Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Timo Wischer

Multicast-Video-Streaming mit vorheriger Binarisierung auf einem FPGA für Weiterverarbeitung in OpenCV

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Timo Wischer

Multicast-Video-Streaming mit vorheriger Binarisierung auf einem FPGA für Weiterverarbeitung in OpenCV

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Stephan Pareigis
Zweitgutachter: Prof. Dr.-Ing. Bernd Schwarz

Eingereicht am: 30. August 2013

Timo Wischer

Thema der Arbeit
Multicast-Video-Streaming mit vorheriger Binarisierung auf einem FPGA für Weiterverarbeitung in OpenCV

Stichworte
Binarisierung, Multicast, Video-Streaming, Ethernet, OpenCV, FPGA, VHDL, C++, UDP, Sun Raster Format

Kurzzusammenfassung
Diese Arbeit umfasst die Implementierung eines UDP/IP-Hardware-Stacks, mit dem ein Video-Stream über Ethernet versendet werden kann. Dabei besteht dieser Stream aus unkomprimierten Einzelbildern. Diese Bilder werden zur Verifikation dieser Arbeit von einem Simulator generiert. Die generierten Graustufen-Bilder werden vor dem Versand in Hardware binarisiert und erodiert. Somit ist das zu versendende Bild ein Schwarz-Weiß-Bild. Die entsprechende Empfänger-Seite ist in Software implementiert. Diese Software hat die Aufgabe den Stream zu empfangen und jedes Bild in ein für die OpenCV-Bibliothek verständliches Format zur Verfügung zu stellen. Der Hauptaugenmerk dieser Arbeit liegt dabei in der Binarisierung und Erodierung eines Live-Streams einer Kamera in Echtzeit und der Übertragung dieser Bilddaten zu einem PC. Dieser PC ist zusammen mit dem FPGA auf einem Modellfahrzeug montiert und kann dann einen Fahrspurerkennungs-Algorithmus ausführen, der dieses Fahrzeug steuert.

Timo Wischer

Title of the paper
Binarize a picture with multicast video streaming in a FPGA for using it with OpenCV

Keywords
Binarization, Multicast, Video-Streaming, Ethernet, OpenCV, FPGA, VHDL, C++, UDP, Sun Raster Format

Abstract
This thesis describes the implementation of an UDP/IP hardware stack for sending video streams via ethernet. The stream consists of uncompressed single pictures. The pictures will be generated by a simulator for the verification of this implementation. The generated image is a gray scale image. The image will then be binarized and eroded in hardware. So the transmitted image is monochrome. The receiving side is implemented in software. It makes the single pictures of the stream available for the OpenCV library. The reason for this work is to binarize a live stream from a camera in real time and make it available on a computer. The computer can then detect the lane and control a model car.

Inhaltsverzeichnis

1. Einführung **1**
- 1.1. Motivation . . . 1
- 1.2. Anforderung und Zielsetzung . . . 2
- 1.3. Struktur der Arbeit . . . 3

2. Voruntersuchungen **5**
- 2.1. Berechnung auf einer CPU . . . 5
  - 2.1.1. Binarisierung . . . 5
  - 2.1.2. Erosionsfilter . . . 7
- 2.2. Berechnung auf einer GPU . . . 8
  - 2.2.1. Architektur einer GPU . . . 8
  - 2.2.2. Datenaustausch zwischen CPU und GPU . . . 9
  - 2.2.3. CUDA und OpenCL . . . 11
- 2.3. Berechnung auf einem FPGA . . . 11
  - 2.3.1. Wahl des Übertragungsweges . . . 12
  - 2.3.2. Anforderungen an ein FPGA-Board . . . 13

3. Anbindung der Kamera **14**
- 3.1. Kamera Interface . . . 14
- 3.2. Konfiguration . . . 17

4. Bildverarbeitungskette **18**
- 4.1. Adaptive Binarisierung . . . 19
- 4.2. Erosionsfilter . . . 20

5. Bilddaten-Sender **23**
- 5.1. Deserialisierung . . . 23
- 5.2. Bilddaten-Puffer . . . 25
- 5.3. Bildgrenzen-Zähler . . . 27
- 5.4. Absende-Steuerung . . . 29
- 5.5. Bilddaten-Header . . . 30
- 5.6. UDP-Header . . . 32
- 5.7. IP-Header . . . 39
- 5.8. Ethernet-Header . . . 42
- 5.9. XPS Ethernet Lite MAC Wrapper . . . 43

6. Bilddaten-Empfänger **48**
6.1. Stream öffnen . . . 48
6.2. Synchronisation . . . 50
6.3. Einlesen der Daten . . . 52
6.4. Dekodierung . . . 55

7. Evaluation **57**
7.1. Bilddaten-Simulator . . . 57
7.2. Funktionsanalyse . . . 62
7.2.1. Binarisierung . . . 64
7.2.2. Erosionsfilter . . . 65
7.2.3. Verifikation mit verschiedenen Takten . . . 65
7.3. Performanceanalyse . . . 66
7.3.1. Bild-Dekodierung . . . 69
7.3.2. Latenz des Gesamtsystems . . . 70
7.3.3. CPU-Auslastung . . . 72
7.3.4. Wahl des Übertragungsweges . . . 72
7.4. Kamera-Anbindung . . . 74

8. Zusammenfassung **77**

9. Ausblick **80**
9.1. Adapter für VHDC-Buchse anfertigen . . . 80
9.2. Schnellere Kamera verwenden . . . 80
9.3. Projektiv transformiertes Bild übertragen . . . 81
9.4. Adaptive Binarisierung mit lokalen Schwellwerten . . . 81
9.5. Wiederherstellung der Paketreihenfolge . . . 81
9.6. Andere Kamera verwenden . . . 82

A. CD-Anhang **83**
A.1. FAUSTcore . . . 83
A.2. FAUSTplugins . . . 83
A.3. FPGA . . . 84

Abbildungsverzeichnis **86**

Tabellenverzeichnis **88**

Listings **89**

Literaturverzeichnis **90**

Onlinequellen **92**

1. Einführung

Bildverarbeitung erfordert oft sehr rechenintensive Verfahren. Ein Verfahren ist die Binarisierung von Bildern. Hierbei wird aus einem Farb- oder Graustufenbild ein Schwarz-Weiß-Bild berechnet. Da dabei jedes Pixel mindestens einmal betrachtet werden muss, ist diese Aufgabe nicht mit jeder Hardware zu bewältigen. [vgl. NFHS11b, S. 102] Erschwerend kommt hinzu, dass in vielen Anwendungen nicht nur ein einziges Bild binarisiert werden soll, sondern eher 30-70 Bilder pro Sekunde.

Dieses Performance-Problem existiert auch im FAUST-Projekt. Das FAUST-Projekt befasst sich mit dem autonomen Fahren von Modellfahrzeugen. Eine Disziplin ist das Folgen einer Fahrspur.

1.1. Motivation

Dieses Projekt stellt sich der Herausforderung, die Fahrspur anhand von Live-Bildern einer Kamera sicher zu erkennen. Die zur Zeit gewählte Variante binarisiert die Bilddaten nicht, ist aber auch nicht vollkommen zuverlässig. Die Fahrspur wird durch ein Polynom dritten Grades approximiert. Für diese Approximation werden Punkte, die einen bestimmten Helligkeitswert überschreiten, verwendet. [Jen08, S. 1] Hauptproblem ist dabei, dass auf Grund von ungleichmäßiger Ausleuchtung teilweise Punkte erkannt werden, die für dieses Polynom verwendet werden, aber nicht zur Fahrspurmarkierung gehören. Hierfür soll die vorherige Binarisierung der Bilddaten Abhilfe schaffen. Dabei wird ein Schwellwert verwendet, der die Grenze für die Umwandelung in schwarze und weiße Bildpunkte definiert. Der Vorteil dieses Schwellwerts ist, dass dieser für jedes Bild neu berechnet wird. Somit ist diese Binarisierung unempfindlich gegenüber unterschiedlichen Lichtverhältnissen.

Da diese Berechnung auf dem Fahrzeug stattfinden soll und auf diesem nicht unbegrenzt Platz und elektrische Energie zur Verfügung steht, kann kein leistungsstärkerer Rechner verwendet werden. Der zurzeit verbaute PC ist allein mit der Kamera-Anbindung über USB schon zu mehr als $\frac{1}{4}$ ausgelastet. Dies wird in Kapitel 7.3.3 genauer untersucht.

Die bisherige Lösung setzt, wie schon erwähnt, auf die Approximation der Fahrspur durch Polynome auf. Da eine vollständige Fahrspurmarkierung aus zwei Außenlinien und einer Mittellinie besteht, wird versucht diese drei Fahrspuren zu approximieren. Dies minimiert die Fehleranfälligkeit durch Fehlstellen in der Fahrspur. Diese Software wird auf dem PC ausgeführt. [Jen08, S. 2] Das Problem dieser Lösung ist die Empfindlichkeit gegenüber unregelmäßiger Ausleuchtung der Strecke.

Eine andere Variante nutzt die Hough-Transformation. Auf diesen Fahrzeugen ist nur ein FPGA verbaut, auf dem die komplette Fahrspurführung ausgeführt wird. Der Vorteil eines FPGAs liegt in der hohen Parallelisierung, die erreicht werden kann. Es können zum Beispiel mehrere Bildfilter auf ein neues Bild angewendet werden, währenddessen für das letzte Bild die Hough-Transformation durchgeführt wird. In dieser Lösung wird unter anderem eine adaptive Binarisierung vorher ausgeführt, um die Probleme mit den Helligkeitsunterschieden zu eliminieren. Bei dieser verwendeten Hough-Transformation wird jedoch nur eine Fahrspur analysiert. Erschwerend kommt hinzu, dass nur ein sehr kleiner Bildausschnitt für die Analyse verwendet wird und damit Verunreinigungen der Strecke zu Fehlverhalten führen können. [Kir12, S. 35]

1.2. Anforderung und Zielsetzung

Da beide Varianten Schwächen aufweisen, ist das Ziel dieser Arbeit die Vorteile beider zu kombinieren. Zum einen soll die fehlerunanfälligere Approximation der Fahrspur durch drei Polynome weiter verwendet werden, zum anderen sollen aber auch die Bildfilter der Hardware-Lösung integriert werden. Dabei liegt es nahe, für die Bildfilter einen FPGA zu verwenden und für die Approximation weiterhin auf einen PC zu setzen. Somit ergibt sich jedoch das Problem, wie die Daten vom FPGA zum PC übertragen werden sollen. Deshalb wurden mehrere Untersuchungen in Kapitel 2 unternommen, um die Komplexität sowie die Anzahl der Komponenten und der Kommunikationswege zu minimieren. Alle diese Faktoren erhöhen die Fehleranfälligkeit und den Wartungsaufwand der umgesetzten Lösung. Die betrachteten Komponenten sind dabei eine CPU, eine GPU und ein FPGA.

Im Ergebnis dieser Arbeit werden die gut parallelisierbaren Bildoperationen auf einem FPGA und die sehr rechenintensive Spurführung mit Hilfe von Polynomen auf dem PC ausgeführt. Die Kamera-Anbindung [Sal13], die adaptive Binarisierung und das Erosionsfilter [vgl. Kir12, S.

27-29] wird dabei aus schon vorhandenen Arbeiten übernommen. Um die Funktionsweise dieser Lösung testen und nachweisen zu können, wird anstelle einer echten Kamera ein Simulator verwendet. Das eingesetzte Verfahren zur Spurführung wird weiterverwendet. Hauptaugenmerk dieser Arbeit liegt somit auf der Bilddaten-Übertragung zwischen FPGA und PC. Dabei wurde sich, auf Grund der in Kapitel 2.3 genannten Punkte, für eine Ethernet-Schnittstelle entschieden. Für diese Schnittstelle wird ein UDP-IP-Stack in Hardware implementiert, um die Bilddaten mit möglichst geringen Latenzzeiten versenden zu können. Jedem Bild soll ein Bild-Header vorangestellt werden, so dass auf der Empfänger-Seite nicht zusätzlich noch Eigenschaften wie Auflösung, Farbtiefe und Komprimierungsverfahren konfiguriert werden müssen. Die Bilddaten sollen mit einer Farbtiefe von einem Bit unkomprimiert übertragen werden. Da die Spurführung auf OpenCV aufbaut, sollen die Bilddaten in einem für OpenCV verständliches Bildformat auf der Empfängerseite zur Verfügung stehen.

Bei dieser Implementierung entstehen mehrere Probleme, die zu lösen waren. Ein Problem besteht darin, dass die Kamera mit einem anderen Takt arbeitet als die Ethernet-Schnittstelle. Weiterhin enthält der IP-Header eine Prüfsumme, die berechnet werden muss. Diese Berechnung ist dabei von der variierenden IP-Paketlänge abhängig. Des Weiteren ist mit UDP die Paketreihenfolge nicht garantiert. Die Bilder sollen aber vollständig und in der richtigen Reihenfolge empfangen werden. Auf der Empfängerseite muss der Beginn eines neuen Bildes sicher erkannt werden, da der Empfänger irgendwann eingeschaltet werden könnte und sich dann synchronisieren können muss. Die empfangenen Bilddaten müssen möglichst schnell in ein OpenCV-verständliches Format gewandelt werden können. Insgesamt muss die Verzögerung zwischen der Bildaufnehme und dem zur Verfügung stellen auf dem PC ausreichend kurz sein.

1.3. Struktur der Arbeit

Im folgenden Kapitel werden mehrere Untersuchungen beschrieben, um möglichst die performanteste und kompakteste Lösung zu finden. Dabei wird auf die möglichen Wege mit nur einer CPU, mit der zusätzlichen Verwendung einer GPU und der Kombination aus CPU und FPGA eingegangen. Die Kommunikation zwischen den Komponenten erhält dabei eine besondere Begutachtung.

Unter Kapitel 3 werden Grundlagen der verwendeten Kamera dargestellt. Dabei wird auf eine mögliche Anschlussbelegung eingegangen. Wichtige Punkte, die bei der Belegung zu

beachten sind, werden ebenfalls erläutert. Im Kapitel 4 wird die Funktionsweise der adaptiven Binarisierung und des Erosionsfilters erklärt. In diesem Zusammenhang wird auch ein Überblick über das Gesamtsystem gegeben.

In Kapitel 5 werden die Komponenten des in Hardware implementierten Bilddaten-Senders vorgestellt. Da dieser Teil den Großteil dieser Arbeit ausmacht, wird in diesem Abschnitt auch auf viele Probleme und deren Lösungen eingegangen.

Das Kapitel 6 beschreibt den Bilddaten-Empfänger, der PC-seitig ausgeführt wird. Dabei wird auf Probleme, wie die Synchronisation und die Bilddaten-Dekodierung eingegangen.

In Kapitel 7 wird die implementierte Lösung auf ihre allgemeine Funktionsweise, sowie auf ihre Performance untersucht. Die notwendige geringe Verzögerung der gesamten Pipeline ist dabei einer der Hauptbetrachtungspunkte.

Eine Zusammenfassung dieser Arbeit ist in Kapitel 8 zu finden. Mögliche Erweiterungen in Bezug auf Funktionsumfang und Performance bei der Verarbeitung werden in Kapitel 9 gegeben.

2. Voruntersuchungen

Im Folgenden werden Ergebnisse der durchgeführten Voruntersuchungen erläutert. Diese Untersuchungen wurden unternommen, um eine ausreichend schnelle und kostengünstigste Lösung zu finden.

2.1. Berechnung auf einer CPU

Die erste Möglichkeit die untersucht wurde, war die Bildverarbeitung direkt auf der CPU ausführen zu lassen. Dabei ist zu beachten, dass bei beiden hier verwendeten Bildoperationen jedes Pixel mindestens einmal in die Berechnung einfließt und somit eine hohe Belastung der CPU entsteht.

2.1.1. Binarisierung

Grundsätzlich wird eine Bildbinarisierung verwendet, um zum Beispiel aus einem Graustufenbild ein Schwarz-Weiß-Bild zu berechnen. Dabei existieren Verfahren bei denen feste oder sich automatisch anpassende Schwellwerte verwendet werden. Der Schwellwert bestimmt bis wann ein Pixel im Ergebnisbild Schwarz sein soll und ab wann dieser Pixel in einen Weißen gewandelt werden soll. Die in dieser Arbeit verwendete Binarisierung nutzt einen variablen Schwellwert und wird deshalb auch als adaptive Binarisierung bezeichnet. [vgl. NFHS11b, S. 102]

Dieser variable Schwellwert wird aus dem Maximalwert des gesamten Bildes berechnet. Dabei wird nachfolgende Funktion 2.1 verwendet. Wobei g der 8-Bit-Wert des aktuellen Pixels ist und g_{max} der Maximalwert aller Pixel eines Bildes. [vgl. Kir12, S. 27]

$$f_{binarize}(g) = \begin{cases} 0 & g < (g_{max} * 0,85) \\ 1 & g \geq (g_{max} * 0,85) \end{cases} \tag{2.1}$$

Um Performance-Messungen auf einer CPU durchführen zu können, musste diese Lösung in Software implementiert werden. Die Softwarelösung enthält bereits Optimierungen. Anstatt den Maximalwert des gesamten Bildes zuerst zu berechnen und anschließend das Bild zu binarisieren, wird dieser Wert, währenddessen die Bildpunkte umgewandelt werden, bestimmt. Das bedeutet, dass der Maximalwert und somit auch der Schwellwert sich adaptiv während der Binarisierung eines Bildes ändert.

Die Implementierung dieses Algorithmuses ist in Listing 2.1 zu finden.

```
1  void UEyeDriver::binarize(cv::Mat& imageMat) const
2  {
3    unsigned char maxValue = 0;
4    unsigned char threshold = 0;
5    for(int y=0; y<imageMat.rows; y++)
6    {
7      for(int x=0; x<imageMat.cols; x++)
8      {
9        const unsigned char pixel = imageMat.at<unsigned char>(y,
           x);
10       if (pixel > threshold)
11       {
12         if (pixel > maxValue)
13         {
14           maxValue = pixel;
15           threshold = pixel * 0.85 + 0.5;
16         }
17
18         imageMat.at<unsigned char>(y, x) = 0xFF;
19       }
20       else
21       {
22         imageMat.at<unsigned char>(y, x) = 0x00;
23       }
24     }
25   }
26 }
```

Listing 2.1: Binarisierung in Software [JBW13]

Mit Zeile 5 und 7 wird über das Bild zeilen- und spaltenweise iteriert. Der Wert des aktuellen Bildpunktes wird in Zeile 9 in der Variablen "pixel" gespeichert. Wenn dieser Wert nun größer

als der Schwellwert “threshold” ist, wird der aktuelle Bildpunkt in Zeile 18 durch FF_{16} ersetzt. Falls der Wert des Bildpunktes auch größer als der Maximalwert “maxValue” ist, wird dieser in Zeile 14 durch den neuen Maximalwert ersetzt und in Zeile 15 der neue daraus resultierende Schwellwert berechnet. Der Maximalwert bezieht sich somit immer nur auf die bis jetzt verarbeiteten Bildpunkte. Falls der aktuelle Bildpunkt kleiner als der Schwellwert ist, wird dieser durch 00_{16} in Zeile 22 ersetzt.

Der Performance-Test mit dieser Implementierung wurde auf der auf dem Fahrzeug verbauten CPU durchgeführt. Diese CPU besitzt die in Tabelle 2.1 aufgelisteten Merkmale.

Plattform	Prozessor	Takt
Commell LP-172D	Intel® Atom CPU D2700	2,13 GHz

Tabelle 2.1.: Spezifikation der auf den CaroloCup-Fahrzeugen verbaute CPU [Com13]

Für die Messung wurden reale Bilddaten von der auf den CaroloCup-Fahrzeugen verbauten Kamera benutzt. Diese Kamera besitzt die in Tabelle 2.2 aufgeführten Merkmale.

Bezeichnung	Konfigurierte Auflösung	Konfigurierte FPS
IDS UI1221-LE-M-GL	752 x 480	40

Tabelle 2.2.: Spezifikation der auf den CaroloCup-Fahrzeugen verbaute Kamera [IDS10]

Mit der oben verwendeten Konfiguration benötigt die Softwarelösung für die Binarisierung im Mittel 8,94 ms für ein Bild. Dabei muss die CPU $752 * 480 = 360.960$ Bildpunkte verarbeiten. Da die Kamera mit 40 fps betrieben wird und jedes Bild in die Berechnung der Fahrspur einfließen soll, steht ein Zeitfenster von 25 ms zur Verfügung. Da jedoch zur Bestimmung des Lenkwinkels noch weitere Berechnungen wie eine projektive Transformation, Linsenverzeichnungskorrektur und die Interpolation der Fahrspur ausgeführt werden müssen, sind die fast 9 ms alleine für eine Binarisierung zu lange. [vgl. Jen08, S. 23]

2.1.2. Erosionsfilter

Das Erosionsfilter soll der Binarisierung nachgeschaltet werden. Es bewirkt, dass vereinzelte weiße Bildpunkte, die unter anderem durch rauschen entstehen können, entfernt werden. Im einfachsten Fall wird hierfür eine Faltungsmaske mit der Größe 3x3 Bildpunkte verwendet. Alle 9 Pixel werden durch UND-Operatoren miteinander verknüpft. Das Ergebnis dieser Operationen ist dann der Wert des Zielbildpunktes. [vgl. Erh08, S. 171]

Das bedeutet, dass im Vergleich zur Binarisierung nicht nur ein Quellbildpunkt für die Berechnung eines Zielbildpunktes einfließt. Sondern es müssen mindestens neun Quellbildpunkt für die Erzeugung eines Zielbildpunktes verwendet werden. Somit ist das Erosionsfilter circa neun mal komplexer und benötigt dementsprechend neun mal mehr Zeit. Dies ist wiederum ein weiteres KO-Kriterium, die Binarisierung und das Erosionsfilter auf der CPU zu berechnen.

2.2. Berechnung auf einer GPU

Da die auf den CaroloCup-Fahrzeugen verbaute CPU nicht performant genug ist, muss die Berechnung auf einer anderen Hardware ausgeführt werden. Eine weitere Möglichkeit wäre die Verwendung der Grafikkarte, die mit auf dem Commell-Board zu finden ist. Diese Grafikkarte hat die in Tabelle 2.3 aufgeführten Merkmale.

Bezeichnung	Takt	OpenGL
Intel® GMA 3650	640 MHz	3.0

Tabelle 2.3.: Spezifikation der auf den CaroloCup-Fahrzeugen verbauten Grafikkarte [Hin12a]

Da die Grafikkarte OpenGL 3.0 unterstützt, unterstützt sie auch die OpenGL Shader Language. Das bedeutet, dass die Möglichkeit besteht, Teile der Grafikkarte frei zu programmieren. [vgl. NFHS11a, S. 51] Somit ist es möglich die Grafikkarte mit Bilddaten zum Beispiel von einer Kamera zu versorgen und diese Daten dann von der Grafikkarte manipulieren zu lassen. [AJ12] Das Problem, dass sich jedoch dabei stellt, ist, wie die manipulierten Bilddaten wieder in den CPU-Speicher zurück kopiert werden können. Die CPU soll anschließend in der Lage sein, weitere Berechnungen, wie die projektive Transformation oder die eigentliche Spurführung, durchführen zu können. Um zu verstehen, welche Möglichkeiten es gibt, die Bilddaten wieder in den CPU-Speicher zu laden, ist das Verständnis über die Architektur einer GPU von großem Interesse.

2.2.1. Architektur einer GPU

Eine GPU besteht aus mehreren Pipeline-Stufen, die für unterschiedliche Aufgaben optimiert sind. Ab OpenGL 2.0 sind der Vertex-Shader und der Fragment-Shader programmierbar. Dabei verarbeitet der Vertex-Shader ganze Polygone und der Fragment-Shader arbeitet mit einzelnen Bildpunkten. In der Abbildung 2.1 sind die einzelnen Pipeline-Stufen mit ihren Kommunikationswegen untereinander dargestellt.

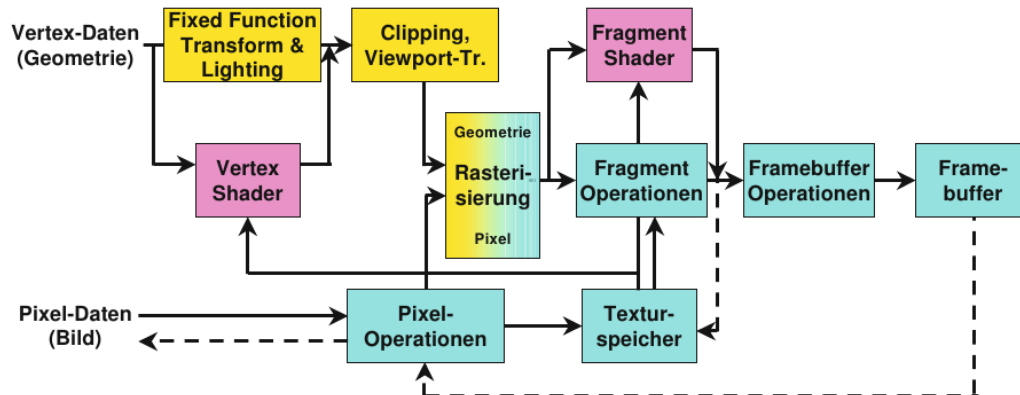

Abbildung 2.1.: Darstellung der Pipeline-Stufen einer Grafikkarte die OpenGL 2.0 unterstützt [vgl. NFHS11a, S. 51]

Das durch die einzelnen Schritte erzeugte Bild wird dann im Framebuffer abgelegt. Das im Framebuffer abgelegte Bild kann anschließend auf einem Monitor angezeigt werden. [vgl. NFHS11a, S. 46-54] Dieses manipulierte Bild aus dem Framebuffer soll jedoch nicht nur auf dem Monitor angezeigt werden, sondern auch von der CPU weiter verarbeitet werden können. Deshalb wurde ein Weg gesucht, den Framebuffer der GPU mit auf der CPU ausgeführten Software auszulesen.

2.2.2. Datenaustausch zwischen CPU und GPU

Um Bildinformationen zur Grafikkarte zu senden, kann eine Textur angelegt werden, die die Informationen eines Bildes enthält. Diese Textur steht dann im GPU-Speicher zur Verfügung. Der unter 2.2 dargestellte Beispiel-Code implementiert genau dieses Verhalten.

```
1  IplImage* frame = cvQueryFrame( capture );
2  destination = cvCreateImage ( cvSize(texWidth, texHeight),
3    frame->depth, frame->nChannels );
4  //sets size of destination- should contain captured frame
5  cvResize(frame, destination);
6  if(destination->nChannels == 3) {
7    glTexImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, GL_RGB,
8      texWidth, texHeight, 0, GL_BGR,
9      GL_UNSIGNED_BYTE, destination->imageData);
10 } else if(destination->nChannels == 4) {
11   glTexImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, GL_RGBA,
12     texWidth, texHeight, 0, GL_BGRA,
```

```
13        GL_UNSIGNED_BYTE, destination->imageData);
14    }
```

Listing 2.2: Erstellen einer Textur eines Kamerabildes [AJ12]

Mit Zeile 1 wird ein Bild von zum Beispiel einer Kamera aufgenommen. In Zeile 5 wird dieses Bild in den Speicher “destination” kopiert, wobei dieses auf die benötigte Größe skaliert wird. Anschließend wird abhängig von der Anzahl der Farbkanäle eines Bildpunktes in Zeile 7 oder 11 eine Textur aus den Daten des Bildes erstellt.

Damit ist das Problem, die Bilddaten an die Grafikkarte zu übertragen, gelöst. Weiterhin besteht jedoch das Problem, die veränderten Bilddaten aus dem Framebuffer der Grafikkarte wieder zurück zu kopieren. Dies kann mit dem OpenGL-Befehl glReadPixels durchgeführt werden. Dieser Funktion kann die Größe eines Bereiches in Bildpunkten übergeben werden. Zusätzlich wird ein Zeiger auf eine Speicheradresse übergeben. Bei dieser Speicheradresse werden die Bilddaten abgelegt und liegen somit nach Ausführung des OpenGL-Befehls im Speicherbereich der CPU. [SA06]

Um die Performance dieses Befehls zu überprüfen, wurde mit einem Testprogramm die Dauer des Kopiervorganges gemessen. Die Hardware, die für diese Messung verwendet wurde, hat die in Tabelle 2.4 gelisteten Eigenschaften.

Plattform	Prozessor	CPU-Takt	GPU-Takt
Lenovo ThinkPad L520	Intel® Core i5-2520M	2,50 GHz	650 MHz

Tabelle 2.4.: Für glReadPixels-Performance-Messung verwendete Hardware [Hin12b]

Um einen Mittelwert bilden zu können, wurde dieser Kopiervorgang mehrfach durchgeführt. Der dafür verwendete Sourcecode ist unter 2.3 dargestellt. Die dort genutzte Klasse mit dem Namen DiffTimeCalc, speichert beim Aufruf des Konstruktors in Zeile 3 einen Zeitstempel mit einer Genauigkeit von einer Mikrosekunde. Beim Aufruf der Methode getDiffTimeInUSec in Zeile 8 wird die Differenz zu einem zweiten Zeitstempel berechnet und zurück gegeben.

```
1   GLubyte *pixels = new GLubyte[640*480*4];
2
3   for (int i=0; i<50; i++)
4   {
5     const DiffTimeCalc diffTime;
6     glReadPixels(0, 0, 640,480, GL_RGBA, GL_UNSIGNED_BYTE,
        pixels);
```

```
7
8   std::cout << "Image␣reading␣needs␣" <<
        diffTime.getDiffTimeInUSec() << "␣us." << std::endl;
9 }
```

Listing 2.3: Auslesen des Framebuffers einer Grafikkarte [Wis13b]

Die Mittelwertbildung über die so ermittelten Zeiten ergab einen Wert von 4.602 µs. Dies ist, aus den schon bei der Binarisierung in Software genannten Gründen, eine zu große Zeitspanne, da unter anderem weiterhin die projektive Transformation und die Spurführung auf der CPU ausgeführt werden müssen. Erschwerend kommt hinzu, dass der Kopiervorgang beim ersten Aufruf immer über 10 ms benötigt hat und zwischendurch weitere Ausreißer mit bis zu 13 ms vorgekommen sind. Die für den Test verwendete Plattform, die in Tabelle 2.4 beschrieben ist, ist eine Andere, als die die auf den Fahrzeugen verbaut ist. Wenn diese aber mit den Eigenschaften aus Tabelle 2.1 und 2.3 verglichen wird, fällt auf, dass die für den Test verwendete Hardware performanter ist. Somit würde der Test auf dem Fahrzeug noch gravierender ausfallen.

2.2.3. CUDA und OpenCL

Es existieren noch weitere Möglichkeiten Grafikkarten zu programmieren. Diese bieten auch andere Möglichkeiten für den Datenaustausch zwischen Grafikkarte und CPU. CUDA ist dabei die Kurzform für "Compute Unified Device Architecture" und ist eine proprietäre Variante von NVIDIA. OpenCL ist ein offener Standard der vom Khronos Konsortiums entwickelt und gepflegt wird. [vgl. NFHS11a, S. 481] Die in Tabelle 2.3 beschriebene GPU, unterstützt keine dieser Standards. Die meisten Grafikkarten, die CUDA oder OpenCL unterstützen, haben eine sehr hohen Energiebedarf und sind somit für ein akkubetriebene Fahrzeug ungeeignet. [Hin12a]

2.3. Berechnung auf einem FPGA

Da die Berechnung auf der CPU und auf der GPU nicht schnell genug ausgeführt werden kann, viel die Wahl auf zusätzliche Hardware. Diese muss aber, wie schon erwähnt, energiesparend sein. So wurde sich für einen FPGA entschieden. Dieser eignet sich besonders gut für Bildverarbeitung, da oftmals nur wenige Bildpunkte für eine Berechnung benötigt werden und somit diese Operationen parallel oder in einer Pipeline angeordnet sein können. Das größte Problem stellt sich hierbei bei der Kommunikation zwischen PC und FPGA. Diese muss auf der einen Seite die großen Datenmengen der Bilder übertragen können. Auf der anderen Seite sollen

aber keine großen Latenzen entstehen, damit das aufgenommene Bild möglichst schnell für die Spurführung verwendet werden kann.

2.3.1. Wahl des Übertragungsweges

Um Abschätzen zu können, welche Schnittstellen für diese Problemstellung geeignet sind, ist die zu übertragende Datenmenge sehr von Relevanz. Da mit dieser Lösung die Auflösung der Kamera möglichst nicht schlechter werden soll, gehen wir von einer Auflösung von $752 * 480 = 360.960$ Bildpunkte aus. Diese Auflösung stellt die zur Zeit auf den Fahrzeugen verbaute Kamera aus Tabelle 2.2 zur Verfügung. Da nach der Binarisierung ein Pixel durch ein Bit dargestellt werden kann, müssen, ohne die Daten zu komprimieren, $\frac{360.960}{8} = 45.120$ Byte pro Bild übertragen werden. Diese Abschätzung wird ohne Berücksichtigung von Komprimierungsverfahren durchgeführt, da abhängig vom Bild und vom Verfahren die Größe variieren kann. So kann mit dieser Worst-Case-Abschätzung sichergestellt werden, dass das Bild auch wirklich übertragen werden kann. Die verwendete Kamera wird des Weiteren mit 40 Bilder pro Sekunde betrieben. Das bedeutet, dass auch 40 Bilder in einer Sekunde übertragen werden können sollen. Somit ergibt sich ein Datendurchsatz der Nutzdaten von $45.120 * 40 = 1.804.800$ Byte pro Sekunde.

Die serielle RS-232-Schnittstelle bietet eine maximale Übertragungsrate von 115.200 Bit pro Sekunde. Also ist sie für den benötigten Datendurchsatz zu langsam. Die parallele Schnittstelle, die unter IEEE 1284 spezifiziert ist, stellt der verwendete PC nicht zur Verfügung [Com13]. Mit Ethernet sind Übertragungsraten von 100 MBit/s möglich und mit dem Universal Serial Bus 2.0 bis zu 60 MB/s. [vgl. BT06, S. 473] Somit sind beide Bussysteme schnell genug. Der USB hat nur einen gravierenden Nachteil. Die Daten können nur durch Polling des Host-Controllers von den Geräten kopiert werden. Also müsste der PC, der in diesem Fall der Host-Controller ist, den FPGA ständig fragen, ob neue Daten zur Verfügung stehen. Das bedeutet zusätzlich, dass der FPGA einen Puffer für diese Daten haben müsste, was wieder herum die Latenz erhöhen würde. Mit Ethernet wird es dem FPGA ermöglicht, die Daten direkt weg zu schicken. Deshalb fällt die Wahl in dieser Arbeit auf die Ethernet-Schnittstelle. Als Übertragungsprotokoll wird das UDP/IP-Protokoll verwendet. Dieses Protokoll hat den Nachteil, dass nicht sichergestellt wird, ob das Paket auch wirklich ankommt. Es wurde aber als besser eingestuft, mal ein Bild nicht erfolgreich zu übertragen, als wiederholt zu versuchen ein veraltetes Bild zu versenden. Weiterhin minimiert diese Entscheidung die Komplexität der Sende-Hardware um einen großen Teil. Dies minimiert wieder herum die Latenz beim Versenden.

Ein in Hardware implementierter UDP/IP-Stack wird in dem Paper “An analysis of FPGA-based UDP/IP stack parallelism for embedded Ethernet connectivity” beschrieben. Dabei werden drei unterschiedlich komplexe Varianten gegenüber gestellt. Alle diese Varianten verwenden Dual-Port-RAMs als Sende- und Empfangsspeicher und stellen eine Schnittstelle für einen externen Physical Layer (PHY) Transceivers zur Verfügung. Die Unterschiede finden sich hauptsächlich in den unterstützten Protokollen wieder. Dabei handelt es sich um zusätzlich Protokolle wie ICMP, ARP, RARP und eine Schnittstelle für TCP-Verbindungen. Da aber alle Varianten RAM-Speicher für ein vollständiges Paket verwenden, muss dieses Paket auch vollständig im RAM liegen bevor es versendet werden kann. Mit einem FIFO-Speicher könnte das Versenden schon gestartet werden bevor alle Daten im Speicher liegen. Mit dieser Begründung wurde sich gegen diese Implementierung entschieden. Des Weiteren enthält auch die minimale Lösung einen Empfänger-Teil. Dieser wird aber unter Verwendung von festen IP- und MAC-Adressen oder Multicast-Kommunikation nicht benötigt. [LSH05] Weitere Untersuchungen zur Wiederverwendung kleinerer Komponenten sind in Kapitel 5.6 zu finden.

2.3.2. Anforderungen an ein FPGA-Board

Das verwendete FPGA-Board muss einige Anforderungen erfüllen. Zum Einen wird eine 100-MBit-Ethernet-Schnittstelle benötigt. Diese sollte mit Hilfe eines externen Physical Layer (PHY) Transceivers zur Verfügung gestellt werden, damit die in Kapitel 5.9 beschriebene Hardware verwendet werden kann. Des Weiteren werden IO-Pins, die mit hohen Taktraten betrieben werden können, benötigt, um die Kamera anbinden zu können. Die Größe des Boards spielt auch eine große Rolle, da nicht unbegrenzt Platz auf dem Fahrzeug zur Verfügung steht.

Anhand dieser Kriterien fiel die Entscheidung auf ein “Digilent™ Nexys 3 Board”. Mit dem verbauten SMSC LAN8710 PHY steht eine 100-MBit-Ethernet-Schnittstelle zur Verfügung. [vgl. Dig13, S. 11] Die VHDC-Buchse stellt IO-Ports zur Verfügung, die für hohe Frequenzen geeignet sind. Also kann auch die Kamera an dieses Board angeschlossen werden. [vgl. Dig13, S. 20] Ein anderes Board, das die Anforderungen erfüllt, ist das “Spartan 3E Starter Board”. Dieses stellt aber auch weitere nicht benötigte Komponenten zu Verfügung. Dazu gehören unter anderen ein LC-Display und eine zweite RS232-Schnittstelle. Dies hat einen größeren Stromverbrauch, größere Abmessungen und einen größeren Anschaffungspreis zur Folge. [Xil06] Das “Atlys™ Spartan-6 FPGA Development Board” würde den Anforderungen auch genügen. Nur auch dieses Board stellt weitere ungenutzte Komponenten zur Verfügung. Dazu zählen mehrere HDMI-Schnittstellen und AC-97-Audio-Ports. [Dig12]

3. Anbindung der Kamera

Die Anbindung der Kamera an den FPGA ist ein wichtige und nicht zu vernachlässigende Aufgabe. Das Anfertigen eines Adapter-Kabels ist zwar nicht Bestandteil dieser Bachelorarbeit, aber dennoch wird eine mögliche Anschlussbelegung vorgestellt. Schwierigkeiten bereitet dabei, eine Verbindung zwischen FPGA und Kamera herzustellen, die auch bei hohen Taktraten zuverlässig arbeitet.

3.1. Kamera Interface

Eine erste Idee wäre, die Kamera über drei Pmod-Ports anzuschließen. Diese Ports sind aber nur für geringe Frequenzen geeignet. Die Leiterbahnen für diese Ports wurden so verlegt, wie sie am besten gepasst haben. Das bedeutet, dass dabei weder auf Impedanzen noch auf gleiche Laufzeiten der nebeneinander liegenden Datenleitungen geachtet wurde. [vgl. Dig13, S. 21] Somit besteht die Gefahr, dass das Signal durch diese Einwirkungen unbrauchbar wird. Beim Planen des Verlaufs der Leiterbahnen der VHDC-Buchse wurden diese Probleme berücksichtigt. Deshalb sind mit diesem Port Taktraten von mehreren 100 MHz möglich. [vgl. Dig13, S. 20]

Eine mögliche Anschlussbelegung wird in Tabelle 3.1 beschrieben. Bei der Wahl dieser Belegung wurde auf mehrere Kriterien geachtet. Zum Einen wurde versucht, die einzelnen Leitungen möglichst weit voneinander zu trennen, um ein Übersprechen zu vermeiden. Dies soll das gegenseitige Beeinflussen der Datenleitungen minimieren. Des Weiteren wurde der System-Takt und der Bild-Takt der Kamera auf Takteingänge des FPGAs gelegt. Diese besonderen Pins garantieren einen minimalen Clock-Skew. Wenn ein Pin des FPGAs, der kein globaler Takteingang ist, mit einem Takt verbunden wird, kann dieser intern nicht als Takt für die Komponenten verwendet werden. [vgl. Xil13, S. 11-16] Ein eher nebensächliches aber nicht ganz unwichtiges Kriterium war die Reihenfolge der Leitungen. Es wurde dabei versucht, die gleiche Reihenfolge die schon der Postenstecker der Kamera vorgibt, beizubehalten. Dies minimiert Kreuzungen der Leitungen und vereinfacht somit die Herstellung dieses Adapters.

FPGA		Pfosten-Leiste der Kamera				FPGA	
PAD	VHDC	Nr.	Bezeichnung	Bezeichnung	Nr.	VHDC	PAD
B2	IO1-P	1	DOUT0	DOUT1	2	IO1-N	A2
D6	IO2-P	3	DOUT2	DOUT3	4	IO2-N	C6
B4	IO4-P	5	DOUT4	DOUT5	6	IO4-N	A4
B6	IO6-P	7	DOUT6	DOUT7	8	IO6-N	A6
D8	IO8-P	9	DOUT8	DOUT9	10	IO8-N	C8
D11	IO10-P	11	FRAME	LINE	12	IO10-N	C11
N/C	N/C	13	SYSCLK	PIX_CLK	14	IO11-P	C10
B11	IO13-P	15	STANDBY	RESET	16	IO13-N	A11
C13	IO15-P	17	SER_O+ (LVDS)	SER_O- (LVDS)	18	IO15-N	A13
F13	IO17-P	19	ERROR	LED_OUT	20	IO17-N	E13
D14	IO19-P	21	I2C_CLK	I2C_DATA	22	IO19-N	C14
N/C	GND	23	N/C	GND	24	GND	N/C

Tabelle 3.1.: Mögliche Pin-Belegung zwischen Kamera und FPGA [vgl. Sal13, S. 51]

Mit DOUT0 bis DOUT9 wird der Wert jedes Bildpunktes übertragen. Dabei hat dieser eine Auflösung von 10 Bit. Die Position des gerade übertragenden Bildpunktes wird mit den Leitungen FRAME und LINE bekannt gemacht. [vgl. Mic06, S. 12-13] Ein Bildpunktwert liegt mit jeder steigenden Flanke des Bildtaktes (PIXCLK) an. Dieses Verhalten ist in Abbildung 3.1 zu finden. Dabei ist zu beachten, dass dieser Bildpunkt nur gültig ist, wenn das Line-Valid-

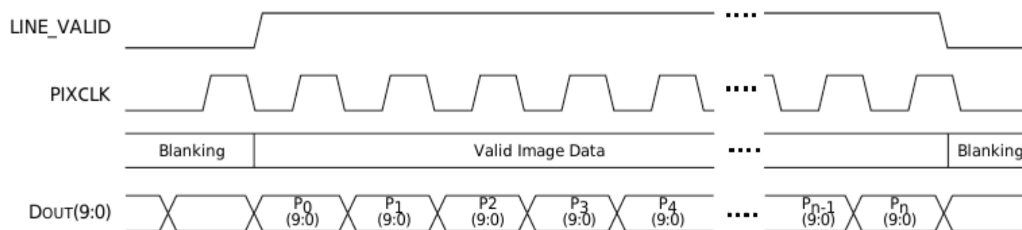

Abbildung 3.1.: Zeitverhalten des Line-Valid-Signals der Kamera [vgl. Mic06, S. 5]

Signal High-Pegel aufweist. Dieses Signal hält den High-Pegel für eine vollständige Bildzeile. Damit können zu einer Zeile gehörige Bildpunkte identifiziert werden. Um die Zugehörigkeit von Bildpunkten zu einem gesamten Bild herstellen zu können, wird das Frame-Valid-Signal benötigt. Dieses hat während der Übertragung eines vollständigen Bildes High-Pegel. Diese Abhängigkeiten sind auch noch mal in Abbildung 3.2 dargestellt. Hierbei ist zu erkennen, dass das Line-Valid-Signal nur dann high ist, wenn auch das Frame-Valid-Signal high ist. Des Weiteren ist die Dauer des High-Pegels des Line-Valid-Signals immer konstant. Diese

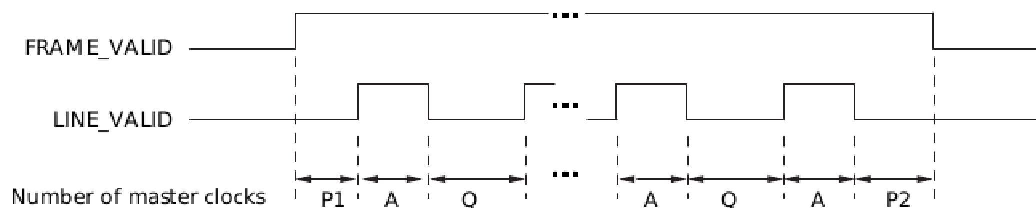

Abbildung 3.2.: Zeitverhalten des Line- und Frame-Valid-Signals der Kamera [vgl. Mic06, S. 5]

Zeitspanne wurde in der Grafik mit A gekennzeichnet. Zwischen zwei Line-Valid-Signalen gibt es einen Low-Pegel mit einer konstanten Länge. Dieser wurde mit Q bezeichnet. Die Zeiten P1 und P2 geben die Dauer zwischen einem High-Pegel des Frame-Valid-Signals und dem Anfang beziehungsweise Ende des gültigen Bilddatenstrom eines Bildes an. Da alle genannten Zeiten konstant sind, ist auch die Dauer des High-Pegels des Frame-Valid-Signals konstant. [vgl. Mic06, S. 4-5]

Mit den Standard-Einstellungen der Kamera beträgt der Bildtakt 26,6 MHz. Daraus ergeben sich High-Phasen des Line-Valid-Signals von 28,02 µs und eine High-Phase des Frame-Valid-Signals von 15,23 ms. Daraus folgt, dass 60 Bilder pro Sekunde übertragen werden. [vgl. Mic06, S. 5-7]

Mit einem High-Pegel auf der STANDBY-Leitung kann die Kamera in einen Energiesparmodus versetzt werden. Durch einen Low-Pegel auf der RESET-Leitung wird die Kamera zurückgesetzt und stellt ihre Standard-Einstellungen wieder her. Diese können, wie in Kapitel 3.2 erklärt, verändert werden. Um diese Änderungen der Einstellungen an die Kamera übertragen zu können, werden die Leitungen I2C_CLK und I2C_DATA benötigt. Diese bilden eine I^2C-Schnittstelle.

Da diese Leitungen nur mit einer geringen Frequenz von 500 Hz betrieben werden, könnten sie auch über die Pmod-Anschlüsse mit dem FPGA verbunden werden. Da die Verbindung mit der VHDC-Buchse aber ehe benötigt wird, bietet es sich an diese auch für die I^2C-Schnittstelle zu verwenden. Damit besteht die Möglichkeit einen zusammenhängenden Bus zu verwenden und somit fehleranfällige Verbindungen, die über einzelnen Leitungen hergestellt werden, zu vermeiden.

3.2. Konfiguration

Die für die Konfiguration benötigte I^2C-Schnittstelle der Kamera wird auch mit dem FPGA verbunden. Dabei wird im FPGA eine I^2C-Schnittstelle als Master betrieben. Diese wird durch einen MicroBlaze angesteuert. Der MicroBlaze führt Software aus, die die Kamera letztendlich konfiguriert. Die dafür benötigte Hardware wird von Xilinx zur Verfügung gestellt. Die benötigte Software wurde aus der Bachelorarbeit mit dem Namen "Integration einer CMOS-Kamera mit parallelem Interface in ein System-on-Chip basiertes Spurführungssystem" übernommen. [Sal13] Abgeändert wurden hierbei nur vereinzelnde Parameter. So wurde die Auflösung von 640x480 auf 752x480 Bildpunkten erhöht. Damit musste auch die erste Spalte um 56 Bildpunkte weiter nach außen geschoben werden. Des Weiteren wurden die Pausen-Zeiten zwischen den Zeilen und jedem Bild soweit minimiert, wie es das Timing der Kamera zuließ. Dies wurde durch Funktionstests mit unterschiedlichen Konfigurationen ermittelt.

Vorteil dieses Lösungsweges ist die hohe Flexibilität. So kann zum Beispiel der MicroBlaze durch kleine Änderungen auch mit der in Kapitel 5.9 beschriebenen Ethernet-Schnittstelle verbunden werden. Damit bestünde die Möglichkeit, die Software so zu verändern, dass die Kamera auch direkt über Ethernet vom PC aus konfiguriert werden könnte. Weiterhin wäre die Übertragung von anderen Parametern zur Konfiguration des FPGAs vom PC aus möglich. Denkbar wäre dabei eine Erweiterung um eine projektive Transformation in Hardware, wobei die benötigte Transformationsmatrix vom PC aus konfiguriert werden könnte.

4. Bildverarbeitungskette

Wie schon in der Einleitung erwähnt wurde, ist Ziel dieser Arbeit, einen Bilddatenstrom zu binarisieren. Dabei wird die Umsetzung der adaptiven Binarisierung in diesem Kapitel erläutert. Auf das Erosionsfilter wird auch eingegangen.

Das Zusammenspiel der einzelnen Verarbeitungsstufen ist in Abbildung 4.1 dargestellt. Dabei

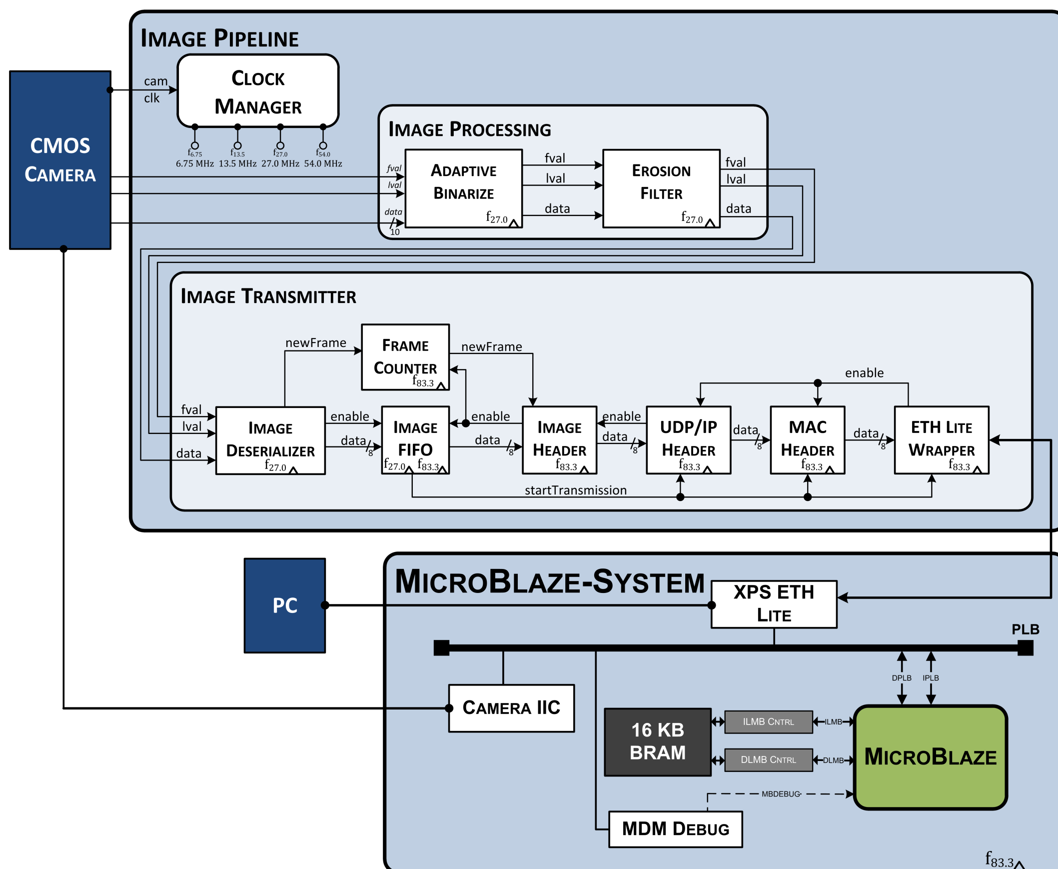

Abbildung 4.1.: Übersicht des Gesamtsystems mit Xilinx Micro-Blaze [vgl. Kir12, S. 22] [Wis13b]

ist zu erkennen, dass die Kamera direkt mit der adaptiven Binarisierung verbunden ist. Anschließend wird der Bilddatenstrom im Erosionsfilter weiter verarbeitet. Nachdem auch dieser Schritt durchgeführt wurde, werden die Daten im Bilddaten-Sender aufbereitet. Dieser wird im nachfolgenden Kapitel beschrieben. Das Versenden der Daten wird vom "Xilinx Ethernet Lite MAC" übernommen. Dieser ist in dem Block, der den MicroBlaze darstellt, zu finden.

4.1. Adaptive Binarisierung

Die Grauwert-Bilddaten der Kamera werden direkt in der adaptiven Binarisierung weiter verarbeitet. Dabei wird das selbe Verfahren, wie auch schon in Kapitel 2.1.1 beschrieben, verwendet. Der Unterschied ist nur, dass es nicht als Software ausgeführt wird, sondern extra Hardware für diese Berechnung im FPGA implementiert wurde. Die Binarisierung wurde mit dem Xilinx System Generator erstellt. Sie ist in Abbildung 4.2 dargestellt. Die Herkunft dieser Implementierung ist die Masterarbeit "FPGA-basierte MPSoC-Plattform zur Integration eines Antikollisionssystems in die Fahrspurführung eines autonomen Fahrzeugs". [vgl. Kir12, S. 27-29] Der in der Abbildung zu sehende Eingang mit dem Namen "data_in" ist 8 Bit breit und wird

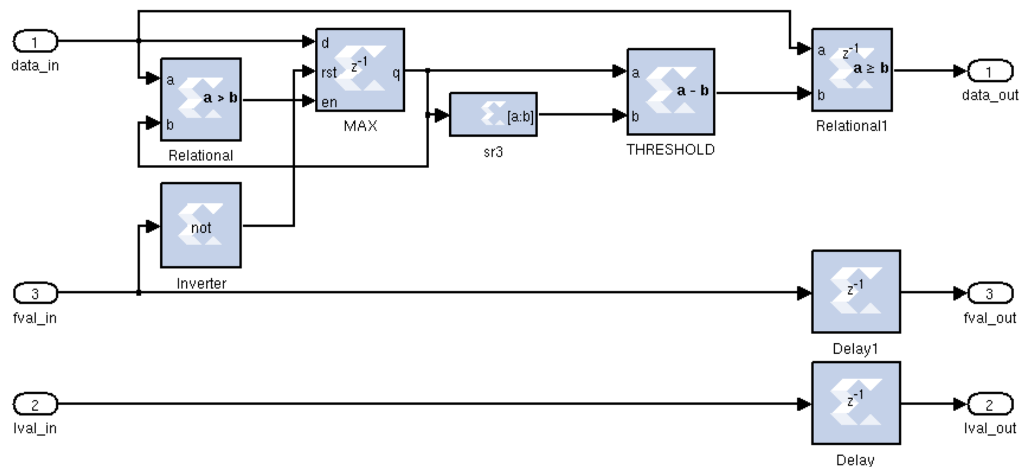

Abbildung 4.2.: Adaptive Binarisierung mit Xilinx System Generator erstellt [Kir12]

mit den Grauwerten der Kamera für jeden Bildpunkt versorgt. Mit jedem Takt wird der nächste Bildpunkt übergeben. Der Wert des hellsten Bildpunktes wird in dem Register "MAX" gespeichert. Dieses wird mit Beginn eines neuen Bildes durch das "fval"-Signal wieder zurück gesetzt.

Der Schwellwert wird berechnet, indem der um drei Bit nach rechts geschobene Maximalwert vom Maximalwert subtrahiert wird. Drei Bit nach rechts schieben ist im Dual-System mit durch acht dividieren gleich zusetzen. Also kann dies auch durch $g_{max} - \frac{g_{max}}{8}$ ersetzt werden. Wobei g_{max} der Maximalwert aus dem Register “MAX” ist. In Prozent ausgedrückt ist durch acht dividieren mit 12,5 % darstellbar. Daraus ergibt sich $g_{max} - g_{max} * 12,5\%$. Wenn jetzt noch g_{max} ausgeklammert wird, ergibt sich daraus $g_{max} * 87,5\%$. Dies ist sehr ähnlich der Formel die unter 2.1 erläutert wurde.

Damit ist sichergestellt, dass der Schwellwert in Hardware dem in Software sehr ähnelt. Der Vergleich zwischen Schwellwert und Ist-Wert wird von “Relational1” übernommen. Der Ausgang “data_out” hat damit nur zwei Zustände und repräsentiert entweder einen schwarzen oder weißen Bildpunkt. In der weiteren Bildverarbeitungskette werden nur noch diese durch ein Bit dargestellten Bildpunkte verwendet.

4.2. Erosionsfilter

In Kapitel 2.1.2 wurde schon kurz auf die Funktionsweise einer Erosion eingegangen. Wie dort beschrieben, wird auch in der Hardware-Implementierung ein Bildpunkt mit seinen acht Nachbarn UND-Verknüpft. Diese Operation ist mit dem Suchen des Minimums vergleichbar. Wenn ein Bildpunkt der neun untersuchten Punkte den Wert 0 aufweist, übernimmt der mittlere Bildpunkt den Wert 0. Erst wenn alle acht Nachbarn und der bearbeitet Bildpunkt selbst mit dem Wert 1 belegt sind, behält der resultierende Bildpunkt auch den Wert 1. [vgl. Erh08, S. 171]

Die Hardware, die dies erfüllt, wurde mit Hilfe des Xilinx System Generators erstellt. Sie ist in Abbildung 4.3 zu finden. Die Implementierung wurde wie auch die Binarisierung aus der Masterarbeit “FPGA-basierte MPSoC-Plattform zur Integration eines Antikollisionssystems in die Fahrspurführung eines autonomen Fahrzeugs” übernommen. [vgl. Kir12, S. 27-29] So wie die Daten von der Kamera gesendet werden, werden sie auch durch das Erosionsfilter geschoben. Das bedeutet, dass der binarisierte Pixel-Strom Zeile für Zeile verarbeitet wird. Die Minimum-Berechnung bzw. UND-Verknüpfung ist in dem Block “Logical” zu finden. Dieser besitzt neun Eingänge, da der aktuelle Bildpunkt und seine acht Nachbarn verglichen werden.

Da es sich um einen Bildpunkt-Strom handelt und die Bildpunkte somit nicht in einem Speicher abgelegt werden, auf den wahlfrei zugegriffen werden kann, müssen diese in Schieberegistern zwischengespeichert werden. Die drei Bildpunkte der aktuellsten Zeile werden in den

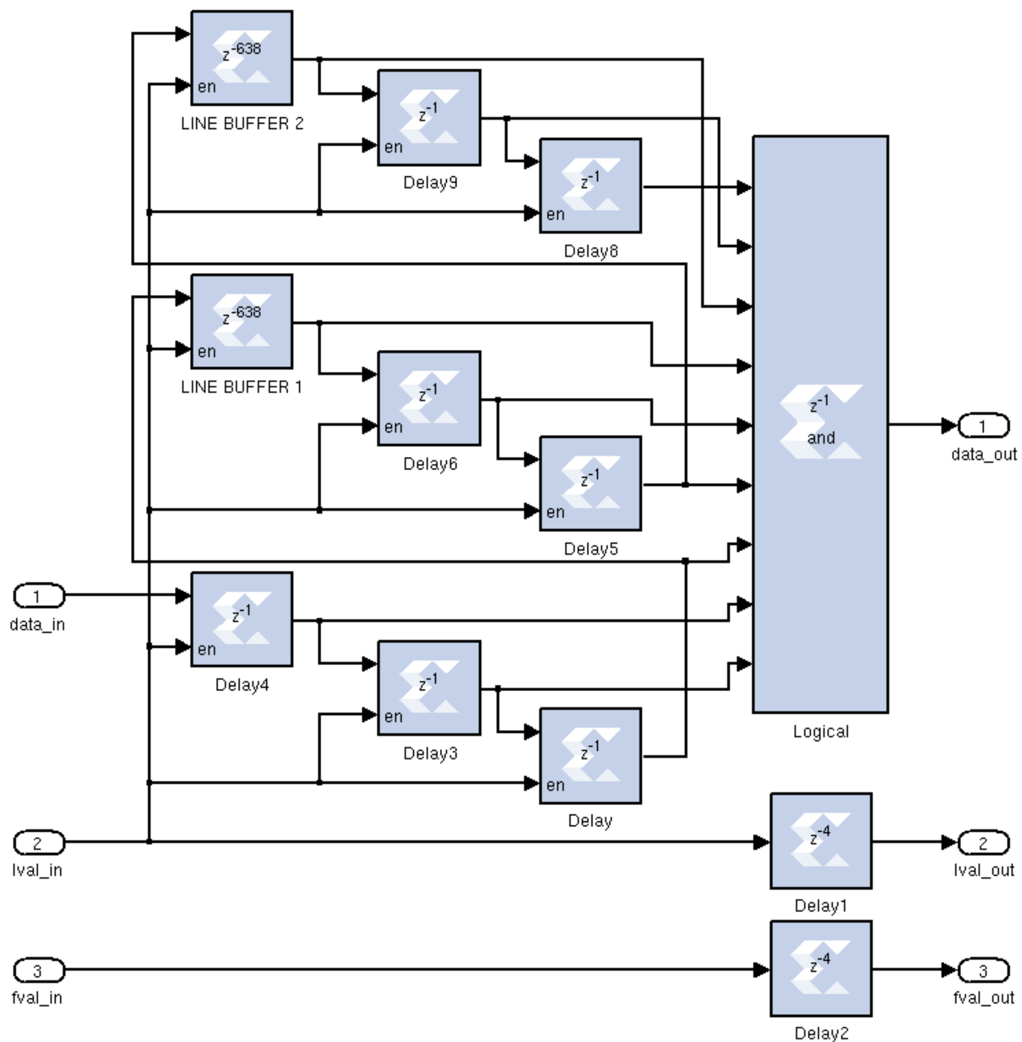

Abbildung 4.3.: Erosionsfilter mit Xilinx System Generator erstellt [Kir12]

Registern “Delay”, “Delay3” und “Delay4” zwischengespeichert. Für die mittlere Zeile sind die Register “Delay5”, “Delay6” und “LINE BUFFER 1” zuständig. Wobei der Block “LINE BUFFER 1” eine Verzögerung von 638 Bildpunkten erzeugt. Dies ergibt mit den voran geschalteten Registern “Delay” und “Delay3” eine Verzögerung von 640 Pixeln. Dies ist genau die Breite des Kamerabildes, was bedeutet, dass dadurch eine Verzögerung um genau eine Zeile entsteht. Die älteste der drei benötigten Zeilen wird durch die Register “Delay6”, “Delay9” und “LINE BUFFER 2” gespeichert. Hierbei erzeugt die Kombination aus Register “Delay5”, “Delay6” und “LINE BUFFER2” wieder eine Verzögerung um eine ganze Zeile.

Da die angeschlossene Kamera zwischen den Zeilen keine gültigen Bildpunktwerte sendet, wird das Signal “lval_in” berücksichtigt. Nur wenn “lval_l” einen High-Pegel aufweist und somit die Bilddaten gültig sind, wird in den Schieberegistern weitergeschoben. Andernfalls wird der alte Wert beibehalten und auf den nächsten gültigen Bildpunkt gewartet.

5. Bilddaten-Sender

Nachdem die Bilddaten binarisiert und erodiert wurden, sollen sie zum PC übertragen werden. Diese sollen, wie in der Voruntersuchung in Kapitel 2.3 begründet, über eine Ethernet-Schnittstelle versendet werden.

Um die Daten versenden zu können, werden sie aufbereitet. Eine Übersicht des Bilddaten-Senders ist in Abbildung 5.1 dargestellt. Der Eingang für die binarisierten und erodierten Daten

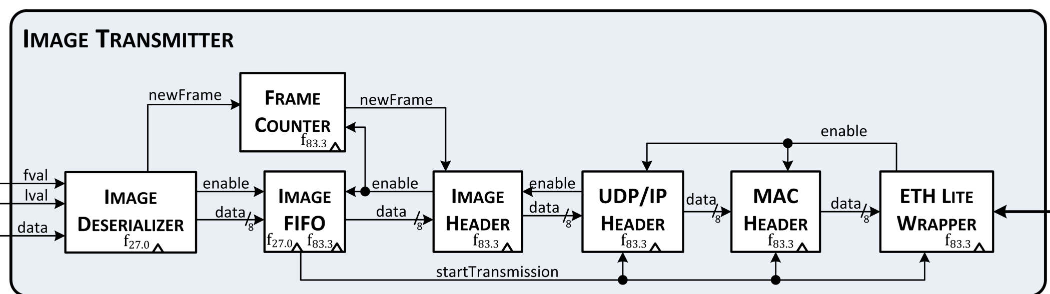

Abbildung 5.1.: Zusammenhang der einzelnen Komponenten des Bilddaten-Senders [Wis13b]

befindet sich auf der linken Seite. Auf der rechten Seite ist die Anbindung an den "Xilinx Ethernet Lite MAC" zu finden. Die Teilabschnitte, die durchlaufen werden, befassen sich mit dem Zusammenfassen mehrere Bildpunkte, der Zwischenspeicherung und dem Hinzufügen von für den Versand benötigten Header-Informationen. Der erste Schritt ist das Zusammenfassen der Bildpunkte. Dieser wird im folgenden Unterkapitel beschrieben.

5.1. Deserialisierung

Um einen größeren Durchsatz der Daten bei gleichem Takt zu erzielen, werden die Daten deserialisiert. Damit ist gemeint, dass immer acht Bildpunkte des Bilddatenstroms zu einem Byte zusammen gefasst werden und diese dann parallel weiter durch die Pipeline geschoben werden. Diese Deserialisierung ist speziell beim Senden eines Ethernet-Paketes von Interesse. Denn ein Ethernet-Paket muss zusammenhängend versendet werden. Die Ethernet-Schnittstelle

wird, um die ausreichend Bandbreite zur Verfügung zu haben, mit 100 MHz betrieben. Um die Nutzdaten nun schnell genug übertragen zu können, muss alle 10 ns ein Bit übertragen werden. Der Bilddaten-Sender wird aber nur mit 83,3 MHz getaktet. Somit wäre eine weiterhin seriell aufgebaute Pipeline zu langsam.

Ein weiterer Vorteil beim Zusammenfassen der Bilddaten auf acht Bit ergibt sich beim Hinzufügen der benötigten Header-Informationen. Fast alle Header haben ihre Eigenschaftsfelder auf Byte-Grenzen angeordnet und damit auch immer eine Länge, die in Byte angegeben wird.

Unter 5.1 ist der wichtigste Teil der Deserialisierung dargestellt.

```
1     if (frameValidIn = '0') then
2       counter_v           := SHIFT_COUNTER_INIT;
3     end if;
4
5
6     -- only save the serial image data, if it is a valid line
         and frame
7     if (lineValidIn = '1' and frameValidIn = '1') then
8       parallelData_v(counter_v) := serialIn;
9
10      -- output an short pulse for one clk, if a new byte is
           available in the shift register
11      if (counter_v = 0) then
12        counter_v           := SHIFT_COUNTER_INIT;
13        newByte_v         := '1';
14      else
15        counter_v         := counter_v - 1;
16      end if;
17    end if;
```

Listing 5.1: Deserialisierung der Bilddaten mit Synchronisation auf den Bildanfang [Wis13b]

Die Zeilen 1 bis 3 werden für die Synchronisation mit dem Anfang eines neuen Bildes benötigt. Damit wird bewirkt, dass ein Byte nicht aus Teilen des letzten und des neuen Bildes besteht. Um Bildgrenzen feststellen zu können, wird das Frame-Valid-Signal in Zeile 1 ausgewertet. Dieses Signal wird von der Kamera generiert und begleitet dabei den Bilddatenstrom durch die gesamte Pipeline. Der Vektor “counter_v” beschreibt die Bit-Position im parallelen Ausgangsvektor “parallelData_v”. Das bedeutet, dass nach Beginn eines neuen Bildes in Zeile 8 mit dem Überschreiben des LSBs vom Ausgangsvektor begonnen wird. Der Vektor “counter_v”

wird in Zeile 15 solange dekrementiert bis der Ausgangsvektor mit acht Bit überschrieben wurde. Wenn dies statt gefunden hat, wird mit Zeile 13 signalisiert, dass ein neues Byte für die Abspeicherung im FIFO-Speicher zur Verfügung steht. Im gleichen Zuge wird der Vektor “counter_v” in Zeile 12 zurück gesetzt. Somit wird im nächsten Schritt wieder das LSB überschrieben.

Nicht nur die Bilddaten sind in den nachfolgenden Pipeline-Stufen von Interesse. Auch der Beginn eines neuen Bildes ist sehr wichtig. Um diesen signalisieren zu können, wird die unter Listing 5.2 gezeigte Hardware verwendet.

```
1      if (shortFrameValidIn_s = '1') then
2        newFrameOut_v := '1';
3      elsif (newByte_cs = '1') then
4        newFrameOut_v := '0';
5      else
6        newFrameOut_v := newFrameOut_cs;
7      end if;
```

Listing 5.2: Generierung der Signalisierung für den Beginn eines neuen Bildes [Wis13b]

Das Signal “shortFrameValidIn_s” stellt das Frame-Valid-Signal der Kamera da, das durch einen Pulse-Shorter auf eine Länge von einem Takt verkürzt wurde. Somit wird “newFrameOut_v” gesetzt, wenn ein neues Bild beginnt. Dieser Zustand wird mit Zeile 6 solange gespeichert, bis das erste Byte des neuen Bildes zur Verfügung steht. Dann wird es in Zeile 4 wieder zurückgesetzt. Der Speicher, der die deserialisierten Daten zwischenspeichert wird im nächsten Kapitel beschrieben.

5.2. Bilddaten-Puffer

Um eine schnellere Übertragung von einzelnen Paketen ermöglichen zu können, müssen die Nutzdaten für ein Paket zwischengespeichert werden. Dies passiert in dem in diesem Kapitel beschriebenen FIFO-Speicher. Die Größe dieses Speichers ist stark von der Größe eines Paketes abhängig. Ein Ethernet-Paket kann maximal 1500 Byte Nutzdaten aufweisen. Dies wird durch die MTU (Maximum Transmission Unit) beschrieben. [vgl. Pos83, S. 5] Wenn alle weiteren Header vernachlässigt werden, müsste der FIFO-Speicher eine Mindestgröße von 1500 Byte haben. Damit mögliche Verzögerungen bis zum Übertragungsbeginn keine kritischen Auswirkungen haben, wurde ein größerer Speicher gewählt. Des Weiteren bewährt es sich in Hardware-Beschreibungssprachen Zahlen, die vielfache von zwei sind, zu verwenden. Somit wurde für diesen Speicher eine Größe von 2048 Byte gewählt.

Wie schon des öfteren erwähnt, besteht das Gesamtsystem aus zwei unterschiedlichen Takten. Der eine Takt der von der Kamera generiert wird und mit dem die Bilddaten zum FPGA übertragen werden und der andere mit dem der Bilddaten-Sender betrieben wird. Das Problem ist dabei, dass die Bilddaten zwischen diesen zwei Takt-Systemen ausgetauscht werden müssen. Dafür könnten zwei Flip-Flops, die hintereinander geschaltet werden, verwendet werden. Dieser Ansatz wird auch als Double-Flopping bezeichnet. Damit wäre die Wahrscheinlichkeit von metastabilen Zuständen zwar schon sehr minimiert, aber noch nicht vollkommen ausgelöscht. Besser bietet sich die Verwendung eines FIFO-Speichers mit zwei Takt-Eingängen an. Hierbei muss intern weiterhin das Problem der metastabilen Zustände behoben werden. Dabei wird auch hier Double-Flopping verwendet. Zusätzlich wird aber für den Austausch der Lese- und Schreib-Zähler der Gray-Code genutzt. Der Vorteil des Gray-Codes liegt darin, dass sich beim Inkrementieren oder Dekrimentieren immer nur ein Bit verändert. Somit wird der alte oder der neue Zählerstand aber nie ein ungültiger Wert übertragen. [vgl. Kil07, S. 89-96] Diese Zählerstände werden von den weiteren Pipeline-Stufen auch benötigt.

Der in dieser Arbeit verwendete FIFO-Speicher wurde mit Hilfe des Xilinx-Core-Generators erstellt. Dieser ist für viele Anwendungsfälle konfigurierbar und somit bei Erweiterungen schnell durch eine andere Variante ersetzbar.

Um zu bestimmen, wann eine neue Übertragung eines Paketes gestartet werden kann, wird ein Schwellwert für den Füllstand des FIFO-Speichers verwendet. Dieser ist auf 1500 Byte eingestellt. Sobald dieser Schwellwert erreicht wird, wird ein Übertragungsstart bei der in Kapitel 5.9 beschriebenen Ethernet-Anbindung in Auftrag gegeben. Nun können noch einige Takte vergehen, bis Daten aus dem FIFO-Speicher gelesen werden, da vorher noch der Beginn eines Ethernet-Paketes mit Hilfe einer Präambel bekannt gemacht wird. Somit kann der Füllstand des FIFO-Speichers diese 1500-Byte-Grenze überschreiten.

Um Fehlverhalten bezogen auf physikalisch bedingte Grenzen des FIFO-Speichers schnell erkennen zu können, wird das Voll- und Leer-Signal dieses Speichers ausgewertet. Die dafür entwickelte Logik ist in Listing 5.3 zu finden.

```
1  errorOutput: process (wasEmpty_cs, isFull_s, dataOut_s) is
2  begin
3    if (wasEmpty_cs = '1') then
4      dataOut <= x"EE";
```

```
5      elsif (isFull_s = '1') then
6        dataOut <= x"FF";
7      else
8        dataOut <= dataOut_s(1 to 8);
9      end if;
10   end process errorOutput;
```

Listing 5.3: Fehlerbehandlung für Voll- und Leer-Signal des Bilddaten-Puffers [Wis13b]

Sofern der Puffer nach dem ersten Auffüllen einmal leer war, wird jedes Byte, das aus dem Speicher gelesen wird, in Zeile 4 mit EE_{16} ersetzt. Dies schlagt bis zur Anzeige auf dem PC durch und kann schnell erkannt werden. Wenn der Puffer voll ist, wird jedes Byte in Zeile 6 durch FF_{16} ersetzt. Auch dieses wäre im dekodierten Bild erkennbar. Somit ist ein schnelles Feststellen von Underflow- oder Overflow-Fehlern garantiert.

5.3. Bildgrenzen-Zähler

Um das spätere Teilen des Datenstroms in die einzelnen Bilder auf der Empfängerseite zu vereinfachen, bietet es sich an, den Anfang eines neuen Bildes auch mit einem neuen Paket beginnen zu lassen. Das bedeutet aber auch, dass das Ende des vorherigen Bildes mit einer Paketgrenze abschließen muss. Die Länge eines Paketes muss aber vor dem Start der Übertragung dieses Paketes feststehen, da diese mit in die Header-Informationen einfließt. Um dies ermöglichen zu können, muss ein Zähler existieren, der die noch verbleibende Datenmenge des aktuellen Bildes wieder spiegelt. Dieser Zähler wird beachtet, wenn das erste Byte eines neuen Bildes in den FIFO-Speicher eingefügt wurde. Damit wird ein neuer Sendeauftrag mit der Nutzdatenlänge des Zählerstandes generiert. Somit ist es hinreichend, wenn dieser Zähler parallel zum FIFO-Speicher angeordnet wird und keine weiteren Informationen von anderen Pipeline-Stufen erhält.

Um einen von den Bildgrenzen abhängigen Zähler zu ermöglichen, wird der Füllstand-Zähler des FIFO-Speichers mit ausgewertet. Die Logik, die auf diesen Zähler aufbaut, ist unter Listing 5.4 wieder zu finden.

```
1      if (newFrameInCounter_v = '1') then
2        if (readEnable = '1') then
3          if (countTillFrame_v = x"0001") then
4            countTillFrame_v  := elementCounter;
5            newFrameInCounter_v := '0';
6          else
```

```
7            countTillFrame_v   := countTillFrame_v - 1;
8          end if;
9        end if;
10     else
11       -- save the current size of the queue in the counter until
            a new frame start was received
12       countTillFrame_v    := elementCounter;
13     end if;
14
15
16     if (shortWriteEnable_s = '1') then
17       -- a new frame begins so remember that this is in the
            counter
18       if (frameStartIn = '1') then
19         newFrameInCounter_v := '1';
20       end if;
21     end if;
```

Listing 5.4: Elementzähler zur Berechnung der Bildgrenzen [Wis13b]

Solange sich keine Bildgrenze in dem FIFO-Speicher befindet, wird der Bildgrenzen-Zähler "countTillFrame_v" mit dem FIFO-Zählerstandes beschrieben. Dieser Kopiervorgang ist in Zeile 12 implementiert. Sobald in den Zeilen 16 bis 21 der Beginn eines neuen Bildes erkannt wurde, wird dies in der Variable "newFrameInCounter_v" gespeichert. Das hat zur Folge, dass der Bildgrenzen-Zähler nicht mehr vom Zählerstand des FIFO-Speichers abhängig ist. Der Bildgrenzen-Zähler wird ab jetzt in Zeile 7 dekrementiert, wenn ein Element aus dem FIFO-Speicher gelesen wird. Dies passiert solange, bis das letzte Byte des aktuellen Bildes aus dem Speicher gelesen wurde. Wenn dies geschehen ist, wird der Bildgrenzen-Zähler in Zeile 4 wieder mit dem Wert des FIFO-Zählerstandes überschrieben und der Ablauf beginnt von neuem. Der Bildgrenzen-Zähler gibt damit die Nutzdatenlänge an. Deshalb wird dieser Wert an die nachfolgenden Module weitergereicht, da die Paketlängen und damit auch teilweise Eigenschaftsfelder der Header davon abhängig sind.

5.4. Absende-Steuerung

Dieser zuvor beschriebene Zählerstand wird speziell beim Start einer neuen Übertragung benötigt. Das kann durch den Füllstand des FIFO-Speichers oder den Beginn eines neuen Bildes im Speicher ausgelöst werden. Dies bedeutet aber nicht immer einen sofortigen Sendebeginn. Vorerst wird überprüft, ob die Sende-Einheit nicht belegt ist und bereit für den Start einer neuen Übertragung.

Die in Listing 5.5 zu finden Hardware startet eine Übertragung kontrolliert an.

```
1  startTransmissionCtrl: process (fifo2ctrlStartTrans_s, txDone,
       txDone_cs, ctrl2imgStartTrans_cs) is
2  begin
3    if (fifo2ctrlStartTrans_s = '1') then
4      ctrl2imgStartTrans_ns <= '1';
5    elsif (txDone_cs = '0' and txDone = '1') then
6      -- only reset the start trans request on a rising edge
7      ctrl2imgStartTrans_ns <= '0';
8    else
9      ctrl2imgStartTrans_ns <= ctrl2imgStartTrans_cs;
10   end if;
11 end process startTransmissionCtrl;
12
13
14 ctrl2imgShortStartTrans_s <= ctrl2imgStartTrans_ns and not
       ctrl2imgStartTrans_cs;
```

Listing 5.5: Steuerung für den Beginn einer neuen Übertragung [Wis13b]

Das Signal "fifo2ctrlStartTrans_s" enthält die Anforderung vom Füllstand des FIFO-Speichers und für einen neuen Bildanfang. Dabei wird über dieses Signal ein kurzer Pulse mit einer Länge von einem Takt übertragen, wenn eine dieser Anforderungen anliegen. Wenn dieser Pulse anliegt wird in Zeile 4 die Sendeanforderung in dem Signal "ctrl2imgStartTrans_ns" gespeichert. Diese Anforderung wird in Zeile 7 zurück genommen, wenn in Zeile 5 erkannt wurde, dass die Sendeeinheit das Senden beendet hat. Wenn keine dieser Bedingungen zutrifft, wird der alte Zustand in Zeile 9 übernommen. Die Sendeeinheit und einige Header erwarten einen kurzen Pulse und kein andauernden High-Pegel, um eine Übertragung zu beginnen. Dieser Pulse wird in Zeile 14 erzeugt und hat eine Länge von genau einem Takt. Die dort genutzte Logik wird auch Pulse-Shorter genannt.

5.5. Bilddaten-Header

Um die Bilddaten auf der Empfänger-Seite richtig interpretieren zu können, müssen diese mit Header-Informationen versehen werden. Eigenschaften die in diesem Header kodiert sind, sind unter anderem die Auflösung des Bildes, Farbtiefe eines Bildpunktes und das verwendete Komprimierungsverfahren. Dabei sollte dieser Header möglichst klein sein, damit die Belastung der Ethernet-Schnittstelle minimiert wird. Der Header sollte auch eine gut von den Nutzdaten unterscheidbare Magic-Number zur Identifikation des Datentyps enthalten, um sich auf der Empfänger-Seite schnell auf den Datenstrom aufsynchronisieren zu können. Da ein Bildpunkt durch ein Bit repräsentiert werden kann, sollte diese kompakte Farbtiefe vom Bilddaten-Header unterstützt werden können.

Die Suche begann bei sehr verbreiteten Bildformaten. Ein sehr bekanntes Format ist die "Microsoft Windows Bitmap". Bei näherer Betrachtung der Version 4 fällt die Header-Größe mit 108 Byte negativ auf. Die Magic-Number dieses Headers ist mit 2 Byte für eine Synchronisation in einem Stream auch sehr kurz. Positiv ist aufgefallen, dass dieses Format die benötigte Farbtiefe von einem Bit unterstützt. Bei weiteren Analysen musste aber festgestellt werden, dass bei einer Verwendung dieser Farbtiefe eine zusätzliche Farbpalette benötigt wird. Dies bedeutet, dass zusätzlich zum minimalen Header mit einer Länge von 108 Byte noch weitere Bytes für diese Farbpalette benötigt werden. [Mv96] Somit wurde das "Microsoft Windows Bitmap"-Format als ungünstige Lösung für den Anwendungsfall in dieser Arbeit beurteilt.

Ein vielversprechenderes Format trägt den Namen "Sun Raster". Mit einer Header-Länge von nur 8 Byte, hat es schon einen großen Vorteil gegenüber der Microsoft-Variante. Die Magic-Number hebt sich mit einer Länge von 4 Byte und der Folge $59A66A95_{16}$ sehr gut von den zu erwarteten Nutzdaten ab. Oftmals haben in unkomprimierten Bilddaten die Nachbar-Pixel den selben Farbwert. Das bedeutet wieder herum, dass es sehr selten zu Bit-Änderungen innerhalb eines Bytes kommen wird. Eine Farbpalette wird für die Farbdarstellung mit einem Bit in diesem Format auch nicht benötigt. So ergibt sich ein Overhead von 8 Byte bei Verwendung des "Sun Raster"-Bildformates. [Mv96]

Der Header des "Sun Raster"-Format besteht aus den, in Tabelle 5.1 dargestellten, Feldern. Die jeweiligen Werte, die in dieser Arbeit verwendet werden sind in dieser Tabelle in der letzten Spalte gelistet. Die Größe des Bildes wird in Byte angegeben und lässt sich durch $\frac{Width*Height*Depth}{8}$ berechnen. Das durch 8 dividieren ist notwendig, da die Farbtiefe eines

Bezeichnung	Länge	Beschreibung	Wert
MagicNumber	4 Byte	Identifikation des Bildformates	$59A66A95_{16}$
Width	4 Byte	Breite des Bildes in Pixel	640_{10}
Height	4 Byte	Höhe des Bildes in Pixel	480_{10}
Depth	4 Byte	Farbtiefe eines Pixels in Bit	01_{16}
Length	4 Byte	Größe der Bilddaten in Byte	38.400_{10}
Type	4 Byte	Version oder Komprimierungsverfahren	01_{16}
ColorMapType	4 Byte	Art der Farbpalette	00_{16}
ColorMapLength	4 Byte	Größe der Farbpalette in Byte	00_{16}

Tabelle 5.1.: Eigenschaftsfelder des Sun Raster Headers [Mv96]

Bildpunktes in Bit angegeben wird. Als Komprimierungsverfahren wurde in diesem Fall "Standard" gewählt. Das bedeutet, dass die Bilddaten nicht komprimiert werden. Da keine zusätzliche Farbpalette verwendet werden sollen, wird die Farbpaletten-Art und die Farbpaletten-Größe auf 0 gesetzt. [Mv96]

Um diesen Header nun vor dem Beginn jedes neuen Bildes senden zu können, müssen Bytes in die Bild-Pipeline eingefügt werden können. Dies wird in diesem Fall durch einen Multiplexer, der durch einen Zähler adressiert wird, realisiert. Die entsprechende Hardware für dieses Verhalten ist unter Listing 5.6 zu finden.

```
1      -- Only load on the rising edge
2      if (loadHeader = '1' and loadHeader_cs = '0') then
3        header_v   := headerIn;
4        counter_v  := 0;
5      end if;
6
7
8      if (counter_v = LENGTH) then
9        dataOut_v    := dataIn;
10     else
11       dataOut_v    := header_v( counter_v*WIDTH to
            (counter_v*WIDTH + WIDTH - 1) );
12       -- Do not reqest the next byte from the dataIn because
            firstly the header should be sent
13       nextByteReq_v := '0';
14
15       -- only decriment the counter if a byte is realy read
16       if (getNextByte = '1') then
```

```
17          counter_v    := counter_v + 1;
18        end if;
19      end if;
```

Listing 5.6: Multiplexer und Zähler zum Einfügen des Bild-Headers [Wis13b]

Der Zähler wurde "counter_v" genannt. Solange dieser kleiner ist als die Länge des einzufügenden Headers, adressiert der Zähler ein entsprechendes Byte im Header. Dies ist in Zeile 11 implementiert. In dieser Zeit wird durch Zeile 13 bewirkt, dass keine Bilddaten aus dem FIFO-Speicher geholt werden. Wenn der Zähler seinen Endwert erreicht hat, werden die Bilddaten durch gereicht. Dies ist in Zeile 9 umgesetzt. Die Anforderung auf das nächste Byte aus dem FIFO-Speicher wird auch nicht mehr überschrieben und somit werden neue Daten aus dem Speicher geholt. Um einen Header einzufügen wird der Zähler in Zeile 4 auf 0 gesetzt. Dies wird durch die Signalisierung des FIFO-Speichers, dass ein neues Bild beginnt, ausgelöst.

Diese gesamte Funktionalität ist auch mit einem Schieberegister lösbar. Dabei wäre die Implementierung aber komplexer, da es schwierig ist, im Voraus festzustellen, wann das nächste Bild beginnt. Das Problem ist, dass alle Bilddaten die sich gerade im Schieberegister befinden, versendet werden müssen und erst dann können die Register mit den Header-Daten gefüllt werden. Wenn die sich in den Registern befindlichen Daten einfach überschrieben werden würden, würden Bilddaten verloren gehen.

5.6. UDP-Header

Damit Pakete im Ethernet ihren richtigen Weg finden, müssen die Daten mit Protokollen gekapselt werden. Diese unterschiedlichen Protokolle können Schichten zugeordnet werden. Die Einordnung in diese Schichten wird im TCP/IP-Referenzmodell in Abbildung 5.2 veranschaulicht. Im Gegensatz zum OSI-Referenzmodell sind in diesem Modell die Schichten 5 und 6 nicht weiter definiert. Alles was sich oberhalb der Transportschicht befindet, ist dem Anwender überlassen. So kann der in dieser Arbeit verwendete "Sun Raster"-Header mit zur Verarbeitungsschicht gezählt werden.

Das User Datagram Protocol (UDP) wird mit in die Transportschicht eingeordnet. Die Wahl für die Transportschicht viel auf dieses Protokoll, da es verbindungslos ist. Das bedeutet, dass keine bidirektionale Kommunikation nötig ist, was den Implementierungsaufwand in Hardware stark minimiert. [vgl. MBW10, S. 226] Ein Gegenbeispiel wäre das Transmission Control Protocol. Dieses ist verbindungsorientiert und benötigt somit eine bidirektionale

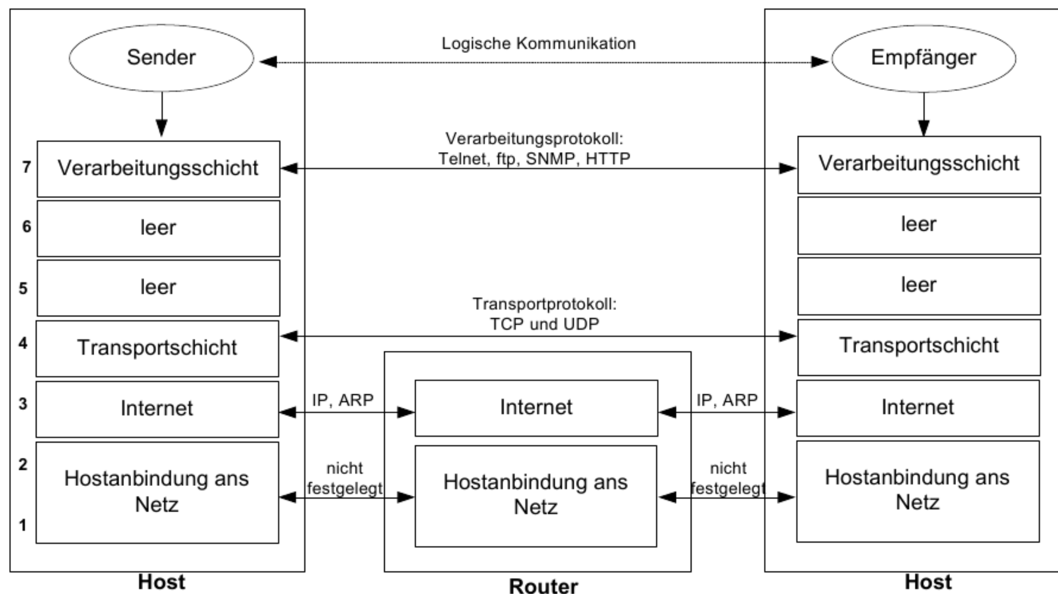

Abbildung 5.2.: TCP/IP-Referenzmodell zur Veranschaulichung der Protokoll-Schichten [vgl. MBW10, S. 10]

Kommunikation. [vgl. MBW10, S. 198]

Auch für dieses Protokoll existieren Hardware-Implementierungen. Eine Implementierung trägt den Namen "SiTCP". Dabei wurde sehr auf wenig Hardware-Bedarf aber große Übertragungsraten geachtet. Die Implementierung unterstützt Gigabit-Ethernet und ermöglicht damit Übertragungsraten von bis zu 949 Mb/s. [vgl. Uch07, S. 314] Um nur wenig Hardware zu benötigen, wurden einige Einschränkungen getroffen. Eine Einschränkung besteht zum Beispiel darin, dass nur eine TCP-Verbindung zu einer Zeit bestehen kann. [vgl. Uch07, S. 310] Dies ist aber für angedachte Erweiterungen, die für diese Bachelorarbeit angedacht sind, sehr ungünstig. So wäre es zum Beispiel nicht möglich, gleichzeitig einen zweiten Video-Strom zu versenden, der ein projektiv Transformiertes Bild enthält. Ein weiteres Argument gegen eine Übertragung der Daten über eine TCP/IP-Verbindung ist auf der Empfängerseite zu finden. Diese besteht weiterhin aus einem PC. Dabei ist der TCP/IP-Stack so wie auch der UDP/IP-Stack in Software implementiert. Da eine TCP-Verbindung aber einen mehr Ressourcen benötigt, hätte dies eine höhere Auslastung der CPU zur Folge. [vgl. Uch07, S. 209] Dies wieder herum wäre natürlich sehr von Nachteil, da die wertvollen CPU-Ressourcen für den Fahrspur-Algorithmus benötigt werden.

Ein weiterer aus der verbindungslosen Kommunikation hervorgehender Vorteil des UDPs entsteht dadurch, dass Bilddaten sofort versendet werden können, wenn die Maximalgröße eines Paketes erreicht wurde. Somit muss kein Puffer mit einer schwer berechenbaren Größe verwendet werden, da keine Bestätigung des Empfängers erwartet wird, um das nächste Paket versenden zu dürfen. Nachteil ist, dass der Verlust von Paketen auf der Empfängerseite nicht festgestellt werden kann. Das Interesse am neu Senden der Daten besteht aber ehe nicht, da die Daten dann schon veraltet wären. Ganz im Gegenteil ist es von Vorteil, wenn die Verzögerung bei der Übertragung der Daten minimiert werden kann und nicht auf die Antwort des Empfängers gewartet werden muss.

Die Paketreihenfolge kann mit UDP auch nicht wieder hergestellt werden. Dies wird aber nur relevant, wenn Pakete über Zwischenstationen mit Puffer-Funktionalitäten gesendet werden. Die Verbindung zwischen FPGA und PC wird in diesem Fall mit einem Crossover-Ethernet-Kabel hergestellt. Somit existieren keine puffernden Geräte zwischen dem Sender und dem Empfänger und die Reihenfolge der Pakete kann nicht durcheinander kommen.

Die Begründung für die Verwendung eines Protokolls in der Transportschicht liegt in der Flexibilität. Durch die Implementierung dieser Protokoll-Hierarchie ist grundsätzlich eine Kommunikation über Router-Grenzen hinweg möglich. Dabei ist die eben erwähnte Paketreihenfolge aber nicht mehr sicher gestellt. Dies kann für Erweiterungen sehr effizient nach implementiert werden, indem in jedem Paket ein Zählerstand mitgesendet wird. Dieser Zähler wird beim Versenden jedes Paketes inkrementiert. So kann die Empfängerseite die Reihenfolge der Pakete wieder herstellen. Ähnlich wird dies auch bei TCP umgesetzt. Hier wird der Zähler nicht nur inkrementiert, sondern dieser spiegelt die Byte-Position im Datenstrom wieder. [vgl. MBW10, S. 210-211] Ein denkbares Szenario für eine solche Erweiterung wäre das Weiterleiten des Video-Stroms über WLAN, um diesen während der Fahrt des CaroloCup-Fahrzeuges analysieren zu können.

Der Sinn eines Protokolls auf der Transportschicht liegt in der Identifikation eines Prozesses auf einem Endknoten. [vgl. MBW10, S. 4] Diese Identifikation wird bei UDP über einen Port, der eine Größe von zwei Byte hat, umgesetzt. Weitere Eigenschaftsfelder des UDP-Headers sind in Tabelle 5.2 aufgelistet. Die letzte Spalte enthält die Werte, die für diese Implementierung verwendet wurden.

Bezeichnung	Länge	Beschreibung	Wert
Quell-Port	2 Byte	Port des senden Prozesses	0_{10}
Ziel-Port	2 Byte	Port des empfangenen Prozesses	1234_{10}
Länge	2 Byte	Länge des gesamten Paketes mit Header	nicht konstant
Prüfsumme	2 Byte	Einer-Kompliment über 16-Bit-Wort-Summe	0000_{16}

Tabelle 5.2.: Eigenschaftsfelder des UDP-Headers [vgl. MBW10, S. 228-229]

Der Quell-Port ist dabei immer 0, da keine Kommunikation von PC zu FPGA statt findet. Die Länge des UDP-Paketes muss jedes mal auf dem FPGA neu berechnet werden, da das Ende eines Bildes nicht die maximale Länge eines Ethernet-Rahmens ausnutzt und somit ist das UDP-Paket kürzer als die anderen Pakete. Die Prüfsumme ist optional. Um Hardware zu sparen wird sie nicht berechnet. Damit die Prüfsumme beim Empfänger nicht überprüft wird, muss sie den Wert 0 aufweisen. Andernfalls würde das Paket bei nicht übereinstimmen verworfen werden. [vgl. MBW10, S. 228-229]

Um das Einfügen des Headers zu ermöglichen, werden Schieberegister verwendet. Zu Beginn jeder Übertragung eines Paketes werden die Register mit den Header-Daten geladen. Anschließend werden die Bilddaten durch diese Register geschoben. Die Schieberegister werden aus mehreren Teilen zusammen gesetzt. Es beginnt mit einem Schieberegister, das eine konfigurierbare Länge hat, aber nur ein Bit breit ist. Dieses Schieberegister ist unter 5.7 zu finden.

```
1  shiftRegister: process (clk) is
2  begin
3    if (clk'event and clk='1') then
4      if (load='1') then
5        register_cs           <= loadIn;
6      elsif (enable='1') then
7        register_cs(0 to LENGTH-2)   <= register_cs(1 to
            LENGTH-1);
8        register_cs(LENGTH-1)    <= dataIn;
9      end if;
10   end if;
11 end process shiftRegister;
12
13 dataOut <= register_cs(0);
```

Listing 5.7: Serielles Schieberegister das für die Netzwerk-Header verwendet wird [Wis13b]

Wenn das Signal “load” den Wert 1 hat, werden die Daten in Zeile 5 aus “loadIn” ins Schieberegister geladen. Deshalb liegt an “loadIn” die UDP-Header-Informationen an. Wenn das Signal “enable” den Wert 1 hat, werden die Daten in Zeile 7 und 8 bei jedem Takt um ein Bit weiter von “dataIn” nach “dataOut” geschoben. An “dataIn” liegen die Bilddaten an. Die Länge des Schieberegister kann statisch durch die Konstante “LENGTH” festgelegt werden.

Die Daten sollen, um einen höheren Durchsatz zu erzielen, nicht seriell sondern parallel durch das Schieberegister geschoben werden. Deshalb werden immer acht serielle Schieberegister kombiniert. Dies ist unter Listing 5.8 zu erkennen.

```
parallelShiftRegister: for i in 0 to WIDTH-1 generate
  signal singleLoadIn_s : std_logic_vector(0 to LENGTH-1);
begin
  -- Map the load data to the right serial shift register
  loadDataMapper: for k in 0 to LENGTH-1 generate
  begin
    singleLoadIn_s(k) <= loadIn(k * WIDTH + i);
  end generate loadDataMapper;

  -- Instanciate a serial shift register
  shiftReg_i : entity work.shiftReg
  generic map (
    LENGTH    => LENGTH
  )
  port map (
    clk     => clk,
    enable    => enable,
    dataIn    => dataIn(i),
    load    => load,
    loadIn    => singleLoadIn_s,
    dataOut   => dataOut(i)
  );
end generate parallelShiftRegister;
```

Listing 5.8: Paralleles Schieberegister das für die Netzwerk-Header verwendet wird [Wis13b]

Die Anzahl der Instanzen des seriellen Schieberegisters wird durch die Konstante “WIDTH” festgelegt. Da dieser Konstante der Wert acht zugewiesen wird, ist der Eingang “dataIn” acht Bit breit. Das bedeutet, dass immer ein ganzes Byte pro Takt von “dataIn” nach “dataOut” geschoben wird. Das Signal “enable” wird vor jedem Senden für einen Takt auf 1 gelegt. Somit

wird der komplette Header der an "loadIn" anliegt in die Register geladen. Danach wird dieser Byte für Byte aus "dataOut" geschoben.

Der Header enthält unter anderem die variierende Paketgröße. Das Berechnen dieser Größe und Zusammensetzen des Headers wird in Listing 5.9 dargestellt.

```
packageLengthCalc: process (dataLen) is
  variable payloadLength_v  : std_logic_vector(dataLen'RANGE);
begin
  if (dataLen > MAX_PAYLOAD_LENGTH) then
    payloadLength_v   := MAX_PAYLOAD_LENGTH;
  else
    payloadLength_v   := dataLen;
  end if;

  payloadLength_s   <= payloadLength_v;
end process packageLengthCalc;


packageLength_ns  <= HEADER_LENGTH + payloadLength_s;

-- Build a valid udp header which can be loaded into the shift
    register
udpHeaderData_s   <= SRC_PORT & DEST_PORT & packageLength_ns &
    CHECK_SUM;
```

Listing 5.9: Zusammensetzen des UDP-Headers [Wis13b]

Wie schon in Kapitel 5.2 beschrieben, existiert eine maximale Paketlänge für das Übertragen im Ethernet, die durch die MTU definiert wird. Das Einhalten dieser Maximalgröße wird durch die Zeilen 4 bis 8 garantiert. Hierbei wird die Nutzdatenlänge "payloadLength_s" mit der Datenlänge "dataLen", die vom Bilddaten-Header kommt oder mit der maximalen Nutzdatenlänge beschrieben. Die Berechnung der UDP-Paketgröße wird in Zeile 14 durch das Addieren der Nutzdatenlänge und der UDP-Header-Größe in Byte umgesetzt. Schließlich wird der UDP-Header in Zeile 17 durch das Konkatenieren von Bitvektoren zusammengesetzt. Der Bitvektor "udpHeaderData_s" ist dann mit den Schieberegistern verbunden und wird beim Laden übernommen.

Bei der Verwendung von Schieberegistern besteht, wie in Kapitel 5.5 schon beschrieben,

das Problem des Leerens der Register. Da die Paketgröße aber schon vor Beginn des Versendens feststeht, ist dies weniger problematisch. Der Vorteil von Registern im Gegensatz zu Multiplexern betrifft den kritischen Pfad. Wenn die komplette Pipeline aus Multiplexern bestehen würde, müssten zwischendurch wieder Register verwendet werden, um den kritischen Pfad zu verkleinern. Die für das Leeren der Register nötige Logik ist unter Listing 5.10 aufgelistet.

```
packageSplitter: process (payloadCounter_cs, nextByte) is
  variable payloadCounter_v : std_logic_vector(
      payloadLength_s'RANGE );
  variable nextByteReq_v    : std_logic;
begin
  payloadCounter_v  := payloadCounter_cs;
  nextByteReq_v    := '0';

  -- through put the nextByte request until the end of the
      package was reached
  if (nextByte = '1' and payloadCounter_v /= x"0000") then
    payloadCounter_v  := payloadCounter_v - 1;
    nextByteReq_v    := '1';
  end if;

  payloadCounter_ns <= payloadCounter_v;
  nextByteReq      <= nextByteReq_v;
end process packageSplitter;



packageSplitterReg: process (clk) is
begin
  if (clk'event and clk = '1') then
    if (startTransIn = '1') then
      payloadCounter_cs <= payloadLength_s;
    else
      payloadCounter_cs <= payloadCounter_ns;
    end if;
  end if;
end process packageSplitterReg;
```

Listing 5.10: Leeren des UDP-Schieberegisters [Wis13b]

Wenn die Übertragung eines neuen Paketes beginnt, wird die Nutzdatenlänge "payload-Length_s" in Zeile 24 in einen Zähler geladen. Dieser Zähler wird in Zeile 10 bei jedem Weiterschieben der Register dekrementiert. In dieser Zeit werden in Zeile 11 mit dem Signal "nextByteReq" weitere Bilddaten angefordert. Sobald der Zähler den Wert 0 in Zeile 9 erreicht hat, werden keine Anforderung auf weiter Daten an das Schieberegister für den Bilddaten-Header weitergegeben. Somit können die Bilddaten aus allen Registern noch versendet werden. Erst wenn eine neue Übertragung durch das Signal "startTransIn" in Zeile 23 gestartet wird, werden wieder Bilddaten angefordert.

5.7. IP-Header

Das Internet-Protokoll kann im TCP/IP-Referenzmodell in der Internet-Schicht eingeordnet werden. Das bedeutet, dass es für eine Ende-zu-Ende-Kommunikation mit Zwischenknoten zuständig ist. [vgl. MBW10, S. 4] Also ist es mit dem Internet Protokoll möglich, über zum Beispiel Router-Grenzen hinweg adressieren zu können. Das hier verwendete Protokoll hat die Version 4. Der Vorteil dieser Variante ist die hohe Kompatibilität mit anderen Geräten. Um Geräte zu identifizieren, wird eine 32 Bit lange Adresse verwendet. [vgl. MBW10, S. 105]

Die Felder dieses Headers und deren Werte für diese Lösung sind in Tabelle 5.3 aufgelistet.

Bezeichnung	Länge	Beschreibung	Wert
Version	4 Bit	Versionsnummer	4_{16}
Header-Länge	4 Bit	Länge des IP-Headers	5_{16}
Type of Service	8 Bit	Priorität des Paketes	00_{16}
Paketlänge	16 Bit	Länge des gesamten Paketes	nicht konstant
Identifikation	16 Bit	Für Fragmentierung verwendet	0000_{16}
Flags	3 Bit	Kontrolle der Fragmentierung	0_{16}
Fragment Offset	13 Bit	Wiederherstellung der Fragmentreihenfolge	00_{16}
Time to Live	8 Bit	Maximale Anzahl an Router	01_{16}
Protokoll	8 Bit	Identifikation des enthaltenden Protokolls	11_{16}
Prüfsumme	16 Bit	Prüfsumme über IP-Header-Daten	nicht konstant
Quell-Adresse	32 Bit	IP-Adresse des Senders	192.168.0.5
Ziel-Adresse	32 Bit	IP-Adresse des Empfängers	224.0.0.1

Tabelle 5.3.: Eigenschaftsfelder des IP-Headers [vgl. MBW10, S. 119-121]

Die Länge des Headers wird in 4-Byte-Worten angegeben. Somit hat der verwendete Header

eine Länge von 20 Byte. Nützlich ist diese Angabe für die Verwendung optionaler Eigenschaftsfelder. Mit Hilfe des Type-of-Service können Prioritäten und Übertragungsverhalten festgelegt werden. Da diese Eigenschaften aber nur von Routern beachtet werden und bei dieser Verwendung Sender und Empfänger direkt über ein Crossover-Ethernet-Kabel verbunden sind, müssen sie nicht weiter beachtete werden. Laut RFC 791 bedeutet der gewählte Wert die Verwendung der Standardpriorität. [vgl. Pos81, S. 12] Die Paketlänge gibt die Größe des gesamten Paketes inklusive Headers in Byte an. Da unterschiedlich Lange Pakete versendet werden, beinhaltet dieses Feld keine Konstante. Der Wert muss vor jedem Senden eines neuen Paketes neu bestimmt werden Die Eigenschaften Identifikation, Flags und Fragment-Offset sind für eine mögliche Fragmentierung des Paketes reserviert. Dies kann auftreten, wenn Pakete über Router-Grenzen mit unterschiedlicher MTU versendet werden. Mit dem Time-to-Live Wert kann angegeben werden über wie viele Router-Grenzen ein Paket versendet werden darf. Jeder Router über den das Paket weiter vermittelt wird. dekrementiert diesen Wert. Wenn das Feld den Wert 0 erreicht hat, wird das Paket verworfen. Dieses Feld wird genutzt, um zu vermeiden, dass ein Paket eine unendlich lange Lebensdauer im Internet hat. Hier wird dieses Feld mit dem Wert 1 belegt, da hinter einem Router die Reihenfolge der Pakete nicht mehr garantiert werden kann und somit ein fehlersicheres Zusammensetzen der Bilder nicht mehr garantiert wird. Mit dem Feld Protokoll wird das in den Nutzdaten enthaltende Protokoll benannt. In diesem Fall hat es den Wert 11_{16}, der für ein UDP-Protokoll steht. Die Prüfsumme bezieht sich nur auf den IP-Header. Da die Länge des Paketes nicht konstant ist, muss diese Summe auch bei jedem Versenden neu berechnet werden. [vgl. MBW10, S. 119-121]

Die Quell-IP-Adresse ist eine private Klasse-C Adresse. Der Vorteil von privaten Adressen ist, dass sie im Internet nicht verwendet werden und von einem Router nicht weitergeleitet werden. Somit können keine externen Dienste mit der internen Vergabe von privaten Adressen überdeckt werden. [vgl. MBW10, S. 106-109]

Die Ziel-IP-Adresse ist eine Multicast-Adresse. Der entscheidende Vorteil von Multicast ist die nicht benötigte Adressauflösung. Bei der Verwendung von Unicast-Adressen muss vor dem Versenden die passende MAC-Adresse zu der Ziel-IP-Adresse gefunden werden. Dies kann mit Hilfe vom Address Resolution Protocol (ARP) durchgeführt werden. Dabei wird aber eine bidirektionale Kommunikation benötigt, die den Hardwareaufwand um einiges komplexer machen würde. Für Multicast wird keine bidirektionale Kommunikation benötigt, da, wie im Kapitel 5.8 beschrieben wird, die MAC-Adresse aus der Multicast-IP-Adresse berechnet werden kann. [vgl. MBW10, S. 148-149]

Für das Einfügen des IP-Headers wird eine weitere Instanz, des in Listing 5.8 vorgestellten Schieberegisters, verwendet. Also wird auch dieser Header vor jedem Start einer Übertragung in die Register geladen. Unter Listing 5.11 ist die Berechnung der Prüfsumme und das Zusammensetzen des Headers zu sehen.

```
packageLength_s      <= HEADER_LENGTH + dataLen;



-- Calculate the positiv value of the header check sum (the
   ceck sum is not a part of the calulation)
-- package length is the only not constant value in this
   calulation (so hopfully synthese will optimise it)
longHeaderCheckSum_s   <=  (x"0" & ipHeaderData_s(0 to 15)) +
  (x"0" & ipHeaderData_s(32 to 47)) +
  (x"0" & ipHeaderData_s(48 to 63)) +
  (x"0" & ipHeaderData_s(64 to 79)) +
  (x"0" & ipHeaderData_s(96 to 111)) +
  (x"0" & ipHeaderData_s(112 to 127)) +
  (x"0" & ipHeaderData_s(128 to 143)) +
  (x"0" & ipHeaderData_s(144 to 159)) +
  (x"0" & packageLength_s);

-- Add the 4 carry bits to the sum and invert it
headerCheckSum_s     <= not( longHeaderCheckSum_s(4 to 19) +
   longHeaderCheckSum_s(0 to 3) );



-- Build a valid ip header which can be loaded into the shift
   register
ipHeaderData_s       <= VERSION & IP_HEADER_LENGTH &
   TYPE_OF_SERVICE & packageLength_s &
  IDENTIFICATION & FLAGS & FRAGMENT_OFFSET & TIME_TO_LIVE &
     PROTOCOL & headerCheckSum_s & SRC_ADDRESS & DEST_ADDRESS;
```

Listing 5.11: IP-Header berechnen [Wis13b]

In der ersten Zeile wird die Paketlänge des IP-Pakets berechnet. Diese setzt sich aus der UDP-Paketgröße und der Länge des IP-Headers zusammen. Nachfolgend wird die benötigte

Prüfsumme für den IP-Header berechnet. Im ersten Schritt wird der komplette Header in 16 Bit große Felder unterteilt. Diese werden alle aufsummiert mit Ausnahme der Prüfsumme an sich. Die dabei entstehenden 4 Übertragsbits werden in Zeile 18 mit dazu addiert. Des Weiteren wird in Zeile 18 auch das Kompliment dieser Summe berechnet. [vgl. Pos81, S. 14] In den Zeilen 23 und 24 wird dann schlussendlich der Header aus seinen Einzelteilen zusammen gesetzt. Dieser Vektor wird beim Beginn einer neuen Übertragung in die Register geladen.

5.8. Ethernet-Header

Das letzte benötigte Protokoll ist das Ethernet-Protokoll. Dieses Protokoll kann in der Sicherungsschicht eingeordnet werden. Damit ist es für die Ende-zu-Ende-Kommunikation ohne Zwischenknoten zuständig. [vgl. MBW10, S. 4] Um die benötigte Bandbreite zur Verfügung zu stellen, wurde sich hierbei für 100Base-TX, auch Fast Ethernet genannt, entschieden. Diese Kommunikationsstrecke ermöglicht Übertragungsraten von bis zu 100 MBit pro Sekunde. [vgl. MBW10, S. 61]

Der Header für dieses Protokoll umfasst die in Tabelle 5.4 beschriebenen Felder. Dabei werden die in der letzten Spalte gezeigten Werte verwendet. Die Präambel benötigt der Empfänger,

Bezeichnung	Länge	Beschreibung	Wert
Präambel	7 Byte	Synchronisation des Empfängers	$AAAA..AAAA_{16}$
Start Frame Delimiter	1 Byte	Beginn des Paketes	AB_{16}
Ziel-Adresse	6 Byte	MAC-Adresse des Empfängers	01-00-5E-00-00-01
Quell-Adresse	6 Byte	MAC-Adresse des Senders	00-00-5E-00-FA-CE
Pakettyp	2 Byte	Identifiziert enthaltenes Paket	0800_{16}

Tabelle 5.4.: Eigenschaftsfelder des Ethernet-Headers [vgl. MBW10, S. 62-63]

um sich auf den Takt des Senders aufsynchronisieren zu können. Anhand des Start-Frame-Delimiter kann der Empfänger erkennen, dass ein Paket beginnt. Diese zwei Felder müssen nicht von der in dieser Arbeit entwickelten Hardware gesendet werden, da dies die von Xilinx bereitgestellte Hardware "XPS Ethernet Lite MAC" übernimmt. Die Ziel-MAC-Adresse ist eine Multicast-Adresse. Diese lässt sich aus der Ziel-IP-Adresse berechnen. [vgl. Dee89, S. 6] Die Quell-MAC-Adresse wurde vom "XPS Ethernet Lite MAC", welcher von Xilinx zur Verfügung gestellt wird, übernommen. Der Pakettyp enthält die ID für ein nachfolgendes Internet Protokoll der Version 4. [vgl. MBW10, S. 150]

Auch für das Einfügen dieses Headers wird eine weitere Instanz des in Listing 5.8 gezeigten Schieberegisters verwendet. Die einzige zusätzliche Aufgabe, die von der Ethernet-Entity bewältigt werden muss, ist die Berechnung der Paketlänge. Diese setzt sich aus der IP-Paketlänge und der Ethernet-Header-Größe zusammen.

Der Footer eines vollständigen Ethernet-Paketes enthält noch optionale Padding-Bytes und eine CRC-Prüfsumme. Aber auch diese Felder werden, soweit benötigt, vom "XPS Ethernet Lite MAC" angehängt.

5.9. XPS Ethernet Lite MAC Wrapper

Im voran gegangenen Kapitel 5.8 wurde die "XPS Ethernet Lite MAC"-Hardware schon erwähnt. Diese wurde von Xilinx entwickelt, um eine Möglichkeit zu schaffen, mit einem Prozessor über das Netzwerk kommunizieren zu können. Dafür besitzt diese Hardware auf der einen Seite eine Processor-Local-Bus-Slave-Schnittstelle (PLB), die von einem Prozessor mit PLB-Master angesprochen werden kann. Auf der anderen Seite wird ein Media Independent Interface (MII) zur Verfügung gestellt. Das MII ist ein Standard der von externen Physical Layer (PHY) Transceivern unterstützt wird. [vgl. Xil10, S. 1]

Der Vorteil dieses externen Bauteils liegt darin, dass dieser eine 4 Bit breite Anbindung zum FPGA besitzt und somit schon mit einer geringen Taktfrequenz von 50 MHz innerhalb des FPGAs ermöglicht wird, über Fast Ethernet mit 100 MBit pro Sekunde kommunizieren zu können. [vgl. Xil10, S. 29] Diese Schnittstelle bieten aber auch andere Hardware-Implementierungen, die nicht von Xilinx stammen, an.

Ein großer Anbieter für Open-Source-Hardware ist OpenCores.org. Auf dieser Seite sind viele in VHDL oder Verilog implementierte Hardware-Komponenten zu finden. So lässt sich dort auch eine Implementierung eines vollständigen UDP/IP-Stacks mit dem Namen UDP/IP Core finden. Diese Variante bietet eine 8 Bit breite Anbindung für das Übermitteln der Nutzdaten. Dies ist von sehr großem Vorteil, da keine zusätzlichen Komponenten wie ein Bus-Interface benötigt werden. Diese zusätzlich benötigten Komponenten bei der Verwendung eines Bussystems erzeugen eine Erhöhung der Latenzen beim Versenden von Daten, die vermeidbar ist. Bei weiteren Untersuchungen viel aber ein KO-Kriterium auf. Die Hardware benötigt einen Takt von 125 MHz. Da die Kamera, wie in Kapitel 3 beschrieben, mit circa $\frac{1}{4}$ dieses Taktes arbeitet und die Ethernet-Kommunikation Dank der Parallelisierung der Daten

durch den PHY auch nur einen Takt von 50 MHz benötigt, wäre dies ein unnötig hoher Takt. Um diese hohe Frequenz ermöglichen zu können, würden die Anforderungen an den verwendeten FPGA steigen und somit würden auch die Kosten für solch eine Lösung wachsen. [ABS10]

Eine andere Variante ist der EthMAC. Dieser implementiert nur die Sicherungsschicht. Die anderen Schichten könnten durch extern oder eigen entwickelte Komponenten nachgebildet werden. Problematisch ist hierbei die Verwendung des Wishbone-Bussystem. Die Vorteile eines Bussystems würden in dieser Arbeit nicht ausgenutzt werden und somit würde nur eine störende Erhöhung der Latenz entstehen. [MK02]

Die "XPS Ethernet Lite MAC"-Hardware wird zwar normalerweise auch über ein Bussystem angesprochen, aber auf Grund des gut verständlichen Programmierstils konnte schnell eine Lösung zum Umgehen des Bussystems gefunden werden. Der große Vorteil der Verwendung dieser Hardware ist zum Einen das schnelle Versenden von Daten, das durch die Anpassung ermöglicht wurde und zum Anderen die Möglichkeit durch nur kleine Änderungen parallel die Anbindung über den PLB weiterhin nutzten zu können. Die Datenkommunikation über den PLB wäre zum Beispiel angebracht, um mit Hilfe eines MicroBlaze's auf dem FPGA Parameter vom PC aus über Ethernet konfigurieren zu können.

Zu Testzwecken dieser von Xilinx bereitgestellten Hardware wurde erst ein Beispielprojekt mit einem MicroBlaze und der 'XPS Ethernet Lite MAC"-Hardware erstellt. Dabei fiel in der Beschreibung für dieses Beispiel ein großes Problem auf. Mit diesem Projekt sind nur Übertragungsraten von 10 Mb/s möglich. [vgl. Dig11, S. 5] Dies wäre, wie in der Voruntersuchung in Kapitel 2.3 begründet, nicht schnell genug für die Übertragung der Bilddaten. Der Auslöser für dieses Problem war ein nicht beschalteter Pin des PHY-ICs. Der Pin "TXERR", der am FPGA mit dem Pin "P2" verbunden ist, muss mit Ground verbunden sein. [F.13] Somit konnten nun auch 100 Mb/s übertragen werden und es konnte mit dem Umbau des MAC begonnen werden.

Die Schnittstelle, die nach dem Umbau des "XPS Ethernet Lite MAC" zu Verfügung steht, ist in Listing 5.12 aufgeführt.

```
1  -- ports for an external ethernet transmitter
2  EXT_TX_PACKAGE_LEN : in std_logic_vector(0 to 15);
3  EXT_TX_START       : in std_logic;
4  EXT_TX_DATA        : in std_logic_vector(0 to 3);
```

```
5   EXT_TX_ENABLE         : out std_logic;
6   EXT_TX_DONE           : out std_logic
```

Listing 5.12: Hinzugefügte Schnittstelle für “XPS Ethernet Lite MAC” [Wis13b]

Die Paketlänge wird benötigt, damit die Xilinx-Hardware feststellen kann, wie viele Bytes versendet werden sollen und wann der Ethernet-Footer angehängt werden soll. Mit dem Signal “EXT_TX_START” übermittelt die angeschlossene Hardware dem Ethernet-Lite-MAC, dass ein neues Paket versendet werden kann. Über den Vektor “EXT_TX_DATA” werden die zu senden Daten ausgetauscht. Dabei muss nach jeder steigenden Flanke des Signals “EXT_TX_ENABLE” ein weiteres Nibble des Paketes anliegen. Durch einen High-Pegel auf dem Signal “EXT_TX_DONE” signalisiert der Ethernet-Lite-MAC, dass er gerade nicht mit dem Versenden von Paketen beschäftigt ist und bereit für die Annahme eines neuen Sendeauftrages ist.

Nun besteht das Problem, dass die Schieberegister für das Einfügen der Header eine acht Bit breite Pipeline verwenden, aber die Schnittstelle des “XPS Ethernet Lite MAC” nur eine Breite von vier Bit besitzt. Die komplette Pipeline auf eine Breite von vier Bit zu begrenzen wäre auch nicht praktikabel, da die Paketlängenangabe Byte-orientiert ist und somit alle Paketlängen umgerechnet und Datenlängen-Zähler angepasst werden müssten. Um dies zu umgehen, wurde die unter Listing 5.13 eingefügte Logik entwickelt.

```
1   nibbleMux: process (highNibble_cs, mac2ethData_s) is
2   begin
3     if (highNibble_cs = '1') then
4       txDataOut <= mac2ethData_s(0 to 3);
5     else
6       txDataOut <= mac2ethData_s(4 to 7);
7     end if;
8   end process nibbleMux;
9
10
11
12  nextByteGen: process (nextNibble, highNibble_cs) is
13  begin
14    if (nextNibble = '1') then
15      highNibble_ns    <= not highNibble_cs;
16      -- get next byte if the last nibble was the high nibble
17      nextByteReq_ns    <= highNibble_cs;
18    else
```

```
19        highNibble_ns    <= highNibble_cs;
20        nextByteReq_ns     <= '0';
21      end if;
22    end process nextByteGen;
```

Listing 5.13: Wrapper zwischen Schieberegistern und “XPS Ethernet Lite MAC” [Wis13b]

Der Prozess aus den Zeilen 1 bis 8 beschreibt einen Multiplexer, der zwischen dem oberen und unteren Nibble aus den Schieberegistern umschaltet. Dabei ist der Ausgangsvektor “txDataOut” direkt mit dem Eingang “EXT_TX_DATA” des MAC verbunden. Die Ansteuerung dieses Multiplexers wird vom zweiten Prozess übernommen. Dieser reagiert auf das Signal “nextNibble”, welches mit dem Signal “EXT_TX_ENABLE” des MAC verbunden ist. Somit wechselt “highNibble_cs” in Zeile 15 bei jeder Anforderung eines weiteren Nibbles zwischen High- und Low-Pegel. Das bedeutet wieder herum, dass mit jeder Anforderung zwischen oberen und unterem Nibble umgeschaltet wird. Das Signal “nextByteReq_ns” wird in Zeile 17 auf 1 gesetzt, wenn als letztes das obere Nibble gelesen wurde. Daraus folgt, dass nach jedem Lesen des oberen Nibbles die Schieberegister für die Header um eins weiter geschoben werden.

Damit der Ethernet-Lite-MAC das Ende dieses Datenstroms erkennen kann, wird ihm die Gesamtpaketlänge übermittelt. Die Berechnung dieses Wertes erstreckt sich über alle Module, die Header-Informationen in den Datenstrom einfügen wollen. Die Module für den UDP-, IP- und Ethernet-Header müssen dabei nur ihre feste Header-Länge mit aufaddieren und diesen Wert an das nachfolgende Modul weitergeben. Das Modul für den Bilddaten-Header muss zuvor entscheiden, ob ein Header eingefügt werden soll und abhängig davon dann die Länge aufaddieren. In Abbildung 5.3 finden sich die einzelnen Schritte für die Berechnung der Paketlänge wieder. Mit dem Register “dataLengthReg” wird der aktuelle Zählerstand des Bildgrenzen-

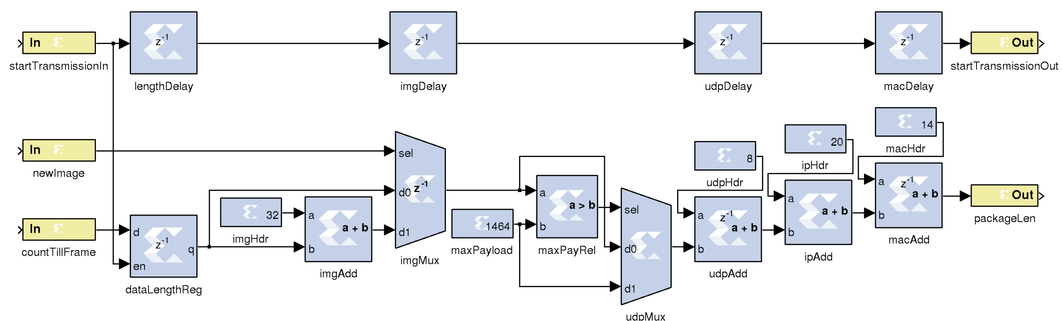

Abbildung 5.3.: Übersicht über die Berechnung der Paketlänge [Wis13b]

Zählers eingefroren, wenn ein neues Paket versendet werden soll. Das variable Anhängen

der Bilddaten-Header-Größe wird von den Elemente “imgAdd” und “imgMux” übernommen. Dabei durchläuft das Ergebnis ein Register, um den kritischen Pfad zu verkleinern. Die in Kapitel 5.6 beschriebene Begrenzung der Paketgröße wird von den Komponenten “maxPayRel” und “udpMux” umgesetzt. Der UDP-Header wird von “udpAdd” aufaddiert. Dieses Ergebnis durchläuft ein weiteres Register. Der IP-Header wird von “ipAdd” und der MAC-Header von “macAdd” aufaddiert. Zum Ende durchläuft dieses Ergebnis zum dritten Mal ein Register und wird dann an den Ethernet-Lite-MAC übergeben.

6. Bilddaten-Empfänger

Um die Bilddaten verwenden zu können, muss PC-seitig ein Stück Software existieren, die den Netzwerkdatenstrom einließt und aufbereitet. Neben der Umwandlung und Abspeicherung der Bilddaten in ein für OpenCV verständliches Format, spielt die Synchronisation, um die Bildgrenzen einzuhalten, auch eine große Rolle. Das Bild soll schließlich nicht nach unten oder oben verschoben sein. Um die Daten einlesen zu können, muss erst einmal die Datenquelle geöffnet werden.

6.1. Stream öffnen

Da die Daten über die Netzwerk-Schnittstelle empfangen werden, bieten sich zum Öffnen dieses Streams Sockets an. Diese verhalten sich ähnlich wie das Öffnen von Dateien. Nach dem Öffnen kann mit Funktionen zum Lesen und Schreiben auf diesen Stream zugegriffen werden.

Die genaue Abfolge der verwendeten Funktionsaufrufe wird in Listing 6.1 gezeigt.

```
1  const int StreamReader::openUDPListener(const std::string
       mCastAddress, const uint16_t port) const
2  {
3    const int socketHandler = socket( AF_INET, SOCK_DGRAM,
         IPPROTO_UDP );
4    if (socketHandler < 0) {
5      throw SocketNotOpenedException();
6    }
7
8    struct ip_mreq mc_req;
9    bzero( &mc_req, sizeof(struct ip_mreq) );
10   if (  inet_pton( AF_INET, mCastAddress.c_str(),
         &(mc_req.imr_multiaddr.s_addr) ) != 1  ) {
11     throw UnableToConvertMCastAddressException();
12   }
13   mc_req.imr_interface.s_addr = htonl(INADDR_ANY);
```

```
14
15    if ( ( setsockopt ( socketHandler , IPPROTO_IP ,
         IP_ADD_MEMBERSHIP , &mc_req , sizeof(mc_req))) < 0) {
16      LOG4CXX_WARN(logger , "::openUDPListener␣The␣stream␣reader␣
           could␣not␣join␣to␣the␣multicast␣address␣" << mCastAddress
           <<
17          "!␣Udp␣packages␣of␣the␣same␣hop␣will␣only␣be␣received.");
18    }
19
20    struct sockaddr_in mcast_Addr;
21    bzero ( &mcast_Addr , sizeof(mcast_Addr) );
22    mcast_Addr.sin_family = AF_INET;
23    mcast_Addr.sin_addr.s_addr = htonl(INADDR_ANY);
24    mcast_Addr.sin_port = htons ( port );
25
26    // bind to specified port on any interface
27    if (  bind ( socketHandler , (struct sockaddr*)&mcast_Addr ,
         sizeof(struct sockaddr_in) ) < 0  ) {
28      throw PortNotBoundException();
29    }
30
31    return socketHandler;
32  }
```

Listing 6.1: Öffnen des UDP-Multicast-Streams [Wis13a]

In Zeile 3 wird der Socket geöffnet. Dabei wird über die Parameter festgelegt, dass es sich um das Internet Protokoll der Version 4 handelt und die Daten in UDP-Paketen gekapselt sind. Falls das Öffnen des Sockets fehlschlägt, wird in Zeile 5 eine Ausnahmebehandlung ausgelöst.

Da die Multicast-Empfänger-Adresse als Zeichenkette vorliegt, damit sie vom Benutzer leicht konfiguriert werden kann, wird sie in ein für die Socket-Umgebung verständliches Format konvertiert. Dies wird in Zeile 10 durchgeführt. Falls dabei Probleme auftreten sollten, weil es sich zum Beispiel um keine gültige IPv4-Adresse handelt, wird in Zeile 11 eine Ausnahmebehandlung ausgelöst. In Zeile 15 wird diese umgewandelte Adresse für das Beitreten der Multicast-Gruppe verwendet. Wenn dies fehlschlägt, werden Pakete mit dieser Multicast-Adresse, die aus anderen Segmenten gesendet werden, nicht empfangen. Dies hat keine weiteren Auswirkungen, da sich der Sender im selben Segment befinden sollte. Mit dieser Lösung wäre, wie in Kapitel 5.6 schon behandelt, das Versenden aus einem anderen Segment

ehe ungünstig, da der Erhalt der Paketreihenfolge nicht garantiert werden kann.

Ab Zeile 20 wird eine Struktur erstellt, mit der die Quell-Adresse und der Quell-Port definiert wird. Dabei wird als Quell-Adresse in Zeile 23 eine beliebige festgelegt. Als Port wird in Zeile 24 der vom Benutzer konfigurierte Wert übernommen. Anschließend werden diese Einstellungen in Zeile 27 gesetzt. Wenn dies fehlschlagen sollte, wird in Zeile 28 eine Ausnahmebehandlung ausgelöst.

Mit Hilfe des Rückgabewerts, der in Zeile 31 zurückgegeben wird, kann mit Lese und Schreib-Funktionen auf den geöffneten Stream zugegriffen werden. Deshalb wird dieser Wert allen anderen Methoden, die vom Stream lesen wollen, übergeben.

Die erwähnten und alle folgenden Ausnahmebehandlungen werden an einer zentralen Stelle abgefangen und als aussagekräftige Einträge im Fehlerprotokoll gespeichert.

6.2. Synchronisation

Den Anfang eines binären Datenstroms sicher zu erkennen, gestaltet sich nicht immer ganz einfach. Das Problem ist dabei, das ein Pattern der in dem Strom gesucht werden soll und in diesem Fall den Beginn eines neuen Bildes markiert, auch in den binären Nutzdaten auftreten kann. Um diese Wahrscheinlichkeit möglichst gering zu halten, sollte eine lange und möglichst nicht in den Nutzdaten vorkommende Bit-Folge verwendet werden. Das in Kapitel 5.5 beschriebene Sun-Raster-Format besitzt mit seiner 4 Byte langen Magic-Number eine doppelt so langen Header-Beginn wie der auch untersuchte Microsoft Windows Bitmap Header. Die abwechselnde Kombination aus Einsen und Nullen in der Folge $59A66A95_{16}$ erhöht die Eindeutigkeit auch um ein weiteres Stück. Da die Bilddaten unkomprimiert übertragen werden, repräsentiert ein Bit die Farbinformation für einen Bildpunkt. Die verwendete Erosion bewirkt, dass einzelne gesetzte Bildpunkte ausgelöscht werden. Somit ist die Wahrscheinlichkeit, dass die gewählte Bit-Folge in den Nutzdaten auftritt verschwindend gering.

Um die Komplexität der Suche nach der Bit-Folge zu minimieren, beginnt ein neues Bild immer mit dem Beginn eines Paketes. Somit muss nach der Magic-Number nur zu Beginn eines neuen UDP-Paketes gesucht werden. Dies hat des Weiteren den positiven Nebeneffekt, dass die Magic-Number noch seltener in den Nutzdaten gefunden werden kann.

Unter Listing 6.2 wird der Algorithmus der zum Finden der Magic-Number verwendet wird gezeigt.

```
const size_t StreamReader::findSunRasterHeader(const int
    socketHandler, SUNRASTER* header) const
{
  bool wasResyced = false;
  size_t readBytes = 0;

  bool lookingForHeader = true;
  while (lookingForHeader)
  {
    readBytes = read(socketHandler, header, ETHERNET_MTU);
    if (readBytes >= sizeof(SUNRASTER))
    {
      lookingForHeader = false;
      for (unsigned int i=0; i<sizeof(SUN_RASTER_MAGIC_NUMBER);
          i++)
      {
        if (header->magicNumber[i] != SUN_RASTER_MAGIC_NUMBER[i])
        {
          wasResyced = true;
          lookingForHeader = true;
          break;
        }
      }
    }
    else
    {
      throw ReadHeaderFailsException();
    }
  }

  if (wasResyced)
    LOG4CXX_WARN(logger,"::findSunRasterHeader():␣Reader␣has␣to␣
        be␣resynchronized␣in␣stream");

  return readBytes;
}
```

Listing 6.2: Synchronisation der Bildgrenzen des Bilddatenstroms [Wis13a]

In Zeile 9 wird ein vollständiges UDP-Paket eingelesen. Falls kein Paket zur Verfügung steht, blockiert dieser Lese-Befehl. Die maximale Paketgröße, die auf einmal eingelesen wird und deshalb auch die Puffergröße von "header" ist, ist mit der Ethernet-MTU festgelegt. Die MTU gibt die maximale Nutzdaten-Menge eines Ethernet-Frames an und ist somit größer als die Nutzdaten-Menge eines maximal langen UDP-Paketes. Damit ist sichergestellt, dass es zu keinem Überlauf des Puffers kommen kann.

In Zeile 10 wird überprüft, ob mindestens ein Paket mit der Länge des Bilddaten-Headers eingelesen wurde. Wenn dies nicht der Fall ist, wird in Zeile 25 eine Ausnahmebehandlung ausgelöst. Dies kann nur in einem Fehlerfall auftreten, da der Sun-Raster-Header mit einer Länge von 32 Byte weit unter der maximalen Nutzdatenlänge eines UDP-Paketes im Fast-Ethernet liegt.

Falls diese Fehlerüberprüfung erfolgreich überstanden wurde, wird kontrolliert, ob der Anfang der eingelesenen Nutzdaten einen Sun-Raster-Header enthält. Dafür werden die vier Byte der Magic-Number des Headers mit den ersten vier Byte der eingelesenen Daten verglichen. Wenn eines dieser Bytes nicht übereinstimmt, wird das UDP-Paket verworfen. Durch Zeile 19 beginnt die Schleife dann von vorn und es wird ein neues UDP-Paket eingelesen.

Dieser Abbruch wird in der Variablen "wasResyced" in Zeile 17 gespeichert und anschließend ab Zeile 29 ausgewertet. Dort wird eine Warnung in das Fehlerprotokoll geschrieben. Anhand dieser Warnung kann festgestellt werden, ob sich der Empfänger erneut auf den Video-Stream aufsynchronisieren musste. Dabei ist aber zu beachten, dass eine Synchronisation und eine dadurch entstehende Warnmeldung beim Einschalten des Empfängers vollkommen normal ist.

Mit den beschriebenen Schritten wird maximal nur ein UDP-Paket eingelesen. Die hier verwendete Kodierung der Daten und die Auflösung des Bildes fordert aber mehr als nur ein UDP-Paket, um ein vollständiges Bild übertragen zu können. Deshalb müssen anschließend alle weiteren dazugehörenden Pakete auch eingelesen werden.

6.3. Einlesen der Daten

Um feststellen zu können, wie viele Bytes für ein vollständiges Bild fehlen, wird der Bilddaten-Header ausgewertet. Dieser enthält ein Feld, dass die Länge der Bilddaten angibt. Wenn dieses mit der Länge des Sun-Raster-Headers addiert wird, entsteht die Länge an Bytes, die für ein

Bild eingelesen werden muss.

Im Listing 6.3 ist der Code für das Berechnen der Bildlänge und das Einlesen der noch benötigten Bytes dargestellt.

```
{
  SUNRASTER* header =
      reinterpret_cast<SUNRASTER*>(encodedImageBuffer.ptr());
  size_t readBytes = findSunRasterHeader(socketHandler, header);

  const uint32_t imageLength = be32toh(header->length);
  const size_t imageSize = sizeof(SUNRASTER) + imageLength;
  // Grow the encoded image buffer if the buffer is to small
  if ( static_cast<size_t>(encodedImageBuffer.rows *
      encodedImageBuffer.cols) < imageSize )
  {
    cv::Mat grownMat( cv::Size(imageSize, 1), CV_8UC1 );
    memcpy(grownMat.ptr(), encodedImageBuffer.ptr(), readBytes);
    encodedImageBuffer = grownMat;
  }

  while (readBytes < imageSize)
  {
    uint8_t* offsetPointer = encodedImageBuffer.ptr() +
        readBytes;
    const size_t newReadBytes = read( socketHandler,
        offsetPointer, imageSize-readBytes );
    if (newReadBytes <= 0) {
      throw ReadImageFailsException();
    }

    readBytes += newReadBytes;
  }


  const DiffTimeCalc diffTime;

  const cv::Mat frame = useFastImageDecoder ?
      fastImageDecode(encodedImageBuffer, imageLength) :
      cv::imdecode(encodedImageBuffer, 1);
```

```
32
33   const unsigned long usTime = diffTime.getDiffTimeInUSec();
34   LOG4CXX_INFO(logger, "::getNextFrame():␣Image␣decoding␣needs␣"
        << usTime << "␣us.");
35
36   return frame;
37 }
```

Listing 6.3: Bildlänge berechnen und fehlende Daten einlesen [Wis13a]

In Zeile 4 wird die Methode zur Bildgrenzen-Synchronisation aufgerufen. Diese speichert das erste Paket des einzulesenden Bildes in dem Puffer "encodedImageBuffer". Durch die Ausführung der Synchronisation zu Beginn jedes Bildes können Fehler entdeckt werden, die durch fehlende oder zusätzliche Daten entstehen können.

Ab Zeile 6 wird die für den Puffer benötigte Größe berechnet. Diese setzt sich aus dem Sun-Raster-Header und den eigentlichen Bilddaten zusammen. Anschließend wird in Zeile 9 überprüft, ob der dafür angelegte Puffer genügend Platz zur Verfügung stellt. Falls dieser zu klein ist wird ein neuer Speicherplatz reserviert und der alte Puffer mit dem neuen größeren überschrieben. Da die Referenz auf den Puffer "encodedImageBuffer" ein Übergabeparameter der Methode "getNextFrame" ist, steht der neue größere Speicherbereich auch beim Einlesen des nächsten Bildes zur Verfügung. Somit muss der Speicherbereich maximal einmal nach dem Start vergrößert werden. Dies gilt nur, wenn sich die Größe des Bildes nicht ändert. Da die Daten aber unkomprimiert übertragen werden und die Auflösung die gleiche bleibt, ändert sich dies auch nicht.

Die restlichen benötigten Pakete werden in einer Schleife, die ab Zeile 16 beginnt, eingelesen. Diese weiteren Pakete werden in Zeile 19 solange an den Puffer angehängt, bis die für dieses Bild benötigte Anzahl an Bytes eingelesen wurde.

Um die eingelesenen Daten in OpenCV verwenden zu können, werden sie dekodiert. Der Aufruf einer Methode zum Dekodieren wird ab Zeile 30 durchgeführt. Im nächsten Kapitel werden die Vor- und Nachteile der zwei unterschiedlichen Algorithmen der Dekodierung erklärt.

6.4. Dekodierung

Die empfangenden Bilddaten liegen zwar in einem unkomprimierten Format vor, sind aber trotzdem mit vielen OpenCV-Methoden, die den Datencontainer "cv::Mat" als Übergabeparameter erwarten, nicht kompatibel. Ein Problem ist der vorangestellte Header. Im OpenCV-Datencontainer werden Eigenschaften eines Bildes als Attribute dieses Objektes abgespeichert und somit nicht direkt in dem Puffer für die Bilddaten abgelegt. Dabei sind dies Informationen über die Größe und Farbtiefe des Bildes. Des Weiteren wird bei den empfangenden Daten ein Bildpunkt durch ein Bit dargestellt. OpenCV erwartet aber mindestens acht Bit für die Darstellung eines Bildpunktes. [Ope13a] Also ist eine Umwandelung der Daten notwendig.

Diese Umwandelung kann mit der OpenCV-eigenen Methode "cv::imdecode" durchgeführt werden. Es wird neben anderen Bildformaten auch das Sun-Raster-Format unterstützt. [Ope13b] Somit ist diese Art der Dekodierung sehr flexibel. Unter anderem ist es damit auch möglich, komprimierte Bilddaten im Sun-Raster-Format einzulesen. Dies wäre ein erster Schritt, um die Daten auch über langsamere Medien übertragen zu können.

Diese Flexibilität in Bezug auf die unterstützten Formate bringt aber auch einen großen Nachteil mit sich. Die Verarbeitung kann für den benötigten Fall nicht sehr gut optimiert werden. Deshalb wird eine Eigenentwicklung verwendet, die nur das Dekodieren eines Sun-Raster-Bildes, bei dem ein Bildpunkt durch ein Bit dargestellt wird, unterstützt. Durch diese Einschränkungen konnte diese Methode stark optimiert werden. Performance-Messungen für beide Varianten sind in Kapitel 7.3.1 zu finden.

Unter Listing 6.4 ist diese Eigenentwicklung dargestellt. Dabei wurde, um hoch Performance zu erreichen, auf C-typische Zeiger-Arithmetik gesetzt.

```
1  const SUNRASTER* header = reinterpret_cast<const
      SUNRASTER*>(encodedImageBuffer.ptr());
2
3  const uint32_t imageWidth = be32toh(header->width);
4  const uint32_t imageHeight = be32toh(header->height);
5
6  cv::Mat frame(imageHeight, imageWidth, CV_8UC1);
7
8  uint8_t* inPointer = const_cast<uint8_t*>(header->image);
9  const uint8_t* endInPointer = inPointer + imageLength;
10 uint8_t* outPointer = static_cast<uint8_t*>(frame.ptr());
```

```
11   while (inPointer < endInPointer)
12   {
13     uint8_t pixelData = *inPointer;
14
15     const uint8_t* endOutPointer = outPointer + 8;
16     while (outPointer < endOutPointer)
17     {
18       *outPointer = (pixelData & 0x01) ? 0xFF : 0x00;
19       pixelData >>= 1;
20       outPointer++;
21     }
22
23     inPointer++;
24   }
25
26   return frame;
27 }
```

Listing 6.4: Schnelle Sun-Raster-Bild-Dekodierung [Wis13a]

Um das Bild umwandeln zu können, wird ein neuer Speicherbereich benötigt, in dem das Ergebnis abgelegt werden kann. Dieser wird in Zeile 7 angelegt. Dabei wird auf den OpenCV-eigenen Datencontainer "cv::Mat" gesetzt. Somit bleibt die optimierte Implementierung in Bezug auf den Rückgabewert kompatibel zur OpenCV-Variante. Dieser Container speichert die Eigenschaften des Enthalten Bildes in seinen Attributen. Deshalb werden Informationen wie Breite, Höhe und Farbtiefe bei der Initialisierung mit übergeben. Um Bilder unterschiedlicher Größe umwandeln zu können, wird die Auflösung des Bildes in Zeile 4 und 5 aus dem zugehörigen Sun-Raster-Header gelesen.

Ab Zeile 9 beginnt die eigentliche Umwandelung. Der Zeiger "inPointer" wird für das Einlesen verwendet und zeigt immer auf das Byte, dass die umzuwandelnden Bildpunkte enthält. Der Zeiger "outPointer" wird für das Abspeichern verwendet und zeigt auf die Byte-Position an der der nächste Bildpunkt gespeichert werden soll.
Die Schleife, die in Zeile 12 beginnt, wird solange durchlaufen bis alle Bildpunkte des empfangen Bildes verarbeitet wurden. Die innere Schleife, die in Zeile 17 beginnt, wird immer acht mal durchlaufen. Dabei wird bei jedem Durchlauf das eingelesene Byte um ein Bit nach rechts verschoben. Somit entsteht aus jedem Eingabe-Bit ein Ausgabe-Byte.

7. Evaluation

In diesem Kapitel wird die Funktionsfähigkeit der entwickelten Lösung bewiesen. Dafür wurden Tests zur Belegung der allgemeinen Funktion und zur Messung von Zeitanforderungen durchgeführt. Alle in den voran gegangenen Kapiteln behaupteten Performance-Verbesserungen werden hier nachgewiesen. Um eine aussagekräftige Funktions- und Performanceanalyse aufstellen zu können, wurde Hardware implementiert, die das Verhalten der Kamera simuliert.

7.1. Bilddaten-Simulator

Dieser hier beschriebene Simulator wurde für alle Tests verwendet. Die Begründung für den Simulator liegt darin, dass der von der Kamera produzierte Bilddatenstrom nicht angehalten werden kann. Dies Option wird aber benötigt, um zu überprüfen, wie viel Zeit von der Aufnahme eines bestimmten Bildes bis zur Dekodierung dieses Bildes benötigt wird.

Um die Funktion beim späteren Anschluss einer echten Kamera garantieren zu können, ähnelt das erzeugte Bildsignal des Simulators dem der Kamera sehr. Der Simulator erzeugt, das von der Kamera bekannte Frame-Valid-, Line-Valid und Bilddaten-Signal. Dabei wurde besonders auf das Verhalten des Frame- und Line-Valid-Signals geachtet. Wie in Kapitel 3 schon beschrieben, liegt mit jeder steigenden Flanke des Bild-Taktes (PIXCLK) der Kamera, bei dem das Line-Valid-Signal High-Pegel aufweist, ein gültiger Bildpunkt am 10-Bit breiten Datenbus an. Ein weiterer wichtiger Zusammenhang ist die Kombination von Frame- und Line-Valid-Signal. Das Frame-Valid-Signal behält seinen High-Pegel während der gesamten Übertragungszeit für ein Bild. Das Line-Valid-Signal behält seinen High-Pegel für die Übertragungszeit einer Zeile. Das bedeutet, dass das Line-Valid-Signal niemals einen High-Pegel aufweisen darf, währenddessen das Frame-Valid-Signal einen Low-Pegel hat. Des Weiteren ist zu überprüfen, ob die Dauer des Line- und Frame-Valid-Signals konstant sind.

Unter Abbildung 7.1 ist das vom Simulator erzeugte Frame-Valid-Signal dargestellt. Für dieses Signal wurde ein Bild-Takt von 25 MHz genutzt. Dabei ergab sich eine konstante High-Phase

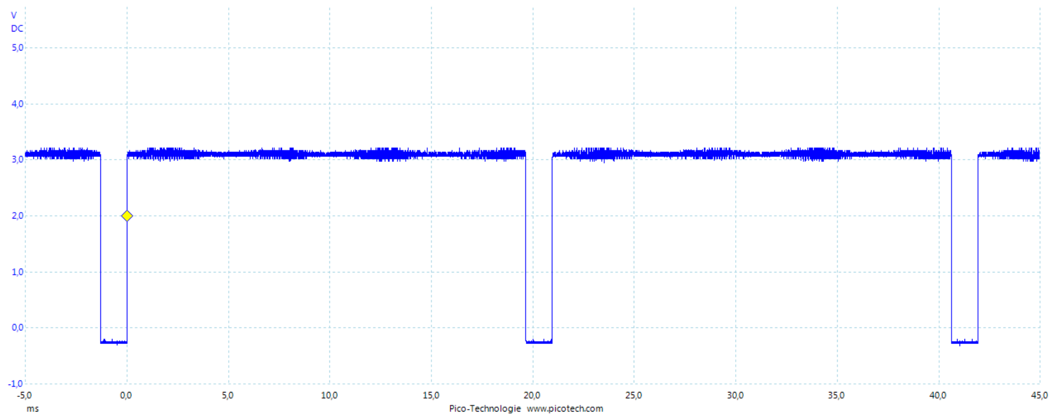

Abbildung 7.1.: Zeitverhalten des Frame-Valid-Signals vom Simulator (PicoScope 3204)

von 19,67 ms und eine Low-Phase von 1,304 ms. Zusammen ergibt dies eine Periodendauer von 20,97 ms. Das bedeutet, dass der Simulator mit einem Bild-Takt von 25 MHz 47,69 Bilder pro Sekunde generiert.

In der Abbildung 7.2 ist das Frame-Valid-Signal mit der oberen und das Line-Valid-Signal mit der unteren Kurve dargestellt. Auch hier wurde wieder ein 25 MHz Bild-Takt verwendet.

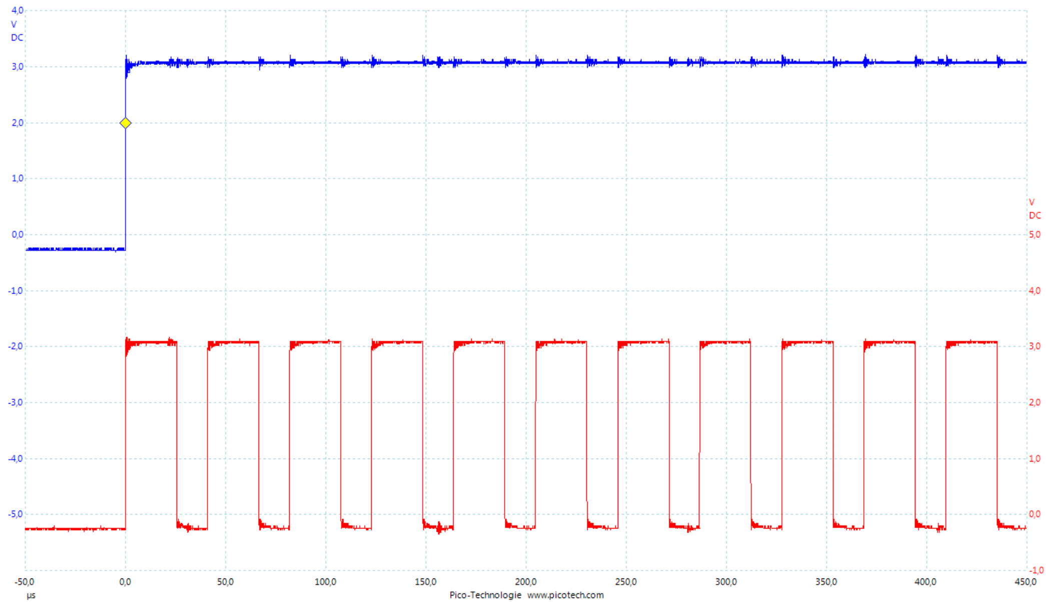

Abbildung 7.2.: Zeitverhalten des Line-Valid-Signals des Simulators (PicoScope 3204)

Dieser erzeugte im Line-Valid-Signal eine High-Phase von 25,62 µs und eine Low-Phase innerhalb eines Bildes von 15,34 µs. Zu beobachten ist an dem simulierten Signal, dass im Gegensatz zum im Datenblatt beschriebenen Signal der Kamera keine Verzögerung zwischen der steigenden Flanke des Frame-Valid-Signales und der steigenden Flanke des ersten Line-Valid-Signals pro Bild existiert. Anders beschrieben, entspricht die Dauer der unter Abbildung 3.1 mit P1 bezeichneten Verzögerung 0 s. Dies hat aber keine weiteren Auswirkungen auf die Verifikation mit Hilfe dieses Simulators, da dadurch eine nur noch schnellere Reaktion der zu testenden Hardware gefordert wird. Des Weiteren ist festzustellen, dass das Line-Valid-Signal nur dann einen High-Pegel aufweist, wenn das Frame-Valid-Signal auch high ist.

Das Bild wird vom Simulator mit einer Auflösung von 640x480 Bildpunkten erzeugt. Es besteht aus einem diagonalen Verlauf von Schwarz nach Weiß. Dabei bewegt sich dieser Verlauf von der linken oberen Ecke zur rechten unteren Ecke des Bildes. Das erzeugte Bild ähnelt der in Abbildung 7.3 gezeigten Grafik. Der Entschluss viel auf diese Struktur des Bildes,

Abbildung 7.3.: Vom Simulator generiertes Bild

da die Funktion der Binarisierung nachgewiesen werden kann. Es kann sogar der adaptiv berechnete Schwellwert erkannt werden. Dafür muss nur die Breite des erzeugten weißen Streifens gezählt werden. Die Funktion der Erosion kann auch getestet werden, da der durch die adaptive Binarisierung erzeugte weiße Streifen bei Aktivierung der Erosion dünner werden sollte. Ein weiterer Vorteil des gewählten Musters ist die einfache Implementierung eines Generators.

Wie zu Beginn dieses Kapitels erwähnt, benötigen einige Messungen die Möglichkeit den

Simulator anzuhalten. Dies kann über die Schalter auf dem "Digilent Nexys 3"-Board aktiviert werden. Mit dem Schalter SW0 kann der Simulator angehalten werden. Wenn dieser Schalter eingeschaltet wird, wird das aktuelle Bild noch zu Ende übertragen. Anschließend bleibt das Frame- und Line-Valid-Signal so lange auf dem Low-Pegel, bis der Schalter wieder ausgeschaltet wird.

Um überprüfen zu können, ob das Bild sich durch ein Fehlverhalten periodisch verändert, kann ein Standbild erzeugt werden. Dieses besteht auch aus einem Verlauf von Schwarz nach Weiß. Damit ein stehendes Bild generiert wird, kann der Schalter SW1 eingeschaltet werden. Auch hier kann der Simulator durch aktivieren des Schalters SW0 angehalten werden.

Der Simulator besteht in der Implementierung aus drei Zählern. Einer zählt die Bildpunkt innerhalb einer Zeile, der andere die Zeilen innerhalb eines Bildes und der dritte zählt das Bild. Der Bild-Zähler wird nur für das Generieren des bewegten Bildes benötigt. Die Logik für den Zähler des Bildpunktes einer Zeile ist unter Listing 7.1 zu finden.

```
1    if (STOP_GEN = '1' and frameCounter_cs = x"1FF" and
        lineCounter_cs = x"3FF") then
2      lineCounter_v    := lineCounter_cs;
3    else
4      lineCounter_v    := lineCounter_cs + 1;
5    end if;
```

Listing 7.1: Zähler-Logik für einen Bildpunkt [Wis13b]

Dieser Zähler mit dem Namen "lineCounter_v" wird mit jedem Takt in Zeile 4 inkrementiert. Wenn der Simulator angehalten werden soll, zählt dieser noch bis zum Bildende weiter. Ab dem Bildende wird in Zeile 2 immer der letzte Wert übernommen und nicht mehr inkrementiert.

Der Zähler für die Zeilen ist direkt abhängig vom Zähler für die Bildpunkte. Die etwas komplexere Logik für diesen Zähler ist in Listing 7.2 dargestellt.

```
1    if (lineCounter_v = IMAGE_WIDTH(22 to 31) ) then
2      lineValid_v    := '0';
3    elsif (lineCounter_v = x"000" ) then
4      -- increment frame counter because it is a new line
5      frameCounter_v  := frameCounter_v + 1;
6      lineValid_v    := '1';
7
8      -- increment the image offset on each new line
```

```
9       -- so it will be a stair
10      if (frameValid_v = '1') then
11        -- only increment it in a valid frame
12        -- so it will be a moving image
13        imageCounter_v  := imageCounter_v + 1;
14      end if;
15    end if;
```

Listing 7.2: Zähler-Logik für Zeilen und Bilder [Wis13b]

Mit jedem neuen Beginn einer Bild-Zeile, wird in Zeile 5 der Zeilen-Zähler "frameCounter_v" inkrementiert. Gleichzeitig wird das Line-Valid-Signal in Zeile 6 wieder auf High-Pegel gesetzt. Wenn der Bildpunkt-Zähler "lineCounter_v" die Breite des Ausgabebildes abgelaufen hat, wird das Line-Valid-Signal in Zeile 2 wieder auf Low-Pegel gesetzt. Der Bild-Zähler "imageCounter_v" wird in Zeile 13 nur dann inkrementiert, wenn eine neue Bild-Zeile beginnt und diese Zeile gültige Bilddaten enthält. Diese Überprüfung der Gültigkeit der Bilddaten wird in Zeile 10 durchgeführt.

Damit werden alle drei Zähler abhängig voneinander inkrementiert. Die Generierung des Frame-Valid-Signals wurde aber noch nicht beschrieben. Dies ist unter Listing 7.3 dargestellt.

```
1     if (frameCounter_v = IMAGE_HEIGHT(23 to 31) ) then
2       frameValid_v    := '0';
3     elsif (frameCounter_v = x"000" ) then
4       frameValid_v    := '1';
5     end if;
```

Listing 7.3: Generierung des Frame-Valid-Signals [Wis13b]

Zu Beginn eines neuen Bildes wird das Frame-Valid-Signal in Zeile 4 auf High-Pegel gesetzt. Wenn der Zeilen-Zähler den Wert der Bildhöhe erreicht hat, wird dieses Signal in Zeile 2 wieder auf Low-Pegel gesetzt.

Das Generieren der Bilddaten und damit verbunden Umschalten zwischen einem stehenden oder sich bewegenden Bild wird in Listing 7.4 aufgeführt.

```
1   outputSwitch: process (SHOW_FROZEN, lineCounter_cs,
       frameCounter_cs, imageCounter_cs) is
2   begin
3     if (SHOW_FROZEN = '1') then
4       -- generates a still standing picture
```

```
5       DATA_OUT     <= ( lineCounter_cs(2 to 9) +
            frameCounter_cs(1 to 8) ) & "00";
6     else
7       -- generates a running picture
8       DATA_OUT     <= ( lineCounter_cs(0 to 7) + imageCounter_cs
            ) & "00";
9     end if;
10  end process outputSwitch;
```

Listing 7.4: Bild-Generierung mit Hilfe der Zähler [Wis13b]

Für die Generierung eines sich bewegenden Bildes werden in Zeile 8 der Bildpunkt-Zähler und der Bild-Zähler addiert. Das Ergebnis wird um zwei Bit nach links geschoben. Somit wird nur ein Bild mit einer Farbtiefe von 8 Bit generiert. Diese geringere Farbtiefe wurde verwendet, um innerhalb eines Bildes den gesamten Farbverlauf unterbringen zu können. Für die Generierung des Standbildes werden in Zeile 5 der Bildpunkt-Zähler und der Zeilen-Zähler addiert. Auch dieser wird wegen des eben genannten Grundes um 2 Bit nach links geschoben.

Damit wurde die verwendete Testumgebung beschrieben. Nun kann die eigentliche Funktionsanalyse unter Verwendung dieses Simulators durchgeführt werden.

7.2. Funktionsanalyse

Mit der ersten Überprüfung, sollte sichergestellt werden, dass bei eingehenden Bilddaten auch Ethernet-Pakete versendet werden. Dafür wird in Abbildung 7.4 durch die ober Kurve das Frame-Valid-Signal vom Simulator und durch die untere Kurve das Belegt-Signal des "XPS Ethernet Lite MAC" dargestellt. Dieses Belegt-Signal weist immer dann einen High-Pegel auf, wenn der Sender gerade kein weiteres Paket versenden kann. Da nicht mehrere Pakete gleichzeitig versendet werden können, ist dies auch der Fall, wenn gerade ein Paket versendet wird. Wie im Bild zu erkennen ist, werden mehrere Pakete pro Bild versendet. Immer wenn keine gültigen Bilddaten vom Simulator generiert werden, werden auch keine Pakete versendet. Das verringert schon einmal die Wahrscheinlichkeit, dass ungültige Bilddaten mit versendet werden.

Da ein größeres Paket mehr Zeit für die Übertragung als ein kleines benötigt, kann anhand der Länge der Belegt-Pulse die Länge eines Paketes abgelesen werden. Bei genauerem hin gucken, fällt auf, dass alle Pakete die gleiche Länge aufweisen, außer das eine, dass zu Beginn eines neuen Bildes übertragen wird. Dieses kleiner Paket enthält die Rest-Daten des letzten Bildes.

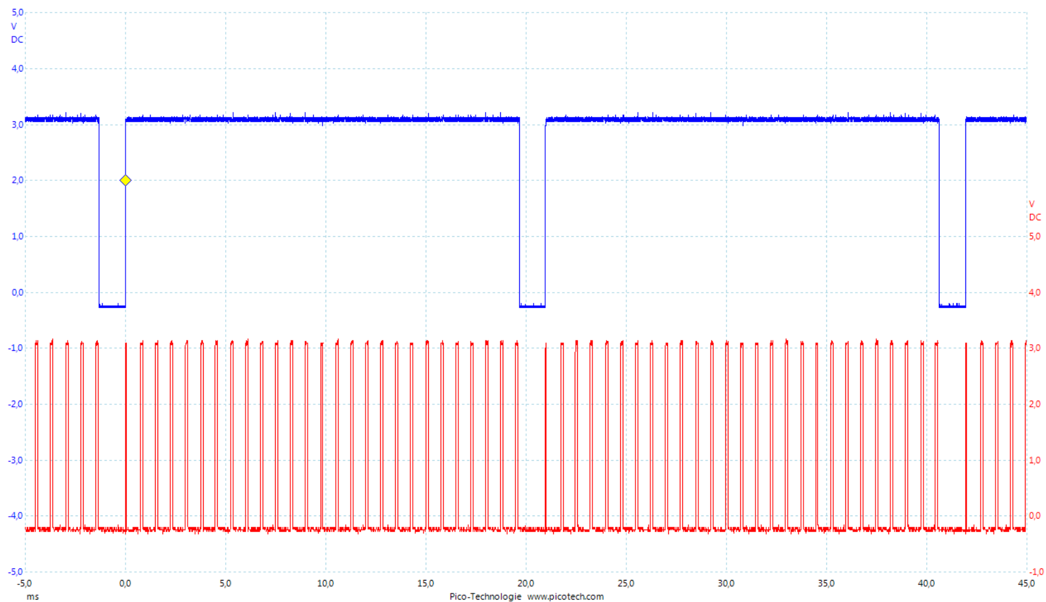

Abbildung 7.4.: Bilddaten-Eingabe und Paketversand (PicoScope 3204)

Weiterhin ist zu erkennen, dass zwischen den Belegt-Pulsen längere Low-Phasen existieren. Somit ist die Bandbreite der Ethernet-Schnittstelle noch nicht ausgelastet und es können noch weitere Daten übermittelt werden, wie zum Beispiel ein weiterer Stream.

Bei der nächsten Prüfung wurde der FIFO-Speicher genauer betrachtet. Eine häufige Fehlerursache ist ein Underflow oder Overflow solcher Speicherarten. Dies wird durch die in Kapitel 5.2 beschriebene Logik in dieser Lösung sehr schnell kenntlich gemacht. Wie schon in dem Kapitel beschrieben, ist dieses Fehlverhalten auch im dekodierten Bild zu erkennen. Um aber sehr dicht am FIFO-Speicher zu messen und somit die Wahrscheinlichkeit von Fehlern, die das Fehlverhalten wieder verdecken, zu minimieren, wurde der Ethernet-Verkehr mit Wireshark analysiert. Ein Auszug der Analyse ist in Abbildung 7.5 dargestellt. Im unteren Teil der Abbildung befindet sich ein Ausschnitt der Daten des markierten Paketes. In diesem Fall ist es das erste Paket eines Bildes. Die markierten Daten enthalten den in Kapitel 5.5 beschrieben Bilddaten-Header. Anschließend an diesen Header folgen die unkomprimierten Bilddaten. Im Fehlerfall würden an dieser Stelle die sich wiederholenden Fehlercodes stehen. Somit ist nachgewiesen, dass der FIFO-Speicher weder einen Overflow noch einen Underflow produziert.

Des Weiteren kann in der Übersicht über die empfangen Pakete, die sich in der oberen Hälfte

No.	Time	Source	Destination	Protocol	Length	Info
4305	3.343222000	192.168.0.5	224.0.0.1	UDP	410	Source port: 0 Destination port: search-agent
4306	3.344123000	192.168.0.5	224.0.0.1	UDP	1506	Source port: 0 Destination port: search-agent
4307	3.344827000	192.168.0.5	224.0.0.1	UDP	1506	Source port: 0 Destination port: search-agent

```
Frame 4306: 1506 bytes on wire (12048 bits), 1506 bytes captured (12048 bits) on interface 0
Ethernet II, Src: UscInfor_00:fa:ce (00:00:5e:00:fa:ce), Dst: IPv4mcast_00:00:01 (01:00:5e:00:00:01)
Internet Protocol Version 4, Src: 192.168.0.5 (192.168.0.5), Dst: 224.0.0.1 (224.0.0.1)
User Datagram Protocol, Src Port: 0 (0), Dst Port: search-agent (1234)
Data (1464 bytes)

0000  01 00 5e 00 00 01 00 00  5e 00 fa ce 08 00 45 00   ..^..... ^.....E.
0010  05 d4 00 00 00 00 01 11  13 6b c0 a8 00 05 e0 00   ........ .k......
0020  00 01 00 00 04 d2 05 c0  00 00 59 a6 6a 95 00 00   ........ ..Y.j...
0030  02 80 00 00 01 e0 00 00  00 01 00 00 96 00 00 00   ........ ........
0040  00 01 00 00 00 00 00 00  00 00 fc ff ff 7f 00 00   ........ ........
0050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
```

Abbildung 7.5.: Ethernet-Verkehr mit erstem Paket eines neuen Bildes (Wireshark 1.10.1)

befindet, die Paketlänge der einzelnen Pakete abgelesen werden. Dabei ist, wie aus Abbildung 7.4 schon abgelesen, zu erkennen, dass die Pakete unterschiedliche Längen aufweisen. Das Paket vor dem markierten Paket ist kürzer als alle anderen. Da das markierte Paket das erste eines Bildes ist, ist damit bewiesen, dass das kleiner Paket das letzte eines vorherigen Bildes ist.

7.2.1. Binarisierung

Die erste Sichtprüfung besteht darin, dass PC-seitig dekodierte Ergebnisbild zu begutachten. Dafür wurde Schalter SW3 ausgeschaltet und somit das Erosionsfilter deaktiviert. Damit ist in Abbildung 7.6 nur das Ergebnis der Binarisierung zu finden. Wenn dieses mit dem

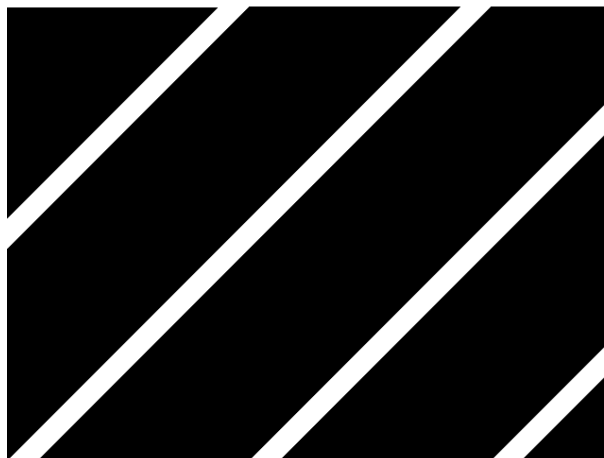

Abbildung 7.6.: Durch die adaptive Binarisierung verarbeitetes Bild

vom Simulator erzeugten Bild, dass in Abbildung 7.3 vorgestellt wurde, verglichen wird, fällt eine Ähnlichkeit auf. Das Bild wechselt ab einem bestimmten Grauwert von Schwarz nach Weiß. Ein weißer Streifen hat eine Breite von 32 Bildpunkten. Zwischen Ende eines Streifens und Ende des nächsten Streifens befinden sich 256 Bildpunkte. Daraus ergeben sich genau die in Kapitel 4.1 beschriebenen $\frac{32}{256} = 12,5\%$. Diese Rechnung geht aber nur auf, da im simulierten Bild zwischen zwei Bildpunkten immer der gleiche Farbunterschied besteht. Mit dieser Überprüfung ist nachgewiesen, dass der Schwellwert wie beschrieben berechnet wird.

7.2.2. Erosionsfilter

Zur Überprüfung der Erosion wird sie über Einschalten des Schalters SW3 aktiviert. Das Ausgabebild entspricht dem unter Abbildung 7.7 dargestelltem. Bei genauem Vergleichen

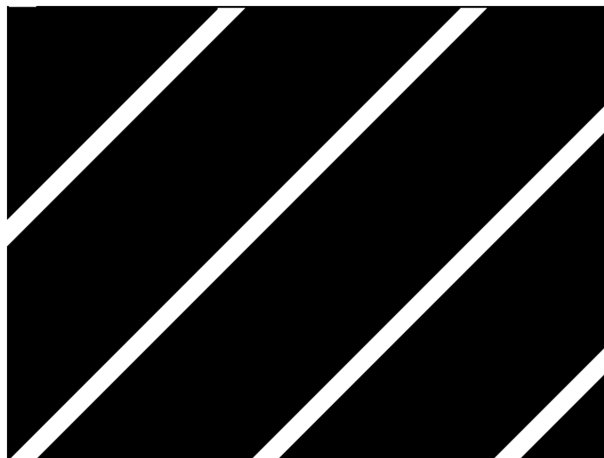

Abbildung 7.7.: Durch die adaptive Binarisierung und das Erosionsfilter verarbeitetes Bild

dieses Bildes mit dem aus Abbildung 7.6, bei dem die Erosion deaktiviert war, fällt auf, dass die weißen Streifen nur noch 28 Bildpunkte breit sind. Daraus folgt, dass an jedem Schwarz-Weiß-Übergang zwei weiße Bildpunkte in schwarze umgewandelt wurden. Das ist auch das in Kapitel 4.2 beschrieben Verhalten einer Erosion mit einer 3x3 Faltungsmaske. Somit ist auch diese Funktion nachgewiesen.

7.2.3. Verifikation mit verschiedenen Takten

Da das Gesamtsystem mit zwei unterschiedlichen Takten betrieben wird, ist eine Überprüfung mit verschieden Kombinationen von unterschiedlich schnellen Takten sehr Ratsam. Hauptaugenmerk ist dabei das Ausfindig machen von metastabilen Zuständen. Hierfür wurden

Takte gewählt, die kein vielfaches voneinander sind. Dadurch kann es zu sehr kurzen Verzögerungen zwischen den steigenden Flanken der beiden Taktsysteme kommen. Unter Tabelle 7.1 sind die verwendeten Takte und die daraus resultierenden Ergebnisse aufgeführt. Die

Bild-Takt	Sender-Takt	FPS	Netzwerkauslastung	Bilder fehlerfrei dekodiert?
12,5 MHz	83,3 MHz	23,84	921 KiB/s	Ja
25 MHz	83,3 MHz	47,69	1,8 MiB/s	Ja
33,3 MHz	83,3 MHz	63,58	2,4 MiB/s	Ja
50 MHz	83,3 MHz	95,37	3,6 MiB/s	Ja

Tabelle 7.1.: Funktionstest der Hardware mit verschiedenen Takten (PicoScope 3204)

Funktionsprüfung für die in dieser Tabelle aufgeführten Takte wurde mit einer Sichtprüfung des dekodierten Bildes durchgeführt. Im Fehlerfall würden im dekodierten Bild Abweichungen vom in Abbildung 7.7 gezeigten Bild auftreten.

7.3. Performanceanalyse

Die in diesem Unterkapitel beschriebenen Tests sollen die ausreichend schnelle Verarbeitung der Bilddaten belegen. Dabei werden mehrere Teile der Hardware und das Gesamtsystem betrachtet. Der erste durchgeführte Test bezieht sich auf die Dauer zwischen dem Beginn eines neuen Bildes und dem Versand des ersten Paketes. Dabei liegt das Interesse in der gleichmäßigen Verteilung der Pakete. Wenn alle Pakete erst am Ende eines Bildes gesendet werden würden, wäre der empfangende PC zu diesem Zeitpunkt sehr belastet. Möglicherweise würden dann andere wichtige Prozesse zu wenig Ressourcen zugeteilt bekommen. In Abbildung 7.8 wird diese Verzögerung bis zum Versand des ersten Paketes dargestellt. Diese Zeitspanne beträgt 757,9 µs. Somit ist dies circa $\frac{1}{20}$ der gesamten Zeit von 19,67 ms, die für die Übertragung eines Bildes vom Simulator benötigt wird. Das bedeutet, dass das erste Paket ziemlich dicht am Beginn eines Bildes versendet wird. Somit wird die CPU nicht durch das schlagartige Empfangen aller Pakete belastet. Diese gleichmäßige Verteilung ist auch sehr gut in der unteren Kurve der Abbildung 7.4 zu erkennen.

Die gemessene Verzögerung kommt hauptsächlich durch das Füllen des FIFO-Speichers zustande. Damit ist diese sehr stark vom verwendeten Bilddaten-Takt abhängig, da bei einem schnelleren Takt sich der Speicher auch schneller füllen würde. Weiterhin werden in den einzelnen Pipeline-Stufen Register verwendet. Diese haben meistens das Ziel, den kritischen Pfad in der Hardware zu minimieren. Damit werden weitere Verzögerungen erzeugt. Die adaptive

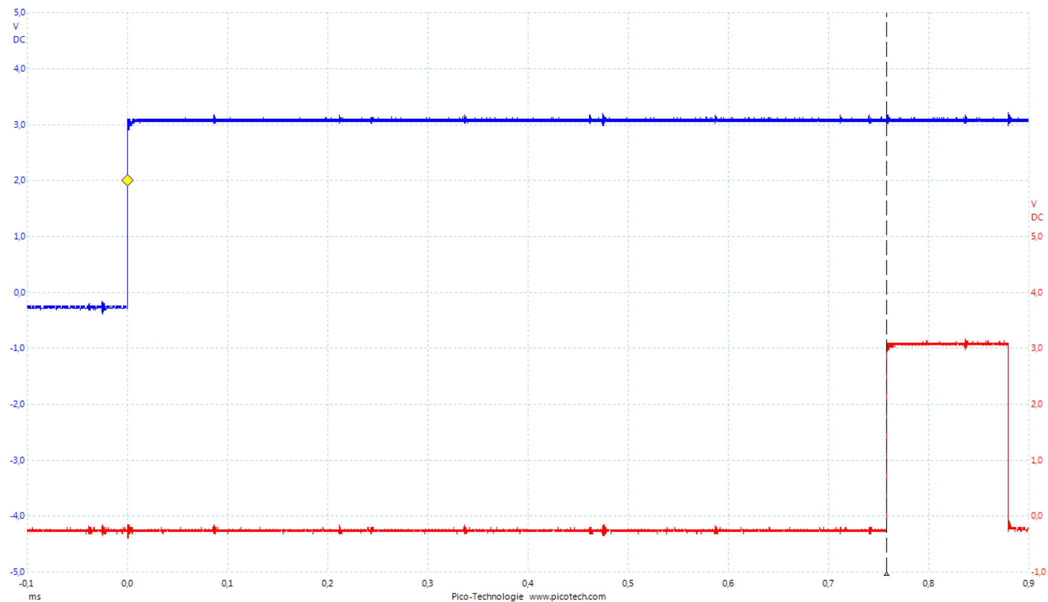

Abbildung 7.8.: Verzögerung vom Beginn eines neuen Bildes bis zum Versand des ersten Paketes (PicoScope 3204)

Binarisierung benötigt nur eine Periode des Taktes, um einen Bildpunkt zu verarbeiten. Das Erosionsfilter benötigt schon vier Perioden, um einen Bildpunkt zu verarbeiten. Da dieses aber als Pipeline aufgebaut wurde, liegt das Ergebnis des Gesamtbildes nur diese vier Takte später nachdem, das Ende des Eingabebildes erreicht wurde, erodiert vor. Bei einem Bilddaten-Takt von 25 MHz bedeutet das eine Verzögerung von $\frac{1}{25\text{MHz}} * 4 = 160\text{ns}$.

Eine größere Verzögerung entsteht im Bilddaten-Sender. Dieser wird aber nicht mit dem langsameren Bilddaten-Takt betrieben, sondern mit dem Takt, der auch die Sende-Einheit verwendet. In Abbildung 7.9 ist diese Verzögerung von der Signalisierung des FIFO-Speichers, dass Daten zum Versand bereit stehen, bis zum Versand des ersten Bytes aus dem Speicher dargestellt. Die oberste Kurve zeigt dabei den Systemtakt. Die Kurve in der Mitte stellt die

Abbildung 7.9.: Verzögerung zwischen FIFO-Signalisierung und erstes Byte der Bilddaten versendet (ModelSim SE-64 6.5b)

Signalisierung des FIFO-Speichers, durch die eine neue Übertragung ausgelöst wird, da. In der

unteren Kurve sind die Daten, die Übertragen werden zu finden. Wenn diese Daten mit den beschriebenen Headern verglichen werden, kann festgestellt werden, dass diese identisch sind. Insgesamt werden von der Signalisierung bis zum vollständigen Versand 90 Takte benötigt. Die meisten Takte werden dabei, wie in der Abbildung zu erkennen ist, für den Versand der Header benötigt.

Wie in der Funktionsanalyse schon erläutert wurde, wird das letzte Paket zu Beginn des nächsten Bildes versendet. Diese Zeit könnte kritische Auswirkungen haben, da damit eigentlich ein veraltetes Bild versendet wird. Dabei spielen auch die eben beschriebenen 90 Takte mit in diese Verzögerung rein. In Abbildung 7.10 ist diese dargestellt. Die Zeit beträgt hierbei

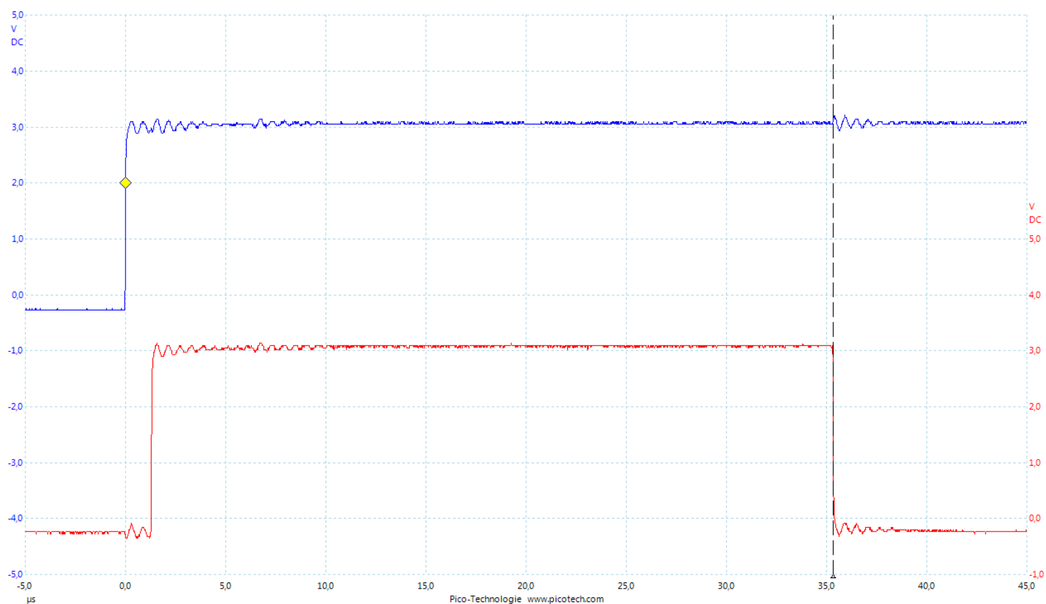

Abbildung 7.10.: Verzögerung vom Beginn eines neuen Bildes bis zum Versand des letzten Bildes (PicoScope 3204)

35,34 µs. Die untere Kurve stellt das Belegt-Signal der Versende-Hardware da. Somit wird, wie in der Abbildung zu sehen, fast die gesamte Zeit für das Übertragen der Daten über die Ethernet-Schnittstelle benötigt. Da diese Verzögerung auch nur ein Bruchteil der Übertragung eines gesamten Bildes des Simulators beträgt, ist sie vernachlässigbar. Zu beachten ist hierbei aber, dass diese Zeit durch erhöhen des Bilddaten-Taktes nicht minimiert wird, da die Bilddaten schon im FIFO-Speicher liegen und nur noch auf den Versand warten.

Um zu berechnen, wie lange die entwickelte Hardware für die Übertragung eines gesamten Bildes benötigt, muss diese Verzögerung nur mit der Übertragungszeit, die die verwendete Kamera für ein Bild benötigt, addiert werden. Für die vom Simulator erzeugten Bilddaten bedeutet das also eine Verzögerung von 20,97 ms + 35,34 µs = 21,01 ms. Dies wurde durch Messung noch mal verifiziert und ist in Abbildung 7.11 dargestellt. Dabei stellt die obere

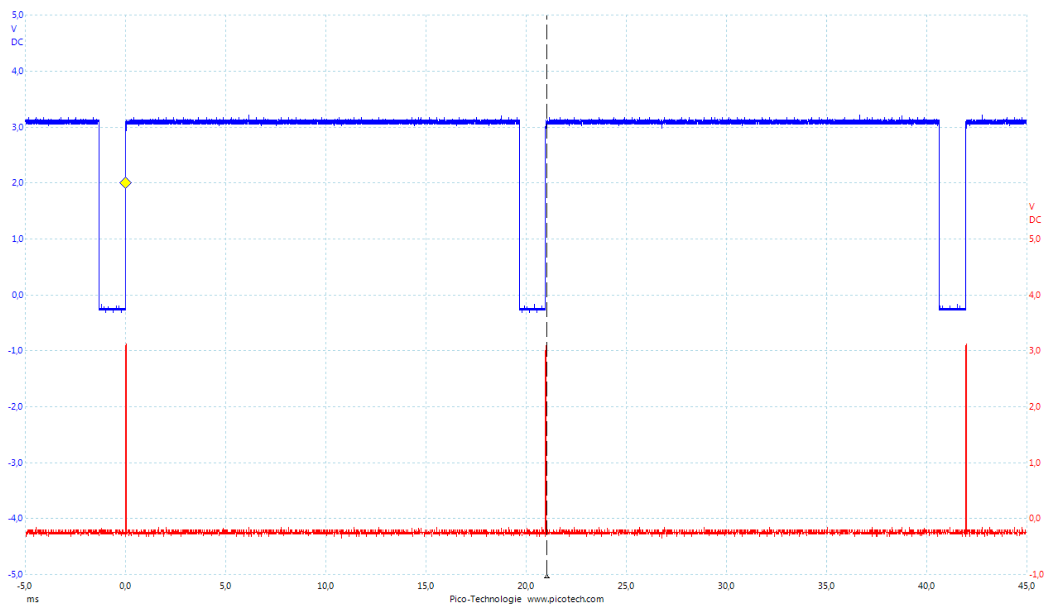

Abbildung 7.11.: Verzögerung vom Beginn eines neuen Bildes bis zum vollständigen Versand dieses Bildes (PicoScope 3204)

Kurve das Frame-Valid-Signal der Kamera da. Die untere Kurve spiegelt das Belegt-Signal der Sende-Einheit für kleiner Pakete wieder. Da nur das letzte Paket kleiner als die anderen ist, ist auch nur der Versand dieses Paketes dort wieder zu finden.

7.3.1. Bild-Dekodierung

Die bis jetzt gemessenen Zeiten bezogen sich nur auf das Verhalten der Hardware. Somit ergeben die Messungen immer das gleiche Ergebnis, da keine Schwankungen in der Verarbeitungskette auftreten. Bei der Verwendung von Software ist dies nicht der Fall, da andere Prozesse zwischen durch Ressourcen belegen können.

In Kapitel 6.4 wurde beschrieben, dass eine Eigenentwicklung für die Bild-Dekodierung verwendet wird. Der Hintergrund hierfür war eine performantere Lösung zu schaffen. Um dies belegen zu können, wurden die beiden Methoden zur Dekodierung verglichen. Die Ergebnisse

sind in Tabelle 7.2 zu finden. Für beide Messungen generierte der Simulator 47,69 Bilder

Untersuchte Methode	Minimale Zeit	Maximale Zeit	Mittelwert
cv::imdecode (OpenCV)	2157 µs	5069 µs	3068 µs
StreamReader::fastImageDecode	346 µs	962 µs	364 µs

Tabelle 7.2.: Vergleich der Bild-Dekodierung von OpenCV und der Eigenentwicklung (Debian 7.0, GNU C Library 2.13-38, gettimeofday)

pro Sekunde. Diese Bilder hatten eine Auflösung von 640x480 Bildpunkten. Das Bild wurde unkomprimiert mit einer Farbtiefe von einem Bit im "Sun Raster"-Format übertragen. Zur Mittelwert-Berechnung wurde das arithmetische Mittel gebildet. Anhand dieses Mittelwerts ist der Geschwindigkeitsvorteil der Eigenentwicklung von fast $\frac{1}{10}$ gut zu erkennen. Wenn die maximal benötigte Rechenzeit der OpenCV-Variante von circa 5 ms mit der Bildwiederholrate von 20 ms bei 40 Bildern pro Sekunde verglichen wird, fällt auf, dass allein die Dekodierung die CPU für $\frac{1}{4}$ belegen würde. Dies ist ein klares K.O.-Kriterium für die Verwendung der OpenCV-Dekodierung, da andere Ressourcen-hungrigere Aufgaben für die Fahrspurführung auch mit ausgeführt werden müssen. Um belegen zu können, dass die Eigenentwicklung schnell genug ist, kann nicht nur die Dekodierung betrachtet werden. Die Verzögerung, die durch den Empfang der Daten entsteht, sollte auch mit berücksichtigt werden.

In Abbildung 7.12 wird die Verzögerung vom Versand des letzten Paketes bis zur vollständigen Dekodierung dieses Bildes auf dem PC dargestellt. Die hier gezeigte Grafik wurde mit Hilfe der Speicher-Funktion des PicoScope aufgezeichnet. Es beschreibt eine maximale Verzögerung von einer Messung die sich über eine Minute erstreckte. Dabei wurde eine maximale Zeit von 983,0 µs gemessen. Diese Zeit ist sehr stark CPU-Abhängig und könnte zum Beispiel durch eine schnellere CPU minimiert werden. Die obere Kurve stellt den Versand des letzten Paketes eines Bildes da. In der unteren Kurve wird die Sende-Leitung einer RS232-Schnittstelle des Empfänger-PCs dargestellt. Über diese RS232-Schnittstelle wurde Software-seitig nach jeder Dekodierung ein Byte mit dem Wert FF_{16} versendet. Der dabei verwendete RS232-Adapter arbeitet mit einem TTL-Pegel, somit bewegen sich gültige Signale nur zwischen 0 und 5 Volt.

7.3.2. Latenz des Gesamtsystems

Das am Ende ausschlaggebende ist die Verzögerung die das Gesamtsystem erzeugt bis das Bild für die Fahrspurerkennung genutzt werden kann. Um diese zu berechnen muss auf die eben beschriebene Zeit, die für den Empfang und die Dekodierung benötigt wird, die Zeit, die die Hardware für das Versenden eines vollständigen Bildes benötigt, addiert werden.

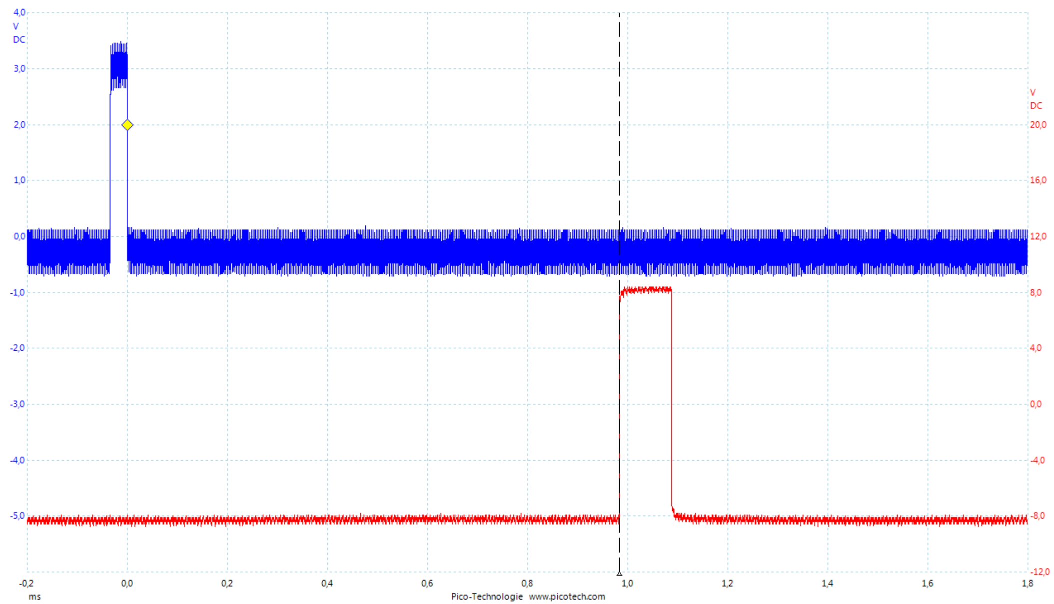

Abbildung 7.12.: Verzögerung vom Versand des letzten Paketes bis zur vollständigen Dekodierung (PicoScope 3204)

Daraus ergibt sich eine Zeit von 21,01 ms + 983,0 µs = 21,99 ms. Dies konnte auch mit der vollständigen Messung, die in Abbildung 7.13 zu finden ist, nachgewiesen werden. Dabei stellt

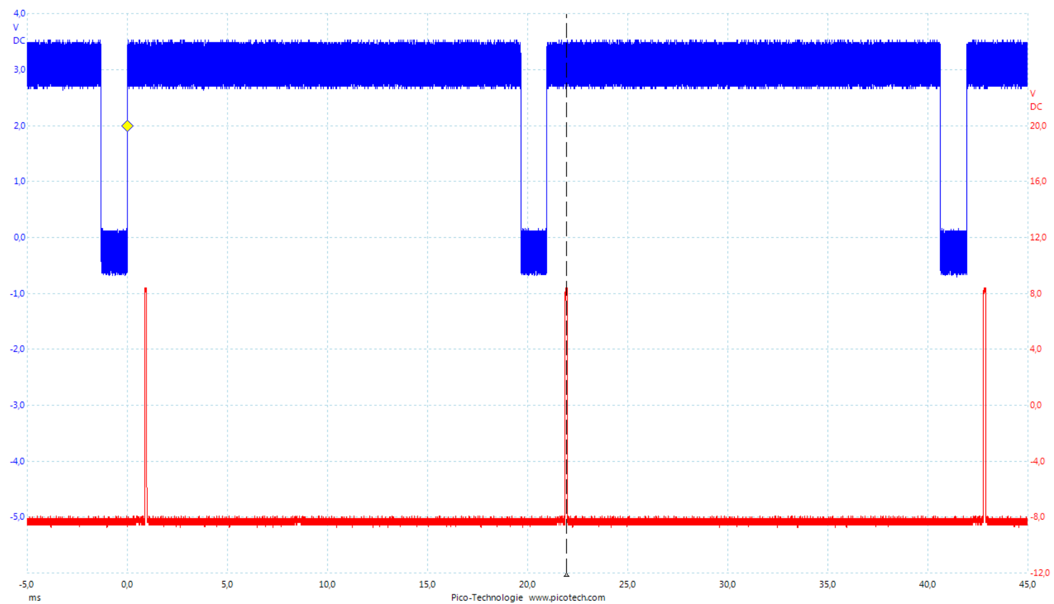

Abbildung 7.13.: Latenz von Bildbeginn bis zur abgeschlossenen Dekodierung (PicoScope 3204)

die obere Kurve das Frame-Valid-Signal des Simulators da. Die untere Kurve repräsentiert wieder die Sende-Leitung der RS232-Schnittstelle des Empfänger-PCs. Um nun einschätzen zu können, ob diese Verzögerung kritisch ist, sollte die Geschwindigkeit des Fahrzeuges mit betrachtet werden. Bei aktuellen Fahrten des Fahrzeuges liegen die Geschwindigkeiten unter $1,5\frac{\text{m}}{\text{s}}$. [Bec13] Das bedeutet, dass sich das Fahrzeug $1,5\frac{\text{m}}{\text{s}} \cdot 21,99\text{ms} = 32,99\text{mm}$ innerhalb dieser Verzögerung bewegt. Somit ist diese Zeitspanne unkritisch.

7.3.3. CPU-Auslastung

Ein anderer sehr wichtiger Punkt, der eine Aussage über die Performance der Lösung macht, ist die Auslastung der CPU. Diese wird mit dem Empfang der Pakete und der Dekodierung beschäftigt. Dies werden bei Verwendung auf einem FAUST-Fahrzeug aber nicht die einzigen Komponenten sein, die für eine Fahrspurführung ausgeführt werden müssen. Deshalb spielen die noch zur Verfügung stehenden Ressourcen eine große Rolle.

Zur Messung der CPU-Auslastung und des Ressourcen-Bedarfs wurde das Linux-Tool "top" verwendet. Die Ergebnisse sind in der Tabelle 7.3 dargestellt. Diese Messungen wurden auf

Lösung	FPS	Auslastung	Ressourcen (über 1 min)	Ressourcen (über 5 min)
USB-Kamera	40	29,6 %	1,23	1,11
Ethernet-Stream	47,69	5,8 %	0,07	0,10

Tabelle 7.3.: Auslastung der Empfänger-PC-Ressourcen (Debian 7.0, top 3.3.3)

der CPU, die auf den Fahrzeugen verbaut ist, durchgeführt. Die Spezifikation dieser CPU sind in Tabelle 2.1 zu finden. Der Ressourcen-Bedarf wurde über eine und fünf Minuten gemittelt. Dabei ist die Aussagekraft dieser Werte von System zu System unterschiedlich. Berechnet wird dieser aus der Summe aller aktiven Prozesse. Deshalb beschreibt ein größerer Wert einen höheren Ressourcen-Bedarf. [vgl. PPOL02, S. 1060] Somit ist aus der Tabelle klar zu entnehmen, dass die CPU-Auslastung und der Ressourcen-Bedarf im Vergleich zur aktuellen USB-Lösung stark gesunken ist.

7.3.4. Wahl des Übertragungsweges

Durch die hohe Auslastung der CPU bei Verwendung der USB-Kamera kann auch die Wahl der Ethernet-Schnittstelle begründet werden. Die Bilddaten werden bei der USB-Kamera so wie bei der hier entstandenen Lösung unkomprimiert übertragen. Der Nachweis dafür ist in Abbildung 7.14 zu finden. Diese Abbildung zeigt einen Auszug aus Wireshark. Dabei wird der

No.	Time	Source	Destination	Protocol	Length	Info
7587	17.94503500(	10.2	host	USB	16448	URB_BULK in
7588	17.94511200(	host	10.2	USB	64	URB_BULK in
7589	17.94603700(	10.2	host	USB	16448	URB_BULK in

```
▷ Frame 7589: 16448 bytes on wire (131584 bits), 16448 bytes captured (131584 bits) on interface 0
▽ USB URB
    URB id: 0xffff880128e19740
    URB type: URB_COMPLETE ('C')
    URB transfer type: URB_BULK (0x03)
  ▷ Endpoint: 0x82, Direction: IN
    Device: 10
    URB bus id: 1
    Device setup request: not relevant ('-')
    Data: present (0)
    URB sec: 1363269578
    URB usec: 198751
    URB status: Success (0)
    URB length [bytes]: 16384
    Data length [bytes]: 16384
    [Request in: 7588]
    [Time from request: 0.000925000 seconds]
    [bInterfaceClass: VENDOR_SPECIFIC (0xff)]
  Leftover Capture Data: ffffffffffffffffffffffffffffffffffffffffffffffff...
```

Abbildung 7.14.: Übertragung eines weißen Bildes der USB-Kamera (Wireshark 1.10.1)

Datenverkehr zwischen Kamera und Host beim Übertragen eines weißen Bildes dargestellt. Wie zu erkennen ist, hat jedes Byte der Nutzdaten des markierten Paketes den Wert FF_{16}. Dies spiegelt Bildpunkte mit weißer Farbe wieder. Insgesamt werden $752 * 480 = 360.960$ Byte pro Bild übertragen. Somit ist bewiesen, dass die Bilddaten weder komprimiert noch anderweitig verändert werden. Damit muss die Hauptaufgabe des USB-Kamera-Treibers sein, die Bilddaten von der Kamera abzuholen und unter einer Speicheradresse im User-RAM zur Verfügung zu stellen. Diese Aufgabe belastet die CPU aber schon zu mehr als $\frac{1}{4}$. Daraus lässt sich schließen, dass das Empfangen von Daten über die USB-Schnittstelle höheren Softwareaufwand bedarf als über die Ethernet-Schnittstelle.

In Kapitel 7.3 wurde erwähnt, dass ein Großteil der Verzögerungen durch das Auffüllen des FIFO-Speichers entstehen. Wenn alternativ der USB verwendet werden würde, würde der PC den FPGA nach Daten fragen. Somit müsste der FPGA mindestens ein Bild zwischenspeichern können, damit bei der nächsten Anfrage vom PC ein vollständiges Bild übertragen werden könnte. Das bedeutet aber wieder herum, dass ein größerer Speicher im FPGA existieren müsste. Dieser würde dem entsprechend auch eine größere Verzögerung zu Folge haben.

7.4. Kamera-Anbindung

In Kapitel 3 wurde beschrieben, dass die Pmod-Anschlüsse ungeeignet für die Anbindung der Kamera seien. Hintergrund hierfür ist der hohe Takt mit dem die Daten übertragen werden. Um zu verifizieren, dass die Pmod-Anschlüsse wirklich ungeeignet sind und die VHDC-Buchse dafür verwendet werden kann, wurden von beiden Varianten ein Pin als Taktausgang genutzt. Das angelegte Signal hatte eine Frequenz von 20,9 MHz. Somit liegt es noch unter dem von der Kamera zu erwartenden Takt von 27 MHz.

In Abbildung 7.15 wird das vom Pmod-Anschluss abgegriffene Signal dargestellt. Dieses

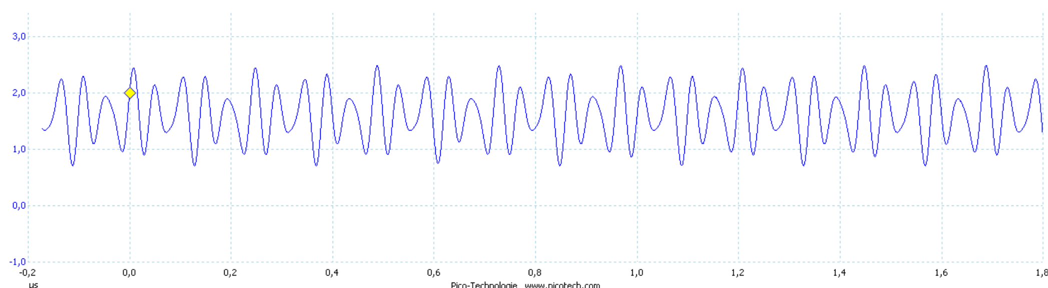

Abbildung 7.15.: Verfälschtes Taktsignal an einem Pin eines Pmod-Anschlusses gemessen (PicoScope 3204)

Signal weicht sehr stark von einem Rechtecksignal ab. Die gravierenderen Verfälschungen sind dabei aber die Spannungspegel. Der maximale Low-Pegel von 0,8 Volt und der minimale High-Pegel von 2 Volt kann nicht immer eingehalten werden. Damit wird der gültige Bereich von 3,3V-LVCMOS-Technik verlassen und dieses Signal kann nicht zuverlässig weiterverarbeitet werden. [vgl. JED07, S. 8]

Anders sieht es mit der VHDC-Buchse aus. In Abbildung 7.16 wird die Messung mit dieser Buchse dargestellt. Dieses Signal besitzt zwar auch keine rechteckigen Verläufe mehr, aber die Spannungen für High- und Low-Pegel liegen weiter auseinander. Der maximale Low-Pegel wird nie überschritten und der minimale High-Pegel wird nie unterschritten.

Bei weiteren Untersuchungen wurde ein generiertes Taktsignal über einen Pmod-Pin mit dem FPGA verbunden. Dieser Takt wurde für die Bildverarbeitungskette verwendet. Also wurde damit genau der Teil betrieben, der später vom Kamera-Takt betrieben werden soll. Um

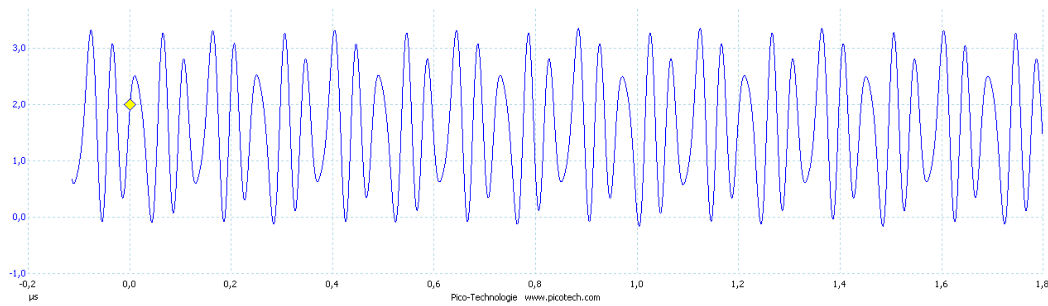

Abbildung 7.16.: Taktsignal an einem Pin eines VHDC-Anschlusses gemessen (PicoScope 3204)

die Funktionsweise bei unterschiedlichen Taktraten zu überprüfen, wurde das Frame-Valid-Signal des Simulators und die Sendesignalisierung des Bilddaten-Senders betrachtet. Mit dem Frame-Valid-Signal wurde auch gleichzeitig die simulierte Bildwiederholrate gemessen. Die Ergebnisse sind in Tabelle 7.4 dargestellt. Damit ist bestätigt, dass eine Verwendung der Pmod-

Angelegter Takt	FPS	Sendesignal generiert?
2,6 MHz	4,967	Ja
5,21 MHz	9,933	Nein
10,42 MHz	19,87	Nein

Tabelle 7.4.: Funktionstest bei Verwendung eines über einen Pmod-Pin angelegten Taktes (PicoScope 3204)

Anschlüsse nicht möglich ist. Somit ist die Herstellung eines Adapters für die VHDC-Buchse notwendig.

Um auch ohne diesen Adapter die richtige Initialisierung der Kamera verifizieren zu können, wurden die Frame- und Line-Valid-Signale der Kamera begutachtet. Da die für die Konfiguration der Kamera benötigte I^2C-Schnittstelle mit einem viel kleineren Takt als die Kamera an sich betrieben wird, konnte diese Verbindung über die Pmod-Anschlüsse hergestellt werden. Ziel ist es dabei die Kamera mit der maximalen Auflösung von 752x480 Bildpunkten und einer maximalen Bildwiederholrate zu konfigurieren. Die Messungen der zwei Synchronisationssignale ist in Abbildung 7.17 zu finden. Dabei wurde für das Frame-Valid-Signal eine High-Phase von 34,55 ms ermittelt. Das Line-Valid-Signal wies einen 55,47 µs langen High-Pegel auf. Diese großen Zeiten kommen zu Stande, da bei den Untersuchungen festgestellt werden musste, dass die von der FRAMOS GmbH gelieferte Kamera nicht mit dem erwarteten 27 MHz sondern nur mit 13,56 MHz getaktet wird. Deshalb können auch keine Bildraten von 60 Bildern pro Sekunde

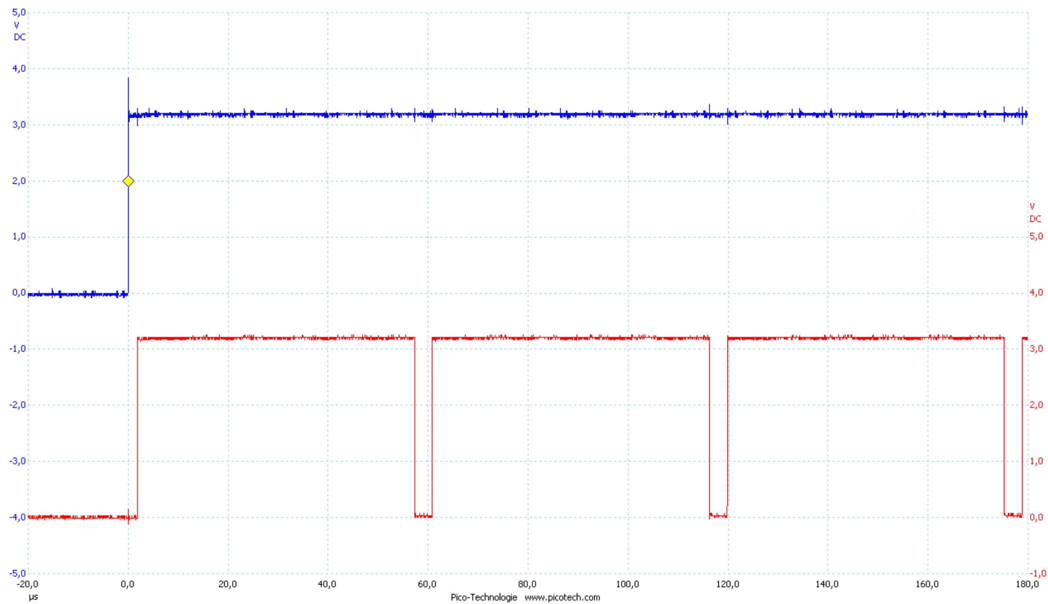

Abbildung 7.17.: Frame- und Line-Valid-Signal der Kamera bei einer Auflösung von 752x480 Bildpunkten (PicoScope 3204)

bei voller Auflösung erreicht werden. Anhand dieses langsameren 13,56 MHz Taktes und dem Line-Valid-Signal kann die Anzahl der Bildpunkte pro Zeile bestimmt werden. Daraus ergibt sich, dass die Periodendauer für die Übertragung eines Bildpunktes bei $\frac{1}{13{,}56\text{MHz}} = 73,75\text{ns}$ liegt. Das bedeutet, dass $\frac{55{,}47\mu\text{s}}{73{,}75\text{ns}} \approx 752$ Bildpunkte pro Zeile übertragen werden. Dies ist auch die maximale Anzahl an Bildpunkten in der Breite, die von der Kamera unterstützt werden.

Für die Bildwiederholrate wurden 33,5 Bilder pro Sekunde gemessen. Diese kann bei dem gegebenen Takt nicht weiter erhöht werden, da dafür die Low-Phasen des Line-Valid-Signals verkleinert werden müssten. Dies ist aber nicht möglich, da die Kamera eine minimale Zeit benötigt, um das Auslesen in der nächsten Zeile fort zusetzen. Um diese minimale Zeit zu ermitteln wurden die Konfiguration der Kamera so lange angepasst, bis die kürzeste Low-Phase, bei der noch ein gültiges Signal übertragen wird, gefunden wurde. Die Funktion der entwickelten Hardware bei weitaus höheren Bildraten wurde schon in Kapitel 7.2.3 mit Hilfe des Simulators nachgewiesen. Dabei ist Tabelle 7.1 zu entnehmen, dass Raten von über 90 Bildern pro Sekunde erfolgreich getestet wurden. Somit ist in diesem Fall die Kamera der Engpass.

8. Zusammenfassung

Ziel dieser Arbeit war das Binarisieren und Erodieren von Live-Bildern in Echtzeit. Dabei wurden Lösungsansätze, die nur auf einer CPU ausgeführt werden, betrachtet. Hier wurde festgestellt, dass diese Berechnung einen sehr hohen Ressourcen-Bedarf hat und somit auf keiner energiesparenden CPU in einer akzeptablen Zeit ausgeführt werden kann. Des Weiteren wurde versucht, die GPU für die Berechnung der Binarisierung zu verwenden. Diese Variante scheiterte jedoch beim Kopiervorgang der binarisierten Bilddaten aus dem GPU-Speicher in den CPU-Speicher. Dieses Zurückkopieren wäre jedoch nötig gewesen, da die Daten von der CPU weiter analysiert werden sollten.

Der Lösungsansatz, der zum Erfolg führte, lagert die Binarisierung auf einen FPGA aus. Dabei bestand das Problem, die Bilddaten in der notwendigen Geschwindigkeit zur CPU zu übertragen. Nachdem die RS232-Schnittstelle, der Parallel-Port und der USB ausgeschlossen wurden, fiel die Entscheidung auf die Ethernet-Schnittstelle. Diese ist mit einer Übertragungsrate von 100 Mb/s ausreichend schnell, um die Bilddaten unkomprimiert zu übertragen. Darüber hinaus bietet diese Schnittstelle genügend Bandbreite für eine zeitgleiche Übertragung weitere Streams an.

Nachdem auch die Schnittstelle festgelegt wurde, musste eine Implementierung für die benötigten Protokolle gefunden werden. Zuvor wurde jedoch noch abgewogen, welche Protokoll-Schichten sich am besten eignen. Dabei wurde sich aus mehreren Gründen gegen TCP/IP und für UDP/IP entschieden. Ein Grund ist die benötigte bidirektionale Kommunikation, die sowohl auf Empfänger- als auch auf der Sender-Seite einen höheren Ressourcen-Bedarf gefordert hätte. Dementsprechend wären die Latenzzeiten aufgrund der komplexeren Struktur auch höher. Das Erkennen und wiederholte Versenden von verloren gegangenen Paketen würde sich zusätzlich negativ auswirken, da ein veraltetes Bild als irrelevant zu betrachten ist und das wiederholte Versenden wichtige Ressourcen belegen würde. Durch die Verwendung von UDP in Kombination mit Multicast wurde auf der FPGA-Seite keine Empfänger-Hardware benötigt.

Bei der Suche nach einer geeigneten Implementierung für die Protokoll-Schichten wurden mehrere Varianten ausgeschlossen. Dies wurde durch unnötig hohe Taktraten und Verwendung eines Speichers, der erst das komplette Paket enthalten muss, bevor mit dem Versenden begonnen werden kann, begründet. Ein weiteres Gegenargument war die Verwendung von Bus-Systemen, die die Übertragungszeiten erhöht hätten, ohne direkte Vorteile für diese Arbeit zu bieten. Schlussendlich wurde sich für eine eigens entwickelte Implementierung entschieden.

Diese Implementierung baute auf einer modifizierten Variante des "XPS Ethernet Lite MAC" von Xilinx auf. Dabei wurde ein Weg erarbeitet, das Bus-System und den Speicher, in dem immer erst ein ganzes Paket abgelegt werden muss, zu umgehen. Somit musste das UDP, das IP und das Ethernet-Protokoll implementiert werden. Diese Implementierungen wurden in Form von Schieberegistern aufgebaut. Um die Bilddaten zu kapseln, wurde des Weiteren Hardware entwickelt, die einen Sun-Raster-Header einfügt.

Die vorherige Binarisierung und Erodierung wurde aus einer an der HAW Hamburg im Jahre 2012 fertiggestellten Masterarbeit übernommen. Diese Implementierungen lagen als Modelle für den "Xilinx System Generator" vor. Dabei war zu beachten, dass die Bildverarbeitung mit einem anderen Takt als der Sender arbeitet. Dieser Taktübergang wurde von einem FIFO-Speicher übernommen. Der FIFO-Speicher wurde mit dem "Xilinx Core Generator" erstellt.

Mit Hilfe eines Simulators konnte anschließend die Funktionsweise dieser Lösung ausgiebig getestet werden. Dabei produzierte dieser Simulator ein Signal, das dem der Kamera, die später mit dem FPGA verbunden werden soll, stark ähnelt.

Im Gegensatz zum Simulator kann die Kamera über eine I^2C-Schnittstelle konfiguriert werden. Die dafür benötigten Software-Komponenten wurden aus einer Bachelorarbeit, die ebenfalls an der HAW Hamburg im Jahre 2013 entstand, übernommen. So ist mit dieser benötigten Software-Komponente und die intensive Verifikation mit hoher Wahrscheinlichkeit sichergestellt, dass die Kamera nach Anfertigung eines Adapter-Kabels für die VHDC-Schnittstelle am FPGA fehlerfrei betrieben werden kann. Das bedeutet, dass die Kamera sowie ein FPGA auf einem Fahrzeug montiert werden können. Damit steht eine hoch performante Kamera-Anbindung mit zusätzlicher Binarisierung und Erodierung zur Verfügung.

Unabhängig von einer Kamera kann die Sendeeinheit, die den Großteil dieser Arbeit ausmacht, auch zum Versenden anderer Daten verwendet werden. Um dies zu ermöglichen, kann der Bilddaten-Header gegen einen beliebigen anderen Header ausgetauscht werden. Damit wird eine Möglichkeit geschaffen, Daten über das Netzwerk sehr schnell und mit minimalen Latenzen übertragen zu können. Auf Grund der Pipeline-Architektur eignen sich hierfür Datenströme, die ein konstantes Datenaufkommen aufweisen, besonders gut. Die Grenze für das Zerteilen der Daten in einzelne Pakete kann dabei durch einen Schwellwert eingestellt werden. Bei Überschreiten dieses Schwellwerts wird von der Pipeline ein weiteres Paket selbstständig versendet. Somit wäre es zum Beispiel auch möglich, die Sendeeinheit zum Übertragen von Audio-Streams zu verwenden.

9. Ausblick

In den nachfolgenden Unterkapiteln werden mögliche Erweiterungen für diese Arbeit beschrieben. Dabei beziehen sich einige auf die Vergrößerung des Funktionsumfangs, andere befassen sich wiederum mit der Erhöhung der Performance.

9.1. Adapter für VHDC-Buchse anfertigen

In Kapitel 7.4 wurde belegt, dass die Pmod-Schnittstellen des FPGA-Boards nur für geringe Frequenzen geeignet sind. Somit kann eine Kamera nicht über diese Anschlüsse mit dem FPGA verbunden werden, weshalb sich ein Adapter für die VHDC-Buchse des FPGA-Boards anbietet. Eine mögliche Anschlussbelegung wurde schon in Tabelle 3.1 vorgestellt.

9.2. Schnellere Kamera verwenden

In Kapitel 7.2.3 wurde bewiesen, dass die entwickelte Hardware mit Bildraten über 90 Bilder pro Sekunde umgehen kann. Die bei der FRAMOS GmbH bestellte Kamera ermöglicht aber nur 33,5 Bilder bei Verwendung der maximalen Auflösung. Dies liegt an dem langsameren Quartz mit einem Takt von 13,56 MHz, der auf der Kamera verbaut wurde. Genau diese Kamera mit der gleichen Bezeichnung wird aber auch mit einem 27 MHz-Quartz angeboten. Mit dieser Variante sind laut Datenblatt Bildraten über 60 Bilder pro Sekunde möglich. Deshalb sollte eine schnellere Kamera verwendet werden oder der Quartz auf den vorhanden Kameras gegen einen schnelleren ausgetauscht werden. Da, wie schon erwähnt, beide Kameras unter der selben Bezeichnung geführt werden, ist es sehr wahrscheinlich, dass diese langsamere Variante mit einem schnelleren Quartz betrieben werden kann.

Dabei ist jedoch zu beachten, dass die aktuelle Konfiguration der Kamera teilweise durch empirische Tests entstanden ist. Somit könnte es möglich sein, dass zum Beispiel die Pausenzeiten zwischen den Zeilen und jedem Bild erhöht werden müssen, damit die Kamera zuverlässig arbeitet.

9.3. Projektiv transformiertes Bild übertragen

Eine projektive Transformation kann sehr rechenintensiv werden. Zur Zeit wird diese auf der CPU berechnet. Solange nur wenige Bildpunkte transformiert werden sollen, stellt dies auch kein größeres Problem dar. Aber wenn ein ganzes Bild transformiert werden soll, ist dies auf der verwendeten CPU in keiner akzeptablen Zeit umsetzbar. Da dem FPGA das aktuelle Bild bereits zur Verfügung steht, besteht die Möglichkeit darauf eine projektive Transformation anzuwenden. Anschließend kann über einen zweiten Stream dieses transformierte Bild zusätzlich mit an den PC übertragen werden. Nur das transformierte Bild zu übertragen würde nicht ausreichen, da unter anderem die zur Zeit verwendete Hinderniserkennung auf dem nicht transformierten Bild aufbaut.

Durch den Einbau eines Multiplexers in den modifizierten "XPS Ethernet Lite MAC" wäre eine parallele Nutzung der Sendeseite vom Bilddaten-Sender und vom MicroBlaze über den PLB möglich. Damit bestünde die Möglichkeit die Transformationsmatrix vom PC aus über Ethernet zu konfigurieren. Um eine sichere Verbindung aufbauen zu können, könnte die LwIP-Bibliothek auf dem MicroBlaze verwendet werden. Damit könnte eine Kommunikation über TCP/IP hergestellt werden.

9.4. Adaptive Binarisierung mit lokalen Schwellwerten

Die zur Zeit verwendete Binarisierung berechnet zwar ihren Schwellwert für jedes Bild neu, jedoch wird ein Schwellwert für das gesamte Bild verwendet. Zu untersuchen wäre, ob eine Binarisierung mit mehreren lokalen Schwellwerten, die zum Beispiel jeweils für einen rechteckigen Bereich gültig sind, noch bessere Ergebnisse erzielen würde. Hintergrund ist dabei die ungleichmäßige Ausleuchtung des Raumes. So fordern die hellen Wände einen viel höheren Schwellwert als die doch meist dunkle Fahrbahn.

9.5. Wiederherstellung der Paketreihenfolge

Bei einer Verbindung zwischen FPGA und PC mit einem Crossover-Kabel kann die Reihenfolge nicht durcheinander geraten. Dankbar wäre aber, den Stream vom PC über WLAN weiter leiten zu lassen. Somit bestünde die Möglichkeit diesen Stream während der Fahrt des Fahrzeuges zu begutachten, um zum Beispiel die Fehlersuche zu vereinfachen. Hierbei ist die Reihenfolge der Pakete aber nicht mehr sicher gestellt. Deshalb wäre eine Untersuchung mit Protokollen, die die Paketreihenfolge wieder herstellen können, sinnvoll. Einen Lösungsansatz würde das

“Real-Time Transport Protocol” (RTP) bieten. Dabei sollte bei der Untersuchung die Latenz zwischen Bildaufnahme und Verfügbarkeit auf dem PC im Vordergrund stehen.

9.6. Andere Kamera verwenden

Ein Großteil der Verzögerung bei der Übertragung des Bildes entsteht dadurch, dass die Kamera ihre kontinuierliche Übertragungsrate einhält. Wenn die Kamera ihr Bild nach Aufzeichnung mit einem schnelleren Takt versenden würde und anschließend bis zum nächsten Bild keine Daten übermitteln würde, könnte auch das binarisierte Bild schneller versendet werden. Wie in der Evaluation zu erkennen war, ist die Ethernet-Schnittstelle nicht ausgelastet und zwischen den einzelnen Paketen eines Bildes vergeht sehr viel Zeit. Würde das Bild schlagartig im FPGA zur Verfügung stehen, könnte es auch sehr schnell versendet werden.

A. CD-Anhang

Dieser Bachelorarbeit ist eine CD beigefügt. Diese CD enthält unter anderm den Quellcode der entwickelten und benötigten Komponenten. Dabei sind im Wurzelverzeichnis drei Ordner zu finden.

- FAUSTcore
- FAUSTplugins
- FPGA

Alle diese Ordner sind vollständige Git-Repositories und enthalten dem entsprechend alle bis zur Abgabe dieser Arbeit entwickelten Versionen und Varianten. Der Inhalt dieser Ordner wird in den folgenden Unterkapiteln beschrieben.

Des Weiteren ist eine Datei mit dem Namen "Binarization with UDPSender.vsd" im Wurzelverzeichnis abgelegt. Dies ist eine "Microsoft Visio"-Projektdatei. Sie enthält eine Übersicht über das Gesamtsystem, das in dieser Arbeit beschrieben wurde. Als Bild ist diese Grafik in Abbildung 4.1 zu finden.

A.1. FAUSTcore

Dieser Ordner enthält die Quellen für die auf dem Fahrzeug ausgeführte Kern-Software. Diese Software wurde im Laufe der Arbeit nicht verändert. Sie ist aber für den Betrieb notwendig. Die eigentliche Funktionalität wird erst durch die Einbindung von Plugins hergestellt. Diese Plugins befinden sich in einer dynamischen Bibliothek und werden zur Laufzeit verlinkt.

A.2. FAUSTplugins

Dies sind die Quellen dieser dynamischen Bibliothek. Dabei wird zwischen Treibern und Funktionen unterscheiden. Treiber stellen Anbindungen an die Außenwelt zur Verfügung. Funktionen verarbeiten diese Daten. Der in dieser Arbeit entwickelte Bilddaten-Empfänger

stellt eine Verbindung zur Außenwelt zur Verfügung und ist somit ein Treiber. Die Quellen dieser Software-Komponente sind im Unterordner "CaroloCup/Second/FAUSTplugins/Drivers/StreamReader/" zu finden.

In Kapitel 2.1.1 wurde eine Implementierung einer adaptiven Binarisierung in Software vorgestellt. Dies Methode wurde in dem vorhanden Treiber für die USB-Kamera eingefügt. Die Quellen dieses Treibers sind im Unterordner "CaroloCup/Second/FAUSTplugins/Drivers/uEye/" zu finden. Im Laufe der Arbeit wurden nur diese beiden Komponenten entwickelt oder angepasst. Andere Teile dieser Bibliothek wurden nicht verändert. Teilweise werden sie aber für den erfolgreichen Betrieb des Fahrzeuges benötigt.

A.3. FPGA

In diesem Repository befindet sich der Großteil dieser Bachelorarbeit. Dabei enthält dieser mehrere Teile der Arbeit. Im Verzeichnis "CrossingClockImageFIFO" ist der mit dem "Xilinx Core Generator" erstellte Bilddaten-Puffer zu finden. Im Ordner "Documents" sind hauptsächlich die Quellen dieses mit Latex erstellten Dokumentes abgelegt. Es sind aber auch weitere Teile wie Messergebnisse und Datenblätter anzufinden. Um einen Mittelwert über die Messergebnisse bilden zu können wurde ein Tool entwickelt und genutzt. Dieses ist unter dem Pfad "Documents/Results/calcMeanValue.pl" abgelegt. Die in VHDL implementierte Berechnung der Paketlänge wurde mit dem "Xilinx System Generator" nach gebaut, um die Grafik aus Abbildung 5.3 zu erstellen. Die entsprechende Projektdatei ist unter "Documents/PackageLenCalc/pakageLenCalc.mdl" auffindbar.

Die mit dem "Xilinx System Generator" erstellte Bildverarbeitungskette befindet sich in dem Ordner "ImageProcessing". Diese wurde, wie schon erwähnt, aus einer anderen Masterarbeit übernommen und angepasst. Eine Simulationsumgebung für ModelSim mit dem der Bilddaten-Sender getestet wurde, wurde im Verzeichnis "ImageTransmitterSim" gespeichert. Das Verzeichnis "ReadFrameBuffer" enthält die Test-Anwendung, die in Kapitel 2.2 zum Auslesen des Framebuffers der GPU verwendet wurde.

In dem Ordner "VideoStreamer" befindet sich das "Xilinx Platform Studio"-Projekt, dass die erwähnten Einzelkomponenten vereint. Das für die Initialisierung der Kamera benötigte Software-Projekt ist in dem Unterordner "VideoStreamer/SDK/" anzufinden. Ein Bit-Stream

für die Initialisierung eines "Digilent Nexys 3"-Board befindet sich unter "VideoStreamer/SD-K/VideoStreamer_hw_platform/download.bit". Dieser Bit-Stream enthält die mit dieser Arbeit entwickelte Hardware und die Software für die Kamera-Initialisierung.

Abbildungsverzeichnis

2.1. Darstellung der Pipeline-Stufen einer Grafikkarte die OpenGL 2.0 unterstützt [vgl. NFHS11a, S. 51] . . . 9

3.1. Zeitverhalten des Line-Valid-Signals der Kamera [vgl. Mic06, S. 5] . . . 15
3.2. Zeitverhalten des Line- und Frame-Valid-Signals der Kamera [vgl. Mic06, S. 5] 16

4.1. Übersicht des Gesamtsystems mit Xilinx Micro-Blaze [vgl. Kir12, S. 22] [Wis13b] 18
4.2. Adaptive Binarisierung mit Xilinx System Generator erstellt [Kir12] . . . 19
4.3. Erosionsfilter mit Xilinx System Generator erstellt [Kir12] . . . 21

5.1. Zusammenhang der einzelnen Komponenten des Bilddaten-Senders [Wis13b] 23
5.2. TCP/IP-Referenzmodell zur Veranschaulichung der Protokoll-Schichten [vgl. MBW10, S. 10] . . . 33
5.3. Übersicht über die Berechnung der Paketlänge [Wis13b] . . . 46

7.1. Zeitverhalten des Frame-Valid-Signals vom Simulator (PicoScope 3204) . . . 58
7.2. Zeitverhalten des Line-Valid-Signals des Simulators (PicoScope 3204) . . . 58
7.3. Vom Simulator generiertes Bild . . . 59
7.4. Bilddaten-Eingabe und Paketversand (PicoScope 3204) . . . 63
7.5. Ethernet-Verkehr mit erstem Paket eines neuen Bildes (Wireshark 1.10.1) . . . 64
7.6. Durch die adaptive Binarisierung verarbeitetes Bild . . . 64
7.7. Durch die adaptive Binarisierung und das Erosionsfilter verarbeitetes Bild . . 65
7.8. Verzögerung vom Beginn eines neuen Bildes bis zum Versand des ersten Paketes (PicoScope 3204) . . . 67
7.9. Verzögerung zwischen FIFO-Signalisierung und erstes Byte der Bilddaten versendet (ModelSim SE-64 6.5b) . . . 67
7.10. Verzögerung vom Beginn eines neuen Bildes bis zum Versand des letzten Bildes (PicoScope 3204) . . . 68

7.11. Verzögerung vom Beginn eines neuen Bildes bis zum vollständigen Versand dieses Bildes (PicoScope 3204) 69
7.12. Verzögerung vom Versand des letzten Paketes bis zur vollständigen Dekodierung (PicoScope 3204) 71
7.13. Latenz von Bildbeginn bis zur abgeschlossenen Dekodierung (PicoScope 3204) 71
7.14. Übertragung eines weißen Bildes der USB-Kamera (Wireshark 1.10.1) 73
7.15. Verfälschtes Taktsignal an einem Pin eines Pmod-Anschlusses gemessen (PicoScope 3204) 74
7.16. Taktsignal an einem Pin eines VHDC-Anschlusses gemessen (PicoScope 3204) 75
7.17. Frame- und Line-Valid-Signal der Kamera bei einer Auflösung von 752x480 Bildpunkten (PicoScope 3204) 76

Tabellenverzeichnis

2.1. Spezifikation der auf den CaroloCup-Fahrzeugen verbaute CPU [Com13] . . . 7
2.2. Spezifikation der auf den CaroloCup-Fahrzeugen verbaute Kamera [IDS10] . . 7
2.3. Spezifikation der auf den CaroloCup-Fahrzeugen verbauten Grafikkarte [Hin12a] 8
2.4. Für glReadPixels-Performance-Messung verwendete Hardware [Hin12b] . . . 10

3.1. Mögliche Pin-Belegung zwischen Kamera und FPGA [vgl. Sal13, S. 51] 15

5.1. Eigenschaftsfelder des Sun Raster Headers [Mv96] 31
5.2. Eigenschaftsfelder des UDP-Headers [vgl. MBW10, S. 228-229] 35
5.3. Eigenschaftsfelder des IP-Headers [vgl. MBW10, S. 119-121] 39
5.4. Eigenschaftsfelder des Ethernet-Headers [vgl. MBW10, S. 62-63] 42

7.1. Funktionstest der Hardware mit verschiedenen Takten (PicoScope 3204) . . . 66
7.2. Vergleich der Bild-Dekodierung von OpenCV und der Eigenentwicklung (Debian 7.0, GNU C Library 2.13-38, gettimeofday) 70
7.3. Auslastung der Empfänger-PC-Ressourcen (Debian 7.0, top 3.3.3) 72
7.4. Funktionstest bei Verwendung eines über einen Pmod-Pin angelegten Taktes (PicoScope 3204) 75

Listings

2.1. Binarisierung in Software [JBW13] 6
2.2. Erstellen einer Textur eines Kamerabildes [AJ12] 9
2.3. Auslesen des Framebuffers einer Grafikkarte [Wis13b] 10

5.1. Deserialisierung der Bilddaten mit Synchronisation auf den Bildanfang [Wis13b] 24
5.2. Generierung der Signalisierung für den Beginn eines neuen Bildes [Wis13b] . 25
5.3. Fehlerbehandlung für Voll- und Leer-Signal des Bilddaten-Puffers [Wis13b] . . 26
5.4. Elementzähler zur Berechnung der Bildgrenzen [Wis13b] 27
5.5. Steuerung für den Beginn einer neuen Übertragung [Wis13b] 29
5.6. Multiplexer und Zähler zum Einfügen des Bild-Headers [Wis13b] 31
5.7. Serielles Schieberegister das für die Netzwerk-Header verwendet wird [Wis13b] 35
5.8. Paralleles Schieberegister das für die Netzwerk-Header verwendet wird [Wis13b] 36
5.9. Zusammensetzen des UDP-Headers [Wis13b] 37
5.10. Leeren des UDP-Schieberegisters [Wis13b] 38
5.11. IP-Header berechnen [Wis13b] 41
5.12. Hinzugefügte Schnittstelle für "XPS Ethernet Lite MAC" [Wis13b] 44
5.13. Wrapper zwischen Schieberegistern und "XPS Ethernet Lite MAC" [Wis13b] . 45

6.1. Öffnen des UDP-Multicast-Streams [Wis13a] 48
6.2. Synchronisation der Bildgrenzen des Bilddatenstroms [Wis13a] 51
6.3. Bildlänge berechnen und fehlende Daten einlesen [Wis13a] 53
6.4. Schnelle Sun-Raster-Bild-Dekodierung [Wis13a] 55

7.1. Zähler-Logik für einen Bildpunkt [Wis13b] 60
7.2. Zähler-Logik für Zeilen und Bilder [Wis13b] 60
7.3. Generierung des Frame-Valid-Signals [Wis13b] 61
7.4. Bild-Generierung mit Hilfe der Zähler [Wis13b] 61

Literaturverzeichnis

[BT06] Bastian, Peter ; Tkotz, Klaus: *Fachkunde Elektrotechnik.* 25. Aufl. Haan-Gruiten : EUROPA-LEHRMITTEL, 2006. – ISBN 3–8085–3159–2

[Erh08] Erhardt, Angelika: *Einführung in die Digitale Bildverarbeitung.* Wiesbaden : Vieweg+Teubner, 2008. – ISBN 978–3–519–00478–3

[Jen08] Jenning, Eike: *Systemidentifikation eines autonomen Fahrzeugs mit einer robusten, kamerabasierten Fahrspurerkennung in Echtzeit.* Hamburg, Hochschule für Angewandte Wissenschaften, Masterarbeit, 2008. `http://opus.haw-hamburg.de/volltexte/2009/742/pdf/Masterarbeit_Jenning.pdf`, Abruf: 2013-01-07

[Kil07] Kilts, Steve: *Advanced FPGA Design.* Hoboken, New Jersey : John Wiley & Sons Inc., 2007. – ISBN 978–0–470–05437–6

[Kir12] Kirschke, Marco: *FPGA-basierte MPSoC-Plattform zur Integration eines Antikollisionssystems in die Fahrspurführung eines autonomen Fahrzeugs.* Hamburg, Hochschule für Angewandte Wissenschaften, Masterarbeit, Juli 2012. `http://opus.haw-hamburg.de/volltexte/2012/1769/pdf/ma_kirschke.pdf`, Abruf: 2013-05-07

[LSH05] Lodesten, Lucas ; Sjöholm, Stefan ; Hansson, Hans: An analysis of FPGA-based UDP/IP stack parallelism for embedded Ethernet connectivity. In: Löfgren, Andreas (Hrsg.): *NORCHIP Conference.* Vasteras, Sweden : Mälardalen University, Oktober 2005 (23rd). – ISBN 1–424–40064–3, 94–97

[MBW10] Mandl, Peter ; Bakomenko, Andreas ; Weiß, Johannes: *Grundkurs Datenkommunikation.* 2. Aufl. Wiesbaden : Vieweg+Teubner, 2010. – ISBN 978–3–8348–0810–3

[Mv96] Murray, James D. ; vanRyper, William ; Russell, Deborah (Hrsg.): *Encyclopedia of Graphics File Formats.* 2. Aufl. California : O'Reilly Media, 1996. – 1152 S. – ISBN 978–1–56592–161–0

[NFHS11a] Nischwitz, Alfred ; Fischer, Max ; Haberäcker, Peter ; Socher, Gudrun: Band I: Computergrafik. In: *Computergrafik und Bildverarbeitung* Bd. 1. 3. Aufl. Wiesbaden : Vieweg+Teubner, 2011. – ISBN 978–3–658–00777–5, S. 1–532

[NFHS11b] Nischwitz, Alfred ; Fischer, Max ; Haberäcker, Peter ; Socher, Gudrun: Band II: Bildverarbeitung. In: *Computergrafik und Bildverarbeitung* Bd. 2. 3. Aufl. Wiesbaden : Vieweg+Teubner, 2011. – ISBN 978–3–658–00777–5, S. 533–1000

[PPOL02] Peek, Jerry ; Powers, Shelley ; O'Reilly, Tim ; Loukides, Mike: *UNIX Power Tools*. 3. Aufl. California : O'Reilly Media, 2002. – 1156 S. – ISBN 978–0–596–00330–2

[Sal13] Salathé, Jasper: *Integration einer CMOS-Kamera mit parallelem Interface in ein System-on-Chip basiertes Spurführungssystem*. Hamburg, Hochschule für Angewandte Wissenschaften, Bachelorarbeit, Januar 2013. `http://opus.haw-hamburg.de/volltexte/2013/1926/pdf/BA_Salathe.pdf`, Abruf: 2013-07-10

[Uch07] Uchida, Tomohisa: Hardware-based TCP processor for Gigabit Ethernet. In: University, Tokyo (Hrsg.): *Nuclear Science Symposium Conference Record*. Honolulu, HI, November 2007. – ISBN 978–1–4244–0922–8, 309–315

Onlinequellen

[ABS10] Alachiotis, Nikos ; Berger, Simon A. ; Stamatakis, Alexandros: *UDP/IP Core*. http://opencores.org/project,udp_ip__core. Version: Februar 2010, Abruf: 2013-08-08

[AJ12] Aurisano, Jillian ; Johnson, Andy: *GPU Programming Project 1: GLSL Webcam shading*. https://sites.google.com/site/jaurisano/portfolio/cs-525-gpu-programming/glsl-webcam-shaders. Version: Februar 2012, Abruf: 2013-05-22

[Bec13] Becker, Torben: *FAUST News: Streckenkartografierung auf Grundlage des xBIMU*. http://faust.informatik.haw-hamburg.de/index.php?section=news&news_id=34. Version: Mai 2013, Abruf: 2013-07-23

[Com13] Commell: *Commell LP-172D Datasheet*. 1, Januar 2013. – 2 S. http://www.anteor.com/catalog/product_info.php?products_id=1034, Abruf: 2013-05-21

[Dee89] Deering, Steve: RFC 1112: Host Extensions for IP Multicasting / Computer Science Department, Stanford University. Version: August 1989. http://tools.ietf.org/html/rfc1112, Abruf: 2013-08-04. Network Working Group, August 1989 (1). – Spezifikation

[Dig11] Digilent: *Nexys3 BSB Support Files for PLB-based Designs*. 1, Oktober 2011. – 6 S. http://www.digilentinc.com/Data/Products/NEXYS3/Nexys3_BSB_Support_v_2_7.zip, Abruf: 2013-08-04

[Dig12] Digilent: *Atlys Board Reference Manual*. 1, Dezember 2012. – 22 S. http://www.digilentinc.com/Data/Products/ATLYS/Atlys_rm.pdf, Abruf: 2013-07-25

[Dig13] Digilent: *Nexys3 Board Reference Manual* . 1, Januar 2013. – 22 S. http://www.digilentinc.com/Data/Products/NEXYS3/Nexys3_rm.pdf, Abruf: 2013-07-25

[F.13] F., Claus: *Spartan6 on Nexys3 Board - Ethernet communication problems.* http://forums.xilinx.com/t5/Embedded-Processors-and/Spartan6-on-Nexys3-Board-Ethernet-communication-problems/m-p/292473#M7246. Version: Juli 2013, Abruf: 2013-08-04

[Hin12a] Hinum, Dipl.-Ing. Dr. Klaus A.: Intel Graphics Media Accelerator 3650 Vergleich. Version: Juli 2012. http://www.notebookcheck.com/Intel-Graphics-Media-Accelerator-GMA-3650.60194.0.html, Abruf: 2013-05-21. Notebookcheck Publishing GmbH, Juli 2012 (1). – Forschungsbericht

[Hin12b] Hinum, Dipl.-Ing. Dr. Klaus A.: Intel HD Graphics 3000 Vergleich. Version: November 2012. http://www.notebookcheck.com/Intel-HD-Graphics-3000.37907.0.html, Abruf: 2013-05-22. Notebookcheck Publishing GmbH, November 2012 (1). – Forschungsbericht

[IDS10] IDS: *IDS UI-1221LE-M-GL Datasheet.* 1, 2010. – 1 S. http://de.ids-imaging.com/IDS/spec_pdf.php?sku=AB.0010.1.25700.23, Abruf: 2013-05-21

[JBW13] Jenning, Eike ; Becker, Torben ; Wischer, Timo: *UEyeDriver.* http://faust.informatik.haw-hamburg.de/gitweb/?p=FAUSTplugins.git;f=CaroloCup/Second/FAUSTplugins/Drivers/uEye/UEyeDriver.cpp;hb=52d89dfebea95be4c1cee719eb8af7146ff91d31. Version: Mai 2013, Abruf: 2013-05-21

[JED07] JEDEC: JESD8C.01: Interface Standard for Nominal 3 V/3.3 V Supply Digital Integrated Circuits. Version: September 2007. http://www.jedec.org/sites/default/files/docs/jesd8c-01.pdf, Abruf: 2013-08-08. JEDEC Solid State Technology Association, September 2007 (1). – Spezifikation. – 16 S.

[Mic06] Micron: *Micron MT9V022 Datenblatt.* 1, 2006. – 15 S. http://datasheet.seekic.com/PdfFile/MT9/Micron_MT9V022IA7ATM08678.pdf, Abruf: 2013-07-16

[MK02] Mohor, Igor ; Kindgren, Olof: *EthMAC.* http://opencores.org/project,ethmac. Version: 2002, Abruf: 2013-06-13

[Ope13a] OpenCV: *OpenCV 2.4.6.0 documentation: Basic Structures.* 1, Juli 2013. http://docs.opencv.org/modules/core/doc/basic_structures.html, Abruf: 2013-08-05

[Ope13b] OpenCV: *OpenCV 2.4.6.0 documentation: Reading and Writing Images and Video.* 1, Juli 2013. http://docs.opencv.org/modules/highgui/doc/reading_and_writing_images_and_video.html?#imread, Abruf: 2013-08-05

[Pos81] Postel, Jon: RFC 791: Internet Protocol. Version: September 1981. http://tools.ietf.org/html/rfc791, Abruf: 2013-08-04. Network Working Group, September 1981 (1). – Spezifikation

[Pos83] Postel, Jon: RFC 879: The TCP Maximum Segment Size and Related Topics. Version: November 1983. http://tools.ietf.org/html/rfc879, Abruf: 2013-05-27. Network Working Group, November 1983 (1). – Spezifikation

[SA06] Segal, Mark ; Akeley, Kurt: *OpenGL SDK Documentation: glReadPixels.* 1, 2006. http://www.opengl.org/sdk/docs/man/xhtml/glReadPixels.xml, Abruf: 2013-07-25

[Wis13a] Wischer, Timo: *Stream-Reader Release 1.0.* http://faust.informatik.haw-hamburg.de/gitweb/?p=FAUSTplugins.git;f=CaroloCup/Second/FAUSTplugins/Drivers/StreamReader/StreamReader.cpp;hb=1db016217b398e818b625ce2615c96a1a8be911e. Version: August 2013, Abruf: 2013-05-22

[Wis13b] Wischer, Timo: *Video-Streamer Release 1.0.* http://faust.informatik.haw-hamburg.de/gitweb/?p=FPGA.git;a=tree;h=refs/tags/VideoStreamerRelease1.0. Version: August 2013, Abruf: 2013-05-22

[Xil06] Xilinx: *Spartan-3E Starter Kit Board User Guide.* 1, März 2006. – 164 S. http://www.digilentinc.com/Data/Products/S3EBOARD/S3EStarter_ug230.pdf, Abruf: 2013-07-25

[Xil10] Xilinx: *LogiCORE IP XPS Ethernet Lite Media Access Controller.* 1, September 2010. – 42 S. http://www.xilinx.com/support/documentation/ip_documentation/xps_ethernetlite.pdf, Abruf: 2013-06-13

[Xil13] Xilinx: *UG382: Spartan-6 FPGA Clocking Resources.* 1, Juni 2013. – 116 S. http://www.xilinx.com/support/documentation/user_guides/ug382.pdf, Abruf: 2013-07-26

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Pröfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 30. August 2013	
Ort, Datum	Timo Wischer