Theorie
und Praxis
für Karrieren
von morgen

UNIVERSITÄT SIEGEN

Fachbereich Elektrotechnik und Informatik

Institut für Betriebssysteme und verteilte Systeme

Diplomarbeit

Erstellung eines hardwarenahen Systementwurfs zur Realisierung eines neuartigen, im Rahmen der Automatisierungstechnik echtzeitfähigen Ethernet-Switches

Samer Hijazi
Matr. Nr. : 539306

Erstprüfer: Prof. Dr. Roland Wismüller
Zweitprüfer: Dipl. Inf. Frank Dopatka M. Sc.

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Diplomarbeit allein angefertigt und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Samer Hijazi

Siegen, 26.09.2007

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In der Automatisierungstechnik werden heutzutage Ansätze des Echtzeit-Ethernets eingesetzt. Dadurch wird eine durchgängige Kommunikation von Büros bis Sensoren und Aktoren ermöglicht. Standard-Ethernet erfüllt in der verbreiteten Form nicht die speziellen Anforderungen der Automatisierungstechnik. Daher muss das Standard-Ethernet bestimmte Anpassungen unterzogen werden, um es echtzeitfähig zu machen. Eine Möglichkeit der Anpassung besteht in der Modifikation von Ethernet-Switches.

Auf dem Markt existieren echtzeitfähige Ethernet-Lösungen, wie Ethernet/IP, Ethernet PowerLink, ProfNet, SERCOS III und EtherCAT. Die ProfiNet-Lösung basiert auf einem modifizierten Switch. Es wird ein neuartiger Ansatz entwickelt, der die Eigenschaften der gängigen Echtzeit-Ethernet-Lösungen miteinander kombiniert. Das neue Verfahren bedient sich der Modifikation von Switches, um den Anforderungen der Automatisierungstechnik gerecht zu werden. Bei diesem Verfahren wird der Switch modifiziert, indem er mit einem Umschaltungsmechanismus (als RTC „Real Time Crossbar“ bezeichnet) und einem Scheduler versehen wird.

Der Umschaltungsmechanismus RTC, der an der Schnittstelle MII (Media Independent Interface) realisiert wird, kann zu Fehlern führen. Daher ist es erforderlich in Versuchen die Möglichkeiten der Realisierung zu analysieren, womit sich der erste Teil dieser Arbeit beschäftigt. Zur Lösung der aufgetretenen Fehler können beispielsweise VHDL-Module für FPGA-Bausteine entwickelt werden, was Thema des zweiten Teiles dieser Arbeit ist.

1.2 Aufgabenstellung

Um herauszufinden, ob der Umschaltungsmechanismus RTC realisierbar ist, soll zunächst im Rahmen dieser Arbeit die MII-Schnittstelle, auf der das RTC basiert, untersucht werden. Dafür sind die technischen Eigenschaften der MII-Schnittstelle zu untersuchen.

Des Weiteren soll ein Experiment durchgeführt werden, um festzulegen, ob die Realisierung des Umschaltungsmechanismus RTC möglich ist. Dafür sollen zunächst zwei PHY-Bausteine (Physical Layer Entity) direkt an der MII-Schnittstelle verbunden werden. Hieraus werden Erkenntnisse gewonnen, die die Ursache der fehlerhaften Verbindung zwischen zwei PHYs bzw. der fehlerhaften Übertragung der Daten darstellen.

Anhand der gesammelten Erkenntnisse sollen Lösungsansätze zur Behebung der aufgetretenen Probleme, die die Realisierung des Umschaltmechanismus RTC ermöglichen, vorgestellt werden. Danach sollen die Lösungsansätze für mögliche Umsetzung auf einem FPGA-Baustein moduliert

werden. Der gesamte Entwurf des modifizierten Switchs mit Umschaltmechanismus soll anhand eines Blockschaltplans erstellt werden.

1.3 Gliederung

Im nachfolgenden Kapitel erfolgt eine Einführung in notwendiges Basiswissen zum Thema Ethernet. Es findet eine detaillierte Beschreibung der MII-Schnittstelle statt, da sie als Basis für den Umschaltungsmechanismus RTC dient. Weiterhin wird im Kapitel 3 der Architektur-Ansatz des neu zu entwerfenden Echtzeit-Switches vorgestellt. Des Weiteren wird im Rahmen dieser Arbeit auf die Vorgehensweise eingegangen. Die Durchführung des Experiments an der MII-Schnittstelle, wobei zwei PHY-Bausteine, die für Codierung und Decodierung der übertragnen Daten zuständig sind, miteinander direkt verbunden werden (Kapitel 4). Ebenso werden die gewonnen Erkenntnisse und Lösungsansätze der aufgetretenen Probleme dargestellt. Die Implementierung der Lösungsansätze, die in der VHDL-Sprache und anhand von Simulationen realisiert wurde, und der Blochschaltplan des Echtzeit-Switchs werden im Kapitel 5.9 vorgestellt.

2 Grundlagen

Um ein besseres Verständnis über diese Arbeit zu erlangen, werden zu Beginn dieses Kapitels wichtige Grundlagen erklärt. Der Titel dieser Arbeit beinhaltet drei wichtige Begriffe, die hier erklärt werden. Zunächst wird auf die Struktur des Ethernets eingegangen, dabei wird besonderes Augenmerk auf die MII-Schnittstelle gerichtet. Des Weiteren werden die Funktion und die Architektur eines Ethernet-Switches dargestellt. Der letzte Abschnitt beschäftigt sich damit, wie das Ethernet angepasst wird, damit es die Anforderungen der Automatisierungstechnik erfüllt.

2.1 Ethernet

Ethernet ist eine Technologie, mit der in einem lokalen Netzwerk (LAN) Daten in Form von Pakete über ein Übertragungsmedium mit unterschiedlichen Geschwindigkeiten von 10, 100 und 1000 Mbit/s oder 10 Gbit/s übertragen werden können. Die Spezifikation der Ethernet-Technologie ist in der IEEE-Norm 802.3 „IEEE 802.3 Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications" standardisiert.

2.1.1 Struktur

Die Ethernet-Struktur in seiner Originalform besteht aus einem passiven Datenübertragungsmedium, des sogenannten „Ether", das von an ihm angeschlossenen Teilnehmern gemeinsam genutzt wird. Das Übertragungsmedium und die Teilnehmer repräsentieren ein Netz „Net" in Form einer Bus-Topologie. Daten können über das Ethernet im Halbduplex-Modus oder im Vollduplex-Modus übertragen werden. Die Übertragungsmodi werden im Kapitel 2.1.8 ausführlich diskutiert.

Aus Sicht des OSI-Referenzmodells (welches ein allgemeines Modell für die standardisierte Kommunikation zwischen Rechensystemen darstellt) umfasst das Ethernet die Bitübertragungsschicht und den unteren Teil der Sicherungsschicht, welcher MAC (Media Access Control) genannt wird.

Die nachfolgende Abbildung 1 repräsentiert das OSI-Referenzmodell. Die Bitübertragungsschicht besteht aus den Unterschichten „Reconciliation", „MII – Media Indepedet Interface", „PHY" und „MDI – Media Dependet Interface". Die Sicherungsschicht besteht aus den Unterschichten „MAC – Media Access Control", „MAC Control" und „LLC – Logical Link Control". Diese Schichten sind anhand der Abbildung 1 zu sehen und werden in den Kapitel 2.1.3 bis 2.1.7 ausführlich erläutert. In Rahmen dieser Arbeit wird nur auf die 100 Mbit/s Ethernet-Variante angegangen.

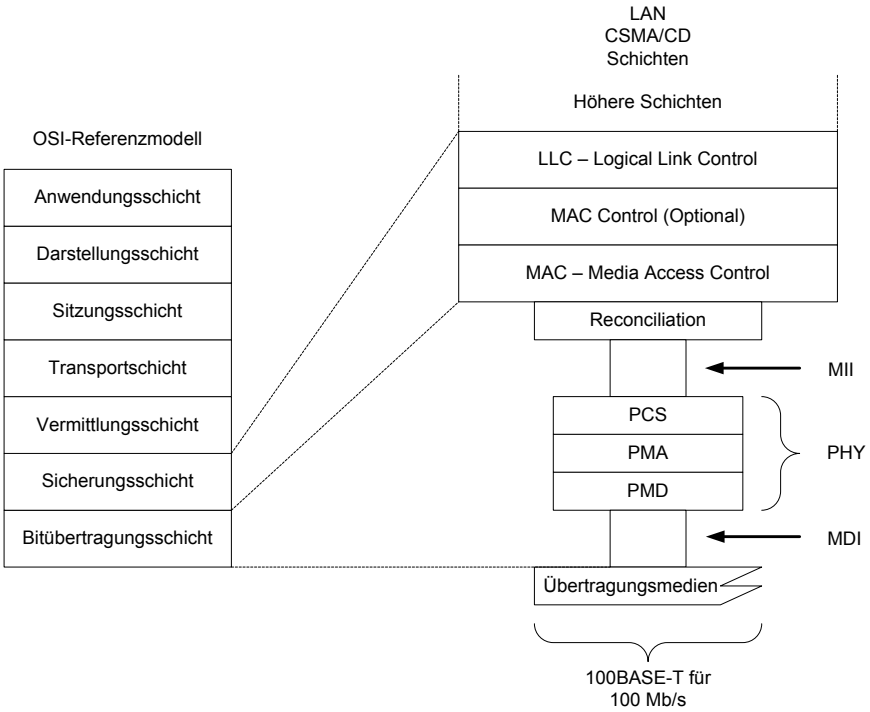

Abbildung 1: Ethernet in dem OSI-Referenzmodell [03]

2.1.2 Ethernet-Paket

Die über das Ethernet übertragenen Daten werden durch den Standard IEEE 802.3 [03] in zwei Formaten definiert, zum einen „IEEE 802.3.3.1.a Basic MAC Frame“ und zum anderen „IEEE 802.3.3.1.b Tagged MAC Frame“. Ein Ethernet-Paket besteht aus einer bestimmten Anzahl von Abschnitten, welche Präambel, SFD (Start Frame Delimiter), die Ziel-Adresse, die Quell-Adresse, Typ, Daten, PAD und FCS (Frame Check Sequence) beinhalten. Die folgende Abbildung 2 weist die Felder auf und gibt zugleich ihre jeweilige Größe in Byte an. Ein Ethernet-Frame kann ohne die Präambel und das SFD, eine Größe von 64 bis 1518 Bytes haben [33].

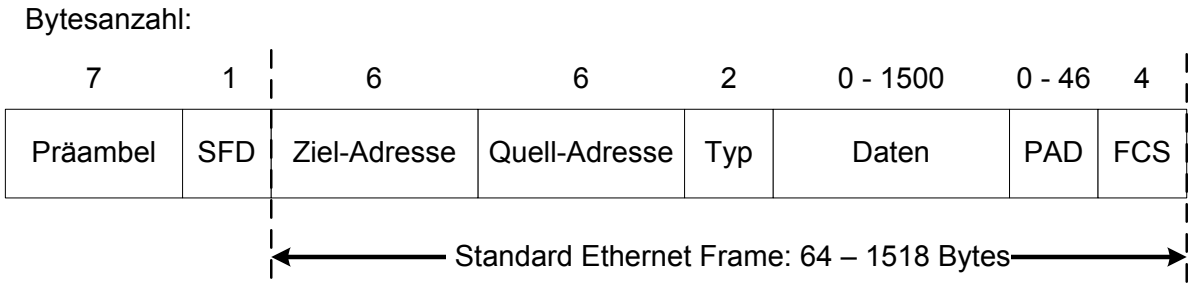

Abbildung 2: Ethernet-Frame [nach 33]

Jedes Ethernet-Paket beginnt mit einer Präambel, die aus sieben Byte besteht. Sie dient zur zeitlichen Synchronisation der Netzwerkgeräte. Eine Präambel hat das Hex-Muster “0x55“. Die Präambel wird bei Erreichung eines Teilnehmers von dem Ethernet-Paket entfernt, und wird zum Paket-Anfang beim Senden des Pakets hinzugefügt [33].

Das SFD „Start of Frame Delimiter“-Feld folgt der Präambel und erfüllt die Aufgabe, den Anfang des Frames zu markieren. Das SFD-Feld ist ein Byte lang und hat das Hex-Muster „0xD5“, das dem folgendem Bit-Muster entspricht: ´10101011´. Beim Senden eines Pakets wird das SFD-Feld hinzugefügt. Beim Empfang des Frames wird das SFD-Feld entfernt. Das Hinzufügen als auch die Entfernung der Präambel und des SFD findet in der MAC-Unterschicht statt [33].

Das Zieladresse-Feld folgt dem SFD-Feld und definiert den Zielrechner, an den das Paket gesendet wird. Die Zieladresse ist sechs Bytes lang und anhand des ersten übertragenen Bits wird durch die MAC-Unterschicht entschieden, ob es sich um eine Einzeln-/Unicast- oder Gruppen-/Multicast-Adresse handelt [33].

Das Quelladresse-Feld folgt dem Zieladresse-Feld und definiert den Quellrechner, aus dem das Paket gesendet wird. Die Adresse ist sechs Byte lang und wird von der MAC-Unterschicht nicht ausgewertet [33].

Nach der Quelladresse folgt das Typ- bzw. Längen-Feld, das je nach Inhalt unterschiedlich ausgewertet wird. Das Feld ist zwei Byte lang und wird als Typ-Feld ausgewertet, wenn sein Wert gleich oder größer als $0x600_{Hex}$ ist. Andernfalls wird es als Längen-Feld ausgewertet, und gibt die Anzahl der Bytes in dem darauffolgenden Daten-Feld an [33].

Das Daten-Feld kann eine Größe von minimal 0 Byte bis maximal 1500 Byte haben und folgt nach dem Typ-/Längen-Feld. Das Feld enthält die eigentlichen Informationen, die übertragen werden sollen [33].

Das PAD-Feld kann eine Größe von minimal 0 Byte bis maximal 46 Byte haben. Es wird eingesetzt, um zu gewährleisten, dass eine minimale Größe eines Frames auf 64 Byte (Ohne Präambel und SFD) erreicht wird [33].

Um Übertragungsfehler zu erkennen, wird eine 32-Bit CRC (Cyclic Redundancy Check) über die Felder Ziel-Adresse, Quell-Adresse, Typ-Feld, Daten-Feld und das PAD-Feld berechnet. Die Präambel und SFD werden nicht mit berechnet. Der Wert der Berechnung von CRC wird im FCS (Feld Check Sequenz) abgelegt [33].

Der VLAN Ethernet Frame unterscheidet sich von dem Standard Ethernet Frame durch das zusätzliche VLAN-Feld. Das vier Byte VLAN-Feld, das der Quell-Adresse folgt, ermöglicht auf einem physikalischen Medium bis zu 4096 virtuelle lokale Netzwerke (VLANs). Die nachfolgende Abbildung 3 zeigt den Aufbau des VLAN-Frames [33].

Die Frames werden über das Ethernet-Medium in einem zeitlichen Sicherheitsabstand, der sogenannte „InterFrame Gap“, gesendet. Dabei wird gesichert, dass der von IEEE 802.3 definierte „InterFrame Gap“ eine Bit-Zählung von minimal 96 Bit ist. Zwischen zwei gesendeten Frames vergehen bei den verschiedenen Ethernet-Varianten (9,6 µs bei 10 Mbps, 0,96 µs bei 100 Mbps und 96 ns bei 1 Gbps)[33]. Einige Hersteller haben bei ihren Produkten diesen Abstand

verkürzt, um die Performance zu steigern. Dies kann jedoch zu Kompatibilitätsproblemen führen.

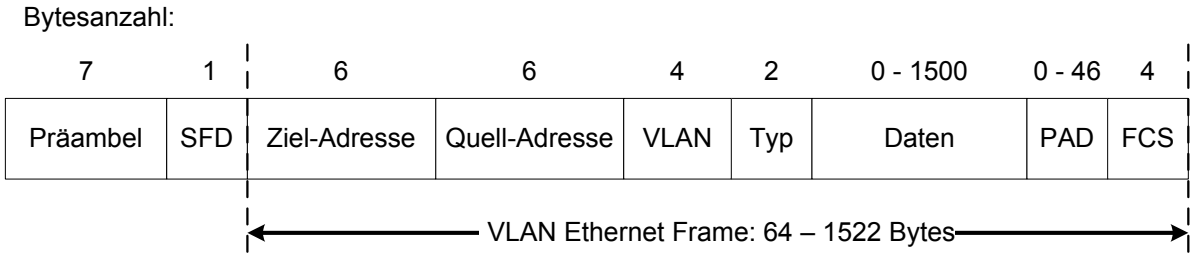

Abbildung 3: Tagged Ethernet-Frame [nach 33]

2.1.3 Medium Dependent Interface (MDI)

Die MDI (Medium Dependent Interface) ist eine physikalische und elektrische Schnittstelle, die eine Verbindung zwischen dem Übertragungsmedium und dem PHY herstellt. Die MDI ist bei den unterschiedlichen 802.3-Ausprägungen unterschiedlich aufgebaut. Bei den Ethernet-Medientypen wie z.B. 10BASE-T und 100BASE-TX wird der RJ45-Stecker und bei 100BASE-FX MCI-Stecker eingesetzt [08]. Bei der 100BASE-TX Variante gehören die Signale „TX+“, „TX-“, „RX+“ und „RX-“ zur MDI.

2.1.4 Physical Layer Entity (PHY)

Der PHY, der als Ethernet-Transceiver bezeichnet wird, hat die Funktionalitäten Empfangen, Senden und die Verwaltung der kodierten Signale, die über das physikalische Übertragungsmedium übertragen werden. Der PHY ist über die MDI-Schnittstelle mit dem Übertragungsmedium und über die MII-Schnittstelle mit der Sicherungsschicht verbunden. Bei den 100 Mbit/s Ethernet besteht der PHY aus den Schichten PCS (Physical Coding Sublayer), PMA (Physical Medium Attachment) und PMD (Physical Medium Dependent). Im Folgenden werden diese Teilschichten genauer erläutert.

Die PCS-Teilschicht codiert und decodiert Ethernet-Pakete in 4B/5B- bzw. 8B/6T-Codegruppen, außerdem generiert sie die Signale „Carrier-Sense-Signal“ und „Collision-Detect-Signal“ [06]. Die Bedeutung der Signale „Carrier-Sense-Signal“ und „Collision-Detect-Signal“ wird in Abschnitt 2.1.5 ausführlich erläutert.

Die PMA-Teilschicht passt die 4B/5B-Codegruppen der NRZI-Codierung an, bei der eine logische ´1´ mit einem Wechsel des Signalpegels verbunden ist, eine logische ´0´ hingegen keinen Wechsel hervorruft. Aufgrund der Limitierung der Länge der Nullen und Einsen – eine der wesentlichen Eigenschaften der 4B/5B-Codegruppen – entstehen keine lange Nullen-Sequenzen, infolgedessen der Empfänger stets über ausreichende Taktinformationen verfügt [09]

In der PMD-Teilschicht werden physikalischen Eigenschaften beschrieben, Übertragungs-Parameter definiert (z.B. die Übertragungsgeschwindigkeit, die bei 10BASE-T bei 20MHz und bei 100BASE-TX bei 125MHz liegt), und die passende Signalisierung zum Übertragungsmedium realisiert [10].

Die nachfolgende Abbildung 4 repräsentiert die Architektur eines PHYs von 100 BASE-T, mit den Varianten 100BASE-T4, 100BASE-FX, 100BASE-TX und 100BASE-T2 in Bezug auf die drei Schichten des PHYs. In der PCS-Ebene werden die verwendeten Kodierungsverfahren der entsprechenden Ethernet-Varianten dargestellt.

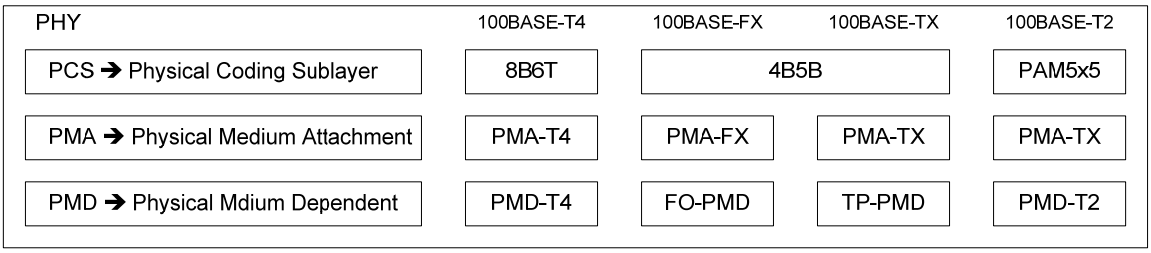

Abbildung 4: PHY der verschiednen Ethernet-Varianten [nach 08]

2.1.5 Media Independent Interface (MII)

Die MII (Media Independent Interface) ist eine Schnittstelle, die eine einfache und leichte zu implementierende Verbindung zwischen der MAC-Unterschicht der Sicherungsschicht und den PHY für Datenübertragung im 10 Mbit/s und 100 Mbit/s Ethernet darstellt. Die MII-Schnittstelle ist in der IEEE 802.3 Klausel 22 definiert. Ein Schwerpunkt dieser Arbeit besteht in der Auseinandersetzung mit der MII-Schnittstelle. Deshalb werden wesentliche Charakteristiken der Schnittstelle ausführlich dargelegt.

Laut dem IEEE 802.3 [03] Standard kann die MII-Schnittstelle in drei Applikationen implementiert werden. In der ersten Applikation liegt die Schnittstelle zwischen zwei Bausteinen auf einer Leiterplatte. Eine Implementierung dieser Applikation ist z.B. eine Netzwerkkarte-Implementierung mit PHY-Baustein und MAC-Baustein, wobei die MII-Schnittstelle die Verbindung zwischen den beiden Bausteinen herstellt. In der zweiten Applikation stellt die MII-Schnittstelle die Verbindung zwischen zwei oder mehreren Leiterplatten mit einem geeigneten Stecker her. Als dritte Applikation kann die MII-Schnittstelle als Verbindung zwischen zwei Leiterplatten über ein Kabel, das eine maximale Länge von 0,5 m beträgt, mit dem geeigneten Stecker eingesetzt werden.

Die MII-Schnittstelle teilt sich in zwei Schnittstellen, die erste ist die Empfang-Schnittstelle (engl.: Receive-Interface) und die zweite ist die Sende-Schnittstelle (engl.: Transmit-Interface). Das Receive-Interface stellt eine Schnittstelle für die aus dem PHY empfangenen Daten dar. Die Daten, die aus dem MAC zum PHY gesendet werden, werden über das Transmit-Interface übertragen. Das Receive-Interface hat die Signale „RX_CLK“, „RX_DV“ „RX_D[3:0]“ und „RX_ER“,

das Transmit-Interface hat die Signale „TX_CLK“, „TX_DV“, „TX_D[3:0]“ und „TX_ER“. Somit besitzt jede Schnittstelle sieben Signale. Die Signale „RX_D[3:0]“ und „TX_D[3:0]“ stellen ein Bündel dar, und werden als Nibble bezeichnet. Die nachfolgende Tabelle 1 enthält die gesamten Signale der MII-Schnittstelle und deren Erklärungen.

Signal-Name	Erklärung
RX_CLK „*Receive Clock*“	Ist ein kontinuierliches Takt-Signal, das vom PHY erzeugt wird. Die aus dem PHY ankommenden Signale „RX_DV“, „RX_D[3:0]“ und „RX_ER“ werden in Beziehung zum Signal „RX_CLK“ empfangen.
RX_DV „*Receive Data Valid*“	Das Signal „RX_DV“ gibt an, ob die Daten, die an den Signalen „RX_D[3:0]“ anliegen, gültige Daten sind.
RX_D[3:0] „*Receive Data*“	Die aus dem PHY empfangenen Daten werden über die Signale „RX_D[3:0]“ dem MAC übergeben.
RX_ER „*Receive Error*“	Das Signal „RX_ER“ wird vom PHY gesteuert und signalisiert, ob ein fehlerhaftes Paket im PHY empfangen bzw. dekodiert wurde.
TX_CLK „*Transmit Clock*“	Ist ein kontinuierliches Takt-Signal, das vom PHY erzeugt wird. Die aus dem MAC erzeugten Signale „TX_EN“, „TX_D[3:0]“ und „TX_ER“ werden in Beziehung zum Takt-Signal „TX_CLK“ übertragen.
TX_EN „*Transmit Enable*“	Das Signal „TX_EN“ signalisiert, ob die aus dem MAC kommenden Daten, die an den Signalen „TX_D[3:0]“ liegen, gültig sind.
TX_D[3:0] „*Transmit Data*“	Die aus dem MAC erzeugten Daten werden über die Signale „TX_D[3:0]“ dem PHY übergeben.
TX_ER „*Transmit Error*“	Das Signal „TX_ER“ wird vom MAC erzeugt und signalisiert dem PHY, ob der zu sendende Frame als fehlerhafter Frame weiterhin übertragen wird oder nicht.
COL „*Collision Detect*“	Das Signal „COL“ signalisiert, ob eine Kollision während der Übertragung eines Frames stattgefunden hat.
CRS „*Carrier Sense*“	Das Signal „CRS“ signalisiert, ob das Medium vom anderen Teilnehmer benutzt wird bzw. ob das Medium frei ist.
MDC „*Management Data Clock*“	Ist ein kontinuierliches Takt-Signal, das in den PHY gespeist wird. Daten über das Signal „MDIO“ werden in Beziehung zum „MDC“ übertragen
MDIO „*Management Data Input/Output„*	Das Signal „MDIO“ ist ein bidirektionales Signal, das in den PHY gespeist wird. Über das Signal „MDIO“ werden Daten übertragen, um den PHY-Baustein zu konfigurieren.

Tabelle 1: MII-Signalen [nach 19]

In den nachfolgenden Abschnitten wird auf die physikalischen und logischen Eigenschaften der MII-Schnittstelle eingegangen. Des Weiteren werden die zeitlichen Anforderungen an die MII-Signale vorgestellt.

2.1.5.1 Physikalische Eigenschaften der MII-Signale

Da sich der Hauptbestandteil dieser Arbeit mit der MII-Schnittstelle beschäftigt, ist es von großer Bedeutung die physikalischen Eigenschaften der MII-Signale zu kennen. Man erhält dadurch Informationen über das Verhalten der Signale in der Waveform-Darstellung. Diese Informationen sind sehr hilfreich für die im Kapitel 4 durchgeführte Analyse der MII-Signale anhand eines Logik-Analyser.

Die Takt-Signale „TX_CLK“ und „RX_CLK“ steuern den Übertragungsprozess auf die MII-Schnittstelle. Die Frequenz der Takt-Signale ist im IEEE 802.3 Klausel 22.7.3.2 [03] festgelegt. Die Frequenz der Takt-Signale beträgt 25% der Übertragungsgeschwindigkeit der Ethernet-Variante. Dies bedeutet, dass z.B. bei 10 Mbit/s eine Frequenz von 2,5 MHz und bei 100 Mbit/s eine Frequenz von 25 MHz vorliegt.

Laut IEEE 802.3 werden die Signale „TX_ER“, „TX_EN“ und „TX_D[3:0]“ in Beziehung zum Signal „TX_CLK“ aus dem MAC an die MII-Schnittstelle übertragen. Bei Übertragung eines Frames wird das Signal „TX_EN“ mit dem ersten gesendeten Nibble auf ´1´ gesetzt und behält diesen Wert ´1´ bis das letzte Nibble übertragen ist. Anschließend wird das Signal „TX_EN“ auf ´0´ gesetzt, wobei der Wertwechsel des Signals „TX_EN“ (von ´1´ nach ´0´) vor der nachfolgenden aktiven Flanke des Takt-Signals „TX_CLK“ stattfinden muss. Dies gewährleistet, dass das letzte Nibble nicht durch den nachfolgenden Takt ausgewertet wird. Würde das letzte Nibble durch den nachfolgenden Takt erkannt, führt dies zu einem Fehler (Beispiel: CRC-Fehler oder Frame-Größe-Fehler). Die nachfolgende Abbildung 5 zeigt ein Frame, welcher mit der Präambel beginnt und mit CRC endet. Hierbei wird die Beziehung zwischen den Signalen „TX_EN“ und „TX_D[3:0]“ im Hinblick auf das Takt-Signal „TX_CLK“ verdeutlicht. Zugleich sieht man wie das Signal „TX_EN“ auf ´1´ am Anfang bzw. ´0´ am Ende des Frames gesetzt wird.

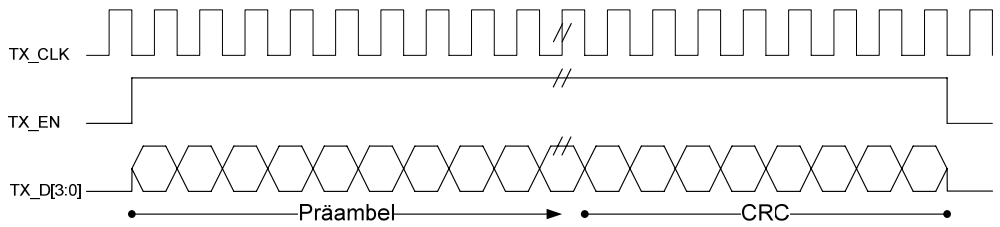

Abbildung 5: Beziehung zwischen "TX_CL", "TX_EN" und "TX_[3:0]" [nach 03]

Sowohl das Signal „RX_DV“ als auch die Daten am Bündel-Signal „RX_D[3:0] sollen in Beziehung zum „RX_CLK“ übertragen werden. Das IEEE 802.3 definiert das Signal „RX_DV“ als das vom PHY gespeiste Signal, welches Aufschluss darüber gibt, ob der PHY gültige Daten wiederhergestellt bzw. dekodiert hat. Das Signal „RX_DV“ verweilt auf ´1´ vom ersten bis zum letzten

wiederhergestellten Nibble. Ist das letzte Nibble des Frames aus dem PHY wiederhergestellt, so wird das Signal „RX_DV“ zugleich auf ´0´ gesetzt, jedoch muss dies vor dem nächsten aktiven Takt-Signal „RX_CLK“ erfolgen. Bei der Herausfilterung der Taktinformationen im PHY-Baustein können gewisse Teile der Präambel verloren gehen [14][15]. Damit die MAC-Unterschicht einen Frame mit fehlenden Teilen der Präambel fehlerfrei interpretieren kann, muss der Frame mindestens das SFD beinhalten. Das Signal „RX_DV“ muss in Beziehung zu den erkannten Teilen auf ´1´ gesetzt sein. Die nachfolgende Abbildung 6 zeigt die möglichen Fälle, in denen ein Frame seitens des MACs erkannt wird. Wie anhand der Abbildung 6 zu sehen ist, kann ein Frame mit drei, zwei, einem oder keinem Teil der Präambel zum MAC gesendet und zugleich erkannt werden. Jedoch muss das SFD vollständig im Frame enthalten sein. Das Signal „RX_DV“ ist in Bezug auf die fehlenden Stellen der Präambel auf ´0´ gesetzt. Die Präambel-Teile sind irrelevant für den MAC jedoch sehr wichtig für den PHY.

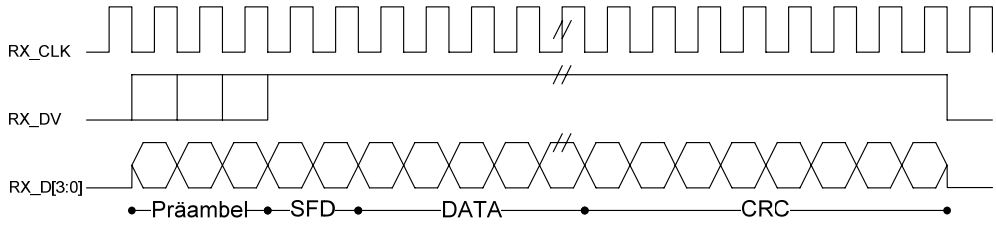

Abbildung 6: Beziehung zwischen "RX_CLK", "RX_DV" und "RX_D[3:0]" [nach 03]

Wenn das Signal „RX_ER“ in Bezug zu bestimmten Teilen des Frames auf ´1´ gesetzt ist, deutet dies darauf hin, dass das PHY an den entsprechenden Teilen Fehler entdeckt hat. Diese Stellen werden durch den MAC korrigiert oder verworfen. Der Wert des Signals „RX_ER“ hat ebenso wie der Wert des Signals „TX_ER“ keinerlei Einfluss darauf, wie das Paket auf die MII-Ebene interpretiert bzw. übertragen wird. Jedoch muss der Wert zum MAC bzw. zum PHY weitergeleitet werden. Das Signal „TX_ER“ kann an bestimmten Stellen im Frame auf ´1´ gesetzt sein, während das Signal „TX_EN“ auch auf ´1´ gesetzt ist. Dies bedeutet für den PHY, dass er die entsprechenden Stellen fehlerhaft weiterleiten soll, infolgedessen der Empfänger diese Stellen als fehlerhaft erkennt. Dies kann sinnvoll beim Einsatz von Repeatern oder einer Verbindung zwischen PHYs auf die MII-Ebene (welches Kern dieser Arbeit ist) sein, da auf der MII-Ebene keine Fehler-Behandlung durchgeführt werden kann. Durch diese Signalkombination können fehlerhafte Pakete durch die MII-Ebene weitergeleitet werden, und somit von End-Ziel-Teilnehmer behandelt werden.

Die nachfolgende Abbildung 7 veranschaulicht ein Frame, der eine RX-Übertragung als auch eine TX-Übertragung darstellt. Hierbei ist die Stelle im Frame fehlerhaft, wo das Signal „TX_ER“ bzw. „RX_ER“ den Wert ´1´ hat. Diese Stelle, die mit „XX“ in der Abbildung 7 markiert ist, hat keinen Einfluss darauf, wie das Paket auf die MII-Ebene übertragen wird. Jedoch müssen diese Signale zum PHY für Propagieren gesendet bzw. zum MAC für Korrigierung der Fehler gesendet werden.

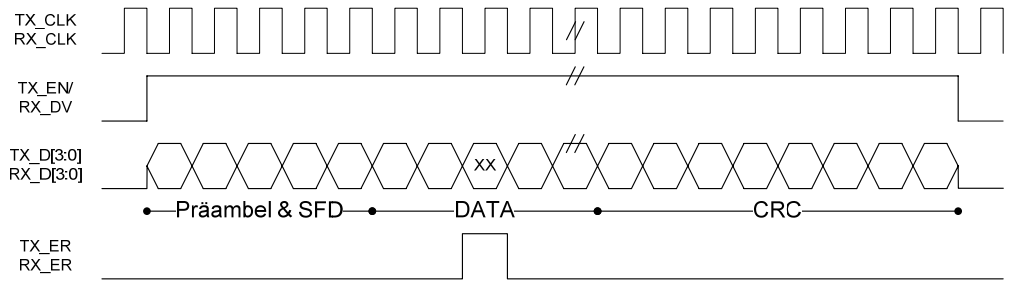

Abbildung 7: Ethernet-Frame mit fehlerhafter Stelle [nach 03]

2.1.5.2 Logische Eigenschaften der MII-Signale

In diesem Abschnitt werden die Eigenschaften der MII-Signale innerhalb der digitalen logischen Ebene erläutert. Die Eigenschaften der Signale auf dieser Ebene sind sehr nützlich zur Regenerierung der Präambel und des SFDs. Im Rahmen dieser Arbeit werden die Präambel und das SFD zur Realisierung des RTCs rekonstruiert (Kapitel 5). Die Informationen in diesem Abschnitt stellen Grundlage für die Implementierung im Kapitel 5 dar.

Die Bytes eines Pakets werden in Form von Nibbles übertragen. Die entsprechenden Nibbles werden durch die MII-Signale „RX_D[3:0]“ bzw. „TX_D[3:0]“ übertragen. Das IEEE 802.3 beschreibt diesen Prozess anhand der nachfolgenden Abbildung 8.

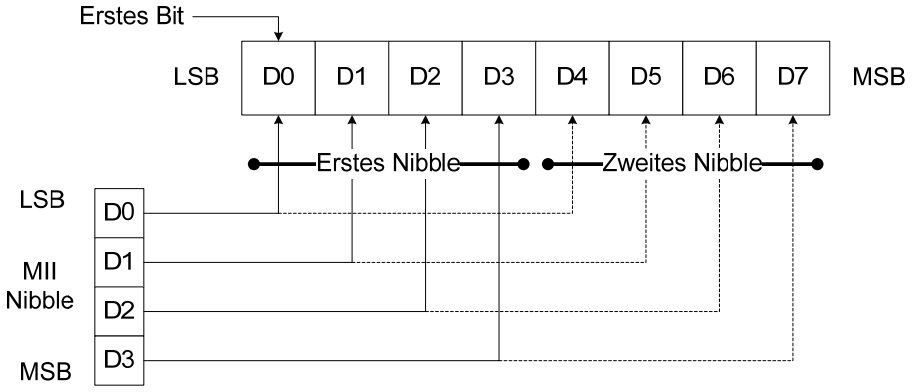

Abbildung 8: Übertragung von Bytes als Nibbles [nach 03]

Ein Byte wird beim Empfangen und beim Senden als zwei Nibbles zu je einem bestimmten Zeitpunkt übertragen, so dass die Bits eines Bytes von 0 bis 3 den Bits des ersten Nibbles von 0 bis 3 entsprechen und die Bits des Bytes von 4 bis 7 den Bits des zweiten Nibbles von 0 bis 3 entsprechen. Zum Beispiel wird bei einem 100 MBit/s Ethernet-Variante ein Nibble alle 40 ns übertragen. Der MAC beginnt erst nach der Erkennung des SFDs mit der Umwandlung der Nibbles in Bytes. Für den Daten-Teil, der aus N Bytes besteht, werden 2N Nibbles an der MII-Schnittstelle übertragen. Der MAC erzeugt die Fehler „Alignment Error“, wenn die Länge des Frames kein ganzzahliges Vielfaches von 8 Bit beträgt.

Die nachfolgende Tabelle 2 präsentiert die zeitliche Reihenfolge, in der die Nibbles beim Senden von Paketen aus der MAC-Unterschicht an der MII-Schnittstelle übertragen werden. Solange das Signal „TX_EN“ auf ´0´ gesetzt ist, werden die gesendeten Nibbles nicht ausgewertet und somit als „DON’T CARE“ betrachtet (bei „a“). Sobald das Signal „TX_EN“ auf ´1´ gesetzt ist, wird das erste Nibble der Präambel gesendet (bei „b“). Die 14-Nibble Präambel, die aus dem Muster „1010“ besteht, wird vollständig gesendet, dabei ist das Signal „TX_EN“ immer auf ´1´ gesetzt. Das 2-Nibble SFD folgt direkt auf die Präambel. Der erste Nibble des SFD hat den Wert „1010“, der zweite Nibble hat den Wert „1011“ (bei „c“ und „d“). Bei „e“ wird der erste Nibble des ersten Bytes von Frame gesendet. Zu beachten ist hierbei, dass das Signal „TX_EN“ an den entsprechenden Nibbles immer den Wert ´1´ aufweist. Des Weiteren müssen die Nibbles ohne Unterbrechung übertragen werden. Um dem Standard zu genügen, und, noch wichtiger, um ein Paket fehlerfrei zu erkennen können, müssen die Nibbles eines Paketes in der vorgeschriebenen Reihenfolge seitens der MAC-Unterschicht über die MII-Schnittstelle zum PHY gesendet werden.

Signal	**Werte von Bits des Nibbles an die MII-Schnittstelle**																		
TX_D0	X[a]	1[b]	1	1	1	1	1	1	1	1	1	1	1	1	1	1[c]	1[d]	D0[e]	D4
TX_D1	X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	D1	D5
TX_D2	X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	D2	D6
TX_D3	X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	D3	D7
TX_EN	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Tabelle 2: Nibbles für TX-Übertragung [nach 03]

Die Reihenfolge, in der die Nibbles eines Paketes empfangen werden, ist in Tabelle 3 ersichtlich. Die in der Tabelle 3 dargestellte Reihenfolge entspricht der Reihenfolge, in der das Paket gesendet wurde. Dabei wird jeder Nibble, der mit dem Signal „RX_DV“ = ´0´ empfangen wird, nicht ausgewertet und somit als ungültig bzw. als „DON’T CARE“ ausgewertet (bei „a“). Bei „b“ wird der erste Nibble der 14-Nibbles Präambel empfangen. Bei „c“ und „d“ werden die ersten und die zweite Nibble des SFD empfangen. Nach dem Empfangen der Nibbles von der Präambel und dem SFD wird der erste Nibble des Frames empfangen (bei „e“). Es ist erforderlich, dass das Signal „RX_DV“ während der Übertragung der entsprechenden Nibbles auf ´1´ gesetzt ist. Diese Reihenfolge der Nibbles sollen seitens des PHY über der MII-Schnittstelle zum MAC-Unterschicht für fehlerfreie Übertragung gesendet werden.

Es besteht die Möglichkeit, dass der Anfang der Präambel während der Herausfilterung der Taktinformationen im Ethernet-Transceiver bzw. im PHY-Baustein nicht an der ihr zugehörigen Stelle erkannt wird, deshalb gewisse Teile der Präambel verloren gehen [14][15]. Aus diesem Grund wird die Präambel in Hubs neu konstruiert. Laut IEEE 802.3 [03] darf ein empfangenes Paket ohne Präambel zur MAC-Unterschicht weitergeleitet werden, zugleich wird es erkannt. In der nachfolgenden Tabelle 4 wird der Empfang eines Paketes ohne Präambel dargestellt. Es ist hierbei jedoch unbedingt notwendig, dass das SFD vollständig vorhanden sein muss. Anhand der Tabelle 4 erkennt man, dass zuerst die zwei Nibbles der SFD empfangen wurden (bei „a“ und

„b“). Danach folgen die Nibbles des Daten-Frames. Des Weiteren wird das Signal „RX_DV“ an die entsprechenden Nibbles auf ´1´ gesetzt.

Signal	Werte von Bits des Nibbles an die MII-Schnittstelle																		
RX_D0	X[a]	1[b]	1	1	1	1	1	1	1	1	1	1	1	1	1	1[c]	1[d]	D0[e]	D4
RX_D1	X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	D1	D5
RX_D2	X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	D2	D6
RX_D3	X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	D3	D7
RX_DV	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Tabelle 3: Nibbles für RX-Übertragung mit vollständiger Präambel [nach 03]

Signal	Werte von Bits des Nibbles an die MII-Schnittstelle									
RX_D0	X	X	X	X	X	X	1[a]	1[b]	D0[c]	D4
RX_D1	X	X	X	X	X	X	0	0	D1	D5
RX_D2	X	X	X	X	X	X	1	1	D2	D6
RX_D3	X	X	X	X	X	X	0	1	D3	D7
RX_DV	0	0	0	0	0	0	1	1	1	1

Tabelle 4: Nibbles für RX-Übertragung mit unvollständiger Präambel [nach 03]

Es wurden in diesem Abschnitt die Binärdarstellung der Präambel und das SFD anhand der Tabellen 2, 3 und 4 erklärt, wie sie laut des Standards IEEE 802.3 dargestellt sein müssen. Die hier vorgestellten Informationen werden im Kapitel 5 als Basis eingesetzt, um den Standard IEEE 802.3 einzuhalten.

2.1.5.3 Zeitliche Anforderungen an der MII-Signale

Die Signale „RX_DV“, „RX_ER“ und „RX_D[3:0]“ bzw. „TX_EN“, „TX_ER“ und „TX_D[3:0]“ werden in Beziehung zu den Takt-Signalen „RX_CLK“ und „TX_CLK“ übertragen. Diese Beziehung ist durch bestimmte Zeit-Größen definiert, die in diesem Abschnitt vorgestellt werden. Dieser Arbeit beschäftigt sich mit der Realisierung eine Umschaltmechanismus, dem so genannten RTC „Real Time Crossbar“. Der RTC wird auf die MII-Schnittstelle implementiert, weshalb es von großer Bedeutung ist, die Zeitanforderungen der MII-Signale kennen zu lernen und durch die Realisierung des RTCs einzuhalten.

In der digitalen Technik wird ein Signal als „HIGH“ oder „LOW“ anhand vordefinierter Werte „$V_{H(min)}$“ und „$V_{L(max)}$“ beschrieben. Als „HIGH“ bezeichnet man ein Signal, dessen Potential größer als „$V_{H(min)}$“ ist, wohingegen als „LOW“ ein Signal, dessen Potential kleiner als „$V_{L(max)}$“ ist.

Die nachfolgende Abbildung 9 zeigt die Kennlinien, die das „HIGH“ bzw. „LOW“ für Takt- und Daten-Signale definiert.

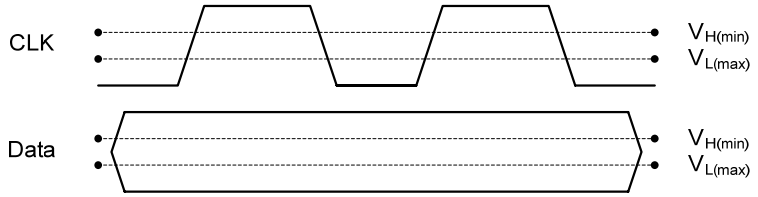

Abbildung 9: "HIGH" und "LOW" Kennlinien [nach 03]

Das IEEE 802.3 definiert die Werte „$V_{H(min)}$“ bei 2,0V und „$V_{L(max)}$“ bei 0,8V für die MII-Schnittstelle (Klausel 22.4.3). In diesem Zusammenhang werden in dieser Arbeit die Werte „$V_{H(min)}$“ und „$V_{L(max)}$“ der Signal-Darstellungen nicht angegeben bzw. dargestellt, jedoch wird ein Signal als „HIGH“ bzw. als „LOW“ entweder bei mittlerem Anstieg bzw. mittlerem Abstieg des Signals gekennzeichnet. Zunächst wird drei wichtigen Zeit-Größen („Delay-Time“, „Setup-Time“ und „Hold-Time“) der digitalen Technik vorgestellt, die die Interpretierung der MII-Signale beschreiben.

Die Verzögerungs-Zeit „Delay-Time“ beschreibt die Zeit, nach der der Wert der Daten als gültig erkannt werden muss bzw. die gültigen Werte der Daten an den Leitungen gespeist werden müssen. Die nachfolgende Abbildung 10 repräsentiert die minimale und maximale, von der IEEE 802.3 definierte Zeit, die nach der steigenden Flanke des Takt-Signals „TX_CLK“ die gültige Daten „TX_EN“, „TX_D[3:0]“ und „TX_ER“ auf die Leitungen gespeist werden muss. Der minimale Wert liegt bei 0ns und der maximale Wert bei 25ns.

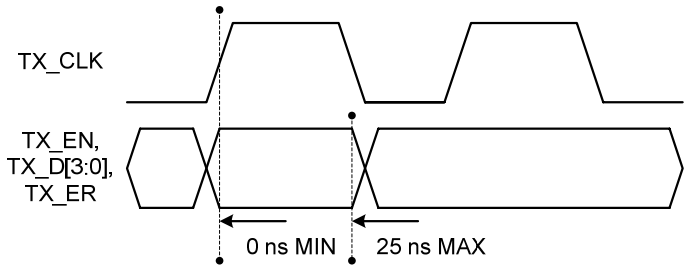

Abbildung 10: TX-Prozess "Delay-Zeit“ [nach 03]

Das „Setup-Time“ wird als „Die Zeitdauer vor der aktiven Taktflanke, ab dem das Eingangssignal eines Flipflops einen stabilen Wert aufweisen muss.“ definiert. Das „Hold-Time“ wird als „Die Zeitdauer nach der aktiven Taktflanke, bis zu dem das Eingangssignal eines Flipflops mindestens stabil anliegen muss.“ definiert [06]. Der IEEE 802.3 definiert nur die minimale Zeit für die „Setup-Time“ und „Hold-Time“. Hierbei müssen die gültigen Werte der Signale „RX_DV“, „RX_D[3:0]“ und „RX_ER“ für eine Zeitdauer von minimal 10ns vor der steigenden Flanke des Takt-Signals „RX_CLK“ einen stabilen Zustand aufweisen. Ebenso müssen sie für eine Zeitdauer von minimal 10ns nach der steigenden Flanke des Takt-Signals „RX_CLK“ einen stabilen Zustand aufweisen. Die nachfolgende Abbildung 11 zeigt die Zeitanforderungen „Setup-Time“ und „Hold-

Time“, die die empfangenen Daten vor die steigende Takt-Flanke des Signals „RX_CLK“ einhalten müssen.

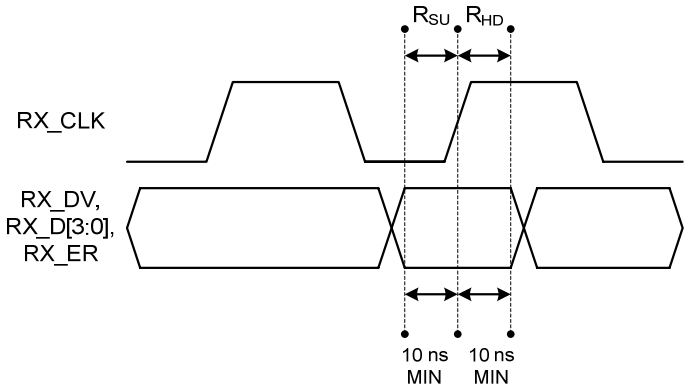

Abbildung 11: RX-Prozess "Setup-Time" und "Hold-Time" [nach 03]

Ein Takt-Signal ist durch das „Duty-Cycle“ charakterisiert. Das „Duty-Cycle“ beschreibt das zeitliche Verhältnis, das in Prozent angegeben wird, von Takt-Dach zu Takt-Boden, der Periodendauer eines Takt-Signals. Die nachfolgende Abbildung 12 repräsentiert ein Takt-Signal, dabei haben das Takt-Dach und der Takt-Boden unterschiedliche Größen. IEEE 802.3 definiert die „Duty-Cycle“ für die Takt-Signale „TX_CLK“ und „RX_CLK“ bei einem minimalen Wert von 35% und maximalen Wert von 65%.

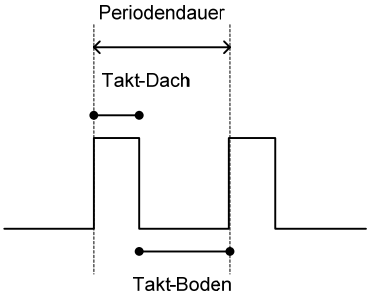

Abbildung 12: "Duty-Cycle" [18]

Zur Realisierung des Umschaltmechanismus RTC wird ein Versuch durchgeführt (Kapitel 4.3), in welchem zwei PHYs über die MII-Schnittstelle miteinander verbunden werden. Demzufolge werden die übertragen Daten im Vollduplex-Modus durch die zwei Takt-Signale „RX_CLK“ und „TX_CLK“ gesteuert. Dies kann zur Phasenverschiebung bei den Takt-Signalen „RX_CLK“ und „TX_CLK“ führen. Man bezeichnet somit die Übertragung als asynchron. Dies führt zur Verletzung der Zeitanforderungen „Setup-Time“ und „Hold-Time“. Die Nichteinhaltung der Zeitanforderung kann zur fehlerhaften Übertragung der Daten auf der MII-Ebene führen. Dies wird repräsentativ im Versuchsaufbau (Kapitel 4.4) anhand einer Waveform gezeigt.

2.1.5.4 MII-Sticker

Das IEEE 802.3 spezifiziert einen 40-PIN MII-Stecker, welcher in der vorher genannten Applikation eingesetzt werden kann. In der nachfolgenden Abbildung 13 ist der MII-Stecker zu sehen, dabei zeigt die Abbildung 13 die Anordnung der Pins des MII-Steckers.

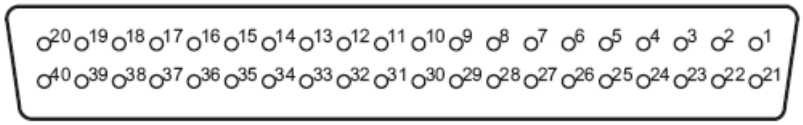

Abbildung 13: MII-Stecker Pins-Anordnung [nach 03]

Die Belegung von dem 40-PIN des Steckers an den MII-Signalen und an der Masse wird in der nachfolgenden Tabelle 5 beschrieben. Der MII-Stecker kann laut der IEEE-Spezifikation mit +5V oder +3,3V betrieben werden.

Kontakt	Signal Name	Kontakt	Signal Name
1	+5V/ +3,3V	21	+5V/ +3,3V
2	MDIO	22	COMMON
3	MDC	23	COMMON
4	RX_D3	24	COMMON
5	RX_D2	25	COMMON
6	RX_D1	26	COMMON
7	RX_D0	27	COMMON
8	RX_DV	28	COMMON
9	RX_CLK	29	COMMON
10	RX_ER	30	COMMON
11	TX_ER	31	COMMON
12	TX_CLK	32	COMMON
13	TX_EN	33	COMMON
14	TX_D0	34	COMMON
15	TX_D1	35	COMMON
16	TX_D2	36	COMMON
17	TX_D3	37	COMMON
18	COL	38	COMMON
19	CRS	39	COMMON
20	+5V/ +3,3V	40	+5V/ +3,3V

Tabelle 5: Belegung der MII-Signale an den Pins [nach 03]

2.1.6 Reconciliation-Unterschicht

Die MAC-Unterschicht kommuniziert mit der Bitübertragungsschicht mit primitiven Signalen, die als „PLS-Primitive“ (engl.: Physical Level Signaling) bezeichnet werden. Die Reconciliation-Unterschicht stellt die PLS-Primitive dem MAC zur Verfügung vor, sie dient als logische Schnittstelle zwischen der MAC-Unterschicht und der MII-Schnittstelle. Sie wandelt MII-Signale in MAC-PLS-Primitive um. Der technische Hintergrund dieser Schicht ist im Rahmen dieser Arbeit irrelevant und wird daher nicht weiter diskutiert. Die nachfolgende Abbildung 14 zeigt, wie die Signale der MII-Schnittstelle in die PLS-Primitive umgewandelt werden.

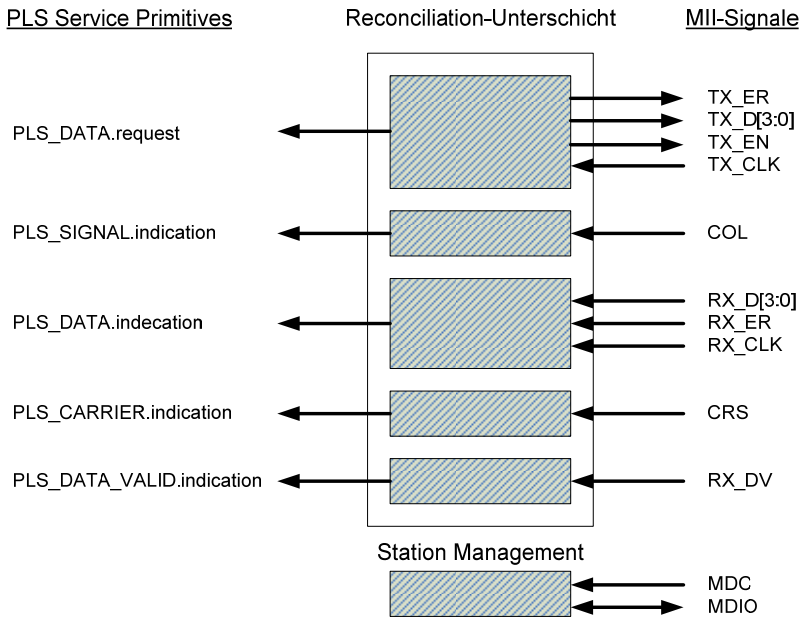

Abbildung 14: PLS-Primitive und MII-Schnittstelle [nach 03]

Die PLS-Primitive basieren auf den Signale „Indication“ und „Request“, die der MAC nutzt, um das Senden und das Empfangen von Daten zu steuern. Die Funktion „Request“ beinhaltet die Anforderung zur Übertragung von Daten in eine andere Station. Bei der Funktion „Indication“ wird darauf hingewiesen, dass die MAC-Unterschicht ein ankommendes Paket entgegengenommen hat.

2.1.7 MAC-Ebene

Die MAC-Ebene hat zwei grundlegende Funktionen. Die eine ist die Verwaltung von Frames, wobei Datenaufbereitung, Adressierung und Fehlerkontrolle stattfinden. Die andere ist die Zugangskontrolle, welche die Übertragung der Daten ins Netz ohne Kollision gewährleistet. Diese wird im nächsten Abschnitt ausführlich diskutiert. Als nächstes folgt eine Zusammenfassung der Aufgaben der MAC-Ebene.

Beim Senden eines Frames wird der Daten-Teil von der LLC-Ebene entgegengenommen. Die Frame-Länge wird überprüft, falls notwendig (Frame-Länge kleiner als 64 Byte), wird ein PAD-

Feld erzeugt und dem Frame angefügt. Anschließend wird der CRC berechnet und am Frame-Ende im FCS-Teil angefügt. Dann werden die Präambel, das SFD und die Ziel-Adresse dem Frame-Anfang hinzugefügt [11].

Beim Empfangen eines Frames wird zuerst untersucht, ob die Zieladresse mit der lokalen Station übereinstimmt. Im Falle der nicht Übereinstimmung wird der Frame verworfen. Bei Übereinstimmung des Frames zur Station wird die CRC auf Korrektheit überprüft. Danach werden die Präambel, das SFD und eventuell die PAD-Bytes abgeschnitten. Zuletzt wird der Daten-Teil an die LLC-Ebene weitergeleitet.

Für Ausführliche Informationen über die Struktur und die Funktionalität der MAC-Ebene sei auf den 802.3 Klausel 2, 3 und 4 verwiesen.

2.1.8 Übertragungsmodus

Die Übertragung von Paketen über das Übertragungsmedium wird in zwei Modi realisiert. Man unterscheidet zwischen Netzwerk-Systemen, die mit Halbduplex-Modus oder Vollduplex-Modus betrieben werden.

2.1.8.1 Halbduplex-Modus

Die Nutzung von mehreren Stationen auf ein gemeinsames Medium wird in Halbduplex Systemen durch CSMA/CD (Carrier Sense Multiple Access / Collision Detection) gesteuert.

Solange kein Frame übertragen wird, überwacht der MAC das Ethernet-Medium auf Übertragungen anderer Stationen durch Überwachung des Signals CRS, das aus dem PHY kommt. Ist das Medium besetzt (CRS = 1), verschiebt der MAC seiner Übertragungen bis das Medium frei wird bzw. bis das Signal (CRS = 0) ist, dann kann ein Frame gesendet wird. Nach der Übertragung des letzten Bit des Frames (hierbei ändert sich das Signal CRS von TRUE zu FALSE) startet der MAC seine Bit-Zählung bzw. „InterFrame Gap“-Zählung, bevor er mit der Übertragung des nächsten Frame starten darf. Eine Kollision (COL = 1) kann auftreten, wenn zwei Stationen gleichzeitig zu Senden beginnen. Hierbei senden die in der Kollision beteiligten Stationen ein 32 Bit „Jam-Signal“, um den anderen Stationen die Kollision zu signalisieren. Bevor die Stationen mit einem neuen Sendeversuch starten können, warten sie eine zufällige generierte Zeit, dieser Prozess ist als „Backoff“ bekannt [33].

2.1.8.2 Vollduplex-Modus

Der Vollduplex-Modus stellt eine Punkt-zu-Punkt Verbindung zwischen zwei Stationen mit einheitlicher Bandbreite-Verkabelung dar. Hierbei wird die Transmit-Schnittstelle zum Senden von Daten und die Receive-Schnittstelle zum Empfangen von Daten genutzt. Dadurch kann in diesem Modus keine Kollision auftreten. Demzufolge wird das Signal „COL“ immer auf Null sein. Da sowohl die Transmit-Schnittstelle als auch die Receive-Schnittstelle nur von einer Sende- bzw. einer Empfänger-Station genutzt wird, bleibt das Signal „CRS“ ebenso immer auf Null. Es

wird somit nicht von der MAC-Unterschicht in diesem Modus verwendet. In Vollduplex-Modus überwacht der MAC lediglich seine Daten-Übertragungen. Hierbei wartet er nach dem letzten gesendeten Frame, die „InterFrame Gap“ Zeit ab, dann sendet er den nächsten Frame [33].

2.2 Ethernet-Switchs

Switches sind Netzwerkgeräte, die auf der Schicht 2 des OSI-Modells arbeiten und mehrere Stationen bzw. Netz-Segmente verbinden. Die meisten Netzwerkinfrastrukturen basieren heutzutage auf Switches, die die lang vorherrschenden Hubs und Bridges ablösen. Switches werden für Ethernet, Fast-Ethernet und Gigabit-Ethernet von unterschiedlichen Herstellern angeboten.

In Hinblick auf die Funktionalität interpretiert der Switch die Zeil-Adressen der einkommenden Frames und vermerkt sie, falls sie unbekannt sind. Die Vermerkung der MAC-Ziel-Adressen wird in einer Look-Up-Tabelle einschließlich des Ports, an dem der Frame empfangen hat, gespeichert. Im Prinzip können Switches nach unterschiedlichen Verfahren arbeiten, die in diesem Abschnitt kurz beschrieben werden [12].

Bei „Cut-Through-Verfahren“ entscheidet der Switch unmittelbar nach Einlesen der MAC-Adresse des Ziel-Rechners, zu welchem Port der Frame weitergeleitet werden muss. Eine Speicherung des Paketes ist bei diesem Verfahren unnötig. Diese erbringt den Vorteil, dass der Switch extrem geringe Latenzzeit hat, unabhängig von der Paketlänge [12].

Bei „Store-and-Forward-Verfahren“ werden die angekommenen Frames in einem Puffer gelegt und dann vollständig eingelesen. Danach wird der Frame zum betroffenen Ausgangsport weitergeleitet. Dies hat zur Folge, dass die Verweilzeit des Frames im Switch sich vergrößert, womit die Latenzzeit von der Paketlänge abhängig wird [12].

In Hinblick auf die Hardware-Architektur besitzt ein Switch eine bestimmte Anzahl an Ports, die über die MDI-Schnittstelle mit dem PHY verbunden sind, der wiederum über die MII-Schnittstelle mit der MAC-Ebene verbunden ist. Die nachfolgende Abbildung 15 zeigt einen Switch-Baustein der Firma „Broadcom“. Der Switch-Baustein besteht aus fünf MDI-Ports ebenso einen MII-Port.

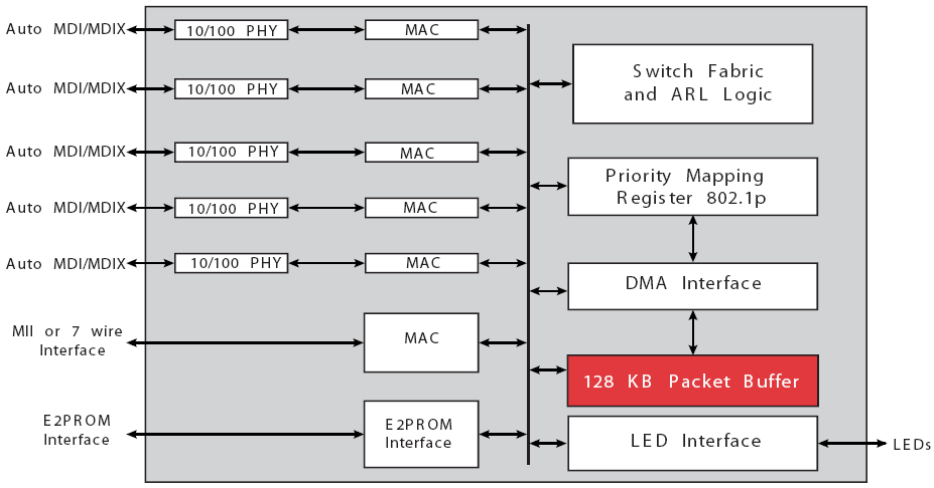

Abbildung 15: Aufbau eines Switch-Bausteines der Firma Broadcom [nach 22]

Neben den MAC-Bausteinen, die auf der MAC-Ebene liegen, befindet sich eine Switching-Fabric, die den Kern des Switches bildet. Durch die Switching-Fabric werden die Frames vom Eingangsport zum Ausgangsport transferiert [12].

2.3 Ethernet in der Automatisierungstechnik

In diesem Abschnitt wird zu Beginn der Begriff Automatisierungstechnik erklärt. Daraufhin wird ein Überblick darüber gegeben, welchen Zeitanforderungen die Datenübertragung in der Automatisierungstechnik standhalten soll. Des Weiteren wird kurz erläutert, warum Ethernet in der Automatisierung eingesetzt wird. Es wird auf die notwendigen Maßnahmen eingegangen, die an Standard-Ethernet durchgeführt werden müssen, um den Anforderungen der Automatisierungstechnik gerecht zu werden sodass er echtzeitfähig wird.

2.3.1 Automatisierungstechnik

Die Automatisierungstechnik ist eine technische, übergreifende Disziplin, die sich mit der Konzipierung und Entwicklung automatisch ablaufender Vorgänge zur Automatisierung von Prozessen beschäftigt [17].

Die Architektur eines Systems in der Automatisierungstechnik lässt sich anhand der Informationen, die in ihren verschieden Betriebsebenen zu übertragen sind, in einem Ebenen-Modell betrachten. Jede einzelne Ebene besitzt eigene Eigenschaften, wobei der Informationsaustausch nur zwischen den benachbarten Ebenen stattfindet. Die Abbildung 16 repräsentiert die Automatisierungs-Ebenen.

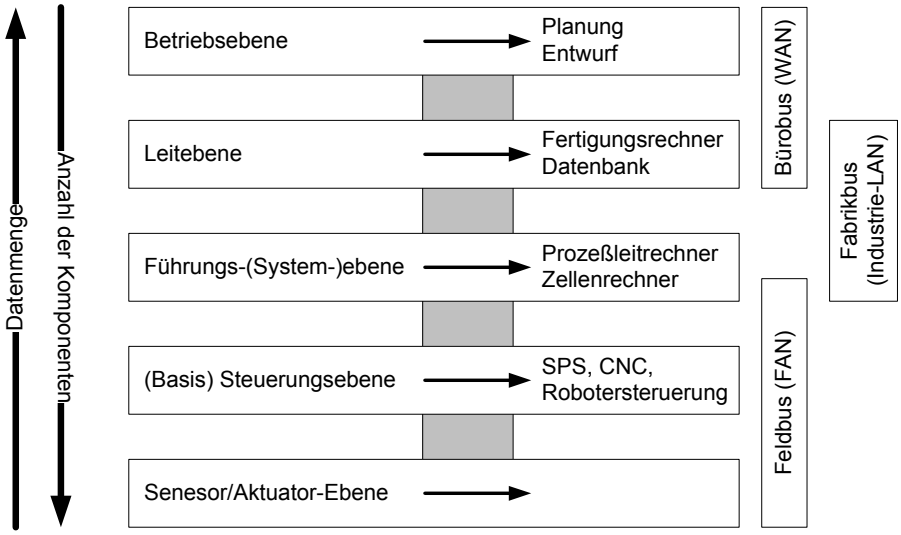

Abbildung 16: Automatisierungs-Ebenen [nach 05]

In der Betriebs- und Leitebene werden die Entscheidungen, die das gesamte System steuern bzw. kontrollieren, getroffen, wobei die Anzahl der Komponenten klein und die Datenmenge am größten ist. In den ersten beiden Ebenen werden Bürobusse (Ethernet-Kabel) eingesetzt. In der Steuerungsebene werden Befehle erteilt, die die Sensoren und Aktoren auf der Sensor- und Aktor-Ebene kontrollieren; hierbei sind die Anzahl der Komponenten groß und die Datenmenge klein. In den letzten beiden Ebenen werden jedoch Feldbusse eingesetzt, hierbei ist die Datenmenge klein und die Anzahl der angeschlossenen Komponenten groß. In den letzten beiden Ebenen besteht die Anforderung, dass die Reaktion eines Ereignisses innerhalb einer bestimmten Zeit vorliegen muss.

Die Forderung, dass eine Reaktion auf ein Ereignis innerhalb einer bestimmten Zeit vorliegen muss, wird als „Echtzeit“ bezeichnet [04]. Die IAONA (Industrial Automation Open Network Alliance) definiert Echtzeitklassen für die Automatisierungstechnik, diese Klassifizierung ist anhand der nachfolgenden Tabelle 6 ersichtlich.

Echtzeitklasse	1	2	3	4
Anforderung	Gering	mittel	hoch	extrem hoch
Typische Ebene	Leitebene	Führungsebene	Steuerungsebene	Sensor-/Aktor-Ebene
Nutzdaten max.	500 KB	500 Byte	32(-200) Byte	20 Byte
Verzögerung max.	5 s	500 ms	5 ms	0,5 ms
Jitter max.	> 1 s	0,1 ms – 3 ms	10 µs – 400 µs	0,5 µs – 15 µs

Tabelle 6: Echtzeitklassen nach IAONA [5]

Das Ethernet besitzt Vorteile wie günstige Netzwerkadapter, Flexibilität und einfache Konfiguration des Netzwerkes. Außerdem wird über das Ethernet eine Verbindung zum Internet hergestellt, sodass Kunden bspw. eine Durchgängigkeit der Unternehmensprozesse von der

Verwaltung bis hin zur Produktion über das Internet erhalten können. In der Industrie möchte man von diesen Vorteilen des Ethernets als Feldbus profitieren [04].

Das Ethernet in seiner Standard-Form erfüllt die Echtzeitanforderungen der Automatisierungstechnik nicht. Daher müssen zahlreiche Anpassungen erfolgen, um das Ethernet nach dem Standard IEEE 802.3 echtzeitfähig zu machen.

2.3.2 Anpassung des Ethernets für Echtzeit

In der Automatisierungstechnik wird von den eingesetzten Netzwerken verlangt, dass sie Echtzeit-Verhalten aufweisen, indem sie die Daten rechtzeitig und deterministisch den Stationen zur Verfügung stellen. Das Ethernet bei Halbduplex-Verbindung, das auf CSMA/CD basiert, stellt ein Problem für deterministische Kommunikation dar. Dieses Problem lässt sich durch die Verwendung der kollisionsfreien Vollduplex-Verbindung und 100 MBit/s Verbindung umgehen. Dadurch lassen sich die durch Pufferung in Switches aufgetretenen Latenzen vermeiden. Um sicher zu stellen, dass nur ein Teilnehmer zugleich senden darf, werden modifizierte Netzwerkgeräte mit Hilfe eines Zeitmultiplexverfahren (TDMA, engl.: Time Division Multiple Access) eingesetzt. Des Weiteren müssen die Stationen im Netzwerk sich synchron verhalten. Dafür wird unter anderen das Protokoll IEEE 1588 zur Synchronisation der Stationen eingesetzt [04].

Auf dem Markt existieren echtzeitfähige Ethernet-Lösungen, wie Ethernet/IP, Ethernet PowerLink, ProfiNet, SERCOS III und EtherCAT. Diese Lösungen wurden in der Arbeit von Westerholdt [04] mit detaillierten technischen Daten analysiert und dargestellt. Es wird ein neuartiger Ansatz entwickelt, der die Eigenschaften der gängigen Echtzeit-Ethernet-Lösungen miteinander kombiniert und sie somit im Einzelnen übertrifft. Dieser neue Ansatz, der auf modifizierten Switches basiert, wird im Weiteren ausführlich dargelegt.

3 Ansatz für echtzeitfähiger Ethernet-Switch

Standard-Ethernet erfüllt nicht die Echtzeit-Anforderungen der Automatisierungstechnik. In diesem Kapitel wird ein neuer Ansatz zur Erfüllung der Echtzeitfähigkeit vorgestellt, der die Vorteile der bestehenden Ansätze verbindet und auf modifizierten Switchs basiert. Des Weiteren wird eine Switching-Architektur vorgestellt, die im ProfiNet eingesetzt wird. Die Switching-Architektur des ProfiNet aus dem Blick der Hardware ähnelt der Architektur des Switchs, die im Rahmen dieser Arbeit vorgestellt wird. Das Vorgehen im Rahmen dieser Arbeit wird im Kapitel 3.4 diskutiert.

3.1 Netzwerk Infrastruktur

Die gängigen echtzeitfähigen Ethernet-Lösungen bilden Subnetze, in denen ausschließlich echtzeitfähige Teilnehmer erlaubt sind. Dabei ist asynchroner Datenverkehr möglich. Jedoch zerstört ein Laptop, der ohne Kenntnis des Echtzeit-Protokolls angeschlossen wird, die Echtzeitfähigkeit des Netzes [01].

Dopatka [01] geht von einer auf IEEE 802.3 basierten Netzwerk-Infrastruktur in Baum-Topologie aus, dass aus einem Switched-Ethernet in Vollduplex-Modus mit einheitlicher Bandbreiten-Verkabelung besteht. An das Netzwerk können sowohl Echtzeit-Geräte (engl.: RT-Devices), wie z.B. Aktoren oder Sensoren, als Nicht-Echtzeit-Geräte (engl.: nRT-Devices), beispielsweise Laptops an einem modifizierten Switch, angeschlossen werden. Hierbei können nRT- und RT-Geräte transparent in das echtzeitfähiges Netzwerk hinzugefügt und wieder entfernt werden. Sie können asynchrone Daten senden und empfangen, ohne das Echtzeitverhalten des Netzwerks zu beeinflussen. Zur Vermeidung von Pufferung der RT-Übertragungen in der Switchs erfolgt die Echtzeit über ein TDMA-Zugriffsverfahren, die anhand eines Schedulers durchgeführt wird.

Da ein RT-Scheduler die RT-Übertragungen organisiert, müssen die RT-Geräte Informationen über den Scheduler haben, damit sie wissen, wann sie asynchrone Abfragen (engl.: Polling) innerhalb eines vordefinierten Zeitschlitzes beantworten können. Dopatka stellt zwei Lösungen für diese Netzwerk Infrastruktur vor, so dass die Geräte entweder als aktive Geräte oder als passive Geräte betrieben werden können. Die nachfolgende Abbildung 17 repräsentiert die beiden möglichen Infrastrukturen, die von Dopatka vorgestellt sind. In Teil (a) der Abbildung 17 werden die Geräte als aktiv dargestellt, wohingegen in Teil (b) als passiv dargestellt werden.

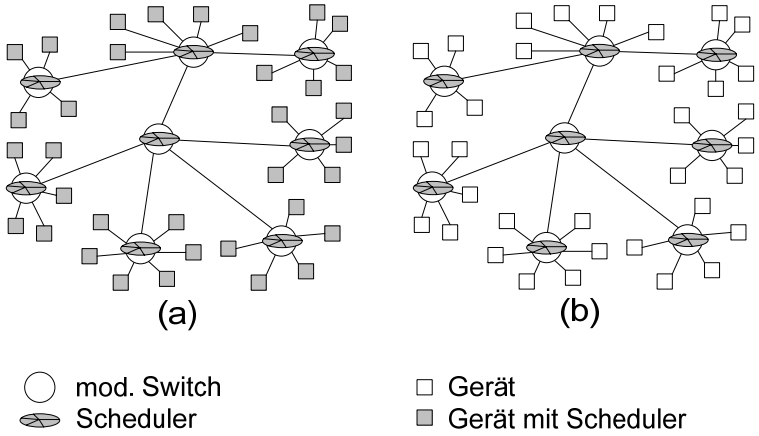

Abbildung 17: Netzwerk-Infrastruktur und Verteilung von den Scheduler [02]

Werden die Geräte als aktiv betrieben, so wird in den RT-Geräten ihr entsprechender RT-Scheduler gelegt, der sie beschreibt, wie sie die RT-Daten senden können. Dies hat zur Folge, dass die Uhren der RT-Geräte synchronisiert sein müssen, dies geschieht unter Zuhilfenahme der Synchronisations-Protokolle IEEE 1588 oder PTCP (Precision Transport Clock Protocol). Um die Geräte als passiv zu betreiben, so dass sie keine Informationen über den RT-Scheduler brauchen, müssen modifizierte Switches eingesetzt werden. Hierbei beinhaltet der modifizierte Switch einen Scheduler, der die Anfragen an die Geräte steuert. Dies hat zur Folge, dass die Uhren der Geräte nicht synchronisiert sein müssen, da die modifizierten Switches die Abfragen an die Geräte stellen und entscheiden, wann die RT-Daten gesendet werden. Jedoch muss die Zeit, die der Switch für die Abfrage an die Geräte benötigt, mit dem Zeitschlitz berechnet und in dem RT-Scheduler betrachtet werden.

3.2 ProfiNet

Das von Siemens entwickelte ProfiNet IRT (Industrial Realtime) basiert auf einem Standard Switching-Ethernet, jedoch werden im eingesetzten Netzwerk modifizierte Switches zur Erreichung von Echtzeitfähigkeit eingesetzt. Im Switch wird ein Schedule geladen, das Offline während der Konfiguration der Anlage berechnet wird. Das Switch kann sowohl im RT-Modus als auch im nRT-Modus betrieben werden. Dafür muss eine Zykluszeit festgelegt werden, die mit Hilfe des PTCP (Precision Transport Clock Protocol) berechnet wird. Die Zykluszeit wird in zwei Phasen aufgeteilt, wobei eine Phase für RT-Verkehr und die andere Phase für nRT-Verkehr verantwortliche ist. Die Funktionalität des Switches ist anhand der nachfolgenden Abbildung 18 ersichtlich [16].

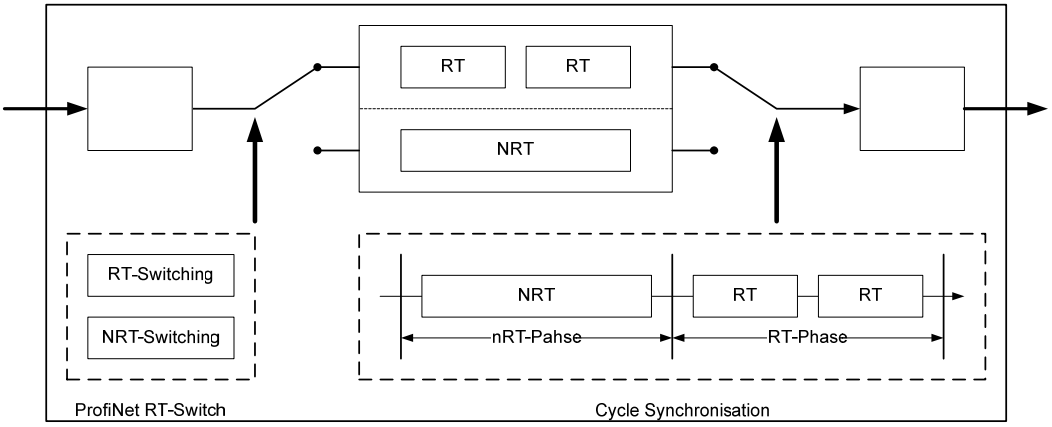

Abbildung 18: Architektur des ProfiNet-Switchs „ERTEC 400“ [nach 16]

Die Architektur des ProfiNet-Switches „ERTEC 400“ wird durch ein ASIC realisiert [16]. Ausführliche Beschreibung der Funktion dieser Architektur wird im Rahmen dieser Arbeit nicht behandelt.

3.3 Modifikation am Ethernet-Switch

Die Architektur eines Standard-Switches wurde in dem Kapitel 2.2 erläutert. In diesem Abschnitt werden die von Dopatka angenommenen Modifikationen vorgestellt, die an einem Standard-Switch durchgeführt werden können, damit er echtzeitfähig wird. Im Standard-Switch werden die Ziel-Adressen in der Sicherungsschicht ausgewertet, und anhand dessen werden die Pakete weitergeleitet. Dopatka geht davon aus, dass keine Auswertung der Ziel-Adressen für die RT-Übertragungen im Switch stattfindet, sondern durch den Scheduler gesteuert wird.

Zur Vermeidung von Kollisionen und Pufferung in einem Switch wird ein Steuerungsmechanismus zwischen der Bitübertragungsschicht und Sicherungsschicht innerhalb des Switches vorgeschlagen. Dieser Mechanismus, welcher RTC (Real Time Crossbar) genannt wird, basiert auf der MII-Schnittstelle. Des Weiteren muss der Switching-Kern modifiziert werden, um fehlerfreie und gleichzeitige Übertragung von RT-Daten und nRT-Daten innerhalb des Switches zu gewährleisten, dies wird ausführlicher im Abschnitt 3.3.2 diskutiert.

Die nachfolgende Abbildung 19 weist den von Dopatka [01] vorgestellten Entwurf für den modifizierten Switch auf. Das zu implementierende RTC und der Scheduler-Halter können in einem FPGA oder ASIC implementiert werden. Wohingegen der MAC und der Switching-Kern entweder durch gefertigte Bausteine oder Neu-Implementierung durch FPGA oder ASIC realisiert werden.

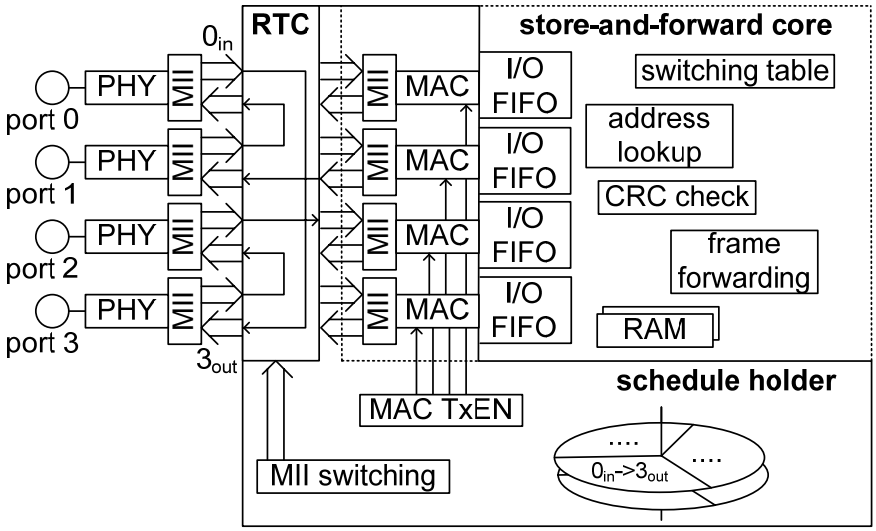

Abbildung 19: Entwurf für den modifizierten Switch [01]

3.3.1 RTC (Real Time Crossbar)

In diesem Abschnitt werden die von Dopatka [01] vorgeführten Spezifikationen der RTC vorgestellt. Die Aufgabe des RTC (Real Time Crossbar) ist es, eine Verbindung zwischen den PHY-Bausteinen anhand eines vordefinierten „Real Time"-Schedulers herzustellen und die einkommenden und ausgehenden nRT-Übertragungen vom Switching-Kern zu steuern. Das selbst zu entwerfende RTC wird zwischen den PHY-Bausteinen und den MAC-Bausteinen gelegt. Auf beiden Seiten des RTCs liegen MII-Schnittstellen, die als „Crossover Verbindung" angeschlossen werden. Hierbei wird die Transmit-Schnittstelle jedes Senders mit der entsprechenden Receive-Schnittstelle des Empfängers verbunden. Die Verbindungen, die das RTC stellt, werden anhand eines vordefinierten Schedulers gesteuert. Durch die Modifizierung des Switches erreicht man eine schnellere Bearbeitung der Pakete als in den „Cut-Through" Switches.

Wann und wie der RTC umgeschaltet werden soll, wird von dem Scheduler festgelegt. Während des Konfigurationsprozess wird der Scheduler berechnet und in den Switch geladen. Die Berechnung des Schedulers ist nicht Teil dieser Arbeit und wird daher nicht weiter diskutiert.

Der RTC soll drei Modi unterstützen. Im RT-Modus werden alle PHYs miteinander verbunden, dabei wird eine schnellere RT-Übertragung erzielt. Beim nRT-Modus werden die PHYs zu den zugehörigen MACs verbunden, hierbei funktioniert der Switch als Standard „Store-and-Forward" Switch. Wird ein Eingang oder ein Ausgang eines PHYs nicht von der RT-Schedule benutzt, dann soll dieser PHY während des entsprechenden Zeitschlitzes an den zugehörigen MAC verbunden werden, damit nRT-Frame empfangen bzw. gesendet werden können. In der vorherigen Abbildung 19 sind diese Modi zu erkennen. Anhand der Abbildung 19 ist zu sehen, wie der Eingang von Port 0 mit dem Ausgang von Port 3 verbunden ist. Eine asynchrone nRT-Übertragung wird zum Ausgang von Port 0 weitergeleitet. Im Falle, dass ein Eingang oder ein

Ausgang bestimmter Port nicht im RT-Scheduler enthalten ist, so wird an ihrem entsprechenden MAC verbunden.

In Kern dieser Arbeit wird untersucht, ob der vorgestellte RTC realisierbar ist. Des Weiteren wird die Implementierung des RTC (mit reduzierter Funktionalität) in der VHDL-Sprache für einen FPGA-Baustein realisiert.

3.3.2 Modifizierung der Switching-Architektur

Die von den PHY ankommenden RT-Daten werden durch den RTC an den entsprechenden PHY weitergeleitet bzw. gesteuert. Zugleich gehen die aus dem MAC gesendeten nRT-Daten verloren, weil der MAC nicht mit dem entsprechenden PHY verbunden ist. Daher muss die Architektur des MACs modifiziert werden, um dieses Problem zu beheben. Dopatka [01] stellt drei Ansätze zur Implementierung der Switching-Architektur vor. In diesen Ansätzen wird dem Mechanismus RTC dazu eingefügt, um eine „Real-Time"-Fähigkeit innerhalb des Switches zu garantieren. Der Switching-Kern, der aus Switching-Fabric und Look-Up-Tabelle besteht, beeinflusst die Echtzeit-Fähigkeit des Switchs nicht.

Durch den Einsatz eines modifizierten MACs, kann ein Steuer-Signal, welches „TxEN" genannt wird, den Sende-Prozess von nRT-Daten steuern. Das Signal „TxEN" wird durch das RTC implementiert und steuert dem MAC, ob er die nRT-Daten an den entsprechenden PHY über das RTC senden darf.

Die komplette Switching-Architektur kann in einem ASIC implementiert werden. Mit dieser Architektur können Prozeduren der Switching, Auswertung der Ziel-Adresse, Überprüfung der CRC und Segmentierung von Frames unterzogen werden. Des Weiteren kann die Sendung von nRT-Daten zum PHY ebenso gesteuert werden. Diese Variante ist für Massen-Produktion wirtschaftlich geeignet.

Beim dritten Ansatz werden Standard-Switching-Kern und ein Standard-MAC eingesetzt. Der Scheduler-Halter steuert den Sende-Prozesses von nRT-Daten zum PHY. Dies geschieht durch die Steuerung des Signals „CRS", indem er es auf ´1´ setzt. Damit signalisiert er, dass das Übertragungsmedium nicht frei ist. Diese Variante ist für einen Prototyp wirtschaftlich geeignet und wird im Rahmen dieser Arbeit als Basis betrachtet. Bei Vollduplex-Modus reagieren nicht alle MAC-Bausteine zum Signal „CRS". Daher muss ein MAC-Baustein gesucht wird, der zum Signal „CRS" reagieren kann, um der Steuerung über das Signal „CRS" zu realisieren.

3.4 Vorgehen bei dieser Arbeit

Zur Erstellung eines hardwarenahen Entwurfes des bereits vorgestellten Ethernet-Switches, der auf RTC basiert, wurden im Rahmen dieser Arbeit diverse Schritte durchgeführt, welche in diesem Abschnitt vorgestellt werden.

Der Umschaltmechanismus RTC muss zuerst auf mögliche Realisierung untersucht werden. Die Architektur des RTCs hat die Funktion, eine unbestimmte Anzahl von PHYs auf der MII-Schnittstelle miteinander zu verbinden. Die möglichen Applikationen bzw. Einsatz-Möglichkeiten der MII-Schnittstelle wurden im Kapitel 2.1.5 diskutiert. Eine direkte Verbindung zwischen dem PHY und der MAC stellt eine dieser Applikationen dar. Eine direkte Verbindung zwischen PHYs auf der MII-Schnittstelle wurde von den IEEE 802.3 nicht vorgesehen. Wie im Kapitel 2.1.5 erklärt wurde, erzeugt der PHY die Takt-Signale „RX_CLK“ und „TX_CLK“, die das Empfangen und Senden von Daten steuern. Die Zeitanforderung „Setup-Time“ und „Hold-Time“ an das Takt-Signal „TX_CLK“ muss eingehalten werden, um die Daten aus dem MAC fehlerfrei weiter senden zu können. Ebenso gilt dies für die Zeitanforderung „Delay-Time“ an das Signal „RX_CLK“, um die aus dem PHY empfangenen Daten zu MAC fehlerfrei weiter zu geben. Da die Architektur „RTC“ mehrere PHYs nutzt, die die Empfangen und Senden von Daten steuern, werden die Daten mit dem Takt-Signal des Quelle-PHYs gesendet und mit dem Takt-Signal des Ziel-PHYs empfangen. Dies bedeutet, dass die Übertragung der Daten auf die MII-Ebene von zwei Takt-Signalen gesteuert wird. Die Steuer-Takt-Signale „RX_CLK“ und „TX_CLK“ werden als Ausgänge betrieben und können somit nicht als Eingänge verwendet werden, um die Daten-Übertragung zu steuern. Demzufolge besteht die Möglichkeit einer Phasenverschiebung zwischen den Takt-Signalen „RX_CLK“ und „TX_CLK“ zu entstehen. Dies würde zur Verletzung der Zeitanforderungen „Setup-Time“ und „Hold-Time“ führen. In diesem Fall ist zu erwarten, dass die Daten auf der MII-Ebene fehlerhaft übertragen werden. Des Weiteren wurde bereits im Kapitel 2.1.5 erklärt, dass der PHY-Baustein während der Herausfilterung der Taktinformationen gewisse Teile der Präambel verlieren kann. Zum Senden eines Pakets wird die Präambel im MAC erzeugt und zum Paket-Anfang hinzugefügt. Da die Pakete den MAC in der vorgestellten Architektur „RTC“ nicht erreichen werden, wird erwartet, dass diese mit unvollständiger Präambel weitergeleitet werden. Um genauere Erkenntnisse über die erwarteten Probleme zu gewinnen, aus welchen dann Lösungsansätze erstellt wurden, die zur Realisierung des RTCs dienen können, wurden folgende Schritte unternommen.

Um die Realisierbarkeit einer möglichen direkten Verbindung zwischen den PHYs auf der MII-Schnittstelle zu überprüfen, wurde ein Versuch durchgeführt (Kapitel 4.3). Hierbei wurde eine Platine, die sogenannte „MII-Crosslink-Platine“, erstellt, die zwei PHYs direkt miteinander über die MII-Signale verbindet. Die MII-Signale können an einem Logik-Analyser angeschlossen werden. Dadurch lassen sich die MII-Signale während der Übertragung der Daten betrachten. Ebenso lässt sich bei fehlerhafter Übertragung durch die Waveform der Signale der Grund der Fehler feststellen. Die erwarteten Probleme (die Phasenverschiebung und Unvollständigkeit der Präambel) lassen ebenso dadurch feststellen. Nach den gewonnen Erkenntnissen aus den Daten-Übertragungen auf die MII-Ebene zwischen den zwei PHYs lassen sich Lösungsansätze vorschlagen, die zur Realisierung einer Verbindung auf der MII-Ebene führen.

Um eine fehlerfreie Verbindung auf der MII-Ebene realisieren zu können, wurden digitale Schaltungen für FPGA-Baustein mit der Programmiersprache VHDL entwickelt (Kapitel 5.4). Um die Hardware-Entwicklung voranzutreiben, war es nötig die Grundkenntnisse über die FPGA-

Architektur und VHDL-Sprache zu vertiefen. Das Vorgehen zur Implementierung von digitalen Schaltungen an einem FPGA-Baustein richtet sich an einem Entwurfsmodell aus, das im Kapitel 5.2 vorgestellt wird. Zum besseren Verständnis der entworfenen digitalen Schaltungen werden im Kapitel 5.3 die wichtigsten Konzepte der VHDL-Programmiersprache anhand von kurzen Beispielen erläutert. Um die Funktionalität der entworfenen Schaltungen zu überprüfen, wurden Simulationen durchgeführt. Dies wird anhand von Simulations-Auszügen dargestellt (Kapitel 5.5). Die sogenannte Synthese gibt an, ob den programmierten digitalen Komponenten an einem FPGA realisierbar sind. Anhand von Log-Datei-Auszügen wird gezeigt, dass die programmierten Schaltungen an einem FPGA implementierbar sind. Eine Zeit-Analyse wurde durchgeführt, um zu zeigen, dass die entworfenen Module den Zeitanforderungen „Setup-Time“, „Hold-Time“ und „Delay-Time“ entsprechen. Dies wird anhand von Log-Datei-Auszügen ersichtlich. Der Umschaltmechanismus RTC wurde durch das Modul „RTC_CORE“ (mit reduzierter Funktionalität) in der VHDL-Sprache realisiert. Das Modul „RTC_CORE“ kann zwar zwischen vier PHYs umschalten, jedoch nicht zum MAC.

Ein Blockschaltplan für eine RTC-Test-Platine wurde erstellt (Abschnitt 5.8). Die Platine dient zum praktischen Testen der entworfen Module. Augrund der begrenzten Zeitdauer der Diplomarbeit wurde nur der Blockschaltplan für den Echtzeit-Switch erstellt (Abschnitt 5.9).

4 Implementierung der MII-Crosslink Platine

Eine direkte Verbindung auf die MII-Ebene kann zu fehlerhafter Übertragung der Pakete führen. Probleme, wie z.B. die Phasenverschiebung der Signale „RX_CLK“ und „TX_CLK“, die zu Verletzung der Zeitanforderungen führen können, sind durch diese Verbindung zu erwarten. Ebenso Teile der Präambel können durch diese Verbindung fehlen. Um diese Probleme untersuchen zu können, wird in diesem Kapitel die Realisierung der MII-Crosslink-Platine vorgestellt, die die Verbindung zwischen zwei PHYs realisiert. Des Weiteren wird die Auswertung der übertragenen Ethernet-Pakete über die MII-Schnittstelle diskutiert. Zunächst wird der MicroPHY-DemoBoard vorgestellt, der als Implementierung der Bitübertragungsschicht entspricht. Auf die Inbetriebnahme der MII-Crosslink-Platine wird eingegangen. Anschließend werden Lösungsansätze für die aufgetretenen Probleme bei der Übertragung der Pakete auf die MII-Ebene diskutiert.

4.1 MicroPHY-DemoBoard

Der MicroPHY-DemoBoard der Firma Teridian ist eine Auswertungsleiterplatte, die zur Auswertungszwecken dient. Der MicroPHY-DemoBoard implementiert die Bitübertragungsschicht des OSI-Modells. Sie realisiert die MDI-Schnittstelle, den PHY und die MII-Schnittstelle. Die nachfolgende Abbildung 20 repräsentiert das MicroPHY-DemoBoard. Der MicroPHY-DemoBoard beinhaltet einen RJ45-Stecker, einen „78Q21X3-DB“-Baustein der Firma Teridian und einen MII-Stecker. Der RJ45-Steker ist mit den MDI-Signalen verbunden und stellt die Signale zum Außenwelt zur Verfügung. Der „78Q21X3-DB“-Baustein ist mit dem RJ45-Steker durch die MDI-Signale und durch die MII-Signale mit dem MII-Stecker verbunden. Der MII-Stecker ist mit den MII-Signalen verbunden und stellt die Signale der Außenwelt zur Verfügung.

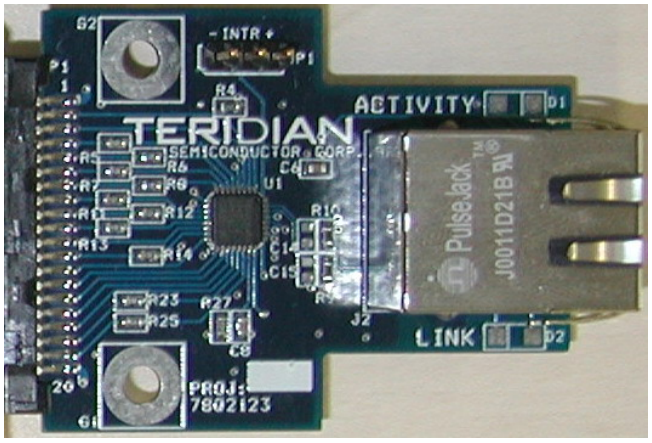

Abbildung 20: MicroPHY DemoBoard[06] [21]

Der „78Q21x3-DB“-Baustein implementiert die PHY-Unterschicht der Bitübertragungsschicht. Die nachfolgende Abbildung 21 repräsentiert einen Blockschaltplan der „78Q21x3-DB“-Bausteine und zeigt, wie der Baustein die MDI-Signale und die MII-Signale verwaltet.

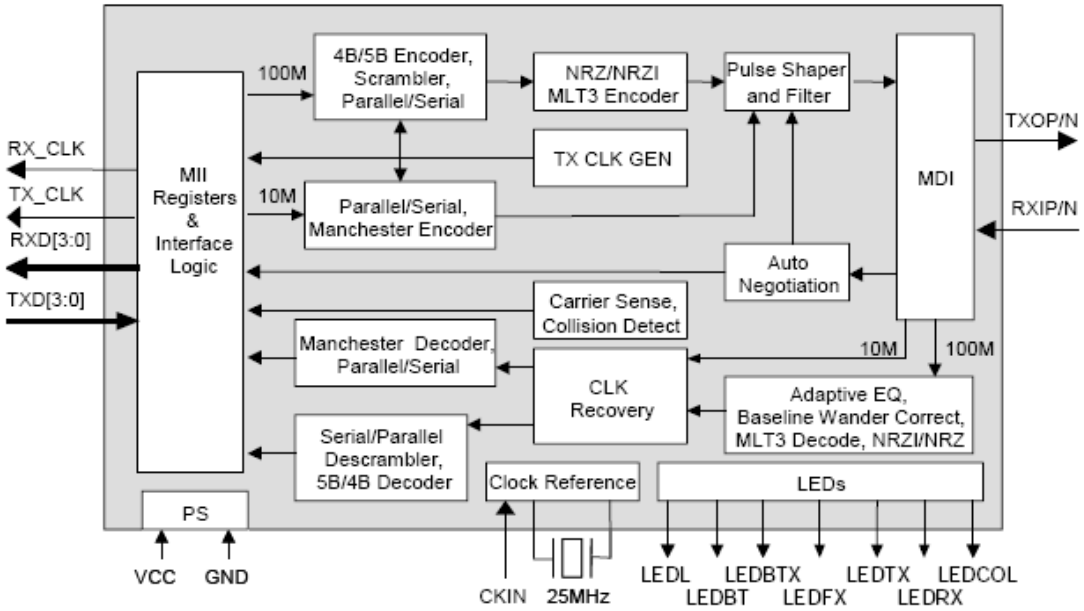

Abbildung 21: Blockschaltplan des "78Q21x3-DB"-Bausteins [20]

Die wesentliche Charakteristik des PHY-Bausteins, die in diesem Versuch von Bedeutung ist, sind die Zeitanforderungen bzw. die zur Verfügung gestellten Zeiten für die Signale der MII-Schnittstelle. Die wesentlichen Signale der MII-Schnittstelle sind die Signale der Receive-Interface und der Transmit-Interface, diese sind nämlich „RX_CLK“, „RX_DV“, „RX_ER“ und „RX_D[3:0]“ für Receive-Interface und „TX_CLK“, „TX_EN“, „TX_ER“ und „TX_D[3:0]“ für Transmit-Interface.

Die nachfolgende Tabelle 7 repräsentiert eine Auflistung der Zeiten, durch die die Signale „RX_CLK“, „RX_DV“, „RX_ER“ und „RX_D[3:0]“ von dem „78Q21x3-DB“-Baustein am Receive-Interface zur Verfügung gestellt werden. Die Liste liegt den minimalen und den maximalen Wert der „RX_{DLY}“ fest. “. Der minimale Wert der „RX_{DLY}“ Zeit beträgt 10ns, hingegen beträgt der maximale Wert der „RX_{DLY}“ 30ns. Der Arbeitszyklus (engl.: Duty-Cycle) des Signals „RX_CLK“ beträgt minimal 40% und maximal 60%.

CHARACTERISTICS	SYMBOL	CONDITIONS	MIN	NOM	MAX	UNIT
Receive Output Delay: RX_CLK to RXD[3:0], RX_DV, RX_ER	RX_{DLY}		10		30	ns
RX_CLK Duty-Cycle			40		60	%

Tabelle 7: "Delay-Zeit" vom "78Q21x3-DB"-Baustein [21]

Die nachfolgende Abbildung 22 repräsentiert die Beziehungen der Signale „RX_DV“, „RX_ER“ und „RX_D[3:0]“ zum Signal „RX_CLK“. Die gültigen Werte der Signale „RX_DV“, „RX_ER“ und „RX_D[3:0]“ werden in Beziehung zu der steigenden Flanke des Signals „RX_CLK“ nach

minimalen 10ns bzw. maximalen 30ns an den Leitungen gespeist bzw. als gültige Daten erkannt werden.

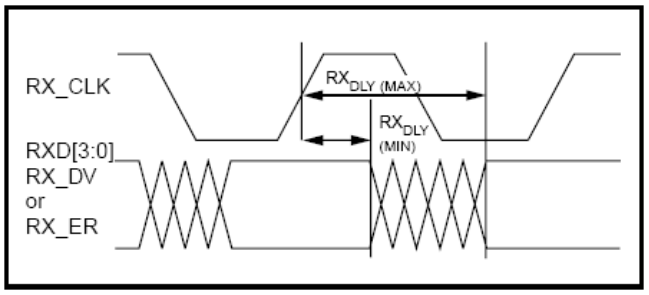

Abbildung 22: "Delay-Time" vom "78Q21x3-DB"-Baustein [21]

Eine Auflistung der Zeitanforderungen, die den „78Q21x3-DB“-Baustein an die Signale „TX_CLK“, „TX_EN“, „TX_ER“ und „TX_D[3:0]“ fordert, repräsentiert die nachfolgende Tabelle 8. Die Liste liegt die minimale und maximale Werte der Zeiten „TX_{SU}“ und „TX_{HD}“ fest. Der minimale Wert der „TX_{SU}“ beträgt 15ns und der minimale Wert der „TX_{HD}“ 0ns. Der Arbeitszyklus (engl.: Duty-Cycle) des Signals „TX_CLK“ beträgt minimal 40% und maximal 60%. Das Signal „T_{CKIN}“ findet im Rahmen dieser Arbeit keine Anwendung, da der „78Q21x3-DB“-Baustein sein Takt-Signal aus dem „On-Chip“ Crystal-Oscillatoer erhält.

CHARACTERISTICS	SYMBOL	CONDITIONS	MIN	NOM	MAX	UNIT
Setup Time: TX_CLK to TXD[3:0], TX_EN, TX_ER	TX_{SU}		15			ns
Hold Time: TX_CLK to TXD[3:0], TX_EN, TX_ER	TX_{HD}		0			ns
CKIN-to-TX_CLK Delay	T_{CKIN}		0		40	ns
TX_CLK Duty-Cycle			40		60	%

Tabelle 8: "Setup-Time" und "Hold-Time" der "78Q21x3-DB"-Baustein [21]

Die Beziehung zwischen den Signalen „TX_CLK“, „TX_EN“, „TX_ER“ und „TX_D[3:0]“ wird in der nachfolgenden Abbildung 23 anhand von Waveforms dargestellt. Die Signale „TX_EN“, „TX_ER“ und „TX_D[3:0]“ müssen die Zeiten „TX_{SU}“ und „TX_{HD}“ einhalten, um eine fehlerfreie Übertragung bzw. ein Aufnehmen seitens des „78Q21x3-DB“-Bausteins zu ermöglichen. Hierbei müssen die Signale „TX_EN“ und „TX_D[3:0]“ einen stabilen Wert für eine minimale Zeitdauer von 15ns vor der aktiven Taktflanke des Signals „TX_CLK“ und für eine minimale Zeitdauer von 0ns nach der aktiven Taktflanke des Signals „TX_CLK“ aufweisen.

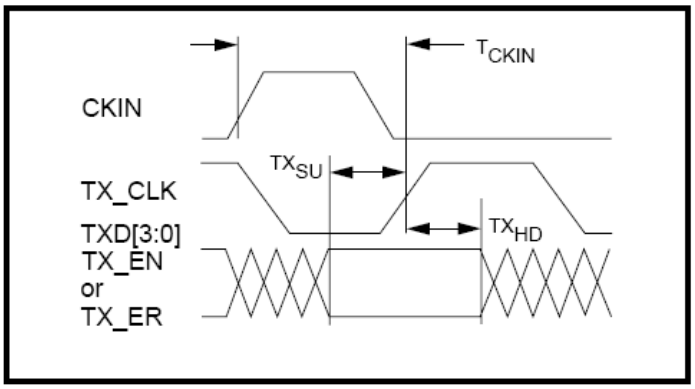

Abbildung 23: "Setup-Time" und "Hold-Time" der "78Q21x3-DB"-Baustein [21]

Die von dem „78Q21x3-DB“-Baustein zur Verfügung gestellten Zeitanforderungen entsprechen den minimalen Zeitanforderungen des IEEE 802.3.

4.2 Entwurf der MII-Crosslink Platine

Das Ziel des Entwurfs der MII-Crosslink-Platine ist es, heraus zu finden, welche Probleme bei der direkten Verbindung von zwei PHYs auf der MII-Schnittstelle-Ebene auftreten. Des Weiteren können die zu übertragenen Ethernet-Pakete anhand eines Logik-Analysers verfolgt werden. Fehlern, die aus der direkten Verbindung auf der MII-Schnittstelle-Ebene aufgetreten sind, können analysiert werden und es können Ansätze zur Lösung dieser Fehler vorgeschlagen werden. In diesem Abschnitt wird ein Entwurf und eine Implementierung der Platine, die die Verbindung auf der MII-Schnittstelle-Ebene realisiert, vorgestellt und erläutert.

Da die Bitübertragungsschicht durch das MicroPHY-DemoBoard implementiert wird, wird die wesentliche Funktionalität der entworfenen MII-Crosslink-Platine sein, die MII-Signale miteinander zu verbinden und das MicroPHY-DemoBoard mit der entsprechenden Stromversorgung zu speisen. Außerdem muss die Platine mit einem MII-Stecker versehen werden, an dem die MicroPHY-DemoBoard angeschlossen werden kann. Um die Signale der MII-Schnittstelle beobachten zu können muss die Platine mit Test-Pins, die mit den MII-Signalen direkt verbunden sind, versehen werden, an denen einen Logik-Analysator angeschlossen werden kann.

Die nachfolgende Abbildung 24 repräsentiert einen Blockschaltplan der MII-Crosslink-Platine. Die zwei auf die Platine befindlichen MII-Stecker sind über die MII-Leitungen mit einander verbunden. An jedem MII-Stecker kann ein MicroPHY-DemoBoard angeschlossen werden.

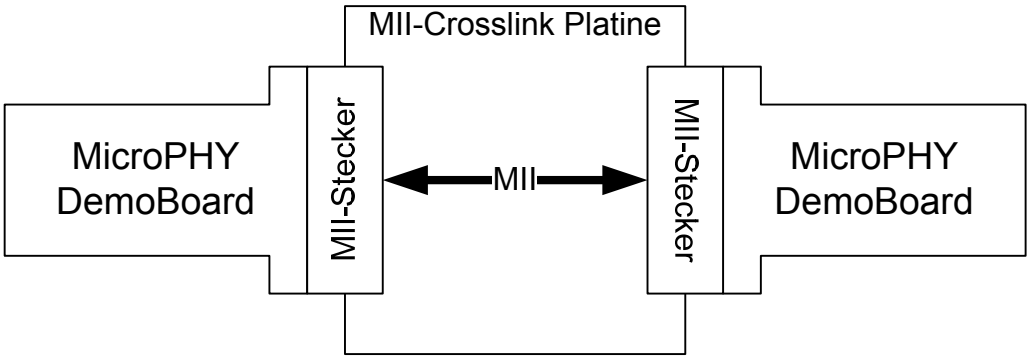

Abbildung 24: Blockschaltplan der MII-Crosslink Platine

Da auf dem Markt keinen 40-Pin-Stecker zur Verfügung steht, wurde einen 50-Pin-Stecker eingesetzt. Die Stecker wurden gegenüber gelegt, so dass die Pins der MII-Schnittstelle über gerade Leitungen verbunden wurden. Die Anleitung zur Anschließung eines 40-Pin-Steckers ist im Kapitel 2.1.5.4 und im Datenplatt des MicroPHY-DemoBoard beschrieben wurden. Die Belegung des 50-Pins-Steckers wurde modifiziert, so dass die Pins 26 und 45 statt den Pins 21 und 40 an der Stromversorgung angeschlossen wurden. Die restlichen Pins des Steckers wurden an die Masse geschlossen. Es ist zu beachten, dass der MicroPHY-DemoBoard an den richtigen Pins des 50-Pins-Steckers angeschlossen werden muss, um einen Kurzschluss zu vermeiden. Zur Sicherheit wurden die MII-Stecker an den nicht verwendeten Pins mit einem Kleber versehen, um einen Fehler während der Anschließung von dem MicroPHY-DemoBoard zu vermeiden.

Die nachfolgende Abbildung 25 stellt die zwei 50-Pins-MII-Stecker dar. Des Weiteren wird anhand der Abbildung 25 gezeigt, wie die Stecker miteinander über die MII-Signale verbunden werden. Hierbei wurden die Pins der „RX_D[3:0]“ Signale eines Steckers an die Pins der „TX_D[3:0]“ Signale der gegenüber liegenden Stecker verbunden. Ebenso wurden die Signale „RX_DV“ und „RX_ER“ an die jeweils entsprechenden Signale „TX_EN“ und „TX_ER“ überführt. Da die Signale „MDIO“ und „MDC“, jeder Stecker keine Anwendung im Rahmen dieser Arbeit finden, wurde sie ohne weitere offen gelassen bzw. mit keinem verbunden.

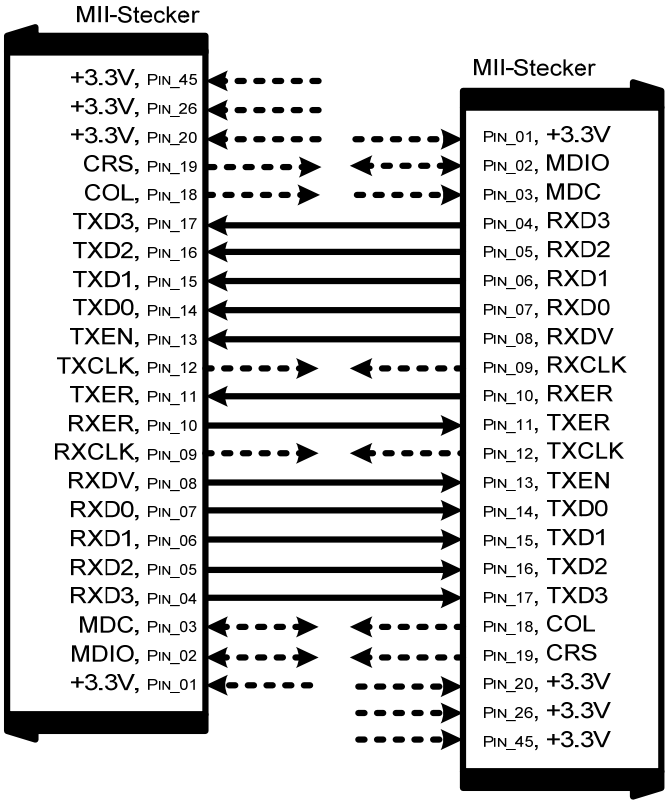

Abbildung 25: Die zwei 50-Pins MII-Steckern

Eine Spannungsregler „LT1086“ der Firma „Linear Technology“ wird eingesetzt, um eine Versorgungsspannung für die MII-Platine von 5V zu 3.3V zur Verfügung zu stellen. Die Abbildung 26 zeigt eine Eingangsspannung mit einem minimalen Wert von 4,75V, während am Ausgang eine Spannung von 3.3V reguliert wird.

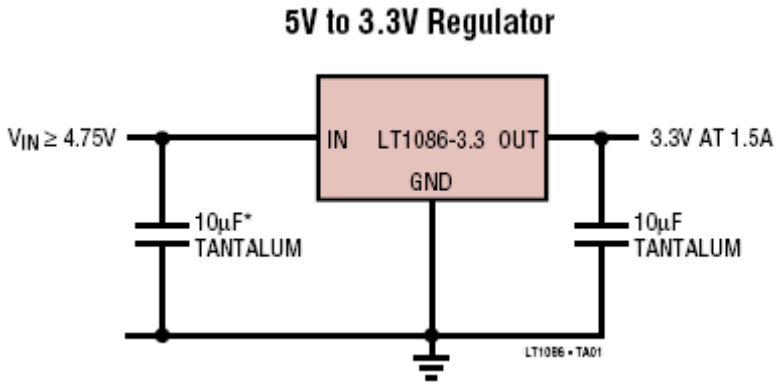

Abbildung 26: „LT1086“-Spannungsregler der Firma "Linear Technology" [23]

Die nachfolgende Abbildung 27 stellt die gefertigte MII-Crosslink-Platine dar. Die MII-Signale wurden mit Test-Stiften versehen, damit die MII-Signale mit Hilfe eines Logik-Analysers untersucht werden können. Des Weiteren wurden die Test-Stifte doppelt angelegt, um bei einer möglichen zukünftigen Trennung der Leitungen die MII-Signale untersuchen zu können.

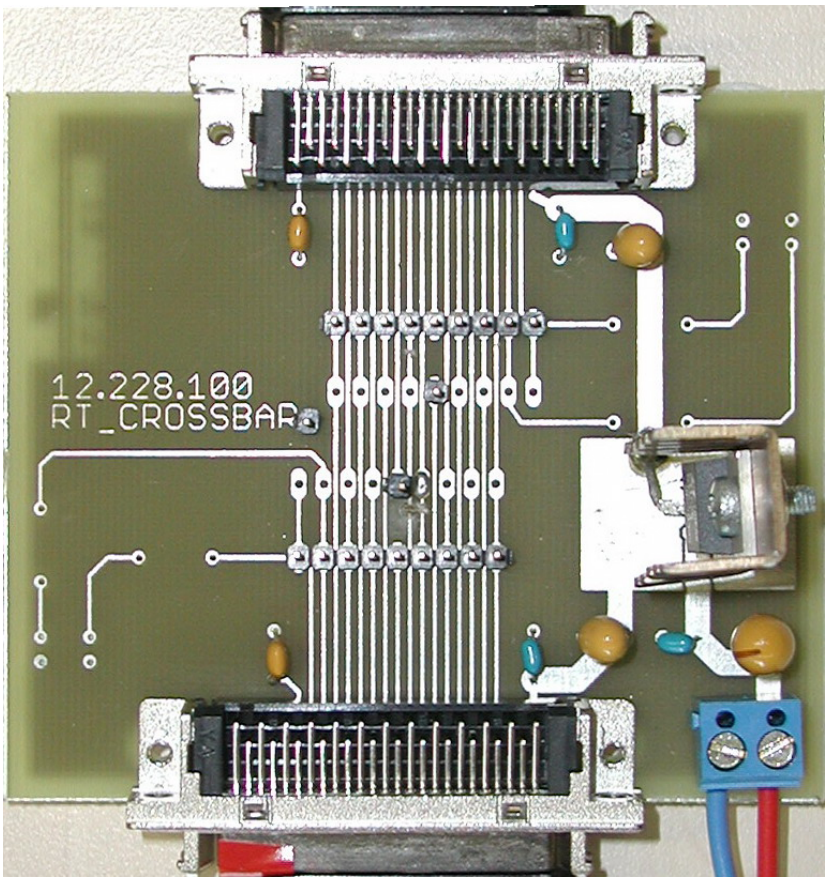

Abbildung 27: Die gefertigte MII-Crosslink-Platine

4.3 Inbetriebnahme der MII-Crosslink Platine

In diesem Abschnitt wird die Übertragung der Ethernet Pakete mit Hilfe der MII-Crosslink-Platine untersucht. Zwei Einsätze wurden durchgeführt. Als Erstes wurde die Platine an das Internet angeschlossen, zweites wurde die Platine zwischen zwei ETHERNUT-Systemen angeschlossen.

4.3.1 Verbindung mit dem Internet

Die MII-Crosslink-Platine wurde zwischen einem Rechner und dem Internet eingesetzt, dabei wurde eine Seite mit einem Rechner und die andere Seite mit dem Internet verkabelt. Die Verkabelung ist durch die Abbildung 28 ersichtlich.

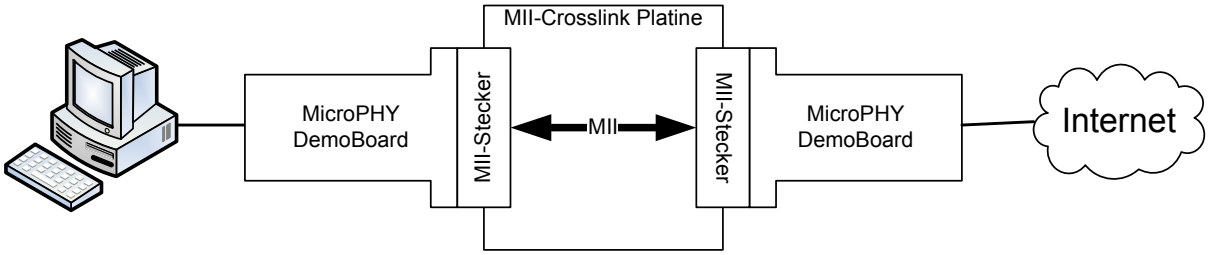

Abbildung 28: Schließung der MII-Crosslink-Platine an das Internet

Eine Verbindung mit dem Internet wurde erfolgreich hergestellt, es waren Pakete zwischen dem Internet und dem Rechner ausgetauscht worden. Es wurde festgestellt, dass durch das Surfen die Geschwindigkeit der Übertragung langsam war. Um das Verhalten der Übertragung genauer zu untersuchen, wurden Software-Tools wie z.B. „ifconfig“ und „ethtool“ unter LINUX-OS eingesetzt.

Die Abbildung 29 repräsentiert die Ausgabe des ausgeführten „ifconfig“-Tools. Das „ifconfig“-Tool gibt aus, wie viel Pakete als empfangene, gesendete und fehlerhaft sind. Nach Durchführung des Tools wurden aus 2463 empfangenen Paketen 282 Pakete als fehlerhaft erkannt.

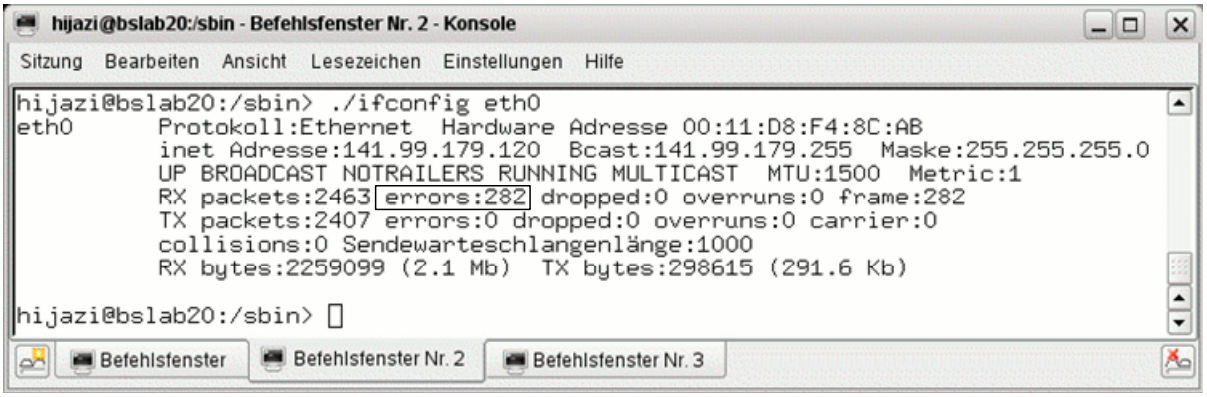

Abbildung 29: Ausgabe der "ifconfig"-Tool

Das „ethtool“-Tool gibt detaillierte Informationen über die übertragenen Paketen. Die nachfolgende Abbildung 30 zeigt die Ausgabe des „ethtool“-Tools an. Hierbei wird eine detaillierte Information der empfangenen und gesendeten Pakete ausgegeben. Aus 2463 empfangenen Paketen wurden 282 Pakete als fehlerhaft erkannt. Aus den 282 fehlerhaften Paketen wurden 15 als „length_errors“, 240 als „crc_errors“ und 27 als „frame_errors“ erkannt.

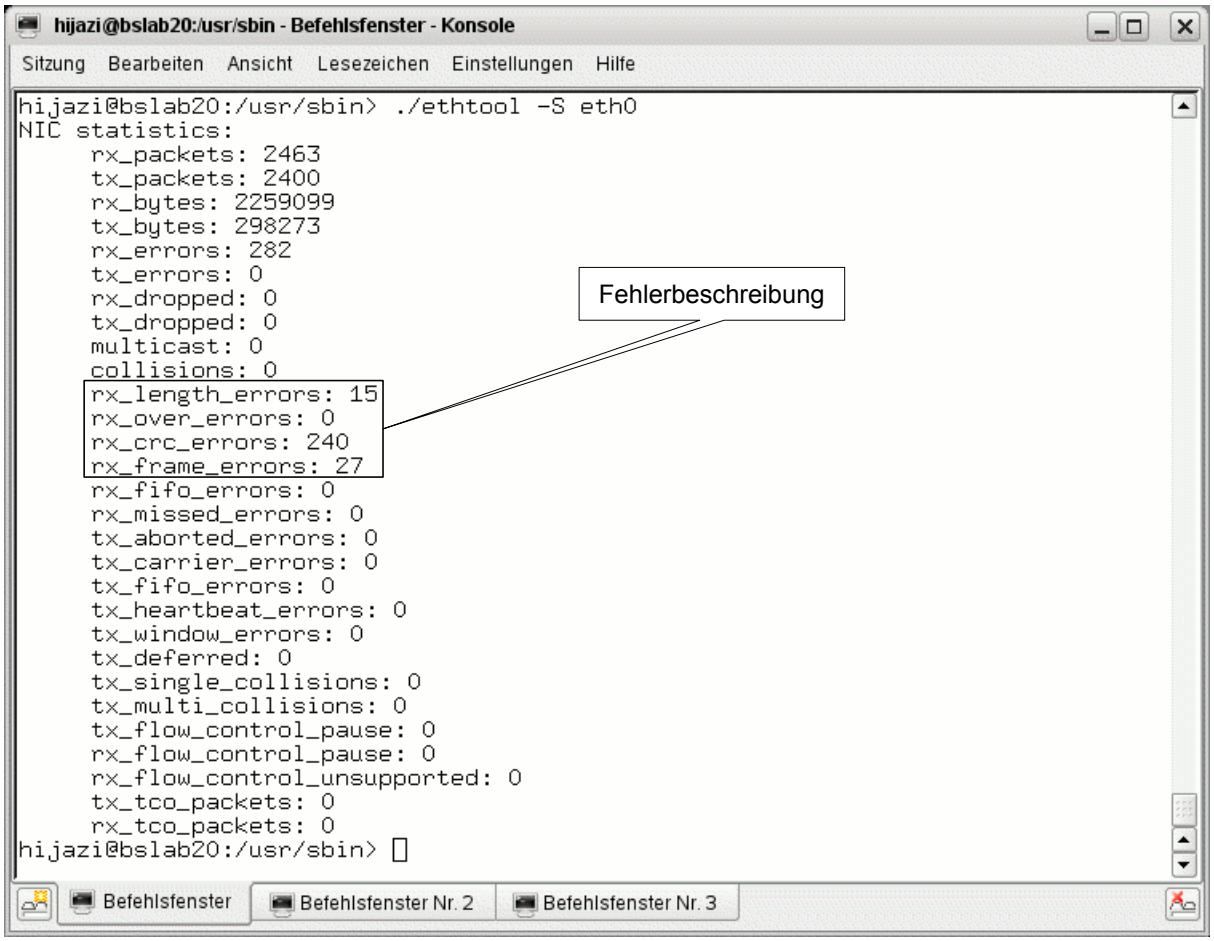

Abbildung 30: Ausgabe der "ethtool"-Tool

Mit „rx_length_errors“ wird gemeint, dass die Länge des empfangenen Frames entweder kleiner als 64 Byte oder größer als 1522 Byte ist. Dies kann durch falsche Auswertung des Typ/Länge-Felds verursacht werden. Die falsche Auswertung des Typ/Länge-Felds findet bei einer Phasenverschiebung der Takt-Signale „RX_CLK“ und „TX_CLK“ bzw. einer Verletzung der Zeitanforderung „Setup-Time“ statt. Dadurch weist der Wert im Typ/Länge-Feld einen anderen Wert als die tatsächliche Größe des Frames auf. Der Fehler „rx_crc_erorrs“ bedeutet, dass die berechnete CRC nicht mit der im Paket gelieferten CRC übereinstimmt. Dies ist auf die fehlerhafte Erkennung der empfangen Bits, welches durch die Verletzung der Zeitanforderungen „Setup-Time“ verursacht wurde zurückzuführen. Die Fehler „rx_frame_errors“, die als „Alignment Error“ bezeichnet werden, deuten darauf hin, dass ein empfangenes Frame, eine falsche CRC aufweist und dessen Länge kein ganzzahliges Vielfaches von 8 Bit beträgt. Dieser Fehler wird verursacht, wenn das letzte Nibble eines Frames falsch ausgewertet wird. Wird am Ende des Frames der Wert des Signals „RX_DV“ von ´1´ nach ´0´ geändert, kann das letzte Nibble bei einer Phasenverschiebung und Verletzung der Zeitanforderung „Setup-Time“ nicht erkannt werden oder zwei Mal durch den nachfolgenden Takt erkannt werden.

4.3.2 Einsatz der Ethernut & Logik-Analyser

Da die Software-Tools nur die Anzahl der fehlerhaften Pakete und keine weiteren Informationen über die Art des Fehlers geben können, wird in diesem Abschnitt auf die Signal-Ebene der übertragenen Pakete untersucht. Hierbei werden die Signale der MII-Schnittstelle mit Hilfe eines Logik-Analysers durch die Anschließung an den Test-Stiften, die sich auf die MII-Crosslink-Platine befinden, genauer untersucht.

Um eine genaue Auswertung der MII-Signale eines übertragen Paketes und eine Steuerung des Flusses der Pakete durchführen zu können, müssen die Sende-Prozesse der Pakete gesteuert werden. Hierbei muss überprüft werden ob die gesendeten Pakete angekommen sind oder nicht. Durch die Anschließung der MII-Crosslink-Platine an ein Netzwerk werden unerwünschte Netzwerk-Pakete beispielsweise ARP-Pakete gesendet. Dadurch wird es zu schwer, die Übertragung der Pakete einzeln auf fehlerhafte Stellen zu untersuchen. Es ist erwünscht den Sende-Prozess eines Pakets zu steuern, um zu wissen, wann und welches Paket gesendet wurde. Dafür wurde ein Ethernut-System eingesetzt, das in der Lage ist, programmierbare Ethernet-Pakete mit modifiziertem Abstand zu senden. Des Weiteren kann das System Ethernet-Pakete empfangen und die Ergebnisse der analysierten Pakete über ein Terminal übergeben. Durch den Einsatz des Ethernut-Systems ist es in der Lage einzelne Pakete gezielt zu senden und zu empfangen. Dadurch wird es einfach, diese Pakete einzeln zu untersuchen. Die nachfolgende Abbildung 31 repräsentiert eine Ethernut-Platine, die sowohl aus einem RJ45-Port (für Verbindung zur Ethernet) als auch einer seriellen Schnittstelle (für Verbindung zu einem Terminal) besteht.

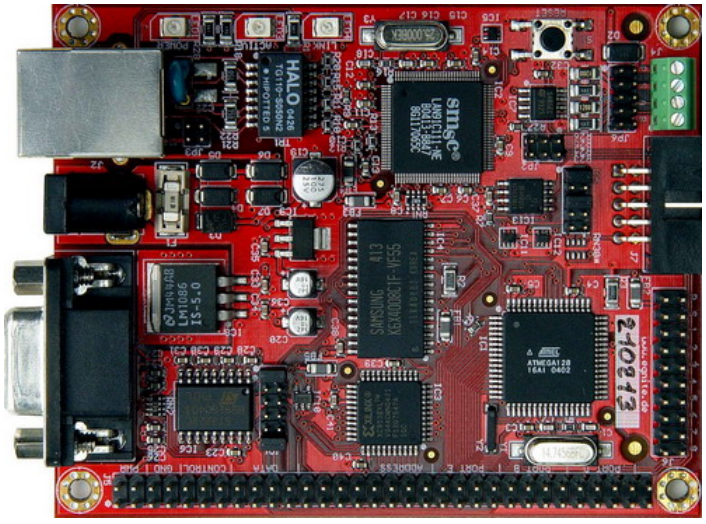

Abbildung 31: Ethernut-Platine [24]

Um die Untersuchung durchführen zu können, wurden zwei Ethernut-Systeme eingesetzt, einer als Sender und der andere als Empfänger. Die Verschaltung wurde über der MII-Crosslink-Platine, an deren Stecker jeweils eine MicroPHY-Platine angeschlossen wurde, realisiert. An einer Seite der MII-Crosslink-Platine wurde Ethernut-System als Sender angeschlossen, an der

anderen Seite der MII-Crosslink-Platine wurde ein Ethernut-System als Empfänger angeschlossen. Das Empfänger-Ethernut-System wurde über die serielle Schnittstelle an einem Rechner angeschlossen, dadurch konnten durch ein Hyper-Terminal die ankommenden Ethernet-Pakete analysiert und betrachtet werden. Zum Analysieren der MII-Signale auf der Signal-Ebene wurden die Test-Stifte der MII-Crosslink-Platine an die Kanäle des Logik-Analysers angeschlossen. Diese Verschaltung ist in der Abbildung 32 ersichtlich.

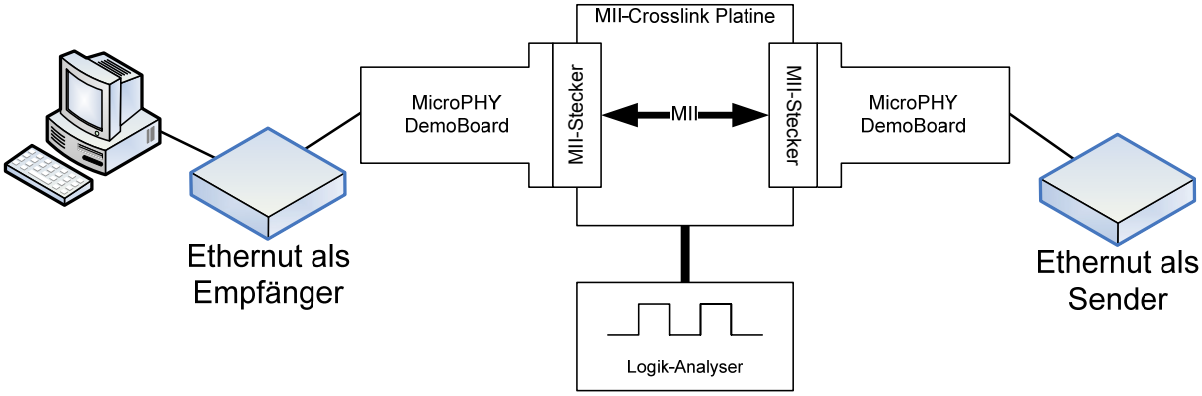

Abbildung 32: MII-Crosslink-Platine zwischen zwei Ethernut-Systeme

Die Zusammenschaltung der Komponenten wurde in Betrieb genommen und die Waveform der MII-Signale analysiert. Im nächsten Abschnitt werden die Ergebnisse der Analyse diskutiert.

4.4 Auswertung der MII-Crosslink-Platine

Das logische Verhalten der MII-Signale wird mit Hilfe des Logik-Analysers dargestellt. Im Rahmen dieses Versuchs werden die wichtigsten Signale der MII-Schnittstelle analysiert. Hierfür sind die Signale „RX_CLK“, „RX_DV“ und „RX_D[3:0]“ der Receiver-Schnittstelle und das Signal „TX_CLK“ der Transmitter-Schnittstelle von Bedeutung.

In diesem Abschnitt werden zwei Aspekte der Analyse diskutiert, hierbei wird nämlich im Abschnitt 4.4.1 auf die Phasenverschiebung der Signale „TX_CLK“ und „RX_CLK“ und die Verletzung der Zeitanforderungen geachtet. Ebenso wird die Vollständigkeit der Präambel im Abschnitt 4.4.2 analysiert. Im Abschnitt 4.4.3 folgt eine Zusammenfassung der durch die Versuche festgelegten Ergebnisse. Es wird erwartet, dass die Phasenverschiebung bzw. die Verletzung der Zeitanforderungen „Setup-Time“ und „Hold-Time“ die Gründe für die fehlerhafte Übertragung der Pakete sind.

4.4.1 Der asymmetrische Verlauf der Signale (TX_CLK & RX_CLK)

Für eine fehlerfreie Übertragung der Ethernet-Pakete müssen die Zeitanforderungen, die von der MicroPHY-DemoBoards festgelegt sind und im Abschnitt 4.1 diskutiert wurden, nämlich die „Setup-Time“ und „Hold-Time“, eingehalten werden. In diesem Abschnitt werden die Gründe, welche zu einer fehlerhaften Übertragung der Pakete geführt haben, anhand von Auszügen der

Waveform der Logik-Analyser aufgezeigt. Die Signale „RX_DV“ und „RX_D[3:0]“ an der Receiver-Schnittstelle eines Steckers werden mit den Signalen „TX_EN“ und „TX_D[3:0]“ an der Transmitter-Schnittstelle eines Steckers verbunden. Somit werden die Daten in Beziehung zum Signal „RX_CLK“ empfangen und müssen in Beziehung zum Signal „TX_CLK“ gesendet werden. Hierbei entsteht das Problem, dass die zu übertragenden Daten von zwei Takt-Signalen gesteuert werden, die seitens des PHYs als Ausgangssignale betrieben werden. Demzufolge kann durch die mögliche entstehende Phasenverschiebung zwischen den Takt-Signalen „RX_CLK“ und „TX_CLK“ eine Verletzung der Zeitanforderungen „Setup-Time“ und „Hold-Time“ stattfinden.

Zunächst wird das Verhältnis zwischen dem Signal „TX_CLK“ auf der Transmitter-Seite und den Signalen „RX_DV“ und „RX_D[3:0]“ auf der Receiver-Seite diskutiert. Hierbei wird auch auf die „Duty-Cycle“ der beiden Takt-Signale „TX_CLK“ und „RX_CLK“ eingegangen. Wie bereits erwähnt fordert das PHY in Beziehung zum Takt-Signal „TX_CLK“ eine „Setup-Time“ von minimal 15ns und eine „Hold-Time“ von minimal 0ns.

In der Abbildung 33 ist eine deutliche Verletzung der Zeitanforderung „Setup-Time“ und „Hold-Time“ des Takt-Signals „TX_CLK“ an der Stelle vom zweitem Nibble des SFDs zu sehen. Bei ① wird der Gültigkeitsbereich des zweiten Nibbles des SFDs gezeigt. Dieser Bereich muss einen stabilen Zustand seitens des Takt-Signal „TX_CLK“ aufweisen, um ihn fehlerfrei erkennen zu können. Bei der steigenden Flanke des Signals „TX_CLK“ [bei②] beginnt der Gültigkeitsbereich des zweiten Nibble des SFDs vor den minimal geforderten 15ns (Zeit für „Setup-Time“). Geht man davon aus, dass das SFD-Nibble im nächsten Takt-Signal erkannt würde, so betrachtet man die zweite steigende Flanke des nächsten Takt-Signals [bei③]. Ebenso weist der Gültigkeitsbereich des SFD-Nibbles keinen stabilen Zustand bei der steigenden Flanke des Signals „TX_CLK“ auf. Diese Übertragung führt dazu, dass die SFD-Nibble nicht richtig erkannt werden, demzufolge wird der Anfang des Frames nicht erkannt und somit findet eine Übertragung eines fehlerhaften Pakets statt.

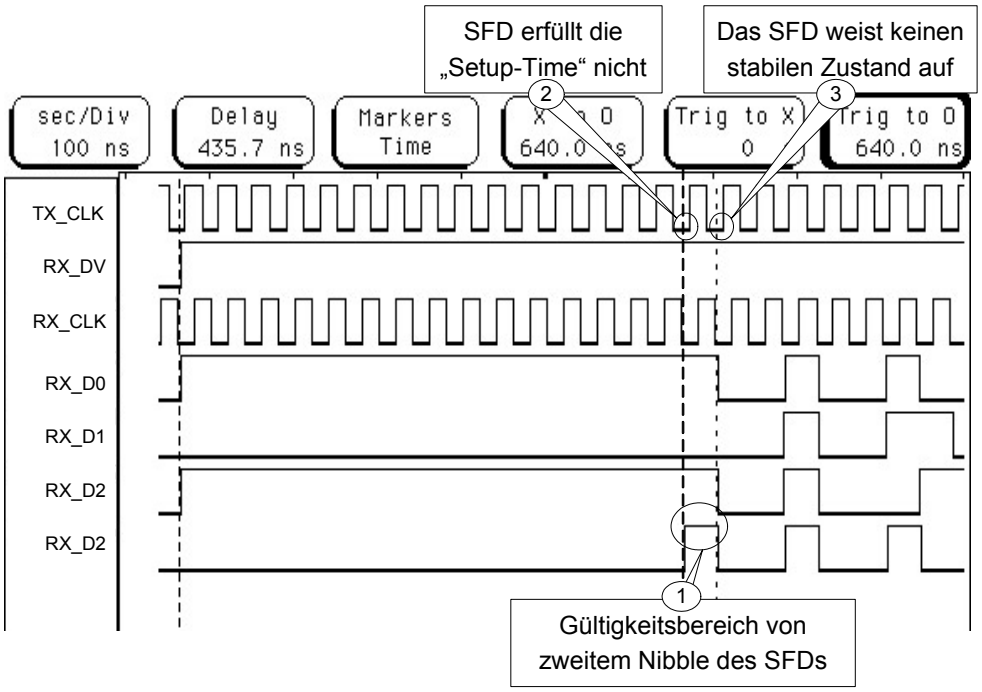

Abbildung 33: SFD-Teil eines fehlerhaften Frames

Die Abbildung 34 zeigt den Abschnitt des FCS-Teils eines Pakets. Hierbei wird deutlich, dass eine Verletzung der Zeitanforderung zum Signal „TX_CLK“ zur fehlerhaften Übertragung führt. Das Signal „RX_D2“ [bei①] verletzt die Zeitanforderung „Setup-Time“ in Beziehung zum Signal „TX_CLK“. Die Zeit-Achsen „X“ und „0“ [bei②] markieren den geforderten Zeitbereich von 15ns der „Setup-Time“ [bei③]. Die Zeit-Achse „X“ markiert den Zeitpunkt, ab welchem ein Nibble starten muss, um ihn fehlerfrei zu erkennen. Es ist anhand der Waveform zu erkennen, dass das Signal „RX_D2“ nicht die geforderte Zeitbedingung erfüllt, und somit kann es nicht korrekt interpretiert werden. Dabei wird es als Null statt Eins erkannt. Das hier vorgestellte Beispiel verursacht einen Fehler, der bei der Überprüfung der CRC auftaucht. Dies ist auf die falsche Interpretierung des Signals „RX_D2“ zurückzuführen [bei①]. Anhand der Abbildung 34 sind die unterschiedlichen Werte der „Duty-Cycle“ des Signals „TX_CLK“ [bei④] und das Signals „RX_CLK“ [bei⑤] zu erkennen. Der Wert der „Duty-Cycle“ hat in der in diesem Beispiel bedeutendem Einfluss auf die Übertragung. Wenn das Takt-Signal „TX_CLK“ das gleiche „Duty-Cycle“ wie vom Takt-Signal „RX_CLK“ besitzt, kann dieses Paket in diesem Beispiel als fehlerfrei erkannt werden.

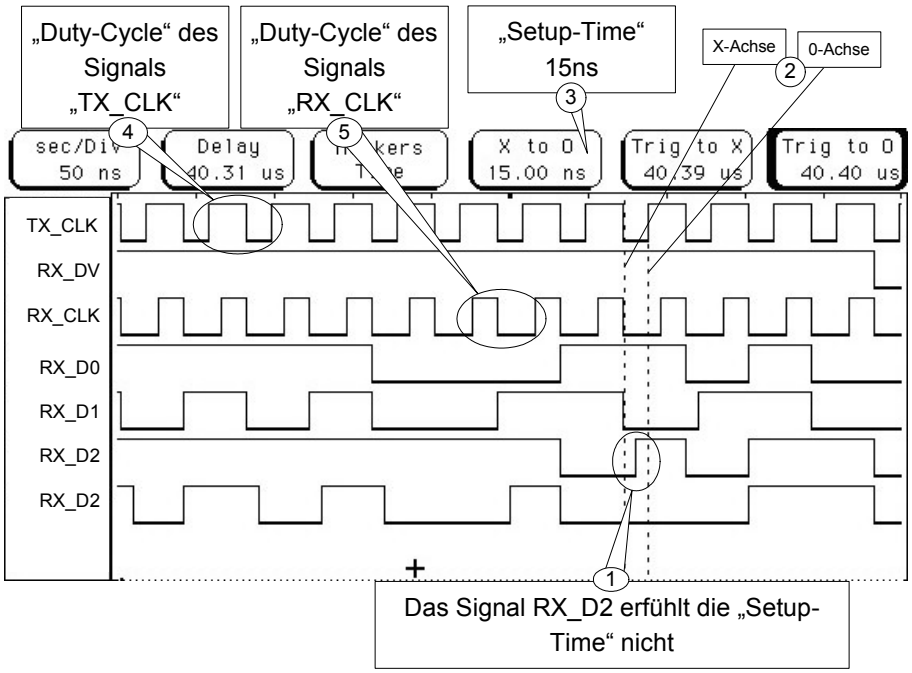

Abbildung 34: CRC-Teil eines fehlerhaften Frames

Die nachfolgende Abbildung 35 zeigt, wie bei einem Takt-Signal beispielweise „RX_CLK“ oder „TX_CLK“ das „Duty-Cycle“ unterschiedliche Werte haben kann, dadurch können an manche Stellen die Zeitanforderungen „Setup-Time“ und „Hold-Time“ erfüllt sein. Das „Duty-Cycle“ des Signals „TX_CLK“ (bei①) hat einen kleineren Wert als die anderen Stellen. Ebenso verhält sich das Signal „RX_CLK“, das „Duty-Cycle“ (bei②) ist kleiner als die anderen Stellen. Dies hat die Auswirkung, dass an manchen Stellen die empfangenen Nibbles einen stabilen Zustand (bei③) und an anderen Stellen keinen stabilen Zustand aufweisen (bei④). Wäre die geforderte Zeit „Setup-Time“ kleiner als 15ns und eine „Hold-Time“ von 0ns, dann wird das Nibble vom Takt-Signal „TX_CLK“ (bei④) und zugleich von nächstem Takt erkannt. Im Abschnitt 4.3.1 wurde erklärt, dass beim „Alignment Error“ die Länge eines Frames kein ganzzahliges Vielfaches von 8 Bit beträgt. Dies kann bedeuten, dass ein Nibble fehlen oder überflüssig sein kann. Bei ⑤ und ⑥ ist dieser Fall zu sehen. Das letzte Nibble (bei⑥) des Frames kann nicht vom letzten Takt erkannt werden, weil die Wertänderung das Signals „RX_DV“ vor der steigenden Flanke des Takt-Signals „TX_CLK“ stattgefunden hat.

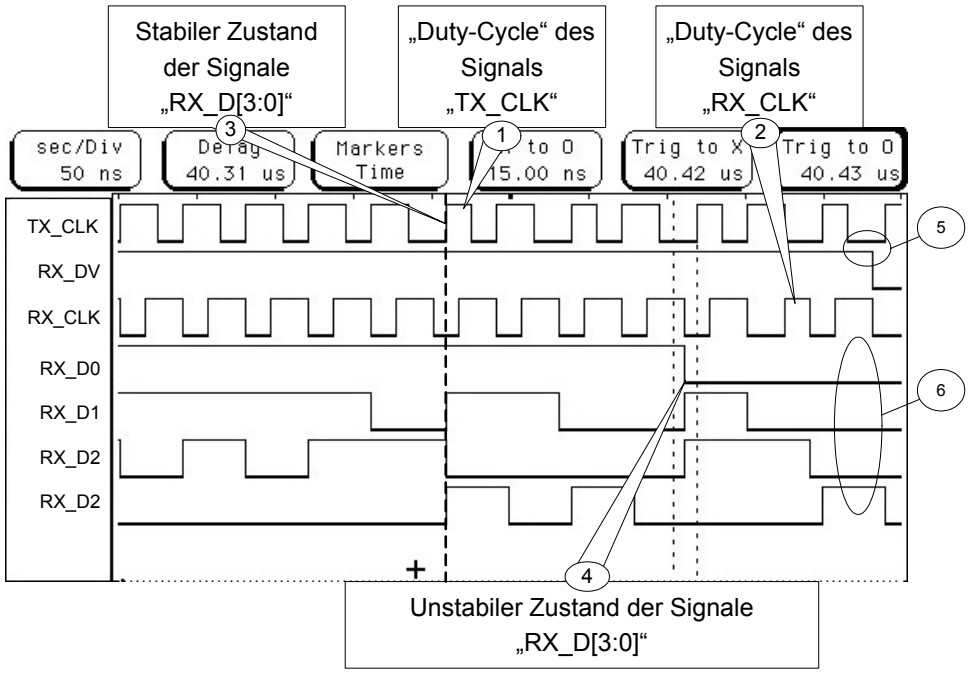

Abbildung 35: Verhaltung von "Duty-Cycle"

Der Grund für die in diesem Abschnitt vorgestellten Probleme liegt darin, dass die Übertragung der Daten von den zwei Takt-Signalen „RX_CLK“ und „TX_CLK“ gesteuert wird, die als Ausgänge betrieben werden. Dadurch kann eine Phasenverschiebung entstehen. Dem zu Folge werden die Zeitanforderungen „Setup-Time“ und „Hold-Time“, die seitens der Daten zum Takt-Signal „TX_CLK“ für fehlerfreie Übertragung eingehalten werden müssen, verletzt werden. Des Weiteren wird gezeigt, dass das „Duty-Cycle“ einen Einfluss auf die Übertragung der Daten haben kann.

Generell lassen sich diese Übertragungen als „asynchron“ bezeichnen. Dieses Problem lässt sich durch die Verwendung von FIFOs lösen [27]. “A FIFO is a memory subsystem in which a data sequence is written and retrieved in exactly the same order. No explicit addressing is required, and the write and read operations can be completely independent, using unrelated clocks.”[29]. Ein herkömmliches FIFO wird durch die Signale „empty“ und „full“ gesteuert. Wenn das FIFO Full ist (full=1) können keine Daten im FIFO geschrieben werden, bevor alle Daten gelesen werden. Da die Übertragung der Daten auf die MII-Ebene nicht eingehalten werden dürfen, muss die Funktionalität des FIFOs angepasst werden. Erwünscht ist ein Ringspeicher komponiert mit der Funktionalität eines FIFOs. Dafür muss die Architektur eines FIFO ohne die Signale “empty” und „full“ betrachtet werden. „… The design seems to be trivial, using a RAM with two independently clocked ports, one for writing, one for reading, plus two independent address counters to steer write and read data.“ [29]. Zur Lösung des Phasenverschiebungsproblems wird die Architektur eines FIFOs eingesetzt. Die Implementierung wird in der VHDL-Sprache für einen FPGA-Baustein realisiert und wird im Kapitel 5.4 diskutiert. Das Problem des „Duty-Cycle“ soll anhand der o. g. Lösungsansätze behoben werden.

4.4.2 Vollständigkeit der Präambel

Im Kapitel 2.1.5.1 wurde erklärt, dass gewisse Teile der Präambel fehlen können. Dies kann durch die Herausfilterung der Taktinformationen im PHY-Baustein entstehen [14][15]. Es wurde nach diesem Fall die logische Darstellung der übertragenen Pakete untersucht. Die meisten Waveforms der Übertragungen haben gezeigt, dass der PHY die eingehenden Pakete mit vollständiger Präambel empfangen hat. Die Rückgewinnung der Präambel hängt immer von dem PHY-Baustein ab. In diesem Abschnitt werden zwei Beispiele gezeigt, in denen eine zeitliche Verzögerung bei Ausgabe der Präambel liegt.

Die Abbildung 36 und die Abbildung 37 repräsentieren die Abschnitte eines Frames an der Stelle der zweiten Nibble des SFDs. Bei 100 MBit/s Ethernet wird für die Übertragung der Präambel und des SFDs mindestens 640 ns benötigt. Die Abbildung 36 zeigt, wie das SFD in der 632 ns empfangen wurde. Die Messung der Zeit beginnt bei der Wertänderung des Signals „RX_DV“ von ´0´ nach ´1´. Dies bedeutet, dass der PHY die Präambel nicht an der richtigen Stelle ausgibt. Dies ist anhand der zeitlichen Verzögerung zu erkennen.

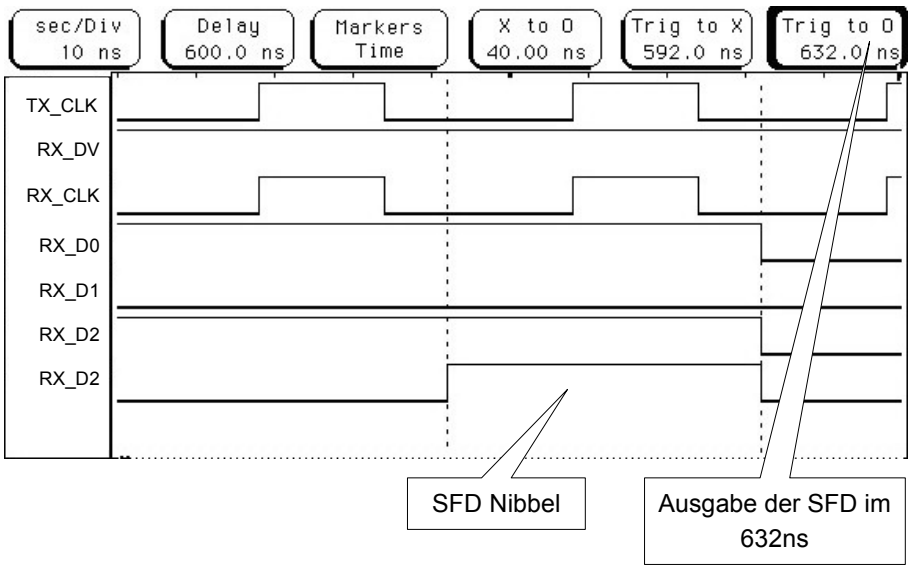

Abbildung 36: Ende des SFDs bei 632ns

Ebenso wurde der Frame, der durch die Abbildung 37 dargestellt wird, von dem Empfänger-Ethernut-System als fehlerfrei erkannt, wohingegen die zweite Nibble des SFDs in 616ns empfangen wurde. Hierbei findet ebenso eine zeitliche Verzögerung statt, die durch eine Kaskaden-Übertragung begründet sein kann, so dass gewisse Teile der Präambel verloren gehen.

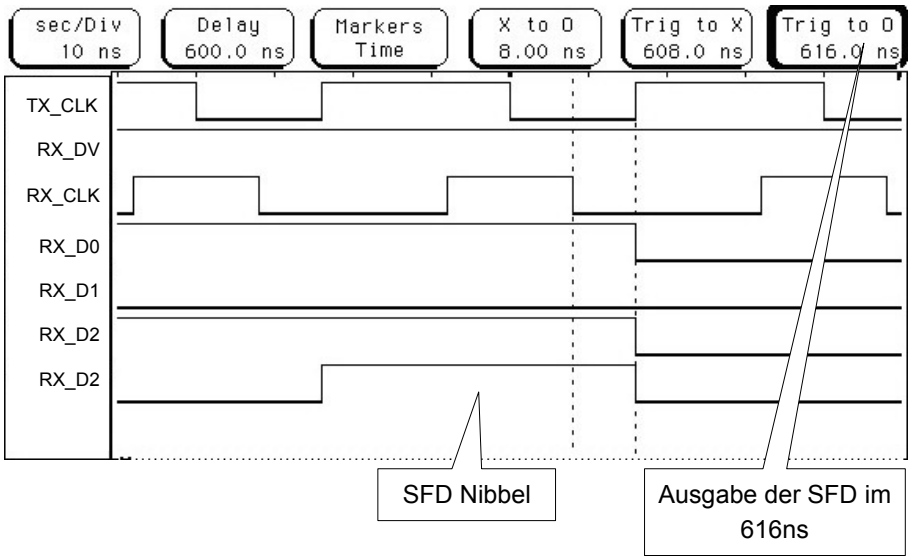

Abbildung 37: Ende des SFDs bei 616ns

Fehlende Teile der Präambel haben keinen Einfluss auf die Erkennung eines Frames, da der MAC nur die zwei Nibbles des SFDs benötigt, um den Anfang eines Frames zu erkennen. Bei der Kaskaden-Übertragung der Pakete können die zwei Nibbles des SFDs verloren gehen. Dadurch kann das Frame nicht erkannt werden. Um den Standard zu behalten und den Fall, dass das SFD verloren geht, zu vermeiden, muss die Präambel neu konstruiert werden. Ein Lösungsansatz ist, nach Erkennung des zweiten Nibbles des SFDs eine vollständige Präambel auszugeben und anschließend die Daten anzuhängen. Dieser Lösungsansatz wird im Kapitel 5.4 in der VHDL-Sprache für eine Implementierung auf einem FPGA-Baustein realisiert.

5 Modellierung in VHDL

Zur Lösung der Probleme, die durch die Realisierung der RTC (Real Time Crossbar) aufgetreten können und die im Kapitel 4.4 diskutiert wurden, werden in diesem Kapitel digitale Schaltungen vorgestellt, die in der VHDL-Sprache entworfen sind und auf einem FPGA-Baustein realisiert werden können. Die entworfenen digitalen Schaltungen, die im Rahmen dieses Kapitels als VHDL-Module bezeichnet werden, wurden durch Simulation im Abschnitt 5.5, Synthese im Abschnitt 5.6 und Zeit-Analyse im Abschnitt 5.7 verifiziert.

Zum besseren Verständnis der in diesem Kapitel durchgeführten Arbeit wird im Abschnitt 5.1 auf die Grundlagen des FPGAs eingegangen. Im Abschnitt 5.2 wird einleitend der Entwurfsablauf an einem FPGA dargestellt. Um die Programmierung der VHDL-Module, die im Abschnitt 5.4 erklärt werden, besser nachvollziehen zu können, werden die Grundlagen der VHDL vorab im Abschnitt 5.3 ausgeführt.

5.1 FPGA Grundlagen

Zum Aufbau eines Systems mit anwendungsspezifischer Funktion kann einer der Entwurfsansätze „Standardbausteine“, „ASICs“ und „FPGA“ verwendet werden. In diesem Abschnitt wird zunächst der Unterschied zwischen den Entwurfsansätze dargestellt. Danach wird in den strukturellen Aufbau eines FPGA-Bausteins eingeführt.

Durch Standardbausteine, wie z.B. Mikroprozessoren und Mikrocontroller, kann ein Mikroprozessor-System aufgebaut werden, dessen anwendungsspezifische Funktion sich aus der Erstellung eines Programms für den Prozessor ergibt. Die Standardbausteine bieten vielfältige Einsatzmöglichkeiten und haben niedrige Herstellungskosten, weil sie in großen Stückzahlen hergestellt werden. Mikroprozessoren weisen heutzutage eine hohe Rechengeschwindigkeit auf, jedoch sind sie naturgemäß nicht für Anwendungen geeignet, bei denen eine komplizierte parallele Berechnung erforderlich ist.

Anwendungsspezifische integrierte Schaltungen (ASICs) (engl.: Application Specific Integrated Circuit) sind Bausteine, die mit einer anwendungsspezifischen Funktion für bestimmte Anwendungen hergestellt werden. Die ASICs sind zwar sehr schnell, haben aber höhere Herstellungskosten, da sie in kleinen Stückzahlen produziert werden.

FPGAs (engl.: Field Programmable Gate Array) sind Bausteine, die zum Aufbau digitaler und logischer, vom Anwender entwickelter Schaltungen dienen. Sie kombiniert die Vorteile der beiden zuvor genannten Ansätze. Erwünschte anwendungsspezifische Funktionen können in einen FPGA implementiert werden. Sie arbeiten dann mit einer höheren Geschwindigkeit als die Standardbausteine. FPGAs sind im Gegensatz zu den Standardbausteinen in der Lage, Operationen sowohl sequentiell als auch parallel durchzuführen. Die Entwurfskosten des FPGAs

entsprechen im Wesentlichen den Entwurfskosten des ASICs, jedoch werden FPGAs mit steigender Stückzahl preiswerter hergestellt [26].

Die nachfolgende Abbildung 38 zeigt die drei vorgestellten Ansätze und ihre Anordnung nach Flexibilität, Geschwindigkeit und Herstellungskosten.

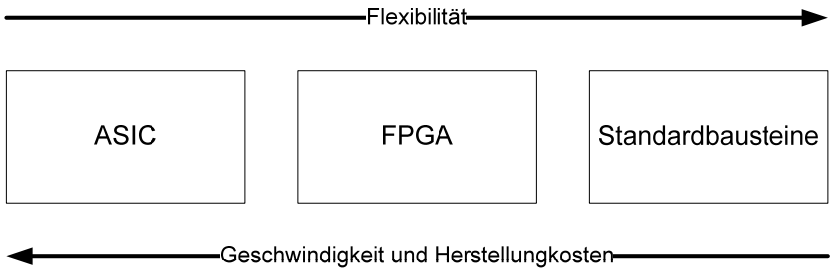

Abbildung 38: Anordnung von Bausteinen [26]

FPGAs bestehen im Wesentlichen aus einzelnen, in einer regelmäßigen Feldstruktur (engl.: Array) angeordneten Logikblöcken und einem Netzwerk von Verbindungsleitungen zwischen den Logikblöcken. Die Funktion der Logikblöcke und die Auswahl der Verbindungen sind durch den Anwender konfigurierbar [25].

Zur Konfiguration der auf dem FPGA vorhandenen (Hardware-) Ressourcen werden Konfigurationsdaten elektrisch in den FPGA geschrieben. Es hat sich eingebürgert, den Begriff Programmierung anstatt Konfigurierung des FPGA zu benutzen, obwohl kein ablauffähiges (Software-) Programm wie bei einem Mikroprozessor geschrieben wird [26].

Die Abbildung 39 zeigt die Architektur des FPGA-Bausteins „Sparten-3“ der Firma Xilinx. Die Architektur der „Sparten-3“ besteht aus Logikblöcken, die als CLB (engl.: Configurable Logic Block) bezeichnet werden. An den Seiten befinden sich RAM-Blöcke, die als Speicherblöcke in einem Design verwendet werden können. Im Randbereich des Chips befinden sich die sogenannten Ein-/Ausgangs-Blöcke (engl.: Input/Output-Block, I/O-Block, IOB), die zur Anpassung der internen Chipsignale mit den externen Signalen dienen. Ein I/O-Block besetzt zumeist ein Register zum Zwischenspeicher der Ein- und Ausgangssignale [25].

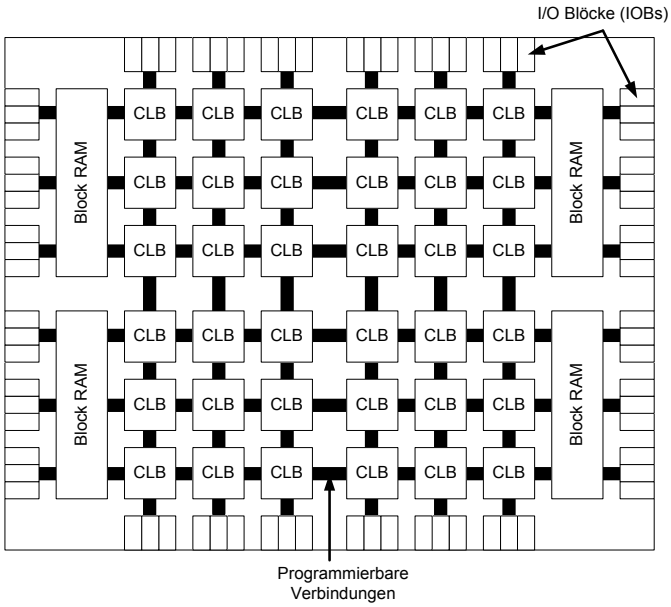

Abbildung 39: Architektur eines FPGA [nach 25]

Die CLBs haben je nach Hersteller unterschiedliche Bezeichnungen und einen unterschiedlichen Aufbau. Sie werden z.B. bei der Firma „Altera“ als „Logic Element“ (LE); bei die Firma „Actel“ als „Logic Tile“ (LT) bezeichnet. Die nachfolgende Abbildung 40 zeigt den Aufbau einer CLB der „Sparten-3“-Familie. Eine CLB besteht aus vier Slices, die der Implementierung einer Funktion dienen. Innerhalb einer CLB ist ein lokales Routing möglich, das ein einfaches Feedback zwischen Slices im selben und benachbarten CLB realisiert; dieses Routing wird in hoher Geschwindigkeit durchgeführt. Zu jeder CLB gehört eine „Switch Matrix“, die ein Routing-Pfad zum entfernten CLB ermöglicht. Ist eine CLB-Zelle weit entfernt, führt dies zum langsamen Design [25].

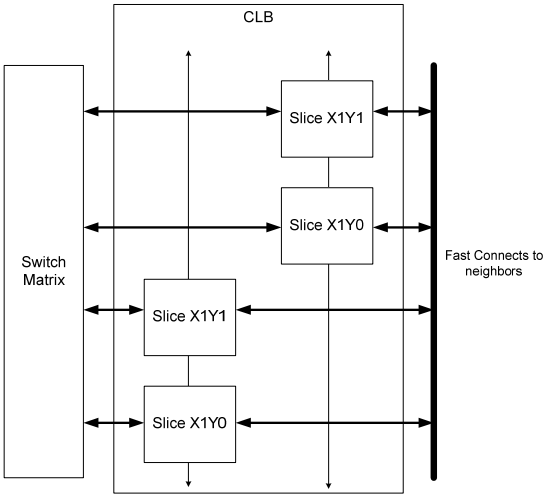

Abbildung 40: Aufbau eines CLBs [nach 36]

Die einzelnen Slices innerhalb einer CLB können zur Implementierung von Logik, Arithmetik, ROM und Distributed RAM Funktionen eingesetzt werden. Die nachfolgende Abbildung 41

präsentiert einen einzelnen Slice. Ein Slice besteht aus zwei LUTs (Look-Up-Table) und zwei zugehörigen Flipflops. Die Flipflops können mit Set- oder Reset-Funktionen (PRE, CLR) gesteuert werden. Zur Unterstützung von arithmetischen Funktionen sind zwei Carry-Einheiten in einem Slice vorhanden [25].

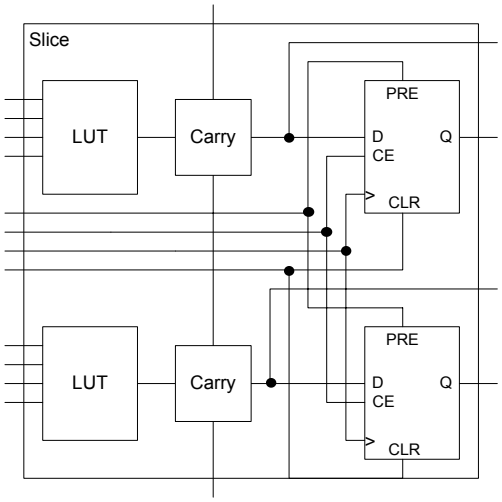

Abbildung 41: Aufbau einer Slice [nach 25]

Die tatsächliche Implementierung der Gatterfunktionen findet in einer LUT (Look-Up-Tabelle) statt. Die nachfolgende Abbildung 42 verdeutlicht eine LUT, die typischerweise vier Eingänge besitzt. Eine LUT besteht aus SRAM-Zellen, die in einem Feld angeordnet sind. Die Speicher-Zellen werden während der Konfiguration des FPGA mit Hilfe eines Decoders durch die Variable (S) ausgewählt und über die Eingangsdatenleitung (D) beschrieben. Im Betriebsfall werden die SRAM-Zellen von dem Multiplexer auf den Ausgang geschaltet [25].

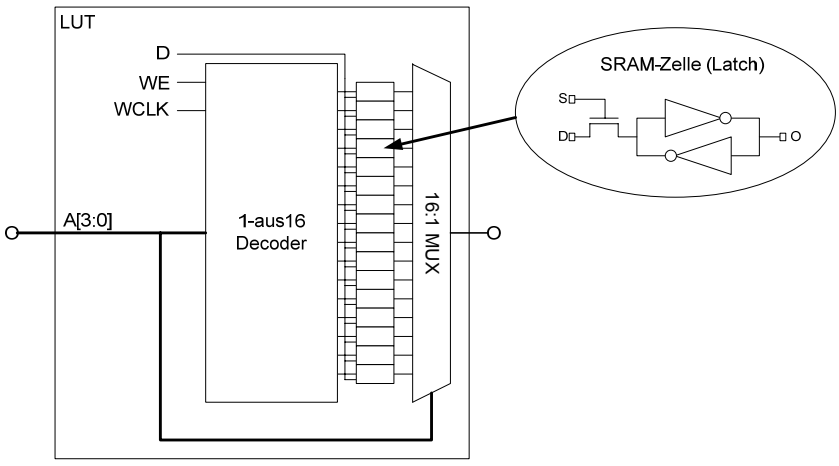

Abbildung 42: Aufbau einer LUT [nach 25]

Die Abbildung 43 zeigt, wie CLBs an den Verbindungsleitungen angeschlossen werden und wie die horizontalen und vertikalen Leitungen miteinander verbunden werden können. An den

Kreuzstellen der horizontalen und vertikalen Leitungen liegen C-Box (Connetion Box) und S-Box (Switch Box).

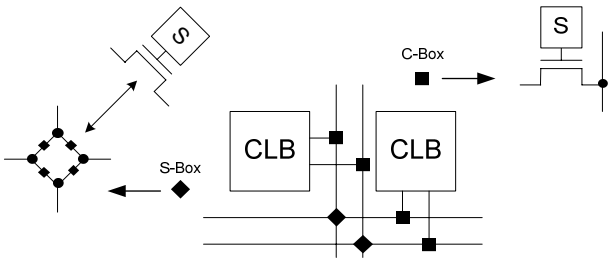

Abbildung 43: Verbindung von CLB an der Leitungen [nach 25]

Die Eingänge und die Ausgänge der CLBs werden über die C-Box, die aus einem Transistor und eine an der Gate des Transistors angeschlossenen Speicherzelle besteht, an die globale Verdrahtung in den horizontalen und vertikalen Leitungen angeschlossen. Die Verbindung der horizontalen und vertikalen Leitungen findet über die S-Box statt, welche aus vier Transistoren für je eine Speicherzelle besteht. Der Wert der Speicherzellen in der C-Box und S-Box wird während des Programmierungsprozesses des FPGA festgelegt [25].

FPGA-Bausteine werden anhand der Anzahl der CLBs, der LUTs und der Pins charakterisiert. Beispielsweise besetzt der Xilinx „Spartan-3“ (XC3S400-144) 896 CLBs, jeder CLB enthält 4 Slices, jeder Slices enthält 2 LUTs; dies macht 7168 LUTs. Des Weiteren besetzt der Baustein 97 User-Pins und 8 Pins für die Takt-Signale. Die VHDL-Module, die die im Kapitel 5.4 beschrieben Probleme lösen werden, werden auf dem Baustein „Spartan-3“ simuliert und synthetisiert.

5.2 Ablauf des Entwurfs auf einem FPGA

In diesem Kapitel wird der Ablauf des Entwurfs auf einem FPGA erklärt. Er beschreibt die geforderten Schritte, die zu einem funktionsfähigen Schaltungs-Modell führen. Die nachfolgende Abbildung 44 zeigt den entsprechenden Entwurfsablauf.

Ein FPGA-Entwurf kann mittels einer VHDL-Schaltungsbeschreibung, die als VHDL-Modell oder auch als VHDL-Programm bekannt ist, eingegeben werden. Die einzelnen Module können durch Simulation auf Korrektheit verifiziert werden. Hierbei wird von funktionaler Simulation gesprochen, da keine Informationen über das zeitliche Verhalten der einzelnen Komponenten und Verbindungen vorhanden sind. Es wird nur die Funktionalität des Entwurfs untersucht. Sind die Ergebnisse der Simulation zufriedenstellend, wird aus der VHDL-Datei mittels der Synthese eine Beschreibung des VHDL-Modells auf die Logikebene erzeugt. Diese Beschreibung wird in eine Netzlisten-Datei gelegt, die in einem Format des entsprechenden FPGA-Herstellers oder wiederum eine VHDL-Beschreibung sein kann. Da die Netzliste Informationen, die aus den Herstellerbibliotheken stammen, über die Durchlaufzeiten der einzelnen Komponenten und Verbindungen enthält, kann eine Timing Simulation durchgeführt werden. Hierbei wird von einer Pre-Layout-Simulation gesprochen, da die Verzögerungszeiten der Verbindungsleitungen

nur nach dem Layout bekannt sind. Demzufolge liefert diese Simulation nur einen begrenzten Eindruck über das zeitliche Verhalten der Schaltung. Sind Entwurfsfehler bei der Simulation aufgetreten, so wird im VHDL-Programm korrigiert. Die vorherigen Stufen werden wiederholt, bis ein zufriedenstellendes Ergebnis der Simulation erzielt wird. Mit einem zufriedenstellenden Ergebnis kann man die nächste Stufe fortsetzen, wobei die Komponenten bzw. die Elemente der Schaltung in dem FPGA platziert und verdrahtet werden. Dieser Prozess, der zumeist über automatische Werkzeuge ausgeführt wird, wird als „Place and Route“ bezeichnet.

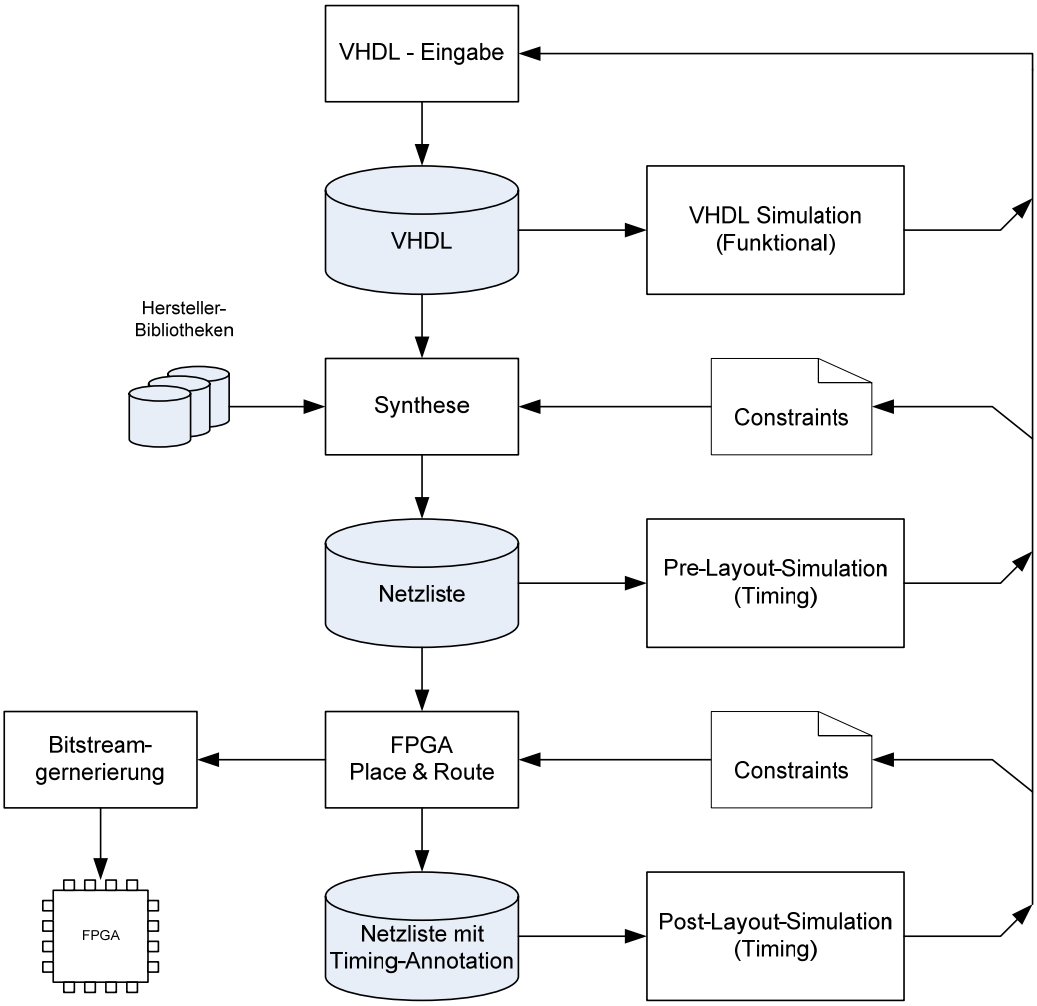

Abbildung 44: Entwurfsablauf auf einem FPGA [nach 26]

Durch die Eingabe von Einschränkungen, die man „Constraints“ bezeichnet, können die Prozesse „Synthese“ und „Place and Route“ gesteuert werden. Ist der Prozess „Place and Route“ vollzogen, können die endgültigen Laufzeiten der Signale berechnet werden. Mit einer erneuten Timing-Simulation, auch bekannt als „Post-Layout-Simulation“, kann das zeitliche Verhalten der Schaltung genauer untersucht werden. Nicht zufriedenstellende Ergebnisse bedingen eine Rückkehr zur ersten Stufe der Simulation, in welcher das VHDL-Programm geändert werden muss. Dieser Prozess wird wiederholt bis ein zufriedenstellendes Ergebnis der „Post-Layout-Simulation“ erreicht ist. Anschließend können die Konfigurationsdaten für den FPGA in Form eines Bitstreams erstellt und auf FPGA programmiert werden [26].

5.3 VHDL Grundlagen

Zum bessern Verständnis der zu implementierenden VHDL-Module wird in diesem Abschnitt eine kurze Einführung in die grundlegenden Konzepte von VHDL gegeben. Hierbei werden anhand von Beispielen die wichtigsten Elemente der Programmier-Sprache VHDL erläutert.

Digitale Schaltungen werden in einzelne Komponente aufgeteilt, um die Komplexität bei ihrer Entwicklung zu verringern. Die einzelnen Komponenten werden später miteinander verbunden, um die gesamte digitale Schaltung zu erstellen. Jede Komponente wird in VDHL über eine Entity „VHDL: entity“ beschrieben. In der Entity werden die Anschlüsse „port“ der Komponente definiert. Entweder können die Anschlüsse als Eingänge, Ausgänge oder bidirektionale Anschlüsse definiert werden. In dem nachfolgenden Code-Listing 1 wird eine Komponente mit dem Namen „dev“ definiert. Hierbei besitzt die Komponente die Eingangs-Anschlüsse „clk“, „res“ und „data_in“ und den Ausgangs-Anschluss „data_out“.

```
entity dev is
    port(
         clk      : in         std_logic;
         res      : in         std_logic;
         data_in  : in         std_logic_vector(3 downto 0);
         data_out : out        std_logic_vector(3 downto 0);
    );
end dev
```

Code-Listing 1: Entity einer Komponente

Die interne Funktion, welche auch als das Verhalten oder die Struktur einer Komponente betrachtet wird, beschreibt man durch eine „Architecture“. Jede Entity besitzt mindestens eine Architektur, die das Verhalten der Entity beschreibt. In Programmiersprachen, wie beispielweise C, wird der Code sequentiell in der Reihenfolge abgearbeitet. Ein wesentliches Modellierungselement in VHDL ist ein „Process“, der innerhalb einer Architektur definiert wird. Alle Prozesse werden parallel bezüglich einer Modellzeit ausgeführt, wobei die Modellzeit den tatsächlichen zeitlichen Verlauf der Signale später in der Hardware nachbildet. Nur der Code innerhalb eines Prozesses wird sequentiell ausgeführt. Die Reihenfolge der Prozesse innerhalb einer Architektur ist beliebig und führt in Simulation und Synthese zum gleichen Ergebnis. Alle Prozesse innerhalb einer Architektur sind sowohl untereinander als auch zu den Ports über Signale „VHDL: signal“ verbunden. Ein Prozess kann über eine Sensitivitätsliste gesteuert werden. Der nachfolgende Code-Listing 2 repräsentiert die Architektur einer Entity; hierbei werden jeweils die Architecture- und Process-Schlüsselwörter mit „VHDL: begin“ eingeleitet und mit „VHDL: end“ beendet. Die Architektur „arch“ definiert in der Code-Listing 2 die Komponente „dev“ und enthält die zwei Prozesse „p1“ und „p2“. Des Weiteren werden die zwei Signale „sig_data_in“ und „sig_data_out“ initialisiert.

```
architecture arch of dev is
    signal   sig_data_in      : std_logic_vector(3 downto 0) := (others => '0');
    signal   sig_tmp          : integer := 0;
begin
    sig_data_in <= data_in;

    p1 : process(clk)
    begin
    end process p1;

    p2 : process(src)
    begin
    end process p2;
end arch;
```

Code-Listing 2: Architecture einer Komponente

Die Zuweisung „sig_data_in <= data_in“ sowie die zwei Prozesse laufen parallel bzw. nebenläufig ab. Das Signal „sig_data_in“ wird mit dem Datentyp „std_logic_vector“ definiert, der aus einem Vektor besteht und den logischen Wert ´0´ oder ´1´ besitzt. Die Breite und der Index der Feldelemente des Vektors werden durch „(3 downto 0)“ definiert. Mit „(other => '0')“ werden alle Felder des Vektors mit dem Wert ´0´ initialisiert.

Der Sensitivitätsliste kommt eine große Bedeutung bei einem Prozess zu, wobei dieser nur dann „ausgeführt“ wird, wenn sich der Wert eines Signals in der Sensitivitätsliste ändert. Durch „Ausführung“ werden alle Anweisungen im Körper des Prozesses zwischen „begin“ und „end“ sequentiell ausgeführt. Solange sich keines der Signale in der Sensitivitätsliste ändert, bleibt der Prozess unterbrochen, wobei keine Anweisungen ausgeführt werden. Wenn keine Sensitivitätsliste vorhanden ist, wird ein Prozess immer ausgeführt und über die „VHDL: wait“-Anweisung gesteuert. Üblicherweise wird ein Prozess über die Sensitivitätsliste gesteuert, in Testbenches hingegen wird er über die „wait“-Anweisung gesteuert. Die Steuerung des Prozesses „p1“ in der Code-Listing ist von dem Signalwechsel an „clk“ abhängig, demzufolge wird der Prozess „p1“ nur dann ausgeführt, wenn der Wert an „clk“ sich ändert. Innerhalb eines Prozesses vor dem „begin“-Schlüsselwort können lokale Variablen „VHDL: variable“ deklariert werden. Die nachfolgende Code-Listing 3 repräsentiert die Implementierung des Prozess „p1“, in dem die Unterschiede zwischen Signal und Variable erklärt werden. Hierbei wird zwischen dem Signal „“sig_tmp“, das in der vorherigen Code-Listing mit dem Wert ´0´ definiert wurde, und der Variablen „var_tmp“ mit dem Wert ´1´ unterschieden.

```
p1 : process(clk)
    variable var_tmp : integer := 1;
begin
    if rising_edge(clk) then
        -- sig_tmp = 0
        sig_tmp <= sig_tmp + 1; -- sig_tmp = 1
        var_tmp := sig_tmp + 1; -- var_tmp = 1 und nicht 2
        data_out <= conv_std_logic(var_tmp, 4) after 20 ns;
    end if;
end process p1;
```

Code-Listing 3: Struktur eines Prozesses

Signal und Variable unterscheiden sich besonders im Zeitpunkt der Gültigkeit der Zuweisung des Wertes. Die Zuweisung des neuen Werts am Signal „sig_tmp“ wird nicht direkt im Prozess-Ablauf berücksichtigt, sondern erst im darauffolgenden. Wohingegen die Wertzuweisung der Variablen „var_tmp“ zum gleichen Zeitpunkt des Prozess-Ablaufes berücksichtigt wird. Um eine Unterscheidung zu ermöglichen; werden Variablen dem „:=“-Operator und Signale dem „<=“-Operator zugewiesen. Die Funktion „con_std_logic“ wandelt den Typ „integer“ zu „stud_logic“ um. Des Weiteren findet die Zuweisung an den Signalen bzw. an der Variablen innerhalb des Prozesses nur dann statt, wenn „rising_edge(clk)“ wahr ist. Die Funktion „rising_edge(clk)“ gibt an, ob der Zustand des Signals „clk“ von ´0´ zu ´1´ gewechselt ist. Eine Verzögerung der Zuweisung an die Signale findet durch die Anweisung „after“ statt, dadurch kann der neue Wert nach bestimmten zeit zugewiesen werden. „data_out <= conv_std_logic(var_tmp, 4) after 20 ns;“ bedeutet, dass der neue Wert nach 20ns an das Signal „data_out“ zugewiesen wird.

In VHDL ist neben der Syntax der Programmiersprache auch auf den Codierstil zu achten, denn um digitale Logikschaltungen wie z.B. (Registers, Latches, RAMs, Counters, Multiplexers, Decoders und ALUs) zu implementieren, muss ein bestimmter Codierstil angewendet werden. Weitere Auskünfte über die Syntax und Codierstil der VHDL-Sprache erhalten Sie im „XST User Gide“ Kapitel 2 und 6.

5.4 VHDL-Modelle

In diesem Abschnitt werden drei Entwürfe dargestellt, die die in dem Kapitel 4.4 diskutierten Probleme lösen werden. Es wird für jeden Entwurf das Ziel, die Komponenten des Entwurfs und deren Funktionalität vorgestellt. Die Modellierung der „Synchronisation zweier Domänen“ wird mit der Abkürzung „RTC_Sync“ dargestellt, die für „Real-Time Crossbar Synchronisation“ steht. Eine erweiterte Version mit Lösung des Präambel-Problems wird mit der Abkürzung „RTC_Sync_PA“ dargestellt, die für „Real-Time Crossbar Synchronisation mit Präambel“ steht. Der Switching-Entwurf, der vier MII-Ports miteinander verbindet, wird mit der Abkürzung „RTC_CORE“ dargestellt und steht für „Real-Time Crossbar CORE“. Zur Erklärung der Funktionalität der Module werden Blockschaltpläne für jeden Entwurf repräsentiert. Die vollständigen Schaltpläne, die durch den Synthese-Prozess erzeugt werden, stehen im Anhang.

5.4.1 RTC_Sync

Dieser Entwurf hat das Ziel, die zu übertragenen Daten zwischen zwei MII-Schnittstellen zu synchronisieren. Das Prinzip besteht darin, dass die ankommenden Daten („RX_DV“, „RX_D[3:0]“ und „RX_EN“), welche in Beziehung zum Takt-Signal „RX_CLK“ übertragen werden, in einem Speicher mit dem Takt-Signal „RX_CLK“ abgelegt werden. Die gespeicherten Daten werden durch den Wert ´000000´ getrennt. Die abgelegten Daten werden durch das Takt-Signal „TX_CLK“ wieder aus dem Speicher gelesen. Das Signal „RX_DV“ ist sowohl für den Start des Speicherungsprozesses als auch für den Start des Leseprozesses verantwortlich. Zum Stoppen des Speicherungsprozesses ist das Signal „RX_DV“ zuständig. Der Leseprozess wird mit dem Einlesen des Trennungs-Werts ´00000´, welcher zwischen zwei Paketen liegt, gestoppt.

Der Entwurf besteht aus einem Speicher und zwei Zählern. Ein Zähler ist zuständig für die Generierung der Schreibadressen „RX_ADDR“ des Speichers. Der zweite Zähler ist zuständig für die Generierung der Leseadressen „TX_ADDR“ des Speichers. Die genannten Komponenten sind anhand der Abbildung 45 ersichtlich.

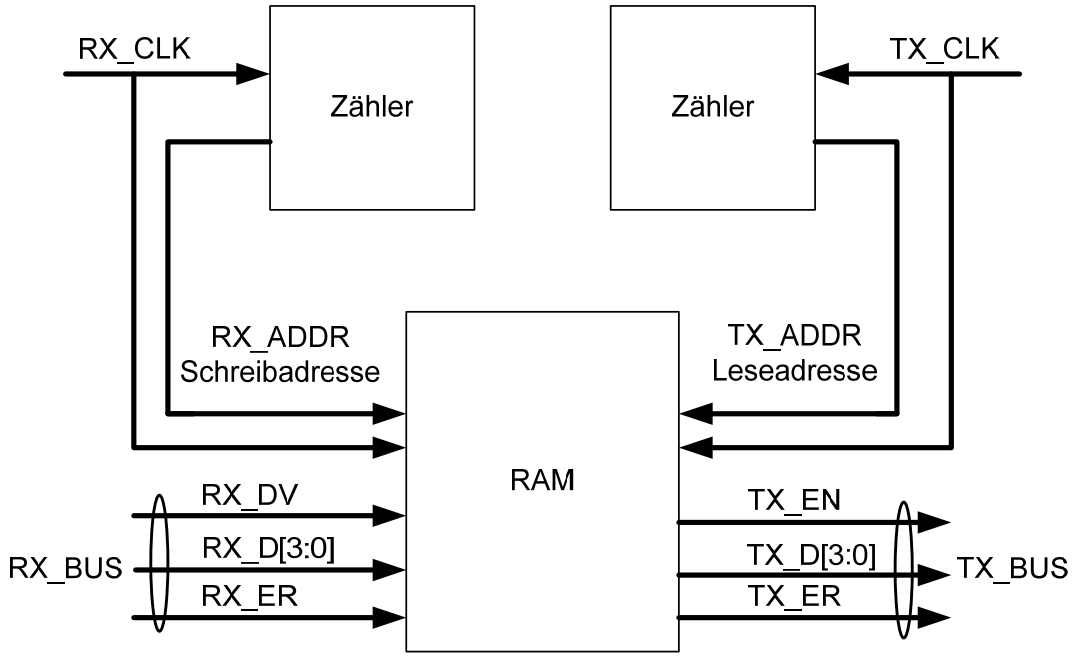

Abbildung 45: Blockschaltplan von „RTC_Sync“

Zur besseren Handhabung der Signale werden die empfangenen Signale „RX_DV“, „RX_D[3:0]“ und „RX_ER“ zum 6-Bit-Breit Signal „RX_BUS“ zusammengefasst. Die zu sendenden Signale „TX_EN“, „TX_D[3:0]“ und „TX_ER“ werden ebenso zum 6-Bit-Breit Signal „TX_BUS“ zusammengefasst. Dies ist anhand der Code-Listing 4 ersichtlich.

```
RX_BUS(5) <= RX_ER;
RX_BUS(4) <= RX_DV;
RX_BUS(3 downto 0) <= RX_DATA;
TX_ER <= TX_BUS(5);
TX_EN <= TX_BUS(4);
TX_DATA <= TX_BUS(3 downto 0);
```

Code-Listing 4: Zusammenfassung von den Signalen

Die Speicherung der empfangenen Daten und die Erzeugung der Schreibadressen findet mit steigender Flanke des Signals „RX_CLK“ statt. Das Lesen der Daten aus dem Speicher und die Erzeugung der Leseadresse finden mit steigender Flanke des Signals „TX_CLK“ statt. Ist das Signal „RX_DV“ auf ´1´ gesetzt, so werden die angekommenen Daten im Speicher abgelegt und der Zähler-Wert auf eins erhöht. Ist das Signal „RX_DV“ auf ´0´ gesetzt, so findet keine Speicherung der Daten statt, der Zähler wird auf Null gesetzt.

Um die oben gewünschte Funktionalität zu realisieren, werden zwei parallel durchführbare Prozesse definiert, einer für die Behandlung von empfangenen Daten, der sogenannte „RX_Process“ und der zweite für die Behandlung von zu sendenden Daten, der sogenannte „TX_Process“. Jeder Prozess kann sich in zwei Zuständen befinden, die durch die zwei Signale „rx_state“ und „tx_state“ bezeichnet werden. Das Signal „rx_state“ kann entweder den Wert „rx_hold“ oder „rx_write“ haben. Das Signal „tx_state“ kann den Wert „tx_hold“, oder den Wert „tx_read“ haben.

Beim RX-Prozess wird im Zustand „rx_hold“ verweilt, bis das Signal „RX_DV“ = ´1´ wird. Erst dann werden die Daten im Speicher abgelegt und zugleich in den Zustand „rx_write“ überführt. Solange sich der RX-Prozess im Zustand „rx_write“ befindet, werden alle ankommenden Daten im Speicher abgelegt, bis das Signal „RX_DV“ seinen Wert von ´1´ auf ´0´ ändert. Dann wird in den Zustand „tx_hold“ überführt, wobei das letzte Nibble mit dem Signal „RX_DV“ = ´0´ ebenso im Speicher abgelegt wird. Dieses Nibble gilt als Zeichen, dass das gespeicherte Paket zu Ende ist. Zum Zurücksetzen des Entwurfs ist das Signal „RESET“ verantwortlich, das asynchron zum Signal „RX_CLK“ ausgeführt werden kann. Hierbei wird der Zähler auf Null gesetzt und in den Zustand „rx_hold“ überführt. Die Implementierung ist durch den nachfolgenden Code-Listing 5 realisiert.

```
RX_Process : process(RX_CLK, RESET)
begin
    if (RESET = '1') then
         RX_ADDR <= RX_ADDR - RX_ADDR;
         rx_state <= rx_hold;
    elsif rising_edge(RX_CLK) then
         case rx_state is
              ---------------------------------------------
              when rx_hold =>
                   if (RX_DV = '1') then
                        mem(conv_integer(RX_ADDR)) <= RX_BUS;
                        RX_ADDR <= RX_ADDR + '1' after 20 ns;
                        rx_state <= rx_write;
                   else
                        rx_state <= rx_hold;
                   end if;
              ---------------------------------------------
              when rx_write =>
                   if (RX_DV = '0') then
                        mem(conv_integer(RX_ADDR)) <= RX_BUS;
                        RX_ADDR <= RX_ADDR - RX_ADDR after 20 ns;
                        rx_state <= rx_hold;
                   else
                        mem(conv_integer(RX_ADDR)) <= RX_BUS;
                        RX_ADDR <= RX_ADDR + '1' after 20 ns;
                        rx_state <= rx_write;
                   end if;
              ---------------------------------------------
         end case;
    end if;
end process RX_Process;
```

Code-Listing 5: RX-Prozess im Modul "RTC_Sync"

Der TX-Prozess befindet sich im „tx_hold“ Zustand, solange das Signal „RX_DV“ = ´0´ ist. Wechselt der Wert des Signals „RX_DV“ auf ´1´, so wird der TX-Prozess in den Zustand „tx_read“ überführt. Nur im Zustand „tx_read“ können die Daten aus dem Speicher gelesen werden. Da die beiden Prozesse (RX & TX) durch das Signal „RX_DV“ gestartet werden, findet eine Unterscheidung in diesem Fall durch die Behandlung des ersten Nibbles statt. Bei Erkennung des Signals „RX_DV“ = ´1´ im RX-Prozess wird das erste Nibble des Pakets im gleichen Takt im Speicher abgelegt, wohingegen im TX-Prozess das Lesen des ersten Nibbles aus dem Speicher im nächst folgendem Takt stattfindet. Dies bedeutet, dass der Leseprozess einen Takt später nach der Speicherung des ersten angekommen Nibbles stattfindet. Die Ursache liegt darin, dass die Daten eine gewisse Zeit benötigen, bis sie den Speicher erreichen. Somit beginnt der Leseprozess einen Takt später. Es wird gewährleistet, dass das erste ankommende Nibble sicher im Speicher liegt. Dieser Fall wird im Simulations-Kapitel 5.5 näher erklärt. Im „tx_read“ Zustand werden die Daten der Variable „tmp“ zugewiesen, damit sie im gleichen Takt ausgewertet werden können. Ist das aus dem Speicher gelesene Datum ein gültiges, was durch den fünften Bit (diese entspricht „TX_DV“) ausgewertet wird, dann wird das Nibble mit dem Gültigkeits-Signal an den Bus gelegt. Der TX-Prozess verweilt im Zustand „tx_read“. Ist das fünfte Bit kein gültiges Bit, so wird das letzte Datum, das aus Nullen besteht, an den Bus gelegt. Der TX-Prozess wird dann in den Zustand „tx_hold“ überführt. Dies wird in dem folgenden Code-Listing 6 ersichtlich.

```
TX_Process : process(TX_CLK, RESET)
   variable tmp : std_logic_vector(5 downto 0) := (others => '0');
begin
   if RESET = '1' then
   tmp := (others => '0');
   TX_ADDR <= TX_ADDR - TX_ADDR;
   elsif rising_edge(TX_CLK) then
        case tx_state is
             ---------------------------------------------
             when tx_hold =>
                  if RX_DV = '1' then
                        tx_state <= tx_read;
                  else
                        tx_state <= tx_hold;
                  end if;
             ---------------------------------------------
             when tx_read =>
                  tmp := mem(conv_integer(TX_ADDR));
                  if tmp(4) = '0' then
                        tx_state <= tx_hold;
                        TX_ADDR <= TX_ADDR - TX_ADDR;
                  else
                        tx_state <= tx_read;
                        TX_ADDR <= TX_ADDR + '1';
                  end if;
             ---------------------------------------------
        end case;
   end if;
   TX_BUS <= tmp;
end process TX_Process;
```

Code-Listing 6: TX-Prozess im Modul "RTC_Sync"

5.4.2 RTC_Sync_PA

Das Ziel dieses Entwurfs ist es, die fehlende Präambel des empfangenen Paketes zu rekonstruieren. Das Modul muss das Paket mit der neu konstruierten Präambel synchron zum Signal „TX_CLK“ weitersenden. Die ggf. unvollständige Präambel des empfangenen Paketes wird abgeschnitten. Bereit während der Frame empfangen wird, wird damit begonnen, eine in einem ROM gespeicherte vollständige Präambel und SFD zu senden. Im Anschluss daran wird der Daten-Frame dem Ausgabestrom angehängt.

Der Entwurf besteht aus zwei parallel durchführbaren Zustandsmaschinen, eine für die Steuerung der empfangenen Daten und eine für die Steuerung der zu sendenden Daten. Das Modul besteht aus einem RAM zur Speicherung und zu Lesen von Daten. Des Weiteren besteht es aus zwei Zählern, die die Adressen für die Speicherung und für das Lesen der Daten des RAMs erzeugen. Zur Rekonstruktion der fehlenden Präambel wird ein ROM eingesetzt, wobei eine komplette Präambel mit SFD programmiert ist. Ein Zähler für den ROM wird zur Generierung der Adresse genutzt, mit der aus dem ROM die Nibbles der Präambel bzw. SFD ausgelesen werden. Auf dem „RX_BUS“, das zum RAM überführt, liegen die Signale „RX_D[3:0]“ und „RX_DV“ und „RX_ER“. Die Daten, die aus dem RAM bzw. aus dem ROM ausgelesen werden, werden in den „TX_BUS“ überführt, demzufolge werden die angekommenen Daten „RX_DV“, „RX_D[3:0]“ und „RX_ER“ als „TX_EN“, „TX_D[3:0]“ und „TX_ER“ ausgelesen. Der gesamte Entwurf wird durch die

Signale „RX_CLK“ und „TX_CLK“ gesteuert. Die Komponenten des Entwurfs sind in der Abbildung 46 ersichtlich.

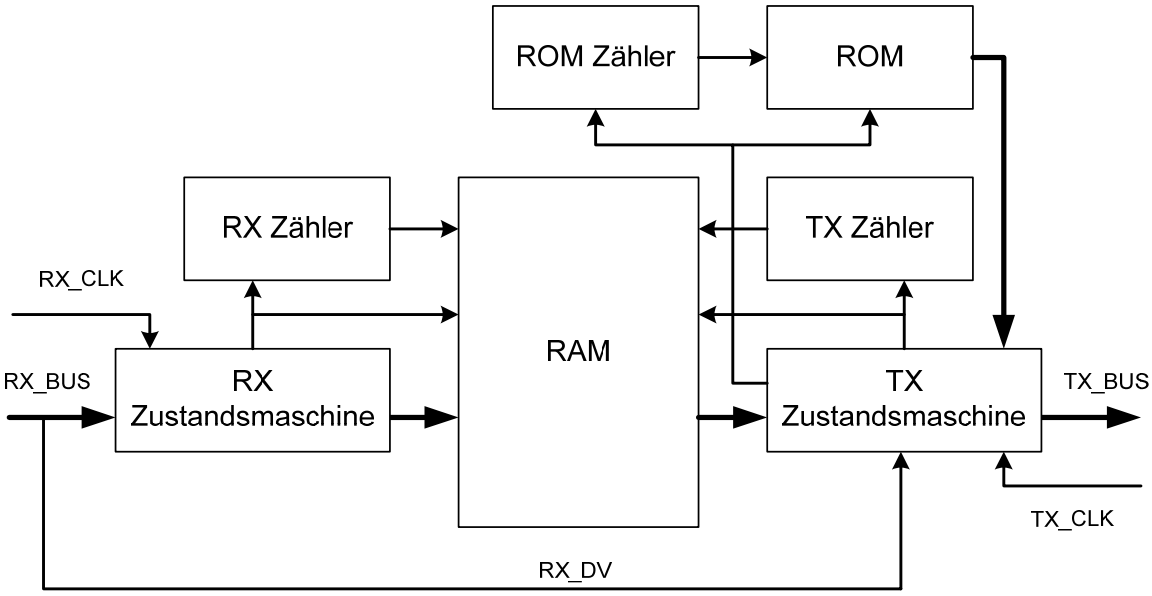

Abbildung 46: Blockschaltplan von „RTC_Sync_PA“

Die RX-Zustandsmaschine besitzt drei Zustände (Abbildung 47), die das Verhalten der Speicherung der empfangenen Daten steuern. Solange das Signal „RX_DV“ auf ´0´ gesetzt ist, befindet sich die RX-Zustandmaschine im Zustand „hold_rx“. Ist das Signal „RX_DV“ auf ´1´ gesetzt, wird in den Zustand „search_sfd“ überführt, in welchem alle gelesenen Daten verworfen werden. Sobald in diesem Zustand das Nibble ´1101´, welches dem „Start Frame Delimiter“ entspricht, gelesen wird, findet eine Überführung der Zustandmaschine in den Zustand „write_ram“ statt. In diesem Zustand werden alle ankommenden Daten im Speicher abgelegt, bis das Signal „RX_DV“ seinen Zustand von ´1´ zu ´0´ ändert.

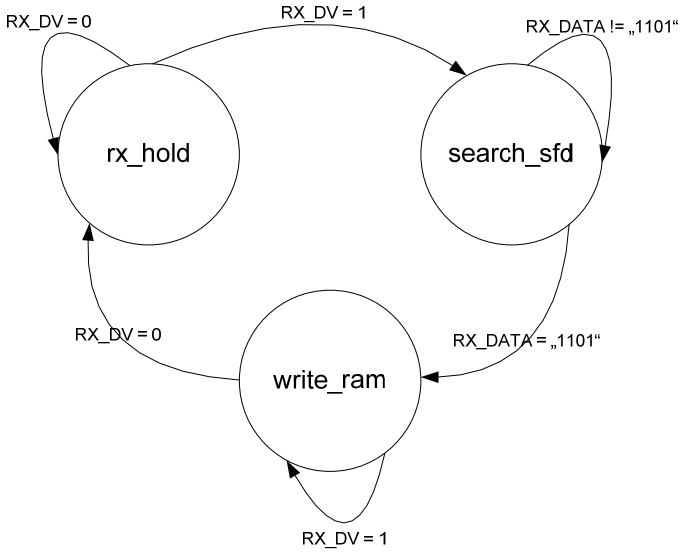

Abbildung 47: RX-Zustandsmaschine

Der nachfolgende Code-Listing 7 repräsentiert die Implementierung der RX-Zustandsmaschine, die durch Case-Anweisung in VHDL moduliert wird.

```
case rx_state is
    ------------------------------------------------
    when rx_hold =>
         if RX_DV = '1' then
              rx_state <= search_sfd;
         else
              rx_state <= rx_hold;
         end if;
    ------------------------------------------------
    when search_sfd =>
         if RX_DATA ="1101" then
              rx_state <= write_ram;
         else
              rx_state <= search_sfd;
         end if;
    ------------------------------------------------
    when write_ram =>
         RAM(conv_integer(ram_wa)) <= RX_BUS;
         if RX_DV = '0' then
              RAM_WA <= RAM_WA - RAM_WA after 20 ns;
              rx_state <= rx_hold;
         else
              RAM_WA <= RAM_WA + '1' after 20 ns;
              rx_state <= write_ram;
         end if;
    ------------------------------------------------
    when others =>
         rx_state <= rx_hold;
    ------------------------------------------------
end case;
```

Code-Listing 7: RX-Zustandsmaschine

Die TX-Zustandsmaschine, die in Abbildung 48 ersichtlich ist, kann in drei Zustände überführt werden. Die Zustandsmaschine, die das Verhalten des Lesens steuert, bleibt im Zustand „hold_tx“, solange das Signal RX_DV auf ´0´ gesetzt ist. Sobald das Signal auf ´1´ gesetzt wird, wird

die Zustandsmaschine in den Zustand „read_rom“ überführt. Im Zustand „read_rom“ wird eine neue vollständige Präambel aus dem ROM gelesen und durch das Signal „TX_BUS“ übertragen. Der nachfolgende Code-Listing 8 repräsentiert die Implementierung eines ROMs, das mit der Präambel und der SFD programmiert ist. Im ROM sind die Signale „RX_DV“ mit ´1´ und „RX_ER“ mit ´0´ programmiert, dies bedeutet, das die Präambel und das SFD keine Fehler beinhalten und gültig sind.

```
type rom_type is array (0 to 15) of std_logic_vector(5 downto 0);
constant ROM : rom_type :=(
                                --[TX_ER, TX_EN, TX[3:0]]-- Belegung des ROMs
                                0 => "010101",
                                1 => "010101",
                                2 => "010101",
                                3 => "010101",
                                4 => "010101",
                                5 => "010101",
                                6 => "010101",
                                7 => "010101",
                                8 => "010101",
                                9 => "010101",
                                10 => "010101",
                                11 => "010101",
                                12 => "010101",
                                13 => "010101",
                                14 => "010101",
                                15 => "011101"); -- zweite Nibble des SFDs
```

Code-Listing 8: ROM-Programmierung

Ist im ROM die letzte Adresse ausgelesen, welche dem Speicherplatz des zweiten Nibbles des SFD entspricht, so wird die Zustandsmaschine in den Zustand „read_ram“ überführt. Im „read_ram“-Zustand werden die Daten aus dem RAM gelesen und durch den „TX_BUS“ übertragen. Die Zustandsmaschine verlässt den Zustand „read_ram“ nur dann, wenn das vierte Bit des „TX_BUS“ seinen Zustand von ´1´ zu ´0´ ändert. Das bedeutet, dass das Signal „TX_EN“ = ´0´ ist. Damit wird signalisiert, dass das Ende des übertragenden bzw. gelesenen Paketes erreicht wurde.

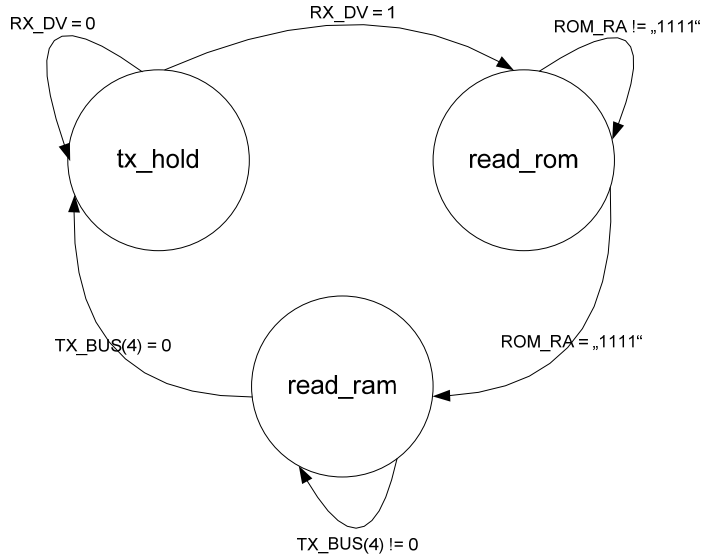

Abbildung 48: TX-Zustandsmaschine

Die nachfolgende Code-Listing 9 repräsentiert die TX-Zustandsmaschine.

```
case tx_state is
    ------------------------------------------------
    when tx_hold =>
         if RX_DV = '1' then
         tx_state <= read_rom;
    else
         tx_state <= tx_hold;
    end if;
    ------------------------------------------------
    when read_rom =>
         TX_BUS <= ROM(conv_integer(ROM_RA));
         if ROM_RA = "1111" then
              ROM_RA <= ROM_RA - ROM_RA;
              tx_state <= read_ram;
         else
              ROM_RA <= ROM_RA + '1';
              tx_state <= read_rom;
         end if;
    ------------------------------------------------
    when read_ram =>
         if (TX_BUS(4) = '0') then
              RAM_RA <= RAM_RA - RAM_RA;
              tx_state <= tx_hold;
         else
              TX_BUS <= RAM(conv_integer(ram_ra));
              RAM_RA <= RAM_RA + '1';
              tx_state <= read_ram;
         end if;
    ------------------------------------------------
    when others =>
         tx_state <= tx_hold;
    ------------------------------------------------
end case;
```

Code-Listing 9: TX-Zustandsmaschine

5.4.3 RTC_CORE

Das Ziel dieses Entwurfs ist es, vier MII-Schnittstellen miteinander zu verbinden. Die Verbindung zwischen den vier MII-Schnittstellen soll nach einer definierten Zeit geändert werden. Wann und wie welche Schnittstelle mit der anderen verbunden werden soll, wird durch einen vorgegebenen Scheduler festgelegt. Das Modul realisiert keine Umschaltung zum MAC-Ebene.

Dieser Entwurf besteht aus einem Takt-Generator, der ein sogenanntes „Sync_CLK“ Takt-Signal generiert. Das Signal „Sync_CLK“ steuert die Durchführung der Umschaltung zwischen den Schnittstellen und gibt zugleich den Zeitpunkt an, wann die Pakete von den Teilnehmern gesendet werden sollen. Das Signal „Sync_CLK“ entspricht dem Polling-Signal für die Teilnehmer. Der Entwurf besteht auch aus einer Switching-Fabric, die die ankommenden Pakete an die entsprechende MII-Schnittstelle transferiert. Um die Probleme der Synchronisation der fehlenden Präambel zu beheben, wird jede Transmit-Schnittstelle auf jedem Port des „RTC_CORE“ Modul mit einem „RTC_Sync_PA“-Modul versehen. Der Scheduler-Halter wird durch einen RAM-Speicher implementiert. In der folgenden Abbildung 49 können die vorgestellten Komponenten betrachtet werden.

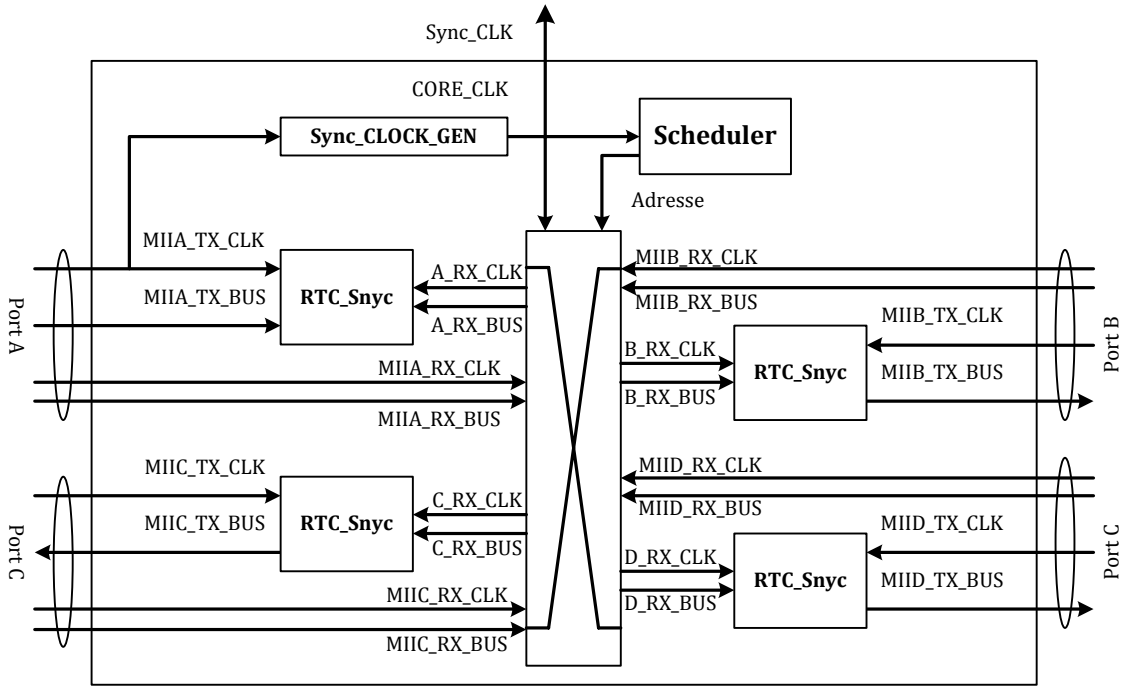

Abbildung 49: Blockschaltplan von „RTC_CORE“

Wie in der Abbildung 49 ersichtlich ist, wird jede Sync-Einheit an der Transmit-Schnittstelle über die Signale „MII#_TX_CLK“ und „MII#_TX_BUS“ mit jedem Port verbunden. Das Symbol „#“ steht für die entsprechenden Ports (A, B, C, und D). Jede Sync-Einheit an der Receive-Schnittstelle ist über die Signale „*_RX_CLK“ und „*_RX_BUS“ mit der Switching-Fabric

verbunden. Das Symbol „*“ steht für die entsprechenden umgeschalteten Ports (A, B, C, und D). Ein Bündel-Signal „MII#_TX_BUS“ besteht aus den Signalen „TX_EN“, „TX_D[3:0]“ und „TX_ER“ für die entsprechende MII-Schnittstelle. Die Bündel-Signale „A_RX_BUS“, „B_RX_BUS“, „C_RX_BUS“ und „D_RX_BUS“ bestehen hingegen aus den Signalen „RX_DV“, „RX_D[3:0]“ und „RX_ER“.

Der Takt-Generator erzeugt einen Zeitschlitz, der aus dem Takt-Signal „TX_CLK“ des Ports „A“ gebildet wird. Der Wert des Zeitschlitzes entspricht der Zählung der Periodendauer des Signals „TX_CLK“. Die nachfolgende Code-Listing 10 zeigt einen Prozess, in dem der Synchronisationstakt „Sync_CLK“ aus einem eingegebenen Takt-Signal „CLKIN“ erzeugt wird. Die Variable „TIMESLOT“ entspricht der Anzahl der gewünschten Perioden, die den Wert des Zeitschlitzes definiert.

```
process(CLKIN)
    variable count : integer := 0;
    constant top_clk : integer := (TIMESLOT*90)/100;
begin
    if rising_edge(CLKIN) then
         if(count = top_clk) then
              int_CLK <= not int_CLK;
              count := count + 1;
         elsif (count = TIMESLOT) then
              int_CLK <= not int_CLK;
              count := 0;
         else
              count := count + 1;
         end if;
    end if;
end process;
```

Code-Listing 10: Synchronisationstakt „Sync_CLK“

Die Switching-Fabric bedient sich der Informationen des Schedulers, und regelt dann, wie die einkommenden Bündel-Signale „MIIA_RX“, „MIIB_RX“, „MIIC_RX“ und „MIID_RX“ umgeschaltet werden. Jeder Eintrag im Scheduler definiert den Zustand eines Ports. Die nachfolgende Tabelle 9 repräsentiert die mögliche Kodierung der Umschaltung zwischen den Ports.

	MII_A	MII_B	MII_C	MII_D
MII_A	00	00	00	00
MII_B	01	01	01	01
MII_C	10	10	10	10
MII_D	11	11	11	11

Tabelle 9: Kodierung der Umschaltung

Ein Eintrag im Scheduler umfasst 8-Bit, wobei der Zustand eines jeden Ports durch 2-Bit definiert ist. Die letzten 2-Bit (an der 7 und 8 Stelle) beschreiben den Zustand des Ports A, die nächsten 2-Bit beschreiben den Zustand des Ports B usw.. Der Eintrag [01101101] beispielsweise, bedeutet, dass die Daten im Port A von B, im Port B von C, im Port C von D, und

im Port D von B kommen. Die Übertragung der Daten an den entsprechenden Ports stellt hierbei eine Vollduplex-Übertragung dar.

Die nachfolgende Code-Listing 11 zeigt einen Prozess, in dem der Zustand des Ports „A“ definiert wird. Hierbei wird definiert, welches von den Bündel-Signalen „MIIB_RX_BUS“, „MIIC_RX_BUS“ und „MIID_RX_BUS“ zur bestimmten Zeit an das Bündel „A_RX_BUS“ geschaltet wird.

```
process(schedule, MIIB_RX_BUS, MIIC_RX_BUS, MIID_RX_BUS, MIIB_RX_CLK,
MIIC_RX_CLK, MIID_RX_CLK)
begin
    case schedule(7 downto 6) is
         when "00" =>
              A_RX_CLK <= '0';
              A_RX_BUS <= (others => '0');
              --MIIA_state <= MIIA2MIIA;
         when "01" =>
              A_RX_CLK <= MIIB_RX_CLK;
              A_RX_BUS <= MIIB_RX_BUS;
              --MIIA_state <= MIIB2MIIA;
         when "10" =>
              A_RX_CLK <= MIIC_RX_CLK;
              A_RX_BUS <= MIIC_RX_BUS;
              --MIIA_state <= MIIC2MIIA;
         when "11" =>
              A_RX_CLK <= MIID_RX_CLK;
              A_RX_BUS <= MIID_RX_BUS;
              --MIIA_state <= MIID2MIIA;
         when others =>
              A_RX_CLK <= '0';
              A_RX_BUS <= (others => '0');
              --MIIA_state <= MIIA2MIIA;
    end case;
end process;
```

Code-Listing 11: Umschaltung des Ports A

Das Vorgehen bei der Zustandsdefinition eines Ports steht in Verbindung mit der VHDL-Semantik, da das Bündel-Signal „A_RX_BUS“ nicht von mehren Prozessen zugewiesen werden kann.

Das Modul kann modifiziert werden, um eine Umschaltung zur MAC-Ebene zu realisieren. Lösungsansätze dafür werden im Kapitel 5.9 diskutiert.

5.5 Simulation der Module

In diesem Abschnitt wird das Verhalten der Schaltung bzw. des VHDL-Entwurfs sowie die gewünschten Funktionalitäten simuliert und verifiziert. Jeder VHDL-Entwurf der vorherigen Abschnitte wird bestimmten Szenarien unterzogen, die einzeln diskutiert werden. Hierbei wird anhand der Simulationen die funktionale Korrektheit der entworfenen Module bewiesen. Zunächst wird eine kurze Erklärung von der Testbench, die die Basis der Simulation darstellt, vorgestellt. Da die Waveform der funktionalen Simulation der Post-Layout Simulation entspricht, stellen die Simulationen in diesem Abschnitt die endgültige Simulation dar, nämlich die Post-Layout Simulation.

5.5.1 Testbench

Um die Simulation durchführen zu können, muss eine Testbench erstellt werden. Eine Testbench erzeugt die nötigen Signalwechsel an den Eingängen der zu verifizierenden Schaltung. Diese Signale werden „Stimuli“ genannt. Die zu verifizierende Schaltung wird als „DUV“ (engl.: Device-Under-Verification) bezeichnet. Der zeitliche Verlauf der Ausgangssignale wird in einem Betrachtungsprogramm, dem sogenannten „Waveform Viewer“, dargestellt. Die Testbenches dienen nur der Simulation und müssen später nicht in die Hardware umgesetzt werden. Die Abbildung 50 zeigt den Simulationsprozess [25].

Abbildung 50: Ablauf der Simulation [nach 25]

Für die Modellierung einer Testbench wird der Umfang der VHDL-Sprache eingesetzt. Eine Testbench kann sowohl für funktionale Simulation als auch für zeitliche Simulation eingesetzt werden. Die im Rahmen dieser Arbeit erstellten Testbenches werden sowohl für die funktionale als auch für die zeitliche Simulation eingesetzt. Die geforderten Eingangssignale, die zur Simulation aller VHDL-Entwürfe und von der Testbench erzeugt werden sollen, sind „RX_CLK“, „RX_DV“, „RX_D[3:0]“, „RX_ER“ und „TX_CLK“.

Zur Generierung der Taktsignale „RX_CLK“ und „TX_CLK“ wird jeweils ein Prozess implementiert, in dem die steigende und fallende Takt-Flanke nach Eingabe der „Periodendauer“ und „Duty Cycle“ erzeugt werden. Die folgende Code-Listing 12 implementiert das Takt-Signal „RX_CLK“.

```
CLK_RX : PROCESS
BEGIN
    WAIT for RX_OFFSET;
    LOOP
         RX_CLK <= '0';
         WAIT FOR (PERIOD - (PERIOD * RX_DUTY_CYCLE));
         RX_CLK <= '1';
         WAIT FOR (PERIOD * RX_DUTY_CYCLE);
    END LOOP;
END PROCESS CLK_RX;
```

Code-Listing 12: Generierung der Taktsignale „RX_CLK“ und „TX_CLK“

Die Signale „RX_DV“ und „RX_DATA“ stellen ein einkommendes Ethernet-Paket dar, welches aus dem Präambel-Teil und dem Daten-Teil besteht. Die nachfolgende Code-Listing 13 realisiert ein Ethernet-Paket. Die Daten werden mit einer fallenden Flanke des Signals „RX_CLK“ gesendet.

```
FRAME : process
   variable tmp : std_logic_vector(3 downto 0) := (others => '0');
begin
   wait for 2*PERIOD;
   wait until RX_CLK = '0';
   loop
        -------------------------------------------------------
        -- Preamble nibbles
        for i in 0 to 9 loop
             wait for PERIOD;
             RX_DV <= '1';
             RX_DATA   <= "0101";
        end loop;
        -------------------------------------------------------
        -- SFD nibble
        wait for PERIOD;
        RX_DV <= '1';
        RX_DATA <= "1101";
        -------------------------------------------------------
        -- Daten nibbles
        for i in 0 to 9 loop
             wait for PERIOD;
             RX_DV <= '1';
             RX_DATA <= tmp;
        end loop;
        tmp := tmp + 1;
        -------------------------------------------------------
        -- EFD nibble
        wait for PERIOD;
        RX_DV <= '1';
        RX_DATA <= "1111";
        -------------------------------------------------------
        -- IFG nibbles
        for i in 0 to 9 loop
             wait for PERIOD;
             RX_DV <= '0';
             RX_DATA <= (others => '0');
        end loop;
        -------------------------------------------------------
   end loop;
end process FRAME;
```

Code-Listing 13: Generierung eines Frames

Zur besseren Übersicht der Simulation und einfacheren Betrachtung des zeitlichen Verlaufs der erzeugten Signale wurde die Anzahl der Nibbles der Präambel und des Datenteils jeweils auf 10 gesetzt (bei①). Der Wert des Datenteils repräsentiert die Nummer des gesendeten Paketes (bei②). Das EFD (End Frame Dilimeter)-Nibble repräsentiert das letzte Nibble (mit dem Wert „0xF“), das in einem Paket gesendet wird (bei③), es dient ausschließlich der Übersichtlichkeit der Simulation und hat keine weitere Funktionalität. Die IFG-Nibbles „InterFrame Gap“ stellen den zeitlichen Abstand zwischen zwei gesendeten Paketen dar. Auch hier wurde es aufgrund der besseren Übersichtlichkeit der Simulation auf 10 gesetzt. Ein aus der Testbench erzeugtes Ethernet-Paket ist in der folgenden Abbildung 51 ersichtlich.

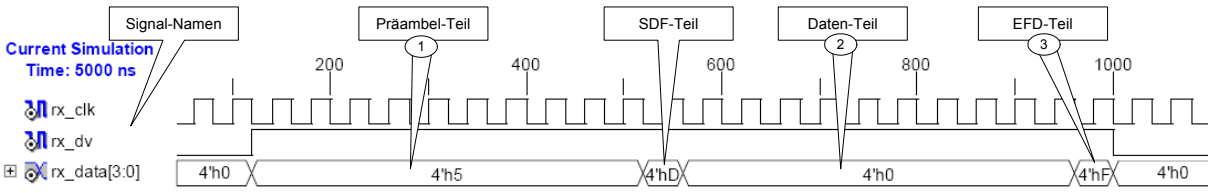

Abbildung 51: Waveform von einem Ethernet-Paket

Die nachfolgende Abbildung 52 zeigt mehrere Ethernet-Pakete, die durch IFG getrennt sind. Das Paket Nr. 1 steht im Daten-Teil und ist in Hex dargestellt.

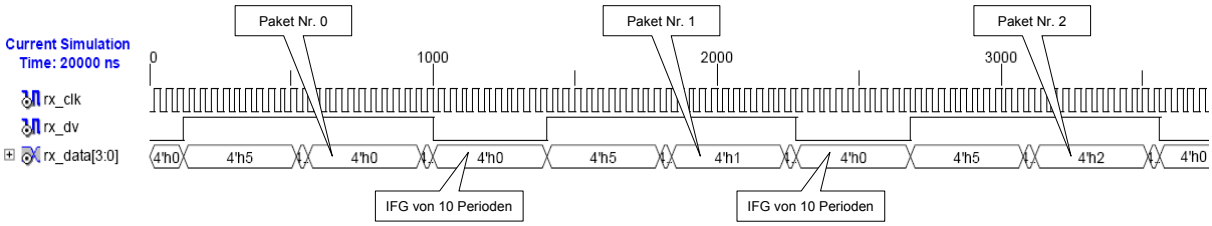

Abbildung 52: Waveform von mehreren Paketen

Das Signal „RX_ER“ wird in Rahmen der Simulation nicht in Waveform dargestellt. Da die Wertänderung des Signals „RX_ER“ keinen Einfluss auf den Übertragungsmechanismus in der MII-Ebene hat, wird es immer den Wert ´0´ haben und wird ohne Darstellung in der Waveform in den Speicher gelegt.

5.5.2 RTC_Sync

In diesem Abschnitt wird gezeigt, wie die Synchronisation zweier Takt-Signale anhand der Waveform der Simulation realisiert wird. Als nächstes wird ein einkommendes Paket durch die nachfolgende Abbildung 53 behandelt.

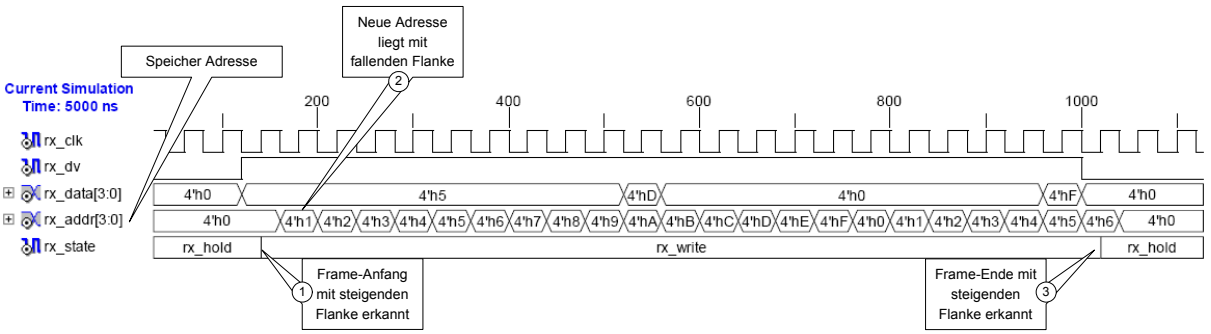

Abbildung 53: Speicherung eines Paketes

In der Abbildung 53 ist zu sehen, wie das einkommende Paket mit einer steigenden Flanke erkannt wird (bei①), dabei werden im gleichen Takt die ankommenden Daten in den Speicher gelegt. Eine neue Adresse, mit der die Nibbles gespeichert werden, wird mit der fallenden Takt-Flanke nach der Erkennung des Pakets inkrementiert und an den Leitungen zur Verfügung

gestellt (bei②). Ist das Paket zu Ende, wird dies nur mit steigender Flanke erkannt und der Speicherungsprozess wird angehalten (bei③)

Zunächst wird ein Speicherungsprozess anhand der Abbildung 54 betrachtet. In der Waveform wird der Wert ´0xD´, welches den SFD ´1101´ repräsentiert, als Beispiel eines Speicherungsprozesses betrachtet. Hierbei wird das Signal „RX_DV = 1“ (bei①) und die Bündel-Signale „RX_DATA = 0xD“ (bei②) im Speicher mit der Zieladresse ´0xA´ (bei③) als Wert ´0x1D´ (bei④) gelegt. Beim Start der Simulation ist der Speicher mit dem Wert ´0xUU´ belegt, was für „uninitialized“ steht. Der Speicher kann mit dem Wert ´0x00´ initialisiert werden, jedoch wird während der Simulation für die Übersichtlichkeit der Speicher uninitialisiert gelassen. Der Wert der Signale „RX_DV“ und das Nibble „RX_DATA“ wird mit der entsprechenden Adresse „RX_ADDR“ im Speicher bei steigender Flanke des Signals „RX_CLK“ gelegt (bei④).

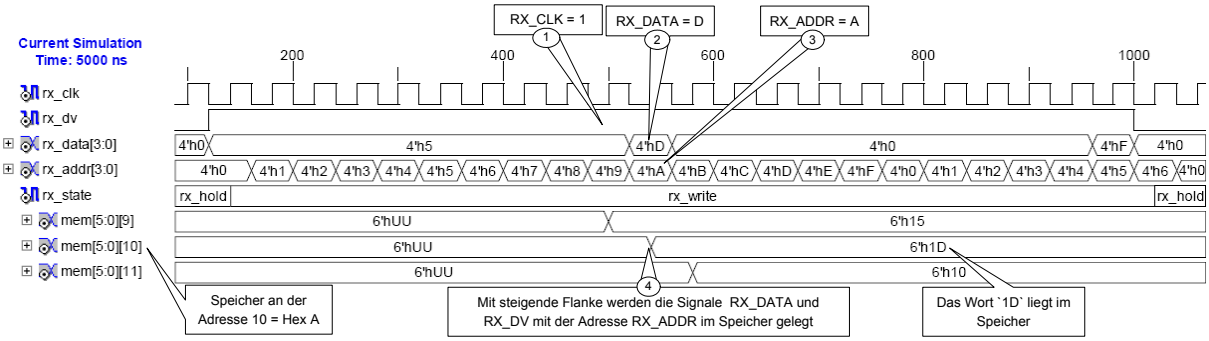

Abbildung 54: Speicherprozess eines Paketes

Die Erkennung das Ende eines einkommenden Pakets wird anhand der nachfolgenden Abbildung 55 diskutiert. Bei einer steigenden Flanke wird erkannt, dass das Signal „RX_DV“ sein Zustand von ´1´ zu ´0´ ändert (bei①). So wird hierbei der Wert ´000000´ mit der Speicheradresse ´0x6´ [bei②] in den Speicher gelegt (bei③). Der Wert ´000000´ deutet, das Ende des im RAM gespeicherten Pakets. Zugleich wird die Speicheradresse “RX_ADDR“ auf ´0x0´ gesetzt (bei④). Des Weiteren wird das Steuer-Signal „rx_state“ auf „rx_hold“ gesetzt und somit findet bei den nachfolgenden Takten keine weitere Speicherung der Daten statt (bei⑤).

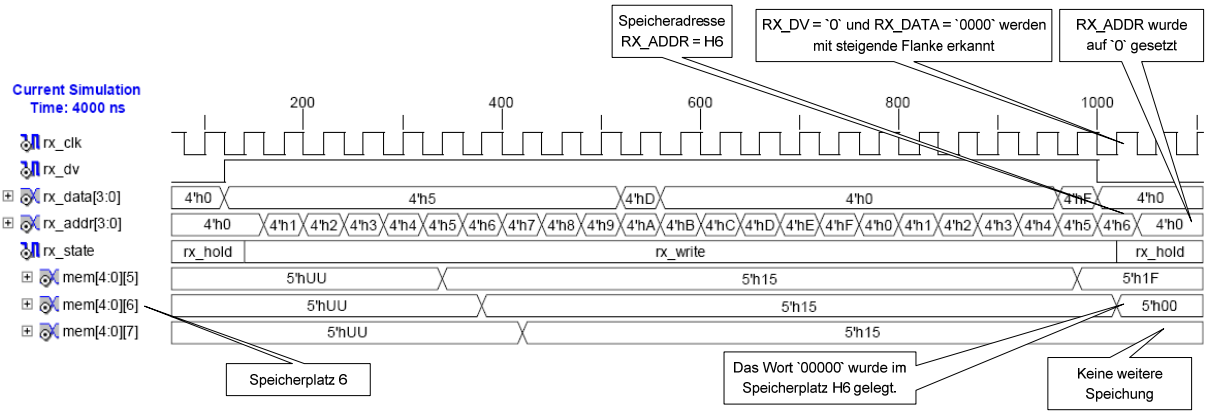

Abbildung 55: Erkennung das Ende eines einkommenden Pakets

Die Abbildung 56 repräsentiert eine Waveform, die den Leseprozess eines gespeicherten Pakets darstellt. Das Lesen eines Pakets wird in Beziehung zum Takt-Signal „TX_CLK“ durchgeführt, und wird durch das Signal „RX_DV“ gestartet. Die Erkennung des Zustandes vom Signal „RX_DV“ findet beim Leseprozess in der steigenden Flanke des Signals „TX_CLK“ statt (bei①). Ändert das Signal „RX_DV“ seinen Zustand von ´0´ zu ´1´, so signalisiert dies, dass der Anfang eines angekommenen Pakets erkannt wurde. Hierbei bekommt das Signal „tx_state“ den Wert „tx_read“ (bei②). Da der Schreib- und Lese-Prozess vom Signal „RX_DV“ gesteuert wird, wird in den nächsten Takt die Leseadresse „TX_ADDR“ (bei③) inkrementiert und ein gespeichertes Paket (bei④) gelesen. Der Leseprozess wird einen Takt verzögert, um sicher zu stellen, dass die erste Nibble des angekommenen Pakets im Speicher liegt. Ist der Wert ´000000´ aus dem Speicher gelesen, so bedeutet dies, dass das Ende des gespeicherten Pakets erreicht ist (bei⑤). In diesem Zustand wird die Leseadresse auf ´0x0´ gesetzt (bei⑥) und das Steuer-Signal „tx_state“ bekommt den Wert „tx_hold“ (bei⑦).

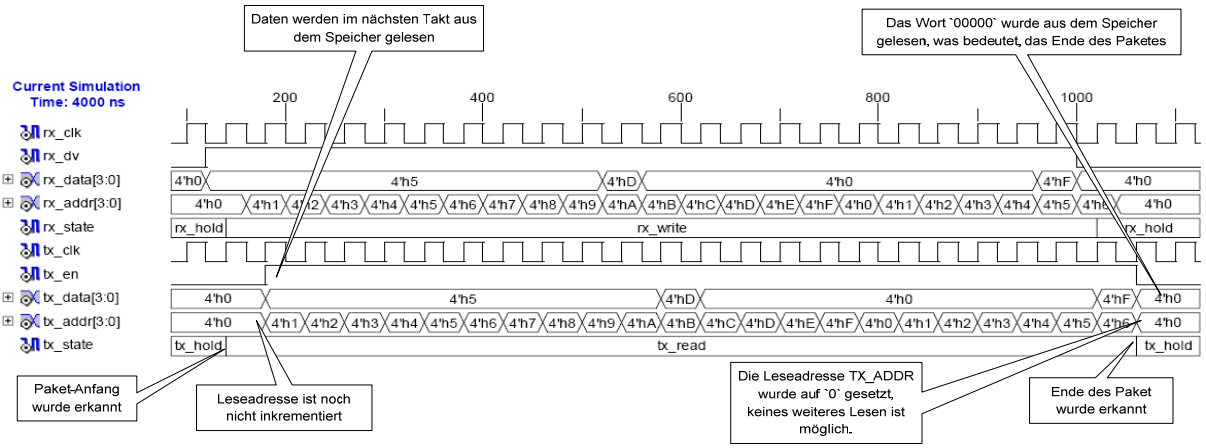

Abbildung 56: Leseprozess eines Pakets

In der Abbildung 57 wird eine fehlerhafte Übertragung gezeigt. Dabei startet der Leseprozess im gleichen Takt, in dem der Paket-Anfang erkannt wurde. Dadurch werden ungültige Daten aus dem Speicher gelesen (bei④). Bei Erkennung des Paket-Anfangs (bei①) wird im gleichen Takt die Leseadresse „TX_ADDR“ inkrementiert (bei②). Dies führt dazu, dass der Wert ´0xUU´ (bei③)

an der Speicheradresse ´0x0´ gelesen wird. Dies hat den Grund, dass das erste Nibble des Paketes noch nicht im Speicher liegt. Daher müsste der Leseprozess einen Takt verzögert werden, um sicher zu stellen, dass das erste Nibble schon im Speicher liegt. In dem in der Abbildung 57 gezeigte Fall können gültige Daten (bei⑤) erst bei der Leseadresse „TX_ADDR = 0“ gelesen werden, solange sie nicht überschrieben werden.

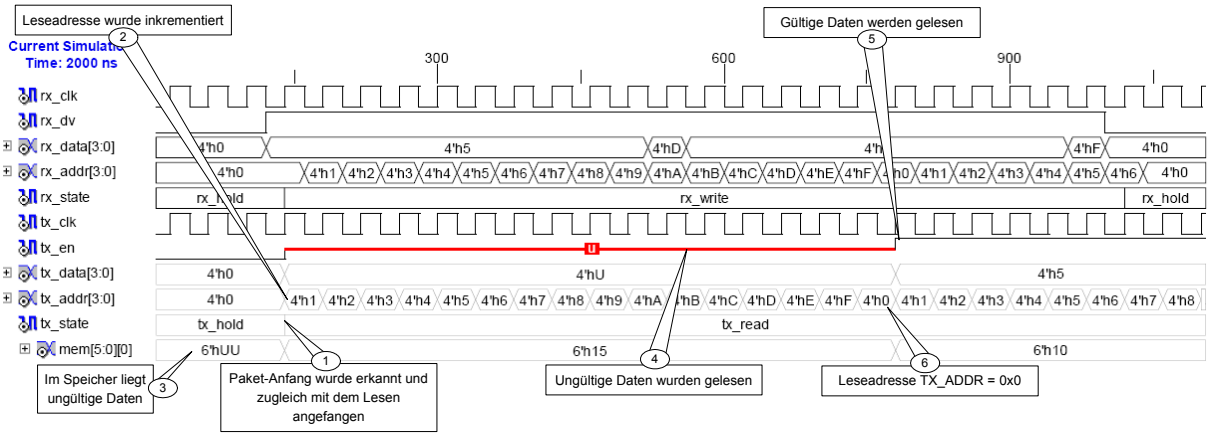

Abbildung 57: Leseprozess ohne zeitliche Verzögerung

Bei hintereinander gelesenen Paketen wird ein zerstörter Datenfluss entstehen, wenn es keine Verzögerung beim Start des Leseprozesses gibt. Der Grund dafür ist, dass eine Überschreibung der gültigen Daten im Speicher stattfindet. Dieser Fall ist durch die Abbildung 58 ersichtlich. Sind ungültige Daten aus dem Speicher gelesen (bei①), so findet nach zwei gelesenen Paketen eine Überschreibung der gültigen Daten im Speicher durch den Schreibprozess statt, weil der Abstand zwischen den Paketen kleiner (bei②) wird. Dies führt für zu einem zerstörten Datenfluss (bei②). Dieses Problem wird durch eine Verzögerung einer Takt-Periode beim Leseprozess gelöst.

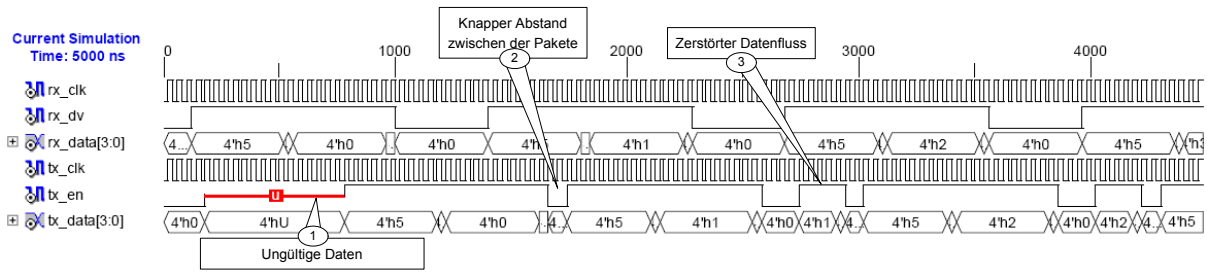

Abbildung 58: Zerstörter Datenfluss beim Leseprozess

Die nachfolgende Abbildung 59 der Simulation repräsentiert eine fehlerfreie Übertragung der einkommenden Pakete. Dabei wird der Abstand zwischen zwei Paketen eingehalten. Ebenfalls wird der Leseprozess mit dem nächsten Takt gestartet. In dieser Simulation haben die beiden Takt-Signale „RX_CLK“ und „TX_CLK“ keine Phasenverschiebung (bei①). Anhand der Abbildung

59 ist zu sehen, wie das ankommende Paket Nr. 2 (bei②) nach einer Verzögerung von einer Takt-Periode fehlerfrei gelesen wurde (bei③).

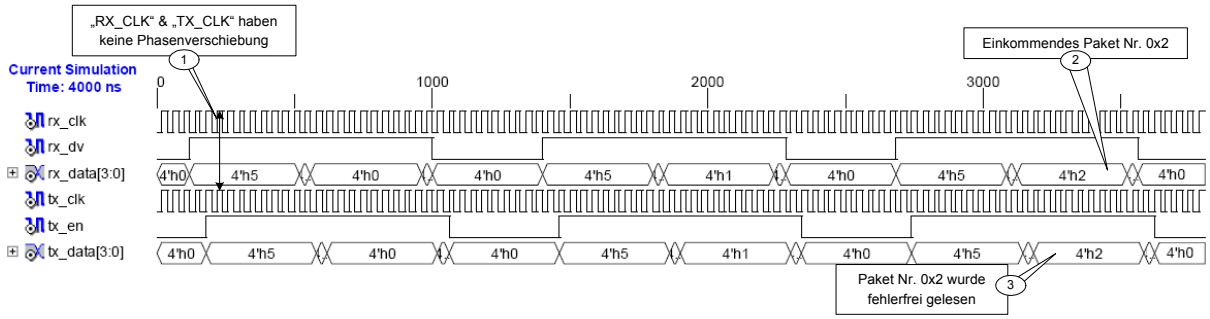

Abbildung 59: Fehlerfreie Daten-Übertragung

Ein Übertragungs-Szenario, in dem eine Phasenverschiebung für die beiden Takt-Signale „RX_CLK“ und „TX_CLK“ bei 20ns liegt (bei①), ist durch die Abbildung 60 dargestellt. Das Signal „RX_DV“ wurde mit der steigende Flanke des Takt-Signals „TX_CLK“ erkannt (bei②). Daten wurden fehlerfrei aus dem Speicher mit einer Verzögerung von einer Takt-Periode gelesen (bei③).

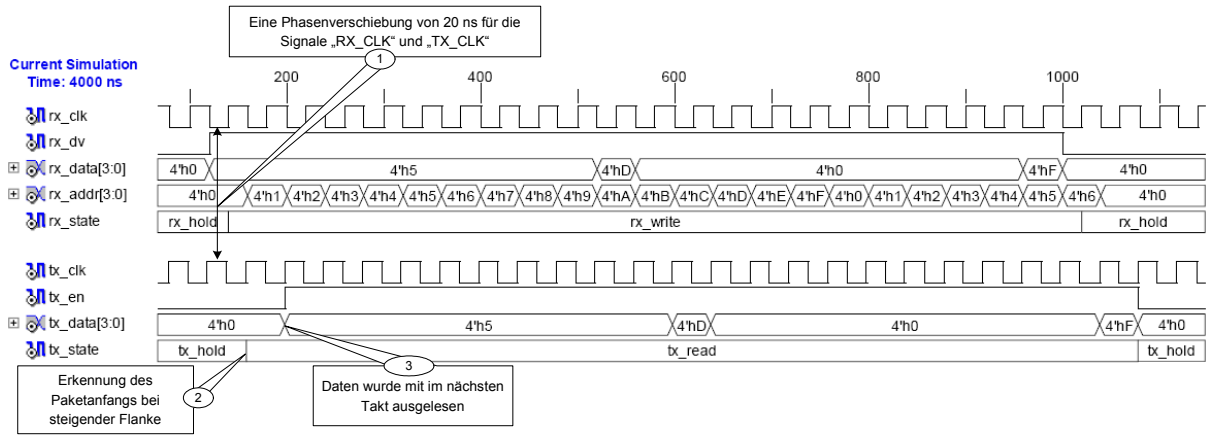

Abbildung 60: Eine Übertragung mit Phasenverschiebung von 20ns

Anhand der Waveform in der Abbildung 61 ist zu erkennen, dass durch den Einsatz des „RTC_Sync“ Modul das Problem der Phasenverschiebung der Takt-Signale „RX_CLK“ und „TX_CLK“ keinen Einfluss mehr auf die Übertragung der Pakete hat. Für mehrere Übertragungen der Pakete, und zwar nach 15µs, werden die empfangenen Pakete durch das Modul fehlerfrei weitergesendet. Dies ist durch die Abbildung 61 ersichtlich.

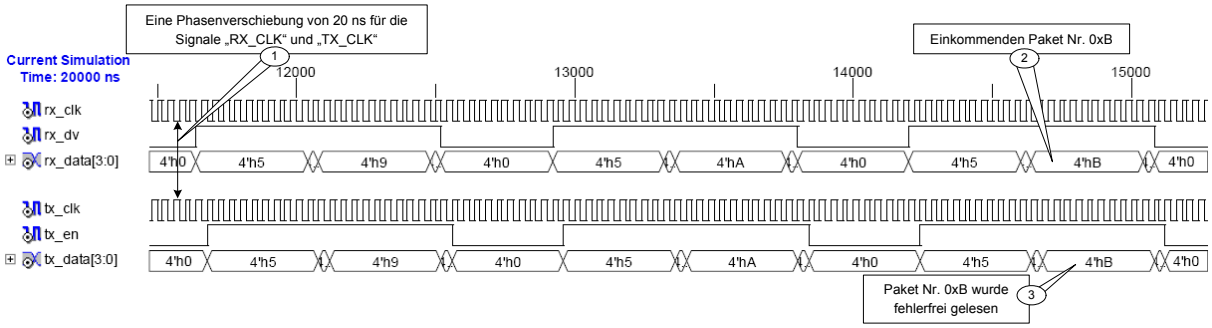

Abbildung 61: Fehlerfreie Übertragungen mit Phasenverschiebung

Ein weiters Problem, das durch das Modul „RTC_Sync“ gelöst werden soll, ist die Größe der „Duty-Cycle“ von den beiden Takt-Signalen „RX_CLK“ und „TX_CLK“. Die Abbildung 62 repräsentiert eine Waveform für eine Übertragung nach 10µs. Die „Duty-Cycle“ des Signals „RX_CLK“ liegt bei 20% (bei①). Die „Duty-Cycle“ für das Signal „TX_CLK“ liegt bei 80% (bei②). Anhand der Abbildung 62 ist zu merken, dass das empfangene Paket mit der Nr. ´0x8´ vollständig und fehlerfrei mit unterschiedlichen „Duty-Cycle“ der Takt-Signale „RX_CLK“ und „TX_CLK“ weitergeleitet wurde.

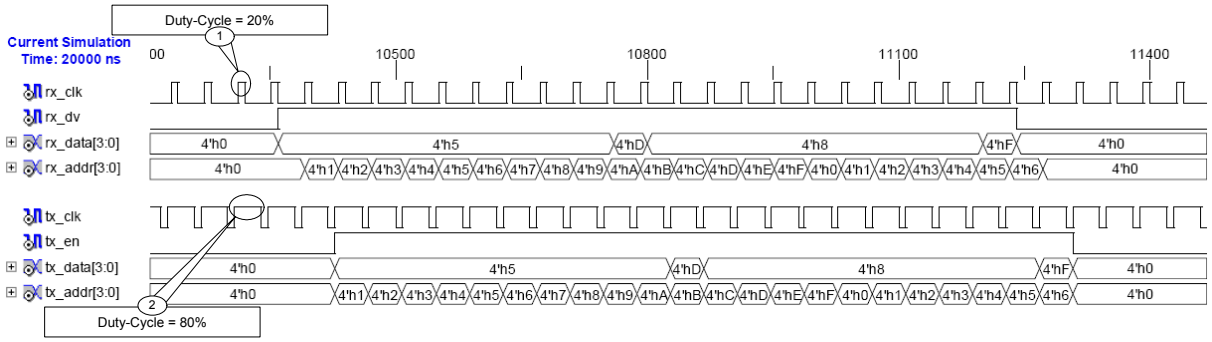

Abbildung 62: Unterschiedlichen „Duty-Cycle“ der Signale „RX_CLK“ und „TX_CLK“

Der Fall einer ungleichen Größe der „Duty-Cycle“ innerhalb eines Takt-Signals ist in der Abbildung 63 repräsentiert. In der Waveform liegt die Größe der „Duty-Cycle“ vom Signal „RX_CLK“ bei 50% (bei①) und nach neun Takten bei 20% (bei②). Für das Signal „TX_CLK“ liegt sie fest bei 80% (bei③) und nach neun Takten bei 50% (bei④). Hierbei liegt in diesem Fall ebenso eine Phasenverschiebung für die beiden Signale „RX_CLK“ und „TX_CLK“ (bei⑤) vor. An Hand der Waveform in der Abbildung 63 erfühlt das Modul „RTC_Sync“ die erwünschten Funktionalitäten für eine fehlerfreie Übertragung bei einer Phasenverschiebung und einem unterschiedlichen Wert der „Duty-Cycle“ für die Takt-Signale „RX_CLK“ und „TX_CLK“. Die Pakete wurden durch das Modul erkannt und in den Speicher gelegt und beim Leseprozess fehlerfrei aus dem Speicher gelesen.

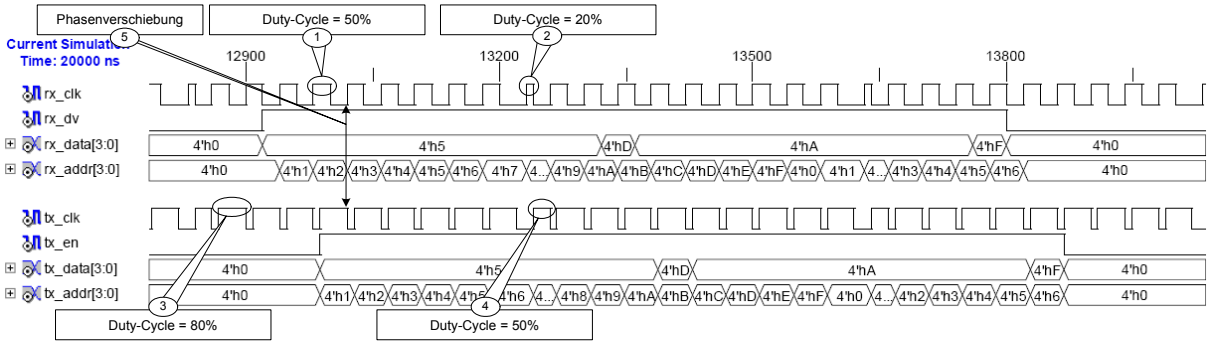

Abbildung 63: Ungleiche „Duty-Cycle“ bei einem Signal

5.5.3 RTC_Sync_PA

Das Modul „RTC_Sync_PA“ erweitert die Funktionalität des vorherigen Moduls „RTC_Sync“. In diesem Abschnitt wird anhand von Waveforms gezeigt, wie ein Paket mit unvollständiger Präambel erkannt wird und wie die neue erzeugt Präambel an dem empfangenen Paket angehängt wird. Die vorher diskutierte Funktionalität des Moduls „RTC_Sync“ soll bei Weiterleitung der Pakete erhalten bleiben. Dies wird ebenso anhand der Waveform gezeigt. Des Weiteren wird gezeigt, dass ein bestimmter Zeitabstand zwischen den von Teilnehmern gesendeten Paketen, die das Modul „RTC_Sync_PA“ empfangen, liegen muss. Es wird ebenso eine Übertragung gezeigt, die den Zeitabstand zwischen den gesendeten Paketen nicht erfüllt, was zur fehlerhaften Übertragung führt.

Die in der Abbildung 64 ersichtliche Waveform stellt das Empfangen eines Pakets mit einer Präambel von neun Nibbles und mit einem SFD von zwei Nibbles (bei①) dar. Das Paket wurde mit der steigende Flanke erkannt (bei②). Nach der Erkennung des Anfangs des Pakets befindet sich die Zustandmaschine im Zustand „search_sfd“ und wartet dabei, bis die zweite SFD-Nibble erkannt wird (bei③). Während des Wartens findet keine Speicherung für die einkommenden Daten statt. Die Speicherzelle Nr. ´0´ ist bis zur Erkennung des zweiten SFD-Nibbles mit dem Wert „UU“ belegt (bei④). Nach der Erkennung der zweiten SFD-Nibble wird die Zustandmaschine in den Zustand „write_ram“ überführt (bei⑤) und dann werden die empfangenen Daten in den Speicher gelegt. Das erste Nibble des Daten-Teiles (bei⑥) wird in der Speicherzelle Nr. ´0´ gelegt. Die Zustandmaschine wird bei steigender Flanke in den Zustand „rx_hold“ überführt, wenn der Wert des Signals „RX_DV“ von ´1´ nach ´0´ gesetzt wird (bei⑦). Zugleich wird der Trennungs-Wert „000000“ im Speicher gelegt (bei⑧), und die Speicheradresse „RX_ADDR“ wird auf ´0´ gesetzt.

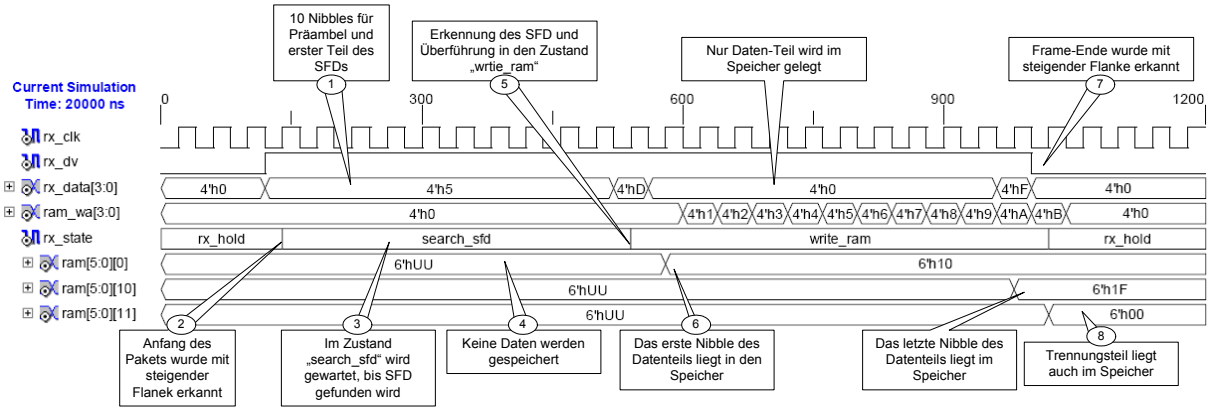

Abbildung 64: Empfangen eines Pakets

Der Prozess der Weitersendung von einem empfangen Paket ist in der Abbildung 65 dargestellt. Nach Erkennung des Anfangs des angekommenen Pakets (bei①) wird die Zustandmaschine in den Zustand „read_rom“ (bei②) überführt, und startet den Weiterleitungs-Prozess im nächsten Takt. In diesem Zustand wird die Präambel und das SFD aus dem vorprogrammierten ROM (bei③) gelesen. Anhand der Waveform ist zu sehen, wie das erste Nibble aus der Zelle ´0´ (bei④) und das letzte Nibble (=zweite SFD-Nibble) aus der Zelle ´15´ (bei⑤) gelesen wurden. Ist die Speicherzelle mit der Nummer „0xF“ im ROM erreicht (bei⑥), wird die Zustandmaschine im Zustand „read_ram“ überführt und die Daten werden aus dem RAM gelesen (bei⑦). Der ganze Leseprozess wird gestoppt, wenn das Signal „TX_EN“ sein Zustand von ´1´ nach ´0´ ändert, dies wird anhand des Lesens von dem im RAM gespeicherten Trennungs-Wert „000000“ erkannt (bei⑧). Danach wird die Zustandmaschine in den Zustand „tx_hold“ überführt und es findet kein Lesen von Daten mehr statt.

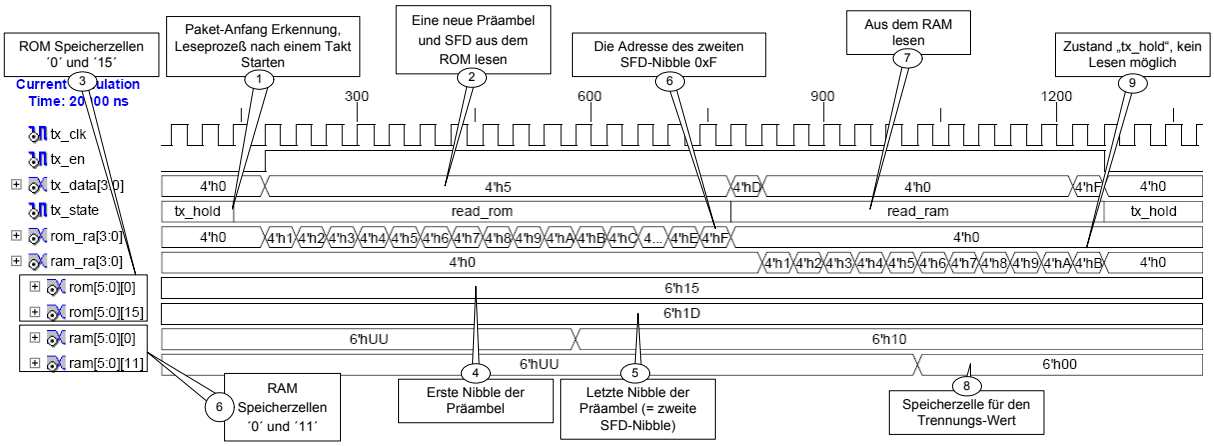

Abbildung 65: Weiterleiten eines Pakets

Die nachfolgende Abbildung 66 zeigt eine Waveform für drei hintereinander einkommende Pakete, die unvollständige Präambel und SFD beinhalten. Die Pakete wurden fehlerfrei und mit vollständiger Präambel und SFD weiter gesendet. Anhand der Simulation kann man sehen, wie das angekommene Paket mit der Nummer ´0xA´ (bei①), das 11 Nibbles für Präambel und SFD enthält, mit einer neuen vollständigen Präambel und SFD weiter übertragen wurde (bei②). Ein

wichtiges Element für fehlerfreie Übertragung ist das IFG (bei③). Die IFG-Zählung zwischen den gesendeten Paketen muss größer als die Anzahl der fehlenden Teile von der Präambel sein.

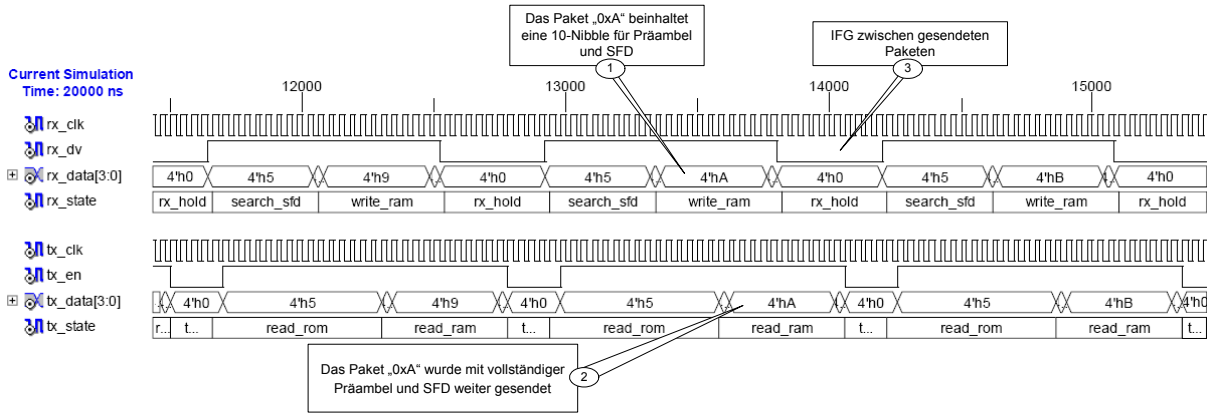

Abbildung 66: Weitersendung von mehreren Paketen

In der nachfolgenden Abbildung 67 wird eine fehlerhafte Übertragung dargestellt. Die Zählung zwischen den ankommenden Paketen ist kleiner als die fehlende Präambel (bei①), demzufolge findet nach bestimmter Übertragung eine Überschreibung der Pakete statt. Hierbei wird der Wert eines Pakets durch das nächste ankommende Paket überschrieben. Anhand der Waveform ist zu sehen, wie das empfangene Paket mit der Nummer ´0x6´ (bei②) nicht weitergesendet wurde. Stattdessen wurde es durch das Paket Nr. ´0x7´ überschrieben und das Paket Nr. ´0x7´ weitergesendet.

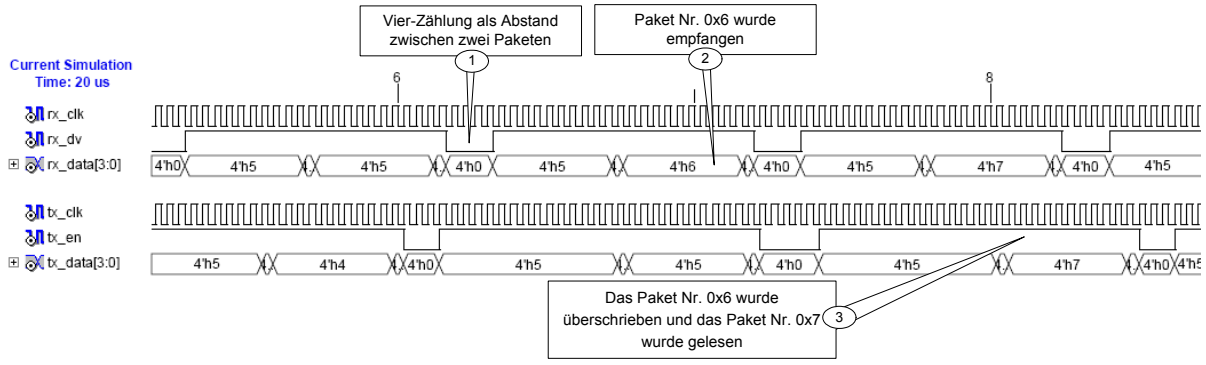

Abbildung 67: Fehlerhafte Übertragung

5.5.4 RTC_CORE

In diesem Abschnitt wird die Funktionalität des Moduls „RTC_CORE“ verifiziert. Die durchgeführte Post-Layout-Simulation für dieses Modul wurde aufgrund des Skew-Problems des Synchronisations-Signals „Sync_CLK“ mittels des Bausteins „Virtex-5 LX“ durchgeführt. Die Da die Aufgabe dieses Moduls die Umschaltung zwischen vier MII-Schnittstellen ist, wird im Rahmen dieser Simulation darauf geachtet, ob anhand eines vordefinierten Schedulers die ankommenden Pakete an der entsprechenden Transmit-Schnittstelle jedes Ports richtig und fehlerfrei weitergeleitet werden. Des Weiteren wird natürlich die Phasenverschiebung der Takt-Signale „RX_CLK“ und „TX_CLK“ und das „Duty-Cycle“ an jedem Port simuliert. Um die Komplexität der Betrachtung von den ankommenden Paketen an den jeweiligen Ports (Port A, Port B, Port C und Port D) zu vereinfachen, wird eine neue Testbench erzeugt. Die Testbench sendet an jeden Port ein Paket mit festem Wert im Datenteil. Beispielsweise empfängt der Port A immer das Paket mit dem Wert ´0xA´ im Datenteil. Dies macht es einfacher zu erkennen, wenn z.B. der Port A ein Paket mit dem Wert ´0xC´ weiterleitet, von welchem Port dieses Paket empfangen wurde (in diesem Fall aus dem Port C). In Rahmen dieser Simulation sendet die Testbench Pakete mit dem Wert „A“ zum Port A, dem Wert „B“ zu Port B, dem Wert „C“ zu Port C und dem Wert „D“ zu Port D. Die Abbildung 68 stellt einen Blockschaltplan dar, in dem jeder Port bzw. jede MII-Schnittstelle in „MII_RX“-Schnittstelle und „MII_TX“-Schnittstelle geteilt ist. „MIIA_RX“-Schnittstelle bekommt immer Pakete mit dem Wert „A“ im Daten-Teil und aus der „MIIA_TX“-Schnittstelle sollen die zu leitenden Pakete heraus kommen. Dies verläuft anlog für die restlichen Ports.

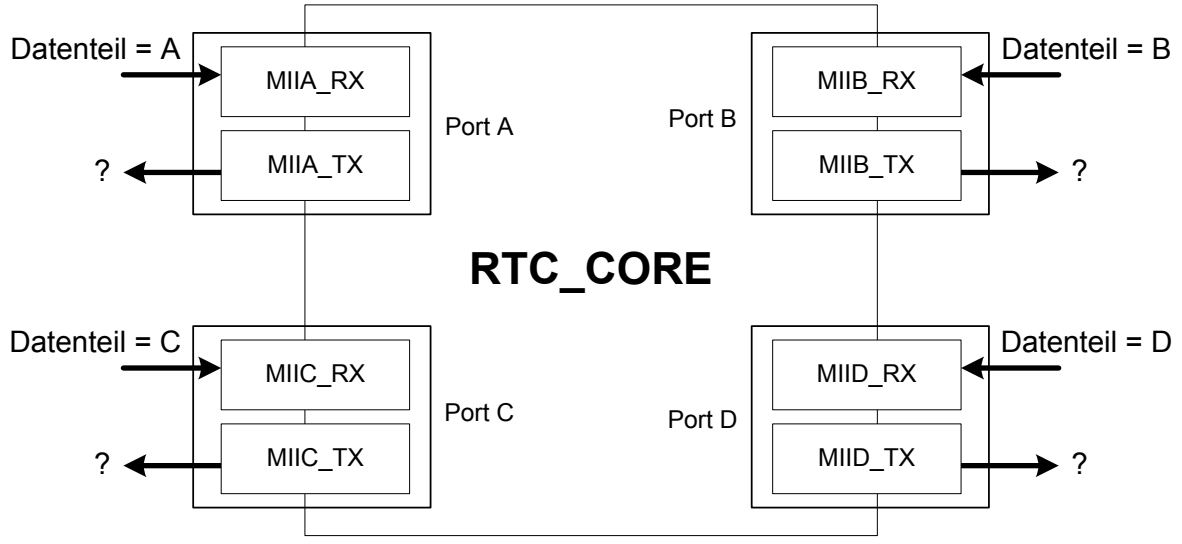

Abbildung 68: Port des "RTC_CORE"

Der Scheduler in der Tabelle 10 wird für Simulationszwecke eingesetzt. Die erste Zeile in der Tabelle 10 definiert die Umschaltung für den ersten Zeitschlitz, die Codierung der Scheduler wurde im Kapitel 5.4.3 diskutiert und ist anhand der Tabelle 9 zu interpretieren. Die erste Zeile mit dem Wert „A, C, B, D“ in der Spalte „Richtung“ wird von links nach richt wie folgt gelesen:

Port A bekommt Pakete von Port A; Port B bekommt Pakete von Port C; Port C bekommt Pakete von Port C und Port D bekommt Pakete von Port D.

Index	Richtung	Codierung
0x0	A, C, B, D	00100111
0x1	B, D, C, A	01111000
0x2	C, A, D, B	10001101
0x3	D, B, A, C	11010010
0x4	C, C, A, C	10100010
0x5	B, D, A, C	01110010
0x6	A, D, C, C	00111010
0x7	B, A, A, C	01000010

Tabelle 10: Scheduler der Umschaltung

Die Waveform in der Abbildung 69 stellt die empfangenen Pakete an den einzelnen Ports (A, B, C, und D) dar. Bei dieser Simulation werden nur die Signale „RX_DV“ und „RX_D[3:0]“ an jedem Port betrachtet (von①bis④). Die Größen (Phasenverschiebung und „Duty Cycle“) für die Takt-Signale „RX_CLK“ und „TX_CLK“ an den Ports sind zuerst gleich und werden in den weiteren Simulationen betrachtet. Mit der steigenden Flanke des Signals „Sync_CLK“ (bei⑤) starten die Teilnehmer, Pakete mit 10-Nibbles-Präambel und 10-Nibbles-Daten-Teil an den entsprechenden Ports zu senden. An den jeweiligen Ports wurden Pakete empfangen, deren Daten-Teil gleich dem Wert der Port-Bezeichnung ist (bei❶❷❸❹). Der Abstand zwischen den empfangenen Paketen (bei⑥]) ist durch den Zeitschlitz bzw. das Signal „Sync_CLK“ definiert.

Die Abbildung 70 stellt den TX-Prozess des Moduls „RTC_CORE“ dar. Hierbei sendet das Modul die empfangenen Pakete, die in der Abbildung 69 dargestellt sind, mit vollständiger Präambel aus. Die Umschaltung zwischen den Ports wird in Bezug zum Scheduler durchgeführt. In der Waveform wird die Scheduler-Adresse „schedule_ad“ dargstellt, die für die nächste Umschaltung der Ports repräsentiert (bei①) ist. Der aktuelle Zustand jedes Ports ist durch das Signal „MII#_state“ dargestellt (bei②). Das Symbol „#“ entspricht der Bezeichnung der Ports. Der bei② gezeigte Zustand implementiert den Eintrag Nummer ´0x1´ in der Tabelle 10. Dabei wurden die Daten aus dem Port B zum Port A weitergeleitet, dies entspricht den Eintrag Nummer ´0x1´ in der Tabelle 10. Die gesendeten Pakete wurden mit vollständiger Präambel aus 14-Nibble weitergeleitet (bei③). Man kann leider wegen der Auflösung der Waveform die Teile der Präambel nicht zählen, jedoch kann man sie mit dem Daten-Teil, der aus 10-Nibble besteht, vergleichen. Der Eintrag Nummer ´0x4´ lässt das Port A offen (bei④), hierbei werden keine Pakete der anderen Ports zum Port A weitergeleitet. Nach der Spezifikation der RTC, die im Kapitel 3.3.1 vorgestellt wurde, muss das Port A zum MAC umgeschaltet werden. Jedoch werden die Anschlüsse zum MAC im Rahmen dieser Arbeit wegen Untersuchungszwecke nicht implementiert.

Die Abbildung 71 repräsentierte eine Waveform mit fehlerfreier Übertragung von Paketen. Die Pakete wurden mit einer Phasenverschiebung der Takt-Signale „TX_CLK“ gesendet. Die Phasenverschiebung der Signale ist durch die blauen Linien zu sehen. Des Weiteren finden die

Übertragungen ebenso mit unterschiedlichem Wert der „Duty-Cycle“ statt, die durch die blauen Kreise zu sehen sind.

Anhand der hier gezeigten Waveform der Simulation an dem Modul „RTC_CORE“ ist zu beweisen, dass die aufgetretenen Probleme durch die Realisierung von RTC durch das Modul zu lösen. Durch den Einsatz des Moduls „RTC_CORE“ haben Probleme wie Phasenverschiebung, unvollständige Präambel und unterschiedliche „Duty-Cycle“ keinen Einfluss mehr auf die Übertragung der Pakete auf die MII-Ebene.

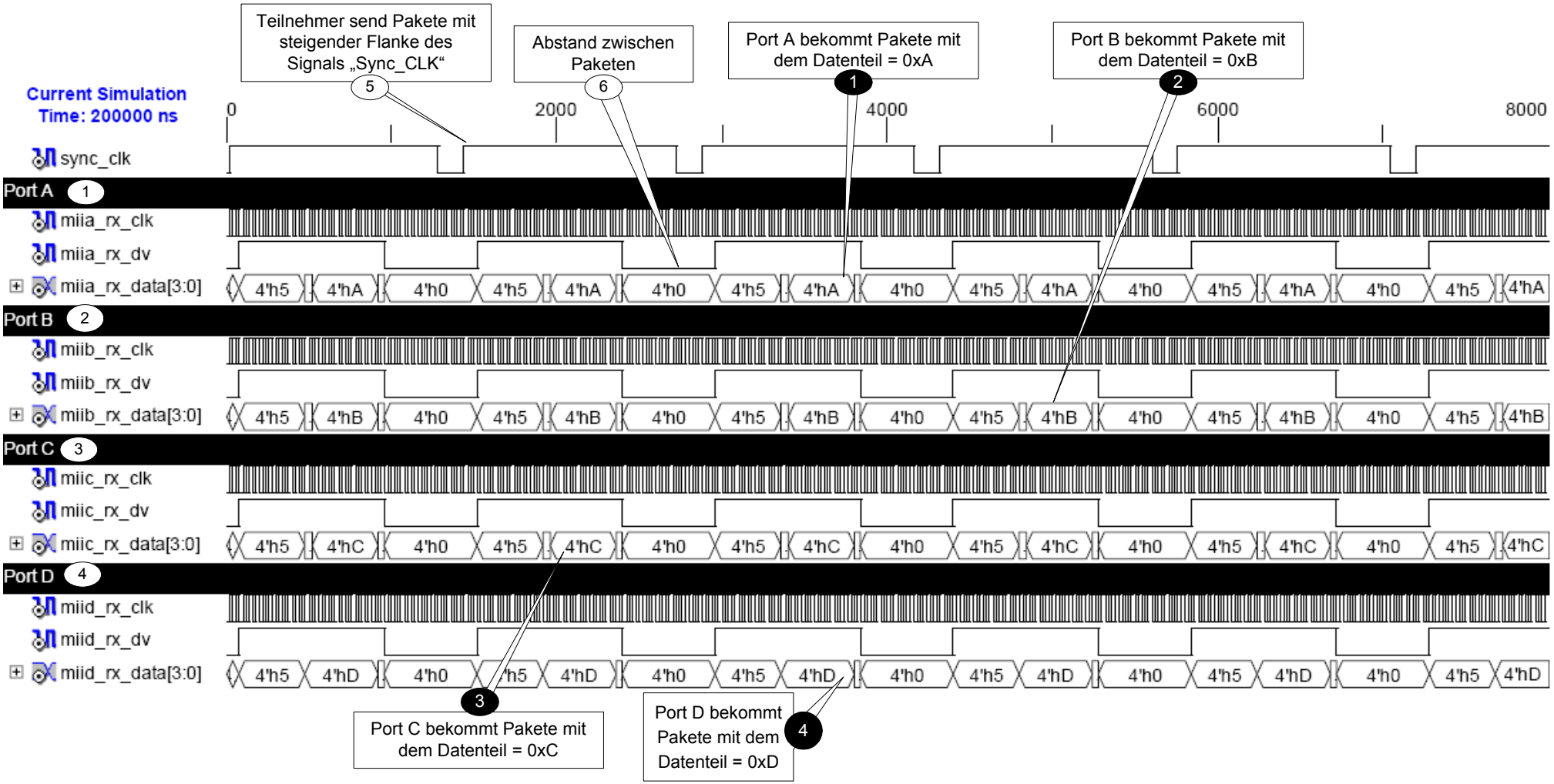

Abbildung 69: Empfangene Pakete an den jeweiligen Ports

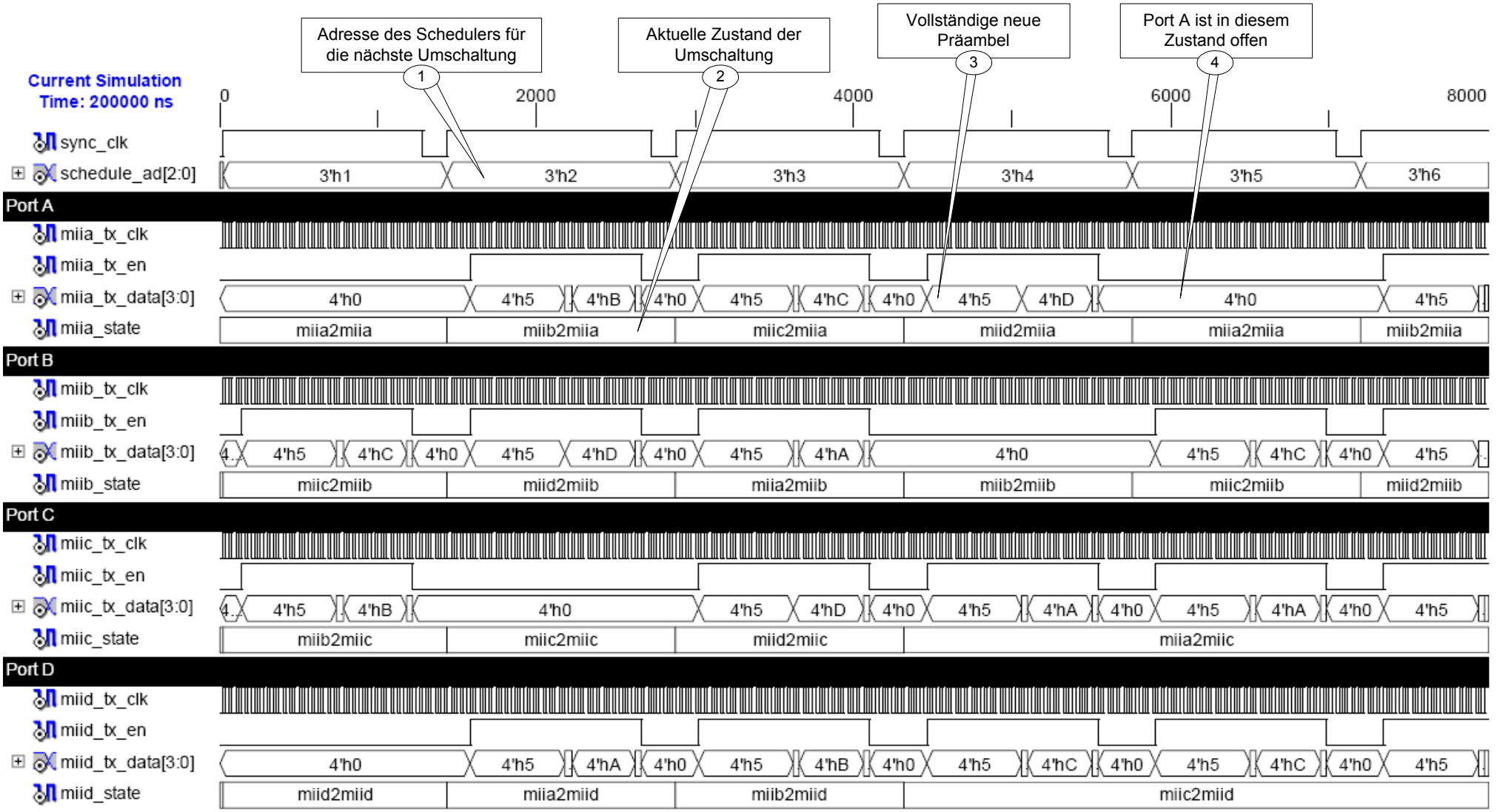

Abbildung 70: Gesendete Pakete an den jeweiligen Ports

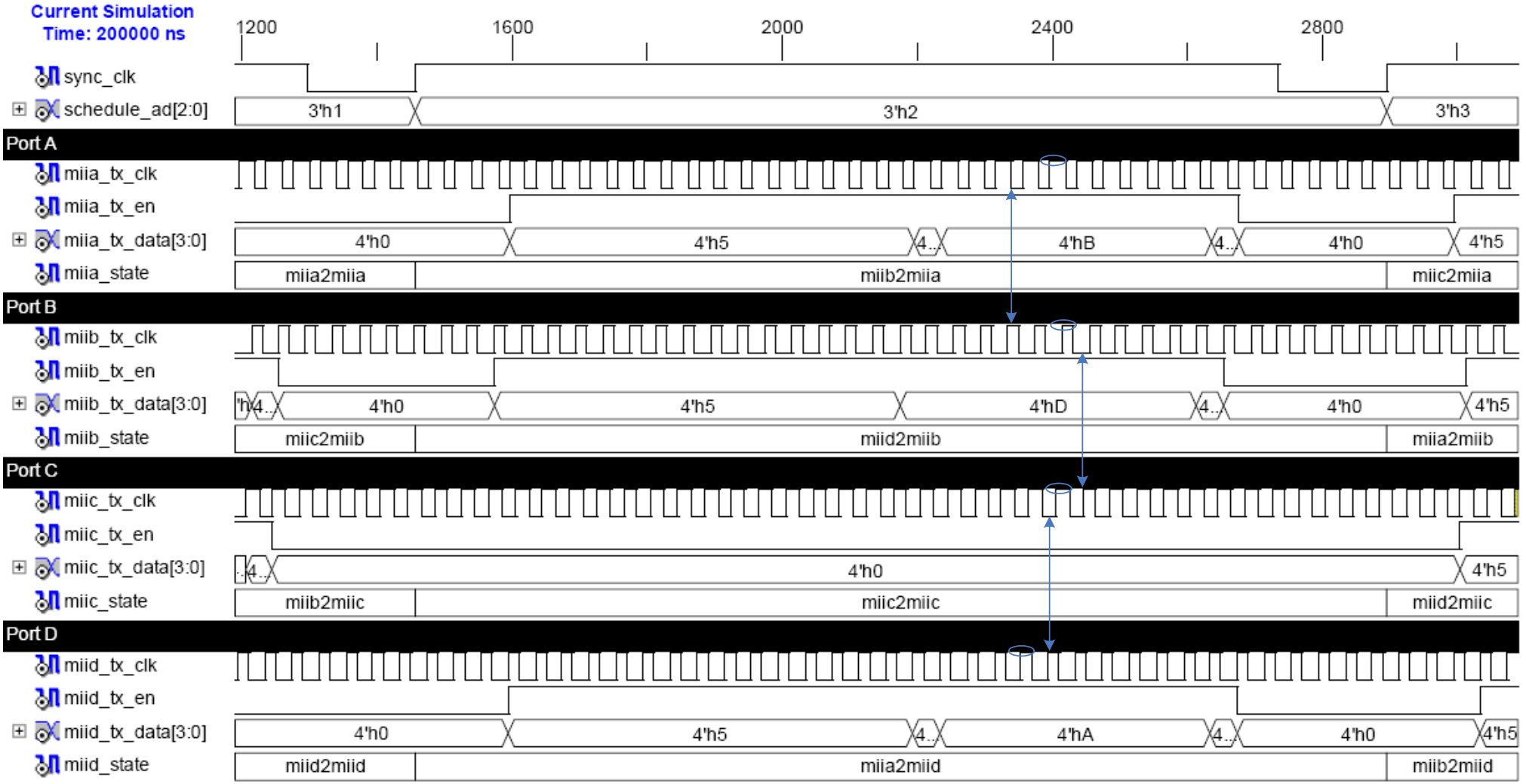

Abbildung 71: Gesendete Pakete mit Phasenverschiebung & "Duty-Cycle"

5.6 Synthese der Module

In diesem Abschnitt wird die Synthese der drei VHDL-Modelle „RTC_Sync“, „RTC_Sync_PA“ und „RTC_CORE“ behandelt. Durch die Synthesis werden die VHDL-Modelle bzw. die entworfenen VHDL-Programme, die eine algorithmische Verhaltungsbeschreibung repräsentieren, in eine strukturelle Darstellung, wie z.B. Speicher, Flipflop und Multiplexer übersetzt. Anhand der Synthese-Log-Dateien wird in diesem Abschnitt gezeigt bzw. bewiesen, dass die erwünschten Hardware-Komponenten in den vorherigen Abschnitten realisierbar sind. Die Synthese an den Modulen findet an den FPGA-Baustein-Sparten-3 statt. Der Bausteine „Sparten-3“ wurde wegen Anzahl der PINs, Verfügbarkeit im Markt und der Preise gewählt. Hierbei wird gezeigt, wie viele Ressourcen (Slices, LUT und IO) des FPGA-Bausteins genutzt werden, um die entworfenen Module zu realisieren.

Für die Synthese kann man Constraints definieren, die die Implementierung der VHDL-Module beschreibt. Dabei kann die Implementierung der Speicher oder der Zustandsmaschinen auf die integrierte RAMs im FPGA oder auf LUTs realisiert werden. Die Implementierung der Komponenten in der hier durchgeführten Synthese wird auf LUTs realisiert.

5.6.1 RTC_Sync

Nach Durchführung der Synthese (Auszug 1) für das Modul „RTC_Sync“ wurde das Signal „mem“ als 16x16-Bit Dual-Port RAM (Zeile 3) erkannt. Die Signale „RX_ADDR“ (Zeile 4) und „TX_ADDR“ (Zeile 6) wurden als 4-Bit-Zähler erkannt. Zwei 1-Bit Register wurden für die Steuer-Signale „rx_state“ (Zeile 5) und „tx_state“ (Zeile 8) erkannt. Der nachfolgende Auszug der Synthese-Log-Datei zeigt die erkannten Komponenten an.

```
1  ynthesizing Unit <RTC_Sync>.
2      Related source file is "D:/FPGA/RTC_CORE/RTC_Sync.vhd".
3      Found 16x6-bit dual-port RAM <Mram_mem> for signal <mem>.
4      Found 4-bit up counter for signal <RX_ADDR>.
5      Found 1-bit register for signal <rx_state<0>>.
6      Found 4-bit up counter for signal <TX_ADDR>.
7      Found 6-bit register for signal <TX_BUS>.
8      Found 1-bit register for signal <tx_state<0>>.
9      Summary:
10     inferred   1 RAM(s).
11     inferred   2 Counter(s).
12     inferred   8 D-type flip-flop(s).
13 Unit <RTC_Sync> synthesized.
```

Auszug 1: Erkennung der Komponenten für "RTC_Sync"

Der nachfolgende Auszug 2 zeigt die Anzahl der benutzten Slices, LUTs und IOs für die Realisierung des Moduls „RTC_Sync“ an. Es wurden 18 aus 3584 Slices benötigt, was 0% entspricht. Aus den 7168 LUTs wurden 32 LUTs verwendet, dabei wurden 20 LUTs zur Realisierung von Logik-Funktionen und 12 LUTs zur Realisierung des Speichers verwendet. Es wurden 15 IOBs (Input Outpt Blocks) aus den 97 verfügbaren IOBs verwendet, um die 14-

Leitungen der MII-Schnittstelle und eine Leitung für RESET am FPGA anzuschließen. Aus dem 8-PIN verfügbaren GCLKs (Global Clocks) wurden 2 (Zeile 13) für die Signale „RX_CLK“ und „TX_CLK“ genutzt.

```
1  Device utilization summary:
2  ---------------------------
3
4  Selected Device : 3s400tq144-5
5
6   Number of Slices:                      18  out of   3584     0%
7   Number of Slice Flip Flops:            19  out of   7168     0%
8   Number of 4 input LUTs:                32  out of   7168     0%
9      Number used as logic:               20
10     Number used as RAMs:                12
11  Number of IOs:                         15
12  Number of bonded IOBs:                 15  out of     97    15%
13  Number of GCLKs:                        2  out of      8    25%
```

Auszug 2: Aufteilung der Ressourcen für „RTC_Sync“

5.6.2 RTC_Sync_PA

Die Funktionalität des Moduls „RTC_Sync“ wurde durch das Modul „RTC_Sync_PA“ erweitert. Die zusätzlichen Komponenten auf dem Modul „RTC_Sync“, welche zum Modul „RTC_Sync_PA“ führen, werden nun näher betrachtet. Der ROM-Speicher, der die vollständige Präambel enthält, wurde als 16x6-Bit (Zeile 3) erkannt. Die Zustandsmaschine, die das Empfangen von Daten steuert, wurde durch das Signal „rx_state“ (Zeile 5) erkannt. Durch das Signal „rx_state“ (Zeile 6) wurde die Zustandmaschine erkannt, die für den Lese-Prozess verantwortlich ist.

```
1  Synthesizing Unit <RTC_Sync_PA>.
2      Related source file is "D:/FPGA/RTC_CORE/RTC_Sync_PA.vhd".
3      Found 16x6-bit ROM for signal <TX_BUS$rom0000> created at line 129.
4      Found 16x6-bit dual-port RAM <Mram_RAM> for signal <RAM>.
5      Found finite state machine <FSM_0> for signal <rx_state>.
6      Found finite state machine <FSM_1> for signal <tx_state>.
7      Found 4-bit up counter for signal <RAM_RA>.
8      Found 4-bit up counter for signal <RAM_WA>.
9      Found 4-bit up counter for signal <ROM_RA>.
10     Found 6-bit register for signal <TX_BUS>.
11     Summary:
12     inferred   2 Finite State Machine(s).
13     inferred   1 RAM(s).
14     inferred   1 ROM(s).
15     inferred   3 Counter(s).
16     inferred   6 D-type flip-flop(s).
17 Unit <RTC_Sync_PA> synthesized.
```

Auszug 3: Erkennung der Komponenten für "RTC_Sync_PA"

Die Anzahl der genutzten Slices, die Anhand des Auszugs 4 ersichtlicht ist, sind von anfänglich 18 im Modul „RTC_Sync“ auf 24 im Modul „RTC_Sync_PA“ gestiegen (Zeile 6). Des Weiteren wurden 44 LUTs benötigt, dies sind 12 LUTs mehr als beim Ausgangsmodul „RTC_Sync“.

```
1  Device utilization summary:
2  ---------------------------
3
4  Selected Device : 3s400tq144-5
5
6   Number of Slices:                       24  out of   3584     0%
7   Number of Slice Flip Flops:             22  out of   7168     0%
8   Number of 4 input LUTs:                 44  out of   7168     0%
9      Number used as logic:                32
10     Number used as RAMs:                 12
11  Number of IOs:                          15
12  Number of bonded IOBs:                  15  out of     97    15%
13  Number of GCLKs:                         2  out of      8    25%
```

Auszug 4: Aufteilung der Ressourcen für „RTC_Sync_PA“

5.6.3 RTC_CORE

Das Modul „RTC_CORE“ besteht unter anderem aus dem Modul „CORE_CLOCK_GEN“ und vier weiteren „RTC_Sync_PA“ Modulen. Bei Durchführung der Synthese an dem Modul „RTC_CORE“ findet an jedem erkannten Modul eine separate Synthese statt. Das Ergebnis der Synthese des Moduls „RTC_Sync_PA“ wurde bereits im vorherigen Abschnitt diskutiert. Bei der Synthese des Moduls „CORE_CLOCK_GEN“, siehe Auszug 5, wurde ein Zähler (Zeile 3) erkannt, der bei einer bestimmten Anzahl externer Takt-Signale ein Synchronisationssignal erzeugt.

```
1  Synthesizing Unit <CORE_CLOCK_GEN>.
2      Related source file is "D:/FPGA/RTC_CORE/CORE_CLOCK_GEN.vhd".
3      Found 32-bit up counter for signal <count>.
4      Found 1-bit register for signal <int_CLK>.
5      Summary:
6      inferred   1 Counter(s).
7      inferred   1 D-type flip-flop(s).
8  Unit <CORE_CLOCK_GEN> synthesized.
```

Auszug 5: Erkennung der Komponenten für „CORE_CLOCK_GEN“

Der Auszug 6 zeigt die Ausgabe der Synthese des Moduls „RTC_CORE“. Der Scheduler-Halter wurde als ROM (Zeile 3) erkannt. Für das Switching zwischen den vier Ports wurden vier Multiplexer je 7-Bit verwendet. Ein Zähler wurde erkannt, der die Lese-Adresse für den Scheduler erzeugt.

```
1  Synthesizing Unit <RTC_CORE>.
2      Related source file is "D:/FPGA/RTC_CORE/RTC_CORE.vhd".
3      Found 8x8-bit ROM for signal <schedule$mux0000> created at line 201.
4      Found 7-bit 4-to-1 multiplexer for signal <A_RX>.
5      Found 7-bit 4-to-1 multiplexer for signal <B_RX>.
6      Found 7-bit 4-to-1 multiplexer for signal <C_RX>.
7      Found 7-bit 4-to-1 multiplexer for signal <D_RX>.
8      Found 8-bit register for signal <schedule>.
9      Found 3-bit up counter for signal <schedule_ad>.
10     Summary:
11     inferred   1 ROM(s).
12     inferred   1 Counter(s).
13     inferred   8 D-type flip-flop(s).
14     inferred  28 Multiplexer(s).
15 Unit <RTC_CORE> synthesized.
```

Auszug 6: Erkennung der Komponenten für "RTC_CORE"

Der nächste Auszug 7 listet alle erkannten Komponenten des Moduls „RTC_CORE“ auf. Da das Modul „RTC_CORE“ vier „RTC_Sync_PA“ beinhaltet, werden normalerweise acht Zustandsmaschinen benötigt. Jedoch wird infolge eines Optimierungsprozesses (in der Datei „RTC_CORE.syr“ beschrieben) die Verwendung von zwei Zustandsmaschinen (Zeile 2) statt anfänglich acht Zustandsmaschinen erlaubt. Es wurden vier RAMs erkannt (Zeile 3), die sich in je einem der vier Module „RTC_Sync_PA“ befindet. Fünf ROMs wurden erkannt (Zeile 5), vier 16x6-Bit ROM für die Module „RTC_Sync_PA“ und der 8x8-Bit ROM im „RTC_CORE“. 14 Zähler benötigt das Modul „RTC_CORE“. Der 3-Bit-Zähler erzeugt Lese-Adressen für den Scheduler, der 32-Bit-Zähler zählt die Takt-Signale in Modul „CORE_CLOCK_GEN“. Jeder der vier Module „RTC_Sync_PA“ benötigt drei Zähler, dies macht in der Summe 12 Zähler (Zeile 11).

```
1  Macro Statistics
2  # FSMs                                          : 2
3  # RAMs                                          : 4
4   16x6-bit dual-port distributed RAM             : 4
5  # ROMs                                          : 5
6   16x6-bit ROM                                   : 4
7   8x8-bit ROM                                    : 1
8  # Counters                                      : 14
9   3-bit up counter                               : 1
10  32-bit up counter                              : 1
11  4-bit up counter                               : 12
12 # Registers                                     : 49
13  Flip-Flops                                     : 49
14 # Multiplexers                                  : 4
15  7-bit 4-to-1 multiplexer                       : 4
```

Auszug 7: Auflistung der gesamten Komponenten für "RTC_CORE"

Die Verteilung der programmierten Komponenten an die Ressourcen des FPGA-Bausteins ist anhand des Auszugs 8 nachzuvollziehen. Es wurden 4% aller vorhandenen Slices verwendet. Da jedes Slice aus LUT und Flipflop besteht, wurde für dieses Modul 132 Flipflops aus dem 7168 verwendet. Von allen 7168 LUTs wurden 3% benutzt, davon wurden 233 LUTs für Logik-Implementierungen und 48 LUTs für RAM-Implementierung eingesetzt. Das Modul „RTC_CORE“

benötigt 58 IOs aus den 97 vorhandenen IOs. Es wurden die kompletten acht GCLKs verwendet, hierbei wird eine 100% Verwendung der Ressourcen belegt.

```
Device utilization summary:
---------------------------

Selected Device : 3s400tq144-5

 Number of Slices:                      149  out of   3584     4%
 Number of Slice Flip Flops:            132  out of   7168     1%
 Number of 4 input LUTs:                281  out of   7168     3%
    Number used as logic:               233
    Number used as RAMs:                 48
 Number of IOs:                          58
 Number of bonded IOBs:                  58  out of     97    59%
 Number of GCLKs:                         8  out of      8   100%
```

Auszug 8: Aufteilung der Ressourcen für „RTC_CORE“

Das Modul „RTC_CORE“ wurde aufgrund des Skew-Problems des Synchronisations-Signal „Sync_CLK“ auf dem FPGA-Baustein „Virtex5-XC55VLX30“ synthetisiert. Ein ausführlicher Auszug der Ergebnisse steht im Anhang.

5.7 Zeit-Analyse an der Module

Nach Ablauf der Synthesis wird der Prozess „Place und Route“ durchgeführt. Im Prozess „Place & Route“ können die durch die Synthesis erkannten Komponenten auf bestimmte Positionen des FPGA-Bausteins platziert (Place) und miteinander verdrahtet (Route) werden. Der Prozess „Place & Route“ kann sowohl automatisch als auch manuell ausgeführt werden. In dieser Arbeit wird die automatische Variante durchgeführt. Da jeder FPGA-Entwurf bestimmten Zeitanforderungen unterliegt, werden diese Anforderungen als „Constraints“ angegeben. Die „Constraints“ werden während des automatischen „Place & Route“ Prozesses berücksichtigt. Des Weiteren wird bei Durchführung dieses Prozesses die Geschwindigkeit des FPGA-Bausteins berechnet bzw. bestimmt.

Im nächsten Abschnitt werden im Bezug auf die im Rahmen dieser Arbeit entworfenen Module die Zeitanforderungen festgelegt. Des Weiteren werden Auszüge des Zeit-Analyse-Berichts dargestellt.

5.7.1 Festlegung von Constraints

Um eine Zeit-Analyse auf einem FPGA durchzuführen, werden die Constraints „Pad-to-Setup“, „Clock-to-Pad“ und „Clock-to-Setup“ untersucht. Diese Constraints sind Zeit-Größen, die in diesem Fall durch die Zeitanforderungen „Setup-Time“, „Hold-Time“ und „Delay-Time“ der MII-Schnittstelle festgelegt werden.

In den nachfolgenden Unterabschnitten werde ich näher auf die Constraints eingehen. Des Weiteren gebe ich einleitend ein, dass unter einem synchronen Element ein Flip-Flop, Latch oder ein RAM zu verstehen ist, welches durch ein Takt-Signal gesteuert wird.

5.7.1.1 Pad-to-Setup

Die „Pad-to-Setup“ Zeit ist die maximale Zeit, die die Daten benötigen, um in den FPGA-Baustein zu gelangen und dort die interne Logik und die Routing zu durchlaufen, bis sie (die Daten) den Eingang des ersten synchronen Elements erreichen. Die synchronen Elemente besitzen eine Zeitanforderung „Setup-Time“ an ihrem Eingang. Die nachfolgende Abbildung 72 zeigt den Weg der Daten zum Eingang eines synchronen Flip-Flop. Für den FPGA-Baustein bedeutet dies, dass die ankommenden Daten (bei 1) für eine bestimmte Zeit (bei 2) vor der aktiven Flanke des Takt-Signals (bei 3) vorliegen müssen.

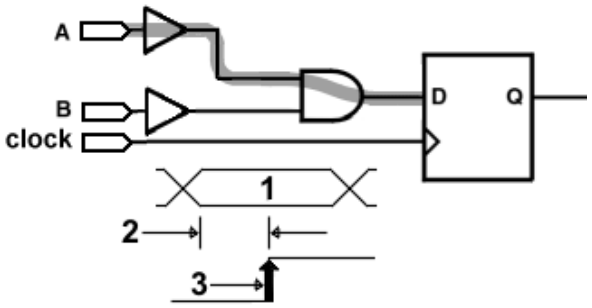

Abbildung 72: "Pad-to-Setup" Weg [30]

Bei Definition der „Pad-to-Setup“ Zeit aller entworfenen Module im Rahmen dieser Arbeit entsprechen die ankommenden Daten den RX-Daten „RX_DV“, „RX_D[3:0]“ und „RX_ER“. Das Takt-Signal lautet hierfür „RX_CLK“. Die zeitliche Beziehung zwischen den RX-Daten und dem RX-Takt-Signal wird durch den IEEE 802.3 bzw. MicroPHY-Baustein festgelegt bzw. definiert. Wie im Kapitel 2.1.5.3 gezeigt wurde, definiert der IEEE 802.3, dass die RX-Daten 10ns vor der steigenden Flanke des RX-Takt-Signals liegen müssen. Im Kapitel 4.1 wurde gezeigt, dass der MicroPHY-Baustein die von den IEEE 802.3 festgelegten Anforderungen erfüllt. Im Rahmen der Zeit-Analyse wird die maximal erlaubte „Pad-to-Setup“ Zeit durch die Variable „OFFSET IN BEFORE“ definiert und mit 10ns festgelegt.

5.7.1.2 Clock-to-Pad

Die „Clock-to-Pad“ definiert wie viel Zeit vergehen darf, bis die Daten nach der aktiven Takt-Flanke an den Ausgangs-Pins des FPGA-Bausteins vorliegen. Die Abbildung 73 zeigt den Weg, wie die „Clock-to-Pad“ Zeit berechnet wird. Der Weg der Daten erstreckt sich von dem Eingang des synchronen Elements, über die interne Logik und die Routing, bis zum Ausgangs-Pins des FPGA-Bausteins.

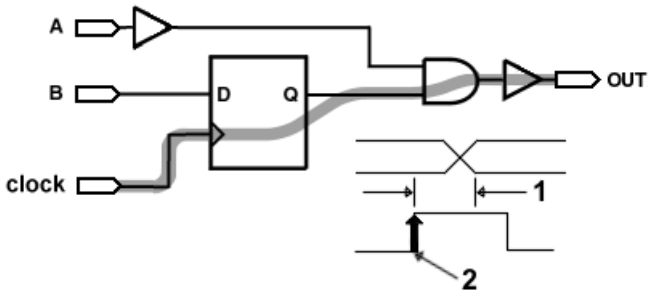

Abbildung 73: "Clock-to-Pad" Weg [30]

Für alle entworfenen Module im Rahmen dieser Arbeit beinhaltet die „Clock-to-Pad"-Zeit, die maximal erlaubte Zeit nach der aktiven Flanke des TX-Takt-Signals, bis die TX-Daten an den Ausgangs-Pins des FPGA vorliegen müssen. Der IEEE 802.3 definiert diese Zeit durch die „Delay-Time" und legt sie auf maximal 25ns fest. Im Rahmen der hier durchzuführenden Zeit-Analyse wird die maximale erlaubte „Clock-to-Pad" Zeit durch die Variable „OFFSET OUT AFTER" definiert und mit 25ns festgelegt.

5.7.1.3 Clock-to-Setup

Die „Clock-to-Setup"-Zeit ist die maximale Zeit, die die Daten im Inneren des FPGA benötigen, um vom Eingang eines synchronen Elements über die interne Logik und die Routing bis zum Eingang des nächsten synchronen Elements zu gelangen. Beide synchronen Elemente werden vom selben Takt-Signal gesteuert. Die Übertragung der Daten muss innerhalb derselben Takt-Periode stattfinden. Die nachfolgende Abbildung 74 zeigt den Weg der Daten im Bezug auf die „Clock-to-Setup" Zeit.

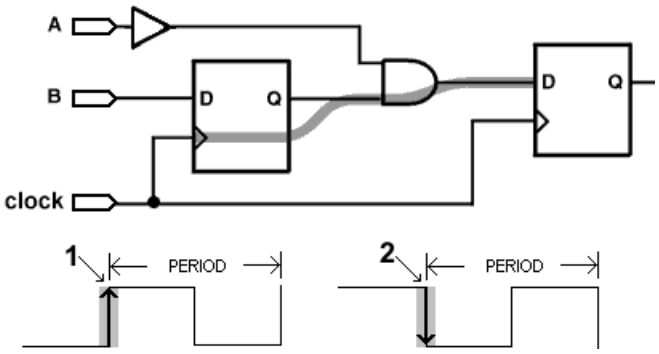

Abbildung 74: "Clock-to-Setup" Weg [30]

Die „Clock-to-Setup"-Zeit dient zur Bestimmung der Geschwindigkeit von FPGA-Bausteinen. Im Allgemeinen wird die maximale Taktfrequenz eines FPGA-Bausteins durch den kritischen Pfad bestimmt. Der kritische Pfad wird als längster Pfad vom Flipflop-Eingang zum nächsten Flipflop-Eingang definiert. Demzufolge wird die längste „Clock-to-Setup" Zeit gesucht, die die maximale Taktfrequenz bestimmt. Die Berechnung der maximalen Taktfrequenz des FPGA-Bausteins findet automatisch statt. Im Rahmen der hier durchzuführenden Zeit-Analyse wird anhand des

Auszüge nur die längste „Clock-to-Setup“-Zeit und die maximale Taktfrequenz des FPGA-Bausteins gezeigt.

5.7.2 Ergebnisse der Zeit-Analyse

Im vorherigen Abschnitt wurden Zeitanforderungen festgelegt, die von den entworfenen VHDL-Modulen auf einem FPGA-Baustein eingehalten werden müssen. In diesem Abschnitt wird anhand von Auszügen der Zeit-Analyse „Static-Time-Report“ gezeigt, dass die im vorherigen Abschnitt festgelegten Zeitanforderungen nicht verletzt und somit eingehalten werden. Die Zeit-Analyse wird an dem FPGA-Baustein „Sparten-3“ der Firma XILINX durchgeführt, der eine maximale Taktfrequenz von 288 MHz haben kann. Der „Sparten-3“ wurde wegen seiner Anzahl der Pins, seines günstigen Preises und seiner Verfügbarkeit im Markt ausgewählt.

Der nachfolgende Auszug 9 repräsentiert das Ergebnis der Zeit-Analyse vom Modul „RTC_Sync“. In der Spalte „Constraint“ stehen die erwünschten Zeiten, die nicht verletzt sein dürfen. Die Signale „RX_DV“, „RX_D[3:0]“ und „RX_ER“ sind in einer Gruppe, die als „RX_Signale“ bezeichnet werden, zusammengefasst. Die Signale „TX_EN“, „TX_D[3:0]“ und „TX_ER“ sind ebenso zu einer Gruppe, die als „TX_Signale“ bezeichnet werden, zusammengefasst. In den Zeilen 5 und 6 ist die „Pad-to-Setup“ durch die Anweisung „OFFSET = IN 10 ns BEFOR“ definiert. Damit wird impliziert, dass der FPGA-Baustein eine „Setup-Time“ von nicht mehr als 10 ns vor dem aktiven Takt-Signal „RX_CLK“ an die Signal-Gruppe „RX_Signale“ fordert. In der Spalte „Best Case Achievable“ werden die erforderlichen Zeitanforderungen bzw. „Setup-Time“ seitens des FPGA-Bausteins angegeben. Die geforderten Werte liegen zwischen 6.165ns und 3.835ns, welche somit weniger als 10ns betragen. In den Zeilen 8 und 9 ist die „Clock-to-Pad“ durch die Anweisung „OFFSET = OUT 25 ns AFTER“ definiert. Dies impliziert, dass der gültige Wert der Signal-Gruppe „TX_Signale“ mit einer maximalen Verzögerung von 25 ns nach dem aktiven Takt-Signal „TX_CLK“ an die PINs des FPGA-Bausteins eintreffen darf. Die Spalte „Best Case Achievable“ zeigt die Verzögerungszeit des FPGA-Bausteins, die bei 6.424ns liegt und damit die Bedingung erfüllt. In den Zeilen 11 und 14 stehen die längsten „Clock-to-Setup“-Zeiten der Takt-Signale „RX_CLK“ und „TX_CLK“. Bei dem Takt-Signal „RX_CLK“ liegt die längste „Clock-to-Setup“-Zeit bei 5.652ns. Die längste „Clock-to-Setup“-Zeit des Takt-Signals „TX_CLK“ liegt bei 4.473 ns. Da der FPGA-Baustein durch die beiden Takt-Signale „RX_CLK“ und „TX_CLK“ gesteuert wird, wird die längste Zeit der beiden „Clock-to-Setup“ zur Festlegung der Taktfrequenz des FPGA-Bausteins herangezogen. In den Zeilen 18 bis 20 findet man eine Zusammenfassung der Tabelle. In Zeile 18 steht die Taktfrequenz des FPGA-Bausteins 176.929MHz, Zeile 19 stellt die „Setup-Time“ des FPGA-Bausteins dar und Zeile 20 repräsentiert die Verzögerungszeit der Daten nach dem Steuer-Takt-Signal. Die Spalte „Worst Case Slack“ ist irrelevant für den hier durchgeführten Beweis. Für weitere Informationen schauen Sie sich bitte „Development System Reference Guide“ S.230 [37] an. Der vollständige Bericht der Zeit-Analyse ist in den Dateien „RTC_Sync.par“ „RTC_Sync.twr“ nachzuvollziehen, welche auf der CD sich befinden.

```
1  ------------------------------------------------------------------------------------------------------
2  Constraint                                |  Check  | Worst Case |  Best Case | Timing |   Timing
3                                            |         |    Slack   | Achievable | Errors |    Score
4  ------------------------------------------------------------------------------------------------------
5  TIMEGRP "RX_Signale" OFFSET = IN 10 ns BE | SETUP   |     6.165ns|     3.835ns|       0|           0
6  FORE COMP "RX_CLK" HIGH                   |         |            |            |        |
7  ------------------------------------------------------------------------------------------------------
8  TIMEGRP "TX_Signale" OFFSET = OUT 25 ns A | MAXDELAY|    18.576ns|     6.424ns|       0|           0
9  FTER COMP "TX_CLK" HIGH                   |         |            |            |        |
10 ------------------------------------------------------------------------------------------------------
11 TS_RX_CLK = PERIOD TIMEGRP "RX_CLK" 40 ns | SETUP   |    34.348ns|     5.652ns|       0|           0
12  HIGH 50%                                 | HOLD    |     0.844ns|            |       0|           0
13 ------------------------------------------------------------------------------------------------------
14 TS_TX_CLK = PERIOD TIMEGRP "TX_CLK" 40 ns | SETUP   |    35.527ns|     4.473ns|       0|           0
15  HIGH 50%                                 | HOLD    |     0.848ns|            |       0|           0
16 ------------------------------------------------------------------------------------------------------
17
18  Minimum period:   5.652ns   (Maximum frequency: 176.929MHz)
19  Minimum input required time before clock:   3.835ns
20  Minimum output required time after clock:   6.424ns
```

Auszug 9: Zeit-Analyse für das Modul "RTC_Sync"

Die Ergebnisse der Zeit-Analyse des Moduls „RTC_Sync_PA“ sind im nachfolgenden Auszug 10 ersichtlich. In der Zeile 18 ist die minimale Periode 4.410ns angegeben, durch welche die Taktfrequenz des FPGA-Bausteins festgelegt wird. Die minimale „Setup-Time“ und minimale „Delay-Time “, die seitens des FPGA-Bausteins gefordert sind und in den Zeilen 19 und 20 aufgeführt sind, entsprechen dem Zeitkriterium von 10ns bzw. 25ns. Das vollständige Ergebnis der Zeit-Analyse ist den Dateien „RTC_Sync_PA.par“ „RTC_Sync_PA.twr“ zu entnehmen.

```
1  ------------------------------------------------------------------------------------------------------
2  Constraint                                |  Check  | Worst Case |  Best Case | Timing |   Timing
3                                            |         |    Slack   | Achievable | Errors |    Score
4  ------------------------------------------------------------------------------------------------------
5  TIMEGRP "RX_Signale" OFFSET = IN 10 ns BE | SETUP   |     7.210ns|     2.790ns|       0|           0
6  FORE COMP "TX_CLK" HIGH                   |         |            |            |        |
7  ------------------------------------------------------------------------------------------------------
8  TIMEGRP "TX_Signale" OFFSET = OUT 25 ns A | MAXDELAY|    15.770ns|     9.230ns|       0|           0
9  FTER COMP "TX_CLK" HIGH                   |         |            |            |        |
10 ------------------------------------------------------------------------------------------------------
11 TS_TX_CLK = PERIOD TIMEGRP "TX_CLK" 40 ns | SETUP   |    35.590ns|     4.410ns|       0|           0
12  HIGH 50%                                 | HOLD    |     0.850ns|            |       0|           0
13 ------------------------------------------------------------------------------------------------------
14 TS_RX_CLK = PERIOD TIMEGRP "RX_CLK" 40 ns | SETUP   |    36.658ns|     3.342ns|       0|           0
15  HIGH 50%                                 | HOLD    |     0.803ns|            |       0|           0
16 ------------------------------------------------------------------------------------------------------
17
18  Minimum period:   4.410ns   (Maximum frequency: 226.757MHz)
19  Minimum input required time before clock:   2.790ns
20  Minimum output required time after clock:   9.230ns
```

Auszug 10: Zeit-Analyse für das Modul "RTC_Sync_PA"

Das Modul „RTC_CORE“ erfüllt die Zeitanforderungen der MII-Schnittstelle. Die Ergebnisse der Zeit-Analyse sind im Auszug 11 ersichtlich. Die „Best Case Achievable“ Zeit 0.086ns für „Setup-Time“ überschreitet nicht die geförderte Zeit von 10ns. Die Daten werden mit der steigenden Flanke des Signals „TX_CLK“ nach maximal 7.391ns an den PINs des FPGA-Bausteins liegen. Dies überschreitet nicht die geförderte Zeit von 25ns. Eine Zusammenfassung der erreichten Zeit ist in den Zeilen 30-32 dargestellt.

```
------------------------------------------------------------------------------------------------------------------------
  Constraint                                |  Check  | Worst Case |  Best Case | Timing |   Timing
                                            |         |    Slack   | Achievable | Errors |    Score
------------------------------------------------------------------------------------------------------------------------
  TIMEGRP "RX_Signale_D" OFFSET = IN 10 ns  | SETUP   |     9.914ns|     0.086ns|       0|           0
  BEFORE COMP "MIID_RX_CLK" HIGH            |         |            |            |        |
------------------------------------------------------------------------------------------------------------------------
  TIMEGRP "RX_Signale_A" OFFSET = IN 10 ns  | SETUP   |    10.085ns|    -0.085ns|       0|           0
  BEFORE COMP "MIIA_RX_CLK" HIGH            |         |            |            |        |
------------------------------------------------------------------------------------------------------------------------
  TIMEGRP "RX_Signale_B" OFFSET = IN 10 ns  | SETUP   |    10.202ns|    -0.202ns|       0|           0
  BEFORE COMP "MIIB_RX_CLK" HIGH            |         |            |            |        |
------------------------------------------------------------------------------------------------------------------------
  TIMEGRP "RX_Signale_C" OFFSET = IN 10 ns  | SETUP   |    10.617ns|    -0.617ns|       0|           0
  BEFORE COMP "MIIC_RX_CLK" HIGH            |         |            |            |        |
------------------------------------------------------------------------------------------------------------------------
  TIMEGRP "TX_Signale_D" OFFSET = OUT 25 ns | MAXDELAY|    17.609ns|     7.391ns|       0|           0
   AFTER COMP "MIID_TX_CLK" HIGH            |         |            |            |        |
------------------------------------------------------------------------------------------------------------------------
  TIMEGRP "TX_Signale_C" OFFSET = OUT 25 ns | MAXDELAY|    17.816ns|     7.184ns|       0|           0
   AFTER COMP "MIIC_TX_CLK" HIGH            |         |            |            |        |
------------------------------------------------------------------------------------------------------------------------
  TIMEGRP "TX_Signale_A" OFFSET = OUT 25 ns | MAXDELAY|    17.823ns|     7.177ns|       0|           0
   AFTER COMP "MIIA_TX_CLK" HIGH            |         |            |            |        |
------------------------------------------------------------------------------------------------------------------------
  TIMEGRP "TX_Signale_B" OFFSET = OUT 25 ns | MAXDELAY|    17.872ns|     7.128ns|       0|           0
   AFTER COMP "MIIB_TX_CLK" HIGH            |         |            |            |        |
------------------------------------------------------------------------------------------------------------------------
Design statistics:
   Minimum period:   3.401ns   (Maximum frequency: 294.031MHz)
   Minimum input required time before clock:   0.086ns
   Minimum output required time after clock:   7.391ns
```

Auszug 11: Zeit-Analyse für das Modul "RTC_CORE"

Die Zeit-Analyse wurde für das Modul „RTC_CORE“ aufgrund des Skew-Problems des Synchronisations-Signals „Sync_CLK“ mittels des FPGA-Bausteins „Virtex5-XC55VLX30“ durchgeführt.

Anhand der in diesem Abschnitt durchgeführten Zeit-Analysen wurde gezeigt, dass die Module nicht die geforderten Zeitanforderungen „Setup-Time“ und „Delay-Time“ verletzten. Bei der Daten-Übertragung findet eine zeitliche Verzögerung von einer Taktperiode statt, die bei Ethernet-Variante 100 MBit/s 40ns beträgt. Bei dieser Verzögerung wird sichergestellt, dass die erste empfangene Nibble im Speicher liegt, wodurch vermieden wird, ungültige Daten auszulesen, die zu fehlerhaften Übertragungen führen. Bei einer Phasenverschiebung zwischen den Takt-Signalen „RX_CLK“ und „TX_CLK“ kann eine weitere 40ns Verzögerung eintreten. Im Fall von unvollständiger Präambel entstehen keine weiteren Verzögerungen. Dies heißt, dass nach den ersten empfangenen Nibble eine maximale Verzögerung von 80ns entstehen kann, bis die Nibble weitergeleitet wird.

5.8 Entwurf der RTC-Test-Platine

In diesem Abschnitt wird ein Blockschaltplan der „RTC-Test-Platine“ vorgestellt. Zum einen hat die Platine den Zweck die entworfenen VHDL-Module „RTC_Sync“ und „RTC_Sync_PA“ zu validieren. Zum anderen wird dadurch praktisch untersucht, ob keine weiteren Probleme neben den zwei Hauptproblemen der Synchronisation der Takt-Signale und der Vervollständigung der fehlenden Präambel auftreten. Des Weiteren gibt der Entwurf der RTC-Test-Platine die Möglichkeit, eine Umschaltung zwischen vier MII-Schnittstellen durchzuführen. Jedoch ist dieser Entwurf für den Prototyp des Echtzeit-Switchs ungeeignet. Zu Validierung des Moduls „RTC_COR“ muss ein schneller FPGA-Baustein (Beispiel: Sparten-3E oder Virtex-5) verwendet werden. Die nachfolgende Abbildung 75 repräsentiert die Komponenten der RTC-Test-Platine.

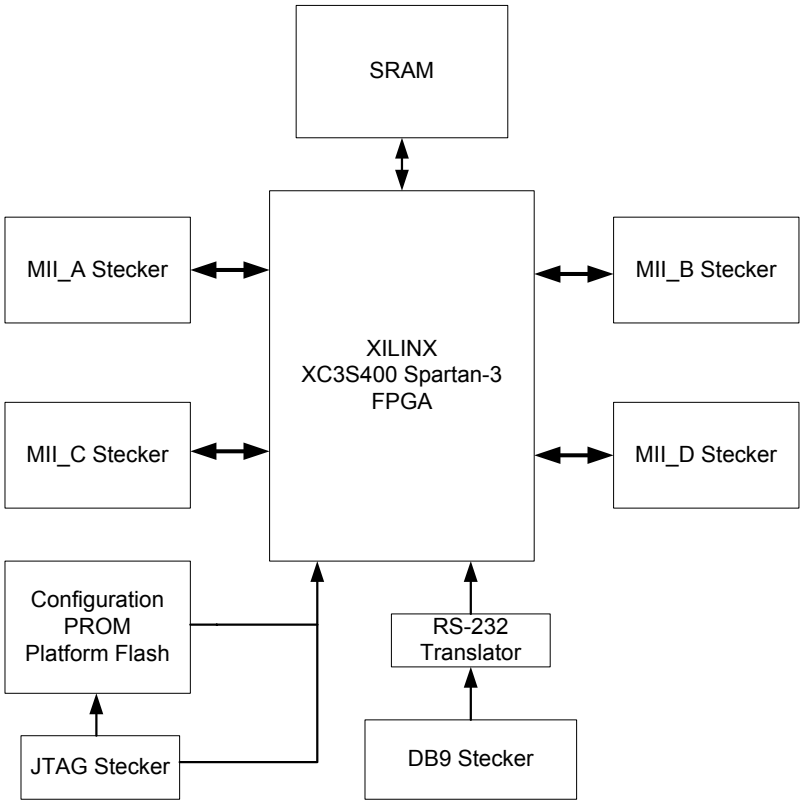

Abbildung 75: RTC-Test-Platine

Da bereits die Synthese, Implementierung und die Zeit-Analyse der entworfenen Module auf den FPGA-Baustein „XC3S400 Sparten-3“ der Firma Xilinx durchgeführt wurden, wird dieser Baustein auch für dieser Platine eingesetzt. Der FPGA-Baustein ist mit vier MII-Steckern über die MII-Signale verbunden. Für zusätzlichen Speicher ist die Platine mit einem externen SDRAM Speicher ausgerüstet. Da der FPGA-Baustein seine Konfiguration bzw. seine Programmierung nach dem Ausschalten verliert, wird die Konfiguration des FPGA-Bausteins in einem „Platform Flash PROM“ gespeichert. Der FPGA-Baustein erhält seine Konfiguration beim Neustart von dem Flash-PROM. Die Konfiguration des FPGA-Bausteins findet über die Schnittstelle JTAG statt. Eine weitere Möglichkeit den FPGA-Baustein zu konfigurieren, besteht über die RS-232 Schnittstelle.

5.9 Entwurf des Ethernet-Switches

Im Rahmen dieser Arbeit wird kein vollständiger Entwurf eines Echtzeit-Switchs aufgrund begrenzter Zeitdauer der Diplomarbeit erstellt. Es wird stattdessen ein Blockschaltplan vorgestellt, der als Basis zum vollständigen Entwurf dienen kann. Dies ist anhand der Abbildung 76 ersichtlich.

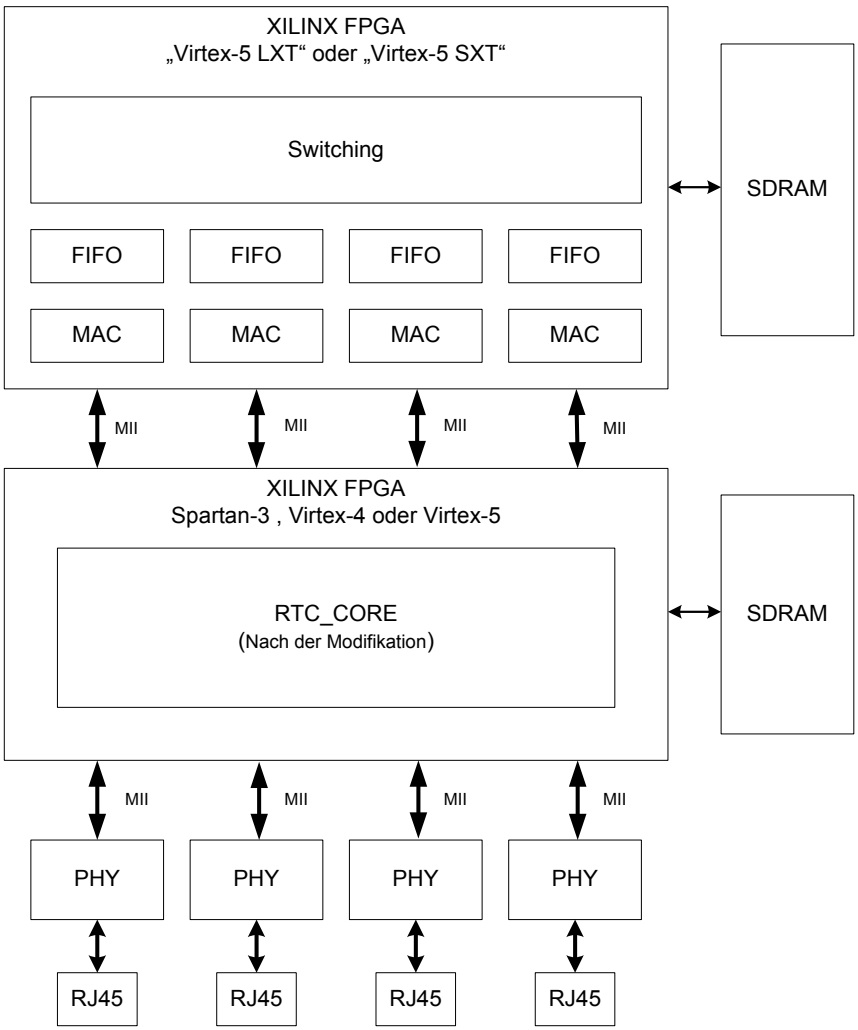

Abbildung 76: Blockschaltplan von Echtzeit-Switch

Das Modul „RTC_CORE“ muss modifiziert werden, um die Übertragungen zum MAC weiterzuleiten. Dafür müssen die Anschlüsse des Moduls modifiziert werden. Des Weiteren muss der Kodierungs-Mechanismus der Umschaltung zwischen den MII-Schnittstellen und der MACs neu entwickelt werden. Die MACs sollen auf Empfindlichkeit des CRS-Signals untersucht werden, oder müssen über das TxEN-Siganl gesteuert werden. Des Weiteren muss die Synchronisations-Signal „Sync_CLK“ spezifiziert werden. Hierbei muss die Funktion genauer beschrieben werden. Das zu modifizierende Modul „RTC_CORE“ kann auf einem FPGA-Baustein beispielsweise (Sparten-3, Virtex-4 oder Virtex-5) implementiert werden. Der FPGA-Baustein, an dem das Modul „RTC_CORE“ implementiert wird, ist mit vier PHYs verbunden. Jeder PHY ist

mit einem RJ45-Stecker verbunden. Des Weiteren kann ein SDRAM-Speicher mit dem FPGA-Baustein verbunden werden, um mehr Speicher zur Verfügung zu stellen.

Die Switching-Architektur, die aus den MACs, FIFOs und eine Switching-Fabric besteht, kann auf einem separaten FPGA-Baustein beispielsweise („Virtex-5 LXT“ oder „Virtex-5 SXT“) implementiert werden. Die genannten Modelle der FPGA-Baustein-Familie „Virtex-5“ der Firma Xilinx besitzen integrierte MACs und FIFOs, die durch IP-Cores (Intellectual Property) implementiert werden können. Diese Modelle haben jedoch den Nachteil, dass sie einen höheren Kauf-Preis haben und eine Voll-Version der Implementierungs-Software „ISE“ benötigen. Für die zusätzliche Speicher-Kapazität kann ein SDRAM-Speicher mit dem FPGA-Baustein verbunden werden.

Die beiden FPGA-Bausteine können über die MII-Schnittstelle miteinander verbunden werden. Das Modul „RTC_CORE“ und die Switching-Architektur können auf einem FPGA-Baustein implementiert werden, jedoch muss diese Realisierung noch untersucht werden.

Dieser Entwurf des Echtzeit-Switchs muss noch auf mögliche Realisierung untersucht werden. Dafür muss eine Markt-Übersicht anderer verfügbarer FPGA-Bausteine im Hinblick des Preises und der möglichen Realisierbarkeit durchgeführt werden.

6 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde der Umschaltungsmechanismus RTC zur Erstellung eines Entwurfs für Echtzeit-Switch auf mögliche Realisierung untersucht. Dafür musste ein Experiment durchgeführt werden. Es wurde eine Platine hergestellt, die eine direkte Verbindung zwischen zwei PHYs an der MII-Schnittstelle ermöglicht. Anhand der Experimente ergab sich, dass eine Übertragung der Daten an der MII-Schnittstelle möglich ist. Jedoch bestehen Möglichkeiten von fehlerhaften Übertragungen. Die Übertragung der Daten an der MII-Schnittstelle wird von zwei Takt-Signalen gesteuert, die als Ausgänge betrieben werden. Dadurch kann keiner der Takt-Signale als Eingang verwendet werden, um die Übertragung steuern zu können. Eine solche Übertragung führt zu einer Phasenverschiebung der Takt-Signale und wird somit als asynchron bezeichnet. Die MII-Schnittstelle stellt Zeitanforderungen „Setup-Time“ und „Hold-Time“ an die Daten, um eine fehlerfreie Übertragung zu gewährleisten. Anhand der Waveform eines Logik-Analysers konnte festgestellt werden, dass die fehlerhaften Übertragungen auf die Verletzungen der Zeitanforderungen zurückzuführen sind. Ein weiteres Problem, das keinen Einfluss auf die Übertragung der Daten hat bzw. keine fehlerhafte Übertragung verursachte, ist die Unvollständigkeit der Präambel. Bei empfangenen Paketen können gewisse Teile der Präambel fehlen. Die Unvollständigkeit der Präambel kann bei Kaskaden-Übertragung der Pakete zu einem Fehler führen, da der SFD-Teil dadurch verloren gehen kann. Somit wird ein Paket ohne den SFD-Teil vom MAC nicht erkannt.

Lösungsansätze zur Behebung des Phasenverschiebungs-Problems und zur Regenerierung der Präambel wurden in der VHDL-Sprache für die Implementierung auf einem FPGA-Baustein realisiert. Das Modul „RTC_Sync“, das auf Basis eines FIFO entworfen ist, kann das Problem der Phasenverschiebung lösen. Das Modul „RTC_Sync_PA“ behandelt zum einen das Problem der Phasenverschiebung und zum anderen das Problem der Unvollständigkeit der Präambel. Der Umschaltmechanismus RTC wurde durch das Modul „RTC_CORE“, das vier „RTC_Sync_PA“ Module implementiert, realisiert. Die Funktion des Moduls „RTC_CORE“ besteht nur in der Umschaltung zwischen vier PHYs an der MII-Schnittstelle, dabei wird keine Umschaltung zum MAC ermöglicht. Sämtliche Module wurden anhand von Simulationen, Synthese und Zeit-Analysen verifiziert.

Eine Überprüfung der realisierten Module „RTC_Sync“ und „RTC_Sync_PA“ kann nach Erstellung der RTC-Test-Platine, die anhand des im Kapitel 5.8 vorgestellten Blockschaltplans realisiert werden kann, erfolgen. Dadurch können Erkenntnisse gewonnen werden, ob weitere Probleme zur Realisierung des Umschaltmechanismus RTC entstehen können. Der Entwurf des Blockschaltplans für den Echtzeit-Switch im Kapitel 5.9 muss noch untersucht werden. Einerseits muss die Funktion mehr im Hinblick der Kodierung des Umschaltungsmechanismus und Synchronisations-Signals „Sync_CLK“ spezifiziert werden. Zum anderen muss eine Markt-Übersicht über die verfügbaren FPGA-Bausteine im Hinblick des Preises und der möglichen Realisierung durchgeführt werden. Des Weiteren muss das Modul „RTC_CORE“ modifiziert werden, sodass eine mögliche Verbindung zum MAC realisiert wird.

Abbildungsverzeichnis

Auszügeverzeichnis

Code-Listing

Tabellenverzeichnis

Literaturverzeichnis

[01] Dopatka, Frank; Wismüller, Roland: “Design of Realtime Industrial-Ethernet Network Including Hot-Pluggable Asynchronous Devices”, IEEE International Symposium on Industrial Electronics (ISIE), 2007.

[02] Dopatka, Frank; Wismüller, Roland: “A Top-Down Approach for Realtime Industrial-Ethernet Networks using Edge-Coloring of Conflict-Multigraphs”, SPEEDAM International Symposium on Power Electronics, Electrical Drives, Automation and Motion, 2006.

[03] IEEE Std. 802.3-2002: “IEEE 802.3 Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications”, 2005, http://standards.ieee.org/getieee802.3.html.

[04] Westerholt, Henning: „Entwurf und Entwicklung eines Simulationsmodells für ein neuartiges Echtzeit-Ethernet System im Industrieeinsatz“, Diplomarbeit Universität Siegen, 2007.

[05] Wismüller, Roland: „Folien zur Vorlesung Rechnernetze II“, Universität Siegen,2005, http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/ws05/rn2/v04_4.pdf.

[06] Meyer, Matthias: „Folien zur Vorlesung Entwurf Digitaler Systeme“, Universität Stuttgart, http://www.ikr.uni-stuttgart.de/Content/Courses/Home/L_EDS/

[07] Tanenbaum, Andrew S. : „Computernetzwerke“, Pearson Studium , 2007

[08] Borowka, Petra: „Netzwerk Technologien“, mitp, 2002

[09] Riggert, Wolfgang: „Rechnernetze“, Fachbuchverlag Leipzig, 2005.

[10] Connectivity Knowledge Platform: „Ethernet (IEEE802.3)“, http://ckp.made-it.com/ieee8023.html

[11] PC OPEN.DE: „Einordnung Ethernet ins OSI-7-Schichten-Referenzmodell“, http://www.pcopen.de/netzwerke/netze/17.html“

[12] Hein, Mathias: „Switching-Technologie“, Franzis, 2002

[13] Perlman, Radia: „Bridges, router, switches und Internetworking-Protokolle“, Addison-Wesley, 2002

[14] Fairhurst, Godred: „Ethernet Repeaters and Hubs“, http://www.erg.abdn.ac.uk/users/gorry/course/lan-pages/hub.html

[15] Industrial Ethernet University: „Hubs Versus Switches“, IE 105, http://www.industrialethernetu.com/courses/105_2.htm

[16] M. Felser: “PROFINET – Isochronous Real-Time”, IFAC Summer School, Prage, 2005, http://www.felser.ch/download/7_PN_IRT_E02.pdf

[17] Wikipedia: „ Automatisierungstechnik “, http://de.wikipedia.org/wiki/Automatisierungstechnik

[18] Wikipedia: „Tastverhältnis“, http://de.wikipedia.org/wiki/Tastverh%C3%A4ltnis

[19] Teridian Semiconductor Corporation: “10/100 Ethernet PHYs”; 78Q2133 MicroPHY, 2006, http://www.teridian.com/products/product_detail.cfm?product_key=73

[20] Teridian Semiconductor Corporation: “78Q2123/78Q2133 MicroPHY Datasheet”; 78Q2133 MicroPHY, 2006; http://www.teridian.com

[21] Teridian Semiconductor Corporation: “MicroPHY MII Eval Board”; 78Q2133 MicroPHY, 2006; http://www.teridian.com

[22] Broadcom Corporation: „Unmanaged Integrated 10/100BASE-T/TX 5/6-Port Switch”; BCM5325; 2004, http://www.broadcom.com

[23] Linear Technology: “ Voltage Stabiliser- LT1086“, Spannungsregler, http://www.linear.com/pc/productDetail.jsp?navId=H0,C1,C1003,C1040,C1055,P1358

[24] Egnite Software GmbH: “Ethernut 2”, http://www.ethernut.de/en/hardware/enut2/index.html

[25] F. Kesel & R. Bartholomä: „Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGA“, Oldenbourg, 2006

[26] Wannemacher, Markus: “Das FPGA-Kochbuch”, MITP-Verlag, 1998, http://www.aufzu.de/FPGA/kochbuch

[27] ASIC-WORLD: “Interfacing Two Clock Domains”, 2007, http://www.asic-world.com/tidbits/clock_domain.html

[28] ASIC-WORLD: “VHDL Tutorial”, 2007, http://www.asic-world.com/vhdl/tutorial.html

[29] Xilinx Inc.: “FIFO and Virtex-4”, TechXclusives, 2004, http://www.xilinx.com

[30] Xilinx Inc.: “Constraints Guide - v9.1i”, 2007, http://www.xilinx.com.

[31] Xilinx Inc.: “XST User Guide – v9.1i”, 2007, http://www.xilinx.com.

[32] Xilinx Inc.: “ISE 9.1i Quick Start Tutorial”, 2007, http://www.xilinx.com.

[33] Xilinx Inc.: "Virtex-5 Embedded Tri-Mode Ethernet MAC User Guide“, User Guide UG194, 2007. http://www.xilinx.com.

[34] Xilinx Inc.: “Synthesis and Simulation Design Guide”, 2007, http://www.xilinx.com.

[35] Xilinx Inc.: “What are OFFSET Constraints“, White Paper_WP237, 2006, http://www.xilinx.com.

[36] Xilinx Inc.: “Spartan-3 FPGA Family”, Data Sheet DS099, 2007, http://www.xilinx.com.

[37] Xilinx Inc.: “Development System Reference Guide”, 2007, http://www.xilinx.com.

Anhnag A. Beschreibung der beigefügten CD

Alle Ergebnisse dieser Arbeit befinden sich auf der beigefügten CD. Im Folgenden wird kurz der Inhalt des einzelnen Verzeichnisses erläutert.

Das Verzeichnis „FPGA“ enthält die Source-Codes der programmierten Module. Die Module wurden mit der Software „Xilinx ISE 9.1i“ entwickelt. Ein Tutorial zur Software kann die Datei „04_ISE 9.1i_Quick Start Tutorial.pdf“ im Verzeichnis „Xilinx Documents“ dienen.

Die Datei „RTC_Sync.ise“ ist das Projekt-Datei für das Modul „RTC_Sync“, das durch die Datei „RTC_Sync.vhd“ realisiert wird.

Die Datei „RTC_Sync_PA.ise“ ist das Projekt-Datei für das Modul „RTC_Sync“, das durch die Datei „RTC_Sync_PA.vhd“ realisiert wird.

Die Datei „RTC_CORE_V5.ise“ ist das Projekt-Datei für das Modul „RTC_Sync“, das durch die Datei „RTC_CORE“ realisiert wird. In diesem Projekt erfolgen die Synthese und die Zeit-Analyse für das Modul auf dem Baustein „Virtex-5“.

Die Dateien „RTC_Sync.syr“, „RTC_Sync_PA.syr“ und „RTC_CORE“ entsprechen die Synthese-Log-Datei. Die Dateien „RTC_Sync.par“ „RTC_Syn_PA.par“, „RTC_CORE.par“, „RTC_Sync.twr“, „RTC_Syn_PA.twr“, „RTC_CORE.twr“ entsprechen die Zeit-Analyse-Dateien.

Das Verzeichnis „Xilinx Documents“ enthält alle Daten-Sheets und Dokuments der Firma Xilinx.

Das Verzeichnis „Ausarbeitung“ enthält alle Daten der Diplomarbeit.

Anhnag B. VHDL-Source-Code

Im Folgend werden die Source-Codes des programmierten VHDL-Module angehangt. Des Weiteren werden Auszüge der Synthese und die Zeit-Analyse des Moduls „RTC_CORE“ auf dem FPGA-Bausteine „Virtex-5“. Zunächst folgt eine Listung der Inhalt dieses Anhangs:

- Source-Code für Modul “RTC_Sync”
- Source-Code für Testbench “RTC_Sync_TB”
- Source-Code für Modul “RTC_Sync_PA
- Source-Code für Testbench “RTC_Sync_PA_TB”
- Source-Code für Modul “RTC_CORE”
- Source-Code für Testbench “RTC_CORE_TB”

```
1  ----------------------------------------------------------------------------
2  library IEEE;
3      use IEEE.STD_LOGIC_1164.ALL;
4      use IEEE.STD_LOGIC_ARITH.ALL;
5      use IEEE.STD_LOGIC_UNSIGNED.ALL;
6  ----------------------------------------------------------------------------
7  entity RTC_Sync is
8      generic(
9          RAM_ADDR_WIDTH : integer := 4
10     );
11     port(
12         RX_CLK  : in std_logic;
13         RX_DV   : in std_logic;
14         RX_DATA : in std_logic_vector(3 downto 0);
15         RX_ER   : in std_logic;
16         TX_CLK  : in std_logic;
17         TX_EN   : out std_logic;
18         TX_DATA : out std_logic_vector(3 downto 0);
19         TX_ER   : out std_logic;
20         RESET   : in std_logic
21     );
22 end RTC_Sync;
23
24 architecture Behavioral of RTC_Sync is
25     ----------------------------------------------------------------------------
26     constant initRAM : std_logic_vector(0 to 2**RAM_ADDR_WIDTH-1) := (others => '0'
   );
27     type ram_type is array (0 to 2**RAM_ADDR_WIDTH-1) of std_logic_vector(5 downto
   0);
28     signal mem : ram_type := (others => initRAM);
29     ----------------------------------------------------------------------------
30     type rx_state_type is (rx_hold, rx_write);
31     type tx_state_type is (tx_hold, tx_read);
32     signal rx_state : rx_state_type;
33     signal tx_state : tx_state_type;
34     ----------------------------------------------------------------------------
35     signal RX_BUS   : std_logic_vector(5 downto 0) := (others => '0');
36     signal TX_BUS   : std_logic_vector(5 downto 0) := (others => '0');
37     signal RX_ADDR  : std_logic_vector(RAM_ADDR_WIDTH-1 downto 0) := (others => '0'
   );
38     signal TX_ADDR  : std_logic_vector(RAM_ADDR_WIDTH-1 downto 0) := (others => '0'
   );
39 begin
40     ----------------------------------------------------------------------------
41     RX_BUS(5) <= RX_ER;
42     RX_BUS(4) <= RX_DV;
43     RX_BUS(3 downto 0) <= RX_DATA;
44     TX_ER <= TX_BUS(5);
45     TX_EN <= TX_BUS(4);
46     TX_DATA <= TX_BUS(3 downto 0);
47     ----------------------------------------------------------------------------
48     -- RX_PROCESS ------------------------------------------------------------
49     ----------------------------------------------------------------------------
50     RX_Process : process(RX_CLK, RESET)
51     begin
52         if (RESET = '1') then
53             RX_ADDR <= RX_ADDR - RX_ADDR;
```

```
54              rx_state <= rx_hold;
55          elsif rising_edge(RX_CLK) then
56              case rx_state is
57                  ---------------------------------------
58                  when rx_hold =>
59                      if (RX_DV = '1') then
60                          mem(conv_integer(RX_ADDR)) <= RX_BUS;
61                          RX_ADDR <= RX_ADDR + '1' after 20 ns;
62                          rx_state <= rx_write;
63                      else
64                          rx_state <= rx_hold;
65                      end if;
66                  ---------------------------------------
67                  when rx_write =>
68                      if (RX_DV = '0') then
69                          mem(conv_integer(RX_ADDR)) <= RX_BUS;
70                          RX_ADDR <= RX_ADDR - RX_ADDR after 20 ns;
71                          rx_state <= rx_hold;
72                      else
73                          mem(conv_integer(RX_ADDR)) <= RX_BUS;
74                          RX_ADDR <= RX_ADDR + '1' after 20 ns;
75                          rx_state <= rx_write;
76                      end if;
77                  ---------------------------------------
78              end case;
79          end if;
80      end process RX_Process;
81      ------------------------------------------------------------------------
82      -- TX_PROCESS ----------------------------------------------------------
83      ------------------------------------------------------------------------
84      TX_Process : process(TX_CLK, RESET)
85          variable tmp : std_logic_vector(5 downto 0) := (others => '0');
86      begin
87          if RESET = '1' then
88              tmp := (others => '0');
89              TX_ADDR <= TX_ADDR - TX_ADDR;
90          elsif rising_edge(TX_CLK) then
91              case tx_state is
92                  ---------------------------------------
93                  when tx_hold =>
94                      if RX_DV = '1' then
95                          tx_state <= tx_read;
96                      else
97                          tx_state <= tx_hold;
98                      end if;
99                  ---------------------------------------
100                 when tx_read =>
101                     tmp := mem(conv_integer(TX_ADDR));
102                     if tmp(4) = '0' then
103                         tx_state <= tx_hold;
104                         TX_ADDR <= TX_ADDR - TX_ADDR;
105                     else
106                         tx_state <= tx_read;
107                         TX_ADDR <= TX_ADDR + '1';
108                     end if;
109                 ---------------------------------------
110             end case;
```

```
111             end if;
112             TX_BUS <= tmp;
113         end process TX_Process;
114         ----------------------------------------------------------------------------
115     end Behavioral;
```

```
1  --------------------------------------------------------------------------------
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  use IEEE.STD_LOGIC_ARITH.ALL;
5  use IEEE.STD_LOGIC_UNSIGNED.ALL;
6  --------------------------------------------------------------------------------
7  entity RTC_Sync_TB is
8  end RTC_Sync_TB;
9  --------------------------------------------------------------------------------
10 architecture Behavioral of RTC_Sync_TB is
11     signal RX_CLK   : std_logic := '0';
12     signal RX_DV    : std_logic := '0';
13     signal RX_DATA  : std_logic_vector(3 downto 0) := (others => '0');
14     signal RX_ER    : std_logic := '0';
15     signal TX_CLK   : std_logic := '0';
16     signal TX_EN    : std_logic := '0';
17     signal TX_DATA  : std_logic_vector(3 downto 0) := (others => '0');
18     signal TX_ER    : std_logic := '0';
19     signal RESET    : std_logic := '0';
20
21     constant PERIOD : time := 40 ns;
22     constant RX_DUTY_CYCLE : real := 0.5;
23     --SHARED variable RX_DUTY_CYCLE : real := 0.5;
24     constant TX_DUTY_CYCLE : real := 0.5;
25     --SHARED variable TX_DUTY_CYCLE : real := 0.4;
26     constant RX_OFFSET : time := 0 ns;
27     constant TX_OFFSET : time := 20 ns;
28
29     component RTC_Sync is
30     port(
31         RX_CLK  : in std_logic;
32         RX_DV   : in std_logic;
33         RX_DATA : in std_logic_vector(3 downto 0);
34         RX_ER   : in std_logic;
35         TX_CLK  : in std_logic;
36         TX_EN   : out std_logic;
37         TX_DATA : out std_logic_vector(3 downto 0);
38         TX_ER   : out std_logic;
39         RESET   : in std_logic
40     );
41     end component;
42
43 begin
44     UUT : RTC_Sync port map(
45         RX_CLK, RX_DV, RX_DATA, RX_ER, TX_CLK, TX_EN, TX_DATA, TX_ER, RESET
46     );
47     -----------------------------------------------------
48     CLK_RX : PROCESS
49         variable tmp : integer := 0;
50     BEGIN
51         WAIT for RX_OFFSET;
52         LOOP
53             RX_CLK <= '0';
54             WAIT FOR (PERIOD - (PERIOD * RX_DUTY_CYCLE));
55             RX_CLK <= '1';
56             WAIT FOR (PERIOD * RX_DUTY_CYCLE);
57             -------------------------------------------------
```

```
58  --              tmp := tmp + 1;
59  --              if tmp = 10 then
60  --                  RX_DUTY_CYCLE := 0.2;
61  --                  tmp := 0;
62  --              else
63  --                  RX_DUTY_CYCLE := 0.5;
64  --              end if;
65                  ----------------------------------------------------
66              END LOOP;
67          END PROCESS CLK_RX;
68          ----------------------------------------------------------
69          CLK_TX : process
70              variable tmp : integer := 0;
71          begin
72              wait for TX_OFFSET;
73              LOOP
74                  TX_CLK <= '0';
75                  wait for (PERIOD - (PERIOD * TX_DUTY_CYCLE));
76                  TX_CLK <= '1';
77                  WAIT FOR (PERIOD * TX_DUTY_CYCLE);
78                  ----------------------------------------------------
79  --              tmp := tmp + 1;
80  --              if tmp = 10 then
81  --                  TX_DUTY_CYCLE := 0.5;
82  --                  tmp := 0;
83  --              else
84  --                  TX_DUTY_CYCLE := 0.8;
85  --              end if;
86                  ----------------------------------------------------
87              END LOOP;
88          end process CLK_TX;
89          ----------------------------------------------------------
90          FRAME : process
91              variable tmp : std_logic_vector(3 downto 0) := (others => '0');
92          begin
93              wait for 2*PERIOD;
94              wait until RX_CLK = '0';
95              loop
96                  ----------------------------------------------------
97                  -- Preamble nibbles
98                  for i in 0 to 9 loop
99                      wait for PERIOD;
100                     RX_DV       <= '1';
101                     RX_DATA <= "0101";
102                 end loop;
103                 ----------------------------------------------------
104                 -- SFD nibble
105                 wait for PERIOD;
106                 RX_DV   <= '1';
107                 RX_DATA <= "1101";
108                 ----------------------------------------------------
109                 -- Daten nibbles
110                 for i in 0 to 9 loop
111                     wait for PERIOD;
112                     RX_DV   <= '1';
113                     RX_DATA <= tmp;
114                 end loop;
```

```
115                 tmp := tmp + 1;
116                 ----------------------------------------------------
117                 -- EFD nibble
118                 wait for PERIOD;
119                 RX_DV   <= '1';
120                 RX_DATA <= "1111";
121                 ----------------------------------------------------
122                 -- IFS nibbles
123                 for i in 0 to 9 loop
124                     wait for PERIOD;
125                     RX_DV   <= '0';
126                     RX_DATA <= (others => '0');
127                 end loop;
128                 ----------------------------------------------------
129             end loop;
130         end process FRAME;
131         ------------------------------------------------------
132         Wait_Daten : process
133         begin
134             wait for 30 us;
135             ASSERT (FALSE) REPORT
136             "Ende der Simulation."
137             SEVERITY FAILURE;
138         end process;
139         ------------------------------------------------------
140     end Behavioral;
```

```
1  ----------------------------------------------------------------------------
2  library IEEE;
3      use IEEE.STD_LOGIC_1164.ALL;
4      use IEEE.STD_LOGIC_ARITH.ALL;
5      use IEEE.STD_LOGIC_UNSIGNED.ALL;
6  ----------------------------------------------------------------------------
7  entity RTC_Sync_PA is
8      generic(
9          RAM_ADDR_WIDTH : integer := 4
10     );
11     port(
12         RX_CLK  : in std_logic;
13         RX_DV   : in std_logic;
14         RX_DATA : in std_logic_vector(3 downto 0);
15         RX_ER   : in std_logic;
16
17         TX_CLK  : in std_logic;
18         TX_EN   : out std_logic;
19         TX_DATA : out std_logic_vector(3 downto 0);
20         TX_ER   : out std_logic;
21         RESET   : in std_logic
22     );
23 end RTC_Sync_PA;
24
25 architecture Behavioral of RTC_Sync_PA is
26     ----------------------------------------------------------------------------
27     type rom_type is array (0 to 15) of std_logic_vector(5 downto 0);
28     constant ROM : rom_type :=( --[TX_ER, TX_EN, TX[3:0]]-- Belegung des ROMs
29                                              0 => "010101",
30                                              1 => "010101",
31                                              2 => "010101",
32                                              3 => "010101",
33                                              4 => "010101",
34                                              5 => "010101",
35                                              6 => "010101",
36                                              7 => "010101",
37                                              8 => "010101",
38                                              9 => "010101",
39                                             10 => "010101",
40                                             11 => "010101",
41                                             12 => "010101",
42                                             13 => "010101",
43                                             14 => "010101",
44                                             15 => "011101"); -- zweite Nibble des SFDs
45     ----------------------------------------------------------------------------
46     type ram_type is array (0 to 2**RAM_ADDR_WIDTH-1) of std_logic_vector(5 downto
   0);
47     signal RAM : ram_type;
48     ----------------------------------------------------------------------------
49     type rx_state_type is (rx_hold, search_sfd, write_ram);
50     type tx_state_type is (tx_hold, read_rom, read_ram);
51     signal rx_state : rx_state_type;
52     signal tx_state : tx_state_type;
53     ----------------------------------------------------------------------------
54     signal RX_BUS : std_logic_vector(5 downto 0) := (others => '0');
55     signal TX_BUS : std_logic_vector(5 downto 0) := (others => '0');
56     signal ROM_RA : std_logic_vector(3 downto 0) := (others => '0');
```

```
57      signal RAM_WA : std_logic_vector(RAM_ADDR_WIDTH-1 downto 0) := (others => '0');
58      signal RAM_RA : std_logic_vector(RAM_ADDR_WIDTH-1 downto 0) := (others => '0');
59  begin
60      ------------------------------------------------------------------------
61      RX_BUS(5) <= RX_ER;
62      RX_BUS(4) <= RX_DV;
63      RX_BUS(3 downto 0) <= RX_DATA;
64      TX_ER <= TX_BUS(5);
65      TX_EN <= TX_BUS(4);
66      TX_DATA <= TX_BUS(3 downto 0);
67      ------------------------------------------------------------------------
68      -- TX_PROCESS ----------------------------------------------------------
69      ------------------------------------------------------------------------
70      RX_Process : process(RX_CLK, RESET, RX_DV)
71      begin
72          if RESET = '1' then
73              RAM_WA <= RAM_WA - RAM_WA;
74              rx_state <= rx_hold;
75          elsif rising_edge(RX_CLK) then
76              case rx_state is
77                  ----------------------------------------
78                  when rx_hold =>
79                      if RX_DV = '1' then
80                          rx_state <= search_sfd;
81                      else
82                          rx_state <= rx_hold;
83                      end if;
84                  ----------------------------------------
85                  when search_sfd =>
86                      if RX_DATA ="1101" then
87                          rx_state <= write_ram;
88                      else
89                          rx_state <= search_sfd;
90                      end if;
91                  ----------------------------------------
92                  when write_ram =>
93                      RAM(conv_integer(ram_wa)) <= RX_BUS;
94                      if RX_DV = '0' then
95                          RAM_WA <= RAM_WA - RAM_WA after 20 ns;
96                          rx_state <= rx_hold;
97                      else
98                          RAM_WA <= RAM_WA + '1' after 20 ns;
99                          rx_state <= write_ram;
100                     end if;
101                 ----------------------------------------
102                 when others =>
103                     rx_state <= rx_hold;
104                 ----------------------------------------
105             end case;
106         end if;
107     end process RX_Process;
108     ------------------------------------------------------------------------
109     -- TX_PROCESS ----------------------------------------------------------
110     ------------------------------------------------------------------------
111     TX_Process : process(TX_CLK, RESET, RX_DV)
112     begin
113         if RESET = '1' then
```

```
114             ROM_RA <= ROM_RA - ROM_RA;
115             RAM_RA <= RAM_RA - RAM_RA;
116             tx_state <= tx_hold;
117         elsif rising_edge(TX_CLK) then
118             case tx_state is
119                 ------------------------------------------
120                 when tx_hold =>
121                     if RX_DV = '1' then
122                         tx_state <= read_rom;
123                     else
124                         tx_state <= tx_hold;
125                     end if;
126                 ------------------------------------------
127                 when read_rom =>
128                     TX_BUS <= ROM(conv_integer(ROM_RA));
129                     if ROM_RA = "1111" then
130                         ROM_RA <= ROM_RA - ROM_RA;
131                         tx_state <= read_ram;
132                     else
133                         ROM_RA <= ROM_RA + '1';
134                         tx_state <= read_rom;
135                     end if;
136                 ------------------------------------------
137                 when read_ram =>
138                     if (TX_BUS(4) = '0') then
139                         RAM_RA <= RAM_RA - RAM_RA;
140                         tx_state <= tx_hold;
141                     else
142                         TX_BUS <= RAM(conv_integer(ram_ra));
143                         RAM_RA <= RAM_RA + '1';
144                         tx_state <= read_ram;
145                     end if;
146                 ------------------------------------------
147                 when others =>
148                     tx_state <= tx_hold;
149                 ------------------------------------------
150             end case;
151         end if;
152     end process TX_Process;
153     -------------------------------------------------------------------------
154 end Behavioral;
```

```
1  --------------------------------------------------------------------------------
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  use IEEE.STD_LOGIC_ARITH.ALL;
5  use IEEE.STD_LOGIC_UNSIGNED.ALL;
6  --------------------------------------------------------------------------------
7  entity RTC_Sync_PA_TB is
8  end RTC_Sync_PA_TB;
9  --------------------------------------------------------------------------------
10 architecture Behavioral of RTC_Sync_PA_TB is
11     signal RX_CLK   : std_logic := '0';
12     signal RX_DV    : std_logic := '0';
13     signal RX_DATA  : std_logic_vector(3 downto 0) := (others => '0');
14     signal RX_ER    : std_logic := '0';
15     signal TX_CLK   : std_logic := '0';
16     signal TX_EN    : std_logic := '0';
17     signal TX_DATA  : std_logic_vector(3 downto 0) := (others => '0');
18     signal TX_ER    : std_logic := '0';
19     signal RESET    : std_logic := '0';
20 
21     constant PERIOD : time := 40 ns;
22     constant RX_DUTY_CYCLE : real := 0.5;
23     constant TX_DUTY_CYCLE : real := 0.5;
24     constant RX_OFFSET : time := 0 ns;
25     constant TX_OFFSET : time := 0 ns;
26 
27     component RTC_Sync_PA is
28     port(
29         RX_CLK  : in std_logic;
30         RX_DV   : in std_logic;
31         RX_DATA : in std_logic_vector(3 downto 0);
32         RX_ER   : in std_logic;
33         TX_CLK  : in std_logic;
34         TX_EN   : out std_logic;
35         TX_DATA : out std_logic_vector(3 downto 0);
36         TX_ER   : out std_logic;
37         RESET   : in std_logic
38     );
39     end component;
40 
41 begin
42     UUT : RTC_Sync_PA port map(
43         RX_CLK, RX_DV, RX_DATA, RX_ER, TX_CLK, TX_EN, TX_DATA, TX_ER, RESET
44     );
45     -------------------------------------------------------
46     CLK_RX : PROCESS
47     BEGIN
48         WAIT for RX_OFFSET;
49         CLOCK_LOOP : LOOP
50             RX_CLK <= '0';
51             WAIT FOR (PERIOD - (PERIOD * RX_DUTY_CYCLE));
52             RX_CLK <= '1';
53             WAIT FOR (PERIOD * RX_DUTY_CYCLE);
54         END LOOP;
55     END PROCESS CLK_RX;
56     -------------------------------------------------------
57     CLK_TX : process
```

```
58      begin
59          wait for TX_OFFSET;
60          CLOCK_LOOP : LOOP
61              TX_CLK <= '0';
62              wait for (PERIOD - (PERIOD * TX_DUTY_CYCLE));
63              TX_CLK <= '1';
64              WAIT FOR (PERIOD * TX_DUTY_CYCLE);
65          END LOOP;
66      end process CLK_TX;
67      -------------------------------------------------------
68      Input_Data : process
69          variable tmp : std_logic_vector(3 downto 0) := (others => '0');
70      begin
71          wait for 2*PERIOD;
72          wait until RX_CLK = '0';
73          loop
74              -------------------------------------------------------
75              -- Preamble nibbles
76              for i in 0 to 9 loop
77                  wait for PERIOD;
78                  RX_DV       <= '1';
79                  RX_DATA <= "0101";
80              end loop;
81              -------------------------------------------------------
82              -- SFD nibble
83              wait for PERIOD;
84              RX_DV   <= '1';
85              RX_DATA <= "1101";
86              -------------------------------------------------------
87              -- Daten nibbles
88              for i in 0 to 9 loop
89                  wait for PERIOD;
90                  RX_DV   <= '1';
91                  RX_DATA <= tmp;
92              end loop;
93              tmp := tmp + 1;
94              -------------------------------------------------------
95              -- EFD nibble
96              wait for PERIOD;
97              RX_DV   <= '1';
98              RX_DATA <= "1111";
99              -------------------------------------------------------
100             -- IFS nibbles
101             for i in 0 to 9 loop
102                 wait for PERIOD;
103                 RX_DV   <= '0';
104                 RX_DATA <= (others => '0');
105             end loop;
106             -------------------------------------------------------
107         end loop;
108     end process Input_Data;
109     -------------------------------------------------------
110     Wait_Daten : process
111     begin
112         wait for 30 us;
113         ASSERT (FALSE) REPORT
114         "Ende der Simulation."
```

```
115             SEVERITY FAILURE;
116         end process;
117         -------------------------------------------------------
118     end Behavioral;
```

```
1  ----------------------------------------------------------------------
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  use IEEE.STD_LOGIC_ARITH.ALL;
5  use IEEE.STD_LOGIC_UNSIGNED.ALL;
6  ----------------------------------------------------------------------
7  entity RTC_CORE is
8      port(
9          ------------------------------------------------------------------
10         Sync_CLK            : out std_logic;
11         ------------------------------------------------------------------
12         MIIA_RX_CLK     : in std_logic;
13         MIIA_RX_DV      : in std_logic;
14         MIIA_RX_DATA    : in std_logic_vector(3 downto 0);
15         MIIA_RX_ER      : in std_logic;
16         MIIA_TX_CLK     : in std_logic;
17         MIIA_TX_EN      : out std_logic;
18         MIIA_TX_DATA    : out std_logic_vector(3 downto 0);
19         MIIA_TX_ER      : out std_logic;
20         ------------------------------------------------------------------
21         MIIB_RX_CLK     : in std_logic;
22         MIIB_RX_DV      : in std_logic;
23         MIIB_RX_DATA    : in std_logic_vector(3 downto 0);
24         MIIB_RX_ER      : in std_logic;
25         MIIB_TX_CLK     : in std_logic;
26         MIIB_TX_EN      : out std_logic;
27         MIIB_TX_DATA    : out std_logic_vector(3 downto 0);
28         MIIB_TX_ER      : out std_logic;
29         ------------------------------------------------------------------
30         MIIC_RX_CLK     : in std_logic;
31         MIIC_RX_DV      : in std_logic;
32         MIIC_RX_DATA    : in std_logic_vector(3 downto 0);
33         MIIC_RX_ER      : in std_logic;
34         MIIC_TX_CLK     : in std_logic;
35         MIIC_TX_EN      : out std_logic;
36         MIIC_TX_DATA    : out std_logic_vector(3 downto 0);
37         MIIC_TX_ER      : out std_logic;
38         ------------------------------------------------------------------
39         MIID_RX_CLK     : in std_logic;
40         MIID_RX_DV      : in std_logic;
41         MIID_RX_DATA    : in std_logic_vector(3 downto 0);
42         MIID_RX_ER      : in std_logic;
43         MIID_TX_CLK     : in std_logic;
44         MIID_TX_EN      : out std_logic;
45         MIID_TX_DATA    : out std_logic_vector(3 downto 0);
46         MIID_TX_ER      : out std_logic;
47         ------------------------------------------------------------------
48         RESET               : in std_logic
49         ------------------------------------------------------------------
50     );
51
52 end RTC_CORE;
53
54 architecture Behavioral of RTC_CORE is
55     type ram_type is array(0 to 7) of std_logic_vector(7 downto 0);
56     constant schedule_value : ram_type :=
57         (
```

```
58                "00100111", -- A, C, B, D
59                "01111000", -- B, D, C, A
60                "10001101", -- C, A, D, B
61                "11010010", -- D, B, A, C
62                "00100010", -- A, C, A, C
63                "01110010", -- B, D, A, C
64                "10111010", -- C, D, C, C
65                "11000011"  -- D, A, A, D
66            );
67        signal schedule_ad : std_logic_vector(2 downto 0) := (others => '0');
68        ----------------------------------------------------------------------
69    --  signal MIIA_src, MIIB_src, MIIC_src, MIID_src : std_logic_vector(1 downto 0);
70    --
71    --  type MIIA_state_type is (MIIA2MIIA, MIIB2MIIA, MIIC2MIIA, MIID2MIIA);
72    --  type MIIB_state_type is (MIIA2MIIB, MIIB2MIIB, MIIC2MIIB, MIID2MIIB);
73    --  type MIIC_state_type is (MIIA2MIIC, MIIB2MIIC, MIIC2MIIC, MIID2MIIC);
74    --  type MIID_state_type is (MIIA2MIID, MIIB2MIID, MIIC2MIID, MIID2MIID);
75    --  signal MIIA_state : MIIA_state_type; -- := MIIA2MIIA;
76    --  signal MIIB_state : MIIB_state_type; -- := MIIB2MIIB;
77    --  signal MIIC_state : MIIC_state_type; -- := MIIC2MIIC;
78    --  signal MIID_state : MIID_state_type; -- := MIID2MIID;
79        ----------------------------------------------------------------------
80        signal MIIA_RX_BUS : std_logic_vector(5 downto 0) := (others => '0');
81        signal MIIB_RX_BUS : std_logic_vector(5 downto 0) := (others => '0');
82        signal MIIC_RX_BUS : std_logic_vector(5 downto 0) := (others => '0');
83        signal MIID_RX_BUS : std_logic_vector(5 downto 0) := (others => '0');
84        ----------------------------------------------------------------------
85        signal A_RX_CLK : std_logic;
86        signal A_RX_BUS : std_logic_vector(5 downto 0) := (others => '0');
87        signal B_RX_CLK : std_logic;
88        signal B_RX_BUS : std_logic_vector(5 downto 0) := (others => '0');
89        signal C_RX_CLK : std_logic;
90        signal C_RX_BUS : std_logic_vector(5 downto 0) := (others => '0');
91        signal D_RX_CLK : std_logic;
92        signal D_RX_BUS : std_logic_vector(5 downto 0) := (others => '0');
93        ----------------------------------------------------------------------
94        signal CORE_CLK  : std_logic;
95        signal schedule : std_logic_vector(7 downto 0) := (others => '0');
96        ----------------------------------------------------------------------
97        attribute clock_signal : string;
98
99        attribute clock_signal of CORE_CLK : signal is "no";
100
101       attribute clock_signal of MIIA_TX_CLK : signal is "yes";
102       attribute clock_signal of MIIB_TX_CLK : signal is "yes";
103       attribute clock_signal of MIIC_TX_CLK : signal is "yes";
104       attribute clock_signal of MIID_TX_CLK : signal is "yes";
105
106       attribute clock_signal of MIIA_RX_CLK : signal is "yes";
107       attribute clock_signal of MIIB_RX_CLK : signal is "yes";
108       attribute clock_signal of MIIC_RX_CLK : signal is "yes";
109       attribute clock_signal of MIID_RX_CLK : signal is "yes";
110
111       attribute buffer_type : string;
112       attribute buffer_type of CORE_CLK : signal is "BUFG";
113
114       attribute buffer_type of A_RX_CLK : signal is "BUFG";
```

```
115     attribute buffer_type of B_RX_CLK : signal is "BUFG";
116     attribute buffer_type of C_RX_CLK : signal is "BUFG";
117     attribute buffer_type of D_RX_CLK : signal is "BUFG";
118
119     attribute buffer_type of MIIA_RX_CLK : signal is "IBUFG";
120     attribute buffer_type of MIIB_RX_CLK : signal is "IBUFG";
121     attribute buffer_type of MIIC_RX_CLK : signal is "IBUFG";
122     attribute buffer_type of MIID_RX_CLK : signal is "IBUFG";
123
124     attribute buffer_type of MIIA_TX_CLK : signal is "BUFGP";
125     attribute buffer_type of MIIB_TX_CLK : signal is "BUFGP";
126     attribute buffer_type of MIIC_TX_CLK : signal is "BUFGP";
127     attribute buffer_type of MIID_TX_CLK : signal is "BUFGP";
128     ------------------------------------------------------------------
129     component CORE_CLOCK_GEN
130     generic(
131         TIMESLOT : integer
132     );
133     port(
134         CLKIN : in std_logic;
135         Sync_CLK : out std_logic
136     );
137     end component;
138     ------------------------------------------------------------------
139     component RTC_Sync_PA
140     port(
141         RX_CLK  : in std_logic;
142         RX_DV       : in std_logic;
143         RX_DATA : in std_logic_vector(3 downto 0);
144         RX_ER       : in std_logic;
145         TX_CLK  : in std_logic;
146         TX_EN       : out std_logic;
147         TX_DATA : out std_logic_vector(3 downto 0);
148         TX_ER       : out std_logic;
149         RESET       : in std_logic
150     );
151     end component;
152     ------------------------------------------------------------------
153 begin
154     ------------------------------------------------------------------
155     CLOCK : CORE_CLOCK_GEN
156     generic map(35)
157     port map(MIIA_TX_CLK, CORE_CLK);
158     ------------------------------------------------------------------
159     MIIA_Sync : RTC_Sync_PA
160     port map(
161         A_RX_CLK,
162         A_RX_BUS(4),
163         A_RX_BUS(3 downto 0),
164         A_RX_BUS(5),
165         MIIA_TX_CLK,
166         MIIA_TX_EN,
167         MIIA_TX_DATA,
168         MIIA_TX_ER,
169         RESET
170     );
171     ------------------------------------------------------------------
```

```
172       MIIB_Sync : RTC_Sync_PA
173       port map(
174           B_RX_CLK,
175           B_RX_BUS(4),
176           B_RX_BUS(3 downto 0),
177           B_RX_BUS(5),
178           MIIB_TX_CLK,
179           MIIB_TX_EN,
180           MIIB_TX_DATA,
181           MIIB_TX_ER,
182           RESET
183       );
184       ----------------------------------------------------------------------
185       MIIC_Sync : RTC_Sync_PA
186       port map(
187           C_RX_CLK,
188           C_RX_BUS(4),
189           C_RX_BUS(3 downto 0),
190           C_RX_BUS(5),
191           MIIC_TX_CLK,
192           MIIC_TX_EN,
193           MIIC_TX_DATA,
194           MIIC_TX_ER,
195           RESET
196       );
197       ----------------------------------------------------------------------
198       MIID_Sync : RTC_Sync_PA
199       port map(
200           D_RX_CLK,
201           D_RX_BUS(4),
202           D_RX_BUS(3 downto 0),
203           D_RX_BUS(5),
204           MIID_TX_CLK,
205           MIID_TX_EN,
206           MIID_TX_DATA,
207           MIID_TX_ER,
208           RESET
209       );
210       ----------------------------------------------------------------------
211       ----------------------------------------------------------------------
212       Sync_CLK <= CORE_CLK;
213       ----------------------------------------------------------------------
214       MIIA_RX_BUS(4) <= MIIA_RX_DV;
215       MIIA_RX_BUS(3 downto 0) <= MIIA_RX_DATA;
216       MIIA_RX_BUS(5) <= MIIA_RX_ER;
217       ----------------------------------------------------------------------
218       MIIB_RX_BUS(4) <= MIIB_RX_DV;
219       MIIB_RX_BUS(3 downto 0) <= MIIB_RX_DATA;
220       MIIB_RX_BUS(5) <= MIIB_RX_ER;
221       ----------------------------------------------------------------------
222       MIIC_RX_BUS(4) <= MIIC_RX_DV;
223       MIIC_RX_BUS(3 downto 0) <= MIIC_RX_DATA;
224       MIIC_RX_BUS(5) <= MIIC_RX_ER;
225       ----------------------------------------------------------------------
226       MIID_RX_BUS(4) <= MIID_RX_DV;
227       MIID_RX_BUS(3 downto 0) <= MIID_RX_DATA;
228       MIID_RX_BUS(5) <= MIID_RX_ER;
```

```
229       ---------------------------------------------------------------------
230       ---------------------------------------------------------------------
231       process(CORE_CLK)
232       begin
233           if rising_edge(CORE_CLK) then
234               schedule <= schedule_value(conv_integer(schedule_ad));
235               schedule_ad <= schedule_ad + '1';
236           end if;
237       end process;
238       ---------------------------------------------------------------------
239       process(schedule, MIIB_RX_BUS, MIIC_RX_BUS, MIID_RX_BUS, MIIB_RX_CLK,
      MIIC_RX_CLK, MIID_RX_CLK)
240       begin
241           case schedule(7 downto 6) is
242               when "00" =>
243                   A_RX_CLK <= '0';
244                   A_RX_BUS <= (others => '0');
245                   --MIIA_state <= MIIA2MIIA;
246               when "01" =>
247                   A_RX_CLK <= MIIB_RX_CLK;
248                   A_RX_BUS <= MIIB_RX_BUS;
249                   --MIIA_state <= MIIB2MIIA;
250               when "10" =>
251                   A_RX_CLK <= MIIC_RX_CLK;
252                   A_RX_BUS <= MIIC_RX_BUS;
253                   --MIIA_state <= MIIC2MIIA;
254               when "11" =>
255                   A_RX_CLK <= MIID_RX_CLK;
256                   A_RX_BUS <= MIID_RX_BUS;
257                   --MIIA_state <= MIID2MIIA;
258               when others =>
259                   A_RX_CLK <= '0';
260                   A_RX_BUS <= (others => '0');
261                   --MIIA_state <= MIIA2MIIA;
262           end case;
263       end process;
264       ---------------------------------------------------------------------
265       process(schedule, MIIA_RX_BUS, MIIC_RX_BUS, MIID_RX_BUS, MIIA_RX_CLK,
      MIIC_RX_CLK, MIID_RX_CLK)
266       begin
267           case schedule(5 downto 4) is
268               when "00" =>
269                   B_RX_CLK <= MIIA_RX_CLK;
270                   B_RX_BUS <= MIIA_RX_BUS;
271                   --MIIB_state <= MIIA2MIIB;
272               when "01" =>
273                   B_RX_CLK <= '0';
274                   B_RX_BUS <= (others => '0');
275                   --MIIB_state <= MIIB2MIIB;
276               when "10" =>
277                   B_RX_CLK <= MIIC_RX_CLK;
278                   B_RX_BUS <= MIIC_RX_BUS;
279                   --MIIB_state <= MIIC2MIIB;
280               when "11" =>
281                   B_RX_CLK <= MIID_RX_CLK;
282                   B_RX_BUS <= MIID_RX_BUS;
283                   --MIIB_state <= MIID2MIIB;
```

```
284             when others =>
285                 B_RX_CLK <= '0';
286                 B_RX_BUS <= (others => '0');
287                 --MIIB_state <= MIIB2MIIB;
288         end case;
289     end process;
290     ---------------------------------------------------------------------
291     process(schedule, MIIA_RX_BUS, MIIB_RX_BUS, MIID_RX_BUS, MIIA_RX_CLK,
    MIIB_RX_CLK, MIID_RX_CLK)
292     begin
293         case schedule(3 downto 2) is
294             when "00" =>
295                 C_RX_CLK <= MIIA_RX_CLK;
296                 C_RX_BUS <= MIIA_RX_BUS;
297                 --MIIC_state <= MIIA2MIIC;
298             when "01" =>
299                 C_RX_CLK <= MIIB_RX_CLK;
300                 C_RX_BUS <= MIIB_RX_BUS;
301                 --MIIC_state <= MIIB2MIIC;
302             when "10" =>
303                 C_RX_CLK <= '0';
304                 C_RX_BUS <= (others => '0');
305                 --MIIC_state <= MIIC2MIIC;
306             when "11" =>
307                 C_RX_CLK <= MIID_RX_CLK;
308                 C_RX_BUS <= MIID_RX_BUS;
309                 --MIIC_state <= MIID2MIIC;
310             when others =>
311                 C_RX_CLK <= '0';
312                 C_RX_BUS <= (others => '0');
313                 --MIIC_state <= MIIC2MIIC;
314         end case;
315     end process;
316     ---------------------------------------------------------------------
317     process(schedule, MIIA_RX_BUS, MIIB_RX_BUS, MIIC_RX_BUS, MIIA_RX_CLK,
    MIIB_RX_CLK, MIIC_RX_CLK)
318     begin
319         case schedule(1 downto 0) is
320             when "00" =>
321                 D_RX_CLK <= MIIA_RX_CLK;
322                 D_RX_BUS <= MIIA_RX_BUS;
323                 --MIID_state <= MIIA2MIID;
324             when "01" =>
325                 D_RX_CLK <= MIIB_RX_CLK;
326                 D_RX_BUS <= MIIB_RX_BUS;
327                 --MIID_state <= MIIB2MIID;
328             when "10" =>
329                 D_RX_CLK <= MIIC_RX_CLK;
330                 D_RX_BUS <= MIIC_RX_BUS;
331                 --MIID_state <= MIIC2MIID;
332             when "11" =>
333                 D_RX_CLK <= '0';
334                 D_RX_BUS <= (others => '0');
335                 --MIID_state <= MIID2MIID;
336             when others =>
337                 D_RX_CLK <= '0';
338                 D_RX_BUS <= (others => '0');
```

```
339                     --MIID_state <= MIID2MIID;
340             end case;
341         end process;
342         -------------------------------------------------------------------------
343     end Behavioral;
```

```
1   library IEEE;
2       use IEEE.STD_LOGIC_1164.ALL;
3       use IEEE.STD_LOGIC_ARITH.ALL;
4       use IEEE.STD_LOGIC_UNSIGNED.ALL;
5   ---------------------------------------------------------------------------
6   entity RTC_CORE_TB is
7   end RTC_CORE_TB;
8   ---------------------------------------------------------------------------
9   architecture Behavioral of RTC_CORE_TB is
10      -----------------------------------------------------------------------
11      constant PERIOD : time := 40 ns;
12
13      constant MIIA_RX_DUTY_CYCLE : real := 0.4;
14      constant MIIA_TX_DUTY_CYCLE : real := 0.6;
15      constant MIIA_RX_OFFSET     : time := 10 ns;
16      constant MIIA_TX_OFFSET     : time := 20 ns;
17
18      constant MIIB_RX_DUTY_CYCLE : real := 0.5;
19      constant MIIB_TX_DUTY_CYCLE : real := 0.4;
20      constant MIIB_RX_OFFSET     : time := 15 ns;
21      constant MIIB_TX_OFFSET     : time := 30 ns;
22
23      constant MIIC_RX_DUTY_CYCLE : real := 0.6;
24      constant MIIC_TX_DUTY_CYCLE : real := 0.5;
25      constant MIIC_RX_OFFSET     : time := 20 ns;
26      constant MIIC_TX_OFFSET     : time := 25 ns;
27
28      constant MIID_RX_DUTY_CYCLE : real := 0.6;
29      constant MIID_TX_DUTY_CYCLE : real := 0.4;
30      constant MIID_RX_OFFSET     : time := 30 ns;
31      constant MIID_TX_OFFSET     : time := 10 ns;
32      -----------------------------------------------------------------------
33      signal Sync_CLK: std_logic;
34      signal RESET : std_logic := '0';
35      -----------------------------------------------------------------------
36      signal MIIA_RX_CLK  : std_logic := '0';
37      signal MIIA_RX_DV   : std_logic := '0';
38      signal MIIA_RX_DATA : std_logic_vector(3 downto 0) := (others => '0');
39      signal MIIA_RX_ER   : std_logic := '0';
40      signal MIIA_TX_CLK  : std_logic := '0';
41      signal MIIA_TX_EN   : std_logic := '0';
42      signal MIIA_TX_DATA : std_logic_vector(3 downto 0) := (others => '0');
43      signal MIIA_TX_ER   : std_logic := '0';
44      -----------------------------------------------------------------------
45      signal MIIB_RX_CLK  : std_logic := '0';
46      signal MIIB_RX_DV   : std_logic := '0';
47      signal MIIB_RX_DATA : std_logic_vector(3 downto 0) := (others => '0');
48      signal MIIB_RX_ER   : std_logic := '0';
49      signal MIIB_TX_CLK  : std_logic := '0';
50      signal MIIB_TX_EN   : std_logic := '0';
51      signal MIIB_TX_DATA : std_logic_vector(3 downto 0) := (others => '0');
52      signal MIIB_TX_ER   : std_logic := '0';
53      -----------------------------------------------------------------------
54      signal MIIC_RX_CLK  : std_logic := '0';
55      signal MIIC_RX_DV   : std_logic := '0';
56      signal MIIC_RX_DATA : std_logic_vector(3 downto 0) := (others => '0');
57      signal MIIC_RX_ER   : std_logic := '0';
```

```
58      signal MIIC_TX_CLK  : std_logic := '0';
59      signal MIIC_TX_EN   : std_logic := '0';
60      signal MIIC_TX_DATA : std_logic_vector(3 downto 0) := (others => '0');
61      signal MIIC_TX_ER   : std_logic := '0';
62      ------------------------------------------------------------------
63      signal MIID_RX_CLK  : std_logic := '0';
64      signal MIID_RX_DV   : std_logic := '0';
65      signal MIID_RX_DATA : std_logic_vector(3 downto 0) := (others => '0');
66      signal MIID_RX_ER   : std_logic := '0';
67      signal MIID_TX_CLK  : std_logic := '0';
68      signal MIID_TX_EN   : std_logic := '0';
69      signal MIID_TX_DATA : std_logic_vector(3 downto 0) := (others => '0');
70      signal MIID_TX_ER   : std_logic := '0';
71      ------------------------------------------------------------------
72      component RTC_CORE is
73      port(
74          Sync_CLK        : out std_logic;
75          ------------------------------------------------------------------
76          MIIA_RX_CLK     : in std_logic;
77          MIIA_RX_DV      : in std_logic;
78          MIIA_RX_DATA    : in std_logic_vector(3 downto 0);
79          MIIA_RX_ER      : in std_logic;
80          MIIA_TX_CLK     : in std_logic;
81          MIIA_TX_EN      : out std_logic;
82          MIIA_TX_DATA    : out std_logic_vector(3 downto 0);
83          MIIA_TX_ER      : out std_logic;
84          ------------------------------------------------------------------
85          MIIB_RX_CLK     : in std_logic;
86          MIIB_RX_DV      : in std_logic;
87          MIIB_RX_DATA    : in std_logic_vector(3 downto 0);
88          MIIB_RX_ER      : in std_logic;
89          MIIB_TX_CLK     : in std_logic;
90          MIIB_TX_EN      : out std_logic;
91          MIIB_TX_DATA    : out std_logic_vector(3 downto 0);
92          MIIB_TX_ER      : out std_logic;
93          ------------------------------------------------------------------
94          MIIC_RX_CLK     : in std_logic;
95          MIIC_RX_DV      : in std_logic;
96          MIIC_RX_DATA    : in std_logic_vector(3 downto 0);
97          MIIC_RX_ER      : in std_logic;
98          MIIC_TX_CLK     : in std_logic;
99          MIIC_TX_EN      : out std_logic;
100         MIIC_TX_DATA    : out std_logic_vector(3 downto 0);
101         MIIC_TX_ER      : out std_logic;
102         ------------------------------------------------------------------
103         MIID_RX_CLK     : in std_logic;
104         MIID_RX_DV      : in std_logic;
105         MIID_RX_DATA    : in std_logic_vector(3 downto 0);
106         MIID_RX_ER      : in std_logic;
107         MIID_TX_CLK     : in std_logic;
108         MIID_TX_EN      : out std_logic;
109         MIID_TX_DATA    : out std_logic_vector(3 downto 0);
110         MIID_TX_ER      : out std_logic;
111         ------------------------------------------------------------------
112         RESET               : in std_logic
113         ------------------------------------------------------------------
114     );
```

```
115     end component;
116 ------------------------------------------------------------------------
117 ------------------------------------------------------------------------
118 ------------------------------------------------------------------------
119 begin
120     ------------------------------------------------------------------
121     UUT : RTC_CORE port map(
122         Sync_CLK,
123         MIIA_RX_CLK, MIIA_RX_DV, MIIA_RX_DATA, MIIA_RX_ER, MIIA_TX_CLK, MIIA_TX_EN,
     MIIA_TX_DATA, MIIA_TX_ER,
124         MIIB_RX_CLK, MIIB_RX_DV, MIIB_RX_DATA, MIIB_RX_ER, MIIB_TX_CLK, MIIB_TX_EN,
     MIIB_TX_DATA, MIIB_TX_ER,
125         MIIC_RX_CLK, MIIC_RX_DV, MIIC_RX_DATA, MIIC_RX_ER, MIIC_TX_CLK, MIIC_TX_EN,
     MIIC_TX_DATA, MIIC_TX_ER,
126         MIID_RX_CLK, MIID_RX_DV, MIID_RX_DATA, MIID_RX_ER, MIID_TX_CLK, MIID_TX_EN,
     MIID_TX_DATA, MIID_TX_ER,
127         RESET
128     );
129     ------------------------------------------------------------------------
130     ------------------------------------------------------------------------
131     ------------------------------------------------------------------------
132     MIIA_RX_Clock : process
133     begin
134         wait for MIIA_RX_OFFSET;
135         CLOCK_LOOP : LOOP
136             MIIA_RX_CLK <= '0';
137             wait for (PERIOD - (PERIOD * MIIA_RX_DUTY_CYCLE));
138             MIIA_RX_CLK <= '1';
139             WAIT FOR (PERIOD * MIIA_RX_DUTY_CYCLE);
140         END LOOP;
141     end process MIIA_RX_Clock;
142     ------------------------------------------------------------------------
143     MIIA_TX_Clock : process
144     begin
145         wait for MIIA_TX_OFFSET;
146         CLOCK_LOOP : LOOP
147             MIIA_TX_CLK <= '0';
148             wait for (PERIOD - (PERIOD * MIIA_TX_DUTY_CYCLE));
149             MIIA_TX_CLK <= '1';
150             WAIT FOR (PERIOD * MIIA_TX_DUTY_CYCLE);
151         END LOOP;
152     end process MIIA_TX_Clock;
153     ------------------------------------------------------------------------
154     ------------------------------------------------------------------------
155     ------------------------------------------------------------------------
156     MIIB_RX_Clock : process
157     begin
158         wait for MIIB_RX_OFFSET;
159         CLOCK_LOOP : LOOP
160             MIIB_RX_CLK <= '0';
161             wait for (PERIOD - (PERIOD * MIIB_RX_DUTY_CYCLE));
162             MIIB_RX_CLK <= '1';
163             WAIT FOR (PERIOD * MIIB_RX_DUTY_CYCLE);
164         END LOOP;
165     end process MIIB_RX_Clock;
166     ------------------------------------------------------------------------
167     MIIB_TX_Clock : process
```

```
168       begin
169           wait for MIIB_TX_OFFSET;
170           CLOCK_LOOP : LOOP
171               MIIB_TX_CLK <= '0';
172               wait for (PERIOD - (PERIOD * MIIB_TX_DUTY_CYCLE));
173               MIIB_TX_CLK <= '1';
174               WAIT FOR (PERIOD * MIIB_TX_DUTY_CYCLE);
175           END LOOP;
176       end process MIIB_TX_Clock;
177       ----------------------------------------------------------------------
178       ----------------------------------------------------------------------
179       ----------------------------------------------------------------------
180       MIIC_RX_Clock : process
181       begin
182           wait for MIIC_RX_OFFSET;
183           CLOCK_LOOP : LOOP
184               MIIC_RX_CLK <= '0';
185               wait for (PERIOD - (PERIOD * MIIC_RX_DUTY_CYCLE));
186               MIIC_RX_CLK <= '1';
187               WAIT FOR (PERIOD * MIIC_RX_DUTY_CYCLE);
188           END LOOP;
189       end process MIIC_RX_Clock;
190       ----------------------------------------------------------------------
191       MIIC_TX_Clock : process
192       begin
193           wait for MIIC_TX_OFFSET;
194           CLOCK_LOOP : LOOP
195               MIIC_TX_CLK <= '0';
196               wait for (PERIOD - (PERIOD * MIIC_TX_DUTY_CYCLE));
197               MIIC_TX_CLK <= '1';
198               WAIT FOR (PERIOD * MIIC_TX_DUTY_CYCLE);
199           END LOOP;
200       end process MIIC_TX_Clock;
201       ----------------------------------------------------------------------
202       ----------------------------------------------------------------------
203       ----------------------------------------------------------------------
204       MIID_RX_Clock : process
205       begin
206           wait for MIID_RX_OFFSET;
207           CLOCK_LOOP : LOOP
208               MIID_RX_CLK <= '0';
209               wait for (PERIOD - (PERIOD * MIID_RX_DUTY_CYCLE));
210               MIID_RX_CLK <= '1';
211               WAIT FOR (PERIOD * MIID_RX_DUTY_CYCLE);
212           END LOOP;
213       end process MIID_RX_Clock;
214       ----------------------------------------------------------------------
215       MIID_TX_Clock : process
216       begin
217           wait for MIID_TX_OFFSET;
218           CLOCK_LOOP : LOOP
219               MIID_TX_CLK <= '0';
220               wait for (PERIOD - (PERIOD * MIID_TX_DUTY_CYCLE));
221               MIID_TX_CLK <= '1';
222               WAIT FOR (PERIOD * MIID_TX_DUTY_CYCLE);
223           END LOOP;
224       end process MIID_TX_Clock;
```

```
225         ------------------------------------------------------------------------
226         ------------------------------------------------------------------------
227         ------------------------------------------------------------------------
228         MIIA_input_data : process
229         begin
230             wait until Sync_CLK = '1';
231             wait until MIIA_RX_CLK = '0';
232             ------------------------------------------------------------------------
233             -- Preamble nibbles
234             for i in 0 to 9 loop
235                 wait for PERIOD;
236                 MIIA_RX_DV  <= '1';
237                 MIIA_RX_DATA <= "0101";
238             end loop;
239             ------------------------------------------------------------------------
240             -- SFD nibble
241             wait for PERIOD;
242             MIIA_RX_DV  <= '1';
243             MIIA_RX_DATA <= "1101";
244
    ------------------------------------------------------------------------

245             -- Daten nibbles
246             for i in 0 to 9 loop
247                 wait for PERIOD;
248                 MIIA_RX_DV  <= '1';
249                 MIIA_RX_DATA <= "1010";
250             end loop;
251             ------------------------------------------------------------------------
252             -- EFD nibble
253             wait for PERIOD;
254             MIIA_RX_DV  <= '1';
255             MIIA_RX_DATA <= "1111";
256             ------------------------------------------------------------------------
257             -- IFS nibbles
258             for i in 0 to 1 loop
259                 wait for PERIOD;
260                 MIIA_RX_DV  <= '0';
261                 MIIA_RX_DATA <= (others => '0');
262             end loop;
263             ------------------------------------------------------------------------
264             wait until Sync_CLK = '0';
265             ------------------------------------------------------------------------
266         end process MIIA_input_data;
267         ------------------------------------------------------------------------
268         ------------------------------------------------------------------------
269         ------------------------------------------------------------------------
270         MIIB_input_data : process
271         begin
272             wait until Sync_CLK = '1';
273             wait until MIIB_RX_CLK = '0';
274             ------------------------------------------------------------------------
275             -- Preamble nibbles
276             for i in 0 to 9 loop
277                 wait for PERIOD;
278                 MIIB_RX_DV  <= '1';
279                 MIIB_RX_DATA <= "0101";
```

```
280             end loop;
281             ------------------------------------------------------------------------
282             -- SFD nibble
283             wait for PERIOD;
284             MIIB_RX_DV  <= '1';
285             MIIB_RX_DATA <= "1101";
286
    ------------------------------------------------------------------------

287             -- Daten nibbles
288             for i in 0 to 9 loop
289                 wait for PERIOD;
290                 MIIB_RX_DV  <= '1';
291                 MIIB_RX_DATA <= "1011";
292             end loop;
293             ------------------------------------------------------------------------
294             -- EFD nibble
295             wait for PERIOD;
296             MIIB_RX_DV  <= '1';
297             MIIB_RX_DATA <= "1111";
298             ------------------------------------------------------------------------
299             -- IFS nibbles
300             for i in 0 to 1 loop
301                 wait for PERIOD;
302                 MIIB_RX_DV  <= '0';
303                 MIIB_RX_DATA <= (others => '0');
304             end loop;
305             ------------------------------------------------------------------------
306             wait until Sync_CLK = '0';
307             ------------------------------------------------------------------------
308         end process MIIB_input_data;
309         ------------------------------------------------------------------------
310         ------------------------------------------------------------------------
311         MIIC_input_data : process
312         begin
313             wait until Sync_CLK = '1';
314             wait until MIIC_RX_CLK = '0';
315             ------------------------------------------------------------------------
316             -- Preamble nibbles
317             for i in 0 to 9 loop
318                 wait for PERIOD;
319                 MIIC_RX_DV  <= '1';
320                 MIIC_RX_DATA <= "0101";
321             end loop;
322             ------------------------------------------------------------------------
323             -- SFD nibble
324             wait for PERIOD;
325             MIIC_RX_DV  <= '1';
326             MIIC_RX_DATA <= "1101";
327
    ------------------------------------------------------------------------

328             -- Daten nibbles
329             for i in 0 to 9 loop
330                 wait for PERIOD;
331                 MIIC_RX_DV  <= '1';
332                 MIIC_RX_DATA <= "1100";
```

```
333             end loop;
334             ----------------------------------------------------------------------
335             -- EFD nibble
336             wait for PERIOD;
337             MIIC_RX_DV  <= '1';
338             MIIC_RX_DATA <= "1111";
339             ----------------------------------------------------------------------
340             -- IFS nibbles
341             for i in 0 to 1 loop
342                 wait for PERIOD;
343                 MIIC_RX_DV  <= '0';
344                 MIIC_RX_DATA <= (others => '0');
345             end loop;
346             ----------------------------------------------------------------------
347             wait until Sync_CLK = '0';
348             ----------------------------------------------------------------------
349         end process MIIC_input_data;
350         ------------------------------------------------------------------------
351         ------------------------------------------------------------------------
352         MIID_input_data : process
353         begin
354             wait until Sync_CLK = '1';
355             wait until MIID_RX_CLK = '0';
356             ----------------------------------------------------------------------
357             -- Preamble nibbles
358             for i in 0 to 9 loop
359                 wait for PERIOD;
360                 MIID_RX_DV  <= '1';
361                 MIID_RX_DATA <= "0101";
362             end loop;
363             ----------------------------------------------------------------------
364             -- SFD nibble
365             wait for PERIOD;
366             MIID_RX_DV  <= '1';
367             MIID_RX_DATA <= "1101";
368
    ------------------------------------------------------------------------

369             -- Daten nibbles
370             for i in 0 to 9 loop
371                 wait for PERIOD;
372                 MIID_RX_DV  <= '1';
373                 MIID_RX_DATA <= "1101";
374             end loop;
375             ----------------------------------------------------------------------
376             -- EFD nibble
377             wait for PERIOD;
378             MIID_RX_DV  <= '1';
379             MIID_RX_DATA <= "1111";
380             ----------------------------------------------------------------------
381             -- IFS nibbles
382             for i in 0 to 1 loop
383                 wait for PERIOD;
384                 MIID_RX_DV  <= '0';
385                 MIID_RX_DATA <= (others => '0');
386             end loop;
387             ----------------------------------------------------------------------
```

```
388             wait until Sync_CLK = '0';
389             ----------------------------------------------------------------------
390         end process MIID_input_data;
391         ----------------------------------------------------------------------
392         ----------------------------------------------------------------------
393         ASSERT_Prozess : process
394         begin
395             wait for 30 us;
396             ASSERT (FALSE) REPORT
397             "Ende der Simulation."
398             SEVERITY FAILURE;
399         end process ASSERT_Prozess;
400         ----------------------------------------------------------------------
401     end Behavioral;
```