
Erweiterung feldprogrammierbarer Bausteine um eine
PCI Express Schnittstelle als Schlüsseltechnologie zur
Vernetzung digitaler Systeme und künstlicher Intelligenz

Masterthesis

Fachhochschule Dortmund

Fachbereich Informationstechnik

Interessengemeinschaft für Mikroelektronik und Eingebettete Systeme

Masterstudiengang

Informations- und Elektrotechnik

Betreuer: Prof. Dr.-Ing. Michael Karagounis

Autor: Philipp Ledüç

Datum: 05.06.2021

Kurzfassung

Erweiterung feldprogrammierbarer Bausteine um eine PCI Express Schnittstelle als Schlüsseltechnologie zur Vernetzung digitaler Systeme und künstlicher Intelligenz.

In dieser Masterthesis wird die Entwicklung eines PIPE IP-Cores als erster Entwicklungsschritt hin zu einem PCI Express *Soft Core* für die FPGA-basierte Implementierung beschrieben. Der Entwicklungsansatz hat zum Ziel, FPGAs mit integriertem Serializer/Deserializer (SerDes) auf den Einsatz in hardwareübergreifenden Systemen der Künstlichen Intelligenz (KI) vorzubereiten. Die Entwicklung basiert hierbei auf der FPGA-Produktfamilie GateMate™ des deutschen Unternehmens Cologne Chip AG. Allerdings versteht sich die Entwicklung als allgemeingültiger Ansatz, der auch anderen FPGA-Herstellern die Herangehensweise an die Thematik erleichtern und helfen soll, den notwendigen Entwicklungsaufwand abzuschätzen und wenn möglich zu verringern.

Abstract

Enhancement of field-programmable gate arrays using PCI Express interface as key technology for networking digital systems and artificial intelligence.

In this master thesis the development of a PIPE IP-Core is described as the first development step towards a PCI Express *Soft Core* for FPGA-based implementation. The development approach aims at preparing FPGAs with integrated serializer/deserializer (SerDes) for the use in cross-hardware artificial intelligence (AI) systems. The development is based on the FPGA product family GateMate™ by the German company Cologne Chip AG. However, the development is intended as a generally valid approach that should also make it easier for other FPGA manufacturers to approach the topic and help to estimate and, if possible, reduce the necessary development effort.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass die von mir vorgelegte Prüfungsleistung selbstständig und ohne unzulässige fremde Hilfe erstellt worden ist. Alle verwendeten Quellen sind in der Arbeit so aufgeführt, dass Art und Umfang der Verwendung nachvollziehbar sind.

Ort, Datum

Unterschrift

Danksagung

Ich danke Herrn Dr. Michael Gude und Herrn Prof. Dr. Michael Karagounis, dass sie mir die eigenständige Bearbeitung einer so interessanten Thematik ermöglichen konnten und mir das notwendige Vertrauen entgegengebracht haben. Ich möchte auch Herrn Andreas Stiller für die zusätzliche Betreuung während der Durchführung des Projektes danken.

Ich bedanke mich bei der Cologne Chip AG für die einzigartige Chance an einem so interessanten Produkt wie dem GateMateTM mitzuarbeiten und die Entwicklung eines PCI Express Soft-Cores voranzutreiben. Ebenfalls möchte ich mich für die positive Arbeitsatmosphäre und die offene Diskussionsbereitschaft bei der Behandlung von Problemstellungen bedanken.

Der Stiftung Industrieforschung danke ich für die finanzielle Unterstützung im Rahmen des vergebenen Stipendiums, welche mir ein freies und fokussiertes Arbeiten an der Thematik meiner Masterthesis ermöglicht hat.

Besonderer Dank gilt meiner Familie und meinen Freunden, die mich auf meinem akademischen Werdegang stets unterstützt haben und ohne die diese Masterthesis niemals möglich gewesen wäre.

Philipp Ledüç

Dortmund, 05.06.2021

Inhaltsverzeichnis

1	Einleitung	1
1.1	Ziel dieser Arbeit	2
2	Künstliche Intelligenz	3
2.1	Maschinelles Lernen	5
2.2	Künstliche Neuronale Netze	7
2.2.1	Neuronenmodell	7
2.2.2	Struktur und Funktion	9
2.2.3	Training eines Netzwerks	13
2.3	Convolutional Neural Networks	15
3	Technologieübersicht	17
3.1	FPGA	17
3.1.1	Hardware Description Languages	20
3.1.2	Einsatz in KI-Anwendungen	23
3.2	PCI Express	29
3.2.1	Transaction Layer	30
3.2.2	Data Link Layer	31
3.2.3	Physical Layer	31
3.3	PHY Interface for the PCIe Architecture	32
3.3.1	Interface	34
3.3.2	Reset	39
3.3.3	Power Management	39
3.3.4	Receiver Detection	45
3.3.5	Clock Tolerance Compensation	46
3.3.6	Error Detection	47
3.3.7	Loopback Mode	50
3.3.8	Rx Polarity	52
3.3.9	Negative Disparity (Compliance)	53
3.3.10	Electrical Idle	54
3.4	SerDes Architektur	55
3.4.1	ADPLL	59
3.4.2	Reset	62
3.4.3	Tx-/Rx-Datenpfad	64
3.4.4	Tx Configurable Driver	68
3.4.5	Rx Analog Front End	69
3.4.6	Rx Word Alignment	70
3.4.7	Loopback	73
4	Entwicklung	75
4.1	Entwicklungsansatz	75
4.2	PIPE IP-Core	76
4.2.1	PIPE Logic	77
4.2.2	PIPE FSM	90

4.2.3	Word Alignment	93
4.2.4	Tx Demultiplexer	95
4.2.5	Rx Multiplexer	97
5	Test & Verifikation	101
5.1	Testbench	101
5.2	Methoden	104
5.2.1	finalize	104
5.2.2	resetSerDes	104
5.2.3	writeCfg	105
5.2.4	writeRegfile	105
5.2.5	readRegfile	106
5.2.6	round	106
5.2.7	readSerIOADPLLStatus	107
5.2.8	startSerIOADPLL	107
5.3	PIPE Testcase	109
5.3.1	Reset	111
5.3.2	Power States	113
5.3.3	Word Alignment	115
5.3.4	Normal Transmission	116
5.3.5	Error Detection	117
5.3.6	Negative Disparity	119
5.3.7	Loopback Modus	121
5.3.8	Receiver Detection	122
6	Fazit und Ausblick	124
	Literaturverzeichnis	125
	Anhang	129
A.1	8b/10b Dekodiertabellen	129
A.2	SerDes Schnittstellen	140
A.3	Script zum Aufruf der Simulationsumgebung	150
A.4	Quellcode des PIPE IP-Cores	152
A.4.1	ccfpga_pipe_if.v	152
A.4.2	ccfpga_pipe_logic.v	156
A.4.3	ccfpga_pipe_fsm.v	162
A.4.4	ccfpga_pipe_fsm_word_align.v	166
A.4.5	ccfpga_pipe_tx_demux.v	167
A.4.6	ccfpga_pipe_rx_mux.v	169
A.4.7	CC_PLL.v	171
A.5	Quellcode der Testumgebung	172
A.5.1	tb_ccfpga_serdes.v	172
A.5.2	testcase_pipe.v	188

Abbildungsverzeichnis

2.1	Einteilung der genannten Themenbereiche für KI	5
2.2	Grundstruktur eines künstlichen Neurons	7
2.3	Auswahl von wichtigen Aktivierungsfunktionen	9
2.4	Beispielstruktur eines einfachen KNNs	10
2.5	<i>Layer</i> -Struktur und Verbindungen eines KNNs	11
2.6	Klassifikation von Objekten anhand von Bilddaten	12
2.7	Struktur eines <i>Convolutional Neural Networks</i>	15
2.8	2D-Faltung in einem <i>Convolutional Neural Network</i>	15
3.1	Vereinfachtes Modell eines FPGA-Bausteins	17
3.2	FPGA Logikzelle mit LUT	18
3.3	<i>Temporal Architecture</i> und <i>Spatial Architecture</i>	24
3.4	Speicherzugriff einer MAC Operation	24
3.5	Struktur für <i>Weight Stationary</i>	25
3.6	Struktur für <i>Output Stationary</i>	25
3.7	Struktur für <i>No Local Reuse</i>	26
3.8	Ablauf einer eindimensionalen Faltung	27
3.9	Architektur der PCI Express <i>Device Layer</i>	30
3.10	Position des PIPE im <i>Physical Layer</i>	33
3.11	PIPE Interface als Block Diagramm	34
3.12	Resetverhalten des PIPE	39
3.13	Einsatz der <i>Power States</i> in der LTSSM	42
3.14	Übergang des PHY in <i>Power State P0</i>	44
3.15	Übergang des PHY in <i>Power State P1</i>	45
3.16	Durchführung einer <i>Receiver Detection</i>	46
3.17	SKP <i>Ordered Set</i> mit hinzugefügtem SKP Symbol	47
3.18	SKP <i>Ordered Set</i> mit vollständig entfernten SKP Symbolen	47
3.19	Ausgabe eines 8b/10b Fehlers für Symbol Rx-f	48
3.20	Signalisierung des <i>Underflows</i> und Ausgabe EDB Symbol	49
3.21	Signalisierung eines <i>Overflows</i> für Symbol Rx-g	49
3.22	Signalisierung eines <i>Overflows</i> für Symbol Rx-g	50
3.23	Initialisierung des Loopback Modus	51
3.24	Austritt aus dem Loopback Modus	52
3.25	Invertierung der Polarität der Empfangsdaten	53
3.26	Setzen einer negativen Disparität	53
3.27	Übergang in den <i>Electrical Idle</i> Zustand	54
3.28	Blockschaltbild des SerDes	57
3.29	Blockschaltbild des Tx/Rx-Datenpfads	58
3.30	Vereinfachtes Blockschaltbild der ADPLL	59
3.31	Schematischer Ablauf des <i>Word Alignment</i>	70
3.32	Konfigurierbares <i>Alignment</i> für die <i>Byte Boundaries</i>	71
4.1	Erweiterung des SerDes um einen PIPE IP-Core.	75
4.2	Struktureller Aufbau des PIPE IP-Cores.	76

4.3	Blockschaltbild der PIPE Logic ohne Darstellung aller konstanten Portzuweisungen (64-Bit Version)	84
4.4	Blockschaltbild der PIPE Logic ohne Darstellung aller konstanten Portzuweisungen (16-Bit Version)	85
4.5	Blockschaltbild des Tx-/Rx-Datenpfads mit PLL (16-Bit Version) . .	86
4.6	Grafische Darstellung der PIPE FSM.	91
4.7	Grafische Darstellung der FSM für das <i>Word Alignment</i>	94
5.1	Blockschaltbild der durch die PIPE Logic erweiterten <i>Testbench</i>	101
5.2	Angabe des gewünschten Testfalls in der Konsole.	103
5.3	Ablauf des Reset für den PIPE IP-Core (<i>Power On Sequence</i>).	112
5.4	Reset des PIPE IP-Core im normalen Betriebszustand.	113
5.5	Test der Übergänge zwischen den <i>Power States</i> (Teil 1).	114
5.6	Test der Übergänge zwischen den <i>Power States</i> (Teil 2).	114
5.7	Ablauf des <i>Word Alignment</i> und Zuweisung von <i>o_RxValid</i>	115
5.8	Ausgabe der Methode <i>check_testdata()</i> in der Konsole.	117
5.9	Erzeugung von Bit-Fehlern auf den Empfangsleitungen.	118
5.10	Ausgabe der Methode <i>check_err_detect()</i> in der Konsole.	119
5.11	Anpassung der Disparität durch <i>i_TxCompliance</i> (Teil 1).	120
5.12	Anpassung der Disparität durch <i>i_TxCompliance</i> (Teil 2).	120
5.13	Aktivierung des <i>far-end PCS Loopback</i>	121
5.14	Deaktivierung des <i>far-end PCS Loopback</i>	121
5.15	Durchführung der <i>Receiver Detection</i> in <i>Power State</i> P1 (Teil 1) . . .	122
5.16	Durchführung der <i>Receiver Detection</i> in <i>Power State</i> P1 (Teil 2) . . .	123

Tabellenverzeichnis

3.1	PIPE Schnittstellen für die Sendedaten	35
3.2	PIPE Schnittstellen für die Empfangsdaten	36
3.3	PIPE Schnittstellen für die Kontrollsignale	37
3.4	PIPE Schnittstellen für die Statussignale	38
3.5	Reset-Werte der Kontrollsignale	39
3.6	Definition des Kontrollsignals <i>PowerDown</i>	40
3.7	Funktion von TxDetectRx/Loopback und TxElecIdle	43
3.8	Auszug der ADPLL-Registerfelder	60
3.9	Teilerwerte von <i>AddDiv</i> (64-Bit Datenpfad)	61
3.10	Schnittstellen zur Durchführung verschiedener Resets	63
3.11	SerDes Schnittstellen für den Reset-Status der Datenpfade	63
3.12	SerDes Schnittstellen für die Sendedaten	64
3.13	SerDes Schnittstellen für die Empfangsdaten	65
3.14	SerDes Schnittstelle für die Empfangsdaten (COM Symbol)	65
3.15	SerDes Schnittstellen für die Disparität der Sendedaten	66
3.16	SerDes Schnittstellen für die Fehlererkennung	67
3.17	SerDes Schnittstellen für die Fehlererkennung	68
3.18	Schnittstellen für <i>Electrical Idle</i> und <i>Receiver Detection</i>	69
3.19	Schnittstellen für die Detektierung von <i>Electrical Idle</i>	70
3.20	Schnittstellen für das <i>Symbol</i> bzw. <i>Word Alignment</i>	72
3.21	Schnittstelle für die Aktivierung des <i>Loopback</i> Modus	73
4.1	Schnittstellen für die <i>Custom Support Signals</i>	80
A.1	8b/10b Dekodiertabelle für Datenbytes	139
A.2	8b/10b Dekodiertabelle für Kontrollbytes	140
A.3	Schnittstellen des integrierten SerDes	149

Quellcodeverzeichnis

3.1	Port-Beschreibung für einen 2:1 Multiplexer in VHDL	21
3.2	Port-Beschreibung für einen 2:1 Multiplexer in Verilog	21
3.3	Multiplexer Funktion in VHDL	22
3.4	Multiplexer Funktion in Verilog	22
4.1	Parameter des PIPE IP-Core (<i>ccfpga_pipe_if.v</i>).	77
4.2	PIPE und CSS Schnittstellen in der PIPE Logic.	77
4.3	Auszug der SerDes Schnittstellen in der PIPE Logic.	80
4.4	Konstante Portzuweisungen in der PIPE Logic	81
4.5	Verschaltung der Taktleitungen in der PIPE Logic	87
4.6	Zuweisung des Signals <i>s_clk</i> (64-Bit Version)	87
4.7	Instanziierung der zusätzlichen PLL des GateMate™	87
4.8	Invertierung des Reset und Verschaltung mit der ADPLL.	88
4.9	Zuweisung des Signals <i>s_reset_done</i> (64-Bit Version)	88
4.10	Zuweisung des Signals <i>s_reset_done</i> (16-Bit Version)	88
4.11	Tx-Datenpfad in der PIPE Logic (64-Bit Version)	89
4.12	Rx-Datenpfad in der PIPE Logic (64-Bit Version)	89
4.13	Zuweisung des Signals <i>s_transit</i> in der FSM.	92
4.14	Multiplexer für die Zuweisung von <i>o_RxStatus</i>	93
4.15	Speicherung des Wertes für <i>o_RxValid</i>	95
4.16	Multiplexer für die Zuweisung von <i>o_RxValid</i>	95
4.17	Schnittstellen des Moduls <i>ccfpga_pipe_tx_demux</i>	95
4.18	Lokale Parameter und Variablen in <i>ccfpga_pipe_tx_demux</i>	96
4.19	<i>Always</i> -Blöcke im Modul <i>ccfpga_pipe_tx_demux</i>	96
4.20	Schnittstellen des Moduls <i>ccfpga_pipe_rx_mux</i>	97
4.21	Lokale Parameter und Signale in <i>ccfpga_pipe_rx_mux</i>	98
4.22	verab Empfangsdaten in <i>ccfpga_pipe_rx_mux</i>	98
4.23	Priorisierte Zuweisung des Statussignals <i>o_RxStatus</i>	99
4.24	<i>Generate</i> -Block zur iterativen Erzeugung der Zuweisungen	100
5.1	Generierung der notwendigen Taktsignale (<i>Clocks</i>).	102
5.2	Generierung der internen Resetsignale (<i>Physical</i>).	102
5.3	Definierte Parameter für die Konfiguration der ADPLL	108
5.4	Verbindung der Sende- und Empfangsleitungen des SerDes.	109
5.5	Reset-Verbindung zwischen der ADPLL und den Datenpfaden	110

Abkürzungsverzeichnis

AI	Artificial Intelligence
ADPLL	All Digital Phase Locked Loop
ASIC	Application-Specific Integrated Circuit
BISC	Built-In Self-Calibration
BNN	Binary Neural Network
CDR	Clock Data Recovery
CLB	Configurable Logic Block
CNN	Convolutional Neural Network
CPE	Central Programming Element
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CRD	Current Running Disparity
CSS	Custom Support Signals
DLLP	Data Link Layer Packet
DFE	Decision Feedback Equalizer
DNN	Deep Neural Network
DUT	Design Under Test
ECRC	End-to-End Cyclic Redundancy Check
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPU	Graphic Processing Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
IC	Integrated Circuit
IMES	Interessengemeinschaft für Mikroelektronik und Eingebettete Systeme
I/O	Input/Output
IUT	Implementation Under Test
KI	Künstliche Intelligenz
KNN	Künstliches Neuronales Netz
LAB	Logic Array Block
LCRC	Link Cyclic Redundancy Check
LSB	Least Significant Bit
LTSSM	Link Training and Status State Machine
LUT	LookUp Table
MAC	Media Access Control
ML	Machine Learning
MLP	Multi Layer Perceptron
MSB	Most Significant Bit
MUX	Multiplexer
NLP	Natural Language Processing
NLU	Natural Language Understanding
NPU	Network Processing Unit

PAL	Programmable Array Logic
PCI	Peripheral Component Interconnect
PCI-SIG	Peripheral Component Interconnect Special Interest Group
PCS	Physical Coding Sublayer
PFU	Programmable Function Unit
PIPE	PHY Interface for the PCI Express Architecture
PISO	Parallel-In Serial-Out
PLA	Programmable Logic Array
PLD	Programmable Logic Device
PLL	Phase Locked Loop
PMA	Physical Media Attachment
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SDK	Software Development Kit
SIMT	Single Instruction Multiple Threads
SIPO	Serial-In Parallel-Out
TLP	Transaction Layer Packet
Verilog	Verifying Logic (Verilog HDL)
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

1 Einleitung

Im Verlauf der letzten Jahrzehnte konnte die Künstliche Intelligenz (KI) in verschiedenen technologischen Bereichen großartige Erfolge erzielen und hat für die Industrie immens an Bedeutung gewonnen. Gerade Teilbereiche, wie Machine und Deep Learning, sind weiter in den Fokus von Forschern und Ingenieuren gerückt und gelten als wegweisend für zukünftige Entwicklungen. Die hardware-technischen Anforderungen an entsprechende KI-Systeme haben dazu geführt, dass sich neben Graphic Processing Units (GPU) auch Field Programmable Gate Arrays (FPGA) als Plattform für KI etabliert haben. Diese zeichnen sich vor allem durch ihren geringen Stromverbrauch und ihre Rekonfigurierbarkeit aus, wodurch sie ideal sind, um mit den raschen Entwicklungen Schritt zu halten und notwendige Anpassungen mit geringem Aufwand zu erzielen. Dies haben auch die führenden FPGA-Hersteller wie Xilinx oder Intel erkannt und seitdem daran gearbeitet, die Entwicklung von Neuronalen Netzen auf FPGA-Basis zu erleichtern und durch ihre Produkte zu unterstützen. Doch auch deutschen Unternehmen ist das große Potenzial von FPGAs in KI-Anwendungen sowie anderen technologischen Bereichen nicht verborgen geblieben. So hat die Cologne Chip AG bereits im Jahr 2017 mit der Realisierung einer eigenen FPGA-Produktfamilie namens GateMate™ begonnen, deren erster Ableger CCGM1A1 noch in diesem Jahr auf den Markt kommen soll [Col20]. Das in Köln angesiedelte Unternehmen kann dabei auf die jahrelange Erfahrung in der Entwicklung von Application-Specific Integrated Circuits (ASICs) für den Bereich der Telekommunikation zurückblicken und wagt als derzeit einziges deutsches Unternehmen den Schritt in Richtung FPGA-Technologie [Str20]. Das Vorhaben des Unternehmens wird zudem durch das Bundesministerium für Wirtschaft und Energie (engl. *Federal Ministry for Economic Affairs and Energy*) unterstützt, da es hilft den Entwicklungsstandort Deutschland unter dem Prädikat *Made in Germany* weiter zu stärken [Col20, S.9]. Eine Voraussetzung für den Aufbau von KI-Systemen bildet jedoch zwangsläufig die Vernetzung mehrerer FPGAs, da sich komplexe Systeme nur bedingt auf einem einzigen Integrated Circuit (IC) realisieren lassen. Dies setzt wiederum den Einsatz einer zuverlässigen Hochgeschwindigkeits-Schnittstelle mit hohem Datendurchsatz, wie etwa PCI Express (PCIe), voraus. Diese Schnittstelle wird auch bei der Vernetzung von GPUs eingesetzt, welche bedingt durch ihre Verwendung in kommerziellen PCs direkt über eine entsprechende PCIe Schnittstelle verfügen. Als Ausgangspunkt bei FPGAs dient zunächst ein integrierter Serializer/-Deserializer (SerDes) mit einer Übertragungsrate von mindestens 2,5 Gbit/s. Dieser ist eine Grundvoraussetzung und stellt den untersten Teil des *Physical Layers* für

PCIe dar. Die restliche Logik hin zum *Transaction Layer* für die Verarbeitung des PCIe Protokolls wird immer in Kombination mit einem PCIe Core realisiert, welcher auf dem SerDes aufbaut und die Verbindung zum *Application Layer* herstellt. Verfügt ein FPGA über keinen integrierten Hard Core, setzt dies die Implementierung eines PCIe Soft Cores in der Logik des FPGAs voraus und stellt dadurch einen notwendigen Schritt für den Einsatz in KI-Systemen dar.

1.1 Ziel dieser Arbeit

Diese Masterthesis stellt einen der ersten Entwicklungsschritte hin zu einem vollständigen PCIe Soft Core für die Implementierung auf FPGAs mit integriertem SerDes dar, um diese für den Einsatz in KI-Systemen vorzubereiten. Der Fokus liegt dabei auf der Entwicklung eines IP-Cores, welcher die Kompatibilität eines SerDes zum *PHY Interface for the PCI Express Architecture* (PIPE) gewährleisten soll, falls dieser über kein solches *Interface* verfügt. Der PIPE IP-Core soll zudem als Ausgangsbasis für zukünftige Entwicklungen dienen und den Einsatz von PCIe Soft Cores externer Hersteller vereinfachen. In Kooperation mit der Cologne Chip AG dient der CCGM1A1 der GateMateTM Produktfamilie als Ausgangsbasis für diese Entwicklungen. Es soll jedoch auch anderen Herstellern die Herangehensweise an die Thematik auf Basis ihrer eigenen FPGAs erleichtern und helfen den notwendigen Entwicklungsaufwand abzuschätzen und zu verringern. Zu Beginn wird ein Einblick in das Thema Künstliche Intelligenz mit Fokus auf Maschinelles Lernen und Neuronale Netze gegeben, bevor in der Technologieübersicht verschiedene Grundlagen für die Entwicklung des IP-Cores vermittelt werden. Die Masterthesis setzt jedoch ein entsprechendes Grundlagenwissen für PCI Express sowie den Zugriff auf die PCI Express Basisspezifikation der PCI-SIG voraus. In diesem Zusammenhang sei auf die beiden Quellen [PCI09] und [BASI12] verwiesen. In den darauffolgenden Kapiteln wird die Entwicklung und der Test des PIPE IP-Cores thematisiert. Den Abschluss bildet ein entsprechendes Fazit mit Ausblick auf das weitere Vorgehen und zukünftige Entwicklungsschritte.

2 Künstliche Intelligenz

Der Begriff Künstliche Intelligenz (engl. *Artificial Intelligence*) lässt sich am besten als ein übergeordnetes Fachgebiet verstehen, welches sich aus verschiedenen Teilbereichen der Naturwissenschaften und Technik zusammensetzt, und Systeme hervorbringt, die das logische Verhalten eines Menschen aufweisen. In diesem Zusammenhang werden KI-Systeme oft einfach als Künstliche Intelligenz oder KI bezeichnet. Eine genaue Definition von Künstlicher Intelligenz ist wie beim eigentlichen Begriff der Intelligenz nicht eindeutig möglich, da diese nur schwer messbar ist und sich auch immer auf bestimmte Anwendungsbereiche bezieht. Es wird jedoch allgemein von einer *schwachen KI* ausgegangen, die sich auf die Bearbeitung spezieller Aufgaben fokussiert und versucht, diese mit annähernd menschlichem Niveau zu lösen. Eine *starke KI* würde hingegen den bisher unrealistischen Ansatz einer Maschine bezeichnen, die in der Lage wäre nach menschlichem Vorbild zu denken und diesen sogar in seinen Fähigkeiten zu übertreffen [KS19, S.20]. Der erfolgreiche Einsatz von KIs ist heute in der Praxis bereits allgegenwärtig und kann in die folgenden Einsatzfelder aufgeteilt werden, die sich je nach Anwendung auch überschneiden [KS19, S.26]:

- Sprachverarbeitung
- Bildverarbeitung
- Expertensysteme
- Robotik

In der Sprachverarbeitung kommen KI-Systeme zum Einsatz, die in der Lage sind, die menschliche Sprache durch automatisierte Mustererkennung zu verstehen. Die Sprache kann dabei sowohl in textueller als auch in akustischer Form vorliegen. Die Schwierigkeit liegt hierbei neben der automatisierten Mustererkennung, vor allem in der Interpretation der Sprache und ihrer Bedeutung. Das Themengebiet wird unter dem Begriff *Natural-Language-Processing* (NLP) zusammengefasst, wobei dessen Teilbereich *Natural-Language-Understanding* (NLU), den KI-Prozess für die Verarbeitung gesprochener Sprache bezeichnet [KS19, S.30]. Zu den bekannten Vertretern von KI-Systemen in der Spracherkennung gehören etwa die Sprachassistenten *Alexa* und *Google Assistant* oder die Übersetzer *DeepL* und *Google Translate*. Die Bildverarbeitung, häufig mit dem Begriff *Computer-Vision* bezeichnet, entspricht der Verarbeitung von Bildern bzw. Fotos oder auch Videos, als Abfolge von Bildinhalten.

Unterschieden werden muss hier zunächst zwischen der Bildbearbeitung, wie sie etwa durch bestimmte Bildbearbeitungsprogramme (*Photoshop*) erfolgt und der Bilderkennung, welche neue zuvor unbekannte Informationen liefert. Letztere ermöglicht die zielgerichtete Identifikation von Bildinhalten und stellt den Kern automatisierter Musterkennungen in KI-Systemen dar. Eingesetzt wird die Computer-Vision etwa bereits in kamerabasierten Fahrerassistenzsystemen zur Überwachung der Straße, für die Gesichtserkennung von Personen oder die Analyse von medizinischen Bildmaterial (Röntgenaufnahmen). Bei Expertensystemen handelt es sich um Systeme, welche den "Menschen bei der Lösung komplexer Fragestellungen unterstützen"[KS19, S.41 Z.16] und "konkrete Handlungsempfehlungen auf Grundlage einer systematisch verfügbaren Wissensbasis"[KS19, S.41 Z.17] ableiten. Obwohl Expertensysteme keine Neuerung darstellen und schon seit Jahrzehnten im Einsatz sind, konnten bestehende Systeme durch den Einsatz der KI deutlich verbessert werden [KS19, S.41]. Ein interessantes Beispiel für ein Expertensystem, stellt etwa ein medizinische Anwendung dar, welche durch Auswertung der Informationen einer schwangeren Patientin eine Empfehlung abgibt, ob ein Kaiserschnitt durchzuführen sei [GBC18, S.3]. Ein solches System könnte sogar in der Lage sein, Zusammenhänge besser zu erkennen als ein Arzt und dadurch zu einer genaueren Einschätzung der medizinischen Situation gelangen, und im Zweifelsfall sogar Leben retten. Das Einsatzgebiet der Robotik kommt der allgemeinen Vorstellung von menschenähnlichen KIs am nächsten und umfasst alle Arten von technischen Maschinen, die gesteuert, automatisiert oder autonom mechanische Bewegungen bzw. Arbeiten durchführen können. Doch Roboter verfügen nicht zwangsläufig über eine KI, weshalb sich KI-Systeme meist auf Roboter beschränken, die automatisiert oder autonom handeln sollen. Ein Roboter verfügt zunächst über eine entsprechende Sensorik um seine Außenwelt wahrzunehmen und die eingehenden Daten, auch durch andere Einsatzfelder wie etwa Computer-Vision, zu analysieren. Letztendlich ist der Roboter, durch seine mechanischen Komponenten zur Fortbewegung (bspw. Räder) und Interaktion (bspw. Greifer) in der Lage, die Entscheidungen eines KI-Systems in die reale Welt zu übertragen und gezielt Aufgaben zu übernehmen. Ein Beispiel für die Umsetzung im Bereich der Robotik stellen mitunter auch autonome Fahrzeuge dar, deren Entwicklung in den letzten Jahren große Fortschritte gemacht hat. Hierbei erhält eine KI die Kontrolle über die Steuerung eines Fahrzeugs und nimmt auf Grundlage der eingehenden Sensordaten sowie der einprogrammierten Regeln am realen Straßenverkehr teil [KS19, S.44]. Die Einsatzfelder von KI-Systemen sind also sehr umfangreich und haben sich in den letzten Jahren auf viele Anwendungsbereiche ausgedehnt. Einen der wichtigsten Ansätze der KI stellt zum gegenwärtigen Zeitpunkt das sogenannte *Deep Learning*

auf Basis von Künstlichen Neuronalen Netzen (KNN) dar. Als Teilbereich des Maschinellen Lernens (ML) liefert es spezielle Algorithmen um die Leistung des ML in KI-Systemen zu steigern.

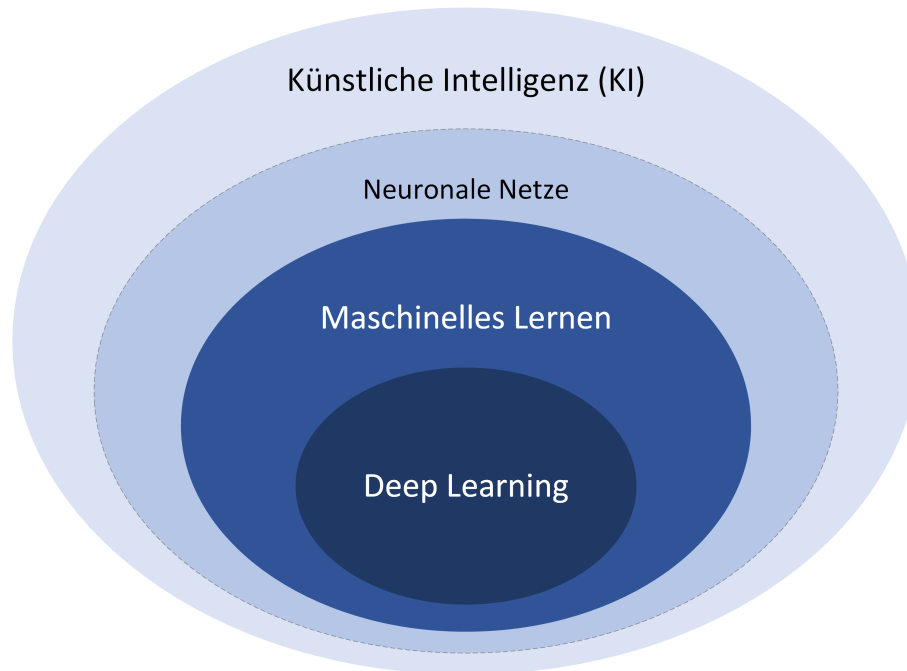


Abb. 2.1: Einteilung der genannten Themenbereiche für KI nach [KS19]

Eine gute Einteilung der Thematik liefert das in Abb. 2.1 gezeigte Venn-Diagramm. So setzt sich das Fachgebiet KI aus den Teilbereichen Neuronale Netze, Maschinelles Lernen und *Deep Learning* zusammen, die hierarchisch aufeinander aufbauen, sich in der Realität jedoch auch häufig überschneiden und gegenseitig beeinflussen. Die Neuronale Netze bzw. KNNs verstehen sich hierbei als Konzept zur Modellbildung, welche als Grundlage für KI-Systeme und *Deep Learning* dienen. ML ermöglicht es dann, aus den Testdaten der jeweiligen Anwendung Informationen zu gewinnen und die KNNs in KI-Systemen durch diese Erfahrungswerte anzupassen [KS19, S.4].

2.1 Maschinelles Lernen

Maschinelles Lernen bzw. *Machine Learning* (ML) befähigt KI-Systeme dazu, anhand von bereitgestellten Daten, Zusammenhänge und Muster zu erlernen, wodurch sie in der Lage sind, komplexe Aufgaben zu übernehmen, deren Lösung mit einem konventionellen Programm nicht möglich wäre. ML gilt gemeinhin als „der einzige praktikable Ansatz für die Entwicklung von KI-Systemen [...], die in der komplexen

realen Welt funktionieren.“[GBC18, S.9 Z.36]. In einem konventionellen Programm, welches etwa für die Bilderkennung eingesetzt werden soll, müssten alle Algorithmen und Regeln zur Kategorisierung von bestimmten Bildinhalten durch explizite Anweisungen des Entwicklers definiert werden. Eine solche Vorgehensweise ist zeitaufwendig und sorgt zudem dafür, dass das Programm bzw. System nur im Rahmen seiner vorher definierten Grenzen einsetzbar ist. Das System müsste schon bei leichten Veränderungen der Daten immer wieder manuell angepasst werden. Beim ML soll ein System selbständig einen Algorithmus erlernen wie die Verarbeitung der Daten auf Basis von Mustererkennung durchzuführen ist. Die Daten werden hierbei durch entsprechende Merkmale (*Features*) spezifiziert. Bei ML unterscheidet man zwischen dem überwachten Lernverfahren (*Supervised Learning*), dem unüberwachten Lernverfahren (*Unsupervised Learning*) und dem bestärkendem Lernverfahren (*Reinforced Learning*). Diese Teilgebiete zielen zudem auf unterschiedliche Anwendungsbereiche ab. Beim überwachten Lernverfahren werden dem System Trainingsdaten zur Verfügung gestellt, die bereits über ein *Label* verfügen und die erwartete Ausgabe des Systems definieren. Daher muss eine Datenvorverarbeitung der eingesetzten Trainingsdaten erfolgen. Das System erhält während des Trainings jeweils eine Rückmeldung (*Feedback*), ob die Daten von erwarteten Ergebniswerten abweichen. Dadurch kann das System seinen Algorithmus, durch Optimierungsverfahren, soweit anpassen, bis eine entsprechende Näherung erreicht worden ist. Nach Abschluss des Trainings lässt sich das System auch auf unbekannte Daten anwenden und greift dabei auf die zuvor erlernte Wissensbasis zurück. Die Ziele des überwachten Lernens sind zum einen die *Klassifikation*, also die Einteilung bestimmter Daten in vordefinierte Klassen, und zum anderen die *Regression*, also die Vorhersage von kontinuierlichen Werten. Bei unüberwachten Lernverfahren sind die Daten hingegen unbekannt und das System muss diese selbständig nach ihren Merkmalen klassifizieren und in *Labels* bzw. Klassen einteilen. In diesem Zusammenhang wird vom *Clustering* gesprochen. Ein weiteres Ziel dieser Lernverfahrens ist die Anomalie-Erkennung, also die Erkennung von Daten, die vom restlichen Datensatz abweichen. Beim bestärkenden Lernverfahren wird ein System (*Agent*) durch direktes positives oder negatives *Feedback* trainiert. Das System erhält dabei keine Vorgabe für korrekte Entscheidungen, sondern wird angehalten, Strategien für die Steigerung des positiven *Feedbacks* zu erlernen. Insgesamt wird ML vor allem dann eingesetzt, wenn große komplexe Datenmengen verarbeitet werden sollen und die Kodierung der Regeln von vielen Faktoren abhängig ist. In diesem Fall ist eine einfache regelbasierte Lösung nicht mehr ausreichend oder nur schwer umzusetzen [KH20].

2.2 Künstliche Neuronale Netze

Bei Künstlichen Neuronalen Netzen (engl. Artificial Neural Networks) handelt es sich um informationsverarbeitende Systeme, welche in ihrer Struktur und Funktionsweise dem Nervensystem des Menschen oder anderer Lebewesen nachempfunden sind. Sie setzen sich wie ihr biologisches Vorbild aus einer großen Anzahl von Neuronen bzw. künstlichen Neuronen mit einer simplen Struktur zusammen. Diese parallel arbeitenden Basiseinheiten übernehmen einfache Berechnungen und können für die Lösung komplexerer Aufgaben miteinander verschaltet werden [KBB⁺15, S.8]. Die KNNs sind historisch gesehen eine der wichtigsten Entwicklungen und bilden zum heutigen Zeitpunkt die Basis für nahezu alle KI-Systeme.

2.2.1 Neuronenmodell

Ein künstliches Neuron stellt im Kern eine mathematische Funktion dar und orientiert sich in seiner Struktur am Neuron des Gehirns, wobei dessen Funktion stark vereinfacht übernommen wird und in der Realität deutlich komplexer ausfällt.

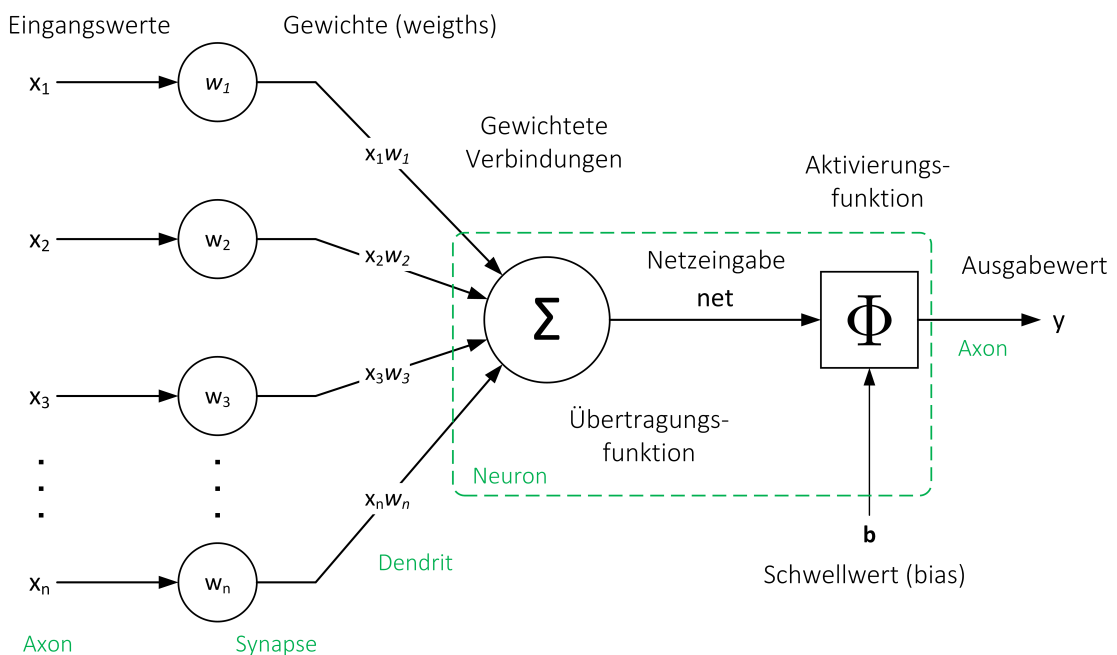


Abb. 2.2: Grundlegende Struktur eines künstlichen Neurons nach [OR06]

Die Grundstruktur eines künstlichen Neurons setzt sich aus den Eingangswerten

$(x_1 \dots x_n)$, den jeweiligen Gewichtungen dieser Eingangswerte $(w_1 \dots w_n)$, der Übertragungsfunktion und der durch den Schwellwert angepassten Aktivierungsfunktion für die Ausgabe des Neurons zusammen (Abb. 2.2). Im biologischen Vorbild werden die Eingänge eines Neurons als *Dendriten* und die Ausgänge als *Axone* bezeichnet. Die Eingangswerte, die das Neuron über seine Eingangsknoten erhält, stammen in der Regel von den Ausgängen anderer Neuronen auf vorherigen Ebenen im KNN. Es ist jedoch auch möglich, dass diese den direkten Datenwerten der jeweiligen Anwendung entsprechen, falls sich das Neuron auf der vordersten Ebene befindet. Das künstliche Neuron führt für die einzelnen Eingangswerte zunächst eine Multiplikation mit den Gewichtungen $(w_1 \dots w_n)$ durch. Dadurch werden die Eingangswerte, je nach Wert der Gewichtung, in ihrer Relevanz für das KNN und ihrem Einfluss auf die nachfolgende Übertragungsfunktion geschwächt oder gestärkt. Man spricht ausgehend vom biologischen Vorbild mit Bezug auf die gewichteten Verbindungen auch von *Synapsen*. Bei der Übertragungsfunktion, häufig auch als Propagierungs- oder Netzeingabefunktion bezeichnet, handelt es sich meist um eine einfache Summation der Eingangswerte. Die gesamte Netzeingabe *net* des Neurons entspricht demnach der Summe der gewichteten Eingangssignale (Formel 2.1).

$$net = \sum_{n=1}^N x_n w_n \quad (2.1)$$

$$y = \Phi(net) = \Phi\left(\sum_{n=1}^N x_n w_n + b\right) \quad (2.2)$$

Die Aktivierungsfunktion (Φ) wird auf das Ergebnis der Summation unter Berücksichtigung des Schwellwertes (b) angewendet, um den Ausgabewert des künstlichen Neurons (y) zu bestimmen (Formel 2.2). Der Schwellwert, auch als *Bias* oder *Threshold* bezeichnet, ermöglicht die Verschiebung der Netzeingabe (*net*) und bestimmt den Punkt ab dem ein Neuron *aktiviert* wird. Die Aktivierungsfunktion bestimmt im allgemeinen den Ausgabewert anhand der Netzeingabe und definiert somit den Grad der Aktivierung eines Neurons. Daher wird der Ausgang eines Neurons auch als *Activation* bezeichnet. Im einfachsten Fall besteht die Aktivierungsfunktion aus einer Schwellwertfunktion (*Binary Step*), welche die diskreten Werte von 0 oder 1 annimmt. Die eingesetzten Aktivierungsfunktionen sind auch immer abhängig von der Aufgabe des Neurons im KNN und der Art der Anwendung. Im Bereich des *Deep Learning* werden vor allem die *Rectified Linear Unit (ReLU)*, die *Leaky ReLU*

und die *Exponential Linear Unit* eingesetzt (Abb. 2.3). Andere nichtlineare Funktionen, wie die Sigmoidale Funktion oder der Tangens Hyperbolicus, sind ebenfalls von großer Bedeutung, kommen im Vergleich jedoch weniger häufig zur Anwendung, als die zuvor genannten Funktionen. Eingesetzt werden sie häufig, um die Ausgabe zwischen den Werten 0 und 1 bzw. -1 und 1 zu normieren (Abb. 2.3). Künstliche Neuronen verfügen normalerweise auch über eine Ausgabefunktion, die auf den Wert der Aktivierungsfunktion angewendet wird. Da diese jedoch in den meisten Fällen der Identitätsfunktion (*Identity*) entspricht, hat sie keinen direkten Einfluss auf den Ausgabewert des Neurons und kann daher vernachlässigt werden.

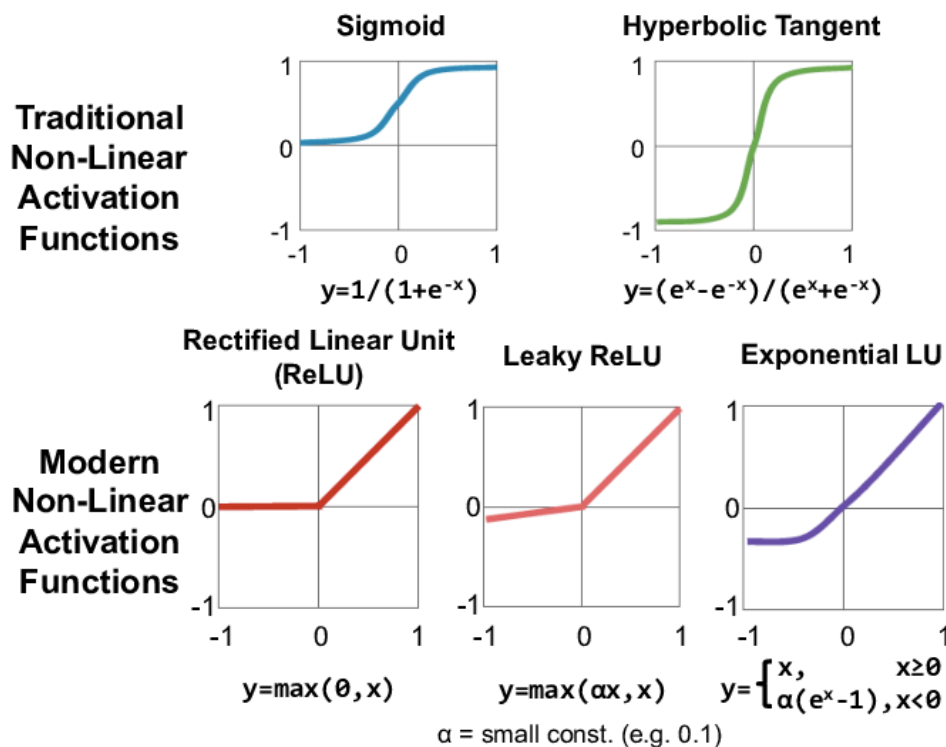


Abb. 2.3: Auswahl von wichtigen Aktivierungsfunktionen nach [SCYE17]

2.2.2 Struktur und Funktion

Erst die Verschaltung mehrerer künstlicher Neuronen zu Netzwerken, ermöglicht gewisse Funktionen der Aussagenlogik umzusetzen. So lässt sich bereits mit zwei Neuronen eine XOR-Verknüpfung umsetzen, welche mit einem einzelnen Neuron nicht möglich wäre und in der frühen Anfangsphase der KNNs die Entwicklung unter dem Begriff XOR-Problem zunächst ins Stocken brachte [GBC18, S.189]. Ein KNN setzt sich aus einer Vielzahl von Neuronen zusammen, die untereinander durch gerichtete und gewichtete Verbindungen vernetzt sind. Die Neuronen sind dabei in mehreren

Schichten (*Layern*) organisiert. Einzelne Neuronen, die sich auf der selben Schicht befinden, führen ihre Berechnungen parallel zueinander aus. Ein Netzwerk setzt sich aus einer Eingangsschicht (*Input Layer*), ein oder mehr verborgenen Schichten (*Hidden Layer*) und einer Ausgangsschicht (*Output Layer*) zusammen (Abb. 2.4).

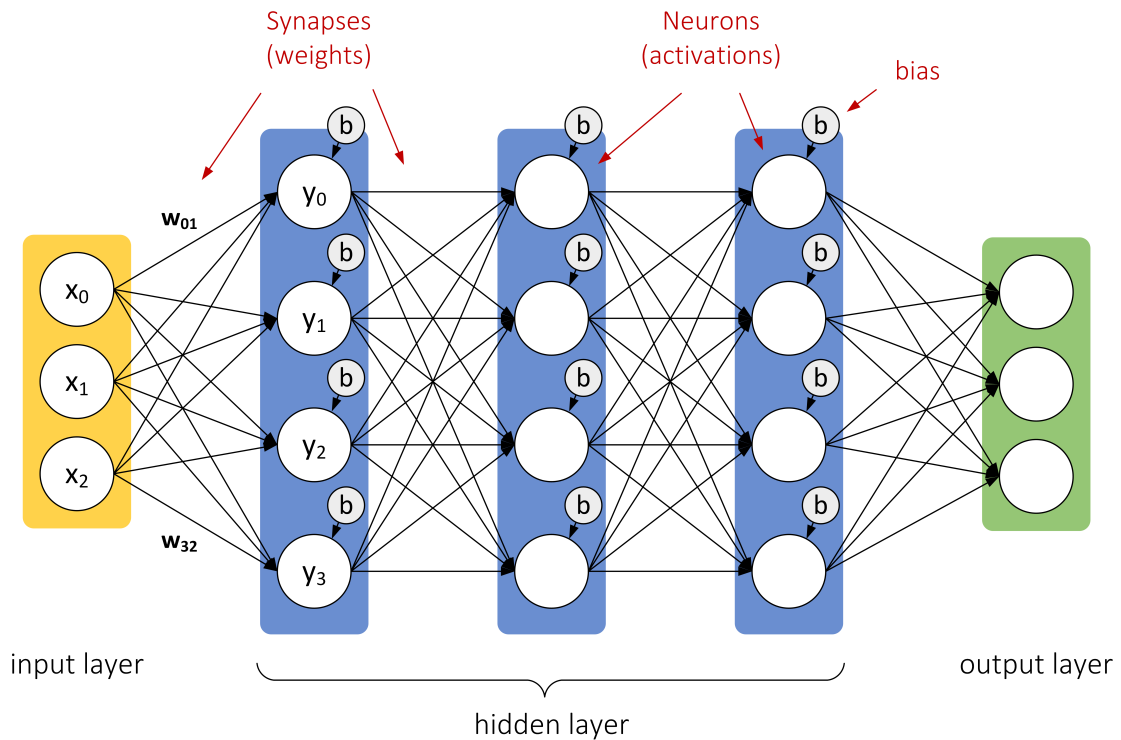


Abb. 2.4: Beispielstruktur eines einfachen KNNs nach [SCYE17]

Unterschiedliche Architekturen werden durch die Richtung und Anzahl der Verbindungen zwischen den einzelnen Schichten definiert. Der *Input Layer* nimmt die Eingangsdaten für das Netzwerk entgegen und bereitet diese für die numerische Verarbeitung durch das KNN auf (Normierung). Daher entsprechen die *Activations* der Eingangsschicht ($x_0 \dots x_n$) den normierten Eingangsdaten, wobei die Anzahl der Neuronen von den Eingangsdaten abhängt. Der *Output Layer* stellt die Ausgabe des KNN dar und entspricht im Rahmen der Klassifikation der Anzahl der möglichen Klassen. Weist ein KNN mehr als einen *Hidden Layer* auf, spricht man in der Regel bereits von einem *Deep Neural Network* (DNN), aus dem sich auch der Begriff *Deep Learning* ableitet. Heutige DNNs reichen von fünf bis zu über tausend *Layern* [SCYE17]. Man unterscheidet zudem zwischen vorwärts gerichteten Netzen (engl. *Feed-Forward Network*) und rekurrenten Netzen (engl. *Recurrent Network*). In vorwärts gerichteten Netzen sind nur Verbindungen auf die nachfolgenden *Layer* gestattet, wodurch die Daten das Netzwerk nur in eine Richtung passieren (*Forward*

Propagation). Die Neuronen in einer Schicht nehmen dabei die Ausgangsdaten der Neuronen der vorherigen Schicht entgegen und leiten ihre eigenen Ausgangsdaten an die Neuronen der nächsten Schicht weiter. Die Verarbeitung der Daten erfolgt also sequentiell, wobei diese jede Schicht einmalig durchlaufen. In einem rekurrenten Netzwerk sind hingegen Rückkopplungen (*Feedback Connections*) möglich, wodurch Daten eine Schicht auch erneut passieren können (Abb. 2.5).

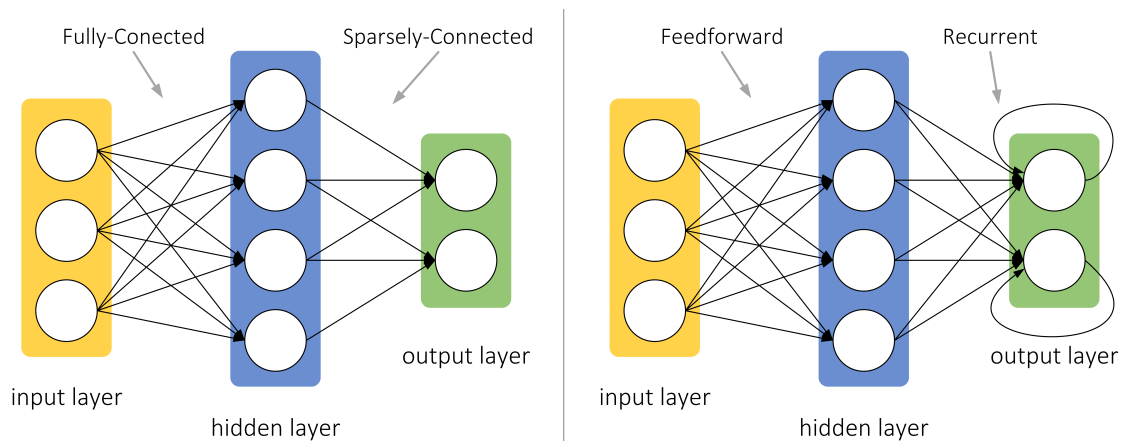


Abb. 2.5: *Layer*-Struktur und Verbindungen eines KNNs nach [SCYE17]

Ein solches Netzwerk, muss im Gegensatz zu einem vorwärts gerichteten Netzwerk über zusätzlichen Speicher verfügen, da bestimmte Ausgangswerte für einen erneuten Durchlauf zwischengespeichert und als Eingangswerte bereitgestellt werden müssen. Im Gegensatz zu einem vorwärts gerichteten Netzwerk sind die Ausgaben des Netzwerks (*Output Layer*) auch von der Anzahl der Durchläufe abhängig. Generell gelten für *Recurrent Neural Networks* (RNNs), abgesehen von den Rückkopplungen, beim Durchlauf der Daten die selben Funktionsprinzipien wie für *Feed-Forward* Netze. Da es sich bei den meisten KNNs um vorwärtsgerichtete Netze handelt und der hardwaretechnischen Umsetzung von RNNs bisher wenig Aufmerksamkeit zu Teil geworden ist, können diese jedoch vernachlässigt werden [SCYE17, S.7]. Ein KNN kann verschiedene Arten von *Layer*n aufweisen. Bei einem *Fully-Connected Layer* besteht die Netzeingabe aller Neuronen aus den gewichteten Ausgangswerten aller Neuronen des vorherigen *Layers*. Daher ist jedes Neuron des *Fully-Connected Layers* mit allen Ausgängen der Neuronen des vorherigen *Layers* verbunden. Ein vorwärts gerichtetes DNN, das vollständig aus *Fully-Connected Layers* besteht, wird als *Multi Layer Perceptron* (MLP) bezeichnet. Im Gegensatz dazu spricht man von einem *Sparsely-Connected Layer*, wenn nicht alle Verbindungen vorhanden sind. Dies kann etwa durch gewichtete Verbindungen mit Null erreicht werden. Die Funktion eines

vorwärts gerichteten DNNs lässt sich im Grunde wie eine mathematische Funktion verstehen, welche für die Eingangswerte des Datensatzes einen entsprechenden Ausgangswert für das Netzwerk erzeugt. Im Fall der *Klassifikation* verarbeitet diese die anliegenden Daten und ordnet den Datensatz einer bestimmten Klasse zu [GBC18, S.185]. Die Funktion wird dabei durch die Neuronen der einzelnen Schichten sowie die Aktivierungsfunktionen realisiert und über die Parameter in Form der verschiedenen Gewichte (*Weights*) und Schwellwerte (*Bias*) angepasst. Die Anpassung der Parameter wird im Rahmen des Trainings durchgeführt (siehe Kap. 2.2.3). Die Ausgabe für einen *Layer* des neuronalen Netzwerks lässt sich mit Vektoren und Matrizen wie folgt darstellen, wobei k das Ziel-Neuron und n das Ausgangs-Neuron kennzeichnet (Formel 2.3).

$$\vec{y} = \Phi \left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} \right) \quad (2.3)$$

Der Vorteil von DNNs liegt vor allem in ihrer Tiefe begründet, da diese im Gegensatz zu flacheren Netzen in der Lage sind zunächst sogenannte *Low-Level Features* in den Datensätzen zu erkennen, um aus diesen mit fortschreitendem Durchlauf der Daten durch das Netzwerk wichtige *High-Level Features* abzuleiten [SCYE17, S.3]. Im Bereich der Bildverarbeitung würden *Low-Level Features* etwa die Erkennung von Kanten oder anderer einfacher Formen darstellen, aus denen sich in tieferen *Layern* immer komplexere Bildinhalte wie Gesichter oder Tiere herleiten lassen (Abb. 2.6).

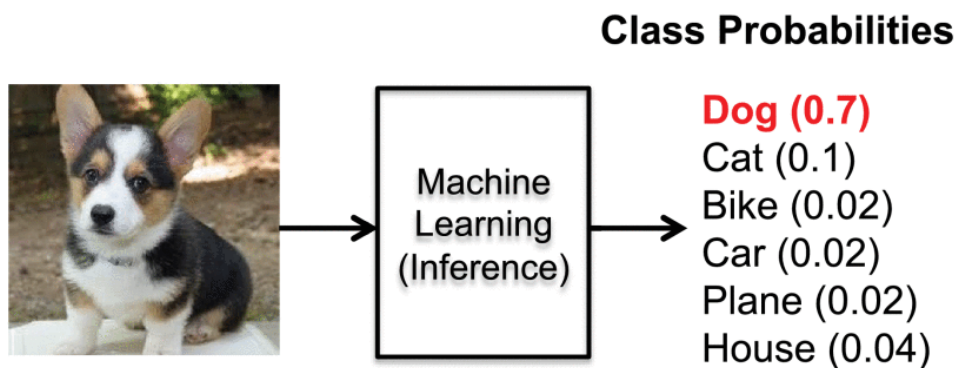


Abb. 2.6: Klassifikation von Objekten anhand von Bilddaten nach [SCYE17]

Im Gegensatz zu den bekannten Verfahren des *Machine Learnings* (Entscheidungsbäume, etc.) zeichnet sich das *Deep Learning* dadurch aus, dass die Daten keine Vorverarbeitung in Form von *Features* benötigen. Das *Deep Learning* setzt jedoch einen erheblich größeren Satz an Daten voraus, um diese *Features* beim Training des Netzwerks zu erlernen.

2.2.3 Training eines Netzwerks

Damit ein DNN erfolgreich auf Daten angewendet werden kann, müssen zunächst die Gewichte (*Weights*) und die Schwellwerte (*Bias*) durch das Training des Netzwerks für die entsprechende Anwendung eingestellt werden. Die Anwendung eines DNN auf Daten wird allgemein als Inferenz (engl. *Inference*) bezeichnet. Die häufigste Methode ein Netzwerk zu trainieren, stellt das *Supervised Learning* dar (siehe Kap. 2.1). Während des Trainings werden an das Netzwerk bestimmte Trainingsdatensätze angelegt, für die der Sollwert am Ausgang des Netzwerk bereits festgelegt ist. Bei der Klassifizierung von Bildmaterial würde bereits vorher feststehen, welcher Objektklasse der Datensatz zuzuordnen ist. Das DNN gibt für jede Klasse einen Zahlenwert aus und gibt dadurch indirekt an wie hoch die Wahrscheinlichkeit ist, dass der Datensatz der jeweiligen Klasse zuzuordnen ist (Abb. 2.6). Die Abweichung zwischen den idealen Werten und den vom Netzwerk ausgegebenen Werten wird als Verlust (engl. *Loss*) bezeichnet und über die Fehlerfunktion E definiert. Gängige differenzierbare Fehlerfunktionen sind die *Binary Cross Entropy* (Formel 2.4), die *Categorical Cross Entropy* (Formel 2.5) und der *Mean Square Error* (Formel 2.6).

$$E(\hat{y}) = -(y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y})) \quad (2.4)$$

$$E(\hat{y}) = - \sum_{i=0}^i y_i \cdot \log(\hat{y}_i) \quad (2.5)$$

$$E(\hat{y}) = \frac{1}{n} \sum_{i=0}^i (y_i - \hat{y}_i)^2 \quad (2.6)$$

Das Ziel des Trainings ist den Wert der Fehlerfunktion soweit wie möglich zu verringern. Da die Fehlerfunktion von allen Gewichten (*Weights*) und Schwellwerten (*Bias*) abhängig ist, müssen diese Parameter angepasst werden. Daher kommt in

diesem Fall häufig ein Verfahren namens *Backpropagation* zum Einsatz. Dieses resultiert aus der logischen Überlegung den Fehler durch das Gradientenabstiegsverfahren zu minimieren und die Fehlerfunktion, unter Zuhilfenahme der Kettenregel, partiell nach allen Schwellwerten hin abzuleiten (Formel 2.7).

$$\frac{\partial E(\hat{y})}{\partial w_{ij}} = \frac{\partial E(\hat{y})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial net_i} \cdot \frac{\partial net_i}{\partial w_{ij}} \quad (2.7)$$

$$\Delta w_{ij} = -\alpha \frac{\partial E}{\partial w_{ij}} \quad (2.8)$$

$$w_{ij}^{(neu)} = w_{ij}^{(alt)} - \Delta w_{ij}^{(neu)} = w_{ij}^{(alt)} - \alpha \frac{\partial E}{\partial w_{ij}} \quad (2.9)$$

Damit das Verfahren funktioniert müssen, jedoch die Ausgabewerte der einzelnen *Layer* zwischengespeichert werden, weshalb das Training eines Netzwerks entsprechende Speicherkapazitäten voraussetzt. Für die Anpassung der Gewichte ergibt sich der in Formel 2.8 und 2.9 gezeigte Zusammenhang, wobei die Lernrate α angibt wie stark sich dem lokalen Minimum der Fehlerfunktion pro Anpassung angenähert wird. Die Wahl der Lernrate erfolgt zu Beginn des Trainings zufällig. Sie beeinflusst sowohl die Genauigkeit bei der Bestimmung des Minimums als auch die Anzahl der benötigten Anpassungen. Um die benötigte Trainingszeit zu verringern, wird die Lernrate zu Beginn häufig größer gewählt und mit Voranschreiten des Trainings bzw. der Annäherung an das Minima kontinuierlich verringert. Die Fehler werden rückwärts durch das Netzwerk propagiert, bis alle Parameter hin zum *Input Layer* angepasst worden sind. Danach kann ein neues Set aus Trainingsdaten angelegt und die *Backpropagation* erneut durchgeführt werden. Dieser Vorgang wird nun hunderte bis mehrere tausendmal durchgeführt, bis sich am Ausgang ein akzeptabler *Loss* einstellt. In der Praxis hat es sich als vorteilhaft herausgestellt, einzelne Trainingsdaten zu sogenannten *Batches* zusammenzufassen und die Anpassung des Netzwerks für den durchschnittlichen ermittelten Fehler durchzuführen. Dies minimiert ebenfalls die benötigte Zeit für das Training und zum anderen den schädlichen individuellen Einfluss einzelner Trainingsdaten. Das Training eines DNN erfordert große Datenmengen und eine leistungsstarke Hardware zur Durchführung. Die Beschaffung von notwendigen Datensätzen stellt durch das Voranschreiten der Digitalisierung (*Big Data*) für die meisten Anwendungsbereiche kein Problem mehr dar. [Vor19] [SCYE17]

2.3 Convolutional Neural Networks

Eine sehr populäre Form von DNNs stellen, neben MLPs, die *Convolutional Neural Networks* (CNN) dar. Diese faltenden neuronalen Netzwerke werden vor allem im Bereich der Bild- und Audioverarbeitung eingesetzt. Sie setzen für die Datenverarbeitung mehrere aufeinanderfolgende *Convolutional Layer* ein (Abb. 2.7).

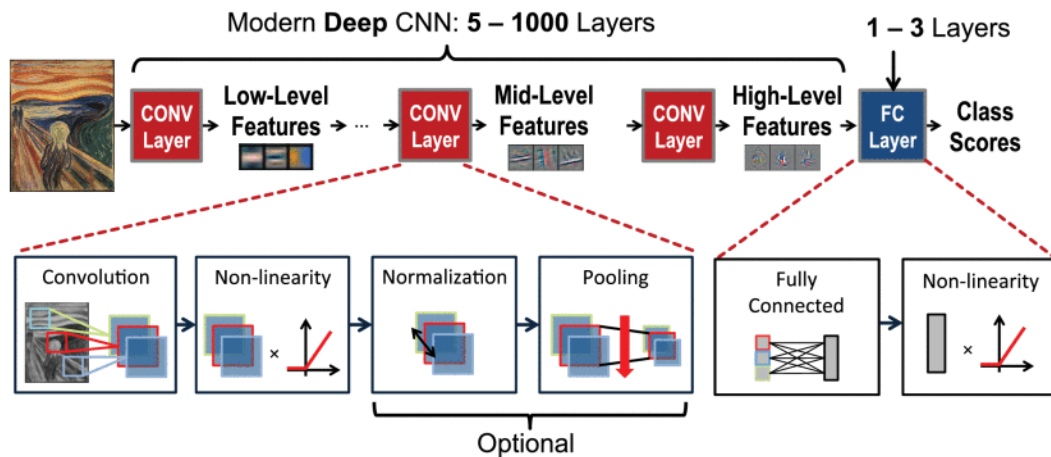


Abb. 2.7: Struktur eines *Convolutional Neural Networks* nach [SCYE17]

In einem *Convolutional Layer* werden die Eingangsdaten durch den aus der Bildverarbeitung bekannten Vorgang der Faltung (engl. *Convolution*) verarbeitet und führen zu einer Abstraktion der Daten auf höherer Ebene. Die Ausgabe resultiert dabei in sogenannten *Feature Maps*. Mit Bezug auf die Bildverarbeitung wird ein Filter (*Kernel*) auf einzelne Pixel und ihr direktes Umfeld aus benachbarten Pixeln angewendet. Dabei werden die jeweiligen Pixelwerte mit den korrespondierenden Werten der Filtermatrix multipliziert und deren Summe inklusive des *Bias* in der *Output Feature Map* abgelegt (Abb. 2.8).

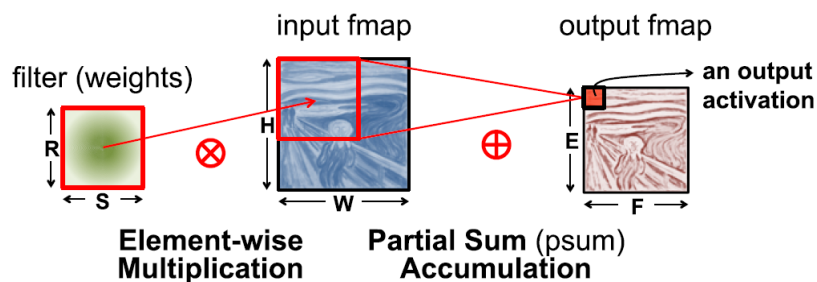


Abb. 2.8: 2D-Faltung in einem *Convolutional Neural Network* nach [SCYE17]

Der Filter wird auf das gesamte Bildmaterial angewendet, wobei die Größe und Schrittweite des Filters frei wählbar ist. Da sich Pixel häufig aus mehreren Datenwerten ergeben (bspw. RGB), können auch eben so viele Eingangsdaten (*Input Feature Maps*) existieren. Diese werden als *Channel* bezeichnet und führen zu ebenso vielen Filtern und *Output Feature Maps*. Häufig wird die zweidimensionale Faltung (2D) über mehrere *Channel* auch als dreidimensional (3D) bezeichnet. Die Daten der *Feature Maps* entsprechen den Ein- bzw. Ausgaben der Neuronen und die Werte der Filtermatrix entsprechen den Gewichten (*Weights*). Auf die Daten der *Feature Maps* wird ebenfalls die entsprechende Aktivierungsfunktion (*ReLU*) angewendet. Weitere optionale Verarbeitungsschritte stellen die *Normalization* und das *Pooling* dar (Abb. 2.7). Die *Normalization* wird benötigt, da sich die Verteilung der Ausgabewerte beim Durchlauf der Daten verändert und angepasst werden muss, um das Training des Netzwerks zu vereinfachen. In der Regel kommt hierbei ein Verfahren namens *Batch Normalization* zum Einsatz, dass im Gegensatz zur *Local Response Normalization* direkt zwischen dem *Convolutional Layer* und der Aktivierungsfunktion durchgeführt wird. Das *Pooling* wird eingesetzt, um die Größe der Ausgabedaten zu verringern, indem mehrere benachbarte Datenwerte (Pixel) zu einem Datenwert zusammengefasst werden. So kann bspw. eine 4x4 Pixelmatrix in einer 2x2 Pixelmatrix resultieren, wenn jeweils vier benachbarte Datenwerte durch das *Pooling* zusammengefasst werden. Die neuen Werte ergeben sich häufig aus dem größten Datenwert (*Max Pooling*) oder dem Durchschnittswert (*Average Pooling*). In einem CNN bestehen die letzten *Hidden Layer* meist aus *Fully Connected Layern*, die dazu genutzt werden, die Klassifikation auf Basis der Ergebnisse der *Convolutional Layer* durchzuführen. Das Training von CNNs funktioniert nahezu identisch wie bei allen anderen DNNs und nutzt ebenfalls das Verfahren der *Backpropagation*. Der Vorteil eines CNN liegt vor allem darin begründet, dass die Filter eines *Convolutional Layer* nicht fest definiert sind und während des Trainings angepasst werden. In einem CNN wird das Verfahren der *Faltung* und die Lernfähigkeit eines DNN kombiniert. Ein solches System ist in der Lage, den richtigen Filter für die Verarbeitung der während des Trainings genutzten Bilddaten zu erlernen, und mit dessen Hilfe *High Level Features* aus dem Bildmaterial abzuleiten. [Vor19] [SCYE17]

3 Technologieübersicht

3.1 FPGA

Der Begriff *Field Programmable Gate Array* (FPGA) bezeichnet einen IC, dessen integrierte Digitalschaltung entsprechend der benötigten Funktionen und Anwendungen frei konfiguriert werden kann. Die digitale Schaltung kann dabei direkt vom Anwender im Feld (*Field*) implementiert und auch nach der Implementierung noch geändert werden. FPGAs stellen eine Weiterentwicklung der bekannten PLDs (*Programmable Logic Device*) bzw. CPLDs (*Complex Programmable Logic Device*) dar und verfügen im Gegensatz zu diesen nicht über eine zentrale Schaltmatrix mit angeschlossenen PAL oder PLA Strukturen, sondern basieren auf Logikzellen, die in einer regelmäßigen Matrix Anordnung (*Array*) platziert sind (siehe Abb. 3.1). [KB13], [SN16]

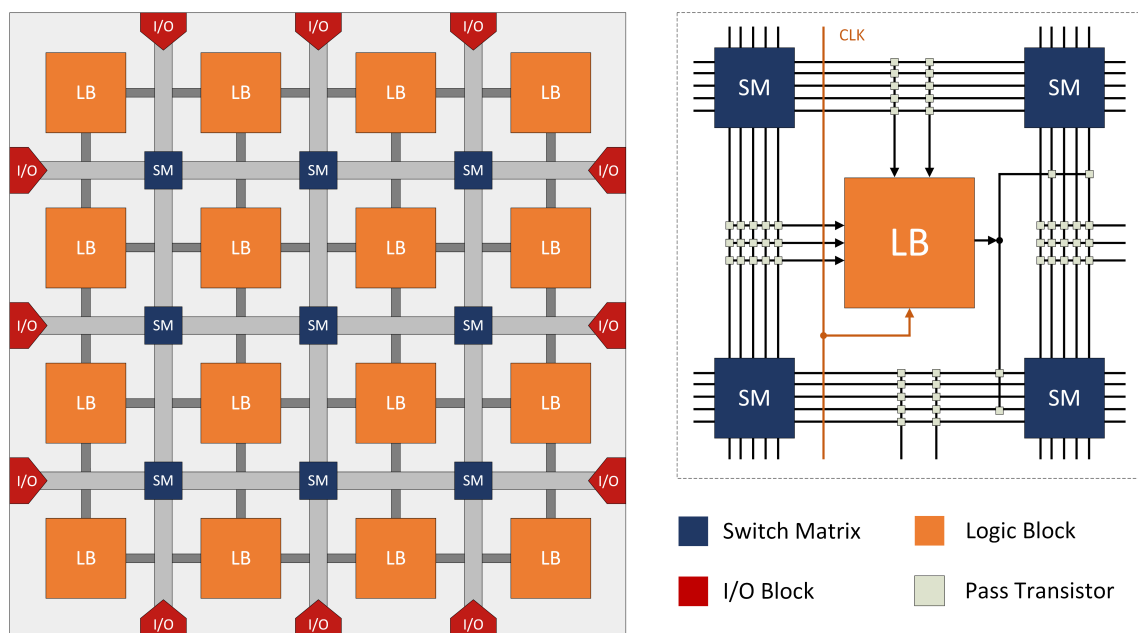


Abb. 3.1: Vereinfachtes Modell eines FPGA-Bausteins nach [SN16]

Die Logikzellen basieren im Kern auf einem kombinatorischen Anteil mit dem die Gatterfunktionen realisiert werden, einem Register-Anteil, der meist aus einem 1-Bit breiten D-Flip-Flop besteht, sowie einem Ausgangsmultiplexer, der zur Selektierung der beiden Anteile dient (Abb. 3.2). Das D-Flip-Flop ermöglicht es die Ausgangswerte des kombinatorischen Anteils zwischenspeichern bzw. Verzögerungszeiten

zu implementieren. Für die Umsetzung des kombinatorischen Anteils in Logikzellen haben sich zwei grundlegende Ansätze etabliert. Zum einen der Einsatz von Multiplexern und zum anderen der Einsatz von sogenannten *Look-Up-Tabellen* (LUT). Beim Einsatz von Multiplexern werden auf Basis der booleschen Schaltalgebra die benötigten Logikfunktionen über die programmierbare Verschaltung der verschiedenen Multiplexer implementiert. Hierbei werden die Flash- und die Antifuse-Technologie genutzt, wobei durch die Verwendung der zuletzt genannten die Reprogrammierbarkeit des FPGA und somit eine erneute Anpassung der implementierten Schaltungen entfällt. Logikzellen in FPGAs, welche auf der SRAM-Technologie basieren, verwenden die bereits angesprochenen LUTs. Diese entsprechen im Grunde einem kleinen 1-Bit breiten SRAM Speicher, dessen Speicherplätze über die Eingänge des Schreibdecoders angesteuert werden. Dies ermöglicht die Abspeicherung der entsprechenden logischen Funktion als einfache Wahrheitstabelle in einem LUT. LUTs verfügen je nach Hersteller und FPGA Familie typischerweise über 4 bzw. 6 Eingänge. Die einzelnen Logikzellen sind meist in größeren Basiszellen wie den *Configurable Logic Blocks* (CLB) bei Xilinx, den *Programmable Function Units* (PFU) bei Lattice, den *Logic Array Blocks* (LAB) bei Altera/Intel oder den *Central Programming Elements* (CPE) bei Cologne Chip zusammengefasst. Die Basiszellen verfügen in der Regel über mehrere Flip-Flops und machen FPGAs daher auch ideal für registerintensive Schaltungen wie Mikroprozessor- oder Signalverarbeitungssysteme. [KB13], [SN16]

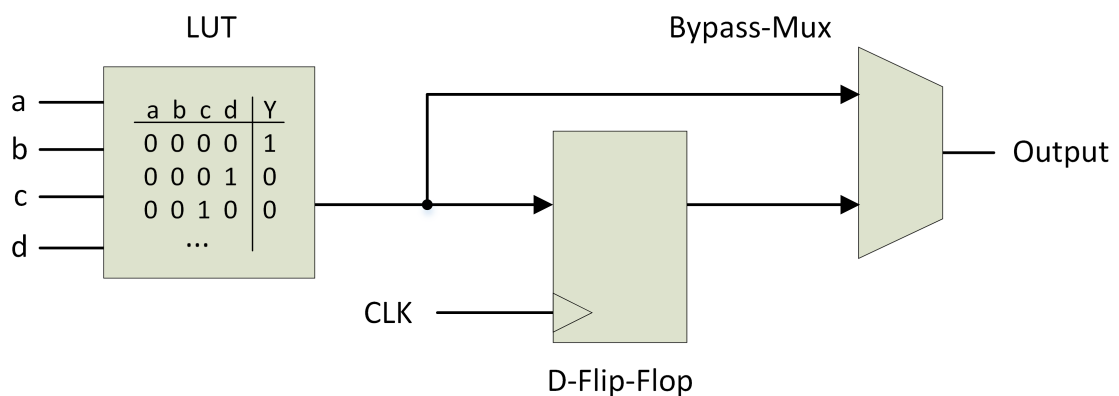


Abb. 3.2: FPGA Logikzelle mit LUT nach [SN16]

Da zur Erzeugung komplexer Schaltungen mehrere Logikzellen miteinander verschaltet werden müssen, ergibt sich innerhalb des FPGAs ein entsprechend hoher Verdrahtungsaufwand, der durch eine hierarchische und segmentierte Verbindungsarchitektur gelöst wird. Dazu werden Logikzellen zunächst innerhalb einer Basiszelle

über lokale Verbindungen verschaltet, bevor die Basiszellen selbst an ein Gitter aus horizontalen und vertikalen Leitungen unterschiedlicher Länge angeschlossen werden. Die Verknüpfung der horizontalen und vertikalen Leitungen im Gitter erfolgt anschließend durch die vorhandenen Schaltmatrizen bzw. *Switchboxes* (Abb. 3.1). Alle angesprochenen Verbindungen zur Erzeugung einer Schaltung sind entsprechend frei programmierbar. Für die Kommunikation mit der Außenwelt sind in FPGAs I/O-Blöcke vorhanden, deren Pads über Bonddrähte mit den Pins des FPGA-Gehäuses verbunden sind. Die I/O-Blöcke können ebenfalls über das Gitter aus Verbindungsleitungen angesteuert und mit den Basiszellen verbunden werden. Zusätzlich verfügen FPGAs teilweise über herstellerspezifische Arithmetikblöcke, wie etwa den DSP48 Slice von Xilinx, deren Architektur auf die Implementierung von Rechenschaltungen bzw. Anwendungen der digitalen Signalverarbeitung ausgelegt ist. Dadurch können Designs, die von diesen Blöcken Gebrauch machen, eine deutlich höhere Durchsatzrate und Taktfrequenz erreichen als dies durch die Verschaltung einzelner Basiszellen möglich wäre. [KB13], [SN16]

Die Programmierung eines FPGA bzw. die Schaltungsentwicklung erfolgt in der Regel in einer Hardwarebeschreibungssprache (HDL) wie VHDL oder Verilog. Mit diesen HDLs können ganze digitale Systeme und Schaltungen beschrieben werden und später mithilfe einer *Synthese* in eine entsprechende Hardwareschaltung übersetzt werden. Dabei muss zwischen einer herstellerunabhängigen Umsetzung auf der Register Transfer Ebene (RTL) durch standardisierte digitale Bauteile und einer herstellerabhängigen Abbildung der Schaltung auf die Architektur des entsprechenden FPGAs unterscheiden werden. Für die Abbildung der Schaltung auf die FPGA-Architektur und die anschließende Implementierung werden vom Hersteller in der Regel entsprechende Entwicklungsumgebungen zur Verfügung gestellt. Diese bieten neben einer Verhaltenssimulation auch eine Zeitsimulation auf Basis der hardware spezifischen Verzögerungszeiten nach der Abbildung einer Schaltung auf die FPGA Architektur (*Place & Route*) an. [KB13], [SN16]

Der große Vorteil von FPGAs gegenüber der direkten Entwicklung einer Schaltung als *Application-Specific Integrated Circuit* (ASIC) liegt vor allem in der Reprogrammierbarkeit und der damit verbunden möglichen Anpassung der digitalen Schaltung begründet. Dies bedeutet, dass eine digitale Schaltung auch Jahre nach der Veröffentlichung des Gesamtsystems noch angepasst werden kann. Damit ergibt sich, für Hersteller und Anwender gleichermaßen, die Möglichkeit auf zukünftige Anforderungen oder notwendige Änderungen reagieren zu können, ohne ein entsprechendes *Redesign* des gefertigten Chips oder gar des Gesamtsystems zu veranlassen. In

der Entwicklung bieten FPGAs zudem kürzere Entwicklungszyklen und damit einhergehende geringere Entwicklungskosten gegenüber ASICs an. Allerdings besitzen FPGAs im Gegensatz zu ASICs auch einige entscheidende Schwächen, die sie für die Industrie unattraktiv machen. So sind ASICs, da ihre integrierten Hardwareschaltungen nicht wie bei FPGAs auf eine bestehende Architektur abgebildet werden müssen, sondern spezifisch für einen Anwendungsfall entwickelt werden, im Vorteil wenn es um den Leistungsbedarf, die Logikdichte oder die maximale Taktfrequenz geht. Die Wahl zwischen dem Einsatz von FPGAs und ASICs ist auch immer eine Frage des Anwendungsgebietes und der Entwicklungskosten. Diese fallen bei ASICs entsprechend höher aus, weshalb sich diese nur bei entsprechenden Stückzahlen oder bei der nötigen Integration spezifischer analoger Schaltungsanteile eignen. Es besteht jedoch die Möglichkeit FPGA basierte Schaltungen mit überschaubarem Aufwand in einen ASIC Designansatz zu überführen. Denn im Bereich der ASIC Entwicklung erfolgt die Entwicklung von digitalen Schaltungen ebenfalls in HDLs wie VHDL oder Verilog, wobei diese in der Regel, das sogenannte *Frontend* darstellt. [KB13], [SN16]

3.1.1 Hardware Description Languages

Im Rahmen der Hardwarebeschreibungssprachen existieren zwei weit verbreitete und untereinander konkurrierende Sprachen namens VHDL und Verilog. Die Hardwarebeschreibungssprache VHDL ist Anfang der 80er Jahre durch die VHSIC-Initiative des amerikanischen Verteidigungsministeriums entwickelt worden. Die Abkürzung VHDL leitet sich aus der Initiative *Very High Speed Integrated Circuit* (VHSIC) und der Abkürzung für *Hardware Description Language* (HDL) her. Ziel war es eine einheitliche Beschreibungssprache zu schaffen, mit der die Funktion und Struktur komplexer Schaltungen, unabhängig von ihrer Technologie oder Entwurfsmethodik, beschrieben und simuliert werden kann. Denn bis zu diesem Zeitpunkt wurden Schaltungen von unterschiedlichen Herstellern oft in zueinander inkompatiblen und firmenspezifischen Beschreibungssprachen entwickelt oder nur unzureichend dokumentiert. Dieser Umstand führte zu entsprechenden Problemen, weshalb die Firmen Intermetrics, IBM und Texas Instruments damit beauftragt wurden VHDL zu entwickeln. Als Ausgangspunkt diente hierbei die objekt-basierte Programmiersprache ADA, an der sich die Entwicklung stark orientierte und dazu führte, dass VHDL-Beschreibungen einen deutlich höheren Schreibaufwand erfordern. Die Hardwarebeschreibungssprache ist 1987 durch das *Institute of Electrical and Electronic Engineers* (IEEE) unter der Bezeichnung IEEE 1076-1987 als Standard veröffentlicht und

1993 unter der Bezeichnung IEEE 1076-1993 überarbeitet worden [KB13]. Die HDL Verilog wurde 1983 mit ähnlichen Ambitionen entwickelt und hatte ebenfalls zum Ziel die Entwicklung, sowie die Simulation und Analyse von digitalen Schaltungen zu vereinfachen. Zunächst war Verilog noch alleiniges Eigentum der Firma Cadence, wurde jedoch im Jahr 1995 für die Verwendung durch Dritte freigegeben und noch im selben Jahr als IEEE Standard 1364-1995 (Verilog-95) spezifiziert. Es folgten die zwei Erweiterungen Verilog-2001 und Verilog-2005, welche den IEEE Standard um wichtige Bestandteile ergänzten und eine angepasste Formatierung (*ANSI-C Style*) ermöglichten. Die HDL ist heutzutage im nordamerikanischen Raum weit verbreitet und steht in direkter Konkurrenz zu VHDL, die vor allem im europäischen Raum Verwendung findet. Der Trend geht jedoch in den letzten Jahren auch hier zu Lande in Richtung Verilog, was nicht zuletzt den amerikanischen Firmen und ihren Entwicklungstools geschuldet ist. Auch Erweiterungen wie Verilog-AMS, welche das Design von Schaltungen im Bereich *Analog and Mixed-Signal* (AMS) erlaubt oder die weiterentwickelte *Hardware Description and Verification Language* (HDVL) SystemVerilog haben diesen Trend bestärkt. So bildet SystemVerilog zudem die Basis für die *Universal Verification Methodology* (UVM), einer heute bevorzugten Praxis für die Verifikation von Schaltungen auf höheren Abstraktionsebenen. Generell verstehen sich VHDL und Verilog als Beschreibungssprachen zur Hardware-Modellierung und stehen damit im klaren Gegensatz zu Hochsprachen für die Softwareentwicklung wie C oder Java, die lediglich ein sequenziell ausgeführtes Programm für ein bereits vorhandenes Hardwaresystem beschreiben. Die Herangehensweise bei der Entwicklung entspricht daher nicht der Umsetzung eines Algorithmus zur Datenverarbeitung, sondern zielt auf das Design einer eigenen Hardwareschaltung auf RTL-Ebene ab, mit der auch eine parallele Verarbeitung möglich ist. Die Beschreibung einer Schaltung erfolgt dabei immer in entsprechenden Komponenten die in VHDL als *Entity* und in Verilog als *Module* bezeichnet werden. Diese definieren die vorhandenen Schnittstellen der Schaltung (*Ports*), die über einen Datentyp verfügen und als Eingangs-, Ausgangs- oder bidirektionale Schnittstelle definiert werden (Quellcode 3.1 und 3.2).

```
1  entity Multiplexer is
2      port( X,Y: in std_logic;    -- Eingangsdaten
3          S: in std_logic;       -- Datenauswahl
4          A: out std_logic);     -- Ausgangsdaten
5  end Multiplexer;
```

Quellcode 3.1: Port-Beschreibung für einen 2:1 Multiplexer in VHDL

```
1  module Multiplexer(           // ANSI-C Style
2      input  wire X, Y,         // Eingangsdaten
```

```

3   input wire S,      // Datenauswahl
4   output reg A      // Ausgangsdaten
5 );

```

Quellcode 3.2: Port-Beschreibung für einen 2:1 Multiplexer in Verilog

Weiterhin kann durch Verhaltens- und Strukturbeschreibungen die Funktion der Schaltung definiert werden (Quellcode 3.3 und 3.4). Dabei kommen parallele Anweisungen und sowie Konstrukte für die Ausführung von sequentiellen Anweisungen (*Process* bzw. *always*-Block) zum Einsatz, die neben den *Ports* auch auf definierte Signale (*signal* bzw. *wire*) oder Variablen (*variable* bzw. *reg*) zugreifen können. Diese stellen zum einen die Leitungen der Schaltung dar und werden zum anderen genutzt, um Speicherelemente zu beschreiben. Bei VHDL erfolgt die funktionale Beschreibung innerhalb eines zugehörigen Blocks mit der Bezeichnung *architecture*.

```

1 architecture Funktion_Multiplexer is
2     begin
3         A <= X when (S = '0') else B;    -- Schaltungsfunktion
4     end architecture Funktion;

```

Quellcode 3.3: Multiplexer Funktion in VHDL

```

1     ...
2     assign A = (S == 1'b1) ? X : Y // Schaltungsfunktion
3 endmodule

```

Quellcode 3.4: Multiplexer Funktion in Verilog

Bei komplexeren Systemen ermöglichen HDLs, durch ihre komponentenbasierte Darstellung, das Gesamtsystem hierarchisch zu organisieren und aus kleineren Schaltungen zusammensetzen. Dazu können Schaltungen durch Strukturbeschreibungen instanziiert werden. Die übergeordnete Schaltung stellt dabei das sogenannte *Top-Level* dar. Um die Funktion einer Schaltung zu testen, wird zusätzlich eine *Testbench* benötigt, welche als *Top-Level* für die zu testende Schaltung fungiert und genutzt wird, um die benötigte Testumgebung der Schaltung aufzubauen. Durch diese lassen sich der Schaltung dann entsprechende Stimulussignale zuführen, deren Reaktionen mithilfe einer Simulationssoftware analysiert und kontrolliert werden kann. Wie in Kap. 3.1 bereits angesprochen, können HDL Beschreibungen durch eine entsprechende Synthese und Implementierung auf die Architektur eines FPGA abgebildet werden. Bei der Schaltungsentwicklung ist jedoch zu beachten, dass die HDLs bei ihrer Entwicklung ursprünglich für die Simulation und nicht für die Erzeugung von Schal-

tungen entwickelt worden sind. Daher lassen sich zwar alle Konstrukte und Datentypen ausnahmslos simulieren, aber nur ein gewisser Sprachbestandteil gilt als synthetisierbar und lässt sich von den Entwicklungswerkzeugen in eine Hardwareschaltung überführen. Diese Einschränkung gilt jedoch nur für die HDL-Beschreibungen, die auch Teil der Schaltung sind und hat keinen Einfluss auf die in der Simulation eingesetzten *Testbenches*, die auf den vollen HDL Umfang zurückgreifen können. Für eine genaue Beschreibung der beiden HDLs und ihrer Konzepte sei auf die Quellen [KB13] und [Hop06] verwiesen.

3.1.2 Einsatz in KI-Anwendungen

Der Einsatz von FPGAs im Bereich der Künstlichen Intelligenz, genauer für die hardwaretechnische Umsetzung von DNNs, stellt für Firmen und Forscher eine deutliche Alternative zu anderen Hardwareplattformen wie *Graphic Processing Units* (GPU) oder *Central Processing Units* (CPU) dar. Betrachtet man die grundlegenden Operationen innerhalb eines DNNs, setzen sich diese im Kern aus parallelisierbaren *Multiply and Accumulate* Operationen (MAC) zusammen. Die Parallelisierung stellt dabei eine Schlüsselkomponente für die Performance dar und hat dafür gesorgt, dass sich GPUs und FPGAs aufgrund ihrer parallelisierbaren Verarbeitung als Standard für leistungsstarke KI-Systeme etabliert haben. Eine GPU besitzt im Gegensatz zu einem FPGA über eine fest-verdrahtete Architektur aus mehreren *Arithmetic Logic Units* (ALU) bzw. *Cores*. Diese sind in einer *Array*-Struktur angeordnet, aber verfügen untereinander über keine direkte Verbindung. Sie werden zentral gesteuert und greifen auf einen gemeinsamen Speicher zu. Die parallele Datenverarbeitung wird durch *Single Instruction Multiple Threads* (SIMT) realisiert, wobei die Operationen der entsprechenden Anwendung nach Möglichkeit in parallelisierbare *Threads* aufgeteilt werden. Die Verarbeitung der Daten kann jedoch nicht immer effizient auf die interne Struktur der GPU abgebildet werden und ist zudem an ein entsprechendes *Instruction Set* gebunden [MF17]. Eine solche Systemarchitektur (CPU/GPU) folgt einem Software-basierten Ansatz und wird als *Temporal Architecture* bezeichnet (Abb. 3.3) [SCYE17, S.13]. Den Gegensatz dazu bildet die *Spatial Architecture*, die dem Ansatz des *Dataflow Processing* folgt und die FPGA- oder ASIC-basierten Architekturen umfasst. Hierbei werden die Recheneinheiten (ALUs) für die Verarbeitung der Daten untereinander verschaltet und können über einen eigenen Speicher (Register) sowie über eine eigene Kontrolllogik verfügen (Abb. 3.3).

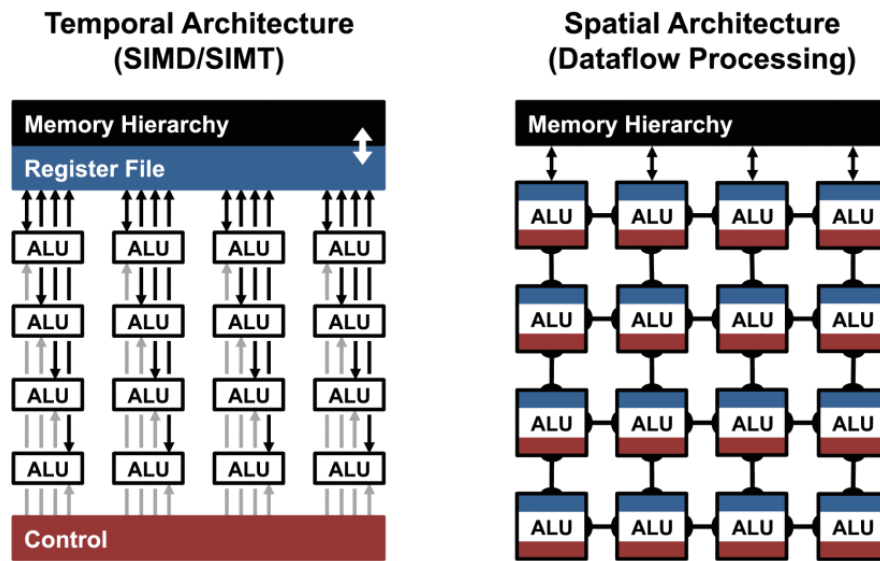


Abb. 3.3: *Temporal Architecture* und *Spatial Architecture* nach [SCYE17]

Die Anwendung von *Spatial Architectures* ermöglicht es, im Gegensatz zu *Temporal Architectures*, die vordefinierten Strukturen und Datenbreiten eines DNN auf die Hardware zu übertragen und im Fall von FPGAs flexibel anzupassen. Eine gesteigerte Effizienz im Energieverbrauch und im Datendurchsatz wird dabei vor allem durch die Anpassung der Speicherstrukturen erreicht. Da eine MAC Operation drei Speicherzugriffe für die Eingangsdaten (*Weight*, *Activation*, *Partial Sum*) und einen Speicherschrieb für die Ausgangsdaten (*Updated Partial Sum*) erfordert, ist die Art des verwendeten Speichers ausschlaggebend für die Energieeffizienz (Abb. 3.4). Es wird daher versucht, den Zugriff auf externe Speicher (DRAM) durch den Einsatz eines globalen Buffers, lokale Register und interne Verbindungen so weit wie möglich zu minimieren. Dies bringt Speicherstrukturen und MAC Operation dichter zusammen und resultiert ebenfalls in verringerten Zugriffszeiten für die benötigten Daten. [SCYE17, S.15]

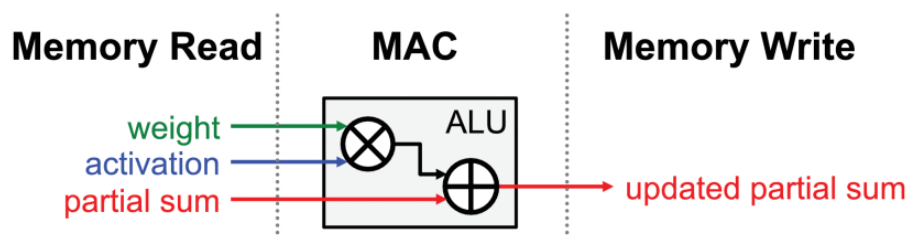


Abb. 3.4: Speicherzugriff einer MAC Operation nach [SCYE17]

Damit die energieeffizienten lokalen Speicherstrukturen effektiv genutzt werden können, müssen Daten, die bereits aus dem externen Speicher (DRAM) geladen worden sind, so oft wie möglich wiederverwendet werden. Für die Optimierung des Speicherzugriffs haben sich folgende vier Verfahren bzw. Strukturen für die Datenverarbeitung heraus kristallisiert [SCYE17, S.16].

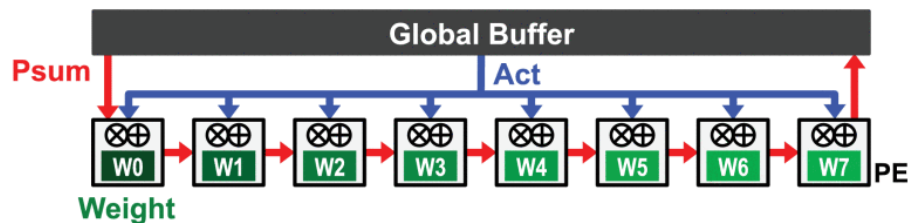


Abb. 3.5: Struktur für *Weight Stationary* nach [SCYE17]

Beim *Weight Stationary* ist die Datenverarbeitung darauf ausgelegt die Gewichte (*Weights*) in den Registern der MAC-Einheiten abzulegen (W0 - W7) und so oft wie möglich für die Multiplikationen wieder zu verwenden (Abb. 3.5). Die notwendigen Eingangsdaten (*Activations*) werden durch den globalen Buffer jeweils an die Recheneinheiten für die Multiplikation angelegt. Die partielle Summe wird anschließend über deren Ergebniswerte gebildet und wieder zurück in den globalen Buffer geschrieben. Dieses Vorgehen eignet sich besonders für den Einsatz in CNNs, da in diesen Netzen verschiedene Bildausschnitte der *Input Feature Maps* mit den selben Filterwerten (*Weights*) multipliziert werden.

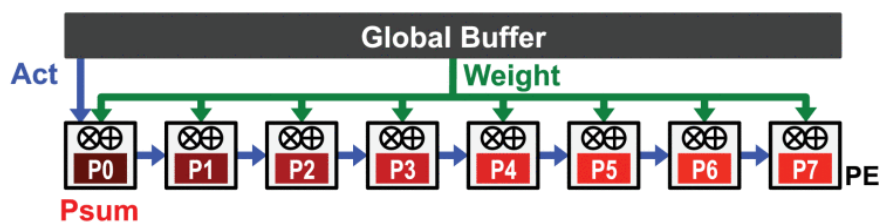


Abb. 3.6: Struktur für *Output Stationary* nach [SCYE17]

Die Verarbeitung nach *Output Stationary* versucht hingegen die Speicherzugriffe für die partiellen Summen zu reduzieren und legt diese in den Registern der Recheneinheiten ab (Abb. 3.6). Hierbei wird die Berechnung der partiellen Summen für einzelne Eingangsdaten (*Activations*) mit unterschiedlichen Gewichten (*Weights*)

durchgeführt. Die Gewichte und Eingangsdaten werden durch den globalen Buffer bereitgestellt. Für die Struktur bei *Output Stationary* existieren verschiedene Ansätze, die von der gezeigten Struktur abweichen können. [SCYE17, S.17]

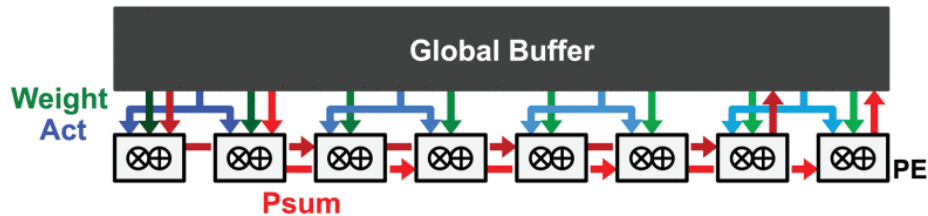


Abb. 3.7: Struktur für *No Local Reuse* nach [SCYE17]

Der Ansatz des *No Local Reuse* verzichtet auf die Nutzung von lokalen Registern und verlagert die interne Speicherstruktur vollständig in den globalen Buffer. Dabei stehen keine stationären Datenwerte mehr für die Berechnung in den Registern zur Verfügung und müssen durch den globalen Buffer bereitgestellt werden. Dies resultiert in einem deutlich erhöhten Datenaufkommen für alle notwendigen Daten, wobei die Weitergabe der partiellen Summen zwischen den Recheneinheiten erfolgt (Abb. 3.7). [SCYE17, S.17]

Einen Ansatz, der mit Bezug auf CNNs versucht die Wiederverwendbarkeit der gesamten Daten zu steigern, stellt das *Row Stationary* dar. Bei diesem Verfahren führt eine Recheneinheit eine eindimensionale Faltung für die Zeile bzw. Reihe einer zweidimensionalen *Input Feature Map* durch (Abb. 3.8). Die Recheneinheit verfügt dazu über ein Register für die Gewichte (*Weight*), die Eingangsdaten (*Activation*) und die partielle Zwischensumme. Während der Faltung wird der Filter mit einer bestimmten Schrittweite über die *Feature Map* bewegt (*Sliding Window*) und mit den korrespondierenden Werten multipliziert, sowie anschließend aufsummiert. Die Filterwerte für die Multiplikation bleiben wie schon beim *Weight Stationary* Verfahren unverändert und werden erneut verwendet. Ein Teil der Eingangsdaten kann je nach Schrittweite des Filters und der daraus resultierenden Überlappung der Daten ebenfalls wiederverwendet werden. Im gezeigten Beispiel besitzt der Filter eine Breite von 3 und eine Schrittweite von 1, wodurch die Datenwerte der *Feature Map* (bspw. c) insgesamt drei mal wiederverwendet werden können (Abb. 3.8). Um die Ergebnisse für eine zweidimensionale Faltung zu erhalten, können die Ergebnisse der eindimensionalen Faltung mit den Ergebnissen der anderen eindimensionalen Faltungen aufsummiert werden. Diese bilden dann die erste Zeile einer zweidimensionalen

Output Feature Map. Die Möglichkeit zur Wiederverwendbarkeit steigt dabei mit der Anzahl der *Feature Maps* der verschiedenen Filter und der vorhandenen Kanäle (*Channel*). Für eine entsprechende Beschreibung siehe [SCYE17, S.17].

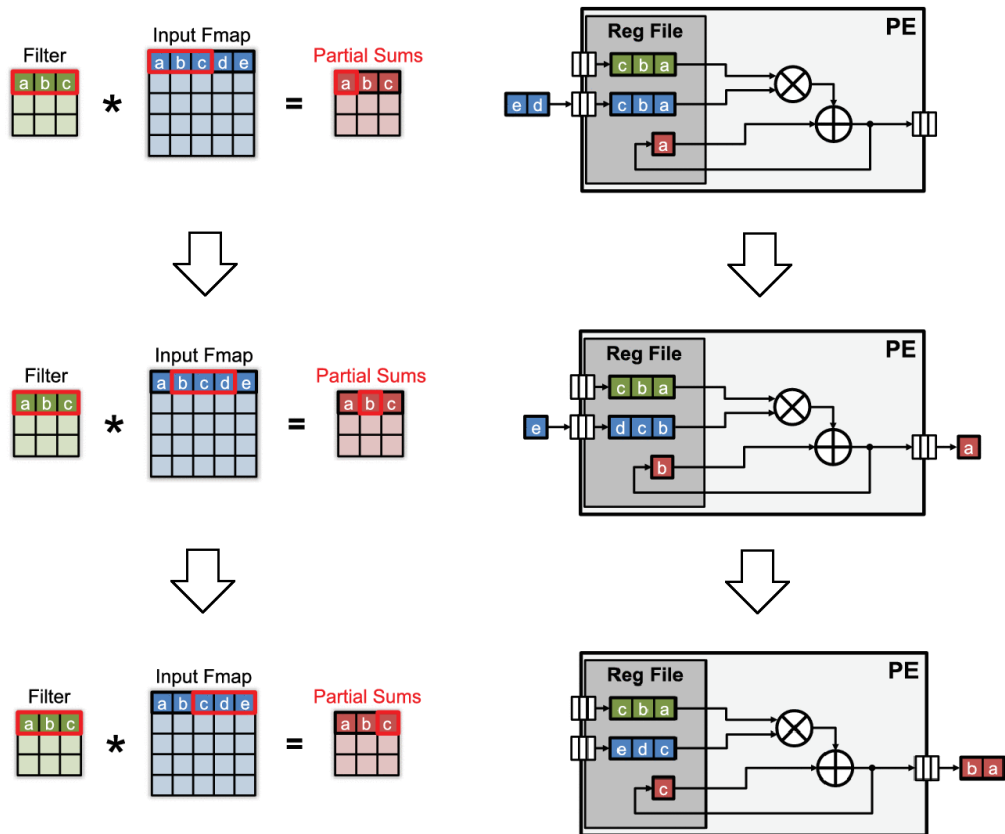


Abb. 3.8: Ablauf einer eindimensionalen Faltung nach [SCYE17]

Obwohl die Architektur eines ASICs in der Entwicklung optimal auf das verwendete DNN angepasst werden kann, was in schnelleren Operationen und einer noch geringeren Leistungsaufnahme als bei FPGAs resultiert, sind sie auf Grund der langen Entwicklungszeit und der unflexiblen Struktur für DNNs uninteressant [GFC18, S.2]. Die Rekonfigurierbarkeit von FPGAs ermöglicht es hingegen DNNs im Fall von neuen Entwicklungen oder nach Praxistests anzupassen. Dies ist vor allem für einen sich so rasch entwickelnden Technologiebereich wie *Deep Learning* eine zwingende Voraussetzung. FPGAs stellen allgemein einen guten Kompromiss zwischen ASICs und den *Temporal Architectures* in Form von GPUs dar. Beide Technologien (FPGA/GPU) bieten eine vergleichbare Performance, besitzen in einigen Bereichen

jedoch Vor- und Nachteile, wie etwa die geringere Leistungsaufnahme bei FPGAs. Ein Vorteil von GPUs gegenüber FPGAs stellt die garantiert hohe Präzision bei Berechnungen (64/32-Bit floating point) dar und hat dafür gesorgt, dass GPUs häufig für das Training von DNNs bevorzugt werden, da dieses von einer erhöhten Präzision profitiert. Im Fall der Inferenz hat sich jedoch gezeigt, dass eine hohe Präzision nicht zwingend nötig ist und viele Optimierungsverfahren auf eine verringerte Präzision abzielen. Im Zusammenhang mit GPUs bringt die Reduzierung der Präzision mit Bezug auf die Reduzierung der Hardware jedoch generell keine Vorteile und resultiert eher in einer ineffizienten Auslastung. Auch wenn GPU-Hersteller diesen Trend erkannt haben und auch reduzierte Datentypen (16-Bit floating point, 16/8-Bit Integer) anbieten, lassen diese eine flexible Anpassung vermissen und resultieren mitunter in teuren Umrüstungen auf neue GPU-Hardware. Die Optimierungen bei DNNs reichen bis hin zu sogenannten *Binary Neural Nets* (BNN) in denen die Daten (*Activations*) und Gewichte (*Weights*) auf binäre Werte reduziert werden. Hierbei werden die MAC Operationen durch XNOR-Verbindungen ersetzt, welche durch GPUs nicht mehr effizient umgesetzt werden können [MF17, S.5]. Ein weiterer Aspekt den FPGA-Hersteller mit ihren Produkten schon länger fokussieren, stellt der Einsatz in sicherheitskritischen Anwendungen dar. Im Gegensatz zu GPU-Herstellern, deren Produkte für den Konsumentenmarkt keine strengen Sicherheitskriterien für die Funktionalität erfüllen müssen, werden FPGAs schon länger für sicherheitskritische Anwendungen entwickelt und verfügen bereits über entsprechend hohe Sicherheitszertifizierungen. Daher verfügen FPGA-Hersteller in diesem Bereich noch über einen Vorsprung, der sich auf ihre früheren Erfahrung in der Entwicklung stützt [MF17, S.8, S.13]. Dennoch werden GPUs seit langem im Bereich der DNNs eingesetzt und erfreuen sich auch wegen der Software-basierten Herangehensweise durch *Frameworks* wie CUDA großer Popularität. Im Bereich von FPGAs sehen sich Entwickler und Programmierer zwangsläufig mit HDLs wie VHDL, Verilog oder SystemVerilog konfrontiert. Diese stellen häufig eine nicht zu unterschätzende Hürde da und erfordern von Programmierern ein radikales Umdenken und eine andere Methodik. Die Bestrebungen der FPGA-Hersteller gehen daher in Richtung *High-Level Synthese* (HLS) um den Einsatz von FPGAs als *Hardware Accelerator* interessanter zu machen. So haben Xilinx und Intel (ehem. Altera) bereits 2013 damit begonnen eigene *Software Development Kits* (SDKs) auf Basis von OpenCL, einem standardisierten open-source Framework, bereit zu stellen. [LTA16, S.4] [GFC18, S.7]

Ein Beispiel für den erfolgreichen Einsatz von FPGAs in kommerziellen KI-Systemen liefert das südkoreanische Unternehmen SK Telecom [Eat19a]. Das Unternehmen setzt für seinen digitalen Heim-Assistenten NUGU auf eine sprachgesteuerte Kom-

munikation und soll in der Lage sein, in flüssiger Sprache zu kommunizieren. Dies umfasst auch die Erkennung von sprachlichen Merkmalen wie Dialekten, Akzenten oder Sprachmelodien. Für die Sprachverarbeitung hat das Unternehmen ein auf die Inferenz ausgelegtes KI-System namens *AI Inference Accelerator* (AIX) entwickelt. Die Sprachverarbeitung erfolgt dabei in den Datenzentren von SK Telecom. Das KI-System gliedert sich in mehrere kundenspezifische *Network Processing Units* (NPU) und setzt für den Aufbau der DNNs auf die Beschleunigerkarten KCU1500 von Xilinx [Xil21]. Diese Karten verfügen über einen Kintex UltraScale FPGA (XCKU115) und sind für die Kommunikation mit einer PCI Express 3.0 Schnittstelle ausgestattet. Für die Umsetzung der notwendigen Berechnungen werden die integrierten DSP *Slices* der Kintex FPGA-Serie genutzt. Durch den Einsatz der FPGA-Technologie konnte das Unternehmen nach eigenen Angaben eine höhere Performance und Energieeffizienz als mit CPUs oder GPUs erzielen [Eat19a] [Eat19b].

3.2 PCI Express

Bei PCI Express bzw. PCIe handelt es sich um eine serielle Hochgeschwindigkeits-Schnittstelle, die als Nachfolger von PCI vorwiegend in Computer-Systemen zum Einsatz kommt. Im Gegensatz zu ihrem Vorgänger handelt es sich bei PCIe um ein serielles *Interface* für eine Punkt-zu-Punkt-Verbindung zwischen zwei Teilnehmern. Die Verbindung wird in diesem Fall als *Link* bezeichnet und kann aus mehreren *Lanes* bestehen. Diese weisen ihrerseits auf ein Leitungspaar zur differenziellen Signalübertragung (*Differential Signal Pair*) für beide Übertragungsrichtungen hin. Beim *Differential Signaling* ergeben sich die logischen Werte für die Übertragung aus der Differenz der beiden Signale D+ und D-, wodurch diese sehr robust gegenüber Störeinflüssen sind. Bei einer Differenz von 0 V befindet sich der *Transmitter* im sogenannten *Electrical Idle*. In diesem Zustand werden die Signale nah an der *Common Mode Voltage* V_{cm} gehalten. Der PCIe Standard ermöglicht in Gen1/Gen2 eine Übertragungsrate von 2,5 Gbit/s bis 5 Gbit/s pro *Lane*. Der Einsatz mehrerer *Lanes* (1x, 2x, 4x, 8x, 12x, x16, x32) kann die Bandbreite des *Links* dabei um ein Vielfaches steigern. Für die PCIe Übertragung ist keine eigene Taktleitung (*Common Clock*) vorhanden, da das Taktsignal in die Daten eingebettet ist (*Embedded Clock*) und vom Empfänger zurückgewonnen werden kann. Die PCIe Architektur lässt sich in den *Transaction Layer*, den *Data Link Layer* und den *Physical Layer* aufteilen (Abb. 3.9). [PCI09] [BASI12]

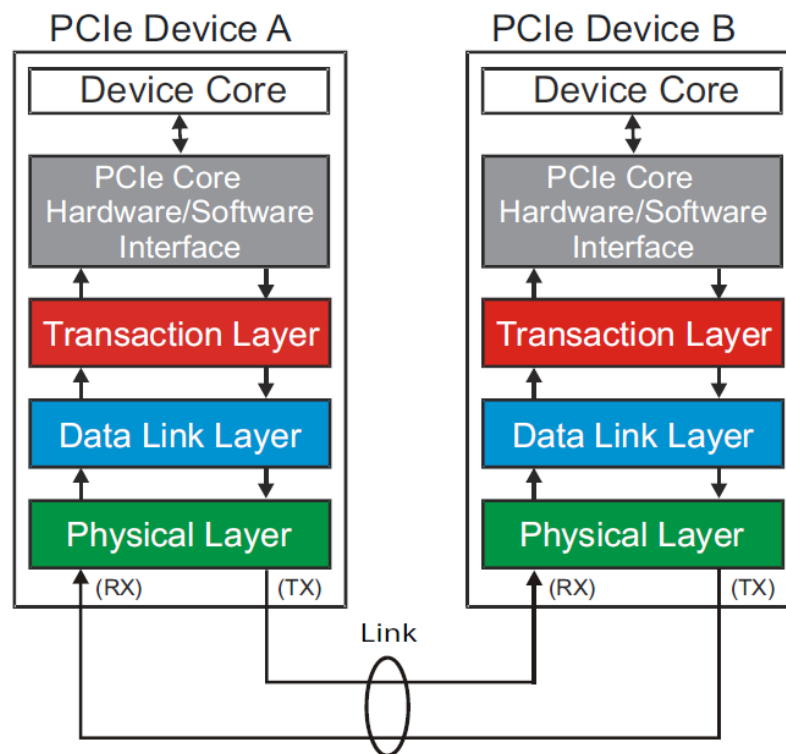


Abb. 3.9: Architektur der PCI Express *Device Layer* nach [BASI12]

3.2.1 Transaction Layer

Der *Transaction Layer* stellt die oberste Ebene der PCI Express *Device Layer* dar und kommuniziert direkt mit dem *Device Core* über ein entsprechendes *Software Interface* (*Software Layer*). Er zeigt sich für die Verarbeitung der eingehenden und ausgehenden Daten für den *Device Core* verantwortlich und organisiert diese in sogenannten *Transaction Layer Packets* (TLPs). Ein TLP besteht in der Regel aus einem *Header*, den entsprechenden Daten und einer ECRC-Summe (*End-to-End Cyclic Redundancy Check*). Durch den *Header* des TLP werden die unterschiedlichen Arten der Übertragungen wie etwa verschiedene Speicherzugriffe definiert. Die Kommunikation eines *Transaction Layers* findet immer mit dem *Transaction Layer* auf der anderen Seite des *Links* statt, wobei das TLP zunächst die beiden anderen PCI Express *Device Layer* durchläuft. Der *Transaction Layer* unterstützt eine *Flow Control* für die Datenübertragung, durch die unnötige Übertragungen verhindert werden sollen. Der Kern dieser Datenflusssteuerung basiert darauf, dass ein *Device* die Gegenseite in regelmäßigen Abständen über den vorhandenen Platz in den *Buffern* des *Transaction Layers* informiert. Für weitere Informationen über die Struktur und die Funktionen des *Transaction Layers* siehe [BASI12, S.59ff], [BASI12, S.226ff] oder

[PCI09, S.52ff].

3.2.2 Data Link Layer

Der *Data Link Layer* stellt das Bindeglied zwischen dem *Transaction Layer* und dem *Physical Layer* dar (Abb. 3.9). Seine Funktion ist es, die Kommunikation zwischen den beiden Teilnehmern auf Übertragungsfehler zu überprüfen und wenn nötig eine erneute Übertragung einzuleiten. Dazu nimmt der *Data Link Layer* die TLPs entgegen und fügt den zu übertragenden Daten eine *Sequence Number* und eine LCRC-Summe (*Link Cyclic Redundancy Check*) hinzu. Diese werden später wieder entfernt, so dass nur das TLP an den *Transaction Layer* weitergereicht wird. Die Daten werden für die Übertragung an den *Physical Layer* weitergeleitet. Empfängt der *Data Link Layer* eine Übertragung mit einer falschen *Sequence Number* oder liegt ein Fehler in der LCRC vor, beginnt dieser mit der Übertragung eines *Data Link Layer Packet* (DLLP) und fordert die Gegenseite auf, die Übertragung zu wiederholen. Dieses Vorgehen geschieht unabhängig vom *Transaction Layer* und spielt sich nur auf Ebene der *Data Link Layer* ab. Der *Data Link Layer* nimmt, neben der Fehlererkennung, noch andere Funktionen, wie etwa die Übertragung von DLLPs für die Angabe der *Buffer*-Auslastung (*Flow Control*), wahr. Für weitere Informationen über die Struktur und die Funktionen des *Data Link Layers* siehe [BASI12, S.72ff], [BASI12, S.364ff] oder [PCI09, S.151ff].

3.2.3 Physical Layer

Der *Physical Layer* stellt die unterste Schicht der PCI Express *Device Layer* dar und zeigt sich für die Übertragung der Daten über den *Link* verantwortlich. Dazu verfügt er über die notwendigen Schaltungsanteile auf Leitungsebene als auch über die Funktionslogik zur Verarbeitung der Sende- und Empfangsdaten. Für die Datenübertragung werden die Sendedaten um Kontrollsymbole ergänzt, die den Start und das Ende einer Übertragung kennzeichnen sollen. Diese werden beim Empfang der Daten durch den *Physical Layer* wieder entfernt. Besteht der *Link* aus mehreren *Lanes* werden die Sendedaten für die Übertragung gleichmäßig auf diese aufgeteilt. Dieser Vorgang wird als *Byte Striping* bezeichnet und erfolgt in umgekehrter Weise auch auf der Seite des Empfängers. Die nachfolgenden Operationen werden für die Sendedaten jeder einzelnen *Lane* durchgeführt. Um in der Übertragung sich wiederholender Muster auf der Sendeleitung und damit eine erhöhte *Electro-Magnetic Interference* (EMI) zu verhindern, werden die Sendedaten durch einen *Scrambler*

geschickt. Anschließend erfolgt eine 8b/10b Encodierung für die Bytes der Sendedaten, wodurch diese als 10-Bit übertragen werden. Der Einsatz dieses Leitungscodes gewährleistet einen entsprechenden Flankenwechsel für die Rückgewinnung des Taktsignals (*Clock Data Recovery*), sorgt für einen entsprechenden Gleichspannungsausgleich und erhöht zudem die Fehlererkennung auf der Seite des Empfängers. Auf der Empfangsseite verfügt der *Physical Layer* über ähnliche Komponenten zur Umkehr der durchgeführten Operationen. Diese werden durch die Funktionslogik für die *Clock Data Recovery* (CDR) und einen *Elastic Buffer* für den Ausgleich der auftretenden Takttoleranzen ergänzt [BASI12, S.397]. Der *Physical Layer* zeigt sich zudem für die Initialisierung und das Training des *Links* verantwortlich. Dieses wird intern durch die *Link Training and Status State Machine* (LTSSM) implementiert, welche auch den derzeitigen Zustand bzw. das Verhalten des *Physical Layers* definiert. Im Rahmen des Trainings werden verschiedene Konfigurationsparameter, wie etwa die Anzahl der *Lanes* oder die maximale Übertragungsrate, zwischen den beiden Teilnehmern bestimmt. Dabei kommen verschiedene Übertragungen namens *Ordered Sets* zum Einsatz, die auch im Rahmen anderer Funktionen des *Physical Layers* Verwendung finden. Diese Übertragungen finden nur auf Ebene der beiden *Physical Layer* statt und erfolgen unabhängig vom *Data Link Layer* und dem *Transaction Layer*. Für weitere Informationen über die Struktur und die Funktionen des *Physical Layers* siehe [BASI12, S.76ff], [BASI12, S.361ff] oder [PCI09, S.187ff].

3.3 PHY Interface for the PCIe Architecture

Das *PHY Interface for the PCI Express Architecture* (PIPE), ist eine von der Intel Corporation entwickelte Spezifikation, welche die Schnittstelle zwischen der eigentlichen PCI Express Logik und den analogen sowie digitalen Schaltungsanteilen zur Realisierung des *Links* definiert. Innerhalb des *Physical Layers* ordnet sich die Schnittstelle bzw. das Interface daher zwischen dem *Media Access Layer* (MAC) und dem *Physical Coding Sublayer* (PCS) ein (Abb. 3.10).

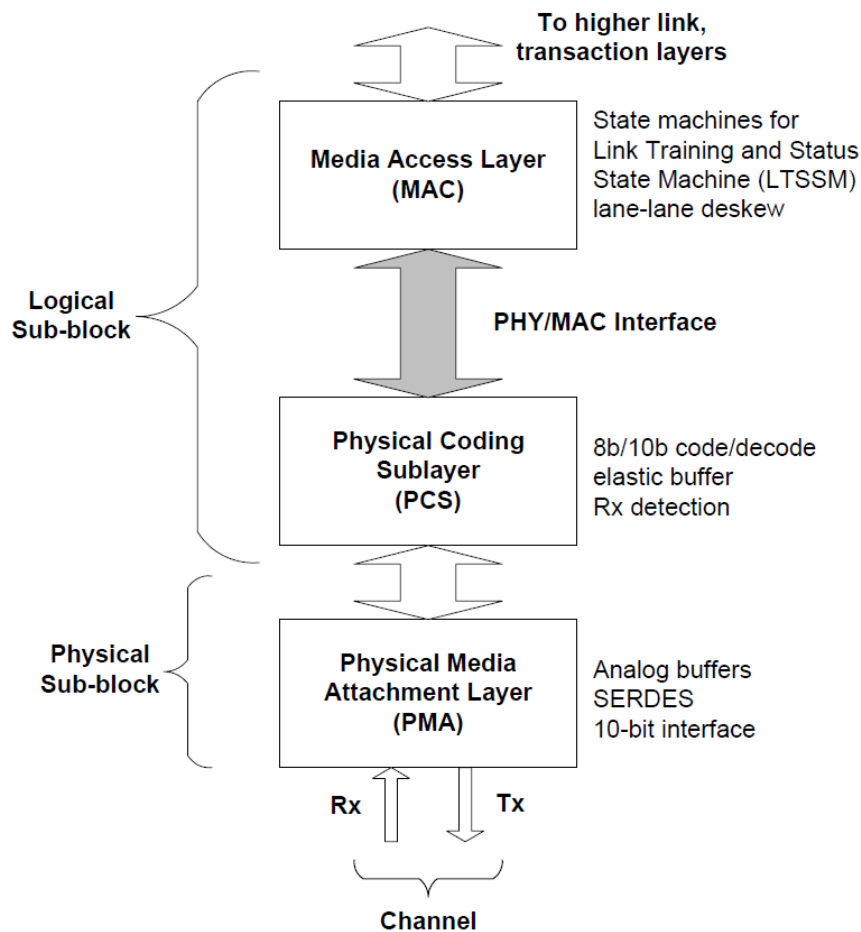


Abb. 3.10: Position des PIPE im *Physical Layer* nach [Int07]

Zusätzlich zur parallelen Schnittstelle definiert die Spezifikation die Funktionalität und das Verhalten des PHY, wobei sie jedoch keine zwingenden Vorgaben für den internen Aufbau des PCS oder des *Physical Media Attachment Layers* (PMA) macht. Die Bezeichnung PHY beschreibt in diesem Zusammenhang die Kombination aus PCS und PMA. Generell gilt jedoch, dass die PCI Express Spezifikation immer Vorrang gegenüber der PIPE Spezifikation hat, sollten sich diese in irgend einer Form widersprechen. Die Intention für die Entwicklung der Spezifikation war und ist es eine gemeinsame Entwicklungsbasis für die Hersteller von ASICs und die Entwickler von PCI Express Cores bereit zu stellen. Durch die Vorgabe der Schnittstellen und der grundlegenden Funktionen, erreicht das PIPE eine Trennung zwischen der rein digitalen Soft-Core Entwicklung und der Entwicklung von Makrozellen oder PHY-Chips auf Basis von *Mixed-Signal* Simulationen. Dieser Ansatz führt letztendlich zu geringeren Entwicklungszeiten, da beide Komponenten unabhängig voneinander

und ohne Risiken durch inkompatible Systeme entwickelt werden können. Als Informationsgrundlage für diese Arbeit dient die Version 2.0 der PIPE Spezifikation [Int07].

3.3.1 Interface

Die vom PIPE definierten Schnittstellen zwischen MAC und PHY, lassen sich in das Taktsignal (PCLK), die Kontrollsignale (*Command*), die Statussignale (*Status*) und die entsprechenden Datenports für den Rx- und Tx-Datenpfad aufteilen. Die Spezifikation führt zusätzlich die analogen Schnittstellen für die Rx- und Tx-Leitung, sowie den notwendigen Referenztakt (REFCLK) für die interne PLL auf. Die REFCLK wird hierbei vereinfacht als CLK bezeichnet und dient zur Generierung der internen *Bit Rate Clock* für die PCIe Kommunikation (Abb. 3.11).

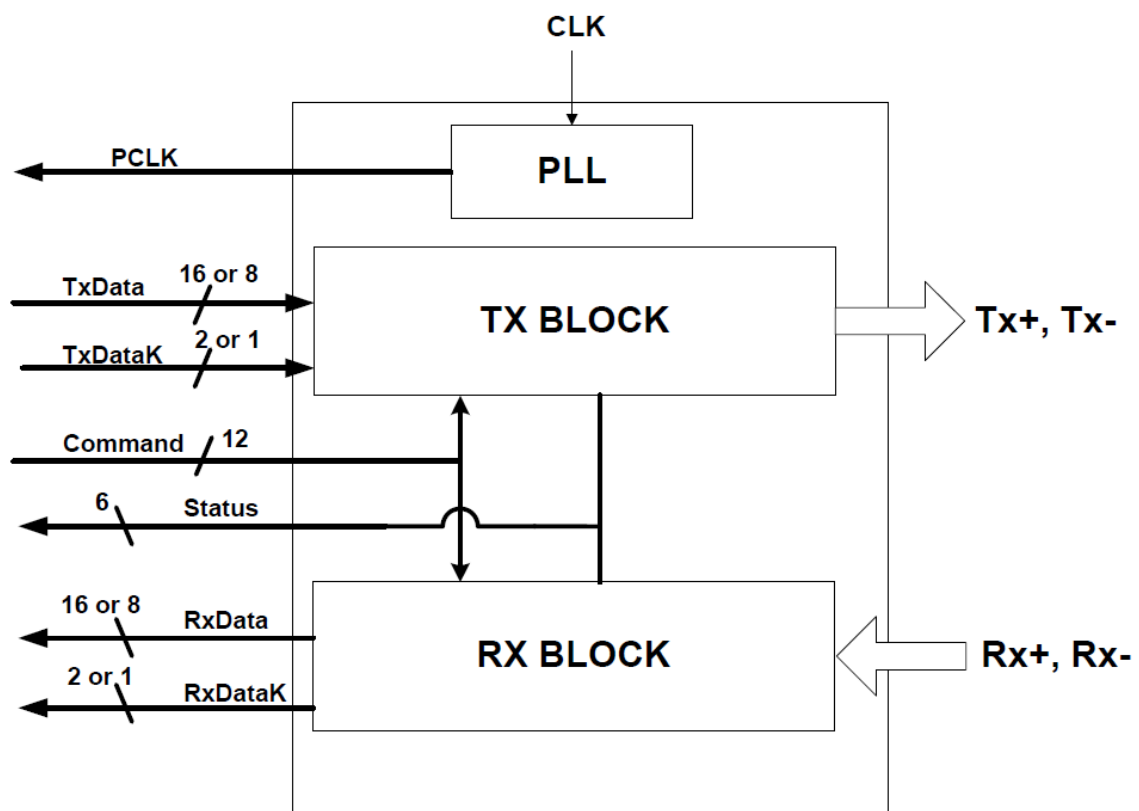


Abb. 3.11: PIPE Interface als Block Diagramm nach [Int07]

Zusätzlich stellt die PLL die *Parallel Interface Clock* (PCLK) bereit, welche dazu genutzt wird, den Datentransfer des parallelen Interfaces zwischen MAC und PHY zu synchronisieren. Als Referenzpunkt dienen die steigenden Taktflanken der PCLK.

Die PCLK besitzt abhängig von der gewählten Übertragungsrate (2,5 Gbit/s oder 5 Gbit/s) und der gewählten Bitbreite des Datenpfads (8-/16-Bit) eine Frequenz von 125MHz, 250MHz oder 500MHz. Bei einer Übertragungsrate von 2,5 Gbit/s ergeben sich für die beiden Bitbreiten folgende Frequenzen für die PCLK, wobei die Übertragung von einem Datenbyte als 10 Bit erfolgt.

$$\frac{2,5 \text{ Gbit/s}}{10 \text{ Bit}} = 250 \text{ MHz} \quad (3.1)$$

$$\frac{2,5 \text{ Gbit/s}}{20 \text{ Bit}} = 125 \text{ MHz} \quad (3.2)$$

In den nachfolgenden Tabellen sind die wichtigsten Schnittstellen des Interfaces aufgeführt. Da in der späteren Entwicklung aufgrund der konstanten Übertragungsrate von 2,5 Gbit/s keine Implementierung der Kontrollsignale *Rate*, *TxDemph*, *TxMargin* und *TxSwing* vorgesehen ist, werden diese nicht mit aufgeführt. Eine Beschreibung dieser Kontrollsignale findet sich in der Spezifikation [Int07, S.13ff].

Name	Typ	Active	Beschreibung
TxData[16:0] TxData[7:0]	Input	N/A	Schnittstelle für die Sendedaten der PCI Express Kommunikation. Kann wahlweise als 16- oder 8-Bit definiert werden. 8-Bit repräsentieren jeweils ein Symbol für die Übertragung ([15:8], [7:0]). Bei 16-Bit beginnt die Übertragung mit dem <i>Lowest Byte</i> ([7:0]).
TxDataK[1:0] TxDataK	Input	N/A	Schnittstelle für die Definition der übertragenen Symbole. Definiert ob es sich um ein Daten- oder Kontrollsymbol handelt. Bei 16-Bit Datenbreite kennzeichnet das LSB [0] das <i>Lowest Byte</i> [7:0]. Eine 1 definiert das Symbol als Kontrollsymbol und eine 0 als Datensymbol. Wird für den 8b/10b Decoder benötigt.

Tabelle 3.1: PIPE Schnittstellen für die Sendedaten

Name	Typ	Active	Beschreibung
RxData[16:0] RxData[7:0]	Output	N/A	Schnittstelle für die Empfangsdaten der PCI Express Kommunikation. Kann wahlweise als 16- oder 8-Bit definiert werden. 8-Bit repräsentieren jeweils ein Symbol der empfangenen Daten ([15:8], [7:0]). Bei 16-Bit stellt das <i>Lowest Byte</i> ([7:0]) das erste empfangene Symbol dar.
RxDataK[1:0] RxDataK	Output	N/A	Schnittstelle für die Definition der empfangenen Symbole. Definiert ob es sich um einen Daten- oder Kontrollsymbol handelt. Bei 16-Bit Datenbreite kennzeichnet das LSB [0] das <i>Lowest Byte</i> [7:0]. Eine 1 definiert das Symbol als Kontrollsymbol und eine 0 als Datensymbol.

Tabelle 3.2: PIPE Schnittstellen für die Empfangsdaten

Name	Typ	Active	Beschreibung
Reset#	Input	Low	Asynchrones Signal, das einen Reset für den PHY auslöst.
TxDetectRx/ Loopback	Input	High	Führt je nach <i>Power State</i> dazu, dass der PHY eine <i>Receiver Detection</i> durchführt (P1) oder in den <i>Loopback</i> Modus wechselt (P0). Für eine Erläuterung der <i>Power States</i> siehe Kap. 3.3.3.
TxElecIdle	Input	High	Führt in allen <i>Power States</i> dazu, dass der Transmitter des PHY in <i>Electrical Idle</i> übergeht bzw. verbleibt. Für die <i>Power States</i> P0s und P1 muss dieses Signal immer gesetzt sein. In den <i>Power States</i> P0 und P2, führt dessen Inaktivität zu einer normalen Übertragung der Sendedaten (P0) oder zu einer <i>Beacon</i> Übertragung (P2). Für eine Erläuterung der <i>Power States</i> siehe Kap. 3.3.3.
TxCompliance	Input	High	Setzt die Disparität (<i>Disparity</i>) der Sendedaten auf einen negativen Wert. Wird genutzt, um das <i>Compliance Pattern</i> zu erzeugen.
RxPolarity	Input	High	Wenn gesetzt führt der Receiver des PHY eine Invertierung der Polarität für die Empfangsdaten durch.
PowerDown[1:0]	Input	N/A	Wird genutzt, um den PHY in einem entsprechenden <i>Power State</i> zu überführen. Die 2-Bit breite Codierung gibt den gewünschten <i>Power State</i> an (siehe Tab. 3.6). Beim Übergang aus dem <i>Power State</i> P2 nach P1 ist das Signal als asynchron zu betrachten, da die PCLK deaktiviert ist. Für eine Erläuterung der <i>Power States</i> siehe Kap. 3.3.3.

Tabelle 3.3: PIPE Schnittstellen für die Kontrollsignale

Name	Typ	Active	Beschreibung																		
RxValid	Output	High	Das Signal weist auf einen vorhandenen <i>Symbol Lock</i> und valide Empfangsdaten an den entsprechenden Datenports (RxData, RxDataK) hin.																		
PhyStatus	Output	High	Das Signal PhyStatus wird dazu genutzt den erfolgreichen Übergang in einen <i>Power State</i> oder eine abgeschlossene <i>Receiver Detection</i> anzuzeigen. Im Fall von Übergängen in und aus <i>Power State</i> P2 ist das Signal auf Grund der deaktivierten PCLK als asynchron zu betrachten. Bleibt die Signalisierung des PHY über <i>PhyStatus</i> aus, deutet dies auf einen internen Fehlerzustand hin und die MAC kann entsprechende Maßnahmen einleiten.																		
RxElecIdle	Output	High	Das Signal ist als asynchron zu betrachten. Es zeigt an ob der Receiver auf der physikalischen Empfangsleitung ein <i>Electrical Idle</i> detektiert hat. Im Fall des <i>Power States</i> P2 weist die Unterbrechung des Signals auf die Detektierung eines <i>Beacon</i> hin.																		
RxStatus[2:0]	Output	N/A	Die 3-Bit breite Codierung des Signals gibt je nach <i>Power State</i> Aufschluss über verschiedene Fehlerzustände der Empfangsdaten (P0) oder zeigt das Ergebnis einer <i>Receiver Detection</i> an (P1). <table border="1" data-bbox="762 1323 1396 2000"> <thead> <tr> <th>RxStatus</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>000</td> <td>Received Data OK (P0), Receiver not detected (P1)</td> </tr> <tr> <td>001</td> <td>SKP added</td> </tr> <tr> <td>010</td> <td>SKP removed</td> </tr> <tr> <td>011</td> <td>Receiver detected (P1)</td> </tr> <tr> <td>100</td> <td>8b/10b Error and optionally Disparity Error</td> </tr> <tr> <td>101</td> <td>Elastic Buffer Overflow</td> </tr> <tr> <td>110</td> <td>Elastic Buffer Underflow</td> </tr> <tr> <td>111</td> <td>Disparity Error. Unused if reported together with 8b/10b Error (100).</td> </tr> </tbody> </table>	RxStatus	Description	000	Received Data OK (P0), Receiver not detected (P1)	001	SKP added	010	SKP removed	011	Receiver detected (P1)	100	8b/10b Error and optionally Disparity Error	101	Elastic Buffer Overflow	110	Elastic Buffer Underflow	111	Disparity Error. Unused if reported together with 8b/10b Error (100).
RxStatus	Description																				
000	Received Data OK (P0), Receiver not detected (P1)																				
001	SKP added																				
010	SKP removed																				
011	Receiver detected (P1)																				
100	8b/10b Error and optionally Disparity Error																				
101	Elastic Buffer Overflow																				
110	Elastic Buffer Underflow																				
111	Disparity Error. Unused if reported together with 8b/10b Error (100).																				

Tabelle 3.4: PIPE Schnittstellen für die Statussignale

3.3.2 Reset

Im Falle eines Reset muss die MAC den PHY über das Reset Signal (*Reset#*) so lange im Reset-Zustand halten bis die Spannungsversorgung und der Referenztakt (REFCLK) für diesen stabil sind. Ab diesem Zeitpunkt sollte auch die von der PLL generierte PCLK mit annähernder Zielfrequenz vorhanden sein. Hat der PHY seinen angestrebten *Power State* erreicht (in diesem Fall P1) und somit den Reset-Zustand verlassen, teilt er dies der MAC über den Flankenwechsel am Ausgangsport *PhyStatus* mit (Abb. 3.12).

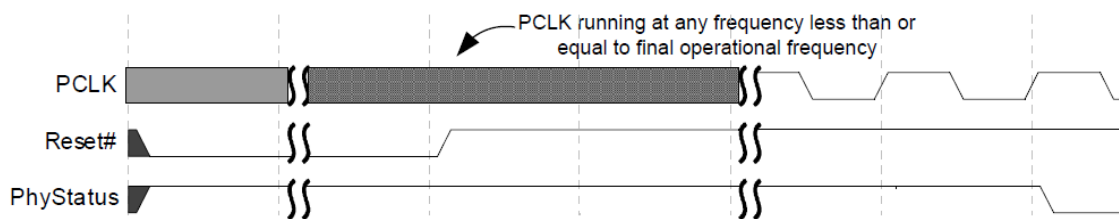


Abb. 3.12: Resetverhalten des PIPE nach [Int07]

Die Spezifikation legt für die Kontrollsignale während des gesamten Reset-Vorgangs folgende Datenwerte fest (Tab. 3.5).

Command Signal	Data Value
TxDetectRx/Loopback	0
TxElecIdle	1
TxCompliance	0
RxPolarity	0
PowerDown	10 (P1)

Tabelle 3.5: Reset-Werte der Kontrollsignale

3.3.3 Power Management

Das *Power Management* des PHY erfolgt über das Kontrollsignal *PowerDown* und ist eng mit der LTSSM in der MAC und dem *Power Management* der PCI Express Basisspezifikation verknüpft [PCI09, S.313ff]. Die PIPE Spezifikation definiert für

den PHY die vier *Power States* P0, P0s, P1 und P2. Der Übergang in einen anderen *Power State* wird über den entsprechenden Datenwert des Kontrollsignals eingeleitet (Tab. 3.6).

Power State	PowerDown
P0	00
P0s	01
P1	10
P2	11

Tabelle 3.6: Definition des Kontrollsignals *PowerDown*

Der *Power State* P0 entspricht dem normalen Betriebszustand des PHY, in dem die Datenkommunikation stattfindet und wird für die meisten Zustände der LTSSM verwendet. Innerhalb der *Low Power States* (P0s, P1 und P2) kann der PHY angemessene Maßnahmen ergreifen, um den Energieverbrauch im PHY zu verringern. Die Maßnahmen sind jeweils von der internen Struktur des PHY abhängig und werden individuell vom Hersteller definiert. Der PHY darf in diesem Zusammenhang jedoch nicht die Zeitvorgaben der PCIe Basisspezifikation für das *Link Training* oder die *Clock Recovery* (*Bit Lock* und *Symbol Lock*) verletzen. Die Spezifikation führt für die *Power States* folgende Definitionen an.

▪ **Power State P0:**

- Normaler Betriebszustand, es findet eine PCIe Kommunikation statt.
- Alle internen Taktgeber des PHY (*Clocks*) sind aktiv.
- Receiver kann selbstständig Energiesparmaßnahmen durchführen, falls *Electrical Idle* anliegt.

▪ **Power State P0s:**

- PCLK muss weiterhin aktiv bleiben.
- Transmitter befindet sich im *Electrical Idle* Zustand.

- Receiver kann selbstständig Energiesparmaßnahmen durchführen, falls *Electrical Idle* anliegt.
- Verwendung in LTSSM Zuständen:
 - Tx_L0s.Idle
- **Power State P1:**
 - PCLK muss weiterhin aktiv bleiben.
 - Ausgewählte interne Taktgeber (*Clocks*) können deaktiviert werden.
 - Initialisierungszustand im Reset-Fall.
 - Verwendung in LTSSM Zuständen:
 - Disable
 - Detect
 - L1.Idle
- **Power State P2:**
 - Ausgewählte interne Taktgeber (*Clocks*) können deaktiviert werden.
 - PCLK ist deaktiviert, das Interface befindet sich im asynchronen Modus.
 - PHY arbeitet innerhalb der Grenzen der Auxiliary Power (V_{aux}).
 - Verwendung in LTSSM Zuständen:
 - L2.Idle
 - L2.TransmitWake

Mit Ausnahme des *Power State P2* wird der erfolgreiche Übergang in einen anderen *Power State* der MAC über den *PhyStatus* Port mitgeteilt, indem dessen Signal für genau eine Taktflanke von PCLK aktiv ist. Die Generierung des entsprechenden *PhyStatus* Signals für den *Power State P1* ist zudem vom Resetvorgang abhängig,

da dieser *Power State* auch als Initialisierungszustand genutzt wird. Daher muss der PHY sicherstellen, dass die PCLK und die *Common Mode Voltage* stabil sind, bevor er den erfolgreichen Übergang in den *Power State* P1 signalisiert. Im Fall des *Power State* P2 wird beim Eintritt in den *Power State* der *PhyStatus* auf 1 gezogen, bevor die PCLK deaktiviert ist und wieder auf 0 gezogen wenn die PCLK ausgeschaltet und der PHY endgültig in den *Power State* eingetreten ist. Beim Verlassen von P2 wird *PhyStatus* auf 1 gezogen und zu einer steigenden Flanke von PCLK wieder auf 0 gezogen, sobald diese stabil anliegt. Die nachfolgende Abbildung, verdeutlicht den Zusammenhang zwischen den definierten *Power States* der PIPE Spezifikation und den Zuständen der LTSSM aus der PCIe Basisspezifikation (Abb. 3.13).

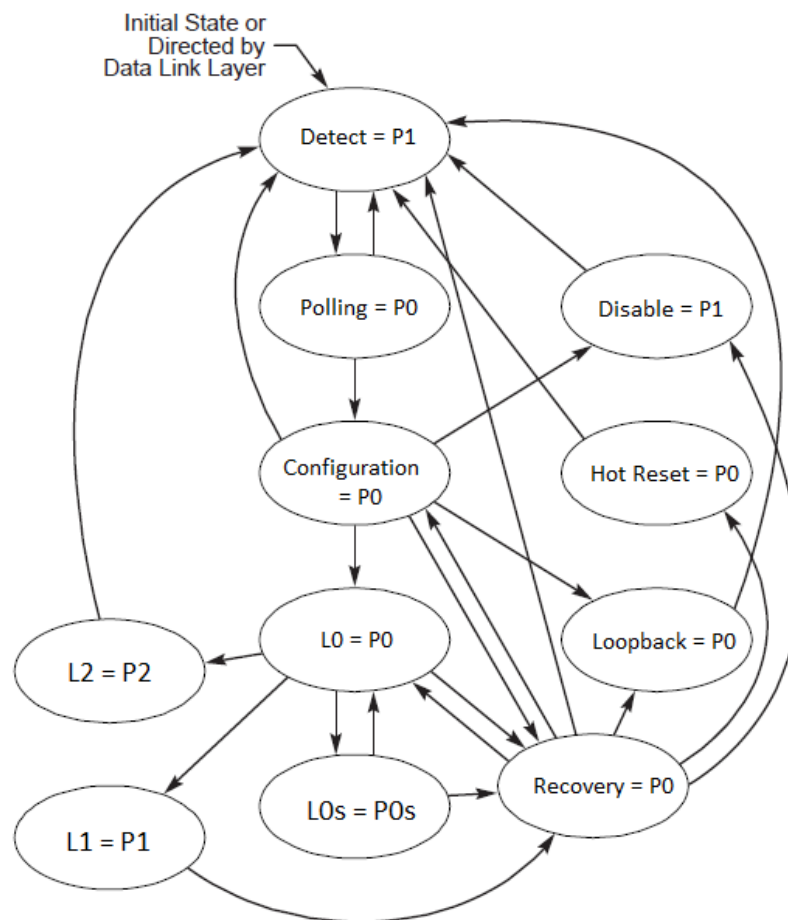


Abb. 3.13: Einsatz der *Power States* in der LTSSM nach [Syn21]

In der LTSSM werden die *Low Power States* P0s und P2 direkt eingesetzt, um die korrespondierenden *Low Power States* L0s und L2 der PCIe Basisspezifikation umzusetzen. Lediglich der *Low Power State* P1 wird nicht nur für L1 eingesetzt, sondern kommt auch in den Zuständen *Detect* und *Disable* zum tragen. Ausgehend von der

Verwendung der *Power States* in der LTSSM ergibt sich für den PHY eine begrenzte Anzahl an zulässigen Übergängen zwischen den einzelnen *Power States*. Die notwendigen Bedingungen für die Übergänge werden durch die LTSSM bestimmt. Für den PHY ergeben sich die folgenden zulässigen Übergänge:

- P0 ↔ P0s
- P0 ↔ P1
- P0 → P2
- P2 → P1

Die *Power States* des PHY haben ebenfalls Einfluss auf die anliegenden Kontrollsignale des Interface. Hierbei variieren die zulässigen Datenwerte oder führen je nach *Power State* und Kontrollsignal zu unterschiedlichen Aktionen des PHY. In der nachfolgenden Tabelle sind die zulässigen Datenwerte und Funktionen der Kontrollsignale *TxDetectRx/Loopback* und *TxElecIdle* in Abhängigkeit zu den *Power States* dargestellt (Tab. 3.7).

PowerDown	TxDetectRx/ Loopback	TxElecIdle	Beschreibung
P0	0	0	PHY ist im normalen Betriebszustand. MAC sendet und empfängt Daten.
	0	1	PHY befindet sich in Electrical Idle. Es findet keine Übertragung statt.
	1	0	PHY befindet sich im Loopback Modus.
	1	1	Ungültig, nicht definiert.
P0s	-	0	Ungültig, nicht definiert.
		1	PHY befindet sich in Electrical Idle.
P1	-	0	Ungültig, nicht definiert.
	0	1	PHY befindet sich in Electrical Idle.
	1	1	PHY führt eine Receiver Detection durch.
P2	-	0	PHY führt eine Beacon Übertragung durch.
		1	PHY befindet sich in Electrical Idle.

Tabelle 3.7: Funktion von TxDetectRx/Loopback und TxElecIdle

Die Funktion der Kontrollsignale *TxCmpliance* und *RxPolarity* bleiben in ihrer Funktion unverändert, sind jedoch nur während P0 gültig, wenn der PHY entsprechende Übertragungen durchführt. Das Signal *Reset#* wird erwartungsgemäß priorisiert und löst unabhängig vom derzeitigen *Power State* oder den anliegenden Kontrollsignalen den Reset des PHY aus.

Beispieloperation: L0 und L1

Die Spezifikation führt für die Übergänge des PHY zwischen den beiden *Power States* L0 und L1 bzw. P0 und P1 entsprechende Beispiele für das Zeitverhalten an. Bei einem Übergang aus dem *Power State* P0 nach P1 wird die entsprechende Codierung an *PowerDown* angelegt und zeitgleich das Signal für *TxElecIdle* auf 1 gezogen (Abb. 3.14).

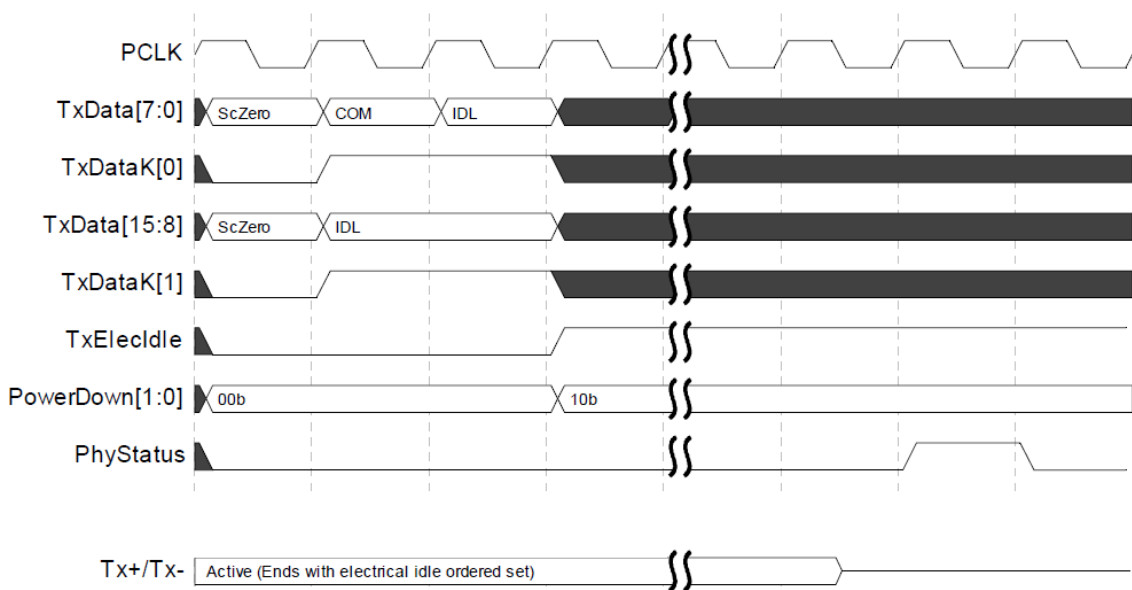


Abb. 3.14: Übergang des PHY in *Power State* P0 nach [Int07]

Ebenfalls erfolgt die notwendige Versendung des *Electrical Idle Ordered Set* für den Eintritt in *Electrical Idle* (siehe Kap. 3.3.10). Zu beachten ist, dass der erfolgreiche Übergang in den *Power State* P1 erst nach Erreichen des *Electrical Idle* Zustands auf der physikalischen Leitung durch *PhyStatus* signalisiert wird. Der *Electrical Idle* Zustand sollte daher als Teil des *Power States* P1 betrachtet werden und beeinflusst die Signalisierung durch *PhyStatus*. Im Gegenzug wird der *Electrical Idle* Zustand beim Übergang von P1 nach P0 zunächst aktiv gehalten und der PHY signalisiert

über *PhyStatus* seine Bereitschaft Übertragungen durchzuführen (Abb. 3.15). Die Signalisierung scheint sich hierbei auf die internen Maßnahmen für die Reduzierung des Energieverbrauchs zu beziehen und nicht von *Electrical Idle* abhängig zu sein.

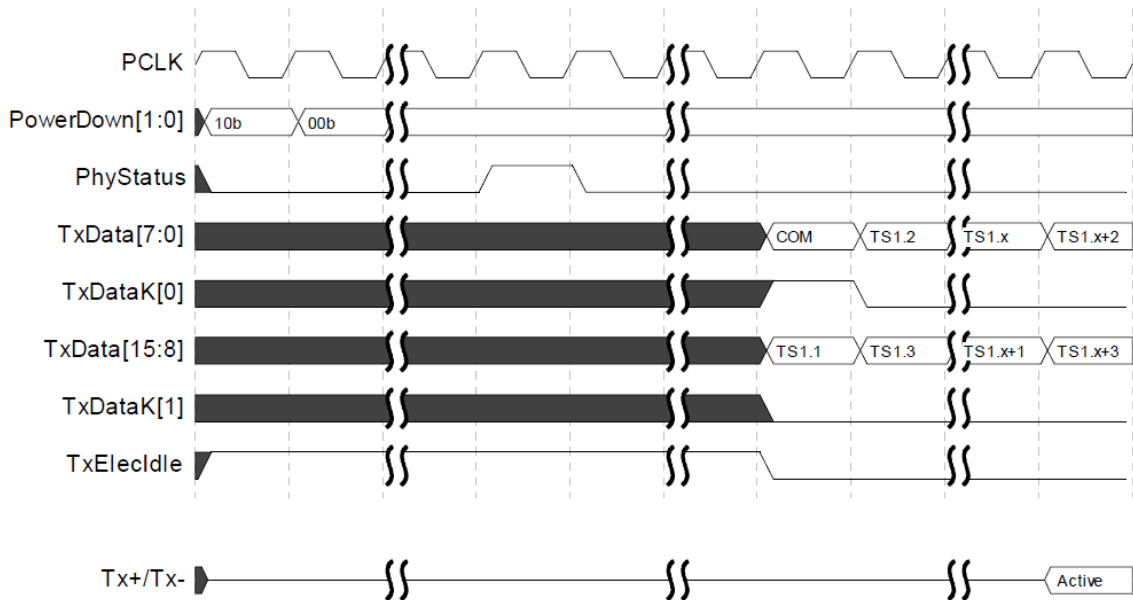


Abb. 3.15: Übergang des PHY in *Power State P1* nach [Int07]

3.3.4 Receiver Detection

Befindet sich der PHY im *Power State P1* kann die MAC diesen über das Kontrollsignal *TxDetectRx/Loopback* dazu veranlassen, eine *Receiver Detection* durchzuführen. Der PHY startet daraufhin die notwendige Sequenz und teilt der MAC das Ergebnis über das Statussignal *RxStatus* mit. Dazu zieht der PHY das Statussignal *PhyStatus* für genau einen Takt der PCLK auf 1 und weist *RxStatus* zeitgleich die entsprechende Codierung zu (Abb. 3.16).

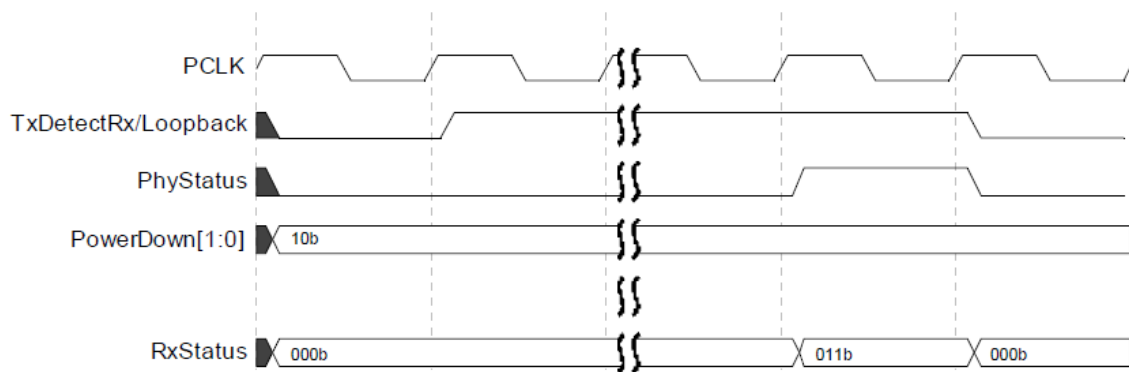


Abb. 3.16: Durchführung einer *Receiver Detection* nach [Int07]

Im gezeigten Beispiel wird die MAC durch die Codierung (011) über die erfolgreiche Detektierung eines *Receivers* auf der Gegenseite informiert. Während der Durchführung der *Receiver Detection* muss das Kontrollsignal *TxDetectRx/Loopback* dauerhaft aktiv bleiben. Danach muss das Kontrollsignal zwischenzeitlich wieder auf 0 gezogen werden, um eine erneute *Receiver Detection* einleiten zu können.

3.3.5 Clock Tolerance Compensation

Der *Elastic Buffer* des PHY führt wie von der Basisspezifikation gefordert die nötige *Clock Tolerance Compensation* aus, um einen *Overflow* oder *Underflow* des *Buffers* zu verhindern. Dazu fügt er entweder den SKP *Ordered Sets* im *Buffer* zusätzliche SKP Symbole hinzu oder entfernt einige dieser Symbole. Erreicht ein SKP *Ordered Set* die Ausgangsports für die Empfangsdaten (*RxData*), teilt der PHY während der Übertragung des COM Symbols über *RxStatus* mit, ob SKP Symbole hinzugefügt (001) oder entfernt (010) worden sind (Abb. 3.17 und 3.18).

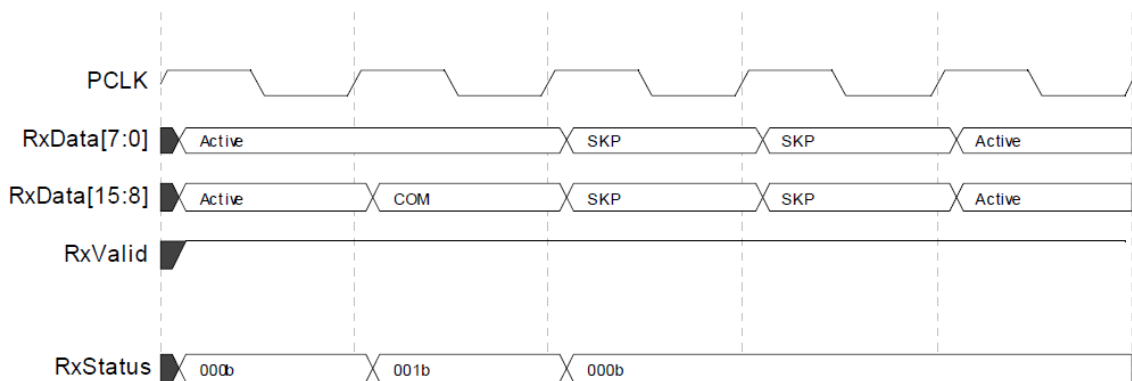


Abb. 3.17: SKP *Ordered Set* mit hinzugefügten SKP Symbol nach [Int07]

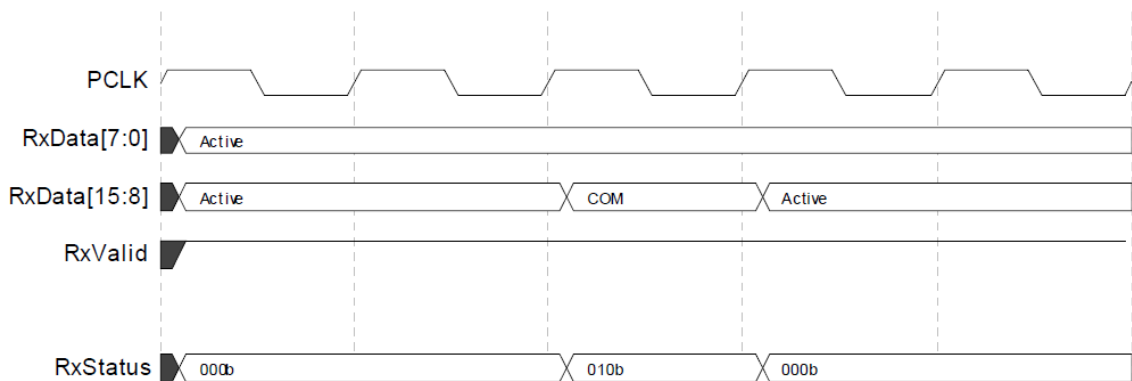


Abb. 3.18: SKP *Ordered Set* mit vollständig entfernten SKP Symbolen nach [Int07]

3.3.6 Error Detection

Der PHY ist über das Statussignal *RxStatus* in der Lage die MAC über Fehler zu informieren, die mit Bezug auf die Empfangsdaten (*RxData*) bzw. den *Elastic Buffer* auftreten. Da während des Betriebs verschiedene Fehlerfälle gleichzeitig auftreten können, sind diese für die Zuweisung von *RxStatus* wie folgt priorisiert zu betrachten und überwiegen auch die möglichen Ausgaben für die *Clock Tolerance Compensation* (SKP).

1. 8b/10b Error
2. Elastic Buffer Overflow

3. Elastic Buffer Underflow
4. Disparity Error
5. Clock Tolerance Compensation (SKP)

8b/10b Error

Liegt bei der 8b/10b Dekodierung ein Fehler vor, konnte einer der empfangenen 10-Bit Datenwerte keinem zulässigen Byte Symbol zugeordnet werden. In diesem Fall fügt der PHY an dessen Stelle ein EDB Symbol (K30.7) ein und informiert die MAC während der Übertragung durch die Codierung von *RxStatus* (111) über den vorliegenden Fehler (Abb. 3.19). Im gezeigten Signalverlauf ist das Symbol Rx-f ersetzt worden.

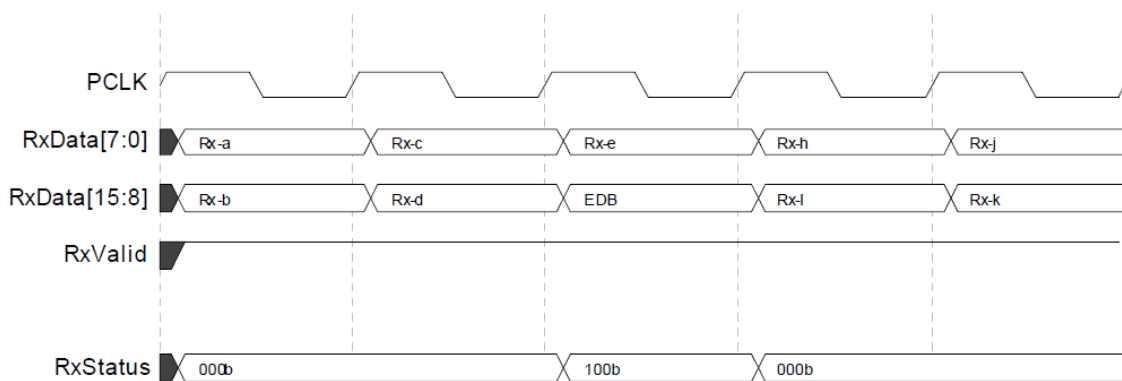


Abb. 3.19: Ausgabe eines 8b/10b Fehlers für Symbol Rx-f nach [Int07]

In diesem Zusammenhang ist zu beachten, dass das fehlerhafte Symbol zusätzlich über eine fehlerhafte Disparität (*Disparity*) verfügen kann. Dies gilt ebenfalls für das zweite Symbol im Fall einer 16-Bit breiten Schnittstelle (RxData). Daher kann ein Fehler in der *Disparity*, aufgrund der Priorisierung des 8b/10b Fehlers, nicht zeitgleich mitgeteilt werden.

Elastic Buffer Error

Mit Bezug auf den *Elastic Buffer* können Fehler in Form eines *Underflows* oder eines *Overflows* auftreten. Im Fall eines *Underflows* fügt der PHY für das fehlende Symbol ein EDB Symbol (K30.7) an der entsprechenden Stelle der Empfangsdaten ein.

Während der Übertragung des Symbols über die Schnittstelle für die Empfangsdaten (RxData), signalisiert der PHY über die entsprechende Codierung von *RxStatus* (100) den *Underflow* (Abb. 3.20).

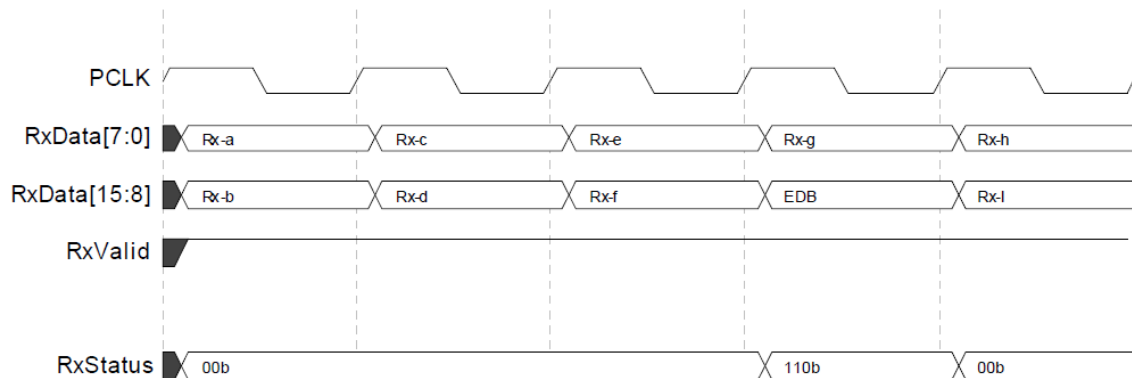


Abb. 3.20: Signalisierung des *Underflows* und Ausgabe EDB Symbol nach [Int07]

Tritt hingegen ein *Overflow* auf, konnten nicht alle Empfangsdaten in den *Elastic Buffer* übernommen werden, wodurch Symbole weggefallen sind. Der PHY kennzeichnet dies indem er zum Zeitpunkt, bei dem die Übertragung dieser Symbole erfolgt wäre, die MAC über die entsprechende Codierung von *RxStatus* (101) informiert (Abb. 3.21).

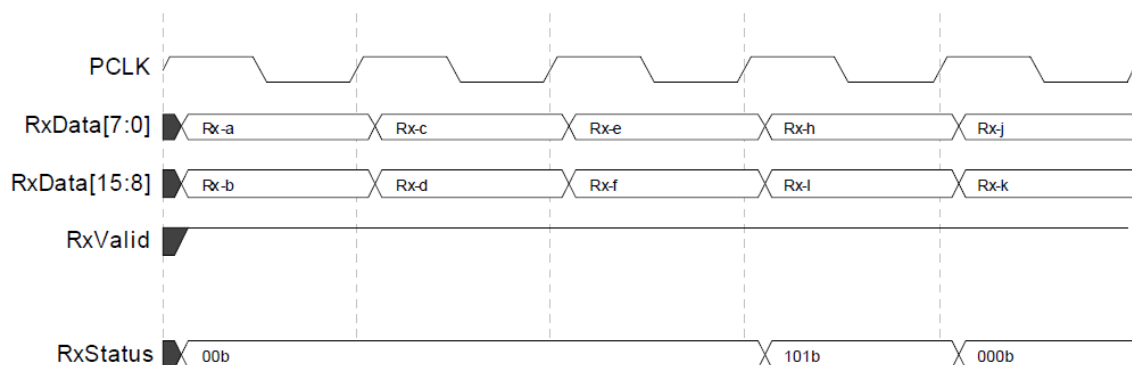


Abb. 3.21: Signalisierung eines *Overflows* für Symbol Rx-g nach [Int07]

Disparity

Ein Fehler in der Disparität (*Disparity*) führt im Gegensatz zu den anderen Fehlerfällen nicht zu einer Manipulation der Empfangsdaten. Liegt für einen empfangenen

10-Bit Datenwert eine fehlerhafte Disparität vor, kann dieser trotzdem in ein gültiges Symbol überführt werden. Während der Übertragung des entsprechenden Symbols signalisiert der PHY über die Codierung von *RxStatus* (111), dass ein Fehler der Disparität vorliegt (Abb 3.22).

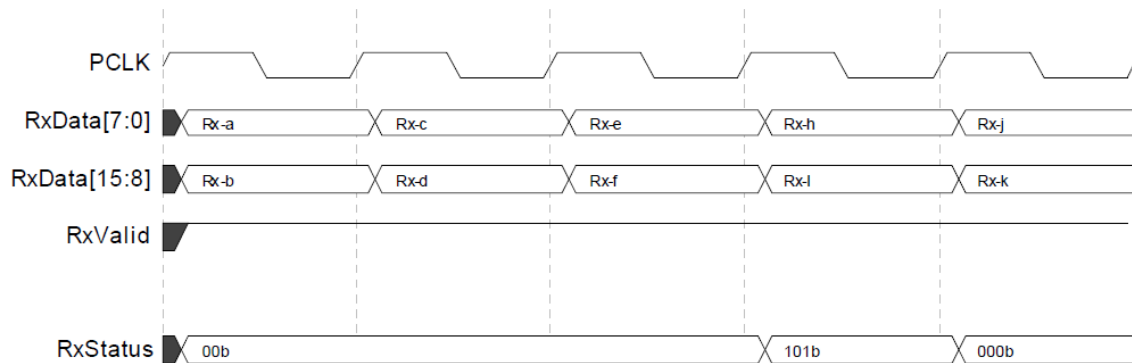


Abb. 3.22: Signalisierung eines *Overflows* für Symbol Rx-g nach [Int07]

Es ist zu beachten, dass bei einem 16-Bit breiten Datenpfad keine eindeutige Aussage darüber getroffen werden kann, welches der beiden anliegenden Bytes an *RxData* ([15:8], [7:0]) den Fehler aufweist, oder ob beide Bytes diesen Fehler aufweisen.

3.3.7 Loopback Mode

Befindet sich der PHY im *Power State* P0 und damit im normalen Betriebszustand in dem er PCIe Daten sendet und empfängt, kann die MAC den PHY über das Signal *TxElecIdle/Loopback* in den Loopback Modus versetzen. In diesem Modus beginnt der PHY damit, die anliegenden Sendedaten an *TxDData* zu ignorieren und die Empfangsdaten zurück zu senden. Dabei gibt der PHY die Empfangsdaten weiterhin über die Schnittstelle *RxDData* aus (Abb. 3.23).

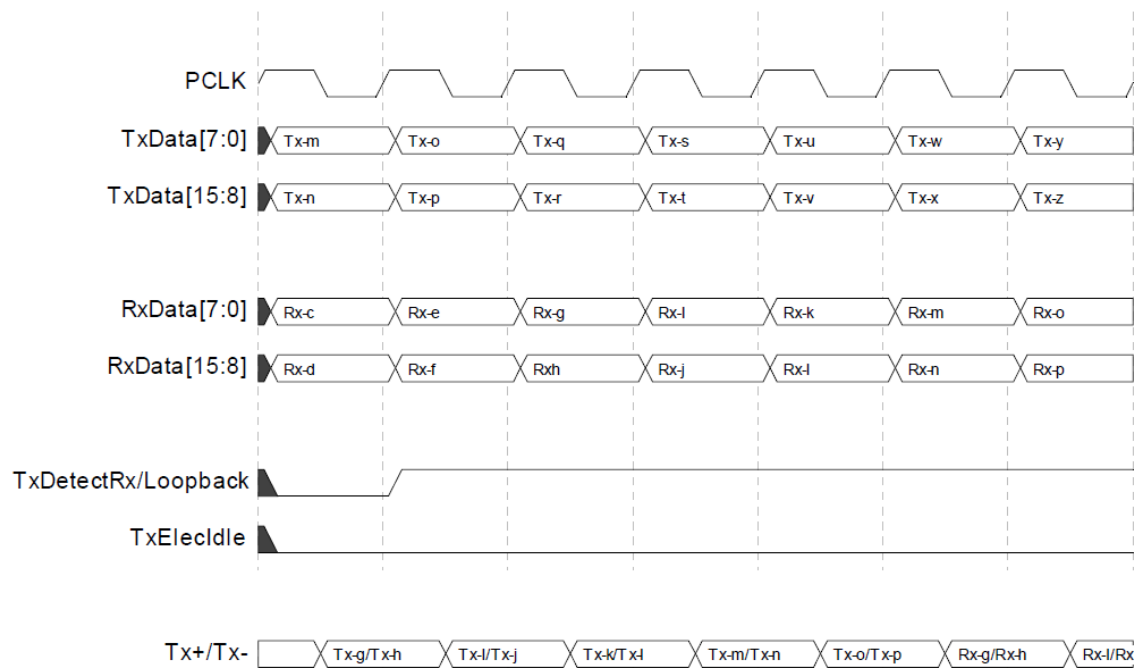


Abb. 3.23: Initialisierung des Loopback Modus nach [Int07]

In oben dargestellten Signalverlauf initialisiert die MAC den *Loopback* Modus über *TxElecIdle/Loopback*, wodurch die anliegenden Sendedaten auf der physikalischen Sendeleitung (Tx+/Tx-) bei Tx-o/Tx-p unterbrochen und die Übertragung der Empfangsdaten mit Rx-g/Rx-h beginnt. Die Verzögerungszeit zwischen dem Kontrollsignal *TxDetectRx/Loopback* bis zum Zeitpunkt an dem die Empfangsdaten auf der Sendeleitung erscheinen, ist vom jeweiligen PHY abhängig.

Der *Loopback* Modus wird nach der PCI Express Basisspezifikation beendet, sobald der *Loopback Slave*, in diesem Fall der PHY, vom *Loopback Master* ein *Electrical Idle Ordered Set* (EIOS) empfängt [PCI09, S.255 Z.1ff]. Die MAC kann den *Loopback* Modus des PHY danach wieder über das Signal *TxDetectRx/Loopback* deaktivieren und diesen zeitgleich durch das Signal *TxElecIdle* in den *Electrical Idle* Zustand überführen. Voraussetzung ist, dass das vom Master empfangene EIOS noch übertragen wird. Ebenfalls darf der PHY die Daten nach Erhalt des EIOS (*Junk*) noch ungestraft weiterleiten (Abb. 3.24).

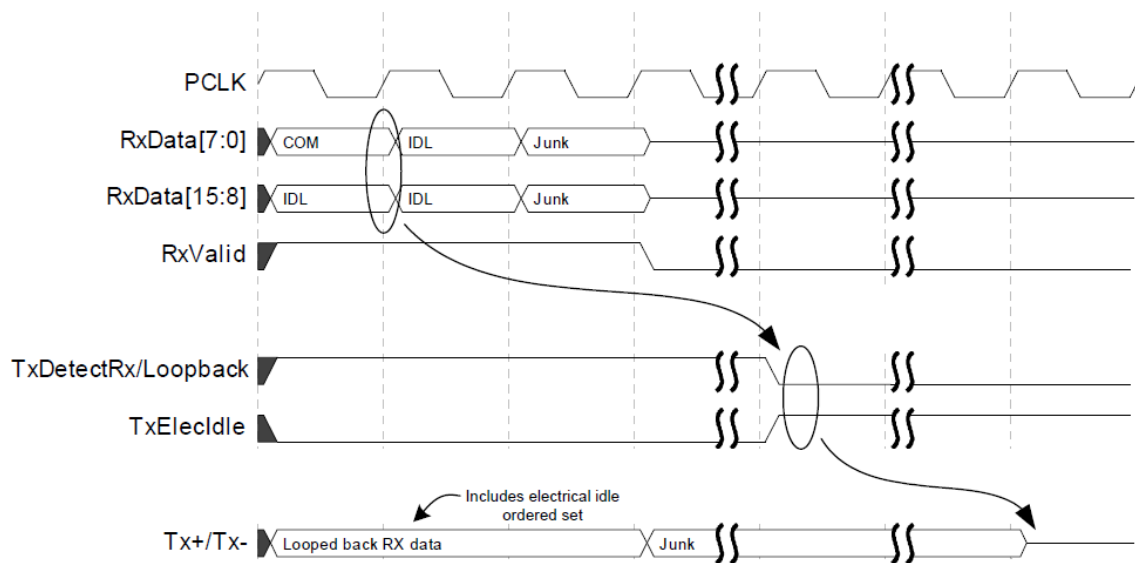


Abb. 3.24: Austritt aus dem Loopback Modus nach [Int07]

Die PIPE Spezifikation [Int07, S.27] gibt für die Erkennung des EIOS bis zur Durchführung der entsprechenden Maßnahmen (*Electrical Idle*) eine Reaktionszeit von 1ms vor und beruft sich dabei auf die PCIe Basisspezifikation [PCI09, S.256]. Die mögliche Reaktionszeit des PHY ergibt sich dabei Sinngemäß aus der Verzögerungszeit zwischen physikalischer Empfangsleitung und der parallelen Empfangsschnittstelle (*Receive Latency*) sowie der benötigten Sendezeit des *Loopbacks*.

3.3.8 Rx Polarity

Der PHY ist in der Lage die Polarität seiner Empfangsdaten zu invertieren. Wird während des Training Prozesses festgestellt, dass eine entsprechende *Polarity Inversion* durchgeführt werden muss, kann die MAC den PHY mithilfe des Kontrollsignals *RxPolarity* dazu veranlassen (Abb. 3.25). Die interne technische Umsetzung der Invertierung ist dabei vom individuellen Design des PHY abhängig und nicht vorgegeben. Die PIPE Spezifikation legt jedoch fest, dass die invertierten Daten nach Aktivierung des Kontrollsignals innerhalb von 20 Takten der PCLK an der Schnittstelle für die Empfangsdaten (*RxData*) anliegen müssen.

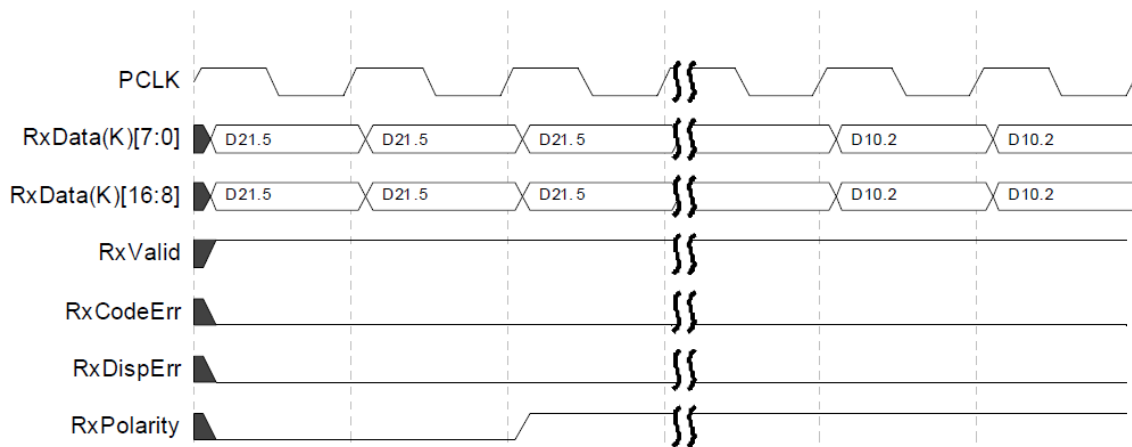


Abb. 3.25: Invertierung der Polarität der Empfangsdaten nach [Int07]

3.3.9 Negative Disparity (Compliance)

Zur Generierung des *Compliance Pattern* das zu Testzwecken im Rahmen des *Poling* der LTSSM genutzt wird (siehe [PCI09, S.260]), ist es notwendig, die Disparität einzelner Symbole unabhängig von der *Current Running Disparity* (CRD) zu beeinflussen. Dazu kann die MAC über das Kontrollsignal *TxCompliance* bestimmen, ob die Disparität einzelner Sendedaten auf einen negativen Wert gesetzt werden soll. Das Kontrollsignal wird dazu zeitgleich mit Anlegen der entsprechenden Sendedaten (*TxData*) auf 1 gezogen (Abb. 3.26).

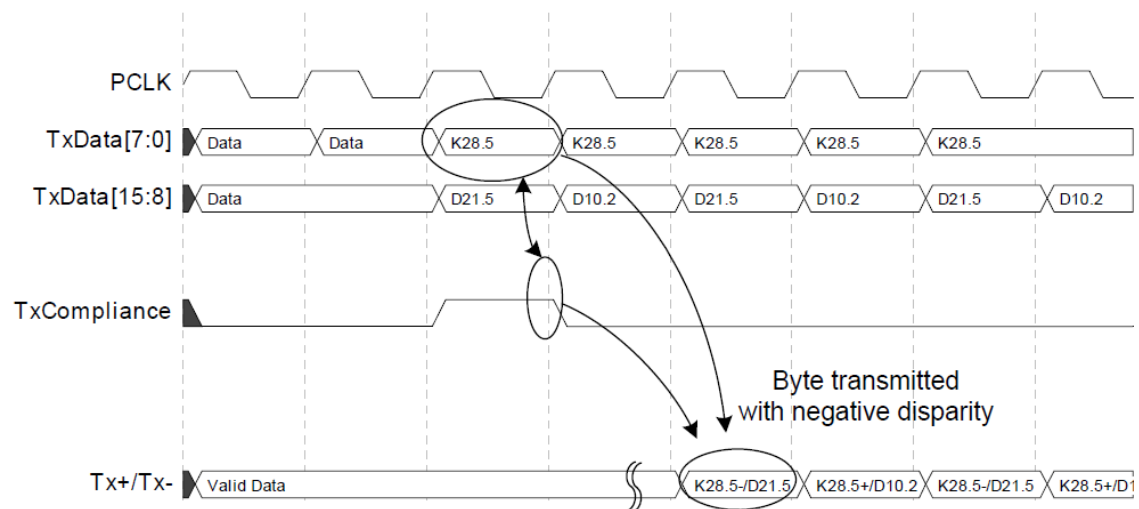


Abb. 3.26: Setzen einer negativen Disparität nach [Int07]

Wird für die Sendedaten ($TxData$), wie in der oberen Abbildung dargestellt, ein 16-Bit breiter Datenpfad eingesetzt, hat die Verwendung des Kontrollsignals lediglich Einfluss auf das *Lower Byte* ([7:0]). Diese Funktionalität ist für die Erzeugung des *Compliance Pattern* unter der Berücksichtigung von jeweils zwei aufeinander folgenden Bytes bzw. Symbolen ausreichend.

3.3.10 Electrical Idle

Damit der PHY auf der Sendeleitung in den *Electrical Idle* übergeht, muss die MAC das Kontrollsignal $TxElecIdle$ auf logisch 1 ziehen. Der PHY verbleibt so lange im *Electrical Idle* Zustand, wie das Kontrollsignal aktiv ist. Die Zeitspanne vom Setzen des Kontrollsignals bis zum Eintreten des *Electrical Idle* Zustands auf der physikalischen Sendeleitung, wird von der PIPE Spezifikation oder der PCI Express Basisspezifikation nicht eindeutig definiert. Die PCIe Basisspezifikation definiert jedoch im Zusammenhang mit dem vorher zu übertragenden *Electrical Idle Ordered Set* (EIOS) eine maximale Zeitspanne $T_{TX-IDLE-SET-TOIDLE}$ von 8 ns zwischen dem letzten übertragenden Bit des EIOS und dem Eintritt in den *Electrical Idle* Zustand (siehe [PCI09, S.270]). Die PIPE Spezifikation führt für den Übergang in den *Electrical Idle* Zustand und der damit verbundenen Übertragung des EIOS ein entsprechendes Beispiel an (Abb. 3.27).

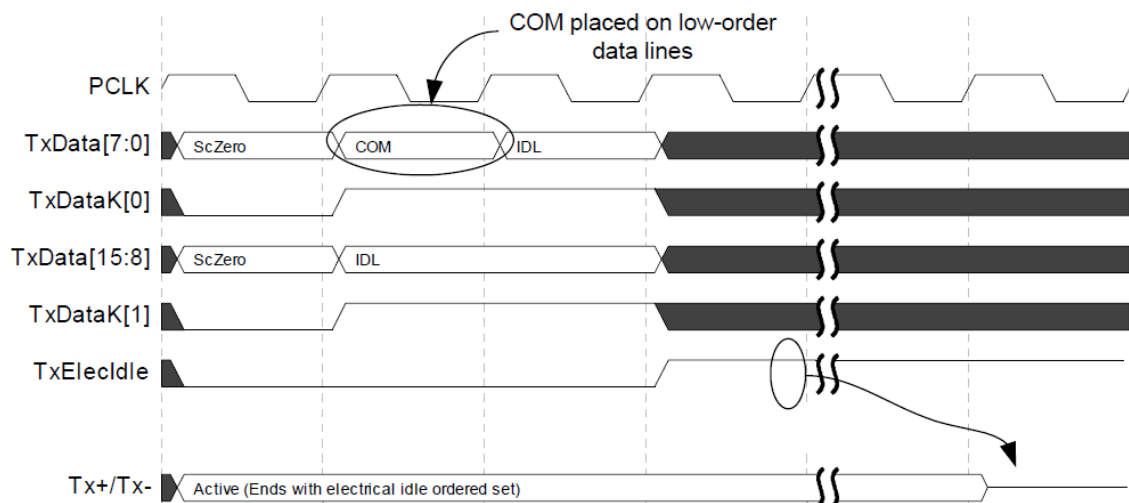


Abb. 3.27: Übergang in den *Electrical Idle* Zustand nach [Int07]

Aus diesem Beispiel geht hervor, dass das letzte Symbol (*IDL*), welches auf das *Higher Byte* ([15:8]) der Schnittstelle für die Sendedaten ($TxData$) gelegt worden ist,

auch das letzte übertragene Byte vor dem Eintritt in den *Electrical Idle* Zustand darstellt. Somit scheint das Anlegen des Kontrollsignals *TxElecIdle* einen PCLK Takt später so aufeinander abgestimmt zu sein, dass die Übertragung der vorherigen Sendedaten passend mit dem Eintritt in den *Electrical Idle* zusammen fällt. Der zeitliche Zusammenhang wird jedoch nicht explizit erwähnt oder beschrieben, weshalb hier auch eine spätere Zuweisung des Kontrollsignals *TxElecIdle* denkbar wäre, um den passenden Zeitpunkt über die MAC zu erzeugen.

3.4 SerDes Architektur

Im folgenden Kapitel wird die Architektur und die grundlegende Funktionsweise des im GateMate FPGA integrierten Serialisierer/Deserialisierers (SerDes) beschrieben. Der SerDes setzt sich im Kern aus einer *All Digital Phase Locked Loop* (ADPLL), einem *Transmitter* und einem *Receiver* zusammen (Abb. 3.28). Die Konfiguration des SerDes erfolgt über einen integrierten Registersatz (*Register File*), welcher sowohl von der Anwenderseite als auch durch einen *Configuration Controller* zugänglich ist. Durch den *Configuration Controller* wird beim Resetvorgang des FPGAs die gewünschte Konfiguration (*Config File*) des SerDes in den Registersatz geschrieben. Diese Vorgehensweise wird für den zukünftigen Einsatz in SerDes-Anwendungen priorisiert, da eine erneute Anpassung der Konfiguration während des laufenden Betriebs durch den Anwender nicht explizit vorgesehen ist. Der Tx-/Rx-Datenpfad des SerDes setzt sich schematisch aus den in Abb. 3.29 gezeigten Komponenten zusammen, welche zum Großteil über das FPGA Interface angesteuert werden können. Innerhalb des Tx-Datenpfad verfügt der SerDes im PCS über einen 8b/10b Encoder zur Encodierung der Sendedaten, einen *Pattern Generator* zur Erzeugung eines *Pseudo Random Bit Stream* (PRBS) und einen Funktionsblock zur Invertierung der Polarität der Sendedaten (*Polarity*). Der dargestellte Funktionsblock *Phase Adjust FIFO*, der im Rahmen der Port-Beschreibungen auch als *Tx Buffer* bezeichnet wird, dient dazu, die Phasenverschiebung zwischen den verwendeten *Clock Domains* in PCS und PMA abzugleichen. Der Funktionsblock wird in der SerDes Spezifikation [Rac20] nicht genauer beschrieben, aber scheint sich analog zum GTH/GTX Transceiver von Xilinx zu verhalten [Xil18, S.133]. Der vorherige Phasenabgleich ist für die Übergabe der parallelen Daten an die serielle Übertragung zwingend erforderlich, welche durch den Funktionsblock PISO (*Parallel In Serial Out*) dargestellt wird. Der eigentliche *Transmitter* bzw. *Tx-Driver* auf Leitungsebene ermöglicht für die Übertragung eine *Feed-Forward Equalization*. Die analogen Betriebsparameter des *Tx-Driver* lassen sich über den Registersatz des SerDes frei konfigurieren. Analog

zum Tx-Datenpfad verfügt der SerDes auf der Seite des Rx-Datenpfads im PCS über einen Funktionsblock zur Invertierung der Empfangsdaten (*Polarity*), eine Kontrolleinheit zur Überprüfung des PRBS (*Checker*) und einen entsprechenden 8b/10b Decoder zur Dekodierung der Empfangsdaten. Der Rx-Datenpfad besitzt des Weiteren einen Funktionsblock für die *Comma Detection* und das *Symbol Alignment* sowie einen *Elastic Buffer* zum Ausgleich der Takttoleranzen. Der SerDes verfügt ebenfalls über die notwendige Logik für die *Clock Data Recovery* (CDR), welche jedoch nicht im Blockschaltbild dargestellt ist. Das *Analog Frontend* des *Receivers* (PMA) verfügt über einen *Linear Equalizer* (RX EQ) dem ein *Decision Feedback Equalizer* (DFE) nachfolgt. Für bestimmte Anwendungsfälle verfügt der SerDes über mehrere gerichtete Verbindungen zwischen dem Tx- und Rx-Datenpfad, die genutzt werden können, um einen *Loopback* der Daten auf bestimmten Ebenen des Datenpfades zu ermöglichen.

In den nachfolgenden Kapiteln werden die wichtigsten Funktionen des SerDes im Zusammenhang mit den Entwicklungen dieser Masterthesis beschrieben. Für eine Erläuterung, der nicht näher thematisierten Funktionen sowie zusätzlichen Schnittstellen und Register siehe [Rac20]. Eine Übersicht der vorhandenen Schnittstellen des SerDes, findet sich zudem in Anhang A.2. Dabei ist zu beachten, dass einige der Schnittstellen nicht von der FPGA Fabric aus zugänglich sind.

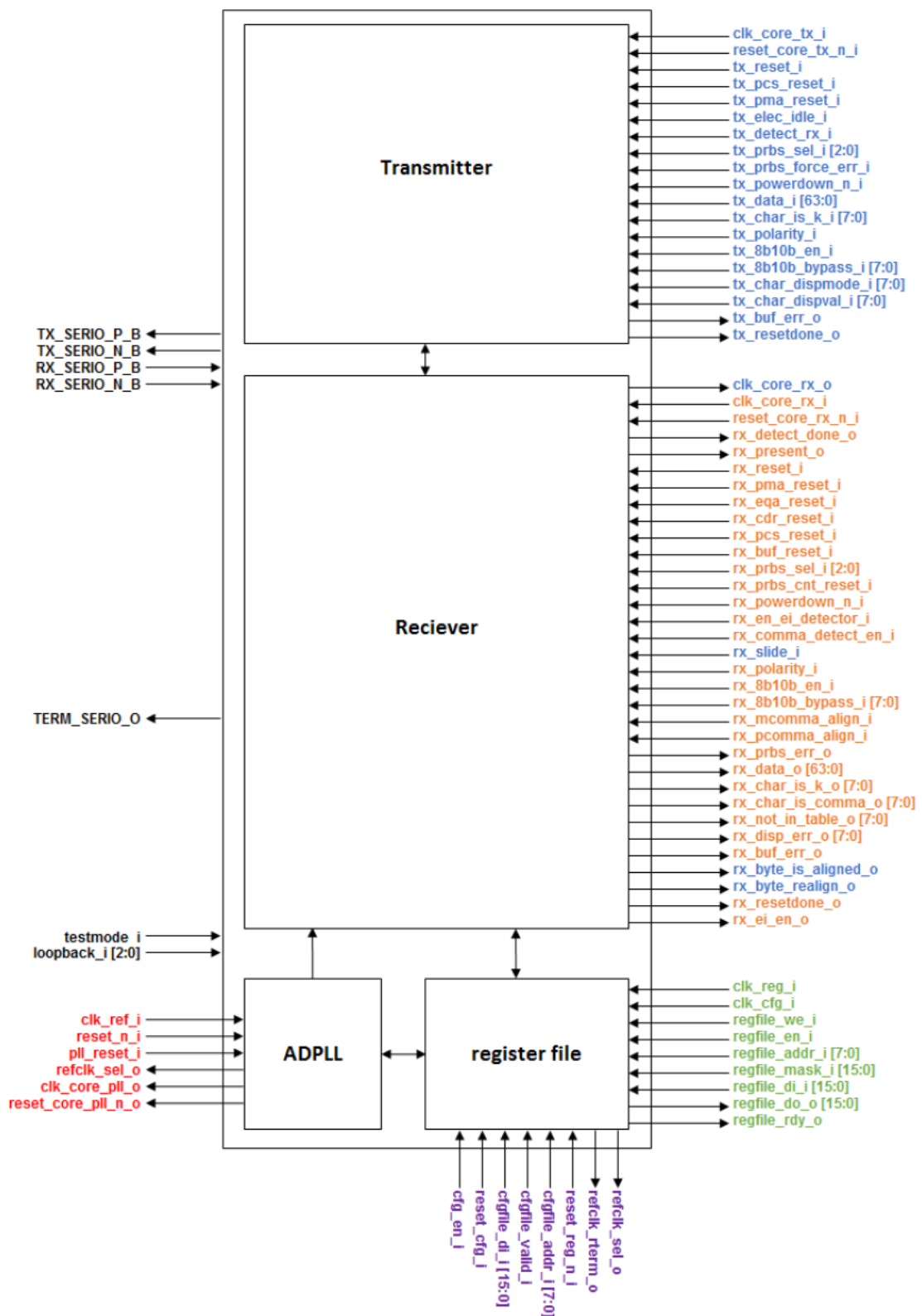


Abb. 3.28: Blockschaltbild des SerDes nach [Rac20]

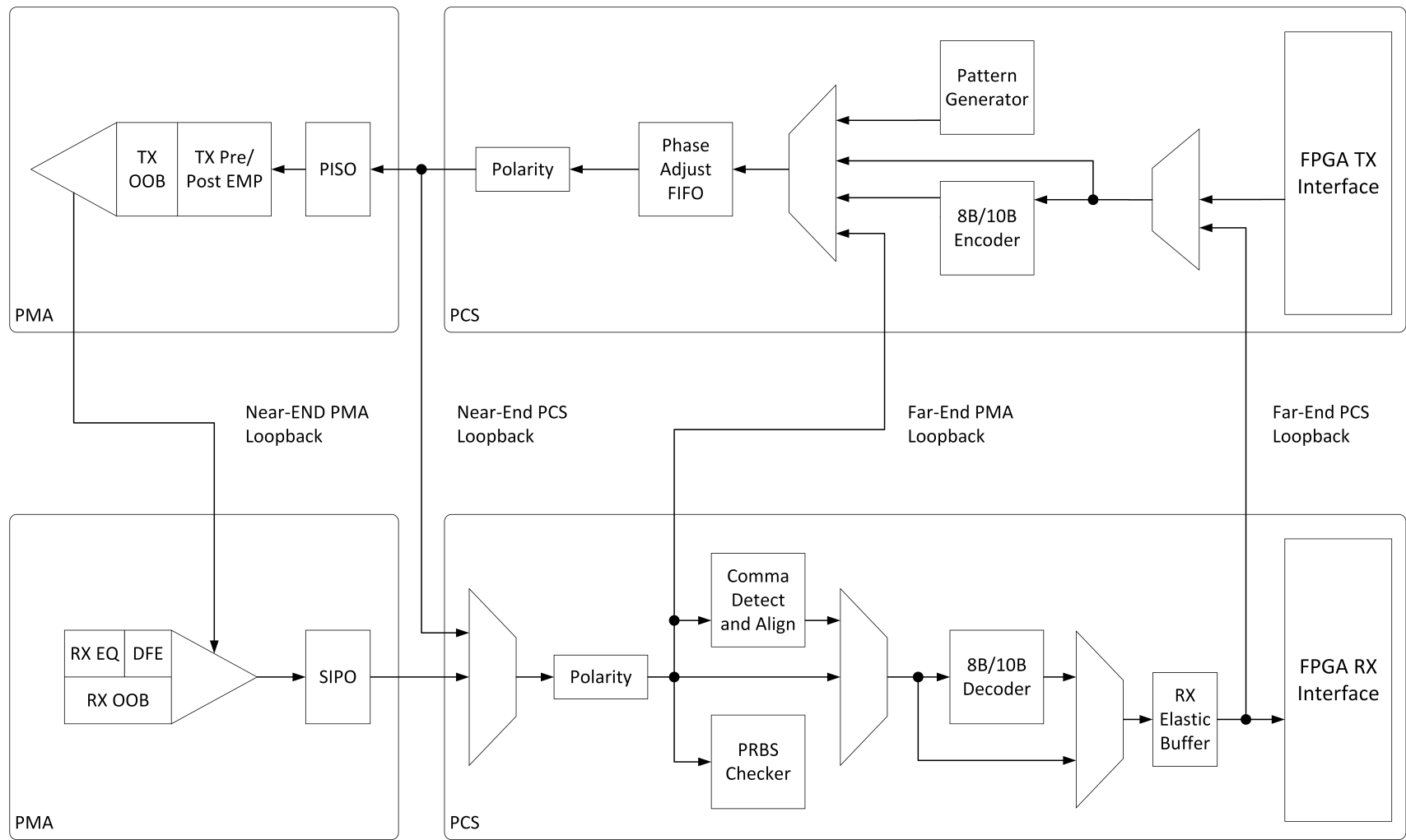


Abb. 3.29: Blockschaltbild des Tx/Rx-Datenpfads nach [Rac20]

3.4.1 ADPLL

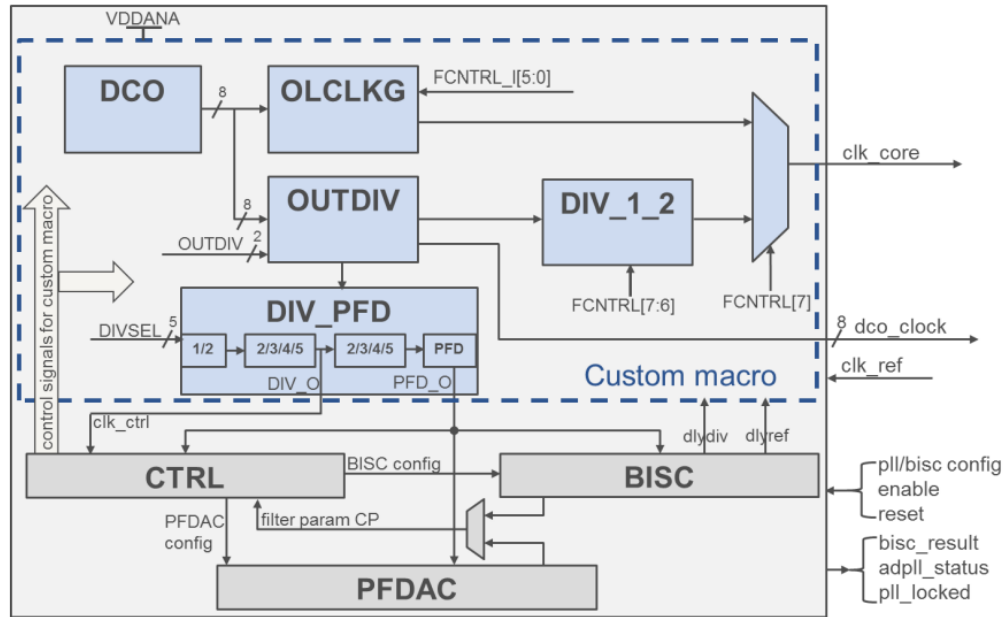


Abb. 3.30: Vereinfachtes Blockschaltbild der ADPLL nach [Rac20]

Die ADPLL des SerDes generiert aus dem anliegenden Referenztakt (clk_ref) die notwendige *Bit Rate Clock* (brc) bzw. dco_clock für die serielle Übertragung. Zusätzlich wird der notwendige Takt für den Betrieb des Rx- und Tx- Datenpfads auf PCS-Ebene (clk_core) erzeugt (Abb. 3.30). Dieses als *Core Clock* bezeichnete Taktsignal wird von der ADPLL über die Ausgangsschnittstelle $clk_core_pll_o$ bereitgestellt und muss innerhalb der FPGA-Logik mit den beiden Schnittstellen $clk_core_tx_i$ und $clk_core_rx_i$ des *Transmitters* und *Receivers* verbunden werden. Alternativ kann für den Rx-Datenpfad, das aus den Empfangsdaten gewonnene Taktsignal verwendet werden. Dieser Anwendungsfall tritt ein, wenn anstelle einer *Common Refclk Rx Architecture* eine entsprechende *Data Clocked Rx Architecture* vorliegt [PCI09, S.309]. Die Ausgabe des Taktsignals erfolgt dabei durch den Port $clk_core_rx_o$. Die *Core Clock* dient zusätzlich als Referenz für den Datentransfer der parallelen Schnittstellen zwischen FPGA und SerDes. Die Taktfrequenzen der *Bit Rate Clock* und der *Core Clock* ergeben sich bei einem Referenztakt von 100 MHz nach den folgenden Formeln.

$$dco_clock = brc = 100 \text{ MHz} \cdot N1 \cdot N2 \cdot N3 \cdot \frac{1}{OutDiv} \quad (3.3)$$

$$clk_core_pll_o = \frac{100 \text{ MHz} \cdot N1 \cdot N2 \cdot N3}{N(OLCLKGEN)} \cdot \frac{1}{AddDiv} \quad (3.4)$$

Die *Bit Rate Clock* berechnet sich aus den Teilerwerten N1-N3, sowie einem zusätzlichen Teiler für das ausgegebene Taktsignal (*OutDiv*). Die *Core Clock* leitet sich analog zur *Bit Rate Clock* ab und wird allerdings über den Teiler N(OLCLKGen) und *AddDiv* bestimmt. Die Parameter besitzen einen vordefinierten Wertebereich und können über die korrespondierenden Registerfelder des SerDes konfiguriert werden (Tab. 3.8).

Name	Bits	Default Value	Description
PLL_MAIN_DIVSEL	6	27	ADPLL main loop divider N=N1*N2*N3: PLL_MAINDIV[1:0]: 00: N2=3 01: N2=2 10: N2=4 11: N2=5 PLL_MAINDIV[2]: 0: N1=1 1: N1=2 PLL_MAINDIV[4:3]: 00: N3=3 01: N3=FORBIDDEN 10: N3=4 11: N3=5 PLL_MAINDIV[5]: not used
PLL_OUT_DIVSEL	2	0	ADPLL output divider (<i>OutDiv</i>): 00: Divide by 1 01: Divide by 2 10: n.a. 11: Divide by 4
PLL_FCNTL	6	58	frequency divider selection for SerDes TX clock output to FPGA core N(OLCLKGEN)

Tabelle 3.8: Auszug der ADPLL-Registerfelder nach [Rac20]

Anmerkung: Die in der SerDes Spezifikation enthaltene Beschreibung für die Teilerwerte N1-N3 in PLL_MAIN_DIVSEL ist fehlerhaft. Die Tabelle 3.8 enthält die bereits korrigierte Beschreibung mit den richtigen Werten. Für die Dekodiertabelle der möglichen Teilerwerte für N(OLCLKGEN) in PLL_FCNTL siehe [Rac20, S.12].

Der zusätzliche Teiler *AddDiv* lässt sich im Gegensatz zu den anderen Parametern nicht direkt setzen, sondern ist indirekt vom Wert für *OutDiv* und der Verwendung des 64-Bit Datenpfads abhängig. Der Teilerwert von *AddDiv* wird dabei innerhalb des Registermoduls wie folgt bestimmt (Abb. 3.9).

PLL_OUT_DIVSEL		AddDiv	
Binary	Value	Binary	Value
00	1	010	2
01	2	100	4
10	-	010	2
11	4	010	2

Tabelle 3.9: Teilerwerte von *AddDiv* (64-Bit Datenpfad)

Mit Bezug auf die *Bit Rate Clock* bzw. *dco_clock* muss beachtet werden, dass die ADPLL mehrere phasenverschobene Taktsignale generiert. Diese werden vom SerDes genutzt, um für die vorliegende *Bit Rate Clock* die doppelte Übertragungsrate zu erzielen. Für eine Übertragungsrate von 2,5 Gbit/s ergibt sich demnach eine notwendige Frequenz von 1,25 GHz, welche mit folgenden Einstellungen realisiert werden kann.

$$dco_clock = 100 \text{ MHz} \cdot 1 \cdot 5 \cdot 5 \cdot \frac{1}{2} \quad (3.5)$$

$$= 2,50 \text{ GHz} \cdot \frac{1}{2} \quad (3.6)$$

$$= 1,25 \text{ GHz} \quad (3.7)$$

Ausgehend von einer Übertragungsrate von 2,5 Gbit/s und einem 64-Bit breiten

Datenpfad ergibt sich für die *Core Clock* eine notwendige Frequenz von 31,25 MHz. Es ist zu beachten, dass die Übertragung der 64-Bit nach der 8b/10b-Encodierung als 80-Bit erfolgt.

$$\frac{2,5 \text{ Gbit/s}}{80 \text{ Bit}} = 31,25 \text{ MHz} \quad (3.8)$$

Die Frequenz kann über den Teiler N(OLCCLKGEN) und unter Berücksichtigung des Teilers *AddDiv* wie folgt eingestellt werden. Für N(OLCCLKGEN) wird der vordefinierte Wert von 20 genutzt.

$$clk_core_pll_o = 100 \text{ MHz} \cdot 1 \cdot 5 \cdot 5 \cdot \frac{1}{20} \cdot \frac{1}{4} \quad (3.9)$$

$$= \frac{2,5 \text{ GHz}}{20 \cdot 4} \quad (3.10)$$

$$= 31,25 \text{ MHz} \quad (3.11)$$

Der Teilerwert von *AddDiv* entspricht hierbei dem Wert 4, wenn von einem Teilerwert von 2 für *OutDiv* ausgegangen wird (Tab. 3.9).

3.4.2 Reset

Der SerDes verfügt über mehrere Schnittstellen, welche über die FPGA-Logik zugänglich sind und für die Durchführung unterschiedlicher Resets innerhalb der SerDes-Architektur genutzt werden können (Tab. 3.10).

Name	Typ	Active	Beschreibung
pll_reset_i	Input	High	Asynchroner Reset ADPLL.
tx_reset_i	Input	High	Asynchroner Reset Tx-Datenpfad.
tx_pcs_reset_i	Input	High	Asynchroner Reset Tx-Datenpfad (PCS).
tx_pma_reset_i	Input	High	Asynchroner Reset Tx-Datenpfad (PMA).

rx_reset_i	Input	High	Asynchroner Reset Rx-Datenpfad
rx_pcs_reset_i	Input	High	Asynchroner Reset Rx-Datenpfad (PCS).
rx_cdr_reset_i	Input	High	Asynchroner Reset Rx-Datenpfad (CDR).
rx_eqa_reset_i	Input	High	Asynchroner Reset Rx-Datenpfad (DFE).

Tabelle 3.10: Schnittstellen zur Durchführung verschiedener Resets

Bei den Reset-Signalen handelt es sich vollständig um asynchrone Signale. Für den normalen Betrieb sind vor allem der Reset der ADPLL sowie des Tx- und Rx-Datenpfads von Bedeutung. Für die beiden Datenpfade hält der SerDes zusätzliche Schnittstellen für den derzeitigen Reset-Status bereit (Tab. 3.11).

Name	Typ	Active	Beschreibung
tx_resetdone_o	Output	High	Die Schnittstelle zeigt den Reset-Status des Tx-Datenpfads an. Die Ausgabe wechselt von 0 auf 1 wenn der Reset abgeschlossen ist. Das Signal bleibt so lange aktiv (1) bis ein erneuter Reset ausgelöst wird.
rx_resetdone_o	Output	High	Die Schnittstelle zeigt den Reset-Status des Rx-Datenpfads an. Die Ausgabe wechselt von 0 auf 1 wenn der Reset abgeschlossen ist. Das Signal bleibt so lange aktiv (1) bis ein erneuter Reset ausgelöst wird.

Tabelle 3.11: SerDes Schnittstellen für den Reset-Status der Datenpfade

Für die Funktionsweise des SerDes ist es wichtig zu wissen, dass die Resets der einzelnen Bestandteile intern miteinander verknüpft sind. Ein Reset des Tx- oder Rx-Datenpfads wird immer auch einen Reset des zugehörigen PCS, PMA, CDR oder DFE auslösen. Für den Rx-Datenpfad schließt dies auch den *Elastic Buffer* mit ein. Ein besonderer Zusammenhang ergibt sich zwischen dem Zustand der ADPLL und den beiden Datenpfaden. Die ADPLL hält den Tx- und Rx-Datenpfad so lange im Reset-Zustand bis ein entsprechender *Lock* der PLL vorliegt und es möglich

ist die benötigte *Core Clock* bereitzustellen. Ein Reset der ADPLL hat daher auch immer einen Reset der beiden Datenpfade zur Folge. Dadurch ergibt sich für den Resetvorgang des FPGAs und der damit verbundenen Konfiguration des SerDes eine automatische Verkettung der Resets. Diese beginnt mit der Aktivierung der ADPLL über das Registerfeld PLL_EN_ADPLL_CTRL und findet mit den Flankenwechseln der Statussignale tx_resetdone_o und rx_resetdone_o ihren Abschluss. Zu beachten ist hierbei auch, dass der Rx-Datenpfad seinen Reset-Zustand zeitlich verzögert nach dem Tx-Datenpfad verlässt.

3.4.3 Tx-/Rx-Datenpfad

Der SerDes verfügt im Tx- und Rx-Datenpfad über 64-Bit breite Schnittstellen für die Sende- und Empfangsdaten, welche durch eine 8-Bit breite Schnittstelle zur Definition des Datentyps (Daten- oder Kontrollsymbol) ergänzt werden (Tab. 3.12 und 3.13).

Name	Typ	Active	Beschreibung
tx_data_i[63:0]	Input	N/A	Schnittstelle für die Sendedaten der PCI Express Kommunikation. Die Übertragung beginnt immer mit dem <i>Lowest Byte</i> ([7:0]).
tx_char_is_k_i[7:0]	Input	N/A	Schnittstelle für die Definition der übertragenen Symbole. Definiert ob es sich um einen Daten- oder Kontrollsymbol handelt. Ein Bit repräsentiert jeweils ein Byte von tx_data_i. Eine 1 definiert das Symbol als Kontrollsymbol und eine 0 als Datensymbol. Wird für den 8b/10b Decoder benötigt.

Tabelle 3.12: SerDes Schnittstellen für die Sendedaten

Name	Typ	Active	Beschreibung
rx_data_o[63:0]	Output	N/A	Schnittstelle für die Empfangsdaten der PCI Express Kommunikation. Das <i>Lowest Byte</i> ([7:0]) stellt das erste empfangene Symbol dar.

rx_char_is_k_o[7:0]	Output	N/A	Schnittstelle für die Definition der empfangenen Symbole. Definiert ob es sich um einen Daten- oder Kontrollsymbol handelt. Ein Bit repräsentiert jeweils ein Byte von rx_data_o. Eine 1 definiert das Symbol als Kontrollsymbol und eine 0 als Datensymbol.
---------------------	--------	-----	--

Tabelle 3.13: SerDes Schnittstellen für die Empfangsdaten

Name	Typ	Active	Beschreibung
rx_char_is_comma_o[7:0]	Output	N/A	Schnittstelle gibt an ob es sich bei dem korrespondierenden Byte um ein COM Symbol handelt. Ein Bit repräsentiert jeweils ein Byte von rx_data_o. Eine 1 weist auf ein COM Symbol hin.

Tabelle 3.14: SerDes Schnittstelle für die Empfangsdaten (COM Symbol)

Zusätzlich verfügt der SerDes über die Schnittstelle rx_char_is_comma_o, die den Anwender darüber informiert, ob es sich bei den entsprechenden Bytes um ein COM Symbol handelt (Tab. 3.14). Als COM Symbol wird hierbei nicht nur das für PCIe definierte COM Symbol K28.5 (0xBC) erkannt, sondern auch die Symbole K28.1 (0x3C) und K28.7 (0xFC). Für die Sendedaten ergibt sich für die Zeit zwischen der Übernahme an tx_data_i bis zum Erscheinen des ersten Datenbits auf der physikalischen Leitung, eine Verzögerung von 180ns (*Transmit Latency*). Im Gegenzug ergibt sich für die Zeit vom Auftreten des ersten Bits eines 64-Bit Datenwortes bis hin zur Ausgabe der Daten über rx_data_o eine Verzögerung von 480ns (*Receive Latency*).

Disparität der Sendedaten

Die Anpassung der Disparität (*Disparity*) der Sendedaten erfolgt beim SerDes über die Kombination der Schnittstellen tx_char_dispmode_i und tx_char_dispval_i (Tab. 3.15).

Name	Typ	Active	Beschreibung
tx_char_dispmode_i[7:0]	Input	N/A	Schnittstelle wird genutzt, um die Disparität zu überschreiben. Ein Bit repräsentiert jeweils ein Byte von tx_data_i. Wird das Signal auf 1 gezogen, wird für das korrespondierende Byte die durch tx_char_dispval_i definierte Disparität übernommen. Funktion ist abhängig vom Einsatz des 8b/10b Decoders.
tx_char_dispval_i[7:0]	Input	N/A	Schnittstelle definiert die Art der Disparität, wenn diese durch tx_char_dispmode_i überschrieben wird. Ein Bit repräsentiert jeweils ein Byte von tx_data_i. Eine 1 definiert das Symbol als positive Disparität und eine 0 als negative Disparität. Funktion ist abhängig vom Einsatz des 8b/10b Decoders.

Tabelle 3.15: SerDes Schnittstellen für die Disparität der Sendedaten

Wird der 8b/10b Encoder über die Schnittstelle tx_8b10b_en_i ausgeschaltet dienen die beiden Schnittstellen tx_char_dispmode_i und tx_char_dispval_i als Erweiterung der Schnittstelle tx_data_i auf eine Bitbreite von 80-Bit. Für genauere Informationen über die Erweiterung der Schnittstelle siehe [Rac20, S.16, Tab.11].

Fehlererkennung in den Empfangsdaten

Der SerDes ist in der Lage die Fehler in der 8b/10b-Kodierung oder der Disparität zusammen mit den Empfangsdaten über die Schnittstellen rx_not_in_table_o und rx_disp_err_o auszugeben (Tab. 3.16).

Name	Typ	Active	Beschreibung
rx_not_in_table_o[7:0]	Output	N/A	Schnittstelle wird genutzt, um einen 8b/10b Fehler in der Dekodierung der Empfangsdaten anzugeben. Ein Bit repräsentiert jeweils ein Byte von rx_data_o. Eine 1 weist darauf hin, dass beim zugehörigen Byte ein entsprechender Fehler vorliegt. Funktion ist abhängig vom Einsatz des 8b/10b Encoders.
rx_disp_err_o[7:0]	Output	N/A	Schnittstelle wird genutzt, um einen Fehler in der Disparität anzugeben. Ein Bit repräsentiert jeweils ein Byte von rx_data_o. Eine 1 weist darauf hin, dass bei der Dekodierung des zugehörigen Byte ein Fehler in der Disparität aufgetreten ist. Funktion ist abhängig vom Einsatz des 8b/10b Encoders.

Tabelle 3.16: SerDes Schnittstellen für die Fehlererkennung

Wird der 8b/10b Encoder über die Schnittstelle rx_8b10b_en_i ausgeschaltet, dienen die beiden Schnittstellen tx_char_dispmode_i und tx_char_dispval_i als Erweiterung der Schnittstelle rx_data_o auf eine Bitbreite von 80-Bit. Die Erweiterung der Schnittstelle erfolgt dabei analog zum *Transmitter* [Rac20, S.16, Tab.11].

Die Angabe von *Overflows* oder *Underflows* des *Elastic Buffers* erfolgt über die Schnittstelle rx_buf_err_o. Diese unterscheidet jedoch nicht zwischen den beiden Möglichen Fehlerfällen und steht auch in keinem direkten Bezug zur Ausgabe der Empfangsdaten über rx_data_o (Tab. 3.17).

Name	Typ	Active	Beschreibung
rx_buf_err_o	Input	High	Schnittstelle wird genutzt, um einen <i>Overflow</i> oder <i>Underflow</i> des <i>Elastic Buffer</i> anzuzeigen (1). Wird nur durch einen Reset des <i>Elastic Buffers</i> über rx_buf_reset_i zurückgesetzt.
rx_buf_reset_i	Input	High	Schnittstelle wird genutzt, um einen Reset des <i>Elastic Buffers</i> auszulösen.

Tabelle 3.17: SerDes Schnittstellen für die Fehlererkennung

Eine Angabe durch den *Elastic Buffer*, ob SKP Symbole in die Datenübertragung integriert oder entfernt worden sind, ist im Design des SerDes nicht vorgesehen.

3.4.4 Tx Configurable Driver

Über die Schnittstellen des *Tx Configurable Driver* kann der SerDes in den *Electrical Idle* Zustand auf der Sendeleitung überführt werden, sowie die Durchführung einer *Receiver Detection* veranlasst werden (Tab. 3.18).

Name	Type	Active	Beschreibung
tx_elec_idle_i	Input	High	Die Schnittstelle wird genutzt, um den SerDes in den <i>Electrical Idle</i> Zustand zu überführen.
tx_detect_rx_i	Input	High	Die Schnittstelle dient dazu einer <i>Receiver Detection</i> zu starten. Signal sollte aktiv bleiben bis die <i>Receiver Detection</i> abgeschlossen und ausgelesen ist.
rx_detect_done_o	Output	High	Zeigt an in welchem Zustand sich die <i>Receiver Detection</i> befindet. Bei einer 1 ist die <i>Receiver Detection</i> abgeschlossen und das entsprechende Ergebnis kann über rx_present_o ausgelesen werden.

rx_present_o	Output	N/A	Zeigt das Ergebnis der Receiver Detection an. Bei einer 1 konnte ein <i>Receiver</i> am anderen Ende der Leitung detektiert werden. Bei einer 0 ist kein <i>Receiver</i> vorhanden.
--------------	--------	-----	---

Tabelle 3.18: Schnittstellen für *Electrical Idle* und *Receiver Detection*

Im Fall der *Receiver Detection* wird die Detektierung über das Signal tx_detect_rx_i gestartet, wobei das Signal so lange anliegen muss, bis über rx_detect_done_o der Abschluss der Detektierung angezeigt wird und das entsprechende Ergebnis über rx_present_o ausgelesen werden kann. Der Übergang in, sowie aus dem *Electrical Idle* Zustand tritt nach dem Kenntnisstand zum Zeitpunkt dieser Masterthesis mit einer Verzögerung von 48ns ein. Die Einstellung der elektrotechnischen Parameter für die *Common Mode Voltage*, den *Electrical Idle* Zustand und die *Receiver Detection*, erfolgt über spezifische Register des SerDes. Je nach Anliegen der entsprechenden Signale an tx_elec_idle_i und tx_detect_rx_i greift der SerDes auf den korrespondierenden Registersatz zu und passt das Verhalten seines gesteuerten Stromspiegels an. Daher liegen dessen Konfigurationsparameter für jeden Anwendungsfall in einem individuellen Registersatz vor. Die Konfiguration der gesamten Parameter innerhalb der Register erfolgt während des Resetvorgangs durch den *Configuration Controller*. Für genauere Informationen über die Funktionsweise des Leitungstreibers und der vorliegenden Parameter siehe [Rac20, S.22].

3.4.5 Rx Analog Front End

Der *Receiver* besitzt für die Erkennung des *Electrical Idle* Zustands auf der Empfangsleitung einen entsprechenden Detektor. Dieser ist über die Schnittstellen in Tabelle 3.19 zugänglich und sollte für den Einsatz im PCIe immer aktiv sein.

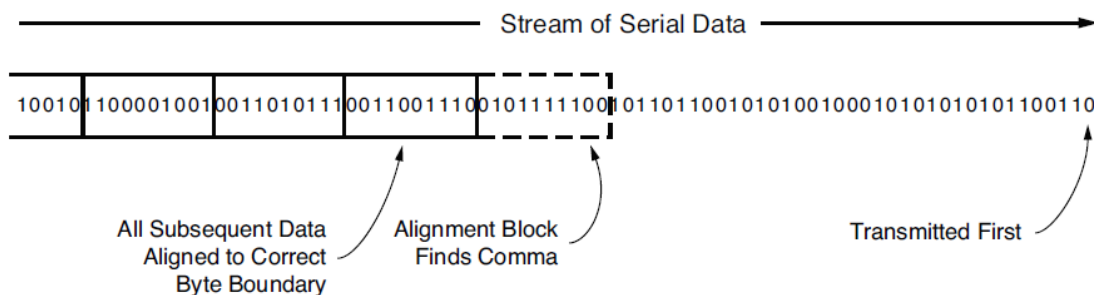
Name	Type	Active	Beschreibung
rx_en_ei_detector_i	Input	High	Schnittstelle wird genutzt, um die Detektierung des <i>Electrical Idle</i> Zustands für die Empfangsleitung zu aktivieren. Bei einer 1 ist der entsprechende Detektor aktiv.

rx_ei_en_o	Input	High	Die Schnittstelle gibt Auskunft, ob auf der Empfangsleitung der <i>Electrical Idle</i> Zustand anliegt. Bei einer 1 konnte <i>Electrical Idle</i> detektiert werden. Bei einer 0 liegt kein <i>Electrical Idle</i> vor.
------------	-------	------	---

Tabelle 3.19: Schnittstellen für die Detektierung von *Electrical Idle*

3.4.6 Rx Word Alignment

Das *Word Alignment* muss bei Empfang einer Übertragung durch den *Receiver* erfolgen, damit dieser aus dem empfangenen *Bitstream* den Anfang und das Ende eines 10-Bit Datenwortes ableiten kann. Das *Word Alignment* erfolgt über die Detektierung eines COM Symbols im *Bitstream* und sorgt dafür, dass die ankommenden Daten auf die internen *Byte Boundaries* abgebildet werden können (Abb. 3.31).

Abb. 3.31: Schematischer Ablauf des *Word Alignment* nach [Xil18]

Das *Word Alignment* wird alternativ auch als *Symbol* oder *Byte Alignment* bezeichnet. Der SerDes verfügt für die Durchführung des *Word Alignment* über die entsprechende Logik und kann über die Registerfelder `ALIGN_MCOMMA_VALUE` und `ALIGN_PCOMMA_VALUE` auch auf die Detektierung anderer Symbole eingestellt werden. Die vordefinierten Werte enthalten jedoch die positive und negative 10-Bit Codierung für das im PCIe eingesetzte COM Symbol (K28.5), weshalb eine Anpassung nicht notwendig ist. Der SerDes erlaubt für die internen *Byte Boundaries* eine Einstellung des *Alignments* von 8-, 16- oder 32-Bit. Durch diese Einstellung wird nach der Detektierung des COM Symbols die Position des Datenworts, sowie der

nachfolgenden Datenworte mit Bezug auf den 64-Bit breiten Datenpfad bestimmt (Abb 3.32).

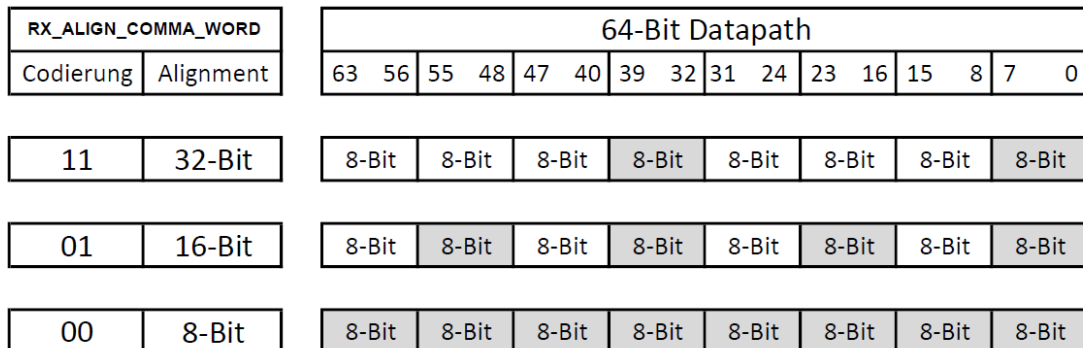


Abb. 3.32: Konfigurierbares *Alignment* für die *Byte Boundaries*

In der oben gezeigten Abbildung weisen grau hinterlegte Felder auf die mögliche Positionierung des COM Symbols durch das *Alignment* hin. Das *Alignment* wird über das Registerfeld RX_ALIGN_COMMA_WORD konfiguriert und besitzt als vordefinierte Konfiguration ein 8-Bit *Alignment*. Für die Durchführung des *Word Alignment* verfügt der SerDes über die in Tabelle 3.20 aufgeführten Schnittstellen, wobei die Schnittstelle rx_slide_i für die Durchführung eines manuellen *Alignments* nicht aufgeführt ist.

Name	Type	Active	Beschreibung
rx_byte_is_aligned_o	Output	High	Zeigt an ob die eingehenden COM Symbole erfolgreich detektiert und auf die <i>Byte Boundaries</i> angeglichen werden konnten. Es besteht <i>Symbol Lock</i> .

rx_byte_realign_o	Output	High	Das Signal zeigt an ob ein COM Symbol detektiert worden ist, das nicht mit den <i>Byte Boundaries</i> übereinstimmt und angepasst wurde, falls das <i>Symbol Alignment</i> noch aktiviert ist. Das Signal bleibt in diesem Fall für eine Taktflanke aktiv. Bei der fallenden Flanke des Signals, sollte rx_byte_is_aligned_o wieder auf 1 wechseln.
rx_mcomma_align_i	Input	High	Aktiviert das <i>Symbol Alignment</i> für die negative Disparität des COM Symbols (8b/10b-Codierung) .
rx_pcomma_align_i	Input	High	Aktiviert das <i>Symbol Alignment</i> für die positive Disparität des COM Symbols (8b/10b-Codierung).
rx_comma_detect_en_i	input	High	Aktiviert (1) oder Deaktiviert (0) die Detektierung von COM Symbolen und das <i>Symbol Alignment</i> .

Tabelle 3.20: Schnittstellen für das *Symbol* bzw. *Word Alignment*

Im normalen Betrieb muss das automatische *Word Alignment* zunächst über die Schnittstelle rx_comma_detect_en_i aktiviert werden. Anschließend lässt sich dieses über die Schnittstellen rx_mcomma_align_i und rx_pcomma_align_i für die positive und negative Codierung des COM Symbols separat aktivieren. Eine separate Ansteuerung ist jedoch in den seltensten Fällen erforderlich, weshalb diese immer zeitgleich aktiv sind. Ist das *Word Alignment* erfolgreich durchgeführt worden, wird dies über die Schnittstelle rx_byte_is_aligned_o angezeigt. Man spricht in diesem Zusammenhang auch von *Symbol* oder *Byte Lock*. Ab diesem Zeitpunkt kann auf zwei unterschiedliche Weisen verfahren werden. Entweder wird das automatische *Word Alignment* über die Schnittstellen rx_mcomma_align_i und rx_pcomma_align_i wieder ausgeschaltet oder es bleibt weiterhin aktiv. Im ersten Fall bleibt das *Word Alignment* bzw. *Symbol Lock* zwar weiterhin bestehen, wird jedoch nicht automatisch angepasst, wenn ein COM Symbol außerhalb der *Byte Boundaries* auftreten

sollte. Im zweiten Fall sorgt die bestehende Aktivierung dafür, dass der *Receiver* versucht das *Word Alignment* selbstständig wieder herzustellen. Falls dies gelingt, gibt der *Receiver* zunächst einen Impuls von einer Taktperiode über die Schnittstelle `rx_byte_realign_o` aus, bevor `rx_byte_is_aligned_o` wieder auf den Wert 1 übergeht. In beiden Fällen wird der fehlende *Symbol Lock* über eine 0 an der Schnittstelle `rx_byte_is_aligned_o` angezeigt.

3.4.7 Loopback

Der SerDes unterstützt verscheide *Loopback* Modi, welche an bestimmten Punkten innerhalb seiner Architektur für eine Rückführung der Daten sorgen. Um einen *Loopback* zu aktivieren, kann die entsprechende Codierung an die Schnittstelle `loopback_i` angelegt werden (Tab. 3.21).

Name	Type	Active	Beschreibung
<code>loopback_i[2:0]</code>	input	N/A	000 normal operation 001 near-end PCS Loopback 010 near-end PMA Loopback 011 reserved 100 far-end PMA Loopback 101 reserved 110 far-end PCS Loopback 111 reserved

Tabelle 3.21: Schnittstelle für die Aktivierung des *Loopback* Modus

Den wichtigsten *Loopback* Modus für den Einsatz im PCI Express stellt der *far-end PCS Loopback* dar. Dieser Modus greift die Empfangsdaten im *Receiver* direkt hinter dem *Elastic Buffer* ab und leitet sie als Sendedaten an den *Transmitter* weiter (Abb. 3.29).

Anmerkung: Das in der SerDes Spezifikation gezeigte Blockschaltbild enthält einen Darstellungsfehler, da es den Abgriff des *far-end PCS Loopback* vor dem *Elastic*

Buffer zeigt [Rac20, S.8]. Bei Abb. 3.29 handelt es sich um eine bereits korrigierte Darstellung des Blockschaltbildes.

4 Entwicklung

In diesem Kapitel werden der Entwicklungsansatz und die Architektur des PIPE IP Cores für den GateMate™ FPGA thematisiert. Für die Quellcodes der einzelnen Verilog-Module siehe Anhang A.4.

4.1 Entwicklungsansatz

Um den Einsatz des GateMate™ FPGA in KI-Anwendungen zu beschleunigen, stellt der erste Schritt für die Anbindung an eine PCI Express Kommunikation, die Entwicklung eines PIPE IP-Cores dar. Der IP-Core soll den integrierten SerDes um ein PIPE erweitern und so zusätzlich den Einsatz von PIPE-kompatiblen PCIe Soft Cores externer Hersteller ermöglichen (Abb. 4.1).

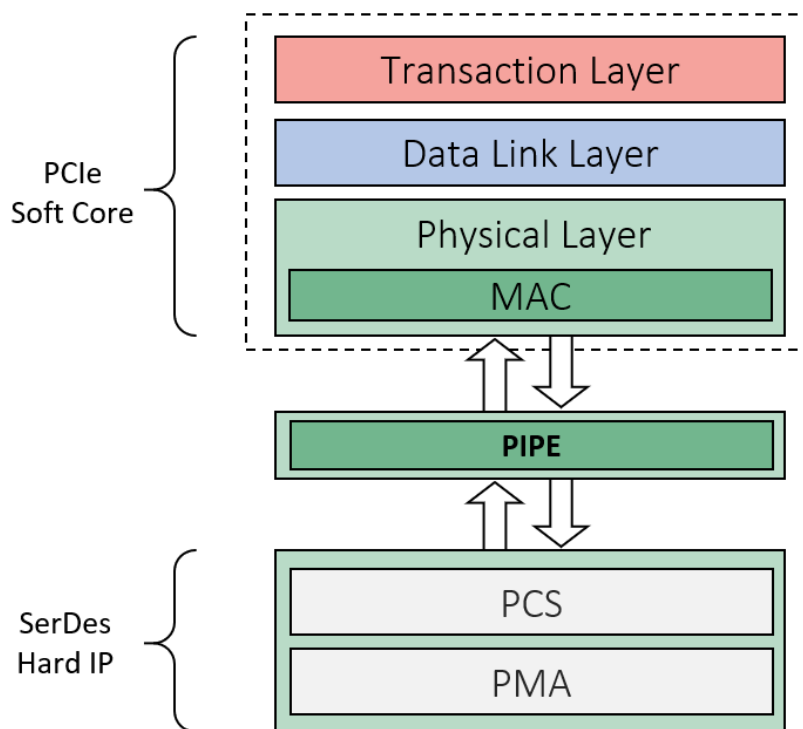


Abb. 4.1: Erweiterung des SerDes um einen PIPE IP-Core.

Die Anforderungen an den IP-Core beinhalten zum einen die Implementierung der notwendigen Schnittstellen des PIPE und zum anderen die Umsetzung der zu erwartenden Funktionalität (siehe Kap. 3.3). Daher dient der IP-Core als Bindeglied

zwischen den Schnittstellen des PIPE und des integrierten SerDes, wobei er deren Funktionalität durch die entsprechende Logik kombinieren muss. Der PIPE IP-Core kann für die Entwicklung eines vollständigen PCIe Soft Cores als Ausgangsbasis genutzt werden.

4.2 PIPE IP-Core

Das *Top Level Module* des IP-Cores (`ccfpga_pipe_if.v`) besteht aus der Instantiierung des SerDes und der erforderlichen PIPE Logik (Abb. 4.2). Es lässt sich als *Wrapper Block* verstehen, der außer der Instantiierung und der Verschaltung der Module, mit den Schnittstellen des IP-Cores sowie der Module untereinander über keine weitere Funktionslogik verfügt. Die Instantiierung des SerDes erfolgt durch ein entsprechendes *Instantiation Module* bzw. eine *Blackbox Instantiation*. Diese Beschreibung verdeutlicht, dass das instanziierte Modul lediglich die notwendigen Schnittstellen für den Zugriff durch die FPGA-Logik bereitstellt und in der Synthese durch die Verbindung mit den integrierten Schaltungsanteilen ersetzt wird.

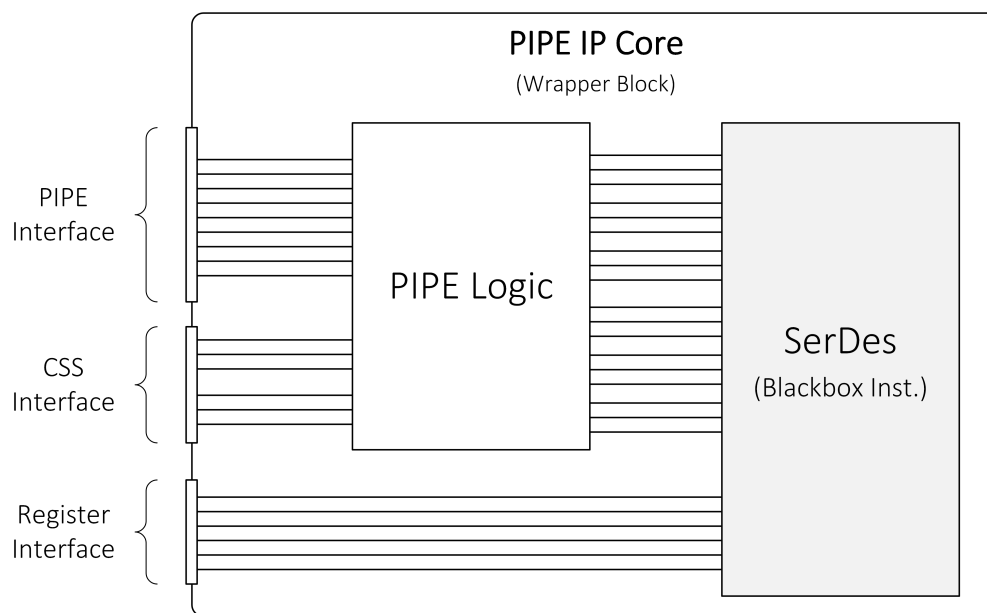


Abb. 4.2: Struktureller Aufbau des PIPE IP-Cores.

Innerhalb des *Top Level Modules* werden mit Ausnahme der Schnittstellen für den Registerzugriff alle Schnittstellen des SerDes mit der PIPE Logik verbunden. Die PIPE Logik implementiert die eigentliche Funktionslogik. Die Schnittstellen des IP-

Cores lassen sich in die Register-Schnittstellen (*Register Interface*), die Schnittstellen für das PIPE und in zusätzliche Schnittstellen zur Unterstützung des PIPE bzw. zur Ansteuerung des SerDes aufteilen (*CSS Interface*). Die zusätzlichen Signale werden in diesen Fall als *Custom Support Signals* (CSS) bezeichnet. Der PIPE IP-Core unterstützt einen Datenpfad mit einer konfigurierbaren Bitbreite von 8-, 16-, 32- oder 64-Bit. Die Bitbreite wird über den Parameter `DATA_BYTES` definiert, der die Anzahl der entsprechenden Bytes für den Datenpfad angibt. Aus diesem leitet sich auch der Parameter `DATA_WIDTH` für die spezifische Anpassung gewisser Ports oder Signale ab. Diese Art der Parametrisierung zieht sich durch den gesamten IP-Core und dessen Sub-Module. Generell werden alle Bitbreiten unterstützt, jedoch stehen die 16-Bit und die 64-Bit Version des IP-Cores im Vordergrund. Die 16-Bit Version ermöglicht bei einer realisierbaren Frequenz von 125 MHz, eine direkte Kompatibilität zur PIPE Spezifikation, wobei die 64-Bit Version sich einfacher auf die Schnittstellen des SerDes übertragen lässt und weniger zusätzliche Logik erfordert. Für die Konfiguration des SerDes enthält das *Top Level Modul*, ebenfalls den Parameter `CCAG_CFG_PARAM`, der im Rahmen der *Blackbox Instantiation* verwendet wird (Quellcode 4.1).

```

34 module ccfpga_pipe_if #(
35     parameter DATA_BYTES = 8,           // Configure width of PIPE datapath in
        Bytes
36     parameter DATA_WIDTH = DATA_BYTES*8,
37     parameter CCAG_CFG_PARAM = 1488'b0 // Parameter for Configuration
38 ) (

```

Quellcode 4.1: Parameter des PIPE IP-Core (*ccfpga_pipe_if.v*).

4.2.1 PIPE Logic

Die PIPE Logic (*ccfpga_pipe_logic.v*) bildet den Kern des PIPE IP-Cores und übernimmt auf Grundlage der PIPE Schnittstellen die Ansteuerung des SerDes. Dazu implementiert das Modul zunächst die erforderlichen Schnittstellen für das PIPE und erweitert diese um zusätzliche CSS (Quellcode 4.2).

```

38 module ccfpga_pipe_logic #(
39     parameter DATA_BYTES = 8,           // Width of PIPE Datapath in Bytes
40     parameter DATA_WIDTH = DATA_BYTES*8 // Do not set independet from Bytes
41 )
42 (
43     // ----- PIPE INTERFACE PORTS -----
44     // External
45     output wire o_PCLK,                 // PCLK (user side)
46
47     // Command

```

```

48     input wire                i_Reset,           // Async. Reset for Transceiver (Tx/Rx)
49     input wire [ 1:0]        i_PowerDown,      // Power States (P0 - P2)
50     input wire                i_TxDetectRx,    // Receiver Detection (P0) or Loopback (P1)
51     input wire                i_TxElecIdle,    // Tx Electrical Idle, Valid Data (P0) or Beacon (P2)
52     input wire [DATA_BYTES-1:0] i_TxCompliance, // Tx negative Disparity LSB (Compliance Pattern)
53     //input wire                i_TxSwing,      // Tx Voltage Swing Level [Optional by Spec]
54     input wire                i_RxPolarity,    // Rx Polarity Inversion
55
56     // Status
57     output wire               o_RxValid,       // Symbol Lock and Valid Data on RxData and RxDataK
58     output wire               o_PhyStatus,     // Status of several PHY functions (Transition)
59     output wire               o_RxElecIdle,    // Rx Detection of Electrical Idle
60     output wire [ 2:0]        o_RxStatus,     // Receiver Status and Received Data Status
61
62     // Transmit Data
63     input wire [DATA_WIDTH-1:0] i_TxData,      // Tx Data
64     input wire [DATA_BYTES-1:0] i_TxDataK,    // Tx K Data
65
66     // Receive Data
67     output wire [DATA_WIDTH-1:0] o_RxData,     // Rx Data
68     output wire [DATA_BYTES-1:0] o_RxDataK,    // Rx K Data
69
70     // ----- CSS INTERFACE PORTS -----
71
72     output wire               o_tx_buf_err,    // Tx Buffer Error
73     output wire               o_clk_core_rx_rec, // Rx Recovered Clock
74     input wire                 i_rx_buf_reset, // Rx Buffer Reset
75     output wire               o_rx_buf_err,    // Rx Buffer Error
76     output wire [3:0]         o_fsm_state_pipe, // State of PIPE FSM
77     output wire [1:0]         o_fsm_state_align, // State of Align FSM
78
79     output wire [DATA_BYTES-1:0] o_RxDataComma, // Rx Byte Comma Indication
80     output wire [DATA_BYTES-1:0] o_RxDataDispErr, // Rx Byte Disparity Error Indication
81     output wire [DATA_BYTES-1:0] o_RxDataDecErr, // Rx Byte Decode Error Indication

```

Quellcode 4.2: PIPE und CSS Schnittstellen in der PIPE Logic.

Die Schnittstellen, die in Zusammenhang mit der Bitbreite des Datenpfades stehen, werden über die Parameter `DATA_BYTES` und `DATA_WIDTH` angepasst. Die Schnittstelle `i_TxCompliance` weicht in ihrer Bitbreite von der PIPE Spezifikation ab. Diese leitet sich, wie etwa bei der Schnittstelle `i_TxDataK`, aus der Anzahl der Bytes im Datenpfad ab. Dadurch lassen sich die übertragenen Bytes individuell in ihrer Disparität (*Disparity*) beeinflussen. Soll die Schnittstelle wie in der PIPE Spezifikation verwendet werden (siehe Kap. 3.3.9), kann bei einem 16-Bit Datenpfad das MSB für das *Highest Byte* ([15:8]) konstant auf 0 gehalten werden. Die CSS umfassen verschiedene Schnittstellen, die direkt mit dem SerDes oder den Sub-Modulen der PIPE Logic in Verbindung stehen (Tabelle 4.1). So lassen sich über diese die *Buffer* für den Tx- und Rx-Datenpfad des SerDes ansteuern oder die *Recovered Clock* auslesen. Für das Reset-Signal des *Elastic Buffers* (`i_rx_buf_reset`) ist zu beachten, dass dieses *High Active* ist und während des normalen Betriebs auf 0 gehalten werden muss. Über die Schnittstellen für die FSMs, werden die derzeitigen Zustände für die beiden FSMs innerhalb der PIPE Logic ausgegeben. Des Weiteren umfassen die CSS drei Schnittstellen, welche zusätzliche Informationen zu den Empfangsdaten liefern.

Name	Typ	Active	Beschreibung
o_tx_buf_err	Output	High	Die Schnittstelle zeigt an ob ein <i>Overflow</i> oder <i>Underflow</i> im Tx-Buffer bzw. im <i>Phase Adjust FIFO</i> aufgetreten ist. Das Signal kann über einen Reset des Transmitters zurückgesetzt werden. Dies bedingt für das PIPE einen Reset des PHY. (SerDes Signal)
o_clk_core_rx_rec	Output	N/A	Über diese Schnittstelle wird die <i>Recovered Clock</i> ausgegeben, die durch die CDR aus den Empfangsdaten abgeleitet worden ist. Sie ist in ihrer Funktion identisch mit der SerDes Schnittstelle clk_core_rx_o. (SerDes Signal)
i_rx_buf_reset	Input	High	Die Schnittstelle löst einen Reset des <i>Elastic Buffer</i> aus. Sie ist identisch zur SerDes Schnittstelle rx_buf_reset_i. (SerDes Signal)
o_rx_buf_err	Output	High	Die Schnittstelle zeigt einen <i>Overflow</i> oder <i>Underflow</i> des <i>Elastic Buffer</i> an. Sie ist identisch zur SerDes Schnittstelle rx_buf_err_o. (SerDes Signal)
o_fsm_state_pipe[3:0]	Output	N/A	Gibt die Kodierung des derzeitigen Zustands der PIPE FSM aus.
o_fsm_state_align[1:0]	Output	N/A	Gibt die Kodierung des derzeitigen Zustands der FSM für das <i>Word Alignment</i> aus.
o_RxDataComma[7:0] o_RxDataComma[3:0] o_RxDataComma[1:0] o_RxDataComma	Output	N/A	Die Schnittstelle gibt an ob es sich bei dem korrespondierenden Byte um ein COM Symbol handelt. Die Bitbreite ist abhängig von der gewählten Bitbreite des Datenpfads (64-Bit, 32-Bit, 16-Bit, 8-Bit).

o_RxDataDispErr[7:0] o_RxDataDispErr[3:0] o_RxDataDispErr[1:0] o_RxDataDispErr	Output	N/A	Die Schnittstelle zeigt an, ob ein Fehler in der Disparität des Bytes vorliegt. Die Bitbreite ist abhängig von der Bitbreite des Datenpfads (64-Bit, 32-Bit, 16-Bit, 8-Bit).
o_RxDataDecErr[7:0] o_RxDataDecErr[3:0] o_RxDataDecErr[1:0] o_RxDataDecErr	Output	N/A	Die Schnittstelle wird genutzt, um einen 8b/10b Fehler in der Dekodierung der Empfangsdaten anzugeben. Die Bitbreite ist abhängig von der Bitbreite des Datenpfads (64-Bit, 32-Bit, 16-Bit, 8-Bit).

Tabelle 4.1: Schnittstellen für die *Custom Support Signals*.

Das Modul implementiert zudem die entsprechenden Ports für die Signale des SerDes. Diese sind namensgleich zu dessen Schnittstellen, wobei lediglich die Beschreibung für die Art des Ports (o_, i_) an den Anfang gesetzt worden ist und aus Sicht der PIPE Logic erfolgt (Quellcode 4.3).

```

84 // ----- CCAG SERDES PORTS -----
85 // info: port directions seen from PIPE interface side
86
87 // Clock
88 input wire      i_clk_core_pll,      // ADPLL
89 input wire      i_clk_core_rx_rec,   // Recovered Clock from Rx
90 output wire     o_clk_core_tx,      // Transmitter (Tx)
91 output wire     o_clk_core_rx,      // Receiver (Rx)
92
93 // Reset
94 output wire     o_pll_reset,        // ADPLL
95 output wire     o_tx_reset,        // Tx Reset
96 input wire      i_tx_reset_done,    // Tx Reset Status
97
156 output wire    o_rx_polarity,      // Rx Polarity Control
157
158 output wire    o_rx_en_ei_detector, // Rx Electrical Idle Detection Enable
159 input wire     i_rx_ei_en         // Rx Electrical Idle Detection Response
160 );

```

Quellcode 4.3: Auszug der SerDes Schnittstellen in der PIPE Logic.

Eine Ausnahme bildet die Schnittstelle `clk_core_rx_o`, die unter der Bezeichnung `i_clk_core_rx_rec` geführt wird, um Verwechslungen mit der Schnittstelle für den Takt des *Receivers* (`clk_core_rx_i`) zu vermeiden. Eine vollständige Darstellung der Ports findet sich im Quellcode der PIPE Logic in Anhang A.4.2. Für eine Übersicht der SerDes Schnittstellen und ihrer Funktion siehe [Rac20] oder Anhang A.2.

Konstante Portzuweisungen

Für die Umsetzung des PIPE, werden nicht alle Funktionen des SerDes benötigt oder müssen während des Betriebs angepasst werden. Daher werden einigen der Schnittstellen konstante Werte zugewiesen. Diese Schnittstellen umfassen die individuellen Resets der SerDes Architektur, die 8b/10b Kodierung, den PRBS Generator, das *Power Management* sowie verschiedene andere Funktionen des SerDes (Quellcode 4.4).

```
203 // Constant Port Value Assignments
204
205 assign o_tx_reset      = 1'b0;
206 assign o_tx_pcs_reset = 1'b0;
207 assign o_tx_pma_reset = 1'b0;
208
209 assign o_rx_reset      = 1'b0;
210 assign o_rx_pcs_reset = 1'b0;
211 assign o_rx_pma_reset = 1'b0;
212 assign o_rx_cdr_reset = 1'b0;
213 assign o_rx_eqa_reset = 1'b0;
214
215 assign o_tx_8b10b_bypass = 8'b0000_0000;
216 assign o_rx_8b10b_bypass = 8'b0000_0000;
217 assign o_tx_8b10b_en    = 1'b1;
218 assign o_rx_8b10b_en    = 1'b1;
219
220 assign o_tx_prbs_sel     = 3'b000;
221 assign o_tx_prbs_force_err = 1'b0;
222 assign o_rx_prbs_sel     = 3'b000;
223 assign o_rx_prbs_cnt_reset = 1'b0;
224
225 assign o_rx_slide       = 1'b0;
226 assign o_rx_comma_detect_en = 1'b1;
227
228 assign o_tx_polarity    = 1'b0;
229
230 assign o_tx_powerdown_n = 1'b1;
231 assign o_rx_powerdown_n = 1'b1;
232
233 assign o_rx_en_ei_detector = 1'b1;
234
235 assign o_tx_char_dispvall = 8'b0000_0000;
```

Quellcode 4.4: Konstante Portzuweisungen in der PIPE Logic

Struktur der Funktionslogik

Die Funktionslogik wird durch verschiedene Sub-Module realisiert oder ist teilweise direkt innerhalb der PIPE Logic implementiert. Es werden die folgenden Module instanziiert, wobei die letzten drei Module nur im Fall der 8-Bit, 16-Bit oder 32-Bit Version zum Einsatz kommen.

1. `ccfpga_pipe_fsm [fsm_inst_0]`:

Die PIPE FSM realisiert das *Power Management* des PIPE und verarbeitet auf

Grundlage der unterstützten *Power States* (P0, P1), die anliegenden Kontrollsignale. Diese Kontrollsignale umfassen die *Receiver Detection*, den *Electrical Idle* und den *Loopback*. Die FSM übernimmt die Generierung der Statussignale *o_PhyStatus* und *o_RxStatus*. Getaktet wird die FSM über die PCLK.

2. **ccfpga_pipe_fsm_word_align** [fsm_inst_1]

Die FSM führt die entsprechenden Schritte für das *Word Alignment* durch und generiert das Statussignal *o_RxValid*. Sie wird in Abhängigkeit zum *Power State* P0 durch die PIPE FSM aktiviert. Getaktet wird die FSM über die *Core Clock* des SerDes.

3. **ccfpga_pipe_clk_counter** [clk_count_inst_pipe_fsm]

Dieser Zähler wird durch die PIPE FSM aktiviert und über die steigenden Taktflanken der *Core Clock* angesteuert. Er bestimmt den zeitlichen Eintritt des *Transmitters* in den *Electrical Idle* Zustand und meldet das Erreichen des gewünschten Zählerstandes an die FSM über die Ausgangsschnittstelle *o_flag*.

4. **ccfpga_pipe_clk_counter** [clk_count_inst_align]

Dieser Zähler wird durch die FSM für das *Word Alignment* aktiviert und über die steigenden Takflanken der *Core Clock* angesteuert. Er wird genutzt, um die Zeit zwischen erfolgreichem *Word Alignment* und dem Auftreten valider Daten an den Empfangsschnittstellen (*o_RxData*, *o_RxDataK*) zu bestimmen. Das Erreichen des gewünschten Zählerstandes wird über die Ausgangsschnittstelle *o_flag* angezeigt.

5. **ccfpga_tx_demux** [demux_inst_0]

Das Modul wird dazu genutzt die Sendedaten des PIPE IP-Cores (*i_TxData*, *i_TxDataK*) auf die Schnittstellen des SerDes zu demultiplexen. Die Bitbreiten des PIPE sind in diesem Zusammenhang geringer als die konstanten Bitbreiten des SerDes. Daher werden die Daten in Abhängigkeit zur schnelleren PCLK, teilweise auf die Schnittstellen des SerDes übertragen. Damit die Daten bis zur Übernahme durch den SerDes bei steigenden Flanken der *Core Clock* bereit stehen, werden entsprechende Speicherelemente implementiert. Das Kontrollsignal *i_TxCompliance* zur Anpassung der Disparität wird dabei ebenfalls mit berücksichtigt.

6. `ccfpga_rx_mux` [`mux_inst_0`]

Das Modul dient dazu die Empfangsdaten des SerDes auf die Schnittstellen des PIPE (`o_RxData`, `o_RxDataK`) zu multiplexen. Die Bitbreiten des PIPE sind in diesem Zusammenhang geringer als die konstanten Bitbreiten des SerDes. Daher werden die Daten in Abhängigkeit zur schnelleren PCLK, teilweise von den SerDes Schnittstellen übernommen und über das PIPE ausgegeben. Die ausgegebenen Daten schließen auch die CSS `o_RxDataComma`, `o_RxDataDispErr` und `o_RxDataDecErr` mit ein. Das Modul erzeugt zudem das Statussignal `o_RxStatus`, welches von den Empfangsdaten abhängig ist.

7. `CC_PLL` [`i_cc_pll_0`]

Bei diesem Modul handelt es sich um ein *Instantiation Module*, welches dazu dient, eine der integrierten PLLs des GateMate™ zu instanziiieren. Diese zusätzliche PLL wird dazu genutzt, um aus der *Core Clock* der ADPLL die notwendige PCLK zu generieren. Innerhalb der PIPE Logic wird der notwendige Parameter (*String*) für die Konfiguration der PLL über die Bitbreite des Datenpfads (`DATA_BYTES`) bestimmt. Für einen 16-Bit breiten Datenpfad ergibt sich hierbei eine Frequenz von 125 MHz. Für die Simulation wird anstatt der PLL ein entsprechendes *Functional Module* eingebunden, welches nicht synthetisierbar ist und das Verhalten der PLL weitestgehend nachbildet.

Das Modul `ccfpga_pipe_clk_counter` wird insgesamt zwei mal instanziiert. Zum einen im Zusammenhang mit der PIPE FSM (`ccfpga_pipe_fsm`) für die Vorhersage des *Electrical Idle* Zustands und zum anderen im Zusammenhang mit der Generierung des Statussignals `o_RxValid` durch die FSM für das *Word Alignment* (`ccfpga_pipe_fsm_word_align`). In der PIPE Logic wird die abweichende Instanzierung der Module und die damit verbundene Verschaltung durch ein entsprechendes *Generate Statement* implementiert. Dieses ist abhängig vom Parameter `DATA_BYTES` und unterscheidet zwischen der 64-Bit und den restlichen Bit Versionen (8-Bit, 16-Bit, 32-Bit). Eine Übersicht über die angesprochenen Module und der internen Verschaltung, kann den beiden Blockschaltbildern für die 64-Bit und die 16-Bit Version entnommen werden (Abb. 4.3 und Abb. 4.4).

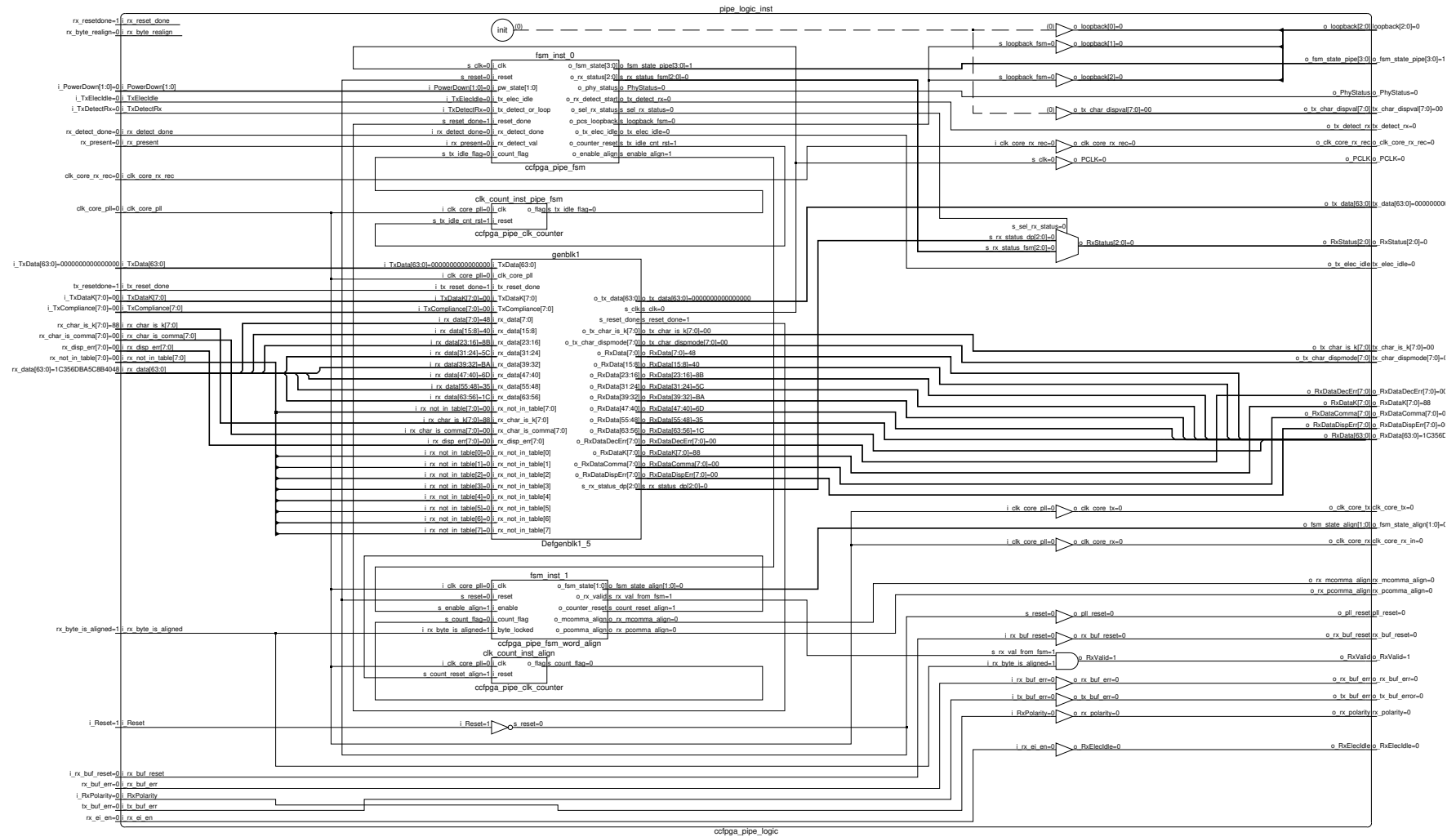


Abb. 4.3: Blockschaltbild der PIPE Logic ohne Darstellung aller konstanten Portzuweisungen (64-Bit Version)

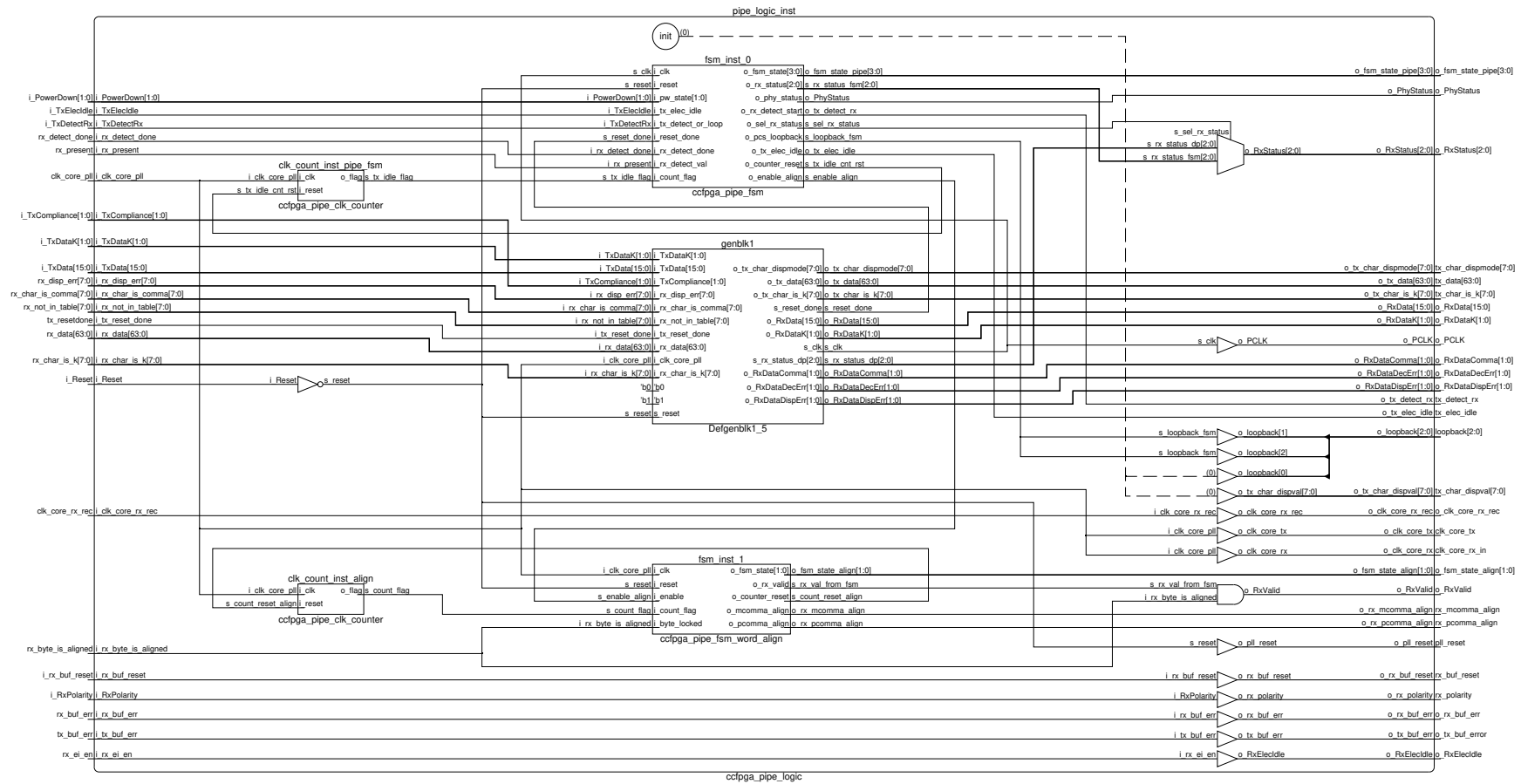


Abb. 4.4: Blockschaltbild der PIPE Logic ohne Darstellung aller konstanten Portzuweisungen (16-Bit Version)

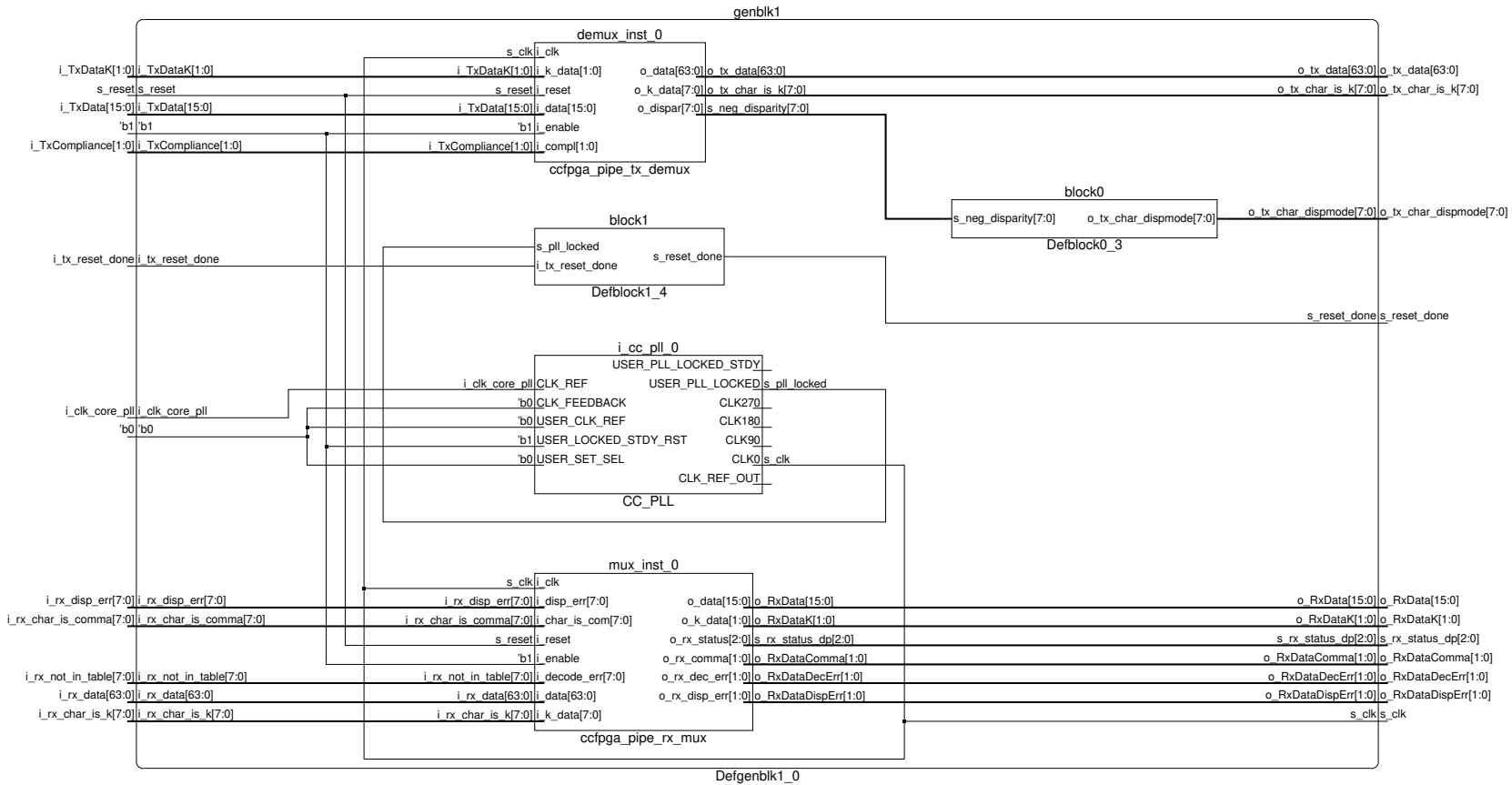


Abb. 4.5: Blockschaltbild des Tx-/Rx-Datenpfads mit PLL (16-Bit Version)

Implementierte Logik oder Module, die durch das *Generate Statement* erzeugt werden, sind im Blockschaltbild durch den Block *genblk1* dargestellt. Für die 16-Bit Version findet sich ein zusätzliches Blockschaltbild von *genblk1* in Abb. 4.5. In diesem wird die interne Verschaltung der Module darstellt.

Systemtakt und Reset

Die PIPE Logic implementiert die notwendige Verschaltung zwischen der ADPLL und den beiden Datenpfaden, damit der *Transmitter* und der *Receiver* durch die *Core Clock* mit 31,25 MHz getaktet werden können (Quellcode 4.5).

```

210 // Clock
211 assign o_PCLK           = s_clk;
212
213 assign o_clk_core_tx    = i_clk_core_pll;
214 assign o_clk_core_rx    = i_clk_core_pll;
215 assign o_clk_core_rx_rec = i_clk_core_rx_rec;

```

Quellcode 4.5: Verschaltung der Taktleitungen in der PIPE Logic

Die *Recovered Clock* des *Receivers* (*i_clk_core_rx_rec*) wird über die gleichnamige CSS (*o_clk_core_rx_rec*) ausgegeben. Die Ausgabe der PCLK erfolgt durch das Signal *s_clk*, welches je nach Anwendungsfall unterschiedlich zugewiesen wird. Die Sub-Module werden entweder direkt über die *Core Clock* oder die PCLK getaktet. Im Fall der 64-Bit Version, sind die *Core Clock* und die PCLK identisch, weshalb die *Core Clock* direkt dem Signal *s_clk* zugewiesen wird (Quellcode 4.6).

```

310 // Clock
311 assign s_clk = i_clk_core_pll;

```

Quellcode 4.6: Zuweisung des Signals *s_clk* (64-Bit Version)

Im Fall kleinerer Bitbreiten (8-Bit, 16-Bit, 32-Bit) wird für die PCLK eine deutlich höhere Frequenz von 250 Mhz, 125 MHz oder 64,5 MHz verlangt. Daher wird die Generierung der PCLK durch die zusätzlich implementierte PLL übernommen und das Signal *s_clk* mit dieser verschaltet (Quellcode 4.7).

```

386 CC_PLL #(
387     .CCAG_CFG_PARAM      ( PLL_PARAM      )
388 )
389 i_cc_pll_0 (
390     .CLK_REF              ( i_clk_core_pll ), // Refclk ADPLL
391     .CLK_FEEDBACK        ( 1'b0          ), // (const value)
392     .USER_CLK_REF        ( 1'b0          ), // (const value)
393     .USER_LOCKED_STDY_RST ( 1'b1        ), // Reset Locked state
394     .USER_SET_SEL        ( 1'b0          ), // (const value)

```

```

395     .USER_PLL_LOCKED_STDY (          ), // (float)
396     .USER_PLL_LOCKED    ( s_pll_locked ), // Locked state
397     .CLK270              (          ), // (float)
398     .CLK180              (          ), // (float)
399     .CLK90               (          ), // (float)
400     .CLK0                ( s_clk      ), // PCLK
401     .CLK_REF_OUT        (          ) // (float)
402 );

```

Quellcode 4.7: Instanziierung der zusätzlichen PLL des GateMate™

Die Durchführung eines Reset erfolgt über die Schnittstelle *i_Reset* des PIPE und zielt auf den Reset der ADPLL und der Sub-Module des PIPE IP-Cores ab. Der Reset des PIPE wird zunächst invertiert auf das Signal *s_reset* zugewiesen, da der Reset der ADPLL und der Sub-Module *High Active* ist (Quellcode 4.8).

```

215 // Reset Logic
216 assign s_reset = ~i_Reset;
217 assign o_pll_reset = s_reset;

```

Quellcode 4.8: Invertierung des Reset und Verschaltung mit der ADPLL.

Im Gegensatz zur PIPE Spezifikation muss der Reset des PIPE IP-Core nicht gehalten werden, bis die PCLK anliegt, sondern es reicht ein entsprechender Impuls des Reset Signals aus. Sobald ein Reset erfolgt ist, werden die Sub-Module sowie die PIPE FSM in ihren Reset Zustand versetzt. Durch den Reset der ADPLL wird aufgrund der internen Verschaltung mit den Datenpfaden der gesamte SerDes in den Reset Zustand überführt. Ab diesem Zeitpunkt geht der Resetvorgang von der ADPLL aus, die nach einem gewissen Zeitraum mit der Generierung der *Core Clock* beginnt und die beiden Datenpfade aus dem Reset nimmt. Die PIPE FSM befindet sich währenddessen in ihrem Reset Zustand und wartet darauf, dass der *Transmitter* durch das Signal *s_tx_reset_done* den Abschluss seines Resetvorgangs signalisiert (Quellcode 4.9).

```

316 // Reset Logic (excludes PLL)
317 //assign s_reset_done = i_rx_reset_done & i_tx_reset_done;
318 assign s_reset_done = i_tx_reset_done;

```

Quellcode 4.9: Zuweisung des Signals *s_reset_done* (64-Bit Version)

Im Fall der zusätzlichen PLL wird der Austritt der FSM aus dem Reset-Zustand auch von dessen Statussignal *s_pll_locked* abhängig gemacht (Quellcode 4.10).

```

361 // Reset Logic (includes PLL)
362 //assign s_reset_done = s_pll_locked & i_rx_reset_done & i_tx_reset_done;
363 assign s_reset_done = s_pll_locked & i_tx_reset_done;

```

 Quellcode 4.10: Zuweisung des Signals *s_reset_done* (16-Bit Version)

Die Reset-Bedingungen werden jeweils über das Signal *s_reset_done* zusammengefasst und mit der PIPE FSM verschaltet. Das ursprüngliche Design der PIPE Logic, sah auch den abgeschlossenen Reset des *Receivers* durch das Signal *s_tx_reset_done* vor, jedoch zeigte sich während der Simulation, dass dieser auch von der CDR abhängig ist und womöglich den Reset-Vorgang behindert (siehe Kap. 5.3).

Datenpfad in der 64-Bit Version

In der 64-Bit Version wird die Logik für den Tx- und Rx-Datenpfad innerhalb der PIPE Logic implementiert. Die Schnittstellen für die Sendedaten des PIPE *i_TxData* und *i_TxDataK* sowie des Kontrollsignals *i_TxCompliance* können aufgrund der identischen Bitbreiten direkt mit den Schnittstellen des SerDes verbunden werden (Quellcode 4.11).

```

319 // Tx Datapath
320 assign o_tx_data      = i_TxData;
321 assign o_tx_char_is_k = i_TxDataK;
322
323
324 // Disparity (Enable and Value)
325 assign o_tx_char_dispmode = i_TxCompliance;
  
```

Quellcode 4.11: Tx-Datenpfad in der PIPE Logic (64-Bit Version)

Auch die meisten Schnittstellen für die Empfangsdaten können durch die identischen Bitbreiten direkt mit den Schnittstellen des SerDes verbunden werden (Quellcode 4.12). Die PIPE Spezifikation fordert jedoch die Ausgabe des EDB Symbols (0xFE) im Fall eines Fehlers in der 8b/10b Kodierung. Daher erfolgt die Zuweisung jedes Bytes einzeln und wird von der Schnittstelle *i_rx_not_in_table* abhängig gemacht. Die Zuweisung des Statussignals *o_RxStatus* über das Signal *s_rx_status_dp* erfolgt in Abhängigkeit zu den aufgetretenen Fehlern in der Disparität (*i_rx_disp_err*) und den Fehlern in der 8b/10b Kodierung (*i_rx_not_in_table*).

```

328 // Rx Datapath
329 assign o_RxData [ 7: 0] = ( i_rx_not_in_table[0] ) ? 8'hFE : i_rx_data [ 7: 0]; // EDB Symbol (8'
    hFE)
330 assign o_RxData [15: 8] = ( i_rx_not_in_table[1] ) ? 8'hFE : i_rx_data [15: 8];
331 assign o_RxData [23:16] = ( i_rx_not_in_table[2] ) ? 8'hFE : i_rx_data [23:16];
332 assign o_RxData [31:24] = ( i_rx_not_in_table[3] ) ? 8'hFE : i_rx_data [31:24];
333 assign o_RxData [39:32] = ( i_rx_not_in_table[4] ) ? 8'hFE : i_rx_data [39:32];
334 assign o_RxData [47:40] = ( i_rx_not_in_table[5] ) ? 8'hFE : i_rx_data [47:40];
335 assign o_RxData [55:48] = ( i_rx_not_in_table[6] ) ? 8'hFE : i_rx_data [55:48];
  
```

```

336     assign o_RxData [63:56] = ( i_rx_not_in_table[7] ) ? 8'hFE : i_rx_data [63:56];
337
338     assign o_RxDataDecErr = i_rx_not_in_table;
339     assign o_RxDataK      = i_rx_char_is_k;
340     assign o_RxDataComma = i_rx_char_is_comma;
341     assign o_RxDataDispErr = i_rx_disp_err;
342
343     assign s_rx_status_dp = |i_rx_not_in_table ? 3'b100 : ( |i_rx_disp_err ? 3'b111 : 3'b000 );

```

Quellcode 4.12: Rx-Datenpfad in der PIPE Logic (64-Bit Version)

Dadurch wird bei der Übertragung der 64-Bit breiten Daten eine entsprechende Fehlermeldung über das Statussignal *o_RxStatus* ausgegeben, sobald bei einem der acht übertragenen Bytes ein Fehler vorliegt. Da vor allem bei einem Fehler in der Disparität keine direkte Angabe über das auslösende Byte getroffen werden kann, sollte in diesem Fall auf die Schnittstelle *o_RxDataDispErr* zurückgegriffen werden. Eine Anzeige für das Hinzufügen oder das Entfernen von SKP Symbolen im *Elastic Buffer*, wird durch das Statussignal *o_RxStatus* nicht unterstützt. Ebenfalls wird die Anzeige eines separaten *Overflow* oder *Underflow* mit Bezug auf die übertragenen Daten nicht unterstützt. Für diesen Fall kann die Schnittstelle *o_rx_buf_err* genutzt werden.

4.2.2 PIPE FSM

Bei der PIPE FSM handelt es sich um einen Moore-Automaten, der über die Kontrollsignale *i_PowerDown*, *i_TxElecIdle* und *i_TxDetectRx* des PIPE angesteuert wird. Die FSM implementiert die *Power States* sowie die damit in Verbindung stehenden Funktionen des PHY. Im Rahmen des *Power Management* werden zum gegenwärtigen Zeitpunkt nur die *Power States* P0 und P1 unterstützt, da diese essentiell für die Umsetzung der wichtigsten Zustände in der LTTSM sind (siehe Kap. 3.3.3). Diese *Power States* beeinflussen zudem die Funktion des Kontrollsignals *i_TxDetectRx*. Die Umsetzung der *Power States* P0s und P2 wird für zukünftige Erweiterungen des IP-Cores angestrebt, wobei deren Implementierung noch genauer geprüft werden muss. Der SerDes verfügt für das eigene *Power Management* lediglich über die beiden Schnittstellen *o_tx_powerdown_n* und *o_rx_powerdown_n*. Durch diese ist es möglich, den *Transmitter* und *Receiver* auszuschalten. Diese Schnittstellen werden in der PIPE Logic nicht genutzt und konstant auf logisch 1 gehalten, aber könnten zur Implementierung des *Power States* P2 eingesetzt werden. In diesem Zusammenhang müsste auch die Implementierung eines *WAKE*-Signals geprüft werden, da der SerDes nicht über die Funktionslogik zur Übertragung eines *Beacon* verfügt [PCI09, Kap. 4.3.5.8.].

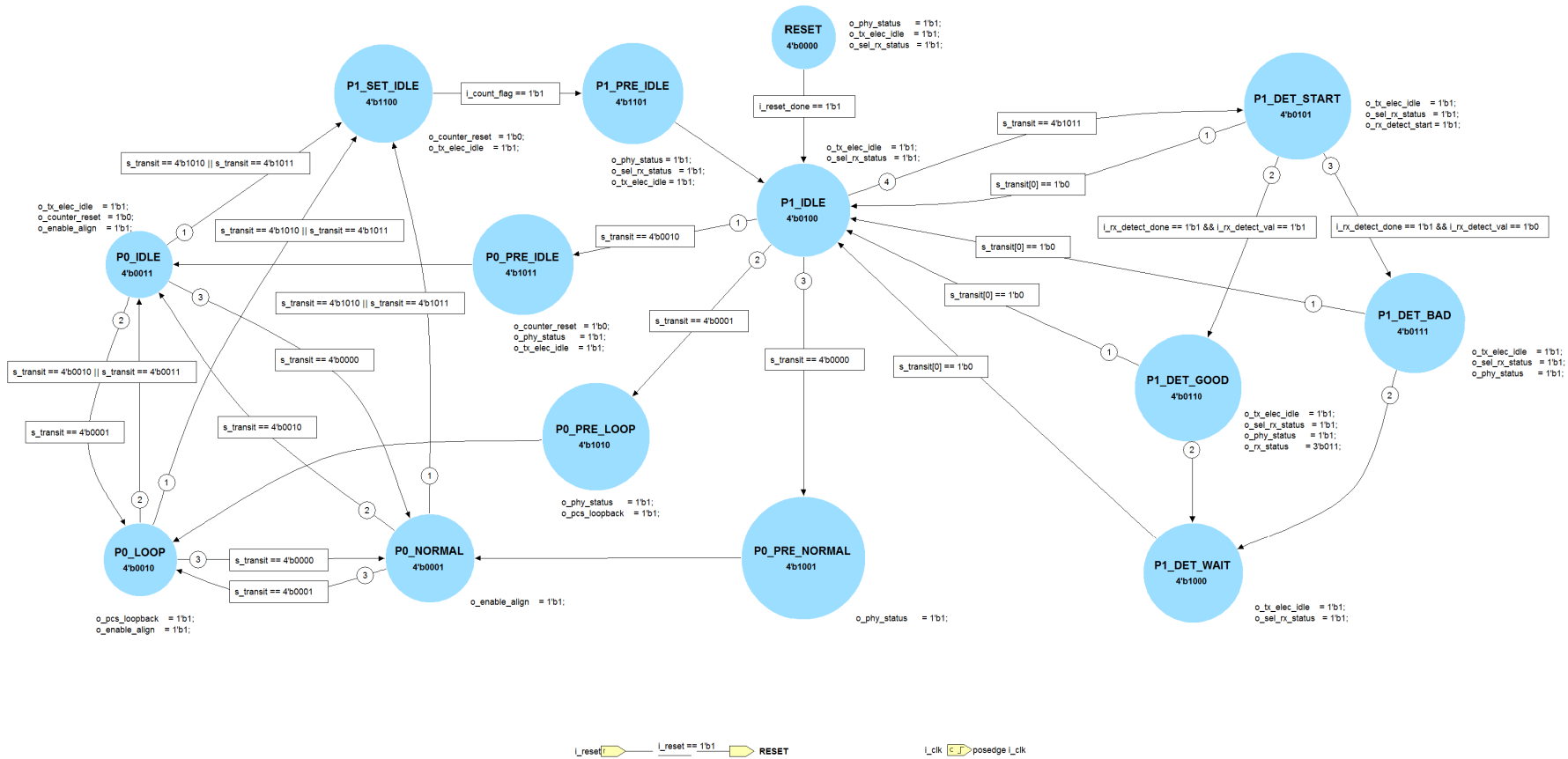


Abb. 4.6: Grafische Darstellung der PIPE FSM.

Der SerDes verfügt selbst über keine expliziten *Power States*, weshalb der *Power State* P1 zwar nach außen hin durch die FSM angezeigt wird, aber intern keinen Einfluss auf den Zustand des SerDes hat. Der Ausgangszustand der PIPE FSM ist der Zustand RESET, in den die FSM bei einem Reset am Eingangsport *i_reset* überführt wird (Abb. 4.6). In diesem Zustand verbleibt die FSM bis ihr über den Eingangsport *i_reset_done* mitgeteilt wird, dass die entsprechenden Reset-Bedingungen eingetreten sind. Beim Eintritt in den Folgezustand P1_IDLE wird der *PhyStatus* über den Ausgangsport *o_phy_status* auf 0 gezogen und signalisiert das Ende des Resets. Alle anderen Ausgangssignale sind identisch zum Zustand RESET. Die Übergänge in andere FSM Zustände sind von den Eingangsports *i_pw_state*, *i_tx_elec_idle* und *i_tx_detect_or_loop* abhängig. Diese Ports sind in der PIPE Logic mit den korrespondierenden Schnittstellen des PIPE verbunden. Innerhalb der FSM werden die Ports unter dem Signal *s_transit* zusammengefasst, um das Überführungsschaltnetz der FSM übersichtlicher zu gestalten (Quellcode 4.13).

```
75 assign s_transit = {i_pw_state, i_tx_elec_idle, i_tx_detect_or_loop};
```

Quellcode 4.13: Zuweisung des Signals *s_transit* in der FSM.

Der derzeitige Zustand der FSM wird über den Port *o_fsm_state* ausgegeben und ist mit der Schnittstelle *o_fsm_state_pipe* des IP-Cores verbunden. Der Zustand P1_IDLE stellt den normalen *Power State* P1 dar und hält den PHY über den Ausgangsport *o_tx_elec_idle* im *Electrical Idle* Zustand. Zur Durchführung der *Receiver Detection*, wird das Kontrollsignal *i_TxDetectRx* des PIPE im *Power State* P1 auf logisch 1 gezogen und führt zum Übergang in den Zustand P1_DET_START. In diesem Zustand wird die *Receiver Detection* des SerDes über den Ausgangsport *o_rx_detect_start* aktiviert. Sobald deren Abschluss durch den Eingangsport *i_rx_detect_done* signalisiert worden ist, erfolgt auf Basis des Werts am Eingangsports *i_rx_detect_val* der Übergang in den Zustand P1_DET_GOOD oder P1_DET_BAD. In diesen beiden Zuständen wird der *PhyStatus* auf logisch 1 gezogen und zusammen mit der entsprechenden Kodierung von *RxStatus* (000 oder 011) übertragen. Beim Eintritt in den Folgezustand P1_DET_WAIT werden die Ausgangssignale zurückgesetzt und die FSM verbleibt so lange in diesem Zustand, bis das Kontrollsignal *i_TxDetectRx* wieder auf 0 gezogen worden ist. Wird das Kontrollsignal vorzeitig während der *Receiver Detection* wieder auf 0 gezogen, erfolgt der direkte Übergang zurück in P1_IDLE. Damit das Statussignal *RxStatus* durch die PIPE FSM ausgegeben werden kann und nicht mit dem Rx-Datenpfad kollidiert, ist in der PIPE Logic ein Multiplexer für die Zuweisung der Schnittstelle

o_RxStatus implementiert worden (Quellcode 4.14).

```
218 // RxStatus Mux  
219 assign o_RxStatus = s_sel_rx_status ? s_rx_status_fsm : s_rx_status_dp;
```

Quellcode 4.14: Multiplexer für die Zuweisung von *o_RxStatus*.

Befindet sich die FSM in einem der Zustände des *Power State* P1, wird die Ausgabe von *o_RxStatus* auf das Signal der FSM (*s_rx_status_fsm*) umgestellt. Die Ansteuerung des Multiplexers erfolgt über den Ausgangsport *o_sel_rx_status* in der FSM. Der *Power State* P0 wird in der FSM durch die Zustände P0_IDLE, P0_LOOP und P0_NORMAL realisiert. Da es sich bei der FSM um einen Moore-Automaten handelt, implementieren die einzelnen Zustände die verschiedenen Funktionen innerhalb des *PowerState* P0. Der Übergang in die jeweiligen Zustände wird über die entsprechenden Kontrollsignale des PIPE ausgelöst. Für jeden der drei Zustände existiert jeweils ein Vorzustand PRE_*, der für einen Übergang aus dem *Power State* P1 bzw. dem Zustand P1_IDLE zwischen geschaltet ist. Die Ausgabe dieser Zustände, ist identisch mit den eigentlichen Zuständen und dient lediglich zur Generierung des Statussignals *PhyStatus* über Ausgangsport *o_phy_status*. Da die FSM mit der PCLK getaktet wird, bleibt das Statussignal für genau eine Taktperiode der PCLK aktiv. Die Übergänge zwischen den verschiedenen Zuständen des *Power State* P0 erfordern keine Generierung des *PhyStatus*, da kein Wechsel des *Power States* vorliegt. Der Übergang aus dem *Power State* P0 in den *Power State* P1 führt immer über den Zustand P1_SET_IDLE. In diesem Zustand wird der externe Zähler zur Bestimmung des *Electrical Idle* aus dem Reset genommen (*o_counter_reset*) und beginnt damit als Referenz die Takte der *Core Clock* zu zählen. Hat der Zähler den finalen Zählwert erreicht, signalisiert er dies der FSM (*i_count_flag*) und löst den Übergang in den Zustand P1_PRE_IDLE aus. Dieser Zustand dient wie die anderen Vorzustände zu Generierung des Statussignals *PhyStatus* und erzeugt die selben Ausgaben wie der Folgezustand P1_IDLE. Die vorherige Bestimmung des *Electrical Idle* Zustands für den Übergang in den *Power State* P1 ergibt sich aus den Beispieloperation der PIPE Spezifikation (siehe Kap. 3.3.3).

4.2.3 Word Alignment

Das *Word Alignment* sowie die Zuweisung des Statussignals *o_RxValid* wird innerhalb der PIPE Logic durch eine FSM realisiert. Bei dieser handelt es sich wie bei

der PIPE FSM um einen Moore-Automaten (Abb. 4.7). Der Zustand WAIT dient hierbei als Reset-Zustand. In diesem Zustand verbleibt die FSM, bis sie durch die PIPE FSM aktiviert wird (i_enable) und durch den SerDes kein vorhandener *Byte Lock* (i_byte_locked) signalisiert wird. Die FSM wird nur während des *Power State* P0 aktiv gehalten, so dass beim Verlust des *Byte Locks* im *Power State* P1 kein unnötiges *Word Alignment* durchgeführt wird. Das *Word Alignment* findet erst nach Rückkehr in den *Power State* P0 statt, welcher automatisch durch die MAC veranlasst wird. Beim Eintritt in den Zustand ALIGN_START wird über die beiden Ausgangsports o_mcomma_align und o_pcomma_align das *Word Alignment* des SerDes aktiviert. Ist ein entsprechender *Byte Lock* erfolgt (i_byte_locked), wechselt die FSM in den Zustand COUNT_START und nimmt den zugehörigen Zähler aus dem Reset ($o_counter_reset$). Der Zähler ist durch die *Core Clock* getaktet und bestimmt den Zeitpunkt zu dem die validen Empfangsdaten an den Schnittstellen des SerDes anliegen.

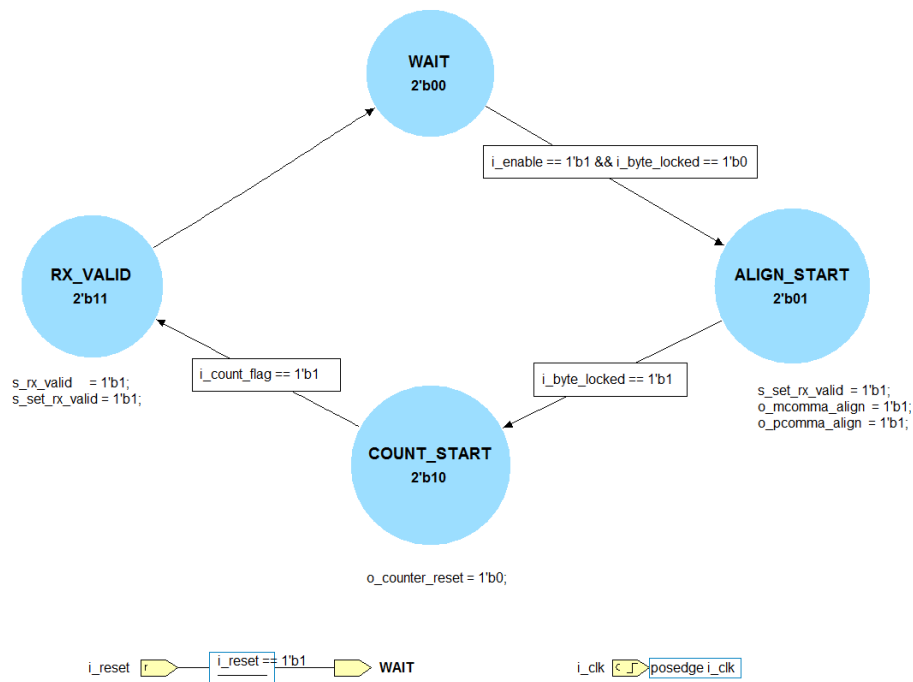


Abb. 4.7: Grafische Darstellung der FSM für das *Word Alignment*.

Der Übergang in den Zustand RX_VALID erfolgt, nachdem der Zähler das Erreichen seines Zielwertes signalisiert hat (i_count_flag). In diesem Zustand wird der Wert für das Statussignal $o_RxValid$ gesetzt und in einem 1-Bit breiten Speicher-element abgelegt (Quellcode 4.15).

```

118 // DFF Rx Valid
119 always@(posedge i_clk, posedge i_reset) begin
120     if (i_reset == 1'b1)
121         o_rx_valid <= 1'b0;
122     else
123         if ( s_set_rx_valid == 1'b1 )
124             o_rx_valid <= s_rx_valid;
125     end

```

Quellcode 4.15: Speicherung des Wertes für *o_RxValid*

Die Ansteuerung erfolgt über die Signale *s_rx_valid* und *s_sel_rx_valid*. Der Wert wird im Fall eines Reset oder bei Eintritt in den Zustand ALIGN_START zurückgesetzt. Tritt während des normalen Betriebs im *Power State P0* der Verlust des *Word Alignments* ein, wird das Statussignal *o_RxStatus* sofort über die Verschaltung in der PIPE Logic auf 0 gezogen (Quellcode 4.16).

```

228 // Rx Valid
229 assign o_RxValid = s_rx_val_from_fsm & i_rx_byte_is_aligned;

```

Quellcode 4.16: Multiplexer für die Zuweisung von *o_RxValid*.

Liegt ein entsprechender *Byte Lock* durch den SerDes vor (*i_rx_byte_is_aligned*) und die FSM zeigt den Ablauf der *Receive Latency* an (*s_rx_val_from_fsm*), wechselt das Statussignal *o_RxValid* auf den logischen Wert 1.

4.2.4 Tx Demultiplexer

Das Modul *ccfpga_pipe_tx_demux* wird dazu genutzt, die Sendedaten des PIPE und dessen Kontrollsignal *i_TxCompliance* zur Anpassung der Disparität auf die Schnittstellen des SerDes zu demultiplexen. Dazu verfügt das Modul über die notwendigen Schnittstellen für das PIPE und den SerDes (Quellcode 4.17).

```

29 module ccfpga_pipe_tx_demux #(
30     parameter DATA_BYTES = 2, // Set to configure width of used datapath (PIPE)
31     parameter DATA_WIDTH = DATA_BYTES*8 // Used to configure bitwidth of datapath
32 )
33 (
34     input wire i_clk,
35     input wire i_reset,
36     input wire i_enable,
37     input wire [DATA_BYTES-1:0] i_k_data, // TxK Data (PIPE)
38     input wire [DATA_WIDTH-1:0] i_data, // Tx Data (PIPE)
39     input wire [DATA_BYTES-1:0] i_compl, // Tx Compliance (PIPE)
40     output reg [63:0] o_data, // Tx Data (SerDes)
41     output reg [ 7:0] o_k_data, // TxK Data (SerDes)
42     output reg [ 7:0] o_dispar // Tx Disparity (SerDes)
43 );

```

Quellcode 4.17: Schnittstellen des Moduls *ccfpga_pipe_tx_demux*

Die PIPE Schnittstellen des Moduls werden über die Parameter `DATA_BYTES` und `DATA_WIDTH` angepasst und variieren je nach der gewählten Bitbreite des Datenpfads. Die Funktion des Moduls besteht darin, die Daten des PIPE in Abhängigkeit zur PCLK (*i_clk*) auf die wechselnden Positionen der breiteren SerDes Schnittstellen zu übertragen. Die Daten müssen bis zur Übernahme durch den SerDes zur steigenden Taktflanke der *Core Clock* gespeichert werden, weshalb die Ausgangs-ports als *reg* definiert werden. Zur Umsetzung der Funktion werden zunächst die lokalen Parameter `MSB_INIT` und `MSB_INIT_K` sowie die beiden Zählvariablen *s_msb* und *s_msb_k* definiert (Quellcode 4.18).

```

45     localparam MSB_INIT   = DATA_WIDTH-1;    // Used to configure the msb init position
46     localparam MSB_INIT_K = DATA_BYTES-1;    // Used to configure the msb k init position
47
48
49     reg [ 5:0] s_msb;                          // Used for partial vector selection of SerDes Port
50     reg [ 2:0] s_msb_k;                        // Used for partial vector selection of SerDes Port

```

Quellcode 4.18: Lokale Parameter und Variablen in *ccfpga_pipe_tx_demux*

Innerhalb des Moduls wird die Zuweisung der Daten, durch zwei *Always*-Blöcke implementiert, welche im Fall eines Reset (*i_reset*) die Ausgangs-ports für den SerDes auf Null setzen und den beiden Zählvariablen ihre Initialisierungswerte zuweisen (Quellcode 4.19).

```

53     // clocked demux for data
54     always@ (posedge i_clk, posedge i_reset) begin
55         if(i_reset == 1'b1) begin
56             o_data <= {64{1'b0}};
57             s_msb <= MSB_INIT;
58         end
59         else begin
60             if (i_enable == 1'b1) begin
61                 o_data[s_msb -: DATA_WIDTH] <= i_data;
62                 if (s_msb == 6'd63)
63                     s_msb <= MSB_INIT;
64                 else
65                     s_msb <= s_msb + DATA_WIDTH;
66             end
67         end
68     end
69
70     // clocked demux for k data and disparity (compliance)
71     always@ (posedge i_clk, posedge i_reset) begin
72         if(i_reset == 1'b1) begin
73             o_dispar <= {8{1'b0}};
74             o_k_data <= {8{1'b0}};
75             s_msb_k <= MSB_INIT_K;
76         end
77         else begin
78             if (i_enable == 1'b1) begin
79                 o_k_data[s_msb_k -: DATA_BYTES] <= i_k_data;

```

```

80         o_dispar[s_msb_k -: DATA_BYTES] <= i_compl;
81
82         if (s_msb_k == 3'd7)
83             s_msb_k <= MSB_INIT_K;
84         else
85             s_msb_k <= s_msb_k + DATA_BYTES;
86         end
87     end
88 end

```

Quellcode 4.19: *Always*-Blöcke im Modul *ccfpga_pipe_tx_demux*

Als Initialisierungswerte dienen die zuvor definierten Parameter `MSB_INIT` und `MSB_INIT_K`. Wird das Modul aus dem Reset genommen, beginnt es die Daten des PIPE bei steigenden Taktflanken auf einen Teilvektor der Ausgangsports des SerDes zuzuweisen. Der ausgewählte Teilvektor wird dabei durch die Zählvariablen (MSB) und die korrespondierenden Parameter `DATA_WIDTH` oder `DATA_BYTES` bestimmt. Anschließend werden die Zählvariablen um die erforderliche Datenbreite aufaddiert, so dass diese das MSB des nächsten Teilvektors darstellen. Sind alle Teilvektoren durchlaufen worden, erreichen die Zählvariablen ihren Endwert (6'd63, 3'd7) und werden auf ihren Initialisierungswert zurück gesetzt. Der Initialisierungswert entspricht dem MSB des ersten Teilvektors. Die integrierte *Enable*-Schnittstelle wird innerhalb der PIPE Logic nicht explizit angesteuert und konstant aktiv gehalten.

4.2.5 Rx Multiplexer

Das Modul *ccfpga_pipe_rx_mux* wird genutzt, um die Empfangsdaten des SerDes auf die Schnittstellen des PIPE zu multiplexen. Dies schließt auch die zusätzlichen Statussignale (CSS) und die Fehlererkennung des SerDes mit ein. Die Generierung des Statussignals *o_RxStatus* und die Implementierung der EDB Symbole im Fall eines 8b/10b Fehlers sind ebenfalls Teil des Moduls. Für die Daten des PIPE und des SerDes besitzt das Modul die folgenden Schnittstellen (Quellcode 4.20).

```

30 module ccfpga_pipe_rx_mux #(
31     parameter DATA_BYTES = 2,                // Set to configure width of used pipe datapath
32     parameter DATA_WIDTH = DATA_BYTES*8    // Used to configure bitwidth of datapath
33 )
34 (
35     input wire          i_clk,
36     input wire          i_reset,
37     input wire          i_enable,
38
39     input wire          [ 7:0 ] i_disp_err,    // Disparity Error (SerDes)
40     input wire          [ 7:0 ] i_char_is_com, // Byte is COM symbol (SerDes)
41     input wire          [ 7:0 ] i_decode_err,  // Decoding Error 8b/10b (SerDes)
42     input wire          [63:0] i_data,        // Rx Data (SerDes)
43     input wire          [ 7:0 ] i_k_data,    // Rk Data (SerDes)
44
45     output wire [DATA_WIDTH-1:0] o_data,    // Rx Data (PIPE)

```

```

46  output reg  [DATA_BYTES-1:0] o_k_data,      // Rk Data (PIPE)
47  output reg  [2:0] o_rx_status,           // Rx Status Signal (PIPE)
48  output reg  [DATA_BYTES-1:0] o_rx_comma,   // Rx Comma Signal (CSS)
49  output wire [DATA_BYTES-1:0] o_rx_dec_err, // Rx Decode Error (CSS)
50  output wire [DATA_BYTES-1:0] o_rx_disp_err // Rx Disparity Error (CSS)
51  );

```

Quellcode 4.20: Schnittstellen des Moduls *ccfpga_pipe_rx_mux*

Die Schnittstellen des PIPE sowie die zugehörigen CSS werden durch die Parameter `DATA_BYTES` und `DATA_WIDTH` in ihrer Bitbreite angepasst. Ebenfalls werden wie schon im Modul *ccfpga_pipe_tx_demux* (Kap. 4.2.4) die lokalen Parameter `MSB_INIT` und `MSB_INIT_K` als auch die Zählvariablen `s_msb` und `s_msb_k` eingesetzt (Quellcode 4.21).

```

54  localparam MSB_INIT  = DATA_WIDTH-1;      // Used to configure the msb data init position
55  localparam MSB_INIT_K = DATA_BYTES-1;     // Used to configure the msb k data init position
56
57  reg [DATA_WIDTH-1:0] s_data;
58  reg [DATA_BYTES-1:0] s_dec_err;
59  reg [DATA_BYTES-1:0] s_disp_err;
60  wire                s_dec_err_flag;
61  wire                s_disp_err_flag;
62  reg [5:0] s_msb;      // Used for partial vector selection of SerDes Port
63  reg [2:0] s_msb_k;   // Used for partial vector selection of SerDes Port

```

Quellcode 4.21: Lokale Parameter und Signale in *ccfpga_pipe_rx_mux*

Die Parameter dienen auch in diesem Fall als Initialisierungswerte und die Zählvariablen geben die Position des MSB für die Auswahl des Teilvektors an. Zusätzlich werden noch einige interne Signale definiert, die zum Teil von der gewählten Bitbreite des Datenpfads abhängig sind. Die Zuweisung der Daten erfolgt innerhalb der folgenden *Always*-Blöcke, wobei einige der Daten zunächst auf die entsprechenden Signale geschrieben werden (Quellcode 4.22).

```

72  // clocked mux for data
73  always@(posedge i_clk, posedge i_reset) begin
74      if(i_reset == 1'b1) begin
75          s_data <= {DATA_WIDTH{1'b0}};
76          s_msb <= MSB_INIT;
77      end
78      else begin
79          if (i_enable == 1'b1) begin
80              s_data = i_data [s_msb -: DATA_WIDTH];
81              if (s_msb == 6'd63)
82                  s_msb <= MSB_INIT;
83              else
84                  s_msb <= s_msb + DATA_WIDTH;
85          end
86      end
87  end
88
89  // clocked mux for k data, comma, error signals
90  always@ (posedge i_clk, posedge i_reset) begin
91      if(i_reset == 1'b1) begin
92          o_k_data  = {DATA_BYTES{1'b0}};
93          o_rx_comma = {DATA_BYTES{1'b0}};

```

```

94     s_dec_err  = {DATA_BYTES{1'b0}};
95     s_disp_err = {DATA_BYTES{1'b0}};
96     s_msb_k   = MSB_INIT_K;
97     end
98   else begin
99     if (i_enable == 1'b1) begin
100       o_k_data   = i_k_data [s_msb_k -: DATA_BYTES];
101       o_rx_comma = i_char_is_com [s_msb_k -: DATA_BYTES];
102       s_dec_err  = i_decode_err [s_msb_k -: DATA_BYTES];
103       s_disp_err = i_disp_err [s_msb_k -: DATA_BYTES];
104
105       if (s_msb_k == 3'd7)
106         s_msb_k <= MSB_INIT_K;
107       else
108         s_msb_k <= s_msb_k + DATA_BYTES;
109     end
110   end
111 end

```

Quellcode 4.22: verab Empfangsdaten in *ccfpga_pipe_rx_mux*

Für die Generierung des Statussignals *o_RxStatus* wird dessen Zuweisung in einem *Always*-Block von den beiden Signalen *s_dec_err_flag* und *s_disp_err_flag* abhängig gemacht. Die beiden Signale ergeben sich jeweils durch eine bitweise *OR*-Verschaltung der einzelnen Signale *s_dec_err* und *s_disp_err*. Diese Signale zeigen die Fehler in der Kodierung und der Disparität für die derzeit übertragenen Bytes an. Sollte bei einem dieser Bytes ein Fehler vorliegen, muss das Statussignal *o_RxStatus* entsprechend angepasst werden. Die Zuweisung durch die *If*-Bedingung implementiert zudem, die durch die PIPE Spezifikation geforderte Priorisierung der Fehlerfälle (Quellcode 4.23).

```

66     assign s_dec_err_flag = |s_dec_err;
67     assign s_disp_err_flag = |s_disp_err;

```

```

113 // logic for setting rx status
114 always@* begin
115   if (s_dec_err_flag == 1'b1) // 8b/10b error
116     o_rx_status = 3'b100;
117   else if (s_disp_err_flag == 1'b1) // Disparity error
118     o_rx_status = 3'b111;
119   else // Data OK (rst value)
120     o_rx_status = 3'b000;
121 end

```

Quellcode 4.23: Priorisierte Zuweisung des Statussignals *o_RxStatus*

Die Implementierung des EDB Symbols (0xFE) erfolgt identisch zur Implementierung im Fall eines 64-Bit breiten Datenpfads (siehe Kap. 4.2.1). Allerdings werden die bedingten Zuweisungen durch eine parametrisierte Schleife innerhalb eines *Generate*-Blocks erzeugt. Denn die Anzahl und Art der Zuweisungen ist von der gewählten Bitbreite des Datenpfads abhängig (Quellcode 4.24).

```
126 generate
127     for (i = 0; i < DATA_BYTES ; i = i + 1) begin : gen_block_rx_mux
128         localparam integer j = i*8 + 7;
129         assign o_data [ j -: 8] = ( s_dec_err[i] == 1'b1 ) ? 8'hFE : s_data [ j -: 8];
130     end
131 endgenerate
```

Quellcode 4.24: *Generate*-Block zur iterativen Erzeugung der Zuweisungen

5 Test & Verifikation

In diesem Kapitel werden die Struktur der eingesetzten *Testbench* sowie die durchgeführten Testfälle und Simulationsergebnisse für den PIPE IP-Core erläutert. Für die Quellcodes der *Testbench* und des *Testcase* (PIPE) siehe Anhang A.5.1 und A.5.2.

Anmerkung: Zum gegenwärtigen Zeitpunkt beinhaltet der Test nur die Verifikation des PIPE IP-Cores in der Version mit 64-Bit breiten Datenpfad. Eine Anpassung des *Testcase* für andere Versionen ist jedoch möglich.

5.1 Testbench

Die *Testbench* von RacyICs dient als Ausgangsbasis zur Verifikation des IP-Core und ist dahingehend um die PIPE Logic erweitert worden (Abb 5.1).

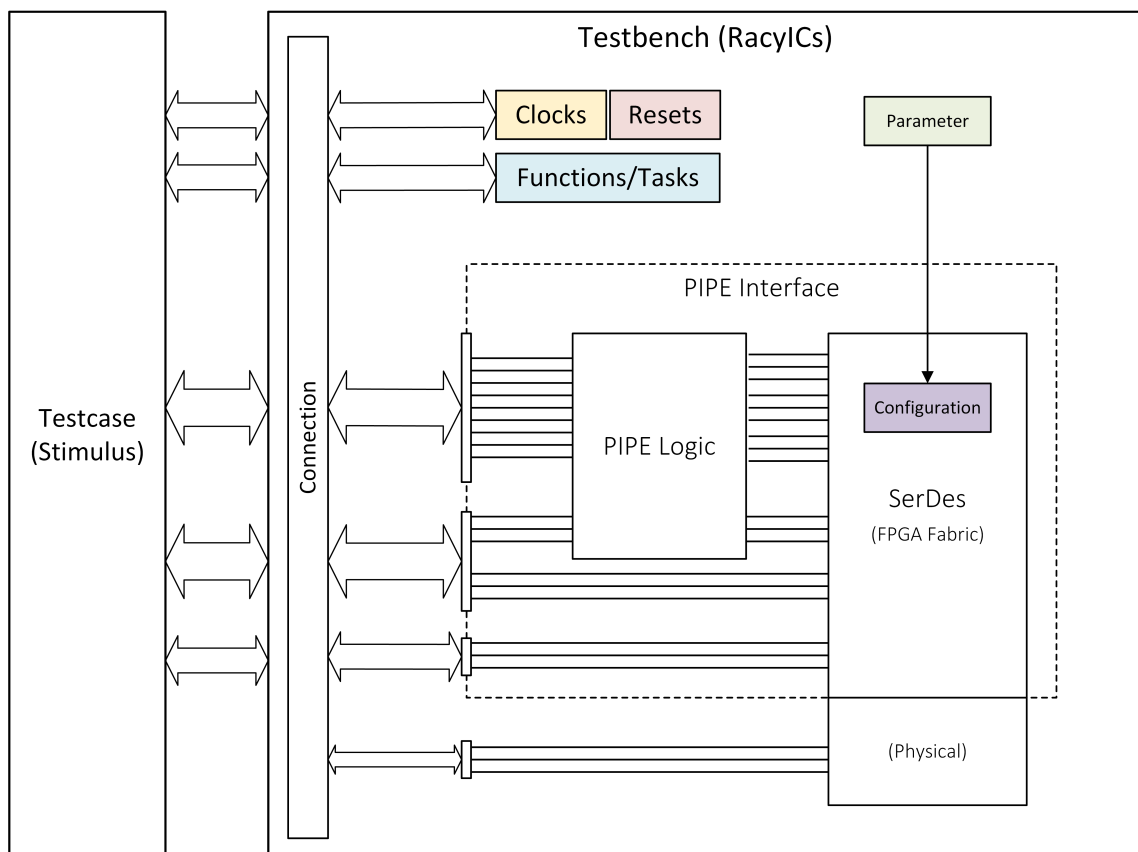


Abb. 5.1: Blockschaubild der durch die PIPE Logic erweiterten *Testbench*.

In ihrer Kernstruktur beinhaltet die *Testbench* den vollständigen SerDes, welcher sowohl die Schnittstellen von der Anwenderseite bzw. der *FPGAs Fabric* als auch die hart verdrahteten Schnittstellen innerhalb des FPGA (*Physical*) beinhaltet. Diese Schnittstellen umfassen beispielsweise den Zugriff des *Configuration Controllers* auf den Registersatz oder verschiedene interne Resets und Taktsignale (*Clock*). Innerhalb der *Testbench* werden die notwendigen Taktsignale für die ADPLL sowie den Registerzugriff aus der FPGA *Fabric* und durch den *Configuration Controller* erzeugt (Quellcode 5.1). Zu Beginn der Simulation werden zudem der interne Reset für die ADPLL und den Registersatz generiert (Quellcode 5.2).

```

695 parameter CLKPERIOD_REF = 10.0; // ADPLL
696 parameter CLKPERIOD_REG = 8.0; // Register
697 parameter CLKPERIOD_CFG = 40.0; // Config
698
699 initial clk_ref = 1'b1;
700 always #(CLKPERIOD_REF / 2) clk_ref = ~clk_ref; // 100 MHz
701
702 initial clk_reg = 1'b1;
703 always #(CLKPERIOD_REG / 2) clk_reg = ~clk_reg; // 125 MHz
704
705 initial clk_cfg = 1'b1;
706 always #(CLKPERIOD_CFG / 2) clk_cfg = ~clk_cfg; // 25 MHz

```

Quellcode 5.1: Generierung der notwendigen Taktsignale (*Clocks*).

```

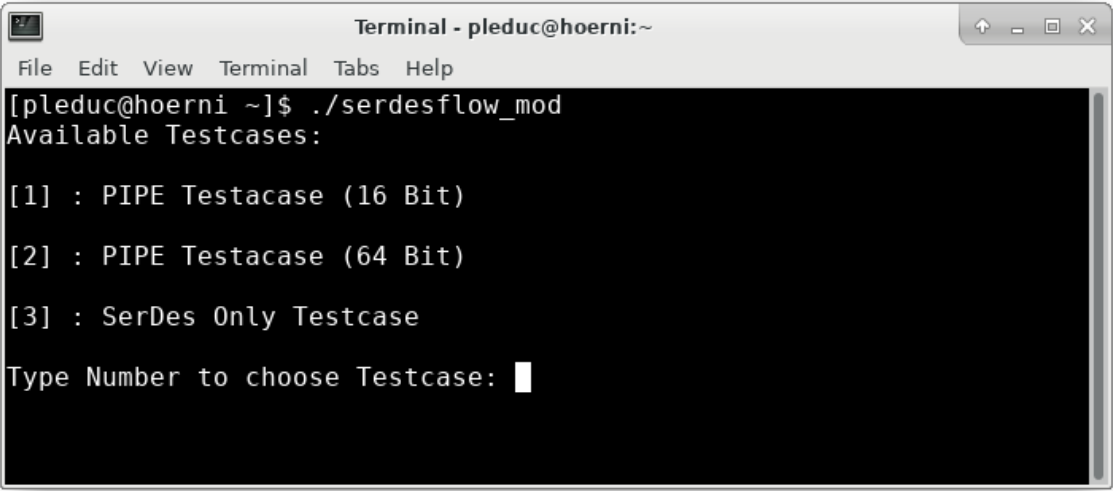
724 initial // ADPLL
725 begin
726     reset_n = 1'b0;
727     #(CLKPERIOD_REF * 10 + 1);
728     reset_n = 1'b1;
729 end
730
731 initial // Register
732 begin
733     reset_reg_n = 1'b0;
734     #(CLKPERIOD_REG * 10 + 1);
735     reset_reg_n = 1'b1;
736 end

```

Quellcode 5.2: Generierung der internen Resetsignale (*Physical*).

Für die Simulation inkludiert die *Testbench* am Ende ihres Moduls einen entsprechenden *Testcase*. Dieser enthält die Beschreibung des Simulationsablaufs in Form von Stimulussignalen und der Überprüfung des DUT. Diese Trennung zwischen *Testbench* und *Testcase* ermöglicht es, die *Testbench* als Ausgangsbasis für verschiedene Testfälle zu nutzen. Um die Durchführung der Simulation und die Ansteuerung des SerDes als DUT zu vereinfachen, sind in der *Testbench* mehrere Methoden (*Tasks*

und *Functions*) definiert, welche aus dem *Testcase* heraus aufgerufen werden können. Eine entsprechende Beschreibung der einzelnen Methoden findet sich in Kap. 5.2. Die Erweiterung der *Testbench* für den Test des IP-Cores ist modular gestaltet und lässt sowohl den Test des PIPE als auch weiterhin den normalen Test des SerDes zu. Die Auswahl der entsprechenden Anwendungsfälle wird über die beiden Makros PIPE und SERDES bestimmt, wobei jeweils eines der beiden Makros beim Aufruf der Simulationsumgebung definiert werden muss. Das jeweilige Makro sorgt für die entsprechende Instanziierung der Module und die Inkludierung des zugehörigen *Testcase*. Im Rahmen der Simulation mit Cadence Incisive und SimVision werden die Makros, innerhalb des entsprechenden *Scripts* (*serdesflow_mod*) definiert. Dies schließt auch den Wert des Parameters DATA_BYTES für die Bitbreite des PIPE IP-Cores mit ein. Daher wird der Anwender beim Start der Simulationsumgebung über das *Script* aufgefordert den gewünschten Testfall anzugeben (Abb. 5.2).



```
Terminal - pleduc@hoerni:~
File Edit View Terminal Tabs Help
[pleduc@hoerni ~]$ ./serdesflow_mod
Available Testcases:

[1] : PIPE Testacase (16 Bit)
[2] : PIPE Testacase (64 Bit)
[3] : SerDes Only Testcase

Type Number to choose Testcase: █
```

Abb. 5.2: Angabe des gewünschten Testfalls in der Konsole.

Eine vollständige Darstellung des *Scripts* befindet sich in Anhang A.3. Im Fall des IP-Cores wird auf die Integration des gesamten PIPE mit *Wrapper Block* verzichtet, da dieser die Instantiierung der *Blackbox Instantiation* für den SerDes vorsieht und nicht mit dem vollständigen SerDes vereinbar ist. Daher wird lediglich die PIPE Logic in die *Testbench* integriert und mit den entsprechenden Schnittstellen des SerDes (*FPGA Fabric*) verbunden. Der Test des IP-Core ist jedoch vollständig gewährleistet, da der *Wrapper Block* keine zu testende Funktionslogik implementiert. Ein Zugriff auf die Konfiguration des SerDes innerhalb des Registersatzes, kann über die entsprechenden Methoden für den Registerzugriff erfolgen. Alternativ

lassen sich über die Parameter des SerDes, auch direkt die Reset-Werte der Registerfelder anpassen. Dadurch verfügt der SerDes bereits zu Beginn der Simulation über die entsprechende Konfiguration und ein erneuter Zugriff entfällt. Für zukünftige Erweiterungen ist angedacht, den Zugriff des *Configuration Controllers* für die Konfiguration des SerDes mit in die *Testbench* zu integrieren. So könnte durch eine zusätzliche Methode die notwendige Konfiguration aus einem entsprechenden *Config File* in alle Register des SerDes geladen werden.

5.2 Methoden

Im folgenden Abschnitt werden die Methoden (*Functions* und *Tasks*), welche innerhalb der *Testbench* zur Verfügung stehenden, erläutert.

5.2.1 finalize

```
finalize();
```

Beschreibung: Die Methode kann innerhalb oder zum Ende eines Testfalls, dazu genutzt werden eine Konsolenausgabe für das Ergebnis der Simulation zu erzeugen und diese über den *System Task* \$finish zu beenden. Die Methode wertet im Rahmen ihres Aufrufs die Variablen *checks_done* und *errors* aus, welche innerhalb der *Testbench* erzeugt und mit 0 vorinitialisiert werden. Die Variable *checks_done* sollte nach jedem durchgeführten Test inkrementiert werden. Tritt ein entsprechender Fehlerfall auf kann die Variable *errors* inkrementiert werden, worauf normalerweise direkt der Aufruf der Methode *finalize()* erfolgt. Es werden je nach Wert der beiden Variablen die Ausgaben PASSED, FAILED oder UNKNOWN in der Konsole erzeugt.

Anmerkung: Diese Methode entstammt der ursprünglichen SerDes *Testbench*, aber kann auch im Rahmen der PIPE Erweiterung eingesetzt werden.

5.2.2 resetSerDes

```
resetSerDes();
```

Beschreibung: Die Methode führt nacheinander einen Reset des Tx-Datenpfades und des Rx-Datenpfades des SerDes durch und kontrolliert die entsprechenden Sta-

tussignale. Bei der Durchführung der Resets wird zudem kontrolliert, ob die Flankenwechsel der beiden Statussignale innerhalb eines gewissen Zeitrahmens stattfinden und auch keinen unbekanntem Wert aufweisen. Sollte ein Fehlerfall auftreten, erhöht die Methode die Fehlervariable (*error*) um 1 und ruft die Methode *finalize()* auf.

Anmerkung: Diese Methode entstammt der ursprünglichen SerDes *Testbench* und kann nicht im Rahmen der PIPE Erweiterung eingesetzt werden.

5.2.3 writeCfg

```
writeCfg(<addr>, <data>);
```

Argumente:

addr	wire [7:0]	Registeradresse
data	wire [15:0]	Datenwert

Beschreibung: Die Methode ermöglicht es über die Schnittstellen des *Configuration Controllers* Daten in ein bestimmtes Register des SerDes zu schreiben. Als Argumente müssen die 16-Bit bereiten Daten (*data*) und die 8-Bit breite Adresse des Registers (*addr*) übergeben werden. Es ist zu beachten, dass die Methode das gesamte Register überschreibt.

Anmerkung: Diese Methode entstammt der ursprünglichen SerDes *Testbench*, aber kann auch im Rahmen der PIPE Erweiterung eingesetzt werden. Die *Testbench* enthielt ebenfalls eine Methode mit identischer Funktion (*writeCfgFile*). Diese ist im Verlauf der Erweiterung entfernt worden, um mögliche Verwechslungen auszuschließen.

5.2.4 writeRegfile

```
writeRegfile(<addr>, <data>, <mask>);
```

Argumente:

addr	wire [7:0]	Registeradresse
data	wire [15:0]	Datenwert
mask	wire [15:0]	Registermaske

Beschreibung: Die Methode ermöglicht es, über die Schnittstellen für den Registerzugriff (FPGA-Logik) Daten in ein bestimmtes Register des SerDes zu schreiben. Als Argumente müssen die 16-Bit breiten Daten (*data*), die 8-Bit breite Adresse des Registers (*addr*) sowie eine Maskierung für die Daten (*mask*) übergeben werden. Die Maskierung sorgt dafür, dass Daten nur an den Stellen mit einer 1 in der Registermaske übernommen werden.

Anmerkung: Diese Methode entstammt der ursprünglichen SerDes *Testbench*, aber kann auch im Rahmen der PIPE Erweiterung eingesetzt werden.

5.2.5 readRegfile

```
readRegfile (<addr>, <data>);
```

Argumente:

addr	wire [7:0]	Registeradresse
data	wire [15:0]	Ausgelesener Datenwert

Beschreibung: Die Methode ermöglicht es, über die Schnittstellen für den Registerzugriff (FPGA-Logik) die Daten eines bestimmten Registers auszulesen. Als Argument muss zunächst die 8-Bit breite Adresse des Registers (*addr*) übergeben werden. Die Rückgabe des Datenwertes erfolgt auf die angegebene Variable im zweiten Argument (*data*).

Anmerkung: Diese Methode entstammt der ursprünglichen SerDes *Testbench*, aber kann auch im Rahmen der PIPE Erweiterung eingesetzt werden.

5.2.6 round

```
round (<number>);
```

Argumente:

number	real	Gleitkommazahl
--------	------	----------------

Beschreibung: Diese Funktion kann dazu genutzt werden, die übergebene Gleitkommazahl auf eine nächstgelegene ganze Zahl auf oder abzurunden. Die Funktion setzt dazu die mathematischen Funktionen `$ceil()` und `$floor()` ein. Der Rückgabewert der Funktion ist ebenfalls vom Datentyp *real*.

Anmerkung: Diese Methode entstammt der ursprünglichen SerDes *Testbench*, aber kann auch im Rahmen der PIPE Erweiterung eingesetzt werden.

5.2.7 readSerIOADPLLStatus

```
readSerIOADPLL(<status>);
```

Argumente:

status	wire[31:0]	ADPLL Statusregister
--------	------------	----------------------

Beschreibung: Diese Methode liest nacheinander die beiden 16-Bit breiten Statusregister mit den Adressen 0x55 und 0x56 aus und gibt diese als 32-Bit breiten Vektor (*status*) zurück. Die Methode nutzt dabei die Methode *readRegfile()*. Sollten nach Auslesung der Register unbekannte Werte auftreten, erhöht die Methode die Fehlervariable (*error*) um 1 und ruft die Methode *finalize()* auf.

Anmerkung: Diese Methode entstammt der ursprünglichen SerDes *Testbench*, kann aber auch im Rahmen der PIPE Erweiterung eingesetzt werden.

5.2.8 startSerIOADPLL

```
startSerIOADPLL(<mainDivN1>, <mainDivN2>, <mainDivN3>,
                <OutDiv>, <enableCalib>);
```

Argumente:

mainDivN1	integer	Teilerwert N2 für die ADPLL
mainDivN2	integer	Teilerwert N1 für die ADPLL
mainDivN3	integer	Teilerwert N3 für die ADPLL
outDiv	integer	Teilerwert für den <i>Output Divider</i>
enableCalib	bit	Kalibrierung (1 : aktiviert, 0 : deaktiviert)

Beschreibung: Mit dieser Methode lässt sich die ADPLL hinsichtlich ihrer Frequenz konfigurieren und wahlweise auch kalibrieren. Für die Konfiguration der generierten Frequenz, werden die Hauptteilerwerte (N1-N3), sowie der Wert des zusätzlichen Teilers (*outDiv*) übergeben. Dabei können die Teilerwerte direkt ohne ihre Kodierung angegeben werden, da diese automatisch innerhalb der Methode erzeugt wird. Zusätzlich erfolgt eine Überprüfung der angegebenen Teilerwerte, die bei Werten außerhalb des zulässigen Wertebereichs zu einem Abbruch der Simulation und zur Fehlerausgabe durch die Methode *finalize()* führt. Zu Beginn prüft die Methode über das Registerfeld PLL_LOCKED den Status der ADPLL und schaltet diese anschließend über das Registerfeld PLL_EN_ADPLL_CTRL ab. Die übergebenen Teilerwerte werden mit der angepassten Kodierung in die entsprechenden Registerfelder PLL_MAIN_DIVSEL und PLL_OUT_DIVSEL geladen. Das Registerfeld PLL_FCNTL mit Bezug auf die *Core Clock* bleibt unverändert. Falls die Kalibrierung aktiviert worden ist, schaltet die Methode über das Registerfeld PLL_BISC_MODE die *Build-In-Self-Calibration* (BISC) ab und passt deren Registerfelder innerhalb der Registeradressen 0x58 und 0x57 an. Die Konfiguration erfolgt auf Basis von Parametern, die innerhalb der *Testbench* definiert sind (Quellcode 5.3).

```

264     parameter      ADPLL_PFDAC_TIMER      = 12;
265     parameter      ADPLL_PFDAC_COR_DLY    = 1;
266     parameter      ADPLL_PFDAC_CP_MIN     = 6;
267     parameter      ADPLL_PFDAC_CP_MAX     = 30;
268     parameter      ADPLL_PFDAC_CP_START   = 6;
269     parameter      ADPLL_PFDAC_CAL_SIGN   = 1;
270     parameter      ADPLL_PFDAC_AUTO_CAL   = 1;

```

Quellcode 5.3: Definierte Parameter für die Konfiguration der ADPLL

Nach der Konfiguration werden die ADPLL und die BISC wieder über die Registerfelder PLL_EN_ADPLL_CTRL und PLL_BISC_MODE aktiviert. Anschließend ruft die Methode dauerhaft den Status der ADPLL ab und überprüft, ob das Regis-

terfeld PLL_LOCKED gesetzt ist. Sollte das Registerfeld innerhalb eines gewissen Zeitraums nach der Aktivierung der ADPLL nicht gesetzt sein, führt dies zu einer entsprechenden Fehlerausgabe über die Methode *finalize()* und dem Abbruch der Simulation. Die Methode liefert im Rahmen der Kalibrierung auch die Statusregister der BISC mit den Adressen 0x5A und 0x5B aus und gibt deren Statuswerte neben bestimmten Statuswerten der ADPLL in der Konsole aus. Es wird intern auf die Methoden *round()*, *writeRegfile()*, *readRegfile* und *readSerIOADPLLStatus()* zurück gegriffen.

Anmerkung: Diese Methode entstammt der ursprünglichen SerDes *Testbench*, kann aber auch im Rahmen der PIPE Erweiterung eingesetzt werden. Es ist zu beachten, dass die Methode einen Fehler in der Verwendung ihrer Argumente besitzt. Die Definition der Argumente *mainDivN1* und *mainDivN2* ist in ihrer Implementierung vertauscht.

5.3 PIPE Testcase

Der PIPE Testcase (*testcase_pipe.v*) wird über die Definition des entsprechenden *Macros* mit eingebunden und dient dazu, die verschiedenen Anwendungsfälle des PIPE IP-Core zu testen. Die Funktionalität der einzelnen Module ist bereits im Zusammenhang mit der Entwicklung in kleineren *Testbenches* erfolgt, weshalb der *Testcase* vorwiegend auf das gesamte Design unter Einbezug des SerDes abzielt. Innerhalb des *Testcase* werden die Sendeleitungen des SerDes mit dessen Empfangsleitungen verbunden und zusätzlich mit einer Verzögerungszeit von 10 ns belegt (Quellcode 5.4).

```
33 always @(TX_SERIO_P) begin
34
35     if ( S_FAULT_INJECTION == 1'b1 )
36         RX_SERIO_P <= #10 ~TX_SERIO_P;
37     else
38         RX_SERIO_P <= #10 TX_SERIO_P;
39
40     end // always
41
42 always @(TX_SERIO_N) begin
43
44     if ( S_FAULT_INJECTION == 1'b1 )
45         RX_SERIO_N <= #10 ~TX_SERIO_N;
46     else
47         RX_SERIO_N <= #10 TX_SERIO_N;
48
49     end // always
```

Quellcode 5.4: Verbindung der Sende- und Empfangsleitungen des SerDes.

Die Verschaltung zwischen *Transmitter* und *Receiver* beinhaltet zusätzlich die notwendige Logik, die es ermöglicht einzelne Bits in der Übertragung zu invertieren. Diese Funktion wird für den Test der *Error Detection* und die damit verbundene Generierung von Übertragungsfehlern benötigt (Kap. 5.3.5). Die Reset-Verbindung zwischen der ADPLL und den beiden Datenpfaden wird ebenfalls mit in den *Testcase* aufgenommen, obwohl diese auch innerhalb der *Testbench* erfolgen könnte. Diese Verbindung liegt im FPGA vor und ist nicht aus der *FPGA Fabric* zugänglich. Sie besteht zwischen der Ausgangsschnittstelle *reset_core_pll_n_o* der ADPLL und den beiden Eingangsschnittstellen *reset_core_tx_n_i* und *reset_core_rx_n_i* der Datenpfade. Im *Testcase* wird die Verbindung über die Zuweisung der entsprechenden Signale durchgeführt (Quellcode 5.5).

```
57 assign reset_core_tx_n = reset_core_pll_n;  
58 assign reset_core_rx_n = reset_core_pll_n;
```

Quellcode 5.5: Reset-Verbindung zwischen der ADPLL und den Datenpfaden

Der *Testcase* führt zu Beginn der Simulation einen asynchronen Reset von 20 ns für die PIPE Logic aus und legt die Initialisierungswerte für den Resetfall an. Dadurch befindet sich die PIPE Logic im Ausgangszustand und wartet den Reset und Konfigurationsvorgang des SerDes ab. Die Erzeugung der einzelnen Testfälle findet anschließend innerhalb eines *Initial* Block statt.

***Receiver* : Zusammenhang zwischen Reset und CDR**

Das SerDes-Modul *ri_serdes_tx_driver* definiert für den Fall von *Electrical Idle*, dass der *Transmitter* auf seinen Sendeleitungen den Wert *unknown* bzw. 1'bx ausgibt. Dies führte im Zusammenhang mit der Empfangslogik des *Receivers* zu Problemen mit dem Reset-Verhalten. Während des Resets durchläuft der *Receiver* im Modul *ccfpga_serdes_rx_rst_core* mehrere Zustände einer FSM. Diese gewährleisten, dass nacheinander alle notwendigen Resets durchgeführt und bestimmte Ereignisse eingetreten sind. Einer dieser FSM Zustände (*S_WAIT_CDR_LOCK*) stellt sicher, dass die CDR synchronisiert bzw. *locked* ist oder alternativ ein *Loopback* aktiviert worden ist.

Eine Synchronisierung und Zuweisung des Signals `rx_cdr_locked_sync_i` erfolgt im Zusammenhang mit der Simulation jedoch nicht, da im *Testcase* die Sendeleitungen des *Transmitters* mit denen des *Receivers* verbunden sind. Der *Transmitter* sorgt abhängig vom PIPE in seinem Initialisierungszustand (Reset bzw. P1) dafür, dass die Sendeleitung sich im *Electrical Idle* Zustand befindet. Der daraufhin zugewiesene Wert für die Sendeleitung (1'bx) führt in der CDR-Logik (`ccfpga_serdes_rx_cdr.v`) dazu, dass keine Synchronisation erfolgt und die FSM niemals den FSM Zustand verlässt. Im Gegenzug führen konstante Signale von 1'b1 oder 1'b0 auf den Empfangsleitungen jedoch zur einer erfolgreichen Synchronisation der CDR, auch wenn keine alternierende Übertragung und die damit verbundenen Flankenwechsel auftreten. Die Spezifikation des SerDes [Rac20] geht nicht näher auf das Resetverhalten des *Receivers* im Zusammenhang mit der CDR ein, jedoch erscheint ein Flankenwechsel für die Ableitung der *Recovered Clock* und des Abgriffzeitpunktes der Daten zwingend erforderlich. Daher wird zum gegenwärtigen Zeitpunkt davon ausgegangen, dass der Reset des *Receivers* von der Synchronisation der CDR und einer eingehenden Übertragung abhängig ist. Dahingehend kann der Reset des PIPE IP-Cores nicht mehr, wie ursprünglich vorgesehen, von dem Statussignal `rx_resetdone_o` des *Receivers* abhängig gemacht werden.

5.3.1 Reset

Der Reset des PIPE IP-Cores sieht zwei mögliche Testfälle vor. Zum einen den Reset in Verbindung mit dem Reset des gesamten FPGAs (*Power On Sequence*) und zum anderen einen alleinigen Reset des SerDes bzw. des PHY während des normalen Betriebszustands. Zu Beginn der Simulation werden die Resetsignale des SerDes (`reset_n_i`) und des PIPE (`i_Reset`) automatisch erzeugt und versetzen beide in einen definierten Ausgangszustand. Zu diesem Zeitpunkt sind der SerDes und die ADPLL noch nicht über die entsprechenden Registerfelder aktiviert worden. Dies erfolgt anschließend über die Methode `writeRegfile()` und führt zu Generierung der *Core Clock* durch die ADPLL. Die notwendige Konfiguration für die ADPLL ist zuvor bereits durch die Anpassung der Reset-Werte (*Parameter*) in der *Testbench* erfolgt. Alternativ kann hier die Methode `startSerIOADPLL()` mit den entsprechenden Argumenten für 1,25 GHz eingesetzt werden.

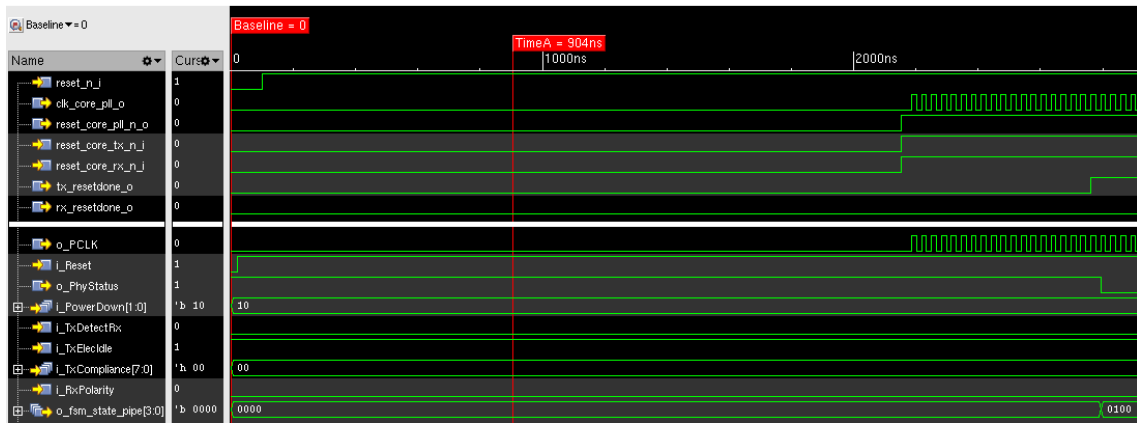


Abb. 5.3: Ablauf des Reset für den PIPE IP-Core (*Power On Sequence*).

In Abb 5.3 ist der Simulationsverlauf für den Reset dargestellt. Nachdem der Reset für den SerDes und die PIPE Logic durchgeführt worden ist, werden nach einer Wartezeit von 500 ns der SerDes und die ADPLL aktiviert (*TimeA*). Sobald die ADPLL über einen *Lock* verfügt, beginnt diese mit der Generierung der *Core Clock* (*clk_core_pll_o*) und nimmt den Tx- und Rx-Datenpfad aus dem Reset (*reset_core_tx_n_i* und *reset_core_rx_n_i*). Nachdem der Tx-Datenpfad signalisiert, dass er seinen Reset abgeschlossen hat (*tx_resetdone_o*), wechselt die FSM der PIPE Logic in den Zustand P1_IDLE (0100) und hält das Statussignal *PhyStatus* nicht mehr länger aktiv. Der Resetvorgang des PIPE IP-Cores ist damit abgeschlossen. Für einen Reset während des normalen Betriebs werden durch die MAC der asynchrone Reset (*i_Reset*) und die geforderten Reset-Werte für die Kontrollsignale angelegt. Im Gegensatz zur PIPE Spezifikation darf der Reset nicht gehalten werden, sondern erfolgt nur über einen entsprechenden Zeitraum (Abb. 5.4).

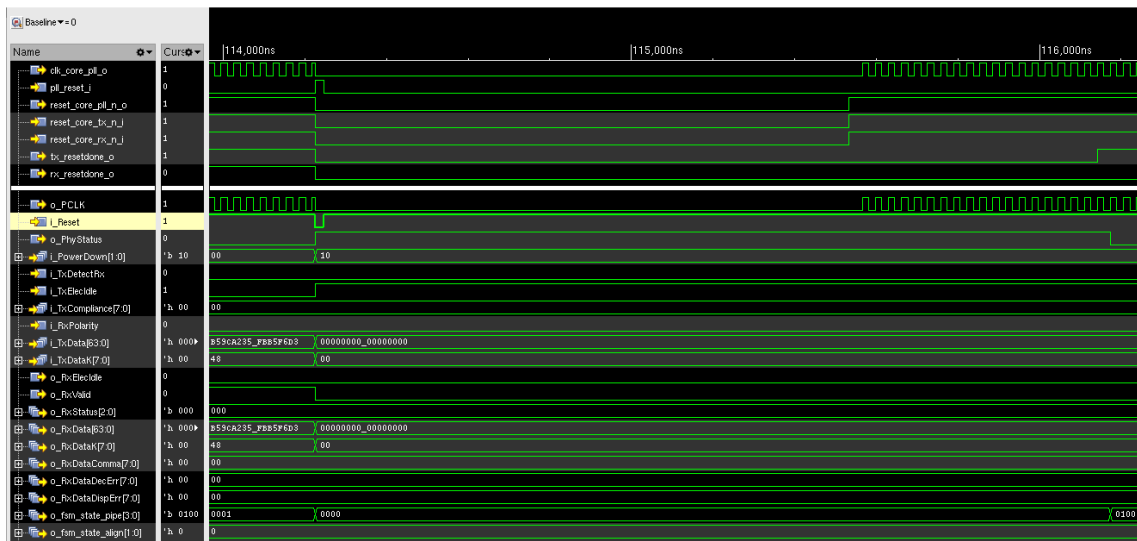
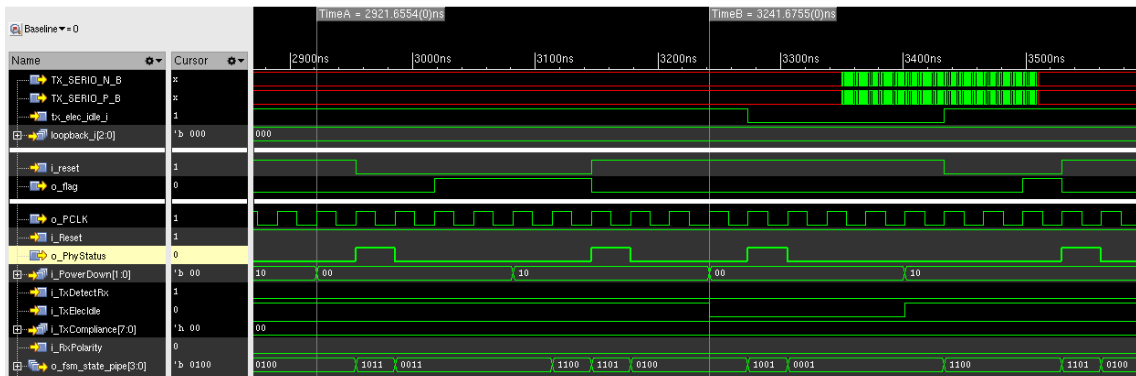
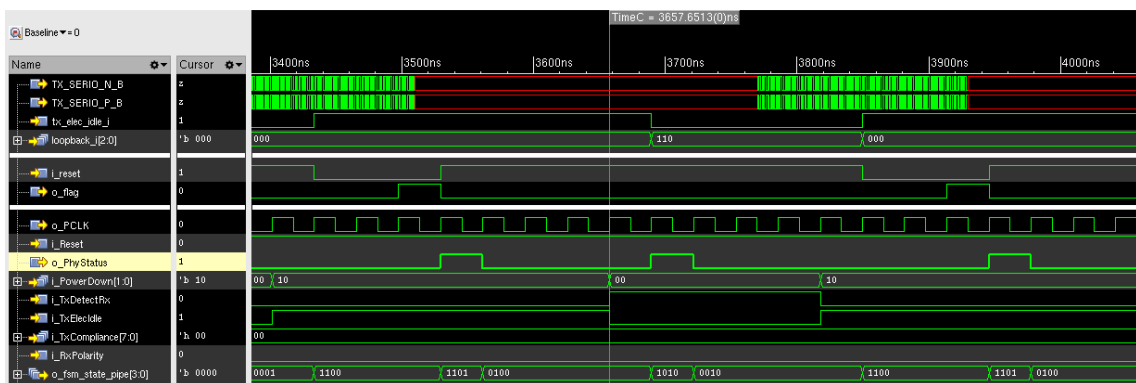


Abb. 5.4: Reset des PIPE IP-Core im normalen Betriebszustand.

Der Simulationsverlauf zeigt, dass der Reset des PIPE (i_Reset), den Reset der ADPLL von Seiten der *FPGA Fabric* (pll_reset_i) auslöst. Die Zuweisung erfolgt dabei invertiert, da der Reset der ADPLL im Gegensatz zum PIPE *High Active* ist. Die ADPLL versetzt daraufhin auch den Tx- und Rx-Datenpfad in den Reset ($reset_core_tx_n_i$ und $reset_core_rx_n_i$). Innerhalb der PIPE Logic wird der Reset an alle internen Module weiter geleitet und sorgt für den Übergang in den Reset-Zustand, was sich auch an den Ausgangsschnittstellen zeigt. Die interne FSM der PIPE Logic befindet sich im Zustand RESET (0000) und zieht das Statussignal *PhyStatus* auf 1. Ab diesem Zeitpunkt verhält sich der Ablauf des Reset analog zum vorherigen Reset (*Power On Sequence*).

5.3.2 Power States

Für den Wechsel zwischen den *Power States* muss gewährleistet werden, dass das Statussignal *PhyStatus* beim Eintritt in den entsprechenden FSM Zustand für genau eine Taktperiode von PCLK aktiv ist und die notwendigen Kontrollsignale für den SerDes generiert werden. Daher wird im *Testcase* der Übergang zwischen dem Zustand P1 und den drei möglichen Zuständen für P0 (*Normal Transmission*, *Electrical Idle* und *Loopback*) getestet (Abb. 5.5 und 5.6).

Abb. 5.5: Test der Übergänge zwischen den *Power States* (Teil 1).Abb. 5.6: Test der Übergänge zwischen den *Power States* (Teil 2).

Es erfolgt zunächst der Übergang aus P1 (10) in den Zustand P0 (00) mit aktiviertem *Electrical Idle* und anschließend zurück in den Zustand P1 (*TimeA*). Es folgen auf identische Weise der Übergang in den Zustand P0 für die normale Übertragung (*TimeB*) und in den Zustand P0 mit aktivierten *Loopback* (*TimeC*). Die Übergänge zwischen den Zuständen werden jeweils durch das Anlegen der notwendigen Kontrollsignale des PIPE ausgelöst. Aus dem Simulationsverlauf lässt sich entnehmen, dass die interne FSM der PIPE Logic auf die anliegenden Kontrollsignale reagiert und die entsprechenden Zustände durchläuft (*o_fsm_state_pipe*). In diesem Zusammenhang erfolgt auch die Generierung des *PhyStatus* und der Kontrollsignale für den SerDes (*tx_elec_idle* und *loopback_i*). Beim Übergang in den *PowerState* P1 ist zu beachten, dass der Eintritt in den *Power State* auch vom Flag des zugehörigen Zählers (*o_flag*) abhängig ist, welcher je nach FSM Zustand aktiv ist oder im Reset gehalten wird (*i_reset*).

5.3.3 Word Alignment

Der Test des *Word Alignment* und die damit verbundene Zuweisung des Statussignals *o_RxValid* erfolgt durch die Übertragung von Datensätzen, welche an das *TS1 Ordered Set* angelehnt sind. Die Übertragung wird durch die Methode *send_TS1_OS()* realisiert, die innerhalb des *Testcase* definiert ist und einen Datensatz von 16 Bytes erzeugt. Der Datensatz beginnt wie das *TS1 Ordered Set* mit einem COM Symbol (0xBC) und wird für die restlichen Bytes mit statischen Datenbytes aufgefüllt. Die Daten werden über die entsprechenden *Transmitter*-Schnittstellen (*i_TxData* und *i_TxDataK*) an den *Receiver* gesendet.

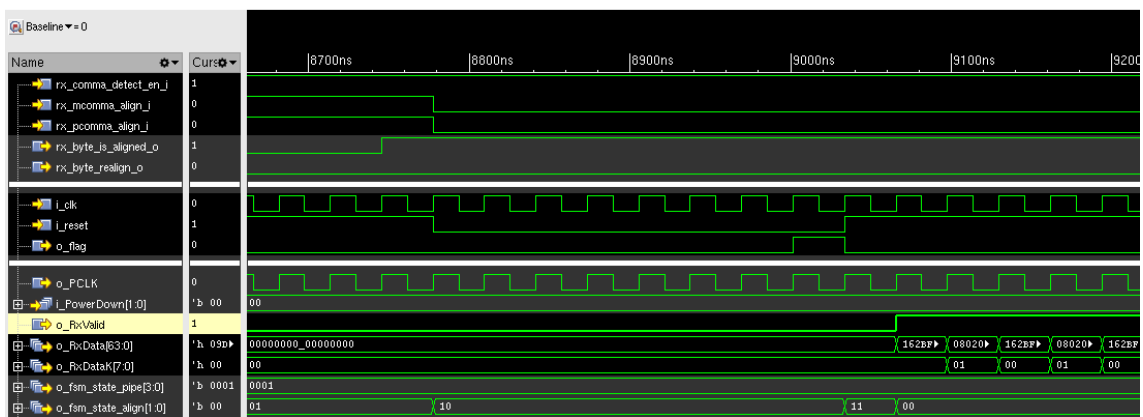


Abb. 5.7: Ablauf des *Word Alignment* und Zuweisung von *o_RxValid*.

Im Simulationsverlauf ist zu erkennen, dass sich die PIPE Logic im *Power State P0* (00) befindet und somit die FSM für das *Byte Alignment* aktiv ist (Abb. 5.7). Da kein *Byte Lock* (*rx_byte_is_aligned_o*) vorliegt, hat die FSM die Kontrollsignale für das *Word Alignment* (*rx_mcomma_align_i* und *rx_pcomma_align_i*) bereits auf logisch 1 gezogen. Nachdem der *Byte Lock* erfolgt ist, nimmt die FSM den zugehörigen Zähler aus dem Reset (*i_reset*) und wartet bis dieser seinen definierten Wert erreicht und das entsprechende *Flag* (*o_flag*) ausgibt. Anschließend erfolgt die Ausgabe des Statussignals *o_RxValid* passend zum Auftreten der Empfangsdaten an den Schnittstellen des *Receivers* (*i_RxData* und *i_RxDataK*). Die FSM durchläuft während des Vorgangs die notwendigen Zustände (*o_fsm_state_align*). Das *Word Alignment* des PIPE IP-Cores sieht beim Eintritt in den *Power State P1* und dem Verlust des *Alignments* zurzeit noch keine Verzögerung des Statussignals *o_RxValid* vor. Ein Verlust des *Alignments* hat zeitgleich den Wechsel des Statussignals *o_RxValid* auf 0 zur Folge. Es ist allerdings möglich, dass für einen gewissen

Zeitraum weiterhin valide Daten aus dem *Elastic Buffer* die Empfangsschnittstellen erreichen.

5.3.4 Normal Transmission

Der Test der normalen Übertragung zwischen *Transmitter* und *Receiver* ist weitestgehend automatisiert worden, so dass keine manuelle Überprüfung mehr notwendig ist. Der *Testcase* verfügt für die Automatisierung über die Methoden *generate_data*, *send_testdata()* und *check_testdata()*. Des Weiteren deklariert der *Testcase* für die erforderlichen Testdaten entsprechende *Arrays* (*testdata*, *testdata_k* und *testdata_com*). Die *Arrays* umfassen je 40 Elemente und sind im Fall der Sendedaten (*testdata*) jeweils 8-Bit breit. Die Elemente der Kontrolldaten (*testdata_k* und *testdata_com*) sind je 1-Bit breit. Um eine möglichst hohe Testabdeckung zu erreichen, werden die *Arrays* über die Methode *generate_data* mit Testdaten gefüllt. Die Testdaten bestehen dabei aus verschiedenen Daten- und Kontrollbytes und werden vollständig in ihren zulässigen Werten und ihrer Position im *Array* randomisiert. Sind die Testdaten erzeugt worden, können diese über die Methode *send_testdata()* auf die Schnittstellen des *Transmitters* gelegt und an den *Receiver* gesendet werden. Nach der entsprechenden Verzögerungszeit zwischen *Transmitter* und *Receiver*, wird im *Testcase* die Methode *check_testdata()* aufgerufen. Diese greift auf die *Arrays* der Testdaten zu und vergleicht diese mit den eingehenden Empfangsdaten an den Schnittstellen *o_RxData*, *o_RxDataK* und *o_RxDataComma*. Es ist wichtig, dass die Testdaten nicht durch einen erneuten Aufruf von *generate_data* überschrieben werden, bis die Überprüfung durch *check_testdata()* erfolgt ist. Während des Datenabgleiches gibt die Methode kontinuierlich die verglichenen Daten in der Konsole aus (Abb. 5.8). Tritt ein entsprechende Fehler auf, erhöht die Methode den Fehlerzähler (*error*) und beendet die Simulation über Aufruf der Methode *finalize()*. Innerhalb des *Testcase* wird der eben beschriebene Testablauf insgesamt einhundertmal wiederholt.

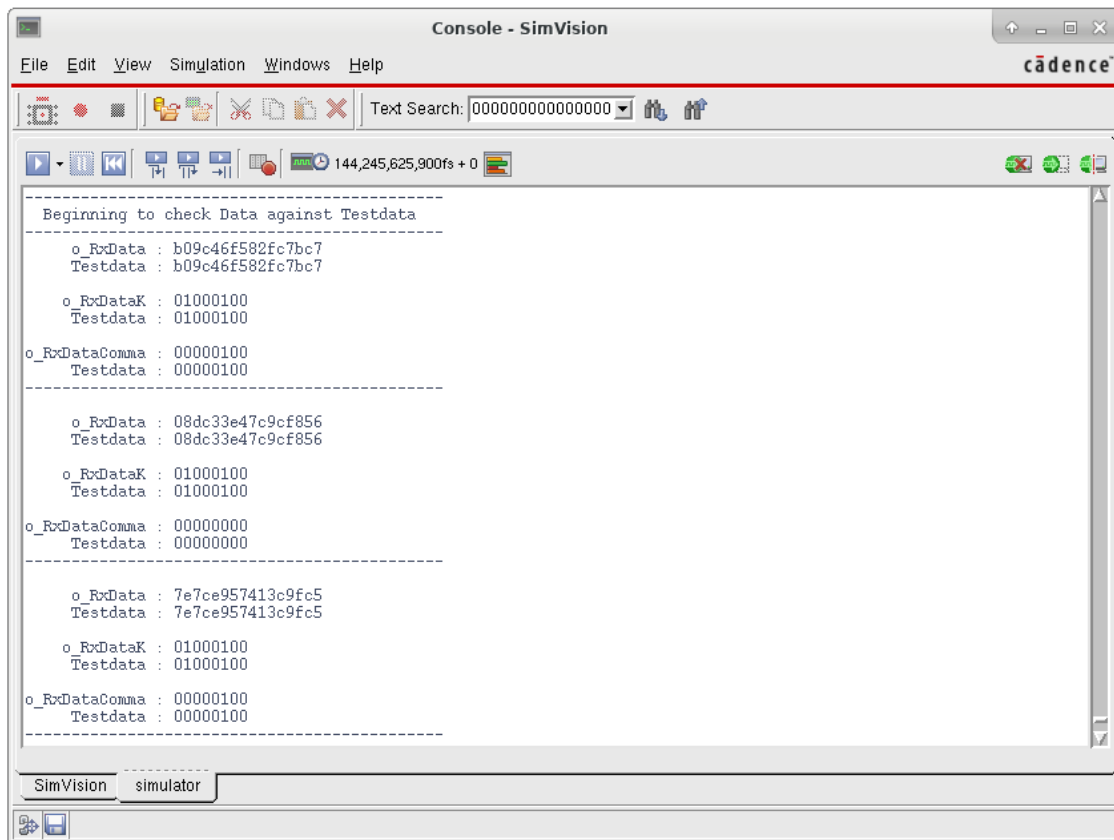


Abb. 5.8: Ausgabe der Methode `check_testdata()` in der Konsole.

5.3.5 Error Detection

Im Rahmen der *Error Detection* wird sichergestellt, dass der PIPE IP-Core die Fehler in der 8b/10b Kodierung und der Disparität (*Disparity*) vom SerDes übernimmt. Ebenfalls wird die Ausgabe von `o_RxStatus` und die Integration des EDB Symbols (0xFE) in die Ausgabedaten (`o_RxData`) kontrolliert. Der Test ist in Anlehnung an den Test für die normale Übertragung (Kap. 5.3.4) weitestgehend automatisiert worden, um eine möglichst hohe Testabdeckung zu erreichen. Innerhalb des *Testcase* sind dafür die beiden Methoden `force_transm_err()` und `check_err_detect()` definiert. Die Methode `force_transm_err()` weist dem Signal `S_FAULT_INJECTION` in regelmäßigen Abständen eine logische 1 zu und hält es für eine Zeitspanne von 0,4 ns aktiv. Die Zuweisung des Signals führt in den *Always*-Blöcken für die Verschaltung zwischen dem *Transmitters* und dem *Receiver* dazu, dass in der Übertragung genau 1-Bit invertiert zugewiesen wird (Quellcode 5.4).

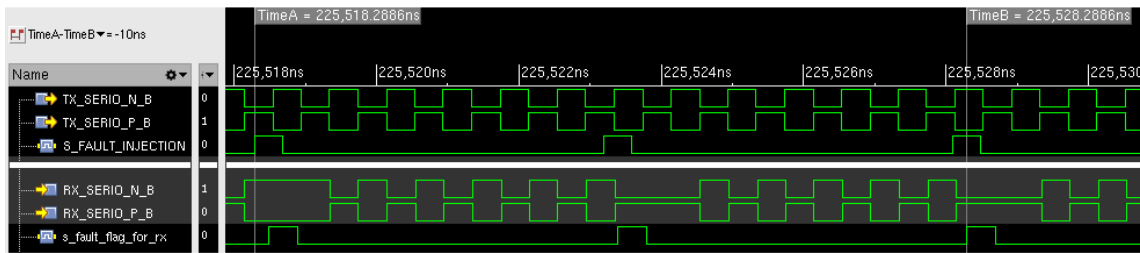


Abb. 5.9: Erzeugung von Bit-Fehlern auf den Empfangsleitungen.

Die entsprechende Invertierung der Übertragung lässt sich auch deutlich im Simulationsverlauf erkennen (Abb. 5.9). Die Zuweisung von `S_FAULT_INJECTION` zum Zeitpunkt `TimeA`, beeinflusst beim Flankenwechsel auf den Sendeleitungen (`TX_SERIO_*_B`) die Zuweisung der Empfangsleitungen (`RX_SERIO_*_B`). Die Zuweisung der Empfangsleitung erfolgt 10 ns später und damit nachdem Zeitpunkt `TimeB`. Das Signal `s_fault_flag_for_rx` dient hierbei als Referenz mit 10 ns Verzögerung zum eigentlichen Fehlersignal. Es verdeutlicht, dass während des aktiven Referenzsignals der eigentliche Flankenwechsel auf der Sendeleitung nicht auf der Empfangsleitung erfolgt. Die Invertierung kann dabei sowohl Fehler in der 8b/10b Kodierung als auch in der Disparität (*Disparity*) hervorrufen. Die zeitlichen Abstände zwischen den integrierten Bitfehlern sind in der Methode auf 4,5 ns festgelegt worden, wodurch pro übertragenem Byte genau 1-Bit invertiert bzw. *gekippt* wird. Aufgrund des zeitlichen Abstands ist es jedoch auch möglich, dass die Übertragung eines Bytes (4 ns) übersprungen wird und keine Integration eines Bitfehlers stattfindet. Die Überprüfung der Ausgabedaten des PIPE IP-Core wird durch die Methode `check_err_detect()` durchgeführt. Die Methode wird parallel zur Methode `force_transm_err()` ausgeführt und beginnt zeitverzögert zum Auftritt der Fehler im *Receiver* mit der Überprüfung. Dabei werden die Werte der Ausgangsschnittstellen des SerDes (`rx_not_in_table_o` und `rx_disp_err_o`) mit denen des PIPE (`o_RxDataDecErr` und `o_RxDataDispErr`) abgeglichen. Ebenfalls wird in Abhängigkeit zu den vorliegenden Fehlern die Zuweisung von `o_RxStatus` und die Integration des EDB Symbols an der richtigen Bytestelle der Empfangsdaten (`o_RxData`) überprüft. Tritt bei der Kontrolle der Daten ein entsprechender Fehler auf, erhöht die Methode den Fehlerzähler (`error`) und beendet die Simulation durch Aufruf der Methode `finalize()`. Während des Datenabgleiches gibt die Methode kontinuierlich Informationen über die abgeglichenen Daten in der Konsole des Simulators aus (Abb. 5.10).

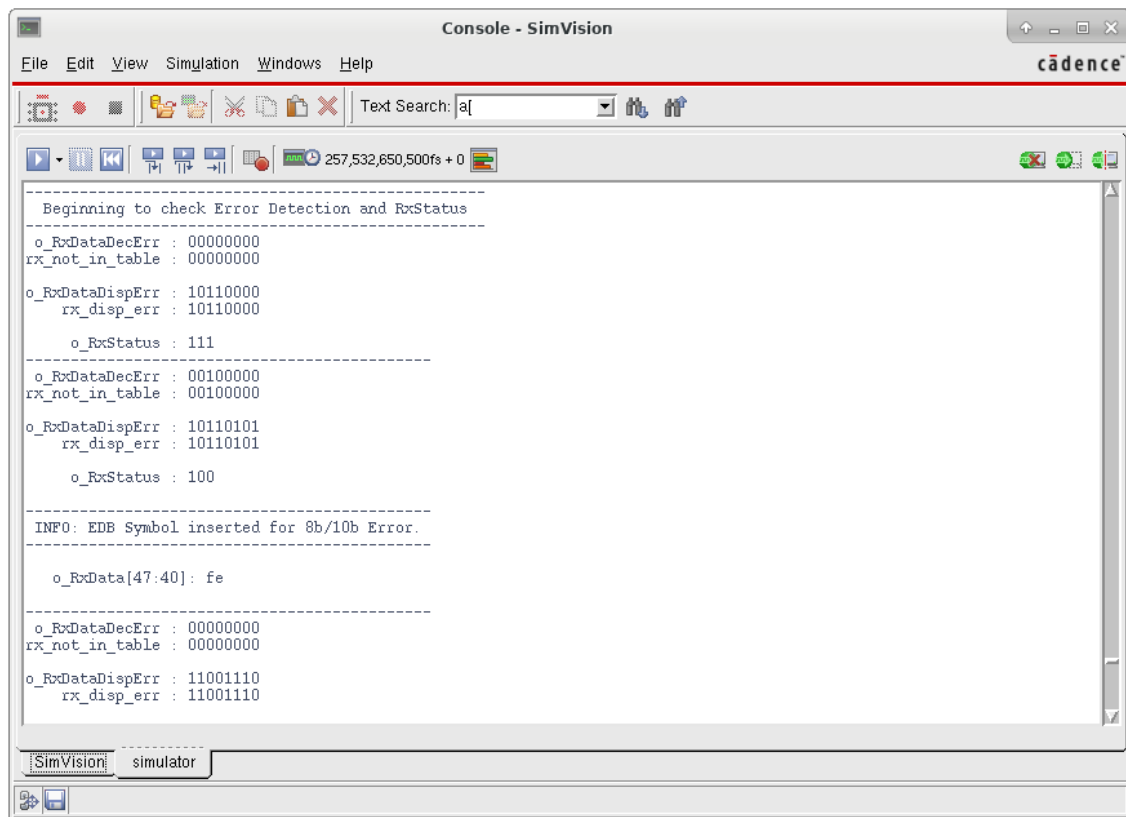


Abb. 5.10: Ausgabe der Methode `check_err_detect()` in der Konsole.

5.3.6 Negative Disparity

Für die Anpassung der Disparität (*Disparity*) muss gewährleistet werden, dass über die Schnittstelle `i_TxCompliance` die 8b/10b Kodierung der übertragenen Bytes entsprechend angepasst wird. Für den Test wird die Übertragung des COM Symbols (0xBC) für alle Byte-Positionen durchgeführt und über `i_TxCompliance` angepasst. Im Simulationsverlauf ist zu erkennen, dass die Zuweisung von `i_TxCompliance` an der SerDes Schnittstelle `tx_char_dispmode_i` übernommen wird und zusammen mit den konstanten Werten an `tx_char_dispval_i` (0x00) zu einer negativen Disparität führen (Abb. 5.11).

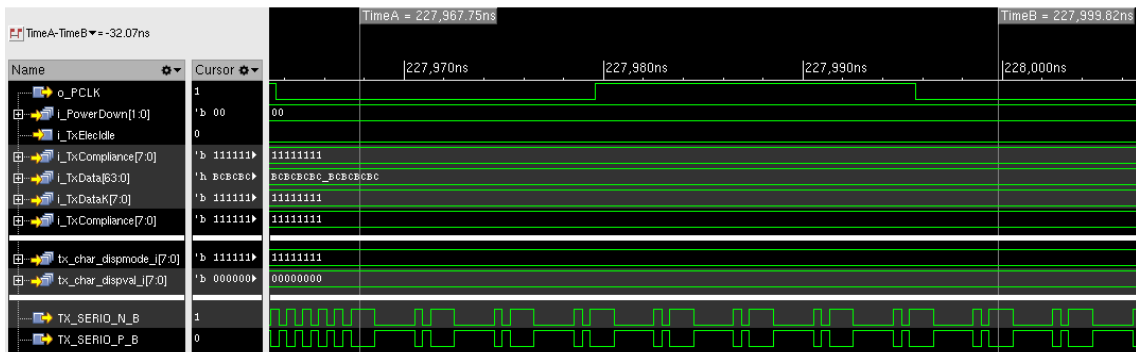


Abb. 5.11: Anpassung der Disparität durch $i_TxCompliance$ (Teil 1).

Die negative Disparität zeigt sich auf der Empfangsleitung durch die aufeinander folgende Kodierung $110000\ 0101$, welche der Kodierung mit negativer Disparität für das COM Symbol (0xBC) entspricht. Die beiden Marker $TimeA$ und $TimeB$ rahmen hierbei die Übertragung von insgesamt 8 Byte bzw. 80-Bit ein. Des Weiteren wird kontrolliert, ob ein darauf folgendes Byte sich an der neuen CRD, die durch das angepasste Byte erzeugt worden ist, orientiert. Dazu erfolgt die gleiche Übertragung der COM Symbole, wobei nur jedes zweite Byte durch $i_TxCompliance$ angepasst wird (Abb. 5.12).

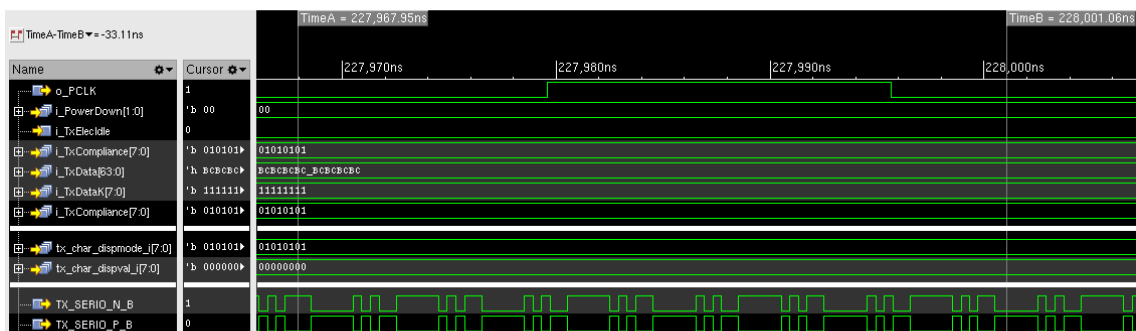


Abb. 5.12: Anpassung der Disparität durch $i_TxCompliance$ (Teil 2).

Im Simulationsverlauf zeigt sich, dass die darauf folgenden COM Symbole automatisch mit der invertierten Kodierung $001111\ 1010$ bzw. einer positiven Disparität übertragen werden und sich so an der veränderten CRD durch ihre Vorgänger orientieren. Die beiden Marker $TimeA$ und $TimeB$ rahmen hierbei erneut die Übertragung von insgesamt 8 Byte bzw. 80-Bit ein.

5.3.7 Loopback Modus

Für den Test des *Loopback* muss gewährleistet werden, dass der PIPE IP-Core durch das Anlegen einer logischen 1 an die Schnittstelle *i_TxDetectRx* den SerDes in den *far-end PCS Loopback* überführt. Für die Simulation wird der PIPE IP-Core zunächst in den *Power State* P0 gebracht, da die Schnittstelle *i_TxDetectRx* nur in diesem Fall über diese entsprechende Funktion verfügt. Im Fall des *Power State* P1 würde diese eine *Receiver Detection* auslösen. Im Simulationsverlauf ist zu erkennen, dass nach Aktivierung des Kontrollsignals die FSM der PIPE Logic in den FSM Zustand P0_LOOP (0010) übergeht und die notwendige Kodierung für die Schnittstelle *loopback_i* des SerDes erzeugt (Abb. 5.13).

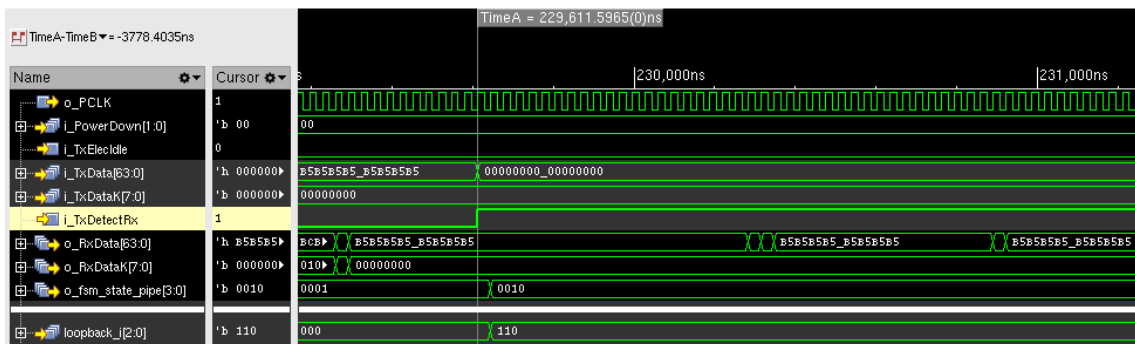


Abb. 5.13: Aktivierung des *far-end PCS Loopback*.

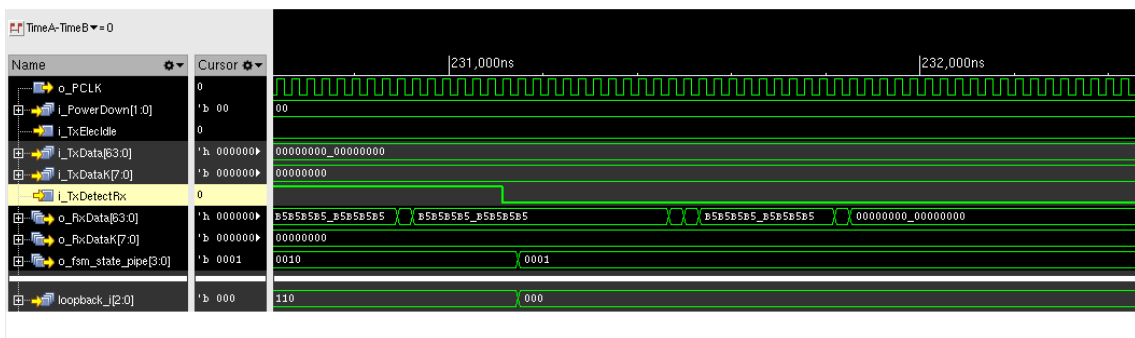


Abb. 5.14: Deaktivierung des *far-end PCS Loopback*.

Der SerDes tritt daraufhin in den *Loopback* Modus ein und beginnt die empfangenen Daten über den *Transmitter* zu versenden. Dies zeigt sich in der Simulation dadurch, dass an den Schnittstelle für die Empfangsdaten (*o_RxData*) nicht mehr die zu erwartenden Sendedaten (*i_TxData*) ausgegeben werden, sondern stattdessen

eine sich wiederholende Ausgabe der zuvor anliegenden Empfangsdaten auftritt. Die redundanten Daten bestehen dabei jeweils aus dem sich wiederholenden Datenbyte 0xB5 und einem kleinen Anteil mit dem Datenbyte 0x00. Letzteres ist aufgrund der gewählten Simulationsdarstellung jedoch nur als Datenwechsel ohne Wertangabe dargestellt. Die Deaktivierung des *Loopback* Modus führt zur entsprechenden Rückkehr der FSM in den Zustand P0_NORMAL (0001). Der SerDes wird dabei durch die entsprechende Kodierung an der Schnittstelle *loopback_i* zurück in den normalen Sendemodus versetzt (Abb. 5.14). Abschließend beginnt der *Transmitter* wieder mit der Übertragung der anliegenden Sendedaten an *i_TxData*.

5.3.8 Receiver Detection

Im Rahmen der *Receiver Detection* muss sich zeigen, ob der PIPE IP-Core im *Power State* P1 die notwendige Ansteuerung des SerDes durchführt und die entsprechenden Werte für die Statussignale *o_PhyStatus* und *o_RxStatus* erzeugt. Die Ansteuerung erfolgt über das Kontrollsignal *i_TxDetectRx* und wird von der PIPE FSM verarbeitet, welche sich für die *Receiver Detection* verantwortlich zeigt. Innerhalb der *Testbench* lässt sich die *Receiver Detection* zum gegenwärtigen Zeitpunkt zwar durchführen, hat jedoch immer ein negatives Ergebnis zur Folge, da kein *Receiver* detektiert wird. Die Gründe für dieses Ergebnis konnten nicht abschließend geklärt werden. Es ist jedoch denkbar, dass der Test der *Receiver Detection* im Rahmen der digitalen Simulation nicht durchgeführt werden kann. In diesem Zusammenhang ist auf die weitere Simulation mit der erweiterten *Testbench* verzichtet worden und auf die *Testbench* für die PIPE FSM zurückgegriffen worden, welche bereits während der Entwicklung zum Einsatz gekommen ist. Für den Test wird die IP-Core zunächst in den *Power State* P1 überführt (Abb. 5.15).

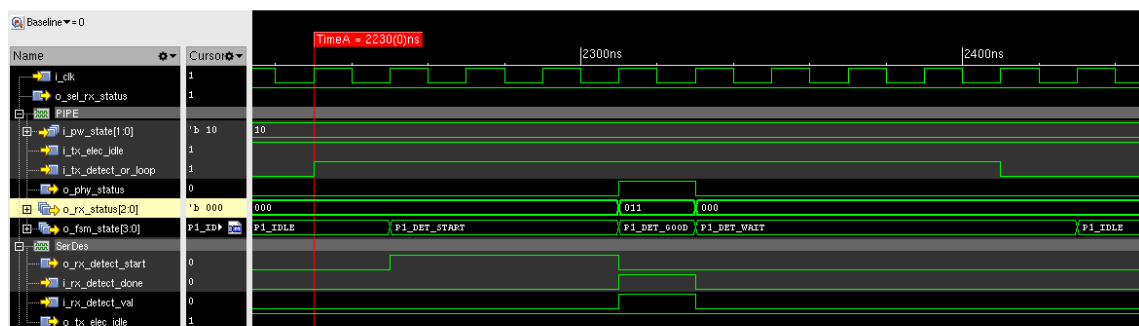


Abb. 5.15: Durchführung der *Receiver Detection* in *Power State* P1 (Teil 1)

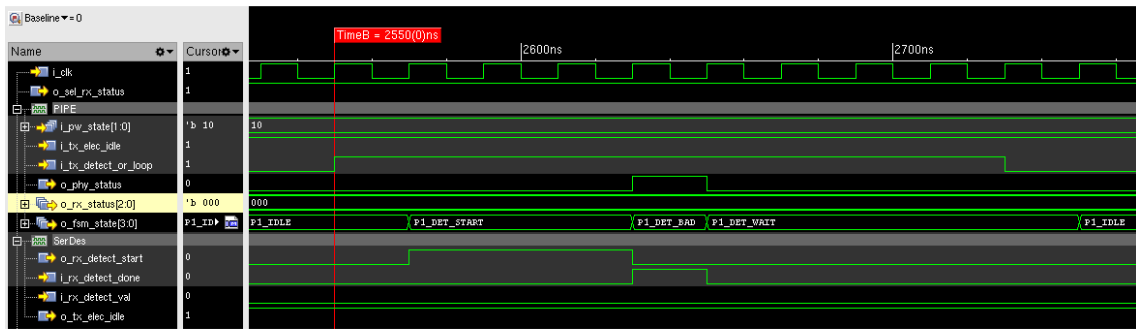


Abb. 5.16: Durchführung der *Receiver Detection* in *Power State P1* (Teil 2)

Anschließend wird das Kontrollsignal *i_TxDetectIdle*, welches mit dem Eingangsport *i_tx_detect_or_loop* der FSM verbunden ist, auf logisch 1 gezogen (*TimeA*). Es ist zu erkennen, dass die FSM in den Zustand *P1_DET_START* übergeht und die *Receiver Detection* des SerDes über die Schnittstelle *o_rx_detect_start* initialisiert. Nachdem die *Receiver Detection* abgeschlossen ist (*i_rx_detect_done*), zieht die FSM den *PhyStatus* für eine Taktperiode auf 1 und signalisiert über *RxStatus* die erfolgreiche Detektierung eines *Receivers* auf der Gegenseite (011). Die FSM befindet sich dabei im Zustand *P1_DET_GOOD* und geht anschließend in den Zustand *P1_DET_WAIT* über. In diesem Zustand verbleibt die FSM, bis das Kontrollsignal *i_TxDetectRx* durch die MAC wieder zurück auf 0 gezogen worden ist. Danach ist eine erneute *Receiver Detection* möglich. In gleicher Weise wird die *Receiver Detection* für den Fall getestet, dass kein *Receiver* vorhanden ist (Abb. 5.16).

6 Fazit und Ausblick

Im Rahmen dieser Masterthesis konnte ein strukturierter Einblick in die grundlegenden Konzepte und Komponenten der Künstlichen Intelligenz (KI) gegeben werden, wobei vor allem die Struktur und Funktion der Künstlichen Neuronalen Netze (KNN) im Fokus steht. Des Weiteren ist auf den Einsatz der FPGA-Technologie in der KI und die damit verbundene Implementierung der KNNs eingegangen worden. In der Entwicklung konnte der integrierte SerDes des GateMate™ FPGA erfolgreich um einen IP-Core zur Implementierung des *PHY Interface for the PCIe Architecture* (PIPE) erweitert werden. Der IP-Core unterstützt dabei nahezu alle grundlegenden Funktionen und Schnittstellen, welche in der PIPE Spezifikation [Int07] gefordert werden. Die Masterthesis umfasste vor der Entwicklung des IP-Cores, zunächst die Analyse und Aufbereitung der zur Verfügung gestellten *Design Files* (SerDes) sowie einer Testumgebung (*Testbench*) mit mehreren Testfällen (*Testcases*). Die Testumgebung konnte erfolgreich in den *Design Flow* von Cadence Incisive integriert werden und machte die Simulation mit SimVision möglich. Für den Test des IP-Cores, ist dieser als zusätzliches DUT in die Testumgebung integriert und durch einen eigenen *Testcase* (*testcase_pipe.v*) verifiziert worden. Aufgrund des fortgeschritten Projektverlaufs, konnte der IP-Core jedoch nur in der Version mit 64-Bit breitem Datenpfad verifiziert werden. Die Testumgebung selbst ist modular gestaltet, so dass auch weiterhin der alleinige Test des SerDes möglich ist. Während der Entwicklung hat sich gezeigt, dass die Systemarchitektur und die Funktionen des SerDes nicht immer eindeutig aus dessen Spezifikation [Rac20] hervorgehen. Daher musste die Funktionalität teilweise aus der Simulation extrahiert und dokumentiert werden. Die zukünftigen Entwicklungsschritte sollten zunächst den Test des IP-Cores mit geringeren Bitbreiten als 64-Bit umfassen. Dazu kann der eingesetzte *Testcase* modifiziert und für den Test der verschiedenen Bitbreiten modular gestaltet werden. Einige der eingesetzten Methoden innerhalb des *Testcase* sind dahingehend bereits vorbereitet worden. Des Weiteren muss geklärt werden, in wie fern die Implementierung der *Power States* P0s und P2 in den IP-Core möglich bzw. notwendig ist. Als Orientierung können hierbei der PIPE IP-Core von Lattice [Lat09] oder der XIO1100 PHY von Texas Instruments [Tex06] dienen. Für die Entwicklung eines vollständigen PCIe Soft Cores, stellt der nächste logische Schritt die Entwicklung der LTSSM als Kernkomponente der *Medium Access Control* (MAC) und Ausgangspunkt der Kontrollsignale dar.

Literaturverzeichnis

- [BAS12] BUDRUK, Ravi ; ANDERSON, Don ; SHANLEY, Tom ; INC, MindShare: *PCI Express Technology - Comprehensive Guide to Generations 1.x, 2.x, 3.0*. 1. Auflage. Cedar Park, TX 78613 : Mindshare, Inc., 2012. – ISBN 978-0-9836465-2-5
- [Col20] COLOGNE CHIP (Hrsg.): *GateMate FPGA Datasheet*. : Cologne Chip, 10 2020. (DS1001) . – Pre-Release Datasheet
- [Eat19a] EATON, Daniel: FPGAs beschleunigen KI-Inferenz bei SK Telecom. In: *Elektronikpraxis* (2019), 11, Nr. 21, S. 48–49
- [Eat19b] EATON, Daniel: Xilinx FPGAs Accelerate Artificial Intelligence Inferencing at SK Telecom. In: *eeNewsEmbedded* (2019), 9, S. 4–5
- [GBC16] GOODFELLOW, Ian ; BENGIO, Yoshua ; COURVILLE, Aaron: *Deep Learning*. 1. Auflage. MIT Press, 2016. – <http://www.deeplearningbook.org>
- [GBC18] GOODFELLOW, Ian ; BENGIO, Yoshua ; COURVILLE, Aaron: *Deep Learning: Das umfassende Handbuch*. 1. Auflage. MIT Press, 2018. – Übersetzung aus dem amerikanischen von Guido Lenz.
- [GFC18] GAGLIARDI, M. ; FUSELLA, E. ; CILARDO, A.: Improving Deep Learning with a customizable GPU-like FPGA-based accelerator. In: *2018 14th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*, 2018, S. 273–276
- [Hop06] HOPPE, Bernhard: *Verilog - Modellbildung für Synthese und Verifikation*. 1. Auflage. München : Oldenbourg Wissenschaftsverlag, 2006
- [Int07] INTEL CORPORATION (Hrsg.): *PHY Interface for the PCI Express Architecture*. Rev. 2.00. : Intel Corporation, 7 2007. – R
- [KB13] KESEL, Frank ; BARTHOLOMÄ, Ruben: *Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs - Einführung mit VHDL und SystemC*. 3. Auflage. München : Oldenbourg Wissenschaftsverlag, 2013

- [KBB⁺15] KRUSE, Rudolf ; BORGELT, Christian ; BRAUNE, Christian ; KLAWONN, Frank ; MOEWES, Christian ; STEINBRECHER, Matthias: *Computational Intelligence*. 2. Auflage. Wiesbaden : Springer Vieweg, 2015
- [KH20] KOITZ-HRISTOV, Roxane: *Onlinekurs: Machine Learning Grundlagen*. <https://de.linkedin.com/learning/machine-learning-grundlagen>, 2020. – [Online; Zugriff am 25.02.2021]
- [KS19] KREUTZER, Ralf T. ; SIRRENBURG, Marie: *Künstliche Intelligenz verstehen - Grundlagen – Use-Cases – unternehmenseigene KI-Journey*. 1. Auflage. Berlin Heidelberg New York : Springer-Verlag, 2019
- [Lat09] LATTICE (Hrsg.): *Lattice ECP2M PCS PIPE IP Core - User Guide*. ipug77_01.0. : Lattice, 3 2009
- [Led17] LEDÜC, Philipp: *Entwicklung einer DSP32c-kompatiblen Gleitkommaeinheit für die FPGA-basierte Implementierung*. Dortmund, 4 2017
- [LTA16] LACEY, Griffin ; TAYLOR, Graham W. ; AREIBI, Shawki: Deep Learning on FPGAs: Past, Present, and Future. In: *CoRR* abs/1602.04283 (2016). <http://arxiv.org/abs/1602.04283>
- [MF17] MURPHY, Cathal ; FU, Yao: Xilinx All Programmable Devices: a Superior Platform for Compute-Intensive Systems. (2017), June
- [OR06] OMONIDI, Amaos R. ; RAJAPASKE, Jagath C.: *FPGA Implementations of Neural Networks*. 1. Auflage. Dordrecht : Springer, 2006
- [PCI09] PCI-SIG (Hrsg.): *PCI Express Base Specification*. Rev. 2.1. www.pcisig.com: PCI-SIG, 3 2009
- [Rac20] RACYICS (Hrsg.): *SerDes Specification for Cologne Chip FPGA*. : RacyICs, 10 2020. – Rev. 1.2, Status: Release
- [SCYE17] SZE, V. ; CHEN, Y. ; YANG, T. ; EMER, J. S.: Efficient Processing of Deep Neural Networks: A Tutorial and Survey. In: *Proceedings of the IEEE* 105 (2017), Nr. 12, S. 2295–2329. <http://dx.doi.org/10.1109/JPROC.2017.2761740>. – DOI 10.1109/JPROC.2017.2761740

- [SN16] SCHULZ, Peter ; NAROSKA, Edwin: *Digitale Systeme mit FPGAs entwickeln*. 1. Auflage. Aachen : Elektor-Verlag, 2016
- [Str20] STROH, IRIS: *New FPGA architecture - With the most cost-effective FPGAs against the Big Player*. <https://www.elektroniknet.de/international/with-the-most-cost-effective-fpgas-against-the-big-player.176029.html>. Version: April 2020
- [Syn21] SYNOPSIS: *Power Management of PCIe PIPE Interface*. <https://blogs.synopsys.com/vip-central/2015/03/03/power-management-of-pipe-interface-part-i/>, 2021. – [Online; Zugriff am 03.05.2021]
- [Tex06] TEXAS INSTRUMENTS (Hrsg.): *XIO1100 - Data Manual*. Dallas, Texas 75265: Texas Instruments, 4 2006. (SLLS690C) . – Revised August 2011
- [Vor19] VORHEMUS, Christian: *Onlinekurs: Deep Learning: Grundlagen, Tools und Anwendungen*. <https://de.linkedin.com/learning/deep-learning-grundlagen-tools-und-anwendungen>, 2019. – [Online; Zugriff am 28.02.2021]
- [WG19] WONG, Alex ; GYORGY, Elod: Handgeschriebene Zahlen mit Künstlicher Intelligenz erkennen. In: *Elektronikpraxis* (2019), 11, Nr. 21, S. 50–52
- [WST03] WILEN, Adam H. ; SCHADE, Justin P. ; THORNBURG, Ron: *Introduction to PCI Express - A Hardware and Software Developer's Guide*. Hillsboro : Intel Press, 2003. – ISBN 978–0–970–28469–3
- [Wü21] WÜRTZ, Udo: *WIEGEHTKI.DE - DER deutschsprachige YouTube - Kanal rund um das Thema KI*. <https://www.udowuertz.de/ki>, 2021. – [Online; Zugriff am 13.02.2020]
- [Xil18] XILINX (Hrsg.): *7 Series FPGAs GTX/GTH Transceivers - User Guide*. v1.12.1. : Xilinx, 8 2018. (UG476)
- [Xil21] XILINX: *Xilinx Kintex UltraScale FPGA KCU1500 Acceleration Development Kit*. <https://www.xilinx.com/products/boards-and-kits/>

dk-u1-kcu1500-g.html#hardware, 2021. – [Online; Zugriff am 28.02.2021]

Anhang

A.1 8b/10b Dekodiertabellen

Data Byte Notation	Data Byte (hex)	Data Bits		CRD-		CRD+	
		HGF	EDCBA	abcdei	fghj	abcdei	fghj
D0.0	00	000	00000	100111	0100	011000	1011
D1.0	01	000	00001	011101	0100	100010	1011
D2.0	02	000	00010	101101	0100	010010	1011
D3.0	03	000	00011	110001	1011	110001	0100
D4.0	04	000	00100	110101	0100	001010	1011
D5.0	05	000	00101	101001	1011	101001	0100
D6.0	06	000	00110	011001	1011	011001	0100
D7.0	07	000	00111	111000	1011	000111	0100
D8.0	08	000	01000	111001	0100	000110	1011
D9.0	09	000	01001	100101	1011	100101	0100
D10.0	0A	000	01010	010101	1011	010101	0100
D11.0	0B	000	01011	110100	1011	110100	0100
D12.0	0C	000	01100	001101	1011	001101	0100
D13.0	0D	000	01101	101100	1011	101100	0100
D14.0	0E	000	01110	011100	1011	011100	0100
D15.0	0F	000	01111	010111	0100	101000	1011
D16.0	10	000	10000	011011	0100	100100	1011
D17.0	11	000	10001	100011	1011	100011	0100
D18.0	12	000	10010	010011	1011	010011	0100
D19.0	13	000	10011	110010	1011	110010	0100
D20.0	14	000	10100	001011	1011	001011	0100

Data Byte Notation	Data Byte (hex)	Data Bits		CRD-		CRD+	
		HGF	EDCBA	abcdei	fghj	abcdei	fghj
D21.0	15	000	10101	101010	1011	101010	0100
D22.0	16	000	10110	011010	1011	011010	0100
D23.0	17	000	10111	111010	0100	000101	1011
D24.0	18	000	11000	110011	0100	001100	1011
D25.0	19	000	11001	100110	1011	100110	0100
D26.0	1A	000	11010	010110	1011	010110	0100
D27.0	1B	000	11011	110110	0100	001001	1011
D28.0	1C	000	11100	001110	1011	001110	0100
D29.0	1D	000	11101	101110	0100	010001	1011
D30.0	1E	000	11110	011110	0100	100001	1011
D31.0	1F	000	11111	101011	0100	010100	1011
D0.1	20	001	00000	100111	1001	011000	1001
D1.1	21	001	00001	011101	1001	100010	1001
D2.1	22	001	00010	101101	1001	010010	1001
D3.1	23	001	00011	110001	1001	110001	1001
D4.1	24	001	00100	110101	1001	001010	1001
D5.1	25	001	00101	101001	1001	101001	1001
D6.1	26	001	00110	011001	1001	011001	1001
D7.1	27	001	00111	111000	1001	000111	1001
D8.1	28	001	01000	111001	1001	000110	1001
D9.1	29	001	01001	100101	1001	100101	1001
D10.1	2A	001	01010	010101	1001	010101	1001
D11.1	2B	001	01011	110100	1001	110100	1001
D12.1	2C	001	01100	001101	1001	001101	1001

Data Byte Notation	Data Byte (hex)	Data Bits		CRD-		CRD+	
		HGF	EDCBA	abcdei	fghj	abcdei	fghj
D13.1	2D	001	01101	101100	1001	101100	1001
D14.1	2E	001	01110	011100	1001	011100	1001
D15.1	2F	001	01111	010111	1001	101000	1001
D16.1	30	001	10000	011011	1001	100100	1001
D17.1	31	001	10001	100011	1001	100011	1001
D18.1	32	001	10010	010011	1001	010011	1001
D19.1	33	001	10011	110010	1001	110010	1001
D20.1	34	001	10100	001011	1001	001011	1001
D21.1	35	001	10101	101010	1001	101010	1001
D22.1	36	001	10110	011010	1001	011010	1001
D23.1	37	001	10111	111010	1001	000101	1001
D24.1	38	001	11000	110011	1001	001100	1001
D25.1	39	001	11001	100110	1001	100110	1001
D26.1	3A	001	11010	010110	1001	010110	1001
D27.1	3B	001	11011	110110	1001	001001	1001
D28.1	3C	001	11100	001110	1001	001110	1001
D29.1	3D	001	11101	101110	1001	010001	1001
D30.1	3E	001	11110	011110	1001	100001	1001
D31.1	3F	001	11111	101011	1001	010100	1001
D0.2	40	010	00000	100111	0101	011000	0101
D1.2	41	010	00001	011101	0101	100010	0101
D2.2	42	010	00010	101101	0101	010010	0101
D3.2	43	010	00011	110001	0101	110001	0101
D4.2	44	010	00100	110101	0101	001010	0101

Data Byte Notation	Data Byte (hex)	Data Bits		CRD-		CRD+	
		HGF	EDCBA	abcdei	fghj	abcdei	fghj
D5.2	45	010	00101	101001	0101	101001	0101
D6.2	46	010	00110	011001	0101	011001	0101
D7.2	47	010	00111	111000	0101	000111	0101
D8.2	48	010	01000	111001	0101	000110	0101
D9.2	49	010	01001	100101	0101	100101	0101
D10.2	4A	010	01010	010101	0101	010101	0101
D11.2	4B	010	01011	110100	0101	110100	0101
D12.2	4C	010	01100	001101	0101	001101	0101
D13.2	4D	010	01101	101100	0101	101100	0101
D14.2	4E	010	01110	011100	0101	011100	0101
D15.2	4F	010	01111	010111	0101	101000	0101
D16.2	50	010	10000	011011	0101	100100	0101
D17.2	51	010	10001	100011	0101	100011	0101
D18.2	52	010	10010	010011	0101	010011	0101
D19.2	53	010	10011	110010	0101	110010	0101
D20.2	54	010	10100	001011	0101	001011	0101
D21.2	55	010	10101	101010	0101	101010	0101
D22.2	56	010	10110	011010	0101	011010	0101
D23.2	57	010	10111	111010	0101	000101	0101
D24.2	58	010	11000	110011	0101	001100	0101
D25.2	59	010	11001	100110	0101	100110	0101
D26.2	5A	010	11010	010110	0101	010110	0101
D27.2	5B	010	11011	110110	0101	001001	0101
D28.2	5C	010	11100	001110	0101	001110	0101

Data Byte Notation	Data Byte (hex)	Data Bits		CRD-		CRD+	
		HGF	EDCBA	abcdei	fghj	abcdei	fghj
D29.2	5D	010	11101	101110	0101	010001	0101
D30.2	5E	010	11110	011110	0101	100001	0101
D31.2	5F	010	11111	101011	0101	010100	0101
D0.3	60	011	00000	100111	0011	011000	1100
D1.3	61	011	00001	011101	0011	100010	1100
D2.3	62	011	00010	101101	0011	010010	1100
D3.3	63	011	00011	110001	1100	110001	0011
D4.3	64	011	00100	110101	0011	001010	1100
D5.3	65	011	00101	101001	1100	101001	0011
D6.3	66	011	00110	011001	1100	011001	0011
D7.3	67	011	00111	111000	1100	000111	0011
D8.3	68	011	01000	111001	0011	000110	1100
D9.3	69	011	01001	100101	1100	100101	0011
D10.3	6A	011	01010	010101	1100	010101	0011
D11.3	6B	011	01011	110100	1100	110100	0011
D12.3	6C	011	01100	001101	1100	001101	0011
D13.3	6D	011	01101	101100	1100	101100	0011
D14.3	6E	011	01110	011100	1100	011100	0011
D15.3	6F	011	01111	010111	0011	101000	1100
D16.3	70	011	10000	011011	0011	100100	1100
D17.3	71	011	10001	100011	1100	100011	0011
D18.3	72	011	10010	010011	1100	010011	0011
D19.3	73	011	10011	110010	1100	110010	0011
D20.3	74	011	10100	001011	1100	001011	0011

Data Byte Notation	Data Byte (hex)	Data Bits		CRD-		CRD+	
		HGF	EDCBA	abcdei	fghj	abcdei	fghj
D21.3	75	011	10101	101010	1100	101010	0011
D22.3	76	011	10110	011010	1100	011010	0011
D23.3	77	011	10111	111010	0011	000101	1100
D24.3	78	011	11000	110011	0011	001100	1100
D25.3	79	011	11001	100110	1100	100110	0011
D26.3	7A	011	11010	010110	1100	010110	0011
D27.3	7B	011	11011	110110	0011	001001	1100
D28.3	7C	011	11100	001110	1100	001110	0011
D29.3	7D	011	11101	101110	0011	010001	1100
D30.3	7E	011	11110	011110	0011	100001	1100
D31.3	7F	011	11111	101011	0011	010100	1100
D0.4	80	100	00000	100111	0010	011000	1101
D1.4	81	100	00001	011101	0010	100010	1101
D2.4	82	100	00010	101101	0010	010010	1101
D3.4	83	100	00011	110001	1101	110001	0010
D4.4	84	100	00100	110101	0010	001010	1101
D5.4	85	100	00101	101001	1101	101001	0010
D6.4	86	100	00110	011001	1101	011001	0010
D7.4	87	100	00111	111000	1101	000111	0010
D8.4	88	100	01000	111001	0010	000110	1101
D9.4	89	100	01001	100101	1101	100101	0010
D10.4	8A	100	01010	010101	1101	010101	0010
D11.4	8B	100	01011	110100	1101	110100	0010
D12.4	8C	100	01100	001101	1101	001101	0010

Data Byte Notation	Data Byte (hex)	Data Bits		CRD-		CRD+	
		HGF	EDCBA	abcdei	fghj	abcdei	fghj
D13.4	8D	100	01101	101100	1101	101100	0010
D14.4	8E	100	01110	011100	1101	011100	0010
D15.4	8F	100	01111	010111	0010	101000	1101
D16.4	90	100	10000	011011	0010	100100	1101
D17.4	91	100	10001	100011	1101	100011	0010
D18.4	92	100	10010	010011	1101	010011	0010
D19.4	93	100	10011	110010	1101	110010	0010
D20.4	94	100	10100	001011	1101	001011	0010
D21.4	95	100	10101	101010	1101	101010	0010
D22.4	96	100	10110	011010	1101	011010	0010
D23.4	97	100	10111	111010	0010	000101	1101
D24.4	98	100	11000	110011	0010	001100	1101
D25.4	99	100	11001	100110	1101	100110	0010
D26.4	9A	100	11010	010110	1101	010110	0010
D27.4	9B	100	11011	110110	0010	001001	1101
D28.4	9C	100	11100	001110	1101	001110	0010
D29.4	9D	100	11101	101110	0010	010001	1101
D30.4	9E	100	11110	011110	0010	100001	1101
D31.4	9F	100	11111	101011	0010	010100	1101
D0.5	A0	101	00000	100111	1010	011000	1010
D1.5	A1	101	00001	011101	1010	100010	1010
D2.5	A2	101	00010	101101	1010	010010	1010
D3.5	A3	101	00011	110001	1010	110001	1010
D4.5	A4	101	00100	110101	1010	001010	1010

Data Byte Notation	Data Byte (hex)	Data Bits		CRD-		CRD+	
		HGF	EDCBA	abcdei	fghj	abcdei	fghj
D5.5	A5	101	00101	101001	1010	101001	1010
D6.5	A6	101	00110	011001	1010	011001	1010
D7.5	A7	101	00111	111000	1010	000111	1010
D8.5	A8	101	01000	111001	1010	000110	1010
D9.5	A9	101	01001	100101	1010	100101	1010
D10.5	AA	101	01010	010101	1010	010101	1010
D11.5	AB	101	01011	110100	1010	110100	1010
D12.5	AC	101	01100	001101	1010	001101	1010
D13.5	AD	101	01101	101100	1010	101100	1010
D14.5	AE	101	01110	011100	1010	011100	1010
D15.5	AF	101	01111	010111	1010	101000	1010
D16.5	B0	101	10000	011011	1010	100100	1010
D17.5	B1	101	10001	100011	1010	100011	1010
D18.5	B2	101	10010	010011	1010	010011	1010
D19.5	B3	101	10011	110010	1010	110010	1010
D20.5	B4	101	10100	001011	1010	001011	1010
D21.5	B5	101	10101	101010	1010	101010	1010
D22.5	B6	101	10110	011010	1010	011010	1010
D23.5	B7	101	10111	111010	1010	000101	1010
D24.5	B8	101	11000	110011	1010	001100	1010
D25.5	B9	101	11001	100110	1010	100110	1010
D26.5	BA	101	11010	010110	1010	010110	1010
D27.5	BB	101	11011	110110	1010	001001	1010
D28.5	BC	101	11100	001110	1010	001110	1010

Data Byte Notation	Data Byte (hex)	Data Bits		CRD-		CRD+	
		HGF	EDCBA	abcdei	fghj	abcdei	fghj
D29.5	BD	101	11101	101110	1010	010001	1010
D30.5	BE	101	11110	011110	1010	100001	1010
D31.5	BF	101	11111	101011	1010	010100	1010
D0.6	C0	110	00000	100111	0110	011000	0110
D1.6	C1	110	00001	011101	0110	100010	0110
D2.6	C2	110	00010	101101	0110	010010	0110
D3.6	C3	110	00011	110001	0110	110001	0110
D4.6	C4	110	00100	110101	0110	001010	0110
D5.6	C5	110	00101	101001	0110	101001	0110
D6.6	C6	110	00110	011001	0110	011001	0110
D7.6	C7	110	00111	111000	0110	000111	0110
D8.6	C8	110	01000	111001	0110	000110	0110
D9.6	C9	110	01001	100101	0110	100101	0110
D10.6	CA	110	01010	010101	0110	010101	0110
D11.6	CB	110	01011	110100	0110	110100	0110
D12.6	CC	110	01100	001101	0110	001101	0110
D13.6	CD	110	01101	101100	0110	101100	0110
D14.6	CE	110	01110	011100	0110	011100	0110
D15.6	CF	110	01111	010111	0110	101000	0110
D16.6	D0	110	10000	011011	0110	100100	0110
D17.6	D1	110	10001	100011	0110	100011	0110
D18.6	D2	110	10010	010011	0110	010011	0110
D19.6	D3	110	10011	110010	0110	110010	0110
D20.6	D4	110	10100	001011	0110	001011	0110

Data Byte Notation	Data Byte (hex)	Data Bits		CRD-		CRD+	
		HGF	EDCBA	abcdei	fghj	abcdei	fghj
D21.6	D5	110	10101	101010	0110	101010	0110
D22.6	D6	110	10110	011010	0110	011010	0110
D23.6	D7	110	10111	111010	0110	000101	0110
D24.6	D8	110	11000	110011	0110	001100	0110
D25.6	D9	110	11001	100110	0110	100110	0110
D26.6	DA	110	11010	010110	0110	010110	0110
D27.6	DB	110	11011	110110	0110	001001	0110
D28.6	DC	110	11100	001110	0110	001110	0110
D29.6	DD	110	11101	101110	0110	010001	0110
D30.6	DE	110	11110	011110	0110	100001	0110
D31.6	DF	110	11111	101011	0110	010100	0110
D0.7	E0	111	00000	100111	0001	011000	1110
D1.7	E1	111	00001	011101	0001	100010	1110
D2.7	E2	111	00010	101101	0001	010010	1110
D3.7	E3	111	00011	110001	1110	110001	0001
D4.7	E4	111	00100	110101	0001	001010	1110
D5.7	E5	111	00101	101001	1110	101001	0001
D6.7	E6	111	00110	011001	1110	011001	0001
D7.7	E7	111	00111	111000	1110	000111	0001
D8.7	E8	111	01000	111001	0001	000110	1110
D9.7	E9	111	01001	100101	1110	100101	0001
D10.7	EA	111	01010	010101	1110	010101	0001
D11.7	EB	111	01011	110100	1110	110100	1000
D12.7	EC	111	01100	001101	1110	001101	0001

Data Byte Notation	Data Byte (hex)	Data Bits		CRD-		CRD+	
		HGF	EDCBA	abcdei	fghj	abcdei	fghj
D13.7	ED	111	01101	101100	1110	101100	1000
D14.7	EE	111	01110	011100	1110	011100	1000
D15.7	EF	111	01111	010111	0001	101000	1110
D16.7	F0	111	10000	011011	0001	100100	1110
D17.7	F1	111	10001	100011	0111	100011	0001
D18.7	F2	111	10010	010011	0111	010011	0001
D19.7	F3	111	10011	110010	1110	110010	0001
D20.7	F4	111	10100	001011	0111	001011	0001
D21.7	F5	111	10101	101010	1110	101010	0001
D22.7	F6	111	10110	011010	1110	011010	0001
D23.7	F7	111	10111	111010	0001	000101	1110
D24.7	F8	111	11000	110011	0001	001100	1110
D25.7	F9	111	11001	100110	1110	100110	0001
D26.7	FA	111	11010	010110	1110	010110	0001
D27.7	FB	111	11011	110110	0001	001001	1110
D28.7	FC	111	11100	001110	1110	001110	0001
D29.7	FD	111	11101	101110	0001	010001	1110
D30.7	FE	111	11110	011110	0001	100001	1110
D31.7	FF	111	11111	101011	0001	010100	1110

Tabelle A.1: 8b/10b Dekodiertabelle für Datenbytes

Data Byte Notation	Data Byte (hex)	Data Bits		CRD-		CRD+	
		HGF	EDCBA	abcdei	fghj	abcdei	fghj
K28.0	1C	000	11100	001111	0100	110000	1011
K28.1	3C	001	11100	001111	1001	110000	0110
K28.2	5C	010	11100	001111	0101	110000	1010
K28.3	7C	011	11100	001111	0011	110000	1100
K28.4	9C	100	11100	001111	0010	110000	1101
K28.5	BC	101	11100	001111	1010	110000	0101
K28.6	DC	110	11100	001111	0110	110000	1001
K28.7	FC	111	11100	001111	1000	110000	0111
K23.7	F7	111	10111	111010	1000	000101	0111
K27.7	FB	111	11011	110110	1000	001001	0111
K29.7	FD	111	11101	101110	1000	010001	0111
K30.7	FE	111	11110	011110	1000	100001	0111

Tabelle A.2: 8b/10b Dekodiertabelle für Kontrollbytes

A.2 SerDes Schnittstellen

(siehe nächste Seite)

Scope of Application	Port	Type	Width	Description (Source Code)	Description/Comment
ADPLL	clk_ref_i	input	1	// Reference Clock (*)	(*) not accessible from FPGA fabric
	reset_n_i	input	1	// async. reset (*)	(*) not accessible from FPGA fabric
	testmode_i	input	1	// Testmode (*)	(*) not accessible from FPGA fabric
	scan_enable_i	input	1	// Scan Enable (*)	(*) not accessible from FPGA fabric
	pll_reset_i	input	1	// async. PLL Reset	Asynchronous ADPLL Reset
	clk_core_pll_o	output	1	// PLL clock output	parallel clock output, frequency divider depending on div_val_o which is set by the register file (see OLKCLK-GEN).
	reset_core_pll_n_o	output	1	// PLL reset output (*)	(*) not accessible from FPGA fabric
	refclk_sel_o	output	1	// Reference Clock Selection (*)	(*) not accessible from FPGA fabric
Register File Access	clk_reg_i	input	1	// Register File Clock	register file interface clock
	reset_reg_n_i	input	1	// async. Register File reset (*)	(*) not accessible from FPGA fabric
	regfile_we_i	input	1	// Register File write enable	register file write enable signal
	regfile_en_i	input	1	// Register File enable	register file enable signal
	regfile_addr_i	input	8	// Register File address	register file address signal
	regfile_di_i	input	16	// Register File data input	register file data input signal
	regfile_mask_i	input	16	// Register File data mask	register file data input mask signal
	regfile_do_o	output	16	// Register File data output	register file data output signal
	regfile_rdy_o	output	1	// Register File ready strobe	register file ready indicator signal
Configuration File	clk_cfg_i	input	1	// Configuration Clock (*)	(*) not accessible from FPGA fabric
	cfg_en_i	input	1	// Configuration enable (*)	(*) not accessible from FPGA fabric
	cfgfile_di_i	input	16	// Configuration data input (*)	(*) not accessible from FPGA fabric
	cfgfile_valid_i	input	1	// Configuration data valid (*)	(*) not accessible from FPGA fabric
	cfgfile_addr_i	input	8	// Configuration address (*)	(*) not accessible from FPGA fabric

Scope of Application	Port	Type	Width	Description (Source Code)	Description/Comment
	reset_cfg_i	input	1	// Configuration synch. reset (*)	(*) not accessible from FPGA fabric
	refclk_sel_o	output	1	// Reference Clock Selection (*)	(*) not accessible from FPGA fabric
	refclk_rterm_o	output	1	// Differential Ref. Clock Rterm enable (*)	(*) not accessible from FPGA fabric
Loopback	loopback_i	input	3	// Loopback Mode	000 normal operation 001 near-end PCS Loopback 010 near-end PMA Loopback 011 reserved 100 far-end PMA Loopback 101 reserved 110 far-end PCS Loopback 111 reserved
Tx Interface	clk_core_tx_i	input	1	// TX datapath clock	Tx datapath clock
	reset_core_tx_n_i	input	1	// async. TX reset (*)	(*) not accessible from FPGA fabric
	tx_reset_i	input	1	// async. TX datapath reset	asynchronous Tx Datapath reset
	tx_pcs_reset_i	input	1	// async. TX PCS reset	asynchronous Tx PCS reset
	tx_resetdone_o	output	1	// TX datapath reset done indicator	Tx reset state indicator: 0 TX reset not finished 1 TX reset completed
	tx_data_i	input	64	// TX data	Tx Data bus
	tx_char_dispmode_i	input	8	// TX per byte disparity mode	data bus extension for 80-bit interfaces when 8b/10b is not used
	tx_char_dispval_i	input	8	// TX per byte disparity value	data bus extension for 80-bit interfaces when 8b/10b is not used

Scope of Application	Port	Type	Width	Description (Source Code)	Description/Comment
Tx 8b/10b Encoder	tx_8b10b_bypass_i	input	8	// TX per byte 8B/10B bypass	one bit bypasses corresponding byte of 8B/10B encoder: 0 8B/10B encoder is used 1 8B/10B encoder is bypassed bit has no effect if the 8B/10B encoder is disabled via tx_8b10b_en_i input
	tx_8b10b_en_i	input	1	// TX 8B/10B enable	enable/disable 8B/10B encoder: 0 8B/10B encoder is bypassed 1 8B/10B encoder enabled
	tx_char_dispmode_i	input	8	// TX per byte disparity mode	Bitwise disparity overwriting if 8B/10B encoder is used. If 1 the corresponding tx_char_dispmode_i bit is forced for input disparity
	tx_char_dispval_i	input	8	// TX per byte disparity value	bitwise disparity value for disparity overwrite via tx_char_dispmode_i
	tx_char_is_k_i	input	8	// TX Control/Data word select	if high corresponding tx_data_i byte is interpreted as a valid K character
Tx Buffer	tx_buf_err_o	output	1	// TX buffer error	high for buffer overflow or underflow cleared by resetting tx_reset_i and reset_core_tx_n_i after 7 times the error has occurred, receiver recalibrates if Near-End_PMA Loop-back is enabled

Scope of Application	Port	Type	Width	Description (Source Code)	Description/Comment
Tx Pattern Generator	tx_prbs_sel_i	input	3	// TX PRBS Mode select	TX PRBS generator selection: 000 Bypass PRBS generator 001 PRBS-7 010 PRBS-15 011 PRBS-23 100 PRBS-31 101 RESERVED 110 2 UI square wave 111 20/40/80 UI square wave, depending on data path width
	tx_prbs_force_err_i	input	1	// TX PRBS Error injection	inject errors in the transmitted PRBS sequence
Tx Polarity Control	tx_polarity_i	input	1	// TX polarity inversion control	if high invert polarity of outgoing data stream
Tx Configurable Driver	tx_pma_reset_i	input	1	// async. TX PMA reset	asynchronous PMA reset
	tx_elec_idle_i	input	1	// TX electrical idle enable	enable TX electrical idle: 0 no electrical idle 1 electrical idle
	tx_powerdown_n_i	input	1	// async. TX power down	PMA power-state selection: 0 power down 1 power on (normal operation)
	tx_detect_rx_i	input	1	// RX detection enable	enable RX detection circuit: 0 Disable RX detection circuit 1 Start RX detection circuit
	rx_detect_done_o	output	1	// RX detection done indicator	RX detection circuit status: 0 RX detection circuit running 1 RX detection completed

Scope of Application	Port	Type	Width	Description (Source Code)	Description/Comment
	rx_present_o	output	1	// RX present indicator	RX detection result when circuit has completed operation: 0 RX not detected 1 RX detected
Rx Interface	clk_core_rx_i	input	1	// RX datapath clock	RX datapath clock
	clk_core_rx_o	output	1	// RX recovered clock output	RX recovered clock
	reset_core_rx_n_i	input	1	// async. RX reset (*)	(*) not accessible from FPGA fabric
	rx_reset_i	input	1	// async. RX datapath reset	asynchronous RX datapath reset
	rx_pcs_reset_i	input	1	// async. RX PCS reset	asynchronous RX PCS reset
	rx_resetdone_o	output	1	// RX datapath reset done indicator	RX reset state indicator: 0 RX reset not finished 1 RX reset completed
	rx_data_o	output	64	// RX data	RX data bus
	rx_disp_err_o	output	8	// RX 8B/10B disparity error	data bus extension for 80-bit interfaces, when 8B/10B decoder is not used
	rx_char_is_k_o	output	8	// RX Control/Data word	data bus extension for 80-bit interfaces, when 8B/10B decoder is not used
Rx Elastic Buffer	rx_buf_reset_i	input	1	// async. RX Buffer reset	resets and initializes the RX elastic buffer
	rx_buf_err_o	output	1	// RX elastic buffer error	high for buffer overflow or underflow cleared by resetting elastic buffer
	rx_8b10b_bypass_i	input	8	// RX per byte 8B/10B bypass	one bit bypasses one byte of 8B/10B decoder: 0 8B/10B encoder is used 1 8B/10B encoder is bypassed bit has no effect if the 8B/10B decoder is disabled

Scope of Application	Port	Type	Width	Description (Source Code)	Description/Comment
Rx 8b/10b Decoder	rx_8b10b_en_i	input	1	// RX 8B/10B enable	enable/disable 8B/10B decoder: 0 8B/10B decoder is bypassed 1 8B/10B decoder enabled
	rx_char_is_comma_o	output	8	// RX Comma detected	bit is high if corresponding rx_data_i byte is a valid comma character
	rx_char_is_k_o	output	8	// RX Control/Data word	high if corresponding rx_data_i byte is a valid K character. In case of disabled or bypasses 8B/10B decoder or decoder error indicated by the corresponding rx_not_in_table_i bit, this output is bit 8 of the non-decoded data
	rx_disp_err_o	output	8	// RX 8B/10B disparity error	high for corresponding rx_data_i byte disparity error. In case of disabled or bypassed 8B/10B decoder or decoder error indicated by the corresponding rx_not_in_table_i bit, this output is bit 9 of the non-decoded data
	rx_not_in_table_o	output	8	// RX 8B/10B decode error	high if corresponding rx_data_i byte is no valid character

Scope of Application	Port	Type	Width	Description (Source Code)	Description/Comment
Rx Pattern Checker	rx_prbs_sel_i	input	3	// RX PRBS Mode select	RX PRBS pattern checker selection: 000 PRBS checker disabled 001 PRBS-7 010 PRBS-15 011 PRBS-23 100 PRBS-31 101 reserved 110 reserved 111 reserved
	rx_prbs_cnt_reset_i	input	1	// RX PRBS Error counter reset	reset the PRBS checker error counter
	rx_prbs_err_o	output	1	// RX PRBS Error	RX PRBS error detector: 0 no error detected 1 error detected
Rx Byte and Word Alignment	rx_byte_is_aligned_o	output	1	// RX Comma aligned indicator	high if comma detection has aligned incoming commas to the byte boundaries successfully
	rx_byte_realign_o	output	1	// RX Comma realignment indicator	High if alignment circuit has detected an unaligned comma and realigned it to the new boundary. If signal turns from high to low the circuit has aligned to the new boundary and rx_byte_is_aligned_o will switch to 1
	rx_mcomma_align_i	input	1	// RX Minus Comma alignment enable	enable minus comma detection and alignment
	rx_pcomma_align_i	input	1	// RX Plus Comma alignment enable	enable plus comma detection and alignment

Scope of Application	Port	Type	Width	Description (Source Code)	Description/Comment
	rx_slide_i	input	1	// RX manual comma slide	if automatic comma alignment is not used port can be used for manual comma alignment. A single cycle strobe slides the incoming data stream by one bit depending on rx_slide_mode setting. rx_slide_i must be deasserted for 11 clk_core_rx_o cycles before next slide operation can be requested. rx_slide_i must be synchron with clk_core_rx_o
	rx_comma_detect_en_i	input	1	// RX Comma detect enable	enable/disable comma detection and alignment circuit: 0 alignment circuit is bypassed 1 use comma detection and alignment circuit
Rx Polarity Control	rx_polarity_i	input	1	// RX polarity inversion control	if high invert polarity of incoming data stream
Rx Clock Data Recovery	rx_cdr_reset_i	input	1	// async. RX CDR reset	asynchronous CDR reset
Rx Equalizer	rx_eqa_reset_i	input	1	// async. RX DFE reset	asynchronous DFE reset
Rx Analog Front End	rx_pma_reset_i	input	1	// async. RX PMA reset	asynchronous PMA reset
	rx_powerdown_n_i	input	1	// async. RX power down	PMA power-state selection: 0 power down 1 power on (normal operation)
	rx_en_ei_detector_i	input	1	// RX Elec. Idle detect enable	RX electrical idle detector control: 0 disable RX electrical idle detector 1 enable RX electrical idle detector
	rx_ei_en_o	output		// RX Elec. Idle status	RX electrical idle detector status: 0 electrical idle not detected 1 electrical idle condition detected
Scan Chain	scan_in_i	input	7	// Scan Input (*)	(*) not accessible from FPGA fabric
	scan_out_o	output	7	// Scan Output (*)	(*) not accessible from FPGA fabric

Scope of Application	Port	Type	Width	Description (Source Code)	Description/Comment
Serial_IO	RX_SERIO_P_B	input	1	// Serial Receiver positive (*)	(*) not accessible from FPGA fabric
	RX_SERIO_N_B	input	1	// Serial Receiver negative (*)	(*) not accessible from FPGA fabric
	TX_SERIO_P_B	output	1	// Serial Transmitter positive (*)	(*) not accessible from FPGA fabric
	TX_SERIO_N_B	output	1	// Serial Transmitter negative (*)	(*) not accessible from FPGA fabric
Termination Resistor	TERM_SERIO_O	output	1	// Calibration output (*)	(*) not accessible from FPGA fabric

Tabelle A.3: Schnittstellen des integrierten SerDes nach [Rac20]

A.3 Script zum Aufruf der Simulationsumgebung

```

1  #!/bin/bash
2
3  # Default timescale is set to 1ns/100fs (common lowest precision)
4
5  #Choose testcase
6
7  echo -n -e "Available Testcases:\n
8  [1] : PIPE Testacase (16 Bit) \n
9  [2] : PIPE Testacase (64 Bit) \n
10 [3] : SerDes Only Testcase \n
11 Type Number to choose Testcase: "
12
13 read VAR
14
15 if [[ $VAR -eq 1 ]]
16 then
17 echo -n -e "\nStarting PIPE Testacase (16 Bit) \n\n"
18 TESTCASE=PIPE
19 BITWIDTH=2
20 elif [[ $VAR -eq 2 ]]
21 then
22 echo -n -e "\nStarting PIPE Testacase (64 Bit) \n\n"
23 TESTCASE=PIPE
24 BITWIDTH=8
25 else
26 echo -n -e "\nStarting SerDes Only Testcase\n\n"
27 TESTCASE=SERDES
28 fi
29
30 #TESTCASE=SERDES # Only SerDes
31 #TESTCASE=PIPE # PIPE Interface
32
33 # If TESTCASE=PIPE, also choose Bitwidth of PIPE Iterface
34
35 #BITWIDTH=8
36 #BITWIDTH=2
37
38 #serdes file path
39 SERDES_FILE_PATH=/user/pledud/Desktop/ccfpga_serdes_mod/Serdes_files
40
41 #pipe file path
42 PIPE_FILE_PATH=/user/pledud/Desktop/ccfpga_serdes_mod/Pipe_files
43
44 module load Cadence/VIPCAT/11.30.057
45 cd /user/pledud/Desktop/ccfpga_serdes_mod/
46
47 #nclaunch
48 irun -64 -access +rw \
49 -define ${TESTCASE} \
50 -incdir ${SERDES_FILE_PATH} \
51 ${SERDES_FILE_PATH}/ccfpga_serdes_cpctr_bisc.v \
52 ${SERDES_FILE_PATH}/ccfpga_serdes_decode_8b10b_core.v \
53 ${SERDES_FILE_PATH}/ccfpga_serdes_decode_8b10b_mux.v \
54 ${SERDES_FILE_PATH}/ccfpga_serdes_decode_8b10b.v \
55 ${SERDES_FILE_PATH}/ccfpga_serdes_encode_8b10b_core.v \
56 ${SERDES_FILE_PATH}/ccfpga_serdes_encode_8b10b_mux.v \
57 ${SERDES_FILE_PATH}/ccfpga_serdes_encode_8b10b.v \
58 ${SERDES_FILE_PATH}/ccfpga_serdes_pads.v \
59 ${SERDES_FILE_PATH}/ccfpga_serdes_pd.v \
60 ${SERDES_FILE_PATH}/ccfpga_serdes_regfile.v \
61 ${SERDES_FILE_PATH}/ccfpga_serdes_rterm_calib.v \
62 ${SERDES_FILE_PATH}/ccfpga_serdes_rx_buffer.v \
63 ${SERDES_FILE_PATH}/ccfpga_serdes_rx_cdr_lockdetr.v \
64 ${SERDES_FILE_PATH}/ccfpga_serdes_rx_cdr_pfd.v \
65 ${SERDES_FILE_PATH}/ccfpga_serdes_rx_cdr_piloop.v \
66 ${SERDES_FILE_PATH}/ccfpga_serdes_rx_cdr_prdec.v \
67 ${SERDES_FILE_PATH}/ccfpga_serdes_rx_cdr.v \
68 ${SERDES_FILE_PATH}/ccfpga_serdes_rx_comma.v \
69 ${SERDES_FILE_PATH}/ccfpga_serdes_rx_cpctr_dec.v \
70 ${SERDES_FILE_PATH}/ccfpga_serdes_rx_cpctr_raw.v \
71 ${SERDES_FILE_PATH}/ccfpga_serdes_rx_ctrl.v \
72 ${SERDES_FILE_PATH}/ccfpga_serdes_rx_dbg.v \

```

```
73 $$SERDES_FILE_PATH/ccfpga_serdes_rx_eq_adapt_dec.v \  
74 $$SERDES_FILE_PATH/ccfpga_serdes_rx_eq_adapt_eyemeas.v \  
75 $$SERDES_FILE_PATH/ccfpga_serdes_rx_eq_adapt_fpd_hf.v \  
76 $$SERDES_FILE_PATH/ccfpga_serdes_rx_eq_adapt_fpd_lf.v \  
77 $$SERDES_FILE_PATH/ccfpga_serdes_rx_eq_adapt_intg.v \  
78 $$SERDES_FILE_PATH/ccfpga_serdes_rx_eq_adapt_lockdetr.v \  
79 $$SERDES_FILE_PATH/ccfpga_serdes_rx_eq_adapt_sel_fpdin.v \  
80 $$SERDES_FILE_PATH/ccfpga_serdes_rx_eq_adapt.v \  
81 $$SERDES_FILE_PATH/ccfpga_serdes_rx_if.v \  
82 $$SERDES_FILE_PATH/ccfpga_serdes_rx_prbs.v \  
83 $$SERDES_FILE_PATH/ccfpga_serdes_rx_rate.v \  
84 $$SERDES_FILE_PATH/ccfpga_serdes_rx_rst_core.v \  
85 $$SERDES_FILE_PATH/ccfpga_serdes_rx_rst.v \  
86 $$SERDES_FILE_PATH/ccfpga_serdes_rx.v \  
87 $$SERDES_FILE_PATH/ccfpga_serdes_tx_buffer.v \  
88 $$SERDES_FILE_PATH/ccfpga_serdes_tx_cm.v \  
89 $$SERDES_FILE_PATH/ccfpga_serdes_tx_if.v \  
90 $$SERDES_FILE_PATH/ccfpga_serdes_tx_prbs.v \  
91 $$SERDES_FILE_PATH/ccfpga_serdes_tx_rst_core.v \  
92 $$SERDES_FILE_PATH/ccfpga_serdes_tx_rst.v \  
93 $$SERDES_FILE_PATH/ccfpga_serdes_tx.v \  
94 $$SERDES_FILE_PATH/ccfpga_serdes.v \  
95 $$SERDES_FILE_PATH/common_and2.v \  
96 $$SERDES_FILE_PATH/common_buf.v \  
97 $$SERDES_FILE_PATH/common_clkbuf.v \  
98 $$SERDES_FILE_PATH/common_clkdiv_by_n.v \  
99 $$SERDES_FILE_PATH/common_clkgate.v \  
100 $$SERDES_FILE_PATH/common_clkinv.v \  
101 $$SERDES_FILE_PATH/common_clkmux.v \  
102 $$SERDES_FILE_PATH/common_or2.v \  
103 $$SERDES_FILE_PATH/common_reset_sync.v \  
104 $$SERDES_FILE_PATH/common_sync_pedge.v \  
105 $$SERDES_FILE_PATH/common_sync.v \  
106 $$SERDES_FILE_PATH/R_14T_SER_RVT_CBUF16.v \  
107 $$SERDES_FILE_PATH/R_14T_SER_RVT_CBUF8.v \  
108 $$SERDES_FILE_PATH/R_14T_SER_RVT_DFPQX2.v \  
109 $$SERDES_FILE_PATH/R_14T_SER_RVT_INVX2.v \  
110 $$SERDES_FILE_PATH/R_14T_SER_SLVT_CXOR2X1_2.v \  
111 $$SERDES_FILE_PATH/ri_adpll_bb_pfd.v \  
112 $$SERDES_FILE_PATH/ri_adpll_bisc.v \  
113 $$SERDES_FILE_PATH/ri_adpll_core_div_1_2.v \  
114 $$SERDES_FILE_PATH/ri_adpll_dco_en_sync.v \  
115 $$SERDES_FILE_PATH/ri_adpll_dco_lj_div_pfd_mclk.v \  
116 $$SERDES_FILE_PATH/ri_adpll_dco_lj_outdiv.v \  
117 $$SERDES_FILE_PATH/ri_adpll_dco_lj.v \  
118 $$SERDES_FILE_PATH/ri_adpll_div_ref_sync.v \  
119 $$SERDES_FILE_PATH/ri_adpll_loop_div_2345.v \  
120 $$SERDES_FILE_PATH/ri_adpll_olclk.v \  
121 $$SERDES_FILE_PATH/ri_adpll_pfdac.v \  
122 $$SERDES_FILE_PATH/ri_adpll_pfd_mon_dly.v \  
123 $$SERDES_FILE_PATH/ri_adpll_serdes_core_mclk.v \  
124 $$SERDES_FILE_PATH/ri_adpll_serdes_ctrl.v \  
125 $$SERDES_FILE_PATH/ri_adpll_serdes.v \  
126 $$SERDES_FILE_PATH/ri_adpll_serio_div_bb_pfd.v \  
127 $$SERDES_FILE_PATH/RI_SERDES_CALPAD.v \  
128 $$SERDES_FILE_PATH/RI_SERDES_CAPPAD.v \  
129 $$SERDES_FILE_PATH/ri_serdes_clkphase_buf_1.v \  
130 $$SERDES_FILE_PATH/ri_serdes_clkphase_buf_8_rx.v \  
131 $$SERDES_FILE_PATH/ri_serdes_clkphase_buf_8_tx.v \  
132 $$SERDES_FILE_PATH/ri_serdes_cm_detection.v \  
133 $$SERDES_FILE_PATH/ri_serdes_cm_generation.v \  
134 $$SERDES_FILE_PATH/ri_serdes_rterm_calib_comp.v \  
135 $$SERDES_FILE_PATH/ri_serdes_rtrns.v \  
136 $$SERDES_FILE_PATH/ri_serdes_rx_afe_peakamp.v \  
137 $$SERDES_FILE_PATH/ri_serdes_rx_afe_vgaamp.v \  
138 $$SERDES_FILE_PATH/ri_serdes_rx_ch.v \  
139 $$SERDES_FILE_PATH/ri_serdes_rx_datapath.v \  
140 $$SERDES_FILE_PATH/ri_serdes_rx_des10b.v \  
141 $$SERDES_FILE_PATH/ri_serdes_rx_descrtl.v \  
142 $$SERDES_FILE_PATH/ri_serdes_rx_edgepath.v \  
143 $$SERDES_FILE_PATH/ri_serdes_rx_eidetect.v \  
144 $$SERDES_FILE_PATH/ri_serdes_rx_monitorpath.v \  
145 $$SERDES_FILE_PATH/ri_serdes_rx_phgen.v \  
146 $$SERDES_FILE_PATH/ri_serdes_rx_phmon.v \  
147 $$SERDES_FILE_PATH/ri_serdes_rx_repbias.v \  
148 $$SERDES_FILE_PATH/ri_serdes_rx_rterm.v \  
149 $$SERDES_FILE_PATH/RI_SERDES_RX.v \  

```

```

150 $SERDES_FILE_PATH/ri_serdes_rx_vcmsel.v \
151 $SERDES_FILE_PATH/ri_serdes_tx_biasing.v \
152 $SERDES_FILE_PATH/ri_serdes_tx_ddr.v \
153 $SERDES_FILE_PATH/ri_serdes_tx_din.v \
154 $SERDES_FILE_PATH/ri_serdes_tx_driver.v \
155 $SERDES_FILE_PATH/ri_serdes_tx_logic.v \
156 $SERDES_FILE_PATH/ri_serdes_tx_rterm.v \
157 $SERDES_FILE_PATH/ri_serdes_tx_ser.v \
158 $SERDES_FILE_PATH/RI_SERDES_TX.v \
159 $SERDES_FILE_PATH/RM_GF28SLP_2P_512x20_c2.v \
160 $SERDES_FILE_PATH/RM_GF28SLP_2P_core_1cr.v \
161 $SERDES_FILE_PATH/SRAM_2P_behavioral.v \
162 $SERDES_FILE_PATH/thermo_decoder.v \
163 $PIPE_FILE_PATH/CC_PLL.v \
164 $PIPE_FILE_PATH/ccfpga_pipe_tx_demux.v \
165 $PIPE_FILE_PATH/ccfpga_pipe_rx_mux.v \
166 $PIPE_FILE_PATH/ccfpga_pipe_fsm_word_align.v \
167 $PIPE_FILE_PATH/ccfpga_pipe_fsm.v \
168 $PIPE_FILE_PATH/ccfpga_pipe_clk_counter.v \
169 $PIPE_FILE_PATH/ccfpga_pipe_logic.v \
170 $SERDES_FILE_PATH/tb_ccfpga_serdes.v \
171 -defparam tb_ccfpga_serdes.DATA_BYTES=${BITWIDTH} \
172 -top tb_ccfpga_serdes \
173 -timescale 1ns/100fs \
174 -gui
175 #-override_precision $SERDES_FILE_PATH/tb_ccfpga_serdes.v \
176 #-override_timescale $SERDES_FILE_PATH/tb_ccfpga_serdes.v \

```

A.4 Quellcode des PIPE IP-Cores

A.4.1 ccfpga_pipe_if.v

```

1 //////////////////////////////////////
2 //
3 // Interessengruppe fuer Mikroelektronik und Eingebettete Systeme (IMES)
4 // Fachhochschule Dortmund
5 //
6 // Development in cooperation with Cologne Chip AG
7 //
8 // Filename       : ccfpga_pipe_if.v
9 // Author        : Philipp Leduc
10 // Tool         :
11 // Description   : PIPE Interface Wrapper for SerDes Block of Gatemate FPGA.
12 //               Contains the instance for SerDes (Blackbox) and necessary PIPE Logic.
13 //               Additionally provides some Control Support Signals (e.g. Register Ports).
14 // Commentary   : Development according to PIPE Specification Version 2.0 by Intel.
15 //               Parameters for DATA_BYTES defines PIPE Interface width.
16 //               RX_PRBS_ERR_0 is not important and stays unconnected.
17 // Abbreviations : [i_] > input (port)
18 //               [o_] > output (port)
19 //               [s_] > signal
20 //               [n_] > low active
21 //
22 // Changelog:
23 // -----
24 // Version | Author           | Date           | Changes
25 // -----
26 // 1.0     | Leduc              | 05.06.2021    | pre released
27 // -----
28 //////////////////////////////////////
29
30 //resetall
31 `timescale 1ns/10ps
32 //default_nettype none
33
34 module ccfpga_pipe_if #(
35     parameter DATA_BYTES = 8,           // Configure width of PIPE datapath in Bytes

```

```

36 parameter DATA_WIDTH = DATA_BYTES*8,
37 parameter CCAG_CFG_PARAM = 1488'b0 // Parameter for Configuration
38 )(
39 // PIPE Ports
40 output wire o_PCLK,
41 input wire i_Reset,
42 input wire [ 1:0] i_PowerDown,
43 input wire i_TxDetectRx,
44 input wire i_TxElecIdle,
45 input wire [DATA_BYTES-1:0] i_TxCompliance,
46 input wire i_RxPolarity,
47 input wire o_RxValid,
48 input wire o_PhyStatus,
49 input wire o_RxElecIdle,
50 input wire [ 2:0] o_RxStatus,
51 input wire [DATA_WIDTH-1:0] i_TxData,
52 input wire [DATA_BYTES-1:0] i_TxDataK,
53 output wire [DATA_WIDTH:0] o_RxData,
54 output wire [DATA_BYTES:0] o_RxDataK,
55
56 // Register Ports
57 input wire i_clk_reg,
58 input wire i_regfile_we,
59 input wire i_regfile_en,
60 input wire i_regfile_addr,
61 input wire i_regfile_di,
62 input wire i_regfile_mask,
63 output wire o_regfile_do,
64 output wire o_regfile_rdy,
65
66 // Support Signals
67 output wire o_tx_buf_err,
68 output wire o_clk_core_rx_rec,
69 input wire i_rx_buf_reset,
70 output wire o_rx_buf_err,
71 output wire [3:0] o_fsm_state_pipe,
72 output wire [1:0] o_fsm_state_align,
73
74 output wire [DATA_BYTES-1:0] o_RxDataComma,
75 output wire [DATA_BYTES-1:0] o_RxDataDispErr,
76 output wire [DATA_BYTES-1:0] o_RxDataDecErr
77
78 );
79
80 // Signals
81
82 wire [63:0] s_tx_data;
83 wire [7:0] s_tx_char_is_k;
84 wire [63:0] s_rx_data;
85 wire [7:0] s_rx_char_is_k;
86 wire s_clk_reg;
87 wire s_regfile_we;
88 wire s_regfile_en;
89 wire [ 7:0] s_regfile_addr;
90 wire [15:0] s_regfile_mask;
91 wire [15:0] s_regfile_di;
92 wire [15:0] s_regfile_do;
93 wire s_regfile_rdy;
94 wire s_clk_core_tx;
95 wire s_clk_core_rx;
96 wire s_clk_core_rx_rec;
97 wire [ 2:0] s_loopback;
98 wire s_pll_reset;
99 wire s_clk_core_pll;
100 wire s_tx_reset;
101 wire s_tx_pcs_reset;
102 wire s_tx_pma_reset;
103 wire s_tx_elec_idle;
104 wire s_tx_detect_rx;
105 wire [ 2:0] s_tx_prbs_sel;
106 wire s_tx_prbs_force_err;
107 wire s_tx_powerdown_n;
108 wire s_tx_polarity;
109 wire s_tx_8b10b_en;
110 wire [ 7:0] s_tx_8b10b_bypass;
111 wire [ 7:0] s_tx_char_dispmode;
112 wire [ 7:0] s_tx_char_dispvai;

```

```

113 wire      s_tx_buf_err;
114 wire      s_tx_reset_done;
115 wire      s_rx_detect_done;
116 wire      s_rx_present;
117 wire      s_rx_reset;
118 wire      s_rx_pma_reset;
119 wire      s_rx_eqa_reset;
120 wire      s_rx_cdr_reset;
121 wire      s_rx_pcs_reset;
122 wire      s_rx_buf_reset;
123 wire [ 2:0] s_rx_prbs_sel;
124 wire      s_rx_prbs_cnt_reset;
125 wire      s_rx_powerdown_n;
126 wire      s_rx_en_ei_detector;
127 wire      s_rx_comma_detect_en;
128 wire      s_rx_slide;
129 wire      s_rx_polarity;
130 wire      s_rx_8b10b_en;
131 wire [ 7:0] s_rx_8b10b_bypass;
132 wire      s_rx_mcomma_align;
133 wire      s_rx_pcomma_align;
134 wire      s_rx_prbs_err;
135 wire [ 7:0] s_rx_char_is_comma;
136 wire [ 7:0] s_rx_not_in_table;
137 wire [ 7:0] s_rx_disp_err;
138 wire      s_rx_buf_err;
139 wire      s_rx_byte_is_aligned;
140 wire      s_rx_byte_realign;
141 wire      s_rx_reset_done;
142 wire      s_rx_ei_en;
143
144 // Register Interface to SerDes
145 assign i_clk_reg      = s_clk_reg;
146 assign i_regfile_we   = s_regfile_we;
147 assign i_regfile_en   = s_regfile_en;
148 assign i_regfile_addr = s_regfile_addr;
149 assign i_regfile_mask = s_regfile_mask;
150 assign i_regfile_di   = s_regfile_di;
151 assign o_regfile_do   = s_regfile_do;
152 assign o_regfile_rdy  = s_regfile_rdy;
153
154 // Instantiation of PIPE Logic
155 ccfpga_pipe_logic #(
156     .DATA_BYTES = ( DATA_BYTES )
157 )
158 i0_ccfpga_pipe_logic (
159     .o_PCLK          ( o_PCLK          ), // PIPE
160     .i_Reset         ( i_Reset         ),
161     .i_PowerDown     ( i_PowerDown     ),
162     .i_TxDetectRx    ( i_TxDetectRx    ),
163     .i_TxElecIdle    ( i_TxElecIdle    ),
164     .i_TxCompliance  ( i_TxCompliance  ),
165     .i_RxPolarity    ( i_RxPolarity    ),
166     .o_RxValid       ( o_RxValid       ),
167     .o_PhyStatus     ( o_PhyStatus     ),
168     .o_RxElecIdle    ( o_RxElecIdle    ),
169     .o_RxStatus      ( o_RxStatus      ),
170     .i_TxData        ( i_TxData        ),
171     .i_TxDataK       ( i_TxDataK       ),
172     .o_RxData        ( o_RxData        ),
173     .o_RxDataK       ( o_RxDataK       ),
174     .o_tx_buf_err    ( o_tx_buf_err    ), // CSS
175     .o_clk_core_rx_rec ( o_clk_core_rx_rec ),
176     .i_rx_buf_reset  ( i_rx_buf_reset  ),
177     .o_rx_buf_err    ( o_rx_buf_err    ),
178     .o_fsm_state_pipe ( o_fsm_state_pipe ),
179     .o_fsm_state_align ( o_fsm_state_align ),
180     .o_RxDataComma   ( o_RxDataComma   ),
181     .o_RxDataDispErr ( o_RxDataDispErr ),
182     .o_RxDataDecErr  ( o_RxDataDecErr  ),
183     .i_clk_core_pll   ( s_clk_core_pll   ), // SerDes
184     .i_clk_core_rx_rec ( s_clk_core_rx_rec ),
185     .o_clk_core_tx    ( s_clk_core_tx    ),
186     .o_clk_core_rx    ( s_clk_core_rx    ),
187     .o_tx_reset       ( s_tx_reset       ),
188     .o_rx_reset       ( s_rx_reset       ),
189     .i_tx_reset_done  ( s_tx_reset_done  ),

```

```

190     .i_rx_reset_done      ( s_rx_reset_done      ),
191     .o_loopback          ( s_loopback          ),
192     .o_rx_prbs_sel      ( s_rx_prbs_sel      ),
193     .o_tx_prbs_sel      ( s_tx_prbs_sel      ),
194     .o_tx_data          ( s_tx_data          ),
195     .o_tx_char_is_k     ( s_tx_char_is_k     ),
196     .o_tx_char_dispmode ( s_tx_char_dispmode ),
197     .o_tx_char_dispval  ( s_tx_char_dispval  ),
198     .o_tx_8b10b_bypass ( s_tx_8b10b_bypass ),
199     .o_tx_8b10b_en     ( s_tx_8b10b_en     ),
200     .o_tx_elec_idle     ( s_tx_elec_idle     ),
201     .o_tx_powerdown_n   ( s_tx_powerdown_n   ),
202     .o_tx_detect_rx     ( s_tx_detect_rx     ),
203     .i_rx_detect_done   ( s_rx_detect_done   ),
204     .i_rx_present       ( s_rx_present       ),
205     .i_rx_data          ( s_rx_data          ),
206     .i_rx_char_is_k     ( s_rx_char_is_k     ),
207     .i_rx_disp_err      ( s_rx_disp_err      ),
208     .i_rx_not_in_table  ( s_rx_not_in_table  ),
209     .i_rx_buf_err       ( s_rx_buf_err       ),
210     .o_rx_8b10b_bypass ( s_rx_8b10b_bypass ),
211     .o_rx_8b10b_en     ( s_rx_8b10b_en     ),
212     .i_rx_char_is_comma ( s_rx_char_is_comma ),
213     .i_ry_byte_is_aligned ( s_rx_byte_is_aligned ),
214     .i_rx_byte_realign  ( s_rx_byte_realign  ),
215     .o_rx_mcomma_align  ( s_rx_mcomma_align  ),
216     .o_rx_pcomma_align  ( s_rx_pcomma_align  ),
217     .o_rx_comma_detect_en ( s_rx_comma_detect_en ),
218     .o_rx_polarity      ( s_rx_polarity      ),
219     .o_rx_powerdown_n   ( s_rx_powerdown_n   ),
220     .o_rx_en_ei_detector ( s_rx_en_ei_detector ),
221     .i_rx_ei_en         ( s_rx_ei_en         ),
222     .o_pll_reset        ( s_pll_reset        ),
223     .o_tx_pcs_reset     ( s_tx_pcs_reset     ),
224     .o_tx_pma_reset     ( s_tx_pma_reset     ),
225     .o_rx_pcs_reset     ( s_rx_pcs_reset     ),
226     .o_rx_pma_reset     ( s_rx_pma_reset     ),
227     .o_rx_cdr_reset     ( s_rx_cdr_reset     ),
228     .o_rx_eqa_reset     ( s_rx_eqa_reset     ),
229     .o_rx_prbs_cnt_reset ( s_rx_prbs_cnt_reset ),
230     .o_tx_prbs_force_err ( s_tx_prbs_force_err ),
231     .o_rx_buf_reset     ( s_rx_buf_reset     ),
232     .i_tx_buf_err       ( s_tx_buf_err       ),
233     .o_tx_polarity      ( s_tx_polarity      ),
234     .o_rx_slide         ( s_rx_slide         ),
235 );
236
237 // Instantiation of FPGA SERDES Block
238 CC_SERDES #(
239     CCAG_CFG_PARAM ( CCAG_CFG_PARAM ) // Configuration
240 )
241 i_cc_serdes (
242     .CLK_REG_I      ( s_clk_reg      ),
243     .CLK_CORE_TX_I ( s_clk_core_tx  ),
244     .CLK_CORE_RX_I ( s_clk_core_rx  ),
245     .CLK_CORE_RX_O ( s_clk_core_rx_rec ),
246     .LOOPBACK_I    ( s_loopback     ),
247     .PLL_RESET_I   ( s_pll_reset     ),
248     .CLK_CORE_PLL_0 ( s_clk_core_pll ),
249     .REGFILE_WE_I  ( s_regfile_we    ),
250     .REGFILE_EN_I  ( s_regfile_en    ),
251     .REGFILE_ADDR_I ( s_regfile_addr  ),
252     .REGFILE_MASK_I ( s_regfile_mask ),
253     .REGFILE_DI_I  ( s_regfile_di    ),
254     .REGFILE_DO_0  ( s_regfile_do    ),
255     .REGFILE_RDY_0 ( s_regfile_rdy   ),
256     .TX_RESET_I    ( s_tx_reset      ),
257     .TX_PCS_RESET_I ( s_tx_pcs_reset  ),
258     .TX_PMA_RESET_I ( s_tx_pma_reset  ),
259     .TX_ELEC_IDLE_I ( s_tx_elec_idle  ),
260     .TX_DETECT_RX_I ( s_tx_detect_rx  ),
261     .TX_PRBS_SEL_I ( s_tx_prbs_sel   ),
262     .TX_PRBS_FORCE_ERR_I ( s_tx_prbs_force_err ),
263     .TX_POWERDOWN_N_I ( s_tx_powerdown_n ),
264     .TX_DATA_I     ( s_tx_data       ),
265     .TX_CHAR_IS_K_I ( s_tx_char_is_k  ),
266     .TX_POLARITY_I ( s_tx_polarity   ),

```

```

267     .TX_8B10B_EN_I      ( s_tx_8b10b_en      ),
268     .TX_8B10B_BYPASS_I ( s_tx_8b10b_bypass ),
269     .TX_CHAR_DISPVAL_I ( s_tx_char_dispval ),
270     .TX_CHAR_DISPMODE_I ( s_tx_char_dispmode ),
271     .TX_BUF_ERR_0      ( s_tx_buf_err      ),
272     .TX_RESETDONE_0    ( s_tx_reset_done    ),
273     .RX_DETECT_DONE_0  ( s_rx_detect_done  ),
274     .RX_PRESENT_0     ( s_rx_present     ),
275     .RX_RESET_I       ( s_rx_reset       ),
276     .RX_PMA_RESET_I   ( s_rx_pma_reset   ),
277     .RX_EQA_RESET_I   ( s_rx_eqa_reset   ),
278     .RX_CDR_RESET_I   ( s_rx_cdr_reset   ),
279     .RX_PCS_RESET_I   ( s_rx_pcs_reset   ),
280     .RX_BUF_RESET_I   ( s_rx_buf_reset   ),
281     .RX_PRBS_SEL_I    ( s_rx_prbs_sel    ),
282     .RX_PRBS_CNT_RESET_I ( s_rx_prbs_cnt_reset ),
283     .RX_POWERDOWN_N_I ( s_rx_powerdown_n ),
284     .RX_EN_EI_DETECTOR_I ( s_rx_en_ei_detector ),
285     .RX_COMMA_DETECT_EN_I ( s_rx_comma_detect_en ),
286     .RX_SLIDE_I       ( s_rx_slide       ),
287     .RX_POLARITY_I    ( s_rx_polarity    ),
288     .RX_8B10B_EN_I    ( s_rx_8b10b_en    ),
289     .RX_8B10B_BYPASS_I ( s_rx_8b10b_bypass ),
290     .RX_MCOMMA_ALIGN_I ( s_rx_mcomma_align ),
291     .RX_PCOMMA_ALIGN_I ( s_rx_pcomma_align ),
292     .RX_PRBS_ERR_0    ( s_rx_prbs_err    ), // unconnected
293     .RX_DATA_0        ( s_rx_data        ),
294     .RX_CHAR_IS_K_0   ( s_rx_char_is_k   ),
295     .RX_CHAR_IS_COMMA_0 ( s_rx_char_is_comma ),
296     .RX_NOT_IN_TABLE_0 ( s_rx_not_in_table ),
297     .RX_DISP_ERR_0    ( s_rx_disp_err    ),
298     .RX_BUF_ERR_0     ( s_rx_buf_err     ),
299     .RX_BYTE_IS_ALIGNED_0 ( s_rx_byte_is_aligned ),
300     .RX_BYTE_REALIGN_0 ( s_rx_byte_realign ),
301     .RX_RESETDONE_0   ( s_rx_reset_done   ),
302     .RX_EI_EN_0       ( s_rx_ei_en       )
303 );
304
305 endmodule

```

Quellcode A.1: Verilog Modul ccfpga_pipe_if

A.4.2 ccfpga_pipe_logic.v

```

1  //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
2  //
3  // Interessengruppe fuer Mikroelektronik und Eingebettete Systeme (IMES)
4  // Fachhochschule Dortmund
5  //
6  // Development in cooperation with Cologne Chip AG
7  //
8  // Filename      : ccfpga_pipe_logic.v
9  // Author        : Philipp Leduc
10 // Tool          :
11 // Description   : PIPE Interface Logic for SerDes Block of GateMate FPGA.
12 // Commentary   : Development according to PIPE Specification Version 2.0 by Intel.
13 //              : DATA_BYTES has to match 1 (8-Bit), 2 (16-Bit) 4 (32-Bit) or 8 (64-Bit).
14 //              : Use of the PIPE Interface requires specific Config-Fields to be set (see Doc).
15 //              : Tx Compliance:
16 //              : The use of the Tx Compliance Port differs from the description in the intel spec.
17 //              : Port width is equal to number of bytes, which can be set individually. In case of
18 //              : 16-Bit PIPE Interface the Tx Compliance port has a width of 2-Bit. The upper Bit
19 //              : can be tied to Zero and the lower Bit can be used as stated in the intel spec.
20 //
21 // Abbreviations : [i_] > input (port)
22 //                [o_] > output (port)
23 //                [s_] > signal
24 //                [n_] > low active
25 //

```



```

26 // Changelog:
27 // -----
28 // Version | Author          | Date       | Changes
29 // -----
30 // 1.0     | Leduc       | 05.06.2021 | released
31 // -----
32 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
33
34 // 'resetall
35 'timescale 1ns/10ps
36 // 'default_nettype none
37
38 module ccfpga_pipe_logic #(
39     parameter DATA_BYTES = 8,           // Width of PIPE Datapath in Bytes
40     parameter DATA_WIDTH = DATA_BYTES*8 // Do not set independet from Bytes
41 )
42 (
43     // ----- PIPE INTERFACE PORTS -----
44     // External
45     output wire          o_PCLK,         // PCLK (user side)
46
47     // Command
48     input wire           i_Reset,        // Async. Reset for Transceiver (Tx/Rx)
49     input wire [ 1:0]    i_PowerDown,    // Power States (P0 - P2)
50     input wire           i_TxDetectRx,   // Receiver Detection (P0) or Loopback (P1)
51     input wire           i_TxElecIdle,   // Tx Electrical Idle, Valid Data (P0) or Beacon (P2)
52     input wire [DATA_BYTES-1:0] i_TxCompliance, // Tx negative Disparity LSB (Compliance Pattern)
53     //input wire          i_TxSwing,     // Tx Voltage Swing Level [Optional by Spec]
54     input wire           i_RxPolarity,   // Rx Polarity Inversion
55
56     // Status
57     output wire          o_RxValid,      // Symbol Lock and Valid Data on RxData and RxDataK
58     output wire          o_PhyStatus,    // Status of several PHY functions (Transition)
59     output wire          o_RxElecIdle,   // Rx Detection of Electrical Idle
60     output wire [ 2:0]   o_RxStatus,     // Receiver Status and Received Data Status
61
62     // Transmit Data
63     input wire [DATA_WIDTH-1:0] i_TxData, // Tx Data
64     input wire [DATA_BYTES-1:0] i_TxDataK, // Tx K Data
65
66     // Receive Data
67     output wire [DATA_WIDTH-1:0] o_RxData, // Rx Data
68     output wire [DATA_BYTES-1:0] o_RxDataK, // Rx K Data
69
70     // ----- CSS INTERFACE PORTS -----
71
72     output wire          o_tx_buf_err,    // Tx Buffer Error
73     output wire          o_clk_core_rx_rec, // Rx Recovered Clock
74     input wire           i_rx_buf_reset,  // Rx Buffer Reset
75     output wire          o_rx_buf_err,    // Rx Buffer Error
76     output wire [3:0]    o_fsm_state_pipe, // State of PIPE FSM
77     output wire [1:0]    o_fsm_state_align, // State of Align FSM
78
79     output wire [DATA_BYTES-1:0] o_RxDataComma, // Rx Byte Comma Indication
80     output wire [DATA_BYTES-1:0] o_RxDataDispErr, // Rx Byte Disparity Error Indication
81     output wire [DATA_BYTES-1:0] o_RxDataDecErr, // Rx Byte Decode Error Indication
82     //output wire          o_TxIdleEntry,     // Tx Electrical Idle Counter Flag
83
84     // ----- CCAQ SERDES PORTS -----
85     // info: port directions seen from PIPE interface side
86
87     // Clock
88     input wire           i_clk_core_pll,  // ADPLL
89     input wire           i_clk_core_rx_rec, // Recovered Clock from Rx
90     output wire          o_clk_core_tx,   // Transmitter (Tx)
91     output wire          o_clk_core_rx,   // Receiver (Rx)
92
93     // Reset
94     output wire          o_pll_reset,     // ADPLL
95     output wire          o_tx_reset,      // Tx Reset
96     input wire           i_tx_reset_done, // Tx Reset Status
97     output wire          o_rx_reset,      // Rx Reset
98     input wire           i_rx_reset_done, // Rx Reset Status
99
100    output wire          o_tx_pcs_reset,   // Tx PCS Reset
101    output wire          o_tx_pma_reset,   // Tx PMA Reset
102

```

```

103 output wire o_rx_pcs_reset, // Rx PCS Reset
104 output wire o_rx_pma_reset, // Rx PMA Reset
105 output wire o_rx_cdr_reset, // Rx CDR Reset
106 output wire o_rx_eqa_reset, // Rx DFE Reset
107
108 // Power Control
109 output wire o_tx_powerdown_n, // Tx Power Control
110 output wire o_rx_powerdown_n, // Rx Power Control
111
112 // Loopback and PRBS
113 output wire [2:0] o_loopback, // Loopback Control
114 output wire [2:0] o_rx_prbs_sel, // Rx PRBS Checker Mode
115 output wire o_rx_prbs_cnt_reset, // Rx PRBS Error Counter Reset
116 output wire [2:0] o_tx_prbs_sel, // Tx PRBS Generator Mode
117 output wire o_tx_prbs_force_err, // Tx PRBS Error Injection
118
119 // Buffer
120 output wire o_rx_buf_reset, // Rx Buffer Reset
121 input wire i_rx_buf_err, // Rx Buffer Error (Over- or Underflow)
122 input wire i_tx_buf_err, // Tx Buffer Error (Over- or Underflow)
123
124 // Tx-Datapath
125 output wire [63:0] o_tx_data, // Tx Data
126 output wire [ 7:0] o_tx_char_is_k, // Tx K-Data
127 output wire [ 7:0] o_tx_char_dispmode, // Tx Disparity Enable
128 output wire [ 7:0] o_tx_char_dispval, // Tx Disparity Values (0:neg, 1:pos)
129 output wire o_tx_8b10b_en, // Tx 8b/10b Decoder Enable
130 output wire [ 7:0] o_tx_8b10b_bypass, // Tx 8b/10b Decoder Bypass per Bit
131
132 output wire o_tx_polarity, // Tx Polarity Control
133
134 output wire o_tx_elec_idle, // Tx Electrical Idle Control
135
136 output wire o_tx_detect_rx, // Tx Receiver Detection Control
137 input wire i_rx_detect_done, // Tx Receiver Detection Status
138 input wire i_rx_present, // Tx Receiver Detection Response (1: Present)
139
140 // Rx-Datapath
141 input wire [63:0] i_rx_data, // Rx Data
142 input wire [ 7:0] i_rx_char_is_k, // Rx K Data
143 input wire [ 7:0] i_rx_char_is_comma, // Rx COM Data
144 input wire [ 7:0] i_rx_disp_err, // Rx Disparity Error
145 input wire [ 7:0] i_rx_not_in_table, // Rx Decoding Error
146 output wire o_rx_8b10b_en, // Rx 8b/10b Decoder Enable
147 output wire [ 7:0] o_rx_8b10b_bypass, // Rx 8b/10b Decoder Bypass per Bit
148
149 input wire i_rx_byte_is_aligned, // Rx Byte Alignment Status (Not used atm.)
150 input wire i_rx_byte_realign, // Rx Byte Realignment Status
151 output wire o_rx_mcomma_align, // Rx Byte Alignment Control COM (neg. CRD)
152 output wire o_rx_pcomma_align, // Rx Byte Alignment Control COM (pos. CRD)
153 output wire o_rx_comma_detect_en, // Rx Byte Alignment Enable
154 output wire o_rx_slide, // Rx Manual Comma Slide
155
156 output wire o_rx_polarity, // Rx Polarity Control
157
158 output wire o_rx_en_ei_detector, // Rx Electrical Idle Detection Enable
159 input wire i_rx_ei_en // Rx Electrical Idle Detection Response
160 );
161
162 wire s_reset, s_clk;
163 wire s_sel_rx_status; // Mux Select Signal for RxStatus
164 wire [2:0] s_rx_status_dp; // RxStatus Signal from Rx Datapath
165 wire [2:0] s_rx_status_fsm; // RxStatus Signal from FSM (Rx Detection)
166 wire s_loopback_fsm;
167 wire s_rx_val_from_fsm;
168 wire s_tx_idle_flag;
169
170
171 // Constant Port Value Assignments
172
173 assign o_tx_reset = 1'b0;
174 assign o_tx_pcs_reset = 1'b0;
175 assign o_tx_pma_reset = 1'b0;
176
177 assign o_rx_reset = 1'b0;
178 assign o_rx_pcs_reset = 1'b0;
179 assign o_rx_pma_reset = 1'b0;

```

```

180 assign o_rx_cdr_reset = 1'b0;
181 assign o_rx_eqa_reset = 1'b0;
182
183 assign o_tx_8b10b_bypass = 8'b0000_0000;
184 assign o_rx_8b10b_bypass = 8'b0000_0000;
185 assign o_tx_8b10b_en = 1'b1;
186 assign o_rx_8b10b_en = 1'b1;
187
188 assign o_tx_prbs_sel = 3'b000;
189 assign o_tx_prbs_force_err = 1'b0;
190 assign o_rx_prbs_sel = 3'b000;
191 assign o_rx_prbs_cnt_reset = 1'b0;
192
193 assign o_rx_slide = 1'b0;
194 assign o_rx_comma_detect_en = 1'b1;
195
196 assign o_tx_polarity = 1'b0;
197
198 assign o_tx_powerdown_n = 1'b1;
199 assign o_rx_powerdown_n = 1'b1;
200
201 assign o_rx_en_ei_detector = 1'b1;
202
203 assign o_tx_char_dispvall = 8'b0000_0000;
204
205 // Clock
206 assign o_PCLK = s_clk;
207
208 assign o_clk_core_tx = i_clk_core_pll;
209 assign o_clk_core_rx = i_clk_core_pll;
210 assign o_clk_core_rx_rec = i_clk_core_rx_rec;
211
212
213 // Reset Logic
214 assign s_reset = ~i_Reset;
215 assign o_pll_reset = s_reset;
216
217
218 // RxStatus Mux
219 assign o_RxStatus = s_sel_rx_status ? s_rx_status_fsm : s_rx_status_dp;
220
221
222 // Loopback
223 assign o_loopback [0] = 1'b0;
224 assign o_loopback [1] = s_loopback_fsm;
225 assign o_loopback [2] = s_loopback_fsm;
226
227
228 // Rx Valid
229 assign o_RxValid = s_rx_val_from_fsm & i_rx_byte_is_aligned;
230
231
232 // Rx Electrical Idle
233 assign o_RxElecIdle = i_rx_ei_en;
234
235
236 // Buffer Support Signals
237 assign o_tx_buf_err = i_tx_buf_err;
238 assign o_rx_buf_reset = i_rx_buf_reset;
239 assign o_rx_buf_err = i_rx_buf_err;
240
241 // Rx Polarity
242 assign o_rx_polarity = i_RxPolarity; // Setting not FSH related atm.
243
244
245 // FSM PIPE
246 ccfpga_pipe_fsm fsm_inst_0 (
247     .i_clk          ( s_clk          ),
248     .i_reset        ( s_reset        ),
249     .o_fsm_state    ( o_fsm_state_pipe ),
250     .i_pw_state     ( i_PowerDown    ),
251     .i_tx_elec_idle ( i_TxElecIdle   ),
252     .i_tx_detect_or_loop ( i_TxDetectRx ),
253     .o_phy_status   ( o_PhyStatus    ),
254     .o_rx_status    ( s_rx_status_fsm ),
255     .i_reset_done   ( s_reset_done   ),
256     .i_rx_detect_done ( i_rx_detect_done ),

```

```

257     .i_rx_detect_val    ( i_rx_present    ),
258     .o_rx_detect_start ( o_tx_detect_rx  ),
259     .o_sel_rx_status   ( s_sel_rx_status ),
260     .o_pcs_loopback    ( s_loopback_fsm  ),
261     .o_tx_elec_idle    ( o_tx_elec_idle  ),
262     .o_counter_reset   ( s_tx_idle_cnt_rst ),
263     .i_count_flag      ( s_tx_idle_flag  ),
264     .o_enable_align    ( s_enable_align  ) // Activate Word Alignment (FSM)
265 );
266
267
268 // Clock Counter for Tx Idle
269 ccfpga_pipe_clk_counter #(
270     .BIT_WIDTH ( 3 ),
271     .CLK_NUMB  ( 2 )
272 )
273 clk_count_inst_pipe_fsm (
274     .i_clk ( i_clk_core_pll ),
275     .i_reset ( s_tx_idle_cnt_rst ),
276     .o_flag ( s_tx_idle_flag )
277 );
278
279
280 // FSM Word Alignment
281 ccfpga_pipe_fsm_word_align fsm_inst_1 (
282     .i_clk ( i_clk_core_pll ),
283     .i_reset ( s_reset ),
284     .i_enable ( s_enable_align ),
285     .o_fsm_state ( o_fsm_state_align ),
286     .o_rx_valid ( s_rx_val_from_fsm ),
287     .o_counter_reset ( s_count_reset_align ),
288     .i_count_flag ( s_count_flag ),
289     .o_mcomma_align ( o_rx_mcomma_align ),
290     .o_pcomma_align ( o_rx_pcomma_align ),
291     .i_byte_locked ( i_rx_byte_is_aligned )
292 );
293
294
295 // Clock Counter for Alignment
296 ccfpga_pipe_clk_counter #(
297     .BIT_WIDTH ( 4 ),
298     .CLK_NUMB  ( 7 )
299 )
300 clk_count_inst_align (
301     .i_clk ( i_clk_core_pll ),
302     .i_reset ( s_count_reset_align ),
303     .o_flag ( s_count_flag )
304 );
305
306
307 generate
308     if (DATA_BYTES == 8) begin // 64-Bit PIPE
309
310         // Clock
311         assign s_clk = i_clk_core_pll;
312
313
314         // Reset Logic (excludes PLL)
315         //assign s_reset_done = i_rx_reset_done & i_tx_reset_done;
316         assign s_reset_done = i_tx_reset_done;
317
318
319         // Tx Datapath
320         assign o_tx_data      = i_TxData;
321         assign o_tx_char_is_k = i_TxDataK;
322
323
324         // Disparity (Enable and Value)
325         assign o_tx_char_dispmode = i_TxCompliance;
326
327
328         // Rx Datapath
329         assign o_RxData [ 7: 0 ] = ( i_rx_not_in_table[0] ) ? 8'hFE : i_rx_data [ 7: 0 ]; // EDB Symbol (8'
330             hFE)
331         assign o_RxData [15: 8] = ( i_rx_not_in_table[1] ) ? 8'hFE : i_rx_data [15: 8];
332         assign o_RxData [23:16] = ( i_rx_not_in_table[2] ) ? 8'hFE : i_rx_data [23:16];
333         assign o_RxData [31:24] = ( i_rx_not_in_table[3] ) ? 8'hFE : i_rx_data [31:24];

```

```

333     assign o_RxData [39:32] = ( i_rx_not_in_table[4] ) ? 8'hFE : i_rx_data [39:32];
334     assign o_RxData [47:40] = ( i_rx_not_in_table[5] ) ? 8'hFE : i_rx_data [47:40];
335     assign o_RxData [55:48] = ( i_rx_not_in_table[6] ) ? 8'hFE : i_rx_data [55:48];
336     assign o_RxData [63:56] = ( i_rx_not_in_table[7] ) ? 8'hFE : i_rx_data [63:56];
337
338     assign o_RxDataDecErr = i_rx_not_in_table;
339     assign o_RxDataK      = i_rx_char_is_k;
340     assign o_RxDataComma  = i_rx_char_is_comma;
341     assign o_RxDataDispErr = i_rx_disp_err;
342
343     assign s_rx_status_dp = |i_rx_not_in_table ? 3'b100 : ( |i_rx_disp_err ? 3'b111 : 3'b000 );
344
345     end
346
347     else begin // 8-Bit / 16-Bit / 32-Bit PIPE
348
349         wire [7:0] s_neg_disparity;
350         wire      s_pll_locked;
351
352         // Disparity (Enable and Value)
353         assign o_tx_char_dispmode = s_neg_disparity;
354
355         //assign o_tx_char_dispmode = {{8-DATA_BYTES{1'b0}}, s_neg_disparity};
356         //assign o_tx_char_dispval = {{8-DATA_BYTES{1'b0}}, ~s_neg_disparity};
357         //assign o_tx_char_dispval = ~s_neg_disparity; // Constant value!
358
359         // Reset Logic (includes PLL)
360         //assign s_reset_done = s_pll_locked & i_rx_reset_done & i_tx_reset_done;
361         assign s_reset_done = s_pll_locked & i_tx_reset_done;
362
363
364         // Additional PLL for PCLK
365
366         localparam [8*47:1] PLL_PARAM = (DATA_BYTES == 1) ? " 82, 20, 04, 08, 01, 04, 00, 64, 10, 01, CB,
367             01" :
368             (DATA_BYTES == 2) ? " 82, 20, 04, 10, 01, 04, 00, 64, 10, 01, CB,
369             01" :
370             (DATA_BYTES == 4) ? " 82, 20, 04, 10, 02, 04, 00, 64, 10, 01, CB,
371             01" : "X";
372
373         CC_PLL #(
374             .CCAG_CFG_PARAM      ( PLL_PARAM      )
375         )
376         i_cc_pll_0 (
377             .CLK_REF              ( i_clk_core_pll ), // Refclk ADPLL
378             .CLK_FEEDBACK         ( 1'b0          ), // (const value)
379             .USER_CLK_REF         ( 1'b0          ), // (const value)
380             .USER_LOCKED_STDY_RST ( 1'b1          ), // Reset Locked state
381             .USER_SET_SEL         ( 1'b0          ), // (const value)
382             .USER_PLL_LOCKED_STDY (              ), // (float)
383             .USER_PLL_LOCKED     ( s_pll_locked   ), // Locked state
384             .CLK270               (              ), // (float)
385             .CLK180               (              ), // (float)
386             .CLK90                (              ), // (float)
387             .CLKO                 ( s_clk        ), // PCLK
388             .CLK_REF_OUT          (              ), // (float)
389         );
390
391         // Tx Datapath Demultiplexer
392         ccfpga_pipe_tx_demux #(
393             .DATA_BYTES ( DATA_BYTES )
394         )
395         demux_inst_0 (
396             .i_clk      ( s_clk      ),
397             .i_reset    ( s_reset    ),
398             .i_enable   ( 1'b1      ),
399             .i_k_data   ( i_TxDataK  ), // PIPE
400             .i_data     ( i_TxData   ), // PIPE
401             .i_compl    ( i_TxCompliance ), // PIPE
402             .o_data     ( o_tx_data   ), // SerDes
403             .o_k_data   ( o_tx_char_is_k ), // SerDes
404             .o_dispar   ( s_neg_disparity ), // SerDes
405         );
406
407         // Rx Datapath Multiplexer
408         ccfpga_pipe_rx_mux #(
409             .DATA_BYTES ( DATA_BYTES )
410         )

```



```

46     output reg          o_pcs_loopback,      // Loopback Enable
47     output reg          o_tx_elec_idle,
48
49     output reg          o_counter_reset,     // Reset Clock Counter
50     input  wire         i_count_flag,       // Flag for Count Indication
51
52     output reg          o_enable_align      // Activate Word Alignment (FSM)
53 );
54
55     localparam [3:0] RESET          = 4'b0000,
56                    PO_NORMAL      = 4'b0001,
57                    PO_LOOP        = 4'b0010,
58                    PO_IDLE        = 4'b0011,
59                    P1_IDLE        = 4'b0100,
60                    P1_DET_START   = 4'b0101,
61                    P1_DET_GOOD   = 4'b0110,
62                    P1_DET_BAD    = 4'b0111,
63                    P1_DET_WAIT   = 4'b1000,
64                    PO_PRE_NORMAL = 4'b1001, // Used for PhyStatus Generation
65                    PO_PRE_LOOP   = 4'b1010, // Used for PhyStatus Generation
66                    PO_PRE_IDLE   = 4'b1011, // Used for PhyStatus Generation
67                    P1_SET_IDLE   = 4'b1100, // Used to wait for Tx Idle
68                    P1_PRE_IDLE   = 4'b1101; // Used for Phystatus Generation
69
70     reg [3:0] s_state, s_next_state;
71
72     wire [3:0] s_transit;
73     reg        s_rx_polarity;
74
75     assign s_transit = {i_pw_state, i_tx_elec_idle, i_tx_detect_or_loop};
76     assign o_fsm_state = s_state;
77
78     // State Register
79     always@(posedge i_clk, posedge i_reset) begin
80         if (i_reset == 1'b1)
81             s_state <= RESET;
82         else
83             s_state <= s_next_state;
84     end
85
86     // // Transition Logic
87     always@(*) begin
88         s_next_state = s_state;
89         case(s_state)
90             RESET : begin
91                 if ( i_reset_done == 1'b1 )
92                     s_next_state = P1_IDLE;
93                 else
94                     s_next_state = RESET;
95             end
96             P1_IDLE : begin
97                 if ( s_transit == 4'b0010 )
98                     s_next_state = PO_PRE_IDLE;
99                 else if ( s_transit == 4'b0001 )
100                    s_next_state = PO_PRE_LOOP;
101                 else if ( s_transit == 4'b0000 )
102                    s_next_state = PO_PRE_NORMAL;
103                 else if ( s_transit == 4'b1011 )
104                    s_next_state = P1_DET_START;
105                 else
106                    s_next_state = P1_IDLE;
107             end
108             PO_NORMAL : begin
109                 if ( s_transit == 4'b1010 || s_transit == 4'b1011 )
110                    s_next_state = P1_SET_IDLE;
111                 else if ( s_transit == 4'b0010 )
112                    s_next_state = PO_IDLE;
113                 else if ( s_transit == 4'b0001 )
114                    s_next_state = PO_LOOP;
115                 else
116                    s_next_state = PO_NORMAL;
117             end
118             PO_LOOP: begin
119                 if ( s_transit == 4'b1010 || s_transit == 4'b1011 )
120                    s_next_state = P1_SET_IDLE;
121                 else if ( s_transit == 4'b0010 || s_transit == 4'b0011 )
122                    s_next_state = PO_IDLE;

```

```

123         else if ( s_transit == 4'b0000 )
124             s_next_state = P0_NORMAL;
125         else
126             s_next_state = P0_LOOP;
127         end
128     P0_IDLE: begin
129         if ( s_transit == 4'b1010 || s_transit == 4'b1011 )
130             s_next_state = P1_SET_IDLE;
131         else if ( s_transit == 4'b0001 )
132             s_next_state = P0_LOOP;
133         else if ( s_transit == 4'b0000 )
134             s_next_state = P0_NORMAL;
135         else
136             s_next_state = P0_IDLE;
137         end
138     P1_DET_START : begin
139         if ( s_transit[0] == 1'b0 )
140             s_next_state = P1_IDLE;
141         else if ( i_rx_detect_done == 1'b1 && i_rx_detect_val == 1'b1 )
142             s_next_state = P1_DET_GOOD;
143         else if ( i_rx_detect_done == 1'b1 && i_rx_detect_val == 1'b0 )
144             s_next_state = P1_DET_BAD;
145         else
146             s_next_state = P1_DET_START;
147         end
148     P1_DET_GOOD : begin
149         if ( s_transit[0] == 1'b0 )
150             s_next_state = P1_IDLE;
151         else
152             s_next_state = P1_DET_WAIT;
153         end
154     P1_DET_BAD : begin
155         if ( s_transit[0] == 1'b0 )
156             s_next_state = P1_IDLE;
157         else
158             s_next_state = P1_DET_WAIT;
159         end
160     P1_DET_WAIT : begin
161         if ( s_transit[0] == 1'b0 )
162             s_next_state = P1_IDLE;
163         else
164             s_next_state = P1_DET_WAIT;
165         end
166     P0_PRE_NORMAL : begin
167         s_next_state = P0_NORMAL;
168         end
169     P0_PRE_LOOP: begin
170         s_next_state = P0_LOOP;
171         end
172     P0_PRE_IDLE: begin
173         s_next_state = P0_IDLE;
174         end
175     P1_SET_IDLE: begin
176         if ( i_count_flag == 1'b1 )
177             s_next_state = P1_PRE_IDLE;
178         else
179             s_next_state = P1_SET_IDLE;
180         end
181     P1_PRE_IDLE: begin
182         s_next_state = P1_IDLE;
183         end
184     endcase
185 end
186
187 // Output Logic
188 always@(s_state)
189 begin
190     // o_rx_polarity = 0;
191     o_enable_align = 1'b0;
192     o_counter_reset = 1'b1; // new
193     o_tx_elec_idle = 1'b0;
194     o_sel_rx_status = 1'b0;
195     o_phy_status = 1'b0;
196     o_rx_status = 3'b000;
197     o_pcs_loopback = 1'b0;
198     o_rx_detect_start = 1'b0;
199     case(s_state)

```



```
200     RESET : begin
201         o_phy_status      = 1'b1;
202         o_tx_elec_idle    = 1'b1;
203         o_sel_rx_status   = 1'b1;
204     end
205     P1_IDLE : begin
206         o_tx_elec_idle    = 1'b1;
207         o_sel_rx_status   = 1'b1;
208     end
209     P0_NORMAL : begin
210         o_enable_align    = 1'b1;
211     end
212     P0_LOOP: begin
213         o_pcs_loopback    = 1'b1;
214         o_enable_align    = 1'b1;
215     end
216     P0_IDLE: begin
217         o_tx_elec_idle    = 1'b1;
218         o_counter_reset   = 1'b0;
219         o_enable_align    = 1'b1;
220     end
221     P1_DET_START : begin
222         o_tx_elec_idle    = 1'b1;
223         o_sel_rx_status   = 1'b1;
224         o_rx_detect_start = 1'b1;
225     end
226     P1_DET_GOOD : begin
227         o_tx_elec_idle    = 1'b1;
228         o_sel_rx_status   = 1'b1;
229         o_phy_status      = 1'b1;
230         o_rx_status       = 3'b011;
231     end
232     P1_DET_BAD : begin
233         o_tx_elec_idle    = 1'b1;
234         o_sel_rx_status   = 1'b1;
235         o_phy_status      = 1'b1;
236     end
237     P1_DET_WAIT : begin
238         o_tx_elec_idle    = 1'b1;
239         o_sel_rx_status   = 1'b1;
240         //o_phy_status     = 1'b1;
241         //o_rx_status      = 3'b011;
242     end
243     P0_PRE_NORMAL : begin
244         o_phy_status      = 1'b1;
245     end
246     P0_PRE_LOOP : begin
247         o_phy_status      = 1'b1;
248         o_pcs_loopback    = 1'b1;
249     end
250     P0_PRE_IDLE : begin
251         o_counter_reset   = 1'b0;
252         o_phy_status      = 1'b1;
253         o_tx_elec_idle    = 1'b1;
254     end
255     P1_SET_IDLE : begin
256         o_counter_reset   = 1'b0;
257         o_tx_elec_idle    = 1'b1;
258     end
259     P1_PRE_IDLE : begin
260         o_phy_status      = 1'b1;
261         o_sel_rx_status   = 1'b1;
262         o_tx_elec_idle    = 1'b1;
263     end
264 endcase
265 end
266
267 endmodule
```



```
73     ALIGN_START : begin
74         if ( i_byte_locked == 1'b1 )
75             s_next_state = COUNT_START;
76         else
77             s_next_state = ALIGN_START;
78         end
79     COUNT_START : begin
80         if ( i_count_flag == 1'b1 )
81             s_next_state = RX_VALID;
82         else
83             s_next_state = COUNT_START;
84         end
85     RX_VALID: begin
86         s_next_state = WAIT;
87     end
88 endcase
89 end
90
91 // Output Logic
92 always@(s_state)
93 begin
94     o_counter_reset = 1'b1;
95     o_mcomma_align  = 1'b0;
96     o_pcomma_align  = 1'b0;
97     s_rx_valid      = 1'b0;
98     s_set_rx_valid  = 1'b0;
99     case(s_state)
100        WAIT : begin
101            // NONE
102            end
103        ALIGN_START : begin
104            s_set_rx_valid = 1'b1;
105            o_mcomma_align = 1'b1;
106            o_pcomma_align = 1'b1;
107            end
108        COUNT_START : begin
109            o_counter_reset = 1'b0;
110            end
111        RX_VALID: begin
112            s_rx_valid      = 1'b1;
113            s_set_rx_valid = 1'b1;
114            end
115    endcase
116 end
117
118 // DFF Rx Valid
119 always@(posedge i_clk, posedge i_reset) begin
120     if (i_reset == 1'b1)
121         o_rx_valid <= 1'b0;
122     else
123         if ( s_set_rx_valid == 1'b1 )
124             o_rx_valid <= s_rx_valid;
125     end
126
127 endmodule
```

A.4.5 ccfpga_pipe_tx_demux.v

```
1  ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////
2  //
3  // Interessengruppe fuer Mikroelektronik und Eingebettete Systeme (IMES)
4  // Fachhochschule Dortmund
5  //
6  // Development in cooperation with Cologne Chip AG
7  //
8  // Filename      : ccfpga_pipe_demux.v
9  // Author        : Philipp Leduc
10 // Tool          :
11 // Description   : Used in PIPE Logic to demux Tx PIPE (8-,16-Bit) across TX SerDes (64-Bit).
12 //               Also includes Control Symbols (K Data) and Compliance (running disparity).
```

```

13 //          TX PIPE ---> DEMUX ---> TX SERDES
14 // Commentary : Parameter DATA_BYTES refers to Tx PIPE.
15 // Abbreviations : [i_] > input (port)
16 //                [o_] > output (port)
17 //                [s_] > signal
18 //                [_n] > low active
19 //
20 //
21 // Changelog:
22 // -----
23 // Version | Author           | Date           | Changes
24 // -----
25 // 1.0     | Leduc                | 05.06.2021    | released
26 // -----
27 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
28
29 module ccfpga_pipe_tx_demux #(
30     parameter DATA_BYTES = 2,           // Set to configure width of used datapath (PIPE)
31     parameter DATA_WIDTH  = DATA_BYTES*8 // Used to configure bitwidth of datapath
32 )
33 (
34     input wire          i_clk,
35     input wire          i_reset,
36     input wire          i_enable,
37     input wire [DATA_BYTES-1:0] i_k_data, // TxK Data (PIPE)
38     input wire [DATA_WIDTH-1:0] i_data,   // Tx Data (PIPE)
39     input wire [DATA_BYTES-1:0] i_compl,  // Tx Compliance (PIPE)
40     output reg          [63:0] o_data,    // Tx Data (SerDes)
41     output reg          [ 7:0] o_k_data,  // TxK Data (SerDes)
42     output reg          [ 7:0] o_dispar  // Tx Disparity (SerDes)
43 );
44
45 localparam MSB_INIT  = DATA_WIDTH-1; // Used to configure the msb init position
46 localparam MSB_INIT_K = DATA_BYTES-1; // Used to configure the msb k init position
47
48
49 reg [ 5:0] s_msb; // Used for partial vector selection of SerDes Port
50 reg [ 2:0] s_msb_k; // Used for partial vector selection of SerDes Port
51
52
53 // clocked demux for data
54 always@ (posedge i_clk, posedge i_reset) begin
55     if(i_reset == 1'b1) begin
56         o_data <= {64{1'b0}};
57         s_msb <= MSB_INIT;
58     end
59     else begin
60         if (i_enable == 1'b1) begin
61             o_data[s_msb -: DATA_WIDTH] <= i_data;
62             if (s_msb == 6'd63)
63                 s_msb <= MSB_INIT;
64             else
65                 s_msb <= s_msb + DATA_WIDTH;
66         end
67     end
68 end
69
70 // clocked demux for k data and disparity (compliance)
71 always@ (posedge i_clk, posedge i_reset) begin
72     if(i_reset == 1'b1) begin
73         o_dispar <= {8{1'b0}};
74         o_k_data <= {8{1'b0}};
75         s_msb_k <= MSB_INIT_K;
76     end
77     else begin
78         if (i_enable == 1'b1) begin
79             o_k_data[s_msb_k -: DATA_BYTES] <= i_k_data;
80             o_dispar[s_msb_k -: DATA_BYTES] <= i_compl;
81
82             if (s_msb_k == 3'd7)
83                 s_msb_k <= MSB_INIT_K;
84             else
85                 s_msb_k <= s_msb_k + DATA_BYTES;
86         end
87     end
88 end
89

```



```

67  assign s_disp_err_flag = |s_disp_err;
68  assign o_rx_dec_err    = s_dec_err;
69  assign o_rx_disp_err   = s_disp_err;
70
71
72  // clocked mux for data
73  always@(posedge i_clk, posedge i_reset) begin
74      if(i_reset == 1'b1) begin
75          s_data <= {DATA_WIDTH{1'b0}};
76          s_msb <= MSB_INIT;
77      end
78      else begin
79          if (i_enable == 1'b1) begin
80              s_data = i_data [s_msb -: DATA_WIDTH];
81              if (s_msb == 6'd63)
82                  s_msb <= MSB_INIT;
83              else
84                  s_msb <= s_msb + DATA_WIDTH;
85          end
86      end
87  end
88
89  // clocked mux for k data, comma, error signals
90  always@ (posedge i_clk, posedge i_reset) begin
91      if(i_reset == 1'b1) begin
92          o_k_data    = {DATA_BYTES{1'b0}};
93          o_rx_comma  = {DATA_BYTES{1'b0}};
94          s_dec_err   = {DATA_BYTES{1'b0}};
95          s_disp_err  = {DATA_BYTES{1'b0}};
96          s_msb_k     = MSB_INIT_K;
97      end
98      else begin
99          if (i_enable == 1'b1) begin
100             o_k_data    = i_k_data [s_msb_k -: DATA_BYTES];
101             o_rx_comma  = i_char_is_com [s_msb_k -: DATA_BYTES];
102             s_dec_err   = i_decode_err [s_msb_k -: DATA_BYTES];
103             s_disp_err  = i_disp_err [s_msb_k -: DATA_BYTES];
104
105             if (s_msb_k == 3'd7)
106                 s_msb_k <= MSB_INIT_K;
107             else
108                 s_msb_k <= s_msb_k + DATA_BYTES;
109         end
110     end
111 end
112
113 // logic for setting rx status
114 always@* begin
115     if (s_dec_err_flag == 1'b1) // 8b/10b error
116         o_rx_status = 3'b100;
117     else if (s_disp_err_flag == 1'b1) // Disparity error
118         o_rx_status = 3'b111;
119     else // Data OK (rst value)
120         o_rx_status = 3'b000;
121 end
122
123 // logic for implementing EDB Symbol (8b/10b Error)
124 genvar i;
125
126 generate
127     for (i = 0; i < DATA_BYTES ; i = i + 1) begin : gen_block_rx_mux
128         localparam integer j = i*8 + 7;
129         assign o_data [ j -: 8] = ( s_dec_err[i] == 1'b1 ) ? 8'hFE : s_data [ j -: 8];
130     end
131 endgenerate
132
133
134 endmodule

```

A.4.7 CC_PLL.v

```

1  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
2  //
3  // Interessengruppe fuer Mikroelektronik und Eingebettete Systeme (IMES)
4  // Fachhochschule Dortmund
5  //
6  // Development in cooperation with Cologne Chip AG
7  //
8  // Filename      : CC_PLL.v
9  // Author       : Philipp Leduc
10 // Tool        :
11 // Description  : Simple "Functional Module" of CCAG Gate Mate PLL.
12 // Commentary  : Not for Synthesis, only intended for Simulation.
13 //              : The Delay Lock Time of the PLL is defined as 1100 Cycles of the RefClk (32ns):
14 //              : Delay Lock Time = 1100 * Tref = 1100 * 32ns = 35200ns.
15 // Abbreviations :
16 // Changelog:
17 // -----
18 // Version | Author          | Date       | Changes
19 // -----
20 // 1.0     | Leduc            | 05.06.2021 | released
21 // -----
22 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
23
24 `timescale 1ns/10ps
25
26 module CC_PLL #(
27     parameter [8*47:1] CCAG_CFG_PARAM = " 82, 20, 04, 08, 01, 04, 00, 64, 10, 01, CB, 01"
28 )
29 (
30     input wire CLK_REF,           // Refclk ADPLL
31     input wire CLK_FEEDBACK,     // (const value)
32     input wire USER_CLK_REF,    // (const value)
33     input wire USER_LOCKED_STDY_RST, // Reset Locked state
34     input wire USER_SET_SEL,    // (const value)
35     output wire USER_PLL_LOCKED_STDY, // (float)
36     output wire USER_PLL_LOCKED, // Locked state
37     output wire CLK270,         // (float)
38     output wire CLK180,        // (float)
39     output wire CLK90,         // (float)
40     output wire CLK0,          // PCLK
41     output wire CLK_REF_OUT    // (float)
42 );
43
44 // synthesis translate_off
45
46 localparam CLK_PERIOD = (CCAG_CFG_PARAM == " 82, 20, 04, 08, 01, 04, 00, 64, 10, 01, CB, 01") ? 4 :
47 // 250 Mhz
48 (CCAG_CFG_PARAM == " 82, 20, 04, 10, 01, 04, 00, 64, 10, 01, CB, 01") ? 8 :
49 // 125 Mhz
50 (CCAG_CFG_PARAM == " 82, 20, 04, 10, 02, 04, 00, 64, 10, 01, CB, 01") ? 16 : 0;
51 // 62,5 Mhz
52
53 integer cnt;
54
55 reg s_active;
56 reg s_clk_out;
57 reg s_present;
58 reg s_merker;
59
60 initial s_clk_out = 1'b1;
61 initial s_active = 1'b0;
62 initial s_merker = 1'b0;
63 initial s_present = 1'b0;
64
65 assign CLK0 = s_clk_out & s_active;
66
67 assign USER_PLL_LOCKED = s_active;
68
69 // PLL Clock Generation
70 always begin
71     #(CLK_PERIOD/2) s_clk_out = ~s_clk_out;
72 end

```

```

70
71
72 // PLL Control
73 always@(posedge s_clk_out) begin
74     if (s_merker == 1'b0) begin
75         @(posedge CLK_REF);
76         #35200 s_active = 1'b1;
77         s_merker = 1'b1;
78     end
79     else if ( s_merker == 1'b1 & s_present == 1'b0 ) begin
80         s_active = 1'b0;
81         s_merker = 1'b0;
82     end
83 end
84
85 // ADPLL Refclock Detection
86 always@(negedge s_clk_out) begin
87     if (CLK_REF == 1'b0) begin
88         cnt = cnt + 1;
89         if ( cnt > CLK_PERIOD+1 ) begin
90             s_present = 1'b0;
91         end
92     end
93     else if (CLK_REF == 1'b1) begin
94         cnt = 0;
95         s_present = 1'b1;
96     end
97 end
98
99 // synthesis translate_on
100
101 endmodule

```

A.5 Quellcode der Testumgebung

A.5.1 tb_ccfpga_serdes.v

```

37 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
38 //
39 // Interessengruppe fuer Mikroelektronik und Eingebettete Systeme (IMES)
40 // Fachhochschule Dortmund
41 //
42 // Development in cooperation with Cologne Chip AG
43 //
44 // Filename      : tb_ccfpga_serdes.v
45 // Author       : Philipp Leduc, S. Hartman (Author of Original Testbench Design)
46 // Tool        :
47 // Description  : Testcase for PIPE Interface and SerDes of the Gatemate FPGA.
48 // Commentary  : Testbench is based on the former Testbench by RacyIC GmbH (tb_ccfpga_serdes.v).
49 //              The Testbench has been expanded to test SerDes and the PIPE.
50 //
51 //              > To choose between Test, define PIPE or SERDES Macro.
52 //
53 //              > For PIPE test define Datapath width through Parameter DATA_BYTES.
54 //
55 //              8 > 64 Bit Datapath (PIPE)
56 //              4 > 32 Bit Datapath (PIPE)
57 //              2 > 16 Bit Datapath (PIPE)
58 //              1 > 8 Bit Datapath (PIPE)
59 //
60 // Abbreviations : [i_] > input,
61 //                 [o_] > output,
62 //                 [_n] > low active
63 //
64 // Changelog:
65 // -----
66 // Version | Author              | Date      | Changes

```



```

67 // -----
68 // 1.0      | Hartmann          | 07.08.2015 | black box release (original serdes version)
69 // -----
70 // 1.1      | Leduc            | 05.06.2021 | released (PIPE interface version)
71 // -----
72 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
73
74 `ifndef AMS
75     `include "disciplines.vams"
76 `endif
77
78 `timescale 1ns/100fs
79
80 module tb_ccfpga_serdes ();
81
82 `ifndef PIPE
83
84     // PIPE Parameter
85
86     parameter          DATA_BYTES = 8;
87     parameter          DATA_WIDTH = DATA_BYTES*8;
88
89     // Signals
90
91     // PIPE Interface
92     wire                o_PCLK;           // PCLK (user side)
93     reg                 i_Reset;          // Async. Reset for Transceiver (Tx/Rx)
94     reg                 [ 1:0] i_PowerDown; // Power States (P0 - P2)
95     reg                 i_TxDetectRx;    // Receiver Detection (P0) or Loopback (P1)
96     reg                 i_TxElecIdle;    // Tx Electrical Idle, Valid Data (P0) or Beacon (P2)
97     reg [DATA_BYTES-1:0] i_TxCompliance; // Tx negative Disparity LSB (Compliance Pattern)
98     // wire               i_TxSwing;      // Tx Voltage Swing Level [Optional by Spec]
99     reg                 i_RxPolarity;    // Rx Polarity Inversion
100    wire                 o_RxValid;       // Symbol Lock and Valid Data on RxData and RxDataK
101    wire                 o_PhyStatus;     // Status of several PHY functions (Transition)
102    wire                 o_RxElecIdle;    // Rx Detection of Electrical Idle
103    wire                 [ 2:0] o_RxStatus; // Receiver Status and Received Data Status
104    reg [DATA_WIDTH-1:0] i_TxData;        // Tx Data
105    reg [DATA_BYTES-1:0] i_TxDataK;      // Tx K Data
106    wire [DATA_WIDTH-1:0] o_RxData;       // Rx Data
107    wire [DATA_BYTES-1:0] o_RxDataK;     // Rx K Data
108
109    // Support Interface
110
111    wire                 o_tx_buf_error;   // Tx Buffer Error
112    wire                 o_clk_core_rx_rec; // Rx Recovered Clock
113    reg                 i_rx_buf_reset;    // Rx Buffer Reset
114    wire                 o_rx_buf_err;     // Rx Buffer Error
115    wire                 [ 3:0] o_fsm_state_pipe; // State of PIPE FSM
116    wire                 [ 1:0] o_fsm_state_align; // State of Align FSM
117
118    wire [DATA_BYTES-1:0] o_RxDataComma;   // Rx Byte Comma Indication
119    wire [DATA_BYTES-1:0] o_RxDataDispErr; // Rx Byte Disparity Error Indication
120    wire [DATA_BYTES-1:0] o_RxDataDecErr;  // Rx Byte Decode Error Indication
121
122    // Used by PIPE Logic (SerDes)
123
124    wire                 clk_core_tx;
125    wire                 clk_core_rx_in; // rx core clock
126    wire                 clk_core_rx_rec; // rx recovered clock
127    wire                 clk_core_pll;
128
129    wire                 pll_reset;
130    wire                 tx_reset;
131    wire                 tx_pcs_reset;
132    wire                 tx_pma_reset;
133
134    wire [2:0] loopback;
135
136    wire                 tx_elec_idle;
137    wire                 tx_detect_rx;
138    wire [2:0] tx_prbs_sel;
139    wire                 tx_prbs_force_err;
140    wire                 tx_powerdown_n;
141    wire [63:0] tx_data;
142    wire [7:0] tx_char_is_k;
143    wire                 tx_polarity;

```

```

144     wire          tx_8b10b_en;
145     wire [7:0] tx_8b10b_bypass;
146     wire [7:0] tx_char_dispmode;
147     wire [7:0] tx_char_dispval;
148
149     wire          rx_reset;
150     wire          rx_pma_reset;
151     wire          rx_eqa_reset;
152     wire          rx_cdr_reset;
153     wire          rx_pcs_reset;
154     wire          rx_buf_reset;
155     wire [2:0] rx_prbs_sel;
156     wire          rx_prbs_cnt_reset;
157     wire          rx_powerdown_n;
158     wire          rx_en_ei_detector;
159     wire          rx_comma_detect_en;
160     wire          rx_slide;
161     wire          rx_polarity;
162     wire          rx_8b10b_en;
163     wire [7:0] rx_8b10b_bypass;
164     wire          rx_mcomma_align;
165     wire          rx_pcomma_align;
166
167     wire          tx_buf_err;
168     wire          tx_resetdone;
169     wire          rx_detect_done;
170     wire          rx_present;
171     wire          rx_prbs_err;
172     wire [63:0] rx_data;
173     wire [7:0] rx_char_is_k;
174     wire [7:0] rx_char_is_comma;
175     wire [7:0] rx_not_in_table;
176     wire [7:0] rx_disp_err;
177     wire          rx_buf_err;
178     wire          rx_byte_is_aligned;
179     wire          rx_byte_realign;
180     wire          rx_resetdone;
181     wire          rx_ei_en;
182
183     `elsif SERDES
184
185     wire          clk_core_tx;
186     wire          clk_core_rx_in; // rx core clock
187     wire          clk_core_rx_rec; // rx recoverd clock
188     wire          clk_core_pll;
189
190     reg          pll_reset          = 1'b0;
191     reg          tx_reset           = 1'b0;
192     reg          tx_pcs_reset       = 1'b0;
193     reg          tx_pma_reset       = 1'b0;
194
195     reg [2:0] loopback             = 3'b0;
196
197     reg          tx_elec_idle       = 1'b0;
198     reg          tx_detect_rx       = 1'b0;
199     reg [2:0] tx_prbs_sel          = 3'b0;
200     reg          tx_prbs_force_err  = 1'b0;
201     reg          tx_powerdown_n     = 1'b0;
202     reg [63:0] tx_data             = 64'h0;
203     reg [7:0] tx_char_is_k         = 8'h0;
204     reg          tx_polarity        = 1'b0;
205     reg          tx_8b10b_en        = 1'b0;
206     reg [7:0] tx_8b10b_bypass      = 8'h0;
207     reg [7:0] tx_char_dispmode     = 8'h0;
208     reg [7:0] tx_char_dispval     = 8'h0;
209
210     reg          rx_reset = 1'b0;
211     reg          rx_pma_reset = 1'b0;
212     reg          rx_eqa_reset = 1'b0;
213     reg          rx_cdr_reset = 1'b0;
214     reg          rx_pcs_reset = 1'b0;
215     reg          rx_buf_reset = 1'b0;
216     reg [2:0] rx_prbs_sel = 3'b0;
217     reg          rx_prbs_cnt_reset = 1'b0;
218     reg          rx_powerdown_n = 1'b0;
219     reg          rx_en_ei_detector = 1'b0;
220     reg          rx_comma_detect_en = 1'b0;

```

```

221     reg          rx_slide          = 1'b0;
222     reg          rx_polarity       = 1'b0;
223     reg          rx_8b10b_en      = 1'b0;
224     reg [7:0]    rx_8b10b_bypass  = 8'h0;
225     reg          rx_mcomma_align  = 1'b0;
226     reg          rx_pcomma_align  = 1'b0;
227
228     wire         tx_buf_err;
229     wire         tx_resetdone;
230     wire         rx_detect_done;
231     wire         rx_present;
232     wire         rx_prbs_err;
233     wire [63:0]  rx_data;
234     wire [7:0]   rx_char_is_k;
235     wire [7:0]   rx_char_is_comma;
236     wire [7:0]   rx_not_in_table;
237     wire [7:0]   rx_disp_err;
238     wire         rx_buf_err;
239     wire         rx_byte_is_aligned;
240     wire         rx_byte_realign;
241     wire         rx_resetdone;
242     wire         rx_ei_en;
243
244     `endif
245
246     // Not available through FPGA Fabric
247
248     reg          clk_ref;
249     reg          clk_reg;
250     reg          clk_cfg;
251
252     reg          reset_n;
253     reg          reset_reg_n;
254     wire         reset_core_pll_n;
255     wire         reset_core_tx_n;
256     wire         reset_core_rx_n;
257
258     reg          testmode          = 1'b0;
259     reg          scan_enable       = 1'b0;
260
261     reg          cfg_en            = 1'b0;
262     reg          reset_cfg        = 1'b0;
263     reg [15:0]   cfgfile_di;
264     reg          cfgfile_valid    = 1'b0;
265     reg [7:0]    cfgfile_addr;
266
267     // Register Interface
268
269     reg          regfile_we;
270     reg          regfile_en      = 1'b0;
271     reg [7:0]    regfile_addr;
272     reg [15:0]   regfile_mask;
273     reg [15:0]   regfile_di;
274     wire [15:0]  regfile_do;
275     wire         regfile_rdy;
276
277     // Analog
278
279     `ifdef AMS
280         electrical TERM_SERIO;
281         electrical RX_SERIO_P;
282         electrical RX_SERIO_N;
283         electrical TX_SERIO_P;
284         electrical TX_SERIO_N;
285     `else
286         wire         TERM_SERIO;
287         reg          RX_SERIO_P;
288         reg          RX_SERIO_N;
289         wire         TX_SERIO_P;
290         wire         TX_SERIO_N;
291     `endif // !`ifdef AMS
292
293
294     // Tb Variables
295     integer      errors            = 0;
296     integer      checks_done      = 0;
297

```

```

298 // Parameters for Self Calibration in task startSerIOADPLL
299
300 parameter ADPLL_PFDAC_TIMER = 12;
301 parameter ADPLL_PFDAC_COR_DLY = 1;
302 parameter ADPLL_PFDAC_CP_MIN = 6;
303 parameter ADPLL_PFDAC_CP_MAX = 30;
304 parameter ADPLL_PFDAC_CP_START = 6;
305 parameter ADPLL_PFDAC_CAL_SIGN = 1;
306 parameter ADPLL_PFDAC_AUTO_CAL = 1;
307
308 // SerDes Config Parameter
309
310 parameter [5:0] PLL_FCNTL = 58; // (Default = 58 = T:20d)
311 parameter [5:0] PLL_MAIN_DIVSEL = {1'b0,2'b11,1'b0,2'b11}; // (Default = 27)
312 parameter [1:0] PLL_OUT_DIVSEL = 2'b01; // (Default = 0 = T:id)
313
314 parameter [1:0] RX_DATAPATH_SEL = 3; // (Default = 3)
315 parameter [1:0] TX_DATAPATH_SEL = 3; // (Default = 3)
316
317 parameter [9:0] ALIGN_MCOMMA_VALUE = 10'b1010000011; // (Default = 10'b1010000011)
318 parameter [9:0] ALIGN_PCOMMA_VALUE = 10'b0101111100; // (Default = 10'b0101111100)
319 parameter [9:0] ALIGN_COMMA_ENABLE = 10'b1111111111; // (Default = FFF) : Maske
320 parameter [1:0] ALIGN_COMMA_WORD = 2'b11; // (Default = 2'b00 : 8 Bit)
321 parameter [1:0] RX_SLIDE_MODE = 0;
322
323 parameter PLL_CONFIG_SEL = 0; // (Default = 1'b0) (spec!)
324
325 parameter TX_BUFFER_ADDR_WIDTH = 5;
326 parameter RX_WAIT_CDR_LOCK = 1;
327 parameter RX_RESETDONE_GATE = 0;
328 parameter [4:0] RX_RESET_TIMER_PRESC = 0;
329 parameter [4:0] RX_PMA_RESET_TIME = 3;
330 parameter [4:0] RX_EQA_RESET_TIME = 3;
331 parameter [4:0] RX_CDR_RESET_TIME = 3;
332 parameter [4:0] RX_PCS_RESET_TIME = 3;
333 parameter [4:0] RX_BUF_RESET_TIME = 3;
334 parameter RX_CALIB_OVR = 0;
335 parameter [3:0] RX_CALIB_VAL = 0;
336 parameter [2:0] RX_RTERM_VCMSEL = 4;
337 parameter RX_RTERM_PD = 0;
338 parameter [7:0] RX_EQA_CKP_HF = 8'b101_00011;
339 parameter [7:0] RX_EQA_CKP_LF = 8'b101_00011;
340 parameter [15:0] RX_EQA_CONFIG = 16'h01C0;
341 parameter [7:0] RX_EQA_CKP_OFFSET = 8'b000_00001;
342 parameter RX_EN_EQA = 0;
343 parameter [3:0] RX_EQA_LOCK_CFG = 0;
344 parameter RX_TH_MON1_OVR = 0;
345 parameter RX_TH_MON2_OVR = 0;
346 parameter RX_TAPW_OVR = 0;
347 parameter RX_AFE_OFFSET_OVR = 0;
348 parameter [4:0] RX_TH_MON1 = 8;
349 parameter [4:0] RX_TH_MON2 = 8;
350 parameter [4:0] RX_TAPW = 8;
351 parameter [4:0] RX_AFE_OFFSET = 8;
352 parameter [2:0] RX_AFE_VCMSEL = 4;
353 parameter [3:0] RX_AFE_GAIN = 8;
354 parameter [4:0] RX_AFE_PEAK = 16;
355 parameter [7:0] RX_CDR_CKP = {3'b101, 5'b01111};
356 parameter [7:0] RX_CDR_CKI = {3'b000, 5'b00000};
357 parameter [14:0] RX_CDR_LOCK_CFG = 15'b000_1000_1101_0101;
358 parameter [1:0] RX_CDR_SET_ACC_CONFIG = 2'b00;
359 parameter RX_CDR_FORCE_LOCK = 0;
360 parameter [14:0] RX_CDR_FREQ_ACC = 15'h0;
361 parameter [15:0] RX_CDR_PHASE_ACC = 16'h0;
362 parameter [14:0] RX_EYE_MEAS_CFG = 0;
363 parameter [5:0] RX_MON_PH_OFFSET = 0;
364 parameter [3:0] RX_EI_BIAS = 4;
365 parameter [3:0] RX_EI_BW_SEL = 4;
366 parameter RX_BUF_BYPASS = 0;
367 parameter RX_CLKCOR_USE = 0;
368 parameter [9:0] RX_CLKCOR_SEQ_1_0 = 10'b0111110111;
369 parameter [9:0] RX_CLKCOR_SEQ_1_1 = 10'b0111110111;
370 parameter [9:0] RX_CLKCOR_SEQ_1_2 = 10'b0111110111;
371 parameter [9:0] RX_CLKCOR_SEQ_1_3 = 10'b0111110111;
372 parameter [5:0] RX_CLKCOR_MIN_LAT = 32;
373 parameter [5:0] RX_CLKCOR_MAX_LAT = 39;
374 parameter [5:0] RX_DBG_SRAM_DELAY = 6'b00_01_01;

```

```

375 parameter [3:0] RX_DBG_SEL = 0;
376 parameter RX_DBG_MODE = 0;
377 parameter [4:0] TX_SEL_PRE = 0;
378 parameter [4:0] TX_SEL_POST = 0;
379 parameter [2:0] TX_TAIL_CASCADE = 3'b100;
380 parameter [4:0] TX_BRANCH_EN_PRE = 0;
381 parameter [5:0] TX_BRANCH_EN_MAIN = 6'b111111;
382 parameter [4:0] TX_BRANCH_EN_POST = 0;
383 parameter [6:0] TX_DC_ENABLE = 63;
384 parameter [4:0] TX_AMP = 15;
385 parameter [4:0] TX_DC_OFFSET = 0;
386 parameter [4:0] TX_CM_RAISE = 0;
387 parameter [4:0] TX_CM_THRESHOLD_0 = 14;
388 parameter [4:0] TX_CM_THRESHOLD_1 = 16;
389 parameter TX_CALIB_OVR = 0;
390 parameter [3:0] TX_CALIB_VAL = 0;
391 parameter TX_CM_REG_EN = 1;
392 parameter [7:0] TX_CM_REG_KI = 8'h80;
393 parameter [4:0] TX_PMA_RESET_TIME = 3;
394 parameter [4:0] TX_PCS_RESET_TIME = 3;
395 parameter TX_PMA_LOOPBACK = 0;
396 parameter PLL_EN_ADPLL_CTRL = 0;
397
398 parameter PLL_SET_OP_LOCK = 1;
399 parameter PLL_ENFORCE_LOCK = 0;
400 parameter PLL_DISABLE_LOCK = 0;
401 parameter PLL_LOCK_WINDOW = 1;
402 parameter PLL_FAST_LOCK = 1;
403 parameter PLL_SYNC_BYPASS = 0;
404 parameter PLL_PFD_SELECT = 0;
405 parameter PLL_REF_BYPASS = 0;
406 parameter PLL_REF_SEL = 0;
407 parameter PLL_REF_RTERM = 1;
408 parameter [4:0] PLL_CI = 1;
409 parameter [9:0] PLL_CP = 60;
410 parameter [3:0] PLL_AO = 0;
411 parameter [2:0] PLL_SCAP = 0;
412 parameter PLL_SCAP_AUTO_CAL = 1;
413 parameter [1:0] PLL_FILTER_SHIFT = 2;
414 parameter [2:0] PLL_SAR_LIMIT = 2;
415 parameter [10:0] PLL_FT = 512;
416 parameter PLL_OPEN_LOOP = 0;
417 parameter [2:0] PLL_BISC_MODE = 3'b100;
418 parameter [3:0] PLL_BISC_TIMER_MAX = 15;
419 parameter [4:0] PLL_BISC_CP_MIN = 4;
420 parameter [4:0] PLL_BISC_CP_MAX = 18;
421 parameter [4:0] PLL_BISC_CP_START = 12;
422 parameter PLL_BISC_OPT_DET_IND = 0;
423 parameter PLL_BISC_PFD_SEL = 0;
424 parameter PLL_BISC_DLY_DIR = 0;
425 parameter PLL_BISC_CAL_SIGN = 0;
426 parameter PLL_BISC_CAL_AUTO = 1;
427 parameter [4:0] PLL_BISC_DLY_PFD_MON_REF = 0;
428 parameter [4:0] PLL_BISC_DLY_PFD_MON_DIV = 2;
429 parameter [2:0] PLL_BISC_COR_DLY = 1;
430
431
432 // SerDes Instantiation
433 ccfpga_serdes #(
434     .TX_BUFFER_ADDR_WIDTH ( TX_BUFFER_ADDR_WIDTH ),
435     .RX_WAIT_CDR_LOCK ( RX_WAIT_CDR_LOCK ),
436     .RX_RESETDONE_GATE ( RX_RESETDONE_GATE ),
437     .RX_RESET_TIMER_PRESC ( RX_RESET_TIMER_PRESC ),
438     .RX_PMA_RESET_TIME ( RX_PMA_RESET_TIME ),
439     .RX_EQA_RESET_TIME ( RX_EQA_RESET_TIME ),
440     .RX_CDR_RESET_TIME ( RX_CDR_RESET_TIME ),
441     .RX_PCS_RESET_TIME ( RX_PCS_RESET_TIME ),
442     .RX_BUF_RESET_TIME ( RX_BUF_RESET_TIME ),
443     .RX_CALIB_OVR ( RX_CALIB_OVR ),
444     .RX_CALIB_VAL ( RX_CALIB_VAL ),
445     .RX_RTERM_VCMSEL ( RX_RTERM_VCMSEL ),
446     .RX_RTERM_PD ( RX_RTERM_PD ),
447     .RX_EQA_CKP_HF ( RX_EQA_CKP_HF ),
448     .RX_EQA_CKP_LF ( RX_EQA_CKP_LF ),
449     .RX_EQA_CONFIG ( RX_EQA_CONFIG ),
450     .RX_EQA_CKP_OFFSET ( RX_EQA_CKP_OFFSET ),
451     .RX_EN_EQA ( RX_EN_EQA ),

```

452	.RX_EQA_LOCK_CFG	(RX_EQA_LOCK_CFG),
453	.RX_TH_MON1_OVR	(RX_TH_MON1_OVR),
454	.RX_TH_MON2_OVR	(RX_TH_MON2_OVR),
455	.RX_TAPW_OVR	(RX_TAPW_OVR),
456	.RX_AFE_OFFSET_OVR	(RX_AFE_OFFSET_OVR),
457	.RX_TH_MON1	(RX_TH_MON1),
458	.RX_TH_MON2	(RX_TH_MON2),
459	.RX_TAPW	(RX_TAPW),
460	.RX_AFE_OFFSET	(RX_AFE_OFFSET),
461	.RX_AFE_VCMSEL	(RX_AFE_VCMSEL),
462	.RX_AFE_GAIN	(RX_AFE_GAIN),
463	.RX_AFE_PEAK	(RX_AFE_PEAK),
464	.RX_CDR_CKP	(RX_CDR_CKP),
465	.RX_CDR_CKI	(RX_CDR_CKI),
466	.RX_CDR_LOCK_CFG	(RX_CDR_LOCK_CFG),
467	.RX_CDR_SET_ACC_CONFIG	(RX_CDR_SET_ACC_CONFIG),
468	.RX_CDR_FORCE_LOCK	(RX_CDR_FORCE_LOCK),
469	.RX_CDR_FREQ_ACC	(RX_CDR_FREQ_ACC),
470	.RX_CDR_PHASE_ACC	(RX_CDR_PHASE_ACC),
471	.ALIGN_MCOMMA_VALUE	(ALIGN_MCOMMA_VALUE),
472	.ALIGN_PCOMMA_VALUE	(ALIGN_PCOMMA_VALUE),
473	.ALIGN_COMMA_ENABLE	(ALIGN_COMMA_ENABLE),
474	.ALIGN_COMMA_WORD	(ALIGN_COMMA_WORD),
475	.RX_SLIDE_MODE	(RX_SLIDE_MODE),
476	.RX_EYE_MEAS_CFG	(RX_EYE_MEAS_CFG),
477	.RX_MON_PH_OFFSET	(RX_MON_PH_OFFSET),
478	.RX_EI_BIAS	(RX_EI_BIAS),
479	.RX_EI_BW_SEL	(RX_EI_BW_SEL),
480	.RX_BUF_BYPASS	(RX_BUF_BYPASS),
481	.RX_CLKCOR_USE	(RX_CLKCOR_USE),
482	.RX_CLKCOR_SEQ_1_0	(RX_CLKCOR_SEQ_1_0),
483	.RX_CLKCOR_SEQ_1_1	(RX_CLKCOR_SEQ_1_1),
484	.RX_CLKCOR_SEQ_1_2	(RX_CLKCOR_SEQ_1_2),
485	.RX_CLKCOR_SEQ_1_3	(RX_CLKCOR_SEQ_1_3),
486	.RX_CLKCOR_MIN_LAT	(RX_CLKCOR_MIN_LAT),
487	.RX_CLKCOR_MAX_LAT	(RX_CLKCOR_MAX_LAT),
488	.RX_DATAPATH_SEL	(RX_DATAPATH_SEL),
489	.RX_DBG_SRAM_DELAY	(RX_DBG_SRAM_DELAY),
490	.RX_DBG_SEL	(RX_DBG_SEL),
491	.RX_DBG_MODE	(RX_DBG_MODE),
492	.TX_SEL_PRE	(TX_SEL_PRE),
493	.TX_SEL_POST	(TX_SEL_POST),
494	.TX_TAIL_CASCADE	(TX_TAIL_CASCADE),
495	.TX_BRANCH_EN_PRE	(TX_BRANCH_EN_PRE),
496	.TX_BRANCH_EN_MAIN	(TX_BRANCH_EN_MAIN),
497	.TX_BRANCH_EN_POST	(TX_BRANCH_EN_POST),
498	.TX_DC_ENABLE	(TX_DC_ENABLE),
499	.TX_AMP	(TX_AMP),
500	.TX_DC_OFFSET	(TX_DC_OFFSET),
501	.TX_CM_RAISE	(TX_CM_RAISE),
502	.TX_CM_THRESHOLD_0	(TX_CM_THRESHOLD_0),
503	.TX_CM_THRESHOLD_1	(TX_CM_THRESHOLD_1),
504	.TX_CALIB_OVR	(TX_CALIB_OVR),
505	.TX_CALIB_VAL	(TX_CALIB_VAL),
506	.TX_CM_REG_EN	(TX_CM_REG_EN),
507	.TX_CM_REG_KI	(TX_CM_REG_KI),
508	.TX_PMA_RESET_TIME	(TX_PMA_RESET_TIME),
509	.TX_PCS_RESET_TIME	(TX_PCS_RESET_TIME),
510	.TX_PMA_LOOPBACK	(TX_PMA_LOOPBACK),
511	.TX_DATAPATH_SEL	(TX_DATAPATH_SEL),
512	.PLL_EN_ADPLL_CTRL	(PLL_EN_ADPLL_CTRL),
513	.PLL_CONFIG_SEL	(PLL_CONFIG_SEL),
514	.PLL_SET_OP_LOCK	(PLL_SET_OP_LOCK),
515	.PLL_ENFORCE_LOCK	(PLL_ENFORCE_LOCK),
516	.PLL_DISABLE_LOCK	(PLL_DISABLE_LOCK),
517	.PLL_LOCK_WINDOW	(PLL_LOCK_WINDOW),
518	.PLL_FAST_LOCK	(PLL_FAST_LOCK),
519	.PLL_SYNC_BYPASS	(PLL_SYNC_BYPASS),
520	.PLL_PFD_SELECT	(PLL_PFD_SELECT),
521	.PLL_REF_BYPASS	(PLL_REF_BYPASS),
522	.PLL_REF_SEL	(PLL_REF_SEL),
523	.PLL_REF_RTERM	(PLL_REF_RTERM),
524	.PLL_FCNTL	(PLL_FCNTL),
525	.PLL_MAIN_DIVSEL	(PLL_MAIN_DIVSEL),
526	.PLL_OUT_DIVSEL	(PLL_OUT_DIVSEL),
527	.PLL_CI	(PLL_CI),
528	.PLL_CP	(PLL_CP),

```

529     .PLL_AO                ( PLL_AO                ),
530     .PLL_SCAP             ( PLL_SCAP             ),
531     .PLL_SCAP_AUTO_CAL   ( PLL_SCAP_AUTO_CAL   ),
532     .PLL_FILTER_SHIFT    ( PLL_FILTER_SHIFT    ),
533     .PLL_SAR_LIMIT       ( PLL_SAR_LIMIT       ),
534     .PLL_FT               ( PLL_FT               ),
535     .PLL_OPEN_LOOP       ( PLL_OPEN_LOOP       ),
536     .PLL_BISC_MODE        ( PLL_BISC_MODE        ),
537     .PLL_BISC_TIMER_MAX   ( PLL_BISC_TIMER_MAX   ),
538     .PLL_BISC_CP_MIN      ( PLL_BISC_CP_MIN      ),
539     .PLL_BISC_CP_MAX      ( PLL_BISC_CP_MAX      ),
540     .PLL_BISC_CP_START    ( PLL_BISC_CP_START    ),
541     .PLL_BISC_OPT_DET_IND ( PLL_BISC_OPT_DET_IND  ),
542     .PLL_BISC_PFD_SEL     ( PLL_BISC_PFD_SEL     ),
543     .PLL_BISC_DLY_DIR     ( PLL_BISC_DLY_DIR     ),
544     .PLL_BISC_CAL_SIGN    ( PLL_BISC_CAL_SIGN    ),
545     .PLL_BISC_CAL_AUTO    ( PLL_BISC_CAL_AUTO    ),
546     .PLL_BISC_DLY_PFD_MON_REF ( PLL_BISC_DLY_PFD_MON_REF ),
547     .PLL_BISC_DLY_PFD_MON_DIV ( PLL_BISC_DLY_PFD_MON_DIV ),
548     .PLL_BISC_COR_DLY     ( PLL_BISC_COR_DLY     )
549 )
550 i_ccfpga_serdes (
551     .clk_ref_i             ( clk_ref             ),
552     .clk_reg_i            ( clk_reg            ),
553     .clk_cfg_i            ( clk_cfg            ),
554     .clk_core_tx_i        ( clk_core_tx        ),
555     .clk_core_rx_i        ( clk_core_rx_in     ),
556     .clk_core_rx_o        ( clk_core_rx_rec    ),
557     .reset_n_i           ( reset_n            ),
558     .reset_reg_n_i        ( reset_reg_n        ),
559     .reset_core_tx_n_i    ( reset_core_tx_n    ),
560     .reset_core_rx_n_i    ( reset_core_rx_n    ),
561     .testmode_i           ( testmode           ),
562     .scan_enable_i        ( scan_enable        ),
563     .scan_in_i            (                    ), // floating ports
564     .scan_out_o           (                    ),
565     .loopback_i           ( loopback           ),
566     .pll_reset_i          ( pll_reset          ),
567     .clk_core_pll_o        ( clk_core_pll       ),
568     .reset_core_pll_n_o   ( reset_core_pll_n   ),
569     .refclk_sel_o         (                    ), // floating ports
570     .refclk_rterm_o       (                    ),
571
572     .RX_SERIO_P_B         ( RX_SERIO_P         ),
573     .RX_SERIO_N_B         ( RX_SERIO_N         ),
574     .TX_SERIO_P_B         ( TX_SERIO_P         ),
575     .TX_SERIO_N_B         ( TX_SERIO_N         ),
576
577     .regfile_we_i         ( regfile_we         ),
578     .regfile_en_i         ( regfile_en         ),
579     .regfile_addr_i       ( regfile_addr       ),
580     .regfile_mask_i       ( regfile_mask       ),
581     .regfile_di_i         ( regfile_di         ),
582     .regfile_do_o         ( regfile_do         ),
583     .regfile_rdy_o        ( regfile_rdy        ),
584
585     .cfg_en_i             ( cfg_en             ),
586     .reset_cfg_i          ( reset_cfg          ),
587     .cfgfile_di_i         ( cfgfile_di         ),
588     .cfgfile_valid_i      ( cfgfile_valid      ),
589     .cfgfile_addr_i       ( cfgfile_addr       ),
590
591     .tx_reset_i           ( tx_reset           ),
592     .tx_pcs_reset_i       ( tx_pcs_reset       ),
593     .tx_pma_reset_i       ( tx_pma_reset       ),
594     .tx_elec_idle_i       ( tx_elec_idle       ),
595     .tx_detect_rx_i       ( tx_detect_rx       ),
596     .tx_prbs_sel_i        ( tx_prbs_sel        ),
597     .tx_prbs_force_err_i  ( tx_prbs_force_err   ),
598     .tx_powerdown_n_i     ( tx_powerdown_n     ),
599     .tx_data_i            ( tx_data            ),
600     .tx_char_is_k_i       ( tx_char_is_k       ),
601     .tx_polarity_i        ( tx_polarity        ),
602     .tx_8b10b_en_i       ( tx_8b10b_en       ),
603     .tx_8b10b_bypass_i   ( tx_8b10b_bypass   ),
604     .tx_char_dispmode_i   ( tx_char_dispmode   ),
605     .tx_char_dispval_i    ( tx_char_dispval    ),

```

```

606     .tx_buf_err_o      ( tx_buf_err      ),
607     .tx_resetdone_o   ( tx_resetdone   ),
608     .rx_detect_done_o ( rx_detect_done  ),
609     .rx_present_o     ( rx_present     ),
610
611     .TERM_SERIO_0     ( TERM_SERIO     ),
612
613     .rx_reset_i       ( rx_reset       ),
614     .rx_pma_reset_i   ( rx_pma_reset   ),
615     .rx_eqa_reset_i   ( rx_eqa_reset   ),
616     .rx_cdr_reset_i   ( rx_cdr_reset   ),
617     .rx_pcs_reset_i   ( rx_pcs_reset   ),
618     .rx_buf_reset_i   ( rx_buf_reset   ),
619     .rx_prbs_sel_i    ( rx_prbs_sel    ),
620     .rx_prbs_cnt_reset_i ( rx_prbs_cnt_reset ),
621     .rx_powerdown_n_i ( rx_powerdown_n ),
622     .rx_en_ei_detector_i ( rx_en_ei_detector ),
623     .rx_comma_detect_en_i ( rx_comma_detect_en ),
624     .rx_slide_i       ( rx_slide       ),
625     .rx_polarity_i    ( rx_polarity    ),
626     .rx_8b10b_en_i    ( rx_8b10b_en    ),
627     .rx_8b10b_bypass_i ( rx_8b10b_bypass ),
628     .rx_mcomma_align_i ( rx_mcomma_align ),
629     .rx_pcomma_align_i ( rx_pcomma_align ),
630     .rx_prbs_err_o    ( rx_prbs_err    ),
631     .rx_data_o        ( rx_data        ),
632     .rx_char_is_k_o   ( rx_char_is_k   ),
633     .rx_char_is_comma_o ( rx_char_is_comma ),
634     .rx_not_in_table_o ( rx_not_in_table ),
635     .rx_disp_err_o    ( rx_disp_err    ),
636     .rx_buf_err_o     ( rx_buf_err     ),
637     .rx_byte_is_aligned_o ( rx_byte_is_aligned ),
638     .rx_byte_realign_o ( rx_byte_realign ),
639     .rx_resetdone_o   ( rx_resetdone   ),
640     .rx_ei_en_o       ( rx_ei_en       ),
641 );
642
643 #ifdef PIPE
644
645 // PIPE Logic Instantiation
646 ccfpga_pipe_logic #(
647     .DATA_BYTES ( DATA_BYTES )
648 )
649 pipe_logic_inst (
650     .o_PCLK      ( o_PCLK      ),
651     .i_Reset     ( i_Reset     ),
652     .i_PowerDown ( i_PowerDown ),
653     .i_TxDetectRx ( i_TxDetectRx ),
654     .i_TxElecIdle ( i_TxElecIdle ),
655     .i_TxCompliance ( i_TxCompliance ),
656     // .i_TxSwing ( i_TxSwing ),
657     .i_RxPolarity ( i_RxPolarity ),
658     .o_RxValid    ( o_RxValid    ),
659     .o_PhyStatus  ( o_PhyStatus  ),
660     .o_RxElecIdle ( o_RxElecIdle ),
661     .o_RxStatus   ( o_RxStatus   ),
662     .i_TxData     ( i_TxData     ),
663     .i_TxDataK    ( i_TxDataK    ),
664     .o_RxData     ( o_RxData     ),
665     .o_RxDataK    ( o_RxDataK    ),
666     .o_tx_buf_err ( o_tx_buf_err ),
667     .o_clk_core_rx_rec ( o_clk_core_rx_rec ),
668     .i_rx_buf_reset ( i_rx_buf_reset ),
669     .o_rx_buf_err  ( o_rx_buf_err ),
670     .o_fsm_state_pipe ( o_fsm_state_pipe ),
671     .o_fsm_state_align ( o_fsm_state_align ),
672     .o_RxDataComma ( o_RxDataComma ),
673     .o_RxDataDispErr ( o_RxDataDispErr ),
674     .o_RxDataDecErr ( o_RxDataDecErr ),
675     .i_clk_core_pll ( clk_core_pll ),
676     .i_clk_core_rx_rec ( clk_core_rx_rec ),
677     .o_clk_core_tx  ( clk_core_tx  ),
678     .o_clk_core_rx  ( clk_core_rx  ),
679     .o_pll_reset    ( pll_reset    ),
680     .o_tx_reset     ( tx_reset     ),
681     .i_tx_reset_done ( tx_resetdone ),
682     .o_rx_reset     ( rx_reset     ),

```



```

683     .i_rx_reset_done      ( rx_resetdone      ),
684     .o_tx_pcs_reset      ( tx_pcs_reset      ),
685     .o_tx_pma_reset      ( tx_pma_reset      ),
686     .o_rx_pcs_reset      ( rx_pcs_reset      ),
687     .o_rx_pma_reset      ( rx_pma_reset      ),
688     .o_rx_cdr_reset      ( rx_cdr_reset      ),
689     .o_rx_eqa_reset      ( rx_eqa_reset      ),
690     .o_tx_powerdown_n    ( tx_powerdown_n    ),
691     .o_rx_powerdown_n    ( rx_powerdown_n    ),
692     .o_loopback          ( loopback          ),
693     .o_rx_prbs_sel       ( rx_prbs_sel       ),
694     .o_rx_prbs_cnt_reset ( rx_prbs_cnt_reset ),
695     .o_tx_prbs_sel       ( tx_prbs_sel       ),
696     .o_tx_prbs_force_err ( tx_prbs_force_err ),
697     .o_rx_buf_reset      ( rx_buf_reset      ),
698     .i_rx_buf_err        ( rx_buf_err        ),
699     .i_tx_buf_err        ( tx_buf_err        ),
700     .o_tx_data           ( tx_data           ),
701     .o_tx_char_is_k      ( tx_char_is_k      ),
702     .o_tx_char_dispmode  ( tx_char_dispmode  ),
703     .o_tx_char_dispval   ( tx_char_dispval   ),
704     .o_tx_8b10b_en       ( tx_8b10b_en       ),
705     .o_tx_8b10b_bypass   ( tx_8b10b_bypass   ),
706     .o_tx_polarity       ( tx_polarity       ),
707     .o_tx_elec_idle      ( tx_elec_idle      ),
708     .o_tx_detect_rx      ( tx_detect_rx      ),
709     .i_rx_detect_done    ( rx_detect_done    ),
710     .i_rx_present       ( rx_present       ),
711     .i_rx_data          ( rx_data          ),
712     .i_rx_char_is_k      ( rx_char_is_k      ),
713     .i_rx_char_is_comma  ( rx_char_is_comma  ),
714     .i_rx_disp_err       ( rx_disp_err       ),
715     .i_rx_not_in_table   ( rx_not_in_table   ),
716     .o_rx_8b10b_en       ( rx_8b10b_en       ),
717     .o_rx_8b10b_bypass   ( rx_8b10b_bypass   ),
718     .i_rx_byte_is_aligned ( rx_byte_is_aligned ),
719     .i_rx_byte_realign   ( rx_byte_realign   ),
720     .o_rx_mcomma_align   ( rx_mcomma_align   ),
721     .o_rx_pcomma_align   ( rx_pcomma_align   ),
722     .o_rx_comma_detect_en ( rx_comma_detect_en ),
723     .o_rx_slide          ( rx_slide          ),
724     .o_rx_polarity       ( rx_polarity       ),
725     .o_rx_en_ei_detector ( rx_en_ei_detector ),
726     .i_rx_ei_en         ( rx_ei_en         )
727 );
728
729 `endif
730
731 // Clock Generation
732
733 parameter CLKPERIOD_REF = 10.0; // ADPLL
734 parameter CLKPERIOD_REG = 8.0;  // Register
735 parameter CLKPERIOD_CFG = 40.0; // Config
736
737 initial clk_ref = 1'b1;
738 always #(CLKPERIOD_REF / 2) clk_ref = ~clk_ref; // 100 MHz
739
740 initial clk_reg = 1'b1;
741 always #(CLKPERIOD_REG / 2) clk_reg = ~clk_reg; // 125 MHz
742
743 initial clk_cfg = 1'b1;
744 always #(CLKPERIOD_CFG / 2) clk_cfg = ~clk_cfg; // 25 MHz
745
746 // Reset Generation (SerDes and Regfile)
747
748 initial // ADPLL
749 begin
750     reset_n = 1'b0;
751     #(CLKPERIOD_REF * 10 + 1);
752     reset_n = 1'b1;
753 end
754
755 initial // Register
756 begin
757     reset_reg_n = 1'b0;
758     #(CLKPERIOD_REG * 10 + 1);
759     reset_reg_n = 1'b1;

```

```

760     end
761
762     // Testbench Tasks and Functions
763
764     function real round(input real number);
765         begin
766             round = ( number < 0.0 ) ? $ceil(number - 0.5) : $floor(number + 0.5);
767         end
768     endfunction // round
769
770     task finalize();
771         begin
772             $display("");
773             if ( checks_done == 0 )
774                 begin
775                     $display(" # # # # # # # # # # # # # # # # ");
776                     $display(" # # # # # # # # # # # # # # # # ");
777                     $display(" # # # # # # # # # # # # # # # # ");
778                     $display(" # # # # # # # # # # # # # # # # ");
779                     $display(" # # # # # # # # # # # # # # # # ");
780                     // $display("\nRI_TESTBENCH:WARNING:SIMUNKNOWN: Test result Unknown");
781                 end
782             else if ( errors > 0 )
783                 begin
784                     $display(" ##### # # # # # # # # # # ");
785                     $display(" # # # # # # # # # # # # # # # # ");
786                     $display(" # # # # # # # # # # # # # # # # ");
787                     $display(" # # # # # # # # # # # # # # # # ");
788                     $display(" # # # # # # # # # # # # # # # # ");
789                     // $display("\nRI_TESTBENCH:ERROR:SIMFAIL: Test FAILED");
790                 end
791             else
792                 begin
793                     $display(" ##### # # # # # # # # # # ");
794                     $display(" # # # # # # # # # # # # # # # # ");
795                     $display(" # # # # # # # # # # # # # # # # ");
796                     $display(" # # # # # # # # # # # # # # # # ");
797                     $display(" # # # # # # # # # # # # # # # # ");
798                     // $display("\nRI_TESTBENCH:NOTE:SIMPASS: Test PASSEd");
799                     end // else: !if( errors > 0 )
800             $finish;
801         end
802     endtask // finalize
803
804     task writeCfg(input [7:0] addr,
805                 input [15:0] data);
806         begin
807             @(posedge clk_cfg); #1;
808             cfg_en = 1'b1;
809             @(posedge clk_cfg); #1;
810             @(posedge clk_cfg); #1;
811             cfgfile_valid = 1'b1;
812             cfgfile_di = data;
813             cfgfile_addr = addr;
814             @(posedge clk_cfg); #1;
815             cfgfile_valid = 1'b0;
816             cfgfile_di = 16'h0;
817             cfgfile_addr = 8'h0;
818             @(posedge clk_cfg); #1;
819             @(posedge clk_cfg); #1;
820             cfg_en = 1'b0;
821         end
822     endtask // writeCfg
823
824     task writeRegfile(input [7:0] addr,
825                    input [15:0] data,
826                    input [15:0] mask);
827         integer cnt;
828         begin
829             @(posedge clk_reg); #1.5;
830             regfile_en = 1'b1;
831             regfile_we = 1'b1;
832             regfile_addr = addr;
833             regfile_mask = mask;
834             regfile_di = data;
835             @(posedge clk_reg); #1.5;
836             regfile_en = 1'b0;

```

```

837     regfile_we   = 1'b0;
838     regfile_addr = 8'h0;
839     regfile_mask = 16'h0;
840     regfile_di   = 16'h0;
841     while ( regfile_rdy != 1'b1 )
842     begin
843         @(posedge clk_reg); #1.5;
844         cnt = cnt + 1;
845         if ( cnt == 32 )
846             begin
847                 checks_done = checks_done + 1;
848                 $display("ERROR: Waiting for Register File ready timed out.");
849                 errors = errors + 1;
850             end
851         end // while ( regfile_rdy != 1'b1 )
852     end
853 endtask // writeRegfile
854
855 task readRegfile(input  [7:0] addr,
856                 output [15:0] data);
857     integer cnt;
858     begin
859         @(posedge clk_reg); #1;
860         regfile_en   = 1'b1;
861         regfile_we   = 1'b0;
862         regfile_addr = addr;
863         @(posedge clk_reg); #1;
864         regfile_en   = 1'b0;
865         regfile_addr = 8'h0;
866         cnt = 0;
867         while ( regfile_rdy != 1'b1 )
868             begin
869                 @(posedge clk_reg); #1;
870                 cnt = cnt + 1;
871                 if ( cnt == 32 )
872                     begin
873                         checks_done = checks_done + 1;
874                         $display("ERROR: Waiting for Register File ready timed out.");
875                         errors = errors + 1;
876                     end
877                 end // while ( regfile_rdy != 1'b1 )
878             data = regfile_do;
879         end
880 endtask // readRegfile
881
882 task writeCfgFile(input  [7:0] addr,
883                  input [15:0] data);
884     begin
885         @(posedge clk_cfg); #1;
886         cfg_en       = 1'b1;
887         cfgfile_valid = 1'b1;
888         cfgfile_addr  = addr;
889         cfgfile_di     = data;
890         @(posedge clk_cfg); #1;
891         cfg_en       = 1'b0;
892         cfgfile_valid = 1'b0;
893         cfgfile_addr  = 1'b0;
894         cfgfile_di     = 1'b0;
895     end
896 endtask // writeCfgFile
897
898 task readSerIOADPLLStatus(output [31:0] status);
899     reg [31:0] pll_status;
900     begin
901         readRegfile(8'h55, pll_status[15:0]);
902         readRegfile(8'h56, pll_status[31:16]);
903         checks_done = checks_done + 1;
904         if ( (~pll_status) == 1'bx )
905             begin
906                 $display("ERROR: SerIO ADPLL status is invalid.");
907                 errors = errors + 1;
908                 finalize;
909             end
910         status = pll_status;
911     end
912 endtask // readSerIOADPLLStatus
913

```

```

914 task startSerIOADPLL(input integer mainDivN1, // N1 and N2 Names are twisted compared to Spec
915                    input integer mainDivN2,
916                    input integer mainDivN3,
917                    input integer outDiv,
918                    input enableCalib);
919     real    dcoFreq;
920     real    freq;
921     reg [7:0] unit;
922     reg [15:0] pllDiv;
923     reg [31:0] status;
924     reg [31:0] bisc_result;
925     time    start;
926     integer i;
927     begin
928         $display("INFO: Configuring SerIO ADPLL ...");
929
930         checks_done = checks_done + 1;
931         if ( mainDivN1 < 2 || mainDivN1 > 5 )
932             begin
933                 $display("ERROR: Main divider N1 of SerIO ADPLL is limited to 2, 3, 4 or 5.");
934                 errors = errors + 1;
935                 finalize;
936             end
937
938         checks_done = checks_done + 1;
939         if ( mainDivN2 < 1 || mainDivN2 > 2 )
940             begin
941                 $display("ERROR: Main divider N2 of SerIO ADPLL is limited to 1 or 2.");
942                 errors = errors + 1;
943                 finalize;
944             end
945
946         checks_done = checks_done + 1;
947         if ( mainDivN3 < 3 || mainDivN3 > 5 )
948             begin
949                 $display("ERROR: Main divider N3 of SerIO ADPLL is limited to 3, 4 or 5.");
950                 errors = errors + 1;
951                 finalize;
952             end
953
954         checks_done = checks_done + 1;
955         if ( outDiv != 1 && outDiv != 2 && outDiv != 4 )
956             begin
957                 $display("ERROR: Output divider of SerIO ADPLL is limited to 1, 2 or 4.");
958                 errors = errors + 1;
959                 finalize;
960             end
961
962         dcoFreq = 1000.0 / CLKPERIOD_REF * mainDivN1 * mainDivN2 * mainDivN3;
963         freq = dcoFreq / outDiv;
964         unit = "M";
965         if ( $rtoi(round(freq * 1000.0)) > 1000000 )
966             begin
967                 freq = freq / 1000;
968                 unit = "C";
969             end
970
971         $display("INFO: SerIO ADPLL frequency will be %.3f %cHz.",
972               freq, unit);
973         pllDiv = ( ( outDiv == 1 ) ? 16'h0000 :
974                 ( outDiv == 2 ) ? 16'h1000 : 16'h3000 );
975         if ( mainDivN1 == 5 )
976             pllDiv[7:6] = 2'b11;
977         else if ( mainDivN1 == 4 )
978             pllDiv[7:6] = 2'b10;
979         else if ( mainDivN1 == 2 )
980             pllDiv[7:6] = 2'b01;
981
982         if ( mainDivN2 == 2 )
983             pllDiv[8] = 1'b1;
984
985         if ( mainDivN3 == 5 )
986             pllDiv[10:9] = 2'b11;
987         else if ( mainDivN3 == 4 )
988             pllDiv[10:9] = 2'b10;
989
990         $display("INFO: Checking state of SerIO ADPLL ...");

```

```

991     readSerIOADPLLStatus(status);
992     if ( status[0] == 1'b1 )
993     begin
994         $display("INFO: Disabling SerIO ADPLL ...");
995         writeRegfile(8'h50, 16'h0000, 16'h0001);
996         for ( i = 0; i < 100; i = i + 1 )
997             begin
998                 @(posedge clk_reg); #1;
999             end
1000     end
1001
1002     $display("INFO: Writing SerIO ADPLL divider settings ...");
1003     writeRegfile(8'h51, pllDiv, 16'h3FC0);
1004     if ( enableCalib == 1'b1 )
1005     begin
1006         $display("INFO: Stopping ADPLL self-calibration ...");
1007         writeRegfile(8'h57, 16'h0004, 16'h0007);
1008         writeRegfile(8'h57,
1009             ( ( ADPLL_PFDAC_TIMER      & 16'h000F ) << 3 ) |
1010             ( ( ADPLL_PFDAC_COR_DLY    & 16'h0007 ) << 10 ) |
1011             ( ( ADPLL_PFDAC_CAL_SIGN   & 16'h0001 ) << 13 ) |
1012             ( ( ADPLL_PFDAC_AUTO_CAL   & 16'h0001 ) << 14 ) ,
1013             16'hFFF8);
1014         writeRegfile(16'h58,
1015             ( ( ADPLL_PFDAC_CP_MIN     & 16'h001F ) << 0 ) |
1016             ( ( ADPLL_PFDAC_CP_MAX     & 16'h001F ) << 5 ) |
1017             ( ( ADPLL_PFDAC_CP_START   & 16'h001F ) << 10 ) ,
1018             16'hFFFF);
1019     end // if ( enableCalib == 1'b1 )
1020
1021     $display("INFO: Starting SerIO ADPLL ...");
1022     writeRegfile(8'h50, 16'h0002, 16'h0007); // Config Sel
1023     writeRegfile(8'h50, 16'h0003, 16'h0003); // Config Sel + ADPLL Enable
1024
1025     if ( enableCalib == 1'b1 )
1026     begin
1027         $display("INFO: Starting ADPLL self-calibration ...");
1028         writeRegfile(8'h57, 16'h0004, 16'h0007);
1029         writeRegfile(8'h57, 16'h0005, 16'h0007);
1030     end
1031
1032     $display("INFO: Waiting for SerIO ADPLL to lock ...");
1033     start = $realtime;
1034     status = 16'h0;
1035     while ( status[0] == 1'b0 )
1036     begin
1037         for ( i = 0; i < 1000; i = i + 1 )
1038             begin
1039                 @(posedge clk_reg); #1;
1040             end
1041         readSerIOADPLLStatus(status);
1042         if ( status[0] == 1'b0 )
1043             $display("INFO: LCK: %1d FTO: %1d FTU: %1d FT: %4d SY: %3d ST: %1d",
1044                 status[0], status[1], status[2],
1045                 status[12:3], status[23:16], status[14:13]);
1046         if ( ( $realtime - start ) > 2000000 && status[0] == 1'b0 )
1047             begin
1048                 checks_done = checks_done + 1;
1049                 $display("ERROR: Waiting for SerIO ADPLL lock timed out.");
1050                 errors = errors + 1;
1051                 finalize;
1052             end
1053     end // while ( status[0] == 1'b0 )
1054
1055     if ( enableCalib == 1'b1 )
1056     begin
1057         readRegfile(8'h5A, bisc_result[15:0]);
1058         readRegfile(8'h5B, bisc_result[31:16]);
1059         if ( (~bisc_result) == 1'bx )
1060             begin
1061                 $display("ERROR: SerIO ADPLL BISC result is invalid.");
1062                 errors = errors + 1;
1063                 finalize;
1064             end
1065
1066     $display("INFO: PFDAC Result: Done: %1d, Counter: %6d, CP: %2d",
1067         bisc_result[0], bisc_result[31:16],

```

```

1068         bisc_result[7:1]);
1069     end // if ( enableCalib == 1'b1 )
1070
1071     $display("INFO: SerIO ADPLL locked.");
1072     $display("INFO: LCK: %1d FTO: %1d FTU: %1d FT: %4d SY: %3d ST: %1d",
1073         status[0], status[1], status[2],
1074         status[12:3], status[23:16], status[14:13]);
1075
1076     end
1077 endtask // startSerIOADPLL
1078
1079 `ifdef SERDES
1080
1081 task resetSerDes();
1082     integer cnt;
1083     reg done;
1084     begin
1085         tx_reset = 1'b1;
1086         cnt = 0;
1087         done = 1'b0;
1088         @(posedge clk_core_tx); #1;
1089         while ( done == 1'b0 )
1090             begin
1091                 if ( tx_resetdone === 1'b0 )
1092                     done = 1'b1;
1093                 else
1094                     begin
1095                         checks_done = checks_done + 1;
1096                         if ( tx_resetdone === 1'bx )
1097                             begin
1098                                 $display("ERROR: TX Resetdone indicator is invalid.");
1099                                 errors = errors + 1;
1100                                 finalize;
1101                             end
1102                         cnt = cnt + 1;
1103                         if ( cnt == 64 )
1104                             begin
1105                                 checks_done = checks_done + 1;
1106                                 $display("ERROR: Waiting for TX Resetdone release timed out.");
1107                                 errors = errors + 1;
1108                                 finalize;
1109                             end
1110                         end // else: !if( tx_resetdone === 1'b0 )
1111                         @(posedge clk_core_tx); #1;
1112                     end // while ( done == 1'b0 )
1113
1114         tx_reset = 1'b0;
1115         cnt = 0;
1116         done = 1'b0;
1117         while ( done == 1'b0 )
1118             begin
1119                 if ( tx_resetdone === 1'b1 )
1120                     done = 1'b1;
1121                 else
1122                     begin
1123                         checks_done = checks_done + 1;
1124                         if ( tx_resetdone === 1'bx )
1125                             begin
1126                                 $display("ERROR: TX Resetdone indicator is invalid.");
1127                                 errors = errors + 1;
1128                                 finalize;
1129                             end
1130                         cnt = cnt + 1;
1131                         if ( cnt == 64 )
1132                             begin
1133                                 checks_done = checks_done + 1;
1134                                 $display("ERROR: Waiting for TX Resetdone set timed out.");
1135                                 errors = errors + 1;
1136                                 finalize;
1137                             end
1138                         end // else: !if( tx_resetdone === 1'b1 )
1139                         @(posedge clk_core_tx); #1;
1140                     end // while ( done == 1'b0 )
1141
1142         rx_reset = 1'b1;
1143         cnt = 0;
1144         done = 1'b0;

```

```

1145     @(posedge clk_core_rx_in); #1;
1146     while ( done == 1'b0 )
1147     begin
1148         if ( rx_resetdone === 1'b0 )
1149             done = 1'b1;
1150         else
1151             begin
1152                 checks_done = checks_done + 1;
1153                 if ( rx_resetdone === 1'bx )
1154                     begin
1155                         $display("ERROR: RX Resetdone indicator is invalid.");
1156                         errors = errors + 1;
1157                         finalize;
1158                     end
1159                 cnt = cnt + 1;
1160                 if ( cnt == 64 )
1161                     begin
1162                         checks_done = checks_done + 1;
1163                         $display("ERROR: Waiting for RX Resetdone release timed out.");
1164                         errors = errors + 1;
1165                         finalize;
1166                     end
1167                 end // else: !if( rx_resetdone === 1'b0 )
1168             @(posedge clk_core_rx_in); #1;
1169             end // while ( done == 1'b0 )
1170
1171     rx_reset = 1'b0;
1172     cnt = 0;
1173     done = 1'b0;
1174     while ( done == 1'b0 )
1175     begin
1176         if ( rx_resetdone === 1'b1 )
1177             done = 1'b1;
1178         else
1179             begin
1180                 checks_done = checks_done + 1;
1181                 if ( rx_resetdone === 1'bx )
1182                     begin
1183                         $display("ERROR: RX Resetdone indicator is invalid.");
1184                         errors = errors + 1;
1185                         finalize;
1186                     end
1187                 cnt = cnt + 1;
1188                 if ( cnt == 128 )
1189                     begin
1190                         checks_done = checks_done + 1;
1191                         $display("ERROR: Waiting for RX Resetdone set timed out.");
1192                         errors = errors + 1;
1193                         finalize;
1194                     end
1195                 end // else: !if( rx_resetdone === 1'b1 )
1196             @(posedge clk_core_rx_in); #1;
1197             end // while ( done == 1'b0 )
1198     end
1199     endtask // resetSerDes
1200
1201 'endif
1202
1203 // Include Testcases
1204
1205 'ifdef PIPE
1206     'include "testcase_pipe.v"
1207 'elsif SERDES
1208     'include "testcase_serdes.v"
1209 'else
1210     $display("ERROR: TESTCASE NOT DEFINED")
1211 'endif
1212
1213 endmodule // tb_ccfpga_serdes

```

A.5.2 testcase_pipe.v

```

37 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
38 //
39 // Interessengruppe fuer Mikroelektronik und Eingebettete Systeme (IMES)
40 // Fachhochschule Dortmund
41 //
42 // Development in cooperation with Cologne Chip AG
43 //
44 // Filename      : testcase_pipe.v
45 // Author       : Philipp Leduc
46 // Tool        :
47 // Description  : Testcase for PIPE Interface of the GateMate FPGA.
48 // Commentary  : The testcase is meant for usage with the modified SerDes Testbench.
49 //               Abbreviations: [i_] > input,
50 //                               [o_] > output,
51 //                               [_n] > low active
52 //
53 // Changelog:
54 // -----
55 // Version | Author          | Date       | Changes
56 // -----
57 // 1.0     | Leduc                | 05.06.2021 | released
58 // -----
59 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
60
61
62 // Connect Tx and Rx Line (incl. Delay)
63
64 //always @(TX_SERIO_P) RX_SERIO_P <= #10 TX_SERIO_P;
65 //always @(TX_SERIO_N) RX_SERIO_N <= #10 TX_SERIO_N;
66
67 reg S_FAULT_INJECTION = 1'b0;
68
69 always @(TX_SERIO_P) begin
70
71     if ( S_FAULT_INJECTION == 1'b1 )
72         RX_SERIO_P <= #10 ~TX_SERIO_P;
73     else
74         RX_SERIO_P <= #10 TX_SERIO_P;
75
76     end // always
77
78 always @(TX_SERIO_N) begin
79
80     if ( S_FAULT_INJECTION == 1'b1 )
81         RX_SERIO_N <= #10 ~TX_SERIO_N;
82     else
83         RX_SERIO_N <= #10 TX_SERIO_N;
84
85     end // always
86
87 // Connect ADPLL Core Clock to Tx/Rx Datapath (happens now inside PIPE logic)
88 //assign clk_core_tx   = clk_core_pll;
89 //assign clk_core_rx_in = clk_core_pll;
90
91 // Reset Connection (not Fabric)
92
93 assign reset_core_tx_n = reset_core_pll_n;
94 assign reset_core_rx_n = reset_core_pll_n;
95
96 // Variables
97 //reg [2:0] alignment      = 3'b0;
98 //reg [63:0] recv_data     = 64'h0;
99 //reg [15:0] rterm_status  = 16'h0;
100 //reg      align_done_flag = 1'b0;
101 reg      transmission_start_flag = 1'b0;
102
103 reg      status_flag_1 = 1'b0;    // Status flags for simulation
104 reg      status_flag_2 = 1'b0;
105 reg      status_flag_3 = 1'b0;
106 reg      init_status_flag_done = 1'b0;
107 reg      s_fault_flag_for_rx = 1'b0;
108

```



```

109 integer i;
110 integer j;
111 integer k;
112 integer comma_pos;
113
114 // Testdata Arrays
115 reg [ 7:0] testdata    [39:0];
116 reg      testdata_k   [39:0];
117 reg      testdata_com [39:0];
118
119
120 // Functions and Tasks
121
122 task generate_data ();
123     integer i;
124     integer j;
125     integer start_pos;
126     integer pos_k;
127
128     begin
129
130         // fill array with data
131         for ( i = 0; i < 40; i = i + 1 ) begin
132             testdata[i] = $urandom%255;
133             testdata_k[i] = 1'b0;
134             testdata_com[i] =1 'b0;
135         end
136
137         // implement control bytes
138         start_pos = $urandom%4;
139
140         for ( j = start_pos; j < 39; j = j + 4 ) begin
141             pos_k = $urandom%11;
142
143             testdata[j] = setControlByte(pos_k);
144
145             testdata_k[j] = 1'b1;
146
147             // Check for COM Symbol
148             if      ( testdata[j] == 8'hBC )
149                 testdata_com [j] = 1'b1;
150             else if ( testdata[j] == 8'h3C )
151                 testdata_com [j] = 1'b1;
152             else if ( testdata[j] == 8'hFC )
153                 testdata_com [j] = 1'b1;
154             else
155                 testdata_com [j] =1'b0;
156         end
157     end
158 endtask
159
160 function [7:0] setControlByte(input integer pos);
161     begin
162         setControlByte = ( pos == 0 ) ?
163             8'h1C :
164             ( pos == 1 ) ?
165             8'h3C :
166             ( pos == 2 ) ?
167             8'h5C :
168             ( pos == 3 ) ?
169             8'h7C :
170             ( pos == 4 ) ?
171             8'h9C :
172             ( pos == 5 ) ?
173             8'hBC :
174             ( pos == 6 ) ?
175             8'hDC :
176             ( pos == 7 ) ?
177             8'hFC :
178             ( pos == 8 ) ?
179             8'hF7 :
180             ( pos == 9 ) ?
181             8'hFB :
182             ( pos == 10 ) ?
183             8'hFD :
184             8'hFE;
185     end

```

```

186 endfunction
187
188 task send_testdata (input integer bytes);
189     integer k;
190     // integer i;
191     begin
192
193         if ( bytes == 2 ) begin // PIPE 16-Bit
194
195             for ( k = 0; k < 40; k = k + 2 ) begin
196                 // i = k + 1;
197                 @(posedge o_PCLK); #1;
198                 i_TxData = {testdata[k], testdata[k+1]};
199                 i_TxDataK = {testdata_k[k], testdata_k[k+1]};
200             end
201         end
202         else begin // PIPE 64-Bit
203             for ( k = 0; k < 40; k = k + 8 ) begin
204                 //i = k + 1;
205                 @(posedge o_PCLK); #1;
206                 i_TxData = { testdata[k+7],testdata[k+6],testdata[k+5],testdata[k+4],
207                             testdata[k+3],testdata[k+2],testdata[k+1],testdata[k]};
208                 i_TxDataK = { testdata_k[k+7],testdata_k[k+6],testdata_k[k+5],testdata_k[k+4],
209                             testdata_k[k+3],testdata_k[k+2],testdata_k[k+1],testdata_k[k]};
210             end
211         end
212
213         @(posedge o_PCLK); #1;
214         i_TxData = {DATA_WIDTH{1'b0}};
215         i_TxDataK = {DATA_BYTES{1'b0}};
216     end
217 endtask
218
219 task send_TS1_OS (input integer bytes);
220     integer k;
221     integer j;
222     begin
223
224         if ( bytes == 2 ) begin // PIPE 16-Bit
225             for ( k = 0; k < 1023; k = k + 1 ) begin // Create TS1 Ordered Set
226                 @(posedge o_PCLK); #1;
227
228                 i_TxData = 16'h01BC; // TS1 first two bytes
229                 i_TxDataK = 2'b01;
230
231                 for ( j = 0; j < 15; j = j + 1 ) begin
232                     #4;
233                     i_TxData = 16'h23CA;
234                     i_TxDataK = 8'b00;
235                 end
236             end
237         end
238         else begin // PIPE 64-Bit
239             for ( k = 0; k < 1024; k = k + 1 ) begin // Create TS1 Ordered Set
240                 @(posedge o_PCLK); #1;
241
242                 i_TxData = 64'h0802_0100_012A_01BC; // TS1 first part
243                 i_TxDataK = 8'b0000_0001;
244                 #32;
245                 i_TxData = 64'h162B_F45F_238A_0103; // TS1 second part
246                 i_TxDataK = 8'b00_00_00_00;
247             end
248         end
249         // i_TxData = 64'h0000_0000_0000_0000;
250         // i_TxDataK = 8'b0000_0000;
251     end
252 endtask
253
254 task check_testdata (input integer bytes);
255     integer k;
256     reg [15:0] testdata_16;
257     reg [1:0] testdata_k_16;
258     reg [1:0] testdata_com_16;
259     reg [63:0] testdata_64;
260     reg [7:0] testdata_k_64;
261     reg [7:0] testdata_com_64;
262     // integer i;

```

```

263     begin
264
265     $display("-----");
266     $display(" Beginning to check Data against Testdata ");
267     $display("-----");
268
269     if ( bytes == 2) begin // PIPE 16-Bit
270
271
272
273         for ( k = 0; k < 40; k = k + 2 ) begin
274             @(posedge o_PCLK); #1;
275
276             testdata_16 = {testdata[k], testdata[k+1]};
277
278             testdata_k_16 = {testdata_k[k], testdata_k[k+1]};
279
280             testdata_com_16 = {testdata_com[k], testdata_com[k+1]};
281
282             // Check RxData
283             if ( o_RxData != testdata_16 ) begin
284
285                 checks_done = checks_done + 1;
286                 $display("ERROR: Received Data does not match Test Data.");
287                 $display("   o_RxData : %h", o_RxData);
288                 $display("   Testdata : %h", testdata_16);
289                 errors = errors + 1;
290                 finalize;
291             end
292             else begin
293                 checks_done = checks_done + 1;
294                 $display("   o_RxData : %h", o_RxData);
295                 $display("   Testdata : %h", testdata_16);
296                 $display("");
297             end
298
299             // Check RxDataK
300             if ( o_RxDataK != testdata_k_16 ) begin
301
302                 checks_done = checks_done + 1;
303                 $display("ERROR: Received K Data does not match Test Data.");
304                 $display("   o_RxDataK : %b", o_RxDataK);
305                 $display("   Testdata : %b", testdata_k_16);
306                 errors = errors + 1;
307                 finalize;
308             end
309             else begin
310                 checks_done = checks_done + 1;
311                 $display("   o_RxDataK : %b", o_RxDataK);
312                 $display("   Testdata : %b", testdata_k_16);
313                 $display("");
314             end
315
316             // Check o_RxDataComma
317             if ( o_RxDataComma != testdata_com_16 ) begin
318
319                 checks_done = checks_done + 1;
320                 $display("ERROR: Received COM Data does not match Test Data.");
321                 $display("o_RxDataComma : %b", o_RxDataComma);
322                 $display("   Testdata : %b", testdata_com_16);
323                 errors = errors + 1;
324                 finalize;
325             end
326             else begin
327                 checks_done = checks_done + 1;
328                 $display("o_RxDataComma : %b", o_RxDataComma);
329                 $display("   Testdata : %b", testdata_com_16);
330                 $display("-----");
331             end
332         end // for loop
333     end // if
334
335     else begin // PIPE 64-Bit
336
337         for ( k = 0; k < 40; k = k + 8 ) begin
338             @(posedge o_PCLK); #1;
339

```

```

340     testdata_64 = { testdata[k+7],testdata[k+6],testdata[k+5],
341                   testdata[k+4],testdata[k+3],testdata[k+2],
342                   testdata[k+1],testdata[k] };
343
344     testdata_k_64 = { testdata_k[k+7],testdata_k[k+6],testdata_k[k+5],
345                     testdata_k[k+4],testdata_k[k+3],testdata_k[k+2],
346                     testdata_k[k+1],testdata_k[k] };
347
348     testdata_com_64 = { testdata_com[k+7],testdata_com[k+6],testdata_com[k+5],
349                       testdata_com[k+4],testdata_com[k+3],testdata_com[k+2],
350                       testdata_com[k+1],testdata_com[k] };
351
352     // Check RxData
353     if ( o_RxData != testdata_64 ) begin
354         checks_done = checks_done + 1;
355         $display("ERROR: Received Data does not match Test Data.");
356         $display("    o_RxData : %h", o_RxData);
357         $display("    Testdata : %h", testdata_64);
358         errors = errors + 1;
359         finalize;
360         end
361     else begin
362         checks_done = checks_done + 1;
363         $display("    o_RxData : %h", o_RxData);
364         $display("    Testdata : %h", testdata_64);
365         $display("");
366     end
367
368     // Check RxDataK
369     if ( o_RxDataK != testdata_k_64 ) begin
370         checks_done = checks_done + 1;
371         $display("ERROR: Received K Data does not match Test Data.");
372         $display("    o_RxDataK : %b", o_RxDataK);
373         $display("    Testdata : %b", testdata_k_64);
374         errors = errors + 1;
375         finalize;
376         end
377     else begin
378         checks_done = checks_done + 1;
379         $display("    o_RxDataK : %b", o_RxDataK);
380         $display("    Testdata : %b", testdata_k_64);
381         $display("");
382     end
383
384     // Check o_RxDataComma
385     if ( o_RxDataComma != testdata_com_64 ) begin
386         checks_done = checks_done + 1;
387         $display("ERROR: Received COM Data does not match Test Data.");
388         $display("o_RxDataComma : %b", o_RxDataComma);
389         $display("    Testdata : %b", testdata_com_64);
390         errors = errors + 1;
391         finalize;
392         end
393     else begin
394         checks_done = checks_done + 1;
395         $display("o_RxDataComma : %b", o_RxDataComma);
396         $display("    Testdata : %b", testdata_com_64);
397         $display("_____");
398         $display("");
399     end
400     end // for loop
401 end // else
402
403 $display("INFO: Task check_testdata completed without Errors.");
404 $display("");
405
406 end // task
407 endtask
408
409
410 task force_transm_err (input integer bit_error_cnt);
411     integer i;
412     begin
413         for (i = 0; i < bit_error_cnt ; i = i + 1) begin
414             s_fault_flag_for_rx <= #10 1'b1;
415             S_FAULT_INJECTION <= 1'b1;
416             #0.4;

```

```

417         S_FAULT_INJECTION <= 1'b0;
418         s_fault_flag_for_rx <= #10 1'b0;
419         #4.5;
420     end
421 end // task
422 endtask
423
424
425 task check_err_detect (input integer checks);
426     integer k;
427     integer i;
428     integer j;
429
430     begin
431
432         // Included Delay Time
433         #450;
434
435         $display("-----");
436         $display(" Beginning to check Error Detection and RxStatus ");
437         $display("-----");
438
439         for ( k = 0; k < checks; k = k + 1 ) begin
440             @(posedge o_PCLK); #1;
441
442             // Check 8b/10b Error
443             if ( o_RxDataDecErr != rx_not_in_table) begin
444                 checks_done = checks_done + 1;
445                 $display("ERROR: 8b/10b Error of SerDes and PIPE do not match.");
446                 $display(" o_RxDataDecErr : %b", o_RxDataDecErr);
447                 $display("rx_not_in_table : %b", rx_not_in_table);
448                 errors = errors + 1;
449                 finalize;
450             end
451         else begin
452             checks_done = checks_done + 1;
453             $display(" o_RxDataDecErr : %b", o_RxDataDecErr);
454             $display("rx_not_in_table : %b", rx_not_in_table);
455             $display("");
456         end
457
458         // Check Disparity Error
459         if ( o_RxDataDispErr != rx_disp_err ) begin
460
461             checks_done = checks_done + 1;
462             $display("ERROR: Disparity Error of SerDes and PIPE do not match.");
463             $display("o_RxDataDispErr : %b", o_RxDataDispErr);
464             $display(" rx_disp_err : %b", rx_disp_err);
465             errors = errors + 1;
466             finalize;
467         end
468     else begin
469         checks_done = checks_done + 1;
470         $display("o_RxDataDispErr : %b", o_RxDataDispErr);
471         $display(" rx_disp_err : %b", rx_disp_err);
472         $display("");
473     end
474
475     // Check RxStatus and RxData (8b/10b)
476     if ( |o_RxDataDecErr == 1'b1 ) begin
477
478         if (o_RxStatus != 3'b100 ) begin
479             checks_done = checks_done + 1;
480             $display("ERROR: RxStatus does not match Error Detection.");
481             $display(" o_RxStatus : %b", o_RxDataComma);
482             $display(" Should be : 100");
483             errors = errors + 1;
484             finalize;
485         end
486     else begin
487         checks_done = checks_done + 1;
488         $display(" o_RxStatus : %b", o_RxStatus);
489         $display("");
490     end
491
492     // Check EDB Symbol Insertion
493     j = 7;

```

```

494         for (i = 0; i < 8; i = i + 1) begin
495             if ( o_RxDataDecErr[i] == 1'b1 ) begin
496                 if ( o_RxData[j -: 8] != 8'hFE ) begin
497                     checks_done = checks_done + 1;
498                     $display("ERROR: EDB Symbol not inserted for 8b/10b Error.");
499                     $display("    o_RxData : %h", o_RxData);
500                     $display(" o_RxDataDecErr : %b", o_RxDataDecErr);
501                     errors = errors + 1;
502                     finalize;
503                     end
504                 else begin
505                     checks_done = checks_done + 1;
506                     $display("_____");
507                     $display(" INFO: EDB Symbol inserted for 8b/10b Error. ");
508                     $display("_____");
509                     $display("");
510                     $display("    o_RxData[%0d:%0d]: %h",j,j-7,o_RxData[j -: 8]);
511                     $display("");
512                     $display("_____");
513                     end
514                 end
515                 j = j + 8;
516             end
517         end
518
519     else if ( |o_RxDataDispErr == 1'b1) begin
520
521         if (o_RxStatus != 3'b111 ) begin
522             checks_done = checks_done + 1;
523             $display("ERROR: RxStatus does not match Error Detection.");
524             $display("    o_RxStatus : %b", o_RxDataComma);
525             $display("    Should be : 111");
526             errors = errors + 1;
527             finalize;
528             end
529         else begin
530             checks_done = checks_done + 1;
531             $display("    o_RxStatus : %b", o_RxStatus);
532             $display("_____");
533             end
534         end
535
536     else if (|o_RxDataDispErr == 1'b0 && |o_RxDataDecErr == 1'b0) begin
537
538         if (o_RxStatus != 3'b000 ) begin
539             checks_done = checks_done + 1;
540             $display("ERROR: RxStatus does not match Error Detection.");
541             $display("    o_RxStatus : %b", o_RxDataComma);
542             $display("    Should be : 000");
543             errors = errors + 1;
544             finalize;
545             end
546         else begin
547             checks_done = checks_done + 1;
548             $display("    o_RxStatus : %b", o_RxStatus);
549             $display("_____");
550             end
551         end
552
553     else begin
554         $display("ERROR: RxStatus in unknown.");
555     end
556
557     end // for loop
558
559     $display("");
560     $display("INFO: Task check_err_detect completed without Errors.");
561     $display("");
562
563     end // task
564 endtask
565
566
567 // Testcase
568
569 initial
570     begin

```

```

571
572   $sprinttimescale(tb_ccfpga_serdes);
573
574
575   i_Reset = 1'b0; // Low active
576
577   // Reset Values PIPE Control Signals
578
579   i_PowerDown      = 2'b10;
580   i_TxDetectRx     = 1'b0;
581   i_TxElecIdle     = 1'b1;
582   i_TxCompliance   = {DATA_BYTES{1'b0}};
583   i_RxPolarity     = 1'b0;
584   i_TxData         = {DATA_WIDTH{1'b0}};
585   i_TxDataK        = {DATA_BYTES{1'b0}};
586
587   i_rx_buf_reset   = 1'b0;
588
589   #20;
590
591   i_Reset = 1'b1; // Release Reset
592
593   // Wait for different Resets to be done.
594
595   #500;
596
597   writeCfg(8'h5C, 16'h0001); // Enable Serdes
598
599   #100;
600
601   writeRegfile(8'h50, 16'h0002, 16'h0007); // Config Sel
602   writeRegfile(8'h50, 16'h0003, 16'h0003); // Config Sel + Adpll Enable
603
604   // Alternative Method
605   //startSerIOADPLL(5, 1, 5, 2, 1'b0); // 1,25 GHz -> 2,5Gb/s (PCIe)
606
607
608   // Wait for PhyStatus
609
610   #2000;
611
612   if (o_PhyStatus == 1'b1) begin
613     checks_done = checks_done + 1;
614     $display("ERROR: PhyStatus did not signal successive Reset");
615     errors = errors + 1;
616     finalize;
617   end
618   else if (o_PhyStatus == 1'b0) begin
619     checks_done = checks_done + 1;
620     $display("INFO: PhyStatus turned to 0 after Reset");
621   end
622   else begin
623     checks_done = checks_done + 1;
624     $display("ERROR: PhyStatus is unknown");
625     errors = errors + 1;
626     finalize;
627   end
628
629   // Start of Powerstate Transition Tests
630
631   // Transit to Powerstate P0 (IDLE)
632
633   @(posedge o_PCLK);
634
635   i_PowerDown      = 2'b00; // P0
636   i_TxDetectRx     = 1'b0;
637   i_TxElecIdle     = 1'b1; // Elec Idle
638   i_TxCompliance   = {DATA_BYTES{1'b0}};
639   i_RxPolarity     = 1'b0;
640   i_TxData         = {DATA_WIDTH{1'b0}};
641   i_TxDataK        = {DATA_BYTES{1'b0}};
642
643   #128;
644
645   // Transit to Powerstate P1
646
647   @(posedge o_PCLK);

```

```

648
649     i_PowerDown      = 2'b10;      // P1
650     i_TxDetectRx    = 1'b0;
651     i_TxElecIdle    = 1'b1;      // Elec Idle
652     i_TxCompliance  = {DATA_BYTES{1'b0}};
653     i_RxPolarity    = 1'b0;
654     i_TxData        = {DATA_WIDTH{1'b0}};
655     i_TxDataK       = {DATA_BYTES{1'b0}};
656
657     #128;
658
659     // Transit to Powerstate P0 (Normal Transmission)
660
661     @(posedge o_PCLK);
662
663     i_PowerDown      = 2'b00;      // P0
664     i_TxDetectRx    = 1'b0;
665     i_TxElecIdle    = 1'b0;      // Elec Idle
666     i_TxCompliance  = {DATA_BYTES{1'b0}};
667     i_RxPolarity    = 1'b0;
668     i_TxData        = {DATA_WIDTH{1'b0}};
669     i_TxDataK       = {DATA_BYTES{1'b0}};
670
671     #128;
672
673     // Transit to Powerstate P1
674
675     @(posedge o_PCLK);
676
677     i_PowerDown      = 2'b10;      // P1
678     i_TxDetectRx    = 1'b0;
679     i_TxElecIdle    = 1'b1;      // Elec Idle
680     i_TxCompliance  = {DATA_BYTES{1'b0}};
681     i_RxPolarity    = 1'b0;
682     i_TxData        = {DATA_WIDTH{1'b0}};
683     i_TxDataK       = {DATA_BYTES{1'b0}};
684
685     #256;
686
687     // Transit to Powerstate P0 (Loopback)
688
689     @(posedge o_PCLK);
690
691     i_PowerDown      = 2'b00;      // P0
692     i_TxDetectRx    = 1'b1;      // Loopmode
693     i_TxElecIdle    = 1'b0;      // Elec Idle
694     i_TxCompliance  = {DATA_BYTES{1'b0}};
695     i_RxPolarity    = 1'b0;
696     i_TxData        = {DATA_WIDTH{1'b0}};
697     i_TxDataK       = {DATA_BYTES{1'b0}};
698
699     #128;
700
701     // Transit to Powerstate P1
702
703     @(posedge o_PCLK);
704
705     i_PowerDown      = 2'b10;      // P1
706     i_TxDetectRx    = 1'b0;
707     i_TxElecIdle    = 1'b1;      // Elec Idle
708     i_TxCompliance  = {DATA_BYTES{1'b0}};
709     i_RxPolarity    = 1'b0;
710     i_TxData        = {DATA_WIDTH{1'b0}};
711     i_TxDataK       = {DATA_BYTES{1'b0}};
712
713     #384;
714
715     // End of Powerstate Transition Tests
716
717
718     // Transit to Powerstate P0 (Idle)
719
720     //@(posedge o_PCLK);
721
722     i_PowerDown      = 2'b00;      // P0
723     i_TxDetectRx    = 1'b0;
724     i_TxElecIdle    = 1'b1;      // Elec Idle

```



```

725     i_TxCompliance      = {DATA_BYTES{1'b0}};
726     i_RxPolarity        = 1'b0;
727     i_TxData            = {DATA_WIDTH{1'b0}};
728     i_TxDataK          = {DATA_BYTES{1'b0}};
729
730     #128;
731
732     // Transit to Powerstate P0 (Normal)
733
734     @(posedge o_PCLK); #1;
735
736     i_PowerDown         = 2'b00;      // P0
737     i_TxDetectRx        = 1'b0;
738     i_TxElecIdle        = 1'b0;      // Elec Idle
739     i_TxCompliance      = {DATA_BYTES{1'b0}};
740     i_RxPolarity        = 1'b0;
741     i_TxData            = {DATA_WIDTH{1'b0}};
742     i_TxDataK          = {DATA_BYTES{1'b0}};
743
744     #128;
745
746     // Send TS1 Ordered Sets for Word Alignment
747
748     send_TS1_OS(8);
749
750     // Normal Transmission Test
751
752     for (i = 0; i < 100 ; i = i + 1) begin
753         generate_data;
754         send_testdata(8);
755         #534;
756         check_testdata(8);
757     end
758
759     // Reset Test ( During Normal Operation )
760
761     @(posedge o_PCLK);
762     i_TxData            = 64'hB59C_A235_FBB5_F6D3;
763     i_TxDataK          = 8'b0100_1000;
764
765     #1000;
766
767     i_Reset             = 1'b0;      // Low Active
768     i_PowerDown         = 2'b10;
769     i_TxDetectRx        = 1'b0;
770     i_TxElecIdle        = 1'b1;
771     i_TxCompliance      = {DATA_BYTES{1'b0}};
772     i_RxPolarity        = 1'b0;
773     i_TxData            = {DATA_WIDTH{1'b0}};
774     i_TxDataK          = {DATA_BYTES{1'b0}};
775
776     #20;
777     i_Reset             = 1'b1;      // Release Reset
778
779     // Wait for end of Reset
780
781     while (o_PhyStatus == 1'b1) begin
782         @(posedge o_PCLK); #1;
783     end
784
785     #128;
786
787     // Transition to P0 (Normal Transmission)
788
789     @(posedge o_PCLK);
790
791     i_PowerDown         = 2'b00;
792     i_TxDetectRx        = 1'b0;
793     i_TxElecIdle        = 1'b0;
794     i_TxCompliance      = {DATA_BYTES{1'b0}};
795     i_RxPolarity        = 1'b0;
796     i_TxData            = {DATA_WIDTH{1'b0}};
797     i_TxDataK          = {DATA_BYTES{1'b0}};
798
799     // Reestablish Word Alignment
800
801     #155;

```

```

802
803     send_TSi_OS(8);
804
805     #50;
806
807     // Error Detection Test
808
809     @(posedge o_PCLK);
810
811     i_TxData          = 64'hB5B5_B5B5_B5B5_B5B5;
812     i_TxDataK         = 8'b0000_0000;
813
814     #212;
815
816     fork
817         check_err_detect (50);
818         force_transm_err (400);
819     join
820
821     #200;
822
823     // Negative Disparity Test
824
825
826     @(posedge o_PCLK);
827
828     i_TxData          = 64'hBCBC_BCBC_BCBC_BCBC;
829     i_TxDataK         = 8'b1111_1111;
830     i_TxCompliance   = 8'b0101_0101;
831
832     #500;
833
834     @(posedge o_PCLK);
835     i_TxData          = 64'hBCBC_BCBC_BCBC_BCBC;
836     i_TxDataK         = 8'b0101_0101;
837     i_TxCompliance   = 8'b0101_0101;
838
839     #300;
840
841     // Loopmode Test
842
843
844     // Transition to P0 (Loopback)
845
846     @(posedge o_PCLK);
847
848     i_TxData          = 64'hB5B5_B5B5_B5B5_B5B5;
849     //i_TxData        = 64'hD1B4_C345_DE_E545;
850     i_TxDataK         = 8'b0000_0000;
851     i_TxCompliance   = {DATA_BYTES{1'b0}};
852
853     #1000;
854
855     // Activate Loopback Mode
856
857     @(posedge o_PCLK);
858     i_PowerDown       = 2'b00;
859     i_TxDetectRx      = 1'b1;    // Loopback Modus
860     i_TxElecIdle      = 1'b0;
861     i_TxData          = {DATA_WIDTH{1'b0}};
862     i_TxDataK         = {DATA_BYTES{1'b0}};
863     i_RxPolarity      = 1'b0;
864
865     #1500;
866
867     // Deactivate Loopback Mode
868
869     @(posedge o_PCLK);
870     i_PowerDown       = 2'b00;
871     i_TxDetectRx      = 1'b0;    // Loopback Modus
872     i_TxElecIdle      = 1'b0;
873     i_TxData          = {DATA_WIDTH{1'b0}};
874     i_TxDataK         = {DATA_BYTES{1'b0}};
875     i_RxPolarity      = 1'b0;
876
877     #500;
878

```

```
879 // Receiver Detection Test
880
881 @(posedge o_PCLK);
882
883 i_PowerDown      = 2'b10; // P1
884 i_TxDetectRx     = 1'b0;
885 i_TxElecIdle     = 1'b1; // Elec Idle
886 i_TxCompliance  = {DATA_BYTES{1'b0}};
887 i_RxPolarity     = 1'b0;
888 i_TxData         = {DATA_WIDTH{1'b0}};
889 i_TxDataK        = {DATA_BYTES{1'b0}};
890
891 #256;
892
893 // Start Detection
894
895 @(posedge o_PCLK);
896 init_status_flag_done = 1'b1;
897
898 i_PowerDown      = 2'b10; // P1
899 i_TxDetectRx     = 1'b1;
900 i_TxElecIdle     = 1'b1; // Elec Idle
901 i_TxCompliance  = {DATA_BYTES{1'b0}};
902 i_RxPolarity     = 1'b0;
903 i_TxData         = {DATA_WIDTH{1'b0}};
904 i_TxDataK        = {DATA_BYTES{1'b0}};
905
906 #30000;
907
908 finalize;
909
910 end // initial
```