MPC

MULTI PROJEKT CHIP GRUPPE
BADEN-WÜRTTEMBERG

Herausgeber: Hochschule Ulm **Ausgabe:** 45 **ISSN** 1868-9221 **Workshop:** Albstadt-Sigmaringen Februar 2011

Cooperating Organisation
Solid-State Circuit Society Chapter
IEEE German Section

Inhaltsverzeichnis

Tagungsband zum Workshop der Multiprojekt-Chip-Gruppe Baden-Württemberg

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie.

Herausgeber: Gerhard Forster, Hochschule Ulm, Prittwitzstraße 10, D-89075 Ulm

Diesen Workshopband und alle bisherigen Bände finden Sie im Internet unter:

http://www.mpc.belwue.de

High-Voltage Structured ASICs for Industrial Applications - A Single Chip Solution

Yipin Zhang, Cor Scherjon

***Abstract*—This paper presents a novel, low-cost single chip solution optimized for the integration of industrial electronic systems with low to medium production volumes, where both analog/mixed-signal and smart power capabilities are required, by using the novel high-voltage structured ASIC, a hybrid integration of low-voltage structured ASIC and smart power functions. Construction of the high-voltage structured ASIC platform is detailed. System development approaches basing on the platform are described and benefits of using this platform are analyzed.**

***Index Terms*—HV, Mixed-Signal, Structured ASIC, Smart Power IC, SoC, Industrial Application.**

I. INTRODUCTION

In electronic systems, system integration and miniaturization play the key rolls in performance improvement, energy saving and cost reduction. Over the past decades device dimensions and supply voltages of CMOS technologies are steadily scaled down to enable increasingly large scale of system integration. Greatly benefiting from using submicron devices geometries todays highly complex electronic systems with millions of transistors can be integrated on a single chip.

Most electronic systems can be cost-efficiently integrated by one of the three primary integration approaches: the application specific integrated circuit (ASIC) approach, the field programmable gate array (FPGA) approach or the intermediate approach of structured ASIC. ASICs provide the largest scale of integration and highest performance. But due to the high research, designing and testing costs ASICs are merely economical for high volume products, where the high non-recurring engineering (NRE) costs can be compensated by reduced unit variable costs. Offering high flexibility, low NRE costs and short design time, FPGAs provide a good solution for low volume products and prototyping. However realizable functionalities and reachable performance by FPGA are strongly limited due to its digital nature. Structured ASIC is an intermediate technology between ASIC and FPGA, offering high performance of ASICs and low NRE costs of FPGAs, allows products to be introduced quickly to market at lower cost and with less design effort.

However, some systems for industrial applications, for instance the industrial automation control systems, can not be cost efficiently integrated by one of the three approaches mentioned above due to their relatively small production volumes, huge varieties of functionalities and large differences in operating voltages of in-system components. A typical industrial automation control system is exposed in Fig. 1, where sensor signals have to be conditioned by high performance analog circuits, the digitized signals are processed by data processing component and finally the low-voltage output signals must be transferred and amplified by mixed-voltage interface circuits into high-voltage high-current outputs for the actuators and/or control networks, which are often required to operate at high supply voltages. Additionally, a power management device has to be employed to supply the low-voltage components. Currently integration of many systems for industrial applications still hesitates at board level, where each function is carried out by a purpose specific IC. This low-grade integration leads to degraded performance, low energy efficiency, bulky construction and high cost.

Focusing on improvement of system integration for industrial applications the Institute for Microelectronics Stuttgart (IMS-CHIPS) launches a hybrid-integrated high-voltage structured ASIC platform which combines the established IMS GateForest™ low-voltage structured ASIC technology and a flexibly configurable custom designed integrated smart power core. This enables the monolithic integration of frequently utilized components for industrial applications, e.g. high performance analog circuits, digital data processing units, power management devices and high-voltage high-current IO interfaces, on a single chip. Fig. 2 shows an application example of the single chip solution for an industrial automation control system by using the IMS high-voltage structured ASIC.

Institute for Microelectronics Stuttgart, Allmandring 30a, 70569 Stuttgart, Tel: 0 711 / 21 855 -10, E-Mail: info@ims-chips.de

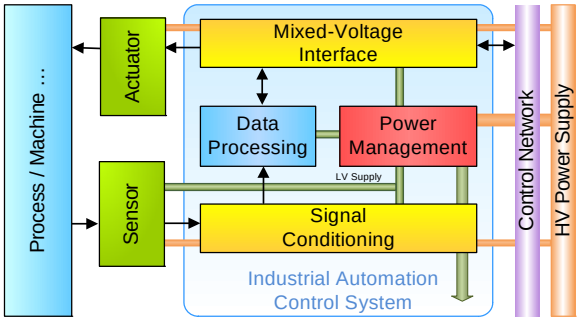


Fig.1 Typical industrial automation system.

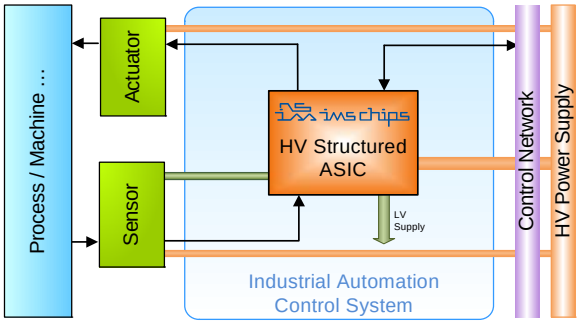


Fig. 2: Single chip solution for industrial automation control system by using IMS high-voltage structured ASIC.

II. The HV Structured ASIC Platform

Fig. 3 shows the IMS high-voltage structured ASIC master chip, which consists of a low-voltage analog core, a low-voltage digital core and a smart power core. Using the low-voltage analog and digital core application and customer specific mixed-signal circuits like signal conditioning circuits and signal processing circuits can be developed. The smart power core contains a power management unit and an interface unit, which again consists of 10 channels of high-voltage high-current smart IOs. The power management unit can both supply all on-chip low-voltage components and optional extern components, such as sensors, microcontrollers and/or storage devices. The 10 channel smart IOs offer numerous configuration possibilities for a broad spectrum of industrial applications.

A Structured ASIC based Mixed-Signal Solution

Fig. 4. illustrates a general development approach of mixed-signal circuits using the IMS structured ASIC technology. Master chips consisting arrays of unit devices are pre-manufactured in high volume to achieve low piece-cost. Basing on the master chips application and costumer specific circuits can be individually realized by personalized generation of the metal interconnections.

The analog core is an array of analog unit cells, which again consists of unit resistors, capacitors and

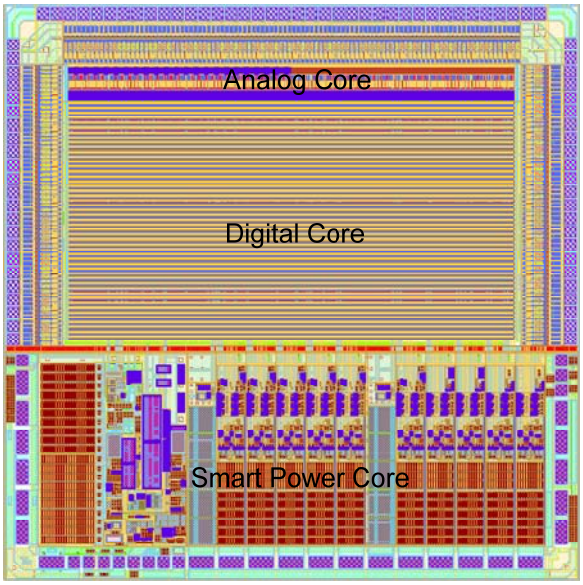


Fig. 3: The IMS HV structured ASIC master chip

transistors. The detailed construction of the analog core is shown in Fig. 5. Combining unit devices by means of serial and/or parallel connections desired device sizes can be obtained. Besides the flexibility in sizing the uniformity in every device type concurrently improves the device matching characteristic, which plays a crucial roll in high performance analog circuits. Basing on the analog core a wide range of analog circuits can be realized to have equivalent performance and area-demand of full-custom designed analog circuits

The digital core, as shown in Fig. 6, is constructed by alternating rows of unit PMOS and NMOS transistors optimized for time and area efficient implementation of CMOS digital circuits. By employing the established IMS semi-custom design flow highly complex digital signal processing systems can be elegantly and time-efficiently developed using high-level hardware description languages (HDL), like VHDL or Verilog HDL. Subsequently synthesized designs can be physically implemented with ease by using industry standard place-and-route tools.

Technical parameters and key features of the low-voltage cores are summarized below:

- 5V nominal supply voltage
- Low parasitic capacitances
- resistance to latch-up owing to SOI technology
- 15 000 (NAND2) equivalent gates
- 76 general purpose IO pads, configurable into direct, input, output and/or bidirectional pads
- Configurable current driving strength from 4mA to 12mA per output pad
- Pull-down / pull-up options for input pads
- Integrated ESD protection.

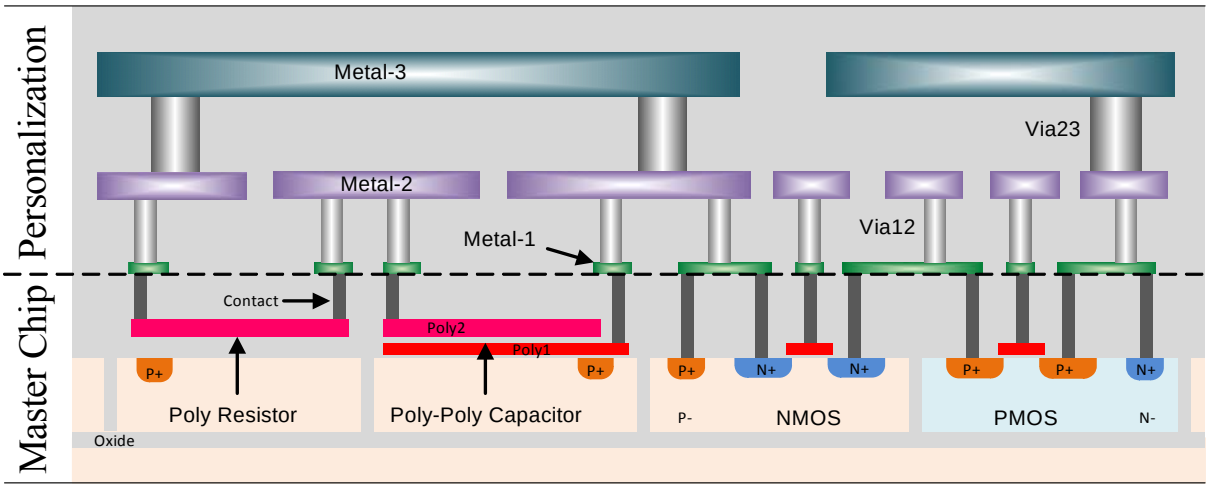


Fig. 4: Application and costumer specific chip personalization basing on IMS structured ASIC master chip

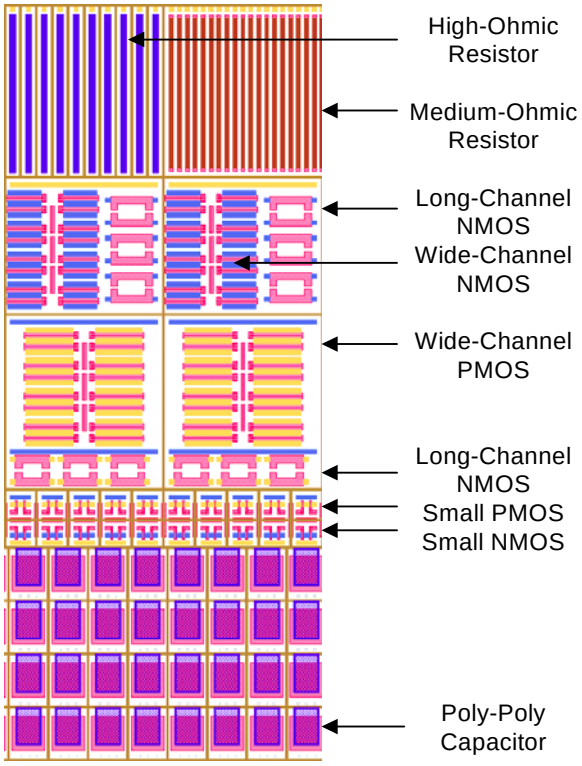


Fig. 5: Details of the analog core.

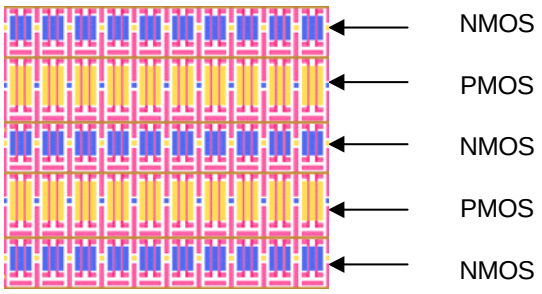


Fig. 6: Details of the digital core.

B. Smart Power Core Based High-Voltage Solutions

Industrial electronic systems are generally supplied with high voltages to power high-voltage components like motors and relays. On the other hand large scale integrated signal processing ICs are required to operate with low supply voltages. Due to the different component supply voltages power management components have to be employed to manage the power supply issues. Because high-voltage supplied power management components are not compatible with low-voltage CMOS technologies, typically discrete power management ICs are used.

Industrial electronic systems usually have to operate in noisy environments, where distributed components, sub-systems or systems have to communicate through networks where signals with relatively high voltage levels are used in order to guarantee sufficient signal-to-noise ratios securing immaculate data transmissions. Due to the different operating voltages between communication network and signal processing components mixed-voltage interfaces are required to convert signals between the voltage domains. Both for obtaining adequately high data rate and for driving the actuators, the interface components are obliged to provide satisfactory current driving capabilities. Due to the low-voltage CMOS incompatibility of the interface circuits discrete interface ICs are usually utilized.

Power management components and high-voltage, high-current interface components are two of the most frequently employed component categories, which can not be cost-efficiently integrated with the signal processing components on a single chip. Using of discrete ICs leads to bulky system construction and high overall costs. The high-voltage structured ASIC platform exactly fills this integration gap by enabling monolithic integrations of power management, high-

voltage, high-current interface components and the low-voltage structured ASIC on a single chip.

A block diagram of the power management unit is shown in Fig. 7. The power management unit consists of a synchronous DC-DC converter and a configurable linear voltage regulator. Key features and parameters of the power management unit are listed below:

- Wide DC input voltage range from 8V to 40V
- High conversion efficiency up to 90%
- Direct DC output at 5V and up to 500mA
- Post-regulated DC output, configurable from 1V to 4.5V and up to 100mA
- Low output voltage ripple of <2% with µF-range capacitances
- Integrated switching transistors, oscillator and reference voltage generators
- Two step over-temperature protection (warning and shutdown)
- Integrated under-voltage, over-load and ESD protections.

A block diagram of the interface unit is shown in Fig. 8. The interface unit consists of 10 channels of high-voltage, high-current smart IOs. The key features and parameters of the interface unit are:

- High operating voltage up to 40V
- High driving capability of 150mA DC current per IO
- Independent *ENABLE*s for low-side drivers and high-side drivers allowing flexible configurations into low-side switches, high-side switches, half bridges, H-bridges and/or three phase inverters
- Wide receiving range high-voltage receivers
- Programmable pull-down or pull-up current up to 2mA per IO
- Integrated two step over-temperature protection (high-temperature warning and emergency shut-down) against thermal destruction of the chip.
- Integrated over-current protection against damages caused by excessive current e.g. over-heating, electron-migration, or non-invertible breakdowns.
- Integrated ESD protection and freewheeling function.

III. Conclusion

Due to small production volumes, huge varieties of functionalities and large difference in operating voltages of in-system components, systems for industrial applications often can not be economically integrated at chip level. System constructions using discrete components lead to degraded performance and high costs. The IMS high-voltage structured ASIC offers peak performance of an ASIC, low NRE of a FPGA and functionalities beyond FPGA. By using the IMS high-voltage structured ASIC platform complete

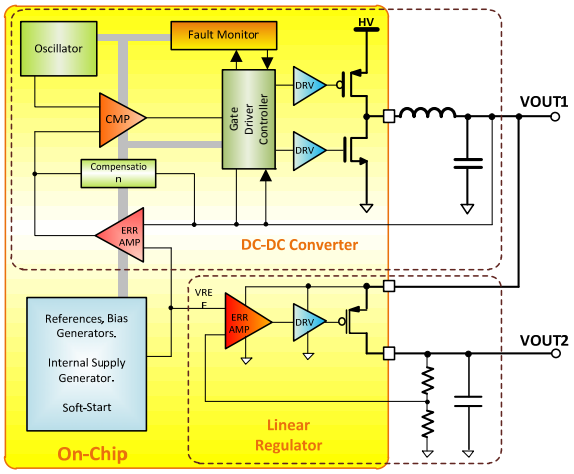


Fig. 7: Block diagram of the power management unit.

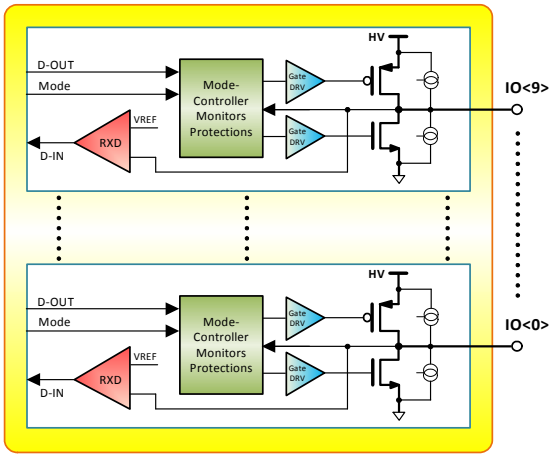


Fig. 8: Block diagram of the interface unit.

systems for industrial applications including sensor signal conditioning, data processing, actuator driving and power management can be implemented on a single chip with small board area demand, good signal integrity, high performance, low overall cost and quick turnaround time.

Combining the low-voltage structured ASIC technology for high performance analog/mixed-signal processing with a smart power core for efficient power management, mixed-voltage signal conversion and high-voltage, high-current driving, the high-voltage structured ASIC platform provides a optimal single chip solution for low to medium volume industrial application systems, where both mixed-signal and smart power capabilities are requisite.

Acknowledgement

The authors gratefully acknowledge the financial support of the *Deutsche Forschungsgesellschaft fuer Automatisierung und Mikroelektronik e.V. (DFAM).* We'd also like to thank *Telefunken Semiconductors GmbH* for constructive technological cooperation.

32 Bit Softcore Sirius Hulk mit Harvard Architektur und Double Cache

Sebastian Stickel, Dirk Jansen, Florian Zowislok, Marc Durrenberger

***Zusammenfassung*—Auf Basis der Softcore Sirius Janus wurden zur weiteren Beschleunigung der Ausführung interne Strukturen optimiert und eine Harvard Architektur mit getrenntem Daten- und Instructioncache realisiert. Die Implementierung bezüglich Startup, Befehlsausführung und Anbindung eines DDR-SDRAM-Controllers werden dargestellt.**

***Schlüsselwörter*—Sirius, Softcore, Harvard Architektur, Cache, DDR-SDRAM, FPGA**

I. Einleitung

Das ASIC Design Center der Hochschule Offenburg entwirft und fertigt seit vielen Jahren Softcores in Form von Studentenprojekten. Eine großes Projekt der letzten Jahre war der RISC Prozessor Sirius. Als RISC Softcore in v. Neumann Architektur mit einem 16 Bit breiten Datenbus, verfügt er über einen stark eingeschränkten Befehlssatz der quasi nur die Befehle enthält, die ein C-Compiler zum hocheffizienten Übersetzen in die Assemblerebene benötigt. Er ist in der Lage, sämtliche Register-Register Operationen in einem Taktzyklus auszuführen. Die meisten Komplexbefehle werden ebenfalls in einem, maximal jedoch in 5 Zyklen ausgeführt. Er kann im 16- oder im 32 Bit Modus betrieben werden [1]. Das Design des Sirius wurde in zwei Richtungen weiterentwickelt, der Name *„Sirius“* wurde zum Familienname und die ursprüngliche Core wurde in *„Sirius Janus“* umbenannt.

„Sirius Tiny“ ist eine noch kompaktere Variante des ursprünglichen Entwurfs. Seine Busbreiten sowie die interne Verarbeitung wurden konsequent auf 16 Bit reduziert und der Befehlssatz wurde weiter eingeschränkt. Der Tiny wird an der Hochschule Offenburg auch zu Lehrzwecken eingesetzt. In einem Praktikum haben Studenten hierbei die Möglichkeit, den Kern des Tiny selbst in VHDL zu erzeugen und in Betrieb zu nehmen. Hierdurch bietet sich ihnen die Gelegenheit, einen tiefen Einblick in die Architektur moderner RISC Prozessorsysteme zu bekommen.

Sebastian Stickel (sebastian.stickel@fh-offenburg.de) und Dirk Jansen (d.jansen@fh-offenburg.de), Mitglieder des Instituts für angewandte Forschung der Hochschule Offenburg

Abbildung 1: StudPod mit Sirius Janus ASIC

Die Entwicklung des *„Sirius Hulk“* hatte nicht das Ziel den kleinsten, aber den leistungsfähigsten Kern der Familie hervorzubringen und aus den Designansätzen des Janus die maximal mögliche Rechenleistung herauszuholen. Er ist ein reiner 32 Bit Kern und die konsequente Weiterentwicklung der Ansätze des Sirius Janus. Der aktuelle Stand dieser Entwicklung soll hier vorgestellt werden.

II. Struktur des Janus

Als Grundlage für die Softcore Hulk diente die Architektur des Janus. Deshalb soll an dieser Stelle auf die letzten Entwicklungen an diesem Kern kurz eingegangen werden. Der grundsätzliche Aufbau des Janus entspricht einer v. Neumann Architektur mit dreistufiger Pipeline, d.h. er hat einen 16 Bit breiten Bus zur Anbindung von Speicher. Er ist jedoch in der Lage, diesen in einem 32 Bit Adressraum anzusprechen und kann somit bis zu 4 GB adressieren. Das Schreiben von Daten auf den Bus benötigt vier Taktzyklen, das Lesen ebenfalls. Der Kern wurde mehrfach in ASICs eingesetzt, wofür er mit SRAM Speicher ausgestattet wurde. 2009 wurde an der Hochschule auf Basis des Sirius Kerns ein PDA entwickelt. Dieser ist u.a. mit einem Touchscreen, einer SD-Karte und einem selbstentwickelten Betriebssystem ausgestattet und soll die Möglichkeiten der Sirius Plattform demonstrieren. (Abbildung 1) Zum aktuellen Zeitpunkt benötigt das Sirius OS mit FAT32 Dateisystem und ausgeführter Software ca. 100 kB SRAM Speicher. Da somit die 64 kB interner Speicher des FPGA-PDA und die 32 kB des ASIC-PDA zum Ausführen von Betriebssystem

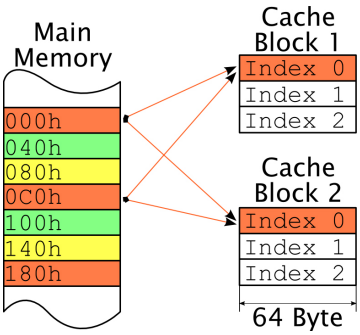


Abbildung 2: Zweifach satzassoziativer Cache

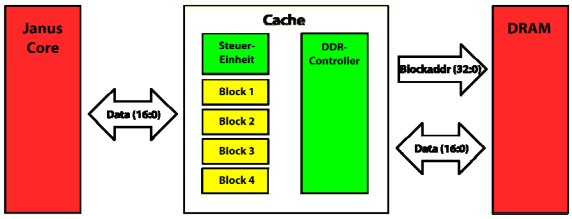


Abbildung 3: Janus mit Cache

und Programmen nicht ausreichten, wurde dieser SRAM durch 2 MB externen SRAM erweitert. Hierdurch wurde die Geschwindigkeit des Systems jedoch stark ausgebremst, weil bei jedem Zugriff auf den langsamen externen Speicher Wartezyklen nötig waren. SRAM ist nicht in beliebiger Größe und Geschwindigkeit verfügbar, relativ teuer und ist im Vergleich zu DRAM sehr stromhungrig. DRAM hingegen ist ein blockorientierter Speicher und benötigt mehrere Takte zum Adressieren eines einzelnen Blocks. Innerhalb dieses Blocks kann dann jedoch in Form eines sogenannten Bursts gelesen oder geschrieben werden, also mit jeder Taktflanke ein Datum verarbeitet werden. Greift man auf DDR-SDRAM zurück, ist sogar das Verarbeiten eines Datums bei jeder Taktflanke möglich, nicht nur bei der steigenden.

Im Gegensatz zu DRAM ist SRAM nicht blockorientiert und deshalb der Zugriff auf eine einzelne Speicherzelle immer gleich schnell, jedoch mit den oben genannten Nachteilen.

III. Aufbau des Cachesystems

A. Organisation des Cache

Um den Speicherzugriff unter Verwendung eines großen DRAM Speichers zu beschleunigen wurde für den Janus ein Cache System entwickelt, dessen grundsätzlichen Aufbau Abbildung 3 zeigt. Im Folgenden soll dieses Cache System kurz erläutert werden, eine ausführliche Beschreibung findet sich in [2] und [3].
Die Grundidee eines solchen Systems ist es, dass häufig genutzte Daten automatisch aus dem schnellen Cache verfügbar sind und selten genutzte im Arbeitsspeicher liegen. Hierbei versucht man die Vorteile des DRAM (sehr schnelle Bursts bei niedrigem

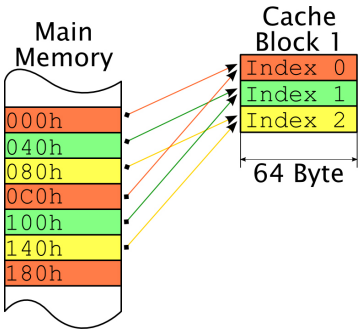


Abbildung 4: Direktverdrahteter Cache

Stromverbrauch und großer Kapazität) und die Vorteile des SRAM (sofortiger Zugriff auf jede Zelle) zu vereinen. Abbildung 4 zeigt die einfachste Variante eines solchen Caches. Der Hauptspeicher ist hier in Pages von 64 Byte gegliedert, genauso wie der Cache. Jede Page im Hauptspeicher ist mit einer Page im Cache fest verdrahtet. Da der Cache im Normalfall nur einen Bruchteil der Speicherkapazität des Hauptspeichers hat, werden jeder Page im Cache mehrere Pages des Hauptspeichers zugewiesen. Ein solcher Cache wird *„direktverdrahtet“* genannt und ist die einfachste Art der Implementierung. Das Gegenstück eines solchen Caches wäre ein sog. vollassoziativer Cache. Hier kann jede Page des Hauptspeichers an einem beliebigen Punkt im Cache abgelegt werden. Dies ist die flexibelste aber auch die am schwierigsten zu implementierende Variante. Sie benötigt eine große Menge an Logik zur Organisation des Caches und lohnt sich nur in seltenen Fällen. Ein möglicher Mittelweg zu diesen beiden Varianten ist ein n-fach satzassoziativer Cache. Hierzu werden mehrere direktverdrahtete Cacheblöcke verwendet. Im Fall eines Schreibvorgangs kann somit nicht mehr wie beim vollassoziativen Cache zwischen allen Pages eines Blocks gewählt werden, sondern nur noch zwischen n Pages in n Cacheblöcken. Abbildung 2 zeigt einen solchen zweifach satzassoziativen Cache. Hierbei halten sich der Implementierungsaufwand und die Größe der Organisationlogik noch in Grenzen, gleichzeitig ist ein wahlweises Schreiben von Daten in zwei verschiedene Pages möglich. Das entworfene System für die Janus Softcore ist als 4-fach satzassoziativer Cache organisiert.

B. Cachestrategien

Es gibt verschiedene Möglichkeiten das Schreiben und Lesen des Caches zu organisieren. Eines haben alle Varianten gemeinsam. Der Cache ist in jedem Fall transparent, also für die Core nicht sichtbar. Er puffert den gesamten Adressraum des Arbeitsspeichers. Sollte die Core lesend oder schreibend ein Datum anfordern das der Cache nicht vorrätig hat, entzieht dieser der Core das „ready“ Signal und friert sie somit ein. Wird ein Datum vom Bus gelesen oder auf ihn geschrieben gibt es jeweils zwei Möglichkeiten: *„Hit“*

Tabelle 1: Cachestrategien

Read Miss	No Read Through
Write Hit	Write Back
Write Miss	Write Allocate

Tabelle 2: Anzahl der nötigen Takte bei Komplexbefehlen, Vergleich zwischen Janus im 32 Bit Modus und Hulk

Befehl		Anzahl Takte	
Opcode	**Erklärung**	**Janus**	**Hulk**
LDX	Speicher lesen	4	2
STX	Speicher schreiben	4	1
PIN	Peripherie lesen	3	2
POT	Peripherie schreiben	3	1
CAL	Unterprog. aufrufen	5	3
RET	Unterprog. verlassen	5	4

oder *„Miss“*. Das heißt, der Cache hat die Daten vorrätig, oder nicht.

- *„Read Hit“*: Die Daten werden ohne Verzögerung aus dem Cache gelesen.
- *„Read Miss“*: Die Core muss angehalten werden. Die entsprechende Page wird in den Cache geladen und anschließend der Read-Hit-Fall ausgeführt.
- *„Write Hit“*: Da die entsprechende Page im Cache vorrätig ist, werden die Daten nur in diesem verändert. Da nun jedoch die Konsistenz zwischen DRAM und Cache nicht mehr gegeben ist, wird die entsprechende Page im Cache mit einem sog. *„Dirty Bit“* markiert.
- *„Write Miss“:* Die Core muss angehalten werden. Die entsprechende Page wird in den Cache geladen. Sind alle für den jeweiligen Speicherbereich im Cache zur Verfügung stehenden Pages mit einem Dirty Bit markiert, muss deren Inhalt zuerst mit dem DRAM abgeglichen werden, bevor die neue Zeile geladen und ihr Inhalt überschrieben werden kann. Anschließend wird in den „Write Hit“ Fall gewechselt.

Tabelle 1 fasst die Cachestrategien beim Lesen wie beim Schreiben auf den Bus zusammen. Falls eine Page verdrängt werden muss um eine neue abzulegen, wird diejenige verworfen, auf die am längsten nicht mehr zugegriffen wurde. Hierzu wird jede Page im Cache mit einem 2 Bit Zähler ausgestattet. Die zuletzt zugegriffene Zeile hat die Bitkombination „00“, die am längsten nicht zugegriffene „11“ Das Verfahren wird ebenfalls in [1] ausführlich erläutert.

IV. Struktur des Sirius Hulk

Hulk als leistungsstärkstes Mitglied der Siriusfamilie wurde auf Grundlage der Erfahrungen bei der Entwicklung des Janus und dessen Architektur entworfen. Hierbei wurde weniger auf ein möglichst kompaktes Design geachtet, als auf möglichst hohe

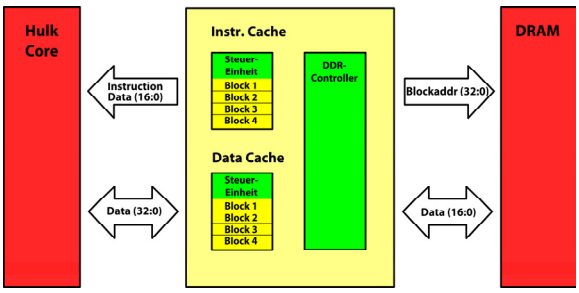


Abbildung 5: Struktur des Bussystems des Hulk

Rechenleistung. Dazu wurden Ansätze implementiert, die für Janus bisher nicht in Frage kamen.

Die von Neumann Architektur hat den entscheidenden Nachteil, dass nicht gleichzeitig ein neuer Befehl geholt (Fetch) und gleichzeitig auf den Speicher zugegriffen werden kann. Bei Befehlen die nicht auf den Bus zugreifen, kann gleichzeitig mit dem Ausführen des Befehls schon der nächste Befehl über den Bus geladen werden (Pipelining). Handelt es sich um einen Befehl der mehr als einen Taktzyklus dauert, z.B. LDI, wird im letzten Teil des Befehls der nächste Befehl am Bus angefordert und steht somit rechtzeitig zur Verfügung um ohne Unterbrechung ausgeführt zu werden. Bei den in Tabelle 2 genannten Befehlen ist dies jedoch nicht möglich, da hier der Bus schon für den Daten-, bzw. Peripheriezugriff belegt ist. Um diese Verzögerung zu vermeiden, wurde die Sirius Hulk Softcore in Harvard Architektur aufgebaut. Sie hat zwei getrennte Busse, für Daten und Instruktionen und ist dadurch auch in der Lage, Speicher und I/O-Zugriffe verzögerungsfrei durchzuführen, siehe Abbildung 5. Da die Hulk Core für die reine 32 Bit Verarbeitung ausgelegt ist, wurde konsequenterweise auch der Datenbus auf 32 Bit verbreitert. Tabelle 2 zeigt die Einsparungen bei den zur Ausführung benötigten Taktzyklen für Komplexbefehle, die auf den Datenbus zugreifen. Die Vergleiche wurden hier mit dem Prozessorkern Janus im 32 Bit Betrieb gemacht.

Beim Instructionbus ist eine Breite von 32 Bit nicht notwendig. Register-Register-Befehle setzen sich weiterhin aus 6 Bit Opcode, 4 Bit Targetregister und 4 Bit Sourceregister zusammen. Die übrigen 2 Bit wurden beim Janus zur Steuerung der Registerbreite genutzt und werden beim Hulk nicht mehr benötigt, da dieser eine feste Registerbreite von 32 Bit verwendet. Weiterhin sind im Instructioncache keine Dirty Bits nötig, da hier seitens der Core keine Schreibzugriffe stattfinden. Der Einsatz einer Harvard Architektur zieht jedoch eine Konsequenz nach sich: Die vorhandene Cachestruktur muss verdoppelt werden, um beide Busse zu puffern. Weiterhin musste die Cachesteuerung bezüglich des DRAM-Zugriffs so modifiziert werden, dass nun zwei Cachespeicher bedient werden können. Die Größe der beiden Caches sollte pro Bus mit 4 Blöcken à 4 kB beibehalten werden, so dass das neue System insgesamt 16 kB Daten-, und 16 kB Instructioncache besitzt.

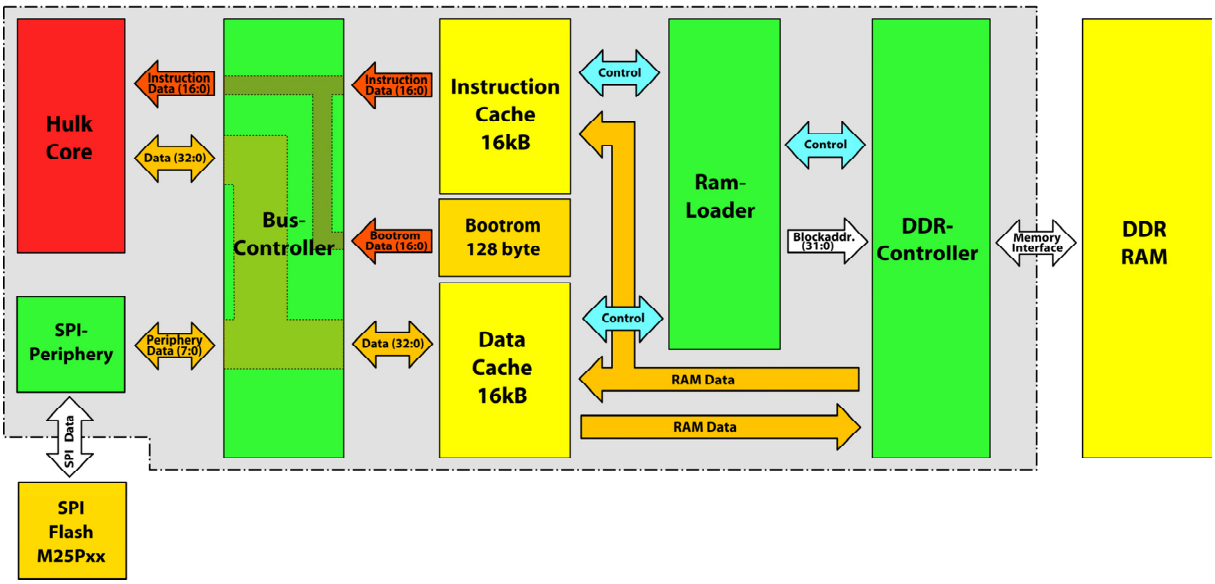


Abbildung 6: Detaillierte Struktur des Peripherie- und Cachesystems beim Hulk

Abbildung 6 zeigt eine detailliertere Übersicht über den Aufbau des Hulk, inklusive der Caches und ihrer Steuerung. Grau hinterlegt ist hier der Teil des Systems, der im FPGA und später innerhalb des ASIC liegt. Gut zu erkennen ist hier auch, wie beide Caches an den DDR-Controller angebunden sind. Hier war keine weitere Logik notwendig, da aus Sicht des DRAM nur der Datencache schreibbar ist. Der RAM-Loader wurde erweitert, da er jetzt den Zugriff zweier Cacheblöcke verwalten muss. Hierzu wurde das System um mehrere Kontrollsignale erweitert, die den exklusiven Zugriff eines Cachebausteins auf den DDR-Controller gewährleisten. Da der RAM-Loader und die Cachesteuerungen auf State Machines basieren, waren die hier notwendigen Änderungen relativ leicht zu implementieren.

V. Bootvorgang

Das folgende Kapitel erläutert, wie der Bootvorgang im Sirius Hulk implementiert wurde. Nach dem Einschalten des Systems sind alle Speicher, Caches wie DRAM noch leer. Um nun ein möglichst einfaches und flexibles Hochfahren des Systems zu erreichen, wurde ein 128 Byte großes Bootrom implementiert. Im aktuellen Stand der Entwicklung ist dieses durch ein weiteres SRAM realisiert, welches über ein *„Memory Initialization File“*, kurz *„mif-File“* bei der Startkonfiguration des FPGA initialisiert wird. Zu einem späteren Zeitpunkt wird dieses durch ein rein kombinatorisches ROM ersetzt, welche auch in einen ASIC implementiert werden kann. Die Verwendung eines generischen SRAM mit mif-File hat den Vorteil, dass es über die JTAG Schnittstelle zur Laufzeit des FPGA gelesen und auch beschrieben werden kann, was das Debugging deutlich vereinfacht. Der im Bootrom hinterlegte Code hat die Aufgabe, den Inhalt des in Abbildung 6 zu erkennenden SPI Flash in den DRAM zu Laden und zur Ausführung zu bringen. Nach einem Reset fragt die Core über den Instructionbus die Adresse 7FFFFF00h an. Der hierauf folgende Adressbereich wird durch den Buscontroller auf das Bootrom abgebildet und die Core führt die hier hinterlegten Instruktionen aus, ohne auf das Cachesystem angewiesen zu sein. Folgende Operationen werden an dieser Stelle z.Zt. ausgeführt:

- Initialisieren der SPI Peripherie.
- Laden eines festgelegten Adressbereichs aus dem angeschlossenen SPI Flashbaustein.
- Auslösen eines *„Flush Befehls“*, der die Konsistenz von DRAM und Datencache erzwingt.
- Einsprung ins Hauptprogramm.

Nach der Initialisierung der SPI Peripherie wird der Inhalt des SPI Flash mithilfe des Peripheriebefehls byteweise in ein Register der Core geladen (PIN) und anschließend durch den Store-Befehl (STX) im Speicher abgelegt. Dies wird wiederholt, bis der gesamte gewünschte Inhalt vom Flash in den Speicher transferiert wurde. Da beim Schreiben des Caches jedoch mit der *„write back“* Strategie gearbeitet wird, die Daten also nur im Cache abgelegt und als *„dirty“* markiert werden, muss im nächsten Schritt durch einen sogenannten *„Flush Befehl“* die Konsistenz zwischen Datencache und DRAM wiederhergestellt werden, damit der auszuführende Code im DRAM zur Verfügung steht. Der Einfachheit halber wird hierzu der gesamte Inhalt des Datencache in das DRAM übertragen. Wenn die Core im Anschluss daran ins Hauptprogramm springt, kann das jetzt im DRAM vorliegende Programm pageweise in den Instructioncache geladen und zur Ausführung gebracht werden. Von hier an werden weder das Bootrom noch der Flush Befehl benötigt, der Cache organisiert sich wie in Kapitel III und ausführlich in [1] und [2] beschrieben

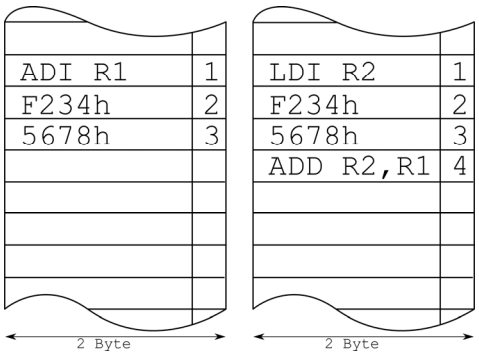


Abbildung 7: Aufbau des Befehls ADI als Hardwarebefehl links und über ein Makro rechts

von selbst. Es ist jedoch jederzeit möglich, den Flush Befehl zur Laufzeit durch einen einfachen Befehl auszulösen um die Konsistenz zwischen Cache und DRAM zu erzwingen, da dieser als Peripheriebaustein implementiert wurde und deshalb per Peripheriebefehl (POT) angesprochen werden kann.

VI. Immediate Befehle

Immediate Befehle sind Instruktionen, die weder auf den Bus zugreifen, noch Register-Register Befehle sind. Als Parameter besitzen sie ein Zielregister und eine Konstante. Zurzeit sind sieben Immediate Befehle implementiert:

- LDI (Load Immediate)
- ADI (Add Immediate)
- SBI (Subtract Immediate)
- ANI (AND Immediate)
- ORI (OR Immediate)
- XRI (XOR Immediate)
- CPI (Compare Immediate)

Als Beispiel sei hier der Befehl *„ADI“* – „Add Immediate“ angeführt: *ADI R_{trgt}, const32*

Hierbei wird zum Inhalt des Registers R_{trgt} der Wert einer 32 Bit Konstante addiert, nicht der Inhalt eines zweiten Registers, wie beim Register-Register Pendant *„ADD“*. Der Unterschied dieser Immediate Befehle ist also, dass die Konstante über den Instructionbus zur Core gelangt weil sie an den Befehl angehängt ist und nicht aus dem Datenspeicher geladen wird. Der Compiler muss hierdurch in vielen Fällen kein zusätzliches Konstantenhandling durchführen. Zu den Konstanten zählen auch Pointer, wodurch eine absolute Adressierung sehr einfach umzusetzen ist. Als weiteres Beispiel sei hier der einmalige Aufruf einer Addition einer Konstante in einem C-Programm genannt. Ohne Immediate Befehle müsste dieser Wert erst aus dem Speicher in ein Register

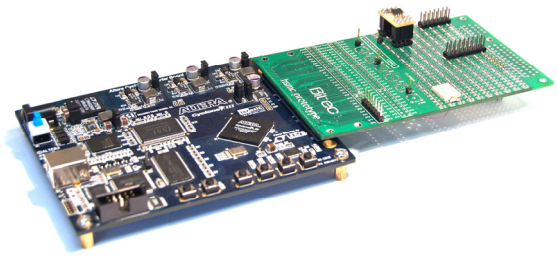


Abbildung 8: Verwendetes Evalboard der Firma Altera mit angesteckter Erweiterungskarte (Altera Cyclone III FPGA Starter Kit)

geladen werden. Mithilfe der Immediate Befehle können solche Speicherzugriffe vermieden werden, da die Konstante direkt am Befehl angehängt wird und über den Instructionbus geladen wird. Abbildung 7 links zeigt den Aufbau der Immediate Befehle. Gezeigt wird hier ein Ausschnitt des Datenstroms am 16 Bit breiten Instructionbus. An dieser Stelle ist auch der Nachteil von Immediate Befehlen erkennbar. Da allein die Konstante 32 Bit lang ist, haben Immediate Befehle eine Gesamtgröße von 6 Byte. Es werden also 3 Takte zum Einlesen des Befehls benötigt, was prinzipiell im Widerspruch zum RISC Konzept steht, jedoch zu sehr kompaktem Code führt.

Im Tiny wurde auf die hardwareseitige Implementierung der Immediate Befehle, außer LDI, verzichtet. Stattdessen wurde mit Makros gearbeitet, die beim Übergang von Assembler auf Maschinencode den Befehl durch ein dem in Abbildung 7 rechts entsprechendes Konstrukt ersetzt. Hier wurde also für die Verkleinerung der Core in Kauf genommen, dass für jeden Immediate Befehl ein LDI plus entsprechenden Register-Register-Befehl benötigt wird, was einen Takt mehr bedeutet. Weiterhin spart die Verwendung von hardwareseitigen Immediate Befehlen 2 Byte Code im Vergleich zur Umsetzung als Makro.

VII. Aktueller Entwicklungsstand

Zum aktuellen Zeitpunkt ist das System in einer Simulation erfolgreich getestet. Der in Kapitel V beschriebene Bootvorgang funktioniert hier, genauso wie das anschließende Ausführen eines Testprogramms, das alle Befehle schrittweise testet und bei einem Fehler abbricht. In dieser Phase wurde unter anderem auch das Interruptsystem getestet, an welchem Modifikationen vorgenommen wurden, welche hier unerwähnt blieben. In der Simulation konnte kein Fehlverhalten festgestellt werden.

Zurzeit wird das System in einem FPGA verifiziert. Hierzu kommt ein Evaluationsboard der Firma Altera, ausgestattet mit einem Cyclone III FPGA zum Einsatz, welches in Abbildung 8 zu sehen ist. Auf der Erweiterungskarte ist z.Zt. nur der SPI Flash untergebracht, weitere Komponenten können so jedoch sehr einfach angebunden werden.

Tabelle 3: Füllgrad des verwendeten FPGA (EP3C25F324)

	Core	Cache	Rest	Gesamt
Anzahl Logikblöcke	10391	5189	4840	20420 (83%)
Größe Speicher	0 kB	33,6 kB	0,4 kB	34 kB (46%)
9 Bit Hardw. Multiplizierer	12	0	0	12 (9%)

Eine vollständige Verifikation im FPGA steht noch aus. Ein kleines Testprogramm wird zwar geladen und auch ausgeführt, jedoch werden hierbei noch nicht alle Befehle getestet. Tabelle 3 zeigt den aktuellen Füllgrad des verwendeten FPGA. Die Größe der Core mit über 10000 Logikblöcken lässt sich durch die Implementierung der Division in Hardware erklären. Hierzu existieren vier Befehle:

- DVU (Division unsigned)
- DVS (Division signed)
- MDU (Modulo unsigned)
- MDS (Modulo signed)

Testweise wurden diese vier Befehle in der ALU durch einfache Additionen ersetzt, um den Bedarf an Logikblöcken für die Division zu ermitteln. Nach dem Ersetzen schrumpfte die Größe der Core auf 5966 Logikblöcke, was noch einem Gesamtfüllgrad von 65% entspricht.

VIII. Ausblick

Nach Abschluss der Verifikation des Designs in Hardware steht in naher Zukunft die Einbindung des Sirius OS an. Hierzu muss u.a. ein erweiterter Bootloader aus dem Flash nachgeladen werden, welcher in der Lage ist, ein Dateisystem von SD-, oder CF-Karte zu lesen (Chainloading). Im Anschluss daran ist geplant, eine neue Hardware für den PDA auf Basis des Hulk zu entwerfen. Auch wäre nach Abschluss der Entwicklungsarbeiten am Kern ein Leistungsvergleich des Hulk mit anderen kommerziellen RISC Architekturen wie z.B. dem ARM Cortex M3 sehr interessant, da dieser nicht unerhebliche Ähnlichkeiten mit der Architektur des Hulk aufweist. Hierzu könnten zum Beispiel Benchmarks wie „Dhrystone“ [4] oder ähnliche genutzt werden.

IX. Fazit

Mit dem Sirius Hulk ist an der Hochschule Offenburg ein hochleistungsfähiges RISC Prozessordesign entstanden, das Techniken einsetzt, wie sie heutzutage in vielen modernen Rechnerarchitekturen zum Einsatz kommen. Hierzu zählen der Einsatz einer Harvard Architektur bei gemeinsamer Nutzung nur eines DDR-SDRAM, die Verwendung getrennter Caches für Daten und Instruktionen und der stark reduzierte Befehlssatz.

Literaturverzeichnis

[1] D. Jansen, N. Fawaz, M. Durrenberger, „A Small High Performance Microprocessor Core SIRIUS for Embedded Low Power Designs, Demonstrated in a Medical Mass Application Of an Electronic Pill (ePille®)“, *Embedded System Design Topic, Techniques and Trends*, ISBN 978-0-387-72257-3, p. 363-372, California, USA, June 2007

[2] F. Zowislok, „Cache-Speicher für den Softprozessor SIRIUS mit DDR-Interface“, *Workshop der Multiprojekt-Chip-Gruppe Baden-Württemberg*, ISSN 1862-7102, Karlsruhe, Juli 2009

[3] F. Zowislok, „Entwicklung eines 4-fach assoziativen Cache-Speichers für ein 32-Bit-Mikrokontrollersystem“ *Masterthesis der Hochschule Offenburg*, 2008/2009

[4] Englischsprachige Wikipedia: „Dhrystone“ http://en.wikipedia.org/wiki/Dhrystone

Sebastian Stickel erhielt den akademischen Grad des Dipl.-Ing. (FH) in Electrical Engineering im Jahr 2009 von der Hochschule Furtwangen. Sein Masterstudium der Elektrotechnik und Informationstechnik an der Hochschule Offenburg wird er 2011 beenden.

Realzeit-Bildverarbeitung auf einem FPGA

Wolfgang Rülling

Zusammenfassung **– Es wird eine FPGA-Implementierung eines One-Pass Algorithmus zur Erkennung von Zusammenhangskomponenten in Bildern vorgestellt. Typischerweise nutzt man solche Algorithmen zur Objekterkennung. Eine Beispielanwendung ist die automatische Erkennung von Verkehrszeichen durch Fahrerassistenzsysteme im Automotive-Bereich.**

Schlüsselwörter **– FPGA, Bildverarbeitung, CCL-Algorithmus, Zusammenhangskomponenten**

I. Einleitung

Bei Bildverarbeitungsanwendungen stellt sich häufig das Problem, Objekte in Bildern zu erkennen. Das kann zum Beispiel die Erkennung von Buchstaben eines Textes sein, oder die Erkennung von Personen auf einem Foto. Im Automotive-Bereich stellen Fahrer-Assistenzsysteme eine interessante Anwendung dar, wenn sie die Bilder einer Frontkamera am Fahrzeug auf Verkehrszeichen hin durchsuchen.

Im Allgemeinen ist die sichere Erkennung komplizierter Objekte eine relativ rechenintensive Arbeit, die für große Bilder sehr lange dauern kann. Deshalb ist man daran interessiert, zunächst durch eine schnelle Vorverarbeitung relevante Bildbereiche zu selektieren, in denen es sich lohnt, aufwändigere Untersuchungen anzustellen. Für eine solche Vorverarbeitung nutzt man die Suche nach Zusammenhangskomponenten.

Sucht man beispielsweise nach Verkehrszeichen mit Geschwindigkeitsbeschränkungen, dann ist bekannt, dass diese Schilder kreisförmig sind und einen roten Rand besitzen. Deshalb sucht man zunächst nach roten zusammenhängenden Bereichen, die etwa genauso breit wie hoch sind und im Inneren ein weißes Gebiet mit schwarzen Objekten enthalten.

Nur für Bildregionen mit diesen Eigenschaften lohnt es sich, aufwändigere Untersuchungen durchzuführen, um festzustellen ob es sich tatsächlich um eine Geschwindigkeitsbeschränkung handelt und um gegebenenfalls die zulässige Maximalgeschwindigkeit auszulesen.

Zur Erkennung von Zusammenhangskomponenten benutzt man meist einen „Connected Component Labelling“ Algorithmus (CCL), der das Bild nacheinander in zwei Durchläufen untersucht und die Speicherung des kompletten Bildes voraussetzt.

In [1] wurde erstmals ein „Single Pass CCL-Algorithmus“ vorgestellt, der nur einen Durchlauf benötigt. Allerdings hängt der Speicherplatzbedarf der ersten FPGA-Implementierung [2] weiterhin linear von der Bildgröße ab. In [3] wurde dann eine Implementierung angegeben, deren Speicherplatz nur noch linear von der Zeilenlänge abhängt, was einer drastischen Speicherreduzierung entspricht.

Unabhängig von diesen Arbeiten wurde in [4] ein ähnlicher Ansatz vorgestellt, der lediglich Informationen über zwei Zeilen des Bildes speichert und der pro Pixel nur konstante Laufzeit benötigte. In der vorliegenden Arbeit soll nun die Implementierung einer Erweiterung dieses Algorithmus vorgestellt werden.

Der neue Algorithmus kann außer horizontal und vertikal benachbarten Pixeln auch diagonale Nachbarschaften untersuchen. Außerdem werden jetzt Farbbilder statt Schwarz-Weiß-Bilder verwendet, so dass gleichzeitig Zusammenhangskomponenten (ZHKs) beliebiger Farben gefunden werden können.

II. Aufgabenstellung

Oft hängt die Rechenzeit eines CCL-Algorithmus von der Anzahl der gefundenen ZHKs, also von der Komplexität des Bildinhalts ab. Beispielsweise wird in [5] ein Verfahren mit einer speziellen Bildcodierung genutzt, das bei typischen Bildern besonders schnell arbeitet, aber in ungünstigen Fällen bedeutend länger braucht.

Um die neue FPGA-Implementierung möglichst

W. Rülling, ruelling@hs-furtwangen.de, ist Professor an der Hochschule Furtwangen, Fakultät Computer & Electrical Engineering, Robert Gerwig Platz 1, 78120 Furtwangen.

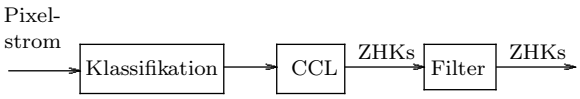


Abbildung 1: Pipelining bei der Erkennung von Zusammenhangskomponenten

vielseitig einsetzen zu können, sollen beliebig komplexe Bilder korrekt verarbeiten werden und unabhängig von Bildinhalten sollen pro Pixel maximal zwei Takte benötigt werden. Um mit relativ wenig Speicher auszukommen, darf nicht das gesamte Bild gespeichert werden, sondern lediglich ein Buffer für die Daten zweier Bildzeilen verwendet werden.

Wie in Abbildung 1 dargestellt, soll als Eingabe ein sequentieller Pixelstrom verwendet werden und als Ausgabe soll ein sequentieller Strom von erkannten Zusammenhangskomponenten geliefert werden.

Im ersten Schritt der zeilenweisen Abarbeitung des Bildes werden die RGB-Daten jedes Pixels entsprechend der gewünschten Anwendung klassifiziert. Beispielsweise wird man zur Erkennung von Verkehrszeichen nur zwischen roten, blauen, gelben, weißen und schwarzen, sowie sonstigen Pixeln unterscheiden. Für diese 6 Möglichkeiten kann man die RGB-Daten eines Pixels jeweils durch eine 3-Bit lange Zahl codieren.

Der CCL-Algorithmus verarbeitet dann eine abstrahierte Darstellung der Pixel, bei der zwei benachbarte Pixel zur gleichen ZHK gehören, wenn sie durch den gleichen Code dargestellt werden. Sobald der Algorithmus erkennt, dass eine ZHK in der aktuellen Bildzeile nicht mehr fortgesetzt wird, wird ihre Beschreibung ausgegeben.

Schließlich wird im Ausgabestrom eine Filterung vornehmen, so dass je nach Anwendung nur ZHKs mit bestimmten Eigenschaften ausgegeben werden. Beispielsweise sollte ein Verkehrszeichen eine bestimmte Mindestgröße haben, um bei der weiteren Auswertung Details erkennen zu können.

III. CCL-Algorithmus

Die zeilenweise Abarbeitung eines Bildes soll am Beispiel aus Abbildung 2 vorgestellt werden. In diesem Bild müssen drei ZHKs erkannt werden.

Beim Durchlaufen einer Zeile von links nach rechts werden die Bereiche gleichfarbiger Pixel als lokale ZHKs identifiziert. Ihr jeweils erstes Pixel in der Zeile wird als Repräsentant verwendet. Für die erste Bildzeile sind dies beispielsweise die Positionen $(1, 1)$ und $(7, 1)$ (siehe Abbildung 3).

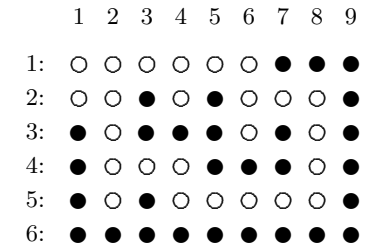


Abbildung 2: Beispielbild mit 9×6 Pixeln

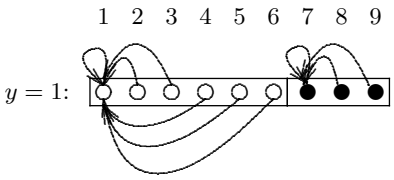


Abbildung 3: Die erste Zeile des Bildes besitzt 2 lokale ZHKs

Um die Zusammengehörigkeit von Pixeln zu dokumentieren, verweist jedes Pixel `p` auf den Repräsentanten `rep(p)` seiner lokalen ZHK. Repräsentanten erkennt man dann daran, dass sie auf sich selber zeigen.

Beim Durchlaufen der weiteren Zeilen müssen jeweils auch benachbarte Pixel der darüberliegenden Zeile betrachtet werden. Immer wenn ein Pixel `p` der aktuellen Zeile einen (vertikalen oder diagonalen) Nachbarn `q` gleicher Farbe in der darüberliegenden Zeile hat und `rep(p)` $\neq$ `rep(q)` gilt, müssen die betreffenden ZHKs miteinander vereinigt werden. Dazu verwendet man die folgenden beiden Regeln:

1. Liegt `rep(q)` in der oberen Zeile, lässt man `rep(q)` auf `rep(p)` verweisen. Einen solchen Verweis aus der oberen Zeile in die untere Zeile bezeichnen wir als einen „downlink“.

2. Liegt `rep(q)` in der unteren Zeile, d.h. es existiert bereits ein „downlink“, dann lässt man `rep(p)` auf `rep(q)` verweisen. In diesem Fall entsteht ein horizontaler Verweis von rechts nach links in der aktuellen Zeile.

In Abbildung 4 sind die relevanten Nachbarschaften zwischen der ersten und zweiten Zeile durch gepunktete Linien und die entstehenden Verweise als Pfeile eingetragen. Zur besseren Übersicht sind die Verweise innerhalb von lokalen ZHKs nicht mehr dargestellt. Man erkennt, dass die zweite Zeile derzeit die vier Repräsentanten (1,2), (3,2), (5,2) und (9,2) enthält. Dabei sind sämtliche weißen Pixel untereinander verbunden und es gibt drei schwarze ZHKs.

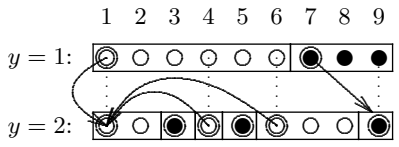


Abbildung 4: Die vertikalen Nachbarschaften zwischen Zeile 1 und 2 führen zu 4 ZHKs in Zeile 2

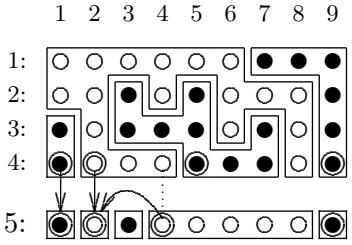


Abbildung 5: Zeile 4 des Beispiels mit Darstellung aller ZHKs des oberen Bildbereichs

Ist eine Zeile komplett abgearbeitet, braucht die darüberliegende Zeile nicht länger gespeichert zu werden. Sofern sie einen Repräsentanten ohne downlink-Verweis besitzt, ist seine ZHK vollständig und kann ausgegeben werden.

Abbildung 5 zeigt am Beispiel der Zeile 4 dass die Verkettungen innerhalb der Zeile tatsächlich die kompletten ZHKs des oberen Bildteils repräsentieren. Beim Übergang zu Zeile 5 entsteht dann für den Repräsentanten (5,4) kein downlink, so dass die erste abgeschlossene ZHK gefunden ist. Entsprechend wird später in Zeile 5 der Repräsentant (2,5) sämtliche weiße Pixel des Bildes repräsentieren und zum Schluss wird (1,6) in Zeile 6 alle verbleibenden schwarzen Pixel zusammenfassen.

Man beachte, dass das beschriebene Verfahren weiße und scharze Pixel auf die gleiche Weise behandelt, es also keine Sonderbehandlung des „Bildhintergrunds“ gibt. Deshalb kann es problemlos für Bilder mit mehr als zwei Farben genutzt werden, ohne dass sich der Zeit- oder Speicherbedarf erhöht. Im Extremfall könnte ein Bild mit $n \times m$ Pixeln insgesamt $n \cdot m$ ZHKs besitzen, wenn je zwei benachbarte Pixel unterschiedliche Farben haben. Innerhalb einer Zeile der Länge n gibt es dabei jeweils maximal n ZHKs.

Würde man stattdessen, wie in [3] vorgeschlagen, nur Schwarz-Weiß-Bilder betrachten und den weißen Hintergrund ignorieren. läge eine einfachere Aufgabenstellung mit maximal $\lceil n/2 \rceil$ ZHKs pro Zeile und maximal $\lceil n/2 \rceil \cdot \lceil m/2 \rceil$ ZHKs im Gesamtbild vor.

Tabelle 1: C++ Prozedur zum Verschmelzen der Merkmale zweier ZHKs `c1` und `c2`

```
void join(comp& c1, comp c2)
{// fügt die Merkmale von c2 zu c1 dazu
 if (c1.minx > c2.minx) c1.minx = c2.minx;
 if (c1.maxx < c2.maxx) c1.maxx = c2.maxx;
 if (c1.miny > c2.miny) c1.miny = c2.miny;
 if (c1.maxy < c2.maxy) c1.maxy = c2.maxy;
 c1.size += c2.size;
 c1.sumx += c2.sumx;
 c1.sumy += c2.sumy;
}
```

IV. Merkmale von ZHKs

Mit jedem vom CCL-Algorithmus erzeugten Verweis zwischen ZHKs müssen auch die geometrischen Daten der ZHKs im neuen Repräsentanten zusammengefasst werden.

Dazu betrachten wir nochmal den Übergang von Zeile 1 zu Zeile 2 in Abbildung 4. Hier bedeuten die drei Verweise auf den Repräsentanten (1,2), dass der lokalen ZHK (1,2) mit 2 Pixeln drei ZHKs der Größen 6, 1 und 3 Pixeln zugefügt werden. Damit stellt der Repräsentant schließlich eine ZHK mit 2+6+1+3=12 Pixel dar.

In der vorliegenden Implementierung werden neben der Anzahl `size` der Pixel auch die minimalen und maximalen x- und y-Koordinaten erfasst. Sie beschreiben das umschließende Rechteck (boundingbox) einer ZHK. Schließlich werden auch die Koordinatensummen `sumx` und `sumy` ermitteln, um mit (`sumx/size`,`sumy/size`) den Schwerpunkt der ZHK darzustellen.

Tabelle 1 zeigt die für diese Merkmale notwendigen Berechnungen durch eine Prozedur `join`. Prinzipiell kann man dem Algorithmus beliebig viele weitere Merkmale hinzufügen. Sofern die Operation `join` weiterhin in Hardware innerhalb eines Taktes ausgeführt werden kann, erhöht sich dabei lediglich der Speicherplatzbedarf. Merkmalsdaten, die in einer Applikation nicht verwendet werden, werden bei der Schaltungssynthese typischerweise automatisch wegoptimiert, so dass die Schaltung entsprechend kleiner wird.

V. Zeilenspeicher

Wie bereits vorgestellt, soll der CCL-Algorithmus nur die Daten über zwei Bildzeilen in RAMs speichern und dabei mit möglichst wenigen Speicherzugriffen auskommen.

Dazu werden zwei separate RAMs `cur_row` und

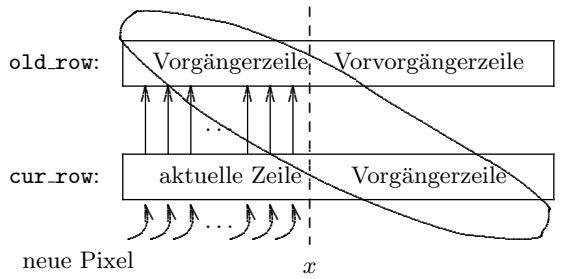


Abbildung 6: Speicherung von Bildzeilen in den RAMs `cur_row` und `old_row`

`old_row` verwendet. Bei der pixelweisen Eingabe einer neuen Bildzeile werden die neuen Pixel in `cur_row` geschrieben und überschreiben dort die Daten der Vorgängerzeile. Diese überschriebenen Daten werden gleichzeitig nach `old_row` kopiert.

Nach dem Einlesen der aktuellen Pixelposition x steht dann der linke Teil der Vorgängerzeile bereits in `old_row` und der rechte Teil noch in `cur_row` (siehe Abbildung 6). Der rechte Teil von `old_row` enthält zu diesem Zeitpunkt noch Daten der Vorvorgängerzeile. Man kann also zeitweise auf Daten dreier Zeilen zugreifen.

Im Abschnitt III. hat sich gezeigt, dass innerhalb einer Zeile Kettenstrukturen entstehen können. Beim Zugriff auf den Repräsentanten `rep(p)` eines Pixels `p` muss man also einer Kette von `p` zum Repräsentanten folgen. Um das Durchlaufen langer Ketten zu vermeiden, komprimiert der CCL-Algorithmus parallel zum Einlesen der aktuellen Zeile die Ketten der Vorgängerzeile.

In Abbildung 7 ist in Teil a) eine solche Kette dargestellt, die auf die beiden RAMs `cur_row` und `old_row` verteilt ist, Liest man in `cur_row` das Pixel der aktuellen Position `x` erhält man den Verweis auf ein Pixel in `old_row`. Liest man dessen Verweis, erhält man die Position des Repräsentanten von `x` und kann schließlich das Pixel `x` mit dem direkten Verweis auf seinen Repräsentanten in `old_row` eintragen. Das Ergebnis ist in Abbildung 7 in Teil b) zu sehen.

Eine entsprechende C++ Befehlssequenz ist in Tabelle 2 angegeben. Führt man sie für von links nach rechts für alle Pixel der Vorgängerzeile aus, ergibt sich in `old_row` die im Teil c) der Abbildung 7 gezeigte komprimierte Verkettungsstruktur, bei der jedes Pixel direkt auf seinen Repräsentanten verweist.

Es ist also möglich, parallel zum Einlesen der aktuellen Zeile die Ketten der Vorgängerzeile so zu komprimieren, dass die Repräsentanten der Vorgängerzeile direkt zugreifbar werden.

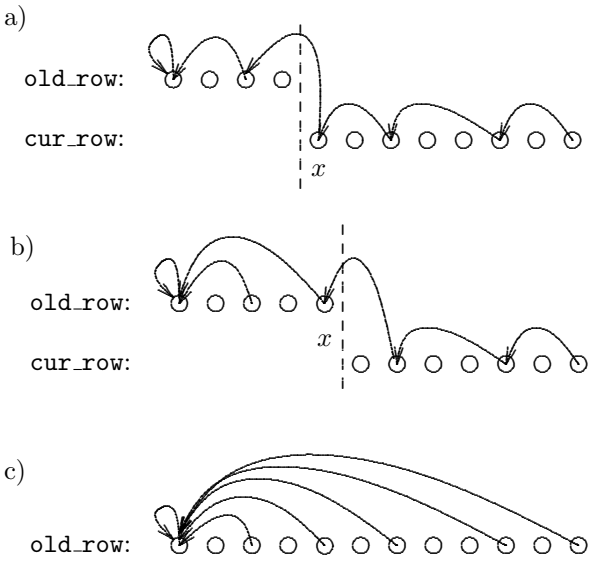


Abbildung 7: Vorgehensweise beim Komprimieren von Ketten. a) zeigt die Situation vor dem Umkopieren des Knotens `x`, b) zeigt das Ergebnis der ersten Komprimierung und c) zeigt das Ergebnis nach dem Umkopieren aller Knoten von `cur_row` nach `old_row`

Tabelle 2: C++ Programm zur Komprimierung der Kette an `x` beim Umkopieren von `cur_row` nach `old_row`

```
// Lese aktuellen Knoten
node= read(cur\_row, x);
// Lese Verweis auf Repräsentanten
node.link= read\_rep(old\_row,node.link);
// Schreibe komprimierten Knoten
write(old\_row,x,node);
// Lese Repräsentanten
rep= read(old\_row,node.link);
```

VI. Realisierung von Software-Lösungen in Hardware

Eine einfache C++ Implementierung des CCL-Algorithmus benötigte etwa 120 Programmzeilen. Für eine hardwarenahe C++ Implementierung, die nur die Daten zweier Bildzeilen speichert, waren etwa 500 Programmzeilen erforderlich und die Hardware-Beschreibung in VHDL umfasst schließlich etwa 1200 Zeilen. Die C++ Implementierungen wurden mit verschiedenen Testbilder erprobt und dabei wurden Protokolle über die gefundenen Zusammenhangskomponenten ausgegeben. Entsprechend wurde für die VHDL-Implementierung eine Testbench geschrieben, die bei der Schaltungssimulation eine gleichartige Protokolldatei erzeugt (Abbildung 8). Auf diese Weise ist es einfach möglich, die Ergebnisse verschiedener Software-

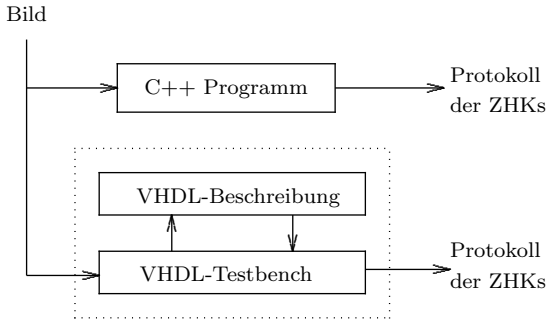


Abbildung 8: Vergleich von Software- und Hardware-Implementierungen durch Vergleich der Ergebnisprotokolle

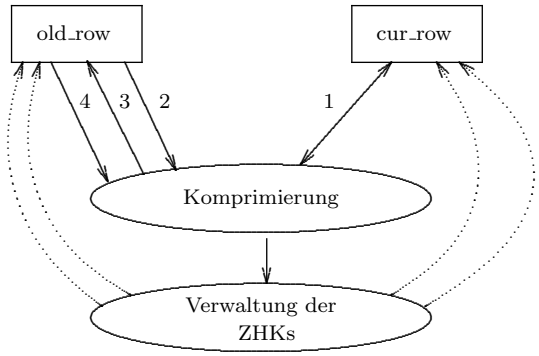


Abbildung 9: Notwendige Speicherzugriffe für die Komprimierung von Ketten und die Verwaltung von ZHKs

und Hardware-Implementierungen zu vergleichen und gegebenenfalls Implementierungsfehler zu lokalisieren.

A. Minimierung der Arbeitstakte

Im folgenden soll am Beispiel der Komprimierung von Ketten gezeigt werden, wie sich eine hardwarenahe C++ Beschreibung in eine VHDL Beschreibung umsetzen lässt. Das angegebene C++ Programm in Tabelle 2 benutzt dazu 4 Speicherzugriffe, die in Abbildung 9 durch Pfeile veranschaulicht werden. Zusätzlich werden allerdings auch noch weitere Schreibzugriffe aus anderen Teilen des CCL-Algorithm benötigt, so dass sich 5 Zugriffe auf `old_row` ergeben.

Da ein RAM pro Takt nur einen Zugriff erlaubt, bräuchte die Hardware-Implementierung mindestens 5 Takte pro Pixel. Verwendet man Dual-Port RAMs, sind 2 Zugriffe pro Takt möglich. Dies reicht aber immer noch nicht, um die in Abschnitt II. geforderten 2 Takten pro Pixel zu erreichen.

Deshalb wird wie in Abbildung 10 dargestellt, ein drittes RAMs `tmp_row`, genutzt, das eine Kopie der in `old_row` gespeicherten Verweise enthält. Das Schreiben der Verweise (Schritt 3) geschieht

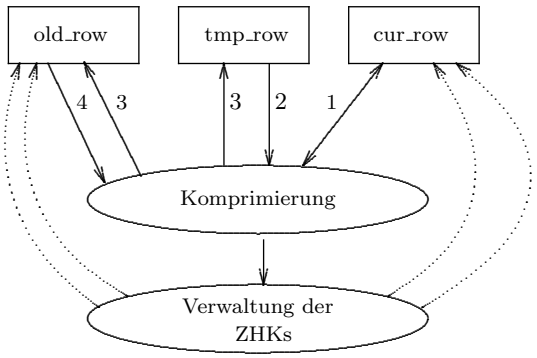


Abbildung 10: Einsatz eines dritten RAMs zur Reduzierung der Rechenzeit

Tabelle 3: Komprimierung einer Kette unter Verwendung eines dritten RAMs `tmp_row`

```
// Lese aktuellen Knoten
node= read(cur_row, x)
// Lese Verweis auf Repräsentanten
node.link= read_rep(tmp_row,node.link);
// Schreibe komprimierten Knoten
write(old_row,x,node);
// Schreibe Kopie
write(tmp_row,x,node);
// Lese Repräsentanten
rep= read(old_row,node.link);
```

parallel in beiden RAMs, so dass die Lesezugriffe auf beide RAMs verteilt werden können.

Tabelle 3 zeigt das entsprechend modifizierte C++ Programm für die Komprimierung von Ketten. Dabei ist durch horizontale Linien dargestellt, wie die Speicherzugriffe bei der Verwendung von Dual-Port RAMs auf die Taktzyklen aufgeteilt werden können.

B. Ablaufsteuerung durch Automaten

In der VHDL-Beschreibung der Hardware, wird diese Befehlssequenz durch einen Automaten abgearbeitet. Tabelle 4 stellt einen Auszug aus der Übergangsfunktion des Automaten dar. Der aktuelle Automatenzustand (current state) wird durch die Komponenten eines Records `cs.step` dargestellt. Entsprechend steht `ns.step` für den Folgezustand (next state) des Automaten. Zur besseren Verständlichkeit sind die ursprünglichen C++ Anweisungen fast identisch in VHDL formuliert. In der Praxis wird man die Prozedur- und Funktionsaufrufe allerdings durch elementare Anweisungen ersetzen.

Tabelle 4: VHDL-Code zur Komprimierung einer Kette

```
ns.step:= (others=> false);
if cs.step.read_cur_node
  then node:= read(cur_row, x);
       ns.step.read_link:=true;
end if;
if cs.step.read_link
  then node.link:=read_rep(tmp_row,node.link);
       ns.step.write_compressed:=true;
end if;
if cs.step.write_compressed
  then write(old_row,x,node);
       write(tmp_row,x,node);
       rep:= read(old_row,node.link);
end if;
```

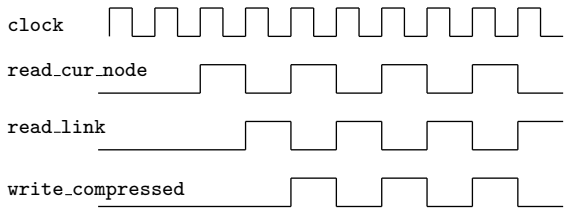


Abbildung 11: Verhalten von `cs.step` beim Komprimieren von Ketten

Die verwendete Vorgehensweise, jedem Schritt des Algorithmus eine Boolsche Größe im Automatenzustand zuzuordnen, hat große Ähnlichkeit mit einer „One-Hot“-Codierung. In der vorliegenden Implementierung können jedoch auch mehrere Komponenten von `cs.step` gleichzeitig gesetzt sein, so dass der Automat mehrere Schritte des Algorithmus gleichzeitig ausführt.

Beispielsweise kann ein jedem zweiten Takt eine neue Komprimierung gestartet werden, obwohl die vorherige noch nicht abgeschlossen ist. Als Konsequenz wird der erste Schritt (`read_cur_node`) gleichzeitig zum vorherigen dritten Schritt (`write_compressed`) ausgeführt. In Abbildung 11 ist das typische Verhalten von `cs.step` dargestellt.

Die vorgestellte einfache Umsetzung eines (hardwarenahen) Algorithmus in eine Automatenbeschreibung liefert eine synthetisierbare VHDL-Beschreibung. Allerdings kann sich ein schlechtes Zeitverhalten der Schaltung ergeben, weil die Synthesetools nicht sicher erkennen können, welche Schritte des Algorithmus tatsächlich im gleichen Takt ausgeführt werden können. Dadurch entstehen irrelevante unnötig lange kritische Pfade.

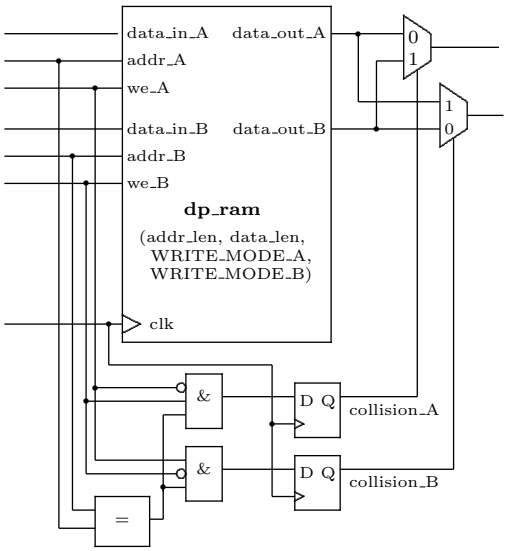


Abbildung 12: Schaltung zur Kollisionsvermeidung Bei Dual-Port RAMs

Als Abhilfe kann man eine für die Synthese günstige Reihenfolge der `if`-Anweisungen wählen und für Anweisungen in verschiedenen Takten immer unterschiedliche Variablen verwenden.

C. Kollisionsproblem bei Dual-Port RAMs

Wie bereits gezeigt, sind die Dual-Port RAMs für das Zeitverhalten der CCL-Schaltung von besonderer Bedeutung. Sie werden an beiden Ports mit dem gleichen Takt betrieben und es ist sichergestellt, dass nicht gleichzeitig über beide Ports auf die gleiche Adresse geschrieben wird. Auf diese Weise kann es zu keinen Konflikten zwischen Schreibzugriffen kommen.

Allerdings sind Kollisionen möglich, wenn man über ein Port auf eine Adresse schreibt und über das andere Port von der gleichen Adresse liest. In diesem Fall ist das Leseergebnis undefiniert. D.h. es ist unklar, ob man beim Lesen noch den alten oder schon den neuen Speicherinhalt erhält. Zwar kann man beim Schreiben zwischen einem `READ_FIRST`- und einem `WRITE_FIRST`-Mode wählen, aber dieser Modus betrifft jeweils nur einzelne Ports und nicht das Zusammenspiel beider Ports.

Als Abhilfe wurde deshalb die in Abbildung 12 dargestellte Zusatzschaltung genutzt, Sie bewirkt, dass im Fall der geschilderten Kollision der Lesezugriff ausnahmweise über das schreibende Port unter Verwendung des dort eingestellten Modus geschieht.

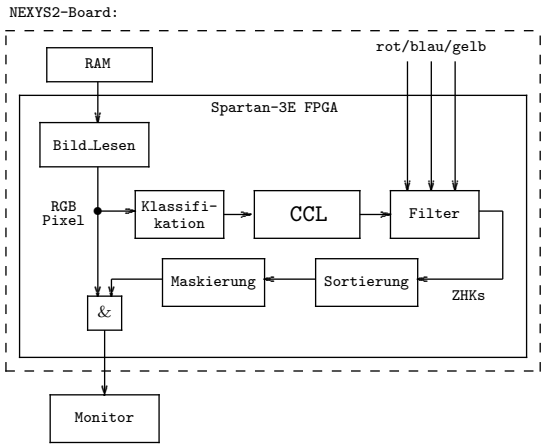


Abbildung 13: Demonstrationsumgebung zur Ausgabe von CCL-Ergebnissen auf einem Monitor

Abbildung 14: Tabelle mit einigen Verkehrszeichen

D. Demonstrationsumgebung

Zur Demonstration des neuen CCL-Verfahrens wurde es auf einem NEXSYS2-Board mit XILINX Spartan-3E FPGA getestet. Dabei werden RGB-Bilder der Größe 800 × 600 aus einem externen RAM gelesen und mit einer Bildwiederholrate von 50 Hz auf einem VGA-Monitor ausgegeben. Gleichzeitig wird der Ausgabestrom von der CCL-Schaltung ausgewertet und die Bounding-Boxes der gefundenen Zusammenhangskomponenten werden genutzt, um die irrelevanten Bildregionen im Folgebild auszublenden.

Abbildung 13 zeigt eine Skizze dieser Schaltung. `Bild_Lesen` ist für den Zugriff auf das RAM zuständig erzeugt den sequentiellen Pixelstrom für den Monitor. Der Block `Sortierung` sortiert die gefundenen Regionen der ZHKs, so dass sie in der Reihenfolge vorliegen, in der sie im Pixelstrom erstmals auftreten. Danach wird in `Maskierung` an-

Abbildung 15: Foto der Bildausgabe am Monitor, wenn der CCL-Algorithmus nur rote, blaue und gelbe Verkehrszeichen selektiert

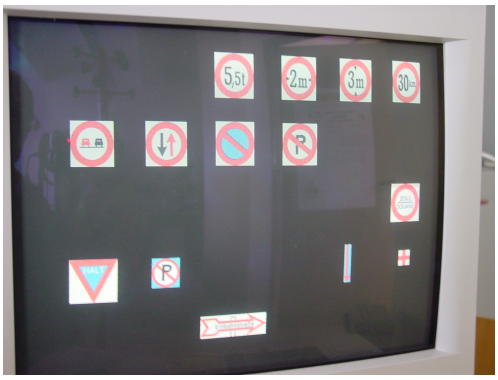


Abbildung 16: Foto der Bildausgabe am Monitor, wenn der CCL-Algorithmus nur rote, Verkehrszeichen selektiert

hand der sortierten Liste entschieden, ob die aktuelle Pixelposition zu einer relevanten Region gehört oder nicht. Schließlich sorgt das erzeugte Maskierungssignal dafür, dass irrelevante Regionen am Bildschirm ausgeblendet werden. Über drei Tasten kann man die Farben zur Auswahl relevanter ZHKs vorgeben.

Als Anwendungsbeispiel zeigt Abbildung 14 eine Tafel mit einigen Verkehrszeichen und die Abbildungen 15 und 16 zeigen Fotos der entsprechenden Ausgabe auf dem Monitor, wenn nur nach Verkehrszeichen einer bestimmten Farbe gesucht wird. Man beachte, dass dabei erklärende Texte und schwarz-weiße Verkehrszeichen ausgeblendet werden.

VII. Fazit

Es wurde ein 1-Pass CCL-Algorithmus in VHDL implementiert, der einen sequentiellen Pixelstrom in Realzeit in einen Ausgabestrom erkannter Zusammenhangskomponenten überführt. Das Verfah-

ren funktioniert für Farbbilder und benötigt pro Pixel zwei Taktzyklen. Die Ergebnisse entstehen dabei unmittelbar nach Abschluss der jeweiligen Zusammenhangskomponenten und nicht erst nach dem Einlesen des kompletten Bildes.

Mit einer Beispielsanwendung auf einem Spartan-3E FPGA wurde die korrekte Arbeitsweise demonstriert. Dabei wurde eine Taktrate von 64.28 MHz verwendet.

VIII. Ausblick

Derzeit wird die Schaltung auf ein FPGA-Board mit einem schnellerem Spartan-6 FPGA und HDMI Ein- und Ausgängen portiert. Erste Syntheseversuche haben gezeigt, dass sich die Taktrate dabei mehr als verdoppeln lässt. Damit wird die CCL-Implementierung Videosequenzen mit höherer Auflösung und größerer Bildwiederholrate in Realzeit analysieren können.

Später sollen die erzeugten Ausgabeströme von erkannten Objekten per Software auf einem Prozessor im FPGA weiter untersucht werden. Dabei ist auch geplant, die Objektpositionen in aufeinander folgenden Bildern zu vergleichen, um Bewegungsabläufe zu analysieren.

Außerdem wird noch nach zeitkritischen Anwendungsproblemen gesucht, bei denen auch das frühzeitige Erkennen von Zusammenhangskomponenten von Bedeutung ist.

Literaturverzeichnis

[1] D.G. Bailey, C.T. Jonston, „Single Pass Connected Components Analysis“, in *Image and Vision Computing New Zealand*, pp. 282-287, Hamilton, New Zealand, 2007

[2] C.T.Jonston, D.G. Bailey, „FPGA Implementation of a Single Pass Connected Components Algorithm“, *4th IEEE International Symposium on Electronic Design, Test & Applications*, 2008

[3] Ni Ma, D.G. Bailey, C.T. Jonston, „Optimized Single Pass Connected Components Analysis“, *International Conference on Field-Programmable Technology*, Taipeh, 2008

[4] S. Jaeckel, A. Sikora, W. Rülling, „Implementierung eines Single Pass Connected Component Labelling Algorithmus zur Detektion von leuchtenden Objekten in Nachtszenen im Automotive Umfeld“, *43. MPC-Workshop*, Göppingen, Februar 2010

[5] K. Appiah, A. Hunter, P. Dickinson, J. Owens, „A Run-Length Based Connected Component Algorithm for FPGA Implementation“ *International Conference on Field-Programmable Technology*, Taipeh, 2008

Wolfgang Rülling ist seit 1989 Professor für Informatik und Mikroelektronik an der Hochschule Furtwangen. Zu seinen Lehr- und Forschungsgebieten zählen „Hardware-Algorithmen“, „Programmierbare Bausteine“ und „Algorithmen und Datenstrukturen“. Zuvor war er unter anderem leitender Mitarbeiter an der Universität Saarbrücken im Sonderforschungsbereich „VLSI-Entwurfsmethoden und Parallelität“ und Mitarbeiter am Institut für Theoretische Informatik der Technischen Hochschule Darmstadt.

hochschule aschaffenburg
university of applied sciences

Modulare Hardware-Software Bildverarbeitungsplattform am Beispiel einer Vordergrund-Hintergrundtrennung

Johannes Kempf, Konrad Doll

***Zusammenfassung*—Field Programmable Gate Arrays (FPGAs) werden häufig eingesetzt, um Bildverarbeitungsalgorithmen schnell und effizient realisieren zu können. Ein beachtlicher Anteil der Gesamtentwicklungszeit wird dabei häufig aufgewendet, um den FPGA in ein Gesamtsystem zu integrieren. Vorgestellt wird eine modulare Implementierungsplattform, die an unterschiedliche Aufgabenstellungen flexibel angepasst werden kann. Das System umfasst eine PCI-Express Schnittstelle, zwei PowerPC Prozessorkerne mit DDR2 SDRAM sowie anwendungsspezifisch anpassbare Bildverarbeitungsmodule. Je nach Einsatzzweck können die Module unterschiedlich kombiniert und die Berechnungen auf Hard- und Software verteilt werden. Die Leistungsfähigkeit des Gesamtsystems wird anhand einer echtzeitfähigen Vordergrund-Hintergrundtrennung auf Basis des Mixture of Gaussians (MoG) Algorithmus präsentiert.**

***Schlüsselwörter*—FPGA, Mixture of Gaussians, Hintergrund-Vordergrundsegmentierung.**

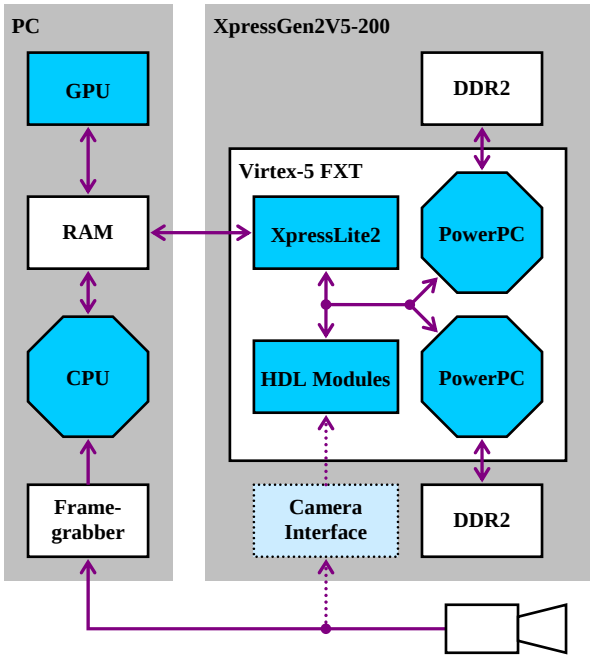


Abbildung 1: Blockschaltbild der Bildverarbeitungsplattform

I. Einleitung

Im Folgenden wird eine aus CPU, GPU und FPGA bestehende Bildverarbeitungsplattform präsentiert, mit der Bildverarbeitungsalgorithmen flexibel auf Hard- und Software aufgeteilt werden können.

A. Stand der Technik

Bildverarbeitungsalgorithmen sind im Allgemeinen rechenzeitintensiv und benötigen viele Speicherressourcen. Deswegen werden in vielen Anwendungsfällen leistungsfähige Prozessoren eingesetzt. Bei zeitkritischen Anwendungen kommen zusätzlich Grafikprozessoren, sog. Graphics Processing Units (GPUs) [1] oder FPGAs [2] zum Einsatz, wobei viele Bildverarbeitungssysteme jeweils entweder nur GPUs oder FPGAs nutzen. Bauer u.a. [3] stellen eine Architektur zur Fußgängererkennung vor, welche aus CPU, GPU und FPGA besteht. Ein Teil der Funktionalität wird dort mit einem FPGA realisiert, das mit der Entwicklungsumgebung Visual Applets [4] von Silicon Software programmierbar ist. Die nachfolgend beschriebene Bildverarbeitungsplattform besteht aus CPU, GPU und FPGA, wobei das FPGA mit Hardwarebeschreibungssprachen frei programmierbar und somit wesentlich flexibler eingesetzt werden kann. Als Beispielanwendung ist eine Vordergrund-Hintergrundtrennung realisiert. Infrastrukturbasierte Kreuzungsassistenzsysteme zur Erhöhung der Verkehrssicherheit erkennen und klassifizieren bewegte Objekte häufig auf Basis von Videodaten. Hierbei werden u.a. Methoden zum Trennen von Vorder- und Hintergrund angewendet, um mit Hilfe der Vordergrundinformation Verkehrsteilnehmer zu detektieren [3]. Gängige Verfahren zum Segmentieren von Vorder- und Hintergrund sind Running Gaussian Average (RGA) [5], Mixture of Gaussians (MoG) [6] sowie Kernel Density Estimation (KDE) [7], die sich in Geschwindigkeit, Speicheranforderungen und Genauigkeit unterscheiden [8]. Der RGA Algorithmus lässt sich sehr leicht implementieren, kann jedoch nur unzureichend auf dynamische Veränderungen des Hintergrunds reagieren. MoG und KDE sind dagegen in der Lage, auch

J. Kempf, kempf.johannes@gmx.de, und K. Doll, konrad.doll@h-ab.de, sind Mitglieder der Hochschule Aschaffenburg, Fakultät Ingenieurwissenschaften, Labor für Rechnergestützten Schaltungsentwurf, Würzburger Straße 45, 63743 Aschaffenburg.

multi-modale Hintergrundbilder zu handhaben, wie sie beispielsweise bei bewegten Blättern an Bäumen oder bei Schattenwurf auftreten. Dafür benötigen die Algorithmen jedoch wesentlich mehr Speicherplatz, wobei der MoG Algorithmus mit Abstand den besten Kompromiss zwischen Speicherbedarf und Genauigkeit darstellt [8]. Beim MoG Verfahren wird der Hintergrund mit Hilfe mehrerer Gaußverteilungen modelliert, wobei diese von Bild zu Bild aktualisiert werden [9]. Jiang u.a. [10] modifizieren die Aktualisierungsgleichungen, sodass diese in einem FPGA umgesetzt werden können. Neben der Bildverarbeitungsplattform werden Vereinfachungen vorgestellt, die den Ressourcenverbrauch weiter senken und dabei die Genauigkeit kaum beeinflussen.

B. Übersicht

Zunächst wird in Abschnitt II der Hardwareaufbau vorgestellt, welcher in Abbildung 1 schematisch dargestellt ist. Anschließend geht Abschnitt III auf den MoG Algorithmus näher ein und beschreibt Modifikationen, welche zur FPGA Implementierung notwendig sind. Diese Implementierung wird schließlich in Abschnitt IV vorgestellt, bevor Abschnitt V die Leistungsfähigkeit der Beispielanwendung präsentiert. Zuletzt wird ein Ausblick auf zukünftige Arbeiten gegeben.

II. Architektur der Hardware

Die Bildverarbeitungsplattform besteht aus einer Kombination eines handelsüblichen PCs mit dem Entwicklungsboard XpressGen2V5-200 von PLDA [11]. Dieses ist mit zwei 256 MB DDR2 Speichermodulen und dem FPGA XC5VFX200T von Xilinx, Inc. bestückt. Mit den PowerPC 440 Processor Blocks [12] besitzt der Virtex-5 FXT FPGA zwei Prozessorkerne, die als Hardcore-Prozessoren realisiert sind. Daneben verfügt das Entwicklungsboard über eine PCI-Express Schnittstelle, mit der Direct Memory Access (DMA) Übertragungen zwischen dem Arbeitsspeicher des PCs und dem FPGA durchgeführt werden können. Hierzu wird der PCI-Express XpressLite2 IP Core von PLDA [13] genutzt.

A. PC

Der PC dient bei dieser Bildverarbeitungsplattform als Bilddatenquelle und -anzeige, wobei die Bilddaten wahlweise von einem Framegrabber oder einer Videodatei stammen können. Er ist mit einem Intel Core2 Prozessor mit einer Taktfrequenz von 2,4 GHz und 3 GB Arbeitsspeicher ausgestattet. Zudem kann auf dem PC eine Vor- oder Nachverarbeitung der Bilddaten z.B. mit OpenCV durchgeführt werden. Mit Hilfe der GPU einer Grafikkarte ist es zudem möglich, Berechnungen durch parallelisierte Algorithmen zu beschleunigen. In einer zukünftigen Ausbaustufe können die Bilddaten direkt, z.B. über eine Gigabit-Ethernet (GigE) oder eine CameraLink Schnittstelle, in das FPGA eingespeist werden.

B. PCI-Express Schnittstelle

Mit dem XpressLite2 IP Core kann das Entwicklungsboard über DMA Kanäle Daten bidirektional mit dem PC austauschen, ohne dass dabei Rechenleistung der CPU des PCs benötigt wird. FPGA-seitig stellt das PCI-Express Modul je DMA Kanal einen First-In-First-Out (FIFO) Speicher als Datenschnittstelle bereit. Dieser puffert die Transferdaten und ermöglicht eine Taktentkopplung von der PCI-Express Schnittstelle.

C. PowerPC 440 Prozessorkerne

Die PowerPC 440 Processor Blocks sind 32 Bit Prozessorkerne, die als integrierte Schaltkreise im FPGA verfügbar sind. Diese können über den Fabric Co-processor Bus (FCB) mit sog. Fabric Co-processor Modulen (FCM) erweitert werden, die mit einer Hardwarebeschreibungssprache im FPGA erzeugt wurden. Der PowerPC Instruktionssatz enthält mehrere Assemblerbefehle, welche speziell für den Einsatz von FCMs reserviert sind. Damit können u.a. Daten mit einer Breite von 128 Bit mit nur einem Assemblerbefehl zwischen PowerPC und FCM transferiert werden. Diese Befehle werden genutzt, um die PowerPC Prozessoren in die Bildverarbeitungsplattform zu integrieren. Hierbei setzt eine FCM die Programminstruktionen um, indem es die Daten anderen Hardwaremodulen über FIFO Speicher zur Verfügung stellt. Beide PowerPCs werden mit einem 400 MHz Takt betrieben und verfügen zudem über diverse Ein- und Ausgangsports sowie über jeweils eine EIA-232 Schnittstelle und 256 MB DDR2 SDRAM. Damit lassen sich auch komplexe C++ Programme im PowerPC realisieren.

D. Hardwaremodule

Je nach Anwendungsfall können die PowerPC Prozessoren und die PCI-Express Schnittstelle mit unterschiedlichen Hardwaremodulen verbunden werden. In diesen lassen sich diverse Bildverarbeitungsalgorithmen, wie z.B. Gradientenberechnung, Farbraumkonvertierungen, Glättungsfilter oder morphologische Operationen direkt in logischen Schaltungsstrukturen realisieren. Aufgrund der modularen Struktur der Hardwareplattform können unterschiedliche Datenflüsse realisiert werden. Abbildung 2 stellt hierzu ein Beispiel dar und wird in Abschnitt IV näher erläutert.

III. Mixture of Gaussians

Um in einem Video den Vorder- und Hintergrund trennen zu können, muss für jedes Pixel entschieden werden, ob es zum Vorder- oder Hintergrund gehört. Stauffer und Grimson [6] stellen ein Verfahren zur

Vordergrund-Hintergrundsegmentierung vor, das auf der Verwendung von K Gaußverteilungen zur Modellierung eines Hintergrundbildes beruht. Diese Gaußverteilungen werden durch ihren Mittelwert $\hat{\mu}_k$ und ihre Varianz $\hat{\sigma}_k^2$ dargestellt $\left(k \in \{1,2,\ldots,K\}\right)$. Ein zusätzliches Gewicht $\hat{\omega}_k$ beschreibt die Relevanz dieser Gaußverteilung in Bezug auf das Hintergrundmodel. Ein Pixel wird dem Hintergrund zugeordnet, wenn sein Grauwert x_t zum Zeitpunkt t innerhalb des Intervalls $\left[\hat{\mu}_k - \tau \cdot \hat{\sigma}_k^2 \;;\; \hat{\mu}_k + \tau \cdot \hat{\sigma}_k^2\right]$ einer der K Gaußverteilungen liegt. Der Parameter τ wird anwendungsspezifisch vom Benutzer gewählt. Die Theorie des MoG Algorithmus basiert auf Farbbildern, kann jedoch auch auf Graustufenbildern angewendet werden. Da in dieser Arbeit ausschließlich Graustufenbilder zum Einsatz kommen, sind alle Gleichungen für diesen Fall vereinfacht dargestellt. Die Wahrscheinlichkeit $P(x_t)$, dass in einem Pixel der Grauwert x_t auftritt, beschreiben die Gleichungen (1) und (2), wobei η die zugehörige Wahrscheinlichkeitsdichtefunktion ist [6].

$$P(x_t) = \sum_{i=1}^{K} \hat{\omega}_{i,t} \cdot \eta\left(x_t, \hat{\mu}_{i,t}, \hat{\sigma}_{i,t}\right) \tag{1}$$

$$\eta(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{1}{2}\sigma^2 (x-\mu)^2} \tag{2}$$

A. MoG Implementierung

Jiang u.a. [10] berechnen die für eine FPGA Implementierung modifizierten Mittelwerte μ_k und Varianzen σ_k^2 nicht mehr algebraisch. Stattdessen werden sie in Abhängigkeit von einem Distanzmaß $d = \mu_{k,t-1} - x_t$ Bild für Bild inkrementell erhöht oder erniedrigt. Liegt der Grauwert x_t eines Pixels innerhalb des Intervalls $\left[\mu_k - \tau \cdot \sigma_k^2 \;;\; \mu_k + \tau \cdot \sigma_k^2\right]$ einer der K Gaußverteilungen, so werden die zugehörigen Parameter gemäß den Gleichungen (3), (4) und (5) aktualisiert:

$$\mu_{k,t} = \begin{cases} \mu_{k,t-1} + 1 & \text{wenn} \quad d < 0 \\ \mu_{k,t-1} - 1 & \text{wenn} \quad d > 0 \\ \mu_{k,t-1} & \text{sonst} \end{cases} \tag{3}$$

$$\sigma_{k,t}^2 = \begin{cases} \sigma_{k,t-1}^2 + \frac{1}{4} & \text{wenn} \quad d^2 - \sigma_{k,t-1}^2 > 0 \\ \sigma_{k,t-1}^2 - \frac{1}{4} & \text{wenn} \quad d^2 - \sigma_{k,t-1}^2 < 0 \\ \sigma_{k,t-1}^2 & \text{sonst} \end{cases} \tag{4}$$

$$\overline{\omega}_{k,t} = (1-\alpha) \cdot \hat{\omega}_{k,t-1} + \alpha \tag{5}$$

Der Wert von α bestimmt die Lernrate. Stimmt der Grauwert nicht mit der Gaußverteilung überein, so wird lediglich das Gewicht gemäß Gleichung (6) aktualisiert:

$$\overline{\omega}_{k,t} = (1-\alpha) \cdot \hat{\omega}_{k,t-1} \tag{6}$$

Konnte für den Grauwert keine übereinstimmende Gaußverteilung gefunden werden, so wird die Gaußverteilung mit dem geringsten Gewicht durch eine neue Verteilung ersetzt. Der Mittelwert wird dabei mit dem Grauwert x_t, die Varianz mit einem Startwert σ_0^2 und das Gewicht mit α initialisiert. Abschließend werden die Gewichte nach Gleichung (7) normiert und in absteigender Reihenfolge sortiert.

$$\hat{\omega}_{k,t} = \frac{\overline{\omega}_{k,t-1}}{\sum_{i=1}^{K} \overline{\omega}_{i,t}} \tag{7}$$

B. Modifikationen des MoG Algorithmus

Auf Basis der Implementierung von Jiang u.a. [10] können weitere Vereinfachungen getroffen werden, um den Ressourcenverbrauch zu senken. In Gleichung (4) erscheint die Varianz σ_k^2 nur als Schwellwert in Vergleichen und wird zu keiner weiteren Berechnung benötigt. Sie kann deshalb durch die Standardabweichung σ_k ersetzt werden, wodurch eine ressourcenintensive Quadrierungsoperation eingespart und durch einen Multiplexer zur Betragsbildung ersetzt werden kann. Wegen der kleineren Zahlenwerte reduziert sich zudem die nötige Bitanzahl zur Darstellung von σ_k. Aufgrund dieser Vereinfachung wurde als Inkrement $\frac{1}{2}$ anstatt $\frac{1}{4}$ gewählt.

$$\sigma_{k,t} = \begin{cases} \sigma_{k,t-1} + \frac{1}{2} & \text{wenn} \quad |d| > \sigma_{k,t-1} \\ \sigma_{k,t-1} - \frac{1}{2} & \text{wenn} \quad |d| < \sigma_{k,t-1} \\ \sigma_{k,t-1} & \text{sonst} \end{cases} \tag{8}$$

Bedingt durch in der Praxis auftretende, kleine Lernraten $\alpha < 0.01$ kann der Faktor $(1-\alpha)$ aus den Aktualisierungsgleichungen entfernt werden, wodurch sich weitere Ressourcen des FPGAs einsparen lassen. Die Gleichungen (5), (6) und (7) zur Berechnung der Gewichte ω_k vereinfachen sich somit zu:

$$\widetilde{\omega}_{k,t} = \omega_{k,t-1} + \alpha \tag{9}$$

$$\widetilde{\omega}_{k,t} = \omega_{k,t-1} \tag{10}$$

$$\omega_{k,t} = \frac{\widetilde{\omega}_{k,t-1}}{\sum_{i=1}^{K} \widetilde{\omega}_{i,t}} \tag{11}$$

Die dadurch entstehende Abweichung bei der Aktualisierung von ω_k kann mit α abgeschätzt werden und

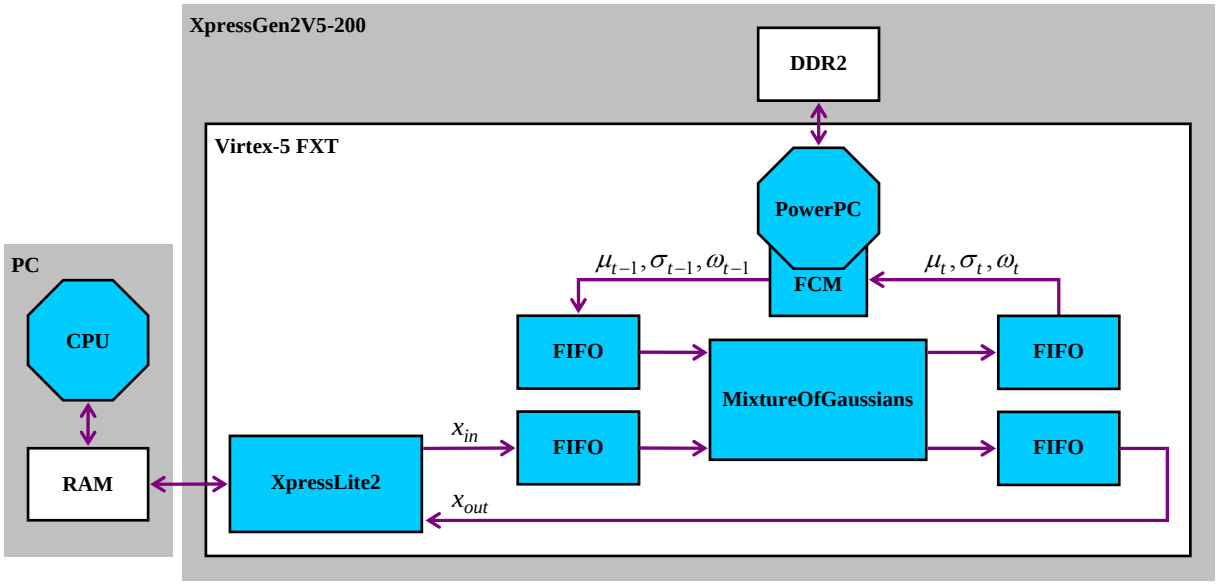


Abbildung 2: Systemarchitektur und Datenfluss der Beispielimplementierung

wird durch die Normierung (11) wieder größtenteils ausgeglichen. Die Abweichung wirkt sich deshalb nur unwesentlich auf das Ergebnisbild aus.

IV. Beispielimplementierung

Der im vorherigen Abschnitt erläuterte, modifizierte MoG Algorithmus wurde als Beispielimplementierung mit der vorgestellten Bildverarbeitungsplattform umgesetzt.

A. Systemarchitektur

Wie in Abbildung 2 illustriert, besteht das System aus mehreren Modulen, deren Ein- und Ausgänge über FIFO Speicher gepuffert sind. Der XpressLite2 IP Core liest dabei Bilddaten via PCI-Express aus dem Arbeitsspeicher des PCs und leitet sie an das Hardwaremodul *MixtureOfGaussians* weiter. Parallel dazu werden vom PowerPC die zugehörigen MoG Parameter aus dem DDR2 Speicher des FPGAs ausgelesen. Das Modul *MixtureOfGaussians* berechnet mit diesen Eingangsdaten ein Vordergrundbild, welches vom *XpressLite2* IP Core in den Arbeitsspeicher des PCs geschrieben wird. Die aktualisierten MoG Parameter werden parallel dazu vom PowerPC zurück in den DDR2 Speicher des Entwicklungsboards geschrieben. Mit Hilfe von morphologischen Operatoren der Bildverarbeitungsbibliothek OpenCV wird Rauschen des Vordergrundbildes auf dem PC reduziert und das Ergebnisbild am Monitor visualisiert. In Abbildung 4 sind das Eingangs- sowie das Ausgangsbild vor und nach der morphologischen Operation abgebildet.

B. Realisierung des MoG Algorithmus

Das Modul *MixtureOfGaussians*, dessen innere Struktur Abbildung 3 verdeutlicht, ist zur Berechnung des MoG Algorithmus in mehrere Untermodule aufgeteilt. Zunächst bestimmt das Modul *MatchAndFit*, ob eine bzw. welche der Gaußverteilungen aktualisiert werden muss. Ein nachgeschalteter Multiplexer extrahiert die zugehörigen Parameter und leitet sie an das Modul *ParameterUpdate* weiter. Dieses aktualisiert die Parameter gemäß den Gleichungen (3), (8) und (9) oder generiert eine neue Gaußverteilung. Das Modul *SortMux* sortiert die aktualisierten oder neu generierten Parameter anhand des Gewichtswertes. Parallel zur Parameteraktualisierung entscheidet das Modul *ForegroundDetection*, ob das Pixel zum Vorder- oder Hintergrund gehört und summiert alle Gewichtswerte auf. Diese Summe wird vom Modul *NormalizeWeights* genutzt, um die Gewichte abschließend zu normieren. Während dem Erstellen der Hardwaremodule mit VHDL wurden mehrere Pipelinestufen festgelegt, um den Datendurchsatz des Systems zu erhöhen.

C. Festlegung von Designparametern

Der MoG Algorithmus wurde als generisches Hardwaremodul realisiert, welches sich über Konstanten und Generics parametrieren lässt. Aus Gleichung (3) folgt, dass für die Darstellung der Mittelwerte μ_k keine Nachkommastellen benötigt werden. Die Datenbreite der Mittelwerte ist deshalb gleich der Auflösung eines Pixels, die bei dieser Anwendung 8 Bit beträgt. Gemäß Gleichung (8) wird für die Standardabweichung σ_k 1 Bit für die Nachkommastelle benötigt. Zivkovic [9] begrenzt die Varianz σ_k^2 auf das Intervall $\left[2\sigma_0^2 \;;\; 5\sigma_0^2\right]$, wodurch 5 Bit zur Darstellung von typischen Werten der Standardabweichung ausreichen. Die Datenbreite der Gewichte ω_k ist abhängig von

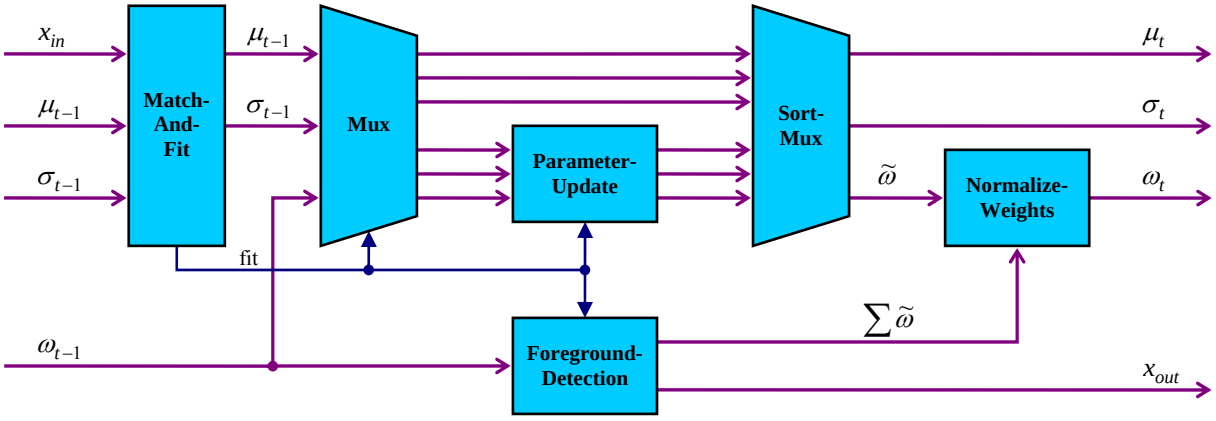


Abbildung 3: Innere Struktur des Moduls *MixtureOfGaussians*

der Lernrate α. Für diese Anwendung wurde eine Datenbreite von 16 Bit gewählt, wodurch α in einem großen Bereich variiert werden kann. Die Parameter einer Gaußverteilung benötigen somit insgesamt 29 Bit. Da der PowerPC mit einem FCM Assemblerbefehl 128 Bit transferieren kann, wurde die Anzahl der Gaußverteilungen $K = 4$ gesetzt.

D. Softwareseitiger Speicherzugriff

Mit Hilfe von auf dem PowerPC ausgeführter Software wird auf den DDR2 Speicher des Entwicklungsboards zugegriffen und dessen Daten dem Modul *MixtureOfGaussians* zur Verfügung gestellt. Dazu wurde der PowerPC um eine FCM erweitert, wie in Abbildung 2 dargestellt ist. Dieser Rapid-Prototyping-Ansatz wurde in der vorgestellten Implementierung ganz bewusst gewählt, um u.a. die Leistungsfähigkeit und Flexibilität des Bildverarbeitungssystems zu demonstrieren.

V. Ergebnisse

Es wurde eine Bildverarbeitungsplattform bestehend aus CPU, GPU und FPGA realisiert und als Beispielanwendung eine Vordergrund-Hintergrundsegmentierung auf Basis des MoG Algorithmus implementiert, womit Videodaten mit einer Auflösung von 800 × 600 Pixel bei 22 fps verarbeitet werden können. Das System benötigt einen PowerPC und ca. 23 % der im FPGA verfügbaren Ressourcen. Somit können die Hardwaremodule im FPGA dupliziert werden, wodurch Bildgröße oder Framerate verdoppelt werden kann. Der PowerPC wird in diesem System verwendet, um den DDR2 SDRAM als Ringspeicher nutzen zu können. Zudem ermöglicht die EIA-232 Schnittstelle des PowerPCs, Diagnosemeldungen auszugeben. In Tabelle 1 sind die Kenndaten der Implementierung von Jiang u.a. [10] und der präsentierten Bildverarbeitungsplattform gegenübergestellt. Die vorgestellte MoG Implementierung erreicht einen vergleichbaren Datendurchsatz, obwohl auf eine nicht deterministische Datenkomprimierung und einen direkten Speicherzugriff verzichtet wurde. Die nicht deterministische Datenkomprimierung wird von Jiang u.a. [10] benötigt, um die erforderliche Speicherbandbreite zu reduzieren.

Tabelle 1: Vergleich der MoG Implementierungen

	Jiang u.a. [10]	vorgestelltes System
Auflösung	640 × 480 Pixel	800 × 600 Pixel
Farbkanäle	3	1
Gaußverteilungen	3	4
Framerate	25 fps	22 fps
Plattform	FPGA	CPU+FPGA+ PowerPC

VI. Ausblick

Die maximale Framerate der MoG Implementierung wird derzeit durch die Geschwindigkeit des PowerPCs und der Bandbreite des DDR2 Speichers begrenzt. Deshalb soll die Hardwareplattform um eine Schnittstelle erweitert werden, mit der auf den DDR2 Speicher ohne PowerPC zugegriffen werden kann. Mit Hilfe einer Schattenbehandlung [9] kann zudem die Qualität der Vordergrund-Hintergrundtrennung erhöht werden. Weiterhin soll eine Kameraschnittstelle implementiert werden, mit der Bilddaten über GigE Vision oder CameraLink direkt in das FPGA eingespeist werden können.

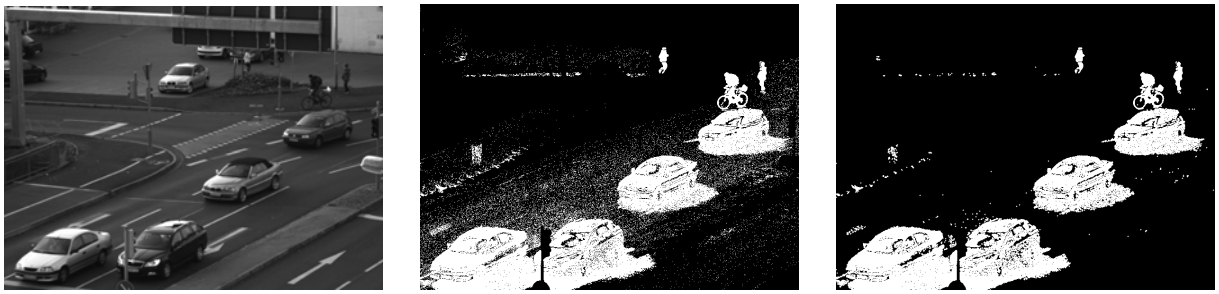
Danksagung

Die Autoren bedanken sich für die Unterstützung des Forschungsschwerpunktes „Intelligente Sensorik“ durch das Bayerische Staatsministerium für Wissenschaft, Forschung und Kunst.

Literaturverzeichnis

[1] NVIDIA: NVIDIA Adds GPU Acceleration for OpenCV Application Development. *NVIDIA press releases*, sep. 23, 2010.

[2] SONY: XCI-Series Inteligent Cameras. *Datenblatt*, 2009.



(a) (b) (c)

Abbildung 4: Vordergrundtrennung durch MoG Algorithmus. (a) Eingangsbild. (b) Ausgangsbild des Hardwaremoduls. (c) Ausgangsbild nach morphologischer Operation.

[3] S. Bauer, S. Köhler, K. Doll, U. Brunsmann: FPGA-GPU Architecture for Kernel SVM Pedestrian Detection. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2010, S. 61–68.

[4] Silicon Software: Visual Applets Operator Overview. *Datenblatt*, Version 1.3, 2008.

[5] C. Wren, A. Azarbayejani, T. Darrell, A. Pentland: Pfinder: Real-Time Tracking of the Human Body. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Band 19, No. 7, 1997, S. 780–785.

[6] C. Stauffer, W.E.L Grimson: Adaptive background mixture models for real-time tracking. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Band 2, 1999, S. 246–252.

[7] A. Elgammal, D. Harwood, L. Davis: Non-parametric Model for Background Subtraction: *6th European Conference on Computer Vision*, 2000, S. 751–767.

[8] M. Piccardi: Background subtraction techniques: a review. *IEEE International Conference on Systems, Man and Cybernetics*, Band 4, 2004, S. 3099–3104.

[9] Z. Zivkovic: Improved Adaptive Gaussian Mixture Model for Background Subtraction. *17th International Conference on Pattern Recognition*, Band 2, 2004, S 28–31.

[10] H. Jiang, H. Ardö, V. Öwall: A Hardware Architecture for Real-Time Video Segmentation Utilizing Memory Reduction Techniques. *IEEE Transactions on Circuits and Systems for Video technology*, Band 19, 2009, S. 226–236.

[11] PLDA: XpressGen2V5-200. *Reference Manual*, Version 1.0, 2008.

[12] Xilinx, Inc.: Embedded Processor Block in Virtex-5 FPGAs. *Reference Guide*, Version 1.8, 2010.

[13] PLDA: PCI Express XpressLite2. *Reference Manual*, Version 2.0.5, 2010.

Johannes Kempf erhielt den akademischen Grad des B.Eng. in Elektro- und Informationstechnik im Jahr 2010 von der Hochschule Aschaffenburg. Den Grad des M.Eng. in Elektro- und Informationstechnik wird er voraussichtlich im Jahr 2011 von der Hochschule Aschaffenburg erhalten.

Konrad Doll erhielt den akademischen Grad des Dipl.-Ing. im Jahr 1989 und den Grad des Dr.-Ing. im Jahr 1993 in Elektro- und Informationstechnik von der Technischen Universität München. Er ist Professor für Informatik und den Entwurf integrierter Schaltungen an der Hochschule Aschaffenburg.

hochschule aschaffenburg
university of applied sciences

Highly Flexible FPGA-Architecture of a Support Vector Machine

Markus Berberich, Konrad Doll

***Abstract*—Complex classification tasks like pedestrian detection in driver assistance systems require the processing of large amounts of image data. The nature of such classifications implies the requirement of real time data processing which can easily overload conventional computation systems or rather leads to the need of high performance processors. The work presented in this paper implements a complete Support Vector Machine (SVM) for the above described classification tasks on a reconfigurable FPGA hardware. The proposed architecture is highly flexible concerning input data format, Kernel selection, dimension and number of support vectors and thus is suitable for a variety of classification problems. It can be parameterized during run-time for linear, polynomial as well as radial bias function (RBF) Kernel computation. The massively parallel implementation of the scalar product calculation solves the real-time needs whereby the required resources fit on a moderately sized FPGA device which results in a cost and energy effective solution.**

***Index terms*—Support Vector Machine, FPGA.**

I. Introduction

The progress in computer science and image processing systems allows for a variety of new image classification applications like pedestrian detection in driver assistance systems. Traditional image processing problems are based on analytical models to detect a specific pattern inside the image. Examples for such applications are the recognition of defects or missing parts in production plants. This approach doesn't fit for applications like pedestrian detection because there is no sufficient analytical description available due to the complexity of the classification task and the variance of ambient conditions. This leads to the requirement of a very robust classification algorithm that is based on empirical image data.

M. Berberich, markusberberich@gmx.de, is employee at Leica Biosystems Nussloch GmbH, Heidelberger Str.17-19, 69226 Nussloch. K. Doll, konrad.doll@h-ab.de, is member of Hochschule Aschaffenburg, Fakultät Ingenieurwissenschaften, Labor für Rechnergestützten Schaltungsentwurf, Würzburger Straße 45, 63743 Aschaffenburg.

The utilized algorithm in this work is the SVM which has shown outstanding performance in several classification tasks like handwritten digit recognition on mail envelopes (U.S. postal service). An experimental study on pedestrian detection comparing different classifiers is presented in [1]. Another successful SVM based example for pedestrian classification is presented in [2].

The SVM algorithm is very flexible concerning the application area. It can solve numerous pattern recognition tasks also in other areas than image recognition. This leads to the requirement for being flexible concerning the input data that must be processed. Due to the empirical characteristic of an SVM the amount of data that has to be processed can be very high. Combining this with the real-time requirement of the classification task, there is a need for high computation power to solve the problem in a timely manner. This high computation power can be found in FPGA devices. Due to the possibility of parallel computation they outperform ordinary processors at much lower frequencies.

Some works in this field like [3] use the FPGA device only as co-processor for the dot product calculation. The work presented in [4] focuses on simple datasets with less parallelization and a linear Kernel function. Another recent work described in [5] focuses on more generic applications. However this work instantiates a separate logic block for every support vector. This leads to high resource requirements for larger problems combined with a need for FPGA reconfiguration every time the number of support vectors change. The work presented in this paper delivers a highly flexible utilizable SVM core that can address a variety of classification tasks. It covers the most important Kernel functions and can be easily integrated in other projects. The realized SVM core covers the complete feed-forward phase instead of working as a co-processor. It addresses dynamically selectable Kernel functions with scalable input data and offers a fixed size of required logic resources independent of the classification task. In section II we describe the basics of SVMs. Section III describes the proposed architecture of the SVM core. In section IV we give a deeper insight into the implementation of the SVM core. In Section V we present the results whereas section VI gives a final conclusion.

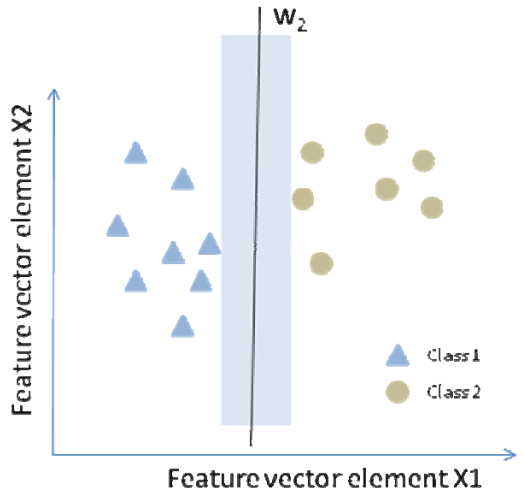


Fig. 1: Optimum hyperplane location

II. Support Vector Classification

The SVM is a supervised learning algorithm which consists of training and a feed-forward phase. It is ideally suited for binary classification problems but can also be extended to multi class problems. During the training phase a training data set of M data elements xi and the known corresponding class label yi is applied. The data **x** is an N-dimensional feature vector. We call this dimension "feature dimension". A feature vector can be a single image. For an exemplary image of 28x28 pixel with 8 Bit grayscale, we get a feature dimension of 784 grayscale values. For a binary classification problem the class label has the domain

$$Y_i = \pm 1$$

The SVM training algorithm which is out of scope of this paper, tries to separate the data with different classes by creating an optimum hyperplane. This is shown in fig. 2 for two-dimensional feature vectors. The training algorithm calculates the hyperplane that seperates the two classes optimally, i.e. with a maximum distance to the hyperplane between the nearest data points of each class. This distance is called margin. The margin plays an important role for the generalization ability during the feed-forward phase. We can state that the algorithm gets more and more robust against noise the bigger the margin is.

Another aspect is the linearly inseparable case as illustrated in Fig. 2 above. The data points of class 1 and class 2 cannot be separated by a hyperplane. One solution to overcome this problem is to map the input data space into a higher dimensional feature space where such a hyperplane exists (see Fig. 2 below). This can be realized with Kernel functions.

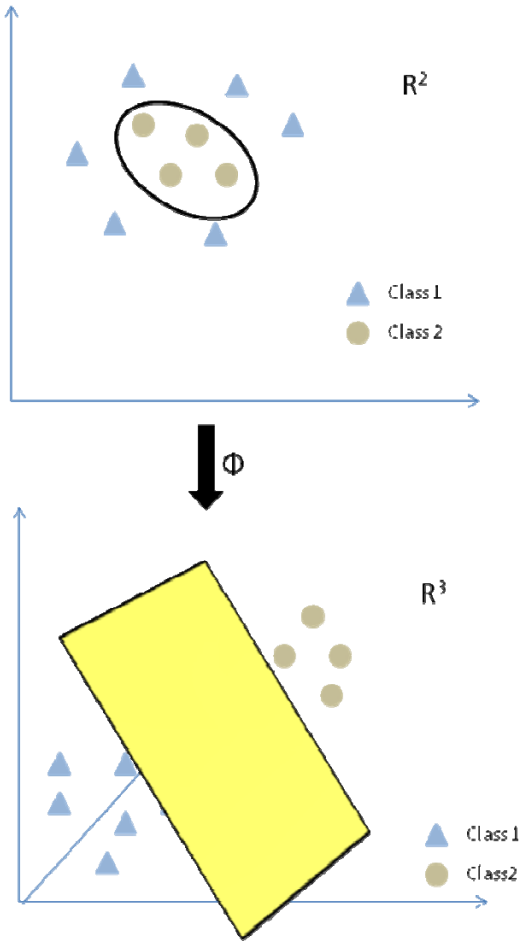


Fig. 2: Linearly inseperable cace

The mathematical formulation for the training phase is as follows.

Maximize:

$$W(\boldsymbol{\alpha}) = \sum_{i=1}^{M} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{M} \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j) \qquad (1)$$

with the constraints $\sum_{i=1}^{M} \alpha_i y_i = 0$ and $\alpha_i > 0$

$y_i = ClassLabel$
$\alpha_i = LagrangeMultiplier$
$\mathbf{x}_i = FeatureVector$
$k(...) = KernelFunction\,\mathrm{k}(\ldots)$

The result of the constrained quadratic programming problem is a set of feature vectors with corresponding class labels and Lagrange multipliers which can be regarded as input to the feed-forward phase in which a

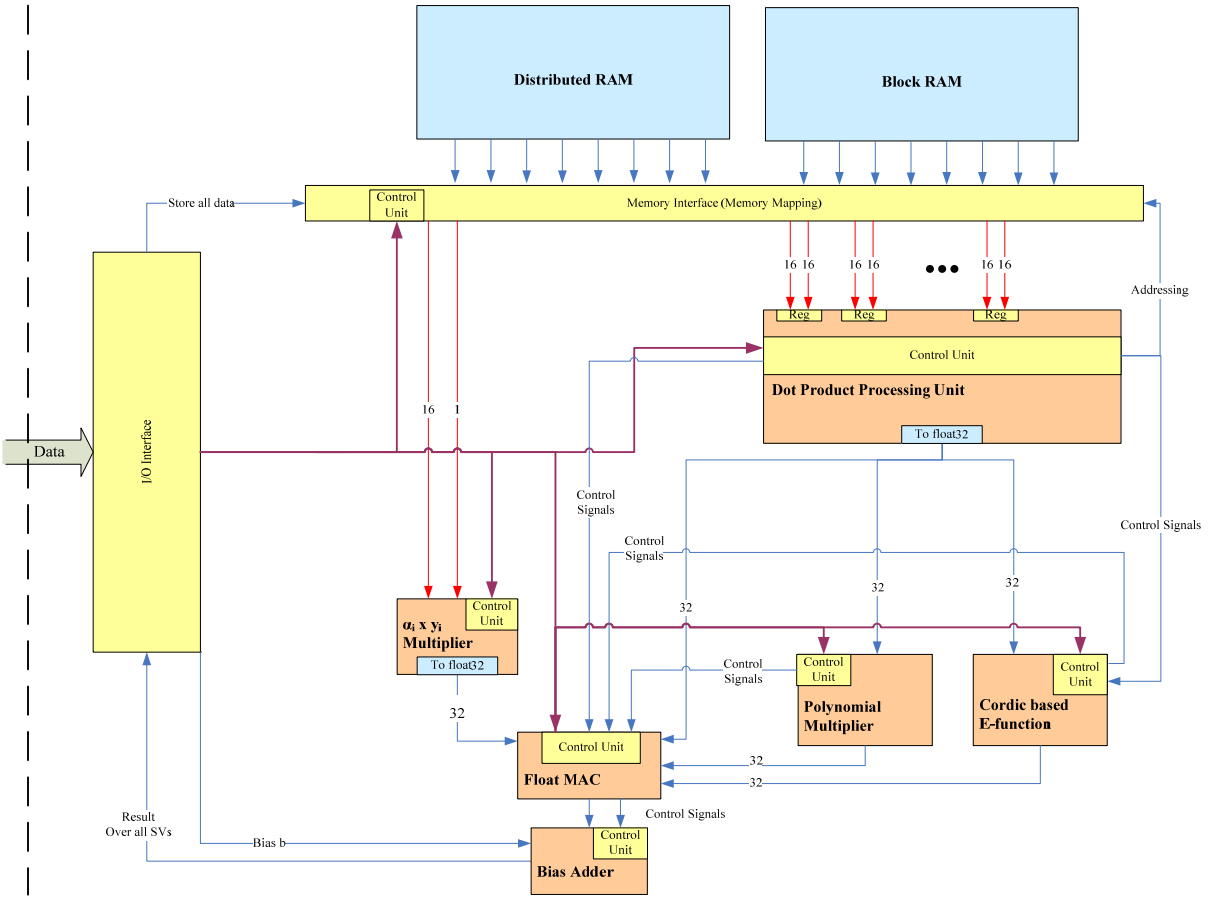


Fig. 3: Top level diagram of SVM architecture

decision function is calculated to determine if a new input vector corresponds to class 1 or class 2. This set of feature vectors is a subset of the complete training set and named as support vectors. The decision function can be described as follows:

$$D(\mathbf{x}) = \mathrm{sgn}\left(\sum_{i=1}^{S} \alpha i y i k(\mathbf{x}i, \mathbf{x}i) + \mathrm{b}\right) \qquad (2)$$

with S = Number of support vectors
b = Bias.

The calculation of this decision function must be processed in real time where the number of support vectors can vary between a few tens and a few thousands of vectors. The work described in this paper implements the linear, polynomial and RBF Kernels:

Linear Kernel: $k(\mathbf{x}, \mathbf{x}i) = \langle \mathbf{x}, \mathbf{x}i \rangle$

Polynomial Kernel: $k(\mathbf{x}, \mathbf{x}i) = \langle \mathbf{x}, \mathbf{x}i \rangle^d$

RBF Kernel: $k(\mathbf{x}, \mathbf{x}i) = e^{\left(\frac{\|\mathbf{x}i - \mathbf{x}\|}{2\sigma^2}\right)}$

III. Architecture

The architecture described in this section realizes the calculation of the decision function by fulfilling the following requirements at the same time:

- Linear, polynomial and RBF Kernel must be dynamically selectable during run-time.
- Number of support vectors and feature dimension is dynamically selectable.
- Input data can be scaled to fit the desired application.
- Sufficient performance is required to allow also the computation of problems with a large number of support vectors.
- The core must be easy to integrate into other projects.

Fig. 4 shows the top level block diagram of the SVM FPGA architecture. The particular modules will be described subsequently.

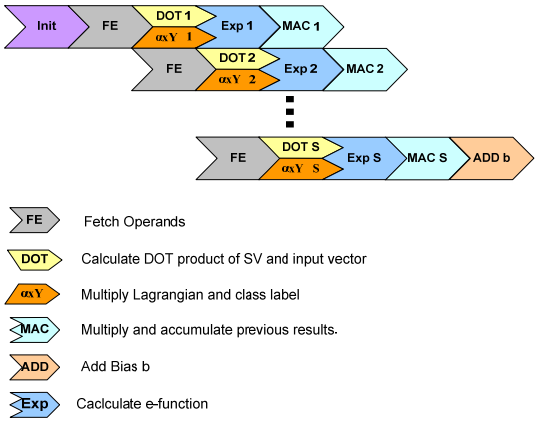


Fig. 4: Pipeline structure for RBF Kernel

The system is encapsulated and provides an interface to allow the communication and data transfer with the SVM core. Before any classification can be started there are a couple of parameters and data necessary like kernel type or support vectors. Classification can be started when these parameters and data are loaded into the core. During classification new input vectors to be classified must be loaded into the SVM.

The data is stored in the internal block RAM and distributed RAM resources of the FPGA. This allows for an easy implementation. However it limits the number of data that is storable. The "Memory Interface" module could be modified in future implementations to overcome this limitation by integrating DDR-RAM controllers.

The SVM architecture is completely designed as a pipeline structure which is shown for the RBF Kernel in Fig. 4. After the initialization, the first 64 elements of the support vector and input vector are fetched. The dot product is calculated in parallel to the multiplication of the Lagrange multiplier and the class label. The highest calculation effort needs the dot product which scales directly with the dimension and number of support vectors. The result of the "Dot Product Processing Unit" is fed to the base-e-exponential module "CORDIC based E-Function". The module "Float MAC" multiplies the two previous results (Lagrange – class label multiplication and exponential result) and accumulates them over all support vectors. Finally the bias is added to obtain the classification result for one input vector.

Inside the SVM core there are two different number formats that are used. Whereas a fixed point number format is used for the dot product processing, a single precision floating point format is used for the other components. This is necessary to avoid overflow problems within the SVM.

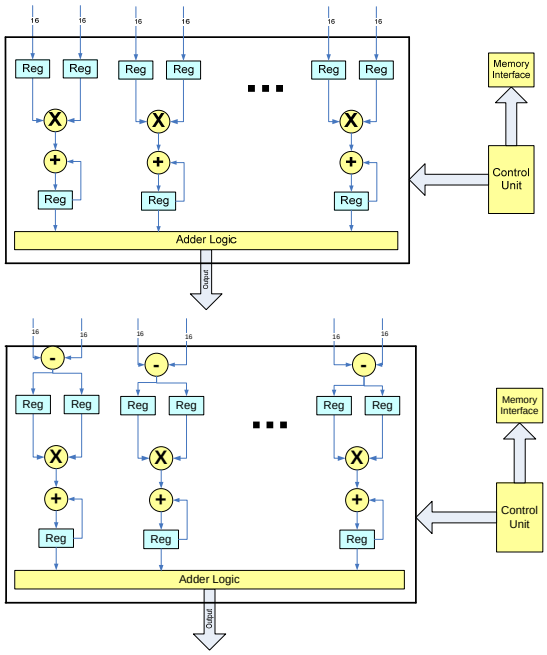


Fig. 5: Dot Product Processing Unit

A. Dot Product Processing Unit

The block diagram of the dot product calculation can be seen in Fig.5. The upper figure shows the diagram for a linear or polynomial Kernel whereas the lower figure belongs to the RBF Kernel.

Although there are two different versions of the calculations it is just one module. The subtraction part of the lower diagram is skipped if a non RBF Kernel is selected. As mentioned earlier most of the performance power is required in this module. 64 parallel (subtraction)-multiplication-accumulator calculation paths exist to speed-up the calculation. Each path processes a sub-part of the total dot product result. This requires 64 support vector and input vector elements to be present at the input ports which are multiplied and accumulated. This procedure is repeated until one complete support vector is being processed. Then all the sub-results are added to a single result which is converted to the 32 Bit floating point format and subsequently scaled by the scale factor. The control unit handles the pipelining in this module by generating the necessary control signals for the arithmetic units.

B. Memory Interface

The memory interface solves two different tasks. First it is responsible for the storage of the support vectors, input vector, Lagrange multipliers and class labels when they are loaded into the SVM core from outside. The second task is to provide the required operands to the dot product processing unit.

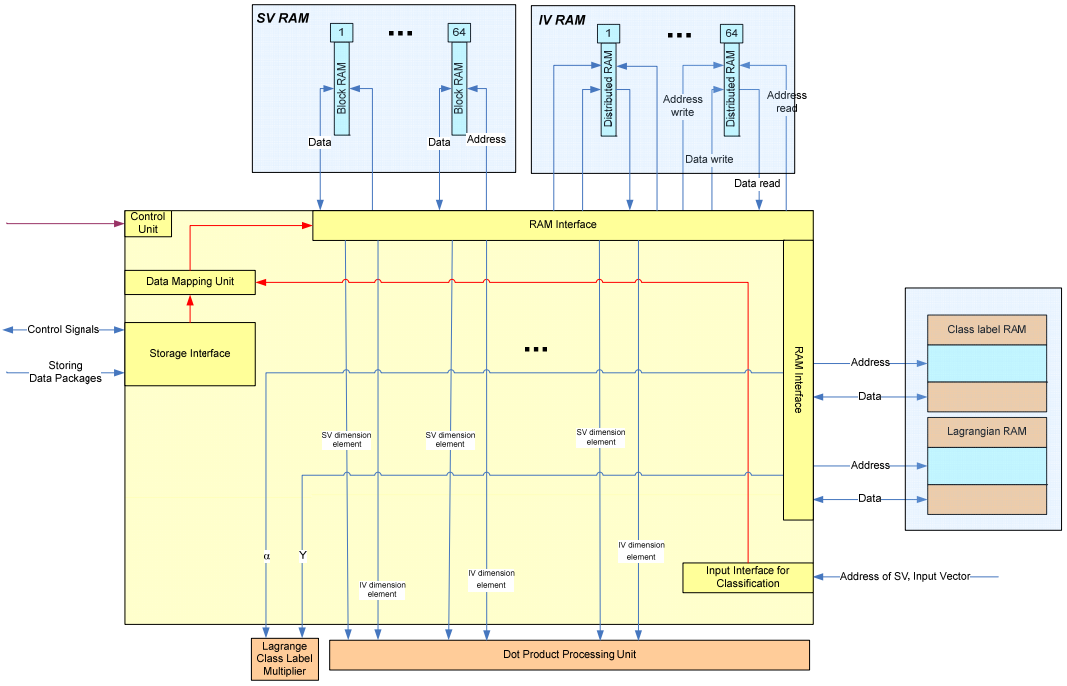


Fig. 6: Memory Interface module

Due to the 64 parallel calculation paths, there is a need to provide 64 support vector elements and 64 input vector elements simultaneously to the dot product processing unit which can be seen in the bottom part of Fig. 6. Otherwise the memory bandwidth would become a bottleneck. This has several implications: The support vectors that are stored in the block RAM of the FPGA device („SV RAM" in Fig. 6) must be distributed over 64 separate block RAM components with an independent address- and data bus for each RAM. The input vectors must also be distributed over 64 independent RAM components. Additional they have to be implemented as dual-port RAMs to allow read and write accesses at the same time. The read access is required to provide the input vector elements to the dot product processing unit. The write access is necessary to allow the storage of a new input vector while the current one is being processed. Therefore the address space is divided into two areas, one area for the current input vector and a second one as buffer for the next input vector.

Another implication results from the choice of 64 parallel paths. The feature dimension must always be a multiple of 64. Non used elements in the support vector respectively input vectors must be filled with zeros.

C. Other Modules

In this section all other modules in Fig. 3 are described.

Lagrange Class Label Multiplier:

The Lagrange multiplier and the class label are multiplied once per support vector in parallel to the dot product. The result is converted into a floating point format.

Polynomial Multiplier:

The dot product is raised to the power of the parameter "polynomial degree" which is loaded into the SVM core. The exponentiation is realized with a series of sequential multiplications.

Cordic based e-Function:

First the dot product is divided by σ. The result is the exponent for the exponential function which is computed using the CORDIC algorithm.

Float MAC:

According to the Kernel selection the result of the dot product, polynomial or e-function module is selected as input. It is then multiplied with the result of the Lagrange class label multiplication and accumulated over all support vectors

Bias Adder:

The last module in the processing chain adds the bias in the floating point domain and forwards the result to the output of the SVM.

Fig. 7: SVM core port map

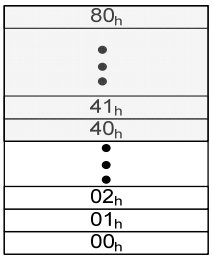


Fig. 8: Address ranges of input vector RAM module

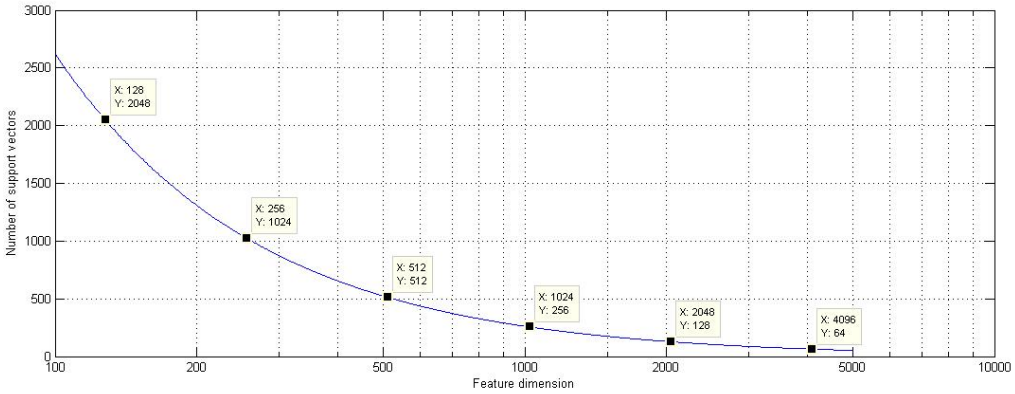


Fig. 9: Support vector storage capability

IV. Implementation

The above described SVM architecture has been implemented with VHDL-93 on a XC5VLX110T device from Xilinx. The SVM core can be integrated in other projects. The corresponding port connections are shown in Fig. 7. The port SVMState is used to control the state of the SVM core. Three states are available: Init, store and classify. During initialization the parameters shall be applied. During the state "store" the support vector data, the first input vector, the Lagrange multipliers and the class labels are saved into the internal RAM modules. In the classification state incoming input vectors are processed and the result is delivered to the corresponding output port. The port signal NewOnlineIVStorage is used to signal the SVM core that a new input vector shall be stored during the state "classify". The data to be loaded into the SVM must be applied to the 512Bit storage interface. The port ClassificationResultValid indicates that an input vector has been processed.

A. Memory Interface

The FPGA device XC5VLX110T provides 5328 kBits of block RAM resources. However, this amount of RAM cannot be utilized completely due to data alignment and the required 64 independent parallel storage modules. Our implementation instantiates 64 block RAM modules with 65536 Bit per module which results in 4 MBit in total for storing support vectors.

The input vectors are stored in the distributed RAM. Each of the 64 dual port modules is divided into address ranges as shown in Fig. 8.

This allows for holding one input vector in the buffer to keep the pipeline filled when a current classification process is finished. Each module can store 64 words at 16 Bits per address range which results in a maximum feature dimension of 4096 words. Class labels and Lagrange multipliers are stored in the residual block RAM resources. The maximum number of class labels and Lagrange multipliers that can be stored is 4096 which limits the maximum number of

support vectors also to 4096. The relation between feature dimension and number of support vectors that can be stored is shown in the Fig. 9.

B. Dot Product Processing

The 64 parallel subtraction units of the RBF Kernel SVM are implemented completely with LUT resources of the FPGA. The MAC units in this module are realized with the multiplier and adder inside the hard-macro DSP48E slices of the FPGA which deliver a 48 Bit result. The maximum parallelization is limited to the 64 available DSP slices of the FPGA device used in this work.

The sub-results of the calculation path are summed up to the dot product with a synchronous adder tree logic that has been implemented with LUT resources. This results in a 54 Bit fixed point result which is then converted to a 32 Bit floating point format.
Before forwarding the result to subsequent arithmetic modules, it is scaled with the "ScaleFactor" due to the internal multiplication in the dot product calculation. The result is forwarded to the exponential function module, the polynomial exponentiation module and to the floating point MAC stage where it is further processed. The further description follows the path for an RBF Kernel selection.

C. Exponential Function

The exponential function of the dot product has to be calculated if the RBF Kernel is selected. The dot product result has to be scaled with the parameter $2\sigma^2$ before the exponential function can be calculated.

$$y = e^x; x = \frac{Dotproduct}{2\sigma^2} \tag{3}$$

The exponential function is traced back to the hyperbolic sine and cosine function which can be calculated with the CORDIC algorithm

$$\begin{aligned} e^x &= e^{(a+m*\ln 2)} = e^a * e^{m*\ln 2} = 2^m * e^a \\ &= 2^m * [\cosh(a) + \sinh(a)] \end{aligned} \tag{4}$$

The steps required to calculate these formulas are illustrated in Fig. 10. The scaling of the dot product is realized with the reciprocal value of $2\sigma^2$ for performance reasons. Afterwards the overflow detection checks if the exponent is within the possible range regarding a single precision floating point format as result. The CORDIC algorithm is realized by utilizing an IP core from Xilinx which expects the input data in the 2QN format which leads to the need of a conversion stage. The accuracy of the CORDIC result depends on the number of decimal places of the input data. A value of 16 Bits showed satisfying results

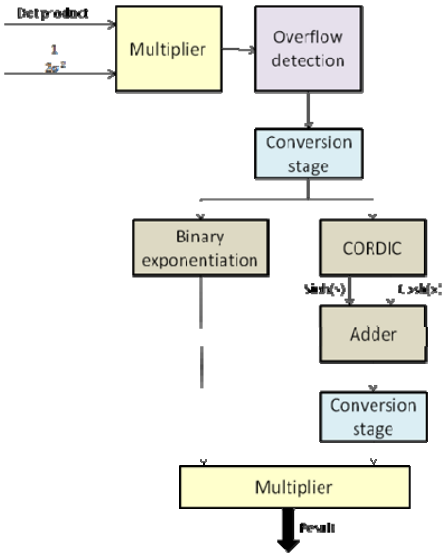


Fig. 10: Block diagram of exponential function module

(2Q16). Simultaneously the binary exponentiation according to Equation (4) is computed by shifting the exponent fraction of the floating point format. The multiplier stage works in the floating point domain which makes it necessary to add another conversion stage after the CORDIC results are added.

D. Float MAC Stage

This module selects the appropriate input operand and "data valid" signal with 32 Bit or 1 Bit multiplexers according to the selected Kernel function. The floating point multiplier and adder are realized with IP cores from Xilinx. An internal counter counts the number of MAC operations that are done until the number of support vectors is reached. Each MAC operation is triggered from a "data valid" signal of the previous stages. The result is output to the subsequent bias adder and the result valid signal is set. The bias adder performs then the last operation and forwards the final result to the output port of the SVM core.

V. RESULTS

The presented design has been successfully simulated and verified in hardware using the proposed FPGA device XC5VLX110T. The achieved accuracy for the RBF Kernel computation has an upper bound of 10 ppm. The following resources are required by the core:

Slice Registers:	12674
Slice LUTs:	41135
36Kb Block RAMs:	132
Number of DSP48Es:	64

Table 1: Latency times for 92 MHz clock

	92 MHz				
Number of support vectors	64	128	512	1024	2048
Feature dimension	64	128	512	1024	2048
Response time linear Kerne [µs]	1,01	3,10	44,84	178,40	712,66
Response time poly Kerne [µs]	1,05	3,14	44,88	178,45	712,71
Response time RBF Kerne [µs]	1,54	3,63	45,37	178,93	713,20

Table 1: Latency times for 50 MHz

	50 MHz				
Number of support vectors	64	128	512	1024	2048
Feature dimension	64	128	512	1024	2048
Response time linear Kerne [µs]	1,86	5,70	82,50	328,26	1311,30
Response time poly Kerne [µs]	1,94	5,78	82,58	328,34	1311,38
Response time RBF Kerne [µs]	2,84	6,68	83,48	329,24	1312,28

The maximum frequency stated by the timing analysis is 92 MHz. The design has been verified with 50 MHz in hardware on an ML505 evaluation board. Tables 1 and 2 list various latency times (time from classification start of new input vector to classification result) depending on feature dimension, number of support vectors and clock frequency.

VI. Conclusion

This paper describes a successful implementation of an SVM core for multi-purpose classification tasks which is highly flexible concerning the input data and type of Kernel function. The core can be integrated in a variety of projects such as the pedestrian detection for driver assistance systems.
Possible future work contains the testing of the SVM core with the maximum frequency of 92 MHz. Additionally further improvements are conceivable for future research to enhance the maximum frequency beyond 92 MHz. Applications that require a large feature dimension and a large number of support vectors at the same time would exceed the current RAM resource limitations. A possible solution to overcome this problem is the usage of external memory.

References

[1] S. Munder, D. M. Gavrila: “An Experimental Study on Pedestrian Classification”, *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 2006

[2] S. Bauer, S. Köhler, K. Doll, U. Brunsmann: “FPGA-GPU Architecture for Kernel SVM Pedestrian Detection”, *IEEE Conference on Computer Vision and Pattern Recognition*, San Francisco, 2010

[3] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, H.P. Graf: “A Massively Parallel Digital Learning Processor”, *22th Annual Conference on Neural Information Processing Systems*, Vancouver, 2008

[4] O. Pina-Ramirez, R. Valdes-Cristema, O. Yanez-Suarez: “A FPGA Implementation of Linear Kernel support Vector Machines”, *IEEE International Conference on Reconfigurable Computing and FPGA’s*, 2006

[5] M. Ruiz-Llata, G. Guarnizo, M. Yebenes-Calvino: “FPGA Implementation of a Support Vector Machine for Classification and Regression”, *World Congress on Computational Intelligence*, Barcelona, 2010

[6] B. Schölkopf, A.J. Smola: „Learning with Kernels”, MIT-Press, Cambridge, 2002

[7] CORDIC.Core: Stand 24. April 2009Xilinx DS249, CORDIC v4.0 Product Specification

[8] D. Anguita, A.Boni, S. Ridella: “A Digital Architecture for Support Vector Machines: Theory, Algorithm, and FPGA Implementation”, *IEEE Transactions on Neural Networks*, 14(5) 993-1009

[9] J. Manikandan, B. Venkatramani, V. Avanthi: „FPGA Implementation of Support Vector Machine Based Isolated Digit Recognition System”, *22nd International Conference on VLSI Design*, New- Delhi, 2009

HOCHSCHULE KONSTANZ
TECHNIK, WIRTSCHAFT UND GESTALTUNG

Einsatz von RAM-Blöcken als mikroprogrammierte Steuerwerke in FPGAs

Christian Kielmann, Dominik Stengele, Irenäus Schoppa

***Zusammenfassung*—Die meisten modernen, hochkomplexen programmierten Logikbausteine vom Typ FPGA sind heute mit zahlreichen RAM-Blöcken in Form von sog. primitiven Komponenten ausgestattet. Häufig werden diese RAM-Blöcke als interne Pufferbereiche zur Datenspeicherung eingesetzt. Allerdings werden nicht in jeder Applikation alle RAM-Blöcke vollständig ausgelastet. Oft bleiben einige von ihnen völlig ungenutzt. Eine interessante Möglichkeit ist es, diese RAM-Blöcke als Mikroprogrammspeicher in mikroprogrammierten Steuerwerken zu verwenden, wodurch sich auch noch Logikzellen einsparen lassen.**

***Schlüsselwörter*—FPGA, Spartan-3, mikroprogrammierte Steuerwerke, Mikroprogrammspeicher, KCUART, M68HC05.**

I. Einleitung

In den meisten modernen, hochkomplexen FPGA-Bausteinen sind zahlreiche RAM-Blöcke zu finden. Die Tabelle 1 gibt uns einen Überblick über die Anzahl von RAM-Blöcken und deren Kapazität in den aktuellen fünf FPGA-Familien von Xilinx [7]. Innerhalb einer FPGA-Familie hängt die Anzahl der RAM-Blöcke mit der Größe eines Bausteins zusammen. So enthält bspw. der kleinste Spartan-3 vom Typ XC3S50 nur 3 RAM-Blöcke. Im größten Spartan-3 vom Typ XC3S5000 sind insgesamt 126 RAM-Blöcke verfügbar [8] und [9].

Neben dem Einsatz solcher RAM-Blöcke als interne Pufferbereiche (FIFOs) zur Datenspeicherung gibt es laut [1] auch weitere Anwendungsmöglichkeiten, wie Code-Konverter (µ/A-Law), schnelle Funktionsgeneratoren (SIN/COS-Lookup-Tabellen), BCD-Zähler/-Konverter oder mikroprogrammierte Zustandsmaschinen.

II. RAM-Blöcke

Die meisten modernen FPGAs verfügen über verschiedene On-Chip-Speicher-Ressourcen, die als RAM- oder ROM-Blöcken organisiert werden können. Die Speicherblöcke haben eine typische Blockgröße von 18 Kbit, die in Spalten organisiert sind. Auf die RAM-Blöcke kann synchron lesend und schreibend zugegriffen werden. Die Größe des zur Verfügung stehenden Speichers, hängt in erster Linie von der Größe des verwendeten FPGAs ab. Die Architektur von RAM-Blöcken auf einem FPGA, kann dabei als Single-Port-Ram (SPR) oder Dual-Port-Ram (DPR) realisiert werden.

Tabelle 1: Anzahl von RAM-Blöcken und deren Kapazität in den einzelnen FPGA-Familien von Xilinx [7].

Familie	# RAMB	Blockgröße
Virtex-6	156 - 1064	36 kbit
Virtex-5	26 - 516	36 kbit
Virtex-4	36 - 552	18 kbit
Spartan-6	12 - 268	18 kbit
Spartan-3	3 - 126	18 kbit

Der DPR verfügt über zwei physikalisch voneinander getrennte Ports. Die Struktur dieser Ports ist symmetrisch und beide unterstützen jeweils Lese- und Schreibzugriffe. Jeder Port ist an ein eigenes Bussystem angeschlossen, über das Daten oder Adressen übertragen werden. Des Weiteren kann jeder Port über einen weiteren Takt verfügen. Erfolgt die Realisierung des DPR mit einem gemeinsamen Takt, ist es wichtig zwei Arten von Problemen zu beachten um Konflikte durch die beiden Ports zu verhindern. Ein Problem entsteht wenn beide Ports auf die gleiche Adresse und im gleichen Takt schreiben möchten. Kommt es zu dieser Situation und es ist keine höhere Priorität für einen Port vergeben, erfolgt ein unbestimmter Schreibzugriff auf die Adresse. Ein weiteres Problem entsteht, indem ein Port schreibt und der andere Port im gleichen Takt an derselben Adresse lesen möchte.

Dafür unterstützen die RAM-Blöcke drei Modi, um sicherzustellen, welche Daten am Ausgang zu Verfügung gestellt werden. Der Standartmodus ist bei Xilinx FPGAs, aufgrund der Abwärtskompatibilität, auf den Write-First-Modus festgelegt. Im Write-First-Modus werden die zu schreibenden Daten an die Adresse geschrieben und gleichzeitig an den Ausgangsport für die Datenausgabe weitergeleitet, siehe Abbildung 1. Der zweite Modus ist der Read-First-Modus.

Ch. Kielmann, chkielma@htwg-konstanz.de, und D. Stengele, dostenge@htwg-konstanz.de sind Studenten an der HTWG-Konstanz, I. Schoppa, ischoppa@htwg-konstanz.de, ist Mitglied der HTWG Konstanz, Brauneggerstr. 55, 78462 Konstanz.

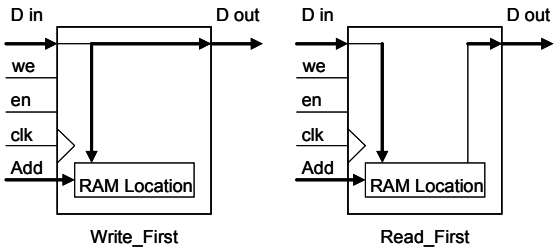


Abbildung 1: Konfiguration der RAM-Blöcke [10].

```
-- Write_First für einen Port.
process (clk)
  variable a0: integer;
begin
  if rising_edge(clk) then
    a0 := to_integer(unsigned(add));
    if we = '1' then
      mem(a0) := din;
      dout <= din;
    else
      dout <= mem(a0);
    end if;
  end if;
end process;

-- Read_First für einen Port.
process (clk)
  variable a0: integer;
begin
  if rising_edge(clk) then
    a0 := to_integer(unsigned(add));
    dout <= mem(a0);
    if we = '1' then
      mem(a0) := din;
    end if;
  end if;
end process;
```

In diesem werden die Daten, welche vorher an der Schreibadresse gespeichert war, weitergeleitet. Somit erscheinen die älteren Daten am Datenausgang, während die neuen an die entsprechende Adresse geschrieben werden. Der Read-First-Modus ist aufgrund der Effizienz der empfohlene Betriebsmodus. Der dritte Modus ist der No-Change-Modus, der während einer Schreiboperation seinen Ausgang abschaltet. Dieses Verhalten ähnelt einem konventionellen Speicher, wo während eines Taktzyklus entweder gelesen oder geschrieben werden kann.

Core-Generatoren sind Modul- oder Codegeneratoren, mit denen Basismodule der Digitaltechnik mittels einer grafischen Oberfläche erzeugt werden können. Diese Module lassen sich anschließend relativ einfach in einem VHDL-Entwurf einbinden. Mit dem Core-Generator von Xilinx lassen sich Single- als auch Dual-Port-RAMs erzeugen. Für die Generierung eines RAM-Block werden der Name, die benötigten Steuersignale und die Größe definiert. Für die Initialisierung kann ein beliebiger Speicherinhalt vorgegeben werden, ansonsten erfolgt eine Initialisierung der Speicherzellen mit 0. Die Übergabe des spezifischen Speicherinhalts erfolgt über eine binäre Initialisierungsdatei durch den Core-Generator. Alternativ dazu lässt sich auch eine sog. Koeffizientendatei (.coe) erstellen, die nicht nur den Inhalt des Speichers definiert, sondern auch Steuerparameter für den Core-Generator enthält. Der Core-Generator erzeugt anschließend Dateien für die Steuerung des Implementierungsvorgangs. Der Implementierungsvorgang wiederum erzeugt eine Bit-Datei, die zur Konfiguration des FPGA verwendet wird.

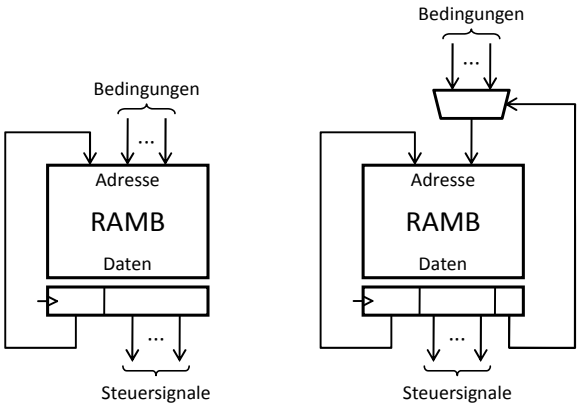


Abbildung 2: Prinzipieller Aufbau von Steuerwerken auf der Grundlage synchroner RAM-Blöcke mit paralleler und sequentieller Abfrage von Eingangsbedingungen.

III. Prinzip der Mikroprogrammierung

Der klassische Entwurf eines Steuerwerks auf der Basis festverdrahteter Logik kann bei komplexen digitalen Systemen mit zahlreichen Zuständen und Zustandsübergängen, sowie mit vielen Bedingungen/Statussignalen schnell recht aufwändig werden. Mit der steigenden Komplexität erhöht sich oft auch die Fehleranfälligkeit und sinkt die Flexibilität sowie die Erweiterbarkeit. Anstelle eines festverdrahteten Steuerwerks kann deshalb oft ein Steuerwerk auf der Grundlage eines (Mikroprogramm-)Speichers verwendet werden. Dabei wird ein (Steuerungs-)Algorithmus als Programm im Mikroprogrammspeicher des Steuerwerks abgelegt und als Folge von (Mikro-) Befehlen abgearbeitet.

In diesem Beitrag wird vor allem auf diejenigen Architekturen mikroprogrammierter Steuerwerke eingegangen, die sich besonders gut zur Realisierung mit synchronen Speichern (also RAM-Blöcken in FPGAs) eignen. Eine umfassende und praxisorientierte Beschreibung spezieller, mikroprogrammierter Steuerwerke im Prozessorentwurf und für Steuerungsanwendungen findet man in [5]. Einen sehr guten Überblick über die Synthese mikroprogrammierter Steuerwerke für Rechenwerke in universellen und speziellen Prozessoren gibt es auch in [4] und [6].

Auf der Grundlage eines synchronen RAM-Blocks lässt sich besonders einfach ein mikroprogrammiertes Steuerwerk mit paralleler Abfrage von Eingangsbe-

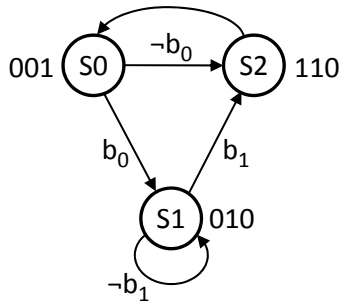


Abbildung 3: Zustandsgraph eines einfachen Moore-Automaten mit drei Zuständen.

dingungen realisieren. In einem solchen Fall kommt nur der RAM-Block alleine zum Einsatz, ganz ohne zusätzliche Logik. Eine leicht modifizierte Variante ist ein mikroprogrammiertes Steuerwerk mit sequentieller Abfrage von Eingangsbedingungen. Hier erfolgt die Auswahl einer aus mehreren Eingangsbedingungen mit Hilfe eines 1-aus-n-Multiplexers. In der Abbildung 2 sind die beiden Varianten dieser Steuerwerke schematisch dargestellt.

In den folgenden Abschnitten werden der Entwurf und die Implementierung eines mikroprogrammierten Steuerwerkes an einem einfachen Beispiel systematisch aufgezeigt.

A. Entwurf

In der Abbildung 3 ist der Zustandsgraph eines einfachen Moore-Automaten gezeigt. Der Automat ist definiert durch drei Zustände S0, S1 und S2, fünf Zustandsübergänge, zwei Eingangsbedingungen b_0 und b_1 sowie drei Ausgangsvektoren $Y_0 := (001)$, $Y_1 := (010)$ und $Y_2 := (110)$.

Bei der Realisierung dieses Automaten mit einem mikroprogrammierten Steuerwerk werden zuerst die symbolischen Zustände binär codiert. Die drei Zustände S0, S1 und S2 lassen sich mit einem zweistelligen Vektor *state* = (s_1, s_0) codieren, z.B. S0 := (00), S1 := (01) und S2 := (10). Durch die Konkatenation der codierten Zustände mit den beiden Eingangsbedingungen *cond* = (b_1, b_0) wird hier ein vierstelliger Vektor gebildet, der positionsweise den einzelnen Adressbits zugeordnet wird, z.B. (a_3, a_2, a_1, a_0) := (s_1, s_0, b_1, b_0). Mit der so gebildeten effektiven Adresse wird der RAM-Block adressiert. Die binär-codierten Adressen sind im linken Teil der Tabelle 2 aufgelistet. Im rechten Teil dieser Tabelle ist der binär-codierte Speicherinhalt zu sehen. Jedes Datenwort (d_4, d_3, d_2, d_1, d_0) wird positionsweise der zweistelligen Folgeadresse *next* = (s_1^+, s_0^+) := (d_4, d_3) und dem dreistelligen Ausgangsvektor *outs* = (y_2, y_1, y_0) := (d_2, d_1, d_0) zugeordnet.

Tabelle 2: Steuertabelle des einfachen Moore-Automaten aus der Abbildung 3.

	state		cond		next		outs		
	s_1	s_0	b_1	b_0	s_1^+	s_0^+	y_2	y_1	y_0
	a_3	a_2	a_1	a_0	d_4	d_3	d_2	d_1	d_0
0	0	0	0	0	1	0	0	0	1
1	0	0	0	1	0	1	0	0	1
2	0	0	1	0	1	0	0	0	1
3	0	0	1	1	0	1	0	0	1
4	0	1	0	0	0	1	1	1	0
5	0	1	0	1	0	1	1	1	0
6	0	1	1	0	1	0	1	1	0
7	0	1	1	1	1	0	1	1	0
8	1	0	0	0	0	0	0	1	0
9	1	0	0	1	0	0	0	1	0
10	1	0	1	0	0	0	0	1	0
11	1	0	1	1	0	0	0	1	0

B. Moore/Mealy -Transformationsalgorithmus

Da RAM-Blöcke in den FPGA-Familien von Xilinx als synchrone Speicher ausgelegt sind, muss man diese Besonderheit beim Entwurf des Mikroprogramms zwingend berücksichtigen. Wird ein Zustandsautomat als Moore-Automat entworfen, so lässt er sich relativ einfach in einen Mealy-Automaten mit Ausgangsflipflops mit Hilfe eines Algorithmus transformieren, und anschließend leicht mit einem synchronen Speicher realisieren.

Die Hauptidee dieses Algorithmus ist es, in einer Steuertabelle für einen Mealy-Automaten mit Ausgangsflipflops die Einträge für Ausgangsvektoren so umzusortieren, dass dessen Verhalten mit dem eines Moore-Automaten übereinstimmt. Der Moore/Mealy-Transformationsalgorithmus ist in Abbildung 4 dargestellt.

Die Datenstruktur *TFrame* definiert einen Eintrag für die Steuertabelle und enthält Datenfelder mit Information über den aktuellen Zustand *state*, Eingangsbedingungen *cond*, den Folgezustand *next* und Ausgangsvektoren *outs*. Auf der Grundlage dieser Datenstruktur sind zwei Tabellen namens *Moore* und *Mealy* deklariert worden. Beide Tabellen haben die gleiche Anzahl von Einträgen, wobei der Inhalt der *Moore*-Tabelle initialisiert werden muss, was hier nur angedeutet ist.

Der Algorithmus benutzt eine Hilfsfunktion namens *index*, die in einer *Moore*-Tabelle zeilenweise nach Einträgen sucht, in denen der Zustand *state* eines Eintrages mit dem als Parameter übergebenen Folgezustand *next* übereinstimmt. Bei einer gefundenen Übereinstimmung wird die passende Zeilennummer des Eintrages zurückgeliefert.

```
typedef struct {
  TState  state;
  TCond   cond;
  TState  next;
  TOuts   outs;
} TFrame;

TFrame Moore[N] = { . . . }
TFrame Mealy[N];

int index(TState next) {
  for (int i=0; i<N; i++)
    if (Moore[i].state == next)
      return i;
}

for (int i=0; i<N; i++) {
  int k = index(Moore[i].next);
  Mealy[i] = Moore[i];
  Mealy[i].outs = Moore[k].outs;
}
```

Abbildung 4: Notation des Moore/Mealy-Transformationsalgorithmus in C, ohne Initialisierungsinhalt.

Die Hauptschleife durchläuft zeilenweise alle Einträge in der *Moore*-Tabelle und übernimmt zuerst jeden Eintrag in die *Mealy*-Tabelle. Anschließend wird der Eintrag in der *Mealy*-Tabelle korrigiert, indem der Ausgangsvektor *outs* anhand der Zeilennummer aus der Hilfsfunktion *index* durch den entsprechenden Ausgangsvektor *outs* aus der *Moore*-Tabelle ersetzt wird. Das Ergebnis nach der Anwendung des Moore/Mealy-Transformationsalgorithmus auf der Tabelle 2 sind in der Tabelle 3 zu sehen.

C. Implementierung in VHDL und Synthese

Die Implementierung eines mikroprogrammierten Steuerwerks kann in VHDL mit Hilfe einer tabellarischen Beschreibung erfolgen. Eine solche Beschreibung ist zwar simulations- aber nicht synthesefähig. Sie ermöglicht aber dem Anwender, das Steuerwerk teilweise abstrakt zu formulieren, was vor allem bei sehr komplexen Steuerwerken die Lesbarkeit deutlich erleichtert.

In der Abbildung 5 sehen wir den Auszug aus dem Deklarationsbereich eines VHDL-Programms, in dem der Automat aus der Tabelle 3 modelliert ist. Aus dieser Beschreibung lassen sich mit Hilfsprozeduren passende Textdateien generieren: eine COE-Datei für den Einsatz mit dem Core-Generator von Xilinx sowie eine INIT-Datei mit Initialisierungsvektoren, die bei der Instanzierung der Komponente RAMB verwendet wird.

IV. Universal Asynchronous Receiver Transmitter

Die ersten beiden Untersuchungsobjekte sind ein UART mit einem festverdrahteten Steuerwerk sowie ein mit einem mikroprogrammierten Steuerwerk.

Tabelle 3: Steuertabelle des einfachen Automaten aus der Abbildung 3 unter Berücksichtigung von Ausgangsflipflops.

	state		cond		next		outs		
	s_1	s_0	b_1	b_0	s_1^+	s_0^+	y_2	y_1	y_0
	a_3	a_2	a_1	a_0	d_4	d_3	d_2	d_1	d_0
0	0	0	0	0	1	0	1	1	0
1	0	0	0	1	0	1	0	1	0
2	0	0	1	0	1	0	1	1	0
3	0	0	1	1	0	1	0	1	0
4	0	1	0	0	0	1	0	1	0
5	0	1	0	1	0	1	0	1	0
6	0	1	1	0	1	0	1	1	0
7	0	1	1	1	1	0	1	1	0
8	1	0	0	0	0	0	0	0	1
9	1	0	0	1	0	0	0	0	1
10	1	0	1	0	0	0	0	0	1
11	1	0	1	1	0	0	0	0	1

```
CONSTANT SIZE: natural := 16;
SUBTYPE TAdr IS natural RANGE 0 TO SIZE-1;

TYPE TWord IS RECORD
  adr:   std_logic_vector(1 DOWNTO 0);
  outs:  std_logic_vector(2 DOWNTO 0);
END RECORD;

CONSTANT rst_vector: TWord := ("00", "000");

TYPE TRam IS ARRAY (natural RANGE <>)
             OF TWord;

CONSTANT mem: TRam(0 TO SIZE-1) := (
  2#0000# => (adr=>"10",  outs=>"110"),
  2#0001# => (adr=>"01",  outs=>"010"),
  2#0010# => (adr=>"10",  outs=>"110"),
  2#0011# => (adr=>"01",  outs=>"010"),
  2#0100# => (adr=>"01",  outs=>"010"),
  2#0101# => (adr=>"01",  outs=>"010"),
  2#0110# => (adr=>"10",  outs=>"110"),
  2#0111# => (adr=>"10",  outs=>"110"),
  2#1000# => (adr=>"00",  outs=>"001"),
  2#1001# => (adr=>"00",  outs=>"001"),
  2#1010# => (adr=>"00",  outs=>"001"),
  2#1011# => (adr=>"00",  outs=>"001"),
  OTHERS  => rst_vector
);
```

Abbildung 5: Auszug aus dem Deklarationsbereich eines VHDL-Programms, mit dem der Inhalt des Mikroprogrammspeichers modelliert ist.

A. UART mit einem festverdrahteten Steuerwerk

Die Firma Xilinx stellt einen universellen, asynchron-seriellen Sender und Empfänger, einen sog. KCUART als IP-Module bereit, vgl. [2] und [3]. Der KCUART ist auf einen hohen Durchsatz hin entworfen und im Sende- und Empfangspfad mit je einem 16-Byte-FIFO (auf der Basis von Look-Up-Tabellen) ausgestattet. Beide IP-Module arbeiten mit festen Übertragungsparametern: einem Startbit, acht Datenbits und einem Stoppbit, ohne Parität, und sind somit mit dem Datenübertragungsformat der V24-

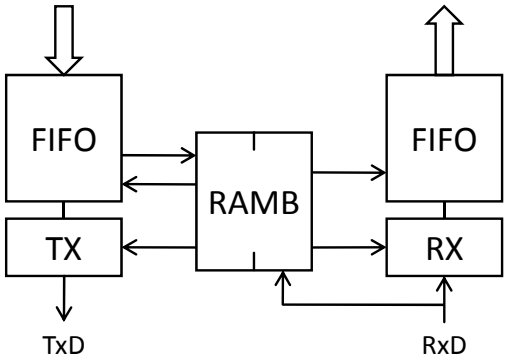


Abbildung 6: Blockschaltbild des UART-Moduls mit einem mikroprogrammierten Steuerwerk.

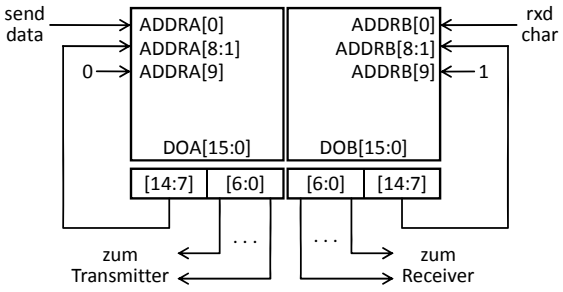


Abbildung 7: Steuerwerk im mikroprogrammierten UART.

Schnittstelle kompatibel. Die Baudrate lässt sich mit einem externen Baudratengenerator auf der Basis eines Modulo-Zählers einstellen. Auf diese Weise kann der KCUART bei einer Taktfrequenz von 50 MHz eine maximale Übertragungsgeschwindigkeit von bis zu 1,5625 Mbps erreichen. Die beiden IP-Module sind zwar modular in VHDL dafür aber vollständig mit primitiven FPGA-Komponenten als Netzlisten beschrieben.

Die Analyse des VHDL-Quelltextes des KCUART-Transmitters ergab, dass das Steuerwerk dort aus einigen Zustandsflipflops mit der dazugehörigen Übergangslogik sowie aus zwei Zählern besteht: einem 3-Bit-Zähler und einem Modulo-16-Zähler auf der Basis von SRL16. Mit dem 3-Bit-Zähler werden einzelne Datenbits beim Senden gezählt. Der Modulo-16-Zähler dient dazu, Strobe-Impulse aus dem Baudratengenerator zu zählen, so dass nach 16 Impulsen ein weiteres Datenbit ausgewählt und auf die Ausgangsleitung des Transmitters ausgegeben werden kann. Ähnliche Ergebnisse ergab die Analyse des VHDL-Quelltextes des KCUART-Receivers, allerdings mit dem Unterschied, dass dort zusätzlich ein 8-Bit-Shiftregister vorhanden ist.

B. UART mit einem mikroprogrammierten Steuerwerk

RAM-Blöcke in den Spartan/Virtex-Serien von Xilinx lassen sich als Dual-Port-Speicher konfigurieren [10]. Diese Eigenschaft ermöglicht es, mit einem einzelnen RAM-Block zwei völlig unabhängig voneinander arbeitende Steuerwerke zu realisieren.

Für diese Realisierung des MCUART-Moduls mit einem mikroprogrammierten Steuerwerk war es zunächst notwendig, alle Instanzen in den KCUART-Quelltextdateien zu identifizieren und anschließend zu entfernen, die eindeutig dem Receiver- bzw. dem Transmitter-Steuerwerk zugeordnet worden waren. In den so aufbereiteten Quelltextdateien wurde anstelle der beiden Steuerwerke ein gemeinsamer Dual-Port-RAM-Block mit Hilfe der primitiven Komponente eingesetzt.

In der Abbildung 6 ist das Blockschaltbild des mikroprogrammierten UART-Moduls zu sehen. Das Modul besteht im Sende- und Empfangspfad aus je einem 16-Byte-FIFO, aus dem Transmitter (TX) für die parallel/serielle Umsetzung sowie aus dem Receiver (RX) für die seriell/parallele Umsetzung. Beide Pfade werden durch ein gemeinsames Steuerwerk gesteuert. Das Steuerwerk, das in der Abbildung 7 zu sehen ist, ist auf der Grundlage eines RAM-Blocks in der Dual-Port-Konfiguration realisiert und kommt in dieser Anwendung ohne zusätzliche Logik aus.

Das gesamte Mikroprogramm besteht eigentlich aus zwei separaten Mikroprogrammen, die im RAM-Block in getrennten Adressbereichen untergebracht sind. Die Mikroprogramme für den Transmitter und für den Receiver belegten entsprechend die Adressbereiche 0x000 bis 0x1FF und 0x200 bis 0x3FF. Das Grundformat eines Mikrobefehls ist für beide Mikroprogramme weitgehend identisch und besteht aus einem Datenfeld mit Folgeadressen und einem Datenfeld mit Steuersignalen.

Der 3-Bit-Zähler, der Modulo-16-Zähler sowie die Zustände zur Generierung der Start-/Stoppbits aus dem ursprünglichen festverdrahteten Steuerwerk des Transmitters werden beim Übergang in eine mikroprogrammierte Steuerung in Folgen von Mikrobefehlen implementiert. Das Mikrobefehlsformat des Transmitters umfasst ein 8-Bit-Datenfeld für Folgeadressen und ein 7-Bit-Datenfeld für Steuersignale. Das Mikroprogramm des Transmitters belegt insgesamt 322 Speicherzellen. Das Mikrobefehlsformat des Receivers umfasst ein 8-Bit-Datenfeld für Folgeadressen und ein 2-Bit-Datenfeld für Steuersignale. Das Mikroprogramm des Receivers belegt insgesamt 308 Speicherzellen.

C. Gegenüberstellung

Der direkte Vergleich im Ressourcen-Verbrauch zwischen dem KCUART und dem MCUART nach der Synthese mit ISE 9.2i unter denselben Syntheseparametern ist in der Tabelle 4 gezeigt.

Anhand der Anzahl der belegten Slices sieht man, dass der MCUART mit dem mikroprogrammierten Steuerwerk um 34% weniger Ressourcen verbraucht als der KCUART mit dem festverdrahteten Steuer-

Tabelle 4: Detaillierte Gegenüberstellung im Ressourcen-Verbrauch zwischen dem KCUART mit festverdrahtetem Steuerwerk und dem MCUART mit mikroprogrammiertem Steuerwerk.

	KCUART	MCUART	FSM	MCU
# Slices	48	25	23	0
# FF	51	17	34	0
# LUT4	86	23	63	0
# RAMB	0	1	0	1
f_{max} [MHz]	145,6	145,6	–	–

werk. Die maximale Taktfrequenz ist in beiden Modulen unverändert geblieben und liegt bei über 145 MHz. Es war auch möglich, die Komplexität des festverdrahteten Steuerwerks aus dem KCUART separat zu ermitteln. Diese Werte sind in der FSM-Spalte zu sehen. Zum Vergleich sind die Daten des mikroprogrammierten Steuerwerks in der MCU-Spalte aufgelistet. Der Unterscheid ist hier deutlich zu erkennen: das Steuerwerk im MCUART verbraucht keine Slices, nur einen einzelnen RAM-Block.

V. Embedded-Mikrocontroller M68HC05

Das zweite Untersuchungsobjekt ist der Embedded-Mikrocontroller M68HC05 von Motorola, der in zwei Versionen implementiert wurde, und zwar mit einem festverdrahteten (M68HC05-FSM) und mit einem mikroprogrammierten Steuerwerk (M68HC05-MCU).

Der M68HC05 ist ein universeller 8-Bit-Mikrocontroller auf der Basis einer 8-Bit-CISC-CPU in einer 1-Adressarchitektur. Die CPU verfügt über einen 8-Bit-Akkumulator und ein 8-Bit-Index-Register, und unterstützt acht Adressierungsarten. Sie ist als eine von-Neumann-Architektur für Systeme mit gemeinsamem Daten-/Programmspeicher konzipiert. Der Befehlsvorrat basiert auf 62 Operationscodes, die in Kombinationen mit den Adressierungsarten insgesamt 210 Befehle ergeben. Einzelne Befehle zeichnen sich durch unterschiedliche Ausführungszeiten, und können von 2 bis 11 Maschinenzyklen betragen. Sie sind stark vom Operationscode und der Adressierungsart abhängig. Eine detaillierte und umfassende Beschreibung der Architektur und des Befehlssatzes dieses Mikrocontrollers findet man in z.B. [12] und [13]. Auf der Grundlage dieser Beschreibungen wurde der Mikrocontroller systematisch und modular in VHDL implementiert und anschließend in der Entwicklungsumgebung ISE 9.2i synthetisiert.

Das festverdrahtete Steuerwerk des M68HC05-Prozessors wurde als Zustandsmaschine mit einem „synchronen“ Prozess mit der in VHDL typischen CASE-Anweisung implementiert. Diese Zustandsmaschine basiert auf einem Mealy-Automaten mit Ausgangsflipflops und umfasst insgesamt 358 Zustände.

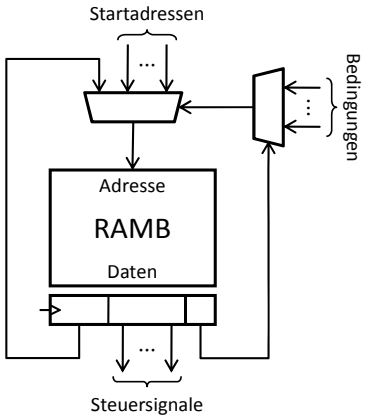


Abbildung 8: Prinzipieller Aufbau von Steuerwerken auf der Grundlage synchroner RAM-Blöcke mit bedingten Startadressen.

Die beiden in der Abbildung 2 dargestellten Steuerwerke können zwar Folgen bedingter und unbedingter Mikrobefehle abarbeiten, sind aber als Steuerwerke in einem Prozessor weniger geeignet. Die Struktur der Steuerwerke muss derart erweitert werden, dass sie auch anhand eines Operationscodes den richtigen Abschnitt im Mikroprogrammcode für den passenden Befehl finden, und somit direkte Sprünge unterstützen. Dies ist relativ leicht mit zusätzlichen Multiplexern realisierbar und in der Abbildung 8 dargestellt.

Das mikroprogrammierte Steuerwerk von M68HC05 basiert auf einem einzelnen RAM-Block in der Single-Port-Konfiguration mit 512 Speicherzellen sowie 36-Bit-Mikrobefehlen. Ein Mikrobefehl enthält mehrere Felder, und zwar ein 9-Bit-Datenfeld für Folgeadresse, ein 2-Bit-Datenfeld für lokale Select-Signale für Multiplexer im Steuerwerk, sowie ein 25-Bit-Datenfeld mit Steuersignalen. Die Steuersignale in diesem Datenfeld sind in weitere Signalgruppen unterteilt: acht Write-Enable-Signale für Register, vier Enable-Signale für den Daten/Programmspeicher, drei Auswahlsignale für Adressierungsarten, fünf Funktionssignale für den Befehlszähler PC und den Stack-Pointer SP sowie fünf Select-Signale für Multiplexer im Rechenwerk.

Das Mikroprogramm im Steuerwerk kann entweder den Befehlszyklus ausführen oder einen Interruptzyklus einleiten. Da der Operationscode von M68HC05 einheitlich für alle Befehle in einem Byte enthalten ist, war es relativ einfach, aus dem Operationscode die Startadresse des dazugehörigen Mikroprogramms zu ermitteln. Der Operationscode wird dabei direkt als Adresse verwendet, und zwar ohne zusätzliches Adress-Mapping. Das gesamte Mikroprogramm des M68HC05 belegt 358 Speicherzellen was einer 70%-iger Ausnutzung des RAM-Blocks entspricht.

Die Ergebnisse aus der Synthese sind in der Tabelle 5 zusammengefasst. Der vollständige Prozessor M68HC 05-FSM mit dem festverdrahteten Steuerwerk belegt 399 Slices. Das sind rund 20% der Ressourcen

Tabelle 5: Detaillierte Gegenüberstellung im Ressourcen-Verbrauch zwischen der festverdrahteten (FSM) und der mikroprogrammierten (MCU) Version des Steuerwerks.

	M68HC05-			
	FSM	MCU	FSM	MCU
# Slices	399	170	220	6
# FF	79	44	36	0
# LUT4	716	325	402	10
# RAMB	0	1	0	1
f_{max} [MHz]	63,6	66,2	–	–

im FPGA-Baustein vom Typ XC3S200. Der Prozessor M68HC05-MCU mit dem mikroprogrammierten Steuerwerk belegt lediglich 170 Slices, und das sind knapp 8% der Ressourcen im FPGA-Baustein desselben Typs. Beiden Steuerwerke wurden auch separat synthetisiert und die Ergebnisse sind in der dritten und vierten Spalte zu sehen. Das mikroprogrammierten Steuerwerk (MCU) besteht aus einem RAM-Block und zusätzlichen Multiplexern, und belegt insgesamt nur 6 Slices. Das festverdrahtete Steuerwerk (FSM) belegt 220 Slices. In diesen Experimenten wurde ein hardwareoptimiertes Rechenwerk eingesetzt, das teilweise mit primitiven Komponenten aufgebaut war.

VI. Fazit

Der folgende Beitrag hat in systematischen Schritten gezeigt, wie man RAM-Blöcke in FPGA-Bausteinen als mikroprogrammierte Steuerwerke einsetzen kann. An beiden Anwendungsbeispielen konnte auch gezeigt werden, dass solche Steuerwerke zu einer bedeutsamen Einsparrung von Ressourcen (Slices) in FPGAs führen.

Der Programmieraufwand bei der Erstellung eines Mikroprogramms ist mit dem während der Implementierung einer Zustandsmaschine mit einem CASE-Statement durchaus vergleichbar. Allerdings muss man bei der Realisierung eines mikroprogrammierten Steuerwerks auf die physikalischen Kapazitäten und die Konfigurationsmöglichkeiten des RAM-Blocks achten. Übersteigt die Länge der Steuervektoren die Wortbreite des RAM-Blocks, so müssen ggf. mehrere RAM-Blöcke parallel zusammengeschaltet werden.

Ein mikroprogrammiertes Steuerwerk auf der Basis eines RAM-Blocks hat eine feste Größe, was eine genauere Abschätzung der Schaltungskomplexität bereits in der Entwurfsphase ermöglicht. Änderungen und Erweiterungen im Mikroprogramm haben keinen Einfluss auf den Ressourcenverbrauch einer Schaltung, solange physikalische Grenzen eines RAM-Blocks nicht überschritten werden. Dagegen haben Änderungen in der Codierung der Signale oder Erweiterungen in einem festverdrahteten Steuerwerk unmittelbare Auswirkungen auf die Komplexität der Übergangs- und Ausgangsschaltnetze und somit auch auf den Ressourcenverbrauch.

Dieser Ansatz stellt eine interessante und vollwertige Alternative zu der herkömmlichen Realisierung von festverdrahteten Steuerwerken dar.

Literaturverzeichnis

[1] P. Alfke, „Creative Uses of Block RAM“, White Paper: Virtex and Spartan FPGA Families, *Xilinx, Inc*., San Jose, Juni 2008.

[2] K. Chapman, „UART Transmitter and Receiver Macros“, UART Manual, *Xilinx, Inc*., San Jose, Jan. 2003.

[3] K. Chapman, „200 MHz UART with Internal 16-Byte Buffer“, App. Note 223, *Xilinx, Inc.*, San Jose, April 2008.

[4] J. D. Carpinelli, „Computer Systems Organisation & Architecture“, *Addison Wesley Longman*, Boston, 2001.

[5] M. A. Lynch, „Microprogrammed State Machine Design“, *CRC Press*, Boca Raton, 1993.

[6] D. E. White, „Bit-Slice Design: Controllers an ALUs“, *Garland Publishing*, New York, 1981.

[7] Xilinx, „Product Selection Guides“, *Xilinx, Inc.*, San Jose, Mai 2010.

[8] Xilinx, „Spartan-3 Generation FPGA, User Guide“, *Xilinx, Inc.*, San Jose, Feb. 2008.

[9] Xilinx, „Spartan-3 FPGA Family, Data Sheet“, *Xilinx, Inc*., San Jose, Juni 2008.

[10] Xilinx, „Using Block RAM in Spartan-3 Generation FPGAs“, App. Note 463, *Xilinx, Inc*., San Jose, März 2005.

[11] Xilinx, „Spartan-3 Generation FPGA User Guide“, *Xilinx, Inc*., San Jose, Feb. 2008.

[12] Motorola, „M68HC05 Family - Understanding Small Microcontrollers“, Rev. 2.0, *Motorola, Inc*., Denver, 1998.

[13] Motorola: „M68HC05 Microcontroller Applications Guide“, Rev. 3.0, *Motorola, Inc*., Phoenix, 1998.

Christian Kielmann studierte Technische Informatik an der HTWG Konstanz und erhielt dort im Jahre 2010 den akademischen Grad B. Sc. Im Rahmen einer Bachelor-Arbeit befasste er sich mit Soft-Core-Prozessoren. Neben dem Studium arbeitete er in der Spezialmesstechnik beim Triebwerkshersteller Rolls-Royce Deutschland.

Dominik Stengele studiert Technische Informatik an der HTWG Konstanz im 7. Semester und befasste sich im Rahmen eines TIB-Projektes mit mikroprogrammierten und festverdrahteten Steuerwerken in FPGAs. Neben dem Studium arbeitet er als Werkstudent in der ASIC-Entwicklung bei Bosch Automotive Electronics.

Irenäus Schoppa studierte Informatik an der Technischen Universität Berlin und erhielt dort im Jahre 1993 den akademischen Grad Dipl.-Informatiker. Im Jahre 1998 promovierte er dort zum Dr.-Ing.. Seit dem Jahr 2008 ist er Professor für Hardware-Software Codesign an der HTWG Konstanz.

HOCHSCHULE KONSTANZ
TECHNIK, WIRTSCHAFT UND GESTALTUNG

Ressourcen-Optimierung durch Einsatz primitiver FPGA-Komponenten

Albert Schaaf, Bastian Teppert, Tobias Tornar, Irenäus Schoppa

***Zusammenfassung*—Die Komplexität in der Synthese digitaler Systeme lässt sich heute nur mit einem modularen und hierarchischen Entwurf beherrschen, wobei die unterste Hierarchiestufe meistens die Registertransferebene ist. Einzelne Module werden dann mit Hilfe einer Hardwarebeschreibungssprache z.B. VHDL durch Registertransferoperationen verhaltensorientiert beschrieben. Ist der Beschreibungsstil solcher Module auch synthesegerecht, so lassen sich solche Systeme mit Hilfe passender Synthese-Software für eine gewählte FPGA-Technologie automatisch synthetisieren. Der folgende Beitrag befasst sich mit Analyse- und Synthesemaßnahmen, die bei der Entwicklung einer komplexen Schaltung und deren Beschreibung in VHDL berücksichtigt werden müssen, damit der Verbrauch an Ressourcen (Logikzellen) in einem FPGA-Baustein möglichst minimal ist. Der Weg zu diesem Ziel führt über den Einsatz primitiver Komponenten einer FPGA-Architektur.**

***Schlüsselwörter*—FPGA, Spartan-3, primitive Komponenten, Soft-Core-Prozessor, M68HC05.**

I. Einleitung

Im Rahmen eines Teamprojektes an der HTWG Konstanz wollten wir Antworten auf folgende Fragen finden:

1) Wie viele Ressourcen (vor allem Logikzellen/ Slices) können in einem FPGA-Baustein eingespart werden, wenn Teile eines komplexen Schaltwerks mit Hilfe primitiver FPGA-Komponenten beschrieben sind?
2) Durch welche Optimierungsmaßnahmen kann man die Einsparung erreichen und wie aufwändig ist das?

Als Beispiel eines komplexen Schaltwerks wurde die CPU des 8-Bit-Mikrocontrollers M68HC05 gewählt. Diese CPU stellt das Untersuchungsobjekt dar, und wurde zu diesem Zweck als Soft-Core-Prozessor in VHDL implementiert. Als Referenzobjekt wurde der PicoBlaze von Xilinx gewählt, denn dieser Soft-Core-Prozessor ist vollständig mit primitiven FPGA-Komponenten realisiert.

A. Schaaf, alschaaf@htwg-konstanz.de, B. Teppert, batepper@htwg-konstanz.de und T. Tornar, totornar@htwg-konstanz.de, sind Studenten an der HTWG Konstanz, I. Schoppa, ischoppa@htwg-konstanz.de, ist Mitglied der HTWG Konstanz, Brauneggerstr. 55, 78462 Konstanz.

II. Soft-Core-Prozessor PicoBlaze

Der PicoBlaze alias KCPSM3 (Programmable State Machine) ist ein einfacher Soft-Core-Prozessor mit einem 8-Bit-RISC-Kern auf der Basis einer 2-Adreß-architektur. Er hat einen orthogonalen Registersatz mit 16 8-Bit-Registern und einen reduzierten Befehlssatz mit insgesamt 57 Befehlen. Alle Befehle werden einheitlich in zwei Takten ausgeführt. Außerdem unterstützt er sechs Adressierungsarten. Seine Architektur ist eine typische Load-/Store-Architektur mit getrennten Daten- und Programmspeichern. Sein Rechenwerk und Steuerwerk sind auf Laufzeit und Ressourcen-Verbrauch hin optimiert und wurden vollständig mit primitiven FPGA-Komponenten implementiert. Nach der Synthese belegt der PicoBlaze 96 Slices in einem FPGA-Baustein vom Typ Spartan-3 und erreicht eine maximale Taktfrequenz von 120,4 MHz. Weitere Informationen sind in [4] und [5] zu finden.

III. Soft-Core-Prozessor M68HC05

Der M68HC05 ist ein Universalprozessor von Motorola/Freescale mit einem 8-Bit-CISC-Kern auf der Basis einer 1-Adreßarchitektur. Er hat einen 8-Bit-Akkumulator und ein 8-Bit-Indexregister. Er zeichnet sich durch acht Adressierungsarten sowie durch einen umfangreichen Befehlssatz mit insgesamt 210 Befehlen aus. Seine Befehle weisen unterschiedliche Ausführungszeiten auf. Seine Architektur ist eine typische Von-Neumann-Architektur mit gemeinsamem Daten- und Programmspeicher. Eine umfassende Beschreibung des Prozessors findet man in [6] und [7].

Im Rahmen des Teamprojektes wurde der M68HC05 auf der Registertransferebene als Soft-Core-Prozessor in VHDL beschrieben und synthetisiert. In der Abbildung 1 ist der Ausschnitt aus seinem Rechenwerk als Registertransferstruktur dargestellt. Neben dem Rechenwerk gibt es in der Implementierung des Soft-Core-Prozessors eine weitere, komplexe Komponente, nämlich die Adressierungseinheit, in der die effektive Adresse berechnet wird.

Diese Struktur des Rechenwerks ist durch die Analyse des Befehlssatzes, der Architektur und der daraus

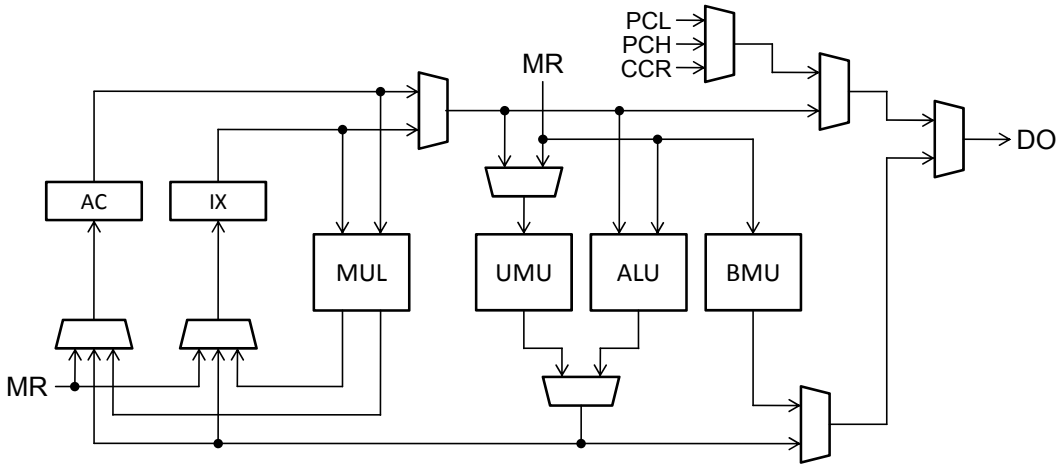


Abbildung 1: Struktur des Rechenwerks von M68HC05 auf der Registertransferebene.

resultierenden Registertransferoperationen entwickelt worden. Der gesamte Soft-Core-Prozessor wurde modular aufgebaut, so dass einzelne Module separat getestet werden konnten. In der Abbildung 1 sind vier Funktionseinheiten zu sehen: MUL ist ein 8x8-Bit-Multiplizierer, realisiert mit der primitiven FPGA-Komponente MULT18X18S, UMU ist eine universelle Modifizierungseinheit, ALU ist die arithmetisch-logische Einheit und BMU ist die Bit-Manipulationseinheit. Diese Funktionseinheiten sind für die Ausführung bestimmter Befehlsklassen zuständig. Da im FPGA-Baustein keine internen Busstrukturen vorhanden sind, wurden alle Verbindungswege zwischen Registern und Funktionseinheiten mit mehreren Multiplexern realisiert. Außerdem sieht man hier noch die beiden 8-Bit-Register des Prozessors: den Akkumulator AC und das Indexregister IX.

In der ersten Version wurde der Soft-Core-Prozessor M68HC05 hauptsächlich technologieunabhängig mit den Anweisungen der Hardwarebeschreibungssprache VHDL beschrieben. Die einzige primitive Komponente im Rechenwerk war der Multiplizierer. Alle Register und einzelne Speicherelemente, wie z.B. das Carry-Flag im Statusregister CCR, sind mit „synchronen“ Prozessen modelliert worden. Multiplexer wurden einheitlich mit selektierten Signalzuweisungen beschrieben. Für arithmetische Operationen, also Additionen und Subtraktionen, wurden Plus- und Minus-Operatoren aus der IEEE-Bibliothek eingesetzt. Das Steuerwerk wurde parallel zum Rechenwerk entwickelt, und in der endgültigen Version als mikroprogrammiertes Steuerwerk mit einem RAM-Block realisiert. Nach der Synthese belegt der Soft-Core-Prozessor M68HC05 in dieser Version insgesamt 211 Slices und erreicht eine maximale Taktfrequenz von 52 MHz. Diese Version stellte den Ausgangspunkt für weitere Ressourcen-Optimierungen durch den Einsatz primitiver FPGA-Komponenten dar.

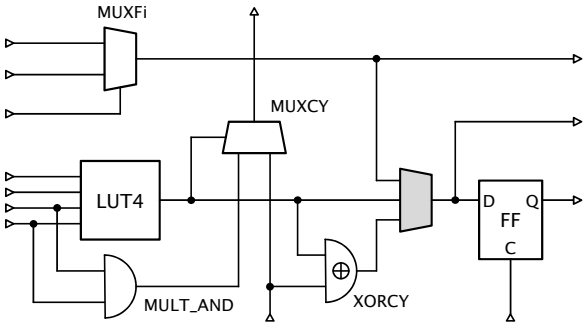


Abbildung 2: Vereinfachter Aufbau der Logikzelle in einem FPGA-Baustein vom Typ Spartan-3.

IV. Primitive FPGA-Komponenten

Um eine Ressourcen-Optimierung mit Hilfe primitiver FPGA-Komponenten durchführen zu können, ist es hilfreich, zunächst den Aufbau der Logikblöcke des verwendeten FPGA-Bausteines detaillierter zu betrachten. Eine umfassende Beschreibung der primitiven FPGA-Komponenten findet man in [1] und [2].

Jeder Logikblock (CLB) der Spartan-3-Familie besteht aus zwei Slice-Paaren (SLICEL und SLICEM). Beide Slices beinhalten jeweils zwei 4-Bit-Look-Up-Tabellen (LUT4), die als universelle Funktionsgeneratoren dienen, sowie zwei Speicherelemente (FDRSE), die als synchrone Flipflops oder Latches konfiguriert werden können. In der Abbildung 2 ist der vereinfachte Aufbau einer konfigurierbaren Logikzelle im Spartan-3 dargestellt. Desweiteren verfügt jede Logikzelle über zwei Modulo-2-Addierer (XORCY), zwei schnelle Carry-Multiplexer (MUXCY) und zwei universelle 1-aus-2-Multiplexer. Letztere unterscheiden ich in MUXF5 und MUXFi (mit i = 6, 7, 8). Die Eingänge des MUXF5 sind dabei direkt mit den Ausgängen der beiden Look-Up-Tabellen verbunden. Die SLICEM-Logikzellen enthalten zusätzliche Funktionalität, und zwar die Look-Up-Tabellen können als 16-Bit-Schieberegister (SRL16)

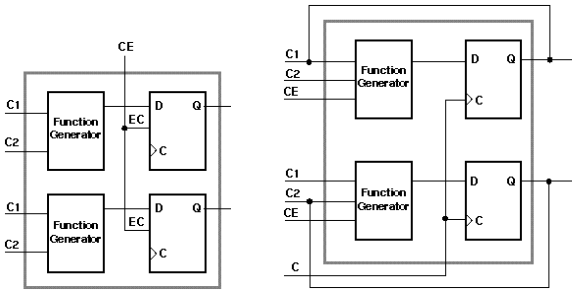


Abbildung 3: Ansteuerung von Flipflops: direkt über Clock-Enable-Signale und indirekt über Look-Up-Tabellen aus [3].

bzw. als kleiner RAM-Block (RAM16) verwendet werden. Jede primitive Komponente lässt sich mit Hilfe der UNISIM-Bibliothek im VHDL-Quelltext instanziieren [3].

V. Optimierungsmassnahmen

A. Optimierung von Registern

Speicherelement in FPGA-Bausteinen von Xilinx basieren auf universellen D-Flipflops mit verschiedenen Konfigurationsmöglichkeiten wie z.B. das Clock-Enable-Signal oder asynchrone/synchrone Set/Reset-Signale. In unseren Experimenten wurde festgestellt, dass Clock-Enable-Signale während der Synthese nicht immer benutzt werden.

In der Literatur sind zwei Methoden beschrieben, wie Speicherelemente (D-Flipflops) angesteuert werden können. Es handelt sich dabei um die direkte Ansteuerung mit einem Clock-Enable-Signal oder um die indirekte Ansteuerung über eine Look-Up-Tabelle. Beide Methoden sind schematisch in der Abbildung 3 zu sehen. Bei der indirekten Ansteuerung werden die Ausgänge von Flipflops auf die Dateneingänge über Look-Up-Tabellen rückgekoppelt. Hier werden die Flipflops auch mit jedem Takt getaktet. Deshalb muss mit Hilfe der Look-Up-Tabelle eine Funktion realisiert werden, die in der Abhängigkeit von einem Steuersignal (z.B. Enable oder Select) entweder ein neues Datum oder das Datum vom Ausgang des Flipflops durchschaltet. Diese Funktion entspricht dem Verhalten eines Multiplexers. Diese Art der Beschaltung ist aber nicht in jeder Schaltung erwünscht, denn sie verbraucht Logik, zwar nicht viel, aber bei vielen Flipflops lassen sich einige Look-Up-Tabellen einsparen.

Wenn die Datenübernahme nur sporadisch, also nicht in jedem Takt, erfolgen soll, dann lohnt es sich, die Clock-Enable-Funktion in der VHDL-Beschreibung explizit zu modellieren. Auf diese Weise werden Rückkopplungen und somit auch Multiplexer eliminiert. Dadurch lassen sich einige Ressourcen einsparen.

```
PROCESS (clk) BEGIN
  IF rising_edge(clk) THEN
     CASE code IS
       WHEN "01"   => reg <= din;
       WHEN "10"   => reg <= reg + 1;
       WHEN "11"   => reg <= reg - 1;
       WHEN OTHERS => NULL;
     END CASE;
   END IF;
END PROCESS;



CE <= code(1) OR code(2);
u1: FOR i IN 0 TO N-1 GENERATE
     ff: FDE
         PORT MAP(Q  => reg(i),
                  C  => clk,
                  CE => CE,
                  D  => D(i));
     END GENERATE;

WITH code SELECT
  d <= reg + 1 WHEN "10",
       reg - 1 WHEN "11",
       din     WHEN OTHERS;
```

Abbildung 4: VHDL-Beschreibung eines Up-/Down-Counters mit synchronem Prozess und unten mit dem primitiven FDE-Flipflop.

Im obigen Teil der Abbildung 4 ist eine typische VHDL-Beschreibung eines Up-/Down-Counters mit einer Load-Funktion zu sehen. So ein Counter kann z.B. als Stack-Pointer eingesetzt werden, was beim Soft-Core-Prozessor M68HC05 auch der Fall war. Weil die Anzahl der Steuersignale im mikroprogrammierten Steuerwerk begrenzt war (nur 25 standen zur Verfügung), war es notwendig, diesen Counter mit einem Funktion-Code anzusteuern. Das ist auch im obigen Programmausschnitt zu sehen. In der Abhängigkeit vom 2-Bit-Funktions-Code werden hier drei unterschiedliche Aktionen selektiert: Laden, Inkrementieren und Dekrementieren. Der vierte Fall bildet die Rückkopplung in der Schaltung. Diese Beschreibung hat für einen 8-Bit-Counter 10 Slices belegt, wobei die Synthese hier Flipflops vom Typ FDC instanziiert hat. Die FDC-Flipflops sind Flipflops ohne Clock-Enable-Eingang.

Im unteren Teil der Abbildung 4 ist eine modifizierte VHDL-Beschreibung zu sehen. Hier wird das Clock-Enable-Signal (CE) explizit mit einem ODER-Ausdruck generiert und weiter in den instanziierten primitiven Flipflops vom Typ FDE mit Clock-Enable-Eingängen verwendet. Außerdem wurde das ursprüngliche CASE-Statement aus dem obigen Prozess durch eine selektierte Signalzuweisung ersetzt. Diese Modifikationen haben zur Einsparung von drei LUTs und sechs MUXF5 geführt. Die so modellierte Komponente hat nur 9 Slices belegt. Diese Ergebnisse konnte auch mit Technology-Viewer verifiziert werden.

```
WITH code SELECT
res <= lval - rval       WHEN SUB,
       lval - rval - cin WHEN SBC,
       lval + rval + cin WHEN ADC,
       lval + rval       WHEN ADD;


u1: ADSU
      GENERIC MAP(N => 8)
      PORT MAP(sub  => code(1),
               sel  => code(0),
               cin  => cin,
               cout => cout,
               hout => hout,
               op1  => lval,
               op2  => rval,
               res  => res);
```

Abbildung 5: Einsatz von ADSU-Komponenten statt Plus-/Minus-Operatoren aus der IEEE-Bibliothek.

B. Optimierung arithmetischer Operationen

In unseren Experimenten wurde festgestellt, dass der Einsatz der Plus- und Minus-Operatoren aus der IEEE-Bibliothek nicht immer optimale Ergebnisse hervorbringt. Vor allem solche Schaltnetze, wie im oberen Teil der Abbildung 5 durch die selektierte Signalzuweisung beschrieben sind, erreichen nicht immer den minimalen Ressourcen-Verbrauch. Solche Konstruktionen sind typisch bei arithmetisch-logischen Funktionseinheiten. Die Synthese dieser VHDL-Beschreibung für 8-Bit-Operanden hat in der Regel doppel so viele Slices belegt, wie die Realisierung mit einer ADSU-Komponente.

Auf der Grundlage eines kaskadierbaren und modularen 1-Bit-Addierers/Subtrahierers wurde eine generische ADSU-Komponente in der Form einer Schaltkette entwickelt. Die ADSU-Komponente ist keine primitive Komponente in der Spartan-3-Serie. Man findet sie aber z.B. in der CoolRunner-Serie. Bei der Implementierung der ADSU-Komponente wurde die schnelle Carry-Kette in der Logikzelle verwendet, und zwar durch die Instanziierung primitiver Komponenten MUXCY und XORCY. Die effiziente Nutzung der Carry-Kette und der Arithmetik-Logik ist im User-Guide von Xilinx [1] detailliert beschrieben.

Es gab noch einen weiteren Grund, warum arithmetische Operationen mit der ADSU-Komponente realisiert worden sind. Der M68HC05 hat in seinem Statusregister das sog. Half-Carry-Flag. Das ist ein Bit, mit dem der Wert beim Übertrag von der 3-ten auf die 4-te Stelle im Addierer angezeigt wird. Diese Funktionalität lässt sich besonders einfach mit der ADSU-Komponente realisieren. Man kann dieses Bit auch durch einem zusammengesetzten logischen Ausdruck berechnen lassen, das ist aber etwas umständlicher.

Tabelle 1: Detaillierte Gegenüberstellung im Ressourcen-Verbrauch zwischen dem PicoBlaze und den beiden Versionen des Soft-Core-Prozessors M68HC05.

		M68HC05-	
	PicoBlaze	Standard	Unisim
# Slices	96	211	170
# LUT	181	410	325
# FF	76	44	44
# RAMB	0	1	1
# MULT	0	1	1
f_{max} [MHz]	120,4	52,1	66,2

C. Optimierung von Multiplexern

Eine weitere Optimierungsmaßnahme war der Einsatz von MUXF5, 6, 7 und 8. Allerdings ist deren Einsatz eine höchst sensible Angelegenheit. Hier muss man erst einen gewissen Erfahrungsschatz aufbauen. Denn sonst führt der Einsatz solcher Komponenten nicht zum gewünschten Erfolg.

VI. SYNTHESERESULTATE

In der Tabelle 1 sind die Syntheseergebnisse zusammengefasst. Der PicoBlaze dient dabei als Referenz-Modell. Der M68HC05 wurde mit einem mikroprogrammierten Steuerwerk synthetisiert, denn das Steuerwerk hat eine feste Größe, und so lässt die Komplexität des Rechenwerks leicht bestimmen. Das Rechenwerk wurde in zwei Versionen modelliert und auch synthetisiert. Der Ausgangspunkt war eine Version mit der „reinen“ VHDL-Beschreibung, die in der Anlehnung an die Benutzung der IEEE-Bibliothek mit der Std-Logic auch Standard genannt ist. In dieser Beschreibung belegt der Prozessor insgesamt (also mit dem mikroprogrammierten Steuerwerk) 211 Slices. Wenn man davon die 6 Slices für das Steuerwerk abzieht, kommen wir auf 205 Slices für das Rechenwerk. In dieser Version erreicht der Soft-Core-Prozessor eine maximale Taktfrequenz von 52 MHz.

Nachdem einzelne VHDL-Module systematisch analysiert und anschließend in mehreren Schritten durch geeignete Schaltungen, bestehend aus primitiven FPGA-Komponenten, ersetzt wurden, haben wir eine Version des Soft-Core-Prozessors M68HC05 bekommen, die hier in Anlehnung an die Benutzung der Unisim-Bibliothek auch Unisim genannt ist. In dieser Version belegt der Soft-Core-Prozessor insgesamt 170 Slices. Auch hier kann man nach dem Abzug der Slices für das mikroprogrammierte Steuerwerk den Ressourcen-Verbrauch fürs Rechenwerk bestimmen. Das Rechenwerk belegt somit 164 Slices. Durch den Einsatz der primitiven FPGA-Komponenten haben sich auch die Timing-Eigenschaften des Prozessors verbessert. Nach den Optimierungen kann er mit bis zu 66 MHz getaktet werden.

VII. FAZIT

Durch den Einsatz primitiver FPGA-Komponenten ist es uns gelungen, den Ressourcen-Verbrauch bei der Implementierung des Soft-Core-Prozessors M68HC05 um fast 20% zu reduzieren. Allerdings haben sich solche Optimierungen als sehr zeitaufwendig erwiesen. In der Anfangsphase wurden sie nach dem Trial-And-Error-Prinzip durchgeführt. Erst im Laufe der Zeit haben wir neue Erkenntnisse gewonnen und Erfahrungen gesammelt, so dass wir später leichter und schneller entscheiden konnten, welche Teile der VHDL-Beschreibung optimiert werden können, und welche nicht. Ab und zu sind wir in einer Sackgasse geraten, denn manche Optimierungen haben sich als kontraproduktiv erwiesen, und der Ressourcen-Verbrauch wurde größer. Vor allem der Einsatz von Multiplexern MUXFi ist problematisch und sehr sensibel.

Diese Ressourcen-Optimierung durch den Einsatz primitiver FPGA-Komponenten lohnt sich zwar, ist aber zeitaufwändig und erfordert viel Erfahrung. Deutlich bessere Resultate lassen sich erzielen, wenn nur das Steuerwerk optimiert wird, d.h. das Steuerwerk nicht in der festverdrahteten Version sondern in der mikroprogrammierten Version implementiert wird.

LITERATURVERZEICHNIS

[1] Xilinx, „Spartan-3 Generation FPGA, User Guide“, *Xilinx, Inc.*, San Jose, Feb. 2008.

[2] Xilinx, „Spartan-3 FPGA Family, Data Sheet“, *Xilinx, Inc*., San Jose, Juni 2008.

[3] Xilinx, „Spartan-3 Libraries Guide for HDL Designs“, ISE 10.1, *Xilinx, Inc*., July 2008.

[4] K. Chapmen, „KCPMS3 8-bit Micro Controller for Spartan-3, Virtex-II and Virtex-II PRO“, *Xilinx, Ltd*., Rev. 7, Oct. 2005.

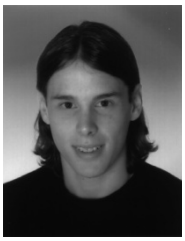
[5] Xilinx, „PicoBlaze 8-bit Embedded Microcontroller User Guide for Spartan-3, Spartan-6, Virtex-5, and Virtex-6 FPGAs“, User’s Guide 129, Rev. 2.0, *Xilinx, Inc*., Jan. 2010.

[6] Motorola, „M68HC05 Family - Understanding Small Microcontrollers“, Rev. 2.0, *Motorola, Inc*., Denver, 1998.

[7] Motorola: „M68HC05 Microcontroller Applications Guide“, Rev. 3.0, *Motorola, Inc*., Phoenix, 1998.

Bastian Teppert studiert Technische Informatik an der HTWG Konstanz im 6. Semester und befasste sich im Rahmen eines TIB-Projektes mit der Optimierung von VHDL-Quelltext am Beispiel eines Soft-Core-Prozessors.

Albert Schaaf studiert Technische Informatik an der HTWG Konstanz im 6. Semester und befasste sich im Rahmen eines TIB-Projektes mit der Optimierung eines M68HC05 unter der Anwendung von primitiven Komponenten.



Tobias Tornar studiert Technische Informatik an der HTWG Konstanz im 7. Semester und befasste sich im Rahmen eines TIB-Projektes mit der Verifizierung des implementierten Soft-Core-Prozessors M68HC05 durch eine Testbench.

Irenäus Schoppa studierte Informatik an der Technischen Universität Berlin und erhielt dort im Jahre 1993 den akademischen Grad Dipl.-Informatiker. Im Jahre 1998 promovierte er dort zum Dr.-Ing.. Seit dem Jahr 2008 ist er Professor für Hardware-Software Codesign an der HTWG Konstanz.

HOCHSCHULE KONSTANZ
TECHNIK, WIRTSCHAFT UND GESTALTUNG

Technologie und Realisierung von Hardwarebeschleunigung in einem Altera Softcore

Christian Siebert, Christian Seibt, Gregor Burmberger

***Zusammenfassung*—Dieses Paper erläutert den technischen Stand der aktuellen Methoden zur Hardwarebeschleunigung der Firma Altera. Dazu werden die unterschiedlichen Verfahren zur Realisierung von Hardwarebeschleunigungen aufgezeigt. Die Geschwindigkeitsverbesserung der Verfahren werden anhand von praktischen Beispielen veranschaulicht. Hierzu werden Tools von Altera verwendet.**

***Schlüsselwörter*—VHDL, FPGA, hardware acceleration, custom instruction, C2H.**

I. Einleitung

In vielen Bereichen, insbesondere bei Audio-, Video- und Netzwerkanwendungen, stoßen Standard-Mikrocontroller oft an ihre Grenzen. Die gewünschten Anforderungen an Kosten, Geschwindigkeit und Energieverbrauch werden nicht erreicht. Eine Möglichkeit, um diese Anforderungen in FPGA-basierten Systemen dennoch zu erfüllen, ist der Einsatz sog. *Hardwarebeschleuniger*. Hierbei werden rechenaufwendige Routinen von der Software in die Hardware verlagert.

Eine Art dieser Beschleunigungstechnologien sind *Custom Instructions*. Hierbei werden, wie auch bei der klassischen Hardwarebeschleunigung durch IP-Blöcke, kritische Teile des Codes in Hardware ausgelagert. Die Befehle werden dabei direkt an die ALU des Softcore Prozessors angebunden. Auf diese Weise werden sehr hohe Geschwindigkeiten für einzelne Befehle erreicht. Mit den heutigen Softwarewerkzeugen ist die Implementierung von Custom Instructions relativ einfach möglich.

Christian Siebert, csiebert@htwg-konstanz.de,
Christian Seibt, chseibt@htwg-konstanz.de,
Gregor Burmberger, gregor.burmberger@htwg-konstanz.de,
HTWG Konstanz, Fakultät EI, Brauneggerstr. 55, 78462 Konstanz

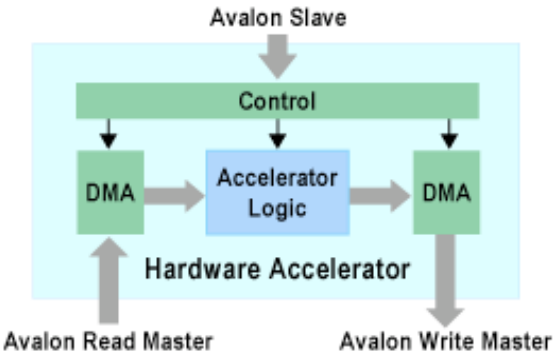


Abbildung 1: Typische Realisierung der Hardwarebeschleunigung

II. Unterschied zwischen Hardwarebeschleunigung und Custom Instructions

In zeitkritischen Anwendungen kann die Ausführung von Code in Software zu viel Zeit in Anspruch nehmen. Die einfachste Lösung, die Erhöhung der Systemfrequenz, ist immer seltener möglich. Die maximale Taktfrequenz wird im Allgemeinen durch Layout-Eigenschaften wie z.B. maximale Leitungslängen und Abstände der Leitungen zueinander oder durch Vorgaben bezüglich der Versorgungsspannung ($f \sim U^2$) bzw. der maximalen Leistungsaufnahme ($f \sim E$) begrenzt.

Um höhere Geschwindigkeiten zu erreichen, werden rechenintensive Anwendungen in Hardware realisiert. Ein Beispiel hierfür ist die FPU (*Floating Point Unit*). Durch Nutzung einer FPU werden alle Fließkomma-Rechenoperationen wie z. B. Multiplikation, Division, Quadratwurzel etc. vollständig in Hardware realisiert. Die Bearbeitungsgeschwindigkeit entsprechender Programmroutinen steigt drastisch.

Im gleichen Sinne von *Hardwarebeschleunigung*, spricht man bei der Implementierung rechenintensiver Routinen in Hardware (IP-Blöcke) auf einem FPGA. Der Softcore Prozessor gibt über einen Steuerbus Befehle an die Logik auf dem FPGA weiter. Daten werden hierbei über einen DMA-Controller gelesen bzw. geschrieben. Abbildung 1 zeigt die typische Umsetzung einer Hardwarebeschleunigung (hier abstrakt *Accelerator Logic*) auf einem FPGA [1].

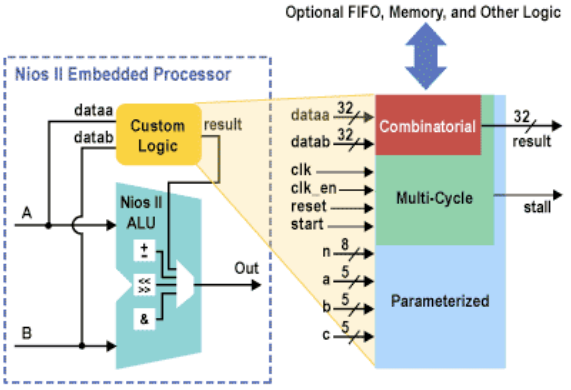


Abbildung 2: Anbindung eines Custom Logic Blocks

Table 5. Custom Instruction vs. Software-Only Performance Comparison (1)

Floating-Point Operation	CPU Clock Cycles		Speed Increase
	Software Library	Custom Instruction (FPU)	
Multiplication a × b	2,874	19	151.26
Multiply and Negate -(a × b)	3,147	19	165.63
Absolute |a|	1,769	18	98.28
Negate -(a)	284	19	14.95

Abbildung 3: Vergleich der Beschleunigung durch Custom Instructions

Wie Abbildung 3 zeigt, sind die Leistungsberechnungen abhängig vom verwendeten Compiler. Hier wurde der Cygnus Compiler verwendet, welcher in Version 2.1 des *NIOS II Embedded Processor* enthalten ist.

Custom Instructions sind in erster Linie, genau wie bei der Hardwarebeschleunigung mit IP-Blöcken, in Hardware ausgelagerte Befehle. Allerdings werden Custom Instructions direkt in die ALU (*Arithmetic Logic Unit*) eines Softcore Prozessors integriert. Der Befehlssatz der CPU wird hierbei um die erstellten Custom Instructions erweitert [2]. So kann rechenaufwändiger Code, wie beispielsweise die Bearbeitung von Paket-Headern eines Übertragungsprotokolls, auf einen einzelnen Befehl reduziert werden.

Abbildung 2 zeigt die Anbindung eines Custom (Instruction) Logic Block an die ALU eines *Altera NIOS II* Prozessors [1]. Die Leitungen zur ALU dienen dem Custom Logic Block als Eingang (Input). Das Ergebnis wird über eine interne Logik der ALU ausgewählt.

Abbildung 3 zeigt die Gegenüberstellung der benötigten CPU-Takte zur Ausführung verschiedener Funktionen, bei einer Umsetzung als reine Software im Vergleich zu einer Umsetzung als Custom Instruction auf einem NIOS II Prozessor [3]. Es wird eine Verbesserung der Geschwindigkeit bis zu einem Faktor 165 erreicht.

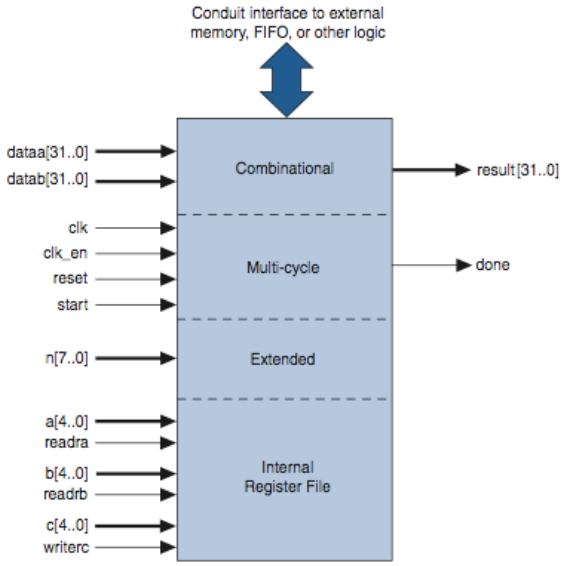


Abbildung 4: Custom Logic Block Interface

III. Altera Custom Instruction Technologie

Eine *Custom Instruction* besteht grundsätzlich aus einem *Custom Logic Block* und einem *Software Macro*. Der Custom Logic Block enthält die Hardware, welche an die ALU angebunden wird. Das Software Macro ermöglicht den Aufruf der Custom Instruction in der Software. Dazu generiert die *NIOS IDE Entwicklungsumgebung* automatisch das entsprechende Macro in der System Header Datei `system.h`. Das Macro wird direkt im C/C++ Code verwendet.

Der Custom Logic Block führt einen Befehl `op` aus und greift dabei auf die beiden Register `dataa` und `datab` zu. Das Ergebnis `result` wird in `dataa` abgelegt. `dataa`, `datab` und `result` sind 32 Bit Register. Ein Aufruf im Programmcode erfolgt durch die Anweisung `dataa = dataa op datab`.

Ein Custom Logic Block wird über das Custom Logic Block Interface bzw. dessen vordefinierte Ports an die ALU angeschlossen [4]. Jedem Port ist per Definition ein fester Name zugeteilt. Je nach Anwendung kann so der Befehlssatz des Prozessors um individuelle Funktionen erweitert werden. Abbildung 4 zeigt das Custom Logic Interface. Je nach Option (Combinatorial usw.) stehen eine Auswahl vordefinierter Ports bzw. Eingänge zur Verfügung. Die Benutzung von Eingängen ist dabei optional. Insgesamt kann zwischen vier verschiedenen Optionen gewählt werden:

A. Combinatorial

Die komplette Logik der kombinatorischen Hardwarebeschleunigung muss in genau einem Takt ausgeführt werden können. Dabei werden optional die Datenre-

Port Name	Direction	Required	Application
clk	Input	Yes	System clock
clk_en	Input	Yes	Clock enable
reset	Input	Yes	Synchronous reset
start	Input	No	Commands custom instruction logic to start execution
done	Output	No	Custom instruction logic indicates to the processor that execution is complete.
dataa[31..0]	Input	No	Input operand to custom instruction
datab[31..0]	Input	No	Input operand to custom instruction
result[31..0]	Output	No	Result from custom instruction

Abbildung 5: Zusätzliche Ports des Multi Cycle Custom Logic Block Interface

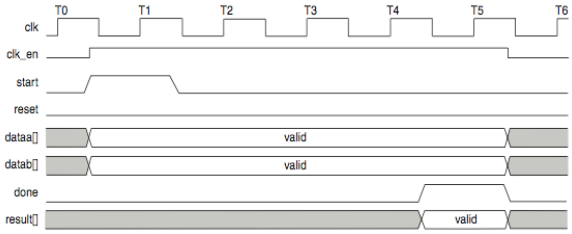


Abbildung 6: Timing einer Multi Cycle Custom Instruction [4]

gister dataa und datab berücksichtigt. Steuersignale bleiben unberücksichtigt. Wird nur ein Input benötigt, so ist dataa zu verwenden. Der NIOS II Prozessor prüft, ob das Ergebnis auch tatsächlich benötigt wird. Ist dies nicht der Fall, so wird es verworfen.

B. Multi Cycle

Multi Cycle Custom Instructions benötigen, wie der Name verrät, mehr als einen Takt zur Ausführung. Die Anzahl der Takte kann dabei konstant oder variabel sein. Bei der Multi Cycle Logic Option werden neben den Datenregistern auch Steuersignale berücksichtigt [4]. Durch die Steuersignale clk, reset, clk_en, start und done wird die Custom Logic mit dem NIOS II Prozessor synchronisiert. Abbildung 5 zeigt die Ports eines Custom Logic Block Interface.

Beim Aufruf der Custom Instruction durch das Software Macro wird die Custom Logic durch das start Signal ausgeführt. start geht für einen Takt auf High. Nach der vom Benutzer vorgegebenen Anzahl Takte liest die CPU das Ergebnis vom entsprechenden Port. Bei einer variablen Bearbeitungsdauer liest die CPU das Ergebnis erst in dem Takt, in dem das done Signal high wird. Sowohl bei einer konstanten als auch bei einer variablen Bearbeitungsdauer müssen die Daten dataa und datab während der gesamten Ausführung anliegen und konstant bleiben.

Der Custom Logic clk Port und der Custom Logic reset Port werden vom NIOS II System Clock bzw. vom NIOS II Master Reset gesteuert. Bei einem low-Signal an clk_en wird das Taktsignal an clk igno riert. clk_en wird nur während der Ausführung einer Custom Instruction auf High gesetzt. Abbildung 6 zeigt das Timing einer Multi Cycle Custom Instruction.

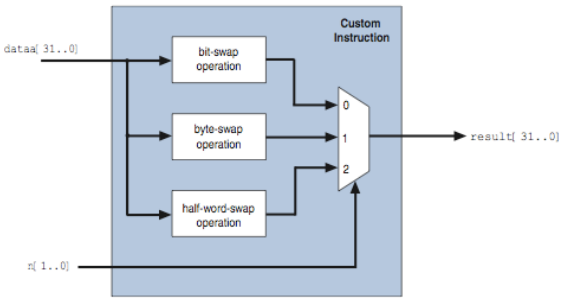


Abbildung 7: Custom Instruction im Extended Mode

C. Extended Mode

Im *Extended Mode* werden mehrere Funktionen in einer Custom Instruction implementiert. Über einen maximal 8-bit breiten Index *n* kann ausgewählt werden, welche Funktion der Custom Logic Block ausführen soll [4]. Die maximale Anzahl an Funktionen in einem Custom Logic Block beträgt somit 256. Abbildung 7 zeigt ein Beispiel für die Implementierung einer Custom Instruction im Extended Mode. Über den hier 2-bit breiten Index *n* wird gesteuert, welche der drei Bitoperationen auf das dataa Register angewandt werden soll. Wie die Auswahl der gewünschten Hardware realisiert wird, ist Sache des Entwicklers. In diesem Beispiel wurde hierzu ein Multiplexer implementiert.

Sowohl *Combinatorial* als auch *Multi Cycle Custom Instructions* können als *Extended* deklariert werden. Dazu wird lediglich der Index *n* eingefügt, mit dem dann die entsprechende Logik ausgewählt werden kann.

D. Internal Register File

Es gibt die Möglichkeit, für die Parameterübergabe an die Custom Instruction auch interne Register zu benutzen. Der Entwickler hat die Möglichkeit, zwischen den Registern dataa bzw. datab des NIOS II Prozessors und den internen Registern der Custom Instruction zu wählen. Ebenso besteht die Möglichkeit, das Ergebnis in ein internes Register der Custom Instruction umzuleiten. Über die Steuersignale readra, readrb und writerc wird festgelegt, ob zum Lesen bzw. Schreiben die internen Register oder die des NIOS II Prozessors verwendet werden sollen. Wird beispielsweise readra auf low gesetzt, so wird der erste Parameter aus dem internen Register der entsprechenden Custom Instruction und nicht aus dataa gelesen. Über die Ports a, b und c wird ein internes Register adressiert. a, b und c sind 5 Bit breit. Somit können jeweils 32 Register adressiert werden.

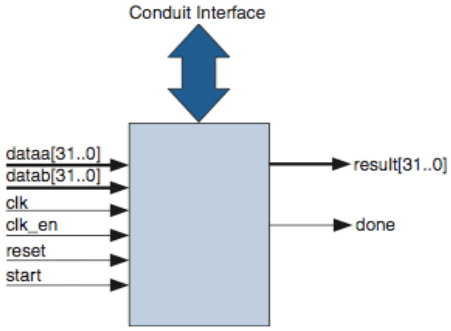


Abbildung 8: Anbindung des Conduit Interface an eine Multi Cycle Custom Instruction

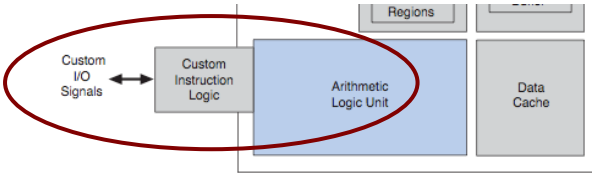


Abbildung 9: Anbindung der Custom Instruction Logik an die ALU des NIOS II Prozessorkern [1]

Bei jeder der vier hier genannten Optionen zur Realisierung einer Custom Instruction besteht außerdem die Möglichkeit, ein zusätzliches Interface an die Custom Instruction anzubinden. Über dieses sogenannte *Conduit Interface* kann die Custom Instruction mit Logik außerhalb des Prozessor-Datenbusses kommunizieren [4]. Abbildung 8 zeigt die Anbindung des *Conduit Interface* an eine *Multi Cycle Custom Instruction*.

Die Anbindung eines *Conduit Interface* an eine Custom Instruction kann verschiedene Gründe haben. Es kann z.B. der Programmfluss der Custom Instruction durch einen externen Speicher gesteuert werden. Es können Ergebnisse zwischengespeichert werden. Auch kann der Datenfluss von den entsprechenden Registern der Custom Instruction direkt in einen externen Speicher umgelenkt werden. Abbildung 9 zeigt den relevanten Ausschnitt des NIOS II Prozessorkerns. In der Abbildung zu sehen ist die Anbindung der Custom Instruction Logic an die ALU. Der Doppelpfeil signalisiert die Anbindung von Custom I/O Leitungen über das Conduit Interface. Dazu werden während der System-Entwicklung Verbindungen an die oberste Schicht des *SOPC Builder* Modul gelegt.

IV. Tools zur Realisierung einer Hardwarebeschleunigung und Custom Instruction in einem Altera FPGA

Altera stellt lizenzfreie als auch kommerzielle Softwaretools zur Programmierung von FPGAs sowie zur Erstellung eines System on a Programmable Chip (SOPC) zur Verfügung (siehe Abbildung 10).

Abbildung 10: Softwaretools von Altera

Die *Quartus II 10 Web Edition* ermöglicht die Erstellung eines VHDL- oder Verilog-Projekts. Der integrierte, java-basierte *SOPC-Builder* in der *Quartus II 10 Web Edition* ermöglicht die Erstellung eines SOPC mit einem von Altera entwickelten NIOS II Softcore Mikroprozessors. Im zugehörigen *NIOS Configuration Wizard* können hierbei die Custom Instructions erstellt werden.

Mit der *NIOS II 10 IDE* Entwicklungsumgebung kann die dazugehörige Applikationssoftware für den NIOS II Prozessor erstellt werden. Die Entwicklungsumgebung besitzt zusätzlich einen *NIOS II C-to-Hardware Acceleration Compiler*. Mit diesem können zeitaufwendige C-Funktionen in der Applikationssoftware mit wenigen Mausklicks auf die Hardwareebene ausgelagert werden, was eine erhebliche Beschleunigung des Algorithmus zur Folge hat.

Diese Funktionalität ist für Testzwecke begrenzt lauffähig und muss optional lizensiert werden. Eine auf diese Weise beschleunigte C-Funktion kann zwar in einen Hardwareblock umgewandelt werden, jedoch kann das System mit diesem Hardwareblock nur temporär betrieben werden.

Mit der Simulationssoftware *ModelSim-Altera 6.5e* können die erstellten VHDL- und Verilog-Projekte, sowie auch das komplette SOPC inklusive der Applikationssoftware, simuliert werden. Auch können eigens erstellte Customs Instructions Projekte in VHDL oder Verilog simuliert werden.

Eine *Mixed Signal* Simulation, d.h. VHDL- und Verilog-Code gemischt in einem Projekt, wird in der lizenzfreien *ModelSim-Altera Starter Edition* der Web Edition leider nicht unterstützt. Hierzu wird ein Mixed Signal Simulator benötigt der z.B. in der Vollversion von *ModelSim* integriert ist.

```
int do_bubble( int *pt_feld )
{
  int i=0,j=0,a,k;
  for(i=0;i<=arraysize;i++){
    if (pt_feld[i]>pt_feld[i+1]){
      a=pt_feld[i];
      pt_feld[i]=pt_feld[i+1];
      pt_feld[i+1]=a;
         for(k=i;k>=0;k--){
           if (pt_feld[i]<=pt_feld[i-1]){
             a=pt_feld[i];
             pt_feld[i]=pt_feld[i-1];
             pt_feld[i-1]=a;
           }
         }
    }
    else {
      j=i;
      while (j>=0 && j<=i){
        j--;
        if (pt_feld[i]<pt_feld[i-j]){
            a=pt_feld[i];
            pt_feld[i]=pt_feld[i-j];
            pt_feld[i-j]=a;
        }
      }
    }
  }
  return 0;
}
```

Abbildung 11: C-Code des Bubble Sort Sortieralgorithmus

V. Praktisches Beispiel von Hardwarebeschleunigung und Custom Instruction

Die folgenden Beispiele wurden mit den in Kapitel vier beschriebenen Werkzeugen erstellt. Als Prototyp-Hardware wird ein Altera DE1 Entwicklungsboard mit einem *Cyclone II* FPGA verwendet. Das eingesetzte EP2C20 FPGA stellt knapp 20.000 LE (Logik Elemente) zur Verfügung. Die Taktfrequenz beträgt in allen nachfolgenden Anwendungen 50 MHz.

A. Erstellen einer Hardwarebeschleunigung

In nachfolgenden Beispiel wird ein *Bubble Sort* Sortieralgorithmus (Abbildung 11) implementiert, welcher ein Feld mit 1000 Zufallszahlen nach der Größe ordnet. Zur Umwandlung in einen Hardwareblock wird der in der *NIOS II 10 IDE* Entwicklungsumgebung verfügbare *NIOS II C-to-Hardware Acceleration Compiler* verwendet.

1) Erstellen eines SOPC

Zu Beginn wird ein neues Quartus-Projekt mit eingebettetem SOPC-Projekt erstellt (Abbildung 12). Nach Erstellung des Projektes werden im *SOPC-Builder* folgende Komponenten instanziert: *NIOS II/e* (e = economy) als Mikroprozessor, 21 kByte On-Chip RAM für Programm- und Arbeitsspeicher, Interval Timer zum Messen der Programmausführzeit, JTAG UART zur Ausgabe der Ausführzeit.

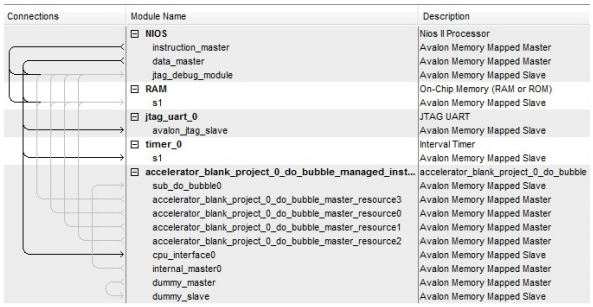


Abbildung 12: SOPC mit Hardwarebeschleunigungsblock

Bei der Generierung des SOPC wird VHDL- oder Verilog-Code erstellt, welche das SOPC in der jeweiligen Hardwarebeschreibungssprache implementieren. Nach Erstellen der HDL-Dateien des SOPC wird das Toplevel als Komponente im Quartus-Projekt eingebunden. Nach dem darauf folgenden Kompiliervorgang des Gesamtprojekts, kann das System mit dem integrierten Quartus-Programmer in das FPGA geladen werden.

2) Erstellen des Programmcodes für den NIOS II Mikroprozessor

Im Verlauf der Generierung des SOPC im *SOPC-Builder* wird eine `ptf`-Datei erzeugt. Auf Basis dieser Systembeschreibungsdatei wird in der NIOS Entwicklungsumgebung ein neues Projekt erstellt. NIOS übernimmt automatisch die Architektur des SOPC in das Projekt. Wird nun ein C-Programm in der Entwicklungsumgebung erstellt, kann es, nach erfolgreichem Kompilierorgang, auf dem FPGA Board ausgeführt und gedebuggt werden.

3) Erzeugen der Hardwarebeschleunigung aus einer C-Funktion

a. Verwendung des NIOS II C-to-Hardware Acceleration Compilers

Nach dem Erstellen des Programmcodes lassen sich in der NIOS Entwicklungsumgebung alle in der Software verwendeten C-Funktionen in einem *Outline*-Fenster anzeigen. Soll eine bestehende C-Funktion nun als Hardwareblock ausgelagert und beschleunigt werden, wird diese im *Outline*-Fenster ausgewählt. Mit einem Rechtsklick und der Auswahl der Funktion *Accelerate with the NIOS II C2H Compiler* wird die C-Funktion für die Umwandlung in HDL-Code vorbereitet. Nach Kompilieren des gesamten NIOS Projektes wird der HDL-Code der ausgewählten C-Funktion erzeugt. Der erstellte Hardwareblock wird automatisch von der Entwicklungsumgebung als IP-Core Komponente dem bestehenden SOPC hinzugefügt. Der Vorgang des Generierens der Hardwarekomponente dauert bei diesem Beispiel-Tutorial mit ca. 20 min. relativ lange. Das SOPC wird nun erneut generiert um die

Änderung wirksam zu machen. Ist der Vorgang abgeschlossen, wird das komplette Quartus HDL-Projekt kompiliert und anschließend auf den FPGA geladen. In der NIOS Entwicklungsumgebung kann das NIOS Projekt erneut mit der jetzt eingefügten Hardwarebeschleunigung gestartet werden. Im Softwarecode müssen bei dieser Implementierung keinerlei Veränderungen vorgenommen werden.

b. Verwendung eines allgemeinen C-to-Hardware Compilers

Eine weitere Möglichkeit eine C-Funktion in einen Hardwareblock auszulagern ist, einen Hardwareblock manuell zu erzeugen. VHDL- oder Verilog-Code einer C-Funktion wird hierzu mit einem beliebigen zu Verfügung stehenden *C-to-HDL Compiler* [5] erzeugt. Für die Verwendung des erzeugten HDL-Codes im SOPC als IP-Core wird eine Schnittstelle zur *Avalon Switch Fabric* des NIOS II erstellt, welche in [6] detailliert beschrieben ist. Bei korrekter Schnittstellenfunktionalität wird dann mittels der *Create New Component*-Funktion des SOPC Builder von dieser Hardwarekomponente ein IP-Core erzeugt. Dieser IP-Core kann einem SOPC hinzugefügt (Abbildung 12) und in der Benutzersoftware des NIOS Prozessors verwendet werden. Bei dieser Implementierung muss für die Verwendung des IP-Cores noch ein eigener Treiber geschrieben werden, welcher den Zugriff auf die Hardwarekomponente in der Software ermöglicht.

B. Erstellung einer Custom Instruction

Im weiteren Verlauf ist zwischen den im *SOPC-Builder* bzw. im NIOS II Softcore bereits vorhandenen Custom Instructions und den benutzerdefinierten Custom Instructions zu unterscheiden.

1) Vorhandene Custom Instruction

Für das nächste Beispiel wird ein von Quartus verfügbares Floating Point Tutorial Softwareprogramm verwendet. Das Programm misst die Ausführdauer von Multiplikation, Addition, Subtraktion und Division zum einen in der Software-Implementierung, zum anderen in der Custom Instruction Implementierung. Zur Zeitmessung werden jeweils 1000 Werte pro Rechenoperation berechnet.

Im NIOS II sind vier Custom Instructions vordefiniert:
Bitswap: Diese Funktion tauscht alle Bits eines 32 Bit Worts in einem Takt.
Endian Converter: Mit dieser Custom Instruction ist es möglich, Daten mit Prozessoren, welche *Big Endian* unterstützen, auszutauschen. Die NIOS II CPU gehört zu den *Little Endian* Prozessorarchitekturen.
Floating Point Hardware: Beschleunigte Fließkomma-Rechenoperationen für mathematische aufwändigen Programmcode.
Interrupt Vector: Verbessert die Interrupt-Latenzzeit um bis zu 20%.

a. Erstellen des SOPC

Das im vorherigen Beispiel beschriebenen SOPC wird übernommen. Zusätzlich wird wegen des größeren Programmspeicherbedarfs ein externer 8 MByte SDRAM Speicher verwendet. Für eine einfachere Zeitmessung wird ein Performance Counter hinzugefügt.

b. Erstellen einer vorhandenen Custom Instruction

Mit dem *SOPC Builder* wird im erstellten System der NIOS II Prozesserbaustein ausgewählt. Es erscheint der *NIOS Configuration Wizard*. Unter dem Menüpunkt *Customs Instructions* erscheinen die vier bereits vorhandenen Custom Instructions. Durch Doppelklick auf die *Floating Point Custom Instruction*, wird diese dem NIOS II Prozessor hinzugefügt. Die Schritte vom Generieren des SOPC, bis hin zum Erstellen des NIOS Projekts sind mit dem vorherigen Beispiel identisch.

c. Erstellen eines Software Floating Point Beispiels

Für die Verwendung der vorhanden Custom Instruction ist keine zusätzliche Softwareimplementierung notwendig. Bei einer Rechenoperation mit Fließkomma-Datentypen in der Applikationssoftware wird automatisch die *Floating Point Custom Instruction* verwendet.

2) Benutzerdefinierte Custom Instruction

In diesem Beispiel wird eine von Altera verfügbare *CRC Custom Instruction* verwendet. Die zugehörige Software berechnet die CRC von 2^{20} Zufallszahlen mit zwei verschiedenen Verfahren:

1. `crcSlow`: Modulo-2 Division in Software (Abbildung 13)
2. `crcFast`: Lookup-Tabelle in Software
3. `crc`-Opcode als *CRC Custom Instruction*

Das Importieren einer benutzerdefinierten Custom Instruction ist in der Quartus II 10 Web Edition unter Windows 7 aufgrund eines Programmfehlers (Bug?) leider nicht möglich. Aus diesem Grund wurde für dieses Beispiel die Vorgängerversion Quartus II 9.1 SP2 Web Edition eingesetzt. Das im letzten Beispiel beschriebene SOPC wird übernommen.

a. Erstellen einer benutzerdefinierten Custom Instruction

Eine Custom Instruction Datei muss in folgenden fünf Dateiformaten vorliegen: VHDL, Verilog, EDIF (netlist-Datei), .bdf oder .vqm. Wird diese Datei in VHDL oder Verilog erstellt, muss eine korrekte

```
crc crcSlow(unsigned char const message[],
int nBytes)
{
    crc remainder = INITIAL_REMAINDER;
    int byte;
    unsigned char   bit;
    //Perform mod2 div, a byte at a time.
    for (byte = 0; byte < nBytes; ++byte)
    {
       //Bring next byte into remainder
       remainder^=(REFLECT_DATA (message
                   [byte])<<(WIDTH-8));
       //Perform mod2 div, a bit at a time
        for (bit = 8; bit > 0; --bit)
        {
            //Try to div current data bit
            if (remainder & TOPBIT)
            {
                remainder=(remainder<<1)
                          ^POLYNOMIAL;
            }
            else
            {
                remainder=(remainder<<1);
            }
        }
    }
    //final remainder is the CRC result
    return(REFLECT_REMAINDER(remainder)
           ^FINAL_XOR_VALUE);
}   /* crcSlow() */
```

Abbildung 13: CRC C Code (`crcSlow`)

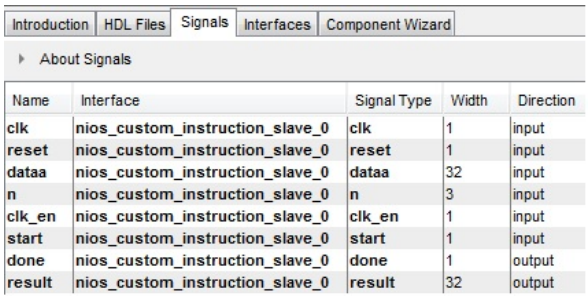
Schnittstelle zum NIOS II Softcore definiert sein (Abbildung 4).

b. Importieren einer benutzerdefinierten Custom Instruction

Im Menüpunkt *Custom Instructions* im *NIOS Configuration Wizard* können benutzerdefinierte Custom Instructions unter dem Button *Import* hinzugefügt werden. Es erscheint der *Component Editor*. In diesem werden die vorhandenen Custom Instruction VHDL- oder Verilog-Dateien hinzugefügt. Die Ein- und Ausgänge dieser Custom Instruction werden bei korrekter Schnittstellendefinition automatisch eingelesen. Eine Signalzuordnung des importierten CRC HDL-Codes ist in Abbildung 14 zu sehen. Ist dies nicht der Fall müssen die Schnittstellen manuell zugewiesen werden, was in [4] detailliert beschrieben ist. Nach Fertigstellung wird wie im vorherigen Beispiel die Custom Instruction dem NIOS II hinzugefügt.

c. Softwarezugriff auf benutzerdefinierte Custom Instructions

Eine `system.h` Datei, welche in [7] beschrieben ist, wird automatisch bei der Erzeugung des SOPC durch den *SOPC Builder* erstellt. In dieser Datei sind die für den Hardwarezugriff notwendigen Adressen und Makros, wie z.B. Zugriff auf Custom Instructions



Introduction | HDL Files | Signals | Interfaces | Component Wizard

About Signals

Name	Interface	Signal Type	Width	Direction
clk	nios_custom_instruction_slave_0	clk	1	input
reset	nios_custom_instruction_slave_0	reset	1	input
dataa	nios_custom_instruction_slave_0	dataa	32	input
n	nios_custom_instruction_slave_0	n	3	input
clk_en	nios_custom_instruction_slave_0	clk_en	1	input
start	nios_custom_instruction_slave_0	start	1	input
done	nios_custom_instruction_slave_0	done	1	output
result	nios_custom_instruction_slave_0	result	32	output

Abbildung 14: Signalzuordnung einer Custom Instruction

Typ	Ausführzeit	LE Verbrauch
Sortieralg. (SW)	7,4 s	1550
Sortieralg. (C2H)	0,5 s	3960

Abbildung 15: Resultat von Software und Hardwarebeschleunigung beim Sortieren von 1000 Werten

oder Peripherie definiert. Makros, welche auf eine Custom Instruction zugreifen, können auch selbst definiert werden. Der Befehl `__builtin_custom_` signalisiert dem Compiler den Zugriff auf eine Custom Instruction. Genauere Angaben zu den Compilerbefehlen werden in [4] erläutert. Mit dem Befehl `__builtin_custom_ini(int n, int dataa)` wird beispielsweise eine Custom Instruction adressiert. Der Parameter `n` legt hierbei die Zieladresse fest. Die Zieladresse ist die im Adressbereich verwendete Basisadresse. Besitzt die Custom Instruction mehrere Funktionen, wie es z.B. im Extended Mode möglich ist, wird für `n` ein entsprechender Offset für die jeweilige Funktion verwendet. Mit dem Parameter `dataa` werden die Daten übergeben.

VI. Vergleich von Geschwindigkeitsverbesserungen an den Beispielen

A. Ergebnisse der Hardwarebeschleunigung

Abbildung 15 zeigt die Ausführzeit und den Verbrauch an *Logic Elements* (LE) im FPGA des Hardwarebeschleunigungsbeispiels. Der LE-Verbrauch der Hardwarebeschleunigung ist um den Faktor 2,5 größer als bei reiner Softwareimplementierung. Jedoch ist der zeitliche Vorteil beim Sortieralgorithmus mit dem Faktor 15 erheblich größer. Der Beschleunigungsfaktor hängt von der Softwareimplementierung des jeweiligen Algorithmus wie auch vom verwendeten C-to-HDL Compiler ab. Bei der C-Implementierung des Algorithmus sollte zuerst die Umsetzung des C-to-HDL Compiler verstanden werden. Dies ermöglicht einen optimalen LE-Verbrauch und Geschwindigkeitsanstieg [8]. In diesem Beispiel besteht der Algorithmus aus `for`-Schleifen und `if`-Abfragen (Abbildung 11).

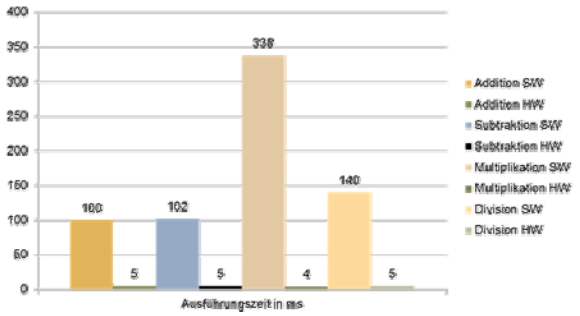


Abbildung 16: Grafik der Ausführungszeiten der Rechenoperationen mit und ohne *Floating Point Custom Instruction*

Typ	Ausführzeit Software	Ausführzeit Custom Instruction	Beschleunigungs-Faktor
Addition	100 ms	5 ms	20
Subtraktion	102 ms	5 ms	20
Multiplikation	338 ms	4 ms	85
Division	140 ms	5 ms	28

Abbildung 17: Tabelle der Ausführungszeiten der Rechenoperationen mit und ohne *Floating Point Custom Instruction*

B. Ergebnisse der Floating Point Custom Instruction

Abbildung 17 zeigt die Ausführzeit zwischen Rechenoperationen, welche mit und ohne *Floating Point Custom Instruction* ausgeführt wurden. Man erkennt, dass der Beschleunigungsfaktor mindestens um den Faktor 20 höher ist, als bei der normalen Rechenoperation in Software. Die *Floating Point Custom Instruction* verwendet bei der Multiplikation sieben integrierte Hardware-Multiplizierer des FPGAs und verbraucht ca. 1300 zusätzliche LEs. Das Resultat der verschiedenen Ergebnisse hängt natürlich vom Hardwaredesign, von vorhandenen Hardcore-IP-Blöcken, z.B. den Multiplizierern im Cyclone II, als auch von der Implementierung des Programmcodes ab [9].
Abbildung 17 zeigt die Messwerte in Tabellenform.

C. Ergebnisse der CRC Custom Instruction

In Abbildung 19 werden die Ausführungszeiten der CRC-Berechnung dargestellt. Hierbei wurden nun mehrere Verfahren miteinander verglichen. Die beiden C-Code Implementierungen (Slow bzw. Fast Software CRC Algorithmus) wurden in Software vermessen als auch per C2H-Compiler beschleunigt. Der fünfte Messwert ist die Implementierung der CRC-Berechnung als Custom Instruction.

Die C2H-CRC beschleunigt die schnelle Software CRC-Implementierung um den Faktor 50. Bei der Beschleunigung der langsamen Software CRC-Implementierung wurde sogar ein Beschleunigungsfaktor von 78 erreicht. Entgegen den Erwartungen sind die in der VHDL-Implementierung erzielten ab-

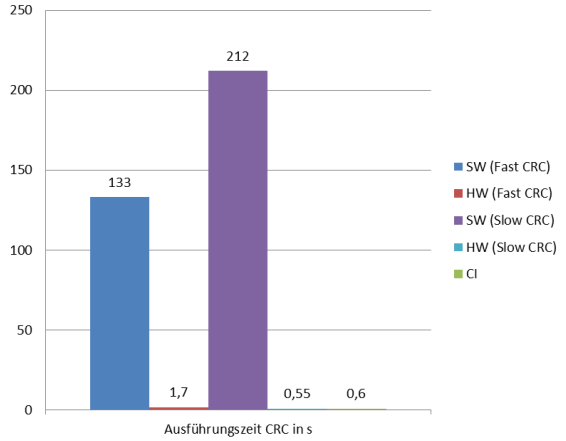


Abbildung 18: Grafik der Ausführungszeiten der per Software und per Custom Instruction implementierten CRC-Berechnung

Typ	Ausführzeit des C-Codes	Ausführzeit in VHDL	LE-Verbrauch
Slow Software CRC	212 s	0,55 s	1260
Fast Software CRC	133 s	1,7 s	1330
Custom Instruction CRC	2,66 s	0,6 s	304

Abbildung 19: Ausführungszeiten der Software und Custom Instruction implementierten CRC

soluten Ausführungszeiten noch besser als die des vermeintlich schnelleren Codes. Hier ist zu vermuten, dass sich der langsamere, aber damit weniger komplexe CRC-Code erfolgreicher automatisch in VHDL übersetzen und so besser beschleunigen lässt.

In der letzten Spalte der tabellarischen Ansicht der Ausführungszeiten (Abbildung 17) ist der Verbrauch an LE angegeben. Hier zeigt sich ganz deutlich der Vorsprung der direkten Implementierung eines Algorithmus in einer VHDL_Beschreibumg. Lediglich 304 LE werden für die CRC-Berechnung in Form einer Custom Instruction benötigt. Im Gegensatz dazu benötigen die automatisch übersetzten Versionen 4-5 mal so viele Logikelemente.

Auf alle Fälle überraschend ist im direkten Vergleich das gute Abschneiden der vollautomatischen HDL-Transformation der beiden C-basierten CRC-Funktionen per C2H-Compiler. Die Ausführungszeiten sind konkurrenzfähig mit der reinen CRC-Implementierung als Custom Instruction. In einem weiteren Schritt wären nun die unterschiedlichen CRC-Implementierungen hinsichtlich ihrer Code- bzw. HDL-Umsetzung genauer zu analysieren und zu vergleichen.

In diesem Zusammenhang wichtig zu erwähnen ist der Einfluss der Rechenleistung des Mikrocontrollers. Betrachtet man verschiedene Software-Algorithmen in der Praxis, so lässt sich bei der Umsetzung einzelner Funktionen in VHDL zwar eine erhebliche Beschleunigung erreichen, jedoch muss man immer die Laufzeit der gesamten Applikation betrachten. Der verbliebene Rest des ja weiterhin konventionell abgearbeiteten C-Codes profitiert von einem schnelleren Prozessor erheblich.

So wurde bei zusätzlichen Messungen mit der schnelleren *NIOS II/f* (fast)-CPU-Variante eine weitere Beschleunigung der Gesamtapplikation um den Faktor 10 erreicht, bei ca. 1,6 fachem Verbrauch an LEs. Hierbei treten jedoch vermehrt nichtdeterministische Effekte auf, welche vermutlich auf den Einsatz der modernen CPU-Beschleunigungsverfahren wie Pipelining und Caches in der f-Version zurück zu führen sind. Abhängig von der Applikation führen diese zu Geschwindigkeitsverlusten bei eigentlich schnelleren Prozessoren.

VII. FAZIT

In diesem Paper wurden die unterschiedlichen Methoden der Hardwarebeschleunigung in FPGAs gezeigt. Dabei wurde speziell auf die Implementierung von Custom Instructions von Altera eingegangen. In einem praktischen Beispiel wurde demonstriert, wie diese Methoden mit den Softwarewerkzeugen von Altera in relativ kurzer Zeit in einem FPGA umgesetzt werden können.

Die verfügbaren Werkzeuge bieten unterschiedlich komplizierte und zeitaufwändige Verfahren eine Hardwarebeschleunigung zu erreichen. Bei umfangreichen C-Funktionen benötigt der sehr anwenderfreundlich zu bedienende *Altera C-to-Hardware Acceleration Compiler* jedoch relativ lange Rechenzeiten, um die Funktionen in HDL Code zu übersetzen. Durch den geringeren Entwicklungsaufwand können sich die Investitionskosten des Compilers von ca. 3.000 € jedoch schnell amortisieren. Aufgrund der unterschiedlichen Ergebnisse ist eine genaue Vorab-Analyse des Algorithmus anzuraten. Insbesondere sollte bestehender VHDL-Code vor einem Einsatz intensiv auf seine Synthesequalität hin untersucht werden.

Generell lässt sich durch alle hier vorgestellten Verfahren zur Hardwarebeschleunigung eine erhebliche Geschwindigkeitsverbesserung erreichen, was wiederum den Energieverbrauch reduziert bzw. die Akkulaufzeit bei kabellosen Geräten erhöht.

LITERATURVERZEICHNIS

[1] Altera, “NIOS II Processor Reference Handbook”, July 2010
[2] Aoudni, “Custom Instruction Integration Method within Reconfigurable SoC and FPGA Devices”, *The 18th International Conference on Microelectronics*, 2006
[3] Altera, “Custom Instructions for the NIOS II Embedded Processor”, *NIOS 2 Performance Benchmarks, AN 188*, Version 1.2, 2002
[4] Altera, “NIOS II Custom Instruction User Guide”, May 2008
[5] Nadav Rotem, Online C-to-Verilog Compiler, Haifa University, Israel, November 2010
[6] Altera, “Avalon Interface Specification”, August 2010
[7] Altera,” NIOS II Software Developers Handbook”, July 2010
[8] Altera, “NIOS II C2H Compiler User Guide”, November 2009
[9] Altera, “Using NIOS II Floating-Point Custom Instructions Tutorial”, February 2010

Christian Siebert erlangte den Grad des Bachelor of Engineering, 2010 an der HTWG Konstanz. Nach Abschluss seines Bachelorstudiums mit der Bachelorarbeit „Energieversorgung von Mikropowerschaltungen über Plastik-Lichtwellenleiter“ studiert er aktuell im Master „Elektrische Systeme“. Danach stebt Christian Siebert eine Tätigkeit als Hardware-Entwicklungsingenieur an.

Christian Seibt erlangte den Grad des Bachelor of Engineering, 2010 an der HTWG Konstanz. Mit dem Thema seiner Bachelorarbeit „CAN-IP Core Integration in ein FPGA“ erarbeitete er sich die Grundlagen im Bereich SOPC, welche für dieses Paper von Nutzen waren. Herr Seibt wird 2012 sein derzeitiges Masterstudium an der HTWG abschließen.

Gregor Burmberger ist Professor für Eingebettete Systeme und Prozessorarchitekturen an der HTWG Konstanz. Vor seiner Berufung war er als Experte u.a. auf dem Gebiet der Bussysteme für namhafte Automobilhersteller tätig. Mit seiner langjährigen Erfahrung mit FPGAs und SOPCs unterstützt er Bachelor- und Masterarbeiten im Bereich Digitale Systeme. Gregor Burmberger hat an der TU München Elektrotechnik studiert und in Bereich Realzeitsysteme promoviert.

Entwicklung einer 8-Kanal-AD-Wandlerkarte mit Datenpufferung und Ethernetanbindung

Dirk Benyoucef, Steffen Mauch

***Zusammenfassung*—Dieser Beitrag diskutiert die Entwicklung eines intelligenten Stromzählers im Kontext des Forschungsprojekts „Smart Metering: Disagreggation von Energieverbrauchern“. Der Schwerpunkt der Betrachtung liegt auf der notwendigen Data Acquisition Card, welche die analogen Messwerte für Strom und Spannung erfasst und über einen AD-Wandler digitalisiert. Die Messdaten werden vom AD-Wandler kontinuierlich per SPI-Bus bereitgestellt und müssen zwischengespeichert werden. Für diese Aufgabe wird ein FPGA mit angebundenem RAM eingesetzt, so dass die Daten anschließend als Pakete über die USB-Schnittstelle in Echtzeit in das Linux-System übertragen werden können.**

***Schlüsselwörter*—AD-Wandler, FPGA, Smart Metering, RAM.**

I. Einleitung

Das Thema Energiesparen ist eines der wichtigsten Themen unserer Zeit. Durch die große Durchdringung von elektrischen Systemen in unserem häuslichen Umfeld haben sich viele elektrische Verbraucher im Laufe der Zeit angesammelt. Der Energieverbrauch jedes einzelnen summiert sich zu einem erheblichen Bedarf auf [1]. Der Nutzer des elektrischen Verbrauchers hat nur die Information des gesamten Energiebedarfs und kann damit nicht auf den Energiebedarf des einzelnen Verbrauchers schließen. Auch erhält er erst nach einem Jahr eine Abrechnung, die den Energieverbrauch aller seiner Geräte kumuliert darstellt. Die Einführung der intelligenten Stromzähler (Smart Meter) erlaubt zwar einen Überblick über den aktuellen Gesamtverbrauch, liefert aber keine Information, welche Verbraucher aktiv sind und wie groß deren Energiebedarf ist [2]. Der Nutzer der Geräte benötigt für das Verändern seines Verbrauchsverhaltens, also das Energiesparen, fundierte und detaillierte Informationen. Diese wären

- wann welcher Verbraucher am Netz ist,
- wie groß der aktuelle und der kumulierte Verbrauch dieses Verbrauchers ist und
- wie der Energiebedarf eines Verbrauchers im Vergleich zu ähnlichen Verbrauchern ist.

Diese Informationen münden in eine Monatsabrechnung, die detailliert alle Verbraucher mit ihrer Nutzungsdauer, dem Energiebedarf und einem Vergleich zu ähnlichen Verbrauchern darstellt. Anhand dieser Daten kann der Kunde sein Verhalten überdenken und gegebenenfalls anpassen.

Seit 1. Januar 2010 sind Energieversorger gesetzlich dazu verpflichtet, intelligente Stromzähler Kunden zur Verfügung zu stellen. Diese neuen Energiezähler erlauben es zwar, Informationen in elektrischer Form über den Energieverbrauch der letzten 15 Minuten zu erhalten, liefern aber keine detaillierte Informationen, welche eine Disaggregation von Energieverbrauchern ermöglichen würden [3]. Aus diesem Grund ist die Zielsetzung, einen intelligenten Stromzähler zu entwickeln, welcher eine Identifikation einzelner Verbraucher aus dem Gesamtverbrauch erlaubt [4].

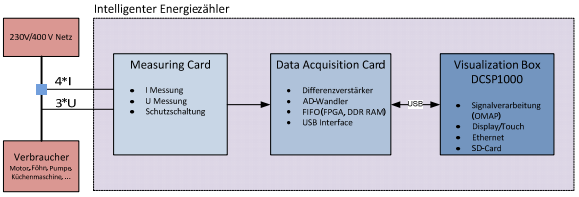


Abbildung 1: Blockschaltbild des intelligenten Stromzählers

II. Intelligenter Stromzähler

Die Realisierung des intelligenten Stromzählers ist in Abbildung 1 als Blockschaltbild dargestellt. Das System besteht aus drei Einheiten: der *Measuring Card* für das Messen der Ströme und Spannungen und die Skalierung der Größen, der *Data Acquisition Card* für das Digitalisieren der analogen Signale und die *Visualization Box* für die Speicherung der Messdaten beziehungsweise die Ausführung der Algorithmen für die Disaggregation der verschiedenen Verbraucher. Darüber hinaus hat die Visualization Box die Aufgabe der Visualisierung der Ergebnisse und der externen Kommunikation.

D. Benyoucef, bed@hs-furtwangen.de ist Mitglied der Hochschule Furtwangen, Robert-Gerwig-Platz 1, 78120 Furtwangen und Steffen Mauch, steffen.mauch@gmail.com ist Doktorand an der TU Ilmenau, Ehrenbergstr. 29, 98693 Ilmenau

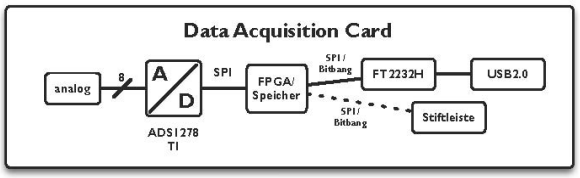


Abbildung 2: Blockschaltbild der Data Acquisition Card

A. Data Acquisition Card

Die Data Acquisition Card besitzt acht parallel arbeitende differentielle Eingangskanäle und einen USB-Anschluss als Ausgang, siehe Abbildung 2. Die Eingangsstufe wird jeweils durch einen Differenzverstärker (OPA1632 von Texas Instrument) gebildet. Dieser führt zugleich eine Tiefpassfilterung durch, um das Abtasttheorem einzuhalten. Als AD-Wandler wird der ADS1278 von TI verwendet, mit einer Auflösung von 24 Bit bei acht parallelen differentiellen Kanälen. Die maximale Abtastfrequenz des ADS1278 beträgt 125 kHz. Somit ergibt sich eine maximale Datenrate von drei Megabyte pro Sekunde. Als Ausgabeformate des AD-Wandlers werden sowohl der Frame-Sync- als auch der SPI-Modus unterstützt. Der nachfolgende FPGA hat die Aufgabe, die Daten aus dem Analog-Digitalwandler periodisch auszulesen und diese zwischenzuspeichern. Die externe Anbindung an die USB-Schnittstelle erfolgt über den Baustein FT2232H[1] der Firma FTDI. Dieser bietet keinen isochronous Transfer, daher ist ein vorheriges Zwischenspeichern des Datenstroms unbedingt notwendig. Für diesen Baustein gibt es eine freie Bibliothek „libftdi“, welche auf „libusb“ zurückgreift. Diese Kombination hat den Vorteil, dass kein spezieller USB-Treiber geschrieben werden muss und das Programm plattformübergreifend lauffähig ist.

B. Datentransfer

Der AD-Wandler liefert ein DATA_READY-Signal, welches das Startsignal für den FPGA darstellt. Je nach Anzahl der aktiven Kanäle liest der FPGA über den implementierten SPI-Master pro Kanal drei Bytes. Bei acht Kanälen würde der SPI-Master somit insgesamt 24 Bytes lesen. Werden die Daten nicht rechtzeitig gelesen, überschreibt der AD-Wandler allerdings die vorhandenen Werte. Dies wird durch einen kurzen Puls an dem DATA_READY-Pin signalisiert.

Der FT2232H unterstützt eine Vielzahl von Protokollen, von JTAG über SPI hin zu synchronem FIFO Modus. Bei SPI liegt die maximale Taktfrequenz bei 30 MHz, was zur Folge hat, dass die maximale Transferrate 3,75 MByte pro Sekunde beträgt. Im synchronen FIFO-Modus liegt die maximale Transferrate bei 25 MByte pro Sekunde. Ein kontinuierlicher Datenstrom ist jedoch nicht ohne Zwischenspeicherung möglich. Der FIFO-Modus kommt bisweilen nicht zum Einsatz, da das Beagleboard per SPI direkt für Testzwecke mit dem FPGA verbunden werden kann. In zukünftigen Versionen des Systems ist die Verwendung des FIFO-Modus geplant.

Die Vorgabe war, dass es, selbst wenn zwei bis drei Sekunden der USB-Bus ausgelastet ist, zu keinem Datenverlust kommt. Die zwischengespeicherten Daten werden in den DDR-SDRAM geschrieben und von dem SPI-Slave gelesen, sobald der FT2232H Daten über SPI anfordert.

In Abbildung 2 ist gleichzeitig auch noch als optionale Möglichkeit der Bitbang Modus beziehungsweise sync. FIFO Modus eingezeichnet. Der Vorteil hierbei liegt in der höheren Datenrate. Dies ermöglicht einen Ausbau des Systems auf höhere Abtastfrequenzen beziehungsweise eine größere Anzahl von Kanälen.

C. FPGA

Die Kernaufgabe des FPGA ist die Zwischenspeicherung der Daten. Zum Einsatz kommt ein Spartan 3E FPGA XC3S500E-4FG320C von Xilinx. Das Datenvolumen beläuft sich momentan auf maximal 24 MBit/s = 3 MByte/s (8 Kanäle * 125 kHz * 24 Bit), die ohne Datenverlust der Visualization Box zugeführt werden müssen. Dabei arbeitet der USB-Bus in Kombination mit dem verwendeten Linux System (Angstrom) paketorientiert und kann somit nicht in Echtzeit die Daten entgegennehmen. Die Verwendung des isochronous Transfer bedarf spezieller Treiber, welche meistens nicht für alle Plattformen angeboten werden und somit möglicherweise selbst entwickelt werden müssen.

Im Rahmen dieses Beitrags wird erläutert, wie die Zwischenspeicherung der Daten auf einem FPGA mit angebundenem DDR-SDRAM realisiert wurde.

III. DAC-BUFFER-KONZEPT

Im folgenden Abschnitt werden zwei Konzepte der DAC-Buffer detailliert vorgestellt. Die Hauptaufgabe ist die Zwischenspeicherung der Daten für eine blockweise Verarbeitung. Gleichzeitig soll eine Vorverarbeitung der Daten im FPGA ermöglicht werden.

A. Konzept DDR-Core

Bei dem in Abbildung 3 dargestelltem Konzept wird ein SPI-Master, ein SPI-Slave und ein DDR-Core miteinander verschaltet. Xilinx liefert einen Memory Interface Generator, kurz MIG, bei seiner ISE Entwicklungsumgebung mit. Dieses Tool kann für verschiedene Memory-Bausteine komplette Entities und Architectures erzeugen, in welchen die gesamte Ansteuerung des DDR-SDRAMs

[1] USB 2.0 High-Speed Gerät

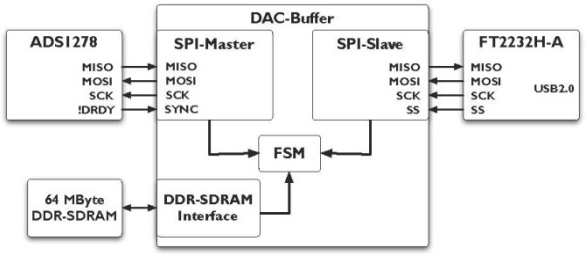


Abbildung 3: FPGA Design

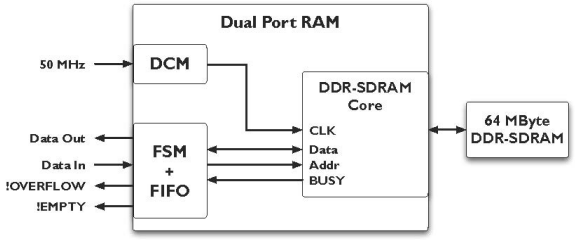


Abbildung 4: Blockschaltbild Dual Port Interface

programmiert ist. Ein Vorteil ist, dass die Timings eingehalten werden und die Constraints automatisch in die UCF-Datei eingefügt werden.

Des Weiteren muss kein Speichercontroller in VHDL implementiert werden. Der DDR-SDRAM ist nicht Dual Port-fähig, das heißt, dass man entweder lesen oder schreiben kann, jedoch nicht beides gleichzeitig. Jedoch kann man durch einen Trick den DDR-SDRAM Dual Port fähig machen. Der Haupttakt, der bei 50 MHz liegt, wird durch eine DCM Einheit verdoppelt. Dieser Takt wird in dem Dual Port Modul benutzt und somit verhält sich das Modul nach außen wie ein Dual Port RAM.

Der große Vorteil hierbei ist, dass SPI-Master beziehungsweise SPI-Slave keine Rücksicht aufeinander nehmen müssen. Die Dual Port Einheit kann bei 50 MHz in einem Zyklus lesen und schreiben. Dieses Modul, schematisch dargestellt in Abbildung 4, erfüllt die gestellten Aufgaben als schneller und großer Zwischenspeicher.

Um zu zeigen, dass die Anforderungen erfüllbar sind, wurde dieses Konzept implementiert. Auf Grund der eingeschränkten Erweiterbarkeit wurde diese Variante jedoch nicht weiter realisiert. Hauptgrund sind die im Laufe des Projektes geänderten Randbedingungen, welche eine Anbindung an Ethernet notwendig machen. Stattdessen wurde auf das Konzept mit dem MicroBlaze-Softcore gewechselt. MicroBlaze ist ein für FPGAs der Firma Xilinx entworfener Mikrocontroller, welcher als Besonderheit nicht als physische Hardware realisiert ist, sondern in einer Hardwarebeschreibungssprache vorliegt.

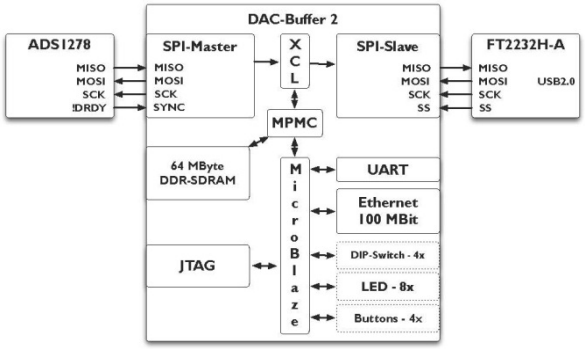


Abbildung 5: FPGA Design MicroBlaze

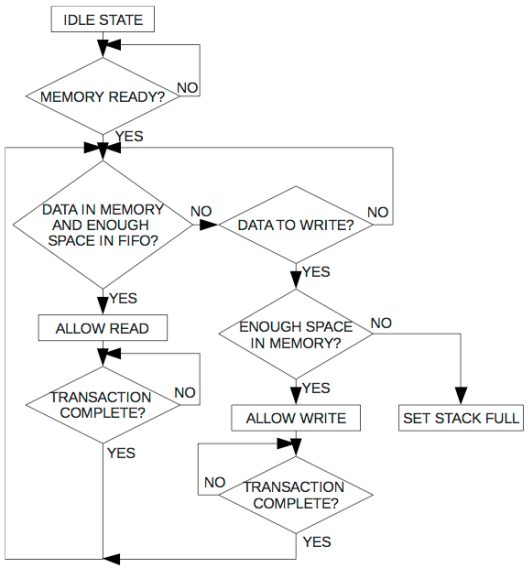


Abbildung 6: Zustandsmaschine - XCL Interface

B. Konzept MicroBlaze

Das in Abbildung 5 dargestellte Konzept bietet gegenüber dem aus Abbildung 3 enorme Vorteile in Bezug auf die Erweiterbarkeit. Durch den Einsatz des MicroBlaze können die Daten relativ einfach in ein angeschlossenes Netzwerk gestreamt werden. Ebenso ist eine Konfiguration über UART, Ethernet oder auch einem weiteren SPI-Slave, angeschlossen an den OPB-Bus des MicroBlaze, problemlos möglich.

Der MicroBlaze benutzt den Multi-Port Memory Controller, kurz MPMC. Dies ist ein voll parametrisierbarer Speichercontroller, der sowohl SDRAM, DDR- und DDR2-SDRAM-Speicher unterstützt. Der MPMC bietet zwischen einem und acht Ports. Als Interface kann zum Beispiel der Xilinx CacheLink (XCL) oder auch der LocalLink benutzt werden. Der MPMC kann jedoch ebenso ohne MicroBlaze als standalone System genutzt werden.

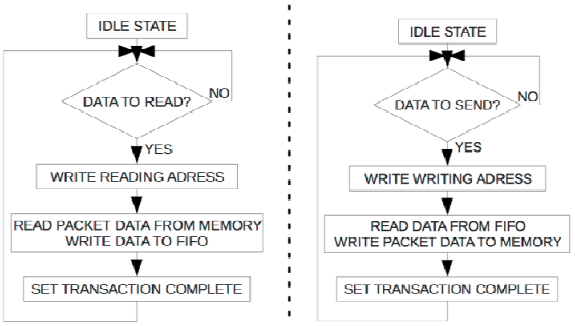


Abbildung 7: Zustandsmaschine - rechts Schreiben und links Lesen

Der MircoBlaze bietet jedoch den großen Vorteil, dass er Linux-kompatibel ist. Somit kann bei der Entwicklung eines Programms für das Streamen über Ethernet auf Linux-Bibliotheken zurückgegriffen werden.

Abbildung 5 zeigt schematisch die Realisierung. Der SPI-Master und -Slave werden über eine einfache Zustandsmaschine mit Hilfe eines XCL Interface an den MPMC angeschlossen. Die Zustandsmaschine ist in Abbildung 6 abgebildet. Nach einem RESET-Signal wird so lange gewartet, bis die Initialisierung des externen Memory abgeschlossen ist. Anschließend wird die Zustandsmaschine je nach Transaktion abgearbeitet. Sowohl für Lese- als auch Schreibprozesse wird auf die Beendigung der Transaktion gewartet. Der Lese- beziehungsweise Schreibprozess setzt nach Beendigung der Transaktion das COMPLETE Flag. Falls der Speicher voll ist, wird das STACK FULL Bit gesetzt und in diesem Zustand verharrt. Dieser Zustand kann lediglich durch ein RESET-Signal verlassen werden.

In der Abbildung 7 ist die Zustandsmaschine für den Lese- und Schreibprozess detailliert abgebildet. Bei der Initiierung eines Lesezugriffs wird die Leseadresse geschrieben und die entsprechenden Daten werden gelesen. Diese werden in das FIFO geschrieben, anschließend wird das TRANSACTION COMPLETE Bit gesetzt. Schließlich wird auf weitere Lesezugriffe gewartet. Der Schreibzugriff erfolgt analog zu dem Lesezugriff, jedoch werden die Daten aus dem FIFO in den DDR-SDRAM geschrieben.

Für die Verifikation und Messung der Performance wurden ein Datengenerator und ein -komparator geschrieben. Der Datengenerator legt anstelle des SPI-Masters Daten periodisch an und der Datenkomparator liest diese Daten wieder aus und vergleicht sie mit den angelegten. Diese beiden Module wurden ebenso synthetisiert und auf den FPGA integriert. Die Taktrate wurde bei den Tests variiert. Bei 12,4 MHz ergab sich eine Lese- und Schreibrate von 148,8 Megabytes pro Sekunde. Hierbei wurde bei jeweils sechs Bytes geschrieben und sechs Bytes gelesen. Das Ergebnis übertrifft um mehr als den Faktor 20 die geforderte Datenrate.

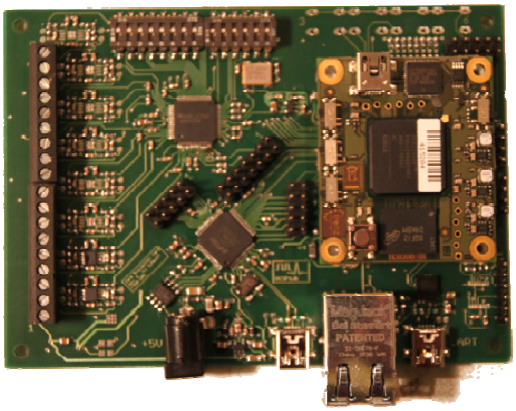


Abbildung 8: Platine mit FPGA Modul

Abbildung 9: Blockschaltbild Control ADS1278

C. Integration des FPGA-Moduls auf die DAC-Platine

Die Integration des FPGA auf die entwickelte DAC Platine erfolgt mit einem Mikromodul Spartan 3E, welches den aktuellen Spartan-3E 1600k FPGA zusammen mit einem USB 2.0 Microcontroller (Cypress FX2 USB2.0 CY7C68013A-56LFXC), einem DDR-SDRAM (512 MBit-DDR-SDRAM Micron MT46V32M16BN-6:F), einem 32 MBit SPI Flash und einer Spannungsversorgung enthält. Die Programmierung des SPI Flash erfolgt über die USB-Schnittstelle. Der Takt des Onboard beträgt 100 MHz und die Abmessungen des Moduls liegen bei 40,5 mm x 47,5 mm.

Der Cypress FX2 USB 2.0 wird lediglich für die Programmierung des FPGAs gebraucht. Er wird nicht für die Datenübertragung über die USB-Schnittstelle benutzt. Das Mikromodul kann mit Hirose Steckverbinder direkt auf die Data Acquisition Card gesteckt werden. Dies hat den Vorteil, dass die kritischen Pfade zwischen DDR-SDRAM und FPGA nicht geroutet werden müssen und das Mikromodul nicht in den Schaltplan komplett eingearbeitet werden muss. Zudem ist es vergleichsweise billig, da es industriell mit hoher Stückzahl produziert wird.

Das Design der Data Acquisition Card gestaltet sich deutlich einfacher, da kein BGA geroutet werden muss. Zudem kann hierdurch die Data Acquisition Card von Hand bestückt werden, ansonsten wäre eine BGA Bestückungsmaschine notwendig. Die Produktion eines Komplettsystems wäre unverhältnismäßig teuer, da momentan geringe Stückzahlen benötigt werden. In Abbildung 8 ist die fertige Platine dargestellt.

D. Konfiguration des AD-Wandlers

Der ADS1278 besitzt für jeden seiner acht Kanäle einen Powerdown-Pin, mit welchem der zugehörige Kanal deaktiviert werden kann. Weiterhin kann über die Eingänge FORMAT[2:0] das Ausgabeformat geändert werden. CLKDIV und MODE[1:0] stellen eine rudimentäre Option dar, den Takt einzustellen. In Abbildung 9 ist an den zweiten Port des FT2232H der MCP23S17 16Bit Portexpander angeschlossen.

Über diesen Portexpander ist die on-the-fly Softwarekonfiguration des ADS1278 möglich. Der ADS1278 besitzt verschiedene Prescaler. Durch diesen Wert wird das Verhältnis zwischen f_{clk} und f_{data} festgelegt. Der Prescaler kann auf 256, 512 beziehungsweise 2560 eingestellt werden. Durch Variation von f_{clk} kann weiterhin die Abtastfrequenz eingestellt werden. Jedoch darf f_{clk} maximal 37 MHz betragen und somit ergibt sich eine maximale Datenrate von 144531 SPS[2].

Der Spartan 3E besitzt jedoch keine eigene PLL-Einheit, sondern nur eine DCM-Einheit, wodurch die Frequenzen stark eingeschränkt sind. Die Anbindung einer externen PLL-Einheit ist deswegen sinnvoll. Als externer Baustein wäre zum Beispiel von SiLabs der Si570 einsetzbar, welcher über ein I^2C-Interface konfiguriert werden kann.

IV. Zusammenfassung und Ausblick

Ausgehend von der Problemstellung aus dem Forschungsgebiet Smart Metering wurde die Entwicklung einer Data Acquisition Card diskutiert. Hierbei wurde der Aufbau der Hardware dargestellt und die Problematik der Datenübertragung herausgearbeitet. Das bedeutet, dass im Rahmen der Messung keine Daten verloren gehen dürfen. Um dies zu realisieren, wurden zwei FIFO-Konzepte mit einem FPGA und DDR-SDRAM präsentiert.

Der wenig komplexe Aufbau des ersten Konzepts resultiert jedoch in der Einschränkung bei einer zukünftigen Erweiterung. Die Integration einer Ethernetschnittstelle wäre nur mit erheblichem Mehraufwand möglich. Das zweite Konzept hingegen ermöglicht nach einer höheren Anfangskomplexität eine einfachere Anbindung der Ethernetschnittstelle. Grundsätzlich ist die hier vorgestellte DAC Karte mit schnellem und großem Zwischenspeicher für eine Vielzahl unterschiedlicher Aufgaben einsetzbar. Dieses Konzept kann ebenso für einen Logikanalysator benutzt werden, bei welchem eine Komprimierung des Signals direkt in Hardware erfolgt.

Als Ausblick ist die komplette Entwicklung einer Datenerfassungsplattform mit entsprechender Software basierend auf Qt4 zu nennen. Durch den Einsatz von Qt4 und „libftdi“ ist die Software plattformunabhängig. Weiterhin ist die Integration in Matlab und LabVIEW vorgesehen. Gerade die Integration in Matlab wäre für einen zukünftigen Einsatz im Bereich der Lehre vorteilhaft.

[2] Samples per Second

Danksagung

Das Projekt „Smart Metering“ wird durch das Land Baden-Württemberg gefördert.

Literaturverzeichnis

[1] P. Bertoldi and B. Atanasiu, “Electricity consumption and efficiency trends in the enlarged European Union”, *Technical Report*, EUR 22753 EN-DG Joint Research Centre, Institute for Environment and Sustainability, 2007

[2] VDE, „Smart Energy 2020“ *VDE Verband der Elektrotechnik Elektronik Informationstechnik e.V.*, Germany, 2010.

[3] D. Benyoucef, T. Bier, P. Klein. „Planning of Energy Production and Management of Energy Resources with Smart-Meters“ *International Conference on Advances in Energy Engineering*, Beijing 2010. IEEE Proceedings pp. 170-173 ISBN 978-1-4244-7830-9.

[4] S. Mauch, „Non Intrusive Load Monitoring Data Acquisition Card“ (German), *Bachelor's Thesis*, Hochschule Furtwangen, 2010

Dirk Benyoucef erhielt den akademischen Grad des Dipl.-Ing. in Allgemeiner Elektrotechnik im Jahr 1998 von der Universität des Saarlandes und den Grad des Dr.-Ing. in Elektrotechnik im Jahr 2002 von der Universität des Saarlandes. Er ist Professor für digitale Signalverarbeitung und digitale Kommunikationstechnik an der Hochschule Furtwangen.

Steffen Mauch erhielt den akademischen Grad des Bachelor in Electrical Engineering im Jahr 2010 von der Hochschule Furtwangen und den Master in Mikrosystemtechnik im Jahr 2011 von der Hochschule Furtwangen. Er ist Doktorand an der TU Ilmenau.

True, unless disproven, or false, unless verified? Avoiding False Negatives in Functional Verification of Digital Circuits

Peer Johannsen

***Abstract*—False negatives in functional circuit verification are one of the major obstacles for acceptance of formal techniques and formal tools in digital circuit design. This article reviews the basic concepts of formal property checking and illustrates recent developments and improvements in the attempt of commercial tools to automate the handling of false negatives.**

***Index Terms*—False Negatives, Formal Property Checking, Functional Hardware Verification.**

I. Introduction

Functional errors are the number one cause of silicon re-spins. In digital circuit design, up to 70 percent of development time and resources are spent on functional verification, and between 20 to 40 percent of designs require as many as 3 spins before the product is released to the market, stressing the need for efficient verification techniques.

In simulation-based verification, corner case bugs are often missed due to its non-exhaustive nature. The size of the state space of a circuit grows exponentially with the number of input bits, making exhaustive simulation infeasible even on the fastest computers. No matter how many test benches are simulated, validating VHDL or Verilog implementations by means of simulation is inherently incomplete for all but the smallest designs. As a consequence, simulation can only falsify the claim that a circuit implementation meets a given specification, but it cannot proof its correctness beyond all doubt. With ever increasing circuit complexity, the percental portion of the state space which can effectively be covered by simulation even shrinks proportionally.

Formal verification techniques are based on principles of mathematics and will either compute a counterexample or will formally prove that an implementation is correct by exhaustively taking into account the complete combinatorial state space of the design (Fig. 1).

Peer Johannsen, peer.johannsen@hs-pforzheim.de, is a professor for Computer Science and Software Engineering at Pforzheim University, Tiefenbronner Straße 65, 75175 Pforzheim.

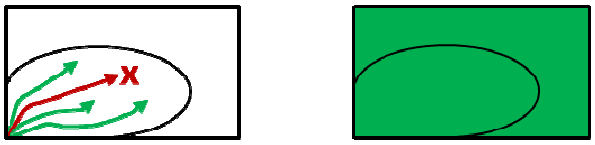


Figure 1: Simulation vs. Formal Verification

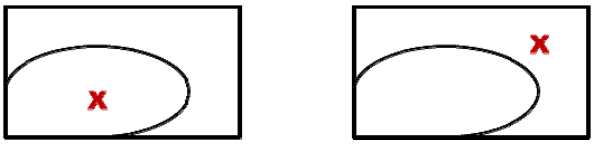


Figure 2: Reachable Failures vs. False Negatives

This is one of the main advantages of formal verification over simulation, but at the same time constitutes one of its weak spots. Since the whole combinatorial state space of a design is relevant in a formal proof, also those situations cause a proof to fail which violate implementation intent but are not sequentially reachable from the initial reset state of the design (Fig. 2).

If a circuit property holds true within the reachable state space of the design, but is false in some sequentially unreachable state, then a formal proof that checks the complete combinatorial behavior cannot succeed and will flag an unreachable situation as an error. In that case such a counterexample is called a *false negative*. Simulation can also produce false negatives if started from an unreachable state. Therefore, in practice, simulation test benches are usually restricted to start by triggering a circuit reset.

A fundamental problem is to determine whether a formal counterexample is a false negative or a situation that is reachable from reset and thus a true bug in the implementation. Exact computation of the reachable state space is computationally as hard as exhaustive simulation. Up to 50 percent of effort in verification projects is spent on the analysis of counterexamples and on using methodology to prevent false negatives from occurring in formal proofs. In the past, these techniques have lacked automation and have been applied manually by verification engineers, requiring a deep understanding of how formal proofs work. These obstacles have caused formal verification to have obtained a reputation as heavy weight mathematics and of being for experts only [4][5][9].

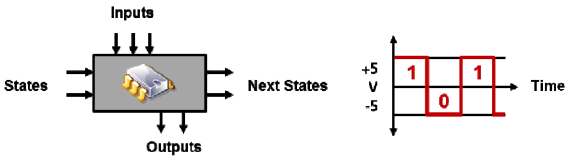


Figure 3: Functional Behavior of Digital Circuits

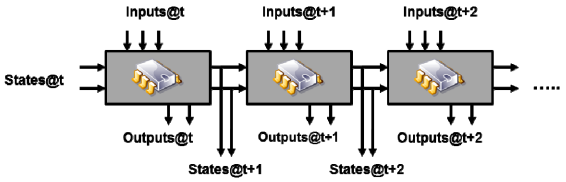


Figure 4: Sequential Circuit Behavior

But application of formal verification techniques and formal verification tools has become indispensable in the *Electronic Design Automation* (EDA) industry. Formal tools have seen a change in recent years and have matured from academic command line tools to commercial EDA solutions with graphical user interfaces and support for massive parallelization and network clustering. At the same time, standardization and acceptance of formal property languages, like e.g. *System Verilog Assertions* (SVA), and improvements in ease of use of formal tools have made formal verification accessible also for non-experts. Methodology for dealing with false negatives has been automated and integrated in the tools, which are now targeted to be used by the hardware designer herself, and as early in the development process as possible.

In the following sections, the underlying concepts of formal hardware verification and of formal property checking of digital circuits are reviewed. Firstly, an outline is given of how formal techniques work and how they can verify the complete state space of a design without having to explicitly simulate each possible input combination of the circuit. It is explained why false negatives can occur as results of formal proofs. Secondly, an overview of basic proof methodology for preventing unreachable counterexamples is provided and illustrated by small examples. Finally, a sketch is given how these techniques can be automated and used in formal tools to provide stronger verification results.

II. Formal Property Checking

The problem of verifying that digital circuit behavior is in accordance with the intention of the circuit designer is a problem of checking whether design behavior meets specific properties [2]. The combinatorial behavior of digital circuits can formally be described by Boolean functions mapping input and register values to Boolean output and next-state values (Fig 3).

Likewise, the sequential circuit behavior is described by the variation of signal values over a period of time. Time is modeled as a linear sequence of abstract and discrete time points t, $t+1$, $t+2$, …, corresponding to clock cycles of the design. The sequential circuit model is then obtained by unrolling the combinatorial model (Fig. 4).

For each new time point, all variables representing circuit signals are copied, while the new state variables are substituted by the next state functions of the previous time point. Behavior of the sequential circuit model solely depends on the initial values of the state variables in the very first time frame and on the sequence of input values over time.

A *Bounded Temporal Property* defines an intended behavior of a design within a finite bounded interval $[t, t+n]$ of time. Given such a property and a circuit implementation in a hardware description language like VHDL or Verilog, the so called *Bounded Model Checking Problem* asks, if the property holds true for all possible stimuli traces of constant length n, starting at any arbitrary point of time t. Bounded temporal properties can be specified by using formal property languages or standard assertion languages. In the past, most tools for formal verification came along with proprietary property languages, requiring a verification engineer to learn a new language with each new tool. Today, standardized languages like SVA, PSL (*Property Specification Language*) and OVL (*Open Verification Library*) are available and are supported by the majority of commercial verification and simulation tools. In addition to that, hardware description languages also provide means of specifying inline properties (e.g. *assert(...) report* statements in VHDL) which are embedded within the source code and ignored by synthesis tools, but can be analyzed by simulation and verification tools.

An abstract formal property usually consists of a set of assumptions implying a set of commitments. For the sake of simplicity, the following pseudo code like notation is used for the examples given in this article.

```
property example
{
  assume:
    at t:   assumptions_0;
    at t+1: assumptions_1;
    at t+2: assumptions_2;
    ...
    at t+n: assumptions_n;

  assert:
    at t:   commitments_0;
    at t+1: commitments_1;
    at t+2: commitments_2;
    ...
    at t+m: commitments_m;
}
```

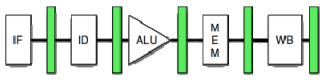


Figure 5: Five Stage Processor Pipeline

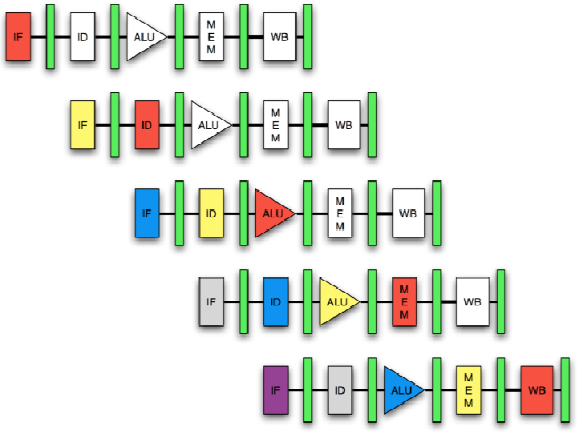


Figure 6: Pipeline Fill Level

Let us consider the example of a five stage processor pipeline (Fig. 6). We will assume that the specification requires that an implementation must have the following property:

"If a multiplication instruction is fetched, then the product of the values of the two operand registers must be present in the destination register exactly after the next four clock cycles."

The functionality described above depends on the input values of the two operand registers and on the other instructions currently present in the pipeline (Fig. 6). A simulator test bench therefore needs to specify definite signal values for all input signals which are relevant for the fill level of the pipeline, otherwise it cannot be simulated. A test bench thus checks very specific circuit behavior. In the VHDL example given below, the test bench is restricted to a single trace checking a multiplication of operand values 23 and 19, followed by four subsequent NOP instructions.

```
instr <= MUL_AB; reg_A = 23; reg_B = 19;
wait for 125ms;

instr <= NOP; reg_A = 0; reg_B = 0;
wait for 125ms;

instr <= NOP; reg_A = 0; reg_B = 0;
wait for 125ms;

instr <= NOP; reg_A = 0; reg_B = 0;
wait for 125ms;

instr <= NOP; reg_A = 0; reg_B = 0;
assert(reg_C = 437) report "Error!";
```

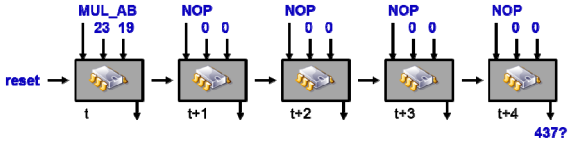


Figure 7: Test Bench Simulation of a Single Trace

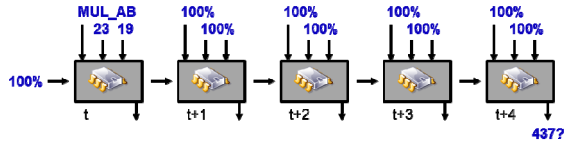


Figure 8: Exhaustive Verification

The property either holds true for the given trace, or results in a failure of the *assert(...)* statement which is used to control the simulator output. A failure that is detected will always be a reachable failure, because stimuli traces for test bench simulation usually start by triggering a reset of the design (Fig. 7). However, detection of failures heavily depends on the specific selection of stimuli traces and on the specific values chosen for the input signals of the circuit. Stimuli traces for test benches have to be generated either by hand according to the experience of the designer, in a randomized fashion, or by employing special test bench tools, but even large sets of stimuli traces can cover only a tiny portion of the exponentially large number of possible traces.

In contrast to that, in formal properties only the relevant signal values need to be explicitly specified.

```
property multiply_23_19
{
  assume:
    at t:   instr = MUL_AB;
    at t:   reg_A = 23;
    at t:   reg_B = 19;

  assert:
    at t+4: reg_C = 437;
}
```

Formal verification techniques will take into account all possible combinations of starting states and sequences of input values and thus verify the complete combinatorial state space of the design. All possible traces matching the assumptions of the property are exhaustively verified, allowing all possible combinations of instructions and register values in the other stages of the pipeline (Fig. 8).

Formal properties also allow signal values to be treated as variables which can be referenced across time frames. Therefore, there is no need to restrict the relevant register inputs to dedicated values like 23 and 19 only. Thus, the multiplication property shown above can be further generalized (Fig. 9).

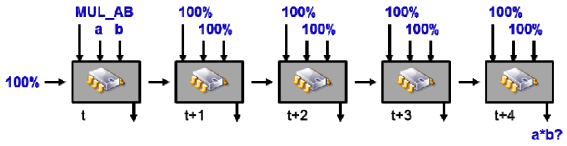


Figure 9: Formal Property Checking

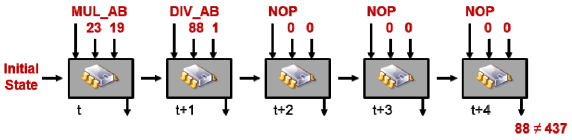


Figure 10: Counterexample Trace

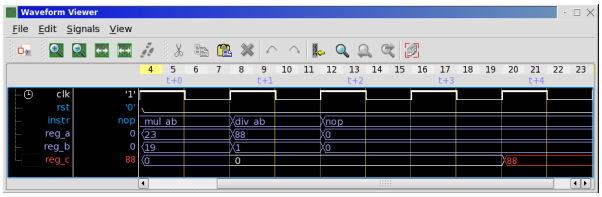


Figure 11: Counterexample as Waveform

The modified property shown below now checks *all* possible combinations of values for the two operand registers *in one go*.

```
property multiply
{
  variables: a, b;

  assume:
    at t:   instr = MUL_AB;
    at t:   a = reg_A;
    at t:   b = reg_B;

  assert:
    at t+4: reg_C = a * b;
}
```

A formal tool will either determine that the property exhaustively holds for a given implementation, or will automatically compute a counterexample consisting of a trace of signal stimuli which cause a violation of the property. Let us assume that a faulty implementation of the division instruction DIV_AB causes the quotient to be written to reg_C after three instead of four clock cycles, but only in case of the divisor being 1. The multiplication property then fails, if a MUL_AB instruction is immediately followed by a division by 1 (Fig. 10). In contrast to the before mentioned sample test bench, formal techniques will detect this failure and compute a respective counterexample trace. Such a sequence of stimuli values can be displayed in waveform viewers for debugging purposes or can itself be used as input for simulation tools for further analysis of the root cause of the failure in the HDL implementation (Fig. 11).

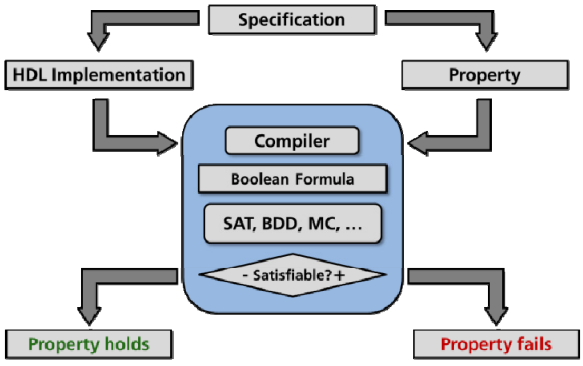


Figure 12: Property Checking Flow

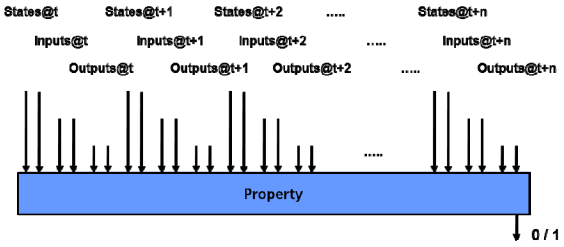


Figure 13: Property Relation

Formal property checking is intended to be used in early design stages. In order to determine whether a property holds or fails, the HDL source code of a circuit design and the formal property are read by a frontend compiler which then generates a Boolean formula which is satisfiable if and only if the property does not hold for the given implementation (Fig. 12). A Boolean formula is called *satisfiable*, if there exists an assignment of Boolean values to all variables, such that the formula evaluates to true. If such an assignment, also called a *satisfying solution*, does not exist, then the formula is called *unsatisfiable*.

The process of trying to find a satisfying solution for a given Boolean formula is called *satisfiability checking* or *SAT checking* and is an NP-complete problem. Other techniques for determining Boolean satisfiability for example are *Model Checking* or BDD (*Binary Decision Diagrams*) based approaches [3].

Technically, a formal property of length *n* is a mathematical relation over sequences of fixed length *n* of signal values, indicating whether or not a specific sequence of input values, state values and output values fulfills a given specification (Fig. 13). The sequential circuit model is computed by unrolling the combinatorial model for *n* time frames, and by combining the sequential model with the negation of the formal property relation, the so called *Bounded Model Checking Problem* (BMCP) is obtained (Fig. 14).

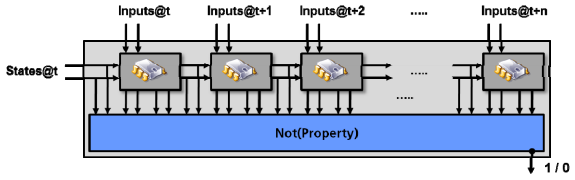


Figure 14: Bounded Model Checking Problem

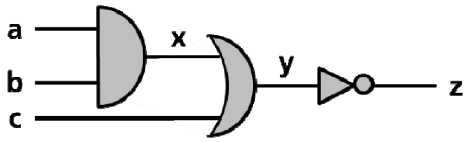


Figure 15: Netlist

The output of the Bounded Model Checking Problem is 0 for all starting states and input sequences for which the functional circuit behavior is in accordance with the property. The output of the Bounded Model Checking Problem is 1 for starting states and input sequences for which the circuit implementation shows a violation of the property [1]. In order to determine whether or not such a value assignment exists, the Bounded Model Checking Problem is first compiled into a netlist, i.e. a circuit-like representation constructed of Boolean logic gates (Fig. 15).

A netlist can efficiently be translated into a Boolean formula in *conjunctive normal form* (CNF). By introducing intermediate variables x and y, the characteristic function $B(a,b,c,x,y,z)$ of the netlist shown above can be written as:

$$(x = a \cdot b) \cdot (y = x + c) \cdot (z = \neg y)$$

Each of the three equations representing the atomic Boolean gates AND, OR and NOT, can itself be written in CNF:

1. $(x = a \cdot b) \leftrightarrow (x + \neg a + \neg b) \cdot (\neg x + a) \cdot (\neg x + b)$
2. $(y = x + c) \leftrightarrow (y + \neg x) \cdot (y + \neg c) \cdot (\neg y + x + c)$
3. $(z = \neg y) \leftrightarrow (z + y) \cdot (\neg z + \neg y)$

The resulting Boolean formula for $B(a,b,c,x,y,z)$ is in CNF, and it is satisfiable if and only if the Bounded Model Checking Problem is satisfiable. The Boolean variables directly correspond to binary signals of the HDL implementation and to internal signals of the netlist. The size of the formula is linear in the number of logic gates and thus linear in the size of the circuit. By symbolic traversal of the netlist, a procedure of polynomial complexity can be implemented which efficiently converts a circuit representation into a Boolean formula in CNF. Note that there exists no efficient (meaning non-exponential) procedure to construct a Boolean circuit representation in *disjunctive normal form* (DNF) unless P = NP.

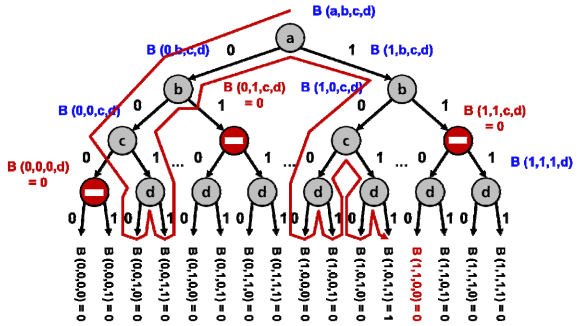


Figure 16: Satisfiability Checking

When trying to find a satisfying solution for such a Boolean formula, the fact is exploited that a CNF is true if and only if each conjunctive element is true. In other words, as soon as just one single conjunctive element evaluates to false for a given partial assignment, then this assignment cannot be extended to a satisfying solution of the whole formula. The basic strategy used in SAT checking algorithms performs a recursive search on the tree of all possible value assignments. After each local variable assignment, the procedure tries to determine the resulting partial Boolean function. By *symbolic* inspection of the CNF, *conflicts* are identified. Conflicts either result from Boolean implications of the current (partial) variable assignment which require the same Boolean variable to be true and false at the same time, or from CNF clauses evaluating to false. If a conflict is detected, then the partial Boolean function of the CNF is constantly false. In that case the algorithm backtracks, reverts the last variable assignment and continues with trying the next possible value assignment [6][7]. Thus, the overall portion of the search space which is effectively visited can significantly be reduced. Note that the size of this portion also depends on the variable order which is chosen and on whether the value 0 or the value 1 is tried first (Fig. 16).

SAT checking procedures essentially are heuristics, which have exponential worst case complexity. However, digital circuit designs have certain logical structures, and so have the Boolean formulae occurring in Bounded Model Checking Problems. Formal verification tools compete in finding the best search heuristics by exploiting structural circuit information in order to determine the best variable order for detecting conflicts as early as possible. Thus, even very large satisfiability checks can effectively be handled, ranging up to, say, 50.000 variables and 10 million CNF clauses [8][10].

The great strength of formal SAT checking, namely taking into account *all* possible value assignments of the Bounded Model Checking Problem, at the same time has its drawbacks. A SAT checker will try to assign any possible value to the initial starting state of

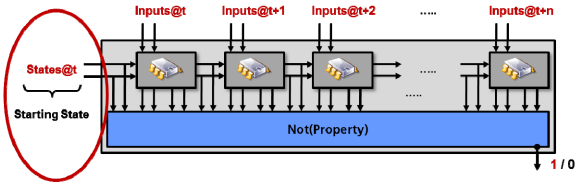


Figure 17: False Negative

the BMCP and then check if a trace from here on exists which leads to a violation of the property. If the combination of values assigned to the starting state of a counterexample is not reachable from reset, then such a counterexample is a false negative (Fig. 17). In classic property checking, in fact each failure is an uncertain failure, unless analysis yields that the starting state of the counterexample indeed lies within the reachable state space of the design. Exact computation of the reachable state space itself is an NP complete problem and thus usually not feasible in practice for complex designs. Consequently, property checking requires additional technology or methodology for dealing with false negatives.

Reachability analysis of counterexamples, when done manually, is a tedious and time consuming task. If a counterexample is analyzed to be reachable, then the underlying bug can be fixed, and the property can be checked again. If a counterexample is analyzed to be unreachable, then further proof methodology can be tried to exclude this counterexample in future checks of the property (Fig. 18).

A fundamental problem occurs, if neither is possible, or if repeated checks of the property continue to result in false negatives. In that case the question arises, how long should this procedure be iterated? And how should this property be treated with regard to the verification of the circuit? Should it be considered to hold until disproven, or should it be regarded as false until successfully verified?

III. Avoiding False Negatives

Methodology for dealing with false negatives consists of strengthening the formal proof by rewriting the property. This means that the same circuit property is phrased in a different way which is more beneficial for a formal proof to succeed and which is less prone to produce unreachable counterexamples. In the following sections, some examples of standard methods of reachability methodology will be given, illustrating techniques like *prefix unrolling*, *k-induction*, *bounded proofs from reset*, *conjunctive proof reasoning* and *invariant computation*.

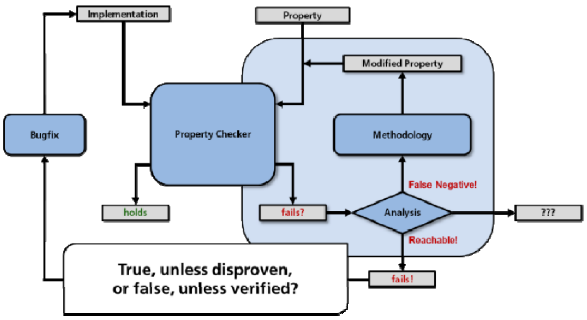


Figure 18: Formal Verification Flow

A. Prefix Unrolling

The Verilog example given below implements a simple state machine for an instruction which is processed by the pipeline shown in the previous section.

```
// ======================================
// pipeline state machine
// ======================================

`define fetch  3'b000
`define decode 3'b001
`define alu    3'b010
`define memory 3'b011
`define write  3'b100

module pipeline_state(
  input clock, reset,
  output reg[2:0] state);

  always @(posedge clock or negedge reset)
    if (reset == 1'b0)
      state <= `fetch;
    else
      case (state)
        `fetch  : state <= `decode;
        `decode : state <= `alu;
        `alu    : state <= `memory;
        `memory : state <= `write;
        `write  : state <= `fetch;
        default : state <= `fetch;
      endcase;
endmodule
```

In order to model the five distinct stages of the pipeline, a 3 bit state variable is needed. Let us assume that the following simple property is to be verified which checks that the state machine is always in one of the five predefined modes.

```
property state_machine
{
  assert:
    at t: (state == fetch)  or
          (state == decode) or
          (state == alu)    or
          (state == memory) or
          (state == write);
}
```

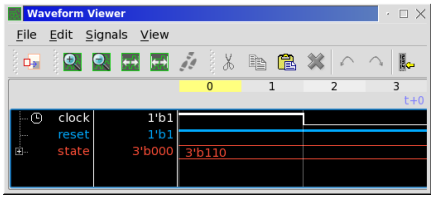


Figure 19: Unreachable Counterexample

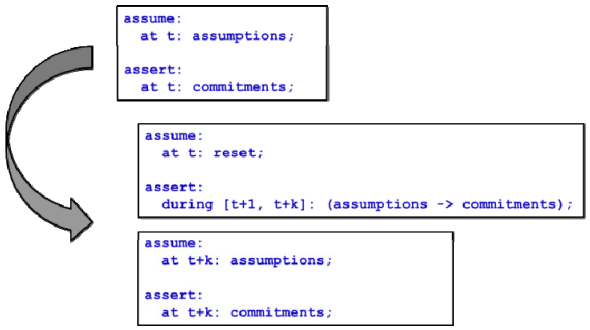


Figure 20: Prefix Unrolling

When started from reset, the circuit will show exactly the desired behavior which is described by the property, i.e. the property holds for all states which are sequentially reachable from reset. However, when formally checked, the property is indicated to fail and a counterexample is returned, in which the value of *state* is set to 3'b110 (Fig. 19). A formal tool takes into account the whole combinatorial state space, i.e. all the eight possible bit values of the state variable, and thus concludes a possible violation of the property. However, such a state will never be reached, due to the initialization of the design and due to the construction of the case-statement. The counterexample shown above is a false negative, but a simple trick can be used to exclude it from occurring. In order to do so, a prefix of one timeframe is added to the original property:

```
property prefix_1
{
  assert:
    at t+1: (state == fetch)  or
            (state == decode) or
            (state == alu)    or
            (state == memory) or
            (state == write);
}
```

Semantically, *t* is a free (time) variable. The original property checks that at each time point *t* the state machine is in one of the five predefined states. The modified property checks that for each time point *t*, the state machine will be in one of the predefined states *in the next cycle*. If the modified property holds, and if the original property holds in the very first cycle after pulling the reset, then it follows that the original property also holds within the complete reachable state space.

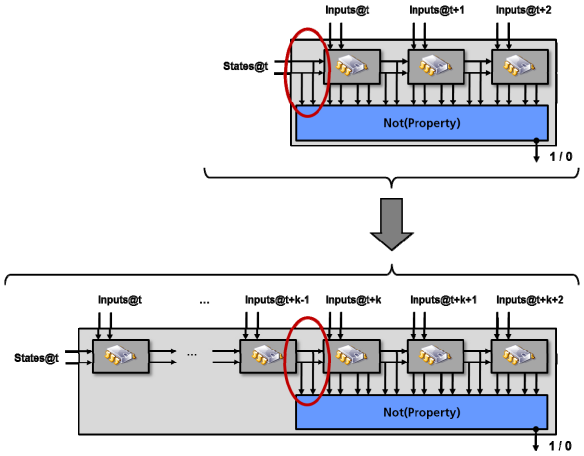


Figure 21: Restriction of Starting States

This technique is called *prefix unrolling* (Fig. 20). In general, instead of the original property, two new properties are checked. If the original property holds, then so will the two modified ones. Those, however, will not be able to produce the unreachable counterexample shown above. The offset of *k* time frames used as prefix in the modified properties can be arbitrarily enlarged. The effect is that the potential set of starting states of a counterexample is restricted according to the functional behavior of the circuit design (Fig. 21). The larger the chosen prefix depth *k* is, the better the reachable state space of the design is approximated. Put differently, the strategy works whenever a design must end up in a reachable state after at most *k* steps, even when starting in an unreachable state. However, there are examples, for which prefix unrolling does not help at all. Consider the following modification of the pipeline state machine.

```
// ======================================
// pipeline state machine
// ======================================

`define fetch  3'b000
`define decode 3'b001
`define alu    3'b010
`define memory 3'b011
`define write  3'b100

module pipeline_state(
  input clock, reset, next,
  output reg[2:0] state);

  always @(posedge clock or negedge reset)
    if (reset == 1'b0)
      state <= `fetch;
    else
      if (next == 1'b1)   // modification!
        case (state)
          `fetch  : state <= `decode;
          `decode : state <= `alu;
          `alu    : state <= `memory;
          `memory : state <= `write;
          `write  : state <= `fetch;
          default : state <= `fetch;
        endcase;
endmodule
```

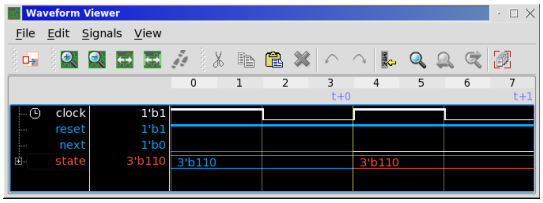


Figure 22: Unreachable Counterexample for Prefix Depth 2

The state machine will proceed to the next state only if a specific input signal is set. The original property will produce a false negative, and so will any prefix enhanced property. If the switch is not set, then the circuit can stay in the unreachable state space for an indefinite period of time. In this case, prefix unrolling cannot prevent false negatives (Fig. 22).

B. K-Induction

Induction is a strong technique known from mathematical proofs, and the underlying principle can readily be applied to formal property checking as well. Instead of the original property, two modified separate proofs are executed: a *base case*, checking if the property holds for a fixed number of *k* cycles after reset, and a *step case*, which checks that the property will hold for one more step under the assumption that it holds for the previous *k* steps (Fig. 23). If both base case and step case can successfully be proven to hold, then it follows that the original property holds in the complete reachable state space of the design. A simple 1-step induction will be able to successfully verify the state machine property for the second implementation. The base case property is the same as for 1-step prefix unrolling and holds trivially.

```
property base_case
{
  assume:
    at t: reset == 1'b0;

  assert:
    at t+1: (state == fetch)  or
             (state == decode) or
             (state == alu)    or
             (state == memory) or
             (state == write);
}

property step_case
{
  assume:
    at t: (state == fetch)  or
           (state == decode) or
           (state == alu)    or
           (state == memory) or
           (state == write);

  assert:
    at t+1: (state == fetch)  or
             (state == decode) or
             (state == alu)    or
             (state == memory) or
             (state == write);
}
```

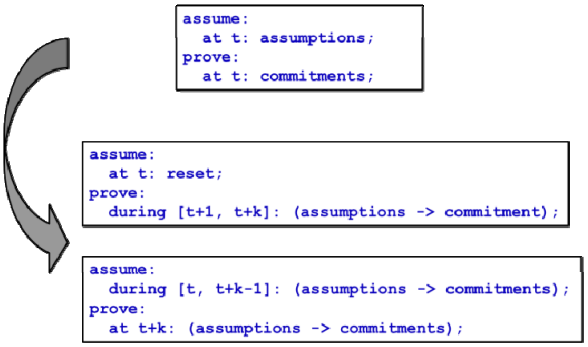


Figure 23: *K*-Induction

Since the step case assumes the original property to be true in the first time frame *t*, a counterexample would have to start in a state for which this assumption holds and from there would have to proceed to a state in which the original property is violated. Such sequential behavior is not possible for the given implementation of the design, and therefore no such counterexample exists and a formal proof of the step case property succeeds.

However, 1-step induction will not succeed for the following example, which implements a small ring buffer from which pipeline instructions are fetched. The buffer is modeled as an array of size 60, and an incremental index counter is used which continuously counts up and accesses the ring buffer in each clock cycle. In the implementation given below, the four topmost entries of the array are not used and the index counter repeatedly counts from 1 to 56.

```
// ======================================
// instruction fetch from buffer
// ======================================

`define max_index 6'b111000 // 56

module fetch_from_ring_buffer(
  input clock, reset,
  output [4:0] instr);

  reg [4:0] buffer[60:1];
  reg [5:0] index;

  assign instr = buffer[index];

  always @(posedge clock or negedge reset)
    if (reset == 1'b0)
      index <= 6'b1;
    else
      if (index == `max_index)
        index <= 6'b1;
      else
        index <= index + 6'b1;

endmodule
```

A safe implementation requires that arrays must only be accessed within index bounds, corresponding to the following formal property.

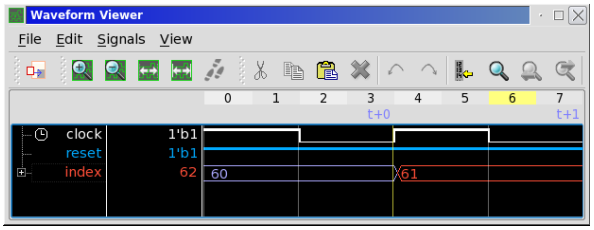


Figure 24: Unreachable Counterexample for 1-Step Induction

```
property array_access
{
  assert:
    at t: (index >= 1) and (index <= 60);
}
```

Rewriting the property for 1-step induction results in the following two properties:

```
property base_case
{
  assume:
    at t: (reset = 1'b0);

  assert:
    at t+1: (index >= 1) and (index <= 60);
}

property step_case
{
  assume:
    at t: (index >= 1) and (index <= 60);

  assert:
    at t+1: (index >= 1) and (index <= 60);
}
```

The base case property of the induction holds. The inductive step will assume the original property to hold at an arbitrary point of time *t*, and then check if it also holds in the next clock cycle. Such a proof does not succeed and results in a false negative (Fig. 24). An index value of 60 is sequentially unreachable, but satisfies the assumption of the step case and will be considered as a starting state of the Bounded Model Checking Problem which is constructed for the step case property. The sequential behavior of the circuit will increase the value of the index counter to 61, falsifying the commitment of the step case. The problem can be solved by increasing the radius of the inductive proof to a value of five.

```
property base_case
{
  assume:
    at t: (reset = 1'bo);

  assert:
    during [t+1,t+5]: (index >= 1) and
                      (index <= 60);
}
```

The original property holds for five cycles after reset, so the formal proof of the base case succeeds.

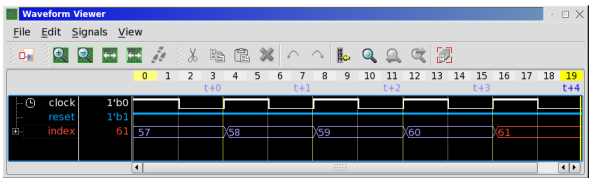


Figure 25: Unreachable Counterexample for 4-Step Induction

```
property step_case
{
  assume:
    during [t,t+4]: (index >= 1) and
                    (index <= 60);
  assert:
    at t+5: (index >= 1) and
            (index <= 60);
}
```

If assumed to be true for five clock cycles, the original property also holds in the sixth cycle, because being within range 1 to 60 for five cycles can only be true when starting with a value of at most 56, causing the index counter to wrap around in the subsequent step. In other words, there is no trace, in which the property can be true for five steps and then fail in step six. Therefore, the formal proof of the step case also succeeds.

Success of *k*-induction depends on the value of the parameter *k*. In the previous example, a shorter induction does not work. If a four-step induction is used instead of five steps, then the result would still be an unreachable counterexample (Fig. 25). On the one hand, higher values for *k* will yield stronger formal proofs, excluding more unreachable behavior, but on the other hand the Boolean satisfiability problems which are constructed from the resulting Bounded Model Checking Problems of the base case and the step case properties will be more and more complex and will require exponentially growing computational resources with linear increasing *k*.

C. Bounded Proofs from Reset

If the base case property of an inductive proof fails, then in fact a reachable counterexample has been found, since each counterexample of the base case must start in the reset state of the circuit.

Checking the base case of an *n*-induction corresponds to an exhaustive proof for a complete limited radius of *n* steps from reset. Formally, all possible sequences of length *n* starting in the reset state of the design are checked. This technique conceptually corresponds to exhaustively simulating all possible stimuli traces of a fixed length *n* while checking the original property in each time frame of each trace. The concept can be tried on its own and is called a *bounded proof from reset* (Fig. 26).

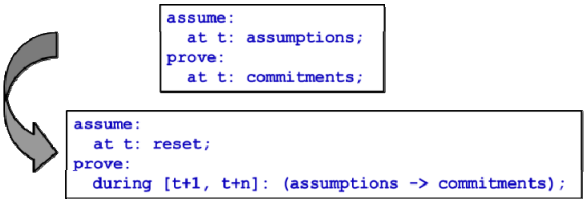


Figure 26: Bounded Proofs from Reset

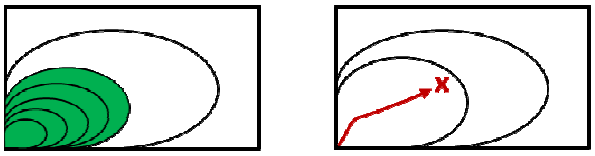


Figure 27: Bounded Correctness and Reachable Failure from Reset

With increasing *n*, this technique will reach computational limits and resource limits, depending on the complexity of the property and on the complexity of the circuit. Like simulation, bounded proofs from reset cannot verify the complete design behavior but only behavior within a bounded radius. The big advantage is that the radius is exhaustively covered, and if the formal proof fails, then a *reachable* counterexample is obtained, indicating a true bug in the implementation (Fig. 27).

D. Constraints and Invariants

If a counterexample has been detected to be a false negative, a simple approach to prevent this counterexample from occurring again in future proofs is to add *constraints* to the assumption part of the property which exclude exactly the combination of signal values which constitute the (unreachable) starting state of the counterexample. The modified property is then checked again and cannot end up in the same false negative anymore. However, the set of unreachable states usually is large, and the modified property will most likely end up in a counterexample with a different unreachable starting state. Repetition of the described procedure is tedious and will take too long to be feasible in practice. A better approach is to exclude whole sets of unreachable states instead of excluding just one specific starting state of a counterexample. This can be achieved by computing *invariants* which hold true in the complete reachable state space of the design. Such invariants can safely be added as additional assumptions to the property. The attempt seems to be recursive in itself, because invariants also have to be formally verified before they can be assumed. However, very often simple invariants can easily be computed using special invariant computation techniques. Furthermore, when verifying a whole set of properties, all properties which have already successfully been proven can be used as additional assumptions in subsequent formal proofs of other properties.

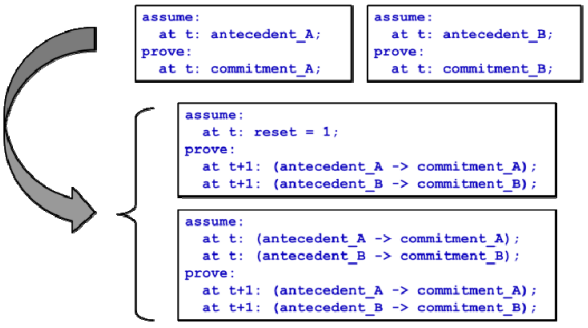


Figure 28: Conjunctive Proof Reasoning

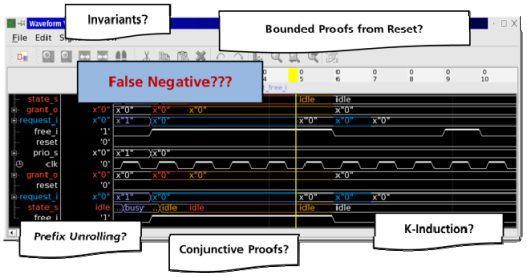


Figure 29: Analyzing Counterexamples

E. Conjunctive Proof Reasoning

A similar possibility to strengthen inductive proofs and to exclude unreachable starting states is *conjunctive proof reasoning*. Let us assume that two properties *A* and *B* are to be checked which both fail to be proven on their own. Instead of checking both properties separately, the logical conjunction can be tried. In the inductive step, both A and B are assumed to be true at the same time, thus excluding more unreachable starting states than it is the case when induction is tried separately for each property (Fig. 28). If proving the conjunction succeeds, then it follows that *both* properties hold. The approach is promising whenever there exists a logical dependency between both properties. However, if the conjunction fails, then no definite result can be concluded since it can still be the case that one of the two properties is valid and the other is not.

F. And now?

The list of methods which have been illustrated here is not complete and there are numerous other techniques for avoiding unreachable counterexamples. Analyzing reachability of a complex counterexample is a difficult task (Fig. 29), and rewriting a long and complex property for, say, *k*-induction, requires detailed knowledge of formal property languages. When applied manually, the major drawback of these techniques is that they also require a deeper understanding of how formal proofs work. Strategy and parameters have to be chosen according to experience.

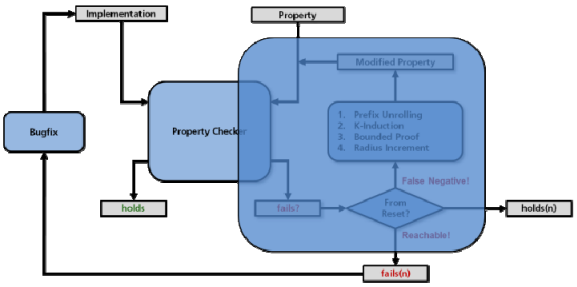


Figure 30: Improved Tool Flow

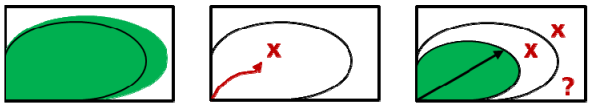


Figure 31: Proofs Results of type a), b) and c)

These are the main reasons why formal verification long since has been (and commonly still is) considered as an expert technology for experts and verification engineers only which is not well suited for application by the hardware designer herself.

IV. Automation of Reachability Methodology

Being able to deal with false negatives is crucial for a broader application and a broader acceptance of formal verification techniques (and thus also for the sales figures of formal tools). Development of formal tools has seen significant improvements in the recent past. Reachability technology has been integrated into formal tools in a way that allows for automation. Former property checkers provided binary answers only: "property holds" (certain answer) or "property fails" (uncertain answer, possibly a false negative). Nowadays formal tools internally use improved property checking technology combined with automated proof methodology to provide as much information about the validity of a formal property as could be found out (Fig. 30). In general, the results can be grouped into three different categories (Fig. 31):

a) A property is proven to hold exhaustively in a superset of the reachable state space (unbounded correctness).
b) A property is falsified by providing a reachable counterexample trace of a finite length of *n* clock cycles, starting in the reset state of the design (a certain failure of depth *n*).
c) A property is proven to hold exhaustively within a radius of *n* clock cycles from reset, but a counterexample is found, which is either unreachable, or occurs beyond a depth of *n* clock cycles from reset (bounded correctness for depth *n*).

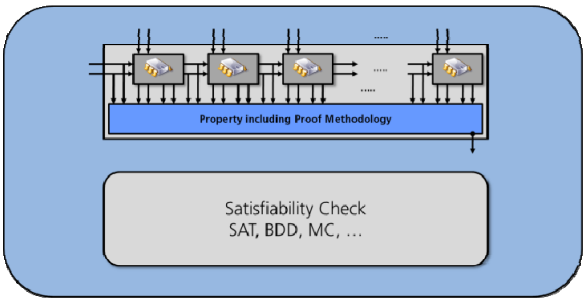


Figure 32: Opaqueness of the BMC in a Single Property Check

Note that simulation can only provide answers of the second type, and furthermore note that in case of bounded correctness still a counterexample is available which might be a false negative or a true bug, but which can provide valuable debugging information to the designer.

The abovementioned categories of results can be obtained by repeatedly applying the methodology which has been illustrated in the previous section, while successively incrementing the depth of the inductive proofs and the bounded proofs.

Manual application is tedious and inefficient. The property checker starts from scratch with a new compilation for each modified property and for each new depth for prefix unrolling, induction and bounded proofs from reset. Each time, the corresponding bounded model checking problem is newly generated, although the logical components of the proof problems are very much alike. The resulting Boolean formulas are intransparent, in the sense that a SAT checker does not know which part of the Boolean satisfiability problem originates from the original property, which part originates from proof methodology, and which part originates from unrolling the circuit implementation (Fig. 32).

The process can be automated by transferring the application of reachability methodology from the property level to the level of the bounded model checking problems. Proof methodology is kept separated from the property which is to be verified. Methodology is not applied on the level of the properties, but successively on the level of the Bounded Model Checking Problems. Instead of modified properties, the inputs to the decision procedure here are the functional model of the circuit and the property only. The Boolean problem which represents the satisfiability check then is incrementally augmented to contain all the parts which are necessary to perform inductive proofs and bounded proofs from reset. The radius of the proofs is increased in an internal loop, until a definite result is determined, a maximum radius is reached or resource limits are met.

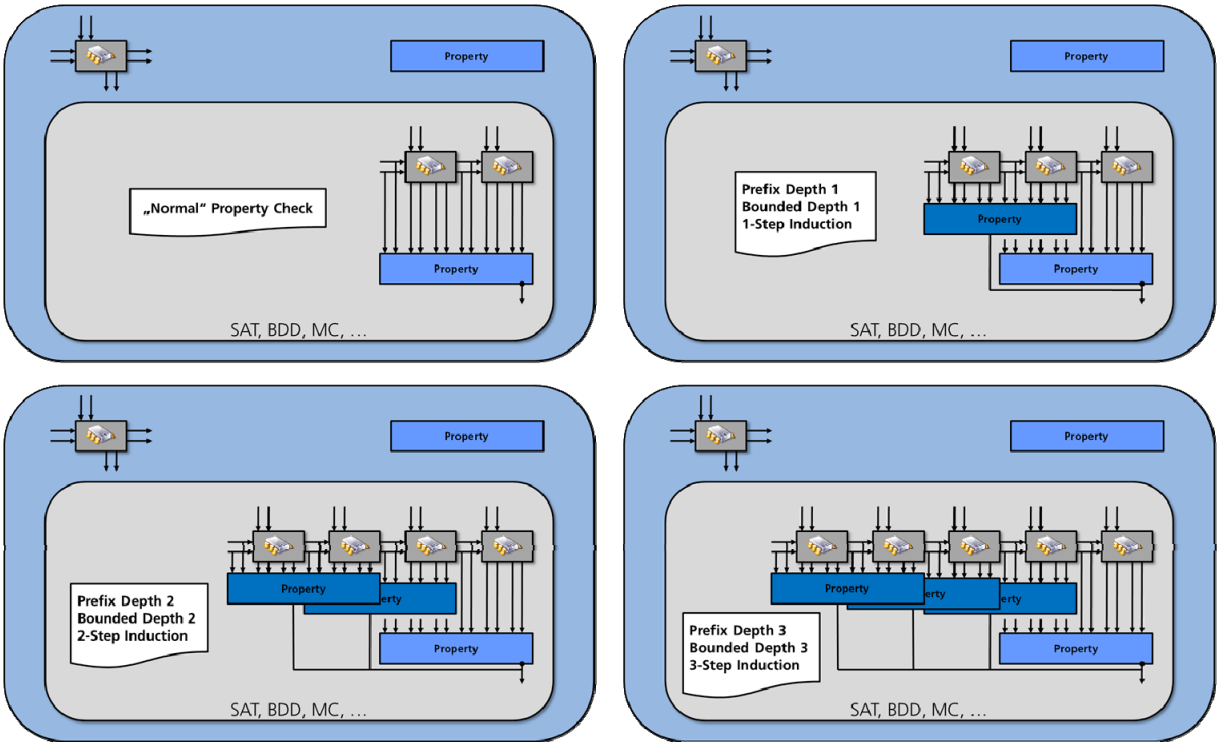


Figure 33: Automated Application of Reachability Methodology and Successive Construction of the Bounded Model Checking Problems

The sequence of the models constructed for a property of length 2 is shown above (Fig. 33). In each step, all information is present to perform the following sequence of checks:

1. A "regular" property check is performed. If the check holds, then a definite result is reached, and the procedure terminates. If the check fails (uncertain), then an additional prefix frame is unrolled.
2. The counterexample is extended by an initial activation of the circuit reset, and the extended sequence is simulated on the enlarged model. If simulation detects a failure, then the counterexample in fact is a reachable counterexample (certain failure) and the procedure terminates.
3. Otherwise, an exhaustive bounded check from reset is done on the prefix enlarged model. If this check fails, then a new and reachable counterexample has been found, and the procedure terminates. If the check holds, then in fact the inductive base case for the current depth has been shown to hold.
4. A copy of the property is inserted in the first frame of the model and used as assumption to check the inductive step case. If this check holds, then a definite result is reached, and the procedure terminates. Otherwise, an additional prefix frame is unrolled, and the procedure continues with step 2.

The process is interrupted if a user defined maximum depth is reached or resource limits or time limits are exceeded. In that case, only bounded correctness of the currently reached depth can be asserted.

Besides automation, the benefits of implementing such a procedure on the level of the satisfiability checker are manifold: incremental construction of the Bounded Model Checking Problems saves compilation time (HDL to netlist), and partial knowledge (Boolean implications) which is learned while traversing the search space of the decision trees can be reused in future steps of the procedure.

V. Experimental Results

With reachability methodology automated and integrated into formal tools, a significant increase in productivity in the verification flow could be observed. Feedback from verification engineers of two major German chip manufacturers reported up to 25 percent less counterexamples which required closer analysis (certain failures instead of uncertain failures). In the attempts of completing formal proofs, i.e. trying to turn an uncertain failure into formally proven correctness, up to 50 percent of time could be saved due to tool automation.

In another experiment conducted at Pforzheim University, a test suite of 30 circuit designs was subject to formal verification. The designs were taken from the Open Cores Test Suite (www.opencores.org), a collection of more than 500 industrial designs, ranging from CPU processors and memory controllers to full ASICs. The source code of the designs is available under the GNU Lesser General Public License.

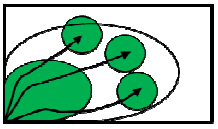


Figure 34: Combining Simulation and Formal Verification

A set of automatically generated standard assertions was checked with full tool capacity enabled, and the results were compared to those obtained with all reachability technology switched off. The outcome was approximately 50 percent less uncertain checks (meaning formally proven assertions instead of former false negatives, and reachable counterexamples instead of former uncertain failures).

VI. Conclusion

Formal verification technology and formal tools have seen significant improvements in usability and quality of results in the recent years. Specifying formal properties is not any harder than writing test benches anymore, but the benefit is exhaustive coverage without having to care for stimuli values which are not explicitly stated in the property. Furthermore, with standard assertion languages like SVA, PSL and OVL, formal properties can directly be embedded within the HDL source code of the design.

The issue of having to deal with false negatives, one of the major drawbacks when comparing formal techniques to simulation, has been addressed, and the ability of commercial tools to deal with false negatives has become the main differentiator between vendor solutions. With increasing computational power of today's computers and with usage of massive parallelization and compute farms, runtime is not the primary bottleneck anymore. When having to choose, a chip manufacturer will prefer a tool that reports less uncertain results (less false negatives), even if the same tool possibly needs more resources for computing the definite results.

The results of formal tools are much stronger than the results of test bench simulation. Yet, formal techniques and simulation do not contradict each other, they should be used in combination and complement each other. Wherever formal tools reach their limits, simulation can be used to further explore the state space. Similarly, in hybrid verification approaches stimuli traces constitute reachable starting states for formal verification. Thus, more and more portions of the state space can exhaustively be covered.

Yet, the problem of false negatives is inherent to formal verification and false negatives can still occur. The question posed in the title of this paper, how to handle uncertain failures, still remains, but their likeliness to occur has significantly decreased. At the same time, the limits of how deep formal tools can explore design behavior have clearly been pushed and the degree of automation in formal verification flows has increased by orders of magnitude.

References

[1] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, Y. Zhu, "Symbolic Model Checking using SAT Procedures instead of BDDs", *Proc. DAC*, 1999, pp. 317 – 320

[2] R. Brinkmann, P. Johannsen, K. Winkelmann, "Application of Property Checking and Underlying Techniques", in "*Advanced Formal Verification*", pp. 125-166, Ed. R. Drechsler, Kluwer Academic Publishers, 2004

[3] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers 35(8)*, 1986, pp. 677 – 691

[4] R. Drechsler, G. Fey, D. Große, "Avoiding false negatives in formal verification for protocol-driven blocks", *Proc. DATE*, 2006,

[5] J. Bormann, C. Spalinger, "Formal Verification for Non-Formalists)", *Information Technology*, Vol. 43, Issue 1, 2001, pp. 22

[6] J.P.M. Silva, K.A. Sakallah, "Boolean satisfiability checking in electronic design automation", *Proc. DAC*, 2000, pp. 675 – 680

[7] J.P.M. Silva, "Search algorithms for satisfiability problems in Combinational switching circuits", *Ph.D. thesis*, University of Michigan, 1995

[8] O.Shtrichman, "Tuning SAT Checkers for Bounded Model Checking", *Proc. CAV*, 2000, pp. 480 – 494

[9] R. Drechsler, "*Formal Verification of Circuits*", Kluwer Academic Publishers, 2000

[10] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, M.Y. Vardi, "Benefits of Bounded Model Checking at an Industrial Setting", *Proc. CAV*, 2001, pp. 436 – 453

Peer Johannsen holds a Ph.D. degree in computer science from Kiel University, Germany, and a master's degree in computer science. He is a professor for Computer Science and Software Engineering at Pforzheim University, Germany.

Rauschen in Stromspiegelschaltungen

Albrecht Zwick, Bernd Vettermann

***Zusammenfassung*—In integrierten Analogschaltungen werden sehr häufig Stromspiegelschaltungen verwendet. Der Beitrag beschreibt, wie sich das Rauschen der einzelnen Bauteile zu einem Gesamtrauschen zusammensetzt. Die einzelnen Rauschquellen werden mit Hilfe der Methode der Quellenverschiebung zum Ausgang verschoben und ihre Werte werden dort quadratisch addiert. Nach einigen sinnvollen Vernachlässigungen erhält man meistens dadurch sehr einfache Ergebnisse.**

***Schlüsselwörter*—Rauschen, Stromspiegel, Widlar, Wilson, Quellenverschiebung**

I. Voraussetzungen (Grundwissen)

A. Quellenverschiebung

Abbildung 1.1 zeigt die Verschiebung von Spannungsquellen [1,2]. Eine einzelne Spannungsquelle zwischen den Punkten a und b/c wird durch zwei identische Spannungsquellen über den Punkten a, b und a, c ersetzt.

Vor der Verschiebung:

$$U_{ab} = U + 0 \quad U_{ac} = U + 0 \quad U_{bc} = 0 + 0$$

Nach der Verschiebung:

$$U_{ab} = 0 + U \quad U_{ac} = 0 + U \quad U_{bc} = -U + U = 0$$

Die Spannungen zwischen den Endpunkten sind vor und nach dem Verschieben gleich.

Eine Stromquelle kann man immer ersetzen durch zwei oder mehrere identische Stromquellen in Reihe. Die dazwischenliegenden Punkte können jetzt mit jedem beliebigen Punkt verbunden werden. Die Funktion der Schaltung ändert sich damit nicht, da der Strom immer zum Punkt hin-und wieder wegfließt.

A. Zwick, a.zwick@hs-mannheim.de und B. Vettermann, b.vettermann@hs-mannheim.de sind Mitglieder der Hochschule Mannheim, Paul-Wittsack-Straße 10, 68163 Mannheim im Institut für Biomedizinische Technik und im Institut für Entwurf Integrierter Schaltkreise.

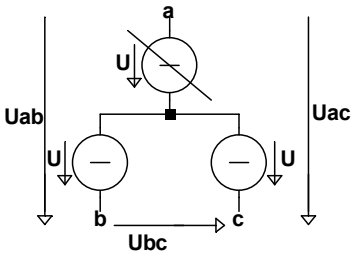


Abbildung 1.1: Spannungsquellenverschiebung

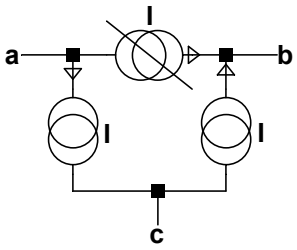


Abbildung 1.2: Stromquellenverschiebung

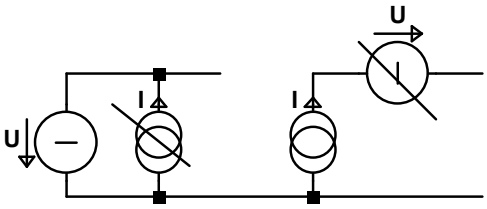


Abbildung 1.3: Quellen entfernen

In Abbildung 1.2 wird eine Stromquelle zwischen den Punkten a und b ersetzt durch zwei Stromquellen zwischen den Punkten a und c und den Punkten b und c mit passender Stromrichtung. Sehr oft können diese beiden neuen Stromquellen leichter verrechnet werden als vorher die ursprüngliche Stromquelle.

Stromquellen parallel zu einer Spannungsquelle ebenso wie Spannungsquellen in Serie zu einer Stromquelle spielen keine Rolle. Neue Quellen, die durch Quellenverschiebung entstehen, können auch wieder an den in der Abbildung 1.3 gezeigten Stellen verschwinden.

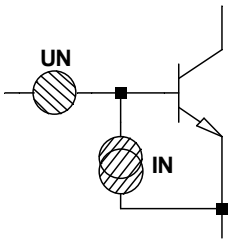


Abbildung 1.4: Transistor mit äquivalenten Rauschquellen

A. Rauschquellen eines Transistors

Unter der Annahme kleiner Ströme I_C (< 1 mA) und vernachlässigtem Basisbahnwiderstand R_B= 0 können folgende vereinfachte Gleichungen für U_N und I_N verwendet werden [3] (Abbildung 1.4):

$$U_N = \sqrt{4kT\frac{r_E}{2}\Delta f} \sim \sqrt{\frac{1}{I_C}}$$

$$r_E = \frac{U_T}{I_C} \quad \text{hierbei beträgt } U_T = 26\text{ mV}$$

$$I_N = \sqrt{2e\frac{I_C}{B}\Delta f} = \sqrt{\frac{4kT}{B2r_E}\Delta f} \sim \sqrt{I_C}$$

Die Gleichungen wurden zusätzlich mit einer Bandbreite Δf = 1 Hz multipliziert. Dadurch wird die Schreibweise $\frac{\text{nV}}{\sqrt{\text{Hz}}}$ vermieden.

B. Einfache Rauschbetrachtungen

In Abbildung 1.5 kann die Rauschstromquelle I_N auch zwischen dem Kollektor und dem Emitter gezeichnet werden. Die zugehörige Transistorersatzschaltung in Abbildung 1.6 besteht aus einem frequenzunabhängigen Widerstand r_E und dem frequenzabhängigen Widerstand Br_E. Sehr häufig wird der Widerstand r_E weggelassen. Er bietet jedoch den Vorteil, dass man an ihm auch die Spannung u_{BE} ablesen kann.

In Abbildung 1.7 wurde die Spannungsquelle U_N verschoben. In Serie zu den beiden Stromquellen können jedoch die weiteren Spannungsquellen entfallen. Außerdem kann man auch den differentiellen Widerstand r_E in Serie zur Stromquelle immer weglassen. Abb. 1.7 zeigt, dass durch die Stromquelle der B mal so große Strom fließt, wie durch den Wiederstand Br_E bei gleicher Spannung. Die Stromquelle arbeitet demnach so wie ein Widerstand r_E.

Die Rauschspannung $IN \cdot r_E$ kann man gegenüber der Rauschspannung U_N vernachlässigen. Sie ist $\frac{1}{\sqrt{B}}$ mal so klein. Bei quadratischer Addition und B = 100 ist der Wert somit 100 mal so klein (Abb. 1.8).

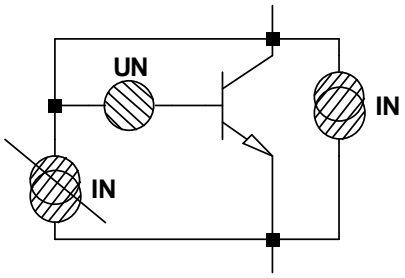


Abbildung 1.5: Transistordiode

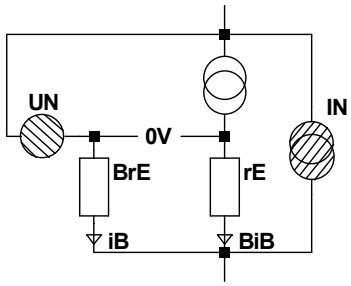


Abbildung 1.6: Transistorersatzschaltung zu Abbildung 1.5

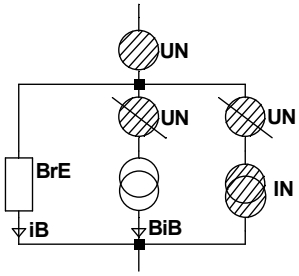


Abbildung 1.7: Verschiebung von U_N

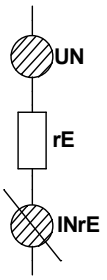


Abbildung 1.8: Rauschersatzschaltbild einer Transistordiode

$$U_N = \sqrt{4kT\frac{r_E}{2}\Delta f}$$

$$I_N \cdot r_E = \sqrt{\frac{4kT}{B2r_E}\Delta f} \cdot r_E = U_N \cdot \frac{1}{\sqrt{B}}$$

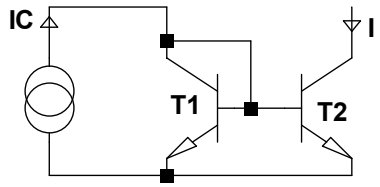


Abbildung 2.1: Einfacher Stromspiegel

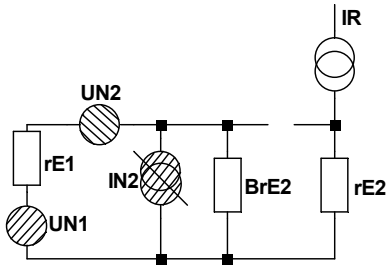


Abbildung 2.2: Rauschersatzschaltung des Stromspiegels

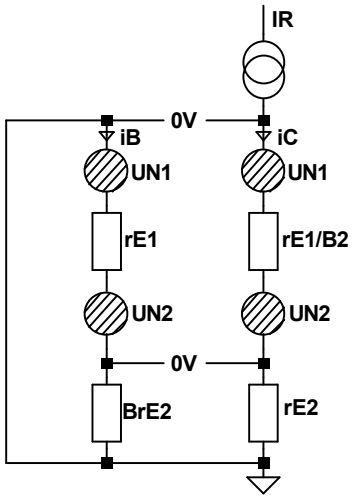


Abbildung 2.3: Umkonstruktion des Ersatzschaltbildes

II. Einfacher Stromspiegel

Abbildung 2.1 zeigt das Schaltbild des einfachen Stromspiegels. Bei Vernachlässigung der Basisströme beträgt der Strom $I = I_C$. Der Transistor T_1 wirkt wie eine Transistordiode. Abbildung 2.2 zeigt die Rauschersatzschaltung mit allen Rauschquellen. Da beide Transistoren den gleichen Kollektorstrom haben, sind auch die Rauschgrößen beider Transistoren gleich groß. I_{N2} liegt an derselben Stelle, wie die vorher bei der Transistordiode vernachlässigte Rauschstromquelle I_{N1} und kann deshalb ebenfalls vernachlässigt werden.

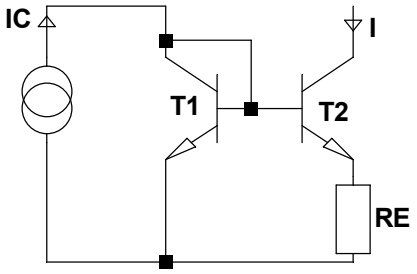


Abbildung 3.1: Widlar-Stromquelle

In Abbildung 2.3 können die 3 Bauteile oberhalb Br_{E2} auf die Seite des Kollektorstromes transformiert werden. Die Spannungsquellen bleiben unverändert, die Widerstände werden mit $\frac{1}{B}$ transformiert. Die Gesamtschaltung wird dadurch nicht verändert, denn in Serie zu einer Stromquelle können alle Bauteile vernachlässigt werden. Unterhalb der Stromquelle entsteht jetzt aber eine virtuelle Null und dadurch ein geschlossener Kirchhoff´scher Kreis.

$$I_R = \sqrt{\frac{U_{N1}^2 + U_{N2}^2}{\left(r_{E2} + \frac{r_{E1}}{B_2}\right)^2}} \approx \sqrt{2} \cdot \frac{U_N}{r_E}$$

Die Rauschspannungen und differenziellen Widerstände der Transistoren sind gleich, da die gleichen Ströme fließen.

III. Widlar-Stromspiegel

In der Widlar-Stromquelle ist aufgrund des Widerstandes R_E der Strom I des Transistors T_1 größer als der Strom des Transistors T_2. Dadurch werden auch die Rauschspannungen und Rauschströme verschieden:

$$U_{N1} < U_{N2} \text{ und } I_{N1} > I_{N2}$$

In Abbildung 3.2 wurde die Rauschstromquelle I_{N2} in zwei Rauschstromquellen umgewandelt. Eine davon kann wieder vernachlässigt werden, da sie parallel zur Transistordiode liegt. Um aus dieser Ersatzschaltung den Rauschstrom I_R zu berechnen werden wieder alle Bauteile so transformiert, dass sie sowohl im Basisstromkreis als auch im Kollektorstromkreis erscheinen. Dabei wird der Basisstrom im Widerstand R_E vernachlässigt. Der dabei entstehende Fehler ist sehr klein.

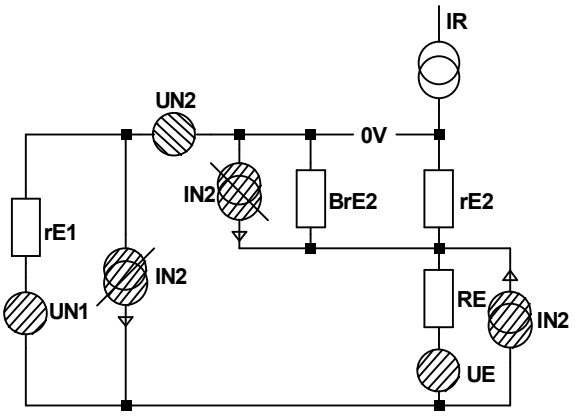


Abbildung 3.2: Ersatzschaltung der Widlar-Stromquelle

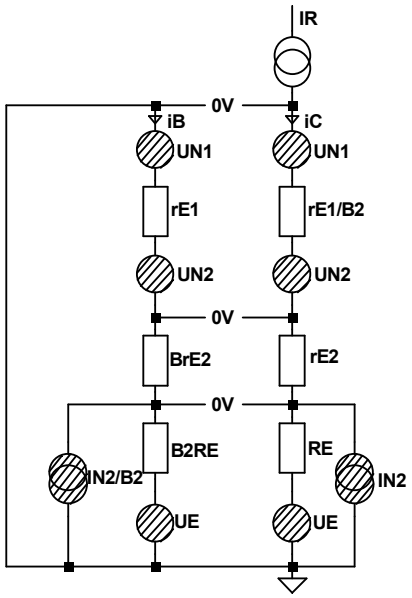


Abbildung 3.3: Umkonstruktion der Ersatzschaltung

In Abbildung 3.3 erhält man wieder zwei gleichwertige Stromkreise. Im rechten Stromkreis kann man jetzt direkt den Rauschstrom I_R berechnen.

$$I_R^2 = \frac{U_E^2 + U_{N1}^2 + U_{N2}^2 + (I_{N2} \cdot R_E)^2}{\left(R_E + r_{E2} + \frac{r_{E1}}{B_2}\right)^2}$$

IV. WILSON-STROMQUELLE

Vernachlässigt man in der Wilson-Stromquelle wieder die Basisströme, so sind alle 3 Kollektorströme gleich groß und somit auch alle Rauschstrom- und Spannungsquellen.

$$I_{N1} = I_{N2} = I_{N3}$$
$$U_{N1} = U_{N2} = U_{N3}$$

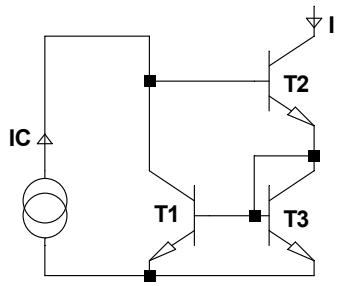


Abbildung 4.1: Wilson-Stromquelle

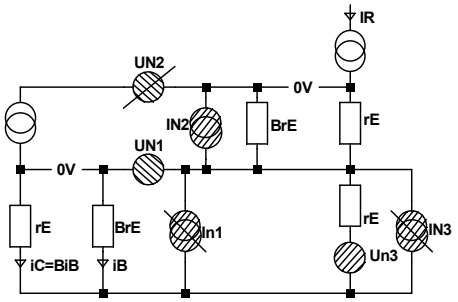


Abbildung 4.2: Ersatzschaltung der Wilson-Stromquelle

Die stromtreibende Quelle I_C kann bei der Rauschberechnung weggelassen werden. Abbildung 4.2 zeigt die Rauschersatzschaltung mit allen Rauschquellen. Die Quelle U_{N2} liegt in Serie zu einer Stromquelle und entfällt. Die Rauschstromquellen I_{N1}und I_{N3} liegen parallel zum niederohmigen Diodenwiderstand r_E und können deshalb vernachlässigt werden. Die Rauschspannungsquelle U_{N1} kann nach U_{N3} und U_{N2} verschoben werden. Bei U_{N2} entfällt sie wieder. U_{N1} und U_{N3} liegen somit an der gleichen elektrischen Stelle und können gemeinsam verrechnet werden.

Mit der Ersatzschaltung in Abbildung 4.3 kann jetzt mit dem Überlagerungssatz für I_R und U_{N1} der Kollektorstrom i_C berechnet werden. Der Einfluss von I_R wird mit der Stromteilerregel berechnet.

$$i_C = -I_R \frac{r_E}{r_E + Br_E} \cdot B + U_N \frac{1}{r_E + Br_E} \cdot B$$

$$i_C = \frac{I_R}{B} \qquad r_E < Br_E$$

$$I_R \left(\frac{1}{B} + 1\right) \approx U_N \frac{1}{r_E} \qquad I_R \approx U_N \frac{1}{r_E}$$

In Abbildung 4.4 wird die Rauschquelle I_{N2} in zwei Rauschquellen aufgeteilt. Die rechte Rauschquelle wird in eine äquivalente Rauschspannungsquelle $I_{N2} \cdot r_E$ umgerechnet.

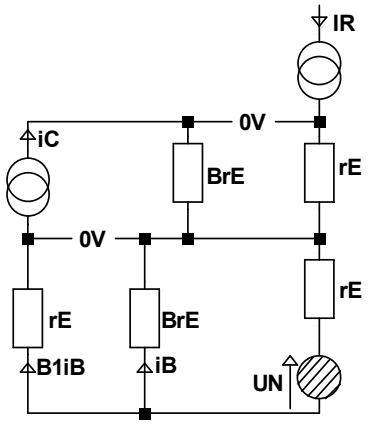


Abbildung 4.3: Ersatzschaltung zur Verrechnung der Rauschquelle U_{N1}

Diese Rauschquelle wird aber wie U_N in Abbildung 4.3 verrechnet. Die linke Rauschquelle verrechnet sich wie der Signalstrom I_C direkt zu I_R.

$$I_R = I_{N2} + \frac{I_{N2}.r_E}{r_E} = 2 \cdot I_{N2}$$

Für das gesamte Rauschen ergibt sich somit:

$$I_R{}^2 = (2 \cdot I_{N2})^2 + \left(\frac{U_{N1}}{r_E}\right)^2 + \left(\frac{U_{N3}}{r_E}\right)^2$$

mit $U_{N1} = U_{N3}$ Außerdem gilt für I_{N2}

$$(2 \cdot I_{N2})^2 < 2 \cdot \left(\frac{U_N}{r_E}\right)^2$$

$$4 \cdot \frac{4kT}{B2r_E}\Delta f < \frac{4kTr_E}{2r_E^2}\Delta f \quad B > 4$$

Für den Wilson-Stromspiegel erhält man somit näherungsweise:

$$I_R \approx \sqrt{2} \cdot \frac{U_N}{r_E}$$

Mit den Werten I_C = 10 µA, dem differentiellen Widerstand r_E = 2,6 kΩ und B = 200 ergeben sich mit der Näherungsformel die Rauschspannung und der Rauschstrom zu:

U_N = 4,64 nV und I_N = 125 fA

Für das Gesamtrauschen erhält man

I_R = 2,52 pA (Δf = 1 Hz)

Mit der genaueren Spice-Simulation erhält man für das Gesamtrauschen: I_R = 2,53 pA (Δf = 1 Hz).

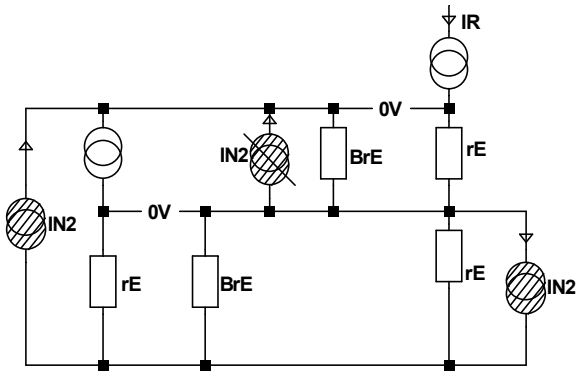


Abbildung 4.4 Ersatzschaltung zur Verrechnung der Quelle I_{N2}

Der Wilson-Stromspiegel ist im Gegensatz zu den anderen beiden Stromspiegeln im Rauschen lastunabhängig, da der Transistor *T2* im Rauschen keinen Einfluss hat und die Spannung U_{CE} über *T3* und somit dessen Arbeitspunkt in weiten Bereichen konstant bleibt.

LITERATURVERZEICHNIS

[1] A. Zwick, J. Arriaga, Y.A. Cloute, „Objetos de aprendizaje sobre el tratamiento del ruido en circuitos electrónicos mediante el desplazamiento de fuentes de tensión y corriente“, *Congreso TAEE*, 2006*, Madrid*

[2] A. Zwick, „Desplazamiento de fuentes en circuitos electrónicos“, *Congreso TAEE, 1998, Madrid*

[3] A. Zwick, „Analogtechnik 2-Rauscharme Schaltungen”, *Vorlesungsunterlagen Hochschule Mannheim*

Albrecht Zwick lehrt seit 1974 als Professor an der Hochschule Mannheim und hält Vorlesungen auf dem Gebiet der analogen Schaltungstechnik mit Spezialisierung auf rauscharme Schaltungen.

Bernd Vettermann ist seit 1994 an der Hochschule Mannheim im Institut für integrierte Schaltkreise als Ingenieur tätig. Er erhielt 2006 von der Universität Mannheim den akademischen Grad eines Doktors der Naturwissenschaften.

Multilayer-Lagenaufbauten für die Baugruppen der nächsten Generation unter dem Aspekt der EMV, Signal- und Powerintegrität

Arnold Wiemers

***Zusammenfassung*—Die Hochwertigkeit moderner elektronischer Komponenten und ihre zuverlässige Integration in eine leistungsfähige Baugruppe erfordern ein definiertes physikalisches Umfeld auf der Leiterplatte. Neben der reinen Fertigungstechnologie für Leiterplatten bekommen die strategischen Überlegungen bei der Konstruktion eines Multilayers zunehmend entscheidende Bedeutung.**

***Schlüsselwörter*—EMV, Signalintegrität, Powerintegrität, High-Speed, Leiterplatten, Impedanz, Multilayer, CAD- Designregeln**

I. Einleitung

Die Anforderungen an elektronische Baugruppen nehmen kontinuierlich zu. Im Prinzip ist die Diskussion um hochwertige Baugruppen überfällig. Strategien, Materialien, Funktion, Zuverlässigkeit und Wirtschaftlichkeit sind zu berücksichtigen.

Eine Baugruppe muss als High-Speed-Baugruppe klassifiziert werden, wenn die Übertragungsfrequenzen bei über 1 GHz liegen, wenn die Datentransferraten 1 GBit/s überschreiten, vor allem aber, wenn die Signalanstiegszeiten deutlich unter 0.3 ns liegen.

Die zunehmende Leistungsfähigkeit integrierter elektronischer Komponenten hat einen massiven Einfluss auf die physikalischen Anforderungen an Leiterplatten. Die Konstruktion der Leiterplatten für High-Speed-Baugruppen muss deshalb (auch) strategische und funktionale Aspekte beachten.

Die Umsetzung der Anforderungen ist prinzipiell nur noch mit Multilayern ab 6 Lagen aufwärts machbar. Für komplexere Baugruppen sind 10 bis 12 Lagen realistisch. Diese Entwicklung führt uns direkt zu den sogenannten „Multilayersystemen“.

II. High-Speed-Parameter

Für die Konstruktion eines zuverlässig funktionierenden Multilayers müssen drei hochwertige Anforderungen beachtet werden.

Arnold Wiemers, awi@leiterplattenakademie.de, ist Technischer Direktor der LA - LeiterplattenAkademie GmbH mit Sitz in 10555 Berlin, Krefelder Str. 18, www.leiterplattenakademie.de

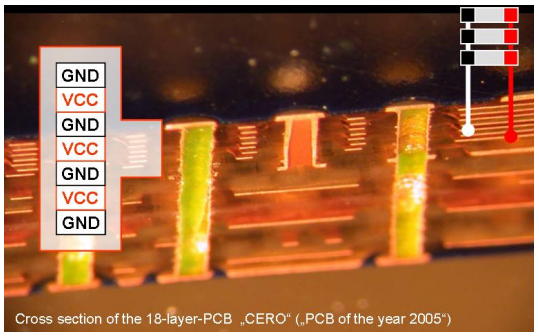


Abbildung 1: MPS (Multilpowersystem): Ein Stromversorgungsstapel mit 7 GND und VCC-Planes. Der minimale Abstand von 50 µm wird durch den Einbau von dünnen Laminaten und Prepregs erreicht.

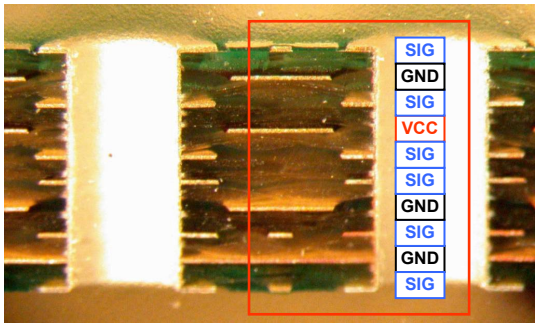


Abbildung 2: Lagenzuordnung: Bei der Verteilung der Signale, der GND- und der VCC-Planes muß auf die verbindliche Zuordnung der Rückstromwege geachtet werden.

Die *Powerintegrität* ist zu beachten. Die Spannungsversorgung der Baugruppe muss stabil und leistungsfähig sein. Die klassische Entkopplung der Schaltung kann durch den Einbau kapazitiver Powerplanes ersetzt werden.

Die Abstände zwischen GND und VCC sollten mindestens 100 µm betragen, besser sind 75 µm, ideal sind 50 µm. Als Ergänzung können gerechnete Kondensatorgruppen (…bestehend aus minimal 1 bis maximal 4 Keramikkondensatoren) die breitbandige Entkopplung vervollständigen (Abb. 1).

Die *Signalintegrität* ist zu beachten. Für alle Signalwege muss definitiv ein Rückstromweg zur Verfügung stehen. Die Realisierung dieser Forderung bedeutet den Einbau mehrerer GND-Planes (Abb. 2) in einen höherlagigen Multilayer (ab 8 Lagen).

Mit der Signalintegrität ist auch die definierte Signallaufzeit verbunden, die hauptsächlich über einen

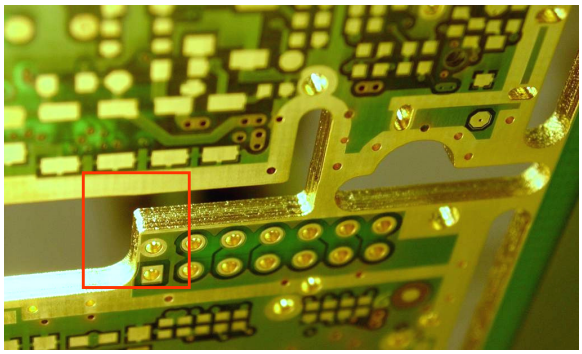


Abbildung 3: Kantenmetallisierung: Die Kantenmetallisierung (oder auch: Kantenkontaktierung) schirmt innere Bereiche eines Multilayers ab. Durch den Einsatz der Frästechnologie sind Konturen möglich.

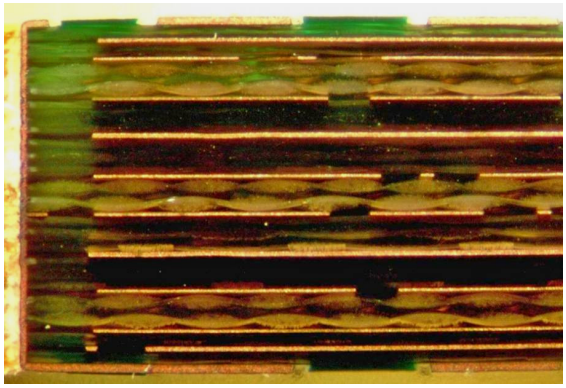


Abbildung 4: Schliff durch einen 12-Lagen-Multilayer: Laminate & Prepregs, Signal & Power wechseln sich im Lagenaufbau ab.

vorgegebenen Impedanzwert charakterisiert ist. Damit ergeben sich Vorgaben für die Abstände von Signal- und GND-Lagen innerhalb des Lagenaufbaus und für die Leiterbahnbreite und den Querschnitt.

Die *EMV-Eigenschaften* der Baugruppe sind zu beachten. Das interne und externe EMI-/EMV-Verhalten einer Baugruppe kann durch die Metallisierung der Leiterplattenkanten deutlich verbessert werden (Abb. 3). Die Strategie ist, für die Abschirmung des inneren Bereiches eines Multilayers zu sorgen. Dazu werden innenliegende GND-Planes ankontaktiert. Die Abschirmung vermeidet eine Störabstrahlung zum Nachteil benachbarter Baugruppen und sie reduziert die Störeinstrahlung zum Vorteil der eigenen Baugruppe.

Ein willkommener und sehr nützlicher Nebeneffekt ist die Wärmespreizung, die deutlich zur Kühlung der Baugruppe im Betrieb beiträgt.

High-Speed-Baugruppen finden sich in vielen Bereichen unseres täglichen Lebens. Dazu gehören Produkte und Geräte aus der Medizintechnik, der Luft- und Raumfahrt, der Sicherheitstechnik, der Verkehrstechnik, der Industrieelektronik, der Unterhaltungselektronik, dem Transportwesen, der Haustechnik und der Kommunikation.

Typisch ist eine Datenrate über 4 GBit/s für den Transport von Daten, Bildern, Videos, Musik und HDTV. Frequenzen über 15 GHz werden benötigt für die Sensortechnologie und für Fahrerassistenzsysteme (...beispielsweise das Abstandsradar).

Entscheidend ist jedoch, dass die High-Speed-Eigenschaft an die Bauteilkomponente geknüpft ist.

In wenigen Jahren wird jede digitale Schaltung die High-Speed-Bedingungen berücksichtigen müssen.

Sobald das Konzept für die Konstruktion einer Baugruppe vorliegt, gibt es für den weiteren Ablauf fünf elementare Abschnitte:

1) Erstellen des *Konzeptes* für die Konstruktion eines Gerätes
2) Erstellen des *Schaltplans* und der weiteren Vorlagen für CAD
3) Erstellen des *CAD-Layouts* und Weitergabe der Daten an CAM
4) Produktion der *Leiterplatten,* Anlieferung an den Baugruppenfertiger
5) Fertigung der *Baugruppen*, Funktionstest, Lieferung an den Kunden

Zum Zeitpunkt der Schaltplanerstellung muss bereits der verbindliche Multilayerbauplan vorliegen. Wesentliche Eigenschaften der späteren Baugruppe (Schaltungssimulation, Funktion, Prozessierbarkeit) sind sonst nicht rechtzeitig zuverlässig berechenbar und planbar.

III. Basismaterialien

Die Kenntnis der wichtigsten Eigenschaften von Basismaterialien ist der Schlüssel zum Verständnis der Funktion und der Möglichkeiten, die Leiterplatten haben können (Abb. 4).

Die technischen Eigenschaften des eingesetzten Basismaterials haben einen Einfluss auf die Eigenschaften der Leiterplatte und damit auf die Funktion der späteren Baugruppe.

Mit Bezug auf die vorausgegangenen Erläuterungen betrachten wir nachfolgend den Einfluss von Basismaterialien auf die Funktion einer Baugruppe. Das Material der nächsten Jahre werden FR4-Derivate sein.

Von speziellem Interesse sind die Prepregs (Abb. 5). Die Bewertung der mechanischen Eigenschaften wird zeigen, dass bereits die Dickentoleranz eines Prepregs einen Einfluss auf das physikalische Verhalten eines Multilayers hat. Betroffen ist die Signalintegrität, da sich durch die Dickenänderung die kapazitiven Eigenschaften auf den benachbarten Lagen ändern.

Abbildung 5: Glasgewebe und Prepregs

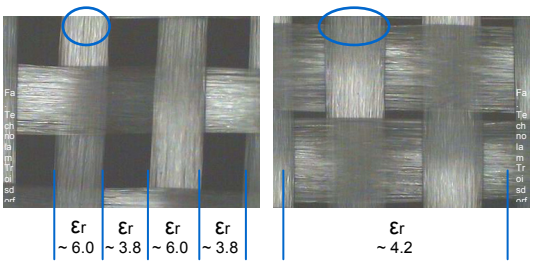


Abbildung 6: Dielektrische Eigenschaft von Prepregs

Das wirkt sich direkt auf die resultierenden Impedanzwerte der Signalleiterbahnen aus. Ob eine Abweichung zu einer vernachlässigbaren oder einer schwerwiegenden Toleranz der Impedanzwerte führt, hängt letztlich von der Geometrie der gerouteten Leiterbilder ab und von der Multilayer-Konstruktion.

Die physikalischen Eigenschaften von Leiterplatten werden weitgehend von den Eigenschaften der verbauten Basismaterialien bestimmt.

(Fast) alle elektronischen Baugruppen basieren auf dem Basismaterial FR4. Durch die Integration der Geometrien, die durch das CAD-System konstruiert werden, ergibt sich die individuelle physikalische Funktion eines Multilayers.

Der Aufbau eines Multilayers mit Laminaten und Prepregs legt seine physikalischen Eigenschaften fest. Glasgewebe können unterschiedliche Gewebestrukturen haben. Unterschiedliche Abstände zwischen den Gewebefäden (Abb. 6) beeinflussen die lokalen dielektrischen Eigenschaften (Beispiel: Prepreg Typ 1080).

Die Signallaufzeit hängt (auch) von der Platzierung der Leiterbahn auf der Oberfläche des Prepregs ab. Die Werte für differentielle Impedanzen können wegen der unterschiedlichen dielektrischen Werte für Harz und Glasgewebe stark abweichen (Abb. 7).

IV. Fertigungsbedingte Einflüsse

Die Produktionstechnologie für die Fertigung von Leiterplatten orientiert sich (auch) an der seitens des Kunden definierten Spezifikation der Leiterplatte. Je nach Anforderung sind die Auswirkungen auf die Funktion der Baugruppe zu berücksichtigen. Die Anforderung an die Qualität der Signalübertragung steigt.

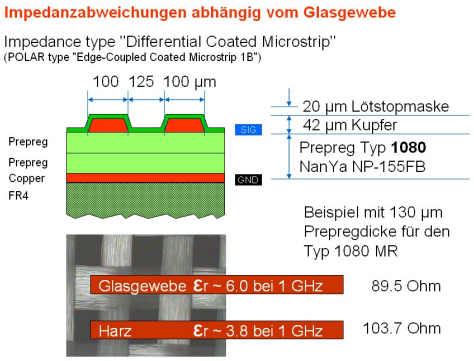


Abbildung 7: Abhängigkeit der Impedanzen vom Glasgewebe

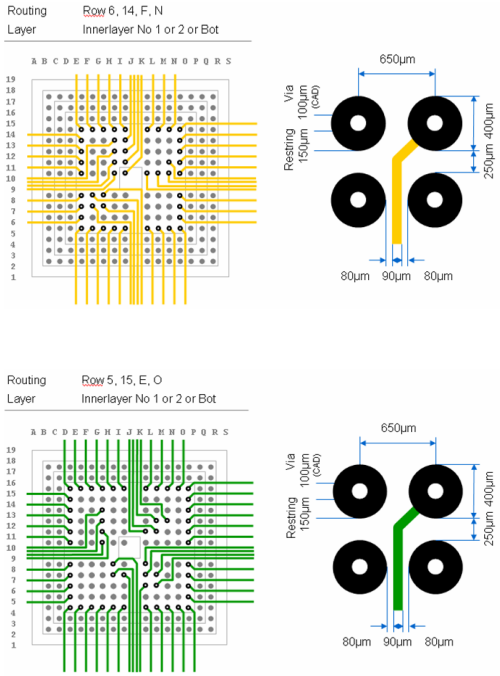
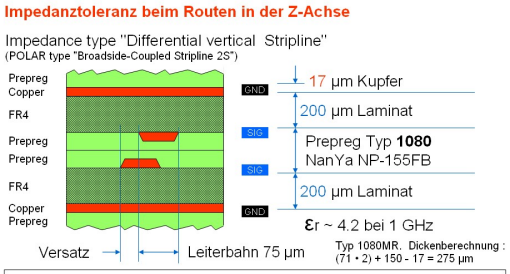


Abbildung 8: LVDS-Routing in Z-Achse

Der Pitchabstand bei integrierten Komponenten nimmt ab, die Anzahl der zu verbindenden Netze nimmt zu. Bei FPGAs hat sich die Übertragung via LVDS (**L**ow **V**oltage **D**ifferentiell **S**ignaling) durchgesetzt.

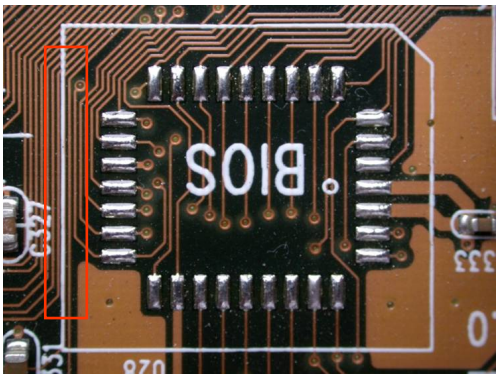
Der geringe Raum zwischen den BGA-Pads reicht für das Routing eines differentiellen Leiterbahnpaares in einer Ebene nicht mehr aus. Die Verdrahtung differentieller Leiterbahnpaaren muss deshalb über benachbarte Multilayerlagen erfolgen (Abb. 8). Das geht nur, wenn Lagenabstände und -versätze eng toleriert sind (Abb. 9).

Der *Bestückungsdruck* hat vornehmlich die Aufgabe, Bauteile auf der Leiterplatte zu kennzeichnen. Als typisches Bibliothekselement bei der Anlage eines Bauteils am CAD-System steht der informative Charakter des Bestückungsdruckes im Vordergrund (Abb. 10).



Impedanzabweichung auf Grund des Leiterbahnversatzes	
Versatz (Ecke zu Ecke)	Impedanz
+ 0 µm	100.0 Ohm
+ 10 µm	100.1 Ohm
+ 25 µm	100.5 Ohm
+ 50 µm	101.9 Ohm

Abbildung 9: Impedanztoleranz wegen Leiterbahnversatz



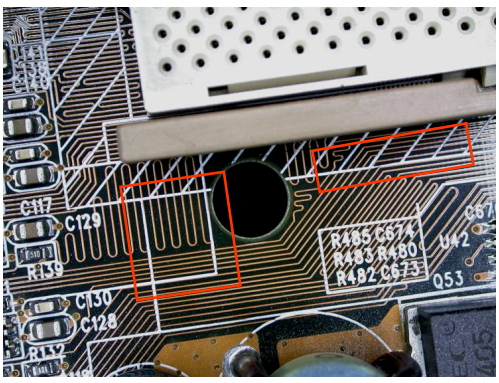
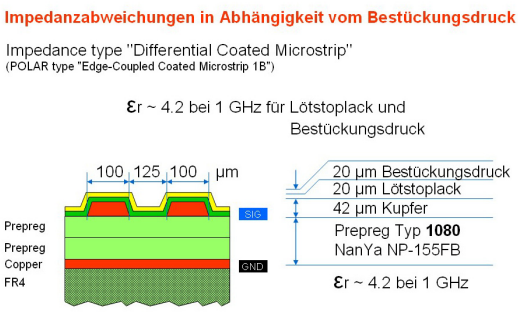


Abbildung 10: Bestückungsdruck

Dass in der Praxis „Bestückungsdruck“ ein Substrat aus Epoxydharz ist, wird dabei nicht berücksichtigt. Seine Eigenschaften sind vergleichbar dem Lötstoplack. Das Epoxydharz entspricht in seinem dielektrischen Verhalten dem Epoxydharz des FR4-Materials.

Bei einer Lackdicke von 15 bis 20 µm hat der Bestückungsdruck eine Auswirkung auf die Signallaufzeit, wenn Leiterbahnen direkt überdruckt werden (Abb. 11).



Impedanzabweichung mit Lötstoplack und mit Bestückungsdruck				
MR - resin	71-8 = 63 • 2 = 126	99.9 Ohm	93.4 Ohm
MR + resin	71+8 = 79 • 2 = 158	104.4 Ohm	97.3 Ohm

Abbildung 11: Impedanzfehler bei Bestückungsdruck

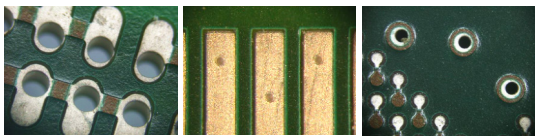


Abbildung 12: Bohrungsvarianten

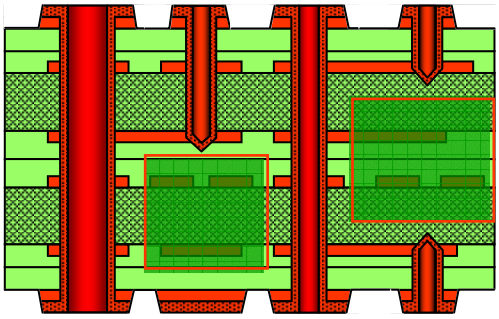


Abbildung 13: Bauteilbohrungen, DK-Vias und BlindVias

V. KONTAKTIERUNGSSTRATEGIEN

Damit die Signale über mehrere Lagen eines Multilayers transportiert werden können, müssen in Z-Achsenrichtung Verbindungen geschaffen werden. Das Bohren und Metallisieren von Vias ist dafür die technische Lösung (Abb. 12).

Bohrungen in Multilayern sind unverzichtbar. Wegen der steigenden Komplexität der Multilayer müssen „Bohrungen“ heute in die Klassen „Bauteilbohrungen“ und „Vias“ aufgesplittet werden (Abb. 13).

Vias sind als durchgehende (DK-Bohrung) oder als partielle Bohrung (BlindVia, BuriedVia) ausführbar. „BlindVias“ können im technischen Prozess mit einem Laser oder einem Bohrwerkzeug ausgeführt werden. Die Zunahme der Verdrahtungsdichte auf der Leiterplattenfläche führt notwendigerweise zu einer Reduzierung des Viadurchmessers.

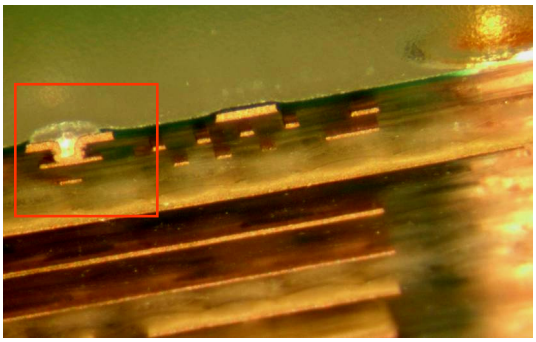


Abbildung 14: BlindVia und DK-Via

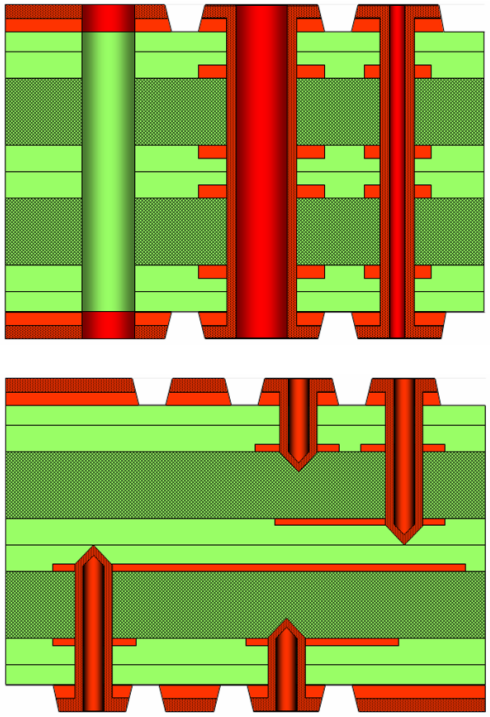


Abbildung 15: Aspect Ratio

Das wiederum beeinflusst die Kontaktierbarkeit der Viahülsen in den zur Verfügung stehenden galvanotechnischen Prozessen. Bei der Konstruktion von High-Speed-Multilayern sind diese Einschränkungen zu berücksichtigen.

Das „Aspect Ratio“ ist das Verhältnis zwischen dem Bohrwerkzeugdurchmesser und der kontaktierbaren Hülsenlänge/Bohrtiefe. Für DK-Bohrungen sind Werte von 1:8 üblich, weil die Bohrung in der Galvanik durchflutet wird, für BlindVias liegt der Wert bei 1:1. Eine DK-Bohrung von 0.2 mm Durchmesser kann auf einer Länge (i.e. die Leiterplattendicke) von 1.6mm zuverlässig kontaktiert werden (Abb. 15).

Durch die Abhängigkeit von der Leiterplattendicke müssen die Viadurchmesser bei höherlagigen (und damit dickeren) Multilayern zunehmen.

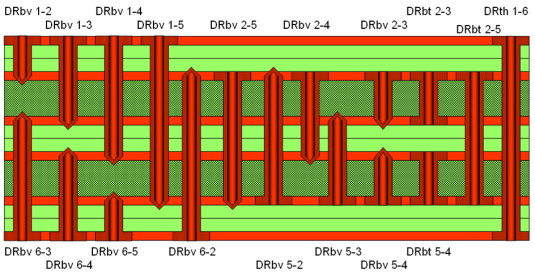


Abbildung 16: Kontaktierungsoptionen: Theoretisch gibt es 262.144 Kontaktierungsvarianten für einen 6-Lagen-Multilayer

Auch die Via-Abstände zueinander müssen dann größer werden. Durchgangsbohrungen belegen in einem Multilayer einen bestimmten Raum. Für andere Bohrungen und für Leiterbahnen ist dieser Raum gesperrt und nicht mehr nutzbar. Die gesperrten Räume multiplizieren sich mit der Anzahl der Lagen des Multilayer.

Mit partiellen Bohrungen (BlindVias, Buried Vias) wird dieser Raumverlust minimiert. Die Schaltung kann kompakter und viel effektiver verdrahtet werden, was gleichzeitig die Signalintegrität verbessert.

Weil Signalwege oft nur über zwei Ebenen führen, wirkt der potentiallose Anhang (Stub) eines DK-Vias wie eine Antenne. Der Einsatz partieller Vias verbessert damit also auch das EMV-Verhalten der Baugruppe.

Die Verbindung der Netze auf einer Leiterplatte erfordert eine durchdachte Kontaktierungsstrategie. Die möglichen Via-Kombinationen (Abb. 16) müssen mit der Anordnung der Laminate und Prepregs vereinbar sein. Neben der physikalischen Funktion einer High-Speed-Baugruppe muss also auch die Technologie der Leiterplattenfertigung beachtet werden.

VI. Strategische Vorgaben

Bedingt durch die abwechselnde Schichtung von Prepregs und Laminaten in einem Multilayeraufbau ergeben sich pauschale Bauvarianten. Von der Funktion, der CAD-Konstruktion und der wirtschaftlichen Bewertung her ergeben sich deutliche Unterschiede.

Der Entwurf eines Multilayersystems orientiert sich an den funktionalen Anforderungen, an den Produktionstechnologien und am wirtschaftlichen Umfeld.

Üblicherweise sind nie alle Anforderungen erfüllbar. Es gilt jeweils, die individuelle Lösung zu finden, mit der die wichtigsten Vorgaben an die Baugruppe erfüllt werden können.

Auswahl und Reihenfolge der Laminate bestimmen die Kontaktierungsstrategie und damit die Constraints für das CAD-System. Die eingesetzten Materialmengen an Prepregs und Laminaten bestimmen in Verbindung mit einfachen aber effektiven Bohrtechnologien (i.e. Laser) insbesondere bei großen LP-Stückzahlen die Kosten.

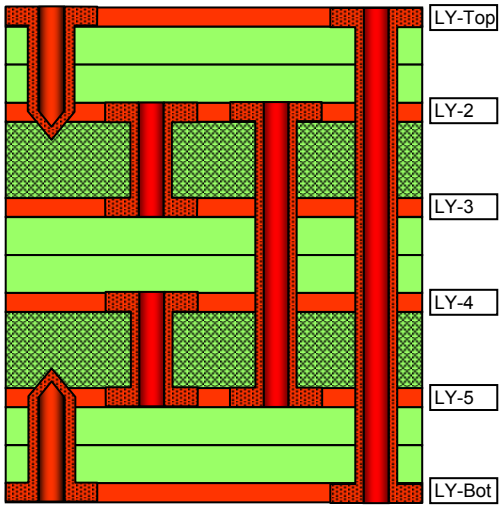


Abbildung 17: 6-Lagen-Multilayer mit zwei innenliegenden Kernen.

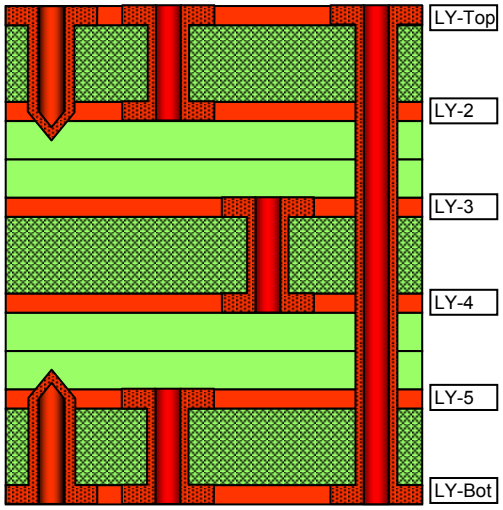


Abbildung 18: 6-Lagen-Multilayer mit zwei außenliegenden Kernen und mit einem inneren Kern.

Die komplexe Konstruktion eines Multilayersystems bietet Lösungen im High-End-Bereich. Ein Multilayer mit *innenliegenden Kernen* ist (ab 6 Lagen) die unkomplizierteste Bauvariante (Abb. 17). Für High-Speed-Baugruppen ist das die Low-Price-Version.

Alle einzelnen Lagen/Laminate sind separat kontaktierbar und können deshalb als temporäre doppelseitige Leiterplatte oder als temporärer 4-Lagen-ML gebaut werden. Die äußeren Lagen sind problemlos mit BlindVias kontaktierbar.

Eine interne Umverdrahtung, zum Beispiel bei hochpoligen BGAs, ist möglich. Große Stückzahlen sind preiswert herstellbar. Ein Hybridaufbau ist nicht ideal.

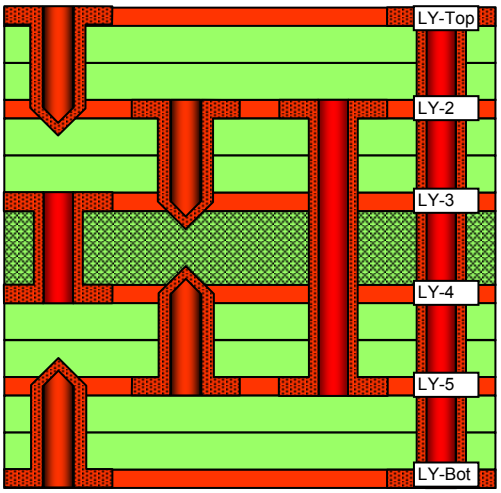


Abbildung 19: 6-Lagen-Multilayer mit einem innenliegenden Kern und mit weiteren Prepreg-Lagen zu beiden Außenseiten hin.

Ein Multilayer mit *außenliegenden Kernen* ist (ab 6 Lagen) ideal, wenn ein Hybrid gebaut werden soll, oder wenn das Aspect Ratio für das Kontaktieren der außenliegenden Kerne das erfordert (Abb. 18).

High-Speed-Baugruppen sind in der Grundversion möglich. Der innere Kern ist separat kontaktierbar. Die äußeren Lagen sind problemlos mit BlindVias als Laservia oder als konventionelle Bohrung kontaktierbar. Eine interne Umverdrahtung, zum Beispiel bei hochpoligen BGAs, ist möglich. Große Stückzahlen sind teuer.

Ein Multilayer mit *sequentiellem Aufbau* hat keinerlei Einschränkungen bezüglich der möglichen Kontaktierungsoptionen (Abb. 19). High-Speed-Baugruppen sind in der Grundversion möglich.

Die sequentielle Aufbaustrategie erfordert einen zusätzlichen Pressvorgang. Bedingt durch Materialverzüge und Produktionstoleranzen ist dieser Aufbau für kleinere Stückzahlen nicht wirtschaftlich. Eine interne Umverdrahtung, zum Beispiel bei hochpoligen BGAs, ist möglich. Große Stückzahlen sind teuer und setzen oft ein Viaplugging voraus.

VII. AUFBAUMODULN

Durch die Reihenfolge von Laminaten und Prepregs werden die Abstände zwischen den Kupferlagen festgelegt. Daraus ergeben sich weitreichende Einflüsse auf die physikalischen und funktionalen Eigenschaften einer Baugruppe.

Die *Vorgabe* an das CAD-Layout ist, dass für SIG, GND und VCC jeweils eine eigene komplette Lage genutzt wird.

Die *Entkopplung* wird als „gut“ bewertet, wenn VCC und GND benachbart sind (Abb. 20). Dann *könnte* bei Abständen ≤ 100 µm ein Flächenkondensator ausgeprägt werden.

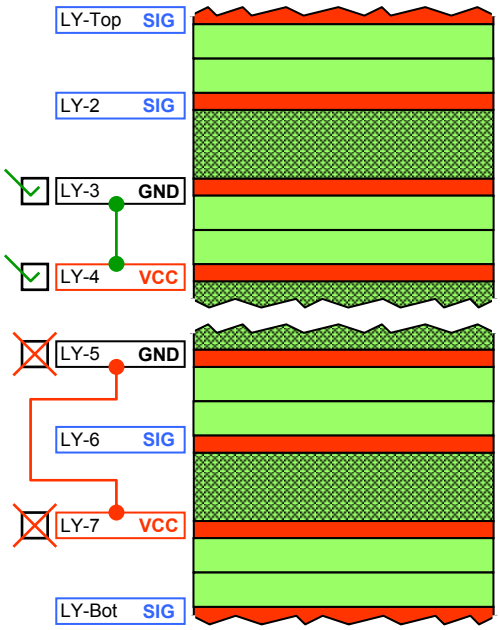


Abbildung 20: Qualität „Entkopplung“

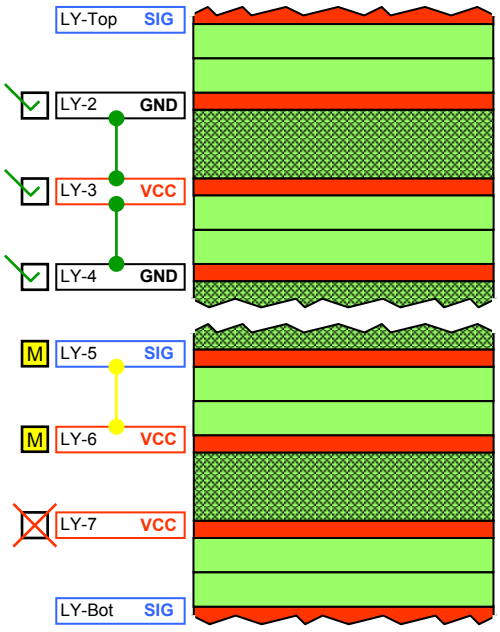


Abbildung 21: Qualität „EMV“

Die Bewertung ist „schlecht“, wenn VCC und GND nicht benachbart sind. Dann können kapazitive Effekte auf keinen Fall genutzt werden.

Die *EMV* wird als „gut“ bewertet, wenn alle inneren SIGNAL-Lagen von GND-Planes abgedeckt werden (Abb. 21) und wenn die VCC-Planes innen liegen und durch Kantenmetallisierung abgeschirmt werden *könnten*.

Die Bewertung ist „mittel“, wenn ein SIGNAL in der Nachbarlage ein VCC hat.

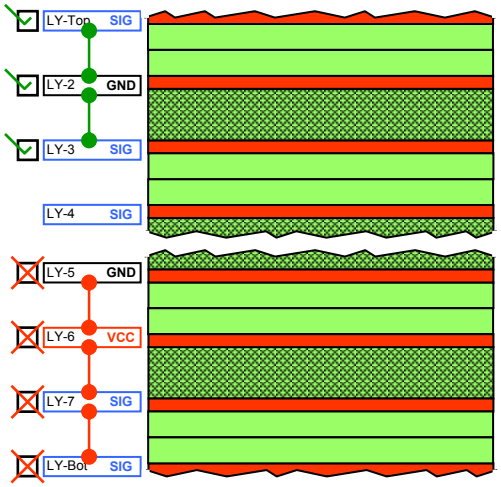


Abbildung 22: Qualität „Signalintegrität“

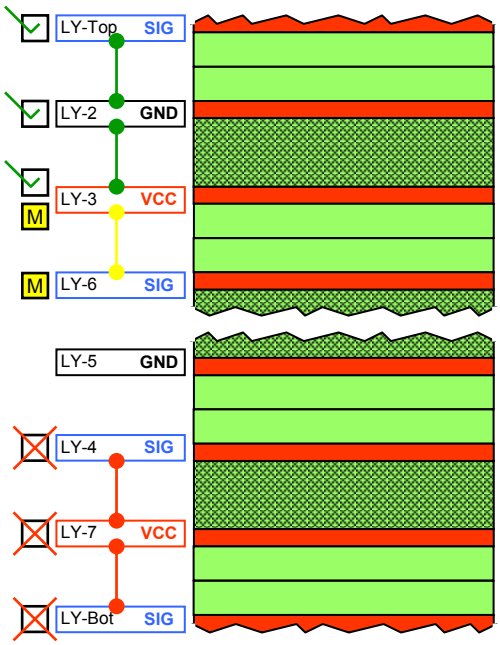


Abbildung 23: Qualität "Eigenstörung"

Die Bewertung ist „schlecht“, wenn VCC nicht beidseitig durch GND abgedeckt wird und wenn VCC nicht durch eine Kantenmetallisierung abgeschirmt werden *könnte*.

Die *Signalintegrität* wird als „gut“ eingestuft, wenn jedes SIGNAL ein GND als direkte Nachbarlage hat (Abb. 22).

Die Bewertung ist „schlecht“, wenn es ein SIGNAL gibt, das zwischen sich und GND ein zweites SIGNAL sieht oder ein VCC.

Die *Eigenstörung* ist „niedrig“, wenn alle SIGNALE durch GND von VCC abgeschirmt sind (Abb. 23).

Sie wird als „mittel“ bewertet, wenn nur ein SIGNAL betroffen ist.

Tabelle 1

	Variante 1	Variante 2	Variante 3	Variante 4	Variante 5	Variante 6
Layer 1	SIGNAL	SIGNAL	SIGNAL	GND	SIGNAL	SIGNAL
Layer 2	GND	GND	SIGNAL	VCC	GND	GND
Layer 3	VCC	VCC	GND	SIGNAL	VCC	SIGNAL
Layer 4	SIGNAL	GND	VCC	SIGNAL	SIGNAL	VCC
Layer 5	SIGNAL	SIGNAL	VCC	SIGNAL	GND	SIGNAL
Layer 6	VCC	SIGNAL	GND	SIGNAL	SIGNAL	SIGNAL
Layer 7	GND	GND	SIGNAL	VCC	GND	GND
Layer 8	SIGNAL	SIGNAL	SIGNAL	GND	SIGNAL	SIGNAL
Entkopplung	gut	gut	gut	gut	gut	schlecht
EMV	mittel	gut	mittel	mittel	mittel	schlecht
Signalintegrität	schlecht	gut	schlecht	schlecht	gut	schlecht
Eigenstörung	hoch	niedrig	niedrig	hoch	mittel	mittel

Abbildung 24: Die meisten theoretisch möglichen Kombinationen sind in der Praxis nicht brauchbar.

Tabelle 2

	Variante 1	Variante 2	Variante 3	Variante 4	Variante 5	Variante 6
Layer 1	SIGNAL	SIGNAL	SIGNAL	GND	SIGNAL	SIGNAL
Layer 2	GND	GND	SIGNAL	VCC	GND	SIGNAL
Layer 3	VCC	VCC	GND	GND	VCC	GND
Layer 4	GND	GND	SIGNAL	SIGNAL	SIGNAL	SIGNAL
Layer 5	VCC	SIGNAL	GND	SIGNAL	GND	VCC
Layer 6	GND	GND	VCC	GND	SIGNAL	SIGNAL
Layer 7	SIGNAL	SIGNAL	GND	VCC	SIGNAL	SIGNAL
Layer 8	SIGNAL	SIGNAL	SIGNAL	SIGNAL	GND	SIGNAL
Layer 9	GND	GND	SIGNAL	SIGNAL	SIGNAL	SIGNAL
Layer 10	SIGNAL	SIGNAL	SIGNAL	GND	SIGNAL	SIGNAL
Entkopplung	gut	gut	gut	gut	gut	schlecht
EMV	gut	gut	gut	mittel	mittel	schlecht
Signalintegrität	gut	gut	schlecht	schlecht	schlecht	schlecht
Eigenstörung	niedrig	niedrig	niedrig	mittel	mittel	hoch

Abbildung 25: Ab 10 Lagen können für High-Speed-Anwendungen gute Lösungen für EMV, SI und PI erwartet werden.

Sie wird als „hoch“ angesehen, wenn mehr als ein SIGNAL betroffen ist.

Im Ergebnis zeigt die tabellarische Übersicht, dass die meisten theoretisch möglichen Kombinationen für 8-Lagen-Multilayer in der Praxis nicht optimal sind wenn nicht sogar unbrauchbar (Abb. 24).

Erst ab 10 Lagen können für High-Speed-Anwendungen gute Lösungen für EMV, SI und PI erwartet werden (Abb. 25).

Die Kombination der technischen Anforderungen an die Signalqualität einer modernen Baugruppe ergibt notwendigerweise das kluge und überlegte Zusammenfügen von Funktionsmoduln zu einem Multilayersystem. In Summe ergibt sich so die effektive Konstruktion von *Funktionsräumen* in einem Multilayersystem (Abb. 26).

Der *tatsächliche* Lagenaufbau muss mit allen mechanischen und technischen Parametern (…den sogenannten „Constraints“) im CAD-System hinterlegt sein. Nur unter dieser Bedingung wird die virtuelle Welt des CAD-Systems mit der realen Welt korrekt abgeglichen (Abb. 27).

Die Simulation der Schaltungseigenschaften und die zuverlässige Produktion der Leiterplatten sind erst dann möglich.

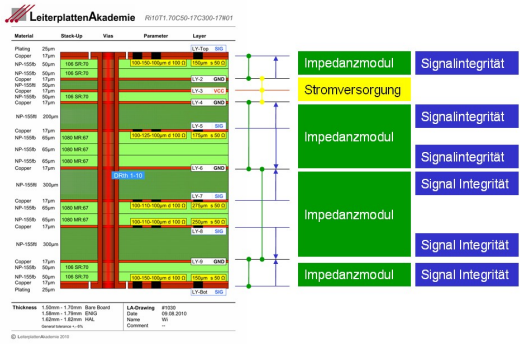


Abbildung 26: Funktionsmoduln in einem Multilayer

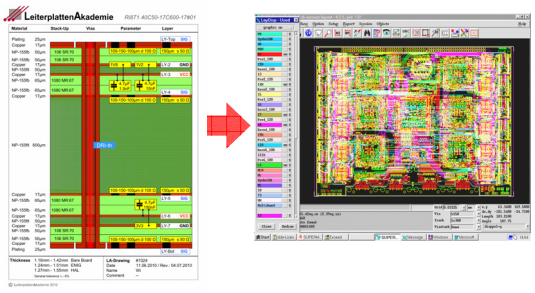


Abbildung 27: Integration der Constraints in das CAD-System

VIII. FAZIT

Die Funktion eines Multilayers oder einer Baugruppe definiert sich nicht allein durch das Aufeinanderschichten von Basismaterialien.

Die Strategie muss sein, alle Partner vom Entwurf des Schaltungskonzeptes bis hin zur Produktion der Baugruppe einzubeziehen.

Das setzt ein hohes Maß an Bereitschaft zur *Kommunikation* voraus. Die erarbeiteten Vorgaben müssen systematisiert eingebunden werden. Das geht nur mit einer verlässlichen *Organisation*.

Weil eine Baugruppenproduktion dynamisch ist, muss diszipliniert mit *Informationen* umgegangen werden.

Die Zuverlässigkeit, mit der ein CAD-Layout, eine Leiterplatte oder eine Baugruppe herstellbar ist, ist direkt proportional zu der Qualität der Information, die zum Zeitpunkt der Bearbeitung zur Verfügung gestanden hat. Die Materialien und die Fertigungsprozesse der Partner müssen den Beteiligten vertraut sein.

Persönliche Kompetenz entscheidet und nur die *qualifizierte* und *kontinuierliche* Ausbildung der Menschen, die dieses Produkt herstellen, garantiert die *zuverlässige* Funktion von High-Speed-Baugruppen.

Arnold Wiemers ist der Leiterplatte seit 1983 verbunden. Von 1985 bis 2009 war er bei der ILFA GmbH in Hannover verantwortlich für die Fachbereiche CAD und CAM, für die Auftragsvorbereitung und für die technische Dokumentation

Seit 2009 ist er Teilhaber und Technischer Direktor der LeiterplattenAkademie.

Diverse Fachveröffentlichungen, Seminare, Konferenzvorträge und Workshops zum Thema Leiterplattentechnologie.

Vom IPC zertifizierter CID, CID+ und Instructor. FED-Designer und FED-Referent. Mitarbeit am Schulungskonzept des FED. Mitarbeit in der internationalen "Projektgruppe Design" des FED und des VdL.

MULTI PROJEKT CHIP GRUPPE

Hochschule Aalen
Prof. Dr. Bartel, (07361) 576-4182
manfred.bartel@htw-aalen.de

Hochschule Albstadt-Sigmaringen
Prof. Dr. Rieger, (07431) 579-124
rieger@hs-albsig.de

Hochschule Esslingen
Prof. Dr. Lindermeir, (0711) 397-4221
walter.lindermeir@hs-esslingen.de

Hochschule Furtwangen
Prof. Dr. Rülling, (07723) 920-2503
rue@hs-furtwangen.de

Hochschule Heilbronn
Prof. Dr. Gessler, (07940) 1306-184
gessler@hs-heilbronn.de

Hochschule Karlsruhe
Prof. Dr. Koblitz, (0721) 925-2238
rudolf.koblitz@hs-karlsruhe.de

Hochschule Konstanz
Prof. Dr. Burmberger, (07531) 206-255
gregor.burmberger@htwg-konstanz.de

Hochschule Mannheim
Prof. Dr. Paul, (0621) 292-6351
g.paul@hs-mannheim.de

Hochschule Offenburg
Prof. Dr. Jansen, (0781) 205-267
d.jansen@hs-offenburg.de

Hochschule Pforzheim
Prof. Dr. Kesel, (07231) 28-6567
frank.kesel@hs-pforzheim.de

Hochschule Ravensburg-Weingarten
Prof. Dr. Ludescher, (0751) 501-9685
ludescher@hs-weingarten.de

Hochschule Reutlingen
Prof. Dr. Kreutzer, (07121) 271-7059
hans.kreutzer@hochschule-reutlingen.de

Hochschule Ulm
Prof. Dipl.-Phys. Forster, (0731) 50-28338
forster@hs-ulm.de

www.mpc.belwue.de