institut de microtechnique
Neuchâtel

Institute of Microtechnology
University of Neuchâtel
Switzerland

Beiträge zu neuen DSP-Architekturen und raschen Entwurfsmethoden im Low-Power-Bereich

Andreas Drollinger

Thèse présentée à la faculté des sciences
de l 'université de Neuchâtel
pour l 'obtention du grade de docteur ès sciences

Juni 2002

institut de microtechnique
Neuchâtel

Institute of Microtechnology
University of Neuchâtel
Switzerland

Beiträge zu neuen DSP-Architekturen und raschen Entwurfsmethoden im Low-Power-Bereich

Andreas Drollinger

Thèse présentée à la faculté des sciences
de l 'université de Neuchâtel
pour l 'obtention du grade de docteur ès sciences

Juni 2002

IMPRIMATUR POUR LA THESE

Beiträge zu neuen DSP-Architekturen und raschen Entwurfsmethoden im Low-Power-Bereich

de M. Andreas Drollinger

UNIVERSITE DE NEUCHATEL

FACULTE DES SCIENCES

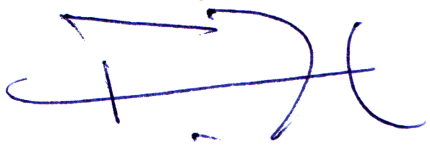
La Faculté des sciences de l'Université de Neuchâtel sur le rapport des membres du jury,

MM. F. Pellandini (directeur de thèse), H. Hügli, P. Balsiger et H. Kaeslin (ETH-Zürich)

autorise l'impression de la présente thèse.

Neuchâtel, le 17 mai 2002

Le doyen:



F. Zwahlen

Zusammenfassung

Dank modernen Entwurfsmethoden können heutzutage in nur einigen Wochen digitale IC mit einer Komplexität von einigen Millionen Transistoren entwickelt werden, wobei die raschen Entwicklungszeiten häufig jedoch unter Einbusse der Designqualität erzielt werden.

Je nach Anwendungsbereich können im Begriff „Designqualität" verschiedene Faktoren berücksichtigt werden wie etwa Stromverbrauch, notwendige Siliziumfläche, maximale Geschwindigkeit, Zuverlässigkeit und Testbarkeit, usw.

Diese Arbeit stellt in den Gebieten DSP-Architekturen (*Digital Signal Processor*), Designmethodik und Automatisierung drei neue Methoden und Lösungsansätze vor, welche versuchen, die Designqualität besser zu berücksichtigen und gleichzeitig der Entwicklungszeit gebührend Rechnung zu tragen.

Der erste Lösungsansatz profitiert von der Einfachheit einer Implementierung mittels eines frei programmierbaren *DSPs*, passt die DSP-Architektur jedoch in einer zusätzlichen Optimierungsphase so an die Zielanwendung an, dass das schlussendlich erhaltene Implementierungsresultat und eine anwendungsspezifische Lösung einen selben Qualitätsgrad besitzen. Dazu verwendet die Designmethode eine DSP-Architektur, welche einerseits programmierbar, aber auch hoch parametrisierbar ist.

Als zweiter Lösungsansatz wird ein neues *high-level* DSP-Synthesetool zur Generierung von anwendungsspezifischen DSP-Architekturen präsentiert, welche den Anforderungen von *Lowest-Power*-Applikationen Rechnung tragen kann. Es bildet einen einfachen und schnellen Designweg von einer Gleichungsbeschreibung eines Filters hin zu einer synthetisierbaren Architekturbeschreibung. Das Tool zeichnet sich aus durch Einfachheit, Schnelligkeit, Flexibilität, Sicherheit und Qualität des erzeugten Designs.

So wichtig wie die Implementierungsmethode ist die vorausgehende Problemanalyse. Sie erlaubt es, Vereinfachungen einer Implementierung bereits auf Stufe Algorithmus zu erkennen. Auf dieser Erkenntnis basiert der dritte Lösungsansatz und leitete den Bau einer makroprogrammierbaren DSP-Architektur für die Verarbeitung von regelmässigen Algorithmen. Das Makrocode-Konzept ermöglicht, so von einer Algorithmusregelmässigkeit zu profitieren, dass im Gegensatz zu einer Lösung mit einem frei programmierbaren *DSP* die Verarbeitungsgeschwindigkeit erhöht und die Leistungsaufnahme reduziert wird.

Résumé

Contributions à de nouvelles architectures DSP et aux méthodes de conception rapide dans le domaine des circuits intégrés numériques à faible consommation

Grâce aux méthodologies de conception modernes, il est possible de développer en quelques semaines seulement des circuits intégrés qui ont une complexité de quelques millions de transistors. La diminution du temps de développement se fait cependant souvent au détriment de la qualité de la réalisation.

Selon le domaine d'application, la notion de qualité du design peut comporter différents facteurs comme, par exemple, la consommation d'énergie, la surface en silicium, la vitesse maximale, la fiabilité et la simplicité pour tester le système.

Ce travail présente dans les domaines d'architecture de DSP (*Digital Signal Processor*), de méthodologie de design et d'automatisation trois approches novatrices qui prennent mieux en compte la qualité du design en optimisant également le temps de développement.

La première approche profite de la simplicité d'une implémentation basée sur un processeur programmable en adaptant par contre l'architecture du DSP dans une phase d'optimisation supplémentaire à l'application visée, d'une manière à ce que la solution finale présente le même degré de qualité qu'une solution spécifique. La méthode utilise pour cela une architecture DSP qui est programmable, mais aussi hautement paramétrable.

Un nouveau synthétiseur de DSP à haut niveau pour la génération des architectures DSP adaptés à des applications à faible consommation constitue la deuxième approche. L'outil garantit un chemin facile et rapide de la spécification d'un filtre via des équations jusqu'à la description architecturale synthétisable de l'application. Simplicité, rapidité, sûreté et qualité des architectures générée sont ses attributs principaux.

L'analyse d'un problème est aussi importante que la méthode d'implantation. Elle permet de voir de possibles simplifications de l'implantation déjà au niveau de l'algorithme. Cette constatation est à la base de la troisième approche et conduit à la définition d'une architecture DSP macro-programmable. Le concept de macro-programmation permet de profiter d'une régularité d'un algorithme de façon à ce que la vitesse soit augmentée et la consommation réduite par rapport à une solution utilisant un *DSP* programmable.

Inhaltsverzeichnis

Zusammenfassung ..... **v**

Résumé ..... **vii**

Inhaltsverzeichnis ..... **ix**

1 Einführung **1**

1.1 Das digitale IC-Design 2

1.2 Die Qualität eines Designs und einer Designmethode 3

1.3 Ein digitaler Hörcomputer an der Grenze der Realisierbarkeit zeigt die Notwendigkeit neuer, spezifischer Entwurfsmethoden 4

1.4 Ein Wort zu dieser Arbeit 5

2 ICs für digitale Signalverarbeitung im Low-Power-Bereich: Entwurf, Techniken, Architekturen und Hintergründe **7**

2.1 Die verschiedenen Implementierungsmethoden und ihre Vor- und Nachteile 8

2.1.1 Überblick *8*

*2.1.2 Der frei programmierbare Allzweck-DSP (*General Purpose DSP*)* *10*

2.1.3 Hardwarebeschreibungssprachen (HDL) *12*

VHDL 13

Verilog 15

PLD-Beschreibungssprachen 16

2.2 Betrachtungen zum Stromverbrauch und zur Verlustleistung 17

2.2.1 Grundlagen *18*

Die dynamische Leistung 19

Kurzschlussstrom 20

Leckstrom 20

Verarbeitungsdurchsatz und Energie zur Durchführung einer Aufgabe 21

Energieeffizienz 22

2.2.2 Die Reduzierung des Stromverbrauchs und der Verlustleistung *22*

2.3 Die verschiedenen Architekturtypen von Prozessoren 24

Generell verwendbare Architekturen und Spezialarchitekturen 24

Klassifizierung gemäss Speicherorganisation 24

CISC-RISC 25

Systeme für die Parallelverarbeitung von Daten 27

2.4 Die Evolution der ASIC-Technologien und der Schaltungs-Entwurfsmethoden 29

2.4.1 Designmethoden für den Lowest-Power-Bereich zum heutigen Zeitpunkt *30*

2.5 Schlussfolgerung 31

Referenzen 32

3 Eine Hardware/Software – Co-Design-Methodik zur Realisierung von Filter- und Kontrollapplikationen 33

3.1 Wiederverwendbarkeit von DSP-Architekturen durch Architekturenkonfigurierung 34

3.2 Die Kombination zweier Implementierungsmethoden zu einer neuen Hardware/Software – Co-Designmethode 35

3.3 Hardware- und Softwarevoraussetzungen 37

3.4 Der Designfluss der Designmethode 38

3.5 Der HotDSP: Eine low-power DSP-Architektur für die Hardware/Software - Co-Designmethode 40

3.5.1 Hardware 41

Klassifizierung der HotDSP-Architektur 41

Detaillierte Beschreibung der HotDSP-Architektur 42

Instruktionen, Verarbeitungssequenz und Pipeline 47

Die Parameter des HotDSPs 49

Low-Power-Techniken 50

Spezielle Eigenschaften des HotDSPs, welche eine effiziente Filterimplementierung erlauben 51

3.5.2 Software 51

Der Assembler 52

Der Debugger: 53

3.6 Anwendungsbeispiel und Resultate 55

Implementierung eines Stereo-Interpolationsfilter unter Verwendung des ersten HotDSP-Typen 55

Implementierung eines Stereo-Dezimationsfilters unter Verwendung des zweiten HotDSP-Typen 57

3.7 Schlussfolgerung 60

Referenzen 62

4 Ein high-level DSP-Synthesetool für schnelle Implementierungen von DSP-Algorithmen 65

4.1 Die Generierung von Filter- und Datenpfad-Architekturen mittels eines *high-level* Synthesetools 66

4.1.1 Der Filter-Designfluss 66

4.1.2 Existierende Tools für die Konzeption und die Generation von DSP-Architekturen 67

Frontier Design 67

Behavioral Compiler ™ von Synopsys 68

IMTs DSP-Synthesetool 69

4.1.3 Weshalb ein neues high-level DSP-Synthesetool für den Low-Power-Bereich? 69

4.2 Anforderungsprofil eines effizienten *high-level* Synthesetools für den *Lowest-Power*-Anwendungsbereich 70

4.3 Die Realisation des DSP-Synthesetools 72

4.3.1 Der Designfluss 72

4.3.2 Architektur der generierten DSPs und dessen Klassifizierung 74
4.3.3 Low-Power-*Techniken* *74*
4.3.4 Die Implementierung des Synthesetools mit all seinen Modulen 75
Parser 76
Merger (Resource allocation) 78
Scheduler 79
Binder (resource assignment oder resource binding) 80
C-Generator 80
Quantizer 81
Power Manager 81
VHDL-Generator 81
4.3.5 Der verwendete Algorithmus für die automatische Festlegung der Datenwortbreiten 82
Festlegung der Integer Bits 83
Festlegung der Gesamtdatenwortweiten 84
4.3.6 Das Anwenderinterface mit den Anwendungsapplikationen 91

4.4 Anwendungsbeispiel und Resultate 95

4.5 Schlussfolgerung 96

Referenzen 98

5 Eine makroprogrammierbare, power- und recheneffiziente DSP-Architektur mit spezieller Eignung für FFT-basierte Algorithmen 101

5.1 Prozessoren für FFTs 102

5.2 Die Ineffizienz von frei programmierbaren Allzweck-DSPs bei der Implementierung von FFT-basierten Algorithmen 103

5.3 Implementierungsvereinfachende Strukturierung von FFT-basierten Algorithmen 104
5.3.1 Die gut organisierte Struktur von FFT-basierten Algorithmen 104
5.3.2 Ein Zeit-Frequenz-Zeit Spektralanalyzer 105
5.3.3 Die Strukturierung des Dataflussgraphes 106
5.3.4 Vorteile und Erkenntnisse einer solchen Strukturierung 107

5.4 Eine DSP-Architektur, welche speziell geeignet ist für FFT-basierte Algorithmen 108
5.4.1 Pflichtenheft einer für die Verarbeitung von FFT-Algorithmen optimalen DSP-Architektur 108
5.4.2 Der Bau der DSP-Architektur 109
Die Grundstruktur des DSPs 109
Der Programmcode 111
Der Kontroller: Arbeitsweise und Architektur 112
Die funktionalen Einheiten 115
Der Datenpfad 115
Adressgeneratoren 116
5.4.3 Die endgültige DSP-Architektur 117
Klassifizierung der DSP-Architektur 118
Low-Power-Techniken 119

5.5 Anwendungsbeispiel der makroprogrammierbaren DSP-Architektur 120
Die Implementierung eines WOLA-Prozessors für Hörgeräte 120
Block-Gleitkommaarithmetik (Block floating point arithmetic)..... 122
Resultate 123
5.6 Schlussfolgerung 125
Referenzen 126

6 Ein digitales Hörgerät als Anwendungsbeispiel für die drei Lösungsansätze 129
6.1 Entwicklung eines Digitalteils für Hörgeräte 131
6.2 Das DSP-Koprozessor-Konzept 133
6.3 Theoretische Aspekte der Audioverarbeitung 135
Der WOLA-Algorithmus..... 135
Die Vor- und Nachfilter..... 137
6.4 Schlussfolgerung 140
Referenzen 140

7 Die Implementierung und Integration des Koprozessors, Resultate..... 143
7.1 Die Implementierung des digitalen Verarbeitungsteils für Hörgeräte und im Speziellen diejenige des DSP-Koprozessors 144
Das IOP 145
Der WOLA-Prozessor 146
7.2 FPGA- und ASIC-Integrationen..... 147
Synthese- und Backend-Flow..... 149
7.3 Resultate 154
Audioqualität 154
Resultate der ASIC-Integrationen 157
7.4 Schlussfolgerungen..... 158
Referenzen 159

8 Schlussfolgerung..... 161

Dank..... 165

Anhang I (Listings und Ergänzungen)..... A
Parameter des HotDSPs (Ergänzung zum Kapitel 3) .....A
Pseudo-Code einiger Routinen für die Festlegung der Gesamtdatenwortbreiten innerhalb des high-level Synthesetools (Ergänz. zum Kapitel 4): .....J
Datenpfad-Beschreibung für FFT-Prozessor (Ergänz. zum Kapitel 4): L
VHDL-Verhaltensbeschreibung der Kontrolleinheit der makroprogrammierbaren DSP-Architektur (Ergänz. zum Kapitel 5): ..... P

Qualitätsvergleich (Kapitelübergreifende Ergänzung) ..... R
Die Art des Vergleichs ..... *R*
Vergleichsbedingungen ..... *R*
Die drei Implementierungen ..... *S*
Resultate ..... *U*
Diskussion ..... *W*

Anhang II (Glossar und Verzeichnisse) ..... a

Glossar ..... a

Abkürzungen ..... e

Abbildungsverzeichnis ..... f

Tabellenverzeichnis ..... g

Listingverzeichnis ..... g

Curriculum vitae i

Verzeichnis der Veröffentlichungen ..... k
Veröffentlichungen zur Hardware/Software – Co-Design – Methodik: ..... *k*
Veröffentlichungen zum high-level DSP-Synthesetool: ..... *k*
Veröffentlichungen zur makroprogrammierbaren DSP-Architektur: ..... *k*

1 Einführung

Dank der seit einigen Jahrzehnten anhaltenden Fortschritte im Bereich der Mikroelektronik können heutzutage digitale IC mit einigen Millionen Transistoren realisiert werden, und das in nur einigen Wochen. Diese schnellen Entwicklungszeiten können u.a. modernen Methoden für den Architekturentwurf verdankt werden, welche einen hohen Automatisierungsgrad zulassen, einfach und flexibel anzuwenden sind und die Designentwicklung sicher gestalten. Sie garantieren den raschen Erfolg beim Entwurf der meisten Anwendungen.
Die Vorteile dieser Methoden werden häufig auf Kosten der erzielten Designqualität erzielt. Unter Qualität kann dabei verschiedenes verstanden werden wie etwa Stromverbrauch, notwendige Siliziumfläche, maximale Geschwindigkeit, Zuverlässigkeit, Testbarkeit, usw. Die erwähnte Qualitätseinbusse zugunsten der Entwicklungszeit ist meist gering und für die meisten Anwendungen unwesentlich. Sie wird aber zu einem echten Problem, wenn man sich an der Grenze des Machbaren befindet.
Diese Grenze wurde bei der Entwicklung einer digitalen Verarbeitungseinheit für Hörgeräte erreicht. Um die mit der Anwendung im Zusammenhang stehenden Designrandbedingungen wie Grösse und Stromverbrauch erfüllen zu können, musste das Design mit besonderer Sorgfalt realisiert werden. Unter anderem mussten spezielle Lösungsansätze und Methoden erarbeitet werden. Diese Arbeit stellt im Wesentlichen drei der verwendeten Ansätze vor, welche in den Gebieten DSP-Architekturen, Designmethodik und Automatisierung neue, innovative Lösungen bilden.

Dieses Kapitel beinhaltet folgende Teile: Es beginnt mit einigen Betrachtungen zur Halbleitertechnik, zum digitalen IC-Design und besonders zu den Entwurfsmethoden fürs IC-Design. Es folgt dann eine kurze Abhandlung über Qualitätskriterien von Designs und von Designmethoden. Der erwähnte digitale Hörcomputer wird anschliessend kurz beschrieben, bevor dieses Einführungskapitel mit einer Vorschau auf die weiteren Kapitel abgeschlossen wird.

1.1 Das digitale IC-Design

Vier Jahrzehnte sind vergangen, seit mit riesigem Aufwand die ersten integrierten Schaltungen realisiert wurden, welche auf einem einzigen Siliziumsubstrat einige Transistoren, Dioden und Widerstände zu einer einfachen, elektronischen Schaltung vereinigten. Die Halbleitertechnologien und die Entwurfsmethoden für das IC-Design haben sich seither stark entwickelt und erlauben heutzutage, dass digitale IC mit einer Komplexität von einigen Dutzend von Millionen Transistoren in nur ein paar Wochen entworfen und produziert werden können.

In einer Halbleitertechnologie vom Anfang der 70er Jahre integriert, würde eine digitale Schaltung mit 20 Millionen Transistoren eine Fläche von der Ordnung eines Quadratmeters benötigen, eine Leistung von mehreren Megawatts konsumieren und hätte eine maximale Arbeitsgeschwindigkeit, welche um den Faktor 1000 unter den heutigen üblichen Produkten liegt. Der technologische Fortschritt wurde mittels Verbesserungen in verschiedenen Bereichen erzielt: Dank Fortschritten im Herstellungsprozess konnten die Layoutdimensionen von integrierten Schaltungen reduziert werden, was gleichzeitig Fläche, Stromverbrauch und Geschwindigkeit optimierte und zudem die notwendige Speisespannung reduzierte. Die Verwendung von CMOS-Gatter anstelle von NMOS-Gatter oder sogar Bipolar-Transistor-basierten Schaltelementen führte zu einem wichtigen Rückgang des Stromverbrauches.

Auch der Entwurfsprozess eines Produktes konnte mit der Entwicklung von Entwurfskonzepten und –methoden verkürzt und sicherer gemacht werden. Moderne Konzepte und Automatisierungstools erlauben einen hierarchischen IC-Entwurf auf einem funktionellen Abstraktionsniveau, was dazu führt, dass besonders das digitale IC-Design heutzutage eher mit Informatik als mit Elektronik zu tun hat. Das digitale IC-Design mittels hierarchischen, funktionaler Schaltungsbeschreibungen ist heute der einzig mögliche Weg, eine Schaltung mit mehreren Millionen Transistoren zu entwerfen, da es nie möglich wäre, eine Schaltung dieses Umfanges von Hand im konventionellen Layoutdesign zu realisieren.

Der Entwicklungsprozess für digitale ICs wird heutzutage in das sogenannte Frontend-Design und das Backend-Design aufgeteilt[1]. Das Frontend-Design beginnt mit dem Architekturentwurf, wo die Grundorganisation und die Funktionsweise des Digitalchips mittels Schemen oder Textbeschreibungen spezifiziert wird (*Implementierung*). Automatische Tools generieren sodann in Funktion dieser Spezifikationen eine elektronische Schaltung, die aus

[1] Diese Beschreibung des Front- und Backends ist stark schematisiert und vereinfacht

einfachen, logischen (digitalen) Standardzellen aufgebaut ist (*Synthese*). Im Backend wird dann der physikalische Chip gestaltet. Zuerst wird die zur Verfügung stehende Siliziumfläche global strukturiert und die *Pads* und sogenannte Makrozellen[1] auf der Fläche platziert. Die Standardzellen werden dann vom *Placer* automatisch platziert. Schlussendlich realisiert der *Router* die Verbindungen zwischen *Pads*, Makrozellen und Standardzellen.

Um der steigende Komplexität digitaler Schaltungen und dem Verlangen nach kürzerer Entwurfszeiten nachzukommen, werden Entwurfsmethoden entwickelt, welche Schaltungsbeschreibungen auf immer höheren Niveaus zulassen. Diese Entwurfsmethoden genügen sicherlich den Ansprüchen der meisten Standardanwendungen, können jedoch bei Anwendungen mit besonders kritischen Designrandbedingungen die Designanforderungen nicht erfüllen, da der Gewinn an Entwurfszeit häufig auf Kosten der Designqualität erzielt wird. Dieser Verlust kann jedoch durch den Fortschritt in der Halbleitertechnologie kompensiert werden.

1.2 Die Qualität eines Designs und einer Designmethode

Das Wort „Designqualität“ kann sich je nach Anforderung auf verschiedene Kriterien eines Designs beziehen. Komplexitätsbedingte Siliziumfläche, Stromverbrauch und Verlustleistung, maximale Geschwindigkeit, Zuverlässigkeit und Testbarkeit sind einige allgemeine Qualitätsfaktoren eines Digitalchips. Um eine hohe Designqualität zu erzielen, ist es unerlässlich, dass alle Etappen des Entwicklungsprozesses mit höchster Sorgfalt durchgeführt werden. So kann eine bestmögliche Platzierung der Standardzellen während des *Backends* die benötigte Siliziumfläche, die Verlustleistung und die Geschwindigkeit optimieren. Diese Qualitätskriterien können aber bereits im Frontend beeinflusst werden, indem während des Architekturentwurfs nach effizienten Schaltungsstrukturen und Prozessorarchitekturen gesucht wird und das Design sorgfältig, mit den richtigen Randbedingungen (*Constraints*) synthetisiert und optimiert wird. Qualitätsfaktoren sollten jedoch bereits von Beginn einer Produktentwicklung berücksichtigt werden. So können oftmals auf System- und Algorithmuslevel Vereinfachungen gefunden werden, welche den Hardwareaufwand stark reduzieren.

Die Effizienz der Designoptimierungsmethoden nimmt häufig im Verlauf des Designflusses ab. So können nicht durchgeführte Vereinfachungen auf Algorithmuslevel während des *Backends* niemals mehr kompensiert werden. Auch während des Architekturentwurfs können viele Qualitätsfaktoren so stark kontrolliert werden, wie dies im Backend nicht mehr möglich ist.

[1] Grosse Zellen, welche als Einheit bearbeitet werden wie RAMs

Selbst Kriterien, welche normalerweise nur im Bezug mit dem *Backend* stehen, können oftmals bereits in dem Architekturentwurf beeinflusst werden. So kann die Strukturierung und die Synchronisierung eines Designs während des Entwurfs derart vorbereitet werden, dass das *Backend* viel einfacher durchgeführt werden kann. Es darf sicherlich gesagt werden, dass der Architekturentwurf die wichtigste und auf die Designqualität einflussreichste aller Etappen des Designflusses ist.

In dem Begriff „Qualität“ oder „Effizienz“ einer Designmethode wird hauptsächlich die mit der Methode zu erzielende Designqualität und die Produktivität oder Entwicklungszeit berücksichtigt. Aber auch Anwendungsfreundlichkeit, Sicherheit, Flexibilität im Gebrauch, Überprüfungsmöglichkeiten der Lösungen und die Kompatibilität mit dem verwendeten Entwurfskonzept sind Faktoren, welche im engen Zusammenhang mit der Effizienz einer Designmethode stehen. Die Wahl einer Designmethode hängt schlussendlich auch noch von den persönlichen Präferenzen und den mit der Methode im Zusammenhang stehenden Kosten ab.

Wird eine Designmethode qualitativ eingeschätzt, so muss gleichzeitig immer der Anwendungsbereich erwähnt werden. Denn die Effizienz einer Designmethode hängt grundsätzlich von den Anwendungsbedingungen ab. Für jede Methode können Vor- und Nachteile gefunden werden. Es kann nicht einmal gesagt werden, dass eine Methode für ein ganzes Design die beste ist. Selbst innerhalb eines Designs kann es vorteilhaft sein, für die verschiedenen Blöcke unterschiedliche Designmethoden zu verwenden. Zudem muss man sich auch bewusst sein, dass eine Designmethode im Verlaufe der Zeit ihre Vorteile verlieren kann, da auf Grund des technologischen Fortschrittes auch die Anforderungen ändern, welche an Designmethoden gestellt werden.

1.3 Ein digitaler Hörcomputer an der Grenze der Realisierbarkeit zeigt die Notwendigkeit neuer, spezifischer Entwurfsmethoden

Die marktgängigen Entwurfsmethoden für das digitale IC-Design sind für eine möglichst schnelle Entwurfszeit entwickelt worden, produzieren ohne grossen Aufwand schnell überzeugende Resultate und lassen sich meist in ein globales Entwicklungskonzept einbinden. Erst wenn man an die Grenze des Machbaren kommt, muss die Art und Weise, wie ein Design realisiert werden soll, sorgfältiger evaluiert werden. Die Wahl der Methoden und Lösungsansätze entscheiden darüber, ob sich ein Produkt schlussendlich realisieren lässt.

Im Falle der Entwicklung einer digitalen Verarbeitungseinheit für Hörgeräte wurde die Machbarkeitsgrenze erreicht. Die Recheneinheit des Digitalteils benötigt zur Verarbeitung der Audiodaten eine Rechenleistung von ungefähr 20-30 RISC[1]-MIPS[2,3]. Ein Prozessor mit dieser Leistung wäre mit Leichtigkeit zu realisieren, wenn nicht noch anwendungsbedingte Randbedingungen bezüglich maximaler Verlustleistung, maximaler Siliziumfläche, hoher Anwendungsflexibilität und der zur Verfügung stehenden Entwicklungszeit gewesen wären.

Auf Systemniveau wurde nach einer flexiblen und doch stromsparenden Lösung gesucht, und in der Kombination eines Allzweck-DSPs[4] zusammen mit einem Koprozessor gefunden. Der Allzweck-DSP ist frei programmierbar und soll die verlangte Flexibilität garantieren. Der Koprozessor ist hingegen hochoptimiert für die Verarbeitung von spezifischen DSP-Algorithmen. Er nimmt dem Allzweck-DSP gewisse rechenintensive Arbeiten ab, wie zum Beispiel Vor- und Nachfilterungen der Audiosignalen und die Zeit-zu-Frequenz- und Frequenz-zu-Zeit-Transformationen. Da der grösste Teil der Datenverarbeitung vom Koprozessor durchgeführt werden muss, hängt es vor allem von diesem ab, ob die Designrandbedingungen erfüllt werden können.

1.4 Ein Wort zu dieser Arbeit

Nur durch die Wahl von gut adaptierten Lösungsansätzen konnte in der zur Verfügung stehenden Zeit die digitale Verarbeitungseinheit für Hörgeräte in einer Weise realisiert werden, dass sie sämtliche Randbedingungen erfüllte. Diese Arbeit stellt drei der verwendeten Ansätze vor, welche in den Gebieten DSP-Architekturen, Designmethodik und Automatisierung neue, innovative Lösungen bilden. Jede der Ansätze schreitet einen eigenen Weg und hat seine Vor- und Nachteile. Gemeinsam erlauben sie aber im Gegensatz zu anderen, kommerziellen Methoden, Qualitätskriterien eines Designs besser zu gewichten. Sie ergänzen sich und ermöglichen, das Design gemäss den Spezifikationen zu bauen und das in der zur Verfügung stehenden Zeit.

Die vorliegende Arbeit ist in 8 Kapitel gegliedert. Auf diese Einleitung folgt in Kapitel 2 eine Einführung in Entwurfsmethoden,

---

[1] RISC: *Reduced Instruction Set Computer*
[2] MIPS: *Million Instruction Per Second*
[3] „MIPS“ ist eine sehr relative Einheit für die Spezifikation einer Rechenleistung. Es ist besser zu sagen, dass die Recheneinheit etwa um die 7 Millionen arithmetische Operationen pro Sekunde verarbeiten muss.
[4] Im Folgenden wird auch der englische Fachbegriff *General Purpose DSP* verwendet.

Prozessorenarchitekturen und Verlustleistung. Der innovative Teil dieser Arbeit liegt in den Kapiteln 3 bis 5. Jedes dieser Kapitel stellen je eine neue Entwurfsmethode oder einen neuen Lösungsansatz vor, welche, wenn sie richtig angewandt werden, zu einer Effizienzsteigerung des Architekturentwurfs führen können.

Die in Kapitel 3 vorgestellte Designmethode profitiert von der Einfachheit einer Implementierung mittels eines frei programmierbaren DSPs, passt die DSP-Architektur jedoch in einer Optimierungsphase möglichst optimal an eine Zielanwendung an.

Kapitel 4 präsentiert die Studie und Realisation eines neuen *high-level* DSP-Synthesetools zur Generierung von anwendungsspezifischen DSP-Architekturen.

In Kapitel 5 wird ein Makrocode-Konzept präsentiert, welches ermöglicht, so von einer Algorithmusregelmässigkeit zu profitieren, dass im Gegensatz zu einer Lösung mit einem frei programmierbaren DSP die Verarbeitungsgeschwindigkeit erhöht und die Leistungsaufnahme reduziert wird.

Das erwähnte Digitalteil für Hörhilfen wird in Kapitel 6 vorgestellt und dient als Anwendungsbeispiel für die drei Entwurfsmethoden. Verschiedene Implementierungen, ASIC[1]- und ASSP[2]-Integrationen sowie Resultate dieses Digitalteils sind in Kapitel 7 zu finden bevor eine allgemeine Schlussfolgerung im letzten Kapitel den Abschluss dieser Arbeit bildet.

[1] ASIC: *Application Specific Integrated Circuit*
[2] ASSP: *Application Specific Standard Product*

2 ICs für digitale Signalverarbeitung im Low-Power-Bereich: Entwurf, Techniken, Architekturen und Hintergründe

---

Verschiedenste Implementierungsmethoden erlauben unterschiedliche Entwicklungsprioritäten und –ziele wie etwa Entwicklungszeit, Flexibilität und Designqualität zu bevorzugen. Jede der Methoden hat Vor- und Nachteile, welche individuell für jedes Projekt oder sogar jeden Projektteil gewichtet werden müssen, um die für eine Anwendung geeignete Designmethode zu finden.

Nebst einer eingehenden Besprechung solcher Implementierungsmethoden enthält dieses Kapitel auch Einführungen in den Stromverbrauch und die Verlustleistung von ICs und in die Prozessorarchitekturen.

2.1 Die verschiedenen Implementierungsmethoden und ihre Vor- und Nachteile

2.1.1 Überblick

Qualitative Anforderungen an ein Design, Sicherheiten der Entwicklung, Strategien des Management, Produktivität und Entwicklungszeit, aber auch finanzielle Kriterien entscheiden darüber, welche Designmethoden bei der Entwicklung eines ASIC verwendet werden. Abbildung 1 ordnet schematisch verschiedene Methoden, welche bei einer Implementierung von Filter- oder Datenpfadstrukturen verwendet werden können, nach Qualität und resultierender Entwicklungszeit ein [1].

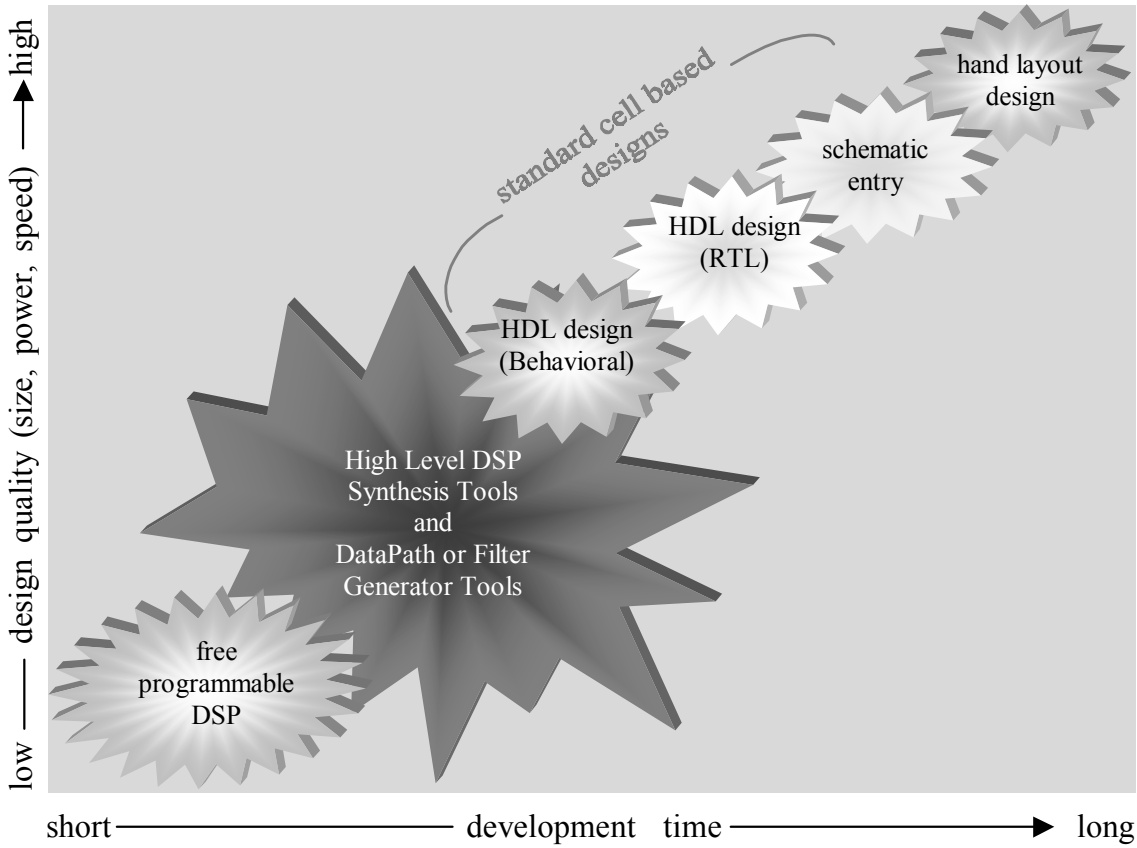


Abbildung 1: Verschiedene Implementierungsmethoden, tendenziell geordnet nach der notwendigen Entwicklungszeit und der zu erwartenden Designqualität

Unter Qualität ist hiermit Leistungsfähigkeit und Effizienz des realisierten Designs gemeint, was im speziellen Fall Fläche, Datenverarbeitungsgeschwindigkeit oder Stromverbrauch bedeuten kann.

Andere, auch wichtige Faktoren sind in der Graphik nicht berücksichtigt. Diese Faktoren können Kosten sein, welche aus der Entwicklungszeit selbst, der Preise der Entwicklungssoftware und IP-Blöcke, Zugriff auf eine

Technologie, usw., resultieren. Weitere Faktoren sind Flexibilität und Sicherheit einer Designmethodik, Verifizierbarkeit eines Designs, usw.

Die detaillierte Form der in Abbildung 1 gezeigten Tendenzkurve hängt von der Anwendung und vom Anwendungsbereich ab. Sie ist monoton ansteigend. Anders ausgedrückt heisst das, dass mit einer ersten Designmethode die Qualität einer zweiten Methode, welche ein höheres Abstraktionsniveau hat, immer erreicht oder übertroffen werden kann, nicht jedoch umgekehrt. Je nach Anwendung ist es so möglich, dass ein frei programmierbarer DSP die selbe Leistung erbringt wie eine HDL-Verhaltensbeschreibung, diejenige zum selben Design führt wie eine HDL-Strukturbeschreibung usw.

Mit einem *Full Custom Layout Design* kann ein Maskenlayout mit der höchstmöglichen Packungsdichte erzielt werden. Demgegenüber stellen sich jedoch Automatisierungsschwierigkeiten und vor allem ein grosser zeitlicher Aufwand. Zudem besitzt die Methode wegen der erschwerten Designautomation auch ein höheres Risiko als andere Designmethoden.

Ein manuelles Zeichnen eines elektrischen Schemas mit Hilfe von sogenannten Standardzellen (digitale Grundschaltungen zur Durchführung logischer Grundoperationen) war noch vor wenigen Jahren die üblichste Designmethodik. Dabei hatte ein Designer seine Schaltung von Hand zu strukturieren und in sequentielle und kombinatorische Logikgruppen zu zerlegen. Die letzteren mussten dann mit Hilfe der klassischen Vereinfachungsmethoden für Logikaufgaben wie zum Beispiel der Karnaugh-Diagramme vereinfacht werden. Schlussendlich konnten die vereinfachten Gleichungen mit kombinatorischen Standardzellen aufgebaut werden. Flipflop-Standardzellen bildeten die sequentiellen Logikgruppen. Automatisierungstools führten dann in einer zweiten Designetappe die Platzierung und Verknüpfung (Routing) der Standardzellen durch, was die Entwicklungszeit stark beschleunigte. Trotzdem war die Tatsache nicht zufriedenstellend, dass eine Bibliothek von Standardzellen immer an eine ASIC-Technologie gebunden ist und somit ein Design bei jeder Integration in einer neuen Technologie ebenfalls neu aufgebaut werden musste.

Dieses Problem wurde gelöst, als die HDL[1]-Beschreibungssprachen wie VHDL[2] und Verilog synthetisierbar wurden. Das Design konnte so in einer technologieunabhängigen Beschreibung spezifiziert werden, welche von einem sogenannten Synthesetool automatisch in einen Standardzellenaufbau

[1] HDL: *Hardware Description Language*
[2] VHDL: *VHSIC Hardware Description Language*
VHSIC: *Very High-Speed Integrated Circuit*

geformt wird. Die Methode einer HDL-Beschreibung ist der heute übliche Weg, einen digitalen ASIC zu entwickeln. Alle Schritte dieser Designmethodik, vom automatischen Designaufbau mittels Standardzellen, über die Platzierung und das Routen, sind heute stark automatisiert und vermeiden dadurch das Risiko eines unachtsamen Einschleichens eines Fehlers. Zudem ist die Kontrolle durch Simulation und Timinganalyse zu jedem Entwicklungszeitpunkt sehr einfach möglich. Die verschiedenen Abstraktionsniveaus der HDL-Beschreibungssprachen ermöglichen ein begrenztes Balancieren zwischen Designqualität und Implementierungszeit.

Ist eine schnelle Entwicklungszeit erwünscht, können frei programmierbare DSP-Kerne in Form von IP-Blöcken in ein Design eingebunden werden. Programmierbare DSP-Kerne bieten einen breiten Anwendungsbereich und eine grössere Flexibilität während des Entwurfs als andere Implementierungsmethoden. Kommerziell verfügbare DSP-Kerne sind bereits auf Silizium erprobt und der ASIC-Entwurf verlagert sich grossteils zur Softwareentwicklung. Eine Implementierung eines Algorithmus kann so via Programmierung des DSPs durchgeführt werden, was normalerweise weniger Zeit braucht als für eine vollständige HDL-Beschreibung notwendig wäre. DSP-Simulatoren und - Emulatoren ermöglichen, dass die Integration und die Softwareentwicklung zeitlich parallel durchgeführt werden können. Und wenn als Programmspeicher anstelle eines ROMs ein EEPROM oder ein Flash-Memory verwendet wird, kann die Software selbst während der Chipproduktion vollendet werden, was die Entwicklungszeit eines Produktes nochmals verkürzt. Da ein frei programmierbarer DSP nicht für eine bestimmte Anwendung optimiert ist, kann die erzielte Designqualität diejenige einer Standardzellen-Lösung nur erreichen, nie aber übertreffen.

Eine letzte Möglichkeit, Filterstrukturen zu realisieren ist die Verwendung von Datenpfad-Generatoren, welche meist eine HDL-Beschreibung des Filters erzeugen. Qualität und Entwicklungszeit sind in diesem Fall abhängig von dem verwendeten Tool. Ist die HDL-Beschreibung einmal erzeugt, wird der bereits oben beschriebene Designfluss angewendet.

Es folgen nun etwas detailliertere Betrachtungen dieser verschiedenen Methoden.

2.1.2 Der frei programmierbare Allzweck-DSP[1] *(*General Purpose DSP*)*

Ein *General Purpose DSP* ist ein frei programmierbarer Mikroprozessor, dessen Architektur für die digitale Signalverarbeitung eines breiten Anwendungsbereiches optimiert ist. Für eine anwendungsspezifische

[1] Im Folgenden soll auch der englische Fachbegriff *General Purpose DSP* verwendet werden

Implementierung kann ein *General Purpose DSP* in Form eines IP-Blockes in ein Design eingefügt werden. DSP-Kerne, zusammen mit entsprechender Entwicklungssoftware, werden dabei entweder direkt von den Halbleiterherstellern angeboten oder können von anderen Firmen bezogen werden. Oftmals wurden sie bereits vielmals verwendet und bieten dadurch eine hohe Sicherheit bei einer Implementierung.

Die Verwendung von DSP-Kernen verlagert die Implementierung hauptsächlich von der Hardware- zur Softwareentwicklung. Ein Algorithmus wird via Programmierung des DSPs implementiert. Hochsprachen-Compiler, DSP-Simulatoren und –Emulatoren ermöglichen eine einfache, schnelle und flexible Implementierung einer Aufgabe. Verhaltensbeschreibungen des DSPs, welche eine schnelle Simulation zulassen und Co-Simulationen erlauben die Verifikation des Gesamtsystems in einer vernünftigen Zeit.

Bei einer Implementierung muss eine Aufgabe mittels eines Ablaufes von sequentiellen Instruktionen beschrieben werden. Je nach Abstraktionsniveau ist dies eine Assembler- oder Hochsprachenbeschreibung. Der Assembler oder der Compiler liest diesen Quellcode und formt ihn in eine Sequenz von Prozessorinstruktionen um. Jede Operation des Algorithmus wird so individuell in den entsprechenden Maschinencode umgesetzt und bildet zusammen mit zusätzlichen Kontrollinstruktionen, welche den richtigen Programmablauf gewährleisten, den Programmcode. Diese individuelle Übersetzung der einzelnen Teile eines Algorithmus ermöglicht es einerseits, dass beinahe jeder beliebige Algorithmus mit einem *General Purpose DSP* implementiert werden kann, hat jedoch auch die Konsequenz und den Nachteil, dass die Grösse des Programmcodes und damit diejenige des benötigten Programmspeichers etwa proportional zu der Grösse des Algorithmus steht. Die Adress- und Kontrollinstruktionen beanspruchen ebenfalls Taktzyklen, was die Datenverarbeitungsgeschwindigkeit des DSPs stark reduziert.

Sogenannte VLIW[1] - DSP-Architekturen versuchen eine Reduktion der Verarbeitungsgeschwindigkeit zu vermeiden indem sie einen sehr breiten Instruktionsbus verwenden. Ein Instruktionswort kann auf diese Weise mehrere arithmetische und logische Operationen mitsamt Parameterinformationen sowie Kontrollinformationen enthalten. Zudem scheint es, dass Hochsprachen-Compiler qualitativ besseren Maschinencode erzeugen können für *VLIW*-DSPs als für klassische DSPs.

[1] VLIW: *Very Long Instruction Word*

2.1.3 Hardwarebeschreibungssprachen (HDL[1])

Seit den frühen 80er-Jahren existieren Hardwarebeschreibungssprachen, mit denen die ersten grossen, digitalen Designs modelliert und simuliert werden konnten. Erst aber seit es Computerprogramme gab, sogenannte Synthesetools, welche solche Designbeschreibungen in eine Hardware umsetzten können, konnten auch digitale Designs von einer Grösse realisiert werden, welche vorher noch utopisch erschien.

Das Synthesetool kann eine Designbeschreibung, welche noch nicht an eine Technologie gebunden ist, lesen, interpretieren, vereinfachen und in die logischen Grundoperationen aufspalten. Zur Hardwareabbildung benötigt das Werkzeug eine Anzahl von Standardzellen, welche in einer Bibliothek zusammengefasst sind. Jede der Zellen kann eine logische Grundoperation durchführen. Mittels Auswahl der richtigen Standardzellen und deren korrekten Verknüpfung ist das Synthesetool imstande, die Designbeschreibung in eine logische Schaltung zu transformieren (Abbildung 2). Der dabei gefundene Schaltungstyp wird Netzliste genannt. Diese Netzliste kann dann an sogenannte Backend-Tools übergeben werden, welche die Standardzellen automatisch platzieren und verbinden.

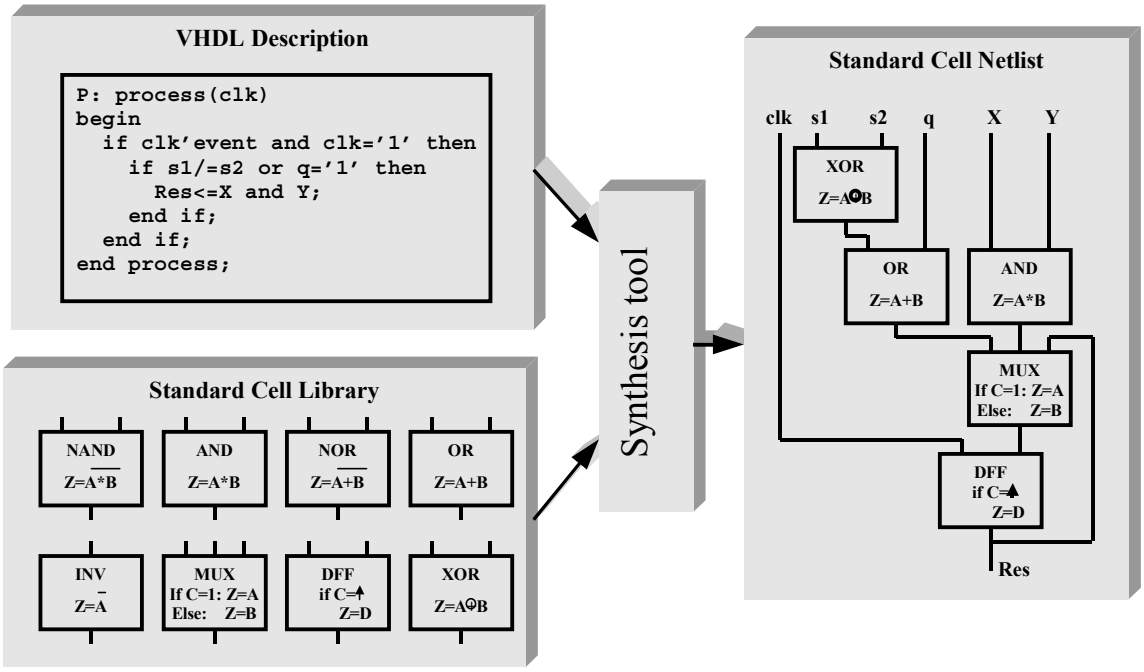


Abbildung 2: Bei der Logiksynthese konstruiert ein Synthesetool mit Hilfe von Standardzellen gemäss der VHDL- oder Verilogbeschreibung ein Design.

Wegen der guten Automatisierungsmöglichkeit und der Sicherheit von dieser Art der Designbeschreibung ist die Methode einer HDL-Beschreibung der heute übliche Weg, einen ASIC zu entwickeln. Auch wenn zur

[1] HDL: *Hardware Description Language*

Generierung einer Hardwarearchitektur zusätzliche Tools verwendet werden, generieren diese meist eine HDL-Beschreibung. Ein Designer von heute kann deshalb HDL-Sprachen nicht umgehen.

Zwei Beschreibungssprachen sind heute von grosser Bedeutung: VHDL und Verilog. Funktionell sind beide Sprachen identisch, unterscheiden sich jedoch in der Syntax und gewissen Möglichkeiten. In Amerika wird meistens Verilog bevorzugt, während der alte Kontinent und Asien eine Vorliebe für VHDL haben. Wegen der Wichtigkeit der beiden Sprachen werden sie im Folgenden kurz vorgestellt[1] [2].

VHDL

VHDL ist eine Programmier- und Beschreibungssprache, welche speziell für die Verhaltensbeschreibung von digitalen Schaltungen und Systemen entworfen und optimiert wurde [3]. Sie vereint Aspekte von Programmiersprachen für Simulationen und Modellierungen, für Designeingabe, für Testzwecke und für Netzlistenbeschreibungen. Es folgt eine kurze Erläuterung dieser Aspekte.

VHDL hat viele Möglichkeiten und Funktionen zur zweckmässigen Verhaltensbeschreibung von elektronischen Komponenten, sei es zur Beschreibung eines einfachen Logikgatters oder aber eines kompletten Mikroprozessors. Dazu besitzt VHDL Beschreibungsformen, um gewisse elektronische Aspekte wie steigende und fallende Flanken von Signalen und zeitliche Verzögerungen zu charakterisieren. Auf diese Weise kann ein exaktes Simulationsmodell von einer digitalen Schaltung gebaut werden.

So wie ein komplexes Funktionieren einer Mikroprozessorenanwendung mit einer Programmierhochsprache definiert werden kann, welche dann von einem Compiler in Instruktionen für den Prozessor umgewandelt werden, so ermöglicht VHDL eine technologienunabhängige Beschreibung eines Designs, welche dann von einem Synthesetool automatisch in eine digitale Schaltung transformiert wird. Wie PASCAL oder C erlaubt VHDL eine strukturierte Programmierung, beinhaltet aber im Gegensatz zu den erstgenannten Sprachen auch das Konzept der zeitlichen Parallelität verschiedener Prozesse.

Was vielleicht als Selbstverständlichkeit betrachtet wird ist die Möglichkeit von VHDL, ein Design zu steuern, zu kontrollieren und sein Verhalten zu erfassen. Dies wird geläufig als *Test Bench* bezeichnet. Ein *Test Bench* ist

---

[1] Teile von dieser Übersicht sind eine freie Übersetzung der von David Pellerin gestalteten WEB-Seite mit dem Titel: „*An Introduction to VHDL for Synthesis and Simulation*“, David Pellerin, President, Accolade Design Automation, Inc., http://www.acc-eda.com/support/aboutvhdl.html

eine VHDL-Beschreibung von Simulationsstimuli und entsprechenden Simulationsresultaten, mit denen das Verhalten des Designs ständig überprüft werden kann. Ein *Test Bench* sollte immer ein integrierter Teil eines VHDL-Projektes sein und sollte parallel mit dem Design kreiert werden.

Nebst Eigenschaften zur Verhaltensbeschreibung auf hohem Abstraktionsniveau besitzt VHDL auch Möglichkeiten, ein Design aus verschiedenen Blöcken strukturell aufzubauen und die Kommunikation zwischen verschiedenen Blöcken zu definieren.

VHDL wurde in den frühen 80er-Jahren von einem Forschungsteam entwickelt, welche zum *High-Speed Integrated Circuit Projekt* (VHSIC-Projekt) gehörte, welches damals vom amerikanischen Verteidigungsministerium gegründet wurde. Während des VHSIC-Projekts waren Forscher mit der für die damalige Zeit immensen Grössen von digitalen Designs konfrontiert, welche mit den zur Verfügung stehenden Entwicklungstools nicht gemeistert werden konnten. Es stellte sich schnell heraus, dass die Designaufgaben nur mit strukturierten Methoden und neuen Tools zu lösen sind.

Zusammen mit Ingenieuren von IBM, Texas Instruments und Intermetrics komplettierte das VHSIC-Team die Spezifikationen für eine neue, sprachbasierte Designbeschreibungsmethode. Die erste Publikation von VHDL (Version 7.2) erschien 1985.

Nach substanziellen Verbesserungen und Modifikationen schlug 1986 unter dem IEEE ein Team bestehend aus Repräsentanten der Wirtschaft, Regierung und Hochschulen eine Standardisierung von VHDL vor. Der erarbeitete Standard ist als IEEE 1076-1987 bekannt und dient als Basis für alle kommerziellen Simulations- und Synthesetools von heute. 1994 wurde eine Verbesserung des Standards unter der Bezeichnung IEEE 1076-1993 freigegeben, welche nun ebenfalls von allen wichtigsten Herstellern von VHDL-Softwaretools unterstützt wird.

Selbst die standardisierte VHDL-Sprache hatte Aspekte, welche zu Kompatibilitätsschwierigkeiten zwischen den Tools der verschiedenen Herstellern führte. Die Probleme kamen davon, dass VHDL selbst zwar viele Daten- und Signaltypen definiert, diese aber nicht alle wichtigen Signalzustände wie Kurzschluss, „undefinierter Zustand“, hohe Impedanz, usw., darstellen können. Hersteller von Simulatoren begannen also, eigene Typen zu definieren (VHDL erlaubt das), welche die Beschreibungsmöglichkeiten erhöhten, jedoch nicht mehr Standard waren. Ein mit einem ersten Simulator entwickeltes Design war häufig inkompatibel mit anderen Simulationsumgebungen oder konnte von den

Synthesetools nicht mehr verarbeitet werden. VHDL verkam schnell zum Nichtstandard.

Ein weiterer Standard wurde deshalb für die Datentypen von VHDL entwickelt und von IEEE übernommen. Dieser Standard ist bekannt unter der Nummer 1164 und definiert mittels einem Standardpaket[1] einen 9-wertigen Datentyp. Dieser Standardtyp wird *std_logic* genannt und das IEEE-1164-Packet wird oft als *standard logic package*, oder auch MVL9[2] bezeichnet.

Die VITAL[3]-Initiative war eine weitere Bemühung, die Möglichkeiten von VHDL zu erweitern um neuen Ansprüchen gerecht zu werden. VITAL ist eine Methode, um in eine bestehende VHDL-Netzliste zusätzliche Zeitinformationen einzufügen oder die existierenden zu modifizieren. Die Methode ist an Techniken von Verilog angelehnt und heute als Standard 1076.4 bekannt.

Verilog

Wie VHDL gibt es seit vielen Jahren als Hardwarebeschreibungssprache auch Verilog HDL, welches von vielen Anhängern für Simulation und Synthese verwendet wird [4].

Verilog HDL wurde Mitte der 80er Jahre von *Gateway Design Automation* eingeführt und etablierte sich als Standard für die Simulation mit ASIC-Bibliotheken. Durch seine hohe Popularität zu dieser Zeit unterstützten Toolhersteller für den ASIC-Bereich nur Verilog und liessen VHDL auf der Seite liegen. Mit der VITAL-Initiative erhielt VHDL eine sehr ähnliche Methode zur Zeitrückführung (*delay* oder *timing (back-) annotation*) wie sie von Verilog benützt wird. Durch VITAL wird der Aufwand der Entwickler von Standardzellenbibliotheken stark vereinfacht, da existierende Verilog-Bibliotheken direkt zur Generierung von VHDL-Modellen verwendet werden können.

Im Gegensatz zu VHDL besitzt Verilog eine Schnittstelle zu anderen Programmiersprachen (PLI[4]). Mit dem PLI können schnelle Simulationsmodelle und Funktionserweiterungen in C geschrieben werden, welche dann in die Verilog-Umgebung importiert werden.

Zwei Vorteile von VHDL bezüglich Verilog sind die IEEE-Standardisierung und bessere Möglichkeiten zur Designverwaltung. Mögliche Deklarationen

[1] Ein Paket ist im VHDL-Jargon eine Bibliothek
[2] MVL9: *MutiValued Logic, 9 values*
[3] VITAL: *VHDL Initiative Toward Asic Libraries*
[4] PLI: *Programming Language Interface*

von Architekturen und Konfigurationen, was Verilog nicht kennt, machen VHDL zur idealen Beschreibungssprache für grosse Projekte.

Nichtsdestotrotz sind VHDL und Verilog funktionell identisch. Sicherlich, die Syntax ist verschieden. Verilog hat eher Ähnlichkeiten mit C und VHDL eher mit Pascal. Die Grundkonzepte sind jedoch die selben. Beide Sprachen sind einfach zu lernen, aber schwierig zu beherrschen. Ist eine der beiden Sprachen gelernt, so kann problemlos auch die zweite verstanden werden wie in Listing 1 ersichtlich ist.

```
MODULE gClkGen(Clk,ClkOut,En,gClk,gClkEn);
   OUTPUT [1:0] gClk;
   INPUT [1:0] gClkEn;
   INPUT Clk, En;
   OUTPUT ClkOut;

   ASSIGN ClkOut=~Clk;
   ASSIGN gClk[0]=(Clk|~(gClkEn[0]|En));
   ASSIGN gClk[1]=(Clk|~(gClkEn[1]|En));
ENDMODULE
```

```
ENTITY gClkGen IS
     PORT (
          Clk,En,: IN std_logic;
          ClkOut: OUT std_logic;
          gClk: OUT std_logic_vector(1 downto 0);
          gClkEn: IN std_logic_vector(1 downto 0));
END gClkGen;

ARCHITECTURE A OF gClkGen IS
BEGIN
     ClkOut<=not(Clk);
     gClk(0)<=Clk or not(gClkEn(0) or En);
     gClk(1)<=Clk or not(gClkEn(1) or En);
END A;
```

Listing 1: Die Verilog-Beschreibung eines kleinen Blockes (oben links) im Vergleich zu der äquivalenten VHDL-Beschreibung (unten rechts)

PLD-Beschreibungssprachen

Verschiedene PLD-Hersteller haben ihre eigenen Beschreibungssprachen entwickelt. Bekannt zum Beispiel sind PALASM, ABEL oder CUPL. Sie wurden als spezialisierte Sprachen entwickelt um ein relativ kleines Design zu definieren. Sie sind einfacher in den Möglichkeiten als VHDL, welches ursprünglich als überall anwendbare Sprache für Modellierung und Simulation entwickelt wurde.

PLD-Sprachen bieten meistens nur einfache Beschreibungsmöglichkeiten und sind dadurch auch einfach zu erlernen. Sie besitzen häufig

Beschreibungsoperatoren für die Spezifikation von Pins und Ausgangspolarität, welche spezifisch für ein oder eine Familie von PLD und FPGA ist. Da diese Sprachen speziell für die Synthese entwickelt wurden, besitzen sie keine Erweiterungen, welche nicht synthetisierbar sind (zum Beispiel Timing-Spezifikationen).

Vielleicht so wichtig wie eine Beschreibungssprache selbst ist, sind auch die damit verbunden Kosten: Viele der Synthesetools für PLD-Sprachen sind heutzutage gratis oder sehr günstig von den PLD- und FPGA-Herstellern zu kriegen.

2.2 Betrachtungen zum Stromverbrauch und zur Verlustleistung

Die steigende Komplexität der ICs und die gleichzeitig anwachsende Packungsdichte der Halbleitertechnologien verursachen heutzutage häufig Probleme bei der Energieabführung, also bei der Kühlung der ICs.

Der folgende Vergleich illustriert dies sehr schön [5]:

- Eine typische Küchenplatte (ø=18cm, P=1800W) hat eine Leistungsdichte von etwa 7W/cm^2.
- Ein DEC Alpha (Version 2000, 0.18µm-Technologie, 100 Million Transistoren, 350mm^2, V=1.5V, I=67A) hat eine Leistungsdichte von 29.7W/cm^2.

Im gezeigten Beispiel besitzt der Mikroprozessor eine etwa 4 mal höhere Heizdichte als die Küchenplatte. Um die hohe Wärmeleistung von den wenigen Quadratmillimetern Oberfläche des Prozessors abzuführen, werden sehr aufwendige und teure Kühlmechanismen benötigt. Die kühlungsbedingten Kosten (spezielle Chipgehäuse + Kühlaggregate) bilden deshalb einen ersten Grund, weshalb der Energieverbrauch von Mikrochips zu reduzieren ist.

Kapazitätslimitation einer Energiequelle liefern einen zweiten Grund, den Energieverbrauch zu senken. So ist es bei mobilen Geräten häufig auf Grund von gegebenen physikalischen Dimensionen nicht möglich, die Funktionsdauer durch eine Vergrösserung der Energiequelle zu erhöhen, sondern nur durch Senkung der Verlustleistung.

Ausschliesslich dieser zweite Grund liefert im Falle dieser Arbeit die Motivation, nach stromsparenden Lösungen zu suchen. Denn schätzt man die Leistungsdichte des im Einführungskapitel erwähnten, digitalen Verarbeitungsteils für Hörgeräte, so erhält man lediglich einen Wert von etwa 30 µW/cm^2 (A=10 mm^2, V=1.5V, I=200 µA).

Einsparungen beim Energieverbrauch können nur dann effizient erzielt werden, wenn die Ursprünge des Energie- und Stromverbrauches klar bekannt sind. Es folgt deshalb eine kurze, theoretische Einführung bezüglich Stromverbrauch und Verlustleistung bei CMOS-Technologien.

2.2.1 Grundlagen

Die Symbole, welche in den Formeln der folgenden, theoretischen Abhandlung verwendet werden, sind zur Übersicht in folgender Tabelle definiert.

∂	Rampenzeit (input slope delay)
a	Relativer Aktivitätsfaktor eines Systems bez. auf die Taktfrequenz
a_k	Relativer Knotenaktivitätsfaktor bezogen auf die Taktfrequenz,
C	Kapazität
CPO	Anz. Zyklen zur Durchführung einer Operation = *Cycles Per Operation*
f	Taktfrequenz
f_{max}	Maximal mögliche Taktfrequenz einer Schaltung
I	Strom
I_s	*direct short circuit current*
I_{sc}	Kurzschlussstrom
L	L: Länge eines *MOS-Gate*-Transistors
MOPS	= *OPS**10^6 = *Million Operations Per Second*
n	Parameter ohne Dimension, der verschiede physikalische Phänomene zweiter Ordnung berücksichtigt
N_{task_cycle}	Anzahl Takte, um eine Aufgabe durchzuführen
OPS	Anzahl Operation pro Sekunde = *Operations Per Second*
P	Leistung
t_{task}	Zeit um eine Aufgabe durchzuführen
Vdd	Speisespannung
V_T	Schwellspannung, =*Threshold Voltage*
W	Breite des *MOS-Gate*-Transistors
β	Transistorenverstärkungsfaktoren

Tabelle 1: Definition der in den folgenden Betrachtungen verwendeten Symbolen

Die Verlustleistung einer CMOS-basierten Schaltung kann mit folgender Formel beschrieben werden ([1], [5]):

$$\begin{aligned} P &= P_{leakage} + P_{direct\_current} + P_{dynamic} \\ &= Vdd * (I_{leakage} + I_{direct\_current} + I_{dynamic}) \end{aligned} \qquad 1)$$

Wie dabei ersichtlich ist, resultiert die Gesamtleistung aus drei Komponenten, welche durch den Leckstrom, den Kurzschlussstrom und den aktivitätsbedingten Strom entstehen.

Die dynamische Leistung

Bei CMOS-Technologien, welche eine (zweiwertige) Standardlogik verwenden, ziehen die Logikgatter die Spannung der Signalleitungen entweder nach *GND* oder nach der Speisespannung *Vdd*. Die Signalleitungen selbst können als Kapazitäten modelliert werden, wobei sich diese aus Kapazitäten des *Routings* und der *Gates* der MOS-Transistoren zusammensetzen, welche auf den Gatterausgang projiziert werden können. Die dynamische Leistung entspricht demnach der Summe aller Knotenleistungen der Schaltung[1].

$$P_{dynamic} = \sum_k f * a_k * 2 * \frac{1}{2} C_k Vdd^2 = f * Vdd^2 * \sum_k a_k * C_k \qquad 2)$$

Die Knotenleistung kann dabei in Funktion der kapazitiven Knotenenergie und der Knotenaktivität ausgedrückt werden.

Soll ein Design aus Systemniveau betrachtet werden, so ist folgende Formel für die dynamische Verlustleistung etwas zweckmässiger:

$$P_{dynamic} = f * C_{sw} * Vdd^2 = a * f * C * Vdd^2$$

$$a = \frac{C_{sw}}{C},\ C = \sum_k C_k,\ C_{sw} = \sum_k a_k * C_k \qquad 3)$$

Der dabei eingeführte relative Aktivitätsfaktor *a* beschreibt das Verhältnis zwischen der gesamten Totalkapazität und der Summe der aktivierten Knotenkapazitäten einer Schaltung. Wie aus der Formel ersichtlich ist, steht die Verlustleistung in Abhängigkeit zur Taktfrequenz, zum Aktivitätsfaktor, zur totalen Schaltungskapazität und zur verwendeten Speisespannung. Eine Reduzierung der Verlustleistung kann deshalb durch Optimierung einer oder mehrerer dieser vier Faktoren erfolgen.

Die Reduzierung der Speisespannung *Vdd* hat jedoch nicht nur einen Einfluss auf die Verlustleistung, sondern hat auch eine Verzögerung der Schaltgeschwindigkeiten der CMOS-Gatter zur Folge, wie folgende Formel zeigt:

$$delay \propto \frac{1}{f_{\max}} \propto \frac{C_{sw} * Vdd}{\frac{W}{L} * (Vdd - V_T)^{\alpha(L)}} \qquad 4)\ ^{2\ 3}$$

Der α-Faktor hängt dabei von der Transistorengatelänge ab:

---

[1] Knoten=Signalleitung

[2] Zur Vereinfachung der Formel wurde eine P-N-Symetrie angenommen.

[3] Gültigkeitsbereich der Formel: *Strong inversion* – Bereich der Transistoren

L (µm)	**3**	**2**	**1**	**0.5**	**0.35**
α	1.65	1.6 ... 1.65	1.45 ... 1.6	1.3 ... 1.5	1.11 ... 1.4

Tabelle 2: Der α-Faktor in Funktion verschiedener Transistorengatelängen

Für sehr lange Transistorengates (L>>1 µm) nimmt α den Wert 2 an, bei Halbleitertechnologien mit kürzeren Transistorengates strebt der α-Faktor gegen 1 hin. Allgemein kann gesagt werden, dass die Speisespannung in jedem Fall einen starken Einfluss auf die Schaltgeschwindigkeit von CMOS-Gatter hat und sie diese annähernd exponentiell in die Höhe schnellen lässt, je näher sie sich der Schwellspannung annähert. Die Anpassung der Speisespannung *Vdd* hat deshalb immer unter Berücksichtigung der notwendigen Taktfrequenz zu erfolgen. Für eine gegebene Architektur ist *Vdd* dann optimal gewählt, wenn die notwendige Taktfrequenz gerade noch erreicht wird.

Kurzschlussstrom

Während dem Zustandswechsel eines CMOS-Gates ist es möglich, dass gleichzeitig die P- und N-MOS-Transistoren leitend sind und so kurzfristig die Speisung mit dem Grund kurzgeschlossen werden. Der auf diese Art entstandene Strom wird Kurzschlussstrom genannt[1]. Für ein Gatter kann die daraus entstandene Verlustleistung approximativ[2] mit folgender Formel modelliert werden [6]:

$$P_{direct\_current} = I_{sc} * f * Vdd \approx \beta * (Vdd - 2\,V_T)^3 * \vartheta * f \qquad 5)$$

Wie aus der Formel ersichtlich ist, hängt die durch den Kurzschlussstrom entstandene Verlustleistung von der Speisespannung, der Schwellspannung, der Taktfrequenz, der Signalformen und der Transistorencharakteristik ab. Der Kurzschlussstrom kann bereits mit der Wahl einer geeigneten Technologie beeinflusst werden. Während der Designphase muss sodann versucht werden, die Standardzellen so zu einer Schaltung zu vereinen, dass einerseits Logikgatter mit kleinen Transistoren verwendet werden (β) und gleichzeitig die Flanken der Signale so steil wie möglich sind (∂). Diese Optimierungen sollten normalerweise automatisch von den Synthese- und Logikoptimierungstools durchgeführt werden.

Leckstrom

Eine Reduktion der Schwellspannung erhöht nicht nur den Kurzschlussstrom sondern ebenfalls den sogenannten Leckstrom, welcher

[1] [shoot, rush] through current, [overlap, contention, short-circuit, dynamic leakage] current

[2] P-N – Symmetrie ($\beta_N = \beta_P$, $V_{TN} = V_{TP}$), Symmetrie im Signalverlauf,

bedingt ist durch den CMOS-Prozess. Die daraus resultierende Verlustleistung kann für ein MOS-Transistor wie folgt modelliert werden:

$$P_{leakage} = Vdd * I_{leakage} = Vdd * I_S * e^{-\frac{Vdd}{n*U_T}^2} \qquad 6)$$

Diese letzte Komponente der totalen Verlustleistung hängt nicht von der Taktfrequenz der Schaltung ab und ist somit statisch. Besonders bei Systemen mit einer geringen Systemaktivität kann sie einen wichtigen Beitrag zum Gesamtleistungsverbrauch liefern.

Verarbeitungsdurchsatz und Energie zur Durchführung einer Aufgabe

Bei einer Optimierung der Verlustleistung müssen immer die Rechenanforderungen, welche an das IC gestellt werden, mitberücksichtigt werden. Grundsätzlich kann zwischen zwei Anforderungsprofilen unterschieden werden:

1) Die Verarbeitungsgeschwindigkeit ist nicht entscheidend für die Anwendung. In diesem Fall ist meistens auch nicht die Verlustleistung des ICs selbst, sondern die notwendige Energie zur Durchführung der Aufgabe von Wichtigkeit.

   Ausgehend von der Formel 3) kann man die für eine Aufgabe notwendige, dynamische Verlustleistung bestimmen:

$$E_{task,dynamic} = t_{task} * P_{dynamic} = N_{taks\_cycle} * C_{sw} * Vdd^2$$

$$N_{task\_cycle} = N_{operation} * CPO; \quad t_{task} = \frac{N_{taks\_cycle}}{f} \qquad 7)$$

   Wie ersichtlich ist, steht die dynamische Verlustleistung in einem linearen Verhältnis zu der Anzahl der Arbeitszyklen, zu der geschalteten Kapazität und zum Quadrat der Speisespannung, ist jedoch unabhängig von der Taktfrequenz.

2) Der Energieverbrauch zur Durchführung einer Aufgabe muss unter Berücksichtigung eines notwendigen Verarbeitungsdurchsatzes reduziert werden. Mit dem gegebenen Verarbeitungsdurchsatz und der Anzahl der Zyklen für eine Operation kann die notwendige Taktfrequenz für eine Anwendung bestimmt werden:

$$OPS = \frac{N_{Operation}}{\text{second}} = \frac{f}{CPO} \Rightarrow f = CPO * OPS \qquad 8)$$

   In Formel 3) kann die Frequenz in Funktion des Verarbeitungsdurchsatzes ausgedrückt werden. Man erhält:

$$P_{dynamic} = OPS * CPO * C_{sw} * Vdd^2 = OPS * \frac{Energy}{Operation} \qquad 9)$$

Wie die Formel zeigt, kann die Verlustleistung gesenkt werden, indem die Energie pro Taktzyklus (C_{sw}*Vdd^2) sowie die Anzahl der zur Durchführung einer Operation notwendigen Zyklen (*CPO*) reduziert werden.

Energieeffizienz

Leistungs- und Energieeffizienz von Computerarchitekturen werden meist mit Hilfe von zwei Grössen beschrieben. Die erste Grösse wiedergibt die Energie pro Operation und wird verwendet, um einen ausgelasteten Prozessor zu charakterisieren.

$$\begin{aligned} \frac{Energy}{Operation} &= Power * Delay = f * CPO * C_{sw} * Vdd^2 * \frac{1}{f} \\ &= CPO * C_{sw} * Vdd^2 \end{aligned} \qquad 10)$$

Diese erste Grösse wird meist in einer invertierten Form verwendet:

$$\frac{OPS}{Watt} = \frac{1}{CPO * C_{sw} * Vdd^2}; \quad \frac{MOPS}{Watt} = \frac{10^{-6}}{CPO * C_{sw} * Vdd^2} \qquad 11)$$

Diese Grösse ist klar unabhängig von der Frequenz f. Würde die Frequenz f reduziert, so könnte man durch Senkung der Speisespannung *Vdd* einen sehr attraktiven MOPS/Watt –Wert erhalten, wobei dies jedoch auf die Kosten des Verarbeitungsdurchsatzes geht.

2.2.2 Die Reduzierung des Stromverbrauchs und der Verlustleistung

Eine Minimierung des Stromverbrauchs und der Verlustleistung eines digitalen Designs muss bereits auf Stufe Algorithmus beginnen und endet auf Niveau Technologie.

Bei der Wahl und Entwicklung eines Algorithmus muss vor allem darauf geachtet werden, dass der Algorithmus mit einer möglichst kleinen Rechenleistung, geringen Busbreiten und wenig Ressourcen auskommt[1].

Bei der Auswahl der Halbleitertechnologie ist es wichtig, die statische Verlustleistung (sehr wichtig bei RAMs) und die mögliche Verwendung einer tiefen Speisespannung zu prüfen. Zu einer „guten“ Technologie sollten ebenfalls entsprechende RAMs und *Low-Power*-Bibliotheken für Standardzellen und *Pads* zur Verfügung stehen.

Die in Kapitel 2.2.1 hergeleiteten Grundlagen zeigen, wie die Faktoren *Vdd*, C_{sw}, *f* und *CPO* die Energieeffizienz, aber auch den

[1] siehe [1] und [5]

Datenverarbeitungsdurchsatz eines Designs beeinflussen. Es folgt eine Auflistung verschiedenster Techniken, wie man während des Entwurfs einer Schaltung die Energieeffizienz und den Datenverarbeitungsdurchsatz gemeinsam optimieren kann.

Die ersten Punkte zeigen, wie auf Systemniveau Optimierungen vorgenommen werden können.

- Das digitale Design soll für die tiefst mögliche Speisespannung konzipiert werden, wobei der durch die tiefe Speisespannung entstandene Geschwindigkeitsverlust mit einer effizienteren, parallelen Architektur und einer schnelleren Logik kompensiert werden kann. Die durch die Parallelisierung entstandene Vergrösserung des Designs sowie die dadurch entstandene Erhöhung des Leckstroms muss natürlich bei der Konzeption des Designs ebenfalls berücksichtigt werden.
- Für das Design oder für einige seiner Unterblöcke kann die Speisespannung sowie die Taktfrequenz dynamisch der jeweiligen Situation angepasst werden, was jedoch die Designkomplexität erhöht.
- Inaktive Blöcke können entweder nicht mehr getaktet oder nicht mehr gespeist werden.
- Verwendung von statischen RAMs anstelle von dynamischen RAMs.
- Verwendung von angepassten Datenwortbreiten für die Busse, Register, Speicher, ALU, usw.
- Verwendung von Datenpfaden mit Zwischenregistern zur lokalen Speicherung von Zwischenresultaten.

Mit Optimierungstechniken auf RTL- und Logikniveau wird versucht, die geschalteten Kapazitäten zu reduzieren:

- Verwendung von optimalen Datenformaten (zum Beispiel *gray code* für Zustandszähler, *bus invert coding* für Datenbusse, *sign magnitude coding* bei arithmetischen Einheiten).
- Verwendung von kombinatorischen Logikstrukturen, welche die *Glitches* reduzieren sowie geringe Eigenaktivitäten haben.
- Verwendung von *Gated Clocks* für Registergruppen, welche nicht ständig aktiviert sein müssen.
- Verwendung von Standardzellen mit geeigneten Transistorengrössen.
- Logikparallelisierung: Nebst der Möglichkeit, die Speisespannung Vdd zu senken, kann eine Parallelisierung in folgenden Fällen auch die geschaltete Kapazität C_{sw} reduzieren:
  - Ein grosser Speicher wird durch verschiedene kleine Speicherzellen ersetzt, welche individuell aktivierbar sind und jeweils weniger Leistung benötigen als der grosse Speicher.
  - Ein synchroner Zähler kann durch k Zähler ersetzt werden, welche entsprechend weniger zählen. Leistungsreduktionsfaktor: k.

  - Eine Reihe von Shift-Registern kann durch k Reihen von n/k Shift-Registern ersetzt werden, welche k mal weniger schnell getaktet werden. Leistungsreduktionsfaktor: k.
- Zwischenregister in Datenpfaden begrenzen toggelnde Datenbusse nach arithmetischen und logischen Einheiten.
- Eingangsregister vermeiden ein Toggeln arithmetischer Einheiten während der *Fetch*-Phase.
- In grösseren Prozessorensystemen reduziert die Verwendung interner Cache-Speicher die Anzahl der Zugriffe auf den externen Hauptspeicher.

2.3 Die verschiedenen Architekturtypen von Prozessoren

Prozessoren und ihre Architekturen können nach verschiedenen Kriterien klassifiziert werden.

Generell verwendbare Architekturen und Spezialarchitekturen

Eine erste Klassifizierung untersucht die Prozessoren nach ihrer Verwendbarkeit[1]. Generell verwendbare Architekturen sind möglichst universell gestaltet und nicht für einen bestimmten Algorithmus optimiert. Sie lassen sich frei programmieren und stellen dazu eine möglichst ausreichende Menge von universell verwendbaren, arithmetischen und logischen Operationen sowie Kontrollinstruktionen zur Verfügung.

Eine Spezialarchitektur ist für einen Algorithmus oder eine Algorithmusfamilie optimiert. Der Datenpfad sowie die Menge der von ihm verarbeitbaren Operation ist anwendungsoptimiert. Die Kontrolleinheit ist meist fest verdrahtet oder lässt sich nur in einem beschränkten Rahmen programmieren.

Klassifizierung gemäss Speicherorganisation

Eine zweite Klassifizierung berücksichtigt die Organisation von Programm- und Datenspeicher und wendet sich sinnvollerweise bei generell verwendbaren Prozessoren an.

Prozessoren, wo die Daten sowie die Instruktionen im selben Speicher liegen, besitzen eine Architektur vom Typ „von Neumann“. Im Gegensatz dazu besitzen Architekturen vom Typ „Harvard“ zwei getrennte Speicher (und damit auch Busse) für Daten und Instruktionen.

Bei den Prozessoren vom Typ „von Neumann“ müssen die Instruktionen und Daten über den selben Bus in den Prozessor geladen werden. Dies kann zu einem Geschwindigkeitsnachteil bezüglich den *Harvard*-Prozessoren

[1] *General purpose architectures ⇔ special purpose architectures*

führen, welche einen parallelen Zugriff auf Instruktionen und Daten haben. Andererseits können Prozessoren vom Typ „von Neumann“ den Speicher effizienter nützen.

Um Daten noch schneller und effizienter verarbeiten zu können, besitzen DSPs meist eine „doppelte Harvard-Architektur“ mit zwei Datenspeichern nebst dem Programmspeicher[1]. Diese Parallelisierung ermöglicht in nur einem einzigen Taktzyklus das Lesen und die Verarbeitung der *MAC*-Instruktion (inklusive dem Lesen des Datenwortes und des Koeffizienten aus den entsprechenden Speichern).

CISC[2]-RISC[3]

Der gleichzeitige Zugriff auf Programmcode und Daten mittels einer Harvard-Architektur ist nur eine der möglichen Techniken, die Verarbeitungsgeschwindigkeit zu erhöhen. Eine weitere Möglichkeit ist, die Instruktionsmenge eines Prozessors mit komplexeren Instruktionen zu ergänzen.

Ein CISC-Prozessor besitzt eine Instruktionsmenge, welche weit mehr als nur einfache Standardinstruktionen und elementare, arithmetische Operationen enthält. Nebst der grossen Instruktionsmenge, welche unter anderem komplexe, arithmetische Operationen wie logarithmische und trigonometrische Funktionen enthalten kann, sind auch die vielen Arten von Adressierungsmodi und Datenformaten, unterschiedliche Weiten der Instruktionswörter, sowie variable Mengen von Operanden typisch für einen CISC-Prozessor[4].

CISC-Instruktionen beinhalten der Komplexität wegen verschiedene Instruktionsphasen, welche jeweils in einem Taktzyklus durchgeführt werden. Besonders bei arithmetischen Instruktionen, welche mittels eines iterativen Algorithmus ein Resultat berechnen, werden eine sehr hohe Anzahl von Taktzyklen benötigt, welche in die Hunderte oder sogar Tausende reichen kann.

Diese Instruktionsphasen werden mittels Prozessor-internen Programmen durchgeführt, welche meist fest in ein ROM programmiert sind. Im Gegensatz zu den Anwenderprogrammen werden diese internen Programme *Mikroprogramme* genannt und die dabei enthaltenen Instruktionen *Mikroinstruktionen*.

---

[1] *[double, advanced, extended] Harvard architecture*
[2] CISC: *Complex Instruction Set Computer*
[3] RISC: *Reduced Instruction Set Computer*
[4] Beispiel VAX 11/780: 304 Instruktionen, 16 Adressierungsmodi, Anzahl Operanden: 0 bis 6, Datenformate: *byte, word, long word, string, packed array, float*

Als Vorteile eines komplexen Instruktionssets ist nicht nur ein möglicher Leistungszuwachs zu erwähnen. Das grosse Instruktionsset lässt auch eine sehr effiziente Programmierung zu. Diese kann positiv die Grösse des Programmcodes, die Anzahl der notwendigen Speicherzugriffe (Programm und Daten) und damit auch den Stromverbrauch des Gesamtsystems beeinflussen.

Auf Grund der Komplexität des Instruktionssets ist jedoch eine effiziente Programmierung häufig nur noch von Hand möglich, da Compiler nicht mehr in der Lage sind, all die vorhandenen Instruktionen optimal zu verwenden. Zudem wurden die Prozessoren so komplex, dass ihre (Weiter-) Entwicklungen sehr schwierig wurde.

Diese Nachteile gaben zu Beginn der 80er-Jahre den Anstoss, nach einfacheren Architekturstrukturen zu suchen, welche die oben aufgelisteten CISC-Probleme nicht mehr kennen. Speziell ein Architekturtyp wurde gefunden, welche heute unter dem Namen RISC[1] bekannt ist. Der Name basiert lediglich auf der Tatsache der limitierten Instruktionsmenge, was jedoch nur eine von vielen Eigenschaften eines RISC-Prozessors ist. Es folgt deshalb eine vollständigere Liste mit typischen RISC-Eigenschaften:

- wenige Instruktionen
- wenige Adressierungsmodi
- kleine Anzahl von Instruktionsformaten
- alle Instruktionen besitzen die selbe Grösse
- jede Instruktion wird in einem einzigen Taktzyklus durchgeführt
- sehr viele Register
- externe Speicher werden nur mit Instruktionen vom Typ *Load/Store* angesprochen
- sehr kleine Kontrolleinheit
- ist sehr geeignet für Techniken, welche von höheren Programmiersprachen verwendet werden wie: Unterprogramme, lokale Variablen, Arrays
- sehr regelmässige Architektur

RISC-Prozessoren können viel effizienter mittels einer Hochsprache programmiert werden als CISC-Prozessoren. Auf Daten- und vor allem Programmspeicher muss aber wegen den simplen Instruktionen häufiger zugegriffen werden. Die RISC-Technologie eignet sich jedoch hervorragend zur Verwendung von Cache-Speicher, und dies erlaubt, dass die Anzahl tatsächlicher Zugriffe auf die externen Speicher nicht grösser sein muss als bei CISC-Prozessoren.

[1] *Reduced Instruction Set Computer*

Systeme für die Parallelverarbeitung von Daten

Ansprüche nach immer leistungsfähigeren Computersystemen sowie die in Kapitel 2.2.2 gezeigte Technik der Senkung der Speisespannung zur Reduktion des Stromverbrauches verlangen häufig eine parallele Art der Datenverarbeitung. Heute existieren verschiedenste Arten von Computer- und Prozessorsystemen, welche versuchen, diese Nachfrage zu befriedigen. Ihre Eigenschaften bezüglich Parallelität können mit einer sehr bekannten Klassifizierungsmethode, welche aus den 60er-Jahren stammt, untersucht werden [7]. In Funktion der Anzahl der Instruktions- und Datenströme innerhalb eines Systems wird dabei zwischen folgenden vier Architekturtypen unterschieden:

Ein SISD[1]-Computer besitzt meist eine einzige Prozessoreinheit (*PE*) und einen einzelner Speicher. Dies ist der üblichste Computertyp.

Ein SIMD[2]-Computer hat typischerweise eine Kontrolleinheit und verschiedene Verarbeitungseinheiten, welche zur selben Zeit die selben Operation auf verschiedene Datenströme anwenden kann (Abbildung 3). Ein Netzwerk ermöglicht den Datenaustausch zwischen den Prozessoreinheiten.

SIMD-Computer sind besonders effizient, wenn parallel die selben Aufgaben durchgeführt werden müssen, wie dies etwa bei Vektor- und Matrixoperationen oder bei der Verarbeitung von Bildelementen der Fall ist. Häufig werden SIMD-Computer auch Array-Prozessoren genannt. SIMD-Computer können auf zwei verschiedene Arten implementiert werden. Eine Möglichkeit ist, eine Serie von Prozessoreinheiten (PE), welche jeweils eine Datenverarbeitungseinheit sowie einen lokalen Speicher besitzen, über ein Netzwerk zu verbinden (PE-PE). Die andere Möglichkeit ist, ein Netzwerk zu verwenden, welches direkt die Datenverarbeitungseinheiten sowie eine Reihe von Speicher verbindet (P-M).

MIMD[3]-Computer werden allgemein *Multi-Prozessor-Systeme* genannt. Im Gegensatz zu den SIMD-Computer haben sie den Vorteil, dass die einzelnen Prozessoreinheiten autonom und nicht-synchron arbeiten und damit die verschiedenen Datenströme unterschiedlich bearbeiten können. Wie bei den SIMD-Systemen sind ebenfalls bei den MIMD-Computern verschiedene Konfigurationen des Netzwerkes möglich (Abbildung 3 zeigt die PE-PE-Konfiguration).

Zur Komplettierung dieser Klassifizierung muss noch die MISD[4]-Architektur erwähnt werden, welche jedoch selten, wenn sogar überhaupt

[1] SISD: *single instruction stream, single data stream*
[2] SIMD: *single instruction stream, multiple data stream*
[3] MIMD: *multiple instruction stream, multiple data stream*
[4] MISD: *multiple instruction stream, single data stream*

nie verwendet wird. Sie enthält verschiedene Prozessoreinheiten, welche in einer Pipeline angeordnet, einen einzelnen Datenstrom bearbeiten.

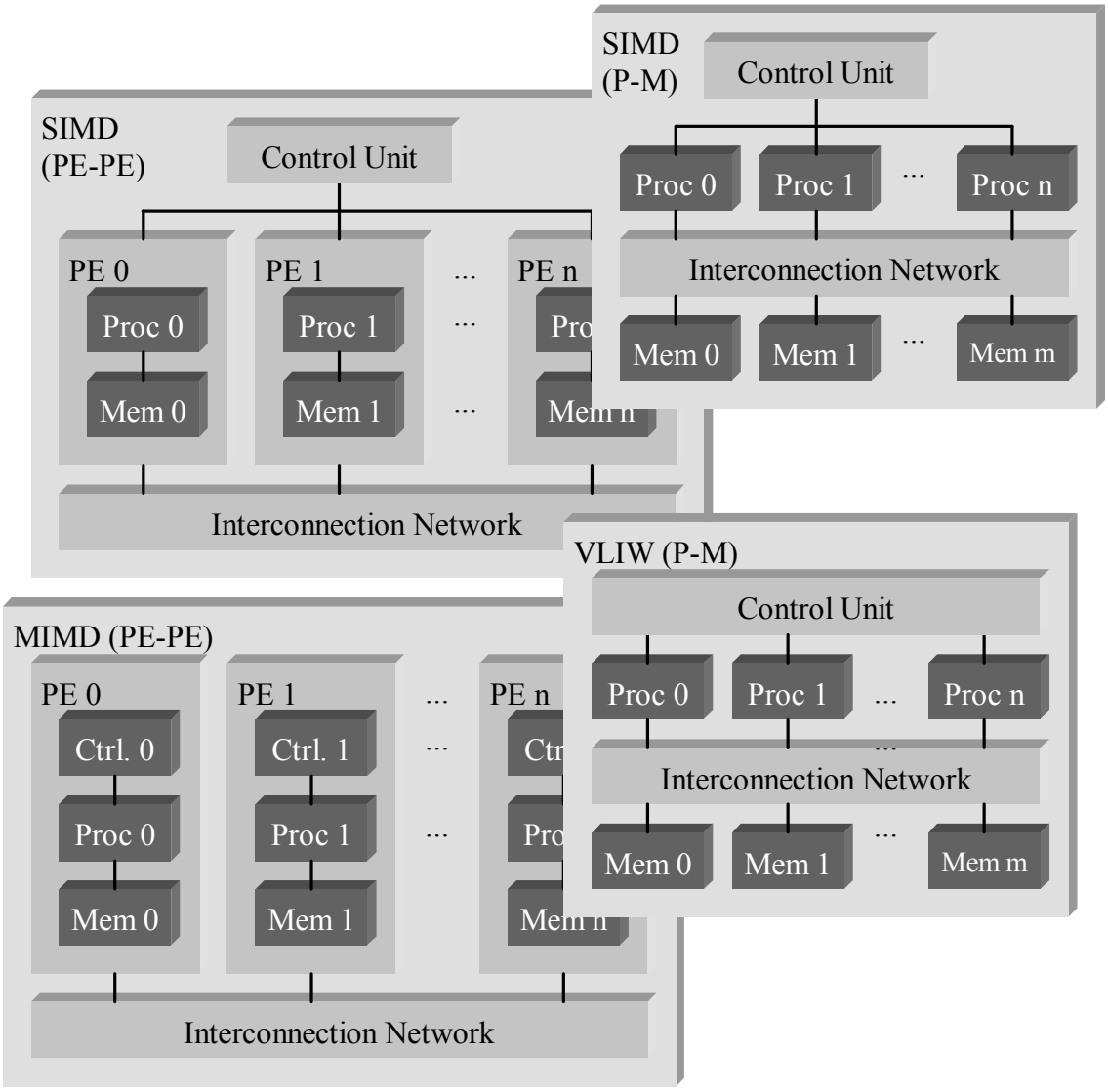


Abbildung 3: Die Haupttypen paralleler Prozessorsysteme

Die vier aufgezählten Computertypen unterliegen zum Teil grossen Restriktionen und besitzen Nachteile, welche ihre Verwendbarkeit begrenzen. So kann ein SISD-Prozessor gleichzeitig nur ein Datenwert auf einmal bearbeiten. Damit die Verarbeitungsgeschwindigkeit doch erhöht werden kann, werden grosse Pipeline-Strukturen verwendet, welche es erlauben, höhere Taktfrequenzen zu verwenden.

Im Gegensatz dazu besitzen die SIMD-Prozessoren nur dann einen Vorteil, wenn sich die zu verarbeitenden Algorithmen parallelisieren lassen, was jedoch nicht immer bis selten der Fall ist.

Das Problem der MIMD-Computer ist, dass die einzelnen Prozessoreinheiten zueinander nicht synchronisiert sind, was für eine aufgeteilte Bearbeitung eines Algorithmus nicht dienlich ist.

Um diesen Nachteilen aus dem Wege zu gehen, werden heutzutage im Bereich der digitalen Signalverarbeitung häufig DSPs mit einer sogenannten VLIW[1]-Architektur verwendet. Diese DSPs besitzen verschiedene Datenverarbeitungseinheiten und Speicher, welche wie bei einem SIMD-Prozessor von einer einzigen Kontrolleinheit gesteuert werden. Der Unterschied bezüglich den Letztgenannten besteht jedoch darin, dass die verschiedenen Datenverarbeitungseinheiten gleichzeitig unterschiedliche Operationen durchführen können. Zur gleichzeitigen Kontrolle aller Datenverarbeitungseinheiten wird ein grösserer Fluss an Kontrollinformationen benötigt. Damit dieser geliefert werden kann, werden sehr breite Instruktionswörter verwendet (deshalb der Name VLIW).

Die verschiedenen Datenverarbeitungseinheiten eines VLIW-DSPs können so in synchronisierter Weise verschiedene Teile (oder Operationen) eines Algorithmus durchführen. Nichtsdestotrotz muss gesagt werden, dass auch VLIW-Architekturen Restriktionen an einen Algorithmus stellen, wenn dieser effizient verarbeitet werden soll.

2.4 Die Evolution der ASIC-Technologien und der Schaltungs-Entwurfsmethoden

Die Degradierung der Qualität von Computerprogrammen im PC-Bereich ist bekannt. Für Aufgaben, welche vor einigen Jahren noch mit einem PC erledigt werden konnten, welcher mit 8 MHz getaktet wurde und der 640 kBytes RAM besass, wird heute ein mit einem *high-end* - Prozessor ausgestatteter PC benötigt, der mit 800 MHz getaktet ist und 128 MBytes RAM besitzt! Sicherlich, um fair zu bleiben muss auch gesagt werden, dass heutige Programme gewisse neue Funktionen besitzen, welche die Anforderungen an einen PC erhöhen. Entscheidend sind jedoch die Methoden, wie Programme heute im Gegensatz zu früher realisiert werden. Wo früher eine Idee manuell in einen Maschinen- oder Assemblercode umgesetzt wurde, kann heute ein Lösungsansatz in einer Hochsprache beschrieben werden, welche dann von einem Compiler in den Maschinencode umgesetzt wird. Die Effizienz des generierten Maschinencodes (Grösse und notwendige Verarbeitungszyklen) ist je nach Zielprozessortyp bezüglich einem handgeschriebenen Code um einen Faktor 2-8 schlechter (bei DSPs sogar oftmals noch schlechter), was jedoch durch leistungsfähigere Prozessoren und Computer kompensiert wird. Dieser Trend hält übrigens an, denn für immer mehr Anwendungen wird nach komfortablen, hardwareunabhängigen Lösungen gesucht und in Form von Interpreter- oder Skriptsprachen wie Java oder Tcl gefunden. Bekanntlich

[1] VLIW: *Very Long Instruction Word*

werden aber interpretierte Programme wesentlich langsamer verarbeitet als kompilierte.

Das selbe Phänomen existiert auch bei den digitalen, integrierten Schaltungen. Auf der einen Seite beobachtet man einen unanhaltsamen Fortschritt im Halbleitertechnologiebereich, welcher unter anderem durch Reduzierung der Transistorengrösse[1] und Erhöhung der Anzahl der Metalllayer erzielt wird und welche die Transistorenpackdichte in die Höhe schnellen lässt. Auf der anderen Seite wird dieser technologische Fortschritt nur allzu häufig zur Kompensation von modernen Entwurfsmethoden verwendet, welche zwar rascher sind als die älteren, jedoch nur eine geringe Entwurfslogikeffizienz besitzen, d.h. grössere Schaltungen entstehen lassen als die alten Entwurfsmethoden.

Soll der technologische Fortschritt nicht lediglich zur Kompensation einer geringen Logikeffizienz von raschen Designmethoden verwendet werden, so kann für den Schaltungsentwurf trotzdem nicht einfach wieder auf die alten Designmethoden und Entwurfstechniken wie etwa Layout-Design oder manuelle Schaltungsoptimierung zurückgegriffen werden. Denn diese Methoden eignen sich nicht mehr für moderne Schaltungen, welche Komplexitäten von vielen Dutzenden von Millionen von Transistoren besitzen. Die Überlegungen zu Beginn der 80er Jahren waren richtig, dass es zur Realisierung von modernen, grossen Designs neue, geeignetere Entwurfsmethoden braucht. Und doch können nicht einfach alle heute üblichen Implementierungsmethoden eingesetzt werden, wenn man sich an der Machbarkeitsgrenze einer Designaufgabe bewegt. Es muss immer nach einem Kompromiss zwischen Entwurfszeit und Entwurfsqualität gesucht werden. Die zu verwendenden Entwurfsmethoden müssen für jedes Projekt erneut, unter Berücksichtigung der Rahmenbedingungen und dem Fortschritt der Technik, evaluiert werden. Eine Methode, welche heute sinnvoll ist, war vor 10 Jahren vielleicht noch undenkbar und hat in 10 Jahren bereits wiederum ihre Vorteile gegenüber anderen Methoden verloren.

2.4.1 Designmethoden für den Lowest-Power-Bereich zum heutigen Zeitpunkt

Die qualitativen Verbesserungen der Synthesetools erlauben heute auch in kritischen Bereichen die Verwendung von Hardwarebeschreibungssprachen zur Designdefinition. Schon vor einigen Jahren konnte bewiesen werden, dass ein Design, welches mittels Logiksynthese realisiert wurde nicht mehr als 3 bis 6 Prozent grösser sein muss als ein Design, welches von hochqualifizierten Spezialisten von Hand gebaut und bis ins letzte Detail

[1] genauer: Minimale Länge der Transistorengates = *min. feature size* = F

optimiert wurde [8] (natürlich auch mit Standardzellen). Auch für lowest-power Entwicklungen ist deshalb die Verwendung von Hardwarebeschreibungssprachen nur sinnvoll.

Das Interpretationsvermögen der Synthesetools erreichte unterdessen ein solches Niveau, dass beinahe alle Möglichkeiten, welche VHDL bietet, korrekt in eine Hardware umgesetzt werden können (ausser natürlich den Operationen, welche Zeitverhalten beschreiben). Der Gebrauch von Funktionen, Schleifen, Paketen und komplexen Parametrisierungen kann gewollt in einer Designbeschreibung verwendet werden, solange der Designer wirklich weiss, wie das Synthesetool diese Möglichkeiten verarbeitet. Es ist also immer noch wichtig, dass ein Designer die Grenzen eines Synthesetools bestens kennt, wenn er die Struktur- oder RTL-Beschreibungsform verlässt und ein Design auf höherem Abstraktionsniveau beschreiben will.

Wird ein Design auf höherem Abstraktionsniveau beschrieben, so können gewisse Low-Power-Techniken wie *Gated Clocks* nicht mehr kontrolliert werden. EDA-Toolhersteller begegnen dem indem sie Tools anbieten, welche ein synchrones Design, welches einen einzigen Clock für alle Register hat, in ein Design mit *Gated Clocks* transformiert. Die Clockstrategie muss auf diese Weise nicht mehr während der Designentwicklung berücksichtigt werden, sondern kann zu einem späteren Zeitpunkt, während der Synthese, definiert werden.

2.5 Schlussfolgerung

Es darf gesagt werden, dass die Leistung der Synthesetools bei der Umsetzung einer Designbeschreibung in eine Hardware bezüglich Zeitaufwand und vor allem Lösungsqualität auch bei Anwendungen mit schwierig zu erfüllenden Randbedingungen zufriedenstellend ist und Verbesserungen der Logiksynthesetools keine Prioritäten besitzen. Steigerungen der Designqualität und Reduzierung der Entwicklungszeiten müssen auf Architekturniveau erzielt werden, und zwar durch die gute Wahl von Methoden zur Architekturentwicklung.

Welches ist aber eine gute Methode? Ist es der frei programmierbare DSP, welcher in Form eines IP-Blockes in einem Design verwendet wird? Oder ist es die Verwendung von Tools, welche automatisch eine Architektur generieren? Oder muss bei der Entwicklung einer Anwendung mit schwierigen Randbedingungen die Architektur manuell gebaut werden? Auf diese Frage kann keine generelle Antwort gegeben werden. Jede Methode hat ihre Vor- und Nachteile. Es kann nicht einmal gesagt werden, dass eine Methode für ein ganzes Design die beste ist. Selbst innerhalb eines Designs

kann es vorteilhaft sein, verschiedene Designmethoden zur Realisation der verschiedenen Unterblöcke zu verwenden.

Referenzen

[1] C. Piguet,“Conception des circuits intégrés numériques”, Cours 1994-1995 6e et 7e semestre, Institut de Microtechnique, Université de Neuchâtel, Suisse

[2] „An Introduction to VHDL for Synthesis and Simulation“, David Pellerin, Accolade Design Automation, Inc., http://www.acc-eda.com/h_intro.htm

[3] 1076-1993 IEEE Standard VHDL Language Reference Manual
1076-1987 IEEE Standard VHDL Language Reference Manual
1076.2-1996 IEEE Standard VHDL Mathematical Packages
1076.3-1997 IEEE Standard VHDL Synthesis Packages
1076.4-1995 IEEE Standard VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification.
1076.6-1999 IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis
1164-1993 IEEE Standard Multivalue Logic System for VHDL Model Interoperability

[4] 1364-1995 (Section 1-4) IEEE Standard Description Language Based on the Verilog(TM) Hardware Description Language

[5] H. Kaeslin, “Design of VLSI Circuits, Energy Efficiency and Heat Removal”, Lectures notes, Integrated Systems Laboratory, ETH Zürich

[6] K. Roy & S. C. Prasad, “Low-power CMOS VLSI circuit design”, “A Wiley-Interscience publication.”, ISBN 0-471-11488-X

[7] MJ Flynn, “Very high-speed computing systems”, Proc. of the IEEEE, Vol. 54, Dec. 1966, pp. 1901-1909

[8] A. Drollinger, H. Hügli, C. Piguet, „Modèle VHDL synthétisable d’un microprocesseur CoolRisc", Diplomarbeit von A. Drollinger.

3 Eine Hardware/Software – Co-Design-Methodik zur Realisierung von Filter- und Kontrollapplikationen

Frei programmierbare DSPs ermöglichen eine schnelle und einfache Implementierung eines Algorithmus, wobei jedoch die an ein Design gestellten Randbedingungen bezüglich Grösse und Stromverbrauch nur teilweise berücksichtigt werden können.
Dieses Kapitel stellt eine neue Designmethode vor, welche versucht, einerseits die Einfachheit der Verwendung eines frei programmierbaren DSPs zu nutzen, andererseits die DSP-Architektur in einer zusätzlichen Optimierungsphase so anwendungsspezifisch zu optimieren, dass das schlussendlich erhaltene Implementierungsresultat einer HDL-basierten Lösung in Sachen Qualität möglichst wenig nachsteht.
Dazu verwendet die Designmethode eine DSP-Architektur, welche frei programmierbar, aber auch hoch parametrisierbar ist. Die Programmierbarkeit gibt der Methode einen Softwarecharakter, die parametrisierbare VHDL-Architekturbeschreibung aber ebenfalls einen Hardwarecharakter.
Um die vorgestellte Designmethode praktisch zu entwickeln, zu testen und zu verwenden, wurde der sogenannte HotDSP[1] entwickelt. Er besitzt eine Architektur, welche frei programmierbar und so für low-power Filteranwendungen optimiert ist, dass Filteralgorithmen möglichst einfach und effizient implementiert werden können. Mit über 130 Parametern kann er optimal an eine Anwendung angepasst werden.

In einem ersten Kapitelteil wird der Designfluss der neuen Implementierungsmethode vorgestellt. Es folgt die Architekturbeschreibung des HotDSPs und die Präsentation der Entwicklungssoftware. Die anschliessenden Anwendungsbeispiele zeigen, wie einfach sich mit Hilfe der Co-Designmethode zum Beispiel Interpolations- und Dezimationsfilter implementieren lassen.

[1] *HotDSP* ist ein willkürlich gewählter Name

3.1 Wiederverwendbarkeit von DSP-Architekturen durch Architekturenkonfigurierung

Die Verwendung eines frei programmierbaren Prozessorkerns in Form eines IP-Blockes innerhalb einer Integrierten Schaltung garantiert meist eine schnelle und einfache Realisation eines Projektes da der Prozessorkern bereits hardwareerprobt ist und Entwicklungswerkzeuge wie Assembler und Compiler bereits existieren. Aus Qualitätssicht ist eine solche Lösung jedoch nicht immer optimal. Der Grund liegt darin, dass die in IP-Form erhältlichen Prozessorenkerne selten auf eine bestimmte Applikation abgestimmt sind, sondern möglichst universell gestaltet sind und damit für eine bestimmte Applikation unoptimale Kapazitäten besitzt.

MIPS, ARM und Tensilica, drei hauptsächlich im Bereich der RISC-Prozessoren tätige IP-Vertreiber, versuchen dieses Problem zu lösen, indem sie beschränkt konfigurier- und erweiterbare Prozessorkerne anbieten [1]. Die Konfiguriermöglichkeiten sind jedoch stark begrenzt. So betreffen die wichtigsten Konfigurationspunkte die Anzahl der Register (32 oder 64), die Implementierungsart der Register (*Latches* oder Register), Art und Anzahl der peripheren Einheiten (Timer, *Interrupt*-Kontroller), Verwendung von Cache-Speicher und die optionale Verwendung von *Gated Clocks*. Die arithmetische Einheit selbst ist entweder nicht konfigurierbar, oder aber es kann lediglich spezifiziert werden, dass einige speziell ressourcenhungrige Operationen wie die Multiplikation oder die *MAC*-Operation nicht implementiert werden. Flexibilität und Rechenleistungen dieser Prozessoren können mit Koprozessoren erhöht werden. Diese Koprozessoren können anwendungsspezifische, vom Programmierer über Bibliotheken aufrufbare Funktionen enthalten und werden direkt in den Datenbus der Prozessoren eingebunden. Wie man sieht, sind die Konfigurationsmöglichkeiten dieser Prozessoren stark beschränkt. Eine Feinabstimmung der Anzahl der Register sowie der Datenbreiten von Register, Daten- und Adressbussen ist nicht möglich (auch aus rechtlichen Gründen).

Als konfigurierbarer DSP ist der *Gepard* bekannt [2][3][4]. Der *Gepard* ist ein *General Purpose DSP*. Seine Konfigurierbarkeit ist im Gegensatz zu den oben beschriebenen RISC-Prozessoren feiner, und so können unter anderem auch die Adressbereiche und im 8-Bit-Raster die Datenweiten den Bedürfnissen angepasst werden. Die Architektur und die Befehlskodierung bieten Gelegenheiten, zusätzliche Befehle, Adressiermodi und Interruptebene zu integrieren, oder fest verdrahtete DSP-Module als Co-Prozessoren anzukoppeln. Aber auch seinen Konfigurationsmöglichkeiten sind Grenzen gesetzt: Wäre zum Beispiel eine 18-Bit-Datenauflösung genügend, so muss ein 24-Bit Datenpfad gewählt werden. Man darf diese Einschränkungen jedoch nicht als negativer Punkt sehen. Der *Gepard* soll absichtlich nicht zu einem *Special Purpose DSP* konfiguriert werden

können, sondern er soll das bleiben, zu dem er konzipiert wurde; ein *General Purpose DSP*.

Hoch konfigurierbare Prozessoren kann man im akademischen Bereich finden, wie etwa die *stack*- und registerbasierten „*Embedded*“ Prozessoren von T. Röwer [5]. Es handelt sich dabei um Prozessoren, welche gewisse optimierte Voraussetzungen liefern für Kontroll- und Protokolloperationen sowie für leichte Datenverarbeitungen. Die Konfigurationsmöglichkeiten beinhalten unter anderem eine frei Wahl der Anzahl und der Breite der verschiedenen Register, die Breite des Datenbusses, eine Selektion der zu implementierenden ALU-Instruktionen und eine Reduzierung der Instruktionswortbreite.

Die bis anhin besprochenen Prozessoren stellen oftmals dank ihrer Konfigurierbarkeit eine optimale Lösung dar bei Anwendungen, welche aus Flexibilitätsgründen einen Prozessor benötigen. Im Gegensatz dazu wird in diesem Kapitel einerseits gezeigt, dass ein konfigurierbarer Prozessor, unter der Voraussetzung, dass seine Konfigurationsmöglichkeiten genügend gross sind, sogar in Anwendungen verwendet werden können, wo normalerweise ein optimierter Hardware-Block die Aufgaben durchführen muss wie zum Beispiel in speziellen Filterblöcken. Andererseits wird mit einem in diesem Kapitel vorgestellten Designflusskonzept auch gezeigt, wie ein hoch konfigurierbarer Prozessor mit den Softwareentwicklungstools wie Assembler und Debugger interagieren müssen, damit diese Tools automatisch an die gewählten Prozessorkonfigurationen angepasst werden.

3.2 Die Kombination zweier Implementierungsmethoden zu einer neuen Hardware/Software – Co-Designmethode

Die hier vorgestellte, neue Designmethode versucht einerseits die Einfachheit einer Implementierung eines Algorithmus mit Hilfe eines frei programmierbaren DSPs zu nutzen, die DSP-Architektur andererseits in einer zusätzlichen Optimierungsphase auf eine solche Weise anwendungsspezifisch zu optimieren, dass das schlussendlich erhaltene Implementierungsresultat einer HDL-Beschreibung in Sachen Qualität nicht nachsteht [6]. Diese Optimierung soll sich nicht lediglich auf eine grobe Anpassung der DSP-Ressourcen und der peripheren Einheiten reduzieren, sondern soll bis ins feinste Detail die DSP-Architektur der Anwendungssituation anpassen.

Die Designmethode benötigt eine DSP-Architektur, welche einerseits frei programmierbar ist und andererseits in höchstem Masse parametrisierbar ist. Die Programmierbarkeit gibt der Methode einen Softwarecharakter, die

parametrisierbare VHDL-Architekturbeschreibung aber ebenfalls einen Hardwarecharakter.

Die Parametrisiermöglichkeiten von VHDL ermöglichen es, dass während der Optimierungsphase der Architektur nicht etwa am Quellencode des DSPs „herumgebastelt“ werden muss, sondern dass die Architekturanpassungen durch eine Adaptation von Parametern erfolgen kann.

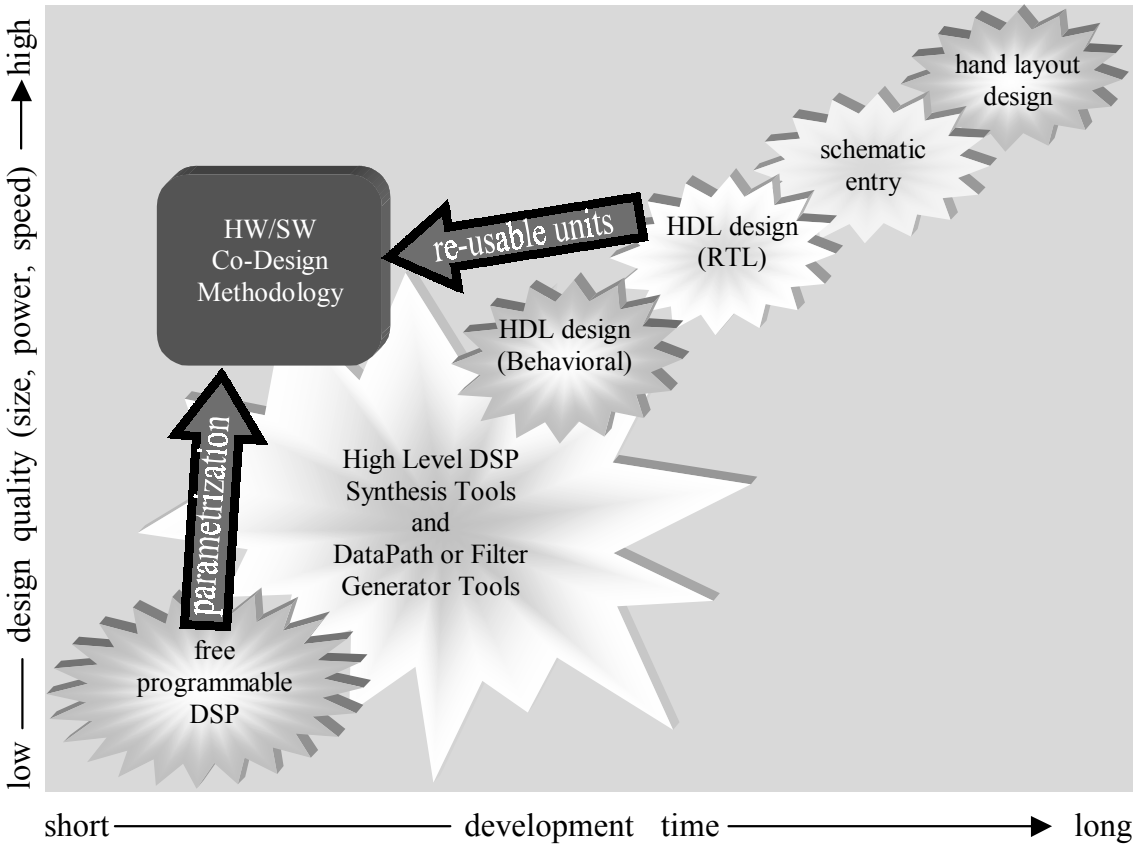


Abbildung 4: Die Einordnung der Co-Designmethode bezüglich anderen Implementierungsmethoden

Die Parameter können in einer VHDL-Beschreibung an verschiedenen Programmstellen als Konstante definiert werden, so zum Beispiel in der *Entity*- Deklaration sowie am Beginn einer Architekturbeschreibung, einer Funktion, einer Prozedur und eines Prozesses. Der jedoch für die besprochene Designmethode beste Ort zur Deklaration aller Parameter ist in einem separaten Deklarationspaket (*declaration package*), welches von den verschiedenen VHDL-Modulen importiert wird. Durch die Verwendung eines solchen Deklarationspakets werden sämtliche Parameter zentral am selben Ort definiert, was zum Vorteil hat, dass der Anwender für die Architekturanpassung lediglich eine VHDL-Datei editieren muss. Zudem lässt sich ein Zugriff von Sekundärprogrammen auf die Parameter, sei es zu deren automatischer Modifikation, sei es zur automatischer Anpassung von

Entwicklungstools an die gewählte DSP-Konfiguration, viel einfacher gestalten.

3.3 Hardware- und Softwarevoraussetzungen

Da die Verwendung des frei programmierbaren DSPs nicht wegen der Möglichkeit der Modifikation des zu implementierenden Algorithmus zu einem späteren Zeitpunkt gewählt wurde, sondern lediglich als Mittel zum Zweck einer schnellen Implementierung eines fixen Algorithmus ist, kann das entwickelte DSP-Programm in einem ROM, oder wenn die Grösse des Programmcodes aufgrund der Komplexität der Anwendung gering ist, in einem mit Standardzellen aufgebauten LUT[1] „eingefroren“ werden.

Zur einfacher Implementierung eines Algorithmus wird nicht nur ein parametrisierbarer DSP benötigt, sondern auch Entwicklungstools wie Assembler, Debugger, Simulator und Testbench. All diese müssen imstande sein, sich den verschiedenen DSP-Konfigurationen anzupassen: Der Assembler muss für den Prozessor in jeder Konfiguration den richtigen Programmcode generieren, der Simulator muss jede Konfiguration simulieren können, usw.

Indem der Assembler imstande ist, das VHDL-Konfigurationspaket mit all seinen Parameterdefinitionen zu lesen (und zu interpretieren), kann der Assembler einfach und automatisch an die momentan definierte DSP-Konfiguration angepasst werden.

Der Simulator und Debugger werden am einfachsten als VHDL-Simulator mit einem Debugger-Erweiterungsmodul realisiert. Voraussetzung dazu ist, dass der VHDL-Simulator eine programmierbare Erweiterungsschnittstelle besitzt. Die Adaptation des Simulators passiert auf diese Weise automatisch bei einer Neukompilierung des DSPs. Auch die Debugger-Erweiterung muss der gewählten Konfiguration folgen können. Er kann dazu das VHDL-Konfigurationspaket lesen. Da dies bereits vom Assembler getan wird, ist es jedoch einfacher, wenn dieser nebst dem Programmcode gleich auch eine Informationsdatei für den Debugger erzeugt, welche die für den Debugger relevanten Informationen des VHDL-Konfigurationspaket zusammen mit anwendungsspezifischen Informationen in einer einfach zu lesenden Syntax enthält. Der Debugger erhält dann durch diese Datei die für ihn notwendigen Informationen zu seiner Adaptation an die gewählte DSP-Konfiguration.

Für die VHDL-Simulation wird ein Testbench benötigt. Wie der Simulator und der Debugger muss dieser ebenfalls an die DSP-Konfiguration angepasst werden. Dies ist einfach, da der Testbench (meist) als VHDL-

[1] LUT: Look-up-Table

Modul konzipiert ist und zur Adaptation an die jeweils gewählte DSP-Konfiguration lediglich das VHDL-Konfigurationspaket zu lesen braucht. Interfaceparameter wie zum Beispiel die Anzahl der Input- und Output-Ports können so vom Testbench automatisch angepasst werden.

3.4 Der Designfluss der Designmethode

Abbildung 5 zeigt den Fluss der Designmethode. Drei Hauptetappen sind ersichtlich: Implementierung, Optimierung der DSP-Struktur und Erzeugung des Programmcodes für die optimierte DSP-Struktur zusammen mit einer letzten Simulation.

Für eine erste Implementierung eines Algorithmus wird der DSP in einer Standardkonfiguration verwendet, welche genügend Ressourcen aufweisen soll für die meisten der Zielanwendung, also meistens viel zu gross ist. Der Assembler liest einerseits den Quellencode des Anwendungsprogramms, andererseits zur Ermittlung der aktuellen DSP-Konfiguration das VHDL-Konfigurationspaket, und generiert einen Programmcode, welchen er in Form einer synthetisierbaren VHDL-Beschreibung eines LUTs ausgibt. Zusätzlich schreibt er die für den Debugger relevanten Informationen vom VHDL-Konfigurationspaket und vom Quellencode in die Debugger-Informationsdatei. Der VHDL-Simulator, zusammen mit dem Debugger, welcher als Simulatorerweiterung konzipiert ist und der durch die Informationsdatei Kenntnisse von der aktuellen DSP-Konfiguration hat, hilft dem Anwender, seine Applikation schnell und effizient zu implementieren.

Läuft das entwickelte Programm einmal korrekt auf dem DSP, so kann die Optimierung der DSP-Struktur in Angriff genommen werden. Während dieser Etappe werden durch Modifikation des VHDL-Konfigurationspakets unbenutzte oder nicht voll ausgenutzte Ressourcen entfernt oder reduziert. Dies können Anpassungen sein von Registern, Input- und Output-Ports, der Grösse des Programmspeichers und des Stackes für den Programmfluss, Breite von Daten- und Programmbussen, *Gated Clocks*, der Menge der verwendeten arithmetischen Operationen und Verzweigungskonditionen, usw. Das Unterkapitel 3.5 erklärt weitere Konfigurationsmöglichkeiten und beschreibt präziser mögliche Optionen.

Der Programmcode, welcher ursprünglich für die Standardkonfiguration des DSPs generiert wurde, kann für die optimierte DSP-Struktur nicht mehr verwendet werden und muss daher neu erzeugt werden. Der Assembler liest diesmal das VHDL-Konfigurationspaket der optimierten DSP-Struktur und generiert für diese den Programmcode. Eine letzte Simulation bestätigt die korrekt verlaufene Optimierungsphase.

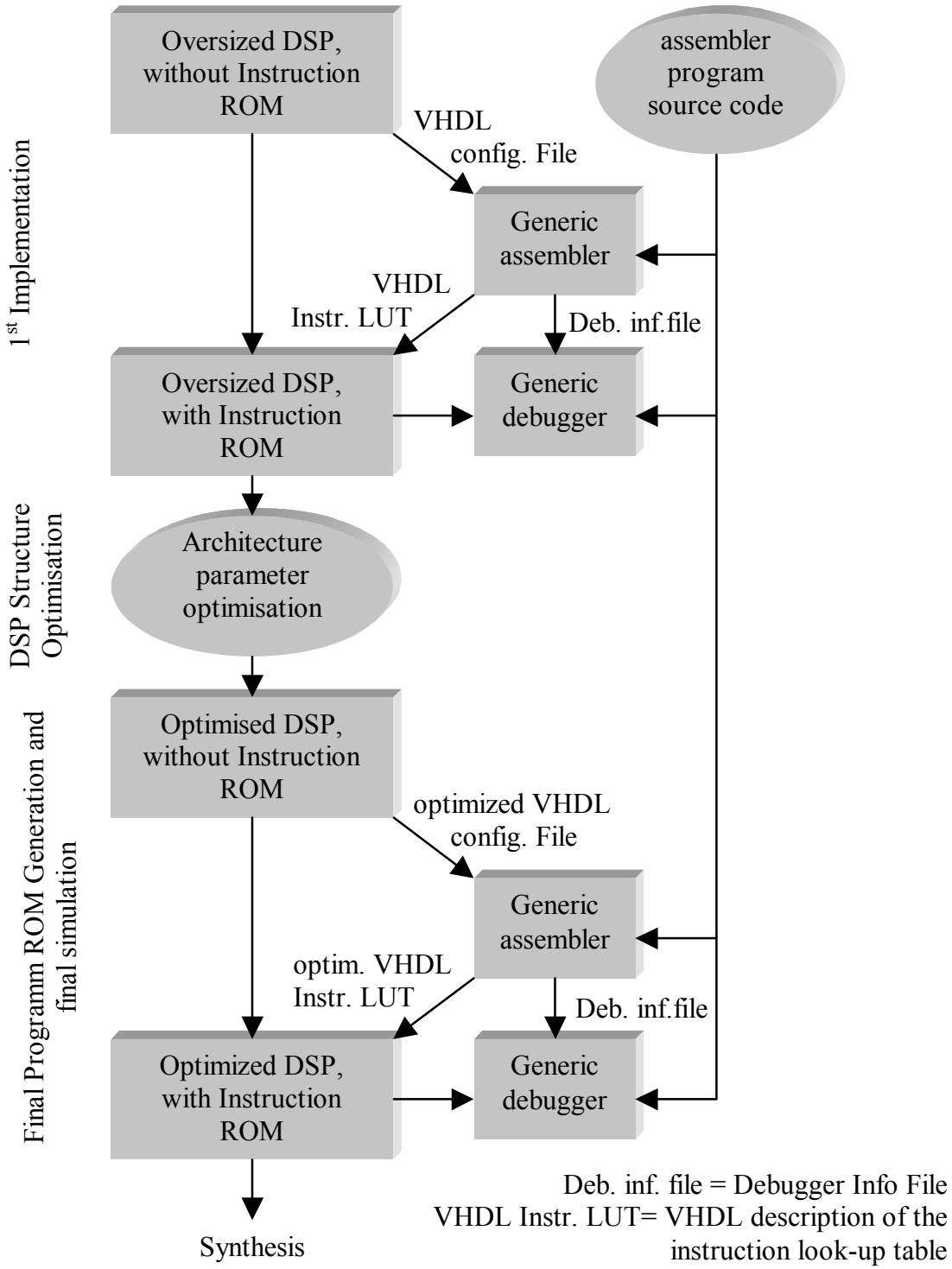


Abbildung 5: Der Designfluss der Designmethode

Die Applikation ist nun vorbereitet für ihre Verwendung. Sie kann zum Beispiel als Block in ein Design eingefügt oder synthetisiert werden.

Die erwartete Qualität eines Designs, für dessen Realisation die vorgestellte Designmethode verwendet wurde, wird in zwei Schritten erzielt. Die Optimierungsphase des Designflusses bringt normalerweise eine deutliche Verringerung der Fläche und indirekt damit auch eine Verbesserung der

Verlustleistung und der Geschwindigkeit. Andererseits bringt bereits die Wahl der Grundstruktur des DSP wichtige Eigenschaften mit sich. So kann für Low-Power-Anwendungen eine Grundstruktur verwendet werden, welche spezielle *Low-power*-Techniken einsetzt und speziell für diesen Anwendungsbereich optimiert ist. Wenn andere Qualitätsmerkmale wie Fläche, Geschwindigkeit, usw. Wichtigkeit besitzen, können andere, jeweils für diese Bereiche optimierte DSP-Strukturen verwendet werden (Abbildung 6).

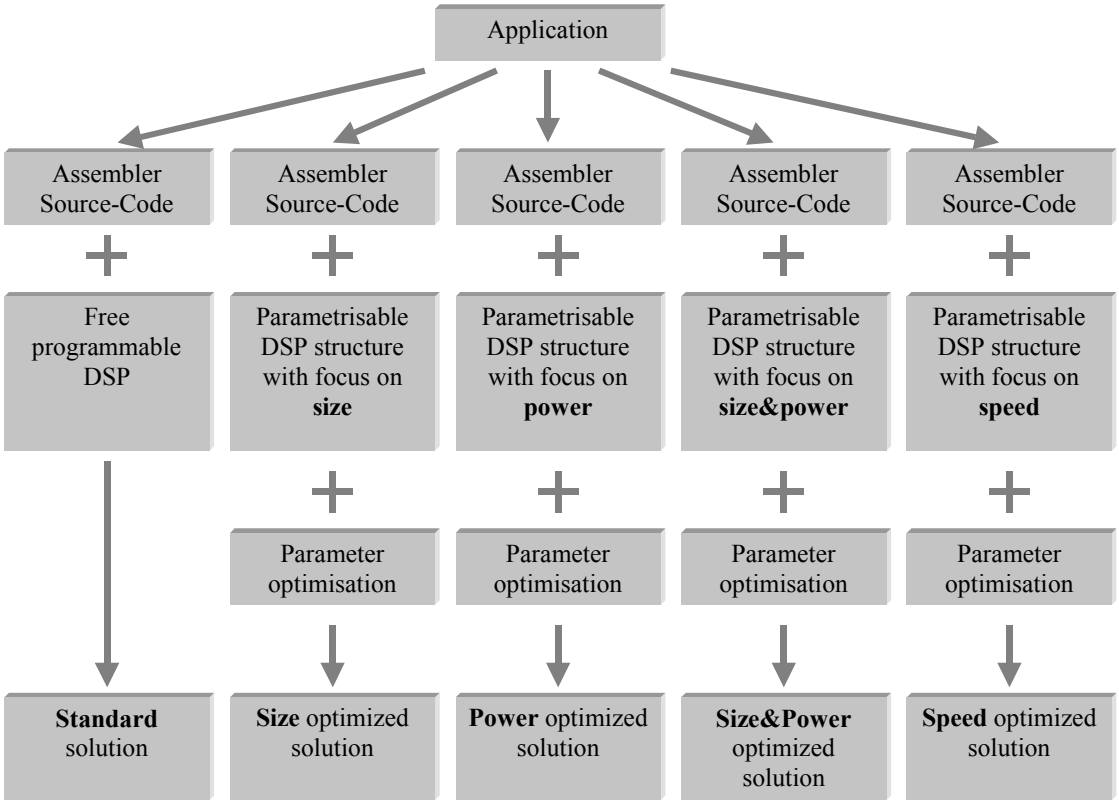


Abbildung 6: Die verschiedenen Anwendungsbereiche, wo die Designmethode eingesetzt werden kann

3.5 Der HotDSP: Eine low-power DSP-Architektur für die Hardware/Software - Co-Designmethode

Um die vorgestellte Designmethode praktisch zu entwickeln, zu testen und zu verwenden, wurde die sogenannte HotDSP[1]-Familie entwickelt. Die HotDSPs, welche für Low-Power-Filteranwendungen optimierte, programmierbare Spezialarchitekturen besitzen, sind mit über 130 Parameter detailliert parametrisierbar, was sie optimal an Anwendungen anpassen lässt. Sie wurden in technologieunabhängigem VHDL implementiert, was Ihre Verwendung für ASIC und FPGA ermöglicht. *Low-power*-Techniken wie

[1] *HotDSP* ist ein willkürlich gewählter Name

zum Beispiel *Gated Clocks* wurden eingesetzt, um optimale Resultate im Low-Power-Bereich zu erzielen, wobei die Modifikation eines einzigen Parameters genügt, um eine synchron getaktete Lösung zu erhalten, was die Realisation von FPGA-Prototypen beschleunigt (FPGA bieten normalerweise keine Möglichkeit, korrekt *Gated Clocks* zu implementieren).

Der erste Teil dieses Unterkapitels beschreibt die Hardware der HotDSPs mit deren Parametriermöglichkeiten, während der zweite Teil kurz ein Auge auf die Entwicklungssoftware, bestehend aus Assembler und Debugger, wirft.

3.5.1 Hardware

Der HotDSP kennt drei Grundarchitekturen um verschiedenen Leistungsansprüchen gerecht zu werden. Auf Top-Level-Niveau sind die drei Architekturen kompatibel, da ihre Interfaces identisch parametrisiert werden können. Ebenfalls intern besitzen die drei Typen viele gemeinsame Eigenschaften.

Unidirektionelle Eingangs- und Ausgangs-Datenports, ein- und ausgehende Kontrollsignale, sowie Reset- und Clocksignale bilden die Schnittstelle des HotDSPs. Die Anzahl der verwendeten Ports und der Kontroll- und Clocksignale ist konfigurierbar. Die eingehenden Kontrollsignale können so konfiguriert werden, dass sie entweder den im Sleep-Modus befindenden HotDSP aufwecken oder als Entscheidungssignale für die konditionellen Sprunginstruktionen und damit zur Programmflusskontrolle zur Verfügung stehen. Die ausgehenden Kontrollsignale vermitteln Informationen über den Status des DSPs und darüber, wann die verarbeiteten Daten am Ausgang bereitstehen.

Klassifizierung der HotDSP-Architektur

Wegen den verwendeten Spezialarchitekturen und den Parametrisiermöglichkeiten ist eine Klassifizierung des HotDSPs gemäss Kapitel 2.3 nur teilweise möglich. Zu einer generellen Einordnung kann folgendes gesagt werden: Die HotDSPs besitzen Register-basierte Architekturen mit Instruktionssets, welche je nach Parametrisierung sehr anwendungsspezifisch sind (ASIP[1]). Jeder Taktzyklus wird mit der Verarbeitung einer neuen Instruktion begonnen, welche direkt, also nicht via Mikroprogrammierung, verarbeitet wird. Kontrollinstruktionen werden in einem einzigen Taktzyklus, Datenverarbeitungsinstruktionen mittels einer 4-stufigen *Execution-Pipeline* in 4 Taktzyklen verarbeitet. Der HotDSP-Typ 1

[1] *ASIP: Application Specific Instruction set Processor*

besitzt eine SISD-Architektur, während die HotDSP-Typen 2 und 3 VLIW-Architekturen haben, deren Instruktionsspeicher auf lokale Speicher aufgeteilt sind. In der nun folgenden Architekturbeschreibung soll detaillierter auf die drei HotDSP-Typen eingegangen werden.

Detaillierte Beschreibung der HotDSP-Architektur

Jeder HotDSP-Typ ist in zwei Hauptblöcke aufgeteilt, und zwar in die Kontroll- und die Datenverarbeitungseinheit (Abbildung 7). Die Kontrolleinheit mit integriertem Programm-LUT steuert den Programmablauf des DSPs. Sie teilt das aktuelle Instruktionswort in die für die Datenverarbeitungseinheit bestimmten Anteile und die Kontrollanteile auf. Die ersten werden zusammen mit zusätzlichen Informationen der Datenverarbeitungseinheit zur Verfügung gestellt, während die letzteren direkt von der Kontrolleinheit selbst verarbeitet werden.

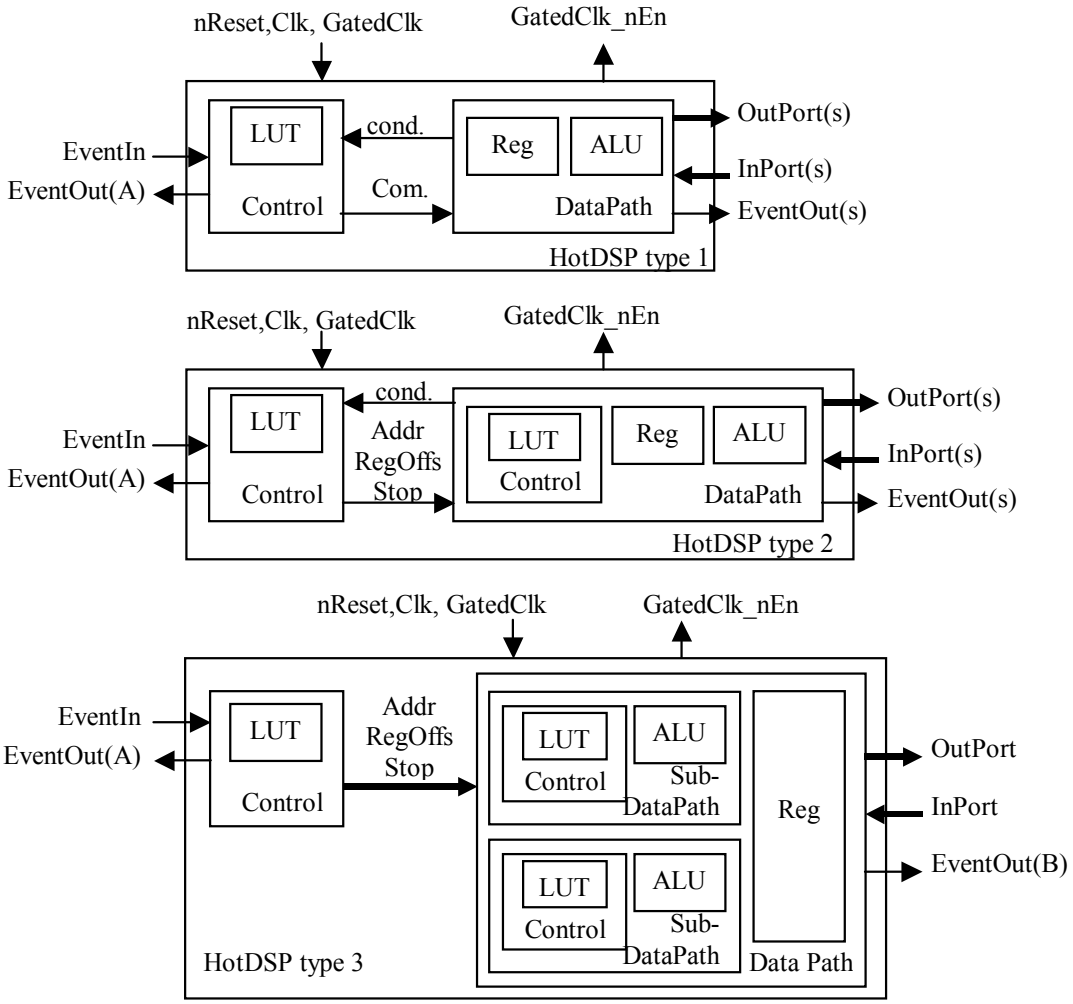


Abbildung 7: Der strukturelle Aufbau der 3 Architekturtypen des HotDSPs

Die Datenverarbeitungseinheit ist im einfachsten Fall (Typ 1) in einen Registerblock und eine arithmetische Einheit aufgeteilt und wird vollständig

von der Kontrolleinheit kontrolliert. Genügt die Verarbeitungsgeschwindigkeit des ersten HotDSP-Typs nicht für die Zielapplikation, so muss der zweite Typ verwendet werden. Die Datenverarbeitungseinheit dieses Typs besitzt einen zusätzlichen Kontroller mit einem integrierten LUT zur Speicherung der arithmetischen Instruktionen, welche nun nicht mehr im LUT des Hauptkontrollers gespeichert werden müssen. Dieser Architekturtyp mit zwei LUTs erlaubt ein paralleles Verarbeiten von Kontroll- und arithmetischen Instruktionen, was je nach Anwendung zu einer wichtigen Leistungserhöhung führen kann. Da die Adressierung beider LUTs synchronisiert ist, d.h. über den selben Adressbus geschieht, können die beiden LUTs auch als dezentralisierte Teile eines grösseren LUTs betrachtet werden. Der dritte Architekturtyp des HotDSPs kann eine konfigurierbare Anzahl von arithmetischen Einheiten besitzen, welche alle durch einen eigenen Kontroller mit integriertem Programm-LUT gesteuert werden, sich den Registerblock jedoch teilen. Mögliche Mehrfachzugriffe seitens der Datenpfade auf das gemeinsame Registerfile müssen dabei vom Anwender kontrolliert werden. Wie beim Typ 2 sind wiederum sämtliche LUTs synchronisiert.

Wegen der gemeinsamen Implementierungstechniken der drei Architekturtypen soll im Folgenden lediglich der erste HotDSP-Typ besprochen werden, mit einigen Ergänzungen zu den anderen beiden Typen.

Die Ressourcen zur Adressierung des Programmspeichers bilden mit diesem zusammen den grössten Teil der Kontrolleinheit. Das Adressregister registriert jeden Zyklus einen neuen Adresswert, welcher von einem Multiplexer selektiert wird. Dieser neue Wert entspricht entweder dem Inkrement der momentanen Adresse (normaler Programmfluss), wird vom aktuellen Instruktionswort geliefert (Verzweigung), wird vom Adressstack geliefert (Rücksprung von einem Unterprogramm) oder wird ganz einfach auf Null gesetzt (synchrones Reset). Der Adressstack speichert beim Aufruf eines Unterprogramms die Rücksprungadresse und ermöglicht auf diese Weise die Verwendung von Funktionen und Prozeduren innerhalb eines Programms. Als Programmspeicher werden synthetisierbare VHDL-Beschreibungen von LUTs, welche den Programmcode enthalten, verwendet.

Eine konfigurierbare Anzahl von Steuersignalen (Event-Signale genannt), welche entweder extern definiert werden oder von der Datenverarbeitungseinheit stammen und optional verzögert werden können, bestimmen die Verzweigungen des Programmflusses und können den deaktivierten DSP wieder aktivieren.

Nebst den Signalen zur Kontrolle der Datenverarbeitungseinheit erzeugt die Kontrolleinheit auch die Adressen der Quell- und Zielparameter der arithmetischen Instruktionen. Dies können Zugriffe auf Register, Input- und

Output-Ports, sowie Koeffizientenspeicher sein. Um keine Arbeitszyklen für eine Parameterübergabe beim Aufruf von Unterprogrammen zu verlieren, bedient sich der HotDSP einer speziellen Technik: Die Parameteradressen einer arithmetischen Instruktion werden innerhalb der Kontrolleinheit optional mit einem Offsetwert kombiniert, welcher vorausgehend von den Verzweigungsinstruktionen definiert werden kann. Soll nun ein Unterprogramm mehrmals aufgerufen werden, so kann mittels verschiedenen Offsetwerten auf verschiedene Quell- und Zieldaten zugegriffen werden. Die Information, ob die Adressen mit einem Offset versehen werden, und welcher Offsetwert verwendet werden soll, ist direkt in die arithmetischen Instruktionen eingebunden. Diese etwas abstrakte Darstellung der Offsetkombination wird im Anwendungsbeispiel am Ende dieses Kapitels in einer anschaulicheren Weise erklärt.

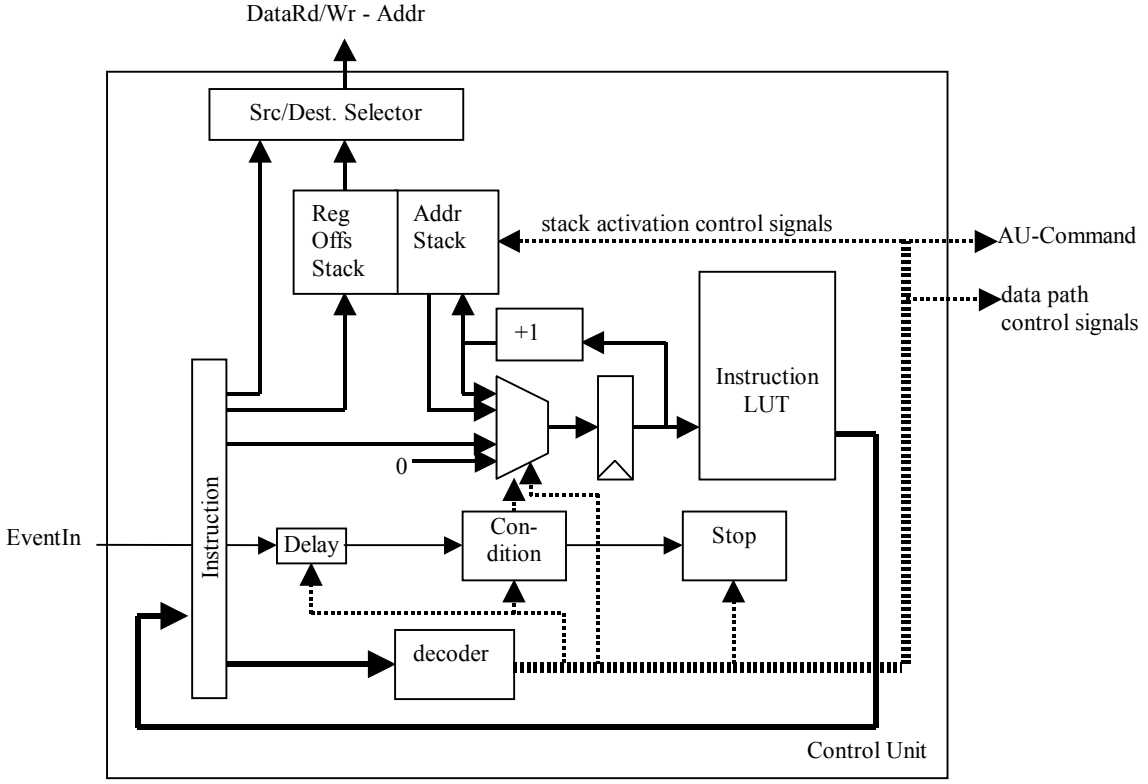


Abbildung 8: Die Kontrolleinheit der HotDSPs vom Typ 1

Die Datenverarbeitungseinheit besteht aus der Registereinheit, der arithmetischen Einheit und einem kleinen LUT für Koeffizienten, welche meistens für Multiplikationen gebraucht werden.

Die zu verarbeitenden Daten, welche aus zwei Datenwörtern, einem Koeffizienten, einem zurückgeführten Datenwert und den Kontrollsignalen bestehen, werden zuerst in den Eingangsregister der *AU* gespeichert und werden dann in einer ersten Verarbeitungsetappe vom *AU-Core* bearbeitet. Dieser enthält die VHDL-Beschreibung aller arithmetischen Operationen, welche für eine Anwendung benötigt werden und muss deshalb meist als

einziger Block mit ein wenig „Handarbeit“ an die Applikation angepasst werden. Dies bedeutet jedoch keinen grossen Aufwand, da für arithmetische Operationen standardisierte, sehr einfach zu verwendende VHDL-Bibliotheken existieren, welche unterdessen gute Resultate bei einer Synthese erzielen.

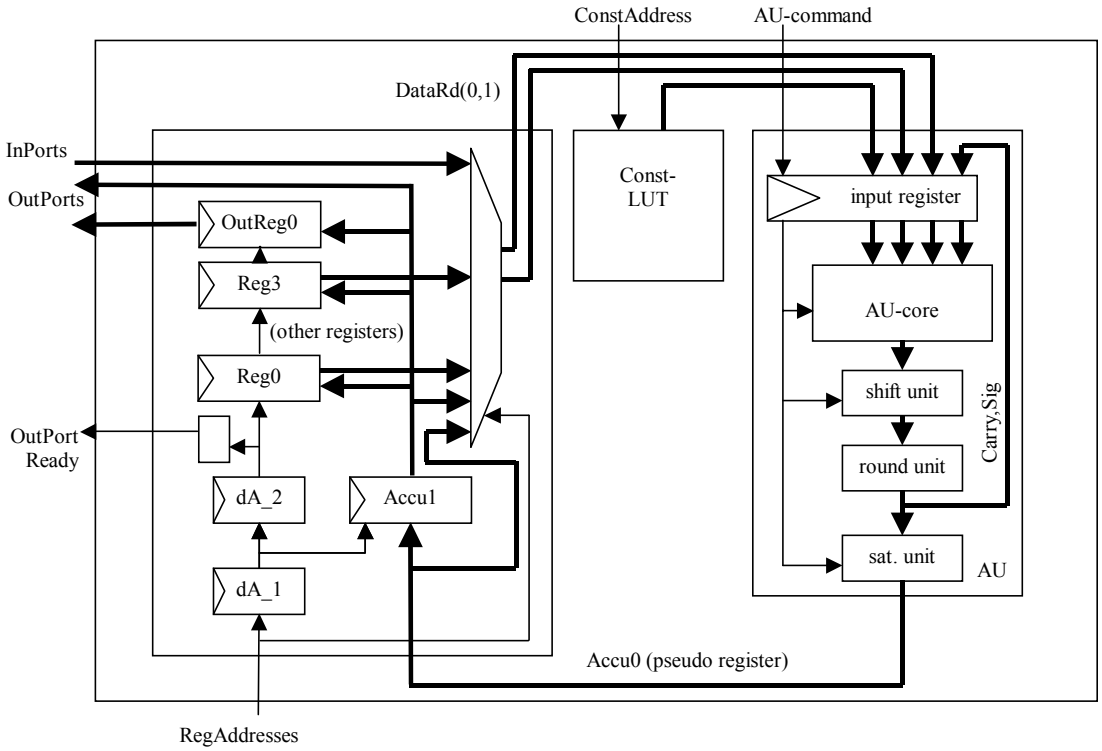


Abbildung 9: Die Datenverarbeitungseinheit des HotDSPs vom Typ 1

Es folgen hintereinander drei automatisch generierte Einheiten zur Nachbearbeitung des vom *AU-Core* erhaltenen Resultates. Dies sind ein Shifter, ein Block zum Runden des Resultates und ein weiterer Block um bei einem Bereichsüberlauf des Resultates mit einer Sättigung eine Korrektur vorzunehmen. Mit dem Wort „automatisch“ ist gemeint, dass die möglichen Nachbearbeitungen der Daten lediglich mit Parametern spezifiziert werden müssen. Mit dem Konzept der Nachverarbeitungsblöcke werden die Aufgaben des *AU-Cores* auf ein Minimum reduziert.

Das Resultat der arithmetischen Einheit kann von Benutzerseite als Wert eines Pseudoregisters betrachtet werden (Akkumulator-Register). Pseudoregister deshalb, da kein eigentliches Register vorhanden ist und nur aus Programmierersicht der Zustand des Accu0-Busses als Wert eines Ausgangsregisters der arithmetischen Einheit betrachtet werden kann.

Die Registereinheit kontrolliert die Versorgung der arithmetischen Einheit mit Quelldaten und speichert die via Akkumulatorregister zurückkommenden, bearbeiteten Daten. Als Datenquellen können sämtliche Register und die Input-Ports dienen. Via zwei Multiplexer werden die

beiden Quelldaten selektiert. Die zurückkommenden Resultate können ein optional zusätzliches Zwischenregister passieren, bevor sie im Zielregister gespeichert werden oder einem Output-Port zugeführt werden. Dieses Zwischenregister, im Folgenden Akkumulator 1 genannt, kann zwei interessante Vorteile mit sich bringen: Erstens verringert es die Kapazität des Akkumulators 0, welche je nach der Anzahl der verwendeten Register und Ausgangsports sehr hoch sein kann. Da sich der Akkumulator 0 am Ende einer langen Verarbeitungskette befindet, ist seine Parasitaktivität besonders hoch. Dies kann, wenn die Leitungen des Akkumulators 0 zusätzlich auch noch eine erhöhte Kapazität besitzen, zu einem nicht zu unterschätzenden Stromverbrauch führen. Zweitens erlaubt dieses Register unter bestimmten Einschränkungen ein lokales Speichern eines zweiten Zwischenresultates, ohne dass die eigentlichen Register verwendet werden müssen.

Die durch die Verwendung zweier Akkumulatoren entstandene Datenpipeline muss im Kontrollteil des Registerblockes kompensiert werden. Deshalb durchqueren die Zieladressen ebenfalls zwei Register, bevor sie ausgewertet werden um die verschiedenen Register zu steuern.

Die Output-Ports können ungepuffert oder mit einem Register gepuffert sein. Im ersten Fall entspricht der Wert am Output-Port dem Wert von Akku 1 und besitzt dadurch generell lediglich während eines Zyklus Gültigkeit. Jeder Output-Port, ob gepuffert oder ungepuffert, wird automatisch durch ein Ready-Signal begleitet, welches während des Schreibzyklus des Portes aktiviert wird. Dies vereinfacht die Einbindung des HotDSPs in seine Zielumgebung.

Eine letzte Technik welche im Registerblock eingesetzt wird, sind die sogenannten Alias-Output-Ports (in Abbildung 9 nicht gezeichnet). Man stelle sich einen Filteralgorithmus vor, welcher zwei verschiedene Resultate bildet, welche jedoch vom HotDSP über den selben Output-Port ausgegeben werden sollen. In diesem Fall kann die Integration des HotDSPs in die Zielumgebung vereinfacht werden, wenn vom HotDSP ein Signal zur Verfügung gestellt wird, welches spezifiziert, welches der beiden Resultate am Output-Port anliegt. Ein sogenannter Alias-Output-Port ist kein reeller, neuer Output-Port, sondern erteilt einem existierenden eine zweite Zieladresse. Er besitzt jedoch ein eigenes Ready-Signal, welches aktiviert wird, wenn ein Port über seine Alias-Adresse angesprochen wird. Auf diese Weise kann der HotDSP Daten jeweils über einen einzelnen, reellen Port ausgeben und doch spezifizieren, welches Resultat am Ausgang anliegt.

Instruktionen, Verarbeitungssequenz und Pipeline

Der HotDSP besitzt lediglich 3 Instruktionstypen: Die arithmetischen Instruktionen, die Verzweigungsinstruktionen und NOP[1], wobei eine Verzweigung ein Funktionsaufruf (CALL) oder ein Sprung (JUMP) sein kann und ebenfalls die arithmetischen Instruktionen einer parametrisierbaren Menge von Operationen entsprechen kann. Drei Bits, welche gemeinsam alle drei Instruktionstypen besitzen, ermöglichen eine direkte Ausführung der am häufigsten verwendeten Programmoperationen, ohne dass zusätzliche Instruktionen und Arbeitszyklen benötigt werden. Im Zusammenhang mit diesen Bits wird auch die NOP-Instruktion, die sonst lediglich einen unproduktiven Zyklus produziert, sinnvoll.

```
Allgemein:  -0---------5--------10--------15--------20--------25--------30---
   Jump0    |x| | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
   Stop     | |x| | | | | | | | | | | | | | | | | | | | | | | | | | | | |
   Ret      | | |x| | | | | | | | | | | | | | | | | | | | | | | | | | | |

Arith.      -0+-+-+-+-+5+-+-+-+-10+-+-+-+-15+-+-+-+-20+-+-+-+-25+-+-+-+-30+--
Instrukt:   | | | |x| | | | | | | | | | | | | | | | | | | | | | | | | | |
   Type     | | | | |x|x|x| | | | | | | | | | | | | | | | | | | | | | | |
   Out      | | | | | | | |x|x|x|x| | | | | | | | | | | | | | | | | | | |
   Cst      | | | | | | | | | | | |x|x|x| | | | | | | | | | | | | | | | |
   Reg1     | | | | | | | | | | | | | | |x|x|x|x|x|x| | | | | | | | | |
   Reg2     | | | | | | | | | | | | | | | | | | | | |x|x|x|x|x|x| | | | |
   Dest     | | | | | | | | | | | | | | | | | | | | | | | | | | |x|x|x|x|x|x|

Verzweig-   -0+-+-+-+-+5+-+-+-+-10+-+-+-+-15+-+-+-+-20+-+-+-+-25+-+-+-+-30+--
Instrukt:   | | | |x|x| | | | | | | | | | | | | | | | | | | | | | | | | |
   Type     | | | | | |x| | | | | | | | | | | | | | | | | | | | | | | |
   PipeStp  | | | | | | |x| | | | | | | | | | | | | | | | | | | | | | | |
   Cond     | | | | | | | |x| | | | | | | | | | | | | | | | | | | | | | |
   Addr0    | | | | | | | | |x|x|x|x|x| | | | | | | | | | | | | | | | | |
   RgOffs0  | | | | | | | | | | | | | |x|x| | | | | | | | | | | | | | |
   Addr1    | | | | | | | | | | | | | | | |x|x|x|x|x| | | | | | | | | |
   RgOffs1  | | | | | | | | | | | | | | | | | | | | |x|x| | | | | | | |
   Addr2    | | | | | | | | | | | | | | | | | | | | | | |x|x|x|x|x| | | | |
   RgOffs2  | | | | | | | | | | | | | | | | | | | | | | | | | | |x|x| | | |
   RgOffs3  | | | | | | | | | | | | | | | | | | | | | | | | | | | | |x|x| |

NOP-        -0+-+-+-+-+5+-+-+-+-10+-+-+-+-15+-+-+-+-20+-+-+-+-25+-+-+-+-30+--
Instrukt:   | | | |x|x| | | | | | | | | | | | | | | | | | | | | | | | | |
            -0---------5--------10--------15--------20--------25--------30---
```

Listing 2: Die Bitfelder der einzelnen Instruktionen

Listing 2 zeigt die einzelnen Bitfelder der Instruktionen. Alle Felder existieren in jeder Konfiguration des HotDSPs, die einzelnen Grössen können jedoch an die Zielanwendung angepasst werden. Die in der Tabelle

[1] NOP: *No Operation*

gezeigten Bitfeldgrössen entsprechen dem am Ende dieses Kapitels folgenden, ersten Anwendungsbeispiel.

In jedem Instruktionstyp sind die ersten drei Bits reserviert und dienen zur effizienten Implementierung der folgenden Programmoperationen: Sprung auf den Programmbeginn (synchrones Reset), Stoppen des Prozessors, Rücksprung aus einer Unterfunktion.

Die Präsenz einer arithmetischen Instruktion wird von einem ersten Bitfeld definiert und hat in diesem Beispiel lediglich ein einzelnes Bit. Das nächste Feld definiert die zu verwendende arithmetische Operation (Addition, Multiplikation, etc.), gefolgt von einem Feld zur Kontrolle der Einheiten für die Nachbearbeitung der Daten. Vier Bitfelder zur Adressierung der Quelldaten, des Koeffizienten und des Zielregisters komplettieren die arithmetische Instruktion.

Die Verzweigungsinstruktionen werden in diesem Beispiel durch ein Feld von zwei Bits definiert. Das darauf folgende Bit präzisiert, ob es sich um einen Sprung oder einen Aufruf eines Unterprogramms handelt (je nachdem wird der Adressstack betätigt). *PipeStop* definiert, ob die Datenpipeline der Datenverarbeitungseinheit während dem Durchführen der Verzweigungsinstruktionen angehalten werden soll und kann eine Algorithmusimplementierung optimieren. Das nächste Feld definiert die anzuwendende Verzweigungsbedingung. Je nach Bedingungszustand wird der Programmzähler auf eine von drei Zieladressen, welche je mit einem Adressfeld definiert sind, gesetzt oder aber einfach um eins erhöht. Jede der vier neuen Adressen ist durch ein Feld begleitet, welches einen neuen Offsetwert für die Registeradressen enthält.

Abbildung 10 zeigt die Verarbeitungssequenz der Verzweigungs- und der arithmetischen Instruktionen. Im ersten Taktzyklus wird das vom Programmspeicher gelieferte Instruktionswort dekodiert und ausgewertet. Für die Verzweigungsinstruktion bedeutet dies, dass die Bedingung einer Verzweigung evaluiert und anschliessend die Quelle der neuen Adresse selektiert wird.

Eine arithmetische Instruktion selektiert in diesem Zyklus die Datenquellen und aktiviert die zur arithmetischen Einheit führenden Konstanten- und Datenbusse. Im nächsten Zyklus werden diese Daten erst in die Eingangsregister der arithmetischen Einheit gelesen, und dann von dieser verarbeitet und via Akkumulator 0 dem Registerblock zur Verfügung gestellt. Im dritten Zyklus wird das Datenwort vom Akkumulator 1 gelesen und im vierten und letzten Zyklus ins Zielregister geschrieben.

Die Tatsachen, dass die bearbeiteten Daten nicht bereits nach einem Zyklus im Zielregister abgelegt sind, sondern erst zwei Zyklen später, muss beim

Programmieren des HotDSPs berücksichtigt werden. So kann eine arithmetische Operation das Resultat der unmittelbar vorausgehenden arithmetischen Operation nicht im Zielregister holen, sondern muss auf den Akkumulator 0 zugreifen. Einen Zyklus später wird das Resultat in Akkumulator 1 angetroffen, und erst wiederum einen Zyklus später befindet es sich im Zielregister. Von dieser Pipeline kann für ein kurzfristiges Speichern von Zwischenresultaten profitiert werden indem als Datenziel einer arithmetischen Instruktion nicht ein Register sondern einer der beiden Akkumulatoren adressiert wird. Die Datenpipeline wird gestoppt, sobald das Resultat den Akkumulator 0 bzw. 1 erreicht hat.

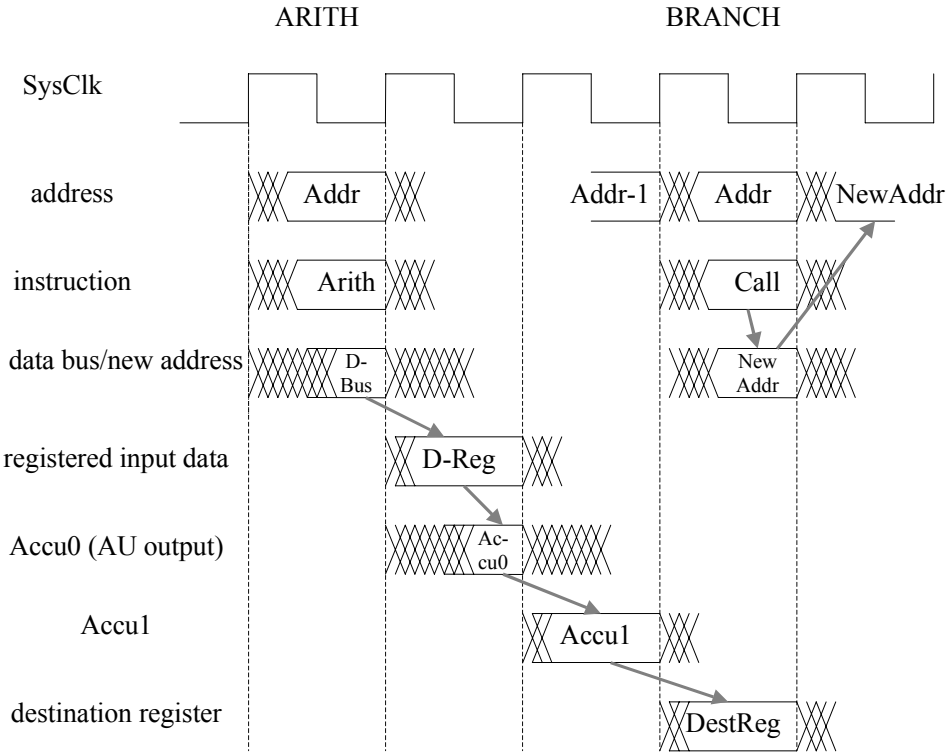


Abbildung 10: Sequenz der Instruktionsverarbeitung

Die Parameter des HotDSPs

Mit über 130 Parametern kann der HotDSP optimal für eine Applikation konfiguriert werden. Eine Beschreibung dieser Parameter sowie ein Listing eines VHDL-Definitionspaketes, in dem all die Parameter definiert werden, ist in Anhang I ab Seite A zu finden. Kurz gesagt können all die Parameter in folgende vier Gruppen eingeteilt werden:

- Die Systemparameter definieren die Schnittstelle und die Verwendung von *Gated Clocks*.
- Mit den an die Kontrolleinheit zugewiesenen Parametern können die Kontrolleinheit-internen Ressourcen konfiguriert werden.

- Die Parameter für den Registerblock spezifizieren die Eigenschaften der Datenbusse, Register und Ports.
- Die Parameter der arithmetischen Einheit spezifizieren die Datenformate innerhalb der arithmetischen Einheit, die möglichen Datennachbearbeitungen und die Menge der möglichen arithmetischen Operationen.

Low-Power-Techniken

Reduzierung der benötigten Ressourcen, Minimierung der totalen Systemaktivität, Zyklus-effiziente Gestaltung der Instruktionen und zeitliche Kürzungen der kritischen Signalpfade für die Verwendung tiefer Speisespannungen sind die beim HotDSP verwendeten Techniken, um die notwendige Energie zur Verarbeitung einer Aufgabe zu minimieren, wie folgende Aufzählung detaillierter zeigt.

- Timing-Optimierung für den Gebrauch von tieferen Speisespannungen[1]:
  - Der Gebrauch einer Pipeline reduziert die Signalverzögerungen innerhalb der Datenverarbeitungseinheit.
- Reduktion der Energie für eine Aufgabe
  - Das Instruktionsset ist klein aber sehr effizient (Optimierung der Anzahl der Arbeitszyklen, siehe die Anwendungsbeispiele am Ende dieses Kapitels).
- Reduktion der Gesamtaktivität[2]:
  - Die hohe Parametrisierbarkeit des HotDSPs erlaubt es, sämtliche redundanten und unbenutzten Ressourcen zu entfernen sowie optimale Datenwortbreiten zu verwenden.
  - Der Gebrauch von einer konfigurierbaren Anzahl von *Gated Clocks* für Kontrolleinheit, Registerblock und arithmetischen Einheit reduziert die totale Aktivität des HotDSPs.
  - Im Sleep-Modus wird lediglich ein D-Flipflop getaktet (plus optional die gepufferten Event-Signale).
  - Die Verwendung des zweiten Akkumulators reduziert die „toggelnde“ Ausgangskapazität von Akkumulator 0.
  - Eingangsregister der arithmetischen Einheit vermeiden ein Toggeln der arithmetischen Logikkomponenten während des Daten-*Fetches*.

[1] Formel 4) in Kapitel 2.2
[2] Formel 3)

Spezielle Eigenschaften des HotDSPs, welche eine effiziente Filterimplementierung erlauben

Da ein Programm eines Filteralgorithmus häufig die selben Abläufe durchführen muss, wurde bei der Entwicklung der DSP-Architektur darauf geachtet, dass diese Abläufe ohne Aufwand, das heisst ohne zusätzliche Instruktionen realisiert werden können. Dies, zusammen mit der einfacher Portierung von VHDL auf die verschiedensten Technologien bilden die speziellen Merkmale zur Vereinfachung einer Filterimplementierung, wie die folgende Auflistung zeigt:

- Da sämtliche Instruktionen drei Bits für die von Programmen am häufigsten verwendeten Kontrollinstruktionen besitzen, können diejenigen gleichzeitig mit einer anderen Instruktion verarbeitet werden, was den Gewinn von Arbeitszyklen zur Folge hat.
- Jede Verzweigungsinstruktion hat 4 Zieladressen und nicht wie normale DSPs lediglich 2. Multi-Programmverzweigungen können dadurch 2 mal schneller durchgeführt werden als üblich.
- Die arithmetischen Instruktionen enthalten sämtliche Informationen über die Konstanten- und Datenquellen, das Zielregister des berechneten Resultates, die durchzuführende arithmetische Operation und die anschliessende Nachverarbeitung des erhaltenen Resultates. Durch die Datenpipeline kann der HotDSP jeden Takt zwei Datenwörter und einen Koeffizienten lesen, eine arithmetische Operation mit anschliessender Nachbearbeitung durchführen, und das Resultat in ein Zielregister schreiben. Andere DSPs benötigen dazu wegen der Load-/Store-Technik verschiedene Takte.
- Die automatische Begleitung jedes Output-Ports von einem Ready-Signal und die Möglichkeit der Verwendung der Alias-Ports verhindert, dass zusätzliche Zyklen zur Erzeugung von Kontrollsignalen benötigt werden.
- Die konfigurierbare Verzögerung der eingehenden Kontrollsignale ermöglicht eine einfache Adaptation des Kontrollinterfaces an eine Zielumgebung.
- Da zur Implementierung VHDL verwendet wurde, lässt sich der HotDSP in sämtlichen ASIC-Technologien integrieren.
- Ein einziger Parameter muss geändert werden, damit der gesamte HotDSP synchron getaktet wird, was ermöglicht, dass er sich ebenfalls leichter auf ein FPGA programmieren lässt.

3.5.2 Software

Für den HotDSP steht ein Assembler und ein Debugger zur Verfügung. Diese beiden Tools lassen sich in Sachen Leistungsumfang sicherlich nicht mit kommerziellen Produkten vergleichen. Für die Implementierung von Filteralgorithmen sind sie jedoch ausreichend.

Der Assembler

Der Assembler liest einen Quellencode und packt, unter Berücksichtigung der gewählten DSP-Konfiguration, den binären Programmcode direkt in eine synthetisierbare VHDL-Beschreibung eines LUTs. Um die aktuelle DSP-Konfiguration zu erfahren, ist er imstande, das VHDL-Definitionspaket mit all seinen Parametern zu lesen und zu interpretieren. Auszüge eines Quellencodes (Listing 3) und eines VHDL-Definitionspaketes (Listing 4) helfen, die Arbeitsweise des Assemblers zu verstehen.

```
Addr0:
  arith mov nrm In0 RegU CstU Accu1
  stop
  call Event12, Adapt3(4), Adapt3(0), Adapt3(6) (2) AU_PipeStop
Adapt3:
  arith mulsub h0z0r1s19 Accu0  Reg1+ Cst1+ Reg1+
```

Listing 3: Ausschnitt eines Assembler-Quellcodes

Der Quellencode wird Zeile für Zeile gelesen, und nach folgendem Schema interpretiert:

- Folgt unmittelbar hinter dem ersten Wort einer Zeile ein Doppelpunkt („:“), so wird ein Label erkannt. Jede verwendete Adressindikation innerhalb des Quellencodes, welche dieses Label verwendet, wird zu einem späteren Zeitpunkt durch die Adressposition des Labels ersetzt.
- Enthält eine Zeile kein Label, so spezifiziert das erste Wort den Instruktionstyp.
- Enthält die Zeile eine arithmetische Instruktion, angekündigt mit dem Schlüsselwort *arith*, so definieren die 6 folgenden Wörter in folgender Reihenfolge den Typ der arithmetischen Operation, die Art der Datennachverarbeitung, die beiden Datenquellen, den Koeffizienten und schliesslich das Zielregister. Das VHDL-Paket liefert nun die notwendigen Informationen, um den korrekten Programmcode zu erzeugen. Als Beispiel soll lediglich die Übersetzung der Operation *mulsub* und des Parameters *Accu1* betrachtet werden. Der Binärwert der Operation wird vom VHDL-Paket direkt geliefert, während das Parameterfeld durch die Adresse des Akku 1 (Accu1_Addr) und seiner Formatierung (InstrArith_Dest_Size) bestimmt wird. Folgt unmittelbar hinter einem Parameter ein „+“, so bedeutet dies, das die Adresse vor Gebrauch mit dem Registeroffset kombiniert werden soll, was mit einem zusätzlich gesetzten Bit im Parameterfeld definiert wird. Die Art der Nachbearbeitung der Daten wird mit einem Code spezifiziert (h=Shift,

z=Zero, r=Round, s=Saturation). Ist keine spezielle Nachverarbeitung der Daten erwünscht, kann ganz einfach „nrm“ verwendet werden.

- Ebenfalls der Binärcode der Bedingung einer Verzweigungsinstruktion kann direkt aus dem VHDL-Paket gelesen werden (BranchCond_Event12="0"). Anstelle von Werten können die Namen von Label als Sprungadressen verwendet werden. Hinter jeder Sprungadresse kann in Klammern optional ein neuer Registeroffset definiert werden. Ebenfalls optional kann mittels dem Wort *AU_PipeStop* am Ende der Zeile definiert werden, ob die Datenpipeline während der Verarbeitung der Verzweigungsinstruktion angehalten werden soll.

```
CONSTANT BranchCond_Event12: std_logic_vector := "0";
CONSTANT ArithType_mov: std_logic_vector := "100";
CONSTANT ArithType_mulsub: std_logic_vector := "101";

CONSTANT Accu1_Addr: integer := 1;
CONSTANT InPortRegisterArray_Addr: integer_vector := (31,32);

CONSTANT InstrArith: std_logic_vector := "1";
CONSTANT InstrArith_Type_Size: integer := 3;
CONSTANT InstrArith_Out_Size: integer := AU_OutShiftR_Size+AU_Ou…
CONSTANT InstrArith_Cst_Size: integer := CstAddrWidth+1;
CONSTANT InstrArith_Reg1_Size: integer := RegAddrWidth+1;
CONSTANT InstrArith_Reg2_Size: integer := RegAddrWidth+1;
CONSTANT InstrArith_Dest_Size: integer := RegAddrWidth+1;
```

Listing 4: Ausschnitt des VHDL-Parameterpaketes

Nebst zur Erzeugung des Programmcodes kann der Assembler auch zur Ausgabe eines Reports und zum Schreiben des Informationsfiles für den Debugger verwendet werden. Der Report enthält dabei Informationen bezüglich der DSP-Konfiguration, sowie die in Listing 2 gezeigte, schematische Anordnungen der Bitfelder innerhalb der verschiedenen Instruktionen.

Das Informationsfile enthält einerseits hardwarespezifische Informationen wie Ort und Grösse der verwendeten Kontrollsignale, der Register und der Flags, aber auch programmspezifische Informationen, welche für ein Quellencode-Debugging benötigt werden. Im Speziellen sind letzteres Korrespondenzen zwischen den Programmzeilen und den Adressen der daraus entstandenen Instruktionen.

Der Debugger:

Der Debugger wurde zu Beginn ausschliesslich für den HotDSP entwickelt, wurde jedoch so universell konzipiert, dass er nun auch für andere, meist

kompliziertere DSPs verwendet wird. Sämtliche notwendigen Kenntnisse bezüglich der Architektur des DSPs bezieht er vom Informationsfile, welches vom Assembler erzeugt wird. Aufgrund dieses Files wird dann die graphische Maske für die Repräsentation der Register und der Flags entfaltet. Der Anzeige des Quellencodes des Programms mit einer Indikation, an welcher Position die Programmverarbeitung stattfindet, ermöglicht ein sehr komfortables Arbeiten. Der Anwender kann sich verschiedene Debugging-Techniken wie einem schrittweisen Abarbeiten des Programms und dem Setzen von Breakpoints bedienen.

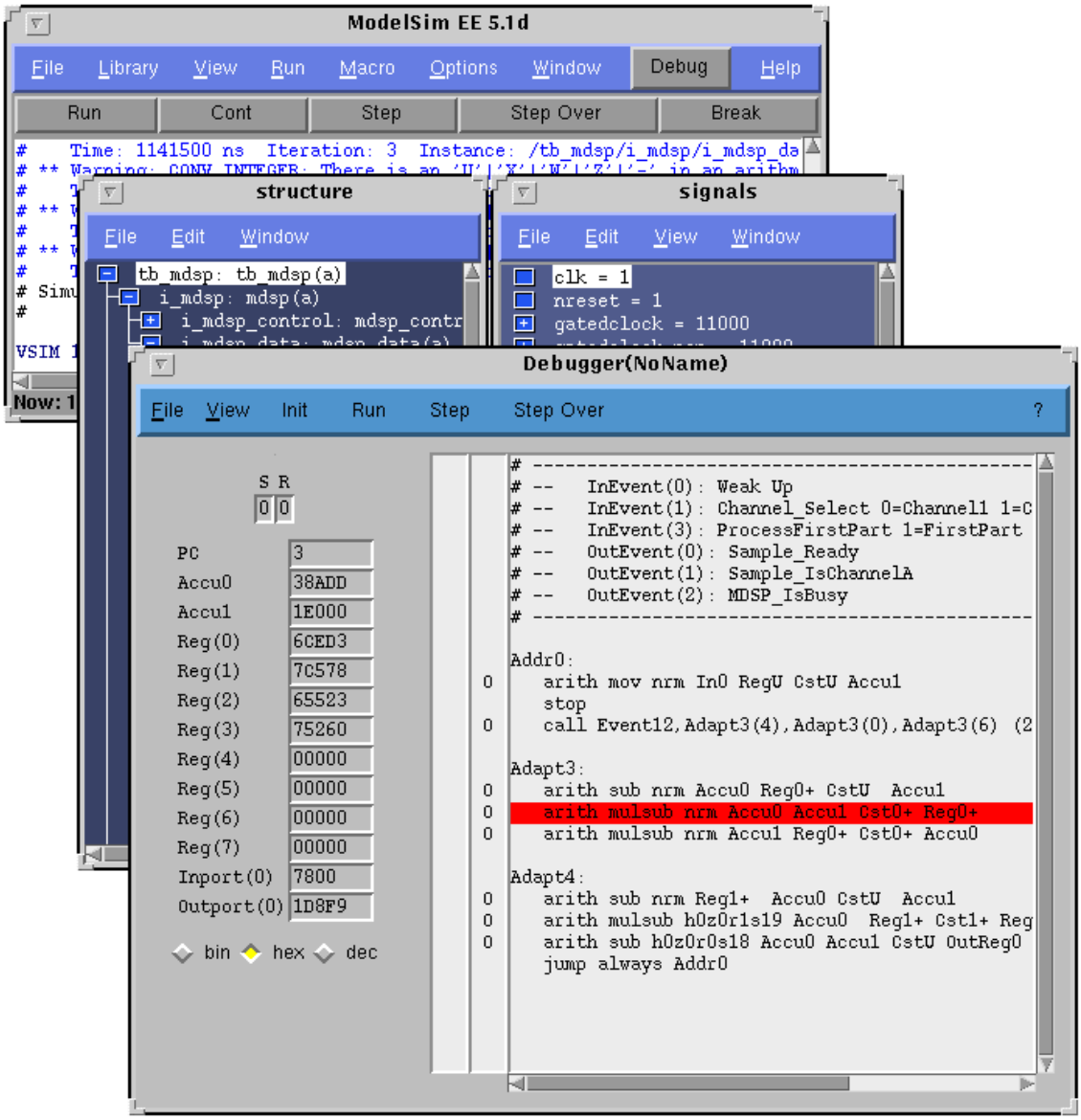


Abbildung 11: Der HotDSP- Debugger, konzipiert als Erweiterung des ModelSim VHDL-Simulators

Stellt man beim Debuggen der Software Probleme innerhalb der Hardware fest, so können zu jeder Zeit die Möglichkeiten des VHDL-Simulators

verwendet werden, um „hinter“ das Softwareniveau zu schauen um zu sehen, was auf Niveau VHDL geschieht.

3.6 Anwendungsbeispiel und Resultate

Fix zu implementierende Filterblöcke lassen sich effizient mit Hilfe der in diesem Kapitel präsentierten Co-Design-Methodik implementieren. Ein Interpolations- und Dezimationsfilter bilden je ein Anwendungsbeispiel für den HotDSP des Typs 1 und 2. Die beiden Beispiele zeigen die Einfachheit einer Filterimplementierung bei der Verwendung des HotDSPs und die zu erzielende Designeffizienz.

Implementierung eines Stereo-Interpolationsfilter unter Verwendung des ersten HotDSP-Typen

Als Stereo-Interpolationsfilter sollen *Bireciprocal Lattice Wave Digital Filters* (WDF) 9ter Ordnung dienen. Die zu verwendete Filterstruktur ist in Abbildung 12 ersichtlich [7][8][9], wobei bemerkt werden muss, dass für jeden der beiden Stereokanäle je ein solcher Filter benötigt wird.

Richtig implementierte WDF-Filter haben exzellente Filterqualitäten. Dies ist jedoch nur zu erzielen, wenn bei der Implementierung einige Eigenschaften berücksichtigt werden.

Bei einer schlechten Realisation eines WDF-Filters steckt der Wurm häufig in der Art des Rundens der Daten. Ohne gross auf die Theorie der WDF-Filter einzugehen, soll hier lediglich auf die „Passivität“ der Adapter hingewiesen werden, damit das Rundungsproblem verstanden wird. Die ursprüngliche Idee der WDF-Filter ist, die passiven LRC-Analogfilter, bestehend aus Spulen (L), Widerständen (R) und Kapazitäten (C) zu imitieren. Die Passivität dieser LRC-Filter bedeutet, dass die Energie des Ausgangssignals immer kleiner oder gleich der Energie des Eingangssignals ist. Wird nun nach einer Multiplikation ein übliches Runden der Daten vorgenommen, kann bei der Verwendung eines konstanten Eingangssignals unter Umständen auf Grund des Quantisierungseffektes eine Schwingung auf dem Ausgangssignal auftreten, was *Limit Cycle* genannt wird. Ist das Eingangssignal auf Null gesetzt und tritt dieses Schwingen am Ausgang aus, wird dies *Zero Limit Cycle* genannt und bedeutet eine Verletzung des Passivitätsgesetzes des WDF-Filters, da die Energie des Ausgangssignals gegenüber derjenigen des Eingangssignals nicht Null ist. Die Verwendung der sogenannten *Sign Magnitude Truncation*, also ein „Runden“ der Daten immer gegen Null hin, ist die Lösung des Problems, da die Amplitude des Datensignals nie erhöht wird sondern immer Energie entzogen wird. Es ist ebenfalls von Wichtigkeit, an welcher Stelle dieses *Sign Magnitude Truncation* stattfindet. Dies soll nicht unmittelbar nach der Multiplikation

stattfinden, sondern möglichst an den Ausgängen der Adapter. Dies ist der Grund, weshalb die für dieses Designbeispiel verwendete ALU des HotDSPs nicht eine Operation für eine Multiplikation, sondern eine Operation für eine kombinierte Multiplikation und Subtraktion besitzt.

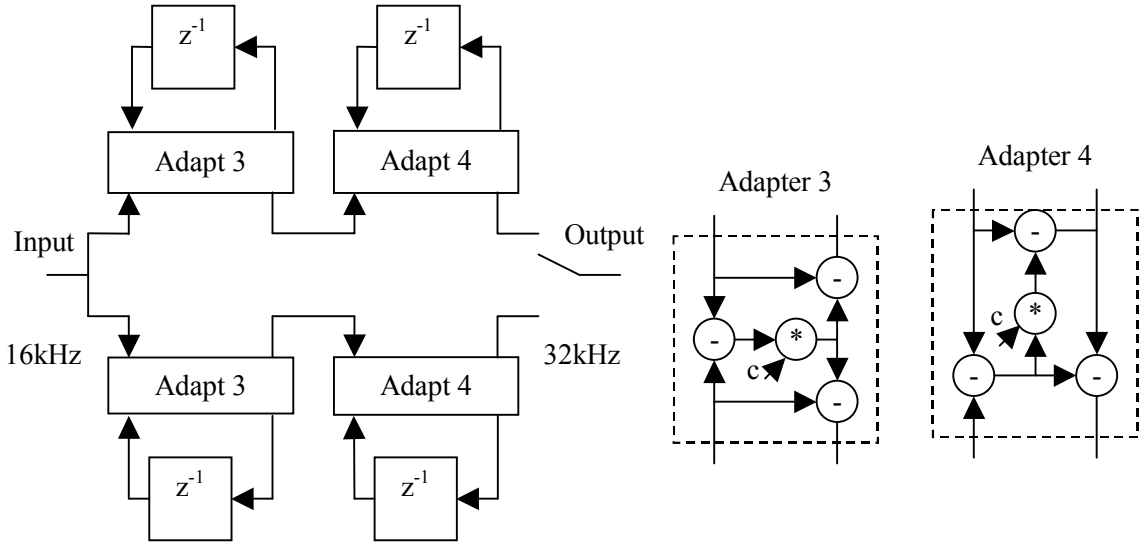


Abbildung 12: Die Struktur des Interpolationsfilter

Ob ein Adapter einen *Zero Limit Cycle* aufweist, hängt von der Struktur des Adapters sowie des für die Multiplikation verwendeten Koeffizienten ab und muss mittels Simulation einzeln ermittelt werden. Im verwendeten Beispiel zeigte es sich, dass die Adapter des Typs 4 kritisch waren und sie deshalb die *Sign Magnitude Truncation* verwenden müssen. Da eine solche Datenmodifikation jedoch ein höheres Quantisierungsrauschen erzeugt als ein übliches Runden, wurde bei der Filterimplementierung für die nicht kritischen Adapter ein normales Runden verwendet.

Das Listing 5 zeigt den Assembler-Quellcode des Stereo-Interpolationsfilters, welches aus lediglich 10 Instruktionen besteht. Da die Stop- und „jump always Addr0"-Instruktionen in die jeweils vorgehende Instruktion eingebunden werden, besteht der binäre Programmcode aus lediglich 8 Instruktionen. Dies will nicht etwa sagen, dass die Problemaufgabe winzig ist, sondern zeigt, mit welch wenigen Instruktionen der HotDSP ein solches Problem zu lösen vermag.

Die Gesamtgrösse der Filterimplementierung beträgt knappe 4 kGates. Die Hälfte davon wird vom Registerblock beansprucht. Die Kontrolleinheit, inklusive des Programm-*LUTs* benötigt lediglich 254 *Gates*. Dies bedeutet, dass der *Overhead* des Kontrollteils lediglich 6.4 % der Gesamtgrösse beträgt.

```
# InEvent(0): Wake Up
# InEvent(1): Channel_Select 0=Channel1 1=Channel0
# InEvent(3): ProcessFirstPart 1=FirstPart 0=SecondPart
# OutEvent(0): Sample_Ready
# OutEvent(1): Sample_IsChannelA
# OutEvent(2): HotDSP_IsBusy

Addr0:
  arith mov nrm In0 RegU CstU Accu1
  stop
  call Event12,Adapt3(4),Adapt3(0),Adapt3(6) (2) AU_PipeStop

Adapt3:
  arith sub nrm Accu0 Reg0+ CstU  Accu1
  arith mulsub nrm Accu0 Accu1 Cst0+ Reg0+
  arith mulsub nrm Accu1 Reg0+ Cst0+ Accu0

Adapt4:
  arith sub nrm Reg1+  Accu0 CstU  Accu1
  arith mulsub h0z0r1s19 Accu0  Reg1+ Cst1+ Reg1+
  arith sub h0z0r0s18 Accu0 Accu1 CstU OutReg0
  jump always Addr0
```

Listing 5: Der Assembler-Quellcode des Stereo-Interpolationsfilters

Name des Blocks	*Grösse (GE[1])*	*Kommentar*
control	254	Kontrollblock mit dem Programm-LUT
instr	45	Programm-LUT
datapath	3702	Datenverarbeitungseinheit
Reg	1939	Registerfile, (kombinatorische Logik: 879, Register:1060)
AU	1757	Arithmetische Einheit (Eingangsregister: 298)
Total	**3958**	

Tabelle 3: Grösse in GE des ersten Beispiels (Referenztechnologie: Alcatel-Mietec 0.35 µm CMOS)

Implementierung eines Stereo-Dezimationsfilters unter Verwendung des zweiten HotDSP-Typen

Ein Stereo-Dezimationsfilter soll wie der Interpolationsfilter ebenfalls als *Bireciprocal Lattice Wave Digital Filter* (WDF) 9ter Ordnung realisiert werden. Ein vorausgehender Filter soll zudem die Audiosignale von einem eventuellen Offsetwert befreien. Eine simple Subtraktion kann dieser Offset, welcher mittels Integration während einer längeren Zeitspanne evaluiert wird, entfernen. Der Filter kann auch einfach als IIR-Tiefpassfilter erster Ordnung betrachtet werden, wobei die Grenzfrequenz so tief liegen muss,

[1] GE: *Gate Equivalents*

dass nur Frequenzen unterhalb der Hörgrenze entfernt werden. Ebenfalls diese Anwendung kann unkompliziert mit Hilfe der Co-Design-Methodik implementiert werden und dient deshalb als Anwendungsbeispiel des zweiten HotDSP-Typen.

Selbstverständlich treten auch bei diesem WDF-Filter *Zero Limit Cycles* auf, wenn nicht geeignete Rundungstechniken eingesetzt werden. Eine weitere Komplikation ist durch den Integrator des Tiefpassfilters gegeben. Um eine Grenzfrequenz von 5 Hz bis 20 Hz zu erhalten, muss das vom Offset befreite Eingangssignal über einen solch langen Zeitraum aufsummiert werden, dass die Daten innerhalb des Integrationskreises riesige Grössen annehmen können und nicht mehr mit dem üblichen Datenformat repräsentiert werden können. Statt übergrosse Busweiten zu verwenden, wird die Addition des Integrators deshalb in zwei Additionen aufgeteilt, welche mittels einem Carry-Flag verbunden sind.

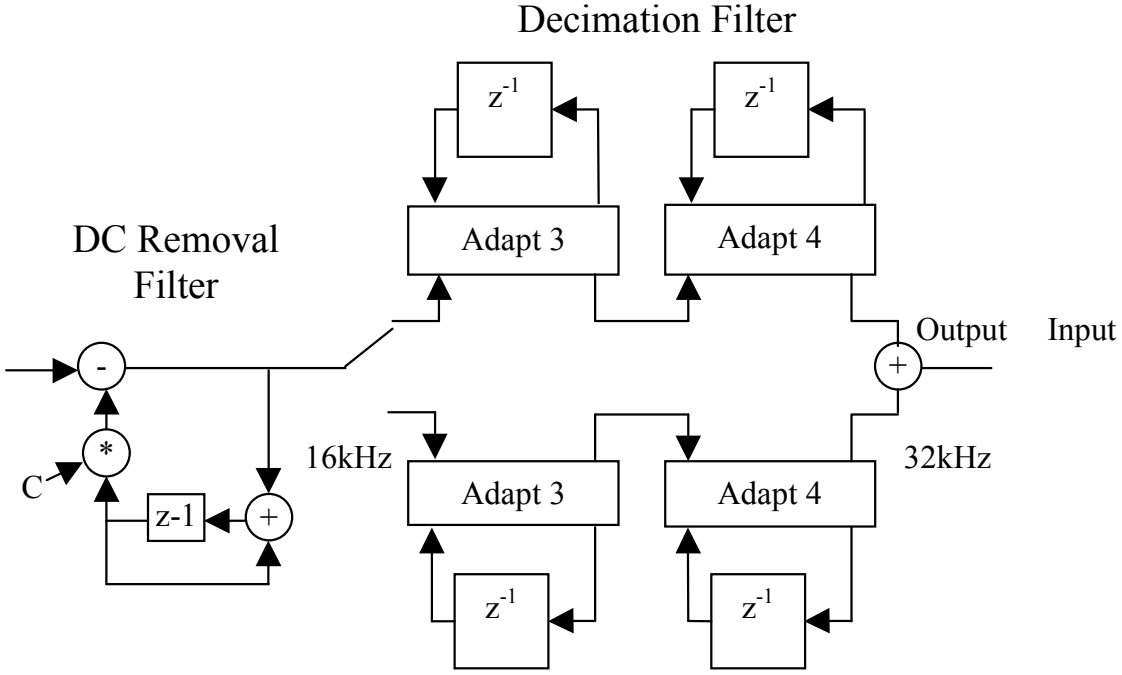


Abbildung 13: Die Struktur des Dezimationsfilter und dem vorausgehenden Offsetentfernungsfilter

Die Struktur des Dezimationsfilters ist derjenigen des Interpolationsfilters sehr ähnlich. Wegen der vorausgehenden Offsetfilterung und der zusätzlichen Addition am Ende des WDF-Filters, welche beide „Arme“ der WDF-Struktur wiederum zusammenführt, ist der Quellencode jedoch aufwendiger als derjenige des Interpolationsfilters, wie in Listing 6 ersichtlich ist.

```
# InEvent(0): Wake Up (Start)
# InEvent(1): Channel_Select 0=Channel0 1=Channel1
# InEvent(2): ProcessFirstPart 1=FirstPart 0=SecondPart
# InEvent(3): DC_Rm_Select 1=With DC_Rm
# OutEvent(0): Sample_Ready
# OutEvent(1): Sample_IsChannelA
# OutEvent(2): HDSP_IsBusy

_addr 0
Addr0:
   arith mov   nrm In0 RegU  CstU  Accu1
   stop
   jump Event12,Ch1_Odd,Ch0_Odd,Ch1_Even

Ch0_Even:
   call Event3, Ch0_DCRm(0), Ch0_Adapt3(0)
   jump always Addr0
Ch0_Odd:
   call Event3, Ch0_DCRm(2), Ch0_Adapt3(2)
   arith add    h3z0r0s16   Reg4  Accu0 CstU  OutAlias0
   jump always Addr0
Ch0_DCRm:
   arith sub    nrm         Accu0 Reg6  CstU  Reg6
   arith add    h0z0r0s0    Reg5  Accu0 CstU  Reg5
   arith incc   nrm         Reg6  RegU  CstU  Reg6
   arith mov    nrm         Reg6  RegU  CstU  Accu1
Ch0_Adapt3:
   arith sub    nrm         Accu0 Reg0+ CstU  Accu1
   arith mulsub nrm         Accu0 Accu1 Cst0+ Reg0+
   arith mulsub nrm         Accu1 Reg0+ Cst0+ Accu0
Ch0_Adapt4:
   arith sub    nrm         Reg1+  Accu0 CstU  Accu1
   arith mulsub h0z0r1s19   Accu0  Reg1+ Cst1+ Reg1+
   arith sub    nrm         Accu0  Accu1 CstU  Reg4
   ret

Ch1_Even:
   call Event3, Ch1_DCRm(0), Ch1_Adapt3(0)
   jump always Addr0
Ch1_Odd:
   call Event3, Ch1_DCRm(2), Ch1_Adapt3(2)
   arith add    h3z0r0s16   Reg11  Accu0 CstU  OutReg0
   jump always Addr0
Ch1_DCRm:
   arith sub    nrm         Accu0  Reg13 CstU  Reg13
   arith add    h0z0r0s0    Reg12  Accu0 CstU  Reg12
   arith incc   nrm         Reg13  RegU  CstU  Reg13
   arith mov    nrm         Reg13  RegU  CstU  Accu1
Ch1_Adapt3:
   arith sub    nrm         Accu0  Reg7+ CstU  Accu1
   arith mulsub nrm         Accu0  Accu1 Cst0+ Reg7+
   arith mulsub nrm         Accu1  Reg7+ Cst0+ Accu0
Ch1_Adapt4:
   arith sub    nrm         Reg8+  Accu0 CstU  Accu1
   arith mulsub h0z0r1s19   Accu0  Reg8+ Cst1+ Reg8+
   arith sub    nrm         Accu0  Accu1 CstU  Reg11
   ret
```

Listing 6: Der Assembler-Quellcode des Dezimationsfilters

Die HotDSP-Konfiguration für diese Applikation verlangt ebenfalls nach mehr Ressourcen im Gegensatz zum ersten Beispiel. Die Vergrösserung des LUTs zur Aufnahme des voluminöseren Programmcodes ist dabei eher vernachlässigbar, wenn die Register zum Speichern der zusätzlichen Zustandsvariablen berücksichtigt werden. Die folgende Tabelle enthält die totale Grösse der verwendeten HotDSP-Konfiguration, aber auch die Grössen der einzelnen Untereinheiten. Da für diese Applikation der zweite HotDSP-Typ verwendet wurde, muss ein Vergleich mit dem ersten Anwendungsbeispiel mit Sorgfalt erfolgen.

Name des Blocks	*Grösse (GE)*	*Kommentar*
Control	218	Kontrollblock mit dem Programm-LUT
LUT	31	Programm-LUT des Kontrollblockes
Datapath	4785	Datenverarbeitungseinheit
Control	186	Kontrollblock der Datenverarbeitungseinheit
ROM	97	LUT des Letztgenannten
Reg	2969	Registerfile, (kombinatorische Logik: 1311, Register: 1657)
AU	1624	Arithmetische Einheit, (Eingangsregister: 282)
Total	**5004**	

Tabelle 4: Grösse in GE des zweiten Beispiels (Referenztechnologie: Alcatel-Mietec 0.35 um CMOS)

Eine detailliertere Analyse des mit Hilfe des HotDSP realisierten Dezimationsfilters ist im Kapitel „Qualitätsvergleich“ auf Seite R des Anhangs I zu finden. Unter anderem wird das Anwendungsbeispiel auch bezüglich Verlustleistung untersucht und mit anderen Designansätzen verglichen.

3.7 Schlussfolgerung

Die in diesem Kapitel präsentierte, neue Implementierungsmethode ermöglicht die Verwendung eines frei programmierbaren Prozessors auch bei jenen Anwendungen, welche normalerweise auf Grund eines schwierig zu erfüllenden Pflichtenheftes nur durch anwendungsspezifische Lösungen realisierbar sind. Die hohe Parametrisierbarkeit der verwendeten Prozessoren lässt die Kluft der Implementierungsqualitäten von Lösungen mit Hilfe von *General Purpose DSP* und von spezifischen Hardwarelösungen, je nach Anwendung, verringern oder sogar verschwinden.

Im Falle des Anwendungsbeispiels, welches für die in Anhang I durchgeführten Qualitätsanalyse (Seite „R“) verwendet wird, erreicht eine mit dem HotDSP realisierte Implementierungslösung nur bezüglich der

Designgrösse, nicht aber bezüglich der Leistungsaufnahme die selbe Designeffizienz wie bei einer von Hand durchgeführten Referenzimplementierung. Je nach Syntheseparameter und Funktionsmodus des Designs ist die Leistungsaufnahme der HotDSP-basierten Designlösung zwischen 20% und 25% höher als diejenige des Referenzdesigns.

Der grosse Vorteil eines HotDSP-basierten Lösungsansatzes liegt jedoch in der Entwicklungszeit. Die Wahl eines effizienten Lösungsansatzes ist immer ein Kompromiss zwischen Designqualität und Entwicklungszeit. Je nach dem Einsatzort eines Designs kann eine 20% höhere Leistungsaufnahme akzeptabel, wenn nicht gar vernachlässigbar sein. Eine deutliche Verkürzung der Entwicklungszeit ist jedoch immer erwünschenswert.

Die in diesem Kapitel vorgestellte Designmethode kann in sämtlichen Bereichen angewendet werden, wo eine Aufgabe fix implementiert werden muss. Mittels der Wahl der Grundstruktur des verwendeten DSPs kann dabei die Methode auf ein Anwendungsbereich zugeschnitten werden. Der in dieser Arbeit vorgestellte HotDSP liefert zum Beispiel auf Grund der Instruktions- und Power-optimierten Architektur ideale Voraussetzungen für Anwendungen, welche bei tiefem Stromverbrauch in Echtzeit Daten verarbeiten müssen. Gewiss, es gibt auch Anwendungen, wo die Methode nicht sinnvoll einsetzbar ist. Muss zum Beispiel die Flexibilität einer Lösung auch in späteren Zeitpunkten garantiert sein, so ist ein frei programmierbarer DSP zu verwenden, welcher nicht nur auf die momentane Anwendung zugeschnitten ist, sondern auch für die eventuell später zu realisierenden Applikationen noch garantiert genügend Ressourcen besitzt. Die Optimierungsphase der Methode garantiert deshalb nur bei fixen Implementierungen eine Effizienzsteigerung.

Der präsentierte Implementierungsansatz hat auch Nachteile und Schwächen, und zwar auf Niveau der Methode wie auch auf Niveau des realisierten HotDSPs.

Bezüglich des HotDSPs muss sicherlich gesagt werden, dass er, so wie er momentan besteht, optimiert ist für „kleinere“ Aufgaben, jedoch nicht mehr verwendet werden kann, wenn Datenspeicher und aufwendigere Funktionen wie Hardware-Loops, Pointer-Adressierung, usw. benötigt werden. Da sein Instruktionsset sehr speziell und zudem konfigurierbar ist, ist die Verwendung eines Hochsprachenkompilers stark erschwert, wenn nicht sogar verunmöglicht. Eine Erweiterung der HotDSP-Architektur um die aufgezählten Funktionen und die damit verbundene Anpassung der Entwicklungssoftware könnte eine mögliche Fortsetzung des Projektes bedeuten.

Auch sonst könnte die Entwicklungssoftware einige Verbesserungen ertragen. Bereits jetzt besitzt der Assembler die Fähigkeit, auf Grund des zum Voraus fixierten Assembler-Sourcecodes festzustellen, welche Ressourcen, und im Speziellen welche Register und Datenports nie verwendet werden und macht den Benutzer darauf aufmerksam. Dies, zusammen mit weiteren Fähigkeiten, die für eine Anwendung notwendigen Ressourcen zu bestimmen, könnte gleich dazu verwendet werden, den verwendeten Prozessor automatisch zu konfigurieren. Dadurch kann die Optimierungsphase der Methode zeitlich stark reduziert werden.

Wie bereits angesprochen wurde, sind die Möglichkeiten Entwicklungssoftware noch sehr beschränkt. Aber gerade in diesem Bereich könnten weitere Optimierungsmöglichkeiten realisiert werden, indem zum Beispiel nach entsprechender Analyse der Assemblercode optimiert wird. So wäre das Ersetzen Ressourcen-hungrigenr Operationen durch Sequenzen einfacherer Instruktionen eine Möglichkeit, weitere Optimierungen der Ressourcen vorzunehmen.

Eine letzte Anmerkung betrifft die Art, wie die DSP-Konfigurationen definiert werden. Die hohen Konfigurierungs- und Parametrisierungsmöglichkeiten von VHDL lassen sicherlich die Verwendung eines VHDL-Definitionspaketes zur Speicherung der Konfigurationen zu. Die Praxis zeigt jedoch, dass nur die besten VHDL-Simulations- und –Synthesetools die verwendete Parametrisierungssyntax korrekt interpretieren. Um die Methode für andere Tools zugänglich zu machen, kann in Erwägung gezogen werden, die VHDL-Beschreibung mittels eines Precompilers zu konfigurieren. Der Precompiler ersetzt dabei in der VHDL-Beschreibung sämtliche Parameterkonstanten durch die eigentlichen Werte und löscht VHDL-Zeilen, welche wegen der gewählten Konfiguration nicht benötigt werden. Der so erhaltene VHDL-Code ist nun selbst frei von Parameter-bezogenen Anweisungen und kann dadurch schneller, und vor allem von allen Simulations- und Synthesetools verarbeitet werden. Die Grundidee des Lösungsansatzes selbst und der Designfluss bleiben dabei bei dieser Modifizierung erhalten.

Referenzen

[1] www.arm.com, www.mips.com, www.tensilica.com
[2] R. Forsynth & al., “Gepard: A Parametrisable Digital Signal Processor”, Proc. ICECS'98 (International Conference on Electronics, Circuits and Systems), 1998, Lisbon, Portugal

[3] E. Ofner & al., „GEPARD, ein parammetrisierbarer DSP Kern für ASICs“, DSP Deutschland '97, Sep/97, München, Germany, pp 176-180
[4] R. Gierlinger & al., „GEPARD, a Parameterisable DSP Core for ASICs“, Proceedings ICSPAT'97, Sep/97, San Diego, USA, pp 203-207
[5] T. Röwer, „Programmable Intellectual Propery Modules for System Design by Reuse”, Diss. ETH No. 13905, ISBN 3-89649-627-1
[6] A. Drollinger & al., "HW/SW Co-design Methodology for Ultra-Power-Applications and Fastest Development Time", Proc. of the 8th International Symposium on Integrated Circuits, Devices & Systems, ISIC-99, pp. 455-458, Grand Hyatt, Singapore, September 8-10, 1999.
[7] A. Fettweis & al., „Proceedings on the IEEE, VOL. 74, No 2, February 1986”, page 270-327
[8] L. Gazsi, „Explicit Formulas for Lattice Wave Digital Filters”, IEEE Trans. Circ. & Sys., Vol 32, 68-88, 1985
[9] N.P.C. Manshanden & al., „Design of Wave Digital Filters with Minimal Coefficientlength”.

4 Ein high-level DSP-Synthesetool für schnelle Implementierungen von DSP-Algorithmen

Dieses Kapitel präsentiert ein neues DSP-Synthesetool, welches anwendungsspezifische DSPs für Filter- und Datenpfadanwendungen erzeugt [1][2]. Low-Power-*Strukturen werden dabei als Basis für die erzeugten Architekturen verwendet.*
Weitere besondere Merkmale des Tools sind seine Einfachheit im Gebrauch, und die Tatsache, dass es alle Schritte der Filterimplementierung, inklusive der automatischen Festlegung der Datenwortbreiten durchführen kann.
Für die Festlegung der Datenwortbreiten wird ein neuer Algorithmus vorgestellt, welcher die optimalen Datenwortbreiten mittels Simulation ermittelt.

In einem ersten Kapitelteil werden die Kriterien und Bedingungen erörtert, welche ein DSP-Synthesetool für den Low-Power-Bereich erfüllen soll. Es folgen dann einige Erklärungen zum verwendeten Filter-Designfluss und zu der Realisation des Tools. Eine Datenverarbeitungseinheit eines FFT-Prozessors bildet das abschliessende Anwendungsbeispiel.

4.1 Die Generierung von Filter- und Datenpfad-Architekturen mittels eines *high-level* Synthesetools

4.1.1 Der Filter-Designfluss

Die Entwicklung eines anwendungsspezifischen, digitalen Filters durchläuft verschiedene Entwicklungsetappen. Zu Beginn definiert die Anwendungsart den Filtertyp, also ob ein Tief-, Hoch- oder Bandpassfilter benötigt wird, oder ob der Filter eher vom Typ einer Filterbank (*Equalizer*) sein muss.

Aufgrund der gewünschten Filtercharakteristik wird in einem nächsten Schritt eine Filterstruktur gesucht, welche das Anforderungsprofil erfüllt. Gute Kenntnisse in Filterstrukturen und –Algorithmen sind ein Muss für eine gute Strukturwahl eines Filters, ob also ein FIR-Filter, IIR-Filter usw. verwendet werden sollte. Die Entwicklung kann dabei von Filter-Designtools unterstützt werden, welche gewisse Entscheidungen automatisieren, wie zum Beispiel die Filterordnung wählen, Koeffizienten optimieren, usw.

In einem nächsten Schritt, der sogenannten Implementierung des Algorithmus, muss eine Hardwarearchitektur gefunden werden, welche den Filteralgorithmus so exakt wie möglich verarbeiten kann und zudem die Randbedingungen der Zielumgebung zufrieden stellt. Damit die Hardware nicht zu gross wird, versucht man, das zeitlich konkurrente Verhalten der verschiedenen Operationen eines Datenflussgraphen sequentiell zu gestalten (*scheduling, resource assignment/binding*), was zur Folge hat, dass die verschiedenen Operationen von den selben arithmetischen Einheiten verarbeitet werden können (*resource sharing*).

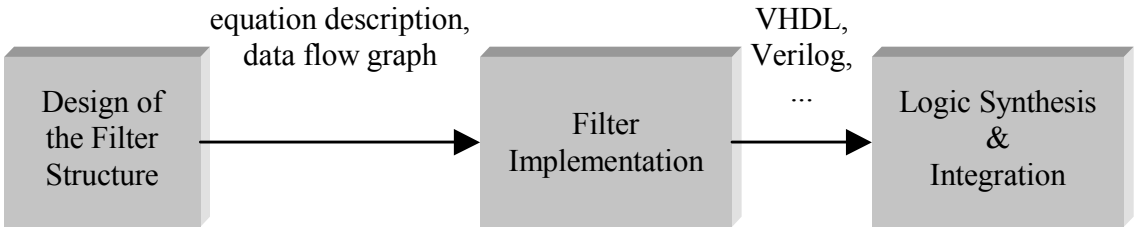


Abbildung 14: Filter-Designfluss

Grundsätzlich wird zwischen zwei Optimierungsarten der Hardwarezuweisung unterschieden. Bei der Ressourcen-limitierten (*resource constrained*) Optimierung muss ein Algorithmus mit einer fixierten Menge von Ressourceneinheiten realisiert werden, wobei die Anzahl der Arbeitszyklen die zu optimierende Grösse bildet [3][4]. Demgegenüber hat die Zeit-limitierte (*time constrained*) Optimierung als Randbedingung eine fixierte Anzahl von Arbeitszyklen. Die Ressourcenkosten bilden in diesem Falle die zu optimierende Grösse

[5][6][7]. Natürlich können beide Optimierungen auch kombiniert durchführen werden [8].

Zur effizienten Filterimplementierung gehört auch die Transformation des Datenformates von der üblichen Gleitkommadarstellung in die für eine Hardware häufig geeignetere Festkommadarstellung. Die verwendeten Datenwortbreiten der Festkommadaten zusammen mit den verwendeten Rundungsfunktionen haben einen direkten Einfluss auf das Quantisierungsrauschen und müssen unter Berücksichtigung der gewünschten Datenverarbeitungsqualität gewählt werden. Für die Implementierungsphase gibt es eine ganze Reihe von Softwaretools, welche die Projektion eines Algorithmus in eine Hardware vereinfachen, wenn nicht sogar automatisieren. Einige solcher existierenden Tools werden gleich anschliessend vorgestellt.

Der erhaltene Hardwareblock kann nun in die Zielumgebung eingefügt werden oder der Logiksynthese zugeführt werden.

4.1.2 Existierende Tools für die Konzeption und die Generation von DSP-Architekturen

Auf dem Markt existieren verschiedenste Softwaretools, welche die verschiedenen Etappen der oben beschriebene Filterimplementierung teilweise oder sogar voll automatisch durchführen. Es folgt eine (sicher nicht komplette) Auflistung solcher Tools.

Frontier Design

Innovativer Anbieter ist Frontier Design, welcher 1990 mit *DSP Station*^{TM} erstmals eine kommerzielle Komplettlösung für die Realisation von DSP-Applikationen anbot [9]. *DSP Station* basiert auf den Arbeiten von *Cathedral* I und II [10] und erhielt seit seiner Einführung verschiedenste Erweiterungen und beinhaltet heute einerseits eine ganze Reihe von Tools wie *DSP Design Lab*^{TM}, *DSP Architekt*^{TM}, *Filter Architekt*^{TM} usw. für die Konzeption, Simulation und Optimierung von Filtern, andererseits mit *Mistral1*^{TM}, *Mistral2*^{TM} und *HDLsyn*^{TM} auch drei Synthesetools, welche die von den Konzeptionstools entworfenen Filterstrukturen in eine Hardware projizieren. Alle drei Tools generieren synthetisierbaren VHDL- oder Verilog-Code, unterscheiden sich jedoch in dem Typ der Hardwarearchitektur, die sie bilden. *Mistral1* generiert eine bit-serielle Architektur, *Mistral2* eine anwendungsspezifische, parallele DSP-Struktur und *HDLsyn* erzeugt einen *hardwired, one-cycle* Datenpfad. Die Schnittstelle zwischen allen Tools bildet die hardwareunabhängige

Beschreibungssprache DFL[1], mit der Filterstrukturen mit einfachen Gleichungen beschrieben werden können.

Neustes Produkt von Frontier Design ist *A|RT*^{TM}, bestehend aus *Designer*, *Builder* und *Library* [11]. Folgende Idee liegt hinter *A|RT*: Die meistverwendete Programmier- und Beschreibungssprache ist C. Sie wird von Ingenieuren aller Bereiche verwendet. Durch Ausbildung und der späteren Arbeit lernen Ingenieure auf eine „programmierungsnahe" Art zu denken, nämlich zeitlich sequentiell organisiert. Die zeitlich parallele oder konkurrente Verarbeitung von verschiedenen Aufgaben, wie es in Datenflussgraphen oder in Verilog- und VHDL-Beschreibungen stattfindet, ist den Ingenieuren weniger vertraut. Frontier Design meinte, dass den Ingenieuren nicht diese Art von Denken aufgezwungen werden soll und lancierte A|RT, welches voll C-basiert ist. Mit Hilfe der *A|RT-Library*, einer Festkommabibliothek, kann der zu implementierende Algorithmus beschrieben werden. Mit in den C-Code eingebundenen Pragmas sowie mit Tooloptionen können Randbedingungen definiert werden, welche die Architekturgenerierung beeinflussen. Wie bei *DSP Station* liefert *A|RT* am Ende der Synthese eine VHDL- oder Verilogbeschreibung der generierten DSP-Architektur, welche mit dem ursprünglichen C-Code bit-gleich sein soll. Wie bei *DSP Station* beinhaltet auch *A|RT* verschiedenste Analyse- und Simulationstools.

Nebenbei: Die Idee, C als Designeingabe zu verwenden, findet auch immer mehr Anhänger, welche C für die Spezifikation von ganzen Systemen verwenden möchten: In der *Open SystemC Initiative* befinden sich bekannte Unternehmungen wie *Synopsys*, *Frontier Design* und *CoWare*, welche die Entwicklung von *SystemC* vorantreiben[12].

Behavioral Compiler^{TM} *von Synopsys*

Einen etwas anderen Weg beschritt Synopsys. Auch mit VHDL kann ein Filter auf hohem Abstraktionsniveau beschrieben werden. Da ein Gesamtdesign sowieso in einer Hardwarebeschreibungssprache definiert wird, weshalb soll dann für den Filterblock eine andere Beschreibungsart verwendet werden? Mit dem *Behavioral Compiler* kann selbst ein komplexer Filter mit einem einzigen VHDL-Prozess auf hohem Abstraktionsniveau in einer sequentiellen Weise beschrieben werden. Via in den VHDL-Code eingefügte Warteschleifen oder via Syntheseoptionen können dem *Behavioral Compiler* mitgeteilt werden, wie viele Taktzyklen zur Verarbeitung des Filteralgorithmus zur Verfügung stehen. Mittels Time-Scheduling und Ressourcenbinding findet dann der *Behavioral Compiler* eine für die Anwendung geeignete DSP-Architektur [13].

---

[1] DFL: *Data Flow Language*

IMTs DSP-Synthesetool

Am IMT[1] wurde von Alexander Heubi speziell für Lowest-Power-Anwendungen ein DSP-Synthesetool entwickelt [14]. Das Ziel dieses Werkzeugs war nicht, eine Komplettlösung für Filterentwicklungen anzubieten, sondern lediglich DSP-Architekturen zu generieren, welche den Ansprüchen von Lowest-Power-Anwendungen gerecht werden. Die verschiedenen, bereits quantisierten Operationen eines Filteralgorithmus werden in Form einer Netzliste dem Tool eingegeben. Die generierte DSP-Architektur wird als VHDL-Beschreibung ausgegeben.

4.1.3 Weshalb ein neues high-level DSP-Synthesetool für den Low-Power-Bereich?

Muss für eine Lowest-Power-Anwendung ein Filteralgorithmus, welcher eine hohe Rechenleistung verlangt, implementiert werden, so kann die Verwendung eines frei programmierbaren DSPs ausgeschlossen werden: Zu ineffizient wäre seine Architektur bezüglich Verlustleistung und Geschwindigkeit. Aber auch die Spezialarchitekturen, welche mit Hilfe der existierenden *high-level* DSP-Synthesetools generiert werden, können die gestellten Ansprüche meist nicht vollständig befriedigen, da diese Synthesetools nicht in der Lage sind, Low-Power-Techniken und –Kriterien bei der Architekturgeneration gebührend zu berücksichtigen.

Synopsys wählte einen sehr interessanten Lösungsansatz, welcher jedoch auch Schwächen hat. Bei Verwendung des *Behavioral Compilers* von *Synopsys* muss die Filterstruktur entweder von Hand oder mit Hilfe zusätzlicher Tools quantisiert werden, was die Entwicklungszeit merklich beeinflusst. Mangelnde Einflussmöglichkeiten des Anwenders während der Architekturgenerierung, eingeschränkte Optimierungskriterien, die undurchsichtige Art der Aufnahme der generierten Architektur in die interne Datenbasis sowie die komplizierte Weise, wie die generierte Architektur via VHDL-Beschreibung ausgegeben wird, verlangt nach mehr Interaktionsmöglichkeiten des Anwenders und einem einfacheren Handling der erzeugten DSP-Architektur.

IMTs DSP-Synthesetool [14] erzeugt und optimiert DSP-Strukturen nach Low-Power-Kriterien, hat aber vor allem bezüglich Flexibilität verschiedene Nachteile: Die Eingabeart der Algorithmen ist unpraktisch, die verschiedenen Operationen des Algorithmus müssen bereits quantisiert sein, und die Optimierung der DSP-Architektur kann vom Anwender nicht beeinflusst werden. Zudem gibt es nur eine beschränkte Möglichkeit, Randbedingungen, welche an die Architektur gestellt werden, zu definieren. Die grösste Einschränkung liegt jedoch darin, dass das DSP-Synthesetool

[1] IMT: Institut de Microtechnique de l'Université de Neuchâtel

nur Architekturen erzeugen kann, welche ständig im selben Arbeitsmodus arbeiten.

4.2 Anforderungsprofil eines effizienten *high-level* Synthesetools für den *Lowest-Power*-Anwendungsbereich

Wie aus vorausgehender Erklärung ersichtlich ist, kann keines der existierenden Tools zur Generierung von DSP-Architekturen alle Anforderungen von Low-Power-Anwendungen vollständig erfüllen. Welches sind aber die exakten Wünsche, die an ein simples, aber doch flexibles *high-level* Synthesetool für den Low-Power-Bereich gestellt werden? Abbildung 15 veranschaulicht die gewünschten Interface- und Interaktionsmöglichkeiten eines DSP-Synthesetools, welche zusammen mit den allgemeinen Anforderungen an das Tool auch in dem nachstehenden Pflichtenheft fixiert sind.

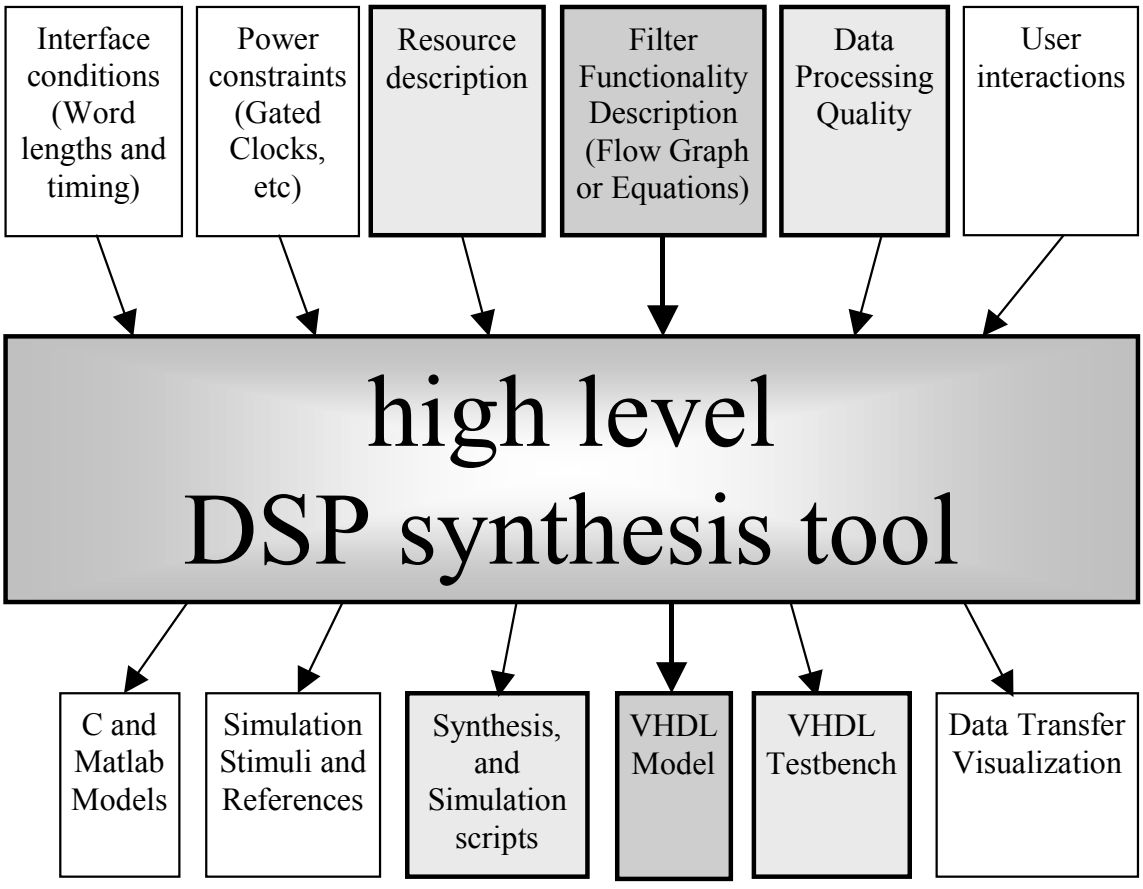


Abbildung 15: Gewünschte Interface- und Interaktionsmöglichkeiten des DSP-Synthesetools

1. Das Synthesetool soll im Low-Power-Bereich für zeitlimitierte Filter- und Datenpfadanwendungen eingesetzt werden können.
2. Es ist autonom und ist nicht gebunden an ein bestimmtes Betriebssystem oder existierende Entwicklungtools.

3. Einfache Gleichungen können zur Beschreibung des Datenflussgraphen (*data flow graph, DFG*) verwendet werden. Die Syntax soll komfortabel sein, Funktionsdeklarationen zulassen und einen Verzögerungsoperator haben.
4. Via einer Ressourcendatei können alle zur Verfügung stehenden Ressourcentypen spezifiziert werden. Die Menge und die Art der Ressourcen soll frei wählbar und an die Zielanwendung anpassbar sein.
5. Das Haupterzeugnis des DSP-Synthesetools soll eine VHDL-Beschreibung sein, welches von den üblichsten Logik-Synthesetools verarbeitet werden kann. Das Abstraktionsniveau der VHDL-Beschreibung soll RTL sein.
6. Der Designweg von den Ursprungsgleichungen bis zum fertigen VHDL-Code soll so einfach wie möglich gehalten werden.
7. Viele Arten von Designrandbedingungen, einfach definierbar, sollen berücksichtigt werden können.
8. Das Tool soll mit Low-Power-Techniken wie *Gated Clocks* vertraut sein.
9. Die Quantisierung der einzelnen Operationen soll vollautomatisch durchgeführt werden. Der Anwender hat dafür lediglich die gewünschte Datenverarbeitungsqualität zu definieren.
10. Das Synthesetool erlaubt, Multimodi-DSP-Strukturen zu erzeugen (DSP-Strukturen, welche in verschiedenen Arbeitsmodi arbeiten können).
11. Es hat ein komfortables, programmierbares Benutzerinterface. Eine graphische Benutzeroberfläche, Applikationen und Schnittstellen zu anderen Tools wie VHDL-Simulatoren, mathematischen Programmen, usw. können mit Hilfe von diesem Interface realisiert werden.
12. Eingriffe von Benutzerseite werden ebenfalls über dieses Benutzerinterface ermöglicht, und zwar zu jedem Zeitpunkt. Damit soll der Designfluss selbst kontrolliert, aber auch auf die Datenbasis des Synthesetools zugegriffen werden können.
13. Das Synthesetool soll auch die Weiterverarbeitung der generierten DSP-Architektur vereinfachen, indem es zum Beispiel für die Simulation eine Testumgebung bereitstellt, Skripts für Simulation und Logiksynthese generiert, Matlab- und C-Modelle erzeugt, usw. Mit Hilfe des programmierbaren Benutzerinterfaces sollen solche Erweiterungen von den Anwendern geschrieben werden können.
14. Es soll völlig offen und erweiterbar sein. Nebst Anpassungen über das programmierbare Benutzerinterface soll auch der Kern des Tools via *Plugins* frei erweiterbar sein.

4.3 Die Realisation des DSP-Synthesetools

Dieser Teil präsentiert die Realisation eines neuen Tools, welches im Low-Power-Bereich die Palette der DSP-Synthesetools komplettiert. Es beinhaltet sämtliche Werkzeuge, um aus einer Anzahl von Gleichungen, welche das Verhalten eines Filters beschreibt, eine vollständige Beschreibung des Filters in synthetisierbarem VHDL zu generieren. Es erfüllt sämtliche Punkte des zuvor aufgelisteten Pflichtenheftes.

4.3.1 Der Designfluss

Zur Entwicklung des DSP-Synthesetools muss der genaue Ablauf des *high-level* Syntheseprozesses gewählt sein. Die Synthese kann in eine Reihe von Prozessen aufgeteilt werden, welche in einer festen Folge abgearbeitet werden. Dieses altbewährte Synthesekonzept [15] besitzt auf globalem Niveau keine Iterationsschleifen und ist deshalb einfach und schnell. Sie scheint auch für das neue DSP-Synthesetool ein guter Ansatz zu sein. Abbildung 16 zeigt den gewählten Designfluss.

Es folgt eine kurze Auflistung aller Operationsschritte des Designflusses:

- Der *Parser* liest die Datei, welche die Filtergleichungen und die Randbedingungen enthält. Die Gleichungen werden interpretiert, vereinfacht und auf eine solche Weise in der internen Datenbasis gespeichert, dass ein einfacher Zugriff gewährleistet ist.
- Der *Merger* sucht für jede arithmetische Operation eine funktionale Einheit, welche die Operation verarbeiten kann. Dies wird auch oftmals als *Allocation* bezeichnet. Zudem werden Multi-Parameter-Operationen aufgeteilt in die Grundoperationen [z.B.: (a+b+c+d) $\Rightarrow$ ((a+b)+(c+d))]. Der *Merger* merkt auch, wenn eine Multiplikation oder Division mit einer Zweierpotenz stattfindet. Er wählt in diesem Fall einen Shifter für die Realisation der Operation.
- Es folgt eine optionale Initialquantisierung, welche allen Operationen eine Initialgrösse zuweist. Dies erlaubt die Berechnung einer Kostenfunktion, welche von den folgenden Optimierungsalgorithmen verwendet wird und ihnen erlaubt, Operationen mit ähnlichen Eigenschaften besser zu gruppieren. Wird dieser Schritt nicht durchgeführt, so werden zur Berechnung der Kostenfunktion vordefinierte Werte verwendet.
  Die Quantisierung geschieht in drei Schritten. Zuerst wird eine C-Beschreibung des Filters erzeugt. Diese wird dann in eine Bibliothek compiliert und dynamisch in das DSP-Synthesetool eingebunden (gelinkt). Die eigentliche Quantisierung aller Operation kann jetzt mittels simpler Simulationen durchgeführt werden. Da die Operationen den

einzelnen funktionalen Einheiten noch nicht definitiv zugeteilt sind, werden sie individuell quantisiert.

- Der *Scheduler* bestimmt für jede Operation den Zyklus, an dem die Operation durchgeführt werden soll, und zwar auf eine Weise, dass die Totalkosten aller notwendigen Ressourcen minimiert werden. Nach diesem Schritt ist der exakte Ausführungszeitpunkt jeder Operation genau bestimmt.

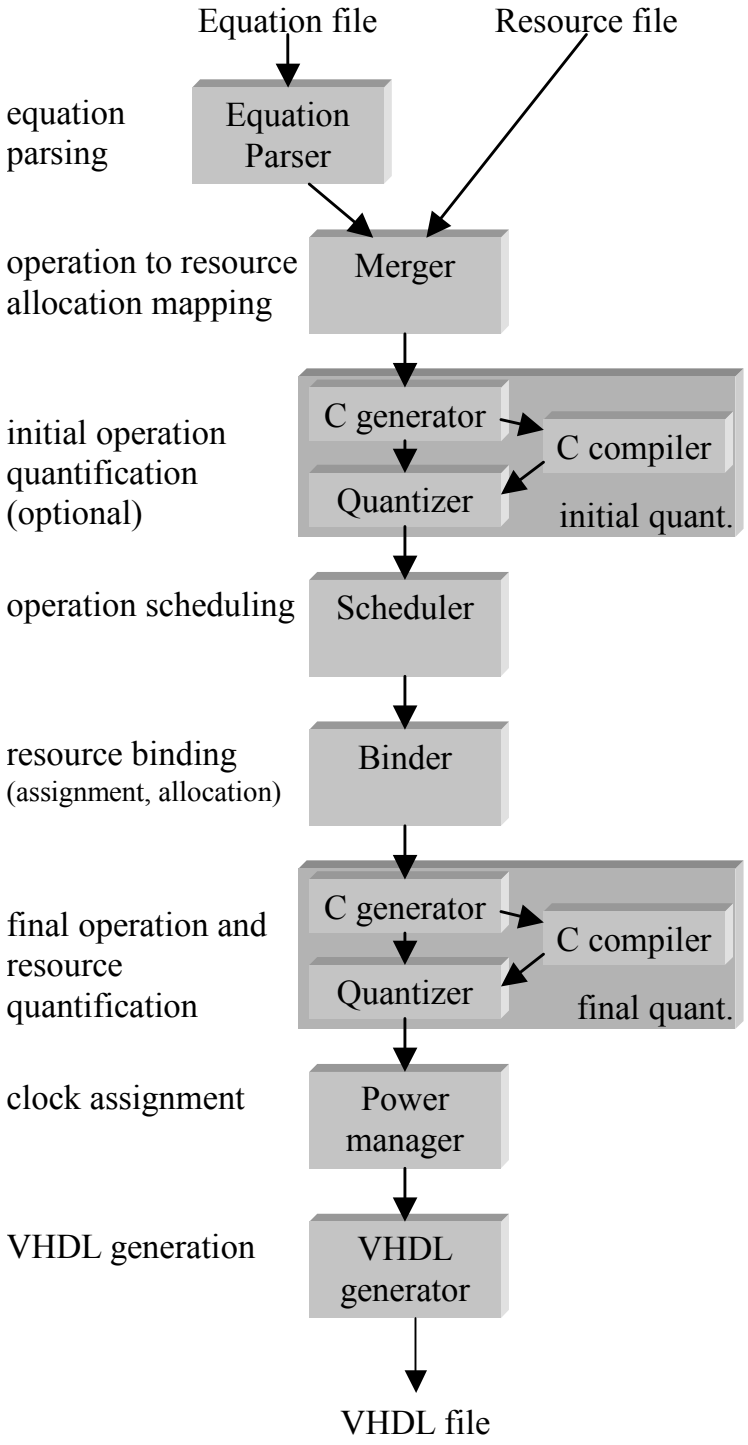


Abbildung 16: Der für Low-Power-Anwendungen geeignete Designfluss für die Filtergenerierung

- Der *Binder* weist jede Operation einer funktionalen Einheit zu und zwar auf eine Art, dass die totale Anzahl der Multiplexer, welche die funktionalen Einheiten verbinden, minimiert ist. Dieser Schritt wird meist als *Resource Assignment* oder *Resource Binding* bezeichnet.
- Nun werden die Operationen endgültig quantisiert. Bei dieser Quantisierung werden die Ressourcen-Abhängigkeiten der Operationen berücksichtigt.
- Der *Power-Manager* teilt jeder funktionalen Einheit einen Clock zu und zwar so, dass die totale Systemaktivität minimiert wird. Die maximale Anzahl der zu verwendenden Clocks kann vom Anwender festgelegt werden. Für FPGA-Anwendungen können auch synchrone Systeme, welche lediglich einen Clock haben, erzeugt werden.
- Schlussendlich generiert der VHDL-Generator die VHDL-Beschreibung des Filterblockes zusammen mit einem VHDL-Testbench, welcher eine schnelle Verifikation des erzeugten Blocks erlaubt.

4.3.2 Architektur der generierten DSPs und dessen Klassifizierung

Die Architekturen der generierten Special-DSPs sind vom Typ VLIW, wobei die Instruktionen von einer fest verdrahteten Zustandsmaschine erzeugt werden. Die Datenpfade besitzen einen in Abbildung 17 ersichtlichen Strukturtyp, wobei während der Ressourcenzuweisung (*Scheduling/Binding*) versucht wird, die Kosten der arithmetischen Einheiten und der Multiplexer zu minimieren.

Die arithmetischen Einheiten besitzen jeweils am Eingang und nicht am Ausgang Datenregister damit während den unbenutzten Zyklen keine interne Aktivität entsteht.

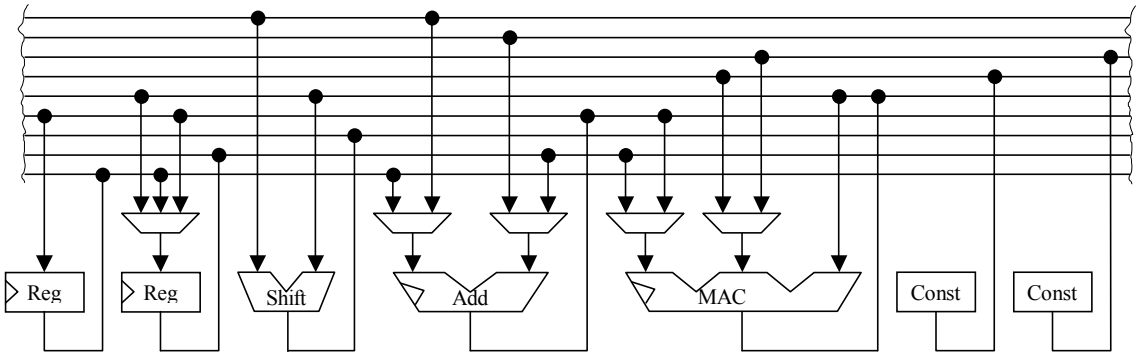


Abbildung 17: Die Ressourceneinheiten wählen via Multiplexer jeweils jeden Zyklus die entsprechenden Datenquellen.

4.3.3 Low-Power-*Techniken*

Die *Low-Power*-Eigenschaften der mit dem DSP-Synthesetool erzeugten Designs sind basiert auf:

- Reduktion der Aktivität durch Reduktion von Komplexität erreicht mittels:
  - Verwendung optimaler Grundstrukturen mit kleinen, lokalen Datenbussen
  - Effiziente *Scheduling/Binding* – Technik

  Dies hat einen direkten, positiven Einfluss auf a_k, C_k und CPO (siehe Kapitel „Betrachtungen zum Stromverbrauch und zur Verlustleistung“, Seite 17), was indirekt auch $E_{task,dynamic}$, $P_{dynamic}$, *Energy/Operation* beeinflusst.
- Verwendung von Ressourceneinheiten, bei denen die internen Register am Eingang anstelle am Ausgang liegen und so bei Nichtgebrauch ein Toggeln der kombinatorischen Logik vermeiden (beeinflusst ebenfalls a_k)
- Verwendung von optimal organisierten *Gated Clocks* (beeinflusst ebenfalls a_k)
- Pipelinetechniken reduzieren die hohe Verzögerungszeiten der Logik und erlauben damit die Verwendung tiefer Betriebsspannungen (beeinflusst: *delay*, und indirekt die zu verwendende Speisespannung *Vdd*)

4.3.4 Die Implementierung des Synthesetools mit all seinen Modulen

Das DSP-Synthesetool ist im Gegensatz zu anderen Ansätzen [8] ein autonomes Werkzeug und besteht aus zwei Teilen. Der Kern ist in C/C++ programmiert und beinhaltet die Datenbasis, sämtliche für den Designfluss benötigte Module, ein Plugin-Loader und ein Tcl/Tk-Interface. Dieses wird für das graphische Benutzerinterface und für mögliche Benutzererweiterungen gebraucht, welche den zweiten Teil oder das Frontend des Tools bilden und im nächsten Unterkapitel näher betrachtet werden soll.

Die Datenbasis enthält die Daten der Filterstruktur und der Ressourcen auf eine Weise, dass sie von sämtlichen Modulen auf unkomplizierte Art verwendet und bearbeitet werden können. „*DB Access*“ stellt den anderen Modulen eine Reihe praktischer Funktionen zur Verfügung. Sie dienen für häufig benötigte Standardmanipulationen der Datenbasis und zum Speichern, Laden und der Verifikation der Letztgenannten. Zudem erweitert sie die Tcl-Skriptsprache um verschiedenste Prozeduren und Pseudovariablen, um einen uneingeschränkter Zugriff auf die Datenbasis von Seite des Tcl-Interfaces zu bilden.

Der *Plugin-Loader* erlaubt, benutzer- und anwendungsspezifische Erweiterungen als sogenannte *Plugins* in das bestehende System einzubinden. Diese Erweiterungen haben dabei einen freien Zugriff auf sämtliche Strukturen, Funktionen und die Datenbasis des DSP-Synthesetools.

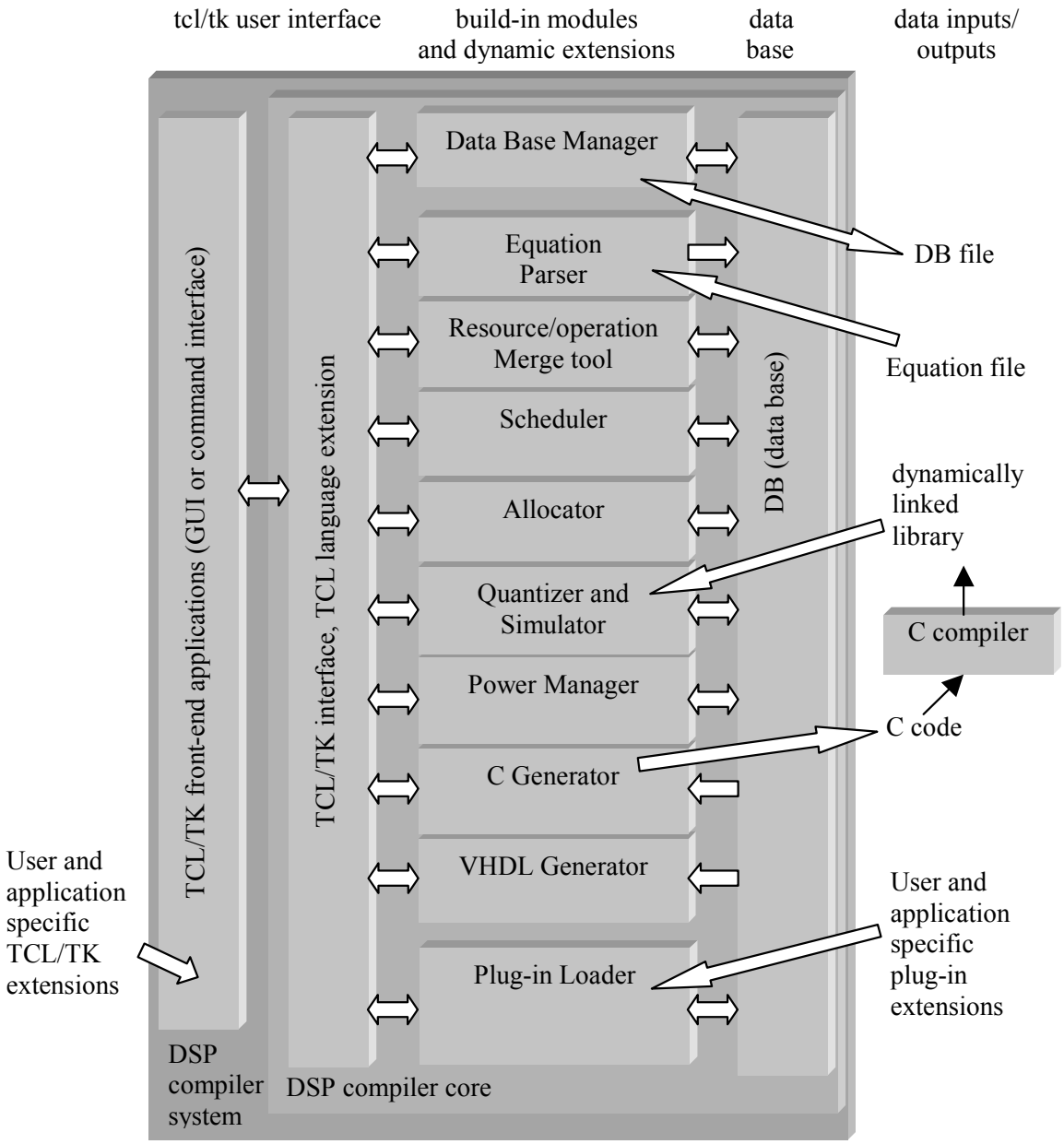


Abbildung 18: Der Implementierungsaufbau des DSP-Synthesetools

Die Module haben einerseits freien Zugriff auf die Datenbasis, können andererseits über das Tcl/Tk-Interface kontrolliert werden. Während der Zeit, wo ein Modul geladen ist, erweitert es gleichzeitig die Tcl-Skriptsprache um verschiedenste neue Funktionen und Pseudovariablen. In den folgenden Unterkapiteln soll die Arbeitsweise der einzelnen Module genauer betrachtet werden.

Parser

Der Parser transferiert die Beschreibung eines Filteralgorithmus in das interne Format der Datenbasis. Listing 7 zeigt an Hand eines sehr einfachen Beispiels die Syntax einer solchen Beschreibung.

```
function R=Diff2(a,b,c)
  R=a-b*c

function [Y D]=summ(X)
  Y=(X*0.50+X@1*0.85+X@2+
    X@3*0.85+X@4*0.50)/4
  D=Diff2(X,Y,0.5)

function [Y D]=nothing(X)
  Y=X
  D=0.0

mode summ
  ModeFunction summ
  OpInfo {X D Y}.NbrBit=[16,15]
  OpInfo X.Cycle=-1
  OpInfo Y.SNR=83
  OpInfo D.SNR=53
  ModeInfo Period=3

mode nothing
  ModeFunction nothing
  OpInfo {X D Y}.NbrBit=[16,15]
  OpInfo X.Cycle=-1
  ModeInfo Period=1

GenInfo NbrClk=3
```

Listing 7: Beschreibung eines einfachen Filters

In einem ersten Teil wird mit arithmetischen Gleichungen der zu implementierende Algorithmus definiert. Die festzustellende Ähnlichkeit mit DFL ist dabei kein Zufall. DFL inspirierte wegen seiner Klarheit und Einfachheit die Definition der Gleichungssyntax. Sehr praktische Elemente wie der *Delay*-Operator und Deklarationen von Funktionen, welche mehrere Resultate zurückgeben können, vereinfachen die Algorithmusbeschreibung.

Im zweiten Teil werden alle Funktionsmodi spezifiziert. Das gewählte Beispiel hat zwei Modi. Für jeden Modus werden nebst der Funktionsweise die verschiedenen Randbedingungen definiert. Dies sind üblicherweise Interfacekonditionen und die zur Verfügung stehende Anzahl von Taktzyklen.

In einem letzten Teil können noch allgemein gültige Bedingungen definiert werden, wie zum Beispiel die Anzahl der zu verwendenden *Gated Clocks*.

Da die Anzahl der Klammerebenen einer Gleichung beliebig sein kann, wird zum Lesen der Gleichungen ein rekursiver Algorithmus verwendet. Dieser transformiert jede Textgleichung in eine Abhängigkeitsliste der verwendeten Operationen. Bevor die Daten dieser Liste in die Datenbasis des Synthesetools transferiert wird, durchlaufen sie verschiedenste

Transformations- und Optimierungsphasen, welche garantieren, dass die Weise, wie die Gleichungen intern repräsentiert sind, am Besten geeignet ist für eine Hardwareimplementierung. Folgende Umformungen werden u.a. vorgenommen: Gleichungen mit verschiedenen Operationen werden in verschiede Gleichungen mit je einer Operation aufgeteilt, Funktionskörper werden in die Hauptfunktion kopiert (*flatten*), arithmetische Vereinfachungen werden durchgeführt, usw.

Merger (Resource allocation)

Bevor *der Merger* gestartet wird, muss die Ressourcen-Definitionsdatei gelesen werden. Diese enthält eine Liste der zur Verfügung stehenden Ressourceneinheitentypen. Die Liste kann frei modifiziert und erweitert werden und kann somit einem Anwendungsbereich spezifisch angepasst werden. Jede definierte Ressourceneinheit besitzt einen Namen, eine Menge der Operationen, welche sie durchführen kann, die Anzahl der Inputs, und verschiedene Attribute für Timing, Kosten und der gestatteten Manipulationen.

Es wird zwischen logischen und funktionalen Einheiten unterschieden. Die Letztgenannten sind richtige Hardwareressourcen und besitzen physikalische und funktionelle Eigenschaften, welche mittels Attributen präzisiert werden müssen. Im Gegensatz dazu werden die logischen Einheiten nur dazu gebraucht, nach einer ersten, provisorischen Zuweisung eines Operators eine geeignete Ressourceneinheit zu finden. Wird in einem Algorithmus zum Beispiel eine Multiplikation oder eine Division verwendet, so heisst das nicht, dass diese Operation mit einem Multiplikator oder einem Divisor durchgeführt werden muss. Entspricht einer der verwendeten Operanden einer Zweierpotenz, so kann eine einfache *Shift*-Operation verwendet werden. Der in Listing 8 gezeigte Ausschnitt zeigt die Syntax, welche verwendet werden muss, damit bei einer arithmetischen Multiplikation die der Situation angemessene Ressourceneinheit gewählt wird.

Der *Merger* weist jedem arithmetischen Operator eine Ressourceneinheit zu, welche wenn möglich die selbe Anzahl von Parametern bzw. Inputs hat. Hat ein Operator mehr Parameter als eine Ressourceneinheit Inputs hat, so kann der Operator, unter Voraussetzung, dass dies für die betroffene Ressourceneinheit gestattet ist, in verschiedene Operatoren mit je einer kleineren Parameterzahl aufgeteilt werden. Wird eine logische Ressourceneinheit gefunden, so selektiert der *Merger* gemäss den definierten Konditionen die definitive Ressourceneinheit.

```
#Resource
  Name MulReal
  Operation Mul
  NbrInput 2
  InputSwap 0
  Cost "4*{nBit}+10*{nBitSrc0}*{nBitSrc1}"
  Cost 3200
  Delay 1
  Period 1
  Function C_DOUBLE "{Out}={In0}*{In1};"
  Function C_LONG "{Out}=MulShR({In0},{In1},{nBitFracSrc0}…
  Function VHDL "{Out}<=Sat(Trim(SIGNED({In0})*SIGNED({In1…
  Function MATLAB "{Out}={In0}*{In1};"

#Resource
  Name Mul
  Operation *
  NbrInput 2
  GenericExpansion 1
  IF    IsPosPower2Cst(In1) USE << In0 Log2(Abs(In1))
  ELSIF IsPosPower2Cst(In0) USE << In1 Log2(Abs(In0))
  ELSIF IsNegPower2Cst(In1) USE >> In0 Log2(Abs(In1))
  ELSIF IsNegPower2Cst(In0) USE >> In1 Log2(Abs(In0))
  ELSIF IsCst(In1) USE Mul In0 In1
  ELSIF IsCst(In0) USE Mul In1 In0
  ELSIF always USE Mul In0 In1
```

Listing 8: Ein Ausschnitt aus dem Ressourcen- Definitionsfile

Scheduler

Der Scheduler weist jeder Operation einen Arbeitszyklus zu und zwar in einer Weise, dass die Totalkosten der notwendigen Ressourcen minimiert werden. Dies geschieht indem die Operationen mit fest zugewiesenen Zyklen fixiert werden und die anderen Operationen so lange „hin- und hergeschoben“ werden, bis die Konfiguration mit den Minimalkosten gefunden wurde. Die Anzahl der Dimensionen des dabei zu lösenden Optimierungsproblems entspricht der Anzahl der „freien“, das heisst, der nicht fixierten Operationen. Je nach dem zu implementierenden Algorithmus ist dadurch die Komplexität des Optimierungsproblems sehr hoch, und das verlangt die Verwendung eines effizienten Optimierungsalgorithmus für das Time-Scheduling.

Die Problematik des Time-Schedulings wurde bereits vor Jahren eingehend studiert und auch gelöst und soll deshalb in dieser Arbeit nicht speziell behandelt werden [16][17]. Die eindrückliche Art, wie IMTs Synthesetool

diese Arbeit verrichtet, ermutigte, wiederum Tabu-Search[1] als Optimierungsalgorithmus zu verwenden [14]. Bezüglich IMTs Lösung bekam der Scheduler verschiedenste Erweiterungen, welche der grösseren Menge der möglichen Randbedingungen und der Tatsache, dass eine Anwendung verschiedene Arbeitsmodi haben kann, Rechnung trägt. Zudem kann der Scheduler direkt vom Anwender beeinflusst werden.

*Binder (*resource assignment *oder* resource binding*)*

Ebenfalls die Zuordnung der verschiedenen Operationen auf die Ressourceneinheiten geschieht mittels iteraktiver Optimierung. Zuerst wird eine Initialzuordnung durchgeführt. Anschliessend werden die den identischen Ressourceneinheiten zugeordneten Operationen so lange vertauscht, bis eine Lösung gefunden wird, welche den Anforderungen genügt.

Tabu-Search dient ebenfalls hier als Optimierungsalgorithmus. Die Kostenfunktion berücksichtigt die Menge der notwendigen Multiplexer und Busse einer Lösung. Nebst dem Austauschen von Operationen zwischen identischen Ressourceneinheiten werden bei gewissen Operationen auch die Eingänge vertauscht und zudem die Verwendung von arithmetischen Ressourcen als Zwischenregister in Betracht gezogen. Um ein optimales Resultat zu erhalten, werden die verschiedenen Arbeitsmodi einer Anwendung gleichzeitig optimiert.

C-Generator

Der C-Generator kann von einer Anwendung verschiedene C-Modelle bilden. Je nach der gewählten Art des C-Models wird die Ressourcenzuweisung berücksichtigt oder aber nicht und es wird eine

---

[1] *Tabu-Search* ist ein iterativer Optimierungsalgorithmus [18]. Bei jeder Iteration wird von der aktuellen Lösung aus eine Menge von Umgebungslösungen gesucht (Kandidaten). Mittels einer Kostenfunktion wird die Beste davon gewählt und dient bei der nächsten Iteration wiederum als aktuelle Lösung. Damit nun der Algorithmus nicht in ein lokales Minimum fällt, welches er nicht mehr verlassen kann, werden all die bereits gefundenen Lösungen in einer Liste registriert und bei der Wahl von möglichen Umgebungslösungen nicht mehr in Betracht gezogen. Da die Lösungen dieser Liste für die Wahl der Umgebungslösungen ein Tabu darstellen, wird der Algorithmus Tabu-Search genannt. Am Ende der Optimierung wird die beste aller gefundenen Lösungen gewählt.
Die Effizienz dieses Grundalgorithmus ist von verschiedenen Faktoren abhängig. So ist die Auswahl und die Anzahl der Umgebungslösungen einer der Faktoren und die Kostenfunktion spielt ebenfalls eine Rolle. Es konnte ebenfalls festgestellt werden, dass die Tabuliste nicht sämtliche Lösungen registrieren muss, damit der Algorithmus korrekt arbeitet. Die Listenlänge ist somit auch einen Faktor, welcher der jeweiligen Anwendung angepasst werden muss.

Gleitkommadarstellung der Daten verwendet oder aber eine Festkommadarstellung. Die Ressourcenzuweisung kann natürlich nur dann berücksichtigt werden, wenn diese bereits stattgefunden hat.

Die verschiedenen Funktionsmodelle erlauben, die Ressourcenzuweisung zu überprüfen und das Quantisierungsverhalten der Anwendung zu analysieren.

Im weiteren wird auch eine Kostenfunktion erzeugt, die dem Quantizer die Totalkosten der Ressourcen in Funktion der Quantisierung aller Operationen gibt.

Sämtliche Funktionen werden so in eine Datei geschrieben, dass durch kompilieren eine dynamisch ladbare Bibliothek gebildet werden kann.

Quantizer

Der Quantizer sucht mittels Simulation für jede Operation eines Algorithmus das optimale Datenformat. Dies findet er in zwei Schritten. In einem ersten Schritt bestimmt er die notwendige Anzahl der *Integer Bits*, in einem zweiten Schritt die notwendige Anzahl aller Bits (*Integer* + *Factional Bits*). Kapitel 4.3.5 diskutiert eingehender den für die Datenquantisierung verwendeten Algorithmus.

Nebst der Quantisierung aller Operationen stellt der Quantizer auch verschiedenste Funktionen zur Verfügung, um die Anwendungen zu simulieren und erzeugt Stimuli und Referenzen für externe Simulationen.

Power Manager

Der Power Manager teilt allen Ressourceneinheiten mit Eingangsregistern einen Clock zu, und zwar so, dass die totale Registeraktivität minimiert ist.

Dazu wird Tabu-Search als Optimierungsalgorithmus verwendet. Die Anzahl der zu verwendenden Clocks kann dabei vom Anwender spezifiziert werden. Ist eine Anwendung für eine FPGA-Implementierung bestimmt, kann auch ein synchrones Design mit nur einem Clock definiert werden.

VHDL-Generator

Der VHDL-Generator produziert schlussendlich eine VHDL-Beschreibung des Designs und einen anwendungsspezifischen Testbench. Der VHDL-Stil ist RTL[1] welcher von den üblichen Synthesetools verarbeitet werden kann. Der VHDL-Code ist mit vielen Kommentaren versehen, welche eine Korrespondenz zum ursprünglichen Anwendungsalgorithmus bildet.

[1] RTL: *Register Transfer Level*

4.3.5 Der verwendete Algorithmus für die automatische Festlegung der Datenwortbreiten

Der Quantizer sucht mittels Simulation für jede Operation eines Algorithmus das optimale Datenformat. Die vom C-Generator erzeugten Gleitkomma- und Festkomma-C-Beschreibungen des zu implementierenden Algorithmus werden dazu compiliert und als Simulationsmodul ins Synthesetool eingebunden.

Bevor jedoch mit der eigentlichen Quantisierung begonnen wird, checkt der Quantizer die ins Synthesetool eingebundenen Simulationsmodelle auf ihre Korrektheit. Im Speziellen wird geprüft, ob eine Übereinstimmung zwischen den nicht-Ressourcen-gebundenen und den Ressourcen-gebundenen Modellen sowie zwischen den Gleitkomma- und den Festkommamodellen besteht.

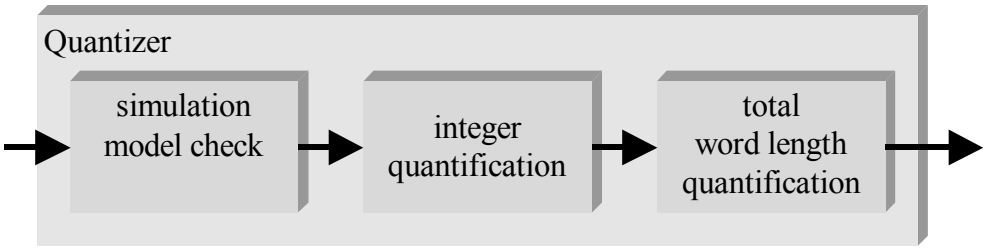


Abbildung 19: Die Überprüfung der Simulationsmodelle und die zwei darauf folgenden Schritte der Festlegung der Datenwortbreiten

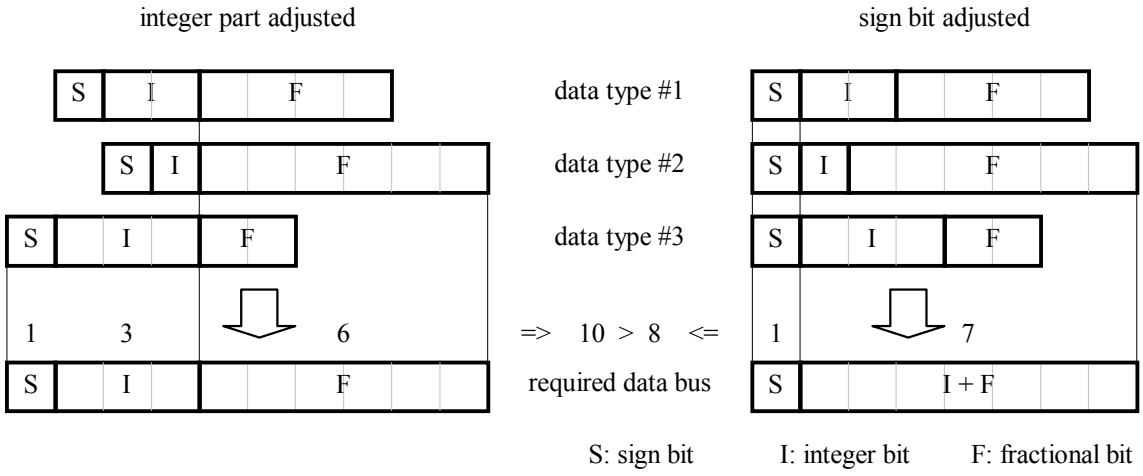


Abbildung 20: Drei verschiede Datentypen müssen vom selben Bus unterstützt werden. Müssen die Datentypen nach den Ganzzahlstellen ausgerichtet werden, wird ein breiterer Datenbus benötigt als bei einer Ausrichtung nach dem Vorzeichenbit. Nicht nur Busse können von dieser zweiten Ausrichtungsart profitieren, sondern auch arithmetische Einheiten.

Die Festlegung der Datenwortbreiten findet sodann in zwei Schritten statt, nämlich in der Festlegung der notwendigen *Integer Bits* und der Festlegung der Gesamtwortbreiten. Der Grund, dass im zweiten Schritt nicht nach den

Nachkommastellen (*Fractional Bits*), sondern nach den Gesamtwortbreiten gesucht wird, ist, dass die Kommastelle einer Zahl lediglich eine Frage der Interpretation ist und sich die Ressourcenabhängigkeit nicht auf den Nachkommateil der Daten bezieht, sondern auf die Gesamtwortbreite (siehe Abbildung 20 zur Visualisierung).

Festlegung der Integer Bits

Es ist von Wichtigkeit, beim Bestimmen der *Integer Bits* und der Gesamtdatenwortbreiten mittels Simulation die „richtigen" Simulationsstimuli zu verwenden, also solche, welche für die Anwendung angemessene Amplituden besitzen. Deshalb müssen in einem ersten Schritt geeignete Stimuli definiert werden.

In der existierenden Version des Synthesetools werden die Stimuli lediglich auf Grund der definierten Datenwortbreiten der Ein- und Ausgangsports definiert. Es werden dabei Stimuli verwendet, deren Werte mit einer ändernden Frequenz zwischen den Maximum- und Minimumwerten der Eingangsports variieren (*Sweep*-Funktion*)*. Der Anwender kann diese gewählten Stimuli auf ihre Eignung kontrollieren und gegebenenfalls korrigieren.

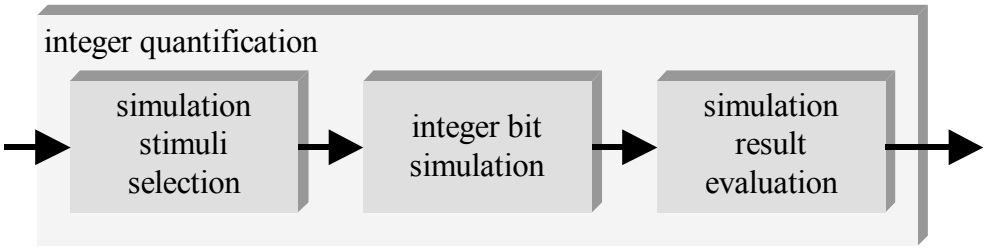


Abbildung 21: Die Operationen zur Ermittlung der Gesamtdatenwortbreiten

Während der Simulation werden die Maximalwerte[1] sämtlicher Knoten aufgezeichnet, welche sodann im letzten Schritt verwendet werden, um für jeden Knoten die notwendige Anzahl von *Integer Bits* zu definieren. Die Anzahl der *Integer Bits* wird dann so gewählt, dass die jeweils grössten aufgetretenen Werte noch repräsentiert werden können. Da dies jedoch keine Garantie dafür ist, dass die wirklich höchstmöglichen Datenwerte keine Datenüberläufe produzieren, werden an den entsprechenden Algorithmusstellen Sättigungsfunktionen verwendet.

Diskussion:

- Die etwas einfache Weise, wie die Stimuli festgelegt werden, kann in

[1] Besser: diejenigen Werte, zu dessen Repräsentation am meisten *Integer Bits* benötigt werden

einigen Fällen unbrauchbare Resultate ergeben, zum Beispiel, wenn an einem Knoten auf Grund der Phasenverschiebungen zwei Signale destruktiv interferieren und auf diese Weise nur kleine Amplituden an diesem Knoten auftreten. Bei einer neueren Version des Synthesetools sollte eventuell eine etwas elaboriertere Methode für die Stimulidefinition verwendet werden, welche versucht, mögliche Phaseninterferenzen zweier Signale zu berücksichtigen.

- Die zweite Anmerkung betrifft das Kriterium der Selektion der *Integer Bits*. Es wird versucht, dass alle in der Simulation aufgetretenen Werte mit der gewählten Anzahl der *Integer Bits* repräsentiert werden können. Die Frage ist, ob es notwendig ist, immer jeden Datenwert einwandfrei repräsentieren zu können oder ob in gewissen Fällen eine fehlerhafte Repräsentation zugelassen werden kann. Speziell wenn bei einem Datenstrom Impulse mit hohen Amplituden auftreten, darf je nach Anwendung erlaubt sein, diese mittels Sättigung zu begrenzen, ohne dass dies für die Anwendung störend ist. Dürfen also gewisse Signalformen bewusst mittels Sättigung korrigiert werden, so können innerhalb des Filterblockes schmälere Datenbusse verwendet werden, was sich positiv auf die Systemkomplexität auswirkt. Als zweite Modifikation bei einer neuen Version des Synthesetools wird deshalb vorgeschlagen, dass die Amplitudenauswertung der *Integer-Bit*-Simulation in einer vom Anwender definierten Flexibilität durchgeführt werden kann.

Festlegung der Gesamtdatenwortweiten

Im zweiten Schritt sucht der Quantizer die beste Bitkonfiguration für die Repräsentation der Gesamtdatenwortbreiten (= *Integer Bits* + *Fractional Bits*). Je nachdem, ob eine Initialquantisierung durchgeführt wird oder ob die Ressourcenzuweisung bereits stattgefunden hat, müssen dabei die Ressourcenabhängigkeiten zwischen den Operationen berücksichtigt werden oder aber nicht.

Verschiedene Elemente eines Algorithmus können quantisiert werden, was jeweils einer Breitenlimitierung eines Hardwareressourcen-Typs entspricht. So ist es möglich, Zwischenwerte (Busse), Zustandsvariablen (Register), Parameter von Operationen (Eingänge arithmetischer Einheiten) und Resultate von Operationen (Ausgänge arithmetischer Einheiten) zu quantisieren.

Das Synthesetool quantisiert nur Operationsresultate, d.h. reduziert nur an den Ausgängen der arithmetischer Einheiten die Datenwortbreiten. Ob dabei ein Runden vorgenommen wird, kann vom Anwender im Ressourcendefinitionsfile definiert werden. Diese Art der Quantisierung verlangt, dass die Register zur Speicherung von Zwischenwerten und

Zustandsvariablen Weiten besitzen müssen, welche den Maximalweiten der zu speichernden Daten entsprechen.

Die beste Bitkonfiguration für die Gesamtzahlenrepräsentation kann nicht wie bei der Suche nach der Ganzzahlrepräsentation in einer einzigen Simulation gefunden werden, sondern muss nach einer ersten Initialkonfiguration mittels einem Optimierungsalgorithmus iterativ bestimmt werden. Wie beim Time-Scheduling und der Ressourcenzuweisung wird dazu ebenfalls Tabu-Search als Suchalgorithmus verwendet. Sind die Operationen noch nicht den einzelnen Ressourceneinheiten zugeordnet, entspricht die Anzahl der Freiheitsgrade dieses Optimierungsproblems der totalen Anzahl der Operationen aller Arbeitsmodi, andernfalls lediglich der Anzahl der arithmetischen Ressourceneinheiten.

Für den Optimierungsalgorithmus wir eine Kostenfunktion benötigt. Als Kriterium für die Kosten werden vom Synthesetool lediglich die Ressourcenkosten berücksichtigt. Es liegt deshalb nahe, als Kostenfunktion direkt die Ressourcenkosten zu verwenden, wobei so jedoch die SNR-Randbedingungen, also die geforderte Datenverarbeitungsqualität nicht berücksichtigt würde.

Eine Möglichkeit, wie bei der Verwendung des Tabu-Search-Algorithmus die SNR-Randbedingungen berücksichtigt werden könnten, ist ein direktes Ignorieren derjenigen Kandidaten, welche die SNR-Randbedingungen nicht erfüllen. Eine andere Möglichkeit ist, die Qualitätsbedingungen in die Kostenfunktion einzubinden. Zum Beispiel können die Kosten der Lösungen mit unerfüllten Qualitätsbedingungen als unendlich hoch definiert werden. Auch hier werden, jedoch indirekt über die Kostenfunktion, diejenigen Kandidaten ignoriert, welche die SNR-Randbedingungen nicht erfüllen. Die Kostenfunktion lässt sich also wie folgt schreiben:

$$Cost(\mathbf{b}) = \begin{cases} ResourceCosts(\mathbf{b}) & ; \ SNR(\mathbf{b}) \ is \ OK \\ \infty & ; \ SNR(\mathbf{b}) \ isn't \ OK \end{cases} \qquad 1)$$

Diese Art, die SNR-Randbedingungen zu berücksichtigen, führten jedoch zu einer Konvergenzschwierigkeit, welche mittels Abbildung 22 erklärt werden kann.

Die Fläche innerhalb der Abbildung, welche aus vielen kleinen Quadraten zusammengesetzt ist, entspricht dabei einem konstruierten, zweidimensionalen Optimierungsproblem. Jedes der kleinen Quadrate repräsentiert eine Lösungskonfiguration, wobei der Grauton die entsprechenden Kosten charakterisieren: Je dunkler ein Quadrat ist, desto teurer sind die Kosten. Die Grenze, wo die SNR-Randbedingungen erfüllt

werden, ist mit der vertikalen Linie gekennzeichnet. Start bezeichnet die Initialkonfiguration des Suchalgorithmus. Werden nun sämtliche Kandidaten ignoriert, welche die SNR-Randbedingungen nicht erfüllen (welche sich also links von der vertikalen Linie befinden), sei es durch direkte Ignorierung dieser Kandidaten, sei es indirekt durch die Verwendung obiger Definition der Kostenfunktion, so konvergiert der Suchalgorithmus unweigerlich gegen das lokale Minimum im Punkt A. Damit das globale Minimum im Punkt B gefunden werden kann, muss dem Suchalgorithmus gestattet werden, auch im linken Bereich zu suchen, also dort, wo die SNR-Randbedingungen nicht erfüllt sind.

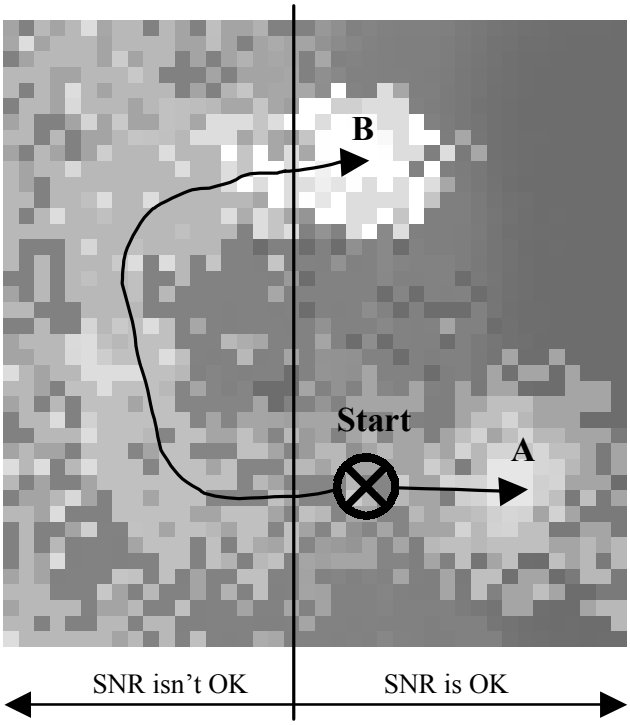


Abbildung 22: Die Konvergenzschwierigkeiten von Tabu-Search bei der Quantisierung (konstruierte Situation): Werden die Kandidaten, welche die SNR-Randbedingungen nicht erfüllen ignoriert, konvergiert der Suchalgorithmus gegen den Punkt A. Werden die Kandidaten jedoch zugelassen, konvergiert der Algorithmus gegen Punkt B.

Die Kostenfunktion muss aus diesem Grunde so modifiziert werden, dass eine Lösung mit unerfüllten SNR-Randbedingungen nicht gleich eliminiert wird, jedoch trotzdem eine gewisse Unattraktivität erhält. Dies kann mittels zusätzlicher Strafkosten geschehen, welche zu den effektiven Ressourcenkosten hinzuaddiert werden, wobei die Höhe der Strafkosten in Abhängigkeit des Verfehlens der SNR-Randbedingung sein soll, damit das Optimierungsproblem gut konvergiert. Die Kostenfunktion kann nun wie folgt geschrieben werden:

$$Cost(\mathbf{b}) = \begin{cases} ResourceCosts(\mathbf{b}) & ;\ SNR(\mathbf{b})\ is\ OK \\ ResourceCosts(\mathbf{b}) + PenalityCost(SNR(\mathbf{b})); & SNR(\mathbf{b})\ isn't\ OK \end{cases} \quad 2)$$

Die Verwendung von SNR-Werten ist aus folgendem Grund unpraktisch; Es kann interessant sein, ein Qualitätsmittelwert verschiedener Signale zu kennen. Ein Mittelwert von SNR-Werten gibt jedoch keinen Sinn, denn ist ein Signal perfekt, so ist der SNR-Wert unendlich. Ein Mittelwert mit anderen Signalen wird deshalb, unabhängig der SNR-Werten der anderen Signale, auch unendlich. Aus diesem Grund verwendet das Optimierungsverfahren nicht direkt die vom Anwender definierten SNR-Werte sondern die umgerechneten, mittleren Quantisierungsfehler. Die Kostenfunktion schreibt sich so wie folgt:

$$Cost(\mathbf{b}) = \begin{cases} ResourceCosts(\mathbf{b}) & ;\ Noise(\mathbf{b})\ is\ OK \\ ResourceCosts(\mathbf{b}) + PenalityCost(Noise(\mathbf{b})); & Noise(\mathbf{b})\ isn't\ OK \end{cases} \quad 3)\ ^{1}$$

Die Datenwortbreiten haben nicht nur einen Einfluss auf die Ressourcenkosten, sondern auch auf die Verarbeitungsqualität. Man kann versuchen, diese Einflüsse um den Ort der „optimalen“ Lösung, also dort, wo die SNR-Randbedingungen gerade noch erfüllt sind, zu linearisieren.

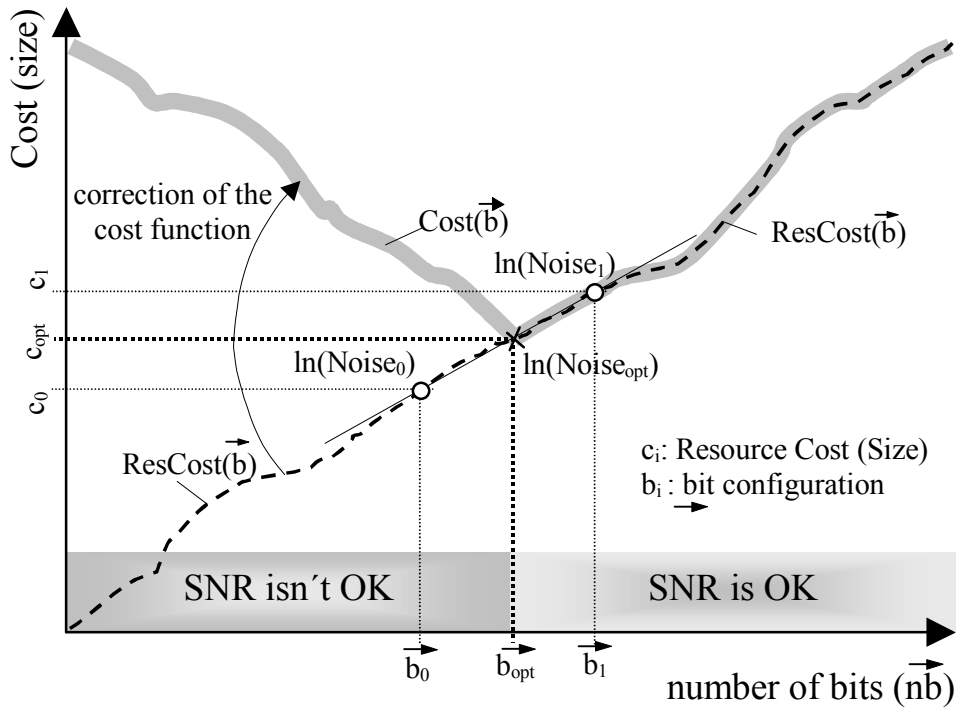


Abbildung 23: Definition der Kostenfunktion für die Quantisierung

Mit dieser Linearisierung kann die Straffunktion so definiert werden, dass die Kostenfunktion um die optimale Lösung herum eine Symmetrie erhält, wie in Abbildung 23 gezeigt wird. Auf der Abszisse liegen die einzelnen Bitkonfigurationen, welche eigentlich mehrdimensional sind, zur

[1] Mit *Noise* ist das Quantisierungsrauschen, oder der durchschnittliche Quantisierungsfehler gemeint.

Darstellung jedoch in eine Dimension projiziert sind. Die Ordinate enthält die Totalkosten aller Ressourcen. Wie zu erwarten ist, erhöhen sich die Ressourcenkosten bei wachsender Wortlänge, wie auch die Funktion *ResCost(**b**)* zeigt.

Zur Korrektur der Kostenfunktion wird als erstes die Position der optimalen Lösung geschätzt. Dazu wird je eine Bitkonfiguration verwendet, welche eine zu gute und eine zu schlechte Datenverarbeitungsqualität ergeben. Nimmt man an, dass eine etwa lineare Abhängigkeit besteht zwischen den Datenwortbreiten, den Ressourcenkosten und dem Logarithmus des resultierenden Quantisierungsfehlers, so können die Strafkosten für die korrigierte, symmetrische Kostenfunktion wie folgt konstruiert werden.

$$PenalityCost\left(Noise\left(\mathbf{b}\right)\right) = PenalityFactor * \left[\ln\left(OptNoise\right) - \ln\left(Noise\left(\mathbf{b}\right)\right)\right] \quad 4)$$

PenalityFactor entspricht dabei der doppelten Steigung der Kostenfunktion an der Stelle der optimalen Lösung und berechnet sich wie folgt:

$$PenalityFactor = 2 * \frac{Cost_1 - Cost_2}{\ln\left(Noise\left(\mathbf{b}_1\right)\right) - \ln\left(Noise\left(\mathbf{b}_2\right)\right)} \quad 5)$$

Da dieser Straffaktor nur sehr grob geschätzt ist, ist es angezeigt, ihn während der Optimierung ständig so zu korrigieren, dass der Suchalgorithmus ungefähr die selbe Anzahl von Lösungen findet, welche die SNR-Randbedingungen erfüllen und solche, welche sie nicht erfüllen. Der Straffaktor wird deshalb nach jeder Iteration des Suchalgorithmus wie folgt modifiziert:

$$PenalityFactor_{new} = PenalityFactor_{old} * \begin{cases} 1 + IntegrationsFaktor & ; SNR\ is\ OK \\ 1 - IntegrationsFaktor & ; SNR\ isn't\ OK \end{cases} \quad 6)$$

Auf diese Weise kann sichergestellt werden, dass der Optimierungsalgorithmus gleichmässig um die optimale Lösung herum sucht. Der Pseudocode auf Seite J des Anhangs I enthält detailliertere Informationen darüber, wie die folgenden Berechnungen durchgeführt werden: Quantisierungsfehler eines Ausgangskanals, mittlerer Quantisierungsfehler, Kostenfunktion und Straffaktor. Die Pseudocode-Routinen berücksichtigen dabei die Existenz verschiedener Ausgangskanäle und verschiedener Arbeitsmodi.

Die Festlegung der Gesamtdatenwortbreiten hat eine Initialisierungs- und eine Optimierungsphase.

In der Initialisierungsphase müssen eine ganze Reihe von Initialisierungen vorgenommen werden, damit eine Konvergenz des iterativen

Optimierungsalgorithmus erstens einmal gewährleistet, und zweitens beschleunigt wird.

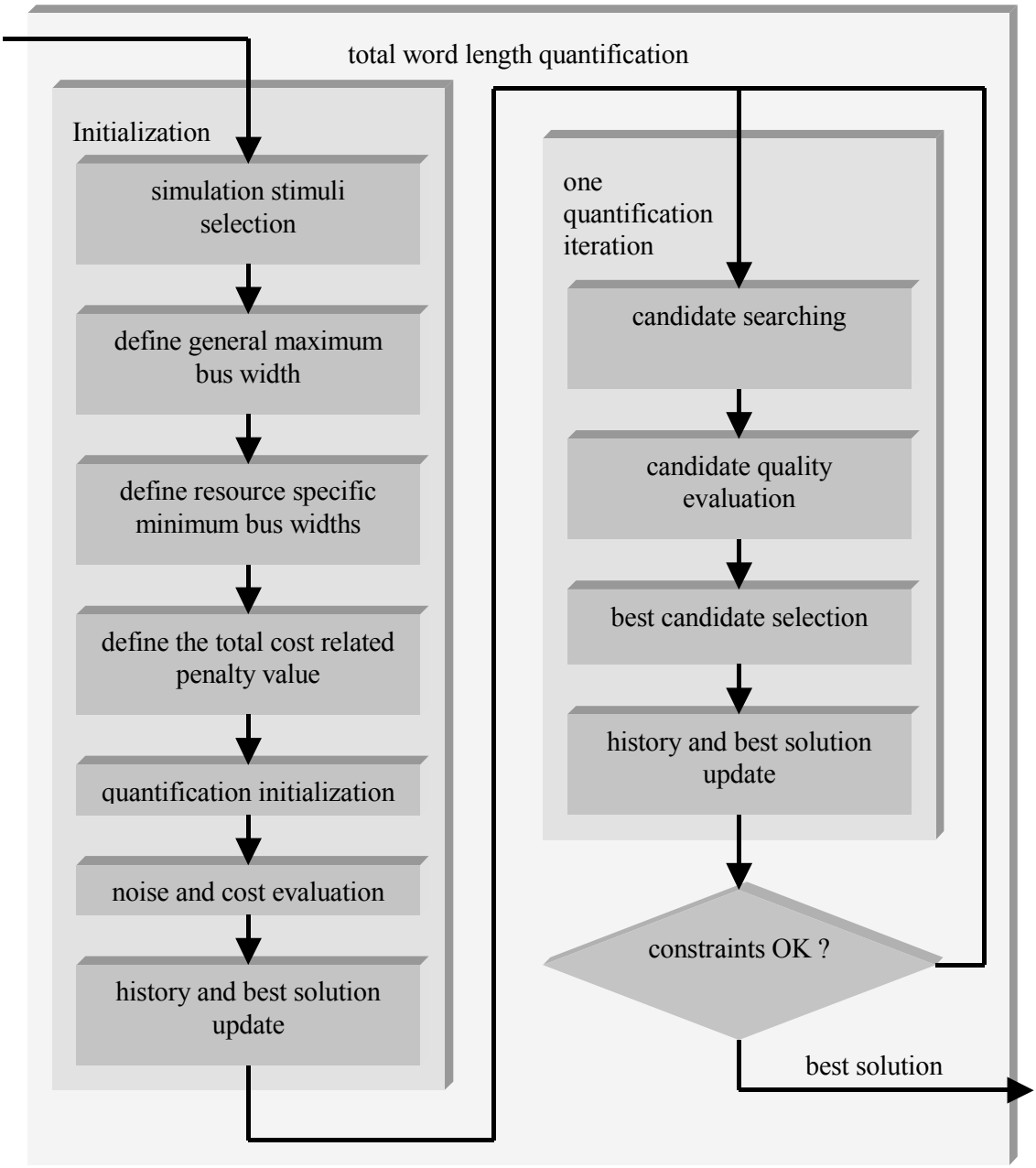


Abbildung 24: Der Programmfluss für die Bestimmung der Gesamtdatenwortbreiten

Zu Beginn der Initialisierungsphase werden geeignete Simulationsstimuli gesucht. In der existierenden Version des Synthesetools werden die Stimuli von einem Zufallsgenerator erzeugt, wobei der Anwender jedoch diese gewählten Stimuli auf ihre Eignung kontrollieren und gegebenenfalls korrigieren kann.

In den nächsten beiden Schritten wird der Suchraum eingeschränkt. Zuerst wird ermittelt, ab welchen Datenwortbreiten bei einer Simulation Überläufe

auftreten können[1] und der Suchbereich darüber abgegrenzt. Dann wird individuell für jede Operation bzw. Ressourceneinheit ermittelt, welches seine minimal notwendige Datenwortbreite ist. Dies definiert eine Optimierungsgrenze nach unten, d.h. zu den kleinen Datenwortbreiten hin.

Im nächsten Schritt wird der Straffaktor berechnet.

Dann folgt die Initialquantisierung, bei denen allen Operationen/Ressourcen eine Initialdatenwortbreite zugewiesen wird. Diese erste Lösung wird dann bezüglich Kosten und Datenverarbeitungsqualität analysiert und in das *History Table* des *Tabu-Search*-Algorithmus eingetragen.

Die Optimierungsphase selbst besitzt die vom Tabu-Search-Algorithmus benötigten Operationen. Dies sind: Suche der Umgebungskandidaten, Analyse dieser Kandidaten, Wahl des besten Kandidaten und *Update* des *History Tables*. Diese Operationen werden so lange wiederholt, bis eine Abbruchbedingung erfüllt ist. Es wird schlussendlich diejenige Lösung verwendet, welche die geringsten Kosten hat und welche zugleich sämtliche SNR-Randbedingungen erfüllt.

Diskussion:

- Konvergenzschwierigkeiten: Damit die optimalen Gesamtdatenwortbreiten mittels Simulation und Verwendung eines iterativen Optimierungsalgorithmus ermittelt werden können, müssen die zu implementierenden Algorithmen folgende Kriterien erfüllen:
  - Es dürfen nur Operationen verwendet werden, welche in der quantisierten Form eine direkte Abhängigkeit zwischen der Datenwortbreite und dem Quantisierungsfehler haben, d.h. wenn die Datenwortbreite grösser wird, muss der Quantisierungsfehler kleiner werden. Alle üblichen, arithmetischen Operationen wie Multiplikation, Addition, Subtraktion, Division, *Shift*, usw. erfüllen dieses Kriterium.
  - Ein Algorithmus darf keine konditionellen Verzweigungen besitzen. Besitzt er solche, muss bei der Gleitkomma- und Fixkomma-Simulation sichergestellt werden, dass beide Simulationen den Algorithmus auf die selbe Weise bearbeiten. Bei der existierenden Version des Synthesetools kann dies nicht garantiert werden.
- Bezüglich einer nächsten Version des Synthesetools könnten folgende Verbesserungen diskutiert werden:

[1] Für die quantisierte Simulation wird das Datenformats „*Long*“ verwendet, welche eine Breite von 32 Bit besitzt.

- Wahl der Stimuli: Wie bei der Festlegung der *Integer Bits* werden die verwendeten Stimuli auf eine sehr einfache Weise definiert. Es sollte nach Methoden gesucht werden, welche besser effiziente und geeignete Stimuli definieren.
- Man muss sich fragen, ob *Tabu-Search* für dieses Optimierungsproblem der geeignete Suchalgorithmus ist. Es gibt häufig Anwendungen, welche lediglich zwei arithmetische Ressourceneinheiten besitzen. Da die Anzahl der Dimensionen des Optimierungsproblems der Anzahl der arithmetischen Ressourceneinheiten entspricht, ist es fraglich, ob für solch „kleine" Anwendungen wirklich ein iterativer Optimierungsalgorithmus verwendet werden soll, oder ob ein einfacherer Algorithmus verwendet werden soll, welcher zum Beispiel auf dem Verfahren der *successiven Approximation* beruht.
- Es werden lediglich die Ausgänge arithmetischer Operationen quantisiert. Es sollte auch in Betracht gezogen werden, die Eingänge arithmetischer Ressourceneinheiten, sowie Zwischenregister zu zwischen-quantisieren.

4.3.6 Das Anwenderinterface mit den Anwendungsapplikationen

Das DSP-Synthesetool kann als Kommandointerpreter, im Graphikmodus und in einem gemischten Modus gestartet werden. Als erstes liest das Tool ein Initialisierungsskript, welches verschiedenste Einstellungen definieren kann und vorhandene Anwenderskripts und *Plugins* lädt.

Die Arbeit kann vereinfacht werden, wenn das Synthesetool im Graphikmodus gestartet wird und eine graphische Bedienungsoberfläche geladen wird, welche in Tcl/Tk realisiert werden kann [19]. DspC_GUI[1] ist eine solche Oberfläche (Abbildung 25). Zur einfachen Handhabung enthält sie zur linken Seite Knöpfe, welche, wenn sie der Reihe nach angewählt werden, die verschiedenen Verarbeitungsschritte des Designflusses automatisch in der richtigen Reihenfolge durchführen. Zur Übersicht wird im rechten, unteren Fensterteil ständig der Status der Datenbasis angezeigt. Der rechte, obere Fensterteil ist für die ladbaren Module reserviert. Verschiedenste zusätzliche Funktionen können via das *Pulldown*-Menü gewählt werden.

Wird im DSP-Synthesetool ein Modul geladen, so wird ebenfalls im Frontend, das heisst auf Niveau Tcl/Tk von einer Datei eine Oberflächenerweiterung geladen. Diese Erweiterungen beanspruchen hauptsächlich den rechten, oberen Fensterteil, wo sie ein zusätzliches Menü

[1] DspC_GUI: *Dsp Compiler's Graphical User Interface*

und je nach Modul unterschiedliche Eingabemöglichkeiten und Anzeigen erzeugen.

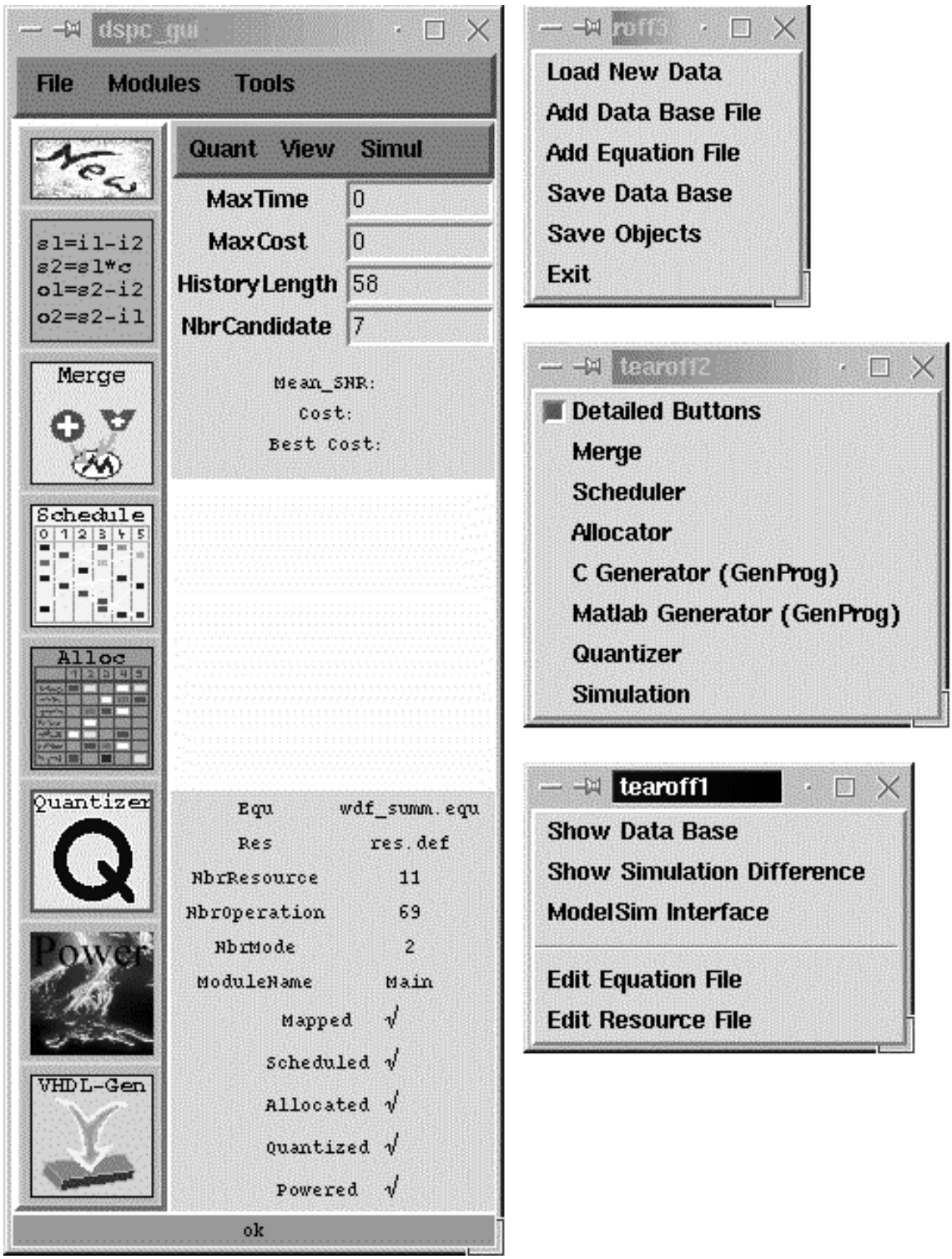


Abbildung 25: DspC_GUI: Das graphische Frontend des DSP-Synthesetools wurde mit Tcl/Tk realisiert. Nebst dem Hauptfenster sind auch die aus der Anwendung herausgezogenen Hauptmenüs abgebildet (tearoff menus).

Genügt der zugeteilte Platz nicht, so können auch zusätzliche Fenster geöffnet werden. Dies wird von denjenigen Modulen gemacht, welche

iterative Optimierungsalgorithmen verwenden. Zur Überwachung können diese den Optimierungsablauf in einer Graphikanimation visualisieren. Abbildung 26 zeigt die Fenster der Module für die Ressourcenzuordnung (*Binding*), die Quantisierung der Operatoren und für das Time-Scheduling. Während dem Time-Scheduling sieht der Anwender, wie die verschiedenen Operationen zur Optimierung hin- und hergeschoben werden. Ein weiteres, kleineres Fenster zeigt zudem die notwendige Anzahl der Ressourcentypen zum aktuellen Optimierungszeitpunkt an. Während der Ressourcenzuordnung werden die Stellen gezeigt, wo ein Austausch der Operationen stattfindet. Und bei der Quantisierung kann beobachtet werden, wie die Wortlängen der Operationen variieren.

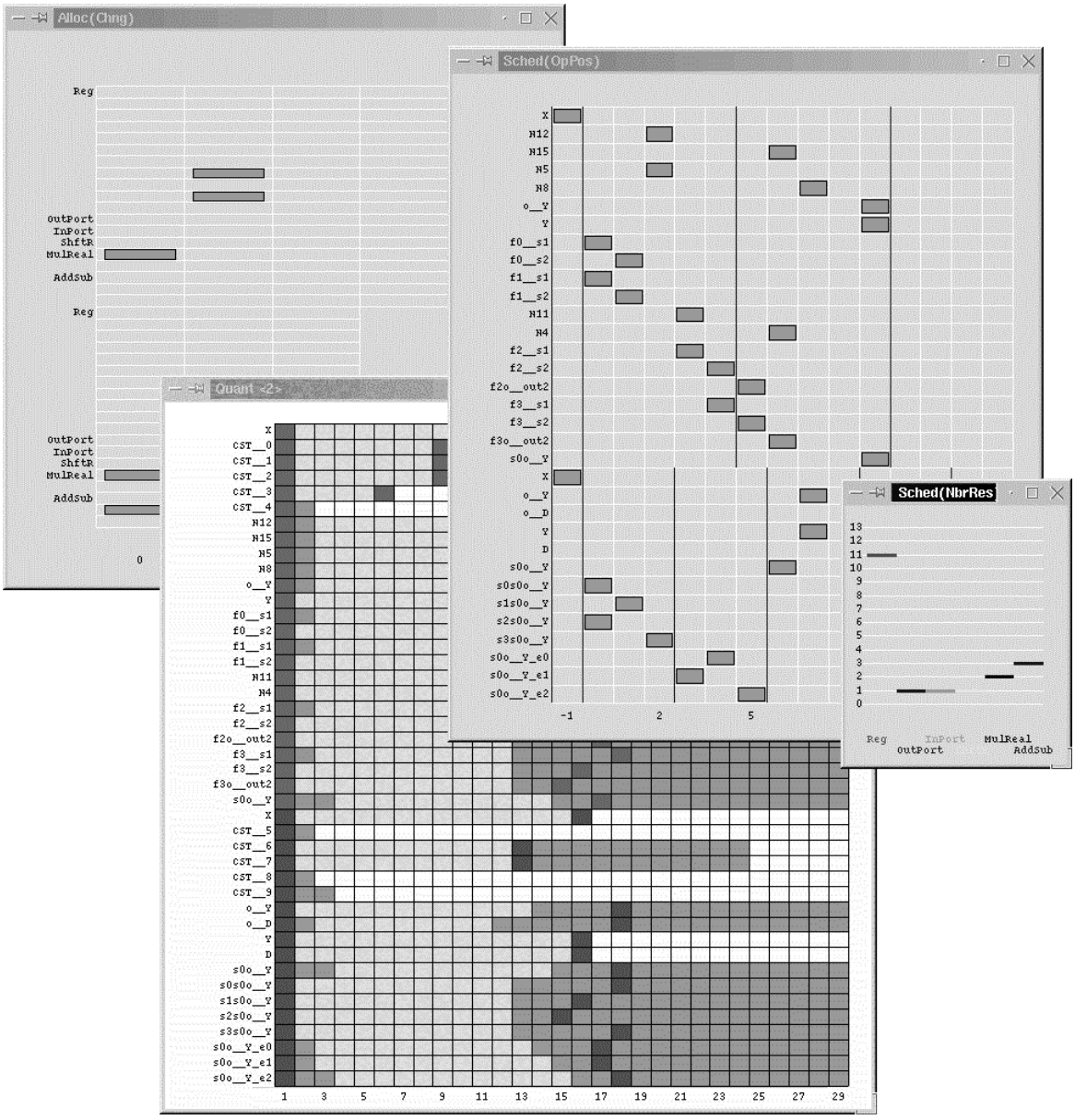


*Abbildung 26: Graphikanimationen visualisieren die verschiedenen iterativen Optimierungen. Von links nach rechts sind die Fenster für die Ressourcenzuordnung (*Binding*), die Quantisierung der Operatoren und das Time-Scheduling zu sehen.*

Das Tcl/Tk-Interface öffnet das Synthesetool und macht es beliebig erweiterbar. Zusätzliche Menüs, Funktionen, Applikationen und Interfaces zu anderen Programmen können auf einfachste Weise realisiert werden. Lediglich zwei Erweiterungen sollen noch kurz vorgestellt werden (Abbildung 27). Die erste Applikation visualisiert graphisch die Ressourcenzuordnung der Operatoren und deren Parameterabhängigkeiten und ist sehr praktisch, wenn zum Beispiel für eine Fehleranalyse verstanden werden muss, wie der Datenfluss im generierten Designs ist. Ebenfalls die zweite Applikation kann bei einer fehlerhaften VHDL-Simulation helfen, die Ursache des Problems zu finden. Sie erlaubt, Zwischenresultate einer internen Simulationen mit denjenigen einer VHDL-Simulation zu vergleichen und Differenzen graphisch hervorzuheben. Auf diese Weise kann die Operation, welche den Fehler erzeugt, schnell und bequem gefunden werden.

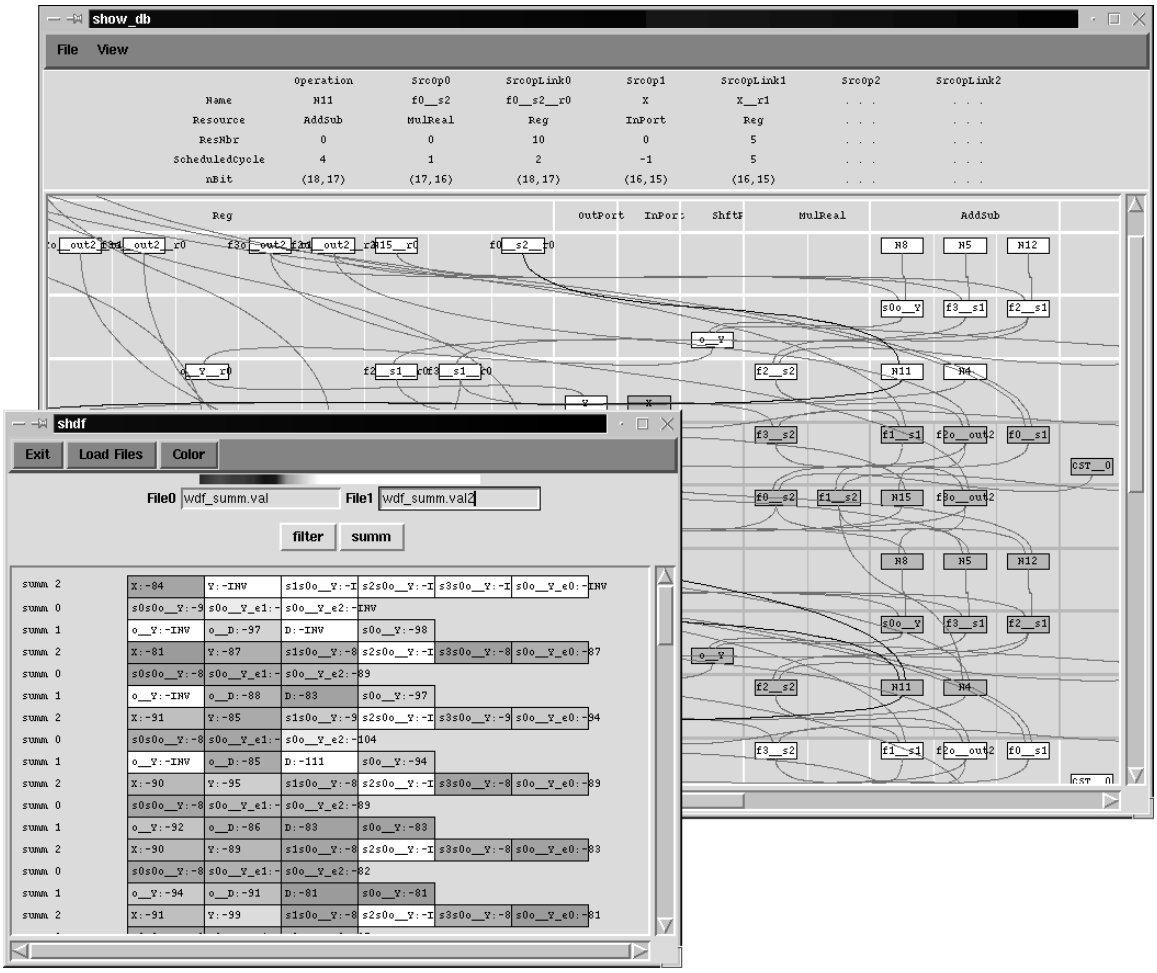


Abbildung 27 zeigt zwei praktische Tcl/Tk-Applikationen. Im grösseren Fenster wird graphisch die Ressourcenzuordnung der Operatoren und die Parameterabhängigkeiten dargestellt. Die zweite Applikation vergleicht die Zwischenresultate einer internen Simulation mit denjenigen einer VHDL-Simulation und färbt die Operationen in Funktion der Differenzen verschiedenfarbig ein.

4.4 Anwendungsbeispiel und Resultate

Nebst den eigentlichen Filterapplikationen, wo in regelmässigen Abständen Samples in den Filterblock fliessen, und diesen nach einer gewissen Zeit verarbeitet wiederum verlassen, kann das DSP-Synthesetool auch zur Generierung von aufwendigen, arithmetischen Verarbeitungseinheiten eingesetzt werden. Ein ausgezeichnetes Anwendungsbeispiel liefert der im nächsten Kapitel entwickelte FFT-Prozessor.

Ein hoher Datendurchsatz bei einer tiefen Taktrate ist eine an den FFT-Prozessor gestellte Randbedingung. Nur ein Datenpfad, welcher eine hohe Parallelität, eine Datenpipeline und Register zur lokalen Speicherung von Zwischenresultaten hat, kann diese Bedingungen erfüllen. Nebst den einfachen Grundoperationen muss der Datenpfad auch komplexere Operationen wie die Radix-4-Operation auf eine effiziente Weise verarbeiten können. Wenn man bedenkt, dass eine Radix-4-Operation bereits 34 arithmetische Operationen hat, der Datenpfad daneben aber noch 4 weitere Operationen zu verarbeiten hat, und wenn man dann noch in Betracht zieht, dass all diese Operationen parallel, von mindestens 6 verschiedenen Multiplikatoren und Addern verarbeitet werden müssen, erkennt man, dass der Zeitaufwand für den Bau eines solchen Datenpfads erheblich ist.

Das Synthesetool kann dieses Problem in einigen Minuten lösen. Der zeitlich grösste Aufwand ist, die verschiedenen Funktionsmodi des Datenpfads in mathematischer Forme auszudrücken und die Interfacebedingungen genau zu beschreiben. Das auf Seite L des Anhangs I befindende Listing 11 enthält die vollständige Beschreibung des für den FFT-Prozessor benötigten Datenpfads.

Je nach dem verwendeten Computer hat das Synthesetool bereits nach einer Minute (Linux auf Pentium III, 450 MHz) bis 5 Minuten (Solaris 2 auf SUN Ultra 2, 180 MHz) eine gute Lösung gefunden. Dass sie in Sachen Qualität den handgeschriebenen VHDL-Lösungen in nichts nachsteht, zeigt Tabelle 5 [1].

Bei der Interpretation der Tabelle müssen jedoch folgende prinzipielle Differenzen der drei Lösungen berücksichtigt werden:

- Das DSP-Synthesetool verwendet zwei Multiplikatoren, die anderen Lösungen zwei MAC.

[1] Wegen mangelnden Informationen innerhalb der verwendeten Standardzellbibliothek konnte keine Verlustleistungsanalyse durchgeführt werden. In einem anderen Anwendungsbeispiel im Kapitel „Qualitätsvergleich“ auf Seite R des Anhangs I wird dies nachgeholt.

- Die beiden VHDL-Lösungen können die Koeffizienten bei ihrer Verarbeitung invertieren. Dies kann ohne grossen Aufwand mit den beiden MACs gemacht werden. Die Lösung des DSP-Synthesetools hat diese Möglichkeit nicht implementiert.
- Die Lösung des DSP-Synthesetools verwendet arithmetische Einheiten von DesignWare™, einer Bibliothek mit u.a. arithmetischen Einheiten von Synopsys. Die beiden anderen Lösungen verwenden für jeden MAC einen nach Booth kodierten Wallace-Tree, gefolgt von einer Brent&Kung-Addiererstufe (VHDL-Strukturbeschreibung).
- VHDL-Lösung II besitzt am Ende der Datenverarbeitungskette Register, welche bei den beiden anderen Lösungen fehlen.

Ein weiteres Anwendungsbeispiel mit detaillierten Analysen bezüglich Grösse und Verlustleistung sowie mit Quervergleichen zu anderen Ansätzen ist im Kapitel „Qualitätsvergleich“ auf Seite R des Anhangs I zu finden.

		DSP-Synthesetool	VHDL-Lösung I	VHDL-Lösung II
Entwicklungszeit	Problemstellung	1 Tag	1 Tag	1 Tag
	Implementierung	5 Minuten	5 Tage	5 Tage
Architektur	# *Gated Clocks*	4	13	11
	# Mult.	2	-	-
	# MAC	-	2	2
	# Adder/Substr	4	4	4
	# Register[1] (16/18 Bit)	11+12	21+14	27+14
	# Mux[2] (16/18 Bit)	96 + (11+12)	49 + (21+14)	33 + (27+14)
Gatecount	Register	2294	3054	2712
	Komb. Logik	7786	8248	7806
	Total	10081	11303	10519

Tabelle 5: Vergleich einer DSP-Synthesetool-Lösung mit handgeschriebenen VHDL-Lösungen. Zur Ermittlung des Gatecount wurde die CMOS 0.35-µm-Technologie Alcatel-Mietec verwendet.

4.5 Schlussfolgerung

Das präsentierte *high-level* DSP-Synthesetool ermöglicht eine rasche Implementierung von anwendungsoptimierten DSP-Architekturen. Vom Moment an, wo ein DSP-Algorithmus und die Designrandbedingungen definiert sind, vergehen nur einige Minuten, bis das Tool eine

[1] Der erste Wert entspricht dem Zwischenregister, der zweite Wert dem Eingangsregister der arithmetischen Einheiten.
[2] Der in der Klammer stehende Teil entspricht der Anzahl der zusätzlich notwendigen Multiplexer bei der Generierung eines synchronen Designs.

synthetisierbare VHDL-Beschreibung von einer optimalen DSP-Architektur zusammen mit einem VHDL-Testbench und Simulationsreferenzen generiert hat. Durch die Verwendung des Synthesetools kann das Risiko von unachtsam eingeführten Fehlern während des Syntheseprozesses minimiert werden, da Anwenderinteraktionen laufend überprüft werden. Designrandbedingungen können in einer einfachen Syntax definiert werden. Sie garantiert die einfache Einfügung eines generierten Blockes in eine bestehende Umgebung. Die Quantisierung der verschiedenen internen Operationen eines Algorithmus wird automatisch durchgeführt. Der Anwender muss dazu lediglich die gewünschte Qualität der Datenverarbeitung spezifizieren. Das DSP-Synthesetool kann entweder voll synchrone Architekturen generieren, oder verwendet aber *Gated Clocks* für Low-Power-Anwendungen. Der Anwender kann den Designfluss zu jedem Zeitpunkt überwachen und steuern und hat freien Zugriff auf die interne Datenbasis des Tools. Zusammen mit dem TCL/TK-Interface und dem Plug-In-Loader wird dadurch die Realisierung von anwendungsspezifischen Erweiterungen ermöglicht. All diese Eigenschaften machen das DSP-Synthesetool zu einem effizienten Werkzeug für die Entwicklung von qualitativ hochstehenden, anwendungsspezifischen DSP-Architekturen.

Selbstverständlich hat das Synthesetool nicht nur Vorteile. Je nach Typ der Designaufgabe kann das Synthesetool auch suboptimale Lösungen generieren (siehe auch „Qualitätsvergleich“, Seite R des Anhangs I). Man darf also nicht von einer generellen Eignung des Synthesetools zur Lösung von Designaufgaben sprechen. Für jede Designaufgabe müssen individuell die Vorteile des Synthesetools, wie etwa die Reduktion der Entwicklungszeit den eventuellen Nachteilen gegenübergestellt werden. Die Wahl des Synthesetools als schlussendlich verwendete Implementierungsmethode kann dann oftmals ein wohlausgewogener Kompromiss zwischen den Vor- und Nachteilen dieser gewählten Methode sein.

Ein aus didaktischer Sicht negativer Punkt bezüglich der Methode ist sicherlich, dass oberflächliche Kenntnisse eines Designers in digitaler Datenverarbeitung und in Filter- und Architekturentwurf ausreichen, um qualitativ hochstehende Filter und Datenpfade zu implementieren. Eine weitere Frage bezüglich der Methode betrifft die Wirtschaftlichkeit von Entwicklungen von solchen Spezial-Designtools. Die Entwicklungszeit eines Synthesetools ist erheblich und muss durch die erwartete Effizienzsteigerung gerechtfertigt werden.

Das Tool als solches besitzt ebenfalls Einschränkungen, deren sich ein Anwender bewusst sein muss und die eine Entwicklungsfortsetzung des

Tools erlauben, wenn nicht gar verlangen. Folgende Liste zeigt die wichtigsten Einschränkungen des Synthesetools:

- Das Synthesetool kann bei fixen, unkonditionellen DSP-Algorithmen verwendet werden. Konditionelle Verzweigungen innerhalb von DSP-Algorithmen können nicht verarbeitet werden.
- Das Synthesetool kennt keine Up- oder *Downsampling*-Funktion. Muss ein Up- oder *Downsampling* realisiert werden, muss diese Funktion implizit im Algorithmus beschrieben werden.
- Das Tool kennt nur Ressourceneinheiten mit einem einzigen Resultatswert. Werden Ressourcen benötigt mit mehreren Resultaten (zum Beispiel ein Adder mit einem zusätzlichen Ausgangs-*Carry*-Bit), so muss dies mit einer kombinierten Ressourceneinheiten-Gruppe realisiert werden.
- Die Aufteilung des Designflusses in verschiedene, sequentiell abgearbeitete Operationen garantiert eine schnelle Filtersynthese und einfache Optimierungsalgorithmen. Eine kombinierte Optimierung des Time-Schedulings und der Ressourcenzuweisung könnte jedoch eine effizientere DSP-Architektur gewährleisten. Für diese Optimierungsproblematik werden heutzutage immer häufiger genetische Algorithmen eingesetzt. Es wäre interessant, die Eigenschaften dieser Algorithmen für eine mögliche Verwendung in dem DSP-Synthesetool zu untersuchen.
- Die Verwendung des *Sign-Manitude*-Datenformates innerhalb einer DSP-Struktur kann bei Datenströmen mit einer niedrigen Amplitude zu einer interessanten Reduktion der Systemaktivität und damit des Stromverbrauches führen. Damit das DSP-Synthesetool dieses Datenformat verarbeiten kann, werden nur sehr wenige und kleine Anpassungen des VHDL-Generator-Moduls benötigt. Die Hauptanpassungen können via Modifikation der Ressourcen-Definitionsdatei durchgeführt werden.

Kennt ein Designer die erwähnten Einschränkungen und kann er sie umgehen, so garantiert das *high-level* DSP-Synthesetool optimale Implementierungslösungen. Ansonsten muss das Tool zum Beispiel über das Tcl-Interface oder den Plug-In-Loader anwendungsspezifisch erweitert werden.

Referenzen

[1] A. Drollinger, A. Heubi, P. Balsiger, F. Pellandini, „Ein DSP-Synthesetool für schnelle Low-Power-Implementierungen von DSP-Algorithmen“, DSP Deutschland 2000, München, 10. – 12. Oktober 2000

[2] A. Drollinger, A. Heubi, P. Balsiger, F. Pellandini, „ASIC DSP Compiler for Optimized Synthesis“, ICSPAT 2000, International Conference on Signal Processing Applications & Technology, Dallas, Texas, USA, October 16-19, 2000.
[3] M. Rim, R. Jain, “Estimating lower-bound performance of schedules using a relaxation technique”, International Converence on Computer-Aided Design, Santa Clara, CA, pp. 290-294, November 1992
[4] A. Sharma, R. Jain, “Estimating architectural resources an performance for high-level synthesis applications”, Design Automation Conference, Dallas, TX, pp. 355-360, June 1993
[5] Heubi, P. Balsiger, F. Pellandini, "An Automated Design Methodology for the Mapping of DSP Algorithms into Low Power VLSI Architectures", 7th International Symposium on IC Technology, Systems & Applications, Singapore, September 10-12, 1997
[6] B. S. Haroun, M. I. Elmasry, “Architectural synthesis for DSP silicon compilers”, IEEE Tansacions on Computer-Aided Design, vol. 8, pp. 431-447, April 1989
[7] C. H. Gebotys, M. Elmasry, “Global optimization approach for architecture synthesis”, IEEE Transactions on Computer-Aided Design, vol. 12, pp. 1266-1278, September 1993
[8] C. Wang, K. K. Parhi, “MARS: A high-level DSP synthesis tool integrated within the Mentor Graphics environnement”, Mentor Graphics Users' Group, 12th Annual International Conference, October 23-27, 1995, Portland Marriott - Portland, OR
[9] DSP StationTM von Frontier Design: http://www.frontierd.com/dspstation.htm
[10] H. de Man et al. “Cathedral: A Synthesis and Module Generation System for Multiprocessor Systems on a Chip”, Design Systems for VLSI Circuit, G. De Micheli, Martinus Nijhoff Publ., 1987
[11] A|RT von Frontier Design: http://www.frontierd.com/art.htm
[12] Open SystemC Initiative: http://www.systemc.org
[13] Behavioral Compiler von Synopsys: http://www.synopsys.com/products/beh_syn
[14] A. Heubi, P. Balsiger, F. Pellandini, "An Automated Design Methodology for the Mapping of DSP Algorithms into Low Power VLSI Architectures", 7th International Symposium on IC Technology, Systems & Applications, Singapore, September 10-12, 1997.
[15] M. C. Mc Farland, A. C. Parker, R. Camposano, “The high-level synthesis of digital systems”, Proceedings of the IEEE, vol 78. pp. 301-318, February 1990
[16] S. Amellal, B. Kaminska, “Scheduling Algorithm in Data Path Synthesis Using the Tabu Search Technique”, Proceedings of the European Conference on Design Automation with the European Event

in ASIC Design. Paris, France, February 22-25 1993.IEEE Computer Society Press, 1993.
[17] S. Amellal, B. Kaminska, “Functional Synthesis of Digital Systems with TASS”, IEEE transactions on computer-aided design of integrated circuits and systems, vol 13., 5 May 1994
[18] F. Glover, “Tabu Search, Part I+II”, ORSA J. Computing, pp. 4-32/190-206, 1989
[19] J. K. Ousterhout, “Tcl and the Tk Toolkit”, Addison-Wesley Publishing Company, Inc, ISBN 0-201-63337-X

5 *Eine makroprogrammierbare, power- und recheneffiziente DSP-Architektur mit spezieller Eignung für FFT-basierte Algorithmen*

Dieses Kapitel präsentiert eine neue, makroprogrammierbare DSP-Architektur, welche aus einem Algorithmus mit einer guten Eigenstruktur so einen Vorteil ziehen kann, dass im Gegensatz zu einer Lösung mit einem frei programmierbaren DSP die Verarbeitungsgeschwindigkeit erhöht und der Energieverbrauch reduziert wird.
Die DSP-Architektur ist eine Spezial-Architektur mit einer Instruktionscodierung, welche sehr effizient ist für regelmässige Algorithmen. Der Datenpfad wird als eine VLIW-Architektur-basierte Prozessoreinheit implementiert, welche ein anwendungsspezifisches Instruktionsset besitzt, das hierarchisch als eine erste Programmebene betrachtet werden kann. Der Makrocode kann demnach als eine zweite Programmebene betrachtet werden.

Ein erster Kapitelteil erörtert die Motivation zum Bau der neuen DSP-Architektur und analysiert die Struktur von FFT-Algorithmen. Es folgt die Konstruktion der Spezialarchitektur. Der letzte Kapitelteil zeigt an Hand eines FFT-Prozessors die praktische Verwendung der makroprogrammierbaren DSP-Architektur.

5.1 Prozessoren für FFTs

Die schnelle Fouriertransformation (FFT[1]) ist ein Algorithmus, welcher eine sehr effiziente Spektrumanalyse und Frequenzfilterung eines Signals erlaubt. Im Gegensatz zur generellen Fouriertransformation erlaubt sie im Falle, wo die Anzahl der Stützstellen eine Potenz von 2 ist, die Anzahl der mathematischen Operationen zur Spektrumsanalyse eines Signals mit der Länge von N Abtastwerten, von N^2 auf N*log2(N) zu reduzieren. Dadurch können Signale schneller und mit weniger Energieaufwand analysiert werden.

In portablen Geräten kann die FFT die ideale Basis für einen Filter- oder Analysealgorithmus bilden, jedoch nur unter Voraussetzung, dass die FFT mit einer sehr geringen Energie durchgeführt werden kann. Energieverbrauch und die Berechnungszeit zur Durchführung einer FFT sind deshalb wichtige Benchmark-Werte eines DSPs. Folgende Tabelle zeigt die Benchmark-Ergebnisse einer freien Auswahl existierender DSPs [1].

Processor	Year	CMOS Tech (µm)	Datapath Width (bits)	Dataword Format[2]	Supply Voltage (V)	1024-pt-FFT Execution Time (µs)	Power (mW)	Clock (MHz)	Energy / 1024-pt-FFT (µWs)
Programmierbare DSPs									
C40, Texas Instruments	-	0.7	-	float	5	**1298**	4500	60	**5841**
NM6403 ,Module	98	0.5	32	fixed	3.3	**439**	1300	50	**571**
HiPAR-DSP 4 ,Universitat Hannover	99	0.5	16	fixed	-	**222**	5000	66	**1110**
Spezial-FFT-Prozessoren									
PDSP16510A Mitel (Plessey)	89	1.4	16	bfloat	5	**98**	3000	40	**294**
CNET E. Bidet	94	0.5	10	-	3.3	**51**	300	20	**15.3**
Spiffee 1, Stanford	95	0.8	20	fixed	3.3	**30**	845	173	**25.35**
Spiffee Low Vt * Stanford	95	0.8	20	fixed	0.4	**93**	9.7	57	**0.902**
Spiffee ULP * Stanford	95	0.5	20	fixed	0.4	**61**	8	85	**0.488**
SNC960A Sicom	96	0.6	16	-	5	**20**	2000	65	**40**
DSP-24,DSP Architectures	97	0.5	24	bfloat	3.3	**21**	3500	100	**73.5**
M.Wosnitza,ETH, Zurich	98	0.5	32	bfloat	3.3	**80**	6000	66	**480**
DoubleBW	00	0.35	24	float	3.3	**10**	8000	128	**80**

Tabelle 6: FFT-Benchmark-Werte einiger DSPs

[1] FFT=*Fast Fourier Transformation*
[2] float=*floating point*, fixed=*fixed point*, bfloat=*block floating point*

Als Referenz für den Benchmark-Test dient eine 1024-Punkte-FFT für welche die Berechnungszeit und die notwendige elektrische Energie ermittelt wurde. Wie Tabelle 6 zeigt, existieren speziell in der Kolonne „Energieverbrauch“ enorme Differenzen zwischen den unterschiedlichen Prozessoren. Die verwendeten Technologien und Speisespannungen sind ein erster Grund für diese Differenzen. Ein weiterer Grund ist auch das verwendete Datenformat: So ist der Stromverbrauch der 32-Bit-Prozessoren sicherlich höher als derjenige des 10-Bit-Prozessors. Der wichtigste Faktor jedoch, welcher die riesigen Differenzen erklären kann, sind die unterschiedlichen Ansätzen, wie die Prozessorarchitekturen für die Verarbeitung von FFTs gebaut und optimiert wurden.

Die ersten drei Prozessoren in der Tabelle 6 sind *General Purpose DSPs*. Damit auch diese DSPs FFTs in einer gewissen Effizienz bearbeiten können, besitzen sie meist einige Spezialinstruktionen, welche es erlauben, die Grundoperationen einer FFT wie etwa eine Radix-2-Operation mittels einer einzigen Instruktion durchzuführen. Auch die Adressgeneratoren können oftmals so programmiert werden, dass die Daten- und Koeffizientenadressierung erleichtert wird. Diese Techniken optimieren zwar die Verarbeitung von FFTs, bilden jedoch noch lange nicht eine optimale Lösung bezüglich Energieverbrauch.

Für eine Energie-optimale Lösung zur Verarbeitung von FFTs ist die Verwendung einer Spezialarchitektur zwingend. Die letzten neun Kandidaten der Tabelle 6 sind solche Spezialarchitekturen, wobei speziell *Spiffee* mit einem extrem geringen Energieverbrauch ins Auge sticht [1]. Dieser geringe Energieverbrauch wird mittels einer Prozessorarchitektur, welche kompromisslos für die 1024-Punkte-FFT gebaut wurde, einem sorgfältig durchgeführten Handlayout und einer Spezialtechnologie erzielt. Wegen dieser totalen Fokussierung auf die 1024-Punkte-FFT besitzt dieser Prozessor jedoch im Gegensatz zum oben besprochenen *General Purpose DSP* überhaupt keine Flexibilität, was seine Verwendung völlig einschränkt.

In diesem Kapitel wird eine Prozessorarchitektur vorgestellt, welche zwar wie die oben beschriebenen Spezialarchitekturen FFTs optimal verarbeiten kann, jedoch zusätzlich eine solche Flexibilität besitzt, dass nicht nur etwa die FFT-Grösse konfigurierbar wird, sondern dass sie generell regelmässige Algorithmen verarbeiten kann.

5.2 Die Ineffizienz von frei programmierbaren Allzweck-DSPs bei der Implementierung von FFT-basierten Algorithmen

Bei der Verwendung eines *General Purpose DSP* erzeugt ein Assembler oder Compiler bei der Implementierung eines Algorithmus individuell für

jeden Teil, jeden Knoten und jede Operation des Algorithmus ein oder mehrere Maschineninstruktionen, welche zusammen mit zusätzlichen Kontrollinstruktionen den Maschinencode bilden. Die Algorithmus-relevanten Operationen sind die arithmetischen Operationen, welche in ALU-Instruktion übersetzt werden. Diese können die Adressierung von Quell- und Zielparameter direkt selbst beinhalten, oder aber müssen von zusätzlichen Instruktionen für die Adressierung der Parameter begleitet werden. Letzteres ist meistens zwingend der Fall wenn die Modifikationen der Parameteradressen nicht einfachsten arithmetischen Regeln wie etwa einer Zählererhöhung folgen. Ebenfalls der Gebrauch von Unterfunktionen verlangt nach Zusatzinstruktionen für die Parameterübergabe.

Konsequenz und Nachteil dieser individuellen Übersetzung der Operationen ist, dass die Grösse des Programmcodes und damit diejenige des benötigten Programmspeichers etwa proportional zu der Grösse des Algorithmus steht. Und weiter beanspruchen die Adress- und Kontrollinstruktionen Arbeitszyklen, was die Datenverarbeitungsgeschwindigkeit des DSPs drastisch reduziert und zudem die Energie, welche vom DSP zur Verarbeitung des Algorithmus benötigt wird, erhöht.

Die Schlussfolgerung ist, dass ein *General Purpose DSP* für strukturell gut organisierte Algorithmen wie FFTs höchst ineffizient ist, denn er verarbeitet einen gut strukturierten Algorithmus wie jeden anderen auch, ohne aus der Regelmässigkeit einen Vorteil zu schöpfen. Im weiteren steht die Menge der möglichen Operationen einer ALU oftmals in keinem Verhältnis zur Menge der tatsächlich gebrauchten Operationen. Eine unnötige Übergrösse der ALU und die fehlende Möglichkeiten, von einer gut organisierten Eigenstruktur eines Algorithmus zu profitieren, zeigen die schlechte Eignung von *General Purpose DSPs* für den diskutierten Algorithmustyp. Viel Fläche, ein hoher Bedarf an Programmspeicher, eine hohe Taktrate und ein mit all dem verbundenen, hoher Stromverbrauch sind die Folge.

5.3 Implementierungsvereinfachende Strukturierung von FFT-basierten Algorithmen [2]

In diesem Kapitelteil wird die Eigenorganisation von FFT-basierten Algorithmen detailliert analysiert. Die gewonnenen Erkenntnisse erlauben es, eine für die Verarbeitung von FFT-basierten Algorithmen effiziente DSP-Architektur zu entwickeln.

5.3.1 Die gut organisierte Struktur von FFT-basierten Algorithmen

Verschiedenste DSP-Algorithmen besitzen eine gut organisierte Eigenstruktur, die man an einem regelmässigen Datenflussgraph erkennen kann. Sie besitzen zudem oft nur eine kleine Menge verschiedener

Grundoperationen. Ein gutes Beispiel ist eine FFT: Die Menge der Operationen beschränkt sich auf die Radix-2-Operation. Mittels einer zusätzlichen Radix-4-Operation kann die Berechnung der FFT beschleunigt werden. Wird die Menge nochmals durch drei Operationstypen erweitert, können bereits aufwendige, FFT-basierte Algorithmen wie eine WOLA[1]-Filterbank realisiert werden. Die drei zusätzlich benötigten Operationstypen sind die Multiplikation, die MAC[2] und die komplexe Multiplikation.

Analysiert man den Datenfluss eines FFT-basierten Filterbank-Algoritmus so stellt man fest, dass dessen Struktur nicht nur graphisch Regelmässigkeiten aufweist, sondern dass unter gewissen Umständen ebenfalls die Adressierung der Daten und Konstanten einfachen Regeln folgt.

5.3.2 Ein Zeit-Frequenz-Zeit Spektralanalyzer

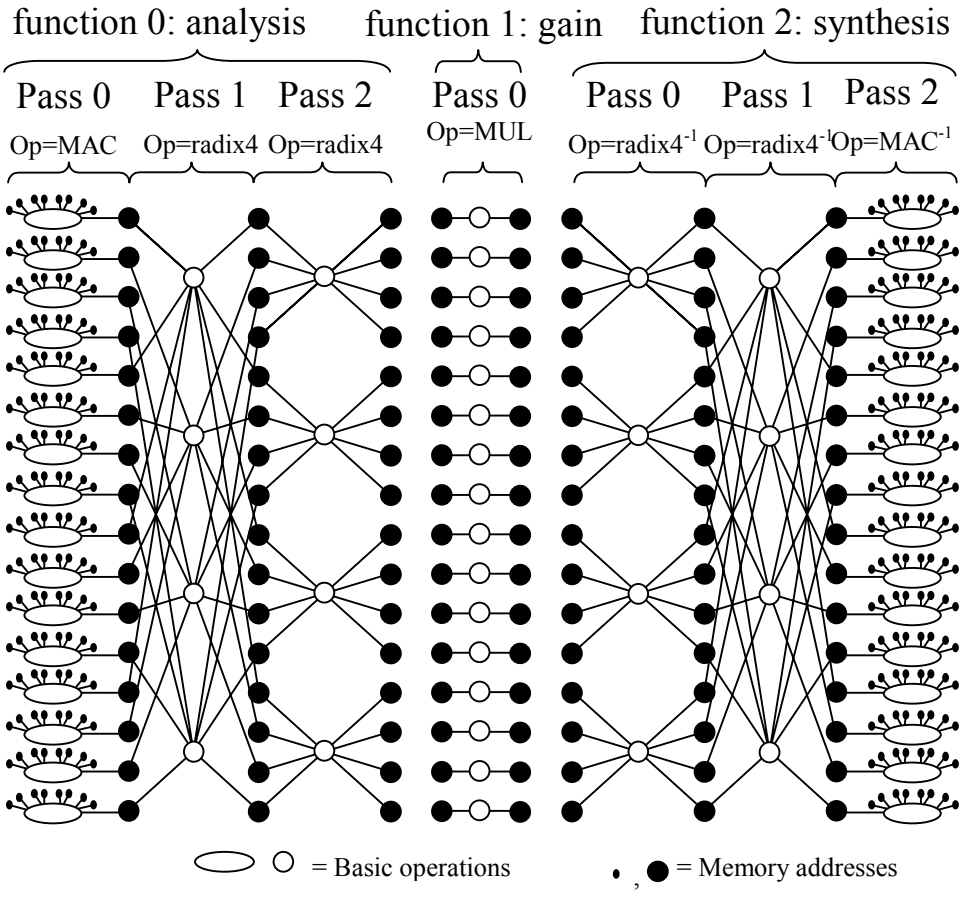


Abbildung 28: Schematischer 3-teiliger Datenfluss eines Zeit-Frequenz-Zeit-Spektralanalyzers

[1] WOLA: *Weighted OverLap Add (*WOLA-Algorithmus: Siehe auch folgendes Kapitel)
[2] MAC: *Multiplikation-Accumulation*

Abbildung 28 zeigt schematisch den Datenfluss eines Zeit-Frequenz-Zeit-Spektralanalyzers. Er kann einerseits eine Sequenz von Signalen (Samples) vom Zeitbereich in den Frequenzbereich transformieren. Dies wird oftmals als Analyse bezeichnet, da eine Frequenzanalyse der Signalsequenz durchgeführt wird. Andererseits kann er eine Rücktransformation vom Frequenzbereich in den Zeitbereich durchführen, was oftmals als Synthese bezeichnet wird. Weiter kann der Analyzer Verstärkungsfaktoren auf die verschiedenen Frequenzbänder anwenden (Modifikation der Frequenzbänder), was dem mittleren Teil entspricht.

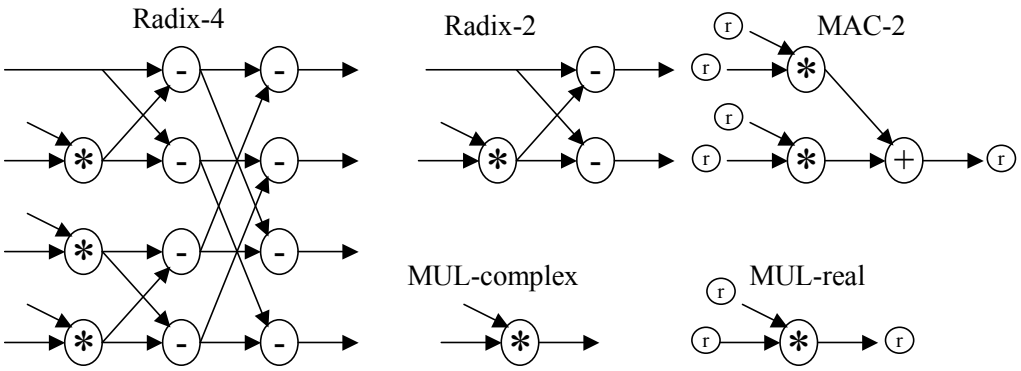


Abbildung 29: Operationen eines FFT-Spektralanalzyer. Alle nicht speziell mit dem real-Symbol (r) gekennzeichneten Daten sind komplex.

Der Datenflussgraph enthält lediglich 3 verschiedene Grundoperationen, deren Feinstrukturen zusammen mit derjenigen der Radix-2-Operation, welche auch für FFTs verwendet wird, in Abbildung 29 dargestellt sind.

5.3.3 Die Strukturierung des Dataflussgraphes

Bei der Betrachtung von Abbildung 28 fällt einem eine gewisse Organisation im Datenfluss auf. So sieht man, dass der gesamte Graph drei Hauptteile hat. Da ein solcher Hauptteil vom DSP meistens an einem Stück bearbeitet wird und somit eine Verarbeitungseinheit bildet, welche in eine Funktion (oder Prozedur) eingepackt werden kann, wird im Folgenden ein solcher Hauptteil eine *Funktion* genannt. Eine Funktion hat eine oder verschiedene Kolonnen von identischen Operationen. Eine solche Kolonne wird *Pass* genannt. Ein *Pass* ist aus verschiedenen *Operationen* zusammengesetzt, deren Feinstruktur wiederum verschiedene *Mikrooperationen* enthält.

Der Algorithmus wurde damit bereits unterteilt in *Funktionen*, *Pässe*, *Operationen* und *Mikrooperationen*. Es folgt eine präzisere Definition dieser Objekttypen begleitet mit resultierenden Eigenschaften.

- Eine *Funktion* ist ein Teil eines Algorithmus, welcher vom DSP ohne Unterbrechung, also an einem Stück, bearbeitet wird. Ein Zeit-Frequenz-Zeit-Spektralanalyzer hat zum Beispiel 3 Funktionen: Analyse,

Modifikation des Frequenzspektrums und die Synthese. Werden für eine Anwendung konstante Verstärkungsfaktoren verwendet, können Analyse, Spektrumsmodifikation und Synthese auch an einem Stück durchgeführt werden. In dem Fall ist der gesamte Algorithmus in eine Funktion eingepackt.

- Ein *Pass* ist eine Untergruppe einer Funktion und gruppiert identische Operationen zusammen in einer Weise, dass sämtliche Daten einmal vom Datenspeicher gelesen, bearbeitet und in den Datenspeicher zurückgeschrieben werden.
  ➔ Man stellt fest, dass jeder Pass 2^n Operationen hat (n=1,2,3,..).
  ➔ Wenn die Operationen eines Passes in einer bestimmten Reihenfolge bearbeitet werden, können die Adressen für Daten und Koeffizienten mit einem einfachen Zähler und einer sogenannten *Bit Reversing Function* erzeugt werden.
- Eine *Operation* kann direkt vom Datenpfad des DSPs ausgeführt werden. Addition, Multiplikation und Multiplikation-Addition sind typische Operationen. Grössere Datenpfade können zum Beispiel auch Radix-2- und Radix-4-Operation verarbeiten.
- Eine Operation kann aus verschiedenen *Mikrooperationen* zusammengesetzt sein, welche den elementaren arithmetischen Operationen entsprechen.

5.3.4 Vorteile und Erkenntnisse einer solchen Strukturierung

Die Implementierung eines Algorithmus, welcher auf die vorgestellte Art strukturiert wurde, profitiert einerseits von der Tatsache, dass ein Pass nur eine Art von Operationen hat, und andererseits von der vereinfachten Adressgenerierung mit Hilfe von *Bit Reversing Functions*.

Da ein Pass nur eine Operationsart hat, ist es nicht notwendig, jede einzelne Operation des Passes einzeln zu kodieren und in den Programmcode einzufügen. Es genügt, den Operationstyp einmal zu Beginn des Passes zu definieren. Da der Programmcode dadurch kleiner wird und weil ebenfalls weniger Zugriffe auf den Programmspeicher erfolgen müssen, kann einerseits die Grösse des Programmspeichers reduziert werden, andererseits kann die Verarbeitungsgeschwindigkeit erhöht werden. Gleichzeitig resultiert dies in einem tieferen Stromverbrauch.

Die selben Vereinfachungen und Vorteile sind möglich, wenn der Programmcode nicht für jede Operation einzeln Parameteradressen enthalten muss, sondern wenn die Adressen einer bestimmten Regel folgen, und lediglich diese Regeln im Programmcode gespeichert werden müssen. Die notwendige Grösse des Codewortes, welches eine Regel definiert, hängt von der Grösse der Menge aller möglichen Regeln ab (Abhängigkeit ~ $\log_2$ [Anzahl der Möglichkeiten]). Eine kleine Menge

möglicher Regeln erlaubt den Gebrauch eines kleinen Codewortes, was für den Bedarf des Programmspeichers wiederum von Vorteil ist.

Bei FFT-basierten Algorithmen kann die Menge der Adressierungsregeln auf eine Untermenge der Familie der sogenannten *Bit Reversing Functions* reduziert werden. *Bit Reversing Functions* modifizieren binäre Zahlen durch Neuordnung der einzelnen Bits. In Hardware kann dies mittels einer simplen Umverdrahtung realisiert werden und kostet so keine zusätzlichen Ressourcen. Die Adressgenerierung der Daten und der Koeffizienten, welche bei einem normalen DSP zusätzliche Ressourcen verlangen und zusätzliche Rechenzeit benötigt, kann somit viel effizienter realisiert werden.

5.4 Eine DSP-Architektur, welche speziell geeignet ist für FFT-basierte Algorithmen [2]

5.4.1 Pflichtenheft einer für die Verarbeitung von FFT-Algorithmen optimalen DSP-Architektur

Folgende Anforderungen und Bedingungen wurden an die DSP-Architektur gestellt:

- Minimale Grösse der Hardware: Dies beinhaltet einerseits die eigentliche Grösse des DSPs, andererseits auch die notwendige Grösse von Daten-, Koeffizienten- und Programmspeicher.
- Minimale Verlustleistung zur Bearbeitung eines Algorithmus: Dies resultiert in einer Minimierung der gesamten Aktivität des DSPs kombiniert mit der Minimierung der Anzahl von Speicherzugriffen.
- Hohe Flexibilität: Der DSP soll ein grosses Spektrum von Algorithmen verarbeiten können. Dies kann von der einfachen FFT bis zur aufwendigen WOLA[1]-Filterbank für die Verarbeitung von Mono- und Stereosignalen reichen.
- Tiefe Speisespannung: Eine Knopfzelle, deren Spannung gegen Lebensende 0.9 Volt erreichen kann, soll für die direkte Speisung des DSPs genügen.
- Minimale Taktrate von 1 MHz: Da sich die Schaltgeschwindigkeit von CMOS-Gates mit tieferwerdender Speisespannung drastisch reduziert, ist diese Bedingung eine direkte Konsequenz der letzten Bedingung. Bei einer Taktrate von 1 MHz sollen noch die wichtigsten FFT- und WOLA-Algorithmen verarbeitet werden können.

[1] WOLA: *Weighted OverLap Add (*WOLA-Algorithmus: Siehe auch folgendes Kapitel)

5.4.2 Der Bau der DSP-Architektur

Aus den definierten Randbedingungen des DSP können Folgerungen gezogen werden, welche den Bau des DSP beeinflussen.

- Eine Reduzierung des notwendigen Datenspeichers kann durch eine gut gewählte Verarbeitungsreihenfolge der Operationen und einer guten Wahl der Adressierung der Datenwörter erzielt werden.
- Indem zum Beispiel nur eine Hälfte von symmetrischen Fensterfunktionen gespeichert wird, kann die Grösse des Koeffizientenspeichers halbiert werden.
- Durch Vermeidung der Redundanz innerhalb des Programmcodes kann der Programmcode und damit auch der Programmspeicher reduziert werden. Gleichzeitig wird damit auch die Anzahl der notwendigen Lesezugriffe auf den Programmspeicher reduziert.
- Register innerhalb des Datenpfades, welche ein lokales Speichern von Zwischenresultaten erlauben, reduzieren den Zugriff auf den Datenspeicher.
- Damit ein umfangreicher Algorithmus bei lediglich einer Taktrate von 1 MHz verarbeitet werden kann, muss einerseits die Anzahl der Kontrollzyklen, an denen der Datenpfad nicht arbeitet, minimiert werden, andererseits muss der Datenpfad selbst verschiedene (arithmetische) Operationen gleichzeitig durchführen können. Dies ist möglich beim Gebrauch einer parallelen Datenpfad-Architektur, welche eventuell auch eine Datenpipeline hat.

Die Grundstruktur des DSPs

Keiner der in Kapitel 2.3 auf Seite 27 beschriebenen Prozessorentypen für die Parallelverarbeitung von Daten schien das in Kapitel 5.4.1 aufgestellte Pflichtenheft voll und ganz erfüllen zu können. Die Algorithmen liessen sich zwar in Gruppen gleichartiger Operationen unterteilen, um so mittels einer SIMD-Architektur gleichzeitig bearbeitet zu werden. Die Daten- und Koeffizientenadressierung innerhalb dieser Gruppen wäre jedoch so komplex, dass eine eigene Prozessoreinheit für die Adressierungen benötigt würde. Mittels einem VLIW-Prozessor kann eine parallele Datenverarbeitung realisiert werden, hat aber den Nachteil, dass kein Vorteil aus der Regularität des Algorithmus geschöpft wird und der Fluss der Programminformationen, und damit auch die Grösse des Programmspeichers und die Menge der Zugriffe auf diesen, beträchtlich sind. Dies sind die Gründe, weshalb für den hier vorgestellten Prozessor nicht eine der klassischen Parallelarchitekturen wie SIMD, MIMD oder VLIW verwendet wurde, sondern die auf den folgenden Seiten beschriebene Spezialarchitektur entwickelt wurde.

Die Wahl der DSP-Architektur wurde hauptsächlich beeinflusst von der Erkenntnis, dass innerhalb eines Passes sämtliche Operationen identisch sind, und dass die Adressen für Daten und Koeffizienten einer wohldefinierten Regel folgen. Das heisst, dass wenn bei Beginn eines Passes sämtliche Parameter wie Operationstyp und Adressenregeln definiert sind, die Informationen dieser Parameter genügen, um den gesamten Pass zu bearbeiten.

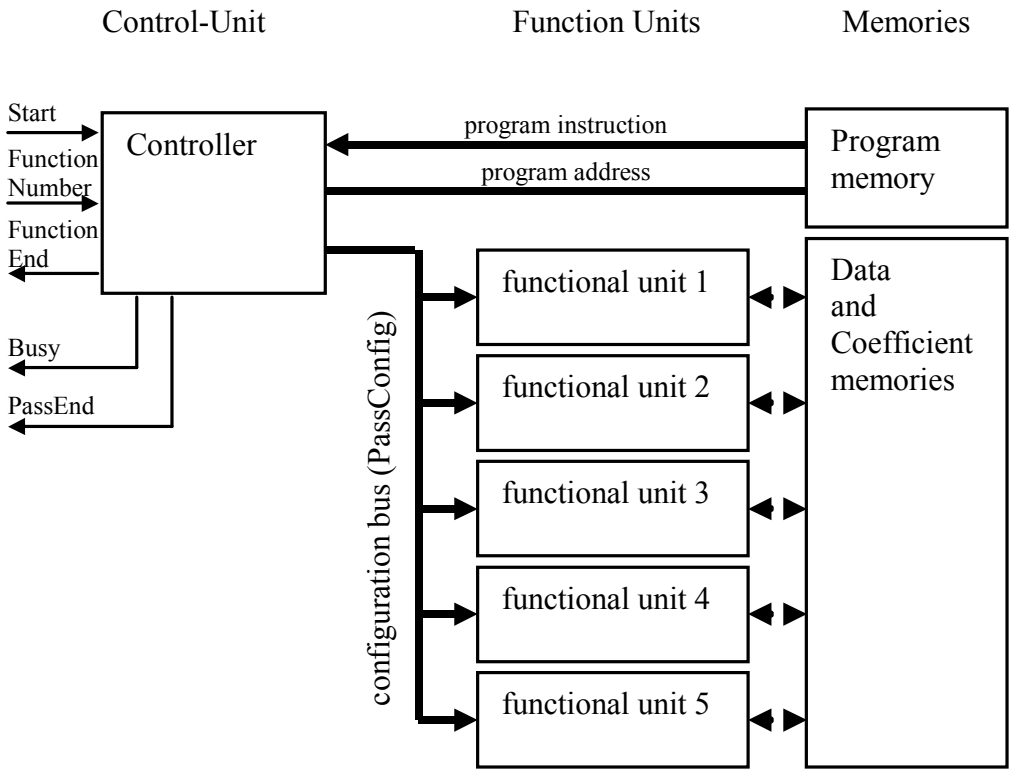


Abbildung 30: Die Grundstruktur des DSPs

Abbildung 30 zeigt, in welche DSP-Grundstruktur diese Erkenntnis transformiert wurde. Das Herz des DSPs ist die Kontrolleinheit. Zu Beginn jedes Passes definiert sie einen Passkonfigurationswert, welchen sie über den Konfigurationsbus den so genannten funktionalen Einheiten zur Verfügung stellt. Die funktionalen Einheiten sind dann für die Verarbeitung des gesamten Passes verantwortlich und enthalten dazu die notwendigen Ressourcen. Konkret sind dies der Datenpfad, Einheiten zur Erzeugung der Adressen und eventuell Schnittstellen zu Daten- und Koeffizientenspeicher. Weshalb und wie diese Aufgaben verschiedenen funktionalen Einheiten zugeteilt werden, soll zu einem späteren Zeitpunkt betrachtet werden. Nachdem die funktionalen Einheiten einen Pass bearbeitet haben, definiert der Kontroller das Konfigurationswort für den nächsten Pass und startet für dessen Bearbeitung wiederum die funktionalen Einheiten. Dies geschieht, bis sämtliche Pässe einer Funktion abgearbeitet sind.

Der Kontroller selbst wird von extern mittels eines Startsignals zum Bearbeiten einer Funktion gestartet. Von extern erhält er ebenfalls die Information, welche Funktion zu bearbeiten ist. Andererseits meldet er das Beenden der Funktionsbearbeitung zurück.

Der Programmcode

Passkonfigurationen sowie andere, zur Bearbeitung einer Funktion notwendige Informationen erhält der Kontroller vom Programmspeicher, der den Programmcode enthält.

Da der Kontroller zu Beginn jedes Passes die notwendigen Informationen zur Erzeugung der Passkonfiguration aus dem Programmspeicher liest, wäre es am Einfachsten, wenn sich jeweils die gesamte Passkonfiguration im Programmspeicher befindet. Dies würde jedoch zu redundanten Informationen innerhalb des Programmcodes führen, da gewisse Parameter während einer ganzen Funktion Gültigkeit haben, und nicht nur während eines Passes. Um die Redundanz innerhalb des Programmcodes zu reduzieren, wird deshalb ein Funktions-Konfigurationswort in den Programmcode eingefügt, welches sämtliche Informationen enthält, welche während einer ganzen Funktion Gültigkeit besitzen.

Einige Parameter können auch für den gesamten Algorithmus Gültigkeit besitzen, also für sämtliche Funktionen. Durch Einfügen eines globalen Konfigurationswortes kann auch hier vermieden werden, dass sich ein solcher Parameter mehrmals im Programmcode befindet.

Da der Programmcode nicht Instruktionen für einzelne Operationen enthält, sondern die Informationen paketweise für die einzelnen Pässe organisiert sind, und weil die funktionalen Einheiten nach Erhalt der Passkonfiguration den gesamten Pass wie ein Makro an einem Stück bearbeiten können, wird anstelle von Programmcode das Wort Makrocode verwendet.

Tabelle 7 zeigt die detaillierte Organisation des Makrocodes. Ein Konfigurationswort kann dabei je nach seiner Grösse eine unterschiedliche Anzahl von Speicheradressen belegen.

Am Beginn des Makrocodes befindet sich die globale Konfiguration, welche einerseits die Parameter enthält, welche für sämtliche Funktionen Gültigkeit haben, andererseits aber auch die Startadressen der einzelnen Funktionen.

Die Funktionskonfigurationen enthalten Parameter, welche für eine ganze Funktion Gültigkeit besitzen wie zum Beispiel die Anzahl der zu verarbeitenden Daten, aber auch die Anzahl der Funktionspässe.

Für jeden der Pässe folgt schliesslich die Passkonfiguration, welche die restlichen Informationen enthält.

globale Konfiguration
Konfiguration der 1. Funktion
Konfiguration des 1. Passes
Konfiguration des 2. Passes
...
Konfiguration der 2. Funktion
Konfiguration des 1. Passes
Konfiguration des 2. Passes
...
Konfiguration der 3. Funktion
Konfiguration des 1. Passes
Konfiguration des 2. Passes
...
Konfiguration der 4. Funktion
...

Tabelle 7: Die Organisation des Makrocodes: Jedes Konfigurationswort kann eine oder mehrere Speicheradressen besetzen.

Der Kontroller: Arbeitsweise und Architektur

Die Aufgabe des Kontrollers ist, eine Schnittstelle zwischen den externen Kontrollsignalen, dem Makrocodespeicher und den funktionalen Einheiten zu bilden und die Arbeit der letzteren zu steuern. Der Anhang I enthält auf Seite P eine VHDL-Verhaltensbeschreibung des Kontrollers und zeigt dessen Schnittstelle. Der Klarheit wegen beschränkt sich die Beschreibung auf das Wesentliche. So belegen zum Beispiel sämtliche Konfigurationswörter lediglich eine Adresse im Makrocodespeicher, auf ein globales Resetsignal wurde verzichtet und das System ist synchron getaktet, d.h. es werden keine *Gated Clocks* verwendet.

Bei Aktivierung des Startsignals liest der Kontroller zuerst das globale Konfigurationswort und dann das Funktions-Konfigurationswort aus dem Makrocodespeicher. Die Startadresse der Funktion ist dabei in der globalen Konfiguration enthalten. Die Funktionskonfiguration enthält u.a. die Anzahl der Pässe. Für jeden dieser Pässe liest er dessen Passkonfiguration und startet den Zykluszähler, welcher die Zyklen des Passes abzählt. Dessen Anzahl ist durch die Passkonfiguration gegeben. Wurde das Ende eines Passes erreicht, wird in selber Weise der nächste bearbeitet. Nach dem Ende des letzten Passes wird durch Aktivierung des *FunctionEnd*-Signals das Funktionsende mitgeteilt.

Sämtliche Konfigurationswörter werden in lokalen Registern gespeichert und zusammen mit einem synchronen Resetsignal (*PassReset*) und dem aktuellen Wert des Zykluszählers über den Konfigurationsbus (*PassConfig*)

den funktionalen Einheiten zur Verfügung gestellt. Das synchrone Resetsignal gibt dabei während der Verarbeitung eines Passes die Arbeit der funktionalen Einheiten frei. Nebst *FunctionEnd* werden noch zwei weitere Interfacekontrollsignale definiert; *Busy* und *PassEnd*. Sie zeigen die Beschäftigung bzw. das Ende eines Passes an.

Abbildung 31 zeigt das Zeitdiagramm einer Simulation des Kontrollers, welcher je einmal für Funktion 0 und für Funktion 1 gestartet wurde. Funktion 0 hat dabei 3 Pässe, während Funktion 1 lediglich 2 hat.

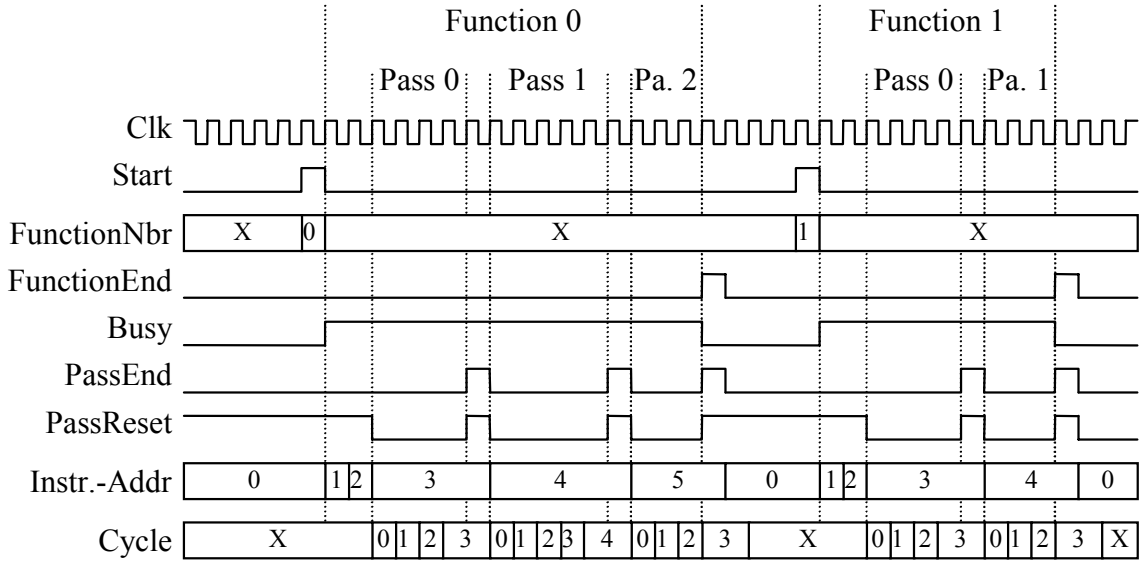


Abbildung 31: Zeitdiagramm des Kontrollers, welcher einmal für die Funktion 0, welche 3 Pässe hat, und einmal für die Funktion 1, welche 2 Pässe hat, gestartet wird.

Zur besseren architektonischen Strukturierung wurde der Kontroller bei dessen Implementierung in 3 hierarchisch organisierte Kontrolleinheiten aufgeteilt, wie in Abbildung 32 gezeigt wird. Bezüglich der ursprünglichen Verhaltensbeschreibung des Kontrollers (Anhang I, Seite P) findet man sämtliche Funktionen und Aufgaben, ausser der innersten Schleife (Zykluszähler) in dem sogenannten Funktionskontroller. Die innerste Schleife wurde in 2 verschachtelte Schleifen aufgeteilt, welche aus architektonischer Sicht bei der Implementierung ebenfalls 2 verschiedene Kontrolleinheiten bilden, nämlich den Passkontroller und den Operationskontroller.

Der Funktionskontroller hat Schnittstellen zur externen Welt und zum Makrocodespeicher. Er enthält hauptsächlich eine Zustandsmaschine und Register zur temporären Speicherung der verschiedenen Konfigurationswerte, welche er den ihm untergeordneten Kontrollern und den funktionalen Einheiten via dem Funktionskonfigurations-Bus zur Verfügung stellt. Die Zustandsmaschine kontrolliert die Zugriffe auf den Makrocodespeicher wie es in der VHDL-Verhaltensbeschreibung der Kontrolleinheit im Anhang I auf Seite P beschrieben ist und zählt die Pässe.

Bei Beginn jedes Passes startet er nach dem Lesen der Passkonfiguration den Passkontroller und wartet dann, bis ihm dieser das Ende des Passes mitteilt.

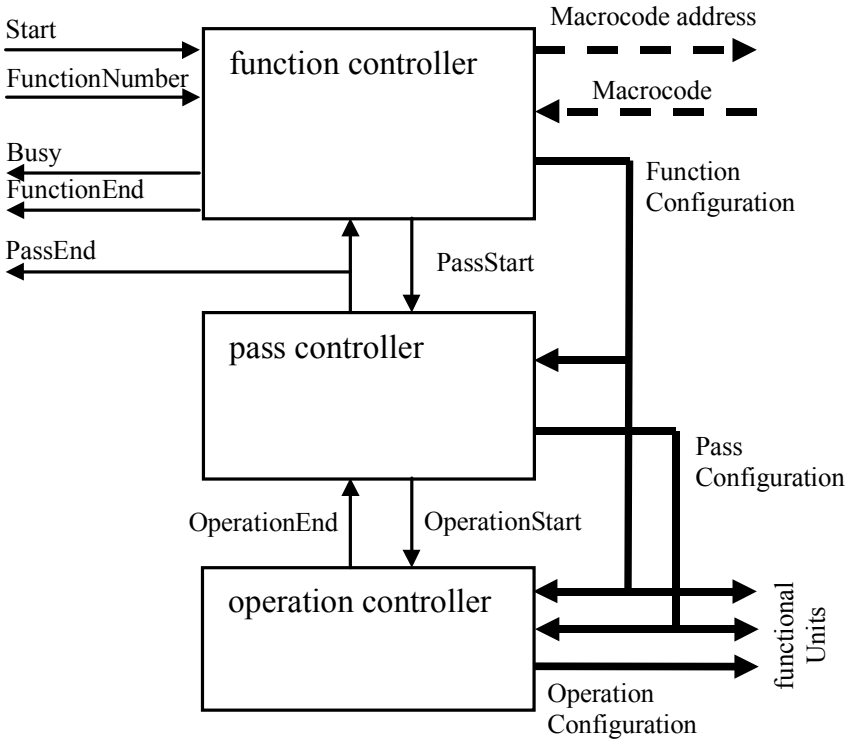


Abbildung 32: Aufteilung des Kontrollers in einen Funktions-, einen Pass- und einen Operationskontroller

Der Passkontroller enthält hauptsächlich einen Operationszähler, welcher die Operationen eines Passes abzählt. Er wird mittels Aktivierung des *PassStart*-Signals vom Funktionskontroller gestartet und meldet diesem das Passende mit dem Signal *PassEnd* zurück. Bei jeder Operation startet er den Operationskontroller und wartet bis dieser die Verarbeitung der Operation zurückmeldet.

Der Operationskontroller schlussendlich enthält einen (oder auch mehrere) Zähler, welcher die operationsinternen Verarbeitungszyklen abzählt. Bei sehr einfachen Grundoperationen kann dies lediglich einen Zyklus dauern, während komplexere Operationen wie zum Beispiel die Radix-4-Operation eine grössere Anzahl von Arbeitszyklen benötigen.

Die Funktionskonfiguration, definiert durch den Funktionskontroller, wird zusammen mit den Zählerwerten des Pass- und Operationskontrollers den funktionalen Einheiten zur Verfügung gestellt. Damit diese den Verarbeitungszustand einer Operation in einer für sie einfacher zu entschlüsselnden Art erhalten, kann der Operationskontroller auch mehrere Operationszähler besitzen, welche zu verschiedenen Zeitpunkten gestartet werden und dadurch zueinander einen gewissen Offset haben.

Die Konfigurationswörter, welche von den 3 Kontrollern zur Verfügung gestellten werden, besitzen zu jedem Zeitpunkt sämtliche notwendige Informationen, um alle Aufgaben durchzuführen, welche für den Zeitpunkt vorgesehen sind. Diese Aufgaben sind zum Beispiel die arithmetische Datenverarbeitung in dem Datenpfad, aber auch die Adressierung der Koeffizienten, Datenquellen und Datendestinationen.

Die funktionalen Einheiten

Da die Konfigurationswörter genügen, um sämtliche Aufgaben präzise zu definieren, kann gesagt werden, dass keine Kontrollabhängigkeit zwischen diesen Aufgaben besteht. Die Schlussfolgerung ist, dass sämtliche Aufgaben autonom durchgeführt werden können.

Diese Tatsache führt dazu, dass die Gesamtaufgabe in Teilaufgaben aufgeteilt werden können, welche von unabhängigen Funktionsblöcken, den sogenannten funktionalen Einheiten, bearbeitet werden.

Der Datenpfad und die Adressgeneratoren sind die üblichsten Beispiele dieser funktionalen Einheiten. Weitere Beispiele sind Interfaceeinheiten, Zirkularspeicherkontroller, Einheiten für *Block Floating Point* - Arithmetik, usw.

Der Datenpfad

Basiert die Architektur des Prozessors selbst nicht auf einer der klassischen Parallelarchitekturen, so wird beim Datenpfad auf eine VLIW-Architektur-ähnliche Struktur zurückgegriffen, wobei die Instruktionen nicht in einem Programmspeicher gelesen werden müssen, sondern von den Kontrolleinheiten Taktzyklus für Taktzyklus geliefert werden.

Der Datenpfad besitzt einen Dekodier- und Kontrollteil und den eigentlichen Datenverarbeitungsteil, welcher den meisten Raum beansprucht.

Je nach Leistungsanforderung der Zielanwendung kann der Datenverarbeitungsteil eine sehr einfache Struktur haben und zum Beispiel aus lediglich einem Multiplikator und einem Adder/Subtraktor bestehen, oder aber eine grössere Anzahl von arithmetischen Einheiten und lokale Register besitzen. Die in Abbildung 33 schematisch dargestellte Struktur bringt bei der Verarbeitung von FFTs verschiedene vorteilhafte Eigenschaften mit sich. Die Organisation, welche in einer ersten Verarbeitungsreihe Multiplikatoren und in 2 darauf folgenden Verarbeitungsreihen Addern/Subtaktoren hat, eignet sich besonders gut für die Bearbeitung von Radix-4-Operationen (siehe Abbildung 29). Je nach der gewünschten Verarbeitungsgeschwindigkeit und den Möglichkeiten eines parallelen Datenzugriffes, kann dabei eine Verarbeitungsreihe eine oder mehrere Multiplikatoren bzw. Adder/Subtraktoren besitzen.

Eingangsregister, welche eine lokale Speicherung der zu lesenden Daten erlauben, vermeiden einen mehrmaligen Speicherzugriff auf ein mehrmals verwendetes Datenwort. Zwischen- und Ausgangsregister ermöglichen eine Aufteilung der Verarbeitungs- und Speicherzeit auf verschiedene Taktzyklen (Pipeline), was bei tiefen Speisespannungen oder einer hohen Taktrate wichtig ist.

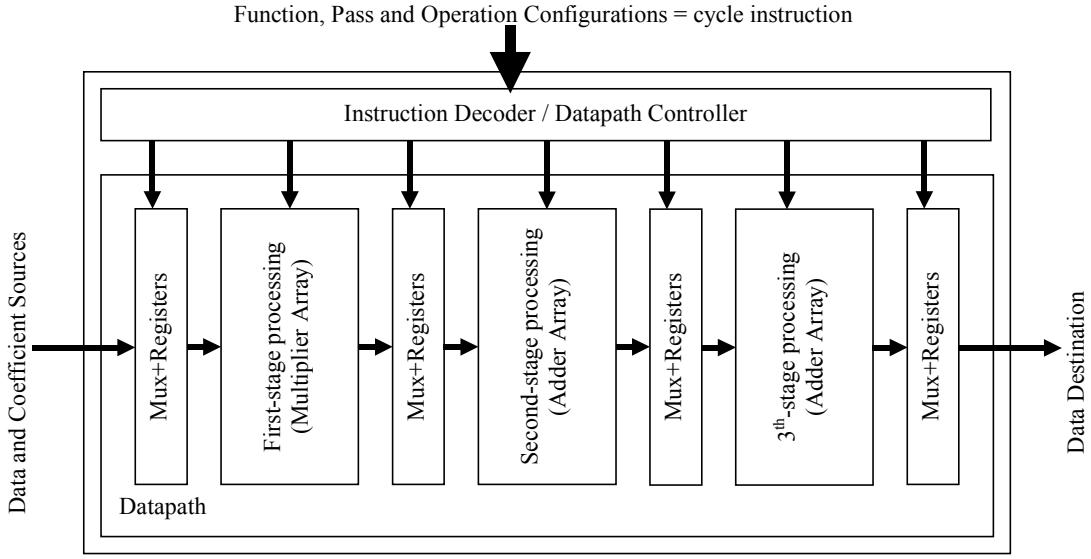


Abbildung 33: Der Datenpfad mit Kontrollteil und Datenverarbeitungsteil

Der Dekodier- und Kontrollteil, welcher sehr einfach gehalten werden kann, erzeugt in Funktion des eingehenden Konfigurationswortes, welches von den 3 Kontrolleinheiten definiert wird und als Zyklusinstruktion betrachtet werden kann, die zur Steuerung der Register, Multiplexer und arithmetischen Einheiten notwendigen Kontrollsignale. Da das Konfigurationswort einerseits die eigentliche Konfiguration, andererseits die Werte von Pass-, Operations- und Zykluszähler enthält, ist dazu lediglich eine kombinatorische Logik notwendig.

Adressgeneratoren

Ebenfalls die Adressgeneratoren sind als funktionale Einheiten konzipiert. Wie der Datenpfad teilen sie das eingehende Konfigurationswort in ein Kontrollsignal und in die Zählerbestandteile auf. Letztere werden von Funktionsblöcken über verschiedene Stufen bis zur eigentlichen Adresse modifiziert. Das Kontrollsignal definiert dabei die Funktionen, welche von den Funktionsblöcken angewandt werden müssen.

Je nachdem, für welche Daten oder Koeffizienten der Adressgenerator die Adressen erzeugen muss, werden unterschiedliche Funktionsblöcke benötigt. Eine typische Anordnung ist in Abbildung 34 gezeigt: Als erstes werden aus den verschiedenen Zählerwerten die benötigten Bits selektiert, welche zusammen einen Initialwert bilden. Dieser wird durch eine *Bit*

Reversing Function modifiziert. Je nach der zu bearbeitenden Datenmenge wird nicht der gesamte Adressierungsbereich gebraucht. Der unbenutzte Bereich kann durch Maskierung der unbenutzten Adressbits eingeschränkt werden. In einem letzten Schritt kann die gefundene Adresse mit einem Offset kombiniert werden.

Der letzte Schritt kann in verschiedenen Fällen von Interesse sein. Er kann zum Beispiel ein Spiegeln der ersten Hälfte einer symmetrischen Fensterfunktion ermöglichen. Eine kombinierte Sinus- und Kosinustabelle kann auch auf einen 8-tel des Gesamtbereiches von 360° reduziert werden, da die Sinus- und Kosinuswerte im Laufe einer 360°-Drehung verschiedene Male entweder die selben Werte oder die negativen Werte annehmen. Und schliesslich ermöglicht dieser Schritt ebenfalls eine Implementierung eines Zirkularspeichers zur Emulation eines FIFOs.

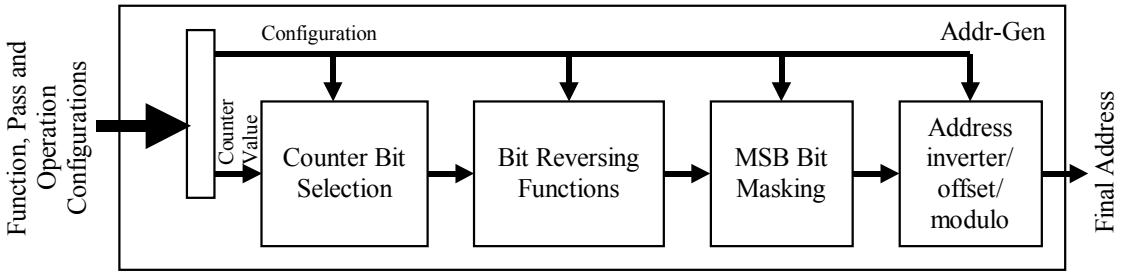


Abbildung 34: Ein Adressgenerator teilt das Konfigurationswort in Konfigurationsteil und Zählerteil auf. Letzteres wird von verschiedenen sequentiell angeordneten Funktionsblöcken bis zum Erhalt der gewünschte Adresse modifiziert.

5.4.3 Die endgültige DSP-Architektur

Da der Funktionskontroller einerseits lediglich zwischen den Pässen auf den Makrocodespeicher zugreift, andererseits nur während der Verarbeitung eines Passes auf Daten- und Koeffizientenspeicher zugegriffen werden muss, kann der Makrocodespeicher mit einem Daten- oder Koeffizientenspeicher zusammengelegt werden ohne dass dabei ein Nachteil entsteht. Der Koeffizientenspeicher eignet sich dafür besonders gut, da er wie der Makrocodespeicher vom DSP lediglich gelesen wird und daher vom selben Speichertyp sein kann.

Die Optimierung des Makrocodes hat zur Folge, dass die benötigte Grösse des Makrocodespeichers in einen Bereich fällt, welcher bei einer ASIC-Integrationen ineffizient ist. Der Grund dazu ist, dass Stromverbrauch und Grösse nicht linear abhängig sind von der Anzahl der Speicherzellen, sondern durch einen Offsetwert beeinflusst sind.

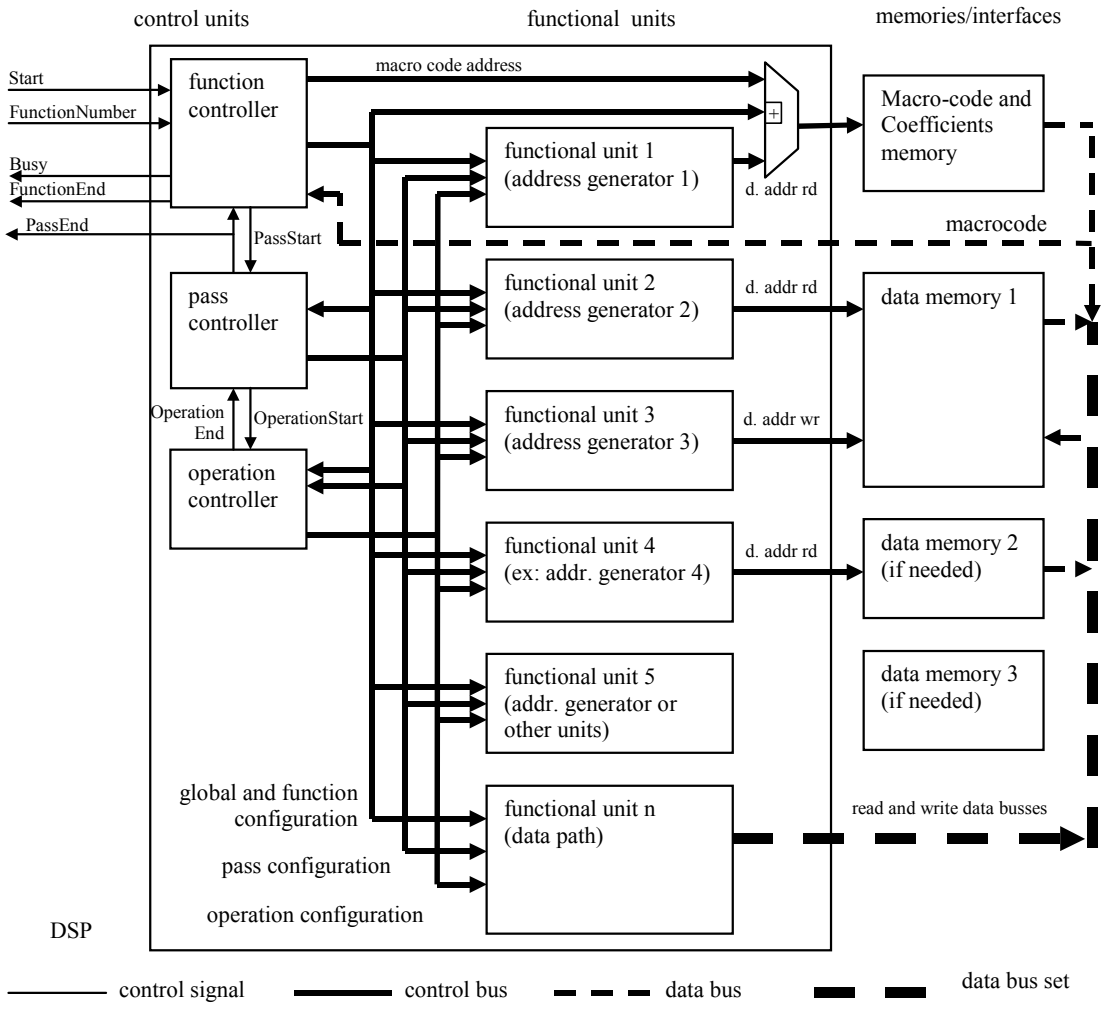


Abbildung 35: Endgültige Struktur der DSP-Architektur

Die Kombination von Koeffizienten- und Makrocodespeicher verringert die Anzahl der notwendigen Speicher und erlaubt den Gebrauch von Speicher mit besser ausgewogenen Grössen. Koeffizienten und Makrocode werden unterschiedlichen Bereichen des Speichers zugeteilt, wobei die Platzierung des Makrocodes an den Speicherbeginn folgende interessante Möglichkeit zulässt: In diesem Fall kann der Speicherbereich, in dem die für einen Pass notwendigen Koeffizienten abgelegt sind, als Parameter im Makrocode enthalten sein und als Offset zu der vom entsprechenden Adressgenerator erzeugen Adresse dazuaddiert werden. Dies erlaubt die Verwendung von verschiedenen Koeffiziententabellen innerhalb eines Algorithmus.

Die endgültige Struktur der DSP-Architektur ist in Abbildung 35 abgebildet.

Klassifizierung der DSP-Architektur

Die DSP-Architektur ist eine Spezial-Architektur mit einer für regelmässige Algorithmen sehr effizienten Instruktionscodierung. Der Programmcode, Makrocode genannt, entspricht aus hierarchischer Sicht einer ersten Programmebene und kodiert in einem einzigen Instruktionswort eine ganze

Gruppe von komplexen Operationen. Eine solche Operationsgruppe wird Pass genannt.

Während der Bearbeitung eines Passes generieren der Funktions-, Pass- und Operationskontroller verschiedene Konfigurationswörter, welche zusammengesetzt jeden Taktzyklus ein neues Instruktionswort ergeben. Diese so generierten Instruktionen entsprechen einer zweiten Programmebene tieferen Niveaus. Man könnte sie Mikroinstruktionen nennen.

Diese Mikroinstruktionen kontrollieren die funktionalen Einheiten. Führten all diese Einheiten jeweils die selbe Operation durch, so könnte man von einer makroprogrammierbaren SIMD-Architektur sprechen. Dies ist jedoch nicht der Fall.

Andererseits können die funktionalen Einheiten, und im Besonderen der Datenpfad, als VLIW-Architektur-basierte Prozessoreinheiten betrachtet werden, denn die Mikroinstruktionen sind sehr breit und steuern zum Beispiel innerhalb des Datenpfades zur selben Zeit verschiedene Ressourceneinheiten.

Low-Power-Techniken

Folgende Punkte minimieren die Leistungsaufnahme des Prozessors:

- Die Reduzierung der notwendigen Grössen für Daten-, Koeffizient- und Programmspeicher (siehe Seite 109) senkt gleichermassen $P_{leakage}$ und $C_{Speicher}$ (siehe Seite 17).
- Die Reduzierung der Anzahl der notwendigen Zugriffe auf den Daten-, Koeffizienten- und Programmspeicher beeinflusst unter anderem positiv $a_{Speicher}$.
- Die „unproduktiven“ Kontrollzyklen sind minimiert (Optimierung von CPO)
- Ein paralleler Datenpfad mit interner Pipeline erlaubt die Verwendung einer tiefen Speisespannung (Optimierung von Vdd).
- Die Grundarchitektur erlaubt die Verwendung verschiedenster, anwendungsspezifischen Low-Power-Techniken wie:
  - Eine Block-Gleitarithmetik erlaubt eine Reduzierung der Datenbusweiten (Senkung der Leistungsaufnahme durch Reduktion der Designkomplexität)
  - *Gated Clocks* (Reduktion von a_k)
  - Optimierter Datenpfad mit geeigneten arithmetischen Einheiten und kleinen, lokalen Datenbussen (Optimierung von a_k, C_k)

5.5 Anwendungsbeispiel der makroprogrammierbaren DSP-Architektur

Die Implementierung eines WOLA[1]-Prozessors für Hörgeräte

Die makroprogrammierbare DSP-Architektur hatte die idealen Voraussetzungen für die Realisation eines WOLA-Prozessors, welcher in einem digitalen Datenverarbeitungsteil für Hörgeräte benötigt wird (siehe Kapitel 6). Sie ist höchst effizient bezüglich Platzbedarf, Datendurchsatz (Rechenleistung), Verlustleistung und Flexibilität. Dies sind exakt die Eigenschaften, welche für eine erfolgreiche Implementierung der WOLA-Filterbank für digitale Hörgeräte benötigt werden [3][4][5].

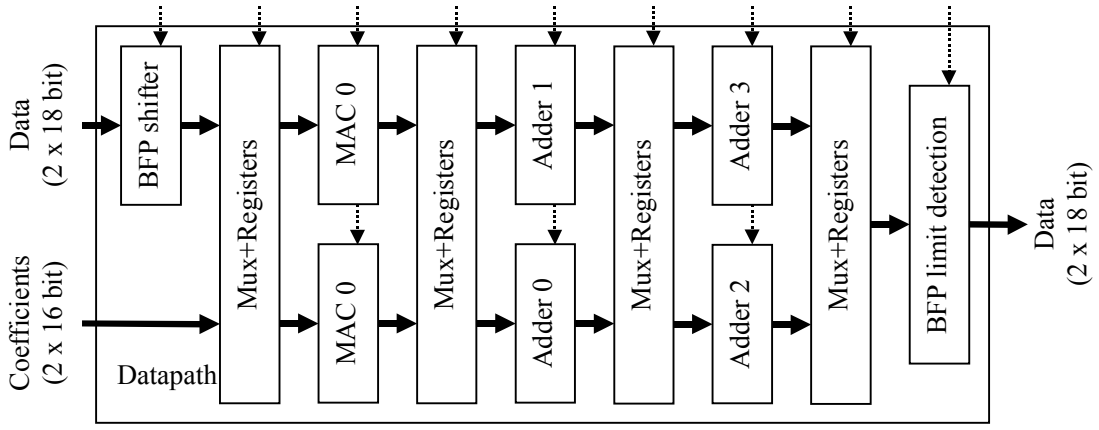


Abbildung 36: Die ALU des WOLA-Prozessors

Eine hohe Rechenleistung bei einer tiefen Taktrate ist eine an den WOLA-Prozessor gestellte Bedingung. Dies implizierte die Verwendung von parallel arbeitenden arithmetischen Einheiten. Der in Abbildung 33 vorgestellte Datenpfad wurde wegen der für die Verarbeitung des WOLA-Algorithmen idealen Grundstruktur für die Implementierung des WOLA-Prozessors gewählt. Die in jeder Verarbeitungsreihe zweifach instanziierten arithmetischen Einheiten (2 *MAC*, 2 x 2 *Adder/Subtractor*) erbringen eine Rechenleistung, welche es zum Beispiel ermöglicht, alle 6 Taktzyklen eine Radix-4-Operation durchzuführen. Wenn man bedenkt, dass eine Radix-4-Operation 12 reelle Multiplikationen und 22 reelle Additionen benötigt, erkennt man die hohe Effizienz der gewählten ALU.

Der hohe Datendurchsatz bedingt auch eine hohe Datenübertragungsrate von und zu den Zwischenspeichern. Die Busse für Daten und Koeffizienten, aber auch die Datenspeicher wurden deshalb immer doppelt angelegt. Im selben Taktzyklus können auf diese Weise 2 Datenwerte und 2 Koeffizienten gelesen und 2 Datenwerte zurückgeschrieben werden.

---

[1] WOLA: *Weighted OverLap Add*. Informationen bezüglich dem WOLA-Algorithmus und dem WOLA–Prozessor sind im folgenden Kapitel zu finden.

Für die Implementierung des WOLA-Prozessors mussten nur wenige Anpassungen der in der Abbildung 35 vorgeschlagenen DSP-Architektur vorgenommen werden. Die Funktionsweise des Kontrollteils blieb erhalten.

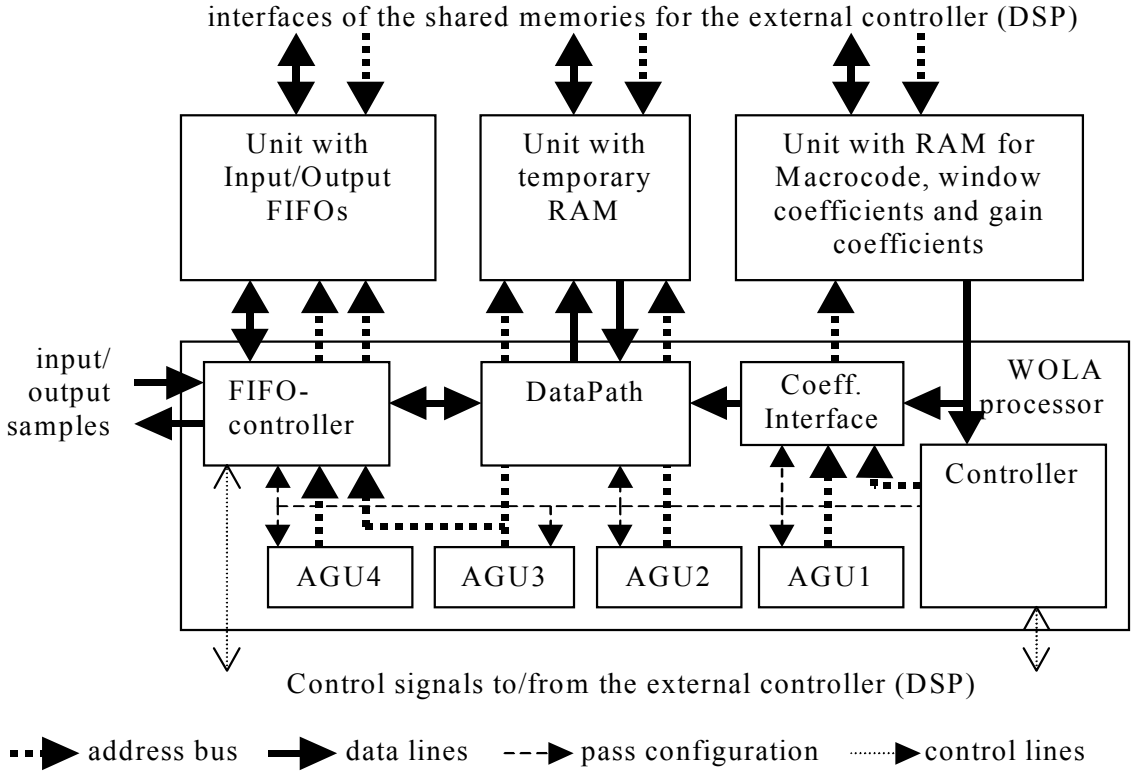


Abbildung 37: Die Struktur des WOLA-Prozessors

Um WOLA-Algorithmen verarbeiten zu können, wurden der Temporärspeicher und der Makrocode- und Koeffizientenspeicher um die beiden FIFOs ergänzt. Um die hohen Datentransferraten von und zu der ALU gewährleisten zu können, wurde jeder Speicher aufgeteilt in zwei Speicher, welche die halbe, ursprüngliche Grösse besitzen und auf die gleichzeitig zugegriffen werden kann. Gemäss der maximal existierenden Anzahl von miteinander gebrauchten Lese- und Schreibvektoren innerhalb des WOLA-Algorithmus (dies befindet sich im letzten Pass der Synthese [siehe Abbildung 43, Kapitel 6]) wurden 4 Adressgeneratoren benötigt, welche alle Lese- und Schreibadressen erzeugen.

Für die beiden FIFOs werden je nach Anwendung *Two-Port-* oder *Dual-Port-RAM* verwendet, auf die mit einem vom FIFO-Kontroller definierten Zeiger in zirkularer Weise zugegriffen werden kann. Der Adressbereich des Eingangs-FIFO ist in 3 Bereiche aufgeteilt. Im ersten Bereich liegen die Daten für die aktuelle WOLA-Transformation, im zweiten Bereich die neuen Samples, welche während der nächsten Transformation zum ersten mal bearbeitet werden und je nach Konfiguration in einen unbenutzten Teil. Das Ausgangs-FIFO ist in ähnlicher Weise in 3 Bereiche eingeteilt (Abbildung 38). Von Transformation zu Transformation werden diese

Bereiche um die Eingangsschrittweite R verschoben, wobei der FIFO-Kontroller die Positionen der Bereiche definiert (siehe auch Flussdiagramm der WOLA-Transformation, Abbildung 43). Wird für eine WOLA-Transformation oder -Rücktransformation auf die Daten der FIFOs zugegriffen, erzeugen zwei Adresskontroller (*AGU4* und *AGU3*) lediglich die relativen Adressen bezüglich dem Blockbeginn der Daten. Der FIFO-Kontroller kombiniert diese mit der Offsetposition der Bereiche. Wie die Adressgeneratoren ist dabei auch der FIFO-Kontroller als funktionale Einheit konzipiert.

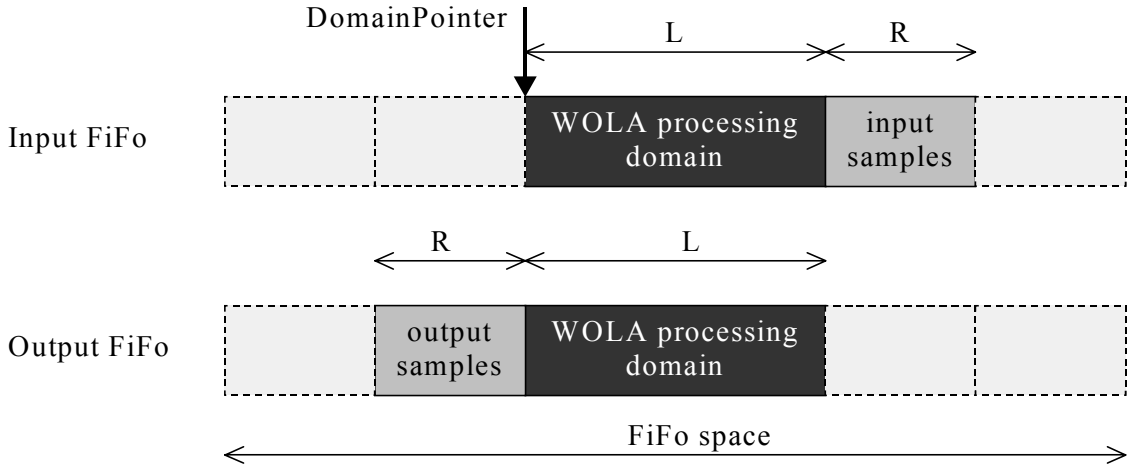


Abbildung 38: Die Bereichsaufteilung der beiden FIFOs

Eine letzte Anpassung betrifft die Schnittstelle, welche den Zugriff auf die Konstanten und Koeffizienten vereinfacht. Dieser Block enthält einerseits arithmetische Operatoren, welche das Kombinieren der von *AGU1* erzeugten Adresse mit einem Offsetwert erlaubt. Dies ist bei Funktionen mit einer periodischen Repetition (Sinus, Kosinus, usw.) oder bei symmetrischen Funktionen von Interesse, da auf diese Weise nur ein Intervallteil der Funktion gespeichert werden muss. Da die FFT und gewisse andere Pässe der WOLA-Transformation trigonometrische Operationen durchführen, enthält der Block andererseits auch direkt eine Sinus- und Kosinustabelle für die ersten 45 Grad, welche mit der vorhin angesprochenen Offsetkombination auf 360 Grad erweitert wird.

Block-Gleitkommaarithmetik (Block floating point arithmetic)

Um an Dynamik zu gewinnen und den Quantisierungsfehler zu reduzieren, wurde dem Datenformat innerhalb des Datenpfades besondere Aufmerksamkeit gewidmet. Die Radix-2-, Radix-4- und MAC-Operationen haben ein generelles Wachsen der Datenamplituden während der WOLA-Transformation zur Folge. Ohne Gleitkommaarithmetik sind Bereichsüberläufe (*Overflow*) der Daten nicht zu verhindern wenn sie nicht zu Beginn durch einen grossen Wert dividiert werden. Diese Vordivision hat

jedoch einen grossen Quantisierungsfehler zur Folge, was nicht wünschenswert ist. Eine Gleitkommaarithmetik andererseits verlangt nach zusätzlichen Ressourcen, da einerseits die arithmetischen Einheiten komplizierter sind als diejenigen für Festkommaarithmetik, andererseits der zusätzliche Exponent einer Gleitkommazahl breitere Busse, Multiplexer, Register und Speicher verlangt. Dies bedeutet eine grössere Fläche und eine höhere Verlustleistung, was ebenfalls nicht gewünscht wird.

Eine sehr interessante Alternative zu der Fest- und Gleitkommaarithmetik ist die sogenannte Block-Gleitkommaarithmetik (*Block floating point arithmetic, BFP*). Die Daten werden dazu in nicht-überlappende Gruppen eingeteilt und erhalten einen gemeinsamen Exponenten [6].

Die hohe Regularität der FFT- und WOLA-Transformation bringt ideale Voraussetzungen mit sich für eine einfache Implementierung einer BFP-Arithmetik. Bei der Entwicklung der makroprogrammierbaren DSP-Architektur wurden die FFT-Algorithmen bereits in Funktionen, Pässe usw. eingeteilt. Per Zufall bilden die Daten, welche während eines solchen Passes bearbeitet werden eine ideale Gruppe für die Zuweisung eines gemeinsamen Exponenten.

Die BFP-Arithmetik funktioniert wie folgt: Ein Register enthält den Exponenten, welcher zu Beginn der WOLA-Transformation auf Null gesetzt wird. Pass für Pass werden dann die Daten verarbeitet. Während jedem Pass wird nach dem grössten produzierten Wert gefahndet[1], und wenn diese am Ende des Passes eine gewisse Grenze überschritten hat, wird der gemeinsame Exponent erhöht und während des nächsten Passes sämtliche Daten beim Lesen um den erhöhten Wert nach rechts geschoben[2].

Die BFP-Arithmetik ermöglicht die Verwendung einer simplen 16-Bit-Arithmetik, erhöht jedoch den Dynamikbereich und den SNR der WOLA-Filterbank in ähnlichem Mass wie eine Gleitkommaarithmetik.

Resultate

In diesem Abschnitt sollen lediglich die Grössen der einzelnen Unterblöcke und die Verarbeitungsgeschwindigkeit der verschiedenen WOLA-Konfigurationen näher betrachtet werden. Weitere, Fertigungsprozess-abhängige Resultate wie Stromverbrauch und maximale Taktrate werden in Kapitel 7 präsentiert.

Tabelle 8 zeigt die Grösse einiger Untereinheiten des FFT-Prozessors. Speziell soll dabei die geringe Grösse der Kontrolleinheiten und der

[1] Richtigerweise: Es wird nach dem Wert gefahndet, zu dessen Repräsentation am meisten Bit benötigt werden.

[2] Ein bitweises Schieben nach rechts um ein Bit bedeutet eine Division durch 2

Adressgeneratoren hervorgehoben werden: Dank konsequentem Erarbeiten von Lösungskonzepten zu jeder Entwicklungsphase konnte der „unproduktive“ Designanteil, also die Einheiten, welche nicht direkt zur Datenverarbeitung benötigt werden, minimiert werden, ohne dabei an Flexibilität des Kontrollsystems einbüssen zu müssen.

Name des Blocks	*Grösse (GE)*	*Kommentar*
Kontroller	2045	Alle Kontroller zusammen (davon 700 GE für Konfigurationsregister)
Adressgeneratoren	1301	Alle Adressgeneratoren zusammen
Coeff. Interface	1269	Mit integrierter Sinus- Kosinustabelle
SinCos-LUT	264	LUT enthaltende Sinus- Kosinustabelle
FIFO-Kontroller	646	
Datenpfad	12606	Inklusive Datenpfad-Kontroller, Register, arith. und BFP-Unit
MAC	2*2668	Ohne Eingangsregister, mit Sättigungs- und Rundungseinheit
Addierer	2*227+ 2*117	Ohne E-Register, 2 Add. mit Sättigungs- und Rundungseinheit
Total	**18753**	Alle vorausgehenden Hauptblöcke + Glue-Logic

Tabelle 8: Grösse in GE des WOLA-Prozessors (Referenztechnologie: Alcatel-Mietec 0.35 um CMOS)

N	**L**	**O F**	**D F**	**M**	**Anzahl Taktzyklen (Analysis + Synthesis)**	**$t_{benötigt}/t_{zur\ Verfügung}$ (Analysis + Synthesis)**
16	128	2	2	no	192	38%
32	256	2	2	no	258	26%
32	128	2	2	yes	328	33%
32	128	2	2	no	289	29%
128	256	4	4	no	788	39%

Tabelle 9: Berechnungszyklen und –zeit für verschiedene WOLA-Konfigurationen. N: FFT-Grösse (komplex), L: Fenstergrösse, OF: Oversampling Factor (R=N/OF), DF: Decimation Factor, M: Modulation (*odd staking Modus*, *siehe Abbildung 55*).

Die Zeit, welche zur Verarbeitung eines WOLA-Algorithmus benötigt wird, ist stark konfigurationsabhängig. Tabelle 9 zeigt die Zeiten einiger Konfigurationen und gibt ebenfalls ein Verhältnis zwischen der für die Verarbeitung benötigten Zeit und die zur Verfügung stehende Zeit. Dabei wird angenommen, dass die Taktrate des Prozessors 1 MHz ist und die Abtastfrequenz 16 kHz beträgt.

5.6 Schlussfolgerung

Das Makrocode-Konzept erlaubt, bei einer Implementierung so von einer gut organisierten Algorithmusstruktur zu profitieren, dass die Verarbeitungsgeschwindigkeit erhöht und die Komplexität, d.h. die Grösse der Hardware, reduziert wird. Die Operationen eines Algorithmus werden dazu zu Makro-Operationen gruppiert, welche an einem Stück verarbeitet werden.

Das Konzept ist auf einem solch globalen Niveau präsentiert, dass es sich für die meisten FFT-basierten Filterbank-Algorithmen verwenden lässt, unabhängig von der Algorithmuskonfiguration und der benötigten Rechenleistung. Das Kriterium, ob sich ein Algorithmus mit Hilfe des Makrocode-Konzepts implementieren lässt, hängt von der Regelmässigkeit der Algorithmusstruktur ab. Unregelmässigkeiten erschweren oder verunmöglichen die Verwendung des Konzeptes. Und hier ist sicherlich die grösste Schwäche des vorgestellten Lösungsansatzes. Denn häufig ist ein Algorithmus nur „beinahe“ ideal strukturiert. Der restliche, unregelmässige Teil verhindert aber, dass das Makrocode-Konzept verwendet werden kann.

In dem in den nächsten beiden Kapiteln vorgestellten Anwendungsbeispiel, wo u. a. auch das Makrocode-Konzept verwendet wird, wird dieses Problem gelöst, indem zusätzlich zum makroprogrammierbaren Prozessor noch ein *General Purpose DSP* verwendet wird. Der makroprogrammierbare Prozessor verarbeitet dabei alle regelmässigen Algorithmusteile und der *General Purpose DSP* die unregelmässigen Teile. Da der Geschwindigkeitsgewinn der Datenverarbeitung auf Kosten der Ressourcenauslastung geht, ist dies wohl keine optimale Lösung, denn meist arbeitet nur einer der beiden Prozessoren, während sich der andere in einem Wartezustand befindet.

Damit die Ressourcenauslastung nicht leidet, sollte nicht nach einer Lösung mit zwei individuellen Prozessoren gesucht werden, sondern die beiden Prozessoren müssen kombiniert werden. Die makroprogrammierbare DSP-Architektur muss die Fähigkeit bekommen, auch individuelle Operationen auf dem Datenpfad auszuführen, und die dazu notwendigen Parameter einzeln zu adressieren. Wohlverstanden, diese Operationen sollen nicht nur eine einzige arithmetische Einheit des Datenpfades kontrollieren, sondern von der Mehrfachexistenz solcher Einheiten profitieren. Je nach der zu erzielenden Flexibilität muss die makroprogrammierbare DSP-Architektur durch verschiedenste Einheiten ergänzt werden, wie z.B. einer logischen Einheit, einem Programmkontroller und einem Stack für den Programmzähler, usw. Eine Möglichkeit, wie die makroprogrammierbare DSP-Architektur modifiziert werden muss, um auch individuelle Operationen verarbeiten zu können, zeigen die folgenden Zeilen.

Ausgehend vom bestehenden Makrocode-Konzept geschieht die grösste Modifikation im Funktionskontroller. Dieser arbeitet bekanntlich wie folgt: Eine Zustandsmaschine lädt vom Makrocodespeicher die verschiedenen Konfigurationswörter in lokale (Konfigurations-) Register und startet für jeden Pass einer Funktion den Passkontroller. Die in den Registern gespeicherten Konfigurationswerte besitzen sämtliche Informationen, damit die verschiedenen funktionalen Einheiten den gesamten Pass verarbeiten können. Im neuen Prozessor wird die Zustandsmaschine durch einen programmierbaren Kontroller ersetzt, dessen Instruktionsset erlaubt, sämtliche Funktionen der Zustandsmaschine zu ersetzen. Er kann also Daten vom Programmspeicher (ehemaliger Makrocodespeicher) lesen und in die Konfigurationsregister schreiben, den Passkontroller starten, auf ein Passende warten, den Programmfluss steuern, usw. Diese erhaltene Lösung ist bis anhin der ursprünglichen funktionell gesehen noch identisch, ermöglicht jetzt aber auf elegante Weise, die Möglichkeiten des Prozessors zu erhöhen, indem nämlich das Instruktionsset des Kontrollers erweitert wird. Zu dieser Erweiterung gehört eine gute Auswahl von Sprung- und Verzweigungsinstruktionen, welche einen gewissen Komfort in der Steuerung von Programmabläufen garantieren sollen. Die interessanteste Flexibilitätserweiterung wird jedoch mit neuen arithmetischen „Individualoperationen“ (im Gegensatz zu den Makro-Operationen) erzielt, welche die existierenden Ressourcen des Datenpfad neu verwenden.

Im bestehenden Prozessor wird die Verarbeitung einer Operation vom Operationskontroller gesteuert, welcher wiederum vom Passkontroller für jede Operation eines Passes (oder Makros) aktiviert wird. Der Passkontroller wird bei Individualinstruktionen nicht mehr benötigt, da diese jeweils nur eine Operation enthalten. Bei arithmetischen Individualinstruktionen kann deshalb der Passkontroller „überbrückt“ und der Operationskontroller direkt von der Programminstruktion kontrolliert werden. Eine angemessene Erweiterung des Operationskontrollers, damit auch für Individualinstruktionen eine effiziente Menge an arithmetischen Operationen zur Verfügung steht, beendet schlussendlich die Modifikationen des Prozessors. Die Operationenmenge soll dabei Operationen beinhalten, welche eine arithmetische Einheit des Datenpfades verwenden, aber auch mehrere.

Referenzen

[1] Bevan M. Baas, Department of Electrical Engineering, Ultra Low Power Technology Group, Stanford University, http://nova.stanford.edu/~bbaas/fftinfo.html

[2] A. Drollinger, A. Heubi, P. Balsiger, F. Pellandini, „Macro-Programmable DSP Architecture for Parallel / Pipelined Data Path

Units, Targeted for FFT Based Algorithms“, ICSPAT 2000, International Conference on Signal Processing Applications & Technology, Dallas, Texas, USA, October 16-19, 2000.

[3] A. Drollinger, C. Wälchli, D. Sun, L. Grisoni, C. Calame, A. Heubi, P. Balsiger, F. Pellandini, R. Brennan, T. Schneider, "An Ultra Low Power WOLA Filterbank Implementation in Deep Submicron Technology", Proc. of COST 254, International Workshop on Intelligent Communication Technologies and Applications, with Emphasis on Mobile Communication, Neuchâtel, CH, May, 5-7, 1999.

[4] A. Drollinger, A. Heubi, P. Balsiger, F. Pellandini, "Low Voltage FFT Co-Processor for Ultra Low Power, Real-Time Applications", Proc. of the DSP Deutschland 99, Forum der Technik, pp. 357-362, Muenchen, D, September 21-23, 1999.

[5] A. Drollinger, R. Brennan, T. Schneider, Ch. Calame, L. Grisoni, D. Sun, Ch. Waelchli, A. Heubi, P. Balsiger, F. Pellandini, "*An Ultra Low Power WOLA Filterbank Implementation in Deep Submicron Technology*", Proc. of the International Conference on Signal Processing Applications and Technology 99, ICSPAT 99, Orlando, Florida, USA, November 1-4, 1999.

[6] A. Chhabra, R. A. Iyer, “A Discussion of the Block Floating-Point FFT Implementation on Fixed-Point DSPs”, Proc. of the International Conference on Signal Processing Applications and Technologyí99, ICSPATí99, Orlando, Florida, USA, November 1-4, 1999.

6 *Ein digitales Hörgerät als Anwendungsbeispiel für die drei Lösungsansätze*

Moderne Entwurfsmethoden für das digitale IC-Design sind für möglichst schnelle Entwurfszeiten entwickelt worden und produzieren ohne grossen Aufwand schnell überzeugende Resultate. Die schnellen Entwurfszeiten gehen jedoch häufig auf Kosten einer Designqualitätseinbusse, welche sicherlich gering sein kann, aber doch zu einem Hindernis wird, wenn dadurch die Machbarkeit einer Anwendung in Frage gestellt wird.
Die Machbarkeitsfrage wurde bei der Entwicklung einer digitalen Verarbeitungseinheit für Hörgeräte gestellt. Die Randbedingungen, denen das Digitalteil unterlag, sind einerseits durch die kleine physikalische Grösse von Hörgeräten, andererseits durch rechenintensive Datenverarbeitungsalgorithmen begründet. Nur wenn anstelle der üblichen Entwurfsmethoden anwendungsspezifische Lösungsansätze verwendet werden, ist es möglich, ein Design zu entwerfen, welches alle Designrandbedingungen erfüllen kann.

Dieses Kapitel erklärt im ersten Teil, was ein digitales, modernes Hörgerät ist und wie es aufgebaut ist. Dann wird detaillierter auf die Anforderungen eingegangen, welche an den Digitalteil gestellt werden und stellt einen ersten Lösungsansatz vor. Einige Hörgeräte-bezogene, theoretische Aspekte der digitalen Datenverarbeitung, schliessen das Kapitel ab.

Das moderne Hörgerät

Die Zeit, wo ein Hörgerät Musik und Gespräch zusammen mit dem Umgebungslärm lediglich banal verstärkte, gehört zur Vergangenheit. Ein modernes Hörgerät ist ein richtiger Hörcomputer, welcher dank digitaler Datenverarbeitung patientenspezifische Korrekturen vornimmt, Umgebungslärm dämpft und Geräuschquellen selektieren kann.

Ein Patient kann auf Grund seines Handicaps und seiner Gewohnheit zwischen verschiedenen Typen von Hörhilfen auswählen.

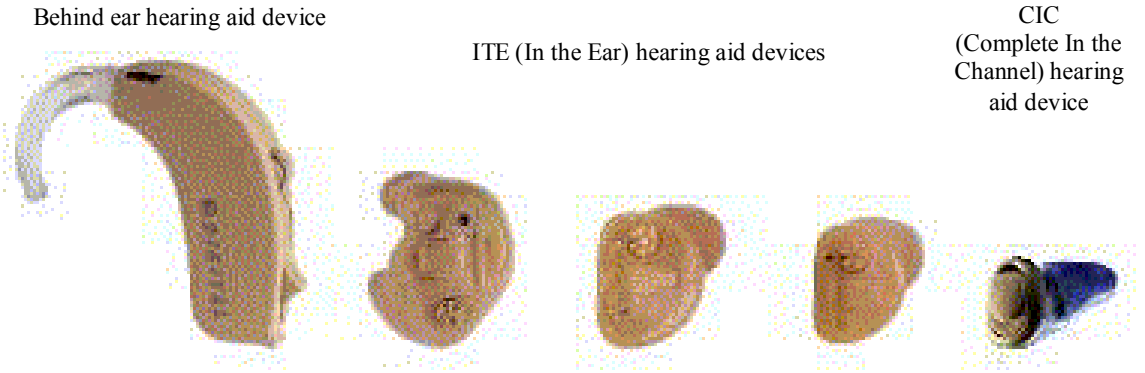


Abbildung 39: Die verschiedenen Typen von Hörgeräten

Das konventionelle Hörgerät wird hinter dem Ohr getragen. Ein dünnes Röhrchen leitet den verstärkten und gefilterten Schall in die Nähe des Trommelfells. Die Grösse dieses Hörgerätentyps vereinfacht seine Bedienung, verhindert jedoch auch ein diskretes Tragen.

Die diskreteste Art von Hörgeräten wird vollständig im Gehörkanal getragen (CIC[1]). Das dabei zur Verfügung stehende Volumen für das gesamte Gerät ist auf etwa 1 cm^3 limitiert, was technisch hohe Einschränkungen zur Folge hat.

Diese Einschränkungen können mit Hörgerätetypen reduziert werden, welche ebenfalls teilweise im Gehörkanal getragen werden, jedoch auch einen Teil besitzen, welcher in der Gehörmuschel zu liegen kommt.

Der prinzipielle Aufbau eines digitalen Hörgerätes unterscheidet sich nicht wesentlich zwischen den verschiedenen Typen. Ein digitales Hörgerät setzt sich aus folgenden Elementen zusammen (siehe auch Abbildung 40):

- Ein oder zwei Mikrophone und ein kleiner Lautsprecher
- AD- und DA-Wandler, welche eine Schnittstelle zwischen dem analogen und digitalen Bereich bilden.

[1] CIC: *Complete In Channel*

- Eine digitale Kontroll- und Datenverarbeitungseinheit
- Ein Festspeicher um patientenspezifische Informationen zu speichern (EEPROM oder Flash-Memory)
- Eine Möglichkeit zur Programmierung und Bedienung des Gerätes. Dies können Schalter und Potentiometer sein, aber auch ein Empfänger für eine Fernbedienung (Infrarot, Ultraschall oder Radiowellen)

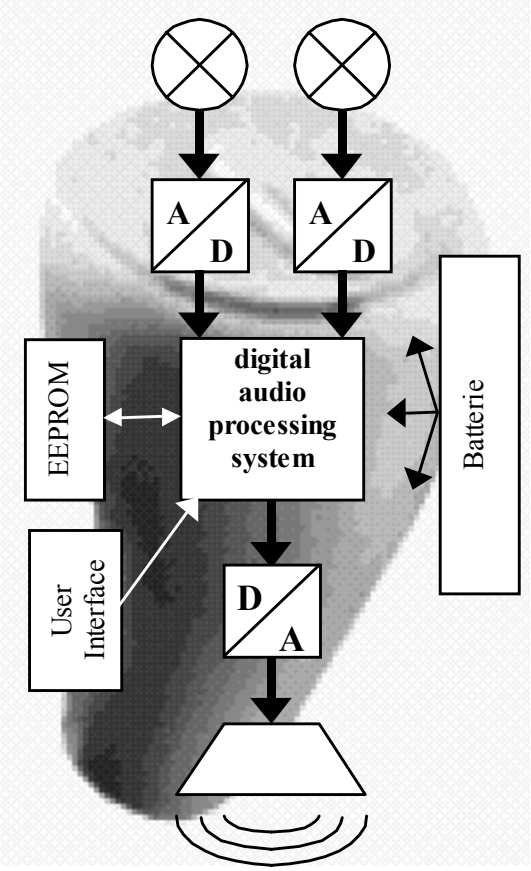


Abbildung 40: Prinzipieller Aufbau eines digitalen Hörgerätes

Die gesamte mikroelektronische Baugruppe, d.h. Wandler, Verstärker, Kontroll- und Datenverarbeitungseinheit, Festspeicher und eventuell gewisse Teile einer Fernbedienung kann auf einem oder mehreren Mikrochips integriert sein. Da Festspeicher, Analogelektronik und Digitalelektronik nicht die selben Anforderungen an eine Halbleitertechnologie stellen, werden diese drei Teile meistens separat auf zwei bis drei Mikrochips integriert, jeweils in einem dafür geeigneten Fertigungsprozess (heutiger Zeitpunkt).

6.1 Entwicklung eines Digitalteils für Hörgeräte

Auch heute noch ist auf Grund der physikalischen Anforderungen die Entwicklung eines Hörgerätes, oder Komponenten davon, technisch gesehen

eine Herausforderung. Die Entwicklung einer neuen, digitalen Datenverarbeitungseinheit für Hörgeräte konkretisierte die Schwierigkeiten, ein modernes Hörgerät zu realisieren. Die in der Entwicklung zu bewältigenden Randbedingungen, welche an den Digitalteil gestellt wurden und einerseits aus der Zielanwendung selbst resultieren, aber auch auf Grund des anspruchsvollen Pflichtenheftes entstanden sind, verlangten nach unkonventionellen Lösungsansätzen in der Designentwicklung. Um eine globale Idee der Entwicklungsschwierigkeiten zu bilden, sind im Folgenden einige wichtige Anforderungen, welche an das Design gestellt wurden, aufgelistet. Die ersten beiden Punkte präzisieren dabei das Einsatzgebiet des Digitalteils, die darauf folgenden Punkte sind eher Konsequenzen daraus.

- Der Einsatzbereich des Digitalchips soll sich nicht nur auf sämtliche Hörgerätetypen beschränken, sondern auch eine Lösung sein für Anwendungen, welche eine der folgenden Funktionen benötigen: Echounterdrückung, dynamische Signalkompression, Reduktion des Umgebungslärmes, Stimmenaktivitätenerkennung, phasenabhängige Filterung, usw.
- Der in Abbildung 41 dargestellte Fluss der Audiodaten soll eine hohe Einsatzflexibilität des Digitalteils garantieren: Verschiedene Arten von Algorithmen für die im letzten Punkt aufgelisteten Anwendungen sollen realisiert werden können, wobei alle diese Algorithmen eine Gemeinsamkeit haben, nämlich dass sie eine FFT- oder WOLA-Filterbank zur Spektralanalyse benutzen, die von der Quelle kommenden Audiodaten in einem Vorfilter um den Faktor 2 dezimieren und die bearbeiteten Audiodaten in einem Nachfilter wiederum um den Faktor 2 interpolieren. Wie aber die Audiodaten zwischen diesen gemeinsamen Verarbeitungsstufen bearbeitet werden, soll dem Anwender überlassen werden. Der Digitalteil soll deshalb auch frei programmiert werden können.
  - ➔ Die Leistungsaufnahme für das gesamte Hörgerät darf 1 mW nicht übersteigen. Für den digitalen Teil bleiben davon etwa 300 uW.
  - ➔ Tiefe Speisespannung: Die Verlustleistung eines CMOS-Schaltungselementes steigt ungefähr quadratisch mit der Speisespannung. Deshalb ist es wichtig, dass der Digitalteil direkt mit der Batteriespannung arbeitet. Beim Gebrauch einer Knopfzelle kann diese auf 0.9 Volt sinken.
  - ➔ Die Grösse des Digitalchips darf 10 mm^2 nicht übersteigen. Dies ist wichtig für den Gebrauch in CIC-Hörgeräten.
  - ➔ Die Schaltgeschwindigkeit von CMOS-Gatter verlangsamt sich mit sinkender Speisespannung. Ein Algorithmus muss noch bei einer Taktrate von 1.0 MHz verarbeitet werden können.
  - ➔ Verarbeitung von Mono- und Stereoquellen.

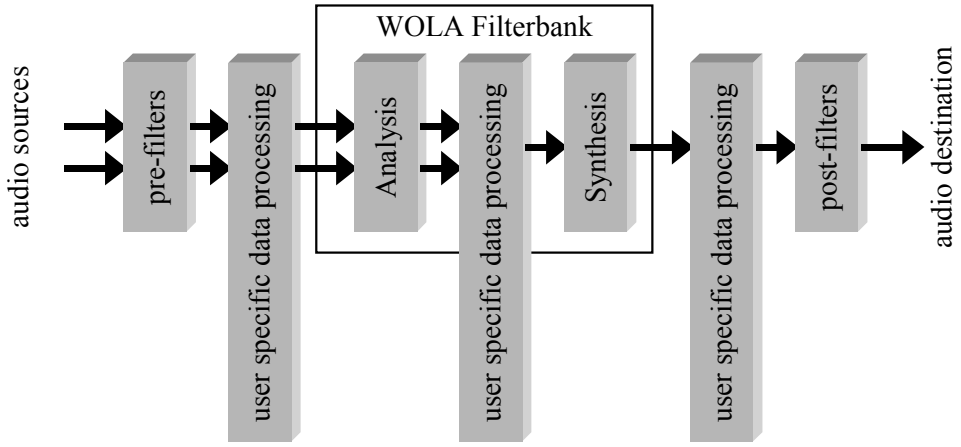


Abbildung 41: Flussdiagramm der Audiodaten innerhalb des Digitalteils

Die hier aufgelisteten Punkte verlangen bei der Implementierung zum Teil nach äusserst widersprüchlichen Entscheidungen. So können die Anforderungen bezüglich Verlustleistung, tiefer Betriebsspannung, kleiner Siliziumfläche und tiefer Taktrate am besten erfüllt werden beim Gebrauch einer anwendungsspezifischen DSP-Architektur, welche eine parallele Datenverarbeitung erlaubt. Andererseits erlaubt ein solcher DSP nicht die notwendige Flexibilität, um benutzerspezifische Algorithmen implementieren zu können.

Das nun folgend beschriebene Konzept war die Lösung, um einen Ausgang aus diesem Dilemma zu finden.

6.2 Das DSP-Koprozessor-Konzept

Eine Unterteilung der Hardware in eine Einheit, welche die fest definierte Bearbeitung der Daten in einer sehr effizienten Weise vornimmt, und in einen zweiten Teil, welcher in Form eines *General Purpose DSP* realisiert wurde und dem Anwender eine hohe Flexibilität garantiert, diente als Ansatz einer Implementierungslösung, welche allen Anforderungen, welche an den Digitalteil gestellt werden, Rechnung trägt [1]. Da die erste Einheit dem *General Purpose DSP* gewisse Aufgaben abnimmt und zudem von diesem kontrolliert und gesteuert wird, wird sie als Koprozessor des DSPs bezeichnet.

Die Audiodaten durchfliessen den Koprozessor immer auf die selbe Weise (Abbildung 42); Mono oder Stereo- Audiosamples werden vom Eingangsinterface von einem von AD-Wandlern üblichen Protokoll in ein internes, paralleles Format transformiert. Die Audiosamples durchlaufen dann die Vorfilter, den WOLA-Prozessor und die Nachfilter, bis sie zuletzt vom Ausgangsinterface in eines für DA-Wandler übliches, serielles Protokoll umgeformt werden und den Koprozessor verlassen.

Der *General Purpose DSP* kann einerseits die verschiedenen Verarbeitungseinheiten des Koprozessors konfigurieren und kontrollieren, andererseits kann er auch zu jedem Verarbeitungszeitpunkt zwischen Vor- und Nachfilter auf die Audiodaten zugreifen, diese lesen, analysieren, modifizieren und zurückschreiben. Die Verarbeitung innerhalb der WOLA-Filterbank kann dazu zu jedem Zeitpunkt gestoppt werden.

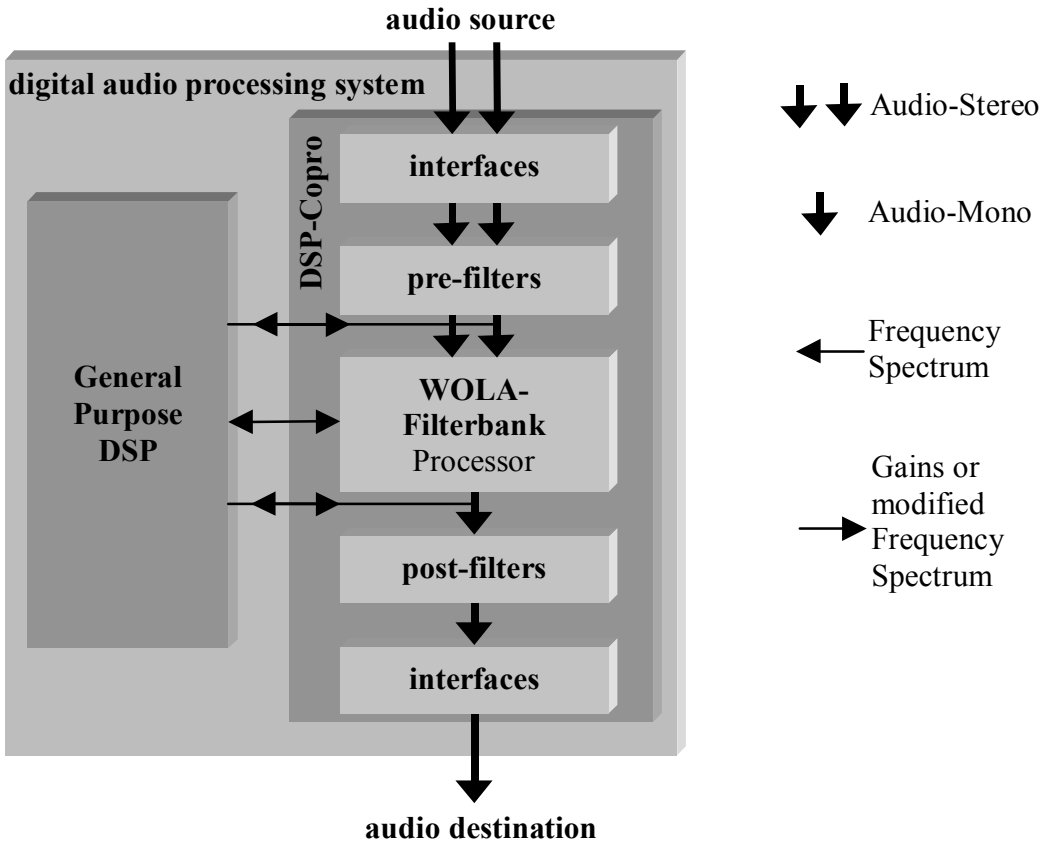


Abbildung 42: Das DSP-Koprozessor-Konzept

Das DSP-Koprozessor-Konzept stellt auf Systemniveau einen eleganten Lösungsansatz dar. Damit aber all die Anforderungen, welche an den Digitalteil gestellt werden, erfüllt werden können, müssen auch der *General Purpose DSP* und besonders der Koprozessor sehr sorgfältig implementiert werden. Nur dann ist es möglich, ein Design mit einer effizienten Datenverarbeitung bezüglich Verlustleistung und Datendurchsatz bei gleichzeitig minimierter Siliziumfläche zu erhalten.

Ob das Design die Randbedingungen erfüllt, hängt zum grössten Teil vom Koprozessor ab. Denn er ist es, welcher den grössten Teil der Datenverarbeitung durchführen muss. Dies ist nicht trivial, denn zur Verarbeitung der WOLA-, Vorfilter- und Nachfilter-Algorithmen wird eine beträchtliche Rechenleistung benötigt (siehe nächstes Unterkapitel). Diese muss bei einer Taktrate von lediglich 1.0 MHz zu erzielen sein, damit der Digitalteil direkt von der Batterie gespeist werden kann. Zudem muss der

Koprozessor sehr flexibel sein, damit er die verschiedenen Konfigurationen der FFT- und WOLA-Algorithmen verarbeiten kann. Da ein Grossteil der Datenverarbeitung vom Koprozessor durchgeführt wird, kann der *General Purpose DSP* relativ einfach sein.

6.3 Theoretische Aspekte der Audioverarbeitung

Der WOLA-Algorithmus

Während den letzten zwei Jahrzehnten wurden *Multi-Rate*-Signalverarbeitungstechniken stark entwickelt und werden heute in den verschiedensten Ingenieurdisziplinen eingesetzt [2][3]. Die Konditionen, um eine perfekte Rekonstruktion einer Filterbank zu erhalten, wurden äusserst gut erforscht und sind nun gut dokumentiert [4]. Eine perfekte Rekonstruktion legt einem System gewichtige Randbedingungen auf, welche bei gewissen Anwendungen nicht erfüllt werden können. In Anwendungen zum Beispiel, welche den Frequenzbereich stark modifizieren, sind andere Signalverarbeitungstechniken zu bevorzugen [5].

Zwei für diese Anwendungen häufig verwendete Strukturen, welche eine erfolgreiche Implementierung einer FFT-Filterbank zulässt, ist die *Polyphase* Struktur und die *Weighted Overlap Add* - Struktur (WOLA). Die mathematischen Grundlagen für diese beiden Strukturen sind identisch, sie unterscheiden sich lediglich in der Art, wie die Daten manipuliert werden. Sie haben deshalb auch ähnliche Vor- und Nachteile für eine Implementierung [6].

Die WOLA-Struktur, welche die Daten paketweise verarbeitet, liefert eine intuitivere Betrachtungsweise der Datenverarbeitung. Weiter eignet sich die WOLA-Struktur hervorragend für die Implementierung einer sogenannten *Block Floating Point Arithmetic* (Block-Gleitkommaarithmetik), welche Quantisierungsfehler, hervorgerufen durch eine Festkommarepräsentation der Daten, drastisch reduziert.

Abbildung 43 stellt vereinfacht das Blockdiagramm einer überabgetasteten[1] WOLA-Filterbank dar [5][6][7][8]. Für eine zweifache Überabtastung entspricht der Eingangsschritt (R) der halben FFT-Grösse (N). Wenn R und N identisch sind, ist die Filterbank kritisch abgetastet, da die Analysis-Fensterfunktion eine Grenzfrequenz von $2\pi/N$ hat. Eine Überabtastung hat folgende zwei Vorteile:

[1] *Oversampling*: Überabtastung

- Das Frequenzspektrum kann über einen weiten Bereich modifiziert werden, ohne dass dies zu einer wahrnehmbaren Distorsion des Audiosignals führt (Aliasing).
- Ein Kompromiss zwischen Gruppenverzögerung und Verlustleistung ist möglich. Oder anders ausgedrückt: Eine erhöhte Überabtastung verringert die Gruppenverzögerung, erhöht jedoch den Rechenbedarf und damit die Verlustleistung.

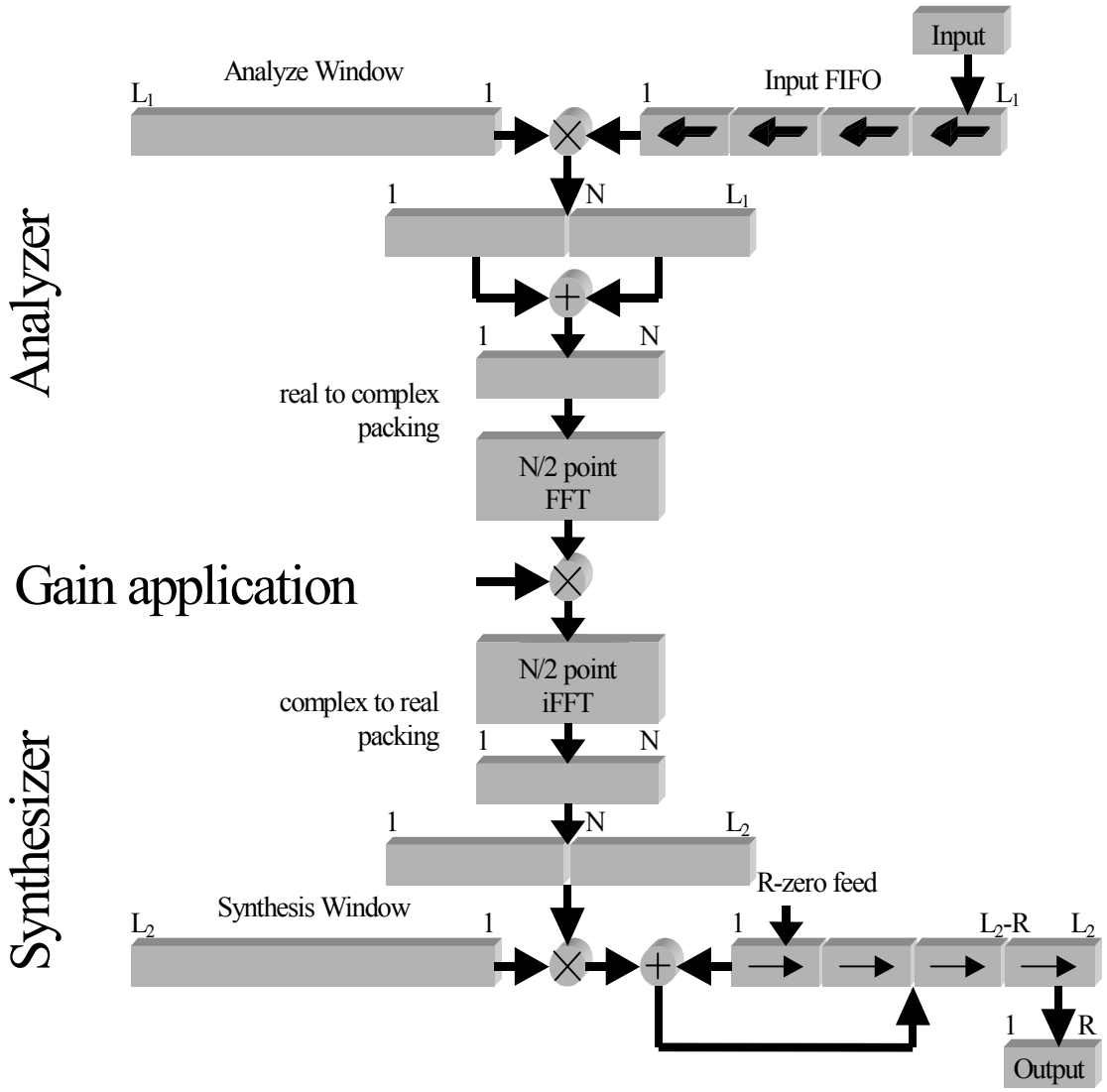


Abbildung 43: Vereinfachtes Blockdiagramm einer WOLA-Filterbank

Die WOLA-Filterbank arbeitet wie folgt: Das Eingangs-FIFO[1] wird um R Stellen nach links geschoben und R neue Samples[2] werden in den frei werdenden Raum geschrieben. Die L Samples des Eingangs-FIFO werden dann mit einer Fensterfunktion, welche die Form eines Tiefpassfilters hat, multipliziert. Der erzeugte Vektor wird in Untervektoren von N Samples

[1] FIFO=*First In, Last Out*
[2] *Sample*: Abtastwert

aufgeteilt, welche aufsummiert werden. Über das Resultat wird anschliessend eine FFT durchgeführt. Die FFT liefert nicht nur die Amplitude der einzelnen Frequenzbänder, sondern auch deren Phasen (komplexes Resultat).

Um eine Modifikation des ursprünglichen Zeitsignals zu erhalten, können die N/2 FFT-Ausgänge mit verschiedenen Verstärkungsfaktoren multipliziert werden. Anschliessend folgt eine invertierte WOLA-Transformation, welche in umgekehrter Reihenfolge die invertierten Operationen sämtlicher vorher beschriebener Operationen vornimmt. Das Resultat befindet sich schlussendlich im Ausgangs-FIFO, welches alle R Samples nach rechts geschoben wird und damit R verarbeitete Datenwerte freigibt.

Das in Abbildung 43 dargestellte Blockdiagramm ist stark vereinfacht. Damit die WOLA-Filterbank optimal arbeitet, sind noch folgende Operationen notwendig:

- FFT-Optimierung: N reelle Zahlen werden in N/2 komplexe Zahlen gepackt um sodann über diesen N/2 komplexen Zahlen eine FFT durchzuführen. Nach der FFT muss das Resultat mittels einer zusätzlichen Operation wiederum entpackt werden.
- Die FFT-Transformation und somit auch die WOLA-Transformation erzeugen zwei „halbe“ Bänder bei der Grenzfrequenz (*Nyquist frequency*) und dem Offset Kanal (*DC frequency*). Eine Modulation des Eingangssignals mit einem Rechtecksignal, zusammen mit einer „Vorrotation“ der komplexen Zahlen bevor die FFT durchgeführt wird, erlaubt die Erzeugung von Kanälen mit der selben Weite. Auch bei der Rücktransformation hat dies in invertierter Form zu geschehen.
- Werden zwei Eingangskanäle verarbeitet, so werden die Samples beider Kanäle abwechslungsweise im Eingangs-FIFO gespeichert (*interleaved*) und als komplexe Daten verarbeitet. Nach der FFT folgt dann eine zusätzliche Operation, welche das Resultat auftrennt um die individuellen Spektren beider Kanäle zu erhalten.

Die Vor- und Nachfilter

Um auch im Analogteil den Stromverbrauch so niedrig wie möglich zu halten, können bei Hörgeräten nicht die selben AD- und DA-Wandler verwendet werden wie bei *Hifi-Highend*-Geräten. Die verwendeten AD-Wandler haben häufig einen unkorrigierten Offsetfehler, welcher in den meisten Anwendungen unproblematisch ist, da ein Offset in einem Audiosignal vom Ohr nicht wahrgenommen wird. Anders sieht es jedoch bei der Verwendung einer WOLA-Filterbank aus. Nach der Zeit-Frequenz-Transformation befindet sich der Offsetwert im ersten Frequenzband. Wird

nun das Spektrum analysiert um den einzelnen Frequenzbändern verschiedene Verstärkungsfaktoren zuzuweisen, so wird der durch den Offsetwert erzeugte Wert innerhalb des ersten Frequenzbandes nicht als Offset erkannt, sondern als Signal mit einer tiefen Frequenz. Je nach Anwendung ist es dadurch möglich, dass die Verstärkungsfaktoren falsch berechnet werden.

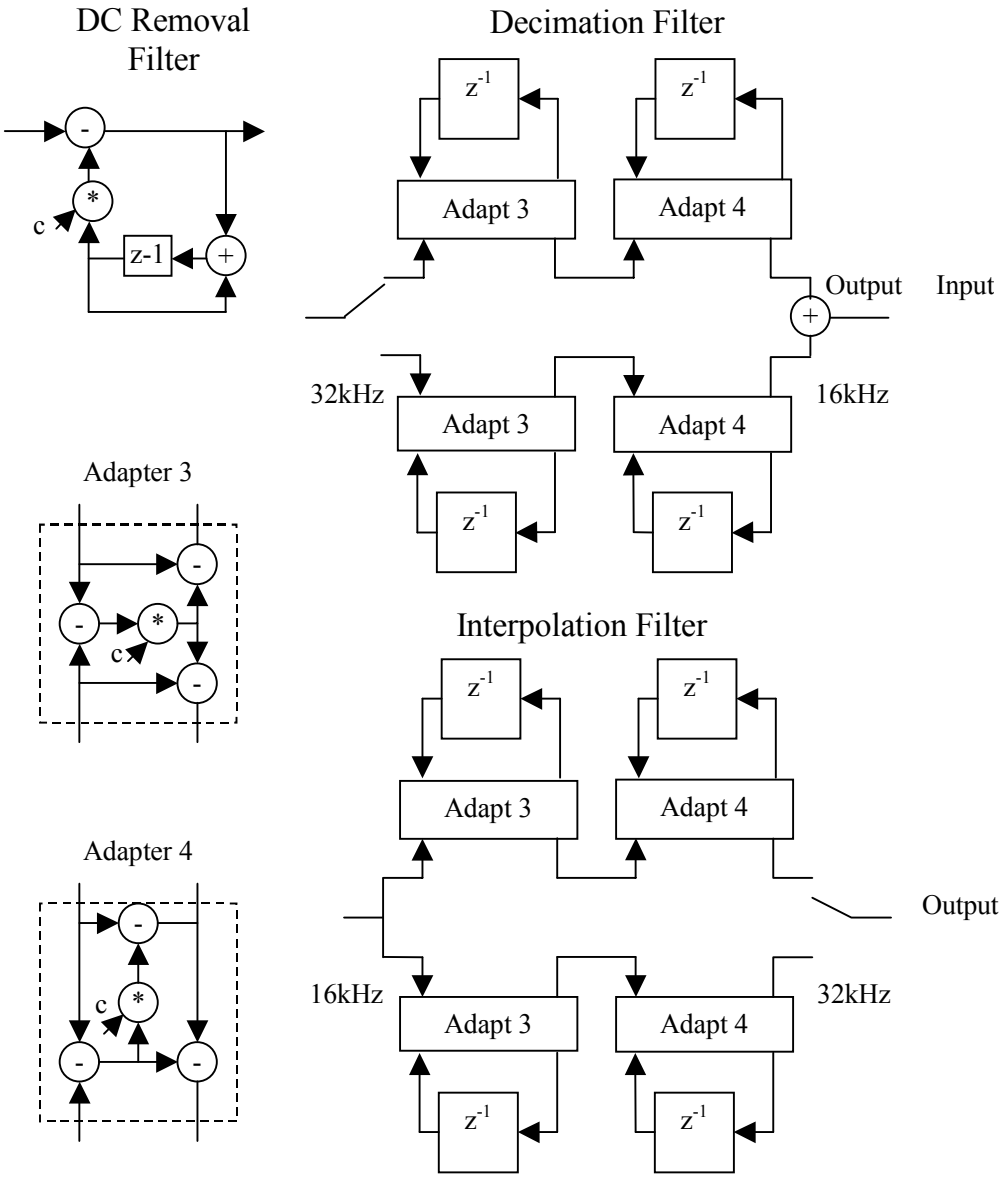


Abbildung 44:Filter zur Entfernung eines Offsets, zur Datendezimation und zur Dateninterpolation

Ein weiteres Problem welches im Zusammenhang mit Wandlern angetroffen wird, ist der Aliasingeffekt, der bei Unterabtastung eines Analogsignals erzeugt wird. Zur Reduktion dieses Effektes werden den AD-Wandlern analoge Tiefpassfilter vorgeschalten, welche die Frequenzen über einer

kritische Frequenz[1] abschneiden. Da diese Analogfilter keine abrupte Grenzfrequenz haben, tasten die AD-Wandler ein Analogsignal häufig in einer um einen ganzzahligen Faktor höheren Frequenz ab und dezimieren mit einem digitalen Filter den gewonnenen Datenstrom wiederum um diesen Faktor. Ebenfalls die DA-Wandler verwenden die selbe Technik um einen Aliasingeffekt zu unterdrücken.

Damit für Hörgeräte technisch einfache Wandler verwendet werden können, welche die vorhin besprochene Überabtastungs-Technik nicht verwenden und zudem einen Offsetfehler haben, erhält der Digitalteil Vor- und Nachfilter, welche die Mankos der Wandler kompensieren.

Der Vorfilter unterdrückt einen eventuell vom AD-Wandler erzeugten Offset und dezimiert den Datenstrom um den Faktor 2. Der Nachfilter erweitert (interpoliert) den Datenstrom wiederum um diesen Faktor[2]. Durch diese Interpolation und Dezimation wird auch eine Überabtastung des Faktors 2 realisiert. Die benötigten, digitalen Filter müssen aber nicht mehr zusammen mit den Wandlern in einem für Digitalelektronik häufig uneffizienten Analog-Fertigungsprozess integriert werden, sondern können in einer Digitaltechnologie realisiert werden.

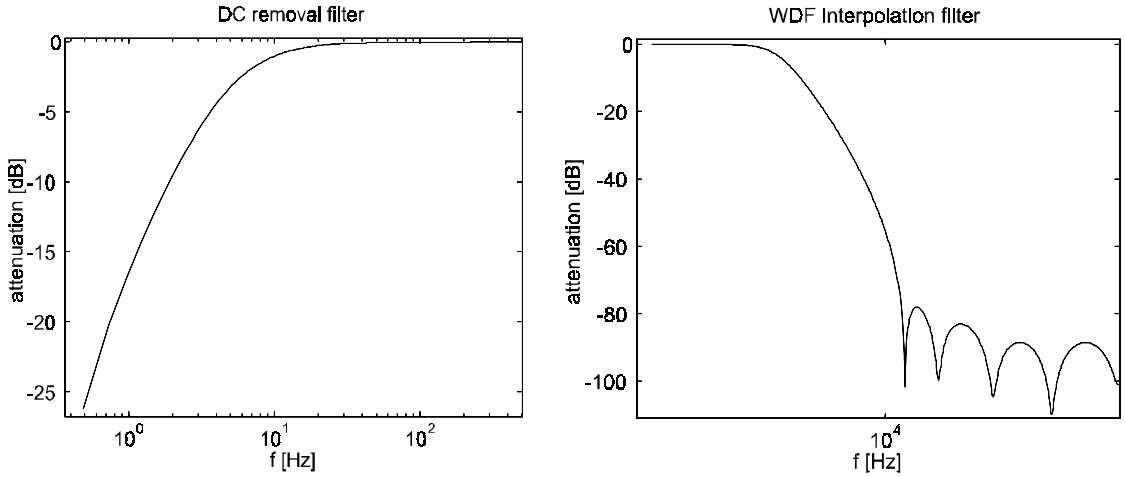


Abbildung 45: Charakteristik des Filters für die Offsetentfernung (links) und des Halbband-Tiefpassfilter (rechts)

Als Halbbandfilter von Dezimation- und Interpolationsfilter wurden *Bireciprocal Wave Digital Filter* (WDF) 9ter Ordnung gewählt [9][10][11]. Die Struktur dieser beiden Filter ist in Abbildung 44 dargestellt. Die Filtercharakteristik ist in Abbildung 45 abgebildet (linke Seite).

[1] Die Frequenz von Nyquist entspricht der halben Abtastfrequenz

[2] Datendezimierung = Halbbandfilter + *Downsampling*
Dateninterpolation = *Upsampling* + Halbbandfilter

Der Offset des Datenstromes kann durch dessen Subtraktion eliminiert werden, wobei dieser durch einen Integrator ermittelt wird, indem er den vom Offset korrigierten Datenstrom über eine sehr lange Zeit aufsummiert. Diese Filterstruktur und die Filtercharakteristik ist ebenfalls in Abbildung 44 bzw. in Abbildung 45 zu sehen.

6.4 Schlussfolgerung

Hörgeräte, und insbesondere CIC-Hörgeräte, sind typische Anwendungen, wo auf Grund technologischer Limiten audiologische Wünsche häufig nicht realisiert werden können. So wird dann auch jeder technologische Fortschritt zur Leistungssteigerung der digitalen Hörgeräte verwendet, anstatt von der Möglichkeit zu profitieren, eine schnellere Produkteentwicklung zu erzielen.

Die in diesem Kapitel vorgestellten Ideen bilden die Basis einer neuen Generation von Hörgeräten. Das präsentierte Konzept wurde so erarbeitet, dass, wenn sämtliche Entwicklungsetappen sorgfältig und optimal durchgeführt werden, eine Realisation der digitalen Verarbeitungseinheit gemäss dem aufgestellten Pflichtenheft mittels einer 0.15 μm bis 0.25 μm – CMOS–Technologie möglich sein soll.

Das DSP-Koprozessor-Konzept bildet auf Systemniveau einen Lösungsansatz, welcher massgeblich zum Gelingen des Projektes beitragen kann. Wichtig ist aber, dass die restlichen Entwurfsteile und das Backend sorgfältig durchgeführt wird. Nur dann ist es möglich, ein so kompaktes und leistungsbescheidenes Produkt herzustellen, dass es für CIC-Hörgeräte verwendet werden kann.

Referenzen

[1] Schneider, R. Brennan, P. Balsiger, A. Heubi, F. Pellandini, "An Ultra Low-power Programmable DSP System for Hearing Aids And Other Audio Applications", Proc. of the International Conference on Signal Processing Applications and Technology, ICSPAT, Orlando, FL, November 1-4, 1999.

[2] N. J. Fliege, “Multirate Digital Signal Processing”, JOHN WILEY & SONS, 1994

[3] W. W. Smith & J. M. Smith, “Handbook of Real-Time Fast Fourier Transforms”, IEEE Press, 1995.

[4] P. P. Vaidyanathan, “Multirate Systems And Filter Banks”, Prentice-Hall, 1993.

[5] R. Brennan & T. Schneider “A Flexible Filterbank Structure for Extensive Signal Manipulations in Digital Hearing Aids,” *Proc. ISCAS-98, Monterey, CA*

[6] R. E. Crochiere & L. R. Rabiner, “Multirate Digital Signal Processing”, Prentice-Hall, 1983
[7] R. Brennan & T. Schneider, “Filterbank Structure and Method for Filtering and Separating an Information Signal into Different Bands, Particularly for Audio Signals in Hearing Aids”, PCT Patent Publication WO09847313A210, October 22, 1998.
[8] T. Schneider & R. Brennan “A Multichannel Compression Scheme for a Digital Hearing Aid,” *Proc. ICASSP-97*, Munich, Germany.
[9] Fettweis & al., “Proceedings on the IEEE, VOL. 74, No 2, February 1986”, page 270-327
[10] L. Gazsi, „Explicit Formulas for Lattice Wave Digital Filters”, IEEE Trans. Circ. & Sys., Vol 32, 68-88, 1985
[11] N.P.C. Manshanden & al., „Design of Wave Digital Filters with Minimal Coefficientlength”.

7 Die Implementierung und Integration des Koprozessors, Resultate

Die drei präsentierten Methoden und Lösungsansätze ermöglichen eine sehr effiziente Implementierung des im letzten Kapitel erwähnten DSP-Koprozessors. Da die meisten Unterblöcke des Koprozessors bereits als Anwendungsbeispiel für die verschiedenen Lösungsansätze verwendet wurden, brauchen diese Blöcke nur noch zusammengestellt werden, um den Koprozessor zu kriegen.
Mittels einer FPGA-Implementierung konnte die Funktionsweise des Digitalteils für Hörgeräte erstmals real-time *verifiziert werden. Der Koprozessor wurde sodann in verschiedenen Technologien integriert.*
Selbst wenn die Grösse des Digitalteils im Gegensatz zu anderen, modernen Prozessoren gering ist, erschwerten gewisse Designeigenschaften, welche meistens durch die Designrandbedingungen bedingt sind, die Integrationen. Diese mussten deshalb speziell sorgfältig durchgeführt werden. Der Standard-Backend-Designfluss musste erweitert werden, damit alle Randbedingungen berücksichtigt werden konnten. Dies wirkte sich im zeitlichen Aufwand des Backends *aus, ermöglichte aber, dass die Integrationen die Spezifikationen erfüllten.*
Die audiologischen und physikalischen Resultate der Integrationen erfüllen vollständig die gestellten Randbedingungen und Spezifikationen. Das Quantisierungsrauschen des Gesamtsystems ist stark konfigurationsabhängig und beträgt für eine Standardkonfiguration etwa 90 dB, was einer 15-Bit-Auflösung entspricht und bezüglich den algorithmenbedingten Aliasingeffekte vernachlässigbar gering ist.
Der Stromverbrauch des Systems ist stark integrationsabhängig und beträgt bei der neusten 0.18 μm –Integration je nach der verwendeten Konfiguration zwischen 95 μA und 215 μA bei einer Speisespannung von 1.2 Volt.

Dieses Kapitel widmet sich den verschiedenen Integrationen, wobei dem Backend-Designfluss und den Integrationsresultaten besondere Aufmerksamkeit zugeteilt wird.

7.1 Die Implementierung des digitalen Verarbeitungsteils für Hörgeräte und im Speziellen diejenige des DSP-Koprozessors

So wie der DSP-Koprozessor konzipiert wurde, kann er für Anwendungen eingesetzt werden, welche einen speziell an die Anwendung angepassten DSP verwenden, welcher zusammen mit dem Koprozessor in einen ASIC integriert wird, oder aber er kann zusammen mit einem frei käuflichen *General Purpose DSP* verwendet werden. Der Unterschied beider Implementierungen befindet sich lediglich in einigen peripheren Anpassungen, nicht aber im Koprozessor selbst.

Aus Platzgründen kommt für digitale Hörgeräte nur eine gemeinsame Integration von DSP und Koprozessor in Frage. Abbildung 42 (vorausgehendes Kapitel) zeigt für diesen Fall den Audiofluss innerhalb des Koprozessors und die Kontroll- und Interventionsmöglichkeiten des DSPs. Abbildung 46 zeigt die architektonische Strukturierung des *Top Levels*.

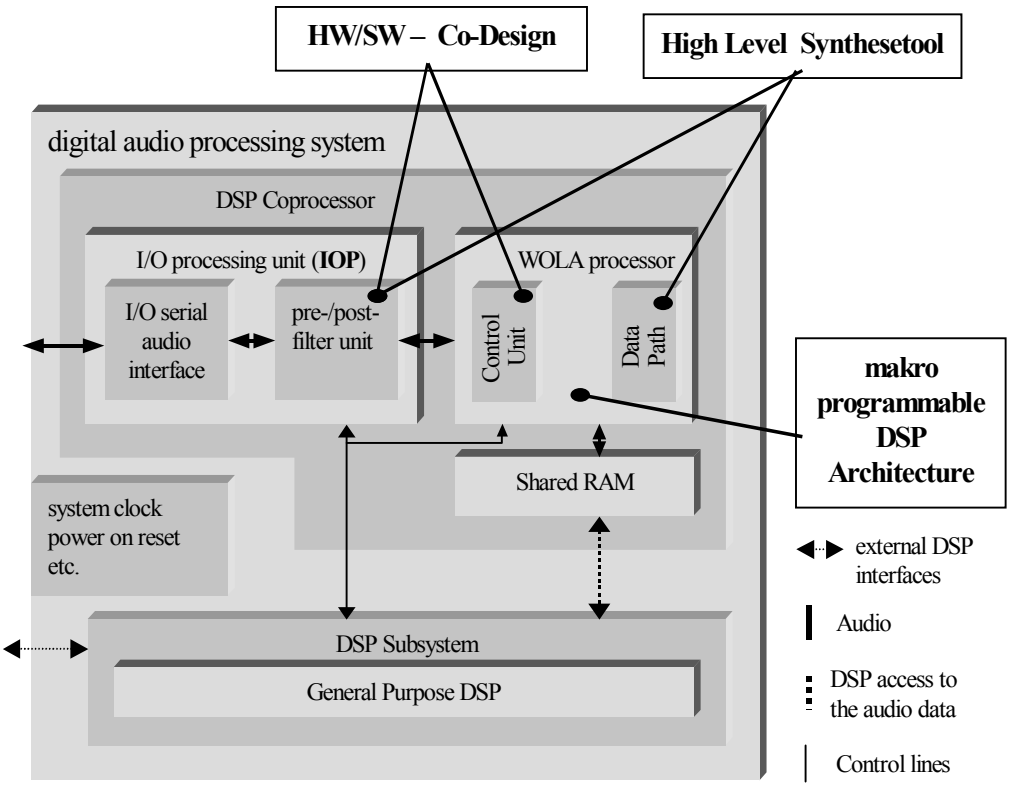


Abbildung 46: Blockdiagramm des Datenflusses und architektonische Strukturierung des Digitalteils / Die Verwendung der drei in den letzten Kapiteln präsentierten Lösungsansätze für die Implementierung des Digitalteils

Der Digitalteil ist auf erster Ebene in drei Hauptblöcke unterteilt. Der erste wird vom DSP und seinen peripheren Einheiten gebildet. Er kommuniziert mit dem zweiten Block, dem Koprozessor, über Kontroll- und Konfigurationsbusse und kann auf die RAMs zugreifen, welche im Koprozessor situiert sind. Im dritten Block sind Teile zu finden, welche für

das gesamte System Bedeutung hat wie zum Beispiel der *Power-On-Reset*, die Generierung des Systemclocks, usw.

Da der Hörgerätedigitalteil bereits als Anwendungsbeispiel für die makroprogrammierbare DSP-Architektur diente und eingehend diskutiert wurde, konzentriert sich dieses Kapitel lediglich auf die Implementierungen des Koprozessors und im speziellen auf die Realisation des FFT-Prozessors.

Sämtliche für eine effiziente Implementierung des DSP-Koprozessors notwendigen Lösungsansätze und Designmethoden wurden in den Kapiteln 3 bis 5 entwickelt und eingehend diskutiert. Die einzelnen Blöcke des Koprozessors dienten dabei als Anwendungsbeispiele und sind nun bereits realisiert. So lassen sich die Vor- und Nachfilter des *IOPs* einfach mittels dem *high-level* DSP-Synthesetool oder dem HotDSP realisieren. Die makroprogrammierbare DSP-Architektur eignet sich hervorragend zum Bau der WOLA-Filterbank und dessen Datenverarbeitungsteil lässt sich wiederum am einfachsten mit dem DSP-Synthesetool generieren. Der HotDSP schlussendlich kann auch zum Bau der Kontrolleinheit der makroprogrammierbaren DSP-Architektur verwendet werden. Was nun noch bleibt, ist die Zusammenstellung dieser Blöcke und die Anbindung an einen *General Purpose DSP*.

Das IOP

Wie in Abbildung 46 ersichtlich ist, besteht der Koprozessor aus dem IOP[1] und dem WOLA-Prozessor (oder: FFT-Prozessor).

Das IOP besteht aus dem seriellen Audiointerface SAI[2] und einer Filtereinheit (Abbildung 47). Das Audiointerface konvertiert einen von einem Mono- oder Stereo-AD-Wandler erzeugten, seriellen Datenstrom in ein internes, paralleles Datenformat und umgekehrt die zurückkommenden, gefilterten Daten von dessen parallelen Format wiederum in einen für DA-Wandler brauchbaren, seriellen *Bitstream*. Da das SAI einen relativ kleinen Block darstellt und im Wesentlichen nur aus zwei *Shift*-Registern besteht, wurde eine Implementierung mittels einer handgeschriebenen VHDL-Beschreibung gewählt.

Der Filterblock enthält die Eingangsfilter, bestehend aus dem Offsetentfernungsfilter und dem Dezimationsfilter sowie den Interpolationsfilter als Ausgangsfilter. Der Filterblock wurde auf verschiedene Arten implementiert. Eine erste Implementierung wurde mit Hilfe einer einfachen Software durchgeführt, welche eine Strukturbeschreibung des Filters in einer speziellen Syntax liest und in eine

[1] IOP: ***Input / Output Processing Unit***
[2] SAI: *Serial Audio Interface*

VHDL-Syntax umschreibt und zugleich ein Postscript-File zur Veranschaulichung des Datenflusses erzeugt (siehe Abbildung 59, Seite T des Anhangs I). Die Strukturbeschreibung enthält dabei sämtliche Informationen bezüglich Zeitpunkt und Ort der Verarbeitung einer Operation. Für eine zweite Implementierung wurden zwei HotDSPs eingesetzt (siehe die Anwendungsbeispiele des Kapitels 3). Eine letzte Implementierungslösung wurde mit Hilfe des *high-level* DSP-Synthesetools durchgeführt.

SAI und der Filterblock können vom DSP kontrolliert werden. So kann der Letztgenannte zum Beispiel das exakte Datenformat der externen Wandler spezifizieren, definieren, ob eine Mono- oder Stereoverarbeitung stattfindet, die Grenzfrequenz des Offsetentfernungsfilter einstellen und Dezimations- und Interpolationsfilter ein- und ausschalten.

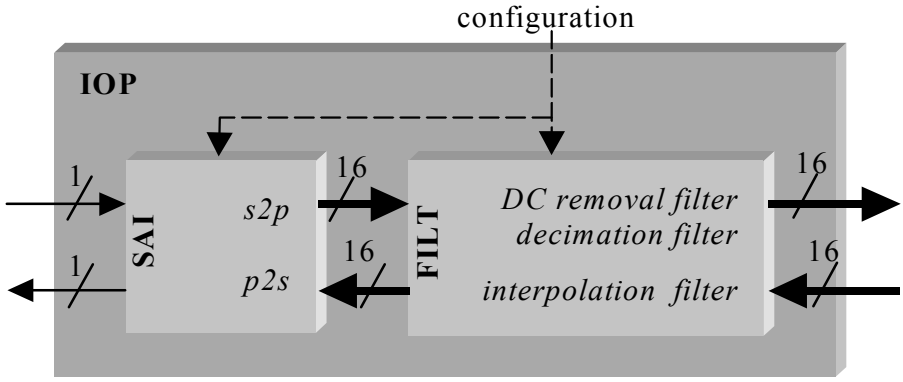


Abbildung 47: Die Input/Output- Verarbeitungseinheit (IOP)

Der WOLA-Prozessor

Die makroprogrammierbare DSP-Architektur des Kapitels 5 wurde zur Realisierung des WOLA-Prozessors verwendet. Der Prozessor wurde mit VHDL „von Hand“ implementiert. Die Effizienz des gewählten Lösungsansatzes liegt dabei natürlich nicht in der Implementierungsmethode selbst, sondern in der Art, wie die FFT-Algorithmen analysiert wurden, um dann aus deren Regularität Konstruktionsvorteile zu gewinnen. Ausser dem Datenpfad beinhalten alle Unterblöcke des WOLA-Prozessors hauptsächlich Kontrollmechanismen und wurden deshalb von Hand implementiert.

Ebenfalls der Datenpfad des WOLA-Prozessors wurde zuerst von Hand implementiert. Er ist der grösste, komplexeste und wichtigste Teil des Prozessors und verlangte demnach auch die entsprechende Aufmerksamkeit, sprich Entwicklungszeit. Wie der Datenpfad auch realisiert werden kann, und dies in einer sehr schnellen Entwicklungszeit, veranschaulicht auf eine sehr eindrückliche Weise Kapitel 4.

7.2 FPGA- und ASIC-Integrationen

Nebst einer FPGA-Implementierung zur Verifizierung des Systems in Echtzeit, wurde der DSP-Koprozessor insgesamt 3 mal integriert.

Abbildung 48 zeigt den FPGA-Prototypen-Aufbau des Hörgerätesystems: Das kleine PCB links enthält den Analogchip. Der gesamte Koprozessor inklusive der Speicher befindet sich auf dem mittleren PCB. Das rechte PCB enthält den DSP mit seinen peripheren Einheiten, Speichern, Systemclock, EEPROM und Interfaces.

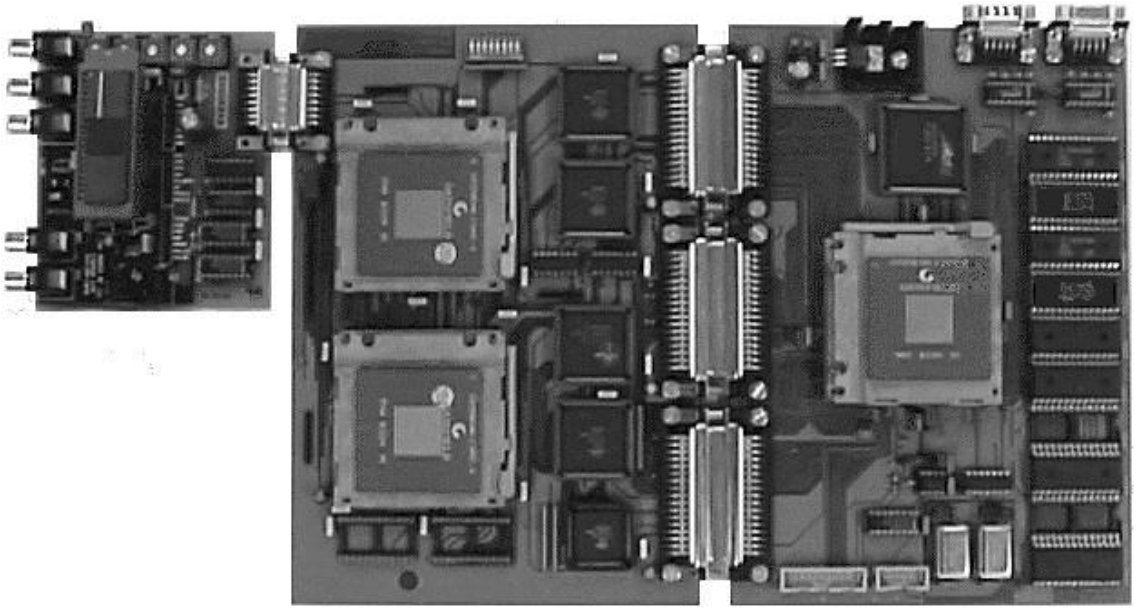


Abbildung 48: FPGA-Prototypen-Aufbau des Hörgerätesystems

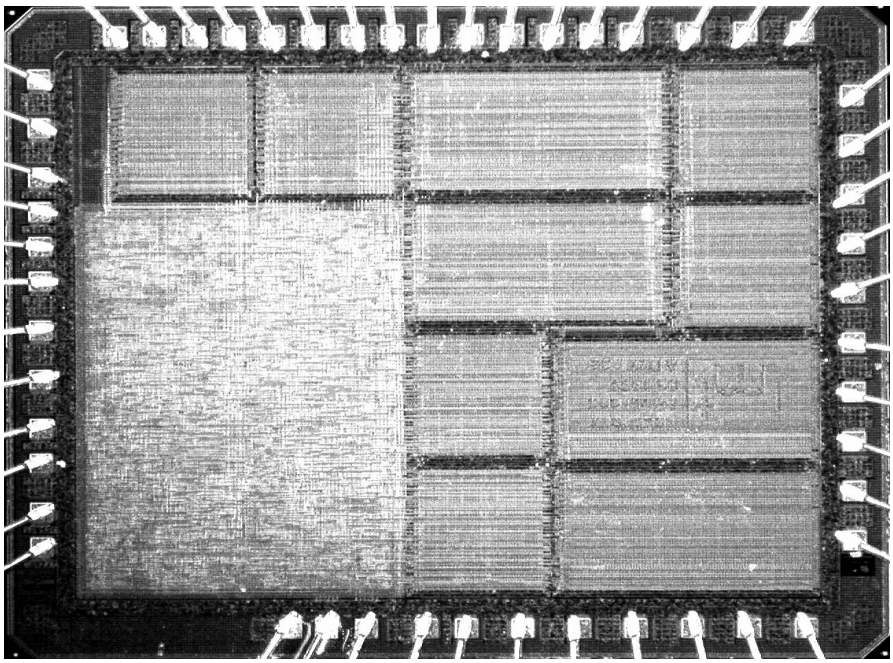


Abbildung 49: Eine Testintegration des Koprozessors (ohne General Purpose DSP) in einer 5-Metal-0.35µm-CMOS-Technologie (Alcatel-Mietec MTC45000)

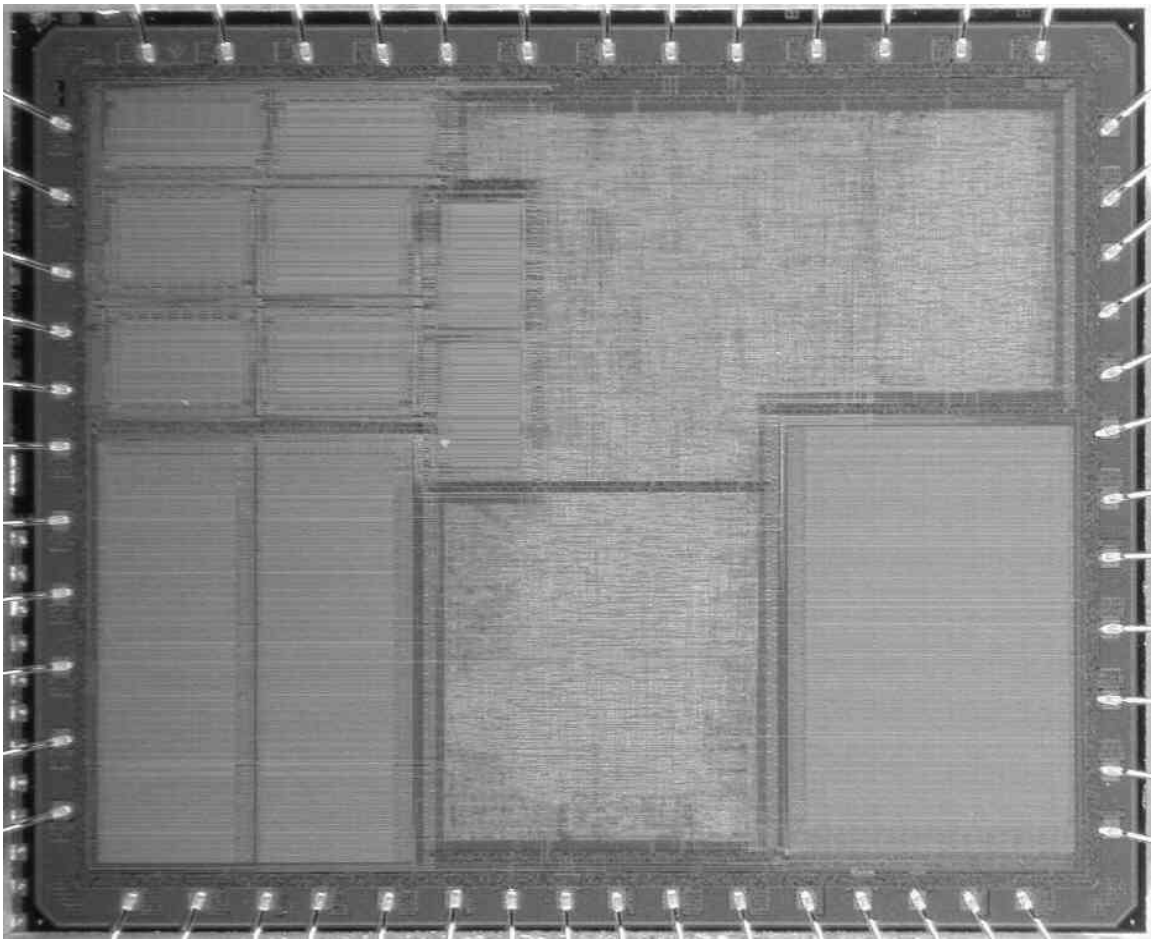


Abbildung 50: Eine Testintegration des Koprozessors zusammen mit dem General Purpose DSP in einer 5-Metal-0.35µm-CMOS-Technologie (Alcatel-Mietec MTC45000)

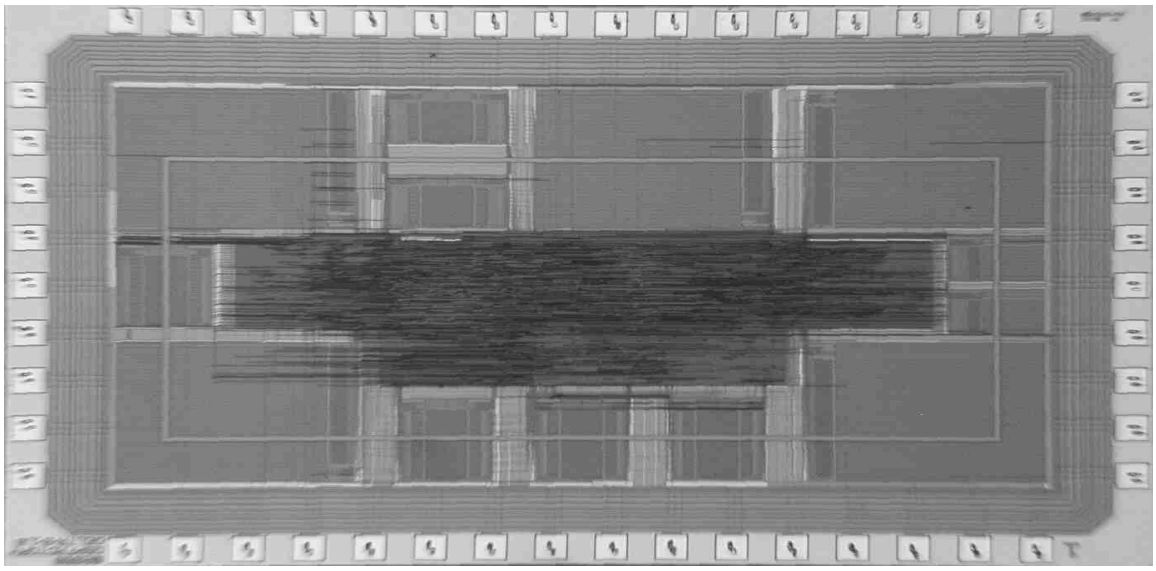


Abbildung 51: Eine nicht am IMT durchgeführte Integration des Koprozessors zusammen mit dem General Purpose DSP in einer 6-Metal-0.18µm-CMOS-Technologie.

Abbildung 49 zeigt eine erste Integration des Koprozessors in einer 5-Metal-0.35µm-CMOS-Technologie. Da der Kontroll-DSP nicht mitintegriert ist, enthält der Chip konfigurierbare Interfaces, welche den Koprozessor an alle

marktüblichen *General Purpose DSPs* anschliessen lässt. Im weiteren enthält der Chip ebenfalls hoch konfigurierbare Audio-Interfaces und ein EEPROM-Interface für den *Stand-Alone*-Betrieb. Dies ist eine am IMT durchgeführte Testintegration und diente zur Studie von Integrationen mit *Deep-Submicron*-Technologien.

Die Abbildung 50 zeigt eine Integration des Koprozessors zusammen mit dem General Purpose DSP in einer 5-Metal-0.35µm-CMOS-Technologie (Alcatel-Mietec MTC45000). Der DSP besitzt zwei 2 *kWord* Datenspeicher (X- und Y-Bus) und einen 4 *kWord* Programmspeicher. Dies ist eine am IMT durchgeführte Testintegration und diente zur Studie und Verifikation von *Scan Chain* - Techniken und komplexen *Gated Clock* - Strukturen.

Eine für kommerzielle Zwecke und nicht am IMT durchgeführte Integration des Koprozessors zusammen mit dem General Purpose DSP in einer 6-Metal-0.18µm-CMOS-Technologie ist in Abbildung 51 zu sehen. Der DSP besitzt zwei 4 *kWord* Datenspeicher (X- und Y-Bus) und einen 12 *kWord* Programmspeicher.

Synthese- und Backend-Flow

Da der Backend-Designfluss nicht das eigentliche Thema dieser Arbeit sein soll, wird im Folgenden nur sehr kurz auf die Integration des Koprozessors und des Digitalteils eingegangen. Für jede der vorhin aufgezählten ASIC-Integrationen wurde ein etwas anderer Designfluss verwendet, wobei natürlich die in Abbildung 52 gezeigte grobe Designflusseinteilung in allen Fällen gültig ist.

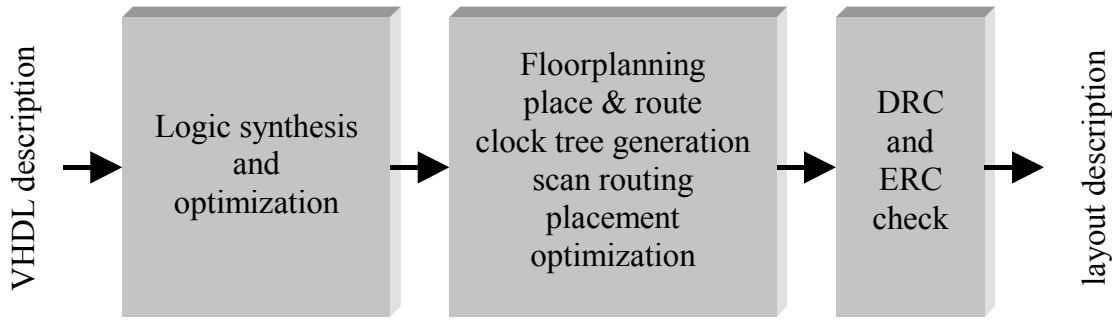


Abbildung 52: Backend-Designfluss

In einem ersten Schritt wird das Design synthetisiert und optimiert, was normalerweise noch als Teil des Frontend-Designs betrachtet wird. Die vom Synthesetool erzeugte Netzliste wird dann dem(n) Backendtool(s) übergeben. In verschiedenen Etappen wird nun der gesamte Mikrochip gestaltet. Bevor das erhaltene Layout zur Fabrikation des Chips verwendet

wird, werden verschiedene Tests wie DRC[1] und ERC[2] durchgeführt und der gesamte Chip ein letztes Mal simuliert.

Die Testintegration des Koprozessors zusammen mit dem *General Purpose DSP* in der 5-Metal-0.35µm-CMOS-Technologie (Abbildung 50) wurde mit spezieller Sorgfalt durchgeführt und soll als Beispiel näher betrachtet werden.

Entgegen allen Versprechungen und der schönen Werbung der Designtoolhersteller konnte bis anhin keine Vereinfachung des Backend-Designflusses festgestellt werden[3]. Sicherlich kann ein Digitalchip, an den keine speziellen Anforderungen gestellt wird, mit einem Standard-Designfluss in sehr kurzer Zeit realisiert werden. Werden jedoch spezielle Designtechniken wie *Gated Clocks*, *Scan Chains* oder verschiedene *Power Domains* verwendet, oder sollen ganz einfach die guten Timingresultate der Synthese nicht durch ein schlechtes *Place&Route* zunichte gemacht werden, so muss ein Designfluss verwendet werden, der alles andere als einen Maus-Klick-Charakter hat.

Das hier betrachtete Design hat folgende Eigenschaften und Fakten, welche während der Integration besonders berücksichtigt werden müssen:

- Zwei völlig getrennte *Clock Domains.*
- Etwa 40 *Gated Clocks.*
- 8 *Scan Chains*, deren Ein- und Ausgänge an bestehende, funktionale *Pins* angeschlossen werden müssen (via Multiplexer).
- Getrennte Speisung von Speicher, Standardlogik und IO-Pads.
- Es gibt einige zeitkritische Signale hauptsächlich wegen der Verwendung der synchronen RAMs, welche bereits nach einer halben Periode die gültigen Adress- und Datenwörter benötigen und andererseits erst nach der ersten halben Periode das ausgelesene Datenwort zur Verfügung stellen.
- Die grosse Menge an Speichern (11 RAMs insgesamt) und die Tatsache, dass verschiedene Speicher vom DSP und vom Koprozessor verwendet werden und damit nicht eindeutig dem DSP oder dem Koprozessor zugeordnet werden können, führt zu Platzierungsschwierigkeiten der RAMs.
- Die angesprochenen Platzierungsschwierigkeiten der RAMs und die von einigen *Two-Port-RAMs* verwendeten Rundherumanordnung der

[1] DRC: *Design Rule Check*
[2] ERC: *Electrical Rule Check*
[3] Dies bezieht sich im Besonderen auf die für diese Arbeit verwendeten *High-End-Tools* und zum Zeitpunkt der Integrationen (98-99)

Anschlusspins führt zu physikalisch langen Adress- und Datenleitungen mit einer hohe Kapazität, was sich negativ auf das *Timing* auswirkt.

Der *Design Compiler* von Synopsys mit all seinen Modulen (*RTL-Analyzer, ECO-Compiler, DesignWare* etc.) standen für die Synthese und für die Logikoptimierung zur Verfügung.
Silicon Ensemble (Ultra= und *Design Planner* (beide *Cadence*) mit ihren Tools (*CT-Gen, Q-Placer, Ultra-Placer, Warp-Router, Perl*, etc.) konnten für das Backend verwendet werden. *Silicon Ensemble* wie *Design Planner* sind P&R-Umgebungen und sind für einfache Designs vollständig. Gewisse Operationen können direkt durchgeführt werden (wie Floorplanning), für andere Funktionen werden Programme im „Hintergrund“ aufgerufen, welche via Datenexport und –Import Zugriff auf die Datenbasis beider *P&R*-Umgebungen haben. Jede der Umgebungen besitzt gewisse Vorteile gegenüber der anderen, welche bei einem aufwendigeren Designs entscheidend sein können bei der Wahl der zu benutzenden Umgebung. Um ein gutes Resultat der hier beschriebenen Integration zu erzielen, mussten von den Vorteilen beider *P&R*-Umgebungen Gebrauch gemacht werden. Dies machte den Designfluss sehr kompliziert wie im folgenden gezeigt wird, war aber notwendig, die gewünschte Designeffizienz zu erzielen.

Die Vorteile von *Silicon Ensemble* bezüglich *Design Planner* sind:
- Lesen und schreiben von hierarchischen Verilog-Netzlisten.
- Besseres (nicht gutes!) manuelles *Routen*, welches zum Beispiel beim *Power Routing* vorteilhaft ist.
- Direkter Aufruf der verschiedenen Router.

Auch der *Design Planner* hat gewisse Vorzüge bezüglich *Silicon Ensemble*:
- Einfacheres Floorplanning.
- Wenn alle Elemente platziert sind, kann *Design Planner* die Kapazitäten der Verbindungsleitungen abschätzen. Auf diese Weise kann eine Reoptimierung durchgeführt werden, ohne dass das Design teilweise oder vollständig geroutet ist.

Abbildung 53 zeigt den für die Integration verwendeten Designfluss, kolonnenweise nach den Tools geordnet. Es folgt eine kurze Auflistung der verschiedenen Etappen:
- Im Synthesetool (*Design Compiler*) wird die Netzliste des gesamten Design für das Backend vorbereitet. Die Register wurden durch *Scan-Register* ersetzt, jedoch noch nicht miteinander verbunden, da die schlussendliche Platzierung der Register noch nicht bekannt ist und deshalb eine optimale Verbindung zu diesem Zeitpunkt noch nicht möglich ist. Da die *Scan*-Verbindungen zu einem späteren Zeitpunkt gemacht werden müssen, und weil nicht explizite *Scan-Ports* verwendet werden, sondern später die *Scan*-Ketten mittels Multiplexer an

bestehende *Pads* angeschlossen werden sollen, ist es wichtig, dass die Hierarchie des Designs für die spätere Modifikation der Netzliste erhalten bleibt (kein *Flatten*).
- Via einer Verilog-Netzliste wird das Design auf die Backendtools übertragen.
- Der *Design Planner* wird wegen des Komforts für das Floorplanning verwendet.
- Via *DEF*-File wird das Design auf *Silicon Ensemble* übertragen.
- Power Routing: Wegen den verschiedenen Speisungen (Speicher, Standardlogik) und der speziellen Anordnung der Standardzellen ist das Routing der Speisung nicht trivial und wird deshalb mit *Silicon Ensemble* gemacht.
- In *Silicon Ensemble* wird eine erste Platzierung der Standardzellen durchgeführt (*timing-driven Ultra-Placement*, ohne Logikoptimierung)
- Da nun die Koordinaten aller Register provisorisch fixiert sind, können die *Scan-Register* optimal miteinander zu *Scan*-Ketten verbunden werden. Ein Verilog- und ein *PDEF*-File transferieren dazu die Netzliste bzw. die Koordinaten aller Standardzellen zum *Design Compiler*, wo die *Scan*-Ketten gebildet werden. Keine Logikoptimierung wird dabei durchgeführt. Die modifizierte Netzliste wird anschliessend wiederum via Verilog-File auf *Silicon Ensemble* übertragen.
- Es folgen nun zweimal eine Neuplatzierung der Standardzellen in *Silicon Ensemble* (*timing-driven Ultra-placement*) mit anschliessender Optimierung der Logik mittels *Design Compiler* (Reoptimierung, *Post-Layout*-Optimierung). Verilog- und PDEF-Files transferieren dabei die Netzliste und Platzierungsinformationen von *Silicon Ensemble* zum *Design Compiler*. *Timing*- und Kapazitätsinformationen können nicht von *Silicon Ensemble* erzeugt werden, da das Design noch nicht geroutet ist. Deshalb wird das Design jeweils mittels *DEF*-File auf den *Design Planner* übertragen, welcher an Hand von statistischen Tabellen die Einflüsse des Routings abschätzen kann. *Design Planner* liefert dann ein *SDF*- und ein *set_load*-File an den *Design Compiler*.
- Eine letzte Platzierung der Standardzellen wird in *Silicon Ensemble* durchgeführt (*incremental placement, ECO*).
- Der *Clock Tree* wird mittels *CT-Gen* erzeugt.
- Die Fläche der Standardzellen wird mit Füllzellen komplettiert (*filler cell*) und die Speisung der Standardzellen sichergestellt.
- Der *Clock tree* wird geroutet.
- Der *WARP*-Router *routet* das gesamte Design.
- Eine Timinganalyse des Design mittels dem *Design Compiler* fand keine *Timing Rule Violations*, eine weitere Optimierungsiteration ist deshalb nicht notwendig.

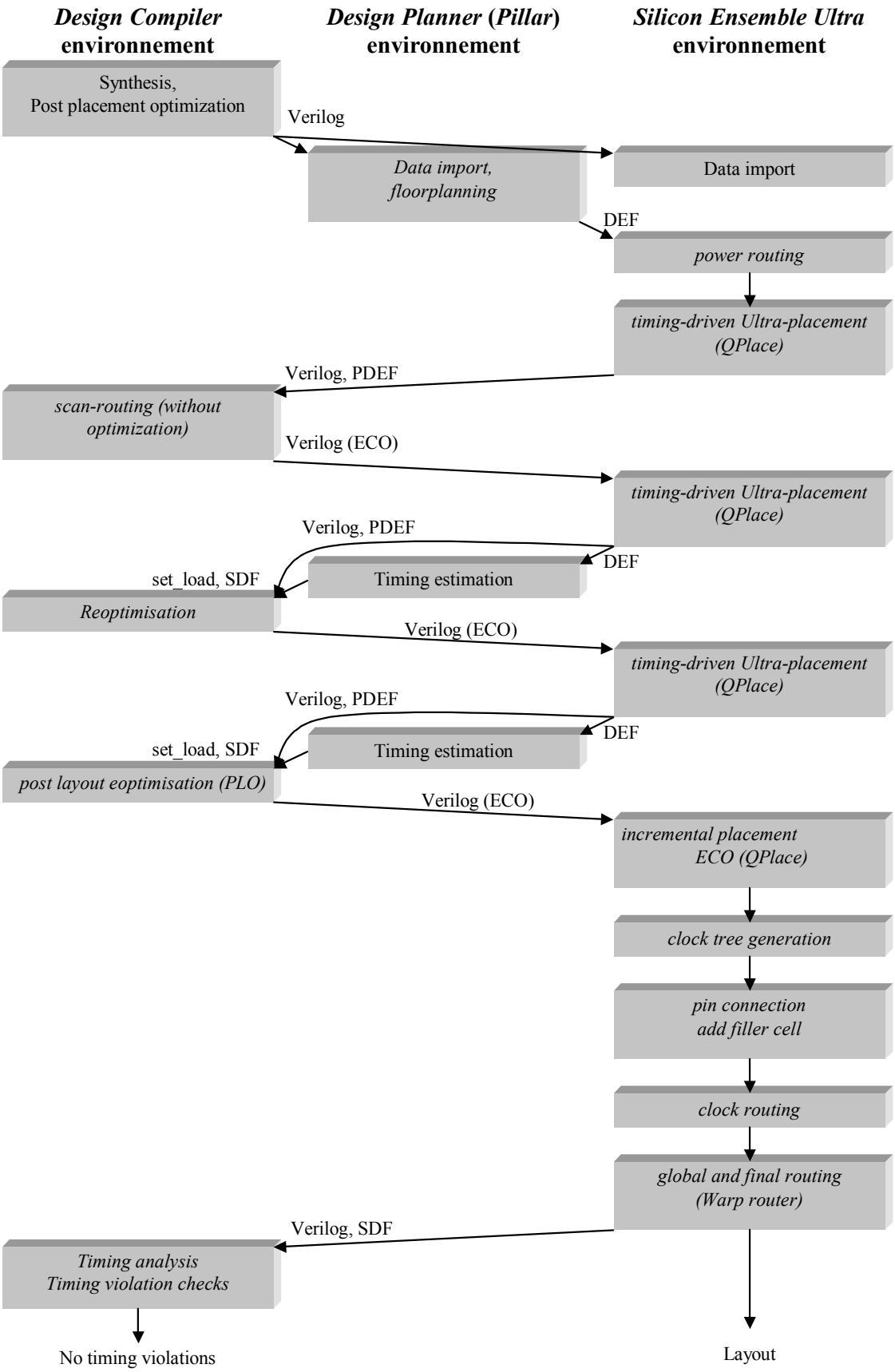


Abbildung 53: Backend-Designfluss: Da bei DSM-Technologien die Leiterkapazitäten nur schlecht geschätzt werden können, werden mehrere Iterationen zwischen den Frontendtools (Logik-Optimizer) und den Backendtools (Place&Route) benötigt.

Der präsentierte Designfluss ist deshalb mühsam, da einerseits verschiedene Optimierungsiterationen zwischen Frontend- und Backendtools durchgeführt werden müssen, was bei DSM-Technologien üblich ist, andererseits nicht dasselbe Tool für die Generierung der hierarchischen Verilog-Netzlisten und die Timing-Abschätzungen verwendet werden konnte.

Wie von den Herstellern der verwendeten EDA-Tools versprochen wird, sollen beide Probleme in der nächsten Zeit verbessert werden. Das Synthesetool soll bereits während der Synthese genügend Informationen bezüglich Anordnung der Blöcke und Zellen besitzen um die Kapazitäten durch das Routing abschätzen zu können[1] [2]. Dadurch sollen Optimierungsiterationen zwischen Frontend- und Backendtools überflüssig werden. Im weiteren werden *Silicon Ensemble* und *Design Planner* zu einem neuen Tool zusammengelegt, was das Backend ebenfalls vereinfachen soll.

7.3 Resultate

Audioqualität

Die Audioqualität der digitalen Datenverarbeitungseinheit für Hörhilfen wird einerseits beeinflusst von den verwendeten Algorithmen, andererseits auch von der Art, wie diese implementiert wurden.

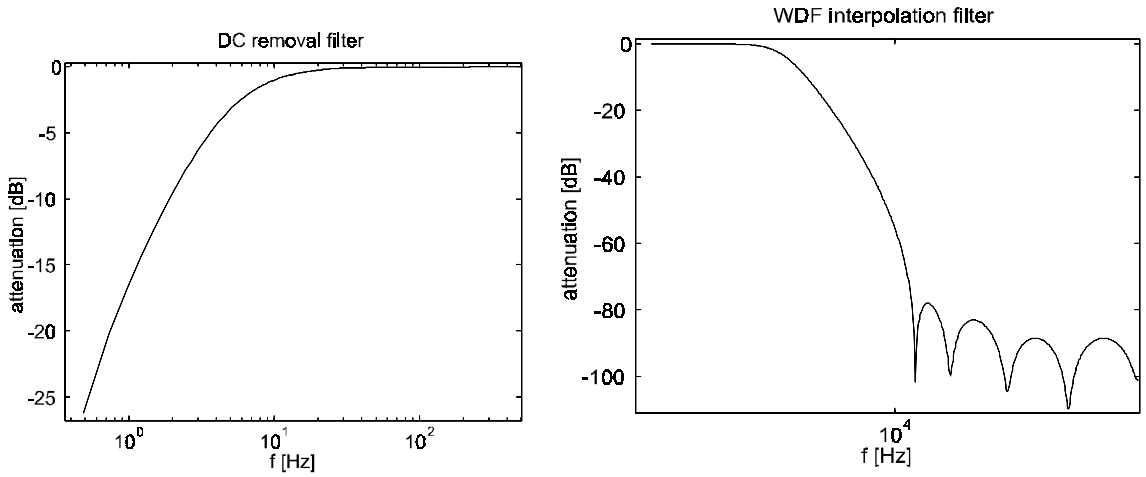


Abbildung 54: Charakteristik des Filters für die Offsetentfernung und diejenige des Halbband-Tiefpassfilter

[1] Der *Physical Compiler* von *Synopsys* erweitert den *Design Compiler* mit der Fähigkeit, ein Design gleichzeitig zu synthetisierten und zu platzieren (sehr vereinfacht).

[2] Die *PKS*-Synthese von *Cadence* ist das Pendant zum *Physical Compiler*.
PKS = Physically Knowledgeable Synthesis

Der linke Teil der Abbildung 54 zeigt die Charakteristik des verwendeten Filters zur Entfernung eines Offsets des eingehenden Datenstroms. Die Grenzfrequenz ist konfigurierbar und kann 5 Hz, 10 Hz oder 20 Hz betragen. *Bireciprocal Wave Digital Filter* (WDF) 9ter Ordnung bilden die Halbpassfilter der Dezimations- und Interpolationsfilter. Die Filterkoeffizienten sind so gewählt, dass das höhere Halbband mit 80 dB unterdrückt wird. Der Aliasingeffekt von Dezimations- und Interpolationsfilter ist dadurch limitiert auf –80 dB (rechter Teil von Abbildung 54).

Die Algorithmus-bedingte Audioqualität der WOLA-Filterbank ist höchst konfigurationsabhängig. Dank der makroprogrammierbaren DSP-Architektur können über 500 verschiedene WOLA-Konfigurationen verarbeitet werden.

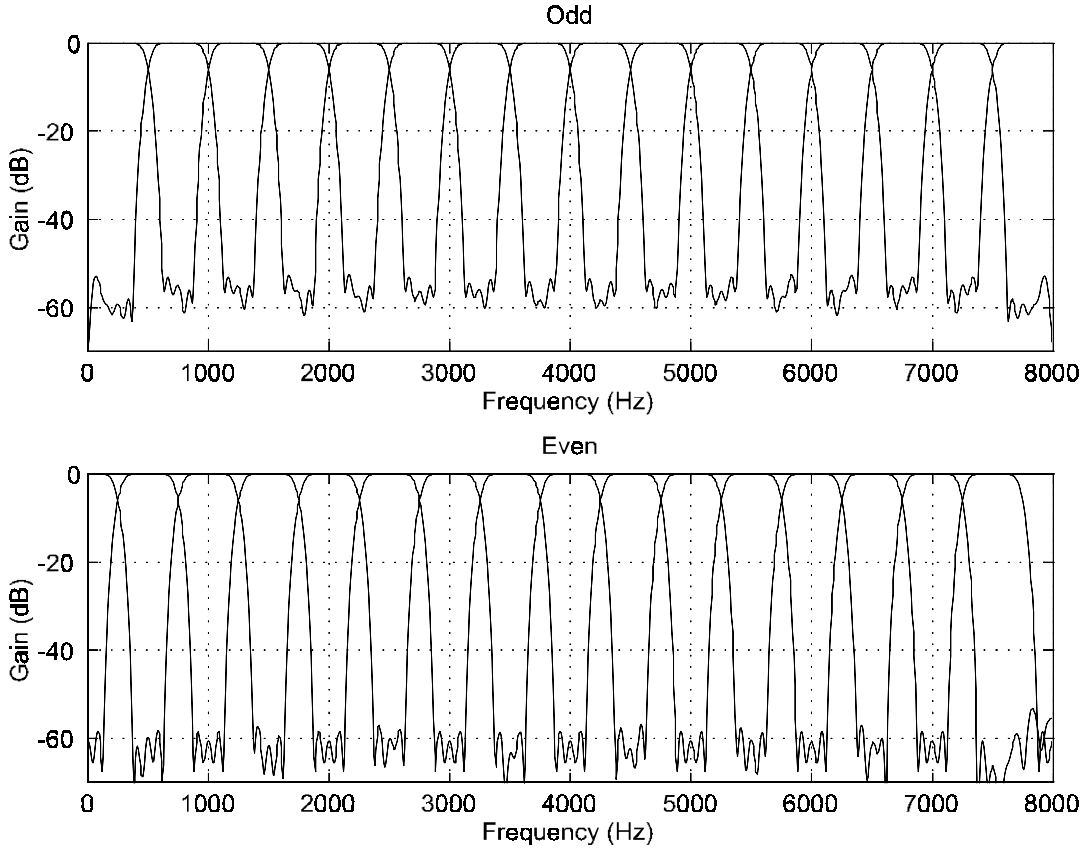


Abbildung 55: Kanaltrennung der WOLA-Filterbank in Odd- *und* Even-Stacking-Modus[1] *mit folgender Konfiguration:* FFT Size *(komplex): 16,* Oversampling*: 4,* Window Size*: 256*

[1] *Odd Stacking*: Eine Modulation der eingehenden Samples führt zu einer Verschiebung der Frequenzbänder um eine halbe Kanalbreite. Diese Arbeitsweise der implementierten Filterbank wird *odd stacking* genannt. Werden die eingehenden Samples nicht moduliert, spricht man von *even stacking*.

Stellvertretend für all die Konfigurationen ist in Abbildung 55 die Filtercharakteristik einer der meist verwendeten WOLA-Konfigurationen abgebildet, und zwar im *Even*- und *Odd-Stacking-Modus* (siehe Kapitel 4.4.3.1: Der WOLA-Algorithmus). Sehr schön ist dabei die Verschiebung der Frequenzbänder im *Odd-Stacking-Modus* um eine halbe Bandbreite bezüglich dem *Even-Stacking-Modus* zu sehen. Die Kanaltrennung im *Even*- und *Odd-Stacking-Modus* ist 60 dB bzw. 55 dB.

Durch die Verwendung von 19-Bit-Daten und durch mathematisch richtiges Runden der Daten wurde erreicht, dass das Quantisierungsrauschen der Vor- und Nachfilter zusammen lediglich –115 dB ist.

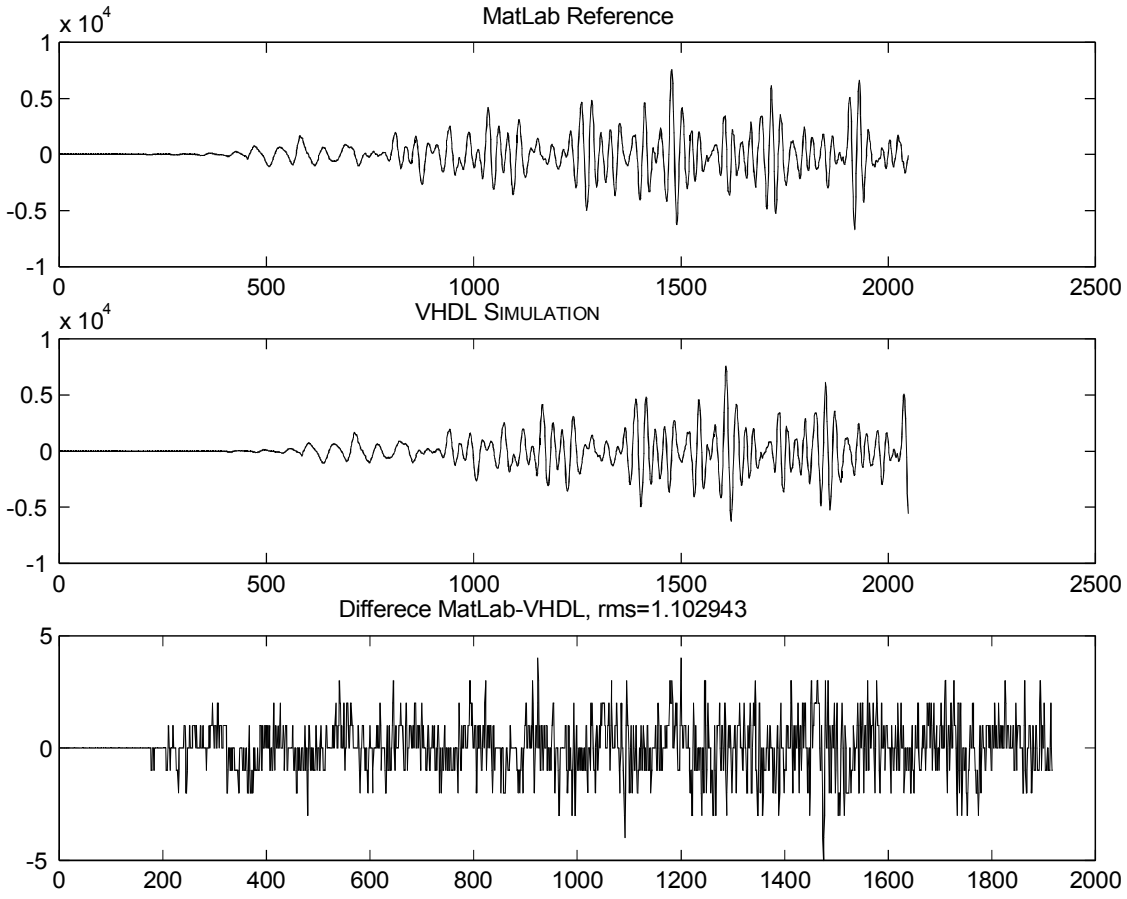


Abbildung 56: Matlab- und VHDL-Simulation des Koprozessors, welcher in einer 16-Kanal-WOLA-Filterbank konfiguriert ist. Der unterste Abbildungsdrittel zeigt die Differenz beider Simulationen, wobei die Dynamik 16 Bit beträgt.

Ebenfalls ein sauberes Runden, vor allem aber die Verwendung der Block-Gleitkommaarithmetik (*BFP Arithmetic*) reduzierte den Quantisierungsfehler der WOLA-Filterbank ebenfalls aufs Minimum. Der Fehler ist stark abhängig von der WOLA-Konfiguration und beträgt bei einer üblichen 16-Kanal-Konfiguration noch mehr als 100 dB. Abbildung 56 zeigt ein vom Koprozessor verarbeitetes Audiosignal, welches ein erstes Mal als Referenz von Matlab und ein zweites mal mittels einer VHDL-

Simulation erzeugt wurde. Es wurde dabei eine 16-Kanal-WOLA-Filterbank verwendet, dessen Fensterlänge 256 Punkte hatte. Die ebenfalls veranschaulichte Quantisierungsdifferenz beträgt im Mittel 1.103 bei einer 16-Bit-Dynamik, was einer Auflösung von 14.9 Bit entspricht.

Resultate der ASIC-Integrationen

Bedingt durch die verschiedenen verwendeten ASIC-Technologien und Implementierungskonfigurationen (mehr oder weniger RAM, mit oder ohne DSP, ...) ergaben die verschiedenen Integrationen auch unterschiedliche Resultate.

Tabelle 10 zeigt den Hardwareaufwand der verschiedenen Blöcke der digitalen Signal-Verarbeitungseinheit, welche in seiner Konfiguration dem in Abbildung 50 gezeigten Chip entspricht. Der DSP-Kern hat eine doppelte Harvard-Architektur. Bis auf wenige Ausnahmen werden alle Befehle in einem einzigen Takt durchgeführt, ebenso ein Speicherzugriff. Nebst dem Kern enthält das DSP-Subsystem periphere Einheiten wie serielle Standard- und *High-Speed-Interfaces*, ein paralleles Interface, *Watchdoc-Timer*, *Interrupt*-Einheit, usw.

Name des Blocks	*Grösse (GE)*	*Kommentar*
Vor-/ Nachfilter	*8597*	*Hier verwendet: Handimplementierung*
SAI	*1050*	
IOP	9797	Vor- und Nachfilter + SAI + Glue-Logic
FFT-Prozessor	18753	Siehe Kapitel 5
Total Koprozessor	***29217***	***IOP + FFT-Prozessor + Glue-Logic***
DSP-Kern	15010	
DSP-Subsystem	***20607***	***DSP-Kern + periphere Einheiten***
Total	**52049**	

Tabelle 10: Die Grössen der verschiedenen Unterblöcke des gesamten Verarbeitungsteils für digitale Hörgeräte (Referenztechnologie: Alcatel-Mietec 0.35 um CMOS).

Bezüglich Leistungsaufnahme ergab die 0.18µm-CMOS-Integration dank der fortschrittlichen Technologie und der speziell für tiefe Spannungen konzipierten RAM und Standardzellenbibliotheken die spektakulärsten Resultate und soll deshalb stellvertretend für alle Integrationen kurz präsentiert werden.

Bereich der Speisespannung			0.9–2.7 V
Maximale Taktrate (Systemclock-Frequenz)	0.9 V		10 MHz
	1.2 V		20 MHz
	1.8 V		50 MHz
Stand-By- Stromverbrauch (1.2 V, 1.34 MHz)			48 µA
Stromverbrauch für 8-Kanal-WOLA-Filterbank – Konfigurationen (N_{FFT}=16) U=1.2 V, f=1.34 MHz	EVEN	Typisch[1]	121 µA
		Minimal	99 µA
		Maximal	150 µA
	ODD	Typisch	132 µA
		Minimal	106 µA
		Maximal	162 µA
Stromverbrauch für 16-Kanal-WOLA-Filterbank – Konfigurationen (N_{FFT}=32) U=1.2 V, f=1.34 MHz	EVEN	Typisch	128 µA
		Minimal	94 µA
		Maximal	185 µA
	ODD	Typisch	143 µA
		Minimal	102 µA
		Maximal	215 µA
Stromverbrauch für 32-Kanal-WOLA-FB – Konfig., (N_{FFT}=64, U=1.2 V, f=1.34 MHz)	EVEN	Typisch	94 µA
	ODD	Typisch	100 µA
Stromverbrauch für 64-Kanal-WOLA-FB - Konfig, (N_{FFT}=128, U=1.2 V, f=1.34 MHz)	EVEN	Typisch	94 µA
	ODD	Typisch	100 µA
Stromverbrauch für eine 128-Kanal-WOLA-Filterbank, (N_{FFT}=128, U=1.2 V, f=1.34 MHz)	EVEN	Typisch	120 µA
Verringerung des Stromverbrauches bei Stille (Eingangssignal=0)			~ - 5 %
Statistische Abweichungen	*Stand-By*- Strom		9 %
	Totalstrom		10 %

Tabelle 11: Stromverbrauch der Digitaleinheit bei verschiedenen Operationen (Referenztechnologie: CMOS 0.18 µm, 6 Metall)

7.4 Schlussfolgerungen

Die Sorgfalt, mit welcher der Entwurf des Digitalteils für Hörgeräte und anschliessend das Backend durchgeführt wurde, brachte zwar einen grossen zeitlichen Aufwand mit sich, garantierte aber schlussendlich, dass die gestellten Randbedingungen und Spezifikationen vollständig erfüllt werden konnten. Das Quantisierungsrauschen des Gesamtsystems ist je nach gewählter Konfiguration etwa 90 dB, was einer 15-Bit-Auflösung entspricht und im Vergleich mit den algorithmenbedingten Aliasingeffekten vernachlässigbar gering ist. Die 0.18 µm –Integration ergab aufgrund der fortschrittlichen Technologie die spektakulärsten Resultate: Die Chipgrösse

[1] Mit *typisch* ist eine typische WOLA-Konfiguration gemeint, welche audiologisch verwendbar ist.

beträgt etwa 10 mm^2, der Stromverbrauch liegt zwischen 95 µA und 215 µA bei einer Speisespannung von 1.2 Volt und die Speisespannung darf bis auf 0.9 Volt absinken, ohne dass die Funktionsweise gestört wird.

Die 0.18 µm –Version des Digitalteils (die Integration wurde nicht am IMT durchgeführt) nennt sich Delta-2 und wird unterdessen kommerziell vertrieben [1]. Nebst als *Naked Chip* kann er auch unter dem Namen *Toccata Hybrid* als Fertiglösung für Hörgeräte gekauft werden (Abbildung 57). Der *Toccata Hybrid* wurde speziell für Hörgeräte entwickelt (inklusive CIC) und beinhaltet nebst dem Digitalteil ein EEPROM und einen Chip mit AD- und DA-Wandler, *Charge Pump*, Verstärker, System-Clock, usw.

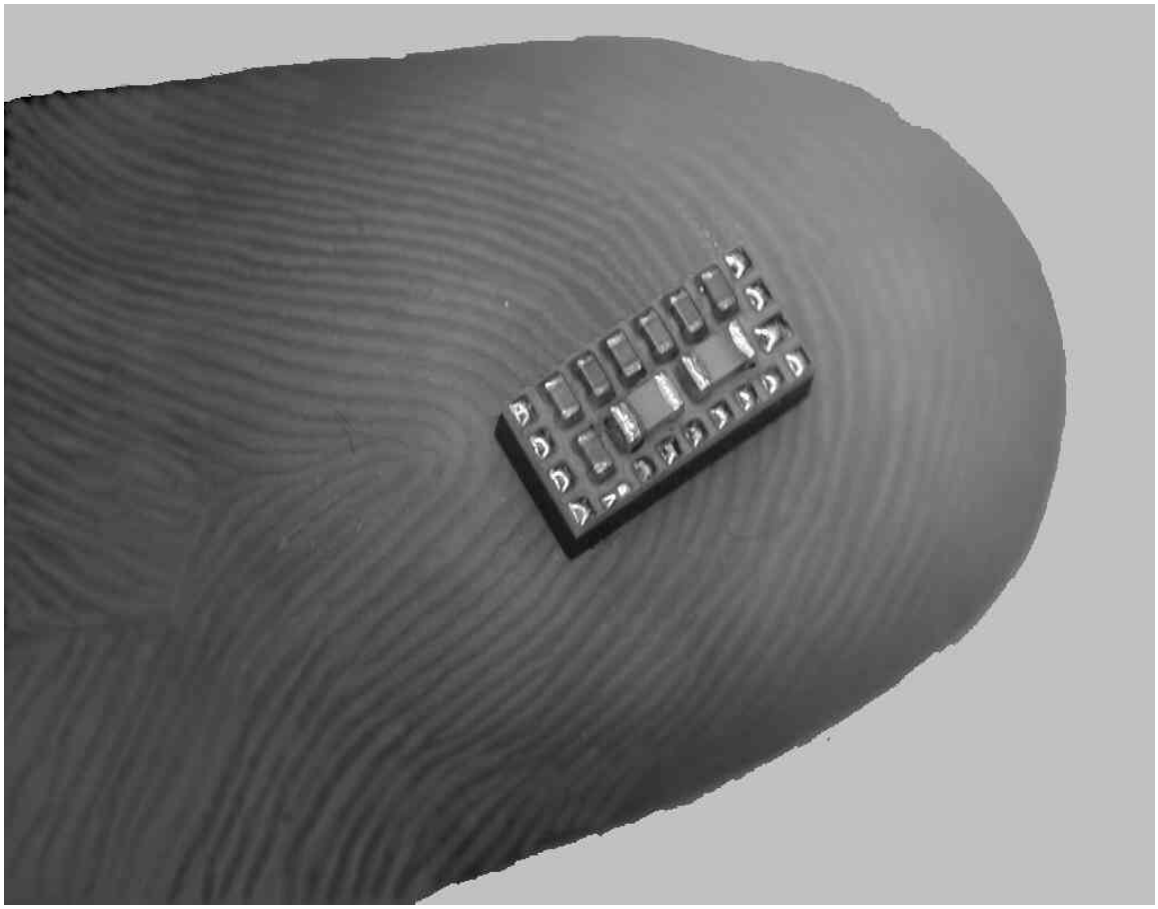


Abbildung 57: Der Toccata Hybrid *schliesst in einigen Kubikmillimetern die digitale Verarbeitungseinheit, ein Chip mit ADDA-Wandlern und ein EEPROM ein.*

Referenzen

[1] dspfactory, 80 King St. South, Suite 206, Waterloo, Ontario. Canada. N2J 1P5. http://www.dspfactory.com

8 Schlussfolgerung

Dieses letzte Kapitel enthält einige Kapitel-übergreifende Gedanken bezüglich den präsentierten Designmethoden.

Erst an Hand einer anspruchsvollen Designaufgabe kann die Wichtigkeit von Lösungsansätzen und Designmethoden erkannt werden. Auch das DSP-Koprozessor-Projekt wurde zu Beginn mit falschen Ansätzen gestartet und musste von Grund auf neu analysiert werden. Die benötigte Rechenleistung zur Verarbeitung der Audiosignale war nicht das eigentliche Problem. Schliesslich können viele der verfügbaren *General Purpose DSP* diese Leistung liefern. Die Randbedingungen bezüglich Leistungsaufnahme, Speisespannung und Siliziumfläche machten die Entwicklung bereits viel schwieriger. Aber auch hier ist es lediglich eine Frage der Zeit, bis eine DSP-Architektur so beschrieben und für die Designaufgabe optimiert ist, bis sie das Pflichtenheft befriedigen kann. Der wirklich schwierigst zu erfüllende Projektfaktor ist aber die Limitierung der Entwicklungszeit. Erst dadurch wird es zur absoluten Notwendigkeit, die Designaufgabe klar zu analysieren und die zu verwendenden Lösungsansätze und Designmethoden sorgfältig zu wählen.

In dieser Arbeit wurden drei Lösungsansätze und Designmethoden für den digitalen Schaltungsentwurf vorgestellt, welche in den Gebieten DSP-Architekturen, Designmethodik und Automatisierung neue, innovative Lösungen bilden. Jeder der Ansätze beschreitet einen eigenen Weg und hat seine Vor- und Nachteile. Gemeinsam erlauben sie aber im Gegensatz zu anderen, kommerziellen Methoden, Qualitätskriterien eines Designs besser zu gewichten.

Die erste Designmethode profitiert von der Einfachheit einer Implementierung mittels eines frei programmierbaren DSPs, passt die DSP-Architektur jedoch in einer Optimierungsphase möglichst optimal an eine Zielanwendung an.

Als zweite Lösung wurde die Studie und Realisation eines neuen *high-level* DSP-Synthesetool zur Generierung von anwendungsspezifischen DSP-Architekturen präsentiert. Das Tool zeichnet sich aus durch Einfachheit, Schnelligkeit, Flexibilität, Sicherheit und Qualität der generierten Architekturen.

Der dritte Ansatz ist ein Makrocode-Konzept, welches ermöglicht, so von einer Algorithmusregelmässigkeit zu profitieren, dass im Gegensatz zu einer Lösung mit einem frei programmierbaren DSP die Verarbeitungsgeschwindigkeit erhöht und die Leistungsaufnahme reduziert wird.

Die Motivation zur Studie und Realisierung der drei präsentierten Lösungsansätze wurde zwar primär beeinflusst von der Entwicklung der digitalen Verarbeitungseinheit für Hörgeräte. Die Ansätze wurden jedoch

gezielt in einer Weise entwickelt und dargelegt, dass sie allgemein verwendet werden können. Sie können deshalb auch in anderen Projekten als mögliche Lösungswege in Betracht gezogen werden.

Die Lösungsansätze ergänzen sich nicht nur, sondern besitzen auch gemeinsame Anwendungsbereiche. So können die Vor- und Nachfilter des Koprozessors mit Hilfe des DSP-Synthesetools, aber auch mit der Co-Design-Methodik und dem HotDSP rasch und einfach implementiert werden. Richtig verwendet garantieren die Ansätze eine erfolgreiche Realisation des DSP-Koprozessors nicht nur in Hinblick des Datendurchsatzes und der physikalischen Randbedingungen, sondern auch bezüglich der Entwicklungszeit.

Selbst wenn bei der Ausarbeitung der Lösungsansätze auf eine allgemeine Verwendung geschaut wurde, muss doch unterstrichen werden, dass ihr Einsatzbereich auf die digitale Signalverarbeitung beschränkt bleibt. Sie können und sollen niemals andere, allgemein verwendbare Designmethoden, wie etwa HDL-Beschreibungen, konkurrenzieren. Im Gegenteil, die Ansätze basieren ja selbst auf VHDL und zeigen, wie der Architekturentwurf mittels einer HDL-Beschreibungssprache effizienter gestaltet werden kann. Und hier ist die Stärke der präsentierten Lösungsansätzen. Sie versuchen nicht, bestehende Methoden zu konkurrenzieren, sondern zu ergänzen. Dadurch ist es möglich, die Ansätze gezielt zur Implementierung von Designteilen zu verwenden. Die Verwendung eines Ansatzes für einen Designteil blockiert nicht die Möglichkeit, für einen anderen Designteil einen anderen Ansatz zu verwenden. Im Falle des vorgestellten DSP-Koprozessors können zum Beispiel die Vor- und Nachfilter mit Hilfe der parametrisierbaren DSP-Struktur realisiert werden, der FFT-Prozessor mit der makroprogrammierbaren DSP-Architektur und die Datenverarbeitungseinheit mit dem *high-level* Synthesetool.

Die präsentierten Lösungsansätze konkurrenzieren im besten Fall äquivalente, kommerzielle Designmethoden. Damit die Vorteile der Ansätze gegenüber kommerziellen Produkten erhalten bleiben, müssen auch sie ständig weiterentwickelt werden, damit sie neuen Designansprüchen Rechnung tragen können.

Dank

Die Durchführung dieser Dissertation war nur möglich dank der grosszügigen Unterstützung, welche ich von vielen Seiten erfahren durfte.

Als erstes möchte ich mich ganz herzlich Professor Fausto Pellandini, Leiter der Gruppe für Elektronik und Signalverarbeitung des IMT Neuchâtel, für seine Unterstützung und seinen Rat bedanken. Ich schätzte die Zeit sehr, an der ich in seiner Gruppe am IMT arbeiten durfte.

Einen grossen Dank möchte ich ebenfalls der Jury dieser Dissertation, Dr. Hubert Kaeslin, Leiter des Mikroelektronik-Designzentrum der ETH Zürich, Dr. Peter Balsiger, Direktor der dspfactory SA, und Professor Heinz Hügli, Leiter der Gruppe für Formen- und Strukturenerkennung des IMT Neuchâtel, übergeben, und zwar dafür, dass sie sich als Jurymitglied zur Verfügung stellten und für die vielen konstruktiven Bemerkungen und Anregungen während der Dissertationszeit.

Einen speziellen Dank geht an meinen direkten Chef, Dr. Peter Balsiger, welcher mir viel Freiheit liess, so dass ich meine Ideen, Konzepte und schliesslich auch diese Arbeit verwirklichen konnte. Er unterstützte mich in meinen Arbeiten mit einem unglaublichem Enthusiasmus.

Ebenfalls einen herzlichen Dank an Dr. Hubert Kaeslin, welcher die Dissertationsskripte verschiedene Male sorgfältigst durcharbeitete und mir mit vielen Ratschlägen half, dieser Arbeit den letzten Feinschliff zu geben.

Dann möchte ich mich natürlich bei all meinen Kollegen des IMTs für die ständig angenehme Arbeitsatmosphäre und die immer wiederkehrenden, interessanten Gespräche bedanken.

Und schlussendlich danke ich meiner Familie für ihre grosse Unterstützung, und im speziellen meiner Mutter für die sorgfältige Durchlese dieser Arbeit und meiner Frau, Marie, für ihre motivierenden Impulse speziell während der Niederschrift, also der technisch uninteressantesten Phase dieser Arbeit.

Andreas Drollinger

Juni 2002

Anhang I (Listings und Ergänzungen)

Parameter des HotDSPs (Ergänzung zum Kapitel 3)

Der in Kapitel 3.5, auf Seite 40 vorgestellte HotDSP hat folgende Parametrisiermöglichkeiten:

- Parameter auf Systemniveau:
  - Definition, ob der HotDSP voll synchron getaktet werden soll oder *Gated Clocks* verwenden soll.
  - Definition, in welchen Untereinheiten *Gated Clocks* verwendet werden sollen.
  - Anzahl der –Event-Signale, der Input- und Output-Ports sowie die Busbreite der Ports.
- Parameter der Kontrolleinheit:
  - Anzahl Daten- und Adressbits für die Instruktionen sowie die Grösse des Instruktions-LUT.
  - Anzahl der Niveaus für den Adressstack.
  - Welche Event-Signale den HotDSP aufwecken können und sollen.
  - Ob und wie die Event-Signale registriert oder verzögert werden sollen.
  - Ob der HotDSP bei einem aktivierten Stopbit einer Instruktion vor oder nach deren Verarbeitung angehalten werden soll.
  - Definition der möglichen Verzweigungsbedingungen und deren Binärcodierungen.
- Parameter des Registerblockes:
  - Allgemeine Breite der Busse für Daten und Koeffizienten
  - Grösse des Koeffizienten-LUT
  - Anzahl Register und wie die Output-Ports realisiert werden sollen (gepuffert/ungepuffert, Alias-Port).
  - Für alle Register, Akkumulatoren, Input-, Output- und Alias-Ports soweit benötigt: Die Busbreiten, die logischen Adressen, die verwendeten *Gated Clock* und gewisse Informationen für das Datencasting[1] bei unterschiedlichen Busbreiten.
- Parameter der arithmetischen Einheit:
  - Existenz eines registrierten Carry-Flags.

[1] Übergang von einem ersten Datenformat in ein zweites

- Spezifikationen der möglichen Nachbearbeitungen der Daten (Runden, Sättigung, Shiften).
- Datenformat aller Zwischenresultate innerhalb der arithmetischen Einheit.
- Parameter für die Instruktionscodierung.
- Definition der Breiten aller Bitfelder.
- Zuweisung der Binärcodes an alle Instruktionen.
- Definition aller möglichen arithmetischen Operationen und Zuweisungen der Binärcodes an diese Operationen.

All Parameter sind beim HotDSP-Typ 1 in einem, bei den anderen beiden Typen in zwei VHDL-Paketen definiert. Folgendes Listing zeigt das VHDL-Definitionspaket für den Typ 1, wobei die Parameterkonfiguration derjenigen entspricht, welche in Kapitel 0 zur Realisierung des Stereo-Interpollationsfilter (siehe Listing 5) verwendet wurde.

Listing 9: Folgende Seiten: VHDL-Definitionspaket für den HotDSP-Typ 1

```
LIBRARY IEEE;
    USE IEEE.std_logic_1164.all;
    USE IEEE.std_logic_arith.all;

PACKAGE hdsp_def IS

    ---- definitions of some usefull types ----

    TYPE integer_vector IS ARRAY( NATURAL RANGE <>) OF integer;

    ---- interface definitions ----

    CONSTANT System_Is_Synchron: std_logic := '0';
    CONSTANT Control_UseGatedClock: std_logic := '0';
    CONSTANT AU_UseGatedClock: std_logic := '1';
    CONSTANT Reg_UseGatedClock: std_logic := '1';

    CONSTANT InstrWidth: integer := 33;
    CONSTANT InstrAddrWidth: integer := 4;
    CONSTANT InstrRomSize: integer := 32;
    CONSTANT NbrInEvent: integer := 5;
    CONSTANT NbrAuxOutEvent: integer := 0;

    CONSTANT BusWidth: integer := 19;
    CONSTANT CstWidth: integer := 9;
    CONSTANT CstAddrWidth: integer := 2;
    CONSTANT ConstRomSize: integer := 4;
    CONSTANT RegAddrWidth: integer := 5;
    CONSTANT InPortWidth: integer := 16;
    CONSTANT OutPortWidth: integer := 18;
    CONSTANT RegOffsetWidth: integer := 3;
```

```
---- control unit definitions ----

CONSTANT NbrAddrStackLevel: integer := 0;
CONSTANT EventWakeUpEnable: std_logic_vector := "10000";

---- register unit definitions ----

CONSTANT NbrRegister: integer := 8;
CONSTANT NbrInPortRegister: integer := 1;
CONSTANT NbrOutPortRegister: integer := 0;
CONSTANT NbrOutRegPortRegister: integer := 1;
CONSTANT NbrOutPortAddressAlias: integer := 1;
CONSTANT NbrRegisterClock: integer := 3;

CONSTANT Accu0_Addr: integer := 0;
CONSTANT Accu0_Width: integer := BusWidth;
CONSTANT Accu0_Rd0_Used: std_logic := '1';
CONSTANT Accu0_Rd1_Used: std_logic := '1';

CONSTANT Accu1_Addr: integer := 1;
CONSTANT Accu1_Width: integer := BusWidth;
CONSTANT Accu1_RegClkNbr: integer := 0;
CONSTANT Accu1_ReservedClock: std_logic := '1';
CONSTANT Accu1_Rd0_Used: std_logic := '1';
CONSTANT Accu1_Rd1_Used: std_logic := '1';

CONSTANT RegisterArray_Addr: integer_vector := (2,3,4,5,6,7,8,9);
CONSTANT RegisterArray_Width: integer_vector := (19,19,19,19,19,19,19,19);
CONSTANT RegisterArray_RegClkNbr: integer_vector := (1,1,1,1,2,2,2,2);
CONSTANT RegisterArray_ReservedClock: std_logic_vector := "00000000";
CONSTANT RegisterArray_BitAjust_Rd: integer_vector := (0,0,0,0,0,0,0,0);
CONSTANT RegisterArray_BitAjust_Wr: integer_vector := (0,0,0,0,0,0,0,0);
CONSTANT RegisterArray_MsbExtention_Rd: integer_vector := (0,0,0,0,0,1,0,0);
CONSTANT RegisterArray_Rd0_Used: std_logic_vector := "01010101";
```

```
CONSTANT RegisterArray_Rd1_Used: std_logic_vector := "11111111";

CONSTANT InPortRegisterArray_Addr: integer_vector := (15,-1);
CONSTANT InPortRegisterArray_Width: integer_vector := (16,-1);
CONSTANT InPortRegisterArray_BitAjust_Rd: integer_vector := (2,-1);    -- Left Shift
CONSTANT InPortRegisterArray_Rd0_Used: std_logic_vector := "1";
CONSTANT InPortRegisterArray_Rd1_Used: std_logic_vector := "0";

CONSTANT OutPortRegisterArray_Addr: integer_vector := (-1,-1);
CONSTANT OutPortRegisterArray_Width: integer_vector := (-1,-1);
CONSTANT OutPortRegisterArray_BitAjust_Wr: integer_vector := (-1,-1);

CONSTANT OutRegPortRegisterArray_Addr: integer_vector := (14,-1);
CONSTANT OutRegPortRegisterArray_Width: integer_vector := (18,-1);
CONSTANT OutRegPortRegisterArray_RegClkNbr: integer_vector := (2,-1);
CONSTANT OutRegPortRegisterArray_ReservedClock: std_logic_vector := "0";
CONSTANT OutRegPortRegisterArray_BitAjust_Wr: integer_vector := (0,-1);

CONSTANT OutPortAddressAlias_Code: integer_vector := (13,-1);
CONSTANT OutPortAddressAlias_Correct: integer_vector := (14,-1);
CONSTANT OutPortAddressAlias_RegisteredOutPort: integer_vector := (1,-1);

TYPE t_InPort IS ARRAY(0 TO NbrInPortRegister-1) OF std_logic_vector(InPortWidth-1 DOWNTO 0);
TYPE t_OutPort IS ARRAY(0 TO NbrOutPortRegister+NbrOutRegPortRegister-1) OF std_logic_vector(OutPort …

---- Event processing ----

CONSTANT Stop_IsAfterInstruction: integer := 0;                    -- 1: After Instruction
CONSTANT InEventDelayed: integer_vector := (0,2,2,0,0);        -- 0:no Delay,1 Delayed by 1 Clk cycle,2 …
CONSTANT NbrOutEvent: integer := NbrOutPortRegister+NbrOutRegPortRegister+NbrOutPortAddressAlias+ …
```

---- Arithmetic Unit, output processing----

```
CONSTANT AU_WithCarry: integer := 1;    -- 0:no Carry, 1:Carry
CONSTANT AU_CarryPos: integer :=BusWidth-1;

CONSTANT AU_NbrShiftR: integer := 2;
CONSTANT AU_NbrRound: integer := 5;
CONSTANT AU_NbrSat: integer := 2;
CONSTANT AU_OutShiftR_Size: integer := 1;
CONSTANT AU_OutRound_Size: integer := 3;
CONSTANT AU_OutSat_Size: integer := 1;

CONSTANT AU_ShiftR: integer_vector := (0,2);        -- AU_ShiftR(0)=default
CONSTANT AU_Round: integer_vector := (0,1,0,0,0);              -- AU_Round(0)=default   0:round 1:SMT,2:trunc
CONSTANT AU_Round_ZeroForcedLSB: integer_vector := (0,0,2,3,4);    -- AU_ZeroForcedLSB=default
CONSTANT AU_Sat: integer_vector := (19,18);         -- AU_Sat(0)=default, 0=no saturation
CONSTANT AU_Round_MaxZeroForcedLSB: integer := 4;

CONSTANT AU_ResWidth: integer := 20;
CONSTANT AU_SupResWidth : integer :=8;
CONSTANT AU_ShiftWidth: integer := 20;
CONSTANT AU_RoundWidth: integer := AU_ShiftWidth+1;
```

---- instruction field definitions, have to be configured by the user----

```
CONSTANT InstrBranch: std_logic_vector := "00";
CONSTANT InstrBranch_Type_Size: integer := 1;                   -- 1=call 0=jump
CONSTANT InstrBranch_AU_PipeStop_Size: integer := 1;            -- 1:AU_Pipe stopped
CONSTANT InstrBranch_DoubleBranch_Size: integer := 1;
CONSTANT InstrBranch_Cond_Size: integer := 1;
CONSTANT InstrBranch_Addr0_Size: integer := InstrAddrWidth;
CONSTANT InstrBranch_RegOffs0_Size: integer := RegOffsetWidth;
CONSTANT InstrBranch_Addr1_Size: integer := InstrAddrWidth;
CONSTANT InstrBranch_RegOffs1_Size: integer := RegOffsetWidth;
```

```
CONSTANT InstrBranch_Addr2_Size: integer := InstrAddrWidth;
CONSTANT InstrBranch_RegOffs2_Size: integer := RegOffsetWidth;
CONSTANT InstrBranch_RegOffs3_Size: integer := RegOffsetWidth;

CONSTANT InstrArith: std_logic_vector := "1";
CONSTANT InstrArith_Type_Size: integer := 3;
CONSTANT InstrArith_Out_Size: integer := AU_OutShiftR_Size+AU_OutRound_Size+AU_OutSat_Size;
CONSTANT InstrArith_Cst_Size: integer := CstAddrWidth+1;
CONSTANT InstrArith_Reg1_Size: integer := RegAddrWidth+1;
CONSTANT InstrArith_Reg2_Size: integer := RegAddrWidth+1;
CONSTANT InstrArith_Dest_Size: integer := RegAddrWidth+1;

CONSTANT InstrNop: std_logic_vector := "01";

--- instruction field definitions, only used for the assembler ----

CONSTANT BranchCond_Event12: std_logic_vector := "0";
CONSTANT BranchCond_Event34: std_logic_vector := "1";
CONSTANT ArithType_add: std_logic_vector := "000";
CONSTANT ArithType_sub: std_logic_vector := "001";
CONSTANT ArithType_incc: std_logic_vector := "010";
CONSTANT ArithType_mov: std_logic_vector := "100";
CONSTANT ArithType_mulsub: std_logic_vector := "101";

--- instruction field definitions, haven't to be adapted by the user----

CONSTANT InstrJump0Bit_Pos: integer := InstrWidth-1;
CONSTANT InstrStopBit_Pos: integer := InstrJump0Bit_Pos-1;
CONSTANT InstrRetBit_Pos: integer := InstrStopBit_Pos-1;

CONSTANT InstrBranch_Pos: integer := InstrRetBit_Pos-InstrBranch'length;
CONSTANT InstrBranch_Type_Pos: integer := InstrBranch_Pos-InstrBranch_Type_Size;
CONSTANT InstrBranch_AU_PipeStop_Pos: integer := InstrBranch_Type_Pos-InstrBranch_AU_PipeStop_Size;
CONSTANT InstrBranch_Cond_Pos: integer := InstrBranch_AU_PipeStop_Pos-InstrBranch_Cond_Size;
```

```
    CONSTANT InstrBranch_Addr0_Pos: integer := InstrBranch_Cond_Pos-InstrBranch_Addr0_Size;
    CONSTANT InstrBranch_RegOffs0_Pos: integer := InstrBranch_Addr0_Pos-InstrBranch_RegOffs0_Size;
    CONSTANT InstrBranch_Addr1_Pos: integer := InstrBranch_RegOffs0_Pos-InstrBranch_Addr1_Size;
    CONSTANT InstrBranch_RegOffs1_Pos: integer := InstrBranch_Addr1_Pos-InstrBranch_RegOffs1_Size;
    CONSTANT InstrBranch_Addr2_Pos: integer := InstrBranch_RegOffs1_Pos-InstrBranch_Addr2_Size;
    CONSTANT InstrBranch_RegOffs2_Pos: integer := InstrBranch_Addr2_Pos-InstrBranch_RegOffs2_Size;
    CONSTANT InstrBranch_RegOffs3_Pos: integer := InstrBranch_RegOffs2_Pos-InstrBranch_RegOffs3_Size;


    CONSTANT InstrArith_Pos: integer := InstrRetBit_Pos-InstrArith'length;
    CONSTANT InstrArith_Type_Pos: integer := InstrArith_Pos-InstrArith_Type_Size;
    CONSTANT InstrArith_Out_Pos: integer := InstrArith_Type_Pos-InstrArith_Out_Size;
    CONSTANT InstrArith_Cst_Pos: integer := InstrArith_Out_Pos-InstrArith_Cst_Size;
    CONSTANT InstrArith_Reg1_Pos: integer := InstrArith_Cst_Pos-InstrArith_Reg1_Size;
    CONSTANT InstrArith_Reg2_Pos: integer := InstrArith_Reg1_Pos-InstrArith_Reg2_Size;
    CONSTANT InstrArith_Dest_Pos: integer := InstrArith_Reg2_Pos-InstrArith_Dest_Size;


    CONSTANT InstrNop_Pos: integer := InstrRetBit_Pos-InstrNop'length;
END hdsp_def;
```

Pseudo-Code einiger Routinen, welche für die Festlegung der Gesamtdatenwortbreiten innerhalb des high-level Synthesetools verwendet werden (Ergänzung zum Kapitel 4):

Siehe „Festlegung der Gesamtdatenwortweiten“, Seite 84

Listing 10: Nächste Seiten: Pseudocode einiger Rutienen zur Quantisierung

```
Get_OneChannelNoise(MoNr,ChNr)
-- Berechnet den Quantisierungsfehler eines Ausgangsignales
   Noise2=0
   for i:=1 to ResultLength[MoNr][ChNr]
     Noise2+=(SimulValue[MoNr][ChNr][i]-ReferenceValue[MoNr][ChNr][i])2
   ChannelNoise[MoNr][ChNr]=sqrt(Noise2)/ResultLength[MoNr][ChNr]
   return ChannelNoise[MoNr][ChNr]

Get_MeanChannelNoise()
-- Berechnet den Durchschnitt der Quantisierungsfehler
-- der Ausgangsignale aller Funktionsmoden
   TotNoise=0
   TotNbrChan=0
   for MoNr:=1 to NbrMode
      for ChNr:=1 to NbrChannel[MoNr]
         TotNoise+=Get_OneChannelNoise(MoNr,ChNr)
         TotNbrChan++
   TotNoise/=TotNbrChan
   return TotNoise

Get_TotCost()
-- Berechnet die Kosten der aktuellen Konfiguration
-- inklusive der Strafkosten, korrigiert den Straffaktor
   TotCost=Get_ResourceCost()
   PanaltyFactor_OK=PanaltyFactor_KO=PanaltyFactor;
   NbrViolation=0
   for MoNr:=1 to NbrMode
      for ChNr:=1 to NbrChannel[MoNr]
        if ChannelNoise[MoNr][ChNr]>MaxAllowedChannelNoise[MoNr][ChNr]
           TotCost-=PanaltyFactor*(ln(ChannelNoise[MoNr][ChNr])-
                    ln(MaxAllowedChannelNoise[MoNr][ChNr]))
           NbrViolation++
           PanaltyFactor_KO*=(1.0+IntegrationFactor)
        else
```

```
        PanaltyFactor_OK*=(1.0+IntegrationFactor)
    if NbrViolation=0
       PanaltyFactor=PanaltyFactor_OK
    else
       PanaltyFactor=PanaltyFactor_KO
    return TotCost

Def_PenaltyFactor()
-- Berechnet den Initial-Straffaktor
    nTotBit_1=round(0.0-ln(MeanChannelNoise)/ln(2) )
    DefineAllWortLength(nTotBit_1)
    TotSimulLongFracBeh();
    Noise_1=Get_MeanChannelNoise();
    Cost_1=Get_TotCost()

    nTotBit_2=round(3.0-ln(MeanChannelNoise)/ln(2) )
    DefineAllWortLength(nTotBit_2)
    TotSimulLongFracBeh();
    Noise_2=Get_MeanChannelNoise();
    Cost_2=Get_TotCost()

    PanaltyFactor=2.0*(Cost_1-Cost_2)/(ln(Noise_1)-ln(Noise_2));
```

Datenpfad-Beschreibung für FFT-Prozessor (Ergänz. zum Kapitel 4):

Listing 11 enthält die komplette Beschreibung der FFT-Prozessor-Verarbeitungseinheit, welche in Kapitel 4 als Anwendungsbeispiel verwendet wurde. Nebst den reellen und komplexen Grundoperationen wie Addition und Multiplikation werden auch die Radix-2- und die Radix-4-Operationen definiert.

```
function [r,i]=RM(r0,i0,r1,i1)
    r=r0*r1
    i=i0*i1
function [r,i]=RMAC(r0,i0,r1,i1)
    r=r0@1+r0*r1
    i=i0@1+i0*i1
function [r,i]=CM(r0,i0,r1,i1)
    r=r0*r1-i0*i1
    i=r0*i1+r1*i0
function [r0o,i0o,r1o,i1o]=radix2(r0i,i0i,r1i,i1i,rc,ic)
    [r2i,i2i]=CM(r1i,i1i,rc,ic)
    r0o=r0i+r2i
    i0o=i0i+i2i
    r1o=r0i-r2i
    i1o=i0i-i2i
function [r0o,i0o,r1o,i1o,r2o,i2o,r3o,i3o]=radix4n(r0i,i0i,r1i,i1i,r2i,i2i, r3i,i3i)
    [r0m,i0m,r1m,i1m]=radix2n(r0i,i0i,r1i,i1i)
    [r2m,i2m,r3m,i3m]=radix2n(r2i,i2i,r3i,i3i)
    [r0o,i0o,r2o,i2o]=radix2n(r0m,i0m,r2i,i2i)
    [r1o,i1o,r3o,i3o]=radix2n(r1m,i1m,r3i,i3i)
function [r0o,i0o,r1o,i1o,r2o,i2o,r3o,i3o]=
        radix4(r0i,i0i,r1i,i1i,r2i,i2i,r3i,i3i,r1c,i1c,r2c,i2c,r3c,i3c)
    [r1m,i1m]=CM(r1i,i1i,r1c,i1c)
    [r2m,i2m]=CM(r2i,i2i,r2c,i2c)
    [r3m,i3m]=CM(r3i,i3i,r3c,i3c)
    [r0o,i0o,r1o,i1o,r2o,i2o,r3o,i3o]=radix4n(r0i,i0i,r1m,i1m,r2m,i2m,r3m,i3m)

mode MultReal
    ModeFunction RM
    ModeInfo Period=1
    OpInfo r0.Resource=Input[0]
    OpInfo i0.Resource=Input[1]
    OpInfo r1.Resource=Input[2]
    OpInfo i1.Resource=Input[3]
```

```
    OpInfo {r0,i0}.NbrBit=[18,15]
    OpInfo {r1,i1}.NbrBit=[16,15]
    OpInfo {r0,i0,r1,i1}.Cycle=0
    OpInfo r.Resource=Output[0]
    OpInfo i.Resource=Output[1]
    OpInfo {r,i}.NbrBit=[18,15]
    OpInfo {r,i}.SNR=83
    OpInfo {r,i}.Cycle=1
  mode MacReal
    ModeFunction RMAC
    ModeInfo Period=2
    OpInfo r0.Resource=Input[0]
    OpInfo i0.Resource=Input[1]
    OpInfo r1.Resource=Input[2]
    OpInfo i1.Resource=Input[3]
    OpInfo {r0,i0}.NbrBit=[18,15]
    OpInfo {r1,i1}.NbrBit=[16,15]
    OpInfo {r0,i0,r1,i1}.Cycle=0
    OpInfo r.Resource=Output[0]
    OpInfo i.Resource=Output[1]
    OpInfo {r,i}.NbrBit=[18,15]
    OpInfo {r,i}.SNR=83
    OpInfo {r,i}.Cycle=2
  mode radix4n
    ModeFunction radix4n
    ModeInfo Period=4
    OpInfo {r0i,r1i,r2i,r3i}.Resource=Input[0]
    OpInfo {i0i,i1i,i2i,i3i}.Resource=Input[1]
    OpInfo {r0i,i0i,r1i,i1i,r2i,i2i,r3i,i3i}.NbrBit=[18,15]
    OpInfo {r1i,i1i}.Cycle=0
    OpInfo {r2i,i2i}.Cycle=1
    OpInfo {r0i,i0i}.Cycle=2
    OpInfo {r3i,i3i}.Cycle=3
    OpInfo {r0o,r1o,r2o,r3o}.Resource=Output[0]
```

```
    OpInfo {r0o,r1o,r2o,r3o}.Resource=Output[0]
    OpInfo {i0o,i1o,i2o,i3o}.Resource=Output[1]
    OpInfo {r0o,i0o,r1o,i1o,r2o,i2o,r3o,i3o}.NbrBit=[18,15]
    OpInfo {r0o,i0o,r1o,i1o,r2o,i2o,r3o,i3o}.SNR=83
    OpInfo {r0o,i0o}.Cycle=6
    OpInfo {r1o,i1o}.Cycle=8
    OpInfo {r2o,i2o}.Cycle=7
    OpInfo {r3o,i3o}.Cycle=9
  mode radix2
    ModeFunction radix2
    ModeInfo Period=2
    OpInfo {r0i,r1i}.Resource=Input[0]
    OpInfo {i0i,i1i}.Resource=Input[1]
    OpInfo {rc}.Resource=Input[2]
    OpInfo {ic}.Resource=Input[3]
    OpInfo {r0i,i0i,r1i,i1i}.NbrBit=[18,15]
    OpInfo {rc,ic}.NbrBit=[16,15]
    OpInfo {r1i,i1i,rc,ic}.Cycle=0
    OpInfo {r0i,i0i}.Cycle=1
    OpInfo {r0o,r1o}.Resource=Output[0]
    OpInfo {i0o,i1o}.Resource=Output[1]
    OpInfo {r0o,i0o,r1o,i1o}.NbrBit=[18,15]
    OpInfo {r0o,i0o,r1o,i1o}.SNR=83
    OpInfo {r0o,i0o}.Cycle=4
    OpInfo {r1o,i1o}.Cycle=5
  mode radix4
    ModeFunction radix4
    ModeInfo Period=6
    OpInfo {r0i,r1i,r2i,r3i}.Resource=Input[0]
    OpInfo {i0i,i1i,i2i,i3i}.Resource=Input[1]
    OpInfo {r1c,r2c,r3c}.Resource=Input[2]
    OpInfo {i1c,i2c,i3c}.Resource=Input[3]
    OpInfo {r0i,i0i,r1i,i1i,r2i,i2i,r3i,i3i}.NbrBit=[18,15]
    OpInfo {r1c,i1c,r2c,i2c,r3c,i3c}.NbrBit=[16,15]
```

```
OpInfo {r1i,i1i,r1c,i1c}.Cycle=0
OpInfo {r2i,i2i,r2c,i2c}.Cycle=2
OpInfo {r0i,i0i}.Cycle=3
OpInfo {r3i,i3i,r3c,i3c}.Cycle=4
OpInfo {r0o,i0o,r1o,i1o,r2o,i2o,r3o,i3o}.NbrBit=[18,15]
OpInfo {r0o,i0o,r1o,i1o,r2o,i2o,r3o,i3o}.SNR=83
OpInfo {r0o,r1o,r2o,r3o}.Resource=Output[0]
OpInfo {i0o,i1o,i2o,i3o}.Resource=Output[1]
OpInfo {r0o,i0o}.Cycle=9
OpInfo {r1o,i1o}.Cycle=11
OpInfo {r2o,i2o}.Cycle=10
OpInfo {r3o,i3o}.Cycle=12
GenInfo NbrClk=4
```

VHDL-Verhaltensbeschreibung der Kontrolleinheit der makroprogrammierbaren DSP-Architektur (Ergänz. zum Kapitel 5):

VHDL-Verhaltensbeschreibung eines einfachen Kontrollers (Kapitel 0, Seite 112)

```
ENTITY FunctionControl IS
    PORT(   Clk: IN std_logic;
        Start: IN std_logic;
        FunctionNbr: IN tFunctionNbr;

        InstrData: IN tInstrData;
        InstrAddr: BUFFER tInstrAddr;

        Busy: OUT std_logic;
        FunctionEnd: OUT std_logic;
        PassEnd: OUT std_logic;
        PassConfig: OUT tPassConfig );
END FunctionControl;

ARCHITECTURE Beh OF FunctionControl IS
    SIGNAL PassReset: STD_LOGIC;
    SIGNAL Cycle: t_Cycle;
BEGIN
    p: PROCESS
        VARIABLE GeneralReg: t_GeneralReg;
        VARIABLE FunctionReg: t_FunctionReg;
        VARIABLE PassReg: t_PassReg;
    BEGIN
        PassReset<='1';Busy<='0';FunctionEnd<='0';PassEnd<='0';
        InstrAddr<=(OTHERS=>'0');
        WAIT UNTIL Clk'event and Clk='1' and Start='1';
        Busy<='1';
        GeneralReg:=InstrData(GeneralReg'length-1 DOWNTO 0);
        InstrAddr<=FunctionAddress(FunctionNbr,GeneralReg);
        WAIT UNTIL Clk'event and Clk='1';
        FunctionReg:=InstrData(FunctionReg'length-1 DOWNTO 0);
        InstrAddr<=UNSIGNED(InstrAddr)+1;
        FOR pass IN 0 TO NbrPass(FunctionReg)-1 LOOP
            WAIT UNTIL Clk'event and Clk='1';
```

```
        PassReg:=InstrData(PassReg'length-1 DOWNTO 0);
        InstrAddr<=UNSIGNED(InstrAddr)+1;
        PassReset<='0'; PassEnd<='0';
        FOR cyc IN 0 TO NbrCycle(PassReg)-1 LOOP
            Cycle<=CONV_STD_LOGIC_VECTOR(cyc,Cycle'length);
            PassConfig<=PassReset & GeneralReg & FunctionReg & (PassReg & Cycle);
            WAIT UNTIL Clk'event and Clk='1';
        END LOOP;
        PassEnd<='1'; PassReset<='1';
    END LOOP;
    Busy<='0'; FunctionEnd<='1';
    WAIT UNTIL Clk'event and Clk='1';
  END PROCESS;
END Beh;
```

Qualitätsvergleich (Kapitelübergreifende Ergänzung)

Die Art des Vergleichs

Das Ziel dieser Qualitätsanalyse ist, Designlösungen, welche mit Hilfe des HotDSPs und des DSP-Synthesetools realisiert wurden, einem Referenzdesign gegenüberzustellen und bezüglich Grösse und dynamischer Verlustleistung zu vergleichen.

Als Designaufgabe wurde der in Kapitel 3.6 verwendete Stereo-Dezimationsfilter mit vorausgehendem Offsetentfehrungsfilter gewählt. Zur Erinnerung ist seine Struktur nochmals in Abbildung 58 ersichtlich.

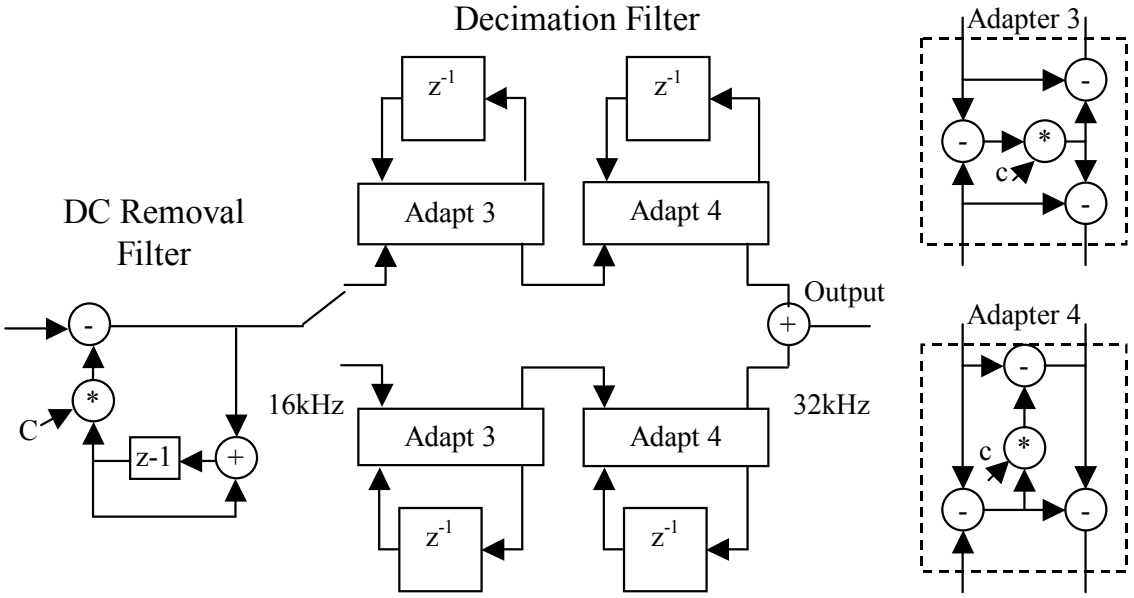


Abbildung 58: Die Struktur des Dezimationsfilter und dem vorausgehenden Offsetentfernungsfilter. Für die Stereo-Verarbeitung werden alle Filter zwei mal verwendet

Vergleichsbedingungen

Bei der Durchführung des Vergleichs wurde darauf geachtet, dass die drei Designlösungen den selben Bedingungen unterliegen, welche in den folgenden Zeilen aufgelistet sind:

- Da die in dieser Arbeit häufig verwendete Alcatel-Mietec-Technologie bezüglich Verlustleistung nicht charakterisiert ist, wurde für diesen Test die 0.25um-CMOS-Technologie von SGS-Thomson verwendet (HCMOS7).
- Die Logiksynthesen wurden mit den selben Interface-Randbedingungen und Taktraten von 12.5 MHz und 100 MHz durchgeführt.

- Der Filterblock soll in zwei Modi arbeiten. Im Monomodus soll der Datenstrom eines Kanals, im Stereomodus die Datenströme zweier Kanäle bearbeitet werden.
- Bei der Leistungsanalyse unterliegen die drei Designs den selben Interface-Randbedingungen (zeitlich sowie Datenformat). Es wurde eine Taktrate von 1 MHz verwendet. Jeweils um 16 Zyklen versetzt liefern die beiden Kanäle alle 32 Zyklen einen neuen 16-Bit-Datenwert.
- Für die Logiksynthese wird *Design Compiler* mit *Design Ware* verwendet (2000.05). Die Analyse der Verlustleistung wird auf *Gate Level* mit *Power Compiler* (2000.05) durchgeführt.
- Mittels VHDL-Simulation werden Signalaktivitätsinformationen für die folgenden drei Datenmodi aufgezeichnet:
  - Monomodus: Sinussignal mit Maximalamplitude und einer Periodizität von 128 Samples.
  - Stereomodus: Sinussignale mit Maximalamplitude und einer Periodizität von 128 Samples.
  - Monomodus: Statisch auf Null gesetztes Signal.

  Für jeden der Datenmodi wurden die Aktivitäten der Design-internen Signale mittels Simulation ermittelt und dem *Power Compiler* zur Berechnung der Verlustleistung übergeben.
- Es wurde das selbe, generelle interne Datenformat verwendet (*Sign, 1 Integer Bit, 17 Fractional Bits*). Konstanten sind 8 Bit weit.
- Die drei Designs besitzen eine ähnliche Anzahl von *Gated Clocks*.
- Auf Grund der relativ kleinen Designgrösse wurde der Modus „Top“ für die *Wire Load Models* verwendet. Auf diese Weise werden die drei Designs, unabhängig von ihrer hierarchischen Strukturierung, auf die selbe Art analysiert.

Die drei Implementierungen

Das Referenzdesign wurde mit Hilfe der einfachen Software *dp_gen*[1] durchgeführt, welche eine Strukturbeschreibung des Filters in einer speziellen Syntax liest und in eine VHDL-Beschreibung umschreibt und zugleich ein Postscript-File zur Visualisierung des Datenflusses erzeugt. Die Strukturbeschreibung enthält dabei sämtliche Informationen bezüglich den verwendeten Ressourcen, *Operation Scheduling*, Ressourcenzuweisung, *Gated Clocks*, usw. In der Tat muss *dp_gen* eher als Übersetzungswerkzeug als ein Generator betrachtet werden.

[1] *dp_gen: data path generator (IMT-internes Tool)*

Da es ist nicht einfach ist, ein Referenzdesign zu kreieren, beschränkt sich die Referenzdefinition dieses Vergleiches auf die Wahl eines „guten“ Designs und dessen möglichst genauen Beschreibung.

Das Referenzdesign wurde sehr sorgfältig entwickelt und kann als eine „optimale“ Lösung betrachtet werden. Regelmässigkeiten des Algorithmus wurden für Vereinfachungen der Hardware verwendet. Die Struktur des Referenzdesigns ist vom selben Typ wie die Architekturen, welche vom Synthesetool erzeugt werden (siehe Kapitel 4.3.2, Seite 74).

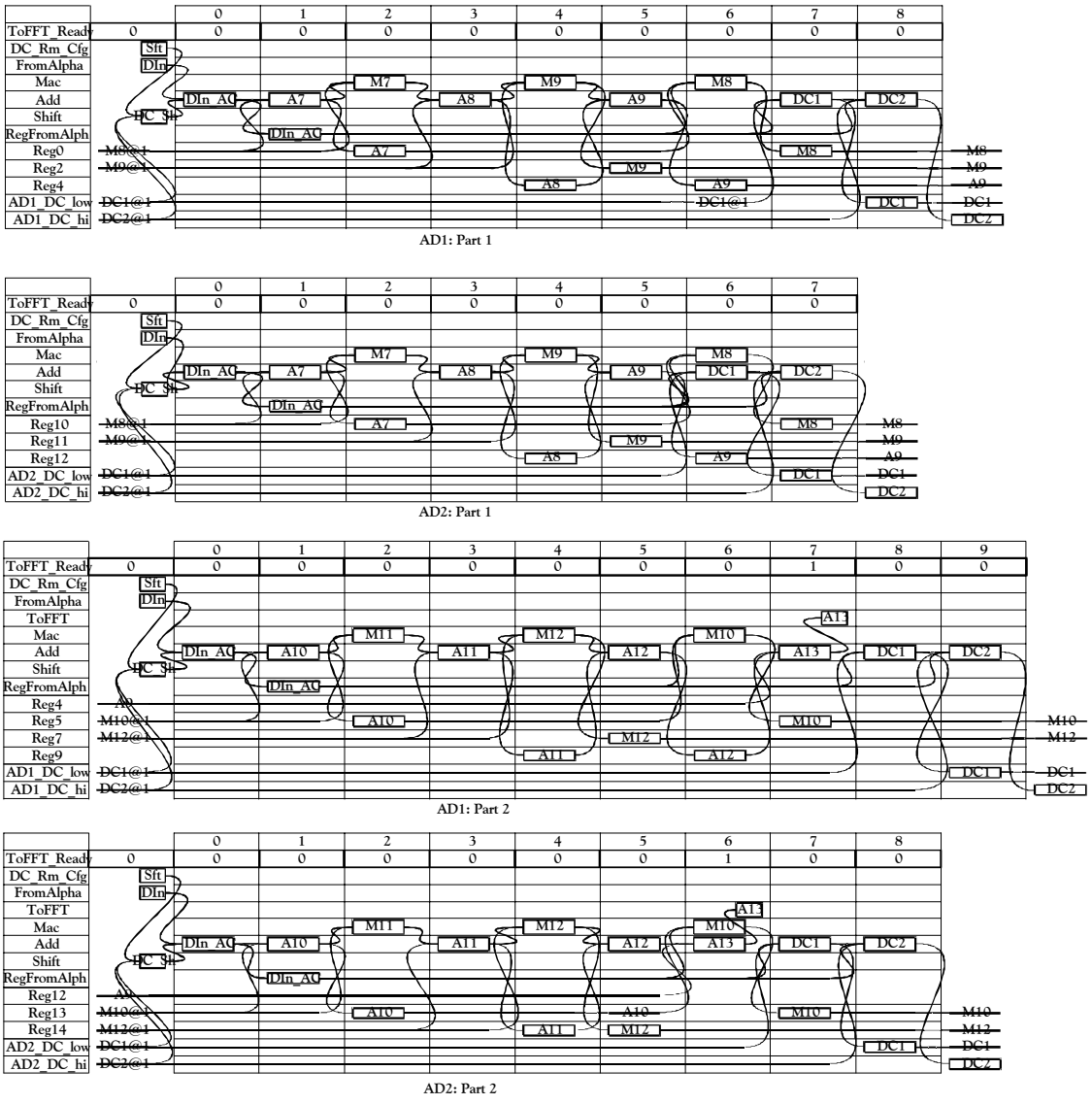


Abbildung 59: Diese von gen_dp generierte Graphik zeigt die interne Funktionsweise des Referenzdesigns. Die vier ersichtlichen Funktionen verarbeiten der Reihe nach folgende Daten: Ungerade Samples Kanal 1, ungerade Samples Kanal 2, gerade Samples Kanal 1, gerade Samples Kanal 2.

Der Datenfluss, das *Operation Scheduling* und die Ressourcenzuweisung des Referenzdesigns ist in Abbildung 59 ersichtlich. Es wurden 4 *Gated*

Clocks verwendet, welche wie folgt auf die einzelnen Ressourceneinheiten verteilt wurden:

- Clock „Clk1“: RegFromAlpha, Reg0, Reg5, AD1_DC_low, AD1_DC_hi
- Clock „Clk2“: Reg2, Reg4, Reg7, Reg9
- Clock „ResClk“: Mac, Add
- Clock „AD2Clk“: Reg10, Reg11, Reg12, Reg13, Reg14, AD2_DC_low, AD2_DC_hi

Sämtliche nur für den Stereomodus benötigten Ressourceneinheiten werden von einem für diesen Modus reservierten *Gated Clock* („AD2Clk) kontrolliert. Im Monomodus können so diese Einheiten deaktiviert werden, was zu einer deutlichen Stromreduktion in diesem Modus führt.

Das in Kapitel 3.6 präsentierte Anwendungsbeispiel des HotDSP-Typs 2 diente als zweite Implementierung für diesen Test.

Die dritte Implementierung wurde mit dem DSP-Synthesetool durchgeführt.

Resultate

$f_{Synthese}$=12.5 MHz			**von Hand / *dp_gen***	**HotDSP**		**DSP-Synthesetool**	
Grösse [GE[1]]	**Komb. Logik**		3155	3261	+3.4%	3183	+0.9%
	Register		1704	1765	+3.6%	1531	-10.2%
	Total		**4859**	**5026**	**+3.4%**	**4714**	**-3.0%**
Verlustleistung [µW]	**Mono Sinus**	Zellen	18.18	20.97	+15.3%	20.63	+13.5%
		Netz	13.68	18.28	+33.6%	15.36	+12.3%
		Total	**31.86**	**39.25**	**+23.2%**	**35.99**	**+13.0%**
	Mono 0-Eingang	Zellen	11.04	11.64	+5.4%	11.14	+0.9%
		Netz	6.79	9.16	+34.9%	8.54	+25.8%
		Total	**17.83**	**20.80**	**+16.7%**	**19.68**	**+10.4%**
	Stereo Sinus	Zellen	33.94	38.38	+13.1%	36.68	+8.1%
		Netz	25.90	34.98	+35.1%	28.09	+8.5%
		Total	**59.84**	**73.36**	**+22.6%**	**64.77**	**+8.2%**

Tabelle 12: Vergleich der drei Designs für eine Taktfrequenz von 12.5 MHz

Tabelle 12 und Tabelle 13 zeigen die Grössen und Verlustleistungen der drei Designlösungen, welche für Taktfrequenzen von 12.5 MHz und 100 MHz

[1] Als Referenzgrösse dient die ND2A-Zelle

synthetisiert worden sind. Die Grössen stellen sich jeweils aus der kombinatorischen Logik und aus den Register zusammen. Da die *Wire Load Models* keine Schätzwerte für die Platzbeanspruchung der Verbindungsnetze besitzen, ist in der Designgrösse das Routing nicht berücksichtigt.

Die Verlustleistung hat ebenfalls zwei Komponenten; einerseits der Zellen-interne Verbrauch und andererseits die Verlustleistung, welche von den Aktivitäten der Verbindungsnetze herstammt. Wohl bemerkt, diese letzte Komponente ist eine grobe, *Wire Load Model* – basierte Schätzung.

$f_{Synthese}$=100MHz			**von Hand / *dp_gen***	**HotDSP**		**DSP-Synthesetool**	
Grösse [GE]	**Komb. Logik**		4896.5	4047.75	-17.3%	4223.75	-13.7%
	Register		1767	1808	+2.3%	1529.25	-13.5%
	Total		**6663.5**	**5855.75**	**-12.1%**	**5753**	**-13.7%**
Verlustleistung [µW]	**Mono Sinus**	Zellen	22.46	25.15	+12.0%	21.39	-4.8%
		Netz	22.58	29.04	+28.6%	23.96	+6.1%
		Total	**45.05**	**54.19**	**+20.3%**	**45.35**	**+0.7%**
	Mono 0-Eingang	Zellen	10.85	12.14	+11.9%	10.92	+0.7%
		Netz	6.97	9.05	+29.9%	8.60	23.5%
		Total	**17.82**	**21.19**	**+18.9%**	**19.52**	**+9.6%**
	Stereo Sinus	Zellen	40.74	47.42	+16.4%	38.49	-5.5%
		Netz	41.89	53.64	+28.1%	43.33	+3.4%
		Total	**82.63**	**101.06**	**+22.3%**	**81.82**	**-1.0%**

Tabelle 13 Vergleich der drei Designs für eine Taktfrequenz von 100 MHz

Tabelle 14 gibt einige Architektur-bezogene Informationen, welche bei der Interpretation der Resultate eine Hilfe sein können.

	von Hand / *dp_gen*	**HotDSP**	**DSP-Synthesetool**
# 16/19-Bit – Datenregister	**16+2½+2 (siehe [1])**	**14+2½+1 (siehe [2])**	**13+1½+2 (siehe [3])**
# *Gated Clocks*	**4**	**6**	**4**

Tabelle 14: Architekturbezogene Daten der drei Lösungen

[1] Datenregister + Eingangsregister des MAC + Eingangsregister des Adders
[2] Datenregister + Eingangsregister der arithmetischen Einheit + Accu1
[3] Datenregister + Eingangsregister des MUL + Eingangsregister des Adders

Diskussion

Auffällig in Tabelle 12 sind die ungefähr selben Grössen der drei Designlösungen, wenn sie für eine tiefe Taktfrequenz synthetisiert werden (12.5 MHz). Selbst die Lösung mit Hilfe des frei programmierbaren HotDSPs ist lediglich um 3 bis 4 Prozent grösser als das Referenzdesign. Das *high-level* DSP-Synthesetool generiert sogar eine Lösung, welche kleiner als das Referenzdesign ist, dies auf Grund der optimalen Weise, wie die Register verwendet werden (siehe Tabelle 14).

Diese sehr positiven Resultate werden durch die Resultate der Verlustleistungen etwas relativiert. Je nach Arbeitsmodus und der verwendeten Stimuli ist die Verlustleistung des HotDSP etwa 20 und diejenige der DSP-Synthesetool-Lösung etwa 10 Prozent höher als beim Referenzdesign. Speziell beim HotDSP stammt dabei ein grosser Teil der zusätzlichen Verlustleistung von den Aktivitäten der Verbindungsnetze. Was aber bei allen Designlösungen funktioniert, ist die effiziente Stromsparfunktion des Monomodus, bei der bezüglich einer Funktionsweise im Stereomodus zwischen 40 und 50 Prozent der Verlustleistung eingespart wird.

Die Resultate ändern rapide, wenn die drei Designs für eine erhöhte Taktfrequenz von 100 MHz synthetisiert werden. Die Verwendung einer von speziell für Geschwindigkeit optimierten Logik lässt die Grösse der drei Designs anwachsen. Die von Hand geschriebene Lösung reagiert dabei besonders empfindlich auf Timing-Randbedingungen, was bewirkt, dass sie über 12 Prozent grösser ist als die beiden anderen Lösungen. Diese besitzen demnach bessere Grundstrukturen für höhere Geschwindigkeiten.

Bezüglich der Verlustleistung ist die Lösung des DSP-Synthesetools etwa äquivalent zu dem Referenzdesign. Im Gegensatz zum Letztgenannten beträgt jedoch der Rückstand der HotDSP-Lösung immer noch etwa 20 Prozent.

Anhang II (Glossar und Verzeichnisse)

Glossar

Abhängigkeits-liste	*Sensitivity list*	Definiert die Signalabhängigkeiten eines VHDL-Prozesses
Abtastung, Abtastwert	*Sampling, Sample*	Signalwert zu einem exakten Zeitpunkt
Adressierungs-bereich	*Address domain*	Bereich, welcher mit einem Adressenformat selektierter werden kann
Adresskontroller	*Address controller*	Einheit zum Berechnen der Parameteradressen
Adressstack	*Address stack*	Stapel zum Zwischenspeichern von Rücksprungadressen
AD-Wandler	*AD-Converter*	Analog/Digital-Wandler
Akkumulator-register	*Accumulator register*	Meist das Hauptregister einer ALU
Aliasingeffekt	*Aliasing effect*	Effekt, der durch Unterabtastung eines Signals entsteht
Allzweck-DSP	*General purpose DSP*	Frei programmierbarer DSP mit einer allgemein verwendbaren Architektur
Assembler-Quellcode	*Assembler source code*	Programmtext eines Maschinenprogramms
Backend-Designfluss	*Backend design flow*	Hardware-bezogener Entwicklungsteil
Blockgleitkom-maarithmetik	*Block floating point arithmetic (BFP)*	Zuweisung eines Gleitkomma-Exponenten nicht an eine individuelle Zahl, sondern an eine Zahlengruppe
	Casting	Übergang von einem ersten Datenformat in ein zweites
CIC-Hörgeräten	*CIC hearing aid devices*	Kleinster Typ von Hörgeräten, welcher vollständig im Ohr getragen werden kann
Clockfrequenz	*Clock frequency*	Entspricht meist der Taktrate eines Prozessors
CMOS-Gatter	*CMOS gate*	Logikgatter in der CMOS-Technologie
Co-Design	*Co-design*	Design gebildet aus mehreren Bereichen
Compilieren	*To compile*	Verarbeiten einer symbolischen Beschreibung in eine hardwarenahe Beschreibung
Datencasting	*Data casting*	Anpassung zweier Datenformaten
Datenflussgraph	*Data flow graph*	Graphische Darstellung der Arbeitsweise eines Algorithmus
Daten-interpolation	*Data interpolation*	Einfügen von zusätzlichen Datenpunkten, korrekt an die Nachbarpunkte angepasst
Datenpfad	*Datapath*	Datenverarbeitungsteil eines Prozessors
Datentransfer-	*Data transfer*	Anzahl übertragener Datenworte pro Sekunde

rate, Daten-übertragungsrate	*rate*	
DA-Wandler	*DA-Converter*	Digital/Analog-Wandler
Debuggen	*To debug*	Befreien eines Programms von Fehlern
Designfluss	*Design flow*	Definiert die Folge der Designmanipulationen
Designrand-bedingungen	*Design constraints*	Normalerweise Bedingungen, welche ein Design zu erfüllen hat, welche nicht direkt an einen Algorithmus gebunden sind
Fest verdrahtet	*Hardwired*	Ein fest verdrahteter DSP besitzt eine fixierte, anwendungsspezifische Kontroll- und Datenverarbeitungseinheit, kann also nicht erneut programmiert werden
FFT-Prozessor	*FFT processor*	Spezieller Prozessor zur Durchführung von FFTs
FFT-Spektral-analyzer	*FFT spectrum analyzer*	FFT-basierter Spektralanalyzer
FFT-Transfor-mation	*FFT transformation*	*Fast Fourier Transformation*: Eine rechenoptimierte Zeit-Frequenz-Transformation
FIFO-Kontroller	*FIFO controller*	Kontroller eines Zirkularringspeichers zur Emulation eines FIFOs
Filter-Designfluss	*Filter design flow*	Designfluss zur Entwicklung und Generierung einer Filteranwendung
Filter-Synthese-tool	*Filter synthesis tool*	Software zur automatischen Generierung einer Filteranwendung
	Floorplanning	Geometrische Organisation der Chipfläche
Frequenzanalyse	*Frequency analysis*	Bestimmung der Frequenzen eines Signals
	Frontend	Logik-bezogener Entwicklungsteil
Ganzzahl-repräsentation	*Integer representation*	Auch Fixkommarepräsentation genannt: Die Zahlen besitzen einen konstanten Exponenten, welcher nicht registriert werden muss, da er eben fix ist
Gatter	*Gate, kGate*	*Gate Equivalent*: Entspricht der Grösse eines NAND- oder NOR-Gatters (kGate=1000 Gates)
Gleitkomma-arithmetik	*Floating point arithmetic*	Arithmetik, welche auf Gleitkommazahlen angewandt wird
Hardwarebe-schreibungs-sprache	*Hardware description language*	Programmiersprache mit einer speziellen Syntax, welche die Beschreibung eines architektonischen Aufbau eines Designs und eines physikalischen Verhaltens ermöglicht
HDL-Beschrei-bungssprachen	*Hardware Description Language*	Sprache mit besonderen Eigenschaften zur effizienten Modellisierung von elektronischen Schaltungen und Bauteile
Interpolations-filter	*Interpolation filter*	Filter, welche in eine Abtastsequenz zusätzliche Abtastwerte einfügt, welche durch Interpolation bestimmt werden
	IO-Pad	Ein- und Ausgangsanschlüsse eines ASICs / FPGAs (Input/Output Pad)
IP-Block	*Intellectual property block*	Vollständiges Design eines Anwendungsblockes, welches unberührt in ein Gesamtdesign eingefügt

		werden kann (IP block)
Kombinatorisch	*Combinatorial*	In einer kombinatorischen Logik folgen die Ausgangssignale in Funktion der Eingangssignale. Sie besitzt keine sequentiellen Elemente
Kontroller	*Controller*	Meist kleiner Prozessor für Kontroll- und Steuerzwecke
Koprozessor	*Coprocessor*	Einem Hauptprozessor beistehenden, meist für eine Aufgabe spezialisierter Zusatzprozessor
Low-Power	*Low power*	leistungssparend, impliziert stromsparend
Low-Power-Anwendung	*Low power application*	Anwendungen, welche mit einer geringen Leistung arbeiten müssen (meist Batterie-betriebene Geräte)
Low-Power-Techniken	*Low power techniques*	Techniken, um den Stromverbrauch zu reduzieren. In einem CMOS-Chip wird dies durch Reduzierung der Chip-Aktivität erzielt
Makro	*Macro*	Zusammengefasste Instruktionssequenz
Makrocode	*Macro code*	Programmcode, dessen Instruktionswörter nicht die Verarbeitung einer Operation, sondern eines Makros kontrollieren
Nachkommazahl	*Fractional part*	Teil der Zahl, welche nach dem Komma steht
NOP-Instruktion	*No Operation instruction*	Bewirkt einen leeren Arbeitszyklus in einem Prozessor
parametrisierbar	*parametrisable*	Im Zusammenhang eines Prozessors: Prozessorarchitektur, welche gewisse Konfigurationen zulassen
	P&R , Place and route, Place&Route	Platzieren und verbinden von elektronischen Elementen (im ASIC-Design: Standardzellen)
Puffern	*To buffer*	Verstärken eines Signals mittels eines Nicht-Inverters. Auch für beim Gebrauch eines Registers verwendet
Quantisierung, quantisieren	*Quantization*	Annäherung einer Gleitkommazahl mittels einer Festkommazahl
Quellcode	*Source code*	Text-basierter Programmtext
Resetsignal	*Reset signal*	Signal zur Zurücksetzung oder Initialisierung eines Systems
Ressourcenallo-kation, Ressour-cenzuordnung	*resource allocation*	Zuweisung einer bestimmten Ressourceneinheit einer logischen Operationen
Ressourcen-einheit	*resource unit*	Im Zusammenhang einer Datenverarbeitungseinheit: Register, arithmetische Einheiten, Ein- und Ausgangsports, Multiplexer
	Router	Tool, welche die elektronischen Elemente einer Schaltung miteinander verbindet
	Scheduling	Das Zuweisen eines Arbeitszeitpunktes an eine Operation wird Time-Scheduling genannt
Schnittstelle	*Interface*	An- oder Zusammenschlussstelle
Sleep-Modus	*Sleep mode*	Ruhe- und Wartezustand eines Prozessors
SNR-Rand-bedingungen	*SNR constraint*	Definiert das maximal erlaubte Quantisierungsrauschen

Spektralanalyzer	*Spectrum analyzer*	Gerät zur Analyse des Frequenzspektrums in Echtzeit
Stack	*Stack*	siehe Adressstack
Standardzellen-bibliotheken	*Standard cell library*	Sammlung logischer Grundschaltungen, welche zum Bau komplexerer Schaltungen im ASIC-Design verwendet werden. Enthält zwischen 100 und 500 Zellen
Stand-By-Modus	*Stand by mode*	Ruhezustand einer Schaltung
Synthetisieren	*To synthesize*	Zusammenführen: In der logischen Synthese wird eine Schaltungsbeschreibung mittels Standardzellen nachgebaut. In der Filtersynthese wird eine Filterspezifikation mit Hardwareressourcen aufgebaut
Systemclock	*System clock*	Clock zur Synchronisation des Systems. Siehe auch Clock
Tabu-Search	*Tabu search*	Suchalgorithmus. Versucht mittels einer Tabu-Tabelle lokale Minima zu überwinden
Takt	*Clock*	Signal zur Aktivierung der Register
Takten	*To clock*	Regelmässige Aktivierung eines Systems oder eines Registers
Taktrate	*Clock frequency*	Arbeitstakt einer digitalen Schaltung
Überabtastung	*Oversampling*	
Verilog-Module	*Verilog module*	Verilog-Blöcke werden Module genannt
VHDL-Biblio-theken	*VHDL library*	Bibliothek von VHDL-Packeten, -Architekturen, -Konfigurationen und –Schnittstellen (*Entity*)
VHDL-Paket	*VHDL package*	VHDL-Funktionen, -Prozeduren, -Deklarationen und –Konstanten können in einem Paket abgelegt werden. Eine Bibliothek wird im VHDL-Jargon ein Paket genannt
VHDL-Prozess	*VHDL process*	Eine autonome Aufgabe wird in VHDL als Prozess bezeichnet
VHDL-Test-bench	*VHDL test bench*	VHDL-Block zum Steuern und Kontrollieren eines VHDL-Designs zu Testzwecken
WDF-Filter	*Wave digital filter*	Spezielle Filterstrukturen, welche auf digitaler Weise analoge, passive Filter nachzubauen versuchen
WOLA-Algorithmus	*Weighted overlap add algorithm*	Algorithmen, welche sich auf eine WOLA-Transformation basieren
WOLA-Filterbank	*WOLA filterbank*	Filterbank, welche mittels der WOLA-Transformation realisiert ist
WOLA-Transformation	*WOLA transformation*	Wie die FFT ist die WOLA-Transformation eine Zeit- Frequenz-Wandlung eines Signals
Zeitrückführung	*Delay or timing (back-) annotation*	
Zirkular-speicher	*Circular memory*	Speicher, bei dem die Elemente rotierend selektiert wird. Wird zur Emulation eines FIFOs verwendet

Abkürzungen

ALU	*Arithmetic and Logic Unit*: Einheit zur logischen und arithmetischen Datenverarbeitung.
ASIC	*Application Specific Integrated Circuit*
ASIP	*Application Specific Instruction set Processor*
ASSP	*Application Specific Standard Product*
BFP	*Block Floating Point*
CIC	*Complete In Channel*
DFL	*Data Flow Language*
DRC	*Design Rule Check*
DSP	*Digital Signal Processor*
DspC_GUI	*Dsp Compiler's Graphical User Interface*
ERC	*Electrical Rule Check*
FIFO	*First In, Last Out*
FFT	*Fast Fourier Transformation*
GE	*Gate Equivalents*
HDL	*Hardware Description Language*
I	Abk. für Strom
IMT	*Institut de Microtechnique de l'Université de Neuchâtel*
IOP	*Input / Output Processing Unit*
LUT	*Look-up Table*
MAC	*Multiplication / Accumulation*
MAOPS	*Million Arithmetic Operations Per Second*
MIMD	*Multiple instruction stream, multiple data stream*
MIPS	*Million Instruction Per Second*
MISD	*Multiple instruction stream, single data stream*
MOPS	$=OPS*10^6$
MVL9	*Multi Valued Logic, 9 values*
NOP	*No Operation*
OPS	*Operation Per Second*
P	*Power* = Leistung
PLI	*Programming Language Interface*
RISC	*Reduced Instruction Set Computer*
RTL	*Register Transfer Level*
SAI	*Serial Audio Interface*
SIMD	*Single Instruction flow, Multiple data flow*
SNR	*Signal to Noise Ratio*: Vergleich zwischen der Signalenergie und der Energie des Lärms .
Tcl/Tk	*Tool command language / tool kit*: Tcl: Skriptsprache wie Pearl. Tk: Widget-Bibliotheken-Erweiterung von Tcl (graphische Erweiterung).
Vdd	Speisespannung
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High-Speed Integrated Circuit*
VITAL	*VHDL Initiative Toward ASIC Libraries*
VLIW	*Very Large Instruction Word*

WDF	*Wave Digital Filter*
WOLA	*Weighted OverLap Add,* FFT basierter Algorithmus

Abbildungsverzeichnis

Verschiedene Implementierungsmethoden, tendenziell geordnet 8
Bei der Logiksynthese konstruiert ein Synthesetool mit Hilfe von Standardzellen 12
Die Haupttypen paralleler Prozessorsystemen 28
Die Einordnung der Co-Designmethode 36
Der Designfluss der Designmethode 39
Die verschiedenen Anwendungsbereiche der Designmethode 40
Der strukturelle Aufbau der 3 Architekturtypen des HotDSPs 42
Die Kontrolleinheit der HotDSPs vom Typ 1 44
Die Datenverarbeitungseinheit der HotDSPs vom Typ 1 45
Sequenz der Instruktionsverarbeitung 49
Der HotDSP- Debugger als Erweiterung des ModelSim VHDL-Simulators 54
Die Struktur des Interpolationsfilter 56
Die Struktur des Dezimationsfilter und dem vorausgehenden Offsetentfernungsfilter 58
Filter-Designfluss 66
Gewünschte Interface- und Interaktionsmöglichkeiten des DSP-Synthesetool 70
Der für Low-Power-Anwendungen geeignete Designfluss für die Filtergenerierung 73
Die Ressourceneinheiten wählen via Multiplexer jeweils jeden Zyklus die 74
Der Implementierungsaufbau des DSP-Synthesetools 76
Die zwei Schritte der Festlegung der Datenwortbreiten 82
Drei verschiede Datentypen müssen vom selben Bus unterstützt werden 82
Die Operationen zur Ermittlung der Gesamtdatenwortbreiten 83
Die Konvergenzschwierigkeiten von Tabu-Search bei der Quantisierung 86
Definition der Kostenfunktion für die Quantisierung 87
Der Programmfluss für die Bestimmung der Gesamtdatenwortbreiten 89
DspC_GUI: Das graphische Frontend des DSP-Synthesetools 92
Graphikanimationen visualisieren die verschiedenen iterativen Optimierungen 93
Zwei praktische Tcl/Tk-Applikationen 94
Schematischer 3-teiliger Datenfluss eines Zeit-Frequenz-Zeit-Spektralanalyzers 105
Operationen eines FFT-Spektralanalzyer 106
Die Grundstruktur des DSPs 110
Zeitdiagramm des Kontrollers 113
Aufteilung des Kontrollers 114
Der Datenpfad mit Kontrollteil und Datenverarbeitungsteil 116
Ein Adressgenerator teilt das Konfigurationswort in Konfigurationsteil und 117
Endgültige Struktur der DSP-Architektur 118
Die ALU des WOLA-Prozessors 120
Die Struktur des WOLA-Prozessors 121
Die Bereichsaufteilung der beiden FIFOs 122
Die verschiedenen Typen von Hörgeräten 130
Prinzipieller Aufbau eines digitalen Hörgerätes 131
Flussdiagramm der Audiodaten innerhalb des Digitalteils 133
Das DSP-Koprozessor-Konzept 134
Vereinfachtes Blockdiagramm einer WOLA-Filterbank 136
Filter zur Entfernung eines Offsets, zur Datendezimation und zur Dateninterpolation 138
Charakteristik des Filters für die Offsetentfernung und des Halbband-Tiefpassfilter 139

Blockdiagramm des Datenflusses und architektonische Strukturierung des Digitalteils 144
Die Input/Output- Verarbeitungseinheit (IOP) ... 146
Prototyp-Chipset des Hörgerätesystems ... 147
Eine Testintegration des Koprozessors in einer 5-Metal-0.35um-CMOS-Technologie 147
Eine Testintegration des Koprozessors zusammen mit dem General Purpose DSP ... 148
Eine nicht am IMT durchgeführte Integration des Kopro. zusammen mit dem DSP ... 148
Backend-Designfluss ... 149
Backend-Designfluss: Da bei DSM-Technologien die Leiterkapazitäten nur 153
Charakteristik der Filter ... 154
Kanaltrennung der WOLA-Filterbank in Odd- und Even-Stacking-Modus ... 155
Matlab- und VHDL-Simulation des Koprozessors ... 156
Der Toccata Hybrid ... 159
Die Struktur des Dezimationsfilter und dem vorausgehenden Offsetentfernungsfilter ... R
Diese von gen_dp ... T

Tabellenverzeichnis

Definition der in den folgenden Betrachtungen verwendeten Symbolen ... 18
Der α-Faktor in Funktion verschiedener Transistorengatelängen ... 20
Grösse in GE des ersten Beispiels ... 57
Grösse in GE des zweiten Beispiels ... 60
Vergleich einer DSP-Synthesetool-Lösung mit handgeschriebenen VHDL-Lösungen .. 96
FFT-Benchmark-Werte einiger DSPs ... 102
Die Organisation des Makrocodes ... 112
Grösse in GE des WOLA-Prozessors ... 124
Berechnungszyklen und –Zeit für verschiedene WOLA-Konfigurationen ... 124
Die Grössen des Verarbeitungsteils für digitale Hörgeräte ... 157
Stromverbrauch der Digitaleinheit bei verschiedenen Operationen ... 158
Vergleich der drei Designs für eine Taktfrequenz von 12.5 MHz ... U
Vergleich der drei Designs für eine Taktfrequenz von 100 MHz ... V
Architekturbezogene Daten der drei Lösungen ... V

Listingverzeichnis

Die Verilog-Beschreibung im Vergleich zu der identischen VHDL-Beschreibung ... 16
Die Bitfelder der einzelnen Instruktionen ... 47
Ausschnitt eines Assembler-Quellcodes ... 52
Ausschnitt des VHDL-Parameterpaketes ... 53
Der Assembler-Quellcode des Stereo-Interpolationsfilters ... 57
Der Assembler-Quellcode des Dezimationsfilters ... 59
Beschreibung eines einfachen Filters ... 77
Ein Ausschnitt aus dem Ressourcen- Definitionsfile ... 79
Folgende Seiten: VHDL-Definitionspaket für den HotDSP-Typ 1 ... D
Nächste Seiten: Pseudocode einiger Rutienen zur Quantisierung ... J
Die komplette Beschreibung der FFT-Prozessor-Verarbeitungseinheit ... L

h

Curriculum vitae

Name	Drollinger
Vorname(n)	Andreas Peter
Geburtsdatum	04. 03. 69
Zivilstand	verheiratet
Heimatort	Muri b. Bern, Schweiz

Ausbildung

1985	- 1989	Ausbildung als Uhrmacher-Rhabilleur, Uhrmacherschule Solothurn
1989	- 1992	Ingenieur HTL in Mikrotechnik, Ingenieurschule Biel
1992	- 1996	Studium in “électronique physique”, Universität Neuchâtel

Berufliche Aktivitäten

1989	- 1996	Studie, Konzeption und Realisation verschiedener Spezialmessgeräte, *ETA SA*
1991	- 1992	Dozent für Mechanik bei *SFB* (*Schweizerische Fachschule für Betriebstechnik TS*)
	1992	Realisation des ersten synthetisierbaren CoolRisc, Diplomarbeit, CSEM
1992	- 2001	*Institute de Microtechnique* in Neuchâtel, Abteilung Signalverarbeitung und Elektronik
seit	2001	Leiter der Prozessorengruppe, dspfactory SA, Marin

Sprachen

Deutsch	Muttersprache
Französisch	Sehr gute Kenntnisse
Englisch	Gute Kenntnisse

Verzeichnis der Veröffentlichungen

Veröffentlichungen zur Hardware/Software – Co-Design – Methodik:

- A. Drollinger, A. Heubi, P. Balsiger, F. Pellandini, "HW/SW Co-design Methodology for Ultra-Power-Applications and Fastest Development Time", Proc. of the 8th International Symposium on Integrated Circuits, Devices & Systems, ISIC-99, pp. 455-458, Grand Hyatt, Singapore, September 8-10, 1999.

Veröffentlichungen zum high-level DSP-Synthesetool:

- A. Drollinger, A. Heubi, P. Balsiger, F. Pellandini, „Ein DSP-Synthesetool für schnelle Low-Power-Implementierungen von DSP-Algorithmen“, DSP Deutschland 2000, München, 10. – 12. Oktober 2000
- A. Drollinger, A. Heubi, P. Balsiger, F. Pellandini, „ASIC DSP Compiler for Optimized Synthesis“, ICSPAT 2000, International Conference on Signal Processing Applications & Technology, Dallas, Texas, USA, October 16-19, 2000.

Veröffentlichungen zur makroprogrammierbaren DSP-Architektur:

- A. Drollinger, C. Wälchli, D. Sun, L. Grisoni, C. Calame, A. Heubi, P. Balsiger, F. Pellandini, R. Brennan, T. Schneider, "An Ultra Low Power WOLA Filterbank Implementation in Deep Submicron Technology", Proc. of COST 254, International Workshop on Intelligent Communication Technologies and Applications, with Emphasis on Mobile Communication, Neuchâtel, CH, May, 5-7, 1999.
- A. Drollinger, A. Heubi, P. Balsiger, F. Pellandini, "Low Voltage FFT Co-Processor for Ultra Low Power, Real-Time Applications", Proc. of the DSP Deutschland 99, Forum der Technik, pp. 357-362, Muenchen, D, September 21-23, 1999.
- A. Drollinger, R. Brennan, T. Schneider, Ch. Calame, L. Grisoni, D. Sun, Ch. Waelchli, A. Heubi, P. Balsiger, F. Pellandini, "An Ultra Low Power WOLA Filterbank Implementation in Deep Submicron Technology", Proc. of the International Conference on Signal Processing Applications and Technology 99, ICSPAT 99, Orlando, Florida, USA, November 1-4, 1999.
- A. Drollinger, A. Heubi, P. Balsiger, F. Pellandini, „Macro-Programmable DSP Architecture for Parallel / Pipelined Data Path Units, Targeted for FFT Based Algorithms“, ICSPAT 2000, International Conference on Signal Processing Applications & Technology, Dallas, Texas, USA, October 16-19, 2000.