Hochschule München
University of Applied Sciences

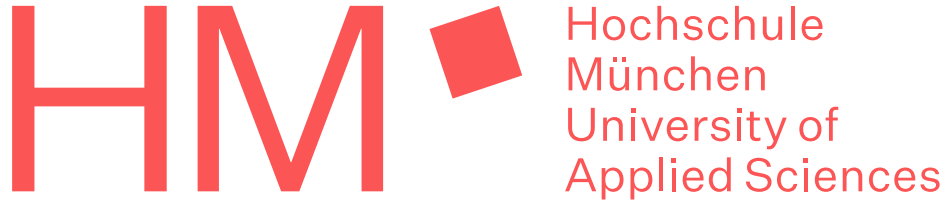
Fakultät für Elektrotechnik und Informationstechnik

Bachelorstudiengang Elektrotechnik und Informationstechnik (EIB)

Bachelorarbeit von Michael Feldmeier

Latenzarme digitale Stereokamera mit FPGA-basierter Videoverarbeitung

Bearbeitungsbeginn: 01.11.2022
Bearbeitungsende: 02.05.2023
Lfd.Nr: 2296



Fakultät für Elektrotechnik und Informationstechnik

Bachelorstudiengang Elektrotechnik und Informationstechnik (EIB)

Bachelorarbeit von Michael Feldmeier

Latenzarme digitale Stereokamera mit FPGA-basierter Videoverarbeitung

Low-Latency Stereo Camera with FPGA-based Video Processing

Betreuerin/Betreuer an der Hochschule: Prof. Dr. Christian Münker
Betreuerin/Betreuer in der Firma: Andreas Kahler, FabLab München e.V.

Bearbeitungsbeginn: 01.11.2022
Bearbeitungsende: 02.05.2023
Lfd.Nr: 2296

Erklärungen des Bearbeiters: Michael Heinrich Feldmeier

1. Ich erkläre hiermit, dass ich die vorliegende Bachelorarbeit selbständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe.
   Sämtliche benutzte Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate sind als solche gekennzeichnet.

Ort, Datum: München, 24.04.2023 Unterschrift: 

2. Ich erkläre mein Einverständnis, dass die von mir erstellte Bachelorarbeit in die Bibliothek der Hochschule München eingestellt wird. Ich wurde darauf hingewiesen, dass die Hochschule in keiner Weise für die missbräuchliche Verwendung von Inhalten durch Dritte infolge der Lektüre der Arbeit haftet. Insbesondere ist mir bewusst, dass ich für die Anmeldung von Patenten, Warenzeichen oder Geschmacksmustern selbst verantwortlich bin und daraus resultierende Ansprüche selbst verfolgen muss.

Ort, Datum: München, 24.04.2023 Unterschrift: 

Kurzfassung

Bachelorarbeit

Betreuer/in:	Studierende/r:	Studiengruppe:
Prof. Dr. Christian Münker	Michael Feldmeier	7DUW

Thema:
Latenzarme digitale Stereokamera mit FPGA-basierter Videoverarbeitung

Kurzfassung:
Diese Bachelorarbeit beschäftigt sich mit der Implementierung einer FPGA-basierten Stereokamera. Dazu soll es mit Hilfe einer Open Source Toolchain erreicht werden, zwei Kamerastreams zu einem 3D-HDMI Format unter Verwendung des ULX3S Developmentboards zu vereinen. Es müssen dabei zwei Kameras vom Typ Raspberry CAMv2 (Sony IMX219) verwendet werden, welche über das MIPI CSI 2 Protokoll angebunden sind. Die Überprüfung der Gesamtlatenz soll einen Wert von unter 40ms mit genanntem Setup ergeben. Somit war es Teil dieser Arbeit, sich zunächst mit den kompatiblen Open Source Toolchains auseinanderzusetzen und nachfolgend eine Konzept auszuarbeiten, welches gegebene Hardware erläutert und verschiedenste Anforderungen an Speicher und Timing bei unterschiedlichen Auflösungen und Frameraten aufzählt. Daraufhin wurden anhand der gegebenen Protokoll- und Hardwarespezifikationen die benötigten Kommunikationsstandards in der Hardwarebeschreibungssprache Verilog implementiert und Funktionen anhand von Simulationen und realen Testaufbauten validiert. Abschließend werden einzelne Ergebnisse genau analysiert, Schwachstellen identifiziert und auf benötigte Erweiterungen sowie mögliche Lösungen eingegangen.

Abstract

bachelor thesis

Supervisor:	Student:	Studiengruppe:
Prof. Dr. Christian Münker	Michael Feldmeier	7DUW

theme:
Low-latency stereo camera with FPGA-based video processing

abstract:
This bachelor thesis deals with the implementation of an FPGA-based stereo camera. The goal is to combine two camera streams to a 3D-HDMI format using an open source toolchain and the ULX3S development board. Two cameras of the type Raspberry CAMv2 (Sony IMX219) have to be used, which are connected via the MIPI CSI 2 protocol. The check of the total latency should result in a value of less than 40ms with the mentioned setup. Thus, it was part of this work to first deal with the compatible open source toolchains and then to work out a concept, which explains the given hardware and lists various requirements for memory and timing at different resolutions and frame rates. Then, based on the given protocol and hardware specifications, the required communication standards were implemented in the hardware description language Verilog and functions were validated using simulations and real test setups. Finally, individual results are analyzed in detail, weak points are identified, and required extensions and possible solutions are discussed.

Inhaltsverzeichnis

Abkürzungsverzeichnis ix

Abbildungsverzeichnis xi

Tabellenverzeichnis xii

Listingverzeichnis xiii

1 Einleitung 1
1.1 Hintergrund und Motivation 1
1.2 Überblick über FPGA-basierte Bildverarbeitung 2
1.3 Zielsetzung 2

2 FPGA Designflow 3
2.1 Überblick über die verwendete FPGA-Plattform 3
2.2 Überblick über die verwendete FPGA-Toolchain 4
2.2.1 Yosys Open Synthese Suite 4
2.2.2 NextPnR Place and Root Tool 5
2.2.3 Ujprog USB JTag Programmer 5
2.2.4 Verilator Simulator 5
2.2.5 Schwachstellen der Toolchain 7

3 Konzept 8
3.1 Speicherbedarf 8
3.2 Timing Anforderungen 9
3.3 IO Anforderungen 11

4 Implementierung 13
4.1 High Definition Multimedia Interface 13
4.1.1 Signaling und Encoding 15
4.1.2 HDMI Transceiver 18
4.2 MIPI CSI 2 Interface 21
4.2.1 I2C Master 22
4.2.2 Kamera Parametrierung 24
4.2.3 Videostream 28
4.2.4 MIPI Receiver IDDRX1F 32
4.2.5 ECC und CRC 38
4.2.6 MIPI Receiver IDDRX2F 42
4.3 Toplevel Komponente 46

5 Ergebnisse 48
5.1 HDMI Videostream 48

5.2 Analyse der Kameradaten 49
5.2.1 IDDR1 Implementierung 50
5.2.2 IDDR2 Implementierung 52
5.2.3 Frametiming 53
5.2.4 Latenzanalyse 54

6 Zusammenfassung 56
6.1 Fazit 56
6.2 Ausblick 57

Literaturverzeichnis 58

Abkürzungsverzeichnis

FPGA Field Programmable Gate Array

PnR Place and Root

HDMI High Definition Multimedia Interface

TMDS Transition-Minimized Differential Signalin

MIPI Mobile Industry Processor Interface

CSI Camera Serial Interface

PLL Phase-Locked Loop

SoT Start of Transmission

LP Low Power

HS High Speed

ECC Error Correction Code

CRC Cyclic Redundancy Check

SDR Single Date Rate

DDR Double Data Rate

RAM Random Access Memory

DRAM Dynamic Random Access Memory

SRAM Static Random Access Memory

SDRAM Syncron Dynamic Random Access Memory

USB Universal Serial Interface

LUT Look up Table

SD Standard Definition

HD High Definition

FHD Full Highg Definition

FPS Frames per Secound

ADC Analog Digital Converter

LVDS Low Voltage Differential Signaling

PH Packet Header

PF Packet Footer

Abbildungsverzeichnis

2.1 ULX3S Devboard 3
2.2 Synthese[1, S. 15] 4
2.3 GtkWave 6

3.1 Prinzipieller Aufbau 8
3.2 ULX3S Schaltplan Ausschnitt Pinheader [5] 11
3.3 Zuordnung Pinheader IO-Banks[5] 12
3.4 Anbindung des HDMI Interfaces[5] 12

4.1 Aufbau einer HDMI Übertragungsstrecke.[7, S. 24] 13
4.2 Übertragungsmodi des HDMI Protokolls.[7, S. 92] 14
4.3 Aufbau TMDS Encoder[7, S. 91] 15
4.4 Pixel Daten Kodierung Algorithmus[7, S. 118] 16
4.5 Komponentendiagramm des TMDS_Encoders 17
4.6 ODDRX1F Primitive 18
4.7 Komponentendiagramm des HDMI Transceivers 19
4.8 Ausschnitt Raspberry Pi Cam v2 Schaltplan [8] 21
4.9 I2C Daten Format[9, S. 17] 22
4.10 IC2 Low Level Protokoll[9, S. 18] 22
4.11 Komponentendiagrammdes I2C Masters 23
4.12 Zustandsmaschine des des I2C Masters 23
4.13 Bayermatrix[9, S. 89] 26
4.14 Clock Diagramm[9, S. 81] 26
4.15 Spannungslevel MIPI CSI 2[10, S. 35] 28
4.16 Anschlussplatine für zusätzliche Anforderungen 29
4.17 Start of Transmission[10, S. 37] 29
4.18 Datenformat Short and Long Packet[9, S. 48] 30
4.19 Datenformat eines Frames[9, S. 47] 31
4.20 Komponentendiagramm des MIPI Empfängers 32
4.21 Zustandsmaschine der SoTFSM 33
4.22 Komponentendiagramm des MIPI Empfängers 35
4.23 Zustandsmaschine der Protocoll Komponente 37
4.24 ECC Error Correction[11, S. 56] 38
4.25 CRC Schieberegister[11, S. 57] 41
4.26 Anwendungsbeispiel IDDRX2F[12, S. 9] 42
4.27 Unterschiedliche Phasenlagen des halbierten DDR-Clocks 43
4.28 Komponentendiagramm des IDDR2 MIPI Empfängers 44
4.29 Komponentendiagramm der Toplevel Komponente 46

5.1 fehlerfreie Übertragung der HDMI Testpattern 48

5.2 Drei unterschiedliche Testpattern der Kamera 50
5.3 Debayering der Testframes mit Pattern GBRG 51
5.4 Debayering der Testpattern 52
5.5 Zeilenübertragung D0 Lane MIPI CSI2 53
5.6 Frameübertragung D0 Lane MIPI CSI2 53
5.7 Simulation einer Zeilenübertragung MIPI CSI2 54

Tabellenverzeichnis

3.1 Speicherbedarf für unterschiedliche Bild- und Farbformate eines Frames . 8
3.2 benötigte Datenraten bei unterschiedlichen Auflösungen und Frameraten . 9
3.3 benötigte Pixel- und Bitclocks bei gegebenen Auflösungen 10

4.1 benötigte Pixel-Clock bei gegebenen Videoformaten bei 60 Hz Framerate[7, S. 137–143] 14
4.2 Kodierung der Steuersignale[7, S. 116] 16
4.3 Access Command Sequence.[9, S. 41] 24
4.4 Hardware spezifisches Kamerasetup[9, S. 29] 24
4.5 anwendungsspezifisches Kamerasetup[9, S. 30–31] 25
4.6 Blockdiagramm des Clocktrees[9, S. 33] 27
4.7 Testpattern Setup 27
4.8 Zustand Kodierung[10, S. 35] und Zeitkonstanten[10, S. 54–55] 30
4.9 Kodierung des 6-Bit Data Types.[9, S. 49] 31
4.10 Regeln für ECC Generierung 39

5.1 HDMI Timinganalyse 48
5.2 Gesamtdesign Timinganalyse 49
5.3 Timinganalyse MIPI IDDR1 51
5.4 Timinganalyse MIPI IDDR2 52

Listingverzeichnis

2.1 Ausschnitt Testbenchumgebung 6

4.1 9-bit Transition Minimized Kodierung 17
4.2 10-bit DC-Balance Kodierung 18
4.3 HDMI Transciever Instanziierungsparameter 19
4.4 Implementierung HDMI State Machine 19
4.5 Implementierung der TMDS Schieberegister 20
4.6 Zeitkonstanten 33
4.7 IDDR1 Komponente 34
4.8 Byte Arranger 34
4.9 Byte Alligner 35
4.10 Implementierung Data Encoder 36
4.11 Implementierung ECC 40
4.12 Implementierung CRC 41
4.13 IDDR2 Impelementierung 44
4.14 Byte Arranger Implementierung 45
4.15 Byte Alligner Implementierung 45
4.16 Implementierung des Dual Port Rams 46
4.17 Terminierung der Single-Ended Pins 47

1 Einleitung

Stereomikroskope stellen einen wichtigen Bestandteil der modernen Wissenschaft und Technologie dar. Sie ermöglichen es, dreidimensionale Bilder von mikroskopischen Objekten zu erfassen, welche mit bloßem Auge nicht zu erkennen sind. In den letzten Jahren haben Fortschritte in der Halbleitertechnologie die Entwicklung von FPGA-basierten Stereomikroskopen ermöglicht, wodurch speziell geringere Latenzzeiten erreicht werden können. Diese Eigenschaft macht sie besonders geeignet für Anwendungen in der industriellen Qualitätskontrolle, der Materialwissenschaft und der biomedizinischen Forschung. Ziel dieser Arbeit soll es sein zu überprüfen, ob es mithilfe des ULX3S FPGA Developmentboards sowie einer Open Source Toolchain möglich ist, zwei Kamerastreams zu einem HDMI 3D Videostream zu vereinen. Dabei soll eine Latenz von 40 ms nicht überschritten werden.

1.1 Hintergrund und Motivation

Herkömmliche digitale Stereomikroskope werden mittels Prozessoren realisiert, wobei diese durch das sequentielle Abarbeiten der Teilprozesse schnell an ihre Grenzen bezüglich Latenz, Auflösung und Framerate stoßen. Somit können diese in anspruchsvolleren Anwendungen, welche beispielsweise Hand-Augen-Koordination benötigen oder für automatische Qualitätskontrollen hochauflösende Bildgebung verlangen, nicht genutzt werden.

FPGAs stellen hierbei eine vielversprechende Lösung dar, welche es ermöglichen, anwendungsspezifische Logik zu implementieren, mit welcher genannte Teilprozesse parallelisiert abgearbeitet werden können. Durch den Einsatz von FPGAs können also Stereomikroskope mit geringerer Latenz, höherer Auflösung und Framerate sowie minimiertem Stromverbrauch entwickelt werden, wodurch neue Anwendungsbereiche in Biologie, Medizin oder anderen Bereichen erschlossen werden können.

Des Weiteren wird durch die Nutzung der Open Source Toolchain das gesamte Hardwaredesign für alle Interessengruppen zugänglich gemacht, wodurch das Design anwendungsspezifisch angepasst werden kann und die Entwicklung von FPGA-basierten Stereomikroskopen von einer breiten Gemeinschaft vorangetrieben wird.

Insgesamt bietet die Kombination aus FPGA-basierter Bildverarbeitung und Open-Source-Toolchain eine leistungsstarke und flexible Plattform für die Stereomikroskopie, die eine schnelle und genaue Verarbeitung von Daten in Echtzeit ermöglicht und gleichzeitig kosteneffektiv ist.

1.2 Überblick über FPGA-basierte Bildverarbeitung

In der FPGA-basierten Bildverarbeitung können verschiedene Algorithmen, wie beispielsweise Filterung, Kanten-, Textur- und Objekterkennung umgesetzt werden. Die Implementierung solcher Algorithmen in Hardware ermöglicht eine schnellere und genauere Verarbeitung von Bildern und erleichtert die Integration in bestehende Systeme. Es gibt jedoch auch einige Herausforderungen bei der Projektierung von FPGAs. Die Hauptprobleme sind die komplexe Programmierung dieser sowie die begrenzte Ressourcenverfügbarkeit, insbesondere bei der Implementierung komplexer Algorithmen. Darüber hinaus erfordert die Integration von FPGAs in ein System in der Regel spezielle Kenntnisse und Erfahrungen, um optimale Ergebnisse zu erzielen.

Nichtsdestotrotz bietet die FPGA-basierte Bildverarbeitung eine leistungsfähige, flexible und energieeffiziente Plattform in verschiedenen Anwendungen.

1.3 Zielsetzung

Konkret wird es Ziel dieser Arbeit sein, zwei Kamerastreams durch das ULX3S FPGA Developmentboard zu vereinen und per HDMI 3D Videostream auszugeben. Dabei werden zwei Kameras vom Typ Raspberry CAM v2 IMX219 genutzt, wobei mithilfe einer Open Source Toolchain ein Hardwaredesign mit einer Latenzzeit von unter 40ms implementiert werden soll. Außerdem gilt es konkrete Anforderungen an Speicher und Timing für unterschiedlichste Auflösungen auszuarbeiten und auftretende Schwachstellen zu identifizieren.

Aufgrund von einiger hardwarebedingten Einschränkungen wurde in der folgenden Arbeit lediglich der Passthrough eines Videostreams erreicht.

2 FPGA Designflow

Im folgenden Kapitel soll ein Überblick über die verwendete FPGA Plattform, über benötigte Tools sowie über den Ablauf des Designflows gegeben werden. Dazu soll eine kurze Einführung der einzelnen Tools sowie eine Funktionsübersicht gegeben werden.

2.1 Überblick über die verwendete FPGA-Plattform

Im Rahmen dieser Arbeit wurde als zentrale Komponente der ECP5 LFE5U-85 FPGA der Firma Lattice Semiconductor verwendet, welcher auf dem Open Hardware Board ULX3S verbaut ist. Der ECP5 wurde dabei speziell auf hohe Leistung, Energieeffizienz sowie niedrige Latenzen optimiert. Er verfügt über eine Vielzahl von Fähigkeiten, welche ihn für eine große Anzahl von Anwendungen qualifiziert. Speziell in Bezug auf digitale Bildverarbeitung sind High Speed IO Interfaces, ausreichend Speicherkapazität und verschiedenste IO-Standards essenziell. Bei dem ULX3S Developmentboard handelt es sich um eine Open Hardware Entwicklung, welche über eine große Anzahl von Features verfügt. Dazu gehören beispielsweise HDMI-kompatibles GPDI Interface, eine große Anzahl von frei konfigurierbaren IOs, zusätzlichem 32MByte SDRAM sowie JTAG USB Programmer Interface.

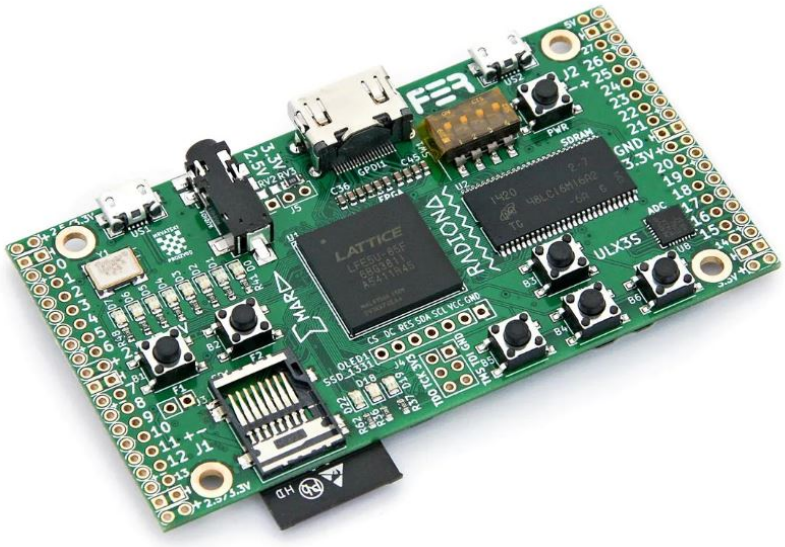


Abbildung 2.1: ULX3S Devboard

2.2 Überblick über die verwendete FPGA-Toolchain

Bei dem Entwurf von Digitalschaltungen sind mehrere Schritte notwendig, um ein effizientes Design zu gewährleisten. Dabei müssen wiederholt Simulationen und reale Tests der einzelnen Submodule ausgeführt werden, um sicherzustellen, dass einzelne Komponenten die gewünschten Funktionen erfüllen. Ein besonderes Augenmerk sollte darauf gerichtet werden, dass im Rahmen dieser Arbeit ausschließlich auf Open-Source-Tools zurückgegriffen wurde, welche auch zum Teil einige Schwachstellen im Vergleich zu kommerzieller Software aufweisen. Speziell werden für die Projektierung eines FPGAs Synthese-Tool, Place-and-Root-Tool, Programming-Tool sowie ein Simulator benötigt. Im Folgenden wird auf die konkreten Tools eingegangen.

2.2.1 Yosys Open Synthese Suite

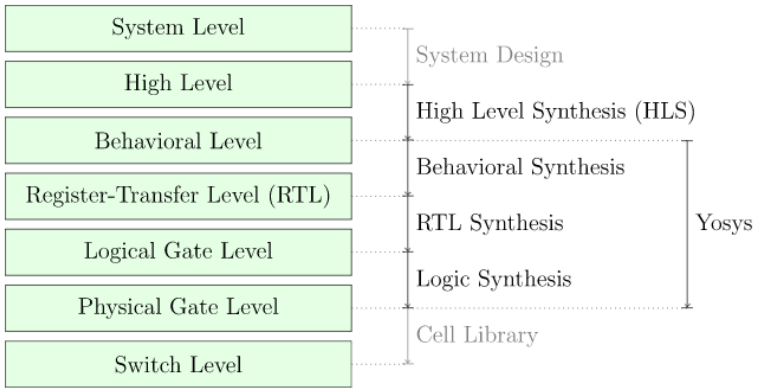


Abbildung 2.2: Synthese[1, S. 15]

Digitale Schaltungen können in verschiedenen Abstraktionsebenen dargestellt werden. Typischerweise werden diese dabei durch Hardwarebeschreibungssprachen wie Verilog oder VHDL in der High Level Abstraktionsebene designt. Das Synthese-Tool erlaubt es nun äquivalente Darstellungen auf niedrigeren Abstraktionsebenen zu finden. Abbildung 2.2 zeigt dabei die verschiedenen Ebenen. Letztendlich wird das High Level Design in die Switch Level Ebene überführt, bei welcher es sich um einen reinen Verdrahtungsplan in Textform handelt. Im Zuge dieser Arbeit wurde das Tool Yosys Open Synthese Suite verwendet. Folgende Schritte sind erforderlich, um mithilfe dieses Tools eine Netzliste aus eines Verilog Designs unter Linux zu erzeugen.

- yosys Öffnen der Yosys Konsole
- read_verilog design.v Einlesen der Verilog Dateien
- synth_ecp5 -json netlist.json

Zunächst muss die Yosys Konsole aufgerufen werden. Anschließend müssen alle Verilog Dateien, welche Teil des Designs sind, eingelesen werden, woraufhin die Synthese gestartet werden kann. Die Netzliste kann in unterschiedlichen Formaten ausgegeben werden, wobei im Rahmen dieses Projekts das Format .json genutzt wurde. Durch zusätzliche Befehlsdirektiven zu dem synth_ecp5 Command können noch weitere Optimierungsverfahren sowie Einschränkungen an den Optimierungsprozess vorgegeben werden. Beispielsweise kann durch den Zusatz -abc9 ein experimenteller Syntheseflow genutzt werden, welcher im Bezug auf LUT Mapping verbesserte Ergebnisse liefert.

2.2.2 NextPnR Place and Root Tool

Dem Place-and-Root-Tool wird zunächst die Netzliste im .json Format übergeben. Des Weiteren benötigt das PnR Tool ein Constraints File, welches die Zuordnung der physikalischen Pins zu den einzelnen Ports des Designs enthält. Außerdem werden im Constraint File die IO-Standards der Pins, zusätzliche Funktionalitäten wie PULL-UP, PULL-DOWN, OPEN-DRAIN sowie einzelne Timing Constraints definiert. Das PnR Tool kann aus diesen Informationen die Gesamtschaltung generieren und letztendlich ein Bitfile erstellen, welches auf den FPGA geflasht werden kann. Dieses wird wie folgt erzeugt.

- nextpnr-ecp5 –85k –json netlist.json –lpf constraints.lpf –textcfg ulx3s_out.conf
- ecppack ulx3s_out.conf bitfile.bit

Zunächst wird unter genauer Spezifikation der Baugröße des FPGAs eine Config Datei erstellt, wofür die Netzliste sowie das Constraint-File(.lpf) übergeben werden muss. Anschließend kann aus der Config Datei das Bitfile erstellt werden.

2.2.3 Ujprog USB JTag Programmer

Um das Bitfile unter Verwendung des Tools Ujprog auf den FPGA zu flashen, muss lediglich der Befehl ujprog bitfile.bit ausgeführt werden. Der Bitstream wird dabei über ein Serielle USB JTag Interface über die USB-Buchse US1 auf das Board übertragen.

2.2.4 Verilator Simulator

Neben der realen Funktionsprüfung des Designs stellen Simulationen einen wichtigen Punkt jedes Designflows dar. Der Simulator erlaubt es hierbei einzelne Komponenten des Designs mit frei konfigurierbaren Anregungen zu testen. Dabei können jegliche Komponenten, RAMs, Register oder Signale überprüft werden, wodurch Debugging erheblich erleichtert wird. In Zuge dieser Arbeit wurde das Simulationstool Verilator genutzt. Dieser erlaubt es Verilog Komponenten mit Hife einer C++ Testbenchumgebung zu simulieren. Es können die erzeugten Simulationsergebnisse mit WaveViewer Software wie beispielsweise GTKWave dargestellt werden. Für die Simulationen wird ein Testbench.cpp File

sowie ein testb.h File benötigt, welche auf die konkreten Designs angepasst werden müssen.

```
#include <verilatedos.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include "verilated.h"
#include "VCam_Init.h"
#include "testb.h"

int     main(int argc, char **argv) {
        Verilated::commandArgs(argc, argv);
        TESTB<VCam_Init>        *tb
        = new TESTB<VCam_Init>;
        tb->opentrace("I2C_Init.vcd");
        //tb->m_core->btn= 0;


        for (int i=0; i < 20000; i++) {
                if( i>10)
                tb->m_core->init=1;
                tb->tick();

        }
        printf("\n\nSimulation complete\n");
}

```

Listing 2.1: Ausschnitt Testbenchumgebung

Listing 2.1 zeigt Ausschnitte aus der Testbench. Das Verilator Tool erzeugt aus jeder einzelnen Verilog Komponente eine Klasse, welche in der Simulationsumgebung importiert werden kann. Die Funktion 'tick', welche sich in der Header Datei befindet, erzeugt nun die benötigten Taktsignale. Die Simulationsdauer und einzelnen Testpatterns können in der 'main' Funktion generiert oder abgeändert werden. Das Simulationsergebnis wird als .vcd File abgespeichert und kann mit GtkWave gemäß Abbildung 2.3 untersucht werden.

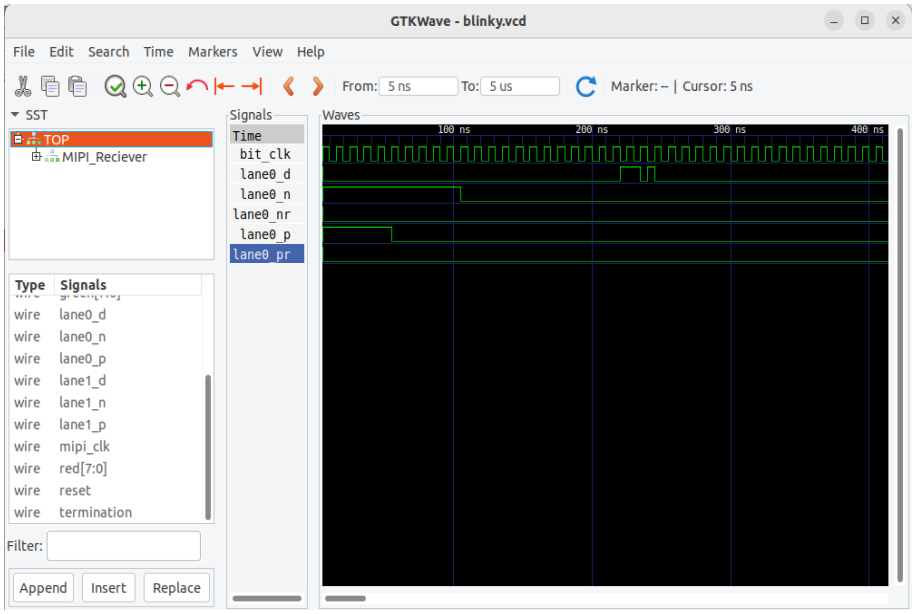


Abbildung 2.3: GtkWave

2.2.5 Schwachstellen der Toolchain

Im Folgenden werden einige Schwachstellen der Toolchain aufgelistet. Eine Schwachstelle des Yosys Synthese-Tool stellt die Ausgabe der Fehler beziehungsweise der Warnungen dar. So werden bei Zuweisung auf nicht vorhandene Register diese durch fehlerhafte Benennung implizit ohne Fehlermeldung deklariert, wodurch fehlerhafte Funktionalitäten entstehen. Dies gilt außerdem auch für Ein- und Ausgangsports, wodurch ganze Komponenten durch Optimierungsprozesse entfernt werden.

Des Weiteren ist das Erzeugen von Simulationen sehr zeitaufwendig, da Simulationsmodelle von essenziellen Primitives wie PLLs oder Clockdivider nicht vorhanden sind. Teilweise kann die Funktionalität genannter Primitives durch Verilog Simulationsmodule ersetzt werden, jedoch werden somit entscheidende Timingaspekte der Primitives nicht berücksichtigt. Besonders für Komponenten mit mehreren Clockeingängen müssen beide Clocks durch die ’tick’ Funktion erzeugt werden, wodurch die Simulationen erschwert werden. Zusätzlich werden einige Primitives und Funktionen durch die Open Source Toolchain nicht unterstützt und es kann beispielsweise nicht auf den intern vorhandenen MIPI IO Standard zurückgegriffen werden, durch welchen das zusätzliche Platinendesign deutlich vereinfacht wird.

3 Konzept

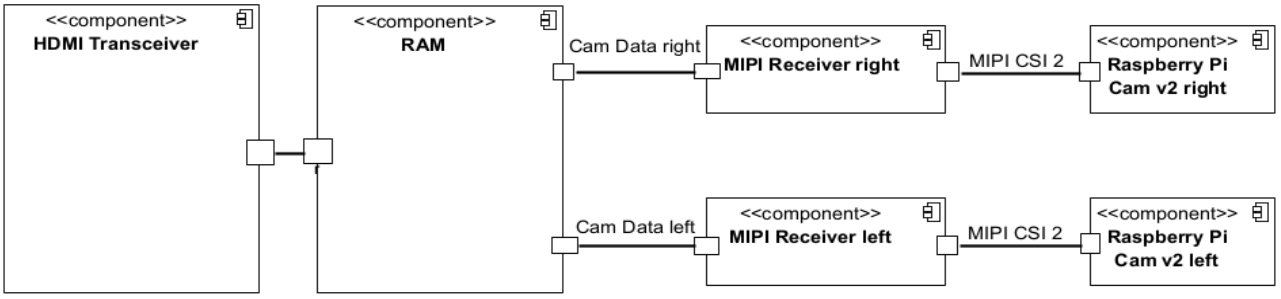


Abbildung 3.1: Prinzipieller Aufbau

Abbildung 3.1 zeigt den prinzipiellen Aufbau der Stereokamera. Dabei sind beide Raspberry Pi Kameras dargestellt, dessen Daten von den MIPI Receivern empfangen werden und anschließend an den RAM weitergegeben werden. Nachfolgend werden die Daten aus dem RAM an den HDMI Transceiver weitergegeben. Abhängig von der Framegröße und des Farbformates werden unterschiedliche RAM Speichergrößen benötigt.

3.1 Speicherbedarf

HDMI Format	Farbformat	Auflösung	Speicherkapazität
SD	RAW8	640x480	2.457,6kBit
SD	RGB8:8:8	640x480	7.372,8kBit
HD	RAW8	1280x720	7.372,8kBit
HD	RGB8:8:8	1280x720	22.118,4kBit
FHD	RAW8	1920x1080	16.588,8kBit
FHD	RGB8:8:8	1920x1080	49.766,4kBit

Tabelle 3.1: Speicherbedarf für unterschiedliche Bild- und Farbformate eines Frames

Tabelle 3.1 zeigt die benötigten Speicherkapazitäten für die Speicherung eines Frames. Es muss beachtet werden, dass jeweils zwei Frames als Stereobild abgespeichert werden müssen, weswegen die benötigte Speicherkapazität noch verdoppelt wird. Des Weiteren ist zu erkennen, dass durch die Abspeicherung der Daten im RAW8 Format, in welchem jeder Pixel nur ein Byte an Farbinformationen benötigt, die Speicherkapazität verringert werden kann. Vergleicht man nun die Werte in Tabelle 3.1 mit der internen Speicherkapazität des ECP5 LFE5U-85 FPGAs, stellt man fest, dass die Speicherkapazität der sysMEM Blocks lediglich ausreichend für die Abspeicherung eines Frames im SD RAW8 Format ist.[2, S. 11]

Die sysMEM Blocks können dabei als True-Dualport-RAM durch die Nutzung der DP16KD Primitives implementiert werden. Bei der Verilog Beschreibung eines Dualportram werden automatisch durch das Place-and-Root Tool die DP16KD sysMEM Blöcke eingebunden. Insgesamt stehen dabei 208 DP16KD Blockrams zur Verfügung, welche jeweils über 18432 Bits an Speicher besitzen, wodurch sich eine maximale Speicherkapazität von 3.833,856 kBit ergibt. Durch die zusätzliche Nutzung des Embedded Memory könnte ein weiteres Frame abgespeichert werden. Diese wurde jedoch im Zuge dieser Arbeit nicht umgesetzt.[3]

Um Frames mit höherer Auflösung abspeichern zu können, kann ebenso der externe SDRAM genutzt werden, welcher über 256MBit an Speicher verfügt und somit ausreichend Speicherkapazität für zwei FHD RGB8:8:8 Frames besitzt.

3.2 Timing Anforderungen

Für die einzelnen Teilkomponenten des prinzipiellen Aufbaus ergeben sich verschiedene Timing Anforderungen. Zunächst muss erwähnt werden, dass die Kamera in der theoretischen Betrachtung Daten mit einer Taktfrequenz von 216MHz bis zu 458MHz pro Lane versenden kann, was einer Datenrate von 432MBit/s bzw. 916MBit/s entspricht. Des Weiteren soll der MIPI Receiver die Kameradaten in einem 32-Bit Datenbus ausgeben, wodurch die Taktung des RAMs aufgrund der zwei Datalanes und der DDR-Clock auf ein Achtel der Taktfrequenz reduziert werden kann. Somit ergeben sich für die Taktung des RAMs mit 32-Bit Parallelinterface Frequenzen von 27MHz bis zu 57.25MHz.

Auflösung	Farbformat	Framerate	benötigte Datenrate
640x480	RAW8	60 FPS	147.456kBit/s
1280x720	RAW8	30 FPS	221.184kBit/s
1280x720	RAW8	60 FPS	442.368kBit/s
1920x1080	RAW8	30 FPS	497.664kBit/s
1920x1080	RAW8	60 FPS	995.328kBit/s
3280x2464	RAW8	21 FPS	1.357.762,56kBit/s
3280x2464	RAW8	30 FPS	1.939.660,8kBit/s
3280x2464	RAW8	60 FPS	3.879.321,6kBit/s

Tabelle 3.2: benötigte Datenraten bei unterschiedlichen Auflösungen und Frameraten

Tabelle 3.2 zeigt die theoretisch benötigten Datenraten bei verschiedenen Auflösungen und Frameraten, wobei die möglichen unteren Grenzen von Line- und Frameblankings vernachlässigt wurden. Auf die Auswirkung der Blankings auf Framerate und Auflösung im Zusammenhang mit der Datenrate wird im Kapitel „Ergebnisse“ eingegangen. Die maximale mögliche insgesamte Datenrate bedingt durch die Kamera beträgt also 1.832MBit/s, wodurch Auflösungen und Frameraten bis einschließlich 3280x2464 Pixeln bei 21 FPS

möglich sind. Durch die physikalische Anbindung der Kamera ist also maximal diese Einstellung möglich.

Des Weiteren wird die maximal mögliche Parametrierung der Kamera durch die IO Spezifikationen des FPGAs begrenzt. So beträgt die maximale mögliche Datenrate des ECP5 FPGAs bei der Nutzung der IDDRX1F Primitives 500MBit/s und bei Nutzung der IDDRX2F Primitives 800MBit/s, wodurch sich die maximal mögliche Datenrate bei zwei Lanes auf 1.600MBit/s verringert. Somit ist die oben genannte Maximaleinstellung weiterhin möglich.[2, S. 66]

Bei der Nutzung des Dualport-RAMs müssen die Timinganforderungen durch das Place and Root Tool ermittelt werden. Der externe SDRAM kann mit einer Taktfrequenz von 166MHz betrieben werden, wobei dieser über ein 16-Bit parallel Interface verfügt, wodurch sich eine maximale Datenrate von 2.656MBit/s ergibt. Da es sich bei dem externen SDRAM um einen Singleport-RAM handelt, können Schreib- und Leseoperationen nicht parallel durchgeführt werden. Im schreibenden Betrieb müsste also die oben genannte benötigte Taktfrequenz aufgrund des halbierten Datenwortes auf 54Mhz bis 114,5MHz erhöht werden, wobei Verzögerungszeiten durch Precharge- ‚Bankwechsel und Rowwechsel vernachlässigt wurden. Des Weiteren wird für das HDMI Interface aufgrund der drei einzelnen Channels eine Datenrate von dem dreifachen Bitclock gemäß Tabelle 3.3 benötigt. Somit können durch die gegebenen maximale Datenrate des SDRAMs maximal eine Auflösung von 1920x1080 Pixeln bei 30 FPS erreicht werden, wodurch beide Speicherzugriffe prinzipiell möglich sind. [4]

Des Weiteren muss noch auf die Timing Anforderungen des HDMI Interfaces eingegangen werden.

Auflösung	Framerate	Farbformat	Pixelclock	Bitclock
640x480	60 FPS	RGB8:8:8	25,175MHz	251,75MHz
800x600	60 FPS	RGB8:8:8	40MHz	400MHz
1280x720	60 FPS	RGB8:8:8	74,25MHz	742,5MHz
1920x1080	30 FPS	RGB8:8:8	74,25MHz	742,5MHz
1920x1080	60 FPS	RGB8:8:8	148.5MHz	1.485MHz
3840x2160	60 FPS	RGB8:8:8	594MHz	5.940MHz

Tabelle 3.3: benötigte Pixel- und Bitclocks bei gegebenen Auflösungen

Tabelle 3.3 zeigt die benötigten Pixel- bzw. Bitclocks für gegebene Auflösungen. Bei dem Vergleich mit den Spezifikationen des FPGAs ist bei der Nutzung der ODDRX1F Primitives eine maximale Taktung von 500MHz sowie bei Nutzung der ODDRX2F Primitives eine maximale Taktung von 800MHz möglich. Somit liegt die maximal mögliche Auflösung bei 1280x720 Pixel 60 FPS. Durch die Halbierung der Framerate wäre jedoch ebenfalls eine Auflösung von 1920x1080 Pixeln möglich. Zusätzlich muss erwähnt werden, dass durch die physikalische Lage des HDMI Connectors die Nutzung der ODDRX2F Primitives nicht möglich ist, wodurch die maximale theoretische Auflösung auf 800x600

Pixeln bei 60 FPS reduziert wird. Eine weitere Analyse des ULX3S Boards wird diese Aussage bestätigen.

3.3 IO Anforderungen

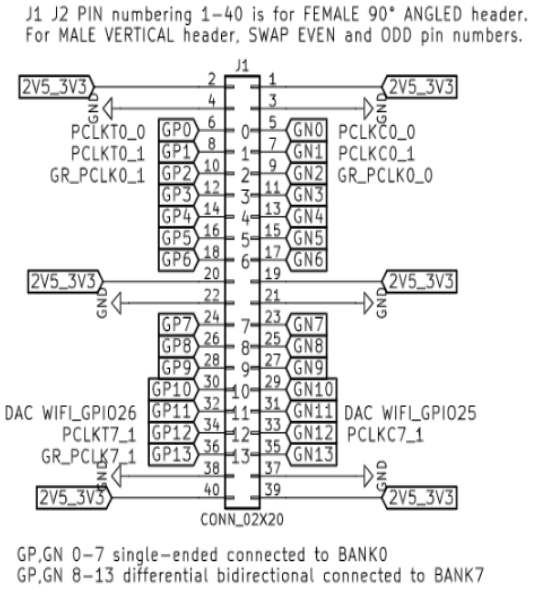


Abbildung 3.2: ULX3S Schaltplan Ausschnitt Pinheader [5]

Abbildung 3.2 zeigt die Belegung des ersten Pinheaders des ULX3S Boards. Der FPGA verfügt dabei über mehrere IO-Banks, für welche abhängig von der physikalischen Position unterschiedliche Funktionen zur Verfügung stehen. Dabei muss zunächst zwischen Top/Buttom und Left/Right Banks unterschieden werden. Bei „Bank0“ handelt es sich um eine Top/Bottom Bank, wodurch für A/B Pairs, welche sich an dieser Bank befinden nur Single-Ended In- und Output Buffer zur Verfügung stehen. Für Left/Right Banks wie „Bank7“ hingegen existieren A/B sowie C/D Pairs, welche über differentielle und Single-Ended In- und Output Buffer verfügen. [6, S. 10]

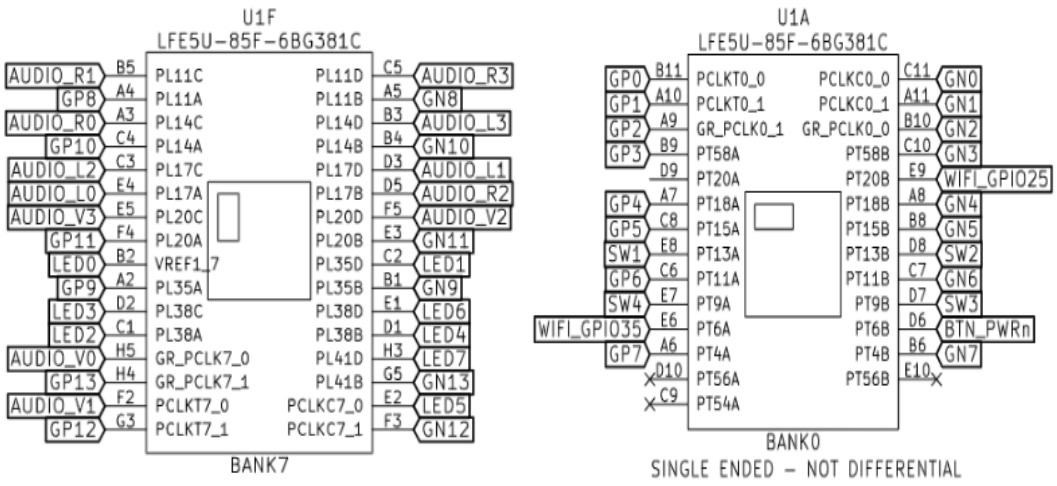


Abbildung 3.3: Zuordnung Pinheader IO-Banks[5]

Des Weiteren sind PCLK Pins zu erkennen, bei welchen es sich um Primimary-Clock-Capable IOs handelt, welche benötigt werden, um externe Clocks in das Clocknetzwerk einzubinden. Diese sind als differential Pairs sowie Single-Ended vorhanden. Es werden pro Kamera insgesamt acht Single-Ended IOs sowie weitere sechs diferentielle IOs benötigt, wobei für die MIPI Clocksignale die Primimary-Clock-Capable IOs genutzt werden müssen. Insgesamt sind an dem ersten Pinheader ausreichend Pins für beide Kameras vorhanden.

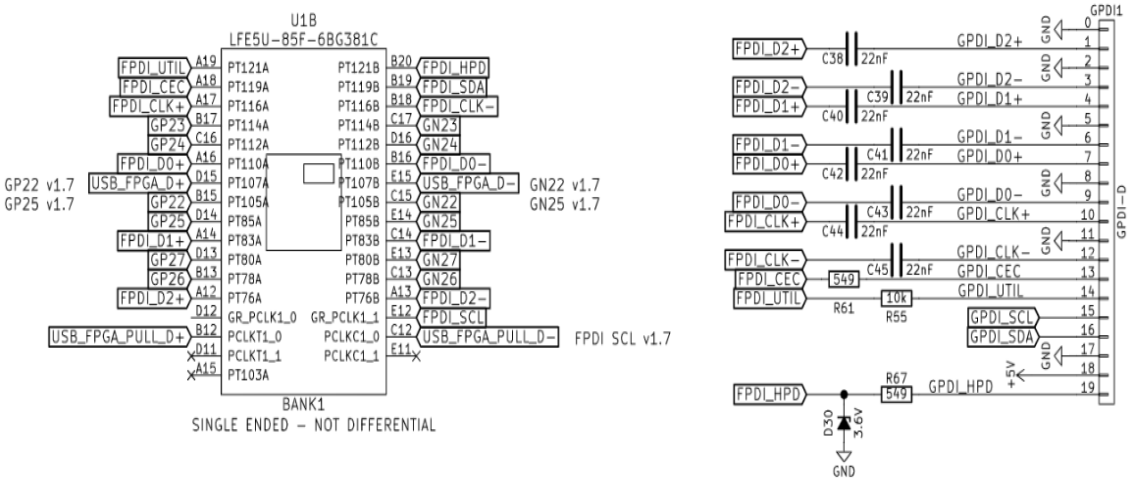


Abbildung 3.4: Anbindung des HDMI Interfaces[5]

In Abbildung 3.4 ist die Belegung des HDMI Interfaces zu erkennen. Dabei ist zu beachten, dass die HDMI Signale an eine Top/Bottom Bank angebunden wurden und somit nicht über True-Differential Outputs verfügen. Dennoch können durch Constraints die für das HDMI Interface benötigte IOs auf den IOStandard differential LVCMOS 3,3V festgesetzt werden. Durch die Nutzung einer HDMI PMOD Platine an dem übrigen Pinheader könnte ein HDMI Connector mit True-Differential Outputs umgesetzt werden, wodurch das ODDRX2F Primitive an diesem Ausgang genutzt werden könnte.

4 Implementierung

4.1 High Definition Multimedia Interface

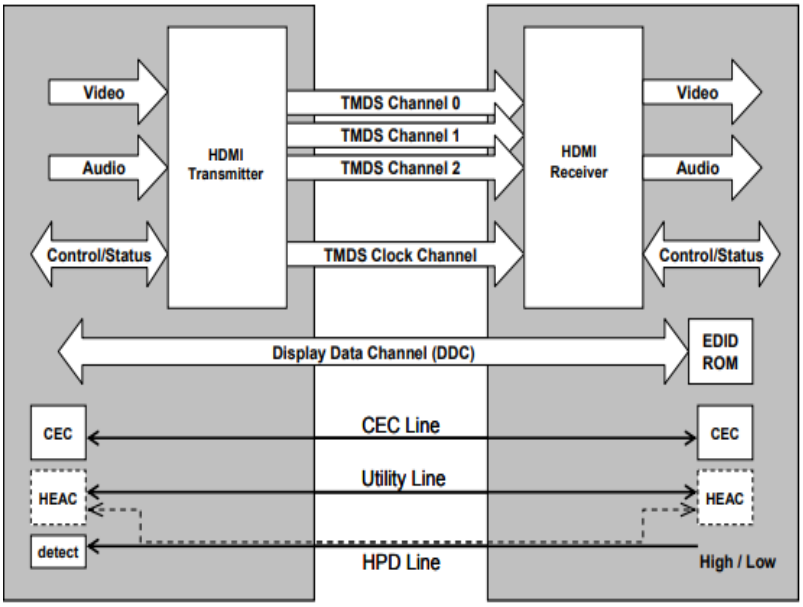


Abbildung 4.1: Aufbau einer HDMI Übertragungsstrecke.[7, S. 24]

Im folgenden Kapitel wird das High-Definition Multimedia Interface (HDMI) Version 1.4 genau erläutert sowie auf die verwendete Implementierung eingegangen. HDMI ermöglicht unter anderen die Übertragung von Videoformaten oder Audiodaten. Prinzipiell ist eine HDMI Übertragungsstrecke dabei aus drei TMDS Channels sowie einem TMDS Clock Channel aufgebaut, welche dabei als differentielle Pairs ausgeführt sind. Die noch weiteren für das HDMI Interface vorgesehenen Leitungen werden jedoch nicht für die Übermittlung der Videodaten benötigt.[7, S. 24–25]

Abbildung 4.1 zeigt dabei den prinzipiellen Aufbau einer HDMI Übertragung. Des Weiteren wird in Abbildung 4.2 die Übertragung eines Frames dargestellt. Zu Beginn eines Frames wird zunächst das Vertical-Blanking übertragen. Folgend darauf werden seriell die einzelnen Zeilen eines Frames beginnend mit einem Horizontal-Blanking übermittelt. Zu Erkennen ist, dass drei verschiedene Übertragungsmodi verwendet werden, um dem HDMI Device verschiedene Informationen zu übermitteln.[7, S. 92–95]

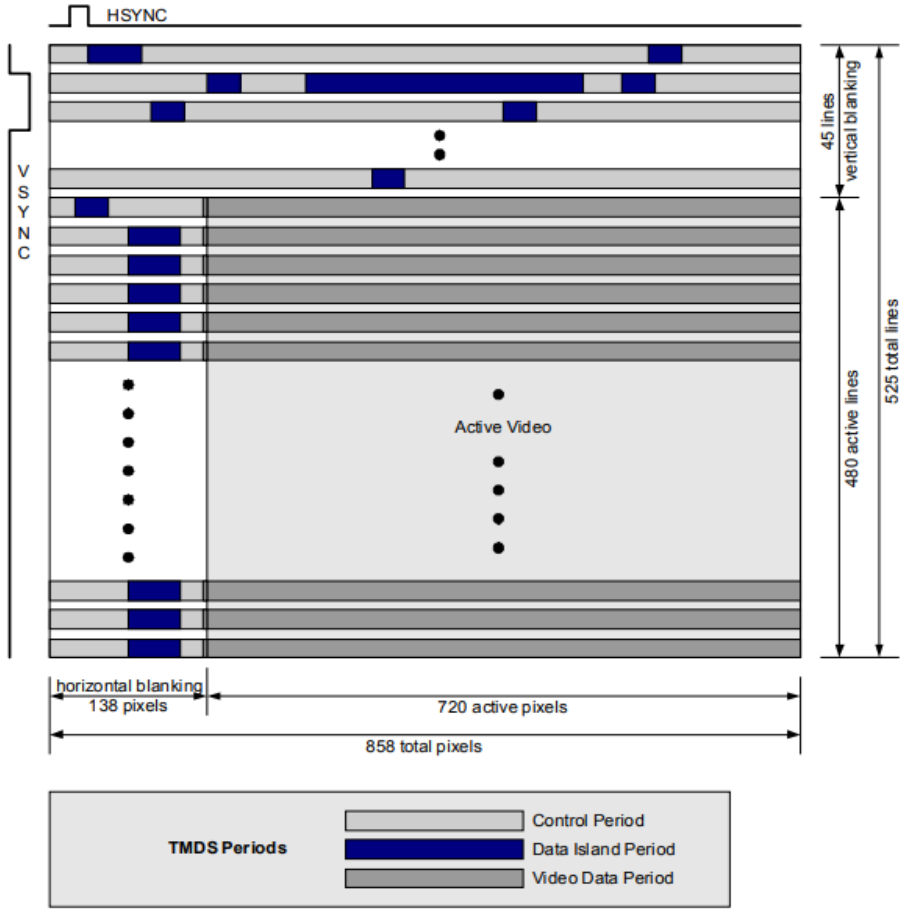


Abbildung 4.2: Übertragungsmodi des HDMI Protokolls.[7, S. 92]

Während der Control Periode werden die TMDS Channels entsprechend für Horizontal oder Vertikale Frame-Synchronisierung kodiert. So wird vor Übertragung der ersten Zeile ein VSYNC angezeigt sowie vor Anfang jeder Zeile ein HSYNC angezeigt. Nachfolgend auf die Blanking Perioden werden in der Video Data Periode die Pixeldaten übertragen. Die Data Island Periode dient unter anderem zur Übertragung von Audiodaten sowie Video Source Informationen. Auf diese soll aber im Weiteren nicht genauer eingegangen werden. Es stehen verschiedene Standards für die RGB Farbkodierung zur Verfügung. Im Zuge dieser Arbeit wurde RGB 8:8:8 verwendet, wobei insgesamt 24 Bit pro Pixel genutzt werden, darunter jeweils 8 Bit für Rot-, Grün- und Blauanteil.[7, S. 93–95]

Pixel-Clock	aktive horizontale Pixel	aktive vertikale Pixels
25MHz	640	480
40MHz	800	600
75MHz	1280	720
150MHz	1920	1080

Tabelle 4.1: benötigte Pixel-Clock bei gegebenen Videoformaten bei 60 Hz Framerate[7, S. 137–143]

Tabelle 4.1 zeigt die benötigten Pixel-Clock Frequenzen für die Übertragung bei jeweiligen Framegrößen mit einer Frequenz von 60 FPS. Es muss beachtet werden, dass während einer Periode der Pixel-Clock 10 Bit pro Channel übertragen werden. Dies führt zu hohen benötigten IO Taktraten, weswegen auf SERDES oder DDR Output Komponenten zurückgegriffen werden muss, um benötigte IOs mit ausreichend hohen Frequenzen betreiben zu können.[7, S. 137–143][2, S. 64–67]

4.1.1 Signaling und Encoding

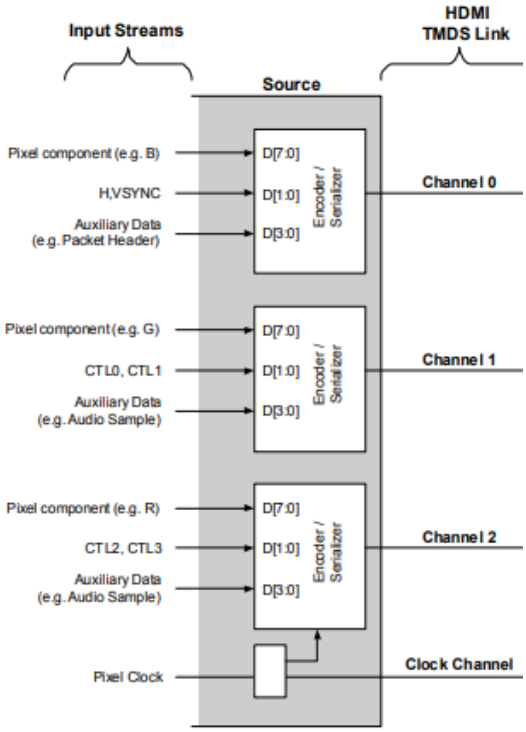


Abbildung 4.3: Aufbau TMDS Encoder[7, S. 91]

Wie bereits erwähnt, müssen pro Pixel-Clock Periode 10 Bit übertragen werden. Dazu müssen jedoch zunächst die benötigten Informationen auf 10 Bit kodiert werden. Es müssen also sowohl RGB 8:8:8 Pixeldaten als auch HSYNC und VSYNC kodiert werden. Abbildung 4.3 zeigt den konkreten Aufbau eines TMDS Channels, wobei jeder einzelne TMDS Channel ausgeschlossen des Clock-Channels dabei über zwei Steuersignale für das Signalisieren der aktuellen Periode (D[1:0]) sowie über ein 8 Bit Eingang zur Übergabe der konkreten Pixelwerte (D[7:0]) verfügt. Der erste Channel überträgt dabei HSYNC und VSYNC sowie die Blauanteile der Pixel. Die beiden weiteren Channels kodieren die Farbanteile Grün und Rot sowie noch zusätzlich benötigte Steuerinformationen, welche benötigt werden, um dem HDMI Device den nächsten Übertragungsmodus anzuzeigen. Jedoch können diese Steuersignale statisch auf den Wert CTL[3:0]=4'h1 belassen werden, da der Data Island Modus im Zuge dieser Arbeit nicht implementiert wurde.[7, S. 91–95]

D1	D0	10 Bit Code
0	0	10'b1101010100
0	1	10'b0010101011
1	0	10'b0101010100
1	1	10'b1010101011

Tabelle 4.2: Kodierung der Steuersignale[7, S. 116]

Tabelle 4.2 zeigt die Kodierung der Steuersignale. Abbildung 4.4 zeigt den verwendeten Algorithmus für die Kodierung der 8 Bit Pixeldaten zu 10 Bit Werten. Die Kodierung kann in zwei Stufen dargestellt werden. In der ersten Stufe wird aus dem 8 Bit Daten ein 9 Bit Transistion-Minimized Codewort erzeugt. Dieser enthält indessen die minimale Anzahl an 0→1 oder 1→0 Übergängen, wodurch ein verbessertes Störverhalten erreicht wird. Die zweite Stufe erzeugt aus den 9 Bit Codewörtern 10 Bit Code, welche bei der Übertragung durch differentielle Pairs zu einer Minimierung der Wechselanteile im Signal (DC-Ballance) führt.[7, S. 116–117]

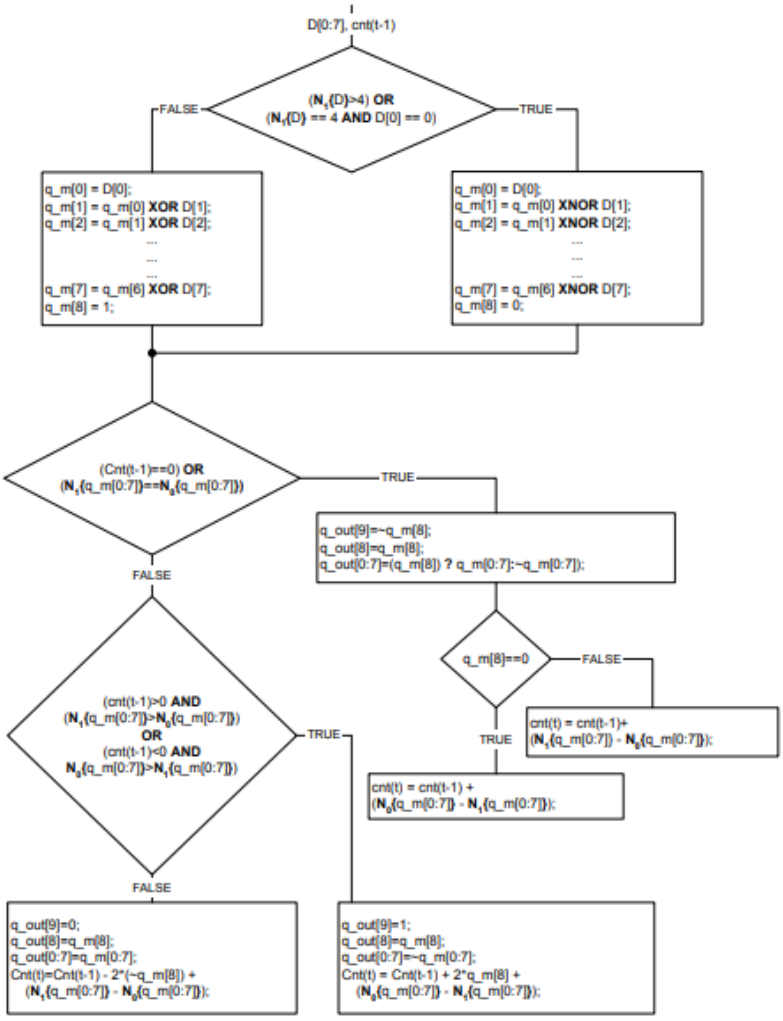


Abbildung 4.4: Pixel Daten Kodierung Algorithmus[7, S. 118]

Abbildung 4.5 zeigt das Komponentendiagramm des implementierten TMDS-Encoders. Der Encoder besitzt als Input die Pixel-Clock 'clklow' genannt sowie den high-aktiven Reset. Über den 'state' Input wird der Komponente der aktuelle Übertragungsmodi, also die Control- oder Video-Data-Periode übergeben. Über den Eingang 'H_VSync_Ctr' werden dem TMDS-Encoder die Steuersignale vorgegeben. Abhängig von dem vorgegebenen Übertragungsmodi werden entweder die Pixeldaten 'pix_data' oder die Steuersignale 'H_VSync_Ctr' kodiert und an den Ausgang 'q_out' ausgegeben.

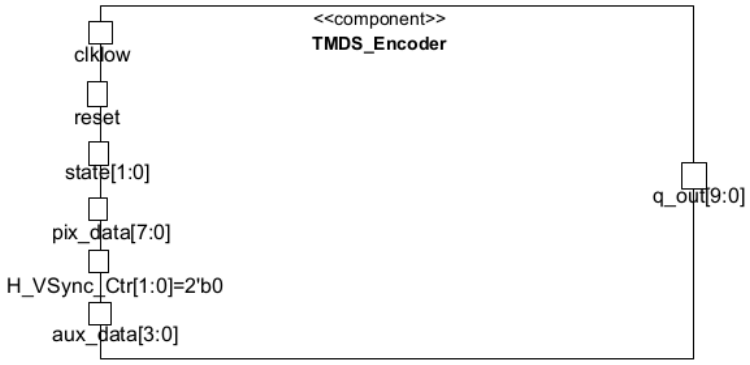


Abbildung 4.5: Komponentendiagramm des TMDS_Encoders

In Listing 4.1 ist die Implementierung der 9 Bit Transition Minimized Kodierung zu erkennen. Dabei berechnen die Funktionen N0() bzw. N1() die Anzahl der Nullen bzw. der Einser des übergebenen 8-Bit Wertes. Es werden dabei die in Abbildung 4.4 dargestellte Bedingung überprüft und abhängig von dieser durch XOR bzw. XNOR Verknüpfung das Zwischenergebnis q_m berechnet.

```
1   wire [8:0] q_m=(reset==1)?0:{q_m8,q_m7,q_m6,q_m5,q_m4,q_m3,q_m2,q_m1,q_m0};
2   wire q_m0=pix_data[0];
3   wire q_m1=(((N1(pix_data)>4)||(N1(pix_data)=='d4 && pix_data[0]=='b0))?
4                               q_m0~^pix_data[1]:q_m0^pix_data[1]);
5   wire q_m2=(((N1(pix_data)>4)||(N1(pix_data)=='d4 && pix_data[0]=='b0))?
6                               q_m1~^pix_data[2]:q_m1^pix_data[2]);
7   wire q_m3=(((N1(pix_data)>4)||(N1(pix_data)=='d4 && pix_data[0]=='b0))?
8                               q_m2~^pix_data[3]:q_m2^pix_data[3]);
9   wire q_m4=(((N1(pix_data)>4)||(N1(pix_data)=='d4 && pix_data[0]=='b0))?
10                              q_m3~^pix_data[4]:q_m3^pix_data[4]);
11  wire q_m5=(((N1(pix_data)>4)||(N1(pix_data)=='d4 && pix_data[0]=='b0))?
12                              q_m4~^pix_data[5]:q_m4^pix_data[5]);
13  wire q_m6=(((N1(pix_data)>4)||(N1(pix_data)=='d4 && pix_data[0]=='b0))?
14                              q_m5~^pix_data[6]:q_m5^pix_data[6]);
15  wire q_m7=(((N1(pix_data)>4)||(N1(pix_data)=='d4 && pix_data[0]=='b0))?
16                              q_m6~^pix_data[7]:q_m6^pix_data[7]);
17  wire q_m8=(((N1(pix_data)>4)||(N1(pix_data)=='d4 && pix_data[0]=='b0))? 'b0:'b1);
18
```

Listing 4.1: 9-bit Transition Minimized Kodierung

Für die Implementierung der DC-Balance Kodierung müssen nun gemäß Abbildung 4.3 vier Unterscheidungen getroffen werden. Das Wire 'q_out2pl' steht dabei für das zehnte Bit des Codes, 'q_out2p2' für das neunte und 'q_out2p3' für die Bits 1 bis 8. Des Weiteren besteht in diesen Teil des Algorithmus eine Abhängigkeit zu vorhergehenden Ergebnissen. So müssen ebenfalls die vier Zähler 'cnt0' bis 'cnt3' je nach aktuell zu kodierenden Pixel auf das Wire 'cnt' zugewiesen werden. Das Wire 'tmds_cnt' stellt dabei die Kodierung während der Controll-Period dar gemäß Tabelle 4.2. Unterdessen wird bei steigender Flanke der Pixel-Clock in Abhängigkeit des aktuell vorgegebenen 'state' Inputs entweder die Pixelkodierung oder Steuerkodierung an den Ausgang geschaltet.

```
1   wire q_out2p1=  ((cnt_old==0)||(N1(q_m[7:0])==N0(q_m[7:0]))) ? !q_m[8] :(((cnt_old>0 &&
2   ((N1(q_m[7:0])>N0(q_m[7:0])))||(cnt_old<0 && ((N0(q_m[7:0])>N1(q_m[7:0])))))?'b1:'b0));
3
4   wire q_out2p2=  ((cnt_old==0)||(N1(q_m[7:0])==N0(q_m[7:0]))) ? q_m[8] : (((cnt_old>0 &&
5   ((N1(q_m[7:0])>N0(q_m[7:0])))||(cnt_old<0 && ((N0(q_m[7:0])>N1(q_m[7:0])))))?q_m[8]:q_m[8]));
6
7   wire[7:0] q_out2p3=(reset==1)?0:((cnt_old==0)||(N1(q_m[7:0])==N0(q_m[7:0]))) ? ((q_m[8]==1)?
8                   q_m[7:0]:~q_m[7:0]):(((cnt_old>0 && ((N1(q_m[7:0])>N0(q_m[7:0])))||
9                           (cnt_old<0 && ((N0(q_m[7:0])>N1(q_m[7:0])))))?~q_m[7:0]:q_m[7:0]));
10
11  wire[9:0] q_out2=(reset==1)?0:{q_out2p1,q_out2p2,q_out2p3};
12
13  wire[31:0] cnt0=cnt_old+(N0(q_m[7:0])-N1(q_m[7:0]));
14  wire[31:0] cnt1=cnt_old+(N1(q_m[7:0])-N0(q_m[7:0]));
15  wire[31:0] cnt2=cnt_old+2*q_m[8]+N0(q_m[7:0])-N1(q_m[7:0]);
16  wire[31:0] cnt3=cnt_old-2*(!q_m[8])+N1(q_m[7:0])-N0(q_m[7:0]);
17
18  wire[31:0] cnt=(reset==1)?0:(((cnt_old==0)||(N1(q_m[7:0])==N0(q_m[7:0]))) ? ((q_m[8]==0)? cnt0 : cnt1 )
19   :(((cnt_old>0 && N1(q_m[7:0])>N0(q_m[7:0]))||(cnt_old<0 && (N0(q_m[7:0])>N1(q_m[7:0]))))? cnt2:cnt3));

20
21   wire[10:0] tmds_cnt=(H_VSync_Ctr[1]==1)?((H_VSync_Ctr[0]==1)?10'b1010101011 :10'b0101010100 )
22          : ((H_VSync_Ctr[0]==1)? 10'b0010101011 : 10'b1101010100);
23
```

Listing 4.2: 10-bit DC-Balance Kodierung

4.1.2 HDMI Transceiver

Wie bereits erläutert, können die benötigten Datenraten bei höheren Auflösungen für das HDMI Videostreaming durch die Nutzung der gewöhnlichen IO Pins nicht erreicht werden, weswegen auf Primitives zurückgegriffen werden muss, welche höhere Datenraten ermöglichen. Im Folgenden wird eine HDMI Implementierung vorgestellt, welche DDR Outputs mit Gearing 2:1 nutzt. Speziell handelt es sich dabei um ODDRX1F (Abbildung 4.6) DDR Output. Dieser ermöglicht gleichzeitig ein Schalten des Ausgangs bei steigender und fallenden Flanke, wodurch eine Verdoppelung der Datenrate erreicht wird. Das ODDRX1F Primitive erreicht dabei Datenraten von maximal 500 Mbit/s.[2, S. 67]

Abbildung 4.6: ODDRX1F Primitive

Es muss also verglichen mit Tabelle 4.1 das Signal ’clk_high’ nur noch mit halber Taktrate betrieben werden. Der HDMI Transceiver in Abbildung 4.7 besitzt dabei zwei Clock Inputs ’clk_low’ und ’clk_high’. Dabei handelt es sich einmal um den Pixel-Clock sowie um den halbierten Bit-Clock. Des Weiteren besitzt er noch Inputs für Rot-, Grün- und Blauanteile sowie einen Output ’addr’, welcher die Adresse des aktuellen Pixels ausgibt. Letztendlich wird das tatsächliche HDMI-Signal durch den Output ’TMDS’ dargestellt. Die HDMI State Machine ist dafür zuständig, im benötigten Format (siehe Abbildung 4.1) dem HDMI Device Controll- und Videodatenperioden zu signalisieren sowie HSync, VSync Signale und Blankings im korrekten Timing zu generieren. Dazu besitzt die Kom-

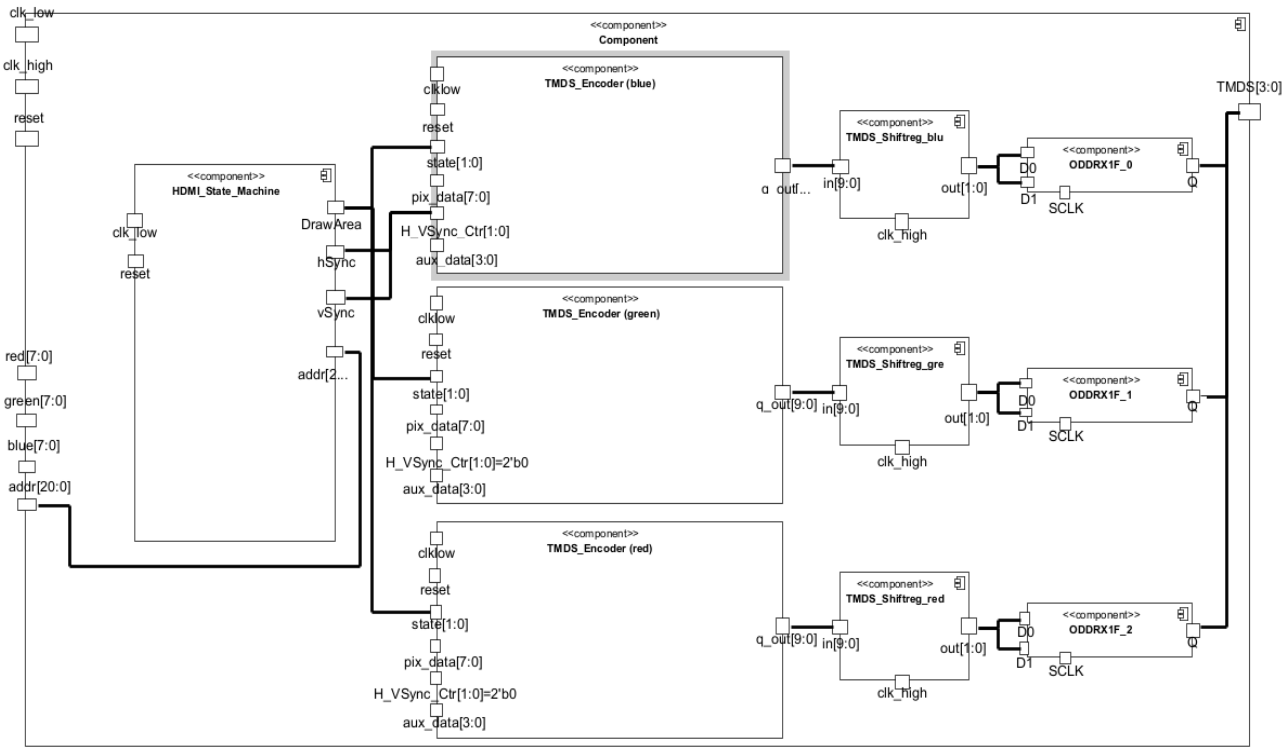


Abbildung 4.7: Komponentendiagramm des HDMI Transceivers

ponente Parameter, welche bei Instanziierung des Moduls siehe Listing 4.2 geeignet abgeändert werden können. Eine Blanking Periode besteht hierbei aus Frontporch-, Sync- und Backporchabschnitt. Während den Syncabschnitten muss das jeweilige HSync oder VSync Signal gesetzt werden. Initial wird der HDMI Transceiver für eine Auflösung von 640x480 Pixeln parametriert.

```
#(parameter
h_pixel=640,
h_front_porch=16,
h_back_porch=48,
h_tot_pixel=800,
v_pixel=480,
v_front_porch=10,
v_back_porch=33,
v_tot_pixel=525)

```

Listing 4.3: HDMI Transciever Instanziierungsparameter

Listing 4.3 zeigt die Implementierung der HDMI State Machine. Dabei wird mit zwei Zählern gearbeitet, welche bis jeweils ’h_tot_pixel’ beziehungsweise ’v_tot_pixel’ inkrementiert werden. Abhängig dieser beiden Zähler werden nun durch Vergleich mit den in Listing 4.4 dargestellten Konstanten die jeweiligen Signalisierungen realisiert.

```
always (posedge clk_low) DrawArea <=(reset==1)?s 0 :( (CounterX<h_pixel) && (CounterY<v_pixel));
always(posedge clk_low)begin
        if(reset==1)begin
                CounterX<=0;
                addr=0;
        end else begin
                if(CounterX==h_tot_pixel-1)begin
                        CounterX <=0;
                end else begin
                CounterX <=CounterX+1;
                addr=(DrawArea==1)?addr+1:addr;
        end
                vSync <= ((CounterY>=v_pixel+v_front_porch) && (CounterY<v_tot_pixel-v_back_porch)
                        &&(CounterX<h_tot_pixel-2))||((CounterY==v_pixel+v_front_porch-1)
```

```
15                                      &&(CounterX>=h_tot_pixel-2));
16                      if(CounterX==h_tot_pixel-1)begin
17                              CounterY <= (CounterY==v_tot_pixel-1) ? 0 : CounterY+1;
18                              addr = (CounterY==v_tot_pixel-1) ? 0 : addr;
19                      end
20              end
21      end
22      always(posedge clk_low) hSync <=(reset==1)?0:((CounterX>=h_pixel+h_front_porch)
23                      &&(CounterX<h_tot_pixel-h_back_porch));
24
```

Listing 4.4: Implementierung HDMI State Machine

Der von den TMDS Encodern übergebene 10-Bit Code muss im folgenden nun noch serialisiert werden. Diese Aufgabe übernehmen die TMDS Schieberegister in Listing 4.5 dargestellt. Bei den Wires 'TMDS_red', 'TMDS_blue' und 'TMDS_green' handelt es sich um den berechneten 10-Bit Code. Dieser wird durch einen Modulo 10 Zähler in das Register 'TMD_shift' des jeweiligen Farbanteils geschrieben und durch das realisierte Schieberegister serialisiert. Des Weiteren ist zu erwähnen, dass die für die jeweiligen Auflösungen benötigten Taktraten durch zum Beispiel eine PLL erzeugt werden müssen. Hier wurden beide Taktraten durch eine EHXPLLL erzeugt, welche in der TOP Komponente des Verilog Designs initialisiert wurden.

```
1       wire [9:0] TMDS_red, TMDS_green, TMDS_blue;
2       wire TMDS_r,TMDS_b,TMDS_g;
3       reg [3:0] TMDS_mod10;   // modulus 10 counter
4       reg [9:0] TMDS_shift_red, TMDS_shift_green, TMDS_shift_blue;
5       reg TMDS_shift_load;
6       always (posedge clk_high) TMDS_shift_load <= (reset==1)? 0 : (TMDS_mod10==4'd4);
7       always (posedge clk_high)begin
8               if(reset==1) begin
9                       TMDS_shift_red<=0;
10                      TMDS_shift_green<=0;
11                      TMDS_shift_blue<=0;
12                      TMDS_mod10<=0;
13              end
14              else begin
15                      TMDS_shift_red   <= TMDS_shift_load ? TMDS_red   : TMDS_shift_red  [9:2];
16                      TMDS_shift_green <= TMDS_shift_load ? TMDS_green : TMDS_shift_green[9:2];
17                      TMDS_shift_blue  <= TMDS_shift_load ? TMDS_blue  : TMDS_shift_blue [9:2];
18                      TMDS_mod10 <= (TMDS_mod10==4'd4) ? 4'd0 : TMDS_mod10+4'd1;
19              end
20      end
21
```

Listing 4.5: Implementierung der TMDS Schieberegister

4.2 MIPI CSI 2 Interface

Für die Weiterverarbeitungen der Kameradaten muss das MIPI CSI 2 Protokoll implementiert werden. Dazu wird folgend nun genauer auf dieses Protokoll eingegangen, wobei zunächst auf die Initialisierung sowie Parametrierung der verwendeten Kamera eingegangen werden muss.

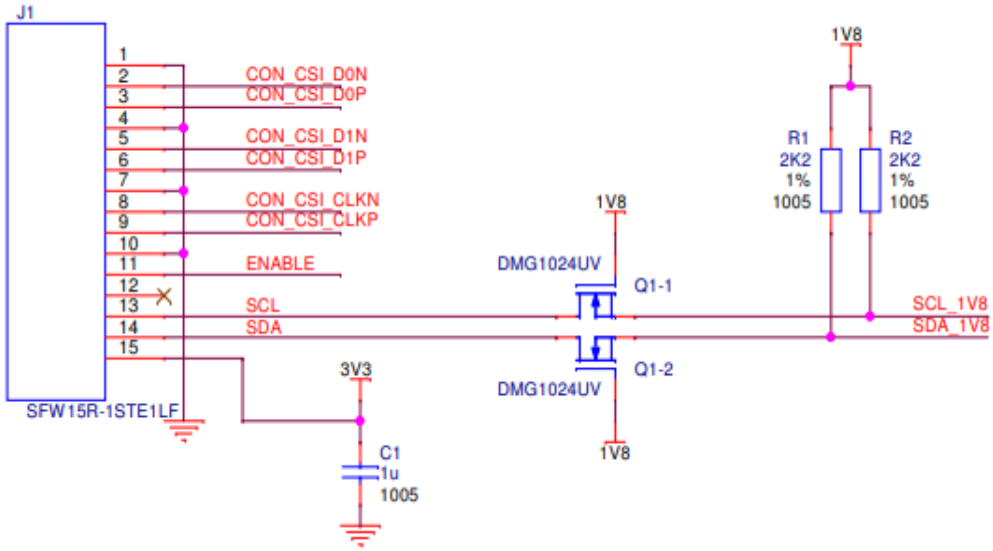


Abbildung 4.8: Ausschnitt Raspberry Pi Cam v2 Schaltplan [8]

Abbildung 4.8 zeigt einen Ausschnitt aus dem Schaltplan der Raspberry Pi v2 Kamera. Es werden dabei drei differentielle Pairs für Videodaten sowie zwei zusätzliche Steuerleitungen für Parametrierung und Initialisierung benötigt.[8]

4.2.1 I2C Master

Bei den zusätzlichen Steuerleitungen handelt es sich um ein I2C Interface, welches benötigt wird um die Kamera mit gewünschter Parametrierung zu initialisieren. Die Parameter betreffen unter anderen Datenformat, Bit-Clock Frequenz, Auflösung oder Testpattern. Die einzelnen Parameter werden dabei durch das Beschreiben von 16-Bit adressierten Registern über das genannte I2C Interface gesetzt. .[9, S. 17]

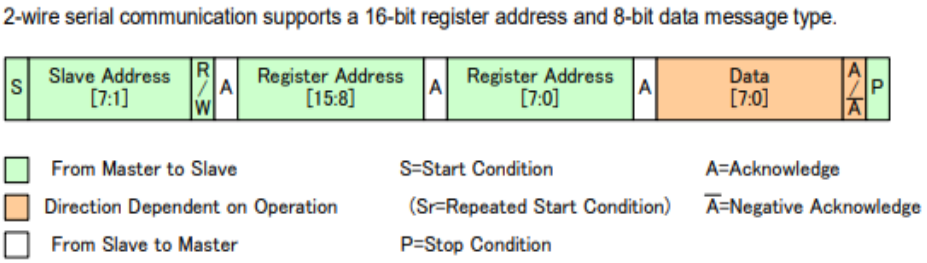


Abbildung 4.9: I2C Daten Format[9, S. 17]

Abbildung 4.9 zeigt das Datenformat des I2C Kommunikationsprotokolls. Zunächst wird folgend auf das Startbit die 7-Bit Slave-Adresse sowie das Read/Write-Bit gesendet. Die Slave-Adresse lautet bei der Kamera Sony IMX219 7'd16. Erkennt der Slave nun die eigene Adresse acknowledged dieser, indem er die SDA Leitung auf Ground zieht. Dies kann der Master erkennen und beginnt mit der Übertragung des Most Significant Byte der 16-Bit Registeradresse. Daraufhin folgt eine weitere acknowledge Periode, gefolgt auf das Least Significant Byte. Nach einer weiteren acknowlege Periode folgen abhängig von dem Read/Write-Bit die Daten zum Schreiben oder Lesen des Registers. Eine Transmission wird letztendlich durch das Stop-Bit gefolgt auf eine letzte acknowledge Periode beendet. [9, S. 17]

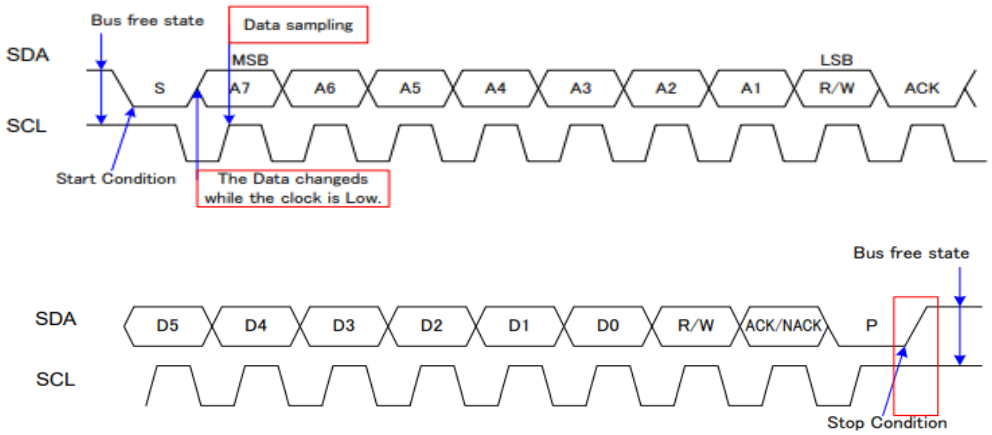


Abbildung 4.10: IC2 Low Level Protokoll[9, S. 18]

In Abbildung 4.10 wird das I2C Protokoll auf Low-Level Ebene dargestellt. Befindet sich der I2C Bus im Idle Modus, sind beide Leitungen im High Zustand. Der Start einer Trans-

mission wird über einen LOW Zustand der SDA Leitung signalisiert. Gefolgt darauf wird nun die SCL Leitung mit einer Frequenz von 400kHz getaktet, wobei nun bei steigender Flanke der SCL-Clock die Daten auf der SDA Leitung gemäß Abbildung 4.10 gesampled werden. Wurden alle benötigten Daten übertragen, folgt die Stop-Condition, welche ebenfalls aus einem LOW Zustand der SDA Leitung signalisiert wird. Nach dem Ende dieser gehen beide Leitungen wieder in den Idle Zustand über. [9, S. 18]

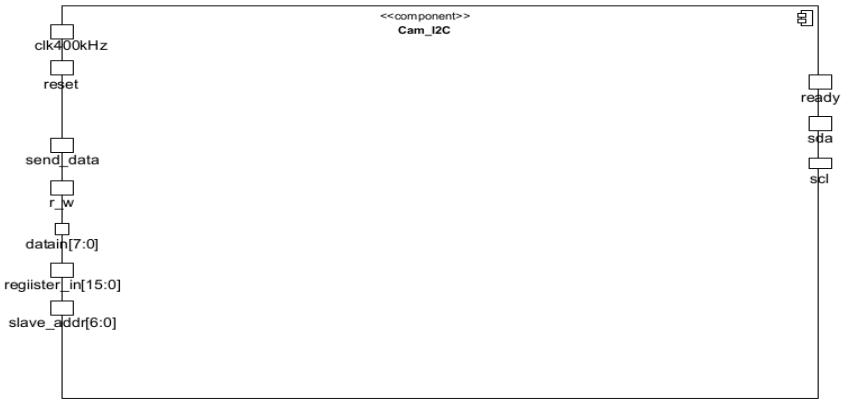


Abbildung 4.11: Komponentendiagrammdes I2C Masters

Abbildung 4.11 zeigt das implementierte I2C Master Modul. Über die Input Ports ’r_w’, ’datain’, ’register_in’ und ’slave_addr’ werden die zu sendenden Daten gemäß Abbildung 4.9 vorgegeben, welche bei einer steigenden Flanke am Eingang ’send_data’ versendet werden. Die 400kHz Clock muss durch beispielsweise eine PLL oder Clock-Divider erzeugt werden. Befindet sich das Modul in einer Transmission, wird der Output ’ready’ auf LOW gesetzt. Bei abgeschlossener Transmission geht dieser wieder in den Initialzustand HIGH über.

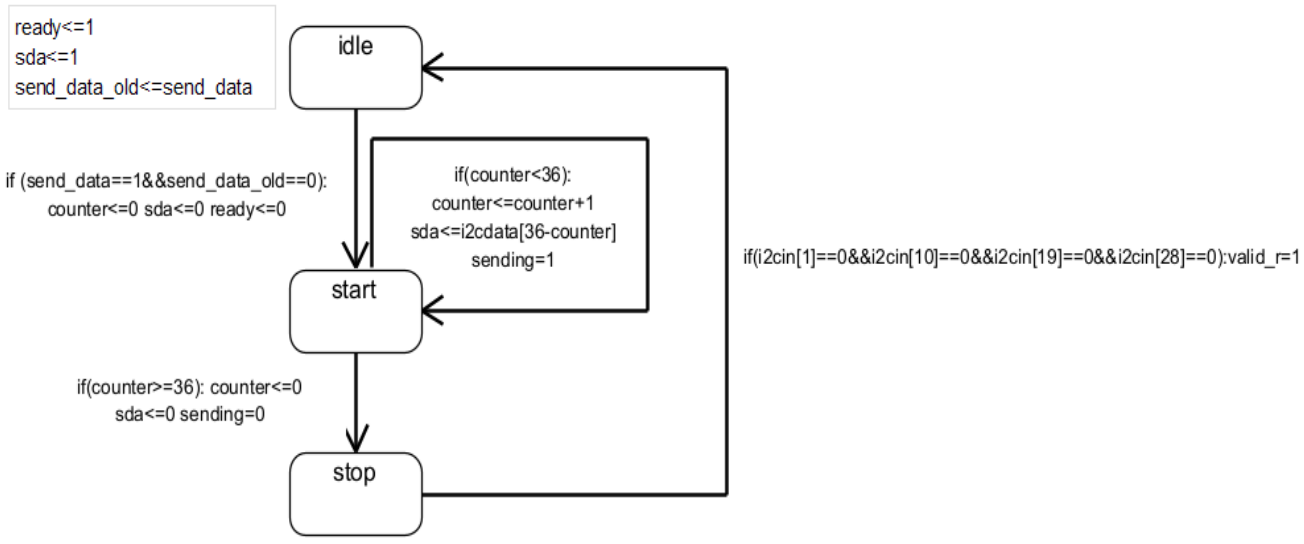


Abbildung 4.12: Zustandsmaschine des des I2C Masters

In Abbildung 4.12 ist die Zustandsmaschine des I2C Masters zu erkennen. Bei einer steigenden Flanke des ’send_data’ Signals wird dabei der Zähler zurückgesetzt, das ’ready’ Signal auf LOW gesetzt, sowie das Startbit gesendet. Das Register ’i2cdata’ beinhaltet die zu übertragenden Daten, welche LSB first an die Leitung SDA angelegt werden. Gleichzeitig wird der Zähler zyklisch inkrementiert. Wurde das letzte Bit übermittelt, kann in den Stop Zustand übergegangen werden, in welchem die optionale Acknowledge Prüfung durchgeführt wird. Des Weiteren wird die 400kHz Clock auf den Ausgang SCLK geschaltet, sobald das Bit ’sending’ auf HIGH gesetzt wurde.

4.2.2 Kamera Parametrierung

Für die Initialisierung der Kamera Sony IMX219 müssen nun mehrere einzelne Register beschrieben werden. Zunächst muss die Access Command Sequence abgearbeitet werden. Dazu müssen folgende Register in Tabelle 4.3 wie folgt beschrieben werden, um Zugriff auf wichtige Kontrollregister zu erhalten.[9, S. 41]

Adresse(Hex)	Daten
30EB	05
30EB	0C
300A	FF
300B	FF
30EB	05
30EB	09

Tabelle 4.3: Access Command Sequence.[9, S. 41]

Nachfolgend müssen nun Parameter abhängig des physikalischen Anschlusses der Kamera eingestellt werden. Dazu muss die Anzahl der verwendeten Lanes auf zwei gesetzt, das automatische Timing aktiviert und die Frequenz des externen Oszillators auf 24MHz eingestellt werden.[9, S. 29]

Adresse(Hex)	Daten	Bemerkung
0114	01	Konfiguration auf zwei Data Lanes
0128	00	MIPI Global Timing Auto
012a	18	MSB External Clock Frequency =24Mhz
012b	0	LSB External Clock Frequency =24Mhz

Tabelle 4.4: Hardware spezifisches Kamerasetup[9, S. 29]

Nun können die Einstellungen bezogen auf Auflösungen, Datenformate und Datenraten vorgenommen werden. Im Folgenden soll die Kamera auf eine Auflösung von 640x480 Pixel parametriert werden. Dazu müssen zunächst vertikale und horizontale Start- und Stopadressen parametriert werden. [9, S. 30–31] Die Kamera besitzt dabei eine maximale

Auflösung von 3280x2464 Pixeln. Über die Start- und Stopadressen kann ein konkreter Ausschnitt des gesamten Bildes bei niedrigeren Auflösungen gewählt werden. Horizontal wurde dabei die Startadresse 1000 sowie die Stopadresse von 2280 gewählt. Die Differenz der beiden Adressen entspricht dabei dem doppelten der horizonalen Framegröße von 640 Pixeln. Dies liegt daran, dass ebenfalls ein x2 Binning Mode parametriert wurde. Des Weiteren verfügt die Kamera über zwei Datenformate, wobei die Kameradaten im RAW8 oder RAW10 Format ausgegeben werden können.[9, S. 52]

Adresse(Hex)	Daten	Bemerkung
0164	03	horizontale Startadresse MSB
0165	e8	horizontale Startadresse LSB
0166	08	horizontale Stopadresse MSB
0167	e7	horizontale Stopadresse LSB
0168	02	vertikale Startadresse MSB
0169	f0	vertikale Startadresse LSB
016a	06	vertikale Endadresse MSB
016b	af	vertikale Endadresse LSB
016c	02	horizontale Framegröße MSB
016d	80	horizontale Framegröße LSB
016e	01	vertikale Framegröße MSB
016f	e0	vertikale Framegröße LSB
0170	01	Inkrementierung für ungerade Pixel horizontal
0171	01	Inkrementierung für ungerade Pixel vertikal
0174	03	horizontales Binning
0175	03	vertikales Binning
018c	08	CSI Datenformat RAW8
018d	08	CSI Datenformat RAW8

Tabelle 4.5: anwendungsspezifisches Kamerasetup[9, S. 30–31]

Bei Rohdatenformaten wie RAW8 oder RAW10 wird pro Pixel nur ein einzelner Farbanteil von der Kamera bereitgestellt. Abbildung 4.13 zeigt dabei die genaue Anordnung der Farbanteile. So besitzt jede gerade Zeile abwechselnd Grün- und Blauanteile, wobei die ungeraden Zeilen über abwechselnd Rot- und Grünanteile verfügen. Auf das Debayering, also der Konvertierung der Rohdaten in z.B. RGB Daten wird im Zuge dieser Arbeit nicht eingegangen. [9, S. 89]

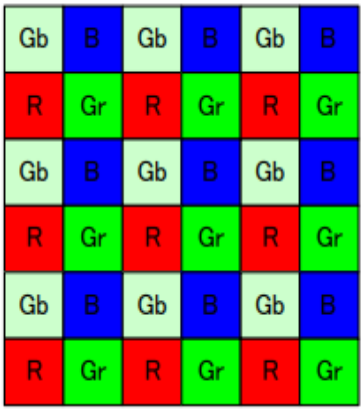


Abbildung 4.13: Bayermatrix[9, S. 89]

Um nun bei niedrigerer Auflösung ein großen Field of View zu erreichen, ermöglicht die Kamera x2 und x4 Binning Modes. Bei Binning handelt es sich um eine Mittelung über benachbarte Pixel, womit die Pixelanzahl halbiert oder geviertelt werden kann. [9, S. 53] Im Folgenden müssen noch Taktraten und interene PLLs der Kamera parametriert werden. Abbildung 4.14 zeigt dabei das Blockdiagramm der benötigten PLLs sowie Clockdivider.

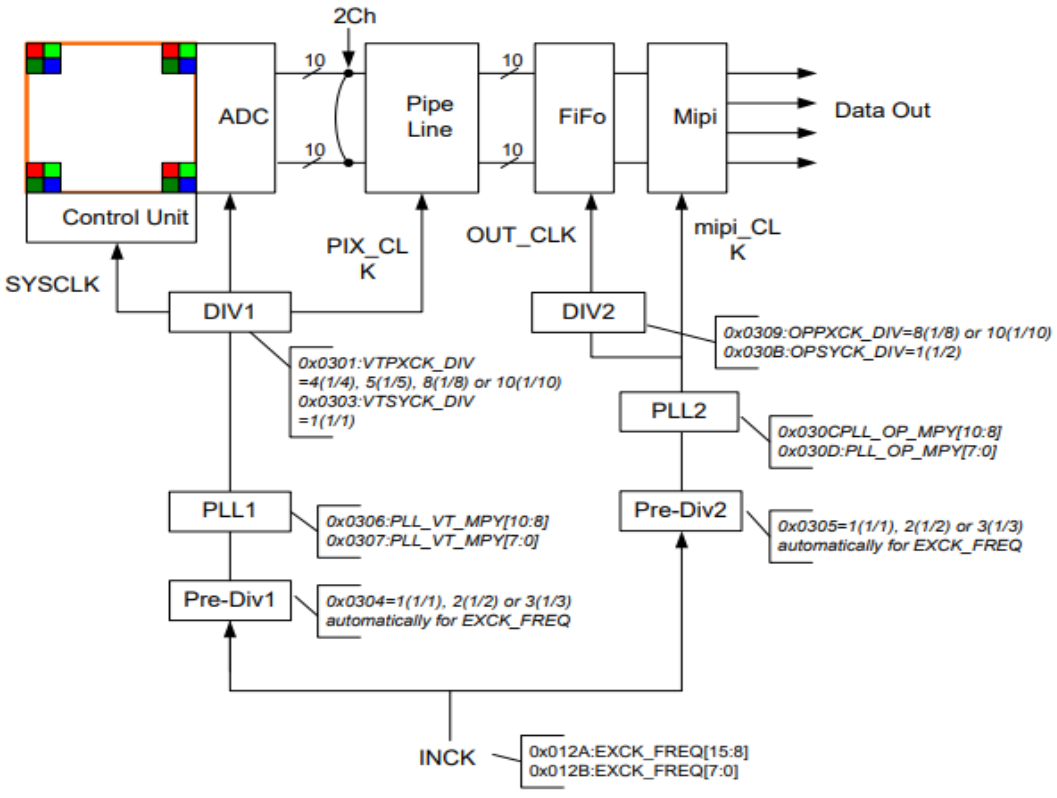


Abbildung 4.14: Clock Diagramm[9, S. 81]

INCK bezeichnet den externen 24MHz Oszillator, welcher auf dem Kameramodul verbaut ist. Der Clocktree spaltet sich einmal in einen Ast, welcher für die Taktung des Analog-

Digital-Konverters zuständig ist sowie in einen zweiten Ast, welcher den Datenoutput reguliert. So wird für den ADC die Taktrate erstmals durch Pre-Div1 geteilt und wiederum vervielfacht durch PLL1. Nun muss noch eine 1/4 Teilung für RAW8 bzw. eine 1/5 Teilung bei RAW10 durch DIV1 eingestellt werden. Für das Versenden der einzelnen Pixeldaten wird der INCK ebenfalls zunächst durch Pre-Div2 geteilt und wiederum über PLL2 vervielfacht. Die 'mipi_CLK' wurde dabei auf eine Frequenz von 912MHz parametriert. DIV2 ist wiederum abhängig von dem gewünschten Datenformates, wobei bei RAW8 eine Teilung von 8 parametriert wird. Die benötigten Register dazu sind in Tabelle 4.6 zu finden.[9, S. 81]

Adresse(Hex)	Daten	Bemerkung
0301	04	DIV1
0303	01	DIV1
0304	03	Pre-Div1
0305	03	Pre-Div2
0306	00	PLL1
0307	20	PLL1
0309	08	DIV2
030b	01	DIV2
030c	00	PLL2
030d	72	PLL2

Tabelle 4.6: Blockdiagramm des Clocktrees[9, S. 33]

Des Weiteren können noch Einstellungen zu Testpattern vorgenommen werden, welche benötigt werden, um das zu implementierende Kamerainterface auf Funktion zu testen. Dazu müssen horizontale und vertikale Framegröße festgelegt werden sowie eine Auswahl des Testpattern vorgenommen werden. Hier wurden als Framegröße 640x480 Pixel festgelegt sowie Testpattern 7 ausgewählt.[9, S. 63] Sind alle Einstellungen getroffen, kann nun das Streaming mit dem Kommando 8'h01 an das Register 16'h0100 gestartet werden.[9, S. 28]

Adresse(Hex)	Daten	Bemerkung
0624	02	horizontale Framegröße Testpattern MSB
0625	80	horizontale Framegröße Testpattern LSB
0626	01	vertikale Framegröße Testpattern MSB
0627	e0	vertikale Framegröße Testpattern LSB
0601	07	Auswahl des Testpatterns

Tabelle 4.7: Testpattern Setup

4.2.3 Videostream

Nun soll genauer auf das Datenformat des MIPI CSI 2 Protokolls eingegangen werden. Physikalisch werden für die Videodaten eine Clocklane sowie Datenlanes benötigt, welche abhängig von der Betriebsart differentiell oder single-ended, bei verschiedenen IO-Standards betrieben werden. So werden die einzelnen Leitungen im Low Power Modus single-ended und im High Speed Modus differentiell genutzt. Da im High-Speed Modus der IOStandard LVDS genutzt wird, wird außerdem eine dynamische Terminierung benötigt.[10, S. 22]

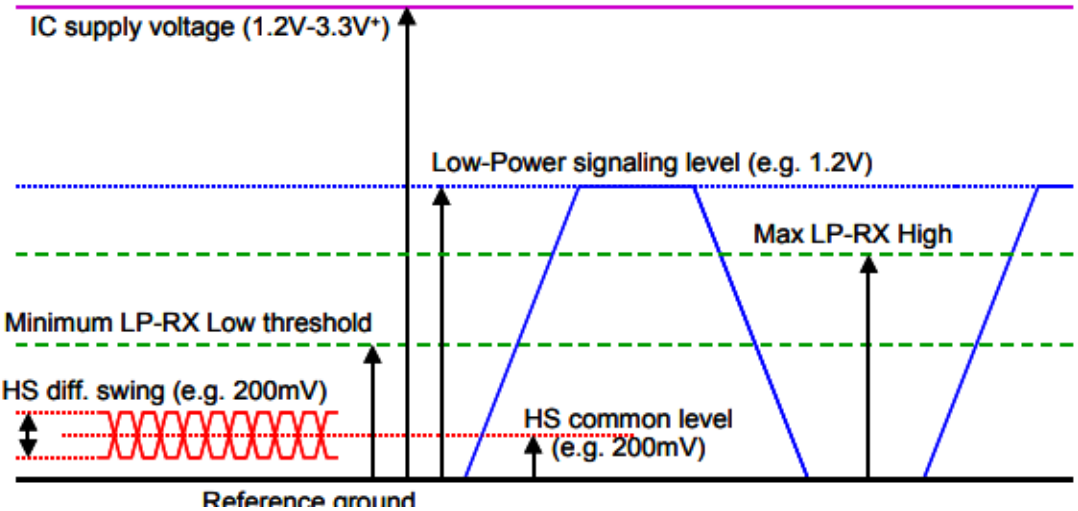


Abbildung 4.15: Spannungslevel MIPI CSI 2[10, S. 35]

Abbildung 4.15 zeigt dabei die unterschiedlichen genutzten Spannungslevels. So liegt der Low-Power Mode bei 1.2V und der High-Speed Modus bei einer Amplitude von 200mV. Bei dem Umschalten zwischen LP und HS Mode muss die dynamische Terminierung ausgeführt werden, um Reflektionen des hochfrequenten Signales zu minimieren.[10, S. 34–37] Da kein interner IOStandard, welcher diese Funktionen unterstützt, genutzt werden kann, müssen die zusätzlichen Anforderungen durch eine Platine umgesetzt werden.

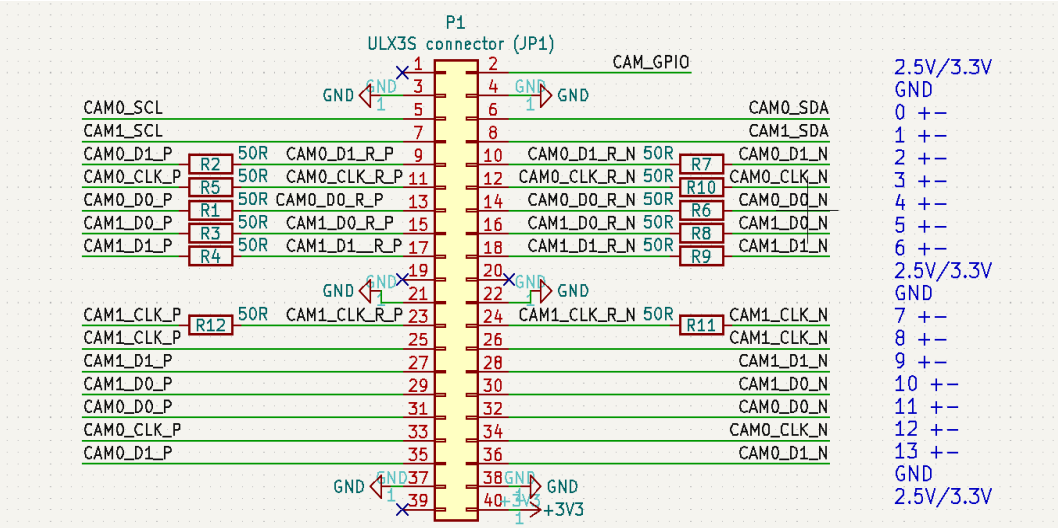


Abbildung 4.16: Anschlussplatine für zusätzliche Anforderungen

Der in Abbildung 4.16 dargestellte Schaltplan zeigt die zusätzliche Platine, welche benötigt wird, um die oben genannten Anforderungen zu erfüllen. Hierbei ist zu sehen, dass jede Datalane, bestehend aus zwei einzelnen Leitungen jeweils einmal an ein differentielles Pair sowie über einen 50 Ohm Widerstand auf zwei single-ended Pins verbunden wurde. Intern im FPGA können nun die IOStandards für die einzelnen Pins festgelegt werden. Durch das Groundschalten der single-ended Pins wird die dynamische Terminierung mittels 100 Ohm Widerstand realisiert. Da der physikalische Aufbau der Lanes geklärt wurde, kann auf die Start of Transmission eingegangen werden, welche den Start einer Datenübertragung signalisiert.

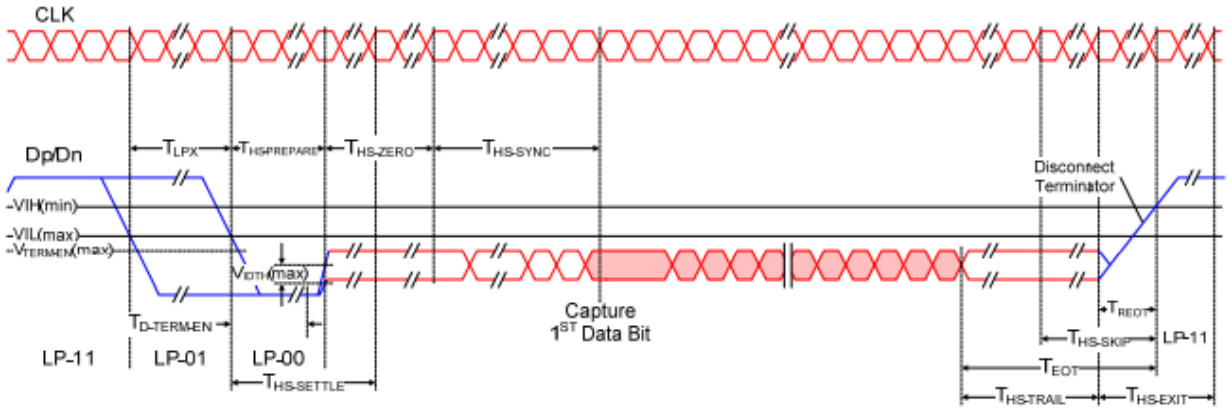


Abbildung 4.17: Start of Transmission[10, S. 37]

Abbildung 4.17 beschreibt den zeitlichen Ablauf der Start of Transmission. Zunächst befindet sich die Datalane im Zustand LP-11, wodurch der Idle Zustand des Low-Power Modus signalisiert wird. Darauf folgend wird der Zustand LP-01 für eine Zeit von T_{LPX} getrieben, woraufhin nach einer Zeit von $T_{D\text{-}TERM\text{-}EN}$ die Terminierung aktiviert werden muss. Anschließend wird von dem Transmitter der LP00 Zustand angezeigt,

nach welchem $T_{HS\text{-}SETTLE}$ abgewartet werden muss, bevor auf das Synchronisationsbyte 8'b10111000 gewartet wird. Die Übertragungsstrecke befindet sich nun im High-Speed Modus.[10, S. 36–37]

Zustand	Dp	Dn
LP-00	LP-Low	LP-Low
LP-01	LP-Low	LP-High
LP-10	LP-High	LP-Low
LP-11	LP-High	LP-High

Name	Min	Max
T_{LPX}	50ns	/
$T_{D\text{-}TERM\text{-}EN}$	/	35+4*UI
$T_{HS\text{-}SETTLE}$	85ns+6*UI	145ns+10*UI

Tabelle 4.8: Zustand Kodierung[10, S. 35] und Zeitkonstanten[10, S. 54–55]

In Tabelle 4.8 sind dabei die Kodierungen der Zustände sowie die Werte der Zeitkonstanten aufgelistet. Dp steht für den single-ended Wert der positiven Leitung des differential Pairs, wohingegen Dn für die negative Leitung steht. Die einzelnen Zeitkonstanten werden abhängig von der Datenrate des MIPI Interfaces angegeben, wobei UI für eine Bitzeit steht.[10, S. 54–55, 35]

Wurde das Synchronisationsbyte erkannt, folgen nun die Nutzdaten der Transmission, wobei zwischen Short Packet und Long Packet unterschieden wird.

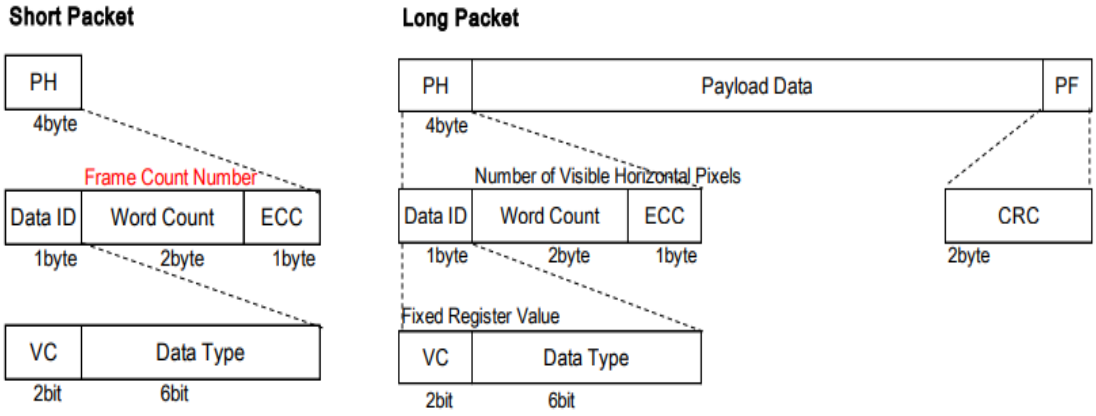


Abbildung 4.18: Datenformat Short and Long Packet[9, S. 48]

Dabei enthält ein Short Packet Informationen zu beispielsweise Framestart oder Frameende, wohingegen in einem Long Packet Embeddet Data oder Pixeldaten übertragen werden. Abbildung 4.18 zeigt den Aufbau der einzelnen Packete. Beide sind dabei aus dem gleichen Packetheader aufgebaut, welcher aus insgesamt aus 4 Bytes besteht. Darunter Data ID, Word Count und dem Error Correction Code(ECC). Das Data ID Byte ist aus 2 Bit Virtual Channel Identifier und einem 6-Bit Data Type zusammengesetzt, welcher die Art der folgenden Daten spezifiziert.[9, S. 48–49]

Code	Bemerkung
6'h00	Frame Start Code
6'h01	Frame Ende Code
6'h12	Embedded Data Code
6'h2A	RAW 8
6'h2B	RAW 10

Tabelle 4.9: Kodierung des 6-Bit Data Types.[9, S. 49]

Tabelle 4.9 listet dabei alle möglichen Data Types mit zugehörigem Code auf. Der Virtuell Channel Identifier wird in der folgenden Implemtierung nicht berücksichtigt, da im Zuge dieser Arbeit keine Virtuell Channels parametriert wurden. Bei Short Packets beinhaltet der Word Count den Frame Count, welcher einem Frame eine Nummer zuordnet. In einem Long Packet wird im WordCount hingegen die Anzahl an Pixeln übertragen, mit welcher dem Empfänger die Länge der Payload Data mitgeteilt wird. Mithilfe des 8-Bit ECC können 1-Bit Fehler inmitten des Packet Headers erkannt und korrigiert werden. Long Packets besitzen außerdem noch CRC- Prüfsummen, welche zur Validierung der empfangenen Daten genutzt werden können.

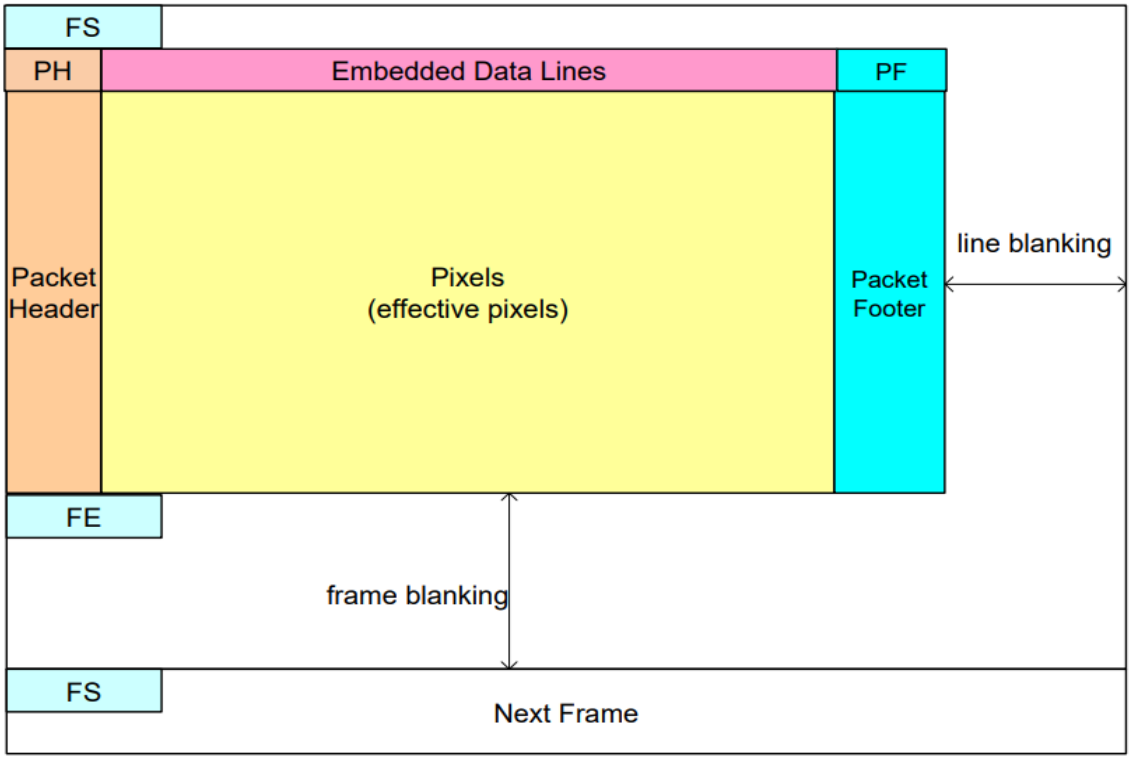


Abbildung 4.19: Datenformat eines Frames[9, S. 47]

Abbildung 4.19 zeigt den konkreten Aufbau der Übertragung eines Frames. Zunächst wird der Start des Frames mit einem Short Packet signalisiert, welcher gemäß Tabelle 4.9 den Frame-Start Code beinhaltet. Nachfolgend werden die Embedded Data Lines als Long Packet übermittelt, welche aber in der folgenden Implementierung nicht berücksichtigt wer-

den. Letztendlich werden nun die eigentlichen Pixeldaten zeilenweise übertragen. Diese beinhalten die Data Types RAW8 oder RAW10 abhängig von der Parametrierung. Wurden alle Zeilen übertragen, folgt ein weiteres Short Packet, welches das Ende des Frames signalisiert.[9, S. 47]

4.2.4 MIPI Receiver IDDRX1F

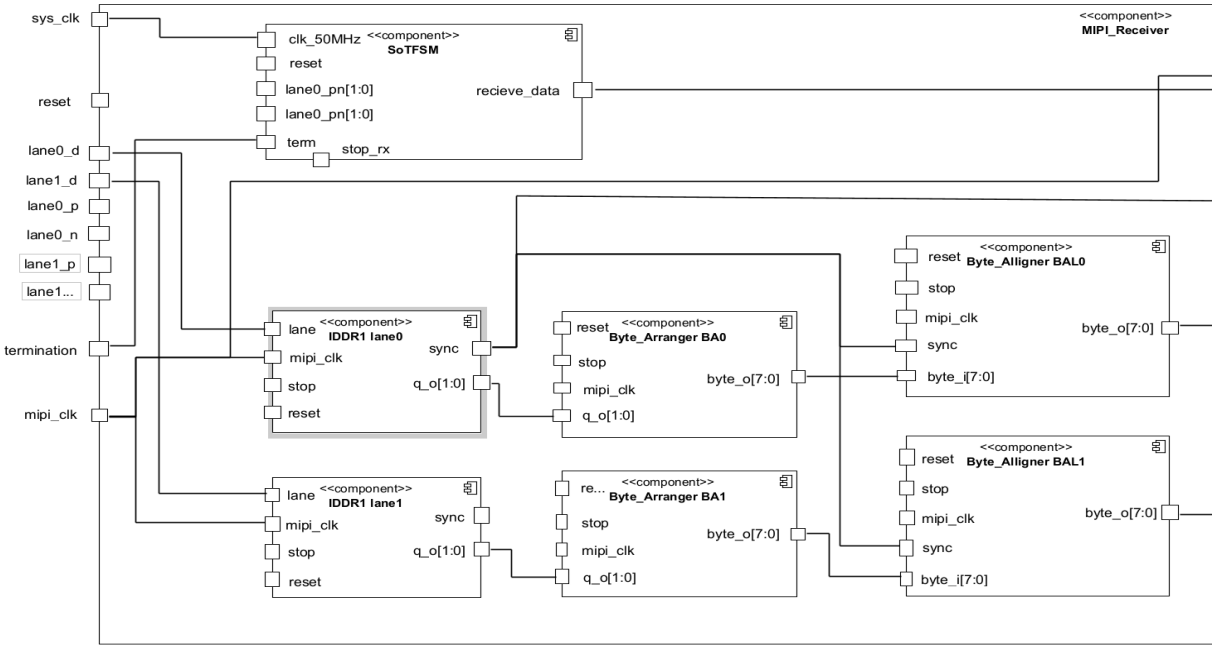


Abbildung 4.20: Komponentendiagramm des MIPI Empfängers

Abbildung 4.20 zeigt das Komponentendiagramm eines Teiles des MIPI Empfängers mit 1:2 Gearing. Dabei ist zu erkennen, dass dieser über drei Eingänge pro Datalane verfügt, darunter ein differentielles sowie zwei single-ended Signale gemäß Abbildung 4.16. Des Weiteren verfügt die Komponente über zwei Clock Eingänge, wobei der Eingang 'sys_clk' für die 50MHz Taktung der SoTFSM benötigt wird, wohingegen es sich bei dem 'mipi_clk' um den von der Kamera erzeugten DDR Clock handelt. Über den Ausgang 'termination' wird die dynamische Terminierung aktiviert.

Um die Kameradaten weiterverarbeiten zu können, müssen diese schrittweise deserialisiert werden. Dazu werden die einzelnen Bitstreams zunächst durch die IDDR1 Komponente auf einen 2-Bit Bus übersetzt, welcher von dem Byte_Arranger in der nächsten Stufe zu 8-Bit parallelisiert wird. Um das korrekte Allignment der Bytes sicherzustellen, werden synchron zu dem Synchronisationsbyte die korrekten Bytes durch den Byte_Alligner ausgegeben.

Die SoTFSM wird dazu benötigt, die einzelnen Start of Transmissions zu erkennen und folgend darauf mittels des Signals 'stop_rx' den Empfang der Daten zu starten. Abbildung 4.21 zeigt die Zustandsmaschine der Komponente, wobei sich die FSM inertial im

Zustand TIMEOUT befindet. Ist nun der LP-11 Zustand gemäß Tabelle 4.8 zu erkennen, wechselt die FSM ebenfalls in den Zustand LP11. Darauf folgend muss durch die Kamera der LP01 sowie der LP00 Zustand angezeigt werden, woraufhin die Zähler 'timer_hs' und timer_term gestartet werden, um im benötigtem Timing, siehe Tabelle 4.8, die Terminierung sowie das Empfangen der Daten zu starten. Sind die Timer abgelaufen, wird der 'term' Ausgang auf HIGH sowie der 'stop_rx' Ausgang auf LOW gesetzt, womit die dynamische Terminierung aktiviert und den weiteren Komponenten das Empfangen der Daten signalisiert wird. Ist die FSM im Zustand SYNC angekommen, so wartet sie auf das 'rec_data' Signal, welches von der Protocoll Komponente erzeugt wird und das Empfangen von gültigen Daten anzeigt. Das Ende einer Übertragung wird ebenfalls durch das Signal 'rec_data' signalisiert, woraufhin die FSM in den Zustand TIMEOUT übergeht und Terminierung sowie den Datenempfang deaktiviert. Um Fehler in der Datenübertragung zu berücksichtigen, wurde außerdem ein Timeout implementiert, mit welchem bei Überschreitung der zulässigen Zeit in den TIMEOUT Zustand zurückgesprungen wird.

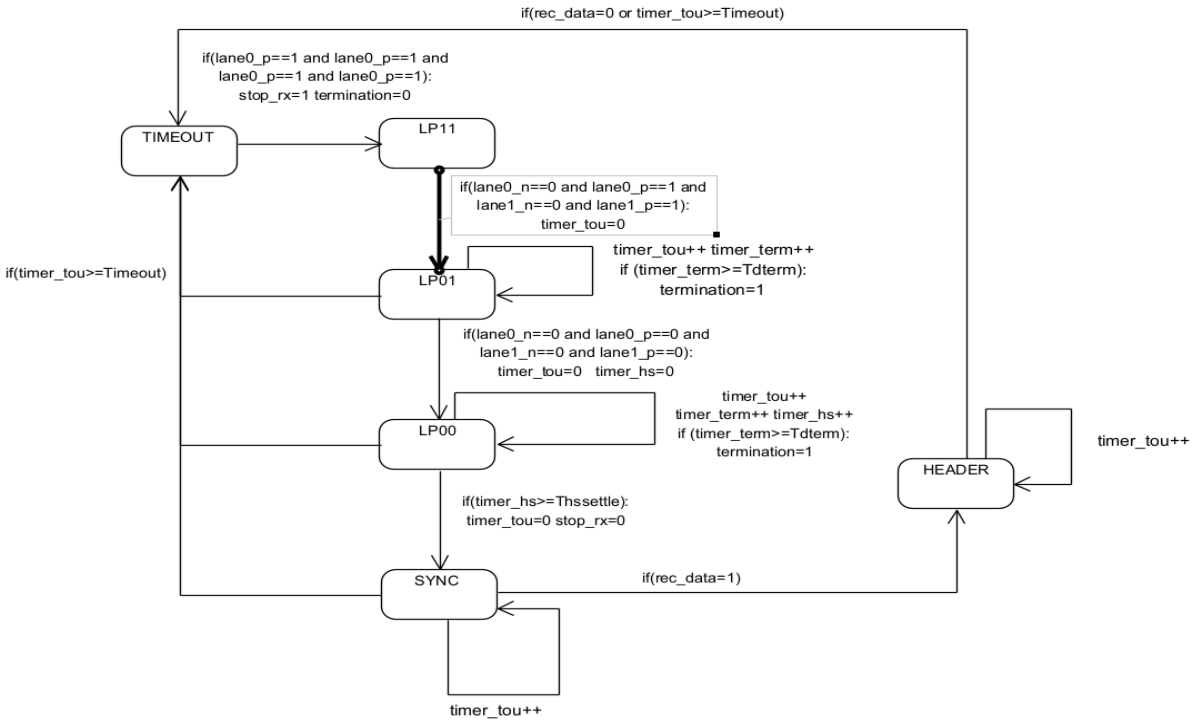


Abbildung 4.21: Zustandsmaschine der SoTFSM

Listing 4.6 zeigt die Deklaration der Zeitkonstanten gemäß Tabelle 4.8, wobei eine Periode der 50 MHz Clock 20 ns entspricht. Es muss beachtet werden, dass es sich um eine DDR Clock handelt, wodurch eine Periode der Mipiclock zwei Bitzeiten UI entspricht. Der Timeout wurde auf etwa die eineinhalb-fache Zeit einer Zeilenübertragung gelegt, womit das Detektieren der nachfolgenden SOT trotz Fehlerfall sichergestellt ist. Des Weiteren muss erwähnt werden, dass die Werte bei verschiedenen Frequenzen getestet und diese auf Funktion optimiert wurden.

```
1        localparam [31:0]   Timeout=(2000*50/mipi_frec);
2        localparam [31:0]   Tdterm=2+(2*50/mipi_frec);
```

```
3       localparam[31:0]   Thssettle=3+(3*50/mipi_frec);
4
```

Listing 4.6: Zeitkonstanten

Da es sich bei der Mipiclock um eine centered DDR Clock handelt, müssen Primitives genutzt werden, um die Daten mit vorherrschender Datenrate empfangen zu können. Die IDDR1 Komponente ist dabei für das Parallelisieren der über die Datalane übertragenen Daten, sowie für das Erkennen der Synchronisationsbytes zuständig. In Listing 4.7 ist die Implementierung der IDDR1 Komponenten zu finden. Dabei wird in der Komponente das Primitive IDDRX1F eingebunden, welches ein Gearing von 1:2 besitzt und die anliegenden Daten an der Datalane bei steigender und fallenden Flanke der Mipiclock sampelt und an den 2-Bit Ausgang ausgibt. Sobald die SotFSM eine SoT mit dem LOW-Zustand des 'stop_rx' signalisiert, werden bei steigender Taktflanke die empfangenen Daten in das 8-Bit Schieberegister geschoben. Enthält dieses das Synchronisationsbyte 8'b10111000, so wird der Ausgang 'sync' auf High gesetzt und die Komponente legt zyklisch an den Ausgang 'q_o' die empfangenen 2-Bit Werte an.

```
1       module IDDR1(input lane,stop,reset,mipi_clk,output sync,output[1:0] q_o);
2       IDDRX1F 10(.D(lane),.SCLK(mipi_clk),.Q0(ddr[0]),.Q1(ddr[1]),.RST('b0));
3       reg[7:0] byte_r;
4       reg sync_r=0;
5       reg[1:0] q_o_r;
6       assign sync=sync_r;
7       assign q_o=q_o_r;
8       wire[1:0] ddr;
9       always (posedge mipi_clk) begin
10              if(stop==1||reset==1)begin
11                      byte_r<=0;
12                      sync_r<=0;
13              end else begin
14                      byte_r<={ddr,byte_r[7:2]};
15                      sync_r<=(byte_r[7:0]==8'b10111000)?1:sync_r;
16                      q_o_r<=ddr;
17              end
18      end
19      endmodule
20
```

Listing 4.7: IDDR1 Komponente

Die folgende Komponente ist nun dafür zuständig, die empfangenen 2-Bit Werte mittels eines Schieberegisters in 8-Bit Werte zu überführen. Dazu werden ebenfalls, wie in Listing 4.8 dargestellt bei LOW-Zustand des 'stop_rx' Signals die Schiebeoperationen taktsynchron ausgeführt.

```
1       module Byte_Arranger(input reset,stop,mipi_clk,input[1:0] q_o,output[7:0] byte_o);
2       reg[7:0] byte_r;
3       assign byte_o=byte_r;
4       always (posedge mipi_clk) begin
5               if(reset||stop)begin
6                       byte_r<=0;
7               end else begin
8                       byte_r<={q_o,byte_r[7:2]};
9               end
10      end
11      endmodule
12
```

Listing 4.8: Byte Arranger

Da die Bytes, welche von der Byte Arranger Komponente ausgegeben werden, nur alle vier Taktzyklen das korrekte Allignment aufweisen, müssen von der Komponente Byte Alligner synchronisiert durch das Signal 'sync' die korrekten Bytes ausgegeben werden.

Sobald von der IDDR1 Komponente das ’sync’ Signal auf HIGH gesetzt wird, startet der Byte Alligner den Zähler ’counter’ und gibt die übergebenen Bytes bei jedem vier Taktzyklus an den Ausgang ’byte_o’ aus. Somit sind nun an dem Ausgang des Alligners nur noch Bytes im korrekten Allignment vorhanden, weswegen bei den folgenden Komponenten mit ganzzahligen Teilungen der Mipiclock gearbeitet werden kann.

```
1   module Byte_Alligner(input reset,stop,mipi_clk,sync,input[7:0] byte_i,output[7:0] byte_o);
2   reg[7:0] byte_o_r;
3   assign byte_o=byte_o_r;
4   reg[7:0]counter;
5   always (posedge mipi_clk) begin
6           if(reset||stop)begin
7                   byte_o_r<=0;
8                   counter<=0;
9           end else begin
10                  if(sync)begin
11                          counter<=(counter>=4)?1:counter+1;
12                          byte_o_r<=(counter>=4)?byte_i:byte_o_r;
13                  end
14          end
15  end
16  endmodule
17
```

Listing 4.9: Byte Alligner

Abbildung 4.22 zeigt den übrigen Teil des Komponentendiagramms des MIPI Empfängers. Dabei ist zu erkennen, dass die enthaltenen Komponenten nur noch mit der geviertelten beziehungsweise mit der geachtelten Frequenz betrieben werden müssen. Um diese neuen Taktfrequenzen zu erzeugen, wurden die Primitives CLKDIVF genutzt, welche den Takt um den Faktor zwei teilen. Die korrekten Bytes werden nun durch die weiteren Komponenten zu einem 32-Bit Bus gleichgerichtet, woraufhin das Protokoll sowie die Nutzdaten dekodiert werden können.

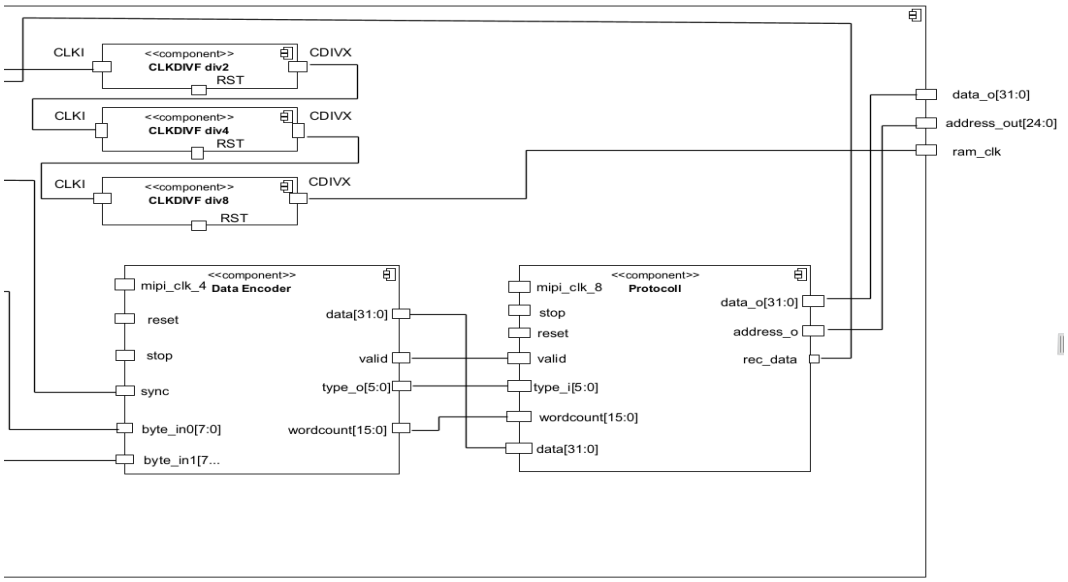


Abbildung 4.22: Komponentendiagramm des MIPI Empfängers

Der Data Encoder ist dazu zuständig, die Bytes beider Lanes korrekt anzuordnen, die jeweiligen Header zu erkennen und zu dekodieren. Dazu wurde ebenfalls die ECC Fehlerkorrektor implementiert. Werden korrekte Header erkannt, soll eine HIGH Zustand am

'valid' Ausgang ausgegeben und die jeweiligen Daten und Datatypes des Headers an den Ausgängen 'wordcount' und 'type_o' angelegt werden. Die nachfolgenden Nutzdaten werden an den Ausgang 'data' angelegt.

```
1   always (posedge mipi_clk_4) begin
2       if(reset||stop)begin
3           out_r<=0;
4           valid_r<=0;
5           start=0;
6           counter<=0;
7           data_r<=0;
8           type_o_r<=0;
9           wordcount_r<=0;
10      end else begin
11          if(sync)begin
12              out_r<={byte_in1,byte_in0,out_r[31:16]};
13              valid_r<=(ecc==regheader_correct[31:24]&&regheader_correct!=0)?1:valid_r;
14              start=(ecc==regheader_correct[31:24]&&regheader_correct!=0)?1:start;
15              type_o_r<=(ecc==regheader_correct[31:24]&&regheader_correct!=0)
16                                  ?regheader_correct[5:0]:type_o_r;
17              wordcount_r<=(ecc==regheader_correct[31:24]&&regheader_correct!=0)
18                                  ?regheader_correct[23:8]:wordcount_r;

19              if(start)begin
20                  counter<=counter+1;
21                  if(counter[0]==0&&counter[1]==1)begin
22                      counter<=1;
23                      data_r<=out_r;
24                  end else begin
25                      counter<=counter+1;
26                  end
27              end
28          end
29      end
30  end
31
```

Listing 4.10: Implementierung Data Encoder

Zunächst müssen die von den einzelnen Lanes empfangenen Bytes richtig angeordnet werden. Dazu werden die Bytes in Listing 4.10 Zeile 12 in das Schieberegister 'out_r' eingeschoben. Ein korrekter Header wird erkannt, indem der berechnete ECC Code mit dem korrigierten Header übereinstimmt, woraufhin der 'valid' Ausgang auf HIGH gesetzt wird. Außerdem werden die TypeID und der Wordcount an den Ausgang geschalten, wodurch die folgende Protocoll Komponente das Protokoll dekodieren kann. Wurde ein vollständiger Header empfangen, werden bei jedem zweiten Taktzyklen des geviertelten Mipiclocks die Nutzdaten an den Ausgang 'data_o' angelegt.

Die Komponente Protocoll ist für die Dekodierung der Datenpakete sowie für das Erzeugen der Daten- und Adressleitungen zuständig, um die empfangenen Pixeldaten in beispielsweise einem RAM zu speichern. Des Weiteren muss in dieser die CRC Überprüfung implementiert werden, um fehlerhafte Übertragungen zu erkennen.

Abbildung 4.23 zeigt dabei die zugehörige Zustandsmaschine der Protocoll Komponente. Dabei wird im Zustand IDLE begonnen und das Register für die CRC Überprüfung auf 16'hffff initialisiert. Wird nun ein korrekter Header erkannt und enthält dieser die TypeID für RAW8 Daten mit einer Pixelanzahl von 'h0280, wird in den Zustand DATA gewechselt und das Hilfssignal 'rec_data' auf HIGH gesetzt, welches der SoTFSM die erfolgreiche Dekodierung des Headers signalisiert. In diesem wird ein Zähler inkrementiert, die Nutzdaten und Adresse an den Ausgang geschaltet sowie die neu berechnete Prüfsumme in das Register 'c' geschrieben. Hat der Zähler den Wert des Registers 'count_val' erreicht, wird 'rec_data' auf LOW gelegt, der Zähler zurückgesetzt und das Prüfsummenregister wieder auf 16'hffff initialisiert. Somit wird der SoTFSM das Ende der Zeilenübertragung signali-

siert. Bei dem Start oder Ende eines Frames werden Short Packets mit TypeID 6'h00 oder 6'h01 versendet, durch welche die Zustandsmaschine das Adressenregister 'counter_addr' resettet wird. Stimmt der errechnete CRC Code mit dem empfangenen überein, wird das Debugsignal auf HIGH gesetzt, wodurch die Übertragungsqualität mittels Logicanalysers gemessen werden kann.

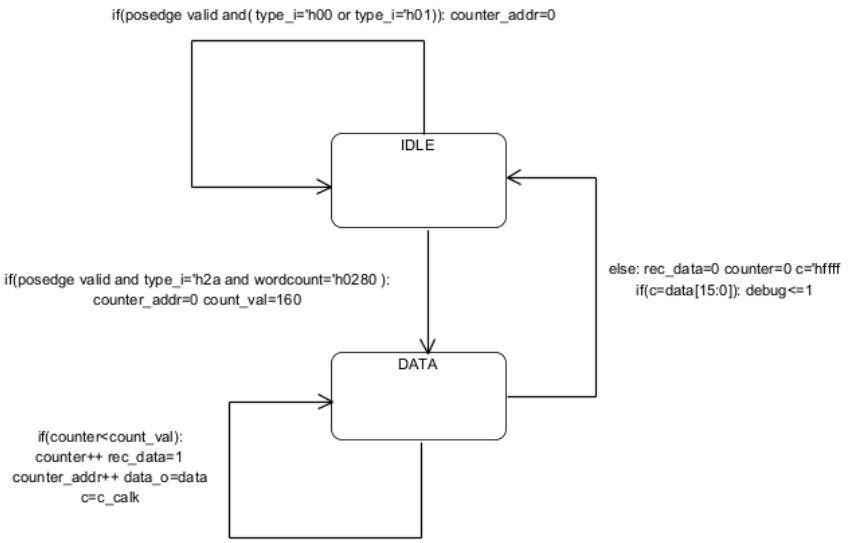


Abbildung 4.23: Zustandsmaschine der Protocoll Komponente

Der MIPI Receiver verfügt nun als Ausgänge über eine 24-Bit RAM Adressierungsleitung, über einen 32-Bit Datenbus und über eine Ramclock. Da insgesamt pro Zeile 640 Pixel übertragen werden, die Daten aber in 32-Bit Blöcken empfangen werden, muss die Ramclock mit einem Achtel der Mipiclock betrieben werden. Entsprechend muss die Zustandsmaschine nur bis zu ein Viertel von 640 zählen.

4.2.5 ECC und CRC

Abbildung 4.24 zeigt den prinzipiellen Aufbau der empfägerseitigen ECC Fehlerkorrektur.[11, S. 51–56]

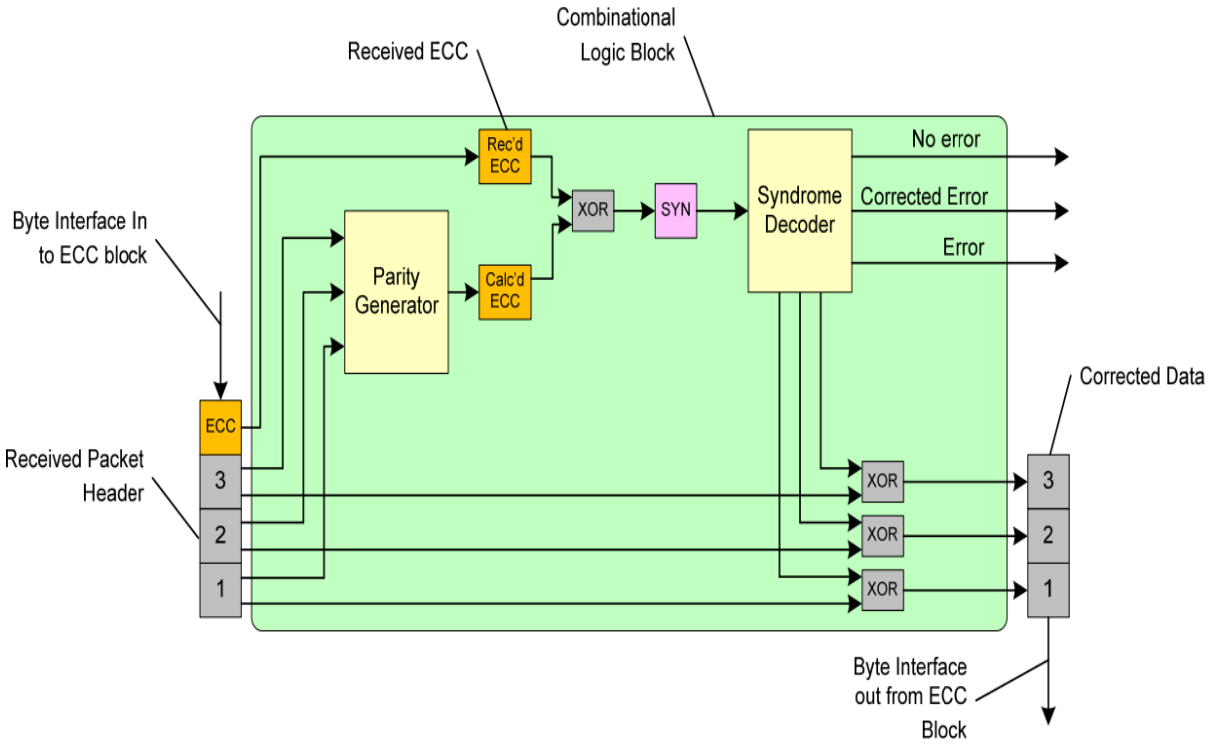


Abbildung 4.24: ECC Error Correction[11, S. 56]

Es müssen zunächst aus dem empfangenen 24-Bit Header der 8-Bit Error Correction Code berechnet werden. Dieser berechnete ECC kann nun mittels einer bitweisen XOR Verknüpfung mit dem empfangenen ECC verglichen werden. Das Ergebnis der XOR Rechnung wird Syndrom genannt. Besitzt das Syndrom den Wert Null, wurde der Header fehlerfrei empfangen. Des Weiteren entstehen bei 1-Bit Fehlern 24 unterschiedliche Bytes, welche eine Charakteristik für die jeweilige fehlerhafte Bit-Nummer aufweisen. Daraus kann die 1-Bit Korrektur durch eine weitere bitweise XOR Verknüpfung erreicht werden. Tabelle 4.10 zeigt dabei die Regeln für die Berechnung des ECC Codes. So müssen für die Berechnung des ersten Bits des ECC Codes die folgenden Bits des empfangenen Headers mit XOR verknüpft werden.[11, S. 56]

$$P0_{\text{24-Bit}} = D0 \wedge D1 \wedge D2 \wedge D4 \wedge D5 \wedge D10 \wedge D11 \wedge D13 \wedge D16 \wedge D20 \wedge D21 \wedge D22 \wedge D23 \tag{4.1}$$

Dabei steht $P0_{\text{24-Bit}}$ für das erste Bit des ECC Codes und $D0$ stellt das erste Bit des empfangenen Headers dar. Es ist zu erkennen, dass jedes Bit, welche in Spalte $P0$ der Tabelle

4.8 den Wert 1 besitzt, Teil der Berechnung ist. So würde beispielsweise $P1_{\text{24-Bit}}$ wie folgt berechnet werden.

$$P1_{\text{24-Bit}} = D0 \wedge D1 \wedge D3 \wedge D4 \wedge D6 \wedge D8 \wedge D10 \wedge D12 \wedge D14 \wedge D17 \wedge D20 \wedge D21 \wedge D22 \wedge D23 \tag{4.2}$$

Durch die bitweise XOR Berechnung des berechneten ECC Codes entsteht wie bereits erwähnt das 8-Bit Syndrom. Dabei können nun drei Fallunterscheidungen getroffen werden. Besitzt das Syndrom den Wert Null, wurde der Header fehlerfrei übertragen. Entspricht das Syndrom einer der Hex Werte in Tabelle 4.10, so ist ein 1-Bit Fehler aufgetreten, welche korrigiert werden kann. Im dritten Fall handelt es sich um eine ungültige Übertragung, welche nicht weiter korrigiert werden kann. In Tabelle 4.10 sind außerdem die korrespondierten fehlerhaften Bitnummern aufgelistet, welche den jeweiligen Syndromen zuzuordnen sind. Durch eine einfache Invertierung des zugehörigen Bits kann der Header korrigiert werden.[11, S. 51–56]

Bit-Nummer	P7	P6	P5	P4	P3	P2	P1	P0	Hex
0	0	0	0	0	0	1	1	1	0x07
1	0	0	0	0	1	0	1	1	0x0B
2	0	0	0	0	1	1	0	1	0x0D
3	0	0	0	0	1	1	1	1	0x0E
4	0	0	0	1	0	0	1	1	0x13
5	0	0	0	1	0	1	0	1	0x15
6	0	0	0	1	0	1	1	0	0x16
7	0	0	0	1	1	0	0	1	0x19
8	0	0	0	1	1	0	1	0	0x1A
9	0	0	0	1	1	1	0	0	0x1C
10	0	0	1	0	0	0	1	1	0x23
11	0	0	1	0	0	1	0	1	0x25
12	0	0	1	0	0	1	1	0	0x26
13	0	0	1	0	1	0	0	1	0x29
14	0	0	1	0	1	0	1	0	0x2A
15	0	0	1	0	1	1	0	0	0x2C
16	0	0	1	1	0	0	0	1	0x31
17	0	0	1	1	0	0	1	0	0x32
18	0	0	1	1	0	1	0	0	0x34
19	0	0	1	1	1	0	0	0	0x38
20	0	0	0	1	1	1	1	1	0x1F
21	0	0	1	0	1	1	1	1	0x2F
22	0	0	1	1	0	1	1	1	0x37
23	0	0	1	1	1	0	1	1	0x3B

Tabelle 4.10: Regeln für ECC Generierung

Listing 4.11 zeigt die Implementierung der ECC Fehlerkorrektur inmitten der Data Encoder Komponente. Dabei wurden die einzelnen Bits des ECC Codes gemäß Tabelle 4.10 berechnet, wodurch letztendlich das Syndrom berechnet werden kann. Da 24 verschiedene charakteristische Bytes entstehen, kann das 24-Bit Wire 'correction' berechnet werden, welches an der Stelle des fehlerhaften Bits eine Eins enthält. Durch die XOR Verknüpfung dieses Wires mit dem empfangenen Header wird die Invertierung des fehlerhaften Bytes erreicht.

```
1   assign ecc[0]=regheader[0]^regheader[1]^regheader[2]^regheader[4]^regheader[5]^
2   regheader[7]^regheader[10]^regheader[11]^regheader[13]^regheader[16]^
3   regheader[20]^regheader[21]^regheader[22]^regheader[23];
4   assign ecc[1]=regheader[0]^regheader[1]^regheader[3]^regheader[4]^regheader[6]^
5   regheader[8]^regheader[10]^regheader[12]^regheader[14]^regheader[17]^
6   regheader[20]^regheader[21]^regheader[22]^regheader[23];
7   assign ecc[2]=regheader[0]^regheader[2]^regheader[3]^regheader[5]^regheader[6]^
8   regheader[9]^regheader[11]^regheader[12]^regheader[15]^regheader[18]^
9   regheader[20]^regheader[21]^regheader[22];
10  assign ecc[3]=regheader[1]^regheader[2]^regheader[3]^regheader[7]^regheader[8]^
11  regheader[9]^regheader[13]^regheader[14]^regheader[15]^regheader[19]^
12  regheader[20]^regheader[21]^regheader[23];
13  assign ecc[4]=regheader[4]^regheader[5]^regheader[6]^regheader[7]^regheader[8]^
14  regheader[9]^regheader[16]^regheader[17]^regheader[18]^regheader[19]^
15  regheader[20]^regheader[22]^regheader[23];
16  assign ecc[5]=regheader[10]^regheader[11]^regheader[12]^regheader[13]^regheader[14]^
17  regheader[15]^regheader[16]^regheader[17]^regheader[18]^regheader[19]^
18  regheader[21]^regheader[22]^regheader[23];
19  assign ecc[6]=0;
20  assign ecc[7]=0;
21
22  assign syndrom=ecc^regheader[31:24];
23
24  assign correction[0]=syndrom==8'h07;
25  assign correction[1]=syndrom==8'h0B;
26  assign correction[2]=syndrom==8'h0D;
27  assign correction[3]=syndrom==8'h0E;
28  assign correction[4]=syndrom==8'h13;
29  assign correction[5]=syndrom==8'h15;
30  assign correction[4]=syndrom==8'h16;
31  assign correction[7]=syndrom==8'h19;
32  assign correction[8]=syndrom==8'h1A;
33  assign correction[9]=syndrom==8'h1C;
34  assign correction[10]=syndrom==8'h23;
35  assign correction[11]=syndrom==8'h25;
36  assign correction[12]=syndrom==8'h26;
37  assign correction[13]=syndrom==8'h29;
38  assign correction[14]=syndrom==8'h2A;
39  assign correction[15]=syndrom==8'h2C;
40  assign correction[16]=syndrom==8'h31;
41  assign correction[17]=syndrom==8'h32;
42  assign correction[18]=syndrom==8'h34;
43  assign correction[19]=syndrom==8'h38;
44  assign correction[20]=syndrom==8'h1F;
45  assign correction[21]=syndrom==8'h2F;
46  assign correction[22]=syndrom==8'h37;
47  assign correction[23]=syndrom==8'h3B;
48
49  assign regheader_correct=regheader^ {8'h00,correction};
```

Listing 4.11: Implementierung ECC

Die CRC Prüfsumme besteht aus einem 16-Bit Code, welcher zyklisch bei dem Empfangen jedes Bytes berechnet werden muss. Die Prüfsumme wird dabei durch folgendes Polynom spezifiziert.[11, S. 56–57]

$$P = x^{16} + x^{12} + x^5 + x^0 \tag{4.3}$$

Die Implementierung kann nun durch ein 16-Bit Schieberegister vergleichsweise Abbildung 4.25 realisiert werden. Dabei wird das Schieberegister auf den Wert 16'hFFFF initialisiert. Des Weiteren muss jedes empfangene Bit durch die unten dargestellten XOR Verknüpfungen in das Register eingeschoben werden. Da im Zusammenhang mit dem MIPI Interface Bits mit einer Datenrate von über 900MBit/s empfangen werden, ist das

Schieben von einzelnen Bits nicht möglich, da dies eine zu hohe Taktrate erfordert. Aus diesem Grund wird im Folgenden das Ergebnis aus 32 aufeinanderfolgenden Schiebeoperationen berechnet. Somit kann mit wesentlich geringerer Taktrate ein gleiches Ergebnis erreicht werden. [11, S. 56–57]

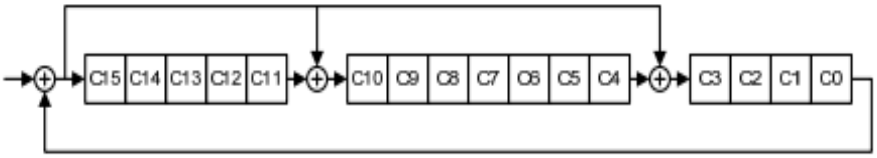


Abbildung 4.25: CRC Schieberegister[11, S. 57]

Bei mehrfacher Iteration der Schiebeoperation ergibt sich dabei für den Wert von $C15$ folgendes Ergebnis.

$$Iteration\ 1: C_{\text{neu}}15 = C0 \wedge D0 \tag{4.4}$$

$$Iteration\ 2: C_{\text{neu}}15 = C2 \wedge D2 \tag{4.5}$$

$$\vdots \quad \vdots$$

$$Iteration\ 4: C_{\text{neu}}15 = C4 \wedge C0 \wedge D0 \wedge D4 \tag{4.6}$$

$$Iteration\ 5: C_{\text{neu}}15 = C5 \wedge C1 \wedge D1 \wedge D5 \tag{4.7}$$

$$\vdots \quad \vdots$$

$$\begin{aligned} Iteration\ 32: C_{\text{neu}}15 = {} & C9 \wedge D9 \wedge D20 \wedge C5 \wedge D5 \wedge \\ & C12 \wedge D12 \wedge C4 \wedge D4 \wedge C11 \wedge C3 \wedge D3 \wedge \\ & D11 \wedge D19 \wedge D23 \wedge D27 \wedge D31 \end{aligned} \tag{4.8}$$

Dabei steht C für die alten Werte des Registers, D für die empfangenen Daten, mit welchen die CRC berechnet werden soll und C_{neu} für die neu berechnete Prüfsumme. Die gleiche Berechnung kann nun für jedes Bit durchgeführt werden. Somit kann nun ebenfalls die CRC Prüfsumme berechnet werden.

```
1  assign c_calk[0]=d[21]^d[10]^c[10]^d[28]^d[6]^c[6]^d[24]^d[13]^c[13]^d[20]^
2  d[5]^c[5]^d[12]^c[12]^d[4]^c[4]^d[0]^c[0];
3  assign c_calk[1]=d[22]^d[11]^c[11]^d[0]^c[0]^d[29]^d[7]^c[7]^d[25]^d[14]^
4  c[14]^d[21]^d[6]^c[6]^d[13]^c[13]^d[5]^c[5]^d[1]^c[1];
5  assign c_calk[2]=d[23]^d[12]^c[12]^d[1]^c[1]^d[30]^d[8]^c[8]^d[26]^d[15]^
6  c[15]^d[22]^d[7]^c[7]^d[14]^c[14]^d[6]^c[6]^d[2]^c[2];
7  assign c_calk[3]=d[24]^d[13]^c[13]^d[2]^c[2]^d[31]^d[9]^c[9]^d[27]^d[16]^
8  d[23]^d[8]^c[8]^d[15]^c[15]^d[0]^c[0]^d[7]^c[7]^d[3]^c[3];
9  assign c_calk[4]=d[20]^d[16]^d[12]^c[12]^d[8]^c[8]^d[0]^c[0]^d[25]^d[14]^
10 c[14]^d[3]^c[3]^d[21]^d[17]^d[6]^c[6]^d[13]^c[13]^d[9]^
11 c[9]^d[5]^c[5]^d[1]^c[1];
12 assign c_calk[5]=d[21]^d[17]^d[13]^c[13]^d[9]^c[9]^d[1]^c[1]^d[26]^d[15]^
13 c[15]^d[4]^c[4]^d[22]^d[0]^c[0]^d[18]^d[7]^c[7]^d[14]^
14 c[14]^d[10]^c[10]^d[6]^c[6]^d[2]^c[2];
15 assign c_calk[6]=d[22]^d[18]^d[14]^c[14]^d[10]^c[10]^d[2]^c[2]^d[27]^d[16]^
16 d[5]^c[5]^d[23]^d[1]^c[1]^d[19]^d[8]^c[8]^d[15]^c[15]^
17 d[11]^c[11]^d[7]^c[7]^d[3]^c[3];
18 assign c_calk[7]=d[23]^d[19]^d[15]^c[15]^d[11]^c[11]^d[3]^c[3]^d[28]^d[17]^
19 d[6]^c[6]^d[24]^d[2]^c[2]^d[20]^d[9]^c[9]^d[16]^d[12]^
20 c[12]^d[8]^c[8]^d[4]^c[4]^d[0]^c[0];
21 assign c_calk[8]=d[24]^d[20]^d[16]^d[12]^c[12]^d[4]^c[4]^d[29]^d[18]^d[7]^
22 c[7]^d[25]^d[3]^c[3]^d[21]^d[10]^c[10]^d[17]^d[13]^c[13]^
23 d[9]^c[9]^d[5]^c[5]^d[1]^c [1];
24 assign c_calk[9]=d[25]^d[21]^d[17]^d[13]^c[13]^d[5]^c[5]^d[30]^d[19]^d[8]^
25 c[8]^d[26]^d[4]^c[4]^d[22]^d[11]^c[11]^d[18]^d[14]^c[14]^
26 d[10]^c[10]^d[6]^c[6]^d[2]^c[2];
27 assign c_calk[10]=d[26]^d[22]^d[18]^d[14]^c[14]^d[6]^c[6]^d[31]^d[20]^d[9]^
```

```
28              c[9]^d[27]^d[5]^c[5]^d[23]^d[12]^c[12]^d[19]^d[15]^c[15]^
29              d[11]^c[11]^d[0]^c[0]^d[7]^c[7]^d[3]^c[3];
30              assign c_calk[11]=d[27]^d[16]^d[5]^c[5]^d[23]^d[1]^c[1]^d[19]^d[8]^c[8]^
31              d[15]^c[15]^ d[0]^c[0]^d[7]^c[7];
32              assign c_calk[12]=d[28]^d[17]^d[6]^c[6]^d[24]^d[2]^c[2]^d[20]^d[9]^c[9]^
33              d[16]^d[1]^c[1]^d[8]^c[8]^d[0]^c[0];
34              assign c_calk[13]=d[29]^d[18]^d[7]^c[7]^d[25]^d[3]^c[3]^d[21]^d[10]^c[10]^
35              d[17]^d[2]^c[2]^d[9]^c[9]^d[1]^c[1];
36              assign c_calk[14]=d[30]^d[19]^d[8]^c[8]^d[26]^d[4]^c[4]^d[22]^d[11]^c[11]^
37              d[18]^d[3]^c[3]^d[10]^c[10]^d[2]^c[2];
38              assign c_calk[15]=d[31]^d[20]^d[9]^c[9]^d[27]^d[5]^c[5]^d[23]^d[12]^c[12]^
39              d[19]^d[4]^c[4]^d[11]^c[11]^d[3]^c[3];
40
```

Listing 4.12: Implementierung CRC

Listing 4.12 zeigt die Implementierung der CRC Prüfsumme in der Komponente Protocoll, welche gemäß Abbildung 4.25 berechnet wird. Dabei handelt es sich bei c[n] um den derzeitigen Wert des n-ten Bits der Prüfsumme, bei d[n] um die neu empfangenen Werte und bei c_calk[n] um den neu berechneten Wert der Prüfsumme. Bei dem Empfangen der nächsten 32-Bit Nutzdaten wird c_calk auf c zugewiesen und der dafür entsprechende neue Wert von c_calk berechnet.

4.2.6 MIPI Receiver IDDRX2F

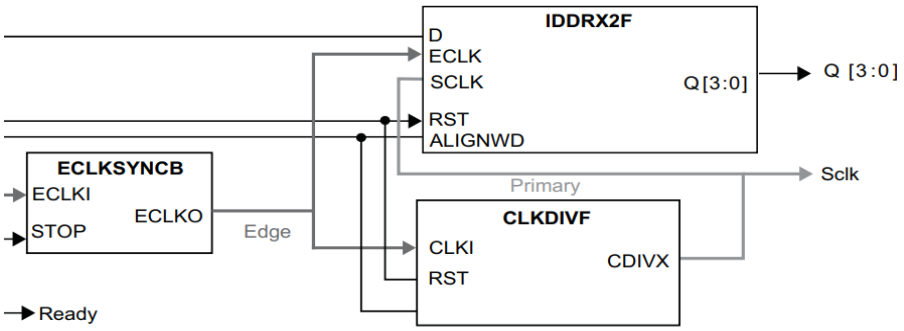


Abbildung 4.26: Anwendungsbeispiel IDDRX2F[12, S. 9]

Um höhere Datenraten bis zu 800Mbit/s pro Lane zu erreichen, kann ebenfalls zum Empfangen der Daten auf das IDDRX2F Primitiv zurückgegriffen werden. Dabei ergeben sich jedoch einige Schwierigkeiten, welche durch unterschiedliche Phasenlagen relativ zu dem Synchronisationsbyte bei dem Halbieren des Mipiclocks durch Clockdivider enstehen.

Abbildung 4.26 zeigt den prinzipiellen Aufbau des IDDRX2F Primitives mit den zusätzlich benötigten Komponenten. Dem IDDR Modul wird dabei die DDR-Clock, die Datenlane sowie die halbierte DDR-Clock übergeben, welche durch das weitere Primitive CLKDIVF erzeugt werden muss. Das ECLKSYNCB Primitiv wird dazu benötigt, die von der Kamera erzeugte DDR-Clock zu aktivieren bzw. zu deaktivieren sowie diese Clock den beiden folgenden Primitives synchronisiert zur Verfügung zu stellen.[12, S. 9]

Die oben genannte Schwierigkeit entsteht nun dadurch, dass für den Ausgang des CLKDIVF zwei mögliche Phasenlagen möglich sind.

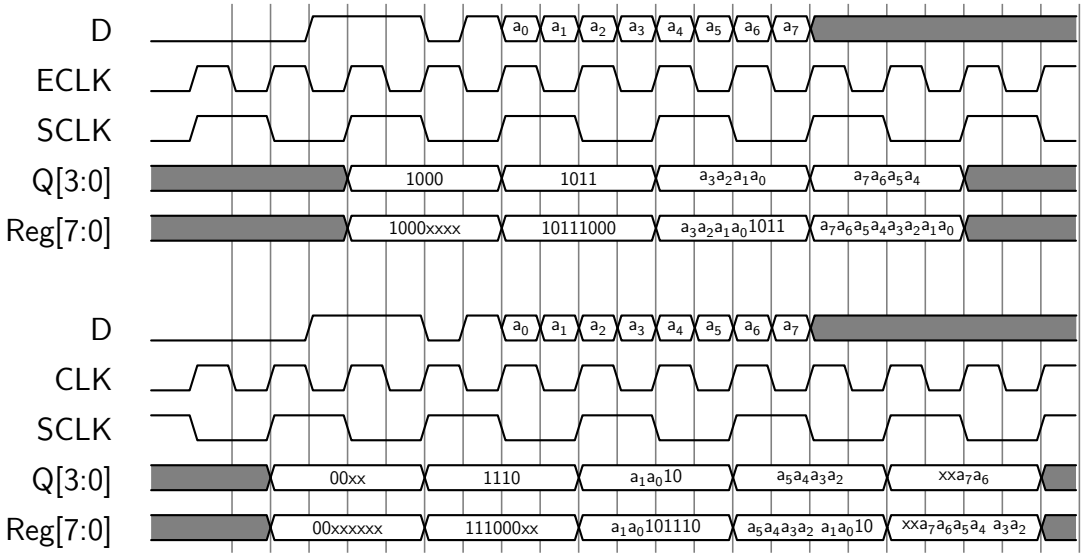


Abbildung 4.27: Unterschiedliche Phasenlagen des halbierten DDR-Clocks

Abbildung 4.27 zeigt die Auswirkungen einer Phase von 180 Grad an der SCLK. Dabei stellt D die Datalane, ECLK die Mipiclock und bei SCLK die halbierte Mipiclock dar. Bei Reg[7:0] handelt es sich um ein 8-Bit Schieberegister, in welches bei steigender Taktflanke von SCLK die 4-Bit Daten des Ausgangs der IDDRX2F Komponente eingeschoben werden. Hierbei wurde zur Verbesserung der Anschaulichkeit für Q[3:0] nur die Werte bei steigender Taktflanke von SCLK dargestellt, welche sich jedoch in der Praxis bei steigender und fallender Taktflanke von ECLK ändern.

In der ersten Waveform ist zu erkennen, dass das Allignment der Bytes korrekt ist und die Bytes im richtigen Format angeordnet sind. Im zweiten Fall jedoch wird deutlich, dass durch die Phasenlage der SCLK das Allignment der Bytes fehlerhaft ist. Aus diesem Grund muss im folgenden mit einem 2-Bit Overflow aus dem vorherig empfangenen Byte gearbeitet werden. Außerdem müssen nun die Bytes für beide Phasenlagen parallel berechnet werden. Im Folgenden wird der erste Fall Ëvenünd die 180 Grad Phasenlage Üneven"genannt. Entdeckt der Empfänger das Synchronisationsbyte 8'b10111000 handelt es sich um eine Even Phasenlage, wohingegen es sich um eine Uneven Phasenlage handelt, wenn das Synchronisationsbyte dem Wert 8'bxx101110 entspricht.

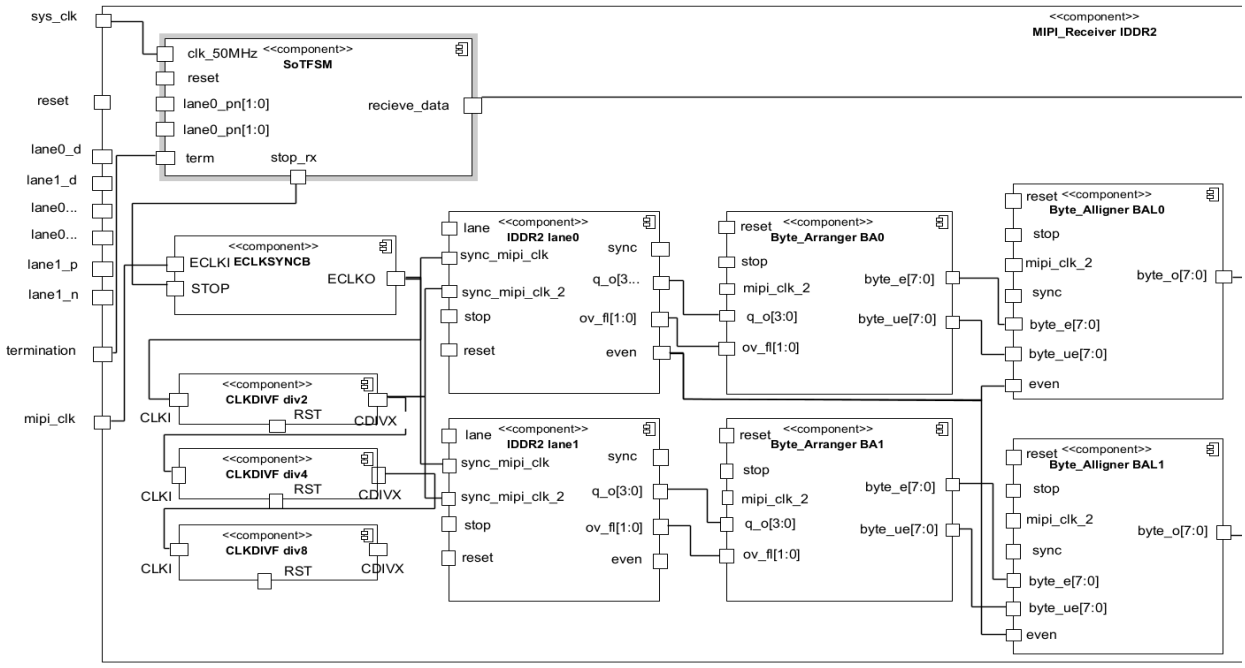


Abbildung 4.28: Komponentendiagramm des IDDR2 MIPI Empfängers

Abbildung 4.28 zeigt die Implementierung des MIPI Empfängers mit IDDR2 Primitives. Gemäß Abbildung 4.27 wurde hier eine ECLKSYNCB Komponente genutzt, um die MIPI-Clock zu schalten, welche dann durch die CLKDIVF Komponenten geteilt wird. Zu sehen ist, dass nun die IDDR2 Komponente über zwei Takteingänge verfügt außerdem können die nachgelagerten Komponenten mit halbierter Taktrate betrieben werden.

```
1           assign detect_e=syncbyte^8'b10111000;
2           assign detect_ue=((8'b00111111)&syncbyte)^8'b00101110;
3           IDDRX2F IDDR (.D(lane),.ECLK(sync_mipi_clk),.SCLK(sync_mipi_clk_2),.RST(reset||stop)
4                   ,.Q0(ddr[0]),.Q1(ddr[1]),.Q2(ddr[2]),.Q3(ddr[3]));
5           always (posedge sync_mipi_clk_2) begin
6                   if(reset||stop)begin
7                           sync_r<=0;
8                           even_r<=0;
9                           ov_fl_r<=0;
10                          q_o_r<=0;
11                          syncbyte=0;
12                  end else begin
13                          syncbyte={ddr,syncbyte[7:4]};
14                          sync_r<=(detect_e==0||detect_ue==0)?1:sync_r;
15                          if(detect_e==0)begin
16                                  even_r<=1;
17                          end
18                          if (detect_ue==0) begin
19                                  even_r<=0;
20                          end
21                          q_o_r<=ddr;
22                          ov_fl_r<=q_o_r[3:2];
23                  end
24          end
25
```

Listing 4.13: IDDR2 Impelementierung

In Listing 4.13 ist die Verilog Implementierung des IDDR2 Moduls zu erkennen. Dabei werden die 4-Bit Werte der IDDRX2F Komponente bei steigender Taktflanke der halbierten Mipiclock in das Schieberegister ’syncbyte’ geschoben. Entspricht nun das Register dem Wert 8’b10111000, wird das ’sync’ sowie das ’even’ Signal auf HIGH gesetzt, da es sich um eine Even Phasenlage handelt. Im Fall einer Uneven Phasenlage, wird im Schiebe-

register der Wert 8'bxx101110 erkannt, wodurch ebenfalls 'sync' auf HIGH gesetzt wird. Jedoch wird das 'even' Signal auf LOW belassen. An den 'ov_fl' Ausgang werden die beiden höchstwertigen Bits der vorherig empfangenen 4-Bit angelegt, wodurch die nachfolgenden Komponenten auch bei einer Uneven Phasenlage die Bytes im korrekten Allignment bestimmen können.

```
1       assign byte_e=byte0_r;
2       assign byte_ue=byte1_r;
3       always (posedge mipi_clk_2) begin
4               if(reset||stop)begin
5                       byte0_r<=0;
6                       byte1_r<=0;
7                       end else begin
8                       byte0_r<={q_o,byte0_r[7:4]};
9                       byte1_r<={q_o[1:0],ov_fl,byte1_r[7:4]};
10              end
11      end
12
```

Listing 4.14: Byte Arranger Implementierung

Ähnlich wie in der IDDR1 Implementierung ist der Byte Arranger (Listing 4.14) für die weitere Parallelisierung der empfangenen Daten zuständig. Das Modul schiebt bei steigender Taktflanke der halbierten Mipiclock die empfangenen 4-Bit Werte in ein 8-Bit Schieberegister. Dabei muss beachtet werden, dass parallel zwei mögliche Bytes entstehen, welche gleichzeitig berechnet werden. Für die Uneven Phasenlage muss das korrekte Byte zusätzlich aus den beiden zusätzlichen Overflowbits errechnet werden.

```
1               assign byte_o_eu=(even)?byte_e:byte_ue;
2               always (posedge mipi_clk_2) begin
3                       if(reset||stop)begin
4                               byte_o_r<=0;
5                               byte_o_r_old<=8'b10111000;
6                               counter<=0;
7                               byte_o_r_s<=0;
8                               end else begin
9                       if(sync)begin
10                              counter<=(counter>=1)?0:counter+1;
11                              byte_o_r<=(counter[0]==0)?byte_o_eu:byte_o_r;
12                      end
13                      end
14              end
15
```

Listing 4.15: Byte Alligner Implementierung

Die Byte Alligner kann nun anhand des 'even' Signals entscheiden, bei welchem Bytes es sich um das Korrekte handelt und legt dieses jeden zweiten Taktzyklus an den Ausgang an. Der noch folgende Aufbau des MIPI Empfängers ist dabei identisch zu der Implementierung mit IDDR1 Komponenten.

4.3 Toplevel Komponente

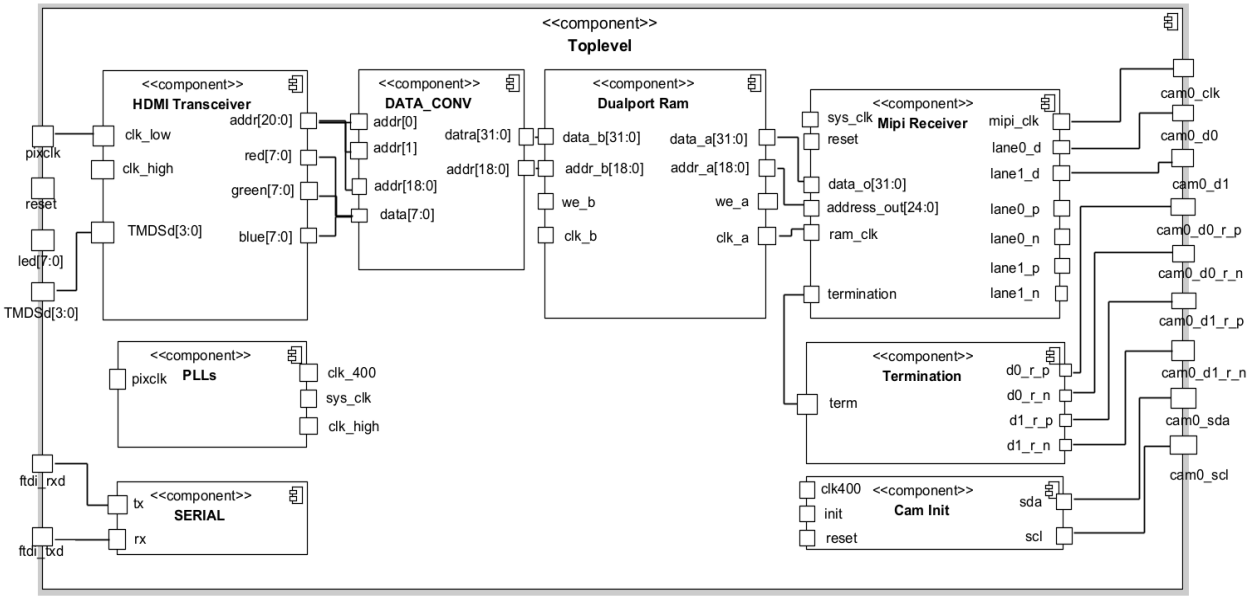


Abbildung 4.29: Komponentendiagramm der Toplevel Komponente

Abbildung 4.29 zeigt den prinzipiellen Aufbau der Toplevel Komponente für die Übertragung der Videodaten per HDMI Interface. Dazu werden zusätzlich zu den schon besprochenen Komponenten noch der Dualport-RAM, die Terminierungsschaltung der DATA_CONV sowie PLLs benötigt. Für Debugging Möglichkeiten wurde ebenfalls eine SERIAL Komponente entworfen, mit welcher verschiedenste Daten über die Serielle Schnittstelle übertragen werden können.

```
1   module Dualport_RAM
2   (
3   input [31:0] data_a,
4   input [16:0] addr_a,input [16:0] addr_b,
5   input we_a, we_b, clk_a, clk_b,
6   output reg [31:0] data_b
7   );
8   reg [31:0] ram[76799:0];
9   always (posedge clk_a)
10          begin
11                  if(we_a)begin
12                  ram[addr_a] <= data_a;
13          end
14  end
15  always (posedge clk_b)begin
16          if(we_b)begin
17                  data_out<=ram[addr_b];
18          end
19  end
20
```

Listing 4.16: Implementierung des Dual Port Rams

Für die Speicherung eines Frames von 640x480 Pixeln im Datenformat RAW8 wird ein Speicher von insgesamt 2.457,6kBit gemäß Tabelle 3.1 benötigt. In Listing 4.16 ist die Implementierung des Dualport-RAMs zu erkennen, wobei jeweils bei steigenden Taktflanken die Daten der zugehörigen Adresse gespeichert bzw. ausgegeben werden. Dabei wird der WriteEnable für Port A dauerhaft auf HIGH belassen, wobei der WriteEnable

Input für Port B dauerhaft auf LOW liegt. Durch dieses Verilog Design werden automatisch durch das Place-and-Root Tool FPGA interne Block-RAM Zellen genutzt, um den benötigten RAM zu entwerfen.

```
1       assign cam0_d0_r_p=(term)?0:'bz;
2       assign cam0_d0_r_n=(term)?0:'bz;
3       assign cam0_d1_r_p=(term)?0:'bz;
4       assign cam0_d1_r_n=(term)?0:'bz;
5
```

Listing 4.17: Terminierung der Single-Ended Pins

Die benötigte Terminierung für das MIPI CSI 2 Interface wird in Listing 4.17 dargestellt, wobei nebenläufig bei HIGH Zustand des 'term' Signals alle vier Pins auf LOW gezogen werden, welche sich sonst im High Impedance Zustand befinden. Die PLL Komponente erzeugt die benötigten Taktraten aus dem 25MHz Oszillator, welcher auf dem ULX3S Board verbaut ist. Es wird dabei ein 400kHz Takt für das I2C Interface, die 50MHz sys_clk für den MIPI Receiver sowie 125MHz für clk_high, welche für den HDMI Transceiver benötigt wird. Bei der DATA_CONV Komponente handelt es sich um ein kombinatorisches Netzwerk, welches anhand der ersten beiden Bits des Adressbusses des HDMI Transceivers die 32-Bit Daten des Dualport-RAMs in vier 8-Bit Bestandteile aufspaltet, wodurch die 32-Bit Daten in 8-Bit Graustufen umgewandelt werden können.

5 Ergebnisse

Im folgenden Kapitel werden nun die Ergebnisse der Implementierung vorgestellt sowie auf auftretende Fehler und Schwachstellen eingegangen.

5.1 HDMI Videostream

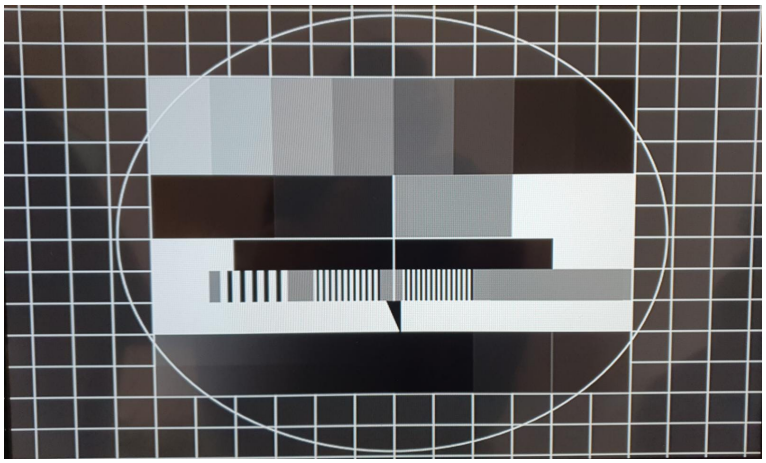


Abbildung 5.1: fehlerfreie Übertragung der HDMI Testpattern

Um das HDMI Interface auf Funktion zu testen, wurden bei unterschiedlichen Auflösungen Testpattern initial in den RAM gespeichert und anschließend über das HDMI Interface ausgegeben. Dabei zeigt Abbildung 5.1 die über HDMI ausgegebene fehlerfreie Testpattern, welches mit einer Auflösung von 640x480 Pixeln bei 60 FPS in Graustufen mit 8-Bit Farbanteil pro Pixel gestreamt wurden. Da bei dem Testen des Interfaces mit Frames, welche über Rot-, Grün- und Blauanteile verfügen, ein höherer Speicher benötigt wird als intern im FPGA vorhanden ist, wurde außerdem der externe SDRAM verwendet, um einzelne größere Testframes abzuspeichern. Dabei war es möglich bis einschließlich einer Auflösung von 1280x720 Pixeln bei 60 FPS statische Frames korrekt darzustellen. Ein korrekt dargestellter Videostream wurde nur bei einer Auflösung von 640x480 Pixeln getestet.

Clock	maximal mögliche Taktrate	genutzte Taktrate
clk_low	140,96MHz	75MHz
clk_high	287,01MHz	375MHz

Tabelle 5.1: HDMI Timinganalyse

Im Hardware Design, in welchem als einzige Komponenten der SDRAM Controller sowie das HDMI Interface vorhanden sind, ergibt sich gemäß Tabelle 5.1 als maximale Taktrate

für die ’clk_low’ 140,96MHz sowie für ’clk_high’ 284,01MHz. Durch die zusätzlichen Komponenten des gesamten Hardware Designs gemäß Abbildung 4.29 wird jedoch die maximal mögliche Taktrate noch weiter verringert.

Clock	maximal mögliche Taktrate	genutzte Taktrate
clk_low	59MHz	25MHz
clk_high	277,32MHz	125MHz

Tabelle 5.2: Gesamtdesign Timinganalyse

Durch die Timinganalyse des Place and Root Tools bei komplettem Hardware Design ergeben sich die maximale Taktfrequenzen von 59MHz für ’clk_low’ und 277,32MHz für ’clk_high’. Zusammengefasst wird bei Vergleich der Timinganalysen mit Tabelle 3.3 deutlich, dass eine theoretische maximale Auflösung von 800x600 Pixeln bei 60 FPS möglich ist. Dabei steht ’clock_low’ für die Pixelclock und ’clock_high’ für die Hälfte der Bitclock. Somit verfügt der genutzte ECP5 FPGA über nicht ausreichend schnelle IO-Buffer und Schaltungskapazitäten um die benötigten Signale für die Mindestauflösung eines 3D Formates von 1280x720 Pixeln bei 60 FPS zu generieren. Des Weiteren stehen an dem genutzten HDMI Connector keine True-Differential Outputs zur Verfügung (Abschnitt 3.3) wodurch eine maximale Datenrate von 500Mbit/s nicht überschritten werden kann. Es wurden keine zusätzlichen Versuche unternommen, um die vorgestellte HDMI Implementierung auf Timingsaspekte zu optimieren.

5.2 Analyse der Kameradaten

Die Kamera IMX219 besitzt einige Einschränkungen, welche die Parametrierung des Clock-Setups betreffen. Laut Datenblatt des Herstellers sind für die Pixelclock der Kamera Werte von 20MHz bis 114.5MHz möglich, wodurch sich für die Mipiclock aufgrund der RAW8 Parametrierung eine DDR-Mipiclock von dem vierfachen der Pixelclock 80MHz bis 458MHz ergibt. Da bei genauerer Betrachtung von Abbildung 4.14 zusätzlich noch PLLs im Clocktree der Kamera zu finden sind, welche für die Generierung der Mipiclock benötigt werden, müssen zusätzlich minimale und maximale Ausgangsfrequenzen dieser beachtet werden. Die PLLs sind auf ein Frequenzintervall von 432MHz bis 916MHz begrenzt, wodurch das Intervall der nutzbaren Frequenzen noch weiter auf 216MHz bis 458MHz eingegrenzt wird. Dies entspricht den Werten von 8’h36 beziehungsweise 8’h72, welche in das Register 16’h030C geschrieben werden müssen. Im realen Versuchsaufbau wurde jedoch festgestellt, dass die Datenpakete bei Frequenzen, welche unterhalb des Wertes von 348MHz liegen, nicht erkannt werden können. Dies entspricht einem Wert von 8’h57 des Registers 16’h030C. Die Implementierung wird im Folgenden bei den Frequenzen 348MHz sowie 456MHz getestet.

Mithilfe der Testpattern der Kamera kann die Qualität der Übertragung getestet werden. Einzelne Videoframes können dabei durch den FPGA empfangen und über die Serielle Schnittstelle an ein USB Interface gesendet werden, wodurch die Frames genau analysiert wurden. Das Debayering wurde nicht implementiert, jedoch kann beispielsweise durch ein Python Skript dieses simuliert werden.

Des Weiteren wurde die Übertragungsqualität anhand der CRC Prüfsummen überprüft. Da die Übereinstimmung der CRC Prüfsumme mit dem Packetfooter an einen Debugpin ausgegeben wird, können mittels Logicanalysers die korrekten bzw. die fehlerhaft übertragenen einzelnen Zeilen identifiziert werden.

5.2.1 IDDR1 Implementierung

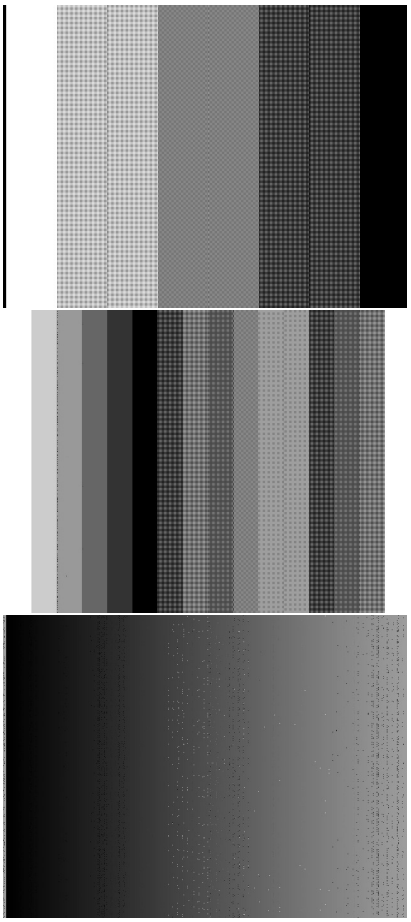


Abbildung 5.2: Drei unterschiedliche Testpattern der Kamera

Abbildung 5.2 zeigt die drei empfangenen Frames der Testpattern. Die Frames wurden in Graustufen ausgeben, wobei einige Übertragungsfehler erkennbar sind. Die Kamera wurde dabei mit einer Datenrate pro Lane von 696MBit/s betrieben. Die empfangenen Frames können nun durch beispielsweise die openCV Python Bibliothek bei Nutzung des korrekten Patterns GBRG gemäß Kameraspezifikation debayert werden. In Abbildung 5.3

sind die debayerten Frames zu erkennen, welche mit Ausnahme der Pixelfehler mit dem im Datenblatt dargestellten Testpattern übereinstimmen.

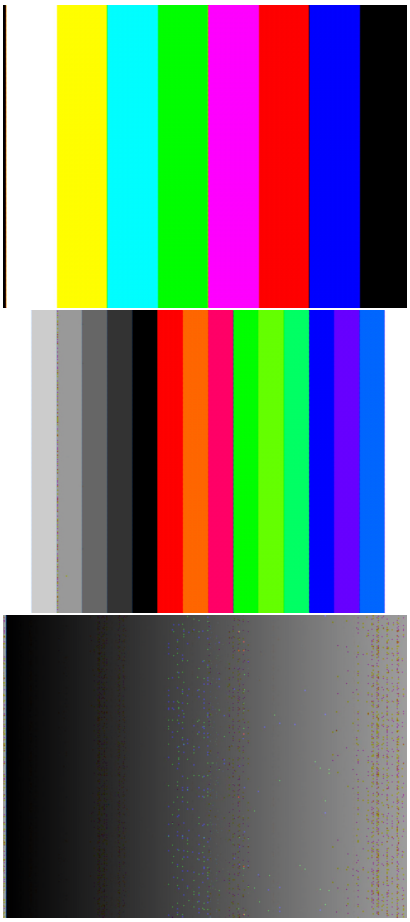


Abbildung 5.3: Debayering der Testframes mit Pattern GBRG

Tabelle 5.3 zeigt die Timinganalyse des MIPI Receivers mit IDDR1 Implementierung des Gesamtdesigns, dabei wurde die Kamera auf eine Mipiclock von 348MHz parametriert. Es ist zu erkennen, dass die Timing Anforderung der Mipiclock nicht erfüllt werden konnte, wobei eine maximale Frequenz von 180MHz angegeben wurde, welche jedoch durch die Kamera nicht möglich ist. Trotz des Überschreitens der maximalen Taktrate bei weitem können die Testpattern erstaunlich gut empfangen werden.

Clock	maximal mögliche Taktrate	genutzte Taktrate
ram_clk	106,41MHz	43,51MHz
sys_clk	141,7MHz	50MHz
mipi_clk	180,28MHz	348,07MHz
mipi_clk_4	109,77MHz	87,02MHz

Tabelle 5.3: Timinganalyse MIPI IDDR1

5.2.2 IDDR2 Implementierung

Abbildung 5.4: Debayering der Testpattern

Die in Abbildung 5.4 dargestellten Testframes wurden mit einer Datenrate von 916MBit/s pro Lane im RAW 8 Format empfangen, über die Serielle Schnittstelle übermittelt und über ein Python Skrit debayert. Die Frames wurden dabei nahezu fehlerfrei übertragen. Nach dem Debayering der Frames entsprechen diese exakt den Testpatterns, wodurch eine fehlerfreie Übertragung erkennbar ist.

Die in Tabelle 5.4 dargestellte Timinganalyse bestätigt zusätzlich, dass nahezu das gesamte Design mit der Mipiclock von 458MHz betrieben werden kann, wobei jedoch die maximale Taktfrequenz für die geviertelte Mipiclock um 3MHz überschritten wird.

Clock	maximal mögliche Taktrate	genutzte Taktrate
ram_clk	89,48MHz	57,26MHz
sys_clk	126,65MHz	50MHz
mipi_clk_4	111,78MHz	114,52MHz
mipi_clk_2	247,04MHz	229,04MHz

Tabelle 5.4: Timinganalyse MIPI IDDR2

Somit ist es durch die IDDR2 Implementierung prinzipiell möglich, die Kameradaten mit maximal möglicher Datenrate von 916Mbit/s pro Lanes zu empfangen, jedoch werden hierbei die maximal möglichen IO Geschwindigkeiten überschritten, wodurch es bei dem Empfangen von realen Kameradaten noch zu einzelnen Pixelfehlern kommt. Dabei wurde die Kamera weiterhin mit einer Mipiclock von 458MHz bei 640x480 Pixel betrieben. Des Weiteren ist zu Erkennen, dass bei der Nutzung von niedrigeren Frequenzen die Übertragungsqualität deutlich reduziert wird. Die Ursache hiervon wurde nicht weiter untersucht.

5.2.3 Frametiming

Abbildung 5.5: Zeilenübertragung D0 Lane MIPI CSI2

Abbildung 5.5 zeigt die erste Datalane einer Übertragung zweier Zeilen des MIPI Protokolls, wobei für eine Zeilenübertragung insgesamt ein Zeitraum von 13,34 µs benötigt wird. Eine Zeilenübertragung besteht dabei aus 3,15 µs Nutzdaten gefolgt von dem Lineblanking. Laut Dokumentation ist ein minimales Lineblanking von 168 Pixelclockzyklen möglich. Da ein Mipiclock von 458MHz einem Pixelclock von ca. 114MHz entspricht, beträgt das minimale Lineblanking also 1,47 µs. Die benötigte Übertragung eines Frames könnte also auf 4,62 µs verringert werden.

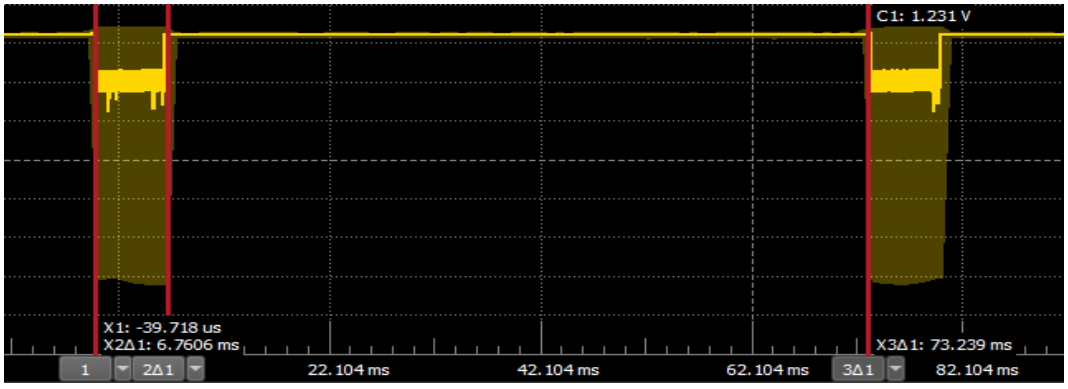


Abbildung 5.6: Frameübertragung D0 Lane MIPI CSI2

Für die Übertragung eines kompletten Frames wird gemäß Abbildung 5.6 eine Zeit von 73,24 ms benötigt, wobei die Nutzdaten innerhalb von 6,76 ms übertragen werden. Somit

ergibt sich eine effektive Framerate von etwa 13 FPS, welche jedoch noch durch die Verkürzung des Framblankings erhöht werden kann. Das minimal mögliche Frameblanking beträgt hierbei 32 Pixelclockzyklen, was etwa einer Zeit von 0,28 µs entspricht. Somit wären Frameraten von bis zu 130 FPS möglich.

5.2.4 Latenzanalyse

Die Latenzzeit des Videostreams kann durch eine Simulation ermittelt werden. Dabei ergibt sich die gesamte Latenz aus der Übertragungszeit eines Kameraframes, der Verarbeitungszeit der Kameradaten sowie dem Senden des Frames über das HDMI Interface. Im vorherigen Abschnitt wurde bereits eine Übertragungszeit der Kamera von 6.7 ms festgestellt.

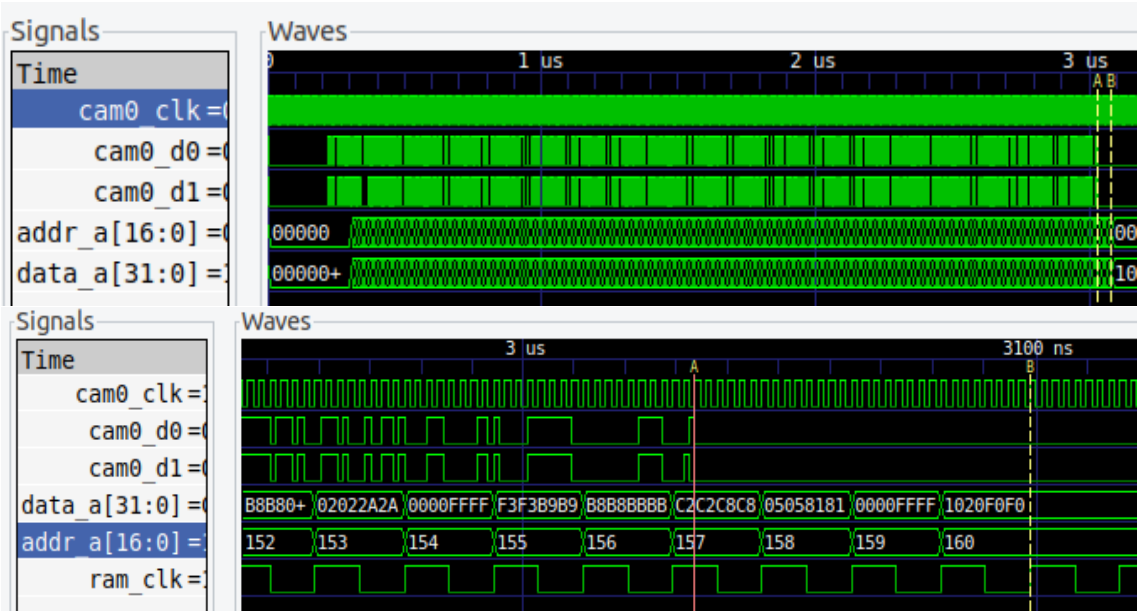


Abbildung 5.7: Simulation einer Zeilenübertragung MIPI CSI2

Abbildung 5.7 zeigt zwei Ausschnitte aus dem Simulationsergebnis einer Zeilenübertragung, wobei eine gute Übereinstimmung zwischen Messung und Simulation der Übertragungsdauer festzustellen ist. Außerdem ist im Zeitintervall A zu B die zusätzliche Verarbeitungszeit dargestellt, welche benötigt wird, um die kompletten Daten in den RAM zu speichern. Diese beträgt 65,4 ns und kann deswegen im Vergleich zu den weiteren Latenzen vernachlässigt werden. Somit kann nun die komplette Latenz des Videostreams von Kamera bis HDMI Transceiver aus der Summe der 6,7 ms Kameraübertragung und der etwa 16 ms HDMI Übertragung ermittelt werden. Die Latenz des HDMI Interfaces wurde ebenfalls mithilfe einer Simulation ermittelt. Somit ergibt sich insgesamt eine Latenz von 22,7 ms, zu welcher jedoch noch die Latenz des genutzten HDMI Monitors addiert werden muss. Zusätzlich muss erwähnt werden, dass durch das hohe Frameblanking große Varianzen in der Latenz entstehen. Da insgesamt mit Frameblanking die Dauer einer Frameübertragung 73,24 ms beträgt, variiert die Gesamtlatenz im Intervall von 22,7 ms

bis zu 89,24 ms. Durch die Verringerung des Frameblankings kann also die Latenzzeit noch deutlich optimiert werden.

6 Zusammenfassung

In dem folgenden Kapitel sollen zum Abschluss die erreichten Ergebnisse zusammengefasst werden und ein Ausblick auf mögliche Erweiterungen gegeben werden.

6.1 Fazit

Ziel dieser Arbeit war die Entwicklung eines latenzarmen FPGA-basierten Stereomikroskopes mittels ausschließlicher Nutzung von Open Source Tools, welches eine Gesamtlatenz von unter 40ms besitzt. Im Rahmen dieser Arbeit wurde es schrittweise erreicht, zunächst ein funktionsfähiges HDMI Interface zu entwerfen, folgend darauf mit gewünschter Parametrierung die Raspberry Cam v2 zu initialisieren und letztendlich ein umfangreiches MIPI CSI 2 Interface zu implementieren. Durch den FPGA können nun die Rohdaten im RAW8 Format mittels des MIPI CSI 2 Interfaces empfangen, in einem RAM zwischengespeichert und anschließend über das HDMI Interfaces in Graustufen als Videostream ausgegeben werden. Dabei wurden die Daten in einer Auflösung von 640x480 Pixeln mit 13 FPS empfangen und mit selber Auflösung bei 60 FPS ausgegeben. Das Einbinden der zweiten Kamera sowie die Implementierung eines HDMI-3D Videostreams wurde nicht erreicht. Es muss erwähnt werden, dass keine Art von Nachbearbeitung des Bildes vorgenommen wurde. Des Weiteren wurde festgestellt, dass die geforderte Höchstlatenz von 40 ms für den Videostream durchaus bei Verringerung des Frameblankings erreicht werden kann, jedoch wurde in dieser Arbeit der Hauptaugenmerk auf den Datenempfang gelegt und die Optimierung der Kameraparametrierung zunächst außen vor gelassen. Es ist also trotz einiger Schwachstellen möglich, High-Speed IO Interfaces durch eine Open Source Toolchain zu projektieren und somit eine solide Videoverarbeitung zu gewährleisten. Für die Umsetzung des Gesamtprojektes müssen zunächst die folgenden drei Problematiken gelöst werden.

Prinzipiell unterstützen 3D-Monitore HDMI-3D Formate erst ab einer Auflösung von 1280x720 Pixeln bei 60 FPS. Diese Auflösung kann jedoch durch die Belegung des HDMI Interfaces auf Fake-Differentials nicht erreicht werden.

Des Weiteren können maximal durch die Nutzung der sysMEM Blocks sowie des Embedded Memorys zwei Frames mit einer Auflösung von 640x480 Pixeln abgespeichert werden, was ebenfalls nicht ausreichend für die Erzeugung eines HDMI 3D Formates ist.

Das Empfangen der Kameradaten mit maximaler Datenrate ist prinzipiell möglich, jedoch wird die maximal angegebenen Datenraten der IO-Buffer überschritten, wodurch noch einzelne Timingfehler entstehen. Eine Verringerung der Mipiclock löst dieses Problem nicht, da hierbei aus nicht geklärten Gründen eine weitaus größere Anzahl von Pixelfehlern zu erkennen sind. Eine maximale Latenz von 40 ms wurde ebenfalls nicht erreicht.

6.2 Ausblick

Einige Problematiken könnten wie folgt gelöst werden. Durch die Nutzung einer PMOD HDMI Platine an True Differentials kann eine Datenrate von bis zu 800Mbit/s erreicht werden, wodurch die benötigte Minimalauflösung erreicht wird. Für ausreichend Speicherkapazität muss ein externer RAM genutzt werden. Um den fehlerfreien Datenempfang der Kameradaten zu gewährleisten, könnten SERDES Komponenten verwendet werden, jedoch werden diese nicht durch die Open Source Toolchain unterstützt.

Neben den Schwachstellen des Projektes, welche durch die verwendete Hardware entstehen, müssen ebenfalls noch weitere Funktionen und Module für das vollständige Stereomikroskop implementiert werden. Dazu gehört einerseits die Nachbearbeitung der Rohdaten sowie die Erweiterung der HDMI Komponente auf HDMI-3D Formate.

Zunächst müssen aus den Rohdaten der Kamera durch ein Debayering Filter die einzelnen Farbverläufe ermittelt werden. Dazu werden die einzelnen Rot-, Grün- und Blauanteile aus den Rohdaten extrahiert und anschließend mittels Interpolation die fehlenden Farbanteile ermittelt. Da das menschliche Auge über kein lineares Helligkeitsempfinden verfügt, müssen die empfangenen Daten nachfolgend noch durch eine Gammakorrektur bearbeitet werden. Dies kann beispielsweise mittels eines LUTs oder RAMs realisiert werden, welcher über eine 8-Bit Adressierung verfügt und die entsprechenden gammakorrigierten Werte bei zugehörigen Adressen enthält.

Eine vielversprechende weitere Möglichkeit für die Realisierung des FPGA Stereomikroskopes stellt die Weiterentwicklung des ULX3S Boards dar. Dabei ist das ULX4M mit einem DDR3 RAM und True-Differential Pairs an dem HDMI Interface ausgestattet, wodurch ein ausreichend schnelles IO-Interface und Speicherplatz für das HDMI Signal und Framebuffering vorhanden ist. Des Weiteren kann das ULX4M auf das Raspberry Pi CM4 IO BOARD aufgesteckt werden, wodurch passende Connectoren für beide Kameras vorhanden sind. Jedoch werden hier die einzelnen Lanes nicht doppelt an den FPGA geführt, somit ist die dynamische Terminierung sowie die bisherige Erkennung der Start of Transmission nicht ohne weiteres möglich.

Literaturverzeichnis

[1] YosysHQ GmbH. *YosysHQ Yosys*. 14. Apr. 2023. URL: https://yosyshq.readthedocs.io/_/downloads/yosys/en/latest/pdf/ (besucht am 19. 04. 2023).

[2] Lattice Semiconductor. *ECP5 and ECP5-5G Family Data Sheet*. Apr. 2021. URL: https://www.latticesemi.com/Products/FPGAandCPLD/ECP5 (besucht am 20. 03. 2023).

[3] Lattice Semiconductor. *ECP5 and ECP5-5G Memory Usage Guide*. Mai 2021. URL: http://www.latticesemi.com/view_document?document_id=50466 (besucht am 04. 04. 2023).

[4] ISSI. *IS42S83200G, IS42S16160G,IS45S83200G, IS45S16160G*. Dez. 2013. URL: https://www.issi.com/WW/pdf/42-45S83200G-16160G.pdf (besucht am 19. 04. 2023).

[5] github/emard. *ULX3S Schematics*. Jan. 2022. URL: https://github.com/emard/ulx3s/commits/master/doc (besucht am 18. 04. 2023).

[6] Lattice Semiconductor. *ECP5 and ECP5-5G sysI/O Usage Guide*. Jan. 2020. URL: https://www.latticesemi.com/-/media/LatticeSemi/Documents/ApplicationNotes/EH/FPGA-TN-02032-1-3-ECP5-ECP5G-sysIO-Usage-Guide.ashx?document_id=50464.

[7] Philips Consumer Electronics International B.V Silicon Image Inc. Sony Cororation Thompson Inc. Toshiba Corporation Hitachi Ltd. Panasonic Corporation. *High-Definition Multimedia Interface Specification Version 1.4*. 5. Juni 2009. URL: https://www.hdmi.org/spec/hdmi1_4b (besucht am 20. 03. 2023).

[8] Raspberry Pi. *Raspberry Pi Cam v2 Circuit*. 24. Mai 2018. URL: https://www.raspberrypi.com/documentation/accessories/camera.html (besucht am 20. 03. 2023).

[9] Sony Corporation. *IMX219PQH5-C*. Apr. 2021. URL: https://github.com/rellimmot/Sony-IMX219-Raspberry-Pi-V2-CMOS/blob/master/RASPBERRY%20PI%20CAMERA%20V2%20DATASHEET%20IMX219PQH5_7.0.0_Datasheet_XXX.PDF (besucht am 20. 03. 2023).

[10] DRAFT MIPI Alliance. *DRAFT MIPI Alliance Specification for D-PHY v1.0*. 22. Sep. 2009. URL: https://www.mipi.org/specifications/d-phy (besucht am 20. 03. 2023).

[11] DRAFT MIPI Alliance. *DRAFT MIPI Alliance Specification for Camera Serial Interface 2 (CSI-2) v1.01*. 2. Apr. 2009. URL: https://www.mipi.org/specifications/csi-2 (besucht am 20. 03. 2023).

[12] Lattice Semiconductor. *ECP5 and ECP5-5G High-Speed I/O Interface*. Nov. 2015. URL: `https://www.raspberrypi.com/documentation/accessories/camera.html` (besucht am 29. 03. 2023).

Anhang

```
1   module TMDS_Encoder(input clklow,input reset,input [1:0]state
2           ,input[7:0] pix_data,input[1:0] H_VSync_Ctr,
3            input[3:0] aux_data,output data_o,output[9:0] q_out);
4   //////////////////////////
5   reg[9:0]q_out1;
6   integer cnt_old=0;
7   wire[7:0] N1qm=(q_m[0]==1)+(q_m[1]==1)+(q_m[2]==1)+(q_m[3]==1)
8                 +(q_m[4]==1)+(q_m[5]==1)+(q_m[6]==1)+(q_m[7]==1);
9   wire[7:0] N0qm=(q_m[0]==0)+(q_m[1]==0)+(q_m[2]==0)+(q_m[3]==0)
10                +(q_m[4]==0)+(q_m[5]==0)+(q_m[6]==0)+(q_m[7]==0);
11  wire[7:0] N1pd=(pix_data[0]==1)+(pix_data[1]==1)+(pix_data[2]==1)+(pix_data[3]==1)
12                +(pix_data[4]==1)+(pix_data[5]==1)+(pix_data[6]==1)+(pix_data[7]==1);
13  wire[7:0] N0pd=(pix_data[0]==0)+(pix_data[1]==0)+(pix_data[2]==0)+(pix_data[3]==0)
14                +(pix_data[4]==0)+(pix_data[5]==0)+(pix_data[6]==0)+(pix_data[7]==0);
15
16
17  wire[8:0] q_m=(reset==1)?0:{q_m8,q_m7,q_m6,q_m5,q_m4,q_m3,q_m2,q_m1,q_m0};
18  wire q_m0=pix_data[0];
19  wire q_m1=(((N1(pix_data)>4)||(N1(pix_data)=='d4 && pix_data[0]=='b0))? q_m0~^pix_data[1]
20                :q_m0^pix_data[1]);
21  wire q_m2=(((N1(pix_data)>4)||(N1(pix_data)=='d4 && pix_data[0]=='b0))? q_m1~^pix_data[2]
22                :q_m1^pix_data[2]);
23  wire q_m3=(((N1(pix_data)>4)||(N1(pix_data)=='d4 && pix_data[0]=='b0))? q_m2~^pix_data[3]
24                :q_m2^pix_data[3]);
25  wire q_m4=(((N1(pix_data)>4)||(N1(pix_data)=='d4 && pix_data[0]=='b0))? q_m3~^pix_data[4]
26                :q_m3^pix_data[4]);
27  wire q_m5=(((N1(pix_data)>4)||(N1(pix_data)=='d4 && pix_data[0]=='b0))? q_m4~^pix_data[5]
28                :q_m4^pix_data[5]);
29  wire q_m6=(((N1(pix_data)>4)||(N1(pix_data)=='d4 && pix_data[0]=='b0))? q_m5~^pix_data[6]
30                :q_m5^pix_data[6]);
31  wire q_m7=(((N1(pix_data)>4)||(N1(pix_data)=='d4 && pix_data[0]=='b0))? q_m6~^pix_data[7]
32                :q_m6^pix_data[7]);
33  wire q_m8=(((N1(pix_data)>4)||(N1(pix_data)=='d4 && pix_data[0]=='b0))? 'b0:'b1);
34
35
36  wire q_out2p1=  ((cnt_old==0)||(N1(q_m[7:0])==N0(q_m[7:0]))) ? !q_m[8] :(((cnt_old>0&&((N1(q_m[7:0])
37                        >N0(q_m[7:0])))||(cnt_old<0 && ((N0(q_m[7:0])>N1(q_m[7:0])))))?'b1:'b0));
38  wire q_out2p2=  ((cnt_old==0)||(N1(q_m[7:0])==N0(q_m[7:0]))) ? q_m[8] : (((cnt_old>0 && ((N1(q_m[7:0])
39                        >N0(q_m[7:0])))||(cnt_old<0 && ((N0(q_m[7:0])>N1(q_m[7:0])))))?q_m[8]:q_m[8]));
40  wire[7:0] q_out2p3=(reset==1)?0:(((cnt_old==0)||(N1(q_m[7:0])==N0(q_m[7:0]))) ? ((q_m[8]==1)?q_m[7:0]:
41                        ~q_m[7:0]) : (((cnt_old>0 &&((N1(q_m[7:0])>N0(q_m[7:0])))
42                        ||(cnt_old<0 &&((N0(q_m[7:0])>N1(q_m[7:0])))))?~q_m[7:0]:q_m[7:0]));
43  wire[9:0] q_out2=(reset==1)?0:{q_out2p1,q_out2p2,q_out2p3};
44
45  wire[31:0] cnt0=cnt_old+(N0(q_m[7:0])-N1(q_m[7:0]));
46  wire[31:0] cnt1=cnt_old+(N1(q_m[7:0])-N0(q_m[7:0]));
47  wire[31:0] cnt2=cnt_old+2*q_m[8]+N0(q_m[7:0])-N1(q_m[7:0]);
48  wire[31:0] cnt3=cnt_old-2*(!q_m[8])+N1(q_m[7:0])-N0(q_m[7:0]);
49  wire[31:0] cnt=(reset==1)?0:(((cnt_old==0)||(N1(q_m[7:0])==N0(q_m[7:0]))) ? ((q_m[8]==0)? cnt0 : cnt1 )
50                        :(((cnt_old>0 && N1(q_m[7:0])>N0(q_m[7:0]))
51                        ||(cnt_old<0 && (N0(q_m[7:0])>N1(q_m[7:0]))))? cnt2:cnt3));
52
53
54  wire[10:0] tmds_cnt=(H_VSync_Ctr[1]==1)?((H_VSync_Ctr[0]==1)?10'b1010101011 :10'b0101010100 ) : ((
    H_VSync_Ctr[0]==1)? 10'b0010101011 : 10'b1101010100);
55
56  always (posedge clklow) begin
57          if(reset==1) begin
58                  cnt_old=0;
59                  q_out1=0;
60          end     else begin
61                  cnt_old=cnt;
62                  case (state)
63                          'b00: begin     //h0 refers to Control Period coding
64                                  cnt_old=0;
65                                  q_out1=tmds_cnt;
66                          end
67                          2'b01: begin    //h2 refers to Video Data coding
68                                  q_out1=q_out2;
69                          end
70                          default: begin
71                                  q_out1='b0000000000;
72                          end
73                  endcase
74          end
75  end
76  assign q_out=q_out1;
77
78  function automatic [7:0] N0(input[7:0 ]data);
79          begin
80                  N0=(data[0]==0)+(data[1]==0)+(data[2]==0)+(data[3]==0)+(data[4]==0)+(data[5]==0)
```

```
                                +(data[6]==0)+(data[7]==0);
        end
endfunction

function automatic [7:0] N1(input[7:0 ]data);
        begin
                N1=(data[0]==1)+(data[1]==1)+(data[2]==1)+(data[3]==1)+(data[4]==1)+(data[5]==1)
                        +(data[6]==1)+(data[7]==1);
        end
endfunction
endmodule



module HDMI_Transciever
#(parameter
h_pixel=640,
h_front_porch=16,
h_back_porch=48,
h_tot_pixel=800,
v_pixel=480,
v_front_porch=10,
v_back_porch=33,
v_tot_pixel=525)
(input clk_low,input clk_high,input reset,input[7:0] red,
                input[7:0] green,input[7:0] blue,output[20:0] addr,output[3:0] TMDSd);
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
reg [10:0] CounterX=0, CounterY=0;
reg hSync=0, vSync=0;
reg[1:0]  DrawArea=0;
reg[20:0] addr_r;

assign addr=addr_r;

always (posedge clk_low) DrawArea <=(reset==1)? 0 :( (CounterX<h_pixel) && (CounterY<v_pixel));
always (posedge clk_low)begin
        if(reset==1)begin
                CounterX<=0;
                addr_r=0;
        end else begin
        if(CounterX==h_tot_pixel-1)begin
                CounterX <=0;
                //addr=CounterY*1280;
                end else begin
                CounterX <=CounterX+1;
                addr_r=(DrawArea==1)?addr_r+1:addr_r;
        end
        vSync <= ((CounterY>=v_pixel+v_front_porch) && (CounterY<v_tot_pixel-v_back_porch)&&
                (CounterX<h_tot_pixel-2))||((CounterY==v_pixel+v_front_porch-1)&&
                        (CounterX>=h_tot_pixel-2));
        if(CounterX==h_tot_pixel-1)begin
                CounterY <= (CounterY==v_tot_pixel-1) ? 0 : CounterY+1;
                addr_r = (CounterY==v_tot_pixel-1) ? 0 : addr_r;
        end
end

always (posedge clk_low) hSync <=(reset==1)? 0 : ((CounterX>=h_pixel+h_front_porch)
                 && (CounterX<h_tot_pixel-h_back_porch));
////////////////////////////////////////////////////////////////////////
wire [9:0] TMDS_red, TMDS_green, TMDS_blue;
wire TMDS_r,TMDS_b,TMDS_g;
////////////////////////////////////////////////////////////////////////
reg [3:0] TMDS_mod10;  // modulus 10 counter
reg [9:0] TMDS_shift_red, TMDS_shift_green, TMDS_shift_blue;
reg TMDS_shift_load;
always (posedge clk_high) TMDS_shift_load <= (reset==1)? 0 : (TMDS_mod10==4'd4);
always (posedge clk_high)
        begin
        if(reset==1) begin
                TMDS_shift_red<=0;
                TMDS_shift_green<=0;
                TMDS_shift_blue<=0;
                TMDS_mod10<=0;
        end else begin
                TMDS_shift_red   <= TMDS_shift_load ? TMDS_red   : TMDS_shift_red   [9:2];
                TMDS_shift_green <= TMDS_shift_load ? TMDS_green : TMDS_shift_green[9:2];
                TMDS_shift_blue  <= TMDS_shift_load ? TMDS_blue  : TMDS_shift_blue [9:2];
                TMDS_mod10 <= (TMDS_mod10==4'd4) ? 4'd0 : TMDS_mod10+4'd1;
        end
end

ODDRX1F ddr_r(.D0(TMDS_shift_red[0]),.D1(TMDS_shift_red[1]),.SCLK(clk_high),.Q(TMDS_r));
ODDRX1F ddr_g(.D0(TMDS_shift_green[0]),.D1(TMDS_shift_green[1]),.SCLK(clk_high),.Q(TMDS_g));
ODDRX1F ddr_b(.D0(TMDS_shift_blue[0]),.D1(TMDS_shift_blue[1]),.SCLK(clk_high),.Q(TMDS_b));


assign TMDSd[2]=TMDS_r;
assign TMDSd[1]=TMDS_g;
assign TMDSd[0]=TMDS_b;
```

```
78      assign TMDSd[3]=clk_low;
79
80
81      TMDS_Encoder encoder0 (.clklow(clk_low),.reset(reset),.state(DrawArea),.pix_data(blue)
82                      ,.H_VSync_Ctr({vSync,hSync}),.q_out(TMDS_blue));
83      TMDS_Encoder encoder1 (.clklow(clk_low),.reset(reset),.state(DrawArea)
84                      ,.pix_data(green),.H_VSync_Ctr(2'b0),.q_out(TMDS_green));
85      TMDS_Encoder encoder2 (.clklow(clk_low),.reset(reset),.state(DrawArea)
86                      ,.pix_data(red),.H_VSync_Ctr(2'b0),.q_out(TMDS_red));
87
```

```
1       module Cam_I2C(output valid,input clk400kHz,reset,send_data,r_w,input[7:0] datain,input[15:0]
    register_in
2                       ,input[6:0] slave_addr,input ackn,inout scl,inout  sda, output ready);
3       localparam reg[7:0] idle=0;
4       localparam reg[7:0] start=1;
5       localparam reg[7:0] send=2;
6       localparam reg[7:0] reg0=3;
7       localparam reg[7:0] reg1=4;
8       localparam reg[7:0] data=5;
9       localparam reg[7:0] stop=6;
10
11      reg valid_r;
12      reg[7:0] state=idle;
13      reg send_data_old=0;
14      reg rising_edge=0;
15      integer counter =0;
16      reg sda0=1;
17      assign sda=sda0;
18      reg sending=0;
19      reg[36:0] i2cdata={slave_addr,1'b0,1'b1,register_in[15:8],1'b1,register_in[7:0],1'b1,datain,1'b1,1'b0};
20      reg [36:0] i2cin=0;
21      reg ready0;
22      assign valid=valid_r;
23      always (posedge clk400kHz) begin
24              if (reset==1) begin
25                      state<=0;
26                      sda0<=1;
27                      counter=0;
28                      valid_r=0;
29              end else begin
30                      //sda=1;
31                      send_data_old<=send_data;
32                      rising_edge=(send_data==1&&send_data_old==0)?1:0;
33                      case (state)
34                              idle: begin
35                                      valid_r=0;
36                                      ready0<=1;
37                                      sda0<=1;
38                                      sending=0;
39                                      if(rising_edge==1) begin
40                                              state<=start;
41                                              ready0<=0;
42                                              counter=0;
43                                              sda0<=0;
44                                              //sending=1;
45                                      end
46                              end
47                              start: begin
48                                      sending=(counter>=36)?0:1;
49                                      sda0<=(counter>=36)?0:i2cdata[36-counter];

50                                      counter=(counter>=36)?0:counter+1;
51                                      state<=(counter>=36)?stop:start;

52                              end
53                              stop: begin
54                                      state<=idle;
55                                      sda0<=0;
56                                      if(i2cin[1]==0&&i2cin[10]==0&&i2cin[19]==0&&i2cin[28]==0)begin
57                                              valid_r=1;
58                                      end
59                              end
60                              default: begin
61                              end
62                      endcase
63              end
64      end
65      reg [7:0]clkcount=0;
66      reg clkdelay0;
67      reg clkdelay1;
68      reg scl0,scl1,scl2;
69      assign scl=sending?~clk400kHz:1;
70      assign ready=ready0;
71
```

```
1               module Cam_Init(input clk400,input reset,input init,inout sda,scl,output cam_ready);
2               reg send_data,cam_ready_r,ready,ready_old,cam_ready0;
3               reg[7:0] datain,slave_addr,state;
4               reg[15:0] register_in;
```

```
5               reg initia=1;
6               reg init_old=0;
7               integer counter;
8
9
10              Cam_I2C cam0 (.clk400kHz(clk400),.scl(scl),.sda(sda),.reset(reset),.send_data(send_data)
11                      ,.datain(datain),.register_in(register_in),.slave_addr(slave_addr)
12                      ,.ackn(1'b0),.ready(ready));
13              localparam reg[7:0] idle=0;
14              localparam reg[7:0] init_s=1;
15              localparam reg[7:0] wakeup=2;
16
17              assign cam_ready=cam_ready0;
18              integer dataint=7;
19
20              reg[23:0] data_init [0:61];
21              initial $readmemh("initdata_lowres.mem",data_init);
22              always (posedge clk400) begin
23                      if(reset==1) begin
24                              state<=idle;
25                              send_data<=0;
26                              cam_ready0<=0;
27                              init_old<=0;
28                              initia<=1;
29                      end else begin
30                              init_old<=init;
31                              ready_old<=ready;
32                              case (state)
33                                      idle: begin
34                                              state<=(init_old==0&&init==1)?init_s:idle;
35                                              counter<=0;
36                                      end
37                                      init_s:begin
38                                              send_data<=0;
39                                              if((ready==1&&ready_old==0)||initia)begin
40                                                      initia<=0;
41                                                      counter<=counter+1;

42                                                      send_data<=1;
43                                                      slave_addr<=16;
44                                                      //datain<='hff;
45                                                      //register_in<='hffff;
46                                                      datain<=data_init[counter][7:0];
47                                                      register_in<=data_init[counter][23:8];
48                                              end
49                                              state<=(counter>60)?idle:init_s;

50                                      end
51                                      wakeup:begin
52                                              send_data<=0;
53                                              if(counter>60) begin
54                                                      state<=idle;
55                                                      send_data<=1;
56                                                      slave_addr<=16;
57                                                      counter<=0;
58                                                      datain<=data_init[59];
59                                                      register_in<=data_init[59];
60                                              end;
61                                              counter<=counter+1;
62                                      end
63                                      default: begin
64                                      end
65                              endcase
66                      end
67              end
68              endmodule
69
```

```
1       module MIPI_Reciever
2       #(parameter
3       mipi_frec=216,
4       iddr_ratio=4
5       )
6       (input sys_clk,reset,lane0_d,mipi_clk,lane1_d,inout lane0_p,lane0_n,lane1_p,lane1_n
7       ,output[31:0] data_o,output[31:0] address_out,output ram_clk
8       ,output reg debug0,debug1,debug3,debug2,output termination,rec_data_o);
9
10      wire stop_clk, rec_data;
11      wire[7:0] lane0byte,lane1byte;
12      SoTFSM #(.mipi_frec(mipi_frec)) RxFSM (.rec_data(rec_data),.clk50MHz(sys_clk),.reset(reset)
13              ,.lane0_p(lane0_p),.lane0_n(lane0_n),.lane1_p(lane1_p),.lane1_n(lane1_n),.stop_rx(stop_clk)
14              ,.term(termination),.debug0(debug0));
15      wire[1:0]q_o_0,q_o_1;
16      wire sync;
17      wire mipi_clk_2,mipi_clk_4,mipi_clk_8;
18
19      CLKDIVF div2 (.CLKI(mipi_clk),.RST(reset),.CDIVX(mipi_clk_2));
20      CLKDIVF div4 (.CLKI(mipi_clk_2),.RST(reset),.CDIVX(mipi_clk_4));
21      CLKDIVF div8 (.CLKI(mipi_clk_4),.RST(reset),.CDIVX(mipi_clk_8));
22
```

```
23          IDDR1 lane0 (.lane(lane0_d),.mipi_clk(mipi_clk),.reset(reset),.stop(stop_clk)
24                  ,.sync(sync),.q_o(q_o_0));
25          IDDR1 lane1 (.lane(lane1_d),.mipi_clk(mipi_clk),.reset(reset),.stop(stop_clk)
26                  ,/*.sync(sync),*/.q_o(q_o_1));
27
28          wire[7:0] byte_o_0,byte_o_1;
29          Byte_Arranger BA0 (.reset(reset),.stop(stop_clk),.mipi_clk(mipi_clk),.q_o(q_o_0),.byte_o(byte_o_0));
30          Byte_Arranger BA1 (.reset(reset),.stop(stop_clk),.mipi_clk(mipi_clk),.q_o(q_o_1),.byte_o(byte_o_1));

31
32          wire[7:0] byte_0,byte_1;
33          Byte_Alligner BAL0(.reset(reset),.stop(stop_clk),.mipi_clk(mipi_clk),.sync(sync)
34                  ,.byte_i(byte_o_0),.byte_o(byte_0));
35          Byte_Alligner BAL1(.reset(reset),.stop(stop_clk),.mipi_clk(mipi_clk),.sync(sync)
36                  ,.byte_i(byte_o_1),.byte_o(byte_1));
37
38          wire [31:0] data;
39          wire valid;
40          wire[5:0] type_w;
41          wire[15:0] wordcount;
42          DATA_Encoder DE (.mipi_clk_4(mipi_clk_4),.reset(reset),.stop(stop_clk),.sync(sync),.byte_in0(byte_0)
43                  ,.data(data),.type_o(type_w),.wordcount(wordcount),.byte_in1(byte_1),.valid(valid));
44
45          Protocol1 Prot (.debug(debug2),.debug1(debug3),.mipi_clk_8(mipi_clk_8),.stop(stop_clk),.reset(reset)
46                  ,.valid(valid),.type_i(type_w),.wordcount(wordcount),.data_o(data_o),.data(data)
47                  ,.rec_data(rec_data),.address_o(address_out));
48          assign rec_data_o=rec_data;
49          assign ram_clk=mipi_clk_8;
50
51          assign debug1=rec_data;
52          //assign debug2=rec_data;
53          //assign debug3=stop_clk;
54          endmodule
55
56
57
58          module IDDR1(input lane,stop,reset,mipi_clk,output sync,output[1:0] q_o);
59          IDDRX1F i0(.D(lane),.SCLK(mipi_clk),.Q0(ddr[0]),.Q1(ddr[1]),.RST('b0));
60          reg[7:0] byte_r;
61          reg sync_r=0;
62          reg[1:0] q_o_r;
63          assign sync=sync_r;
64          assign q_o=q_o_r;
65          wire[1:0] ddr;
66          always (posedge mipi_clk) begin
67                  if(stop==1||reset==1)begin
68                          byte_r<=0;
69                          sync_r<=0;
70                  end else begin
71                          byte_r<={ddr,byte_r[7:2]};
72                          sync_r<=(byte_r[7:0]==8'b10111000)?1:sync_r;
73                          q_o_r<=ddr;
74                  end
75          end
76          endmodule
77
78          module Byte_Arranger(input reset,stop,mipi_clk,input[1:0] q_o,output[7:0] byte_o);
79          reg[7:0] byte_r;
80          assign byte_o=byte_r;
81          always (posedge mipi_clk) begin
82                  if(reset||stop)begin
83                          byte_r<=0;
84                  end else begin
85                          byte_r<={q_o,byte_r[7:2]};
86                  end
87          end
88          endmodule
89
90          module Byte_Alligner(input reset,stop,mipi_clk,sync,input[7:0] byte_i,output[7:0] byte_o);
91          reg[7:0] byte_o_r;
92          assign byte_o=byte_o_r; //////////////////////////SYNC
93          reg[7:0]counter;
94          always (posedge mipi_clk) begin
95                  if(reset||stop)begin
96                          byte_o_r<=0;
97                          counter<=0;
98                  end else begin
99                          if(sync)begin
100                                 counter<=(counter>=4)?1:counter+1;
101                                 byte_o_r<=(counter>=4)?byte_i:byte_o_r;
102                         end
103                 end
104         end
105         endmodule
106
107
108         module DATA_Encoder(input mipi_clk_4,reset,stop,sync,input[7:0] byte_in0,byte_in1,output[31:0]data
109                 ,output valid,output[5:0] type_o,output[15:0] wordcount);
110         reg[31:0] out_r,out_r_old;
111         reg valid_r,start;
```

```
112         assign valid=valid_r&&(!stop);
113         reg[31:0] counter;
114         wire[31:0] regheader;
115         assign regheader=out_r;
116         wire[7:0] ecc;
117         reg [31:0] data_r;
118         reg[5:0] type_o_r;
119         reg[15:0] wordcount_r;
120         assign data=data_r;
121         assign type_o=type_o_r;
122         assign wordcount=wordcount_r;
123         assign ecc[0]=regheader[0]^regheader[1]^regheader[2]^regheader[4]^regheader[5]^regheader[7]
124                 ^regheader[10]^regheader[11]^regheader[13]^regheader[16]s
125                 ^regheader[20]^regheader[21]^regheader[22]^regheader[23];
126         assign ecc[1]=regheader[0]^regheader[1]^regheader[3]^regheader[4]^regheader[6]^regheader[8]
127                 ^regheader[10]^regheader[12]^regheader[14]^regheader[17]
128                 ^regheader[20]^regheader[21]^regheader[22]^regheader[23];
129         assign ecc[2]=regheader[0]^regheader[2]^regheader[3]^regheader[5]^regheader[6]^regheader[9]
130                 ^regheader[11]^regheader[12]^regheader[15]^regheader[18]
131                 ^regheader[20]^regheader[21]^regheader[22];
132         assign ecc[3]=regheader[1]^regheader[2]^regheader[3]^regheader[7]^regheader[8]^regheader[9]
133                 ^regheader[13]^regheader[14]^regheader[15]^regheader[19]
134                 ^regheader[20]^regheader[21]^regheader[23];
135         assign ecc[4]=regheader[4]^regheader[5]^regheader[6]^regheader[7]^regheader[8]^regheader[9]
136                 ^regheader[16]^regheader[17]^regheader[18]^regheader[19]
137                 ^regheader[20]^regheader[22]^regheader[23];
138         assign ecc[5]=regheader[10]^regheader[11]^regheader[12]^regheader[13]^regheader[14]^regheader[15]
139                 ^regheader[16]^regheader[17]^regheader[18]^regheader[19]^regheader[21]^regheader[22]^regheader
    [23];
140         assign ecc[6]=0;
141         assign ecc[7]=0;
142         wire[7:0] syndrom;
143         assign syndrom=ecc^regheader[31:24];
144
145         wire[23:0] correction;
146         wire[31:0] regheader_correct;
147
148         assign correction[0]=syndrom==8'h07;
149         assign correction[1]=syndrom==8'h0B;
150         assign correction[2]=syndrom==8'h0D;
151         assign correction[3]=syndrom==8'h0E;
152         assign correction[4]=syndrom==8'h13;
153         assign correction[5]=syndrom==8'h15;
154         assign correction[4]=syndrom==8'h16;
155         assign correction[7]=syndrom==8'h19;
156         assign correction[8]=syndrom==8'h1A;
157         assign correction[9]=syndrom==8'h1C;
158         assign correction[10]=syndrom==8'h23;
159         assign correction[11]=syndrom==8'h25;
160         assign correction[12]=syndrom==8'h26;
161         assign correction[13]=syndrom==8'h29;
162         assign correction[14]=syndrom==8'h2A;
163         assign correction[15]=syndrom==8'h2C;
164         assign correction[16]=syndrom==8'h31;
165         assign correction[17]=syndrom==8'h32;
166         assign correction[18]=syndrom==8'h34;
167         assign correction[19]=syndrom==8'h38;
168         assign correction[20]=syndrom==8'h1F;
169         assign correction[21]=syndrom==8'h2F;
170         assign correction[22]=syndrom==8'h37;
171         assign correction[23]=syndrom==8'h3B;
172
173         assign regheader_correct=regheader^ {8'h00,correction};
174
175
176         always (posedge mipi_clk_4) begin
177                 if(reset||stop)begin
178                         out_r<=0;
179                         out_r_old<=0;
180                         valid_r<=0;
181                         start=0;
182                         counter<=0;
183                         data_r<=0;
184                         type_o_r<=0;
185                         wordcount_r<=0;
186                 end else begin
187                         if(sync)begin
188                                 out_r_old<=out_r;
189                                 out_r<={byte_in1,byte_in0,out_r[31:16]};
190                                 valid_r<=(ecc==regheader_correct[31:24]&&regheader_correct!=0)?1:valid_r;
191                                 start=(ecc==regheader_correct[31:24]&&regheader_correct!=0)?1:start;
192                                 type_o_r<=(ecc==regheader_correct[31:24]&&regheader_correct!=0)
193                                         ?regheader_correct[5:0]:type_o_r;
194                                 wordcount_r<=(ecc==regheader_correct[31:24]&&regheader_correct!=0)
195                                         ?regheader_correct[23:8]:wordcount_r;
196                                 if(start)begin
197                                         counter<=counter+1;
198                                         if(counter[0]==0&&counter[1]==1)begin
199                                                 counter<=1;
200                                                 data_r<=out_r;
```

```
201                                             end else begin
202                                             counter<=counter+1;
203                                     end
204                             end
205                     end
206             end
207     end
208     endmodule
209
210
211     module SoTFSM
212     #(parameter
213     mipi_frec=350
214     )
215     (input clk50MHz,reset,rec_data,lane0_p,lane0_n,lane1_p,lane1_n,stop_tran,
216     output stop_rx,term,debug0,debug1);
217     ////////////////States for long and short Packet Recieve
218     localparam reg[7:0] TIMEOUT=0;
219     localparam reg[7:0] LP11=1;
220     localparam reg[7:0] LP01=2;
221     localparam reg[7:0] LP00=3;
222     localparam reg[7:0] SYNC=4;
223     localparam reg[7:0] HEADER=5;
224     ////////////////Const for Timing
225     localparam integer Tlpx=2;//50ns -> nearest =40ns
226     localparam[31:0]   Timeout=(2000*50/mipi_frec);
227     localparam[31:0]   Tdterm=2+(2*50/mipi_frec);
228     localparam[31:0]   Thssettle=3+(3*50/mipi_frec);
229     ////////////////
230     reg[7:0] state_mipi=TIMEOUT;
231     reg stop_rx_r,term_r,debug0_r,debug1_r;
232     assign stop_rx=stop_rx_r;
233     assign term=term_r;
234     assign debug0=debug0_r;
235     assign debug1=debug1_r;
236
237     integer timer_tou,timer_term,timer_hs;
238     ////////////////////////////////////////////////FSM
239     always (posedge clk50MHz) begin
240             if(reset==1) begin
241                     state_mipi<=TIMEOUT;
242                     timer_tou<=0;
243                     timer_term<=0;
244                     timer_hs<=0;
245                     term_r<=0;
246                     stop_rx_r<=1;
247             end else begin
248                     case (state_mipi)
249                     TIMEOUT: begin
250                             state_mipi<=(lane0_p==1 && lane0_n==1 &&lane1_p==1 &&lane1_n==1)?LP11:TIMEOUT;
251                             timer_tou<=0;
252                             timer_term<=0;
253                             timer_hs<=0;
254                             term_r<=0;
255                             debug0_r<=0;
256                             stop_rx_r<=1;
257                     end
258                     LP11:begin
259                             state_mipi<=(lane0_p==0 && lane0_n==1 &&lane1_p==0 &&lane1_n==1)?LP01:LP11;
260                             debug0_r<=0;
261                     end
262                     LP01:begin
263                             if(timer_tou>=Timeout) begin
264                                     state_mipi<=TIMEOUT;
265                                     end else begin
266                             if(lane0_p==0 && lane0_n==0 &&lane1_p==0 &&lane1_n==0)begin
267                                     state_mipi<=LP00;
268                                     timer_tou<=0;
269                                     timer_hs<=0;
270                             end
271                             end
272                             if(timer_term>=Tdterm) begin
273                                     term_r<=1;
274                             end
275                             timer_tou<=timer_tou+1;
276                             timer_term<=timer_term+1;
277                             stop_rx_r<=1;
278                     end
279                     LP00:begin
280                             stop_rx_r<=1;
281                             if(timer_term>=Tdterm) begin
282                                     term_r<=1;
283                             end
284                             if(timer_hs>=Thssettle) begin
285                                     state_mipi<=SYNC;
286                                     timer_tou<=0;
287                                     stop_rx_r<=0;
288                             end
```

```
289                             if(timer_tou>=Timeout)begin
290                                     state_mipi<=TIMEOUT;
291                             end
292                             timer_tou<=timer_tou+1;
293                             timer_term<=timer_term+1;
294                             timer_hs<=timer_hs+1;
295                     end
296                     SYNC:begin
297                             debug0_r<=1;
298                             if(timer_tou>=Timeout)begin
299                                     state_mipi<=TIMEOUT;
300                                     stop_rx_r<=1;
301                             end
302                             timer_tou<=timer_tou+1;
303                             if(rec_data==1)begin
304                                     state_mipi<=HEADER;
305                                     timer_tou<=0;
306                             end
307                     end
308                     HEADER:begin
309                             timer_tou<=timer_tou+1;
310                             if(rec_data==0||timer_tou>Timeout)begin
311                                     state_mipi<=TIMEOUT;
312                                     term_r<=0;
313                                     //stop_rx_r<=1;
314                             end
315                     end
316                     default: begin
317                     end
318                     endcase
319             end
320     end
321     endmodule
322
323     module Protocol1(input mipi_clk_8,stop,reset,valid,input[5:0] type_i,input[15:0] wordcount
324             ,input[31:0] data,output[31:0] data_o,output[31:0]address_o,output rec_data
325             ,output reg debug,output reg debug1);
326     reg rec_data_r,state,valid_old;
327     reg[31:0] counter,count_val,data_o_r,counter_addr,cX_r,cY_r;
328     assign data_o=data_o_r;
329     assign rec_data=rec_data_r&&(!stop);
330     assign address_o=counter_addr;
331     reg[15:0] c='hffff;//crc code
332     wire[15:0] c_calk;
333     wire[31:0] d;//recieved data
334     assign d=data;
335     //////////////////////////////////////////////////////CRC Sum
336     assign c_calk[0]=d[21]^d[10]^c[10]^d[28]^d[6]^c[6]^d[24]^d[13]^c[13]^d[20]^d[5]
337             ^c[5]^d[12]^c[12]^d[4]^c[4]^d[0]^c[0];
338     assign c_calk[1]=d[22]^d[11]^c[11]^d[0]^c[0]^d[29]^d[7]^c[7]^d[25]^d[14]^c[14]
339             ^d[21]^d[6]^c[6]^d[13]^c[13]^d[5]^c[5]^d[1]^c[1];
340     assign c_calk[2]=d[23]^d[12]^c[12]^d[1]^c[1]^d[30]^d[8]^c[8]^d[26]^d[15]^c[15]
341             ^d[22]^d[7]^c[7]^d[14]^c[14]^d[6]^c[6]^d[2]^c[2];
342     assign c_calk[3]=d[24]^d[13]^c[13]^d[2]^c[2]^d[31]^d[9]^c[9]^d[27]^d[16]^d[23]
343             ^d[8]^c[8]^d[15]^c[15]^d[0]^c[0]^d[7]^c[7]^d[3]^c[3];
344     assign c_calk[4]=d[20]^d[16]^d[12]^c[12]^d[8]^c[8]^d[0]^c[0]^d[25]^d[14]^c[14]
345             ^d[3]^c[3]^d[21]^d[17]^d[6]^c[6]^d[13]^c[13]^d[9]^c[9]^d[5]^c[5]^d[1]^c[1];
346     assign c_calk[5]=d[21]^d[17]^d[13]^c[13]^d[9]^c[9]^d[1]^c[1]^d[26]^d[15]^c[15]
347             ^d[4]^c[4]^d[22]^d[0]^c[0]^d[18]^d[7]^c[7]^d[14]^c[14]^d[10]^c[10]^d[6]
348             ^c[6]^d[2]^c[2];
349     assign c_calk[6]=d[22]^d[18]^d[14]^c[14]^d[10]^c[10]^d[2]^c[2]^d[27]^d[16]^d[5]
350             ^c[5]^d[23]^d[1]^c[1]^d[19]^d[8]^c[8]^d[15]^c[15]^d[11]^c[11]^d[7]^c[7]
351             ^d[3]^c[3];
352     assign c_calk[7]=d[23]^d[19]^d[15]^c[15]^d[11]^c[11]^d[3]^c[3]^d[28]^d[17]^d[6]
353             ^c[6]^d[24]^d[2]^c[2]^d[20]^d[9]^c[9]^d[16]^d[12]^c[12]^d[8]^c[8]^d[4]
354             ^c[4]^d[0]^c[0];
355     assign c_calk[8]=d[24]^d[20]^d[16]^d[12]^c[12]^d[4]^c[4]^d[29]^d[18]^d[7]^c[7]
356             ^d[25]^d[3]^c[3]^d[21]^d[10]^c[10]^d[17]^d[13]^c[13]^d[9]^c[9]^d[5]^c[5]
357             ^d[1]^c[1];
358     assign c_calk[9]=d[25]^d[21]^d[17]^d[13]^c[13]^d[5]^c[5]^d[30]^d[19]^d[8]^c[8]
359             ^d[26]^d[4]^c[4]^d[22]^d[11]^c[11]^d[18]^d[14]^c[14]^d[10]^c[10]^d[6]
360             ^c[6]^d[2]^c[2];
361     assign c_calk[10]=d[26]^d[22]^d[18]^d[14]^c[14]^d[6]^c[6]^d[31]^d[20]^d[9]^c[9]
362             ^d[27]^d[5]^c[5]^d[23]^d[12]^c[12]^d[19]^d[15]^c[15]^d[11]^c[11]^d[0]
363             ^c[0]^d[7]^c[7]^d[3]^c[3];
364     assign c_calk[11]=d[27]^d[16]^d[5]^c[5]^d[23]^d[1]^c[1]^d[19]^d[8]^c[8]^d[15]
365             ^c[15]^d[0]^c[0]^d[7]^c[7];
366     assign c_calk[12]=d[28]^d[17]^d[6]^c[6]^d[24]^d[2]^c[2]^d[20]^d[9]^c[9]^d[16]
367             ^d[1]^c[1]^d[8]^c[8]^d[0]^c[0];
368     assign c_calk[13]=d[29]^d[18]^d[7]^c[7]^d[25]^d[3]^c[3]^d[21]^d[10]^c[10]^d[17]
369             ^d[2]^c[2]^d[9]^c[9]^d[1]^c[1];
370     assign c_calk[14]=d[30]^d[19]^d[8]^c[8]^d[26]^d[4]^c[4]^d[22]^d[11]^c[11]^d[18]
371             ^d[3]^c[3]^d[10]^c[10]^d[2]^c[2];
372     assign c_calk[15]=d[31]^d[20]^d[9]^c[9]^d[27]^d[5]^c[5]^d[23]^d[12]^c[12]^d[19]
373             ^d[4]^c[4]^d[11]^c[11]^d[3]^c[3];
374     ////////////////////////////////////////////
375     always (posedge mipi_clk_8) begin
376             if(reset)begin
377                     rec_data_r<=0;
378                     state<=0;
```

```
379                     data_o_r<=0;
380                     counter_addr<=0;
381                     valid_old<=0;
382                     cX_r<=0;
383                     cY_r<=0;
384                     debug<=0;
385                     //counter<=0;
386                     //count_val<=0;
387             end     else begin
388                     valid_old<=valid;
389                     case (state)
390                             0:begin
391                                     c<='hffff;
392                                     if(((valid==1&&valid_old==0)&&(type_i=='h00||type_i=='h01)))begin
393                                             counter_addr<=0;

394                                             debug<=1;
395                                     end
396                                     if((valid==1&&valid_old==0)&&type_i=='h2a&&wordcount=='h0280)begin
397                                             state<=1;
398                                             count_val<=160;
399                                             debug<=0;
400                                             debug1<=0;

401                                     end

402                             end
403                             1:begin
404                                     if(counter<count_val)begin

405                                             counter<=counter+1;
406                                             rec_data_r<=1;
407                                             counter_addr<=counter_addr+1;
408                                             data_o_r<=data;
409                                             c<=c_calk;
410                                     end else begin
411                                             rec_data_r<=0;
412                                             state<=0;
413                                             counter<=0;
414                                             if(c==data[15:0])begin
415                                                     debug1<=1;
416                                             end
417                                     end
418                             end
419                             default:begin
420                             end
421                     endcase
422             end
423     end
424     endmodule
425
```

```
1       module MIPI_Reciever
2       #(parameter
3       mipi_frec=456,
4       iddr_ratio=4
5       )
6       (input sys_clk,reset,lane0_d,mipi_clk,mipi_clk_8,lane1_d,inout lane0_p,lane0_n,
7               lane1_p,lane1_n,output[31:0] data_o,output[31:0] adress_out,output ram_clk,
8               output reg debug0,debug1,debug3,debug2,output termination,rec_data_o,output[31:0] cX,cY);
9
10      wire stop_clk, rec_data;
11      wire[7:0] lane0byte,lane1byte;
12      SoTFSM #(.mipi_frec(mipi_frec)) RxFSM (.rec_data(rec_data),.clk100MHz(sys_clk),.reset(reset),
13              .lane0_p(lane0_p),.lane0_n(lane0_n),.lane1_p(lane1_p),.lane1_n(lane1_n),.stop_rx(stop_clk)
14              ,.term(termination),.debug0(debug0));
15      wire[3:0]q_o_0,q_o_1;
16      wire[1:0] ov_fl_0,ov_fl_1;
17      wire even,sync;
18      //assign debug1=even;
19      wire sync_mipi_clk,sync_mipi_clk_2,sync_mipi_clk_4,sync_mipi_clk_8;
20      //assign debug2=termination_r;
21
22
23      ECLKSYNCB SYNC(.ECLKI(mipi_clk),.STOP(stop_clk),.ECLKO(sync_mipi_clk));
24      CLKDIVF div2 (.CLKI(sync_mipi_clk),.RST(reset),.CDIVX(sync_mipi_clk_2));
25      CLKDIVF div4 (.CLKI(sync_mipi_clk_2),.RST(reset),.CDIVX(sync_mipi_clk_4));
26      CLKDIVF div8 (.CLKI(sync_mipi_clk_4),.RST(reset),.CDIVX(sync_mipi_clk_8));
27
28      /*wire mipi_clk_2,mipi_clk_4,mipi_clk_8;
29      CLKDIVF div21 (.CLKI(mipi_clk),.RST(reset),.CDIVX(mipi_clk_2));
30      CLKDIVF div41 (.CLKI(mipi_clk_2),.RST(reset),.CDIVX(mipi_clk_4));
31      CLKDIVF div81 (.CLKI(mipi_clk_4),.RST(reset),.CDIVX(mipi_clk_8));
32      */
33
34
35
36      IDDR2 lane0 (.lane(lane0_d),.sync_mipi_clk(sync_mipi_clk),.sync_mipi_clk_2(sync_mipi_clk_2)
37              ,.reset(reset),.stop(stop_clk),.even(even),.sync(sync),.q_o(q_o_0),.ov_fl(ov_fl_0));
38      IDDR2 lane1 (.lane(lane1_d),.sync_mipi_clk(sync_mipi_clk),.sync_mipi_clk_2(sync_mipi_clk_2)
```

```
39              ,.reset(reset),.stop(stop_clk),/*.even(even),.sync(sync),*/.q_o(q_o_1),.ov_fl(ov_fl_1));
40
41      wire[7:0] byte_e_0,byte_ue_0,byte_e_1,byte_ue_1;
42      Byte_Arrange BA0 (.reset(reset),.stop(stop_clk),.mipi_clk_2(sync_mipi_clk_2),.q_o(q_o_0)
43              ,.ov_fl(ov_fl_0),.byte_e(byte_e_0),.byte_ue(byte_ue_0));
44      Byte_Arrange BA1 (.reset(reset),.stop(stop_clk),.mipi_clk_2(sync_mipi_clk_2),.q_o(q_o_1)
45              ,.ov_fl(ov_fl_1),.byte_e(byte_e_1),.byte_ue(byte_ue_1));
46
47
48      wire[7:0] byte_o_0,byte_o_1;
49      Byte_Alligner BAL0(.reset(reset),.stop(stop_clk),.mipi_clk_2(sync_mipi_clk_2),.sync(sync)
50              ,.even(even),.byte_e(byte_e_0),.byte_ue(byte_ue_0),.byte_o(byte_o_0));
51      Byte_Alligner BAL1(.reset(reset),.stop(stop_clk),.mipi_clk_2(sync_mipi_clk_2),.sync(sync)
52              ,.even(even),.byte_e(byte_e_1),.byte_ue(byte_ue_1),.byte_o(byte_o_1));
53
54
55      wire [31:0] data;
56      wire valid;
57      wire[5:0] type_w;
58      wire[15:0] wordcount;
59      DATA_Encoder DE (.even(even),.mipi_clk_4(sync_mipi_clk_4),.reset(reset),.stop(stop_clk),.sync(sync)
60              ,.byte_in0(byte_o_0),.data(data),.type_o(type_w),.wordcount(wordcount)
61              ,.byte_in1(byte_o_1),.valid(valid));
62      Protocoll Prot (.debug(debug2),.debug1(debug3),.mipi_clk_8(sync_mipi_clk_8),.stop(stop_clk),.reset(
    reset)
63              ,.valid(valid),.type_i(type_w),.wordcount(wordcount),.data_o(data_o),.data(data)
64              ,.rec_data(rec_data),.adress_o(adress_out),.cX(cX),.cY(cY));
65      assign rec_data_o=rec_data;
66      assign debug1=rec_data;
67      //assign debug2=rec_data;
68      //assign debug3=stop_clk;
69      assign ram_clk=sync_mipi_clk_8;
70      endmodule
71
72      module IDDR2 (input lane,sync_mipi_clk,sync_mipi_clk_2,input reset,stop,output[3:0] q_o
73              ,output[1:0] ov_fl,output even,output sync);
74      wire[3:0] ddr;
75      reg sync_r,even_r;
76      reg[3:0] q_o_r;
77      reg[1:0] ov_fl_r;
78      reg[1:0] ov_fl_r1;
79      reg[7:0] syncbyte=0;
80      assign sync=sync_r&(!stop);
81      assign even=even_r;
82      assign ov_fl=ov_fl_r;
83      assign q_o=q_o_r;
84      wire[7:0] detect_e,detect_ue;
85      assign detect_e=syncbyte^8'b10111000;
86      assign detect_ue=((8'b00111111)&syncbyte)^8'b00101110;
87      wire delay_lane;
88      //DELAYG#(.DEL_MODE("ECLK_CENTERED")) delay(.A(lane),.Z(delay_lane));
89      IDDRX2F IDDR (.D(lane),.ECLK(sync_mipi_clk),.SCLK(sync_mipi_clk_2),.RST(reset||stop)
90              ,.Q0(ddr[0]),.Q1(ddr[1]),.Q2(ddr[2]),.Q3(ddr[3]));
91      always (posedge sync_mipi_clk_2) begin
92              if(reset||stop)begin
93                      sync_r<=0;
94                      even_r<=0;
95                      ov_fl_r<=0;
96                      q_o_r<=0;
97                      syncbyte=0;
98              end else begin
99                      syncbyte={ddr,syncbyte[7:4]};
100                     sync_r<=(detect_e==0||detect_ue==0)?1:sync_r;
101                     if(detect_e==0&&sync_r==0)begin
102                             even_r<=1;
103                     end
104                     if (detect_ue==0&&sync_r==0) begin
105                             even_r<=0;
106                     end
107                             q_o_r<=ddr;
108                             ov_fl_r<=q_o_r[3:2];
109             end
110     end
111     endmodule
112
113
114     module Byte_Arrange(input reset,stop,mipi_clk_2,input[3:0] q_o,input[1:0] ov_fl
115             ,output[7:0] byte_e,output[7:0]byte_ue);
116     reg[7:0] byte0_r,byte1_r;
117     assign byte_e=byte0_r;
118     assign byte_ue=byte1_r;
119     always (posedge mipi_clk_2) begin
120             if(reset||stop)begin
121                     byte0_r<=0;
122                     byte1_r<=0;
123             end else begin
124                     byte0_r<={q_o,byte0_r[7:4]};
125                     byte1_r<={q_o[1:0],ov_fl,byte1_r[7:4]};
126             end
127     end
```

```
128         endmodule
129
130         module Byte_Alligner(input reset,stop,mipi_clk_2,sync,even,input[7:0] byte_e
131                 ,input[7:0] byte_ue,output[7:0] byte_o);
132         reg[7:0] byte_o_r;
133         assign byte_o=byte_o_r; //////////////////////SYNC
134         reg[7:0]counter;
135         reg[7:0] byte_o_r_old;
136         reg[15:0] byte_o_r_s;
137         wire[7:0] byte_o_eu;
138         assign byte_o_eu=(even)?byte_e:byte_ue;
139         always (posedge mipi_clk_2) begin
140                 if(reset||stop)begin
141                         byte_o_r<=0;
142                         byte_o_r_old<=8'b10111000;
143                         counter<=0;
144                         byte_o_r_s<=0;
145                 end else begin
146                         if(sync)begin
147                                 counter<=(counter>=1)?0:counter+1;
148                                 `ifdef VERILATOR
149                                         byte_o_r<=(counter[0]==1)?byte_o_eu:byte_o_r;
150                                 `else
151                                         byte_o_r<=(counter[0]==0)?byte_o_eu:byte_o_r;
152                                 `endif
153                         end
154                 end
155         end
156         endmodule
157
158
```

```
1                 module ulx3s(input pixclk,inout cam0_sda,inout cam0_scl,debug0,debug1,debug2,debug3
2                         ,input reset,btn,input fire ,input cam0_clk,inout cam0_d0,cam0_d1,cam0_d0_r_p
3                         ,cam0_d0_r_n,cam0_d1_r_p,cam0_d1_r_n,cam0_clk_r_p,cam0_clk_r_n
4                         ,output[7:0] led,output[3:0] TMDSd,output ftdi_rxd
5                         ,output ftdi_txden);
6
7                 wire clk400;
8                 wire clk100Mhz;
9                 wire clk250;
10                wire cam0_sda_w,cam0_scl_w;
11                wire term;
12                wire mipi_clk05;
13                assign cam0_sda=cam0_sda_w;
14                assign cam0_scl=cam0_scl_w;
15                //Terminierung von cam0
16                assign cam0_d0_r_p=(term)?0:'bz;
17                assign cam0_d0_r_n=(term)?0:'bz;
18                assign cam0_d1_r_p=(term)?0:'bz;
19                assign cam0_d1_r_n=(term)?0:'bz;
20                assign cam0_clk_r_p=1?0:'bz;
21                assign cam0_clk_r_n=1?0:'bz;
22                //
23                Cam_Init i2c (.clk400(clk400),.reset(reset),.init(fire),.sda(cam0_sda_w),.scl(cam0_scl_w));

24                clock2 pll2(.clkin_25MHz(pixclk),.clk_400kHz(clk400));
25                clock8 pll3(.pixclk(pixclk),.clk_100MHz(clk100Mhz),.clk_250MHz(clk250));
26
27                wire[16:0] data_adress;
28                wire[31:0 ] data,cX,cY;
29                wire ram_clk,rec_data;
30                wire[20:0] read_addr;
31                wire[18:0] addr_write;
32                MIPI_Reciever mipi(.cX(cX),.cY(cY),.rec_data_o(rec_data),.sys_clk(clk100Mhz)
33                        ,.mipi_clk(cam0_clk),.reset(reset),.lane0_d(cam0_d0),.lane1_d(cam0_d1)
34                        ,.lane0_p(cam0_d0_r_p),.lane0_n(cam0_d0_r_n),.lane1_p(cam0_d1_r_p)
35                        ,.lane1_n(cam0_d1_r_n),.data_o(data),.adress_out(data_adress)
36                        ,.ram_clk(ram_clk),.debug0(debug0),.debug1(debug1),.debug2(debug2)
37                        ,.debug3(debug3),.termination(term));
38                assign led=0;
39                reg [7:0] color;
40                reg [7:0] red_v,green_v,blue_v;
41                reg[7:0] grey0,grey1,grey2,grey3;
42                wire[31:0] ramdata;
43                wire[7:0] hex;
44                reg[18:0] pixcount=0;
45
46                dpram_dualclock DPR(.data_a(data),.addr_a(data_adress),.addr_b(read_addr[18:2]),
47                        .we_a(btn),.we_b(0),.clk(ram_clk),.clk_b(pixclk),.data_out(ramdata));
48
49                wire[31:0] color_w;
50
51
52                reg[23:0] seraddr=0;
53                reg[23:0] counter_ser=0;
54
55                reg[7:0] serdata=78;
56                wire ready,start;
57                reg ready_old=0;
```

```
58                  reg[7:0] counter=0;
59
60
61                  always (posedge pixclk) begin
62                          if(counter>=3)begin
63                                  counter<=0;
64                                  color_w<=ramdata;
65                          end else begin
66                                  counter<=counter+1;
67                                  color_w<={8'h00,color_w[31:8]};
68                          end
69                          red_v<=color_w[7:0];
70                  end
71                  HDMI_Transciever HDMI(.clk_low(pixclk),.reset(reset),.clk_high(clk250)
72                           ,.red(red_v),.green(red_v),.blue(red_v),.addr(read_addr),.TMDSd(TMDSd));
73
74                  endmodule
75
76                  module clock
77                  (
78                  input mipi_clk,
79                  output mipi_clk_1_4,
80                  output mipi_clk_1_8,
81                  output clk_125MHz,
82                  output clk_150MHz,
83                  output locked
84                  );
85                  wire int_locked;
86
87                  (* ICP_CURRENT="9" *) (* LPF_RESISTOR="8" *) (* MFG_ENABLE_FILTEROPAMP="1" *)
88                   (* MFG_GMCREF_SEL="2" *)
89                  EHXPLLL
90                  #(
91                  .PLLRST_ENA("DISABLED"),
92                  .INTFB_WAKE("DISABLED"),
93                  .STDBY_ENABLE("DISABLED"),
94                  .DPHASE_SOURCE("DISABLED"),
95                  .CLKOS_FPHASE(0),
96                  .CLKOP_FPHASE(0),
97                  .CLKOS3_CPHASE(5),
98                  .CLKOS2_CPHASE(0),
99                  .CLKOS_CPHASE(1),
100                 .CLKOP_CPHASE(3),
101                 .OUTDIVIDER_MUXD("DIVD"),
102                 .OUTDIVIDER_MUXC("DIVC"),
103                 .OUTDIVIDER_MUXB("DIVB"),
104                 .OUTDIVIDER_MUXA("DIVA"),
105                 .CLKOS3_ENABLE("ENABLED"),
106                 .CLKOS2_ENABLE("ENABLED"),
107                 .CLKOS_ENABLE("ENABLED"),
108                 .CLKOP_ENABLE("ENABLED"),
109                 .CLKOS3_DIV(4),
110                 .CLKOS2_DIV(8),
111                 .CLKOS_DIV(4),
112                 .CLKOP_DIV(1),
113                 .CLKFB_DIV(1),
114                 .CLKI_DIV(1),
115                 .FEEDBK_PATH("CLKOP")
116                 )
117                 pll_i
118                 (
119                 .CLKI(mipi_clk),
120                 .CLKFB(clk_125MHz),
121                 .CLKOP(clk_125MHz),
122                 .CLKOS(mipi_clk_1_4),
123                 .CLKOS2(mipi_clk_1_8),
124                 .CLKOS3(clk_150MHz),
125                 .RST(1'b0),
126                 .STDBY(1'b0),
127                 .PHASESEL0(1'b0),
128                 .PHASESEL1(1'b0),
129                 .PHASEDIR(1'b0),
130                 .PHASESTEP(1'b0),
131                 .PLLWAKESYNC(1'b0),
132                 .ENCLKOP(1'b0),
133                 .ENCLKOS(1'b0),
134                 .ENCLKOS2(1'b0),
135                 .ENCLKOS3(1'b0),
136                 .LOCK(locked),
137                 .INTLOCK(int_locked)
138                 );
139                 endmodule
140
141                 module clock2
142                 (
143                 input clkin_25MHz,
144                 output clk_400kHz,
145                 output clk_200kHz,
146                 output clk_25MHz,
147                 output clk_150MHz,
```

```
148                 output locked
149                 );
150                 wire int_locked;
151
152                 (* ICP_CURRENT="9" *) (* LPF_RESISTOR="8" *) (* MFG_ENABLE_FILTEROPAMP="1" *)
153                         (* MFG_GMCREF_SEL="2" *)
154                 EHXPLLL
155                 #(
156                 .PLLRST_ENA("DISABLED"),
157                 .INTFB_WAKE("DISABLED"),
158                 .STDBY_ENABLE("DISABLED"),
159                 .DPHASE_SOURCE("DISABLED"),
160                 .CLKOS_FPHASE(0),
161                 .CLKOP_FPHASE(0),
162                 .CLKOS3_CPHASE(5),
163                 .CLKOS2_CPHASE(0),
164                 .CLKOS_CPHASE(1),
165                 .CLKOP_CPHASE(3),
166                 .OUTDIVIDER_MUXD("DIVD"),
167                 .OUTDIVIDER_MUXC("DIVC"),
168                 .OUTDIVIDER_MUXB("DIVB"),
169                 .OUTDIVIDER_MUXA("DIVA"),
170                 .CLKOS3_ENABLE("ENABLED"),
171                 .CLKOS2_ENABLE("ENABLED"),
172                 .CLKOS_ENABLE("ENABLED"),
173                 .CLKOP_ENABLE("ENABLED"),
174                 .CLKOS3_DIV(4),
175                 .CLKOS2_DIV(250),
176                 .CLKOS_DIV(125),
177                 .CLKOP_DIV(1),
178                 .CLKFB_DIV(2),
179                 .CLKI_DIV(1),
180                 .FEEDBK_PATH("CLKOP")
181                 )
182                 pll_i
183                 (
184                 .CLKI(clkin_25MHz),
185                 .CLKFB(clk_125MHz),
186                 .CLKOP(clk_125MHz),
187                 .CLKOS(clk_400kHz),
188                 .CLKOS2(clk200),
189                 .CLKOS3(clk_150MHz),
190                 .RST(1'b0),
191                 .STDBY(1'b0),
192                 .PHASESEL0(1'b0),
193                 .PHASESEL1(1'b0),
194                 .PHASEDIR(1'b0),
195                 .PHASESTEP(1'b0),
196                 .PLLWAKESYNC(1'b0),
197                 .ENCLKOP(1'b0),
198                 .ENCLKOS(1'b0),
199                 .ENCLKOS2(1'b0),
200                 .ENCLKOS3(1'b0),
201                 .LOCK(locked),
202                 .INTLOCK(int_locked)
203                 );
204                 endmodule
205
206                 module clock8
207                 (
208                 input pixclk,
209                 output byte_clk8,
210                 output clk_1_6Mhz,
211                 output clk_250MHz,
212                 output clk_100MHz,
213                 output locked
214                 );
215                 wire int_locked;
216
217                 (* ICP_CURRENT="9" *) (* LPF_RESISTOR="8" *) (* MFG_ENABLE_FILTEROPAMP="1" *)
218                         (* MFG_GMCREF_SEL="2" *)
219                 EHXPLLL
220                 #(
221                 .PLLRST_ENA("DISABLED"),
222                 .INTFB_WAKE("DISABLED"),
223                 .STDBY_ENABLE("DISABLED"),
224                 .DPHASE_SOURCE("DISABLED"),
225                 .CLKOS_FPHASE(0),
226                 .CLKOP_FPHASE(0),
227                 .CLKOS3_CPHASE(5),
228                 .CLKOS2_CPHASE(0),
229                 .CLKOS_CPHASE(1),
230                 .CLKOP_CPHASE(3),
231                 .OUTDIVIDER_MUXD("DIVD"),
232                 .OUTDIVIDER_MUXC("DIVC"),
233                 .OUTDIVIDER_MUXB("DIVB"),
234                 .OUTDIVIDER_MUXA("DIVA"),
235                 .CLKOS3_ENABLE("ENABLED"),
236                 .CLKOS2_ENABLE("ENABLED"),
237                 .CLKOS_ENABLE("ENABLED"),
```

```
238                 .CLKOP_ENABLE("ENABLED"),
239                 .CLKOS3_DIV(5),
240                 .CLKOS2_DIV(2),
241                 .CLKOS_DIV(128),
242                 .CLKOP_DIV(1),
243                 .CLKFB_DIV(10),
244                 .CLKI_DIV(1),
245                 .FEEDBK_PATH("CLKOP")
246                 )
247                 pll_i
248                 (
249                 .CLKI(pixclk),
250                 .CLKFB(clk_125MHz),
251                 .CLKOP(clk_125MHz),
252                 .CLKOS(byte_clk8),
253                 .CLKOS2(clk_250MHz),
254                 .CLKOS3(clk_100MHz),
255                 .RST(1'b0),
256                 .STDBY(1'b0),
257                 .PHASESEL0(1'b0),
258                 .PHASESEL1(1'b0),
259                 .PHASEDIR(1'b0),
260                 .PHASESTEP(1'b0),
261                 .PLLWAKESYNC(1'b0),
262                 .ENCLKOP(1'b0),
263                 .ENCLKOS(1'b0),
264                 .ENCLKOS2(1'b0),
265                 .ENCLKOS3(1'b0),
266                 .LOCK(locked),
267                 .INTLOCK(int_locked)
268                 );
269                 endmodule
270                 module dpram_dualclock
271                 (
272                 input [31:0] data_a, data_b,
273                 input [16:0] addr_a,input [16:0] addr_b,input[1:0]bank,
274                 input we_a, we_b, clk, clk_b,
275                 output reg [31:0] data_out
276                 );
277                 reg [31:0] ram[76799:0];
278                 initial $readmemh("testimage.mem",ram);
279
280                 // Port A
281                 always (posedge clk) begin
282                         if(we_a)begin
283                                 ram[addr_a] <= data_a;
284                         end
285                 end
286                 // Port B
287                 always(posedge clk_b) begin
288                         data_out<=ram[addr_b];
289                 end
290                 endmodule
291
292
```