MPC

MULTI PROJEKT CHIP GRUPPE
BADEN-WÜRTTEMBERG

Herausgeber: Hochschule Ulm **Ausgabe:** 39 **ISSN** 1862-7102 **Workshop:** Ravensburg-Weingarten Februar 2008

Cooperating Organisation
Solid-State Circuit Society Chapter
IEEE German Section

**Tagungsband zum Workshop der Multiprojekt-Chip-Gruppe Baden-Württemberg,
Ravensburg-Weingarten, 1. Februar 2008**

Inhaltsverzeichnis

Gefertigte Bausteine

Diesen Workshopband und alle bisherigen Bände finden Sie im Internet unter:
http://www.mpc.belwue.de

Digitale Signalvorverzerrung mit FPGAs zur Linearisierung von Audioverstärkern

Friedrich Kiesel, Dipl.-Ing. (FH) Josef Hahn-Dambacher, Prof. Dr. Heinz-Peter Bürkle

HTW Aalen, EDA Zentrum, Anton Huber Strasse 25, 73430 Aalen

Tel. 07361 / 576 – 4247, Fax 07361 / 576 - 444249

Zusammenfassung

Beim Entwurf eines Leistungsverstärkers muss immer ein Kompromiss zwischen Wirkungsgrad und Linearität eingegangen werden. Um eine hohe Linearität zu gewährleisten werden PAs[1] gewöhnlich in einem Arbeitspunkt weit unterhalb der Sättigung betrieben, was aber zu einem niedrigen Wirkungsgrad führt. Durch digitale Signalvorverzerrung ist es möglich auch bei größerer Aussteuerung eine gute Linearität zu erreichen. Diese Verfahren lassen sich mittels FPGA[2]-Bausteinen sehr effizient implementieren. Eine weitere Verbesserung der Linearität kann durch adaptive Anpassung der Vorverzerrungsparameter im laufenden Betrieb erzielt werden.

Ziel dieser Arbeit ist die Realisierung einer solchen Schaltung, die sich adaptiv an den angeschlossenen Verstärker anpassen kann. Die eingesetzten Verstärker können dabei als gedächtnislos betrachtet werden. Des Weiteren wird der Erfolg der Arbeit mit Messreihen belegt.

1. Einführung

1.1. Motivation

Schon seit mehreren Jahren spielen im Bereich der Unterhaltungselektronik Mobilgeräte wie Mobiltelefone oder MP3-Player eine immer wichtigere Rolle. Besonders in Mobiltelefone werden immer mehr Funktionen integriert, darunter in immer stärkeren Maße Multimediafähigkeiten wie MP3-Player, Videoabspielmöglichkeiten oder Rundfunkempfänger. Zu deren Umsetzung werden unbedingt auch Audioverstärker benötigt. Die erreichbare Akkulaufzeit dieser Mobilgeräte hängt direkt vom Energieverbrauch der Einzelkomponenten ab, also auch vom Verbrauch der integrierten Audioverstärker. So spielt der Wirkungsgrad dieser Verstärker eine besonders wichtige Rolle.

Aber auch beim stationären Einsatz von Audioverstärkern sind die Kosten für Energie und Kühlung nicht zu vernachlässigen. So ist im Bereich der professionellen Beschallungstechnik eine kompakte Bauform, sowie ein geringes Gewicht der Verstärker sehr wichtig um die Kosten für Transport und Lagerung zu verringern. Da die zur Kühlung der Leistungskomponenten notwendigen Kühlkörper einen Großteil des Platzverbrauches und des Gewichtes verursachen, sind diese Voraussetzungen nur durch einen hohen Wirkungsgrad zu erreichen.

Im Bereich der Audiotechnik werden aber auch besonders hohe Anforderungen an die Linearität der Leistungsverstärker gestellt. Um eine hohe Linearität der Verstärkung zu erreichen, werden die Verstärker meist in einem Arbeitspunkt betrieben, in dem der Verstärker auch bei maximaler Aussteuerung weit unterhalb der Sättigung bleibt. Dies hat allerdings eine wesentliche Verschlechterung des Wirkungsgrades zur Folge. Aus diesen Gründen muss beim Entwurf von Audioverstärkern immer ein Kompromiss zwischen Linearität und Wirkungsgrad eingegangen werden.

Eine Möglichkeit zur Verbesserung dieser Situation besteht darin, den Verstärker bis nahe an die Sättigung auszusteuern, um den Wirkungsgrad zu verbessern. Die nun verschlechterten Eigenschaften bezüglich der Linearität, können durch digitale Signalvorverzerrung wieder verbessert werden.

2. Grundlagen

2.1. Prinzip der Vorverzerrung

Die digitale Signalvorverzerrung verfolgt den Ansatz, das Eingangssignal schon vor dem Leistungsverstärker so zu verändern, dass die Nichtlinearität des Verstärkers kompensiert wird. Dies kann durch Einfügen eines nichtlinearen Blocks vor dem Leistungsverstärker erreicht werden. Dieser nichtlineare Block muss nun ein Eingangssignal für den Verstärker generieren,

[1] Power Amplifier

[2] Field Programmable Gate Array

welches die zur Verzerrung des Verstärkers genau entgegengesetzte Verzerrung enthält. Die Kaskadierung dieser nichtlinearen Blöcke hat nun ein lineares Verhalten.

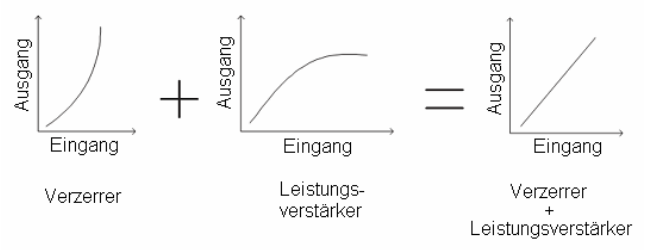


Abbildung 2-1: Vereinfachtes Funktionsprinzip der Vorverzerrung

Das Verhalten des Blocks zur Vorverzerrung kann durch eine Funktion beschrieben werden. Der Verlauf dieser Funktion hängt direkt von der Kennlinie des angeschlossenen Verstärkers ab. Eine Funktion, die das Verhalten ihrer Ursprungsfunktion rückgängig macht, wird in der Mathematik Umkehrfunktion genannt. Der Graph einer Umkehrfunktion kann durch Spiegelung des Graphen der Originalfunktion an der Diagonalen y = x ermittelt werden.

2.2. Grundlegender Aufbau

Der grundlegende Aufbau einer Schaltung zur digitalen Signalvorverzerrung ist in Abbildung 2-2 dargestellt. Dieser Aufbau kann in zwei Teile aufgeteilt werden. Zum einen den Vorwärtspfad, in dem das Eingangssignal verarbeitet wird. Beim zweiten Teil handelt es sich um die Rückkopplung, welche für die Anpassung an das Verstärkerverhalten verantwortlich ist.

2.2.1 Vorwärtspfad

Der Vorwärtspfad besteht aus dem Analog-Digital-Wandler für das Eingangssignal, dem Verzerrer und dem Digital-Analog-Wandler zur Signalausgabe an den Leistungsverstärker. In diesem Zweig wird die eigentliche Vorverzerrung ausgeführt. Zunächst digitalisiert ein Analog-Digital-Wandler das analoge Eingangssignal. Anschließend wird dieses Digitalsignal durch den Verzerrer digital verzerrt. Auf die genaue Funktionsweise des Verzerrers wird später noch eingegangen. Nach dem Verzerrer gibt ein Digital-Analog-Wandler das vorverzerrte Signal analog an den Leistungsverstärker aus.

2.2.2 Rückkopplung

Der Rückkopplungszweig besteht aus dem Analog-Digital-Wandler zur Digitalisierung des Rückkopplungssignales, Verzögerungselement und dem Adaptionsblock. Diese Komponenten sind für die Anpassung des Verzerrers an das aktuelle Verstärkerverhalten verantwortlich. Hierzu wird das vom PA verstärkte Signal wieder digitalisiert und anschließend von der Adaption mit dem zuvor ausgegebenen Signal verglichen. Dabei ist es sehr wichtig, dass der Adaption ein zeitlich zusammengehörendes Wertepaar aus dem ursprünglich an den Verstärker ausgegebenen Wert und dem verstärkten Wert zur Verfügung steht. Aus diesem Grund muss das ausgegebene Signal um die durch den Verstärker verursachte Verzögerung verzögert werden. Im Adaptionsblock werden aus den so erhaltenen Daten die für die Vorverzerrung notwendigen Parameter berechnet.

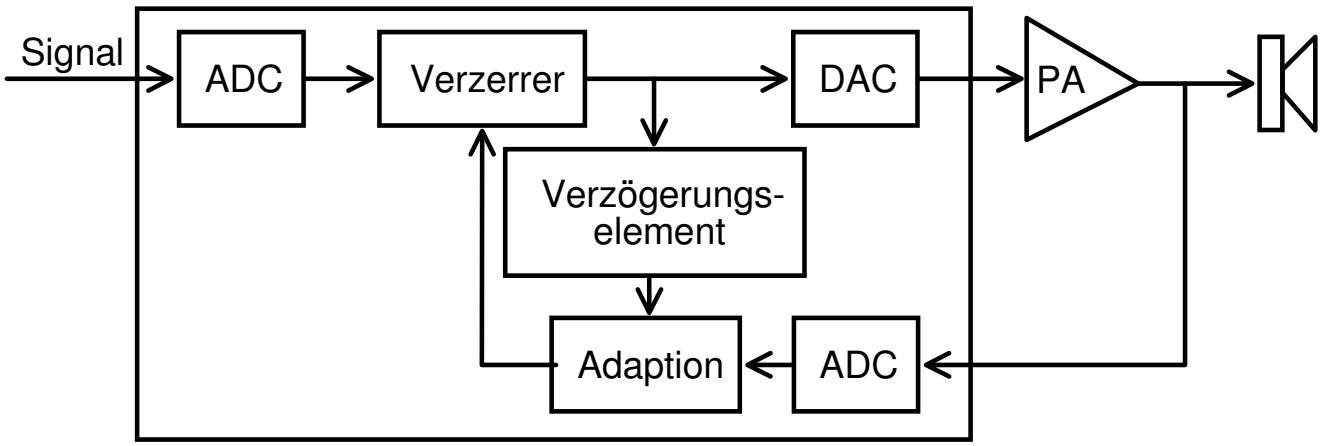


Abbildung 2-2: Grundlegender Aufbau mit Adaption

3. Realisierung

3.1. Der Verzerrer

Der Verzerrer ist eine der wichtigsten Komponenten der Schaltung. Hier wird die eigentliche digitale Vorverzerrung durchgeführt. Für die Realisierung eines solchen Verzerrers sind zwei Lösungsansätze denkbar:

- Look up Table basierte Vorverzerrung
- Polynombasierte Vorverzerrung

3.1.1 Look up Table basierte Vorverzerrung

Bei der LUT[3] basierten Vorverzerrung wird die Funktion, mit der verzerrt werden soll, in einer Tabelle abgespeichert. Jedem möglichen Eingangswert wird dabei genau ein Tabelleneintrag mit dem vorberechneten Ergebnis zugewiesen. Für die Verarbeitung eines Eingangswertes ist es so nur notwendig, den entsprechenden Eintrag aus der Tabelle auszulesen und das Ergebnis auszugeben. Dies bietet den Vorteil einer sehr schnellen Verarbeitung der Daten, da nur ein Speicherzugriff pro Datensatz notwendig ist. Leider benötigt eine solche LUT relativ viel Speicherplatz. Bei einer Auflösung des Eingangssignals von 16 Bit sind beispielsweise 65536 unterschiedliche Werte am Eingang möglich. Für jeden möglichen Eingangswert müssen 2 Byte gespeichert werden. Es werden somit 128 kByte an Speicher für die LUT benötigt. Für eine Auflösung von 16 Bit ist der Speicherbedarf also noch akzeptabel, für größere Auflösungen wie beispielsweise 24 Bit würde sich aber ein Speicherbedarf von 48 MByte ergeben, was sich mit einem FPGA nur mit sehr großem Aufwand realisieren lässt. Außerdem muss bei einer adaptiven Implementierung immer die komplette Funktion berechnet werden, auch wenn bestimmte Funktionsabschnitte nie verwendet werden. Bei einer Vergrößerung der Auflösung steigt der Rechenaufwand für die Berechnung der LUT exponentiell an.

3.1.2 Polynombasierte Vorverzerrung

Eine polynombasierte Vorverzerrung nähert die Funktion, nach der verzerrt werden soll, mittels eines Polynoms der Form

$$F_{(x)} = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$$

an. Der Vorteil dieses Verfahrens liegt in der Speicheplatzersparnis gegenüber LUT basierter Verfahren. Auch ist dieses Verfahren für beliebig große Auflösungen verwendbar. Allerdings steigt der Berechnungsaufwand bei Polynomen höherer Ordnung stark an. Bei Eingangssignalen mit großer Auflösung kann es auch notwendig werden, sequentielle Multiplizierer einzusetzen, um die für den Betrieb der restlichen Komponenten notwendigen Taktraten des FPGAs einhalten zu können. Außerdem ist die Berechnung der Koeffizienten des Polynoms sehr aufwändig und wird vermutlich einen Embedded Prozessor notwendig machen. Weiterhin ist für die Berechnung der Koeffizienten eine größere Menge an Wertepaaren aus Ausgabewerten und gemessenen Werten notwendig, für die unter Umständen auch relativ viel Speicher benötigt werden kann.

3.1.3 Auswahl des Verzerrungsverfahrens

In der vorliegenden Applikation ist die Auflösung des Eingangssignals auf 16 Bit festgelegt. Der Speicherbedarf einer LUT ist mit 128 kByte wie auch der Berechnungsaufwand zur Erstellung dieser Tabelle noch akzeptabel. Außerdem erscheint aufgrund des recht aufwändigen Berechnungsverfahrens der polynombasierten Vorverzerrung der LUT basierte Ansatz für eine Implementierung auf einem FPGA als besser geeignet. Aus diesen Gründen wird zur Implementierung des Verzerrers eine LUT eingesetzt.

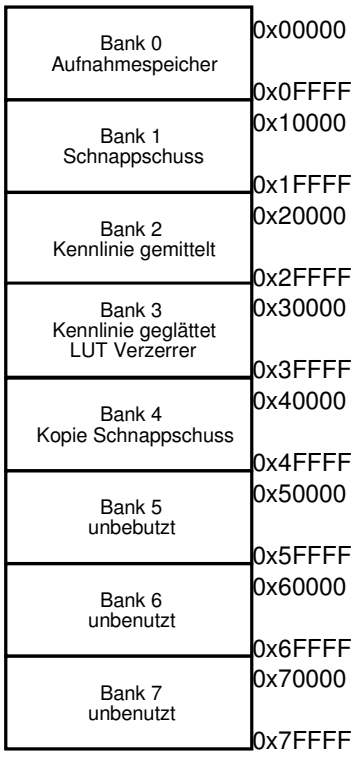


Abbildung 3-1: Speicherdiagramm SRAM

[3] Look up Table

3.2. Anbindung des Speichers

Da das verwendete FPGA nicht über genügend Speicherbits zur Ablage der gesamten LUT im FPGA verfügt, kommt zur Speicherung der LUT und anderer speicherintensiver Daten ein externes SRAM mit 1 MByte Kapazität zum Einsatz. Der Speicher verwendet Speicherworte mit einer Breite von 16 Bit, es stehen also 524288 Speicherplätze zur Verfügung. Um die Verwaltung des Speichers zu erleichtern wird dieser in 8 Bänke zu jeweils 128 kByte eingeteilt. Die Adressen der Speicherplätze setzen sich aus der Speicherbank, welche mit 3 Bit kodiert wird und dem Tabellenindex, welcher mit 16 Bit kodiert ist, zusammen.

3.3. Implementierung des Verzerrers

Der Verzerrer besteht im Wesentlichen aus einer Look up Table. Die Bank 3 wird dabei für die LUT der Vorverzerrung reserviert. Zum Auslesen der LUT ist es nur notwendig die, richtige Bank zu selektieren und den Eingangswert als Adresse anzulegen. Ein Lesezugriff auf den Speicher gibt dann direkt ein 16 Bit breites Ergebnis zurück.

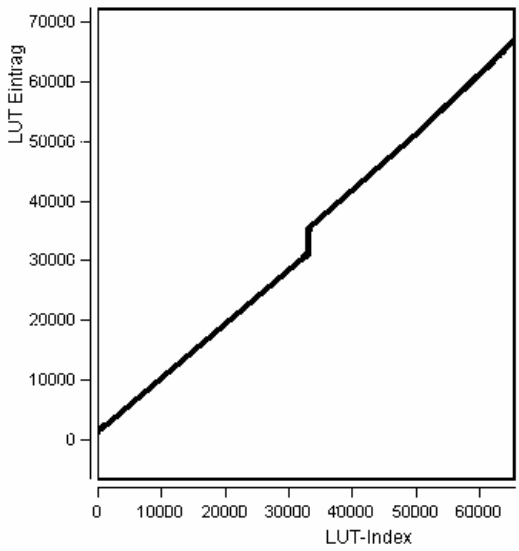


Abbildung 3-2: LUT Klasse B Verstärker

3.4. Adaption

Im Adaptionsblock findet die Anpassung des Verzerrers an das aktuelle Verstärkerverhalten statt. Hierfür stellt die Adaption dem Verzerrer immer eine aktuelle Look up Table bereit. Da der Berechnungsprozess relativ aufwändig ist, kann die Anpassung nicht bei jedem ausgegebenen Abtastwert erneut durchgeführt werden. Aus diesem Grund wird der zeitintensive Teil der Adaption zyklisch in festen Zeitabständen durchgeführt. Zur Berechnung der neuen LUT sind verschiedene Schritte notwendig, die nacheinander in unterschiedlichen Blöcken ausgeführt werden. Die Funktion und der Aufbau dieser Blöcke soll nun genauer betrachtet werden.

3.4.1 Verzögerungselement

Das Verzögerungselement ist der eigentlichen Adaption vorgelagert und arbeitet unabhängig von dessen Berechnungszyklen. Es muss dafür sorgen, dass am Adaptionsblock immer ein zusammengehörendes Wertepaar aus Ausgabewert und Rückkopplungswert anliegt. Hierzu ist es unbedingt notwendig, die durch den Verstärker, wie auch die DA- und AD-Wandler verursachte Verzögerung auszugleichen. Würde man das Ausgangs- und Rückkopplungssignal direkt an die Adaption geben, käme es zu einer Phasenverschiebung zwischen beiden Signalen. Die aufgenommene Kennlinie und damit auch die daraus berechnete Umkehrfunktion würden sich ähnlich einer Lissajous-Figur öffnen.

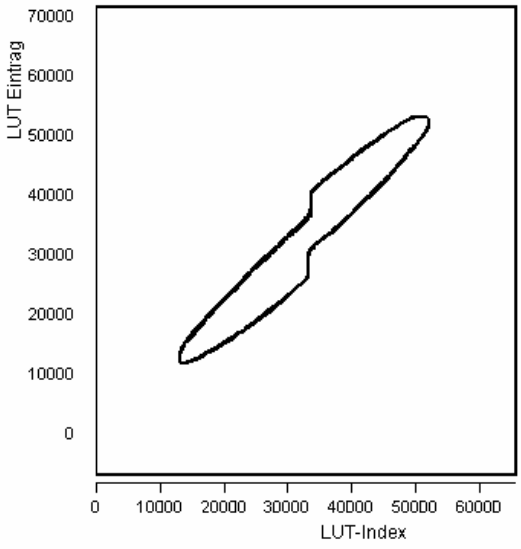


Abbildung 3-3: Umkehrfunktion bei falsch eingestellter Verzögerung

Um dies zu verhindern, muss das Ausgabesignal um exakt die gleiche Zeit verzögert werden, wie es für das Durchlaufen der DA-, AD-Wandler und des Verstärkers benötigt. Da es sehr unwahrscheinlich ist, dass diese Zeitspanne auf ein Vielfaches der Samplezeit fällt, muss dieses Verzögerungselement auch zwischen zwei Samples interpolieren können.

Um die Schaltung einfacher an andere Verstärker anpassen zu können, ist die Verzögerung dieses Verzögerungsements zusätzlich noch im Betrieb variierbar.

3.4.2 Aufnahmespeicher

Der Aufnahmespeicher arbeitet wie auch der Block „Delay“ unabhängig von den Berechnungszyklen der Adaption. Er stellt eine Tabelle dar, in der die an der Adaption ankommenden und an der Diagonalen x = y gespiegelten Wertepaare einsortiert werden. Diese Tabelle ist bereits aufgebaut wie die spätere LUT zur Vorverzerrung, enthält aber nur Rohdaten der Umkehrfunktion, die erst nach einer weiteren Verarbeitung verwendet werden können. Zur Unterdrückung von evtl. auftretenden Störungen wird aus dem bereits gespeicherten Wert und dem neu ankommenden Wert der Mittelwert gebildet. Der gespeicherte Wert ergibt sich somit aus dieser Formel:

$$V = \frac{1}{2}x_t + \frac{1}{4}x_{t-t_s} + \frac{1}{8}x_{t-2t_s} + \ldots$$

Dies entspricht einer Tiefpassfilterung mit den vorherigen Werten auf genau diesem Speicherplatz, was eine gute Unterdrückung von Ausreißern gewährleistet.

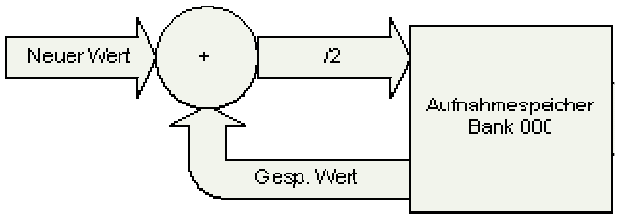


Abbildung 3-4 Aufnahmespeicher

Dieses Verfahren hat allerdings das Problem, dass Werte, die sich außerhalb des aktuellen Aussteuerungsbereiches befinden, nicht aktualisiert werden. Sind diese Speicherinhalte aber durch Störungen zustande gekommen, oder hat sich das Verstärkerverhalten zwischenzeitlich verändert, beeinträchtigen diese fehlerhaften Werte die weitere Bearbeitung der Unkehrfunktion enorm. Aus diesem Grund ist es äußerst wichtig, diese Werte aus dem Speicher zu eliminieren. Hierfür wird die Bitkombination 0xFFFF als Markierung für „Inhalt ungültig“ reserviert. Speicherplätze, die diesen Wert enthalten, dürfen nicht für Berechnungen herangezogen werden. Soll ein als ungültig markierter Speicherplatz mit einem gültigen Wert beschrieben werden, wird dieser Wert direkt ohne Mittelwertbildung abgelegt. Parallel zur Aktualisierung des Speicherinhaltes mit neu ankommenden Werten läuft ein so genannter „Invalid Pointer“ in äquidistanten Schritten langsam durch den Speicher und setzt deren Inhalt auf ungültig. Innerhalb des Aussteuerungbereiches werden die dadurch entstehenden Lücken schnell wieder gefüllt, außerhalb werden die fehlerhaften Werte aus dem Speicher gelöscht. So ist sichergestellt, dass der Speicherinhalt immer aktuell bleibt.

3.4.3 Schnappschuss

Da sich die Daten im Aufnahmespeicher ständig ändern, ist es nicht möglich, diese Daten direkt weiterzuverarbeiten. Deshalb wird beim Start des Adaptionsprozesses ein Schnappschuss des Aufnahmespeichers in einen getrennten Speicherbereich (Speicherbank 1) kopiert. Diese Kopie kann nun soweit bearbeitet werden, bis sie einen kompletten Wertesatz der Umkehrfunktion enthält. Zusätzlich wird noch eine weitere Kopie dieses Schnappschusses erstellt. Diese Kopie spielt für die eigentliche Adaption keine Rolle, sie stellt nur die unbearbeiteten Daten, die der Verarbeitung zugrunde lagen, bereit. So können die ursprünglichen Daten mit den verarbeiteten Daten zu Analysezwecken verglichen werden.

3.4.4 Interpolation

Wie vorher schon angedeutet, entstehen durch den „Invalid Pointer“ Lücken in der aufgezeichneten Gegenfunktion. Diese Lücken müssen nun wieder geschlossen werden. Da diese Lücken aufgrund der ständigen Aktualisierung des Aufnahmespeichers nicht sehr groß werden, ist es ausreichend, sie durch lineare Interpolation zu füllen. Eine lineare Interpolation ist nur zwischen zwei gültigen Punkten möglich, über die Grenzen der aktuellen Aussteuerung hinaus können so also keine neuen Werte erzeugt werden. Für die Implementierung der linearen Interpolation wird der Bresenham–Algorithmus eingesetzt. Dieser Algorithmus ist in der Computergrafik zur Rasterung von Geraden weit verbreitet. Er bietet den Vorteil, dass lediglich Additionen, Subtraktionen sowie Schiebeoperationen für dessen Umsetzung benötigt werden. Auf komplexere Operationen, wie Multiplikation, Division oder Gleitkommaberechnungen kann so komplett verzichtet werden. Aus diesem Grund ist eine äußerst effiziente Implementierung dieses Algorithmus auf einem FPGA möglich.

3.4.5 Extrapolation

Nach der Interpolation ist die Umkehrfunktion innerhalb des Aussteuerungsbereiches lückenlos. Darüber hinaus sind aber noch keine gültigen Werte vorhanden. Es ist aber besonders wichtig, auch über die aktuelle Aussteuerung hinaus, gütige Werte bereitstellen zu können, da sonst bei einer Erhöhung der Signalamplitude fehlerhafte Werte ausgegeben würden, die beträchtliche Störungen verursachen würden. Da der Kennlinienverlauf unbekannt ist, bleibt nur die Möglichkeit, den bereits bekannten Kennlinienabschnitt linear fortzusetzen. Aufgrund der oben beschriebenen Eigenschaften bietet sich auch hier der Bresenham–Algorithmus an. Da aber kein Zielpunkt festgelegt werden kann, mussten einige Modifikationen gegenüber dem Originalalgorithmus durchgeführt werden. Diese beziehen sich hauptsächlich auf die Berech-

nung der Steigung, das Einhalten der Grenzen der Tabelle, sowie das Verhindern von Wertebereichsüberschreitungen.

3.4.6 Mittelung und Glättung

Die Mittelung der berechneten Umkehrfunktion wird durchgeführt, um eine weitere Unterdrückung von Störungen und evtl. von ihnen verursachten Fehlern in der Interpolation zu erreichen. Hierfür wird die zuvor neu berechnete Umkehrfunktion mit dem vom Aufnahmespeicher her bekannten Verfahren mit dem Ergebnis des vorherigen Mittelungsdurchgangs gemittelt und in einem speziellen Speicherbereich für diese gemittelte Kennlinie abgelegt. Anschließend sorgt eine Glättung der resultierenden Umkehrfunktion für einen stetigen Verlauf dieser Funktion. Diese Glättung wird durch ein gewichtetes Mittelwertfilter realisiert. Um unnötige Multiplikationen und Divisionen zu vermeiden, werden alle Gewichtsfaktoren der zur Berechnung herangezogenen Nachbarwerte aus Potenzen von Zwei gebildet. Auch die Summe der Gewichtsfaktoren muss einer Potenz von Zwei entsprechen. Das Ergebnis jeder Glättungsberechnung muss in einem neuen Speicherbereich abgelegt werden, da sonst die Berechnungsgrundlage des nächsten Wertes überschrieben würde. Nach dem Glättungsvorgang ist der Adaptionsprozess abgeschlossen und die nun errechnete Umkehrfunktion kann an den Verzerrer übergeben werden.

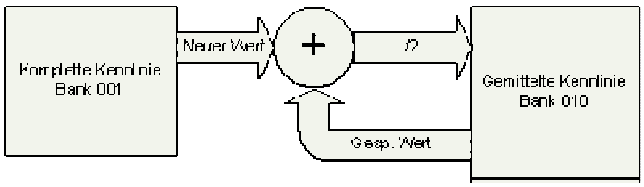


Abbildung 3-5: Mittelung mit anschließender Glättung

4. Voraussetzungen zum Einsatz der Schaltung

Dieser Ansatz zur digitalen Signalvorverzerrung stellt einige wichtige Anforderungen an den angeschlossenen Verstärker. Die wesentlichste Anforderung besteht darin, dass der angeschlossene Verstärker als gedächtnislos betrachtet werden kann. Dies bedeutet, dass der Signalverlauf keinerlei Auswirkung auf die Kennlinie des Verstärkers haben darf. Dies schließt insbesondere den Frequenzgang des Verstärkers ein. Ist die Verstärkung zu stark von der Frequenz abhängig, kann der Verstärker nicht mehr in guter Näherung als gedächtnislos betrachtet werden und ist somit nicht für den Einsatz mit dieser Schaltung geeignet. Auch unbedingt zu beachten ist, dass alle eingesetzten Komponenten, wie beispielsweise Filter, eine von der Frequenz unabhängige Signallaufzeit haben müssen. Dies bedeutet insbesondere, alle eingesetzten Filter müssen einen linearen Phasenverlauf haben, wie dies Beispielsweise bei Bessel-Filtern der Fall ist. Sind andere Filterarten in einer der im Signalweg enthaltenen Komponenten enthalten, müssen diese unbedingt angepasst werden. Dies kann beispielsweise bei den Ausgangsfiltern von Klasse D Verstärkern der Fall sein, da diese üblicherweise als Tschebyscheffilter ausgelegt werden.

5. Messungen

Zur Verifizierung der Schaltung wurden verschiedene Messungen durchgeführt. Zur Erstellung der Messreihen wurde ein Klasse B Verstärker an die Schaltung angeschlossen. Als Testsignal kam ein Sinussignal mit der angegebenen Frequenz zum Einsatz. Den weitaus größten Anteil am Klirrfaktor verursacht die dritte Harmonische der Grundschwingung. Wie in Tabelle 1 zu sehen ist, konnte der Klirrfaktor durch die aufgebaute Schaltung wesentlich verbessert werden. Der verbleibende Klirrfaktor konnte durchgängig auf Werte um 0,3% gedrückt werden, womit der Verstärker nun als Audioverstärker eingesetzt werden kann.

Frequenz	Klirrdämpfung ohne Vorverzerrung/dB	Klirrfaktor ohne Vorverzerrung/%	Klirrdämpfung mit Vorverzerrung/dB	Klirrfaktor mit Vorverzerrung/%
100Hz	-18,7	11,61	-48,8	0,36
250Hz	-19,8	10,23	-52	0,25
500Hz	-20	10,00	-51,6	0,26
1kHz	-18,8	11,48	-51,6	0,26
2kHz	-22	7,94	-54	0,20
3,5kHz	-24,8	5,75	-53,2	0,22
5kHz	-22,8	7,24	-50	0,32
7.5kHz	-22,6	7,41	-50	0,32

Tabelle 1: Messergebnisse Klasse B Verstärker mit und ohne Vorverzerrung

6. Zusammenfassung und Ausblick

6.1. Zusammenfassung

In dieser Arbeit wurde ein System realisiert, das eine Linearisierung eines Audioverstärkers mittels digitaler Signalvorverzerrung umsetzt. Weiterhin kann sich die realisierte Schaltung während des laufenden Betriebs selbständig an das aktuelle Verstärkerverhalten anpassen. Zur Realisierung dieses Systems wurde zunächst eine AD/DA-Wandlerplatine entwickelt und aufgebaut, welche auf allen Kanälen bei 16 Bit Auflösung eine Samplingrate von 200 kSamples/s bietet. Zur Realisierung der digitalen Signalverarbeitung kommt ein FPGA zum Einsatz. Der komplette VHDL-Entwurf wurde im Rahmen dieses Projektes entworfen. Um die Analyse der Schaltung zu erleichtern, wurde außerdem noch eine Anbindung der Schaltung an den PC implementiert. Zur Auswertung der Daten steht ein ebenfalls im Rahmen dieser Arbeit entwickeltes grafisches PC-Programm bereit (Abbildung 6-1). Wie Messreihen belegen, bewirkt das System eine deutliche Verbesserung des Klirrverhaltens eines Klasse B Verstärkers.

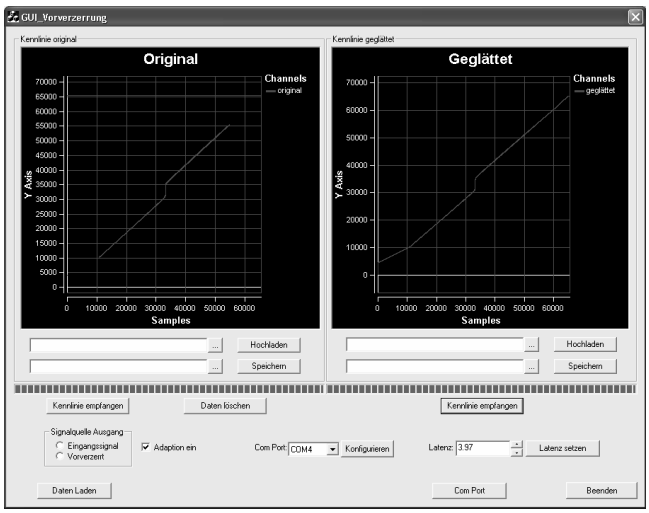


Abbildung 6-1: Grafisches Programm zur Analyse der Schaltungsparameter

6.2. Ausblick

Beim vorliegenden System handelt es sich um einen Prototyp, der die technische Machbarkeit eines solchen Systems demonstriert. Natürlich sind noch etliche Verbesserungen des Systems denkbar. Zum einen ist es für die Qualität der Adaption von entscheidender Bedeutung, möglichst geringe Verzerrungen auf dem Rückkopplungskanal zu erreichen. Aus diesem Grund können weitere Optimierungen der AD/DA-Wandlerplatine eine direkte Verbesserung in der Qualität der Vorverzerrung bringen. Außerdem wird die am Block „Delay“ zum Ausgleich der Verzögerung des Verstärkers eingestellte Verzögerung manuell vorgegeben. Wünschenswert wäre hier noch eine automatische Anpassung an die individuelle Verzögerung des angeschlossenen Verstärkers. Möglich wäre hier beispielsweise eine Berechnung der Verzögerung mittels Kreuzkorrelation. Auch könnten noch weit komplexere Verfahren zur Adaption eingesetzt werden. Allerdings sind wesentlich komplexere Berechnungsverfahren auf einem FPGA nur schwer mit einem vertretbaren Einsatz von Logik–Elementen implementierbar. Um komplexe Adaptionsalgorithmen effizient einsetzten zu können, müsste dann auf einen Embedded Prozessor, wie zum Beispiel Nios oder Sirius Prozessor, zurückgegriffen werden. Als weitere Verbesserungsmöglichkeit kann noch die Entwicklung einer Vorverzerrung für gedächtnisbehaftete Verstärker genannt werden. Für die Implementierung könnten beispielsweise Volterra-Reihen als Grundlage dienen. Allerdings ist die Implementierung einer nicht gedächnislosen Signalvorverzerrung ungleich aufwändiger als bei einem gedächtnislosen Ansatz.

Es ist auch möglich, die Ansteuerung des Leistungsteils eines Klasse-D-Verstärkers in das FPGA zu verlagern. Da die hierfür notwendigen Signale reine Digitalsignale sind, könnte so der DA-Wandler eingespart werden.

7. Literatur

[1] Nazim Ceylan, „Linearization of power amplifiers by means of digital predistortion”, Dissertation, Universität Erlangen-Nürnberg, Technische Fakultät, 2005

[2] Altera Application Note an314, “Digital Predistortion Reference Design“, 2003, http://www.altera.com/literature/an/an314.pdf

[3] Patrick Robin, Suk Keun Myoung, Dominique Chaillot, Young Gi Kim, Ayub Fathimulla, Jeff Strahler, Steven Bibyk, „Frequency-Selective Predistortion Linearization of RF Power Amplifiers“, IEEE Transactions on Microwave Theory and Techniques, Vol. 56, No. 1, January 2008

[4] Texas Instruments Application Note SBFA001A, “FilterPro MFB and Sallen-Key Low-Pass Filter Design Program”, 2001, http://focus.ti.com/lit/an/sbfa001a/sbfa001a.pdf

Ein rauscharmer Verstärker für Infrarot-Signale

Gerhard Forster

Hochschule Ulm, Prittwitzstraße 10, 89075 Ulm

forster@hs-ulm.de

Hintergrund dieses Beitrags ist ein Signalerfassungssystem zur Spektralanalyse von Infrarotlicht. Als kritischer Teil eines solchen Systems wird die monolithische Integration von bis zu 64 Verstärkerkanälen angesehen. Der Beitrag befasst sich mit der Entwicklung eines rauscharmen Verstärkers für zunächst 16 Kanäle, deren Ausgangssignale abgetastet und über einen Multiplexer seriell ausgegeben werden. Die Gesamtverstärkung jedes Kanals beträgt 70 dB und wird ohne externe Bauelemente erzielt. Ausgehend vom Systemkonzept wird der Schaltungsentwurf unter Berücksichtigung des zeitdiskreten Rauschverhaltens vorgestellt. Der Layoutentwurf eines Testchips mit 2 Kanälen in einer 0,35-µm-CMOS-Technologie wird präsentiert. Anhand der erzielten Messergebnisse wird die Realisierbarkeit eines Chips mit 16 Kanälen nachgewiesen. Seine Eingangsempfindlichkeit wird 1,5 µV bei einer Stromaufnahme von weniger als 10 mA betragen.

1. Einleitung

Infrarotstrahlung wird heute in vielfältiger Weise in Wissenschaft und Technik genutzt. Ihr Einsatz reicht von der Astronomie und der Medizin bis hin zu technischen Anwendungen in der Chemie, der Elektronik und der Verkehrstechnik [1,2]. Die technischen Einrichtungen lassen sich unterscheiden in passive und aktive Systeme. In passiven Systemen, wie z.B. der Thermografie in der Medizin, wird die Wärmestrahlung von Körpern ausgewertet. In aktiven Systemen wird Infrarotstrahlung ausgesendet und dessen Reflexion oder Transmission ausgewertet. Beispiele herzu sind die Füllstandsmessung oder die chemische Analyse von Gasen. Hintergrund der vorliegenden Arbeit ist die Gasanalyse mit Hilfe der Molekülspektroskopie [3] im mittleren Infrarotwellenlängenbereich $\lambda \approx 5\ \mu\mathrm{m}$. Um kurze Messzeiten zu erreichen, müssen die unterschiedlichen Spektralanteile parallel ausgewertet werden.

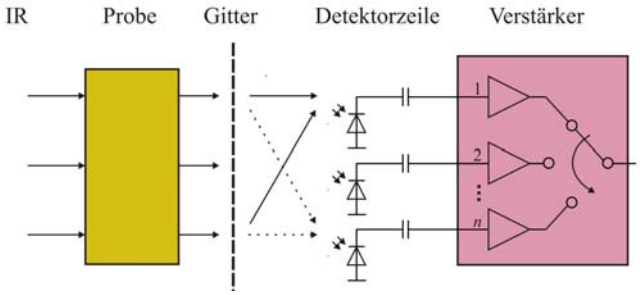


Bild 1: Schematische Darstellung einer Anordnung zur Infrarotspektroskopie

2. Systemkonzept

Das Konzept einer Anordnung zur Absorptionsspektroskopie schematisch in Bild 1 dargestellt. Eine Gasprobe wird von breitbandigem Infrarotlicht durchstrahlt. Das transmittierte Licht wird mittels eines Gitters in seine spektralen Bestandteile zerlegt. Zur Vermeidung der Absorption werden anstelle des transmittierenden Gitters in der Regel reflektierende Gitter, sog. Konkavgitter, verwendet. Die räumlich getrennten Spektralanteile werden von Detektoren in elektrische Signale umgewandelt. Im Wellenlängenbereich um 5 µm handelt es sich typischerweise nicht um Fotodioden, sondern Fotoleiter, die mit einer Gleichspannung betrieben werden. Die optisch induzierte Modulation des Leitwerts ergibt das Nutzsignal, welches der wesentlich größeren Arbeitspunktspannung überlagert ist. Nach Abtrennung des Gleichanteils muss das Nutzsignal jedes einzelnen Detektors verstärkt werden. Im vorliegenden Fall sollte eine Zeile mit 64 Detektoren, gekühlt mit einem Peltier-Kühler, zum Einsatz kommen. Die Gesamtlänge dieser Detektorzeile beträgt nur etwa 3,5 mm.

Aus mechanischen, thermischen und elektrischen Gründen sollten die Verstärker so kompakt wie möglich an die Detektoren herangeführt und über Bonddrähte angeschlossen werden. Aufgabe der Studie war daher die Untersuchung der monolithischen Integrierbarkeit eines Verstärkers mit mindestens 16 Kanälen bei minimaler Pinzahl. Die verstärkten Signale sollten über einen Analogmultiplexer, der ein Abtast-Halte-Glied beinhaltet, zyklisch ausgegeben werden, so dass sie einem A/D-Umsetzer zugeführt werden können.

Tab. 1: Angestrebte Daten des Gesamtsystems

Detektorimpedanz	$R_s = 15\ \mathrm{k\Omega}$
Untere Grenzfrequenz	$f_u = 50\ \mathrm{Hz}$
Obere Grenzfrequenz	$f_o \geq 2\ \mathrm{kHz}$
Max. Detektorspannung	$u_{S,\max} = 370\ \mu\mathrm{V_{eff}}$
Linearer Dynamikbereich	$\Delta u_S > 46\ \mathrm{dB}$
Nullpunktfehler am Ausgang	$u_o < 30\ \mathrm{mV}$
Übersprechen	$u_x / u_y < 50\ \mathrm{dB}$
Verlustleistung je Kanal	$P < 2{,}5\ \mathrm{mW}$

Tab. 2: Zielspezifikation für den Verstärker

Abtastperiode	$T = 125\ \mu\mathrm{s}$
Kleinsignalverstärkung	$V_{ges} = 70\ \mathrm{dB} \pm 1\ \mathrm{dB}$
Untere Grenzfrequenz	$f_u = 50\ \mathrm{Hz}$
Obere Grenzfrequenz	$f_o \geq 2\ \mathrm{kHz}$
Eingangswiderstand	$R_i \geq 200\ \mathrm{k\Omega}$
Ausgangswiderstand	$R_o < 1\ \mathrm{k\Omega}$
Eingangsoffsetspannung	$U_{OS} \leq 9{,}5\ \mu\mathrm{V}$
Äquivalentes Eingangsrauschen	$e_n \leq 15{,}8\ \mathrm{nV}/\sqrt{\mathrm{Hz}}$
Stromaufnahme je Kanal	$I_{DD} \leq 750\ \mu\mathrm{A}$

Als Zielspezifikation für das Gesamtsystem wurden die in Tab. 1 zusammengefassten Daten zugrunde gelegt.

Als maximale Detektorspannung ist in der Praxis durchaus ein Wert bis zu 10 $\mathrm{mV_{eff}}$ zu erwarten, so dass sich der gesamte Dynamikbereich auf über 70 dB erhöht. Diese Spannung darf aber außerhalb des linearen Dynamikbereichs des Verstärkers liegen. Der Verstärker darf in diesem Fall jedoch keine Erholzeiten aufweisen, die den Frequenzbereich beeinträchtigen. Zieltechnologie für den Verstärker ist ein 0,35 µm-Stan-dard-CMOS-Prozess mit $U_{DD} = 3{,}3\ \mathrm{V}$. Der hohe Dynamikbereich erfordert einen Rail-to-Rail-Ausgang und geringes Eigenrauschen. Die geringe Signalspannung erfordert darüber hinaus eine hohe Verstärkung, verbunden mit geringer Offsetspannung. Mit einer angenommenen Restspannung $\Delta U_o = 30\ \mathrm{mV}$ erhält man für die Kleinsignalverstärkung

$$V \geq \frac{\frac{U_{DD}}{2} - \Delta U_0}{\sqrt{2} \cdot u_{S,\max}} = \frac{1{,}65\ \mathrm{V} - 0{,}03\ \mathrm{V}}{\sqrt{2} \cdot 370\ \mu\mathrm{V}} = 3096$$

und für die Offsetspannung

$$\underline{\underline{U_{OS}}} = \frac{u_0}{V} = \frac{30\ \mathrm{mV}}{3160} = \underline{\underline{9{,}5\ \mu\mathrm{V}}}$$

Die letztere Anforderung ist besonders anspruchsvoll, da keinerlei externe Bauelemente im Signalweg erwünscht sind. Die Abtastrate f_{MUX} des Multiplexers sollte aus Gründen der Verlustleistung möglichst gering sein. Ihre Untergrenze wird jedoch zum einen bestimmt durch das Abtasttheorem, wonach $f_{\mathrm{MUX}} > 2f_o = 4\ \mathrm{kHz}$ gefordert ist und zum anderen durch die Forderung, dass die Durchlassdämpfung bei der oberen Grenzfrequenz f_o weit unter 3 dB bleibt:

$$A = \mathrm{sinc}\left(\frac{\pi \cdot f}{f_{MUX}}\right) << 3\ \mathrm{dB}\ \text{bei}\ f = f_o$$

Daraus folgt $f_{MUX} >> 4{,}6\ \mathrm{kHz}$

Aus systemtechnischen Gründen wurde $f_{\mathrm{MUX}} = 8\ \mathrm{kHz}$ gewählt. Innerhalb der Abtastperiode $T = 125\ \mu\mathrm{s}$ werden dann jeweils 16 Kanäle zyklisch ausgelesen. Aus Abtastrate und Dynamikbereich lässt sich die maximal zulässige Rauschspannungsdichte, bezogen auf die Detektorquelle zu

$$e_{n,ges,\max} = 20{,}5\ \frac{\mathrm{nV}}{\sqrt{\mathrm{Hz}}}$$

bestimmen. Wegen des Eigenrauschens des Detektors verbleibt dann für den Verstärker eine Eingangsrauschspannungsdichte von

$$\underline{\underline{e_{n,Verst}}} \leq \sqrt{e_{n,ges,\max}^2 - 4kT \cdot R_S} = \underline{\underline{15{,}8\ \frac{\mathrm{nV}}{\sqrt{\mathrm{Hz}}}}}$$

Mit weiteren Anforderungen resultiert daraus für den Verstärker die in Tab. 2 dargestellte Zielspezifikation.

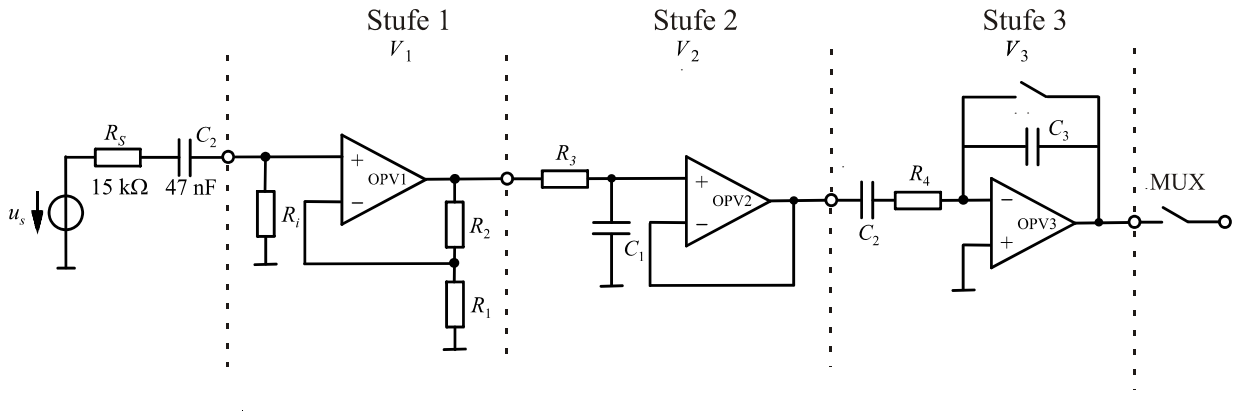


Bild 2: Schaltungstopologie

3. Schaltungskonzept

3.1. Schaltungstopologie

Die Grundkonzeption basiert auf der Idee einer zweistufigen Anordnung, bestehend aus einem rauscharmen Verstärker und einem integrierenden Verstärker sowie einem dazwischen geschalteten Hochpass zur Abtrennung der Offsetspannung. Wegen der Abtastvorgänge und zur Begrenzung der Rauschbandbreite ist ein zusätzliches Tiefpassfilter erforderlich. Daraus resultiert die in Bild 2 dargestellte Schaltungstopologie mit 3 Stufen.

Der nichtinvertierende Verstärker in Stufe 1 erlaubt einen hohen und definierten Eingangswiderstand. Stufe 2 übernimmt die Funktion des Tiefpassfilters mit Pufferung. Der integrierende Verstärker in Stufe 3 akkumuliert das Signal zwischen den Abtastzeitpunkten und erhöht damit den Signal/Rauschabstand. C_2 und R_4 bilden gleichzeitig die Hochpassfunktion zur Elimination der Offsetspannung der davor liegenden Stufen. Die Schaltungstopologie weist insgesamt 2 dominante Hochpässe und 3 dominante Tiefpässe auf:

$$f_{u1} = \frac{1}{2\pi(R_i + R_s)C_s}$$

$$f_{u2} = \frac{1}{2\pi R_4 C_2}$$

$$f_{o1} = f_{-3\mathrm{db}}(\mathrm{OPV1})$$

$$f_{o2} = \frac{1}{2\pi R_3 C_1}$$

$$f_{o3} = 0{,}44 \cdot f_{\mathrm{MUX}}$$

Die Gesamtverstärkung im Durchlassbereich sowie die obere und untere Grenzfrequenz sind nun in geeigneter Weise auf die einzelnen Stufen zu verteilen.

3.2. Aufteilung der Verstärkungsfaktoren im Durchlassbereich

Hinsichtlich des Rauschens und der Offsetspannung wäre es optimal, den gesamten Verstärkungsfaktor V_{ges} mit der ersten Stufe zu realisieren, weil dann die Zusatzrauschspannungen der anderen Stufen und die Offsetspannung der letzten Stufe den geringsten Einfluss hätten. Der Ausgangsspannungshub der ersten Stufe wäre dann allerdings wegen deren Offsetspannung stark eingeschränkt ($U_{\mathrm{OS}} > 0{,}5\ \mathrm{mV}$ würde die erste Stufe bereits in die Begrenzung führen!) und er könnte durch die nachfolgenden Stufen nicht regeneriert werden. Hier muss ein Kompromiss gefunden werden, der sicherstellt, dass der über die Versorgungsspannung U_{DD} zur Verfügung stehende Ausgangsspannungsbereich bei gegebener Offsetspannung und Restspannung der Verstärker maximal ausgeschöpft werden kann. Dies ist in Bild 3 für die Stufen 1 und 3 in Abhängigkeit von Verstärkungsfaktor V, Offsetspannung U_{OS} und Restspannung ΔU_o schematisch dargestellt. Der Einfluss der mittleren Stufe wird vernachlässigt. Er kann in Stufe 1 mit einbezogen werden.

Für die beiden Stufen findet man als maximale Scheitelspannungen

$$\hat{u}_{o1} = \frac{U_{DD}}{2} - \Delta U_{o1} - V_1 U_{OS1}$$

$$\hat{u}_{o3} = \frac{U_{DD}}{2} - \Delta U_{o3} - V_3 U_{OS3}$$

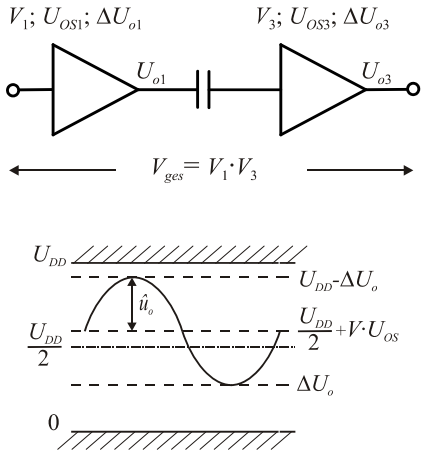


Bild 3: Zusammenhang zwischen Verstärkung, Offsetspannung, Restspannung und Aussteuerbereich

Um $\hat{u}_{o1}$ auf $\hat{u}_{o3}$ zu verstärken, benötigt man mindestens den Verstärkungsfaktor

$$V_3 = \frac{\hat{u}_{o3}}{\hat{u}_{o1}} = \frac{U_{DD} - 2\Delta U_{o3} + 2U_{OS} \cdot V_{ges}}{U_{DD} - 2\Delta U_{o1} + 2U_{OS3}}$$

Damit verbleibt für Stufe 1

$$V_1 = \frac{V_{ges}}{V_3}$$

Unter Annahme der folgenden Daten:

$$U_{DD} = 3{,}3\ \text{V}; \Delta U_{o1} = 200\ \text{mV};$$
$$\Delta U_{o3} = 50\ \text{mV}; U_{OS1} = 1\ \text{mV}; U_{OS3} = 2\ \text{mV}$$

erhält man $V_3 = 3{,}3$ und $V_1 = 957$. Gewählt wurden die Werte $V_3 = 4$ und $V_1 = 790$, um die Anforderung an das Verstärkungs-Bandbreite-Produkt des ersten Verstärkers etwas zu reduzieren. Für Stufe 2 wird $V_2 = 1$ gewählt, so dass keine Widerstände zur Beschaltung des Verstärkers benötigt werden.

3.3. Aufteilung der Grenzfrequenzen

Da die untere Grenzfrequenz f_{u2} nur mit großem Aufwand zu erreichen ist, wird diese auf 50 Hz festgelegt. Die untere Grenzfrequenz f_{u1} muss dann erheblich tiefer liegen, damit $f_u = 50$ Hz nicht wesentlich überschritten wird. Hier sind aber 10 Hz gut erreichbar.

Für die obere Grenzfrequenz wird aus Toleranzgründen $f_o = 2{,}2$ kHz gewählt. Die Durchlassdämpfung des Multiplexers mit 8 kHz Abtastfrequenz beträgt hier ca. 1 dB, so dass auch für die beiden anderen Tiefpässe eine Dämpfung von je 1 dB zulässig ist. Ihre Grenzfrequenzen f_{o1} und f_{o2} müssen daher jeweils oberhalb von 4,4 kHz liegen.

Tab. 3: Zielspezifikation für die Stufe 1

Verstärkung	$V_1 = 790$
Eingangswiderstand	$f_{u1} = 10\ \text{Hz}; C_S = 47\ \text{nF}$ $\Rightarrow R_i = 340\ \text{k}\Omega$
Obere Grenzfrequenz	$f_{o1} > 4{,}4\ \text{kHz}$
Äquivalente Eingangsrauschspannungsdichte	$e_n \le 15{,}8\ \frac{\text{nV}}{\sqrt{\text{Hz}}}$
Offsetspannung	$U_{OS} < 1\ \text{mV}$
Ausgangs-Restspannung (mit Last)	$\Delta U_o < 200\ \text{mV}$

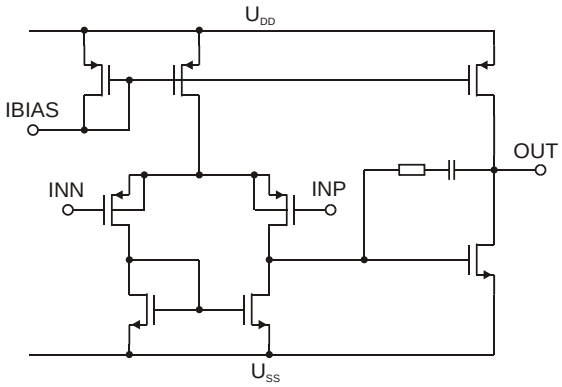


Bild 4: Rauscharmer Operationsverstärker (Miller-OTA)

4. Schaltungsentwurf und Simulation

4.1. Stufe 1

Da der maximale Wert des Koppelkondensators wegen der Baugröße auf $C_S = 47$ nF begrenzt ist, wird für $f_1 = 10$ Hz der hohe Eingangswiderstand $R_i > 340$ kΩ benötigt. Die Anforderungen an die Stufe 1 sind in Tab. 3 zusammengefasst.

Die Daten werden erreicht mit dem in Bild 4 dargestellten Miller-OTA. Wegen der hohen Rauschanforderungen und geringen Anforderungen an die Treiberfähigkeit enthält der Verstärker einen groß dimensionierten Differenzverstärker (Eingangstransistoren mit $W/L = 1215\ \mu\text{m}/10\ \mu\text{m}$) und eine klein dimensionierte zweite Verstärkerstufe, die gleichzeitig als Rail-to-Rail-Ausgangsstufe dient [4].

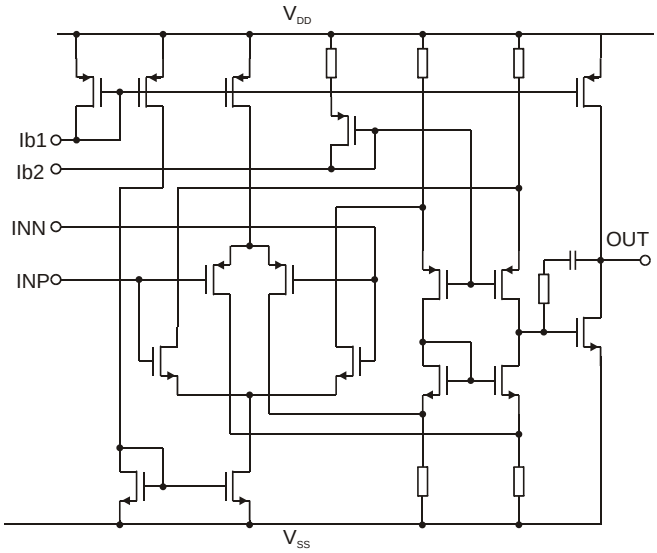


Bild 5: Operationsverstärker mit Rail-to-Rail-Eingangsstufe

Die Simulationsdaten dieses Verstärkers sind: $V_0 = 106$ dB; $f_T = 3{,}5$ MHz; $e_n = 14{,}2$ nV/SqrtHz und $I_{DD} = 84{,}5$ µA. Eingesetzt in Stufe 1 (siehe Bild 2) mit $R_1 = 2$ kΩ und $R_2 = 1{,}64$ MΩ wird eine Verstärkung von $V_1 = 57{,}8$ dB bei einem Gesamtrauschen einschließlich Detektor von $e_n = 22{,}8$ nV/SqrtHz erzielt.

4.2. Stufe 2

Über R_3 und C_1 wird lediglich die obere Grenzfrequenz f_{o2} festgelegt. Da die Absoluttoleranzen vorgehalten werden müssen, wird $f_{o2} = 6$ kHz angesetzt. Die Dimensionierung mit dem Ziel minimaler Chipfläche führt zu $R_3 = 2$ MΩ und $C_1 = 13{,}3$ pF. Der Operationsverstärker wird lediglich als Spannungsfolger beschaltet. Dies bedingt jedoch eine Phasenreserve von mindestens 60°, vor allem aber einen Rail-to-Rail-Eingang, weil die Eingangsspannung als Gleichtaktspannung wirkt. Hierzu wurde der in Bild 5 dargestellte Operationsverstärker entwickelt. Er besteht aus einer Eingangsstufe mit zwei komplementären Differenzverstärkern, einer gefalteten Kaskode und einer Rail-to-Rail-Ausgangsstufe.

Die Simulationsdaten dieses Operationsverstärkers sind:

$V_0 = 112$ dB, $f_T = 1{,}1$ MHz, $\varphi_R = 60°$, $U_{igl,\min} = 0$ V, $U_{igl,\max} = 3{,}3$ V, $I_{DD} = 80$ µA. Eingesetzt in Stufe 2 als Unity-Gain-Buffer zeigt er bei $R_L = 1$ MΩ lediglich eine Restspannung $\Delta U_o < 25$ mV.

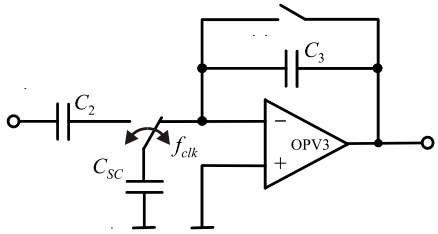


Bild 6: Prinzipschaltbild des SC-Integrators

4.3. Stufe 3

Aufgabe dieser Stufe ist die Funktionalität eines Integrierers mit Rückstellmöglichkeit (siehe Bild 2). Über die Integrationszeit soll die Verstärkung $V_3 = 4$ resultieren. Gleichzeitig soll mit C_2 und R_4 die untere Grenzfrequenz $f_u = 50$ Hz erreicht werden. Dies ist nur mit Hilfe einer Switched-Capacitor-Realisierung von R_4 möglich (Bild 6).

Die Verstärkung berechnet sich zu

$$V_3 = n \cdot \frac{C_{SC}}{C_3}$$

wobei n die Zahl der Taktperioden von f_{clk} bis zum Rücksetzzeitpunkt bedeutet. Für die untere Grenzfrequenz erhält man

$$f_u = \frac{f_{clk} \cdot C_{SC}}{2\pi \cdot C_2}$$

Um das Verhältnis C_{SC}/C_2 nicht zu klein machen zu müssen, sollte f_{clk} möglichst klein sein. Aus Gründen des Aliasing-Effekts besteht jedoch eine Untergrenze im Bereich von 100 kHz. Da im System ein Takt 16 x f_{MUX} für das zyklische Auslesen der Kanäle vorhanden ist, wird dieser Takt verwendet:

$$f_{clk} = 16 \cdot f_{MUX} = 128\,\mathrm{kHz}$$

Als Untergrenze für die Kondensatoren wird $C_{SC} = 0{,}4$ pF angesetzt. Damit ergibt sich $C_3 = 1{,}6$ pF und $C_2 = 150$ pF. Dieser Wert ist für eine Integration extrem groß. Er wir jedoch an dieser Stelle zunächst in Kauf genommen, um die Offsetanforderung mit ausreichender Wahrscheinlichkeit erfüllen zu können. Die Offsetspannung entsteht durch Ladungsinjektion und Clock-Feedthrough (auch an C_{SC}), deren Simulation nur begrenzt möglich ist. Die Realisierung des Integrators erfolgt mit der in Bild 7 dargestellten Schaltung. Sie wird mit einem parasiten-insensitiven Schal-

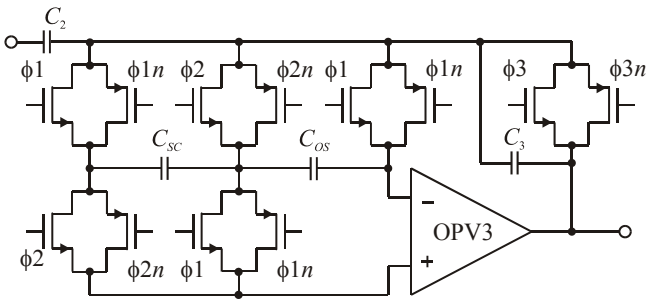


Bild 7: Switched-Capacitor-Integrator mit Rückstellschalter, Offsetkompensation und Koppelkondensator

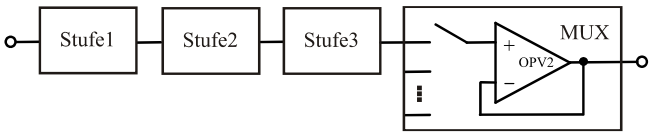


Bild 8: Simulationsmodell eines Verstärkerkanals

terschema betrieben und enthält einen zusätzlichen geschalteten Kondensator C_{OS} zur Offsetspannungskompensation [5]. Alle Schalter sind als Transmission Gates ausgeführt, die mit komplementären, nicht überlappenden Takten betrieben werden.

4.4. Gesamtsimulation

Die Stufen 1 – 3 hintereinander geschaltet und mit einem Abtast-Halteglied zur Nachbildung des Multiplexers versehen, spiegeln die gesamte Übertragungsfunktion eines Verstärkerkanals wider (Bild 8).

An diesem Modell wurden Transient-, AC- und Rauschsimulationen durchgeführt. Da es sich um ein zeitdiskretes System handelt, ist die klassische AC-Analyse nicht möglich. Aus diesem Grund wurden die AC- und die Rauschanalyse mit dem Simulator SpectreRF® [6,7] durchgeführt. Die Simulationsergebnisse sind in den Bildern 9 - 11 dargestellt.

Bild 9 zeigt das Zeitsignal an den Ausgängen der Stufen 1, 2, 3 und des Multiplexers für eine Eingangsamplitude von 100 µV und Bild 10 die entsprechenden Frequenzgänge. In Bild 11 ist das Leistungsspektrum des Ausgangsrauschens dargestellt. Die Simulation beinhaltet auch die Ansteuerschaltungen, z.B. für die Mittenspannungserzeugung und berücksichtigt sämtliche Effekte wie z.B. Nichtidealitäten der Schalter, SC-Rauschen und Faltung durch die Abtastvorgänge.

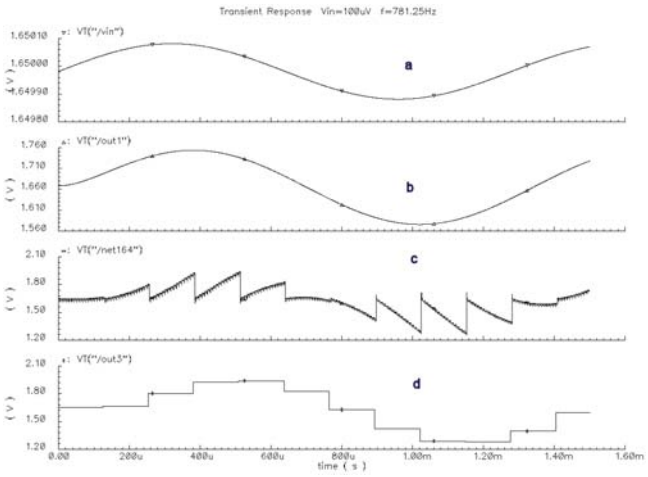


Bild 9: Zeitsignal: (a) Eingangssignal mit $\hat{u}_s = 100$ uV; (b) Ausgang Stufe 1; (c) Ausgang Stufe 3; (d) Ausgang MUX

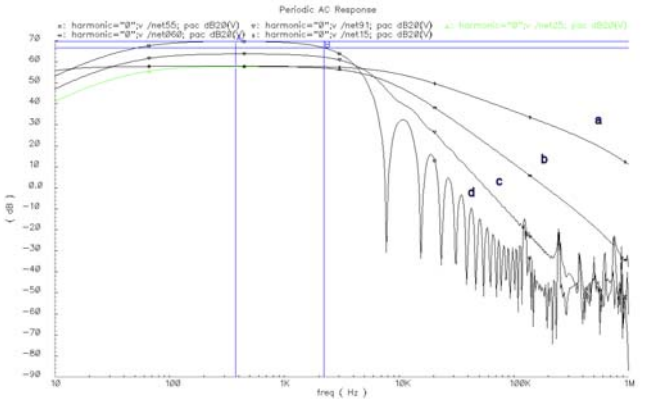


Bild 10: Verstärkung im Frequenzbereich: (a) bis zum Ausgang Stufe 1; (b) bis zum Eingang Stufe 3; (c) bis zum Ausgang Stufe 3; (d) bis zum Ausgang MUX

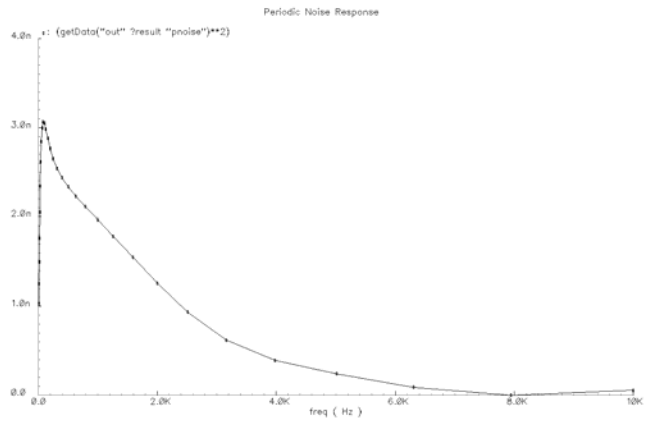


Bild 11: Quadratische Rauschspannungsdichte am Ausgang des MUX ($R_S = 50\ \Omega$)

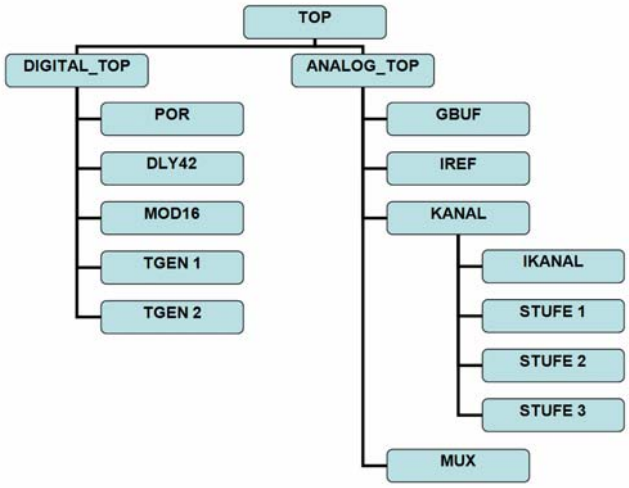


Bild 12: Hierarchischer Aufbau des Verstärker-ICs

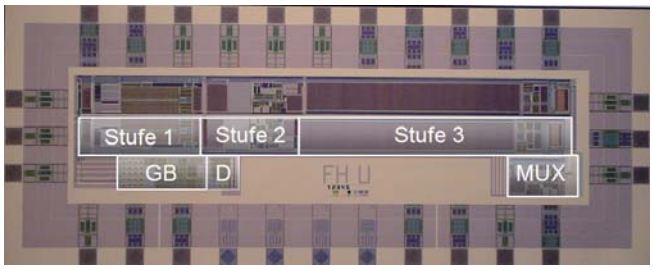


Bild 13: Layout des Testchips mit 2 Kanälen

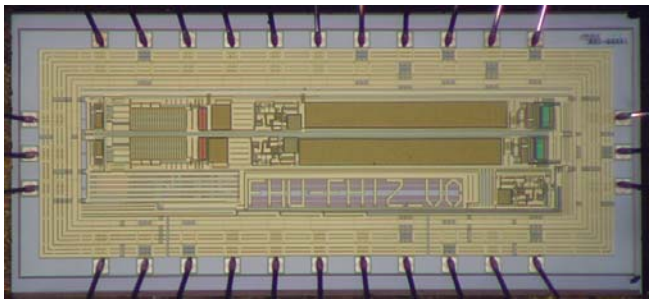


Bild 14: Chipfoto des Testchips mit 2 Kanälen

Die zu erwartende Rauschspannung am Ausgang beträgt danach

$$u_{o,r,eff} = \sqrt{\int_0^{8\,\mathrm{kHz}} \overline{u_o^2}\,\mathrm{df}} = 4{,}5\,\mathrm{mV_{eff}}$$

Dies entspricht einer äquivalenten Eingangsrauschspannung

$$u_{i,r,eff} = 1{,}5\,\mu\mathrm{V_{eff}}$$

5. Testchipentwurf und Messergebnisse

5.1. Layout

Der geplante Chip soll 16 der oben dargestellten Kanäle sowie weitere Komponenten zu deren Betrieb beinhalten. Die hierarchische Struktur ist in Bild 12 dargestellt. Jeder der 16 Kanäle mit den Stufen 1 – 3 beinhaltet einen Block IKANAL zur Stromversorgung. Gemeinsam für alle Kanäle ist eine weitere Stromversorgung IREF, ein Ground Buffer GBUF sowie der Multiplexer. Ein Digitalteil sorgt für die Ablaufsteuerung und liefert die nicht überlappenden Takte für die SC-Komponenten. Zu Testzwecken sollte aus Kostengründen zunächst nur ein Kanal, zusammen mit den weiteren Komponenten, realisiert werden. Damit auch das Kanal-Übersprechen untersucht werden kann, wurden 2 Kanäle eingesetzt.

Bild 13 zeigt einen Zwischenstand des Layouts mit zwei Kanälen. Das Längen/Breitenverhältnis der Kanäle ist sehr groß, damit sich mit 16 Kanälen immer noch ein annähernd quadratisches Chipmaß ergibt. Auffallend ist die große Fläche der Stufe 1, die aus dem großflächigen Differenzverstärker des rauscharmen OTAs resultiert. Besonders auffallend ist allerdings die Stufe 3, deren Größe hauptsächlich durch den Kondensator C_2 bestimmt wird. Bild 14 zeigt den bei AMS gefertigten Chip, eingebaut in ein 48-poliges Gehäuse.

5.2. Messungen

Mit den am Testchip durchgeführten Messungen konnten die Simulationsergebnisse weitgehend bestätigt werden. Bild 15 zeigt den Frequenzgang bis zum Ausgang des Multiplexers mit der Verstärkung $V = 69{,}2$ dB und Grenzfrequenzen $f_u = 55$ Hz sowie $f_o = 2{,}2$ kHz. Er stimmt sehr gut mit dem simulierten Frequenzgang in Bild 10 d) überein.

Die Eingangsrauschspannungsdichte bei $f = 1$ kHz beträgt $e_n = 21$ nV/SqrtHz einschließlich Detektor ($R_S = 15$ kΩ) und $e_n = 15$ nV/SqrtHz ohne Detektor ($R_S = 50$ Ω). In Bild 16 ist das Messergebnis des quadratischen Ausgangsrauschens dargestellt. Darin zeigt sich sehr gute Übereinstimmung mit dem Simulationsergebnis in Bild 11, womit nachgewiesen ist, dass die Simulation tatsächlich alle wesentlichen Rauscheinflüsse erfasst. Die weiteren Ergebnisdaten sind in Tab. 4 zusammengestellt.

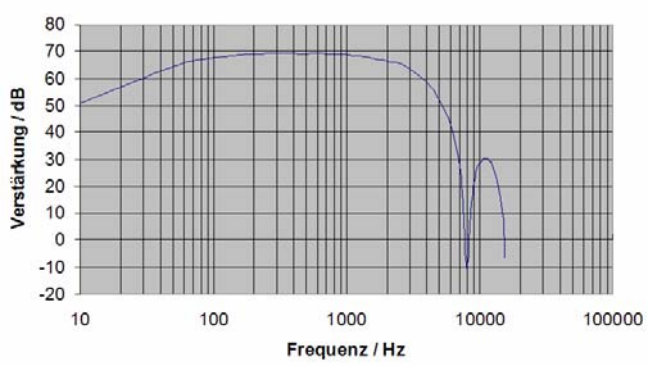


Bild 15: Gemessener Frequenzgang bis zum Multiplexer

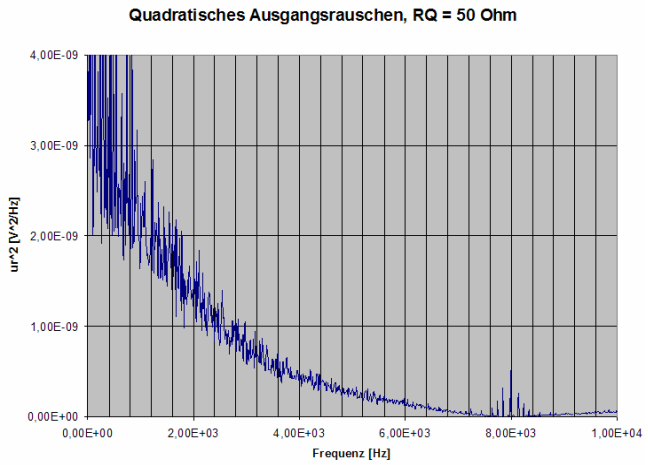


Bild 16: Gemessenes Rauschspektrum am Ausgang des Multiplexers

Tab. 4: Überblick über die erzielten Messergebnisse

Empfindlichkeit	1,5 µV
Verstärkung	69,2 dB
Untere Grenzfrequenz	55 Hz
Obere Grenzfrequenz	2,2 kHz
Kanalübersprechen	< 50 dB
Eingangsoffsetspannung	$+30 \pm 3{,}5$ µV
Stromaufnahme	< 400 µA/Kanal
Chipfläche	3,52 x 1,45 mm^2
Chipfläche/Kanal	2,73 x 0,22 mm^2

6. Zusammenfassung

Im Rahmen einer Machbarkeitsstudie wurde die Entwicklung eines Full-Custom-ICs zur Verstärkung von Signalen aus Infrarot-Detektoren vorgestellt. Ausgehend von den Systemdaten für die Infrarotspektroskopie wurde ein Systemkonzept für den Verstärker entwickelt. Nach Abbildung der Systemdaten auf die Spezifikation des Verstärkers erfolgte der Schaltungsentwurf. Hierzu wurden eine Reihe spezieller Schaltungskomponenten entwickelt, darunter ein rauscharmer Verstärker, ein Rail-to-Rail-Operationsverstärker sowie ein SC-Integrator mit Offsetkompensation. Mit diesen Komponenten erfolgte die komplette Schaltungsentwicklung für einen 16-kanaligen Verstärker. Zur Verifikation wurde ein Testchip mit 2 Kanälen und der zugehörigen Ansteuerschaltung auf der Basis des 0,35 µm-CMOS-Prozesses von AMS gefertigt und ausgetestet. Verstärkung, Grenzfrequenzen, Kanalübersprechen und Rauschen entsprechen weitestgehend den angestrebten und durch Simulation vorausgesagten Ergebnissen. Allein die angesetzte Offsetspannung von ± 9,5 µV konnte nicht erreicht werden. Sie zeigt einen einseitigen Sockel von 30 µV, der noch untersucht werden sollte. Da jedoch die Varianz gering ist, kann der Sockel in der Anwendung kompensiert werden. Damit steht der Weg für einen 16-kanaligen Chip offen. Die Gesamtfläche eines solchen Chips kann auf circa 4,5 x 3,5 mm^2, seine Stromaufnahme auf circa 7 mA geschätzt werden.

Die Entwicklung erfolgte unter Anwendung einer durchgängigen Design-Kette mit Cadence-Werkzeugen. Insbesondere konnte der Simulator SpectreRF erfolgreich für die Frequenzanalyse von zeitdiskreten Systemen eingesetzt werden. So wurden in der Rauschanalyse alle parasitären Effekte einschließlich der Nichlinearität der Transmission Gates, der endlichen Flankensteilheit der Taktsignale, sowie der Faltungsvorgänge erfasst. Auch die Gesamtschaltung mit 16 Kanälen konnte ungeachtet der hohen Komplexität erfolgreich simuliert werden. Damit kann nun der 16-kanalige Verstärker-IC mit großer Design-Sicherheit gefertigt werden.

Literatur

[1] H. Günzler, H.-U. Gremlich: IR-Spektroskopie: Eine Einführung. 4. Auflage. Wiley-VCH, Weinheim 2003

[2] H. Haken, H. C. Wolf: Molekülphysik und Quantenchemie, Springer 2006

[3] M. Ingels, M. Steyaert: Integrated CMOS Circuits for Optical Communications, Springer 2004

[4] R. J. Baker: CMOS Circuit Design, Layout, and Simulation, IEEE Press, 2005

[5] C. C. Enz, G. C. Temes: Circuit Techniques for Reducing the Effects of Op-Amp Imperfections: Autozeroing, Correlated Double Sampling, and Chopper Stabilization, Proceedings of the IEEE, vol. 84, No. 11, November 1996

[6] Affirma Spectre RF. Cadence Training Manual, December 1999

[7] K. Kundert: Simulating Switched-Capacitor Filters with SpectreRF, www.designers-guide.com, January 2004

Danksagung

Der Autor bedankt sich bei den Absolventen der Hochschule Ulm, Herrn Dipl.-Ing. A. Anzenhofer und Herrn Dipl.-Ing. (FH) A. Freudenreich, für die Simulation von Teilschaltungen, sowie bei Herrn Dipl.-Ing. (FH) Jon Ruiz Aguirre für die Arbeiten zur Schaltungsentwicklung und Gesamtsimulation. Weiterer Dank gilt Herrn Dipl.-Ing. (FH) A. Erni für die Unterstützung bei den Layoutarbeiten sowie den Herren Dipl.-Ing. (FH) D. Götz und Dipl.-Ing. (FH) M. Mödinger für die Durchführung messtechnischer Untersuchungen.

Das Projekt wurde gefördert aus Mitteln der baden-württembergischen Multiprojekt-Chip-Gruppe.

Implementierung eines MP3-Dekoders in VHDL

Felix Miller

Hochschule Heilbronn, Max-Planck-Straße 39, 74081 Heilbronn

miller@hs-heilbronn.de

Eine Implementierung eines MP3-Decoder-Cores in der Hardwarebeschreibungssprache VHDL wird vorgestellt.
Ziel dieser Entwicklung ist es, einen Dekoder für den Audiostandard MPEG-1 Layer III Audio zu entwickeln, welcher dedizierte, in VHDL beschriebene Module für die einzelnen Teilschritte der Dekodierung verwendet und somit ohne Prozessorkern auskommt.

Dabei kommen zum Einen handkodierte Module zum Einsatz (Bitstream-Parsing, Huffman-dekodierung, inverse Skalierung und Quantisierung), wie auch mittels High-Level-Synthese erzeugte Module, welche vom Fraunhofer Institut für integrierte Schaltungen zur Verfügung gestellt wurden.

1. Einleitung

MP3-Player und somit auch MP3-Dekoderbausteine sind heutzutage allgegenwärtig. Sowohl der Player mit kleinen Abmessungen und Flash-Speicher, aber auch größere Geräte mit Festplatten, durch welche die Plattensammlung mobil wird, sind aus dem täglichen Leben nicht mehr wegzudenken und haben andere mobile Audioplayer so gut wie vollständig verdrängt

Neben diesen „reinen“ MP3-Produkten befindet sich in immer mehr Geräten der Unterhaltungselektronik ein MP3-Dekoder als zusätzliches Leistungsmerkmal. Gleichgültig ob DVD-Player, Autoradio oder CD-Wecker, auf vielen Geräten findet sich das MP3-Logo und in all diesen Geräten befindet sich eine Hardware zur MP3-Dekodierung. Heute erhältliche MP3-Dekoderbausteine verwenden nach wie vor Prozessorkerne wie zum Beispiel ARM-Kerne. Teilweise sind diese noch um Instruktionen aus der DSP-Welt erweitert [2] um den anfallenden Rechenaufwand bewältigen zu können.

Solch eine Embedded-Processor-Implementierung bietet zwar den Vorteil eines geringen Entwicklungsaufwandes, da CDKs für solche Prozessorkerne beispielsweise von der Fraunhofer-Gesellschaft [3] angeboten werden, jedoch würde ein, vollständig oder teilweise in dedizierter Hardware realisierter MP3-Dekoder neues Potential hinsichtlich Energieeinsparung und Absenkung der Taktfrequenz bieten. In [2] wird zum Beispiel das Potential von Hybridstrukturen für solch eine Realisierung untersucht, bei welchen das Bitstream-Parsing von einem einfachen Prozessor durchgeführt wird und die aufwändigen MP3-spezifischen Rechenoperationen von dedizierten Hardwareblöcken realisiert werden.

Am Ende dieser Veröffentlichung [2] findet sich die Anmerkung, dass sogar ein vollständig in Hardware realisierter MP3-Dekoder denkbar wäre. An diesem Punkt setzt diese Arbeit an, mit dem Ziel solch einen Dekoderkern zu entwickeln.

2. Der MP3-Audiostandard

Genormt ist MP3 im so genannten MPEG-1 Standard. MP3 ist also keine Abkürzung für MPEG-3 wie man auf den ersten Blick vermuten könnte, sondern steht für MPEG-1 Audio Layer III. Also eine „Audioschicht“ im Videostandard. Mit „Schichten“ sind hierbei Kompressionsalgorithmen unterschiedlicher Komplexität gemeint, wobei es sich bei MP3 um die Ebene höchster Komplexität handelt. Es gibt also noch Layer I und Layer II, die Algorithmen geringerer Komplexität darstellen, jedoch auch nur geringere Kompressionsraten ermöglichen (bei gleich bleibender Audioqualität).

2.1. Die MP3-„Idee“

Der Titel diese Abschnitts ist genau genommen etwas irreführend, da es „die“ MP3-Idee eigentlich nicht gibt, denn es kommen zum Einen Verfahren zum Einsatz die auch in Layer I und Layer II verwendet werden, jedoch deutlich erweitert und verfeinert wurden und zum Anderen sind neue Methoden, wie z.B. eine Entropiekodierung oder dynamische Frames hinzugekommen. Diese Verfahren tragen allesamt zur Datenreduktion bei, so dass mit MP3, ein, verglichen mit Layer I und II, sehr viel komplexeres Format existiert. Machen wir dennoch den Versuch eine Grundidee zu beschreiben und suchen nach dem Teilverfahren, welches den größten Beitrag zur Datenreduktion liefert, so führt dies zum so genannten

Maskierungseffekt, eine Methode bzw. ein Effekt welcher in Layer I und II ebenso Anwendung findet bzw. ausgenutzt wird. Diese Maskierung ist neben weiteren Effekten im so genannten *psychoakustischen Modell* hinterlegt. Das psychoakustische Modell bildet somit den Teil des Codecs, der die Eigenschaften der menschlichen Wahrnehmung von Tonsignalen berücksichtigt und die Audiodaten entsprechend kodiert, so dass zwar eine Datenreduktion statt finden kann, diese aber im Toleranzbereich der psychoakustischen Wahrnehmung untergeht und somit nicht hörbar ist. Die „Qualität" des psychoakustischen Modells eines Enkoders ist somit ein maßgeblicher Faktor hinsichtlich der erreichbaren Datenrate und Audioqualität.

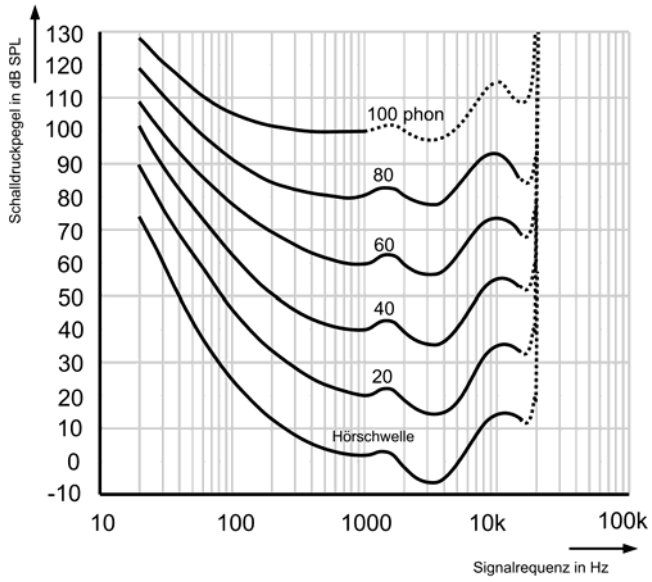


Abbildung 1: menschliche Wahrnehmung von Audiosignalen

Die unterste der in Abbildung 1 dargestellten Kurven zeigt nun die menschliche Hörschwelle, also den mindestens notwendigen Schalldruckpegel, damit ein Tonsignal überhaupt vom Menschen wahrgenommen werden kann. Digitale unkomprimierte Audiodaten die zum Beispiel als PCM-Samples vorliegen repräsentieren ausschließlich den Schalldruckpegel. Für eine PCM-kodierung ist die spektrale Zusammensetzung und die damit verbundene frequenzabhängie Wahrnehmung von Audiosignalen durch das menschliche Gehör daher völlig unerheblich.

Der in Abbildung 1 dargestellte Verlauf der Hörschwelle gilt so in dieser Form eigentlich nur bei absoluter Stille. Das heißt, wenn vom menschlichen Hörer gerade gar kein Ton wahrgenommen wird, gilt für ein neu in Erscheinung tretendes Tonsignal obiger Verlauf. Ist jedoch bereits ein anderes (wahrnehmbares) Signal präsent, so ändert sich der Verlauf der Hörschwellenkurve merklich. Dieser Effekt wird *Maskierung* genannt, hin und wieder findet man auch die Bezeichnung *Verdeckung*. Der Begriff *Maskierung* rührt daher, dass Töne durch andere Töne anderer Frequenz verschluckt, eben maskiert werden können.

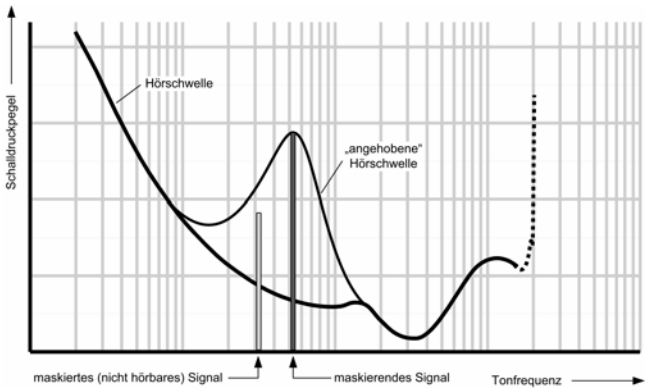


Abbildung 2: Maskierung

Abbildung 2 stellt diesen Sachverhalt schematisch dar. Dabei sind zwei Audiosignale unterschiedlicher Frequenz präsent. Die ursprüngliche Hörschwellenkurve wird dabei durch das lautere Signal derart angehoben, dass das leisere Signal „untergeht" und vom menschlichen Hörer nicht mehr wahrgenommen werden kann.

Bei jeder Form von digitaler Speicherung von ursprünglich analogen Signalen, muss der Signalverlauf auf eine Folge diskreter Signalwerte abgebildet werden, das heißt, quantisiert werden. Je höher nun die Anzahl der möglichen Quantisierungsstufen ist, desto näher liegt das digitale Signal am Original und desto geringer fällt der durch die Quantisierung entstandene Fehler aus. Allerdings geht logischerweise mit einer Erhöhung der Quantisierungsstufen auch eine höhere Datenrate einher, so dass hier ein Kompromiss gewählt werden muss.

Digitales PCM-Audio in CD-Qualität ist mit 16 Bit pro Sample quantisiert. Das heißt es existieren 216 = 65536 Quantisierungsstufen. Den bei der Quantisierung eines analogen Signals entstehenden Fehler bezeichnet man auch als Quantisierungsgeräusch. Quantisierungsgeräusch deshalb, weil der Fehler, also die Differenz zwischen der nächstliegenden Quantisierungsstufe und dem eigentlichem Signalpegel als zusätzliches Rauschsignal auftritt. Reduziert man die Pegelauflösung eines digitalen Audiosignals, so nimmt man diese Quantisierungsgeräusch im

Regelfall als störendes Rauschen war. Dies muss jedoch nicht zwingend der Fall sein. Inwieweit sich das Quantisierungsgeräusch störend bemerkbar macht, hängt ganz von der Beschaffenheit des Audiosignals ab und natürlich auch davon, wie stark die Auflösung nun tatsächlich reduziert wird. Der oben beschriebene Maskierungseffekt maskiert Geräusche die unter der Maskierungsschwelle liegen und somit unter Umständen auch das Quantisierungsgeräusch.

Letztendlich bedeutet dies, dass die Signalauflösung so lange reduziert werden kann, wie das Quantisierungsgeräusch noch unter der Maskierungsschwelle liegt. Wie man in Abbildung 2 deutlich erkennen kann ist der Verlauf der Maskierungsschwelle jedoch stark frequenzabhängig, so dass es nicht sinnvoll wäre die minimale Quantisierung einfach für einen zeitlichen Abschnitt des Audiosignals zu ermitteln. Stattdessen muss die Analyse in unterschiedlichen Frequenzbereichen getrennt durchgeführt werden. Bei allen MPEG-Layern findet daher stets eine Filterung des Audiosignals in mehrere Teilbänder statt.

2.2. Die MP3-Dekodierung

Abbildung 3 zeigt nun den grundlegenden (vereinfachten) Ablauf der MP3-Dekodierung und die nacheinander durchzuführenden Teilschritte der Dekodierung. Im Folgenden findet sich ein kurzer Überblick über die Funktion dieser Teilschritte.

Bitstream-Parsing

Neben der Datenreduktion durch das gezielte Entfernen von nicht hörbarer Audioinformation mittels psychoakustischen Modellen wird bei MP3 ein weiterer Weg gegangen: Ein sehr komplexes Bit Packing um den Datenstrom optimal mit Daten zu füllen und trotzdem ein streamingfähiges Format zu erzeugen. Alle MPEG-1 Audioformate, also Layer I bis III besitzen die Eigenschaft, dass die Wiedergabe an beliebiger Stelle gestartet werden kann. Bei Layer I und II gestaltet sich dies besonders einfach. Hier hängt die Länge eines Frames, also eines für sich selbst dekodierbaren Teil des Bitstromes lediglich von der Abtastrate und der Datenrate ab. Ändern sich diese Werte nicht von Frame zu Frame, so bleibt auch die Framelänge konstant. Abtastrate und Bitrate sind im so genannten *Header* abgespeichert der sich stets am Frameanfang direkt nach einer Bitsequenz die den Beginn des Frames markiert, dem so genannten *Syncword* befindet. *Syncword* und *Header* haben bei allen MPEG-1 Layern den gleichen Aufbau und treten bei allen Layern stets äquidistant im Bitstrom auf (vorausgesetzt Bitrate und Abtastfrequenz ändern sich nicht). Während bei Layer I und II jedoch nun direkt die kodierten Audiodaten folgen und ausgelesen werden können um somit dem eigentlichen Dekoder zugeführt zu werden, so muss bei MP3 ein deutlich aufwändigeres Bitstream-Parsing betrieben werden.

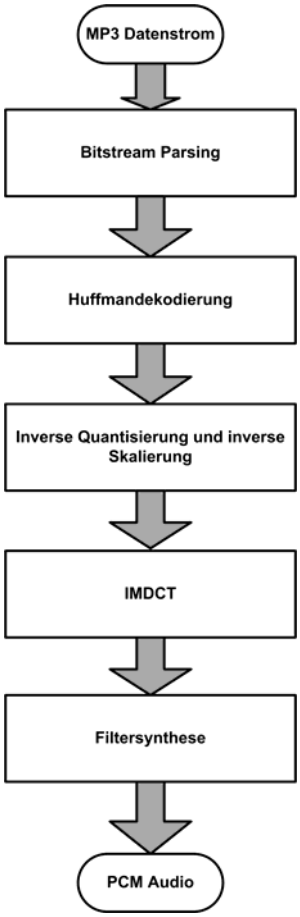


Abbildung 3: MP3-Dekodierung, grundlegender Ablauf

Wie bereits erwähnt treten auch bei MP3 *Syncword* und *Header* in äquidistantem und vorausberechenbarem Abstand auf (unter den ebenfalls genannten Voraussetzungen). Direkt an diesen Block, also ebenfalls in äquidistant, schließt die *Side Information* an, ein Block der zusätzliche Information für die framespezifische Einstellung des Dekoders enthält. Neben dieser Information, die sozusagen die Konfiguration für den Dekoder mitliefert, trifft die *Side Information* auch eine Aussage (in Form eines Zeigers) darüber, wo im Bitstrom sich nun die zum Frame gehörenden kodierten Audiodaten befinden. Diese schließen nämlich nur selten direkt an die Blöcke *Syncword, Header* und *Side Information* an. Abbildung 4 zeigt nun den grundsätzliche Aufbau des MP3-Datenstroms, sowie die Verknüpfung zwischen

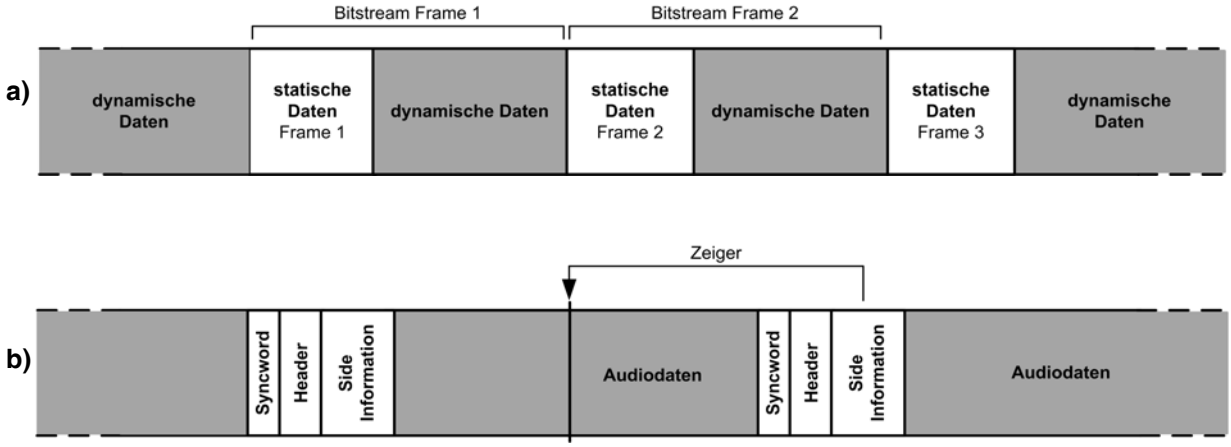


Abbildung 4: a) MP3-Bitstromaufbau b) Verknüpfung der statischen und dynamischen Daten

der Side Information und den dynamisch zugeordneten Audiodaten. Man erkennt hier die Unterteilung des Bitstroms in statische und dynamische Daten. Während sich die dynamischen Daten sofort einem bestimmten Frame zuordnen lassen, ist die Framezugehörigkeit der dynamischen Daten nicht sofort ersichtlich. Zwar bezeichnet man stets einen statischen Bitstromteil plus den darauffolgenden dynamischen Teil als sogenanntes *Bitstreamframe*, jedoch handelt es sich hierbei nicht um einen für sich dekodierbaren Teil des Audiostroms. Hiezu ist ein so genannter *Audio-Frame* notwendig, der aus einem statischen Bitstromteil besteht und einer Menge an dynamischen Daten, die aber nicht zwingend direkt an den betreffenden statischen Part anschließen müssen. Wie bereits erwähnt ist zum Auffinden dieser Daten die Auswertung eines Zeigers notwendig, der sich in jedem statischen Bitstromteil befindet.

Auslesen der Skalenfaktoren

Ist nun die Position im Bitstrom bekannt, an welcher die Audiodaten eines bestimmten Frames beginnen, so kann mit dem Auslesen der Skalenfaktoren begonnen werden. Im Gegensatz zu den anschließenden spektralen Audiodaten sind die Skalenfaktoren nicht huffmankodiert. Allerdings kann die Anzahl und Anordnung der Skalenfaktoren im Bitstrom von Frame zu Frame stark variieren. Dem Dekoder wird diese Struktur zuvor über die Side Information mitgeteilt. Abhängig von der Blockstruktur des Frames werden auch unterschiedliche Skalenfaktoren mit übertragen. Auch über die Bitbreite der Skalenfaktoren, die ebenfalls variiert werden kann, findet man Informationen in der *Side Information*.

Huffmandekodierung

Anschließend kann das huffmankodierte Spektrum des Frames ausgelesen werden. Eine Startposition für den Beginn dieser Daten ist dabei nicht hinterlegt, sie beginnen einfach nach dem letzten Skalenfaktor. Mittels der im Standard [1] festgelegten Huffman-Tabellen, welche dem Dekoder alle bekannt sein müssen, kann nun die Dekodierung statt finden. Welche Tabellen tatsächlich verwendet werden, wurde dem Dekoder bereits in der *Side Information* mitgeteilt und ist daher bekannt.

Inverse Quantisierung und inverse Skalierung

Bei der MP3-Kodierung wird eine Quantisierung vorgenommen, bei welcher die Spektralwerte, welche als Fließkommawerte von der modifizierten Kosinustransformation geliefert werden auf eine eingeschränkte Anzahl möglicher Spektralwerte abgebildet werden um diese dann letztendlich im Bitstrom zu speichern. Beim Dekodierungsvorgang ist nun der umgekehrte Prozess notwendig, d.h. aus den quantisierten Spektralwerten muss nun wieder der eigentliche Signalpegel errechnet werden. Während bei Layer-I und Layer-II die Spektralwerte linear quantisiert werden, also stets in äquidistantem Abstand ein gültiger Signalpegelwert zur Verfügung steht gestaltet sich der Vorgang bei Layer-III etwas aufwändiger. Hier war man bestrebt mit der Quantisierung ein Stück weit eine Anpassung an das menschliche Gehör und dessen Wahrnemung verschiedener Lautstärkepegel zu erreichen. Hierzu wird bei geringen Pegeln eine deutlich höhere Auflösung benötigt als bei sehr lauten Signalen. Diesem Umstand wird mit einer nichtlinearen Quantisierung über eine Potenzfunktion Rechnung getragen.

Wurden die vom Huffman-Dekoder gewonnenen unskalierten und quantisierten Spektralwerte mit der inversen Kennlinie des nichtlinearen Quantisierers entzerrt, so können diese reskaliert werden. Hierzu dienen die ebenfalls im Bitstrom enthaltenen Skalenfaktoren, die jeweils für eine bestimmte Anzahl an Spektralwerten, sogenannte *Skalenfaktorbänder* zuständig sind.

Inverse Modifizierte Diskrete Kosinustransformation

Die reskalierten Spektralwerte können nun mit Hilfe der inversen modifizierten diskreten Kosinustransformation zurück in den Zeitbereich transformiert werden. Dabei entsteht noch kein vollständiges Zeitbereichssignal, sondern es werden die Audiosignale der 32 Subbänder erzeugt. Dabei kann zur Rücktransformation dieselbe Berechnung herangezogen werden, die auch schon bei der Transformation in den Frequenzbereich verwendet wurde.

Filterbanksynthese

Schließlich können die 32 Signale der Subbänder über die Filterbanksynthese wieder zu einem einzigen Audiosignal zusammengefasst werden. Der Standard [1] definiert hierzu eine genaue Schrittkette welche als zentrales Element eine Matrixmultiplikation enthält.

Abschnitte bilden den statischen Teil des MP3-Frames, der folglich auch nur einmal pro Frame gelesen werden muss.

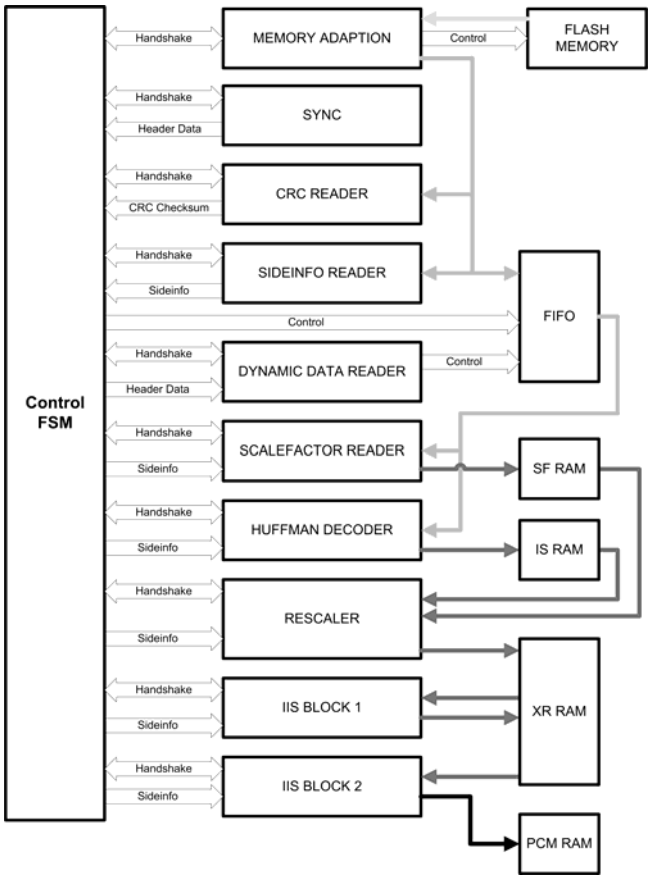


Abbildung 4: Architektur des VHDL-Decoders

3. VHDL Implementierung

3.1. Überblick

Die grundlegende Architektur der VHDL-Implementierung ist in Abbildung 4 dargestellt. Man kann sagen, dass die einzelnen Module im Wesentlichen chronologisch und zwar von oben nach unten aufgerufen werden. Im Folgenden werden die einzelnen Module kurz beschrieben.

3.2. Ablaufsteuerung

Die im vorherigen Abschnitt getroffene Aussage, die Module würden chronologisch nacheinander durchlaufen ist genau genommen nicht ganz korrekt, da einige Module zwei Mal pro Frame aufgerufen werden und einige unter Umständen sogar vier Mal. Betrachten wir einen vollständigen MP3-Frame, so enthält dieser einen einzigen Header und einen einzigen Side Information Abschnitt. Diese beiden

Anders verhält es sich hingegen mit dem dynamischen Teil des Frames. Darin finden sich entweder vier Mal (bei Stereo) oder zwei Mal (bei Mono) Blöcke mit den Skalenfaktoren und den huffmankodierten Spektralwerten. Dass diese Blöcke bis zu vier Mal vorhanden sind resultiert daraus, dass jeder MP3-Frame zwei so genannter *Granules* (weitere Unterteilung eines Frames) enthält und jedes *Granule* bis zu zwei Kanäle. Während der Reskalierung werden Skalenfaktoren und quantisierte Spektralwerte zusammengefasst. Ab diesem Zeitpunkt werden stets beide Kanäle zusammen verarbeitet. Die restlichen Module werden daher nur einmal pro *Granule* aufgerufen. Diesen Sachverhalt veranschaulicht Abbildung 5. Die Grafik zeigt zwei verschachtelte Schleifen, wobei die innere Schleife vier Mal (bei Stereo), und die äußere Schleife zwei Mal pro Frame aufgerufen wird.

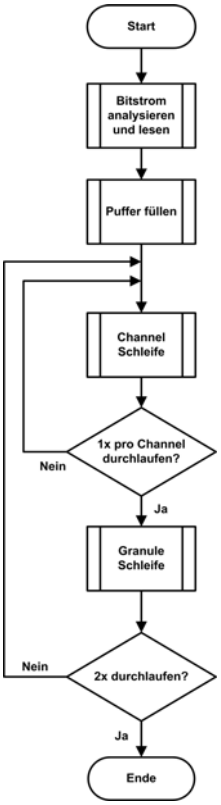


Abbildung 5: Ablaufsteuerung

3.3. Memory Adaption Layer

Der Memory Adaption Layer bildet die Anbindung an einen irgendwie gearteten Datenspeicher, der den MP3-Bitstrom enthält. Im Rahmen dieser Arbeit wurde eine Anbindung eines Flash-Speichers vom Typ *Intel StrataFlash* realisiert, wie er zum Beispiel auf einem Xilinx Evaluation Board *S3EStarter* zum Einsatz kommt. Selbstverständlich können andere Speichertypen verwendet werden, hierzu muss lediglich das Modul *Memory Adaption* angepasst werden und gewährleistet werden, dass dem übergeordneten Modul die gleiche Funktionalität sowie das gleiche Interface bereit gestellt wird.

3.4. Synchronisation

Aufgabe dieses Moduls ist es einen gültigen MPEG-Header zu finden. Hierbei muss zunächst, das im Standard [1] definierte *Syncword* gesucht werden. Da dieses *Syncword* jedoch auch als zufällige Bitfolge im kodierten MP3-Datenstrom auftreten kann müssen noch weitere Daten in, dem *Syncword* folgenden, *Header* überprüft werden. Da auch mit dieser Maßnahme, die Wahrscheinlichkeit einer Fehlsynchronisation, also das fälschlicherweise Erkennen eines MPEG-Headers im Datenstrom noch zu hoch ist, muss zusätzlich noch, mit den im *Header* enthaltenen Daten, die Position des nächsten Frames ermittelt werden und auch dessen Gültigkeit überprüft werden.

Der MP3-Decoder ist so konstruiert, dass eine Resynchronisation nach jedem gelesenen Frame statt findet, egal ob das Frame erfolgreich dekodiert wurde, oder ob ein Fehler aufgetreten ist und ohnehin eine neue Synchronisation notwendig wäre. Dies bietet den Vorteil, dass keine zusätzliche Logik für die Behandlung des Resynchronisationsfalls bereit gestellt werden muss. Selbstverständlich erkauft man sich diesen Vorteil mit zusätzlichen Taktzyklen die bei einer Sonderbehandlung der Resynchronisation wegfallen würden. Diese zusätzlichen Taktzyklen fallen jedoch relativ zum restlichen Timing des Dekodierungsvorgangs äußerst gering aus.

Das Synchronisationsmodul wird von einer FSM gesteuert. Zusätzlich besitzt das Modul einen weiteren Zustandsspeicher den so genannten *Synclevel*. Insgesamt müssen bei der Synchronisation in zwei aufeinander folgenden Frames jeweils vier Byte überprüft werden, die in der richtigen Reihenfolge auftreten müssen. Der *Synclevel* markiert die Position in diesem Synchronisationspfad. Jeder *Synclevel* ist an eine oder mehrere Bedingungen geknüpft, die das aktuell gelesene Byte erfüllen muss. Nur wenn diese Prüfung positiv verläuft wird der *Synclevel* um Eins erhöht und das nächste Byte überprüft. Andernfalls wird er auf Null zurück gesetzt.

3.5. FIFO-Buffer

Ein (erweiterter) FIFO-Buffer bildet nun das Bindeglied zwischen Bitstream- und Audio-Frame, also zwischen statischen und dynamischen Daten. Während die statischen Blöcke bestehend aus *Header* und *Side Information* im Bitstrom bei gleich bleibender Abtastfrequenz und Bitrate immer im gleichen Abstand auftreten, lassen sich die dazwischen liegenden dynamischen Daten nicht sofort einem bestimmten Frame zuordnen. Um die Framegrenzen in den dynamischen Daten zu ermitteln muss zunächst das Datenfeld *Main Data Begin* aus der *Side Information* ausgewertet werden. Der Wert dieses Datenfeldes ist als Zeiger aufzufassen, der die Anzahl an Bytes zwischen Framegrenze und *Syncword* liefert. Da jedoch die statischen Bitstromabschnitte hierbei nicht mitgerechnet sind und ein einziges Frame durchaus den Zwischenraum (dynamischen Teil) zwischen mehreren statischen Blöcken in Anspruch nehmen kann, ist es notwendig die

dynamischen Daten zunächst in einen FIFO-Buffer zu transportieren. Diese Aufgabe übernimmt das Modul *Dynamic Data Reader* (siehe nächster Abschnitt. Die Bezeichnung des Puffers als „erweiterter Buffer“ und nicht nur als „Buffer“ hat den Hintergrund, dass der dynamische Teil des Bitstroms nicht nur eine Dynamik hinsichtlich der Framezugehörigkeit, sondern auch eine Dynamik hinsichtlich der dort verwendeten Wortbreiten besitzt. Die im dynamischen Teil beherbergten Daten, *Skalenfaktoren* und huffmankodierte Spektralwerte, weisen Wortbreiten von 1 bis 13 Bit auf. Dies ist auch eine der Stärken der MP3-Kodierung, ein sehr effizientes Bitpacking, bei dem man stets bestrebt ist, möglichst geringe Wortbreiten einzusetzen. Für den Auslesevorgang des FIFO-Buffers bedeutet dies jedoch, dass er in der Lage sein muss unterschiedliche Wortbreiten zu liefern. Während beim Ladevorgang stets mit bytegroßen Wörtern gearbeitet werden kann, muss der Lesevorgang hinsichtlich der Wortbreite dynamisch sein.

3.6. Auslesen der dynamischen Daten

Nachdem im vorherigen Modul ausreichend dynamische Daten in den FIFO-Buffer geladen wurden und sichergestellt wurde, dass der Lesezeiger sich auf der Anfangsposition der dynamischen Daten des aktuellen Frames befindet, kann mit der Verarbeitung dieser dynamischen Daten begonnen werden. Das entsprechende Modul *Dynamic Data Reader* muss daher die Daten entsprechend ihrer Anordnung im Bitstrom (siehe [1]) auslesen und sortiert in einem Arbeitsspeicher ablegen.

3.7. Huffman-Dekodierung

Nach dem Auslesen der Skalenfaktoren werden nun noch die Spektralwerte benötigt um den eigentlichen Dekodierungsvorgang durchführen zu können. Da diese Spektralwerte jedoch entropiekodiert im Bitstrom untergebracht sind ist ein Modul erforderlich, welches die entsprechende Huffman-Dekodierung durchführt. Für solch eine Huffmandekodierung existieren unterschiedliche Verfahren [4] [5] die sich hauptsächlich hinsichtlich des Speicherbedarfs und der zur Dekodierung notwendigen Taktzyklen unterscheiden. Für die hier beschriebene Dekoderimplementierung wurde das Verfahren mittels bitweisem Lesen des Datenstroms und Dekodierung anhand von Baumstrukturen verwendet. Gründe hierfür sind zum Einen der gering ausfallende Speicherverbrauch, zum Anderen aber auch die Tatsache, dass bei der bitweisen Leseoperation stets nur so viele Daten ausgelesen werden, wie zur Dekodierung des aktuellen Huffmancodeworts nötig sind und somit kein rücksprungfähiger Puffer benötigt wird.

3.8. Reskalierung und Requantisieung

Das Modul *Rescaler* übernimmt die Berechnung der im Abschnitt 2.4.3.4.7 des Standards [1] definierten Potenzfunktion zur Requantisieung der vom Huffman-Decoder gewonnenen Spektraldaten sowie die Berechnung der im Abschnitt 2.4.3.4.7.1 des Standards [1] definierten Exponentialfunktion zur Reskalierung mit Hilfe der zuvor ausgelesenen Skalenfaktoren und den requantisierten Spektralwerten. Es sei angemerkt, dass daher die Bezeichnung „*Rescaler*“ nicht absolut zutreffend gewählt ist, da neben der Reskalierung auch die Requantisieung in diesem Modul bewerkstelligt wird, dieser Umstand jedoch vernachlässigt wird um eine einfache Modulbezeichnung zu erhalten.

Die o.g. Berechnungen müssen auf jeden der 576 Spektralwerte im RAM angewendet werden. Neben der eigentlichen Berechnung ist es also auch die Aufgabe des Moduls ein RAM-Interface zur Verfügung zu stellen, sowohl zum Arbeitsspeicher welcher die Spektraldaten enthält (*IS_RAM*), als auch zum Arbeitsspeicher der die Skalenfaktoren enthält (*SF_RAM*) sowie ein weiteres Interface, welches ausgangsseitig einen Arbeitsspeicher beschreibt, welcher die reskalierten und requantisierten Spektraldaten erhält (*XR_RAM*). Die Spektralwerte werden in Fixkommadarstellung mit einer Breite von 32 Bit im Q4.28 Format im ausgangsseitigen Arbeitsspeicher abgelegt. Auch intern wird stets mit Fixkomma-Darstellung gerechnet, jedoch an verschiedenen Stellen mit deutlich breiteren Wörtern um so die Genauigkeit zu erhöhen.

3.9. Synthese-Filterbank

Nachdem die Spektralwerte requantisiert und reskaliert wurden, muss die Stereo Verarbeitung durchgeführt werden (im Falle von Stereomaterial und Verknüpfung der Kanäle mit Joint Stereo). Anschließend wird die Aliassabschwächung durchgeführt und die *Inverse Modifizierte Diskrete Kosinus Transformation (IMDCT).* Diese drei Teilschritte des Dekodierungsvorgangs werden von dem Modul *IIS Block 1* erledigt, welches freundlicherweise vom Fraunhofer Institut für Integrierte Schaltungen in Erlangen (IIS) zur Verfügung gestellt wurde. Das IIS ist im Besitz eines Fixed-Point-MP3-Referenzedekoders, welcher zur Erstellung dieses Moduls herangezogen wurde. Dieser Dekoder ist in der Programmiersprache C beschrieben. Mit Hilfe des Tools *Catapult C* des Herstellers Mentor Graphics wurde

aus dieser Hochsprachenbeschreibung mittels High-Level-Synthese eine Beschreibung auf RTL-Ebene in VHDL erstellt.

Das zweite vom Fraunhofer IIS zur Verfügung gestellte Modul *IIS Block II* bewerkstelligt die Filterbanksynthese um aus den im vorherigen Modul erzeugten Samples der einzelnen Subbänder nun wieder das Audiosignal im PCM-Format zu generieren. Eine genaue Beschreibung des verwendeten Algorithmuses findet sich im Standard [1] in Abschnitt 2.4.3.4.10.

4. Fazit und Ausblick

In der vorliegenden Arbeit wurde gezeigt, dass eine VHDL-Implementierung eines MP3-Dekoders, welche gewisse Vorteile gegenüber einer reinen Softwareimplementierung bietet, möglich ist. Wie bereits in [2] prognostiziert ist für eine MP3-Dekodierung mittels dedizierter Hardware eine weitaus geringere Taktfrequenz notwendig, als dies bei Softwareimplementierungen auf Prozessoren der Fall ist. Diese, vorliegende Implementierung ist als eine Art „erster Entwurf“ zu sehen, da sie sicherlich noch sehr viel Optimierungspotenzial bietet. So werden in der vorliegenden Version sämtliche an der Dekodierung beteiligten Module nacheinander aufgerufen, eine parallele Verarbeitung der Daten in Verschiedenen Modulen ist noch nicht eingebracht. Diese chronologische Abfolge der Modulaktivitäten bietet sich jedoch geradezu für ein umfangreiches Piplining an. Mit solch einer Piplinestruktur würde sich die Taktfrequenz mit Sicherheit noch deutlich reduzieren lassen. Weiterhin könnten an verschiedenen Stellen Modulgrenzen aufgelöst werden und somit Speicherzugriffe eliminiert werden. So könnte die Requantisierung und Reskalierung direkt im Anschluss an die Huffman-Dekodierung erfolgen, so dass die skalierten und quantisierten Spektralwerte nicht zwischengespeichert werden müssen. Um ein weitgehend modulares Design zu erstellen, in welchem auch eine gute Testbarkeit der Einzelmodule gegeben ist, wurde auf solche Schritte in der vorliegenden Version bisher verzichtet. Weiterhin könnte die Huffmandekodierung optimiert werden. In wie weit hier ein schnelleres Verfahren zum Einsatz kommen kann hängt stark von der vorgesehenen Zielplattform ab und vor allem welche Speicherressourcen diese zur Verfügung stellt.

Literaturangaben

[1] ISO/IEC 11172-3, *Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s – Part 3: Audio,* 1993

[2] Mayer, F.; Dalquen, D.; Dettbarn, *T. Hardware Accelerating Audio Coding Algorithms*, Consumer Electronics, 2006

[3] Gayer, M.; Lohwasser, M.; Lutzky, M. *Implementing MPEG Advanced Audio Coding and Layer-3 encoders on 32-bit and 16-bit fixed-point processors*, Fraunhofer Institute for Integrated Circuits IIS, 2004

[4] Ruckert, M.*, Understanding MP3*, Wiesbaden, 2005, Vieweg & Sohn Verlag, ISBN 3528059052

[5] Pajarola, R., Fast *Prefix Code Processing*, Proceedings IEEE ITCC Conference, 2003

RFID- Frontend ISO 15693

Tobias Volk

Hochschule Offenburg, Badstr. 24, 77654 Offenburg

0781/205-179, 0781/205-174, tobias.volk@fh-offenburg.de

Im Rahmen eines Forschungsprojekts im Institut für Angewandte Forschung der Hochschule entstand ein RFID- Frontend- Baustein. Dieser unterstützt die physikalische Ebene des Standards ISO 15693 „Vicinity Card“.

RFID steht für „Radio Frequency Identification“. Es handelt sich dabei, um ein Verfahren zur kontaktlosen Identifikation über eine Luftschnittstelle. Ein aufwendig gestalte Lesegerät, „Reader“, sammelt Daten von einfachen Transponder- Systemen, meist ohne eigene Energieversorgung, den so genannten „Tags“. Die meisten Tags sind simple Datenspeicher, EEPROMs mit RFID- Frontend und Antenne versehen.

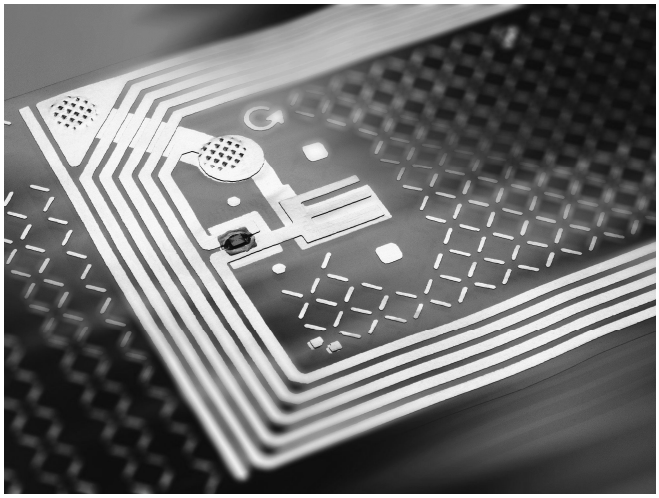


Abbildung 1 RFID- Transponder (Tag)

Ihre hauptsächliche Anwendung findet das Verfahren im Bereich der Logistik, aber auch in Schließanlagen und vielleicht in wenigen Jahren in unseren Geldscheinen.

Um die Kosten möglichst gering zuhalten, sind die Schaltung meist hoch integriert. Schnittstellen-bausteine, die es ermöglichen eigene Erweiterungen dieser Technologie zu entwickeln, sind auf dem Markt kaum vorhanden. Aus diesem Grund hat das Institut für Angewandte Forschung der Hochschule Offenburg einen Frontend- Baustein für den RFID- Standard ISO 15693 entwickelt.

1. Anwendungskonzept

Der Baustein ist als Teil eines Multichip- Designs angedacht. Sein Zweck besteht darin, ein Micro-controller- System um die RFID- Schnittstelle zu erweitern.

Hierzu ein Beispiel: Ein System zur Warensicherung soll entwickelt werden. Geplant ist, alle Stöße und ungewöhnlichen Beschleunigungen zu erfassen. Über die, in der Logistik üblichen, RFID- Schnittstelle sollen diese Daten, neben Kundendaten und der Waren-beschreibung zur Verfügung gestellt werden.

Abbildung 2 Warenlieferung

Aus Kostengründen fällt die Entscheidung auf eine diskrete Realisierung, bestehend aus Detektor-schaltung, Microcontroller, RFID-Frontend und einer möglichst kleinen Batterie zur Stromversorgung. Um den Energieverbrauch möglichst gering zu halten, befindet sich der Controller die meiste Zeit im Energiesparmodus. Im Fall einer Erschütterung wird dieser durch die Detektorschaltung geweckt und speichert das Ereignis im internen EEPROM. Am Ende der Lieferung sollen die Daten durch ein RFID-Lesegerät ausgelesen werden.

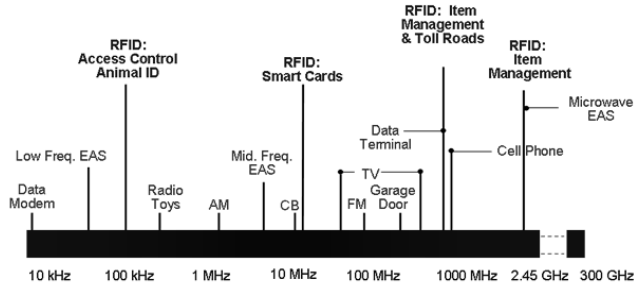
Abbildung 3 RFID- Handlesegerät

Der Mikrokontroller übernimmt dabei die höheren Protokollebenen, Schnittstellenzugriff, Kommando-Interpretation und CRC. Über die optionalen Kommandos kann hier der Standard erweitert werden.

Eine Taktung des Microcontrollers von wenigen Megahertz ist ausreichend, um das byteserielle Interface des Frontends zu bedienen. Solange alle Empfangsbytes innerhalb von 200µs erfasst werden, ist die Programmierung des Controllers von sämtlichen Timings unabhängig. Diese, sowie die physikalisch bedingten Teile des Standards, werden im Baustein gekapselt.

2. Die Luftschnittstelle

Die Funkfrequenzen der RFID- Standards orientieren sich an den ISM- Bändern, da diese ohne große Auflagen frei genutzt werden können. Es wird dabei in zwei Klassen unterteilt: in Tags, die innerhalb des Fernfeldes operieren, UHF-RFID, und Transponder, welche eine Kommunikation über das Nahfeld betreiben, den LF bzw. MF-Typen.



Frequenz	125 kHz	13,56 MHz	900 MHz	2,45 GHz
Wellenlänge	2.3 Km	22 m	33 cm	12 cm

Abbildung 4 RFID- Frequenzen

Der Standard ISO 15693 „Vicinity Cards“ gehört zur letzteren Gruppe. Er verwendet die Frequenz 13,56MHz, was einer Wellenlänge von 22m entspricht.

Die folgende Tabelle zeigt Parameter des Standards zur besseren Übersicht. Eine Aussage über die Reichweite lässt sich jedoch nicht machen, da diese von der Dimensionierung der Antennen abhängt.

Trägerfrequenz	13,56 MHz
Feldstärke	5-0,15 A/m
Datenrate	26,69 kBaud

Tabelle 1 Parameter ISO 15693

Zur Kommunikation im Nahfeld wird ein Magnetfeld verwendet. Es wird vom Reader erzeugt. Das System, Reader und Tag, kann durch einen lose gekoppelten Transformator modelliert werden.

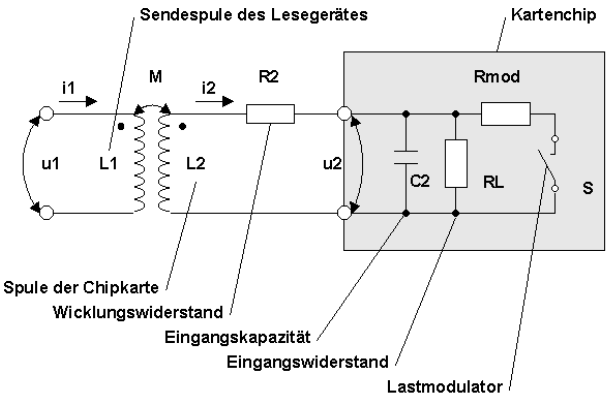


Abbildung 5 Ersatzschaltbild

Aus dem Vergleich mit einem Transformator kann auf die wesentlichen Eigenschaften der Schnittstelle geschlossen werden. Ein Übertrager ist in der Lage Energie mittels magnetischer Kopplung zu transportieren. Es ist somit möglich, ein passives Tag zu versorgen. Die Trägerfrequenz ist dabei in beiden Spulen des Transformators gleich, was einen Oszillator auf der Empfangsseite überflüssig macht. Durch die Verbindung können ferner Daten ausgetauscht werden.

3. Energie und Überspannung

Bei guter Kopplung kann über die RFID- Schnittstelle etliche Milliwatt an Leistung übertragen werden. Bei schlechter Kopplung reicht die Energie kaum aus, um das Rücksenden der Daten zu gewährleisten. Um eine hohe Reichweite zu erreichen, muss sich der Transponder an die Stärke des Feldes anpassen. Ebenfalls ist für die Programmierung des Microcontrollers die Information wichtig, ob überhaupt ein Feld mit entsprechender Stärke anliegt und es sich lohnt, die Energie für den Aufbau einer Kommunikation zu investieren.

Die Anpassung an die Feldstärke, sowie die Detektion des Feldes, sind durch drei Schaltungen realisiert. In

Abbildung 6 sind diese eingezeichnet, Gleichrichter, Schutzschaltung und Feld-Detektion.

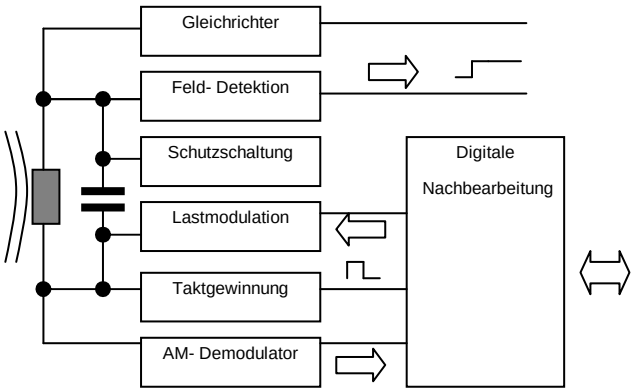


Abbildung 6 Blockdiagramm der Schaltung

Beim Anliegen eines Feldes wird eine Spannung in der Spule induziert. Ein Gleichrichter entnimmt dem Empfangsschwingkreis, welcher durch Antenne und externem Kondensator gebildet wird, Energie. Durch diesen wird die restliche Schaltung versorgt. Ebenfalls könnte ein Akkumulator mit der verbleibenden Energie gespeist werden. Parallel greift eine Schutz-, bzw. Regulator-, Schaltung in den Schwingkreis ein um den Anstieg der Spannung im Kreis zu begrenzen und somit eine Zerstörung der Schaltung durch Überspannung zu verhindern. Nachdem die korrekte Versorgung des Bausteins sichergestellt ist, wird durch einen Detektorschaltung das vorhanden Feld erkannt.

Der Gleichrichter ist in der Zweiwege- Variante ausgeführt.

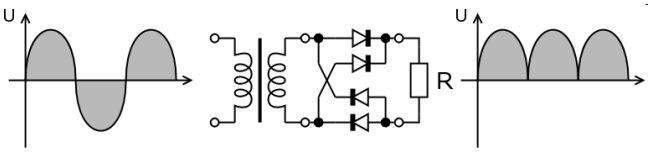


Abbildung 7 Zweiwegegleichrichter

Aus Gründen der Realisierung sind die bei diskreten Aufbauten verwendeten Dioden gegen MOSFET-Transistoren ersetzt worden. Der Unterschied ist lediglich, dass Anstelle der Durchlassspannung der Diode, nun die Schwellspannung des Transistors wirkt.

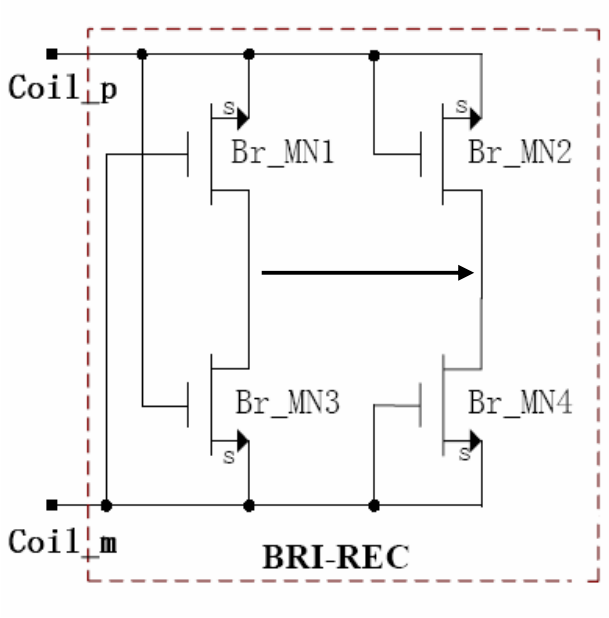


Abbildung 8 Zweiwegegleichrichter mit MOSFETs

Reguliert wird die Spannung durch die Schutzschaltung. Diese greift in die Güte des Schwingkreises ein. Sie kann also auch als Güteregelung bezeichnet werden. Die folgenden Gleichungen verdeutlichen das Prinzip der Regelung:

$U_L = Q \cdot U_i$ **Spannungsüberhöhung des Schwingkreises**

$U_L = \frac{I_B}{I_R} \cdot U_i$ **Überhöhung in Abhängigkeit vom Strom**

Die Spannung im Schwingkreis ist gleich dem Quotient aus Wirk- und Blindstrom multipliziert mit der induzierten Spannung. Letztlich kann jede Stromentnahme als Wirkstrom aufgefasst werden.

Den prinzipiellen Aufbau der Schaltung zeigt Abbildung 9.

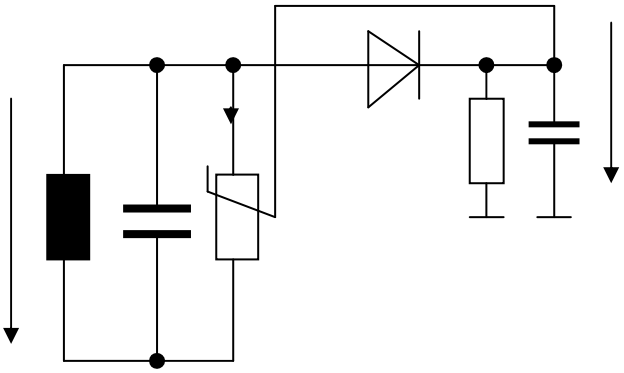


Abbildung 9 Prinzipbild Schutzschaltung

Der hintere Teil, die Diode und das RC- Glied, ermittelt den momentanen Amplitudenwert der Schwingkreisspannung. Über einen steuerbaren Widerstand wird dem Schwingkreis nicht benötigter Wirkstrom entnommen, d. h. durch eine Shunt-Regelung.

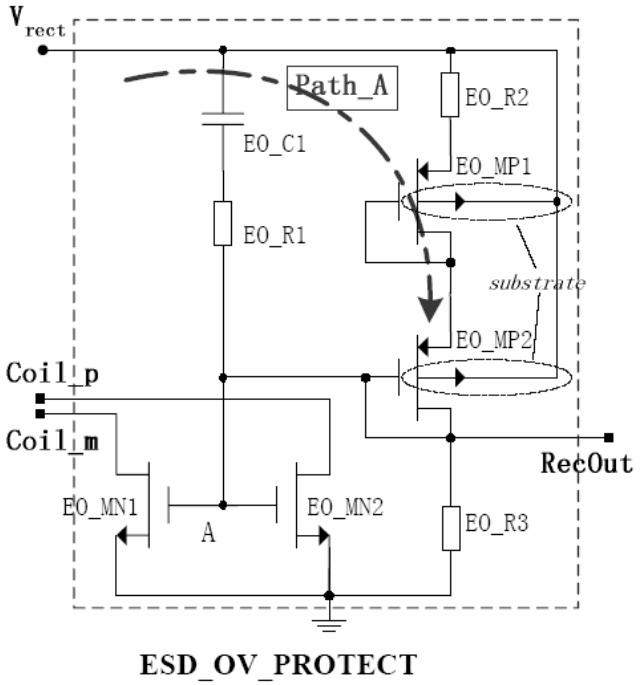


Abbildung 10 spannungsabhängiger Widerstand

Der Vorgang wird in Gang gesetzt, wenn einen gewisse Spannung an Vrec überschritten wird und der Steuerzweig leitet. Hierdurch werden auch die bei n-Kanaltransistoren, welche mit dem Schwingkreis verbunden sind, leitfähig und entnehmen dem Kreis Strom.

Um nun mit der Kommunikation zu beginnen, wird das Vorhandensein der Spannung als Ausgangssignal detektiert. Genutzt wird dabei, dass Gleichrichter und Schutzschaltung die Spannung zur Versorgung hin symmetrieren. Zur Detektion des Feldes genügt nun ein einfacher Schmitt- Trigger.

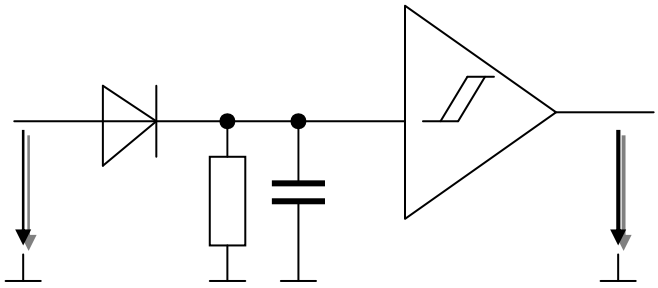


Abbildung 11 Detektorschaltung

4. Empfang

Der Reader kann Daten durch Veränderung der Sendeleistung übermitteln. Auf das Ersatzschaltbild, Abbildung 5, bezogen, durch Änderung der Spannung auf der Primärseite. Das Übertragungsverfahren entspricht einer Amplitudemodulation.

Die gesendeten Daten sind nach dem „Puls Position"-Verfahren kodiert. Dieses Verfahren hat eine große Anzahl von Nullenfolgen und beeinträchtigt somit den Energietransfer zum Transponder nur unmerklich.

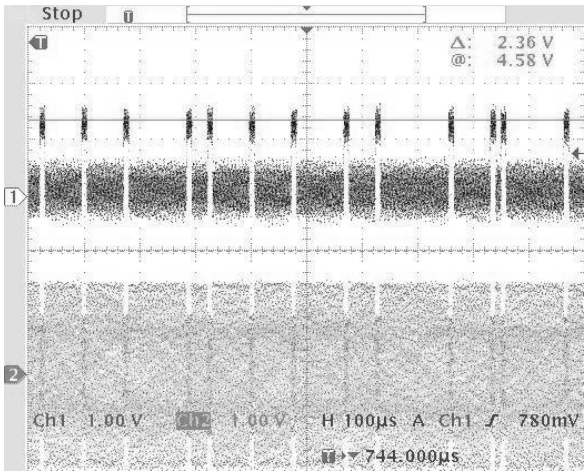


Abbildung 12 "Puls Position" moduliertes Signal

Die Variation der Sendeleistung führt letztlich zur Veränderung der induzierten Spannung in der Antenne. Um gesendeten Daten aus dem Empfangskreis zu gewinnen, wird eine HF- Diskriminator- Schaltung verwendet, ähnlich wie in der Rundfunktechnik. Die hieraus resultierende Gleichspannung mit Wechselanteil wird durch zwei Tiefpässe filtert. Beide Tiefpässe haben unterschiedliche Zeitkonstanten. Auf dem ersten Tiefpass wird der Gleichanteil ermittelt. Diese dient als Referenz für den Vergleich mit dem zweiten Tiefpass. Ein Komparator kann hieraus die Bitfolge gewinnen.

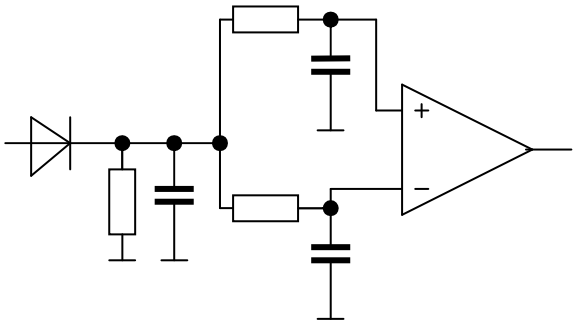


Abbildung 13 Demodulationsschaltung

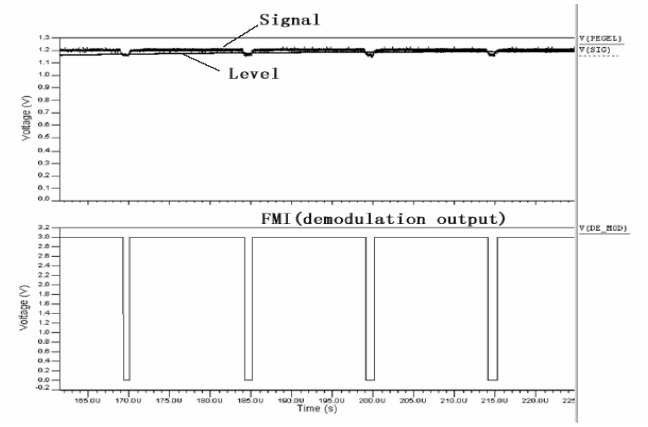


Abbildung 14 Signalverläufe der Demodulation

Das gewonnene Signal kann anschließend digital weiterverarbeitet werden. Letztlich soll der Bitstrom zu einem Bytestrom gewandelt und die entsprechenden Sonderzeichen als Steuersignal ausgewertet werden.

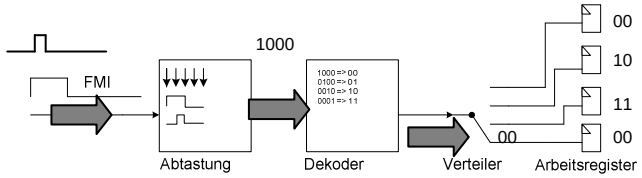


Abbildung 15 Dekoderlogik

Abbildung 16 verdeutlicht den Ablauf. Zunächst wird der Datenstrom einsynchronisiert. Die Arbeitsfrequenz ist der des Feldes entnommen. Die Bitfolge wird mit einer Kodetabelle verglichen.

1	0	0	1	0	0	0	0	SOF
0	1	0	0	0	0	0	0	00
0	0	0	1	0	0	0	0	01
0	0	0	0	0	1	0	0	10
0	0	0	0	0	0	0	1	11
1	0	0	0	X	X	x	x	EOF

Tabelle 2 Kodetablle

Die daraus resultierenden Bitpaare werden in einem Arbeitsregister gespeichert und als vollständiges Byte auf den Bus zwischen Baustein und Microcontroller gelegt. Die Sonderzeichen und die Ausgabe eines Datenbytes lösen Steuersignale aus, welche der Microcontroller als Interrupts auswerten kann.

5. Senden

Gesendet wird in unserem Standard mit Hilfe einer Lastmodulation. Das Prinzip wird deutlich bei Betrachtung des elektrischen Ersatzschaltbildes, Abbildung 5.

Das elektrische System, Reader und Tag, ist miteinander gekoppelt. Verändert sich die Stromaufnahme des Transponders, so wird dies vom Reader durch einen Anstieg seiner Stromabgabe detektiert, letztlich hat sich die Last geändert.

Das Tag erreicht diese Änderung durch Hinzuschalten eines zusätzlichen Modulationswiderstandes, Rmod. Die Modulation der Last entspricht einer Amplitudenmodulation und erzeugt dementsprechend Seitenbänder. Um die Erkennung des Signals für den Reader zu erleichtern wird die Veränderung des Modulationswiderstands zusätzlich mit einem so genannten Hilfsträger kombiniert. Hierdurch wird der Abstand der Seitenbänder zur Trägerfrequenz erhöht. Der Reader kann dadurch besser das Signal filtern.

Auf den Hilfsträger können nun die Daten moduliert werden, so als wäre dieser die Trägerfrequenz eines unabhängigen Kanals.

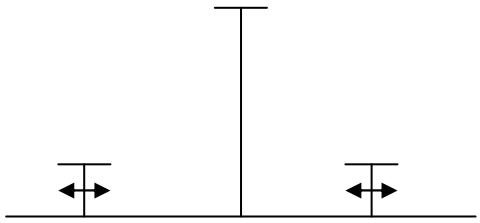


Abbildung 16 Hilfsträger

Im gegebenen Fall handelt es sich um eine digitale Frequenzmodulation, FSK, aber auch eine Amplitudenmodulation ist im Standard vorgesehen.

Zusätzliche Übertragungssicherheit wird durch Synchronisierung des Datenstroms erreicht. Aus dem Zeitpunkt des letzten EOF- Signals kann die Eröffnung eines gültigen Sendefensters ermittelt werden. Das Sendefenster ist so exakt toleriert, dass ein bitweiser Vergleich der Kommunikation möglich ist.

Das Öffnen eines Sendefensters wird durch einen simplen Mechanismus berechnet. Bei Detektion eines EOF- Signals durch die Empfangseinheit, wird ein Zähler synchron rückgesetzt. Dieser wiederholt den Zählvorgang solange, bis der Microcontroller das Steuersignal zum Beginnen der Kommunikation setzt. Nach Setzen des Flags wird beim nächsten gültigen Fenster automatisch mit dem Senden des Rahmens begonnen. Danach wird das erste Byte vom Datenbus abgeholt und dieses quittiert.

Das zu sendende Byte wird bitweise in die unten abgebildete Form umgewandelt, bevor es die Schaltung mittels Lastmodulation verlässt.

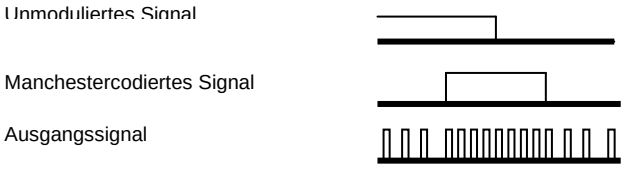


Abbildung 17 Signalaufbereitung

Zunächst wird der Bytestrom über einen synchronen Multiplexer, LSB-First, in einen Bitstrom gewandelt. Das Signal wird mit dem Ausgang eines Frequenzteilers kombiniert. Durch die Kombination der beiden Signale wird eine manchesterkodierte Bitfolge erzeugt. Dieses wiederum ändert den Teilerfaktor eines weiteren Frequenzzählers, wodurch eine Frequenzmodulation entsteht.

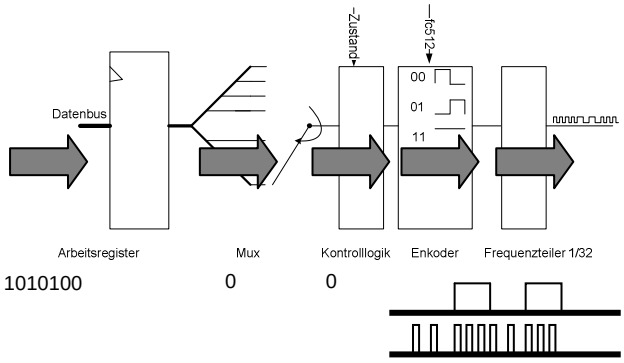


Abbildung 18 Sendeeinheit

Das Ausgangssignal kann nun auf die Last-modulationsschaltung des analogen Teils gegeben werden. Die zusätzliche Energieentnahme wird durch Schalten zwei MOSFET- Transistoren erreicht.

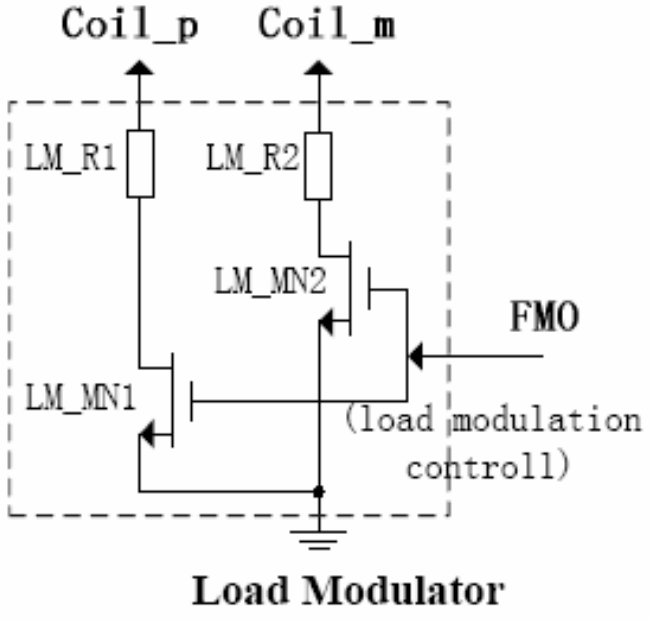


Abbildung 19 Lastmodulator

6. Stand der Entwicklung

Es wurde im Herbst ein Chip gefertigt mit folgenden Kenndaten:

Technologie:	0.35µ CMOS
Fläche:	3.3mm²
Davon analog:	0.65mm²
Gehäuse:	QFN5x5
Anzahl der Zellen:	268
Anforderung µC:	4MHz RISC

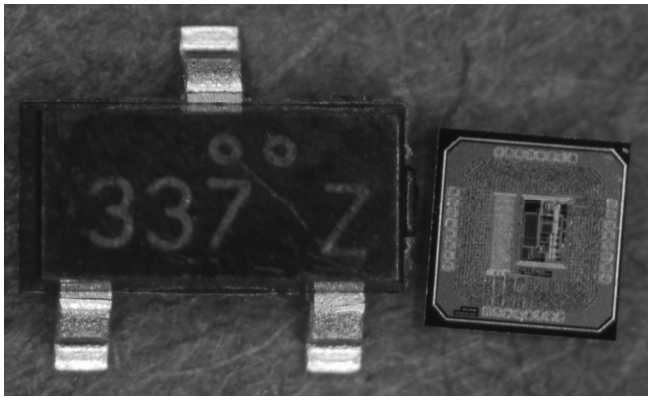


Abbildung 20 Frontendbaustein

Wegen eines Routing- Fehlers im Analogteil des Chips treten Kopplungen auf, die den Chip in der vorliegenden Form nicht einsetzbar machen. Ein Redesign im Sommer 2008 wird diese Probleme hoffentlich beheben.

7. Quellen

- RFID- Handbuch [Klaus Finkenzeller]
- ISO/IEC 15693 „Vicinty Cards“
- ISO/IEC FCD 10373-6 Identification cards – Test methods
- Integration of analog-digital front-end electronics for RFID (Radio Frequency identification) in CMOS 0.35 "u technology [Ji Li]

TTS - Text to Speech / Konkatenierende Synthese

Sven Kursawsky, Dipl.-Ing. Josef Hahn-Dambacher, Prof. Dr. Manfred Bartel

HTW Aalen, EDA Zentrum, Anton-Huber-Strasse 25, 73430 Aalen

Tel. 07361 / 576 – 4182, Fax 07361 / 576 – 444249, manfred.bartel@htw-aalen.de

Diese Arbeit beschäftigt sich mit der Entwicklung eines TTS[1]-Systems auf FPGA[2]-Basis. Mit Hilfe eines Sprachsynthese[3]-Systems wird Text in Sprache umgewandelt. Wie in Abbildung 1 dargestellt, wird ein eingegebener Text in ASCII-Code an das FPGA übertragen. Anschließend wandelt die Hardware den Text in ein Audiosignal, welches als Sprachsignal über den Lautsprecher ausgegeben wird.

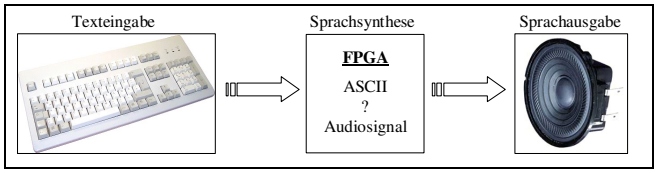


Abbildung 1: Vereinfachte Aufgabendarstellung

Zur Umsetzung der Aufgabe wird ein „Altium LiveDesign Evaluation Board“ mit einem „Altera Cyclone EP1C12“ FPGA eingesetzt. Die Durchführung des VHDL[4]-Entwurfs geschieht in der Entwicklungsumgebung „Mentor Graphics FPGA Advantage“.

1 Motivation

Heutzutage gewinnt die Textausgabe einen immer größeren Stellenwert. Beispielsweise wäre das Navigieren im Auto ohne Sprachausgabe heute nicht mehr denkbar. Ohne die Ansagen, wäre der Fahrer ständig abgelenkt, da er fortwährend auf das Navigationsgerät schauen müsste. Durch die Sprachausgabe kann der Fahrer sich weiter auf den Verkehr konzentrieren.

Des Weiteren ist es vorstellbar, dass die Sprachausgabe Menschen mit starken Sehstörungen im täglichen Leben hilft. So kann beispielsweise eine Waschmaschine oder ein Backofen die Programmeinstellung akustisch wiedergeben.

Heutige Lösungen sind meistens mit einem Prozessorsystem ausgestattet, bei denen die Realisierung der Sprachausgabe mit Software stattfindet.

Kostengünstiger und somit auch in Massenprodukten einsetzbar, ist eine solche Realisierung auf kleinstem Raum mittels einer Hardwareumsetzung möglich. Hierdurch ist auch der Einsatz im mobilen Bereich denkbar. Vorstellbar ist der Einsatz in nahezu allen elektronischen Geräten, beispielsweise in Mobiltelefonen oder elektronischen Übersetzern.

2 Grundlagen der Sprachsynthese

2.1 Allgemeines zur Sprachsynthese

Um Text in Sprache umzusetzen ist zuerst eine Analyse der Wortstruktur notwendig. Hierzu wird das zu analysierende Wort in Sprachsegmente zerlegt. Die kleinsten, Einheiten einer Sprache sind hierbei die Phoneme[5], (Elemente der Lautschrift[6]).

Für die Umsetzung von Buchstaben in Lautschrift, gibt es zwei Möglichkeiten, entweder

1. ein Umsetzungswörterbuch oder
2. ein Regelwerk.

Ferner ist eine Kombination der Verfahren möglich. Für eine Hardware ist ein Regelwerk wegen des kleinen Speicherbedarfs von Vorteil. Um jedoch eine optimale Umsetzung zu erhalten ist ein Umsetzungswörterbuch von Vorteil, hierbei ist jedoch der Speicherbedarf höher. Für eine richtige Umsetzung sind etwa 50.000 Wörter erforderlich.

Sind nun die Phoneme erzeugt, lässt sich die Sprache bilden. Hierzu gibt es unterschiedliche Verfahren und Methoden, welche im Kap. 2.2 Methoden der Sprachsynthese genauer behandelt werden. Um die Natürlichkeit der Sprache zu verbessern, sollte die so genannte Prosodie[7] beachtet werden und somit der Wortakzent, die Silbenbetonung und die Satzmelodie. Ein weiteres Problem stellen die Lautübergänge dar.

[1] **T**ext **t**o **S**peech
[2] **F**ield **P**rogrammable **G**ate **A**rray
[3] Maschinelle Erzeugung von gesprochener Sprache
[4] **V**HSIC **H**ardware **D**escription **L**anguage
[5] Kleinste Bedeutungsunterscheidende, aber nicht Bedeutungstragende Einheit einer Sprache
[6] Schriftsystem, das den Zweck hat, die Aussprache von Lauten oder Lautketten möglichst exakt wiederzugeben
[7] Wortakzent, Silbenbetonung oder Satzmelodie

Sie können durch spektrale Anpassung minimiert und somit Frequenzsprünge vermindert werden.

2.2 Methoden der Sprachsynthese

Als Stand der Technik bei der synthetischen Sprach-erzeugung sind folgende drei Verfahren zu nennen:

- Artikulatorische Synthese
- Formant[8] Synthese
- Konkatenierende[9] Synthese

2.2.1 Artikulatorische Synthese

Die Artikulatorische Synthese basiert auf der Modellierung des Verhaltens der Sprechorgane und dessen Auswirkung auf die Akustik. Bei dieser Art der Synthese werden als Einflussgrößen z.B. die Zungen-, Zäpfchenbewegung, die Bewegungen des Kehlkopfs, sowie der Luftstrom im zweidimensionalen Artikulationsmodell nachgebildet.

Die Sprechorgane zur menschlichen Sprachproduktion, die eine Rolle bei der Synthese spielen, sind in

Abbildung 2-1 dargestellt.

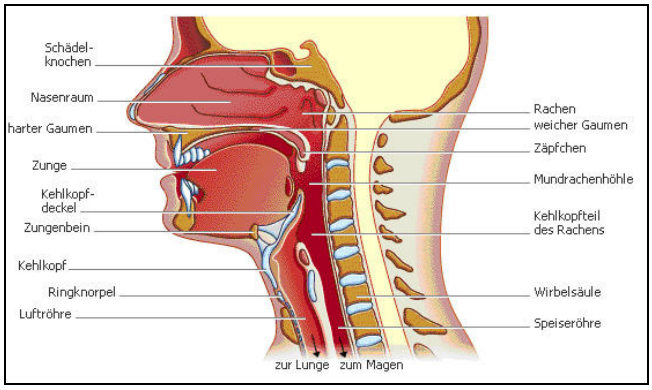


Abbildung 2-1: Artikulation des Menschen

[Quelle: http://de.encarta.msn.com/ (Querschnitt durch den menschlichen Sprechapparat)]

2.2.2 Formantsynthese

Bei dieser Art der Synthese wird mit Formanten[10], den Resonanzfrequenzen der menschlichen Stimme gearbeitet.

Es gibt Tabellen und Diagramme, in denen die Formantfrequenzen für die jeweiligen Laute näherungsweise aufgetragen sind. Für die Erzeugung der Laute sind theoretisch nur die ersten zwei Frequenzen notwendig. Alle weiteren Formanten charakterisieren lediglich die Anatomie des Sprechers, dessen Artikulations-Eigenarten und variieren je nach Sprecher.

Durch die Begrenzung der Anzahl der Formanten entsteht allerdings nur eine unnatürliche Stimme. Um eine nicht ganz unnatürliche Stimme zu erlangen, ist zudem eine aufwendige Regelformulierung erforderlich. Die Laute werden durch Überlagerung der Formanten erzeugt. Hierzu dient das Quelle-Filter-Modell, bzw. der Formantsynthesizer. Im Formantsynthesizer werden die Formanten des jeweiligen Lautes durch Angabe der drei Parameter

- Amplitude (A1 … A3),
- Frequenz (F1 … F3) und
- Bandbreite (BW1 … BW3),

wie in der Abbildung 2-2 dargestellt erzeugt.

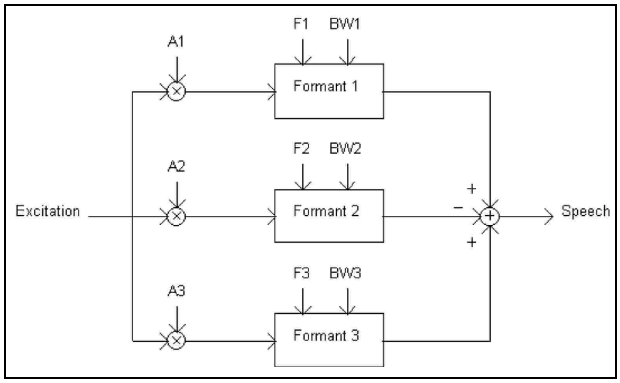


Abbildung 2-2: Quelle-Filter-Modell / Formantsynthesizer

2.2.3 Konkatenierende Synthese

Bei dieser Art der Synthese werden zuvor aufgenommene Sprachbausteine miteinander verkettet. In einer Sprachdatenbank können hierbei unterschiedlichste Sprachsegmente vorhanden sein. Vom Einzellaut (Phonem) bis hin zum ganzen Wort. Alle Arten von Segmenten, die im Kap. 2.3 Sprachsegmente aufgeführt sind, können vorhanden sein. Generell gilt je größer die Auswahl an Sprachsegmenten und je länger die Segmente, desto natürlicher die Sprache. Nachteil dieser Art der Synthese ist jedoch, dass der Sprecher und somit dessen Artikulations-Eigenarten fest vorgegeben sind.

[8] Resonanzspektren von Musikinstrumenten oder der menschlichen Stimme

[9] Verkettung (von Wörtern)

[10] Formanten sind resonanzartig verstärkte Frequenzbereiche von Partialtönen bei menschlicher Stimme und Musikinstrumenten.

2.2.3.1 Unit Selection

Der Stand der Technik im Bereich der Sprachsynthese ist das sogenannte Unit Selection. Dieses ist eine Erweiterung der Konkatenierenden Synthese. Zusätzlich werden ganze Sätze, Phrasen[11] und Texte im Inventar aufgenommen. Für die Synthese werden durch spezielle Suchalgorithmen und gewichtete Entscheidungsbäume eine Reihe von möglichst großen Segmenten bestimmt. Diese kommen der zu synthetisierenden Äußerung hinsichtlich dieser Eigenschaften möglichst nahe. Da diese Reihe ohne oder mit wenig Signalverarbeitung ausgegeben wird, bleibt die Natürlichkeit der gesprochenen Sprache erhalten, solange wenige Verkettungsstellen erforderlich sind. Im Gegensatz zur Diphon[12]-Synthese wird bei dem Unit-Selection das Zehnfache an Speicher benötigt, Aus diesem Grund ist diese Lösung eher auf leistungsfähigen Computern als Softwarelösung einsetzbar.

2.3 Sprachsegmente

Wie schon in den Kapiteln zuvor erwähnt, gibt es unterschiedliche Sprachsegmente. Hier eine Aufzählung der wichtigsten Sprachsegmente, die im Bereich der Sprachsynthese eine Rolle spielen:

- Phoneme (Allophone[13])
- Diphone
- Triphone[14]
- Halbsilben (Silben)
- Morphem[15]
- Wörter

2.3.1 Phonem / Allophon

Die nachfolgende Abbildung 2-3 verdeutlicht, warum Laute bei der Umwandlung von Text in Sprache so wichtig sind.

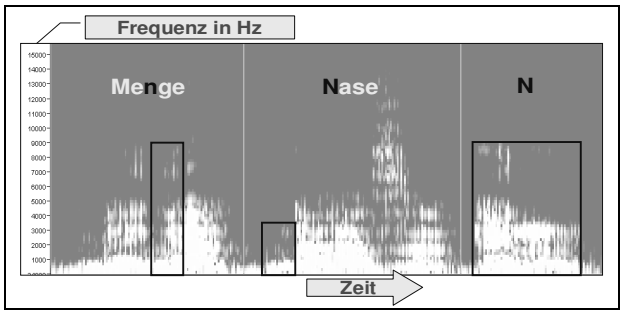


Abbildung 2-3: Unterschiede in der Aussprache von Buchstaben

Hier ist beispielsweise das Frequenzspektrum der Wörter: „Menge, Nase und N“ dargestellt. Deutlich ist zu erkennen wie sich die Dauer, die Frequenzen des Spektrums und die Intensitäten unterscheiden. Bei dem Buchstabieren, wie es hier beim Buchstaben „N“ der Fall ist, sprechen wir es eher als „EN“ aus, also setzt es sich somit aus mehreren Lauten zusammen. Hieraus resultiert die längere Laufzeit. Bei den Wörtern „Menge“ und „Nase“ werden unterschiedliche Laute eingesetzt, deshalb die Unterschiede im Frequenzbereich. Für die deutsche Sprache sind 45 Phoneme notwendig und für die englische Sprache 49. Es gibt dementsprechend mehr Laute als Buchstaben und somit lassen sich die Grapheme nicht einfach in Laute umwandeln.

Die Phoneme werden entweder im 16 Bit Unicode-Format, dem Internationalen Phonetischen Alphabet IPA[16], bzw. im SAMPA[17]-Format als sieben Bit ASCII Zeichen dargestellt. In Abbildung 2-4 ist ein Auszug der Phoneme im IPA-Format dargestellt.

Zeichen	Aussprache Englisch	Aussprache Deutsch
[b]	boat	Boot
[d]	do	du
[dʒ]	jungle	Dschungel
[ʒ]	treasure	*stimmhaftes 'sch' (wie in Dschungel, nur ohne 'd')*
[f]	friend	Freund
[g]	go	gehen
[h]	hotel	Hotel
[j]	young	jung
[k]	kiss	Kuss
[l]	left	links
[m]	mother	Mutter

Abbildung 2-4: Auszug IPA Zeichen

[Quelle: http://www.ego4u.de/de/dictionary/ipa]

[11] Phrase bezeichnet in der Linguistik gemeinhin Satzteile, die nur geschlossen im Satz verschoben werden können

[12] Sprachsegment, welches in der Mitte eines Phonems beginnt und in der Mitte des Nachfolgenden endet

[13] Phonemvarianten , da ein Phonem in unterschiedlichen lautlichen Varianten gesprochen werden kann

[14] Sprachsegment welches über drei Phoneme geht, von der Mitte des Ersten bis zur Mitte des dritten Phonems

[15] Die kleinste bedeutungstragende Einheit einer Sprache auf der Inhalts- und Formebene des Sprachsystems.

[16] **I**nternationales **P**honetisches **A**lphabet

[17] **S**peech **A**ssessment **M**ethods **P**honetic **A**lphabet

Die Phoneme sind allein nicht ausreichend um eine natürliche Aussprache zu erlangen, da diese wiederum auch je nach Kombination unterschiedlich ausgesprochen werden. Diese unterschiedlichen lautlichen Varianten der Phoneme werden als Allophone bezeichnet. Diese Phonemvarianten lassen sich jedoch nicht einfach durch Regeln beschreiben, zudem sind sie je nach Dialekt unterschiedlich.

Diese Thematik lässt sich durch längere Sprachsegmente weitestgehend umgehen. Bei den Allophonen variiert das Frequenzspektrum meistens am Übergang zwischen zwei Phonemen. Aus diesem Grund ist eine reine Konkatenierende Phonemsynthese nicht verständlich. Die Phoneme sind dennoch ein wichtiges Mittel für die Sprachsynthese.

2.3.2 Diphone / Triphone

Ein Diphon beginnt in der Mitte eines Phonems und endet in der Mitte des Nachfolgenden. In der Mitte des Lautes ist das Verhalten der Phoneme nahezu identisch, lediglich der Lautübergang ist kritisch. Durch diese Segmentierung ist der Phonemübergang (Lautübergang) vorgegeben und die allophonischen Eigenschaften sind somit unkritisch. In der deutschen Sprache gibt es ungefähr 2025 Diphone, dieses entspricht allen möglichen Kombinationen der 45 Phoneme (45^2). Diese Kombination der Phoneme ist beliebig erweiterbar, das nächstgrößere Segment ist das Triphon. Dieses beginnt in der Mitte eines ersten Phonems bis zur Mitte des Dritten. In Abbildung 2-5 ist die Zusammensetzung des Diphons und des Triphons dargestellt.

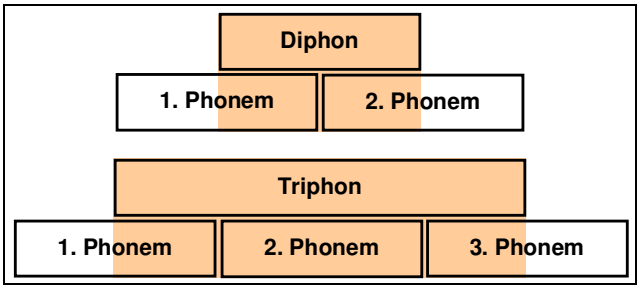


Abbildung 2-5: Aufbau von Diphonen / Triphonen

2.3.3 Silben / Halbsilben

In einigen Sprachsyntheseprogrammen wird die Silbentrennung untersucht, bei solchen Programmen werden auch häufig Halbsilben eingesetzt. Das Prinzip der Halbsilben ist vergleichbar mit dem Prinzip der Diphone, (von der Mitte der ersten Silbe bis zur Mitte der Zweiten).

2.3.4 Morphologische Trennung / Morphem

Bei der Morphologischen Trennung wird in abstrakte Einheiten, die durch primäre Artikulation bzw. Segmentation gewonnen werden unterteilt. Hierbei ist die kleinste bedeutungstragende Einheit das Morphem. Diese Art der Sprachsegmentierung ist nur für die Linguisten von Interesse, da die morphologischen Zusammenhänge sich nicht mit Algorithmen beschreiben lassen. In Abbildung 2-7 sind einige Beispiele der Morphologischen Trennung der Silbentrennung gegenübergestellt.

Wort	Silbentrennung	Morphologische Trennung
Welt	Welt	Welt
Welten (Pl.)	Wel – ten	Welt – en
weltlich	welt – lich	welt – lich
Zettel	Zet – tel	Zettel
verzetteln	ver – zet – teln	ver – zettel – n

Abbildung 2-6: Bsp.: Morphologische Trennung

2.4 Textvorverarbeitung

Bei allen Verfahren ist eine gewisse Vorverarbeitung notwendig, um eine möglichst natürliche Stimme zu erlangen. In Abbildung 2-7 ist der Aufbau eines TTS-Systems dargestellt, Für eine Sprachausgabe werden mindestens die grau dargestellten Blöcke benötigt, wie es gegenwärtig bei diesem Projekt umgesetzt wurde. Zuerst wird der Text vorverarbeitet, hierbei müssen Wörter, Sonderzeichen und Satzzeichen, sowie Zahlen getrennt voneinander behandelt werden. Sonderzeichen und Zahlen können direkt in Wörter überführt werden und die Satzzeichen werden zur Analyse der Wort- und Satzstruktur benötigt. Anschließend wird die Wortstruktur untersucht. Hierzu werden die Grapheme zunächst in Phoneme umgewandelt, wie es bei jeder Art der Sprachsynthese notwendig ist.

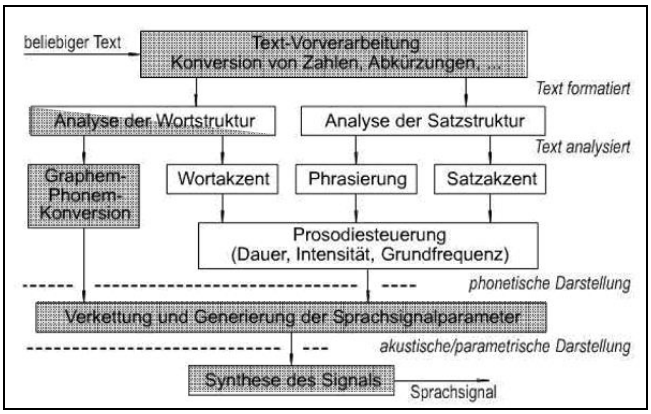


Abbildung 2-7: Aufbau eines TTS-Systems

[Quelle: http://www.ikp.uni-bonn.de/~cwe/SynCL/1.pdf S. 6]

Für eine natürliche Sprachsynthese werden zusätzlich die restlichen weißen Blöcke benötigt. Hierbei wird zusätzlich die Wort- und Satzstruktur analysiert. Nachfolgend wird mit einer Prosodiesteuerung der Wortakzent, die Phrasierung[18] und der Satzakzent angepasst. Für eine solche Prosodiesteuerung wird häufig ein DSP[19] mit einem PSOLA[20]-Verfahren eingesetzt. Mit diesem Verfahren werden die Dauer, Intensität und Grundfrequenz des Sprachsignals beeinflusst. Beispielsweise ist es möglich nun einen Fragesatz von einem Ausrufesatz zu unterscheiden.

Abschließend werden die Sprachsignalparameter generiert, die Sprachsegmente miteinander verkettet und das Sprachsignal ausgegeben.

3 Gewählte Sprachsynthese

Bei der Wahl der Art der Sprachsynthese ist entscheidend, dass diese auch im vorgegebenen Zeitraum bearbeitbar ist und das Ergebnis hierbei möglichst natürlich und verständlich klingt.

Die Artikulatorische Synthese ist nicht geeignet, da hierbei ein großer Regelaufwand notwendig ist und zudem diese Synthese gegenwärtig nicht komplett erforscht ist. Bei der Formantsynthese ist ebenfalls ein großer Regelaufwand erforderlich. Die hierbei entstehende Sprache ist zudem sehr unnatürlich und aus diesem Grunde ungeeignet. Die auch heutzutage bei mobilen Anwendungen häufig eingesetzte Konkatenierende Synthese verspricht die besten Ergebnisse. Zudem ist hierbei ein modularer Aufbau möglich, der es ermöglicht das System immer weiter auszubauen und somit die Qualität zu steigern, theoretisch bis hin zur Unit Selection.

4 Hardware

Für die Umsetzung wird ein „Altium LiveDesign Evaluation Board“ eingesetzt, welches nahezu alle benötigten Anforderungen für die gewählte Art der Sprachsynthese erfüllt. Für die Texteingabe kann die „serielle Schnittstelle (RS 232)“ oder der „PS/2“ Anschluss verwendet werden. Die Sprachausgabe geschieht entweder über die integrierten Lautsprecher oder extern über den Klinkenstecker. Die Verarbeitung geschieht über ein FPGA „Altera Cyclone EP1C12“ mit einer Taktfrequenz von 50 MHz. Außerdem sind noch wie in Abbildung 4-1 dargestellt folgende Komponenten vorhanden:

- sieben Segmentanzeige / LEDs
- Taster / Dipschalter
- I/O Ports

Diese können für die Konfiguration, Funktionsüberwachung und Erweiterung des TTS- Systems eingesetzt werden.

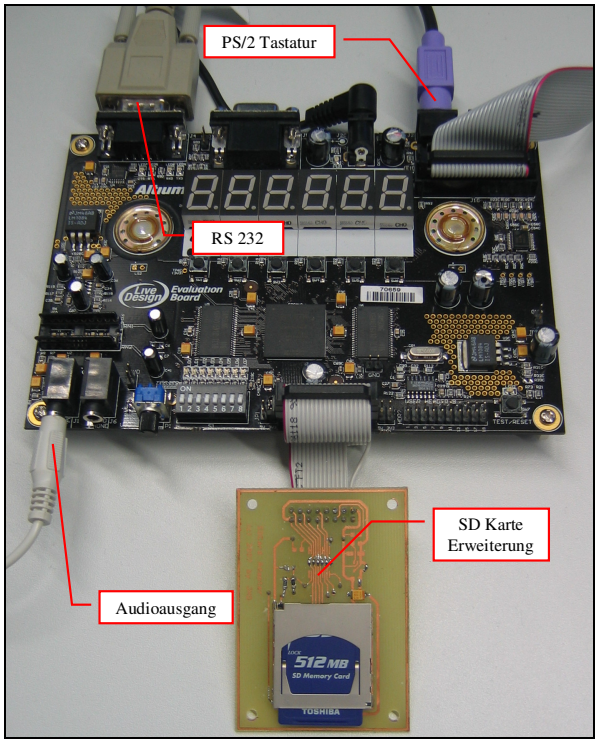


Abbildung 4-1: Altium LiveDesign Board

[Quelle: http://www.altium.com/Products/AltiumDesigner/LiveDesign]

Lediglich der Speicher auf dem Altium Board ist mit seinen 512 kB nicht ausreichend. Für die Audiodaten im Diphoninventar werden ~120 MB Speicherkapizität benötigt. Aus diesem Grund wird eine SD-Karte zur Speicherung der Daten verwendet. Diese SD-Karten-Erweiterung wird mit den I/O Ports verbunden. Der FPGA kann dann mit Hilfe der Implementierung einer SPI[21]-Schnittstelle die SD Karte auslesen.

5 Umsetzung

Dieses Kapitel beschäftigt sich mit der Realisierung des Sprachsynthesesystems. Bevor die Umsetzung auf dem FPGA durchgeführt werden kann, müssen die von dem Institut für Maschinelle Sprachverarbeitung (IMS) der Universität Stuttgart erhaltenen Daten, das Graphem-zu-Phonem-Regelwerk, sowie das Diphoninventar in ein hardwareorientiertes Format konvertiert werden.

[18] Behandlung verschiedener Töne hinsichtlich Lautstärke, Rhythmik, Artikulation und Pausensetzung

[19] **D**igitaler **S**ignal **P**rozessor

[20] **P**itch **S**ynchronous **O**verlap **A**dd

[21] **S**erial **P**eripheral **I**nterface

5.1 Regelwerk zur Umwandlung

Das Regelwerk der IMS Stuttgart muss für die Nutzung auf dem FPGA in ein einfach zu verarbeitendes Datenformat konvertiert werden. Da das ursprüngliche Regelwerk bei einer Softwarelösung eingesetzt wird, ist es für die direkte Nutzung ungeeignet. Das Regelwerk sollte für eine einfache Verarbeitung in allen Zeilen gleich strukturiert sein, so dass sich die gewünschten Daten immer an der gleichen Stelle in der Zeile befinden.

Dieses Format ist auf Hardwareebene schlecht einsetzbar, da sehr viele Regeln eingesetzt werden. Es wird zum Beispiel auf Silben überprüft. Für diese Überprüfung ist ein zusätzlicher Regelaufwand notwendig, welcher im Rahmen der Studienarbeit nicht zu implementieren ist.

Außerdem sind viele Regeln nur selten vorhanden, bzw. nur in wenigen Wörtern angewandt. Für die Implementierung wurde das Regelwerk vereinfacht, indem die am häufigsten vorhandenen Regeln übernommen wurden. Außerdem wurde in speziellen Fällen, bei der Überprüfung von Buchstabenfolgen oder Silben, die komplette Buchstabenfolge, bzw. Silbe mit übernommen. Das so entstandene Regelwerk besitzt nach dieser Optimierung nur noch ungefähr 120 Umsetzungseinträge, mit den folgenden sechs Regeln:

0. Keine Regel, somit Standardwert
1. Graphemkombi vor einem Konsonanten
2. vor mehreren Konsonanten
3. vor einem Vokal
4. am Wortanfang
5. am Wortende
6. vor den Buchstaben „N“ oder „L“

Nachfolgend ein Auszug, aus dem so entstandenen Regelwerk.

-Graphemkombi	Regel –Nr.	Phonemkombi
ig	5	I C
ih	0	i:
ii	0	i:
ion	0	s j o: n
j	0	j
jj	0	j
k	0	k
l	0	l
ll	0	l
m	0	m
n	0	n
ng	0	N
ngau	0	n g aU
ngay	0	n g aI

Tabelle 5-1: Auszug: Optimiertes Regelwerk

Dieses so entstandene Regelwerk muss zur Verwendung auf dem FPGA noch in ein bitorientiertes Format umgewandelt werden. Um eine hardwarenahe Verarbeitung auf dem Altium-Board zu gewährleisten, soll die Länge der Einträge nicht mehr als 64 Bit betragen. Zur Minimierung der Zeilenlänge werden die Graphemkombinationen zerlegt in Einzelgrapheme, welche in Ebenen mit Hilfe einer Baumstruktur eingeteilt werden. Die Baumstruktur ermöglicht es auch lange Wörter in das Regelwerk einzubinden. In Tabelle 5-2 ist diese Baumstruktur anhand der vier Wörter: „Abend, auch und Auge“ dargestellt.

	Baumstruktur der ASCII - Zeichen				
Ebene / Zeile	0 00000	1 00001	2 00010	3 00011	4 00100
1	a				
2		b			
3			e		
4				n	
5					d
9		u			
10			c		
11				h	
12			g		
13				e	

Tabelle 5-2: Beispiel Baumstruktur der Graphemkombination

Für die schnellere Verarbeitung wird zusätzlich eine Look-Up-Tabelle verwendet, hierbei wird ein direkter Zugriff auf die Einträge durch Angabe des Anfangsbuchstaben ermöglicht. Um auch lange Wörter im Regelwerk zu ermöglichen, werden zudem Phonemzusatzzeilen eingeführt. Das so entstandene Format sieht beispielsweise so aus, wie in Tabelle 5-3 dargestellt.

	LUT Eintrag?	Wörterbuch Eintrag?	Phonem Zusatzzeile?	Phonem vorhanden?	Reserve	Graphem ASCII	Ebene	Regel	LUT - Adresse Phonem je. 6 Bit
Adr.	1 Bit	1 Bit	1 Bit	1 Bit	4Bit	8 Bit	5 Bit	7 Bit	24 Bit / 36 Bit
:	:	:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:	:	:
32	1	0	0	0	0	a	0	0	65
:	:	:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:	:	:
65	0	1	0	1	0	a	0	0	a
66	0	1	0	1	0	a	0	2, 5	a:
67	1	0	0	0	0	u	1	0	70
68	0	1	0	1	0	b	1	0	ab
69	0	1	0	1	0	b	1	3	a: b
70	0	1	0	1	0	u	1	0	au
71	0	1	0	1	0	f	2	0	auf
72	0	1	0	0	0	g	3	0	
73	0	1	0	0	0	a	4	0	
74	0	1	0	0	0	b	5	0	
75	0	1	0	1	0	e	6	0	aufgab
76	0	0	1	1	0	0	0	0	e
77	0	1	0	0	0	t	2	0	
78	0	1	0	1	0	o	3	0	auto
:	:	:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:	:	:
Look - Up - Eintrag									
Regeleintrag									
Phonemzusatzzeile									

Tabelle 5-3: Beispiel bitorientiertes Datenformat zur Implementierung auf dem FPGA

Die grundsätzliche Struktur einer Eintragszeile besteht aus fünf Bereichen:

- 8 Statusbits
- Graphem (8 Bit / ASCII)
- Ebene (5 Bit / Binär)
- Regeln (7 Bit / 1 Bit je Regel)
- 6 Phoneme (je 6 Bit), bzw. im LUT[22]-Eintrag die Sprungadresse (24 Bit)

In den acht Statusbits ist definiert, ob es ein LUT Eintrag, ein Wörterbucheintrag oder eine Phonemzusatzzeile ist. Ein weiteres Statusbit zeigt an, ob Phoneme in der Zeile vorhanden sind, die restlichen vier Bit sind Reserve. Darauf folgen das Graphem und die zugehörige Ebene, welche durch die angewandte Baumstruktur, die Buchstabenfolge repräsentiert. Zur Überprüfung der Regeln sind sieben Bit reserviert, pro Regel wird ein Bit verwendet, zudem ist für die Erweiterungen ein weiteres Reservebit vorhanden. Der letzte Eintrag ist entweder die Sprungadresse bei den LUT-Einträgen, oder bis zu sechs Phoneme mit je sechs Bit. Für die Phoneme wurde die in Tabelle 5-4 dargestellte „Gewählte Zuordnung“ verwendet, womit nur sechs Bit je Phonem benötigt werden.

Gewählte Zuordnung	SAMPA Zeichen	Gewählte Zuordnung	SAMPA Zeichen
(000000) 0	_	(010111) 23	b
(000001) 1	I	(011000) 24	t
(000010) 2	E	(011001) 25	d
(000011) 3	a	(011010) 26	k
(000100) 4	O	(011011) 27	g
(000101) 5	U	(011100) 28	pf
(000110) 6	Y	(011101) 29	ts
(000111) 7	9	(011110) 30	tS
(001000) 8	i:	(011111) 31	f
(001001) 9	e:	(100000) 32	v
(001010) 10	E:	(100001) 33	s
(001011) 11	a:	(100010) 34	z
(001100) 12	o:	(100011) 35	S
(001101) 13	u:	(100100) 36	Z
(001110) 14	y:	(100101) 37	C
(001111) 15	2:	(100110) 38	j
(010000) 16	aI	(100111) 39	x
(010001) 17	aU	(101000) 40	h
(010010) 18	OY	(101001) 41	m
(010011) 19	@	(101010) 42	n
(010100) 20	6	(101011) 43	N
(010101) 21	?	(101100) 44	l
(010110) 22	p	(101101) 45	R

Tabelle 5-4: Gewählte Phonemzuordnung

Nicht benötigte Bits werden mit Nullen beschrieben. Das Regelwerk (Tabelle 5-3) ist zudem in drei Adressbereiche eingeteilt:

1. Kopf im Adressbereich 0…31
2. Look-up-Tabellenbereich zwischen 32 und 63
3. Regelwerk zwischen 64 und …

Der Kopf kann beliebig genutzt werden z.B. für Beschreibungen. Im Look-up-Bereich wird für die Buchstaben „A“ bis „Z“, sowie für die Buchstaben „Ä“, „Ö“, „Ü“ und sonstige Zeichen eine direkte Sprungadresse angegeben. Im Regelwerk sind die Wörterbuch Einträge gegebenenfalls mit Zusatzzeilen vorhanden.

Optional ist es möglich, zur schnelleren Suche zusätzliche LUT Eintrage im Regelwerk zu implementieren, wie z.B. in Zeile 67 beim „AU“, wo auf Zeile 70 gezeigt wird. Das Ende des Regelwerkes wird durch eine Nullzeile gekennzeichnet.

5.2 Diphoninventar

Bei der Implementierung auf dem FPGA besteht das Diphoninventar aus WAV[23]-Dateien. Die einzelnen Diphon-Audiodateien beinhalten Kunstwörter, die von einem professionellen Sprecher gesprochen wurden. Die Gesamtgröße beträgt 120 MB. Werden nur die Nutzdaten verwendet, so wäre es eine Größe von 13,3 MB. Durch Einführung einer einfachen Komprimierung ist es möglich, den Speicherbedarf auf nur wenige MB zu vermindern.

Jede Datei besitzt drei Schnittgrenzen zur Kennzeichnung des Diphons.

- Diphonanfang
- Diphonende
- Lautübergang

Für die Verkettung der Diphone werden nur der Diphonanfang, sowie das Ende verwendet. Der Lautübergang und die Nutzung des Kunstwortes über die Grenzen hinaus wären bei der Einführung der Prosodie notwendig.

[22] **L**ook-**u**p-**T**abelle

[23] **Wav**e-Dateiformat ist ein Containerformat zur digitalen Speicherung von Audiodaten

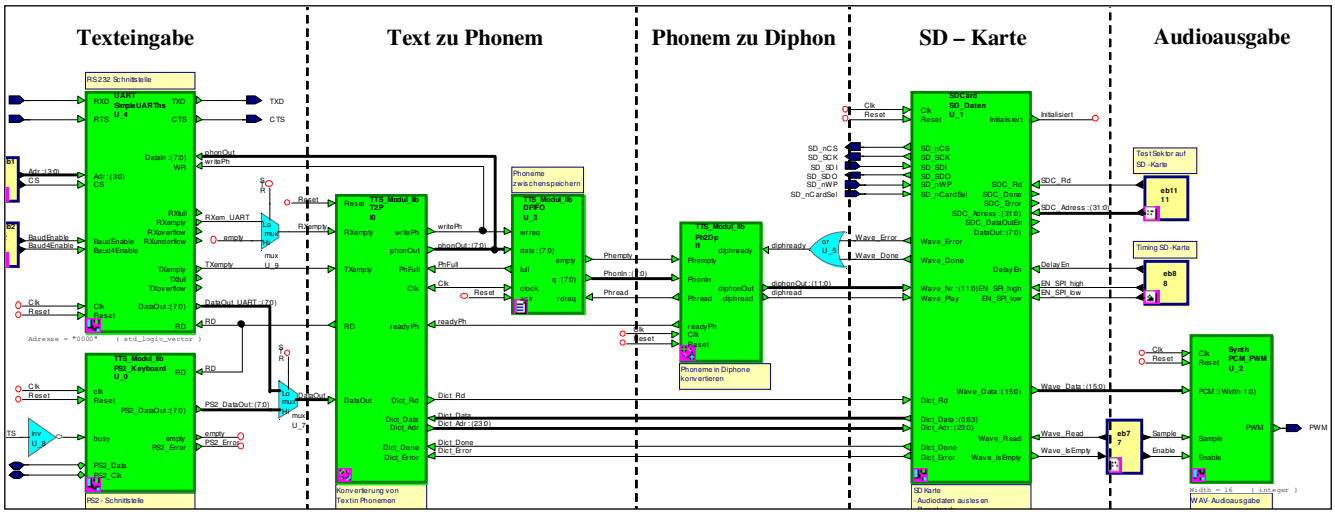


Abbildung 5-1: Blockdiagramm TTS

5.3 Umsetzung auf dem FPGA

In Abbildung 5-1 ist das Gesamt Blockdiagramm dargestellt.

Das Blockdiagramm ist in der Abbildung in fünf Gruppen eingeteilt:

1. **Texteingabe**, wahlweise über „UART[24]“ oder „PS2_Keyboard“ möglich. Bei aktiver RS232 Übertragung wird die Tastatur deaktiviert und die Werte der seriellen Schnittstelle über den Multiplexer „U_7“ an das „T2P“ weitergegeben. Wenn die serielle Schnittstelle inaktiv ist, wird der eingegebene Text der Tastatur über den Multiplexer „U_7“ übermittelt.
2. Beim **Text zu Phonem** wird der Text in Phoneme umgesetzt, hierzu durchsucht „T2P“ das Phonemumsetzungswörterbuch auf der SD Karte (siehe 4.) und übergibt die Phoneme dem „DFIFO[25]“, zur Zwischenspeicherung.
3. Die **Phonem zu Diphon** Umsetzung setzt die Phoneme zu Diphonen zusammen und steuert die Komponente „SD_Daten“ an.
4. Auf der **SD Karte** sind das Phonemumsetzungswörterbuch (siehe 2.) und das Diphoninventar gespeichert. Anhand der Adresse, welche dem Diphonnamen entspricht, wird mit dem Funktionsblock „SD_Daten“, der die SD Karte bedient, die entsprechende Audiodatei aus dem Speicher zwischen den entsprechenden Schnittgrenzen ausgegeben.
5. Bei der **Audioausgabe** werden die Audiodaten im WAV (RAW) Format enkodiert und pulsweitenmoduliert und auf dem Lautsprecher ausgegeben.

5.3.1 Text zu Phonem

Die Umwandlung der Buchstaben in die Phoneme wird in einem Automat vorgenommen. Der prinzipielle Ablauf ist in Abbildung 5-2 dargestellt.

1. Die Zeichen werden einzeln eingelesen, dabei werden die Großbuchstaben in Kleinbuchstaben umgewandelt.
2. Es wird überprüft ob es sich um ein gesprochenes Zeichen oder ein ungesprochenes Zeichen handelt.
3. Die gesprochenen Zeichen werden eingelesen, bis ein Wort komplett ist und mit einem ungesprochenem Satzzeichen, wie z.B.: „SPACE, ENTER, KOMMA“ beendet wird.
4. Das Wort wird anschließend gespeichert.
5. Über die LUT im Umsetzungswörterbuch wird auf dem entsprechenden Eintrag mit dem Anfangsbuchstaben gesprungen.
6. Es werden die Wörterbucheinträge auf der SD Karte zeilenweise durchsucht, bis die letzte Entsprechung gefunden ist. Da das Regelwerk sortiert ist, wird die Suche abgebrochen, wenn der Wörterbucheintrag größer ist als das eingelesene Wort.
7. Die entsprechenden Phoneme werden ausgeben.
8. Wenn noch nicht das ganze Wort abgearbeitet ist werden die Schritte 5. bis 7. wiederholt, bis das Wort komplett ausgegeben wurde.
9. Ist das Wort komplett abgearbeitet, wird eine Pause, das Phonem „NULL“ angefügt. Nun kann das nächste Wort verarbeitet werden.

[24] **U**niversal **A**synchronous **R**eceiver **T**ransmitter
[25] **D**istributed **F**irst **I**n **F**irst **O**ut

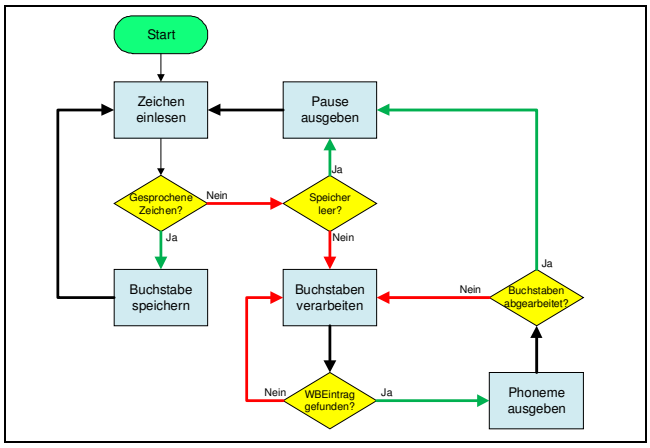


Abbildung 5-2: Ablauf Text zu Phonem

5.3.2 SD Karte

Auf der SD Karte sind die Phonemumsetzungsregeln und die Audiodaten des Diphoninventars gespeichert. Für das TTS-System wurde ein spezielles Dateisystem mit drei Partitionen erstellt, wie in Tabelle 5-5 dargestellt.

MBR	(Master – Boot – Record)
1. Partition:	**FAT 16 – Partition** ■ Zur Speicherung von sonstigen Dokumenten
2. Partition:	**Phonemumsetzungsregelwerk** ■ LUT – Einträge ■ Wörterbuch – Einträge
3. Partition:	**Diphoninventar** ■ LUT – Einträge mit Schnittgrenzen ■ Audiodateien im RAW – Format

Tabelle 5-5: Partitionierung der SD Karte

Die 1. Partition im FAT16 Format dient nur der Speicherung von zusätzlichen Dokumenten. In der 2. und 3. Partition sind die Daten des Phonemumsetzungsregelwerkes und das Diphoninventars gespeichert. Die Daten der 2. und 3. Partition sind am PC[26] nur mit spezielle Programme les- bzw. schreibbar, wie z.B. mit dem Programm „HxD[27]“.

Das Regelwerk und das Diphoninventar sind mit separat gruppierten Signalen am SD Karten Modul versehen, wie in Abbildung 5-3 dargestellt.

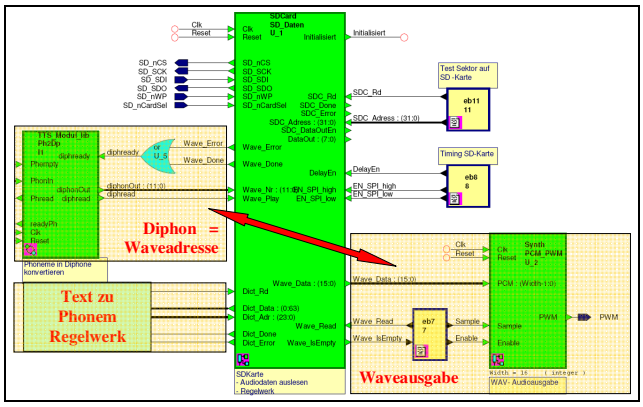


Abbildung 5-3: Komponente SD_Daten

Um Unterbrechungen der Audioausgabe zu vermeiden hat die Ausgabe der Audiodaten höchste Priorität. Der Zugriff auf die SD Karte findet wie folgt statt:

- Die Komponente „SD_Daten“ gibt nach Vorgabe der Wörterbuchadresse „Dict_Adr“, welche wie in der Abbildung 5-3 dargestellt, von dem „Text zu Phonem“ übergeben wird, die dazu gehörige 64 Bit lange Umsetzungszeile „Dict_Data“ zurück.
- Wird dem Block ein Diphon, bzw. eine „Wave_Nr“ übergeben, so gibt die Komponente die WAV- Daten im 16 Bit PCM[28]- Format aus, wenn die Daten von der Komponente „Synth / PCM_PWM“ angefordert werden.

6 Zusammenfassung

Das in dieser Ausarbeitung realisierte Sprachsynthesesystem besitzt eine verständliche Sprachausgabe und stellt somit eine sehr gute Grundlage für die Weiterentwicklungen an diesem Projekt dar.

Folgende Funktionen wurden bei diesem TTS-System realisiert:

- Das TTS System wandelt den über die PS/2 Tastatur oder die serielle Schnittstelle vorgegebenen Text in Phoneme mit Hilfe eines eigens für die Hardware entwickelten Umsetzungswörterbuches um.
- Aus den Phonemen werden die Diphone generiert.
- Mittels der Diphone werden die entsprechenden Audiodateien aus einer SD Karte gelesen. Die SD Karte wird über eine SPI-Schnittstelle betrieben.
- Die Diphon-Audiodateien werden letztendlich als Sprachsignal direkt hintereinander ausgegeben, somit hart verkettet und monoton ausgegeben.

[26] **P**ersonal **C**omputer
[27] Freeware Hex Editor und Disk Editor
[28] **P**uls - **C**ode - **M**odulation

7 Ausblick

Um eine natürliche Sprachausgabe zu erhalten, sollte eine Prosodiesteuerung eingeführt werden. Hierfür ist die Implementierung eines DSP mit einem PSOLA-Verfahren notwendig. Durch das PSOLA können dann die Dauer, die Intensität und die Grundfrequenz angepasst werden. Zudem wird hierdurch eine Art weiche Verkettung der Diphone erzielt. Im FPGA-Projekt müsste hierfür eine DSP-Komponente für das PSOLA entwickelt werden.

Des Weiteren muss bei der Prosodiesteuerung die Satz- und Wortstruktur untersucht werden. Die Parameter für die richtigen Wortakzente könnten im Regelwerk auf der SD Karte für das jeweilige Wort hinterlegt werden. Es wäre vorstellbar, hierfür das Regelwerk von 64 Bit auf 128 Bit Zeilenlänge zu erweitern. Der Satzakzent müsste auf dem FPGA abgehandelt werden. Hier wäre eine Unterscheidung nach Satzzeichen (Komma, Punkt, Fragezeichen, Ausrufezeichen) möglich.

Ferner sind folgende Verbesserungen vorstellbar:

- Optimierung der Phonemumsetzungsregeln
- Einführung eines Ausnahmewörterbuch
- Extrabehandlung von Zahlen (Unterscheidung zwischen Datum, Uhrzeit, Preis usw.)
- Verbesserung der Sonderzeichenbehandlung
- Abkürzungserkennung (z.B. komplett Aussprechen „zum Beispiel“)
- PS/2 Tastatureingabe optimieren: Steuertastenbehandlung und z.B. ermöglichen mit Backspace letzten Buchstaben zu löschen.

8 Quellverzeichnis

[1] http://www.ims.uni-stuttgart.de/

[2] http://www.cis.uni-muenchen.de/

[3] http://www.ikp.uni-bonn.de/

[4] http://www.ego4u.de/de/dictionary/ipa

[5] http://www.ias.et.tu-dresden.de/

[6] http://www.altium.com/

Operationssystem für den SIRIUS Softcore Processor

Florian Zowislok
Hochschule Offenburg, Badstraße 24, 77652 Offenburg
07821/205363, florian.zowislok@fh-offenburg.de

Zusammenfassung

Den Hauptbestandteil des Operationssystems stellt der Zugriff auf SD-Karten mit dem Dateisystem FAT16 von Microsoft dar. Für die Bedienung wurde ein Kommandozeileninterpreter implementiert. Als Ein- und Ausgabegerät dient ein PC mit einem speziellen Terminalprogramm, welcher über USB mit dem Emulationsboard des SIRIUS Softcores verbunden ist. Das System wird über die Eingabe von Befehlen am Terminal gesteuert.

Der SIRIUS Softcore kann nur vom Flash des Emulationsboards booten. Da das Betriebssystem selbst jedoch auf der SD-Karte gespeichert werden soll, ist ein Basis-Betriebssystem erforderlich, welches im Flash abgelegt ist. Das Basis-Betriebssystem lädt gleich nach dem Start das eigentliche Betriebssystem von der SD-Karte. Falls jedoch keine SD-Karte gesteckt ist, ermöglicht das Basis-Betriebssystem mit einem Kommandozeileninterpreter einige Grundfunktionen.

1. Grundlagen

1.1. SIRIUS Prozessor

Der SIRIUS Softcore ist ein an der Hochschule Offenburg entwickelter RISC Prozessor, welcher für die Programmierung in C optimiert wurde und sowohl im 16-Bit als auch im 32-Bit Modus betrieben werden kann [1]. Für den 32-Bit Modus werden dazu immer zwei 16-Bit Register zusammengeschaltet. Die aktuell mögliche Taktfrequenz liegt bei 48 MHz.

1.2. Emulationsboard

Auch das Emulationsboard *(Abbildung 1.1)* wurde vollständig an der Hochschule Offenburg entwickelt. Das Cyclone III FPGA von ALTERA bildet das Kernstück des Boards, in welchem der SIRIUS Prozessor emuliert wird. Im FPGA stehen zusätzlich 64 KB RAM und verschiedene Peripherien wie SPI, RS232 oder USB für den SIRIUS Prozessor zur Verfügung. Da die die 64 KB RAM mit 16 Bit adressiert werden können, wird der Prozessor im 16-Bit Modus betrieben. Des Weiteren befinden sich auf dem Board unter anderem LEDs, DIP-Schalter, ein Piezo-Lautsprecher, ein USB-Anschluss, ein RS232-Anschluß, ein Reset-Knopf, ein Flash-IC zum Booten des Systems sowie ein SD-Karten-Anschluss.

Abbildung 1.1: Emulationsboard mit SD-Karte

1.3. SIRIUS IDE

Die Entwicklungsumgebung SIRIUS IDE basiert auf der früheren FHOP-Umgebung und wurde an den SIRIUS Prozessor angepasst. Mit der IDE kann in der Programmiersprache C oder in Assembler programmiert werden. Der Compiler der SIRIUS IDE übersetzt den C-Code zuerst in Assembler-Code und anschließend mit dem CRASH Assembler ins Intel-Hex-File Format. Des Weiteren verfügt die IDE über einen Simulator, mit dem der fertig übersetzte Code getestet werden kann. Da im Simulator die SD-Karte nicht nachgebildet ist, können damit Programme, die auf die SD-Karte zugreifen, nicht getestet werden. *Abbildung 1.2* zeigt die SIRIUS IDE mit einem Testprogramm.

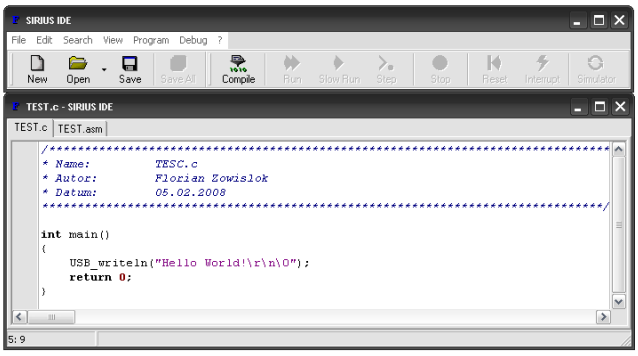


Abbildung 1.2 SIRIUS DIE

1.4. Dateisystem

Als Dateisystem wird FAT16 von Microsoft verwendet [2]. Zur Vereinfachung werden jedoch keine langen Dateinamen unterstützt. Es sind also nur acht Zeichen für einen Datei- bzw. Verzeichnisname möglich, wobei nur große Buchstaben, Ziffern und einige Sonderzeichen zulässig sind.

Beim Dateisystem FAT16 werden immer 512 Bytes zu einem Sektor zusammengefasst. Diese werden so eingeteilt, dass auf dem Datenträger verschiedene Bereiche zur Verfügung stehen, wie in *Abbildung 1.3* dargestellt. Im Datenbereich werden jeweils 2^n Sektoren zu einem Cluster zusammengefasst, wobei n so gewählt wird, dass jedes Cluster noch mit 16 Bit adressiert werden kann. Die Cluster werden fortlaufend durchnummeriert, wobei das erste Cluster mit der Nummer 2 (0x0002) beginnt. Die größte Clusternummer ist 65526 (0xFFF6), womit maximal 65525 Cluster möglich sind.

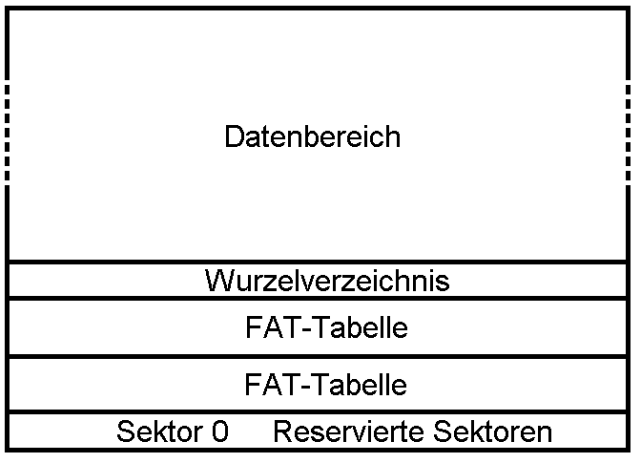


Abbildung 1.3 Datenträger mit FAT16

Im ersten Sektor des Datenträgers, dem Sektor 0, sind alle Daten hinterlegt, die benötigt werden, um auf das Dateisystem zuzugreifen. Anhand dieser Daten kann beispielsweise berechnet werden, an welcher Adresse ein bestimmter Bereich beginnt. Das Wurzelverzeichnis hat eine feste Größe, weshalb die Anzahl von Verzeichniseinträgen im Wurzelverzeichnis begrenzt ist. Jeder Datei wird ein 32-Byte Verzeichniseintrag (*Abbildung 1.4)* zugeordnet, wobei Unterverzeichnisse wie Dateien behandelt werden.

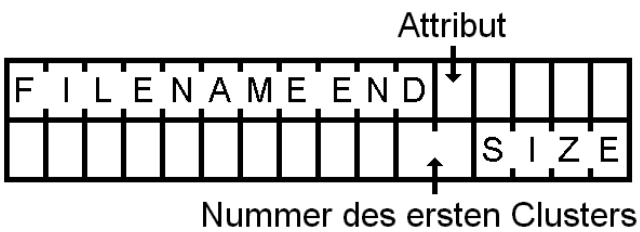


Abbildung 1.4: 32-Byte Verzeichniseintrag

In den ersten acht Bytes wird der Datei- bzw. der Verzeichnisname gespeichert, die Dateiendung in den folgenden drei Bytes, wobei diese bei einem Verzeichnis alle den Wert 0x20 enthalten, was einem Leerzeichen entspricht. Im Attribut-Byte stellen die einzelnen Bits verschiedene Eigenschaften dar, wie z.B. „versteckt“ oder „schreibgeschützt“. Besonders wichtig dabei ist das Bit, welches anzeigt, ob es sich um eine Datei oder um ein Verzeichnis handelt. In dem zwei Bytes großen Feld „Nummer des ersten Clusters“ wird angegeben, in welchem Datencluster die Datei bzw. das Verzeichnis beginnt. Die letzen vier Bytes enthalten die Größe der Datei in Bytes. Bei einem Verzeichnis enthält dieses Feld den Wert „0“. In den übrigen 14 Bytes werden verschiedene Zeitstempel gespeichert, auf die nicht näher eingegangen wird.

Die FAT-Tabelle wird meist doppelt auf dem Datenträger angelegt. Laut Spezifikation wären jedoch auch nur eine oder mehr als zwei möglich. Die FAT-Tabelle bildet alle Daten-Cluster ab, wobei jedem Cluster ein 16-Bit Wert zugeordnet wird:

0x0000:	Cluster ist frei
0x0002 bis 0xFFF6:	Clusternummer des nächsten Clusters
0xFFF7:	Cluster ist defekt
0xFFFF:	Letztes Cluster

Abbildung 1.5 zeigt, wie die Clusterverwaltung auf dem Datenträger funktioniert. In den Daten-Clustern befinden sich die tatsächlichen Dateien, wobei der Verzeichniseintrag einer Datei nur die Nummer des ersten Clusters enthält. Datei 1 beginnt mit Cluster A0. Der 16-Bit Wert der FAT-Tabelle gibt an, dass es das letzte Cluster der Datei ist. Datei 2 beginnt in Cluster A2. Der zugehörige Eintrag in der FAT-Tabelle gibt an, dass die Datei im Cluster A3 fortgesetzt wird, wobei der entsprechende Eintrag in der FAT-Tabelle wiederum die Nummer des nächsten Clustern angibt. Dieser Vorgang wird bis zum letzten Cluster wiederholt, welchem in der FAT-Tabelle der Wert 0xFFFF zugeordnet ist. Die Einträge der FAT-Tabelle ergeben somit eine einfach verkettete Liste von Clustern. Für die freien Cluster steht in der FAT-Tabelle der Wert 0x0000.

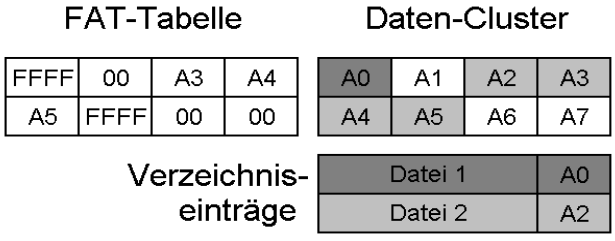


Abbildung 1.5: Clusterverwaltung

2. Betriebssystem

2.1. Kommandozeile

Das Betriebssystem besteht im Wesentlichen aus einem Kommandozeileninterpreter. Die Ein- und Ausgabe erfolgt über das Terminalprogramm USB-COM, welches auf einem Windows PC ausgeführt werden kann, der per USB mit dem Emulationsboard verbunden ist. Mit dem Bedienterminal können Zeichenketten in beide Richtungen übertragen werden. Ankommende Zeichenketten werden im oberen Textfeld ausgegeben, im unteren Textfeld können Befehle eingegeben werden, die erst mit Betätigen der Enter-Taste gesendet werden. Durch Betätigung der Schaltfläche „Clear Text“ wird das obere Textfeld geleert. Mit der Schaltfläche „Load HEX File“ kann eine komplette Intel-Hex-Datei vom PC zeilenweise an das Emulationsboard übertragen werden. *Abbildung 2.1* zeigt das USB-COM bei angeschlossenem Emulationsboard, nachdem das Betriebssystem gestartet wurde.

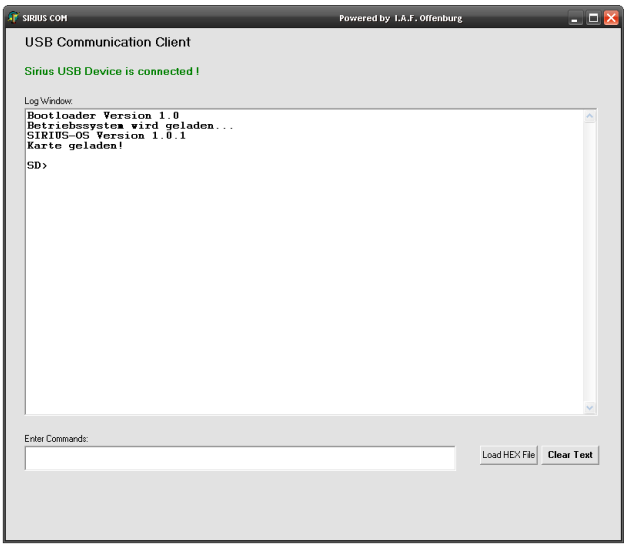


Abbildung 2.1: USB_COM

Beim Start wird zunächst die aktuelle Betriebssystem-version auf dem Terminal ausgegeben. Anschließend wird die SD-Karte initialisiert und die Dateisystem-parameter geladen. Die Eingabeaufforderung erfolgt nun in einer Endlosschleife. Darin wird zuerst der aktuelle Pfad ausgegeben und dann auf eine Eingabe im Terminal gewartet. Nach der Eingabe der Befehlszeile wird nacheinander geprüft, ob die SD-Karte gesteckt ist, initialisiert wurde, und das Dateisystem FAT16 enthält. Sobald eine Bedingung nicht zutrifft, wird die Eingabe verworfen und die Endlosschleife von vorne durchlaufen. Zusätzlich wird eine Fehlermeldung ausgegeben bzw. die Karte neu initialisiert. Werden jedoch alle drei Bedingungen erfüllt, wird die Befehlszeile interpretiert.

2.2. Interpretation der Befehlszeile

Bevor die Befehlszeile interpretiert wird, werden alle Kleinbuchstaben in Großbuchstaben umgewandelt. Danach wird geprüft, ob die Eingabe mit einem vordefinierten Befehl übereinstimmt und die entsprechende Funktion aufgerufen. Falls die Eingabe mit keinem Befehl übereinstimmt, wird überprüft, ob sich auf der SD-Karte im aktuellen oder im Betriebssystem-Verzeichnis eine Hex-Datei mit entsprechendem Dateinamen befindet und gegebenenfalls ausgeführt. Beim Aufruf der jeweiligen Funktion wird ein Zeiger auf die Befehlszeile übergeben, der mit dem Wert „*X*“ um die Länge des Befehls korrigiert werden muss, damit er auf die Parameter des Befehls zeigt, wie in *Abbildung 2.2* dargestellt.

Abbildung 2.2 Zeiger auf Parameter

Folgende Befehle können in die Befehlszeile eingegeben werden:

DIR:	Zeigt den Inhalt des aktuellen Verzeichnisses
CD:	Wechselt das Verzeichnis
MD:	Erstellt ein Verzeichnis
RD:	Löscht ein Verzeichnis
DEL:	Löscht eine Datei
LOAD:	Lädt eine Datei vom Terminal auf die SD-Karte
RENAME:	Benennt eine Datei um
UPDATE:	Schreibt ein Programm von der SD-Karte in den Flash
LED OFF:	Schaltet die LED Anzeige des Timers aus
LED ON:	Schaltet die LED Anzeige des Timers ein

BELIEBIGE EINGABE:

Bei einer beliebigen Eingabe wird auf der SD Karte ein Hex-Programm mit genau diesem eingegebenen Dateinamen gesucht und ausgeführt. Dabei kann es sich um Anwendungen (*Kapitel 2.4*) oder um auf die SD Karte ausgelagerte Programm-Module handeln (*Kapitel 2.5*). Der Anwender merkt dabei keinen Unterschied gegenüber den direkt implementierten Befehlen.

2.3. Basis-Betriebssystem

Beim Einschalten des Emulationsboards kann zunächst nur der Flash gelesen werden. Da sich das eigentliche Betriebssystem jedoch auf der SD-Karte befindet, wird zunächst ein Basis-Betriebssystem aus dem Flash geladen, welches anschließend das eigentliche Betriebssystem von der SD-Karte lädt. Dieser Bootvorgang ist in *Kapitel 3* ausführlich beschrieben. Falls es jedoch nicht möglich ist das Betriebssystem von der SD-Karte zu laden, weil z.B. eine Datei oder auch die komplette SD-Karte nicht vorhanden sind, soll mit dem Basis-Betriebssystem eine gewisse Grundfunktionalität mittels Eingabeaufforderung möglich sein.

Grundlage des Basisbetriebssystems ist ein vorläufiger Entwicklungsstand des eigentlichen Betriebssystems, welcher entsprechend angepasst wurde. Einzelne Funktionen wurden komplett entfernt und stattdessen einige auf die SD-Karte ausgelagerte Programme direkt implementiert. Das Basisbetriebssystem enthält jedoch keine neuen Funktionen und wird deshalb nicht detaillierter beschrieben.

Das Basis-Betriebssystem verfügt ebenso wie das eigentliche Betriebssystem über eine Befehlszeile, die auch genauso funktioniert. Der Zugriff auf die SD-Karte ist jedoch nicht möglich. Stattdessen sind die Programm-Module aus *Kapitel 2.5* direkt implementiert und werden auch genauso aufgerufen. Ebenso ist das Ein- und Ausschalten der LED-Anzeige möglich. Folgende Befehle werden unterstützt:

HELP LHX RUN MEM EDT PTR PTW (*Kapitel 2.5*)

LED OFF LED ON (*Kapitel 2.2*)

2.4. Anwendungen

Mit der SIRIUS-IDE kann ein Benutzer eigene Anwendungen programmieren. Dazu wird bei der Kompilierung im Assembler-Code die Datei „BIOS.asm“ eingebunden. Diese Datei enthält jedoch nicht das BIOS, sondern Referenzen auf bestimmte BIOS-Funktionen bzw. globale Daten, da das eigentliche BIOS schon im Betriebssystem enthalten ist. Durch das Einbinden der Referenzen auf BIOS-Funktionen kann der Anwender die Funktionen verwenden, die schon im BIOS implementiert sind. Außerdem werden somit Funktionen gekapselt, auf die der Anwender keinen Zugriff haben soll. Zusätzlich ist angegeben, bei welcher Speicheradresse der Programmcode beginnen soll. Bei Anwendungen hat diese Startadresse den Wert 0x8000. Bei der Kompilierung ins Intel-Hex-Format wird der Programmcode festen Speicheradressen zugeordnet, beginnend mit der Startadresse. Es stehen maximal 28 KB RAM für die Anwendung zur Verfügung. Somit hat die höchste zulässige Adresse den Wert 0xEFFF.

Es gibt zwei Möglichkeiten eine Anwendung auszuführen:

1. Die Anwendungen werden mit dem Befehl „LHX“ über die USB-Schnittstelle vom PC in den RAM geladen. Mit dem Befehl „RUN 8000“ wird das Programm anschließend gestartet. Die Variante ist auch möglich, wenn nur das Basis-Betriebssystem geladen ist, in dem auf die SD-Karte nicht zugegriffen werden kann.
2. Die Anwendung ist auf der SD-Karte gespeichert und wird über den Dateinamen aufgerufen, wobei die Endung „HEX“ vorausgesetzt ist. Der Vorteil bei dieser Variante ist, dass über einen globalen String Parameter übergeben werden können. Falls sich die Anwendung nicht auf der SD-Karte befindet, kann sie vorher über die USB-Schnittstelle vom PC geladen und auf der SD-Karte gespeichert werden.

2.5. Programm-Module

Um den Speicherbedarf des Betriebssystems im RAM möglichst gering zu halten, werden einzelne Programm-Module auf die SD-Karte ausgelagert. Damit diese Module unabhängig vom aktuellen Pfad verwendet werden können, werden sie als Intel-Hex-Datei im Betriebssystemverzeichnis „OS“ gespeichert. Der Aufruf erfolgt wie bei Anwendungen über den jeweiligen Dateinamen, wobei die Endung .HEX vorausgesetzt wird. Grundsätzlich werden die Programm-Module genauso wie Anwendungen erstellt. Damit es jedoch keinen Speicherkonflikt mit den Anwendungen gibt, wird ein anderer Speicherbereich verwendet. Die Startadresse hat hier den Wert 0x7000, wobei die einzelnen Module nicht mehr als 4 KB Speicher benötigen dürfen, da bei Adresse 0x8000 der Speicherbereich der Anwendungen beginnt. Es gibt insgesamt sieben ausgelagerte Programm-Module:

HELP.HEX:	Ausgabe unterstützter Befehle mit Erläuterung
LHX.HEX:	Laden einen Hex-Datei vom PC in den RAM
RUN.HEX:	Ausführen von Programmcode an einer speziellen Adresse
MEM.HEX:	Ausgabe des RAM-Speichers bei einer bestimmten Adresse
EDT.HEX:	Ändern des RAM-Speichers bei einer bestimmten Adresse
PTR.HEX:	Direktes Lesen eines Ports
PTW.HEX:	Direktes Schreiben eines Ports

2.6. Schreiben auf die SD-Karte

Im Betriebssystem sind bereits Routinen zum Schreiben von Dateien auf die SD-Karte implementiert, die beim Laden eines Programms auf die SD-Karte mit dem Befehl „LOAD“ verwendet werden. Diese Routinen können auch von Anwendungsprogrammen verwendet werden. Dabei wird zuerst eine bereits vorhandene oder neu erstellte Datei geöffnet. Anschließend kann die Datei immer blockweise zwischen einem und 512 Bytes verlängert werden. Die bereits vorhandenen Bytes können dabei nicht überschrieben werden. Die neuen Bytes werden jeweils am Ende der Datei angehängt. Nach dem Schreibvorgang muss die Datei wieder geschlossen werden.

3. Bootvorgang

3.1. Laden des Basisbetriebssystems

Beim Einschalten des Emulationsboard muss das System zunächst gebootet werden. Dazu befindet sich im FPGA ein kleiner ROM mit welchem die ersten 2 KB des Flashs in den untersten Bereich des RAMs kopiert werden (Adresse 0x0000 bis 0x07FF). Darin ist das komplette BIOS enthalten. Anschließend wird ein Sprung an Adresse 0x0780 im RAM durchgeführt. Der hier folgende Code lädt das restliche Programm aus dem Flash, das Basis-Betriebssystem. Dies ist in *Abbildung 3.1* dargestellt. Für das Basis-Betriebssystem ist eine maximale Größe von 28 KB vorgeschrieben. Mit einem Sprung an Adresse 0x0800 wird es gestartet.

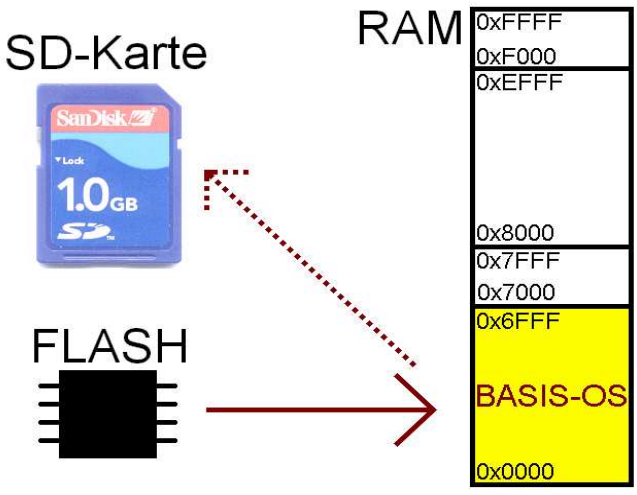


Abbildung 3.1: Laden des Basisbetriebssystems

3.2. Laden des Betriebssystems

Das Basis-Betriebssystem versucht zuerst auf die SD-Karte zuzugreifen (*Abbildung 3.1*). Wenn dies nicht möglich ist, wird der Vorgang abgebrochen und dem Anwender steht nur das Basis-Betriebssystem zur Verfügung. Gelingt der Zugriff jedoch, werden zwei Hex-Dateien aus dem Verzeichnis „OS“ in den RAM geladen, zuerst das Betriebssystem „SD.HEX“ und anschließend die Datei „LOADER.HEX“. *Abbildung 3.2* zeigt diesen Vorgang mit der entsprechenden Speichereinteilung.

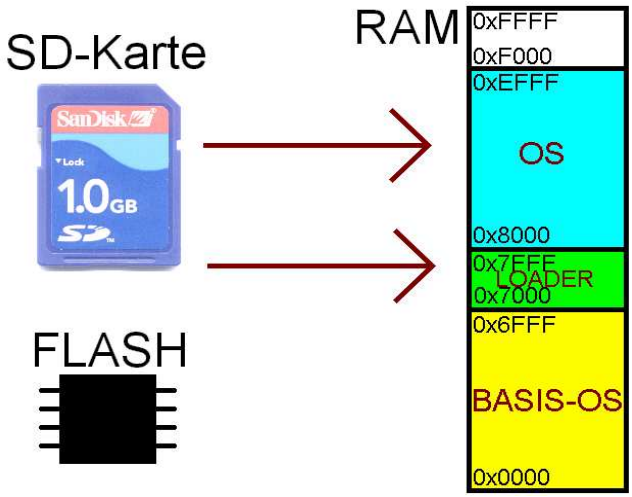


Abbildung 3.2: Laden von der SD-Karte

3.3. Funktion des Loaders

Nach dem Laden in den RAM wird der Loader ausgeführt. Der Loader ist ein eigenständiges Programm und greift nicht auf die BIOS-Routinen im unteren Speicherbereich zu. Er überschreibt das Basis-Betriebssystem mit dem Betriebssystem, das von der SD-Karte geladen wurde, wie in *Abbildung 3.3* dargestellt. Anschließend wird ein Sprung an Adresse 0x0780 durchgeführt, womit die Hardware neu initialisiert und das geladene Betriebssystem gestartet wird.

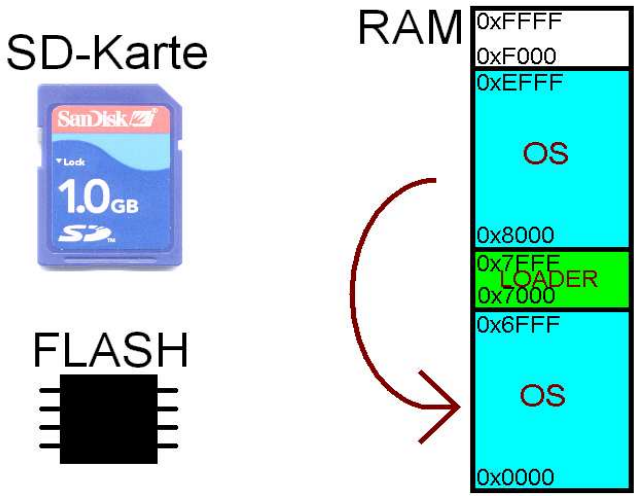


Abbildung 3.3: Überschreiben des Basis-OS

4. Ausblick

Bei dem erstellten Betriebssystem bieten sich viele Erweiterungsmöglichkeiten, die in zukünftigen Projekten implementiert werden könnten. Im Folgenden sind einige Beispiele genannt:

- CRC-Prüfung bei der Kommunikation mit der SD-Karte
- Fehlerbehandlung
- Kopieren und verschieben von Dateien und Verzeichnissen
- Ausführen von Batch-Dateien für automatische Abläufe
- Ausgabe von verschiedenen Dateitypen auf dem Terminal
- Erstellen eines Schedulers, mit dem mehrere Prozesse gleichzeitig ausgeführt werden können (Multitasking)

5. Quellen

[1] Dirk Jansen, Nidal Fawaz, Daniel Bau, Marc Durrenberger:

"A Small High Performance Microprocessor Core SIRIUS for Embedded Low Power Designs, Demonstrated in a Medial Mass Application of an Electronic Pill (ePille®)"

Embedded Systems Design Topic, Techniques and Trend, p. 363-372, June 2007, California, USA

ISBN 978-0-387-72257-3

[2] Microsoft FAT32 File System Specification Version 1.03

http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx

Design eines VGA-Testbildgenerators in VHDL

Julian Arnold, Marc Reschke
Hochschule Weingarten, Doggenriedstraße, 88250 Weingarten

Diese Ausarbeitung entstand für unser Schaltungsentwurf Praktikum an der Hochschule Weingarten im Zeitraum vom Oktober bis Dezember 2007. Dieses Praktikum findet im vierten Semester unseres Bachelor Studiengangs Elektrotechnik und Informationstechnik statt. Die Aufgabenstellung umfasste ein Testbild gemäß der VGA-Spezifikationen in einen FPGA zu programmieren, um damit einen beliebigen Bildschirm anzusteuern.

1. Einleitung

Das Praktikum fand im Laboratorium für die Entwicklung integrierter Schaltkreise der Hochschule Weingarten statt. Zur Entwicklung des VHDL-Codes stand uns ein auf Solaris basierendes Rechnersystem mit der Synthese Software „Precision“ zur Verfügung. Das genutzte Testboard (Abbildung 1) besitzt einen Xilinx Spartan 2 FPGA, einen Parallelport zum programmieren und steuern des FPGAs sowie einen VGA-Anschluss.

Im folgenden wird ein kurzer Überblick über den VGA Standard gegeben. Dann werden wir genauer auf unsere Realisierung und bei der Realisierung aufgetretene Probleme eingehen.

2. Aufgabenbeschreibung

Es ist ein Code zu entwerfen, der in der Lage ist, einen stabilen Bildrahmen zu erzeugen.

Sobald das geschafft war. ging es daran den restlichen Code zu entwickeln. Dieser enthält:

- 16 verschiedene Farbkombinationen
- 4 verschiedene Ausgabemodi
- Textfenster mit Betriebsmodi
- Ändern der Textfarbe
- 2 verschiedene Auflösungen
- Zähler, die für die Farbwechsel und für die Funktion der einzelnen Betriebsmodi notwendig sind.

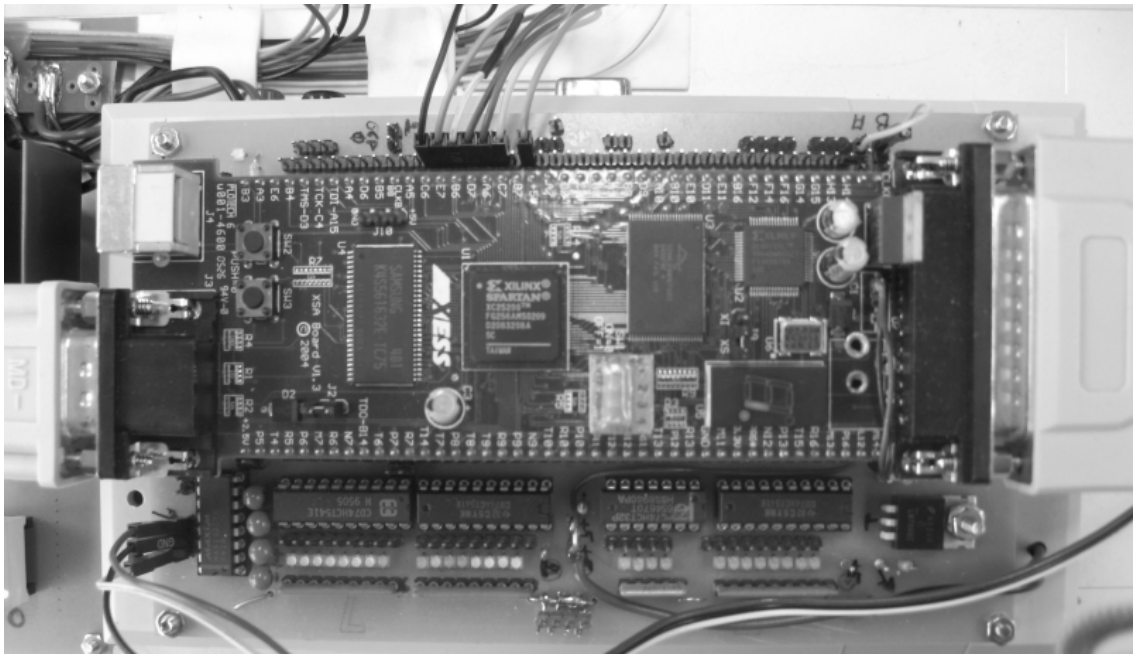


Abbildung 1: Testboard mit Spartan 2 FPGA

2.1. VGA Standard

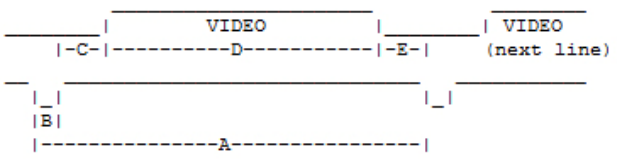
Ein VGA-Stecker enthält primär die drei Leitungen für die RGB-Farben, eine Leitung für die Horizontale Synchronisation („hsync“) und eine Leitung für die Vertikale Synchronisation („vsync“).

Das sichtbare Bild wird in einen Rahmen eingebaut, der umgeben ist von „nicht sichtbaren Pixeln“ und der von einem „hsync“ bzw. „vsync“ Impuls abgeschlossen wird. (Abbildung 2)

Bei der VGA Auflösung 640x480 besteht eine Zeile aus insgesamt 800 Pixel, davon sind jedoch nur 640 sichtbar, 96 sind für den „hsync“ und die restlichen 64 Pixel bilden den Rand.

Horizonal Timing

Horizonal Dots	640	
Vertical Scan Lines	480	
Horiz. Sync Polarity	NEG	
A (us)	31.77	Scanline time
B (us)	3.77	Sync pulse lenght
C (us)	1.89	Back porch
D (us)	25.17	Active video time
E (us)	0.94	Front porch



Vertical Timing

Horizonal Dots	640	
Vertical Scan Lines	480	
Vert. Sync Polarity	NEG	
Vertical Frequency	60Hz	
O (ms)	16.68	Total frame time
P (ms)	0.06	Sync length
Q (ms)	1.02	Back porch
R (ms)	15.25	Active video time
S (ms)	0.35	Front porch

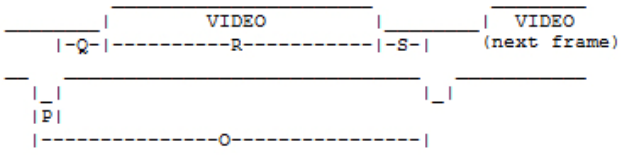


Abbildung 2: VGA Daten für 640x480

Im Vertikalen besteht das Bild aus 525 Zeilen. Davon sind nur 480 Zeilen sichtbar 2 Zeilen lang ist der „vsync“ und die restlichen 43 Zeilen bilden den „nicht sichtbaren“ Rand.

Um bei dieser Auflösung 60Hz Bildwiederholfrequenz zu erreichen muss der Pixeltakt bei 25,175 MHz liegen. Dies ergibt eine Zeilenfrequenz von 31469 Hz und folglich die gewünschten 60Hz Bildwiederholfrequenz.

3. Realisierung

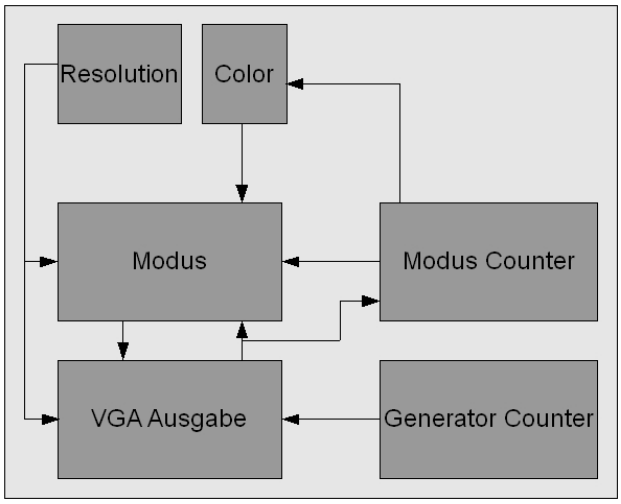


Abbildung 3: Beziehungen zwischen den Prozessen

Unser Programm besteht aus zwei Teilen: Einem Bereich für die Bereitstellung der VGA-Synchronisation und einem Bereich der die Testbilder erstellt. (Abbildung 3)

Für die Erstellung des VGA-Signals sind die Programm-Routinen „VGA-Ausgabe“, „Generator Counter“ und „Resolution“ zuständig.

Die Bereitstellung der vier verschiedenen Betriebsmodi bewirken die restlichen drei Prozesse sowie der Prozess „Resolution“.

4. Programmübersicht

Unser Programm bietet vier verschiedene Ausgabemodi, die durch ändern von Eingangsbits gewechselt werden können. Diese Auswahlbits steuert der Parallelport des PC's (Abbildung 3)

Modus 0: Rechteck
Hier gibt es ein farbiges Rechteck, welches von drei anders gefärbten Rechtecken eingeschlossen wird.

Modus 1: Streifen
In diesem Modus werden alle 80 Pixel acht vertikale Streifen dargestellt, die immer abwechselnd gefärbt sind. Dieser Modus eignet sich besonders gut zum Testen der Konvergenzeigenschaften eines Monitors.

Modus 2: Quadranten
Hierbei wird jede Ecke des Bildschirms in einer anderen Farbe dargestellt. Die einzelnen Quadranten sind durch eine Schwarze Linie getrennt.

Modus 3: Pac-Man
Dieser zeigt einen Pac-Man, welcher von links nach rechts über den Bildschirm läuft und dabei ein Männchen verfolgt.

Es ist zusätzlich immer möglich, über das Setzen des Eingangs „text" ein kleines Textfenster einzublenden, das den aktuellen Modus anzeigt.

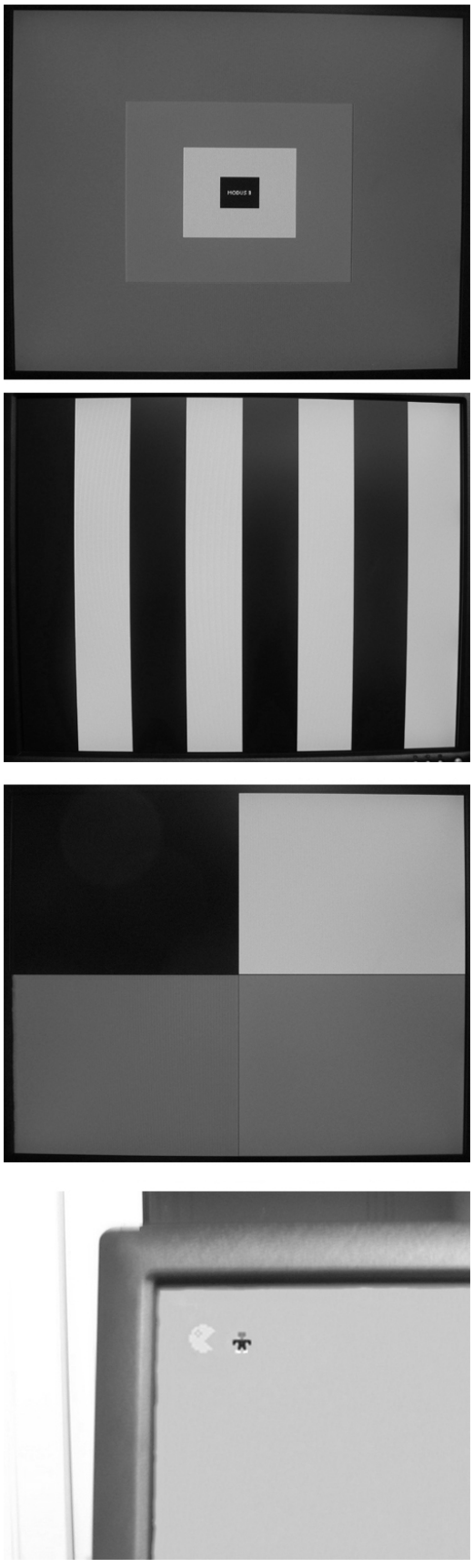


Abbildung 3 Die vier verschiedenen Darstellungsmodi in der Übersicht

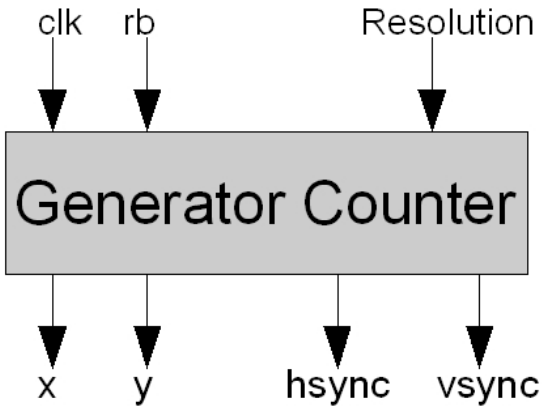
5. Die VHDL-Prozesse

5.1 VGA Signal Generierung

5.1.1 Der Prozess: „Resolution"

Der Prozess „Resolution" hat einen externen Eingang, der die gewünschte Auflösung festlegt. Bei einer „0" am Eingang wird die Auflösung 640x480 gewählt, während bei einer „1" die Auflösung 800x600 gewählt wird.

5.1.2 Der Prozess: „Generator Counter“

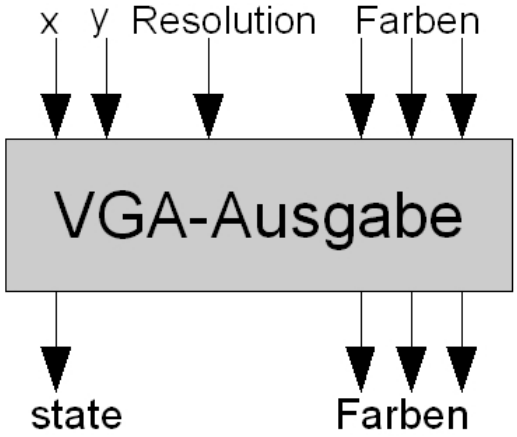


Dieser Prozess besteht aus zwei Zählern, die abhängig von der gewählten Auflösung auf die maximale Anzahl Pixel pro Reihe bzw. Zeilen pro Bild zählen. Also z.B. bei der Auflösung 640x480 auf 800 Pixel und 525 Reihen. Zusätzlich hat der Prozess die Aufgabe, den „hsync“ und „vsync“ zu steuern. Auch dies geschieht in Abhängigkeit der gewählten Auflösung.

Die folgende Erklärung bezieht sich auf die Auflösung 640x480:

Der Zähler setzt bei jedem Takt den „Pixel Zähler“ um eins herauf. Solange der Pixelzähler kleiner wie 96 ist, wird der „hsync“ auf ‚0’ gehalten (low aktiv). Sobald der Zähler 96 erreicht hat, wird „hsync“ auf eins gesetzt und bleibt auf eins, bis das Ende der Reihe erreicht ist (800 Pixel). Dann wird der Zähler wieder auf eins gesetzt und der „Spaltenzähler“ um eins erhöht. Solange der „Spaltenzähler“ unter 523 Spalten steht, ist „vsync“ auf ‚1’(low aktiv). Bei 524 und 525 wird „vsync“ ‚0’. Wenn sowohl der „Pixelzähler“ als auch der „Spaltenzähler“ ihre Maximalwerte erreicht haben werden beide wieder auf eins gesetzt und ein neues Bild wird begonnen.

5.1.3 Der Prozess: „VGA Ausgabe“



In diesem Prozess wird der sichtbare Bereich des Bildes definiert. Die Bilder, die in „Modus“ entstehen, werden ausgegeben und über das Bit „state“ die beiden Prozesse „Modus“ und „Modus Counter“ gesteuert.

Anhand der beiden Zähler aus dem Prozess „Generator Counter“ wird das Bild im nicht sichtbaren Bereich und während der „hsync-“ und „vsync-“ Zeiten auf „schwarz“ geschaltet. Während die beiden Zähler sich im sichtbaren Bereich befinden, wird das Bit „state“ auf ‚1’ geschaltet und zeigt damit den anderen Prozessen, daß sie ein Bild ausgeben können. Die Farbeingänge des Prozesses sind dann an seine Ausgänge durchgeschaltet. Sobald der sichtbare Bereich des Bildes verlassen wird, wechselt das Bit „state“ auf ‚0’, d.h. auf Dunkel.

5.2 Bild Generierung

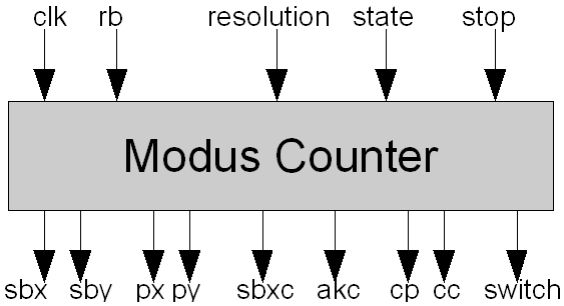
5.2.1 Der Prozess: „Color“

Dieser Prozess hat zwei Aufgaben:

1. Festlegen der Textfarbe
2. Farbwechsel der anderen Modi.

Das wechseln der Textfarbe wird direkt durch einen externen Schalter erledigt: bei einer ‚1' wird der Text Grün, bei einer ‚0' Gelb.
Für das Farbwechseln der vier Betriebsmodi wird eine Integerzahl verwendet. Diese wird im Prozess „Modus Counter" hoch gezählt. Mit jedem Zählerschritt entsteht eine andere aus vier Farben bestehende Kombination.

5.2.2 Der Prozess: „Modus Counter"



Der Prozess „Modus Counter" steuert die verschiedenen Ausgabemodi sowie den Farbwechsel. Dieser Prozess arbeitet nur, wenn das BIT „state" = ‚1' ist.

Die folgende Erklärung bezieht sich auf die Auflösung 640x480:

Der Zähler „akc" wird einmal pro Sekunde hoch gezählt. Dafür haben wir den Zähler „cc" eingeführt. Er wird bei jedem „vsync" um eins erhöht. Sobald er 60 erreicht, wird er wieder auf ‚0' gesetzt und „akc" inkrementiert. Wenn „akc" bei 16 ankommt, fängt er wieder bei ‚0' an.

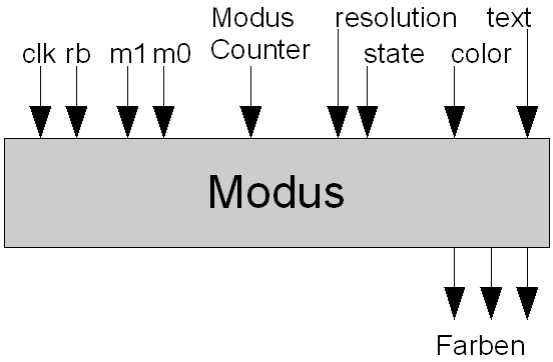
Es gibt zwei Zähler („sbx" und „sby") die das sichtbare Bild aufspannen. Das bedeutet: sie zählen von eins bis 640 bzw. bis 480.

Wenn der Zähler „cp" 20 erreicht hat, wird der Zähler „px" um 20 erhöht und cp wieder auf null gesetzt. Damit erreichen wir die gewünschte Geschwindigkeit, mit der sich die Figuren im Modus Pac-Man bewegen. Auch „cp" wird bei jedem vsync um eins erhöht.

Die beiden Zähler „px" und „py" geben die aktuelle Position der beiden Figuren im Modus „Pac-Man" an. Sobald „px" über 640+30 liegt, wird „px" auf Null gesetzt und „py" um 40 erhöht.
Der Zähler px zählt bis 640+30, damit die Figuren nach einem Durchlauf nicht einfach auf dem Bildschirm „erscheinen", sondern von links wieder in das Bild „hineinlaufen".

Der Zähler „sbxc" wird ausschließlich für den Streifenmodus verwendet. Er zählt bei jedem Takt um eins hinauf und zwar bis 80.
Dann wird er wieder auf eins gesetzt.

5.2.3 Der Prozess: „Modus"



Anhand der beiden Eingänge m0 und m1 entscheidet der Prozess „Modus" welcher Ausgabemodi verwendet wird.

Die folgende Erklärung bezieht sich auf die Auflösung 640x480:

Im Modus 0 entsteht ein um die Variablen „px" und „py" verschobener Pac-Man, der abhängig vom Zustand der Variable „switch" den Mund offen oder geschlossen hat.

5. Ausblick

Um zu sehen, was mit dem Spartan 2 FPGA alles realisierbar ist, haben wir uns dafür entschieden, noch zusätzlich den Modus "Pac-Man" einzuführen. (Abbildung 5)

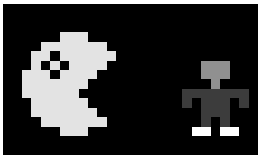


Abbildung 5 Pac Man

Dies brachte den FPGA aber an seine Grenzen, die Gatterauslastung lag bei über 100%. Diese Auslastung ermöglichte keine Synthetisierung. Wir mussten daher viele Ideen die wir hatten leider aufgeben.

So sollte z.B. der Pac-Man von links nach rechts sowie zurück laufen. Dies war jedoch mit dem recht kleinen Spartan 2 nicht umsetzbar.

Doch trotz dieser Einschnitte und einigen Optimierungen kann der Code nur mit einigen Warnungen synthetisiert werden.

```
#      SLICE sbx(9)
#      SLICE sbx(11)
#      SLICE sbx(13)
#      SLICE sbx(15)
#      SLICE sbx(17)
#      SLICE sbx(19)
#      SLICE sbx(21)
#      SLICE sbx(23)
#      SLICE sbx(25)
#      SLICE sbx(27)
#      SLICE sbx(29)
#      SLICE sbx(31)
#
#
# This situation can be resolved by fixing the follo
#
#
# The logic does not fit onto the chip in this form.
```

Durch die fehlerhafte Synthese treten Timingprobleme auf, die dazu führen das die gesamte Bildausgabe um einen Pixel nach rechts verschoben wird.

Mit einem größeren FPGA als Zielsystem sollten diese Probleme jedoch verschwinden.

6. Danksagungen

Wir möchten uns an dieser Stelle nochmals bei Prof. Dr.-Ing. Walter Ludescher und Prof. Dr.-Ing. Andreas Siggelkow für ihre fachlichen Ratschläge und hilfreichen Anregungen bedanken. Zusätzlich danken wir der MultiProjektChip Gruppe Baden-Württemberg, dass sie es uns ermöglichte, im EIS-Laboratorium an diesem Projekt zu arbeiten.

7. Literaturangaben

[1] http://www.epanorama.net/documents/pc/vga_timing.html

Hardware-Software-Codesign in der Radarverarbeitung

Thomas Mahr
EADS Defence & Security
Radar Signal Processing
Wörthstr. 85, D-89077 Ulm

Ralf Gessler
Hochschule Heilbronn,
Fakultät Technik und Wirtschaft
Daimlerstr. 35, D-74653 Künzelsau

Zusammenfassung

In der Radarverarbeitung erfordert die Detektion, Klassifikation und Verfolgung von Zielen unter Echtzeitbedingungen den Einsatz einer den jeweiligen Verarbeitungs- und Steuerungsalgorithmen angepassten Rechnerarchitektur. Ein hybrides Systems, bestehend aus Mikroprozessoren und FPGAs, bietet die Vorzüge beider Rechnertechnologien: hohe Ausführungsgeschwindigkeit durch parallele Pipeline-Verarbeitung und DSP-Funktionsblöcke, taktgenaue Verarbeitung, bequeme Hochsprachen-Programmierung und leistungsfähige Entwicklungshilfsmittel. Die Entwicklung von hybriden System leidet jedoch unter der traditionellen Trennung der Disziplinen „Software-Entwicklung für Mikroprozessoren" und „Digitaler Schaltungsentwurf für FPGAs". In der Radarprozessorentwicklung wollen wir diese Trennung überwinden und den Radarprozessor als Ganzes betrachten, ohne zu früh zwischen einzelnen Baugruppen, zwischen Hardware- und Software-Komponenten und zwischen Mikroprozessoren und FPGAs unterscheiden zu müssen. Hierzu bedienen wir uns eines zentralen UML-basierten Modells, das uns bei der Analyse und dem Architekturentwurf unterstützt und zur automatischen Generierung von Code und Dokumenten dient. Diese Vorgehensweise erlaubt ein hohes Maß an Flexibilität und ermöglicht eine Steigerung der Produktivität bei gleichzeitig wachsender Qualität.

1 Software-Entwicklung und Schaltungsentwurf

Software-Entwicklung für Mikroprozessoren und Schaltungsentwurf für FPGAs! Zwei Technologien zur Programmierung der beiden wichtigsten Rechenmaschinen haben sich in der Vergangenheit weitgehend unabhängig voneinander entwickelt und werden jeweils von zwei verschiedenen Anwendergruppen genutzt: den Informatikern, Software-Entwicklern und Programmierern einerseits und den Elektrotechnikern und Hardware-Entwicklern andererseits. Viele Hochschulen lehren beide Disziplinen getrennt – in unterschiedlichen Studiengängen, in verschiedenen Vorlesungen. Oft setzen Unternehmen entweder ganz auf die eine oder ganz auf die andere Technologie oder weisen beide Technologien jeweils unterschiedlichen, spezialisierten Abteilungen zu. Während viele Computermagazine und Software-Bücher nur den Mikroprozessor zu kennen scheinen, sucht man in der Literatur, die sich der FPGA-Thematik widmet, vergeblich nach modernen Software-Konzepten, die der Lösung von komplexen Aufgaben auf einer angemessenen Abstraktionsstufe dienen.

Dies mag überraschen! Software-Entwicklung *und* Schaltungsentwurf zielen darauf ab, eine vorgegebene Aufgabe zu lösen: Algorithmen zu finden, in einer Programmiersprache zu beschreiben und sie auf einer Hardware auszuführen. In dem einen Fall ist die Programmiersprache z.B. C++ und im anderen Fall VHDL. Ein wichtiger Anteil des Entwicklungsprozesses ist weitgehend unabhängig davon, ob die Aufgabe durch Mikroprozessoren oder FPGAs gelöst wird: die Analyse und Verwaltung der Anforderungen, der Entwurf der Software-Architektur, das Testen der Lösung und das der Entwicklung zugrunde liegende Vorgehensmodell. Auch der Herstellungsprozess der beiden Rechenmaschinen ähnelt sich. Mikroprozessoren und FPGAs werden beide als integrierte Schaltungen auf Silizium-Halbleiterbasis hergestellt.

Die Teilung der Anwender in zwei Gruppen liegt eher an der unterschiedlichen Entwicklung der beiden Technologien während der letzten Jahrzehnte. Die Software-Pioniere programmierten Mitte des 20. Jahrhunderts Mikroprozessoren in einer Maschinensprache und schufen damit – ohne die Rechenmaschine (auf atomarer Ebene) physikalisch zu verändern – ein nicht gegenständliches Gut: die Software. Anfangs war die Software noch stark von der Hardware abhängig (Maschinensprache!), im Laufe der Zeit entstanden aber immer neue Verfahren, um die Abhängigkeit der Software von der Hardware zu reduzieren: Assemblersprache, höhere Programmiersprachen wie C, virtuelle Java-Maschine oder modellgetriebene automatische Codegenerierung.

Der Schaltungsentwurf ist aus der Hardware-Entwicklung, der Entwicklung von materiellen Bausteinen entstanden. Algorithmen, Berechnungsvorschriften zur Lösung einer Aufgabe, werden in einer digitalen Schaltung abgebildet. Anfangs wurden die Schaltungen dem Baustein noch fest aufgeprägt, indem Leiterbahnen auf atomarer Ebene angelegt wurden – ein praktisch irreversibler Prozess. Wollte man die Berechnungsvorschriften ändern, musste man einen neuen Baustein herstellen. Mit dem Aufkommen der programmierbaren Logikbausteine hat sich dies geändert. Die Funktion der Rechenmaschine wurde jetzt nicht mehr über die Anordnung der Atome im Baustein definiert, sondern durch die Konfiguration der Elektronen. Und die ist leicht zu ändern! Auf ein und demselben Baustein können nacheinander viele verschiedene Funktionen programmiert werden. Der Fokus hat sich vom materiellen Prozess der Herstellung eines Stücks Hardware auf den nicht materiellen Prozess des Entwurfs einer Schaltung verschoben.

Um eine Schaltung zu testen, muss nun kein Hardware-Baustein mehr angefertigt werden – ein relativ langwieriger und teurer Prozess! Die Schaltung braucht jetzt für den Test nur auf ein bereits vorhandenes FPGA geladen zu werden. Das verkürzt die Entwicklungszyklen drastisch und erlaubt es, sich auf diejenigen Aktivitäten zu konzentrieren, die eine Entwicklung komplexer Systeme ermöglichen: Anforderungsanalyse, Modellieren und Testen! Diese Aktivitäten wurden in den letzten Jahrzehnten auf breiter Ebene untersucht. Getrieben von der Software-Entwicklung für Mikroprozessoren hängen sie aber streng genommen nicht von der Rechenmaschine ab, auf dem die Software schließlich ausgeführt wird! Warum sollten also diese bewährten Methoden nicht auch für den Schaltungsentwurf nützlich sein?

Nicht nur die Vertreter des historischen „Hardware-Lagers“ können von diesem Umstand profitieren. Auch die Software-Entwickler, die bisher mehr oder weniger bewusst Mikroprozessoren als Zielplattform vor Augen hatten, gewinnen jetzt eine alternative Rechenmaschine, die für bestimmte Anwendungsklassen dem Mikroprozessor überlegen ist:

- parallelisierbare Anwendungen: Unter Umständen lässt sich ein parallelisierbarer Algorithmus kostengünstiger auf einem FPGA ausführen als auf einem Mikroprozessor-Cluster.
- signalverarbeitungsintensive Anwendungen: Auf einem FPGA integrierte DSP-Funktionsblöcke bieten z.B. schnelle MAC-Einheiten zur Multiplikation und Akkumulation an, die unter anderem für Vektoroperationen genutzt werden können.
- nicht sättigbare Anwendungen: Im Gegensatz zu einem Mikroprozessor, dessen Verarbeitung oft durch externe Ereignisse gesteuert wird (Interrupts), verarbeitet ein FPGA die Daten in jedem Takt gleich.

Das bedeutet nicht, die *gesamte* Software soll auf FPGAs ausgeführt werden! Serielle Algorithmen laufen nach wie vor besser auf einem seriell arbeitenden Mikroprozessor, der in der Regel höher getaktet ist als ein technologisch vergleichbares FPGA, und es werden nur die Software-Teile auf FPGAs ausgelagert, für die sich der höhere Implementierungsaufwand lohnt.

2 Begriffe: Hardware, Software und Codesign

Die Trennung der beiden Disziplinen Software-Entwicklung für Mikroprozessoren und Schaltungsentwurf für FPGAs führte bei den Anwendern der Technologien zu teilweise unterschiedlichen Verwendungen der Begriffe Hardware und Software – damit zu einem potentiellen Verständigungsproblem! Wir verstehen darunter folgendes:

Hardware sind alle materiellen Anteile eines Computers, z.B. Prozessor und Speicher.

Software sind alle nicht-materiellen Anteile eines Computers: Programme und zugehörige Daten[1]

Demnach ist ein Hardware-Entwickler also jemand, der im wörtlichen Sinn Materie bewegt und anordnet, also jemand, der

- eine digitale Schaltung (ASIC, FPGA, Mikroprozessor, ...) auf atomarer Ebene herstellt und dabei Siliziumatome anordnet oder
- einzelne Hardwarebausteine (Mikroprozessoren, FPGAs, I/O-Karten, Netzteile, Gehäuse, ...) zu einem System zusammenstellt.

[1] in Anlehnung an John W. Tuckey, der 1957 das Kunstwort *Soft*ware als Ergänzung zu *Hard*ware eingeführt hat

Ein Entwickler, der eine digitale Schaltung für ein FPGA entwirft und das FPGA konfiguriert, entwickelt nach unserer Definition dagegen keine Hardware sondern Software – genau wie ein C-Programmierer.

Folgt man dieser Sprechweise, so kann man, je nach Standpunkt, unter Hardware-Software-Codesign folgendes verstehen:

1. "Chip-Entwickler"-Perspektive: Gemeinsamer Entwurf von Chips (Hardware) und Mikrocode und Compiler (Software)
2. "Eingebettetes System"-Perspektive: Gemeinsamer Entwurf einer Konfiguration aus verfügbaren Hardware-Komponenten und der Entwicklung der Applikations-Software (digitale Schaltungen und Mikroprozessorprogramme).

Die letztere Sichtweise trifft für unsere Entwicklung von Radarprozessoren zu.

Um die Idee des Hardware-Software-Codesigns zu verfolgen ist es unserer Erfahrung nach notwendig, sich auf eine gemeinsame, technologiedomänenübergreifende Nomenklatur zu verständigen und von der herkömmlichen technologiespezifischen Nomenklatur zu lösen („etwas in Hardware realisieren" sagen und FPGA-Programmierung meinen, „etwas in Software realisieren" sagen und Mikroprozessorprogrammierung meinen). Dies hilft die Gemeinsamkeiten zwischen den beiden Disziplinen Software-Entwicklung für Mikroprozessoren und Schaltungsentwurf für FPGAs herauszustellen und freie Sicht auf die eigentliche Entwicklungsaufgabe zu schaffen: ein System bestehend aus Hardware- und Software-Komponenten zu entwickeln, das die Anforderungen des Anwenders erfüllt [1].

3 Radarverarbeitung auf hybriden Plattformen

Ein Radarprozessor digitalisiert von einem Radarsensor empfangene Echos, unterdrückt Störungen, detektiert und klassifiziert Ziele, verfolgt die Ziele und liefert die gewonnenen Informationen an ein Sichtgerät oder andere Verarbeitungskomponenten. Das Unterdrücken von störenden Echos und gleichzeitige Verstärken von Zielechos ist ein nichtlineares Optimierungsproblem, das rechenintensive Algorithmen erfordert [2, 3, 4]. Gut parallelisierbare Verarbeitungsalgorithmen und taktgenaue Steuerungsaufgaben werden daher auf parallel arbeitenden FPGAs ausgeführt. Weniger gut parallelisierbare Algorithmen, sich noch in der Erprobung befindende Algorithmen und Nichtechtzeitsteuerungsaufgaben werden in einer Hochsprache wie C++ oder Java formuliert und auf einem Mikroprozessor ausgeführt.

Ein Beispiel für ein hybrides System aus Mikroprozessoren und FPGAs zeigt Abbildung 1. FPGA 1 empfängt über eine schnelle Verbindung Signale von einem AD-Wandler, filtert diese über einen FIR-Filter und sendet das Ergebnis an Mikroprozessor MP 1. MP 1 verarbeitet die Signale in Abhängigkeit von Benutzerkommandos, die über eine XML-Datei aus dem Intranet empfangen werden. Das Parsen der XML-Datei und die Steuerung der Signalverarbeitung geschehen bequem über ein Java-Programm. Falls der Benutzer eine zweidimensionale Spektralanalyse in Echtzeit wünscht, werden die Signale an FPGA 2 geschickt, dort spektral untersucht und die Ergebnisse zurück an MP 1 gesandt. FPGA 2 wirkt dabei als ein Koprozessor von MP 1. Von MP 1 laufen die Daten zu MP 2, um dort auf ein Anzeige- und Steuerprogramm mit hardwarebeschleunigter OpenGL 3D-Darstellung visualisiert zu werden. Außerdem werden ausgewählte Daten sowohl als Zip-Datei auf Festplatte komprimiert als auch als Java-Objekte von MP 2 an MP 3 weitergereicht. MP 2 ist ein auf FPGA 3 realisierter Prozessor-soft-core (SOC). Die Java-Objekte werden auf MP 3 in einen Bitstrom umgewandelt und außerhalb des soft-core, aber auf demselben Chip, über eine in VHDL programmierte digitale Schaltung weiterverarbeitet und über eine schnelle Verbindung an FPGA 4 geschickt.

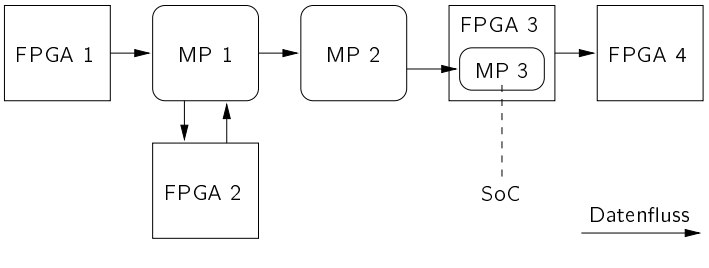


Abbildung 1: Beispiel für ein hybrides System aus FPGAs und Mikroprozessoren (MPs).

Dieses hybride System nutzt die kombinierten Stärken von Mikroprozessoren und FPGAs:

- schnelle, taktgenaue Datenübertragung
- schnelle parallelisierbare Signalverarbeitungsalgorithmen
- leistungsfähige Software-Bibliotheken zum Lesen von XML-Daten und Komprimieren im Zip-Format
- grafische Benutzerschnittstelle mit OpenGL 3D-Darstellung
- Objekte und Java-RMI zur Übertragung von Java-Objekten über Prozessorgrenzen hinweg

Abbildung 2 stellt drei mögliche Wechselwirkungen zwischen einem Mikroprozessor MP und einem FPGA dar:

1. Abbildung 2(a): Der MP liefert Daten an das FPGA. Beispiel: Empfang von einer schnellen Schnittstelle.
2. Abbildung 2(b): Das FPGA liefert Daten an den MP. Beispiel: Senden an eine schnelle Schnittstelle.
3. Abbildung 2(c): Koprozessor
   (a) Das FPGA dient als numerischer Koprozessor eines MP.
   (b) Der MP dient als (Hochsprachen-)Koprozessor eines FPGAs, z.B. zum Lesen oder Schreiben einer XML-Datei.

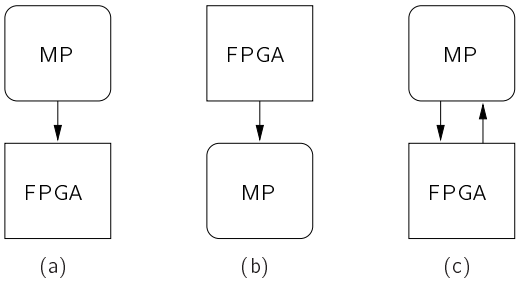


Abbildung 2: Drei Wechselwirkungen zwischen Mikroprozessor MP und FPGA.

4 Modellbasierte Radarprozessor-Entwicklung

Die Abbildung 3 illustriert den Fluss der Radarprozessor-Entwicklung. Diese besteht aus drei Haupt

1. **Analyse** der an den Radarprozessor gestellten Aufgabe – Ergebnis:
   - Verständnis der Zustände in denen sich der Radarprozessor befinden muss und der Übergänge zwischen diesen Zuständen
   - Verständnis der in den jeweiligen Zuständen zu erbringenden Funktionalitäten
2. **Entwurf** der Architektur des Radarprozessors – Ergebnis:
   - Verfeinerung der Funktionalitäten
   - Zerlegung des Radarprozessors in Hardware- und Software-Komponenten
   - Zuweisung der Funktionalitäten zu Software-Komponenten
   - Zuweisung der Software-Komponenten zu Hardware-Komponenten
3. **Realisierung** des Radarprozessors – Ergebnis:
   - Entwicklung der Software-Komponenten
   - Integration der Hardware-Komponenten zu einer Hardware-Plattform
   - Integration der Software- und Hardware-Komponenten zu einem Radarprozessor
   - Test des Radarprozessors

Der Rückkoppelungspfeil in der Abbildung weist auf eine iterativ-inkrementelle Vorgehensweise hin. Diese ist notwendig, da bei unserer Entwicklung von Radarprozessoren die Anforderungen und Randbedingungen weder vollständig zu Beginn des Projekts erfasst werden können noch während der gesamten Projektlaufzeit stabil gehalten werden können [5]. Dieser Umstand verlangt eine Entwicklungsmethodik, die es ermöglicht möglichst flexibel auf Änderungen, Erfahrungen während der Erprobung und Rückmeldungen der Anwender zu reagieren.

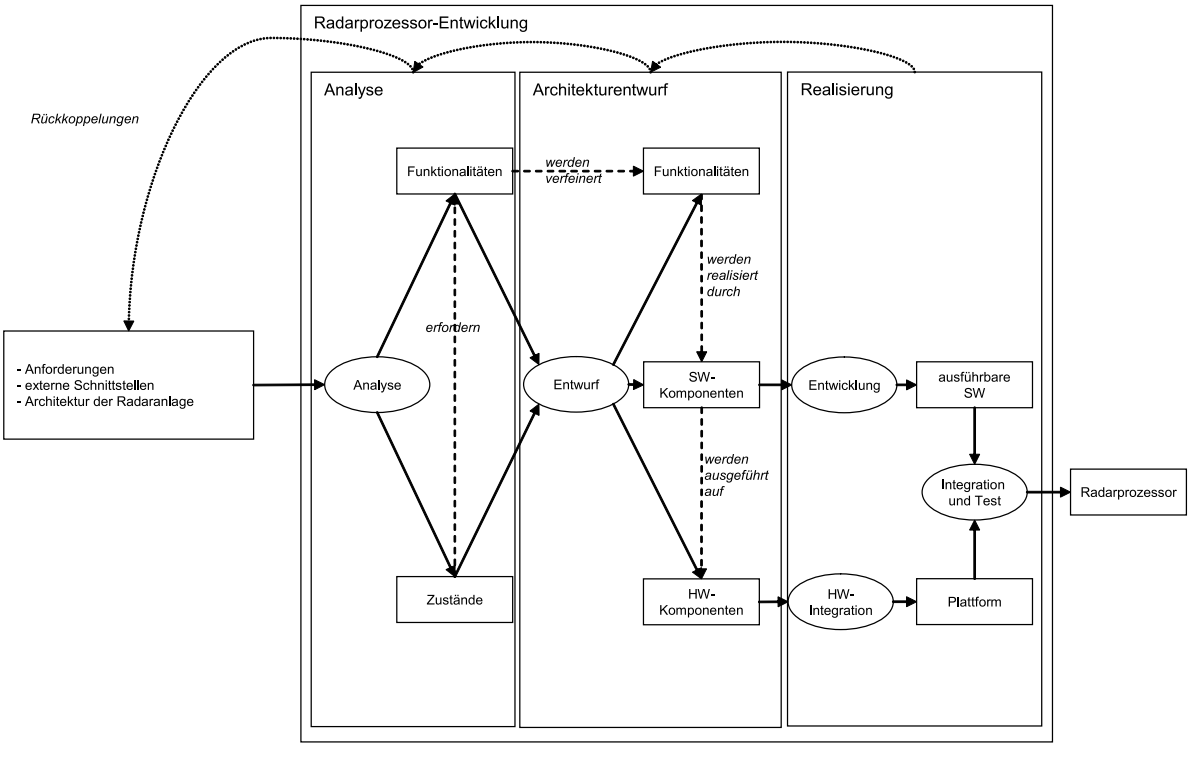


Abbildung 3: (Iterativ-inkrementeller) Entwicklungsfluss des Radarprozessors.

Ein zentrales Mittel dieser Entwicklungsmethodik ist ein formales Modell des Radarprozessors, das die Analyse und den Architekturentwurf unterstützt und die Ergebnisse konsistent notiert. Dieses Modell besteht aus zwei aneinandergekoppelten Teilmodellen:

1. UML-Modell[2]: Dieses UML-Modell enthält die Analyse und Architektur des Radarprozessors [6, 7, 8].
2. Tabellenmodell[3].

Das an das UML-Modell angekoppelte Tabellenmodell ergänzt das UML-Modell, denn mit UML ist es nur schwer möglich Parameter eines technischen Systems auf übersichtliche Weise untereinander in Beziehung zu setzen und zu verwalten. Beispiel: Ein Analog-Digital-Wandler tastet ein Signal mit einer Auflösung von 12 Bit ab. Die Abtastwerte s werden anschließend mit einem 4-Bit Faktor f multipliziert. Das Ergebnis e ist maximal 16 Bit breit. Ändert man die Bitbreite des Faktors von 4 auf 6, so wirkt sich das direkt auf die Bitbreite des Ergebnisses aus. Diese erhöht sich von 16 auf 18 Bit. Mit einem Tabellenkalkulationsprogramm kann man diese Abhängigkeit der Bitbreiten sehr leicht modellieren – mit einem gängigen UML-Werkzeug ist das nur über Umwege möglich. In einer Tabelle lassen sich die drei Parameter s, f und e und deren gegenseitige Abhängigkeiten sehr übersichtlich darstellen. In UML wären die drei Parameter wahrscheinlich als Attribute in (verschiedenen) Klassen aufgehoben.

[2]Werkzeug: Enterprise Architect von `http://www.sparxsystems.de`
[3]Werkzeug: Excel / OpenOffice Calc `http://de.openoffice.org`

4.1 Analyse

Im Analyseschritt werden die an den Radarprozessor gestellten Anforderungen unter Berücksichtigung der Radaranlagenarchitektur, einschließlich der externen Schnittstellen zwischen dem Radarprozessor und dessen Nachbarsystemen, analysiert. Zustandsdiagramme zeigen die Zustände des Radarprozessors (z.B. "Überwachung" oder „Wartung") und die Übergänge zwischen diesen Zuständen. Je nach Zustand werden vom Radarprozessor unterschiedliche Funktionalitäten verlangt. Anwendungsfalldiagramme bieten aus einer Außensicht auf den Radarprozessor eine Übersicht über die Wechselwirkungen des Radarprozessors mit seinen Nachbarsystemen. Diese als Anwendungsfälle dargestellten allgemeinen Funktionen werden mittels Aktivitätsdiagrammen verfeinert. Diese zeigen die inneren Funktionalitäten des Radarprozessors als UML-Aktivitäten und können wiederum mittels weiterer Aktivitätsdiagramme verfeinert werden.

4.2 Architekturentwurf

Basierend auf den Analyseergebnissen werden in diesem Arbeitsschritt die Hardware- und Software-Komponenten des Radarprozessors mittels Komponentendiagrammen und die Beziehungen dieser Komponenten untereinander modelliert. Hardware-Komponenten können beispielsweise Mikroprozessoren, FPGAs, Hauptplatinen, Schnittstellenkarten oder Stromversorgungen sein, Software-Komponenten sind ausführbare Einheiten: Mikroprozessorprogramme und FPGA-Konfigurationsdateien.

Während des Entwurfs werden die im Analyseschritt gewonnenen Funktionalitäten weiter verfeinert bis auf eine Ebene von Verarbeitungs- und Steuerungsalgorithmen, für die eine algorithmische Komplexität (Anzahl Addition, Multiplikationen, trigonometrischer Operationen) und der Grad der Parallelisierbarkeit erkennbar ist. Diese Kenntnis liefert Aufschlüsse über die zu bevorzugende Hardware-Architektur, die Gruppierung der Funktionalitäten (Algorithmen) zu Software-Komponenten und die Abbildung der Software-Komponenten auf Hardware-Komponenten. Im Wesentlichen treiben also die Algorithmen und die zu bewältigenden Datenraten die Hardware-Architektur, d.h. die Auswahl der Rechenbausteine (Mikroprozessoren und FPGAs) und die Rechnertopologie (Verbindungen der Rechenbausteine untereinander). Umgekehrt beeinflusst allerdings auch die Hardware-Architektur die Auswahl der Algorithmen. Beispielsweise kann die in der Radarverarbeitung typische Funktionalität „Pulskompression" sowohl als FIR-Filter als auch über FFTs (Faltung als Multiplikation im Fourier-Raum) verfeinert werden [9]. Bei typischen Pulskompressionslängen zeigt es sich, dass auf einem FPGA ein FIR-Filter gegenüber FFTs aufgrund der höheren Geschwindigkeit zu bevorzugen ist, auf einem Mikroprozessor dagegen die Multiplikation im Fourier-Raum.

Da eine starke Koppelung zwischen Algorithmen und Rechnerarchitektur besteht, kann es keinen von der Radarprozessorsystem-Architektur losgelösten Hardware-Architekturentwurf geben, sondern nur einen gemeinsamen Entwurf der gesamten Radarprozessorsystem-Architektur, die alle Facetten umfasst: Algorithmen, Software und Hardware – also Hardware-Software-Codesign im oben genannten Sinn.

4.3 Realisierung

Im Realisierungsschritt werden die im Modell definierten Hardware-Komponenten zu einer Radarprozessor-Plattform integriert und die ausführbaren Software-Komponenten mittels der jeweils geeigneten Werkzeuge entwickelt. Beispielsweise wird eine auf einem FPGA auszuführende Software-Komponente in Simulink beschrieben und daraus eine Netzliste synthetisiert und eine auf einem Mikroprozessor auszuführenden Software-Komponenten in UML modelliert und in C++ oder Java programmiert.

Hierbei werden Teile des Codes und der Dokumententation automatisch aus dem Modell des Radarprozessors generiert:

- bitgenaue Beschreibung in HTML und vollständige C++/Java-Klassen der Radarsignale (Detektionen, Zielmeldungen, Zielspuren, ...) und Steuerkommandos,
- C++/Java-Coderahmen für Signalverarbeitungsalgorithmen und
- Unit-Testrahmen.

Der Codegenerator ist ein auf Velocity[4] basierender Nachfolger des in [10] beschriebenen Generators.

[4] The Apache Velocity Project `http://velocity.apache.org`

5 Zusammenfassung und Ausblick

Ausgehend von den Anforderungen an den Radarprozessor haben wir die Zustände und Funktionalitäten des Radarprozessors analysiert, den Radarprozessor unter Berücksichtigung der Funktionalitäten auf algorithmischer Ebene in Software- und Hardware-Komponenten zerlegt, die Funktionalitäten den Software-Komponenten zugewiesen und die Software-Komponenten den ausführenden Hardware-Komponenten zugeordnet. Aufgrund der starken Koppelung zwischen Algorithmen und Rechnerarchitektur wird die gesamte Algorithmen-Software-Hardware-Architektur des Radarprozessors als Gesamtsystem in einem Architekturschritt entworfen und nicht – wie häufig beobachtet – zu einem zu frühen Zeitpunkt die Hardware-Entwicklung von der Software-Entwicklung entkoppelt. Zukünftig wollen wir die im Modell enthaltenen Informationen auf automatischem Weg noch weiter in den Schaltungsentwurf für FPGAs hineintragen, so sollen die auf FPGAs zu realisierenden ausführbaren Software-Komponenten direkt in Simulink-Komponenten abgebildet werden oder – wie in [10] – skizziert, in VHDL-Coderahmen transformiert werden. Außerdem verfolgen wir bei EADS im Rahmen des EU-Projekts JEOPARD[5] die Integration von FPGAs und Multi-Core Mikroprozessoren unter Verwendung einer zu entwickelnden Java virtuellen Maschine.

6 Danksagung

Die Entwickler des Radarprozessors bedanken sich herzlich bei Stefan Queins von Sophist[6] für seine Beratung und Unterstützung in der Systemmodellierung mit UML. Außerdem danken die Entwickler ihren Kollegen Pierre Bayerl, Jutta Wenke, Marcel Hallmann, Magnus Jans und Maria Seidel für die Entwicklung des aktuell genutzten Codegenerators.

Literatur

[1] Gessler, Ralf ; Mahr, Thomas: *Hardware-Software-Codesign*. Vieweg-Verlag, 2007

[2] Gessler, Ralf ; Mahr, Thomas ; Wörz, Markus: Modern Hardware-Software Co-design for Radar Signal Processing. In: *ISSSE 2004, Linz*, 2004

[3] Lemke, Christina ; Mahr, Thomas: Weighted spectral estimation for nonuniformly sampled radar signals. In: *Antennas, Radar, and Wave Propagation 2007, Montreal, Canada*, 2007

[4] Lemke, Christina ; Mahr, Thomas ; Kölle, Hans-Georg: Coherent parameter estimation for radar signals. In: *International Radar Symposium 2007 Köln*, 2007

[5] Parnas, David ; Clements, Paul: A Rational Design Process: How and Why to Fake It. In: *IEEE Trans. Software Eng.* (1986), February, S. 251–257

[6] Rupp, Chris ; Queins, Stefan ; Zengler, Barbara: *UML 2 glasklar*. 3. Auflage. Hanser Fachbuchverlag, 2007

[7] Ambler, Scott W.: *The Elements of UML 2.0 Style*. Cambridge University Press, 2005

[8] Pilone, Dan ; Pitman, Neil: *UML 2.0 in a Nutshell*. O'Reilley Media Inc., 2005

[9] Ludloff, A.: *Praxiswissen Radar und Radarsignalverarbeitung*. Vieweg, 1998

[10] Mahr, Thomas ; Schillinger, Patrick ; Kirchner, Daniel ; Fürchthauer, Andreas: UML-based Automatic Code Generation for Hybrid CPU-FPGA Radar Processors. In: *FDL'06, Darmstadt*, 2006

[5] http://www.jeopard.org
[6] http://www.sophist.de

Single Chip FPGA Implementation of a Massive Parallel Real-Time Correlator Architecture

C. Jakob[*], A. Th. Schwarzbacher[+], R. Peters[†] and B. Hoppe[°]

[°]Department of Electronic and Computer Science, Hochschule Darmstadt, Germany

[†*]R&D, ALV-GmbH, Langen, Germany

[+*]School of Electronic and Communications Engineering, Dublin Institute of Technology, Ireland

[*]christian.jakob@dit.ie, [+] andreas.schwarzbacher@dit.ie, [°] hoppe@eit.h-da.de,
[†]rap@alvgmbh.com

The mathematical construct of correlation is used in photon correlation spectroscopy (PCS) experiments to analyse the diffusion behaviour and thereby the size and distribution of nano particle structures in fluid media. The tempo spectral information in scattering light from a sample which is exposed to a laser beam is extracted using correlation techniques. Besides the demand of evaluating the correlation function over a large spread of time scales, ranging from nanoseconds to hours, many experiments require the simultaneous sample investigation under different scattering angles to allow a spatial resolved measurement. This paper presents a novel multichannel single chip correlator architecture which for the first time allows the parallel processing of up to 32 correlation functions. This was achieved by integration of the system in reconfigurable hardware, which is partitioned in a fast hardware front-end combined with an application specific optimised softcore processor.

1. Introduction

Submicron particle characterisation using photon correlation spectroscopy techniques represents a well established method in different scientific and technological areas [1]. The spectrum of application reaches from chemistry, physics, over pharmacology, molecular and cell biology, immunology right to environmental research [2]. Many recent pharmaceutical advances are based on a detailed knowledge of the size of the underlying particle involved [3]. Particle characterisation represents an important part in the product research and development, since even slight variations of the particle size can lead to instability of a product, rendering the product difficult to use or even useless [4]. In case of ink and toner products, the image quality, viscosity and the tendency to aggregate and clog ink delivery nozzles are directly related to the particle size. In light scattering experiments, the detected light in form of single photon counting pulses is well suited for the processing by digital hardware due to their digital nature. The real-time computation of correlations functions has to cope the wide spread of time-scales (ns to hrs), corresponding to a dynamic range of 10^{13}. Simultaneously, a resolution of less than 10ns is desired to precisely locate the photonic events and to discriminate between different pulses. This paper outlines how these conflicting criteria can be balanced by the use of multiple sampling times (Multiple-Tau technique) and an efficient and highly optimised hard- and software architecture. Furthermore, the multichannel correlator architecture, implemented on a FPGA device as a HW/SW partitioned system, is presented. The design considerations are outlined to achieve a high time dynamic range for 32 channels on the one hand and on the other hand to implement as many as possible correlation processing units on just a single chip.

2. Photon Correlation Spectroscopy

Particles suspended in a liquid have never a fixed position. They are constantly moving due to the Brownian motion, which is defined as the movement of particles due to the random collision with the solvent molecules that surrounds the particles. The size of the particles defines their velocity and diffusional behaviour. The larger the particle of interest, the slower its motion. Smaller particles are shoved further by the solvent molecules and thus move more rapidly. Or in other words, a larger particle will diffuse more slowly than a smaller particle and vice versa. The technique of Photon Correlation Spectroscopy (PCS), measures this Brownian motion (also referred to as 'random walk') and relates this directly to the diffusion coefficient, and with that to the hydrodynamic radius of the particle. The hydrodynamic radius of particle denotes the dimension of a sphere that has the same

translational diffusion coefficient as the investigated particle.

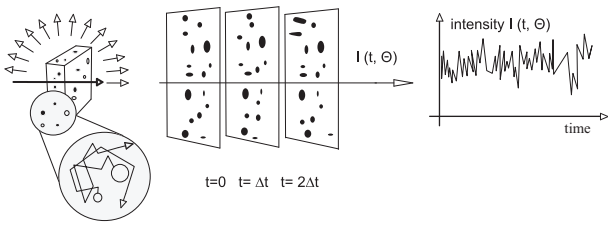


Figure 1: Time Varying Scattering Intensity at Angle θ

The relationship between the size of a particle, the hydrodynamic radius R_H and its diffusion coefficient is defined in the Stokes-Einstein Equation [5]:

$$D = \frac{k_B T}{6\pi R_H \eta} \qquad (1)$$

In order to measure the diffusion coefficient D of the solvent and thus be able to calculate the hydrodynamic radius of a particle R_H, the absolute temperature T and the viscosity η of the solvent must be accurately known. The letter k_B denotes the Boltzmann constant. The determination of the diffusion coefficient starts with illuminating the arrangement of particles by a monochromatic and coherent light beam, such as a laser. While in static light scattering (SLS) experiments the average scattering intensity is measured, in the photon correlation spectroscopy approach the temporal fluctuations of the intensity fluctuations are observed. This coherence is illustrated in Figure 1. The used source of light is a solid state diode laser with a wavelength of 500nm or 600nm. The DLS detectors are interfaced with single avalanche photodiode (APD) detectors.

The random movement of particles leads to a random fluctuation in detected intensity. However, the rate at which these intensity fluctuations occur evidently contains the information about the dynamics of the solute molecules. In general, large particles cause the intensity to fluctuate more slowly than the small ones and vice versa. Thus, these fluctuations contain the information of how fast individual molecules diffuse. It would now be possible to measure the spectrum of frequencies contained in the intensity fluctuations. A better and more efficient approach is to calculate the correlation function of the detected intensity fluctuation signal x:

$$Corr(\theta)_{x,x} = \int_0^T x(t)x(t+\theta)dt \qquad (2)$$

The letter θ denotes the time shift, also referred as the 'lag-time'. If the particles in the solution are large, it can be observed that the intensity signal will change slowly and therefore the corresponding correlation function will persist for a long time. Otherwise, if the sample contains primarily small, fast moving particles, the course of the correlation function tends to decrease more rapidly.

3. Correlation Algorithms

Correlation represents the mathematical construct to measure the similarity between two different arbitrary waveforms and is applied in a variety of digital signal applications [6]. In the context of this paper, correlation techniques are used to extract the tempo spectral information from scattering light as explained in Section 2. A digital correlator calculates a time discrete approximation of the continuous auto-correlation function stated in equation (2) at certain discrete lag-times θ_j:

$$Corr(\theta_j)_{n,n} = \sum_{i=1}^{N} n_i n_{i+j} \quad \Leftrightarrow \quad < n_i n_{i+j} > \qquad (3)$$

The parameter n denotes the detected photon count rate, i.e. the number of photon counts over one sampling time clock (STC) interval. The STC range, which represents the inverse of the system clock f_{sys}, is defined as $\tau = t_{i+1} - t_i$, Considering a linear correlator structure, the dynamic range of the correlator is determined by the spread of discrete lag times θ_j: $\theta_{min} < j\theta < \theta_{max}$. Thus, the different lag-times in the linear correlator scheme are separated by one sampling time interval each. For every single lag-time θ_j, an individual correlation channel $Corr(\theta_j)$ exists, that

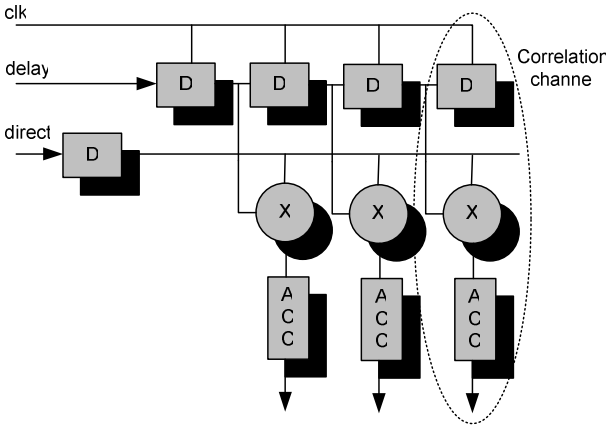


Figure 2: Linear Correlator Architecture

simply sums the products of the direct and delayed samples $n(0)$ and $n(\theta_j)$. This multiplication/summation procedure is done for a certain number of different lag-times in parallel to obtain a time discrete

approximation of the full correlation function. A linear correlator, as illustrated in Figure 2, can be easily implemented of simple multiplier-accumulator structures. Apparently, the lag-time range of a linear correlator is limited by the number of channels. To achieve the covering of a dynamic range of 10^{12}, the number of correlation channels must be at least as high as the temporal range, i.e. 10^{12}. Since hardware resources are limited, the required dynamic range is in direct conflict with a fine timing resolution to accurately approximate the correlation time integral.

A solution to this problem represents the Multiple-Tau algorithm published by K. Schaetzel [7]. The major difference to the linear correlator structure is, that the sampling time is no longer held constant, but rather increased with the lag-time. In fact, this leads to an implicit calculation of all possible products of direct and delayed samples. Blocks of 8 or more correlation channels with common sampling time are formed (in the following referred as sampling time blocks STB) and the sampling time is doubled from one block to another. Thus, the input signals for the different blocks are averaged over longer and longer periods. All events in the delayed and undelayed path are added up for each block over two sampling time periods and serve as input count rates for the next block, which operates again with half of the clock frequency. If a 16 channel sampling time block structure is applied, the first 8 of 16 correlation channels of every sampling time block STC_n but STC_0 are redundant. The lag-time range of these was already covered by the previous STC block, but even at a much higher temporal resolution due to the decreased sampling times. It would thus not make much sense to have all correlation channels for each of the individual STC blocks being computed from lag-time index *j=0* on, but instead only those correlation channels should be computed which cover a not previously covered lag-time regime. The processing is thus reduces to 16/8 scheme [8] and the Multiple-Tau correlation function estimation of the input count rates *n* and *m* can be summarised as follows:

$$Corr(j)_{n,m}^{STC_0} = \sum_{i=1}^{M} n_i^{STC_0} m_{i+j}^{STC_0} \qquad (4)$$

The lag-time *j* in equation (4) is defined over the range 0…15, The additional *l=1..P* sampling time blocks consisting of eight channels each, can be calculated as:

$$Corr(j)_{n,m}^{STC_l} = \sum_{i=1}^{M} n_i^{STC_l} m_{i+j}^{STC_l} \qquad (5)$$

For the STBs *l=1..P*, the lag-time index is restricted to *j=8..15*. The effective sampling time for every correlation channel block is determined by:

$$STC_l = STC_0 2^l \qquad (6)$$

The effective lag-time for every correlation channel can now be calculated as:

$$\Delta\tau = STC_l\, j = STC_0 2^l\, j \qquad (7)$$

Using this procedure, only 40 parallel sampling time blocks (or 328 correlation channels) are required to cover a dynamic range of nearly 10^{13} with sampling times as small as 5ns for STB_0 and nearly 1hr for the last STB block.

4. System Architecture

Due to the small volume market of photon correlation spectroscopy setups, a FPGA implementation was chosen instead of an ASIC design. The fundamental principle of the 32 channel correlator system is based on the Multiple-Tau algorithm discussed in section 3. A direct logic cell integration of 32 Multiple-Tau structures, each with a time dynamic range of 10^{13}, would lead to an immense demand of logic resources. The power dissipation of a direct implementation was estimated to approximately 15W, which represents an enormous effort regarding the heat transfer. Thus new concepts had to be found in architectural correlator design, replacing the original one-to-one implementation. The new developed correlator architecture is based on the extensive usage of FPGA on-chip features as hardwired multiplier-accumulator cells or embedded memory blocks, which work at higher data rates while consuming less power than conventional cell logic. For the used FPGA, the power dissipation of a DSP cell (18x18 MAC) operating for example at 400MHz was estimated to just 8mW, a 4kbit RAM at the same rate to around 1.3mW. The developed multichannel correlator architecture is divided into a fast hardware frontend, which is responsible for the processing of the lower STBs. An application specific and highly optimised softcore processor unit realises the system-host communication as well as the processing of the higher STBs. With this partition and the intelligent and low-power oriented usage of distributed FPGA resources, a time dynamic range of 10^{13} could be achieved for every detector input channel.

4.1. Hardware Implementation

The correlator front-end, operating at 100MHz, processes the first 11 sampling time blocks. Additional 16 STBs for each input channel are processed by the subsequent processor. The front-end is split-up into four identical stages, each responsible for the processing of up to 8 input channels. This structure, illustrated in Figure 3, consists of eight derandomiser

stages which count the photonic events on the respective channels. This is done within a selectable sampling time interval to synchronise the random input signals. Furthermore, the incoming asynchrony pulses are synchronised to the system clock to prevent the input stages from metastable states. The resulting data words are stored in eight FIFO memory blocks and are further processed by the stages correlation core, implementing the actual correlation data processing.

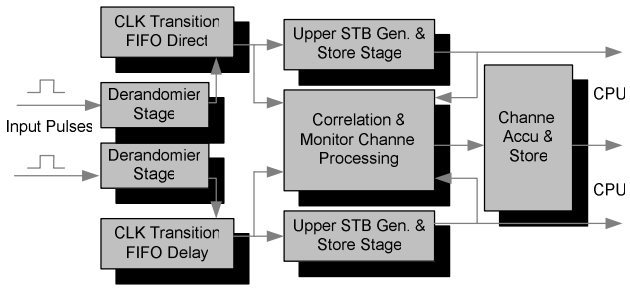


Figure 3: Fast Hardware Front-End

The calculation of the correlation function is based on a bottom-up approach. The first delay and direct data sets of size 4096 are directly processed by the correlation core to calculate STB_0. Simultaneously, they are accumulated and temporarily stored in the upper STB generation block to form the 2048 direct and delay data word sets for STB_1. The correlation results are passed to the channel store unit which adds up the results with the previous calculated ones and stores them in a dual-ported memory block. This allows on the hand the storage of the computed STBs, on the other hand the already calculated STBs can independently be read out by the embedded processor. In the next step the generated data set of STB_1 is again added up to form the following sampling time block, STB_2, while the correlation data for STB_1 is calculated. This is done subsequently for 11 STB. The accumulated data sets for STB_{11} are stored in a FIFO memory which is read out after 64 runs by the processor. To normalise the overall correlation function, a symmetrical normalisation as proposed in [9] is applied. This processing, which is done on the host station, requires the counting of direct samples for every STB as well as the counting of delayed samples for every single STB channel. These counters are implemented as monitor channels within the correlation core. The four stages are scheduled by a single state machine, which generates the respective control signals as well as the memory address signals. The overall system operates in different processing modes what enables the simultaneous real-time computation of 32 independent auto-correlation functions or the calculation of the cross-correlation between selectable input channels so that the setup can be optimally adapted to the requirements of various experiments. Furthermore, a built-in-selftest (BIST) unit was integrated to verify all portions of the system functionality on power-up.

4.2. Software Implementation

The upper 16 sampling time blocks for each channel are computed on an embedded, general-purpose RISC processor, running at 50MHz. Besides the STB processing, the processor is responsible for the system-host communication and the initial configuration process. The evaluation of the correlation function was implemented using a stack processing scheme to guarantee the correct computation of the respective STBs at the necessary points in time. The GNU performance profiler [10] was used to determine in which functions the program is spending most of its execution time. It was figured out that the correlation channel processing is consuming the most CPU time. A pure software based correlation processing procedure requires 37.5 cycles for a single correlation channel computation. This could be speed up by the use of an embedded multiplier, still resulting in 27 cycles per correlation channel processing. Besides the multiplication procedure, the memory accesses were determined as time critical due the inserted wait states by the processors internal bus controller. This was overcome by extending the processors instruction set architecture with an internal correlation processor as illustrated in Figure 4.

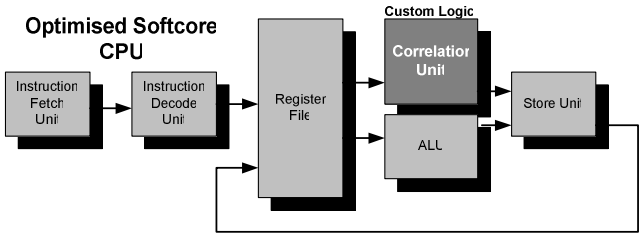


Figure 4: Optimised Softcore CPU

The correlator unit was realised by combining an embedded multiplier with an internal FIFO based correlator delay chain, which significantly reduced the required memory read accesses. In this way, the overall correlation processing performance could be increased by more than a factor of 10, see figure 5. With a mean value of 3.5 cycles per correlation, the customised processor achieves digital signal processor (DSP) qualities with an additional negligible low, hardware unit. By extending the processors instruction set architecture by an additional correlation

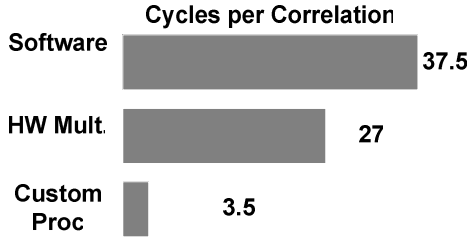


Figure 5: Cycles Per correlation

unit, the processing of 32 independent channels in real-time could be easily met.

5. Infield Measurements

Figure 6 shows an auto-correlation function measured with the presented Multi Correlator system. The sample under investigation was a solution, containing silicate particles of several distinct sizes. Application of silicate particle solutions can be found among others in the semiconductor industry [11]. Wafers are polished in specific steps of the fabrication process with silicate abrasive slurry to planarise a surface.

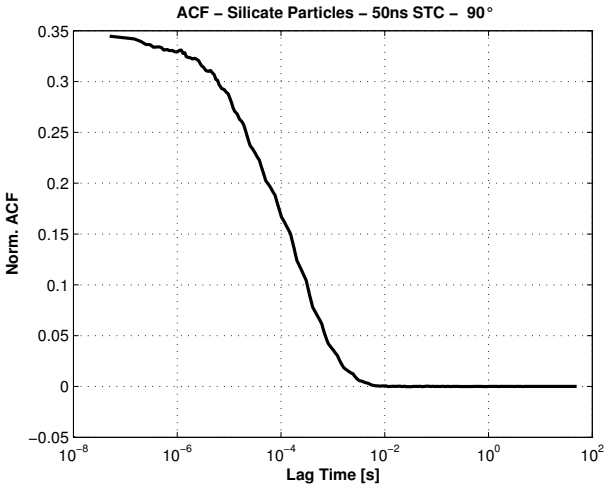


Figure 6: ACF – Silicate Particle Solution

The evaluation of the measured correlation function resulted in a particle diameter distribution of 5nm, 61nm and 400nm. Both latter mentioned particle sizes bear witness to unwanted aggregation and macro-crystal formation effects within the sample. In the carried out DLS experiment, using a mean count rate of 300k counts, a laser wavelength of 532nm and a detection angle of 130°, the MultiCorr system achieved in a measurement time of 100s identical statistical results as the commercial available ALV 5000/EPP [12].

6. Conclusion

This paper has presented the successful implementation of an innovative, completely tested and fully functional system-on-programmable-chip (SOPC) based digital real-time multichannel correlator. This was achieved by replacing the traditional one-to-one implementation of the Multiple-Tau algorithm by a recursive, low power processing architecture, specifically developed for FPGA technology. Furthermore, a highly optimised softcore CPU was implemented. This allowed even the computation of time-critical sampling time block structures in software with a half the clock speed of the front-end without any loss of performance. The correct operation of the newly developed multichannel single chip correlator was fully verified with various embedded system test structures. Furthermore, extensive in-field tests combined with a comparison of measurements taken using the industrial ALV-5000 correlator showed that the implemented system completely fulfils the requirements of real-world, high-end DLS experiments.

7. Acknowledgements

The authors would like to thank the ALV-Laser Vertriebsgesellschaft mbH in Langen, Germany for their support in this project.

8. References

[1] R. Berne, B. Percora. *Dynamic Light Scattering: With Applications to Chemistry, Biology and Physics*. Dover Publications, 2000.

[2] K. Schaetzel. Correlation techniques in dynamic light scattering. *Journal of Applied Physics B*, pages 193–213, 1987.

[3] X. Renliang. *Particle Characterization: Light Scattering Methods*. Springer Netherland, 2000.

[4] D. Johnson, S. Gabriel. *Laser Light Scattering*. Dover Publications, 1995.

[5] P. A. Tipler. Physics for scientists and engineers. *W. H. Freeman*, 2003.

[6] Seo, K. M., BobL. H., *Real-Time Digital Signal Processing: Implementations and Applications*, Wenshun TianWiley; 2nd edition, 2006.

[7] K. Schaetzel. New concepts in correlator design. *Proc. Of the int. Phys. Conference, Ed. E, Hilger*, Ser. 77:175–184, 1987.

[8] R. Peters. The multiple-tau correlation technique. *ALVGmbH*, 1996.

[9] K. Schaetzel. Photon correlation measurements at large lag times: Improving the statistical accuracy. *J. Mod. Opt.*, 35:711–718, 1988.

[10] The GNU Performance Profiler, www.sourceware.org, (January 2008).

[11] C. Huynh. A study of post-chemical-mechanical polish cleaning strategies. *Advanced Semiconductor Manufacturing Conference and Workshop*, page 372 376, 1998.

[12] ALV-GmbH. *www.alvgmbh.com*. 2008.

FPGA Implementation of a Single Pass Real-Time Blob Analysis Using Run Length Encoding

J. Trein[*], A. Th. Schwarzbacher[+] and B. Hoppe[°]

[*°]Department of Electronic and Computer Science, Hochschule Darmstadt, Germany

[*+]School of Electronic and Communications Engineering, Dublin Institute of Technology, Ireland

[*] johannes.trein@web.de, [+] andreas.schwarzbacher@dit.ie, [°] hoppe@eit.h-da.de

In the fields of machine vision and image processing applications the blob analysis became a well known method to detect objects in a digital image. Together with the increasing resolutions and frame rates of recent digital video cameras the requirements on the hardware to perform the blob analysis are very high. Software implementations may not be able to accomplish a satisfying performance. Furthermore, existing hardware solutions require a processing of the picture in multiple passes. This paper describes the development of a novel FPGA algorithm, performing a high speed real-time blob analysis using only one single pass. The input data are compressed using run length encoding. This allows the use of a full configuration camera link interface with data rates of up to 680MByte/s. Furthermore, object properties are passed to the host PC on-the-fly which reduces the latency to a minimum while at the same time allows the reuse of object labels thus achieving a memory efficient implementation.

1. Introduction

Since the development of the digital image processing during the 1980s it has always been very important to extract information from an image to allow decisions depending on the image content. These automated decisions improved many production and scientific processes in *machine vision* and *computer vision* applications [1]. An important step while gathering information from an image is the detection and description of objects. One of the most common techniques to detect objects from of a two-dimensional image is the *blob analysis* [1]. It describes the segmentation and mapping of neighbour foreground pixels to detect an object and the characterisation of shape parameters to determine the properties of the objects found. Afterwards, these object properties or object features can be used for the classification of objects which in turn leads to the possibility of making machine made decisions.

Blob analysis is basically used for the detection, counting and sizing of objects. Figure 1 shows a typical application of the blob analysis. In the original image, on the left, are five plugs where the two overlapped plugs have to be detected. The middle image shows the thresholded version of the original image. Now, every white pixel is assumed to belong to an object. Hence, this image is used to perform the blob analysis itself. The right image shows a visualisation of the results. In this example, the overlapping objects can be found easily by the sum of their foreground (white) pixels.

Respective applications of the blob analysis are the detection of surface defects in materials such as packaging material or on glass used for TFT monitors. Another application is the final inspection of production outcomes, for example the correct packing and shapes of paper carton boxes [2]. The blob

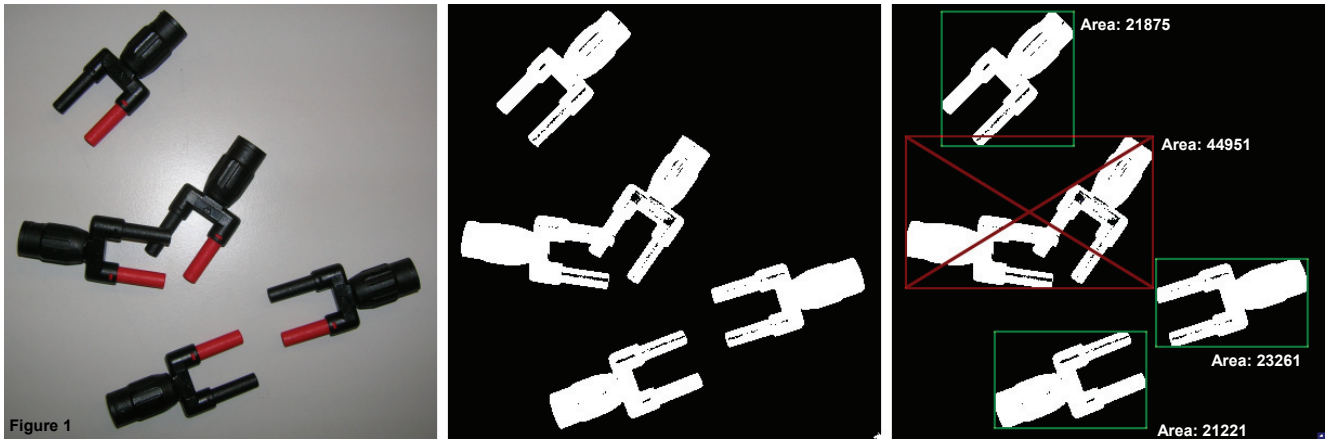


Figure 1

analysis is also used in position compensation [1] and to trigger line-cameras. One of the most famous paradigms to demonstrate the blob analysis is the detection of the correct filling of blister [3].

The growing demands on speed together with the increasing resolutions of digital video cameras require a real-time blob analysis [4]. The high performance of today's digital cameras may only be realised in microprocessor based systems with serious performance constraints and high effort. The problem of a processor based system is the serial processing of data which cannot reach the performance of a parallel hardware implementation. Even modern processors which operate in the gigahertz range cannot compensate this performance gap because of a limited RAM bandwidth [2].

The rapid development of field programmable gate arrays (FPGAs) during the last years, offers new possibilities. Complex algorithms can now be implemented into hardware without a high increase in design effort and cost. Furthermore, FPGAs enable almost unrestricted parallel processing of data and are flexible to use due to their re-programmability. Modern FPGAs facilitate built-in memory blocks and multipliers which enable the realisation of time critical and memory extensive applications.

This paper describes the FPGA investigation and implementation of a real time blob analysis algorithm. Digital images are transferred from the camera into the FPGA and are analysed in real time. The resulting object features are then passed to a host PC for further processing. Also, the algorithm facilitates the calculation of numerous object features such as area, centre of gravity, contour length, orientation and image moments [5]. The hardware is based on a Silicon Software [6] frame grabber system where the FPGA is a XILINX Spartan II or Spartan III [7] device. It operates with a core frequency of 50MHz. Through the parallel processing of up to 32 pixels and a pipelined architecture it is possible to process 1600MPixels/s theoretically. This however, is limited to 680Mpixels/s because of the input interface. The implementation is carried out in a low-cost FPGA and therefore a cost efficient realisation of the developed real-time single pass blob analysis algorithm is possible. Furthermore, the algorithm is suitable for general purpose blob analysis. This means, it is highly adaptable to various applications through many parameters which may set, for example the bit width or the required object features. Together with existing pre-processing operators provided by Silicon Software the resulting implementation is a powerful machine vision tool.

2. Existing Solutions

In literature, such as in [8] and [9], numerous algorithms can be found to perform a blob analysis or an object labelling procedure. These algorithms are based on the idea that in a first step all pixels of objects are labelled with a respective object number. This operation requires two passes through the whole image. Afterwards, in a third pass, the features of the labelled pixels, and so the features of the objects, are calculated. Unfortunately, it is not possible to transfer this basic algorithm into hardware because of FPGA constraints. In a PC the captured image data is stored in the system memory. The microprocessor then has random access to this data. This is not feasible inside a FPGA, because the built-in block RAM is not sufficiently large for the required data volumes. Furthermore, a pipelined implementation for this algorithm is not possible as the data has to be stored first, before post-processing and performing the blob analysis can be carried out. Consequently, the analysis has to be performed in a raster-scan procedure at the rate of the camera transfer. Thus, random access on the image content is not possible. Moreover, it is not feasible to process the whole image twice because of a lack of available memory capacity as well as to reduce system latency. For example it is not possible to process the image from the bottom after the first pass like many algorithms require [9]. The novel proposed algorithm will address these problems and will successfully solve them as will be shown in the following section.

The main advantage of the developed algorithm is to be capable of performing the blob analysis in a pipelined single pass fashion. Through a compression, multiple pixels can be processed in parallel. The object features are passed on to the host PC as soon as they can be determined which reduces the latency to a minimum.

3. Developed Hardware Algorithm

The developed algorithm is based on the idea of directly capturing the frames from a digital camera, perform the blob analysis and afterwards pass the results on to a host PC. To transmit the frames between the camera and the FPGA the 'camera link' standard [10] is used as an interface. This interface allows the transmission of up to eight pixels in parallel at a clock frequency of up to 85 MHz and is used as a standard in machine vision applications. Furthermore, a timing synchronisation before data processing in the FPGA is required. Thus, the pixels are buffered in a SDRAM first. From this buffer, the pixels are read with a typical clock frequency of 50 MHz. As described, the clock frequency of the camera link is higher. To avoid a limitation of these data rates inside the FPGA a set

of subsequent pixels is read from the buffer in parallel. This may be up to 32 pixels which are transmitted in one clock cycle.

3.1. Run Length Encoding

The implemented algorithm is using a raster scan procedure to determine the objects and calculate their properties. This means every pixel of the image has to be scanned line by line to determine the objects. As described above, up to 32 subsequent parallel pixels are read from the buffer in one clock cycle. Hence a raster scan would not be possible. A run length encoding of these parallel pixels solves this problem. This procedure compresses the pixels and therefore on average again multiple pixels can be transmitted in only one clock cycle. The run length encoding is based on the assumption that black and white pixels inside a line continue without the change of colour over longer sections. These sections, called runs, can be described by their start and end position. Thus, in one clock cycle only one start and one end value is transmitted, but the run may describe multiple pixels. This allows the transmission of multiple pixels in parallel, can maintain the parallel input speed and still allows a raster-scan procedure to determine the objects as it is shown in the next section.

3.2. Object Detection

The object detection is based on the idea that every foreground pixel is compared with its neighboured pixels. If one of the neighboured pixels is also a foreground pixel, they must belong to the same object. This is performed by scanning through the image line by line where every pixel is compared with its neighbours. The 'mask' in Figure 1 shows the relation of a pixel to its neighbours. If the current pixel has no other foreground pixels inside its mask it is assumed to be the start of a new object. Hence, the pixel is labelled with a new object number. If at least on of the neighbours of the current pixel is a foreground pixel, it has to belong to the same object and therefore gets the same label number as the neighboured pixel. These two operations are illustrated in Figure 2. In the left image the current pixel has no neighboured foreground pixels in its mask. Thus it is assumed to be the start of a new object and it is labelled with a new object number. In the right image, the current pixel has a neighboured pixel inside the mask. Hence, the current pixel belongs to the same object and is labelled with the same object number.

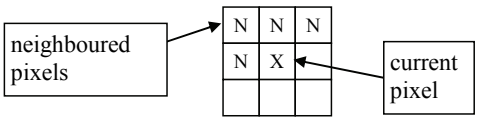


Figure 1: Mask: Pixels in the neighbourhood of X

As explained, the pixels of this implementation are compressed by a run length code before this comparison. But the method of comparing pixels with its neighbours can also be used with runs. Here, a respective current run is compared with the runs in its previous line. If the start and end positions of the runs overlap, they have to belong to the same object. Thus they are labelled with the same object number. Figure 3 represents the same image such as the one shown in Figure 2. Here, the pixels are described by run length codes, hence described by the pixel-sequence start and end positions. In the left image, the current run does not overlap with any of the runs of the previous line. In the right image the current run overlaps with a run of the previous line. Therefore, they must belong to the same object.

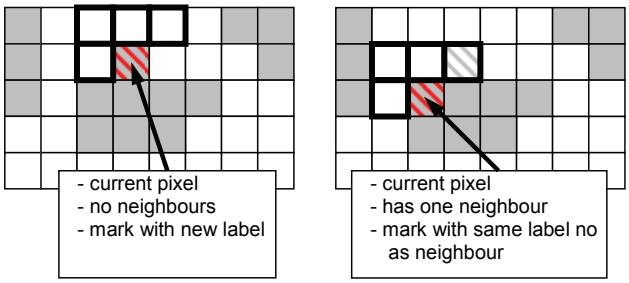


Figure 2: Detection of a new object (left) and adding a pixel to an existing object (right)

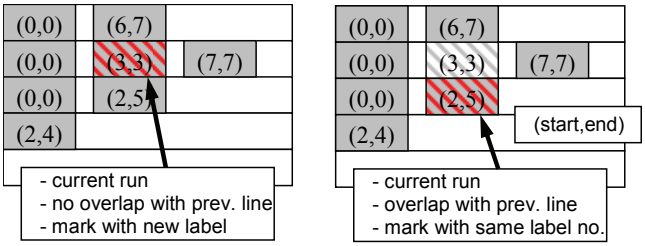


Figure 3: Comparing runs

As seen in this example, it is only necessary to compare a respective current run with the runs in the respective previous image line. The algorithm is scanning through the lines while comparing the runs of a respective current line with its antecedent line. Using this method the relations of the runs are determined and they can be associated to objects and marked with an object number.

Until now, runs only have been marked with an object number, but it should be noted that it is not the aim of a blob analysis to get images of the found objects. The blob analysis only has to determine the object properties. Furthermore, it is not possible inside the FPGA to keep all marked runs of an image in a memory. Therefore, a new method is developed. First, every run is allocated a specified object number, as was explained above. However, this is only applied to a respective current line and the previous line. After the respective previous line has been compared with the current line its data is discarded. Hence, the runs

which have been compared with other runs are not available anymore. But how can the object properties being determined if the runs are discarded. The solution is the object property calculation or object feature calculation right after the comparison of a run and its allocation to an object number. If a new object is detected while comparing the runs, an object number is allocated to this run like explained before. This number is equal to a memory address in a special object feature RAM. For every run which has to be added to an object, the properties are calculated. Next, the properties are added to the properties calculated before in the object feature RAM. Thus, the object properties 'grow' with every new run added to the existing properties in the RAM. If an object in the RAM is not updated any more it is assumed to be completed and its properties are passed to the host PC. The procedure can be summarised as follows:

1. comparison of a run with runs of the previous line
2. association with an existing object or creation of a new object
3. allocation of an object number
4. calculation of the properties of the run and update of the properties in the object feature RAM

3.3. Object Merges

So far, the algorithm works for objects with simple shapes. More complex objects may have shapes which merge or divide. Two objects which are assumed to be individual when comparing early rows can merge together into one object later. An example of this is shown in Figure 4. While performing a blob analysis these merges cause problems. This is a reason why several algorithms need to proceed through the image a second time. These algorithms 'clean up' the object associations by proceeding through the image in inverted direction [9].

The innovative algorithm presented here solves this problem in a novel way. As the properties are calculated while detecting the objects, the problems of merges can be solved without a second pass through the image.

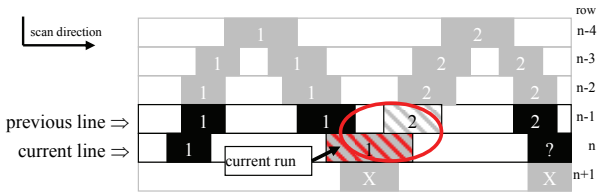


Figure 4: The merging of two objects

In Figure 4 the current run is compared with the run, marked in the previous line. As they overlap, they must belong to the same object. However, they are labelled with different object numbers, because it was assumed that they belong to different objects while scanning the previous lines. To solve this, all properties of object "two" which have been calculated so far, are updated into the properties of object "one". To avoid further updates of the obsolete object "two", a pointer in the object feature RAM at address "two" is set which redirects all further updates to object "one". The flow chart in Figure 5 summarises the object detection algorithm.

This example shows that all object properties can be determined correctly. A second pass is therefore not necessary. Hence, the algorithm can be performed in only a single pass, without the need of complex storage operations.

3.4. Object Completion

In the previous section the 'growing' of object properties in the object feature RAM has been described. Now the completed objects have to be passed to a host PC. This has to be done immediately after the object properties in the RAM are completed, even if the object detection scan through the image is still in progress and other objects may still need to be detected. So the latency and the required memory size of the object feature RAM are reduced because memory addresses and object labels can be reused for other objects.

A garbage collection mechanism searches in the object feature RAM for completed objects and obsolete objects. An obsolete object is an object which points to another object after a merge. An object is completed if it has not been updated during the last 2 lines. The garbage collector passes these completed objects to a host PC and makes the memory addresses in the RAM available for new objects.

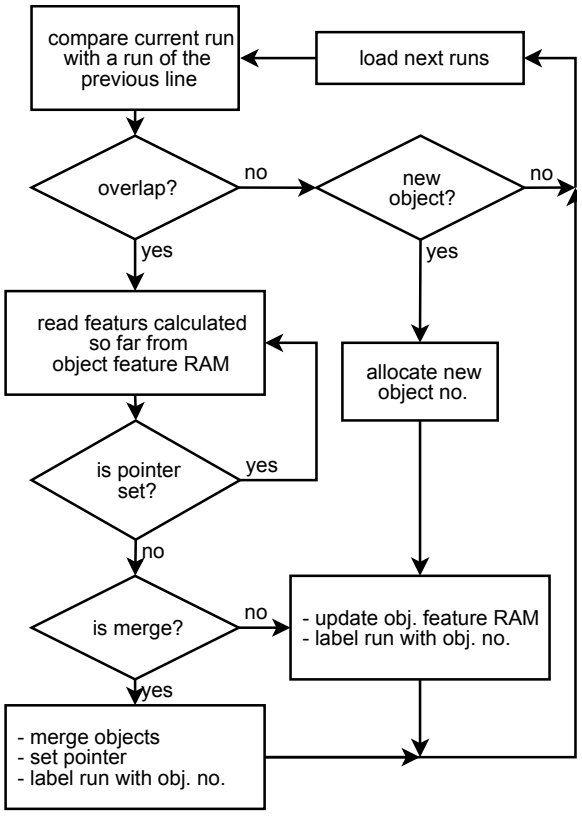


Figure 5: Flow chart of the algorithm

3.5. Feature Calculation

The calculation of object properties or object features can be divided into two parts. On one side, the features calculated so far and stored in the object feature RAM have to be updated by new runs which are added. On the other side, the two object features caused by an object merge have to be added. Here the features *area*, *bounding box* and *centre of gravity* are explained. The *area* is the sum of all pixels of an object. A recurrence relation which describes the update of a run to the area calculated so far can be described by

$$A^* = A + (e - s + 1) \qquad (1)$$

where e is the end-position of a run and s the start-position. Thus, $e - s + 1$ is the length of a run. The *centre of gravity* is determined for the x-axis and y-axis.

$$\bar{x} = \frac{1}{A}\sum_{(u,v)\in R} u \qquad \bar{y} = \frac{1}{A}\sum_{(u,v)\in R} v \qquad (2)$$

where A is the area. u and v are coordinates of the object R. The recurrence relation will then be

$$x^* = x + \frac{e^2 + e - s^2 + s}{2} \qquad y^* = y + (e - s + 1)\cdot l \qquad (3)$$

where l is the line number of the run. x and y have to be divided by A afterwards to complete the calculation. The *bounding box* is the minimum paraxial rectangle which includes all object pixels. It can be easily determined by comparisons. More features are defined in [8].

4. Hardware Used

The implementation of the algorithm is performed on a XILINX Spartan FPGA [7] which is embedded to a Silicon Software frame grabber system. This frame grabber is a PCI-express or PCI64-bit PC extension card [6]. The card has up to four external camera link interfaces for the connection between the camera and the frame grabber. After performing the blob analysis, the results are transferred over the PCI-bus into the main memory of the host PC. The complete configuration of the frame grabber is shown in Figure 6.

The presented algorithm fits into a low-cost FPGA which allows an economic implementation of the blob analysis. Embedded multipliers are used to calculate complex object features such as the centre of gravity and image moments. True dual-port FPGA block RAM is used for memory access and FIFOs.

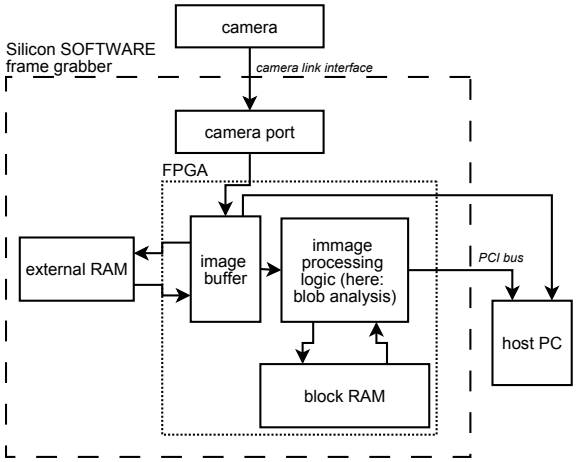


Figure 6: Configuration of the frame grabber

5. Performance

The performance of the implementation in terms of speed depends on the image content. This is because the run length encoding and the number and seize of objects in the image. As explained in Section 3.2 the input pixels are compressed by a run length code to process on average multiple pixels in one clock cycle. Because of block RAM accesses and feature calculation processes the implementation requires on

average 6 clock cycles to process one run. If the compression of the parallel input pixels has a rate of less than 1:6 the object detection will inhibit the input and therefore, cannot perform the blob analysis in real-time. However, extensive tests have shown that only for images with an artificial strong noise this gap cannot be compensated for. Furthermore, the compression results in the fact that the algorithm is independent on the input image resolution. Thus, the content and the number of objects is the determining factor for the required clock cycles.

The performance of the blob analysis is limited to the maximum input speed of the transmission of the raw data between the camera link interface and the run length encoder. An image with a resolution of 8k pixels i.e. an image with a resolution of 8.192 by 8.192 pixels has 67MPixels. At a clock frequency of 58MHz and the parallel transmission of 32 pixels a frame rate of

$$\frac{32 \cdot 58MHz}{67MPixels} \approx 28\,fps \qquad (4)$$

can be achieved. If images with a resolution of 4k are used a frame rate of

$$\frac{32 \cdot 58MHz}{17MPixels} \approx 109\,fps \qquad (5)$$

can be achieved. As explained, only for images with an added strong noise the detection mechanism will inhibit the input. Therefore, the bottleneck is the transmission of the incoming data and the camera link interface and not the blob analysis algorithm itself.

An example of a simulation is shown in Figure 7. The 1MPixels sized input image consists of three main objects. A very strong noise is added to simulate a near worst case scenario. A cut-out of the image after the binarisation is shown in the figure. The algorithm calculates the area, the bounding box, the centre of gravity as well as the orientation of the objects. The blob analysis detects a total of 60.069 objects in the image, due to the high level of added noise. Only object properties with an area greater than 100 pixels are shown in the Figure. The bounding boxes show that the real objects are then detected correctly. The algorithm requires 37.7023 clock cycles for the blob analysis. Therefore, the maximum frame rate will be still 132fps. This shows the ability of the algorithm to handle strong image noise while maintaining the high frame rates and still operating correctly.

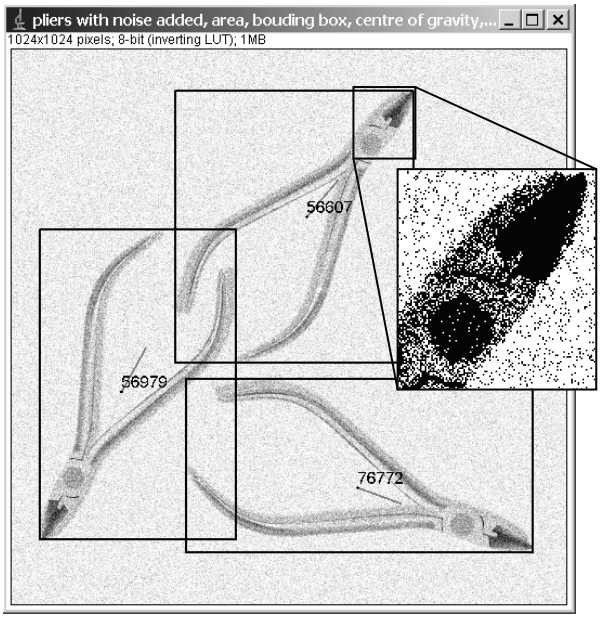


Figure 7: Simulation output of an analysed image

After having presented the image detection performance, the resources are now discussed. They are dependent on the image width, the maximum number of runs and the number of features which have to be calculated. For a normal configuration with input images of a resolution of 4k by 4k pixels the implementation requires 60kBits of block RAM and about 5.678 look up tables (LUT). This is including the blob analysis itself, the DRAM- , the camera link- and the PCI interface. The blob analysis itself only requires about 1.600 LUTs. The XILINX Spartan II XC2S600E device has a total of 15.552 LUTs and 288k block RAM bits. The implementation can be placed and routed up to a frequency of 66MHz in the Spartan II and Spartan III devices.

6. Conclusions

This paper has presented the development of a real-time blob analysis algorithm for a FPGA implementation. The incoming frames from a digital video camera are transferred to the FPGA via the camera link interface. After performing the blob analysis in the FPGA, the determined object features are passed to a host PC. By use of a run length encoding the data rates of the parallel incoming pixels can be met.

The memory inside a FPGA is not sufficient to hold a whole frame. Therefore, the image is processed in a raster scan procedure where only two antecedent rows are stored at any given time. By comparing these two rows, the objects and their properties are determined on-the-fly. This method minimises the amount of memory necessary in the FPGA. Object properties are passed to the host PC as soon as they

are completed while scanning the picture, overcoming a possible communications bottleneck and saving FPGA memory. The problem of merging objects is solved by combining their properties and setting pointers on objects which are out-of-date. Hence, a second pass through the image is not necessary and the detection of all objects is performed in only a single pass. Obsolete object labels can be reused by the algorithm which allows a memory efficient implementation.

The consistent use of dual port block RAM in the FPGA together with the algorithm focused on performance allows the real-time analysis. Furthermore, the algorithm fits in a low-cost XILINX Spartan II/E device requiring a block RAM size of about 60kBits and 5.678 LUTs. Using the run length compression, the implementation can process the data rates of a full configuration camera link interface which is 680MByte/s. Resolutions of up to 8k by 8k pixels and frame rates of up to 400fps are possible.

This blob analysis is implemented into a Silicon Software microEnable III or IV frame grabber system which allows the combination with numerous existing pre-processing operators. This makes the blob analysis a powerful tool for machine vision applications.

7. Acknowledgement

Thanks to Silicon SOFTWARE GmbH [6] which supports the research as well as provides the frame grabber hardware and software environment.

8. References

[1] I. Jahr, *Lexikon der Industriellen Bildverarbeitung,* pth-mediaberatung, Würzburg Germany, 2003

[2] K.-H. Noffz (private communication), Silicon Software GmbH, Mannheim Germany, 2008

[3] B. Jähne, *Digital Image Processing. Concepts, Algorithms and Scientific Applications,* Springer Berlin, 2005

[4] E. Davies, *Machine Vision,* Academic Press, third edition, 2005

[5] M.K. Hu, "Visual Pattern Recognition by Moment Invariants," *IEEE Transactions on Information Theory,* Volume 8, February 1962

[6] Silicon SOFTWARE GmbH, www.silicon-software.com

[7] XILINX Spartan II/III, www.xilinx.com, (April 2007)

[8] W. Burger, M.J. Burge, *Digital Image Processing in Java: An Algorithmic Introduction Using Java*, Springer Berlin, January 2008

[9] R.V. Rachakonda, P.M. Athanas and A.L. Abbott, "High-Speed Region Detection and Labeling using an FPGA-based Custom Computing Platform," *5th International Workshop on Field Programmable Logic and Applications,* Oxford, UK, Septemper 1995

[10] Camera Link by the Automated Imaging Association, http://www.machinevisiononline.org/public/articles/index.cfm?cat=129, (April 2007)

Wearlog V.2

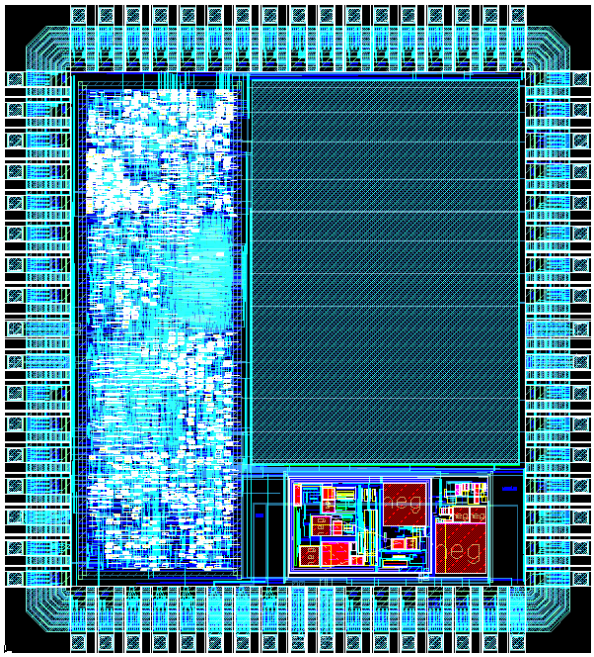


Abbildung 1: Layout auf IC-Station

Entwurf:	Hochschule Offenburg Bearbeiter: Dipl.-Ing. Daniel Bau Betreuer: Prof. Dr.-Ing. Dirk Jansen
Layouterstellung:	Hochschule Offenburg (Standardzellenentwurf & analog Design)
Technologie:	AMIS 0.35 µm CMOS M-A
Chipfertigung:	Europractice, MPW Run 1593
Herstelldatum:	April 2007
Kostenträger:	MPC-Mittel HS-Verbund Baden-Württemberg
Chipdaten:	Chipgröße: 3.5mm x 3.5mm Gehäuse: JLCC 68
Funktion:	Bei dem Chip handelt es sich um einen Microcontroller auf der Basis des FHOP V1.5 Mikroprozessor-Kernel mit integrierter Temperaturzelle und einem PLL. Als Schnittstellen sind drei 8 Bit breite parallele Schnittstellen, eine serielle Schnittstelle, eine SPI Schnittstelle und eine induktive Schnittstelle welche wahlweise nach dem RFID Standard ISO/IEC 14443 oder 15693 kommunizieren kann. Neben den gängigen Peripherien, beinhaltet das Microcontrollersystem einen 8 KB RAM und einen Wake-Up-Manager. Der Wake-Up-Manager kann das System, während einer Phase in der kein Takt anliegt, gezielt aktivieren.
Testergebnisse:	Die Funktionsfähigkeit der digitalen Schaltungen und der Temperaturzelle wurde nachgewiesen. Das RFID-Frontend sowie der PLL müssen noch in weiteren Testverfahren verifiziert werden.

ePille®

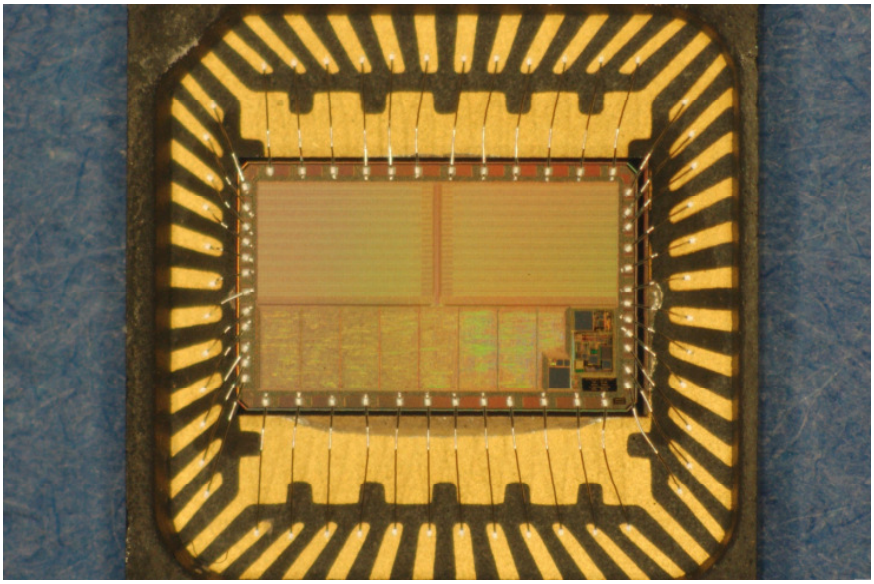


Abbildung 1: ePille-Chip mit einer Fläche von 14mm²

Entwurf:	Hochschule Offenburg Bearbeiter: M.Sc. Nidal Fawaz Dipl.-Ing. Marc Durrenberger Betreuer: Prof. Dr.-Ing. Dirk Jansen
Layouterstellung:	Hochschule Offenburg (Standardzellenentwurf & analog Design)
Technologie:	AMIS 0.35 µm CMOS M-A
Chipfertigung:	Europractice, MPW Run 1724
Herstelldatum:	November 2007
Kostenträger:	MPC-Mittel HS-Verbund Baden-Württemberg
Chipdaten:	Chipgröße: 4550µm x 2920µm Gehäuse: QFN 48 (7x7)
Funktion:	Bei dem Chip handelt es sich um einen Microcontroller auf der Basis des 16/32Bit-SIRIUS Mikroprozessor-Kernel, welcher einen 16KB RAM zur Verfügung hat. Neben den gängigen Peripherien ist eine induktive Telemetrie-Einheit, welche eine Kommunikation im Nahfeld auf einer Trägerfrequenz von 115kHz ermöglicht, implementiert. Über einen integrierten Wake-Up-Manager kann das System, während einer Phase in der kein Takt anliegt, gezielt aktiviert werden. Des Weiteren befindet sich in dem Chip noch ein hochauflösender Temperatursensor.
Testergebnisse:	Der komplette ePille® Chip wurde erfolgreich in Betrieb genommen. In einem aktuellen Testaufbau wird der IC im Sender/Empfänger Betrieb eingesetzt.

RFID 15693

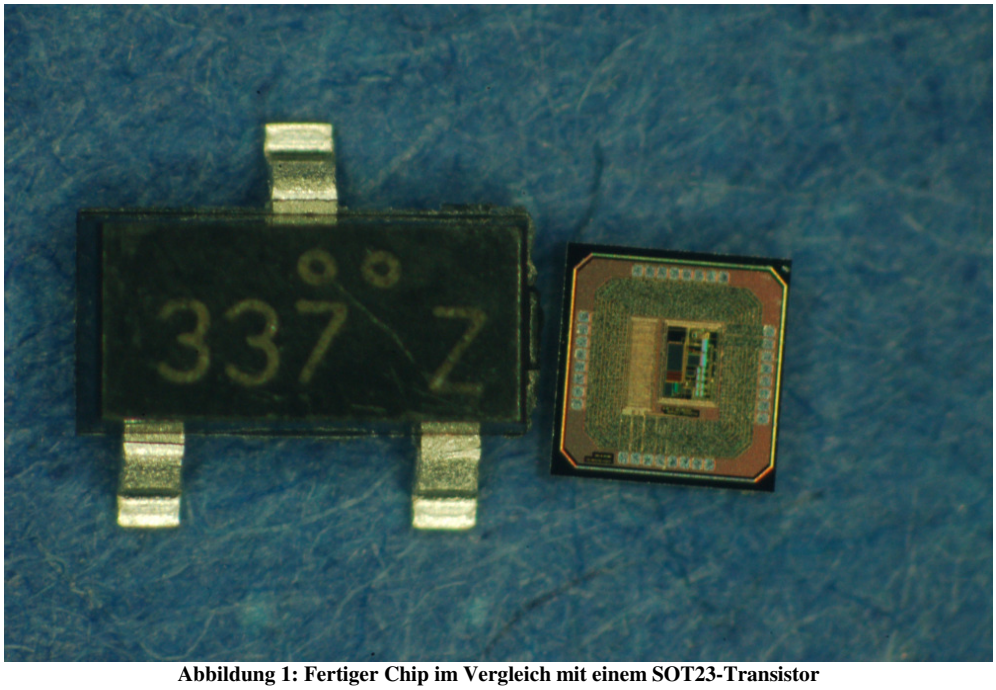
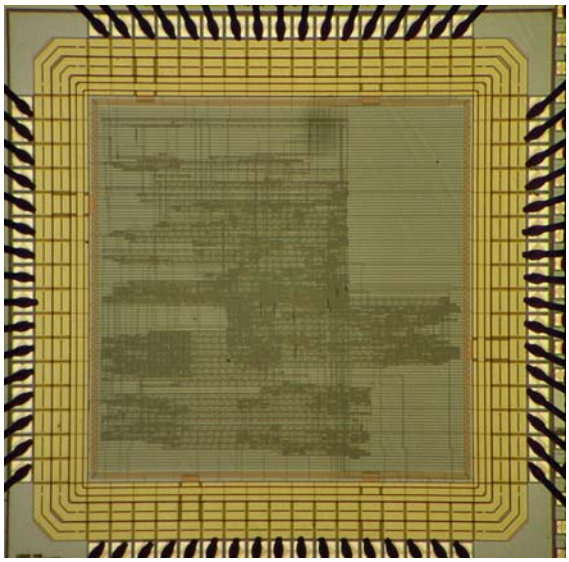


Abbildung 1: Fertiger Chip im Vergleich mit einem SOT23-Transistor

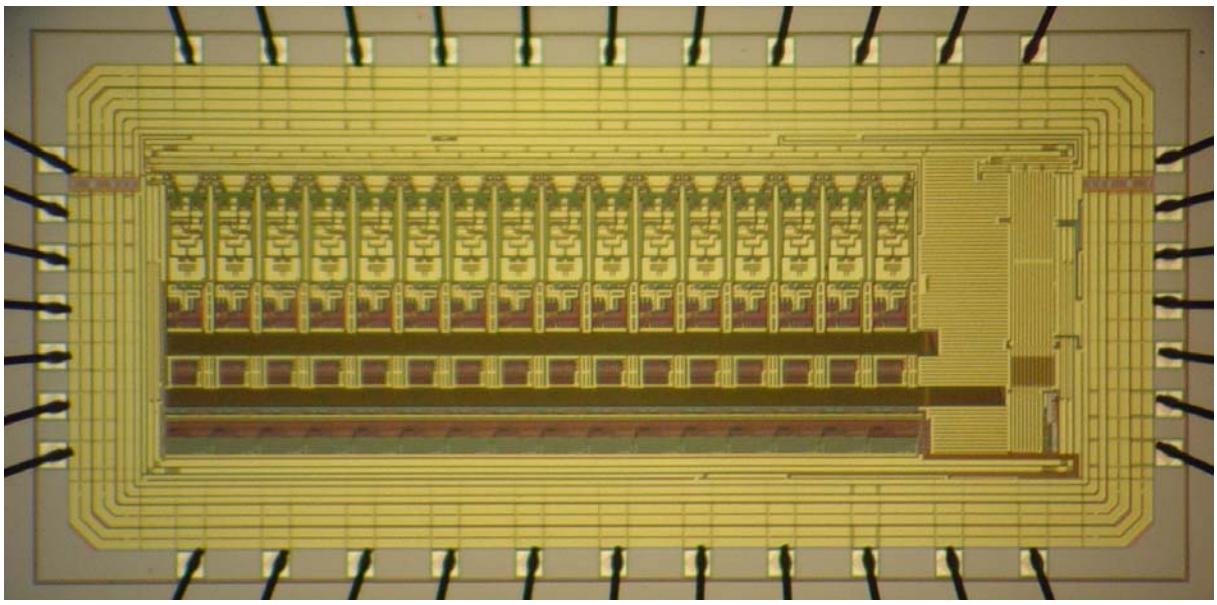
Entwurf:	Hochschule Offenburg Bearbeiter: M.Sc. Ji Li Dipl.-Ing. Tobias Volk Dipl.-Ing. Daniel Bau Betreuer: Prof. Dr.-Ing. Dirk Jansen
Layouterstellung:	Hochschule Offenburg (Standardzellenentwurf & analog Design)
Technologie:	AMIS 0.35 µm CMOS A
Chipfertigung:	Europractice, MPW Run 1657 (mini@sic)
Herstelldatum:	Juli 2007
Kostenträger:	MPC-Mittel HS-Verbund Baden-Württemberg
Chipdaten:	Chipgröße: 1314 µm x 1314 µm Gehäuse: QFN32 (5x5)
Funktion:	RFID Frontend für den ISO-Standard 15693. Mit Hilfe des ASICs kann ein Microcontroller über die RFID-Schnittstelle kommunizieren. Zur Kommunikation ist lediglich ein embedded Prozessor mit 4MHz Taktrate nötig.
Testergebnisse:	Bei der Inbetriebnahme wurde der digitale Schaltungsteil erfolgreich verifiziert. Kleiner Mängel des Chips sind nach ersten Erkenntnissen auf den analogen Schaltungsteil zurückzuführen, was einen Einsatz des ICs nicht ermöglicht.

Testschaltungen zum Praktikum in Schaltungsintegration



Entwurf:	Hochschule Ulm, Institut für Kommunikationstechnik Bearbeiter: Studenten der Fakultät Elektrotechnik und Informationstechnik Betreuer: Prof. Dipl. Ing. Arnold Führer
Layouterstellung:	Bearbeiter: Jürgen Butscher, Bernd Groß Betreuer: Dipl. Ing. (FH) Josef Schäfer
Technologie:	AMS C35B4, 0,35µ CMOS 4 Metal
Chipfertigung:	Fa. AMS, Österreich, über Europractice
Herstellung:	3. Quartal 2007
Kostenträger:	MPC-Gruppe Baden-Württemberg
Chipdaten:	Chipfläche: 2,384 x 2,286 mm^2 Gehäuse: JLCC68 Standardzellen-Entwurf
Funktion:	Studenten des 5. Semesters haben im Rahmen ihres Praktikums zur Schaltungsintegration die drei Schaltungen Schieberegister mit bidirektionalem Ein-Ausgang, Zugangskontrolle für ein Schwimmbad und Fußgängerampelsteuerung in VHDL beschrieben, verifiziert und synthetisiert und auf einem FPGA in Betrieb genommen. Studenten des 7. Semesters haben diese Schaltungen im Rahmen ihrer Studienarbeit zu einer Gesamtschaltung zusammengefasst und als Standardzellenentwurf für die Fertigung bei AMS vorbereitet.

ADC10R0



Entwurf:	Hochschule Ulm, Institut für Kommunikationstechnik Bearbeiter: Florian Mrugalla Betreuer: Prof. Dipl.-Phys. Gerhard Forster		
Layouterstellung:	Hochschule Ulm, Labor Mikroelektronik (Mixed Signal-Entwurf) Analogteil: Full Custom Design Digitalteil: Standardzellen-Entwurf		
Technologie:	C35B4C3 0,35 µm CMOS 4 Metal / 2 Poly / HR		
Chipfertigung:	Fa. AMS, Österreich, über Europractice		
Herstelldatum:	IV. Quartal 2007		
Kostenträger:	MPC-Gruppe Baden-Württemberg		
Chipdaten:	Chipfläche:	3,83 x 1,87 mm^2	
	Gehäuse:	CLCC 44	
	Funktionsblöcke:	Analogteil:	Referenzspannungsquelle, Referenzspannungsteiler, S/H, Vorverstärker, getakteter Komparator, Autozero
		Digitalteil:	Taktgenerator, SAR

Funktion: Der Chip enthält den IP-Core eines 10-Bit-ADCs nach dem Sukzessiv-Approximationsverfahren mit einer aktiven Fläche von 3,45 mm^2. Er besteht aus 16 ADC-Slices, die im Pipeline-Betrieb angesteuert werden, um einerseits eine Abtastrate von 20 MSa/s und andererseits mehrere Kalibierzyklen zu erlauben. Die messtechnischen Untersuchungen sind noch nicht abgeschlossen.

MULTI PROJEKT CHIP GRUPPE

Hochschule Aalen
Prof. Dr. Bartel, (07361) 576-4182
E-Mail: manfred.bartel@htw-aalen.de

Hochschule Albstadt-Sigmaringen
Prof. Dr. Rieger, (07431) 579-124
E-Mail: rieger@hs-albsig.de

Hochschule Esslingen
Prof. Dr. Lindermeir, (0711) 397-4221
E-Mail: walter.lindermeir@hs-esslingen.de

Hochschule Furtwangen
Prof. Dr. Rülling, (07723) 920-2503
E-Mail: rue@hs-furtwangen.de

Hochschule Heilbronn
Prof. Dr. Schröder, (07131) 504-400
E-Mail: jschroeder@hs-heilbronn.de

Hochschule Karlsruhe
Prof. Dr. Koblitz, (0721) 925 –2238
E-Mail: rudolf.koblitz@hs-karlsruhe.de

Hochschule Konstanz
Prof. Dr. Voland, (07531) 206-644
E-Mail: voland@htwg-konstanz.de

Hochschule Mannheim
Prof. Dr. Paul, (0621) 292-6351
E-Mail: g.paul@hs-mannheim.de

Hochschule Offenburg
Prof. Dr. Jansen, (0781) 205-267
E-Mail: d.jansen@fh-offenburg.de

Hochschule Pforzheim
Prof. Dr. Kesel, (07231) 28-6567
E-Mail: frank.kesel@hs-pforzheim.de

Hochschule Ravensburg-Weingarten
Prof. Dr. Ludescher, (0751) 501-9685
E-Mail: ludescher@hs-weingarten.de

Hochschule Reutlingen
Prof. Dr. Kreutzer, (07121) 341-108
E-Mail: hans.kreutzer@fh-reutlingen.de

Hochschule Ulm
Prof. Dipl.-Phys. Forster, (0731) 50-28180
E-Mail: forster@hs-ulm.de

www.mpc.belwue.de