TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Integrierte Systeme

Zum Einsatz dynamisch rekonfigurierbarer eingebetteter Systeme in der Bildverarbeitung

Dipl.-Ing. Christopher S. Claus

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. sc. Samarjit Chakraborty
Prüfer der Dissertation:	1. apl. Prof. Dr.-Ing. habil. Walter Stechele 2. Univ.-Prof. Dr.-Ing. Jürgen Becker, Karlsruher Institut für Technologie (KIT)

Die Dissertation wurde am 15.11.2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 01.02.2011 angenommen.

„Es sind nicht die stärksten Spezies, die überleben, nicht die intelligentesten, sondern die, die sich am schnellsten an Veränderungen anpassen.“

(aus „On the Origin of Species“ von Charles Darwin, 1859)

Vorwort

Die in dieser Dissertation dokumentierte Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Integrierte Systeme (LIS) der Technischen Universität München (TUM). Mein ganz besonderer Dank gebührt meinem Doktorvater Prof. Dr.-Ing. Walter Stechele für die Betreuung der Arbeit, die vielen fruchtbaren Gespräche, die ständige Unterstützung und seinen nie endenden Enthusiasmus. Des weiteren danke ich vielmals Prof. Dr. sc. techn. A. Herkersdorf, Ordinarius des LIS, für die Möglichkeit, an diesem Lehrstuhl zu forschen, für die vielen wertvollen Anregungen und ein traumhaftes Arbeitsklima. Ein weiterer besonderer Dank gilt Prof. Dr.-Ing. Jürgen Becker, Leiter des Instituts für Technik der Informationsverarbeitung (ITIV) am Karlsruher Institut für Technologie (KIT), für das Interesse an dieser Arbeit, die ständige Unterstützung seit meinem Studium in Karlsruhe und die Übernahme des Korreferats.
Insbesondere möchte ich mich bei der Deutschen Forschungsgemeinschaft (DFG) für die vierjährige Förderung des AutoVision Projekts im Rahmen des Schwerpunktprogramms „Rekonfigurierbare Rechensysteme (SPP1148)“ bedanken.
Ich möchte auch meinen jetzigen und ehemaligen Kollegen am LIS für die Unterstützung, die anregenden Diskussionen und grandiose Arbeitsatmosphäre bedanken. Ohne die Bereitschaft meiner Kollegen ihr Wissen zu teilen, wäre die Arbeit in dieser Form nicht möglich gewesen. Ein weiteres spezielles Dankeschön gebührt meinen ehemaligen Kollegen Dr.-Ing. M. Hübner, Dr.-Ing. O. Sander und Dipl.-Ing. L. Braun vom ITIV in Karlsruhe für die Kooperationen, die fachlich wertvollen Diskussionen und Anregungen.
Der Firma Xilinx Inc., besonders P. Lysaght, P. Patel, N. Difiore, P. Zoratti und R. Stewart gebührt Dank für das Interesse an diesem Forschungsthema und dessen Unterstützung in Form von zahlreicher Enwicklungs-Soft- und Hardware. Allen hier namentlich nicht genannten Personen von Industriepartnern wie Bosch, BMW, Sensor to Image GmbH, IBM, Rohde & Schwarz, etc. möchte herzlich für die Unterstützung danken.
Ohne die Mithilfe von über 30 Studenten im AutoVision Projekt wären die in dieser Arbeit entstandenen Prototypen in dieser Form nicht möglich gewesen. Deshalb gebührt mein tiefer Dank auch allen Studenten des AutoVision Teams, die mit ihrer Leidenschaft, ihrem Einsatz und ihrem Durchhaltevermögen einen wesentlichen Beitrag zum Erfolg dieses Forschungsvorhabens beigesteuert haben.
Das wichtigste Dankeschön gebührt meiner Familie, allen voran möchte meiner Frau Susanne, meinen Eltern Michael und Andrea, meinem Bruder Matthias, meinen Großeltern und Schwiegereltern. Ihre moralische Unterstützung, ihr Rückhalt und ihre Geduld führten letztendlich zum Gelingen dieser Arbeit.
Letztendlich sei auch all denjenigen gedankt, die sich hier aus Platzgründen nicht wiederfinden, jedoch meine Promotionszeit in jedweder Form bereichert haben.

Abstract

The *AutoVision* architecture is a novel FPGA-based System-on-Chip architecture for future video-based driver assistance systems. This architecture uses hardware accelerators (HWAs) for video processing that are exchangeable during run-time of the system. According to the changing driving conditions (highway, urban environment, tunnel entrance, daytime/nighttime driving, different weather conditions etc.) different algorithms for the video processing have to be used. These different algorithms require different hardware accelerator engines, that are loaded into the AutoVision chip at run-time depending on the driving conditions. An investigation has been performed how to use dynamic partial reconfiguration (DPR) to load and operate the right hardware accelerator engines in time, while removing unused engines in order to save precious chip area. On the one hand this architecture offers potential for situation adaptive hardware in driver assistance systems and reduces the monetary costs by saving chip area or complete electrical control units (ECUs) on the other hand.

In future automotive systems, video-based driver assistance will improve safety. Video processing for driver assistance requires real time implementation of complex algorithms. Dedicated solutions such as ASICs and ASSPs can offer the required real time processing, but they do not offer the necessary flexibility. As the development costs for ASICs have risen in the past 5 years, alternatives have to be considered from an car manufacturer (OEM) point of view. In driver assistance systems design changes are quite frequent which rarely makes an ASIC/ASSP an appropriate solution. In addition, by using dedicated solutions (such as Mobileye's EyeQ2 chip) all possible hardware accelerators for all driving conditions have to be present on the chip as design changes are not possible after manufacturing.

In the AutoVision project suitable algorithms for driver assistance have been chosen and evaluated and prototypes in software (Matlab, C++) have been implemented. These prototypes have been used to profile the algorithm. This included the identification of performance hungry parts which are going to be implemented in hardware. Algorithms for video processing can be separated into high level application code and low level pixel operations. By utilizing a kind of building block based (modular) pixel processing chain, hardware accelerators for the pixel operations have been implemented. Thereby only the

operation on a single pixel and its neighborhood has to be implemented from scratch as the data-input-, the intermediate-data and the data-output path can be reused due to the modular AutoVision concept. By using defined interfaces between hardware and software the implementation of a hardware accelerator can be done in parallel to the implementation of the software on embedded CPUs (HW/SW Codesign).

In addition it is possible to exchange only a certain portion (partial) of the device during run-time (dynamic) depending on the current driving situation. The dynamic partial reconfiguration of FPGAs does not affect the static (unchanged) part of the design. In the AutoVision project to prototypes have been implemented which are used to prove that the lossless reconfiguration between two video frames (Inter Video Frame Reconfiguration) and multiple reconfigurations within one single video frame (Intra Video Frame Reconfiguration) is possible. Therefore, a major part of the project was dealing with the acceleration of the reconfiguration process. Beginning of 2010, the by far shortest reconfiguration times have been achieved in the AutoVision project .

Kurzfassung

Die *AutoVision* Architektur ist eine neuartige FPGA-basierte System-on-Chip Architektur für zukünftige video-basierte Fahrerassistenz-Systeme (FAS). Um Videodaten in Echtzeit zu verarbeiten werden Beschleuniger, so genannte Hardware-Acceleratoren (HWAs) verwendet, die zur Laufzeit austauschbar sind. Gemäß dem sich verändernden Fahrumfeld (Autobahn, Innenstadt, Tunneleinfahrt, Tagfahrten, Nachtfahrten, verschiedenste Wettersituationen etc.) ist es sinnvoll unterschiedliche Algorithmen für die Videoverarbeitung einzusetzen. Diese Algorithmen bedürfen verschiedener HWAs, die zur Laufzeit des Systems - abhängig von den sich verändernden Fahrzuständen - in den AutoVision-Chip geladen werden. In dieser Arbeit wurde untersucht, wie man die dynamisch partielle Rekonfiguration (DPR) dazu einsetzen kann, die richtigen HWAs rechtzeitig zu laden und anzusteuern. Gleichzeitig werden unbenutzte HWAs vom FPGA entfernt, um wertvolle Chipfläche einzusparen. Diese Architektur bietet auf der einen Seite also die Möglichkeit für situationsadaptive Fahrerassistenz und reduziert auf der anderen Seite die monetären Kosten durch Einsparung von Chipfläche oder ganzer Steuergeräte.

Video-basierte FAS werden in Zukunft die Sicherheit in Fahrzeugen erhöhen. Videoverarbeitung für Fahrerassistenz bedarf Echzeitverarbeitung der Bilddaten durch komplexe Algorithmen. Eine reine Softwarelösung auf standardmäßigen Automotive Prozessoren liefert nicht die gewünschte Echtzeitfähigkeit. Dedizierte Hardware (ASICs oder ASSPs) liefern zwar Echtzeitfähigkeit, jedoch nicht die notwendige Flexibilität. Da die Enwicklungskosten für ASICs in den vergangenen 5 Jahren stark gestiegen sind, ist man aus Sicht der Fahrzeughersteller (OEMs) gezwungen, nach anderen Lösungen zu suchen. In Fahrerassistenzsystemen gibt es häufig Designänderungen, was ASICs/ASSPs aufgrund der damit verbundenen Kosten selten zu einer passenden Lösung macht. Zudem müßten bei einer dedizierten Lösung (vgl. Mobileyes EyeQ2 ASIC) die HWAs (Algorithmen) für alle möglichen Fahrsituationen auf dem Chip vorgehalten werden, was, wenn überhaupt, nur mit sehr großem finanziellen Zusatzaufwand möglich ist.

Im AutoVision Projekt wurden mögliche Algorithmen für die Erkennung im Videobild analysiert, ausgewählt und Prototypen in Software (Matlab, C++) erstellt. Diese Prototypen wurden verwendet, um den Algorithmus zu profilen, was bedeutet die performanz-

lastigen Teile zu identifizieren, die später in Hardawre (HW) ausgelagert werden sollten. Algorithmen für die Videoverarbeitung können in Highlevel- und Pixellevel-Operationen unterteilt werden. Mit Hilfe einer baukastenartigen (modularen) Pixelverarbeitungskette wurden HWAs für die Pixellevel-Operationen implementiert. Dabei muß lediglich die Operation auf ein Pixel und seine Umgebung neu implementiert werden, da Eingangs-, Zwischenergebnis- und Ausgangsdatenpfade durch das modulare Konzept von AutoVision einfach übernommen werden können. Durch definierte Schnittstellen zwischen HW und Software (SW) kann parallel zur Implementierung des HWA die Portierung der SW (Highlevel Operationen) auf die eingebetteten CPUs erfolgen (HW/SW Codesign).

Zudem ist es möglich, einen Teil (partiell) des FPGAs zur Laufzeit (dynamisch) abhängig von der gegenwärtigen Fahrsituation mit den entsprechenden HWA zu konfigurieren. Bei der dynamisch partiellen Rekonfiguration von FPGAs ist sichergestellt, dass der statische (nicht veränderbare) Teil des Systems unabhängig von der Rekonfiguration weiterläuft. Den Abschluß des Projekts bildeten zwei lauffähige Demonstratoren, mit deren Hilfe die verlustfreie Rekonfiguration zwischen zwei aufeinanderfolgenden Videobildern (Inter Video Frame Rekonfiguration), sowie die mehrfache Rekonfiguration innerhalb eines Videobildes (Intra Video Frame Rekonfiguration) prototypisch gezeigt wird. Ein wesentlicher Teil dieser Arbeit bestand daher in der Erhöhung der Rekonfigurationsgeschwindigkeit. Anfang 2010 wurden im AutoVision Projekt die mit Abstand höchsten Rekonfigurationsgeschwindigkeiten weltweit erzielt.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Abkürzung	Bedeutung
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
ASIP	**A**pplication **S**pecific **I**nstructionset **P**rocessor
ASSP	**A**pplication **S**pecific **S**tandard **P**roduct
bpp	**b**its **p**er **p**ixel
BRAM	**B**lock **R**andom **A**ccess **M**emory
CAN	**C**ontroller **A**rea **N**etwork
CG	**C**lock **G**ating
CLB	**C**onfigurable **L**ogic **B**lock
CSE	**C**en**S**us **E**ngine
CTE	**C**on**T**rast **E**ngine
CPP	**C**onfiguration **P**acket **P**rocessor
CPU	**C**entral **P**rocessing **U**nit
CRC	**C**yclic **R**edundancy **C**heck
DCR	**D**evice **C**ontrol **R**egister
DDR	**D**ouble **D**ata **R**ate
DMA	**D**irect **M**emeory **A**ccess
DPR	**D**ynamic **P**artial **R**econfiguration
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DSP	**D**igitaler **S**ignal **P**rozessor
ECU	**E**lectronic **C**ontrol **U**nit
FAS	**F**ahrer-**A**ssistenz **S**ystem
FDRI	**F**rame **D**ata **R**egister for **I**nput
FDRO	**F**rame **D**ata **R**egister for **O**utput
FOM	**F**eature **O**utput **M**emory
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
fps	**f**rames **p**er **s**econd
FSM	**F**inite **S**tate **M**achine
GPU	**G**raphics **P**rocessing **U**nit
HWA	**H**ard**w**are **A**ccelerator *deutsch: Hardware Beschleuniger*
ICAP	**I**nternal **C**onfiguration **A**ccess **P**ort
ILM	**I**ntermediate **L**ocal **M**emory
I2C	**I**nter-**I**ntegrated **C**ircuit
IC	**I**ntegrated **C**ircuit *deutsch: Integrierte Schaltung*

IF	**I**nter**F**ace
IP	**I**ntellectual **P**roperty
LDW	**L**ane **D**eparture **W**arning
LSB	**L**east **S**ignificant **B**it
LIM	**L**ocal **I**nput **M**emory
LIS	**L**ehrstuhl für **I**ntegrierte **S**ysteme
LOM	**L**ocal **O**utput **M**emory
LUT	**L**ook **U**p **T**able
ME	**M**atching **E**ngine
MGT	**M**ulti **G**igabit **T**ranceiver
MIMD	**M**ultiple **I**nstruction **M**ultiple **D**ata
MPMC	**M**ulti **P**ort **M**emory **C**ontroller
MSB	**M**ost **S**ignificant **B**it
NCD	**N**ative **C**ircuit **D**escription
NGC	**N**ative **G**eneric **C**ircuit
NGD	**N**ative **G**eneric **D**atabase
NMC	**N**ative **M**acro **C**ircuit
NPI	**N**ative **P**ort **I**nterface
OEM	**O**riginal **E**quipment **M**anufacturer
OF	**O**ptischer **F**luß
OPB	**O**n-chip **P**eripheral **B**us
PAR	**P**lace **A**nd **R**oute
PCF	**P**hysical **C**onstraints **F**ile
PE	**P**rocessing **E**lement
PU	**P**rocessing **U**nit
PLB	**P**rocessor **L**ocal **B**us
PPB	**P**ixel **Pro** **B**urst
PRM	**P**artially **R**econfigurable **M**odule
PRR	**P**artially **R**econfigurable **R**egion
PSM	**P**rogrammable **S**witch **M**atrix
RAM	**R**andom **A**ccess **M**emory
ROI	**R**egion **O**f **I**nterest
RIF	**R**ekonfigurierbares **I**nter**F**ace
SE	**S**hape **E**ngine
SoC	**S**ystem **o**n **C**hip
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SRAM	**S**tatic **R**andom **A**ccess **M**emory
TE	**T**aillight **E**ngine
TQs	**T**ransfer **Q**ualifier**s**
TRACE	**T**iming **R**eporter **A**nd **C**ircuit **E**valuator
UART	**U**niversal **A**synchronous **R**eceiver **T**ransmitter

UCF	**U**ser **C**onstraints **F**ile
VHDL	**V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage
VLIW	**V**ery **L**ong **I**nstruction **W**ord
XPS	**X**ilinx **P**latform **S**tudio

Symbolverzeichnis

Symbol	Bedeutung
N_R^F	Anzahl (N) an Konfigurations-Frames (F) per Konfigurations-Row (R)
N_D^F	Anzahl (N) an Konfigurations-Frames (F) per Device (D)
N_D^R	Anzahl (N) an Konfigurations-Rows (R) per Device (D)
N_{BL}	Anzahl (N) an Bufferlines (BL)
N^{RB}	Anzahl (N) an benötigten (required) BRAMs (RB)
N^{UB}	Anzahl (N) an verwendeten (used) BRAMs (UB)
V_{DD}	Versorgungsspannung
$V_{Tn,p}$	Schwellwertspannung eines NMOS bzw. eines PMOS Transistors
V_{GS}	Gate Source Spannung
I_D	Drainstrom
P_{dyn}	dynamische Verlustleistung
P_{stat}	statische Verlustleistung
P_{cap}	Verlustleistung bedingt durch Umladen von Kapazitäten
I_{cap}	Strom, der beim Umladen von Kapazitäten fließt
P_{short}	Verlustleistung durch Kurzschlüsse während des Umschaltvorgangs
I_{short}	Kurzschlußstrom während des Umschaltvorgangs
I_H	Bildhöhe
I_W	Bildweite
B_W	Burstweite
D_W	Datenweite
F_W	Featureweite
A_W	Adressweite
ppt	Pixel pro Takt
H^F	Höhe eines Konfigurations-Frames
f_w	Schreibfrequenz des ICAP FIFOs
f_r	Lesefrequenz des ICAP FIFOs
TP_w	Durchsatz an der Schreibseite des ICAP FIFOs
TP_r	Durchsatz an der Leseseite des ICAP FIFOs

Inhaltsverzeichnis

1 Exposé

In Zukunft werden in Automobil-Systemen video-basierte Fahrer-Assistenz Systeme (FAS) dazu beitragen, die Sicherheit zu erhöhen. Video-basierte FAS erfordern Bildverarbeitung in Echzeit mit Hilfe komplexer Algorithmen. Gegenwärtige Video-Kameras liefern 30 Bilder pro Sekunde und mehr, so daß in dieser Arbeit unter Echzeit die Verarbeitung von mindestens 30 Bildern pro Sekunde verstanden wird. Um Bilddaten zu verarbeiten sind spezielle Plattformen notwendig. Application Specific Integrated Circuits (ASICs) sind integrierte Schaltungen, die auf Kundenwunsch für eine ganz bestimmte Aufgabe gefertigt werden. Masken- und Entwicklungskosten sind dabei vom Kunden allein zu tragen. ICs, die für eine bestimmte Anwendungsdomäne gefertigt wurden, jedoch am Markt verkauft werden und in verschiedenen Geräten eingesetzt werden können, werden als Application Specific Standard Products (ASSPs) bezeichnet. Weder bei ASICs noch bei ASSPs lässt sich die Schaltung im Nachhinein verändern. Prozessoren deren Instruktionssatz für bestimmte Aufgaben optimiert ist, werden als Application Specific Instructionset Processors (ASIPs) bezeichnet. Bei einem Digitalen Signal Prozessor (DSP) handelt es sich um einen Mikroprozessor-Baustein, der auf Echtzeit-Verarbeitung von digitalen Signalen optimiert ist. Die Flexibilität von DSPs oder Mikroprozessoren (General Purpose Prozessors) ist daher wesentlich höher, als die von ASICs oder ASSPs. Für die Echtzeitbildverarbeitung werden heutzutage auch Graphics Processing Units (GPUs) eingesetzt. Ihr Einsatz in Automotive- oder Robotikanwendungen ist jedoch aufgrund der sehr hohen Verlustleistung fraglich.
Aufgrund der einfachen Implementierung und Modifikation werden häufig Software (SW) Modelle der Algorithmen erzeugt. Die Verarbeitung auf General-Purpose Prozessoren (GPPs) erfolgt dabei weitgehend sequentiell. Durch eine Lösung in Hardware (HW) lassen sich verschiedene Teile der Algorithmen parallel verarbeiten. Die Implementierung ist jedoch wesentlich komplexer. Eine reine SW-Implementierung der Bildverarbeitungs-Algorithmen auf den GPPs oder Standard-Mikrocontrollern führt durch die sequentielle Verarbeitung häufig zu einer Verletzung der Echtzeitbedingungen. Daher ist eine Auslagerung performanzlastiger Teile in Hardware sinnvoll. Mit Hilfe dedizierter Hardware (ASICs) für die Bildverarbeitung kann zwar die Berechnung in Echtzeit durchgeführt werden, jedoch fehlt ihnen die notwendige Flexibilität für zukünftige Updates, da nach der Fertigung, wie bereits erwähnt keine Änderungen mehr möglich sind. Bildverarbeitungsalgorithmen für video-basierte FAS sind nicht standardisiert und werden es aller Voraussicht nach auch niemals sein. Die Forschung an geeigneten Algorithmen für diese Anwendungsdomäne wird auch zukünftig voranschreiten. Die zu erwartenden algorithmischen Änderungen, die immer kürzeren Time-to-Market- und Produkt-Zyklen

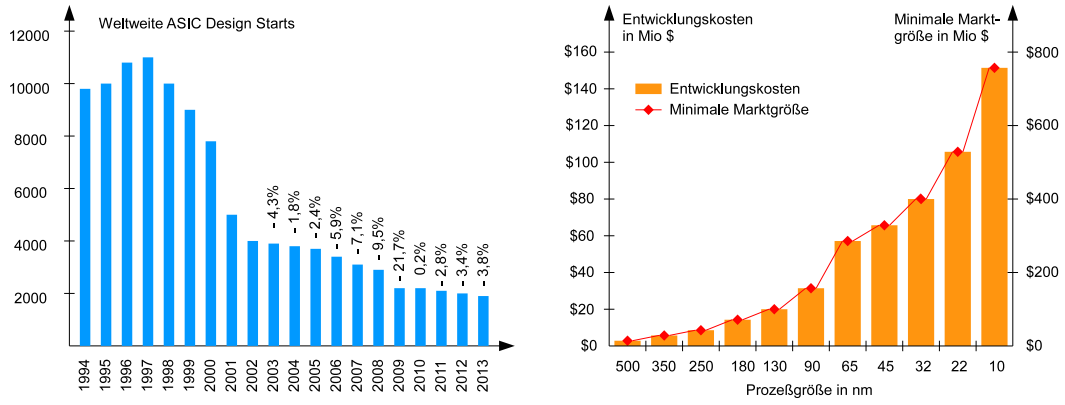
gegenüber stehen, erfordern daher flexible, programmierbare Lösungen für die Echtzeit-Bildverarbeitung. Video-basierte FAS wurden als eines von vielen möglichen Anwendungsgebieten für rekonfigurierbare Hardware gewählt, da sich hier der Vorteil einer flexiblen Hardwarelösung besonders gut demonstrieren lässt. Bestimmte Fahrsituationen, wie beispielsweise das Fahren auf Autobahnen, auf Landstraßen, in Städten, bei Nacht, bei Tag bedürfen spezieller optimierter Algorithmen für die Bildverarbeitung. Field Programmable Gate Arrays (FPGAs) bieten zum einen die Möglichkeit die Bilddaten in Echtzeit zu verarbeiten und zum anderen die Anpassung an verschiedene Fahrsituationen und zukünftige Algorithmen.
Algorithmen für die Bildverarbeitung können in Higher-Level Application Code und die Pixel[1]-Level Operationen unterteilt werden. Der Higher-Level Application Code benötigt ein hohes Maß an Flexibilität und muß einfach zu modifizieren sein, da algorithmische Änderungen häufig hier vorgenommen werden. Die kontrollfluß-dominierten Teile eines Algorithmus sind häufig Teil dieser Klasse. Der Higher-Level Application Code ist daher für eine Implementierung auf einem Standard-Prozessor geeignet. Die Manipulation auf Pixel-Level (häufig die Bildvorverarbeitung) besteht häufig aus den selben sich wiederholenden Operationen, die auf viele Pixel angewendet werden. Die Pixel-Level Operationen lassen sich häufig gut parallelisieren und eignen sich daher für eine Implementierung in Hardware.
Da bei einer Anpassung des Algorithmus häufig sowohl der Higher-Level Application Code und die Pixel-Level Operationen geändert werden müssen, bietet video-basierte Fahrerassistenz die idealen Randbedingungen für den Einsatz von dynamisch partieller Rekonfiguration. Die Echtzeitanforderungen dieser Anwendung machen den Einsatz von Co-Prozessoren zwingend erforderlich. Verschiedene Fahrsituationen bedürfen verschiedener, optimierter Algorithmen für die Bildverarbeitung in der jeweiligen Fahrsituation. Um nicht alle Co-Prozessoren für alle möglichen Fahrsituationen und den damit verbundenen Algorithmen auf dem FPGA vorhalten zu müssen, werden diese nur bei Bedarf auf den FPGA konfiguriert und anschließend wieder entfernt.
In dieser Arbeit wird der Einsatz von dynamisch rekonfigurierbarer Hardware (Field Programmable Gate Arrays) für die Echtzeit-Bildverarbeitung am Beispiel einer Automobilanwendung, eines video-basierten FAS beschrieben.

1.1 Motivation

Die Fortschritte in der Halbleiterindustrie führen zu immer höheren Transistordichten in integrierten Schaltungen (ICs). 1965 sagte Gordon Moore eine Verdopplung der Transistordichte alle 18 Monate bis ins Jahr 1975 voraus [Moo65]. Die Vergangenheit hat jedoch gezeigt, dass das so genannte Moore'sche Gesetz weit länger (bis in die heutige Zeit) Bestand hat, wenn mit einer Verdopplung der Transistordichte auch erst alle 2 Jahre zu

[1] Pixel: Picture Element

rechnen ist. Höhere Transistordichten führen zu immer kleineren und leistungsfähigeren ICs. Auf der anderen Seite wird die Herstellung von immer kleineren Transistoren immer aufwändiger und damit kostspieliger. ASICs sind in der Entwicklung (Maskenkosten etc.) sehr teuer, dagegen in der Massenfertigung relativ preiswert. Im Gegensatz dazu sind die Kosten von FPGAs in der Anschaffung bei hohen Stückzahlen im Vergleich zu ASICs höher, jedoch punkten FPGAs durch ihre Flexibilität, die kostengünstigere Entwicklung und die schnelle Aussicht auf Erfolg.
Das US-Marktforschungsinstitut Gartner sagt jedoch für die kommenden Jahre einen Rückgang von neuen ASIC Designs voraus [Gar09]. Die in Abbildung 1.1(a) dargestellte Prognose, zeigt einen fast stetigen Rückgang bei abnehmender Transistor Strukturgröße. Dies ist bedingt durch die immer aufwendigere Herstellung von ASICs und ASSPs und den damit verbundenen Kosten, deren Entwicklung mit abnehmender Strukturgröße in Abbildung 1.1(b) dargestellt ist.



(a) Jährliche ASIC Design Starts [Gar09] **(b)** ASIC/ASSP Entwicklungskosten [Uhm08]

Abbildung 1.1: Trends in der Halbleiterindustrie

Aus Abbildung 1.1(b) ist ersichtlich, dass in Zukunft der Entwurf neuer ASICs und ASSPs nur noch in Massenmärkten mit sehr hohen Stückzahlen lohnt. Marktsegmente, die aufgrund der Stückzahlen traditionell von ASICs und ASSPs dominiert wurden, werden damit zunehmend unrentabel. Eine Möglichkeit diese Segmente zu bedienen, besteht in der Verwendung von FPGAs. Da gegenwärtige FPGAs über Milliarden an Transistoren verfügen [San10], bilden sie heutzutage eine echte Alternative zu ASICs und ASSPs, deren Entwicklung in vielen Domänen mit zu hohem Risiko behaftet ist. Ein Vergleich zwischen 90nm ASICs und 90nm FPGAs hinsichtlich Logikdichte, erreichbaren Taktfrequenzen und Verlustleistung ist in [KR06] beschrieben. Laut Kuon et. al. [KR06] erzielt ein ASIC bei der Logikdichte im Durchschnitt einen 21-fach höheren Wert als ein SRAM-basierter FPGA, wobei dieser Unterschied durch den Einsatz weiterer hartverdrahteter

Elemente wie CPUs oder DSP Elementen weiter schrumpft. Da nachträgliche Änderungen auf ASICs nicht möglich sind, muß die gesamte Funktionalität auf dem Device verfügbar sein, selbst wenn nur ein Bruchteil davon benötigt wird. Um die Logikdichte bei FPGAs zusätzlich zu erhöhen kann die Rekonfiguration eingesetzt werden. Verschiedene Module, die nicht gleichzeitig benötigt werden, können sich dabei dieselbe Fläche auf dem FPGA teilen. Anstatt also die gesamte Funktionalität auf dem Device vor zuhalten, werden nur die zu einem bestimmten Zeitpunkt benötigten Elemente geladen und anschließend wieder entfernt.

1.2 Zielsetzung der Arbeit

Zielsetzung dieser Arbeit ist die Evaluation von SRAM-basierten FPGAs als mögliche Plattform für die Bildverarbeitung beispielsweise in video-basierten FAS. Im speziellen soll dabei die Einsatzmöglichkeit der dynamisch partiellen Rekonfiguration (DPR) untersucht werden, um während der Laufzeit des Systems benötigte Hardware-Acceleratoren (HWAs) für die Bildverarbeitung auf den FPGA zu konfigurieren, während gleichzeitig nicht benötigte Komponenten entfernt werden sollen. Dieser dynamische Austausch von HWAs dient zur Einsparung von Ressourcen auf dem FPGA. Dadurch kann ein kleineres FPGA verwendet werden, was wiederum die Kosten senkt. Zusätzlich sind Konzepte notwendig, um diesen dynamischen Austausch verlustfrei, das bedeutet ohne dass ein Video-Frame verworfen werden muß, durchzuführen. Um verlustfrei zu rekonfigurieren, sollte dieser Vorgang so schnell wie möglich durchgeführt werden. Das Beschleunigen der Rekonfiguration auf der einen, hat auf der anderen Seite jedoch Auswirkungen auf weitere Optimierungskriterien wie Ressourcenverbrauch und Verlustleistung. Ein weiteres Ziel dieser Arbeit besteht daher in die Entwicklung von Methoden zur besonders schnellen dynamischen Rekonfiguration, bei gleichzeitiger Minimierung des Ressourcenverbrauchs und der Verlustleistung. Aus diesen Zielforderungen leiten sich die folgen Aufgabenstellungen ab:

- Konzeption und Aufbau eines modularen integrierten Systems zur Echtzeit-Bildverarbeitung.
- Erweiterung zu einem dynamisch rekonfigurierbaren System mit Entwicklung aller dafür notwendigen Komponenten.
- Analyse des durch die schnelle dynamisch partielle Rekonfiguration verursachten Overheads.
- Entwurf von Methoden zur Reduktion dieses Overheads.
- Quantifizierung der Vorteile durch den Einsatz dieser Methoden anhand von Hardware-Prototypen für die Echtzeit-Bildverarbeitung.

- Bewertung der Ergebnisse.

1.3 Gliederung der Arbeit

Im folgenden Kapitel 2 werden zunächst die Grundlagen erläutert, die zum späteren Verständnis dieser Arbeit nötig sind. Dieses Kapitel umfasst eine kurze Einführung in die Bildverarbeitung, den Aufbau von FPGAs und grundlegende Eigenschaften von System-on-chip (SoC) Architekturen.
Basierend auf diesem Kapitel wird in Kapitel 3 der Stand der Technik vorgestellt. Dabei werden bereits verfügbare SoCs für die Bildverarbeitung im Automobil als auch existente dynamisch rekonfigurierbare FPGA-Systeme für die Bildverarbeitung vorgestellt. Das Kapitel endet mit einer Zusammenfassung der Punkte, die den Bedarf an Weiterentwicklung charakterisieren. Diese Weiterentwicklungen werden in den folgenden Kapiteln detailliert erörtert und stellen den Kern dieser Arbeit dar.
In Kapitel 4 wird die AutoVision Architektur vorgestellt. Diese Architektur, die beispielhaft für ein video-basiertes FAS eingesetzt wird, nutzt HWAs, um eingebettete Prozessoren von besonders performanzlastigen Bildverarbeitungsaufgaben zu entlasten. Das Grundgerüst der HWAs bildet eine modular aufgebaute Pixelverarbeitungspipeline, deren Komponenten sich flexibel kombinieren lassen, um bestimmte Bildverarbeitungsaufgaben zu übernehmen. Neben einer detaillierten Beschreibung der einzelnen Komponenten dieser Pixelverarbeitungspipeline, werden außerdem die damit realisierten HWAs vorgestellt.
Die Architektur für den Austausch dieser HWAs zur Laufzeit des Systems mit Hilfe der dynamisch partiellen Rekonfiguration wird in Kapitel 5 erläutert. Im AutoVision System sollen die relevanten HWAs möglichst schnell, ohne daß ein Video-Frame verworfen werden muß, geladen werden. Daher befasst sich ein wesentlicher Teil dieses Kapitels mit der Analyse und Reduktion von Overhead, der durch die Verwendung von DPR in einem System entsteht. Die Auswirkungen bei möglichst schneller Rekonfiguration auf weitere Entwurfskriterien, wie Fläche und Verlustleistung wird ebenfalls in diesem Kapitel detailliert vorgestellt. Den Abschluß dieses Kapitels bildet ein Konzept zum dynamischen Powermanagement in DPR Systemen.
Die in Kapitel 4 und Kapitel 5 vorgestellten Methoden und Konzepte wurden in zwei Hardware-Prototypen implementiert und anschließend evaluiert. Zusammenfassend sind daher in Kapitel 6 die Ergebnisse hinsichtlich Verarbeitungsgeschwindigkeit der HWAs, benötigter Fläche des Gesamtsystems bei Einsatz der DPR, erzielte Rekonfigurationsgeschwindigkeit (und der damit verbundene Durchsatz an Rekonfigurationsdaten) und Verlustleistung dargestellt. Um die erzielten Ergebnisse zu bewerten, werden diese mit dem Stand der Technik aus Kapitel 3 verglichen.
Das letzte Kapitel 7 schließt diese Forschungsarbeit ab. Zusammenfassend werden hier die wesentlichen Innovationen dieser Arbeit genannt und daraus weitere Forschungsziele für einen Ausblick abgeleitet.

2 Grundlagen

In diesem Kapitel werden die Grundlagen vorgestellt, die zum späteren Verständnis notwendig sind.

2.1 Bildverarbeitung

Ein digitales Bild besteht aus einer zweidimensionalen Anordnung von einzelnen Bildpunkten, so genannten Pixeln.Um Information aus einem Bild zu extrahieren, muß dieses zunächst analysiert werden. Für diesen Analyseschritt wird oft ein Eingangsbild mit einem Prozessierungsfenster abgesucht, um beispielsweise bestimmte Muster zu finden. Dazu wird (beispielsweise bei Filterungen in digitalen Bildern) eine Koeffizientenmatrix (folgend als Filter-Kern bezeichnet) pixelweise über das Eingangsbild geschoben. Das Schieben eines Filter-Kerns über das Bild wird als Sliding-Window Operation bezeichnet. Bei dieser Operation wird das zentrale Pixel nach einer festen Vorschrift in Bezug auf seine Nachbarschaft umgewandelt. Die Größe des Sliding-Windows kann je nach Bildverarbeitungsoperation unterschiedlich sein. Die Reihenfolge in der das Bild abgesucht wird (Scanprozess) hat keinen Einfluß auf die Ergebnisse der Bildverarbeitung. Jedoch sollte, abhängig davon, wie die Eingangsbilddaten im Speicher angeordnet sind, der geeignete Scanprozess gewählt werden. In dieser Arbeit wird ein zeilenweiser Scan-Prozess von links nach rechts und von oben nach unten verwendet.

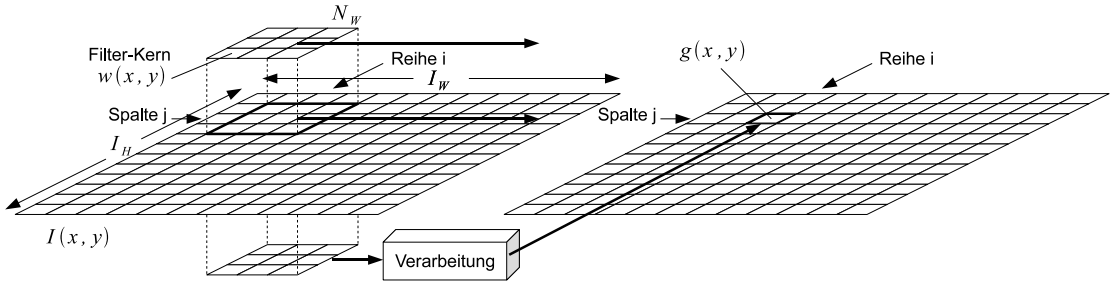


Abbildung 2.1: Verarbeitung mit einem Sliding Window

In Abbildung 2.1 ist ein Prozess mit einem Sliding Window dargestellt. $I(x, y)$ repräsentiert dabei das zweidimensionale Eingangsbild und x und y beliebige Pixelkoordinaten. Über dieses Eingangsbild $I(x, y)$ der Höhe I_H und der Weite I_W wird ein 3×3 Filter-Kern

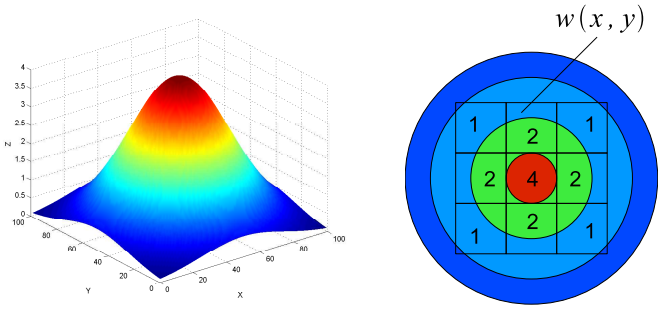
$w(x, y)$ geschoben. Das zentrale Pixel des Filter-Kerns hat somit eine Nachbarschaftsweite N_W von 1. Betrachtet werden nur quadratische Filter-Kerne, deren Dimensionen sich aus der Nachbarschaftsweite N_W bestimmen ($2N_W+1 \times 2N_W+1$). Mathematisch gesehen wird hierbei eine Konvolution (Faltung) bzw. Korrelation [GW08] der Eingangsbilddaten mit dem Filter-Kern durchgeführt. Das Ergebnis ist ein Pixel $g(x, y)$ im Ausgangsbild. Die generalisierte Darstellung einer Konvolution, mit unterschiedlicher Gewichtung der benachbarten Pixel im Filter-Kern $w(x, y)$ wird durch Gleichung 2.1 repräsentiert [GW08].

$$g(x,y) = \sum_{s=-N_W}^{+N_W} \sum_{t=-N_W}^{+N_W} w(s,t) I(x+s, y+t) \tag{2.1}$$

Die Konvolution entspricht einer Sliding-Window Operation über das Eingangsbild. Viele Bildvorverarbeitungsoperationen wie Mittelwertbildung, Konvolutionsoperationen, Morphologische Operatoren, Kantenfilter und Eckendetektoren [GW08] benötigen im initialen Schritt eine Verarbeitung mit einem Sliding-Window.
Ein typisches Filter in der Bildvorverarbeitung ist das Tiefpassfilter zur Glättung von Bildern. Der Filter-Kern des Tiefpassfilters repräsentiert einen Gaußschen Filter, der in Gleichung 2.2 dargestellt ist.

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x^2+y^2)/2\sigma^2} \tag{2.2}$$

Die Koeffizienten im Filter-Kern entsprechen hierbei der „gaußschen Glocke“. Die Pixel in der Nähe des zentralen Pixels werden dabei stärker gewichtet, wie in Abbildung 2.2 zu erkennen ist.



(a) Gaußsche Glocke **(b)** Binomialfilter

Abbildung 2.2: Gaußsche Glocke und Bestimmung der Koeffizienten des Binomialfilters

Hinsichtlich Glättung und Reduktion von Gaußschem Rauschen bietet der in Abbildung 2.2(b) dargestellte 3×3 Binomialfilter optimale Eigenschaften. In Gleichung 2.3 ist

die Berechnung einer Gaußglättung dargestellt, in der ein Bildausschnitt mit dem 3×3 Binomialfilter gefaltet wird.

$$\begin{aligned} g(x,y)_{TP} = \frac{1}{16}[&I(x-1,y-1) + 2I(x,y-1) + I(x+1,y-1) + \\ &2I(x-1,y) + 4I(x,y) + 2I(x+1,y) + \\ &I(x-1,y+1) + 2I(x,y+1) + I(x+1,y+1)] \end{aligned} \tag{2.3}$$

Da die Gewichtungen einzelner Pixel in Gleichung 2.3 voneinander unabhängig sind, lassen sich diese sehr einfach parallelisieren. Die Summation der gewichteten Pixel wird im Anschluß durchgeführt, bevor in einem letzten Schritt die Summe durch 16 geteilt wird. 16 entspricht in diesem Fall der Summe aller Einzelgewichte im Filter-Kern. Eine mögliche Plattform zur Parallelisierung von Bildverarbeitungsalgorithmen stellen Field Programmable Gate Arrays (FPGAs) dar, die im folgenden Abschnitt vorgestellt werden.

2.2 FPGAs

Da Application Specific Integrated Circuits (ASICs) weniger Flexibilität erlauben und die Entwicklungskosten in den letzten Jahren deutlich gestiegen sind [Gar09], gewinnen Field Programmable Gate Arrays (FPGA) in verschiedensten Anwendungsdomänen an Bedeutung. Im Gegensatz zu ASICs ist die realisierte logische Funktionalität bei FPGAs nach der Fertigung des Devices applikationsspezifisch konfigurierbar.
Die Idee eines reprogrammierbaren Devices, dem so genannten Configurable Logic Array Konzept wurde von Sven E. Wahlstrom bereits 1967 vorgestellt [Wah67]. Anstatt festverdrahtete Verbindungen einzusetzen, schlug Wahlstrom die Verwendung von zusätzlichen Logikgattern vor, um dies zu realisieren. 1984, nach 17 Jahren, wurde das erste FPGA [CDF+86] dann von Ross Freeman, einem der Firmengründer von Xilinx erfunden. Das erste kommerziell verfügbare FPGA (Xilinx' XC2064) gibt es seit 1985.
In diesem Abschnitt werden zunächst die verschiedenen Arten von FPGAs vorgestellt. Anschließend wird der Aufbau der verwendeten FPGAs und die zur Konfiguration verwendbaren Schnittstellen erläutert. Danach werden die Konfigurations-Daten, die so genannten Bitströme beschrieben, sowie der Vorgang mit dem ein FPGA durch einen Bitstrom konfiguriert wird. Abschließend werden verschiedene Arten der Konfiguration diskutiert.

2.2.1 Arten von FPGAs

FPGAs verfügen über die Möglichkeit eine bestimmte Konfiguration zu speichern. Abhängig von der Art des verwendeten Speichers unterscheidet man verschiedene Typen von FPGAs. FPGAs können auf flüchtigem und nicht-flüchtigem Speicher basieren. Bei

Flash-basierten FPGAs (nicht-flüchtig) bleibt die Konfiguration erhalten, während bei SRAM[1] basierten FPGAs (flüchtig) jedes mal neu konfiguriert werden muß, wenn die Versorgungsspannung des Devices abgeschaltet wurde. Eine Übersicht von verschiedenen FPGA-Technologien verschiedener Hersteller ist in [Hüb07] zu finden. Da in dieser Arbeit ausschließlich SRAM basierte FPGAs der Firma Xilinx verwendet wurden (da diese die in Abschnitt 2.2.6 vorgestellte dynamisch partielle Rekonfiguration unterstützen), wird im Folgenden nur der Aufbau dieser Art von FPGAs vorgestellt. Betrachtet wurden die verschiedenen FPGA-Familien Virtex-II Pro (V2P), Virtex-4 (V4) und Virtex-5 (V5).

2.2.2 Aufbau von SRAM-basierten FPGAs

Der Aufbau der meisten FPGA Devices ist sehr ähnlich und besteht auch bei Xilinx Devices im Wesentlichen aus einer 2-dimensionalen Anordnung (Array) von konfigurierbaren Logik-Blöcken (CLBs), die von konfigurierbaren Verbindungsleitungen (Verdrahtungskanälen) umschlossen sind und IO-Blöcken (IOBs). Zusätzlich stehen Block RAM (BRAM) Module als on-chip Speicher, sowie Multiplizierereinheiten zur Verfügung. Diese sind als Elemente des funktionalen Layers in Abbildung 2.3 dargestellt.

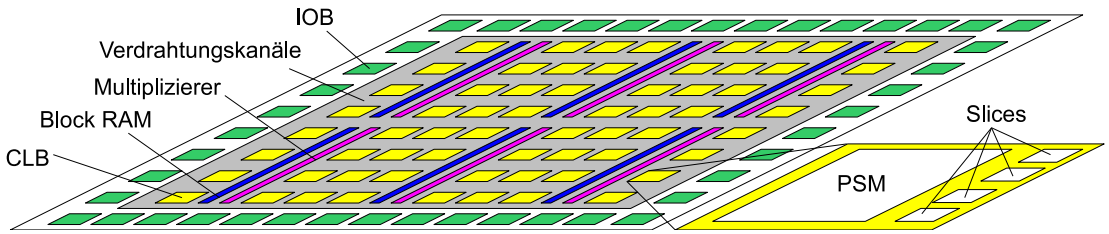


Abbildung 2.3: Schematischer Aufbau des funktionalen Layers

Ein CLB besteht aus einer Programmierbaren Switch Matrix (PSM) und einer Anzahl an Slices, die abhängig von der FPGA Familie ist. In der PSM können verschiedene Leitungssegmente über Pass-Transistoren miteinander verschaltet werden, wie in Abbildung 2.4 dargestellt.
Damit lassen sich beispielsweise beliebige Slices miteinander verbinden. Ein Slice besteht aus Lookup-Tablen (LUTs), Multiplexern, Arithmetik-Carry Logic und Registern, die als Flip Flop oder Latch konfiguriert werden können. Ein Virtex-4 Slice ist schematisch in Abbildung 2.5 dargestellt.
Der Aufbau von V4 und V5 Slices und ein zusätzlicher Vergleich mit Altera und Actel Slices ist in [Cor09] beschrieben. Kontrolliert wird der funktionale Layer vom Konfigurations-Layer. Der Konfigurations-Layer eines FPGA kann als zweidimensionaler, SRAM-basierter Speicher aufgefasst werden. Durch Aufteilung in funktionalen und Konfigurations-Layer

[1] SRAM: Static Random Access Memory

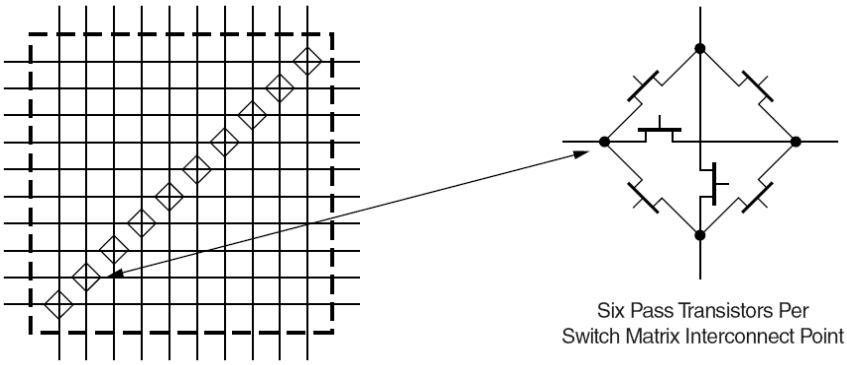


Abbildung 2.4: Programmierbare Switch Matrix [Xil08c]

kann ein SRAM basiertes FPGA als 2-Layer Modell betrachtet werden. Funktionaler Layer und Konfigurations-Layer sind dabei direkt miteinander verbunden. Das bedeutet, dass sich jede Änderung im Konfigurations-Layer direkt auf den funktionalen Layer auswirkt. Durch den Konfigurations-Layer wird festgelegt, welche Schaltung im funktionalen Layer realisiert ist. Direkter Zugriff besteht nur auf den Konfigurations-Layer. Da nur ein Konfigurations-Layer existiert und dieser direkt mit dem funktionalen Layer verbunden ist, kann nicht eine komplette Konfiguration vorgeladen (Prefetching) und in einem Taktzyklus aktiv geschalten werden, wie beispielsweise bei Multi-Context FPGAs [CH00]. Abbildung 2.6 zeigt das Layermodell eines FPGAs.
Der Konfigurations-Layer ist in vertikaler Richtung in so genannte Konfigurations-Rows unterteilt. Die Anzahl der Konfigurations-Rows ist deviceabhängig. In horizontaler Richtung ist der Konfigurations-Layer in so genannte Konfigurations-Frames unterteilt. Ein Konfigurations-Frame hat eine Breite von einem Bit und entspricht der atomaren Einheit, die in einem Xilinx FPGA konfiguriert werden kann.
Die Höhe eines Konfigurations-Frames wird mit H^F bezeichnet und ist ebenfalls deviceabhängig. Eine bitweise Manipulation des Konfigurations-Layers ist nicht möglich, da die einzelnen SRAM-Speicherzellen nicht separat adressiert werden können. Um bitweise zu manipulieren, muß also immer ein kompletter Konfigurations-Frame geschrieben werden. Das Überschreiben der gegenwärtigen Konfiguration von SRAM Zellen mit dem selben Wert verursacht dabei in der Virtex Familie keine Störimpulse. Dies wird als Glitchless Switching bezeichnet [BBHN04].
Konfigurations-Frames sind für die Konfiguration bestimmter Elemente im funktionalen Layer zuständig. Abhängig von dieser Zuordnung werden Konfigurations-Frames zu so genannten Konfigurations-Columns zusammengefasst. Dementsprechend gibt es eigene Konfigurations-Columns für Ein-/Ausgabe Blöcke (IOB), Ein-/Ausgabe Interconnects (IOI), Configurable Logic Blocks (CLBs), Block RAM (BRAM), BRAM Inter-

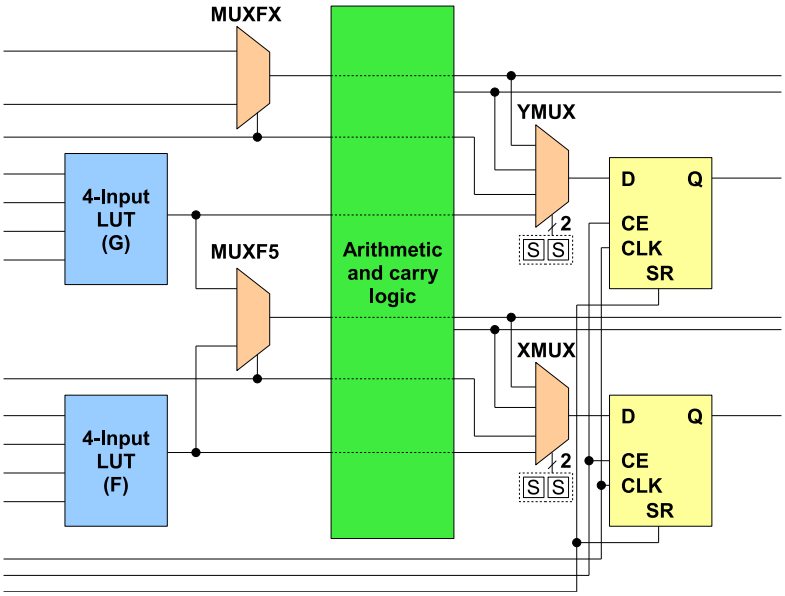


Abbildung 2.5: Virtex-4 Slice [Cor09]

connect (INT), Digitale Signal Prozessor Blöcke (DSP), Multi-Gigabit Tranceiver Blöcke (MGT) und Globale Taktblöcke (GCLK). Abhängig davon für welchen Block im funktionalen Layer die Konfigurations-Columns zuständig sind, besteht eine Konfigurations-Column aus einer unterschiedlichen Anzahl an Konfigurations-Frames. Die Anzahl an Konfigurations-Frames pro Konfigurations-Column variiert aufgrund der Komplexität zwischen den verschiedenen FPGA Familien und ist in Tabelle 2.1 dargestellt.

FPGA Familie	IOB Column	IOI Column	CLB Column	BRAM Column	BRAM INT Column	DSP Column	MGT Column	GCLK Column
V2P	4	22	22	64	22	n.a.	n.a.	4
V4	30	n.a.	22	64	20	21	20	3
V5	54	n.a.	36	128	30	28	32	4

Tabelle 2.1: Anzahl an Konfigurations-Frames pro Konfigurations-Column

Die Anzahl an Konfigurations-Rows pro Device N_D^R und die Anzahl an Konfigurations-Frames pro Konfigurations-Row N_R^F bestimmt somit die Größe des Konfigurations-Layers. Die Gesamtanzahl an Konfigurationframes pro Device N_D^F berechnet sich nach Gleichung 2.4.

$$N_D^F = N_R^F * N_D^R \tag{2.4}$$

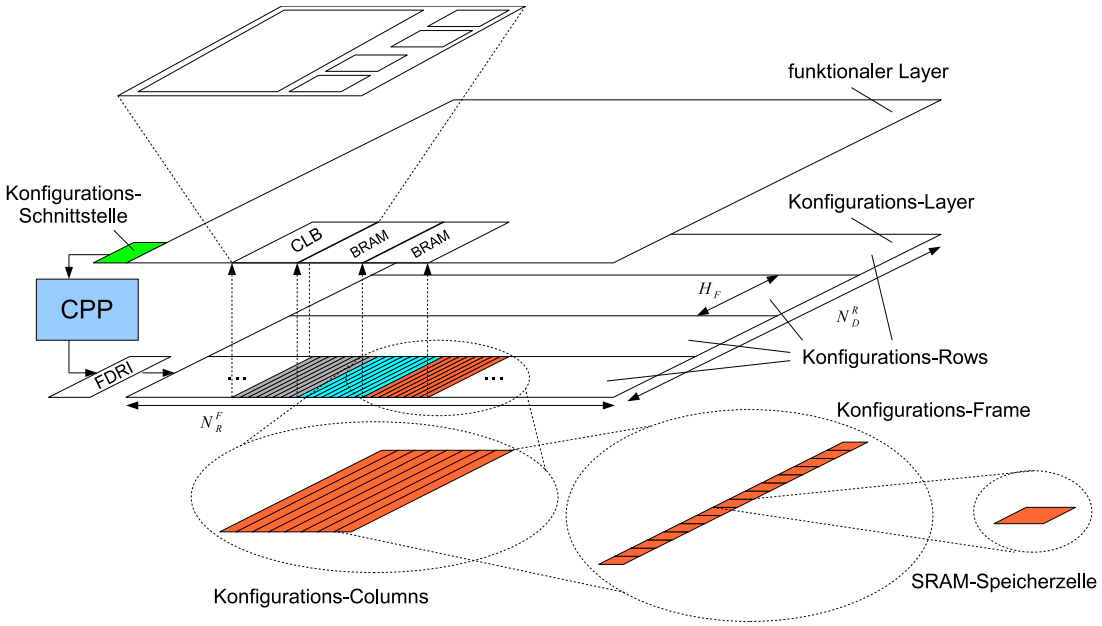
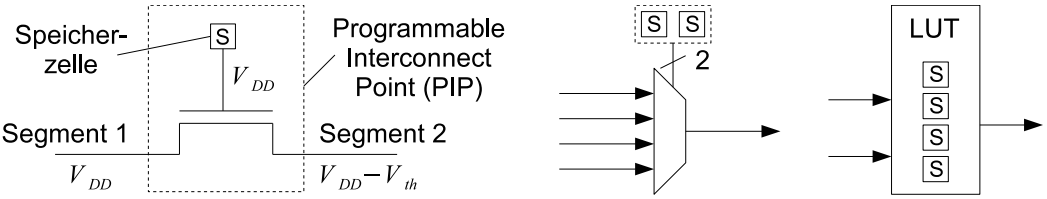


Abbildung 2.6: Die beiden Layer eines FPGA mit der hierarchischen Unterteilung des Konfigurations-Layers.

Die Berechnung von N_D^F funktioniert für alle FPGA Familien analog. Während die Konfigurations-Frames der neueren V4, V5 oder V6 Devices innerhalb der FPGA Familie alle die selbe Höhe H^F besitzen, unterscheidet sich H^F zwischen den Devices der V2P Familie. In V2P Devices überspannt ein Konfigurations-Frame immer die volle Höhe. Damit ist N_D^R innerhalb der V2P Familie immer 1. Die Gesamtanzahl an Konfigurations-Bits eines Devices N_D^B läßt sich mit Gleichung 2.5 berechnen.

$$N_D^B = N_D^F * H^F \tag{2.5}$$

Eine umfassende Auflistung aller in diesem Abschnitt genannten Parameter für verschiedene FPGA Familien ist in den Tabellen A.3 bis A.5 dargestellt.
Die Konfiguration jeder SRAM-Speicherzelle hat direkte Auswirkung auf den funktionalen Layer, um beispielsweise Leitungssegmente zu verbinden oder bestimmte Eingänge von Multiplexern durchzuschalten. In Abbildung 2.7 sind drei Elemente abgebildet, welche direkt durch SRAM-Speicherzellen konfiguriert werden.
Die Verbindungen in Programmierbaren Switch Matrizen (s. Abbildung 2.4) werden durch Konfiguration der SRAM-basierten Speicherzellen realisiert. Durch die Realisierung mit Pass-Transistoren (s. Abbildung 2.7(a)), können zwei Leitungssegmente miteinander verbunden werden. Der Verbund von Speicherzelle und Pass-Transistor wird als Programmable Interconnect Point (PIP) bezeichnet [Wan98]. Wenn die Speicherzelle mit



(a) PIP **(b)** Multiplexer **(c)** Lookup-Table

Abbildung 2.7: Konfiguration verschiedener Elemente durch Speicherzellen eines FPGAs [Wan98]

einer 1 konfiguriert wurde, wie in Abbildung 2.7(a) dargestellt, liegt (V_{DD}) am Gate der NMOS Transistors an. Aufgrund der Schwellspannung V_{th} des verwendeten NMOS Transistors kann ein 1-Pegel (V_{DD}) auf Leitungssegment 2 nicht vollständig erreicht werden. Die dadurch auftretenden Probleme sind in [KB06] und [BR99] erläutert. Der Einsatz von Transmission Gates [Ste09], würde das Problem zwar beheben, jedoch auch die Anzahl der verwendeten Transistoren verdoppeln. Um das Problem zu lösen, kann, wie in [BR99] beschrieben, die Steuerspannung am Gate der Pass-Transitoren erhöht werden (Gate Boosting). Auch der Selecteingang von einigen Multiplexern (beispielsweise XMUX und YMUX in Abbildung 2.5) wird durch SRAM-Speicherzellen gesteuert und entscheidet damit, welcher der Eingänge durchgeschaltet wird [Wan98].
Für einen 2^x-zu-1 Multiplexer werden x Speicherzellen zur Steuerung benötigt. In Abbildung 2.7(b) ist ein 4-zu-1 Multiplexer dargestellt. Der Aufbau einer Lookup Table (LUT) mit 2 Eingängen ist in Abbildung 2.7(c) dargestellt. Eine LUT ist eine Tabelle in der zuvor berechnete Werte in den Speicherzellen hinterlegt sind. Somit lassen sich beliebige boolsche Funktionen realisieren. Die Eingänge einer LUT steuern somit welcher zuvor gespeicherte Wert an den Ausgang durchgeschaltet werden soll. Für eine LUT mit x Eingängen werden 2^x Speicherzellen verwendet. Ähnlich wie die LUTs wird auch der On-Chip BRAM mit Hilfe der SRAM Speicherzellen realisiert.

2.2.3 Schnittstellen zur Konfiguration von FPGAs

Um Konfigurations-Daten in den Konfigurations-Layer zu schreiben oder aus diesem zu lesen, sind so genannte Konfigurations-Schnittstellen notwendig. Im Normalfall können Xilinx FPGAs über 3 Schnittstellen von außen (externe Schnittstelle) und über eine Schnittstelle von innen (interne Schnittstelle) rekonfiguriert werden. Die erste externe Schnittstelle ist der Test Access Port (TAP) oder die JTAG Schnittstelle. JTAG steht für Joint Test Action Group, die den Standard IEEE 1149.1 für das Testen, Debuggen und Programmieren von Schaltungen entwickelte. Durch Hinzufügen von benutzerdefinierten Instruktionen ist es über den TAP möglich, den FPGA zu konfigurieren, zu

verifizieren oder die Konfigurations-Daten auszulesen (Readback). Details über den TAP sind beispielsweise in [Xil05] zu finden.
Eine weitere Möglichkeit FPGAs von außen zu konfigurieren besteht in der Verwendung der seriellen Konfigurations-Schnittstelle. Über diese Schnittstelle kann ein Bit pro Taktzyklus übertragen werden, wodurch sich der Name „Serielle Konfigurations-Schnittstelle“ erklärt. Die Schnittstelle wird häufig verwendet, um Konfigurations-Daten aus beispielsweise einem PROM (Programmable Read-Only Memory) zu laden, um den SRAM-basierten FPGA beim Anlegen einer Versorgungsspannung initial zu konfigurieren. Weitere Details über die serielle Konfigurations-Schnittstelle sind in [Xil05] aufgeführt.
Select-Map stellt eine weitere externe Schnittstelle zur Konfiguration von FPGAs dar. Im Gegensatz zu den anderen beiden externen Schnittstellen verfügt die Select Map Schnittstelle auf V5 über einen bidirektionalen 8-Bit, 16-Bit oder 32-Bit Bus, um parallel Konfigurations-Daten zu schreiben oder zu lesen. In V2P und V4 Devices hat die Select Map Schnittstelle eine Weite von 8 Bit. Durch das parallele Schreiben von Konfigurations-Daten, kann die Zeit, die benötigt wird um ein FPGA zu konfigurieren drastisch verkürzt werden.
In einigen Fällen ist es notwendig, dass das eingebettete System selbst eine Rekonfiguration anstößt, um sich damit selbst zu modifizieren. Diese Selbst-Rekonfiguration oder Rekonfiguration von innen wird durch den Internal Configuration Access Port (ICAP) ermöglicht. Der ICAP hat die selben Anschlüsse wie die Selct MAP Schnittstelle und ist in Abbildung 2.8 dargestellt.

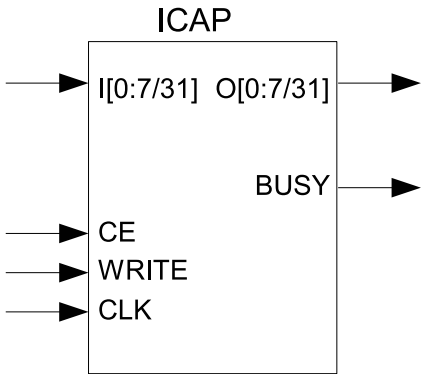


Abbildung 2.8: ICAP Schnittstelle

Der ICAP hat auf V2 und V2P eine Eingangs- und Ausgangsdatenweite von $I = O = 8$ Bit. Auf V4 und V5 kann zwischen einer Eingangs- und Ausgangsdatenweite von $I = O = 8$ Bit oder $I = O = 32$ Bit gewählt werden. Über den CLK Eingang wird der Takt angelegt, mit dem die Konfigurations-Daten an ICAP geliefert werden und mit dem der interne Configuration Paket Prozessor *CPP* versorgt wird. Über das $Write$ Signal wird

gesteuert, ob es sich um einen Konfigurations- oder Auslese-Prozeß handelt. Das low aktive Signal CE aktiviert den I bzw. O Datenbus. Das *busy* Signal zeigt an, ob ICAP gerade mit der Verarbeitung von Daten beschäftigt ist und daher keine neuen Daten annehmen kann. Auf V2P wird dieses Signal beim Lesen und Schreiben der Konfigurations-Daten getrieben, bei V4, V5 und V6 nur noch während des Auslese-Prozesses. Nachdem Aufbau der FPGAs und die Konfigurations-Schnittstellen beschrieben wurden, wird nun der Aufbau der Konfigurations-Daten, den so genannten Bitströmen erläutert.

2.2.4 Konfigurations-Daten - Der Xilinx Bitstrom

Die Konfiguration eines FPGAs erfolgt über so genannte Bitströme, die über eine Konfigurations-Schnittstelle in den Konfigurations-Layer geschrieben werden. Ein Bitstrom besteht aus verschiedenen Segmenten. Dem Bitstromheader, den eigentlichen Konfigurations-Daten und einem Pad-Frame[2].
Im Bitstromheader sind verschiedende Informationen über den Bitstrom, wie der Name des Designs und der Name des Devices, für den der Bitstrom erzeugt wurde enthalten. Zusätzlich sind im Header das Datum und die Uhrzeit der Bitstromerzeugung kodiert. Direkt im Anschluß an den Bitstromheader folgen ein oder mehrere Dummy Worte und ein Synchronisationswort (0xAA995566), zur Initialisierung der Konfigurations-Schnittstelle. Die Bitstromdaten müssen in der Reihenfolge an die Konfigurations-Schnittstelle geliefert werden, in der sie im Bitstrom vorkommen. Da das Synchronisationswort im Bitstrom erst hinter dem Header steht, sind die Daten innerhalb des Headers für die Konfiguration des FPGAs unerheblich. Eine genaue Bedeutung der einzelnen Segmente des Bitstromheaders ist in Appendix A.1 aufgeführt. Nach dem Bitstromheader folgen die eigentlichen Konfigurations-Daten. Die Konfigurations-Daten sind in Pakete unterteilt, die ihrerseits wieder aus Paket-Header und Paket-Daten bestehen. Der Paket-Header besteht aus einem 32 Bit Wort und initialisiert die Konfigurations-Register innerhalb des in Abschnitt 2.2.5 beschriebenen *Configuration Paket Processors (CPP)*. Der *CPP* ist für das Schreiben der Konfigurations-Daten an die richtige Stelle im Konfigurations-Layer zuständig. Die Paket-Daten können eine beliebige Länge aufweisen, die im Paket-Header allerdings spezifiziert sein muß. Auf die Bedeutung des abschließenden Pad-Frames beim Schreiben einer Konfiguration wird im speziellen in Abschnitt 2.2.5 eingegangen. Wird ein Bitstrom verwendet, um nur einen Teil und nicht das gesamte FPGA zu konfigurieren, spricht man von einem partiellen Bitstrom.

2.2.5 Der Konfigurations-Prozeß

Um zu verstehen, wie ein FPGA durch einen Bitstrom konfiguriert wird, ist in diesem Abschnitt der Konfigurations-Prozeß erläutert, der auch für eine Rekonfiguration, also das

[2]Unter dem Begriff Padding wird das Auffüllen mit Daten verstanden. Ein Pad-Frame dient daher als Füll-Frame innerhalb eines Bitstroms.

erneute Programmieren eines FPGAs bei bereits geladener Konfiguration gültig ist. Der Konfigurations- sowie der Auslese-Prozeß der SRAM Zellen eines Xilinx FPGAs ist unter anderem in [Xil05, Xil07, Xil09d] beschrieben. In diesem Abschnitt werden nur die für die Rekonfiguration notwendigen Elemente vorgestellt, die in Abbildung 2.9 dargestellt sind. Die Konfiguration der SRAM Zellen funktioniert in Xilinx FPGAs über das Beschreiben von Konfigurations-Registern innerhalb des *Configuration Packet Processors (CPP)*. Die für den (Re-)Konfigurations-Prozeß wichtigen Register sind das Frame Address Register (FAR), das Frame Data Register for Input (FDRI), das Command Register (CMD) und das Multiple Frame Write Register (MFWR).

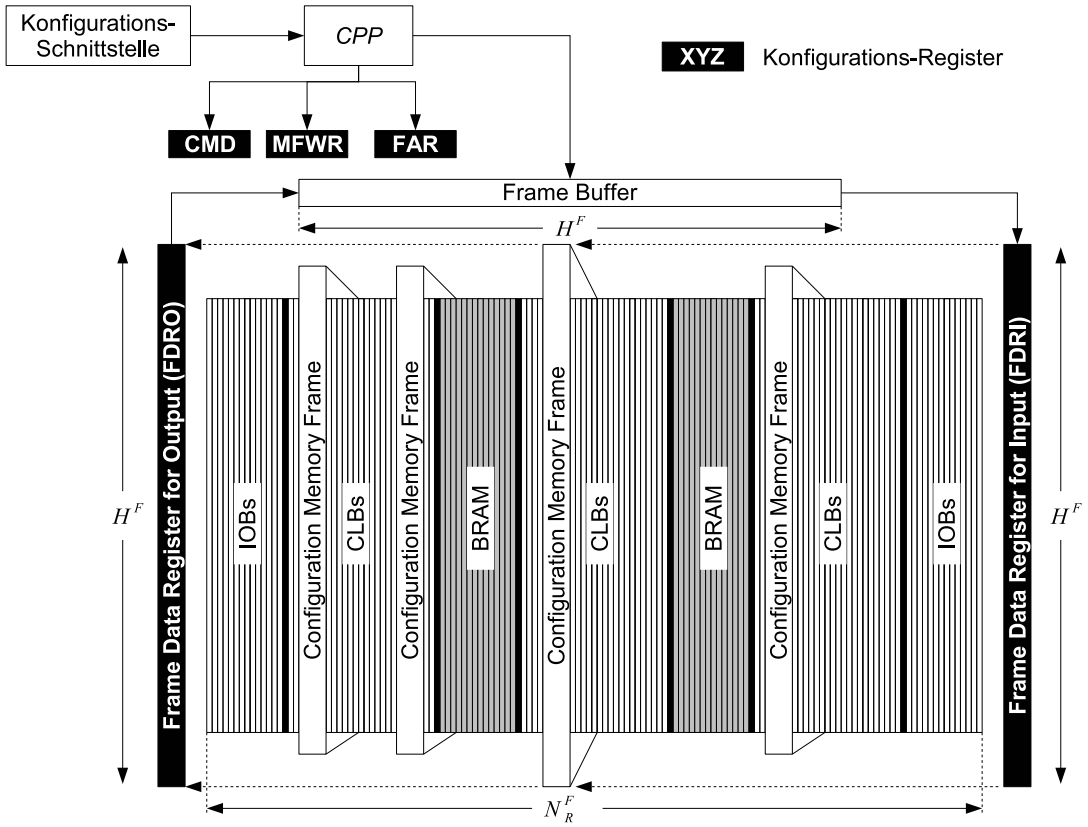


Abbildung 2.9: (Re-)konfigurations-Prozess: Bitstromdaten werden über die Konfigurations-Schnittstelle in den Frame Buffer und von dort in das FDRI geschrieben. Ist das FDRI vollständig gefüllt werden die Konfigurations-Daten in einem Taktzyklus in den Konfigurations-Layer übernommen

Der *CPP* interpretiert nach seiner Synchronisation (durch das Synchronisationswort 0xAA995566) alle ankommenden Daten. Nachdem in den Bitstromdaten ein gültiger

Paket-Header identifiziert wurde, leitet der *CPP* die darin enthaltenen Header-Daten an die entsprechenden Konfigurations-Register weiter. Im Paket-Header ist die Anzahl an 32-Bit Paket-Daten spezifiziert, die in den Frame Buffer geschrieben werden. Ist dies geschehen, wartet der *CPP* erneut auf einen gültigen Paket-Header.
Über die Konfigurations-Schnittstelle (ICAP, Selectmap, etc.) und den *CPP* werden die Paket-Daten in den Frame Buffer geladen. Um die Daten für einen Konfigurations-Frame von dort in das Frame Data Register for Input (FDRI) zu schreiben, muß das Command (CMD) Register zunächst mit dem Write Configuration (WCFG) Kommando beschrieben werden. Das FDRI hat genau die Höhe H_F eines Konfigurations-Frames. Um einen Konfigurations-Frame anschließend vom FDRI in den Konfigurations-Layer an die im FAR spezifizierte Adresse zu schreiben, stehen generell zwei Möglichkeiten zur Auswahl. Welche der beiden Möglichkeiten verwendet wird ist im Bitstrom kodiert.
Die erste Möglichkeit besteht in der Verwendung eines gepipelinten Ansatzes. Hierbei wird zunächst der Frame Buffer mit Konfigurations-Daten gefüllt. In den Frame Buffer passt wie in das FDRI genau ein Konfigurations-Frame. Die Länge des Frame Buffers beträgt somit H^F. Sobald dieser gefüllt ist, sorgt das nächste am Frame Buffer ankommende Paket an Konfigurations-Daten dafür, dass genau diese Anzahl in das FDRI geladen geschoben wird. Gepipelined werden so Daten über den Frame Buffer in das FDRI geladen. Ist das FDRI vollständig gefüllt, werden die Konfigurations-Daten parallel in einem Taktzyklus in den Konfigurations-Layer geladen. Bei dieser Methode werden die Daten aus dem FDRI also **durch den nachfolgenden Konfigurations-Frame** in den Konfigurations-Layer geladen. Um bei Verwendung dieser Methode sicherzustellen, dass der letzte Konfigurations-Frame des Bitstroms in den Konfigurations-Layer übernommen wurde, muß das FDRI abschließend (am Ende der Bitstromdaten) mit einem Pad-Frame geleert werden. Daher besteht bei dieser Methode der kleinstmögliche Bitstrom aus zwei Konfigurations-Frames, einem mit mit Nutz- und einem mit Pad-Daten. Mit dieser Methode ist es sehr einfach aufeinander folgende Frames zu schreiben, da der Eintrag im FAR automatisch inkrementiert wird. Daher ist weder ein Kommando notwendig, um die Adresse zu ändern noch eines, um die Daten aus dem FDRI in den Konfigurations-Layer zu laden. Dies geschieht, wie bereits erwähnt, automatisch durch den folgenden Konfigurations-Frame.
Die zweite Möglichkeit besteht in der Verwendung des Multiple Frame Write Registers (MFWR) Registers. Um das mehrmalige Laden ein und des selben Konfigurations-Frames in das FDRI zu vermeiden, wird das MFWR Kommando verwendet, wenn viele Konfigurations-Frames mit selbem Inhalt vorhanden sind. Um den Konfigurations-Frame mit WFWR aus dem FDRI an beliebig viele Adressen im Konfigurations-Layer zu schreiben, wird die folgende Methodik eingesetzt. Der Konfigurations-Frame wird (wie auch bei der ersten Methode) durch das WCFG Kommando im CMD Register in den Frame Buffer geladen. Die erste gewünschte Adresse wird in das FAR Register geschrieben. Wenn anschließend das MFWR Kommando in das CMD Register und zwei Pad Worte (jeweils 32 bit) an das MFWR Register geschrieben werden, wird der Inhalt des Frame Buffers kom-

plett in das FDRI übernommen und innerhalb eines Taktzyklus an die (durch das FAR) spezifizierte Adresse geschrieben. Im Folgenden wird dann wieder die Adresse im FAR geändert und erneut zwei Pad Worte an das MFWR geschrieben. Dieser Vorgang wird so lange wiederholt, bis der Konfigurations-Frame im FDRI an alle gewünschten Stellen kopiert wurde. Bei dieser Methode werden die Daten aus dem FDRI **durch Schreiben zweier Pad Worte an das MFWR Register** in den Konfigurations-Layer geladen. Die Sequenz zum Ändern der Inhalte der einzelnen Konfigurations-Register ist ebenfalls im Bitstrom kodiert. Da eine einzelne Sequenz zur Adressänderung im FAR und die beiden Pad Wörter im Normalfall wesentlich kürzer sind als ein kompletter Konfigurations-Frame, kann bei mehreren gleichen Konfigurations-Frames im Bitstrom, dessen Größe durch die Verwendung von MFWR signifikant reduziert werden. Der Konfigurations- sowie der Ausleseprozeß des Konfigurations-Layers eines Xilinx FPGAs sind unter anderem in [Xil05] beschrieben.

2.2.6 Arten der Konfiguration

Abhängig davon wie ein FPGA rekonfiguriert wird bzw. welcher Anteil eines FPGAs rekonfiguriert werden, lassen sich verschiedene Arten der Konfiguration unterscheiden. Die verschiedenen Rekonfigurationsarten sind in Abbildung 2.10 dargestellt.

	Anteil des modifizierten Konfigurations-Layers	
	Voll	Partiell
statisch (mit Reset)	Rekonfiguration	Partielle Rekonfiguration
dynamisch (ohne Reset)	n/a Multicontext FPGA	Dynamisch Partielle Rekonfiguration (DPR)

Abbildung 2.10: Verschiedene Arten der Rekonfiguration

Beim Anlegen einer Versorgungsspannung müssen SRAM basierte FPGAs mit einem initialen Bitstrom konfiguriert werden. Von einer Rekonfiguration spricht man, wenn das Device erneut mit einem vollen oder partiellen Bitstrom modifiziert wird. Bei Verwendung eines partiellen Bitstroms, um nur einen gewissen Teil des Konfigurations-Layers zu modifizieren, spricht man von partieller Rekonfiguration. Partielle Rekonfiguration setzt immer eine volle Rekonfiguration voraus, um das SRAM-basierte Device initial zu konfigurieren. Bei der statischen Rekonfiguration muß das Device im Anschluß resettet werden, um lauffähig zu sein. Der nicht rekonfigurierbare Anteil des Designs wird daher während der statischen Rekonfiguration angehalten. Eine Rekonfiguration erfordert jedoch nicht notwendigerweise das Anhalten oder Resetten des Gesamtsystems. Dyna-

mische Rekonfiguration bedeutet, dass die Rekonfiguration zur Laufzeit stattfindet und alle nicht rekonfigurierbaren Module von der Rekonfiguration ungestört weiterlaufen. In Xilinx FPGAs ist eine volle Rekonfiguration immer mit einem Reset verbunden, so dass eine volle dynamische Rekonfiguration auf Xilinx FPGAs nicht existiert. Bei einem zusätzlichen Konfigurations-Layer wie in Multi-Context FPGAss [CH00] wäre das Umschalten in einem Takt ohne Reset allerdings möglich. Die in dieser Arbeit verwendete Rekonfigurationsart ist die dynamisch partielle Rekonfiguration (DPR).
Von DPR spricht man also, wenn nur ein Teil (partiell) des Systems während der Laufzeit (dynamisch), das bedeutet ohne Reset, ausgetauscht wird. Für eine DPR muß das System in einen statischen und einen rekonfigurierbaren Teil aufgespalten werden. Der statische Teil des Systems soll dabei von der Rekonfiguration nicht beeinflußt werden. Als partiell rekonfigurierbares Modul (PRM) bezeichnet man HW Module, die sich die Ressourcen auf einem FPGA mit anderen zeitlich teilen und damit zum dynamischen (austauschbaren) Teil des Systems gehören. Die Fläche (und damit die Ressourcen), in der ein PRM auf dem FPGA platziert werden kann, wird als partiell rekonfigurierbare Region (PRR) bezeichnet. Die verschiedenen PRMs können eine unterschiedliche Anzahl von Ressourcen innerhalb einer PRR belegen. Die PRR muß aber mindestens so viele Ressourcen beinhalten, wie das größte PRM benötigt. Damit bestimmt das größte PRM die Mindestgröße der PRR. In Abbildung 2.11 ist ein System mit zwei PRRs dargestellt.

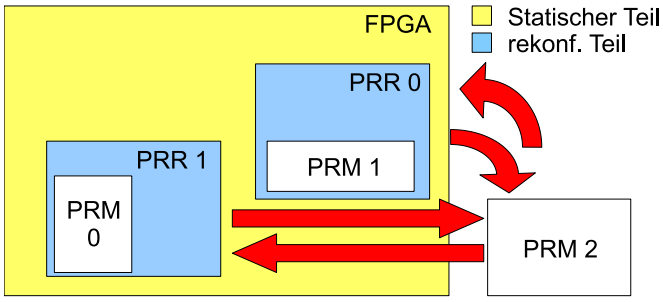


Abbildung 2.11: Elemente eines dynamisch rekonfigurierbaren Systems

Wie in Abbildung 2.11 zu erkennen ist, kann der dynamische Teil eines Systems auch aus mehreren PRRs bestehen. In dieser Arbeit enthält das rekonfigurierbare System jedoch immer eine PRR.

2.2.7 Designflows für DPR Systeme

Zur Erstellung eines rekonfigurierbaren Designs können verschiedene Designflows verwendet werden. In [Xil04] werden zwei Designflows erläutert, der Differenced-based und der Module-Based Reconfiguration Flow. Der aktuellste Designflow, der Partition-based Reconfiguration Flow, wird in [Xil10c] vorgestellt.

2.2.7.1 Difference-Based Reconfiguration Flow

Der Difference-based Reconfiguration Flow (DBRF) wird im Normalfall verwendet, wenn nur kleine Designänderungen vorgenommen werden. Diese Änderungen werden im Normalfall im FPGA Editor vorgenommen oder direkt in XDL, der Xilinx Design Language [CZH+07] programmiert. Mögliche Designänderungen können das Ändern von LUT Inhalten, Frequenzadaption innerhalb einer PLL oder das Neuverdrahten einzelner Netze sein. Bei diesem Designflow werden partielle Bitströme erzeugt, die nur die unterschiedlichen Konfigurations-Frames zwischen 2 Designs enthalten. Zur Generierung der partiellen Bitströme müssen dem Tool zur Erstellung von Bitströmen (Bitgen) zwei Files übergeben werden. Die erste Datei ist ein Bitstrom, der das initiale Design enthält. Die zweite ist eine NCD (Native Circuit Description) Datei, die das platzierte und verdrahtete Design auf dem FPGA repräsentiert und marginale Änderungen enthält. Der daraus resultierende partielle Bitstrom enthält nur die Unterscheide zwischen initialem und Referenzdesign. Die Bitstromgröße wird mit Hilfe des MFWR Kommando noch zusätzlich verkleinert. Anhhand eines Beispiels soll nun erklärt werden, wie sich partielle Bitströme zusammensetzen, die mit dem DBRF erstellt wurden.

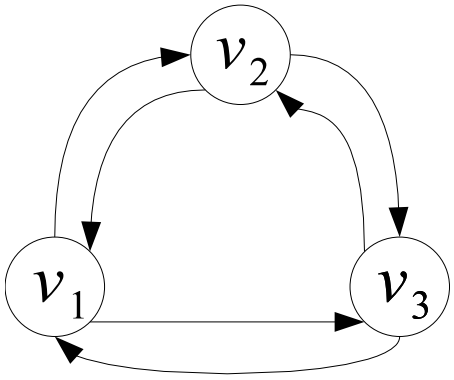


Abbildung 2.12: Rekonfiguration mit partiellen Bitströmen des DBRF: 6 verschiedene partielle Bitströme sind notwendig, um von jedem der 3 Designs (v_1, v_2, v_3) in jedes beliebige andere Design zu wechseln

Abbildung 2.12 beschreibt den Fall, wenn 3 verschiedene Module im Design ausgetauscht werden. Die Knoten v_1, v_2, v_3 des gerichteten Graphen entsprechen den 3 verschiedenen Designs. Um von einem Design zu jedem beliebigen anderen Design mit Hilfe von partiellen Bitströmen zu gelangen, die mit dem DBRF generiert wurden, sind 6 partielle Bitströme notwendig. Die Anzahl der benötigten partiellen Bitströme N_{BS} ist abhängig von der Anzahl an Toplevel Designs N_{TL} und berechnet sich nach Formel 2.6.

$$N_{BS} = {N_{TL}}^2 - N_{TL} \tag{2.6}$$

Der in Abbildung 2.12 dargestellte Fall zeigt, dass $N_{BS} = 6$ partielle Bitsröme, die mit dem DBRF generiert wurden, nötig sind, um von v_1 nach v_2, von v_2 nach v_3, von v_3 nach v_1, von v_1 nach v_3, von v_3 nach v_2 und von v_2 nach v_1 zu rekonfigurieren. Der DBRF erzeugt sehr kleine partielle Bitströme, die Konfigurations-Daten für einzelne Konfigurations-Frames enthalten. Jedoch werden bei der Verwendung des DBRF viele partielle Bitströme benötigt, um von jedem beliebigen Design in jedes beliebige andere Design zu rekonfigurieren. Für mehr als zwei PRMs ist der DBRF daher eher ungeeignet.

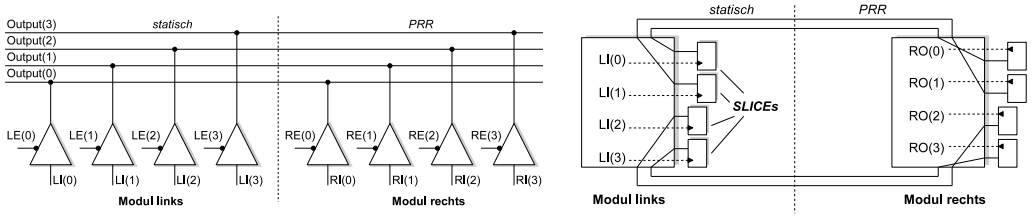
2.2.7.2 Module-Based Reconfiguration Flow

Der Module-Based Reconfiguration Flow (MBRF) wird verwendet, wenn mehr als zwei Module während der Laufzeit des Systems ausgetauscht werden sollen. Beim MBRF wird zunächst der statische Teil des Systems implementiert und anschließend das PRM. In der letzten Phase des MBRF werden der statische und der rekonfigurierbare Teil fusioniert und partielle Bitströme erzeugt. Detaillierte Informationen über den MBRF finden sich in [LBM+06]. Im MBRF werden die Konfigurations-Daten für gesamte Konfigurations-Columns (anstatt für einzelne Konfigurations-Frames wie im DBRF) in den partiellen Bitstrom geschrieben. Alle Konfigurations-Columns, die innerhalb der PRR liegen, werden somit in die partiellen Bitströme übernommen. Die mit dem MBRF generierten Bitströme sind daher im Normalfall wesentlich größer als diejenigen, die mit dem DBRF generiert wurden. Dafür wird für jedes PRM auch nur ein partieller Bitstrom erzeugt. Um die Größe der mit dem MBRF generierten partiellen Bitströme weiter zu verringern, kann ebenfalls das MFWR Kommando eingesetzt werden. Zusammenfassend kann festgestellt werden, dass der MBRF für einen Toplevel genau einen partiellen Bitström erzeugt. Dadurch, dass dieser partielle Bistrom aus Konfigurations-Columns besteht, beinhaltet er im Normalfall sehr viele redundante und damit überflüssige Daten.

2.2.7.3 Partition-Based Reconfiguration Flow

Im Partition-Based Reconfiguration Flow (PBRF) beinhalten die partiellen Bitströme ebenfalls komplette Konfigurations-Columns. Eine Partition entspricht dabei einer PRR. Auch hier kann die Größe der partiellen Bitströme durch den Einsatz von MFWR zusätzlich verkleinert werden. Zwischen dem PBRF und dem MBRF gibt es signifikante Unterschiede, die im Folgenden erläutert werden.
Ein Unterschied zwischen MBRF und PBRF besteht darin, dass die PRMs bei Letzterem, Kenntnis von der Platzierung und Verdrahtung des statischen Teils haben. Nachdem bei der Implementierung des statischen Teils die PRR ausgespart wurde, wird in einem zweiten Schritt das PRM zum bestehenden statischen Teil hinzugefügt. Damit ist im Gegensatz zum MBRF eine Gesamtoptimierung des Timings möglich ist. Beim PBRF handelt es sich also um einen inkrementellen Entwurf, der an einen existierenden statischen Teil ein rekonfigurierbares Modul anhängt. Beim MBRF hingegen werden statischer

Teil und PRM getrennt von einander implementiert.
Um ein PRM während einer DPR überhaupt an den statischen Teil anpassen zu können, müssen beide Teile pin-kompatibel sein. Das bedeutet, dass alle PRMs die selben Anschlußpunkte an den statischen Teil haben müssen. Diese Voraussetzung muß in einem DPR System gegeben sein, unabhängig davon welcher Rekonfigurationsflow (DBRF, MBRF oder PBRF) verwendet wird. Ansonsten ist es möglich, dass nach einer Rekonfiguration einige Netze zwischen statischem Teil und PRM nicht adäquat verbunden sind.
Da beim MBRF der statische und der rekonfigurierbare Teil getrennt von einander platziert und verdrahtet werden, muß eine einheitliche Schnittstelle geschaffen werden. Diese Schnittstelle wird im MBRF durch so genannte Busmacros realisiert.
Busmacros sind vordefinierte Verdrahtungsressourcen, die die Schnittstelle zwischen statischem und rekonfigurierbarem Teil festlegen. Die Macros dienen als Übergang zwischen den Signalen des statischen und rekonfigurierbaren Teils, um Pin-Kompatibilität zwischen den PRMs und dem statischen Teil zu gewährleisten. Daher müssen die Macros für alle PRMs immer an der selben Stelle platziert werden.
Die ersten Macros, die in [Xil04] vorgestellt wurden, waren Tristate Buffer (TBUF) basiert. Diese Macros nutzen die Tristate Buffer innerhalb einer CLB und die entsprechenden Verbindungen. Wie in Abbildung 2.13(a) dargestellt, kann eine CLB Reihe für den Transfer von 4 Bit verwendet werden, da 4 Long-lines [Xil08c] per CLB Reihe existieren. Da diese Art Busmacros als Verdrahtungsressource Long-lines verwendet, wird die Anzahl der möglichen Busmacros in horizontaler Richtung in einem Design durch diese Ressource begrenzt. Durch die Verwendung von Tristate Buffern ist es relativ einfach möglich die Verbindung während der Rekonfiguration zu trennen. Obwohl es theoretisch möglich wäre, diese Art Macros bidirektional zu verwenden, wird empfohlen, die Signalflußrichtung während des Betriebs nicht zu ändern.



(a) Tristate-basiert [Xil04] **(b)** LUT-basiert (L2R)
Abbildung 2.13: Verschiedene Realisierungen von Busmacros

In neueren Xilinx Devices wie V4 und V5 existieren keine Tristate Buffer mehr. Demnach müssen die Busmacros auf andere Art realisiert werden. Hübner et. al. [HBB04] verwendeten als erste LUT-basierte Busmacros. Bei der Verwendung von LUT-basierten Busmacros können horizontal 8 Bit pro CLB übertragen werden. Bei ineinander geschachtelten Busmacros sind höhere Übertragungsweiten möglich. Des weiteren lassen sich mit

Hilfe der LUTs auch sehr einfach Schalter (beispielsweise durch Konfiguration der LUTs als AND Gate) implementieren, um die Signalverbindungen bei Bedarf zu trennen.
Um Busmacros in verschiedenen Weiten nicht manuell mit Hilfe des FPGA Editors implementieren zu müssen, besteht die Möglichkeit Busmacros skriptbasiert mit Hilfe der XDL (Xilinx Definition Language) zu erzeugen [CZH+07].

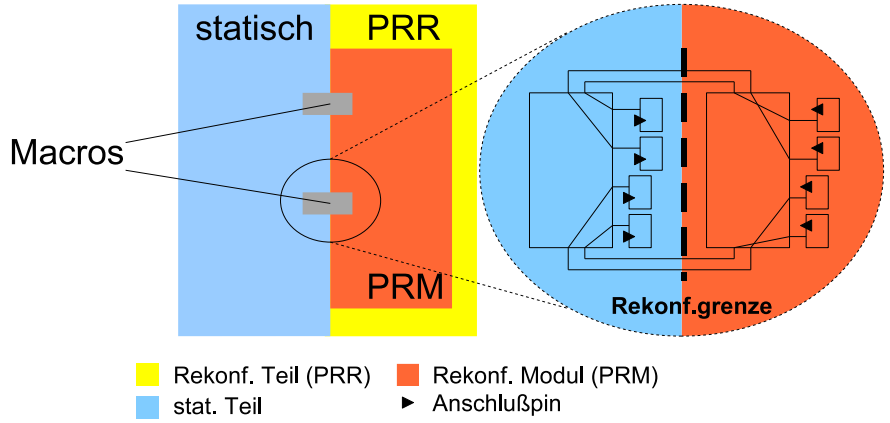


Abbildung 2.14: Platzierung von Busmacros: Busmacros gehören zum statischen und rekonfigurierbaren Teil des Designs (hier realisiert mit LUTs)

Die Busmacros sind immer unidirektional. Bei bi-direktionalen Signalen müssen daher zwei Arten von Macros für eingehende und ausgehende Daten verwendet werden. In Abbildung 2.13(b) ist ein LUT-basiertes Busmacro dargestellt, das Daten von links nach rechts (L2R) überträgt. Je nachdem, ob dieses Macro an der linken oder rechten Seite der PRR platziert wird, werden Daten aus dem statischen Teil in die PRR (links platziert) bzw. aus der PRR in den statischen Teil (rechts platziert)übertragen. Busmacros stehen als synchrone und asynchrone Version zur Verfügung. Die asynchronen Busmacros dienen als reine Verbindungsbrücke zwischen dem statischen und dem rekonfigurierbaren Teil. Sie verlängern den kombinatorischen Pfad im besten Fall (abhängig von der Weite der Macros) um etwa eine halbe Nanosekunde. In Gegensatz dazu enthalten synchrone Busmacros Register und verzögern somit die ankommenden Signale um einen Taktzyklus. Falls diese zusätzliche Pipelinestufe in Form eines zusätzlichen Registers das Gesamtsystem nicht negativ beeinflußt, führt die Verwendung von synchronen Macros generell zu einem besseren Timing. Falls möglich, ist es daher sinnvoll synchrone Macros zu verwenden.
Im PBRF sind keine Busmacros mehr notwendig. Ihre Stelle nehmen so genannte Partitionpins ein. Statt einen Anschlußpunkt im rekonfigurierbaren Modul und einen Anschlußpunkt im statischen Teil zur Verfügung zu stellen, repräsentieren Partitionpins Anschlußpunkte, die zwar zum statischen Teil gehören, jedoch innerhalb der PRR liegen. Die Partitionpins sind in Abbildung 2.15 dargestellt. Für diese Anschlußpunkte wird au-

tomatisch eine LUT innerhalb der PRR instantiiert, was bedeutet, dass keine manuelle Modifikation im Sinne des Einfügens von Busmacros mehr notwendig ist. Diese LUT gehört zum statischen Teil, so dass der Anschluß zwischen statischem und rekonfigurierbarem Teil für alle Module dieselben sind.

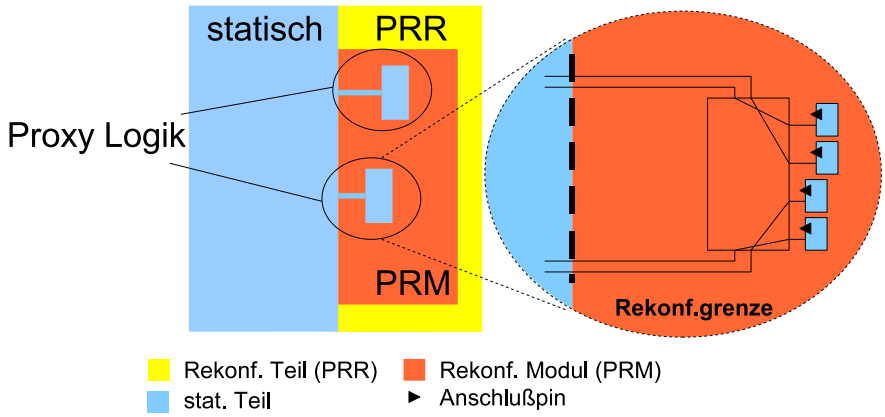


Abbildung 2.15: Partitionpins bilden den Anschluß im rekonfigurierbaren Teil. Sie gehören jedoch zum statischen Teil und werden mit LUTs realisiert.

Der Anschluß eines PRMs an den statischen Teil, funktioniert nun nicht mehr über Rekonfigurationsgrenzen hinweg, sondern innerhalb der PRR. Logik, die für solche Zwecke dem Design hinzugefügt wird, wird als Proxy Logik [Xil10c] bezeichnet. Allerdings ist der PBRF nur für Virtex-Devices der Generation 4 und neuer (V4, V5, V6) verfügbar. Die V2, Spartan-3 und Spartan-6 Familen werden nicht unterstützt.

2.3 System on Chip (SoC)

Unter einem System on Chip (SoC) versteht man die Integration kompletter HW/SW Systeme auf einem einzigen Chip. In einem SoC wird versucht, wesentliche Teile des Systems aus bereits existierenden Funktionsblöcken zusammen zu setzen. Bedingung hierfür ist das modul-basierte Design des SoCs. Charakteristische Module sind dabei verschiedene Prozessoren, Speicher sowie deren Controller und anwendungsspezifische Peripherie-Einheiten (UART, Ethernet oder Video Controller etc.), die sich über verschiedenste Verbindungsmedien (Point-to-Point, Busse, Networks-on-Chip(NoC), Crossbars) mit einander kombinieren lassen. Auch die verschiedenen Verbindungsmedien stehen bei einem SoC als Module zur Verfügung.
Um die in einigen SoCs geforderten Echtzeitbedingungen zu erreichen, können dem System anwendungsspezifische Hardware Acceleratoren (HWAs) hinzugefügt werden. Ein Hardware Accelerator ist ein Coprozessor, der performanzlastige Berechnungen eines Prozessors (Central Processing Unit (CPU)) verarbeitet und im besten Fall beschleu-

nigt, was zu einer Entlastung der CPU führt. Die HWAs werden ebenfalls als Module in ein SoC eingebunden. Die einzelnen Intellectual Property (IP) Module lassen sich hierarchisch weiter unterteilen, beispielsweise in ein IP Interface (IPIF), das für den Anschluß an Bus, NoC, Crossbar, etc. verwendet wird und den Funktionsblock (IP Core), der die eigentliche Operation ausführt. Durch Festlegung geeigneter Schnittstellen, lassen sich IP Cores somit durch modulbasiertes Design ohne Modifikation an verschiedenste Verbindungsmedien anschließen. Durch das modulbasierte Design innerhalb eines SoCs, muß oft nur der anwendungs- bzw. kundenspezifische Teil neu entwickelt werden, was zu kürzeren Entwurfszyklen und geringeren monetären Kosten führt.
Um die Kommunikation zwischen verschiedenen Modulen mit einer oder mehreren CPUs in einem SoC zu realisieren, bieten Firmen wie IBM[3] mit CoreConnect oder ARM[4] mit der Advanced Microcontroller Bus Architecture (AMBA) verschiedene Busse an. Da in dieser Arbeit das CoreConnect System von IBM verwendet wurde, wird dieses im folgenden Abschnitt vorgestellt und auf AMBA nicht weiter eingegangen. Vergleiche zwischen CoreConnect, AMBA und weiteren on-chip Bussystemen, wie Altera's Avalon oder dem open source Wishbone sind in [MSS08, Pel03] dargestellt.
In integrierten Schaltkreisen und damit auch SoCs wird versucht hinsichtlich dreier Zielkriterien zu optimieren. Die logischen Ressourcen (also die Fläche) und die Verlustleistung sollen dabei minimiert, der Durchsatz in jeder Form gleichzeitig maximiert werden. Folgend werden daher zunächst die Komponenten eines SoCs vorgestellt, die gewisse Ressourcen auf dem Chip belegen. Im Anschluß werden Möglichkeiten zur Erhöhung des Durchsatzes in integrierten Systemen erläutert. Das Ende dieses Abschnitts bildet die Betrachtung einzelner Ströme, die zur Verlustleistung des Gesamtsystems beitragen und daher minimiert werden sollen.

2.3.1 IBM CoreConnect

Das CoreConnect System von IBM ist eine Busarchitektur für SoC-Designs. Es wurde entwickelt, um die Integration und Wiederverwendbarkeit von verschiedenen Intellectual Property (IP) cores (z.B. Prozessoren) in SoC-Designs zu erleichtern. Xilinx nutzt CoreConnect für alle seine eingebetteten Prozessor Designs, obwohl nur die wenigsten davon IBM's PowerPC Prozessoren enthalten. Die CoreConnect Architektur beinhaltet den Processor Local Bus (PLB), den On-chip Peripheral Bus (OPB), den Device Control Register (DCR) Bus und verschiedene Busbrücken, um beispielsweise Daten vom PLB auf den OPB oder umgekehrt zu übertragen. Die Busse selbst sind wiederum als IP Cores realisiert und werden zusammen mit ihren Einsatzgebieten in den folgenden Abschnitten vorgestellt. In Abbildung 2.16 ist ein Multi-Bus SoC dargestellt, in dem PLB, OPB und DCR verwendet werden.

[3] IBM: International Business Machines Corp.
[4] ARM: Advanced RISC Machines Ltd.

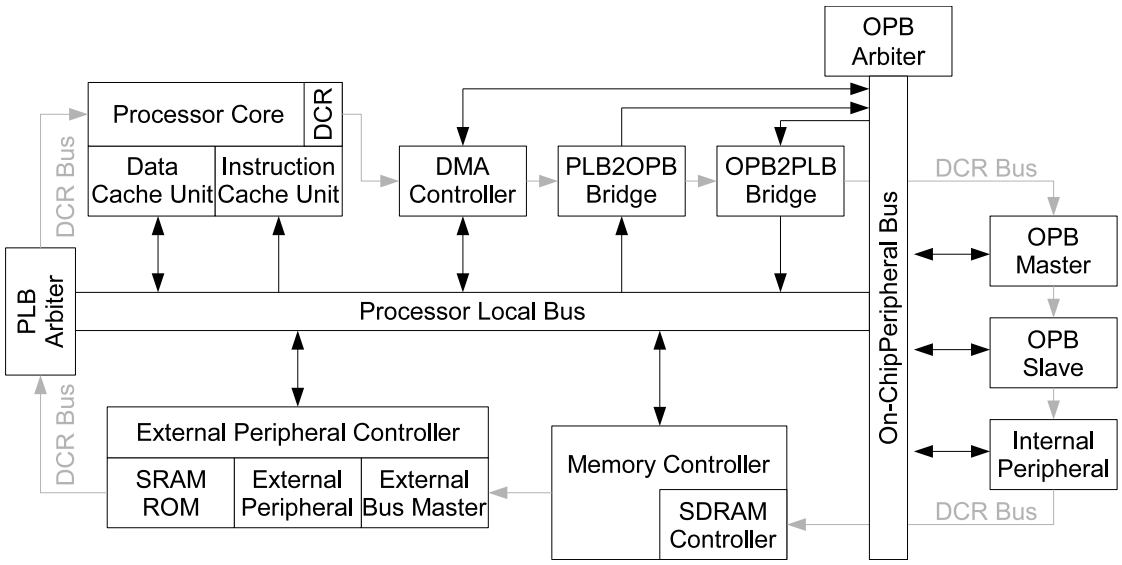


Abbildung 2.16: Multi-Bus SoC mit PLB, OBP und DCR [IBM07]

2.3.1.1 Processor Local Bus (PLB)

Der Prozessor Local Bus (PLB) ist für die Übertragung hoher Datenraten gedacht, um beispielsweise eine oder mehrere CPUs an externe Speicher oder Hardware Acceleratoren anzubinden. Um die Daten auf der gemeinsamen Ressource Bus zwischen den verschiedenen IP Cores übertragen zu können, existiert eine hierarchische Zugriffsverwaltung. Diese besteht aus der Anbindung von IP Cores an den Bus als Master (kann selbstständig Transfers initiieren) die verschiedene Slave IP Cores (können eigenständig keine Transfers initiieren) kontrollieren. Der PLB erlaubt die Anbindung mehrerer Master und Slave IP Cores. Da in Multi-Master Systemen mehrere Komponenten den Bus für sich beanspruchen können, wird ein Busarbiter verwendet, um den Zugriff zu regeln und Zugriffskonflikte aufzulösen. Die Übertragung hoher Datenraten auf dem PLB wird unter anderem durch einen 32-bit Adressbus einen Lese- und einen Schreib-Datenbus ermöglicht. Durch die entkoppelte Realisierung der 3 Busse, können gleichzeitig Daten gelesen und geschrieben werden. Lese- und Schreibdatenbus können auf eine Datenweite D_W von 32, 64 oder 128 Bit konfiguriert werden. Möglichkeiten zur Erhöhung des Durchsatzes auf dem PLB werden in Abschnitt 2.3.2 vorgestellt. Weitere Informationen über den PLB sind in [IBM07, Xil09c] zu finden.

2.3.1.2 On-chip Peripheral Bus (PLB)

Der On-chip Peripheral Bus ist ein synchroner Bus, der hauptsächlich für die Anbindung langsamer Peripherie Controller mit geringeren Bandbreiten-Anforderungen verwendet

wird, um Performanzeinbußen auf dem PLB zu vermeiden. Oft werden Busbrücken verwendet, um Daten vom OPB auf den PLB und umgekehrt zu übertragen. Der OPB verfügt über einen vom Datenbus getrennten Adressbus, der bis auf 64 Bit erweitert werden kann. Getrennte Schreib- und Lese-Datenbusse existieren beim OPB nicht. Allerdings unterstützt der OPB auch mehrere Master durch einen eigenen Arbiter. In [Xil10b, IBM01] sind weitere Informationen über den OPB zusammen gestellt.

2.3.1.3 Device Control Register Bus (DCR)

Kontrolldaten, die den Inhalt der Konfigurationsregister einzelner IP Cores ändern, können ebenfalls über PLB oder OPB übertragen werden. Eine Alternative bietet der Device Control Register Bus (DCR), um Kontroll- und Nutzdaten strikt von einander getrennt zu halten. Der DCR Bus unterstützt einen Master und bis zu 16 Slaves, die in einer Daisy Chain[5] zusammengeschaltet sind. Theoretisch kann alle zwei Taktzyklen ein Lese- bzw. Schreibtransfer stattfinden. Der DCR Bus besteht aus einem 10-Bit Adress- und einem 32-Bit Datenbus [Xil10a].

2.3.2 Standard Bus Transfers auf dem PLB

Eine Transaktion auf dem Processor Local Bus (PLB) besteht aus einem Adress- und einem Daten-Zyklus. Der Adress-Zyklus untergliedert sich in einen Request (Anfrage), einen Adress-Transfer und ein Adress-Acknowledgement (Bestätigung). Ein typischer Datentransfer auf dem PLB Bus beginnt mit einer Anfrage (Request) eines Masters für einen Datentransfer an den Arbiter. Der Arbiter regelt Konflikte beim Zugriff auf den PLB. Gleichzeitig legt der Master die Adresse von der gelesen bzw. an die geschrieben werden soll an den Adressbus an. Dies geschieht zusammen mit den „Transfer Qualifiers (TQs)“. Die TQs legen die Art der Transfers (Lese- oder Schreib-Operation) und dessen Länge fest. Im Normalfall benötigt die Arbitration eine gewisse Zeit $t_{Arbitration}$. Nach der Arbitration legt der PLB dann das Primary Address Valid (PLB_PAValid) Signal an, um eine gültige primäre Adresse und TQs anzuzeigen. Gleichzeitig werden die Adresse und die TQs an den Slave während der Transfer-Phase (Address-Transfer) übertragen. Der Slave bestätigt Adresse und TQs (Address-Acknowledgement) mit dem Sl_AddrAck Signal [IBM07]. Der PLB hat eine 3-Zyklen Arbitration [Xil09c], wie in Abbildung 2.17 dargestellt ist. Zwischen Request und PLB_PAValid vergehen zwei Zyklen.
Ein Daten-Zyklus hingegen besteht aus zwei Phasen. Der Daten-Transfer- und der Daten-Acknowledgement-Phase. Während der Daten-Transfer-Phase legt der Master Daten auf den Schreibbus (M_wrDBus), wenn es sich um einen Schreibtransfer handelt, oder er bekommt Daten über den Lesebus (M_rdDBus) bei einem Lese-Transfer. Gültige Daten

[5] Unter einer Daisy Chain versteht man eine Anzahl von Komponenten, die in Serie geschaltet miteinander verbunden sind. In diesem Fall ist der erste DCR Slave direkt mit dem Master, die anderen Slaves jeweils mit ihren Vorgängern verbunden.

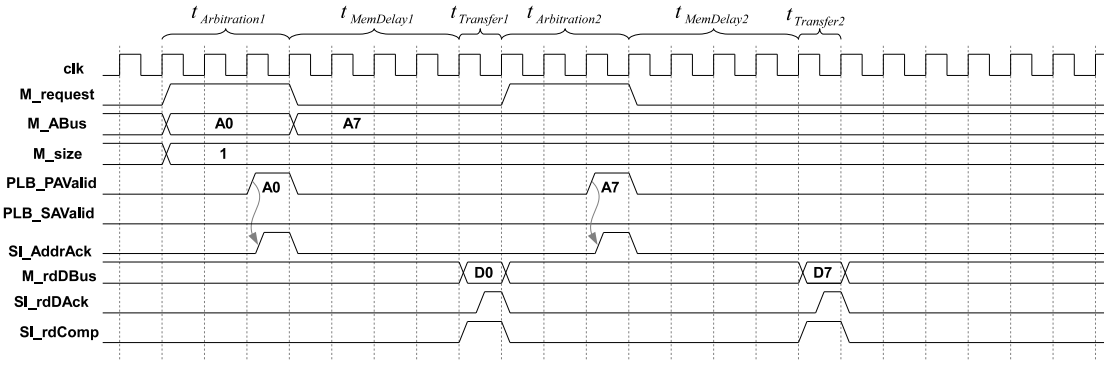


Abbildung 2.17: Timingdiagramm Einzelwort-Transfer [IBM07, Zep07]

auf dem PLB Bus werden bei einem Lese-Transfer durch das Daten-Acknowledgement-Signal Sl_rdDAck bzw. durch Sl_wrDAck bei einem Schreib-Transfer gekennzeichnet. Ist der Slave beispielsweise ein Speichercontroller, ist die Anzahl an Taktzyklen zwischen Request und Sl_rdDAck (genauer gesagt die Latenz zwischen Adress-Acknowldgement Sl_AddrAck und Daten-Acknowldgement Sl_rdDAck) abhängig von der Art Speicher, die im System verwendet wird. Abhängig von der verwendeten Speicherart, ist diese Zeit, die im folgenden mit $t_{MemDelay}$ bezeichnet wird, kürzer oder länger. Ist der Slave beispielsweise ein (DDR/DDR2) SDRAM[6] Controller, kommen zusätzliche Wartezyklen hinzu, um die richtige Adresse im Speicher zu selektieren [UW07]. Die gewünschte Position im Speicher wird dabei in eine Zeilen- und eine Spaltenadresse unterteilt und über dieselben Leitungen im Multiplexbetrieb übertragen. Bei Verwendung eines SRAM ist diese Zeit wesentlich kürzer. Die Zeit, um dann tatsächlich die Daten zu übertragen wird mit $t_{Transfer}$ bezeichnet. Die Gesamtdauer eines Transfers in Taktzyklen auf dem PLB lässt sich mit Gleichung 2.7 bestimmen.

$$t_{gesamt} = t_{Arbitration} + t_{MemDelay} + t_{Transfer} \tag{2.7}$$

Während der Transferzeit $t_{Transfer}$ ist der Bus so lange blockiert, bis der entsprechende Slave die Daten zusammen mit dem Bestätigungssignal Sl_DataAck liefert. Abhängig vom verwendeten Speicher und zugehörigem Controller reicht die Verzögerungszeit $t_{MemDelay}$ bis der Slave antwortet, von etwa zwei Zyklen (schnelles SRAM) bis zu 15 oder mehr Zyklen (DRAM mit der Aktivierung der richtigen Zeilen und Spalten) [UW07]. Die langen Aktivierungszeiten und zusätzlichen Arbitrationszyklen sorgen für eine Busauslastung, die wesentlich geringer ist, als die theoretisch Mögliche (Peak Bandbreite). Bei Verwendung von Standard Bus Transfers kann mit einem neuen Transfer erst begonnen werden, wenn der vorherige komplett abgeschlossen wurde. In vielen eingebetteten Sys-

[6]SDRAM: Synchronous Dynamic Random Access Memory

temen ist der zentrale Bus daher der Flaschenhals. Ein Beispiel in Abbildung 2.17 soll die Problematik verdeutlichen.
Ein Datenbus habe eine Weite D_W von 64 Bit und sei mit einem Takt von 100 MHz versorgt. Daraus ergibt sich ein theoretischer Datendurchsatz von 64 Bit * 100 MHz = 800 MByte/s, wenn jeden Takt Daten übertragen werden. Da der PLB separate Schreib- und Lesebusse verwendet, sind bei der genannten Konfiguration theoretisch 1600 MByte/s möglich (jeweils 800 MByte für Schreiben und Lesen). Werden allerdings für die Arbitration $t_{arbitration}$ 3 Taktzyklen und für das Bereitstellen der Daten aus dem Speicher $t_{MemDelay}$ 10 Taktzyklen angenommen, dauert ein Transfer eines 64-bit Datenpakets $t_{transfer}$ (für das 1 Taktzyklus angenommen wird) nach Gleichung 2.7 insgesamt 14 Taktzyklen. Um 8 Datenpakete zu übertragen, sind demnach 8*14 Takte = 112 Takte notwendig. Der mögliche Durchsatz verkleinert sich damit um den Faktor 14 auf (800/14) MByte/s = 57,14 MByte/s bei diesem Einzelwort-Transfer. Für die Erhöhung des Durchsatzes werden im Folgenden 2 Möglichkeiten vorgestellt: Die Verwendung von Burst-Transfers und Address-Pipelining.

2.3.2.1 Burst

Als einen Burst von Daten bezeichnet man in der Informationstechnologie eine bestimmte Menge an Daten, die in einer ununterbrochenen Operation gesendet oder empfangen werden. Die Menge an Daten, die durch einen Burst der Weite B_W übertragen wird, ist entweder bereits vor dem Transfer bekannt (Bursts fixer Länge), oder der Transfer wird durch den Master oder den Slave beendet (Bursts variabler Länge) [IBM07]. Da in dieser Arbeit ausschließlich Bursts fixer Länge mit einer vorher definierten Weite B_W verwendet werden, sind die Bursts variabler Länge nicht weiter erläutert. Die Burstweite B_W wird beim PLB über die Buskontroll Logik kontrolliert. In einem Burst werden in einem einzigen Transfer anstatt Einzelworten, B_W Worte übertragen. Bedingung für Burst-Transfers sind zusammenhängende Daten im Speicher. Um einen Burst-Transfer zu starten wird ein Request mit einer Startadresse abgesetzt. Die Startadresse gibt die Position im Speicher an, von der gelesen, bzw. an die geschrieben werden soll. Wurde beispielsweise ein Wort von der Startadresse gelesen, ist die entsprechende Zeile im (SDRAM) Speicher noch aktiviert. Um auf die aufeinander folgenden Elemente zuzugreifen, müssen die Spaltenadressen inkrementiert werden, was durch einen internen Zähler erledigt wird. Daten können somit sehr effektiv aus zusammenhängenden Speicherbereichen (innerhalb einer Speicherzeile) ausgelesen werden. Für den Schreibvorgang funktioniert dies analog. Bei Annahme der in Abschnitt 2.3.2 genannten Werte, können im Gegensatz zu einem Einzelwort-Transfer 8 Datenpakete bei einer Burstweite von B_W=8 in $t_{Arbitration} + t_{MemDelay}$ + 8 Taktzyklen = 21 Taktzyklen übertragen werden. Daraus ergibt sich in diesem Fall ein möglicher Durchsatz von (800/21/8) MByte/s = 304,76 MByte/s. Zwei Lese-Burst-Tranfers mit jeweils einer Weite von B_W=4 auf dem PLB sind in Abbildung 2.18 dargestellt.

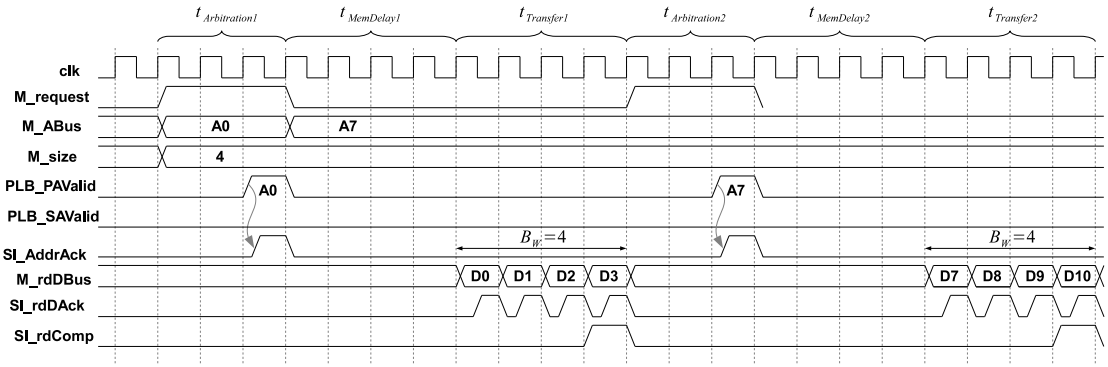


Abbildung 2.18: Timingdiagramm Burst-Transfer [IBM07, Zep07]

Bei der Verwendung von Einzelwort-Transfers, wie auch bei der Übertragung mit Hilfe von Bursts besteht das Problem, dass ein neuer Transfer erst gestartet werden kann, wenn der vorherige Transfer komplett abgeschlossen ist.

2.3.2.2 Pipelining

Eine weitere Möglichkeit den Durchsatz zu erhöhen besteht im so genannten Address-Pipelining. Im Gegensatz zu normalen Einzelwort- und Burst-Transfers, bei denen Adress- und Datenzyklus rein sequentiell ablaufen, überlappen sich Adress- und Datenzyklen beim Address-Pipelining. Damit können also neue Daten angefordert werden, während der vorherige Transfer noch andauert. Zur Nutzung von Adresspipelining müssen Master sowie Slave diese Funktionalität unterstützen. Ein Ablauf bei der Verwendung von Address-Pipelining ist in Abbildung 2.19 dargestellt.
Nachdem der Empfang der ersten gültigen Adresse (A0) durch den Slave bestätigt wurde (Sl_AddrAck), kann nach der Zeit $t_{MemDelay1}$ mit der Übertragung eines Bursts mit $B_W = 4$ (festgelegt durch das Signal M_size) begonnen werden. Während des ersten Transfers wird ein zweiter Transfer gestartet, der 4 Datenpakete ab Adresse A7 lesen soll. Diese zweite gültige Adresse (Secondary Address) wird durch das Signal PLB_SAValid Signal angezeigt und der Erhalt wird umgehend durch den Slave bestätigt. Noch bevor die Daten des ersten Transfers am Lesebus (M_rdDBus) anliegen, wird ein dritter Lese-Transfer von Adresse A14 gestartet. A14 wird jedoch erst als gültig markiert und durch den Slave bestätigt, wenn der erste Transfer abgeschlossen ist (Sl_rdComp). In Abbildung 2.19 ist zu erkennen, dass durch die Überlappung der Buszyklen direkt nach Beendigung des ersten Transfers, die Daten des zweiten Transfers direkt übertragen werden können. Dadurch, dass der Bus durch den ersten Transfer noch blockiert ist, erhöht sich die Zeit $t_{Transfer2}$ von 4 auf 5 Taktzyklen. Direkt im Anschluß an den zweiten Transfer, können dann die Daten des dritten Transfers über den Bus übertragen werden. Abzüglich der Taktzyklen, die durch $t_{Arbitration1}$ und $t_{MemDelay1}$ verursacht werden, können mit Hilfe

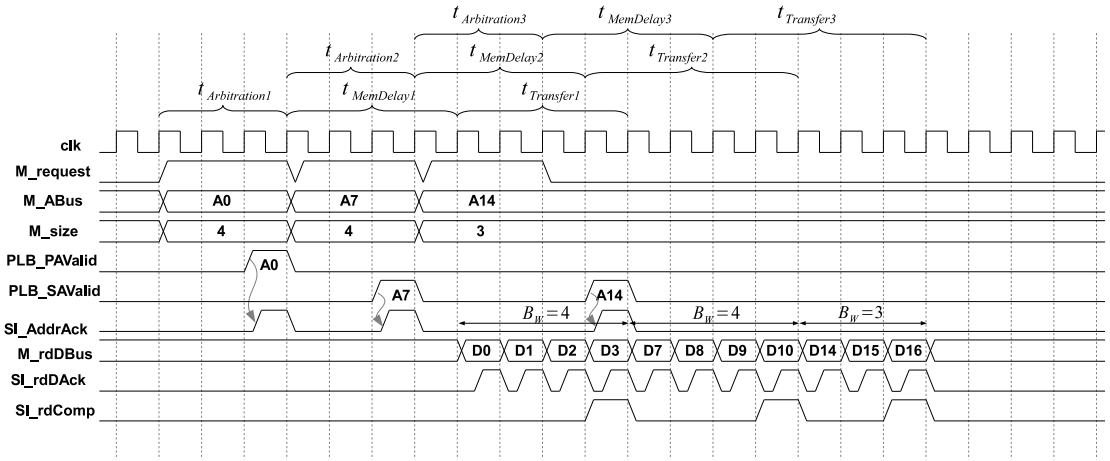


Abbildung 2.19: Timingdiagramm Address-Pipelining [IBM07, Zep07]

von Address-Pipelining jeden Takt Daten übertragen werden. Dies gilt jedoch nur, wenn B_W größer als die Anzahl an Takten in $t_{Arbitration}$ ist. Eine detaillierte Darstellung von Address-Pipelining, sowie dessen Realisierung ist in [Fli05] beschrieben.

2.3.3 Übergänge zwischen verschiedenen Taktdomänen

In vielen integrierten Systemen werden verschiedene Taktdomänen für die vielfältigsten Aufgaben verwendet. In diesen Multi-Taktraten Systemen kommt es immer wieder vor, dass Daten von einer Taktdomäne in eine andere überführt werden müssen. Hier kommt es oft zu metastabilen Zuständen. Um diese Aufgabe sicher auszuführen, werden asynchrone FIFOs (First-In-First-Out) verwendet. Die Pointer zur Anzeige der aktuellen Schreib- bzw. Leseposition sind unabhängig und verwenden unterschiedliche Takte. Richtlinien zum Design von synthetisierbaren und korrekt funktionierenden asynchronen FIFOs auf V2P, sowie Probleme und deren Lösungen sind in [CA02] beschrieben. In V5 Devices existiert ein FIFO Controller in jedem BRAM, um synchrone und asynchrone FIFOs bis zu einer Taktfrequenz von 500 MHz ohne zusätzlichen Logikaufwand zu realisieren [Alf08].

2.3.4 Verlustleistung

In diesem Abschnitt werden die Hauptursachen der Verlustleistung in einem integrierten System vorgestellt. Die Gesamt-Verlustleistung P_{total} in digitalen CMOS Schaltungen setzt sich aus einem statischen und einem dynamischen Anteil zusammen.
Die dynamische Verlustleistung P_{dyn}, die durch Umschaltvorgänge in einer Schaltung entsteht, setzt sich wiederum aus zwei Teilen zusammen. Zum einen aus der Verlustleistung P_{cap}, die durch das Umladen von Kapazitäten innerhalb einer Schaltung entsteht

und zum anderen aus der Verlustleistung P_{short}, die durch den Kurzschlußstrom während des Umschaltvorgangs verursacht wird. Die Ursachen der dynamischen Verlustleistung sollen am Beispiel eines CMOS Inverters in Abbildung 2.20 veranschaulicht werden.

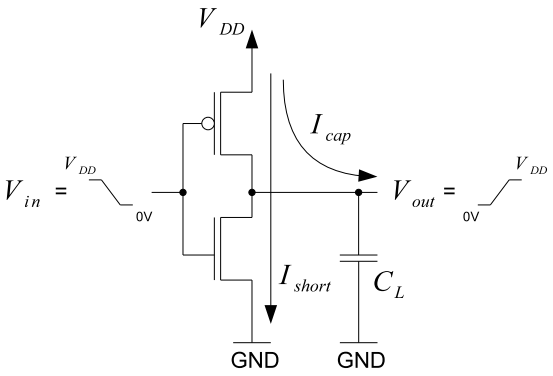


Abbildung 2.20: Quellen der dynamischen Verlustleistung am Beispiel eines CMOS Inverters [Alt02]

Ein CMOS Inverter besteht aus einem NMOS und einem PMOS Transistor. Findet beispielsweise am Eingang V_{in} ein Signalwechsel von *high* nach *low* statt, sind beide Transistoren während des Umschaltvorgangs kurzfristig leitend. In diesem Moment fließt ein Kurzschlußstrom I_{short} von V_{DD} nach GND, wie in Abbildung 2.20 zu erkennen ist. Der Kurzschlußstrom fließt, wenn $V_{DD} - V_{Tp} \geq V_{in} \geq V_{Tn}$ [Alt02] ist, wobei V_{DD} die Versorgungsspannung, V_{Tn} bzw. V_{Tp} die Schwellwertspannung des NMOS bzw. PMOS Transistors und V_{in} die Spannung am Eingang des Inverters ist. Die Verlustleistung P_{short} durch Kurzschlußströme wird mit Gleichung 2.8 berechnet.

$$P_{short} = \frac{\beta}{12}(V_{DD} - 2V_{tn})^3 \tau f \tag{2.8}$$

f ist dabei die Taktfrequenz der Schaltung und τ die Anstiegszeit des Eingangssignals. β ist von den Transistorparametern (Kanallänge und -weite) abhängig.
Die Verlustleistung P_{cap}, die durch das Umladen von Kapazitäten am Ausgang entsteht (s. Abbildung 2.20), wird durch den Strom I_{cap} verursacht, der beim Aufladen bzw. Entladen einer Lastkapazität C_L über den PMOS bzw. NMOS Transistor auftritt. P_{cap} kann mit Hilfe von Gleichung 2.9 berechnet werden.

$$P_{cap} = C_L f V_{DD}^2 \tag{2.9}$$

Die Herleitung von Gleichung 2.8 und Gleichung 2.9 ist in [RCN03, Ste09] beschrieben. Die beiden Anteile P_{cap} und P_{short} sind jeweils proportional zur Taktfrequenz f der Schal-

tung. Aus den Anteilen P_{cap} und P_{short} setzt sich die gesamte dynamische Verlustleistung P_{dyn} zusammen, wie in Gleichung 2.10 zu erkennen ist.

$$P_{dyn} = P_{short} + P_{cap} \tag{2.10}$$

Durch die Realisierung mit NMOS und dazu komplementären PMOS Transistoren existiert in CMOS Schaltungen kein permanenter Gleichstrompfad von V_{DD} nach GND [Sch00]. Verantwortlich für die statische Verlustleistung in CMOS Schaltungen sind rein die auftretenden Leckströme. Statische Verlustleistung P_{stat} bedingt durch Leckströme I_{stat} existiert auch, wenn kein Takt an der Schaltung angelegt ist. Die Ursachen für die auftretenden Leckströme sind im Folgenden aufgeführt. Ein NMOS Transistor leitet, wenn die Gate-Source Spannung größer als die Schwellspannung ($V_{GS} > V_{Tn}$) ist. Bei einem ausgeschalteten Transistor ($V_{GS} < V_{Tn}$) fließt idealerweise kein Strom ($I_{DS} = 0$A). Tatsächlich existiert jedoch ein subthreshold Leckstrom I_{sub}, der im Kanal bei sperrenden Transistoren durch Diffusions- und Emmisionseffekte entsteht [Sil07].
Der Gate (Oxide) Leakage Strom I_g ist ebenfalls ein dominanter Anteil. Er entsteht, wenn Elektronen oder Löcher aus dem Gate durch das Gate-Oxid ins Substrat und umgekehrt tunneln. Bei einem sehr dünnem Gate Oxid (wie in der 65 nm Technologie in der der V5 gefertigt wird) können Elektronen bzw. Löcher sehr einfach durch das Gate-Oxid tunneln. I_g kann in die Komponenten Gate-to-Channel I_{gc}, Gate-to-Bulk I_{gb} und Gate-to-Source/Drain I_{gs}/I_{gd} unterteilt werden.
Einen weiteren großen Einfluß auf die statische Verlustleistung in 65 nm und kleineren Technologien hat der so genannte Band-to-Band Tunneling Strom I_{BTBT}, der auf dem Band-to-Band Tunneling [Sil07, Ras07] basiert. I_{BTBT} repräsentiert den Leckstrom zwischen Source bzw. Drain und dem Substrat.
Einen weiteren Beitrag zur statischen Verlustleistung liefert der Gate Induced Drain Leakage Current I_{GIDL} oder Tunnelleckstrom [Zau09]. Da dieser zusammen mit dem Punchthrough Leakge Strom [Sil07] im Vergleich zu den anderen Leckströmen nur einen sehr geringen Anteil am Gesamtverbrauch hat [Ras07, RMMM03], werden diese beiden Beiträge nicht weiter betrachtet. In Abbildung 2.21 sind die 6 stationären Zustände eines NMOS Transistors zusammen mit den dominanten Leckströmen I_{sub}, I_g und I_{BTBT} dargestellt. Aus Gründen der Vollständigkeit sind die einzelnen Leckströme, die einen Beitrag zur statischen Verlustleistung liefern nochmals dargestellt. Bei den beiden Fällen an denen am Gate '1' anliegt (der NMOS Transistor also leitet) und Source und Drain unterschiedliche Werte aufweisen, handelt es sich um nicht stationäre Zustände.
Wie in Abbildung 2.21 zu erkennen ist, tritt nur in Fall A) keiner der drei betrachteten Leckströme auf. I_{sub} hingegen tritt nur in den Fällen B) und C) auf. I_{BTBT} tritt verstärkt in den Fällen E) und F) auf und I_g dominiert in Fall D). In [RCK07] wurden die auftretenden Leckströme der in Abbildung 2.21 dargestellten Beschaltungen eines NMOS Transistors simuliert. In Tabelle 2.2 sind diese Ergebnisse dargestellt.
Wie in Tabelle 2.2 zu erkennen ist, resultieren verschiedene Beschaltungen von Gate,

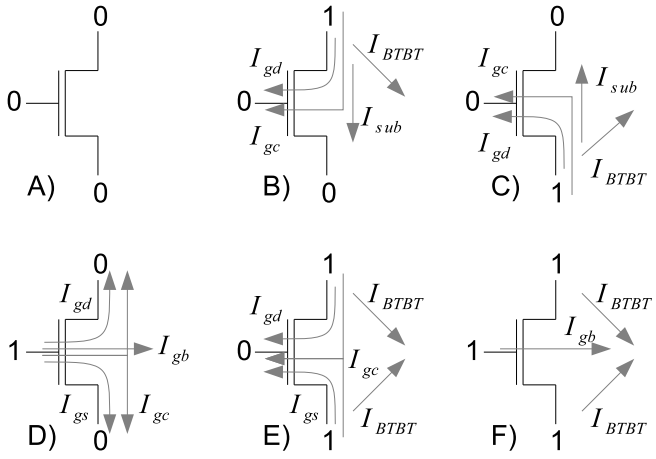


Abbildung 2.21: Leckströme bei verschiedener Beschaltung eines NMOS Transistors [RCK07].

Beschaltung [G][D][S]	I_{sub} nA/μm	I_g nA/μm	I_{BTBT} nA/μm
A) [0][0][0]	0	0	0
B) [0][1][0]	31.93	-8.83	54.6
C) [0][0][1]	-31.93	-8.83	54.6
D) [1][0][0]	0	25.27	0
E) [0][1][1]	0	-17.65	109.3
F) [1][1][1]	0	1.6E-7	109.3

Tabelle 2.2: Abgeschätzte Werte für auftretende Leckströme eines NMOS Transistors in 65nm Technologie [Ras07].

Source und Drain eines Transistors in unterschiedlichen Leckströmen. Zu erkennen ist, dass in den Fällen A) und D), in denen an Source und Drain jeweils '0' anliegt, die niedrigsten Leckströme erreicht werden können. Ergebnisse, die dieselbe Tendenz zeigen sind in [RCK07] dargestellt.
Die frequenzunabhängigen Leckströme I_{sub}, I_g und I_{BTBT} tragen somit zur statischen Verlustleistung P_{stat} bei. Gesamt-Verlustleistung P_{total} lässt sich mit Gleichung 2.11 berechnen.

$$P_{total} = P_{dyn} + P_{stat} = P_{short} + P_{cap} + P_{stat} \tag{2.11}$$

Während die statische Verlustleistung in 130 nm Technologien gegenüber der dynamischen Verlustleistung noch relativ gering war, beginnt sich dieses Verhältnis ab einer

Strukturgröße von etwa 65 nm umzukehren. Abbildung 2.22 zeigt die zunehmende Bedeutung an statischer Verlustleistung abhängig von der Strukturgröße der Transistoren.

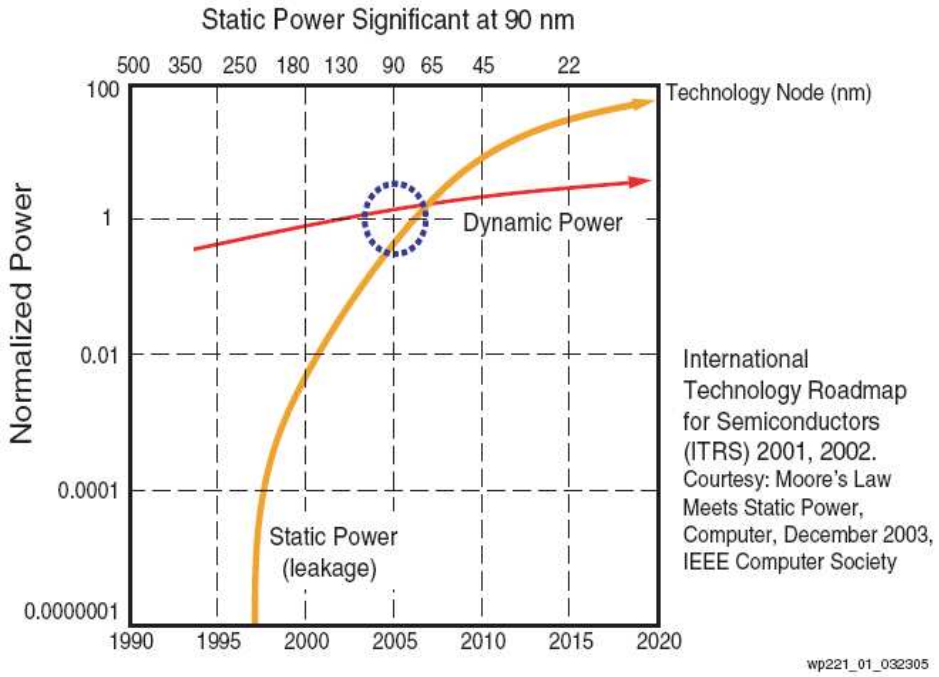


Abbildung 2.22: Statische und dynamische Verlustleistung abhängig von der Strukturgröße [Kle05, KAB+03]. Der Verlauf basiert auf der International Technology Roadmap for Semiconductors (ITRS) von 2001 und 2002.

Die Betrachtung der statischen Verlustleistung in V5 FPGAs (65 nm Strukturgröße) ist daher genau so wichtig, wie die Analyse der dynamischen Verlustleistung. Da der Einfluß statischer Verlustleistung bei zukünftigen kleineren Strukturgrößen immer größer wird, ist in dieser Arbeit auf die Reduktion dieses Verlustleistungsanteils ein besonderes Augenmerk gelegt.

3 Stand der Technik

3.1 Bildverarbeitungsprozessoren

Für die Bildverarbeitung im Automobil sind in den letzten Jahren einige Prozessoren vorgestellt worden. Sie repräsentieren den aktuellen Stand der Technik und werden im Folgenden vorgestellt.

3.1.1 Der IMAPCAR Prozessor

NEC entwickelte den IMAPCAR[1] Chip [KO08, SKO07], der in Echtzeit Objekte wie Fahrzeuge, Fußgänger und Fahrbahnmarkierungen erkennen soll. Die Echzeitverarbeitung (33 fps) von Bilddaten soll dabei durch 128 parallele Verarbeitungseinheiten mit einem SIMD[2] Ansatz sichergestellt werden.

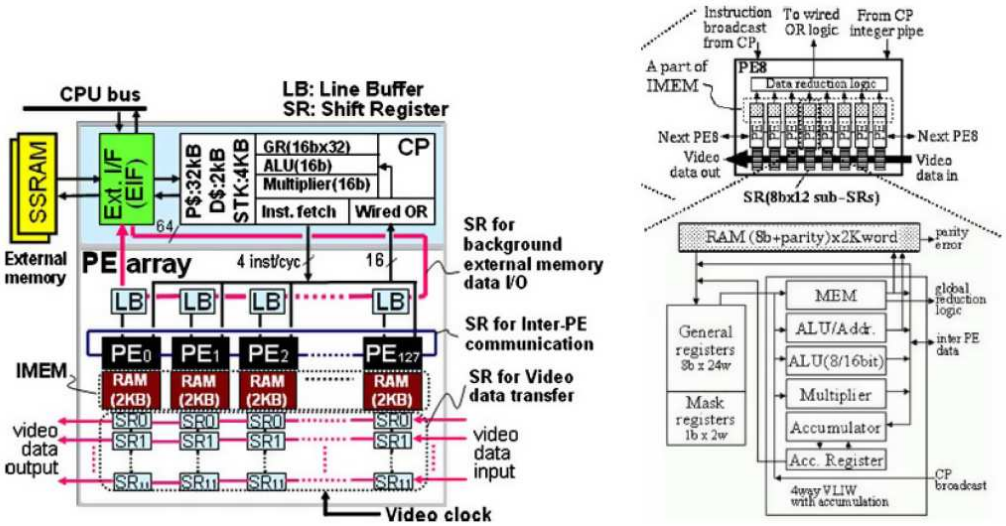


Abbildung 3.1: Blockdiagramm der IMAPCAR Architektur [SKO07]

Abbildung 3.1 zeigt die wesentlichen Teile des IMAPCAR Chips. Jedes der 128 Processing Elements (PEs) verfügt über eine 24-Bit Multiply Add Accumulate (MAC) Einheit und

[1] IMAPCAR: Image Memory Array Processor for CAR
[2] SIMD: Single Instruction Multiple Data

2 KB an lokalem Speicher, dem so genannten IMEM. Ein 16 bit RISC Control-Prozessor (CP) mit 32 KB Instruktions- und 2 KB Daten-Cache kann bis zu 4 Instruktionen pro Taktzyklus ausgeben, die an die PEs gesendet werden können. Ein External Interface (Ext. I/F) verfügt über einen DMA Controller, um Daten möglichst schnell zwischen externem SSRAM[3] (EMEM) und lokalem IMEM auszutauschen. Shift-Register werden verwendet, um Daten parallel zur Verarbeitung in den PEs in das IMEM oder das EMEM zu übertragen. Der IMAPCAR chip ist in 130 nm Technologie gefertigt und verbraucht bei einer Taktfrequenz von 100 MHz etwa 2 Watt. Der IMAPCAR Chip ist in Temperaturbereichen von -40 C bis +85 C getestet, automotive qualifiziert und wird derzeit im Lexus LS-460 eingesetzt.
Wie bereits erwähnt eignet sich die Parallelisierung vor allem für die Pixel-Level Operationen. Der Higher-Level Application Code lässt sich nur schwer auf diese Architektur abbilden. Um flexibler auf verschiedenste algorithmische Anforderungen zu reagieren, stellte NEC 2008 den rekonfigurierbaren XC Core [Pre08, PA09] vor, der in der IMAPCAR2 Architektur verwendet wird. Prengler beschreibt in [PA09] die Notwendigkeit von mehr Flexibilität. Er erwähnt, dass die Bildvorverarbeitung (Pixel-level Operationen) von SIMD Architekturen profitiert, jedoch die restlichen Schritte der Bildverarbeitung (Higher-Level Application code) weniger effektiv durchführen können, da sich diese häufig nicht parallelisieren lassen und der Floating Point Arithmetik bedürfen. In [PA09] wird für den Higher-Level Application code daher ein MIMD[4] Ansatz favorisiert. Um flexibel auf SIMD und MIMD Befehle reagieren zu können, lassen sich immer 4 der 128 SIMD PEs zu insgesamt 32 Processing Units (PUs) zusammenfassen (rekonfigurieren), um effizient die sequentielle Verarbeitung der Higher Level Algorithmen durchzuführen. In Abbildung 3.2 ist die Rekonfiguration in NEC's XC Core dargestellt.

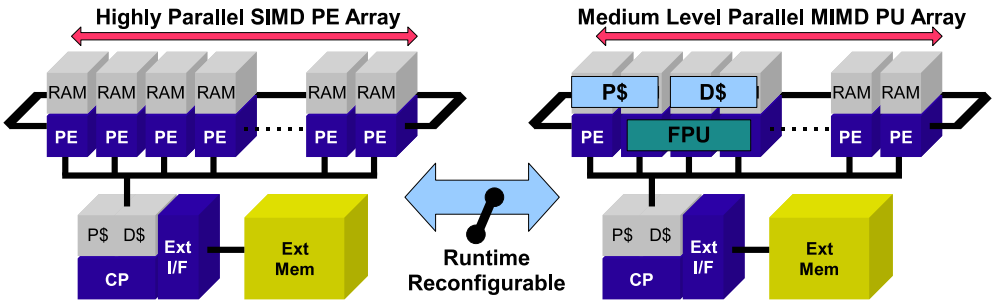


Abbildung 3.2: Rekonfiguration im XC Core von NEC [PA09]

Wie in Abbildung 3.2 zu erkennen ist funktioniert die Rekonfiguration im XC core durch zusammenfassen von 4 PEs. Jede der 32 PUs verfügt dann über ihren eigenen Instuktions-

[3]SSRAM: Synchronous Static Random Access Memory
[4]MIMD: Multiple Instruction Multiple Data

und Daten-Cache und zusätzlich einer Floating-Point Einheit. Somit kann durch Rekonfiguration des Systems ein Multiprozessor-System mit 32 Einheiten realisiert werden.

3.1.2 Der EyeQ Prozessor

In Zusammenarbeit mit ST Microelectronics entwickelte die Firma Mobileye den EyeQ Chip, einen ASIC für die Echtzeitbildverarbeitung und Szeneninterpretation in videobasierten FAS [RV06]. Ein Blockdiagramm der zweiten Generation, des EyeQ2 ASICs ist in Abbildung 3.3 dargestellt.

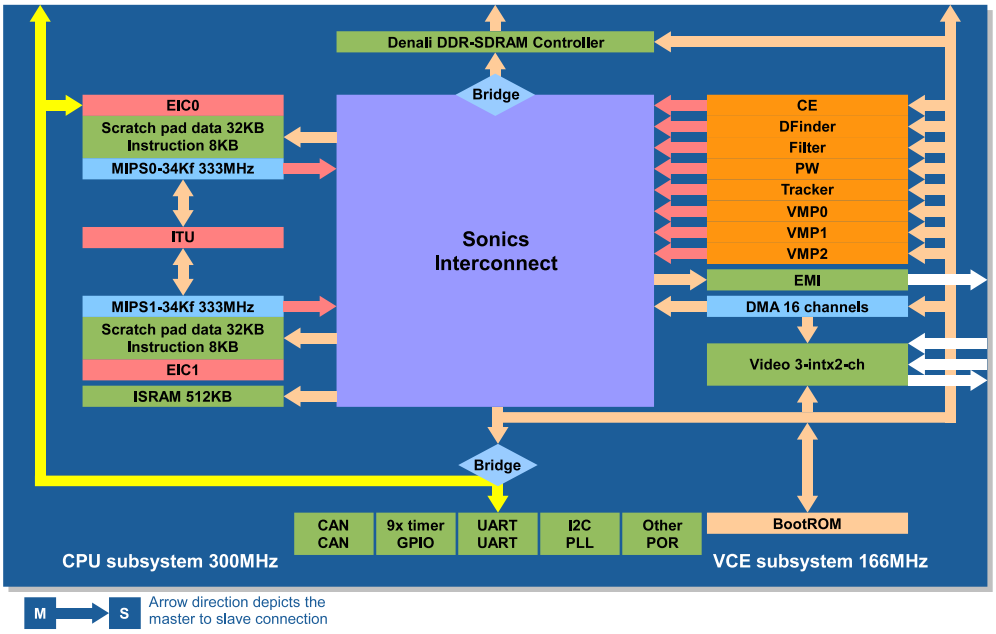


Abbildung 3.3: Blockdiagramm der EyeQ2 Architektur [STM06]

Im EyeQ2 sind 2 MIPS32 34Kf Prozessorkerne für die Highlevel Bildverarbeitung integriert. Untereinander können die beiden Prozessoren Daten über die Inter Thread Communication Unit (ITU) austauschen. Performanzlastige Teile der Bildverarbeitungsalgorithmen können in dedizierten Co-Prozessor Engines beschleunigt werden. Die EyeQ2 Architektur stellt dafür verschiedene Vision Computing Engines (VCEs) zur Verfügung. Eine Classifier Engine für die Saklierung, Vorverarbeitung und Musterklassifikation. Eine Tracker Engine für Bewegungsanalyse und Bildentzerrung. Eine Engine für die Bildvorverarbeitung, wie beispielsweise Kantendetektion, Faltungen oder die Verarbeitung

ganzer Bildpyramiden[5]. Eine Engine zur Bestimmmung von Disparitäten[6] in Stereobildern. Diese 5 VCEs waren bereits in der ersten Generation des EyeQ ASICs vorhanden. Zusätzlich beinhaltet der EyeQ2 Chip zahlreiche Peripherie-Controller zur Ansteuerung von CAN, UART, I^2C, DDR SDRAM Controller und Video Controller.
Wie bereits erwähnt ist eine flexible Plattform notwendig, um auch in Zukunft auf algorithmische Änderungen reagieren zu können. Verglichen mit der ersten Generation existieren im EyeQ2 deshalb zusätzlich so genannte Vector Microcode Prozessoren (VMPs), die die Flexibilität des Systems erhöhen sollen. Die VMPs sind Vector-Prozessor Einheiten ohne Cache und eignen sich daher gut für streaming Anwendungen. Durch die VMPs können sich Programmierer eine Verarbeitungs-Pipeline selbst zusammen stellen. Die beiden MIPS Cores, die 5 VCEs und die 3 VMPs sind über das 128-Bit Sonics Multi-service eXchange Interconnect (SMX) miteinander verbunden. Die 5 VCEs und 3 VMPs können theoretisch simultan Bilddaten verarbeiten, obwohl bei den meisten Anwendungen nur ein kleines Subset der verfügbaren Funktionalität verwendet wird. Die Einführung der VMPs weist darauf hin, dass in zukünftigen Architekturen die Flexibilität immer wichtiger wird. Flexibilität wird in Zukunft nicht nur für die Higher Level Algorithmen wichtig, sondern auch für die Pixel-level Operationen. In einem ASIC lassen sich die Coprozessoren (VCEs) jedoch nicht mehr ändern, sodass bei einer Änderung der Pixel-Level Operationen auf die VMPs ausgewichen werden muß. Die Verlustleistung des EyeQ2 Chips gibt Mobileye mit 3 Watt an [RV06].

3.1.3 Der VIP Prozessor

Infineon hat mit dem Vision Instruction Processor (VIP) [RBH+03] einen Multi-core Microcontroller vorgestellt, der 16 PEs für SIMD Befehle besitzt. Jedes der PEs besitzt einen lokalen Speicher. Zusätzlich verfügt der VIP Prozessor über einen on-chip DSP beispielsweise für High-Level Bildverarbeitungsaufgaben. Wie bei den zuvor erwähnten Prozessoren existiert mittlerweile auch eine zweite Generation des VIP Prozessors (VIP-II) dessen Blockdiagramm in Abbildung 3.4 dargestellt ist.
Wie in Abbildung 3.4 zu erkennen ist, sind im VIP-II Chip 4 SIMD Kerne und ein ARM 9 Prozessor vorhanden. Kontrolliert werden die SIMD Kerne, die wiederum 4 PEs beinhalten, durch einen internen general purpose Kern. Die SIMD Kerne sind über einen 6-Layer System Bus (128 Bit @ 150 MHz) miteinander und über eine Busbrücke mit dem ARM 9 Kern verbunden. Durch VLIW[7] Instruktionen können parallel arithmetische Operationen und Speicherzugriffe erfolgen. Die general purpose Kerne und der ARM 9

[5] Bei der Verarbeitung mit einer Bildpyramide, wird ein Eingangsbild in mehreren Auflösungen verarbeitet.
[6] Unter Disparität versteht man die Verschiebung zwischen korrespondierenden Bildpunkten in zwei unterschiedlichen Bildern eines Stereo-kamera Systems.
[7] VLIW: In einem Very Long Instruction Word (VLIW) wird durch den Compiler eine bestimmte Anzahl mehrerer unabhängiger Befehle zusammengefasst, die in einem Prozessor parallel abgearbeitet werden [GM07].

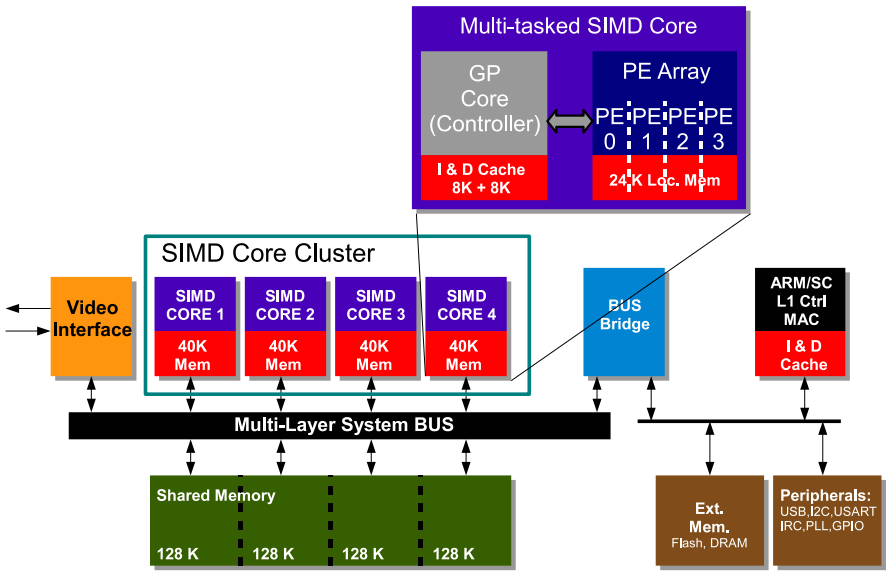


Abbildung 3.4: Blockdiagramm der VIP-II Architektur [Tec07]

Prozessor können durch C-Code programmiert werden. Für die Programmierung der SIMD Kerne wird eine Erweiterung von C namens DPCE (Data Parallel C Extension) verwendet. Erwähnenswert ist die relativ niedrige Verlustleistung von 300 mW bei einer Taktung von 300 MHz für SIMD Kerne und 250 MHz für den ARM Prozessor.

3.1.4 Zusammenfassung Bildverarbeitungsprozessoren

Für die video-basierte FAS existieren weitere kommerziell verfügbare Prozessoren, wie beispielsweise die SH4A ACC Mikrocontroller-Familie von Renesas [Loc10]. Der SH74552 und der SH74562 Mikrocontroller dieser Familie sind optimiert für den Einsatz in kamera- oder radar-basierten Sensoren. Die Mikrocontroller verfügen über Hardwarebeschleuniger für beispielsweise Matrix-Multiplikationen. Da diese Architektur im Vergleich zu den anderen vorgestellten Prozessoren keine wesentlichen Unterschiede oder Neuerungen darstellt, wird an dieser Stelle auf eine Vorstellung verzichtet.
Beekema und Broeders präsentieren in [BB08] eine Zusammenfassung von Architekturen für die video-basierte FAS. Dort wird erwähnt, dass insbesondere der EyeQ Chip vom Einsatz rekonfigurierbarer Architekturen profitieren könnte, da nicht alle VCEs zur selben Zeit benötigt werden.
MacLean beschreibt in [Mac05] eine Analyse über die Eignung von FPGAs als Plattform für eingebettete Systeme zur Bildverarbeitung. Er sieht die größten Vorteile in der Rekonfigurierbarkeit der FPGAs und die Möglichkeit die verfügbare Parallelität zur Beschleunigung von Bildverarbeitungsalgorithmen zu nutzen. MacLean beschreibt den Einsatz von

FPGAs als vielversprechend, falls eine automatisierte Floating-Point Fixed-Point Umwandlung der Algorithmen und deren Partitionierung über mehrere Devices adressiert werden. Sinnvoll ist laut MacLean auch die Einbeziehung von Soft- und Hardcore Prozessoren, um SW Elemente in das Design einzubinden, sowie die Entwicklung eines Modulbaukastens, zum Entwurf komplexerer Systeme. Ein weiterer in [Mac05] genannter Vorteil von (Xilinx) FPGAs besteht laut MacLean in der möglichen Selbst-Rekonfiguration der Devices und deren Einsatz im normalen Betriebsablauf. Gegenwärtige dynamisch rekonfigurierbare Architekturen für die Bildverarbeitung werden daher im folgenden Abschnitt dargestellt.

3.2 Dynamisch Rekonfigurierbare Architekturen für die Bildverarbeitung

Zahlreiche FPGAs werden heutzutage schon in Fahrzeugen verbaut, beispielsweise im Rear Seat Entertainment System von Harmann/Becker oder im Adaptive Cruise Control System von Continental. Der gegenwärtige Einsatz von FPGAs in video-basierten FAS ist, mit Ausnahme des Bosch NightVision [Hau05] Systems, nicht dokumentiert. Jedoch existieren zahlreiche Konzepte für diese Anwendungsdomäne [GP03, Bar08, BMZ08, San08]. In [Eka04] wird ein Konzept für den Einsatz von FPGAs als Coprozessor in Multimedia- und Telematiksystemen vorgestellt, um high performance Anforderungen, wie das Dekodieren von Videos in Echtzeit gerecht zu werden. Gegenwärtig gibt es jedoch keine kommerziell verfügbaren FPGA-Systeme, die die dynamische partielle Rekonfiguration nutzen. Neben der in dieser Arbeit vorgestellten rekonfigurierbaren Architektur existieren jedoch in der Literatur weitere Ansätze für dynamisch rekonfigurierbare, FPGA-basierte Bildverarbeitungssysteme. Sie repräsentieren den aktuellen Forschungsstand und werden im Folgenden vorgestellt.

3.2.1 Die Sonic-on-a-chip Architektur

Eine rekonfigurierbare Architektur zur Verarbeitung von Bilddaten wurde von Sedcole [Sed06] vorgestellt. Ein Block Diagramm der Sonic-on-a-chip Architektur ist in Abbildung 3.5 dargestellt.
In der Sonic-on-a-chip Architektur wird ein Mikroprozessor System um das Sonic-Subsystem erweitert. Über ein Interface lassen sich ein oder mehrere Instanzen eines Sub-Busses, dem so genannten SonicBus, anbinden. Die Kommunikation zwischen diesen Bussen wird durch Busbrücken realisiert. An bestimmten Stellen des SonicBus lassen sich konfigurierbare PEs über so genannte Sockets (Macro-Interfaces) anschließen. Auch für den Chain Bus existiert ein Socket für die Kommunikation zwischen den PEs. Der Chain Bus wird zur Kommunikation und zum Datentransfer benachbarter PEs verwendet, um den 32-Bit breiten SonicBus bei schnellen lokalen Datentransfers zu umgehen. Die PEs selbst bein-

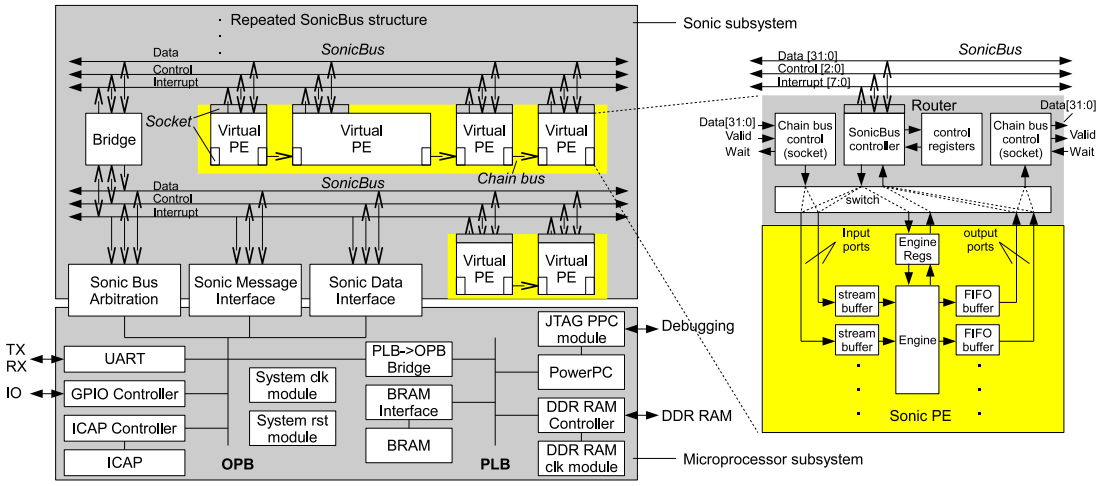


Abbildung 3.5: Sonic-on-a-chip Architektur [Sed06]

halten einen Router, eine Engine für die eigentliche Bildverarbeitung und lokale Speicher für die Pixeldaten. Die Daten werden der Engine durch stream buffer zur Verfügung gestellt und im Anschluß an die Verarbeitung in die FIFO buffer geschrieben. Die Anzahl an stream und FIFO buffern ist konfigurierbar und der Aufbau der Engine bleibt dem Anwender überlassen. Die dynamisch rekonfigurierbaren Anteile im System sind in Abbildung 3.5 gelb hinterlegt.

Mit dieser modularen Architektur lassen sich vor allem streaming Operationen gut realisieren. Allerdings ist das komplette Sonic System über den OPB Bus an das Microprozessor System angeschlossen. Sollte der DDR RAM als Frame Buffer dienen, müssen die Bilddaten über den PLB, den OPB und den Sonic Bus übertragen werden, um zu einem PE zu gelangen. Dies gilt analog für Bilddaten, die von einem PE verarbeitet wurden und in das DDR RAM geschrieben werden sollen.

3.2.2 Die RampSoC Architektur

Das Run-time Adaptive Multiprocessor System-on-Chip (RampSoC) [GHSB08, GPHB09] ist ein FPGA-basiertes, eingebettetes Multiprozessor System für High Performance Computing Anwendungen, wie z.B. Bildverarbeitung oder Bioinformatik. In diesem Multiprozessor System können verschiedene Arten von Microprozessoren über ein heterogenes Network-on-chip miteinander verbunden werden, wie in Abbildung 3.6 dargestellt ist.

Die verwendeten Prozessoren können sich hinsichtlich ihrer Bandbreite und ihrer Leistung unterscheiden. Zusätzlich ist es möglich an jeden Microprozessor einen oder mehrere Hardwarebeschleuniger anzubinden, um damit den Instruktionssatz der Prozessoren zu erweitern und performanzlastige Bildverarbeitungsoperationen zu parallelisieren. Abhän-

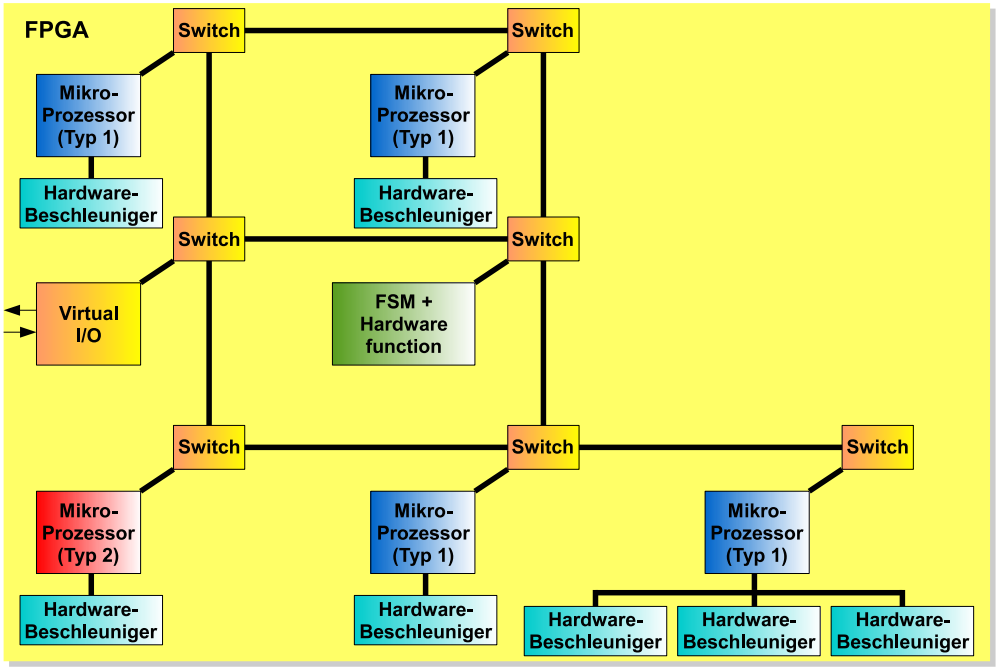


Abbildung 3.6: RampSoC Architektur [GHSB08]

gig von den Anforderungen lässt sich durch dynamische Rekonfiguration zur Laufzeit ein flexibles heterogenes Netzwerk von Mikroprozessoren mit eng gekoppelten Hardware-Beschleunigern realisieren. In diesem Multiprozessorsystem können mehrere Instanzen versuchen auf das Konfigurations-Interface (ICAP) zuzugreifen. Zur Auflösung von Zugriffskonflikten und für das Ressourcenmanagement auf dem FPGA wird ein eigens dafür konzipiertes Betriebssystem namens CAPOS (Configuration Access Port-Operating System) verwendet.

3.3 Forschung an dynamisch rekonfigurierbaren SoCs

In einem 6-jährigen Schwerpunktprogramm (SPP1148) der Deutschen Forschungsgemeinschaft (DFG) wurden „Rekonfigurierbare Rechensysteme“ untersucht. Da es zuvor keine Beschreibung- bzw. Modellierungsmöglichkeit für diese Art von Systemen gab, befassten sich einige der geförderten Projekte mit **Sprachen und Modellen** für Rekonfigurierbare Rechensysteme. Ein weiterer zentraler Teil des Schwerpunktprogramms war die **Analyse**, um dem Anwender Metriken und eine Kosten/Nutzen Abschätzung für rekonfigurierbare Rechensysteme zu liefern. Zur Erstellung Rekonfigurierbarer Rechensysteme sind **Entwurfsverfahren** notwendig, womit sich weitere Projekte innerhalb des Schwerpunktprogramms befassten. Die restlichen Projekte, zu denen auch das in dieser Arbeit

dokumentierte AutoVision Projekt zählte, befassten sich mit **Architekturen und Anwendungen** für Rekonfigurierbare Rechensysteme. Eine Übersicht aller Projekte und Teilforschungsbereiche ist in [CS09] dargestellt.
Durch die Verwendung von dynamischer Rekonfiguration in einem FPGA-basierten System entstehen zusätzliche Kosten, der so genannte Rekonfigurations-Overhead. Rekonfigurations-Overhead äußert sich hinsichtlich zusätzlicher Ressourcen (um die Rekonfiguration zu kontrollieren), zusätzlicher Latenz (abhängig von der Rekonfigurationsgeschwindigkeit) und zusätzlicher Verlustleistung (durch den Rekonfigurationsvorgang selbst).
In annähernd allen Literaturquellen, die ein dynamisch rekonfigurierbares System beschreiben, wird die Rekonfigurations-Geschwindigkeit als kritischer Designparameter genannt. Die Rekonfigurations-Geschwindigkeit ist abhängig vom Durchsatz an Konfigurations-Daten an der Konfigurations-Schnittstelle. In zahlreichen Arbeiten [LD09, SPA$^+$08, MMT$^+$08, LKLJ09, HL09, LPF09a] wurde daher versucht diesen Durchsatz auf Virtex-4 Plattformen zu optimieren. Ein hoher Durchsatz wurde in [LKLJ09] und [HL09] nur auf Kosten eines sehr hohen Verbrauchs an Block RAM ermöglicht.
Ein weiteres Entwurfsziel in dynamisch rekonfigurierbaren Systemen ist die Minimierung von Verlustleistung. Eine Übersicht über Verlustleistung in 65 nm FPGAs und deren Ursachen ist in [Cur07] dargestellt. Gegenwärtige Tools zur Abschätzung der Verlustleistung wie Xpower von Xilinx sind nicht in der Lage die Verlustleistung während der dynamischen Rekonfiguration abzuschätzen. In der Literatur existieren nur wenige Quellen, die dieses Thema adressieren. In [BHU03] sind beispielsweise Verlustleistungsmessungen auf einem Xilinx FPGA während der dynamischen Rekonfiguration dokumentiert. Die Ergebnisse zeigen, dass die I/O Verlustleistung während der Rekonfiguration ansteigt, da die Konfigurations-Daten von extern geladen werden, die Core Verlustleistung jedoch annähernd konstant bleibt. Weitere Messergebnisse während der dynamischen Rekonfiguration eines AT40K20 FPGAs von ATMEL sind in [LMGVE04] aufgeführt. Um die Verlustleistung während der Rekonfiguration so gering wie möglich zu halten, empfehlen Lorenz et. al in [LMGVE04] bei höchstmöglicher Frequenz zu rekonfigurieren, um diesen Vorgang schnell abzuschließen.
Dagegen existieren weitaus mehr Konzepte zur Einsparung dynamischer und statischer Verlustleistung in FPGAs. Ein Ansatz zur Einsparung von dynamischer Verlustleistung durch Modulation der Taktfrequenz, dem so genannten clock scaling, ist in [PHBB07] beschrieben. Zur Erzeugung verschiedener Takte auf dem FPGA wird hier ein Digital Clock Manager (DCM) verwendet. Durch Veränderung der zur Konfiguration des DCMs verwendeten Speicherzellen, lassen sich verschiedene Taktteilerverhältnisse und damit verschiedene Frequenzen erzeugen. Für das Ändern dieser Speicherzelleninhalte zur Laufzeit wird die dynamische Rekonfiguration eingesetzt.
Ein weiterer Ansatz zur Einsparung dynamischer Verlustleistung besteht in der Abschaltung ganzer Takt-Zweige nicht benötigter Einheiten, was als Clock Gating (CG) bezeichnet wird. In [ZRM06, HMA09, LPF09a, OLCM08] werden verschiedene Ansätze vorgestellt, die CG auf einem FPGA realisieren. In der neusten Spartan-6 und Virtex-6

Generation lässt sich CG relativ feingranular, nämlich auf Slice-Ebene durch den Einsatz von Clock Enables[8] realisieren [Riv10]. Deviceseitig existieren pro CLB Slice Clock Enables, um maximal 8 Register von der Taktversorgung zu trennen. Im Normalfall reicht jedoch das Abschalten kompletter Takt-Zweige, die ein ganzes Modul versorgen aus.
Verschiedene Untersuchungen und Ansätze zur Reduktion von statischer Verlustleistung auf FPGAs finden sich in [ANT04, SGVT05, TL03, LPF09a]. In [TL03] wird die Verlustleistung eines in 90 nm Technologie gefertigten Virtex-4 FPGAs beschrieben. Zusätzliche Ursachen für statische Verlustleistung in 65 nm Strukturen und darunter sind in [RCK07, Ras07] aufgeführt. In [ANT04] und [SGVT05] wurde die statische Verlustleistung abhängig von der Beschaltung der Eingänge eines Multiplexers in einem FPGA Design untersucht. Die Ergebnisse zeigen, dass sich die statische Verlustleistung dadurch signifikant ändern kann. Die SRAM Zellen, die den Selecteingang der Multiplexer steuern (siehe Abbildung 2.7(b)), werden dabei allerdings nicht verändert.
In [LPF09a] wird das Löschen eines rekonfigurierbaren Moduls mit dessen Abschalten über CG verglichen. Die Ergebnisse weisen darauf hin, dass sich durch Löschen des Moduls neben der dynamischen Verlustleistung (wie beim CG) auch statische Verlustleistung einsparen lässt. Allerdings wird in [LPF09a] für dieses Verhalten keine Erklärung gegeben. Außerdem wurde in [LPF09a] nur die Verlustleistung des Cores betrachtet, nicht aber dass beim Laden eines Bitstroms aus einem externen Speicher auch die Verlustleistung der IO Pins steigt.

3.4 Zusammenfassung

In diesem Kapitel wurden die aktuellen Forschungsarbeiten auf dem Gebiet der dynamischen Rekonfiguration, sowie eine Reihe an kommerziell erhältlichen Prozessoren für die Bildverarbeitung im Automobil vorgestellt.
Die in Abschnitt 3.1 vorgestellten Prozessoren lassen alle einen Trend zu mehr Flexibilität erkennen. Besonders der EyeQ2 Prozessor, der Hardwarebeschleuniger für die Low-level Bildverarbeitung verwendet, kann vom Einsatz rekonfigurierbarer Hardware profitieren. Durch den Einsatz von FPGAs lässt sich die Funktionalität der Co-Prozessoren nicht nur im Nachhinein ändern, sondern es besteht zusätzlich die Möglichkeit benötigte Co-Prozessoren zur Laufzeit des Systems zu laden.
Viele dynamisch rekonfigurierbare Systeme sind als Subsysteme an beispielsweise ein CoreConnect Mikroprozessor System gekoppelt. Innerhalb der rekonfigurierbaren Systeme werden eigens entwickelte Bussysteme verwendet, die über Brücken an das Microprozessorsystem angeschlossen sind. Beispiele hierfür sind die Sonic-on-a-Chip Architektur [Sed06] und die RAPTOR Architektur [Ket09].
In fast der gesamten verfügbaren Literatur wird die Rekonfigurations-Geschwindigkeit

[8] Durch Clock Enables erfolgt die Freischaltung des Takteingangs.

und der damit verbundene Durchsatz an Rekonfigurationsdaten als kritischer Designparameter erwähnt.
Ein Bedarf an Weiterentwicklungen besteht daher in der Erhöhung des Rekonfigurations-Durchsatzes und einer damit einhergehenden Reduktion der Zeit, die für eine Rekonfiguration aufzuwenden ist. Die Rekonfiguration soll dabei so schnell sein, dass zwischen zwei aufeinanderfolgenden Video-Frames rekonfiguriert werden kann, ohne dass ein Video-Frame verworfen werden muss.
Da gegenwärtig keine Richtlinien oder gar Bibliotheken zur Erstellung von Hardware-Beschleunigern existieren, wird in dieser Arbeit eine modulbasierte Pixelpipline vorgestellt, mit Hilfe derer verschiedenste HWAs durch flexible Verschaltung realisiert werden können.
Da bei abnehmender Strukturgröße die statische Verlustleistung in FPGAs eine immer dominantere Rolle einnimmt, wird in dieser Arbeit ein Konzept und dessen Realisierung zum dynamischen Powermanagement vorgestellt.
Dieser Bedarf an Weiterentwicklungen wird in den folgenden Kapiteln aufgegriffen und detailliert erörtert.

4 Die AutoVision SoC Architektur

In diesem Kapitel wird die AutoVision Architektur vorgestellt. In dieser Architektur werden die performanzlastigen Teile von Bildverarbeitungsalgorithmen in HW ausgelagert, während der Rest frei programmierbar auf eingebetteten Standardprozessoren (PPC, Microblaze) laufen soll. Viele Bildverarbeitungsoperationen auf Pixel(-umgebungen) erfordern sich oft wiederholende Prozesse und eignen sich daher gut für eine HW Implementierung (Bildvorverarbeitung). Diese Operationen werden als Pixel Level Operationen bezeichnet. Diese Vorverarbeitung bildet oft den echtzeitkritischen Teil, den es zu beschleunigen gilt. Kontrollflußbasierte Teile der Bildverarbeitungsalgorithmen sind dagegen für die Implementierung auf einem Prozessor geeignet.
Dieses Kapitel ist folgendermaßen aufgebaut: In Abschnitt 4.1 werden zunächst Designrichtlinien vorgestellt, die maximalen Performanzzugewinn im Vergleich zu einer prozessor-basierten Lösung gewährleisten sollen. Die Umsetzung dieser Richtlinien bildet das Grundkonzept der Hardware Acceleratoren (HWAs)[1] im AutoVision System.
In Kapitel 4.2 wird die Sytemumgebung für die on-chip Bildverarbeitung vorgestellt. Zusätzlich werden hier die für den on-chip Pixeltransfer notwendigen HW Komponenten präsentiert.
In Abschnitt 4.3 wird dann die Realisierung der in Abschnitt 4.1 beschriebenen Konzepte erläutert. Durch den modularen Aufbau der Pixelverarbeitungskette können verschiedenste Bildverarbeitungsroutinen realisiert werden. Die HWAs regeln den Transfer von Pixeldaten von und zum Speicher selbständig über DMA Transfers, was zu einer deutlichen Entlastung der CPU führt. Durch den modulbasierten Aufbau der Pixelverarbeitungskette kann sich der Anwender auf die Implementierung der Operationen auf ein einzelnes Pixel und dessen Nachbarschaft beschränken.
In Abschnitt 4.4 werden dann beispielhaft verschiedene Algorithmen für die Bildverarbeitung im Automobil, deren Umformung und Optimierung für eine HW Implementierung vorgestellt. Die in diesem Abschnitt aufgeführten Algorithmen kommen im AutoVision System zum Einsatz.

4.1 Hardware Beschleunigung

Die in Abschnitt 2.1 vorgestellten Sliding Window Verfahren wenden dieselbe Operation auf eine Vielzahl von Pixel und deren Umgebung an. Diese unabhängigen, daten-

[1] Hardware Acceleratoren sind Co-Prozessoren, deren Aufgabe in der Beschleunigung einer Operation und der gleichzeitigen Entlastung einer CPU besteht.

flußbasierten Operationen eignen sich hervorragend für HW Beschleunigung. In diesem Abschnitt werden Konzepte zur Modellierung von Hardware Acceleratoren (HWAs) vorgestellt, die zu einem möglichst großen Performanzgewinn führen sollen. Hardware Beschleunigung für Bildverarbeitungsoperationen ist insbesondere dann sinnvoll, wenn die folgenden Bedingungen erfüllt werden können.

- Erfordert eine Operation reguläre Scans über das Bild (Sliding Windows), ist es sinnvoll kleine lokale Speicher zu verwenden, um mehrmaliges Lesen der selben Pixeldaten aus dem Hauptspeicher zu vermeiden (s. Abschnitt 4.1.1).
- Erfordert eine Operation Lese-/Schreiboperationen aus zusammenhängenden Speicherbereichen, sind Burst Transfers von Pixeldaten sinnvoll (s. Abschnitt 4.1.2).
- Erfordert eine Operation die Verarbeitung eines zentralen Pixels und einer großen Pixelnachbarschaft, kann die Verarbeitung des zentralen Pixels mit jedem Pixel innerhalb der Nachbarschaft parallel und theoretisch in einem Taktzyklus erfolgen (s. Abschnitt 4.1.3).
- Erfordert der Algorithmus mehrere Scans über Eingangsbilder und Zwischenresultate, kann eine gepipelinete Architektur die Verarbeitungszeit mitunter deutlich verkürzen (s. Abschnitt 4.1.4).
- Führt die Prozessierung in HW zu einer signifikanten Datenreduktion, können eingebettete Prozessoren im Anschluß auf der reduzierten Ergebnismenge (Featureliste) weiterarbeiten (s. Abschnitt 4.1.5).

In der AutoVision SoC Architektur werden nur die performanzlastigen Teile in HW ausgelagert und durch HWAs beschleunigt. Geschickt umgesetzt, führt dies im Normalfall zu einer Beschleunigung und einem reduzierten Ergebnisdatensatz, den weitere Komponenten (CPUs, DSPs) weiterverarbeiten können. Der Rest des Algorithmus, der higher level application code, ist frei programmierbar auf einer eingebetteten Standard CPU (PPC) implementiert, um Flexibilität und einfache Updatebarkeit für zukünftige Algorithmen zu erlauben. Zusätzlich soll es möglich sein, über die eingebettete CPU den Austausch von HW Beschleunigern zu initiieren. Dieser Austausch wird explizit in Kapitel 5 beschrieben.

4.1.1 Lokale Speicher

Wie in Abbildung 4.1 zu erkennen ist, wird bei Sliding Window Operationen ein großer Teil der Pixel im darauf folgenden Schritt erneut benötigt. Während in der ersten 3×3 Nachbarschaft an Position 1 (mit dem zentralen Pixel $\binom{1}{1}$) die Pixel $\binom{0}{0}$, ..., $\binom{2}{2}$ verarbeitet werden, sind es im darauf folgenden Schritt (an Position 2) die Pixel $\binom{0}{1}$,..., $\binom{2}{3}$. Wie in Abbildung 4.1 zu erkennen ist, werden $\frac{2N_W}{2N_W+1}$ % der Pixel in der Nachbarschaft für den

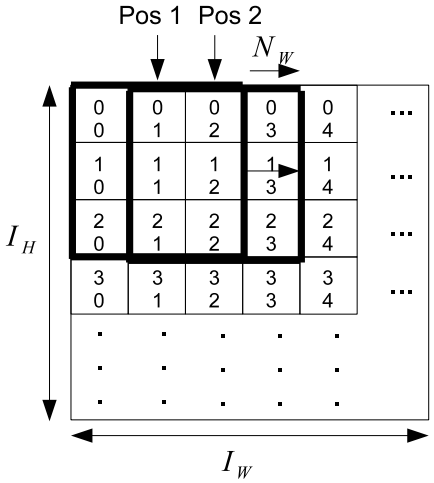


Abbildung 4.1: Beispiel für den sinnvollen Einsatz lokaler Speicher.

Folgeschritt benötigt, wenn das Processing Window einen Pixel weiter rückt. Anstatt diese Pixel erneut aus dem Hauptspeicher zu laden, bietet es sich an, eine gewisse Anzahl an mehrfach verwendeten Pixeln in einem lokalen Speicher innerhalb der HWAs abzulegen und neue Pixel bei Bedarf nachzuladen.

4.1.2 Zugriff auf zusammenhängende Speicherbereiche (Bursting)

Wird mit einem Sliding Window zeilenweise über das Eingangsbild iteriert, macht es Sinn komplette Bildzeilen zu laden. Anstatt dem HWA also die Pixel einzeln zuzuführen und im lokalen Speicher abzulegen, wird lese- sowie schreibseitig auf zusammenhängende Speicherbereiche zugegriffen. Dies hat den entscheidenden Vorteil der Auslastungsminderung auf dem Interconnect (Bsp.:PLB) durch die Vermeidung zusätzlicher Arbitrations- und Adressierungszyklen. So können relativ schnell komplette Bildzeilen geladen werden, was vor allem bei regulären Scans von Sliding Windows sinnvoll ist.

4.1.3 Verarbeitung großer Pixelnachbarschaften (Parallelismus)

Bei Single Instruction Multiple Data (SIMD) (s. [GM07]) wird eine Instruktion auf mehrere voneinander unabhängige Operandenpaare angewendet. Die Operandenpaare können in parallel angeordneten Prozessierungs-Elementen (PEs) verarbeitet werden. Typische SIMD Operationen sind beispielsweise der Vergleich eines zentralen Pixels mit all den umliegenden Pixeln in der Nachbarschaft oder eine Faltung der Nachbarschaft mit einem Filterkern (Bsp.: morphologische Operatoren [GW08]).

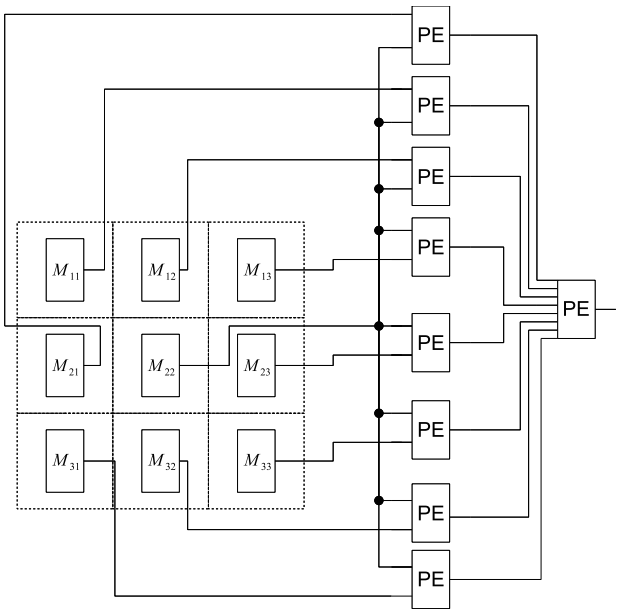


Abbildung 4.2: Konzept zur parallelen Verarbeitung eines zentralen Pixels (M_{22}) mit seiner Nachbarschaft

In Abbildung 4.2 ist die Verarbeitung eines zentralen Pixels mit seinen 8 umliegenden Pixeln (M_{11} bis M_{33}) in einer 3×3 Nachbarschaft dargestellt. Bei sequentieller Ausführung würden beispielsweise die 8 Vergleiche der umliegenden Pixel innerhalb eines PEs 8 Taktzyklen, bei 8-fach paralleler Ausführung jedoch nur einen Taktzyklus dauern. Innerhalb der parallelen PEs muß nicht zwangsläufig die selbe Operation realisiert werden.

4.1.4 Sofortige Verarbeitung von Zwischenergebnissen (Pipelining)

Die Systemfrequenz in synchronen Schaltungen ist beschränkt durch den kritischen Pfad. Der kritische Pfad ist dabei abhängig von der längsten Signallaufzeit durch die kombinatorische Logik. Pipelining kann dazu verwendet werden, diesen kritischen Pfad durch das Einführen zusätzlicher Registerstufen aufzubrechen und damit zu verkürzen. Auf höherer Abstraktionsebene kann Pipelining verwendet werden, um beispielsweise den Durchsatz in Bildverarbeitungssystemen zu erhöhen. Wenn ein Algorithmus mehrere Scans über ein Eingangsbild und die daraus resultierenden Ergebnisbilder benötigt, werden oft mehrere Zwischenresultate erzeugt. Anstatt diese Zwischenresultate komplett zu verarbeiten und zurück in den Hauptspeicher zu schreiben, kann eine gepipelinete Verarbeitung verwendet werden, um die Ausführungszeit drastisch zu verkürzen. Sobald genügend Zwischenresultate für die nächste Pipelinestufe zur Verfügung stehen, kann mit der Verarbeitung begonnen werden.

4.1.5 Reduktion der zu verarbeitenden Datenmenge (Feature conversion)

Die Verarbeitung von Bilddaten in HW macht besonders dann Sinn, wenn die Prozessierung im Vergleich zu den Eingangsdaten eine Datenreduktion zur Folge hat, beispielsweise Features extrahiert oder eine Region of Interest (ROI) bestimmt wurde. Bei einer Feature Extraktion werden die Eingangsdaten in eine Menge von Punkten mit bestimmten Merkmalen (Features) umgewandelt. Ein Feature kann die Koordinaten im Bild, die Größe, eine durchschnittliche Helligkeit etc. von relevanten Daten im Eingangsbild enthalten. Die weitere Verarbeitung einer Feature Liste kann dann auf einer on- oder off-chip CPU oder einem DSP erfolgen.

4.2 System Übersicht

In Abbildung 4.3 ist ein Blockdiagramm der AutoVision Architektur dargestellt.

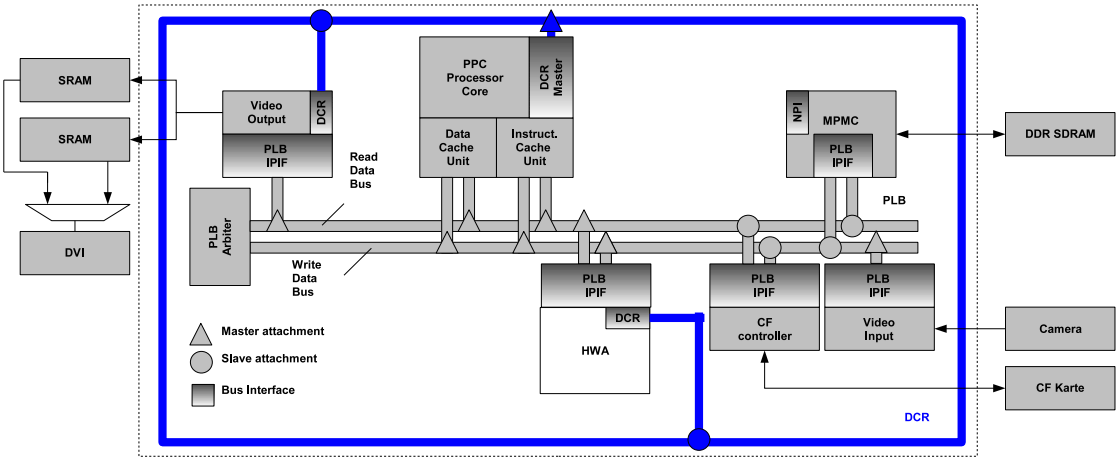


Abbildung 4.3: Blockdiagramm der AutoVision Architektur

Mit Hilfe von Abbildung 4.3 läßt sich der Datenfluß an Pixeldaten verdeutlichen. Die von einer Kamera gelieferten Pixeldaten werden kurzfristig in einem Video Input Core (s. Abschnitt 4.2.2.3) zwischengepuffert und dann über DMA Transfers mit Hilfe von Bursts in den DDR/DDR2 SDRAM geschrieben. Alle Busmaster im AutoVision Sytem regeln die Bursttransfers völlig selbstständig. Dies beinhaltet die Generierung von Requests und die Berechnung der richtigen Speicheradressen. Vier konsekutiv aufgenommene Grauwertbilder der Kamera und die Ergebnisse der Bildverarbeitung werden im DDR/DDR2 SDRAM abgelegt. Von dort können HWAs (ebenfalls über DMA Transfers) oder CPUs auf die Eingangsbilddaten zugreifen, diese verarbeiten und wieder zurückschreiben. Letztendlich kann der Video Output Core die verarbeiteten Daten am Ausgang (VGA oder

DVI Port) darstellen. Die Speicherorganisation im AutoVision System ist dabei wie folgt geregelt. Reguläre Scans (Sliding Windows) über Bilder erlauben das Laden von Pixeldaten vor ihrer eigentlichen Verarbeitung. Durch diese Prefetching Methode kann bei einer geeigneten Speicherorganisation sichergestellt werden, dass die Pixel rechtzeitig an der Verarbeitungseinheit (z.B. HWA) zur Verfügung stehen. Auf den Entwicklungsboards gibt es vier Speicherarten, die für das Puffern von Pixeldaten verwendet werden können. Diese unterscheiden sich in Größe und Zugriffszeit. Ein nicht flüchtiger Flashspeicher (Compact Flash (CF) Karte) kann zum Speichern zahlreicher Bilddaten und partiellen Bitströmen verwendet werden. Bereits aufgenommene Bildsequenzen können hier für Testzwecke gespeichert werden. Teile der Bildsequenzen oder Bilder von der Kamera, werden dann in den Hauptspeicher, ein DDR/DDR2 SDRAM, geladen. Die Bilddaten werden vor ihrer Verarbeitung im Hauptspeicher zwischengepuffert, um beim Austausch von HW Beschleunigern keine Pixel zu verlieren. Außerdem erlaubt das temporäre Speichern im Hauptspeicher einen wesentlich schnelleren Zugriff auf die Bilddaten verglichen mit einem Flash Speicher. Lokale Speicher in den HWAs werden meist durch Block RAMs (BRAMs) seltener durch Distributed RAM realisiert. In den BRAMs werden nur einzelne Pixelzeilen, nicht jedoch ganze Bilder gepuffert. Diese lokalen Speicher sind wesentlich kleiner, dafür aber auch wesentlich schneller als der Hauptspeicher. Letztendlich werden in Registern zentrale Pixel und deren umgebende Nachbarschaft zwischengespeichert, um den simultanen Zugriff auf diese Daten in einem Taktzyklus zu erlauben.

4.2.1 HW Plattformen

Die AutoVision Architektur wurde auf einem XUPV2P board und auf einem ML507 der Firma Xilinx prototypisch implementiert, um damit Messungen an diesem System durchzuführen. Im Folgenden wird nun das Xilinx ML507 Embedded System Entwicklungsboard vorgestellt. Ein Virtex-5 FX FPGA wurde ausgewählt, da diese Plattform zur Implementierungszeit die aktuellste FPGA Plattform von Xillinx mit eingebettetem PPC Core darstellte. Zudem ist die interne Struktur eines Virtex-5 FPGAs sehr ähnlich zu Spartan-6 FPGAs, die vornehmlich bei Automobilanwendung, aufgrund geringer Kosten und Verlustleistung zum Tragen kommen. Die ML507 Entwicklungsplattform ist in Abbildung 4.4 dargestellt.
Auf dem ML507 Entwicklungsboard ist ein Virtex-5 XC5VFX70T-FFG1136 FPGA verbaut, der zur Steuerung zahlreicher Peripherie Komponenten verwendet werden kann. Der FPGA beinhaltet 11200 CLB Slices, 44800 CLB Flip-Flops, 5,328 Kb (148 x 36 Kb) an Block RAM, 128 DSP48E Slices sowie einen IBM PowerPC440 Processor. Zusätzlich sind auf dem Board 256 MB an DDR2 SDRAM Speicher, ein DVI Video Output, Compact Flash Speicher, 5 Push Buttons und 8 LEDs vorhanden. Das JTAG Interface wurde für die Programmierung des FPGAs verwendet während die Kommunikation zwischen PC und FPGA über eine RS232 Schnittstelle stattfindet.
Als Kamera wurde eine monochrome Flächenkamera des Herstellers JAI (CM-140MCL)

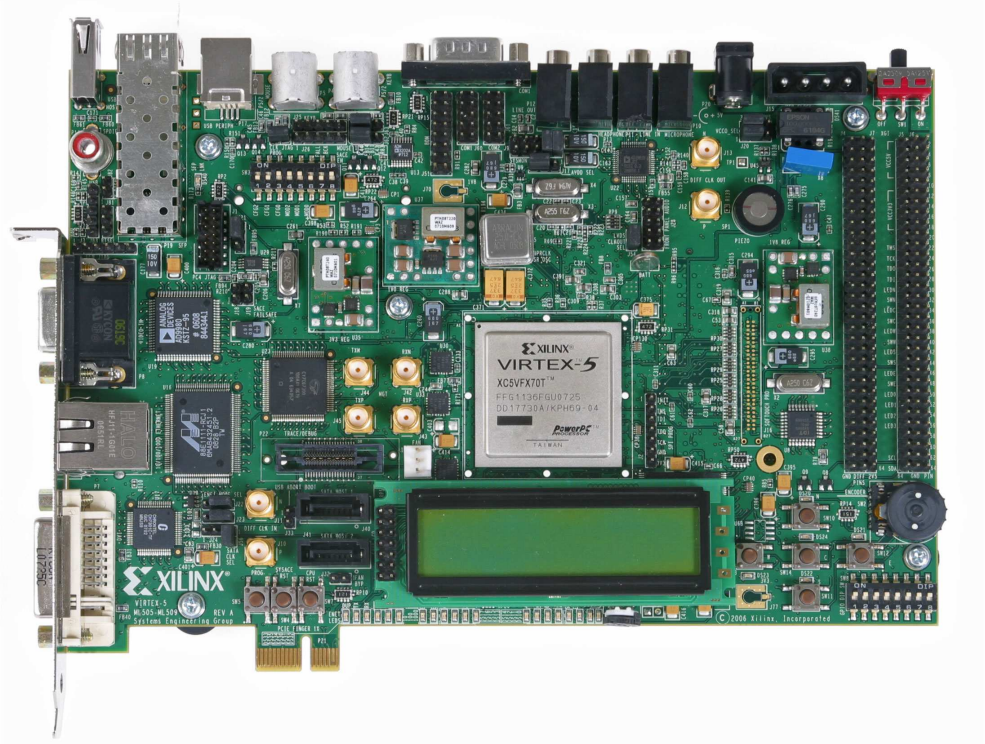


Abbildung 4.4: ML507 Virtex-5 Development Board [Xil09a]

mit Mini CameraLink Anschluß verwendet [JAI10]. Diese Kamera verfügt über einen 1/2" IT-Progressive-Scan-CCD Sensor der Bilddaten mit einer Auflösung von maximal 1392×1040 Pixeln und einem Pixeltakt von 65 MHz liefert. Die 31 Bilder pro Sekunde, die von der Kamera geliefert werden, führen auf eine Verarbeitungszeit von 32,25 ms pro Bild, wenn die Daten in Echtzeit verarbeitet werden sollen.

4.2.2 Vision-based Intellectual Property cores

In diesem Abschnitt werden die IP cores vorgestellt, die zur Übertragung und Verarbeitung von Bilddaten im AutoVision System genutzt werden. Zur Übertragung ist ein IP Interface (IPIF) (s. Abschnitt 4.2.2.1 und 4.2.2.2), ein Video Input Core (s. Abschnitt 4.2.2.3) und ein Video Output Core (s. Abschnitt 4.2.2.4) notwendig.

4.2.2.1 Interface zur Anbindung an den PLB Bus

Das LIS PLB IPIF [Zep07] stellt im Gegensatz zum Xilinx PLB IPIF [Xil06b] eine wesentlich performantere Lösung dar. Es besteht aus getrennten Master und einem Slave Teilen. Die Master- und Slave-Anbindung an einen Bus wurde in Abschnitt 2.3.1.1 vorgestellt. In der AutoVision Architektur wird nur der Master des LIS PLB IPIF verwendet, da alle IP Cores Bilddaten über den Hauptspeicher (DDR/DDR2 SDRAM) austauschen. Die Konfigurationsregister der IP Cores werden über den DCR (siehe Abschnitt 2.3.1.3) beschrieben oder ausgelesen. Mit Hilfe eines IPIF (Intellectual Property Interface) können verschiedene IP cores über den PLB miteinander verbunden werden und damit Daten austauschen. Das LIS PLB IPIF wird verwendet, um Daten über den PLB Bus von oder zu einem bestimmten IP Core zu transportieren. Um den PLB möglichst effizient zu nutzen und um unnötige Wartezyklen zu vermeiden, werden über das LIS PLP IPIF Daten mit Hilfe von Bursts mit maximaler Weite von $B_W = 16$ übertragen.

4.2.2.2 Interface zur Anbindung an den Multiport Memory Controller

Neben der Möglichkeit IP cores an den PLB anzubinden, besteht auch die Möglichkeit der direkten Anbindung von IP cores über ein Native Port Interface (NPI) an den Xilinx Multi-Port Memory Controller (MPMC) [Xil09b]. Die Umsetzung der Kontrollsignale von IP Core zu MPMC übernimmt das LIS Native Port IP Interface (LIS NP IPIF). LIS PLB IPIF und LIS NP IPIF besitzen IP-seitig die selben Anschlußpins, was bedeutet, dass IP cores ohne Änderung des HDL Codes (mit Ausnahme eventueller Syntheseattribute zur Anpassung an verschiedene Bus Datenweiten) an beide Interfaces angeschlossen werden können. Da der MPMC Address Pipelining und eine maximale Burstweite B_W von 32 unterstützt, sind über eine NPI Anbindung im Vergleich zu einer Anbindung über das LIS PLB IPIF höhere Datenraten möglich. Über das LIS PLB IPIF und den PLB werden simultane Lese- und Schreibtransfers unterstützt. Dies ist bei Verwendung eines einzelnen NPI nicht möglich. Um gleichzeitig Lesen und Schreiben zu können, müssen in diesem Fall zwei Ports am MPMC verwendet werden. Im Folgenden werden LIS PLB IPIF und LIS NP IPIF unter dem Sammelbegriff IPIF zusammengefasst. Weitere Einzelheiten zum LIS NP IPIF sind in [Alt09a] zu finden.

4.2.2.3 Video Input Core

Um verschiedene Accessory Boards zum Anschluß einer CameraLink Kamera, einer Analog Kamera oder eines DVI Eingangs an die Entwicklungsboards ohne großen Aufwand zu realisieren ist der Video Input Core (VIN) modular aufgebaut. Im Folgenden wird jedoch nur noch die CameraLink Kamera erwähnt, da diese für Prototypenaufbau und Test hauptsächlich eingesetzt wurde. Der VIN core besteht aus 3 Hauptmodulen. Der Input Control Einheit, einem asynchronen FIFO und der Output Control, die in Abbildung 4.5 dargestellt sind.

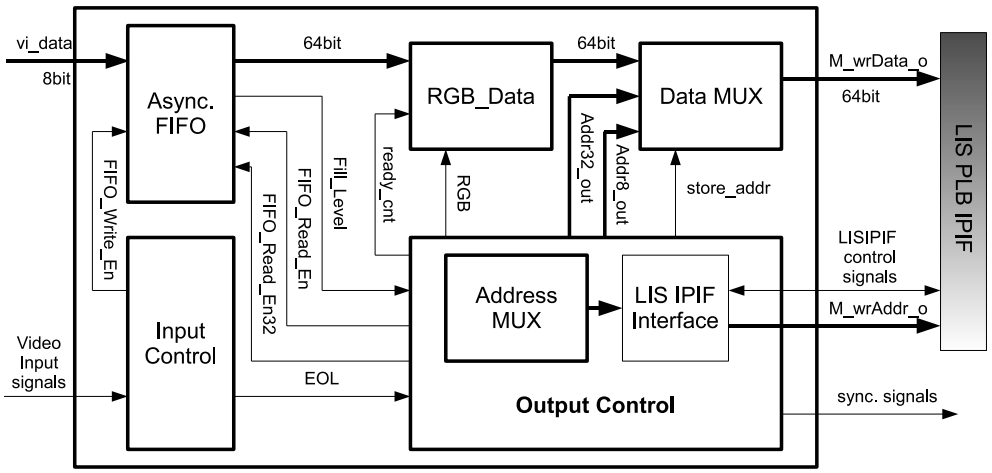


Abbildung 4.5: Video input core

Wie alle eigenen IP cores im AutoVision System lässt sich auch der VIN core über ein IPIF sowohl an PLB als auch direkt an den MPMC anbinden. Die Pixeldaten von der CameraLink Kamera [JAI10] werden mit einer Frequenz von 65 MHz an den VIN geliefert. Aus den Videoframes mit einer Auflösung von 1392×1040 Pixeln wird eine konfigurierbare Region of Interest (ROI) ausgeschnitten. In den Prototypen wurden Bilder mit einer Auflösung von 640×480 Pixeln verarbeitet. Über das Modul Input Control werden die Pixeldaten in das asynchrone FIFO geschrieben. Dabei kann eine ROI definiert werden, die den entsprechenden Bereich aus den Eingangsbilddaten extrahiert, um nur diese in den Hauptspeicher zu übertragen. Die Bilddaten werden so lange im asynchronen FIFO zwischengepuffert, bis genügend Daten für einen Burst der Weite B_W zur Verfügung stehen. Die Übertragung der Daten in den Hauptspeicher findet bei einer Frequenz von 100 MHz statt. Für den sicheren Übergang zwischen den beiden Taktdomänen wird ein asynchrones FIFO verwendet [CA02]. Die Kamera liefert 10-Bit Grauwertpixel, von denen die 8 MSBs von den HWAs weiterverarbeitet werden. Der VIN Core lässt sich einfach eingangsseitig auf 24 bzw. 32 bit Pixel erweitern. Für die farbliche Darstellung am Ausgang sollen 32 Bit Bilder dargestellt werden, in denen erkannte Objekte farblich hervorgehoben werden. Dazu werden die Grauwertdaten im Modul Output Control künstlich in 32-bit Pixeldaten umgewandelt, indem der Grauwert eines Pixels für alle drei Farbkanäle übernommen wird. Der optionale Alphakanal wird dabei auf Null gesetzt. Der Vorgang ist in Abbildung 4.6 dargestellt.
Das Generieren der zusätzlichen Requests für die Farbbilddaten, sowie die entsprechende Adressberechnung für den Hauptspeicher wird ebenfalls vom Modul Output Control übernommen. Mit Hilfe dieses Konzepts lässt sich auch der Anschluß von Farbkameras realisieren. Ein modifiziertes Modul Input Control und eine Farb-zu-Grauwert Konver-

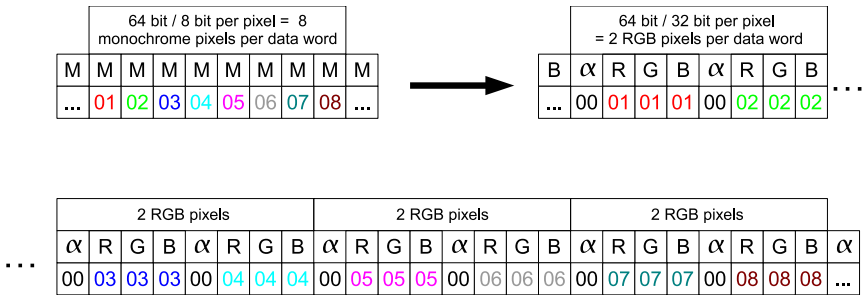


Abbildung 4.6: Umwandlung von 8 Bit Grauwert- in 32 Bit Farbpixel

sion sind dafür ausreichend. Eine CPU kann somit bestimmte Merkmale direkt in den Farbbilddaten kennzeichnen. Der in Abschnitt 4.2.2.4 beschriebene VOUT Core kann dann direkt auf die Farbbilddaten zugreifen und diese am Ausgang darstellen. Weitere Details des VIN Cores sind in [Gat09] beschrieben.

4.2.2.4 Video Output Core

Im AutoVision System wird das DDR/DDR2 SDRAM als Framebuffer verwendet. Zur Darstellung von Bilddaten am Ausgang müssen die Video-Frames aus dem Hauptspeicher zum Video Output (VOUT) Core übertragen werden. Der VOUT Core besteht im Wesentlichen aus vier Komponenten. Einem DCR Interface, einer Input FSM[2], einem asynchronen FIFO und einem DVI/VGA-controller block. Das asynchrone FIFO wird zum temporären Puffern von Pixeldaten verwendet. Der VGA Controller sorgt für das Bereitstellen der Pixeldaten und Steuersignale (HSYNC, VSYNC, blanking interval, etc.) am Ausgang.
Um dem VOUT Core mitzuteilen, von welcher Adresse im Hauptspeicher die 8-Bit Grauwert- oder 32 Bit Farbwertdaten zu lesen sind, wird über das DCR IF die entsprechende Startadresse des Bildes übertragen. Die Verwendung von Grau- oder Farbwertdaten wird über ein Syntheseattribut konfiguriert.
Damit Bilder am Monitor flimmerfrei vom menschlichen Auge wahrgenommen werden, müssen mindestens 60 fps ausgangsseitig dargestellt werden. Bei 31 Eingangsbildern muß also jedes Bild im Schnitt etwa 1,93 mal zum Ausgang übertragen werden. Weitere Details über die Umsetzung des Video Output Cores sind in [Hu08] zu finden. Da bei einem Anschluß des VOUT Cores an den PLB ein großer Teil der Bandbreite für das Darstellen von Bildern verwendet wird, bietet sich in diesem Fall eine Anbindung direkt an den MPMC über ein LIS NP IPIF an. In diesem Fall kann die PLB Bandbreite beispielsweise für HWAs genutzt werden. Alternativ bietet sich auch die Verwendung des in Abschnitt A.2

[2]FSM: Finite State Machine

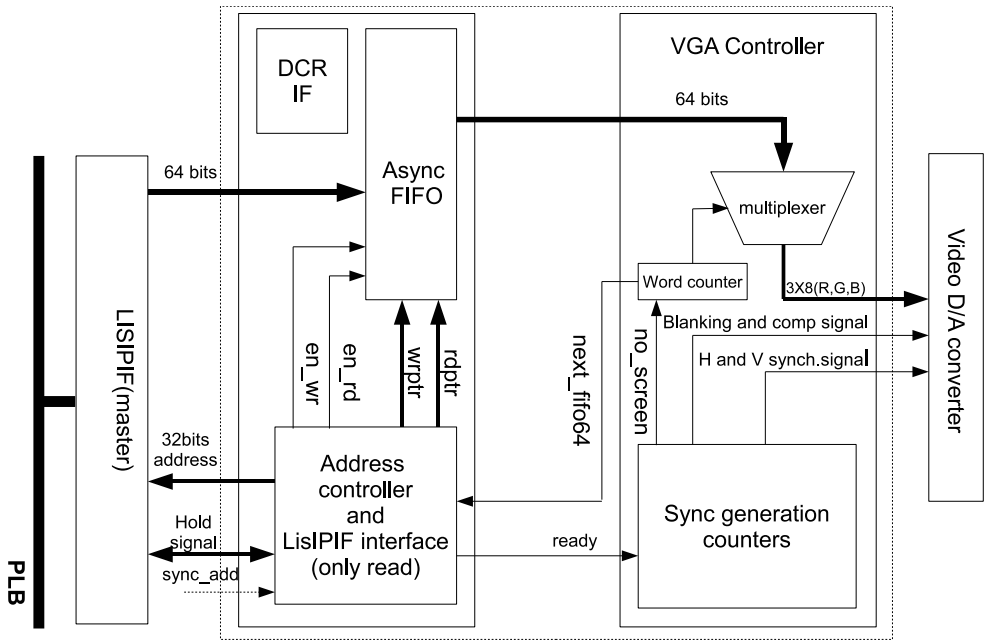
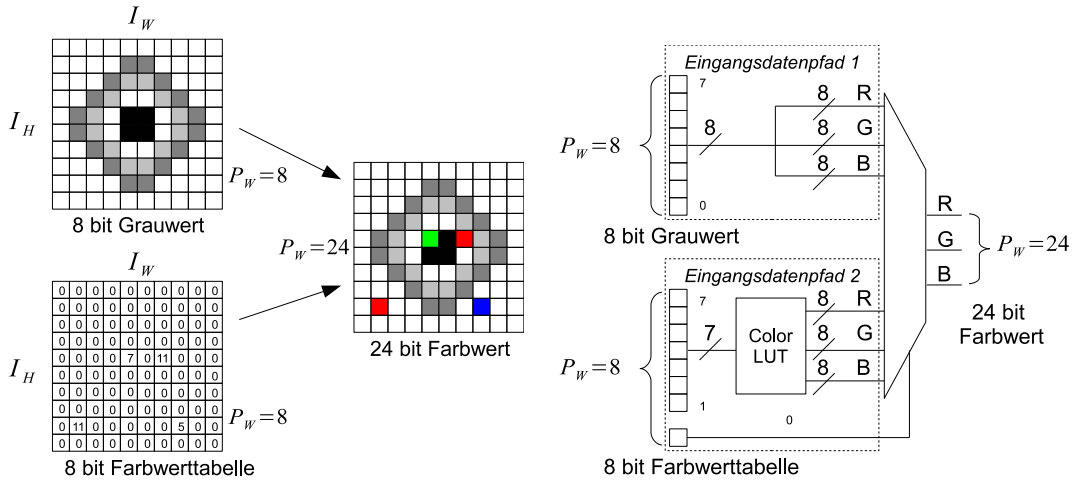


Abbildung 4.7: Video output core

beschriebenen Video Output Boards an. Dieses Board realisiert mit Hilfe von zwei SRAM Speichern das so genannte Doublebuffering von Bilddaten. Bilddaten werden dabei vom FPGA in eines der beiden SRAMs geschrieben, während gleichzeitig die Bilddaten aus dem anderen SRAM dargestellt werden. Anstatt also jedes Bild im Schnitt 1,93 mal aus dem Hauptspeicher zum VOUT Core zu übertragen, werden die Bilddaten nur einmal in eines der beiden SRAMs geladen und von dort im Schnitt 1,93 mal dargestellt.

4.2.2.5 Optimierter Video Output core

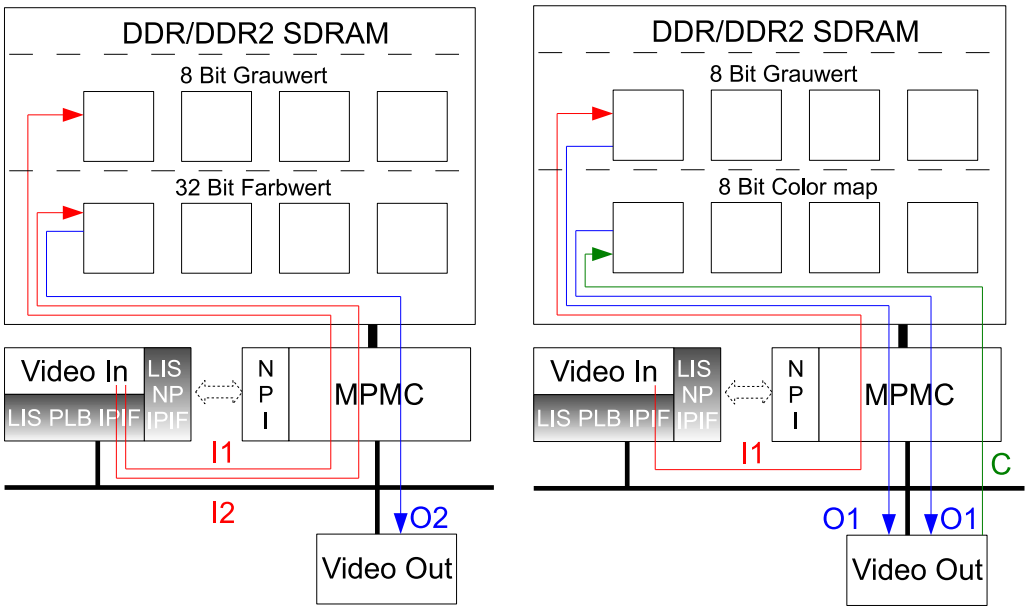
In Abbildung 4.8 sind Konzept und Realisierung eines alternativen VOUT Cores dargestellt. Anstatt die Farbbilddaten direkt zum VOUT zu übertragen, werden bei diesem Konzept die Farben im VOUT aus einem Grauwertbild und einer Farbwerttabelle zusammengesetzt, wie in Abbildung 4.8(a) dargestellt. Zunächst liest der VOUT core sowohl die Grauwertbilddaten, als auch die Farbwerttabelle mit denselben Dimensionen I_W und I_H aus dem Hauptspeicher. Die Pixelweite beträgt bei beiden $P_W = 8$ bit. Die Grauwertpixel werden in unterschiedlichen Eingangsdatenpfaden prozessiert. Wie in Abbildung 4.8(b) zu erkennen ist, wird über eine Multiplexerstruktur entschieden, ob die Grauwertpixel oder ob ein Wert aus einer Color LUT ausgegeben werden soll. Der Selecteingang des



(a) Konzept **(b)** Realisierung

Abbildung 4.8: Alternatives Konzept und Realisierung zur Reduktion der benötigten Bandbreite.

Multiplexers wird durch das Least Significant Bit (LSB = Bit 0) der 8 bit Farbwerttabelle getrieben. Hat das LSB den Wert 1, so wird ein Farbwert aus einer Color LUT übernommen. Ansonsten wird der Wert des Grauwertpixels an alle 3 Farbkanäle geschrieben. Mit den 7 verbleibenden Most Significant Bit (MSBs) der Farbwerttabelle lassen sich $2^7 = 128$ Farben unterscheiden, was mehr als ausreichend für die farbliche Markierung verschiedener Objekte ist. Je nach Wert der eingehenden 7 MSBs wird durch die Color LUT ein vorher festgelegter Farbwert am Ausgang erzeugt. Auch eine 32 Bit Darstellung mit zusätzlichem 8 Bit Alpha Kanal ist mit diesem Konzept möglich. Die Farbwerttabellen werden normalerweise von einer on-chip CPU geschrieben, um beispielsweise Features kenntlich zu machen. Falls kein Farboutput erwünscht ist, kann auf das Laden der Farbwerttabelle auch verzichtet werden, um Bandbreite zu sparen. Die Color LUT kann auch über Konfigurationsregister zur Laufzeit geändert werden. Um zu vermeiden, dass Werte in der Farbwerttabelle aus zeitlich früheren Durchläufen zu einem verfälschten Ausgangssignal führen, muss die Farbwerttabelle 31 mal (aufgrund der Eingangsbildfrequenz) pro Sekunde gelöscht werden. Diese Aufgabe übernimmt der VOUT Core. Bei Erhalt einer Startadresse über den DCR zur Darstellung eines neuen Bildes, wird die zuvor angezeigte Farbwerttabelle vom VOUT Core gelöscht, das heißt mit Nullen überschrieben.



(a) Orginales VOUT Design **(b)** Modifiziertes VOUT Design

Abbildung 4.9: Datenfluß beider Video Output Konzepte

4.2.2.6 Abschätzung des Datenaufkommens durch Pixeltransfers

In Abbildung 4.9 ist jeweils der Datenfluß für die beiden vorgestellten Konzepte dargestellt. I1 bzw. I2 stehen dabei für einen Schreibvorgang von 8 Bit Grauwert- bzw. 32 Bit Farbwertdaten vom VIN core über ein IPIF in den Hauptspeicher. Dementsprechend stehen O1 bzw. O2 für einen Lesevorgang von 8 Bit Grauwert- bzw. 32 Bit Farbwertdaten aus dem Hauptspeicher. Der VOUT core ist für das Löschen der 8-Bit Farbwerttabelle zuständig, was durch den Schreibvorgang C (clear) repräsentiert wird.
Gleichung 4.1 wird lese- sowie schreibseitig für die Berechnung der benötigten Bandbreite verwendet. Dabei repräsentiert F_R die eingangs- bzw. ausgangsseitige Framerate in Hz, I_W und I_H die Bildweite und -höhe und P_W die Weite einzelner Pixel in Bit.

$$I_{BW} = F_R * I_H * I_W * P_W \tag{4.1}$$

Die Kamera liefert 31 Bilder am Eingang. Ausgangseitig werden dagegen 60 Bilder pro Sekunde dargestellt. In Tabelle 4.1 ist die Berechnung des Datenaufkommens für das erste Konzept dargestellt.
Da der PLB über separate Busse für Lese- und Schreibtransfers verfügt, ist eine Aufteilung des Gesamtdatenaufkommens sinnvoll. Im Gegensatz zu dem in Abbildung 4.9(a)

Schreibtransfer von 31 Grauwert-Bildern pro Sekunde vom VIN in den Hauptspeicher (I1)	I_{BW}(I1)=31 Hz*8 Bit*I_W*I_H
Schreibtransfer von 31 Farb-Bildern pro Sekunde vom VIN in den Hauptspeicher (I2)	I_{BW}(I2)=31 Hz*32 Bit*I_W*I_H
Lesetransfer von 60 Farb-Bildern pro Sekunde aus dem Hauptspeicher zum VOUT (O2)	I_{BW}(O2)=60 Hz*32 Bit*I_W*I_H
Schreibseitiges Gesamtdatenaufkommen:	I_{BW}(I1+I2)=1240 Bit/s*I_W*I_H
Leseseitiges Gesamtdatenaufkommen:	I_{BW}(O2)=1920 Bit/s*I_W*I_H

Tabelle 4.1: Datenaufkommen des originalen VOUT Konzepts

dargestellten Konzept, werden bei dem alternativen Konzept (s. Abbildung 4.9(b)) nur noch Grauwertdaten vom VIN in den Hauptspeicher übertragen (I1). Der VOUT Core lädt dann als Bus-Master selbständig die von der Dimension her identischen Grauwert Bildddaten (O1) und die Farbwerttabelle (O1). Anschließend sorgt der VOUT Core für das Überschreiben der gelesenen Farbwerttabelle mit Nullen. Die Berechnung des Datenaufkommens für dieses Konzept ist in Tabelle 4.2 dargestellt.

Schreibtransfer von 31 Grauwert-Bildern pro Sekunde vom VIN in den Hauptspeicher	I_{BW}(I1)=31 Hz*8 Bit*I_W*I_H
Lesetransfer von 60 Grauwert-Bildern pro Sekunde aus dem Hauptspeicher zum VOUT	I_{BW}(O1)=60 Hz*8 Bit*I_W*I_H
Lesetransfer von 60 Farbwerttabellen pro Sekunde aus dem Hauptspeicher zum VOUT	I_{BW}(O1)=60 Hz*8 Bit*I_W*I_H
Schreibtransfer zum Löschen von 31 Farbwerttabellen pro Sekunde	I_{BW}(C)=31 Hz*8 Bit*I_W*I_H
Schreibseitiges Gesamtdatenaufkommen:	I_{BW}(I1+C)=488 Bit/s*I_W*I_H
Leseseitiges Gesamtdatenaufkommen:	I_{BW}(O1+O1)=960 Bit/s*I_W*I_H

Tabelle 4.2: Datenaufkommen des modifizierten VOUT Konzepts

Ein Vergleich der beiden Konzepte zeigt, dass das zweite Konzept zu einem reduzierten Datenaufkommen führt. Bei Verwendung einer Grauwertkamera kann mit dem zweiten Konzept schreibseitig rund 60% und leseseitig 50% an Bandbreite eingespart werden. Jedoch besitzen beide Konzepte ihre Vor- und Nachteile. Mit Hilfe des ersten Konzepts können auch Farbkameras verwendet werden. Innerhalb des VIN Cores werden dann die RGB Daten in Grauwerte umgewandelt und zusammen mit den Farbbilddaten in den Hauptspeicher übertragen. Die Umwandlung der Farbbilddaten in Grauwerte ist notwendig, da die HWAs auf diesen Grauwerten arbeiten.

4.2.3 Videodatenfluß

Die Bilddaten werden von der Camera-Link Kamera [JAI10] an den VIN Core geliefert. Innerhalb des VIN Cores werden die Pixeldaten zwischen gepuffert, bis genügend Daten für einen Full Burst zur Verfügung stehen. Mit Hilfe dieser Burst Transfers über den PLB lassen sich die Daten effizient in den Hauptspeicher übertragen. Sobald ein Bild vollständig in den Speicher geschrieben wurde, wird der PowerPC über einen Interrupt benachrichtigt. Nach der Verarbeitung des Interrupts startet der PowerPC den HWA über den DCR BUS, indem er die Startadresse des zu verarbeitenden Bildes im Hauptspeicher und eventuelle Konfigurationssignale übergibt. Sobald der HWA dann richtig konfiguriert wurde, initiiert er als Bus Master selbstständig DMA Lesetransfers. Dadurch wird die CPU nicht in den Datentransfer involviert und ist während der Verarbeitungszeit in Hardware im Idle Zustand und könnte daher andere Aufgaben übernehmen.
Die Verarbeitung der Pixeldaten erfolgt dann mit einem HWA, der aus den in Abschnitt 4.3 beschriebenen Modulen zusammengesetzt ist. Die verarbeiteten Daten werden dann im Ausgangspfad der Engine zwischen gespeichert, bis genügend Pixel für einen Full Burst zur Verfügung stehen, die dann wiederum über DMA Transfers zurück in den Hauptspeicher übertragen werden. Die Meldung, dass die letzten Daten aus dem HWA in den Hauptspeicher übertragen wurden, erfolgt erneut durch einen Interrupt an den PowerPC.

4.3 Eine Modulbibliothek für generische Pixelverarbeitungspipelines

Die generische Pixelprozessierungspipeline basiert auf dem in [Her05] vorgestellten Konzept der AddressEngine. Dieses Konzept wurde auf eine modulare Architektur erweitert, um eine Vielzahl an verschiedenen Bildverarbeitungsalgorithmen realisieren zu können. Im Gegensatz zu dem in [Her05] präsentierten abstrakten Modell, handelt es sich bei der Pixelverarbeitungspipeline um ein realisiertes Konzept, auf dem zahlreiche HW Beschleuniger im AutoVision System basieren. Die erste Implementierung der AddressEngine in Basiskonfiguration und den dazugehörigen Modulen ist in [Zep05] beschrieben. Details über eine optimierte Version sind hingegen in [Jia08a] zu finden.
Abbildung 4.10 zeigt ein Blockschaltbild der Architektur der Pixelverarbeitungspipeline. In der Basiskonfiguration besteht diese Pipeline aus 7 verschiedenen Modulen. Den *Konfigurationsregistern*, der *InputFSM*, dem *Local Input Memory (LIM)*, der *Matrix*, der *User Logic*, dem *Local Output Memory (LOM)* und der *OutputFSM*. Diese Module bilden einen IP core zur Verarbeitung von Pixeldaten. Über ein (*IPIF*) wird die Verbindung der HWAs an einen PLB Bus oder direkt an einen MPMC realisiert. Ziel dieser Architektur ist die Zusammenstellung der Pipeline aufgrund der gewünschten Bildverarbeitungsoperation. Da die HWAs als Bus-Master selbstständig Pixeltransfers über DMA initiieren,

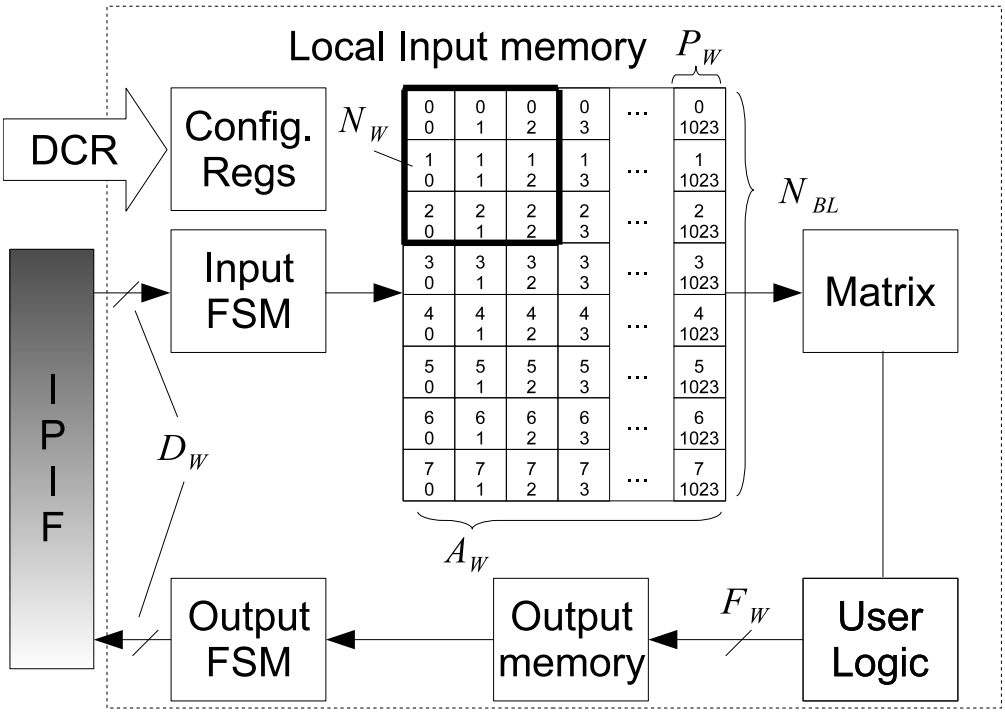


Abbildung 4.10: Blockdiagramm einer modul-basierten Pixelverarbeitungspipeline mit 8 kompletten Bildzeilen im lokalen Eingangsspeicher

kann sich der Anwender rein auf die Implementierung der Bildverarbeitungsoperation und die Konfiguration der Einzelmodule beschränken. Das rechtzeitige und bandbreitenschonende Transferieren von Pixeldaten von und zum HWA aus dem Hauptspeicher soll dabei gewährleistet sein, und die gepipelinte Verarbeitung nicht unterbrechen. Im Folgenden werden die 7 Einzelmodule kurz beschrieben.

Über die *Konfigurationsregister* lassen sich bestimmte Parameter, wie beispielsweise die Größe der ROI ändern. Die Konfigurationsregister werden im AutoVision System über den DCR Bus beschrieben.

Das Modul *InputFSM* ist für das Lesen und das temporäre Speichern der Pixeldaten im *Local Input Memory (LIM)* zuständig. Dies beinhaltet das Senden von Requests und die Adressberechnung der zu lesenden Daten im Hauptspeicher.

Das Modul *LIM* wird als lokaler Speicher für die von der Bildverarbeitungsoperation gerade benötigten Pixeldaten verwendet. Hier werden komplette Bildzeilen zwischengespeichert.

Die *Matrix* wird in einem Scanmodus von links nach rechts und oben nach unten über

das Bild geschoben. Dieses Modul stellt sicher, dass in jedem Taktzyklus innerhalb einer Bildzeile ein zentrales Pixel mit seiner Nachbarschaft der *User Logic* zur Verfügung steht. Die *User Logic* ist eine anwenderspezifizierte Pixel Operation auf ein zentrales Pixel und seine Umgebung, die SIMD Operationen realisiert. Die verarbeiteten Pixel werden im *Local Output Memory (LOM)* zwischengespeichert. Sobald genügend Daten für einen Burst zur Verfügung stehen, werden diese in den Hauptspeicher übertragen. Letztendlich ist die *OutputFSM* für das Initiieren des Schreibtransfers in den Hauptspeicher an die korrekte Adresse verantwortlich. In den folgenden Unterabschnitten werden die eben beschriebenen und weitere Module der Pixelverarbeitungspipeline detailliert erläutert.

4.3.1 Zur Designzeit adaptive Parameter (Generics)

Um maximale Flexibilität zu gewährleisten ist der VHDL Code generisch gehalten. Das bedeutet, dass sich durch Setzen verschiedener Syntheseattribute, Signalbreiten, Dimensionen von lokalen Speichern etc. ohne Änderung des eigentlichen VHDL Codes ändern lassen. In VHDL werden diese Syntheseattribute durch die so genannten Generics realisiert. Die Generics müssen zur Designzeit bekannt sein, da jede Änderung dieser Parameter eine neue Platzierung und Verdrahtung nach sich zieht. In Abbildung 4.10 ist beispielhaft eine Pixelprozessierungspipeline bestehend aus 7 Modulen dargestellt. In dieser Abbildung sind auch die Generics und ihr Einfluß auf die Architektur der Pipeline zu erkennen. Die maximale Bildweite I_W im Hauptspeicher wird durch die Adressweite A_W bestimmt. Die Bildhöhe I_H wird dann nur durch den reservierten Speicherbereich begrenzt. Um dem HWA mitzuteilen, wie lang eine Bildzeile maximal sein kann wird A_W verwendet. Die maximale Bildgröße, die im Moment durch die Grundeinstellungen unterstützt wird ist 1024 x 1024 Pixel ($A_W = 10$) . Eine maximale Auflösung von 2048 x 2048 kann verwendet werden, wenn der Parameter A_W auf 11 ($2^{11} = 2048$) geändert wird. Dementsprechend müssen allerdings auch die Daten im Hauptspeicher angeordnet sein. Die Datenweite D_W entspricht der Busbreite. Der HWA muß theoretisch jeden Takt Daten der Weite D_W annehmen und weiterverarbeiten können. Die Anzahl an Bildzeilen im LIM wird durch den Parameter N_{BL} bestimmt. Abhängig von diesem Parameter ist die maximale Größe der Nachbarschaftsweite N_W. N_W beschreibt wie viele Pixel zwischen zentralem Pixel und der Matrixgrenze liegen. Bei einem einzelnen Pixel ist N_W=0, bei einer 3×3 Matrix ist $N_W = 1$, bei einer 5×5 Matrix ist $N_W = 2$ usw. Ist $N_{BL} = 8$ kann N_W maximal 3 sein (7×7 Matrix). Die unterstützten Pixel-Farbtiefen, im Folgenden als Pixelweite (P_W) bezeichnet, sind Zweierpotenzen. Die gängigen im AutoVision System auch verwendeten P_Ws sind 8, 16, 32 oder 64 Bit. 8 Bit pro Pixel (bbp) werden für die Grauwertedarstellung eines Bildes verwendet. Im 16 Bit Pixelformat werden die drei Grundfarben Rot, Grün und Blau (RGB) mit 5-6-5 Bit dargestellt. Es ist auch möglich dieses Format für die Grauwertrepräsentation eines Bildes mit zusätzlichem 8-Bit Kanal für Transparenz, dem so genannten α Kanal zu nutzen. Das 24-Bit RGB Format kann ebenfalls um diesen Transparenzkanal erweitert werden, was dann in einer P_W von

32 resultiert. Falls keine Transparenz benötigt wird, bleibt dieser Kanal unbenutzt. Eine P_W von 64 kann verwendet werden, um zusätzliche Daten, neben der Farbinformation eines Pixels zu übertragen. An dieser Stelle sei darauf hingewiesen, dass die Daten, die vom HWA gelesen bzw. geschrieben werden nicht notwendigerweise Pixel sein müssen. In vielen Fällen werden zwar Bilddaten vom HWA gelesen, das Resultat ist aber eine Liste von Features. Die Größe eines Features, im Folgenden als Featureweite (F_W) bezeichnet, kann je nach enthaltener Information von HWA zu HWA unterschiedlich sein. B_W bestimmt die Burstweite, also die Anzahl an Wörtern mit Datenweite D_W. In einem Burst der Weite B_W=16 auf einem 64 Bit breiten Bus, können 16 × 64 Bit = 128 Byte übertragen werden. Eine Zusammenfassung aller Generics, die in der Pixelprozessierungspipeline gewählt werden können, ist in Tabelle 4.3 dargestellt.

Symbol	Bedeutung	Range	Description
A_W	Adress-weite	10	Bitweite der Koordinatenzähler innerhalb eines Bildes. Bei A_W =10 wird eine maximale Auflösung von 1024 × 1024 Pixeln unterstützt.
D_W	Daten(Bus)-weite	64	Legt die Weite des Dateneingangs und -ausgangs fest. Entspricht im Normalfall der Busbreite.
P_W	Pixelweite	[8, 64]	Pixelweite oder Farbtiefe der Pixel. Kann als jede Zweierpotenz konfiguriert werden, um Pixelweiten zwischen 8 und 64 Bit zu realisieren.
N_W	Nachbar-schaftsweite	[0, 7]	Legt die Nachbarschaftsweite fest. Die Matrixgröße kann zwischen 1 × 1 und 15 × 15 ausgewählt werden.s
N_{BL}	Anzahl an Bufferlines	[2, 16]	Anzahl an Bufferlines im *LIM*. Als Zweierpotenz konfigurierbar zwischen 2 und 16.
B_W	Burstweite	[2, 32]	Anzahl an DWords (64 Bit) innerhalb eines Bursts.
F_W	Featureweite	[8, 64]	Definiert die Featureweite.

Tabelle 4.3: Liste der verfügbaren Generics

4.3.2 Konfigurationsregister und Adressierung

Um die Flexibilität des HWAs zu erhöhen sind diverse Addressierungsparameter, die in Abbildung 4.11 dargestellt sind, zur Laufzeit adaptiv. Diese Parameter sind in *Konfigurationsregistern* gespeichert. Im Gegensatz zu den zur Designzeit adaptiven Parametern (Generics) bedarf eine Änderung der *Konfigurationsregister* keiner erneuten Synthese. Diese Parameter können beispielsweise dazu verwendet werden eine Region of Interest (ROI) innerhalb des Bildes zu definieren. Pixelwerte, die bei einer Sliding Window Operation außerhalb der ROI liegen, könnten zusätzlich geladen werden. Um wie viele Pixel

die ROI erweitert werden muß, wird durch die Nachbarschaftsweite N_W bestimmt.

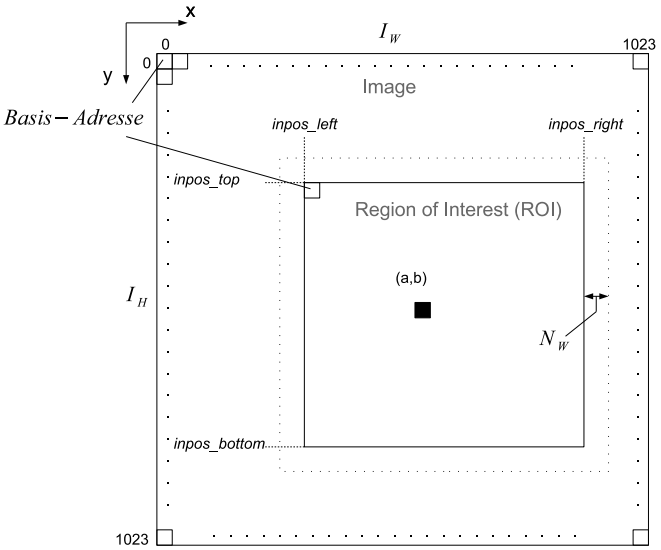


Abbildung 4.11: Parameter, die zur Laufzeit über Register konfigurierbar sind

Der Hauptspeicher ist Byte addressierbar, was bedeutet, dass jeweils 8 Bit adressiert werden können. Daher kann theoretisch durch eine bestimmte Speicheradresse auf jedes einzelne Pixel zugegriffen werden, wenn P_W größer oder gleich 8 Bit ist. Die erste Adresse im Hauptspeicher repräsentiert die Basis-Adresse. Die Basis-Adresse ist im Normalfall das erste Pixel (obere linke Ecke) des Eingangsbildes. Mit Hilfe der Basis-Adresse kann die Adresse jedes Pixel im Bild nach Gleichung 4.2 berechnet werden. Die Variablen a und b repräsentieren die x- bzw. y-Koordinate eines Pixels innerhalb des Bildes. Da ein Pixel aufgrund von P_W aus mehreren Bytes bestehen kann, ist auch die Adressberechnung abhängig von der Pixelweite P_W in Bytes. Basis-Adresse des Eingangs- und Ausgangsbildes können über *Konfigurationsregister* festgelegt werden.

$$Address = (b * I_W + a) * P_W[Byte] + BasisAdresse \tag{4.2}$$

In Tabelle 4.4 ist ist die beispielhafte Adressierung eines Pixels bei verschiedenen Werten von P_W dargestellt. Anhand von Gleichung 4.2 läßt sich nun auch erkennen, warum es sinnvoll ist, für P_W nur Zweierpotenzen zu verwenden. Die Multiplikation mit Zweierpotenzen kann durch Shift-Operationen in HW realisiert werden. Beispielsweise hat ein Pixel mit den Koordinaten (x, y) und einer Pixelweite P_W von 4 Byte nach Gleichung 4.2 die Adresse $(y * 1024 + x) * 4 + BasisAdresse$. Ein Faktor von 1024 in Gleichung 4.2 resultiert in einer Shift-Operation von $log_2(1024) = 10$ Bits nach links. Dementsprechend wird durch die Multiplikation mit 4 auch der Ausdruck $(y * 1024 + x)$ 2 Bits nach links

geschoben und anschließend zur Basis-Adresse addiert. Mit dieser Methodik ist es möglich die Adressberechnung in HW sehr ressourcenschonend ohne die Verwendung von Multiplizierern durchzuführen.

I_W	I_H	P_W	Pixel Koord.	Pixel Addresse im Speicher
1024	1024	8	(128, 1)	Basis-Adresse + $0x480$
1024	1024	16	(128, 1)	Basis-Adresse + $0x900$
1024	1024	32	(128, 1)	Basis-Adresse + $0x1200$
1024	1024	64	(128, 1)	Basis-Adresse + $0x2400$

Tabelle 4.4: Beispielhafte Adressierung eines Pixels bei verschiedenen Pixelweiten P_W

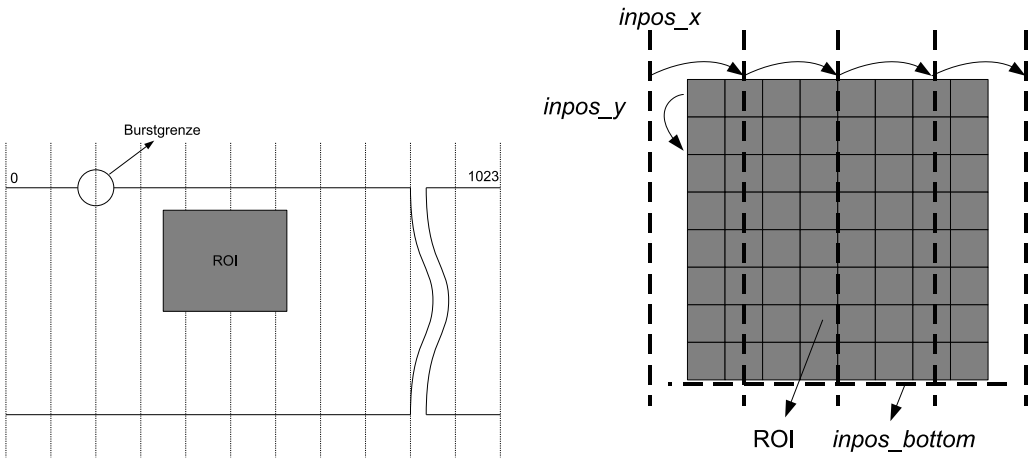
Die Pixeldaten werden über Direct Memory Access (DMA) vom Hauptspeicher zum HWA transferiert. Nachdem der HWA die Basis-Adresse des zu verarbeitenden Bildes erhalten hat, holt dieser sich selbstständig die entsprechenden Daten, verarbeitet sie und schreibt sie zurück in den Hauptspeicher. Neben der Verarbeitung von ganzen Bildern wird auch die Verarbeitung einer ROI unterstützt. Die Größe und Position der ROI innerhalb des Bildes ist über die Parameter inpos_left, inpos_right, inpos_top und inpos_bottom zur Laufzeit konfigurierbar. Für den Fall, dass die gewählte ROI an eine andere Stelle im Bild zurück geschrieben werden soll, stehen weitere Konfigurationsregister (outpos_left, outpos_right, outpos_top und outpos_bottom) zur Verfügung. Zusätzlich sind weitere zur Laufzeit änderbare Parameter (adaptive Thresholds, zuschalten zusätzlicher Funktionalität etc.), abhängig von der Funktion eines HWA denkbar, die in weiteren *Konfigurationsregistern* realisiert werden.

4.3.3 Input FSM

Wie bereits erwähnt, ist das Modul *InputFSM* hauptsächlich für das Lesen und temporäre Ablegen der Bilddaten im *LIM* verantwortlich. Das Abscannen des Eingangsbildes wird durch zwei Finite State Machines (FSMs) realisiert. Einer Request State Machine (RSM) und einer Data State Machine (DSM). Die RSM ist für die Iteration über die Bildzeilen zuständig, während die DSM die Iteration über die Pixel innerhalb einer Zeile übernimmt.

4.3.3.1 Request State Machine (RSM)

Die RSM ist zusätzlich für das Senden von Requests und die erforderliche Adressberechnung für das Laden der Pixel aus dem Hauptspeicher zuständig.
Das Bild im Speicher kann in eine konfigurierbare Anzahl an Bursts mit der Länge B_W (typischerweise 16 (PLB) oder 32 (NPI)) unterteilt werden. Ein Burst der Weite $B_W = 16$ bedeutet, dass 16 Pakete mit Datenweite D_W vom Hauptspeicher zu dem anfordernden IP core übertragen werden. Die grau markierte ROI in Abbildung 4.12(a) zeigt einen



(a) Aufteilung eines Bildes mit ROI aufgrund von Burstgrenzen

(b) Die beiden Zähler der Request State Machine

Abbildung 4.12: Unterteilung des Bildes anhand von Burstgrenzen und Zähler der Request State Machine

normalen Fall wenn linke und rechte Grenze der ROI nicht auf Burstgrenzen[3] fallen. Für das Bursten und die entsprechende Adressgenerierung sind zwei Zähler innerhalb der RSM verantwortlich. Einer dieser Zähler ist inpos_x, der für die Pixelposition in x-Richtung verwendet wird. Dieser Zähler springt von einer Burstgrenze zur nächsten. Dieses Vorgehen ist in Abbildung 4.12(b) dargestellt. Die Anzahl an Pixeln pro Burst PPB ist abhängig von P_W und kann nach Gleichung 4.3 berechnet werden.

$$PPB = \frac{B_W[Dwords/Burst] * D_W[Bits/Dwords]}{P_W[Bits/Pixel]} \tag{4.3}$$

Um von einer Burstgrenze zur nächsten zu springen wird inpos_x um PPB erhöht. Sobald die rechte Grenze der ROI erreicht ist wird inpos_x zurück an die Startposition gesetzt, die immer einer Burstgrenze entspricht. Dies führt im Normalfall dazu, dass auch nicht benötigte Pixel in das *LIM* geladen werden. Diese haben jedoch keine Auswirkung auf das Ergebnisbild, da sie innerhalb des HWAs nicht verarbeitet werden. Jedoch vereinfacht dieses Konzept die Adressberechnung. Der Zähler inpos_y gibt die aktuelle y Position innerhalb des Bildes an und wird Zeile für Zeile um eins erhöht bis das Ende (inpos_bottom) der ROI erreicht ist. Dies ist ebenfalls in Abbildung 4.12(b) dargestellt. Mit den beiden Parametern inpos_x und inpos_y kann die Startadresse für den nächsten

[3] Burstgrenzen sind spezielle Stellen im Speicher ab denen ein Burst maximaler Länge gestartet werden kann.

Burst berechnet werden. Die Adressen bestehen aus 32 Bit. Die von der RSM generierten Zähler inpos_x und inpos_y repräsentieren die Koordinaten und damit die Pixelposition. Die Zusammensetzung der Adresse für verschiedene P_W ist in Abbildung 4.13 illustriert und kann nach Gleichung 4.3 mit Hilfe von Shift-Operationen realisiert werden. Dies geschieht indem die letzten $(log2(P_W) - 3)$ Bit abgeschnitten werden. Somit kann auf teure Divisionen in Hardware verzichtet werden.

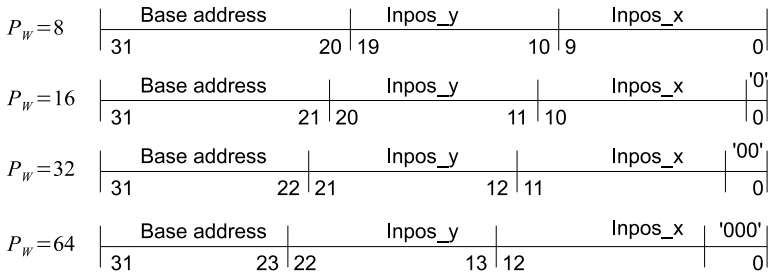


Abbildung 4.13: Adresszusammensetzung für verschiedene Werte von P_W

4.3.3.2 Data State Machine

Nach einer akzeptierten Lese-Anforderung (Read Request) müssen die ankommenden Daten in das *LIM* geschrieben werden, was durch die DSM geregelt wird. Diese iteriert über die Pixel innerhalb einer Zeile, oder genauer über die Pixel innerhalb eines Bursts. Darüber hinaus ist die DSM für die Berechnung der Pixeladressen im *LIM* zuständig.

4.3.4 Local Input Memory (LIM)

Die Hauptaufgabe des *Local Input Memory (LIM)* ist das temporäre Speichern von Pixeldaten, um das mehrmalige Laden von Pixeln aus dem Hauptspeicher zu vermeiden. Das *LIM* ist somit als kleiner lokaler Speicher zu verstehen, der verwendet wird, um zeitraubende Zugriffe aus dem Hauptspeicher zu vermeiden. Aufgebaut ist das *LIM* aus einzelnen so genannten Bufferlines, die durch BRAMs realisiert werden. Eine einzelne Bufferline ist für das Speichern einer kompletten Bildzeile des Eingangsbildes zuständig. Durch die Aufteilung in verschiedene Bufferlines ist es möglich, simultan ein Pixel aus jeder Bildzeile zu laden. Damit kann, wie in Abbildung 4.1 angedeutet, eine komplette Spalte der nachfolgenden *Matrix* pro Taktzyklus geladen werden. Die Anzahl an Bufferlines N_{BL}, und somit die Größe des *LIM*, wird über Generics festgelegt. Das *LIM* wird verwendet, um das in Abschnitt 4.1.1 beschriebene Konzept der lokalen Speicher umzusetzen.

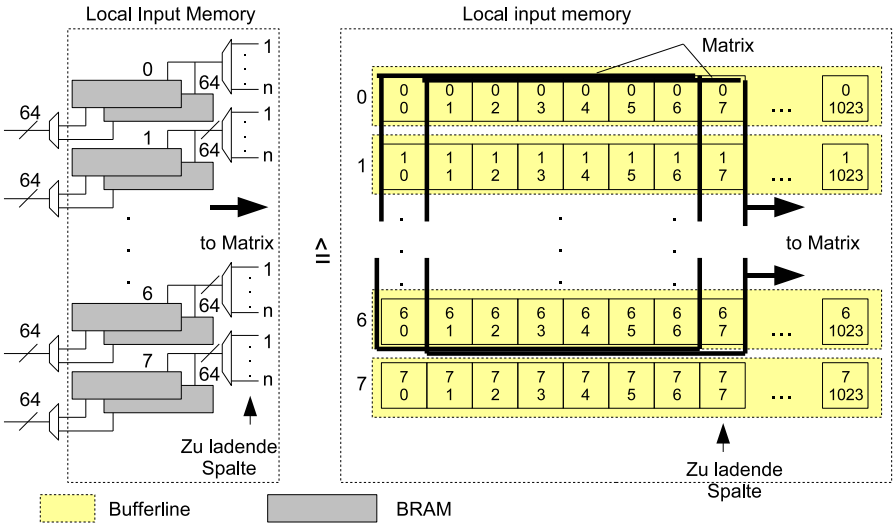


Abbildung 4.14: Local Input Memory konfiguriert für 8 Bufferlines

Nachdem der Aufbau festgelegt ist, stellt sich die Frage nach Dimensionierung und Befüllung des *LIM*. Der naive Ansatz, die kompletten Bilddaten in das on-chip BRAM zu laden, um die Zugriffszeit zu verkürzen ist aufgrund mangelnder Ressourcen nur schwer realisierbar. Die Kapazität eines BRAMs in einem V2P oder V4 Device ist $\Gamma = 512x32$ Bit, ohne Beachtung der Parity Bits (mit Beachtung wären 512x36 Bit möglich). Auf V5 steht die doppelte Größe zur Verfügung. Für das Speichern eines Bildes mit einer Auflösung von 640x480 Pixel und einer Pixelweite von $P_W = 8$ Bit im on-chip BRAM benötigt man $\lceil \frac{640x480x8Bit}{\Gamma} \rceil = 150$ BRAMs. Dies übersteigt die Menge an verfügbaren BRAMs auf beispielsweise dem verwendeten XCV2P30 Device, auf dem nur 136 BRAMs zur Verfügung stehen. Anstatt die gesamten Bilddaten also in BRAMs abzulegen, wird ein gepipelinter Ansatz verwendet.
Aus dem Hauptspeicher werden die Bilddaten via DMA Transfers in die Bufferlines des *LIM* geladen. Die Anzahl an Bufferlines N_{BL} ist im Normalfall wesentlich geringer als die Bildhöhe I_H. Nur die Bilddaten, die für die Verarbeitung eines zentralen Pixel und seiner Nachbarschaft notwendig sind, werden im *LIM* temporär gespeichert. Ein Beispiel ist in Abbildung 4.15 dargestellt.
Abbildung 4.15 zeigt ein *LIM* mit $N_{BL} = 8$ Bufferlines. Die Nachbarschaft um das zentrale Pixel soll eine Größe von 7×7 haben. Die aktuelle 4. Bufferline wird durch das zentrale Pixel bestimmt. Der Scanmodus erfolgt zeilenweise von links nach rechts und von oben nach unten. Wenn also das letzte Pixel der 4. Zeile verarbeitet wurde (kurz bevor inpos_right erreicht ist), wird das zentrale Pixel auf das erste Pixel der fünften Zeile verschoben (welches durch inpos_left definiert wird). Dadurch, dass die 8.

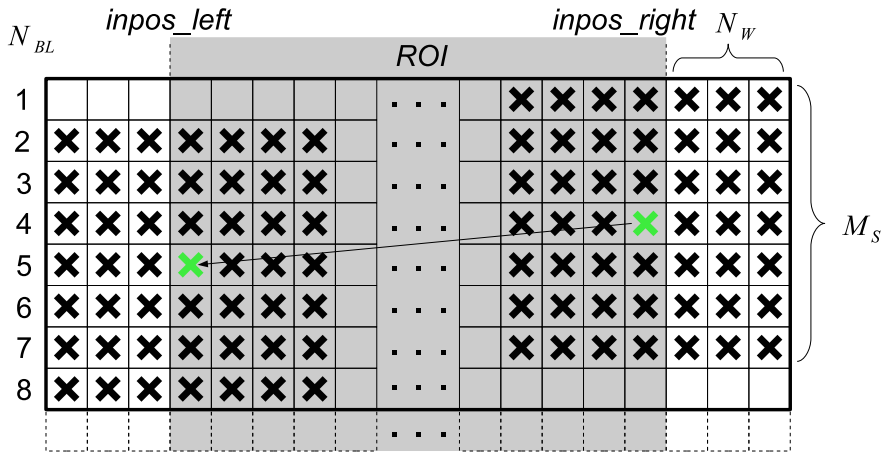


Abbildung 4.15: Notwendigkeit einer zusätzlichen Bufferline beim Verschieben der *Matrix* an den linken Rand der ROI

Bufferline bereits geladen war, kann die 7×7 Nachbarschaft sequentiell über alle Pixel der 5. Zeile geschoben werden. Während der Prozessierung der Pixel in der fünften Zeile kann gleichzeitig die nächste Zeile (im Beispiel die Neunte) nachgeladen werden. Um also die Pipeline durch Nachladen von Pixeln nicht unnötig lange zu blockieren, muß die Anzahl an Bufferlines N_{BL} größer als die Matrixgröße M_S sein. M_S setzt sich aus der Nachbarschaftsweite N_W wie folgt zusammen: $M_S = (2N_W + 1) \times (2N_W + 1)$. In Tabelle 4.5 ist daher der Zusammenhang zwischen N_{BL}, M_S und N_W dargestellt ist.

Matrixgröße M_S	Nachbarschaftsweite N_W	# an Bufferlines N_{BL}
1 X 1	0	2
3 X 3	1	4
5 X 5	2	8
7 X 7	3	
9 X 9	4	16
11 X 11	5	
13 X 13	6	
15 X 15	7	

Tabelle 4.5: Abhängigkeit zwischen Matrixweite und Anzahl an Bufferlines in *LIM*

4.3.5 Matrix

Die *Matrix* stellt der *User logic* pro Taktzyklus ein zentrales Pixel mit entsprechender Nachbarschaft zur Verfügung. Im Gegensatz zum *LIM* verwendet die *Matrix* Register für das Speichern der Pixel. Diese Designentscheidung beruht auf der Notwendigkeit, auf die gesamten Pixeldaten (zentrales Pixel und Umgebung), die von der *User Logic* benötigt werden in einem Taktzyklus zuzugreifen. Das Modul *Matrix* besteht aus einzelnen Matrixelementen, in denen jeweils ein Pixel gespeichert ist. Die Anzahl an verwendeten Registern pro Matrixelement ist hierbei abhängig von P_W. Daher hat die *Matrix* den höchsten Verbrauch an Registern unter allen Modulen. Beim Vorrücken der *Matrix* um eine Position, muß nur eine Spalte mit Pixelwerten geladen werden. Die anderen Matrixelemente können durch Shifts aus den vorherigen Matrixelementen übernommen werden, wie in Abbildung 4.16 dargestellt.

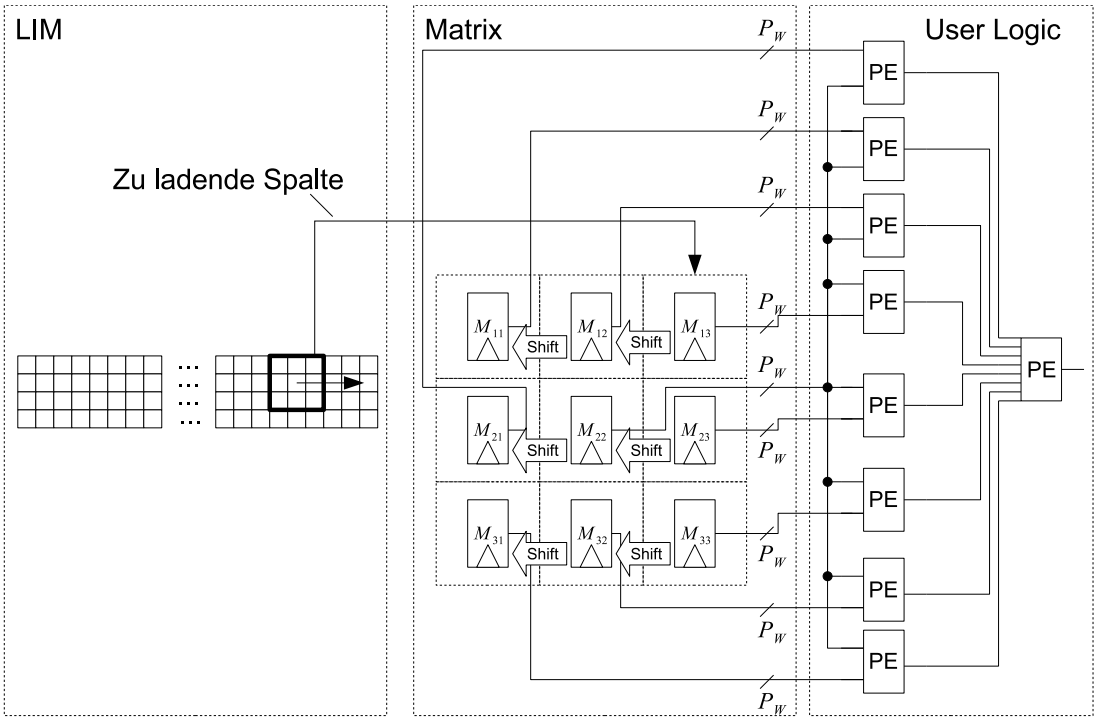


Abbildung 4.16: Aufbau und Funktionsweise der *Matrix*

Der Scan-Modus muß auf die im *LIM* liegenden Daten abgestimmt sein. Im Beispiel in Abbildung 4.16 ist dieser Fall anhand einer 3×3 Nachbarschaft dargestellt. Aus dem *LIM* wird eine neue Pixelspalte extrahiert und in die Matrixelemente M_{13}, M_{23} und M_{33} geladen. Gleichzeitig werden die Werte aus dem Matrixelement M_{13} in Matrixelement

M_{12} und Matrixelement M_{12} in Matrixelement M_{11} übernommen. Die Übernahme durch Shiftoperationen in die anderen Register funktioniert analog. In Abbildung 4.17 ist die Struktur einer 5×5 Matrix dargestellt.

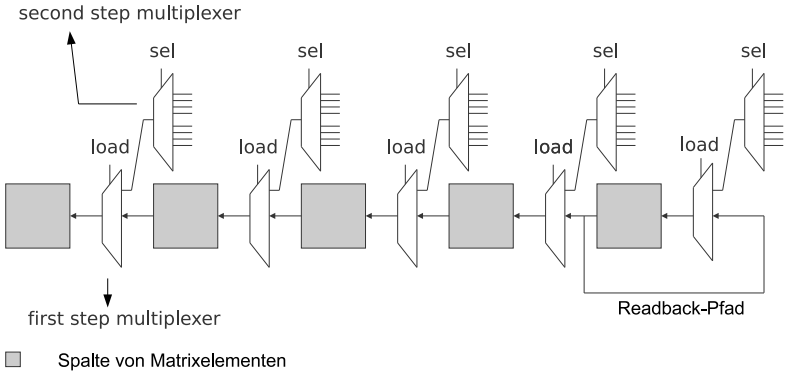
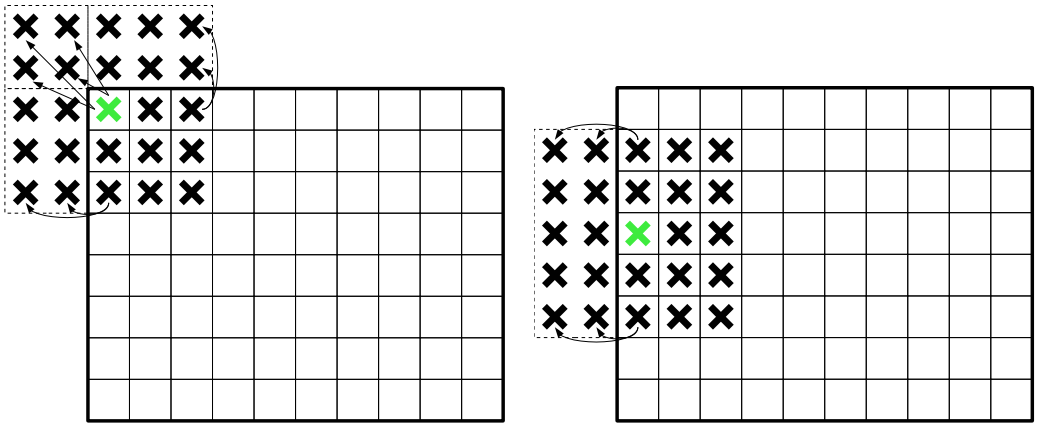


Abbildung 4.17: Struktur einer *Matrix* mit einer Größe von $M_S = 5 \times 5$

Ein grau gefärbtes Quadrat in Abbildung 4.17 repräsentiert eine Spalte von Matrixelementen mit zwei kaskadierten Multiplexern. Das Select Signal des ersten Multiplexers „load“ wird verwendet, um zu entscheiden, ob das Matrixelement eine Spalte aus dem *LIM* laden soll, oder ob eine Kopie der Spalten aus den Matrixelementen der rechten Seite gemacht werden soll. Das Signal des zweiten Multiplexers „sel“ entscheidet aus welchen Bufferlines die Pixeldaten geladen werden sollen.
Die *InputFSM* lädt während der Verarbeitung einer Pixelzeile, die Nächste in das *LIM*. Bevor jedoch eine weitere Pixelzeile in das *LIM* geladen werden kann, muß die *InputFSM* warten, bis die Prozessierung der aktuellen Zeile abgeschlossen ist. Hierfür wird ein Handshaking Protokoll zwischen *Matrix* und *InputFSM* verwendet.
Wenn das Ende der Zeile erreicht ist (definiert durch inpos_right) wird die *Matrix* an den Anfang der nächsten Zeile verschoben (definiert durch inpos_left). Wenn die *Matrix* an den Bildanfang verschoben wird, werden bei einer 5×5 Nachbarschaft 3 neue Spalten geladen. Die beiden Spalten außerhalb des Bildbereichs werden durch Kopieroperationen (definiert durch das Selectsignal „load“) hinzugefügt. Im Folgenden wird nur ein Konzept zum Befüllen der Matrixelemente außerhalb des Bildbereichs vorgestellt.
Wenn für die Verarbeitung eines Bildes ein Sliding Window mit einer Pixelnachbarschaft verwendet wird (beispielsweise bei Faltungsoperationen) und das Ausgangsbild dieselben Dimensionen wie das Eingangsbild haben soll, muß die Randwertbehandlung betrachtet werden. Das Problem bei der Randwertbehandlung ist in Abbildung 4.18(a) zu sehen. Zu Beginn der Prozessierung eines Eingangsbildes mit einer Matrix in der linken oberen Ecke liegen einige Matrixelemente außerhalb der definierten ROI. Falls die ROI kleiner als die Eingangsbildgröße ist, können die benötigten Pixelwerte aus dem *LIM* geladen

werden. Ist die ROI gleich der Eingangsbildgröße, müssen die Werte der außen liegenden Matrixelemente künstlich erzeugt werden. Dazu stehen drei verschiedene Methoden zur Auswahl. Das Setzen von konstanten Werten, das Kopieren der äußersten Zeilen und Spalten, und das Spiegeln an den äußersten Zeilen und Spalten. Das Setzen von Konstanten ist vor allem bei binären Ausgangsbildern sinnvoll. In der AutoVision Architektur werden allerdings fast ausschließlich die Pixel am Rand nach außen kopiert. In einigen Ausnahmefällen wird das Spiegeln der Pixel am Bildrand verwendet (s. [Asc09]), was wahlweise in der *Matrix* oder der *User logic* implementiert werden kann. Die Matrixelemente, die links oben außerhalb des Bildbereichs liegen, werden durch eine Kopie des zentralen Pixels gefüllt. Die verbleibenden Matrixelemente, die außerhalb der ROI liegen, verwenden eine Kopie der am nächsten liegenden Pixelwerte des Bildrandes.

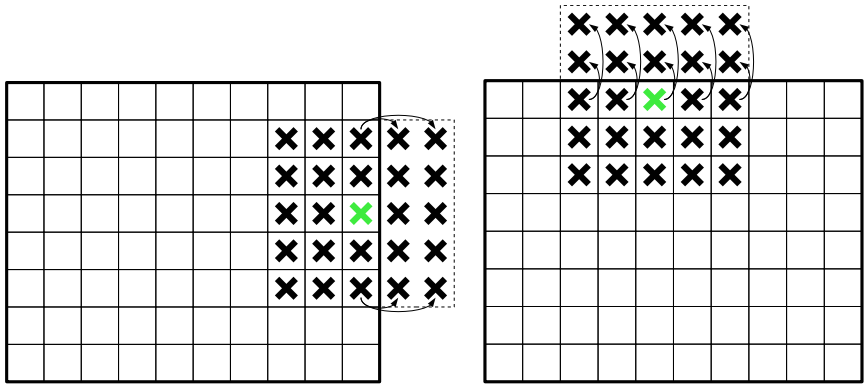


(a) Füllen der Matrixelemente am Anfang des Bildes

(b) Füllen der Matrixelemente am Anfang einer Zeile

Abbildung 4.18: Randwertbehandlung an der linken Seite einer ROI

Wenn die *Matrix* an den Anfang einer Bildzeile gesetzt wird, liegen die Matrixelemente links des zentralen Pixels außerhalb der ROI. Dies ist in Abbildung 4.18(a) dargestellt. Auch hier werden die Pixel außerhalb der zu verarbeitenden ROI durch eine Kopie der am nächsten liegenden Pixelwerte am Rand der ROI erzeugt. Wie in Abbildung 4.18(b) zu erkennen ist, existiert ein ähnliches Problem am Ende einer Bildzeile. Ein so genannter Readback Pfad (siehe Abbildung 4.17) wird verwendet, um die letzte gültige Spalte der Matrixelemente zurückzulesen. Mit Hilfe dieser Kopiertechnik, kann das zentrale Pixel samt Nachbarschaft bis zum Ende der ROI (inpos_right) geschoben werden. Auch in diesem Fall wird eine Kopie der letzten Spalte am Bildrand verwendet, um die nicht existierenden Matrixelemente außerhalb der ROI zu füllen.
Wenn die erste Bildzeile mit einer *Matrix* verarbeitet werden soll, sind überhalb und unterhalb der ROI keine Bilddaten im *LIM* verfügbar, um die Matrixelemente außerhalb



(a) Matrix am Ende einer Bildzeile **(b)** Matrix in der ersten Bildzeile

Abbildung 4.19: Matrixposition an den Rändern des Bildbereichs

der ROI zu füllen. Das Befüllen der Matrixelemente in der ersten Zeile ist in Abbildung 4.19(b) dargestellt. Um das Problem der Befüllung in vertikaler Richtung zu lösen, wird ein ähnliches Verfahren, wie in horizontaler Richtung verwendet. Die Pixel der zentralen Bildzeile werden verwendet, um die außerhalb der ROI liegenden Matrixelemente zu füllen. Es ist offensichtlich, dass das Selectsignal „sel“ zur Auswahl verschiedener Bildzeilen, abhängig von der y Koordinate der Matrix sein muss.

Bei der Verwendung von sehr großen Matrizen werden bei dieser Realisierung nicht nur sehr viele Register, sondern auch sehr viele Multiplexer benötigt. Vergleiche haben gezeigt [Jia08a], dass eine 15×15 *Matrix* für Pixel mit einer Weite von $P_W = 32$ eine Anzahl von 7312 Flip Flops und 12127 LUTs benötigt hat. Der Ressourcenverbrauch wird also nicht von der Anzahl an Registern dominiert, sondern vielmehr von der Anzahl an Multiplexern. Um den Ressourcenverbrauch zu senken, kann statt dem Kopieren der zentralen Bildzeile das Setzen von Konstanten am Bildrand verwendet werden. Durch diese Maßnahme lassen sich alle Multiplexer in Abbildung 4.17 bis auf zwei an der rechten Seite entfernen. Diese Methode kann jedoch nur verwendet werden, wenn die Randwertbehandlung unkritisch ist und nicht zu Standardunkonformität führt, wie beispielsweise in [Asc09] beschrieben. In Abbildung 4.20 ist die Verwendung dieses Konzepts im Fall einer 5×5 Matrix dargestellt. Jedes Matrixelement soll wiederum Pixel mit $P_W = 32$ speichern. Um eine *Matrix* an der linken Bildgrenze der ROI zu füllen, wird ein 32 Bitvektor mit Nullen von rechts nach links durch die Spalten der Matrixelemente geschoben. In den linken beiden Spalten von Matrixelementen in Abbildung 4.20 stehen dann Nullen, in den andern die Pixeldaten des Eingangsbildes. Bei Verwendung dieses Konzepts zeigten die Syntheseergebnisse einen Verbrauch an Ressourcen von 7306 Flip Flops und 4670 LUTs.

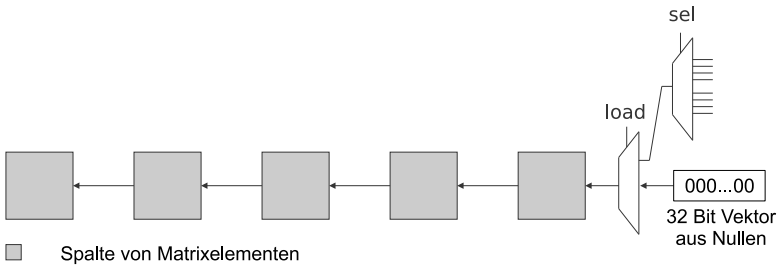


Abbildung 4.20: Modifizierte Struktur der *Matrix* beim Befüllen der außenliegenden Bildbereiche mit Nullen am linken Rand eines Bildes

4.3.6 User Logic

Die *User Logic* verarbeitet das zentrale Pixel und seine Umgebung, welche vom Modul *Matrix* zur Verfügung gestellt wird parallel in verscheidenen PEs. Dies ist in Abbildung 4.16 dargestellt. Die *User Logic* entscheidet, ob der HWA für Rücklichtdetektion, Shapeerkennung oder die Berechnung des Optischen Flusses zuständig ist. Im Vergleich zu anderen Modulen der Pixelprozessierungspipeline, die allesamt für den Pixeltransport bzw. als Pixelpuffer verwendet werden, stellt die *User Logic* ein eigenständiges Modul dar, das die Bildverarbeitungsoperation an sich beinhaltet. Beispiele für verschiedene Arten der *User Logic* und somit auch verschiedene HWAs werden in Abschnitt 4.4 vorgestellt. Erwähnenswert an dieser Stelle ist, dass die *User Logic* nur für die Pixelverarbeitung, nicht aber für die Berechnung der Pixel Koordinaten verantwortlich ist.
Zusammen mit dem Modul *Matrix* realisiert die *User Logic* das Konzept der Parallelverarbeitung (siehe Abschnitt 4.1.3). Da die *Matrix* jeden Taktzyklus ein zentrales Pixel mit Umgebung zur Verfügung stellt, kann die *User Logic* die unabhängigen Pixel massiv parallel verarbeiten.

4.3.7 Intermediate Local Memory (ILM)

Nachdem die Pixeldaten verarbeitet wurden, werden sie im Normalfall zurück in den Hauptspeicher oder zu einem Ausgang (Anzeige) transferiert. Wenn die verarbeiteten Daten jedoch Zwischenresultate darstellen und diese Resultate von einer zusätzlichen *User logic* weiterverarbeitet werden sollen, gibt es zwei Möglichkeiten, wie mit diesen Resultaten verfahren werden kann. Wie bereits erwähnt, ist es möglich die Resultate in den Hauptspeicher zurück zu senden, um sie für eine erneute Operation durch einen anderen HWA von dort zu laden. Eine zweite Möglichkeit besteht im temporären Puffern von Zwischenresultaten. Hierbei werden wenige verarbeitete Bildzeilen der Zwischenresultate in on-chip BRAMs gespeichert. Sobald genügend Daten für die nächste Prozessierungsstufe zur Verfügung stehen, kann die folgende *User Logic* mit der Bearbeitung beginnen. Diese

Methode führt nicht nur zu einer reduzierten Gesamtverarbeitungszeit, sondern reduziert auch deutlich die Buslast dadurch, dass Zwischenresultate weder in den Hauptspeicher geschrieben, noch erneut gelesen werden müssen. Für das Puffern solcher Zwischenresultate kann das Modul *Intermediate Local Memory (ILM)* zur Vertiefung der Pipeline verwendet werden. Das *ILM* dient als temporärer Puffer für Daten, die von einer *User Logic* verarbeitet wurden (im Folgenden als 'upstream' bezeichnet) und als Input einer weiteren *User Logic* dienen (im Folgenden als 'downstream' bezeichnet). Ebenso wie das LIM besteht auch das ILM aus einer konfigurierbaren Anzahl an Bufferlines N_{BL}. Um Daten aus dem *ILM* zu lesen wird eine Spalte adressiert. Am Ausgang erscheint dann eine Anzahl von N_{BL} Pixeln. Zwischen dem *ILM* und der downstream *User Logic* wird erneut das Modul *Matrix* (siehe Abschnitt 4.3.5) benötigt, um einen zentralen Pixel mit seiner Umgebung der downstream *User Logic* zur Verfügung zu stellen.
Ähnlich wie das *LIM* gibt auch das *ILM* ein Signal aus, welches das Laden neuer Bufferlines untersagen kann. Die upstream *User Logic* darf beispielsweise keine neuen Daten schreiben, wenn die downstream *User Logic* mit der Bearbeitung noch benötigter Bildzeilen beschäftigt ist. Daher wird ein Handshaking-Protokoll verwendet, dass die gesamte vorangehende Pipeline stoppen kann. Eine Bufferlines kann geschrieben werden, wenn Gleichung 4.4 erfüllt ist:

$$\mathrm{w\_pos\_y} - \mathrm{r\_pos\_y} < N_{BL} - N_W \tag{4.4}$$

w_pos_y und r_pos_y arbeiten auf den Bildzeilen. w_pos_y repräsentiert in diesem Fall die aktuelle Bildzeile, die durch die upstream Logik geschrieben wurde. Dementsprechend wird mit r_pos_y die Bildzeile adressiert, die gerade von der downstream Logik gelesen wird. Gleichung 4.4 berücksichtigt den Zählerüberlauf bei den Bildzeilen, die überschrieben werden sollen. Zum Beispiel führen Werte von 0, 8, 16, etc. des Zählers w_pos_y immer zur Auswahl der Bufferline 0. Die Differenzanalyse von Lese- und Schreibposition und das ständige Unterschreiten der Grenze $N_{BL} - N_W$ verhindert das ungewollte Überschreiben von Pixeln.
Abbildung 4.21 zeigt diesen Mechanismus für ein *ILM* mit 8 Bufferlines und einer 7×7 *Matrix*. Die gegenwärtige Leseposition r_pos_y und die Nachbarschaftsweite von $N_W = 3$ überspannen die Bildzeilen 2 bis 8, was den Bufferlines 2 bis 7 und 0 entspricht. Bufferline 1, die durch w_pos_y angezeigt wird, ist frei und kann beschrieben werden. Das *ILM* realisiert damit das in Abschnitt 4.1.4 beschriebene Konzept. Weiterführende Details zum *LIM* finden sich in [Hui09] beschrieben.

4.3.8 Local Output Memory (LOM)

Um ausgangsseitig das Blockieren der Pipeline durch Ergebnispixel zu verhindern ist ein *Local Output Memory (LOM)* zwischen *OutputFSM* und der letzten *User Logic* notwendig. Das *LOM* ist für das Sammeln der Ergebnispixel zuständig, die dann über einen Burst Transfer in den Hauptspeicher übertragen werden.

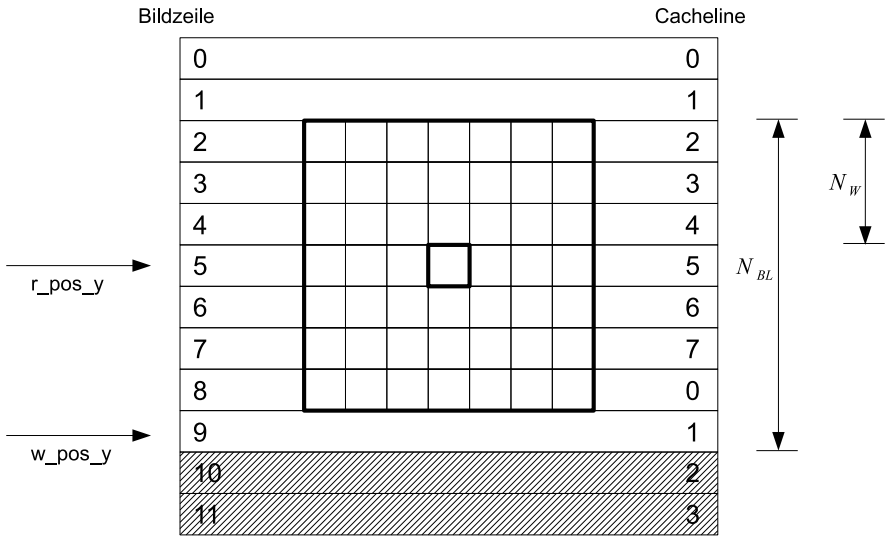


Abbildung 4.21: Verhindern des Überschreibens benötigter Daten durch den upstream

Die Berechnung der Pixelkoordinaten wird vom *LOM* durchgeführt. Diese Koordinaten werden wiederum als Speicheradresse für die Pixel in einer Bufferline verwendet. Die Koordinaten können zusätzlich für die Kommunikation mit der *Output FSM* verwendet werden. Dadurch kann das Überschreiben von Daten im *LOM*, die noch nicht in den Hauptspeicher geschrieben wurden verhindert werden. Analog zum *LIM* verwendet auch das *LOM* BRAMs als lokalen Speicher.

Im Gegensatz zum *LIM* besteht das *LOM* allerdings nur aus einer Bufferline. Diese Bufferline ist aus 2 parallel geschalteten BRAMs aufgebaut. Dies ist notwendig, da während eines Burst-Transfers am Ausgang 64 Bit pro Taktzyklus zum Transfer über den PLB verfügbar sein müssen. Im Gegensatz dazu, muß eingangsseitig ein Pixel mit der Weite P_W in das *LOM* geschrieben werden können. Aus diesem Grund sind mehrere Taktzyklen notwendig, um das *LOM* zu füllen, bevor ein Burst-Transfer gestartet werden kann. Im Normalfall funktioniert das Füllen des *LOM* wesentlich langsamer als das Senden der Pixeldaten in den Hauptspeicher. Daher kommt es innerhalb des *LOM* auch nicht zu Überläufen. Bei der Verwendung von sehr großen Pixelweiten von $P_W = 64$ oder mehr wird einem Überlauf durch ein entsprechendes Signal vorgebeugt, das die Pipeline stoppen kann. Der Aufbau des *LOM* ist dem des *LIM* sehr ähnlich und wird deshalb nicht weiter beschrieben. Eine detaillierte Beschreibung der Implementierung des *LOM* ist jedoch in [Jia08a] zu finden.

4.3.9 Output FSM

Das Modul *OutputFSM* hat verglichen mit der *InputFSM* die genau gegensätzliche Funktion. Die *OutputFSM* ist für das Schreiben der Pixel an die korrekte Adresse in den Hauptspeicher verantwortlich. Dies beinhaltet die Bestimmung des korrekten Transfertyps und der damit verbundenen Burstweite, die Berechnung der Speicheradresse und das Garantieren, dass genügend Daten für den jeweiligen Transfertyp im *LOM* zur Verfügung stehen.
Im Vergleich zum Lesen der Pixeldaten aus dem Hauptspeicher ist das Schreiben ein destruktiver Prozess. Lesen von Pixeldaten ändert im Gegensatz zum Schreiben nicht den Inhalt des Hauptspeichers. Wird eine ROI von einem HWA verarbeitet, werden die originalen Bilddaten häufig direkt mit den Ergebnispixeln überschrieben. Während beim Lesen allerdings ein Burst-Transfer durchaus an einer Burstgrenze beginnen kann, die außerhalb der ROI liegt, würde dies im Schreibfall zu korrupten Daten der Pixel außerhalb der ROI führen. Je nach Größe und relativer Position der ROI kann ein Schreibtransfer in 3 unterschiedliche Transfertypen unterteilt werden. Ein *single transfer* zur Übertragung einzelner Pixel, ein *short transfer* für die Übertragung in kurzen Bursts oder ein *long transfer* für die Übertragung in Bursts maximaler Länge. Dies ist in Abbildung 4.22 dargestellt.

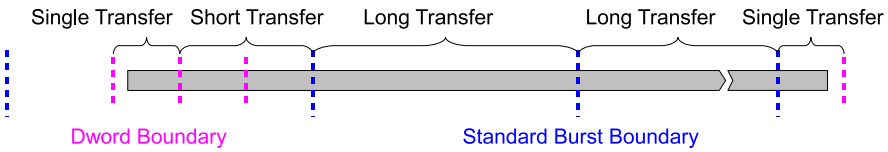
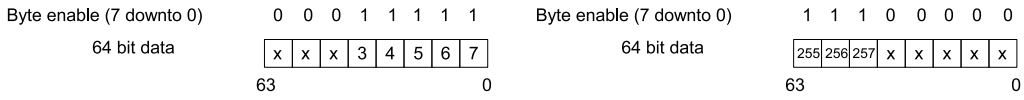


Abbildung 4.22: Verschiedene Arten von Schreibtransfers

Da die Datenweite D_W des PLBs 64 Bit ist, sind Transfers mit weniger als 64 Bit nicht möglich. Aufgrund von P_W können jedoch mehrere Pixel in einem einzelnen 64 Bit Einzeltransfer enthalten sein. Einige der Pixel innerhalb dieses Einzeltransfers könnten jedoch Pixeldaten überschreiben, die außerhalb der ROI liegen. Dies würde zum ungewollten Überschreiben der Pixel des Originalbildes führen. Um dieses Problem zu lösen werden so genannte *Byte Enables* verwendet. Die *Byte Enables* kennzeichnen die gültigen Bytes innerhalb eines Transfers. Um *Byte Enables* sinnvoll verwenden zu können, muß unterschieden werden, ob ein Einzeltransfer an der linken oder rechten Grenze der ROI stattfinden soll. Ein Einzeltransfer an der linken Seite einer ROI ist in Abbildung 4.23(a) dargestellt. Die drei MSBs des *Byte Enable* Vektors sind in diesem Fall 0. Die gültigen Bytes sind hingegen mit einer 1 gekennzeichnet. Im Gegenatz dazu sind die LSBs des *Byte Enable* Vektors 0, wenn ein Einzeltransfer an der rechten Seite der ROI stattfindet. Dies ist in Abbildung 4.23(b) dargestellt. Ein *long transfer* besteht immer aus der maximal verfügbaren B_W, welche 16 beim PLB und 32 beim NPI entspricht. Wenn ein Burst-

Transfer beispielsweise auf dem PLB kürzer als 16 DWords ist, sollte ein *short transfer* verwendet werden. Im Falle eines *short transfers* oder *long transfers* werden die *Byte Enables* nicht verwendet, bzw. alle Bytes werden als gültig markiert. Die *OutputFSM* ist somit für die Berechnung der korrekten Burstweite B_W zuständig.



(a) Single Transfer zu Beginn einer Bildzeile **(b)** Single Transfer am Ende einer Bildzeile

Abbildung 4.23: Beispiel für Single Transfers zu Beginn und am Ende einer Bildzeile

Ein Beispiel, in dem alle drei Transfertypen verwendet werden ist in Abbildung 4.24 dargestellt. In diesem Beispiel wird eine ROI prozessiert. Jedes Pixel hat eine Weite von $P_W = 8$ Bit. inpos_left wurde auf 2 und inpos_right auf 258 gesetzt. Entsprechend den zuvor definierten Transfertypen besteht der Schreibprozess zeitlich gesehen aus einem *single transfer*, einem *short transfer*, einem *long transfer* und abermals einem *single transfer*.

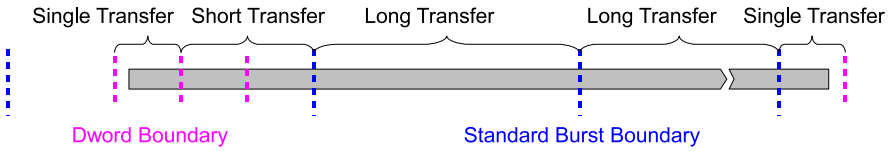


Abbildung 4.24: Beispiel verschiedener Transfertypen beim Schreiben einer Bildzeile

Wie die *Input FSM* besteht die *Output FSM* ebenfalls aus zwei verschiedenen FSMs mit ähnlicher Funktionalität. Da sich die FSMs in *Input FSM* und *Output FSM* nur unwesentlich unterscheiden, wird auf diese hier nicht weiter eingegangen. Zusammen mit der *Input FSM* realisiert die *Output FSM* das in Abschnitt 4.1.2 beschriebene Konzept des Bursting.

4.3.10 Feature Output Memory (FOM)

Das Ergebnis einer oder mehrerer Instanzen einer *User Logic* muß nicht notwendigerweise wieder ein Bild sein. Einige Bildverarbeitungsalgorithmen werden für die Extraktion von wichtigen Merkmalen oder Regionen verwendet, so genannten Features. Unter einem Feature versteht man bestimmte Merkmale in einem Bild. Dies können beispielsweise markante Punkte (Ecken), Objektlinien und Kurven (Kanten) aber auch Farben sein. Die Extraktion von Features führt verglichen mit den Eingangsbilddaten in Normalfall

zu einer signifikanten Datenreduktion. Extrahierte Features sind oft in einer zusammenhängenden Liste, der so genannten Featureliste, im Hauptspeicher abgelegt. Dabei kann die Featureliste weit mehr Informationen als nur Koordinaten und Größe von Features im Eingangsbild enthalten. Ein einzelnes Feature kann aus vielen Einzelinformationen bestehen und kann durch eine Struktur (Struct) oder Klasse in C++ repräsentiert werden. Das *Feature Output Memory (FOM)* nimmt die extrahierten Featuredaten der *User Logic* entgegen und speichert sie bis zur Übertragung eines Bursts. Im Gegensatz zum *LOM*, das einen Ergebnispixel für jedes Eingangspixel speichert, werden Daten nur in das *FOM* geschrieben, wenn ein Feature detektiert wurde.

Die Features haben im Normalfall eine Weite F_W, die sich von der Pixelweite P_W unterscheidet, da ein Feature neben den Koordinaten im Bild die mittlere Helligkeit, die Größe etc. enthalten kann. Ähnlich wie bei der Verwendung des *LOM* werden auch 64 Bit (PLB Breite) über den Bus in den Hauptspeicher geschrieben. Stellt man sich eine Featureweite F_W von 32 Bit vor, muß das *FOM* in der Lage sein zwei Feature pro Taktzyklus am Ausgang bereitzustellen. Ein *long transfer* mit $B_W = 16$ besteht somit aus $16 \cdot 64$ Bit = 128 byte = 32 Feature. Da dies auch die Größe einer Bufferline im *FOM* ist, wird pro *long transfer* eine Bufferline transferiert. Im Gegensatz zum *LOM* das BRAMs als Speicher verwendet, nutzt das *FOM* distributed RAM für eine Bufferline.

Um die Pipeline nicht zu blockieren, verwendet das *FOM* zwei Bufferlines, um double buffering zu realisieren. Sobald eine Bufferline gefüllt ist, wird ein Burst-Transfer initiiert. Währenddessen wird die zweite Bufferline für das Speichern weiterer aus der *User Logic* kommender Feature verwendet. Unter Annahme einer beliebigen *User Logic*, die jeden Takt ein 32-Bit Feature ausgibt, werden für das Füllen einer Bufferline mindestens 32 Takte benötigt. Wie man sich jedoch vorstellen kann, resultiert in der Realität nicht jedes Pixel in einem Feature. Ohne die Beachtung von Arbitrationszyklen etc. dauert ein Transfer einer Bufferline in einem Burst der Weite $B_W = 16$ nur 16 Taktzyklen. Daher wird eine der beiden Bufferlines in jedem Fall neue Features von der *User Logic* akzeptieren können. Das Handshakingsignal, das angibt, dass keine weiteren Daten mehr ins *FOM* geschrieben werden können, wird aus Sicherheitsgründen dennoch verwendet, wenn beispielsweise der PLB durch andere Transfers blockiert ist.

Wenn das *FOM* am Ende einer verarbeiteten ROI nicht leer ist, wird ein Signal benötigt, das der *Feature Output FSM* anzeigt, dass ein letzter Burst-Transfer durchgeführt werden muß, um die verbleibenden Feature in den Hauptspeicher zu schreiben. Wenn das letzte Pixel von der *User Logic* prozessiert wurde, wird von der *User Logic* ein so genanntes „NULL-feature“ generiert. Dieses „NULL-feature“ signalisiert das Ende einer zu verarbeitenden ROI und dass keine weiteren Feature mehr ins *FOM* geschrieben werden. Sobald das „NULL-feature“ in das *FOM* geschrieben wurde, wird ein letzter Burst an Daten initiiert, um die restlichen Feature in den Hauptspeicher zu schreiben. Des weiteren wird das „NULL-feature“ für die Bestimmung der Anzahl an Featuren in der Featureliste durch den PPC verwendet.

4.3.11 Feature Output FSM

Das *FOM* wird durch die *Feature Output FSM* ausgelesen. Die gelesenen Feature werden dann über ein IPIF in den Hauptspeicher übertragen. Um die Last auf dem PLB so gering wie möglich zu halten wird nur der Transfertyp *long transfer* (s. Abschnitt 4.3.9) verwendet. Dies ist ohne Weiteres möglich, da die Feature Liste nicht in das Eingangsbild zurückgeschrieben wird und daher keine Gefahr besteht, ungewollt Daten zu überschreiben. Im Vergleich zur *OutputFSM* ist die *Feature Output FSM* daher einfacher aufgebaut. Wie in der *InputFSM* und der *OutputFSM* gibt es in der *Feature Output FSM* auch zwei FSMs, die ähnlich zu denen in Abschnitt 4.3.3 sind. Die *Feature Output FSM* zusammen mit dem *FOM* sind für die Umsetzung des in Abschnitt 4.1.5 beschriebenen Konzeptes verantwortlich. Weitere Details zur *Feature Output FSM* und zum *FOM* sind in [Hui09] zu finden.

4.3.12 Read Arbiter

Wenn mehr als ein Eingangsdatenpfad verwendet werden soll, um z.B. Pixel aus zwei verschiedenen Eingangsbildern zu laden und in zwei unterschiedlichen LIM abzulegen, wird ein *Read Arbiter* verwendet. Der Vergleich von Daten aus mehreren verschiedenen Eingangsbildern ist beispielsweise bei der Berechnung des Optischen Flusses [Ste04] oder bei der Disparitätsschätzung in Stereokamera Systemen notwendig. Der *Read Arbiter* regelt die Anfragen mehrerer Instanzen der *InputFSM*. Die ankommenden Daten werden der Pipeline zugeführt, deren Anfrage akzeptiert wurde. Details zum Read Arbiter sind in [Jia08b] zu finden.

4.4 Bildverarbeitungsalgorithmen der AutoVision Architektur

In diesem Abschnitt werden verschiedene Bildverarbeitungsalgorithmen vorgestellt, die aufgrund unterschiedlicher Fahrsituationen bei videobasierter Fahrerassistenz zum Einsatz kommen könnten. Es werden jedoch nur die in HW ausgelagerten Teile der Algorithmen vorgestellt, da es sich bei diesen, um den echtzeit-kritischen Teil handelt.

4.4.1 Eckendetektion mit der *ShapeEngine*

Eine der fundamentalen Aufgaben in der Bildverarbeitung ist die Reduktion einer sehr großen Eingangsdatenmenge (Pixel) auf wenige interessante Merkmale (Features). Eckendetektoren repräsentieren eine Möglichkeit, um sinnvolle Merkmale zu extrahieren, die anschließend für die Objektdetektion, das Finden von gemeinsamen Punkten in Stereobildern oder Bewegungstracking eingesetzt werden können. Ein Vergleich dreier bekannter Eckendetektoren und ihre Eignung für eine HW Implementierung finden sich in [WD04].

Aufgrund seiner geringen Komplexität und der damit zu erwartenden ressourcenschonenden Implementierung wurde der SUSAN Algorithmus zur Eckendetektion für eine HW Implementierung gewählt.
SUSAN ist eine Abkürzung für **S**mallest **U**nivalue **S**egment **A**ssimilating **N**ucleus. Der SUSAN Algorithms wurde von S.M. Smith und J.M. Brady vorgestellt und ist in [SB97] beschrieben. Die Eckendetektion mit dem SUSAN Algorithmus kann in zwei Abschnitte unterteilt werden. Im ersten Abschnitt wird für jedes Pixel im Eingangsbild der *univalue segment assimilating nucleus (USAN)* Wert bestimmt. Dieses Zwischenresultat wird als *USAN Bild* bezeichnet. Im zweiten Abschnitt wird das *USAN Bild* nach lokalen Minima untersucht, die Ecken im Eingangsbild repräsentieren.
Der *USAN Wert* wird berechnet, indem zunächst eine zirkuläre[4] Maske um ein bestimmtes Pixel $\mathbf{r_0}$ (der so genannte Nucleus) gelegt wird. Anschließend wird der Grauwert des Nucleus mit allen Pixeln $\mathbf{r}$ innerhalb des Radius der Maske verglichen. Wird ein Pixel innerhalb der Nachbarschaft bei diesem Vergleich (aufgrund eines vorher definierten Schwellwertes) als ähnlich klassifiziert, wird der *USAN Wert* um 1 erhöht. Der *USAN Wert* für den Nucleus $\mathbf{r_0}$ gibt also die Anzahl der ähnlichen Pixel innerhalb der zirkulären Nachbarschaft an.

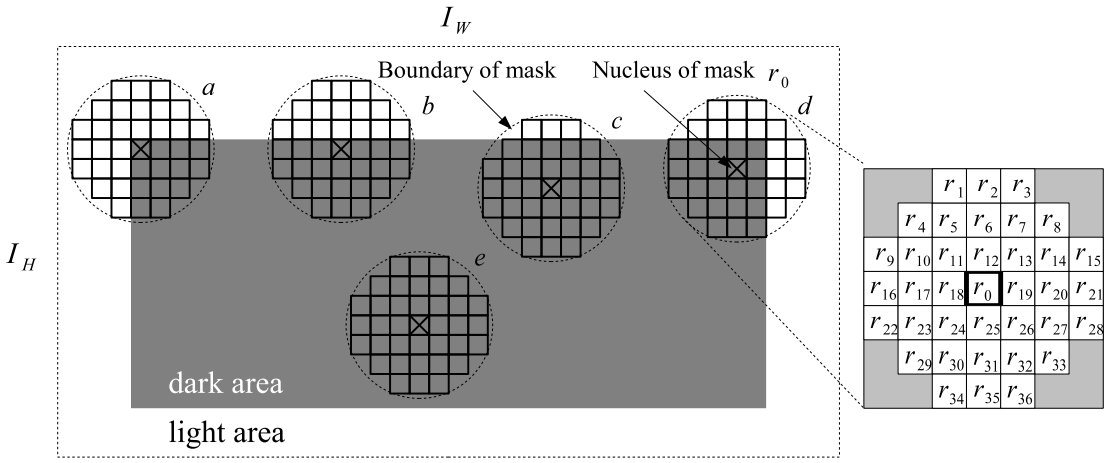


Abbildung 4.25: Berechnung des USAN Wertes: 5 zirkuläre Masken platziert an verschiedenen Stellen eine Bildes liefern unterschiedliche USAN Werte [SB97]

Abbildung 4.25 zeigt ein Beispielbild in dem die Maske an 5 verschiedenen Stellen platziert wurde. Da der Nucleus im Beispiel immer auf einer dunklen Fläche platziert ist, repräsentieren dunkle Pixel innerhalb der Maske die zum Nucleus ähnlichen Pixel. In Abbildung 4.25 ist zu erkennen, dass die Anzahl an ähnlichen Pixel in der Maske (also

[4] zirkulär: kreisförmig

der *USAN Wert*) maximal ist, wenn sich die Maske innerhalb eines homogenen Bereiches befindet (e). Der USAN Wert nimmt ab, wenn die Maske eine Kante erreicht (c, b). Noch kleinere *USAN Werte* können erreicht werden, wenn die Maske nahe einer Ecke platziert wird (d). In diesem Beispiel erreicht der *USAN Wert* ein lokales Minimum, wenn die Maske exakt auf einer Ecke platziert wird (a). Wenn der *USAN Wert* also in etwa der Hälfte des theoretisch möglichen *USAN Wertes* von n_{max}=36 innerhalb der Maske entspricht, kann auf eine Kante im Bild geschlossen werden. Ist der *USAN Wert* kleiner als die Hälfte des theoretisch möglichen *USAN Wertes* innerhalb der Maske ($< \frac{n_{max}}{2}$), kann auf eine Ecke im Bild geschlossen werden. *USAN Werte* größer als als die Hälfte des theoretisch möglichen *USAN Wertes* müssen nicht weiter betrachtet werden und können bereits vor der nächsten Prozessierungsstufe verworfen werden. Um zu entscheiden, ob ein Pixel mit zugehörigem *USAN Wert* als potentieller Kandidat für eine Ecke weiterverarbeitet wird, wird der *geometric threshold* g eingesetzt. g spezifiziert ab welchem *USAN Wert* eine Weiterverarbeitung nicht mehr stattfindet.
Formalisiert bedeutet dies folgendes: Gegeben sei ein Nucleus $\mathbf{r_0}$, ein weiteres Pixel $\mathbf{r}$ innerhalb der Maske und deren Grauwerte $I(\mathbf{r_0})$ und $I(\mathbf{r})$. Aus diesen Parametern wird eine *similarity function* $c(\mathbf{r}, \mathbf{r_0})$ definiert.

$$c(\mathbf{r}, \mathbf{r_0}) = \begin{cases} 1 & \text{if } |I(\mathbf{r}) - I(\mathbf{r_0})| \leq t \\ 0 & \text{if } |I(\mathbf{r}) - I(\mathbf{r_0})| > t \end{cases} \tag{4.5}$$

Im Folgenden wird der Betrag der Grauwertdifferenz zwischen Nucleus und Pixel innerhalb der Maske als Δ_I bezeichnet ($\Delta_I = |I(\mathbf{r}) - I(\mathbf{r_0})|$). Gleichung 4.5 enthält den Parameter t, den so genannten *similarity threshold*. Falls nicht anders spezifiziert, wird fortführend t zu 20 angenommen. Der *similarity threshold* t definiert den betragsmäßig maximalen Helligkeitsunterschied zwischen Nucleus und Pixel innerhalb der Maske. t entscheidet somit, ob ein Pixel ähnlich zum Nucleus ist und damit in die Berechnung des *USAN Wertes* einfließt. Durch die Wahl von t kann der SUSAN Algorithmus an verschiedene Characteristiken des Eingangsbildes, wie etwa Kontrast oder das Rauschen von der Kamera angepasst werden. Je kleiner t gewählt wird, desto unwahrscheinlicher ist es, dass ein Pixel in die Berechnung des *USAN Wertes* einfließt. R_{mask} sei die Menge aller Pixel innerhalb der Maske mit Ausnahme des Nucleus $\mathbf{r_0}$. Der *USAN Wert* kann dann folgendermaßen berechnet werden:

$$n(\mathbf{r_0}) = \sum_{\mathbf{r} \in R_{mask}} c(\mathbf{r}, \mathbf{r_0}) \tag{4.6}$$

Schließlich kann die so genannte Corner Response, die einer Ecke einen bestimmten Wert zuweist, berechnet werden.

$$R(\mathbf{r_0}) = \begin{cases} g - n(\mathbf{r_0}) & \text{if } n(\mathbf{r_0}) < g \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

Gleichung 4.7 bildet geringe *USAN Werte* auf große Response Werte ab und umgekehrt. Homogene Bereiche mit einem hohen *USAN Wert* führen daher zu einer sehr geringen Response und werden nicht weiter betrachtet. An dieser Stelle sei darauf hingewiesen, dass die Einführung der Response Berechnung die Suche nach minimalen USAN Werten in eine Suche nach Maxima umwandelt.
Der bereits erwähnte *geometric threshold g* unterdrückt große *USAN Werte* und schließt diese von einer Weiterverarbeitung aus. Für die Eckendetektion sollte g weniger als $\frac{n_{max}}{2}$ sein. Dieser Wert unterdrückt die Response von Kanten, da diese größere *USAN Werte* als Ecken aufweisen, wie in Abbildung 4.25 ersichtlich ist.
Obwohl die *similarity function* (Gleichung 4.5) sehr einfach zu berechnen ist, wird in der Praxis eine stabilere Version verwendet [SB97]:

$$c(\mathbf{r}, \mathbf{r_0}) = e^{-\left(\frac{\Delta_I}{t}\right)^6} \tag{4.8}$$

Gleichung 4.5 liefert nur eine binäre Entscheidung, d.h. ob ein bestimmtes Pixel abhängig von t ähnlich zum Nucleus ist (=1) oder nicht (=0). Im Gegensatz dazu weist Gleichung 4.8 dem Helligkeitsunterschied einen Wert zwischen 0 und 1 zu. Gleichung 4.8 gibt also an, wie ähnlich ein Pixel innerhalb der Maske zum Nucleus ist. Dies erlaubt kleine Variationen (durch z.B. Rauschen) ohne den Wert von $c(\mathbf{r}, \mathbf{r_0})$ zu sehr zu ändern. Bis zu dieser Stelle sind keine Modifikationen an dem in [SB97] beschriebenen Algorithmus vorgenommen. Um für eine Implementierung in HW jedoch Integer Arithmetik anstatt Floating Point Operationen verwenden zu können, wird Gleichung 4.8 mit dem Faktor 100 multipliziert und auf die nächste Integer Zahl gerundet. Dies ist in Gleichung 4.9 dargestellt.

$$c(\mathbf{r}, \mathbf{r_0}) = \text{round}\left(100 \cdot e^{-\left(\frac{\Delta_I}{t}\right)^6}\right) \tag{4.9}$$

Da sich t während der Verarbeitung eines Bildes nicht ändert und da nur eine begrenzte Anzahl an Helligkeitsunterschieden Δ_I zu berechnen sind (256 bei 8 Wert Grauwertpixeln), ist die Berechnung einer LUT sinnvoll. Die Adressen 0 bis 255 der LUT (insgesamt also 256 Einträge) repräsentieren alle möglichen Helligkeitsunterschiede Δ_I bei Grauwertbildern. Unter diesen Adressen sind die entsprechenden Ergebnisse aus Gleichung 4.9 hinterlegt mit jeweils einem Wert zwischen 0 und 100. Die LUT kann vor jeder Verarbeitung eines Bildes neu erstellt werden. Durch Verwendung einer LUT wird die Berechnung von Gleichung 4.9 nur 256 mal durchgeführt anstatt mehrmals für jedes Pixel im Bild.
Für eine HW Implementierung bieten sich entweder Gleichung 4.5 oder Gleichung 4.9 an. Die Implementierung von Gleichung 4.5 benötigt nur einige Komparatoren. Jedoch können kleine Änderungen der Helligkeitswerte einzelner Pixel (verursacht durch z.B. Rauschen) einen großen Einfluß auf den $c(\mathbf{r}, \mathbf{r_0})$ ausüben und damit die Qualität des Ergebnisses beeinflussen. Im Gegensatz dazu ist die Berechnung von Gleichung 4.9 in HW äußerst komplex. Vor allem die Exponentialfunktion und die Division sind schwer

zu realisieren und müssen entsprechend genaue Werte liefern. Wie bereits erwähnt, kann eine LUT verwendet werden, um die Werte von $c(\mathbf{r}, \mathbf{r_0})$ bereits im Vorfeld der eigentlichen Bildverarbeitung zu berechnen. Die in dieser LUT gespeicherten Werte sind stark abhängig vom *similarity threshold* t. t führt nicht nur zu einer Verschiebung entlang der x-Achse sondern verändert auch den Verlauf der Kurve, wie in Abbildung 4.26 zu sehen ist.

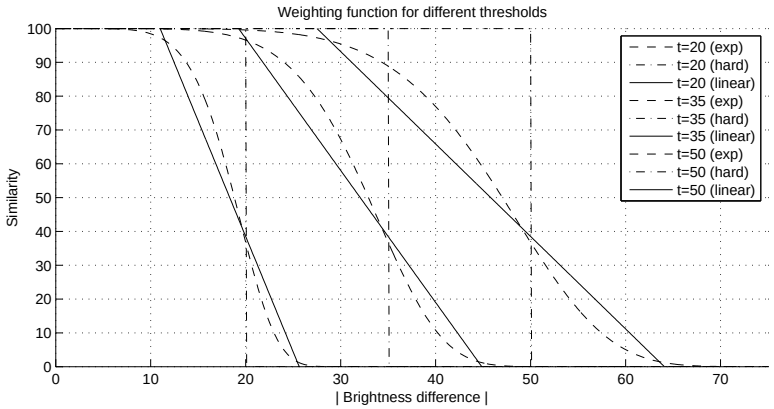


Abbildung 4.26: Die beiden in [SB97] vorgestellten *similarity functions* und deren lineare Approximation nach Gleichung 4.13 abhängig von t

Da t zur Laufzeit adaptierbar sein soll, müssen die Werte der LUT erneut berechnet werden, sobald sich t ändert. Aus Komplexitätsgründen wird daher eine lineare Approximation von Gleichung 4.9 verwendet, die sich mit Addierern und Multiplizierern realisieren lässt. Die zu approximierende Funktion $c(\mathbf{r}, \mathbf{r_0})$ aus Gleichung 4.9 kann abschnittsweise definiert werden. 100, $c_{lin}(\Delta_I)$ und 0. Da $c(\Delta_I)$ symmetrisch zur y-Achse gilt dies auch für die approximierte Funktion. Die abschnittsweise definierte Definition zur Approximation im Wertebereich für $c(\mathbf{r}, \mathbf{r_0})$ von 0 bis 255 ist in Gleichung 4.10 dargestellt.

$$\tilde{c}(\mathbf{r}, \mathbf{r_0}) = \begin{cases} 100 & c_{lin}(\mathbf{r}, \mathbf{r_0}) > 100 \\ 0 & c_{lin}(\mathbf{r}, \mathbf{r_0}) < 0 \\ c_{lin}(\mathbf{r}, \mathbf{r_0}) & \text{otherwise} \end{cases} \tag{4.10}$$

$c(\mathbf{r}, \mathbf{r_0})$ wird also bis zu einem gewissen Wert $\Delta_{I,a}$ den Wert 100, und ab einem Wert $\Delta_{I,b}$ den Wert 0 besitzen. Zwischen $\Delta_{I,a}$ und $\Delta_{I,b}$ wird der Verlauf von Gleichung 4.9 durch eine Gerade mit Gleichung approximiert.

$$c_{lin}(\mathbf{r}, \mathbf{r_0}) = a(t) \cdot \Delta_I + b(t) = -\frac{137,49}{t} \cdot \Delta_I + 175 \tag{4.11}$$

Wie in [Hui09] gezeigt werden konnte, minimiert eine Wahl von a =137,49 und b = 175

den quadratischen Fehler zwischen den approximierten Werten $c_{lin}(\mathbf{r}, \mathbf{r_0})$ und den exakten Werten $c(\mathbf{r}, \mathbf{r_0})$. Der Kurvenverlauf für die beiden in [SB97] vorgestellten *similarity functions* und für die approximierte Version ist in Abbildung 4.26 für die Schwellwerte $t = 20$, $t = 35$ und $t = 50$ dargestellt.
Um eine Division in HW zu vermeiden, wird Gleichung 4.13 (eine Umformung von Gleichung 4.11) verwendet.

$$\begin{aligned} c_{lin}(\mathbf{r}, \mathbf{r_0}) &= \frac{256}{256} \cdot \left(-\frac{137.49}{t} \cdot \Delta_I + 175 \right) \\ &= \frac{1}{256} \cdot (-MST \cdot \Delta_I + 175 \cdot 256) \end{aligned} \tag{4.12}$$

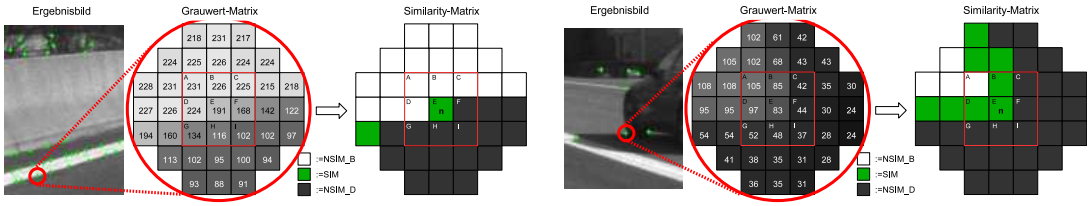
Die Division durch 256 in HW kann mit Hilfe von Shiftoperationen um 8 Bit nach rechts implementiert werden. Damit sind keine Divisionen in HW notwendig und für die Berechnung von 4.13 wird nur ein Multiplizierer verwendet. Der Wert des *modified_similarity_threshold* (MST) hingegen wird in SW (beispielsweise auf einer eingebetteten CPU) berechnet. Die Berechnung von MST erfolgt dabei nur einmal für jedes Bild mit Hilfe von Gleichung 4.13.

$$MST = \text{round} \left(\frac{137.49 \cdot 256}{t} \right) \tag{4.13}$$

Der berechnete Wert für MST, der vom *similarity threshold* t abhängt, wird per Device Control Register (DCR) Bus zum HWA übertragen. Das es sich beim MST um einen zur Laufzeit adaptiven Parameter handelt, wird für diesen Parameter ein Konfigurationsregister (s. Abschnitt 4.3.2) verwendet.
Wie in [WD04] beschrieben, lässt sich der SUSAN Algorithmus durch die zuvor genannten Umformungen einfach in HW implementieren. Trotz der perfekten Lokalisation von Ecken in synthetischen Bildern, entschieden sich Wang et. al. in [WD04] für eine Implementierung des Harris/Plessy Algorithmus [HS88] in HW, aufgrund der qualitativ besseren Ergebnisse. Im allgemeinen schneidet der SUSAN Algorithmus bei nicht synthetischen Bildern im Bezug auf die Genauigkeit schlechter ab als der Harris/Plessy Algorithmus. Wie in Abbildung 4.27 zu sehen ist, werden an vielen Stellen fälschlicherweise Ecken erkannt. Dieses Verhalten des SUSAN Algorithmus liegt darin begründet, das zwar nach ähnlichen Pixeln innerhalb der Maske gesucht wird, deren räumliche Verteilung jedoch außer acht gelassen wird. Im Folgenden werden darum 3 Postulate definiert, um die der originale SUSAN Algorithmus erweitert wird. Die Verwendung der drei Postulate soll die Anzahl an falsch detektierten Ecken (False Positives) reduzieren bei gleichzeitiger Beibehaltung der Rotationsinvarianz[5] des SUSAN Algorithmus. Um auf eine korrekt detektierte Ecke schließen zu können, müssen alle drei Postulate erfüllt sein. Falls nicht

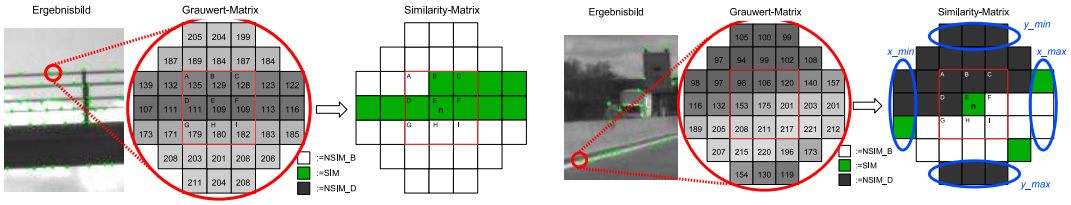
[5] Das Ergebnis des *USAN Wertes* ist unabhängig von einer Drehung der Maske.

anders definiert, wird im Folgenden für t der Wert 20 und für g der Wert 1750 angenommen.



(a) Falsch detektierte Ecke an einer Fahrbahnmarkierung (t=20)

(b) Richtig detektierte Ecke an einem Reifen (t=20)



(c) Falsch detektierte Ecke an einem Geländer (t=20)

(d) Falsch detektierte Ecke an einer Fahrbahnmarkierung (t=20)

Abbildung 4.27: In allen Abbildungen ist ein Ausschnitt des Ergebnisbildes (links), die kreisrunde Grauwert-Maske um die detektierte Ecke (Mitte) und die Maske in der die zum Nucleus ähnlichen Pixel markiert sind (rechts) zu sehen

1. Postulat: Dominante Pixelgruppen

Die Pixel innerhalb der zirkulären Maske können in drei Mengen unterteilt werden. Die Menge an Pixeln, die einen ähnlichen Grauwert wie der Nucleus besitzen SIM, die Menge an unähnlichen Pixeln, die wesentlich dunkler sind, als der Nucleus $NSIM\_D$ und die Menge an unähnlichen Pixeln, die wesentlich heller sind, als der Nucleus $NSIM\_B$. Gleichung 4.14 wird verwendet, um zu bestimmen, ob ein Pixel $\mathbf{r}$ Element der Menge SIM, $NSIM\_D$ oder $NSIM\_B$ ist.

$$\mathbf{r}\,\epsilon \begin{cases} SIM & |I(\mathbf{r}) - I(\mathbf{r_0})| \leq t \\ NSIM\_B & I(\mathbf{r}) - I(\mathbf{r_0}) > t \\ NSIM\_D & I(\mathbf{r}) - I(\mathbf{r_0}) < -t \end{cases} \tag{4.14}$$

Die Unterteilung der Pixel in diese Mengen ist notwendig um das erste Postulat zu formulieren:

1. Postulat: Eine Ecke setzt voraus, dass es immer eine dominante Gruppe innerhalb der zirkulären Maske gibt, die sich vom Nucleus unterscheidet.

Dies bedeutet, dass entweder $NSIM\_D$ oder $NSIM\_B$ eine dominante Gruppe darstellen. Die Kardinalität[6] der Menge $NSIM\_D$ oder der Menge $NSIM\_B$ muß dabei größer als 19 sein, was etwas mehr als die Hälfte der Gesamtzahl der Pixel innerhalb der zirkulären Maske ist. Mit Hilfe dieses Postulats können besonders effektiv False Positives[7] an hell-dunkel bzw. dunkel-hell Übergängen entfernt werden.
Ein Beispiel ist in Abbildung 4.27(a) dargestellt. Der Bildauschnitt links zeigt einen Teil des Ergebnisbildes in dem Ecken detektiert wurden. Wie in diesem Bildausschnitt zu sehen ist, werden besonders an der Fahrbahnmarkierung Ecken falsch detektiert. Der Grund hierfür läßt sich durch die nähere Analyse der entsprechnden Grauwert-Maske (Mitte) einer dieser falsch detektierten Ecken (markiert mit rotem Kreis) erkennen. Die Verwendung eines *similarity thresholds* von $t = 20$ führt auf die in Abbildung 4.27(a) dargestellte Similarity-Matrix. In der Similarity-Matrix sind die Grauwert Pixel den drei Mengen SIM, $NSIM_B$ und $NSIM_D$ zugewiesen. Zu erkennen ist, dass nur ein Pixel (abgesehen vom Nucleus selbst) ähnlich zum Nucleus ist (grün markiert). Da dieses eine Pixel bestimmt geringer als $g = 1750$ ist, wird an solchen Stellen fälschlicherweise eine Ecke detektiert. Da weder die Kardinalität der Menge $NSIM\_D$ noch die der Menge $NSIM\_B$ größer als 19 ist, werden diese falsch detektierten Ecken durch Verwendung des ersten Postulats entfernt. Im Gegensatz dazu hat die Menge $NSIM\_D$ in Abbildung 4.27(b) eine Kardinalität von 25, die Mengen SIM und $NSIM_B$ hingegen nur 4 und 7. Anhand von Postulat 1 wird damit auf eine richtig detektierte Ecke geschlossen.

2. Postulat: Center-Matrix
In natürlichen Bildern existieren viele Fälle in denen Ecken falsch detektiert werden, obwohl das 1. Postulat verwendet wird. Einer dieser Fälle ist in Abbildung 4.27(c) dargestellt. In diesem Fall ist der *USAN Wert* abermals kleiner als g. Der SUSAN Algorithmus detektiert deshalb erneut eine falsche Ecke. Bei der Suche nach dominanten Gruppen (1. Postulat) ist erkennbar, dass $NSIM\_B$ eine Kardinalität von 26 aufweist. Das bedeutet, dass auch das 1. Postulat die falsch detektierte Ecke nicht entfernt. Daher ist ein weiteres Postulat notwendig, dass die Fälle behandelt, die nicht vom 1. Postulat abgedeckt werden.

2. Postulat: Eine Ecke setzt voraus, dass gegenüber liegende Pixel innerhalb der Center-Matrix nicht zur selben Menge der ähnlichen Pixel SIM gehören.

Die Elemente der 3×3 Center-Matritzen, die in Abbildung 4.27 dargestellt sind, werden

[6] Kardinalität: Anzahl der Elemente einer Menge.
[7] False Positives sind fälschlicherweise erkannte Features, die in diesem Fall das Kriterium einer Ecke erfüllen

mit den Buchstaben A bis I markiert. Wenn A und I oder B und H oder C und G oder D und F Elemente der Menge SIM sind, führt das 2. Postulat dazu, dass die zugehörige falsch detektierte Ecke entfernt wird. Im in Abbildung 4.27(c) dargestellten Fall sind D und F Elemente der Menge SIM. Diese Ecke wird daher als falsch erkannt.

3. Postulat: Analyse von Grenzbereichen
Bei der Betrachtung von Abbildung 4.27(d) ist zu erkennen, dass weder das 1. noch das 2. Postulat in der Lage sind, die falsche Ecke, die vom SUSAN Algorithmus detektiert wurde, als solche zu erkennen. Innerhalb der zirkulären Maske existiert $NSIM\_D$ als dominante Gruppe mit einer Kardinalität von 20. Zusätzlich existieren in der Center-Matrix keine gegenüberliegenden Pixel, die der Menge SIM angehören. Das Problem in diesem Fall ist, dass die ähnlichen Pixel räumlich weit über die zirkuläre Maske verteilt sind, obwohl diese Pixel eigentlich zu einer zusammenhängenden Struktur (weiße Fahrbahnmarkierung) gehören. Dieses Problem taucht häufig auch bei verrauschten Bildern auf. Um dieses Problem zu lösen, wird die nicht dominante Gruppe ($NSIM\_B$ in Abbildung 4.27(d)) mit der Menge SIM vereinigt. Die Pixel in dieser Vereinigungsmenge werden im Folgenden als Strukturpixel bezeichnet, wenn, wie in diesem Fall eine dominante Gruppe existiert. Durch Vereinigung der beiden Mengen, kann das 3. Postulat formuliert werden.

3. Postulat: Eine Ecke setzt voraus, dass Pixel aus der Vereinigung der nicht dominanten Gruppe und der Menge an ähnlichen Pixeln SIM nicht in gegenüberliegenden Grenzbereichen der zirkulären Maske liegen.

Die Grenzbereiche x_{min}, x_{max}, y_{min} and y_{max} sind in Abbildung 4.27(d) dargestellt. Das 3. Postulat untersagt bei Ecken die Existenz von Strukturpixeln sowohl in den Bereichen x_{min} und x_{max} als auch in y_{min} und y_{max}. Wie in Abbildung 4.27(d) zu erkennen ist, enthalten die Grenzbereiche y_{min} und y_{max} nur Pixel der dominanten Gruppe. Im Gegensatz enthalten die Grenzbereiche x_{min} als auch x_{max} Strukturpixel, welche diese Ecke als inkorrekt ausweisen. Die Analyse ganzer Grenzbereiche anstatt einzelner Pixel an der Grenze, macht diese Methodik wesentlich robuster gegenüber verschieden Winkelstellungen von z.B. Fahrbahnmarkierungen.

4.4.1.1 Implementierung der *SE* in HW

Die *ShapeEngine (SE)* ist die VHDL Implementierung des SUSAN Algorithmus. Der Name *ShapeEngine* wurde aufgrund der Tatsache gewählt, dass viele Ecken am Profil (Shape) bzw. der Silhouette von Fahrzeugen gefunden werden. Die *SE* besteht aus mehreren gepipelinten User Funktionen und wurde komplett mit der in Abschnitt 4.3 vorgestellten Modulbibliothek realisiert. Durch die gepipelinete Verarbeitung werden Datentransfers zwischen Engine und Hauptspeicher (Video-Frame Buffer) minimiert. Anstatt die Zwi-

schenresultate zurück in den Hauptspeicher zu schreiben, werden diese in *ILMs* (s. Abschnitt 4.3.7) zwischengepuffert. Die *SE* wurde prototypisch auf einem Xilinx XC2VP30 Device implementiert.

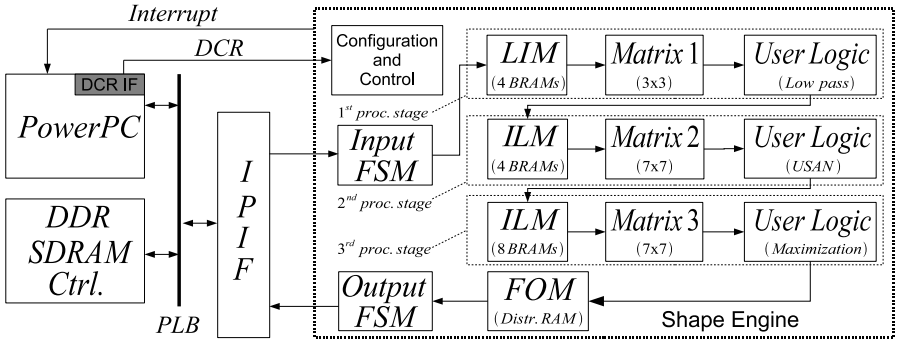


Abbildung 4.28: Block Diagramm des eingebetteten Systems mit *ShapeEngine* aufgebaut aus Elementen der in Abschnitt 4.3 präsentierten Modulbibliothek

In Abbildung 4.28 ist ein Blockdiagramm eines eingebetteten Systems mit den für die Pixelverarbeitung wichtigen Komponenten dargestellt. Der Übersichtlichkeit wegen sind Module, wie beispielsweise der *VIN* und der *VOUT* ausgespart. Wie zu erkennen ist, besteht die *SE* aus drei Teiloperationen, die mit Hilfe von Sliding Window Operationen realisiert werden können. Einem Tiefpaßfilter zur Rauschunterdrückung, der Berechnung der USAN Werte und der Suche nach lokalen Maxima. Alle lokalen Speicher, mit Ausnahme des *FOM* sind mit BRAMs realisiert. Der Verbrauch an BRAMs sowie jeweils die Größe der folgenden Matrix ist in Abbildung 4.28 zu erkennen. Über ein IPIF (in Abbildung 4.28 ist beispielhaft ein LIS PLB IPIF dargestellt) gelangen die Bilddaten in den 1. Verarbeitungsabschnitt. Sobald genügend Pixel für die Verarbeitung im ersten *LIM* zur Verfügung stehen, wird mit der Prozessierung begonnen. Das *LIM* muß dabei mindestens die Größe haben, um die Pixelnachbarschaft, die vom folgenden Verarbeitungsabschnitt benötigt wird, zu speichern. Eine *User Logic* zur Realisierung eines Tiefpassfilters (siehe Abschnitt 2.1) ist optional und wird synthetisiert, wenn das entsprechende Generic gesetzt wird. In diesem Fall besteht der erste Verarbeitungsabschnitt aus einem *LIM* mit $N_{BL} = 4$ Bufferlines, einer 3×3 *Matrix* und der *User Logic*, die eine Tiefpassfilterung der Bildeingangsdaten mit einem 3×3 Filterkern vornimmt. Das ILM (bzw. das LIM, wenn das Generic für den Tiefpassfilter nicht gesetzt ist) bildet den Beginn des nächsten Verarbeitungsabschnitts. Dieser Abschnitt besteht neben dem *ILM* aus einer 7×7 *Matrix* und einer *User Logic* zur Berechnung der Corner Response. Die verarbeiteten Pixel bilden ein Corner Response Bild, das dem nächsten Verarbeitungsabschnitt zur Detektion von lokalen Maxima zugeführt wird. Dieser Abschnitt unterscheidet sich leicht von den anderen, da er nicht für jedes Eingangs- auch ein Ergebnispixel erzeugt. Vielmehr wird

nur ein Ergebnis in das *FOM* zurück geschrieben, wenn eine Ecke tatsächlich detektiert wurde. Neben der Pixelverarbeitungspipeline verfügt die *SE* über *Konfigurationsregister* über die beispielsweise die Größe einer ROI, t oder g geändert werden kann.
Details zur Implementierung der *SE* sind in [Hui09], [CHRS09] und [Kil07] zu finden. Weitere Implementierungsdetails zu den drei Postulaten zur qualitativen Verbesserung der Resultate des SUSAN Algorithmus sowie dem SW Teil sind in [Rau09] aufgeführt.

4.4.2 Rücklichtdetektion mit der *TaillightEngine*

Wie bereits in Abschnitt 4.4.1 erwähnt, ist es sinnvoll die Eingangsdatenmenge (Pixel) auf wenige interessante Merkmale (Features) zu beschränken. Da bei Fahrten in dunklen Tunneln oder bei Nachtfahrten aufgrund der Kontrastverhältnisse eine Eckendetektion nicht mehr zuverlässig funktioniert, müssen in diesen Fällen andere markante Merkmale verwendet werden. Als geeignete Merkmale in dunklen Tunneln oder bei Nachtfahrten bieten sich die Rückleuchten von Fahrzeugen an. Der im Folgenden beschriebene Algorithmus ist speziell für den Einsatz in Tunneln oder während Nachtfahrten mit getrennten Fahrbahnen (Bsp. Autobahnen) gedacht.

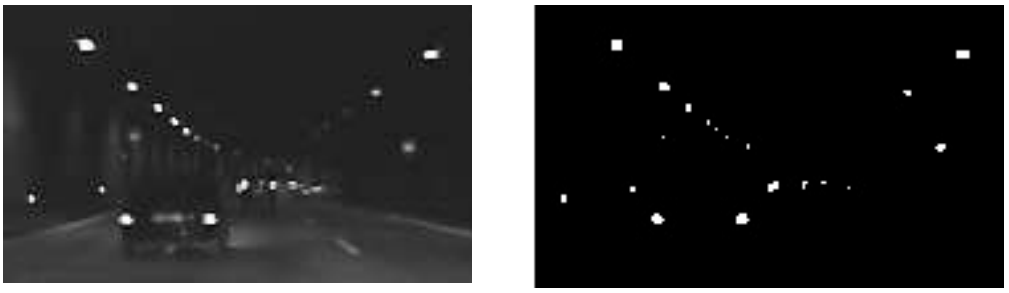
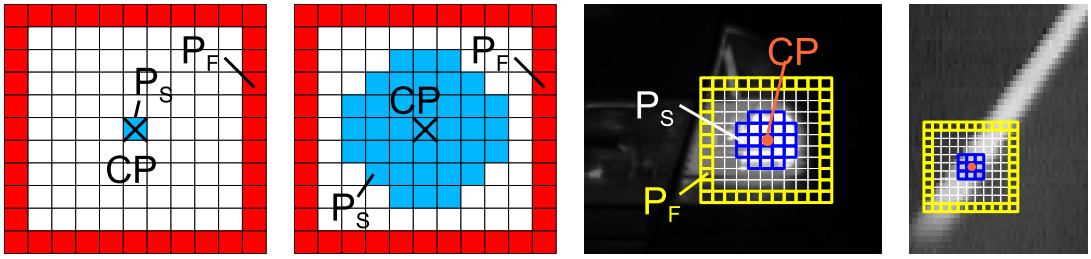


Abbildung 4.29: Eingangsbild (links) und binäres Zwischenresultat (rechts) in dem nur noch annähernd Runde Lichtpunkte (Spotlights) übrig sind

Abbildung 4.29 zeigt eine typische Szene in solchen Umgebungen. Gut zu erkennen, und daher ein markantes Merkmal, sind die aktiven Lichtquellen, wie beispielsweise die Rückleuchten der Fahrzeuge oder die Lichtquellen an der Decke des Tunnels. Viele der gegenwärtigen Methoden zur video-basierten Detektion von Fahrzeugleuchten sind stark abhängig von einem vorher spezifizierten Schwellwert [BHD00, Che09, ABJ+08, CCCW06] und damit auch von den oft stark variierenden Belichtungsverhältnissen in Tunneln. Zur Bestimmung eines geeigneten Schwellwertes wurde beispielsweise die Farbe eines Rücklichts miteinbezogen [ICaB05, OGJ08]. Schwellwertbasierte Verfahren berücksichtigen jedoch nicht die Form eines Objekts. Dies erschwert die Unterscheidung zwischen weiteren hellen Objekten, wie z.B. Fahrbahnmarkierungen, Verkehrsschildern oder der hellen Tunnelausfahrt. Daher wurde in [ACS08] eine HW beschleunigte Version eines Algorithmus zur Rücklichterkennung vorgestellt, die die Form von Objekten miteinbezieht.

Zum besseren Verständnis des später beschriebenen HWAs zur Lichterkennung wird der Algorithmus kurz vorgestellt. Vor der Detektion von Lichtern, den so genannten Spotlights, steht deren Definition. Spotlights sind annähernd runde, besonders helle Stellen im Bild, die von relativ dunklen Pixeln umschlossen werden. Rückleuchten von Fahrzeugen fallen typischerweise in diese Kategorie. Zur Erkennung dieser Rückleuchten wird das Eingangsbild mit einem Strukturfilter durchsucht. Dieser Strukturfilter sucht nach annähernd runden, hellen Objekten und ist damit unabhängig vom umgebenden Licht. Der Strukturfilter wird durch eine Maske repräsentiert, die aus zwei Mengen besteht. P_F repräsentiert die Menge an Pixeln, die den Rahmen (Frame) der Struktur festlegen. Die Menge P_S hingegen definiert die Struktur selbst nach der im Bild gesucht werden soll. Die Pixelmengen P_S und P_F sind in Abbildung 4.30 dargestellt.



(a) **(b)** **(c)** **(d)**

Abbildung 4.30: Suche nach möglichen Lichtpunkten mit zwei unterschiedlichen Masken P_S und P_F in (a) und (b). In (c) und (d) wird die Maske auf verschiedene Pixel des Eingangsbildes angewendet

Wenn alle Pixel der Menge P_S von ihrem Grauwert her heller sind, als die Pixel der Menge P_S, werden die Koordinaten des CP im binären Ausgangsbild markiert. Abhängig davon, wie die Pixel der Menge P_S verteilt sind (s. Abbildung 4.30(a) und Abbildung 4.30(b)) werden mehr oder weniger Spotlights gefunden. Wenn alle Pixel der Menge P_F dunkler sind als alle Pixel der Menge P_S ist das zentrale Pixel CP ein Spotlight. Die Operation funktioniert damit wie eine Erosion [GW08], bei der im Ergebnisbild überall dort ein Pixel markiert wird, wo das Strukturelement gefunden wurde. Diese Bedingung kann folgendermaßen formalisiert werden:

$$\min(lum(P_S)) > \max(lum(P_F)) + threshold \tag{4.15}$$

Gleichung 4.15 repräsentiert damit die Bedingung für ein Spotlightpixel, die durch helle Stellen im Bild charakterisiert sind, die von relativ dunklen Pixeln umschlossen werden. Die Funktionen $\min(lum())$ und $\max(lum())$ liefern dabei den minimalen bzw. maximalen Luminanz- oder Helligkeitswert einer Pixelmenge. Optional kann über den Wert *threshold* zur Laufzeit eingestellt werden, wie sehr sich Pixel der Menge P_S und P_F

unterscheiden. In den meisten Fällen kann auf *threshold* jedoch verzichtet werden. Abbildung 4.30(c) zeigt die auf einem Rücklicht platzierte Maske. Gut zu erkennen ist, dass alle Pixel des Strukturelementes (P_S) heller sind, als die Menge an Pixeln P_F am Rand der Maske, was auf ein Spotlight schließen lässt. Im Gegensatz dazu ist in Abbildung 4.30(d) die Maske an einer Fahrbahnmarkierung platziert. Die hellen Pixel tauchen in diesem Fall sowohl in der Menge P_S, als auch in der Menge P_F auf. Gleichung 4.15 ist damit nicht erfüllt und im Gegensatz zu einem schwellwertbasierten Verfahren, wird bei dieser Art von Fahrbahnmarkierungen kein Pixel im Ausgangsbild markiert. Die Formen von P_S und P_F entscheiden darüber, welche Art von Spotlights gefunden wird. Ein Rücklicht muß mindestens die Pixel der Menge P_S abdecken, um überhaupt erkannt zu werden. Zugleich muß das Rücklicht aber auch in den Rahmen passen, der durch die Pixel der Menge P_F aufgespannt wird. Um auch große Lichter zu erkennen, wurde für P_F eine 15×15 Nachbarschaft gewählt. Wenn P_S annähernd so groß wie P_F gewählt wird (s. Abbildung 4.30(b)), schränkt dies den Suchraum erheblich ein. Wenn auf der anderen Seite P_S auf ein Pixel reduziert wird (s. Abbildung 4.30(a)), werden eine Menge heller Stellen im Bild gefunden, die alle potentielle Rückleuchten darstellen können. In AutoVision wurde der zweite Ansatz gewählt, da mit einem Scan über das Bild eine große Anzahl an Spotlights gefunden wird, die dann im SW Teil ausgedünnt werden kann. Eine Alternative wäre das stufenweise verkleinern (downsampling) von Eingangsbilddaten und das anschließende Absuchen der daraus resultierenden Bildpyramide mit einem Strukturelement derselben Größe, wie in Abbildung 4.31 dargestellt.

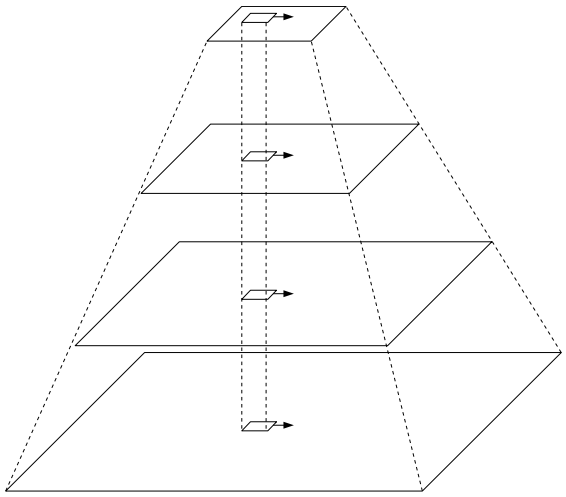
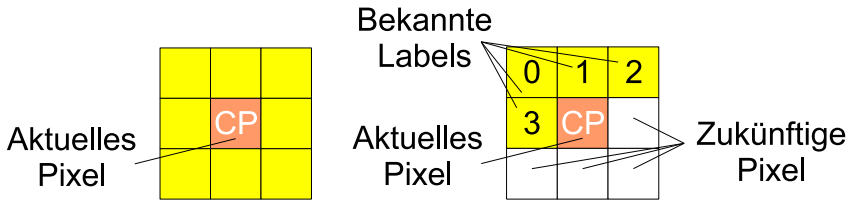


Abbildung 4.31: Scannen einer Bildpyramide mit demselben Strukturelement, zur Detektion verschieden großer Elemente

Das Resultat der Spotlight-Suche ist ein binäres Bild mit den Dimensionen des Eingangsbildes. Einzelne Spotlights können abhängig von ihrer Größe aus einem oder mehreren

Spotlight-Pixeln bestehen. Im folgenden Schritt werden die Spotlights gelabelt und in eine Liste geschrieben. Die nachfolgenden Stufen des Algorithmus arbeiten dann mit dieser Liste aus Lichtpunkten (Features) und deren Eigenschaften. Um ein Bild zu Labeln, wird das binäre Bild nach Spotlight-Pixeln (1-Pixel) durchsucht. Zusammenhängende Spotlight-Pixel werden dann zu Lichtregionen geclustert. Anschließend wird für jede Region ein eigenes Label vergeben. Die Suche nach zusammenhängenden Pixeln wird in der Bildverarbeitung als „Connected Component Labeling“ bezeichnet und ist unter anderem in [GW08] beschrieben. Eine Realisierung des „Connected Component Labeling“ auf Xilinx FPGAs ist beispielsweise in [TVCSC05] beschrieben.



(a) Nachbarschaftspixel in einer 8-er Nachbarschaft

(b) Beim Scannen von rechts nach links bereits verarbeitete Pixel und damit bekannte Labels

Abbildung 4.32: 8-er Pixel-Nachbarschaft und die beim Scannen von links nach rechts 4 bekannten Labels für das Connected Component Labeling [Alt06]

Unter zusammenhängenden Pixeln werden in dieser Arbeit Pixel verstanden, die direkt mit dem zentralen Pixel CP verbunden sind. Das sind alle Pixel in der 8-er Nachbarschaft um CP. In Abbildung 4.32(a) sind diese Pixel (gelb) dargestellt. Gehört einer der Pixel in der 8-er Nachbarschaft bereits einem Label an, erhält das zentrale Pixel ebenfalls dieses Label. Da das binäre Bild jedoch zeilenweise von links nach rechts untersucht wird, ist nur von den in Abbildung 4.32(b) gelb markierten Pixeln bekannt, zu welchem Label sie gehören. Die verschiedenen Fälle, die beim Labeln auftreten können und die daraus resultierende Vorgehensweise im Algorithmus, sind im Entscheidungsbaum in Abbildung 4.33 dargestellt.
Wie in Abbildung 4.33 zu erkennen ist, werden nur Spotlightpixel (1-Pixel) mit Labeln versehen. Die 0-Pixel werden keiner Region zugeordnet. Abhängig davon, ob die 4 Pixel links überhalb (s. Abbildung 4.32(b)) des zentralen Pixels bereits gelabelt wurden, ergibt sich die Zuweisung von CP zu einer neuen oder bereits existenten Region. Um sicher zustellen, daß alle Spotlightpixel dem richtigen Label zugeordnet wurden, muß gewartet werden bis das letzte Pixel verarbeitet wurde.
Nachdem die Rückleuchten erkannt und gelabelt sind, können verschiedene Eigenschaften, wie Anzahl der Pixel in der Region, relative Helligkeit, Position im Bild etc. miteinander verglichen werden, um auf ein Rücklichterpaar eines Fahrzeugs zu schließen.

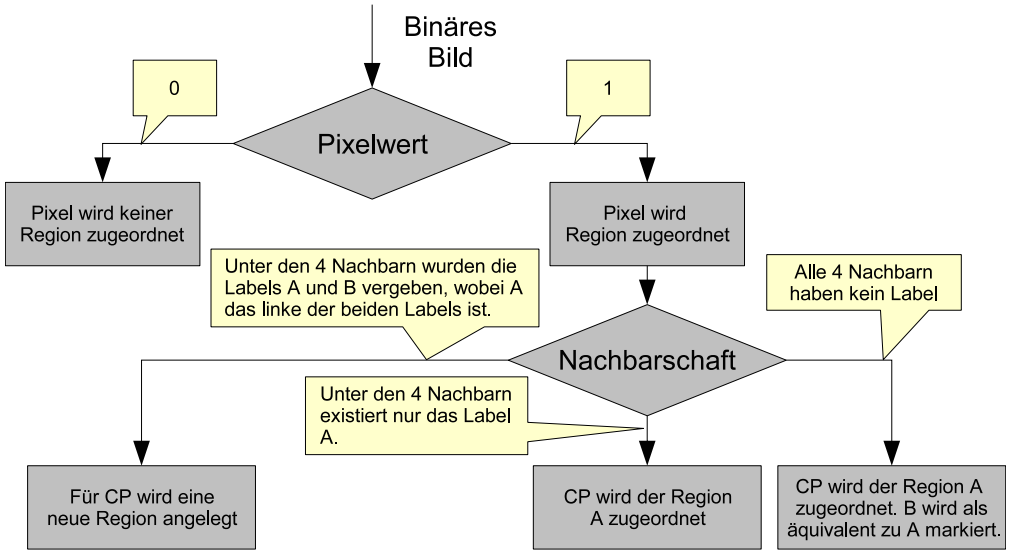


Abbildung 4.33: Entscheidungsbaum zur Vergabe von Labels

Da der Algorithmus nach Lichterpaaren sucht, können beispielsweise Motorräder nicht detektiert werden. Die erkannten Lichtregionen und Lichterpaare werden dann in das Eingangsbild gezeichnet und am Ausgang dargestellt. Ein Ergebnis des Algorithmus zeigt Abbildung 4.34.

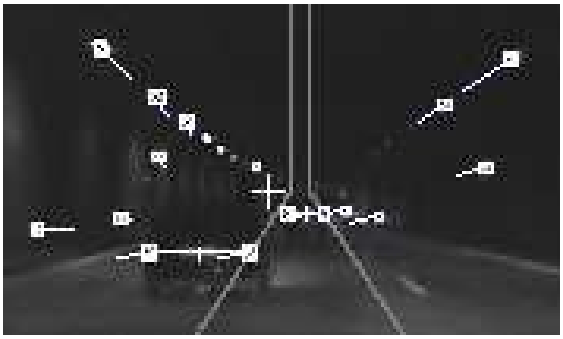


Abbildung 4.34: Ergebnis des Algorithmus zur Rücklichtdetektion. Markiert sind detektierte Lichter, ihre Bewegungsvektoren und die erkannten Fahrzeuge

In Abbildung 4.35 ist ein Ablaufdiagramm des Algorithmus zur Rücklichterkennung dargestellt. Die Aufteilung in HW- und SW-Teil wurde aufgrund eines Profiling durchgeführt. Details hierzu sind in [Alt06] aufgeführt. Durch das Profiling eines SW-Modells des Algorithmus konnten die Spotlightdetektion und das Labeln dieser Spotlights als die echtzeitkritischen Teile identifiziert werden. Der Algorithmus besteht aus den folgenden

Schritten.

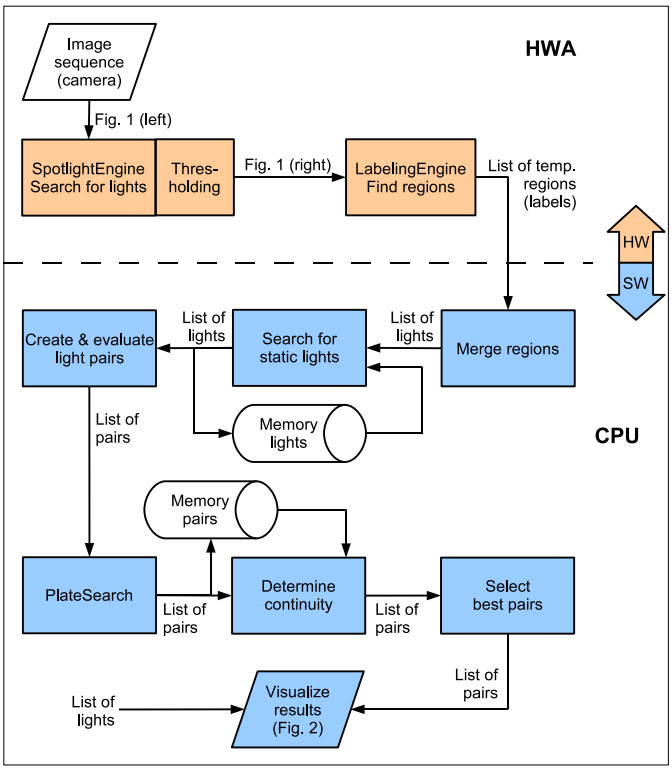


Abbildung 4.35: Ablaufdiagramm des Algorithmus zur Rücklichterkennung

Der HW Teil, der aus Spotlight-Erkennung und Labeln besteht, schreibt eine Liste an gelabelten Lichtern zurück in den Hauptspeicher. Eine CPU kann dann auf die Liste zugreifen, um statische Lichtquellen (Tunnelbeleuchtung) mit Hilfe von Bewegungsvektoren zu detektieren und um Lichter zu potentiellen Lichtpaaren zusammen zu fassen. Weitere Einzelheiten zum HW und SW Teil sind in [ACS08], [Alt06] und [Pom08] aufgeführt.

4.4.2.1 Implementierung der *TE* in HW

Der HWA für die Rücklichterkennung wird als *TaillightEngine (TE)* bezeichnet. Ein Blockdiagramm der *TE* ist in Abbildung 4.36 dargestellt.
Wie in Abbildung 4.36 zu sehen ist, besteht der HW Beschleuniger für die Rücklichterkennung ebenfalls aus der in Abschnitt 4.3 beschriebenen Modulbibliothek. Das Absuchen des Eingangsbildes mit dem Strukturelement und das anschließende Zusammenfassen und Labeln der Lichtregionen, kann durch zwei Sliding Window Operationen realisiert werden. Im Gegensatz zu der in Abschnitt 4.4.1 vorgestellten *SE*, wird für das Labeln der Regionen in der *TE* jedoch keine zweite Pipelinestufe bestehend aus *ILM*, *Matrix* und

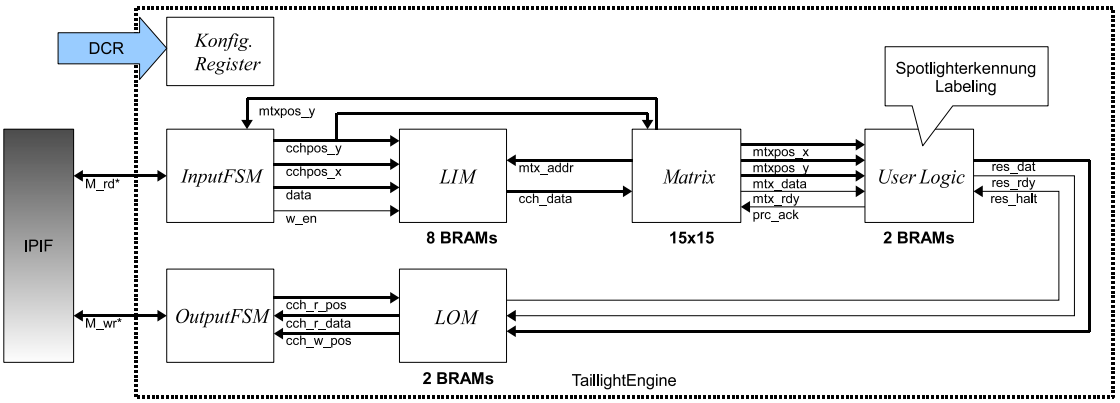


Abbildung 4.36: Blockdiagramm der TaillightEngine

separater *User Logic* verwendet. Da das Ergebnis der Spotlight Suche ein binäres Bild ist, können diese Daten bequem in einem BRAM innerhalb der ersten *User Logic* zwischen gepuffert werden, was zu einer Einsparung an Ressourcen führt. Erst wenn alle Pixel des binären Zwischenergebnisbildes durch den Labeling-Prozeß verarbeitet wurden, kann die Liste von Lichtern in den Hauptspeicher übertragen werden. Die Liste entspricht in diesem Fall also auch einer Featureliste. Jedoch wurde bei der *TE* als Ausgangsdatenpfad das *LOM* und die *OutputFSM* gewählt, anstatt des *FOM* und der *Feature Output FSM*. *FOM* und *Feature Output FSM* werden normalerweise verwendet, wenn nicht kontinuierlich Daten von der *User Logic* an den Ausgang gesendet werden. Dies ist bei einer Featuredetektion natürlich der Fall. Da jedoch alle Lichter (Feature) in der *User Logic* gespeichert werden, können diese am Ende der Verarbeitung kontinuierlich in den Hauptspeicher geschrieben werden. Daher werden, obwohl es sich um Features handelt, in diesem Fall *LOM* und *OutputFSM* verwendet.

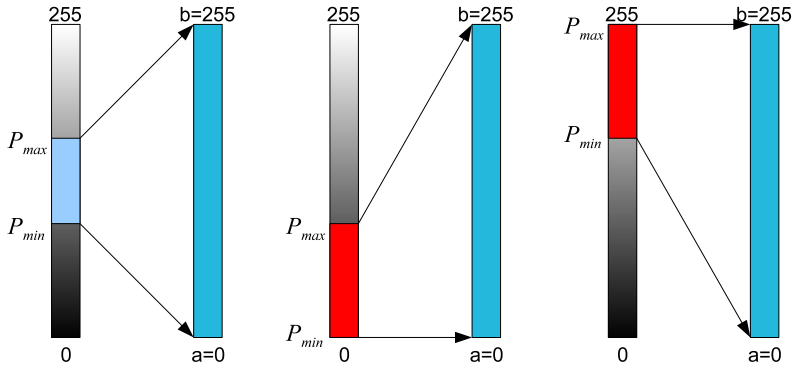
4.4.3 Kontrastveränderung mit der *ContrastEngine*

Das Einsatzgebiet des HWAs zur Kontrastveränderung, der *ContrastEngine (CTE)*, ist der dunkle Einfahrtsbereich von Tunneln. Die Aufgabe der *CTE* besteht in der Kontrasterhöhung innerhalb einer vorher definierten ROI. Die ROI kann entweder das gesamte Eingangsbild umfassen oder nur einen Teilbereich des Eingangsbildes, bespielsweise eine Tunneleinfahrt. Die Koordinaten und Größe der ROI können zur Laufzeit von anderen HWAs geliefert werden, beispielsweise von einem HWA für Tunnelerkennung.
In [Lu09] ist die Analyse und Bewertung zweier Methoden zur Kontrastveränderung beschrieben. Die beiden betrachteten Methoden sind die Histogram Equalization und das Contrast Stretching [GW08]. Aufgrund der qualitativ besseren Ergebnisse wurde das Contrast Stretching für die Implementierung in HW ausgewählt. In Gleichung 4.16 ist

die Berechnung für Histogrammspreizung dargestellt.

$$g(x,y) = (f(x,y) - P_{min}) \times \left(\frac{b-a}{P_{max} - P_{min}}\right) + a \tag{4.16}$$

$f(x,y)$ repräsentiert hierbei die Eingabefunktion, also den orginalen bzw. geglätteten Grauwert. P_{min} und P_{max} legen den Bereich des Histogramms des Eingangsbildes fest, auf den die Kontrasterhöhung angewendet werden soll. a und b geben an, auf welchen Bereich das ursprüngliche Histogramm aufgeweitet bzw. gespreizt werden soll. P_{min} und a sind dabei die unteren, b und P_{max} hingegen die oberen Grenzen.



(a) Generelles Mapping **(b)** Mapping für geringe Luminanzwerte **(c)** Mapping für hohe Luminanzwerte

Abbildung 4.37: Verschiedene Methoden des Contrast Stretching

In Abbildung 4.37(a) ist dies für den generellen Fall dargestellt. Ein bestimmter Bereich aus dem Histogramm des Eingangsbildes, der durch P_{min} und P_{max} festgelegt ist, wird aufgeweitet. Um den möglichen Grauwertbereich möglichst voll auszuschöpfen werden die minimalen und maximalen Werte a und b zu 0 bzw. 255 gewählt. Für den Einsatz der Histogrammspreizung in einer dunklen Umgebung, stellt sich die Frage nach einem passenden Wertepaar P_{min} und P_{max}. In Abbildung 4.38 ist beispielhaft eine dunkle Tunneleinfahrt und das zugehörige Histogramm dargestellt.
Wie in Abbildung 4.38 zu erkennen ist, wird in Bereichen mit schlechten Kontrastverhältnissen (Tunneleinfahrten) eher der untere Teil des Histogramms von Interesse sein. P_{min} wird daher bei dunklen Tunneleinfahrten zu 0 gewählt. Die Abbildung eines dunklen Grauwertbereichs auf das volle Grauwertspektrum bei a=0, b=255 und P_{min}=0 ist in Abbildung 4.37(b) dargestellt. Gleichung 4.16 reduziert sich demnach auf Gleichung 4.17.

$$g(x,y) = f(x,y) \times \left(\frac{255}{P_{max}}\right) \tag{4.17}$$

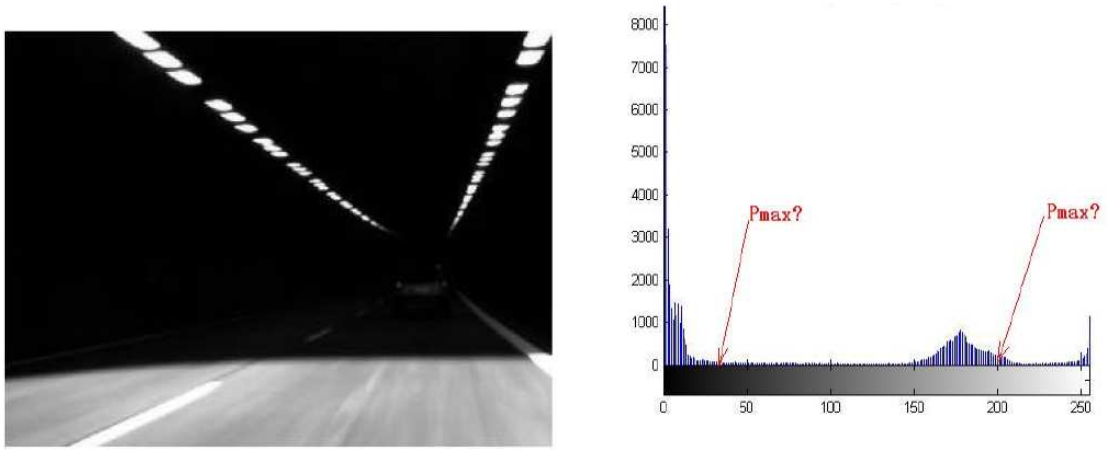


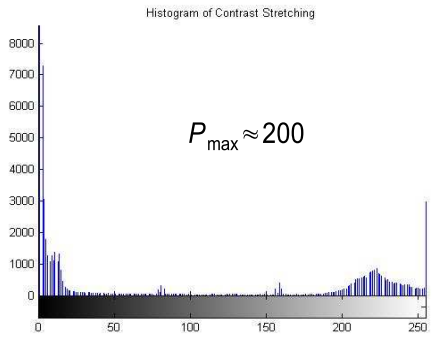
Abbildung 4.38: Eingangsbild und zugehöriges Histogramm

Die letzte Unbekannte in der Gleichung ist P_{max}. P_{max} darf nicht zu groß gewählt werden, da die Histogrammspreizung ansonsten keine Verbesserung bringt, wie in Abbildung 4.39(a) dargestellt. P_{max} wurde kurz hinter dem zweiten lokalen Maximum im Histogramm zu 200 gewählt. Sowohl im gespreizten Histogramm als auch im Ausgangsbild lassen sich keine wesentlichen Unterschiede erkennen. Bei einer Wahl von P_{max}=30, also kurz nach dem ersten lokalen Maximum, ist in Abbildung 4.39(c) ein Fahrzeug ohne Beleuchtung zu erkennen. P_{max} sollte bei dunklen Einfahrten also eher klein gewählt werden.
Um P_{max} automatisch zu bestimmen, besteht die Möglichkeit, die ROI, welche die Tunneleinfahrt repräsentiert, nach den maximal vorkommenden Grauwerten im Bereich der Tunneleinfahrt abzusuchen. Durch die Tunnelbeleuchtung wird dieser Maximalwert jedoch sehr oft über 200 liegen. Damit ist praktisch keine Histogrammspreizung mehr erkennbar, wie in Abbildung 4.39(a) dargestellt ist. Wesentlich sinnvoller ist daher die Definition eines fixen Wertes. Um Divisionen in HW zu vermeiden, wird der Ausdruck $(\frac{255}{P_{max}})$ zu 2, 4, 8 gewählt, was fixen Werten von P_{max} von etwa 128, 64 und 32 entspricht. Bei sehr hellen Eingangsbildern, beispielsweise bei Tunnelausfahrten, besteht die Möglichkeit die Histogrammspreizung für eine Kontrastreduktion zu verwenden. Dieser Fall ist in Abbildung 4.37(c) dargestellt. In diesem Fall wird P_{max} zu 255 gewählt a und b bleiben wie im vorherigen Fall bei 0 und 255. Gleichung 4.16 reduziert sich demnach auf Gleichung 4.18.

$$g(x,y) = (f(x,y) - P_{min}) \times \frac{255}{255 - P_{min}} \tag{4.18}$$

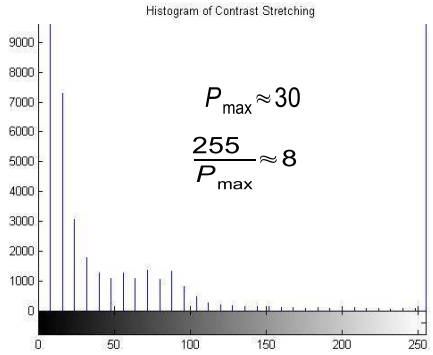
Auch hier besteht die Möglichkeit die ROI nach den minimal vorkommenden Grauwerten

(a) Verarbeitetes Bild bei $P_{max} = 200$



(b) Gespreiztes Histogramm bei $P_{max} = 200$

(c) Verarbeitetes Bild bei $P_{max} = 30$



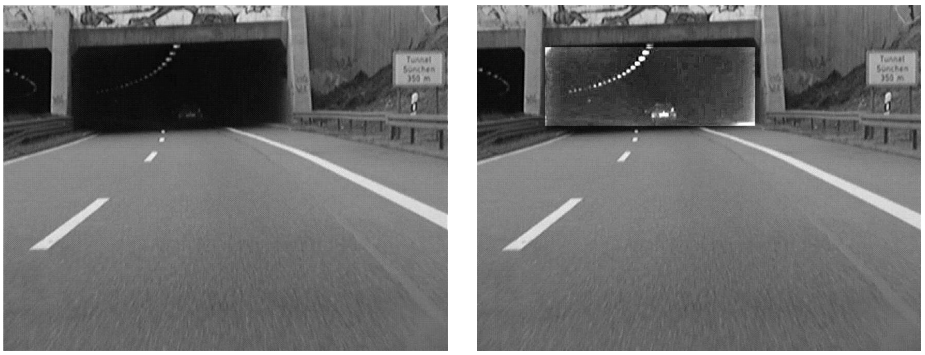
(d) Gespreiztes Histogramm bei $P_{max} = 30$

Abbildung 4.39: Verschiedene Ergebnisse der Kontrastspreizung bei unterschiedlichen Werten von P_{max}

P_{min} zu durchsuchen. Für eine einfache Implementierung in HW bietet sich jedoch auch hier an den Ausdruck $f(x,y) - P_{min}$ mit einer Zweierpotenz zu multiplizieren. Dazu wird der Ausdruck $z = \frac{255}{255-P_{min}}$ konstant auf 2, 4 und 8 gesetzt was P_{min} Werten von ungefähr 128, 192 und 224 entspricht. Bei dieser Art von Berechnung muß noch darauf geachtet werden, dass $f(x,y) > P_{min}$ ist. Daher ergibt sich für die Kontrastspreizung bei sehr hellen Bildbereichen die abschnittsweise definierte Funktion

$$g(x,y) = \begin{cases} (f(x,y) - P_{min}) \times z & f(x,y) > P_{min} \\ 0 & \text{sonst} \end{cases} \tag{4.19}$$

In Abbildung 4.40 ist ein Eingangsbild und das verarbeitete Bild bei geringen Luminanzwerten im Bereich einer Tunneleinfahrt zu sehen.



(a) Eingangsbild **(b)** Verarbeitetes Bild

Abbildung 4.40: Eingangsbild und Ergebnis der *CTE* innerhalb der ROI

Ein Problem bei der Kontrasterhöhung besteht darin, dass Bildrauschen mitverstärkt wird. Deshalb muß vor jeder Kontrasterhöhung eine Rauschunterdrückung stattfinden. Diese Rauschunterdrückung wird mit Hilfe des in Abschnitt 3.1 beschriebenen Tiefpaßfilters durchgeführt.

4.4.3.1 Implementierung der *CTE* in HW

Ein Blockdiagram der *CTE* ist in Abbildung 4.41 dargestellt.

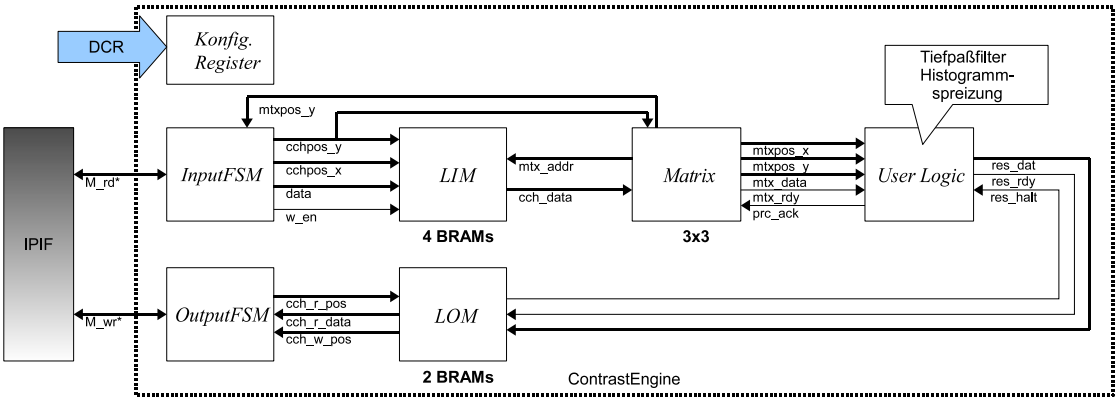


Abbildung 4.41: Blockdiagramm der ContrastEngine

Von allen HWAs ist die *CTE* die am wenigsten Komplexeste. Die *User Logic* besteht aus einem 3×3 Tiefpaßfilter und einer Logic zur Kontrastspreizung. Da die Kontrastspreizung

nicht auf Pixelumgebungen, sondern auf einzelnen Pixeln arbeitet, kann diese Operation direkt auf das Ergebnispixel des Tiefpaßfilters angewendet werden. Über die *Konfigurationsregister* können die Größe und Position der ROI festgelegt werden. Weiterführende Details über die Implementierung der ContrastEngine sind in [Lu09] aufgeführt.

4.4.4 Bewegungsdetektion mit dem *Optischen Fluß*

Bewegte Objekte wie Fußgänger oder Fahrradfahrer zu erkennen ist vor allem in städtischen Gebieten wichtig, um in zukünftigen Fahrerassistenzsystemen gegebenenfalls autonom bremsen zu können. Für die Bewegungsdetektion wird der Optische Fluß (OF) eingesetzt. Nach [HS81] ist der OF die Verteilung von Geschwindigkeitsvektoren zu bewegten Teilen oder Mustern in einem Bild. Der OF entsteht durch eine Bewegung zwischen Objekten in einer Szene und dem Betrachter. Das Grundprinzip der Berechnung des OF ist in Abbildung 4.42 dargestellt. Ein zweidimensionales Video-Frame aufgenommen zum Zeitpunkt t_k wird mit $I_k(x, y)$ und Pixel innerhalb dieses Bildes mit $i_k(x, y)$ bezeichnet. Für ein bestimmtes Pixel $i_k(x_a, y_a)$ im Videoframe $I_k(x, y)$ muss der OF Algorithmus das korrespondierende Pixel $i_k(x_b, y_b)$ im darauffolgenden Videoframe I_{k+1} finden.
Wenn eine Korrespondenz gefunden wurde, kann ein Bewegungsvektor berechnet werden, der in Pixel $i_{k+1}(x_a, y_a)$ entspringt und in Pixel $i_{k+1}(x_b, y_b)$ endet. Dieser Bewegungsvektor wird dann in Frame $I_{k+1}(x, y)$ eingezeichnet. Die Korrespondenzfindung ist in Abbildung 4.42 dargestellt.

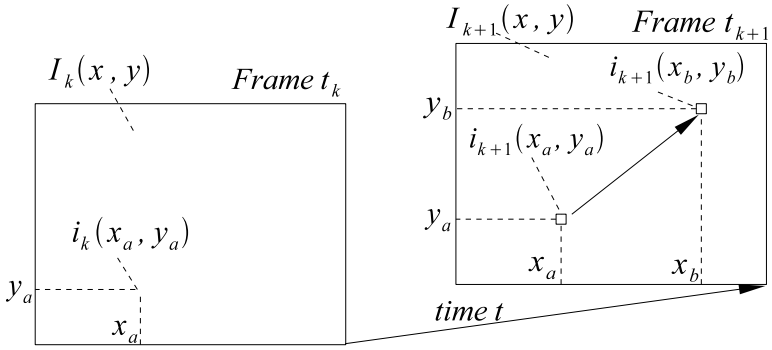


Abbildung 4.42: Korrespondenzfindung beim Optischen Fluß

Eine Übersicht von verschiedenen Algorithmen zur Berechnung des OF ist in [BB95] aufgeführt. Eine weiterer Ansatz zur Berechnung des Optischen Flusses wird in [ZW94] präsentiert. In diesem Ansatz wird die so genannte Census-Transformation verwendet. Bei der Census-Transformation wird ein Pixel durch die Helligkeitsverteilung in seiner unmittelbaren Umgebung dargestellt. Erweiterungen des in [ZW94] beschriebenen Ansatzes, führten in [Ste04] zu der Berechnung des OF. Für die Berechnung des OF in Echtzeit existieren bereits einige FPGA Implementierungen, die von tensor-basierten Me-

thoden in [WLN07], über background subtraction Methoden in [SCW+06] zu Gradienten-Techniken beschrieben in [MRAE03] reichen.
Der in dieser Arbeit verwendete Algorithmus zur Bestimmung des OF basiert auf der in [Ste04] vorgestellten Methode. Stein verwendet in [Ste04] die Census-Transformation um korrespondierende Pixel in aufeinander folgenden Video-Frames zu finden und damit den OF zu bestimmen. Dazu wird jedes Pixel $i_k(x, y)$ des Eingangsbildes $I_k(x, y)$ in eine so genannte Signatur $\xi_k(x, y)$ umgewandelt. Das Ergebnis ist das so genannte Census-Bild $\Xi_k(x, y)$. Die Transformation eines Pixels in eine Signatur geschieht wie folgt:

- Eine bestimmte Anzahl an Nachbarschafts-Pixeln, im Folgenden als Census-Kardinalität N_c bezeichnet, wird für einen Vergleich mit dem zentralen Pixel ausgewählt.

- Wird ein Nachbarschafts-Pixel mit dem zentralen Pixel verglichen, fällt eine ternäre Entscheidung. Entweder der Grauwert des Nachbarschafts-Pixels ist dem des zentralen Pixels ähnlich, er ist wesentlich kleiner oder wesentlich größer. Als ähnlich gilt ein Nachbarschafts-Pixel, wenn sein Luminanzwert innerhalb des Bereichs $[i(x, y) - \varepsilon, i(x, y) + \varepsilon]$ liegt. Wesentlich kleiner bzw. größer ist der Nachbarschafts-Pixel, wenn sein Grauwert unterhalb bzw. überhalb dieses Bereiches liegt. Die Gleichung zur Berechnung der Signatur ist in Abbildung 4.43 dargestellt. Alle N_c ternären Enscheidungen werden zusammengefügt und bilden damit die Signatur $\xi_k(x, y)$ des Pixels $i_k(x, y)$.

- Die direkten Nachbarn des zentralen Pixels fließen nicht notwendigerweise in die Berechnung der Signatur mit ein. Die Census sampling distance d_{c1} gibt daher den Abstand vom zentralen zum nächsten betrachteten Pixel an. Die Census sampling distance d_{c2} hingegen gibt den Abstand vom zentralen bis zu dem am weitesten entfernten Pixel an, dass für die Berechnung der Signatur verwendet wird. Die Wahl von d_{c1} und d_{c2} kann die Qualität der Ergebnisse stark beeinflussen, wie in [CLJS09] gezeigt werden konnte.

In Abbildung 4.43 ist die Berechnung einer Signatur mit $d_{c1} = 2$, $d_{c2} = 4$, $\varepsilon = 4$ and $N_c = 16$ dargestellt.
Die Census-Transformation wird dann auf zwei aufeinander folgende Bilder angewendet. Die beiden daraus resultierenden Bilder $\Xi_k(x, y)$ und $\Xi_{k+1}(x, y)$ werden dann in einem folgenden Matching Schritt nach gleichen Signaturen untersucht. Im Unterschied zu der in [ZW94] vorgestellten Methode, in der Hamming-Distanzen zur Korrespondenzfindung eingesetzt werden, können bei diesem Konzept Korrespondenzen allein durch den Signaturvergleich gefunden werden.
Der in [Ste04] vorgestellte Ansatz ist besonders für eine Implementierung auf einer CPU geeignet, was immer dann vorteilhaft ist, wenn viele sequentielle Anweisungen verarbeitet werden müssen.

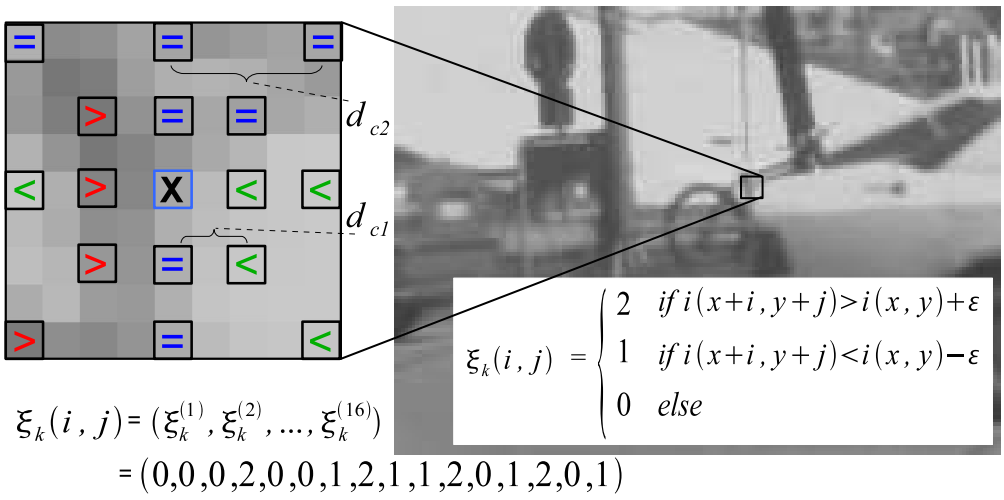


Abbildung 4.43: Beispiel einer durch die Census-Transformation erzeugten Signatur

Die grundlegenden Verarbeitungsschritte des Algorithmus werden im Folgenden dargestellt. Den ersten Schritt bildet ein Tiefpaßfilter (siehe Abschnitt 3.1) zur Rauschunterdrückung im Eingangsbild. Anschließend wird die Census-Transformation auf jedes einzelne Pixel angewandt und im letzten Schritt das Matching von Signaturen durchgeführt.
Wie in Abbildung 4.44 dargestellt, wird eine Census-Signatur $\xi_k(x, y)$ für jedes Pixel aus $I_k(x, y)$ erzeugt. Bei dieser Implementierung dient der Signaturvektor als Speicheradresse. Die Werte, die unter dieser Adresse gespeichert werden, sind die zur Signatur gehörenden Pixelkoordinaten x und y. Die Koordinaten der Pixel aus Video-Frame $I_k(x, y)$ werden dabei in der ersten Spalte abgelegt.
Ein Zähler dient zur Feststellung wie viele gleiche Signaturen im Bild vorkommen. Dieser Zähler wird ebenfalls in der ersten Spalte der Tabelle gespeichert. Wird also über dieselbe Signatur auf diese Werte zugegriffen, wird dieser Zähler erhöht, die Koordinaten jedoch nicht verändert.
Nachdem jedes Pixel in Video-Frame $I_k(x, y)$ in eine Signatur umgewandelt wurde, wird der Video-Frame $I_{k+1}(x, y)$ auf dieselbe Weise verarbeitet. Die erzeugten Signaturen für $I_{k+1}(x, y)$, sowie die jeweiligen Zähler werden in der zweiten Spalte der Tabelle abgelegt.
In einem weiteren Schritt werden Korrespondenzen ermittelt, indem über die Zähleinträge beider Spalten iteriert wird. Gesucht wird nach eindeutigen Korrespondenzen, das bedeutet Einträge, in denen beide Zähler genau 1 sind.
Die interessanten Korrespondenzen sind also diejenigen, die ein Pixel aus Bild t_k injektiv auf ein Pixel aus Bild t_{k+1} abbilden. In diesem Fall wird genau ein Pixel in Videoframe t_{k+1} gefunden mit der selben Signatur wie genau ein Pixel in Videoframe t_k. Die gefundenen Korrespondenzen werden in einer Liste abgelegt und optionale Nachverarbei-

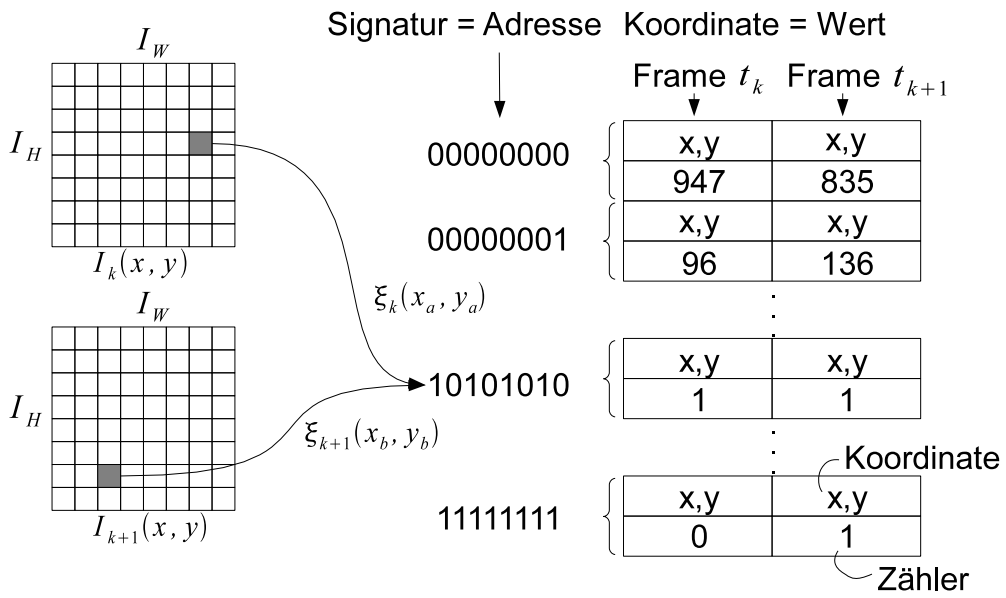


Abbildung 4.44: SW Implementierung

tungsschritte können Fehlkorrespondenzen entfernen.

Der Algorithmus in dieser Form ist für eine Verarbeitung auf einer CPU zugeschnitten. Für eine Implementierung in HW muss speziell ein Augenmerk auf die Faktoren geworfen werden, die schwer in HW zu implementieren sind und in einem hohen Performanzverlust resultieren. Bei Verwendung einer ternären Repräsentation der Signaturen werden $3^{16} \approx 26$-bit verwendet. Diese Signaturen dienen als Speicheradresse für die jeweiligen Pixelkoordinaten. Durch diese Konvention werden die Signaturen von zwei aufeinanderfolgenden Pixeln nicht an aufeinanderfolgende Adressen im Speicher geschrieben. Vielmehr führt die Verwendung von Signaturen als Speicheradresse zu hochgradig unzusammenhängenden Speicherbereichen. Daher können für das Zurückschreiben der Signaturen und den jeweiligen Pixelkoordinaten keine Burst-Transfers verwendet werden (s. Abschnitt 4.1.2), was zu einer hohen Performanzeinbuße führt. Die Änderung eines Zählerstandes in einer Tabelle kann in SW relativ einfach mit der Verwendung von Pointern realisiert werden. Für ein Zählerupdate in HW ist jeweils ein Lese- und ein Schreibtransfer über den Bus notwendig. Zudem werden bei der Verwendung solch eines **globalen Matching Verfahrens** (Finden von Korrespondenzen innerhalb des gesamten Videoframes) Bewegungsvektoren gefunden, die nahezu das gesamte Bild überspannen und oft False Positives sind. In Abbildung 4.45 ist das Problem des **globalen Matching Verfahrens** deutlich zu erkennen.

Um diese Art an Fehlkorrespondenzen zu eliminieren, wird oft der Suchraum für mögliche

Abbildung 4.45: Falsche Bewegungsvektoren an der Peripherie von Fahrzeugen bei globalem Matching Verfahren

Korrespondenzen beschränkt. Auch in [Ste04] wird die Länge der Bewegungsvektoren auf 70 Pixel limitiert. All diese Punkte zeigen, dass der Algorithmus in dieser Form ungeeignet für eine Implementierung in HW ist.

4.4.4.1 Für eine HW Implementierung optimierter Algorithmus

In HW ist es möglich, einfachere, von einander unabhängige Operationen inhärent parallel zu verarbeiten. Um Bursttransfers verwenden zu können, wurde der Algorithmus aus [Ste04] modifiziert. Anstatt die Signaturen als Speicheradressen zu verwenden, unter der die Pixelkoordinaten in Tabellenform gespeichert werden, funktioniert der modifizierte Algorithmus umgekehrt. In der HW Version dienen die Pixelkoordinaten als Speicheradresse, unter der die Signaturen, der jeweiligen Pixel gespeichert werden. In diesem Fall werden aus den x und y Koordinaten die Speicheradressen nach Gleichung 4.2 berechnet. Dieser geringfügige Unterschied ermöglicht die Verwendung von Burst-Transfers. Da das erste Pixel, das verarbeitet wurde auch wieder an die erste Stelle im Hauptspeicher zurück geschrieben wird, sind nun Ergebnistransfers in zusammenhängende Speicherbereiche möglich, was Bursttransfers ermöglicht. Dieser Schritt wird für jedes Pixel in den Videoframes $I_k(x, y)$ und $I_{k+1}(x, y)$ (siehe Abbildung 4.46) durchgeführt.
Das Matching zweier aufeinander folgender Census-Bilder $\Xi_k(x, y)$ und $\Xi_{k+1}(x, y)$ wird in einem nachfolgenden Schritt ausgeführt. Eine Signatur $\xi_k(x, y)$ aus Census-Bild $\Xi_k(x, y)$ wird gegen eine Signatur $\xi_{k+1}(x, y)$ und die umgebende (Signatur-)Nachbarschaft aus Census-Bild $\Xi_{k+1}(x, y)$ verglichen. Mit anderen Worten sucht der Algorithmus die Signatur $\xi_k(x, y)$ in der Nachbarschaft um die Signatur $\xi_{k+1}(x, y)$. In Abbildung 4.47 ist

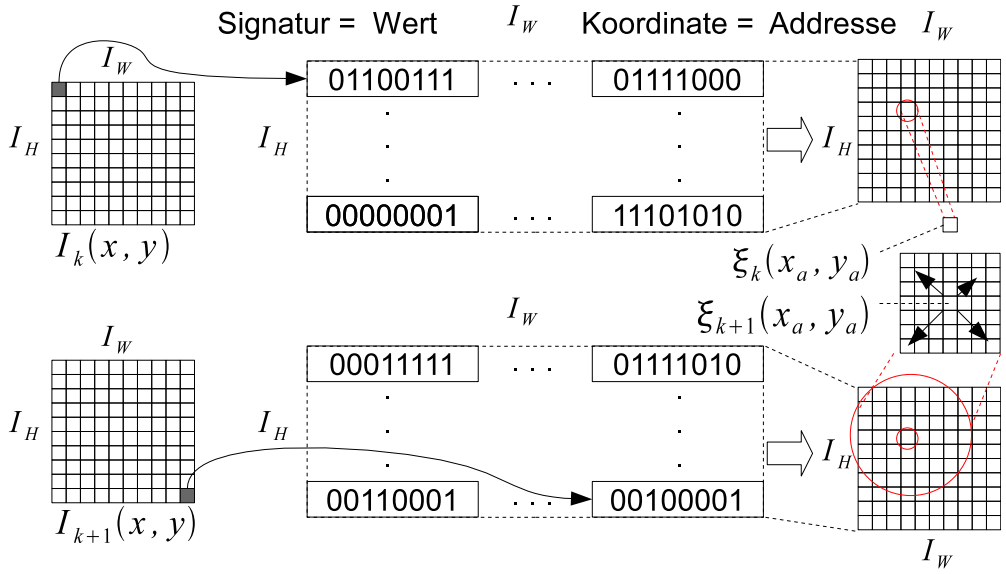


Abbildung 4.46: HW Implementation

die Signatursuche in HW exemplarisch dargestellt.
Durch die Verwendung dieses **lokalen Matching Verfahrens** (Finden von Korrespondenzen innerhalb einer Pixelnachbarschaft) sind Fehlkorrespondenzen über das gesamte Bild nicht mehr möglich. Zusätzlich kann der Vergleich der Signatur $\xi_k(x, y)$ mit allen Signaturen innerhalb der Nachbarschaft um $\xi_{k+1}(x, y)$ parallelisiert werden. Im besten Fall benötigt der Vergleich der Signatur $\xi_k(x, y)$ mit allen anderen Signaturen innerhalb der Nachbarschaft um $\xi_{k+1}(x, y)$ einen Taktzyklus. Gesucht wird nur nach injektiven Korrespondenzen, also Korrespondenzen, die genau einmal innerhalb der Nachbarschaft gefunden werden. Bei diesem Konzept wird daher weder eine Tabelle noch ein Zähler für die Anzahl an Korrespondenzen benötigt. Jedoch ist dieser Algorithmus aufgrund des Vergleichs einer einzelnen Signatur mit einer großen Nachbarschaft ungeeignet für eine Implementierung in SW. Da die Census-Transformation abgeschlossen sein muß, bevor mit dem Matching begonnen wird, werden diese beiden Operationen in zwei unterschiedlichen HWAs realisiert.

Implementierung der Census Transformation in HW Der erste HWA, die *CensusEngine (CSE)* ist für die Transformation jedes Pixels $i_k(x, y)$ in eine Signatur $\xi_k(x, y)$ verantwortlich. Dadurch, dass für jedes Pixel $i_k(x, y)$ im Videoframe $I_k(x, y)$ eine Signatur $\xi_k(x, y)$ erzeugt wird, ist das Ergebnis der *CSE* ein so genanntes Census-Bild $\Xi_k(x, y)$. Die *InputFSM* generiert die Requests und schreibt die ankommenden Pixeldaten in das *LIM*. Das *LIM* kann vier komplette Bildzeilen speichern und besteht aus 4 BRAMs. Die

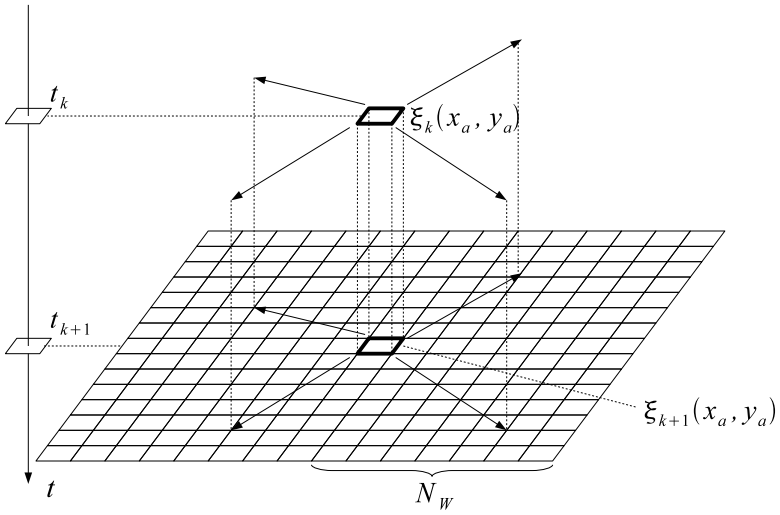


Abbildung 4.47: Signaturmatching innerhalb einer 15×15 Nachbarschaft

darauffolgende *Matrix* stellt in jedem Taktzyklus eine 3×3 Nachbarschaft für die *User Logic* zur Verfügung. Die erste *User Logic* im Falle der *CSE* ist ein Tiefpassfilter, der Rauschen aus dem Eingangsbild $I_k(x, y)$ filtern soll. Der verwendete Tiefpassfilter ist in Abschnitt 3.1 beschrieben. Anstatt die tiefpassgefilterten Pixeldaten in den Hauptspeicher zurück zu schreiben, wird in der *CSE* das Konzept des Pipelining verwendet (siehe Abschnitt 4.1.4). Die tiefpassgefilterten Pixeldaten werden daher in ein *ILM* geschrieben. Sobald genügend Daten vorhanden sind, um eine 15×15 *Matrix* der zweiten *User Logic* zur Verfügung zu stellen, kann mit der Verarbeitung begonnen werden. In der zweiten *User Logic* der *CSE* wird die Census-Transformation realisiert. Jedes tiefpassgefilterte Pixel und seine Nachbarschaft werden hierbei in eine 32-Bit Signatur $\xi_k(x, y)$ transfor-

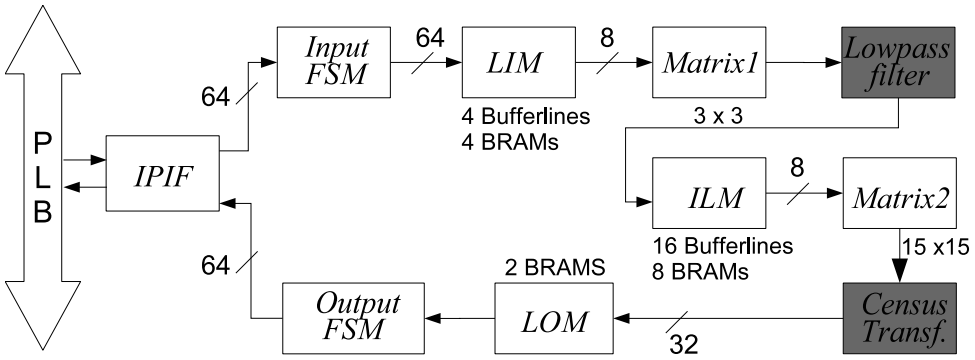


Abbildung 4.48: Blockdiagramm der CensusEngine

miert. Im *LOM* werden dann genügend Signaturen zwischengespeichert, um anschließend einen Bursttransfer über die *OutputFSM* zu starten, um das Census-Bild $\Xi_k(x, y)$ zurück in den Hauptspeicher zu schreiben. Die Daten werden daher exakt gleich verarbeitet, wie in Abschnitt 4.4.4.1 beschrieben.

Implementierung des Signatur-Matchings in HW Die Hauptaufgabe der *MatchingEngine (ME)* besteht im Finden korrespondierender Signaturen in aufeinanderfolgenden Videoframes $I_k(x, y)$ und $I_{k+1}(x, y)$. Die *ME* sucht innerhalb eines zuvor definierten Suchbereichs (15×15 in diesem Fall) nach Korrespondenzen. Ein wesentlicher Unterschied zwischen *CSE* und *ME* ist, dass Letztere zwei Eingangsdatenpfade benötigt. Dies ist notwendig, da die *ME* Daten von zwei verschiedenen Bildern lesen und temporär speichern muß. Ein Blockdiagramm der *ME* ist in Abbildung 4.49 dargestellt.

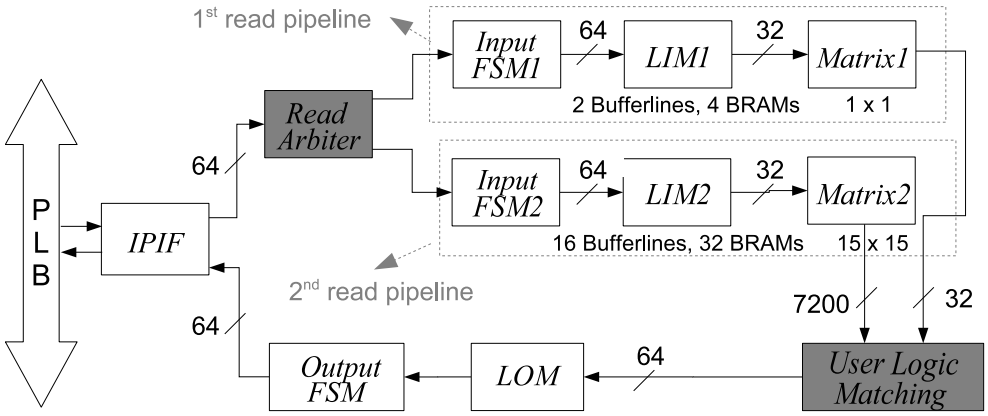


Abbildung 4.49: Blockdiagramm der MatchingEngine

Der Eingang besteht aus zwei separaten Datenpfaden. Ein *Read Arbiter* wird verwendet, um die Requestversuche der beiden *InputFSMs* zu regeln. Zusätzlich ist dieser auch für die Weiterleitung der Pixeldaten aus zwei unterschiedlichen Census-Bildern in den jeweils richtigen Eingangsdatenpfad verantwortlich. Der erste Eingangsdatenpfad fordert Daten des Census-Bildes $\Xi_k(x, y)$ und stellt der *User Logic* eine Signatur $\xi_k(x, y)$ bereit. Der Suchbereich wird durch den zweiten Eingangsdatenpfad zur Verfügung gestellt. Die *Matrix* des zweiten Eingangsdatenpfades übergibt der *User Logic* eine Signatur $\xi_{k+1}(x, y)$ aus dem Census-Bild $\Xi_{k+1}(x, y)$ samt seiner umgebenden 15×15 Nachbarschaft. Das Matching, also die Analyse, ob die Signatur $\xi_k(x_a, y_a)$ in der Suchregion genau einmal vorkommt, kann in einem Taktzyklus durchgeführt werden. Falls die 15×15 Suchregion zu klein ist, um schnellere Bewegungen zu detektieren, kann entweder die *Matrix* vergrößert, oder die Videoframerate verdoppelt werden.

4.5 Zusammenfassung

In diesem Abschnitt wurde das AutoVision System vorgestellt, ein integriertes FPGA-basiertes System zur Echtzeit-Bildverarbeitung. Die AutoVision Architektur nutzt Coprozessoren (HWAs) um performanzlastige Teile (Pixel-level Operationen) von Bildverarbeitungsalgorithmen zu beschleunigen. Dazu wurde zunächst eine SW Version der betrachteten Algorithmen erstellt, die anschließend einem Profiling unterzogen wurden. Durch das Profiling ist es möglich die performanzlastigen Teile zu identifizieren, da nur diese in Hardware umgesetzt wurden. Der verbleibende Rest der Algorithmen, der so genannte Higher-level Application Code wurde dabei in SW belassen, um auf einer eingebetteten CPU (PPC) zu laufen. Durch die Auslagerung der Pixel-level Operationen in HW konnte die Echtzeitfähigkeit (Verarbeitung eines Bildes in 32,25 ms) des Systems erreicht werden.
Der HW-Teil der betrachteten Algorithmen (die HWAs) wurde mit Hilfe einer Modulbibliothek entworfen. Einzelne Elemente dieser Bibliothek lassen sich flexibel zu einer Pixelverarbeitungspipeline zusammenschalten, um verschiedenste Operationen auf Pixel und deren Nachbarschaften zu realisieren. Dabei setzen einzelne Elemente dieser Modulbibliothek verschiedene Konzepte um, die zur Beschleunigung bzw. Datenreduktion in HW führen können. Die Elemente der Modulbibliothek lassen sich zur Designzeit über Syntheseattribute (Generics) bzw. zur Laufzeit über Konfigurationsregister an die jeweiligen Algorithmen anpassen.
Auf der einen Seite unterscheiden sich die betrachteten HWAs durch die unterschiedliche Konfiguration von Generics und die flexible Verschaltung der Bibliothekselemente sehr deutlich. Auf der anderen Seite kann durch die Verwendung der Bibliothekselemente die Entwurfszeit bei neuen HWAs verkürzt werden.
Das in diesem Kapitel vorgestellte typische Szenario zeigt, dass unter Umständen nur ein geringer Anteil der verfügbaren HWAs zu einem bestimmten Zeitpunkt benötigt wird. Um nicht unnötig logische Ressourcen für nicht benötigte HWAs zur Verfügung stellen zu müssen, wird versucht, den zu einem bestimmten Zeitpunkt gewünschten HWA gegen den nicht benötigten HWA auszutauschen. Dies wird durch die Rekonfigurationsmöglichkeit von FPGAs ermöglicht. Durch diesen Austausch lässt sich ein situationsadaptives System zur Echtzeitbildverarbeitung realisieren. Die Realisierung dieses Austauschs von HWAs, wird im folgenden Kapitel beschrieben.

5 Ein dynamisch rekonfigurierbares SoC für die Bildverarbeitung

In der AutoVision Architektur ist es möglich, HW Acceleratoren (HWAs) während der Laufzeit des Systems situationsabhängig auf dem FPGA auszutauschen. Videobasierte Fahrerassistenz ist dabei nur eines von vielen möglichen Anwendungsdomänen für den Austausch von HWAs. Weitere mögliche Anwendungsdomänen sind beispielsweise die Robotik und Software Defined Radio. Generell macht der Austausch von HWAs überall dort Sinn, wo HW Beschleunigung benötigt wird und sich die Anwendungsbereiche verschiedener HWAs gegenseitig ausschließen. Dies ist der Fall bei videobasierter Fahrerassistenz, da HW Beschleunigung notwendig ist, um Echtzeitbildverarbeitung in einem eingebetteten System im Fahrzeug zu realisieren.
Die Idee HW Module während der Laufzeit des Systems durch eine Art Zeit-Multiplex Betrieb die selben Ressourcen eines FPGAs verwenden zu lassen, erlaubt eine wesentlich größere Menge an Funktionalität, als normalerweise auf ein programmierbares Device passen würde. Dieser, in Kapitel 4 bereits erwähnte Austausch von HW Beschleunigern, wird durch die Fähigkeit der Dynamisch Partiellen Rekonfiguration (DPR) (s. Abschnitt 2.2.6) von Xilinx FPGAs ermöglicht. DPR ist die Fähigkeit einen Teil des FPGAs (partiell) während der Laufzeit (dynamisch), das bedeutet ohne Reset des Systems, auszutauschen. Während des Austauschs von HW Beschleunigern wird der Rest des Systems nicht beeinflußt. Dies bedeutet, dass parallel zum Austausch laufende Prozesse auf dem FPGA weder angehalten noch resettet werden müssen. Für die prototypische Implementierung wurden die Xilinx Virtex Plattformen gewählt, da DPR zum Zeitpunkt dieser Arbeit nur auf diesen Plattformen voll unterstützt wird. Die in diesem Kapitel vorgestellten Konzepte lassen sich jedoch auch auf zukünftige FPGA Generationen anwenden. Für einen späteren Einsatz im Fahrzeug sind beispielsweise automotive zertifizierte low-cost Devices, wie die Xilinx Spartan Familie verfügbar. Beim Entwurf von rekonfigurierbaren Systemen ist die Zeit, die für den Austausch der Module benötigt wird ein kritischer Parameter. Vor allem in Echtzeit- oder sicherheitskritischen Systemen muß die Rekonfiguration möglichst schnell durchgeführt werden. Die Rekonfiguration in eingebetteten System ist jedoch mit zusätzlichem Overhead, den so genannten Rekonfigurations-Kosten, verbunden.
Unter Rekonfigurations-Kosten wird der Overhead (Ressourcen, Latenz, Verlustleistung) verstanden, der im System zusätzlich durch DPR entsteht. Ein gewisser Teil an logischen Ressourcen (LUTs, Flip-Flops etc.) muß zunächst reserviert werden, um die DPR auf dem FPGA zu ermöglichen. Durch den Zeit-Multiplex Betrieb verschiedener HWAs wird dann

wiederum versucht, diese zusätzlichen Ressourcen zu kompensieren und im Vergleich zum nicht rekonfigurierbaren System eine zusätzliche Einsparung zu erzielen.
In diesem Kapitel werden daher Konzepte und Methoden vorgestellt, um diese so genannten Rekonfigurations-Kosten zu senken, um die Rekonfiguration bei maximaler Geschwindigkeit durchzuführen. Die im Folgenden vorgestellten Methoden zur Erhöhung des Konfigurationsdatendurchsatzes, der Ressourcen- und Verlustleitungseinsparung sind im Wesentlichen generell anwendbar. Die Methodik zur Reduktion der Bitstromgröße hingegen ist Xilinx-spezifisch. Der hier vorgestellte Ansatz lässt sich auf beliebige Anwendungsbereiche der Video- und Bilddatenverarbeitung übertragen, die der HW Beschleunigung für Echtzeitverarbeitung bedürfen. DPR macht vor allem dann Sinn, wenn sich die Beschleuniger aufgrund ihrer Einsatzgebiete gegenseitig ausschließen. Video-basierte Fahrerassistenz ist eine mögliche Anwendung, bei der die Echtzeit-Bildverarbeitung von mehreren Algorithmen in verschiedensten Fahrsituationen gewährleistet sein muß. Eine Zusammenfassung der Vorteile von DPR wird in [Kao05] gegeben. Bevor man DPR in einem eingebetteten System realisiert, sollten die folgenden Fragen beantwortet werden.

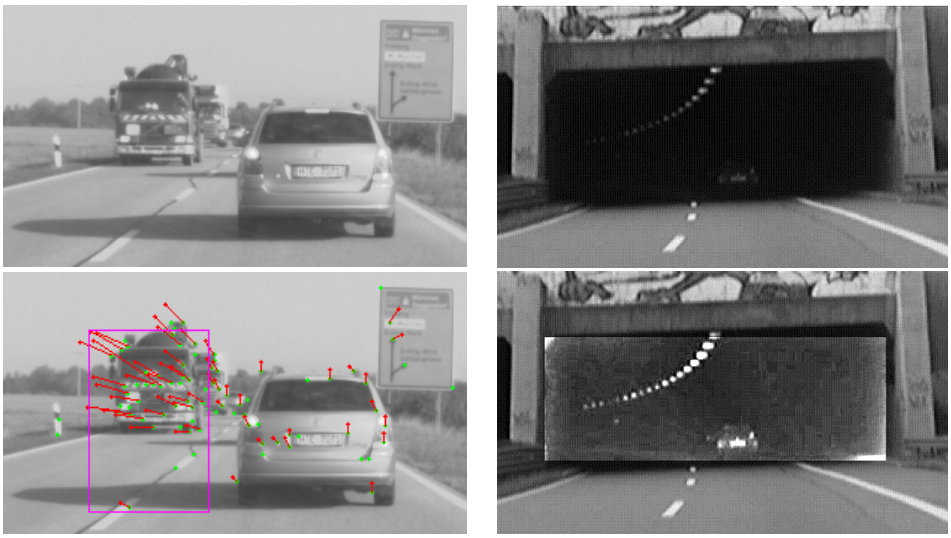
- Ist es sinnvoll DPR einzusetzen? Bedarf die Anwendung mehrerer sich gegenseitig ausschließender HWAs?
- Wo verläuft die Grenze zwischen rekonfigurierbarem und statischem Teil?
- Welche Arten von Konfigurations-Kosten entstehen und wie lassen sich diese minimieren?
- Zu welchem Zeitpunkt wird die Rekonfiguration gestartet und wie läuft der Rekonfigurations-Vorgang ab?
- Welche Schwierigkeiten entstehen im Entwurfsfluß von Rekonfigurierbaren Systemen im Gegensatz zu nicht Rekonfigurierbaren und wie können diese gelöst werden?

Diese Fragen werden in den folgenden Abschnitten dieses Kapitels aufgegriffen und beantwortet. Zunächst wird jedoch ein typisches Szenario vorgestellt, in dem es sinnvoll ist, verschiedene Bildverarbeitungsalgorithmen aufgrund der Fahrsituation einzusetzen. Diese sich gegenseitig ausschließenden Fahrsituationen stellen den idealen Einsatzort für DPR dar.

5.1 Gegenseitig ausschließende Fahrsituationen

Die folgenden Fahrsituationen dienen als Beispiel dafür, dass verschiedene Fahrsituationen unterschiedliche Bildverarbeitungsalgorithmen und damit verschiedene HW Beschleuniger benötigen, um beispielsweise Objekte auf und neben der eigenen Fahrspur zu

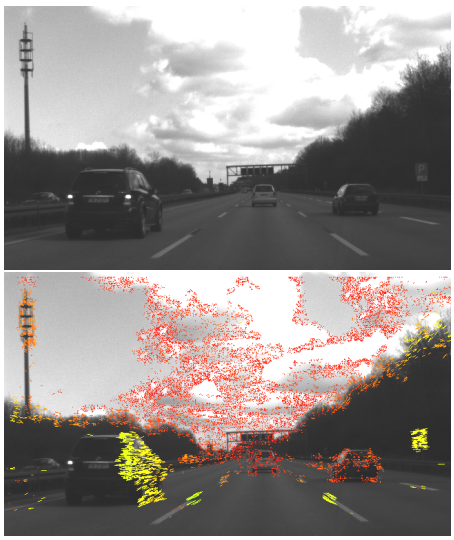
erkennen. Der einfachste Ansatz die Funktionalität für verschiedene Situationen bereitzustellen, ist die gesamte Funktionalität auf einem Device verfügbar zu machen, was wiederum die Kosten erhöht. Dieser Ansatz wird bei der ASIC Fertigung (s. [RV06]) verfolgt, da nachträgliche Änderungen in HW nicht möglich sind. Jedoch existieren auch schon in heutigen Fahrzeugen kamerabasierte Systeme, die nicht permanent genutzt werden und damit unnötig Chipfläche benötigen. Tagesabhängige Funktionalität, wie beispielsweise Nachtsichtsysteme und Fernlichtassistent werden nur Nachts bzw. in der Dämmerung verwendet. Fahrtrichtungsabhängige Funktionalität, wie der videobasierte Einparkassistent oder Lane Departure Warning (LDW) werden nur bei Rückwärts- bzw. Vorwärtsfahrt eingesetzt. Fußgängererkennung wird typischerweise in städtischen Gebieten eingesetzt, nicht auf Autobahnen. Eine Übersicht über diese gegenwärtigen FAS ist in [WHW09] aufgeführt. Zusätzlich lassen sich verschiedene andere Algorithmen vorstellen, die abhängig von der Wettersituation die Bildverarbeitung vornehmen.
Als Beispiel für eine alltägliche Fahrsituation und eine spätere prototypische Implementierung wird in AutoVision das folgende Szenario betrachtet: ein Fahrzeug fährt bei Tageslicht auf einer Autobahn. Andere Fahrzeuge können hier durch Featurepunkte (Ecken) (s. Abschnitt 4.4.1) an ihrer Peripherie erkannt werden, wie in Abbildung 5.1(a) zu erkennen ist. Kurze Zeit später erreicht das Fahrzeug einen Autobahntunnel. Hier macht es wenig Sinn Featurepunkte an der Peripherie von Fahrzeugen zu suchen, da vor dem Fahrzeug nur die dunkle Tunneleinfahrt liegt. Sinnvoller ist es hingegen, die Tunneleinfahrt zu detektieren und eine Kontrasterhöhung (s. Abschnitt 4.4.3) auf die Einfahrt anzuwenden, um eventuell liegen gebliebene Fahrzeuge zu detektieren. Dieser Fall ist in Abbildung 5.1(b) dargestellt. Im Tunnel selbst sind aufgrund der schwachen Kontrastverhältnisse die Rückleuchten der Fahrzeuge markante Featurepunkte, die von den Tunnellichtern unterschieden werden müssen. Die Detektion von Lichtpaaren (s. Abschnitt 4.4.2) ist in Abbildung 5.1(c) zu erkennen. Am Tunnelausgang könnte trotz blendendem Sonnenlicht eine Kontrastreduktion dazu führen, dass zumindest die vorausfahrenden Fahrzeuge erkannt werden können. Innerhalb von Städten kann dann eine Bewegungsdetektion mit Hilfe des Optischen Flusses verwendet werden, um Fußgänger oder Fahrradfahrer zu erkennen. Die Bewegungsdetektion mit Hilfe des Optischen Flusses (s. Abschnitt 4.4.4) ist in Abbildung 5.1(d) dargestellt. Dieses Szenario kann beliebig durch die bereits erwähnten Situationen erweitert werden. Das Szenario hat weder einen Anspruch auf Vollständigkeit, noch müssen die oben genannten Operationen genau in diesen Situationen durchgeführt werden. Es dient rein als Visualisierung dafür, dass verschiedene Fahrsituationen unterschiedlicher Algorithmen für die Bildverarbeitung bedürfen.
Für die Echtzeit-Bildverarbeitung in den oben genannten Situationen werden verschiedene HWAs benötigt. Diese wurden in Abschnitt 4.4 vorgestellt. Die HWAs sind als Bus Master an den Processor Local Bus (PLB) oder direkt an dem MPMC angeschlossen und können daher selbstständig Pixeltransfers via DMA initiieren. Simultane Lese- und Schreibtransfers werden auf dem PLB durch separate Lese- und Schreibbusse unterstützt.



(a) ShapeEngine (*SE*)

(b) ContrastEngine (*CTE*)

(c) TaillightEngine (*TE*)



(d) Optischer Fluß (*OF*)

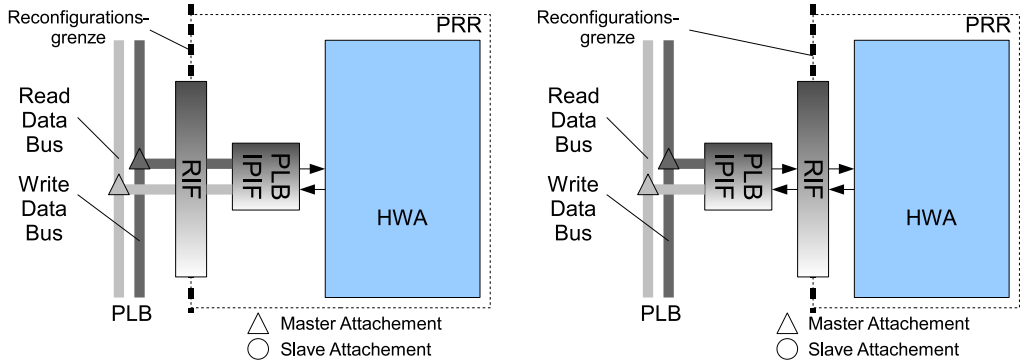
Abbildung 5.1: Originalbild und verarbeitetes Ausgangsbild der ShapeEngine 5.1(a), der ContrastEngine 5.1(b), der TaillightEngine 5.1(c) und des Optischen Flusses 5.1(d)

Die Extraktion von Featurepunkten an der Peripherie von Fahrzeugen wird von der in Abschnitt 4.4.1 vorgestellten ShapeEngine (*SE*) übernommen. Die Kontrasterhöhung im Bereich dunkler Tunneleinfahrten erfolgt durch die ContrastEngine (*CTE*) (s. Abschnitt 4.4.3). Die Pixelbearbeitung innerhalb von Tunneln oder bei Nacht, um Fahr-

zeuglichter zu erkennen, wird durch die TaillightEngine (*TE*) die in Abschnitt 4.4.2 vorgestellt wurde realisiert. Letztendlich kann der HWA für den Optischen Fluß (*OF*) zur Detektion von bewegten Objekten wie beispielsweise Fußgängern und Fahrradfahrern verwendet werden. Der HWA für den Optischen Fluß besteht aus zwei sequentiell laufenden HWAs der CensusEngine (*CSE*) und der MatchingEngine (*ME*), die in Abschnitt 4.4.4 vorgestellt wurden. Die Eingangsbilder und verarbeiteten Bilder aller HWAs sind in Abbildung 5.1 dargestellt.
Wie anhand dieses simplen Szenarios ersichtlich wird, bedürfen unterschiedliche Fahrsituationen auch unterschiedlicher Algorithmen für die Bildverarbeitung und dadurch zwangsläufig verschiedener HWAs. Daher ist es sinnvoll die benötigte Funktion nur bei Bedarf zu laden, anstatt alle HW Beschleuniger parallel zu implementieren. Die Tatsache, dass bestimmte HW Beschleuniger in manchen Situationen nicht verwendet werden, führt auf die Frage, warum diese HWAs Platz auf dem FPGA belegen sollten, obwohl sie eigentlich nicht benötigt werden. Aus diesem Grund wird die schnelle DPR eingesetzt, um die benötigten HWAs zur Laufzeit zu laden, während gleichzeitig unbenutzte Teile entfernt werden.

5.2 Festlegung der Rekonfigurations-Grenze

Bei der Erstellung eines DPR Designs müssen zur Designzeit die Rekonfigurations-Grenzen feststehen. Das bedeutet auch, daß die Elemente des statischen und rekonfigurierbaren Anteils festgelegt sein müssen. In Abbildung 5.2 sind zwei mögliche Rekonfigurations-Grenzen bei einem beispielhaften Anschluß eines HWAs an den PLB abgebildet. Dieselben Fälle sind bei einem Anschluß an den MPMC über ein LIS NP IPIF vorstellbar.



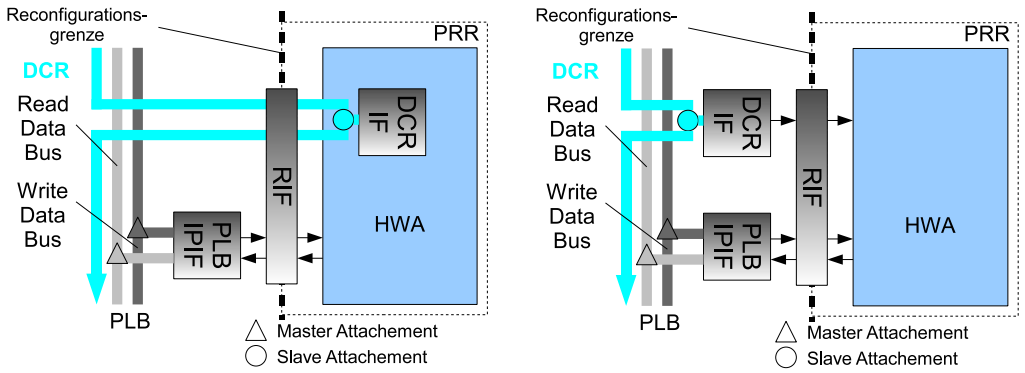
(a) Rekonfigurations-Grenze zwischen PLB und IPIF

(b) Rekonfigurations-Grenze zwischen IPIF und HWA

Abbildung 5.2: Mögliche Rekonfigurations-Grenzen

In Abbildung 5.2(a) liegt die Rekonfigurations-Grenze zwischen PLB und IPIF. In Ab-

bildung 5.2(b) dagegen zwischen IPIF und HWA. Die Festlegung der Rekonfigurations-Grenze hat einen entscheidenden Einfluß auf die Größe der PRR, und somit auch auf die zu erwartende Bitstromgröße und letztendlich auch auf die Rekonfigurations-Zeit. In Abschnitt 4.4 sind die verschiedenen HW Beschleuniger dargestellt. Anhand der Blockdiagramme läßt sich leicht erkennen, daß sich die HWAs (Bsp.: *ME* und *CSE*) bereits nach dem IPIF unterscheiden. Aufgrund der Konfiguration durch die Generics sind weitere Gemeinsamkeiten zwischen den einzelnen HWAs nur mit sehr großem Aufwand identifizierbar. Zur Identifikation welche Strukturen zwischen den einzelnen HWAs noch gleich sind, kann beispielsweise der in [RM06] beschriebene Ansatz verwendet werden. Dort wird versucht auf Netzlistenebene Gemeinsamkeiten zwischen zwei verschiedenen Designs zu erkennen und diese dann dementsprechend gleich zu platzieren und zu verdrahten. Diese gleich verdrahteten Anteile werden dann nicht mehr in die partiellen Bitströme übernommen. Allerdings werden bei dieser Methodik immer nur die Gemeinsamkeiten zwischen zwei PRMs untersucht. Da sich die Engines signifikant hinter dem IPIF unterscheiden (Bsp.: *InputFSM* bei *TE*, *Read Arbiter* bei *ME*) wird die Rekonfigurations-Grenze direkt hinter das IPIF gelegt, wie in Abbildung 5.2(b) dargestellt.



(a) DCR IF innerhalb des PRM **(b)** DCR IF außerhalb des PRM

Abbildung 5.3: Mögliche Anbindung eines PRM an den DCR Bus

Wie in den Blockschaltbildern der HWAs in Abschnitt 4.4 ebenfalls zu erkennen ist, befindet sich das DCR IF innerhalb der HWA Wrapper. Um die HWAs zu rekonfigurieren, ist in diesem Fall der gesamte DCR Bus über das entsprechende Rekonfigurations-Interface (*RIF*, s. Abschnitt 5.3.1.3) zu führen, wie in Abbildung 5.3(a) dargestellt. Der DCR arbeitet jedoch nach dem Daisy Chain Prinzip. Würde ein DCR Slave (wie bei der DPR) kurzfristig aus dieser Kette herausgenommen, so wäre die Daisy Chain unterbrochen. Um dies zu vermeiden, wird das DCR Slave Interface außerhalb des Engine Wrappers platziert und im statischen Teil des Systems untergebracht (s. Abbildung 5.3(b)). Durch diese Designentscheidung werden im Sinne einer Vorfilterung nur noch die Konfigurati-

onsdaten an die Engine übermittelt, die auch für diese bestimmt sind. Zusammenfassend ist festzustellen, dass beim Austausch eines PRMs gegen ein anderes alle Interfaces zu Kommunikationsressourcen außerhalb der PRR zu platzieren sind.

5.3 Reduktion von Rekonfigurations-Kosten

Die Verwendung von DPR in einem FPGA-basierten System ist, wie bereits erwähnt, mit zusätzlichen Rekonfigurations-Kosten verbunden. Unter Rekonfigurations-Kosten wird in dieser Arbeit der Zusatz an Fläche, Zeit und Verlustleistung verstanden, der durch die Einführung von DPR im System entsteht. In den folgenden Abschnitten werden daher die Beiträge zur Fläche, der Zeit und Verlustleistung, sowie Konzepte zu deren Reduktion vorgestellt.

5.3.1 DPR Komponenten (DPR-Overhead)

Die in diesem Abschnitt aufgeführten Komponenten, werden im Folgenden als DPR Overhead bezeichnet. Diese Komponenten werden dem System zunächst hinzugefügt, um später eine Reduktion der logischen Ressourcen durch DPR zu erzielen. Um eine möglichst große Einsparung an logischen Ressourcen zu erhalten, muß der DPR Overhead so gering wie möglich sein.

5.3.1.1 IP Interface (IPIF)

Ebenso wie bei den Engines ist es möglich, den in Abschnitt 5.3.1.2 beschriebenen ICAP Controller selektiv an den PLB Bus oder direkt an den MPMC anzubinden. Dies geschieht über ein IP Interface (IPIF). Als IPIF werden in dieser Arbeit das LIS PLB IPIF [Zep07] oder das LIS NP IPIF [Alt09a] verstanden. Eine detaillierte Beschreibung der Interfaces ist in Abschnitt 4.2.2 zu finden. Falls der Durchsatz des angeschlossenen IP cores wichtiger ist, als die verbrauchten Ressourcen, sollte der Core über ein LIS NP IPIF angeschlossen werden. Beim Einsatz des LIS NPIs wird eine maximale Burstweite B_W von 32 und Address Pipelining unterstützt. Eine ressourcensparende, aber dennoch performante Lösung ist durch den Einsatz eines LIS PLB IPIF gegeben.

5.3.1.2 ICAP Controller

Um den FPGA intern zu (re-)konfigurieren muß eine Schnittstelle zum Konfigurations-Layer existieren. Diese Schnittstelle bildet der in Abschnitt 2.2.3 beschriebene Internal Configuration Access Port (ICAP). Der Controller, um ICAP die notwendigen Konfigurationsdaten zu liefern, besteht aus einer FSM zur Generierung der Requests, einem FIFO, das als temporärer Speicher genutzt wird, einem DCR Interface für die Übertragung für Kontrolldaten und einem CRC IP Core zur Laufzeitverifikation der Bitstromdaten.

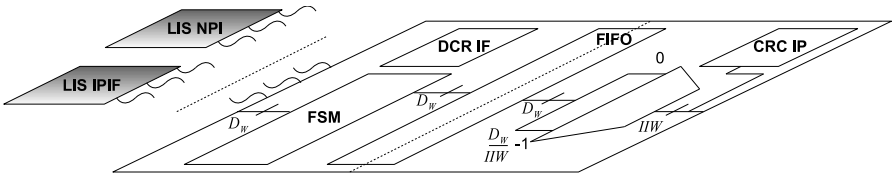


Abbildung 5.4: Blockdiagramm des ICAP Controllers

Die Burstweite B_W, die Datenweite D_W der ankommenden Daten und die ICAP Input Weite IIW sind über Syntheseattribute, so genannte Generics, konfigurierbar, um möglichst viele Plattformen zu unterstützen [CZS+08].
Durch den modularen Aufbau und die Wahl der entsprechenden Generics ist es auch beim ICAP Controller möglich, einen Anschluß an den PLB, oder direkt an den MPMC zu realisieren. Neben den zusätzlichen Ressourcen besteht der einzige Unterschied bei einem Direktanschluß an den MPMC darin, dass Bursts mit einer Weite von B_W=32 Bit und Address Pipelining unterstützt werden.
Das DCR Interface wird für die Kommunikation zwischen PPC und ICAP Controller verwendet. Der PPC sendet die Startadresse des zu ladenden partiellen Bitstroms zusammen mit der Anzahl an zu übertragenden Bursts über den DCR in einem 32-Bit Wort an den ICAP Controller. Diese Kontrolldaten dienen gleichzeitig als Startsignal für den Rekonfigurationsvorgang. An dieser Stelle sei darauf hingewiesen, dass der in dieser Arbeit vorgestellte ICAP Controller weder einen Microprozessor für den Start der Rekonfiguration noch einen Bus wie den PLB zur Datenübertragung benötigt. Durch den modularen Aufbau lässt sich der Controller nicht nur an den MPMC, sondern beispielsweise auch an ein Network on chip (NoC) oder einen Crossbar Switch anbinden. Die Modularität des ICAP Controllers erlaubt ein einfaches Entfernen des DCR Interfaces und eine einfache Anpassung an jede beliebige Interconnect Struktur, um das notwendige 32-Bit Wort zum Start der Rekonfiguration, von jedem beliebigen Modul im SoC senden zu können.
Sobald die Kontrolldaten vom ICAP Controller empfangen wurden, wird der erste Request von der FSM generiert. Über DMA Transfers können Bitstromdaten direkt aus dem Hauptspeicher übertragen werden, ohne daß der PPC in den Transfer involviert werden muß. Dies führt zu einer Auslastungsminimierung auf dem PPC, der sich somit auf die Verarbeitung der Higher Level Algorithmen beschränken kann. Da sowohl der ICAP Controller als auch der MPMC Bursts (s. Abschnitt 2.3.2.1) und Address Pipelining (s. Abschnitt 2.3.2.2) unterstützen, können theoretisch D_W Bit zum ICAP Controller übertragen werden. Da die IIW jedoch in vielen Fällen geringer ist als D_W, sorgt ein Multiplexer am Ausgang dafür, dass die Pakete richtig aufgeteilt und in der richtigen Reihenfolge an ICAP geliefert werden. Sobald gültige Datenworte am Eingang des ICAP Controllers anliegen, ist die FSM für das Schreiben der Daten ins FIFO zuständig und damit auch für die interne Adressgenerierung.

Das FIFO muß die Möglichkeit haben D_W ankommende Bit vom PLB oder direkt vom MPMC zu speichern. Da in V2P und V4 Devices die BRAMs eine maximale Datenbreite von 32 Bit aufweisen, werden für die Implementierung des FIFOs bei einem 64 Bit breiten Anschluß zwei parallele BRAMs verwendet. In V5 Devices ist die maximale Eingangsdatenbreite der BRAMs 64 Bit, so daß bei einem $D_W = 64$ Bit breiten Bus nur noch ein BRAM notwendig ist. In V4 und V5 Devices existieren zusätzlich in jedem BRAM dedizierte, hart verdrahtete FIFO Controller für die Implementierung von schnellen synchronen und asynchronen FIFOs. In [Alf08] wird eine Übersicht über FIFOs in V5 Devices gegeben. Um den Übergang zwischen zwei unterschiedlichen Taktdomämen zu realisieren, werden auch im ICAP Controller asynchrone Fifos verwendet (s. Abschnitt 2.3.3). Damit ist es möglich die Bitstromdaten mit einer Frequenz f_r unabhängig von der Schreibfrequenz f_w auszulesen. Ein Füllstandsanzeiger im FIFO wird verwendet, um die Requestgenerierung der FSM zu steuern. Passt ein kompletter Burst in das FIFO, wird ein neuer Request gestartet. Ist das FIFO annähernd voll, wird auf einen erneuten Request vorerst verzichtet.
Um zu verifizieren, ob die Bitstromdaten korrekt am ICAP Controller angekommen sind, wird ein Modul (CRC IP) zur Laufzeitverifikation der Daten verwendet. Dieses Modul berechnet während der Rekonfiguration einen CRC-Wert und gehört als Teil des ICAP Controllers ebenfalls zum DPR Overhead. Das CRC Modul ist detailliert in Abschnitt 5.3.2.3 beschrieben. In [CZS+08] und [Alt09b] sind weitere Implementierungsdetails des ICAP Controllers aufgeführt.

5.3.1.3 Rekonfigurations-Interface (RIF)

Um vor, während und nach der Rekonfiguration eine sichere Verbindung zwischen rekonfigurierbarem und statischem Teil sicherzustellen, wird ein Rekonfigurations-Interface *(RIF)* verwendet. Das *RIF* sitzt genau auf der Rekonfigurationsgrenze (s. Abschnitt 5.2) und etabliert eine sichere Verbindung zwischen statischem Teil und PRM.
In Abbildung 5.5 sind die (je nach verwendetem Designflow) optionalen, unidirektionalen Busmacros [HBB04] zu erkennen. Da in diesem Beispiel das PRM rechts des statischen Teils liegt, werden die Signale aus dem PRM in den statischen Teil von rechts nach links (R2L) übertragen. Dementsprechend findet die Übertragung der Signale aus dem statischen Teil in das PRM von links nach rechts (L2R) statt.
Während der Rekonfiguration können sich Signale zwischen statischem Teil und PRM unvorhersehbar ändern. Kritisch sind insbesondere die Netze, die ein PRM verlassen, da sich Signaländerungen direkt auf den statischen Teil auswirken und zu Fehlern führen können. Um die Störeinflüsse genau daran zu hindern, muß eine Vorrichtung existieren, die diese Netze während der Rekonfiguration vom statischen Teil abtrennt. In [Xil10c] wird daher die Verwendung eines Enable Signals empfohlen, das den statischen vom rekonfigurierbaren Teil während der DPR abtrennt. Die dazu notwendige Vorrichtung im *RIF* wird im Folgenden als Disconnector bezeichnet und ist ebenfalls in Abbildung 5.5

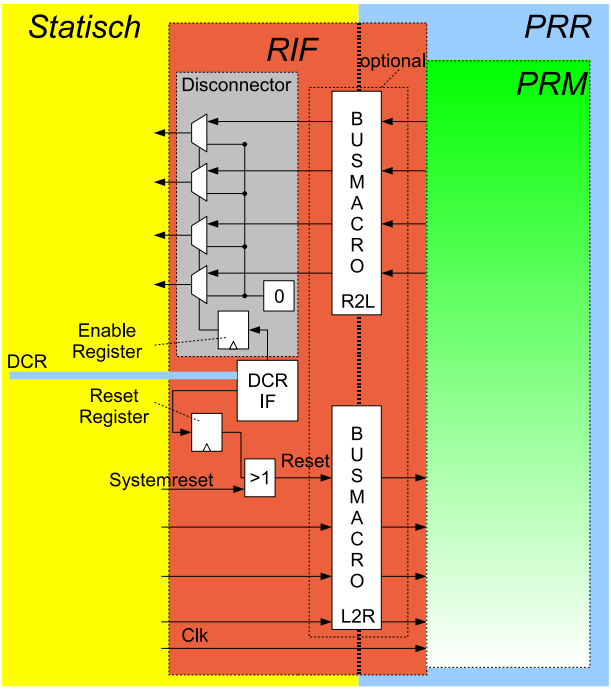


Abbildung 5.5: *RIF* für ein rekonfigurierbares Modul rechts des statischen Teils

dargestellt. Der Disconnector besteht im Wesentlichen aus einer Multiplexer-Struktur, um die vom PRM ausgehenden Signale bei Bedarf über ein DCR Interface durch- bzw. auf konstant '0' zu schalten. Die Steuerung der Multiplexer erfolgt über das Enable Register, welches vom DCR Interface gespeist wird. Somit ist es einfach möglich, die Signale über die Rekonfigurationsgrenze vom PPC (DCR-Master) aus zu trennen. Werden, wie beim MBRF (s. Abschnitt 2.2.7.2) Busmacros verwendet, so bietet sich eine ressourcenschonende Implementierung der Multiplexer-Struktur des Disconnectors direkt in den LUTs der Macros an. Da im Partitions-basierten Flow keine Macros mehr notwendig sind (s. Abschnitt 2.2.7.3), muß in diesem Fall der Disconnector separat implementiert werden.

Um während der Rekonfiguration das PRM zu resetten, kann selektiv über das DCR Interface ein Reset (high-aktiv) initiiert werden. Dieses Resetsignal wird im Resetregister gespeichert. Der Ausgang des Resetregisters, sowie der Reset aus dem statischen Teil (häufig das Bus-Resetsignal), werden über ein Oder-Gatter an das PRM übertragen. Ist einer der beiden Resets (oder beide gemeinsam) aktiv, wird das PRM resettet. Das DCR IF innerhalb des *RIF* kann zur Steuerung weiterer Enable Signale verwendet werden. Beispielsweise könnten über ein Enable Signal ganze Äste eines Taktbaums abgeschaltet werden, wie in Abschnitt 5.3.3 beschrieben.

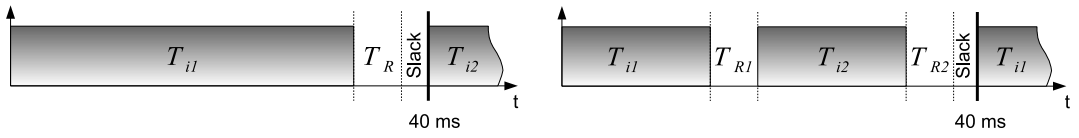
5.3.2 DPR Latenz

In diesem Abschnitt wird der zeitliche Overhead vorgestellt, der durch DPR dem System hinzugefügt wird. Zunächst wird beschrieben von welchen Faktoren die Rekonfigurations-Zeit abhängt. Anschließend werden Konzepte präsentiert, mit denen versucht wird die Rekonfigurations-Zeit zu minimieren.

5.3.2.1 Formalisierung des Rekonfigurations-Durchsatzes

Wie in Abschnitt 5.1 bereits vorgestellt, bedürfen verschiedene Fahrsituationen unterschiedlicher Algorithmen und damit unterschiedlicher HW Beschleuniger für die Bildverarbeitung. Bei der Verwendung von DPR in videobasierten Fahrerassistenzsystemen, besonders wenn es sich um sicherheitskritische Funktionen handelt, darf durch DPR kein Videoframe verloren gehen. Diese Bedingung führt zu der Anforderung, dass die Rekonfiguration so schnell wie möglich durchgeführt wird, zumindest aber so schnell, dass die Echtzeitanforderungen (Verarbeitung von 31 fps = 32.25 ms Verarbeitungszeit für einen Videoframe) nicht verletzt werden. Auch nicht bei mehreren Rekonfigurationen pro Video-Frame. Abhängig davon, wann und wie oft ein System rekonfiguriert wird, spricht man von *Inter Video Frame Reconfiguration* oder *Intra Video Frame Reconfiguration.*

Inter Video Frame Reconfiguration (InterVFR) Die Inter Video Frame Rekonfiguration (InterVFR) ist definiert als der gegenseitige Austausch zweier Module *zwischen zwei aufeinander folgenden* Videoframes. In Abbildung 5.6(a) wird die Verarbeitungszeit des ersten HWAs als T_{i1} definiert. Die Ausführungszeit eines zweiten HWAs wird als T_{i2} definiert. Die Rekonfigurations-Zeit, die für den Austausch der Module benötigt wird ist definiert als T_R. Wenn also $T_{i1} + T_R < 32,25ms$ (31 fps) ist, ist die InterVFR möglich. Das bedeutet, dass sowohl die Bildverarbeitung mit der ersten Engine als auch die nachfolgende Rekonfiguration abgeschlossen werden kann, bevor die Verarbeitung mit der zweiten Engine beginnt. Die InterVFR Rekonfiguration wird für den Austausch von *CTE*, *SE* und *TE* eingesetzt.



(a) InterVFR **(b)** IntraVFR

Abbildung 5.6: Inter und Intra Video Frame Rekonfiguration

Intra Video Frame Reconfiguration (Inter VFR) Die Intra Video Frame Rekonfiguration (IntraVFR) ist definiert als der gegenseitige Austausch zweier Module *innerhalb*

eines Videoframes. Der Optische Fluß [CLJS09] als auch die in [Asc09] beschriebene Bildkompression stellen mögliche Anwendungen für die InraVFR dar. Wie in Abschnitt 4.4.4 bereits beschrieben, besteht der HWA für den OF aus zwei sequentiell arbeitenden Engines, der *CSE* und der *ME*. Die Zeit, um ein Bild mit der *CSE* zu verarbeiten ist definiert als T_{i1}. Die benötigte Zeit für die Verarbeitung mit der *ME* sei T_{i2}. Da die partiellen Bitstromgrößen zweier HWAs im Normalfall nicht gleich sind, unterscheiden sich auch die jeweiligen Rekonfigurations-Zeiten. Die Zeit für die erste Rekonfiguration wird daher mit T_{R1}, die Zeit für die zweite Rekonfiguration mit T_{R2} bezeichnet. Der Ablauf der IntraVFR ist in Abbildung 5.6(b) beschrieben. Wenn $T_{i1} + T_{i2} + T_{R1} + T_{R2} < 32,25ms$ (31 fps) erfüllt ist, ist die IntraVFR realisierbar. IntraVFR is generell dann sinnvoll einsetzbar, wenn ein HWA auf ein globales Resultat eines anderen HWAs warten muss. Dies ist dann der Fall, wenn die beiden Operationen nicht pipelinebar sind. Die Konzepte zu InterVFR und IntraVFR wurden in [CAAS10] und [CAS10] vorgestellt.
Für die schnelle Rekonfiguration ist der Zugriff auf den Speicher, in dem die partiellen Bitströme vorgehalten werden ein wichtiges Kriterium. Beim Speichern der partiellen Bitströme im on-chip BRAM ist zwar ein schneller Zugriff auf die Daten gegeben, jedoch ist diese Ressource bei mehreren großen partiellen Bitströmen schnell erschöpft. Des Weiteren sollten die Ressourcen, die durch DPR dem System hinzugefügt werden, minimal gehalten werden (s. Abschnitt 5.3.1). Eine Alternative besteht im Speichern der partiellen Bitströme in einem off-chip Flash Speicher. Hier könnten problemlos alle partiellen Bitstromdaten gespeichert werden. Dadurch werden keine Ressourcen für die Speicherung auf dem FPGA benötigt. Flash Speicher sind im Normalfall langsamer als SRAM oder SDRAM und daher nur bedingt geeignet. Eine weitere Möglichkeit besteht im Speichern der partiellen Bitströme in einem off-chip SRAM oder DDR/DDR2 SDRAM. Da große SRAM Speicher im Vergleich zu DDR SDRAM sehr teuer sind, weil das DDR SDRAM ohnehin bereits im System zum Puffern von Bilddaten vorhanden ist, werden die partiellen Bitströme dort gespeichert. Da es sich bei BRAM, SRAM und DDR SDRAM um flüchtige Speicher handelt, müssen die partiellen Bitströme während der Initialisierung in die entsprechenden Speicher übertragen werden. Dies kann beispielsweise über Ethernet [BCY+09] oder durch Speichern der Bitströme in einem Flash Speicher geschehen. Neben dem Speichern aller partiellen Bitstromdaten im Flash Speicher, wird häufig ein lokales Puffer kurz vor dem Konfigurations-Interface verwendet. Dieses Puffer dient zum temporären Speichern von Teilen des zu ladenden partiellen Bitstroms.
Bei der on-chip Rekonfiguration verwenden die meisten der in der Literatur beschriebenen Ansätze ein FIFO als lokalen Bitstromspeicher [LPF09a, LKLJ09]. Entweder als temporäres Puffer oder um, wie bereits erwähnt, die Bitstromdaten komplett im on-chip BRAM abzulegen. In Abbildung 5.7 ist schemenhaft der Aufbau eines ICAP Controllers mit lokalem Speicher (FIFO) dargestellt. Der Aufbau entspricht dem in Abbildung 5.4.
Anstatt die Bitstromdaten wie beispielsweise in [LKLJ09] komplett im on-chip BRAM abzulegen, werden diese gepipelined vom Hauptspeicher in das FIFO geladen. Da der Flaschenhals in den meisten Systemen im Transfer von Bitstromdaten zum Konfigurations-

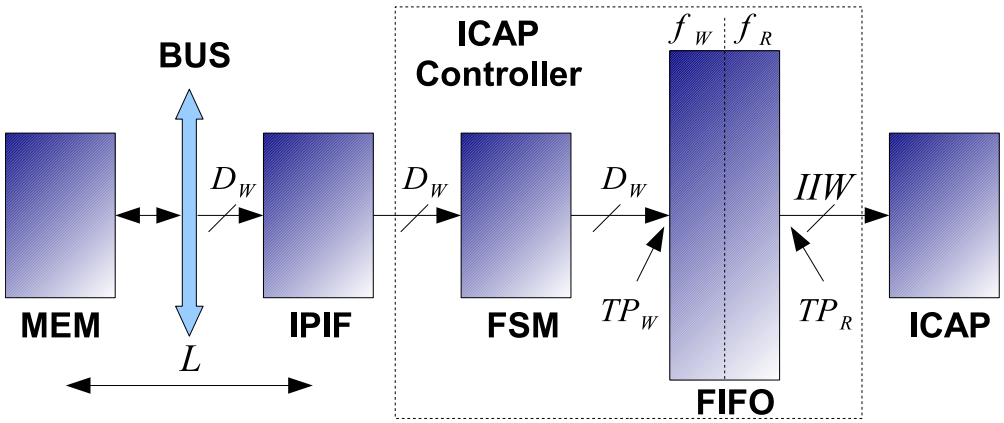


Abbildung 5.7: Blockdiagramm eines ICAP Controllers

Interface (ICAP) besteht, konzentrieren sich viele Autoren auf Techniken, die das FIFO über den gesamten Zeitraum der Rekonfiguration gefüllt halten. Dies wird u.a. durch einen Direktanschluß des ICAP Controllers an einen externen Speicher [LPF09a] oder durch Bitstromkompression [LH01, HUWJ04, PMW04] erreicht.
Die Menge an Daten, die pro Takt in das FIFO geschrieben werden kann ist abhängig von der Datenweite des Busses D_W, der Burstweite B_W und der Latenz L. L ist die Anzahl an Taktzyklen, die vom ersten Request bis zum Zeitpunkt vergeht, an dem die Daten am IP Core zur Verfügung stehen. f_w bezeichnet die Frequenz, die verwendet wird, um eine bestimmte Menge an Daten in das FIFO zu schreiben. f_w ist üblicherweise gleich der Busfrequenz. Mit Hilfe der Gleichung 5.1 kann der Durchsatz am Eingang des FIFOs TP_w, also die Menge an Daten pro Takt, die in das FIFO geschrieben wird, berechnet werden.

$$TP_w = \frac{B_W[cycles] * D_W[bit]}{B_W[cycles] + L[cycles]} * f_w \tag{5.1}$$

Die Busfrequenz wird übelicherweise zu 100 MHz gewählt, was auch der maximal spezifizierten Betriebsfrequenz von ICAP auf V4 und V5 entspricht [Xil10c]. Im Vergleich dazu ist der Durchsatz am Ausgang des FIFOs TP_r, also die Menge an Daten pro Takt, die aus dem FIFO gelesen wird, nicht nur abhängig von der Auslesefrequenz f_r, sondern auch von der ICAP Input Datenweite IIW. Der maximale Durchsatz am Ausgang des ICAP Controllers kann mit Gleichung 5.2 berechnet werden.

$$TP_r = IIW * f_r * BF \tag{5.2}$$

BF ist der Busy Factor. Er zeigt an, welchen Prozentsatz des Konfigurations-Vorgangs ICAP in der Lage ist neue Daten zu akzeptieren und damit nicht „BUSY“ ist. Der BF

wird durch Auswertung des in Abschnitt 5.3.2.3 beschriebenen *busy* Signals bestimmt, das von der ICAP Schnittstelle geliefert wird. Der BF wird in Abschnitt 5.3.2.3 näher erläutert. Grundsätzlich bleibt das FIFO gefüllt, wenn sichergestellt werden kann, dass TP_w größer als TP_r ist, also Gleichung 5.3 erfüllt ist.

$$TP_w > TP_r \tag{5.3}$$

Einsetzen von Gleichung 5.1 sowie Gleichung 5.2 in Gleichung 5.3 resultiert in Gleichung 5.4.

$$\frac{B_W[cycles] * D_W[bit]}{B_W[cycles] + L[cycles]} * f_w > IIW[bit] * f_r * BF \tag{5.4}$$

Gleichung 5.4 kann nun verwendet werden, um zu berechnen, ob das FIFO die gesamte Zeit gefüllt bleiben kann, falls die entsprechenden Parameter bekannt sind. Um zu vermeiden, dass niederpriore IP Cores die Rekonfigurations-Geschwindigkeit verlangsamen, wird der ICAP Controller im AutoVision System mit höchster Priorität an den Bus oder direkt an den MPMC angeschlossen. Dies ist vor allem bei Anwendungen sinnvoll, die maximale Rekonfigurations-Geschwindigkeit erfordern. Ein stets gefülltes FIFO ist wichtig, um garantieren zu können, dass die maximale Datenmenge (IIW Bit pro Taktzyklus) an der ICAP Schnittstelle bereit steht. Ist dies sichergestellt, kann der maximale Durchsatz an Konfigurations-Daten TP_r an der ICAP Schnittstelle mit Gleichung 5.2 berechnet werden. Ist der exakte Durchsatz TP_r (gemessen oder berechnet) und die Bitstromgröße B_S bekannt, berechnet sich die Rekonfigurations-Zeit t_R nach Gleichung 5.5.

$$t_R = \frac{B_S}{TP_r} + L \tag{5.5}$$

L ist in diesem Fall die initiale Latenz, um die ersten Bytes an Konfigurations-Daten in das FIFO zu laden. L wird vom ersten Rekonfigurations-Trigger bis zum Erscheinen der ersten Bitstromdaten an der ICAP Schnittstelle gemessen. Messungen im AutoVision System zeigen, dass L abhängig vom verwendeten Speicher (DDR oder DDR2 SDRAM) und entsprechendem Controller zwischen 30 und 50 Taktzyklen liegt. Für große partielle Bitströme von mehreren KB ist L vernachlässigbar gering.

5.3.2.2 Reduktion der Bitstromgröße (Combitgen)

Bei Betrachtung von Gleichung 5.5 ist klar ersichtlich, dass kürzere Rekonfigurations-Zeiten entweder durch Reduktion von B_S oder Erhöhung von TP erreicht werden können. In diesem Abschnitt wird daher eine Methode vorgestellt, mit der die Bitstromgröße reduziert werden kann. Diese Methode beschreibt die Reduktion von Overhead nachdem der partielle Bitstrom generiert wurde und ist im Combitgen Tool [CMS06] realisiert.
Combitgen ist ein Kommandozeilentool, das optimierte partielle Bitströme erzeugt. Es funktioniert für alle V2P und V2 Devices. Durch den schematisch sehr ähnlichen Aufbau

des Konfigurations-Layers (siehe Abschnitt 2.2.2) ist das Konzept einfach auf neuere Devices (V4, V5, V6) übertragbar. Combitgen wird nach (a posteriori) der Erstellung der initialen Bitströme eingesetzt. Als Eingabe für Combitgen dienen die so genannten Toplevel Bitströme[1], die zur initialen Programmierung des FPGAs verwendet werden. Combitgen erstellt partielle Bitströme, die nur die unterschiedlichen Konfigurations-Frames zwischen allen Toplevel Designs enthalten. Dies bedeutet im Umkehrschluß, dass all diejenigen Konfigurations-Frames, die in den Toplevel Bitströme gleich sind, nicht in die jeweiligen partiellen Bitströme übernommen werden. Damit kombiniert Combitgen die Vorteile des DBRFs (siehe Abschnitt 2.2.7.1) und des MBRFs (siehe Abschnitt 2.2.7.2), während die entsprechenden Nachteile gemindert werden. Combitgen liest die Toplevel Bitströme ein, verarbeitet diese und generiert für jeden Toplevel Bitstrom einen entsprechenden partiellen Bitstrom. Mit diesen partiellen Bitströmen ist es (ähnlich wie beim MBRF) möglich von jedem Design in jedes andere zu wechseln. Abbildung 5.8 zeigt den Ablauf bei 3 verschiedenen Toplevel Bitströme.

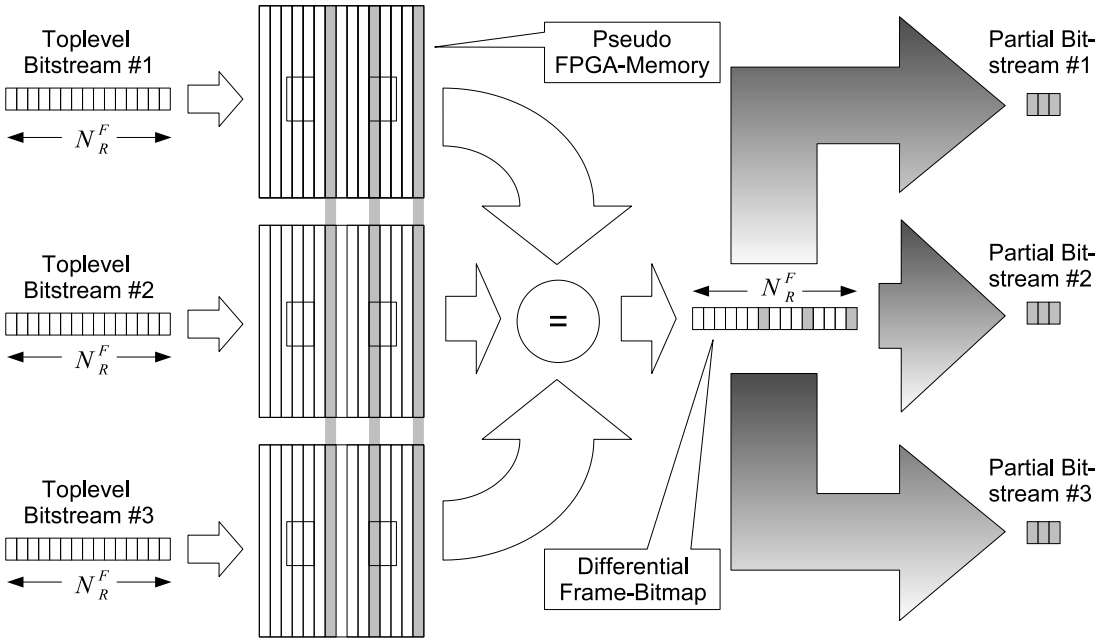


Abbildung 5.8: Combitgendatenfluß mit 3 Toplevel Bitströmen als Eingang und 3 resultierenden partiellen Bitströmen, aus denen der Overhead entfernt wurde

Wie in Abbildung 5.8 zu erkennen ist, wird nach dem Einlesen der Bitströme innerhalb des Programms ein 2-dimensionales Array, das so genannte Pseudo FPGA Memory, pro Toplevel Bitstrom angelegt. Das Pseudo FPGA Memory simuliert den Konfigurations-

[1] Toplevel Bitströme sind vollständige Bitströme, um das Device initial zu konfigurieren

Layer eines FPGAs, oder genauer, eine bestimmte Konfigurations-Row eines Devices bestehend aus N_R^F Konfigurations-Frames. Die aus den Toplevel Bitströmen extrahierten Konfigurations-Daten (ohne Paketheader) werden in die entsprechenden Pseudo FPGA Memories geschrieben. Nachdem alle Pseudo FPGA Memories gefüllt sind, werden Konfigurations-Frames mit gleicher Adresse aus allen Designs miteinander verglichen. Konfigurations-Frames mit gleicher Adresse, jedoch unterschiedlichen Inhalts, sind grau in Abbildung 5.8 hinterlegt. Wird ein Unterschied in den Konfigurations-Frames zwischen den verschiedenen Toplevel Designs festgestellt, wird der entsprechende Konfigurations-Frame in der so genannten *Differential Frame-Bitmap* markiert. Die Länge der *Differential Frame-Bitmap* ist abhängig von der Anzahl an Konfigurations-Frames per Konfigurations-Row N_R^F (siehe Tabellen A.3 bis A.5). Um Combitgen für V4, V5, V6 oder Spartan Devices zu erweitern, muß nur jede Konfigurations-Row einzeln betrachtet werden und die partiellen Bitströme demnach zusammengesetzt werden. An dem Konzept selbst ändert sich dabei nichts. Nachdem die Überprüfung auf Unterschiede abgeschlossen ist, wird die *Differential Frame-Bitmap* prozessiert. Der Konfigurations-Frame aus Toplevel 1, 2 bzw. 3, der dem Eintrag in der *Differential Frame-Bitmap* entspricht, wird aus dem entsprechenden Pseudo FPGA Memory extrahiert und in den partiellen Bitstrom 1, 2 bzw. 3 übernommen. Das bedeutet, dass die Adressen der Konfigurations-Frames in allen partiellen Bitströmen dieselben sind und sich lediglich die Konfigurations-Daten, die sich hinter dieser Adresse verbergen unterscheiden.
Im Gegensatz zu den partiellen Bitströmen, die mit dem MBRF generiert wurden, besteht ein partieller Bitstrom, der mit Combitgen generiert wurde aus einzelnen Konfigurations-Frames statt aus Konfigurations-Columns. Die Verwendung von einzelnen Frames kann mitunter zu erheblich kleineren Bitstromgrößen führen, besonders wenn große Teile der PRMs gleich sind. Zusätzlich bleibt der Vorteil bestehen, von jedem Toplevel zu jedem anderen Toplevel rekonfigurieren zu können. Durch Combitgen wird der partielle Bitstrom völlig neu zusammen gesetzt. Um die Integrität des Bitstroms zu wahren, wird daher ein CRC berechnet und am Ende in den Bitstrom übernommen. Des weiteren wird ein Bitstromheader (s. Abschnitt 2.2.4) generiert.
Um die Bitstromgrößen noch weiter zu reduzieren, wird das Multiple Frame Write Kommando (s. Abschnitt 2.2.5) verwendet. Dies kann zu einer reduzierten Bitstromgröße führen, wenn viele Konfigurations-Frames mit demselben Inhalt vorhanden sind. Im Gegensatz zu Xilinx' Bitgen Tool, das MFWR nur bei mehreren gleichen Frames verwendet, setzt Combitgen dieses Feature ein, um einzelne Konfigurations-Frames ohne Pad-Frame zu schreiben. In Abschnitt 2.2.5 wurden zwei Möglichkeiten beschrieben, mit denen Bitstromdaten aus dem FDRI in das Konfigurations-Memory geschrieben werden können. Mit Hilfe eines gepipelineten Ansatzes durch den folgenden Konfigurations-Frame, oder durch Schreiben zweier Pad Wörter an das MFWR Register. In allen Xilinx Bitströmen werden einzelne Frames mit der gepipelineten Methode geschrieben. Durch den Einsatz der zweiten Methode im Combitgen Tool, können unnötige Pad Daten vermieden und die Bitstromgröße somit weiter reduziert werden. Weitere Einzelheiten über Combitgen

sind in [CMS06] erläutert. Eine Gegenüberstellung der Größe, der mit den Xilinx Tools und der mit Combitgen erzeugten partiellen Bitströmen ist in Tabelle 5.1 aufgeführt.

Modul	MBRF ohne MFWR [byte]	MBRF mit MFWR [byte]	Combitgen [byte]	o. header m. padding [byte]	m. Kompression [byte]
CTE	560.513	403.902	397.454	397.440	66.938
SE	560.513	403.863	397.454	397.440	230.857
TE	560.513	403.898	397.454	397.440	176.099
BM	560.513	310.711	290.634	290.560	32.356

Tabelle 5.1: Ergebnisse von Combitgen auf der V2P Plattform

Mit Combitgen erzeugte partielle Bitströme können von FPGA Programmiertools, wie Xilinx' Impact genutzt werden. Wie in Abschnitt 2.2.4 dargestellt, enthält ein partieller Bitstrom jedoch noch weitere Daten (Header), die für die eigentliche Konfiguration nicht von Bedeutung sind und daher entfernt werden können. An dieser Stelle wird davon ausgegangen, dass nur partielle Bitströme verwendet werden, die auch für das entsprechende Device erzeugt wurden. Während der Initialisierungsphase des AutoVision Systems, wird beim Kopieren der partiellen Bitstomdaten aus einem nicht flüchtigen Speicher in den Hauptspeicher der Header entfernt und die verbleibenden Bitstromdaten auf ein ganzzahliges Vielfaches eines Bursts der Weite B_W gepadded. Dies dient dazu sicherzustellen, dass nach der erfolgreichen Rekonfiguration, der *CPP* innerhalb des ICAP nicht erneut synchronisiert wird, wenn der partielle Bitstrom nur mit Hilfe von vollen Bursts zum ICAP übertragen wird. In der fünften Spalte von Tabelle 5.1 sind daher die Größen, der mit Combitgen erzeugten Bitströme ohne Header, jedoch mit Pad Wörtern dargestellt. All diese Maßnahmen tragen zur Reduktion der Bitstromgröße und nach Gleichung 5.5 deshalb auch zur Reduktion der Konfigurations-Zeit bei. Bitstromkompression hingegen hat bei ständig gefülltem FIFO keine Auswirkung auf die Rekonfigurations-Zeit, da die komprimierten Bitstromdaten wieder dekomprimiert werden müssen, bevor sie der ICAP Schnittstelle zugeführt werden können. Die Kompression von Bitströmen hat jedoch eine positive Auswirkung auf den Speicherplatz der partiellen Bitströme und die Last auf dem Verbindungsmedium (Bus, NoC, etc.) zwischen Hauptspeicher und ICAP. In der letzten Spalte von Tabelle 5.1 sind die Größen der komprimierten partiellen Bitströme, die mit Combitgen generiert wurden dargestellt. Die Kompression erfolgte mit den in [HUWJ04] vorgestellten Ansätzen und Tools.

5.3.2.3 Durchsatzmaximierung von Konfigurations-Daten

Um den maximalen Durchsatz an Rekonfigurations-Daten an der ICAP Schnittstelle zu erreichen, sind nach Formel 5.2 drei Parameter zu berücksichtigen. IIW, BF und

f_r. Da die ersten beiden Parameter devicespezifisch und somit vorgegeben sind, bleibt nur die maximal spezifizierte Frequenz für f_r zu wählen. Laut [Xil10c] ist die maximale ICAP Frequenz für V4 und V5 jeweils 100 MHz. Ohne Verwendung des in Abschnitt 5.3.2 erwähnten *BUSY* Signals liegt die maximale Frequenz in V2 Architekturen, mit der ICAP betrieben werden kann, bei 50 MHz. Zusätzlich ist es notwendig Sorge dafür zu tragen, dass das FIFO im ICAP Controller gefüllt bleibt. Daher muß insbesondere Gleichung 5.4 erfüllt sein. Dies wird durch Bursting (s. Abschnitt 2.3.2.1) und Address Pipelining (s. Abschnitt 2.3.2.2) erreicht.
Um den Durchsatz nach Gleichung 5.2 zu erhöhen, besteht die Möglichkeit ICAP zu übertakten. Da Änderungen der Systemfrequenz meist mit sehr großem Aufwand verbunden sind, wird nur f_r erhöht. Für den Übergang zwischen verschiedenen Taktdomänen werden asynchrone FIFOs verwendet (s. Abschnitt 2.3.3).
Um ICAP außerhalb des spezifizierten Bereichs zu betreiben, ist es möglich auf V2P ein simples Handshaking Protokoll zu verwenden. Falls ICAP nicht bereit ist neue Daten anzunehmen, wird dies über den *BUSY* Ausgang des ICAP angezeigt. Um also jeden Takt, in dem ICAP nicht *BUSY* ist, Daten an die Schnittstelle zu liefern, wird das *BUSY* Signal als Handshaking-Signal verwendet. Der Busy Factor BF in Gleichung 5.2 zeigt an, wie oft ICAP nicht in der Lage ist neue Daten zu akzeptieren, also „busy“ ist. Jedoch kann das *BUSY*-Signal für den Konfigurations-Vorgang nur auf V2 und V2P verwendet werden. Auf V4 und V5 ist dieses Signal nur beim Readback der Konfigurations-Daten aktiv, so dass es beim Konfigurations-Vorgang nicht als Handshaking-Signal verwendet werden kann (s. [Xil07] und [Xil09d]). Somit ist es theoretisch möglich ICAP auf V2P durch Auswertung dieses Signals zu übertakten, um damit den Durchsatz an Konfigurations-Daten zu erhöhen.

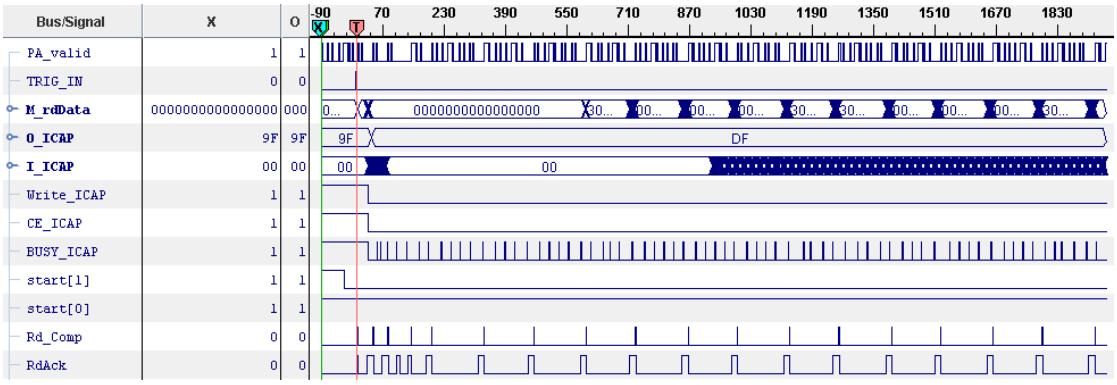


Abbildung 5.9: *Busy* signal während des Konfigurations-Vorgangs auf V2P bei 100 MHz (Screenshot aus Chipscope)

In Abbildung 5.9 ist eine Aufnahme des Konfigurations-Vorgangs auf V2P dargestellt. Klar ersichtlich ist die Häufigkeit des *Busy* Signals (BUSY_ICAP). Ist ICAP beispiels-

weise bei 10% des Konfigurations-Vorgangs nicht in der Lage neue Konfigurations-Daten anzunehmen (also „busy“), liefert dies einen BF von 0,9. Der BF zeigt also prozentual an, wie oft ICAP während des Konfigurations-Vorgangs in der Lage ist neue Daten zu akzeptieren.
Ab einer bestimmten Frequenz, die aufgrund von Prozessschwankungen zwischen verschiedenen Devices derselben Familie unterschiedlich sein kann, kann auch dem *Busy* Signal nicht mehr vertraut werden. ICAP außerhalb der spezifizierten Bereiche zu verwenden, erfordert also eine Möglichkeit der Überprüfung auf Korrektheit. Diese Verifikation des Konfigurations-Prozesses sollte zur Laufzeit stattfinden. Um eine Aussage über die Korrektheit (auch bei Verwendung des *Busy* Signals) zu treffen, muß sichergestellt sein, dass die Konfigurations-Daten sowohl richtig an der ICAP Schnittstelle ankommen, als auch fehlerfrei ins Konfigurations-Memory geschrieben werden.
Das Übertakten des ICAP Controllers jenseits der spezifizierten Frequenzen stellt völlig neue Anforderungen an die AutoVision Architektur. Neben der Funktionalität der rekonfigurierbaren Module muß nun zusätzlich der Konfigurations-Prozess zur Laufzeit verifiziert werden. Beim Übertakten können mehrere Fehlerquellen auftreten. Entweder sind die Daten, die am ICAP Controller ankommen bereits korrupt oder der *Configuration Packet Prozessor (CPP)*, der hinter der ICAP Schnittstelle sitzt, schreibt die Daten nicht korrekt in das Konfigurations-Memory.

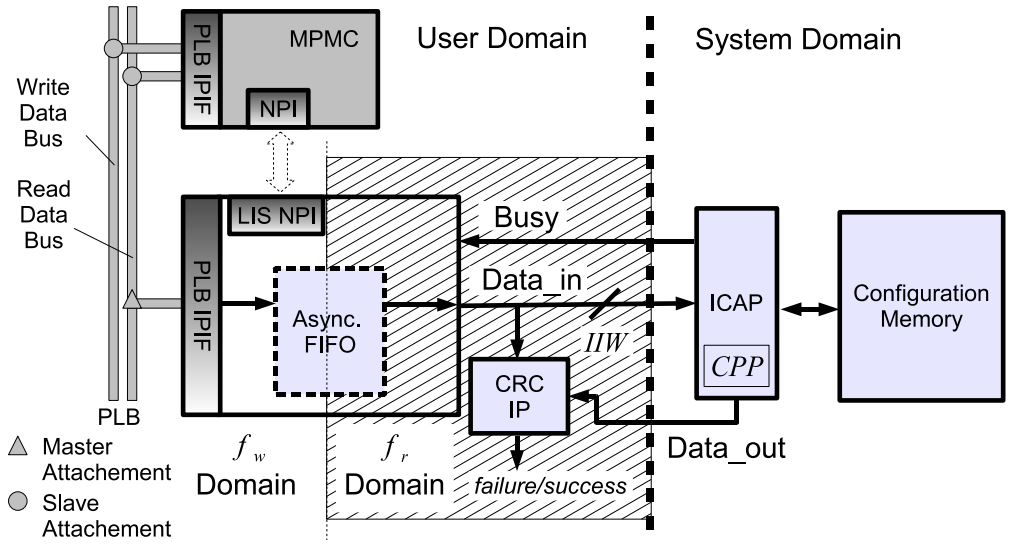


Abbildung 5.10: Systemdomänen in einem dynamisch rekonfigurierbaren System

Um die Laufzeitverifikation zu realisieren, wird das FPGA in die User-Domäne und die System-Domäne unterteilt. Die User-Domäne beinhaltet alle konfigurierbaren Ressourcen (Speicher, Bussystem, ICAP controller etc.) und endet direkt vor der ICAP Schnittstelle,

wie in Abbildung 5.10 zu sehen ist. Die System-Domäne beinhaltet den ICAP und damit den *CPP*, auf dessen Register nicht zugegriffen werden kann und der für das Schreiben der Daten in den Konfigurations-Layer zuständig ist. Während der Laufzeit müssen beide Domänen auf Fehler untersucht werden. Um die Integrität der Bitstromdaten in der User-Domäne zu überprüfen bevor sie der ICAP Schnittstelle zugeführt werden, wird ein eigener IP Core eingesetzt. Der IP Core besteht aus einem Cyclic Redundancy Check (CRC) IP Modul, welches direkt vor die ICAP Schnittstelle geschaltet wird. Dieses Modul analysiert die ankommenden Bitstromdaten und berechnet zur Laufzeit einen CRC-16 Wert. Die Berechnung des CRC-16 Wertes, die auch bei Combitgen (s. Abschnitt 5.3.2.2) eingesetzt wird, ist in [Xil08b] beschrieben.
Die Verifikation in der System-Domäne stellt sicher, dass die Daten an die entsprechende Stelle im Konfigurations-Memory geschrieben wurden. Durch die Zusammenführung der Verifikationsinformationen aus beiden Domänen kann während der Laufzeit eine Aussage über Erfolg (success) oder Scheitern (failure) des Rekonfigurations-Prozesses gemacht werden. Falls korrupte Bitstromdaten beim ICAP Controller ankommen, selbst wenn nur ein einzelnes Bit vertauscht ist, wird dies erkannt und der Konfigurations-Prozess kann neu gestartet werden. Das Data_out Signal in Abbildung 5.10, das einen Konfigurations-Fehler in der System-Domäne anzeigt, ist eine logische ODER-Verknüpfung mehrerer interner, nicht dokumentierter Fehlersignale. Eines dieser Fehlersignale basiert auf einem erneuten CRC Check in der System Domäne. Falls beim Schreiben ins Konfigurations-Memory ein Fehler auftritt, beispielsweise durch Übertakten der ICAP Schnittstelle, wird dies erkannt. Um dem User Informationen aus der System-Domäne über Erfolg oder Scheitern eines Konfigurations-Prozeses mitzuteilen, werden während der Konfigurations-Phase am Ausgang der ICAP Schnittstelle bestimmte Daten über den gegenwärtigen Zustand generiert. Abhängig von den Daten am ICAP Output kann hier überwacht werden, ob ICAP richtig synchronisiert (0xDF) oder desynchronisiert (0x9F) wurde und ob ein Konfigurations-Fehler (0x1F) aufgetreten ist [Xil10c]. Die Information aus beiden Domänen wird dann im CRC IP Core zusammengeführt, um zur Laufzeit den Konfigurations-Vorgang zu verifizieren. Dadurch, dass diese Information zur Laufzeit, also parallel zum Konfigurations-Prozess selbst durchgeführt wird, entstehen beim Rekonfigurations-Durchsatz und der damit verbundenen Rekonfigurations-Zeit keine zusätzlichen Latenzen. Sollte ein Fehler detektiert werden, kann der Rekonfigurations-Prozess beispielsweise erneut gestartet werden. Auf einem V5 bei einer Frequenz von $f_r = 200$ MHz trat bei über 14 Millionen getesteten Rekonfigurationen kein einziger Fehler auf. Weitere Details über die Laufzeitverifikation sind in [CAAS10] und [Ahm09] zu finden.

5.3.3 Verlustleistung während der Rekonfiguration

In diesem Abschnitt wird der Overhead an Verlustleistung vorgestellt, der durch DPR dem System hinzugefügt wird. Die Gesamtverlustleistung in einem FPGA setzt sich laut Abschnitt 2.3.4 aus zwei Teilen zusammen: der statischen und der dynamischen Ver-

lustleistung. Zunächst wird die statische Verlustleistung betrachtet. Zum besseren Verständnis wird die gesamte statische Verlustleistung vereinfachend unterteilt in einen Part P_{device}, der vom Device selbst, und einen weiteren Teil P_{config}, der von der Konfiguration des FPGAs abhängt. P_{device} ist die statische Verlustleisung des FPGAs, die von der Spannungsversorgung der Konfigurations-SRAM-Zellen abhängt. Ihre Höhe ist abhängig von der Architektur und verwendeten Technologie. P_{config} ist die statische Verlustleistung, die von der Konfiguration des FPGAs abhängt. P_{config} ist unabhängig davon, ob in einer SRAM Speicherzelle eine 1 oder eine 0 gespeichert ist. Jedoch beeinflusst die Konfiguration einer SRAM Zelle wiederum die Beschaltung von Transistoren im funktionalen Layer und damit die statische Verlustleistung. Einige Select Eingänge von Multiplexern sind direkt an Konfigurations-SRAMs angeschlossen, was auch in [SGVT05] erwähnt und dargestellt wird. In [TL03] ist zusätzlich ein Beispiel dargestellt, das die Datenabhängigkeit der statischen Verlustleistung von der Beschaltung der Eingangspins eines Multiplexers zeigt. Die statische Verlustleistung ändert sich also je nach Beschaltung der im funktionalen Layer verwendeten Transistoren. Daraus lässt sich schlussfolgern, dass der DPR Overhead, der im Vergleich zu einem nicht rekonfigurierbaren Design hinzukommt und die sich gegenseitig ausschließenden HWAs, die sich im rekonfigurierbaren System die Ressourcen teilen, die statische Verlustleistung beeinflussen. Da die Konfiguration die statische Verlustleistung nachweislich beeinflusst [TL03, SGVT05], stellt sich die Frage, ob DPR genutzt werden kann, um statische Verlustleistung zumindest kurzfristig einzusparen. Durch das konfigurations-Frame-weise Ändern der SRAM Inhalte, ändert sich auch der Wert der statischen Verlustleistung jedes mal, wenn ein Konfigurations-Frame in den Konfigurations-Layer geschrieben wird. Der DPR Overhead verändert die statische Verlustleistung dadurch, dass zusätzliche SRAM Zellen konfiguriert werden müssen.
Die dynamische Verlustleistung hingegen ist laut Gleichung 2.10 abhängig von der Frequenz. Wird beispielsweise der ICAP Controller übertaktet, steigt auch die dynamische Verlustleistung proportional mit der Frequenz. Je höher die Frequenz mit dem der ICAP Controller versorgt wird, desto schneller die Rekonfiguration und desto höher auch die dynamische Verlustleistung. In [TKR$^+$06] werden die Hauptverursacher und ihr Anteil an statischer und dynamischer Verlustleistung aufgeführt.
Die statische Verlustleistung wird nach [TKR$^+$06] vor allem durch die Konfigurations-SRAM-Zellen und die Konfiguration der Routing Switches (Verdrahtung) bestimmt. Die dynamische Verlustleistung hängt vor allem von der Verdrahtung ab. Vergleichbare Ergebnisse für die dynamische Verlustleitung in Virtex-II FPGAs werden in [SKB02] genannt.

5.3.3.1 Konzepte zur qualitativen Einsparung von Verlustleistung

Gegenüber ASICs und ASSPs haben FPGAs bezüglich Power Management noch deutliche Nachteile. Methoden wie, dynamisches power gating, dynamisches voltage oder frequency scaling, und threshold-voltage Manipulation existieren selbst in den neues-

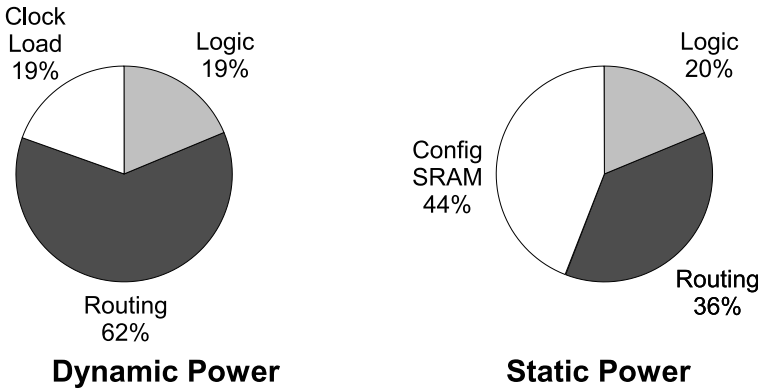


Abbildung 5.11: Verteilung der statischen und dynamischen Verlustleistung [TKR+06] auf Spartan3 FPGAs

ten Xilinx Devices nicht [Wil09]. Jedoch wird eine dieser Techniken, das Kontrollieren der Schwellwertspannung von Multi-threshold Transistoren, auf Altera Devices unterstützt [Alt07]. Auf FPGAs ist es jedoch möglich den Takt für ungenutzte Teile über das so genannte Clock Gating (CG) abzuschalten [ZRM06]. Clock Gating wird auf Register Transfer Ebene implementiert und trennt gezielt Teile des Taktbaums, welche für die Versorgung von unbenutzten sequentiellen Elementen wie BRAMs oder Flip-Flops verwendet werden. Während der Idle Zeit eines IP Cores reduziert sich die dynamische Verlustleistung durch CG um den Anteil, den das nicht benutzte Modul zum Gesamtverbrauch beisteuert. CG hat jedoch keinen Einfluss auf die statische Verlustleistung. Zhang et. al. präsentieren in [ZRM06] einen Ansatz, um CG auf Registerebene zu realisieren. Oft ist diese feingranulare Lösung jedoch nicht notwendig, da es gewünscht ist ganze IP Cores während ihrer Idle Zeit abzuschalten.

In Abbildung 5.12 ist die Realisierung von CG in einem Multitaktratensystem anhand eines Blockdiagramms dargestellt. Der Übersichtlichkeit wegen sind nur die wichtigsten Takte im System aufgeführt. Es fehlen beispielsweise die Takte für den Hauptspeicher und den Systemtakt. Das realisierte CG Konzept in der AutoVision Architektur funktioniert analog. In Virtex-5 FPGAs und späteren Generationen läßt sich CG von einzelnen Modulen relativ einfach implementieren. Bedingung hierfür ist, dass im clock generator wrapper ein eigener Takt für die abschaltbaren Module definiert wird. Der clock generator wrapper dient zur Konfiguration der Phase locked loops (PLLs). In synchronen FPGA Schaltungen muß der Takt gleichmäßig an alle getakteten Bauelemente (Register und BRAMs) verteilt werden. In Xilinx FPGAs wird für die Taktverteilung über den gesamten Chip ein spezieller Takttreiber verwendet, ein BUFG. Dieser Takttreiber wird durch ein Generic im clock generator instantiiert. Um CG für ein spezielles Modul zu

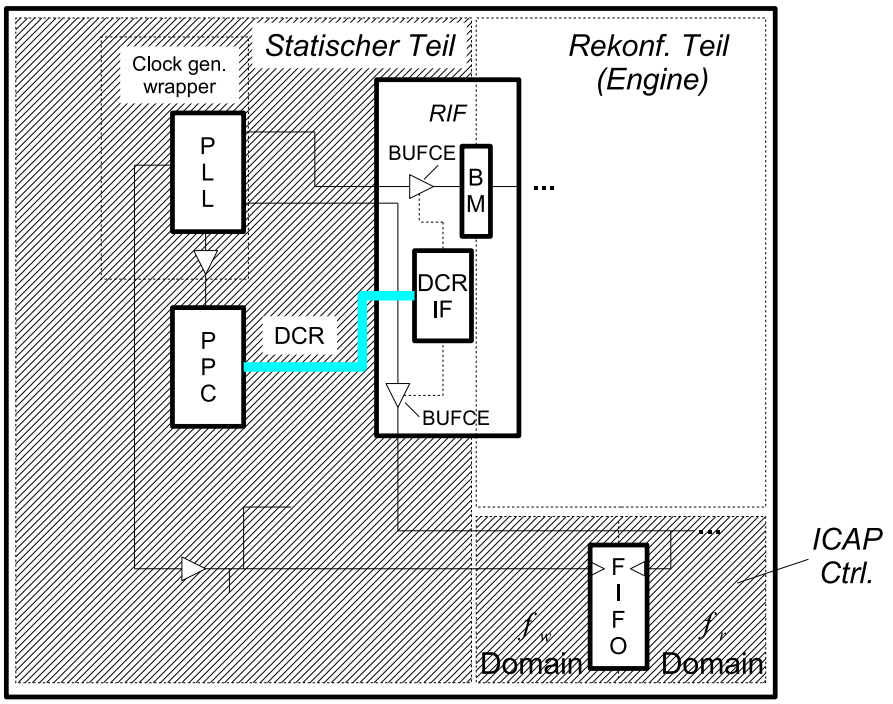


Abbildung 5.12: Realisierung von Clock Gating in einem Multitaktratensystem

realisieren, wird der BUFG für den entsprechenden Takt im clock generator jedoch nicht instantiiert. Stattdessen wird ein BUFCE, ein Takttreiber mit Enableeingang, verwendet. Instantiiert wird der BUFCE außerhalb des abzuschaltenden Moduls. Dies hat zur Folge, dass bei der Platzierung genau dasselbe Xilinx Primitive (ein BUFGCTRL) wie für den BUFG verwendet wird, allerdings mit Enable-Signal.
Liu et. al. vergleichen in [LPF09a] den Einsatz von CG mit dem Überschreiben des PRM mit einem Blank Bitstrom[2]. Dabei wird ein PRM während seiner IDLE Zeit vom Takt durch CG getrennt. Das Abschalten des Taktes über den BUFGCTRL verursacht keine weitere Verlustleistung. Wenn das PRM durch einen Blank Bitstrom komplett entfernt wird, verschwindet der komplette Ast des Taktbaums. Damit sollte die dynamische Verlustleistung im selben Maße sinken wie bei CG. Während des Rekonfigurations-Prozesses steigt die dynamische Verlustleistung des Cores allerdings durch Umladen der Inhalte der SRAM Zellen. Zusätzlich steigt dabei die dynamische Verlustleistung der I/Os, da ein Bitstrom aus einem externen Speicher geladen werden muß. Das bedeutet, dass sich DPR

[2]Ein Blank Bitstrom ist ein partieller Bitstrom, um ein PRM aus einer PRR heraus zu löschen. Anschlusspunkte in Switch-Matrizen von Netzen des statischen Teils, die die PRR überspannen sind auch im Blank Bitstrom enthalten.

gegenüber CG nur lohnt, wenn das abzuschaltende Modul längere Zeit inaktiv ist. Jedoch hat die DPR im Gegensatz zu CG auch einen positiven Einfluss auf die statische Verlustleistung. In [LPF09a] wird beschrieben, ab welcher Zeitspanne die Abschaltung eines Moduls über DPR (blanken der PRR mit Nullen) anstatt über CG sinnvoll ist. Allerdings wird dort nicht die steigende Verlustleistung der I/Os berücksichtigt. Anstatt jedoch wie in [LPF09a] beschrieben, entweder CG oder DPR zur Einsparung von Verlustleistung einzusetzen, lässt die maximale Optimierung erzielen, wenn in einem System sowohl DPR als auch CG eingesetzt wird.
Da es in der Literatur jedoch keine Angaben gibt, wie ein FPGA für eine möglichst geringe statische Verlustleistung zu konfigurieren ist, und da eine Analyse und Auswertung aller möglichen Konfigurationen in sinnvoller Zeit nicht möglich ist, wird versucht den Anteil P_{config} empirisch durch Überschreiben der PRR mit einem Blank Bitstrom zu reduzieren. Durch eine Konfiguration der PRR mit Nullen (exklusive der Anschlusspunkte der die PRR überspannenden Netze des statischen Teils) wird eine Reduktion von P_{config} erwartet.
Alle in der Literatur beschriebenen Ansätze (mit Ausnahme von [LPF09a]) versuchen nachdem der FPGA einmal initial konfiguriert wurde eine Einsparung von statischer Verlustleistung durch Modifikationen im funktionalen Layer zu erreichen. In [ANT04] und [SGVT05] wurde in Simulationen versucht, die Multiplexerein- und ausgänge in Switch Matrizen mit 0 zu beschalten, um auf diese Art statische Verlustleistung einzusparen. Die Ergebnisse in [TL03], [ANT04] und [SGVT05] zeigen, dass Nullen an den Eingängen und am Select-Port eines Multiplexers nicht notwendigerweise zum geringsten Wert von statischer Verlustleistung führt. Allerdings wird dort jeweils auch nur der subthreshold Leckstrom I_{sub} und der Gate (Oxide) Leakage Strom I_g betrachtet. Der Band-to-Band Tunneling Strom I_{BTBT} dagegen nicht.

5.3.3.2 Dynamisches Power Management in DPR Systemen

Wie in Abschnitt 5.1 beschrieben, wird DPR bei sich gegenseitig ausschließenden Fahrsituationen eingesetzt. Im AutoVision System existieren bei der Inter- sowie IntraVFR neben den PRMs noch weitere IP Cores, die sich in ihrer Verarbeitung gegenseitig abwechseln. So verarbeitet ein HWA entweder Bilddaten oder er wird rekonfiguriert. Der ICAP Controller hingegen wird während der Rekonfiguration benötigt, nicht aber wenn Bilddaten verarbeitet werden. Dies gilt insbesondere, wenn nur eine PRR im System existiert. Bei mehreren PRRs kann der ICAP Controller selbstverständlich einen neues PRM laden, während ein Weiteres in einer anderen PRR gerade beschäftigt ist. Die These Verlustleistung durch die Kombination von CG und DPR einsparen zu können, stützt sich auf die Tatsache, dass es IP cores gibt, die einen relativ hohen Anteil an der dynamischen Verlustleistung tragen und nicht durch einen Blank Bitstrom entfernt werden können (ICAP Controller, NPI). Diese IP Cores (DPR Overhead) sind jedoch einen Großteil der Zeit inaktiv und lassen sich während ihrer inaktiven Phase über CG abschal-

ten. In Abbildung 5.13 ist ein möglicher Ablauf eines dynamischen Power Managements während eines Videoframes dargestellt.

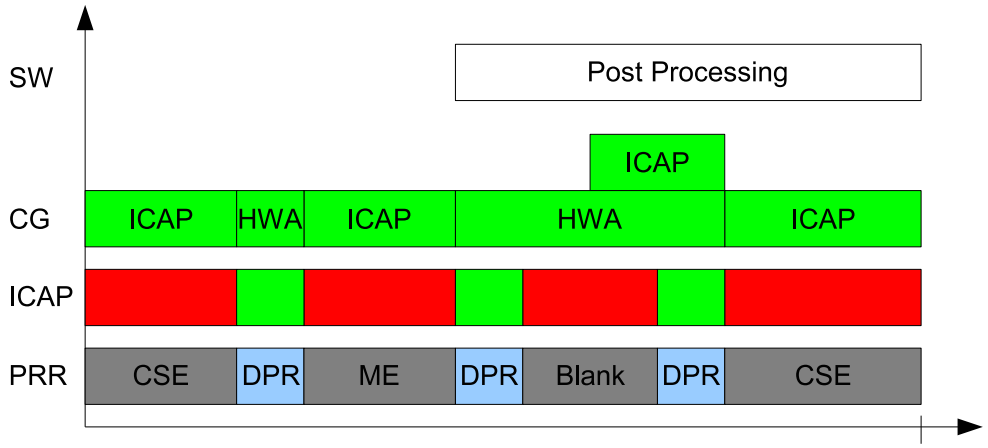


Abbildung 5.13: Aktivitätsdiagramm von DPR und CG während eines Videoframes

In Abbildung 5.13 sind die Zeiten in denen der ICAP Controller inaktiv und aktiv ist, rot bzw. grün dargestellt. Wenn, wie in Abbildung 5.13 ein HWA (*CSE*) mit Bildverarbeitung beschäftigt ist, ist der ICAP Controller inaktiv und kann während dieser Zeit über CG abgeschaltet werden. Wenn über den ICAP Controller rekonfiguriert wird, ist der HWA inaktiv und kann während dieser Zeit (auch während der Rekonfiguration) via CG abgeschaltet werden. Bei der IntraVFR wiederholt sich dieses Vorgehen. Ist ein Blank Modul geladen, können sowohl die leere PRR als auch der ICAP Controller abgeschaltet werden. Dieses Konzept ist auf eine beliebige Anzahl an HWAs übertragbar und lässt sich durch den Einsatz von insgesamt drei BUFCEs realisieren, wie in Abbildung 5.12 dargestellt ist. Eine Instanz für den HWA, eine Instanz für den leseseitigen Teil und eine Instanz für den schreibseitigen Teil des ICAP Controllers. Da es sich im AutoVision System um ein zentral verwaltetes System handelt, übernimmt der PPC die Taktsteuerung. Sobald ein Interrupt von einem HWA abgesetzt wurde, kann der ICAP Controller über CG eingeschaltet und im selben Moment der HWA vom Takt getrennt werden. Wird anschließend ein Interrupt vom ICAP Controller gesendet, um zu signalisieren, dass die Rekonfiguration beendet ist, kann der HWA wieder angeschaltet und der ICAP Controller abgeschaltet werden. Die Logik zum Abschalten des Taktes sollte außerhalb der abzuschaltenden Module platziert werden. Im AutoVision System wurde das *RIF* gewählt, das ohnehin bereits zum Abschalten der ausgehenden Signale der PRR verwendet wird. Im Wrapper des *RIFs* werden somit auch die BUFCEs instantiiert. Über das DCR IF innerhalb des *RIF* können selektiv alle BUFCEs ein- und ausgeschaltet werden. Durch die Entscheidung das Taktmanagement in das *RIF* zu übernehmen, können nun mit ein

und demselben DCR Kommando gleichzeitig der Disconnector deaktiviert, CG für der HWA ein- und für den ICAP Controller abgeschaltet werden. Das DCR IF steuert daher die Enable Leitungen der BUFCEs (gestrichelte Verbindungen in Abbildung 5.12).
Ergebnisse hinsichtlich Ressourcenverbrauch, Performanz und Verlustleistung des rekonfigurierbaren Systems sind in Kapitel 6 aufgeführt.

5.4 Rekonfigurations Scheduling

Um garantieren zu können, dass die Rekonfiguration nicht zu Qualitätseinbußen oder gar zum Verlust ganzer Video-Frames führt, findet die DPR im AutoVision System an zuvor definierten Zeitpunkten statt. Dies ist notwendig, um die Systemintegrität zu gewährleisten. Die HWAs sollen während ihrer Verarbeitung nicht unterbrochen werden. Anstatt bei jeder Anfrage, das Device direkt im Anschluss zu rekonfigurieren, werden die Rekonfigurations-Anfragen gesammelt und das Device wird nach der Prozessierung eines HWAs rekonfiguriert. Um einen reibungslosen Ablauf zwischen Bildverarbeitung und Rekonfiguration zu gewährleisten, muß eine Ablaufplanung, ein so genanntes Rekonfigurations-Scheduling, festgelegt werden. Das Rekonfigurations-Scheduling legt den Zeitpunkt und den Ablauf einer Rekonfiguration fest.

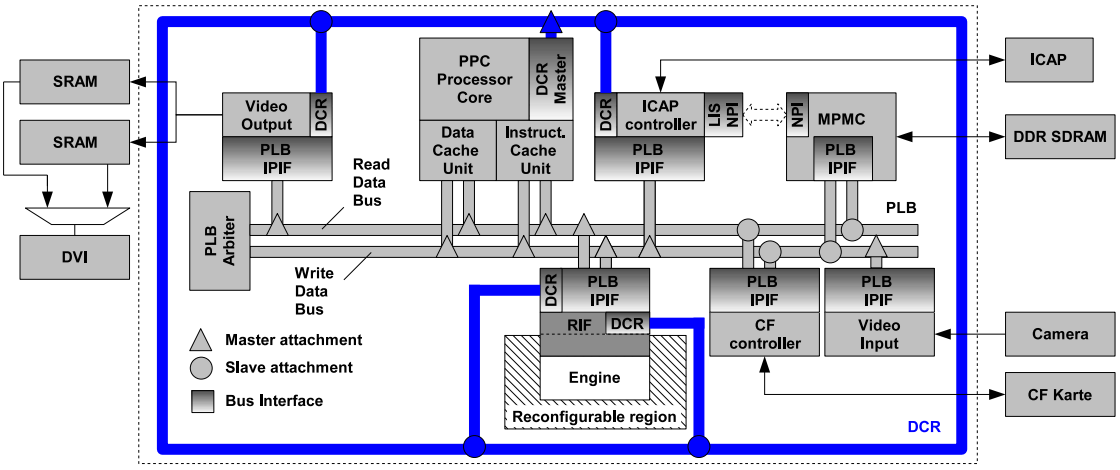


Abbildung 5.14: Blockdiagramm der AutoVision Architektur auf dem Virtex-II Pro

In Abbildung 5.14 sind die wichtigsten Komponenten der rekonfigurierbaren AutoVision Architektur dargestellt. Im Folgenden wird nun eine Übersicht über den Datenfluß im rekonfigurierbaren System gegeben, um anschließend festzulegen wann die Rekonfiguration stattfindet.
Bevor mit der eigentlichen Bildverarbeitung und Rekonfiguration begonnen werden kann, werden während der Initialisierungsphase die partiellen Bitströme aus einem Flash Spei-

cher in den Hauptspeicher (hier DDR SDRAM) geschrieben. Wie bereits in Abschnitt 4.2 erwähnt, werden die Bilddaten vor ihrer Verarbeitung zunächst im Hauptspeicher abgelegt, damit mehrere Instanzen (HWAs , CPUs etc) auf die Orginalbilddaten zugreifen können. DPR liefert einen weiteren Grund für das temporäre Speichern von Bilddaten. Um den Verlust von Bilddaten während der Rekonfiguration zu vermeiden, müssen die Bilddaten zwischengepuffert werden. Erfolgt während der Verarbeitung durch einen HWA eine Rekonfigurationsanforderung, wird dieser Request zurückgestellt, bis der HWA die Verarbeitung abgeschlossen hat. Anschließend wird die Rekonfigurations-Anforderung verarbeitet.
Nachdem dieses Kommando abgesetzt ist, wird die DPR durchgeführt. Bei einem Direktanschluss des ICAP Controllers an den MPMC (gestrichelte Verbindung in Abbildung 5.14) wird das zentrale Interconnect, der PLB Bus, nicht zusätzlich mit Rekonfigurations-Daten belastet. Den Abschluß des Rekonfigurations-Prozesses meldet der ICAP Controller über einen Interrupt an den PPC.
Nach der Festlegung des Rekonfigurations-Zeitpunktes wird nun der Ablauf einer Rekonfiguration erläutert. Der Ablauf eines einzelnen Rekonfigurationsvorgangs bei der InterVFR bzw. der IntraVFR (s. Abschnitt 5.3.2) ist gleich. Nur die Anzahl an Rekonfigurationen unterscheiden sich. Eine Ablaufplanung der Rekonfigurationssteuerung bei der IntraVFR des OF ist in Abbildung 5.15 dargestellt. Die InterVFR Rekonfiguration entspricht der zweiten Rekonfiguration in Abbildung 5.15.
Bei der InterVFR wird die Higher-level Bildverarbeitung auf dem PPC direkt nach dem Absetzen des DCR Kommandos gestartet, um den Disconnector abzuschalten und den ICAP Controller zu initialisieren gestartet. Mit dem Abschalten des Disconnectors wird gleichzeitig der Taktbaum für den HWA abgeschaltet und für den ICAP Controller aktiviert. Der eigentliche Rekonfigurations-Prozess läuft dann parallel zur Verarbeitung in SW. Im Vergleich zu einem nicht rekonfigurierbaren System verlängert sich somit die Verarbeitungszeit um die beiden abzusetzenden DCR Signale und ist mit etwa 20 Prozessorzyklen vernachlässigbar gering.
Verglichen mit einem nicht rekonfigurierbaren OF Design, bei dem der PPC nur mit der Bearbeitung der Featureliste beschäftigt ist, ist dieser in einem rekonfigurierbaren System minimal stärker ausgelastet. Neben der Featureverarbeitung muß sich der PPC nun um weitere Aufgaben kümmern. Der PPC wird immer dann durch einen Interrupt benachrichtigt, wenn ein IP Core seine Aufgabe beendet hat. Darüber hinaus ist der PPC für die Rekonfiguration von Census- und MatchingEngine zuständig, was die Initialisierung des ICAP Controllers und das An- bzw. Abschalten des *RIF* beinhaltet.
Falls ein Prozessor mit den zusätzlichen Aufgaben überlastet ist, lässt sich die AutoVision Architektur auf ein Multiprozessor-System erweitern, wie in [CSH07] und [Che09] beschrieben. Hierbei übernimmt ein weiterer Prozessor das Rekonfigurationsmanagement. Dieser kann das Interrupthandling und die Konfiguration des ICAP übernehmen. Zusätzlich könnte dieser die Bitstromdaten analysieren, um die Platzierung einzelner CLBs und BRAMs zu visualisieren wie in [HBB+07], [Bra06] und [Lan10] beschrieben.

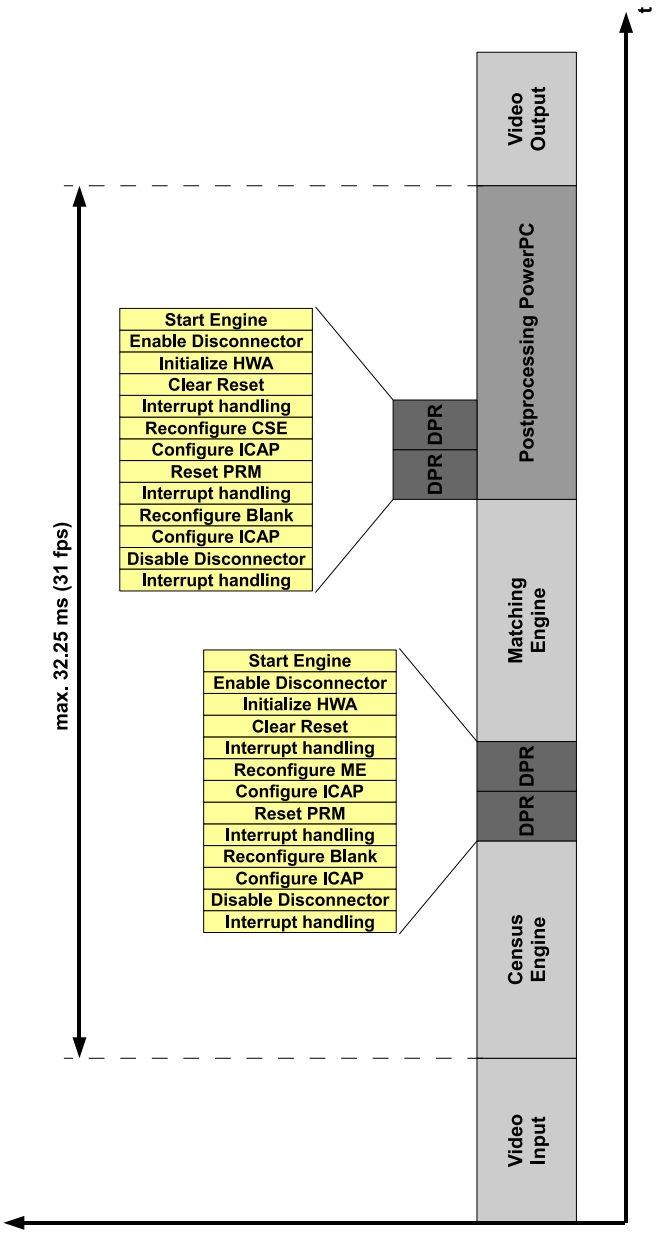


Abbildung 5.15: Verschiedene Schritte beim Ablauf einer DPR im Optischen Fluß System.

Ein Blank Bitstrom kann verwendet werden, um die vorherige Konfiguration zu löschen.

In Abbildung 5.15 ist das OF Design mit vier Rekonfigurationen während eines Video-Frames (IntraVFR) dargestellt. Nach der Verarbeitung der Eingangsbilddaten sendet die *CSE* einen Interrupt an den PPC. Dieser Interrupt wird verarbeitet und die ausgehenden Verbindungen im *RIF* werden geöffnet. Der PPC sendet daher über den DCR Bus ein Kommando, um das *RIF* abzuschalten. Während der Rekonfiguration wird über das DCR Interface im *RIF* ein Resetsignal an das PRM übertragen. Dies stellt sicher, dass nach der Rekonfiguration alle Register in einem definierten Zustand sind. Direkt nach Absetzen des Kommandos zum Abschalten des *RIF*, wird der ICAP Contoller initialisiert. Hierzu werden vom PPC, wiederum über ein DCR Kommando, die Startadresse des zu ladenden Blank Bitstroms und die Anzahl der dazu notwendigen Bursts in einem 32-bit Wort übertragen. Sobald der ICAP Controller diese Daten erhalten hat, beginnt er selbstständig über DMA Transfers die Bitstromdaten aus dem Hauptspeicher zu holen und lokal im FIFO zu puffern. Nach der Rekonfiguration ist das PRM aus der PRR gelöscht und der ICAP Controller sendet einen IRQ über den Interrupt Controller an den PPC. Nach der Verarbeitung dieses Interrupts (etwa 300 Takte) wird sofort die anschließende Rekonfiguration initiiert. Abermals wird über den DCR ein Kommando an den ICAP Controller gesendet, dass die Startadresse des partiellen Bitstroms für die *ME*, sowie die dafür notwendige Anzahl an zu übertragenden Bursts beinhaltet. Nachdem die *ME* auf das Device konfiguriert wurde, wird der PPC über einen Interrupt benachrichtigt. Nun kann der Reset für das PRM zurückgenommen und die Initialisierung der Konfigurationsregister in der *ME* gestartet werden. Zuvor wird der Takt über CG für den HWA wieder aktiviert und der des ICAP Controllers deaktiviert, um dynamische Verlustleistung einzusparen. Nach der vollständigen Initialisierung des PRMs über den DCR kann der Disconnector wieder angeschaltet werden. Letztendlich wird das PRM über einen erneuten DCR Befehl gestartet, sobald ein neuer Frame im Hauptspeicher liegt.

5.5 Ein optimierter modulbasierter Designflow für DPR Systeme

In Abbildung 5.16 ist der Entwurfsablauf für ein rekonfigurierbares FPGA Design dargestellt. Die einzelnen IP Cores, die meist in einer HW Beschreibungssprache (HDL) wie Verilog oder VHDL vorliegen, werden im Xilinx Platform Studio (XPS) zu einem eingebetteten System zusammengeführt. Dieses System liegt zunächst nur als Blackbox Beschreibung vor. Die Boxen, welche die einzelnen Module umfassen und die Schnittstellen zur hierarchisch höchsten Ebene spezifizieren, werden als Wrapper bezeichnet. Durch die Synthese wird diese abstrakte Beschreibung der Schaltung automatisch in Netzlisten (Native Generic Circuit (NGC) bzw. EDIF Files) überführt. Für jeden IP Core Wrapper wird dabei eine eigene Netzliste erzeugt.
Für ein rekonfigurierbares Design auf V2 müssen dem Design beim modulbasierten Ent-

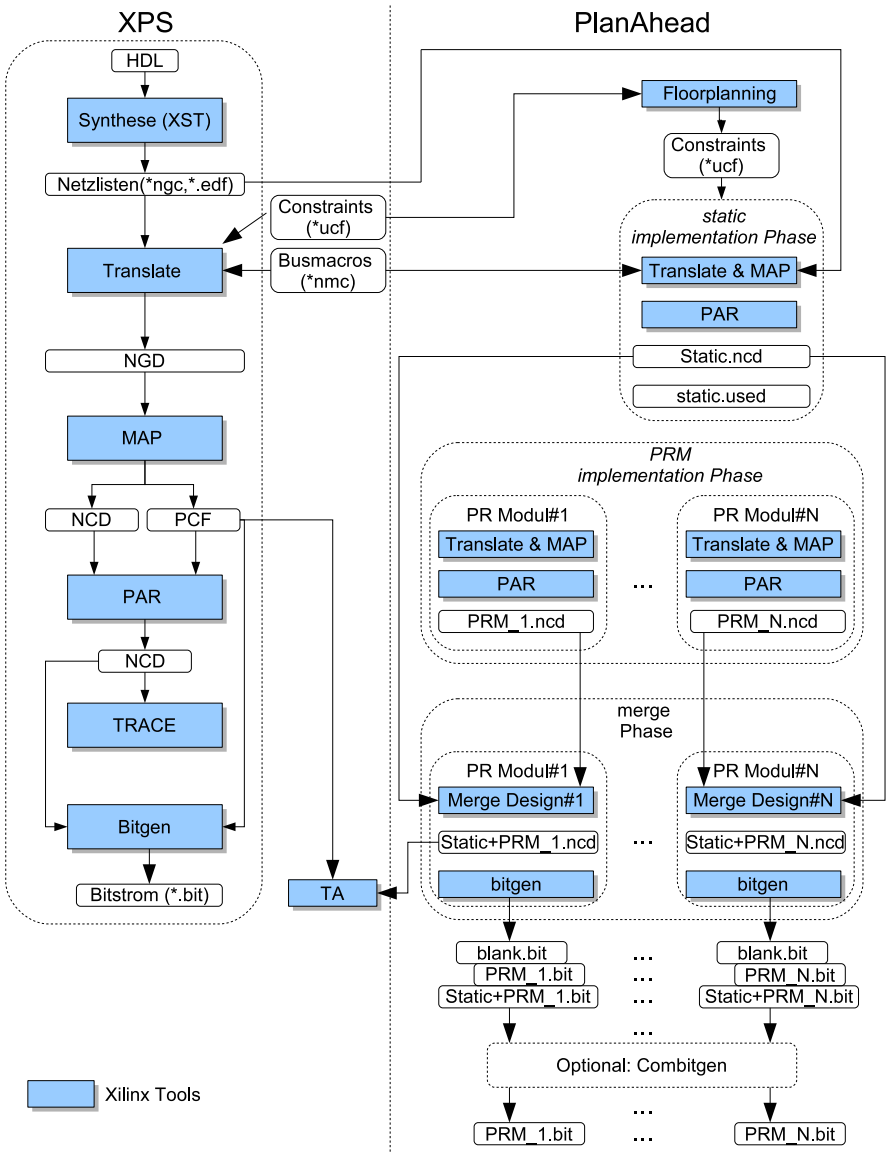


Abbildung 5.16: Designflow für ein dynamisch rekonfigurierbares System

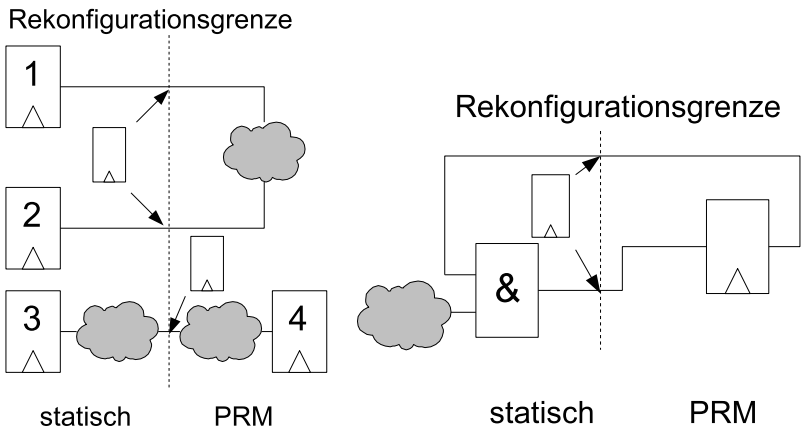
wurfsfluß noch Busmacros hinzugefügt werden. Diese liegen als Native Macro Circuit (NMC) Files vor. Im anschließenden Prozess werden die Netzlisten aller IP Core Wrapper, der Busmacros und ein User Constraint File (UCF), das Timing- und Platzierungsbedingungen enthält, in ein so genanntes Xilinx Native Generic Database (NGD) File umgewandelt. In diesem Translate Prozess werden die logischen Elemente in Xilinx Primi-

tives übersetzt. Im MAP Prozess werden die im NGD File definierten logischen Elemente auf FPGA Elemente, wie BRAMs und CLBs abgebildet. Das erzeugte File ist eine unverdrahtete, so genannte Native Circuit Description (NCD). Die logischen Constraints, die über das UCF in das NGD übernommen wurden, werden vom MAP Prozess für die Abbildung auf FPGA Elemente verwendet. Zusätzlich werden bei diesem Schritt, nach Festlegung einer bestimmten FPGA Zielarchitektur, logische Constraints in so genannte Physical Constraints umgewandelt und in ein Physical Constraints File (PCF) übernommen. Das PCF beinhaltet einen Teil für die im UCF beschreibenden Contraints und einen Teil, der vom Mapper generiert wurde. Der letztere Teil wird bei jedem Durchlauf des MAP Prozesses modifiziert.
Der anschließende Place and Route Prozess (PAR) nutzt das gemappte NCD File für die Platzierung und Verdrahtung des Designs und generiert ein NCD File für die Erstellung eines Bitstrom Files, mit dem der FPGA anschließend konfiguriert werden kann. Vor der Bitstromerstellung sollte allerdings überprüft werden, ob das Design die geforderten Timing Constraints einhält und damit die erforderliche Taktfrequenz im Design erreicht. Dies wird am Ende jedes automatisch erzeugten PAR Berichts angezeigt. Für einen ausführlicheren Timingbericht wird im XPS Entwurfsfluß das TRACE (Timing Reporter And Circuit Evaluator) Tool verwendet. Nach Eingabe des verdrahteten NCDs und des PCFs wird das Einhalten der Timing Constraints überprüft. Alternativ kann der Timing Analyzer (TA) zur Verifikation des Timings verwendet werden. Verletzt das Design die Timingbedingungen nicht, kann ein partieller Bitstrom mit Hilfe des Bitgen Tools generiert werden. Weiterführende Details zu den einzelnen Prozessschritten und den jeweiligen Optionen sind in [Xil08a] zu finden.
Die Überprüfung auf korrekte Funktionsweise des Designs sollte zunächst mit einem nicht rekonfigurierbaren Design im XPS erfolgen, da DPR Designs aufgrund der zeitlichen Veränderung des Konfigurations-Memory wesentlich schwerer zu debuggen sind als statische Designs. Treten bereits Fehler im statischen Design (mit allen für die Rekonfiguration notwendigen Modulen) auf, kann auf die Erzeugung partieller Bitströme verzichtet werden. Die Tests müssen für alle rekonfigurierbaren Module durchgeführt werden, um die korrekte Funktionsweise im nicht rekonfigurierbaren System zu gewährleisten.
Falls das statische Design inklusive DPR Overhead und dem zu rekonfigurierenden Modul fehlerfrei funktioniert, kann mit der Generierung des DPR Systems begonnen werden. Dazu werden die Netzlisten aus dem XPS Design zusammen mit dem größten PRM, den Busmacros und dem UCF in PlanAhead geladen. Beim Floorplanning eines rekonfigurierbaren Designs wird die Größe (abhängig vom größten PRM) und Position der PRR festgelegt.
Zudem werden die Busmacros an den Grenzen der PRR platziert. Die so erzeugten Constraints werden in das UCF File übernommen. Grundsätzlich besteht der MBRF (s. Abschnitt 2.2.7.2) aus 3 Einzelphasen: Der *static implementation Phase*, der *PRM implementation Phase* und der *merge Phase*. In der *static implementation Phase* werden der statische Anteil des Designs verdrahtet und die zuvor definierte PRR ausgespart. Außer

Netzen des statischen Teil, die die PRR überspannen und eventuelle Anschlußpunkte in den Switch Matrizen innerhalb der PRR nutzen, wird innerhalb der PRR nichts konfiguriert. Da diese Anschlußpunkte von den PRMs nicht mehr verwendet werden dürfen, wird neben dem platzierten und verdrahteten NCD File ein weiteres File erzeugt, welches die vom statischen Design bereits verwendeten Anschlußpunkte beinhaltet. Dieses so genannte „static.used“ File wird bei der Platzierung und Verdrahtung jedes einzelnen PRM Moduls verwendet. Dies ist notwendig, um in der anschließenden *merge Phase* sicher zu stellen, dass es zu keinen mehrfach belegten Ressourcen innerhalb der PRR kommt.
Die parallele Implementierung der rekonfigurierbaren Module auf Multiprozessor-Systemen wird von den Xilinx Tools unterstützt. Ergebnis jeder PRM Implementierung ist ein innerhalb der PRR platziertes und verdrahtetes NCD file. Innerhalb bedeutet hier, dass weder logische Elemente noch Netze außerhalb der PRR verlaufen. Die im static.used File aufgeführten Anschlußpunkte werden dabei in den PRMs ausgespart. Die erzeugten NCD Files werden anschließend jeweils mit dem statischen Design in der *merge Phase* zusammengeführt. Zwischenergebnis in diesem Prozessschritt ist ein platziertes und verdrahtetes NCD File, das sowohl den statischen als auch den rekonfigurierbaren Teil beinhaltet. Da in der Merge Phase der MAP Schritt nicht erneut ausgeführt wird, wird kein neues PCF File für das zusammengeführte Design erzeugt. Dieses PCF ist jedoch für die statische Timinganalyse sehr wichtig. Jedoch kann das PCF des entsprechenden XPS Designs zur Verifikation verwendet werden, sofern die Constraints in diesem Design und dem rekonfigurierbaren Design dieselben sind. Sind alle Timing Constraints eingehalten, können die erzeugten partiellen Bitströme verwendet werden. Neben einem Bitstrom für die initiale Programmierung des FPGAs mit statischem und rekonfigurierbarem Teil, wird ein partieller Bitstrom erzeugt, der nur das rekonfigurierbare Modul enthält. Zusätzlich wird ein weiterer partieller leerer Bitstrom (Blank Bitstrom) erzeugt, der zum Löschen der PRR verwendet werden kann.

5.5.1 Probleme des modulbasierten Entwurfsflusses

Wie in Abschnitt 2.2.7.3 bereits erwähnt, gibt es zwei Arten von Macros: Synchrone und Asynchrone. Das Einfügen von asynchronen Macros verlängert den kombinatorischen Pfad, verglichen mit dem nicht rekonfigurierbaren Design. Durch synchrone Macros wird eine zusätzliche Registerstufe in den logischen Pfad eingefügt.
Abbildung 5.17 zeigt zwei Probleme bei der Verwendung von synchronen Macros. In Abbildung 5.17(a) liegen die Register 1 und 2 beispielsweise im statischen Teil und sind durch einen kombinatorischen Pfad im PRM miteinander verbunden. Kombinatorische Pfade sind durch graue Logikwolken dargestellt. Ein weiteres Register 3 im statischen Teil und Register 4 im PRM sind ebenfalls durch kombinatorische Teile im statischen sowie im PRM Teil miteinander verbunden. Die Ergebnisse durch die kombinatorische Logik werden im selben Takt an den Registern 2 und 4 erwartet. Durch Einfügen von synchronen Macros wird das Ergebnis an Register 2 um zwei Takte und an Register 4 um einen Takt



(a) Erstes Problem **(b)** Zweites Problem

Abbildung 5.17: Probleme bei der Verwendung von synchronen Macros

verzögert. Damit erscheinen die Ergebnisse an den Registern 2 und 4, um einen Takt verzögert. Ein weiteres Problem bei der Verwendung von synchronen Macros entsteht bei Feedback Pfaden, wie in Abbildung 5.17(b) zu erkennen ist. Im Takt x werden der Ausgang aus der kombinatorischen Logik und das Ergebnis aus den vorherigen Takt x-1 in einem AND-Gatter zusammengeführt. Verläuft die Rekonfigurations-Grenze zwischen Gatter und Register und werden synchrone Macros verwendet, stimmen die eingehenden Signale am Gatter zeitlich nicht mehr überein. Die Verwendung von synchronen Macros kann also zu Änderungen an der Funktionalität der Schaltung führen. Ist dies nicht der Fall, sollten synchrone Macros auf jeden Fall verwendet werden. Die Anbindung des DCR IF an das PRM wird beispielsweise über synchrone Macros realisiert, da in diesem Fall nur die Latenz erhöht, nicht aber die Funktionalität verändert wird. Falls möglich, sollten immer synchrone Macros verwendet werden.
Die beiden zuvor genannten Probleme treten beispielsweise bei der Anbindung des LIS PLB IPIF über synchrone Macros an den HWA auf. Deshalb müssen in der AutoVision Architektur auch asynchrone Macros verwendet werden. Da es jedoch, wie bereits erwähnt, keine Gesamtoptimierung hinsichtlich Timing beim in [SBB+06] präsentierten modul-basierten Designflow gibt, führt die Verwendung von asynchronen Macros zu anderen Problemen.
Wie beim Entwurfsfluß in Abbildung 5.16 dargestellt, können globale Timing Verletzungen erst detektiert werden, wenn die *merge Phase* abgeschlossen ist. Wenn das Gesamtdesign bei einer Frequenz von 100 MHz laufen soll, kann der kritische Pfad eine maximale Länge von 10 ns haben, ohne das Timing zu verletzen.
Wie in Abbildung 5.18 zu erkennen ist, existiert zwischen Registern und asynchronen

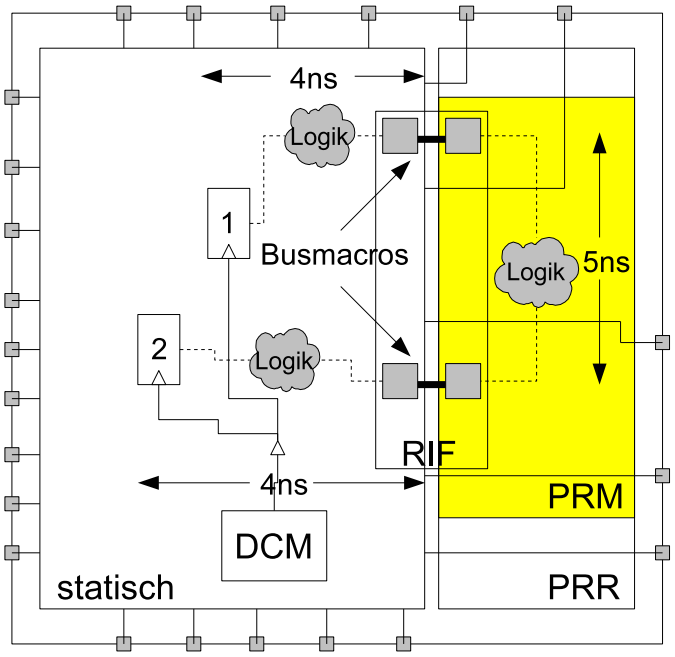


Abbildung 5.18: Probleme bei der Verwendung asynchroner Macros

Busmacros kombinatorische Logik. Bei der Platzierung und Verdrahtung des statischen Teils kommt es zu keinen Timingverletzungen, da beide Pfade zwischen Registern und Busmacros 4 ns betragen. Dies gilt ebenfalls für den Pfad im PRM, der eine Verzögerung von 5 ns hat. Nach dem Zusammenführen beider Designs in der *merge Phase* entsteht jedoch ein langer kombinatorischer Pfad von (4+5+4) = 13 ns, was zu einer Verletzung der Timing constraints führt. Dies resultiert aus dem bereits erwähnten Problem, dass keine globale Optimierung über das Gesamtdesign stattfindet und statischer Teil und PRM keine Kenntnis voneinander besitzen. Die Problematik bei der Verwendung asynchroner Macros ist in [Che07] detailliert beschrieben.
Zur Veranschaulichung der Problematik wird das rekonfigurierbare SoC in verschiedene Mengen unterteilt. Trivial ist die Aufteilung des Systems in zwei disjunkte Mengen: Die Menge S, welche die Elemente des statischen Teils enthält und die Menge R, die ein rekonfigurierbares Modul enthält. S enthält somit die IP Cores des statischen Teils (Video input/output, memory controller, CPU etc.) und R enthält das PRM. Die Aufteilung in verschiedene Mengen ist in Abbildung 5.19 dargestellt.
Methoden, wie die Platzierung von Macros nahe bei einander, können nicht garantieren, dass das Timing tatsächlich eingehalten wird. Dadurch, dass S und R im in [SBB+06] vorgestellten Designflow komplett isoliert voneinander betrachtet werden, handelt es sich bei der Verwendung von asynchronen Macros um Zufall, wenn die Zeitbedingungen ein-

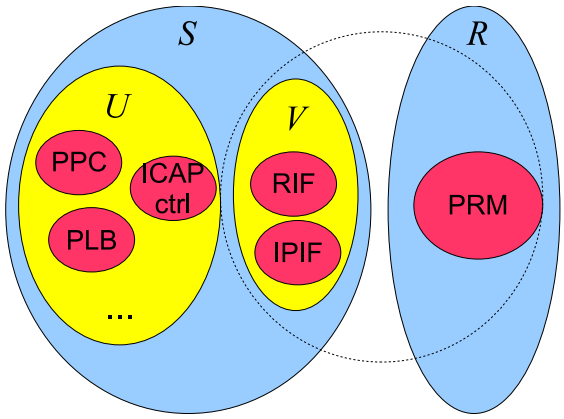


Abbildung 5.19: Aufteilung des rekonfigurierbaren SoCs in verschiedene Mengen

gehalten werden.

5.5.2 Überwindung des modularen Design Gaps

Die Lösung der Problematik besteht in einer weiteren Unterteilung der Menge S. Die IP Cores des statischen Teils, die über asynchrone Macros mit dem PRM verbunden sind werden in der Menge V zusammengefasst. Im AutoVision System sind dies das *RIF* und das IPIF. Der Rest des statischen Designs bildet die Menge U. S ist also die Vereinigungsmenge der beiden disjunkten Mengen U und V. Anstatt nun statischen und rekonfigurierbaren Anteil getrennt von einander zu platzieren und zu verdrahten, werden in der *module phase* V und R zusammen implementiert. Dies ist durch die gestrichelte Linie in Abbildung 5.19 gekennzeichnet. Aus dem bereits platzierten und verdrahteten statischen Design werden mit Hilfe des Xilinx Floorplanners oder Planahead die Platzierung und die Verdrahtung der Elemente der Menge V extrahiert und in das UCF der PRMs übernommen. Die platzierten CLBs etc. werden dabei durch Platzierungsconstraints festgehalten. Die Beibehaltung der Verdrahtung erfolgt in einzelnen, besonders kritischen Fällen durch so genannte Directed Routing Constraints. Dadurch werden die Elemente von V mit in die Platzierung und Verdrahtung während der *module phase* übernommen. Ist die Platzierung und Verdrahtung für die Mengen R und V abgeschlossen, und wurden keine Timingfehler im Design festgestellt, treten durch die Einbeziehung der Menge V nach der Merge Phase auch keine Timingfehler auf. In der Merge Phase wird dann die Menge V aus dem Modulteil entfernt und gegen die platzierten und verdrahteten Elemente von V aus dem statischen Teil ersetzt. Durch diese Methodik wird sichergestellt, dass es nach der Merge Phase zu keinen Timingfehlern kommt.

5.6 Zusammenfassung

Ziel der in diesem Kapitel vorgestellten Konzepte und Realisierungen, war der Aufbau einer dynamisch rekonfigurierbaren Architektur zur Bildverarbeitung. Zunächst wurde ein Szenario vorgestellt, in dem verschiedene Bildverarbeitungsalgorithmen aufgrund der gegenwärtigen Fahrsituation in video-basierten Fahrerassistenzsystemen eingesetzt werden. Da die Umsetzung der Algorithmen im AutoVision System besteht beschleunigten HW-Teil (HWAs) und einem flexiblen SW-Teil. Da in bestimmten Fahrsituationen nur ein Subset der verfügbaren Algorithmen zur Bildverarbeitung und damit auch nur ein geringer Teil der HWAs benötigt wird, wurde eine Architektur zum situationsbedingten Austausch von HWAs während der Laufzeit des Systems vorgestellt.
Die DPR bedarf einer einheitlichen Schnittstelle zum Austausch der HWAs. Die im AutoVision System verwendeten HWAs bestehen alle aus den Elementen der Modulbibliothek. Durch die unterschiedliche Konfiguration und Verschaltung der Einzelelemente der Modulbibliothek, sind HWAs sehr verschieden, so dass es sinnvoll ist, das gesamte Modul auszutauschen.
Um ein nicht rekonfigurierbares System in ein DPR System zu überführen sind zusätzliche Rekonfigurations-Kosten aufzuwenden. Unter Rekonfigurations-Kosten wird der Zusatz an Fläche, Zeit und Verlustleistung verstanden, der durch die Einführung von DPR im System entsteht. Daher wurden in diesem Kapitel Konzepte und deren Umsetzung zur Einsparung von Rekonfigurationskosten vorgestellt. In einem DPR System sollen sich gegenseitig ausschließende HWAs dieselbe Fläche in einer Art Zeit-Multiplexbetrieb teilen. Die Einsparung nicht benötigter HWAs hat ebenfalls einen Einfluß auf Fläche, Zeit und Verlustleistung. Insgesamt wird im rekonfigurierbaren AutoVision System versucht möglichst viel Fläche durch den Einsatz von DPR einzusparen, bei gleichzeitiger Minimierung der für den dynamischen Austausch notwendigen Zeit und Verlustleistung. Aus diesem Grund wurde der Durchsatz an Rekonfigurationsdaten maximiert, um den Austausch verschiedener HWAs möglichst schnell durchzuführen. Zusätzlich wurde ein Konzept für ein dynamisches Powermanagement vorgestellt, mit dem Verlustleistung eingespart werden soll, indem Module vom Takt getrennt werden bzw. komplett vom FPGA mit Hilfe der DPR entfernt werden.
Durch die Einführung von DPR in einem integrierten FPGA-basierten System, soll der normale Ablauf nicht gestört werden. Das bedeutet, dass es inakzeptabel ist, wenn durch die DPR ein oder gar mehrere Video-Frames verworfen werden müssen. Insbesondere, wenn es sich um sicherheitskritische Funktionen handelt. Abhängig davon, ob eine DPR zwischen zwei aufeinander folgenden Video-Frames, oder mehrere DPRs innerhalb eines Video-Frames stattfinden, wird zwischen Inter Video Frame Rekonfiguration (InterVFR) und Intra Video Frame Rekonfiguration (IntraVFR) unterschieden.
Durch Festlegen bestimmter Zeitpunkte während der Verarbeitung eines Video-Frames, kann sichergestellt werden, dass bei möglichst kurzen Rekonfigurations-Zeiten kein Video-Frame verworfen werden muß und die DPR somit „verlustfrei“ abläuft. In einem Re-

konfigurations-Scheduling sind daher Rekonfigurations-Zeitpunkte und -Ablauf festgelegt worden.
Zusätzlich sind Probleme und deren Lösung beim modulbasierten Entwurfsfluß [LBM+06] aufgezeigt worden, der für die Erstellung des InterVFR Designs auf einem V2P FPGA verwendet wurde. Für die Implementierung des IntraVFR Designs auf einem V5 wurde ein neuartiger partitionsbasierter Entwurfsfluß [Xil10c] verwendet, bei dem diese Probleme nicht mehr auftraten.

6 Ergebnisse und Bewertung

In diesem Kapitel werden die am Prototyp gemessenen Ergebnisse hinsichtlich Performanz, Ressourcenverbrauch und Verlustleistung dargestellt. Diese Ergebnisse werden mit in der Literatur verfügbaren Ergebnissen verglichen und anschließend bewertet.

6.1 Verarbeitungszeit der Hardware Acceleratoren

In diesem Abschnitt sind die Verarbeitungszeiten der einzelnen HWAs dargestellt. Die Messungen wurden an lauffähigen, prototypischen Meßaufbauten vorgenommen. Der Prototyp für die InterVFR wurde auf einem XUPV2P Board der Firma Digilent mit einem XC2VP30 FPGA implementiert. Aufgrund der nur 8 Bit breiten ICAP Schnittstelle auf dem V2P Device wurde der Prototyp für die IntraVFR auf einem ML507 Board mit einem XC5VFX70T FPGA implementiert. Zunächst werden die Ergebnisse auf der V2P Plattform betrachtet.
Um zyklengenau die Ausführungszeit zu bestimmen, wurde in allen HWAs ein HW-Zähler implementiert. Dieser Zähler beginnt mit dem ersten empfangenen DCR Kommando, das vom PPC gesendet wurde, um den HWA zu initialisieren und zu starten. Die Messung endet, wenn der HWA ein Interrupt Signal absetzt, um dem PPC mitzuteilen, dass die Verarbeitung abgeschlossen ist. Neben der Verarbeitungszeit spielt auch die Effektivität der Pixelverarbeitung eine wichtige Rolle. Da eine beliebig hohe Taktfrequenz f zu einer beliebig niedrigen Verarbeitungszeit t führen kann, wird der Ergebnisdurchsatz betrachtet, also wie viele Pixel pro Takt ppt verarbeitet werden können. Die Anzahl an Pixeln ist abhängig von den Dimensionen des Eingangsbildes I_H und I_W. Alle Messungen wurden, falls nicht anders erwähnt, mit Bildern der Dimensionen $I_W = 640$ und $I_H = 480$ durchgeführt. In Gleichung 6.1 ist die Berechnung des Durchsatzes ppt dargestellt.

$$ppt = \frac{I_W * I_H}{f * t} \tag{6.1}$$

Als effektiv gilt in dieser Arbeit ein Durchsatz von $ppt = 1$. Das bedeutet, dass in jedem Taktzyklus ein Pixel mit seiner Nachbarschaft verarbeitet werden kann, was auch der Zielsetzung im AutoVision System entspricht. Ein Bild mit VGA Auflösung (640×480) besteht aus insgesamt 307200 Pixeln, die verarbeitet werden müssen. Alle HWAs im InterVFR Design auf dem V2P System arbeiten bei einer Taktfrequenz von 100 MHz (10 ns Zyklenzeit). Daraus ergibt sich eine theoretische Verarbeitungszeit von 307200 x 10 ns = 3072 μs, wenn jeden Taktzyklus ein Pixel verarbeitet werden kann. Da in der modulbasierten Pixelverarbeitungspipeline jedoch zunächst der lokale Speicher (LIM) gefüllt

werden muß, bevor mit der Bildverarbeitung begonnen werden kann, wird die Verarbeitungszeit etwas größer sein. Nachdem das letzte Pixel verarbeitet wurde, werden die letzten verarbeiteten Pixel zurück in den Hauptspeicher geschrieben, was die Verarbeitungszeit zusätzlich erhöht. Bei einer Framerate von 31 fps bleiben $1/31 = 32{,}25$ ms Zeit, um die Pixel-Level Operationen in HW und den Higher-Level Application Code in SW zu verarbeiten. Gelingt es, die Verarbeitung der Pixel-Level Operationen in etwa 3,25 ms abzuschließen, bleiben 29 ms für die Verarbeitung des Higher-Level Application Codes auf dem PPC. Bei der parallelen Implementierung zweier gleicher Hardwarebeschleuniger und der Aufteilung der Bilddaten in zwei gleich große Bereiche, lässt sich die Ausführungszeit in HW auf Kosten der zusätzlichen logischen Ressourcen halbieren. Dies gilt jedoch nur, wenn die Pixeldaten für beide HWAs schnell genug aus dem Speicher geladen werden können und wenn der Algorithmus eine Aufteilung erlaubt. Die Ergebnisse aller im Folgenden vorgestellten Implementierungen sind in Tabelle 6.2 aufgeführt.
Zunächst wird die in Abschnitt 4.4.1.1 beschriebene *ShapeEngine (SE)*, die den modifizierten SUSAN Eckendetektor [CHRS09] realisiert, untersucht. Die *SE* benötigt 3142 μs, um das komplette Bild zu verarbeiten. Insgesamt können laut Gleichung 6.1 mit der *SE* $(640 \times 480)/(100MHz \times 3,142ms) = \mathbf{0.9777}$ Pixel pro Taktzyklus (*ppt*) verarbeitet werden, was einer theoretischen Framerate von 1 Frame/ 3,142 ms = 318.27 fps entspricht. Dies ist annähernd ein Pixel pro Taktzyklus. Die *SE* übertrifft somit die Ergebnisse in [THAE00] (119.76 fps bei 60 MHz, 512×512 Auflösung und *ppt* = **0,526**) und [AERP02] (60 fps bei 66MHz, 640×480 Auflösung und *ppt* = **0,279**).
Um eine Schätzung für die Beschleunigung zu erhalten, die mit der *SE* erreicht werden kann, wurde eine optimierte C Version des SUSAN Algorithmus mit OpenCV [Bra00] implementiert. Tests auf einem Intel Core2Duo E8400 bei einer Taktfrequenz von 3,00 GHz lieferten 11,7 ms Sekunden bei der Verarbeitung mit dem unmodifizierten SUSAN Algorithmus [SB97], 15,6 ms für den Harris/Plessy Algorithmus [HS88] und 27,0 ms für den modifizierten SUSAN Algorithmus, wie er in Abschnitt 4.4.1 beschrieben wurde.
Der Vergleich mit der Implementierung des Algorithmus auf einer CPU zeigt die mögliche Beschleunigung (3,142 ms vs. 27,0 ms) in HW sogar bei 30-fach niedrigeren Taktfrequenzen. Auch auf GPUs lassen sich Bildverarbeitungsalgorithmen beschleunigen. Zum Zeitpunkt dieser Arbeit waren allerdings keine Implementierungen des SUSAN Algorithmus in der Literatur erwähnt. In [TCG08] werden allerdings Ergebnisse des Harris/Plessy Algorithmus auf einer GPU dargestellt. Obwohl die Verarbeitungszeit auf der NVIDIA GeForce 8800 GTX 768 Mb GPU mit einer Shader-Taktfrequenz von 1350 MHz geringer ist, lassen sich umgerechnet nur $(640 \times 480)/(1350MHz \times 2.6ms) = \mathbf{0.088}$ Pixel pro Taktzyklus verarbeiten.
Im Anschluß wurde die Ausführungszeit der in Abschnitt 4.4.2.1 vorgestellten *Taillight-Engine (TE)* untersucht. In der *TE* wird zunächst das gesamte Bild mit einem Sliding Window prozessiert, um mögliche Lichtpunkte (Feature) zu finden. Nach diesem Schritt wird für jede Lichtregion ein Label vergeben. Wie in Abschnitt 4.4.2 beschrieben, muß auf die Verarbeitung des letzten Pixels gewartet werden, bevor die Liste von gelabelten

Lichtern zurück in den Hauptspeicher geschrieben werden kann. Nach der Verarbeitung des letzten Pixels wird die in der *TE* zwischengepufferte Liste von gelabelten Lichtern mit Hilfe von Burst-Transfers komplett in den Hauptspeicher übertragen, was die Zeit insgesamt erhöht. Daher ergibt sich für die *TE* eine Verarbeitungszeit von 3177 μs. Die *TE* kann somit $(640 \times 480)/(100MHz \times 3,177ms)$ = **0.9669** Pixel pro Taktzyklus (*ppt*) verarbeiten. Zwar existieren in der Literatur ähnliche Algorithmen zur nächtlichen Rücklichterkennung von Fahrzeugen, jedoch keine FPGA Realisierungen davon. In [FSF10] wurde ein ähnlicher Algorithmus auf einer Pentium D 1.8 GHz dual core CPU implementiert. Um die reine Verarbeitungszeit zu messen, wurde die Monitorausgabe der verarbeiteten Daten deaktiviert. Im besten Fall konnte somit eine Rate von 47 fps bei einer Bildauflösung von 720×576 erreicht werden. Die Verarbeitung von 47 fps resultiert in einer Verarbeitungszeit pro Bild von 1 Frame/ 47 fps = 21,27 ms. Somit können $(720 \times 576)/(1800MHz \times 21,27ms)$ = **0.0108** Pixel pro Taktzyklus (*ppt*) verarbeitet werden. Weitere Algorithmen zur Rücklichterkennung werden in [OGJ08] und [ABJ+08] vorgestellt. Allerdings fehlen die Angaben zur Verarbeitungszeit und der verwendeten Plattform.
Die in Abschnitt 4.4.3.1 beschriebene *ContrastEngine (CTE)* besitzt nur eine *User Logic* zur Realisierung von Tiefpassfilter und Histogrammspreizung und benötigt daher auch die geringste Verarbeitungszeit von 3081 μs. Dies bedeutet wiederum, dass $(640 \times 480)/(100MHz \times 3,081ms)$ = **0.997** Pixel pro Taktzyklus mit der *CTE* verarbeitet werden können. In [KM09] wird ebenfalls ein HWA für die Histogrammspreizung vorgestellt. Bei einer Frequenz von 209,4 MHz kann dieser HWA 537.9 Bilder pro Sekunde mit einer Auflösung von 640×480 Pixeln verarbeiten. Dies resultiert in einer Verarbeitungszeit von 1/537,9 fps = 1,859 ms pro Bild und damit in $(640 \times 480)/(209,4MHz \times 1,859ms)$ = **0.7891** verarbeiteten Pixeln pro Taktzyklus.

	Verarbeitungszeit	
Settings	ϵ = **30** $d_{c1} = 3$, $d_{c2} = 7$, ca. 17000 features	ϵ = **8** $d_{c1} = 3$, $d_{c2} = 7$, ca. 17000 features
Image Operation	**HW Model** (FPGA 100MHz,) (PPC 300MHz)	**SW Model** (Core 2, 1.86GHz)
Tiefpaßfilter	3.95ms	1.87ms
Census Transformation		38.68ms
Matching	5.92ms	
Einzeichen von Bewegungsvektoren	12.3ms	
Gesamtzeit	22.17ms	40.55ms

Tabelle 6.1: Vergleich der Verarbeitungszeit des Optischen Flusses in HW und SW basierend auf etwa 17000 gefundenen Features

Die Census-Transformation zur Berechnung des Optischen Flusses wurde in den HWAs *CensusEngine (CSE)* und *MatchingEngine (ME)* implementiert und in Abschnitt 4.4.4.1

bzw Abschnitt 4.4.4.1 vorgestellt. Zunächst wird die Verarbeitungszeit der beiden HWAs des Optischen Flusses auf der V2P Plattform betrachtet. Die Ausführungszeit der *CSE* beträgt 3950 μs (*ppt* = **0,7777**) und beinhaltet die Rauschunterdrückung mit einem Tiefpaßfilter und die Census-Transformation (s. Abschnitt 4.4.4.1). Da die *CSE* 8-Bit Grauwert-Pixel liest, jedoch 32-Bit Farbpixel zurück in den Hauptspeicher schreibt, bremst der Ausgangsdatenpfad den Eingangsdatenpfad aus. Zudem unterstützt der verwendete DDR SDRAM Controller auf der V2P Plattform keine simultanen Lese- und Schreib-Transfers, wodurch sich die erhöhte Verarbeitungszeit der *CSE* erklären lässt. Dieses Problem führt in der *ME* zu einer noch höheren Verarbeitungszeit von 5920 μs (*ppt* = **0,5189**). In der *ME* müssen die Daten der beiden Census-Bilder $\Xi_k(x, y)$ und $\Xi_{k+1}(x, y)$ in zwei Eingangspipelines geladen werden, wie in Abbildung 4.49 zu erkennen ist. Messungen zeigen, dass etwa 30 Takte benötigt werden, Daten über einen Burst der Weite $B_W = 16$ komplett zu übertragen. Die 30 Takte wurden vom ersten Request des Masters über das IPIF bis zur Übertragung des letzten Wortes gemessen. Die Request-Phase hat somit eine Länge von 14 Takten bis das erste Datenwort am Eingang der *ME* erscheint. Wenn 2 Census-Bilder mit $P_W = 32$ Bit und einer Auflösung von 640×480 zur *ME* in 128 Byte Bursts übertragen werden sollen, werden $\frac{2*640*480*32bits}{128*8bit} = 19,200$ Burst-Transfers benötigt. Wenn für jeden Burst-Transfer 30 Takte benötigt werden, ergibt sich eine Zeit von $19,200 * 30 * \frac{1}{100Mhz} = 5760\mu s$, was der gemessenen Verarbeitungszeit von 5920 μs sehr nahe kommt. Im System zur Berechnung des Optischen Flusses, kann damit klar der Memory Controller [Xil06a] als Flaschenhals identifiziert werden. Abhilfe könnten in diesem Fall Bursts der Weite B_W=32 und die Verwendung von Address Pipelining schaffen, die beide jedoch nicht vom verwendeten Memory Controller unterstützt werden. Die Verarbeitung der Feature in SW ist stark abhängig von der Anzahl gefundener Korrespondenzen (Feature). Zahlreiche Tests haben belegt, dass etwa 1000 Features pro Millisekunde auf dem mit 300 MHz getakteten PPC verarbeitet werden können. Im Gegensatz dazu ist die Verarbeitungszeit in HW unabhängig von der Anzahl gefundener Korrespondenzen und damit konstant. Um wiederum ein Gefühl für die Beschleunigung mit Hilfe der beiden HWAs zu bekommen, wurde die Ausführungszeit mit einer optimierten SW Variante ermittelt, die auf dem in [Ste04] vorgestellten Algorithmus basiert. Die Verarbeitungszeiten sind in Tabelle 6.1 dargestellt.
Für das IntraVFR Design wurde wie beschrieben, der Optische Fluß auf das ML507 Board portiert. Aufgrund von auftretenden Timingfehlern sind die *CSE* und die *ME* nur mit 75 MHz getaktet. Daraus ergibt sich eine theoretische Verarbeitungszeit der Bilder mit VGA Auflösung von 307200 x 1/75MHZ = 4096 μs. Für die Verarbeitungszeit eines Bildes mit der *CSE* wurde ein Wert von 4,3 ms (*ppt* = **0,9534**) gemessen. Die Verarbeitungszeit mit der *ME* betrug 6,5 ms (*ppt* = **0,6308**). In [WLN07] wird die FPGA Implementierung eines 3D-Tensor basierten Verfahrens [Far00] zur Bestimmung des Optischen Flusses beschrieben. Auf der gewählten Plattform, einem V2P FPGA, können 64 fps bei einer Auflösung von (640×480) mit 100 MHz Taktfrequenz verarbeitet werden. Ein einzelner Frame wird somit in 1 Frame/ 64fps = 15,625 ms prozessiert. Damit

können $(640 \times 480)/(100MHz \times 15,625ms) =$ **0,1966** Pixel pro Taktzyklus berechnet werden. In einem Frühwerk [WH97] wurde die Census-Transformation auf der PARTS[1] Engine zur Berechnung einer Stereo-Korrelation[2] implementiert. Die PARTS Engine ist ein PCI-Board mit 16 Xilinx 4025 FPGAs und verarbeitet 42 fps bei einer Auflösung von (320×240). Die Ergebnisse sind aus heutiger Sicht nicht mehr repräsentativ, die Literaturquelle wurde jedoch aus Gründen der Vollständigkeit trotzdem aufgeführt. Ein Vergleich ist somit unangebracht. Zusammenfassend sind die Ergebnisse aller zuvor beschriebenen Implementierungen nochmals in Tabelle 6.2 dargestellt.

Algorithmus	Plattform	Modell	Frequenz (MHz)	Bild-Auflösung	*ppt*	Zeit- (ms)
Eckendetektion						
SUSAN [THAE00]	FPGA	XCV50	60	512×512	0,526	8,35
SUSAN [AERP02]	FPGA	XCV1000	66	640×480	0,279	16,67
Harris [TCG08]	GPU	8800 GTX	1350	640×480	0,088	2,6
SUSAN	CPU	Intel E8400	3000	640×480	0,0038	27
SUSAN	FPGA	V2P30	100	640×480	0,9777	3,142
Rücklichtdetektion						
[FSF10]	CPU	Pentium D	1800	720×576	0.0108	21,27
[ACS08]	FPGA	V2P30	100	640×480	0,9669	3,177
Kontrasterhöhung						
Histogramm-spreizung [KM09]	FPGA	V4LX160	209,4	640×480	0,7891	1,859
Histogramm-spreizung	FPGA	V2P30	100	640×480	0,997	3,081
Optischer Fluß						
Tensor-basiert[WLN07]	FPGA	V2P30	100	640×480	0,1966	15,625
Census-Transformation *CSE*	FPGA	V2P30	100	640×480	0,7777	3,95
Census-Transformation *ME*	FPGA	V2P30	100	640×480	0,5189	5,92
Census-Transformation *CSE*	FPGA	V5FX70T	75	640×480	0,9534	4,3
Census-Transformation *ME*	FPGA	V5FX70T	75	640×480	0,6308	6,5

Tabelle 6.2: Vergleich der Ausführungszeiten verschiedener Algorithmen für die Bildverarbeitung auf verschiedenen Plattformen

Wie in Tabelle 6.2 zu erkennen ist, arbeiten die HWAs im AutoVision System (mit Ausnahme der *ME*) sehr effektiv ($ppt \sim 1$). In beiden Prototypen wurden die HWAs über ein LIS PLB IPIF angebunden. Bei einem direkten Anschluß über ein LIS NP IPIF ist eine weitere Verkürzung der Verarbeitungszeit zu erwarten.

[1] **P**rogrammable **A**nd **R**econfigurable **T**ool **S**et

[2] In einem Stereo-Kamerasystem werden das linke und das rechte Kamerabild korreliert, um eine 3D Tiefenschätzung zu erhalten.

6.2 Verbrauch an logischen Ressourcen (Fläche)

In diesem Abschnitt ist der logische Ressourcenverbrauch (Flächenbedarf), also die Anzahl an verwendeten LUTs, Registern, BRAMs etc. im InterVFR und IntraVFR Design samt der darin verwendeten HWAs dargestellt. Zusätzlich ist der Ressourcenverbrauch des ICAP Controllers aufgeführt, der mit anderen in der Literatur verfügbaren Ergebnissen verglichen wird.

6.2.1 Ressourcenverbrauch im InterVFR Design

Das InterVFR Design wurde prototypisch auf einem XUPV2P Board der Firma Digilent mit einem XC2VP30 FPGA implementiert. In Tabelle 6.3 ist der Verbrauch für ein nicht rekonfigurierbares Design, mit *SE*, *CTE* und *TE* und deren Anschlüsse über LIS PLB IPIF und DCR IF dargestellt. Die Zahlen in Klammern geben den prozentualen Verbrauch im Bezug auf die insgesamt auf einem XC2VP30 Device verfügbaren Ressourcen an.

Entity	**SLICES**	**REGISTER**	**LUTs**	**BRAMs**
Total Available (XC2VP30)	13696	27392	27392	136
Static	4057 (30%)	4651 (17%)	5564 (21%)	36 (26%)
ShapeEngine (*SE*)	4846 (35%)	2990 (10%)	8706 (32%)	40 (29%)
ContrastEngine (*CTE*)	697 (5%)	383 (1%)	1299 (5%)	6 (4.5%)
Taillight (*TE*)	3282 (23%)	3106 (11%)	6160 (22%)	12 (9%)
LIS PLB IPIFs (für 3 HWAs)	669 (5%)	1041 (3%)	948 (3.5%)	0 (0%)
DCR IFs (für 3 HWAs)	209 (1%)	198 (1%)	117 (1%)	0 (0%)
Total	13760 (101%)	12369 (45%)	22794 (83%)	94 (69%)

Tabelle 6.3: Ressourcenverbrauch des nicht rekonfigurierbaren Designs (XC2VP30)

In Tabelle 6.4 ist der Ressourcenverbrauch durch den DPR Overhead dargestellt. Der Overhead besteht im InterVFR Design aus dem ICAP Controller (ohne Readback), einem zugehörigen Busanschluß (LIS PLB IPIF) und dem *RIF*.

Entity	**SLICES**	**REGISTER**	**LUTs**	**BRAMs**
LIS-IPIF	223 (1.6%)	347 (1.3%)	316 (1.2%)	0 (0.0%)
ICAP Controller	125 (0.9%)	161 (0.6%)	182 (0.7%)	2 (1.5%)
Disconnector	40 (0.3%)	66 (0.2%)	40 (0.1%)	0 (0.0%)
Total Overhead	388 (2.8%)	574 (2.1%)	538 (2.0%)	2 (1.5%)

Tabelle 6.4: Ressourcenverbrauch des DPR Overhead

Für ein DPR System muß der DPR Overhead hinzugefügt werden, um dadurch unbenutzte HWAs, LIS PLB und DCR IFs einzusparen. Um die Einsparung an Ressourcen

durch die Rekonfiguration mit dem Verbrauch des nicht rekonfigurierbaren Designs (siehe Tabelle 6.3) fair zu vergleichen, wird das rekonfigurierbare Design mit dem größten HWA (*SE*) als Benchmark verwendet. Wie bereits erwähnt, bestimmt der größte rekonfigurierbare IP Core die Größe der PRR und damit die vorzusehende Menge an logischen Ressourcen. Der Ressourcenverbrauch dieses Designs ist in Tabelle 6.5 dargestellt.

Entity	SLICES	REGISTER	LUTs	BRAMs
Total Available (XC2VP30)	13696	27392	27392	136
Static	4057 (30%)	4651 (17%)	5564 (21%)	36 (26%)
ShapeEngine (*SE*)	4846 (35%)	2990 (10%)	8706 (32%)	40 (29%)
DPR Overhead	388 (2.8%)	574 (2.1%)	538 (2%)	2 (1.5%)
LIS PLB IPIF (für 1 HWA)	223 (2%)	347 (1%)	316 (1%)	0 (0%)
DCR IF (für 1 HWA)	40 (0%)	66 (0%)	39 (0%)	0 (0%)
Total	9554 (71%)	8504 (31%)	15163 (55%)	78 (57%)

Tabelle 6.5: Ressourcenverbrauch im InterVFR Design (XC2VP30)

Der geringere Ressourcenverbrauch des rekonfigurierbaren im Gegensatz zum nicht rekonfigurierbaren Design führt in diesem Fall dazu, dass ein kleineres und damit kostengünstigeres FPGA, wie zum Beispiel das XC2VP20 Device verwendet werden kann.

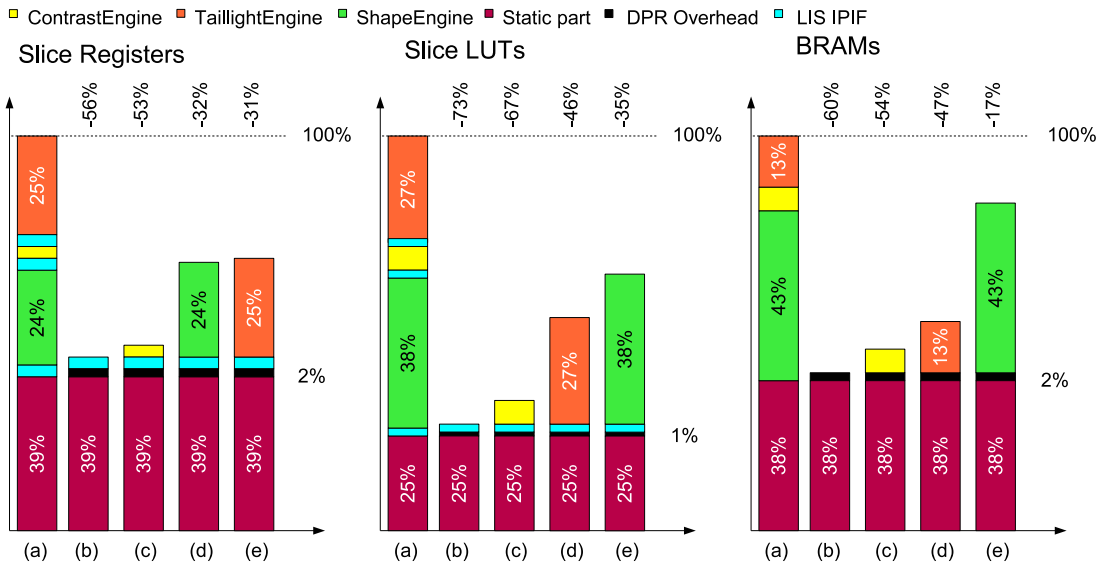


Abbildung 6.1: Ressourcenvergleich zwischen nicht rekonfigurierbarem und InterVFR Design bei unterschiedlichen HWAs

Um den Unterschied zwischen DPR Design und nicht rekonfigurierbarem Design herauszustellen, wird der Ressourcenverbrauch des nicht rekonfigurierbaren Designs als 100%

angenommen. In Abbildung 6.1 ist der Verbrauch in LUTs, Registern und BRAMs angegeben. In den Balkendiagrammen repräsentiert (a) das nicht rekonfigurierbare Design mit statischem Teil, 3 verschiedenen Engines (*CTE*, *TE* und *SE*) und 3 PLB IPIFs. Die DCR Interfaces sind in dieser Darstellung aufgrund ihres geringen Verbrauchs nicht separat aufgeführt. Fall (b) zeigt die Ressourcenauslastung des rekonfigurierbaren Designs, mit DPR Overhead und einem LIS IPIF, wenn ein Blankmodul geladen ist. Der Verbrauch im rekonfigurierbaren Design bei geladener *CTE*, *TE* und *SE* ist in (c), (d) und (e) dargestellt. Die Größe der PRR und damit die verwendbaren Ressourcen werden durch das größte PRM definiert. Denkt man an Einsparung, muß daher immer gegen das Modul verglichen werden, welches am meisten Ressourcen verbraucht. In Abbildung 6.1 ist zu erkennen, dass durch Verwendung von DPR ein erheblicher Anteil eingespart werden kann, nämlich 31% Register, 35% LUTs und 17% BRAMs.

6.2.2 Ressourcenverbrauch im IntraVFR Design

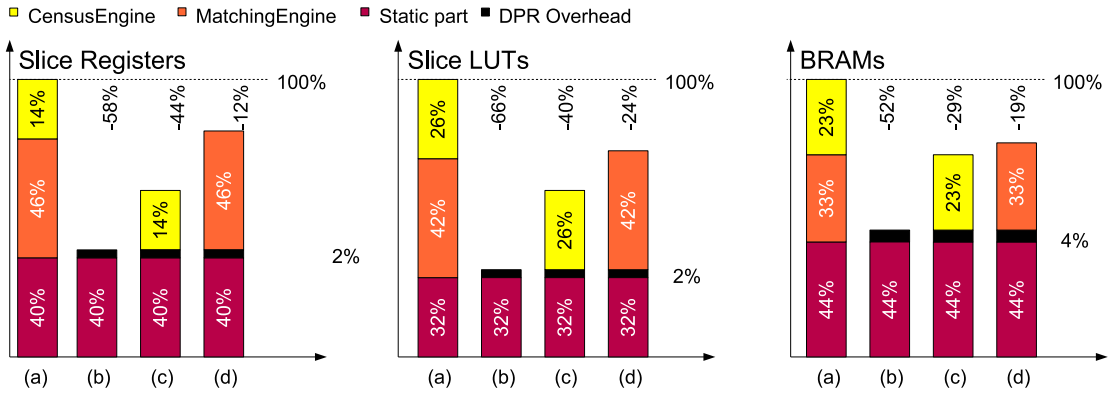


Abbildung 6.2: Ressourcenvergleich zwischen nicht rekonfigurierbarem und IntraVFR Design bei unterschiedlichen HWAs

In Abbildung 6.2 ist der Resourcenverbrauch im IntraVFR System dargestellt. Verglichen wird der Verbrauch an Registern, LUTs und BRAMs des nicht rekonfigurierbaren Designs (a), des rekonfigurierbaren Designs mit einem geladenen Blank Modul (b), geladener CensusEngine (c) und geladener MatchingEngine (d). Um einen fairen Vergleich ziehen zu können, wird wie auch im InterVFR Design, der Ressourcenverbrauch des nicht rekonfigurierbaren Designs als 100% angenommen. Wie aus Abbildung 6.2 ersichtlich ist, können bei der Berechnung des *OF* durch IntraVFR 12% der Register, 24% der LUTs und 19% der BRAMs eingespart werden. Dies scheint auf den ersten Blick nicht sehr beeindruckend. Wenn man sich jedoch vor Augen führt, dass es sich bei der Rekonfiguration von zwei Modulen, um den denkbar schlechtesten Fall handelt, relativiert sich diese geringe Einsparung. Wenn beispielsweise das InterVFR mit dem IntraVFR Design auf

einem Device zusammengeführt wird, sind sehr große Ressourceneinsparungen zu erwarten, da sich dann *SE*, *CTE*, *TE*, *CSE* und *ME* dieselbe Fläche auf dem FPGA teilen würden. Zusätzlich kann man sich DPR für weitere Fahrsituationen vorstellen (z.B. verschiedene Wettersituationen), um den passenden HWA in einer bestimmten Situation zur Verfügung zu stellen.

6.2.3 Ressourcenverbrauch des ICAP Controllers

In Tabelle 6.4 ist der Ressourcenverbrauch des DPR Overhead inklusive ICAP Controller dargestellt. Folgend wird der Ressourcenverbrauch der eigenen Implementierung des ICAP Controllers mit in der Literatur verfügbaren Ergebnissen verglichen. Um einen fairen Vergleich zwischen dem Ressourcenverbrauch bei ICAP Controllern zu erhalten, müssen die unterstützten Funktionalitäten betrachtet und gegeneinander abgewogen werden. Beispielsweise ist wichtig, wo der ICAP Controller angeschlossen ist (OPB, PLB, NPI) und was die Logik innerhalb des Controllers an Fläche (Ressourcen) verbraucht.

	4-LUT (total)	**4-LUT used as logic**	**4-LUT used as shift registers**	**Register**	**BRAMs**
Gesamt (V4FX20)	17088 (100%)	17088 (100%)	17088 (100%)	17088 (100%)	68 (100%)
OPB HWICAP +Bridge	608 (3.6%)	576 (3.4%)	32 (0.2%)	368 (2.2%)	1 (1.5%)
XPS HWICAP	3275 (19.2%)	907 (5.3%)	2368 (13.9%)	417 (2.4%)	0 (0.0%)
DMA HWICAP	4277 (25.0%)	1843 (10.8%)	2434 (14.2%)	977 (5.7%)	0 (0.0%)
MST HWICAP	1083 (6.3%)	1083 (6.3%)	0 (0.0%)	918 (5.4%)	2 (2.9%)
BRAM HWICAP	963 (5.6%)	614 (3.6%)	320 (1.9%)	469 (2.7%)	32 (47.1%)
ICAP_I [LD09]	177 (1.0%)	0 (0.0%)	0 (0.0%)	303 (1.8%)	0 (0.0%)
[HL09]	18 (0.1%)	18 (0.1%)	0 (0.0%)	31 (0.2%)	25 (36.8%)
[LPF09b]	336 (2.0%)	336 (2.0%)	0 (0.0%)	367 (2.1%)	0 (0.0%)
LIS ICAP	334 (2.1%)	332 (2.1%)	2 (0.1%)	269 (1.5%)	2 (2.9%)
LIS PLB IPIF	671 (3.9%)	671 (3.9%)	0 (0.0%)	347(2.0%)	0 (0.0%)
LIS NP IPIF	29 (0.2%)	29 (0.2%)	0 (0.0%)	7 (0.1%)	0 (0.0%)
LIS RB ICAP	529 (3.1%)	529 (3.1%)	0 (0.0%)	218 (1.3%)	2 (2.9%)

Tabelle 6.6: Ressourcenverbrauch verschiedener ICAP Controller auf V4 (Erweiterte Tabelle aus [LKLJ09])

In Tabelle 6.6 ist der Ressourcenverbrauch des ICAP Controllers inklusive der Verifikationslogik (siehe Abschnitt 5.3.2.3) dargestellt. Für einen Vergleich sind zusätzlich in der Literatur verfügbare Ergebnisse aufgeführt. Um einen fairen Vergleich mit den in [LKLJ09] angeführten Ergebnissen zu erhalten, wurde das eigene System für einen V4FX20 synthetisiert. Um einen möglichst großen Vorteil hinsichtlich Ressourceneinsparung durch DPR zu erhalten, muß wie in Abschnitt 5.3.1 beschrieben, der DPR Overhead so gering wie möglich gehalten werden. Auffallend in Tabelle 6.6 sind daher die Ergebnis-

se des BRAM HWICAP aus [LKLJ09] und die Ergebnisse aus [HL09], in denen 32 bzw. 25 BRAMs für den ICAP Controller verwendet werden. In diesen Realisierungen wird versucht, die Bitstromdaten komplett oder zum großen Teil zwischen zu puffern, bevor sie der ICAP Schnittstelle zugeführt werden, um schnellen Zugriff zu gewährleisten. Da diese hohe Anzahl an BRAMs im Vergleich zu einem nicht rekonfigurierbaren System zuerst wieder eingespart werden muß, erscheint diese Lösung unpraktikabel. Mit Ausnahme des OPB HWICAP und des XPS HWICAP unterstützt keiner, der in Tabelle 6.6 aufgeführten ICAP Controller, Readback-Funktionalität. Da eine modifizierte Version des in [Zha07] und [Lan10] beschriebenen ICAP Controllers Readback-Funktionalität besitzt, ist der Verbrauch dieses ICAP Controllers in der letzten Zeile von Tabelle 6.6 angegeben. Wie in [CZS+08] beschrieben, kann der ICAP Controller über das Setzen von Generics an jede Virtex oder Spartan Familie angepasst werden. Aufgrund der 6-Input LUTs und der größeren BRAMs auf V5, verbraucht der LIS ICAP Controller ohne Verifikationslogik 118 Register, 111 LUTs und 1 BRAM. Mit Verifikation kommen 102 LUTs, und 22 Register hinzu.
Werden zusätzlich komprimierte Bitströme wie in [HL09] verwendet, muß der Ressourcenverbrauch für die Dekompressoreinheit mit in den Gesamtverbrauch eingerechnet werden. In [HL09] werden allerdings keine Werte für dieses Modul genannt, jedoch sind vergleichbare Werte für ein HW Modul einer LZSS Dekompression beispielsweise in [HUWJ04] zu finden. Der den jeweiligen ICAP Controllern entsprechende und erreichte Durchsatz an Rekonfigurationsdaten ist im folgenden Abschnitt 6.3 aufgeführt.

6.3 Durchsatz an Rekonfigurationsdaten

In diesem Abschnitt werden die Ergebnisse des auf V2P, V4 und V5 erreichten Rekonfigurationsdurchatzes aufgeführt. Ein hoher Durchsatz an Rekonfigurationsdaten an der ICAP Schnittstelle ist erforderlich, um die zusätzliche Latenz durch die Rekonfiguration möglichst gering zu halten und damit die InterVFR und die IntraVFR realisieren zu können. In Abschnitt 5.3.2.3 wurde erwähnt, dass die maximal spezifizierte Frequenz mit der ICAP getaktet werden kann, auf einem V2P Device bei 50 MHz und auf V4 und V5 Devices bei 100 MHz liegt. Bei einer ICAP Input Datenweite IIW von 8 Bit auf V2P Devices und $IIW = 32$ Bit auf V4 und V5 Devices ergibt sich damit ein maximaler Durchsatz von Rekonfigurationsdaten von 50 MB/s auf V2P Devices bzw. 400 MB/s auf V4 und V5 Devices.
Durch die Verwendung des *busy* Signals, kann ICAP bei Frequenzen über den spezifizierten 50 MHz betrieben werden. Bei diesem Übertakten der ICAP Schnittstelle muß das *busy* Signal als handshaking signal verwendet werden, um sicher zu stellen, dass die Konfigurationsdaten korrekt an den internen Paketprozessor *CPP* weitergegeben werden. Auf V4 und V5 wird das *busy* Signal während der Konfiguration nicht getrieben, nur während des Readbacks. Eine maximale Frequenz, bei der ICAP noch funktioniert,

wird allerdings nicht angegeben. Daher wurde zunächst die maximal mögliche Frequenz auf V2P bestimmt. In Tabelle 6.7 sind die Ergebnisse zusammen gefasst. Für die Bestimmung der maximalen Frequenz wurden 15 verschiedene XUPV2P Boards verwendet. Die Bitströme wurden, wie in den anderen Designs auch, während der Initialisierungsphase in das DDR SDRAM kopiert, um schnelleren Zugriff zu erlauben. Um dieselben Bedingungen in allen Designs zu schaffen, wurde dieselbe Größe für die PRR gewählt. Das MFWR Feature wurde absichtlich deaktiviert, um alle Messungen mit einer Bitstromgröße von 73600 Byte durchzuführen. Die Bitstromdaten wurden in allen Designs mit einer Frequenz von $f_w = 100$ MHz in das FIFO des ICAP Controllers geschrieben. Ein Zähler in HW wird verwendet, um zyklengenau die Rekonfigurationsdauer zu messen. Der *busy* Faktor BF wurde für einen bestimmten Wert von f_r bestimmt, indem über alle Messungen gemittelt wurde. Bis zu einer Frequenz f_r von 150 MHz war die Rekonfiguration auf allen Devices fehlerfrei. Aufgrund von Fertigungstoleranzen erreichten einige der Devices eine Frequenz von 170 MHz. Dies wurde durch ein Zurücklesen der Konfigurationsdaten über die JTAG Schnittstelle bei spezifizierten, wesentlich geringeren Frequenzen verifiziert. Der zurückgelesene Bitstrom wurde anschließend mit dem initialen Bitstrom verglichen. Die Ergebnisse der Messungen sind in Tabelle 6.7 dargestellt. Der Taktzähler unterscheidet sich zwischen den einzelnen Messungen, je nachdem wie oft das *busy* Signal beobachtet werden konnte. An dieser Stelle sei darauf hingewiesen, dass sich das *busy* Signal nicht proportional zur Frequenz f_r verhält.

f_w $[MHz]$	f_r $[MHz]$	Takt- zähler	t_R $[\mu s]$	BF	TP_r $[MB/s]$
100	80	80800	1010.00	0.911	72.87
100	90	81462	905.13	0.903	81.31
100	100	80871	808.71	0.910	91.01
100	110	81054	736.85	0.908	99.88
100	120	81222	676.85	0.906	108.74
100	130	81692	628.40	0.901	117.12
100	140	81935	585.25	0.898	125.75
100	150	80703	538.02	0.912	136.80

Tabelle 6.7: ICAP Performance Ergebnisse auf 15 verschiedenen XUPV2P Boards bei verschiedenen Frequenzen f_r

In [CZS+08] konnte gezeigt werden, dass der maximale Durchsatz T_P für V2P bei 100 MHz erreicht werden konnte. Um auf V4 und V5 ebenfalls den maximal möglichen Durchsatz an Rekonfigurationsdaten zu erreichen, muß der Transfer vom Hauptspeicher zum ICAP Controller optimiert werden. Durch den modularen Aufbau kann der ICAP Controller ohne Modifikation über ein LIS PLB IPIF an den PLB, oder über ein LIS NP IPIF direkt an den MPMC angeschlossen werden. Für beide Möglichkeiten wurden zahlreiche Tests durchgeführt, um den maximalen Durchsatz zu bestimmen. Ein Durchsatz

jenseits der spezifizierten 400 MB/s [Xil10c] kann nur durch Anschluß des ICAP Controllers über ein LIS NP IPIF an den MPMC oder an einen 128-Bit breiten PLB und durch Verwendung von Address Pipelining erreicht werden. In Tabelle 6.8 sind die erzielten Ergebnisse dargestellt. Darin werden diese gegen die in der Literatur genannten Ergebnisse verglichen.

Device	f_w **[MHz]**	**Anschluß**	IIW **[Bit]**	f_r **[MHz]**	TP_r **[MB/s] gemessen**	TP_r **[MB/s] theoretisch**
V4 (Xilinx)	100	PLB	32	100	22.9	400
V4 [LD09]	90	cust. Link	32	90	29.0	360
V4 [SPA$^+$08]	100	cust. Link	32	144	219.31	576
V4 [MMT$^+$08]	100	OPB	32	100	350.0	400
V4 [LKLJ09]	100	PLB/NPI	32	100	371.4	400
V4 [HL09]	100	PLB	32	100	400	400
V4 [LPF09a]	100	PLB	32	100	400	400
V4	100	PLB/NPI	32	100	400	400
V5	100	PLB/NPI	32	100	400	400
V5	100	NPI	32	125	500	500
V5	150	NPI	32	150	600	600
V5	150	NPI	32	200	800	800
V5	150	NPI	32	250	1000	1000
V5	200	NPI	32	300	1200	1200

Tabelle 6.8: Durchsatz verschiedener ICAP Controller auf V4 und V5 bei verschiedenen Frequenzen f_r

Die Angaben ohne Literaturverweis entsprechen den eigenen Ergebnissen mit Ausnahme des Xilinx ICAP Controllers in der ersten Zeile von Tabelle 6.8. Dieser sehr niedrige Durchsatz kommt durch Anbindung des ICAP Controllers als Slave an den PLB zustande. Damit muß eine eingebettete CPU den Transfer von Bitstromdaten zum ICAP Controller regeln. Wie in Tabelle 6.8 zu erkennen ist, wurde in vielen Arbeiten versucht, den maximal spezifizierten Durchsatz von 400 MB/s zu erreichen. In [HL09] und [LPF09a] konnte dieser auch erreicht werden. In [HL09] aber nur auf Kosten eines sehr hohen BRAM Verbrauchs wie in Abschnitt 6.2.3 beschrieben. In der Literatur lagen zum Zeitpunkt dieser Arbeit keine Quellen zum Rekonfigurations-Durchsatz auf V5 vor. Wie auch auf V4, ist ICAP auf V5 bis zu einer Frequenz von 100 MHz spezifiziert [Xil10c]. Die Ergebnisse auf V2P, sowie die Ergebnisse von Shelburne et. al. [SPA$^+$08] lassen jedoch darauf schließen, dass ein Übertakten des ICAP Controllers möglich ist. Um die maximal mögliche Frequenz zu bestimmen, wurde f_r in Schritten von 50 MHz erhöht und die Konfiguration nach der in Abschnitt 5.3.2.3 beschriebenen Methode verifiziert. In Tabelle 6.8 sind die Frequenzen aufgeführt, mit denen die Konfiguration fehlerfrei durchgeführt werden konnte. Die maximale Frequenz f_r von 300 MHz wurde in diesem Fall durch Timingfehler begrenzt (eventuell kann die ICAP Schnittstelle mit höheren Frequenzen umgehen). Wie in Tabelle 6.8 zu erkennen ist, wurde ein maximaler Rekonfigurationsdurchsatz von 1,2 GB/s erreicht. Der erreichte Durchsatz ist damit 3 mal höher als der laut Spezifikation

mögliche Durchsatz von 400 MB/s und somit auch 3 mal höher als alle in der Literatur aufgeführten Ergebnisse, was ein wesentliches Ergebnis dieser Arbeit darstellt.

6.4 Verlustleistung in DPR Systemen

Um einen fairen Vergleich zwischen DPR und nicht rekonfigurierbarem Design anstellen zu können, muß nicht nur der Anteil an Verlustleistung berücksichtigt werden, der während der DPR hinzukommt, sondern auch der Anteil, der durch den DPR Overhead verursacht wird. Dieser Abschnitt ist daher wie folgt unterteilt: zunächst wird der Messaufbau in Abschnitt 6.4.1 beschrieben. In Abschnitt 6.4.2 wird dann isoliert betrachtet, was der DPR Overhead im Vergleich zu Designs ohne DPR Overhead zur Gesamtverlustleistung beiträgt. Abschnitt 6.4.3 befasst sich mit dem Anteil, der durch den Vorgang der DPR selbst erzeugt wird. Im Anschluß wird als Worst Case Abschätzung die Verlustleistung im IntraVFR Design mit mehreren Rekonfigurationen pro Video Frame betrachtet. Die Ergebnisse hierzu sind in Abschnitt 6.4.4 dargestellt. In Abschnitt 6.4.5 wird dann die Optimierung hinsichtlich Verlustleistung durch Einsatz des in Abschnitt 5.3.3.2 vorgestellten dynamischen Power Managements erläutert.

6.4.1 Versuchsaufbau zur Verlustleistungsmessung in DPR Systemen

Wie in Abschnitt 5.3.3 erwähnt, beeinflußt DPR direkt das Konfigurations-Memory des FPGAs, das von der Core Spannung des FPGAs versorgt wird. Um die Core Verlustleistung des FPGA zu bestimmen, müssen sowohl die am FPGA anliegende Spannung, als auch die Stromaufnahme bekannt sein. Die Core Versorgungsspannung V_{Core} beträgt 1,0 V. Zur Messung der Stromaufnahme, kann ein Shunt Widerstand R_{Shunt} eingesetzt werden. Da auf dem verwendeten ML507 Board mit einem XC5VFX70T Device werksseitig kein Shunt Widerstand vorhanden ist, wurden alle Verlustleistungsmessungen auf dem ML510 Board mit einem XC5VFX130T Device durchgeführt und das IntraVFR Design aus diesem Grund auf diese Plattform portiert. Auf dem ML510 Board ist bereits ein Shunt Widerstand von $2,2\ m\Omega$ im Spannungsversorgungspfad des Cores integriert. Dieser Shunt Widerstand wird eingesetzt, um den Spannungsabfall zu messen, da dieser proportional zur Stromaufnahme ist. Die Core Versorgungsspannung sowie der Spannungsabfall am Shunt können dann mit entsprechenden Messverstärkern und Voltmetern gemessen werden. In Abbildung 6.3 ist die Messschaltung dargestellt.
Da bei einem Shunt Widerstand R_{Shunt} von $2,2\ m\Omega$ nur ein sehr geringer Spannungsabfall im mV Bereich gemessen werden konnte, der zusätzlich durch Rauschen verfälscht wurde, ist dieser durch einen Widerstand $R_{Shunt} = 22\ m\Omega$ desselben Typs und Herstellers ersetzt worden. Ein größerer Shunt Widerstand sollte jedoch nicht verwendet werden, da der Spannungsabfall am Widerstand nicht größer als 50 mV sein darf. In [Xil09e] wird eine Toleranz von $1\ V\ \pm\ 50\ mV$ als zulässig angegeben. Gleichung 6.2 dient zur Berechnung der Stromaufnahme des Cores.

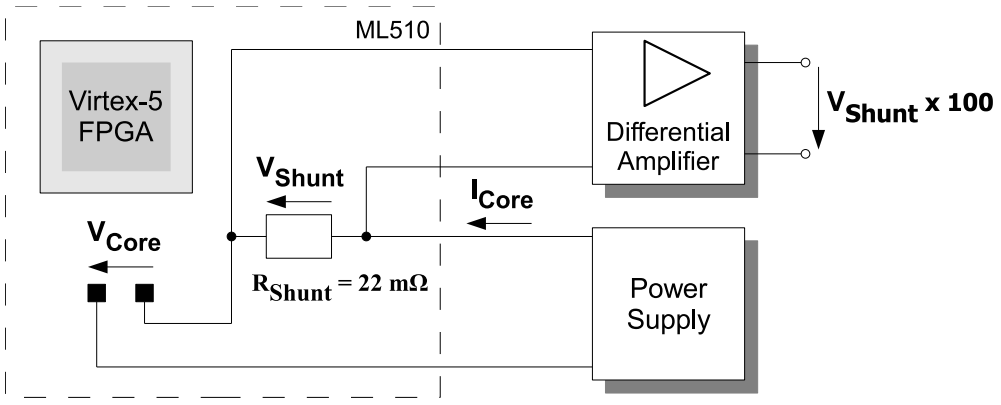


Abbildung 6.3: ML510 Versuchsaufbau

$$I_{Core} = \frac{V_{Shunt}}{R_{Shunt}} = \frac{V_{Shunt}}{22\,m\Omega} \tag{6.2}$$

Die Verlustleistung, die während einer DPR entsteht, wurde mit einem Tektronix TDS 5104B Oszilloskop gemessen. Der Spannungsabfall am Shunt wurde mit einem Tektronix AM502 Differenzverstärker um den Faktor 100 verstärkt. Die Gesamtverlustleistung P_{Gesamt} ergibt sich aus der Summe von Core Verlustleistung P_{Core} und I/O Verlustleistung P_{IO}. Genaue Messungen von P_{IO} sind auf dem ML510 jedoch nicht möglich, da an der Spannungsversorgung der I/O Pins weitere Komponenten angeschlossen sind, die die Messung verfälschen. Zudem fehlt werksseitig ein Shunt Widerstand im Spannungsversorgungspfad der I/O Pins. Von außen betrachtet, besteht kein Unterschied zwischen nicht rekonfigurierbarem und DPR Design. Die Bilddaten, sowie die partiellen Bitströme werden im Hauptspeicher abgelegt und über dieselben Pins auf das FPGA übertragen. Wie allerdings in [Hüb07] gezeigt werden konnte, wird die I/O Verlustleistung während einer DPR ansteigen. Ähnlich wie bei den Ressourcen, wird auch bei der Verlustleistung versucht, den Beitrag an Verlustleistung, den der DPR Overhead Core Verlustleistung P_{Core} beisteuert, so gering wie möglich zu halten. Daher wird im folgenden Abschnitt beschrieben, um welchen Beitrag der DPR Overhead P_{Core} erhöht.

6.4.2 Verlustleistung durch DPR Overhead

Die Leistungsaufnahme des IntraVFR Designs, direkt nach der initialen Konfiguration, ohne dass Bildverarbeitung (Berechnung des Optischen Flusses) oder Rekonfiguration im System stattfindet, bildet die Basis der Messungen in diesem Abschnitt. Die Bildverarbeitung im nicht rekonfigurierbaren und im DPR System ist in beiden Fällen exakt gleich und muß deshalb bei Verlustleistungsuntersuchungen, bedingt durch DPR, nicht

weiter betrachtet werden. Um die Verlustleistung eines DPR Systems mit der eines nicht rekonfigurierbaren Systems zu vergleichen, müssen alle Unterschiede betrachtet werden, die einen Einfluß auf die Verlustleistung haben. Der erste Unterschied besteht im DPR Overhead (siehe Abschnitt 5.3.1). Dieser liefert einen Beitrag zur Core Verlustleistung P_{Core}. Mit Hilfe der Gleichung 6.3 kann P_{Core} berechnet werden.

$$P_{Core} = (V_{Core}) \times I_{Core} \tag{6.3}$$

Die am FPGA anliegende Versorgungsspannung V_{Core} ist die Gesamtversorgungsspannung V_{Supply} von 1.0 V, reduziert um den Spannungsabfall am Shunt Widerstand V_{Shunt} ($V_{Core} = V_{Supply} - V_{Shunt}$). Da V_{Core} im Gegensatz zu V_{Shunt} (max. 50 mV) jedoch wesentlich größer ist und qualitative Aussagen getroffen werden, wird im folgenden V_{Shunt} vernachlässigt. V_{Core} wird bei allen folgenden Messungen zu 1,0 V angenommen. Alle Designs, die für die Messreihen verwendet wurden, sind in Abbildung 6.4 schematisch dargestellt.

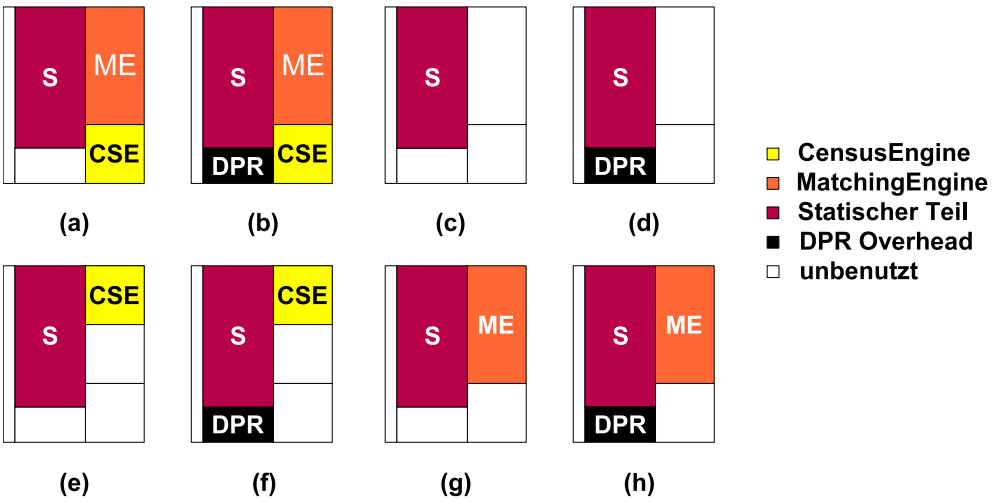


Abbildung 6.4: Layout der Messsysteme mit verschiedenen Komponenten

Fall (a) und Fall (b) zeigen das statische IntraVFR Design (mit *CSE* und *ME*) ohne bzw. mit DPR Overhead. Die Fälle (c) und (d) zeigen den statischen Teil ohne HWAs (bzw. mit geladenem Blank Modul) einmal ohne bzw. mit DPR Overhead. In (e) und (f) sind die beiden Designs dargestellt, die neben dem statischen Teil die CensusEngine beinhalten, einmal ohne bzw. mit DPR Overhead. Die *CSE* ist hierbei innerhalb der PRR platziert, deren Fläche durch das größte rekonfigurierbare Modul im Design (hier: *ME*) bestimmt wird. Analog dazu werden in (g) und (h) die Fälle dargestellt, wenn nur die *ME* statt der *CSE* auf dem Device ist.
Durch die Messungen der verschiedenen Designs einmal ohne und einmal mit DPR Over-

head wurde versucht, den Anteil der Verlustleistung zu bestimmen, der durch Einfügen dieser Komponenten entsteht. Wie bereits in Abschnitt 5.3.1 beschrieben, zählen zum DPR Overhead der ICAP Controller, das *RIF*, das LIS NP IPIF, sowie ein zusätzlicher Port am MPMC. In Abschnitt 6.3 wurde gezeigt, dass ICAP auf V5 bis zu einer Frequenz von 300 MHz fehlerfrei funktioniert. Je höher allerdings die Taktung der zum DPR Overhead gehörenden Komponenten, desto höher auch die dynamische Verlustleistung. Um möglichst aussagekräftige (und frequenzabhängige) Ergebnisse für den Teil an Verlustleistung zu bekommen, der durch den DPR Overhead verursacht wird, wurden verschiedene Messungen durchgeführt. Für die in Abbildung 6.4 aufgeführten Fälle (b), (d), (f) und (h) wurden jeweils 9 verschiedene Systeme aufgesetzt. Die Frequenz f_R wurde in Schritten von 25 MHz von 100 MHz bis 300 MHz erhöht, was auf 9 unterschiedliche Systeme führt. Mit Ausnahme der Lesefrequenz f_r gibt es in diesen 9 Systemen jedoch keine Änderungen. Die Systeme (a), (c), (e) und (g) dienen bei diesen Messungen als Referenz. Die Messungen an den verschiedenen Systemen sind in Tabelle 6.9 dargestellt. Der Zuwachs an Verlustleistung durch DPR Overhead in Tabelle 6.9 rührt hauptsächlich von der dynamischen Verlustleistung her, wie an der starken Frequenzabhängigkeit zu erkennen ist. Anhand der Gleichung für die dynamische Verlustleistung erwartet man hierbei einen linearen Anstieg, der auch in Tabelle 6.9 zu erkennen ist. Dies ist jedoch nur der Fall, wenn sich die Verdrahtung zwischen den einzelnen Messungen nicht ändert. Bei geändertem Routing, steigt die Verlustleistung keineswegs linear und nimmt in bestimmten Fällen bei steigender f_R sogar ab, wie in [Alt09b] gezeigt werden konnte. Wie in Abschnitt 5.3.3, [TKR+06] und [Don07] erwähnt wird, ist für etwa 60% der dynamischen Verlustleistung die Verdrahtung (Routing) verantwortlich. Um also aussagekräftige Ergebnisse zu erhalten, ist auf die gleiche Verdrahtung zwischen den verschiedenen Designs zu achten. Da der einzige Unterschied zwischen den Designs mit verschiedenen Frequenzen f_R im Teilerverhältnis des Clock Generators besteht, stellt sich die berechtigte Frage nach einer Möglichkeit, das Teilerverhältnis, unter Beibehaltung der Verdrahtung, manuell zu ändern. Dies ist mit Hilfe des FPGA Editors möglich. Durch Laden eines NCD Files in den FPGA Editor können die Einstellungen der entsprechenden PLL und damit auch insbesondere das Teilerverhältnis, verändert werden. Damit lassen sich alle Designs mit unterschiedlichen ICAP Frequenzen erzeugen, ohne dass die Verdrahtung modifiziert wird. Die verschiedenen Designs, aus denen die in Tabelle 6.9 dargestellten Ergebnisse gewonnen wurden, sind auf diese Art erzeugt. Die Zeilen in Tabelle 6.9 in denen keine Frequenz f_R angegeben ist, repräsentieren die Designs ohne DPR Overhead. Aus Gründen der Übersichtlichkeit wird jeweils in der ersten Spalte das zur Messung verwendete System aus Abbildung 6.4 angegeben. In der zweiten Spalte ist die Frequenz f_R aufgeführt, die verwendet wurde, um Konfigurationsdaten aus dem FIFO des ICAP Controllers zu lesen und der ICAP Schnittstelle zuzuführen. In der dritten Spalte ist der gemessene Spannungsabfall am Shunt Widerstand aufgeführt. Der Core Strom I_{Core} in der vierten Spalte wird nach Gleichung 6.2 berechnet. Analog dazu berechnet sich die Core Verlustleistung P_{Core} nach Gleichung 6.3. In der fünften Spalte ist die Verlustleistungs-Differenz

zwischen dem Design ohne und mit DPR Overhead aufgeführt. Die Ergebnisse in dieser Spalte zeigen also an, welchen Anteil der DPR Overhead zur Gesamtverlustleistung des Systems beiträgt im Vergleich zum nicht rekonfigurierbaren Design. Im Normalfall ist immer derselbe Zuwachs an Verlustleistung zu erwarten, der durch den DPR Overhead dem System hinzugefügt wird. Die unterschiedlichen Werte in der sechsten Spalte von Tabelle 6.9 kommen dadurch zu Stande, dass die Designs ohne DPR Overhead (a), (c), (e) und (g) eine andere Platzierung und Verdrahtung besitzen, wie die Designs (b), (d), (f) und (h). Die gleiche Verdrahtung zwischen den Designs mit DPR Overhead führt bei linear steigender Frequenz annähernd zum selben Zuwachs an dynamischer Verlustleistung, wie in der letzten Spalte von Tabelle 6.9 zu erkennen ist. Geringe Abweichungen resultieren zwar aus Messungenauigkeiten, jedoch ist der Zuwachs an Verlustleistung augenscheinlich linear. Vergleicht man die Ergebnisse der Systeme mit DPR Overhead mit denen der Systeme ohne DPR Overhead, kann man einen deutlichen Zuwachs an Verlustleistung feststellen.
In allen Fällen lässt sich erkennen, dass P_{Core} zwischen etwa 10% (bei 100 MHz) und 20% (bei 300 MHz) ansteigt, wenn der DPR Overhead dem System hinzugefügt wird. Dieser Zuwachs soll später durch das Entfernen nicht benötigter IP Cores vom FPGA (durch DPR) kompensiert werden. Im folgenden Abschnitt 6.4.3 wird nun der Anteil an Verlustleistung betrachtet, der durch die DPR selbst entsteht.

System	ICAP Frequenz f_R [MHz]	Shunt Voltage [mV]	Core Current I_{Core} [A]	Core Power P_{Core} [W]	Zuwachs durch DPR Overhead [mW]	Δ Power Difference [mW]
(a)	-	39,3	1,79	1,79	-	-
(b)	100	42,6	1,94	1,94	150,00	-
(b)	125	43,1	1,96	1,96	172,73	22,73
(b)	150	43,5	1,98	1,98	190,91	18,18
(b)	175	44,0	2,00	2,00	213,64	22,73
(b)	200	44,4	2,02	2,02	231,82	18,18
(b)	225	44,7	2,03	2,03	245,45	13,64
(b)	250	45,0	2,05	2,05	259,09	13,64
(b)	275	45,4	2,06	2,06	277,27	18,18
(b)	300	45,7	2,08	2,08	290,91	13,64
(c)	-	33,6	1,53	1,53	-	-
(d)	100	34,9	1,59	1,59	59,09	-
(d)	125	35,3	1,60	1,60	77,27	18,18
(d)	150	35,6	1,62	1,62	90,91	13,64
(d)	175	35,9	1,63	1,63	104,55	13,64
(d)	200	36,3	1,65	1,65	122,73	18,18
(d)	225	36,6	1,66	1,66	136,36	13,64
(d)	250	36,9	1,68	1,68	150,00	13,64
(d)	275	37,2	1,69	1,69	163,64	13,64
(d)	300	37,5	1,70	1,70	177,27	13,64
(e)	-	34,1	1,55	1,55	-	-
(f)	100	38,7	1,76	1,76	209,09	-
(f)	125	39,1	1,78	1,78	227,27	18,18
(f)	150	39,4	1,79	1,79	240,91	13,64
(f)	175	39,8	1,81	1,81	259,09	18,18
(f)	200	40,2	1,83	1,83	277,27	18,18
(f)	225	40,5	1,84	1,84	290,91	13,64
(f)	250	40,8	1,85	1,85	304,55	13,64
(f)	275	41,1	1,87	1,87	318,18	13,64
(f)	300	41,6	1,89	1,89	340,91	22,73
(g)	-	36,8	1,67	1,67	-	-
(h)	100	40,0	1,82	1,82	145,45	-
(h)	125	40,4	1,84	1,84	163,64	18,18
(h)	150	40,7	1,85	1,85	177,27	13,64
(h)	175	41,1	1,87	1,87	195,45	18,18
(h)	200	41,4	1,88	1,88	209,09	13,64
(h)	225	41,8	1,90	1,90	227,27	18,18
(h)	250	42,1	1,91	1,91	240,91	13,64
(h)	275	42,5	1,93	1,93	259,09	18,18
(h)	300	42,9	1,95	1,95	277,27	18,18

Tabelle 6.9: Verlustleistungsmessungen für verschiedene Systeme mit und ohne DPR Overhead bei verschiedenen Frequenzen f_R

6.4.3 Verlustleistung während einer DPR

Der in Abschnitt 6.4.2 beschriebene Anteil an Verlustleistung durch DPR Overhead ist ein permanentes Phänomen. Durch die Verwendung dieser zusätzlichen Komponenten ändert sich sowohl die statische als auch dynamische Verlustleistung um einen gewissen Prozentsatz. Der Anteil an Verlustleistung, der durch den Vorgang der DPR erzeugt wird ist hingegen ein temporäres Phänomen. In Designs, die nur sehr selten (ein paar mal pro Stunde/Tag) dynamisch rekonfiguriert werden, spielt die zusätzliche dynamische Verlustleistung, die durch DPR entsteht praktisch keine Rolle. Im IntraVFR Design des Optischen Flusses ist der Beitrag zur Verlustleistung allerdings nicht mehr zu vernachlässigen. Verlustleistungsmessungen in DPR Systemen, in denen sehr oft rekonfiguriert wird, sind alles andere als einfach, da es sich um ein hoch dynamisches System handelt. Rekonfigurationszeiten liegen oft im μs Bereich, was die Messung deutlich erschwert.

Um den Verlauf der Verlustleistung während einer DPR dennoch zu analysieren, wurden die folgenden Experimente durchgeführt. Um sicher zu stellen, dass die PRR vor einer Rekonfiguration komplett leer ist und damit eventuell fehlerhafte Konfigurationen auszuschließen, kann ein Blank Bitstrom verwendet werden. Wie in Abschnitt 5.3.3.1 erwähnt, wird erwartet, dass das Entfernen eines HWAs durch einen Blank Bitstrom einen positiven Einfluß auf die Verlustleistung hat. Daher wurde untersucht, ob es aus Verlustleistungsgesichtspunkten ebenfalls Sinn macht das alte PRM zu löschen, anstatt direkt das neue PRM per Rekonfiguration zu laden. Das Überschreiben einer alten Konfiguration mit einer neuen, ohne vorheriges Löschen der alten Konfiguration, wird als Scrubbing bezeichnet. Hierfür wird wiederum das IntraVFR Design zur Berechnung des Optischen Flusses verwendet. Im ersten Versuch wird die *CSE* direkt durch die *ME* und umgekehrt ersetzt. Im zweiten Versuch wird dann das alte Modul durch einen Blank Bitstrom gelöscht, bevor das eigentliche Modul geladen wird.

Vor der Durchführung der Experimente wurden die folgenden Überlegungen angestellt: die statische Verlustleistung einer einzelnen SRAM Zelle ist weitgehend unabhängig davon, ob eine 0 oder 1 gespeichert ist. Der Unterschied ist verschwindend gering [Ras07]. Daher ändert sich die statische Verlustleistung des Konfigurations-Layers auch nicht (oder kaum) bei einer DPR. Allerdings ist der Konfigurations-Layer direkt mit dem funktionalen Layer verbunden, wie bereits in Abschnitt 2.2.2 beschrieben wurde. Ein Beispiel dafür ist in Abbildung 6.5 dargestellt.

Eine SRAM Zelle aus dem Konfigurations-Layer steuert den Select Eingang eines 2-zu-1 Multiplexers im funktionalen Layer. Dabei wird der Aufbau eines 2-zu-1 Multiplexers in [TL03] wie folgt dargestellt. Der Multiplexer besteht aus zwei NMOS Transistoren N_2 und N_3. Die SRAM Zelle ist dabei zum einen direkt mit dem Gate von N_3 und zum anderen mit dem Eingang eines Inverters (bestehend aus NMOS Transistor N_1 und PMOS Transistor P_2) verbunden. Der Ausgang des Inverters ist mit dem Gate von N_2 verbunden. Somit kann über die Konfiguration der SRAM Zelle genau einer der beiden Multiplexereingänge I_1 und I_2 auf den Ausgang O durchgeschaltet werden. Abhängig

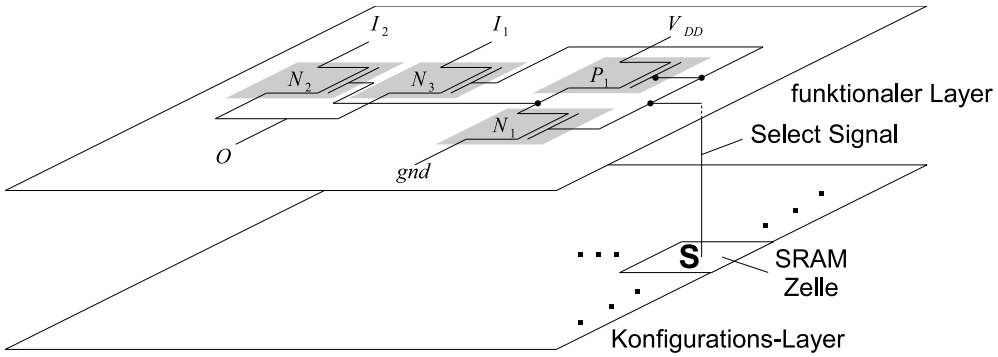
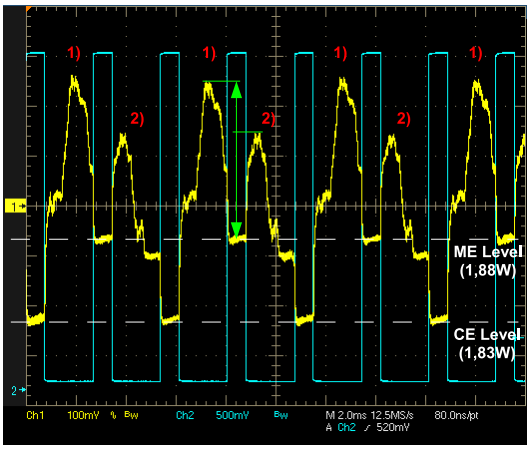
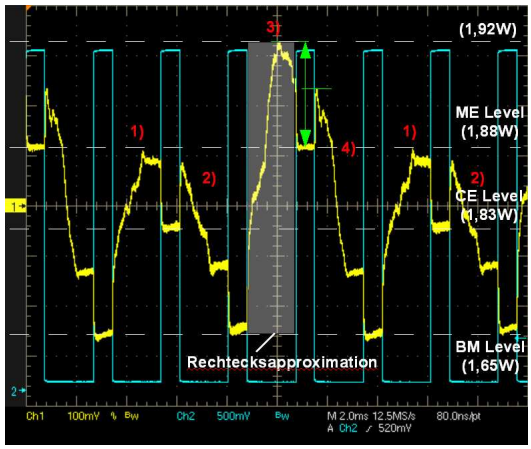


Abbildung 6.5: Realisierung eines 2-zu-1 Multiplexers im funktionalen Layer. Das Select-Signal des Multiplexers ist mit einer SRAM Speicherzelle verbunden

vom Inhalt der SRAM Zelle, der direkte Auswirkung auf die Werte an den Gates von N_1, N_2, N_3 und P_1 hat und den Werten und Drain bzw. Source dieser Transistoren, kann die statische Verlustleistung signifikant variieren wie in Abbildung 2.21 dargestellt ist. Eine Änderung des Konfigurations-Layers führt also unter Umständen zu einer erheblichen Änderung der statischen Verlustleistung im funktionalen Layer. Da die Rekonfiguration frameweise erfolgt(s. Abschnitt 2.2.5), sollte sich beim Schreiben jedes Frames auch ein anderer Wert für die statische Verlustleistung ergeben. Ähnliches gilt auch für eine Realisierung des 2:1 Multiplexers mit einem NMOS und einem PMOS Transistor.
In Abbildung 6.6 ist der Spannungsabfall am Shunt Widerstand, der wie bereits erwähnt proportional zur Stromaufnahme ist, während einer DPR zu sehen. Der wellenförmige Verlauf in 6.6(a) und 6.6(b) weist tatsächlich darauf hin, dass sich die statische Verlustleistung bei jedem geschriebenen Frame ändert.
Abbildung 6.6(a) zeigt den Verlauf des Scrubbing Prozesses, wenn direkt zwischen den beiden HWAs rekonfiguriert wird. In Abbildung 6.6(b) hingegen ist der Verlauf des Spannungsabfalls zu sehen, wenn ein Blank Bitstrom zum Löschen der PRR verwendet wird. Wenn nicht rekonfiguriert wird, bleibt der Spannungsabfall annähernd konstant und ist nur von den gegenwärtig auf dem Device konfigurierten Modulen abhängig. So lassen sich in Abbildung 6.6 verschiedene Spannungslevel erkennen, wenn keine Rekonfiguration aktiv ist. Abhängig von dem gerade konfigurierten Modul wird zwischen dem Level für das Blank Modul (BM-Level), die *CSE* (CSE-Level) und die *ME* (ME-Level) unterschieden. ME-Level und CSE-Level sind in beiden Abbildungen gleich. Die Werte der BM-, CE und ME-Level entsprechen den Werten in Tabelle 6.9 bei $f_R = 200$ MHz.
Der höchste Spannungsabfall am Shunt Widerstand tritt in Abschnitt 3) bei der Rekonfiguration vom BM zur *ME* auf, wie in Abbildung 6.6(b) dargestellt. Betrachtet wird in diesem Fall der Bereich vom BM-Level bis zur höchsten Spannungsspitze. Der Unterschied beträgt 6mV, was umgerechnet an dieser Stelle ungefähr einer Verlustleistung



1) Von CensusEngine zu MatchingEngine
2) Von MatchingEngine zu CensusEngine



1) Von Blank Modul zu CensusEngine
2) Von CensusEngine zu Blank Modul
3) Von Blank Modul zu MatchingEngine
4) Von MatchingEngine zu Blank Modul

(a) Rekonfiguration von *CSE* in *ME* (Scrubbing)

(b) Rekonfiguration von *CSE* in *ME* mit vorherigem Löschen der PRR durch einen Blank Bitstrom

Abbildung 6.6: Stromaufnahme während einer DPR

von 270 mW entspricht. Addiert man diesen Betrag zur Verlustleistung, wenn das Blank Modul geladen und der DPR Overhead bei 200 MHz getaktet ist (1,65W), ergibt sich an der höchsten Spannungsspitze ein Wert von 1,92W.
Um eine Abschätzung der Verlustleistung (Durchschnittswert) während einer DPR zu erhalten, kann für die Dauer des Rekonfigurationsvorgangs die so genannte Rekonfigurationsenergie W_{Core} nach Gleichung 6.4 bestimmt werden.

$$W_{Core} = \int_{t_1}^{t_2} P(x)\,dx \tag{6.4}$$

W_{Core} entspricht dabei der Fläche unterhalb der Kurve in Abbildung 6.6(a) und Abbildung 6.6(b). Teilt man die Rekonfigurationsenergie anschließend durch die Dauer der Rekonfiguration T_R, erhält man die durchschnittliche Verlustleistung während einer DPR. Da sich die genaue Bestimmung der Rekonfigurationsernergie relativ schwierig gestaltet und da qualitative Aussagen getroffen werden, wird die Rekonfigurationsenergie im Folgenden durch ein Rechteck approximiert, wie in Abbildung 6.6(b) dargestellt. Damit kann für die Dauer einer DPR ein konstanter Wert angegeben werden. Für die Höhe des Rechtecks wird die maximal gemessene Verlustleistung von 1,92 W als sinnvolle Worst Case Abschätzung verwendet. Die tatsächliche Rekonfigurationsenergie ist natürlich wesentlich geringer. Obwohl bei den anderen Rekonfigurationen, die gemessenen Spannungsspitzen

und damit auch die Rekonfigurationsenergie ebenfalls augenscheinlich kleiner sind, wird ebenfalls dieser konstante Wert zur Abschätzung verwendet. Dass bei der Rekonfiguration von einem Blank Modul zur *ME* der höchste Spannungsabfall gemessen wurde, läßt sich durch die Tatsache erklären, dass große Teile des Konfigurations-Memories geändert werden müssen, um einen HWA in eine leere PRR zu konfigurieren. Daher ist nicht nur von Bedeutung welches Modul vor einer Rekonfiguration geladen war, sondern auch wie viele Teile des Konfigurations-Memories umgeladen werden müssen. Dies läßt sich bei genauerer Betrachtung auch mit Hilfe von Abbildung 6.6(b) belegen. In Abschnitt 2) und 4) wird derselbe Blank Bitstrom verwendet, um ein Modul aus der PRR zu löschen. Die Verläufe unterscheiden sich jedoch deutlich.
Des Weiteren lässt sich in Abbildung 6.6 beobachten, dass beim Scrubbing Prozess quantitativ höhere Spannungsspitzen erreicht werden, was durch die Pfeile gekennzeichnet ist. Dies weist auf eine höhere Rekonfigurationsenergie hin. Allerdings ist bei der Verwendung des Blank Bitstroms eine zweite Rekonfiguration notwendig, was anhand der in Abbildung 6.6 gezeigten Kurvenverläufe in jedem Fall zu einer höheren Verlustleistung P_{Core} führt.
Anhand dieser Ergebnisse ist ersichtlich, dass es nicht sinnvoll ist einen Blank Bitstrom zur Einsparung von Verlustleistung zu verwenden, um ein altes Modul vor der eigentlichen Rekonfiguration aus der PRR zu löschen. Alternativ kann jedoch Clock-Gating eingesetzt werden, wie in Abschnitt 6.4.5 beschrieben.
An dieser Stelle sei auf den sehr niedrigen BM-Level im Vergleich zu CSE-Level und ME-Level hingewiesen. Die Konfiguration mit einem Blank Bitstrom könnte daher bei längerer Inaktivität eines Moduls also zu einer reduzierten Verlustleistung führen. Ob sich der Aufwand einer Rekonfiguration im Vergleich mit anderen Methoden wie beispielsweise Clock Gatiing (CG) lohnt, wird in Abschnitt 6.4.5 beschrieben.

6.4.4 Verlustleistung im IntraVFR System

Um einen fairen Vergleich zwischen einem DPR System und einem nicht rekonfigurierbaren System zu erhalten, müssen alle bisher aufgeführten Aspekte berücksichtigt werden. Für das nicht rekonfigurierbare Design bleibt die Core Verlustleistung für die Zeitdauer eines Videoframes annähernd konstant und beträgt 1,79 W. Im rekonfigurierbaren System hingegen ist die Verlustleistung stark abhängig davon, welches Modul gerade geladen wurde und mit welcher Frequenz rekonfiguriert wird. Daher bietet es sich an, die Worst Case Verlustleistung abschnittsweise für die Module und die Rekonfiguration anzugeben. Die gemessenen Verlustleistungen für die Rekonfiguration, sowie die Verlustleistungen bei geladenem Blank Modul, *CSE* und *ME*, sind in Tabelle 6.10 nochmals zusammengefasst. Der DPR Overhead wurde mit einer Frequenz von $f_r = f_W = 200$ Mhz getaktet, was einem Rekonfigurationsdurchsatz von 800 MB/s entspricht.
Die Werte in Tabelle 6.10 entsprechen denen aus Tabelle 6.9. Wie bereits erwähnt, ist die Verlustleistung, die durch die Berechnung des Optischen Flusses in den HWAs und

Verlustleistung ohne DPR:	
Blank Module	1.65 W
CSE	1.83 W
ME	1.88 W
Worst case Annahme während DPR:	
DPR	1.92 W

Tabelle 6.10: Verlustleistung des IntraVFR Sytems des Optischen Flusses

auf dem PPC entsteht, im DPR und im nicht rekonfigurierbaren Design gleich und wird daher nicht weiter betrachtet. Für eine Rekonfiguration wird ein konstanter Worst Case Wert von 1,92 W angenommen (s. Abschnitt 6.4.3). Wie bereits gezeigt wurde, ist der Zuwachs an Verlustleistung während einer DPR weitaus geringer. In Abbildung 6.7 ist eine Zeitachse dargestellt, die den Ablauf während eines Video-Frames (bei 31 fps) charakterisiert. Jedem Zeitabschnitt wird hierbei die gemessene bzw. approximierte Verlustleistung aus Tabelle 6.10 zugewiesen.

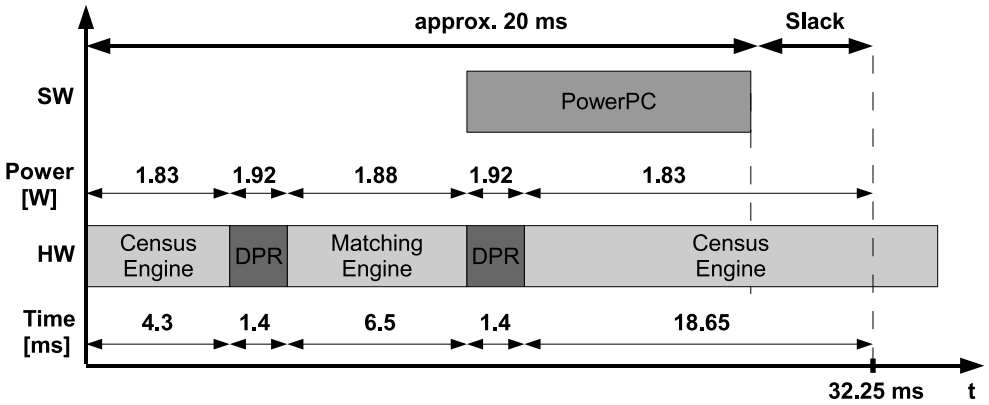


Abbildung 6.7: Zeitachse mit auftretender Verlustleistung im IntraVFR Design

Die partiellen Bitströme für *CSE* und *ME* haben eine Größe von 1,12 MB, was bei einem Durchsatz an Rekonfigurationsdaten von 800 MB/s einer gemessenen Rekonfigurationszeit von 1,4 ms entspricht. Die Ausführungszeiten für die HWAs und Rekonfiguration entsprechen denen in Abschnitt 6.1 und Abschnitt 6.3. Um einen durchschnittlichen Leistungsverbrauch angeben zu können und diesen mit dem des nicht rekonfigurierbaren Designs vergleichen zu können, werden die einzelnen Verlustleistungen mit den entsprechenden Zeitabschnitten gewichtet. Dies entspricht wiederum der Rekonfigurationsenergie für

den bestimmten Zeitabschnitt. Die Summation aller Einzelenergien und die anschließende Division durch 32,25 ms (Framedauer bei 31 fps) resultiert in der durchschnittlichen Verlustleistung während eines Frames. In Tabelle 6.11 sind nochmals alle Einzelenergien, die einen Beitrag zur Verlustleistung liefern dargestellt.

Verlustleistung in unterschiedlichen Sektionen:	
CSE	(4.3 ms + 18.65 ms) x 1.83 W
ME	6.5 ms x 1.88 W
DPR	2 x 1.4 ms x 1.92 W
Durchschnittliche Verlustleistung für die Dauer eines Video-Frames:	
1.85 W	

Tabelle 6.11: Durchschnittliche Verlustleistung im IntraVFR System

Die durchschnittliche Worst Case Gesamtverlustleistung von 1,85 W ist demnach höher, als die des nicht rekonfigurierbaren Designs von 1,79 W. An dieser Stelle sei jedoch darauf hingewiesen, dass es sich bei dem Austausch von 2 Modulen um den denkbar schlechtesten Fall für die Rekonfiguration handelt. Bei 3 Modulen und mehr ist eine Verlustleistungseinsparung zu erwarten.
In Abbildung 6.7 ist erkennbar, dass die *CSE* die meiste Zeit die Ressourcen der PRR belegt. Insgesamt (4.3 + 18.65) ms = 22,95 ms. Die *CSE* ist also während der Census Transformation und der Verarbeitungszeit der Features auf dem PPC auf dem Device präsent. Mit Beginn des neuen Videoframes kann also sofort mit der Verarbeitung der Pixeldaten begonnen werden. Während der Verarbeitungszeit auf dem PPC liefert jedoch die *CSE* einen Beitrag zur Verlustleistung, obwohl sie zu diesem Zeitpunkt nicht aktiv ist. Hier besteht zusätzliches Optimierungspotential. In Abschnitt 6.4.3 wurde gezeigt, dass durch das Laden eines Blank Bitstroms Verlustleistung eingespart werden kann. Daher ist in Abbildung 6.8 ein alternativer Ablauf dargestellt.
Anstatt direkt nach der Verarbeitungszeit der *ME* zurück in die *CSE* zu konfigurieren, wird ein Blank Bitstrom geladen. Die PRR bleibt somit während der Verarbeitungszeit des PPCs zunächst leer. Erst kurz bevor der neue Video Frame zu erwarten ist, wird in die *CSE* zurück konfiguriert. Durch die nicht benutzten Ressourcen während der Verarbeitung in SW, läßt sich ein Teil der Verlustleistung P_{core} einsparen, wie in Tabelle 6.12 zu sehen ist.
Im denkbar schlechtesten Fall (2 austauschbare Module) ist die Worst Case Abschätzung für die Verlustleistung von 1,76 W sogar minimal besser, als die des nicht rekonfigurierbaren Designs mit 1,79 W. Die einzige Änderung, die vorzunehmen ist, besteht in der Änderung des Rekonfigurationsablaufs, wie in Abbildung 6.8 zu sehen ist. Jedoch resultiert das Laden eines zusätzlichen partiellen Bitstroms in einer erhöhten Verlustleistung der I/Os, wie anhand der Ergebnisse in [Hüb07] zu erkennen ist. Insgesamt sollten so we-

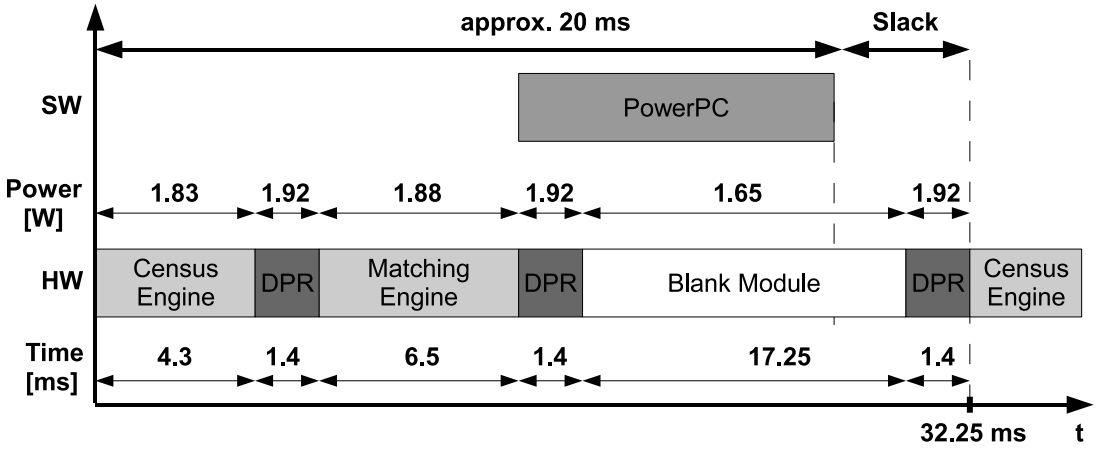


Abbildung 6.8: Zeitachse mit optimierter Verlustleistung im IntraVFR Design

Verlustleistung in unterschiedlichen Sektionen:	
CSE	4.3 ms x 1.83 W
ME	6.5 ms x 1.88 W
Blank Module	17.25 ms x 1.65 W
DPR	3 x 1.4 ms x 1.92 W
Durchschnittliche Verlustleistung für die Dauer eines Video-Frames:	
1.76 W	

Tabelle 6.12: Optimierte durchschnittliche Verlustleistung im IntraVFR System

nig Rekonfigurationen wie möglich durchgeführt werden, wenn die partiellen Bitströme aus einem externen Speicher geladen werden müssen. Das Laden eines Blank Bitstroms im IntraVFR System lohnt sich im Hinblick auf die Gesamtverlustleistung deshalb nicht. Durch die Verwendung von Clock-Gating (CG) besteht dennoch die Möglichkeit dynamische Verlustleistung im IntraVFR Design einzusparen. Die Ergebnisse der in Abschnitt 5.3.3.1 beschriebenen Konzepte werden im folgenden Abschnitt 6.4.5 präsentiert.

6.4.5 Ergebnisse des Dynamischen Power Managements in DPR Systemen

In Abschnitt 5.3.3.2 wurde ein Konzept zum dynamischen Power Management in DPR Systemen vorgestellt. Dabei wird versucht inaktive IP Cores vom Takt über CG zu trennen. Die schematische Darstellung eines Systems zum dynamischen Power Management ist in Abbildung 5.12 dargestellt. Die folgenden Messungen wurden an diesem System

auf dem ML510 Board durchgeführt.
Für die Messungen wurde der FPGA mit einem initialen Bitstrom konfiguriert. Wenn sowohl der HWA (in diesem Fall die *CSE*) als auch der leseseitige Teil des ICAP Controllers abgeschaltet sind (f_R Domain in Abbildung 5.12), konnte ein Spannungsabfall von 35,7 mV gemessen werden. Dies entspricht einer Core Verlustleistung P_{Core} von 1,62 W. Wird hingegen die bereits konfigurierte *CSE* per CG zugeschaltet, erhöht sich dieser Wert auf 37,6 mV, was einer Core Verlustleistung von 1,71 W entspricht. Das bedeutet, dass der Anteil an dynamischer Verlustleistung, den die *CSE* bei der gegebenen Verdrahtung insgesamt zur Core Verlustleistung P_{Core} beiträgt, bei etwa 5% liegt bzw. 0,09 W beträgt. Bei Zuschaltung der f_R Domain des ICAP Controllers und bei abgeschaltetem HWA wurden 38,4 mV gemessen, was den Wert der Core Verlustleistung auf 1,74 W erhöht. Das sind etwa 8% der Gesamtverlustleistung bzw. 0,12 W zusätzlich. Da sich Verarbeitung in einem HWA und Rekonfiguration im gegenwärtigen AutoVision System mit nur einer PRR gegenseitig ausschließen, werden niemals ICAP Controller und HWA gleichzeitig aktiv sein. Dies ändert sich natürlich, wenn mehrere PRRs auf dem Device existieren, da in diesem Fall durchaus ein HWA aktiv sein kann, während ein anderer HWA gerade rekonfiguriert wird. Um Messfehler auszuschließen, wurden der Spannungsabfall bei aktivem HWA und ICAP Controller gemessen. Die Messung lieferte einen Wert von 40,2 mV, was einer Verlustleistung von 1,83 W entspricht. Mit getaktetem ICAP Controller (0,12 W) und HWA (0,09 W) wird im Gesamtsystem eine Core Verlustleistung P_{core} von (1,62 W + 0,21 W =) 1,83 W erzeugt. Dies entspricht auch dem Wert, der in Tabelle 6.9 bei geladener *CSE* und f_R = 200 MHz zu finden ist. Es besteht auch die Möglichkeit, das gegenwärtige PRM durch einen Blank Bitstrom aus der PRR zu entfernen, anstatt über CG abzuschalten. Damit wird die PRR, ähnlich wie beim CG, auch nicht mehr mit Takt versorgt, was zu einer reduzierten dynamischen Verlustleistung führt. Jedoch gibt es einen Unterschied zwischen der Verwendung von CG und der Verwendung eines Blank Bitstroms. Der Unterschied ist in Tabelle 6.13 dargestellt.

geladener HWA	**ICAP getaktet**	**HWA getaktet**	**Messung [mV]**	P_{Core} **[W]**	Δ **[W]**
CSE	nein	nein	35,7	1,62	0,0
CSE	nein	ja	37,6	1,71	0,09
CSE	ja	nein	38,4	1,74	0,12
BM	ja	nein	36,3	1,65	0,03
BM	nein	nein	33,7	1,53	-0,09
CSE	ja	ja	40,2	1,83	0,21

Tabelle 6.13: Verlustleistungsmessungen unter Verwendung von CG bei einer ICAP Frequenz f_R von 200 MHz und geladener *CSE*

Wird die *CSE* durch einen Blank Bitstrom entfernt, ergibt sich ein wesentlich niedrigerer

Wert (1,65 W) an Verlustleistung, verglichen mit dem Abschalten des HWAs über CG. Dies weist drauf hin, dass das Entfernen des PRMs aus der PRR mit einem Blank Bitstrom zusätzlich die statische Verlustleistung beeinflußt, wie schon in Abschnitt 5.3.3.1 beschrieben. Mit diesem Versuchsaufbau läßt sich nun messen, wie hoch der Anteil an dynamischer Verlustleistung ist, der durch den HWA bzw. durch die f_R Domain des ICAP Controllers erzeugt wird. Zusätzlich kann mit diesem Versuchsaufbau gezeigt werden, was sich an statischer Verlustleistung einsparen läßt, wenn ein HWA durch einen Blank Bitstrom entfernt wird. Ist der Takt für ICAP Controller und HWA aktiviert, wurde insgesamt eine Core Verlustleistung von 1,83 W gemessen. Abschalten des Taktes der *CSE* über CG resultiert in einer Verlustleistung von 1,74 W. Daraus läßt sich schlussfolgern, dass die *CSE* 0,09 W an dynamischer Verlustleistung beisteuert. Wird zusätzlich die vom Takt getrennte *CSE* durch einen Blank Bitstrom aus der PRR entfernt, reduziert sich die Core Verlustleistung auf 1,65 W. Somit kann auch die statische Verlustleistung um 0,09 W reduziert werden. Daher ist es bei längerer Inaktivität (länger als 1 Sekunde) eines PRMs ratsam, dieses durch einen Blank Bitstrom aus der PRR zu entfernen, um zusätzlich Verlustleistung einzusparen. Bei längerer Inaktivität kann die zusätzliche dynamische I/O Verlustleistung, die durch das Laden partieller Bitströme aus einem externen Speicher entsteht, kompensiert werden. Dies ist im IntraVFR System nicht der Fall. Das zusätzliche Laden des partiellen Bitstroms für das Blank Modul wird aufgrund der in [Hüb07] präsentierten Messergebnisse zu einer erhöhten Gesamtverlustleistung ($P_{Core} + P_{IO}$) führen.
Durch Einsatz des dynamischen Power Managements (s. Abschnitt 5.3.3.2) kann im IntraVFR dennoch Verlustleistung eingespart werden. Der Ablauf während eines Video-Frames und die Verlustleistungen für jeden Abschnitt bei Verwendung von CG sind in Abbildung 6.9 dargestellt.
Ist der ICAP Controller über CG vom Takt getrennt, lassen sich laut Tabelle 6.13 insgesamt 0,12 W an dynamischer Verlustleistung einsparen. Dementsprechend reduziert sich die Gesamtverlustleistung bei geladener *CSE* von 1,83 W auf 1,71 W. Um den selben Betrag reduziert sich P_{Core}, wenn die *ME* geladen ist. In diesem Fall von 1,88 W auf 1,76 W. Während einer DPR im AutoVision System ist der HWA inaktiv und kann über CG vom Takt getrennt werden. Wie in Tabelle 6.13 zu erkennen ist, reduziert sich die gesamte Core Verlustleistung dann um 0,09 W. Dieser Betrag wird von der Worst Case Abschätzung für eine DPR subtrahiert, so dass sich ein Betrag von 1,83 W anstatt 1,92 W ergibt. Wird die *CSE* direkt im Anschluß an die *ME* wieder auf das Device konfiguriert, kann im Anschluß sowohl der ICAP Controller als auch der HWA vom Takt getrennt werden, da beide für den Rest des Video Frames inaktiv sind. In diesem Fall reduziert sich die gesamte Core Verlustleistung bei geladener *CSE*, um den Betrag, den der ICAP Controller (0,12 W) und der HWA (0,09 W) an dynamischer Verlustleistung beisteuern von 1,83 W auf 1,62 W. Die durchschnittliche Verlustleistung bei Verwendung des dynamischen Power Managements ist in Tabelle 6.14 dargestellt.
Insgesamt resultiert die Verwendung des dynamischen Power Managements in einer

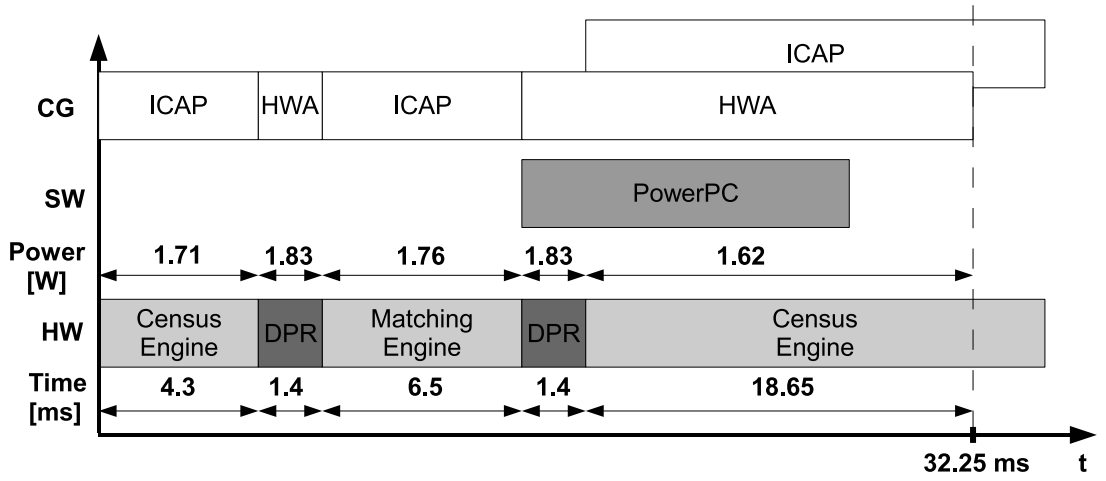


Abbildung 6.9: Zeitachse mit bei Verwendung des dynamischen Power Managements im IntraVFR Design

durchschnittlichen Core Verlustleistung von 1,68 W. Dieser Wert ist wesentlich geringer, als der des nicht rekonfigurierbaren Systems von 1,79 W. Wenn ein nicht rekonfigurierbares System allerdings in ein DPR System umgewandelt werden kann, setzt dies sich gegenseitig ausschließende IP Cores auf dem FPGA voraus. Ist dies in einem nicht rekonfigurierbaren System der Fall, können die inaktiven IP Cores jederzeit über CG abgeschaltet werden, um somit zusätzlich dynamische Verlustleistung einzusparen.
Da die Rekonfigurationszeit, wie in Gleichung 5.5 zu erkennen ist, proportional zur Frequenz f_r ist, stellt sich die Frage, ob die schnelle Rekonfiguration auch aus Verlustleistungsgesichtspunkten Vorteile bringt. In [LPF09a] zeigen die Ergebnisse, dass die Rekon-

Verlustleistung in unterschiedlichen Sektionen:	
CSE	4,3 ms x 1,71 W (ICAP deaktiviert)
CSE	18,65 ms x 1,62 W (ICAP und HWA deaktiviert)
ME	6,5 ms x 1,76 W (ICAP deaktiviert)
DPR	2 x 1,4 ms x 1,83 W (HWA deaktiviert)
Durchschnittliche Verlustleistung für die Dauer eines Video-Frames:	
1,68 W	

Tabelle 6.14: Durchschnittliche Verlustleistung bei dynamischem Power Management im IntraVFR System.

figurationsenergie W_{Core} bei einem Rekonfigurationsdurchsatz von 400 MB/s geringer ist als bei 50 MB/s. Daher scheint eine schnellere Rekonfiguration auch für den Verbrauch an Core Verlustleistung vorteilhaft zu sein. In [LPF09a] wurde der ICAP Controller allerdings immer mit derselben Frequenz von 100 MHz getaktet. Das bedeutet in beiden Fällen ist der Spannungslevel und damit auch die Verlustleistung zu Beginn der Messungen gleich. Dies ist im IntraVFR Design nicht der Fall. Um in diesem Design einen höheren Durchsatz zu erreichen, wurde der ICAP Controller übertaktet. Mit höherer Frequenz erhöht sich aber auch der Spannungslevel und damit auch die Verlustleistung zu Beginn einer Messung, wie in Tabelle 6.9 zu erkennen ist. Durch die Erhöhung des Rekonfigurations-Durchsatzes mit höheren Frequenzen konnte insgesamt keine Einsparung oder Erhöhung der Verlustleistung P_{Core} nachgewiesen werden.

6.5 Zusammenfassung

In Kapitel 6 wurden die Ergebnisse präsentiert, die aus Messungen an zwei prototypischen Implementierungen hervorgingen. Zuerst wurden die Verarbeitungszeiten der HWAs und deren Effektivität vorgestellt. Fast alle HWAs sind in der Lage annähernd 1 Pixel mit seiner Nachbarschaft pro Taktzyklus zu verarbeiten. Bei einer Bildauflösung von 640 × 480 Pixeln entspricht dies einer Verarbeitungszeit von ca. 3 ms, so dass im Normalfall genügend Zeit für die Verarbeitung in SW auf dem PPC bleibt, bevor der nächste Video-Frame im System erscheint.
Durch die Einführung von DPR konnten bei 3 HWAs im InterVFR System 31% an Registern, 35% an LUTs und 17% an BRAMs auf der V2P Plattform eingespart werden. Durch weitere HWAs, wie die in [Cra06] beschriebene EdgeEngine zur Fahrspurerkennung, können insgesamt noch mehr Ressourcen eingespart werden. Im IntraVFR System auf einem V5 mit 2 austauschbaren HWAs konnten 12% der Register, 24% der LUTs und 19% der BRAMs eingespart werden. Der DPR Overhead konnte so gering gehalten werden, dass sich durch Einführung der DPR wie erwartet ein deutlicher Vorteil hinsichtlich der Ressourcen ergab.
Durch das Übertakten und Verifizieren der Konfigurationsdaten konnten die höchsten Durchsätze auf V2P und V5 erreicht werden. Durch die damit verbundenen kurzen Rekonfigurationszeiten, konnte das Konzept der InterVFR und der IntraVFR überhaupt in den Prototypen realisiert werden. Die IntraVFR lässt sich überall dort einsetzen, wo die nächste Prozessierungsstufe ein globales Ergebnis der vorherigen Stufe benötigt, die beiden Stufen also nicht pipelinebar sind.
Letztendlich bleibt festzustellen, dass sowohl bei der InterVFR als auch bei der IntraVFR Ressourcen eingespart werden können. Um überhaupt Verlustleistungsmessungen an FPGAs bei verschiedenen Frequenzen durchführen zu können, ist auf die gleiche Verdrahtung zwischen den verschiedenen Systemen zu achten. Überlappt die Rekonfiguration mit der Verarbeitung der Bilddaten oder Features in SW auf dem PPC, entstehen im System

keine zusätzlichen Latenzen. Dies ist im InterVFR System der Fall. Im IntraVFR System hingegen erhöht sich die Verarbeitungszeit um die Dauer einer Rekonfiguration (bei 2 Rekonfigurationen pro Video-Frame). Das IntraVFR Design wird aufgrund der zu ladenden partiellen Bitströme insgesamt mehr Verlustleistung ($P_{Core} + P_{IO}$) verbrauchen, wie das nicht rekonfigurierbare Design. Falls ein PRM jedoch für längere Zeit (Sekundenbereich und darüber) inaktiv ist (beispielsweise im InterVFR Design), sollte es mit einem Blank Bitstrom vom Device entfernt werden, statt den Takt über CG abzuschalten, um zusätzlich Verlustleistung einzusparen. Im InterVFR Design bei Verwendung des dynamischen Powermanagements ist in den meisten Fällen davon auszugehen, dass die Gesamtverlustleistung niedriger ist als beim nicht rekonfigurierbaren Design.
Im IntraVFR Design kann beim Austausch von zwei Modulen keine Verbesserung der Gesamtverlustleistung erzielt werden. Dennoch kann die Core Verlustleistung P_{Core} durch das dynamische Power Management in einem DPR System deutlich reduziert werden.

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde das AutoVision System, ein integriertes System zur Echtzeit-Bildverarbeitung vorgestellt, das beispielsweise bei videobasierter Fahrerassistenz eingesetzt werden kann. Dieses System nutzt Hardware Acceleratoren (HWAs) um Bildverarbeitungsalgorithmen zu beschleunigen. Durch eine modul-basierte Pixelverarbeitungspipeline kann die Entwicklungszeit der HWAs verkürzt werden. Beispielhaft wurden mit Hilfe einer Modulbibliothek verschiedene HWAs für die Bildverarbeitung erstellt, die CPU-basierte Lösungen in der Ausführungszeit deutlich übertreffen, obwohl die Taktfrequenz der HWAs signifikant niedriger ist. Durch die Konfiguration der Elemente der Modulbibliothek (über Syntheseattribute) und deren flexible Verschaltung mit weiteren Elementen unterscheiden sich die HWAs in ihrem Aufbau deutlich voneinander.
Um nicht alle möglichen HWAs auf dem FPGA vorhalten zu müssen, wurde ein Ansatz und dessen Realisierung vorgestellt, um die HWAs abhängig von der gegenwärtigen Fahrsituation zur Laufzeit des Systems zu laden und gleichzeitig unbenutzte HWAs zu entfernen. Der statische Teil des Systems muß dabei weder angehalten noch resettet werden. In diesem situationsadaptiven SoC wurden Optimierungen durchgeführt, um so schnell wie möglich rekonfigurieren zu können. Die in dieser Arbeit präsentierten Ergebnisse zeigen, dass es bei einer Framerate von 31 fps ohne weiteres möglich ist zwischen zwei Video-Frames (interVFR) und sogar mehrmals innerhalb eines Video-Frames (intraVFR) zu rekonfigurieren, ohne dass ein einziger Video-Frame verworfen werden muß. Diese hochperformante Lösung benötigt verglichen mit dem Gesamtsystem nur einen geringen Anteil der Ressourcen, sodass trotz durch Hinzufügen dieses Overheads, der mit der Einführung von DPR in einem System einhergeht, eine deutliche Einsparung an Ressourcen durch den Austausch von HWAs erzielt werden konnte.
Anstatt die DPR nur für den Austausch von on-chip Funktionionalität zu nutzen, wurde in dieser Arbeit ein Ansatz vorgestellt, um DPR zur aktiven Einsparung von statischer und dynamischer Verlustleistung einzusetzen. Alle Systeme auf SRAM-basierten FPGAs lassen sich durch diesen Ansatz erweitern, um nicht benötigte Module und damit unnötige Verursacher von Verlustleistung vom FPGA zu entfernen. Voraussetzung hierfür ist ein partitionsbasiertes Design, dessen Erstellung beispielsweise von den neusten Xilinx Entwicklungstools unterstützt wird.
Um die DPR in Fahrzeugen einzusetzen muß die AutoVision Architektur auf low-cost Devices wie die Spartan Serie von Xilinx portiert werden. Der modulare Aufbau der Architektur erlaubt dies ohne große Änderungen, wie beispielsweise in [Ali09] gezeigt werden konnte. Der PBRF unterstützt momentan jedoch nicht die Devices der Spartan Serie. Dies wird sich aber in den neuen FPGA Generationen von Xilinx der Artix

und Kintex Serie ändern. Diese Devices präsentieren eine kostengünstige Alternative zu Mikrocontrollern und können durch Automotive-Zertifizierung auch verbreitet in Fahrzeugen eingesetzt werden.
Mit dieser Arbeit wurde gezeigt, dass gegenwärtige ASICs zur Bildverarbeitung im Automobil, wie der EyeQ2 Chip von Mobileye durch den Einsatz von rekonfigurierbarer Hardware profitieren können. Ob sich die dynamisch partielle Rekonfiguration von FPGAs in industriellen Anwendungen durchsetzt, bleibt dennoch abzuwarten. Es wird immer HW Lösungen (ASICs, ASSPs) geben, die mit einer höheren Rechenleistung oder geringerer Verlustleistung aufwarten können. Doch vielleicht setzen sich in Zukunft auch hier nicht die schnellsten oder verlustleistungsärmsten Lösungen durch, sondern die, die sich am schnellsten an Veränderungen anpassen.

A Appendix

A.1 Aufbau des Bitstromheaders

00 09 0f f0 0f f0 0f f0 0f f0 00 00 01	Header	...ð.ð.ð.ð...
61 00 0b 73 79 73 74 65 6d 2e 6e 63 64 00	Design Name	a..system.ncd.
62 00 0b 32 76 70 33 30 66 66 38 39 36 00	Device Name	b..2vp30ff896.
63 00 0b 32 30 30 38 2f 30 38 2f 32 37 00	Datum	c..2008/08/27.
64 00 09 31 38 3a 34 33 3a 30 34 00	Uhrzeit	d..18:43:04.
65 00 16 1b 24	Bitstromlänge	e...$
ff ff ff ff	Dummy word	
aa 99 55 66	Sync word	

Tabelle A.1: Bitstromheader für ein XC2VP30 Device

Wie in Tabelle A.1 zu erkennen ist sind im Bitstromheader verschiedene Informationen über den Bitstrom selbst enthalten. Im Anschluß an diese Informationen folgen ein oder mehrere Dummy Worte (0xFFFFFFFF). Anschließend folgt das Synchronisationswort (SYNC) 0xAA995566, um den Configuration Packet Processor *CPP* zu synchronisieren. Keines der am Konfigurations-Interface bzw. *CPP* ankommenden Pakete wird verarbeitet, solange das SYNC Wort nicht vom *CPP* empfangen wurde.
Aus Gründen der Vollständigkeit sind in diesem Abschnitt die einzelnen Segmente des Bitstromheaders aufgeführt. Aus Tabelle A.1 läßt sich leicht die Struktur des Bitstromheaders anhand der einzelnen Segmente erkennen. Die ersten 2 Byte (0x0009) geben die Länge des folgenden Headers in Byte an. Anschließend symbolisieren 2 weitere Byte (0x0001) die Länge der folgenden Daten, in diesem Fall 1 Byte. Dieses Byte symbolisiert den Beginn der einzelnen Headersegmente. Diese Segmente sind von a bis e durchnummeriert. Der Wert 0x61 entspricht dem ASCII Zeichen a und weißt auf den Beginn des folgenden Design Namens hin. Die Struktur der nun folgenden Segmente wiederholt sich ständig. Das erste Byte weißt auf Design Name (a), Device Name (b), Datum (c) oder Uhrzeit (d) hin. Die anschließenden 2 Byte (0x000b) charakterisieren jeweils die Länge der Segmente. Ein abschließendes Byte (0x00) repräsentiert das Ende, des alten und gleichzeitig den Beginn eines neuen Segments. Eine Ausnahme bildet das Segment für die Bitstromlänge. Hier werden die anschließenden 4 Bytes als Bitstromlänge interpretiert.
Zusammenfassend sind die einzelnen Segmente des Bitstromheaders mit korrespondierender Länge in Tabelle A.2 dargestellt.

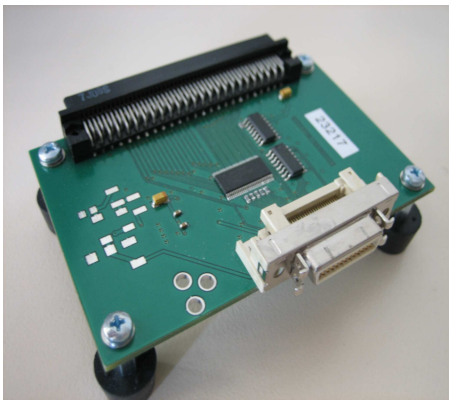
Segment 1		
2 Bytes	Länge: 0x0009 Byte	big endian
9 Bytes	Präambel	
2 Bytes	Länge: 0x0001 Byte	1 Byte
Segment 2		
1 Byte	Wert: 0x61	(„a“)
2 Bytes	Länge: 0x000b Byte	11 Byte
11 Bytes	String des Design Namens	„system.ncd“
Segment 3		
1 Byte	Wert: 0x62	(„b“)
2 Bytes	Länge: 0x000b	11 Byte
11 bytes	String des Device Namens	„2vp30ff896“
Segment 4		
1 Byte	Wert: 0x63	(„c“)
2 Bytes	Länge: 0x000b	11 Byte
11 Bytes	String des Datums	„2008/08/27“
Segment 5		
1 Byte	Wert: 0x64	(„d“)
2 Bytes	Länge: 0x0009	9 Byte
9 Bytes	Zeit-String	„18:43:04“
Segment 6		
1 byte	Wert 0x65	(„e“)
4 Bytes	Länge: 0x00161b24	Länge der folgenden Bitstromdaten

Tabelle A.2: Bitstromheader für ein XC2VP30 Device unterteilt in Segmente

A.2 Accessory Boards

In Abbildung A.1 sind zwei Accessory Boards zum Einspielen digitaler Videodaten dargestellt, die von B. Gatscher bei der Firma Sensor to Image GmbH entwickelt wurden [Gat09]. Abbildung A.1(a) zeigt ein Accessory Board zur direkten Anbindung einer CameraLink Kamera [JAI10]. Die Bilddaten werden in diesem Fall im Rohdatenformat an die Pins des FPGA geliefert. Eine andere Alternative, um Bilddaten digital in das AutoVision System einzuspeisen, besteht in der Verwendung eines DVI Input Boards, das in Abbildung A.1(b) dargestellt ist. In diesem Fall kann beispielsweise ein PC als Bildquelle verwendet werden. Die Anpassung der Framerate wird FPGA-seitig vorgenommen.

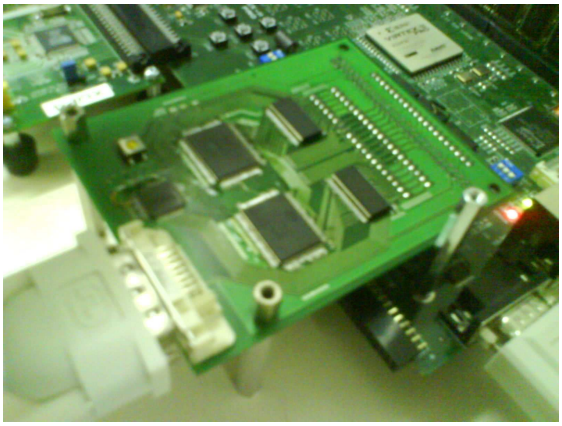
Abbildung A.2(a) zeigt ein Accessory Board zur Ausgabe digitaler Videodaten, dass von R. Hartl implementiert wurde und das zugehörige Blockschaltbild in Abbildung A.2(b). Das Video Output Board realisiert das Konzept des Double Buffering mit Hilfe zweier SRAMs. Somit sind die Bilddaten nur einmal über den PLB Bus und den VOUT Core



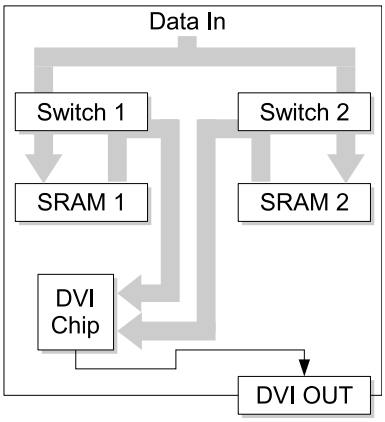
(a) CameraLink Input Board

(b) DVI Input Board

Abbildung A.1: Accessory Boards zum Einspielen digitaler Videodaten



(a) Video Output Accessory Board



(b) Blockschaltbild des Video Output Boards

Abbildung A.2: Accessory Boards zur Ausgabe digitaler Videodaten

in die SRAMs zu übertragen und können von dort mit einer beliebigen Frequenz am Ausgang (Monitor) dargestellt werden. Während Bilddaten über den Switch 1 in das erste SRAM geschrieben werden, stellt der Video Output Core die Bilddaten über Switch 2 aus dem zweiten SRAM am Monitor dar. Ein Blockschaltbild des Video Output Boards ist in Abbildung A.2(b) dargestellt. Über die vom VOUT Core gesteuerten Switches wird entschieden, welches der beiden SRAMs beschrieben und welches ausgelesen wird.

A.3 Konfigurationsframes

In den folgenden Tabellen A.3 bis A.5 sind die Anzahl an Konfigurations-Bits für alle V2P, V4 und V5 Devices dargestellt. In der letzten Zeile der Tabellen ist die Anzahl an Konfigurations-Frames per Konfigurations-Column N_C^F dargestellt. N_C^F ist abhängig von der Device-Familie (V2P, V4 oder V5). Die Spalte O repräsentiert die Anzahl an Overhead-Frames pro Konfigurations-Row. Overhead-Frames werden verwendet, um die Anzahl an Konfigurations-Frames pro Konfigurations-Row auf einen bestimmten Wert zu padden. N_D^R repräsentiert die Anzahl an Konfigurations-Rows pro Device und N_D^F die Gesamtzahl an Konfigurations-Frames auf dem Device. H^F ist die Höhe eines 1-Bit breiten Konfigurations-Frames.

Device	**CLB Array**	**IOB**	**IOI**	**DSP**	**CLB**	**BRAM**	**INT**	**GCLK**	**MGT**	O	N_D^R	N_D^F	H^F [bit]	Config. Bits [bit]
		Anzahl Colums per Konfigurations-Row												
XC2VP2	22 x 16	2	2	0	22	4	4	1	0	0	1	884	1472	1301248
XC2VP4	22 x 40	2	2	0	22	4	4	1	0	0	1	884	3392	2998528
XC2VP7	34 x 40	2	2	0	34	6	6	1	0	0	1	1320	3392	4477440
XC2VP20	46 x 56	2	2	0	46	8	8	1	0	0	1	1756	4672	8204032
XC2VPX20	46 x 56	2	2	0	46	8	8	1	0	0	1	1756	4672	8204032
XC2VP30	46 x 80	2	2	0	46	8	8	1	0	0	1	1756	6592	11575552
XC2VP40	58 x 88	2	2	0	58	10	10	1	0	0	1	2192	7232	15852544
XC2VP50	70 x 88	2	2	0	70	12	12	1	0	0	1	2628	7232	19005696
XC2VP70	82 x 104	2	2	0	82	14	14	1	0	0	1	3064	8512	26080768
XC2VPX70	82 x 104	2	2	0	82	14	14	1	0	0	1	3064	8512	26080768
XC2VP100	94 x 120	2	2	0	94	16	16	1	0	0	1	3500	9792	34272000
N_C^F		4	22	n.a.	22	64	22	4	n.a.					

Tabelle A.3: Anzahl und Eigenschaften von Konfigurations-Frames in V2P Devices

Device	**CLB Array**	**IOB**	**IOI**	**DSP**	**CLB**	**BRAM**	**INT**	**GCLK**	**MGT**	O	N_D^R	N_D^F	H^F [bit]	Config. Bits [bit]
		Anzahl Colums per Konfigurations-Row												
XC4VLX15	64 x 24	3	0	1	24	3	3	1	0	6	4	3600	1312	4723200
XC4VLX25	96 x 28	3	0	1	28	3	3	1	0	6	6	5928	1312	7777536
XC4VLX40	128 x 36	3	0	1	36	3	3	1	0	6	8	9312	1312	12217344
XC4VLX60	128 x 52	3	0	1	52	5	5	1	0	6	8	13472	1312	17675264
XC4VLX80	160 x 56	3	0	1	56	5	5	1	0	6	10	17720	1312	23248640
XC4VLX100	192 x 64	3	0	1	64	5	5	1	0	6	12	23376	1312	30669312
XC4VLX160	192 x 88	3	0	1	88	6	6	1	0	6	12	30720	1312	40304640
XC4VLX200	192 x 116	3	0	1	116	7	7	1	0	6	12	39120	1312	51325440
XC4VSX25	64 x 40	3	0	4	40	8	8	1	0	6	4	6940	1312	9105280
XC4VSX35	96 x 40	3	0	4	40	8	8	1	0	6	6	10410	1312	13657920
XC4VSX55	128 x 48	3	0	8	48	10	10	1	0	6	8	16632	1312	22702848
XC4VFX12	64 x 24	3	0	1	24	3	3	1	0	6	4	3600	1312	4723200
XC4VFX20	64 x 36	3	2	1	36	5	5	1	0	6	4	5488	1312	7200256
XC4VFX40	96 x 52	3	2	1	52	7	7	1	0	6	6	11352	1312	14893824
XC4VFX60	128 x 52	3	2	2	52	8	8	1	0	6	8	15972	1312	20960512
XC4VFX100	160 x 68	3	2	2	68	10	10	1	0	6	10	25170	1312	33023040
XC4VFX140	192 x 84	3	2	2	84	12	12	1	0	6	12	36444	1312	47814528
N_C^F		30	n.a.	21	22	64	20	3	20					

Tabelle A.4: Anzahl und Eigenschaften von Konfigurations-Frames in V4 Devices

Device	CLB Array	IOB	IOI	DSP	CLB	BRAM	INT	GCLK	MGT	O	N_D^R	N_D^F	H^F	Config.
		Anzahl Colums per Konfigurations-Row											[bit]	Bits [bit]
XC5VLX30	80 x 30	3	0	1	30	2	2	1	0	4	4	1594	1312	8365312
XC5VLX50	120 x 30	3	0	1	30	2	2	1	0	4	6	1594	1312	12547968
XC5VLX85	120 x 54	3	0	1	54	4	4	1	0	4	6	2774	1312	21836928
XC5VLX110	160 x 54	3	0	1	54	4	4	1	0	4	8	2774	1312	29115904
XC5VLX155	160 x 76	3	0	2	76	6	6	1	0	4	8	3910	1312	41039360
XC5VLX220	160 x 108	3	0	2	108	6	6	1	0	4	8	5062	1312	53130752
XC5VLX330	240 x 108	3	0	2	108	6	6	1	0	4	12	5062	1312	79696128
XC5VLX20T	60 x 26	3	1	1	26	3	3	1	1	4	3	1586	1312	6242496
XC5VLX30T	80 x 30	3	1	1	30	3	3	1	1	4	4	1784	1312	9362432
XC5VLX50T	120 x 30	3	1	1	30	3	3	1	1	4	6	1784	1312	14043648
XC5VLX85T	120 x 54	3	1	1	54	5	5	1	1	4	6	2964	1312	23332608
XC5VLX110T	160 x 54	3	1	1	54	5	5	1	1	4	8	2964	1312	31110144
XC5VLX155T	160 x 76	3	1	2	76	7	7	1	1	4	8	4100	1312	43033600
XC5VLX220T	160 x 108	3	1	2	108	7	7	1	1	4	8	5252	1312	55124992
XC5VLX330T	240 x 108	3	1	2	108	7	7	1	1	4	12	5252	1312	82687488
XC5VSX35T	80 x 34	3	1	6	34	6	6	1	1	4	4	2542	1312	13340416
XC5VSX50T	120 x 34	3	1	6	34	6	6	1	1	4	6	2542	1312	20010624
XC5VSX95T	160 x 46	3	1	10	46	8	8	1	1	4	8	3402	1312	35707392
XC5VSX240T	240 x 78	3	1	11	78	11	11	1	1	4	12	5056	1312	79601664
XC5VTX150T	200 x 58	3	2	1	58	6	6	1	2	4	10	3298	1312	43269760
XC5VTX240T	240 x 78	3	2	1	78	7	7	1	2	4	12	4176	1312	65746944
XC5VFX30T	80 x 38	3	1	2	38	6	6	1	1	4	4	2574	1312	13508352
XC5VFX70T	160 x 38	3	1	2	38	6	6	1	1	4	8	2574	1312	27016704
XC5VFX100T	160 x 56	3	1	4	56	9	9	1	1	4	8	3752	1312	39380992
XC5VFX130T	200 x 56	3	1	4	56	9	9	1	1	4	10	3752	1312	49226240
XC5VFX200T	240 x 68	3	1	4	68	11	11	1	1	4	12	4500	1312	70848000
N_C^F		54	n.a.	28	36	128	30	4	32					

Tabelle A.5: Anzahl und Eigenschaften von Konfigurations-Frames in V5 Devices

Stichwortverzeichnis

Literaturverzeichnis

[ABJ+08] Alcantarilla, P.F. ; Bergasa, L. M. ; Jimenez, P. ; Sotelo, M. A. ; Parra, I. ; Fernandez, D. ; Mayoral, S. S.: Night time vehicle detection for driving assistance lightbeam controller. In: *Proceedings of IV '08, Eindhoven, Netherlands*, 2008, S. 291 – 296

[AERP02] Arias-Estrada, Miguel ; Rodriguez-Palacios, Eduardo: An FPGA Co-processor for Real-Time Visual Tracking. In: *Proceedings of FPL '02, Montpellier, France*. London, UK : Springer-Verlag, 2002. – ISBN 3–540–44108–5, S. 710–719

[AKP08] Abusaidi, P. ; Klein, M. ; Philofsky, B.: Virtex-5 FPGA System Power Design Considerations (WP258) / Xilinx Inc. Version: February 14^{th} 2008. `http://www.xilinx.com/support/documentation/white_papers/wp285.pdf`. 2008 (v1.0). – Forschungsbericht

[Alf08] Alfke, P.: FIFOs in Virtex-5 FPGAs (WP333) / Xilinx Inc. Version: March 24^{th} 2008. `http://www.xilinx.com/support/documentation/white_papers/wp333.pdf`. 2008 (v1.0). – Forschungsbericht

[Alt02] Altherr, Ralf M.: *Reduktion der Verlustleistung von ICs durch eine chipintegrierte Versorgungsspannungsregelung mit einem Boost-Konverter*, Universität Ulm, Diss., 2002

[Alt07] Altera (Hrsg.): *Stratix III Programmable Power*. 1.1. Altera, May 2007. `http://www.altera.com/literature/wp/wp-01006.pdf`

[ANT04] Anderson, Jason H. ; Najm, Farid N. ; Tuan, Tim: Active leakage power optimization for FPGAs. In: *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays (FPGA '04)*. New York, NY, USA : ACM, 2004. – ISBN 1–58113–829–6, S. 33–41

[ARB99] Adário, Alexandro M. S. ; Roehe, Eduardo L. ; Bampi, Sergio: Dynamically Reconfigurable Architecture for Image Processor Applications. In: *Proceedings of the 36th annual ACM/IEEE Design Automation Conference (DAC '99)*, 1999, 623–628

[Bar08] Barnden, C.: Driver Assistance Systems Pose FPGA Opportunities. In: *Xcell Journal* 66 (2008), 16–18. http://www.xilinx.com/publications/xcellonline/xcell_66/xc_pdf/p16-18_66_Col_XO.pdf

[BB95] Beauchemin, S. S. ; Barron, J. L.: The Computation of Optical Flow. In: *ACM Computing Surveys (CSUR)* 27 (1995), Nr. 3, 433–466. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.3738&rep=rep1&type=pdf

[BB08] Beekema, M. ; Broeders, H.: Computer Architectures for Vision-Based Advanced Driver Assistance Systems / TU Delft. Version: 2008. http://www.xs4all.nl/~hc11/paper_ca_st.pdf. 2008. – Forschungsbericht

[BBHN04] Blodget, B. ; Bobda, C. ; Hübner, M ; Niyonkuru, A.: Partial and Dynamically Reconfiguration of Xilinx Virtex-II FPGAs. In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2004), Leuven*, 2004, 801-810

[BCY+09] Bomel, P. ; Crenne, J. ; Ye, L. ; Diguet, J.-P. ; Gogniat, G.: Ultra-Fast Downloading of Partial Bitstreams through Ethernet. In: *Proceedings of ARCS '09, Delft, The Netherlands*, Springer, 10^{th}-13^{th} September 2009 (Lecture Notes in Computer Science), S. 72–83

[BHD00] Betke, M. ; Haritaoglu, E. ; Davis, L. S.: Real-time multiple vehicle detection and tracking from a moving vehicle. In: *Machine vision and applications* 12 (2000), Nr. 2, S. 69–83

[BHU03] Becker, J. ; Hübner, M.l ; Ullmann, M.: Power Estimation and Power Measurement of Xilinx Virtex FPGAs Trade-offs and Limitations. In: *Proceedings of the 16th symposium on Integrated circuits and systems design (SBCCI '03)*. Washington, DC, USA : IEEE Computer Society, 2003, 283–288

[BMZ08] Bagni, D. ; Marzotto, R. ; Zoratti, P.: Building Automotive Driver Assistance System Algorithms with Xilinx FPGA Platforms. In: *Xcell Journal* 66 (2008), 20–26. http://www.xilinx.com/publications/xcellonline/xcell_66/xc_pdf/p20-26_66_F_XiASIM1.pdf

[BR99] Betz, V. ; Rose, J.: Circuit Design, Transistor Sizing and Wire Layout of FPGA Interconnect. In: *IEEE Custom Integrated Circuits Conference*, 1999, S. 171–174

[Bra00] Bradski, G.: The OpenCV Library. In: *Dr. Dobb's Journal of Software Tools* (2000)

[BSB04] Boschetti, M. R. ; Silva, I. S. ; Bampi, S.: A Run-Time Reconfigurable Datapath Architecture for Image Processing Applications. In: *Proceedings of the conference on Design, automation and test in Europe (DATE'04)*. Washington, DC, USA : IEEE Computer Society, 2004. – ISBN 0–7695–2085–5–3, 242-247

[CA02] Cummings, C. E. ; Alfke, P.: Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons. In: *SNUG*, 2002

[CCCW06] Chen, Y.-L. ; Chen, Y.-H. ; Chen, C.-J. ; Wu, B.-F.: Nighttime Vehicle Detection for Driver Assistance and Autonomous Vehicles. In: *International Conference on Pattern Recognition* Bd. 1. Los Alamitos, CA, USA : IEEE Computer Society, 2006. – ISSN 1051–4651, S. 687–690

[CDF+86] Carter, W. S. ; Duong, K. ; Freeman, R. H. ; Hsieh, H.-C. ; Ja, J. Y. ; Mahoney, J. E. ; Ngo, L. T. ; Sze, S. L.: A User Programmable Reconfigurable Logic Array. In: *Proceedings of the IEEE 1986 Custom Integrated Circuits Conference, Rochester, NY, United States*, 1986, S. 233–235

[CH00] Compton, K. ; Hauck, S.: An Introduction to Reconfigurable Computing / Northwestern University, Deptartment of ECE. Version: 2000. `http://www.ece.wisc.edu/~kati/Publications/Compton_ReconfigIntro.pdf`. 2000. – Forschungsbericht

[Che09] Chen, Y.-L.: Nighttime vehicle light detection on a moving vehicle using image segmentation and analysis techniques. In: *WSEAS Transactions on Computers* 8 (2009), Nr. 3, 506–515. `http://www.wseas.us/e-library/transactions/computers/2009/31-541.pdf`. – ISSN 1109–2750

[Cop08] Cope, B.: *Video Processing Acceleration using Reconfigurable Logic and Graphics Processors*, Imperial College London, Diss., February 2008. `http://eprints.imperial.ac.uk/bitstream/10044/1/1412/1/Cope-%20BT-2008-PhD-Thesis.pdf`

[Cor09] Core Technologies (Hrsg.): *FPGA Logic Cells Comparison*. 24 Radio str., Moscow, Russia, 105005: Core Technologies, 2009. `http://ee.sharif.edu/~asic/Docs/fpga-logic-cells_V4_V5.pdf`

[Cur07] Curd, D.: Power Consumption in 65 nm FPGAs (WP246) / Xilinx Inc. Version: February 2007. `http://www.xilinx.com/support/documentation/white_papers/wp246.pdf`. 2007 (v1.2). – Forschungsbericht

[Don07] Donlin, A.: *Applications, Design Tools and Low Power Issues in FPGA Reconfiguration.* Springer-Verlag, 2007. – 513–541 S. – ISBN 978–1–4020–5868–4

[Eka04] Ekas, Paul: Leveraging FPGA Coprocessors to Optimize Automotive Infotainment and Telematics Systems. In: *Embedded Computing Design* (2004). http://www.embedded-computing.com/pdfs/Altera.Sum04.pdf

[Far00] Farnebäck, G.: Fast and Accurate Motion Estimation using Orientation Tensors and Parametric Motion Models. In: *Proceedings of 15th International Conference on Pattern Recognition* Bd. 1. Barcelona, Spain, September 2000, S. 135–139

[Fli05] Flik, T.: *Mikroprozessortechnik und Rechnerstrukturen.* Springer, 2005

[FSF10] Fossati, A. ; Schönmann, P. ; Fua, P.: Real-time vehicle tracking for driving assistance. In: *Machine Vision and Applications* (2010). http://dx.doi.org/10.1007/s00138-009-0243-6. – DOI 10.1007/s00138–009–0243–6

[Gar09] Gartner, Inc.: Semiconductor DQ Monday Report / Gartner, Inc. 2009. – Forschungsbericht

[GHSB08] Göhringer, D. ; Hübner, M. ; Schatz, V. ; Becker, J.: Runtime Adaptive Multi-Processor System-on-Chip: RAMPSoC. In: *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, 2008, 1-7

[GM07] Gessler, R. ; Mahr, T.: *Hardware-Software-Codesign - Entwicklung flexibler Mikroprozessor-FPGA-Hochleistungssysteme.* Wiesbaden : Vieweg + Teubner, 2007

[GP03] Green, R. ; Parnell, K.: FPGAs Are the Brains Behind "Smart" Cars. In: *Xcell Journal* 56 (2003), 64–66. http://www.zylinks.com/publications/xcellonline/xcell_46/xc_pdf/xc_smartcars46.pdf

[GPHB09] Göhringer, D. ; Perschke, T. ; Hübner, M. ; Becker, J.: A Taxonomy of Reconfigurable Single-/Multiprocessor Systems-on-Chip. In: *International Journal of Reconfigurable Computing* 2009 (2009), 1-11. downloads.hindawi.com/journals/ijrc/2009/395018.pdf

[Gus88] Gustafson, John L.: Reevaluating Amdahl's Law. In: *Communications of the ACM* 31 (1988), 532–533. http://mprc.pku.edu.cn/courses/architecture/autumn2005/reevaluating-Amdahls-law.pdf

[GW08] Gonzalez, R. ; Woods, R.: *Digital Image Processing (3rd Edition)*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 2008. – ISBN 013168728X

[Hau05] Haug, Karsten: *Nachtsichtgerät*. 2005

[Hüb07] Hübner, M.: *Dynamisch und partiell rekonfigurierbare Hardwaresystemarchitektur mit echtzeitfähiger On-Demand Funktionalität*, Universität Karlsruhe (TH), Diss., 2007

[HBB04] Hübner, M. ; Becker, T. ; Becker., J.: Real-time LUT-based network topologies for dynamic and partial FPGA selfreconfiguration. In: *Proceedings of the* 17_{th} *Symposium on Integrated circuits and system design (SBCCI 04)*, 2004, 28–32

[Her05] Herrmann, S.: *Ein Modell zur Beschreibung und Implementierung von Bildanalysefunktionen*, Technische Universität München, Diss., February 2005. `http://deposit.ddb.de/cgi-bin/dokserv?idn=977592103&dok_var=d1&dok_ext=pdf&filename=977592103.pdf`

[HL09] Huang, J. ; Lee, J.: A Self-Reconfigurable Platform for Scalable DCT Computation Using Compressed Partial Bitstreams and BlockRAM Prefetching. In: *ISVLSI '09: Proceedings of the 2009 IEEE Computer Society Annual Symposium on VLSI*, IEEE Computer Society, 2009, 67–72

[HMA09] Huda, S. ; Mallick, M. ; Anderson, J. H.: Clock gating architectures for FPGA power reduction. In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2009), Prague*, 2009, 112 - 118

[HS81] Horn, B. K. P. ; Schunck, B. G.: Determining Optical Flow. In: *Artificial Intelligence* 17 (1981), Nr. 1-3, S. 185–203

[HS88] Harris, C. ; Stephens, M.: A Combined Corner and Edge Detector. In: *4th ALVEY Vision Conference*, 1988, S. 147–151

[HS92] Haralick, R. M. ; Shapiro, L. G.: *Computer and Robot Vision*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1992. – ISBN 0201569434

[HUWJ04] Hübner, M. ; Ullmann, M. ; Weissel, F. ; J.Becker: Real-Time Configuration Code Decompression for Dynamic FPGA Self-Reconfiguration. In: *International Parallel and Distributed Processing Symposium (IPDPS 2004)* Bd. 4, 2004, 138b

[IBM01] IBM ; IBM (Hrsg.): *On-Chip Peripheral Bus Architecture Specifications*. Version 2.1. IBM, April 2001. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/7503E5B68F4AF08C8525770D0010C92B/$file/OpbBus.pdf

[IBM07] IBM ; IBM (Hrsg.): *128-Bit Processor Local Bus Architecture Specifications*. Version 4.7. IBM, May 2007. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3BBB27E5BCC165BA87256A2B0064FFB4/$file/PlbBus_as_01_pub.pdf

[ICaB05] I. Cabani and, G. T. ; Bensrhair, A.: Color-based detection of vehicle lights. In: *Proceedings of the IEEE Intelligent Vehicles Symposium (IVS'05)*, 2005, 278–283

[JAI10] JAI (Hrsg.): *User's Manual - CM/CB-140 MCL/PMCL*. 3.1. JAI, 2010. ftp://ftp2.imaging.de/docmanager/43943.pdf

[KAB+03] Kim, N. S. ; Austin, T. ; Blaauw, D. ; Mudge, T. ; Arbor, A. ; Flautner, F. ; Hu, J. S. ; Irwin, M. J. ; Kandemir, M. ; Narayanan, V.: Leakage Current: Moore's Law Meets Static Power. In: *Computer* 36 (2003), Nr. 12, 68–75. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.80.6927&rep=rep1&type=pdf

[Kao05] Kao, C.: Benefits of Partial Reconfiguration. In: *Xcell Journal* 55 (2005), 65–67. http://www.xilinx.com/publications/xcellonline/xcell_55/xc_pdf/xc_reconfig55.pdf

[KB06] Kesel, F. ; Bartholomä, R.: *Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs*. Oldenbourg, 2006. – ISBN 978–3486575569

[Ket09] Kettelhoit, B.: *Architektur und Entwurf dynamisch rekonfigurierbarer FPGA-Systeme*, Universität Paderborn, Diss., 2009. http://deposit.d-nb.de/cgi-bin/dokserv?idn=993834469&dok_var=d1&dok_ext=pdf&filename=993834469.pdf

[Kle05] Klein, M.: Static Power and the Importance of Realistic Junction Temperature Analysis (WP221) / Xilinx Inc. Version: March 23^{rd} 2005. http://www.xilinx.com/support/documentation/white_papers/wp221.pdf. 2005 (v1.0). – Forschungsbericht

[KM09] Kokufuta, K. ; Maruyama, T.: Real-time processing of local contrast enhancement on FPGA. In: *Proceedings of FPL '09, Prague, Czech Republic*, 2009, S. 288–293

[KO08] Kyo, S. ; Okazaki, S.: In-vehicle vision processors for driver assistance systems. In: *Proceedings of the 2008 Asia and South Pacific Design Automation Conference (ASP-DAC '08)*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2008. – ISBN 978–1–4244–1922–7, S. 383–388

[KR06] Kuon, I. ; Rose, J.: Measuring the gap between FPGAs and ASICs. In: *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays (FPGA '06)*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–292–5, S. 21–30

[Kuz04] Kuzmanov, G. K.: *The Molen Polymorphic Media Processor*, TU Delft, Diss., December 2004. `http://ce.et.tudelft.nl/publicationfiles/970_9_KuzmanovThesis_full.pdf`

[LBM+06] Lysaght, P. ; Blodget, B. ; Mason, J. ; Young, J. ; Bridgford, B.: Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In: *Proceedings of FPL '06, Madrid, Spain*, 2006, 1–6

[LD09] Lai, V. ; Diessel, O.: ICAP-I: A reusable interface for the internal reconfiguration of Xilinx FPGAs. In: *International Conference on Field-Programmable Technology (FPT 2009)*, 2009, 357 -360

[LH01] Li, Z. ; Hauck, S.: Configuration Compression for Virtex FPGAs. In: *Annual IEEE Symposium on Field-Programmable Custom Computing Machines* Bd. 0, 2001, 147-159

[LKLJ09] Liu, M. ; Kuehn, W. ; Lu, Z. ; Jantsch, A.: Run-time Partial Reconfiguration Speed Investigation and Architectural Design Space Exploration. In: *Proceedings FPL '09, Prague, Czech Republic*, 2009

[LMGVE04] Lorenz, M. G. ; Mengibar, L. ; Garcia-Valderas, M. ; Entrena, L.: Power Consumption Reduction Through Dynamic Reconfiguration. In: *14th International Conference on Field Programmable Logic and Application (FPL 2004), Leuven, Belgium*, 2004, 751-760

[Loc10] Loch, M.: MCUs für Radar- und Kamerasensoren. In: *Automobil Elektronik* 1 (2010), 18-20. `http://imperia.mi-verlag.de/imperia/md/content/ai/ae/fachartikel/ael/2010/01/ael10_01_018.pdf`

[LPF09a] Liu, S. ; Pittman, R. N. ; Forin, A.: Energy Reduction with Run-Time Partial Reconfiguration / Microsoft Research, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052. Version: September 2009. `http://research.microsoft.com/pubs/112466/Energy%20Reduction%`

20with%20Run-Time%20Partial%20Reconfiguration.pdf. 2009 (MSR-TR-2009- 2017). – Forschungsbericht

[LPF09b] Liu, S. ; Pittman, R. N. ; Forin, A.: Minimizing Partial Reconfiguration Overhead with Fully Streaming DMA Engines and Intelligent ICAP Controller / Microsoft Research. 2009 (MSR-TR-2009- 150). – Forschungsbericht

[Mac05] MacLean, W. J.: An Evaluation of the Suitability of FPGAs for Embedded Vision Systems. In: *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7695–2372–2–3, S. 131–137

[MMT+08] Manet, P. ; Maufroid, D. ; Tosi, L. ; Gailliard, G. ; Mulertt, O. ; Ciano, M. D. ; Legat, J.-D. ; Aulagnier, D. ; Gamrat, C. ; Liberati, R. ; Cuvelier, V. La Barbaand P. ; Rousseau, B. ; Gelineau, P.: An Evaluation of Dynamic Partial Reconfiguration for Signal and Image Processing in Professional Electronics Applications. In: *EURASIP Journal on Embedded Systems* 2008 (2008), November, 1–11. http://portal.acm.org/citation.cfm?id=1552178#

[Moo65] Moore, G. E.: Cramming More Components onto Integrated Circuits. In: *Electronics* 38 (1965), April, Nr. 8, S. 114–117. http://dx.doi.org/10.1109/JPROC.1998.658762. – DOI 10.1109/JPROC.1998.658762

[MRAE03] Maya-Rueda, S. ; Arias-Estrada, M.: FPGA Processor for Real-Time Optical Flow Computation. In: *Proceedings of FPL '03, Lisbon, Portugal*, 2003, S. 1103–1106

[MSS08] *Kapitel* 7 in Embedded Systems Handbook, Digital Systems and Aplications. In: Mitic, M. ; Stojcev, M. ; Stamenkovic, Z.: *An Overview of SoC Buses*. CRC Press, 2008, 7.1- 7.16

[OGJ08] O'Malley, R. ; Glavin, M. ; Jones, E.: Vehicle Detection at Night Based on Tail-Light Detection. In: *Proceedings of the Annual International Symposium on Vehicular Computing Systems (ISVCS 2008)*, 2008

[OLCM08] Osborne, W. ; Luk, W. ; Coutinho, J. G. F. ; Mencer, O.: Reconfigurable design with clock gating. In: *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS 2008)*, 2008, 187 - 194

[PA09] Prengler, A. ; Adi, K.: A Reconfigurable SIMD-MIMD Processor Architecture for Embedded Vision Processing Applications / NEC Electronics America, Inc. Version: April 20^{th} 2009. `http://america2.renesas.com/applications/automotive/documentation/09AE-0227_reconfigurable_SIMD-MIMD_processor_architecture_for_embedded_vision_processing_applications.pdf`. 2009. – Forschungsbericht

[Pel03] Pelgrims, P.: Overview: AMBA, Avalon, Coreconnect, Wishbone / DE NAYER Instituut. Version: 2003. `http://emsys.denayer.wenk.be/empro/Overview%20of%20Embedded%20Busses.pdf`. 2003 (Project IWT, 020079). – Forschungsbericht

[PHBB07] Paulsson, K. ; Hübner, M. ; Bayar, S. ; Becker, J.: Exploitation of Run-Time Partial Reconfiguration for Dynamic Power Management in Xilinx Spartan III-based Systems. In: *Proceedings of the 3rd International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2007), Montpellier, France*, 2007, S. 1–6

[PMW04] Pan, J. H. ; Mitra, T. ; Wong, W.-F.: Configuration bitstream compression for dynamically reconfigurable FPGAs. In: *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, 2004. – ISBN 0–7803–8702–3, 766–773

[Pre08] Prengler, A.: Developing Integrated Vision Applications for Active Safety Systems / NEC Electronics America, Inc. Version: 2008. `http://america2.renesas.com/applications/automotive/documentation/09AE-0227_reconfigurable_SIMD-MIMD_processor_architecture_for_embedded_vision_processing_applications.pdf`. 2008 (09AE-023). – Forschungsbericht

[Ras07] Rastogi, Ashesh: *COMPREHENSIVE ANALYSIS OF LEAKAGE CURRENT IN ULTRA DEEP SUB-MICRON (UDSM) CMOS CIRCUITS*, University of Massachusetts Amherst, Diplomarbeit, 2007. `http://scholarworks.umass.edu/cgi/viewcontent.cgi?article=1079&context=theses`

[RBH+03] Raab, W. ; Bruels, N. ; Hachmann, U. ; Harnisch, J. ; Ramacher, U. ; Sauer, C. ; Techmer, A.: A 100-GOPS Programmable Processor for Vehicle Vision Systems. In: *IEEE Design and Test of Computers* 20 (2003), Nr. 1, S. 8–16. `http://dx.doi.org/http://dx.doi.org/10.1109/MDT.2003.1173047`. – DOI http://dx.doi.org/10.1109/MDT.2003.1173047. – ISSN 0740–7475

[RCK07] Rastogi, Ashesh ; Chen, Wei ; Kundu, Sandip: On estimating impact of loading effect on leakage current in sub-65nm scaled CMOS circuits based on Newton-Raphson method. In: *Proceedings of the 44th annual Design Automation Conference (DAC '07)*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–627–1, S. 712–715

[RCN03] Rabaey, Jan M. ; Chandrakasan, Anantha P. ; Nikolic, Borivoje: *Digital integrated circuits : a design perspective*. 2. Pearson Education, 2003 (Prentice Hall electronics and VLSI series). – ISBN 0130909963

[Riv10] Rivoallon, F.: Reducing Switching Power with Intelligent Clock Gating (WP370) / Xilinx Inc. Version: March 3rd 2010. `http://www.xilinx.com/support/documentation/white_papers/wp333.pdf`. 2010 (v1.0). – Forschungsbericht

[RM06] Rullmann, M. ; Merker, R.: Maximum Edge Matching for Reconfigurable Computing. In: *Proceedings of IPDPS '06, Rhodos, Greece*, 2006

[RMMM03] Roy, K. ; Mukhopadhyay, S. ; Mahmoodi-Meimand, H.: Leakage Current Mechanisms and Leakage Reduction Techniques in Deep-Submicrometer CMOS Circuits. In: *Proceedings of the IEEE* 91 (2003), 305-327. `http://www.iet.ntnu.no/courses/tfe01/pdfs/Roy1.pdf`

[RV06] Rushinek, E. ; Vecchio, P. D.: Multi-threaded design tackles SoC performance bottlenecks. In: *EmbeddedSystems Europe* (2006), August, 26-28. `https://www.mips.com/media/files/product-materials/multithreadeddesignMIPS34K.pdf`

[San08] Santarini, M.: Driver Assistance Revs Up On Xilinx FPGA Platforms. In: *Xcell Journal* 66 (2008), 8–15. `http://www.xilinx.com/publications/xcellonline/xcell_66/xc_pdf/p08-15_66_CS_auto.pdf`

[San10] Santarini, M.: Xilinx Customer Innovation: 85,000 to 2.5 Billion Transistors and Beyond. In: *Xcell Journal* 2010 Customer Innovation Issue (2010), 8–15. `http://www.xilinx.com/publications/archives/xcell/Xcell-customer-innovation-2010.pdf`

[SB97] Smith, S. M. ; Brady, J. M.: Susan - a new approach to low level image processing. In: *International Journal of Computer Vision* 23 (1997), 45–78. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.24.2763&rep=rep1&type=pdf`

[SBB+06] Sedcole, P. ; Blodget, B. ; Becker, T. ; Anderson, J. ; Lysaght, P.: Modular dynamic reconfiguration in Virtex FPGAs. In: *Computers*

and Digital Techniques, IEE Proceedings - 153 (2006), May, Nr. 3, 157–164. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1626507

[Sch00] Schimpfle, Christian V.: *Entwurfsmethoden für verlustarme integrierte Schaltungen*, Technische Universität München, Diss., July 2000. http://tumb1.biblio.tu-muenchen.de/publ/diss/ei/2000/schimpfle.pdf

[SCW+06] Schlessman, J. ; Chen, C.-Y. ; Wolf, W. ; Ozer, B. ; Fujino, K. ; Itoh, K.: Hardware/Software Co-Design of an FPGA-based Embedded Tracking System. In: *CVPRW '06: Proceedings of the 2006 Conference on Computer Vision and Pattern Recognition Workshop*. Washington, DC, USA : IEEE Computer Society, 2006. – ISBN 0–7695–2646–2, S. 123

[Sed06] Sedcole, N. P.: *Reconfigurable Platform-Based Design in FPGAs for Video Image Processing*, Imperial College London, Diss., January 2006. http://cas.ee.ic.ac.uk/people/nps/thesis/sedcole_thesis.pdf

[SGVT05] Srinivasan, S. ; Gayasen, A. ; Vijaykrishnan, N. ; Tuan, T.: Leakage control in FPGA routing fabric. In: *Proceedings of the 2005 Asia and South Pacific Design Automation Conference (ASP-DAC '05)*. New York, NY, USA : ACM, 2005. – ISBN 0–7803–8737–6, S. 661–664

[Sil07] Sill, Frank: *Untersuchung und Reduzierung des Leckstroms integrierter Schaltungen in Nanometer-Technologien bei konstanten Performanceanforderungen*, Universität Rostock, Diss., 2007. http://www.grin.com/e-book/119167/

[SKB02] Shang, Li ; Kaviani, Alireza S. ; Bathala, Kusuma: Dynamic power consumption in Virtex-II FPGA family. In: *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays (FPGA '02)*. New York, NY, USA : ACM, 2002. – ISBN 1–58113–452–5, S. 157–164

[SKO07] Sakurai, K. ; Kyo, S. ; Okazaki, S.: Overtaking Vehicle Detection Method and Its Implementation Using IMAPCAR Highly Parallel Image Processor. In: *Proceedings of the IAPR Conference on Machine Vision Applications (IAPR MVA 2007)*, 2007. – ISBN 978–4–901122–07–8, 579-582

[SPA+08] Shelburne, M. ; Patterson, C. ; Athanas, P. ; Jones, M. ; Martin, B. ; Fong, R.: Metawire: Using FPGA configuration circuitry to emulate a Network-on-Chip. In: *Proceedings of FPL '08, Heidelberg, Germany*, 2008, 257–262

[SS82] Storer, J.A. ; Szymanski, T.G.: Data compression via textual substitution. In: *Journal of the ACM (JACM)* 29 (1982), October, Nr. 4, 928–951. `http://portal.acm.org/citation.cfm?id=322346`

[Ste04] Stein, F.: Efficient Computation of Optical Flow Using the Census Transform. In: *Proceedings of 26th DAGM Symposium, August 30 - September 1, 2004, Tübingen, Germany*, 2004, 79-86

[Ste09] Stechele, Walter: *Skriptum zur Vorlesung Integrierte Schaltungen 1*. 2009. – Lehrstuhl für Integrierte Systeme, Technishce Universität München

[STM06] STMicroelectronics (Hrsg.): *Driving assistance single camera based System-on-Chip*. STMicroelectronics, October 2006. `www.st.com/stonline/products/promlit/pdf/fleye1006.pdf`

[TCG08] Teixeira, L. P. ; Celes, W. ; Gattass, M.: Accelerated Corner-Detector Algorithms. In: *BMVC08*, 2008

[Tec07] Techmer, A.: Application Development of Camera-based Driver Assistance Systems on a Programmable Multi-Processor Architecture. In: *Proceedings of the IEEE Intelligent Vehicles Symposium (IV'07)*, 2007, S. 1211–1216

[THAE00] Torres-Huitzil, C. ; Arias-Estrada, M.: An FPGA Architecture for High Speed Edge and Corner Detection. In: *CAMP '00: Proceedings of the Fifth IEEE International Workshop on Computer Architectures for Machine Perception (CAMP'00)*. Washington, DC, USA : IEEE Computer Society, 2000. – ISBN 0–7695–0740–9, S. 112–116

[TKR+06] Tuan, T. ; Kao, S. ; Rahman, A. ; Das, S. ; Trimberger, S.: A 90nm low-power FPGA for battery-powered applications. In: *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays (FPGA '06)*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–292–5, 3–11

[TL03] Tuan, T. ; Lai, B.: Leakage Power Analysis of a 90nm FPGA. In: *Proceesings of the IEEE Custom Integrated Circuits Conference*, 2003, 57–60

[TVCSC05] Toledo, A. ; Vicente-Chicote, C. ; Suardíaz, J. ; Cuenca, S.: Xilinx System Generator Based HW Components for Rapid Prototyping of Computer Vision SW/HW Systems. In: *Second Iberian Conference on Pattern Recognition and Image Analysis (IbPRIA 2005)*, 2005, S. 667–674

[Uhm08] Uhm, M.: Multi-Mode Basestation Common Platform Software Defined Radio: Whats In A Name? In: *Proceedings of the Software Digital Radio,TechnicalConference (SDR'08)*, 2008

[UW07] Urbanski, K. ; Woitowitz, R.: *Digitaltechnik: Ein Lehr- und Übungsbuch*. 5. Springer, 2007. – ISBN 978–3540736721

[VWG+04] Vassiliadis, S. ; Wong, S. ; Gaydadjiev, G. ; Bertels, K. ; Kuzmanov, G. ; Panainte, E. M.: The Molen Polymorphic Processor. In: *IEEE TRANSACTIONS ON COMPUTERS* 53 (2004), November, Nr. 11, 1363–1375. http://ce.et.tudelft.nl/publicationfiles/885_2_The%20MOLEN%20Polymorphic%20Processor.pdf

[Wah67] Wahlstrom, S.: Programmable Logic Arrays - Cheaper by the Millions. In: *Electronics* 40 (1967), December, Nr. 25, S. 90–95

[Wan98] Wannemacher, Markus: *Das FPGA-Kochbuch*. MITP-Verlag, 1998. – ISBN 3–8266–2712–1

[WD04] Wang, W. ; Dony, R. D.: Evaluation of image corner detectors for hardware implementation. In: *Canadian Conference on Electrical and Computer Engineering* Bd. 3, 2004. – ISSN 0840–7789, S. 1285–1288

[WH97] Woodfill, J. ; Herzen, B. V.: Real-time stereo vision on the PARTS reconfigurable computer. In: *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97)*. Washington, DC, USA : IEEE Computer Society, 1997. – ISBN 0–8186–8159–4, S. 201

[WHW09] Winner, H. ; Hakuli, S. ; Wolf, G.: *Handbuch Fahrerassistenzsysteme: Grundlagen, Komponenten und Systeme für aktive Sicherheit und Komfort.* Wiesbaden : Vieweg+Teubner Verlag / GWV Fachverlage GmbH, 2009

[Wil09] Wilson, Ron: Xilinx FPGA introductions hint at new realities. In: *Electronic Design, Strategy, News* (2009). http://www.edn.com/article/459148-Xilinx_FPGA_introductions_hint_at_new_realities.php

[WLN07] Wei, Z. ; Lee, D.-J. ; Nelson, B. E.: FPGA-based Real-time Optical Flow Algorithm Design and Implementation. In: *Journal of Multimedia* 2 (2007), Nr. 5, S. 38–45

[Xil04] Xilinx ; Xilinx Inc. (Hrsg.): *Two Flows for Partial Reconfiguration: Module Based or Difference Based*. v1.2. 2100 Logic Drive, San Jose CA 95124: Xilinx Inc., September 9^{th} 2004. http://www.xilinx.com/support/documentation/application_notes/xapp290.pdf

[Xil05] Xilinx ; Xilinx Inc. (Hrsg.): *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*. v4.0. 2100 Logic Drive, San Jose CA 95124: Xilinx Inc., March 23rd 2005. http://www.xilinx.com/support/documentation/user_guides/ug012.pdf

[Xil06a] Xilinx (Hrsg.): *PLB Double Data Rate (DDR) Synchronous DRAM (SDRAM) Controller*. 2.00a. Xilinx, November 1st 2006

[Xil06b] Xilinx ; Xilinx Inc. (Hrsg.): *PLB IPIF*. v2.02a. 2100 Logic Drive, San Jose CA 95124: Xilinx Inc., October 30th 2006. http://www.xilinx.com/support/documentation/ip_documentation/plb_ipif.pdf

[Xil07] Xilinx ; Xilinx Inc. (Hrsg.): *Virtex-4 Configuration Guide*. v1.5. 2100 Logic Drive, San Jose CA 95124: Xilinx Inc., January 12th 2007. http://www.xilinx.com/support/documentation/user_guides/ug071.pdf

[Xil08a] Xilinx ; Xilinx Inc. (Hrsg.): *Development System Reference Guide*. 10.1. 2100 Logic Drive, San Jose CA 95124: Xilinx Inc., 2008. http://www.xilinx.com/itp/xilinx10/books/docs/dev/dev.pdf

[Xil08b] Xilinx ; Xilinx Inc. (Hrsg.): *Spartan-3 FPGA Family Advanced Configuration Architecture*. v1.1. 2100 Logic Drive, San Jose CA 95124: Xilinx Inc., June 25th 2008. http://www.xilinx.com/support/documentation/application_notes/xapp452.pdf

[Xil08c] Xilinx ; Xilinx Inc. (Hrsg.): *Spartan and Spartan-XL FPGA Families Data Sheet*. v1.8. 2100 Logic Drive, San Jose CA 95124: Xilinx Inc., June 26th 2008. http://www.xilinx.com/support/documentation/user_guides/ug702.pdf

[Xil09a] Xilinx ; Xilinx Inc. (Hrsg.): *ML505/ML506/ML507 Evaluation Platform User Guide*. v3.1.1. 2100 Logic Drive, San Jose CA 95124: Xilinx Inc., October 17*th* 2009. http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf

[Xil09b] Xilinx ; Xilinx Inc. (Hrsg.): *Multi-Port Memory Controller (MPMC)*. v6.00a). 2100 Logic Drive, San Jose CA 95124: Xilinx Inc., December 2nd 2009. http://www.xilinx.com/support/documentation/ip_documentation/mpmc.pdf

[Xil09c] Xilinx ; Xilinx Inc. (Hrsg.): *Processor Local Bus (PLB) v4.6*. 1.04a. 2100 Logic Drive, San Jose CA 95124: Xilinx Inc., December 2nd 2009. http://www.xilinx.com/support/documentation/user_guides/ug702.pdf

[Xil09d] Xilinx ; Xilinx Inc. (Hrsg.): *Virtex-5 Configuration Guide*. v3.8. 2100 Logic Drive, San Jose CA 95124: Xilinx Inc., August 14^{th} 2009. www.xilinx.com/support/documentation/user_guides/ug191.pdf

[Xil09e] Xilinx ; Xilinx Inc. (Hrsg.): *Virtex-5 FPGA User Guide*. v5.0. 2100 Logic Drive, San Jose CA 95124: Xilinx Inc., June 19^{th} 2009. http://www.xilinx.com/support/documentation/user_guides/ug190.pdf

[Xil10a] Xilinx ; Xilinx Inc. (Hrsg.): *LogiCORE IP Device Control Register Bus (DCR) v2.9*. v1.00b. 2100 Logic Drive, San Jose CA 95124: Xilinx Inc., *April*19^{th} 2010. http://www.xilinx.com/support/documentation/ip_documentation/dcr_v29.pdf

[Xil10b] Xilinx ; Xilinx Inc. (Hrsg.): *LogiCORE IP On-Chip Peripheral Bus V2.0 with OPB Arbiter*. v1.00d. 2100 Logic Drive, San Jose CA 95124: Xilinx Inc., *April*19^{th} 2010. http://www.xilinx.com/support/documentation/ip_documentation/opb_v20.pdf

[Xil10c] Xilinx ; Xilinx Inc. (Hrsg.): *Partial Reconfiguration User Guide*. v12.1. 2100 Logic Drive, San Jose CA 95124: Xilinx Inc., May 3^{rd} 2010. http://www.xilinx.com/support/documentation/user_guides/ug702.pdf

[Xil10d] Xilinx ; Xilinx Inc. (Hrsg.): *Virtex-6 Configuration Guide*. v3.0. 2100 Logic Drive, San Jose CA 95124: Xilinx Inc., January 18^{th} 2010. www.xilinx.com/support/documentation/user_guides/ug360.pdf

[Zau09] Zaunert, Florian: *Simulation und vergleichende elektrische Bewertung von planaren und 3D-MOS-Strukturen mit high-κ Gate Dielektrika*, TU Darmstadt, Diss., 2009. http://deposit.ddb.de/cgi-bin/dokserv?idn=999383647&dok_var=d1&dok_ext=pdf&filename=999383647.pdf

[Zep05] Zeppenfeld, J.: *Design and Implementation of the AddressEngine*. Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Master thesis, December 13^{th} 2005

[Zep07] Zeppenfeld, Johannes: *LIS-IPIF Specification*, April 5^{th} 2007

[ZRM06] Zhang, Y. ; Roivainen, J. ; Mammela, A.: Clock-Gating in FPGAs: A Novel and Comparative Evaluation. In: *Proceedings of the 9th EUROMICRO Conference on Digital System Design (DSD '06)*. Washington, DC, USA : IEEE Computer Society, 2006. – ISBN 0–7695–2609–8, S. 584–590

[ZW94] Zabih, R. ; Woodfill, J.: Non-parametric Local Transforms for Computing Visual Correspondence. In: *Third European Conference on Computer*

Vision (ECCV'94). Secaucus, NJ, USA : Springer-Verlag New York, Inc., May 2-6 1994, S. 151–158

Eigene Veröffentlichungen

[ABM+08] Angermeier, J. ; Batzer, U. ; Majer, M. ; Teich, J. ; Claus, C. ; Stechele, W.: Reconfigurable HW/SW Architecture of a Real-Time Driver Assistance System. In: *Proceedings of ARC'08, London, UK*, Springer-Verlag, 26^{th}-28^{th} March 2008. – ISBN 978–3–540–78609–2, S. 149–159

[ACS08] Alt, N. ; Claus, C. ; Stechele, W.: Hardware/software architecture of an algorithm for vision-based real-time vehicle detection in dark environments. In: *Proceedings of DATE '08*. New York, NY, USA : ACM, 2008. – ISBN 978–3–9810801–3–1, S. 176–181

[ACS10] Aschauer, F. ; Claus, C. ; Stechele, W.: In-flight verification of CCSDS based on-board real-time video compression. In: *Proceedings of the* 61^{st} *International Astronautical Congress (IAC), Prague, Czech Republic*, 2010

[CAAS10] Claus, C. ; Ahmed, R. ; Altenried, F. ; Stechele, W.: Towards rapid dynamic partial reconfiguration in video-based driver assistance systems. In: *6th International Symposium on Applied Reconfigurable Computing (ARC2010), Bangkok, Thailand*, 2010, S. 55–67

[CAS10] Claus, C. ; Altenried, F. ; Stechele, W.: Dynamic Partial Reconfiguration of Xilinx FPGAs Lets Systems Adapt on the Fly. In: *XCELL Journal* 70 (2010), S. 18–23

[CHRS09] Claus, C. ; Huitl, R. ; Rausch, J. ; Stechele, W.: Optimizing the SUSAN corner detection algorithm for a high speed FPGA implementation. In: *Proceedings of FPL '09, Prague, Czech Republic*, 2009, S. 138–145

[CLJS09] Claus, C. ; Laika, A. ; Jia, L. ; Stechele, W.: High performance FPGA based optical flow calculation using the census transformation. In: *Proceedings of IV '09, Xi'an, China*, 2009

[CMS06] Claus, C. ; Müller, F. H. ; Stechele, W.: Combitgen: A new approach for creating partial bitstreams in Virtex-II Pro devices. In: *Proceedings of the Workshop on Dynamically Reconfigurable Systems (ARCS '06), Frankfurt a. M., Germany*, 2006, S. 122–131

[CMZS07] Claus, C. ; Müller, F. H. ; Zeppenfeld, J. ; Stechele, W.: A new framework to accelerate Virtex-II Pro dynamic partial self-reconfiguration. In: *Proceedings of IPDPS '07, Long Beach, California*, 2007, S. 1–7

[CS09] Claus, C. ; Stechele, W.: *AutoVision - Reconfigurable Hardware Acceleration for Video-Based Driver Assistance*. Springer-Verlag, 2009 (Dynamically Reconfigurable Systems: Architectures, Design, Methods and Applications, M. Platzner, J. Teich, N. Wehn (Eds.)). – 375–393 S. – ISBN 978–90–481–3484–7

[CSH07] Claus, C. ; Stechele, W. ; Herkersdorf, A.: AutoVision - a run-time reconfigurable MPSoC architecture for future driver assistance systems. In: *it-Journal* 49 (2007), Nr. 3, S. 181–187

[CSK+08] Claus, C. ; Stechele, W. ; Kovatsch, M. ; Angermeier, J. ; Teich, J.: A comparison of embedded reconfigurable video-processing architectures. In: *Proceedings of FPL '08, Heidelberg, Germany*, 2008, S. 587–590

[CSS06] Claus, C. ; Shin, H. C. ; Stechele, W.: Tunnel Entrance Recognition for video-based Driver Assistance Systems. In: *Proceedings of International Conference on Systems (IWSSIP 2006), Signals and Image Processing, Budapest, Hungary*, 2006, S. 419–422

[CZH+07] Claus, C. ; Zhang, B. ; Hübner, M. ; Schmutzler, C. ; Becker, J. ; Stechele, W.: An XDL-based busmacro generator for customizable communication interfaces for dynamically and partially reconfigurable systems. In: *Workshop on Reconfigurable Computing Education at ISVLSI '07, Porto Alegre, Brazil*, IEEE Computer Society, 2007

[CZMS07] Claus, C. ; Zeppenfeld, J. ; Müller, F. ; Stechele, W.: Using partial-run-time reconfigurable hardware to accelerate video processing in driver assistance systems. In: *Proceedings of DATE '07, Nice, France*, 2007, S. 498–503

[CZS+08] Claus, C. ; Zhang, B. ; Stechele, W. ; Braun, L. ; Hübner, M. ; Becker, J.: A multi-platform controller allowing for maximum Dynamic Partial Reconfiguration throughput. In: *Proceedings FPL '08, Heidelberg, Germany*, 2008, S. 535–538

[HBB+07] Hübner, M. ; Braun, L. ; Becker, J. ; Claus, C. ; Stechele, W.: Physical Configuration On-Line Visualization of Xilinx Virtex-IIFPGAs. In: *Proceedings of ISVLSI '07, Porto Alegre, Brazil*. Los Alamitos, CA, USA : IEEE Computer Society, 2007, S. 41–46

[HCM+06] Herkersdorf, A. ; Claus, C. ; Meitinger, M. ; Ohlendorf, R. ; Wild, T.: Reconfigurable Processing Units vs. Reconfigurable Interconnects. In: *Dynamically Reconfigurable Architectures*, 2006 (Dagstuhl Seminar Proceedings 06141). – ISSN 1862–4405

[IACH08] Ihmig, M. ; Alt, N. ; Claus, C. ; Herkersdorf, A.: Resource-efficient Sequential Architecture for FPGA-based DAB Receiver. In: *Workshop zu Software Radio (WSR 08), Karlsruhe, Germany*, 2008

[LPC+10] Laika, A. ; Paul, J. ; Claus, C. ; Stechele, W. ; Auf, A. El S. ; Maehle, E.: FPGA-based Real-time Moving Object Detection for Walking Robots. In: *Proceedings of the 8th IEEE International Workshop on Safety, Security and Rescue Robotics (SSRR 2010), Bremen, Germany*, 2010

[SCL10] Stechele, W. ; Claus, C. ; Laika, A.: Lessons Learned from last 4 Years of Reconfigurable Computing. In: *Dagstuhl Seminar Proceedings 10281 on Dynamically Reconfigurable Architectures, Dagstuhl, Germany*, 2010

[VICS06] Vuletic, Miljan ; Ienne, Paolo ; Claus, Christopher ; Stechele, Walter: Multithreaded virtual-memory-enabled reconfigurable hardware accelerators. In: *Proceedings of IEEE International Conference on Field Programmable Technology (FPT 2006)*, 2006, S. 197–204

Betreute Arbeiten

[Ahm09] Ahmed, R.: *Intra-Video Frame Dynamic Partial reconfiguration of Image Processing Accelerators.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Master thesis, September 30^{th} 2009. – Master thesis

[Ali09] Ali, S. A.: *Taillight and headlamp detection on low cost FPGA devices.* Arcisstr. 21, 80280 München, Lehrstuhl für Mikroelektronische Systeme (MES), Technische Universität Darmstadt, Master thesis, December 31^{st} 2009. – Master thesis

[Alt06] Alt, N.: *TaillightEngine - Design und Implementierung.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Bachelor thesis, August 14^{th} 2006. – , Bachelor thesis

[Alt08] Alt, N.: *Optimal Template Selection for Visual Tracking.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Master thesis, December 31^{st} 2008. – Master thesis

[Alt09a] Altenried, F.: *Design, implementation and evaluation of a Multi-Port Memory Controller Interface.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Bachelor thesis, February 16^{th} 2009. – Bachelor thesis

[Alt09b] Altenried, F.: *Time-sharing of hardware resources for image processing accelerators using Dynamic Partial Reconfiguration.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Master thesis, November 15^{th} 2009. – Master thesis

[Asc09] Aschauer, F.: *Video compression system for space applications.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Master thesis, November 15^{th} 2009. – Master thesis

[Bra06] Braun, L.: *Visualisierung von dynamischer und partieller Rekonfiguration auf Xilinx Virtex-II FPGAs.* Arcisstr. 21, 80280 München, Institut für Technik der Informationsverarbeitung (ITIV), Universität Karlsruhe (TH), Diploma thesis, September 28^{th} 2006. – Diploma thesis

[Che07] Chen, N.: *Using Planahead to Dynamically and Partially reconfigure Hardware Accelerators in the AutoVision System.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Student research project, January 17^{th} 2007. – Student research project

[Che09] Chen, Z.: *Implementation of a Multi-Processor Based Dynamic Partial Reconfigurable System.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Bachelor thesis, April 11^{th} 2009. – Bachelor thesis

[Cra06] Cramm, I.: *Entwicklung und Implementierung einer EdgeEngine zur Erkennung von Kanten in zukünftigen Fahrerassistenzsystemen.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Master thesis, April 28^{th} 2006. – , Master thesis

[Gat09] Gatscher, B.: *Design and Implementation of a Digital Video Input Interface for Embedded Video Processing.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Master thesis, May 12^{th} 2009. – Master thesis

[Har08] Hartl, R.: *Fahrzeugsysteme mit FPGA.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Diploma thesis, May 20^{th} 2008. – Diploma thesis

[Hu08] Hu, Y.: *Design, Improvement and Implementation of a video interface for future video-based driver assistance systems.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Master thesis, May 15^{th} 2008. – Master thesis

[Hui09] Huitl, R.: *Development of a hardware accelerator to detect corners in video frames.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Bachelor thesis, August 28^{th} 2009. – Bachelor thesis

[Jia08a] Jia, L.: *Adaption of the AddressEngine to the LIS IPIF.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Bachelor thesis, March 3^{rd} 2008. – Bachelor thesis

[Jia08b] Jia, L.: *FPGA-based estimation of the Optical Flow for automotive Systems.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Master thesis, November 1^{st} 2008. – Master thesis

[Kie07] Kiessling, N.: *Relocating modules in heterogenous coarse grained reconfigurable SRAM based FPGAs.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrier-

te Systeme (LIS), Technische Universität München, Bachelor thesis, November 20^{th} 2007. – Bachelor thesis

[Kil07] Kilinc, F.: *Design and Implementation of the ShapeEngine.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Master thesis, October 9^{th} 2007. – Master thesis

[Lan10] Landgraf, T.: *Optimierung der Platzierungsvisualisierung auf Virtex-II FPGAs.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Bachelor thesis, February 15^{th} 2010. – Bachelor thesis

[Lu09] Lu, J.: *ContrastEngine - A Hardware Accelerator for Image Enhancement.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Bachelor thesis, May 13^{th} 2009. – Bachelor thesis

[May09] Mayer, J.: *Evaluation of the new Achronix Speedster FPGA Generation.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Master thesis, August 31^{st} 2009. – Master thesis

[Mül06] Müller, F. H.: *Partial Dynamic Reconfiguration of FPGAs.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Diploma thesis, January 31^{st} 2006. – , Diploma thesis

[Pol07] Polig, R.: *Modularisierung bestehender Videofilter Engines aus dem AutoVision Design für die Echtzeitbildverarbeitung auf der Erlangen Slot Machine (ESM).* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Student research project, November 30^{th} 2007. – Student research project

[Pol08] Polig, R.: *Accelerating optical mask calculations on a reconfigurable blade.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Diploma thesis, May 20^{th} 2008. – Diploma thesis

[Pom08] Pomsler, D.: *Hardwarebeschleuniger für bildverarbeitende Systeme im automobilen Umfeld.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Diploma thesis, May 13^{th} 2008. – Diploma thesis

[Rau09] Rausch, J.: *Motion detection based on feature points.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Master thesis, April 11^{th} 2009. – Master thesis

[Sch06] Schwarz, D.: *Entwicklung und Bewertung von Verfahren zur Beschleunigung von Bildverarbeitungsalgorithmen zur Fußgängererkennung im Night-Vision Umfeld durch Einsatz von FPGAs.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Diploma thesis, September 15^{th} 2006. – , Diploma thesis

[Sch08] Schropp, S.: *Evaluation and Implementation of an Algorithm for Lossless Image Compression in Soft- and Hardware.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Master thesis, September 29^{th} 2008. – Master thesis

[Ser06] Serra, C. B.: *Dynamic Partial Self Reconfiguration and Hardware ICAP implementation.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Master thesis, August 3^{rd} 2006. – , Master thesis

[Shi06] Shin, H. C.: *Tunnel entrance recognition for video-based driver assistance systems.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Bachelor thesis, June 31^{st} 2006. – , Bachelor thesis

[Tao07] Tao, F.: *Vision based lane boundary detection.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Diploma thesis, October 16^{th} 2007. – Diploma thesis

[You09] Youssef, A.: *Anpassung eines Optical Flow Cores an einen neuartigen Multiport Memory Controller.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Bachelor thesis, February 4^{th} 2009. – Bachelor thesis

[Zai09] Zaib, A.: *Inter-Video Frame Dynamic Partial reconfiguration of Image Processing Accelerators.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Report of internship, October 31^{st} 2009. – Report of internship

[Zha06] Zhang, B.: *A Perl based Bus Macro-Generator on XDL basis for Dynamically and Partially Reconfigurable Systems.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Report of internship, October 31^{st} 2006. – Report of internship

[Zha07] Zhang, B.: *LIS ICAPCtrl - a Multi-Platfrom ICAP Controller for fast dynmaic Partial Self-Reconfiguration.* Arcisstr. 21, 80280 München, Lehrstuhl für Integrierte Systeme (LIS), Technische Universität München, Master thesis, September 8^{th} 2007. – Master thesis