Die Arbitrierung erfolgt analog zu Token Ring über ein Token, das im Ring kreist und von jeder Station mit drei Taktzyklen Verspätung weitergereicht wird, falls die Station keine Daten senden will. Möchte eine Station ein Frame versenden, so muss sie warten bis ein Token bei ihr ankommt. Sie kann dieses Token dann von Ring nehmen und durch einen gültigen Datenrahmen ersetzen.

Das Frameformat der Slow Control ist in Abbildung 2.4 dargestellt. Jedes Frame beginnt mit der Präambel (start-of-frame, `sof`), gefolgt von dem `typ`-Bit. Dieses legt fest, ob es sich um ein Token (`typ='1'`) oder um ein Datenframe (`typ='0'`) handelt. Ein Token-Frame ist nach dem `typ`-Feld zu Ende. Bei Datenframes schließt sich das `mon`-Feld an, das zur Erkennung von 'verwaisten' Rahmen dient (siehe S. 23).
Der Header des Frames enthält die Adressen von Empfänger (`dest`) und Sender (`src`) sowie ein `cmd`-Feld, das die Information enthält um welche Art von Slow Control-Befehl es sich handelt. Dies können z.B. Schreib- oder Lesebefehle sein, oder auch Befehle die der Initialisierung einer Station und der Festlegung einer Stationsnummer dienen. Das sich anschließende `pad`-Feld wurde implementiert damit die Länge des Headers ein Vielfaches von vier Bit ist. Dies ist nötig, da die Prüfsumme des Frames blockweise über 4-Bit-Segmente[11] berechnet wird. Da in jeder Station bis zu 16 unterschiedliche Slow Control-Module implementiert sein können, von denen jedes Modul eine dedizierte Aufgabe hat, befindet sich im Payload eine Modulnummer (`mod`), gefolgt von einer Adresse und den eigentlichen Daten (`addr` und `data`). Es existieren auch Datenrahmen, die keine `addr`- und `data`-Felder besitzen; einige besitzen zudem kein `mod`-Feld. Somit existieren neben dem Token drei Frametypen, deren Unterscheidung anhand des `cmd`-Feldes möglich ist.
Den Trailer bilden schließlich die Prüfsumme (`crc`) und zwei `ack`-Bits. Als Prüfsumme dient ein CRC, der über den Header und den Payload berechnet wird. Die beiden `ack`-Bits dienen der Empfangsbestätigung von Frames. Sie müssen sich nach der Prüfsumme befinden, da die Empfängerstation das Frame erst bestätigen kann *nachdem* sie den CRC empfangen und überprüft hat. Um einer Zerstörung des Be-

[11] häufig als *Nibble* bezeichnet

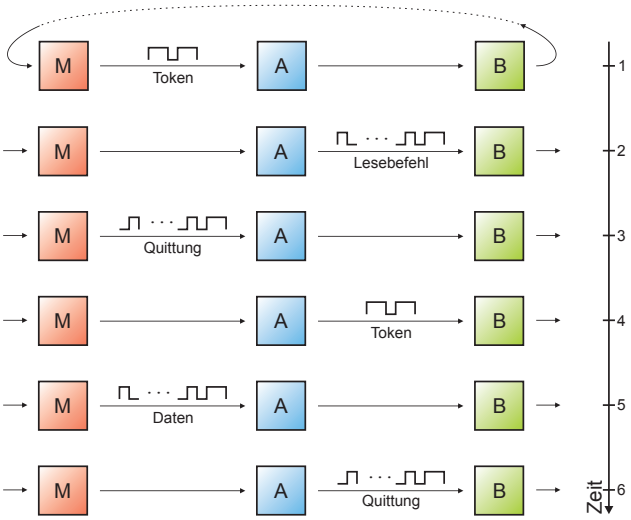


Abbildung 2.5: Schematischer Ablauf eines Slow Control-Lesebefehls. Station M ist der Master, A will Daten von B lesen. Zwischen den einzelnen Stationen können sich beliebig viele andere Stationen befinden.

stätigungsbits, z.B. durch elektrische Störeinflüsse, entgegenzuwirken wurde es redundant implementiert; das erste der beiden Bits ist das eigentliche Bestätigungsbit, das zweite sein Inverses.

Ein wesentlicher Unterschied zur Token Ring-Spezifikation ist, dass es in Slow Control-Netzwerk eine festgelegte Überwachungsstation gibt, die als *Master* bezeichnet wird. Aufgabe dieses Masters ist es die Funktionalität des Ringes zu überwachen und im Falle einer Fehlfunktion einzuschreiten. Der Master muss sicherstellen, dass zu jedem Zeitpunkt nur ein Token im Ring kreist, und dass dieses Token nicht verlorengeht. Im Falle eines verlorengegangenen Tokens muss der Master ein neues Token aussenden. Außerdem muss er verwaiste Frames vom Ring entfernen, also Frames die z.B. ein ungültiges `dest`-Feld haben und deshalb ohne eine ringinterne Überwachungsinstanz unendlich lange im Ring kreisen würden. Dadurch wäre keine Station mehr in der Lage Frames zu senden oder zu empfangen. Verwaiste Frames werden anhand des `mon`-Bits erkannt; dieses Bit ist in jedem Frame zunächst auf `'0'` gesetzt, wird jedoch invertiert, sobald das Frame den Master passiert. Da aufgrund der Ringstruktur jedes Frame im Normalfall nur maximal einmal den Master passiert, muss dieser alle Frames vom Ring entfernen, bei denen das `mon`-Feld schon `'1'` ist, wenn sie am Master ankommen. Prinzipiell kann jede Station die Rolle des Masters einnehmen. Jedoch wird im Betrieb immer der FPGA auf der Darkwing-Platine als Master konfiguriert.

Der genaue Ablauf einer Slow Control-Transaktion wird im Folgenden dargestellt. Für dieses Beispiel wird ein Lesebefehl betrachtet, der von Station A an Station B gerichtet ist (siehe Abb. 2.5). Der Ring besteht in diesem Beispiel aus dem Master M, den beiden an der Transaktion beteiligten Stationen A und B sowie beliebig vielen weiteren Stationen, die in der Abbildung nicht dargestellt sind. In jeder Station ist ein `sctrl_main` Modul implementiert, welches das top-level für den Zugriff auf das Slow Control-Netzwerk darstellt. Um den Lesebefehl senden zu können muss Station A darauf warten, dass sie ein Token erhält (Zeitschritt 1). Sie nimmt dieses dann auf, setzt das `typ`-Bit auf `'0'` und hängt die entsprechenden Felder für ein Lesekommando an (2). Jede der folgenden Stationen überprüft nun anhand des `dest`-Feldes ob sie

der Empfänger des Frames ist. Ist dies der Fall, so wird das Frame in einen internen Speicher kopiert, die beiden `ack`-Bits invertiert und das Frame läuft weiter durch den Ring, bis es wieder bei Station A ankommt (3). Falls eine Station nicht der Empfänger des Rahmens ist, wird das Frame ohne Modifikation weitergereicht. Wenn das quittierte Frame bei Station A angekommen ist weiß diese, dass der Befehl korrekt übermittelt wurde. Da es sich um einen Lesebefehl handelt müssen nun aber noch Daten von Station B zu Station A übertragen werden. Das heißt, dass jetzt Station B ein Frame senden will und daher ihrerseits auf ein Token warten muss (4). Nachdem sie das Token erhalten hat sendet sie ein Frame mit den gelesenen Daten über den Ring zu Station A (5) und erhält von dort ein Bestätigungsframe (6). Damit ist der Lesevorgang abgeschlossen.

2.2 Aufgabenstellung

In diesem Abschnitt wird die dieser Arbeit zugrundeliegende Aufgabenstellung thematisiert. Dabei wird zunächst das prinzipielle Performanceproblem des Slow Control-Netzwerks diskutiert. Anschließend wird auf die vorhandene Gigabit-Ethernet-Schnittstelle und schließlich auf das konkrete Problem dieser Arbeit eingegangen.

2.2.1 Motivation

Die Backplane dient als Trägersystem für bis zu 16 Nathan-Platinen, auf denen sich die ANN-ASICs befinden. Dadurch können bis zu 6.144 Neuronen und fast 1.600.000 Synapsen künstlich nachgebildet werden. Dies entspricht zwar nur einem Bruchteil der Komplexität des menschlichen Gehirns, jedoch arbeiten die künstlichen Neuronen auf einem Spikey-ASIC um einen Faktor von bis zu 10^5 schneller als ihre biologischen Vorbilder [9]. Wenn man für ein biologisches Neuron eine durchschnittliche Feuerrate von 1–10 Hz annimmt, so erhält man in der Hardware eine Feuerrate von bis zu 1 MHz. Da man das zeitliche Feuerverhalten möglichst vieler – im Idealfall aller – Neuronen auslesen möchte wird offensichtlich, dass enorme Datenmengen zu transferieren sind. Die Slow Control erweist sich hierfür als nicht geeignet. Zum einen liefert sie nur eine Übertragungsrate von maximal 100 MBit/s, zum anderen sind nur 32 % eines Slow Control-Frames Daten; der Rest sind Datenrahmen, Steuer- oder Prüfbits und Adressen (siehe Abschn. 2.1.5). Die Slow Control stellt somit kein probates Netzwerk dar um große Datenmengen zu befördern.

Ein weiteres Problem der Slow Control liegt in der Verwendung speziell angefertigter Hardware, nämlich der Darkwing-PCI-Karte, zur Steuerung des Systems. Da das entwickelte künstliche neuronale Netz Teil eines internationalen Forschungsprojektes ist, möchte man die Experimentierplattform auch Projektpartnern zur Verfügung stellen. Dies gelingt am einfachsten, indem man die Anbindung des Systems nach außen an internationalen Standards ausrichtet. Da nahezu jeder PC über eine Netzwerkkarte verfügt und da mittels der Gigabit-Ethernet-Technologie hohe Datenraten erreicht werden können, fiel die Wahl auf diesen Standard als Kommunikationsschnittstelle.

2.2.2 Gigabit-Ethernet

Die Netzwerk-Technologie Ethernet kommt in der Variante Gigabit-Ethernet auf der Hardware zum Einsatz, die in der Electronic Vision(s) Gruppe entwickelt wurde [10].

Dadurch werden Datentransfers zwischen einem PC und der Backplane mit bis zu 1.000 MBit/s ermöglicht.

Die Gigabit-Ethernet-Hardware besteht aus einem Gigabit-Ethernet-PHY und dem entsprechenden MAC. Der Ethernet-PHY DBGIG1 Gigabit Ethernet Phy Module der Firma Devboards [5] kann auf dem Santa-Cruz-Sockel auf der Backplane installiert werden und erlaubt den Anschluss der Backplane an jeden handelsüblichen PC. Der PHY ist für den Betrieb mit 10, 100 und 1.000 MBit/s ausgelegt. Für den Betrieb mit 1.000 MBit/s muss von außen ein Takt von 125 MHz bereitgestellt werden. Dieser wird auf der Backplane aus dem Takt des Quarz-Oszillators mit Hilfe eines DCMs im Balu-FPGA erzeugt.

Der MAC ist im Balu-FPGA implementiert. Als Design-Grundlage für den MAC diente ein 10/100 MBit-Ethernet-Core von OpenCores [19]. Dieser wurde von Christian Gutmann im Rahmen seiner Diplomarbeit so modifiziert und erweitert, dass er als MAC für Gigabit-Ethernet verwendet werden kann [10]. Somit ist prinzipiell eine schnelle Anbindung der Backplane an die Außenwelt realisiert.

2.2.3 Problemstellung und Lösungskonzept

In den letzten Abschnitten wurde die Hardware dargestellt, die in der Electronic Vision(s) Gruppe entwickelt wurde um großskalige künstliche neuronale Netze zu modellieren. Es wurde das Slow Control-Netzwerk dargestellt, über das die FPGAs auf der Backplane und auf den Nathan-Platinen angesprochen werden können. Auch wurde darauf hingewiesen, dass die Slow Control sich als nicht sonderlich geeignet erweist um große Datentransfers zu realisieren und dass die Darkwing-PCI-Karte als Schnittstelle zur Außenwelt den Einsatz bei Projektpartnern erschwert.

Da die Backplane über eine getestete Gigabit-Ethernet-Anbindung verfügt, die eine mehr als zehnmal so große Datenrate ermöglicht wie die Slow Control, liegt es nahe diese Schnittstelle zum Datentransfer zu nutzen. Der Gigabit-Ethernet-Core kann Daten mit bis zu 1 GBit/s von einem PC empfangen und zu ihm senden. Die Daten werden dabei in einem BRAM im Balu-FPGA gespeichert, bzw. aus einem BRAM ausgelesen. Die Anbindung der Nathan-Module an den Gigabit-Ethernet-Core, d.h. die Manipulation und der Transport dieser Daten zu den Modulen ist die zentrale Aufgabenstellung dieser Arbeit. Die dabei entwickelte Logik erweitert einen bestehenden Core (eth2sctrl-Core, siehe Abschn. 2.3). Hierbei war die Vorgabe, die bestehenden, praxiserprobten Designs für die Nathan-FPGAs unverändert weiterbenutzen zu können. Das bedeutet, dass die Nathan-Module nach wie vor über ein Slow Control-Interface angesprochen werden sollen. Die Daten, welche über die Ethernet-Schnittstelle geliefert werden, müssen also von der Logik im Balu-FPGA in Slow Control-Frames gerahmt und zu den Nathan-Modulen transportiert werden. Ebenso müssen die Frames, die von den Modulen zurückgesendet werden, analysiert und gespeichert werden, so dass der Ethernet-Core schließlich die wesentlichen Inhalte der Frames zum PC transportieren kann. Die Zielsetzung bei der Realisierung der Aufgabe war zum einen, die Datenrate von 1 GBit/s, die theoretisch erreichbar ist, möglichst voll auszunutzen. Zum Anderen ist die Komplexität der entwickelten Logik durch die im Balu-FPGA zur Verfügung stehenden Hardware-Ressourcen beschränkt. Durch den Ethernet-Core und die Slow Control-Logik sind bereits etwa 46 % (bezogen auf die Anzahl der Slices) des Balu-FPGAs belegt. Hierbei muss auch bedacht werden, dass ein FPGA nie zu 100 % mit Logik belegt werden kann; die einzelnen logischen Elemente müssen miteinander verbunden werden und dazu müssen bei einem relativ vollen FPGA auch einige LUTs als „route-through“, also als

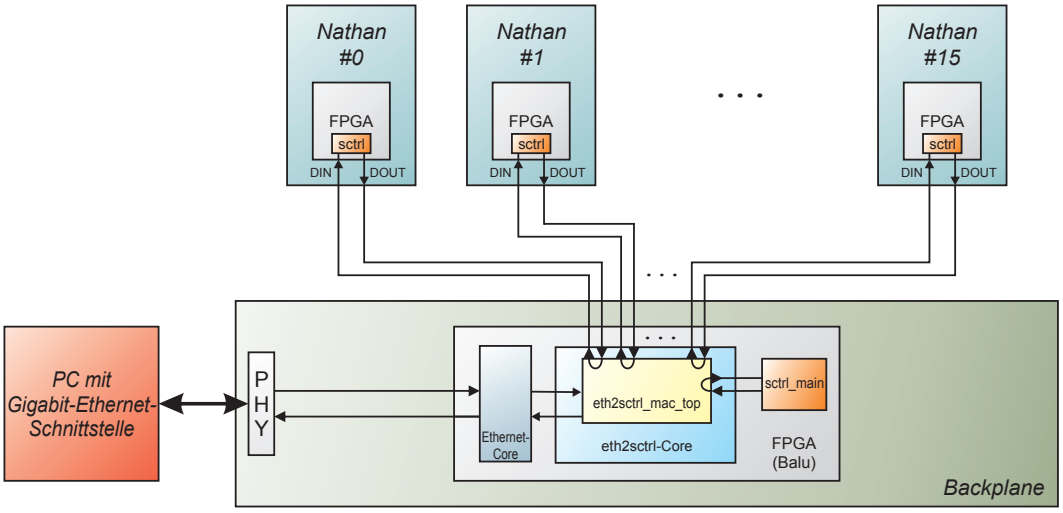


Abbildung 2.6: Schematische Darstellung der Topologie mit 17 minimalen Ringen. Der Datenausgang eines Nathan-Moduls wird von der Logik im Balu-FPGA analysiert und zur Weiterverarbeitung zwischengespeichert. Man beachte, dass hier im Gegensatz zu Abbildung 2.3 jeder Nathan-FPGA sich in einem eigenen Kommunikationsring mit dem Balu-FPGA befindet. Die Takt- und Reset-Leitungen sind nicht eingezeichnet. Sie verlaufen wie bei der ursprünglichen Implementierung der Slow Control sternförmig vom Balu-FPGA zu den Nathan-Sockeln.

Verbindungsglieder ohne jegliche logische Funktion, benutzt werden. Eine sinnvolle Obergrenze für ein FPGA-Design lässt sich bei etwa 90 % setzen. Daher musste bei der Konzeption darauf geachtet werden möglichst wenig redundante Strukturen zu erzeugen. Schließlich muss auch noch die Konfiguration der Nathan-FPGAs via Ethernet vom Balu-FPGA gesteuert werden. Im FPGA muss also zusätzliche Logik implementiert werden, welche einerseits die Steuersignale bei der Konfiguration entsprechend ansteuert und überwacht sowie andererseits den Transport des Bitstroms regelt.

Um einen möglichst hohen Datentransfer zu den Nathan-Modulen zu erreichen wurde die Ringtopologie der Slow Control aufgegeben. Wie in Abbildung 2.6 schematisch dargestellt ist, wurden stattdessen 17 minimale Ringe mit jeweils nur zwei Stationen implementiert. Eine Ringstation stellt dabei jeweils ein `sctrl_main` Modul im Nathan- bzw. Balu-FPGA dar; die andere Station ist der für diesen Zweck entwickelte MAC, dessen Funktion in Abschnitt 3.2 dargestellt wird. Dadurch wird es ermöglicht an mehrere Slow Control-Stationen *parallel* Datenframes zu senden bzw. von den Stationen zu empfangen. Nur durch diese Modifikation sind Übertragungsraten im Bereich von 1 GBit/s überhaupt denkbar.

Mit der Fertigstellung der Anbindung der Slow Control-Stationen an den Gigabit-Ethernet-Core wird eine Kommunikation mit der Darkwing-Karte über den SCSI-Sockel unnötig. Der einzige Kommunikationskanal von der Backplane nach außen ist in diesem System die Gigabit-Ethernet-Schnittstelle. Damit kann die Backplane als Experimentierplattform mit jedem PC verbunden werden.

Die Konzeption und Realisierung dieses Vorhabens wird in den nächsten Kapiteln dargestellt. Dabei wird zuerst die Implementierung der Slow Control-Anbindung behandelt und danach auf die Konfigurationslogik eingegangen. Die entwickelten Module bilden einen Teil des eth2sctrl-Cores. Zunächst wird daher noch auf die bereits

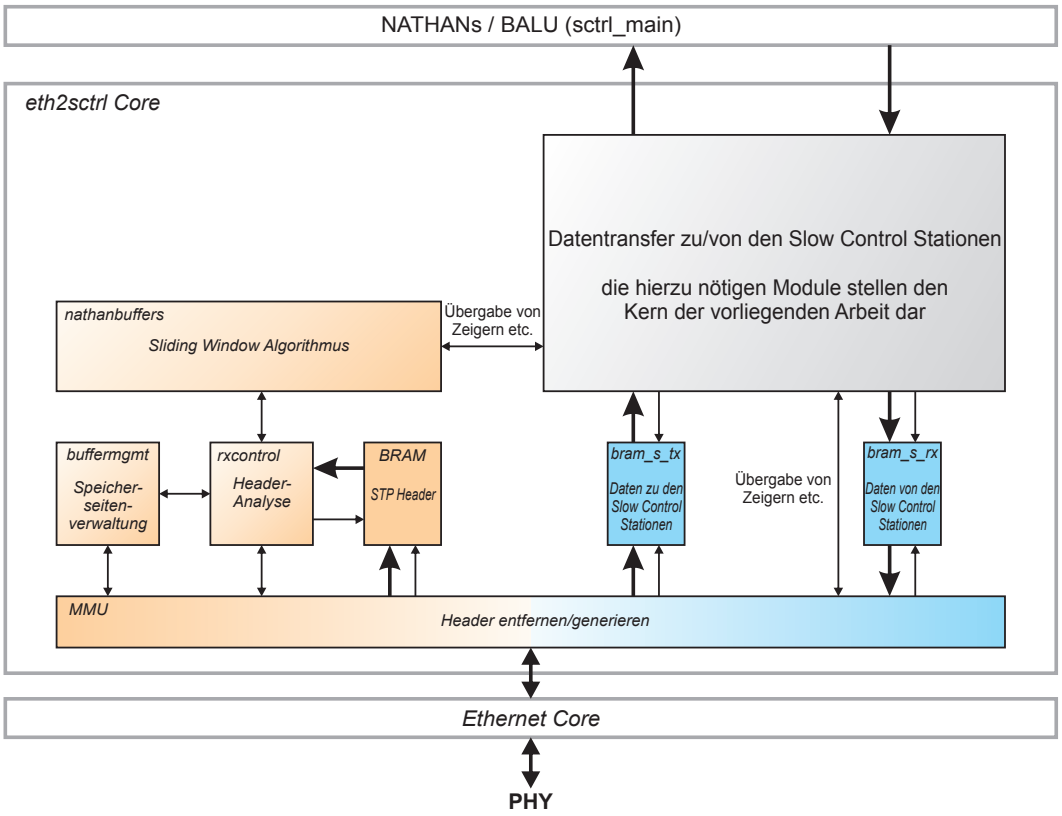


Abbildung 2.7: Blockschaltbild des eth2sctrl-Cores vor der Implementierung der Anbindung an das Slow Control-Interface. Die Module in der linken Hälfte und die MMU wurden von Olivier Jolly entwickelt und dienen der Headeranalyse, der Verwaltung des Speichers und der Umsetzung des Sliding-Window-Algorithmus.

bestehenden Teile dieses Cores eingegangen, die von Olivier Jolly während seiner Masterarbeit entworfen wurden.

2.3 Überblick über die bestehende Logik im Balu-FPGA

In diesem Abschnitt werden die zu Beginn der Arbeit existierenden Teile des eth2sctrl-Cores dargestellt [16]. Dieser Core stellt das Bindeglied zwischen dem Gigabit-Ethernet-Core und den Slow Control-Stationen dar. Er ist verantwortlich für die Bestätigung der empfangenen Ethernet-Frames, regelt die Speicherverwaltung und realisiert den Sliding-Window-Algorithmus (siehe 1.2.3). Das Blockschaltbild mit den zu Beginn der vorliegenden Arbeit vorhandenen Modulen und deren schematischen Verbindungen ist in Abbildung 2.7 gezeigt.

2.3.1 Slow Control Transport Protokoll (STP)

Um Slow Control-Befehle via Ethernet transportieren zu können, musste zunächst ein Protokoll festgelegt werden, nach dem die entsprechenden Felder des Slow Control-Frames in ein Ethernet-Frame verpackt werden. Da der Paket-Transfer via Ethernet den Sliding-Window-Algorithmus (siehe Abschn. 1.2.3) verwenden soll, muss auch das Problem behandelt werden, dass neue Pakete aufgrund von Übertragungsfehlern vor älteren auf der Backplane ankommen. Das Protokoll welches für diesen Zweck de-

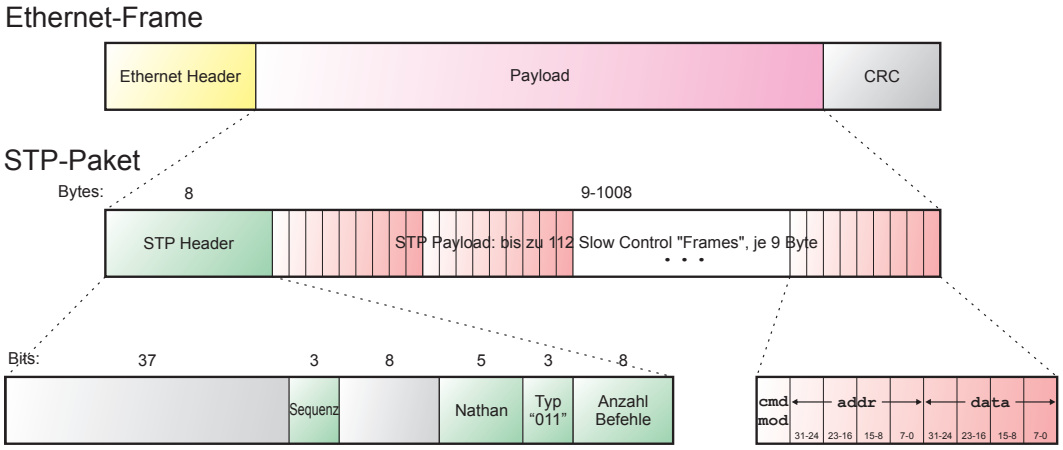


Abbildung 2.8: Definition des STP-Paketformats. Diese Pakete dienen dazu Slow Control-Befehle in Ethernet-Frames zu verpacken um dieses dann zur Backplane zu transferieren. Der Payload stellt bis zu 112 Slow Control-Frames dar, von denen jeweils die Felder `cmd`, `mod`, `addr` und `data` transportiert werden.

finiert wurde wird als *Slow Control Transport Protokoll* (*STP*) bezeichnet. Für Einzelheiten zu diesem Protokoll sei auf [16] verwiesen; hier soll lediglich das Paketformat zum Transport von Slow Control-Befehlen beschrieben werden[12]. Dies wurde im Rahmen der vorliegenden Arbeit modifiziert um die Handhabung der Daten zu vereinfachen. Die neue Definition ist in Abbildung 2.8 dargestellt. Das STP-Paket wird als Payload in das Ethernet-Frame eingebettet. Es trägt einen Header, in dem der Typ des Pakets, eine Sequenz-Nummer bezüglich des Sliding-Window-Algorithmus sowie die Anzahl der im Paket enthaltenen Slow Control-Kommandos hinterlegt sind. Da ein Ethernet-Frame immer nur Befehle für ein spezifisches Nathan-Modul transportiert, wird auch dessen Nummer im Header gespeichert. Den Payload des STP-Pakets bilden die Felder `cmd`, `mod`, `addr` und `data` des Slow Control-Frames. In Abschnitt 2.1.5 wurde darauf hingewiesen, dass nicht jedes Frame über all diese Felder verfügt. Beim Transport mittels STP-Paketen müssen jedoch immer alle Felder angegeben werden. Für jeden Slow Control-Befehl werden somit innerhalb des STP-Pakets neun Byte benötigt. In Bezug auf die Ausnutzung der verfügbaren Bandbreite spielt dies kaum eine Rolle, da bei Weitem die meisten Kommandos Schreib-/Lesebefehle sind und eine vollständige Angabe aller Felder vonnöten ist.

2.3.2 MMU

Die MMU[13] hat die Aufgabe Speicherzugriffe auf die BRAM-Strukturen zu regeln. Dies beinhaltet die Trennung des STP-Headers vom Payload und deren Weiterleitung in das entsprechende BRAM. Ferner muss sichergestellt werden, das nicht mehrere Module gleichzeitig auf verschiedene Adressen in einem der BRAMs zugreifen wollen. Daher richten alle Module ihre Speicheranfragen an die MMU. Diese regelt dann die Reihenfolge der Zugriffe nach ihrer Priorität[14].

[12] Im Folgenden wird sowohl das Protokoll an sich, als auch der spezielle Pakettyp zum Transport von Slow Control-Befehlen als STP bezeichnet.

[13] Memory Management Unit

[14] Bei den implementierten BRAMs handelt es sich um dual-port BRAMs mit je zwei Adressports A und B; nur der Zugriff auf Port A muss von der MMU geregelt werden.

2.3.3 buffermgmt

Das Modul `buffermgmt` dient der Speicherverwaltung und behandelt das Problem der Speicher-Fragmentierung. Aufgrund des Sliding-Window-Algorithmus ist es nicht möglich vorherzusagen, wann ein empfangenes Paket vollständig aus dem BRAM gelesen und zum entsprechenden Nathan-Modul transferiert wurde. Es wird also im realen Betrieb zu Situationen kommen, bei denen manche Pakete sofort nachdem sie gespeichert wurden gelesen werden, wohingegen andere für einen längeren Zeitraum im Speicher verbleiben ohne verarbeitet zu werden. Somit ist es nicht möglich den Speicher einfach von Adresse `0x0` bis zum Ende zu beschreiben und danach in derselben Reihenfolge auszulesen. Um dieses Problem zu lösen wurde das BRAM in Speicher-*Blöcke* oder Seiten eingeteilt. Jeder Block kann dabei entweder leer sein, oder genau ein STP-Paket enthalten. Die Verwaltung welche Blöcke leer und welche mit gültigen Daten belegt sind wird vom Modul `buffermgmt` übernommen.

2.3.4 rxcontrol

Bei diesem Modul handelt es sich um eine FSM, die den Header des STP-Pakets analysiert. Sie entscheidet ob der Speicherblock in dem sich das Paket befindet als gültig angesehen wird oder, falls es sich beispielsweise um einen unbekannten Pakettyp handelt, ob der Speicherplatz wieder freigegeben werden soll.

2.3.5 nathanbuffers

Dieses Modul stellt die Realisierung des Sliding-Window-Algorithmus in der Hardware dar. Es hat die Aufgabe die Fenster anhand ihrer Sequenz-Nummer ggf. neu zu ordnen und dafür Sorge zu tragen, dass niemals neue STP-Pakete vor älteren aus dem BRAM gelesen werden. Zu diesem Zweck wird für jede Slow Control-Station ein Schiebefenster mit einer Tiefe von vier Paketen implementiert. Über die Ausgangsports dieses Moduls werden auf Anfrage die Block-Nummer des STP-Pakets mit der nächsten Sequenznummer sowie die Anzahl der darin gespeicherten Slow Control-Kommandos bereitgestellt. Nachdem alle Slow Control-Befehle aus einem Speicherblock gelesen wurden muss dies dem Modul `nathanbuffers` mitgeteilt werden. Darauf hin kann das Modul die Blocknummer für den Block mit der nächsten Sequenznummer bereitstellen. Dieser Vorgang wird als *Shiften* oder *Inkrementieren* bezeichnet. Die Schnittstelle zu diesem Modul ist für den Datenfluss zu den Slow Control-Stationen von zentraler Bedeutung; hierauf wird in Abschnitt 3.1 noch detailliert eingegangen.

2.3.6 BRAM

Es existieren drei BRAMs die der Speicherung der Header-Daten sowie der zu sendende und der empfangenen Slow Control-Daten dienen (`bram_s_tx` und `bram_s_rx`). An dieser Stelle soll kurz auf die Dimensionierung von `bram_s_tx` eingegangen werden. Das BRAM verfügt über einen 64 Bit breiten Dateneingang seitens der MMU und einen 8 Bit breiten Datenausgang an der Schnittstelle zur Logik für den Datenpfad. Aus der Sicht der MMU können 8.192 Datenworte in 64 Blöcken in diesem BRAM abgelegt werden. Die Kapazität beträgt somit 512 kBit, was weit mehr als die Hälfte der verfügbaren Ressourcen des FPGA in Anspruch nimmt.

3 Anbindung an das Slow Control Interface

In diesem Kapitel wird dargestellt welche Module entwickelt und implementiert wurden um den Transfer von Slow Control-Befehlen via Gigabit-Ethernet zu den Nathan-Modulen zu ermöglichen. Ebenso musste der Datenrücktransfer von den Nathan-Modulen über das Ethernet zum PC konzipiert werden. Die entwickelte Logik soll auch in der Lage sein, den Konfigurationsvorgang der Nathan-FPGAs durchzuführen. Hierauf wird jedoch erst im folgenden Kapitel eingegangen.

Abbildung 3.1 zeigt einen schematischen Überblick über den gesamten eth2sctrl-Core. Die Module `shifters_s_tx`[1], `shifters_s_rx` sowie `eth2sctrl_mac_top` wurden im Rahmen der vorliegenden Arbeit entworfen und implementiert. Sie regeln die Serialisierung von Daten aus `bram_s_tx` und das Verpacken der Daten in Slow Control-Frames. Ebenso wird der Empfang von Frames von den Slow Control-Stationen, deren Bestätigung und Speicherung in einem BRAM (`bram_s_rx`) für den Rücktransport via Ethernet realisiert. Die Konzeption des Moduls `eth2sctrl_mac_top` sieht dabei vor, immer an alle Stationen parallel Slow Control-Frames bzw. Tokens zu senden. Diese Implementierung wurde gewählt um logische Strukturen, wie z.B. Zähler für die Bit-Position innerhalb eines Frame, für alle Stationen nur einmal instantiieren zu müssen und somit Logikressourcen im FPGA zu sparen.

Aus der Abbildung ist ersichtlich, dass die Module in drei unterschiedlichen Takt-Domänen arbeiten. `shifters_s_tx` und `shifters_s_rx` arbeiten jeweils mit einer Frequenz von 125 MHz; dies entspricht der Taktrate mit der der Gigabit-Ethernet-PHY Daten sendet und empfängt. Das Modul `eth2sctrl_mac_top` hingegen wird mit der `sr_clk`[2] betrieben, die mit der Oszillatorfrequenz von 156,25 MHz getaktet ist. Auch die physikalische Anbindung der Nathan-Module an den Core nutzt diese Frequenz. Zum Betrieb der Konfigurationslogik in `eth2sctrl_mac_top` wird noch ein weiterer Takt mit 39 MHz benötigt (`CCLK`). Mit dieser Frequenz erfolgt auch die physikalische Anbindung der Nathan-Module an den Core während der Konfiguration. An den Übergängen zwischen den einzelnen Domänen befinden sich jeweils asynchrone FIFOs.

Die folgenden Abschnitte geben einen detaillierten Einblick in die Aufgaben sowie die konkreten Funktionsprinzipien der einzelnen Module. Dabei wird auch auf deren Kommunikation untereinander und mit der übrigen Logik eingegangen. Die Reihenfolge der Darstellung erfolgt dabei entlang des realen Datenpfades eines Slow Control-Befehls durch das System. Diese Vorgehensweise soll das Verständnis des Systems erleichtern und das Zusammenwirken der einzelnen Komponenten nachvollziehbar machen.

[1] Anmerkung zur Nomenklatur: TX bedeutet transmit data; das Präfix s soll darauf hindeuten, das es sich hierbei auf das Senden von Daten *zu den Slow Control-Stationen* bezieht (und nicht zum Ethernet). Diese Bezeichnungsweise wird im Rahmen dieser Arbeit einheitlich verwendet.

[2] sr: slow control ring

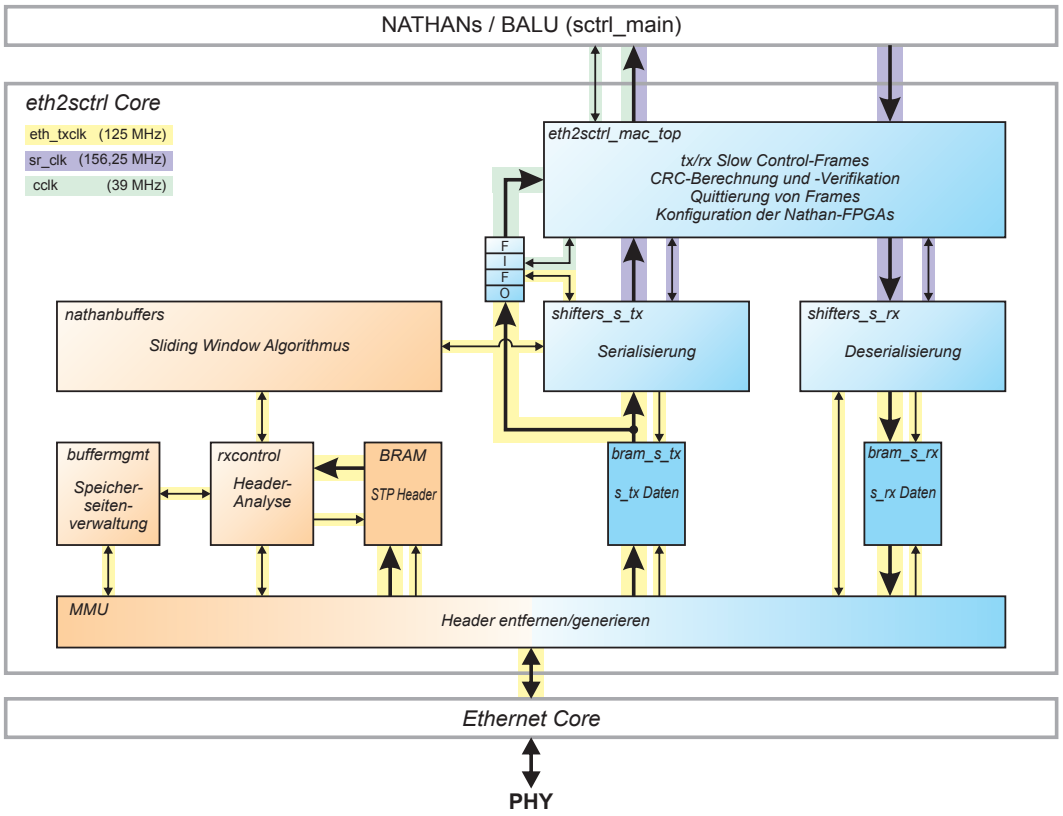


Abbildung 3.1: Schematische Übersicht der Module im eth2sctrl-Core. Datenpfade sind durch dicke Pfeile gekennzeichnet. Die Module in der linken Hälfte der Abbildung und die MMU wurden in Abschnitt 2.3 vorgestellt. Sie regeln die Analyse von empfangenen STP-Paketen sowie die Verwaltung der Empfängerfenster. Die Module im rechten Teil wurden im Rahmen dieser Arbeit entwickelt und regeln die Anbindung der Slow Control-Stationen an den eth2sctrl-Core.

3.1 Serialisierung: shifters_s_tx

Die Slow Control-Stationen werden jeweils über eine serielle ein Bit breite Datenleitung angesprochen. Da die Daten in `bram_s_tx` jedoch in acht Bit breiten Datenworten gespeichert sind, muss eine Serialisierung dieser Daten erfolgen. Diese Aufgabe wird im Modul `shifters_s_tx` realisiert, dessen schematischer Aufbau in Abbildung 3.2 gezeigt ist. Dieses Modul liest Daten aus `bram_s_tx`, wobei es die dafür benötigten Informationen zur Adressberechnung von `nathanbuffers` erhält. Die Daten werden dann über eine Shiftregister-Matrix serialisiert und anschließend in einem asynchronen FIFO gespeichert (`fifo_s_tx`). Das Modul `eth2sctrl_mac_top` kann die Daten dann aus diesem FIFO auslesen und in Frames verpackt an die Slow Control-Stationen weiterleiten. Zur Steuerung der Shiftregister-Matrix und der Kommunikation mit `nathanbuffers` sind mehrere FSMs implementiert, auf deren Funktion noch im Detail eingegangen wird.

3.1.1 Interface zu nathanbuffers

Über die Schnittstelle zum Modul `nathanbuffers` werden die Informationen ausgetauscht, die für den Lesezugriff auf `bram_s_tx` sowie für das Inkrementieren des

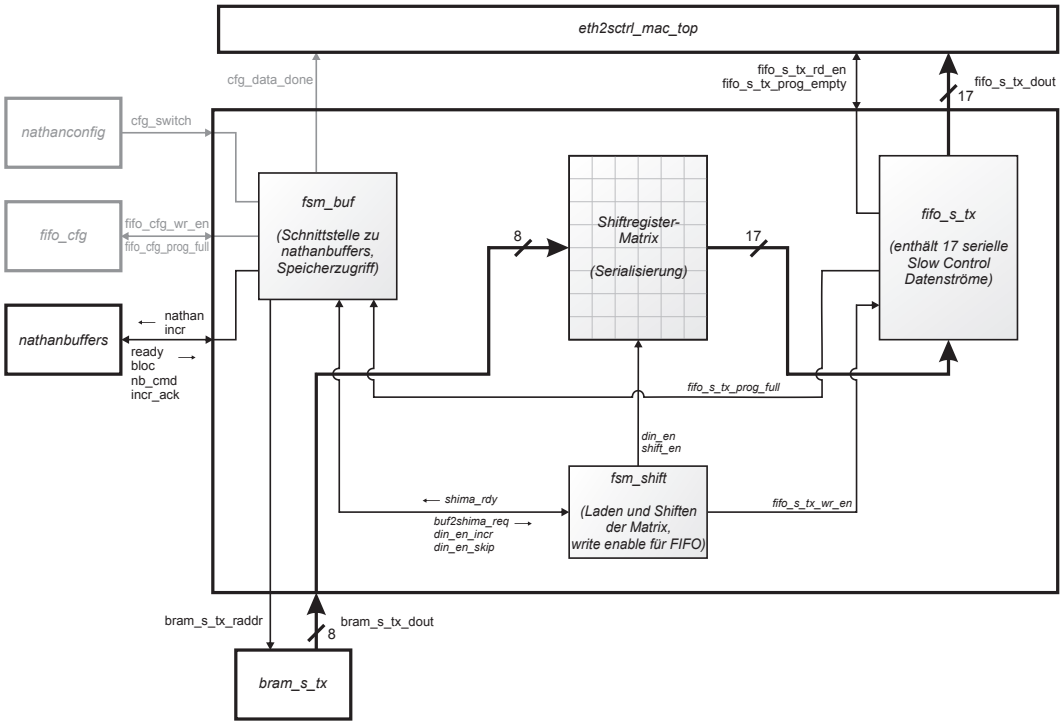


Abbildung 3.2: Blockschaltbild des Moduls `shifters_s_tx`. Dieses Modul hat die Aufgabe die Daten aus `bram_s_tx` zu serialisieren und in `fifo_s_tx` abzulegen. Die zur Konfiguration notwendigen Signale und Module werden im nächsten Kapitel erläutert. Eine vollständige Liste aller Ports sowie eine Beschreibung deren Funktion findet sich in Anhang B.

Empfängerfensters bezüglich des Sliding-Window-Algorithmus von entscheidender Bedeutung sind.

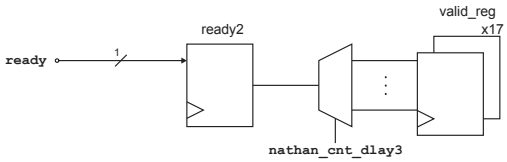
Nachdem `shifters_s_tx` eine gültige Nathannummer an den Ausgangsport `nathan` angelegt hat vergehen exakt zwei Takte, bis `nathanbuffers` an seinem Ausgang den Zeiger auf einen Speicherblock `bloc`, die Anzahl der Slow Control-Befehle in diesem Block `nb_cmd` und das Statussignal `ready` für das entsprechenden Nathan-Modul bereitstellt. Bei dieser Verzögerung handelt es sich lediglich um eine Anlauf-Latenz, d.h. der Port `nathan` darf sich in jedem Takt ändern.

Das Verschieben der Empfängerfenster erfolgt nach einem *Handshake-Protokoll*, d.h. `shifters_s_tx` setzt das Ausgangssignal `incr` solange auf `'1'`, bis es eine Bestätigung in Form des Signals `incr_ack` von `nathanbuffers` erhält (siehe S. 40).

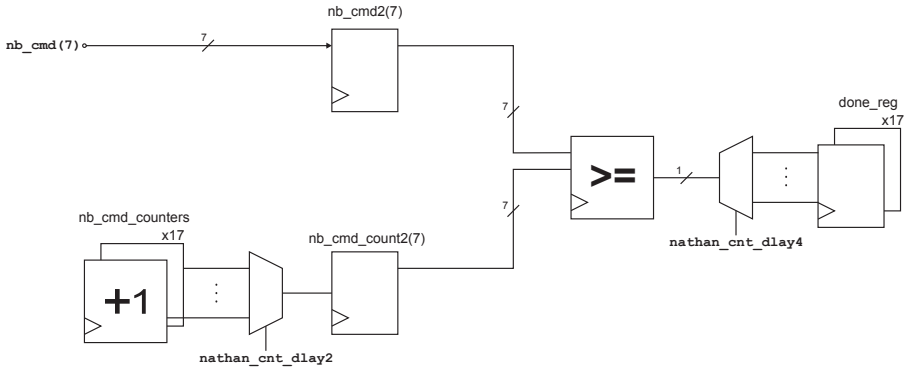
3.1.2 Statusregister

Zur Steuerung des Ablaufs der internen Logik existieren für jede Slow Control-Station zwei Status-Flags[3], `valid_reg` und `done_reg`. Diese Register werden von der FSM `fsm_buf` gesetzt (siehe Abschn. 3.1.5). Falls für eine Station ein gültiges STP-Paket vorliegt, muss der entsprechende Eintrag in `valid_reg` gesetzt werden. Das Register `done_reg` dient der Markierung jener Stationen, für die gerade das letzte Slow Control-Kommando aus dem entsprechenden Speicherblock in `bram_s_tx` gelesen wird.

[3] als Flag wird eine binäre Variable oder ein Signal bezeichnet, welche als Hilfsmittel zur Kennzeichnung bestimmter Zustände benutzt werden kann



(a) `valid_reg`



(b) `done_reg`

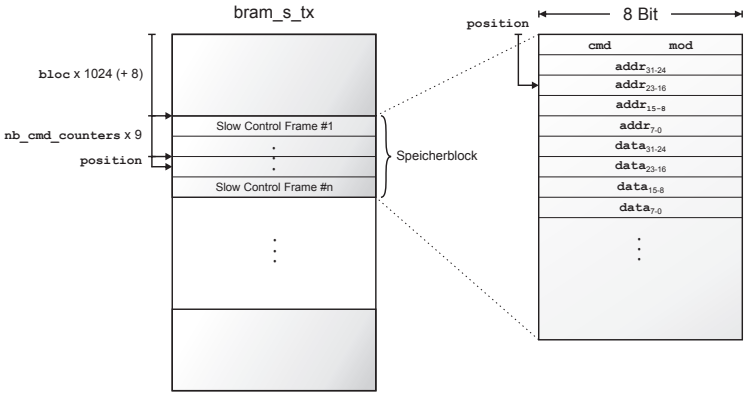
Abbildung 3.3: Schematische Darstellung der Logik für das Setzen der Statusflags im Modul `shifters_s_tx`.

Die Logik, die diese Flags setzt, ist in Abbildung 3.3 dargestellt. Das Setzen des valid Flags wird über einen Demultiplexer realisiert, dessen Steuerleitung vom internen Zähler `nathan_count` abhängt. Dieser Zähler steuert auch den Ablauf von `fsm_buf` und legt den Wert des Ausgangsports `nathan` fest. Aufgrund des zeitlichen Verhaltens des Moduls `nathanbuffers` und aufgrund der Tatsache, dass alle Signaleingänge registriert sind folgt jedoch, dass nicht der Zähler selbst als Steuersignal für den Demultiplexer dienen kann, sondern der Wert des Zählers vor drei Takten benutzt werden muss. Diese Verzögerung wird in Hardware realisiert indem `nathan_count` mit dem Eingang eines Schieberegisters verbunden wird. Dieses Schieberegister stellt Werte von `nathan_count` mit unterschiedlichen Verzögerungen von bis zu vier Takten bereit.

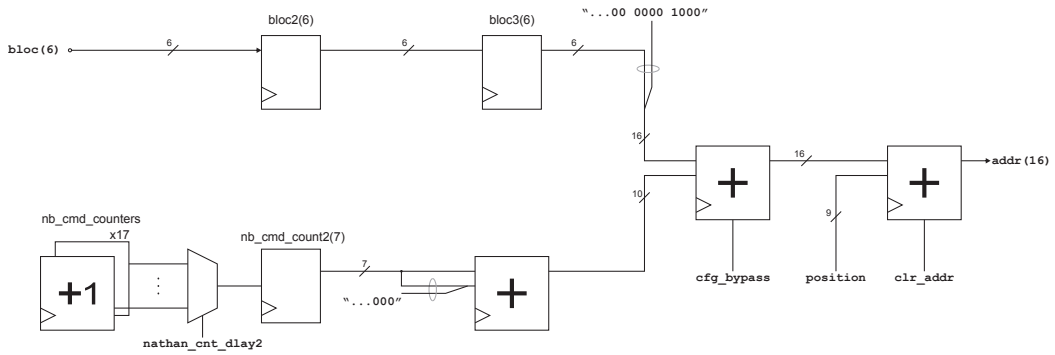
Die Zuweisung des done Flags ist etwas komplizierter. Hier muss die Anzahl der für eine Station im aktuellen Speicherblock verfügbaren Slow Control-Befehle verglichen werden mit der Anzahl der bereits vollständig ausgelesenen. Letztere wird in dem internen Zählerarray `nb_cmd_counters` gespeichert. Jeder Eintrag in diesen Array entspricht einer Slow Control-Station. Um das Flag zuweisen zu können muss zunächst der korrekte Eintrag aus `nb_cmd_counters` in ein Register gemultiplext werden. Als Steuerleitung wird hierbei das Signal `nathan_cnt_dlay2` verwendet, das dem Schieberegister für `nathan_count` entnommen werden kann. Der Wert des Registers kann dann mit dem Eingangsregister für `nb_cmd` verglichen werden. Anschließend erfolgt ein Demultiplexen des Komparatorausgangs auf den korrekten Eintrag in `done_reg`.

3.1.3 Datenlayout und Adressberechnung

In diesem Abschnitt soll auf die Anordnung der Slow Control-Daten in `bram_s_tx` sowie die Berechnung der Leseadressen eingegangen werden.



(a) Schematische Darstellung der einzelnen Summanden bei der Adressberechnung



(b) Realisierung der Adressberechnung im FPGA

Abbildung 3.4: Prinzip und Implementierung der Adressberechnung für den Lesezugriff auf `bram_s_tx`.

Die Aufteilung des BRAMs in Speicherblöcke ist in Abbildung 3.4(a) dargestellt. Jeder Block enthält bis zu 112 Befehle für eine Slow Control-Station. Die Anordnung der einzelnen Slow Control-Felder innerhalb eines Blockes ist ebenfalls der Abbildung zu entnehmen. Um diese Anordnung zu erreichen wurde die bestehende Definition des STP-Paketformats auf das in Abbildung 2.8 (siehe S. 28) gezeigte Format geändert.

Um nun die Adresse für einen Lesezugriff berechnen zu können müssen mehrere Faktoren berücksichtigt werden. Zum einen ist dies die Block-Adresse die von `nathanbuffers` bereitgestellt wird. Des Weiteren müssen aber auch die internen Zähler für die Anzahl der bereits aus diesem Block gelesenen Slow Control-Kommandos (`nb_cmd_counters`) und für die Byteposition innerhalb des aktuellen Kommandos (`position`) berücksichtigt werden. Der Zähler `position` läuft dabei von null bis acht, da für jedes Slow Control-Kommando neun Byte in `bram_s_tx` gespeichert sind. Weil jedoch auch bei der Realisierung des Konfigurationsvorgangs ein Positionszeiger benötigt wird und es keinen Sinn macht hierfür einen eigenen Zähler zu implementieren, musste ein 9-Bit Zähler verwendet werden (siehe Kapitel 4). Dieser Zähler wird nur einmal benötigt, da diese Position für alle Slow Control-Stationen identisch ist.

Die Block-Adresse gibt an, ab welcher Adresse in `bram_s_tx` ein STP-Paket abgelegt ist. Aufgrund der Blockgröße von 1024 Byte muss sie mit 1024 multipliziert werden um hieraus eine BRAM-Leseadresse zu erhalten. Hierbei ist allerdings zu

beachten, dass die ersten acht Byte in `bram_s_tx` aufgrund von Implementierungsdetails der MMU immer frei bleiben, und daher immer ein konstanter Offset Acht aufaddiert werden muss um die Adresse des ersten Datenwortes eines STP-Pakets zu erhalten. Zu der so erhaltenen Basisadresse muss nun das neunfache der Anzahl der bereits aus diesem Speicherblock gelesenen Slow Control-Befehle addiert werden. Der Faktor neun rührt daher, dass für jedes Slow Control-Kommando exakt neun Byte benötigt werden. Die Byte-Position wird schließlich durch den Zähler `position` bestimmt. Dieser muss noch zum Zwischenergebnis addiert werden um die endgültig Leseadresse zu erhalten.

Die Realisierung dieser Additionsschritte wurde in Hardware in einer Pipeline realisiert. Diese ist in Abbildung 3.4(b) dargestellt. Die Anzahl der bereits vollständig aus `bram_s_tx` gelesenen Kommandos ist in dem Array `nb_cmd_counters` gespeichert. Zunächst muss nun der Eintrag für jene Slow Control-Station für den Daten aus `bram_s_tx` gelesen werden sollen aus diesem Array gemultiplext und in einem Pipelineregister gespeichert werden. Da möglichst wenig FPGA-Ressourcen verbraucht werden sollen, bietet es sich an diesen Teil mit der Logik für das Setzen der done Flags zu teilen. Dies hat zur Folge, dass der Eingangsvektor `bloc` um einen zusätzlichen Takt verzögert werden muss um korrekte Additionsergebnisse zu garantieren. Der Durchsatz des Systems wird dadurch jedoch nicht negativ beeinflusst, lediglich wird die Anlauflatenz um einen Takt erhöht. Der zwischengespeicherte Zählerwert `nb_cmd_counter2` muss nun mit neun multipliziert werden. Dies wird in Hardware als Addition des um drei Stellen links-verschobenen Registerwertes mit sich selbst realisiert. Anschließend erfolgt die Addition des so erhaltenen Wertes mit der Basisadresse, wobei diese vorher mit 1024 multipliziert und um den Offset korrigiert wird. Zuletzt wird die Byteposition innerhalb des Slow Control-Frames in Form des Zählers `position` aufaddiert. Das Ergebnis dieser Schritte ist die Leseadresse `bram_s_tx_raddr` und wird direkt mit dem Adressport von `bram_s_tx` verbunden.

Es muss noch darauf hingewiesen werden, dass die im Folgenden dargestellte Logik auch Speicheradressen für Slow Control-Stationen berechnen würde, für die keine gültigen Daten vorliegen. Es muss ein Mechanismus implementiert werden, der verhindert, dass ungültige Daten aus `bram_s_tx` gelesen und weiterverarbeitet werden. Dies wurde hier realisiert, indem der letzte Addierer in der Pipeline, dessen Ausgangssignal direkt mit `bram_s_tx` verbunden ist, über einen synchronen Reset-Eingang verfügt. Das Reset-Signal (`clr_addr`) wird dabei in Abhängigkeit vom Wert `nathan_count` und den Einträgen des Statusregisters `valid_reg` generiert. Diese Methode funktioniert nur deshalb, weil an Adresse `0x0000` in `bram_s_tx` immer der Wert `0x00` gespeichert ist.

3.1.4 Shiftregister-Matrix

Die Shiftregister-Matrix serialisiert die acht Bit breiten Datenworte aus `bram_s_tx`. Ihr Konzept ist in Abbildung 3.5 schematisch dargestellt und soll hier erläutert werden.

Die Matrix besteht aus 17 zeilenweise angeordneten Schieberegistern (eines für jedes Nathan-Modul + eines für den Balu-FPGA) deren Tiefe jeweils der Breite der Worte in `bram_s_tx` entspricht, also acht Bit. Jedes dieser Schieberegister verfügt über einen parallelen Dateneingang. Ziel ist es nun diese Matrix *zeilenweise* zu Füllen, und zwar so, dass in der n-ten Zeile acht Bit für das Nathan-Modul `#n` gespeichert werden. Die letzte Zeile ist für die Slow Control-Station im Balu-FPGA vorgesehen. In den Zeilen werden dabei stets *korrespondierende Bits* der jeweiligen Slow Control-

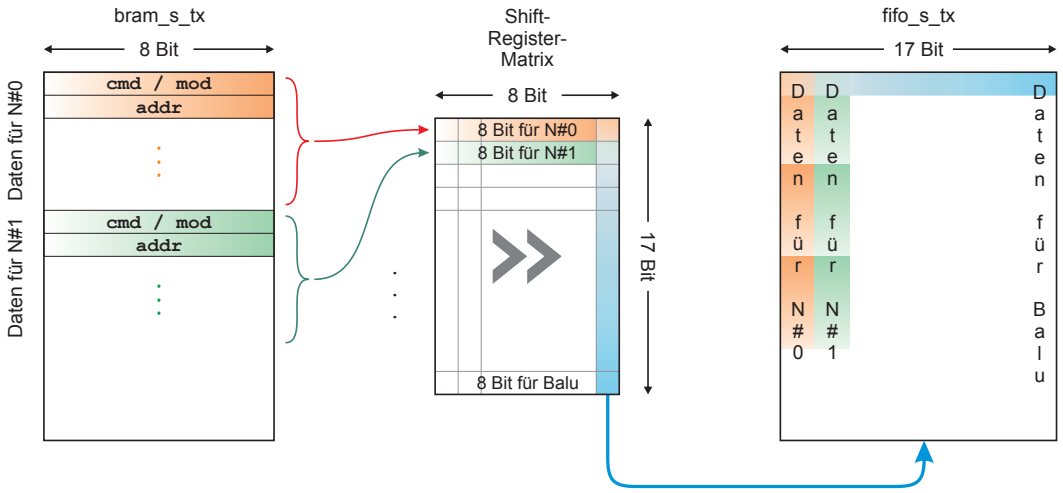


Abbildung 3.5: Schematische Darstellung der Datenanbindung der Shiftregister-Matrix an `bram_s_tx` und `fifo_s_tx`. Die Matrix wird zeilenweise beschreiben, wobei jede Zeile fest einer Slow Control-Station zugeordnet ist. Das Shiften der Matrix erfolgt spaltenweise, so dass `fifo_s_tx` schließlich 17 serialisierte Slow Control-Befehle enthält.

Kommandos für die Slow Control-Stationen gespeichert, also z.B. jeweils acht Bit für die `cmd` und `mod` Felder. Nachdem die Matrix befüllt wurde, wird sie *spaltenweise* in Richtung des MSB geshiftet, wobei der Datenausgang der vordersten Spalte mit dem Dateneingang von `fifo_s_tx` verbunden ist. Dadurch werden die Slow Control-Daten für alle 17 Stationen simultan serialisiert. Falls sich für eine Station keine Daten in `bram_s_tx` befinden wird die entsprechende Zeile der Matrix mit Nullen gefüllt. Dies führt dazu, dass in `fifo_s_tx` eine komplette Spalte leer bleibt. Werden nun Daten aus `fifo_s_tx` gelesen, so erhält man 17 serialisierte Datenströme, die anschließend noch in Frames gerahmt und zu den Stationen transferiert werden müssen. Diese Aufgabe übernimmt das Modul `eth2sctrl_mac_top` (siehe Abschn. 3.2).

3.1.5 fsm_buf

Die FSM `fsm_buf` ist zuständig für die Kommunikation mit `nathanbuffers`, welches den Sliding Window-Algorithmus realisiert und die Informationen bereithält, für welche Slow Control-Station in welchem Speicherblock in `bram_s_tx` wie viele Befehle gespeichert sind. Aus diesen Informationen können dann die Adressen für den BRAM-Lesezugriff berechnet werden (siehe Abschn. 3.1.3).

Zu diesem Zweck werden von `nathanbuffers` 17 Zeiger (*Pointer*) bereitgestellt, die den Beginn der STP-Pakete in `bram_s_tx` für alle Stationen anzeigen. Die FSM `fsm_buf` soll ausgehend von diesen Zeigern Adressen innerhalb der Pakete berechnen und die Slow Control-Daten aus `bram_s_tx` auslesen, um sie über die Shiftregister-Matrix zu serialisieren. Die Zeiger werden dabei intern *nicht* dauerhaft gespeichert und müssen jedes Mal erneut von `nathanbuffers` abgefragt werden wenn auf `bram_s_tx` zugegriffen werden soll. Diese Implementierung wurde gewählt um redundante Register im FPGA zu vermeiden.

Der Zustandsgraph ist in Abbildung 3.6 dargestellt. Die Steuerung der Zustandsübergänge erfolgt anhand der Zähler `nathan_count` sowie `position`. Im Folgenden

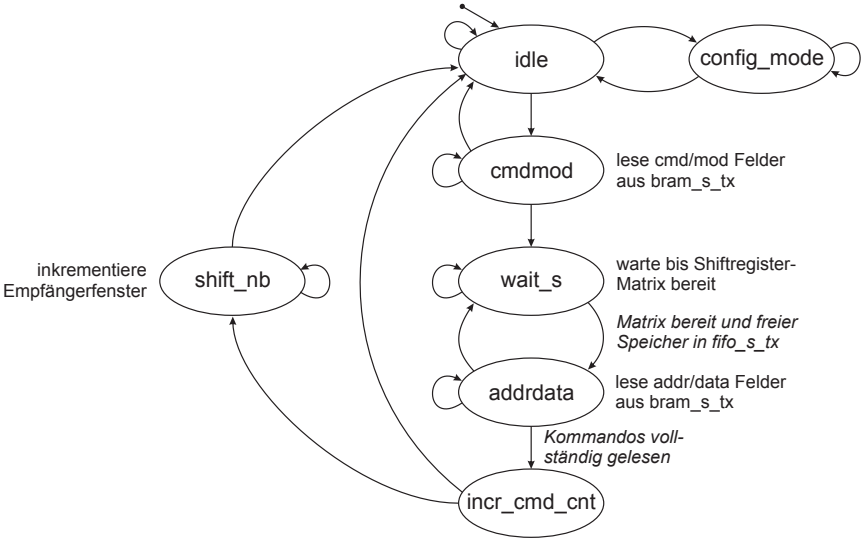


Abbildung 3.6: Zustandsgraph der FSM `fsm_buf`. Die FSM steuert die Kommunikation mit dem Modul `nathanbuffers` und die Berechnung der Leseadressen für `bram_s_tx`.

wird beschrieben welche Aufgaben `fsm_buf` in den jeweiligen Zuständen zu bewältigen hat, und wie dies realisiert ist.

idle `fsm_buf` verlässt den `idle`-Zustand, sobald sie von der FSM `fsm_shift`, welche die Shiftregister-Matrix steuert, ein Signal erhält, das die Bereitschaft der Matrix zur Aufnahme von Daten signalisiert (siehe Abschn. 3.1.6). Außerdem kann `fsm_buf` den Ruhezustand nur verlassen, wenn in `fifo_s_tx` genügend freier Speicherplatz zur Verfügung steht. Daraufhin erfolgt ein Wechsel in den Zustand `cmdmod` und es wird ein Anforderungssignal für `fsm_shift` gesetzt. Es besteht auch die Möglichkeit den `idle`-Zustand in den Konfigurationszustand `config_mode` zu verlassen (siehe Kapitel 4).

cmdmod Hier werden alle gültigen `cmd` und `mod` Felder für die einzelnen Slow Control-Stationen aus `bram_s_tx` ausgelesen und in die Shiftregister-Matrix geschrieben. Nacheinander wird in diesem Zustand für *alle* Stationen überprüft, ob gültige STP-Pakete in `bram_s_tx` gespeichert sind, und in welchem Speicherblock diese sich befinden. Die Anzahl der Slow Control-Befehle im entsprechenden Block wird ebenfalls überprüft. All diese Informationen werden von `nathanbuffers` bereitgestellt. Auch werden die beiden Statusregister `valid_reg` und `done_reg` in diesem Zustand gesetzt und bleiben gültig, bis `fsm_buf` wieder in den Ruhezustand zurückkehrt. Ausgehend von der Blockadresse und den internen Zählern kann dann die Leseadresse berechnet werden (siehe Abschn. 3.1.3).

Falls für keine Station gültige Pakete vorliegen aktiviert `fsm_buf` ein Reset-Signal, welches die FSM `fsm_shift` in den Ruhezustand versetzt und den Inhalt der Shiftregister-Matrix löscht. Dadurch wird der Serialisierungsprozess abgebrochen. In diesem Fall wechselt `fsm_buf` wieder in den Zustand `idle`, ansonsten geht sie nach `wait_s` über.

wait_s In diesem Zustand wartet `fsm_buf` auf das Bereitschaftssignal `shima_rdy`[4] von `fsm_shift`. Dieses Signal zeigt an, dass die Shiftregister-Matrix ihren Inhalt se-

[4] shima: **shi**ft register **ma**trix

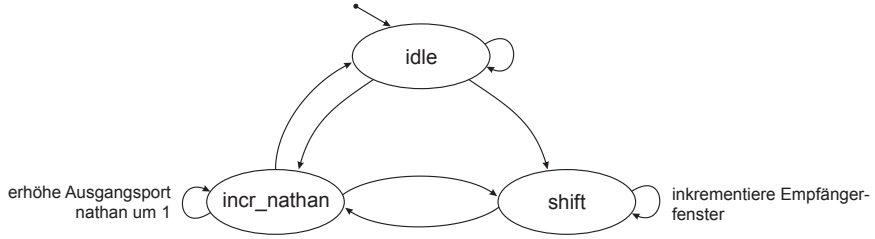
rialisiert hat und bereit ist neue Daten aufzunehmen. Wenn dieses Signal gesetzt und in `fifo_s_tx` genug freier Speicher vorhanden ist, wechselt `fsm_buf` in den Zustand `addrdata`. Außerdem wird der Zähler `position` inkrementiert und das Anforderungssignal für `fsm_shift` generiert.

addrdata Hier werden die `addr` und `data` Felder des Slow Control-Frames für die verschiedenen Slow Control-Stationen aus `bram_s_tx` gelesen. Die Reihenfolge der Lesezugriffe stellt sich dabei wie folgt dar: jedes Mal wenn sich `fsm_buf` in diesem Zustand befindet greift sie auf eine festgelegte Byte-Position innerhalb der in `bram_s_tx` gespeicherten Slow Control-Befehle zu, und liest somit für alle Stationen korrespondierende Bytes. So werden z.B. die Bits 31–24 des `addr` Feldes ausgelesen, wenn die FSM zum ersten Mal in diesen Zustand geht, beim zweiten Eintreten in den Zustand die Bits 23–16 usw. Welches Byte der Slow Control-Befehle jeweils ausgelesen wird ist durch den Zähler `position` festgelegt.

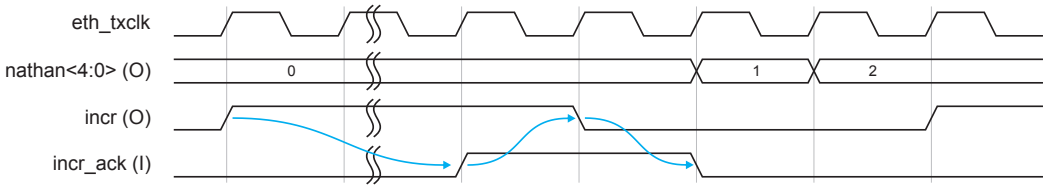
Da die Blockadressen der STP-Pakete in `bram_s_tx` nicht intern gespeichert werden, müssen sie erneut von `nathanbuffers` abgerufen werden. Jedoch muss dies nun nicht mehr für *alle* Slow Control-Stationen erfolgen, da ja in dem internen Register `valid_reg` bereits gespeichert ist, für welche Stationen gültige Pakete vorliegen. Um die Anzahl unnötiger Zugriffe zu reduzieren und gleichzeitig den Logikaufwand möglichst gering zu halten wurde folgendes Konzept realisiert: Die 16 Nathan-Module sind logisch auf vier Blöcke mit jeweils vier Modulen aufgeteilt. Ein fünfter Block repräsentiert den Balu-FPGA. Für jeden dieser Blöcke existiert ein Register `block_valid`, welches `'1'` ist falls mindestens für eine Station innerhalb dieses Blocks gültige Pakete vorliegen (wired-or Verknüpfung). `fsm_buf` fragt nun jeweils die Blockadresse für das erste Nathan-Modul in jedem Block von `nathanbuffers` ab und greift auf den entsprechenden Bereich in `bram_s_tx` zu. Zugleich entscheidet sie anhand des `block_valid` Flags, ob direkt zum Beginn des nächsten Blocks gesprungen werden kann oder ob für alle Stationen in diesem Block Daten gelesen werden sollen. Diese Implementierung hat zur Folge, dass für den Fall dass nur für ein einziges Nathan-Modul gültige Daten vorliegen keine 17 Takte benötigt werden um die gesamte Shiftregister-Matrix zu laden, sondern nur acht: für den Block, in dem sich das Nathan-Modul mit den gültigen Daten befindet, werden vier Leseadressen berechnet, für alle anderen Blöcke wird jeweils nur für die erste Station im Block eine Leseadresse berechnet und dann direkt zum Beginn des nächsten Blocks gesprungen (vier ungenutzte Blöcke, also vier Lesezugriffe). Diese Implementierung berechnet somit auch Leseadressen für Stationen, für die keine gültigen STP-Pakete in `bram_s_tx` gespeichert sind. Die Reset-Leitung des letzten Addierers in der Pipeline zur Adressberechnung muss benutzt werden um ungültige Daten auszufiltern (siehe Abschn. 3.1.3). Prinzipiell wäre es möglich gewesen ein Implementierung zu wählen, die es erlaubt direkt nur für jene Stationen Daten aus `bram_s_tx` zu lesen, für die gültige Daten vorliegen. Jedoch wäre dafür der Logikaufwand größer. Außerdem wird in Abschnitt 5.1.1.2 gezeigt, dass die gewählte Implementierung für die Gesamtperformance des Systems vollkommen ausreichend ist, und dass aufwendigere Mechanismen keine Vorteile bringen würden.

Nachdem Daten für die Slow Control-Station im Balu-FPGA aus `bram_s_tx` gelesen wurden, wechselt `fsm_buf` in den Zustand `wait_s` oder, falls der Zeiger `position` angibt, dass soeben das letzte Byte für alle Slow Control-Kommandos ausgelesen wurde, in den Zustand `incr_cmd_cnt`.

incr_cmd_cnt Hier werden die Zähler `nb_cmd_counters` für jene Slow Control-Stationen inkrementiert, für die ein gültiges Kommando gelesen wurde, für die also



(a) Zustandsgraph



(b) zeitliches Verhalten

Abbildung 3.7: Steuerung der Inkrementierung der Empfängerfenster. Nachdem das STP-Paket im aktuellen Fenster vollständig ausgelesen wurde, muss das Empfängerfensters um eine Position inkrementiert werden. Die FSM `fsm_shift_nb` setzt hierzu die entsprechende Nathannummer und aktiviert das Signal `incr` solange, bis `nathanbuffers` das erfolgreiche Verschieben des Fensters durch das Signal `incr_ack` bestätigt. Danach wird die Nathannummer inkrementiert. In dem hier dargestellten, beispielhaften Verlauf wird das Fenster für die Nathan-Module 0 und 2 inkrementiert, Modul 1 wird übergangen.

der Eintrag im Statusregister `valid_reg` gesetzt ist. Falls ein Eintrag in `done_reg` gesetzt ist wird der entsprechende Zähler auf Null zurückgesetzt, da in diesem Fall alle Slow Control-Befehle aus dem aktuellen Speicherblock in `bram_s_tx` vollständig ausgelesen wurden.

Wenn mindestens ein Eintrag in `done_reg` gesetzt ist, wechselt `fsm_buf` nach `shift_nb` um das Empfängerfenster für inkrementieren, ansonsten in den Zustand `idle`.

shift_nb Dieser Zustand kann als Meta-Zustand betrachtet werden, d.h. hier wird das wake-up-event für den untergeordneten Automaten `fsm_shift_nb` generiert, dessen Ablauf dadurch gekapselt wird. Die FSM `fsm_shift_nb` ist verantwortlich für das Inkrementieren des Empfänger-Fensters bzgl. des Sliding-Window-Algorithmus (siehe Abschn. 2.3.5). Das zugehörige Zustandsdiagramm ist in Abbildung 3.7(a) dargestellt.

Nachdem `fsm_shift_nb` den Ruhezustand verlässt ist der Port `nathan` zunächst `0x00`. Je nachdem ob für das Nathan-Modul auf Sockel 0 das Empfängerfenster inkrementiert werden soll oder nicht erfolgt ein Wechsel in den Zustand `shift` bzw. `incr_nathan`. Im Zustand `shift` wird das zeitliche Verhalten aus Abbildung 3.7(b) realisiert, d.h. `fsm_shift_nb` aktiviert den Port `incr`, der mit dem Modul `nathanbuffers` verbunden ist, und bleibt in diesem Zustand bis sie von `nathanbuffers` das `incr_ack` erhält (Handshake). Nun weiß `nathanbuffers`, dass alle Slow Control-Kommandos des aktuellen STP-Pakets für dieses Nathan-Modul gelesen wurden und kann die Blockadresse für das nächste gültige STP-Paket bereitstellen. Danach erfolgt der Übergang in den Zustand `incr_nathan`. Dort wird

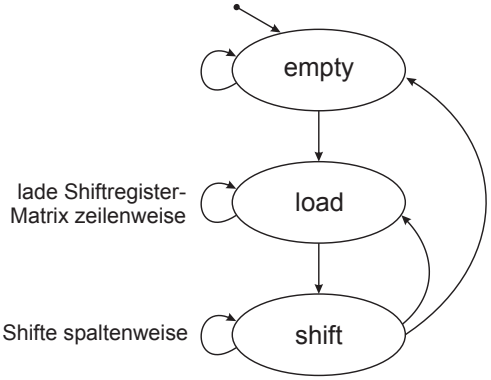


Abbildung 3.8: Zustandsgraph der FSM `fsm_shift`. Diese FSM steuert das Laden und Shiften der Matrix.

der Port `nathan` jeweils um eins erhöht und anhand der Einträge im Statusregister `done_reg` entschieden, ob in den Zustand `shift` gewechselt werden muss oder nicht. Ist `fsm_shift_nb` bei der letzten Slow Control-Station angekommen, bestätigt sie `fsm_buf` das erfolgreiche Inkrementieren der Empfängerfenster und kehrt zurück nach `idle`. Mit dem Erhalt der Bestätigung kehrt auch `fsm_buf` in den Ruhezustand zurück.

3.1.6 fsm_shift

Diese FSM ist verantwortlich für das korrekte Laden und Shiften der Shiftregister-Matrix. Außerdem muss sie das write enable für `fifo_s_tx` setzen. Der Zustandsgraph ist in Abbildung 3.8 dargestellt. Die Shiftregister-Matrix selbst besitzt neben den Daten-, Takt- und Reset-Ports auch einen Port `shift_en` sowie einen 17 Bit breiten Port `din_en`, der festlegt welche Zeile der Matrix mit den Daten an ihrem Eingang beschrieben werden soll. Diese Ports werden von der FSM `fsm_shift` angesteuert.

empty In diesem Zustand ist die Matrix vollkommen leer und statisch. Das Signal `shima_rdy` wird aktiviert und damit der FSM `fsm_buf` signalisiert, dass die Shiftregister-Matrix bereit ist neue Daten aufzunehmen. Falls `fsm_buf` ihrerseits nun den Ruhezustand verlässt setzt sie ein Anforderungssignal und `fsm_shift` wechselt daraufhin in den Zustand `load`.

load Hier wird das Signal `din_en` zunächst so gesetzt, dass an der Shiftregister-Matrix ankommende Daten in die erste Zeile geschrieben werden. Die weitere Steuerung des `din_en`-Vektors muss von `fsm_buf` übernommen werden. Dies ist aufgrund der logischen Aufteilung der Nathan-Module auf mehrere Blöcke notwendig, da nur `fsm_buf` entscheiden kann, ob ein Block übersprungen werden kann oder nicht. Es existieren zwei Signale zwischen den FSMs `fsm_buf` und `fsm_shift`, die den Vektor `din_en` modifizieren. Zum einen das Signal `din_en_incr`, welches `din_en` um eine Position verschiebt, und somit einen Schreibzugriff auf die nächste Zeile ermöglicht. Zum anderen `din_en_skip`, wodurch an den Anfang des nächsten logischen Nathan-Blocks gesprungen wird. Es können somit beim Laden der Matrix gezielt Zeilen übersprungen werden. Nachdem die letzte Zeile geladen wurde wechselt `fsm_shift` in den Zustand `shift`.

shift Die gesamte Shiftregister-Matrix wird nun spaltenweise geshiftet, indem das Signal `shift_en` aktiviert wird. Zugleich muss `fifo_s_tx_wr_en` gesetzt werden, da der Datenausgang der Shiftregister-Matrix direkt mit dem Eingang von `fifo_s_tx` verbunden ist.

Auf Seite 36 wurde darauf hingewiesen, dass durch das Pipeline-Design der Adressberechnung eine Anlauflatenz entsteht. Um die Auswirkungen dieser Latenz auf den weiteren Datenpfad zu reduzieren, bietet es sich an das Bereitschaftssignal `shima_rdy` nicht erst wenn die Shiftregister-Matrix bereits leer ist zu aktivieren, sondern bereits einige Takte zuvor. Nachdem `fsm_buf` das Bereitschaftssignal von `fsm_shift` erhält vergehen acht Takte, bis gültige Daten am Eingang der Shiftregister-Matrix anliegen: ein Takt wird für den Zustandswechsel in `fsm_buf` benötigt, zwei Takte Verzögerung entstehen aufgrund des zeitlichen Verhaltens des Moduls `nathanbuffers` und weitere fünf Takte aufgrund der Adressberechnung und des synchronen BRAM-Zugriffs. Da `fsm_shift` exakt acht Takte zum Shiften der Matrix benötigt, kann somit also das Bereitschaftsflag bereits gesetzt werden sobald `fsm_shift` in den Zustand `shift` wechselt. Durch diese Vorgehensweise lässt sich der Durchsatz des Systems signifikant verbessern und unnötige Wartepausen werden vermieden. Falls die FSM `fsm_buf` anzeigt, dass weitere Daten in `bram_s_tx` bereitstehen, wechselt `fsm_shift` direkt wieder in den Zustand `load`, ansonsten nach `empty`.

3.1.7 fifo_s_tx

Dieser FIFO dient zum einen dem Puffern von Teilen der Slow Control-Frames. Wenn das Modul `eth2sctrl_mac_top` einmal damit begonnen hat Slow Control-Frames zu senden, so kann dieser Vorgang nicht unterbrochen werden, ohne dass bei den Empfänger-Stationen Prüfsummen-Fehler detektiert werden. Es muss somit garantiert werden, dass genügend Daten in `fifo_s_tx` hinterlegt sind wenn mit dem Aussenden von Frames begonnen wird. Des Weiteren ermöglicht `fifo_s_tx` einen Übergang zwischen zwei unterschiedlichen Clock-Domänen, d.h. Schreib- und Lesezugriffe können mit unterschiedlichen Taktfrequenzen erfolgen. Das Modul `shifters_s_tx` arbeitet im FPGA mit der PHY-Frequenz von 125 MHz während `eth2sctrl_mac_top` mit einer Frequenz von 156,25 MHz arbeitet, was der Oszillatorfrequenz auf der Backplane entspricht (siehe Abschn. 2.1.2).

FIFOs lassen sich mit Hilfe des Xilinx LogiCORE Core-Generators in den VHDL-Code integrieren [33]. Der Coregenerator bietet auch die Möglichkeit verschiedene Statussignale zu implementieren. Für `fifo_s_tx` werden beispielsweise programmierbare empty und full Flags benötigt, d.h. Flags, die das Über- bzw. Unterschreiten eines wählbaren Schwellwerts anzeigen. Das `prog_full` Flag wird benutzt um sicherzustellen, dass kein Überlauf stattfindet. Dieses Statussignal ist mit `fsm_buf` verbunden. Die FSM `fsm_buf` darf den Ruhe- bzw. Wartezustand nur verlassen, wenn genügend freier Speicherplatz in `fifo_s_tx` vorhanden ist.

Das `prog_empty` Flag wird benutzt um dem Modul `eth2sctrl_mac_top` zu signalisieren, dass genug Daten in `fifo_s_tx` sind um Slow Control-Frames aussenden zu können. Es ist jedoch nicht erforderlich die für die Frames benötigten Daten vollständig bereitzuhalten, da die zeitliche Abfolge der Zustände in den FSMs `fsm_buf` und `fsm_shift` deterministisch und somit der mittlere Datenzufluss zu `fifo_s_tx` berechenbar ist. Die Berechnung des Schwellwerts für das `prog_empty` Flag soll hier plausibel gemacht werden. Es handelt sich hierbei jedoch lediglich um eine Überschlagsrechnung bei der mittlere Schreib- und Leseraten angenommen werden, d.h.

die genaue Lage und Länge der Zugriffspausen wird nicht berücksichtigt. Der im Folgenden berechnete Wert liefert somit nur eine Abschätzung in welchem Bereich der tatsächliche Wert liegen wird. Der exakte Wert muss schließlich mit Hilfe der funktionalen Simulation gefunden werden.
Die Anzahl geschriebener Datenworte $w(n)$ ist abhängig davon für wie viele und – aufgrund der Aufteilung der Nathan-Module auf logische Blöcke – für welche Stationen Slow Control-Befehle versendet werden sollen. Um unter allen Umständen einen korrekten Ablauf der Hardware garantieren zu können muss hierfür die niedrigste zu erwartende Schreibrate betrachtet werden. Diese wird erreicht, wenn die Shiftregister-Matrix vollständig geladen werden muss, `fsm_shift` also 17 Takte im Zustand `load` verweilt. Dies hat zur Folge, dass in nur acht von 25 Takten Daten in `fifo_s_tx` geschrieben werden. Für die Anzahl der Schreibzugriffe ergibt sich somit

$$w(n_w) = \frac{8}{25} n_w.$$

Dabei bezeichnet n_w die Anzahl der Takte in der Schreib-Domäne. Die Anzahl der Lesezugriffe $r(n)$ auf den FIFO ist gegeben durch

$$r(n_r) = \frac{72}{90} n_r,$$

wobei n_r die Anzahl der Takte in der Lese-Domäne darstellt. Die Formel wird verständlich, wenn man das Slow Control-Frame betrachtet (siehe S. 22) und sich klarmacht, dass nur die Felder `cmd`, `mod`, `addr` und `data` in `fifo_s_tx` abgelegt werden. Dies entspricht 72 Bit pro Frame. Die Länge des Frames bis zum Ende des `data` Feldes beträgt 90 Bit. Somit wird also in den 90 Takten nach Beginn des Aussendens von Frames 72 Mal lesend auf `fifo_s_tx` zugegriffen. Der Füllstand $f(n_r)$ ergibt sich nun als Differenz der Schreib- und Lesezugriffe, korrigiert um den anfänglichen Füllstand f_0:

$$\begin{aligned} f(n_r) &= w(n_w) - r(n_r) + f_0 \\ f(n_r) &= \frac{8}{25} n_w - \frac{72}{90} n_r + f_0 \end{aligned}$$

Ziel ist es nun für den anfänglichen Füllstand f_0 den minimalen Wert zu finden, so dass garantiert kein Unterlauf auftritt. Da die Lesezugriffe auf den FIFO mit einer höheren Frequenz erfolgen als die Schreibzugriffe muss insbesondere am Ende eines Lesezyklus, also 90 Takte nach dem Beginn des Aussendens von Frames, der Füllstand nicht-negativ sein:

$$f(90) \geq 0.$$

Um die Gleichung lösen zu können muss noch n_w durch n_r ausgedrückt werden. Mit einer Schreibfrequenz von 125 MHz und einer Lesefrequenz von 156,25 MHz folgt $n_w = 125/156,25 \cdot n_r$. Man erhält damit

$$f_0 \geq 49.$$

Dieser Wert wurde in der Simulation verifiziert.

Die Tiefe von `fifo_s_tx` wurde auf 128 Worte festgesetzt, ein größerer FIFO bringt keinen Performancezuwachs. Dies wird bei der Diskussion in Abschnitt 5.1.1.2 klar werden. Eine Auflistung aller Kenngrößen dieses FIFOs findet sich in Anhang A.

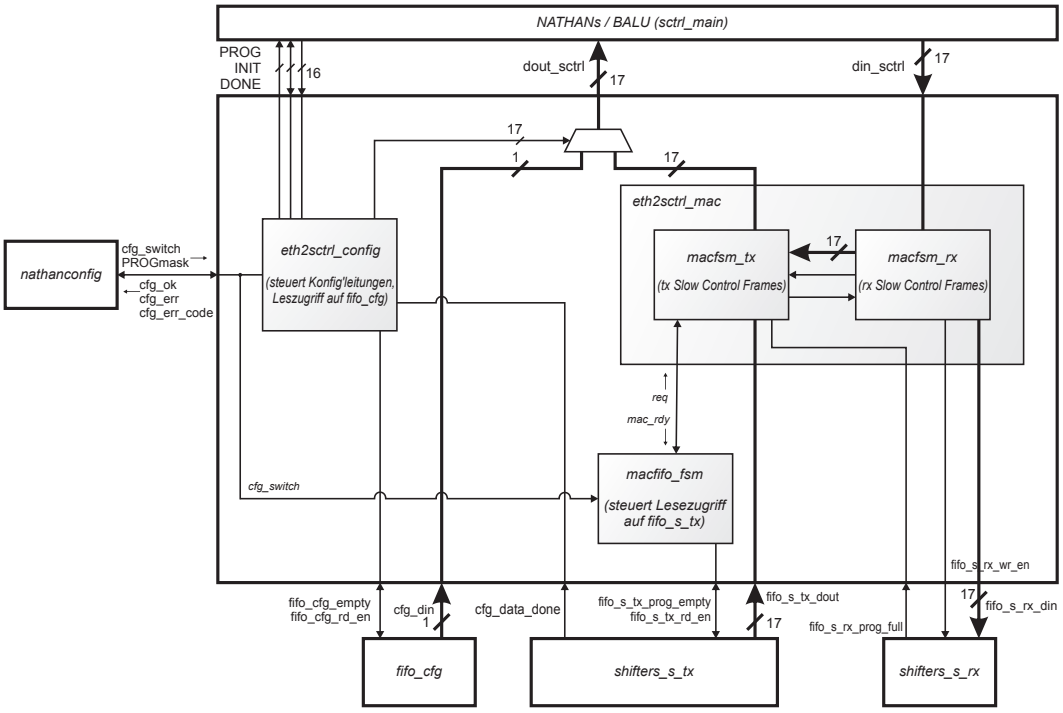
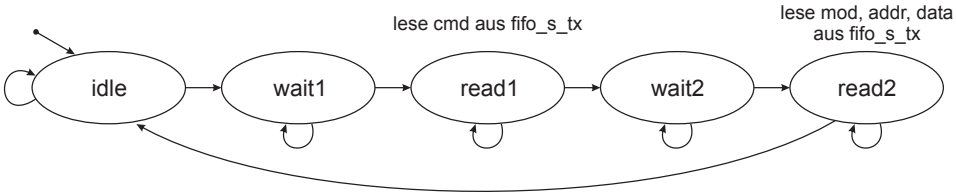


Abbildung 3.9: Blockschaltbild des Moduls `eth2sctrl_mac_top`, welches die Schnittstelle zu den Slow Control-Stationen auf den Nathan-FPGAs und im Balu-FPGA darstellt.

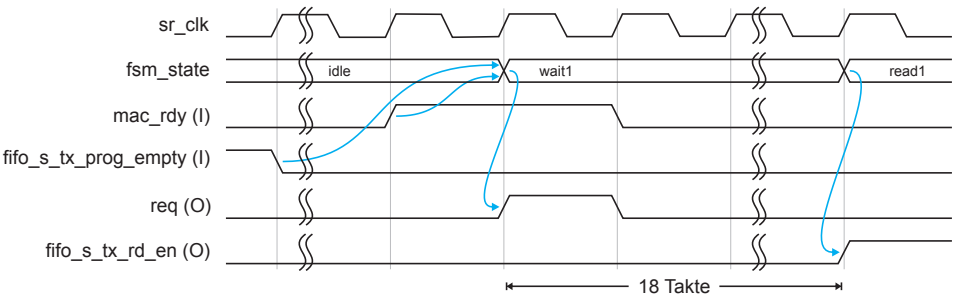
3.2 Interface zu den Slow Control-Stationen: eth2sctrl_mac_top

Das Modul `eth2sctrl_mac_top`, im Folgenden auch einfach als MAC bezeichnet, stellt die Implementierung des Slow Control-Interface zu den Nathan-Modulen dar. Es sei darauf hingewiesen, dass die Bezeichnung als MAC nicht ganz korrekt ist, da das Modul sowohl die MAC- als auch die LLC-Subschicht der Sicherungsschicht im OSI-Referenzmodell darstellt (siehe Abschn. 1.2.2). Somit ist `eth2sctrl_mac_top` verantwortlich für die Arbitrierung des physikalischen Übertragungsmediums, das Erstellen und Analysieren von Frames sowie die Realisierung von Methoden zur Erkennung von Übertragungsfehlern. Im speziellen Fall der Slow Control wird letzteres durch eine CRC-Prüfsumme verwirklicht, die sich über Header und Payload des Slow Control-Frames erstreckt (siehe Abschn. 2.1.5).

Das Hauptaugenmerk bei der Konzeption und Entwicklung des MACs lag darauf den Verbrauch an Hardwareressourcen möglichst gering zu halten. Es war darauf zu achten, möglichst wenig redundante Strukturen zu erzeugen. Das bedeutet, dass wenn möglich Zähler, Multiplexer etc. nur einmal für alle Slow Control-Stationen anstatt für jede Station einzeln implementiert werden. Da die ursprünglich vorhandene Ringtopologie der Slow Control ersetzt wurde durch eine Sterntopologie (siehe S. 26) wurde es möglich das Senden und Empfangen von Frames zu Parallelisieren, d.h. die Kommunikation mit den Slow Control-Stationen so zu gestalten, dass diese immer alle gleichzeitig Frames erhalten und dass auch die Sendeberechtigung in Form des Tokens gleichzeitig an alle Stationen erteilt wird. Kollisionen der Datenframes unterschiedlicher Stationen sind von vornherein ausgeschlossen. Diese Parallelisierung stellt das wesentliche Konzept zur Einsparung von Ressourcen dar.



(a) Zustandsgraph



(b) Zeitliches Verhalten

Abbildung 3.10: Zustandsdiagramm und Zeitverhalten der FSM `macfifo_fsm`. Hiermit wird die Schnittstelle zwischen den Modulen `eth2sctrl_mac_top` und `shifters_s_tx` realisiert. Sie besteht aus dem programmierbaren empty Flag von `fifo_s_tx` sowie einem read enable-Signal und dem entsprechenden Datensignal, welches sich synchron zum Takt der Lesedomäne (`sr_clk`) verhält.

Eine Übersicht über die einzelnen Submodule von `eth2sctrl_mac_top` ist ist Abbildung 3.9 dargestellt. Er besteht aus dem eigentlichen MAC `eth2sctrl_mac`, der FSM `eth2sctrl_config`, welche notwendig ist um dem Konfigurationsprozess via Ethernet zu realisieren sowie einer FSM die den Lesezugriff auf `fifo_s_tx` steuert (`macfifo_fsm`). Das Submodul `eth2sctrl_mac` besteht seinerseits wiederum aus einem Submodul für das Senden und einem für das Empfangen von Frames. An der Schnittstelle zu den Slow Control-Stationen befindet sich ein Multiplexer, der es erlaubt zwischen dem MAC und Konfigurationsdaten aus `fifo_cfg` als Datenquelle zu wählen. Seine Ansteuerung wird in Kapitel 4 dargelegt. Weitere Schnittstellen bestehen zu den Modulen `shifters_s_tx` und `shifters_s_rx`, welche der Serialisierung bzw. Deserialisierung der Slow Control-Daten dienen, zu `fifo_cfg` welcher Daten für den Konfigurationsprozess zwischenspeichert sowie zu `nathanconfig`. Bei letzterem handelt es sich um ein Slow Control-Modul im Balu-FPGA, über welches im Wesentlichen der Konfigurationsvorgang gesteuert wird. Die folgenden Abschnitte bieten einen detaillierten Überblick über die Implementierung der einzelnen Module; die FSM `eth2sctrl_config` wird im nächsten Kapitel erläutert.

3.2.1 macfifo_fsm

Dieses Modul regelt den Lesezugriff auf `fifo_s_tx`. In Abbildung 3.10 ist das Zustandsdiagramm, welches aus einer Abfolge von Lese- und Wartezuständen besteht sowie das zeitliche Verhalten der Kommunikation zwischen den Modulen `macfifo_fsm` und `eth2sctrl_mac` bzw. `shifters_s_tx` das damit realisiert wird, dargestellt.

Um den Zustand `idle` verlassen zu können muss das programmierbare empty Flag von `fifo_s_tx` deaktiviert sein, d.h. es müssen genug Daten gespeichert sein um ein unterbrechungsfreies Senden von Frames zu ermöglichen (siehe Abschn. 3.1.7). Des Weiteren darf aktuell kein Konfigurationsvorgang ablaufen (`cfg_switch='0'`) und der MAC muss durch das Bereitschaftssignal `mac_rdy` anzeigen, dass er mit dem Versenden von Frames beginnen kann. Sind diese Bedingungen erfüllt, so wechselt `macfifo_fsm` in den Zustand `wait1`. Dort wird ein Anforderungssignal gesetzt, das den MAC dazu veranlasst mit dem Versenden der Frames zu beginnen. Über einen internen Zähler kennt `macfifo_fsm` die aktuelle Position innerhalb der Frames und kann somit zum korrekten Zeitpunkt in den Zustand `read1` übergehen, wo vier Worte, also die vier Bit des `cmd` Feldes für alle 17 Slow Control-Stationen, aus `fifo_s_tx` gelesen werden. Danach muss für zwei Takte mit dem Lesen pausiert werden, da die nächsten Worte in `fifo_s_tx` das `mod` Feld enthalten, der MAC jedoch zuvor das `pad` Feld aussenden muss (siehe S. 22). Im Zustand `read2` werden schließlich die Felder `mod`, `addr` und `data` gelesen. Danach kehrt `macfifo_fsm` in den Ruhezustand zurück.

3.2.2 eth2sctrl_mac

Dieses Modul stellt die Schnittstelle zu den Slow Control-Stationen dar. Es enthält jeweils ein Submodul für das Senden bzw. Empfangen von Slow Control-Frames. Zwischen diesen beiden Modulen muss eine Datenverbindung bestehen, da ein Frame, das von einer Station aktiv ausgesendet wird, von `macfsm_rx` bestätigt und von `macfsm_tx` zur entsprechenden Station zurückgesendet werden muss.

Um sicherzustellen, dass keine Station von der Kommunikation ausgeschlossen wird, müssen in kontrollierten Abständen Tokens versendet werden [21]. Wie dies im Modul `eth2sctrl_mac` realisiert ist wird im Folgenden dargelegt.

3.2.2.1 macfsm_tx

Die Aufgaben dieses Moduls umfassen das Verpacken der Slow Control-Rohdaten aus `fifo_s_tx` in gültige Frames sowie das Aussenden von Tokens um den Slow Control-Stationen ein aktives Schreibrecht auf das Medium zu geben. Um gültige Frames erzeugen zu können muss intern auch die Berechnung der Prüfsumme stattfinden.

Die Konzeption sieht vor, Frames oder Tokens immer parallel an alle Slow Control-Stationen zu senden. Durch das parallele Versenden von Tokens an alle Stationen kann erreicht werden, dass auch die von den Stationen aktiv ausgesendeten Frames parallel eintreffen und von `macfsm_rx` simultan analysiert werden können.

Der Datenpfad durch das Modul `macfsm_tx` ist in Abbildung 3.11 dargestellt. Es existiert eine FSM die das Senden von Frames und Tokens steuert. Weiterhin wird ein Zähler für die Bitposition innerhalb der Frames benötigt. Dieser Zähler wird aufgrund des parallelen Sendens der Frames für alle Stationen gemeinsam genutzt und muss nur ein Mal implementiert werden. Außerdem wird für jede Station ein Multiplexer benötigt, der es erlaubt zwischen verschiedenen Datenquellen zu wählen. Die Datenquellen sind hierbei der Datenausgang von `fifo_s_tx`, das Signal `ackreg_dout`, welches mit `macfsm_rx` verbunden ist und zur Bestätigung von Frames dient, die intern berechnete Prüfsumme sowie diverse Konstanten die für die statischen Felder des Slow Control-Frames (`sof`, `type`, `mon`, `dest`, `src`, `pad`, `ack`) benötigt werden. Die Implementierung sieht dabei vor, dass die Stationsnummern, also das Feld `dest`, den Sockelnummern entsprechen. Die Slow Control-Station im Balu-FPGA erhält die Nummer 16. Das Feld `src` wird immer auf 31 gesetzt. Jeder Multiplexer wird

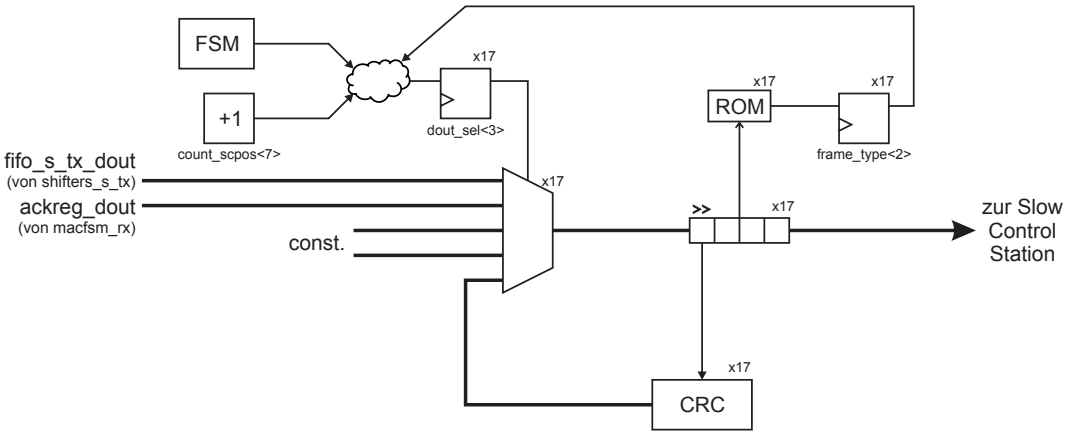


Abbildung 3.11: Datenpfad im Modul `macfsm_tx` für eine Slow Control-Station. Aufgrund der Existenz dreier verschiedener Frametypen lässt sich die mehrfache Implementierung identischer Strukturen weitestgehend nicht vermeiden.

gesteuert über ein 3 Bit breites Register `dout_sel`. Dieses wird in Abhängigkeit vom Zählerstand sowie dem Typ des Frames und dem Zustand der FSM gesetzt.

Es soll hier noch auf den Fall hingewiesen werden, dass nur für eine oder wenige Slow Control-Stationen gültige Befehle in `bram_s_tx` liegen. Das Modul `macfsm_tx` erhält seine Daten aus `fifo_s_tx` in dem die Daten aus `bram_s_tx` serialisiert gespeichert sind. Wenn für eine Station kein gültiges Kommando aus `bram_s_tx` gelesen wurde, befinden sich in der entsprechenden Spalte in `fifo_s_tx` nur Nullen. Da der MAC vor Beginn des Lesens nicht feststellen kann für welche Stationen gültige Befehle vorliegen, wird er also ein Slow Control-Frame an die entsprechende Station aussenden, dessen `cmd`-Feld `0x0` ist. Dieses Frame wird von der Station korrekt bestätigt, führt aber zu keinerlei Veränderungen an der Station, da es sich hierbei um ein nicht definiertes Kommando handelt.

In Abschnitt 2.1.5 wurde bereits erwähnt, dass es drei verschiedene Frametypen gibt, deren Unterscheidung anhand des `cmd`-Feldes möglich ist. Die Existenz dreier Frametypen führt zu einem erhöhtem Bedarf an Hardwareressourcen: zum einen muss dass `cmd`-Feld analysiert werden. Hierfür wurde am Datenausgang ein 4-Bit Schieberegister implementiert, das bei einem vorgegebenen Zählerstand ausgewertet wird. Abhängig von seinem Inhalt wird dann das Register `frame_type` gesetzt. Da drei Typen existieren, muss dieses Register zwei Bit breit sein. Sowohl das Schieberegister, als auch das Register `frame_type` muss für jede Station einmal implementiert werden. Zudem muss das Register `dout_sel` für jede Station einzeln implementiert werden, was bei der Existenz nur eines Frametyps nicht nötig gewesen wäre.

Die Berechnung der Prüfsumme erfolgt mit Hilfe einer von der Firma Easics bereitgestellten VHDL-Funktion [6]. Im Gegensatz zur Implementierung im Modul `sctrl_main` auf den Nathan-FPGAs, wo die Prüfsumme nibbleweise berechnet wird (siehe Abschn. 2.1.5), erfolgt die Berechnung hier bitweise. Als Berechnungsgrundlage dient dabei das Eingangsbit des Schieberegisters, wie in Abbildung 3.11 dargestellt.

Der Zustandsgraph der FSM ist in Abbildung 3.12 gezeigt. Die zentrale Aufgabe dieser FSM ist es in geregelten Abständen Tokens zu versenden sowie den Versand von Frames zu steuern.

idle In diesem Zustand wird ein Bereitschafts-Flag gesetzt, falls das programmierbare full Flag des FIFOs in `shifters_s_rx` (`fifo_s_rx`) deaktiviert ist. Damit wird

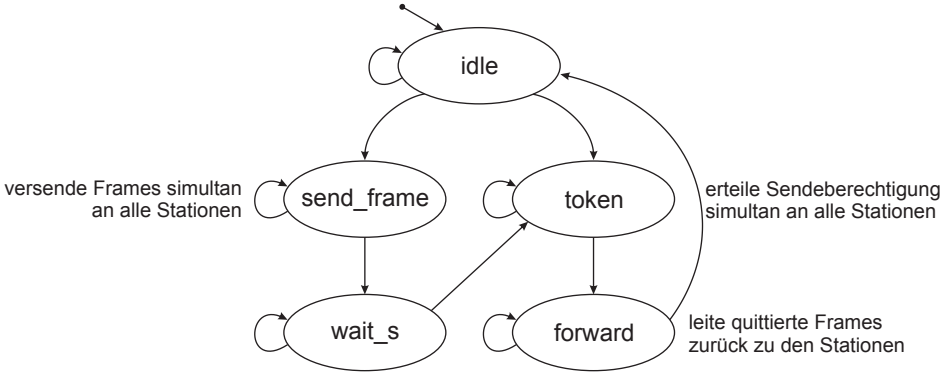


Abbildung 3.12: Zustandsgraph der FSM in `macfsm_tx`. Der Automat steuert das Aussenden von Frames und Tokens. Dabei soll nach jedem Frame sowie regelmäßig in Ruhephasen ein Token versendet werden.

`macfifo_fsm` signalisiert, dass Frames versendet werden können. Wenn die FSM daraufhin eine Anforderung von `macfifo_fsm` erhält, d.h. wenn genug Daten im `fifo_s_tx` gespeichert sind, erfolgt der Wechsel in den Zustand `send_frame`. Andernfalls wartet sie eine definierte Anzahl an Takten und wechselt danach in den Zustand `token`. Dieser Zustandswechsel ist nur möglich wenn in `fifo_s_rx` genug freier Speicher vorhanden ist. Dies ist nötig um ein Überlaufen von `fifo_s_rx` zu verhindern.

send_frame Hier wird der Zähler `count_scpos` gestartet, der die Bitposition innerhalb der Slow Control-Frames definiert. In Abhängigkeit von diesem Zähler werden dann die CRC-Kalkulatoren gestartet oder gestoppt sowie das Register `dout_sel`, welches den Multiplexer im Datenpfad steuert, gesetzt. Das bedeutet, dass der Zähler steuert, wann das start-of-frame gesendet wird, wann die Daten aus `fifo_s_tx` weitergereicht werden etc. Außerdem regelt dieser Zähler wann das `frame_type` Register gesetzt werden soll. Wenn der Zähler einen Schwellwert, der durch die Länge des Slow Control-Frames festgelegt ist, überschreitet wird dieser Zustand nach `wait_s` verlassen.

wait_s Damit keine Slow Control-Station von der Kommunikation ausgeschlossen wird sieht die Konzeption vor, dass nach jedem Frame ein Token versendet wird. Zwischen dem Ende des Frames und dem Versenden des Tokens muss eine Pause eingefügt werden. Hierzu dient dieser Zustand. Die Pause ist notwendig, da sichergestellt werden muss, dass keine weiteren Frames bei `macfsm_rx` ankommen, solange dies nicht bereit ist neue Frames zu analysieren. Aufgrund des zeitlichen Verhaltens der Slow Control-Stationen einerseits und `macfsm_rx` andererseits musste die Pause auf sechs Takte festgelegt werden. Danach wechselt die FSM in den Zustand `token` sobald in `fifo_s_rx` genug freier Speicherplatz vorhanden ist.

token Dieser Zustand ähnelt `send_frame`. Lediglich muss hier das `typ`-Bit des Frames auf `'1'` gesetzt und die Übertragung danach abgebrochen werden. Falls eine Station das Token aufnimmt und ihrerseits ein Datenframe überträgt, muss dieses von `eth2sctrl_mac` bestätigt werden. Hierzu müssen die beiden `ack`-Bits von `macfsm_rx` invertiert werden und das Frame durch `macfsm_tx` an die betroffene Station zurückgesendet werden. Dies wird im Zustand `forward` realisiert. Da die FSM zum Zeitpunkt des Sendens der Tokens nicht feststellen kann, ob eine der Slow Control-

Stationen das Token aufgreift, muss sie nach diesem Zustand immer in den Zustand `forward` wechseln.

forward Die Schnittstelle zu `macfsm_rx` besteht aus drei Signalen. `ackreg_dout` ist der Datenausgang eines Schieberegisters in `macfsm_rx`; die hier ankommenden Slow Control-Frames müssen ohne weitere Manipulation zu den entsprechende Stationen weitergeleitet werden. Das Signal `ackreg_en` dient der Selektion für welche Stationen der Dateneingang weitergeleitet werden soll und für welche nicht. Dieses Signal ist aus folgendem Grund notwendig: Die Konzeption des gesamten MAC sieht vor, dass Frames immer parallel versendet werden und auch parallel im MAC ankommen. Letzteres wird realisiert indem stets parallel an alle Stationen jeweils ein Token gesendet wird. Nimmt man nun jedoch an, dass von diesen parallel versendeten Tokens eines von einer Station aufgenommen wird und die anderen unverändert zum MAC zurücklaufen, so ergibt sich die Situation, dass am Datenausgang des Schieberegisters in `macfsm_rx` (siehe Abschn. 3.2.2.2) für eine Station ein gültiges, bestätigtes Frame erscheint, für alle anderen jedoch das unveränderte Token. Würde `macfsm_tx` nun den Datenausgang des Schieberegisters vollständig, d.h. an alle Stationen, weiterleiten, so würde bei jenen Stationen, die das erste Token nicht aufgenommen haben mit einigen Takten Verzögerung ein weiteres am Dateneingang erscheinen. Wird nun dieses Token aufgenommen und durch ein gültiges Frame ersetzt, so würde dieses Frame verloren gehen, da es zu einem Zeitpunkt bei `macfsm_rx` ankommt, an dem bereits andere Frames analysiert werden und somit das neu ankommende Frame verworfen werden muss. Somit war es notwendig ein enable Signal zu implementieren, welches von `macfsm_rx` gesetzt wird und verhindert, das Tokens mehrfach umlaufen.

Die FSM verlässt diesen Zustand nachdem die bestätigten Frames vollständig versendet wurden. Dies wird mit dem Zähler `count_scpos` gesteuert. Sie setzt dann auch das Signal `ackreg_en_rst` um die eben diskutierten enable Signale in `macfsm_rx` zu löschen. Außerdem wurde eine Möglichkeit implementiert den Zustand vorzeitig zu verlassen, falls keine der angeschlossenen Stationen ein Token aufnimmt, d.h. falls `ackreg_en` nach einem festgelegten Zeitintervall immernoch `0x00000` ist. In jedem Fall geht die FSM in den Zustand `idle` über.

3.2.2.2 macfsm_rx

Die Aufgabe dieses Moduls besteht darin die ankommenden Frames zu analysieren, und festgelegte Bestandteile der Frames in `fifo_s_rx` abzulegen. Das Modul `shifters_s_rx` steuert schließlich die Speicherung der Daten in einem BRAM (`bram_s_rx`). Das Modul muss feststellen, ob ankommende Frames syntaktisch korrekt aufgebaut sind und ob die Prüfsummen richtig übertragen wurden. Außerdem muss anhand der `ack` Bits festgestellt werden, ob es sich um ein bestätigtes Frame handelt oder nicht.

Aufgrund der Tatsache, dass alle Slow Control-Stationen parallel mit Frames oder Tokens angesprochen werden folgt, dass auch der Empfang der Slow Control-Frames parallelisiert werden kann. Im Gegensatz zum Versenden der Frames zu den Nathan-Modulen (TX-Pfad) werden hier dadurch mehr Möglichkeiten geschaffen um Ressourcen einzusparen. Durch die internen CRC-Überprüfung bietet `macfsm_rx` zwar eine Methode um Übertragungsfehler zu erkennen, jedoch keine Möglichkeit diese zu beheben. Insbesondere ist es nicht möglich Frames erneut zu senden nachdem eine Übertragung gescheitert ist. Dies ist auch nicht nötig, da es sich bei den Verbindungen zwischen dem Balu-FPGA und den Nathan-Modulen um Punkt-zu-Punkt-

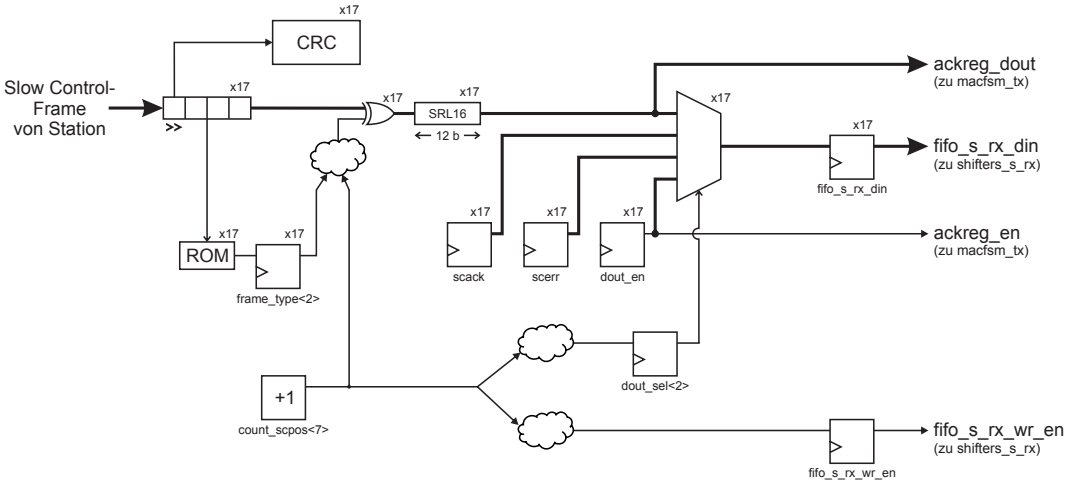


Abbildung 3.13: Darstellung des Datenpfades durch das Modul `macfsm_rx` für eine Slow Control-Station. Alle Strukturen mit Ausnahme des Zählers `count_scpos` und der Register `dout_sel` sowie `fifo_s_rx_wr_en` müssen für jede Station ein Mal instantiiert werden.

Verbindungen von relativ kurzer Länge handelt und somit Fehler durch elektrische Störeinflüsse vernachlässigbar sind. Sollte dennoch ein Übertragungsfehler auftreten, so wird ein Statusflag gesetzt, welches letzten Endes via Ethernet zum PC transferiert wird und der Software dort den Fehler anzeigt. Die Software muss dann entscheiden, wie sie den Fehler behandelt.

Bevor auf die genaue Implementierung des Moduls eingegangen wird soll das angestrebte Datenformat in `fifo_s_rx` erläutert werden. Aus dem Slow Control-Frame werden die Felder `dest`, `src`, `cmd`, `mod`, `addr` und `data` gespeichert. `fifo_s_rx` ist mit einer Breite von 17 Bit implementiert, so dass jedes Wort ein Bit des Slow Control-Frames für jede Station enthält. Zusätzlich soll zu jedem Slow Control-Frame noch ein Gültigkeits-Flag in `fifo_s_rx` gespeichert werden, welches angibt ob ein Frame vom Modul `shifters_s_rx` weiterverarbeitet werden soll. Dies ist aus folgendem Grund notwendig: auf Seite 47 wurde darauf hingewiesen, dass an alle Stationen, für die keine gültigen Slow Control-Befehle aus `bram_s_tx` gelesen wurden, Null-Kommandos gesendet werden, welche von den Stationen auch bestätigt werden. Das bedeutet, dass im realen Betrieb ständig bestätigte Null-Kommandos bei `macfsm_rx` ankommen. Diese sollen aus Effizienz- und Performancegründen nicht in `bram_s_rx` gespeichert werden. Das Gültigkeits-Flag, das hierfür gesetzt werden muss, soll in `fifo_s_rx` als erstes Bit jedes Slow Control-Frames gespeichert werde, also noch *vor* den eigentlichen Daten. Im Anschluss an die Slow Control-Felder sollen die beiden Statusflags `scack`, welches angibt ob das `ack`-Bit des empfangenen Frames gesetzt war, und `scerr`, welches Fehler in der Übertragung anzeigt, gespeichert werden. Insgesamt werden somit für jedes Slow Control-Frame 85 Bit in `fifo_s_rx` abgelegt.

Der Datenpfad und die wichtigsten Register sind in Abbildung 3.13 für eine Station dargestellt. Es existiert wiederum ein gemeinsamer Zähler `count_scpos` der für alle Stationen gemeinsam genutzt werden kann. Analog zur Vorgehensweise in `macfsm_tx` muss auch hier der Frametyp analysiert werden, was die Implementierung eines 4-Bit Schieberegisters mit parallelem Datenausgang sowie eines zwei Bit breiten Registers `frame_type` für jede Station erforderlich macht. Der Zähler wird gestartet, sobald die Bitfolge `"0110"` in einem der Eingangsschieberegister gespei-

chert ist. Diese Bitfolge entspricht den ersten drei Bit des start-of-frame, denen eine `'0'` vorausgeht. Dieses Muster wurde gewählt, da das Triggern auf das start-of-frame `"1101"` als solches ein Problem mit sich bringt: im Falle einer sich permanent auf hohem Logikpegel befindlichen Datenleitung von einem Nathan-Sockel kann durch elektrische Störeinflüsse zu einem zufälligen Zeitpunkt ein Bitmuster entstehen das dem start-of-frame entspricht. Ein solcher permanent hoher Logikpegel kann z.B. auf einem unbesetzten Sockel auftreten, wenn die Erdung des Systems nicht ausreichend ist. In einem solchen Fall könnte also zu irgendeinem Zeitpunkt zufällig das Bitmuster `"1101"` beobachtet werden, was zu einem ungewollten Anlaufen der Logik führen würde. Um diese Wahrscheinlichkeit so gering wie möglich zu halten wurde ein charakteristisches Bitmuster aus je zwei logischen Einsen und Nullen gewählt. Es sei auch darauf hingewiesen, dass dem start-of-frame aufgrund der Implementierung sowohl in `macfsm_tx` als auch in `sctrl_main` immer mindestens zwei logische Nullen vorausgehen.

Da dieses Modul auch dafür zuständig ist, die `ack`-Bits der Frames zu invertieren befindet sich am Ausgang des Schieberegisters ein XOR[5]-Gatter, dessen zweiter Eingang die meiste Zeit `'0'` ist und in Abhängigkeit des Zählers und der Frametypen für zwei Takte auf `'1'` geschaltet wird wenn die `ack`-Bits am Ausgang des Schieberegisters erscheinen.

An das XOR-Gatter schließt sich nun ein weiteres Schieberegister an, für das aber kein paralleler Datenausgang benötigt wird. Damit bietet es sich an dies mit einem SRL16 (siehe S. 20) zu implementieren. Dadurch lässt sich dieses Register mit sehr geringem Hardwareaufwand realisieren. Warum dieses Register notwendig ist wird ersichtlich wenn man sich vor Augen führt, dass das `dest`-Feld das erste ist, welches in `fifo_s_rx` geschrieben werden soll und dass *vorher* ein Gültigkeits-Flag gespeichert werden soll. Dieses Flag kann aber erst gesetzt werden *nachdem* das `cmd`-Feld analysiert und mit `0x0` verglichen wurde. Dies hat zur Folge, dass das Frame nach dem Eingangsschieberegister verzögert werden muss. Die notwendige Länge dieses Schieberegisters lässt sich aus dem Frameformat der Slow Control (siehe S. 22) bestimmen. Man erhält eine notwendige Verzögerung von zwölf Takten. Der Ausgang dieses Schieberegisters wird nun sowohl mit dem Multiplexer im Datenpfad als auch mit dem Ausgangsport `ackreg_dout` verbunden und stellt somit eine direkte Verbindung zum Modul `macfsm_tx` dar, die das Zurücksenden bestätigter Frames erlaubt.

Die Berechnung der Prüfsumme wurde wie in `macfsm_rx` bitweise implementiert. Es besteht hier jedoch ein weiterer wesentlicher Unterschied zur Implementierung in `sctrl_main`. Um Ressourcen zu sparen wurde ausgenutzt, dass es sich bei der CRC-Prüfsumme definitionsgemäß um den Rest einer Polynomdivision handelt. Das bedeutet, dass bei einer Berechnung der Prüfsumme im Empfänger über Header, Payload *und den empfangenen CRC selbst* das Ergebnis bei einer korrekt übermittelten Prüfsumme `0x00` sein muss [3]. Der große Vorteil dieser Implementierung ist, dass der empfangene CRC nicht gespeichert werden muss. Dies erlaubt ein 4-Bit Register pro Slow Control-Station einzusparen: die Prüfsumme hat eine Länge von acht Bit; vier Bit davon befinden sich zu einem definierten Zeitpunkt im Eingangsschieberegister, somit müssten weitere vier Bit in einem zusätzlichen Register gespeichert werden um einen Vergleich zu ermöglichen. Bei den knappen vorhandenen Ressourcen stellt die gewählte Implementierung eine wesentliche Einsparung dar.

[5] Exklusiv-Oder-Verknüpfung

Abschließend soll noch auf die genaue Bedeutung, Verwendung und Zuweisung der Statusregister eingegangen werden. Das Register `dout_en` wird gesetzt, wenn für die entsprechende Slow Control-Station ein korrektes start-of-frame im Eingangsregister detektiert wurde und das `typ`-Feld des Frames `'0'` ist, es sich also um eine Datenframe und kein Token handelt. Es gibt einen Sonderfall in dem der Wert des Registers wieder auf `'0'` zurückgesetzt wird, obwohl ein korrektes start-of-frame empfangen wurde. Dieser Fall tritt ein wenn das `cmd` Feld `0x0` ist. Der Grund hierfür liegt darin, dass der Inhalt dieses Registers als Gültigkeits-Flag benutzt wird, das in `fifo_s_rx` geschrieben wird. Die Aufgabe dieses Flags ist es zu markieren welche Slow Control-Kommandos in `bram_s_rx` geschrieben werden sollen. Andererseits ist es über den Port `ackreg_en` mit dem Modul `macfsm_tx` verbunden und dient somit der Kontrolle für welche Stationen bestätigte Frames weitergereicht werden sollen.
Das Register `scack` wird gesetzt, falls die beiden `ack`-Bits des empfangenen Frames `"10"` sind. In allen anderen Fällen bleibt es ungesetzt.
Um der übrigen Logik und insbesondere auch der Software am PC mitzuteilen, dass ein beschädigtes Frame empfangen wurde muss in einem solchen Fall das Register `scerr` gesetzt werden. Es gibt zwei Fälle in denen ein Frame als beschädigt deklariert wird. Einerseits falls das Ergebnis der Prüfsummenberechnung nicht Null ergibt. Außerdem wird eine Frame als fehlerhaft betrachtet, falls die beiden `ack`-Bits identisch sind, da dies laut ihrer Definition verboten ist (siehe Abschn. 2.1.5).

Da auch Frames ohne `addr`- und `data`-Feld sowie auch solche ohne `mod`-Feld existieren, muss noch diskutiert werden wie mit derartigen Frames verfahren wird. Bei der Weiterleitung der Daten an `fifo_s_rx` nimmt das Modul `macfsm_rx` keine Rücksicht auf den Frametyp; alle Frames werden am Datenausgang so behandelt, als ob alle Felder vorhanden sind. Die Folge hiervon ist, dass beispielsweise für ein kurzes Frame ohne die drei genannten Felder anstatt der Felder `mod` und `addr` die empfangenen Prüfsummenbits des Frames sowie dessen `ack`-Bits in `fifo_s_rx` abgelegt werden. Diese Artefakte werden auch von der nachfolgenden Logik nicht mehr entfernt und werden in `bram_s_rx` geschrieben und letzten Endes über die Ethernet-Verbindung zum PC gesendet. Es ist dort dann die Aufgabe der Software anhand des `cmd`-Feldes den Typ des Frames zu erkennen, und daraufhin zu entscheiden welche Felder gültig sind und welche ignoriert werden müssen. Der Grund dieser Implementierung liegt in der Einsparung von Ressourcen. In Gegensatz zu `macfsm_tx` wird hier mit der gewählten Implementierung nur ein 2-Bit Register für alle Stationen gemeinsam benötigt um den Steuereingang des Multiplexers in Abbildung 3.13 anzusteuern.

3.3 Deserialisierung: shifters_s_rx

Das Modul `eth2sctrl_mac` analysiert die von den Slow Control-Stationen kommenden Frames und legt diese in `fifo_s_rx` ab. Dieser FIFO enthält somit 17 serielle Datenströme oder anders ausgedrückt: jedes Datenwort enthält jeweils ein Bit des Slow Control-Frames von jeder Station. Prinzipiell wäre es möglich die Daten in dieser Form über das Ethernet zum PC zu senden und dort per Software zu sortieren. Dies bringt jedoch zwei wesentliche Nachteile mit sich: zum einen wäre der Softwareaufwand relativ hoch und mit den angestrebten Datenraten nicht vereinbar. Außerdem würde dies zur Folge haben, dass immer die Slow Control-Daten von allen 17 Stationen zum PC transportiert werden müssten. Dies wäre hochgradig ineffizient, insbesondere dann, wenn nur mit einer einzigen Station kommuniziert wird.

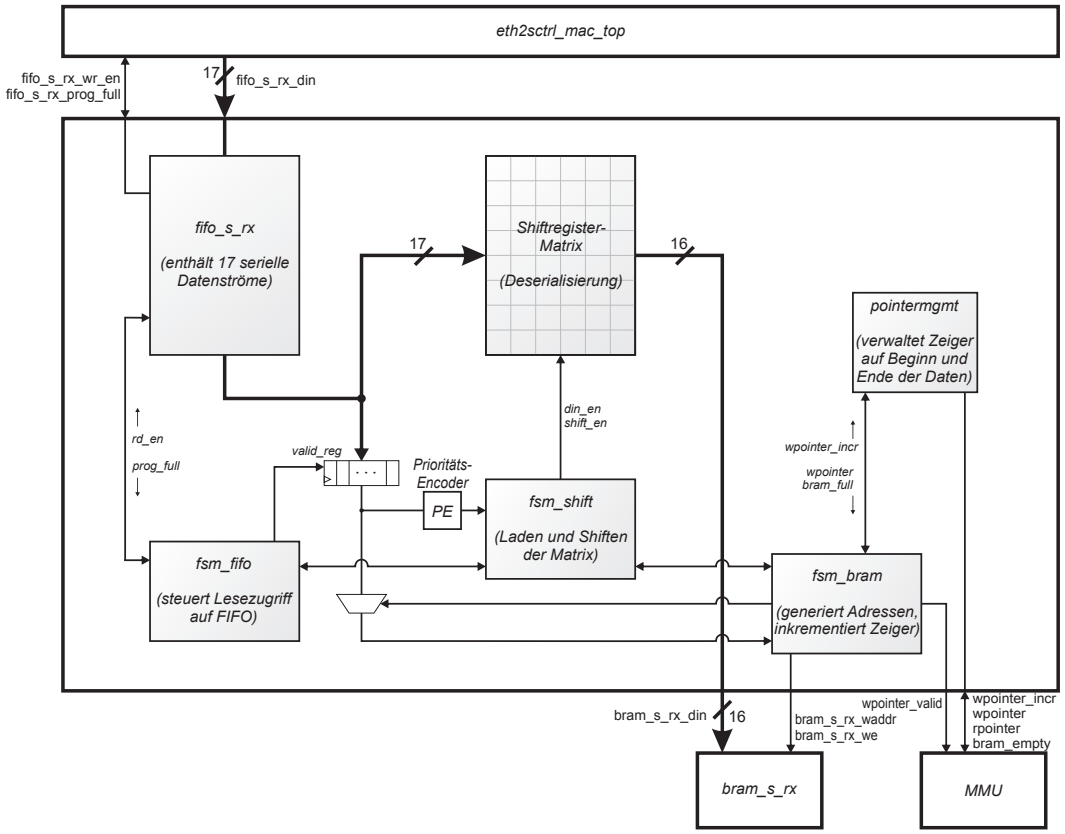


Abbildung 3.14: Blockschaltbild des Moduls `shifters_s_rx`. Dieses Modul hat die Aufgabe die Daten aus `fifo_s_rx` zu deserialisieren und in `bram_s_rx` abzulegen. Eine vollständige Liste aller Ports sowie eine Beschreibung deren Funktion findet sich in Anhang B.

Es muss somit eine Neuordnung und Selektion der empfangenen Daten bereits auf der Hardwareseite erfolgen. Diese Aufgabe wird vom Modul `shifters_s_rx` realisiert. Die Daten sollen dabei deserialisiert und in der Reihenfolge in `bram_s_rx` abgespeichert werden, wie sie in `eth2sctrl_mac` ankommen. Es werden dabei sämtliche Frames abgespeichert, die der MAC empfängt. Insbesondere bedeutet dies, dass bei Schreib-/Lesebefehlen *zwei* Frames verarbeitet und gespeichert werden, nämlich die Empfangsbestätigung des Slow Control-Kommandos und die Antwort der Station darauf, die diese aussendet sobald sie ein Token erhält (siehe Abschn. 2.1.5). Die genaue Anordnung der Daten in `bram_s_rx` wird in Abschnitt 3.3.2 diskutiert. Da die Software am PC die Reihenfolge kennt in der die Befehle für jede einzelne Station gesendet wurden ist es auch möglich die Antwortframes eindeutig zuzuordnen.

Der schematische Aufbau dieses Moduls ist in Abbildung 3.14 dargestellt. Es besteht eine Analogie zu `shifters_s_tx`, da der Deserialisierungsprozess ebenso wie die Serialisierung mittels einer Shiftregister-Matrix erfolgt. Daneben sind FSMs zur Steuerung des FIFO-Lesezugriffs, der Matrix und des Schreibzugriffs auf `bram_s_rx` in dieses Modul eingebettet. Außerdem ist das Submodul `pointermgmt` implementiert, welches Zeiger für den Schreib- und Lesezugriff auf das BRAM verwaltet sowie je ein empty und full Flag bereitstellt. Damit ist klar, dass sowohl `fsm_bram` als auch externe Logik, die auf `bram_s_rx` zugreifen will, mit diesem Submodul kommunizieren müssen. Es wurde auch ein Statusregister (`valid_reg`) implementiert in welchem die Gültigkeits-Flags, die von `macfsm_rx` in `fifo_s_rx` abgelegt wurden, gespeichert

werden. Der Zugriff auf einzelne Elemente dieses Registers ist über einen Multiplexer möglich. Des Weiteren wird über dieses Register ein Prioritätsencoder angesteuert, der an seinem Ausgang die Nummer des höchsten gesetzten Registereintrags bereitstellt. Dies war nötig um den Betrieb des Systems zu beschleunigen falls nur wenige Stationen angesprochen werden (siehe Abschn. 3.3.5).

Das Modul `shifters_s_rx` verfügt über Schnittstellen zu `eth2sctrl_mac_top`, MMU und eine Datenanbindung an `bram_s_rx`. Das Interface zum MAC besteht dabei aus dem programmierbaren full Flag von `fifo_s_rx` sowie einem Dateneingang und einem entsprechenden write enable. Die Anbindung an die MMU erfolgt über die Zeiger und Flags die vom Modul `pointermgmt` bereitgestellt werden.

3.3.1 fifo_s_rx

In diesem FIFO werden Teile der empfangenen Slow Control-Frames sowie drei Statusflags von `eth2sctrl_mac` abgelegt. Dabei enthält jedes Wort in `fifo_s_rx` korrespondierende Bits für alle Stationen. Vor den eigentlichen Slow Control-Daten wird für jede Station ein Gültigkeits-Flag gespeichert. Es wird nun dazu benutzt den Schreibzugriff auf `bram_s_rx` so zu steuern, dass nur für jene Stationen Daten geschrieben werden, von denen auch ein Frame empfangen wurde. Das Gültigkeits-Flag selbst soll dabei aber nicht in `bram_s_rx` gespeichert werden. An dieses Flag schließen sich 82 Bit des Slow Control-Frames an. Abschließend folgen die beiden Statusbits `scack` und `scerr`. In `fifo_s_rx` befinden sich somit für jedes Slow Control-Kommando 85 Bit, von denen 84 Bit in `bram_s_rx` gespeichert werden müssen.

`fifo_s_rx` ist, ähnlich wie `fifo_s_tx`, mit einem programmierbaren full Flag ausgestattet. Es wird dazu benutzt, den MAC daran zu hindern Frames oder Tokens auszusenden, wenn nicht genügend freier Speicher vorhanden ist um die Antwortframes zu speichern. Das bedeutet, dass der Schwellwert 85 Worte unter dem maximale Füllstand des FIFO liegen muss.

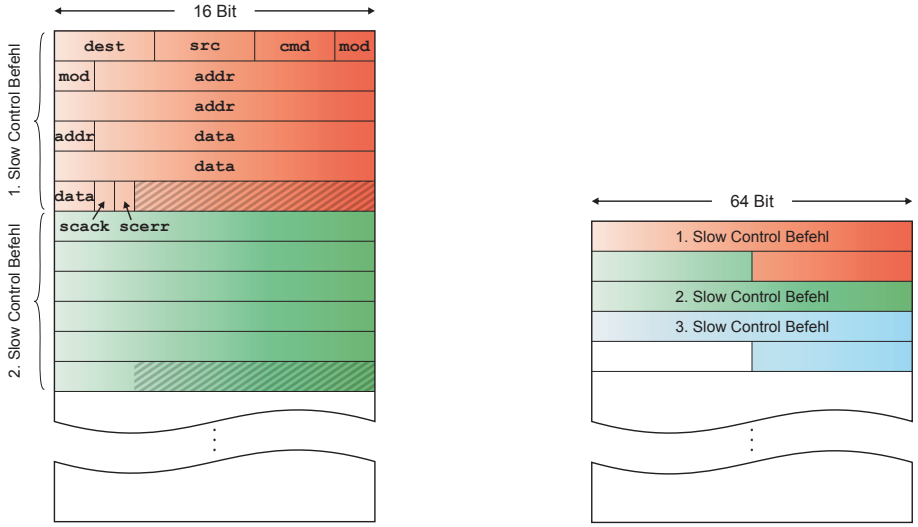
Ein programmierbares empty Flag wird hier nicht benötigt. Zur Steuerung des Lesezugriffs reicht das gewöhnliche empty Flag aus. Dies liegt zum einen daran, dass die Taktfrequenz in der Schreibdomäne deutlich höher ist als in der Lesedomäne. Zum anderen werden beim Speichern der einzelnen Felder keine großen Pausen gemacht, die man berücksichtigen müsste.

`fifo_s_rx` wurde mit einer Tiefe von 128 Worten implementiert, da dies aufgrund des notwendigen Schwellwerts die minimal mögliche Tiefe darstellt. Ein Vergrößerung des FIFOs bringt keinerlei Vorteile. Ein Überblick über alle Parameter für diesen FIFO befindet sich in Anhang A.

3.3.2 Anbindung an bram_s_rx

Um die in den folgenden Abschnitten getroffenen Aussagen nachvollziehen zu können ist es zunächst notwendig die angestrebte Anordnung der Daten in `bram_s_rx` zu verstehen. Diese soll hier dargestellt und motiviert werden.

`bram_s_rx` ist aus der Sicht der Leselogik 64 Bit breit, aus der Sicht der schreibenden Logik 16 Bit. Die Implementierung eines 16 Bit breiten Dateneingangs ergibt sich aus der Dimensionierung der Shiftregister-Matrix (siehe Abschn. 3.3.3). Die Leselogik muss beachten, dass aus ihrer Sicht die 16-Bit Segmente in einer Speicherzeile im little-endian Format angeordnet sind [32]. Die folgenden Aussagen beziehen sich, sofern nichts anderes angegeben ist, auf die Sichtweise der Schreiblogik.



(a) aus der Sicht von `shifters_s_rx` (b) aus der Sicht der MMU

Abbildung 3.15: Layout der empfangenen Daten in `bram_s_rx`. Die schraffierten Flächen links stellen ungenutzten Speicherplatz dar. Beim Lesezugriff auf `bram_s_rx` muss das little-endian Format der Daten beachtet werden.

Für jedes Slow Control-Kommando müssen 84 Bit gespeichert werden, woraus folgt, dass sechs Datenworte pro Frame benötigt werden. Eine aus Hardwaresicht einfach zu realisierende Anordnung der Daten ist eine Abspeicherung der Frames in unmittelbar aufeinanderfolgenden Blöcken zu je sechs Worten, wobei auf die genaue Bedeutung der Daten, also auch auf die Feldgrenzen im Slow Control-Frame, keinerlei Rücksicht genommen wird (siehe Abb. 3.15). Das bedeutet, dass manche Felder über Wortgrenzen hinweg verteilt gespeichert werden. Es ist dann Aufgabe der Software die entsprechenden Felder zusammenzusetzen bzw. sie voneinander zu trennen und korrekt zu interpretieren. Ebenfalls muss die Software den Frametyp erkennen und anhand dessen entscheiden welche Felder bei der Auswertung ignoriert werden müssen (siehe S. 52).

3.3.3 Shiftregister-Matrix

Da das Konzept bereits hinreichend erläutert wurde (siehe Abschn. 3.1.4) soll hier nur auf die Unterschiede zur Implementierung in `shifters_s_tx` eingegangen werden. Die Anbindung der Matrix an `fifo_s_rx` und `bram_s_rx` ist in Abbildung 3.16 dargestellt. Auch hier erfolgt zunächst ein zeilenweises Laden und anschließendes spaltenweises Shiften der Matrix. Der erste Unterschied zur Matrix im TX-Pfad liegt in der Dimensionierung der Matrix. Da der Datenausgang von `fifo_s_rx` 17 Bit breit ist und die Matrix zeilenweise beschrieben wird, muss sie 17 Spalten besitzen. Die Tiefe der Matrix ist im Prinzip beliebig wählbar. Es erscheint zunächst sinnvoll diese Matrix mit acht Zeilen zu implementieren, da hierdurch derselbe maximale theoretische Datendurchsatz ermöglicht wird wie im Modul `shifters_s_tx`. Es muss jedoch beachtet werden, dass das Datenaufkommen im Datenpfad von den Slow Control-Stationen zum Ethernet-Core (RX-Pfad) wesentlich größer ist als im TX-Pfad. Dies kann man wie folgt einsehen: durch die TX-Matrix werden pro Slow Control-Befehl 72 Bit geshiftet. Im RX-Pfad müssen jeweils 82 Bit des Frames zuzüglich der zwei Statusindikatoren `scack` und `scerr` geshiftet werden. Außerdem

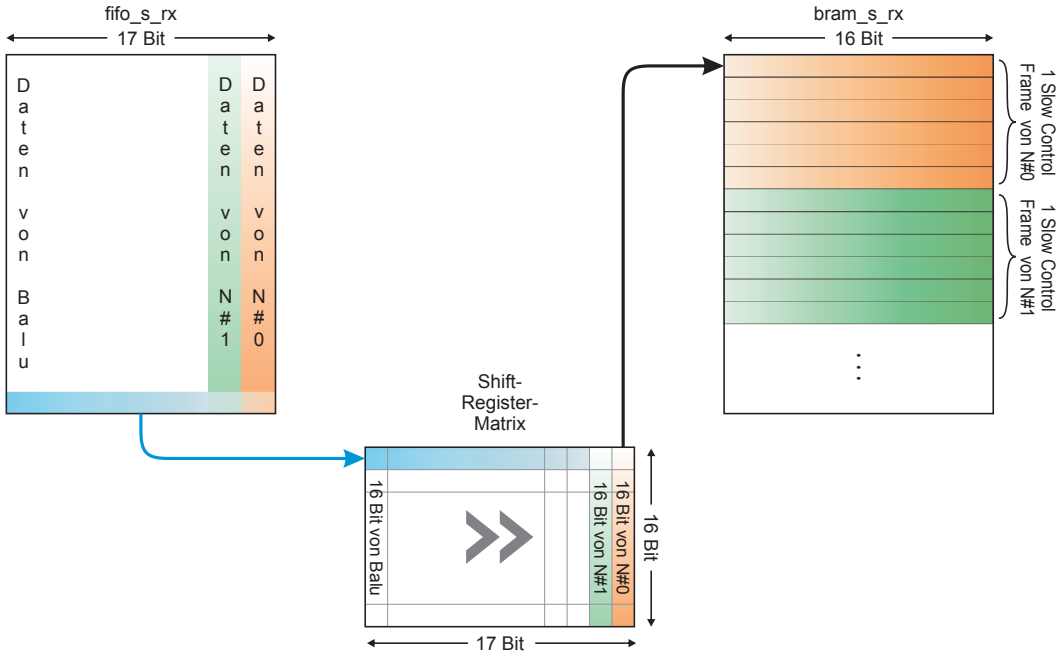


Abbildung 3.16: Schematische Darstellung der Datenanbindung der Shiftregister-Matrix an `fifo_s_rx` und `bram_s_rx`. Die Matrix wird zeilenweise beschreiben, wodurch jede Zeile korrespondierende Bits der empfangenen Slow Control-Frame für alle Stationen enthält. Das Shiften erfolgt spaltenweise, so dass am Datenausgang der Matrix nacheinander jeweils 16 Bit für die Nathan-Module 0–15 und den Balu-FPGA erscheinen.

werden z.B. im Falle eines Lesebefehls via Ethernet an eine Slow Control-Station zwei Frames von `eth2sctrl_mac_top` in `fifo_s_rx` abgelegt. Da im realen Betrieb fast alle übermittelten Befehle Schreib-/Lesekommandos sind ergibt sich somit für den RX-Pfad ein um den Faktor $2 \cdot 84/72 = 2,3$ erhöhtes Datenaufkommen. Um nun zu entscheiden ob eine Matrix mit acht Zeilen ausreicht muss die Bandbreite berücksichtigt werden, die durch die Gigabit-Ethernet-Verbindung zur Verfügung gestellt wird.

Da der Ethernet-Core im Vollduplex-Modus betrieben wird, d.h. simultan senden und empfangen kann, steht eine Bandbreite von 8 Bit/Takt für das Versenden von Frames zur Verfügung. Der Durchsatz einer voll beladenen Matrix ergibt sich aus der Speicherkapazität der Matrix dividiert durch die Anzahl der Takte, die für das Laden und anschließende Shiften der Matrix benötigt werden. Eine Matrix mit acht Zeilen ist somit in der Lage einen Durchsatz von $(17 \cdot 8)/(17+8) = 5,44$ Bit/Takt zu ermöglichen. Bei der Verwendung einer doppelt so tiefen Matrix lässt sich der Durchsatz auf $(17 \cdot 16)/(17 + 16) = 8,24$ Bit/Takt steigern. Daraus folgt, dass mit einer 16-zeiligen Matrix im RX-Pfad die Bandbreite der Gigabit-Ethernet-Verbindung voll ausgenutzt werden kann. Es wurde daher entschieden 16 Zeilen zu implementieren. Sollte sich später herausstellen, dass für die Implementierung zusätzlicher Funktionen nicht mehr genug Slices im FPGA zur Verfügung stehen, so kann die Matrix problemlos auf acht Zeilen reduziert werden. Durch den generischen Programmierstil ist eine derartige Änderung sehr zügig durchzuführen.

Ein weiterer Unterschied zur Matrix in `shifters_s_tx` ist das Layout der Daten innerhalb der Matrix. Während im TX-Pfad die Zeilen der Matrix jeweils fest einer Slow Control-Station zugeordnet waren und jede Spalte einem korrespondierenden

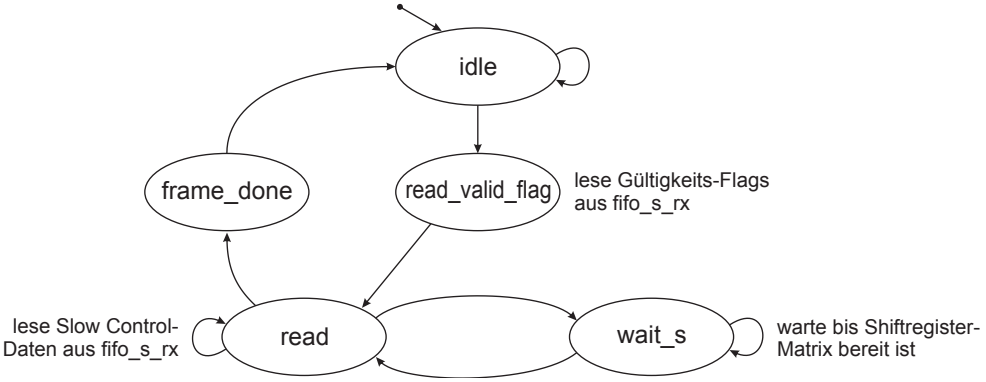


Abbildung 3.17: Zustandsgraph der FSM `fsm_fifo`. Diese FSM steuert den Lesezugriff auf `fifo_s_rx`.

Bit der Frames entsprach, sind hier die Verhältnisse vertauscht. In jeder Zeile werden korrespondierende Bits der empfangenen Frames gespeichert und jede Spalte repräsentiert eine Station (siehe Abb. 3.16). Daraus folgt, dass man durch das Shiften einer befüllten Matrix an ihrem Ausgang nacheinander in jedem Takt jeweils 16 Bit des Slow Control-Frames von aufeinanderfolgenden Stationen erhält. Diese Daten müssen dann noch in `bram_s_rx` gespeichert werden.

3.3.4 fsm_fifo

Das Modul `fsm_fifo` dient dazu das read enable Signal für `fifo_s_rx` zu erzeugen. Das erste Bit, welches *vor* den eigentlichen Slow Control-Befehlen in `fifo_s_rx` gespeichert wurde ist das Gültigkeits-Flag. Dieses muss im Statusregister `valid_reg` gespeichert und von `fsm_bram` benutzt werden um nur gültige Kommandos in `bram_s_rx` zu schreiben und unnötige Datentransfers zu vermeiden. Das Flag soll nicht über die Shiftregister-Matrix deserialisiert werden. Anschließend folgen 82 Bit des Slow Control-Frames und abschließend die beiden Statusbits, welche über die Matrix deserialisiert werden sollen. Ausgehend hiervon lässt sich die FSM für den FIFO-Lesezugriff konzipieren und implementieren. Die Zustände dieses Automaten und deren Zusammenhänge sind in Abbildung 3.17 dargestellt und werden nun diskutiert. Zur Steuerung sind zwei Zähler implementiert: `count_rdstate` zählt die Anzahl der Zustandswechsel von `wait_s` in den Zustand `read`, während `count_rdcycle` die Anzahl der Takte in eben diesem Zustand zählt.

idle In diesem Zustand werden die beiden internen Zähler auf Null zurückgesetzt. `fsm_fifo` bleibt in diesem Zustand bis das empty Flag von `fifo_s_rx` deaktiviert wird und `fsm_shift` Bereitschaft zur Aufnahme von Daten signalisiert. Sind diese Voraussetzungen erfüllt, so wird das read enable Signal für `fifo_s_rx` gesetzt und `fsm_fifo` geht in den Zustand `read_valid_flag` über.

read_valid_flag Am Datenausgang von `fifo_s_rx` liegen nun gültige Daten an, nämlich die Gültigkeits-Flags für sämtliche Slow Control-Stationen. Diese werden in das Statusregister `valid_reg` geschrieben. Danach wird das Register nicht mehr verändert bis die Frames vollständig in `bram_s_rx` abgelegt wurden. Des Weiteren wird in diesem Zustand `fsm_shift` mitgeteilt, dass im nächsten Takt Slow Control-Daten in die Shiftregister-Matrix geschrieben werden müssen. Der Zähler `count_rdcycle` wird gestartet und `fsm_fifo` wechselt nach einem Takt in den Zustand `read`.

read In diesem Zustand werden die Daten zur weiteren Verarbeitung durch die Shiftregister-Matrix aus `fifo_s_rx` gelesen. Es muss auch ein Anforderungssignal für `fsm_shift` gesetzt werden. Das Verlassen dieses Zustands wird gesteuert durch die beiden Zähler `count_rdstate` und `count_rdcycle`. Im Normalfall verweilt `fsm_fifo` für 16 Takte in diesem Zustand. Dies entspricht der Anzahl der Takte die nötig sind um alle Zeilen der Matrix nacheinander zu laden. Es besteht jedoch auch die Möglichkeit, den Zustand bereits früher zu Verlassen, nachdem das letzte Bit aller Slow Control-Befehle in die Shiftregister-Matrix geschrieben wurde. In diesem Fall erfolgt der Übergang in den Zustand `frame_done`, ansonsten in den Zustand `wait_s`

wait_s Hier wird der Zähler `count_rdcycle` auf Null zurückgesetzt und gewartet bis die Shiftregister-Matrix bereit ist neue Daten aufzunehmen. Danach wird der Zähler `count_rdstate` inkrementiert und es erfolgt ein Wechsel in den Zustand `read`.

frame_done Hier wird das Signal `frm_done` gesetzt, welches `fsm_shift` dazu veranlasst den Ladevorgang der Matrix im nächsten Takt zu beenden und die ungeladenen Zeilen nicht zu füllen. Im realen Betrieb muss die Matrix fünf Mal vollständig und ein Mal bis einschließlich der vierten Zeile geladen werden um die 84 Bit des Slow Control-Frames für alle Stationen zu deserialisieren. Durch die Möglichkeit den Ladevorgang vorzeitig abzubrechen können pro Frame-Satz zwölf Takte eingespart werden, wodurch sich der maximal erreichbare Durchsatz um ca. 6 % erhöht. Nach einem Takt kehrt `fsm_fifo` in den Ruhezustand zurück.

3.3.5 fsm_shift

Das Zustandsdiagramm für diese FSM ist identisch mit jenem für `fsm_shift` im Modul `shifters_s_tx` (siehe S. 41). In der Festlegung der Zustandsübergänge unterscheiden sich die beiden Automaten jedoch. Die Unterschiede in den Zuständen sollen hier dargestellt werden.

load Die Implementierung der FSM `fsm_shift` im RX-Pfad lädt die Matrix immer komplett bzw. bis sie das Signal `frm_done` erhält. Es ist nicht möglich einzelne Zeilen beim Laden zu überspringen, wie dies in `shifters_s_tx` der Fall war (siehe Abschn. 3.1.6). Dies ist hier nicht sinnvoll, da jede Zeile Daten für *alle* Slow Control-Stationen enthält. In Betrieb wird die Matrix jeweils fünf Mal komplett und anschließend einmal bis einschließlich der vierten Zeile geladen.

Bevor der Zustand verlassen wird, aktiviert `fsm_shift` ein Anforderungssignal um der FSM `fsm_bram` mitzuteilen, dass ab dem nächsten Takt gültige Daten am Ausgang der Matrix bereitstehen.

shift Der zentrale Unterschied zur Steuerung der Shiftregister-Matrix im TX-Pfad ist, dass der Shiftvorgang bei dieser Implementierung abgebrochen werden kann. Der Grund dafür ist die Anordnung der Daten in der Matrix, bei der die *Spalten* den einzelnen Slow Control-Stationen entsprechen (siehe Abb. 3.16). Wenn nur für manche Stationen ein gültiges Kommando vorliegt, so kann der Shiftvorgang abgebrochen werde, nachdem die entsprechenden Daten bis zum Datenausgang der Matrix geschoben wurden. Für die Steuerung dieser Logik wird das Statusregister `valid_reg` und der damit verbundenen Prioritätsencoder benötigt (siehe Abb. 3.14), dessen Ausgang die Nummer des höchsten gesetzten Gültigkeits-Flags angibt.

In weiterer Unterschied zur Implementierung in `shifters_s_tx` bezieht sich auf das Setzen des Bereitschaftes-Flag `shima_rdy`. Im TX-Pfad hat die FSM direkt beim Eintreten in den Zustand `shift` ihre Bereitschaft zur Aufnahme neuer Daten signali-

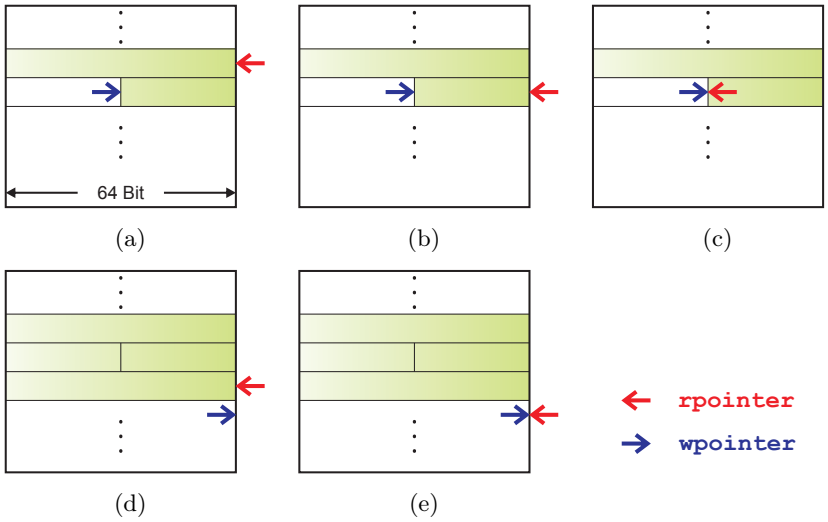


Abbildung 3.18: Prinzip der Zeigerverwaltung im Modul `pointermgmt`. In Situation (a) befindet sich genau ein Slow Control-Frame in `bram_s_rx`, die Zeiger zeigen auf den Anfang bzw. das Ende des gültigen Speicherbereichs. Nachdem ein Wort gelesen wurde muss `rpointer` um zwei inkrementiert werden (b), bei einem erneuten Lesezugriff aber nur um eins (c). Wird nun ein weiteres Frame in das BRAM geschrieben und anschließend gelesen, so wird `rpointer` zunächst um eins inkrementiert (d) und beim nächsten Lesezugriff um zwei (e).

siert. Dies war dort sinnvoll, da die Latenz durch das Pipelinedesign bei der Berechnung der Leseadressen im Modul `shifters_s_tx` (siehe S. 35) dadurch kompensiert werden konnte. In dem hier vorliegenden Fall ist jedoch zum einen die Matrix doppelt so breit, zum anderen ist die Latenz wesentlich geringer, nämlich nur zwei Takte: ein Takt für die Aktivierung der FSM `fsm_fifo` aus dem Ruhe- bzw. Wartezustand und ein weiterer Takt aufgrund des synchronen Leseverhaltens von `fifo_s_rx`. Somit darf `fsm_shift` erst zwei Takte vor dem Ende des Shiftvorgangs das Bereitschafts-Flag setzen.

Von diesem Zustand aus erfolgt der Übergang entweder in den Zustand `empty` oder, falls `fsm_fifo` anzeigt, dass gültige Daten aus `fifo_s_rx` gelesen werden können, nach `load`. Bevor der Zustand verlassen wird, aktiviert `fsm_shift` ein Signal, welches `fsm_bram` das Ende gültiger Daten anzeigt.

3.3.6 pointermgmt

Dieses Modul verwaltet den Speicher `bram_s_rx`. Alle Module, die Daten in dieses BRAM schreiben oder aus ihm lesen wollen müssen mit `pointermgmt` kommunizieren. Das Ziel der Implementierung ist es, `bram_s_rx` wie eine Art FIFO zu behandeln. Dazu müssen Zeiger bereitgestellt werden, die jeweils auf den Beginn des freien Speicher bzw. auf den Beginn gültiger Daten zeigen. Diese Zeiger werden von `pointermgmt` generiert und verwaltet. Anhand der Abbildung 3.15 wird klar, dass es hierbei genügt, diese Zeiger mit einer Granularität von 32 Bit zu implementieren, da gültige Daten – also vollständig abgespeicherte Frames – immer ein Vielfaches von 32 Bit belegen.

Die Kommunikation des Moduls `pointermgmt` mit der übrigen Logik erfolgt über die beiden Zeiger-Ports `rpointer`, welcher den Beginn gültiger Daten markiert, und

wpointer um deren Ende festzulegen. Die Manipulation der Zeiger erfolgt über die Inkrement-Signale rpointer_incr und wpointer_incr. Wird wpointer_incr aktiviert, so erhöht sich mit der nächsten steigenden Taktflanke der Wert von wpointer um drei ($\widehat{=}3 \cdot 32$ Bit). Die Verhältnisse bezüglich des Lesezeigers rpointer sind nicht ganz so einfach, weil der Zeiger nicht um einen konstanten Wert erhöht werden darf, sondern die Inkrementierung von den aktuellen Werten beider Zeiger abhängt. Dies ist aus Abbildung 3.18 ersichtlich. Beim Übergang von (b) nach (c) sowie von (c) nach (d) wird der Zeiger jeweils um eins inkrementiert, sonst um zwei. Die Inkrementierung ist intern dabei wie folgt implementiert: das Inkrement-Signal wird mit der steigenden Taktflanke gesampelt. Es wird verglichen, ob die führenden Stellen beider Zeiger bis auf das LSB[6] identisch sind. Dies entspricht den Situation (b), (c) sowie (e) in der Abbildung, wobei in (c) und (e) auch das LSB übereinstimmt. Die Logik setzt nun das LSB von rpointer gleich dem LSB von wpointer, lässt die restlichen Bits aber unverändert. Somit wird ausgehend von (b) der Zustand (c) erreicht. Wird im Zustand (c) oder (e) fälschlicherweise erneut das Signal rpointer_incr aktiviert, so ändert sich rpointer aufgrund dieser Implementierung nicht. Es wird also verhindert, dass rpointer den Wert von wpointer übersteigt und damit unter Umständen ein völliges Fehlverhalten der Logik verursachen könnte. Wenn die führenden Stellen der beiden Zeiger nicht identisch sind, wird beim Inkrementieren der konstante Wert zwei zu rpointer addiert und anschließend das LSB auf '0' gesetzt. Damit wird sowohl der Übergang von Situation (c) zu (d), als auch von (d) nach (e) ohne zusätzliche Fallunterscheidung und somit ressourcenschonend realisiert.

Neben den Zeigern werden von pointermgmt auch zwei Statusflags zur Verfügung gestellt, ein empty und ein full Flag. Das Signal empty wird von der MMU benutzt um festzustellen ob Daten zum Transport via Ethernet bereitstehen. Es ist '1' wenn beide Zeiger identisch sind. Das Flag full wurde implementiert um die FSM fsm_bram davon abzuhalten Schreibzugriffe zu starten obwohl zu wenig freier Speicher vorhanden ist. Das Flag ist dabei so definiert, dass es aktiv ist wenn weniger als $17 \cdot 6$ Datenworte frei sind. Dies ist die Anzahl an Worten, die benötigt werden um für jede Station genau ein Slow Control-Frame abzulegen.

3.3.7 fsm_bram

Dieses Modul regelt den Schreibzugriff auf den Speicher bram_s_rx. Dazu gehört die Kontrolle ob genügend freier Speicherplatz vorhanden ist und die Berechnung korrekter Schreibadressen. Da in der Shiftregister-Matrix jede Spalte einer Slow Control-Station entspricht, reicht es nicht aus sequentielle Schreibadressen zu erzeugen um die in Abbildung 3.15 gezeigte Datenanordnung zu erreichen. Vielmehr muss nach jedem Zugriff um sechs Worte gesprungen werden. fsm_bram muss außerdem dafür Sorge tragen, dass nur jene Slow Control-Frames im Speicher abgelegt werden, für die der Eintrag im Statusregister valid_reg gesetzt ist.

Das entsprechende Zustandsdiagramm ist in Abbildung 3.19 dargestellt. Es wird ein Zähler zur Steuerung der Zustandsübergänge und gleichzeitig zur Berechnung der Schreibadressen benötigt (wrstate_count). Die Manipulation der Schreibadresse erfolgt mittels der beiden Signale waddr_set und waddr_incr. Bei der Aktivierung von waddr_set wird die Adresse auf $2 \cdot$ wpointer + wrstate_count gesetzt. Der Faktor zwei rührt von der 32-Bit-Granularität des Zeigers her. waddr_incr erhöht

[6] Least Significant Bit

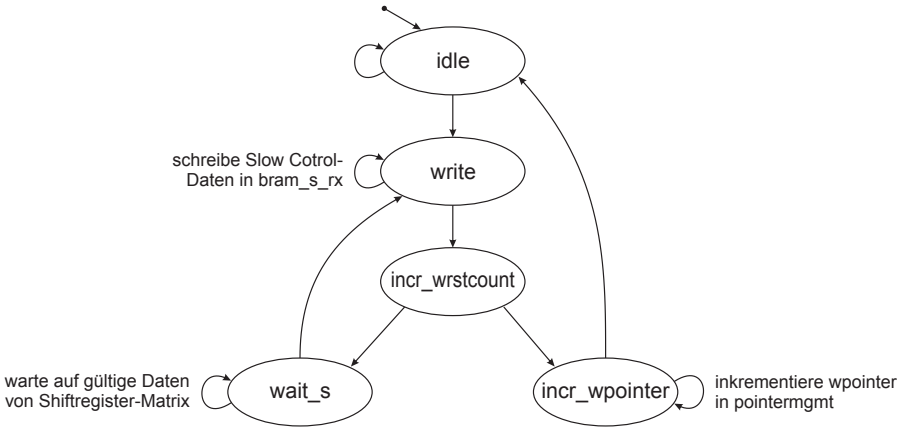


Abbildung 3.19: Zustandsdiagramm der FSM `fsm_bram`. Die Aufgaben dieser FSM beinhalten die Berechnung von Schreibadressen und das Inkrementieren des Schreibzeigers `wpointer`.

das Adressregister um sechs und ermöglicht somit einen Sprung zum nächsten Block in `bram_s_rx`.

idle In diesem Zustand wird überprüft, ob in `bram_s_rx` noch genügend freier Speicher vorhanden ist. Dazu wird das `full` Flag von `pointermgmt` benutzt. Es sei darauf hingewiesen, dass bei der Erzeugung dieses Flags immer davon ausgegangen wird, dass 17 Slow Control-Frames gespeichert werden müssen, obwohl die genaue Anzahl der Frames anhand des Statusregisters `valid_reg` bestimmt werden kann. Die statische Implementierung wurde gewählt um den Logikaufwand so gering wie möglich zu halten. Falls genug Speicher vorhanden ist wird ein Bereitschafts-Signal gesetzt und damit `fsm_shift` signalisiert, dass Daten aufgenommen werden können. Sobald `fsm_shift` die Verfügbarkeit gültiger Daten anzeigt, verlässt `fsm_bram` den Ruhezustand nach `write`. Außerdem wird in diesem Zustand der interne Zähler `wrstate_count` zurückgesetzt.

write In diesem Zustand muss das write enable Signal für `bram_s_rx` erzeugt und die Schreibadresse generiert werden. Das write enable wird mit Hilfe des Multiplexers aus dem Statusregister gewonnen. Damit ist sichergestellt, dass nur Frames in `bram_s_rx` gespeichert werden, die vom Modul `eth2sctrl_mac` als gültig markiert wurden. Des Weiteren wird das gemultiplexte Flag benutzt um die Schreibadresse in 6er-Schritten zu inkrementieren. Hierbei ist zu beachten, dass das Inkrementieren der Adresse erst bei der nächsten steigenden Taktflanke erfolgt, wohingegen das write enable sofort gültig wird. Durch das Inkrement-Signal `waddr_incr` wird also bereits die Adresse für den nächsten Zugriff vorbereitet.

Nachdem alle gültigen, deserialisierten Daten aus der Shiftregister-Matrix in `bram_s_rx` gespeichert wurden erfolgt der Wechsel in den Zustand `incr_wrstcount`.

incr_wrstcount Hier wird anhand des Zählers `wrstate_count` überprüft ob die aktuellen Frames bereits komplett in `bram_s_rx` geschrieben wurden. In diesem Fall wird der Zähler zurückgesetzt und es erfolgt nach einem Takt der Zustandswechsel nach `incr_wpointer`. Andernfalls wird der Zähler um eins inkrementiert und es erfolgt der Übergang in den Zustand `wait_s`.

wait_s Das Signal `waddr_set` wird hier gesetzt, und somit die Schreibadresse auf das erste freie Datenwort in `bram_s_rx` gesetzt. Des Weiteren wird das Bereitschafts-

Flag gesetzt und auf gültige Daten am Ausgang der Shiftregister-Matrix gewartet. Danach wechselt `fsm_bram` wieder in den Zustand `write`.

incr_wpointer Hier wird der Zeiger `wpointer`, der das Ende der gültigen Daten in `bram_s_rx` angibt, auf die richtige Position geschoben. Dazu muss das Signal `wpointer_incr` für die entsprechende Anzahl an Takten aktiviert werden. Da das Signal `wpointer` in diesem Zustand nicht stabil ist und sich in jedem Takt ändert, wurde ein Flag `wpointer_valid` implementiert, welches immer aktiv ist, außer in diesem Zustand.

4 Implementierung des Konfigurationsvorgangs

In diesem Kapitel wird dargestellt wie der Konfigurationsvorgang der Nathan-FPGAs via Ethernet abläuft. Es werden die dazu implementierten Submodule und deren Zusammenwirken erläutert. Außerdem wird ein neuer Pakettyp definiert, der festlegt wie das Bitfile in Ethernet-Frames verpackt und zur Backplane transportiert wird.

Die Konfiguration verläuft im *Slave Serial Programming Mode*, bei dem ein serieller ein Bit breiter Datenstrom zu den FPGAs gesendet wird (siehe Abschn. 1.1.4.3). Der Konfigurationstakt wird dabei vom Balu-FPGA erzeugt. Vor der Konfiguration besteht die Möglichkeit zu selektieren, welche Nathan-FPGAs konfiguriert werden und der Bitstrom wird anschließend simultan an die selektierten Module gesendet werden. Das Konzept sieht weiterhin vor, das die in diesem Kapitel definierten Konfigurations-Pakete (CFG-Pakete) ebenso wie die STP-Pakete vom Ethernet-Core in `bram_s_tx` abgelegt werden. Bezüglich des Sliding-Window-Algorithmus werden die CFG-Pakete behandelt wie STP-Pakete, die an die Slow Control-Station 16 (Balu-FPGA) adressiert sind. Insbesondere wird für den Empfang der Pakete ein Empfängerfenster mit einer Breite von vier Paketen bereitgestellt. Das Modul `shifters_s_tx` wurde so erweitert, dass auch CFG-Pakete korrekt aus `bram_s_tx` gelesen werden. Die Datenworte werden jedoch nicht über die Shiftregister-Matrix serialisiert, sondern in einem FIFO (`fifo_cfg`) gespeichert (siehe Abb. 4.1). Dieser FIFO dient dem Übergang zwischen zwei Taktdomänen, wobei die Schreibzugriffe mit 125 MHz erfolgen und die Lesezugriffe mit 39 MHz. Außerdem wird `fifo_cfg` zur Serialisierung und Pufferung des Bitstroms benötigt.

Da ein Bitfile aufgrund seiner Größe nicht vollständig in ein Ethernet-Paket gerahmt werden kann, wird es auf mehrere CFG-Pakete aufgeteilt. Eine Problematik welche hierdurch entsteht ist, dass es aufgrund von Übertragungsfehlern zu Unterbrechungen im Bitstrom zu den Nathan-FPGAs kommen kann. In einem solchen Fall muss das zu den Nathan-Modulen gehende Taktsignal angehalten werden. Diese Vorgehensweise ist mit der Spezifikation des Konfigurationsablaufs konform [17].

Im Folgenden werden sämtliche Submodule behandelt, welche für die Realisierung des Konfigurationsvorgangs der Nathan-FPGAs via Ethernet benötigt werden. Die FSM `eth2sctrl_config` steuert und überwacht die Takt- und Konfigurationsleitungen zu den Nathan-Modulen. Eine weitere FSM, welche in `shifters_s_tx` implementiert ist, steuert den Zugriff auf `bram_s_tx` (`fsm_buf_cfg`). Des Weiteren existiert das Slow Control-Modul `nathanconfig` im Balu-FPGA, welches eine zentrale Rolle bei der Konfiguration einnimmt, da sich hierüber konfigurationsrelevante Register setzen und auslesen lassen. Es wird auch auf das neu definierte CFG-Paketformat eingegangen, welches festlegt, wie eine .bit-Datei in ein Ethernet-Frame verpackt werden muss.

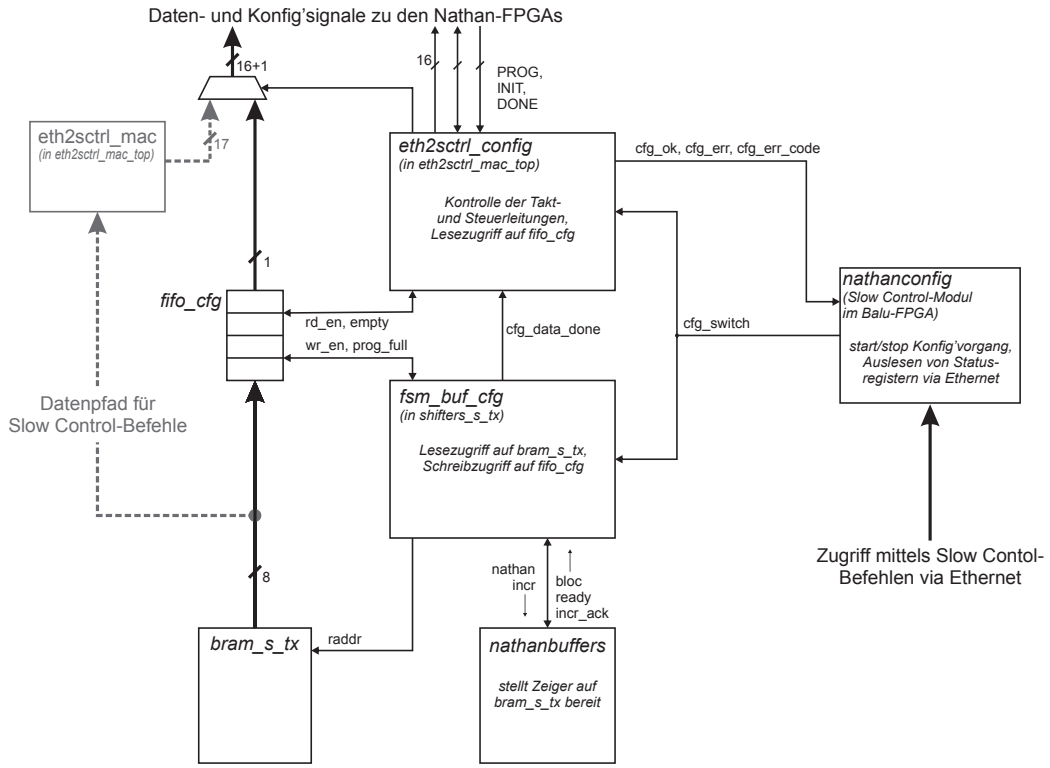


Abbildung 4.1: Darstellung der an der Konfiguration der Nathan-FPGAs beteiligten Module. Die FSM `eth2sctrl_config` steuert die zu den Nathan-FPGAs führenden Takt- und Konfigurationssignale, die FSM `fsm_buf_cfg` liest den Bitstrom aus dem Speicher `bram_s_tx`. Über das Slow Control-Modul `nathanconfig` lassen sich vom PC aus Register setzen und auslesen.

4.1 Das CFG-Paketformat

Um das Bitfile für einen Nathan-FPGA über die Ethernet-Verbindung zur Backplane transportieren zu können muss ein neuer Pakettyp definiert werden. Dieser Typ wird als *CFG-Paket* bezeichnet. Diese CFG-Pakete werden – analog zu STP-Paketen – als Payload in Ethernet-Frames gerahmt. Es wird hier darauf eingegangen, wie ein solches Paket auszusehen hat, und warum diese Definition gewählt wurde.

Die einfachste Möglichkeit auf `bram_s_tx`, in dem diese Pakete gespeichert werden, zuzugreifen, ist ein Lesezugriff auf sequentielle Adressen. Für die Berechnung der Leseadressen würde dies bedeuten, dass lediglich die Speicherblockadresse und ein Zähler für die Byteposition innerhalb des Blocks benötigt werden. Um die erforderliche sequentielle Anordnung der Daten im Speicher zu erreichen muss das Bitfile ebenfalls sequentiell in das Ethernet-Frame verpackt werden (siehe Abb. 4.2). In jedem CFG-Paket befinden sich exakt 512 Byte des Bitfiles. Das letzte CFG-Paket muss mit Nullen aufgefüllt werden. Dies ist notwendig, da die Logik, welche die Daten aus `bram_s_tx` liest, von einer konstanten Größe der Pakete ausgeht. Wenn ein CFG-Paket mit weniger als 512 Byte im Balu-FPGA ankäme, so würde die Logik teilweise veraltete Daten auslesen und als Bitstrom zu den Nathan-FPGAs senden. Die Pakete müssen weiterhin mit den korrekten Sequenz-Nummern versehen werden und als Ziel den Balu-FPGA (Station 16) im Header tragen. Der Transfer der CFG-Pakete ist ebenso wie für die STP-Pakete durch den Sliding-Window-Algorithmus abgesichert.

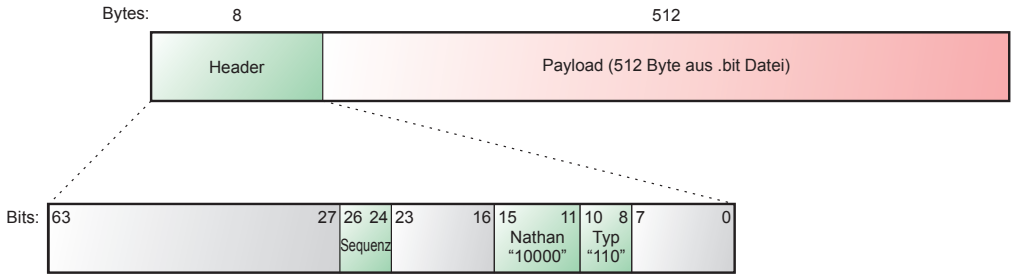


Abbildung 4.2: Definition des CFG-Paketformats. Diese Pakete transportieren ein Segment des Bitfiles zur Backplane. Der Payload besteht aus einem 512 Byte langen Abschnitt des Bitstroms.

4.2 Das Modul nathanconfig

Bei dem Modul `nathanconfig` handelt es sich um ein Slow Control-Modul, welches im Balu-FPGA implementiert ist. Es besitzt drei wesentliche Aufgaben die im Folgenden dargestellt werden.

Damit Nathan-FPGAs konfiguriert werden können, müssen Teile der Logik im Balu-FPGA in den Konfigurationszustand versetzt werden. `eth2sctrl_mac` soll beispielsweise keine Frames mehr aussenden, `shifters_s_tx` muss wissen, dass Konfigurationsdaten aus `bram_s_tx` gelesen werden sollen und `eth2sctrl_config` muss die Konfigurations-Steuerleitungen entsprechend ansteuern. Außerdem muss festgelegt werden, welche Nathan-FPGAs programmiert werden sollen. Das Modul `nathanconfig` stellt zu diesem Zweck zwei Register zur Verfügung. Das Register `cfg_switch` wird gesetzt um den Beginn eines Konfigurationszyklus anzuzeigen, das 16 Bit breite Register `PROGmask` dient der Festlegung welche Nathan-FPGAs konfiguriert werden sollen. Diese Register werden mittels eines Slow Control-Schreibbefehls an eine dedizierte Adresse des Moduls gesetzt bevor mit der Konfiguration begonnen wird.
Eine weitere Aufgabe des Moduls besteht darin, Statusregister bereitzustellen, die von `eth2sctrl_config` gesetzt und über Slow Control-Lesebefehle ausgelesen werden können. Dabei handelt es sich um die Flags `cfg_ok`, `cfg_err` und `cfg_err_code`, die angeben, ob der Konfigurationsvorgang erfolgreich war oder nicht. Falls während des Konfigurationszyklus ein Fehler auftrat gibt `cfg_err_code` die Art des Fehlers an.
Zuletzt bietet das Modul dem Anwender noch die Möglichkeit vom PC aus festzustellen, welche Nathan-Sockel besetzt sind und ob die entsprechenden Nathan-FPGAs bereits programmiert sind. Da die Backplane als Experimentierplattform auch Partnern innerhalb des FACETS-Projekts zur Verfügung gestellt werden soll, kann nicht sichergestellt werden, dass der Experimentator immer vor Ort im Labor ist und den Bestückungszustand des Systems kennt. Somit muss eine Möglichkeit angeboten werden den Zustand des Systems über die Ethernet-Anbindung feststellen zu können. Die Abfrage der Zustände der einzelnen Sockel erfolgt dabei über Lesebefehle, wobei jedem Sockel innerhalb des Moduls eine Adresse zugeordnet ist.

4.3 Steuerung der Konfigurationssignale

Die Steuerung der Konfigurationssignale zu den Nathan-Modulen wird von der FSM `eth2sctrl_config` wahrgenommen. Sie steuert auch den Multiplexer am Datenaus-

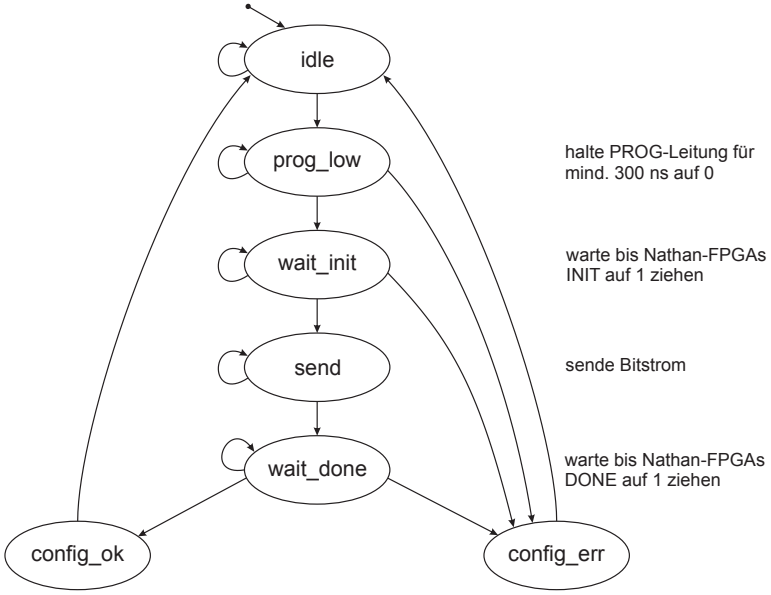


Abbildung 4.3: Zustandsgraph der FSM `eth2sctrl_config` zur Steuerung der Konfigurationsleitungen.

gang zu den Slow Control-Stationen im Modul `eth2sctrl_mac_top` (siehe Abb. 4.1) und legt damit fest ob und an welche Nathan-FPGAs Bitstrom-Daten gesendet werden. Das entsprechende Zustandsdiagramm ist in Abbildung 4.3 dargestellt. Die Grundlagen zum Ablauf der Konfiguration sowie die Bedeutung der einzelnen Signale wurde in Abschnitt 1.1.4.3 erklärt.

idle In diesem Zustand befindet sich die FSM `eth2sctrl_config` während des normalen Betriebs der Backplane. Um die Kommunikation zwischen dem Modul `eth2sctrl_mac` und den Slow Control-Stationen zu ermöglichen wird der Multiplexer so angesteuert, dass die Daten, die `eth2sctrl_mac` aussendet, direkt an die entsprechenden Stationen weitergeleitet werden. Wenn das Register `cfg_switch` von `nathanconfig` gesetzt wird erfolgt ein Übergang in den Zustand `prog_low`.

prog_low Hier wird die `PROG`-Leitung für die durch das Register `PROGmask` festgelegten Nathan-Module auf `'0'` gesetzt und damit die Konfigurationsdaten der betreffenden Nathan-FPGAs gelöscht (`PROG` ist low aktiv, siehe Abschn. 1.1.4.3). Die Leitungen müssen mindestens für 300 ns aktiviert bleiben, was über einen internen Zähler gesteuert wird [30]. Danach erfolgt der Wechsel in den Zustand `wait_init` oder, falls einer der Nathan-FPGAs seine `INIT`-Leitung nicht auf `'0'` gesetzt hat, in den Zustand `config_err`.

In diesem und allen folgenden Zuständen wird der Multiplexer am Datenausgang zu den Slow Control-Stationen so angesteuert, dass der Ausgang von `fifo_cfg` mit den Dateneingängen der selektierten Nathan-Module verbunden ist. Die Leitungen zu den nicht selektierten Modulen werden stumm geschaltet. Das Modul `eth2sctrl_mac` hat ab jetzt bis zum Ende der Konfiguration keine Möglichkeit mehr Frames oder Tokens an die Slow Control-Stationen zu senden.

wait_init In diesem Zustand wird eine definierte Anzahl von Takten gewartet, bis alle selektierten Nathan-Module ihre `INIT`-Leitung auf `'1'` ziehen und damit anzeigen, dass sie bereit sind Konfigurationsdaten zu empfangen. Sollte wenigstens

ein Nathan-FPGA nicht bereit sein Daten aufzunehmen, so erfolgt der Übergang in den Zustand `config_err`, ansonsten in den Zustand `send`.

send Nun können die Bitstrom-Daten zu den Nathan-FPGAs gesendet werden. Falls nötig muss der ausgehende Takt angehalten werden. Dieser Fall tritt dann auf, wenn keine gültigen Daten mehr in `fifo_cfg` gespeichert sind, er also vollständig geleert wurde. Der Bitstrom zu den Nathan-FPGAs ist mit 39 MHz getaktet. Dies entspricht einer Datenrate von $39/8 = 4,88$ MB/s. Ein unterbrechungsfreies Senden des Bitstroms ist nur möglich, wenn der PC in der Lage ist, diese Datenrate zur Backplane zu transferieren. Die Software am PC muss also im Mittel ca. 10.000 CFG-Pakete pro Sekunde erzeugen und versenden. Da dies nicht garantiert werden kann, ist es notwendig eine Möglichkeit zu implementieren, die Bitstrom-Übertragung anzuhalten. Dies wird über das empty Flags von `fifo_cfg` und das Anhalten des ausgehenden Taktsignals gesteuert.

Die FSM `eth2sctrl_config` bleibt in diesem Zustand, bis von der Logik im Modul `shifters_s_tx` das Flag `cfg_data_done` gesetzt wird und `fifo_cfg` geleert wurde. Sind diese Bedingungen erfüllt, so wurde das Bitfile vollständig gesendet und es erfolgt der Übergang in den Zustand `wait_done`.

wait_done Hier wird gewartet, bis das `DONE`-Signal von allen neu konfigurierten Nathan-FPGAs auf `'1'` gesetzt wird, um eine erfolgreiche Konfiguration anzuzeigen. Es wird auch überprüft ob die `INIT`-Leitung `'1'` ist, da der Nathan-FPGA die `INIT`-Leitung aktiv auf `'0'` ziehen kann, wenn während der Übertragung des Bitstroms ein Prüfsummenfehler feststellt wird. Nur wenn beide Leitung bei allen selektierten Nathan-FPGAs aktiviert sind, erfolgt der Wechsel in den Zustand `config_ok`. Falls nach einem festgelegten Zeitintervall mindestens eine der Leitungen deaktiviert ist, erfolgt der Übergang in den Zustand `config_err`.

config_ok / config_err In diesen Zuständen wird das Signal `cfg_ok` respektive `cfg_err` aktiviert. Dadurch wird zum einen das entsprechende Statusflag im Slow Control-Modul `nathanconfig` gesetzt, zum anderen wird das Register `cfg_switch` in `nathanconfig` dadurch gelöscht. Der Konfigurationsvorgang ist damit abgeschlossen, die FSM `eth2sctrl_config` kehrt in den Ruhezustand zurück und die gesamte Logik im Balu-FPGA fährt mit der Bearbeitung von STP-Paketen fort. Das Register `cfg_error_code` wird jeweils beim Wechsel in den Zustand `cfg_err` gesetzt und gibt den Grund für das Scheitern der Programmierung an. Sämtliche Statusflags können vom PC aus via Ethernet durch einen Slow Control-Lesebefehl an `nathanconfig` ausgelesen werden.

4.4 Ansteuerung des Paket-Puffers

Der Zugriff auf den Paket-Puffer `bram_s_tx` wird vom Modul `shifters_s_tx` gesteuert. Dieses Modul muss dazu – ähnlich dem Zugriff auf STP-Pakete – von `nathanbuffers` die Speicherblock-Nummern abfragen und Leseadressen berechnen. Das Ziel bei der Entwicklung lag vor allem darin, möglichst wenig zusätzliche Logik zu implementieren. So wird beispielsweise die Pipeline zur Adressberechnung, die in Abschnitt 3.1.3 diskutiert wurde, komplett wiederverwendet obwohl der Großteil davon nicht benötigt wird. Die Schnittstelle zum Modul `nathanbuffers` konnte ohne Modifikationen übernommen werden, es wurden insbesondere keine neuen Signale zwischen den beiden Modulen benötigt.

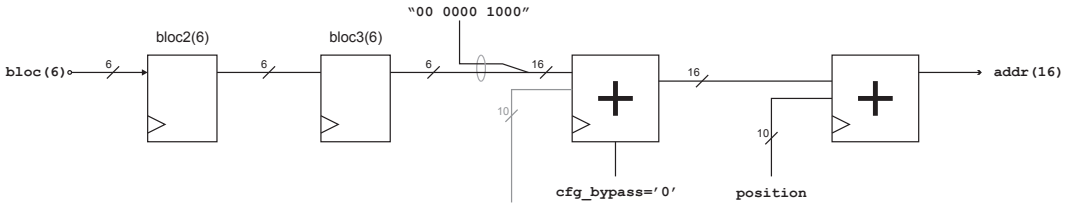


Abbildung 4.4: Pipeline für die Adressberechnung beim Zugriff auf CFG-Pakete. Sämtliche Register und Addierer werden für die Adressberechnung bei CFG- und STP-Paketen gemeinsam genutzt (vgl. Abb. 3.4(b), S. 35). Der erste Addierer verfügt über einen `bypass` Eingang, so dass die mit 1024 multiplizierte und um den Offset korrigierte Blockadresse direkt auf seinen Ausgang geschaltet werden kann.

Aufgrund der Definition des CFG-Pakettyps erfolgt beim Auslesen der Bitstrom-Daten aus `bram_s_tx` der Lesezugriff auf sequentielle Adressen. Für die Pipeline zur Berechnung der Leseadressen im Modul `shifters_s_tx` bedeutet dies, dass nur die Speicherblockadresse `bloc` und ein Zähler für die Byteposition innerhalb des Blocks (`position`) benötigt werden um Leseadressen zu berechnen (siehe Abb. 4.4). Der Einfluss weiterer Summanden, die nur für den Zugriff auf STP-Pakete benötigt werden, kann durch das Signal `cfg_bypass` unterdrückt werden. `cfg_bypass` wurde als low aktives Signal implementiert, da hierdurch weniger Platz im FPGA in Anspruch genommen wird [31]. Die Pipeline nimmt damit vereinfacht die in Abbildung 4.4 dargestellte Form an.

In Abschnitt 3.1.5 wurde erwähnt dass die FSM zur Steuerung des BRAM-Zugriffs, `fsm_buf`, über einen Zustand `config_mode` verfügt. Dieser Zustand stellt einen Meta-Zustand dar, in dem die untergeordnete FSM `fsm_buf_cfg` gestartet wird, welche den BRAM-Lesezugriff während der Konfiguration steuert. Außerdem wird in diesem Zustand das Signal `nathan`, über welches das Modul `shifters_s_tx` mit `nathanbuffers` kommuniziert, konstant auf 16 gesetzt, da CFG-Pakete gemäß ihrer Definition an die Slow Control-Station 16 adressiert sind. Der Zustandsgraph der FSM `fsm_buf_cfg` ist in Abbildung 4.5 gezeigt.

idle Der implementierte Zähler `pkg_count`, welcher die Anzahl der ausgelesenen CFG-Pakete zählt, wird in diesem Zustand auf Null zurückgesetzt. `fsm_buf_cfg` wartet auf ein Anforderungssignal von der FSM `fsm_buf` um diesen Zustand nach `wait_ready` zu verlassen.

wait_ready Hier wird gewartet bis das Flag `ready` von `nathanbuffers` signalisiert, dass die Speicherblocknummer für das nächste CFG-Paket gültig ist und auf den entsprechenden Bereich in `bram_s_tx` zugegriffen werden darf. Mit diesem Flag wechselt `fsm_buf_cfg` in den Zustand `wait_st` und erhöht den Zähler `pkg_count` um eins.

wait_st Hier wird ein weiterer Takt gewartet bevor in den Zustand `read` gewechselt wird. Diese ist aufgrund des Pipelinedesigns notwendig um zu verhindern, dass auf ungültige Speicheradressen zugegriffen wird.

read Dies ist der zentrale Zustand der FSM. Hier wird gesteuert wann der Zähler `position`, welcher die Byteposition der Leseadresse festlegt, erhöht wird und wann das write enable für `fifo_cfg` gesetzt wird. Aufgrund der Tatsache, dass `fifo_cfg` mit acht Bit breiten Datenworten bei 125 MHz beschrieben und bei 39 MHz ein Bit breite Worte gelesen werden ist klar, dass die meiste Zeit nicht in den FIFO geschrie-

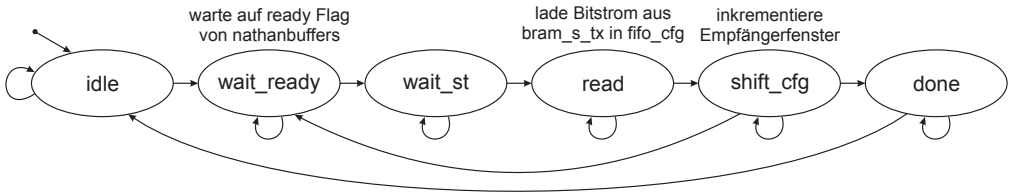


Abbildung 4.5: Zustandsgraph der FSM `fsm_buf_cfg`. Diese FSM steuert des Lesezugriff auf `bram_s_tx` während des Konfigurationszyklus.

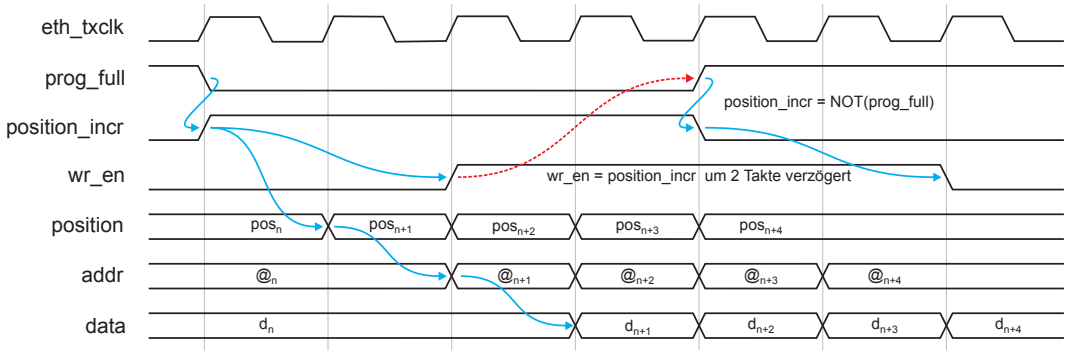


Abbildung 4.6: Timingdiagramm zur Herleitung des Schwellwerts für das programmierbare full Flag für `fifo_cfg`. Aufgrund des synchronen Systemdesigns erfolgen die Schreibzugriffe immer in 'Bursts', der Schwellwert muss passend gewählt werden. In dem hier gezeigten Beispiel werden die Datenworte d_n, d_{n+1}, d_{n+2} und d_{n+3} in einem Burst geschrieben.

ben werden kann. Sowohl die Inkrementierung des Positions-Zählers, also auch das write enable müssen vom Füllstand des FIFOs abhängen. Wie die folgende Betrachtung zeigt, wird dafür das programmierbare full Flag benutzt, dessen Schwellwert mindestens drei Worte unter dem maximalen Füllstand liegt. Das programmierbare full Flag stellt invertiert direkt das Inkrement-Signal für den Positionszähler dar (siehe Abb. 4.6). Aufgrund des synchronen Verhaltens sowohl des Zählers, als auch der Adresse und des Datenausgangs von `bram_s_tx` erfolgt der erste Schreibzugriff auf `fifo_cfg` drei Takte nachdem das Flag auf `'0'` wechselt. Es liegt nahe das write enable Signal für `fifo_cfg` über ein SRL16 (siehe S. 20) aus dem Inkrement-Signal zu erzeugen, da beide immer für die gleiche Anzahl an Takten aktiv sein müssen. Das programmierbare full Flag kann frühestens einen Takt nach dem ersten Schreibzugriff wieder auf `'1'` wechseln. In der Zwischenzeit wurde aber der Positionszähler bereits vier Mal inkrementiert und die dementsprechend aus `bram_s_tx` gelesenen Daten müssen auch noch in `fifo_cfg` gespeichert werden. Daraus folgt, dass der Schwellwert des programmierbaren full Flags so gewählt werden muss, dass mindestens vier Schreibzugriffe erfolgen können nachdem das Flag deaktiviert wurde. Im realen Betrieb des Systems werden somit immer mehrere Datenworte direkt hintereinander in `fifo_cfg` geschrieben (*Bursts*) und anschließend eine Wartepause eingelegt, bis wieder genug freier Speicher vorhanden ist um den nächsten Burst aufzunehmen.

Nachdem ein CFG-Paket vollständig aus `bram_s_tx` ausgelesen wurde, wechselt die FSM `fsm_buf_cfg` in den Zustand `shift_cfg`. Dies wird mittels des Positionszählers gesteuert.

shift_cfg Hier wird das Modul `nathanbuffers` aufgefordert, das Empfängerfenster um eine Position zu verschieben und die Speicherblock-Nummer des Pakets mit

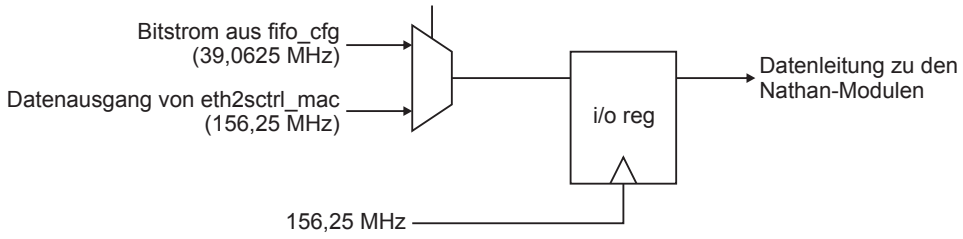


Abbildung 4.7: Datenanbindung der Nathan-Module an den Balu-FPGA. Da das I/O-Register immer mit 156,25 MHz getaktet wird, muss der Konfigurationstakt ein ganzzahliger Bruchteil davon sein.

der nächsten Sequenz-Nummer bereitzustellen. Da der Sliding-Window-Algorithmus nicht zwischen CFG- und STP-Paketen unterscheidet, entspricht das Zeitverhalten dem der Behandlung von STP-Paketen (siehe S. 40). Nachdem `fsm_buf_cfg` eine Bestätigung für das erfolgreiche Inkrementieren des Fensters vom Modul `nathanbuffers` erhält, wechselt sie entweder in den Zustand `wait_ready` um mit der Behandlung des nächsten CFG-Pakets fortzufahren oder, falls der Zähler `pkg_count` anzeigt, dass gerade das letzte Paket gelesen wurde, in den Zustand `done`. Dies ist möglich, da das Bitfile unabhängig von seinem Inhalt für einen spezifischen FPGA immer die gleiche Größe hat.

done Dieser Zustand dient dazu, dem Modul `eth2sctrl_config` mitzuteilen, dass der Bitstrom komplett in `fifo_cfg` abgelegt wurde. Dazu wird das Flag `cfg_data_done` gesetzt. Nachdem das Flag `cfg_switch` vom Slow Control-Modul `nathanconfig` deaktiviert und damit das Ende der Konfiguration signalisiert wurde, kehrt `fsm_buf_cfg` in den Ruhezustand zurück.

4.5 Taktung der ausgehenden Daten

In diesem Abschnitt wird dargestellt wie die ausgehenden Daten zu den Nathan-Modulen getaktet werden und warum eine Frequenz von 39 MHz für die Konfiguration gewählt wurde.

Auf der Backplane steht ursprünglich nur der vom Quarzoszillator erzeugte Takt mit einer Frequenz von 156,25 MHz zur Verfügung. Weitere Taktsignale wie z.B. der 125 MHz Takt für den Ethernet-PHY und Teile des eth2sctrl-Cores müssen über die DCMs im Balu-FPGA erzeugt werden. Dabei muss auch beachtet werden, dass eine zu große Anzahl von Logikkomponenten mit jeweils unterschiedlichen Taktfrequenzen unter Umständen das Routing der Logik im FPGA deutlich erschwert. Das Ziel war es somit mit einer möglichst minimalen Anzahl verschiedener Taktraten auszukommen. Bei der hier dargestellten Realisierung wird lediglich ein einziger DCM für die Erzeugung aller notwendigen Taktsignale benötigt.

Die ausgehenden Slow Control-Daten sollen mit 156,25 MHz getaktet werden. Das Ziel war es nun das Daten-I/O-Register *immer* mit dieser Frequenz zu takten und auch während der Konfiguration der Nathan-FPGAs die Frequenz nicht zu modifizieren (siehe Abb. 4.7). Da die Datenrate bei der Konfiguration aber 66 MBit/s nicht übersteigen darf [30], muss die Taktfrequenz für die Konfigurationslogik als ganzzahliger Bruchteil von 156,26 MHz gewählt werden. Die Wahl fiel auf 39,0625 MHz, da diese Frequenz erzeugt werden kann, ohne einen zusätzlichen DCM zu instantiieren. Außerdem ist der Konfigurationsvorgang nicht zeitkritisch, da die Nathan-FPGAs

relativ selten neu programmiert werden müssen. Wenn die Daten mit einer Frequenz von 39 MHz aus `fifo_cfg` gelesen und dann in das mit 156,25 MHz getaktete Ausgangsregister geschrieben werden, so wird immer vier Takte lang derselbe Wert geschrieben und der ausgehende Datenstrom hat die gewünschte Datenrate. Da der 39 MHz Takt mit dem 156,25 MHz Takt in Phase ist entstehen keine unregelmäßigen Ausgangssignale.

5 Evaluation des entwickelten Systems

Die vorangehenden Kapitel haben gezeigt, wie die Logik sich verhalten soll, wenn STP- bzw. CFG-Pakete zu den Slow Control-Stationen gesendet werden. Dieses Kapitel beschreibt den Test der Funktionsfähigkeit des Systems. Dabei wird zunächst auf Simulationsergebnisse eingegangen. Anschließend folgt eine Diskussion der durchgeführten Hardwaremessungen. Diese waren nur in begrenztem Umfang möglich, da Teile der hierfür notwendigen Software im Zeitraum der Arbeit nicht fertiggestellt werden konnten. Es steht somit Software zur Verfügung um die prinzipielle Funktionalität des Systems zu evaluieren, jedoch nicht um die Leistungsfähigkeit des Systems vollkommen auszunutzen.

5.1 Simulation

In diesem Abschnitt wird die funktionale Simulation des Gesamtsystems diskutiert. Es wird auch gezeigt welcher Durchsatz sich mit diesem System erreichen lässt und wo sich der Flaschenhals[1] befindet. Der Abschnitt ist grob in zwei Teile gegliedert; der erste befasst sich mit der Verarbeitung von Slow Control-Befehlen, während sich der zweite der Konfiguration widmet.

5.1.1 Slow Control-Kommunikation

Zunächst wird das vorhandene System zum Test des Systems (Testbench) dargestellt und es wird erläutert, welche Test mit dieser Testbench durchgeführt wurden. Anschließend wird auf die Performance des System eingegangen, wobei zwei Grenzfälle unterschieden werden.

Eine Beschränkung des implementierten Systems ist die in den Kapiteln 2 und 3 dargestellte Sterntopologie zwischen dem Balu-FPGA und den Nathan-FPGAs. Insbesondere besteht zwischen den einzelnen Minimalringen keine Datenverbindung. Das hat zur Folge, dass mit diesem System keine direkte Kommunikation zwischen zwei Nathan-Modulen über das Slow Control-Interface möglich ist. Die einzige Möglichkeit dies mit vertretbarem Hardwareaufwand zu ändern wäre sämtliche Kommunikation zwischen zwei Stationen über die Software am PC regeln zu lassen. Dazu müsste allerdings das STP-Paketformat geändert und das Feld `src` des Slow Control-Frames über das Ethernet mit übertragen werden. Im vorliegenden System besteht somit die einzige direkte Verbindung zwischen zwei Nathan-Modulen in Gestalt der MGT-Verbindungen.

5.1.1.1 Funktionale Verifikation

Die Testbench enthält ein VHDL-Modul `tb_backplane`, welches die vorhandenen Elemente auf der Backplane simuliert. Neben dem Balu-FPGA befindet sich dort auch die Nathan-Sockel, die entweder besetzt oder unbesetzt sein können sowie der

[1] als Flaschenhals bzw. Bottleneck bezeichnet man jenen Teil eines Hardwaresystems, welcher besonders zeitkritisch ist und sich negativ auf die Gesamtperformance auswirkt

Ethernet-PHY. Letzterer wird dabei in SystemC simuliert, wobei auch stochastische Übertragungsfehler eingefügt werden [16]. Die übrige Hardware wird in VHDL beschrieben.

Die SystemC-Testbench wurde im Rahmen dieser Arbeit so erweitert, dass es möglich wurde die zu sendenden Ethernet-Pakete in einer Textdatei festzulegen. Diese Textdatei enthält somit eine Liste aller Slow Control-Befehle, die in STP-Pakete verpackt und zur Backplane transferiert werden sollen. Prinzipiell ist es möglich eine derartige Eingabedatei von Hand zu erzeugen, jedoch bietet es sich bei großen Testreihen an dies mit Hilfe von Skripten zu automatisieren. Zu diesem Zweck wurde ein Python-Skript [23] erstellt, welches in der Lage ist, eine vorgegebene Anzahl von Kommandos in zufälliger Art und Weise auf mehrere STP-Pakete zu verteilen. Dabei werden sowohl die Kommandos an sich, als auch die Größe der einzelnen Pakete und die Nathan-Module, an die die Pakete gerichtet sind, zufällig bestimmt.

Um die korrekte Funktionalität des gesamten Balu-Designs inklusive eth2sctrl-Core zu überprüfen bietet die Testbench die Möglichkeit die auf jedem einzelnen Nathan-Modul ankommenden Slow Control-Frames in eine Textdatei zu schreiben. Die MMU (siehe S. 28), deren Entwicklung nicht Teil dieser Arbeit ist, bietet bislang jedoch keine Möglichkeit, Daten aus den Empfangspuffer `bram_s_rx` über die Ethernet-Anbindung zum PC zurück zu transferieren. Deswegen musste eine andere Möglichkeit gefunden werden, die korrekte Speicherung der Slow Control-Daten in `bram_s_rx` in der Simulation zu verifizieren. Hierzu wurde ein Modul entwickelt, welches den Inhalt von `bram_s_rx` in eine Textdatei schreibt, sobald gültige Daten vorliegen. Um den Vergleich der auf diese Weise ausgelesenen Daten mit der Stimulidatei zu vereinfachen, wurde ein C++-Parserprogramm geschrieben, welches den Vergleich automatisiert. Bei Schreib- und Lesebefehlen muss dabei überprüft werden, dass sowohl das bestätigte Frame als auch das Antwortframe in `bram_s_rx` gespeichert wurden und die Statusflags korrekt gesetzt sind. Somit erlaubt es die Testbench sowohl die Integrität der Frames im Datenpfad vom PC zu den Slow Control-Stationen (TX-Pfad), als auch deren korrekte Speicherung im RX-Pfad zu verifizieren.

Die Simulationen wurden mit einer unterschiedlichen Anzahl an Slow Control-Befehlen durchgeführt. Dabei trat Anfangs das gravierende Problem auf, dass manche Pakete komplett oder zum Teil nicht aus `bram_s_tx` gelesen wurden. Ebenfalls wurde festgestellt, dass stattdessen falsche Slow Control-Befehle an das entsprechenden Nathan-Modul gesendet wurden. Nach eingehender Analyse wurde festgestellt, dass es sich hierbei um einen Fehler im Ethernet-Core handelt. Nachdem ein Ethernet-Frame vollständig empfangen wurde aktiviert der Ethernet-Core normalerweise das Flag `crc_avail` für einen Takt. Dies ist unabhängig davon ob die übermittelte Prüfsumme korrekt war oder nicht. Anhand des Signalverlaufs-Diagramms konnte man sehen, dass bei manchen Paketen dieses Flag nicht aktiviert wird. Der Ethernet-Core leistet daraufhin ein Fehlverhalten dergestalt, dass Daten an falsche Adressen in `bram_s_tx` geschrieben werden, wobei zum Teil auch gültige, noch nicht gelesene Daten überschrieben werden können. Das Auftreten dieses Fehlers zeigt eine Abhängigkeit von der Größe der Ethernet-Frames, nicht jedoch von deren Inhalt. Bei einer Anzahl von 24–29 Slow Control-Kommandos pro STP-Paket wurde der Fehler nicht beobachtet. Da der Gigabit-Ethernet-Core nicht Teil der vorliegenden Arbeit ist und weil dieses Problem bislang nur in der Simulation und nicht in der realen Hardware auftrat, wurde das Python-Skript so modifiziert, dass die erzeugte Paketgröße in eben diesem Bereich liegt. Dies ist ausreichend um die Funktionalität des Sliding-

Slow Control-Befehle insgesamt	Anzahl beteiligter Nathan-Module	Fehler im TX-Pfad	Fehler im RX-Pfad
100	4	—	—
1.000	16	—	—
10.000	16	—	—
100.000	16	10.873 Befehle korrekt, danach kein Verschieben des Fensters möglich	

Tabelle 5.1: Simulationsergebnisse des Transfers von Slow Control-Befehlen.

Window-Algorithmus sowie des Datenpfades zu den Nathan-Modulen und zurück zu verifizieren. Die Ursache des Fehlers im Ethernet-Core wird derzeit untersucht.

In Tabelle 5.1 sind die Ergebnisse der Simulationen dargestellt. Die Anzahl der beteiligten Nathan-Module hängt dabei von der zufällig erzeugten Stimulidatei ab. Wie man der Tabelle entnehmen kann funktioniert der Datentransfer bis zu etwa 10.000 Schreibbefehlen einwandfrei. Danach tritt das Problem auf, dass das Modul `nathanbuffers`, welches nicht Teil dieser Arbeit ist, das Empfängerfenster des Sliding-Window-Algorithmus nicht mehr verschiebt und somit nicht mehr auf die nachfolgenden STP-Pakete zugegriffen werden kann. Bis zu diesem Zeitpunkt funktioniert das Gesamtsystem – insbesondere die im Rahmen dieser Arbeit konzipierte Logik – einwandfrei. Die Simulation, bei der der Fehler auftrat, wurde nur ein Mal durchgeführt, was durch die sehr lange Simulationszeit begründet ist, die hierfür nötig ist. Die genaue Ursache für dieses Fehlverhalten wird noch untersucht.

5.1.1.2 Performance des Systems

Dieser Abschnitt beschäftigt sich mit der Performance des Systems, also dem erreichbaren Datendurchsatz. Ziel der Implementierung ist es die durch die Gigabit-Ethernet-Technologie bereitgestellte Bandbreite von 1 GBit/s möglichst voll auszunutzen. In diesem Abschnitt wird dargelegt inwiefern dieses Ziel erreicht wurde. Es wird auch dargestellt, wo sich der Flaschenhals des Systems befindet. Bei der Diskussion werden die beiden Grenzfälle betrachtet bei denen nur mit einer Slow Control-Station oder mit allen 16 Nathan-Modulen kommuniziert wird[2].

Kommunikation mit einer Station Für das Laden der Felder `cmd` und `mod` in die Shiftregister-Matrix in `shifters_s_tx` werden *immer* 17 Takte benötigt. Aufgrund der Aufteilung der Stationen auf logische Blöcke (siehe Abschn. 3.1) ist bei der Kommunikation mit nur einer Station zu beachten, dass für das Laden der Felder `addr` und `data` nur acht Takte benötigt werden. Nachdem die Matrix geladen wurde, wird sie acht Takte lang geshiftet und stellt somit an ihrem Ausgang gültige Daten bereit. Daraus lässt sich errechnen, dass die Gesamtanzahl an Takten, die benötigt werden um ein Slow Control-Kommando zu serialisieren $(17 + 8) + 8 \cdot (8 + 8) = 153$ Takte beträgt. Da für ein Kommando 72 Bit (`cmd`, `mod`, `addr`, `data`) serialisiert werden müssen, ergibt sich ein maximaler Durchsatz von $72/153 = 0,47$ Bit/Takt. Aus der Arbeitsfrequenz von 125 MHz folgt für die Shiftregister-Matrix im TX-Pfad eine Datenrate von

$$d_{tx,single} = 0,47 \text{ Bit/Takt} \cdot 125 \text{ MHz} = 58,8 \text{ MBit/s}. \tag{5.1}$$

[2] „Kommunikation“ bezeichnet in diesem Abschnitt immer die Übermittlung von Schreib-/Lesebefehlen. Diese stellen im realen Betrieb bei Weitem die Mehrheit aller Befehle dar.

Obwohl dies gegenüber des alten Slow Control-Rings auf den ersten Blick keine überragende Steigerung darstellt ist diese Datenrate dennoch zufriedenstellend, wie die folgende Betrachtung zeigen wird. Es wird sich auch herausstellen, dass die Shiftregister-Matrix im TX-Pfad nicht den Flaschenhals darstellt.

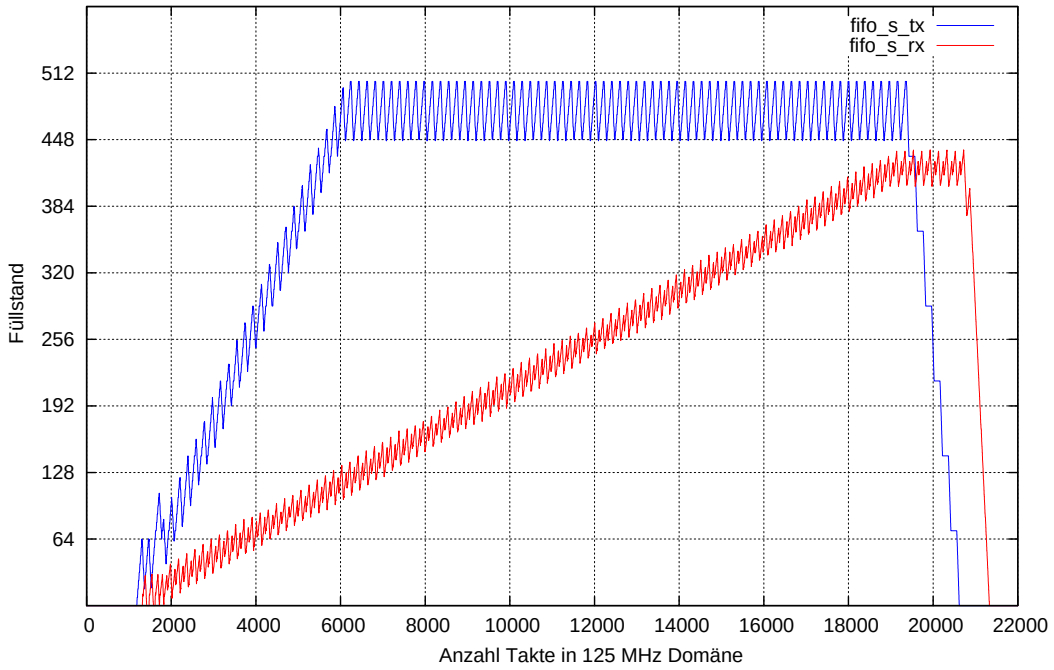
Um dies klarzumachen, soll zunächst überprüft werden, ob die Frequenz der physikalischen Anbindung der Nathan-Module an den Balu-FPGA ausreichend ist um diese Datenrate weiter zu transportieren. Hierbei sind zwei Punkte zu beachten. Zum einen muss nicht das gesamte Slow Control-Frame durch die Matrix serialisiert werden, sondern nur 72 Bit von insgesamt 100 Bit für ein Standardframe. Zum anderen muss beachtet werden, dass bei Schreib- und Lesezugriffen auf die Stationen entsprechend dem Slow Control-Protokoll immer zusätzlich zu dem Frame welches den eigentlichen Befehl enthält noch das Antwortframe des betreffenden Nathan-Moduls ein Mal komplett in einem der Minimalringe umlaufen muss. Aus der Sicht von `eth2sctrl_mac` bedeutet dies, dass für jeden Schreib-/Lesebefehl *zwei* Frames versendet werden müssen. Von diesen beiden Frames muss aber nur eines zum Teil durch die Shiftregister-Matrix im TX-Pfad. Berücksichtigt man noch die notwendigen Pausen, die zwischen zwei Slow Control-Frames eingefügt werde, und die entstehende Verzögerung durch die Register im Nathan-FPGA, so folgt, dass die physikalische Datenübertragung zu den Nathan-Modulen eine Datenrate von $100/72 \cdot 2,41 \cdot 58,8$ MBit/s $= 196,9$ MBit/s ermöglichen müsste um den Durchsatz der Shiftregister-Matrix voll auszunutzen. Da die reale Anbindung mit einer Frequenz von nur 156,25 MHz erfolgt wird ersichtlich, dass die Matrix in `shifters_s_tx` schnell genug Daten liefern kann.

Die Matrix im RX-Pfad wurde so konzipiert, dass der Shiftvorgang vorzeitig abgebrochen werden kann für den Fall dass nur mit wenigen Slow Control-Stationen kommuniziert wird (siehe Abschn. 3.3.3). Wenn nur an ein Nathan-Modul Slow Control-Befehle gesendet werden und dieses Modul auf Sockel 0 platziert wurde, so benötigt die Matrix 99 Takte um ein Frame vollständig zu deserialisieren. Diese Zahl wurde der Simulation entnommen und enthält alle Pausen, die wegen der Interaktion der FSMs in `shifters_s_rx` und wegen des notwendigen Inkrementierens von `wpointer` benötigt werden. Weil durch diese Matrix für jedes Slow Control-Kommando 84 Bit (`dest`, `src`, `cmd`, `mod`, `addr`, `data`, zwei Statusbits) deserialisiert werden, ergibt sich hieraus ein Durchsatz von $84/99 = 0,85$ Bit/Takt oder

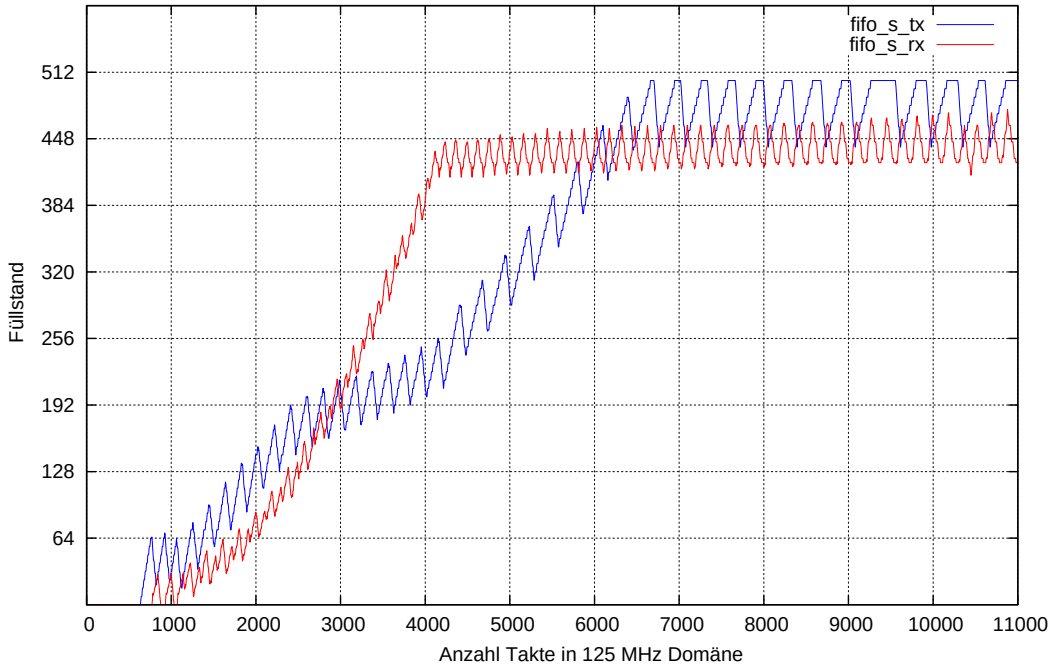
$$d_{rx,single} = 0,85 \text{ Bit/Takt} \cdot 125 \text{ MHz} = 106,1 \text{ Mbit/s}. \tag{5.2}$$

Dieser Wert soll nun wieder mit der Datenrate der physikalischen Anbindung in Beziehung gesetzt werden. Berücksichtigt man sämtliche Pausen die der MAC zwischen den Frames einfügen muss, so erhält man, dass die physikalisch Anbindung mit einer Frequenz von maximal $100/84$ Takte/Bit $\cdot\, 1,2 \cdot 106,1$ MBit/s $= 151,6$ MHz sinnvoll ist. Da die tatsächliche Frequenz etwas höher ist muss man erwarten, dass `fifo_s_rx` volllaufen wird und den MAC dazu zwingt Pausen zwischen den Frames bzw. zwischen Frame und anschließendem Token einzufügen. Somit stellt die Shiftregister-Matrix im RX-Pfad gerade so den Flaschenhals des Systems bei der Kommunikation mit nur einer Slow Control-Station dar.

Diese Überlegungen lassen sich durch Simulationsergebnisse stützen. Die zeitlichen Verläufe der Füllstände beider FIFOs sind in Abbildung 5.1(a) dargestellt. Man beachte dass für diese Simulation beide FIFOs im Gegensatz zur endgültigen Implementierung im FPGA mit einer Tiefe von 512 Worten implementiert wurden um den Verlauf der Füllstände deutlich zu machen. Am Ablauf der Logik ändert sich dadurch



(a) Zugriff auf ein einzelnes Nathan-Modul. Gesendet wurden 100 Slow Control-Schreibbefehle.



(b) Paralleler Zugriff auf alle Nathan-Module. Gesendet wurden 40 Schreibbefehle an jedes Modul.

Abbildung 5.1: Simulierter Verlauf der Füllstände der beiden FIFOs im TX- und RX-Pfad bei Schreibzugriffen auf die Nathan-Module. Zu Simulationszwecken wurde dabei die Größe beider FIFOs auf 512 Worte gesetzt.

nichts. Als Stimulus wurde ein Ethernet-Frame mit einem Initialisierungsbefehl und 100 Schreibbefehlen für das Nathan-Modul auf Sockel 0 gesendet. Man sieht, dass beide Füllstände tendenziell ansteigen, wobei der Anstieg für `fifo_s_rx` deutlich langsamer vonstatten geht. Dieser langsame Anstieg war zu erwarten, da die physikalische Anbindung der Slow Control-Stationen an den Balu-FPGA nur geringfügig zu schnell getaktet ist. Die Tatsache, dass auch der Füllstand von `fifo_s_tx` ansteigt bestätigt die Aussage, dass über die physikalische Anbindung die von der Matrix im TX-Pfad gelieferten Daten nicht schnell genug zu den Slow Control-Stationen transferiert werden können.

Nun soll noch berechnet werden, welche Netto-Datenübertragungsrate vom PC zu einem Nathan-Modul im Grenzfall der Kommunikation mit einer Slow Control-Station möglich ist, d.h. mit welcher Rate das 32-Bit Feld `data`, welches die Nutzdaten des Slow Control-Frames darstellt, transferiert werden kann. Da im TX-Pfad die physikalische Anbindung den Flaschenhals darstellt, ergibt sich eine Nettodatenrate von

$$\begin{aligned} d_{tx,single,netto} &= 156,25\ \text{MHz}/2,41 \cdot \frac{32}{100} \\ &= 20,8\ \text{MBit/s} \\ &= 2,60\ \text{MB/s}. \end{aligned} \tag{5.3}$$

Der Faktor 2,41 rührt von den Pausen zwischen zwei Frames her.
Im RX-Pfad wird die maximale Datenrate durch die Shiftregister-Matrix begrenzt. Die mögliche Nettodatenrate ergibt sich zu

$$\begin{aligned} d_{rx,single,netto} &= 106,1\ \text{MBit/s} \cdot \frac{1}{2} \cdot \frac{32}{84} \\ &= 20,2\ \text{MBit/s} \\ &= 2,53\ \text{MB/s}. \end{aligned} \tag{5.4}$$

Der Faktor 1/2 kommt dadurch zustande, dass zusätzlich zu dem Frame, welches die eigentlichen Daten transportiert auch das bestätigte Frame durch die Matrix deserialisiert werden muss.
Mit der alten Slow Control-Anbindung mittels der Darkwing-PCI-Karte waren experimentell ermittelte Nettodatenraten von bis zu 1,38 MB/s möglich [22]. Die Anbindung der Slow Control-Stationen an den eth2sctrl-Core ermöglicht somit eine etwa um den Faktor zwei gesteigerte Nettodatenrate.

Parallele Kommunikation mit allen Stationen In diesem Grenzfall muss die Matrix im TX-Pfad immer vollständig geladen werden und ermöglicht somit einen Durchsatz von $(16 \cdot 8)/(17 + 8) = 5,12$ Bit/Takt oder

$$d_{tx,multi} = 5,12\ \text{Bit/Takt} \cdot 125\ \text{MHz} = 640\ \text{MBit/s}. \tag{5.5}$$

Dies entspricht 40 MBit/s pro Nathan-Modul. Die physikalische Anbindung der Nathan-Module an den eth2sctrl-Core ist nun aber schnell genug um diese Daten zu verarbeiten, da lediglich eine Datenrate von $100/72 \cdot 2,41 \cdot 40$ MBit/s = 134 MBit/s benötigt wird. Die Matrix im RX-Pfad erlaubt bei der Kommunikation mit 16 Nathan-Modulen einen theoretischen Durchsatz von

$$d_{rx,multi} = \frac{16 \cdot 84}{204}\ \text{Bit/Takt} \cdot 125\ \text{MHz} = 823,5\ \text{MBit/s}. \tag{5.6}$$

Bei einer effektiven physikalischen Datenrate von 134 MBit/s pro Nathan-Modul und unter Berücksichtigung dass 84 Bit über die Matrix deserialisiert werden ergibt sich hieraus, dass der FIFO im RX-Pfad volllaufen wird. Man sieht dass der mögliche Durchsatz nicht weit von der theoretischen Obergrenze von 1 GBit/s entfernt ist. Daraus lässt sich schlussfolgern, dass die Matrix im RX-Pfad zwar auch in diesem Fall den performancelimitierenden Faktor darstellt, dass jedoch im Rahmen der verfügbaren Hardwareressourcen die seitens der Ethernet-Anbindung bereitgestellte Bandbreite weitestgehend ausgenutzt werden kann.

Auch für diesen Grenzfall wurde der Verlauf der Füllstände beider FIFOs in der Simulation überprüft. Die Stimuli-Datei enthielt dabei für jedes Nathan-Modul ein Initialisierungskommando sowie je 40 Schreibbefehle. Das Resultat der Simulation ist in Abbildung 5.1(b) gezeigt. Man sieht, dass der Füllstand von `fifo_s_rx` zunächst langsam, danach jedoch zunehmend immer schneller ansteigt. Dieses Verhalten lässt sich dadurch erklären, dass die 16 Ethernet-Frames nacheinander im Balu-FPGA eintreffen. Anfangs kann die Logik somit nur Daten für eine Slow Control-Station aus `bram_s_tx` auslesen und zur entsprechenden Station transferieren. Diese Situation entspricht also dem bereits diskutierten Szenario der Kommunikation mit einem Nathan-Modul. Im weiteren Verlauf treffen immer mehr Ethernet-Pakete im Balu-FPGA ein, wodurch mit mehreren Nathan-Modulen kommuniziert werden muss und somit der Shiftvorgang der Matrix im RX-Pfad zunehmend länger dauert, was einen steiler werdenden Anstieg des Füllstandes bewirkt.

Interessant ist auch der Verlauf des Füllstandes für `fifo_s_tx`. Hier lässt sich nach ca. 2.500 Takten feststellen, dass sich der Anstieg verlangsamt und schließlich nach ca. 4.000 Takten wieder deutlich steiler wird. Das Abflachen lässt sich durch die Aufteilung der Nathan-Module auf vier Blöcke erklären. Nach ca. 2.500 Takten erreicht ein Ethernet-Paket für Nathan-Modul #4 den Balu-FPGA und sorgt dafür, dass nun auch der zweite logische Block gültige Daten enthält. Ab diesem Zeitpunkt werden somit für das Laden der Matrix nicht mehr acht, sondern elf Takte benötigt. Die Folge hiervon ist, dass der Datenzufluss in `fifo_s_tx` abnimmt, wodurch sich das Abflachen erklären lässt. Der steile Anstieg nach 4.000 Takten ist eine Folge davon, dass `fifo_s_rx` zu diesem Zeitpunkt seinen programmierbaren Schwellwert erreicht. Dadurch muss `eth2sctrl_mac` Pausen zwischen den Slow Control-Frames einfügen und liest somit im Schnitt weniger Worte pro Takt aus `fifo_s_tx`.

Auch für diesen Grenzfall werden die Nettodatenraten betrachtet. Sowohl im TX- als auch im RX-Pfad ist der limitierende Faktor jeweils die entsprechende Shiftregister-Matrix. Die maximal erreichbaren Nettodatenraten ergeben sich zu

$$\begin{aligned} d_{tx,multi,netto} &= 640\ \text{MBit/s} \cdot \frac{32}{72} \\ &= 284,4\ \text{MBit/s} \\ &= 35,6\ \text{MB/s}. \end{aligned} \tag{5.7}$$

$$\begin{aligned} d_{rx,multi,netto} &= 823,5\ \text{MBit/s} \cdot \frac{1}{2} \cdot \frac{32}{84} \\ &= 156,9\ \text{MBit/s} \\ &= 19,6\ \text{MB/s}. \end{aligned} \tag{5.8}$$

Diese Zahlen zeigen, dass die Nettodatenübertragungsrate im Grenzfall der parallelen Kommunikation mit allen 16 Slow Control-Stationen auf den Nathan-FPGAs im Vergleich zur Slow Control-Anbindung über die Darkwing-PCI-Karte um mehr als eine Größenordnung gesteigert werden kann.

Es lässt sich abschließend festhalten, dass die durch den eth2sctrl-Core erreichbaren Datenraten durchaus zufriedenstellend sind. Insbesondere beim parallelen Betrieb mehrerer Nathan-Module lassen sich sehr gute Datenraten erreichen. Auf jeden Fall wurde die Performance im Vergleich zur Ringtopologie mit der Darkwing-Karte signifikant gesteigert.

5.1.2 Konfiguration der Nathan-FPGAs

Die Simulation des Konfigurationsvorgangs erfolgt indem in der Stimuli-Datei ein entsprechender Eintrag erstellt und ein Bitfile für die Konfiguration angegeben wird. Die Testbench erlaubt es die auf den einzelnen Nathan-FPGAs ankommenden Datenströme bitweise in Textdateien zu schreiben. Dabei wird mit der steigenden Flanke des Taktsignals `CCLK` gesampelt. Im Anschluss an die Simulation muss dann der so erhaltene Bitstrom mit der gesendeten .bit-Datei verglichen werden.

Zu Simulationszwecken wurde ein vollständiges Bitfile mit einer Größe von 565 kB versendet. Es wurde versucht dieses Bitfile an mehrere Nathan-FPGAs simultan zu senden. Das Ergebnis zeigt, dass die entwickelte Logik prinzipiell in der Lage ist den Transport des Bitstroms zu realisieren. Auch wenn die ausgehende Taktleitung angehalten werden muss liefert das System korrekte Ergebnisse.

Eine einwandfreie Konfiguration konnte jedoch erst durchgeführt werden, nachdem Parameter bezüglich des Sliding-Window-Algorithmus in der SystemC-Testbench modifiziert wurden: es handelte sich hierbei im wesentlichen um Parameter welche festlegen nach welcher Zeitspanne Pakete erneut gesendet werden sollen, falls kein Bestätigungsframe erhalten wurde. Wurde dieser Parameter zu niedrig gesetzt, so kam es bereits nach wenigen CFG-Paketen zu falsch übermittelten Daten im Bitstrom. Eine derartige Parameterabhängigkeit war nicht zu erwarten und sollte im finalen System auch nicht vorhanden sein. Die Ursache ist nicht vollständig klar, erste Untersuchungen deuten jedoch darauf hin, dass der Fehler im Steuerfluss des eth2sctrl-Cores, wahrscheinlich bei der Implementierung des Sliding-Window-Algorithmus, liegt.

5.2 Hardwaretest

Nachdem die Funktionalität des System anhand der funktionalen Simulation dargestellt wurde, soll nun auf die durchgeführten Hardwaretests eingegangen werden.

5.2.1 Slow Control-Kommunikation

Zunächst wurden Tests durchgeführt, die den korrekten Transfer von Befehlen zur Slow Control-Station im Balu-FPGA demonstrieren. Dazu wurde im FPGA ein Modul implementiert, welches es erlaubt die vier MSBs der Slow Control-Adresse und die acht LSBs der übertragenen Daten auf frei verfügbare Pins auf der Backplane zu legen. Die empfangenen Daten können dann mit Hilfe eines Logikanalysators verifiziert werden. Es wurden mehrere STP-Pakete an den Balu-FPGA gesendet und die empfangenen Daten verglichen. Die gesendeten Slow Control-Befehle wurden dabei mit einem Zufallszahlengenerator erzeugt. Da es sich bei diesem Test nur um eine prinzipielle Verifikation der Funktionalität des System handelt, wurden diese Tests nur mit einer geringen Anzahl an Befehlen (wenige Hundert) durchgeführt. Es traten hierbei keinerlei Fehler auf.

Slow Control-Befehle insgesamt	Anzahl STP-Pakete	Fehler
50	1	0
1.060	20	1 (Bitfehler)
10.798	200	2 (Bitfehler)
104.547	2.000	10 (Bitfehler)
1.041.884	20.000	138 (Bitfehler)

Tabelle 5.2: Ergebnisse der Hardwaretest zum Versand von STP-Paketen.

Um repräsentative Testergebnisse zu erhalten und auch die Anbindung der Nathan-Module an den Balu-FPGA überprüfen zu können, wurden anschließend Schreibtests auf das auf der Nathan-Platine befindliche SDRAM durchgeführt. Die Schreibdaten wurden mit Hilfe eines Python-Skripts zufällig erzeugt, die Adressen waren sequentiell. Das Skript erlaubt es, die Anzahl der STP-Pakete vorzugeben, die Anzahl der Slow Control-Befehle pro Paket ist zufällig. Da der Datenrücktransfer von `bram_s_rx` über die Ethernet-Verbindung zum PC zum Zeitpunkt der Tests noch nicht vollständig implementiert war, war es jedoch nicht möglich die Daten aus dem SDRAM anschließend wieder via Ethernet auszulesen. Nach dem Schreiben der Daten musste daher der Balu-FPGA rekonfiguriert werden, so dass anschließend die Verbindung zum PC über die SCSI-Schnittstelle und die Darkwing-PCI-Karte hergestellt werden konnte. Im Anschluss an das Auslesen der Daten wurden diese mit den geschriebenen Daten verglichen.

Ein erster Test wurde durchgeführt mit 50 Schreibbefehlen in einem STP-Paket. Dabei fiel auf, dass die Daten nur teilweise korrekt in das SDRAM geschrieben wurden, ca. 25 % aller Daten im SDRAM waren falsch. Auffällig dabei war die Regelmäßigkeit mit der die Fehler auftraten. Bei manchen Tests war exakt jedes vierte Datenwort falsch, bei anderen Tests gab es nur geringe Abweichungen von dieser Periodizität. Um den Grund für dieses Problem zu verstehen wurden der VHDL-Code so modifiziert, dass nach jedem gesendeten Slow Control-Frame eine Pause eingefügt wird. Hiermit konnten die 50 Schreibbefehle korrekt übermittelt werden. Als Ursache für dieses Verhalten scheiden somit Prüfsummenfehler und Fehler bezüglich der Phasenlage zwischen ausgehenden Takt- und Datensignalen aus, da sich damit eine derartige Regelmäßigkeit nicht erklären lässt. Da Schreibzugriffe auf die Slow Control-Station im Balu-FPGA erfolgreich durchgeführt werden können und diese Station vollkommen äquivalent zu den Slow Control-Stationen auf den Nathan-FPGAs behandelt wird, kommt auch ein prinzipieller Designfehler des eth2sctrl-Cores nicht in Frage. Vermutlich werden die Fehler durch einen Pufferüberlauf auf der Slow Control-Station im Nathan-FPGA verursacht. In der Simulation lässt sich dieses Verhalten nicht reproduzieren. Die folgenden Testreihen wurden alle mit einer erzwungenen Pause zwischen den Frames durchgeführt.

Tabelle 5.2 bietet einen Überblick über die durchgeführten Testreihen. Man sieht, dass es zu gelegentlichen Bitfehlern kommt, d.h. der Schreibbefehl kommt am Nathan-FPGA an und wird auch ausgeführt, jedoch ist beim anschließenden Auslesen des SDRAM-Inhalts ein Bit des Datenwortes falsch. Da ein erneutes Auslesen das selbe Ergebnis liefert, muss der Fehler beim Schreibzugriff auf das SDRAM entstehen. Falsch übermittelte Ethernet- oder Slow Control-Frames können als Fehlerursache ausgeschlossen werden, da beide Protokolle eine Fehlererkennung

beinhalten. Es liegt daher die Vermutung nahe, dass der Fehler durch die für den SDRAM-Zugriff verantwortliche Schreiblogik im Nathan-FPGA zustande kommt.

Diese Tests – im Kontext mit den durchgeführten Simulationen – zeigen, dass der eth2sctrl-Core in der Lage ist, die über die Ethernet-Verbindung gelieferten Daten korrekt in Slow Control-Frames zu verpacken und zu den Stationen zu transferieren. Es werden zwar sowohl in der Simulation als auch in der Hardwaremessung noch Fehler festgestellt, jedoch lässt sich dafür in beiden Fällen nicht die im Rahmen dieser Arbeit entwickelte Logik verantwortlich machen.

5.2.2 Konfiguration der Nathan-FPGAs

Um die Konfiguration des Nathan-FPGAs zu testen wurde ein Python-Skript erstellt, welches den Konfigurationsvorgang mittels eines Slow Control-Befehls an das Modul `nathanconfig` startet, aus einem vorgegebenen Bitfile Ethernet-Frames erzeugt und diese anschließend zur Backplane sendet. Hierzu wurde ein vollständiges Bitfile für einen Nathan-FPGA auf 1.129 CFG-Pakete aufgeteilt. Die CFG-Pakete wurden hierbei mit einer niedrigen Übertragungsrate zur Backplane transferiert, um gezielt das Anhalten des ausgehenden Taktsignals `CCLK` zu erzwingen. Dadurch konnte sämtliche Logik, die in Zusammenhang mit der Konfiguration steht, getestet werden. Im durchgeführten Test wurde der selektierte Nathan-FPGA erfolgreich konfiguriert.

Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, den bestehenden Gigabit-Ethernet-Core an das Slow Control-Interface anzubinden ohne das zugrunde liegende Token Ring-basierte Protokoll zur Ansteuerung der Nathan-Module zu modifizieren. Das Design wird zum Datentransfer zwischen einem PC und den Nathan-Modulen und zur Durchführung von Experimenten mit künstlichen neuronalen Netzen eingesetzt. Zudem wird die Steuerung des Konfigurationsablaufs der FPGAs auf den Nathan-Platinen von der entworfenen Logik durchgeführt. Daher stellt mit der Fertigstellung des Systems die Ethernet-Schnittstelle die einzige Anbindung der Backplane an die Außenwelt dar.

Vorgabe bei der Konzeption war es, die durch eine Gigabit-Ethernet-Schnittstelle zur Verfügung stehende Bandbreite von 1 GBit/s möglichst optimal auszunutzen. Auch musste auf eine ressourcenschonende Implementierung geachtet werden, um das endgültige Design in dem FPGA auf der Backplane platzieren zu können. Durch die Beibehaltung des Slow Control-Interfaces können die bestehenden, praxiserprobten Designs für den Nathan-FPGA unverändert weiterverwendet werden.

Um die durch die Gigabit-Ethernet-Technologie bereitgestellte Bandbreite ausnutzen zu können, wurde dabei die Schnittstelle zu den Nathan-Modulen parallelisiert. Die entwickelten und implementierten Module wurden in der Hardwarebeschreibungssprache VHDL für einen Virtex-II Pro FPGA als Zieltechnologie implementiert und in der vorliegenden Arbeit ausgiebig diskutiert. Das Ergebnis ist ein vollständig synthesefähiges Design für den Balu-FPGA, welches die angestrebten Anforderungen erfüllt. Das Design belegt etwa 84 % der Logikressourcen im FPGA und erlaubt es alle vorgegebenen Randbedingungen einzuhalten.

Die Funktionalität des Gesamtsystems wurde in intensiven funktionalen Simulationen bestätigt. Hierbei traten zwar teilweise Probleme auf, jedoch deuten die beobachteten Effekte auf eine Ursache in der Ethernet-Paketverwaltung hin, die nicht Teil dieser Arbeit ist. Es wurden auch Hardwaretests durchgeführt, die zeigen, dass das System in der Lage ist, Slow Control-Befehle aus Ethernet-Paketen zu extrahieren und auf den Nathan-Modulen auszuführen. Auch die Konfiguration der Nathan-FPGAs wird von der implementierten Logik korrekt durchgeführt.

Eine experimentelle Verifikation des Datenrückpfades steht noch aus. Dies liegt daran, dass die hierfür benötigte Logik in der MMU noch nicht vollkommen fertiggestellt ist. Auch eine Messung der erreichbaren Performance im realen Hardware-System konnte bislang nicht durchgeführt werde, da ein leistungsfähiges Software-Framework, welches es erlaubt die Bandbreite der Gigabit-Ethernet-Anbindung auszunutzen, bislang fehlt. Die Fertigstellung der letzten Teile sollte kurzfristig zu leisten sein. Somit wird es sehr bald möglich sein, das System ausgiebigen Testreihen zu unterziehen und die vollständige Funktionalität und Leistungsfähigkeit auch in der Hardware nachzuweisen.

Schlussletztendlich wird mit der entworfenen Logik die Abkehr von der proprietären Außenanbindung über die Darkwing-PCI-Karte und die Verwendung der Gigabit-Ethernet-Schnittstelle ermöglicht. Damit kann die Backplane als Experimentierplattform für großskalige, künstliche neuronale Netze bei Forschungspartnern innerhalb des FACETS-Konsortiums eingesetzt und über eine an internationalen Stan-

dards ausgerichtete Schnittstelle von beinahe jedem PC aus gesteuert werden. Durch die erreichbaren Datenraten, die eine Steigerung von bis zu mehr als einer Größenordnung gegenüber der SCSI-Anbindung darstellen, ist es nun möglich ausführliche Experimente, wie z.B. Parameter-Scans, schnell durchzuführen. Damit leistet die vorliegende Arbeit einen bedeutenden Beitrag auf einem der interessantesten und faszinierendsten Forschungsgebiete unserer Zeit, auf dem es sicherlich noch viel zu entdecken gibt.

A Ressourcenverbrauch

Die VHDL-Quellcodes sind unter der Revisionsnummer 5895 im Vision-SVN-Repository zu finden unter *project/fpgasystem_branches/juergen_r5401*. Der Ressourcenverbrauch wurde mit Xilinx ISE 9.2.04i bestimmt und bezieht sich auf die Ergebnisse des Platzierens & Verdrahtens. Dabei wurden nur Constraits für die Frequenzen der benötigten Taktsignale vorgegeben, das Optimierungsziel war 'Speed', alle Effort-Levels waren auf 'Standard'.

eth2sctrl_mac_top

Slices	1.198
LUTs	2.055
Slice FFs	708

shifters_s_tx

Slices	497
LUTs	759
Slice FFs	550

shifters_s_rx

Slices	355
LUTs	560
Slice FFs	471

fifo_s_tx

Breite	17	Bit
Tiefe	128	Worte
prog_empty_tresh	48	Worte
prog_full_tresh	112	Worte
Schreibtakt	125	MHz
Lesetakt	156,25	MHz
Slices	ca. 88	
LUTs	ca. 84	
Slice FFs	ca. 119	

fifo_s_rx

Breite	17	Bit
Tiefe	128	Worte
prog_full_tresh	43	Worte
Schreibtakt	156,25	MHz
Lesetakt	125	MHz
Slices	ca. 83	
LUTs	ca. 75	
Slice FFs	ca. 115	

fifo_cfg

Breite (Schreibdomäne)	8	Bit
Breite (Lesedomäne)	1	Bit
Tiefe (Schreibdomäne)	16	Worte
prog_full_tresh	11	Worte
Schreibtakt	125	MHz
Lesetakt	39,0625	MHz
Slices	ca. 59	
LUTs	ca. 83	
Slice FFs	ca. 85	

B Interfacelisten

eth2sctrl_mac_top

Port	*i/o*	*Bit*	*Beschreibung*
sr_clk	i	1	Takt für ausgehende/ankommende Slow Control-Frames; 156,25 MHz
cclk	i	1	Takt für Konfigurationslogik; 39 MHz
rst	i	1	Reset für interne Logik
cclk_obuf_t	o	1	Signal zum Anhalten des ausgehenden Taktes während Konfiguration
Interface zu shifters_s_tx			
fifo_s_tx_dout	i	17	Datenausgang von fifo_s_tx
fifo_s_tx_rd_en	o	1	read enable für fifo_s_tx
fifo_s_tx_prog_empty	i	1	programmierbares empty Flag; zeigt eth2sctrl_mac_top an, dass genug Daten in fifo_s_tx gespeichert sind um unterbrechungsfreies Senden zu garantieren
cfg_data_done	i	1	Statussignal für eth2sctl_config; zeigt an, dass alle CFG-Pakete aus bram_s_tx gelesen wurden
Interface zu shifters_s_rx			
fifo_s_rx_din	o	17	Daten zu fifo_s_rx
fifo_s_rx_prog_full	i	1	programmierbares full Flag von fifo_s_rx; zeigt an, dass der FIFO kein ganzes Frame mehr aufnehmen kann
fifo_s_rx_wr_en	o	1	write enable für fifo_s_rx
Interface zu sctrl_main auf Nathan/Balu			
din_sctrl	i	17	eingehende Slow Control-Datenleitung
dout_sctrl	o	17	ausgehende Slow Control-Datenleitung
FPGA_PROG	o	16	PROG Leitung zu allen Nathan-FPGAs
FPGA_INIT	i/o	16	INIT Leitung zu/von allen Nathan-FPGAs; bidirektional mit Pull-up
FPGA_DONE	i	16	DONE Leitung von allen Nathan-FPGAs
Interface zu nathanconfig			
cfg_switch	i	1	Statusflag das die Logik in Konfigurationszustand versetzt
PROGmask	i	16	Maske für PROG Leitungen; erlaubt selektive Programmierung einzelner Nathan-FPGAs
cfg_ok	o	1	Statusflag für erfolgreiche Konfiguration; beendet Konfigurationszyklus

cfg_err	o	1	Statusflag für gescheiterte Konfiguration; beendet Konfigurationszyklus
cfg_err_code	o	1	Statusflag; gibt Grund für Scheitern eines Konfigurationsvorgangs an
Interface zu fifo_cfg			
cfg_din	i	1	Datenausgang von fifo_cfg
fifo_cfg_rd_en	o	1	read enable für fifo_cfg
fifo_cfg_empty	i	1	empty Flag von fifo_cfg

shifters_s_tx

Port	*i/o*	*Bit*	*Beschreibung*
clk	i	1	Takt für interne Logik; 125 MHz
sr_clk	i	1	Takt für Lesezugriffe auf fifo_s_tx; 156,25 MHz
rst	i	1	Reset für interne Logik
Interface zu nathanbuffers			
nathan	o	5	fordere Speicherblocknummer für nächstes STP-Paket für die selektierte Slow Control-Station von nathanbuffers an
incr	o	1	verschiebe Empfangsfenster in nathanbuffers
bloc	i	6	Speicherblock für das nächsten Paket für die angefragten Station
nb_cmd	i	7	Anzahl der in diesem Block gespeicherten Slow Control-Befehle
ready	i	1	Flag das anzeigt, ob Daten gültig sind
incr_ack	i	1	Bestätigung für erfolgreiches verschieben des Empfängerfensters
Interface zu bram_s_tx			
bram_s_tx_raddr	o	16	Leseadresse für Port B von bram_s_tx
bram_s_tx_dout	i	8	Datenausgang von bram_s_tx
Interface zu eth2sctrl_mac_top			
fifo_s_tx_rd_en	i	1	read enable für fifo_s_tx
fifo_s_tx_prog_empty	o	1	programmierbares empty Flag von fifo_s_tx
fifo_s_tx_dout	o	17	Datenausgang von fifo_s_tx
cfg_data_done	o	1	Statussignal für eth2sctl_config; zeigt an, dass alle CFG-Pakete aus bram_s_tx gelesen wurden
Interface zu nathanconfig			
cfg_switch	i	1	versetzt interne Logik in Konfigurationszustand
Interface zu fifo_cfg			
fifo_cfg_wr_en	o	1	write enable für fifo_cfg
fifo_cfg_prog_full	i	1	programmierbares full Flag von fifo_cfg

shifters_s_rx

Port	*i/o*	*Bit*	*Beschreibung*
clk	i	1	Takt für interne Logik; 125 MHz
sr_clk	i	1	Takt für Schreibzugriffe auf fifo_s_rx; 156,25 MHz
rst	i	1	Reset für interne Logik
Interface zu eth2sctrl_mac_top			
fifo_s_rx_din	i	17	Dateneingang für fifo_s_rx
fifo_s_rx_prog_full	o	1	programmierbares full Flag von fifo-s_rx
fifo_s_rx_wr_en	i	1	write enable für fifo_s_rx
Interface zu bram_s_rx			
bram_s_rx_waddr	o	11	Schreibadresse für bram_s_rx
bram_s_rx_we	o	1	write enable für bram_s_rx
bram_s_rx_din	o	16	Dateneingang für bram_s_rx
Interface zu MMU			
bram_empty	o	1	empty Flag für bram_s_rx
bram_full	o	1	full Flag für bram_s_rx
wpointer	o	10	Zeiger auf das Ende gültiger Daten in bram_s_rx
wpointer_valid	o	1	gibt an ob wpointer stabil und gültig ist
rpointer	o	10	Zeiger auf den Beginn gültiger Daten in bram_s_rx
rpointer_incr	i	1	Inkrement-Signal für rpointer
Interface zu nathanconfig			
cfg_switch	i	1	versetzt interne Logik in Konfigurationszustand

Literaturverzeichnis

[1] Ashenden, Peter J.: *The Designer's Guide to VHDL.* Morgan Kaufmann Publishers, 2. Auflage, 2002.

[2] Becker, Joachim Philipp: *Ein FPGA-basiertes Testsystem für gemischt analog/digitale ASICs.* Diplomarbeit, Ruprecht-Karls-Universität Heidelberg, 2001.

[3] Borrelli, Chris: *IEEE 802.3 Cyclic Redundancy Check.* Xilinx, 2001. `http://www.xilinx.com/support/documentation/application_notes/xapp209.pdf`.

[4] Chu, Pong P.: *RTL Hardware Design Using VHDL.* John Wiley & Sons, 2006.

[5] devboards GmbH: *http://www.devboards.de/pdf/DBGIG1.pdf*. Vs. 1.0.1.

[6] Easics: *http://www.easics.com/*, 2008.

[7] Fast Analog Computing with Emergent Transient States (FACETS): *http://www.facets-project.org*, 2008.

[8] Grübl, Andreas: *Eine FPGA-basierte Plattform für neuronale Netze.* Diplomarbeit, Ruprecht-Karls-Universität Heidelberg, 2003.

[9] Grübl, Andreas: *VLSI Implementation of a Spiking Neural Network.* Doktorarbeit, Ruprecht-Karls-Universität Heidelberg, 2007.

[10] Gutmann, Christian: *Implementation einer Gigabit-Ethernet-Schnittstelle zum Betrieb eines Künstlichen Neuronalen Netzwerkes.* Diplomarbeit, Ruprecht-Karls-Universität Heidelberg, 2007.

[11] Hedtstück, Ulrich: *Einführung in die theoretische Informatik - formale Sprachen und Automatentheorie.* Oldenbourg, 3 Auflage, 2004.

[12] Institute of Electrical and Electronics Engineers: *ANSI/IEEE Std 802.5-1998E(R2003): Token ring access method and Physical Layer specifications*, 1998.

[13] Institute of Electrical and Electronics Engineers: *IEEE Std 802-2001: IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture*, 2002.

[14] Institute of Electrical and Electronics Engineers: *IEEE Std 802.3-2005: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications*, 2005.

[15] Institute of Electrical and Electronics Engineers: *EtherType Field listing*, 2008. `http://standards.ieee.org/regauth/ethertype/eth.txt`.

[16] Jolly, Olivier: *Implementation of a transport protocol in a FPGA*. Masterarbeit, Ruprecht-Karls-Universität Heidelberg, 2007.

[17] Network of Automated Library and Archives (NALANDA): *http://www.nalanda.nitc.ac.in/industry/appnotes/xilinx/documents/techdocs/5154.htm*, 2008.

[18] Open SystemC Initiative: *http://www.systemc.org/*, 2008.

[19] OpenCores: *http://www.opencores.org/*, 2008.

[20] Patterson, David A. und John L. Hennessy: *Rechnerorganization und -entwurf*. Spektrum Akademischer Verlag, Elsevier, 3. Auflage, 2005.

[21] Philipp, Stefan: *Slow Control and User Access*. Internal Report, 2007.

[22] Philipp, Stefan und Tillmann Schmitz: *Technical Report on the Nathan Backplane System*. Seminarvortrag, Januar 2005.

[23] Python Software Foundation: *http://www.python.org/*, 2008.

[24] Rey, Daniel Günther und Karl F. Wender: *Neuronale Netze*. Verlag Hans Huber, Hogrefe AG, 1. Auflage, 2008.

[25] Sinsel, Alexander: *Linuxportierung auf einen eingebetteten PowerPC 405 zur Steuerung eines neuronalen Netzwerks*. Diplomarbeit, Ruprecht-Karls-Universität Heidelberg, 2003.

[26] Tanenbaum, Andrew S.: *Computer-Netzwerke*. Wolfram's Fachverlag, 1. Auflage, 1992.

[27] Tanenbaum, Andrew S.: *Computer Networks*. Pearson Education, 4. Auflage, 2003.

[28] Thompson, Richard F.: *Das Gehirn*. Spektrum Akademischer Verlag, 2001.

[29] Walker, R. und D. Thomas: *A Model of Design Representation and Synthesis*. In: *Proceedings of the 22nd ACM/IEEE conference on Design automation*, Seiten 453–459, 1985.

[30] Xilinx: *Virtex-II Pro Platfom FPGA Handbook*, 2002.

[31] Xilinx: *Adder/Subtracter v7.0 Data Sheet*, 2003.

[32] Xilinx: *Block Memory Generator v2.6 Data Sheet*, 2007.

[33] Xilinx: *FIFO Generator v4.2 Data Sheet*, 2007.

[34] Xilinx: *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, 2007.

[35] Zimmermann, H.: *OSI Reference model - The ISO model of architecture for open systems intercommunications*. IEEE Transactions on Communications, 28(4):425–432, April 1980.

Danksagung

Ich möchte mich abschließend bei allen bedanken, die mich während meiner Arbeit unterstützt haben. Insbesondere gilt mein Dank:

- Herrn Prof. Dr. Karlheinz Meier, der mir die Möglichkeit gegeben hat an einem interessanten Forschungsprojekt teilzuhaben und meinen eigenen Beitrag dazu zu leisten.
- Herrn Prof. Dr. Udo Kebschull für seine zweite Meinung zur geleisteten Arbeit und dafür, dass er mit seiner Vorlesung über VHDL das Interesse an programmierbarer Logik bei mir geweckt hat.
- Herrn Dr. Johannes Schemmel für die freundliche Aufnahme und Führung während der letzten Monate.
- Dr. Stefan Philipp für die Ratschläge und Diskussionen, die mich vorangebracht haben, die Hilfe bei den aufgetretenen Hardware-Problemen und vor allem auch für das Korrekturlesen meiner Arbeit.
- Dan Husmann de Oliveira für die Einführung, Betreuung und Hilfe zu Beginn der Arbeit.
- Olivier Jolly für die freundliche Zusammenarbeit an einem gemeinsamen Ziel und die hilfreichen Erklärungen.
- Dr. Andreas Grübl für seine Hilfe bei aller Art von elektronischen Problemen.
- Allen anderen aktiven und ehemaligen Mitgliedern der Electronic Vision(s) Gruppe für die stets angenehme Arbeitsatmosphäre.
- Und nicht zuletzt meinen Eltern, die mich während des gesamten Studiums unterstützt haben.

Erklärung:

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 30.09.2008

_____

(Unterschrift)