Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterthesis

Raphael Kemmler

High-Level-Architektursynthese für die Implementierung einer Fingerabdruckmerkmalsextraktion auf einem MPSoC

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Raphael Kemmler

High-Level-Architektursynthese für die Implementierung einer Fingerabdruckmerkmalsextraktion auf einem MPSoC

Masterthesis eingereicht im Rahmen der Masterprüfung

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

Betreuender Prüfer: Prof. Dr.-Ing. Bernd Schwarz
Zweitgutachter: Prof. Dr.-Ing. Andreas Meisel

Eingereicht am: 20. Oktober 2016

Raphael Kemmler

Thema der Arbeit

High-Level-Architektursynthese für die Implementierung einer Fingerabdruckmerkmalsextraktion auf einem MPSoC

Stichworte

Bildverarbeitung, AFIS, Minutienextraktion, High-Level-Synthese, Hochsprachentransformation, Automatische RTL-Architektursynthese, MPSoC, FPGA, AXI, VDMA, Zedboard

Kurzzusammenfassung

Diese Masterarbeit beschreibt die Entwicklungsschritte für eine Bildverarbeitung zur Minutienextraktion auf einem FPGA-SoC. Schwerpunkt ist die Transformation der C++ basierten Referenzimplementierung der Bildverarbeitung mit einer High-Level-Synthese auf das Zynq-SoC. Die Binarisierung und Segmentierung aus der Bildverarbeitungskette der Minutienextraktion wurde für eine prototypische Implementierung auf dem FPGA mit der High-Level-Synthese ausgewählt. Mit dieser automatischen Architektursynthese wurde ein IP-Core entwickelt, der in einem Gesamtsystem mit einer AXI-VDMA-Komponente integriert wurde und von einer Bare-Metal-Applikation auf dem ARM-Prozessor gesteuert wurde.

Title of the paper

High-level synthesis for the implementation of a fingerprint minutiae extraction on a MPSoC

Keywords

Image Processing, AFIS, Minutiae Extraction, High-Level Synthesis, Automatic RTL Architecture Synthesis, MPSoC, FPGA, AXI, VDMA, Zedboard

Abstract

This master thesis describes the development steps for an image processing algorithm for minutiea detection on an embedded system. It emphasises on the transformation of the C++ reference implementation with a high-level synthesis to the Zynq-SoC. The binarization and segmentation was chosen for a prototypical implementation on the FPGA with the high-level synthesis. With this automatic architecture synthesis an IP core was developed, wich was integrated in an overall system with an AXI VDMA component and is controlled by a bare metal application on the ARM processor.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Authentifizierung und Identifizierung mit biometrischen Merkmalen gehört durch fortschreitende Technisierung der Gesellschaft und Miniaturisierung der Technologien bei vielen Menschen zum Alltag und führt zu einer steigenden Akzeptanz [Bit16]. Das Smartphone wird durch einen Gesichts-Scan, einen Fingerabdruck-Scan oder durch einen Iris-Scan entsperrt und das Passbild muss „biometrisch“ sein. Internetplattformen bieten, meist optional, eine Identifizierung von Personen durch Gesichtserkennung auf hochgeladenen Fotos an [Küh15]. Die zur Messung geeigneten biometrische Merkmale sind vielfältig, dazu gehören:

- Der Fingerabdruck
- Die Gesichtsgeometrie
- Die Iris
- Die Handvenenstruktur
- Die Stimme
- Der Herzrhythmus

Der Fingerabdruck ist bei jedem Menschen einzigartig, auch bei eineiigen Zwillingen und verändert sich über die Zeit kaum. Diese Eigenschaften machen ihn seit vielen Jahrzehnten zum wichtigen Gegenstand für die Identifizierung von Menschen, insbesondere bei der Verbrechensbekämpfung. Vor der Verfügbarkeit von digitalen Systemen wurden Fingerabdrücke in zeitintensiver Arbeit von einem Menschen manuell verglichen. Mit der Technologisierung und der fortschreitenden Leistungsfähigkeit der digitalen Systeme in den letzten Jahrzehnten wurde diese Arbeit mit Computersystemen automatisiert. Diese Automatisierten Fingerabdruck-Identifizierungssysteme (AFIS) können die Fingerabdrücke in einem Bruchteil der Zeit identifizieren und an vielen verschiedenen Orten eingesetzt werden, etwa für die Zugangskontrolle zu sicherheitsrelevanten Abschnitten in Flughäfen. Die Identifizierung mit Fingerabdrücken gehört mit einem Marktanteil von über 40% zu den verbreitetsten Verfahren der biometrischen Identifikation [Ker06]. Für staatliche Anforderungen an zuverlässige Fingerabdruckerfassungssysteme gibt es von der zuständigen Behörde in Deutschland, dem Bundesamt für Sicherheit in der Informationstechnik (BSI), Spezifikationen, die Parameter bezüglich der Qualität, Auflösung und Bildverarbeitung definieren [BSI04].

In kompakten Embedded-Systemen, wie zum Beispiel in der Bildverarbeitung von Digitalkameras, kommen aufgrund der hohen Anforderungen an den Pixeldurchsatz FPGAs (Field Programmable Gate Array), ASICs (Application-Specific Integrated Circuit) und DSPs (Digital Signal Processor) zum Einsatz. Diese ICs mit einem großen Angebot an Arithmetikressoucen implementieren eine anwendungsspezifische Architektur und realisieren parallele Rechenströme mit Taktfrequenzen, die geringer als bei Standard-Prozessoren sind. Ein FPGA ist, im Gegensatz zu einem ASIC, rekonfigurierbar und liefert eine ähnliche Flexibilität wie Softwaresysteme.

Funktionalitäten für FPGA-Module, den Intellectual-Property(IP)-Cores, werden in eingebetteten, FPGA-basierten Systemen mit Hardwarebeschreibungssprachen wie VHDL und Verilog implementiert. Die mit diesem Entwurfsweg einhergehende Implementierungskomplexität resultiert in einer, im Vergleich zu Softwareimplementierungen mit Hochsprachen, langen Entwicklungszeit. Zu deren Verkürzung stehen mittlerweile Werkzeuge zur Transformation von Hochsprachenmodellen in Register-Transfer-Level (RTL)-Modelle bereit.

Im Projekt „Biometric Graphic Acceleration“ der HAW Hamburg wird für den Fingerabdruckscanner LF10 von Dermalog [Der13] ein System zur Identifikation von Fingerabdrücken auf einem rekonfigurierbaren Multiprozessor-System-On-Chip (SoC) entwickelt (vgl. Abbildung 1.1). Der LF10-Sensor bietet eine Lebenderkennung und simultane Aufnahme von bis zu vier Fingerkuppen. Das Zynq-7000-SoC [Xil13a] kombiniert FPGA-Ressourcen mit einem ARM-basierten Dualcore-Prozessor und kommt auf dem Zedboard [Avn14] zum Einsatz. Eine Implementierung auf einem eingebetteten System bietet gegenüber der verbreiteten Verarbeitung auf einem Workstation-Computer die Vorteile des geringeren Platz- und Energiebedarfs.

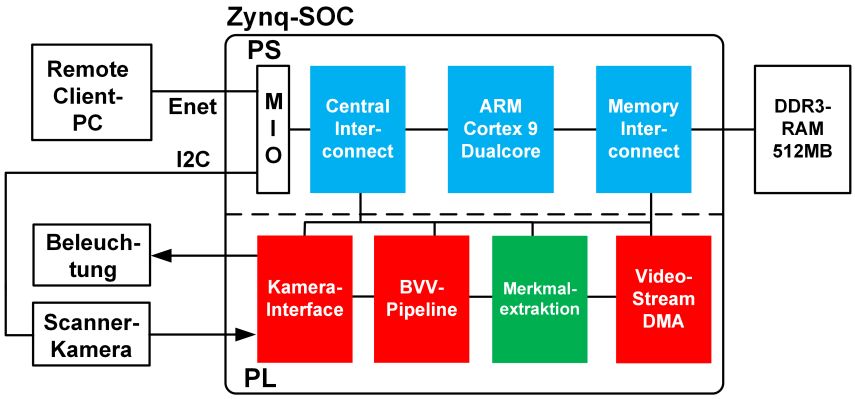


Abbildung 1.1: Prozessormodule (PS, Processing System), rekonfiguerbare Module (PL, Programmable Logic) und externe Komponenten (Kamera, RAM etc.) für das Fingerabdruckscannersystem

Die Anbindung des Scanner-Sensors an das SoC und die Bildvorverarbeitung (BVV) wurden in Abschlussarbeiten [Web14][Bla16] realisiert. Der nächste Schritt in der Ver-

arbeitungskette ist die Merkmalsextraktion und die zugehörige Vorverarbeitung der Fingerabdruckaufnahmen. Die Verarbeitungsschritte der Merkmalsextraktion wurden im Rahmen dieser Arbeit recherchiert und mit OpenCV in C++ implementiert, sodass eine auf C++ basierende Referenzspezifikation für die Verarbeitung mit der High-Level-Synthese vorlag. Die durchgeführten Entwicklungs- und Transformationsschritte vom C++-Code zur SoC-Implementierung zeigt Abbildung 1.2.

1. Automatisches Fingerabduck-identifikations-system	**2. C++-Implementierung**	**3. OpenCV-Code Transformation für HLS**
• Binarisierung • Morphologische Operationen zur Ridge- und Valleyoptimierung • Skelettierung • Merkmalsextraktion • Merkmalsreduktion	• OpenCV-Bibliothek • Funktionsnachweis mit Testbildern	• Bibliothekssubstitution • Interfacing für Datenströme • HLS-Ergebnis: Prozesssequenz, VHDL-Code

4. HLS-Verifikation	**5. SoC-Integration**	**6. Softwarekonfiguration des ARM-Prozessorsystems**	**7. In-System-PL-Debugging**
• C++-Testbench für Bild Eingang/Ausgang • C-Simulation • C/VHDL-Co-Simulation • Ergebnisreport • IP-Core Export	• Systemintegration des IP-Cores • PL-Konfiguration • Stream-to-Memory-Mapped • Verifikation • Synthese	• Initialisierung der VDMA- und Interruptkomponenten der Programmable Logic • Speicherreservierung • Steuerung der FPGA-Komponenten	• Integrated Logic Analyzer (ILA) • Virtual Input/Output (VIO)

Abbildung 1.2: Entwicklungsstrang des Masterprojektes

Ziel der Arbeit ist es, die C++-Referenzimplementierung ohne funktionale Veränderung über die High-Level-Synthese auf das Zynq-FPGA zu portieren. Ein Schwerpunkt ist die beispielhafte Anpassung eines Bildverarbeitungsmoduls für den Prozess der High-Level-Synthese (HLS). Weitere Schwerpunkte bilden die Integration des generierten IP-Cores in ein FPGA-SoC mit Anbindung zum externen RAM über eine VDMA-Komponente und die ARM-Implementierung zur Steuerung der FPGA-Hardware.

Kapitelübersicht

In Kapitel 2 wird ein Überblick über die verwendeten Bildverarbeitungsgrundlagen und FPGA-SoC-Technologien dieser Arbeit gegeben. Das umfasst Automatisierte Fingerabdruck-Identifizierungssysteme (AFIS), die Bildvorverarbeitung und die Biometrie mit Fingerabdrücken. Die SoC-Plattform, die Vorgehensweise der High-Level-Synthese und die DMA- und Kommunikationskomponenten werden beschrieben.

Die Verarbeitungsschritte der entwickelten C++-Module, die die Fingerabdruckmerkmale identifizieren, werden in Kapitel 3 dargestellt. Die Implementierung der sechs Bildverarbeitungsschritte, Binarisierung/Segmentierung, Optimierung, Skelettierung, Minutienextraktion und -reduktion, werden dokumentiert.

Kapitel 4 stellt die durchgeführten Abläufe der High-Level-Synthese und die erforderlichen Anpassungen der C++Implementierung dar. Dazu gehören Bibliotheksanpassungen, das AXI-Interfacing und das Modul-Handshaking sowie das Zeilenbuffering der Bilddaten. Das Syntheseergebnis als VHDL-Top-Entity für das Architekturmodell wird mit einer C-Testbench verifiziert.

Kapitel 5 legt dar, wie die Integration des HLS-Ergebnisses in das FPGA-System implantiert wird. Die Konfiguration des digitalen Systems durch eine Prozessor-Applikation auf dem ARM A9 Dualcore wird mit einer Ablaufübersicht beschrieben.

Die Untersuchungen zu den Ergebnissen der Arbeit werden in Kapitel 6 dokumentiert. Das umfasst die Ergebnisanalyse der High-Level-Synthese, das Ergebnis der Systemintegration sowie den Stand des Projektes und einen Ausblick.

2 Technologieübersicht zu Fingerabdruckidentifikationssystemen und der High-Level-Synthese

Dieses Kapitel gibt einen Einblick in die verwendeten Bildverarbeitungsgrundlagen und FPGA-SoC-Technologien. Die Grundlagen von automatischen Fingerabduck-Identifikationssystemen, die verfügbaren Technologien zur Merkmalsextraktion und der Ablauf der Merkmalsextraktion werden erläutert. Die SoC-Plattform und der Ablauf der High-Level-Synthese sowie die FPGA-Komponenten für die Systemintegration werden erläutert.

2.1 Automatisierte Fingerabdruck-Identifizierungssysteme

Ein Fingerabdruck ist die rauhe Oberfläche der Fingerkuppen, die aus filigranen Erhebungen (Ridges) und Tälern (Valleys) besteht. Die Erhebungen werden als Pappilarleisten bezeichnet und ergeben durch ihre Anordnung ein Muster, das wiederkehrende Merkmale (Minutien, engl. Minutiae) enthält.

(a) Gabelung einer Papillarleiste (Bifurcation)

(b) Ende einer Papillarleiste (Ridge-Ending)

Abbildung 2.1: Grundmerkmale eines Fingerabdrucks

Alle Fingerabdrücke enthalten zwei grundlegende Merkmale: die Gabelung und das Ende einer Papillarleiste (Abbildung 2.1) [MJ09]. Die beiden Grundmerkmale können spatial isoliert sein, oder, durch unmittelbare Nachbarschaft zueinander, komplexere Merkmale ergeben. Zwei nahe gelegene Ridge-Endings auf einer Papillarleiste können beispielsweise eine „Insel“ ergeben (Abbildung 2.2).

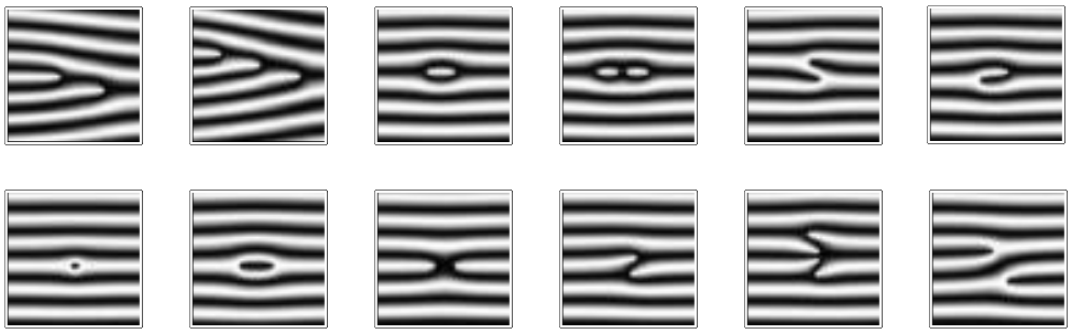


Abbildung 2.2: Komplexere Merkmale, die sich aus den beiden Grundmerkmalen zusammen setzen [BSI04] (Beschreibung vgl. Anhang Abbildung 1)

Die Grundmerkmale der Ridges haben eine inverse Entsprechung bezogen auf die Valleys und können je nach Bezug, Ridge oder Valley, umgewandelt werden. Eine Bifurcation einer Ridge ist das Ende eines Valleys und ein Ridge-Ending einer Ridge ist eine Bifurcation eines Valleys.

Ein AFIS bietet zwei Betriebsvarianten. Entweder es wird ein neuer Fingerabdruck registriert und in einer Datenbank gespeichert (Registrierungsphase), oder ein Fingerabdruck wird mit bereits vorhandenen Datenbankeinträgen abgeglichen, um dem Anwender Zutritt bzw. Zugriff zu gewähren (Authentifizierungsphase). Jede Betriebsvariante arbeitet mehrere Unterschritte ab (Abbildung 2.3).

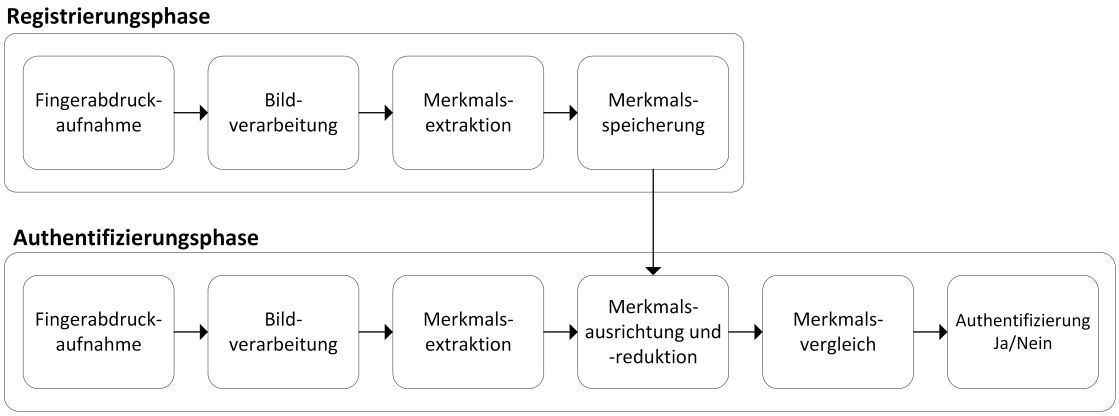


Abbildung 2.3: Arbeitsphasen eines AFIS

Die Registrierungsphase beginnt mit der Aufnahme des Fingerabdrucks durch einen geeigneten Sensor. Das kann beispielsweise ein optischer, kapazitiver oder Ultraschallbasierter Sensor sein. Die Rohdaten des Sensors werden mit einer Bildvorverarbeitungspipeline aufbereitet, um den Einfluss des Sensors auf die Aufnahme zu minimieren. Die Merkmalsextraktion verarbeitet die Fingerabdruckaufnahme, sodass mit einem

Algorithmus Bifurcations und Ridge-Endings gefunden werden. Die gefunden Merkmale werden gespeichert, womit die Registrierungsphase abgeschlossen ist.

Die Authentifizierungsphase verläuft für die ersten Schritte analog zur Registrierung. Nachdem die Merkmale identifiziert wurden, werden sie mit bereits registrierten Daten abgeglichen. Die Merkmale werden zueinander ausgerichtet, da Fingerabdruckaufnahmen des selben Abdrucks im Winkel oder in der Position verschoben sein können. Die Merkmalsreduktion eliminiert fälschlicherweise erkannte Merkmale (false-positives), um die Zuverlässigkeit zu erhöhen. False-positives entstehen durch eine ungünstige Aufnahmequalität, beispielsweise durch verdreckte Aufnahmeflächen, oder durch Artefakte, die in der Bildverarbeitung entstehen können. Das Matching analysiert, ob der neue Fingerabdruck mit einem der bereits gespeicherten übereinstimmt. Das passiert mit einem Schwellenwert, der eine zuverlässige Aussage über die Kongruenz der Fingerabdrücke trifft. Abschließend wird das Ergebnis dem Benutzer mitgeteilt und gegebenenfalls die verbundenen Mechanismen ausgelöst, zum Beispiel das Entsperren einer Tür.

2.1.1 Verarbeitungsschritte der Bildvorverarbeitung

Die bereits für das FPGA-SoC realisierte Bildvorverarbeitung (BVV, vgl. Abb. 2.4) eliminiert spezifische Bildeinflüsse des Fingerabdruckscanners. Die Subtraktion minimiert Umgebungslichteinflüsse, indem von einer NIR(near infrared)-beleuchteten Aufnahme eine unbeleuchtete Aufnahme subtrahiert wird. Der Bayer-Weißabgleich verringert das schachbrettartige Muster der Rohdaten, das durch die RGB-Filter der Subpixel des Farbbildsensors entsteht [Web14].

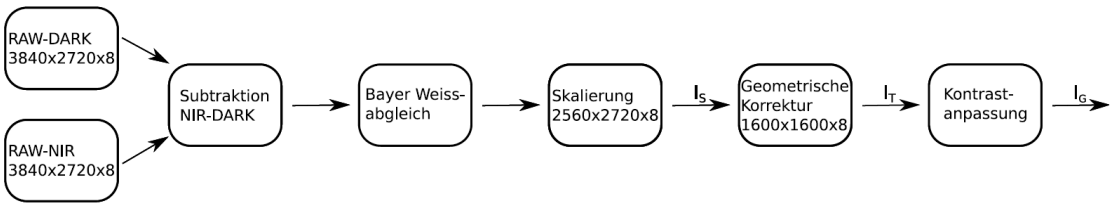


Abbildung 2.4: Bildvorverarbeitungskette, realisiert in [Web14] und [Bla16]

Die Skalierung staucht das Bild auf ein annähernd quadratisches Bildformat und setzt die Auflösung herab, wodurch weniger FPGA-Ressourcen in der weiteren Verarbeitung beansprucht werden. Die geometrische Korrektur gleicht die durch die Prismen und Linsen des Scanners entstandene Bildverzerrungen aus und bringt das Bild auf eine Auflösung von 500 DPI (Dots Per Inch), was bei dem LF10 einer Auflösung $1600 \cdot 1600$ Pixeln entspricht [Bla16]. Der letzte Schritt in der BVV ist eine Kontrastanpassung durch eine lokale Grauwertnormalisierung, womit eine ungleichmäßige Beleuchtungsverteilung gesenkt wird.

2.1.2 Verfahrensübersicht zur Minutien-Extraktion

Für die Minutienextraktion existieren zwei grundlegende Herangehensweisen:

- Direct-Grey-Scale-Extraction (DGSE)
- Extraktion aus binärem Bild

Mit DGSE werden Algorithmen bezeichnet, die Minutien ohne spezielle Bildverarbeitung (Binarisierung und Skelettierung) direkt im Grauwertbild identifizieren. Diese basieren beispielsweise auf Ridge-Following [MM97] oder Topologie-Extraktion [WP93]. Die DGSE ist vergleichsweise jung und komplex und wird in Standards für die Repräsentation und Speicherung von Fingerabdrücken, wie zum Beispiel ANSI/NIST-ITL 1 [ANS11] oder ISO/IEC 19794-2 [ISO11], nicht berücksichtigt [MJ09, S. 155].

Aus diesem Grund wird in dieser Arbeit die Extraktion aus einem binarisierten und skelettierten Bild verwendet.

Die Extraktion der Minutien aus einem binarisierten Bild besteht aus mehreren Bildverarbeitungsstufen:

1. Binarisierung des Quell-Grauwertbildes
2. Optimierung des Binarisierten Bildes, beispielsweise durch Smoothing der Ridges oder die Schließung von Poren auf Ridges
3. Skelettierung des binarisierten Bildes
4. Durchführung der Extraktion auf dem skelettierten Bild
5. Optional: Minutienreduktion durch das Löschen von falsch-positiven Minutien

Existierende Algorithmen zu den einzelnen Bildverarbeitungsschritten wurden im Masterprojekt [Kem15a] beleuchtet. Auf dieser Grundlage wurden folgende Algorithmen für die Verarbeitungsstufen selektiert:

1. Lokale-Schwellwert-Binarisierung. Das Bild wird in gleiche Flächen (Tiles) unterteilt und zu jedem Tile wird ein Schwellenwert berechnet und angewendet
2. Anwendung mathematischer Morphologie (Dilatation und Erosion) zur Schließung von Poren und Unterbrechungen von Ridges [Ike+02]
3. Skelettierung nach Guo et al. [GH89]
4. Crossing-Number-Verfahren zur Identifizierung von Bifurcations und Ridge-Endings
5. Spatiale Filterung der Minutien

Diese Algorithmen und deren Implementierung werden in Kapitel 3 ausführlicher dokumentiert.

2.2 Zedboard: MPSoC mit FPGA-Kapazitäten

Das Zedboard ist eine Zynq-Entwicklerplattform und bietet als Hauptkomponente ein SoC aus der Zynq-7000-Produktfamilie von Xilinx [Avn14]. Das Z-7020 SoC [Xil13a] bietet als Prozessorsystem einen ARM Dualcore Cortex-A9 mit bis zu 866 MHz Taktfrequenz und als programmierbare Logik ein Artix-7 FPGA, das 85.000 logische Zellen besitzt. Softwareapplikationen werden auf Basis eines Betriebssystems, zum Beispiel Linux, oder „bare-metal“ ohne Betriebssystem auf dem SoC ausgeführt. Eine Kombination aus beiden Varianten ist durch Asymmetric Multi Processing (AMP) realisierbar [Xil13b].

Neben dem SoC kommen 512 MB DDR3-Ram sowie I/O-Schnittstellen zum Einsatz, darunter Klinke für Audio, HDMI und VGA für Video, Gigabit-Ethernet für Netzwerk und USB mit OTG, JTAG und UART Unterstützung. Als Speicherort für ein Betriebssystem steht ein SD-Speicherkartenschacht zur Verfügung. Ein kleines OLED-Display, Switches, Buttons, Pmods und andere Schnittstellen sind vorhanden. Abbildung 2 im Anhang zeigt ein Bild der Plattform, in dem die wichtigsten Komponenten markiert sind.

Peripherieschnittstellen stehen dem Prozessorsystem über den Central Interconnect zur Verfügung und werden über ein MIO (Multiplexed Input Output) zu den, in der Anzahl limitierten, SoC-Pins geroutet (vgl. Abbildung 2.5).

Über den EMIO (Extended Multiplexed Input Output) wird die Peripherie zur Programmable Logic (PL) geroutet. Interruptleitungen von der Peripherie und der PL sind mit dem GIC (Global Interrupt Controller) verbunden, sodass sie von den Applikationsprozessoren verarbeitet werden können. Die PL ist über einen Memory-Interconnect mit dem Memory Controller des externen RAMs verbunden, wodurch DMA (Direct Memory Access) realisiert wird.

2.3 High-Level-Synthese zur FPGA-Implementierung der Bildverarbeitung

Die Entwicklung der High-Level-Synthese reicht bis in die 1970er Jahre zurück, als erste Forschungen dazu publiziert wurden. Die erste Generation (1980er bis frühe 1990er Jahre) und die zweite Generation (Mitte 1990er bis frühe 2000er Jahre) der High-Level-Synthese beschränkte sich auf die Forschung oder hatten keinen kommerziellen Erfolg. Die Zielgruppe wurde falsch eingeschätzt, die Eingabesprachen waren zu kompliziert und die Ergebnisse konnten technisch nicht überzeugen. Erst die dritte Generation (2000er Jahre bis heute) kann man als erfolgreich bezeichnen [MS09]. Xilinx kaufte den Hersteller eines Tools der dritten Generation auf und ließ deren Produkt in *Vivado HLS* [Xil13c] aufgehen, welches sich bis heute in ständiger Weiterentwicklung befindet und die Grundlage für die Transformation von C++-Code zu IP-Cores in dieser Arbeit ist.

Abgesehen von Xilinx gibt es viele andere Entwickler von HLS-Werkzeugen: Altera, ebenfalls ein FPGA-Hersteller, bietet ein OpenCL-to-FPGA-Framework, das den OpenCL-Standard nutzt [Atm14]. Der OpenCL-Standard ist auf parallele Programmie-

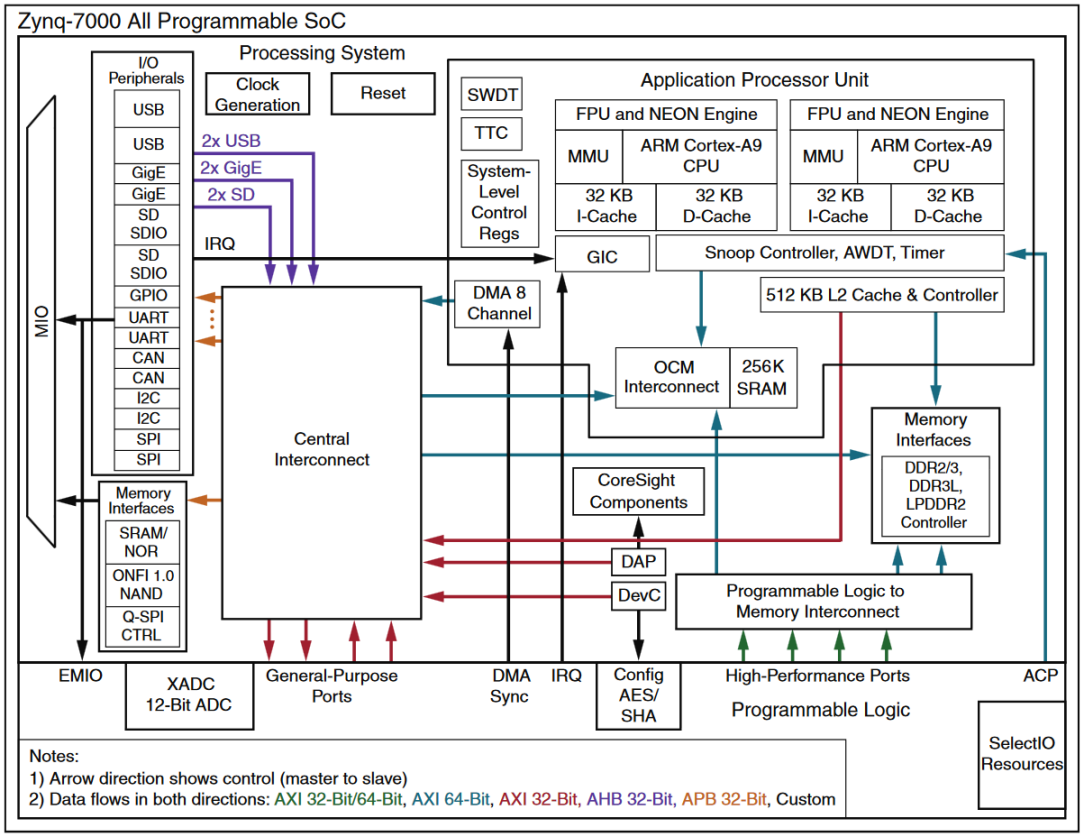


Abbildung 2.5: Funktionseinheiten des Zynq-SoCs [Xil16a]

rung, vor allem für GPUs, ausgerichtet. IBM arbeitet an einem Framework namens Liquid Metal [IBM14], das die eigene Hochsprache Lime inkludiert, die u. a. objektorientiert und Java-kompatibel ist. Lime soll nicht nur HLS für FPGAs ermöglichen, sondern auch die Programmierung für FPGAs, GPUs und CPUs vereinheitlichen. Weitere High-Level-Frameworks sind Esterel [Est14], Kiwi [SG08] und Chisel [Uni14].

Die High-Level-Synthese von Xilinx generiert aus C-, C++- und SystemC-Code synthetisierbaren Verilog- und VHDL-Code (vgl. Abbildung 2.6). Das Hochsprachenmodell ist für die HLS anzupassen, da ein Interfacing zu realisieren ist, damit das HLS-Modul mit Daten versorgt werden kann. Außerdem müssen Aufrufe von Betriebsystemenfunktionen entfernt oder substituiert werden.

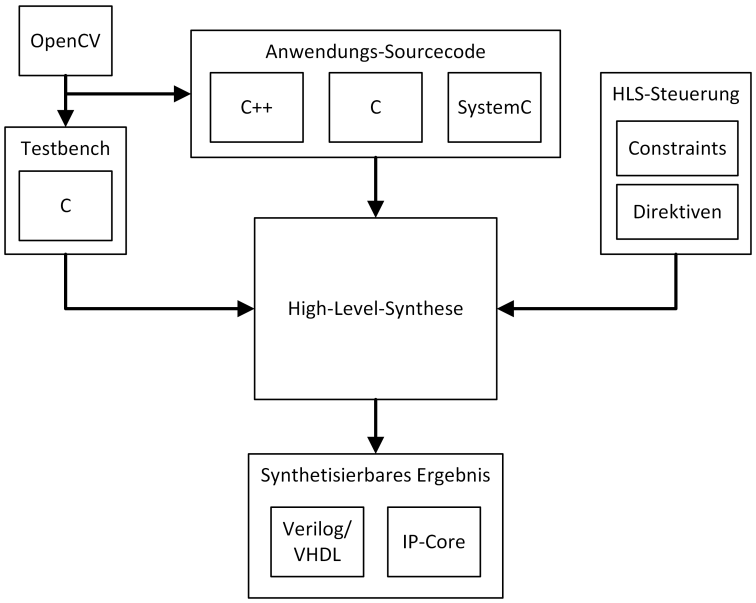


Abbildung 2.6: High-Level-Synthese Workflow zur Generierung eines IP-Cores

Für Benutzer der HLS gibt es folgende Möglichkeiten, auf die Synthese Einfluss zu nehmen und das Ergebnis zu geringerem Ressourcenbedarf und schnellerem Timing zu optimieren:

1. Constraints
2. Direktiven
3. Bit-genaue Datentypen

Constraints

Constraints legen Rahmenbedingungen fest, die das Ergebnis der HL-Synthese erfüllen müssen. Hat man beispielsweise die Anforderung, dass eine Operation eine bestimmte Latenz nicht überschreiten darf, so kann die maximale Latenz als Constraint festgelegt werden und der Compiler wird die Synthese darauf ausrichten. Wenn Constraints nicht erfüllt werden können, gibt der Compiler entsprechende Warnungen aus.

Direktiven

Direktiven sind Anweisungen für den Compiler, die auf Codestrukturen, wie zum Beispiel Variablen, Arrays, Schleifen oder Funktionen, angewendet werden. Es gibt 22 Direktiven [Xil14a], exemplarisch werden folgende näher benannt:

1. **Pipeline:** Durchsatzerhöhung, indem die nebenläufige Ausführung von Operationen einer Schleife oder Funktion erlaubt wird.

2. **Allocation:** Spezifiziert ein Limit für die Benutzung eines Operators. Damit wird das Ressourcesharing bzw. die Nutzung von Ressourcen für Multizyklus-Datenpfade gesteuert.
3. **Inline:** Funktionen werden durch Inline-Ersetzung aufgelöst. Dadurch werden Optimierungen über Funktionsgrenzen hinweg ermöglicht und der Overhead durch Funktionsaufrufe reduziert.
4. **Dataflow:** Task-Level-Pipelining, was mit Berücksichtigung auf Datenabhängigkeiten das nebenläufige Ausführen von Funktionen und Schleifen ermöglicht.
5. **Loop_Flatten:** Verschachtelte Schleifen werden für eine geringere Latenz in eine Schleifenebene transformiert.

Verschiedene Direktiven werden nacheinander angewandt, was in unterschiedlichen „Solutions“ getestet werden kann. Nach der Synthese mit Ressourcen- und Timingabschätzung werden die Solutions miteinander verglichen, wodurch man sich an eine für die Anwendung optimale High-Level-Synthese annähert.

Bit-genaue Datentypen

In Standard-C++ ist es, wie in Standard-C, nicht vorgesehen, Variablen mit beliebiger Bitbreite zu definieren. Das HLS-Tool realisiert das über eine Bibliothek. Bit-genaue Datentypen realisieren einen geringeren Ressourcenverbrauch, indem die erforderliche Genauigkeit bei Operationen geplant wird. Eine Operation, deren Ergebnis beispielsweise immer unter der 4-Bit Grenze bleibt, müsste in C++ mindestens mit einer 8-Bit Zahl implementiert werden, was nach der High-Level-Synthese in einem FPGA zu vermehrten Einsatz von DSP48-Elementen führen würde. Die Bibliothek bietet für C++ mit

```
#include ap_int.h
```

einen bit-genauen Integer-Datentyp. Mit

```
ap_int<N> var
```

wird beispielsweise eine N-bit ($0 < N <= 1024$) breite Integervariable deklariert. Die Bibliothek *ap_fixed.h* bietet bit-genaue Datentypen für Festkommazahlen.

Diese Datentypen werden in den C++-Code integriert, was eine Ressourcen- und Genauigkeitsplanung erfordert.

Testbench zur Funktionalitätsverifikation

Mit einer Testbench wird die Funktionalität des Input-Codes und das Syntheseergebnis über die Funktionen „C-Simulation“ und „Co-Simulation“ verifiziert. Die Co-Simulation benutzt Datenvektoren der C-Simulation, um die Funktionalität des Syntheseergebnisses zu verifizieren. Dadurch kann auf die Erstellung einer dedizierten RTL-Testbench verzichtet werden. Die Anwendung und die Testbench greifen für die Bildverarbeitung auf die OpenCV-Bibliothek [Ope14] zurück. Diese greift auf betriebsystemspezifische Funktionen zurück und kann mit dem HLS-Tool nicht kompiliert werden. Aus diesem

Grund bietet das HLS-Tool eine OpenCV-kompatible Bildverarbeitungsbibliothek an, mit der die OpenCV-Bibliothek mit Einschränkungen ersetzt werden kann (vgl. Abschnitt 4.1.1).

Das Ergebnis der High-Level-Synthese ist ein RTL-Modell, das in Verilog oder VHDL realisiert ist. Mit der Exportfunktion wird das RTL-Modell als Packaged IP zusammengefasst werden, wodurch es als Bibliothekselement bei der Systemintegration in Vivado verfügbar ist.

2.4 AXI-Protokoll und FPGA-Bibliothekskomponenten für die Systemintegration

In diesem Abschnitt werden die Komponenten vorgestellt, die in der Systemintegration einer Bildverarbeitungs-IP verwendet werden. Das zur Kommunikation verwendete AXI-Protokoll wird nachfolgend erläutert.

2.4.1 AXI-Protokoll zur On-Chip-Kommunikation

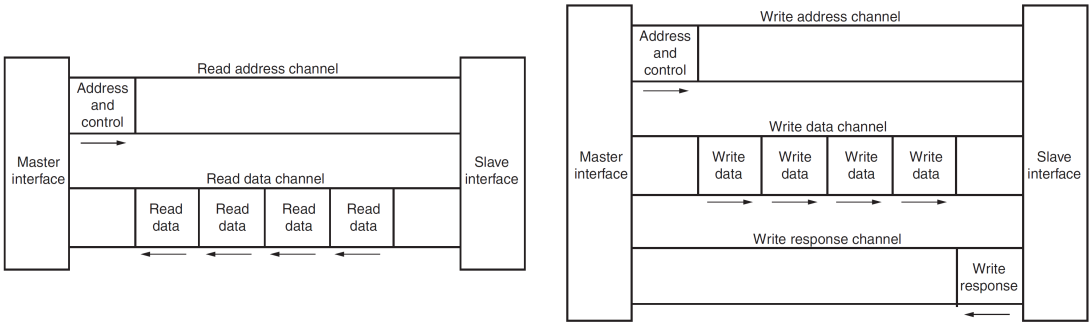
Das Advanced eXtensible Interface (AXI) ist ein Teil des AMBA(Advanced Microcontroller Bus Architecture)-Standards [Arm10] von ARM und dient der On-Chip-Kommunikation von IPs. Das Ziel von AMBA ist, die Kommunikation von SoC-Komponenten zu vereinheitlichen, um die Modularität und Wiederverwertbarkeit zu erhöhen. Das AXI wurde von Xilinx adaptiert [Xil11] und steht dem Benutzer über Bibliotheks-IPs zur Verfügung. AXI4 umfasst drei Protokolle:

- AXI4: Adressbasierte Datenübertragung für Memory-Mapped-Interfaces. Burstgröße von 256 Datentransfers mit einer Anfangsadresse.
- AXI4-Lite: AXI4 ohne Burstfunktionalität mit wenig Implementierungs-Ressourcenbedarf für geringe Anforderungen an den Datendurchsatz, beispielsweise für die Konfiguration von IP-Cores.
- AXI4-Stream: Streaming von Daten ohne Adresse für hohen Datendurchsatz, unbeschränkte Burstgröße, Handshaking-basiert.

Die Burstgröße ist die Menge an Datenworten, die mit einer Startadresse übertragen werden, wodurch der Overhead für neue Adressanfragen eingeschränkt wird. Das AXI-Protokoll verwendet ein Master-Slave-Pattern, in dem der Slave Anfragen des Masters beantwortet. AXI4 und AXI4-Lite verwenden 5 Kanäle zur Kommunikation, damit die simultane Übertragung in Read- und Write-Richtung realisiert wird:

- Read Address Channel
- Write Address Channel
- Read Data Channel

- Write Data Channel
- Write Response Channel



(a) Read-Transaktion (b) Write-Transaktion

Abbildung 2.7: Adressbasierte AXI4- und AXI4-Lite-Datenübertragungsvorgänge zwischen einem AXI-Master und einem AXI-Slave [Xil11]

Bei einer Read-Anfrage sendet der Master über den Read Address Channel die Adresse und Kontrollinformationen, beispielsweise die Größe des Bursts. Der Slave antwortet mit den Daten, die er über den Read Data Channel dem Master übergibt (Abbildung 2.7a). Für eine Write-Transaktion sendet der Master sowohl Adresse und Kontrollinformationen über den Write Address Channel als auch die Daten über den Write Data Channel. Der Slave antwortet über den Write Response Channel, ob die Transaktion erfolgreich ausgeführt wurde (Abbildung 2.7b).

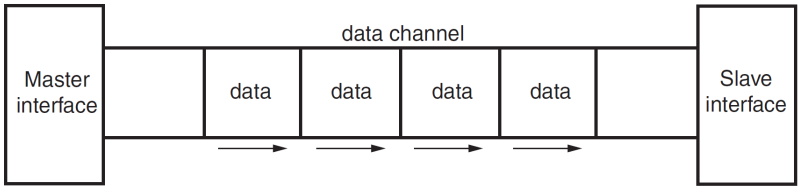


Abbildung 2.8: AXI4-Stream zwischen einem AXI-Master und AXI-Slave [Xil11]

Das AXI4-Stream-Protokoll ist für auf Datenfluss ausgelegte RTL-Implementierungen ohne adressbehaftete Daten bestimmt (Abbildung 2.8) und die Datenflussrichtung ist stets von Master zum Slave. Jeder AXI-Stream besitzt einen unidirektionalen Datenkanal (TDATA), Handshaking Signale (TVALID, TREADY) und optionale Signale (TSTRB, TKEEP, TLAST, TID, TDEST, TUSER). Die optionalen Signale können vom Anwender für IP-spezifische Anforderungen definiert werden.

2.4.2 AXI-Interconnect

Der konfigurierbare AXI-Interconnect-IP-Core [Xil13d] dient der Verbindung von AXI-Slaves und AXI-Master unterschiedlicher Clock-Domains und Protokollversionen. Unterstützt wird AXI3, AXI4 und AXI4-Lite, die jeweils in das andere Protokoll konvertiert werden können. Folgende Komponenten inkludiert der AXI-Interconnect (Abbildung 2.9):

- AXI Crossbar: verbindet einen oder mehrere memory-mapped Master zu einem oder mehreren memory-mapped Slaves
- AXI Data Width Converter: ermöglicht die Konvertierung von Datenbreiten einer AXI-Verbindung, ein Master kann mit einem Slave mit niedrigerer oder größerer Datenbreite verbunden werden
- AXI Clock Converter: verbindet einen Master mit einem Slave, der in einer anderen Clock-Domain operiert
- AXI Protocol Converter: verbindet einen AXI3-, AXI4- oder AXI4-Lite-Master mit einem AXI3-, AXI4- oder AXI4-Lite-Slave
- AXI Data FIFO: Fifo-buffer
- AXI Register Slice: verbindet Master und Slaves über Register zur Realisierung einer Pipelinestruktur, beispielsweise um einen kritischen Timing-Pfad aufzulösen
- AXI MMU: dient der Adressdekodierung und dem Adressremapping

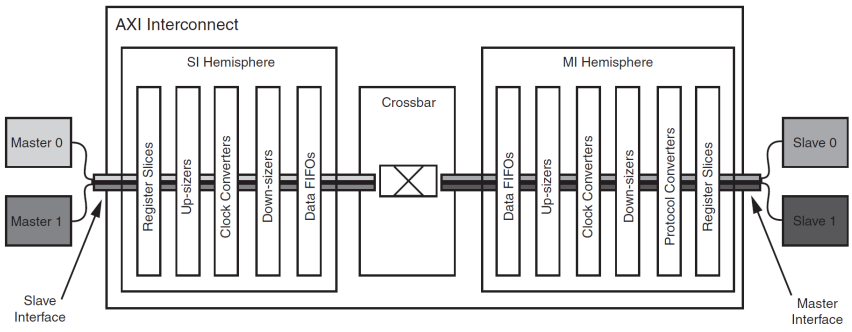


Abbildung 2.9: Top-Level-Ansicht der AXI-Interconnect-IP mit Sublevel-Verarbeitungsstufen: Data-Fifo, Data Width Converter, Clock Converter und Register Slice auf jeder Interfaceseite, mittig die Crossbar

Der AXI-Interconnect ist ausschließlich für die memory-mapped AXI-Protokolle AXI3, AXI4 und AXI4-Lite verwendbar. Für AXI-Stream gibt es einen AXI-Stream-Interconnect [Xil16b], der in dieser Arbeit keine Verwendung findet.

2.4.3 AXI-VDMA

Der Zugriff auf memory-mapped Daten über einen AXI-Stream erfordert eine Konvertierung zwischen der adressbasierten und der adresslosen Datenverbindung. Die rekonfigurierbaren IP-Cores AXI-Direct-Memory-Access (DMA) und AXI-Video-Direct-Memory-Access (VDMA) realisieren Stream-to-Memory-Mapped- (S2MM) und Memory-Mapped-to-Stream-Konvertierungen (MM2S) über einen Datamover (vgl. Abbildung 2.10) [Xil15a].

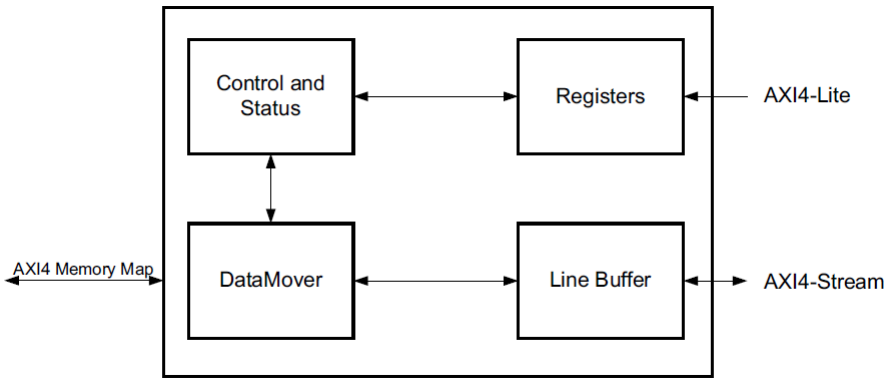


Abbildung 2.10: VDMA-Core bestehend aus einem Datamover, Kontrollregistern und einer konfigurierbaren Anzahl an Linebuffern

Die (V-)DMA-IPs unterstützen je einen Read- und Write-Kanal, deren Daten- und Adressbreite konfiguriert wird. Der VDMA-Core erweitert den DMA-Core um spezifische Funktionen für Bild- und Videoverarbeitungsanwendungen: ein konfigurierbarer Linebuffer ist integriert und eine Frame-Synchronisation (Gen-Lock) mit externen Quellen kann durchgeführt werden.

2.4.4 Processing System

Das Zynq-SoC wird im IP-Integrator von Vivado als Processing-System-IP-Core repräsentiert [Xil13e] und dient als Schnittstelle zwischen dem Processing System und der Programmable Logic (Abbildung 2.11).

Als Wrapper erlaubt es die Konfiguration von:

- I/O Peripherals: QSPI, NOR/NAND Flash, UART, I2C, SPI, SDIO, GPIO, CAN, USB und Ethernet
- MIO Ports: 54 MIO-Ports für die Verbindung der I/O Periphals
- Extended MIO Ports: Routen von I/O Peripherals über PL
- AXI I/O Interfaces: zwei GP-AXI-Master, zwei GP-AXI-Slaves und vier HP-AXI-Slaves

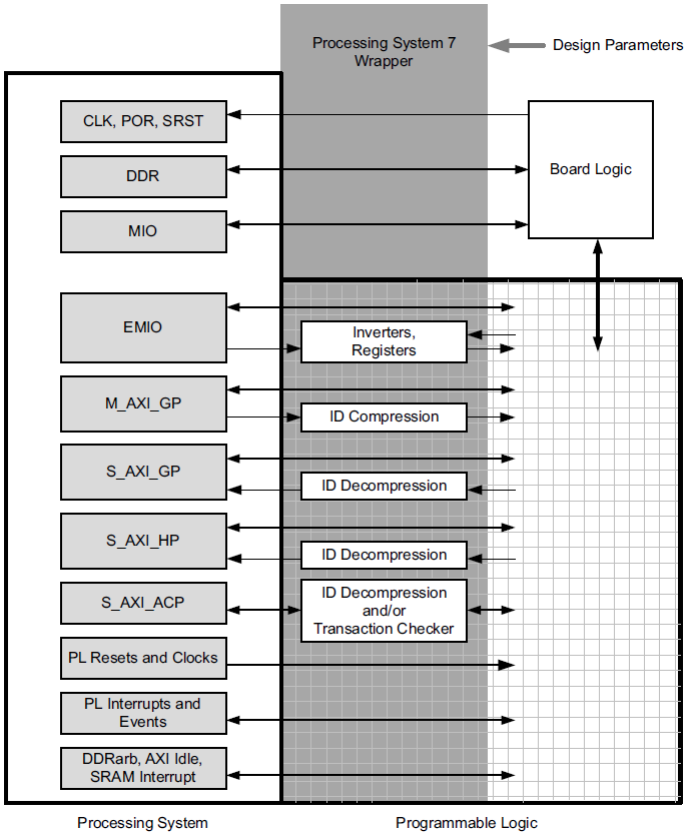


Abbildung 2.11: Wrapper-Logik zwischen dem PS und der PL

- PS und PL Clocks: Prozessor/Speicher Clocks, Peripheral Clocks, Timer und Fabric Clocks für die PL
- PL to PS Interrupts: 16 Shared Interrupts, 4 Private Peripheral Interrupts (PPI)
- PS to PL Interrupts: Weiterleitung von Peripheral Interrupts an die PL

2.4.5 Processor System Reset

Der Processor-System-Reset-IP-Core generiert Reset-Signale für FPGA-Module [Xil15b]. Der Core verarbeitet mehrere Reset-Quellen und leitet diese an die konfigurierten Ziele weiter. Die Art des Resets (Active High oder Active Low) und die Pulsweite kann determiniert werden.

3 Bildverarbeitungsalgorithmenkette zur Extraktion von Fingerabdruckmerkmalen

Als Input für die Extraktionskette dient ein vorverarbeitetes 8-Bit Grauwertbild. Der Output ist eine Liste von gefundenen Grundminutien (Abbildung 3.1).

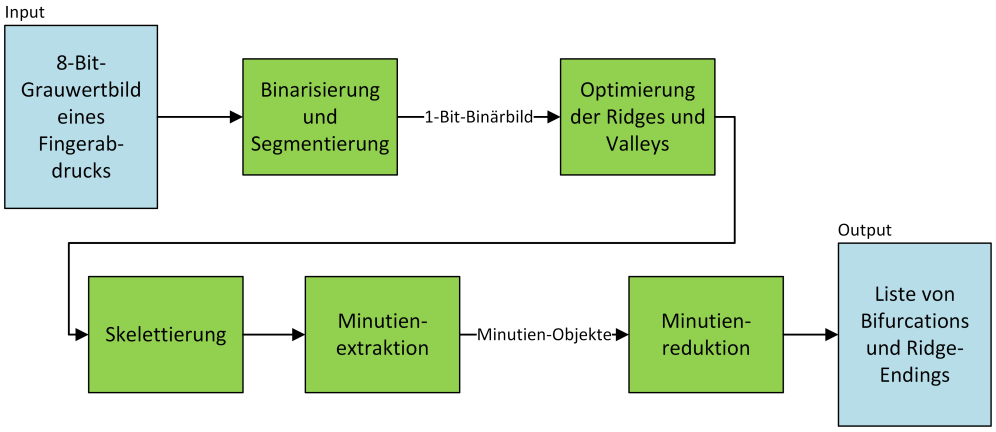


Abbildung 3.1: Input der Bildverarbeitungalgorithmenkette ist ein Grauwertbild, der Output die gefundenen Minutien in Form von Minutienobjekten

Die Verarbeitungsschritte wurden in Kapitel 2.1.2 eingeführt und werden im Folgenden inklusive der realisierten Implementierung erläutert. Für die Implementierung in C++ wurden als Testbilder synthetisch generierte Fingerabdruckaufnahmen verwendet, da zu diesem Zeitpunkt keine Aufnahmen des Fingerabdruckscanners zur Verfügung standen. Die Generierung wurde mit dem Programm „SFinGe“ [Bol14] durchgeführt, welches die Konfiguration der Eigenschaften der Aufnahme, zum Beispiel den Grad der Vignettierung oder die Anzahl und Schwere der Lücken von Ridges, erlaubt. Die Beispielbilder haben folgende Dimensionen:

- Auflösung: $560 \cdot 410$ Pixel
- 256 Graustufen, 1 Byte pro Pixel
- unkomprimierter Speicherbedarf ohne Dateiheader: $560 \cdot 410 = 229.600$ Byte (~224,22 KiB)
- Größe der Tiles: $10 \cdot 10$ Pixel

- Anzahl der Tiles: $56 \cdot 41 = 2296$

Die Verarbeitungsschritte werden an einem dieser synthetischen Fingerabdrücke (vgl. Abbildung 3.2a) veranschaulicht. Dieser stellt eine „ungünstige“ Aufnahme eines Fingerabdrucks dar: Sie weist eine deutliche Vignettierung auf, der Hintergrund ist fleckig und die Strukturen des Fingerabdrucks sind nicht klar und weisen Unterbrechungen auf. Diese Eigenschaften fordern von der Bildverarbeitung eine hohe Effektivität, damit die Minutien zuverlässig erkannt werden.

3.1 Binarisierung und Segmentierung

Die Binarisierung wandelt das 8-Bit Grauwertbild (Schwarz = 0, Weiß = 255, Graustufen = 1-254) in ein 1-Bit Schwarz-Weiß-Bild (Schwarz = 0, Weiß = 1) um, da der Crossing-Numbers-Algorithmus zum Finden der Minutien auf Basis von binären Bilddaten arbeitet. Für die Implementierung wurde ein Local-Thresholding-Verfahren gewählt, da dieses ein gutes Ergebnis mit geringer Implementierungskomplexität liefert. Das Local-Thresholding teilt das Bild in gleichgroße Bildausschnitte, sogenannte Tiles, auf, für die jeweils ein Schwellenwert berechnet und angewendet wird. Für die Schwellenwertermittlung kann eine Mittelwertbildung oder Histogrammanalyse verwendet werden. Im Gegensatz zur Mittelwertbildung, die einen Durchschnitt der Grauwerte berechnet, ist eine Histogrammanalyse aufwendig, aber am zuverlässigsten, da die Menge der Grauwerte akkumuliert wird und mit Funktionsanalysen ein Schwellenwert ermittelt wird. Bei einem Fingerabdruck ist eine Mittelwertbildung ausreichend, da die Region eines Fingerabdrucks, aufgrund der dunklen Ridges und hellen Valleys, kontrastreich ist.

Eine Segmentierung, d.h. die Trennung eines Bildobjekts vom Hintergrund, wird vorgenommen, da sonst Tiles, die nichts vom Fingerabdruck enthalten, in der Binarisierung berücksichtigt werden (siehe Abbildung 3.2b). Diese Tiles werden in den nachfolgenden Modulen verarbeitet, wodurch unerwünschte Minutien um den eigentlichen Fingerabdruck herum identifiziert werden. Die Menge der Tiles wurde so gewählt, dass jedes Tile $10 \cdot 10$ Pixel groß ist.

Auf Basis der Untersuchung im Masterprojekt [Kem15a] wurde ein Algorithmus entwickelt, der auch für schlecht konditionierte Fingerabdruckbilder ein korrektes Ergebnis liefert. Dieser Algorithmus verwendet die Kontrastunterschiede, die innerhalb eines Tiles auftreten, das Ridges und Valleys enthält. Folgende Schritte werden hierfür durchgeführt:

1. Für jedes Tile den minimalen und maximalen Grauwert ermitteln.
2. Das Delta dieser Grauwerte berechnen: Tiles, die keine Fingerabdruckinformationen enthalten, haben ein kleines Delta. Tiles mit Fingerabdruckinformationen haben aufgrund der Ridges und Valleys ein großes Delta.
3. Festgelegten Threshold auf die Deltas anwenden: Pixel eines Tiles mit zu geringem Delta werden auf weiß gesetzt.

(a) Originalbild

(b) Binarisierung ohne Segmentierung

Abbildung 3.2: Binarisierung eines Fingerabdrucks ohne Segmentierung, Tilegröße: 10·10 Pixel

Der Delta-Threshold wurde aus empirischen Analysen ermittelt und statisch festgelegt. Die Analysen haben gezeigt, dass dieses Verfahren robust ist und im Falle des Beispielbildes (Abb. 3.2a) ein Threshold von 80 ein sehr ähnliches Ergebnis wie ein Threshold von 120 liefert. Die Abbildungen 3.3a-c zeigen die Ergebnisse des Segmentierungsverfahrens mit verschiedenen Delta-Schwellwerten. Mit einem Schwellwert von 100 (Abb. 3.3b) produziert der Algorithmus das gewünschte, fehlerfreie Ergebnis, mit dem im Folgenden weiter gearbeitet wird.

Implementierung

Die Übergabeparameter für die Funktion sind eine OpenCV-Objektreferenz auf das Grauwertbild („cv::Mat& im“, im Fall der Testbench das Bild aus Abb. 3.2a) und die Anzahl der Tiles in Höhe und Breite, in die das Bild eingeteilt wird (vgl. „AdaptiveLocalThreshold.cpp“ im Bibliotheksordner „/lib/binarisation“).

```
void binarisation(cv::Mat& im, int numTilesWidth, int numTilesHeight)
    {...}
```

Mit zwei for-Schleifen wird das Bild in Tiles-Form iteriert und zwei weitere, verschachtelte for-Schleifen iterieren jedes Tile. Innerhalb der Tile-Iteration werden die Grauwerte für die Mittelwertbildung addiert und der höchste und niedrigste Grauwert für das Deltathresholding ermittelt.

Nach der Tile-Analyse wird für die Binarisierung der Mittelwert der Grauwerte berechnet. Zudem wird für die Segmentierung die Differenz der maximalen und minimalen

(a) Delta-Schwellwert = 20 (b) Delta-Schwellwert = 100 (c) Delta-Schwellwert = 200

Abbildung 3.3: Ergebnisse des Delta-Thresholdings, (b) zeigt das erwünschte Ergebnis

Grauwerte gebildet. Die Segmentierung und Binarisierung wird auf das aktuelle Tile angewendet:

```
//apply delta-threshold
if(delta<deltaThreshold){
  threshold = 0;  //set to zero, so the tile will be binarized to
      complete white
}
//apply threshold for binarisation on tile
for(k = i; ((k-i)<tileWidth); k++){
  for(l = j; ((l-j)<tileHeight); l++){
    intensity = im.at<uchar>(cv::Point(k, l);
    if(intensity >= threshold){
      im.at<uchar>(l,k) = 255; //white
    }else{
      im.at<uchar>(l,k) = 0;   //black
    }
  }
}
```

Damit ist die Binarisierung und Segmentierung abgeschlossen und das Bild (Abbildung 3.3b) kann an das nächste Verarbeitungsmodul, Ridge- und Valleyoptimierung, weitergereicht werden.

3.2 Ridge- und Valleyoptimierung des binarisierten Bildes

Die in der Arbeit von Ideka et al. [Ike+02] vorgestellten Algorithmen wurden implementiert, da sie sowohl Löcher in Ridges als auch Lücken zwischen Ridges schließen.

Der Algorithmus arbeitet mit mathematischer Morphologie, die folgende Operationen umfasst:

- Erosion zur Verschlankung von schwarzen Strukturen im Bild
- Dilatation zur Ausdehnung von schwarzen Strukturen im Bild
- Opening: Erosion und anschließende Dilatation zur Glättung von schwarzen Strukturen im Bild, wobei kleine Löcher gefüllt werden
- Closing: Dilatation und anschließende Erosion zur Glättung von schwarzen Strukturen im Bild, wobei kleine Löcher vergrößert werden
- Mengenoperationen: Vereinigung und Schnitt

Der Algorithmus von Ideka et al. besteht aus zwei Schritten: dem Ridge-Enhancement und dem Valley-Enhancement. Für das Ridge-Enhancement sind folgende Schritte auszuführen:

1. Quellbild mit einer Opening-Operation glätten
2. Geglättetes Bild skelettieren und darauf eine Dilatation-Operation durchführen
3. Dilatiertes Bild mit dem Quellbild vereinen (logisches „Oder“)

Für das Valley-Enhancement wird ein sogenanntes „Flag-Image“ aus den folgenden Schritten erstellt:

1. Quellbild mit einer Opening-Operation glätten
2. Geglättetes Bild skelettieren
3. Im skelettierten Bild eine Dilatation-Operation durchführen

Dieses Flag-Image wird anschließend für das Vally-Enhancement verwendet:

1. Quellbild wird erodiert und mit dem invertiertem Flag-Image eine Schnittmengenoperation (logisches „Und“) durchgeführt
2. Quellbild wird zudem mit dem Flag-Image geschnitten (logisches „Und“)
3. Diese beiden Bilder ergeben mit einer Vereinigungsoperation (logisches „Oder“) das verbesserte Valley-Bild

Beide Optimierungsalgorithmen werden nacheinander auf das binarisierte Bild angewendet (vgl. Abbildung 3.4).

(a) Binarisiertes Ausgangsbild

(b) Anwendung des auf mathematischer Morphologie basierenden Optimierungsalgorithmus

Abbildung 3.4: Ergebnis der Optimierung nach Ideka et al. [Ike+02]

Implementierung

Dem Ridge-Enhancement entsprechend wird eine Kopie des Ursprungsbildes angelegt und auf beiden Bilder eine bitweise Inverse-Operation angewendet. Das ist nötig, weil im Bild Schwarz als „0“ und weiß als „255“ (binär: 11111111) repräsentiert wird, während in den Logikoperationen auf Bit-Ebene Schwarz als „1“ und weiß als „0“ interpretiert wird (vgl. „Ideka.cpp“ im Pfad „/lib/optimisation“).

```
void ridgeEnhancement(cv::Mat& im){
  cv::Mat source = im.clone();
  cv::bitwise_not(source, source);
  cv::bitwise_not(im, im);
  smooth(im);
  GuoHall::thinning(im);
  cv::Mat element = cv::getStructuringElement(cv::MORPH_CROSS, cv::Size
      (3, 3));
  cv::dilate( im, im, element );
  cv::bitwise_or(im, source, im);
  cv::bitwise_not(im, im);
}
```

Anschließend werden die Schritte Glätten, Skelettieren und Dilatieren ausgeführt. Das Bild wird mit dem Ursprungsbild vereint (bitwise_or(...)) und wieder in das ursprüngliche Bildformat zurück transformiert (bitwise_not(...)). Für die Dilatations-Operation wurde das „Strukturierende Element“ festgelegt, das die Maske für den

Ortsbereich der Operation ist. In diesem Fall wurde eine Elementarraute verwendet, da sie durch ihre rundliche Form ein ansprechendes Ergebnis liefert (siehe Abbildung 3.5). Im Vergleich dazu liefert die Verwendung des 8-Nachbarn-Elements ein grobkantigeres Bild, welches bei der Skelettierung für Abzweigungen auf Ridges sorgen könnte, die real nicht existieren.

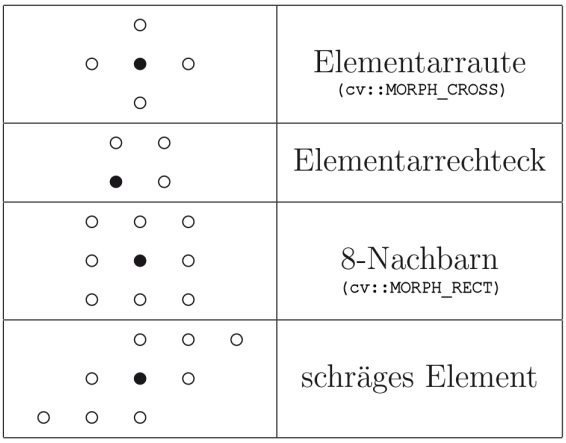


Abbildung 3.5: Beispiele für strukturierende Elemente. Mit • ist der Bezugspunkt und mit ○ sind die Nachbarn markiert (vgl. [Nis+12])

Die Dilatation und Erosion werden durch OpenCV angeboten, ebenso wie eine Funktion, die das passende strukturierende Element auswählt.

Das Ridge-Enhancement benötigt wie oben beschrieben mehr Schritte, vor allem, weil in einem Zwischenschritt ein Flag-Image erstellt wird:

```
void flagImage(cv::Mat& im){
  smooth(im);
  cv::bitwise_not(im, im);
  GuoHall::thinning(im);
  cv::Mat element = cv::getStructuringElement(cv::MORPH_CROSS, cv::Size
      (3, 3));
  cv::dilate( im, im, element );
  cv::bitwise_not(im, im);
}
```

Die Funktion für das Verbessern der Ridges besteht aus zwei Strängen: Zum einen wird das Quellbild erodiert und mit dem negativen Flag-Image geschnitten (cv::bitwise_and(...)) und zum anderen wird das Quellbild mit dem normalen Flag-Image geschnitten. Diese beiden Bilder werden vereint und man erhält das invertierte Ergebnisbild.

```
void valleyEnhancement(cv::Mat& im){
  [...]
  cv::bitwise_and(im, flagImg, im);
  cv::Mat element = cv::getStructuringElement(cv::MORPH_CROSS, cv::Size
      (3, 3));
  cv::erode( im2, im2, element );
```

```
    cv::bitwise_and(im2, flagImgInv, im2);
    cv::bitwise_or(im, im2, im);
    cv::bitwise_not(im, im);
}
```

An der Stelle des Platzhalters „[...]“ wird das Quellbild mehrmals kopiert, die Flag-Image-Funktion aufgerufen und alle Bilder für die Logikoperationen invertiert. Als Ergebnis erhält man ein Bild (vgl. Abbildung 3.4b), das für die Skelettierung bereit ist.

3.3 Skelettierung

Die Merkmale werden aus einem Bild extrahiert, in dem die Ridges eine Breite von einem Pixel haben müssen. Das wird durch die Skelettierung erreicht, die Ridges auf einen Pixel verschlankt, ohne diese zu unterbrechen oder deren Verlauf zu verändern.

(a) nach Guo und Hall (b) nach Zhang und Suen

Abbildung 3.6: Thinning-Algorithmen im Vergleich

Für die Implementierung wurden die Algorithmen von Guo und Hall [GH89] sowie von Zhang und Suen [ZS84] ausgewählt. Da sie sich kaum unterscheiden (Abbildung 3.6), wurden beide für einen Vergleich implementiert. Die Algorithmen sind durch zwei Subiterationen parallelisierbar und verwenden eine $3 \cdot 3$ Matrix (8-Neighborhood eines Pixels, vgl. Abb. 3.9), auf die definierte Bedingungen zutreffen müssen. Abbildung 3.6 zeigt beide Verfahren im direkten Vergleich. Die Unterschiede sind visuell gering: An manchen Stellen hat das Verfahren von Guo und Hall weniger „ausgefranste“ Ridges, an manchen Stellen das von Zhang und Suen. Bei einem Vergleich der Ergebnisse der Minutien-Auswertung stellt sich heraus, dass im Beispielbild (Abb. 3.2a) mit dem Verfahren von Guo und Hall 598 Minutien identifiziert wurden und mit dem Verfahren von Zhang und Suen 522. Demnach liefert der Algorithmus von Zhang und Suen ein

glatteres Skelett und ist für diese Anwendung vorzuziehen. Im Folgenden werden die Schritte für den Algorithmus von Zhang und Suen erläutert.

Der Algorithmus ist in zwei Iterationen eingeteilt: Erstens die Analyse der 8-Nachbarschaft in „Süd-Ost"-Richtung, zweitens in „Nord-West"-Richtung. Wenn für die erste Subiteration folgende Bedingungen erfüllt werden, so wird das aktuell betrachtete Pixel P_1 gelöscht:

1. $2 \leq B(P_1) \leq 6$
2. $A(P_1) = 1$
3. $P_2 \cdot P_4 \cdot P_6 = 0$
4. $P_4 \cdot P_6 \cdot P_8 = 0$

$A(P_1)$ ist die Anzahl der 01-Übergänge in der 8-Nachbarschaft von P_1, $B(P_1)$ ist die Anzahl der Pixel in der 8-Nachbarschaft mit dem Wert 1. Mit dieser Iteration werden Pixel gelöscht, die an einer Nord-West-Grenze liegen.

Die zweite Subiteration löscht das Pixel P_1 wenn:

1. $2 \leq B(P_1) \leq 6$
2. $A(P_1) = 1$
3. $P_2 \cdot P_4 \cdot P_8 = 0$
4. $P_2 \cdot P_6 \cdot P_8 = 0$

Hiermit werden Pixel gelöscht, die an einer Süd-Ost-Grenze liegen. Alle Pixel, die eine der beiden Bedingungen erfüllen und gelöscht werden sollen, werden in einer Matrix M gespeichert, welche nach jeder vollständigen Iteration vom Ursprungsbild subtrahiert wird. Die vollständige Iteration wird wiederholt, bis es keine Änderungen mehr zwischen den Iterationen gibt.

Implementierung

Mit der Anweisung

```
im /= 255;
```

werden die Grauwerte des Bildes umgewandelt, sodass Schwarz mit „1" statt mit „255" repräsentiert wird (vgl. „ZhanSuen.cpp" im Bibliotheksordner „/lib/thinning"), da Logikoperationen mit 1 (=Schwarz) und 0 (=Weiß) arbeiten. Die Schleife

```
do {
  thinningIteration(im, 0);
  thinningIteration(im, 1);
  cv::absdiff(im, prev, diff);
  im.copyTo(prev);
} while (cv::countNonZero(diff) > 0);
```

führt die beiden in einer Funktion ausgelagerten Subiterationen aus, bis es zwischen dem aktuellen Bild und dem vorherigen Bild keinen Unterschied mehr gibt. In die-

ser Funktion werden die Pixel des Bildes iteriert und die im vorherigen Abschnitt vorgestellten Bedingungen geprüft.

```
uchar p2 = im.at<uchar>(i-1, j);
uchar p3 = im.at<uchar>(i-1, j+1);
uchar p4 = im.at<uchar>(i, j+1);
uchar p5 = im.at<uchar>(i+1, j+1);
uchar p6 = im.at<uchar>(i+1, j);
uchar p7 = im.at<uchar>(i+1, j-1);
uchar p8 = im.at<uchar>(i, j-1);
uchar p9 = im.at<uchar>(i-1, j-1);

int A  = (p2 == 0 && p3 == 1) + (p3 == 0 && p4 == 1) +
(p4 == 0 && p5 == 1) + (p5 == 0 && p6 == 1) +
(p6 == 0 && p7 == 1) + (p7 == 0 && p8 == 1) +
(p8 == 0 && p9 == 1) + (p9 == 0 && p2 == 1);
int B  = p2 + p3 + p4 + p5 + p6 + p7 + p8 + p9;
int m1 = iter == 0 ? (p2 * p4 * p6) : (p2 * p4 * p8);
int m2 = iter == 0 ? (p4 * p6 * p8) : (p2 * p6 * p8);

if (A == 1 && (B >= 2 && B <= 6) && m1 == 0 && m2 == 0)
    marker.at<uchar>(i,j) = 1;
```

Die Differenzierung nach Art der Subiteration („Süd-Ost“ oder „Nord-West“) wird mit der Variable „iter“ realisiert.

Nach Beendigung aller Iterationen werden die Werte des Bildes mit

```
im *= 255;
```

in das Ursprungsformat zurück transformiert und stehen für die weitere Analyse zur Verfügung.

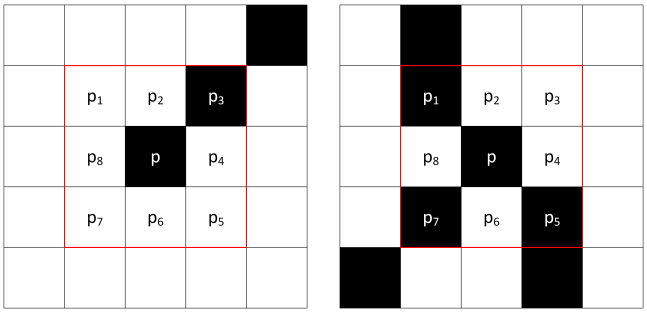
Der Algorithmus nach Guo und Hall befindet sich in der Datei „GuoHall.cpp“ im selben Ordner und unterscheidet sich zum Algorithmus von Zhang und Suen in den Bedingungen, die in einer Subiteration auf ein Pixel angewendet werden.

3.4 Merkmalsextraktion

Die Minutienextraktion wird mit dem „Crossing-Number“-Verfahren [MJ09, S. 149] durchgeführt, welches für jedes schwarze Pixel mit seiner 8-Nachbarschaft (8-Neighborhood) eine „Crossing-Number“ berechnet. Sie dient der Identifizierung der zwei grundlegenden Minutien-Arten: der Bifurcation („Gabelung“) und dem Ridge-Ending. Die Crossing Number $cn(p)$ eines Pixels p ist in einem binarisierten Bild mit $val(p)\,\epsilon\,\{0,1\}$ als die Hälfte der Summe vom Betrag der Differenz zweier nebeneinanderliegenden Pixel der 8-Neighborhood definiert:

$$cn(p) = \frac{1}{2}\Big(\sum_{i=1\ldots8} |val(p_{i \bmod 8}) - val(p_{i-1})|\Big) \tag{3.1}$$

p_i mit $i = 1...8$ ist die 8-Neighborhood in geordneter Reihenfolge des Pixels p. Die Crossing-Number ist die halbierte Anzahl der 01- und 10-Übergänge in der Nachbarschaft. Abbildung 3.7 zeigt Beispiele für die Crossing Number mit zugehöriger 8-Nachbarschaft.



(a) cn(p) = 1 → Ridge-Ending (b) cn(p) = 3 → Bifurcation

Abbildung 3.7: Vergrößerte Bildausschnitte mit rot umrandeter 8-Neighborhood

Ist die Crossing-Number 1, wurde ein Ridge-Ending gefunden, ist sie 3, wurde eine Bifurcation identifiziert. Abbildung 3.8 zeigt, dass viele Minutien erkannt werden, vor allem am Rand des Fingerabdrucks. Im Bild sind Ridge-Endings mit blauen Kästchen gekennzeichnet und Bifurcations mit roten. Viele dieser Minutien sind keine validen Minutien, sondern Artefakte der Aufnahme: Die Ridge-Endings am Rand gehören zum Beispiel nicht zu den Merkmalen des Fingerabdrucks, sondern sind die Grenze der erfassten Fingerkuppe. Diese False-Positives können durch einen Optimierungsschritt des skelettierten Bildes oder durch die Minutienreduktion minimiert werden.

(a) Skelettierte Bild (b) gefundene Minutien

Abbildung 3.8: Ergebnis der Minutienidentifikation

Implementierung

Als Parameter wird neben dem skelettierten Bild der Funktion „getMinutiae(...)“ ein Wert übergeben, der festlegt, wie groß der Randbereich sein soll, der im Bild nicht berücksichtigt wird (siehe „CrossingNumber.cpp“ im Bibliotheksordner „/lib/extraction“). Dieser muss mindestens 1 Pixel groß sein, da für jedes schwarze Pixel die 8-Nachbarschaft analysiert wird und Pixel am Rand keine vollständige Nachbarschaft haben. Mit dieser Berücksichtigung wird das Bild Pixel für Pixel iteriert und wenn das aktuell betrachtete Pixel schwarz ist, wird die Crossing-Number (cn) berechnet:

```
cn = cn + abs(im.at<uchar>(cv::Point(i-1, j-1))/255 - im.at<uchar>(cv::
    Point(i  , j-1))/255);
cn = cn + abs(im.at<uchar>(cv::Point(i  , j-1))/255 - im.at<uchar>(cv::
    Point(i+1, j-1))/255);
cn = cn + abs(im.at<uchar>(cv::Point(i+1, j-1))/255 - im.at<uchar>(cv::
    Point(i+1, j  ))/255);
cn = cn + abs(im.at<uchar>(cv::Point(i+1, j  ))/255 - im.at<uchar>(cv::
    Point(i+1, j+1))/255);
cn = cn + abs(im.at<uchar>(cv::Point(i+1, j+1))/255 - im.at<uchar>(cv::
    Point(i  , j+1))/255);
cn = cn + abs(im.at<uchar>(cv::Point(i  , j+1))/255 - im.at<uchar>(cv::
    Point(i-1, j+1))/255);
cn = cn + abs(im.at<uchar>(cv::Point(i-1, j+1))/255 - im.at<uchar>(cv::
    Point(i-1, j  ))/255);
cn = cn + abs(im.at<uchar>(cv::Point(i-1, j  ))/255 - im.at<uchar>(cv::
    Point(i-1, j-1))/255);
cn = cn/2;
```

Die 8-Nachbarschaft wird, wie in Abb. 3.7 angedeutet, im Uhrzeigersinn iteriert. Erst wird der Betrag der Differenz zwischen p_1 (i-1, j-1) und p_2 (i, j-1) gebildet, dann zwischen p_2 und p_3 (i+1, j-1) und so weiter (siehe Abb. 3.9). Die Beträge werden summiert und am Ende durch 2 geteilt. Der Wert jedes Pixels wird durch 255 geteilt, damit Weiß den Wert 1 hat und Schwarz den Wert 0. Logikoperationen interpretieren Schwarz als 1 und Weiß als 0 (siehe Kapitel 3.2). Da der Betrag der Differenz verwendet wird, bedeutet dies an dieser Stelle aber keinen Unterschied im Ergebnis.

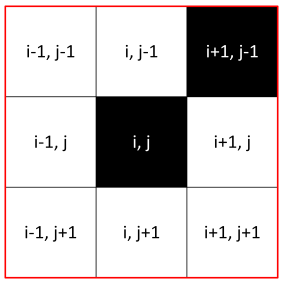


Abbildung 3.9: Indizes der 8-Nachbarschaft des Pixels p (i, j)

Wenn die Crossing-Number 1 oder 3 ist wurde ein gesuchtes Merkmal identifiziert und dessen Koordinaten werden zusammen mit der Art der Minutie, in Form eines Enums, als Minutiae-Objekt in eine Liste gespeichert:

```
if(cn == 1){
  Minutiae minut(i, j, Minutiae::Type::RIDGEENDING);
  minutiae.push_back(minut);
  ridgeEndingCount++;
}else if(cn == 3){
  Minutiae minut(i, j, Minutiae::Type::BIFURCATION);
  minutiae.push_back(minut);
  bifurcationCount++;
}
```

Die Minutien werden im Test-Programm ausgewertet und im skelettierten Bild markiert (Abb. 3.8b).

3.5 Minutienreduktion

Die implementierte Filterung ist als Proof-of-concept zu verstehen. Sie wurde mit einer einfachen Methode eingebaut, damit man eine Vorstellung erhält, wie die validen erkannten Merkmale aussehen (siehe Abbildung 3.10b).

(a) Erkannte Minutien ohne Filterung (b) Erkannte Minutien mit Filterung

Abbildung 3.10: Ergebnis der Minutienfilterung

Die Filterung beruht auf der Annahme, dass Ridge-Endings einen gewissen Mindestabstand zueinander haben. Dementsprechend wird die euklidische Distanz zwischen jedem

Ridge-Ending berechnet und bei Unterschreitung eines bestimmten Abstandes beide Merkmale gelöscht.

Implementierung

Die Minutien liegen in Form einer Liste von Minutien-Objekten vor, die die Koordinaten und die Art der Minutie enthalten (vgl. Abschnitt 3.4). Die Liste wird zweifach iteriert, um die euklidische Distanz zwischen Ridge-Endings zu berechnen (vgl. „Filter.cpp“ im Bibliotheksordner „/lib/extraction“). Die Berechnung wird nur vorgenommen, wenn es sich bei den aktuell betrachteten Minutien nicht um dieselben handelt.

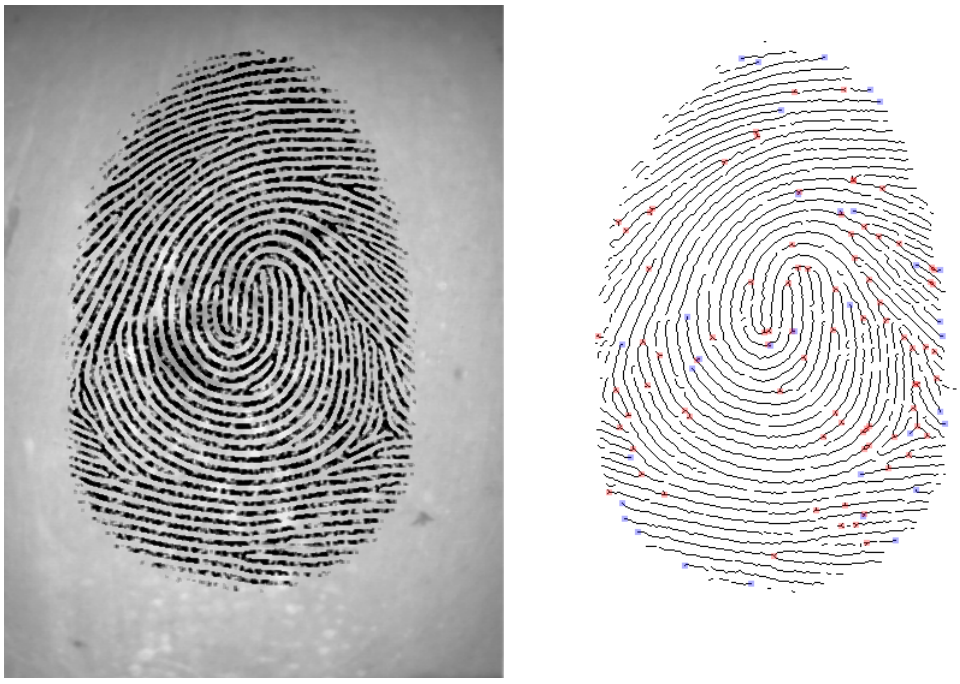
```
if((minutiae[j].getType() == Minutiae::Type::RIDGEENDING) && !same){
  double distance = euclideanDistance(minutiae[i].getLocX(), minutiae[i
      ].getLocY(), minutiae[j].getLocX(), minutiae[j].getLocY());
  if(distance < minDistanceForMinutiae){
    minutiae[i].setMarkTrue();
    minutiae[j].setMarkTrue();
  }
}
```

Der Threshold *minDistanceForMinutiae* ist entsprechend der Auflösung und des Fingerabdruckes festzulegen. Wird der Wert unterschritten, werden beide Minutien für eine spätere Löschung markiert.

```
std::vector<Minutiae> minutiaeNew;
for(std::vector<Minutiae>::size_type i = 0; i<minutiae.size(); i++){
  if(!minutiae[i].getMark()){
    minutiaeNew.push_back(minutiae[i]);
  }
}
minutiae = minutiaeNew;
```

Die markierten Minutien werden gelöscht, indem alle nicht-markierten Minutien in eine neue Liste kopiert werden und diese die alte ersetzt.

Die Grundmerkmale sind nach dem letzten Verarbeitungsschritt erfasst und stellen Rohdaten dar, die für die standardkonforme Speicherung [ANS11][ISO11] weiter zu verarbeiten sind. Abbildung 3.11 zeigt das Ausgangsbild des Fingerabdrucks und das verarbeitete Bild mit markierten Minutien für den direkten Vergleich nebeneinander.



(a) Ausgangs-Grauwerbild des Fingerabdrucks

(b) Verarbeiteter Fingerabdruck mit gefundenen und gefilterten Minutien

Abbildung 3.11: Ergebnis der Bildverarbeitungskette der Minutienextraktion

4 High-Level-Synthese zur FPGA-Implementierung der Bildverarbeitung

Die HLS-spezifischen Anpassungen der C++-Implementierung werden in diesem Kapitel beschrieben. Es werden die Schnittstellen, der Datenverarbeitung und Bibliothek angepasst. Die für die Verifizierung der Funktionalität erforderliche Testbench wird im letzten Abschnitt des Kapitels erläutert.

4.1 HLS-spezifische Strukturierung der Algorithmusimplementierung

Die High-Level-Synthese wird auf die Binarisierung und die Segmentierung angewandt. Diese Bildverarbeitungsstufe wurde für die automatische Architektursynthese ausgewählt, da sie in der Bildverarbeitungskette als Eingangsmodul wirkt und typische Mechanismen einer Bildverarbeitung enthält.

4.1.1 Substituierung der OpenCV-Library mit der HLS-Bibliothek

Die Includes für die OpenCV-Bibliothek der ursprünglichen C++-Implementierung lauten:

```
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
```

Für die HLS werden diese mit der HLS-spezifischen Bibliothek ersetzt (vgl. Datei „AdaptiveLocalTheshold.cpp“ im Ordner „\lib\binarisation\“ oder Anhang 2):

```
#include "hls_video.h"
```

Diese Bibliothek enthält die grundlegenden OpenCV-Funktionen [Xil14a, S. 135], die teilweise an die speziellen Anforderungen von programmierbarer Logik angepasst wurden und synthesefähig sind. Methoden, die auf Betriebssystem-eigene Funktionen, wie zum Beispiel die dynamischen Speicherallozierung oder die Ausgabe im Terminal, zugreifen wurden entfernt oder angepasst.

Die Bibliothek ist auf die Verarbeitung von Datenströmen ausgerichtet wodurch es keinen wahlfreien Zugriff auf die Pixel eines Bildes gibt. Die in der Anwendung dafür benutzte OpenCV-Funktion „at(int i, int j)“ ist in der Implementierung von „hls::mat“ nicht vorhanden. Die Funktionen „read()“ und „write(...)“ sind die Kopplung zwischen

dem AXI-Stream und der Verarbeitung der Bilddaten, indem sequentiell vom Stream gelesen beziehungsweise auf diesen geschrieben wird. Dadurch ist die Implementierung einer Bufferstruktur erforderlich, die in Abschnitt 4.1.3 erläutert wird.

4.1.2 Port-Level- und Block-Level-Interfacing

Das Port-Level-Interface der Synthese wird implizit über die Top-Level-Function oder explizit durch Interface-Direktiven definiert. Sind die Signale nicht explizit oder unzulässig definiert, wird das Default-Protokoll des Funktionsargument-Typs angewendet (Abbildung 4.1) [CEE14].

Argument Type	Variable			Pointer Variable			Array			Reference Variable		
	pass-by-value			pass-by-reference			pass-by-reference			pass-by-reference		
Interface Type[a]	I	IO	O	I	IO	O	I	IO	O	I	IO	O
ap_none	**D**	-	-	**D**	S	S	-	-	-	**D**	S	S
ap_stable	S	-	-	S	S	-	-	-	-	S	S	-
ap_ack	S	-	-	S	S	S	-	-	-	S	S	S
ap_vld	S	-	-	S	S	**D**	-	-	-	S	S	**D**
ap_ovld	-	-	-	-	**D**	S	-	-	-	-	**D**	S
ap_hs	S	-	-	S	S	S	S	-	S	S	S	S
ap_memory	-	-	-	-	-	-	**D**	**D**	**D**	-	-	-
bram	-	-	-	-	-	-	S	S	S	-	-	-
ap_fifo	-	-	-	S	-	S	S	-	S	S	-	S
ap_bus	-	-	-	S	S	S	S	S	S	S	S	S
axis	S	-	-	S	-	S	S	-	S	S	-	S
s_axilite	S	-	S	S	S	S	-	-	-	S	S	S
m_axi	-	-	-	S	S	S	S	S	S	S	S	S

a. Reading along the row: **I** = input port; **IO** = inout (bidirectional) port; **O** = output port.

Abbildung 4.1: Zuweisung der Interface-Protokolle (D=Default, S=Supported), je nach Funktionsargument-Typ [CEE14]

Bild- und Videoverarbeitungsfunktionen, die mit der HLS synthetisiert werden und die OpenCV-Bibliothek nutzen, erhalten die Daten über ein AXI4-Stream-Interface [Xil15c]. Dafür ist eine Anpassung der Übergabeparameter der C++-Implementierung vorzunehmen. Der C++-Datentyp für das AXI4-Stream-Interface lautet *hls::stream<T>*, welcher als Template den zu streamenden Datentyp erhält.

Die Bildverarbeitungsfunktion ist mit zwei AXI4-Streams gekoppelt: einer für das Empfangen der Daten und einer für das Senden der verarbeiteten Daten. Der Funktionsprototyp lautet:

```
void binarisation(AXI_STREAM& INPUT_STREAM, AXI_STREAM& OUTPUT_STREAM);
```

Diese Funktion wird in der HLS in den Projektoptionen als Top-Level-Function definiert, aus deren Parametern die Schnittstellen des IP-Cores (vgl. Abbildung 4.2) synthetisiert werden.

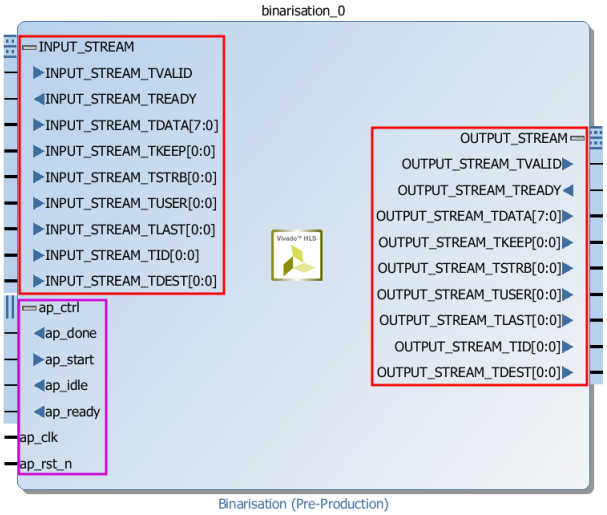


Abbildung 4.2: Das High-Level-Syntheseergebnis mit dem definierten Interfacing, das aus zwei AXI4-Streams (rot umrandet) und den Block-Signalen (violett umrandet) besteht

Die Block-Level-Protokolle sind implizite Signale, die den HLS-IP-Core steuern. Zu den Block-Signalen gehört die Clock, der Reset und ein Steuerungsinterface, die in dem *ap_ctrl_hs*-Protokoll zusammen gefasst sind (vgl. Abbildung 4.2, violette Markierungen). Alternativ kann mit der Definition von *ap_ctrl_none* über eine Direktive das Block-Level-Interfacing deaktiviert werden, was eine manuelle Blocksteuerung über Port-Level-Protokolle erforderlich macht [CEE14]. Für die Binarisierung wurde das standardmäßig aktive *ap_ctrl_hs*-Protokoll verwendet.

4.1.3 Buffering der Bilddaten des AXI4-Streams

Die Ortsoperationen der Binarisierung erfordern einen wahlfreien Zugriff auf die Bilddaten, da sie außerhalb des Bildzeilenstroms auf die Pixel in einem Tile zugreifen. Buffering realisiert dies durch das Zwischenspeichern von Daten, die über einen AXI4-Stream an einen IP-Core übergeben werden. Im HLS-Tool wurde über eine Array-Struktur eine Buffervariante implementiert. Es handelt sich um einen einzelnen Zeilenbuffer, in dem sowohl Inputs, als auch Outputs für 10 Bildzeilen zwischengespeichert werden (vgl. Abbildung 4.3). Zu Beginn wurden Buffervarianten implementiert und mit dem Synthese-Report der High-Level-Synthese analysiert. Ein einzelner Zeilenbuffer hat sich als ressourcensparend und schnell erwiesen [Kem15b].

Die Bildzeilen werden vom Input-AXI-Stream mit der Funktion *read()* gelesen und in den In-Out-Linebuffer geschrieben. Nach erfolgter Bildverarbeitung werden die Daten in den In-Out-Linebuffer zurückgeschrieben, wodurch die ursprünglichen Bilddaten überschrieben werden. Von dem Buffer werden die Daten mit der *write(...)*-Funktion an den Outpu-AXI-Stream übergeben. Die nachfolgenden Abschnitte des Bildes werden aus dem In-AXI-Stream in den Buffer geladen und die Verarbeitung wird weitergeführt.

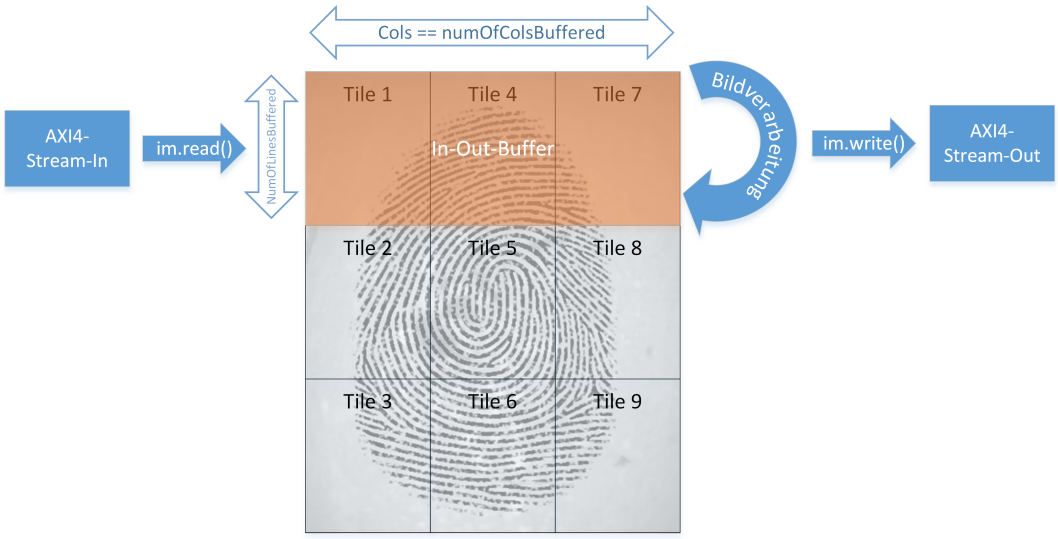


Abbildung 4.3: Einzelner In-Out-Linebuffer (orange) für das Zwischenspeichern eines Bildabschnittes (Bildparameter: vgl. Kapitel 3)

Das AXI4-Streaming wird, durch einen integrierten Buffer und den Signalen TVALID und TREADY [Xil11], bei Unterbrechungen für die Bildverarbeitung an der letzten Stelle fortgeführt.

Die implementierte C/C++-Funktionalität wird im High-Level-Synthese-Tool über die Funktion „Run C Synthesis“ vom HLS-Compiler in ein RTL-Modell übersetzt. Es werden für die ausgewählte „Solution“ SystemC-, Verilog- und VHDL-Dateien generiert und ein Synthesereport erstellt. Das Konzept der Solutions realisiert das Testen von Konfigurationen (z. B. Länge einer Takt-Periode), Constraints und Direktiven für eine HLS-Implementierung, ohne dass jeweils ein neues Projekt erstellt werden muss. In Kapitel 6 werden die Ergebnisse der High-Level-Synthese erläutert.

4.2 Verifikation mit C-Testbench und Co-Simulation

Das HLS-Tool bietet die Verifikation der C++-HLS-Anwendung und des Syntheseergebnisses. Beides greift auf eine selbstprüfende C-Testbench zurück, die für eine Anwendung jeweils zu erstellen ist. Die Funktionalität der Anwendung wird mit der C-Simulation

überprüft, die die Testbench ausführt. Die folgende Co-Simulation ist eine gekoppelte Ausführung der C-Simulation und des Syntheseergebnisses mit Einbeziehung der Daten der C-Simulation.

4.2.1 C-Testbench zur Funktionsverifikation von C++-Modellen

Die C-Simulation des HLS-Tools erwartet eine Testbench, die Testdaten an das *Device Under Test* (DUT) übergibt, das Ergebnis analysiert und mit Referenzdaten (*Golden Reference*) vergleicht [Xil14a]. Stimmen die Ergebnisdaten mit den Referenzdaten überein, ist die Implementierung korrekt und der Test endet mit einer positiven Meldung im Ausgabeterminal des Tools.

Das zu testende Modul ist ein Bildverarbeitungsalgorithmus, der mit OpenCV-Funktionen arbeitet und in C++ implementiert wurde. Die Testbench liest ein Bild ein (vgl. Abbildung 4.4), übergibt die Bilddaten dem DUT (HLS-kompatibler C++-Code) und vergleicht die Ergebnisdaten des DUTs mit einem Bild, das in der ursprünglichen C++-Implementierung als „Golden Reference“ generiert wurde [Kem15a].

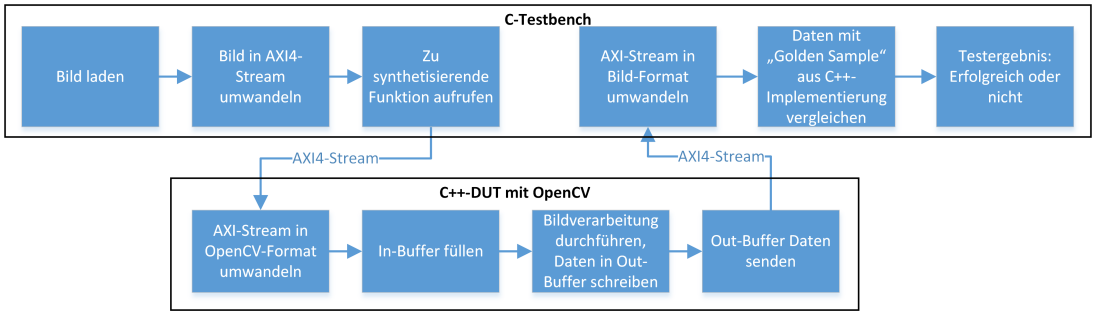


Abbildung 4.4: Funktionsschritte der C-Simulation mit einer Testbench (TB) und dem DUT (Device Under Test), welches mit AXI4-Stream in die Testbench eingebettet wird

Zusätzlich zur Bildverarbeitungsbibliothek mit synthesefähigen Funktionen (vgl. Abschnitt 4.1.1) steht in dem HLS-Tool eine Bildverarbeitungsbibliothek mit nicht-synthesefähigen Funktionen zur Verfügung (hls_opencv.h). Sie ist für den Einsatz in Testbenches konzipiert und bietet u.a. Dateioperationen zum Laden und Speichern von Bilddateien an.

Die Bilddaten, die die Testbench an das DUT übergibt, werden aus einer Datei mit dem Parameter *CV_LOAD_IMAGE_GRAYSCALE* eingelesen (vgl. Anhang 3). Der Parameter legt fest, dass das Bild als 8-Bit-Grauwertbild geladen wird. Für die Kommunikation mit dem DUT werden zwei 8-Bit-breite AXI-Streams erstellt, einer für das Senden der Bilddaten und einer für das Empfangen der verarbeiteten Bilddaten. Das geladene Bild wird mithilfe einer Bibliotheksfunktion in das AXI4-Stream-Format umgewandelt und die Top-Level-Funktion des Binarisations- und Segmentierungsalgorithmus wird aufgerufen:

```
binarisation(src_axi, dst_axi);
```

Die Daten, die die Testbench über den zweiten AXI-Stream vom DUT erhält, werden in das OpenCV-Bildformat zurückgewandelt. Dieses Ergebnisbild wird Pixel für Pixel mit einem Golden Sample verglichen, das mit der ursprünglichen C++-Implementierung ohne HLS-Anpassungen erstellt wurde:

```
//load "golden sample"
IplImage* goldenSample=cvLoadImage("Pfad\\zur\\Datei\\GoldenSample.bmp"
    );
int errorcnt = 0;

//compare "golden sample" with the processed image
for(int i = 0; i<src->width; i++){    //iterate cols
  for(int j = 0; j<src->height; j++){   //iterate rows
    if(!(CV_IMAGE_ELEM(dst, unsigned char, j, i) == CV_IMAGE_ELEM(
        goldenSample, unsigned char, j, i))){
      errorcnt++;
    }
  }
}
```

Haben zwei Pixel mit den selben Indizes unterschiedliche (Grau-)Werte, wird ein Error-Zähler inkrementiert. Die Testbench ist erfolgreich durchgelaufen, wenn der Error-Zähler 0 ist und alle Pixel übereinstimmen. Die Testbench signalisiert dem HLS-Tool mit einem Rückgabewert von 0 den erfolgreichen Abschluss.

4.2.2 C/RTL Co-Simulation

Die Post-High-Level-Synthese-Funktionalität wird im HLS-Tool mit der C/RTL-Co-Simulation sichergestellt. Die Co-Simulation verwendet die C-Testbench und besteht aus drei Schritten (vgl. Abbildung 4.5):

1. Ausführung der C-Simulation (vgl. Abbildung 4.4) und Speicherung der Eingabedaten der Testbench als Input-Vektoren: TV in .dat (vgl. Abbildung 3.2a)
2. Simulation des HLS-Ergebnisses (RTL-Modul) mit den Input-Vektoren der C-Simulation, Speicherung der Ergebnisdaten des DUTs als Output-Vektoren: TV out .dat (vgl. Abbildung 3.2b)
3. Übertragung der Output-Vektoren in die C-Testbench nach Aufruf der Top-Level-Funktion

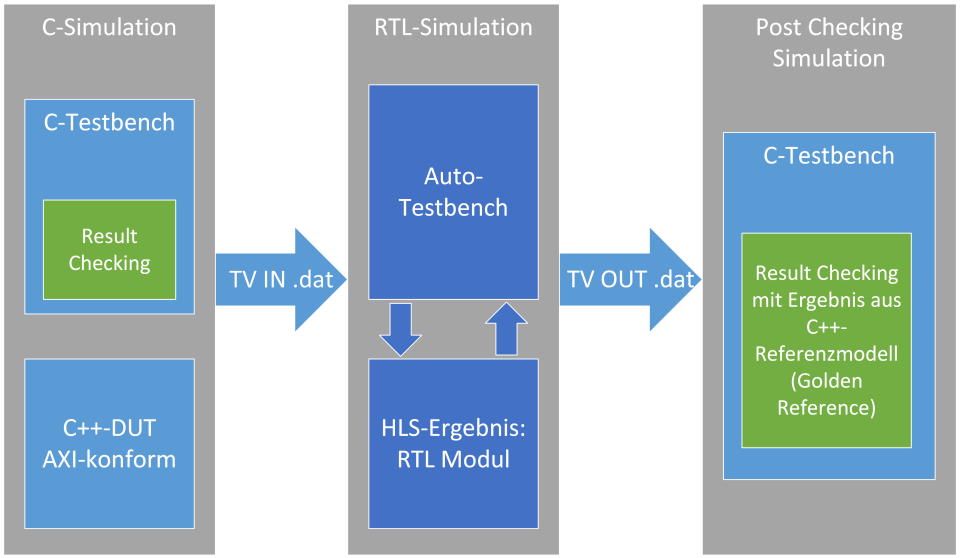


Abbildung 4.5: Co-Simulation: Datenfluss der Testvektoren (TV) für die RTL-Simulation eines HLS-RTL-Moduls [Xil14a]

Die Testvektoren werden im Ordner der Solution gespeichert. Für jedes Signal des AXI-Streams existiert eine eigene Datei, in der die Daten als hexadezimale Zahlen gespeichert werden. Der Nachfolgende Vergleich zeigt Ausschnitte der Dateien *c.binarisation.autotvin_INPUT_STREAM_V_data_V.dat* (links) und *rtl.binarisation.autotvout_OUTPUT_STREAM_V_data_V.dat* (rechts).

```
[[[runtime]]]
[[transaction]] 0
0x05f
0x05f
0x05e
[...]
0x0d6
0x09d
0x099
0x0d0
0x079
0x040
0x03f
0x0cc
[...]
0x06d
0x06c
0x06b
[[/transaction]]
[[[/runtime]]]
```

```
[[[runtime]]]
[[transaction]]    0
0xff
0xff
0xff
[...]
0xff
0xff
0x00
0xff
0x00
0x00
0x00
0xff
[...]
0xff
0xff
0xff
[[/transaction]]
[[[/runtime]]]
```

Wie bereits am Namen der Dateien zu erkennen ist, stellt der linke Ausschnitt den Eingangsdatenstrom des Input-AXI-Streams der C-Testbench an das C++-DUT. Der rechte Ausschnitt ist der Ergebnisdatenstrom des Output-AXI-Streams des RTL-Modells. Die Daten des Output-Datenstroms bestehen aus Schwarz(0x00)- und Weiß(0xff)-Werten, eine Binarisierung wurde vom RTL-Modell durchgeführt. Der mittlere Teil wurde den Zeilen 102179 bis 102186 entnommen, die anderen Teile stammen je vom Anfang bzw. Ende der Dateien.

Die Übertragung der Output-Vektoren in die C-Testbench wird nach Aufruf der Top-Level-Funktion realisiert, sodass die Ergebnisdaten des RTL-Moduls mit der (*Golden Reference*) verglichen werden. Wenn die Ergebnisdaten mit der (*Golden Reference*) übereinstimmen ist die Co-Simulation erfolgreich beendet und die Funktionalität des HLS-Moduls verifiziert.

Für die Co-Simulation gelten folgende Voraussetzungen [Xil14a, S. 230]:

- Testbench ist self-checking: Rückgabewert von 0, wenn die Ergebnisse korrekt sind und die Daten mit dem (*Golden Reference*) übereinstimmen, Rückgabewert von ungleich 0 wenn ein Fehler auftrat
- ap_ctrl_hs oder ap_ctrl_chain als Block-Level-Protokoll (vgl. oder die Implementierung ist rein kombinatorisch
- Eingeschränkte Verwendung von Optimierungsdirektiven

Alle Voraussetzungen wurden bei der Implementation der Binarisierung und Segmentierung erfüllt, weshalb eine Co-Simulation durchgeführt werden konnte.

4.2.3 Verifikationsergebnis der Bildverarbeitung

Die Verifikation der Implementierten Funktionalität wurde sowohl durch die C-Simulation, als auch durch die C/RTL-Co-Simulation erreicht. Die C-Simulation ergab:

```
Starting C simulation ...
[...]
Start
Test Passed!
@I [SIM-1] CSim done with 0 errors.
@I [LIC-101] Checked in feature [HLS]
```

Die Co-Simulation hatte als Ergebnis:

```
Starting C/RTL cosimulation ...
[...]
Time: 19527955400 ps  Iteration: 0  Process: /apatb_binarisation_top/
    generate_done_cnt_proc  File: C:/Users/Kemmler/
    binarisationInOutLittle/solution1/sim/vhdl/binarisation.autotb.vhd
```

```
finish called at time : 19527955400 ps
run: Time (s): cpu = 00:00:00 ; elapsed = 00:01:13 . Memory (MB): peak
    = 57.641 ; gain = 0.000
## quit
INFO: [Common 17-206] Exiting xsim at Thu Oct 13 17:40:29 2016...
@I [SIM-316] Starting C post checking ...
Start
Test Passed!
@I [SIM-1000] *** C/RTL co-simulation finished: PASS ***
@I [LIC-101] Checked in feature [HLS]
```

Beide Simulationen wurden mit positivem Abschluss erfolgreich durchgeführt. Damit ist verifiziert, dass der für die HLS angepasste C++-Code und das RTL-Modul der High-level-Synthese die selbe Funktionalität haben wie die ursprüngliche C++-Implementierung, die auf einem Betriebssystem realisiert wurde (vgl. Kapitel 3).

5 Systemintegration der HLS-IP-Cores auf dem Zynq

Die Integration des in Kapitel 4 vorgestellten HLS-IP-Cores auf dem Zynq, sodass es Testdaten von einer Bare-Metal-ARM-Anwendung erhält und verarbeitet, wird in diesem Kapitel dokumentiert. Die FPGA- und Prozessor-Komponenten werden in einen gemeinsamen Kontext gesetzt und die Schritte ihrer Konfiguration über die Software erläutert.

5.1 Komponentenaufbau zur Eingliederung des HLS-IP-Cores in der PL

Für die Anbindung der Binarisierungs-IP an das Zynq Processing-System sind weitere IP-Cores erforderlich, damit der HLS-IP-Core Testdaten vom externen RAM erhält. Für das AXI-Protokoll bietet Xilinx FPGA-Bibliothekselemente, die in der Entwicklungsumgebung Vivado integriert sind. Die für die Anbindung eines AXI-Stream-Moduls an den externen RAM erforderlichen Komponenten werden im Folgenden erläutert.

Das Blockdesign (Abbildung 5.1) zur HLS-IP-Integration realisiert zwei unabhängige AXI-Interconnects. Einer ist für die DMA-Verbindung zwischen dem VDMA-Core und der HP(High Performance)-AXI Schnittstelle des Zynq-Processingsystems zuständig. Der zweite verbindet das AXI4-Lite-Interface des VDMA-Cores mit dem Zynq-Processingsystem, worüber der VDMA-Core konfiguriert und gesteuert wird (vgl. Abschnitt 5.2). Dieser AXI-Interconnect verbindet die AXI-Lite-Konfigurationsschnittstelle mit einem AXI-GP(General-Purpose)-Port des Zynq-SoCs.

Die Interruptleitungen des VDMA-Cores wurden aktiviert, damit nicht zyklisch auf das Ende-Signal einer Transaktion geprüft werden muss (Polling). Interrupts führen gegenüber Polling zu einer geringeren Latenz. Für den Read- und den Write-Kanal gibt es je einen Interrupt, die mit einer Konkatenations-IP zu einem 2-Bit-Datenvektor zusammen gefügt werden. Durch die Konkatenation sind die Interruptquellen portkompatibel zu den Shared-Interrupt-Eingängen des Zynq-SoC.

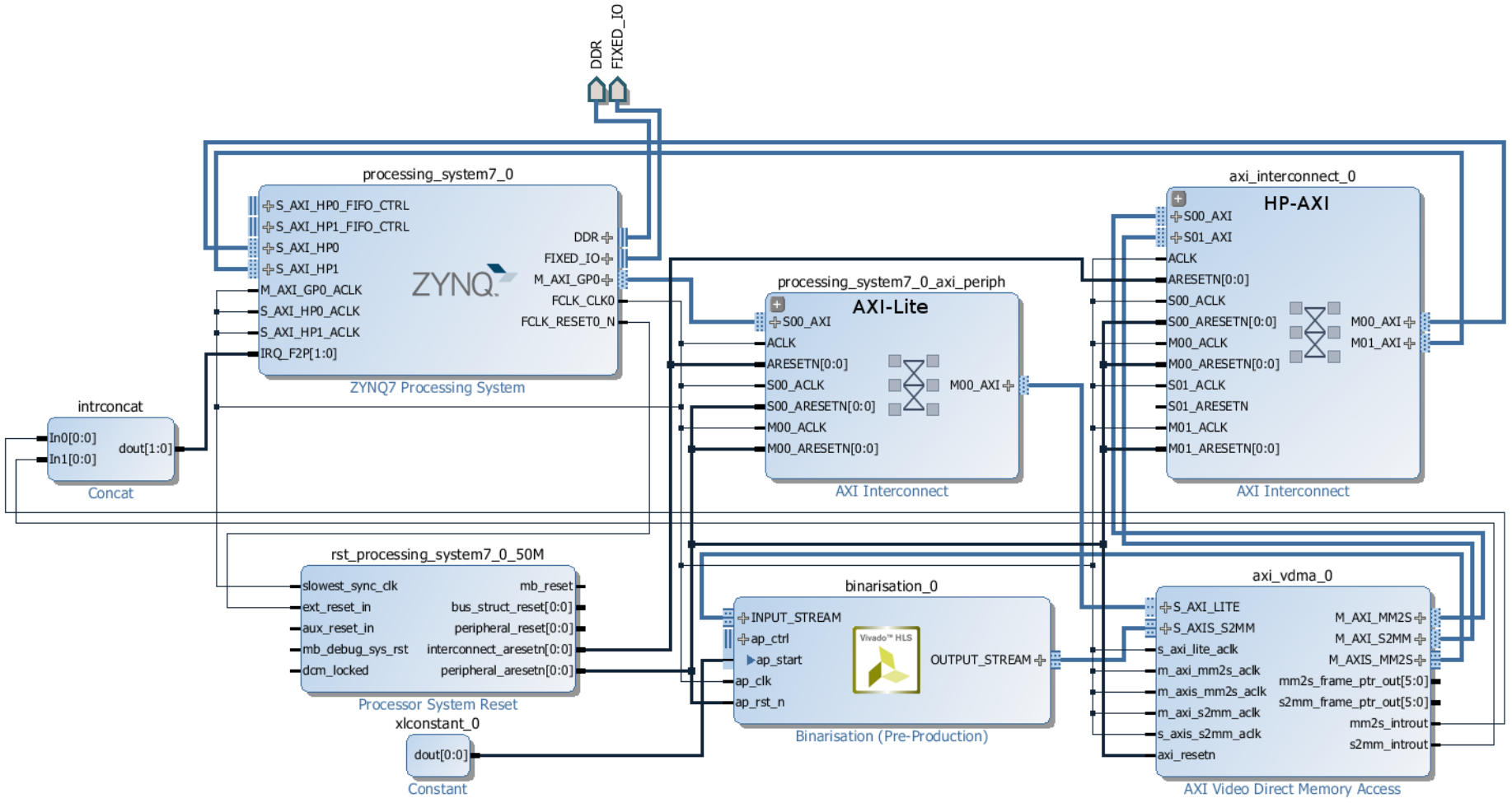


Abbildung 5.1: Integration des Binarisierungs-HLS-IP-Cores in das kombinierte FPGA-Prozessor-System erzeugt mit dem Vivado IP-Integrator [Xil15d]

Für die Read- und Write-Anbindung der VDMA-Komponente wurden zwei High-Performance(HP)-Axi-Slave-Schnittstellen in der Zynq IP aktiviert. Im Gegensatz zu den General-Purpose(GP)-AXI-Schnittstellen sind die HP-AXI-Schnittstellen nicht mit dem Central-Interconnect verbunden, sondern haben über dem Memory-Interconnect eine direkte Leitung zum DDR-Controller (vgl. Abbildung 2.5). Für die PL-Komponenten ist eine Clock aktiv, die mit 100 MHz taktet.

Für die Interrupts der PL wurde der PL-to-PS-Interrupt im Zynq aktiviert. Dieser Port kann durch eine Breite von 16 Bit bis zu 16 Interruptquellen verarbeiten.

Der INPUT-Stream des HLS-Cores (binarisation_0, Abbildung 5.1) ist eine AXI-Stream-Slave-Schnittstelle, die mit dem MM2S-Master des VDMA-Cores (axi_vdma_0, M_AXIS_MM2S) verbunden ist. Der Read-Channel des VDMA-Cores bezieht die adressbehafteten Daten als AXI-Master (M_AXI_MM2S) über die Slave-Schnittstelle des AXI-Interconnects (axi_interconnect_0, S00_AXI), welcher die Daten als Master von der HP-AXI-Slave-Schnittstelle des Zynq-SoC (processing_system7_0), und damit vom RAM, erhält. Das Lesen von RAM-Daten geschieht direkt über den Memory-Controller, ohne Beteiligung des ARM-Processors. Der AXI-Interconnect (axi_interconnect_0) hat zwei Kanäle, um den Read- und den Write-Kanal des VDMA- bzw. des HLS-Cores zu bedienen.

Der OUTPUT-Stream des HLS-Cores ist eine AXI-Stream-Master-Schnittstelle, die mit dem S2MM-Slave des VDMA-Cores (S_AXIS_S2MM) verbunden ist. Der Write-Channel des VDMA-Cores gibt die Daten als AXI-Master (M_AXI_S2MM) adressbehaftet an die Slave-Schnittstelle des AXI-Interconnects (S01_AXI) weiter, welcher als Master die Daten an die HP-AXI-Slave-Schnittstelle des Zynq-SoCs übergibt. Über dem Memory-Controller werden die Daten direkt in den RAM geschrieben, ohne das der ARM-Prozessor beteiligt ist.

Das Block-Level-Interface (vgl. Abschnitt 4.1) wird partiell genutzt, indem dem HLS-Core mit einem konstanten High-Pegel (xlconstant_0, Abbildung 5.1) an dem ap_start-Port fortwährend signalisiert wird, dass Daten kommen. Die Arbeit nimmt der HLS-Core erst auf, wenn über das Handshaking (vgl. Abschnitt 2.4) des AXI4-Streams Daten tatsächlich ankommen.

Der VDMA-Core ist für eine Datenbreite von 8-Bit konfiguriert und der Read- sowie der Write-Kanal sind aktiv. Der VDMA-Core besitzt eine AXI-Lite-Slave-Schnittstelle, die mit dem zweiten AXI-Interconnect des Systems (processing_system7_0_axi_perph, Abbildung 5.1) an der Masterschnittstelle verbunden ist. Der Interconnect ist als Slave an der GP-AXI-Master-Schnittstelle des Zynq-SoC verbunden. AXI4 und AXI4-Lite besitzen Write- und Readkanäle und sind direkt zueinander kompatibel, weshalb an dieser Stelle keine IP für das Übersetzen der Protokolle und der dezidierten Realisierung von Read- und Write-Kanälen erforderlich ist. Die AXI4-Lite-Verbindung wird genutzt, um über die bare-metal ARM-Applikation die Register des VDMA-Cores zu beschreiben und auszulesen. Das wird in der ARM-Software genutzt, um den Status auszulesen, die Interrupts zu initialisieren und die Read- und Write-Transaktionen zu steuern.

Im gesamten System gibt es für die FPGA-Komponenten eine Clock-Domain, deren Quelle eine 100 MHz-Clock des Zynq-SoCs ist. Die Resets der FPGA-Komponenten werden über den Reset des Zynq-SoCs ausgelöst und sind synchron.

Das System wird dem Vivado-Workflow folgend synthetisiert:

1. Verifikation des Block-Designs
2. Generierung des HDL-Wrappers
3. Synthetisierung
4. Implementierung
5. Generierung des Bit-Streams

Sind diese Schritte erfolgreich durchgeführt worden, wird das Hardware-Package für die Entwicklungsumgebung *Vivado SDK* zur Erstellung der ARM-Software exportiert.

5.2 ARM: Software-Konfiguration mit einer C-Applikation

Das FGPA-System wird über eine bare-metal C-Applikation auf einem ARM-Kern initialisiert, gesteuert und mit Testdaten versorgt. Das in Vivado erstellte Hardwaresystem (vgl. vorherigen Abschnitt) wird hierfür exportiert und *Vivado SDK* gestartet, das auf Eclipse basiert. Durch die Exportfunktion wird ein SDK-Projekt erstellt, das das Hardwaresystem in Form einer „Hardware-Plattform“ als Basis hat. Daraus wird im SDK das Board-Support-Package (BSP) erstellt, das Treiberfunktionen zu den synthetisierten FPGA-Komponenten enthält. Zudem wird mit dem BSP festgelegt, auf welchem der zwei ARM-Kerne die Applikation ausgeführt wird. Über die Konfiguration des BSP werden Compiler-Flags gesetzt und mitgelieferte Bibliotheken, zum Beispiel für einen TCP/IP-Stack oder für das FAT-Dateisystem, inkludiert. Eine Zusammenfassung über Einstellungen, Eigenschaften und Treiber des BSPs liefert die system.mss-Datei.

Folgende Schritte werden von der C-Applikation durchgeführt:

1. Definition des VDMA-Buffers
2. Generierung von Testdaten im VDMA-Buffer
3. Initialisierung des VDMA-Treibers
4. Konfiguration des Read- und des Write-Kanals des VDMA-Cores
5. Initialisierung des Generic-Interrupt-Controllers (GIC)
6. Registrieren der Interrupt-Service-Routinen (ISR) und Verarbeitung der Interrupts
7. Anstoßen der Read- und Write-Transaktionen
8. Fehlerbehandlung

Definition des VDMA-Buffers

Für die VDMA-Transaktionen ist ein Speicherbereich im externen RAM zu reservieren, von dem gelesen und in dem geschrieben wird. Für den Read- und den Write-Kanal werden Basis-Speicheradressen festgelegt:

```
#define READ_ADDRESS_BASE 0x16000000
#define WRITE_ADDRESS_BASE  0x17000000
```

Die angegebenen Adressen befinden sich im mittleren Teil des externen RAMs, dessen Speicherbereich die Adressen von 0x00100000 bis 0x1FFFFFFF (510 MiB) umfasst. Damit wird sichergestellt, dass der Programmspeicher der C-Anwendung nicht überschrieben wird. Der Speicherbedarf für ein Bild mit den Dimensionen der Testbilder (vgl. Kapitel 3) liegt bei 229.600 Byte, sodass der reservierte Speicherbereich in Höhe von über 16 Millionen Byte (16.384 KiB) pro Kanal ausreicht.

Generierung von Testdaten im externen RAM

Die Testdaten sind eine Matrix von 8-Bit Werten, die wie Bilddaten behandelt werden. Es wird als Testmuster in einem zweidimensionalem Array abwechseln die Werte 40 und 140 geschrieben, die durch die Binarisierung (vgl. Abschnitt 3.1) zu den Werten 0 und 255 umgewandelt werden sollen. Die Testdaten werden an die Stelle des VDMA-Read-Speichers kopiert:

```
testcopy = (u8 *) READ_ADDRESS_BASE;
for(i=0; i<COLS; i++){
  for(j=0; j<ROWS; j++){
    *testcopy =  testdata[j][i];
    testcopy++;
  }
}
```

Der Zeiger testcopy wird auf die Read-Basisadresse gesetzt und an der dereferenzierten Stelle werden die Testdaten abgelegt. Der Zeiger wird inkrementiert, damit nacheinander die Speicherzellen beschrieben werden.

Initialisierung des VDMA-Treibers

Der VDMA-Treiber wird über zwei Funktionen initialisiert. Die Hardwarekonfiguration wird ausgelesen und die VDMA-Engine mit der ausgelesen Konfiguration initialisiert:

```
Config = XAxiVdma_LookupConfig(DMA_DEVICE_ID);
XAxiVdma_CfgInitialize(&AxiVdma, Config, Config->BaseAddress);
```

Die Initialisierungsfunktion setzt die Interrupts zurück und initialisiert den Read- und den Write-Channel über den Buffer Descriptor Ring. Nach der Initialisierung wird ein Reset der Kanäle durchgeführt.

Konfiguration des Read- und des Write-Kanals des VDMA-Cores

Die Kanäle werden über das XAxiVdma_DmaSetup-Struct konfiguriert, indem die Structelemte gefüllt werden und es der XAxiVdma_DmaConfig-Treiberfunktion übergeben wird. Die konfigurierten Eigenschaften umfassen:

- horizontale Größe des Bildes: *HoriSizeInput*, entspricht der Anzahl der Rows
- vertikale Größe des Bildes: *VertSizeInput*, entspricht der Anzahl der Columns
- Anzahl der Bytes für eine Horizontale Bildlinie: *Stride*, entspricht der Anzahl der Rows, da jeder Pixel ein Byte groß ist
- Option zur Aktivierung eines zirkulären Buffers: *EnableCircularBuf*, aktiviert
- Optionen zur Frame-Synchronisation: *EnableSync* und *PointNum*, deaktiviert
- Option zur Aktivierung eines Frame-Zählers: *EnableFrameCounter*, deaktiviert
- Option zur Aktivierung einer Parking-Funktion, wodurch immer das selbe Bild übertragen wird und der Adresszähler nicht inkrementiert wird: *FixedFrameStoreAddr*, aktiviert
- Startadresse des im RAM definierten Speichers für das Bild: *FrameStoreStartAddr*, auf READ_ADDRESS_BASE bzw. WRITE_ADDRESS_BASE gesetzt

FrameStoreStartAddr ist als Array realisiert, wodurch das Senden einer Bildfolge implementiert werden kann. Für die Anwendung, die ein Test der Systemintegration darstellt, ist nur ein Bild vorgesehen und durch den Parking-Mode wird ohnehin immer wieder das selbe Bild gesendet.

Über die Funktion XAxiVdma_DmaConfig wird die Funktion XAxiVdma_ChannelConfig aufgerufen, welche die gesetzte Konfiguration in die Register des VDMA-Cores überträgt.

Initialisierung des Interrupt- und des Exception-Systems

Der VDMA-Core besitzt zwei Interrupt-Leitungen, je eine für jeden Kanal. Die Interruptfunktionalität des VDMA-Cores wird folgendermaßen konfiguriert:

- Treiberinitialisierung des Interrupts (vgl. „Initialisierung des VDMA-Treibers“)
- Optional: Definition der Priorität und der Trigger-Form (Active-HIGH-Level-sensitiv oder Rising-Edge-sensitiv)
- Verbindung der beiden Interrupts mit dem GIC der ARM-CPU und Definition der jeweiligen Interrupt-Handler
- Aktivierung der Interrupts

- Initialisierung des Exception-Systems
- Registrierung des Exception-Handlers
- Aktivierung Exception-Systems
- Registrierung der ISRs an den Handlern

Mit der Abfolge werden die Interrupts des VDMA-Cores mit dem GIC der CPU verbunden und das Exception-System eingerichtet. Als letzter Schritt werden die ISRs, *ReadCallBack* und *WriteCallBack*, über die Funktion *XAxiVdma_SetCallBack* mit den entsprechenden Handlern, *XAxiVdma_ReadIntrHandler* und *XAxiVdma_WriteIntrHandler*, verbunden. Die Exceptions des VDMA-Cores werden über einen eigenen Handler bedient, der mit entsprechenden ISRs, *ReadErrorCallBack* und *WriteErrorCallBack*, verbunden wird.

Die Quelltextanalyse der Funktion *XScuGic_Enable* zeigt, dass sie einen Interrupt aktiviert, indem im FPGA-Modul im Set-Interrupt-Enable-Register an der zur Interruptquelle korrespondierenden Stelle [Xil15a] ein Bit auf 1 gesetzt wird.

Anstoßen der Read- und Write-Transaktionen

Der VDMA-Core ist, im Gegensatz zum DMA-Core, auf einen kontinuierlichen Bilderstrom ausgelegt. Der Bilderstrom für beide Kanäle wird mit

```
XAxiVdma_DmaStart(InstancePtr, XAXIVDMA_READ);
XAxiVdma_DmaStart(InstancePtr, XAXIVDMA_WRITE);
```

ausgelöst. In diesem Fall ist der Parking-Mode aktiv, sodass immer wieder die selben Daten aus dem definierten Speicherbereich gesendet werden. Der Treiber setzt die Bits in den Control-Registern des VDMA-Cores über *XAxiVdma_ChannelStart*.

Fehlerbehandlung

Die Treiberfunktionen haben einen Rückgabewert, der über Erfolg oder Probleme beim Ausführen Auskunft gibt. Die Werte sind in xstatus.h definiert und werden bei Funktionsaufrufe aufgefangen, beispielsweise:

```
Status = XAxiVdma_DmaStart(InstancePtr, XAXIVDMA_READ);
if (Status != XST_SUCCESS) {
  xil_printf("Start read transfer failed %d\r\n", Status);
  return XST_FAILURE;
}
```

Sobald der Rückgabewert nicht XST_SUCCESS entspricht, wird eine Fehlermeldung im Terminal ausgegeben und die Applikation abgebrochen, indem der Rückgabewert der Main-Funktion auf XST_FAILURE (=1) gesetzt wird.

5.3 FPGA-Debugging mit dem Integrated-Logic-Analyzer

Für das FPGA-Debugging ist eine Anpassung des implementierten Systems erforderlich, da die internen Signale des FPGAs nicht direkt vom Workstation-PC bzw. von Vivado abgegriffen werden können. Für die Workstation, die mit dem Zedboard zur Programmierung verbunden ist, wird eine Schnittstelle definiert, die interne Signale des FPGAs herausführt und an die Workstation weiter gibt. Hierfür bietet Vivado den Integrated-Logic-Analyzer (ILA) [Xil14b], welcher Chipscope des Vivado-Vorgängers ISE ablöst. Der ILA ist ein IP-Core, der in das Blockdiagramm des Systems integriert wird und mit den Signalen von Interesse verbunden wird. Über die Konfiguration des Cores wird die Anzahl und Bitbreite der zu analysierenden Signale (*Probes*) definiert und das Cross-Triggering-Feature kann optional aktiviert werden (vgl. Abbildung 5.2).

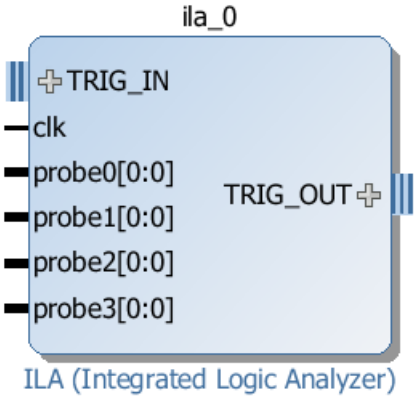


Abbildung 5.2: Integrated-Logic-Analyzer(ILA)-Core mit 4 Signaleingängen (Probes) und aktiviertem Cross-Triggering

Das Cross-Triggering-Feature dient der Synchronisation zwischen dem Debugging der ARM-Applikation und dem FPGA. Es wird in dem Zynq-Core und dem ILA-Core aktiviert und die beiden Kerne werden über die entstehenden Ports verbunden. Über den Hardware Manager von Vivado wird eine Verbindung zum FPGA über die USB-UART-Schnittstelle des Zedboards hergestellt. Nach der erneuten Synthese und Implementation wird die Bitstream-Datei auf das FPGA übertragen. Nachdem das FPGA programmiert ist, wird ein Trigger im Hardware-Manager definiert, der auf das gewünschte Signal horcht. Der Trigger kann konfiguriert werden, beispielsweise ob auf jede Änderung eines Signals reagiert wird, oder nur bei Änderung auf einen bestimmten Pegel. Über eine Schaltfläche wird das FPGA in den *Waiting for Trigger*-Modus versetzt. Im gleichzeitig geöffneten Vivado SDK wird eine Debug-Session gestartet, indem Breakpoints gesetzt werden und die Debug-Run-Configuration *Debug on Device (GDB)* gestartet wird. Löst die ARM-Software aus der Debug-Session des Vivado-SDKs durch die Steuerung der FPGA-Komponenten das Signal aus, auf dem ein Trigger im Vivado Hardware Manager horcht, wird der Trigger ausgelöst und ein Wellendiagramm gezeichnet. Das

Wellendiagramm umfasst die mit dem ILA verbundenen Signale über eine konfigurierte Zeitachse. Standardmäßig ist die Zeitachse auf 1024 eingestellt, was die Aufzeichung von 512 Takten vor Triggerauslösung und 512 Takte nach Triggerauslösung realisiert. Durch die im Wellendiagramm aufgezeichneten Werte ist das Nachvollziehen der Funktionalität des FPGA-Systems möglich (vgl. Abschnitt 6.3 für Debugging-Ergebnisse).

6 Ergebnisanalyse der HLS und der Systemintegration

Dieses Kapitel zeigt das Ergebnis der High-Level-Synthese auf und diskutiert die Ergebnisse des Systems. Dazu gehört eine Analyse des HL-Synthesereports und des generierten Quellcodes sowie die Ergebnisse der FPGA-Synthese und Implementierung.

6.1 High-Level-Syntheseergebnis

Der Synthesereport enthält Abschätzungen zum Timing und zum Ressourcenverbrauch und liefert eine Übersicht der generierten Interfaceports. In der *Analysis Perspective* des HLS-Tools wird dargestellt, wie viele Takte und Speicherressourcen pro Anweisung voraussichtlich benötigt werden. Es handelt sich um Abschätzungen, da das HLS-Tool keine genauen Kenntnisse über die FPGA-Synthese besitzt: die Logik-Optimierungen und das genaue Routing einer FPGA-Synthese sind unbekannt [Xil14a, S. 238]. Standardmäßig wird die Zeit einer Taktperiode auf ein Ziel von 10 ns (= 100 Mhz Taktfrequenz) gesetzt, worauf sich der HLS-Compiler ausrichtet. Der Compiler geht standardmäßig von einer Abweichung der Abschätzung von bis zu 1,25 ns aus. Daraus ergibt sich, dass die Taktperiode, auf die sich die HLS ausrichtet, 8,75 ns beträgt. Im Synthesereport werden die anvisierten 8,75 ns für eine Taktperiode angegeben, die 100 MHz werden vom HLS-Modul erreicht. Die Konfiguration wurde auf dem Standardwert belassen und ist für alle Projektvarianten gleich.

Die *Analysis Perspective* bietet die Detailanalyse der High-Level-Synthese (siehe Abbildung 6.1). Im Hauptbereich wird eine Zuordnung der Operationen zu den zugehörigen *Control States* (CS) dargestellt. Die CSs werden von der High-Level-Synthese verwendet, um für Operationen die Anzahl der Taktzyklen zu ermitteln [Xil13f, S. 12] und werden von C_0 beginnend bis C_n indexiert.

Die CS sind farblich codiert:

- Gelb: Schleifen
- Violett: Standardoperationen
- Grün: Sub-Blocks

Mit der Auswahl eines CS wird die gemappte Operation (Spalte „Operation\Control Step“ in Abb. 6.1) im C++-Sourcecode selektiert.

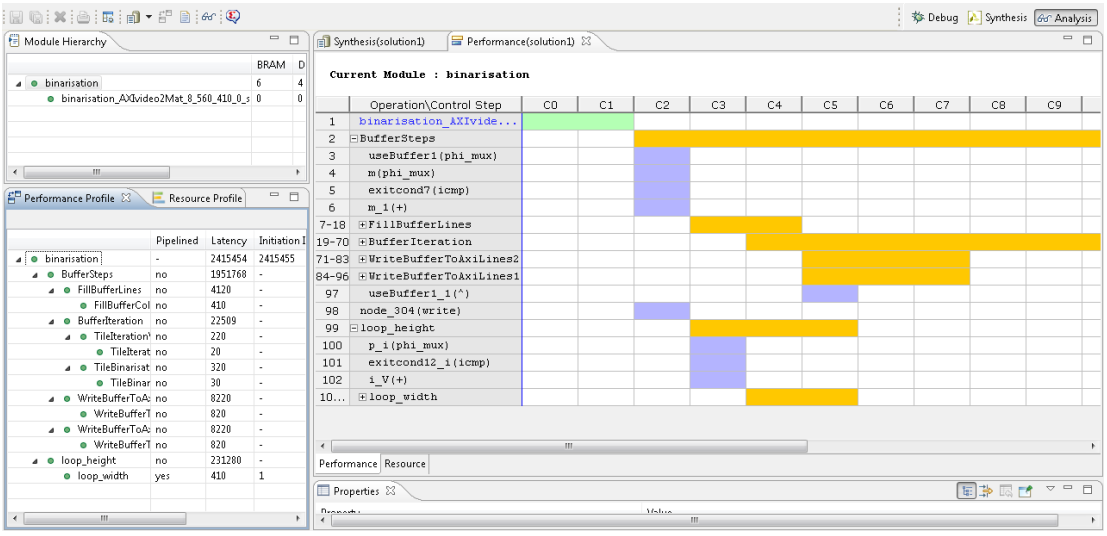


Abbildung 6.1: *Analysis Perspective* des HLS-Tools zur Visualisierung des Timings und des Ressourcenbedarfs, im Hauptfenster die Control States mit zugehöriger Operation

Die Synthese der Binarisierung mit dem einzelnen In-Out-Linebuffer ergibt folgende Performance- und Ressourcenbedarfsabschätzung:

estimated Clockcycle	max. Latency
8,75 ns	2414894

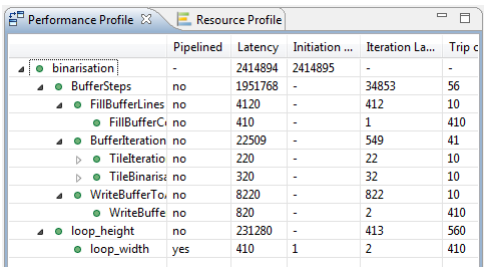
Tabelle 6.1: Performanceabschätzung des High-Level-Synthese-Ergebnisses

Ressource	BRAM_18k	DSP48E	FF	LUT
Verfügbar auf Zynq	280	220	106400	53200
Genutzt	4	4	435	722
Bedarf	~1%	~2%	<1%	~1%

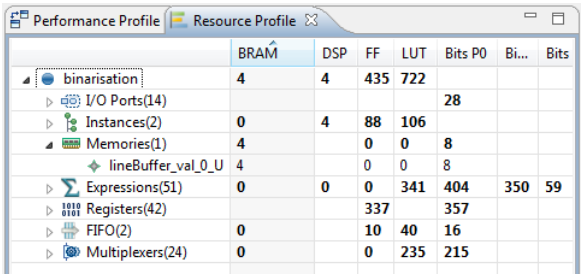
Tabelle 6.2: Ressourcenbedarfsabschätzung des High-Level-Synthese-Ergebnisses

BRAM_18K sind getaktete On-Chip-Blockram-Module mit einer Größe von 18 Kilobit (Kb) [Xil14c, S. 14]. Ein DSP48E ist ein Arithmetikschaltwerk, das einen $25 \cdot 18$ Zweier-Komplement-Multiplizierer und einen 48-Bit-Addierer zu einer steuerbaren MAC-Unit verbindet [Xil16c].

Für die Beispielbilder (vgl. Kapitel 3) ist der Linebuffer 10 Zeilen groß, was einen Speicherbedarf von $10 \cdot 410 = 4100$ Byte (~4 KB) ergibt. Zwei 18K-BRAMs (= 4,5 KB) würden für den einzelnen Linebuffer ausreichen. Der allozierte Speicher ist $(18 \cdot 4)/8 = 9$



(a) „Performance Profile“ der HLS



(b) „Ressource Profile“ der HLS

Abbildung 6.2: Timing- und Ressourcenbedarfsbschätzungen der High-Level-Synthese

KB groß (Abbildung 6.2 (b)), was dem Speicherbedarf von zwei 10-Zeilen-Linebuffern der Beispielbilder entspricht.

High-Level-Synthese mit Standard-Direktiven

Bei der Verwendung von AXI-Streams werden im User-Guide 902 [Xil14a] folgende Direktiven für die Steuerung der HLS nahe gelegt, damit die RTL-Implementierung weniger Ressourcen erfordert und eine geringere Latenz aufweist:

```
#pragma HLS INTERFACE axis port=INPUT_STREAM
#pragma HLS INTERFACE axis port=OUTPUT_STREAM
#pragma HLS dataflow
```

Die Direktive DATAFLOW führt zu Pipelining auf Task-Ebene [Xil14a, S. 181], was zur parallelen Ausführung von Funktionen und Schleifen führt. Die „INTERFACE“-Direktiven spezifizieren explizit das Interface, in diesem Fall zwei AXI-Streaming-Interfaces. Elaborationen im Masterprojekt [Kem15b] haben ergeben, dass zwischen der expliziten und impliziten Interfacespezifikationen der AXI4-Streams keine Unterschiede in Bezug auf Ressourcenbedarf und Latenz bei der HLS-Ergebnisabschätzung existieren. Die Unterschiede zur Synthese ohne Direktiven sind ausschließlich auf die Dataflow-Direktive zurückzuführen.

Die Synthese mit den im vorherigen Abschnitt vorgestellten Optimierungsdirektiven ergibt folgende Abschätzung:

estimated Clockcycle	max. Latency
7.45 ns	1951772

Tabelle 6.3: Performanceabschätzung des High-Level-Synthese-Ergebnisses mit den Optimierungsdirektiven

Ressource	BRAM_18k	DSP48E	FF	LUT
Verfügbar auf Zynq	280	220	106400	53200
Genutzt	4	4	443	724
Bedarf	~1%	~1%	<1%	~1%

Tabelle 6.4: Ressourcenbedarfsabschätzung des High-Level-Synthese-Ergebnisses mit den Optimierungsdirektiven

Die Clock-Periodendauer hat sich von 8,75 ns auf 7,45 ns verringert, was einer Verbesserung von ungefähr 15 % entspricht. Die Latenz hat sich um knapp ca. 1/5 reduziert, beim Ressourcenbedarf gibt es kaum Veränderungen.

Analyse des generierten VHDL-Modells

Das generierte VHDL-Modell besteht aus 8 hierarchischen .vhd-Dateien mit *binarisation.vhd* als Top-Level-Entität (vgl. Abbildung 6.3).

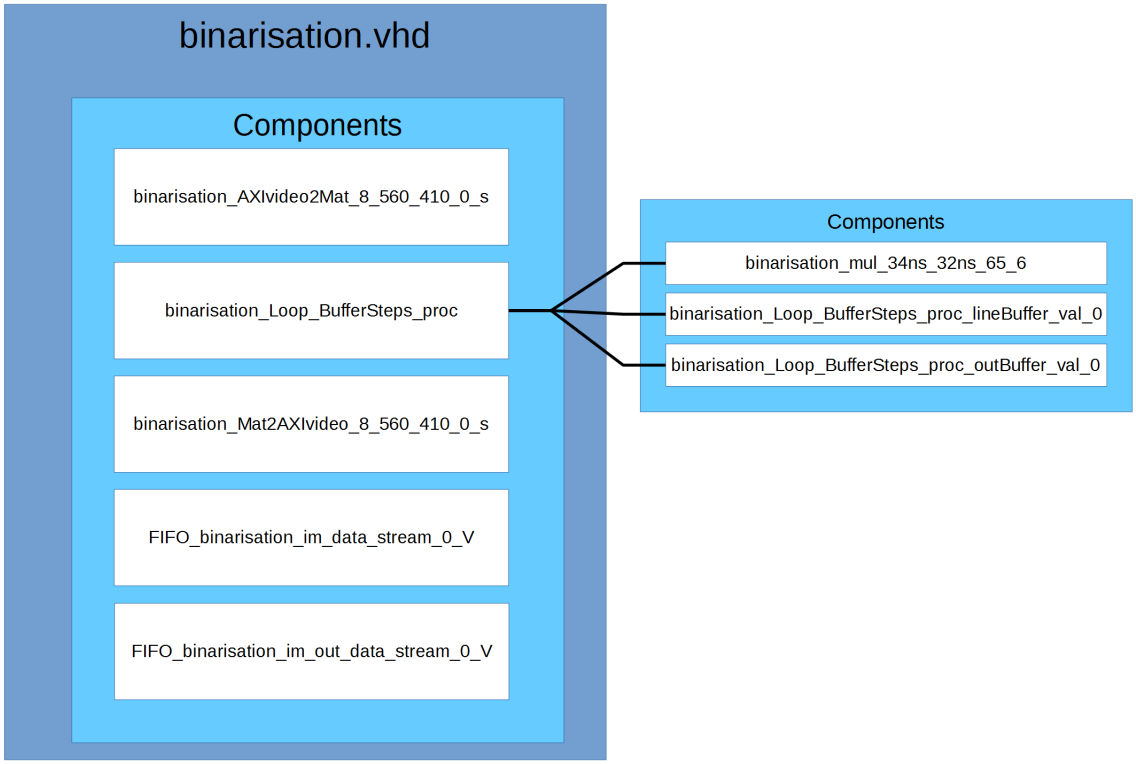


Abbildung 6.3: Das generierte FPGA-Modell mit fünf Komponenten, binarisation_Loop_BufferSteps_proc besitzt drei Unterkomponenten, die die Daten verarbeiten

Die Top-Level-Entität verbindet die Unterkomponenten mit Signalen, setzt Statussignale und definiert die Schnittstellenports:

```
entity binarisation is
port (
    INPUT_STREAM_TDATA : IN STD_LOGIC_VECTOR (7 downto 0);
    INPUT_STREAM_TKEEP : IN STD_LOGIC_VECTOR (0 downto 0);
    INPUT_STREAM_TSTRB : IN STD_LOGIC_VECTOR (0 downto 0);
    INPUT_STREAM_TUSER : IN STD_LOGIC_VECTOR (0 downto 0);
    INPUT_STREAM_TLAST : IN STD_LOGIC_VECTOR (0 downto 0);
    INPUT_STREAM_TID : IN STD_LOGIC_VECTOR (0 downto 0);
    INPUT_STREAM_TDEST : IN STD_LOGIC_VECTOR (0 downto 0);
    OUTPUT_STREAM_TDATA : OUT STD_LOGIC_VECTOR (7 downto 0);
    OUTPUT_STREAM_TKEEP : OUT STD_LOGIC_VECTOR (0 downto 0);
    OUTPUT_STREAM_TSTRB : OUT STD_LOGIC_VECTOR (0 downto 0);
    OUTPUT_STREAM_TUSER : OUT STD_LOGIC_VECTOR (0 downto 0);
    OUTPUT_STREAM_TLAST : OUT STD_LOGIC_VECTOR (0 downto 0);
    OUTPUT_STREAM_TID : OUT STD_LOGIC_VECTOR (0 downto 0);
    OUTPUT_STREAM_TDEST : OUT STD_LOGIC_VECTOR (0 downto 0);
    ap_clk : IN STD_LOGIC;
    ap_rst_n : IN STD_LOGIC;
    INPUT_STREAM_TVALID : IN STD_LOGIC;
    INPUT_STREAM_TREADY : OUT STD_LOGIC;
    OUTPUT_STREAM_TVALID : OUT STD_LOGIC;
    OUTPUT_STREAM_TREADY : IN STD_LOGIC;
    ap_done : OUT STD_LOGIC;
    ap_start : IN STD_LOGIC;
    ap_idle : OUT STD_LOGIC;
    ap_ready : OUT STD_LOGIC );
end;
```

Diese Ports entsprechen den definierten Schnittstellen aus Kapitel 4. Die Input- und Output-AXI-Streams sind die Port-Level-Protokolle für die Datenübertragung und die Clock. Der Reset-Eingang und die Handshaking-Signale gehören zu dem Block-Level-Protokoll *ap_ctrl_hs*. Wie es im C++-Code der HLS definiert wurde, sind die Datenkanäle der AXI-Streams 8 Bit breit. Es werden 67 interne Signale definiert, die über die Port-Maps mit den fünf Unterkomponenten verbunden werden. Der Input-AXI-Stream wird der Komponente *binarisation_AXIvideo2Mat_8_560_410_0_s* zugewiesen:

```
entity binarisation_AXIvideo2Mat_8_560_410_0_s is
port (
    ap_clk : IN STD_LOGIC;
    ap_rst : IN STD_LOGIC;
    ap_start : IN STD_LOGIC;
    ap_done : OUT STD_LOGIC;
    ap_continue : IN STD_LOGIC;
    ap_idle : OUT STD_LOGIC;
    ap_ready : OUT STD_LOGIC;
    INPUT_STREAM_TDATA : IN STD_LOGIC_VECTOR (7 downto 0);
    INPUT_STREAM_TVALID : IN STD_LOGIC;
    INPUT_STREAM_TREADY : OUT STD_LOGIC;
    INPUT_STREAM_TKEEP : IN STD_LOGIC_VECTOR (0 downto 0);
```

```
    INPUT_STREAM_TSTRB : IN STD_LOGIC_VECTOR (0 downto 0);
    INPUT_STREAM_TUSER : IN STD_LOGIC_VECTOR (0 downto 0);
    INPUT_STREAM_TLAST : IN STD_LOGIC_VECTOR (0 downto 0);
    INPUT_STREAM_TID : IN STD_LOGIC_VECTOR (0 downto 0);
    INPUT_STREAM_TDEST : IN STD_LOGIC_VECTOR (0 downto 0);
    img_data_stream_V_din : OUT STD_LOGIC_VECTOR (7 downto 0);
    img_data_stream_V_full_n : IN STD_LOGIC;
    img_data_stream_V_write : OUT STD_LOGIC );
end;
```

Die Bezeichnung der Komponente lässt Rückschlüsse auf die Funktionalität zu: sie wandelt den Stream über eine Finite-State-Machine (FSM) für die Verarbeitung als 8-Bit-Bild mit einer Auflösung von $560 \cdot 410$ um. Über das 8 Bit breite Out-Signal *img_data_stream_V_din* werden die Daten über *binarisation_AXIvideo2Mat_8_560_410_0_U0_img_data_stream_V_din* an die Komponente *FIFO_binarisation_im_data_stream_0_V* mit dem Signal *im_data_stream_0_V_din* weiter gereicht. Die FIFO(First In First Out)-Komponenten *FIFO_binarisation_im_data_stream_0_V* und *FIFO_binarisation_im_out_data_stream_0_V* realisieren FIFO-Buffer über ein Shiftregister, das die Daten von *if_din* erhält und von der jeweiligen FIFO-Komponente gesteuert wird. Die Daten verlassen die FIFO-Komponenten über *if_dout*. Die Daten werden von der Komponente *binarisation_Loop_BufferSteps_proc* entsprechend der Verarbeitungs-Schleife der HLS-Implementierung verarbeitet. Es wird eine FSM gebildet, die den Binarisierungs/Segmentierungs-Algorithmus abbildet. Die Komponente *binarisation_Loop_BufferSteps_proc_lineBuffer_val_0* modelliert den Linebuffer mit Block-RAM-Modulen:

```
generic(
        mem_type    : string := "block";
        dwidth      : integer := 8;
        awidth      : integer := 13;
        mem_size    : integer := 4100
);
```

Die Datenbreite ist 8 Bit, die Adressweite 13 Bit und die Speicherkapazität beträgt 4100 Byte. Letztere ergibt sich aus der Dimension des Linebuffers: 10 Zeilen mit einer Breite von 410 Pixeln à 8 Bit ergeben 4100 Byte. Die Komponente *binarisation_Loop_BufferSteps_proc* hat einen 8-Bit breiten Datenkanal, der mit der Komponente *binarisation_Mat2AXIvideo_8_560_410_0_s* verbunden ist. Diese Komponente bereitet über eine FSM die Daten für den Output-AXI-Stream vor:

```
port (
    ap_clk : IN STD_LOGIC;
    ap_rst : IN STD_LOGIC;
    ap_start : IN STD_LOGIC;
    ap_done : OUT STD_LOGIC;
    ap_continue : IN STD_LOGIC;
    ap_idle : OUT STD_LOGIC;
    ap_ready : OUT STD_LOGIC;
```

```
    img_data_stream_V_dout : IN STD_LOGIC_VECTOR (7 downto 0);
    img_data_stream_V_empty_n : IN STD_LOGIC;
    img_data_stream_V_read : OUT STD_LOGIC;
    OUTPUT_STREAM_TDATA : OUT STD_LOGIC_VECTOR (7 downto 0);
    OUTPUT_STREAM_TVALID : OUT STD_LOGIC;
    OUTPUT_STREAM_TREADY : IN STD_LOGIC;
    OUTPUT_STREAM_TKEEP : OUT STD_LOGIC_VECTOR (0 downto 0);
    OUTPUT_STREAM_TSTRB : OUT STD_LOGIC_VECTOR (0 downto 0);
    OUTPUT_STREAM_TUSER : OUT STD_LOGIC_VECTOR (0 downto 0);
    OUTPUT_STREAM_TLAST : OUT STD_LOGIC_VECTOR (0 downto 0);
    OUTPUT_STREAM_TID : OUT STD_LOGIC_VECTOR (0 downto 0);
    OUTPUT_STREAM_TDEST : OUT STD_LOGIC_VECTOR (0 downto 0) );
end;
```

Die vom HLS-Compiler generierten komplexen FSMs modellieren einen Steuerpfad und einen Datenpfad. Der Steuerpfad integriert ein Überführungsschaltnetz und ein Ausgabeschaltnetz, welches die kombinatorische Logik steuert. Die FSM aus der Komponente *binarisation_Loop_BufferSteps_proc* verwendet 60 Konstanten und 154 Signale bzw. Register.

6.2 Ergebnis der Systemintegration des HLS-IP-Cores

Das in Kapitel 5 vorgestellte System wurde mit Vivado synthetisiert und implementiert. Die Synthese ist das Übertragen einer über VHDL oder Verilog implementierten FPGA-Funktionalität auf einen konkreten FPGA-Chip. Die Synthese hat durch eine chipspezifische Bibliothek Kenntnis über die verfügbaren FPGA-Ressourcen und setzt VHDL-Anweisungen in Ressourcen (LUTs, Flip-Flops, MACs etc.) des FPGAs um. Der Implementations-Vorgang erzeugt aus der Synthese mit *Place & Route* die konkrete Zuweisung der benötigten Ressourcen zu den auf dem FPGA-Chip befindlichen Ressourcen (*Place*). Die Ressourcen werden miteinander und mit externen Pins verbunden, sodass Constraints für das Timing und die Pin-Zuweisungen erreicht werden (*Route*). Mit den Informationen aus der Implementation wird eine Bitstream-Datei generiert, die zur Programmierung des FPGAs auf den Chip übertragen wird.

6.2.1 Syntheseergebnis

Die folgenden Werte sind dem *Utilization Report* entnommen, der den Ressourcenbedarf für einzelne Komponenten des synthetisierten Systems aufschlüsselt.

Ressource	BRAM_18K	DSP	FF	LUT
Verfügbar auf Zynq	280	220	106400	53200
Genutzt	4	4	464	482
Bedarf	~1%	~2%	<1%	~1%

Tabelle 6.5: Ressourcenbedarf nach Synthese des Binarisierungs- und Segmentierungsmodul auf dem Zynq-FPGA

Die vier BRAM_18K-Module werden zu zwei BRAM_36K zusammen gefasst und modellieren den Linebuffer in der VHDL-Komponente *binarisation_Loop_BufferSteps_proc_lineBuffer_val_0_ram*. Die Angaben des Utilization-Reports decken sich mit den Abschätzungen aus der High-Level-Synthese. Die größte Abweichung liegt beim LUT-Bedarf, der in der HLS 33% zu hoch geschätzt wurde (HLS: 724 LUTs, Synthese: 482 LUTs). Beim BRAM und den DSPs ist der reale Bedarf exakt wie vom HLS-Report vorausgesagt. Beim Flip-Flop-Bedarf ist die Abweichung gering (vgl. Tabelle 6.5).

Ressource	BRAM_18K	DSP	FF	LUT
Verfügbar auf Zynq	280	220	106400	53200
Genutzt	9	4	5745	5662
Bedarf	~5%	~2%	~5%	~11%

Tabelle 6.6: Ressourcenbedarf nach Synthese des Gesamtsystems auf dem Zynq-FPGA

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	Block RAM Tile (140)	DSPs (220)
design_1_wrapper	5662	5745	9	4.5	4
design_1_i (design_1)	5662	5745	9	4.5	4
axi_interconnect_0 (design_...	1709	1965	0	0	0
axi_vdma_0 (design_1_axi_...	2847	2657	9	2.5	0
binarisation_0 (design_1_bin...	482	464	0	2	4
intrconcat (design_1_xlconc...	0	0	0	0	0
processing_system7_0 (desi...	112	0	0	0	0
processing_system7_0_axi_...	493	617	0	0	0
rst_processing_system7_0_...	19	42	0	0	0
xlconstant_0 (design_1_xlco...	0	0	0	0	0

Abbildung 6.4: *Utilization Report* der FPGA-Synthese des Gesamtsystem, Aufschlüsselung nach einzelnen FPGA-Modulen

Der größte Zuwachs des Ressourcenbedarfs geht auf das AXI-VDMA-Modul zurück (vgl. Tabelle 6.6). Dieses fordert fünf weitere BRAM_18K-Module und je knapp 3000 LUTs und Flip-Flops. Der AXI-Interconnect erfordert je knapp 2000 LUTs und Flip-Flops (vgl. Abbildung 6.4).

Das Timing-Constraint der Clock, die mit 100 MHz die Taktung für die FPGA-Module vorgibt, wird vom synthetisierten System erfüllt.

6.2.2 Implementierungsergebnis

Unterschiede im Ressourcenbedarf sind nach *Place & Route* in geringem Maße vorhanden (vgl. Tabelle 6.7 und 6.8).

Ressource	BRAM_18K	DSP	FF	LUT
Verfügbar auf Zynq	280	220	106400	53200
Genutzt	4	4	457	406
Bedarf	~1%	~2%	<1%	~1%

Tabelle 6.7: Ressourcenbedarf der Implementierung des Binarisierungs- und Segmentierungsmodul auf dem Zynq-FPGA

Ressource	BRAM_18K	DSP	FF	LUT
Verfügbar auf Zynq	280	220	106400	53200
Genutzt	9	4	5038	4200
Bedarf	~5%	~2%	~5%	~8%

Tabelle 6.8: Ressourcenbedarf der Implementierung des Gesamtsystems auf dem Zynq-FPGA

Die Implementierung hat einen geringeren Ressourcenbedarf, da für *Place & Route* Optimierungsschritte vorgenommen werden. Die Optimierungsstrategie kann in den Implementierungseinstellungen angepasst werden, sodass eine Balance zwischen Verarbeitungszeit und Optimierungsgrad erreicht wird, die sich nach dem zu implementierenden System richtet [Xil15e, S. 124]. Für diese Implementierung wurde die Strategie auf *Vivado Implementation Defaults* belassen, die auf die Einhaltung der Timingconstraints ausgerichtet ist. Der Bedarf an LUTs gegenüber der Synthese ist um ~25% gesunken, bei dem Bedarf der anderen FPGA-Elemente ändert sich wenig.

6.3 Stand des HLS-Projekts und Ausblick

Im Rahmen dieser Arbeit wurden folgende Arbeitspakete umgesetzt:

- Entwicklung einer Algorithmuskette für Minutienextraktion als C++-Referenzimplementierung
- Anpassung der Referenzimplementierung der Binarisierung/Segmentierung für eine High-Level-Synthese
- Durchführung der High-Level-Synthese
- Verifikation der High-Level-Synthese
- Systemintegration des High-Level-Syntheseergebnis
- Analyse der HLS und der Systemintegration
- FPGA-Debugging

Die Systemintegration des HLS-IP-Cores ist nicht abgeschlossen, sodass ein Error-Interrupt bei einer VDMA-Transaktion ausgelöst wird. Das FPGA-Debugging über einen ILA-Core mit Cross-Trigger zeigt dieses Verhalten auf Signalebene (Abbildung 6.5).

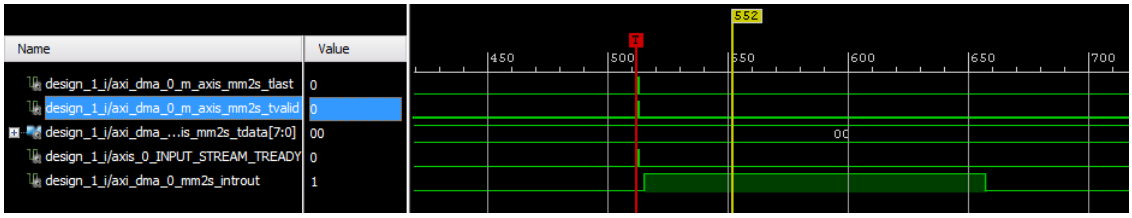


Abbildung 6.5: Verhalten ausgewählter Protokollsignale des INPUT-AXI-Streams am HLS-Core: Der Interrupt (axi_dma_0_mm2s_introut) wird direkt zu Beginn einer Transaktion auf active high gesetzt

Bei der Initialisierung einer AXI-Stream-Übertragung wird direkt der Interrupt ausgelöst und in der ARM-Software der Error-Interrupthandler aufgerufen. Der Fehler kann in der definierten 8 Bit-Datenbreite von TDATA des AXI-Streams liegen. Diese ist keine Standardbreite und liegt innerhalb der 32 Bit des RAMs. Das könnte Alignment-Fehler auslösen, die bei der Systemintegration nicht berücksichtigt wurden.

Erweiterungen für das Fingerabdruckscanner-Projekt umfassen folgende Arbeitspakete:

- Die High-Level-Synthese und die Anpassungen für die restlichen Bildverarbeitungsschritte durchführen: Optimierung des binarisierten Bildes, Skelettierung, Minutienextraktion, Minutienreduktion
- Die Systemintegration der Bildverarbeitungskette für die Minutienextraktion auf dem FPGA-ARM-SoC vervollständigen
- Die Arbeitspakete vorheriger Arbeiten zu einem verzahnten FPGA-System zusammenführen:
  - Anbindung des Fingerabdruckscanners
  - Bildvorverarbeitung nach Weber [Web14]
  - Bildvorverarbeitung nach Blauhut [Bla16]
  - Minutienextraktion dieser Arbeit
- Die Verbesserung und Erweiterung der Minutienextraktion:
  - Zuverlässige Minutienreduktion
  - Standardkonforme Speicherung der Minutien
  - Minutien-Vergleichsalgorithmus zur Verifikation einer Person

Die Realisierung dieser Punkte führt zu einem vollständigen Fingerabdruckverifikationssystem, das auf einem embedded System ohne angebundene Workstation lauffähig ist.

7 Zusammenfassung

Die Referenzimplementierung für eine FPGA-basierte Fingerabdruck-Minutienextraktion wurde als Algorithmuskette in C++ mit der Bildverarbeitungsbibliothek OpenCV realisiert. Diese Minutienextraktion erkennt die Grundmerkmale *Ridge-Ending* und *Bifurcation* in einem Grauwertbild, auch wenn dieses schwach belichtet ist und die Fingerabdruckaufnahme Fehler aufweist. Die Binarisierung wurde mit einem Local-Thresholding-Verfahren implementiert, auf dessen Basis die Segmentierung mit einem entwickelten Algorithmus umgesetzt wurde. Für die Optimierung der binarisierten Ridges und Valleys zur Eliminierung von Lücken wurden morphologische Operationen angewendet, die ebenfalls bei der Skelettierung mit Thinning-Verfahren zum Einsatz kommen. Die Minutienidentifikation auf Basis des skelettierten Bildes wurde mit dem Crossing-Numbers-Algorithmus durchgeführt.

Eine High-Level-Synthese wurde auf Basis der Referenzimplementierung der Binarisierung durchgeführt. Hierfür wurden Codeanpassungen vorgenommen, um den generierten IP-Core mit einer AXI-Stream-Schnittstelle in ein FPGA-SoC zu integrieren. Die Codeanpassungen umfassen das Substituieren der OpenCV-Bibliothek mit einer kompatiblen HLS-Bildverarbeitungsbibliothek, die Einführung eines Linebuffers, die Definition der Block- und Port-Level-Schnittstellen und die Einführung der DATAFLOW-Direktive. Das Ergebnis der High-Level-Synthese wurde mit einer entwickelten C-Testbench und der C/RTL-Co-Simulationsfunktion verifiziert.

Die FPGA-Systemintegration des IP-Cores der High-Level-Synthese wurde mit einem VDMA-Core umgesetzt. Dieser IP-Core ist mit AXI-Interconnect-Komponenten über die AXI4 und AXI4-Lite-Schnitstelle mit dem Zynq-SoC für den Zugriff auf das externe RAM verbunden. Der VDMA-Core wandelt die adressbasierten Daten des AXI-Protokolls in das adresslose AXI-Stream-Protokoll um und reicht die Bilddaten damit an den Binarisierungs-IP-Core weiter.

Eine C-Applikation, die Bare-Metal auf einem der beiden ARM-A9-Prozessoren des Zynq-SoCs läuft, initialisiert und steuert das FPGA-SoC. Hierfür werden die Read- und Write-Channel sowie die Interrupts der VDMA-Komponente initialisiert. Ein Board-Support-Package bietet für diese Aufgaben Treiberfunktionen, die die Konfigurationsregister der VDMA-Komponente setzen.

Die Analyse der Synthese des implementierten FPGA-Systems zeigt, dass maximal 8% der Logikgatter verwendet werden, wovon ein Großteil auf die VDMA-Komponente zurückzuführen ist. Der Ressourcenbedarf der Bildverarbeitungskomponente von 1% zeigt, dass für zahlreiche Bildverarbeitungskomponenten ausreichend Ressourcen auf dem SoC vorhanden sind. Ein Funktionsnachweis des Gesamtsystems mit bearbeiteten Bildern steht aus, da die Konfiguration nicht vollständig ist.

Literatur

[Bit16] Bitcom. *Interesse an biometrischen Verfahren wächst*. 2016. URL: https://www.bitkom.org/Presse/Presseinformation/Interesse-an-biometrischen-Verfahren-waechst.html (besucht am 09. 08. 2016).

[Küh15] Eike Kühl. *Mehr Faces für Facebook*. 2015. URL: http://www.zeit.de/digital/datenschutz/2015-06/facebook-moments-gesichterkennung (besucht am 09. 08. 2016).

[Ker06] Christian Kern. *Anwendung von RFID-Systemen* -. 2. Aufl. Berlin Heidelberg: Springer Science und Business Media, 2006. ISBN: 978-3-540-44477-0.

[BSI04] BSI. *Evaluierung biometrischer Systeme Fingerabdrucktechnologien – BioFinger. Öffentlicher Abschlussbericht*. 2004. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/BioFinger/BioFinger_I_I_pdf.pdf;jsessionid=A89BB95FB011BEE1809C991CA3E6BD5C.2_cid359?__blob=publicationFile&v=1.

[Der13] Dermalog. *DERMALOG LF10 Fingerprint Scanner*. 2013. URL: http://www.dermalog.com/en/products_solutions/fingerprintscanner/lf10.php (besucht am 09. 08. 2016).

[Xil13a] Xilinx. *Zynq-7000 All Programmable SoC*. 2013. URL: http://www.xilinx.com/content/xilinx/en/products/silicon-devices/soc/zynq-7000.html (besucht am 20. 10. 2015).

[Avn14] Inc. Avnet. *ZedBoard*. 2014. URL: http://www.zedboard.org/product/zedboard (besucht am 13. 01. 2014).

[Web14] Rolf Weber. "MPSoC-basierte Bildvorverarbeitung für ein biometrisches Identifikationssystem". Magisterarb. HAW Hamburg, 2014.

[Bla16] Dennis Blauhut. "MPSoC-Echtzeitbildverarbeitung mit wahlfreiem Speicherzugriff über einen nicht blockierenden Cache für ein biometrisches Identifikationssystem". Magisterarb. HAW Hamburg, 2016.

[MJ09] Dario Maio und Anil K. Jain. *Handbook of fingerprint recognition*. Berlin, Heidelberg: Springer, 2009. ISBN: 978-1-848-82254-2.

[MM97] D. Maio und D. Maltoni. "Direct Gray-Scale Fingerprint Minutiae Detection In Fingerprints". In: *IEEE Transactions On Pattern Analysis And Machine Intelligence*. Bd. 19. 1997, S. 27–40.

[WP93] L. Wang und T. Pavlidis. “Direct Gray-Scale Extraction Of Features For Character Recognition”. In: *IEEE Transactions On Pattern Analysis And Machine Intelligence*. Bd. 15. 1993, S. 1053–1067.

[ANS11] ANSI. *Data Format for the Interchange of Fingerprint, Facial and Other Biometric Information*. ISO ANSI/NIST-ITL 1-2011 Update:2013. American National Standards Institute, 2011.

[ISO11] ISO. *Information technology – Biometric data interchange formats – Part 2: Finger minutiae data*. ISO ISO/IEC 19794-2:2011. International Organization for Standardization, 2011.

[Kem15a] Raphael Kemmler. “Entwurf einer Algorithmuskette für eine Fingerabdruckmerkmalsextraktion”. Master Projekt 1. 2015.

[Ike+02] N. Ikeda u. a. “Fingerprint image enhancement by pixel-parallel processing”. In: *Pattern Recognition, 2002. Proceedings. 16th International Conference on*. Bd. 3. 2002, 752–755 vol.3. DOI: 10.1109/ICPR.2002.1048099.

[GH89] Zicheng Guo und Richard W. Hall. “Parallel Thinning with Two-subiteration Algorithms”. In: *Commun. ACM* 32.3 (März 1989), S. 359–373. ISSN: 0001-0782. DOI: 10.1145/62065.62074. URL: http://doi.acm.org/10.1145/62065.62074.

[Xil13b] Xilinx. *Simple AMP Running Linux and Bare-Metal System on Both Zynq SoC Processors. Application Note: Zynq-7000 AP SoC*. XAPP 1078. 2013. URL: http://www.xilinx.com/support/documentation/application_notes/xapp1078-amp-linux-bare-metal.pdf.

[Xil16a] Xilinx. *Zynq-7000 All Programmable SoC Overview. Product Specification*. DS 190. 2016. URL: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[MS09] G. Martin und G. Smith. “High-Level Synthesis: Past, Present, and Future”. In: *Design Test of Computers, IEEE* 26.4 (Juli 2009), S. 18–25. ISSN: 0740-7475. DOI: 10.1109/MDT.2009.83.

[Xil13c] Xilinx. *Vivado Design Suite*. 2013. URL: http://www.xilinx.com/products/design-tools/vivado/ (besucht am 08. 01. 2015).

[Atm14] Atmel. *Altera SDK for OpenCL*. 2014. URL: http://www.altera.com/products/software/opencl/opencl-index.html (besucht am 19. 12. 2014).

[IBM14] IBM. *Liquid Metal*. 2014. URL: http://researcher.watson.ibm.com/researcher/view_group.php?id=122 (besucht am 19. 12. 2014).

[Est14] Esterel. *Esterel*. 2014. URL: http://www.esterel.org/ (besucht am 19. 12. 2014).

[SG08] S. Singh und D. Greaves. "Kiwi: Synthesis of FPGA Circuits from Parallel Programs". In: *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*. Apr. 2008, S. 3–12. DOI: 10.1109/FCCM.2008.46.

[Uni14] Berkeley University of California. *Chisel*. 2014. URL: https://chisel.eecs.berkeley.edu/ (besucht am 19. 12. 2014).

[Xil14a] Xilinx. *Vivado Design Suite User Guide. High-Level Synthesis*. UG 902. 2014. URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf.

[Ope14] Opencv.org. *OpenCV*. 2014. URL: http://opencv.org/ (besucht am 19. 09. 2014).

[Arm10] Arm. *AMBA Specifications. Facilitate right-first-time development of multi-processor designs*. 2010. URL: http://www.arm.com/products/system-ip/amba-specifications (besucht am 09. 08. 2016).

[Xil11] Xilinx. *AXI Reference Guide. User Guide*. UG 761. 2011. URL: http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

[Xil13d] Xilinx. *AXI Interconnect v2.0. LogiCORE IP Product Guide*. PG 059. 2013. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_0/pg059-axi-interconnect.pdf.

[Xil16b] Xilinx. *AXI4-Stream Infrastructure IP Suite. LogiCORE IP Product Guide*. PG 085. 2016. URL: http://www.xilinx.com/support/documentation/ip_documentation/axis_infrastructure_ip_suite/v1_1/pg085-axi4stream-infrastructure.pdf.

[Xil15a] Xilinx. *AXI Video Direct Memory Access v6.2. LogiCORE IP Product Guide*. PG 020. 2015. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v6_2/pg020_axi_vdma.pdf.

[Xil13e] Xilinx. *LogiCORE IP Processing System 7 v5.3. LogiCORE IP Product Guide*. PG 082. 2013. URL: http://www.xilinx.com/support/documentation/ip_documentation/processing_system7/v5_3/pg082-processing-system7.pdf.

[Xil15b] Xilinx. *Processor System Reset Module v5.0. LogiCORE IP Product Guide*. PG 164. 2015. URL: http://www.xilinx.com/support/documentation/ip_documentation/proc_sys_reset/v5_0/pg164-proc-sys-reset.pdf.

[Bol14] University of Bologna. *Fingerprint Generation*. 2014. URL: http://biolab.csr.unibo.it/research.asp?organize=Activities&select=&selObj=12&pathSubj=111%7C%7C12& (besucht am 19. 09. 2014).

[Nis+12] Alfred Nischwitz u. a. *Computergrafik und Bildverarbeitung - Band II: Bildverarbeitung*. Berlin Heidelberg New York: Springer-Verlag, 2012. ISBN: 978-3-834-88300-1.

[ZS84] T. Y. Zhang und C. Y. Suen. "A Fast Parallel Algorithm for Thinning Digital Patterns". In: *Commun. ACM* 27.3 (März 1984), S. 236–239. ISSN: 0001-0782. DOI: 10.1145/357994.358023. URL: http://doi.acm.org/10.1145/357994.358023.

[CEE14] Louise H. Crockett, Ross a. Elliot und Martin a. Enderwitz. *The Zynq Book - Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014. ISBN: 978-0-992-97870-9.

[Xil15c] Xilinx. *Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries. Application Note: Vivado HLS*. XAPP 1167. 2015. URL: http://www.xilinx.com/support/documentation/application_notes/xapp1167.pdf.

[Kem15b] Raphael Kemmler. "High-Level-Synthese eines OpenCV-basierten Binarisierungs- und Segmentierungsalgorithmus". Master Projekt 2. 2015.

[Xil15d] Xilinx. *UltraFast Design Methodology Guide for the Vivado Design Suite*. UG 949. 2015. URL: http://www.xilinx.com/support/documentation/sw_manuals/ug949-vivado-design-methodology.pdf.

[Xil14b] Xilinx. *Integrated Logic Analyzer v5.0. LogiCORE IP Product Guide*. PG 172. 2014. URL: http://www.xilinx.com/support/documentation/ip_documentation/ila/v5_0/pg172-ila.pdf.

[Xil13f] Xilinx. *Vivado HLS Design Flow Lab. Lab Workbook*. 2013.

[Xil14c] Xilinx. *Virtex-6 FPGA Memory Resources. User Guide*. UG 363. 2014. URL: http://www.xilinx.com/support/documentation/user_guides/ug363.pdf.

[Xil16c] Xilinx. *7 Series DSP48E1 Slice. User Guide*. UG 479. 2016. URL: http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf.

[Xil15e] Xilinx. *Vivado Design Suite User Guide. Implementation*. UG 904. 2015. URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_3/ug904-vivado-implementation.pdf.

Glossar

8-Neighborhood (dt. 8-Nachbarschaft) die 8 umgebenden Pixel eines Pixels. iv, 25

AXI Advanced eXtensible Interface; eine Protokollsammlung (AXI, AXI-Lite, AXI-Stream) zur On-Chip-Kommunikation von IP-Cores. iv, 4, 13–16, 34–37, 39, 40, 42, 53, 55, 56, 58, 60, 62

Bifurcation (dt. Gabelung) Teilung einer Ridge, Minutie eines Fingerabdrucks. iv, 6, 7, 27, 28, 62

Binarisierung Umwandlung eines Farb-/Graustufenbildes in ein Schwarz-Weiß-Bild mit einer Farbtiefe von 1 Bit. iv, 4, 8, 19–21, 33, 35, 40, 42, 46, 52, 56, 62

Clock Taktfrequenz, mit der ein FPGA-Komponente Befehle abarbeitet. iv, 15, 35, 44, 45, 54, 55, 58

High-Level-Synthese Verfahren mit einem speziellem Compiler zur automatischen RTL-Codegenerierung aus Hochsprachencode. iv, v, 3, 9, 12, 33

HLS High-Level-Synthese. iv, 3, 4, 9–13, 33–38, 40–42, 44, 51, 53, 55–58, 60, 62

IP-Core (Intellectual-Property-Core), wiederverwendbarer Funktionsblock eines Chipdesigns. iv–vi, 3, 9, 11, 13, 15–17, 35, 42, 49, 60, 62

Minutie Merkmal eines Fingerabdrucks, z.B. Bifurcation oder Ridge-Ending. iv, v, 4, 5, 8, 18, 19, 25, 27, 28, 30–32, 60, 62

Register-Transfer-Level Abstraktionsebene von FPGA-Funktionalität, realisiert mit Hardwarebeschreibungssprachen wie VHDL oder Verilog. iv, v, 2

Ridge (dt. Grat) ist die Erhebung der Haut einer Fingerkuppe. iv, v, 5, 6, 8, 18, 19, 21–25, 62

Ridge-Ending Endung einer Ridge, Minutie eines Fingerabdrucks. iv, 5–7, 27, 28, 30, 31, 62

RTL Register-Transfer-Level. iv, 2, 12–14, 36, 38, 40, 41, 53, 62

Segmentierung Objekttrennung vom Hintergrund in einem Bild. iv, 4, 19–21, 33, 37, 40, 56, 62

SoC System-On-Chip. iv, 2–5, 7, 9, 13, 16, 42, 44, 45, 60, 62

System-On-Chip Ein-Chip-System, das alle oder einen großen Teil der Funktionen eines programmierbaren elektronischen Systems auf einem Chip integriert. iv, vi, 2

Threshold (dt. Schwellenwert) Wert, der eine Grenze zur Verarbeitung von Daten ist. iv, 19, 20, 62

Tile (dt. Kachel) Abschnitte, in die ein Bild eingeteilt wird. iv, 8, 18–21

Valley (dt. Tal) Furche der Haut auf der Fingerkuppe. iv, 5, 6, 19, 21, 22, 62

VDMA Video-Direct-Memory-Access. iv, 16, 42, 44, 46–48, 58, 60, 62

Video-Direct-Memory-Access Ein IP-Core für die Umwandlung von adressbasierten AXI-Verbindungen zu adresslosen AXI-Stream-Verbindungen, die über den VDMA-Core beispielsweise auf den externen RAM zugreifen können. iv, vi, 16

Anhang

1 Ergänzende Grafiken

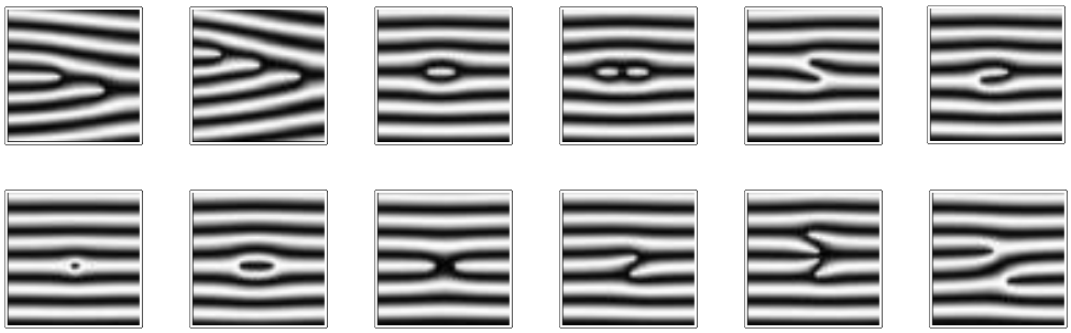


Abbildung 1: Zweifache Gabelung, dreifache Gabelung, einfacher Wirbel, zweifacher Wirbel, seitliche Berührung; Haken, Punkt, Intervall, X-Linie, einfache Brücke, zweifache Brücke, fortlaufende Linie [BSI04]

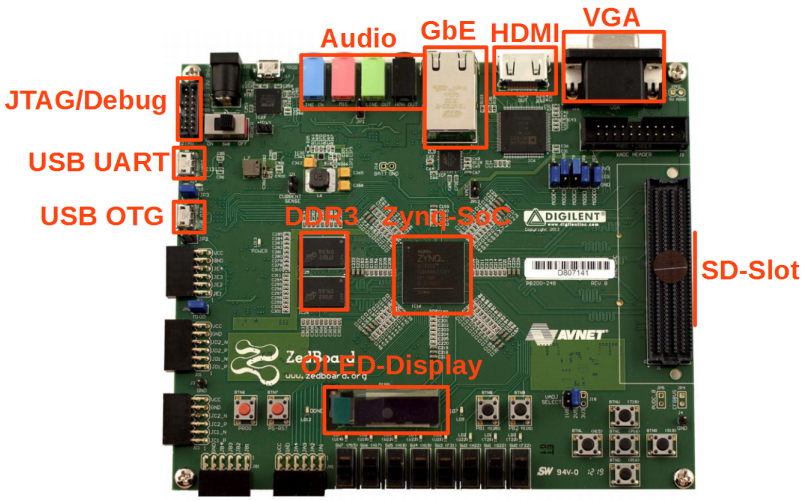


Abbildung 2: Schnittstellen und Komponenten des Zedboards

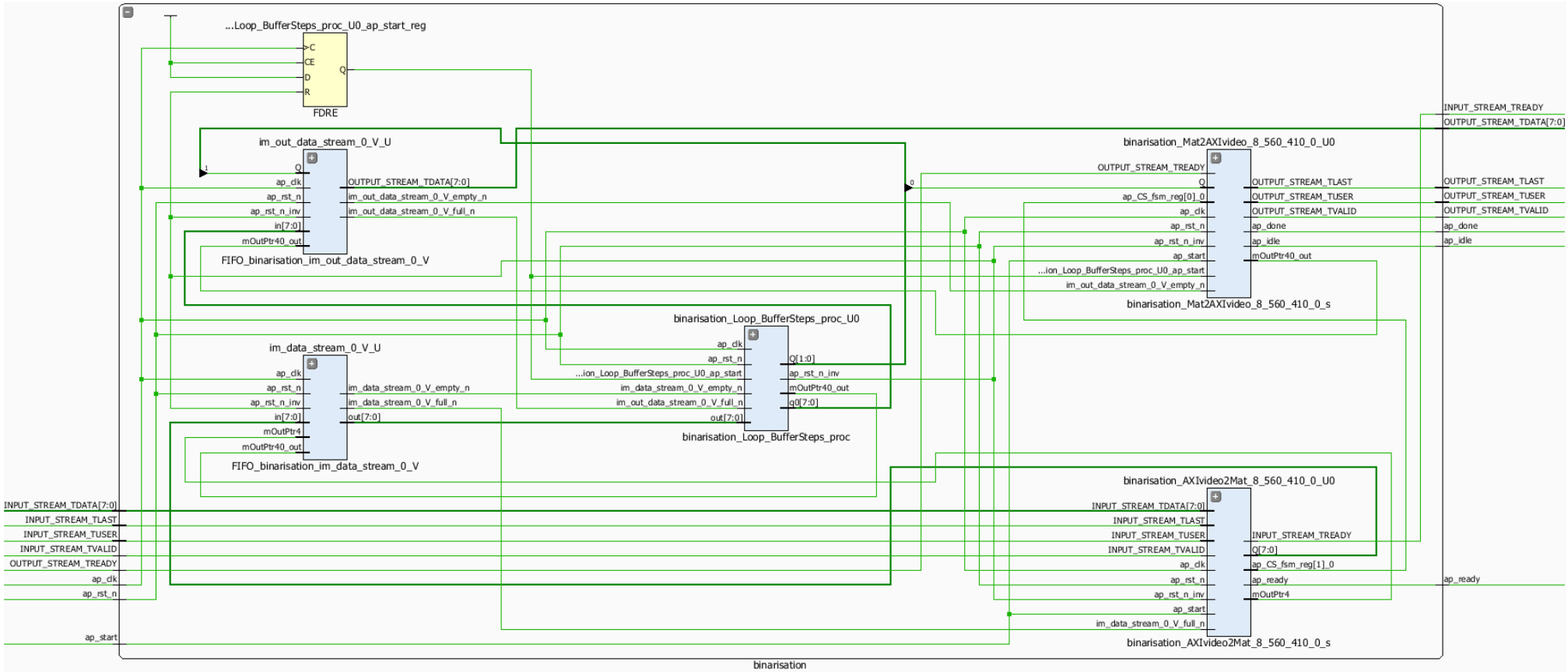


Abbildung 3: Darstellung des synthetisierten RTL-Modells aus der HLS in Vivado

2 Sourcecode der Binarisierung mit einem In-Out-Linebuffer

```
#include "AdaptiveLocalThreshold.h"

void binarisation(AXI_STREAM& INPUT_STREAM, AXI_STREAM& OUTPUT_STREAM){

  const int rows = 560;
  const int cols = 410;
  const int deltaThreshold = 100;
  const int numTilesWidth = 41;
  const int numTilesHeight = 56;
  const int tileWidth = 10; //im.cols/numTilesWidth
  const int tileHeight = 10;  //im.rows/numTilesHeight
  int i, j, k, l, t;
  const int numOfLinesBuffered = 560;
  const int numOfColsBuffered = 410;
  unsigned int greyValuesAdded = 0;
  unsigned int pixelPerTile = tileWidth * tileHeight;
  unsigned char intensity = 0;
  int numOfTiles = numTilesWidth*numTilesHeight;
  unsigned int thresholds [numOfTiles+numTilesWidth+numTilesHeight];
  unsigned char deltas [numOfTiles+numTilesWidth+numTilesHeight];
  unsigned int threshold = 0;
  unsigned char delta = 0;
  G_PIXEL inBuffer[numOfColsBuffered][numOfLinesBuffered];
  G_PIXEL outBuffer[numOfColsBuffered][numOfLinesBuffered];
  G_IMAGE im(rows, cols);
  G_IMAGE im_out(rows, cols);

  // Convert AXI4 Stream data to hls::mat format
  hls::AXIvideo2Mat(INPUT_STREAM, im);

  //fill the buffer numTilesHeight-times to process the whole picture
  for(int m = 0; m<numTilesHeight; m++){
    //fill buffer
    for(j = 0; j<numOfLinesBuffered; j++){
      for (i = 0; i<numOfColsBuffered; i++){
        lineBuffer[i][j] = im.read(); //read from AXI-Stream
      }
    }
    //iterate through the buffer, tile per tile
    for(i = 0; i<numOfColsBuffered; i=i+tileWidth){
      unsigned char darkest = 255;
      unsigned char brightest = 0;
      //iterate through tiles
      for(k = i; ((k-i)<tileWidth); k++){
        for(l = 0; l<tileHeight; l++){
          //sumate all greyvalues
```

```
47            intensity = lineBuffer[k][l].val[0];
48            greyValuesAdded += intensity;
49            //calculate brightest and darkest values of the tile
50            if(intensity < darkest){
51              darkest = intensity;
52            }else if(intensity>brightest){
53              brightest = intensity;
54            }
55          }
56        }
57        //save delta
58        delta = std::abs(brightest-darkest);
59        //calculate the mean value
60        threshold = greyValuesAdded/pixelPerTile;
61        greyValuesAdded = 0;
62        //segmentation
63        if(delta<deltaThreshold){
64          threshold = 0;
65        }
66        //binarization: apply theshold on tile
67        for(k = i; ((k-i)<tileWidth); k++){
68          for(l = 0; l<tileHeight; l++){
69            intensity = lineBuffer[k][l].val[0];
70            if(intensity >= threshold){
71              lineBuffer[k][l] = G_PIXEL(255); //white   G_PIXEL(255)
72            }else{
73              lineBuffer[k][l] = G_PIXEL(0); //black  G_PIXEL(0)
74            }
75          }
76        }
77      }//buffer iterated
78      //write out-buffer to AXI
79      for(j = 0; j<numOfLinesBuffered; j++){
80        for (i = 0; i<numOfColsBuffered; i++){
81          im_out.write(lineBuffer[i][j]); //write to AXI
82        }
83      }
84    }//whole picture processed
85
86    // Convert the hls::mat format to AXI4 Stream format
87    hls::Mat2AXIvideo(im_out, OUTPUT_STREAM);
88  }
```

3 Sourcecode der Testbench

```
1  #include <stdio.h>
2
3  #include "hls_video.h"
4  #include "hls_opencv.h"
5
```

```
6  typedef hls::stream<ap_axiu<8,1,1,1> > AXI_STREAM;
7
8  void binarisation(AXI_STREAM&, AXI_STREAM&);
9
10 int main (int argc, char** argv) {
11   printf("Start\n");
12   IplImage* src = cvLoadImage("Pfad\\zur\\Datei\\Input.bmp",
         CV_LOAD_IMAGE_GRAYSCALE);
13   if (!src){
14     printf("Image could not be loaded!\n");
15     return 1;
16   }
17   IplImage* dst = cvCreateImage(cvGetSize(src),src->depth,src->
         nChannels);
18   AXI_STREAM src_axi, dst_axi;
19   //convert OpenCV format to AXI4 Stream format
20   IplImage2AXIvideo(src, src_axi);
21   //call the function to be synthesized
22   binarisation(src_axi, dst_axi);
23   //convert the AXI4 Stream data to OpenCV format
24   AXIvideo2IplImage(dst_axi, dst);
25   //save image
26   cvSaveImage("Pfad\\zur\\Datei\\Result.bmp", dst);
27   //compare with golden sample
28   IplImage* goldenSample = cvLoadImage("Pfad\\zur\\Datei\\GoldenSample.
         bmp", CV_LOAD_IMAGE_GRAYSCALE);
29   int errorcnt = 0;
30   for(int i = 0; i<src->width; i++){
31     for(int j = 0; j<src->height; j++){
32       if(!(CV_IMAGE_ELEM(dst, unsigned char, j, i) == CV_IMAGE_ELEM(
             goldenSample, unsigned char, j, i))){
33         errorcnt++;
34         printf("[%i, %i] Expected value: %u | Received value: %u\n", i,
               j,CV_IMAGE_ELEM(goldenSample, unsigned char, j, i),
               CV_IMAGE_ELEM(dst, unsigned char, j, i));
35       }
36     }
37   }
38   //return testbench result
39   if(!errorcnt){
40     printf("Test Passed!\n");
41     return 0;
42   }else{
43     printf("Test Failed!\n");
44     return errorcnt;
45   }
46 }
```

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 20. Oktober 2016 Raphael Kemmler